Neural Networks exam 2019/20

No. 1 Topic
a. Local optimization in non-convex cases (reason for non-convexity)
b. RMSP optimizer
c. Dropout
d. ResNet
e. Gradient ascent

No. 2 Topic
a. Weight update strategies
b. ReLU and dying ReLU problem
c. LSTM cell
d. Convolution as a mathematical operation in continuous and discrete cases

No. 3 Topic
a. Newton optimization method
b. Ensembling, bagging
c. Comparison of loss functions
d. Machine learning vs traditional programming
e. Inception

No. 4 Topic
a. McCulloch-Pitts model
b. Parameters of conv - filter, stride, padding, etc
c. Linear classifier, margin of the classifier
d. Data augmentation
e. YOLO

No. 5 Topic
a. Statistical learning theory
b. Various activation functions and their properties
c. Autoencoders
d. Graph unrolling and parameter sharing in recurrent NN
e. MobileNet

No. 6 Topic
a. Machine learning problem definition
b. Newton optimizer
c. Effects and relationship of model capacity and complexity - overfitting, underfitting
d. t-distributed Stochastic Neighbor Embedding
e. ShuffleNet

No. 7 Topic

a. Credit approval problem
b. Objective functions in neural networks
c. Nesterov momentum optimizer
d. Decomposition of convolutional kernels
e. Alexnet + ILSVRC

No. 8 Topic

a. Delta learning rule
b. Batch normalization
c. Transposed conv., atrous conv.
d. Object classification, localization VS object detection, semantic segmentation VS instance segmentation
e. ResNext

No. 9 Topic

a. ADAM optimizer
b. The softmax function
c. R-CNN architectures: R-CNN, Fast R-CNN, Faster R-CNN
d. Supervised VS unsupervised learning
e. EfficientNet

No. 10 Topic

a. Optimization problem of objective functions of NN
b. AdaGrad optimizer
c. Input vector normalization
d. DeconvNet, U-Net
e. Neural style transfer

No. 11 Topic

a. Multilayer perceptron
b. Early stopping
c. Gradient descent (multidimensional cases as well)
d. Weight regularization (L1, L2)
e. Pooling

No. 12 Topic

a. Perceptron convergence theorem (no proof)
b. Momentum optimizer
c. Properties of CNN: sparsity, parameter sharing, equivariance, invariance to shifting
d. RNN examples: predicting the next letter, image captioning
e. Adversarial attacks

No. 13 Topic

a. Elementary set separation by a single neuron
b. Local response normalization
c. Unpooling
d. Representations: Blum and Li theorem, construction

No. 14 Topic

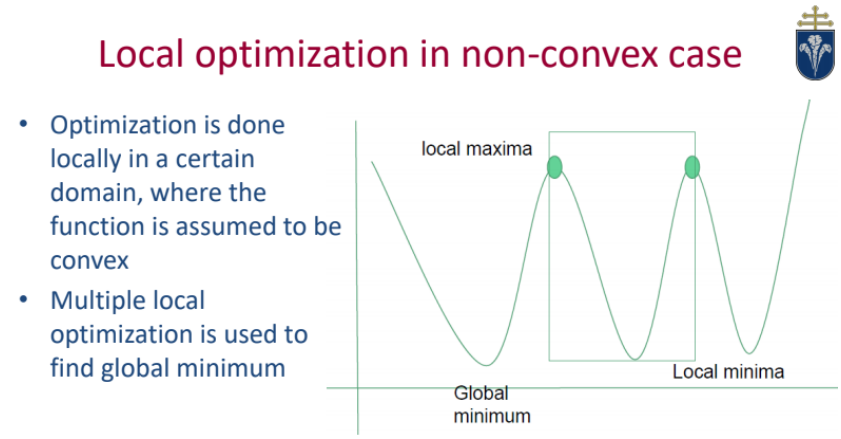
a. Principal component analysis (PCA)
b. Back-propagation through time
c. Stochastic gradient descent optimizer
d. Object detection problem explained
e. Effects of filter size on convolution

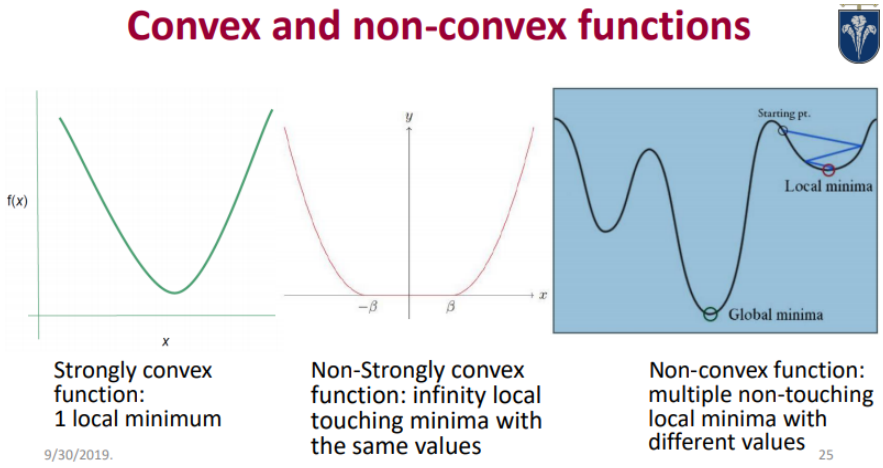
No. 15 Topic

a. Rosenblatt perceptron training algorithm
b. Back-propagation
c. Curse of dimensionality
d. SqueezeNet
e. Back-propagation and gradient-based optimizers

No. 1 Topic

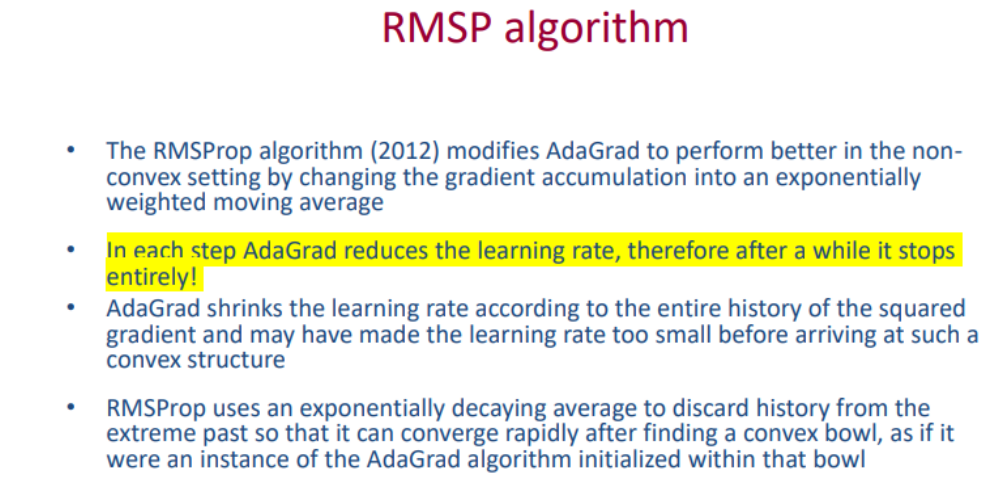
a. Local optimization in non-convex cases (reason for non-convexity)





---

b. RMSP optimizer



RMSP algorithm

Algorithm The RMSProp algorithm

The closer parts of the history are counted more strongly.

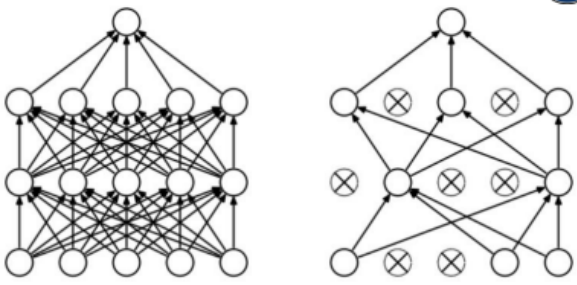
Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

---

c. Dropout

Dropout

- Idea of dropout method:
  - Use mini-batch training approach
  - For each minibatch, a random set of neurons from one or multiple hidden layer(s) (called **droppout layers**) is temporally deactivated
  - Selection and deactivation probability is *p*
  - In testing phase, use all the neurons, but multiply all the outputs with p, to account for the missing activation during training
- Requires more training steps, but each is simpler, due to reduced number of neurons
- No computational penalty in testing phase
- Use it for fully connected layers



(a) Standard Neural Net (b) After applying dropout.

Reduces overfitting, because the network is forced to learn the functionality in different configurations using different neural paths.

Reasoning behind dropout

- Dropout can be considered as averaging of multiple thinned networks ("ensemble")
- Dropout avoids training separate models
  - Would be very expensive
- Avoids computatinal penalty in the test phase
- But still gets benefits of ensemble methods

Intuitive explanation

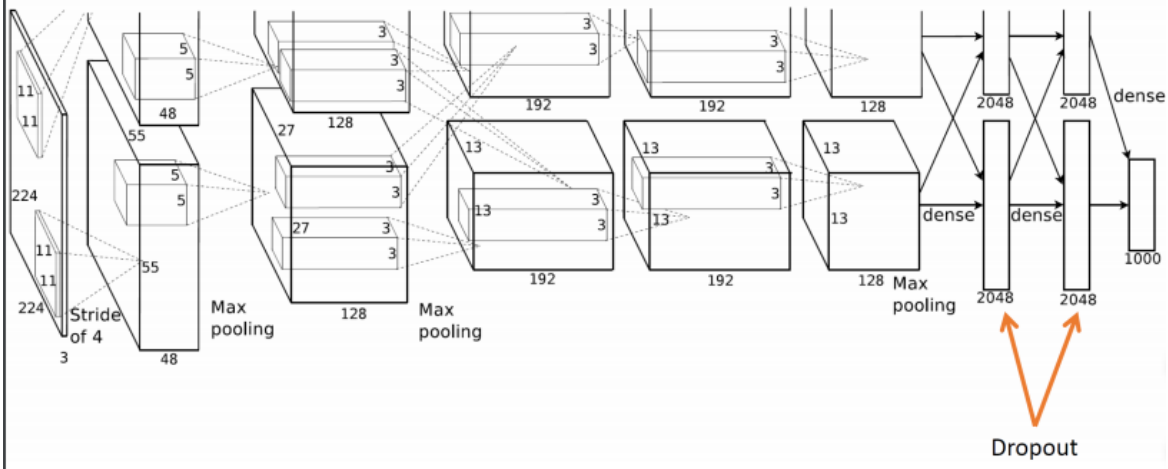
Imagine that you have a team of workers and the overall goal is to learn how to erect a building. When each of the workers is overly specialized, if one gets sick or makes a mistake, the whole building will be severely affected. The solution proposed by “dropout” technique is to pick randomly every week some of the workers and send them to business trip. The hope is that the team overall still learns how to build the building and thus would be more resilient to noise or workers being on vacation.

Data regularization techniques:

Temporal Modification of the net architecture in training phase

- Dropout

Dropout layers



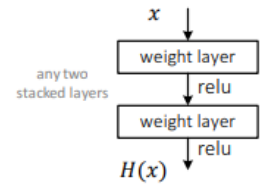
d. ResNet

Deep Residual Networks (ResNets)

- "Deep Residual Learning for Image Recognition". CVPR 2016
- A simple and clean framework of training "very" deep nets
- State-of-the-art performance for
  - Image classification
  - Object detection
  - Semantic segmentation
  - and more…

Deep Residual Learning

- Plaint net

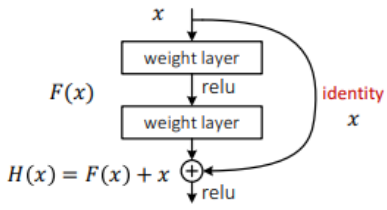


$H(x)$ is any desired mapping,
hope the 2 weight layers fit $H(x)$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Deep Residual Learning

- Residual net

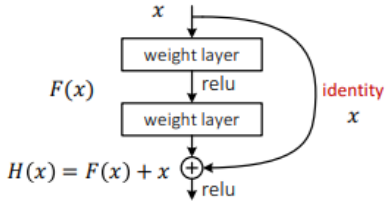


$H(x)$ is any desired mapping,
~~hope the 2 weight layers fit $H(x)$~~
hope the 2 weight layers fit $F(x)$
let $H(x) = F(x) + x$

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". CVPR 2016.

Deep Residual Learning

- $F(x)$ is a residual mapping w.r.t. identity



- If identity were optimal, easy to set weights as 0
- If optimal mapping is closer to identity, easier to find small fluctuations

Static samples vs Data signal flow

AlexNet could recognize 1000s of images.
ResNet could reach better then human performance.

- Though human can recognize
  - Single letters
  - Single sounds
  - Single tunes
  - Single pictures

- But in real life we handle
  - Texts
  - Speech
  - Music
  - Movies

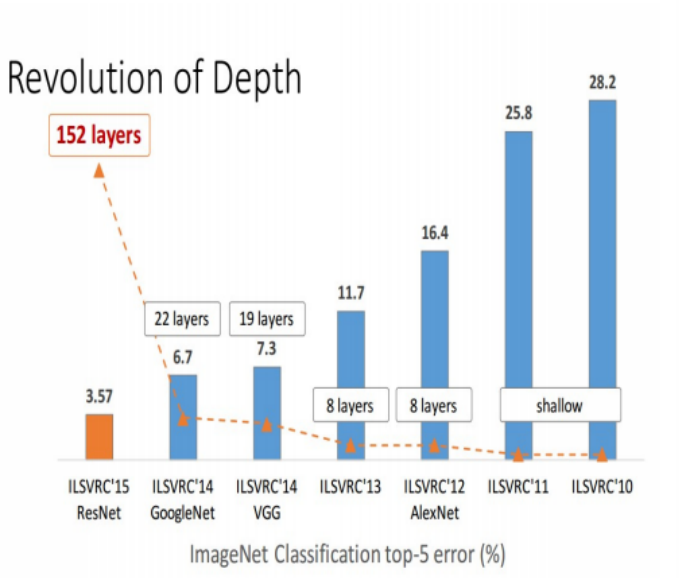
Story
(temporal analysis of sequential data)

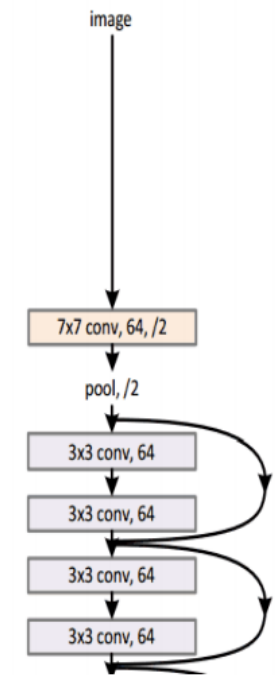
Can feed-forward neural networks (perceptrons, conv. nets) solve these problems?

DATA MEMORY

Naturally, we can extend the data dimension with the time, but this leads to data size and computational load explosion

Results with ResNets





ResNets had the lowest error rate at most competitions since 2015

1st places in all five main tracks

- ImageNet Classification: "Ultra-deep" 152-layer nets
- ImageNet Detection: 16% better than2nd
- ImageNet Localization: 27% better than2nd
- COCO Detection: 11% better than2nd
- COCO Segmentation: 12% better than2nd

How do ResNets address these issues?

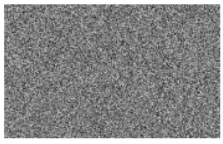
- **Representation** ability
  - No explicit advantage on representation (only re-parameterization), but
  - Allow models to go deeper
- **Optimization** ability
  - Enable very smooth forward/backward prop
  - Greatly ease optimizing deeper models
- **Generalization** ability
  - Not explicitly address generalization, but
  - Deeper+thinner is good generalization

https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4

---

e. Gradient ascent

Gradient Ascent



Generate a synthetic image that maximizes the response of a neuron.

This image has to be „natural". The response should not depend on pixels and can not have arbitrary values

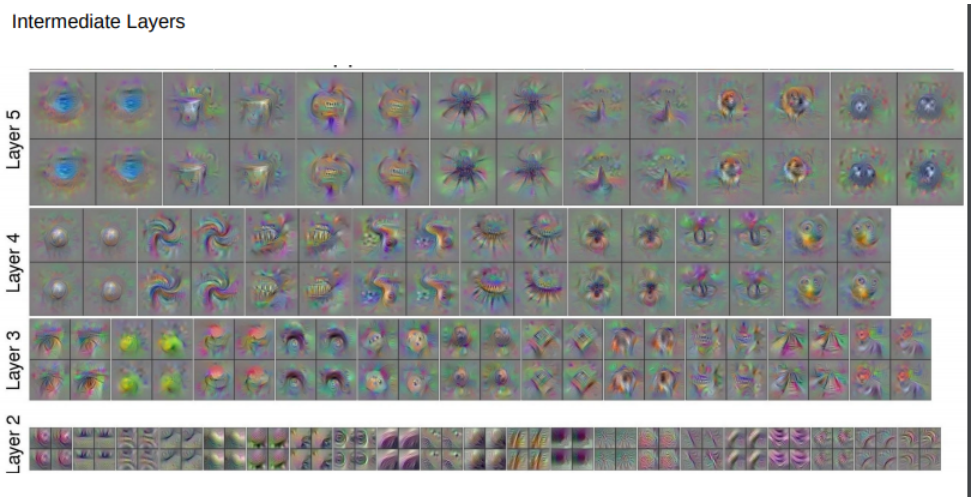
- Guassian blur on the image

- Clipping image values

- Clipping small gradients to 0

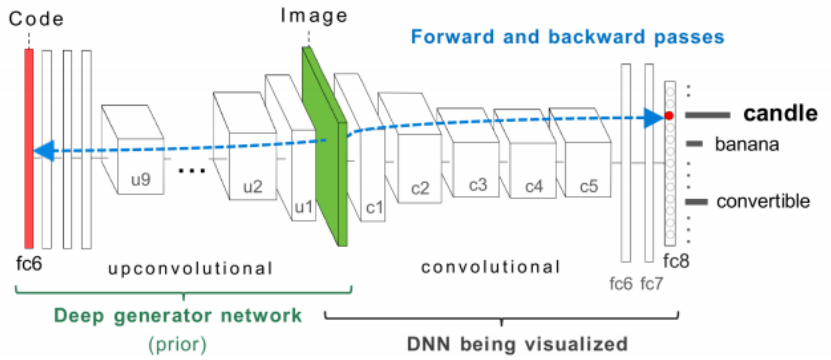
$$I^* = \arg\max_I f(I) + R(I)$$

Neuron value Natural image regularizer



Classes

Using a network which can learn feature inversion



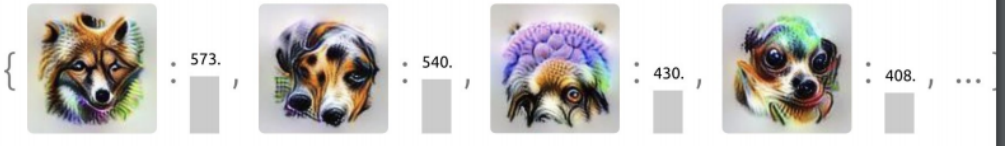
Gradient Ascent

Finding the maximizing patterns for each kernel

Making sense of these activations is hard because we usually work with them as abstract vectors:

$a_{2,4}$ = [0, 0, 0, 0, 31.4, 0, 0, 0, 49.0, 0, 0, 0, ...]

With feature visualization, however, we can transform this abstract vector into a more meaningful "semantic dictionary".



Gradient Ascent

We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

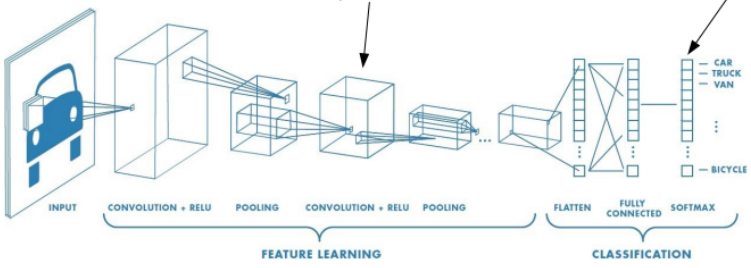
Could we generate and ideal image for a given class?

Normal training

Input Image - Given

Network parameters - Variables

Expected Label - Given



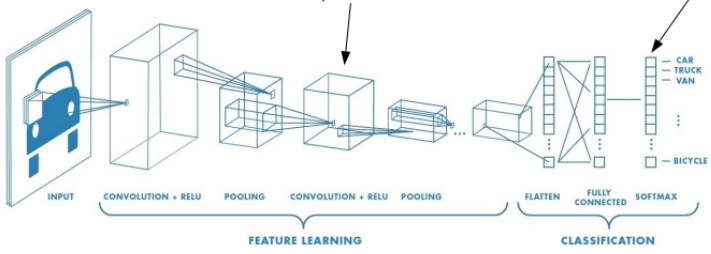
40

The gradient ascent method

Input Image - Variable

Network parameters - Given

Expected Label - Given



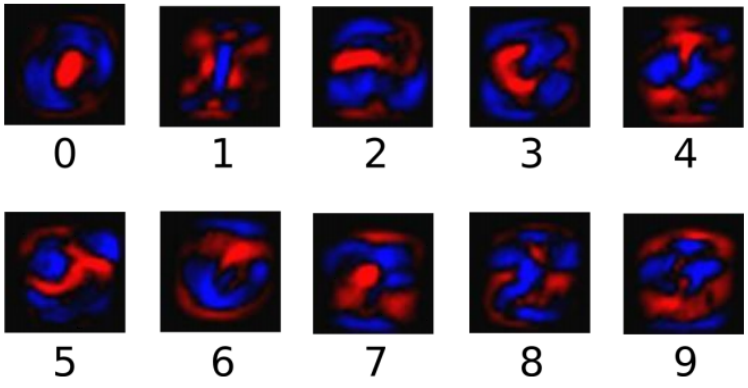
41

Gradient Ascent – activation maximization

We could search in our database and find typical samples.

It helps, but usually the network is good on this set (train accuracy). We are curious about those images which the network has not seen.

Could we generate and ideal image for a given class?



42

Neural Style Transfer

An interesting application of the gradient ascent method is neural style transfer

Could we use an input image and transform it into the style of an other input image?

https://demos.algorithmia.com/deep-style/

Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?

Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?

More weight to content loss

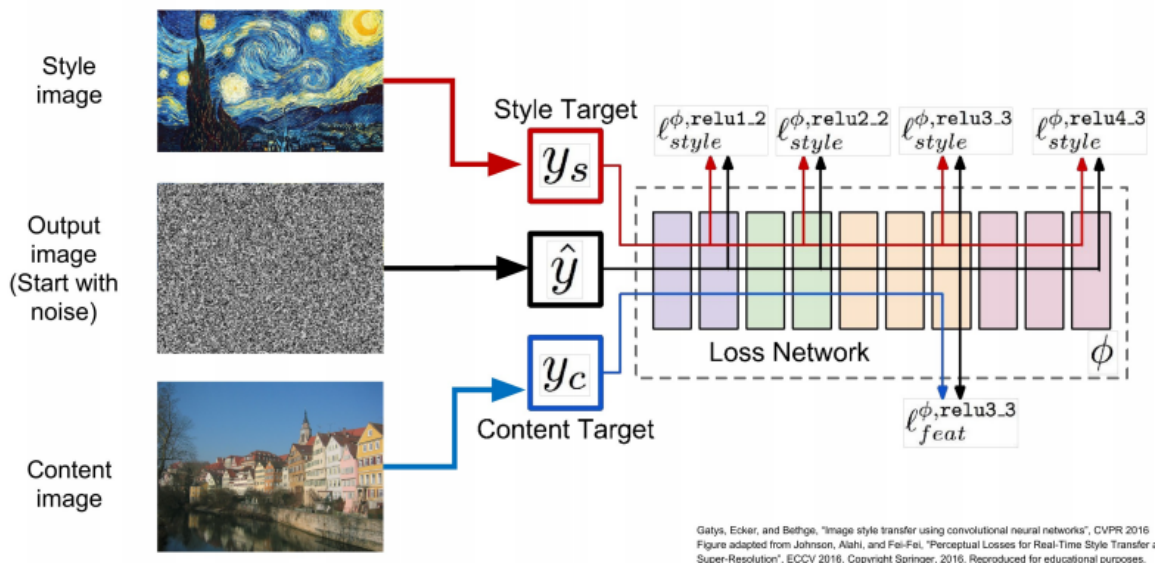
More weigh style loss

Fast Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?



Gatys, Ecker, and Bethge, "Image style transfer using convolutional neural networks", CVPR 2016
Figure adapted from Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer
Super-Resolution". ECCV 2016. Copyright Springer, 2016. Reproduced for educational purposes.

No. 2 Topic

a. Weight update strategies

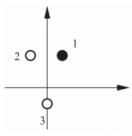
Weight update strategy

- Apply all the input vectors in one after the others, selecting them randomly
- Instance update
  - Update the weights after each input
- Batch update
  - Add up the modifications
  - Update the weights with the sum of the modifications, after all the inputs were applied
- Mini batch
  - Select a smaller batch of input vectors, and do with that as in the batch mode

Weight update: very simple example

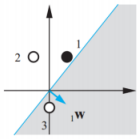
- Given a 3 input vector example

$$\mathbf{x}_1 = \begin{bmatrix}1\\2\end{bmatrix}, d_1 = 1; \quad \mathbf{x}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, d_2 = 0; \quad \mathbf{x}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, d_3 = 0;$$



- Assume that bias is zero
  (decision boundary will cross the origo)
- Random initialization $\mathbf{w}^T(1) = [1 \quad -0.8];$

Remember: the weight vector is orthogonal to the decision boundary!!!
Decision boundary: $x_1 - 0.8x_2 = 0$
Its orthogonal vector is: (1, -0.8)

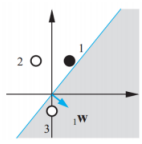


Weight update: very simple example

- Test with the first input vector $\mathbf{x}_1 = \begin{bmatrix}1\\2\end{bmatrix}, d_1 = 1;$ $\mathbf{w}^T(1) = [1 \quad -0.8];$ (d_j-y_j > 0)

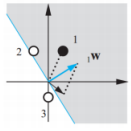
$$y_1(1) = sign(w^T(1)x_1) = sign\left([1 \quad -0.8] \quad \begin{bmatrix}1\\2\end{bmatrix}\right) = sign(1 - 1.6) = 0$$

The result is not OK! Positive misclassification: Instead of 1, the result is 0!!
(The normal vector points to the positive side of the decision boundary.)



Idea: **add** the vector pointing to the positively misclassified point to the orthogonal vector of the decision boundary, to rotate it **towards** the point!
w(k+1)=w(k)+x_1

$\mathbf{w}^T(2) = [1+1 \quad -0.8+2] = [2 \quad 1.2];$

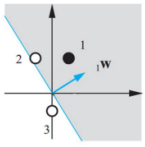


Weight update: very simple example

- Test with the second input vector $\mathbf{x}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, d_1 = 0;$ $\mathbf{w}^T(2) = [2 \quad 1.2];$ (d_j-y_j < 0)

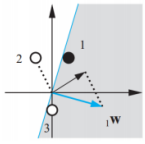
$$y_2(2) = sign(w^T(2)x_2) = sign\left([2 \quad 1.2] \quad \begin{bmatrix}-1\\2\end{bmatrix}\right) = sign(-2 + 2.4) = 1$$

The result is not OK! Negative misclassification: Instead of 0, the result is 1!!



Idea: **subtract** the vector pointing to the negatively misclassified point to the orthogonal vector of the decision boundary, to rotate it **away** the point!
w(k+2)=w(k+1)-x_2

$\mathbf{w}^T(3) = [2-(-1) \quad 1.2-2] = [3 \quad -0.8]$

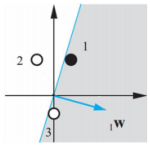


Weight update: very simple example

- Test with the third input vector $\mathbf{x}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, d_3 = 0;$ $\mathbf{w}^T(3) = [3 \quad -0.8];$ (d_j-y_j < 0)

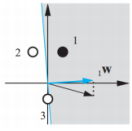
$$y_3(3) = sign(w^T(3)x_3) = sign\left([3 \quad -0.8] \quad \begin{bmatrix}0\\-1\end{bmatrix}\right) = sign(0 + 0.8) = 1$$

The result is not OK! Negative misclassification: Instead of 0, the result is 1!!



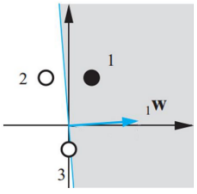
Again: **subtract** the vector pointing to the negatively misclassified point to the orthogonal vector of the decision boundary, to rotate it **away** the point!
w(k+3)=w(k+2)-x_3

$\mathbf{w}^T(4) = [3-0 \quad -0.8-(-1)] = [3 \quad 0.2];$



Weight update: very simple example

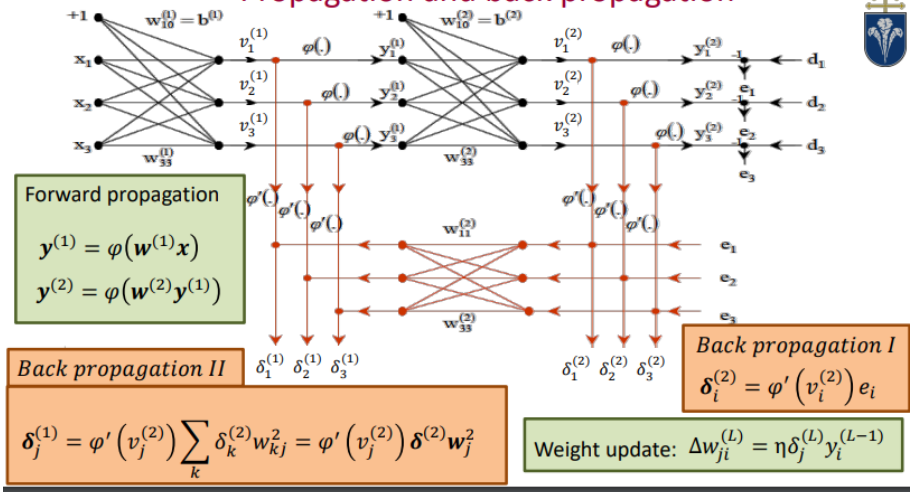
- Start again:
  - Test with the again with the first vector
    The result is OK!
    - Do not modify!!!
  - Test with the again with the second vector
    The result is OK!
    - Do not modify!!!
  - Test with the again with the third vector
    The result is OK!
    - Do not modify!!!
- Since all input vectors are correctly classified: we are ready



Formalization of the update rules

- Positive misclassification : ADD
  $\varepsilon = d_j - y_j = 1$ w(k+1)=w(k)+x_j
- Negative misclassification : SUBTRACT
  $\varepsilon = d_j - y_j = -1$ w(k+1)=w(k)-x_j
- Correct classification : DO NOTHING
  $\varepsilon = d_j - y_j = 0$ w(k+1)=w(k)
- In general:
  w(k+1)=w(k)+ ε x_j

Propagation and back propagation



Back-propagation

- Though we showed how to modify the weights with back propagation, its most important value that it can calculate the gradient
- The weight updates can be calculated with different optimization methods, after the gradients are calculated
- Various optimization method can drastically speed up the training (100x, 1000x)

Sequential back propagation

Adapting the weights of the FFNN (recursive algorithm)

$$w_{ij}^{(l)}(k+1) = w_{ij}^{(l)}(k) + \Delta w_{ij}^{(l)}(k)$$

$$\Delta w_{ij}^{(l)}(k) = ?$$

The weights are modified towards the differential of the error function (delta rule):

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial R_{emp}}{\partial w_{ij}^{(l)}}$$

The elements of the training set adapted by the FFNN sequentially

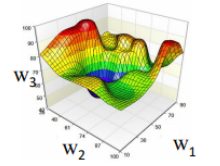
$$R_{emp} = R_{emp}(y(\mathbf{x}), d)$$

What are we optimizing here?

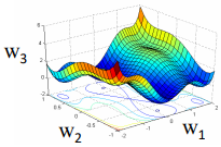
- Cost function in quadratic case for one $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair:

  $$\mathcal{E}_i = \left(\mathbf{d}_i - Net(\mathbf{x}_i, \mathbf{w})\right)^2$$

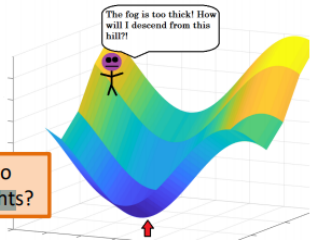
  - Error surface is in the $\mathbf{w}$ space
  - Error surface depends on the $\mathbf{x}_i \rightarrow \mathbf{d}_i$ pair
  - Moreover, we do not see the entire surface, just $\mathcal{E}$ and the gradients $\frac{\partial \mathcal{E}}{\partial w_{ij}^{(l)}}$



Error surface for $\mathbf{x}_i \rightarrow \mathbf{d}_i$



Error surface for $\mathbf{x}_k \rightarrow \mathbf{d}_k$



When and how to update the weights?

Update strategies

- Single vector update approach (instant update)
  - Weights are updated after each input vector
- Batched update approach
  - All the input vectors are applied
    - this is actually the correct entire error funtion, which is used by the original Gradient Descent Method
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the entire batch is calculated
- Mini batch approach
  - When the number of inputs are very high (10^4-10^6), batch would be ineffective
  - Random selection of m input vectors (m is a few hundred)
  - Updates (Δw_{ij}) are calculated for each vector, and averaged
  - Update is done with the averaged values (Δw_{ij}) after the mini batch is calculated
  - Works efficiently when far away from minimum, but inaccurate close to minimum
  - Requires reducing learning rate

Remember, each approach optimizes different error surfaces!!!

---

b. ReLU and dying ReLU problem

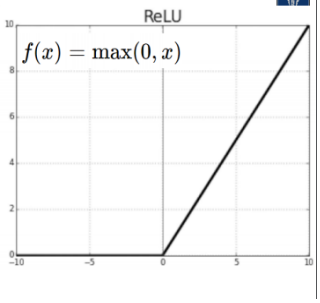
Activation function III

- Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

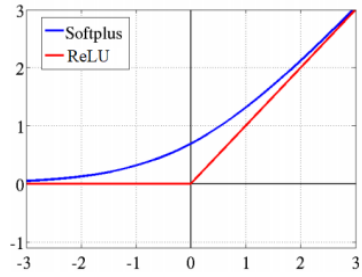
  - Most commonly used nonlinearity in hidden layers of deep neural networks
- Derivative of ReLU

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



ReLU: Rectified Linear Unit

- Very easy to calculate
  - Implementation is a simple sign comparison and replacing with 0 if negative
- Also easy to calculate its derivative
- Also called:
  - Ramp function
  - Half-wave rectifier
- Orders of magnitude learning speed advantage
  - Due to non-compressed gradient
- Smooth analytic approximation is the Softplus function
- Asymptotically reaches ReLU



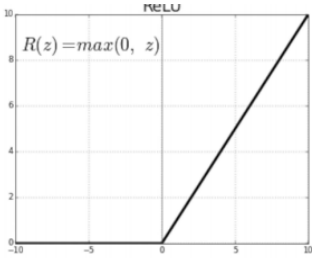
$f(x) = \max(0, x)$ ReLU

$f(x) = \log(1 + e^x)$ Softplus

Most used in hidden layers in deep neural networks (as of 2019)!

- Activation function
- Half-wave rectifier
- Not compressing the gradient
  - learns much faster
- ReLU types
  - Softmax
  - Leaky ReLU
  - ELU, SELU Relu6

Dying ReLU problem



- During training it happens that the weight composition of a neuron got a certain combination in a high gradient situation (when large jump happens during the optimization), which leads to generate zero output from that point on.
  - Happens typically with large learning rate
  - E.g. a very large negative value appears in the bias position
- That neuron will output zero for each input vector from that point
  - Irreversible
  - No contribution to the decision
  - A usefull neuron selectively fires to a set of input vector having the same properties
- In some bad cases, even 40% of the neurons dies in coarse of a long training (Vanishing Gradient problem)

$$\mathbf{y}^{(L)} = R\ (\mathbf{w}^{(L-1)T}\mathbf{y}^{(L-1)} + b^{(L-1)}) = \mathbf{R}(v)$$

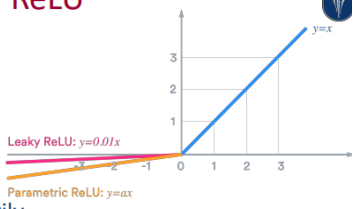
$$\Delta w_{ji}^{L} = \eta R'(v) e_i y_i^{(L-1)}$$

Avoid the absolute zero part!
Introduction of Leaky ReLU.

10/8/2019

Leaky ReLU

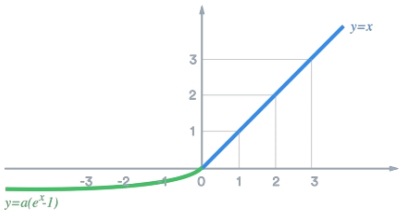
- No constant zero output
- Neurons do not die
- **Parametric ReLU**
  - Variation of leaky ReLU
  - a is a hyper-parameter:
    - Tuned during training
- Leaky ReLUs are not necessarily superior than normal ReLU
- It is an option, if normal ReLU is not performing well



$$f(x) = \max(x, ax)$$
where:
a is a small positive number

ELU: Exponential linear units

- Variation of leaky ReLU
  - No constant zero output
  - Neurons do not die
  - Mean activation closer to 0 in the negative region
- Obtains higher classification accuracy than ReLU, but requires more computations
- a is a hyper-parameter:
  - Tuned during training

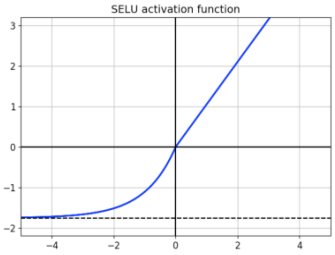


$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ a(e^x - 1) & \text{otherwise} \end{cases}$$

a is a hyper-parameter to be tuned and $a \geq 0$ is a constraint.

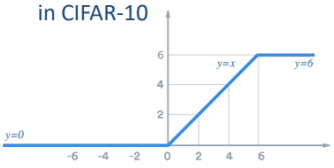
SELU: Scaled Exponential linear units

- Variation of leaky ELU
- Two fixed parameters
  - Not trained, but selected to be fixed
  - λ is the scaling parameter



$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leqslant 0 \end{cases}$$

- Variation of ReLU **ReLU6**
  - Capped at 6
  - 6 is a choosable parameter
- Shown to learn sparse features faster
- Turned out to be usefull in CIFAR-10



10/8/2019 22

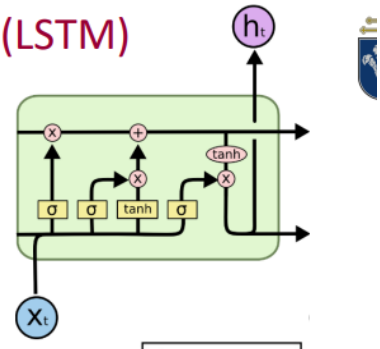
ReLUs do not require input normalization to prevent them from saturating. However, Local Response Normalization aids generalization.

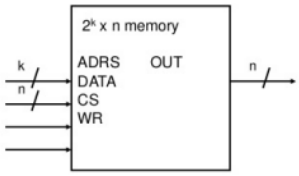
---

c. LSTM cell

Long Short Term Memory (LSTM)

- Was originally introduced Hochreiter & Schmidhuber (1997)
- Idea:
  - To be able to learn long term dependences
  - Collects data when the input is considered to be relevant
  - Keeps it as long as it considers to be important
  - Technique:
    - Handle the state as a memory with minor modifications
      - No matrix multiplication
      - No tanh
      - Apply memory handling kind signals
        - » data in, data out, write, enable

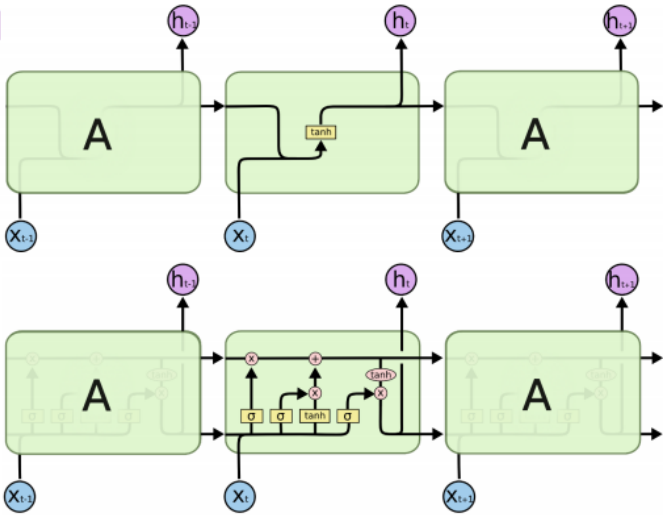




2019-11-25 http://colah.github.io/posts/2015-08-Understanding-LSTMs/ 49

Derivation of LSTM

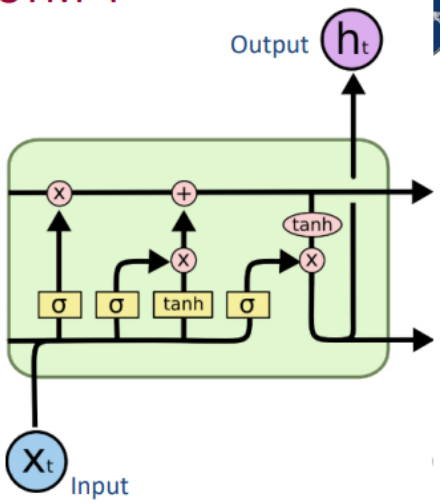
- Repeating module in Normal RNN
  - concatenates the input and the state
  - A neural network with tanh output and repeats the result
- LSTM
  - Uses the state as a memory
  - Uses 4 neural nets to control the memory
    - Forget_gate, Input_gate, State_update, Output_gate



2019-11-25 50

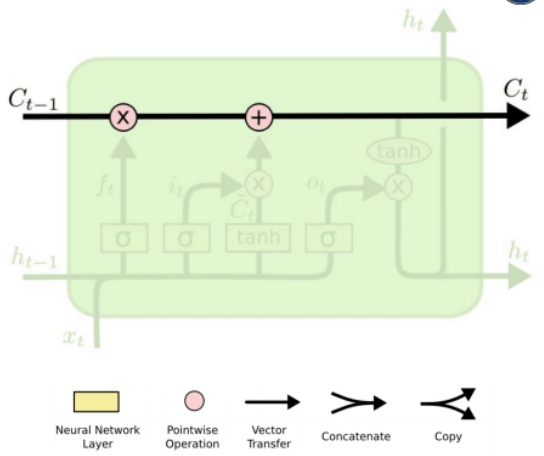
Components of LSTM I

- All wires represents vector
  - Vector transfer
  - Vector concatenation
  - Vector copy
- Neural nets with (yellow boxes)
  - **Multi-layer NN** with *tanh* activation function used for update value calculation
  - **Multi-layer NN** with *logistic* activation function (sigmoid) used for value selection (kind of addressing)
- Pointwise operation (pink circles)
  - Pointwise multifaction
  - Pointwise addition



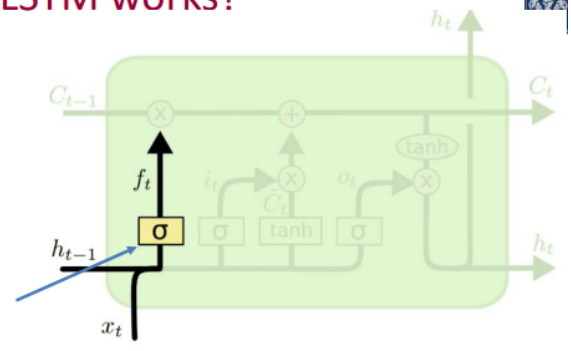
Components of LSTM II

- State of the LSTM
  - This is the actual memory,
  - It can pass the previous values with or without update
  - Represented by the upper black line
  - Indicated with C_t
- Old content can be removed value-by-value
- New content can be added



How LSTM works?

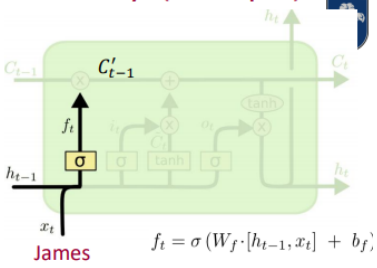
- Step 1
  - Combines input and previous output (concatenation)
  - Selects which values to forget
    - Sort of addressing
    - Done by the **"Forget Gate"**
    - Neural net with sigmoid output



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Updating state memory (Example)

- Input: "James"
- **Forget Neural** network figures out:
  - Analyzes the concatenated vector
  - Name, Subject of a sentence, Male
- Selects which values to forget and how much
  - Position and weight
- Task:
  - Update gender of the subject (forget the old value)
  - Gender might be represented with a variable
    - c_1: value proportional with the probability that the subject is a male
    - c_2: represents weather
  - Calculate the forget factor of the gender memories
    - 0 completely get rid of it
    - 1 keep the previous value
    - 0 .. 1 partial forget
    - Adressing and suppressing!!!



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

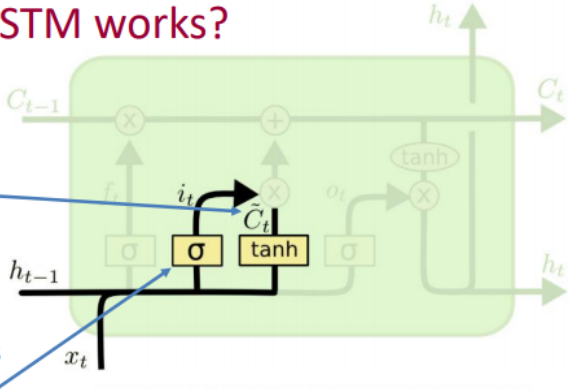
c_1: subject's gender *f1: forget factor of c_1* *c1 value after partial forget*

$$C_{t-1} = \begin{bmatrix} -0.5 \\ 0.2 \\ \vdots \end{bmatrix} \quad f_t = \begin{bmatrix} 0.1 \\ 1 \\ \vdots \end{bmatrix} \quad C'_{t-1} = \begin{bmatrix} -0.05 \\ 0.2 \\ \vdots \end{bmatrix}$$

Not to forget c_2

How LSTM works?

- Step 2
  - Calculation of the state update
    - Done by the **"Cell Network"**
    - Not yet the new value, only the update value
    - Neural Net with *tanh*
  - Selection of the state values to be updates (Addressing)
    - Done by the **"Input Gate"**
    - Neural Net with sigmoid
  -

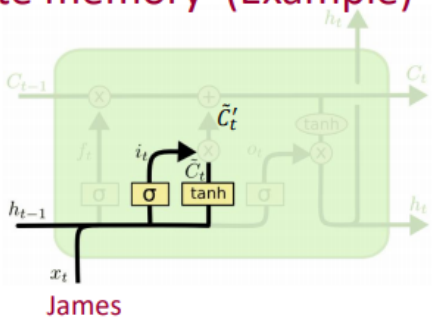


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Updating state memory (Example)

- Input: "James"
- Input Gate figures out:
  - Analyze the concatenated vector
  - Select which values to update (ENABLE!!!)
  - Calculate the update weights
- Cell Network calculates:
  - The update values
- Task:
  - Update gender of the subject (calculate the update value)
  - Gender might be represented with a variable
    - c_1: value proportional with the probability that the gender is male
    - c_2: represents weather
  - Calculate the update factor of the gender memories
    - 0 not to update
    - 1 fully update
    - 0 .. 1 partial update
    - ADRESSING!!!



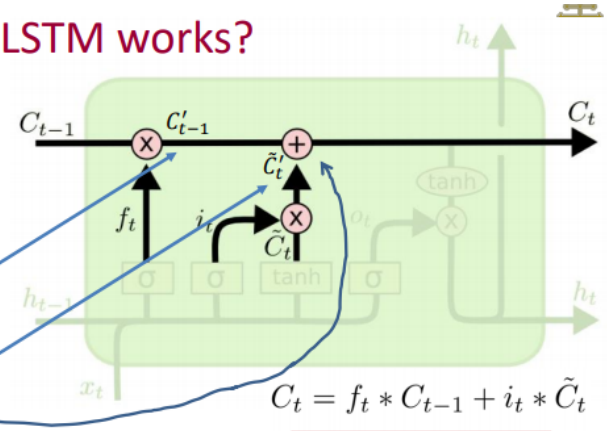
c_1: subject gender estimate value *f1: update factor of c_1* *c1 update value*

$$\tilde{C}_t = \begin{bmatrix} 0.9 \\ -0.75 \\ \vdots \end{bmatrix} \quad i_t = \begin{bmatrix} 0.8 \\ 0 \\ \vdots \end{bmatrix} \quad \tilde{C}'_t = \begin{bmatrix} 0.72 \\ 0 \\ \vdots \end{bmatrix}$$

Not to modify c_2

How LSTM works?

- Step 3
  - Calculation of the state update
    - The old state
      - With the forgotten values in the vector
    - And the state update
      - With update vector
    - **Are added up**



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

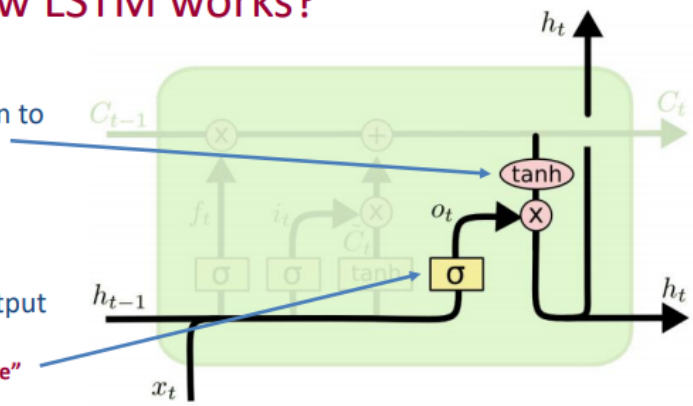
$$C_t = C'_{t-1} + \tilde{C}'_t = \begin{bmatrix} -0.05 \\ 0.2 \\ \vdots \end{bmatrix} + \begin{bmatrix} 0.72 \\ 0 \\ \vdots \end{bmatrix} = \begin{bmatrix} 0.67 \\ 0.2 \\ \vdots \end{bmatrix}$$

c_1: *subject gender's estimate value update*

c_2: *(weather) unchaged*

How LSTM works?

- Step 4
  - Apply activation function to the output
    - Squeeze the values between -1 and +1
    - Done by *tanh* activation function
  - Selection of the new output values (Addressing)
    - Done by the **"Output Gate"**
    - Not all the state value is released in each step
    - **Output Gate** decides which values are relevant in this step



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Output vector can be sparse

- Output gate might enables
  - All values of C_t
  - Fraction of C_t (sparse)
  - None of C_t
- C_t can be sparse

$$tanh(C_t) = \begin{bmatrix} 0.2 \\ 0.98 \\ -0.97 \\ -0.1 \\ 0.98 \\ 0.8 \\ 0.2 \\ 0.3 \\ -0.99 \\ 0.8 \\ 0.7 \end{bmatrix}$$

Values are bounded

$$o_t = \begin{bmatrix} 0.01 \\ 0.85 \\ 0.75 \\ 0.1 \\ 0.2 \\ 0.8 \\ 0.1 \\ 0.1 \\ 0.02 \\ 0.9 \\ 0.8 \end{bmatrix}$$

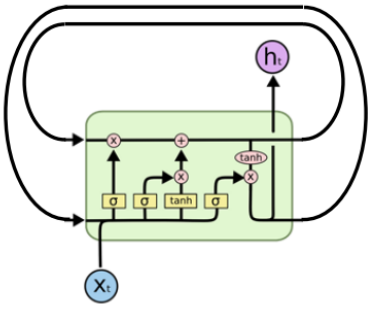
Enabling factor: Enabled values are red

$$h_t = o_t * tanh(C_t) = \begin{bmatrix} 0.002 \\ 0.83 \\ -0.73 \\ -0.01 \\ 0.2 \\ 0.64 \\ 0.02 \\ 0.03 \\ -0.02 \\ 0.72 \\ 0.63 \end{bmatrix}$$

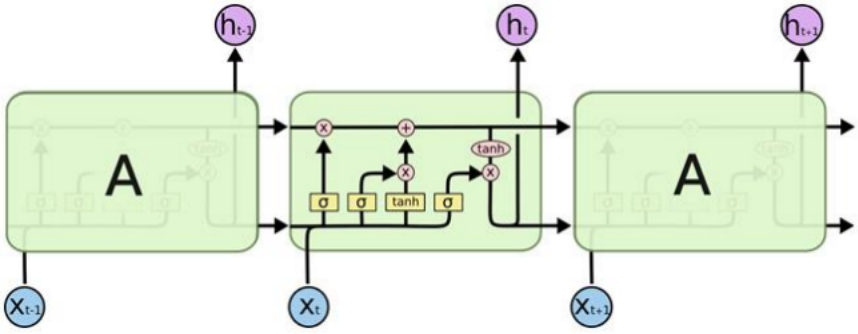
Output vector: Enabled values are red Disabled values (gray) will appear on the output, but with reduced values

LSTM network

- General form of an LSTM network

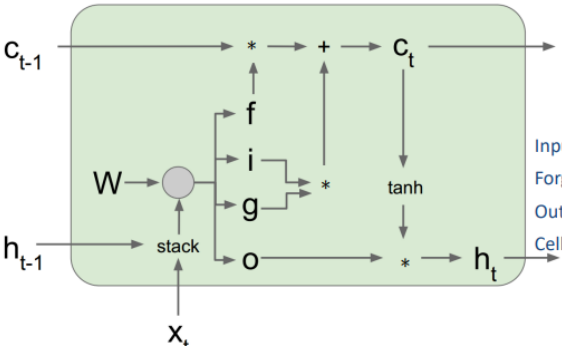


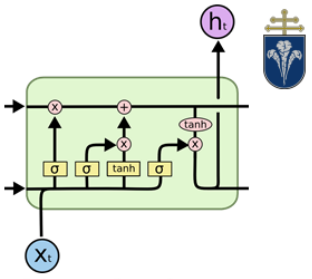
Unrolling LSTM network



Gradient calculation in LSTM

Reformulating equations





Input
Forget
Output
Cell Net

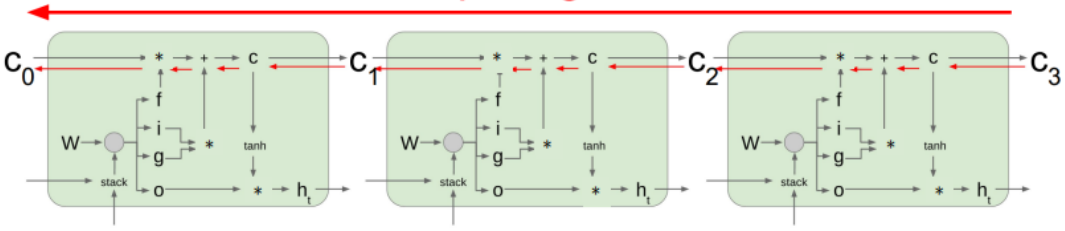
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f * c_{t-1} + i * g$$

$$h_t = o * \tanh(c_t)$$

Gradient calculation in LSTM

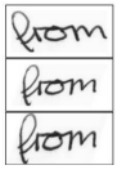
Uninterrupted gradient flow!



- Though we multiply the memory content with a smaller than 1 number
- And the W matrix is part of the memory update
- But it still preserves the content for longer time
- As it comes from the name: It is a elongated time **short term** memory

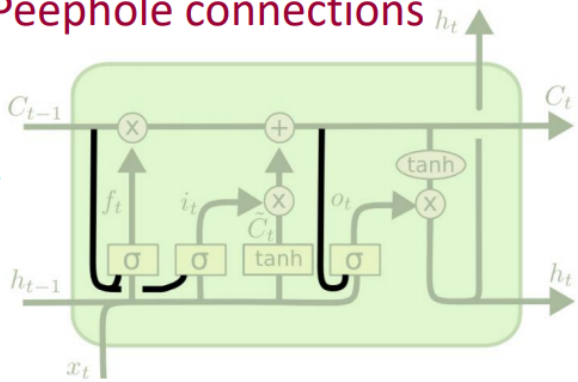
Achevements with LSTM networks

- Record results in natural language text compression
- Unsegmented connected handwriting recognition
- Natural speech recognition
- Smart voice assistants
  - Google Translate
  - Amazon Alexa
  - Microsoft Cortana
  - Apple Quicktype
- 95.1% recognition accuracy on the Switchboard corpus, incorporating a vocabulary of 165,000 words
  - Continuous spontaneous English native speech



Variants of LSTM I : Peephole connections

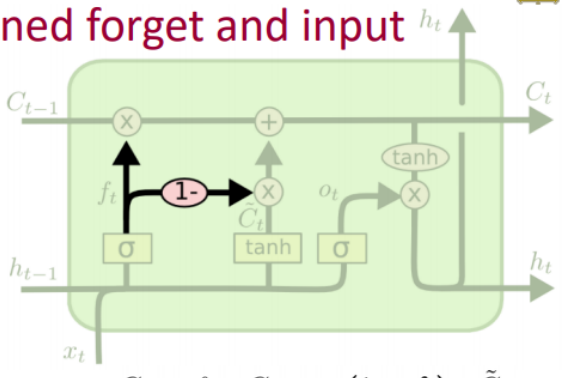
- Introduced by Gers & Schmidhuber (2000)
- All the three gates receives input from the previous state and the input
- Since output can be sparse this version has more information for gating
  - addressing and weighting



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Variants of LSTM II : Joined forget and input

- Input and forget gates has practically the same role
- Why not to join them?



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

d. Convolution as a mathematical operation in continuous and discrete cases

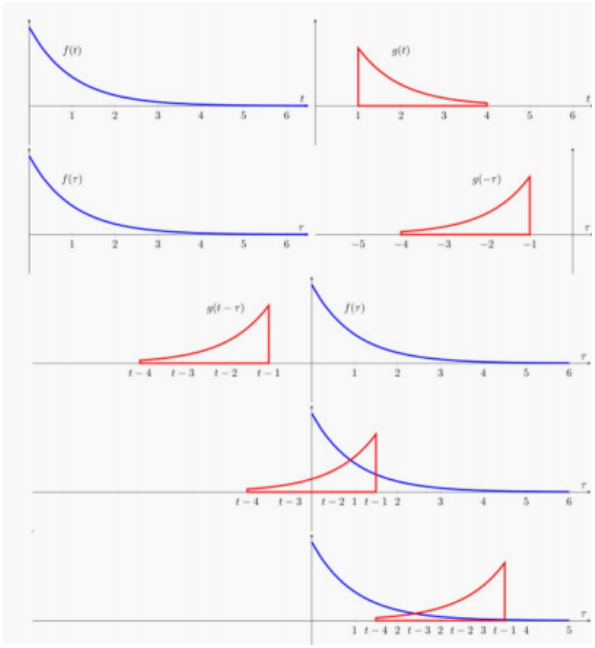
Convolution

- Convolution is a mathematical operation that
  - does the integral of the product of 2 functions (signals),
  - with one of the signals flipped and shifted
- Mathetmatically:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau) g(\tau)\, d\tau$$

- Convolution is commutative

10/22/2019



1. Flipp *g* signal
2. Slide the flipped *g* over *f*
3. Integrate the product in continious space
   or Multiply and accumulate it in discrete space with each shift

Discrete convolution

- For continiuous:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau) g(\tau)\, d\tau$$

- For discrete functions:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n-m]$$
$$= \sum_{m=-\infty}^{\infty} f[n-m] g[m]$$

Convolution vs correlation II

As the only difference is kernel flipping…

Why convolution rather that correlation?

- Commutativity, Associativity, Distributivity helps to prove mathematical statements
- Since the network learns its own weights, it is invariant whether that flip is there or not (just a convention)
- In many cases, correlation is implemented even when it is called convolution

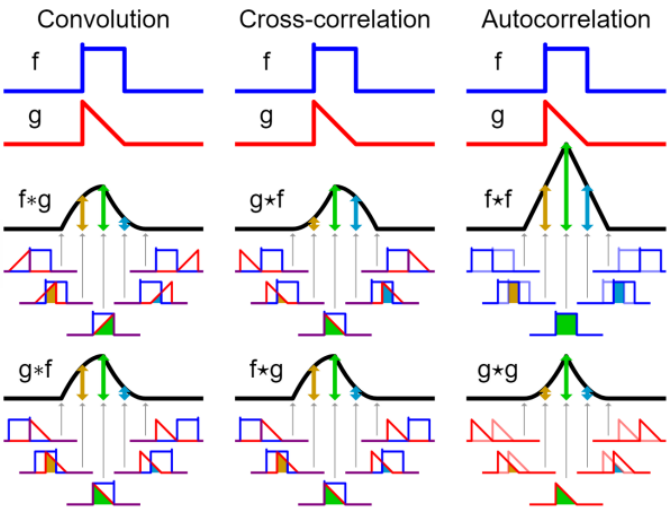
Convolution vs correlation I

- Cross-Correlation:

$$(f \star g)(\tau) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f^*(t)\, g(t+\tau)\, dt$$

- f^* : complex conjugate
- When f is symmetric, $f * g = f \star g$ (otherwise not)

10/22/2019



No. 3 Topic

a. Newton optimization method

Simplest 2nd order Gradient descent method: Newton Method

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \nabla f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^{\mathrm{T}} \mathbf{H}\big(f(\mathbf{x}_0)\big)(\mathbf{x}-\mathbf{x}_0)$$

- Replacing $(\mathbf{x}-\mathbf{x}_0) \to \Delta\mathbf{x}$ and differentiating it with $\Delta\mathbf{x}$, assuming that we can jump to a minima, where: $\nabla f(\mathbf{x}) \approx 0$

$$0 = \frac{\partial}{\partial \Delta\mathbf{x}}\left(f(\mathbf{x}_0) + \Delta\mathbf{x}^{\mathrm{T}} \nabla f(\mathbf{x}_0) + \frac{1}{2}\Delta\mathbf{x}^{\mathrm{T}} \mathbf{H}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x} \right) = \nabla f(\mathbf{x}_0) + \mathbf{H}\big(f(\mathbf{x}_0)\big)\Delta\mathbf{x}$$

Constant→0 $(\Delta x)' \to 1$ $(\tfrac{1}{2}(\Delta x)^2)' \to \Delta x$

Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\big(f(\mathbf{x}_0)\big)^{-1} \nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta \mathbf{H}\big(f(\mathbf{x}(n))\big)^{-1} \nabla f(\mathbf{x}(n))$$

Properties of Newton optimization method

- When ***f*** *is a positive definite quadratic function, Newton's method* jumps in a single step to the minimum of the function directly.

- *Newton's method can* reach the critical point much faster than 1st order gradient descent.

Newton optimization:

$$\Delta\mathbf{x} = -\mathbf{H}\big(f(\mathbf{x}_0)\big)^{-1} \nabla f(\mathbf{x}_0) \qquad \mathbf{x}(n+1) = \mathbf{x}(n) - \eta \mathbf{H}\big(f(\mathbf{x}(n))\big)^{-1} \nabla f(\mathbf{x}(n))$$

Newton's algorithm

Algorithm Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m examples
while stopping criterion not met **do**
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute Hessian inverse: $\boldsymbol{H}^{-1}$
  Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Typically not used, due to the computational complexity
Parameter space much higher than first order (where it is already very high)

b. Ensembling, bagging

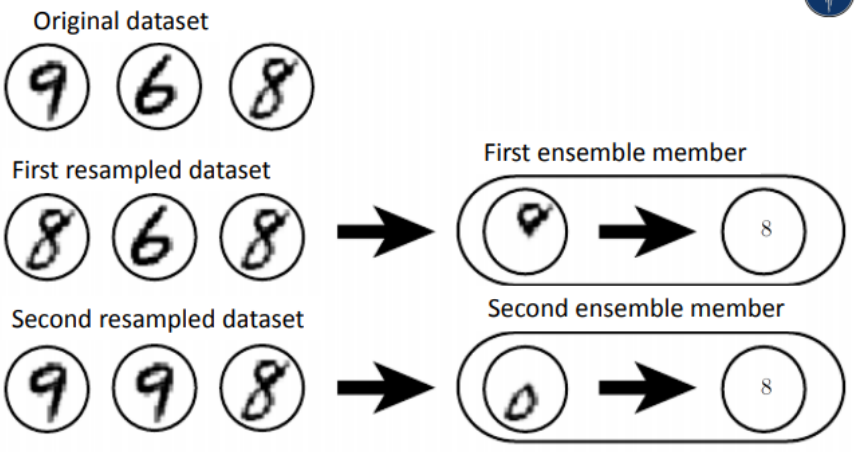
Ensemble methods:
• Network duplication
• Bagging
• Dropout

Ensemble methods

- Idea of ensemble methods:
  - Generate multiple copies of your net
    - Same or slightly modified architectures
  - Train them separately
    - Using different subsets of the training sets
    - Different objective functions
    - Different optimization methods
  - The different trained models have independent error characteristics
  - Averaging the results will lead to smaller error
- Requires more computation and memory both in training and inferencing (testing) phase

Bagging

- Construct k different datasets
- Each with a subset of the data, but with duplications
- Trains with these
- Make result averaging

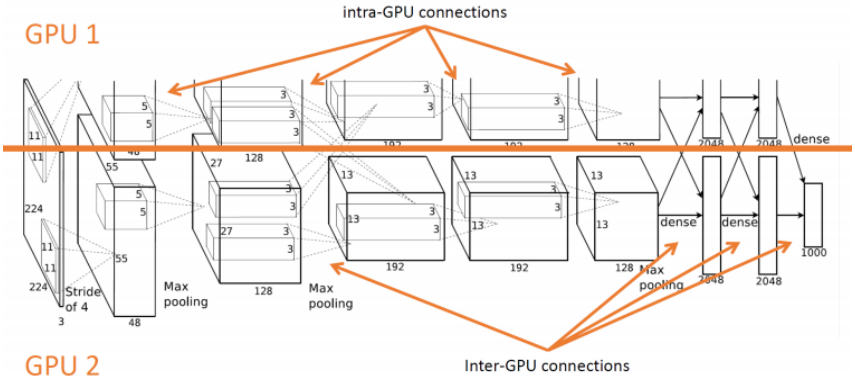


First learns the upper loop, the second the lower. When both say yes, it is an 8.

Ensembling: duplicating the network I

- Train two architecturally identical copies of the network on two GPUs
  - Half of the neuron layers are on each GPU
    - GPUs communicate only in certain layers
  - Improvement (as compared with a net with half as many kernels in each convolutional layer trained on one GPU):
    - Top 1 error rate by 1.7%
    - Top 5 error rate by 1.2%

Ensembling: duplicating the network II



---

c. Comparison of loss function

Loss functions

- Loss function determines the training process
  - Tells the net, whether an error is big or small, and penalize accordingly
  - There can be other errors, not just the difference of the output and the desired output
- Most used loss function types:
  - Quadratic, in case of regression

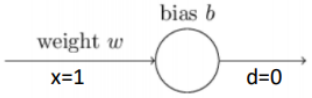
$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

  - Conditional log-likelihood, in case of classification
    The sum of the negative logarithmic likelihood is minimized

$$C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(-logP(\mathbf{y}_k|\mathbf{x}_k, \mathbf{w})\right)$$

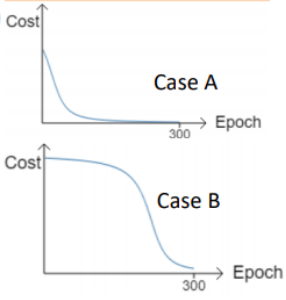
What is the problem with quadratic loss function in classification tasks?

- In case of classification, the convergence can be very slow
- Consider the following very simple case



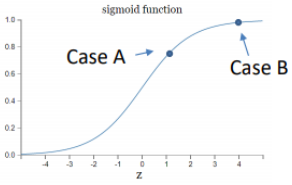
Check out the example! http://neuralnetworksanddeeplearning.com/chap3.html

- **Case A**: Start the learning from w(0)=0.6, b(0)=0.9
  - Loss function decreases quickly
- **Case B**: Start the learning from w(0)=2, b(0)=2
  - Loss function decreases very slowly at the beginning
- Why is that?
  - Because the Δw is proportional with the gradient



Calculation of the gradient

- Loss function:
  $L = {}^1/_2\,(d-y)^2, \quad where \quad y = \sigma(wx+b)$
- Gradient, using chain rule:
  $\frac{\partial L}{\partial w} = (y-d)\,\sigma'(wx+b)x = y\sigma'(wx+b)x$
- **Case A**: w(0)=0.6, b(0)=0.9, x=1, d=0
  - Slope of the gradient is fine: $(wx+b) = 1.5$
  - Fast convergence
- **Case B**: w(0)=2, b(0)=2, x=1, d=0
  - Slope of the gradient is very small: $(wx+b) = 4$
  - Very slow convergence



Sigmoid with quadratic loss function leads to very small gradient even at large error, when the argument of the sigmoid is a large value.

Introducing Cross Entropy

- Idea: replace the quadratic Loss function with a more appropriate Loss function: Try cross entropy!
- In general: $C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$
- $C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(d_k\, logP(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) + (1-d_k)log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\right)$
  - Is it always positive?
    - d_k is either 0 or 1 (binary classification)
      - Either the first or the second term is zero
    - $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) = \sigma(\mathbf{w}\mathbf{x}_k + b)$
      - The probability is the output of the network
      - Due to the sigmoid, it is between 0 and 1
      - Therefore, its logarithm is negative

10/8/2019 http://neuralnetworksanddeeplearning.com/chap3.html

- Is it a good loss function?

  Good decision (small loss):
  - When d_k is 0 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 0, than $-log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right) \sim \mathbf{0}$
  - When d_k is 1 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 1, than $-log\left(P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right) \sim \mathbf{0}$

  Bad decision (large loss):
  - When d_k is 0 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 1, than $-log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right) \sim \infty$
  - When d_k is 1 and $P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})$ is close to 0, than $-log\left(P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right) \sim \infty$

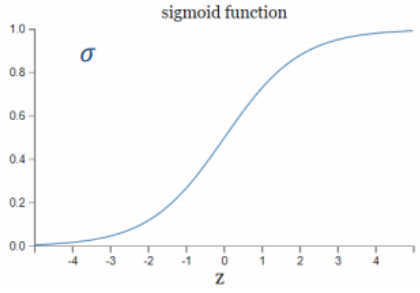
Why is cross entropy good?

$C(\mathbf{w}) = -\frac{1}{K}\sum_{k=1}^{K}\left(d_k\, logP(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w}) + (1-d_k)log\left(1-P(\mathbf{y}_k|\mathbf{x}_k,\mathbf{w})\right)\right)$

- Because its partial derivative does not contain σ'

$$\frac{\partial C}{\partial w_j} = \frac{1}{K}\sum_{k=1}^{K} x_j(\sigma(\mathbf{w}\mathbf{x}+\mathbf{b}) - d)$$

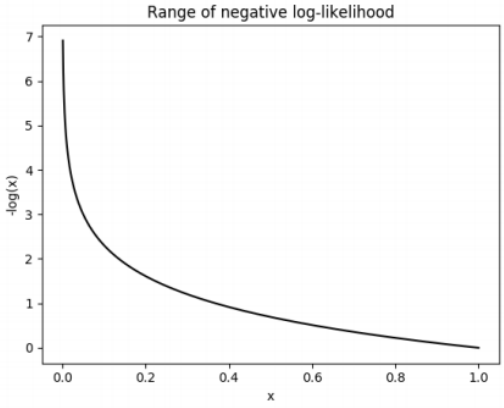
- The gradient is proportional with the value of the sigmoid, and not with its derivative!



Loss function for softmax: Negative log-likelihood

- $L(\boldsymbol{y}) = \sum_{k=1}^{K} -\log(y)$
- The negative logarithm of the probability of the correct decision classes are summed up
- It is small, if the confidence of a good decision was high for a certain class
- Large, when the confidence is low
- Partial derivative of a softmax layer with negative log-likelihood:

$$\frac{\partial C}{\partial v_j} = y_j - 1$$

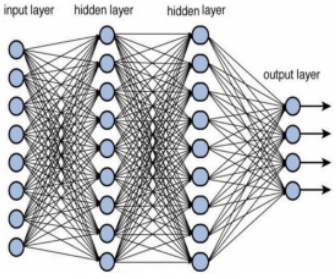


Probability type loss: Cross Entropy and Softmax

- Mathematically:
  - Normalized exponential functions of the units
- Probability distribution of n discrete classes:
  - One-of-n classes problems
  - $0 < y_i < 1$
  - $\sum_{i=1}^{n} y_i = 1$
- Architectural difference:
  - Previously learned activation functions were based on the inputs of one neuron
  - Softmax combines a layer of output neurons

$$y_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

$$\boldsymbol{v} = \boldsymbol{w}^T \boldsymbol{x}$$



10/22/2019

https://medium.com/deep-learning-demystified/loss-functions-explained-3098e8ff2b27

d. Machine learning vs traditional programming

Conventional approach

- Trivial, or at least analitically solvable tasks
  - Well established mathematical solution exist or at least can be derived
- Example:
  - Finding well defined data constellations in a database
  - Formal verification of the operation is easy

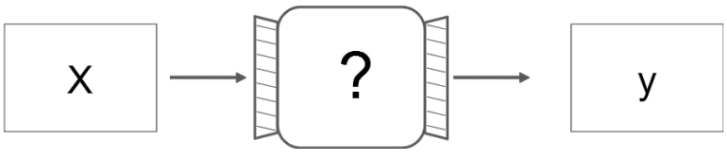
Machine learning approach

- Complex underspecified tasks
  - No exact mathematical solution exists, the function to be implemented is not known
- Example:
  - Searching for "strange" data constellations in a database
  - Verification of the operation is difficult

In case of very complex problems, verification of the operation is very difficult. Typically done by exhaustive testing in case of machine learning.

Machine learning

We consider each task as an input-output problem



X: scalar, vector, array or a sequence of these (incl. text)

size(X) vs size(Y)
Data reduction
Data generation

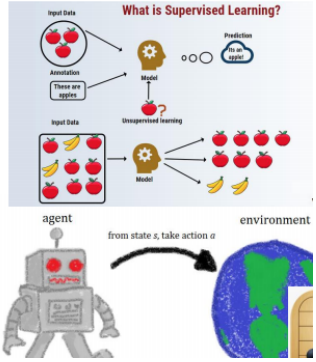
Y: Decision or scalar, vector, array or a sequence of these (incl. text)

Deep learning - why now?

1. Appearance of machine learning methods and frameworks, optimization know-how, new tools for rapid experimentation
2. New architectures are available for computation
   - (1980: VIC-20 5kb RAM, MOS 6502 CPU 1.02Mhz)
   - (2018: NVIDIA GeForce GTX 1080, 8GB RAM, 1733 MHz, 2560 cores)
3. Vast amount of data is available
   - Billions of labeled images available quasi free

Typical Machine Learning Types

- **Supervised Learning**
  - Learning from labeled examples (for which the answer is known)
- **Unsupervised Learning**
  - Learning from unlabeled examples (for which the answer is unknown)
- **Reinforcement Learning**
  - Learning by trial and feedback, like the "child learning" example



What is intelligence? • The ability to acquire and apply knowledge and skills. Intelligence is the ability to adapt to change "Stephen Hawking"

Providing computers the ability to learn without being explicitly programmed:

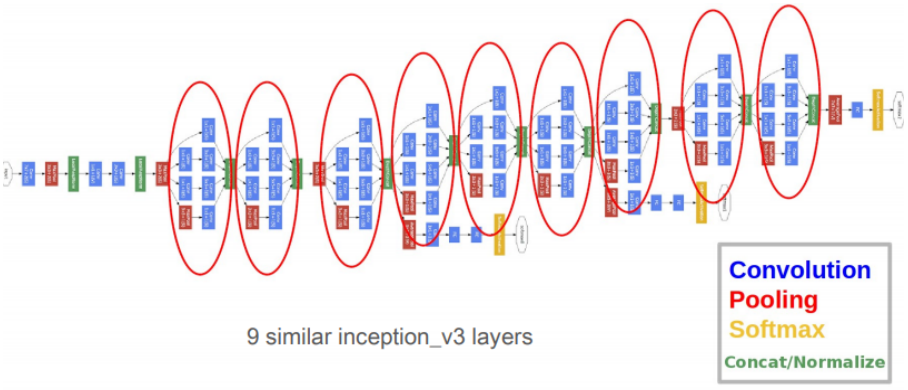
Involves: programming, Computational statistics, mathematical optimization, image processing, natural language processing etc…

As opposed to **Traditional Computers** where	At **Neural Networks**
- the math of the functionality is known - the known math should be programmed	- the math behind the functionality is unknown - the functionality is "illustrated" with examples

---

e. Inception

Inception module:



Inception

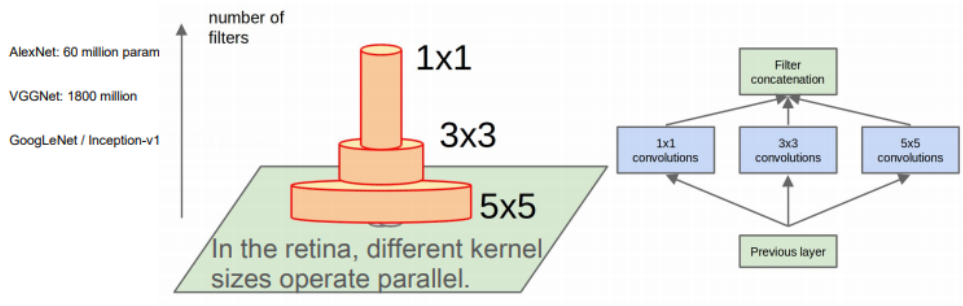
Google, Christian Szegedy

2014 with a top 5 error rate of 6.7%

This can be thought of as a "pooling of features" because we are reducing the depth of the volume, similar to how we reduce the dimensions of height and width with normal maxpooling layers.

Idea:
Not to introduce different size kernels in different layers, but introduce 1x1, 3x3, 5x5 in each layers, and let the Neural Net figure out, what representation is the most useful, and use that!

Parallel multi-scale approach.



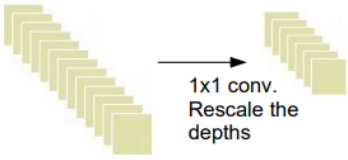
Rethinking Inception

Squeezing the number of channels for each kernel

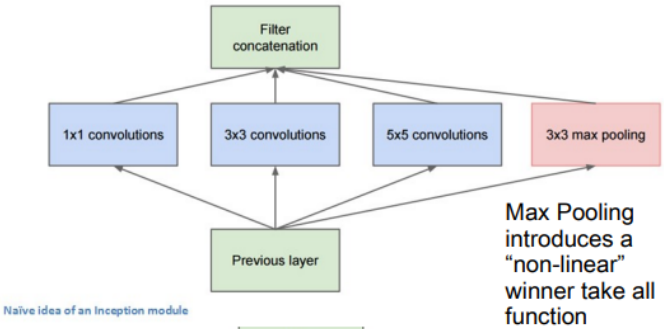
With the concatenations, the number of features increased in each layers, which introduced too many convolution.

To reduce these numbers, they introduced the 1x1 layer.

It can generate e.g. 16 feature maps from 64 feature maps

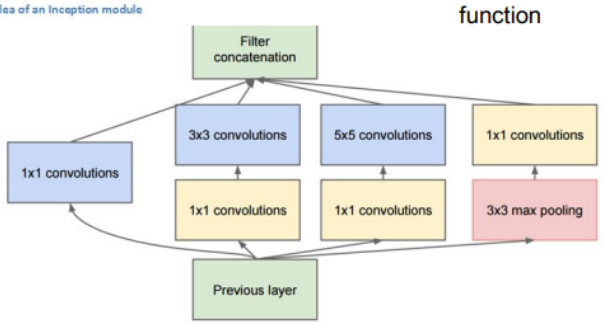


1x1 conv.
Rescale the depths



Naïve idea of an Inception module

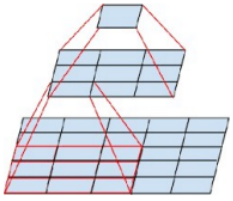
Max Pooling introduces a "non-linear" winner take all function



Rethinking Inception

Larger (5x5) convolutions were substituted by series of 3x3 convolutions

Advantages:
1. Reduction of number of parameters,
2. Additional non-linearities (RELUs) can be introduced



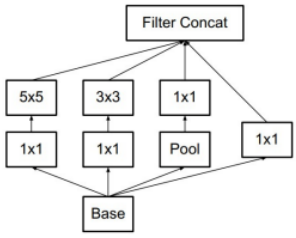
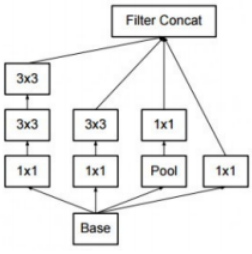
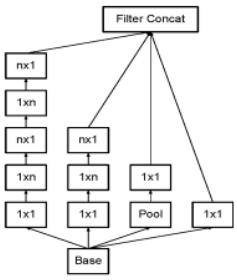


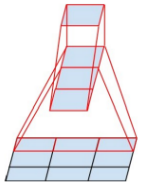
Figure 4. Original Inception module as described in [20].





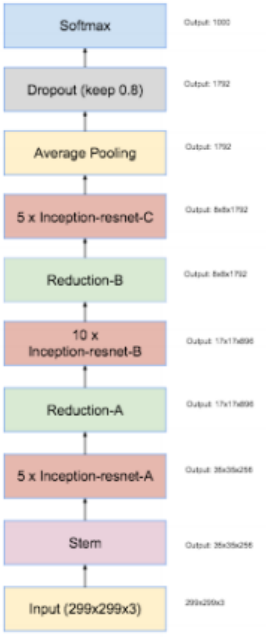
2D convolution were substituted by two 1D convolutions

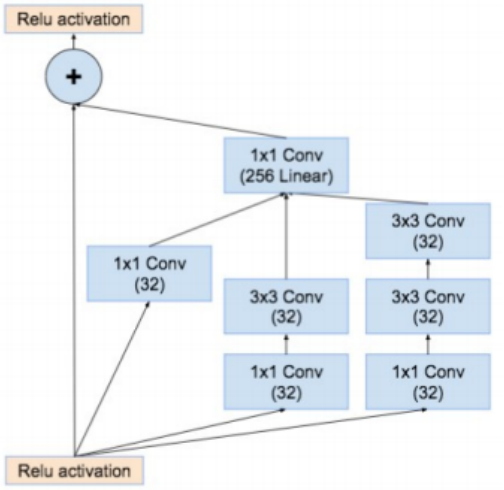
AlexNet: 60 million parameters
VGGNet :180 million parameters
GoogLeNet / Inception-v3: 7 million parameters



GoogleNet Inception v4

Inception architecture applied to residual networks





41

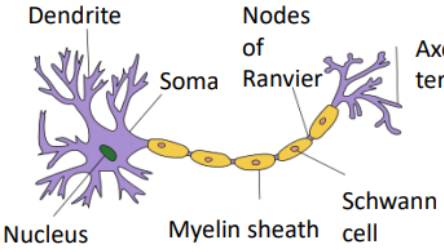
No. 4 Topic

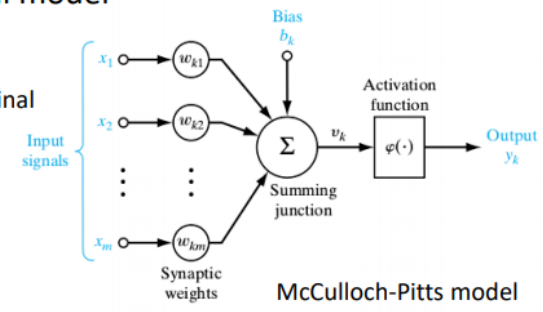
a. McCulloch-Pitts model

Artificial neuron model, 40's (McCulloch-Pitts, J. von Neumann);

The artificial neuron (McCulloch-Pitts)

- The artificial neuron is an information processing unit that is basic constructing element of an artificial neural network.
- Extracted from the biological model

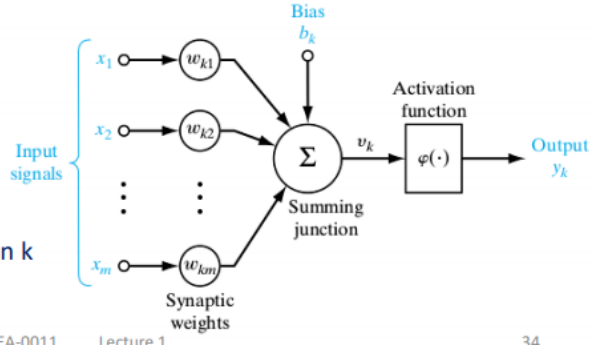




McCulloch-Pitts model

- Receives input through its synapsis (x_i)
- Synapsis are weighted (w_i)
  - if $w_i > 0$: amplified input from that source (excitatory input)
  - if $w_i < 0$: attenuated input from that source (inhibitory input)
- A *b* value biases the sum to enable asymmetric behavior
- A weighted sum is calculated
- Activation function shapes the output signal

x_i : input vector
w_{ki} : weight coefficient vector of neuron k
b_k : bias value of neuron k
o_k : output value of neuron k

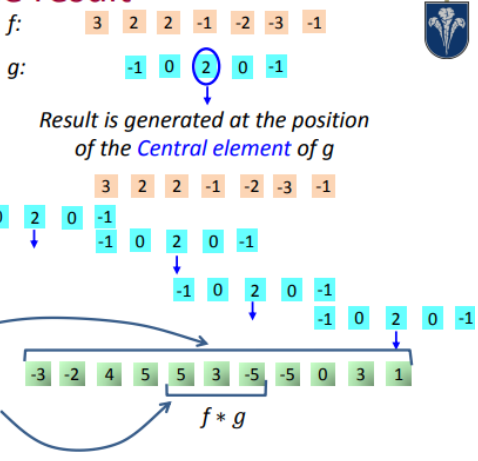


---

b. Parameters of conv - filter, stride, padding, etc

Size of the result

- In practice, convolution is used as a *filter*, where
  - *f* is the measrurement data, *g* is the filter function descriptor (*kernel*)
  - size(*f*) ≫ size(*g*)



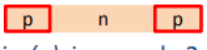
size(*f*)= n size(*g*)= *k* n≥*k*

$$\text{size}(f * g) = \begin{cases} n + k - 1 & (\textit{if all values counted}) \end{cases}$$

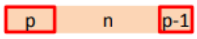
In CNN, we calculate the valid values only!

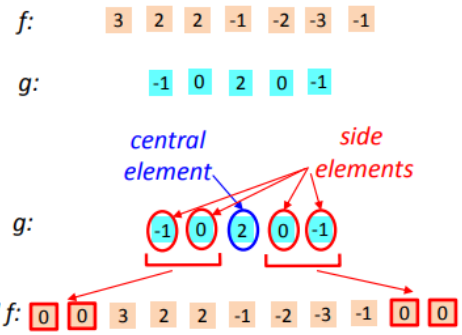
Padding in 1D

- In many case, we use a sequence of convolution filters on the measured data blocks
- We do not want size changes on the data blocks
- To avoid size changes, we have to pad the data block with zeros at the boundaries
  - *k=size(g)* is odd: k=2p+1



  - *k=size(g)* is even: k=2p
    - Padding is asymmetric:

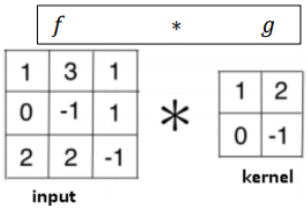




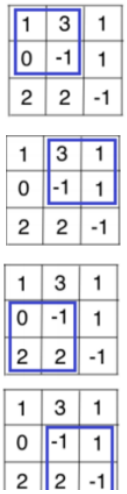
Valid $f * g$ after padding:

24

2D convolution

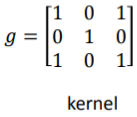


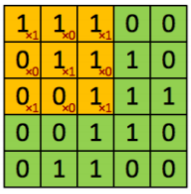
Scanning through the *f* function with the flipped *g* function

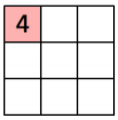


10/22/2019

27



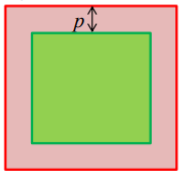




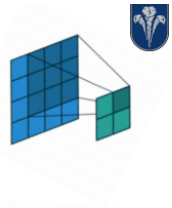
Padding in 2D

Works the same way as in 1D

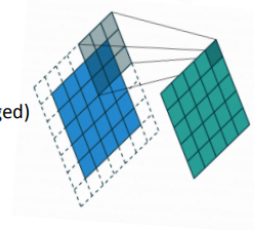
- Boundary layers are added and filled up with zeros
- Size *g* is *k* x *k*,
  - *where: k=2p+1*
- Padding: p layers of zeros



Convolution without padding (valid results)



Convolution with padding (size unchanged)



10/22/2019

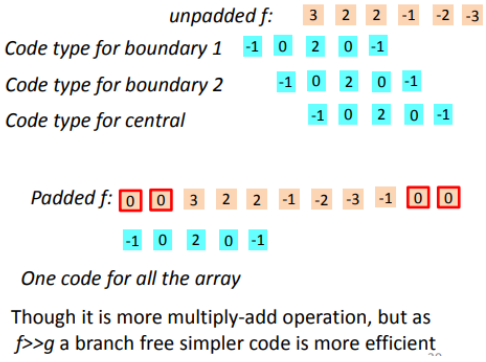
Why use padding:

Simplifies the execution code

No branches

Do not have to deal with the different calculation methods at the boundaries

Same code runs in the entire array



Though it is more multiply-add operation, but as *f*>>*g* a branch free simpler code is more efficient

Parameter number and computational load

- Number of trainable free parameters:
  - k in 1D convolution | size(g)= k
  - k^2 in 2D convolution | size(g)= $k \times k$
- Operation number
  - $k*n$ for a padded 1D convolution | size(f)= n
  - $k^2 * n^2 = O(n^2)$ for a padded 2D convolution | size(f)= $n \times n$

Stride

Stride is the number of pixel what we slide the kernel

- Horizontal stride
- Vertical stride

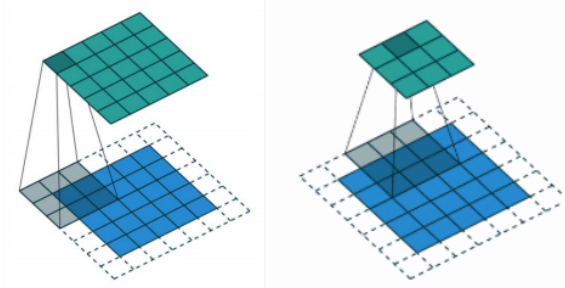
Down sampling the image

- Size:

$$\frac{n+2p-k}{s} + 1$$

- where:

size(f)= n, size(g)= k,
p: padding, s: stride



Padding:1, stride: 1 Padding:1, stride: 2

10/22/2019

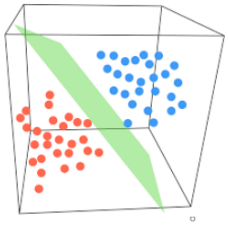
---

c. Linear classifier, margin of the classifier

Linear separability

- Today, we assume that the IO sets are linearly separable
- The decision boundary is a hyperplane defined:

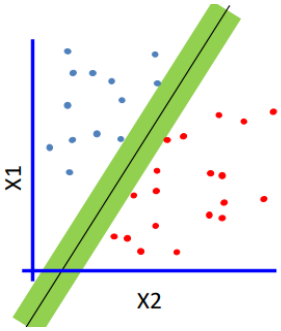
$$\mathbf{w}^T\mathbf{x} = 0$$



• Positive side of the hyperplane is classified: +1 (yes)

• Negative side of the hyperplane is classified : 0 (no).

9/17/2019 P-ITEEA-0011 Lecture 2



Maximum Margin:
Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a data point.

The learning algorithm: Datasets

- Training set
  - *Set of input – desired output pairs*
  - *Will be used for training*

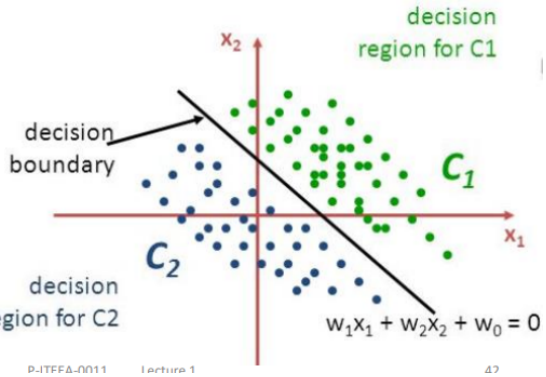
$$X^+ = \{\mathbf{x} : d = +1\}$$
$$X^- = \{\mathbf{x} : d = 0\}$$

- Test set
  - Used, when we have large set of input vectors (not used today)
  - *Set of input – desired output pairs*
  - *Will be used for testing and scoring the result*
- We assumed that X^+ and X^- must be linearly separable
- We are looking for an optimal parameter set:

$$X^+ = \{\mathbf{x} : \mathbf{w}_{opt}^T \mathbf{x} > 0\},$$
$$X^- = \{\mathbf{x} : \mathbf{w}_{opt}^T \mathbf{x} < 0\}.$$

Elementary set separation by a single neuron (2)

- in a 2-D input space, the hyper plane is a straight line.
- Above the line is classified: +1 (C1: yes)
- Below the line is classified : 0 (C2: no)



d. Data augmentation

Regularization and optimization methods

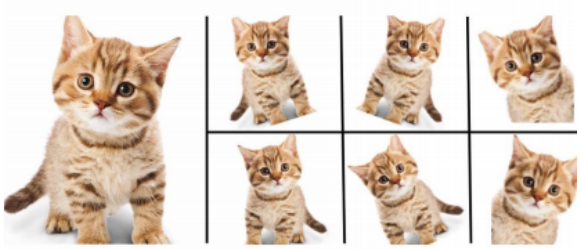
Different methods to increase the loss in the learning phase, but reduce overfitting and increase generalization capabilities

- Local response normalization
- Batch normalization
- Data augmentation (Enriching the data set)
- Early stopping
- Ensemble methods
  - Network duplication
  - Bagging
  - Dropout

Ian Goodfellow: regularization is *"any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error"*

Data augmentation

- **Idea:**
  - *Increase the generalization capability of the net by enlarging the training set*
- Increase the number of the training vector by introducing fake (artificial) input-output pairs
- Typical methods
  - Translating
  - Slight rotation
  - Rescaling
  - Adding noise
  - Flipping
  - Cutting out parts
  - Manipulating with pixel values



Input normalization and Data augmentation I

Images were down-sampled and cropped to 256×256 pixels and normalized

- 1st : image translations and horizontal reflections
  - random 224x224 patches + horizontal reflections from the 256x256 images
  - Testing: five 224x224 patches + horizontal reflections → averaging the predictions over the ten patches

Data augmentation II

- 2nd : change the intensity of RGB channels
  - PCA on the set of RGB pixel values throughout the ImageNet training set
  - To each RGB image pixel $I_{xy} = [I_{xy}^R, I_{xy}^G, I_{xy}^B]$ following is added

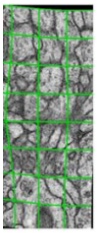
$$[p_1, p_2, p_3][\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3]^T \quad |\alpha_i \sim N(0, 0.1)$$

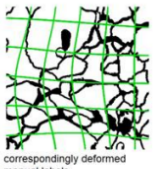
  - Improvement:
    - top-1 error rate by 1%

eigenvalues

eigenvectors

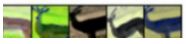
U-Net





correspondingly deformed manual labels

no shift, no extrapolation)



- Designed for biomedical image processing: cell segmentation
- Data augmentation via applying elastic deformations,
  - Natural since deformation is a common variation of tissue
  - Smaller dataset is enough

Alexnet

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012)

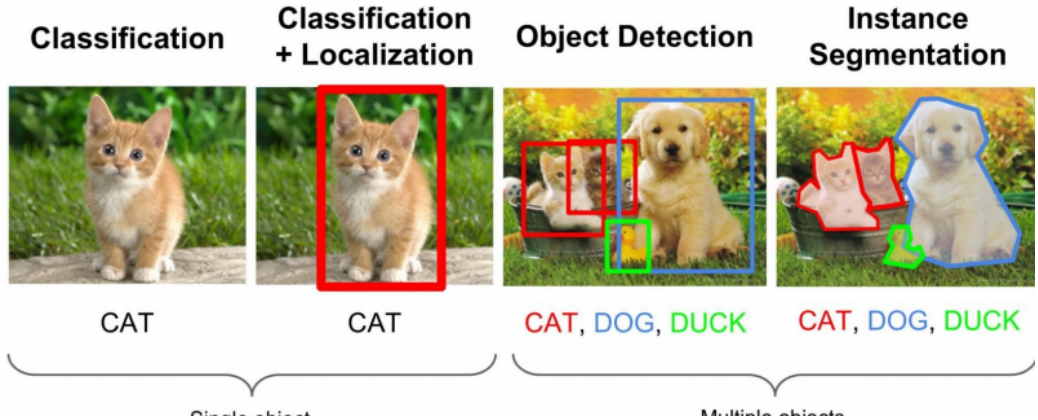
Trained whole ImageNet (15 million,22,000 categories)

Used data augmentation (image translations, horizontal reflections, and patch extractions)

---

e. YOLO = you only look once (2016 May)

-> Object detection as regression



Single objects, Multiple objects

YOLO, Detectnet

Models detection as a regression problem:

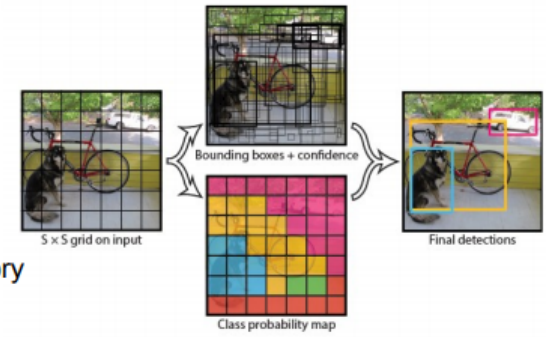
Divide the image into a grid and each cell can vote for the bounding box position of possible object. (Four output per cell for the corner positions.)

Boxes can have arbitrary sizes

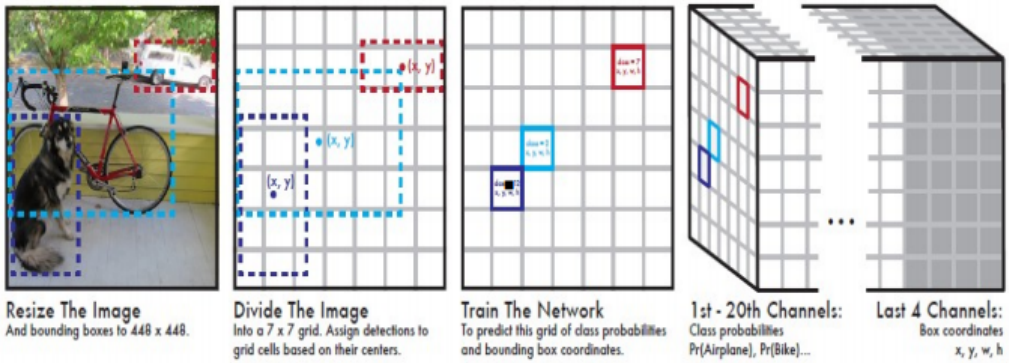
Each cell can proposes a bounding box one category (more layers, more categories per position).

Non-suppression on the boxes

No need for scale search, the image is processed once and objects in different scales can be detected

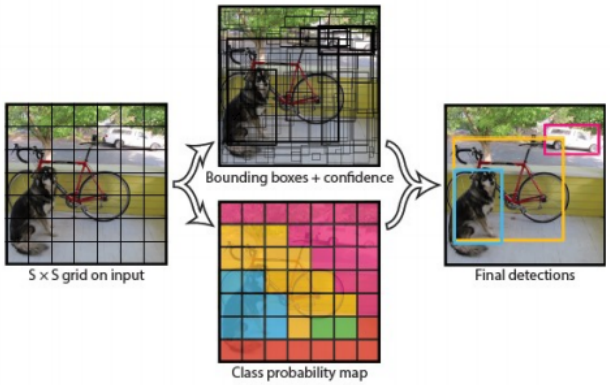


Handles oclusion



Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition

How unified detection works?



confidence scores: reflect how confident is that the box contains an object+how accurate the box is .

$$\Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

conditional class probabilities: conditioned on the grid cell containing an object

$$\Pr(\text{Class}_i|\text{Object})$$

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

- At test time, multiply the conditional class probabilities and the individual box confidence predictions
- giving class-specific confidence scores for each box
- Showing both the probability of that class appearing in the box and how well the predicted box fits the object

Pixel level segmentation

The expected output of the network is not a class, but a map representing the pixels belonging to a certain class.

Creation of a labeled dataset (handmade pixel level mask) is a tedious task

More complex architectures are needed (compared to classification)

Popular architectures (Sharpmask, U-NET ...)

SharpMask: Learning to Refine Object Segments. Pedro O. Pinheiro, Tsung-Yi Lin, Ronan Collobert, Piotr Dollár (ECCV 2016)

SEMANTIC IMAGE SEGMENTATION WITH DEEP CONVOLUTIONAL NETS AND FULLY CONNECTED CRFS Liang-Chieh Chen et al. ICLR 2015

No. 5 Topic

a. Statistical learning theory

Statistical learning theory

- Empirical error

$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

- Theoretical error

$$\left\|\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right\|^2 = \int_X \ldots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right)^2 dx_1 \ldots dx_N$$

- Let us have $\mathbf{x}_k$ random variables subject to uniform distribution

$\mathbf{x}_k$ random variable, where d=F($\mathbf{x}$)

$$\lim_{k\to\infty} = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2 = \mathrm{E}\left(d - Net(\mathbf{x}, \mathbf{w})\right)^2 =$$

$$\int_X \ldots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right)^2 p(\mathbf{x}) dx_1 \ldots dx_N =$$

Because it is ~ constant due to the uniformity

$$\frac{1}{|X|}\int_X \ldots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right)^2 dx_1 \ldots dx_N \square$$

$$\int_X \ldots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right)^2 dx_1 \ldots dx_N$$

Therefore

$$\mathrm{l.i.m.}_{K\to\infty} \mathbf{w}_{\mathrm{opt}} = \mathbf{w}_{\mathrm{opt}}^{(K)}$$

Where l.i.m. means: lim in mean

$$\lim_{K\to\infty} R_{emp}(\mathbf{w}) = R_{th}(\mathbf{w})$$

$$\lim_{K\to\infty} \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2 = \int_X \ldots \int \left(\mathrm{F}(\mathbf{x}) - \mathrm{Net}(\mathbf{x}, \mathbf{w})\right)^2 dx_1 \ldots dx_N$$

Weak learning is satifactory!

Ez mar nem is ide tartozik, de mindu:

Learning – in practice

- Learning based on the training set:

$$\tau^{(K)} = \left\{(\mathbf{x}_k, d_k); k = 1, \ldots, K\right\}$$

- Minimize the empirical error function (R_{emp})

$$\mathbf{w}_{\mathrm{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K}\sum_{k=1}^{K}\underbrace{\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2}_{E_k} = \min_{\mathbf{w}} R_{emp}(\mathbf{w})$$

- Learning is a multivariate optimization task

b. Various activation functions and their properties

- Activation function shapes the output signal
- https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

Activation function: φ(.)

- Always a nonlinear function
- Typically it clamps the output (introduces boundaries)
- Monotonic increasing function
- Differentiable
  - Important from theoretical point of view
- Or at least continuous (except in simplified cases)
  - Sophisticated training algorithms require continuity

Activation function

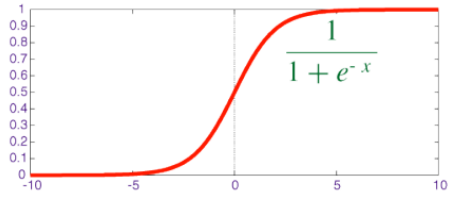
- Originally step function
- Sigmoid or Tanh or their piece-wise linear approximation is used nowadays
- Sophisticated training algorithms require differentiable or at least continuous functions

Sigmoid (or logistic) function is a widely activation function

$$y = \varphi(u) = \frac{1}{1+e^{-\lambda u}}$$

- where

$$u = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

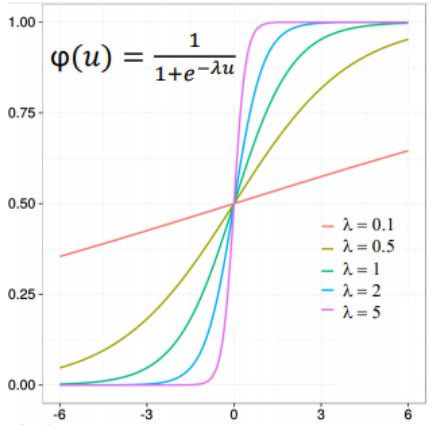


Sigmoid function $S(x) = \frac{1}{1+e^{-x}}$

- Continuous
- Continuously differentiable
- It is used in the output layer of the fully connected neural network

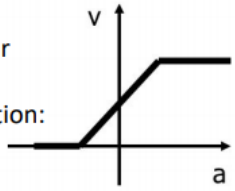
soft nonlinearity

(continuously differentiable)



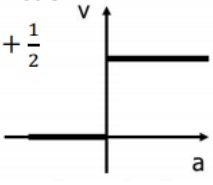
hard nonlinearity

piece-wise linear implementation of sigmoid function:



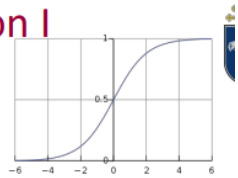
Step (threshold)function

$\lambda \longrightarrow \infty \quad \varphi(u) = \frac{sign(u)}{2} + \frac{1}{2}$



Derivative of sigmoid function I

$$Sigmoid(x) = \frac{1}{1+e^{-x}} \qquad \frac{d}{dx}S(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$$



quotient rule:

$$\frac{d}{dx}f = \frac{(denominator * \frac{d}{dx}numerator) - (numerator * \frac{d}{dx}denominator)}{denominator^2}$$

$$\frac{d}{dx}S(x) = \frac{(1+e^{-x})(0) - (1)(-e^{-x})}{(1+e^{-x})^2}$$

$$\frac{d}{dx}S(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

This is the correct result, but it is not in a nice form.

Derivative of sigmoid function II

$$\frac{d}{dx}S(x) = \frac{e^{-x}}{(1+e^{-x})^2} \Rightarrow \frac{d}{dx}S(x) = \frac{1-1+e^{-x}}{(1+e^{-x})^2}$$

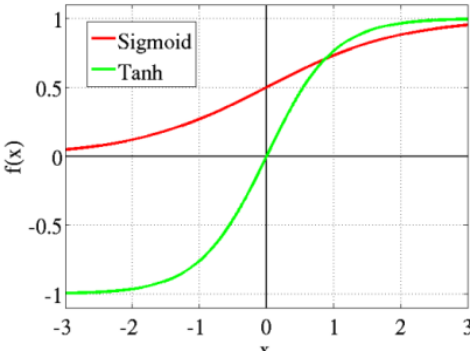
$$\frac{d}{dx}S(x) = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$ reduction

$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2}$$ Multiply out

$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})}(1 - \frac{1}{1+e^{-x}}) \qquad Sigmoid(x) = \frac{1}{1+e^{-x}}$$

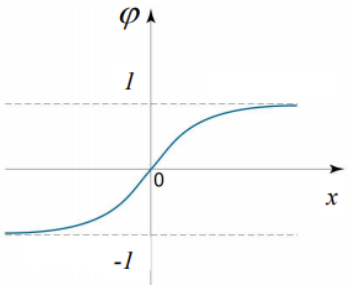
$$\frac{d}{dx}S(x) = S(x)(1-S(x))$$ Much nicer form!

- Bipolar activation function: tanh
- Continuously differentiable
- Monotonic
- Useful, when bipolar output is expected
- Hard approximations:
  - Piece-wise
  - Step-wise



Activation function II

- Hyperbolic tangent function
  - Continuous
  - Continuously differentiable
  - It is used in the output layer of the fully connected neural network



$$\varphi(x) = \tanh(x)$$

$$\frac{d}{dx}\varphi(x) = 1 - \tanh^2(x) = (1-\tanh(x))\,(1+\tanh(x))$$

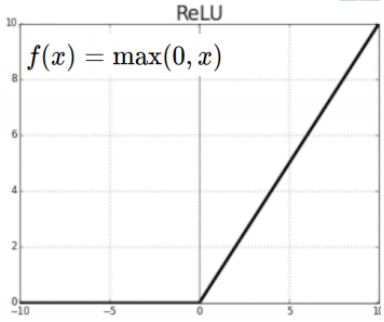
Activation function III

- Rectified Linear Unit (ReLU)

$$f(x) = \max(0, x)$$

  - Most commonly used nonlinearity in hidden layers of deep neural networks
- Derivative of ReLU

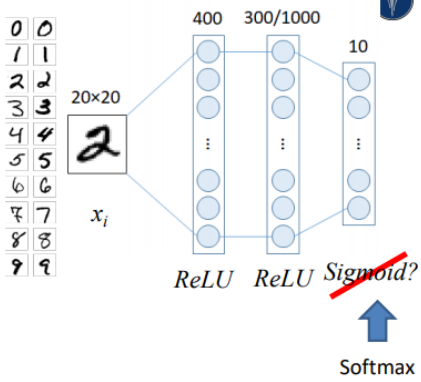
$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



+ other ReLU in No.2 Topic

Probabilistic decision (n discrete categories)

- Assume we have annotated input vectors with *n* different classes (MNIST data base)
- Expect a probability distribution on the output layer!
  - $0 \leq y_i \leq 1$ sigmoid OK!
  - $\sum_{i=1}^{n} y_i = 1$ sigmoid NOT OK!

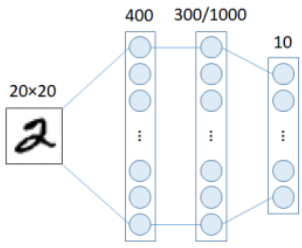


Softmax

- Mathematically:
  - Normalized exponential functions of the output units
- Probability distribution of n discrete classes:
  - One-of-n classes problems
  - $0 \leq y_i \leq 1$
  - $\sum_{i=1}^{n} y_i = 1$
- Architectural difference:
  - Previously learned activation functions were based on the inputs of one neuron
  - Softmax combines a layer of output neurons

$$y_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}$$

$$\boldsymbol{v} = \boldsymbol{w}^T \boldsymbol{x}$$



Properties of Softmax

Generalization of sigmoid function for one-of-n class

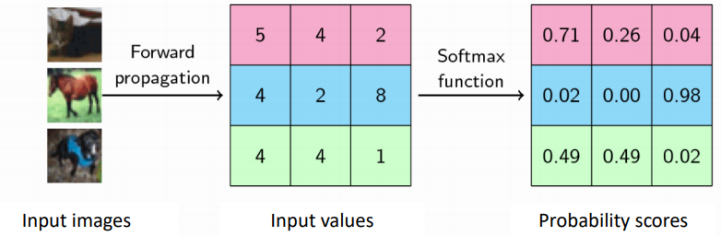
Squashes a vector of size n between 0 and 1

Improves the interpretability of the output of a Neural Net

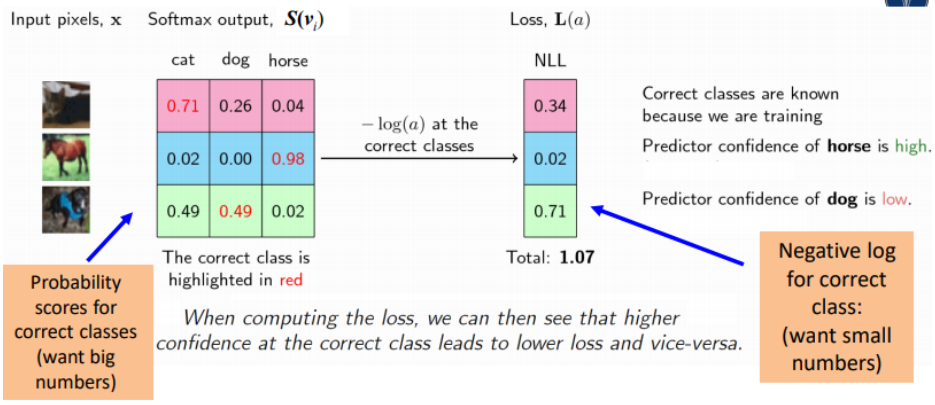
Describes the probability distribution of a certain class
  - We may use the word ”confidence”

Winner take all
  - exponential function strongly penalize the non-winners
  - Similar to lateral negative feedback in the natural neural systems

EXAMPLE



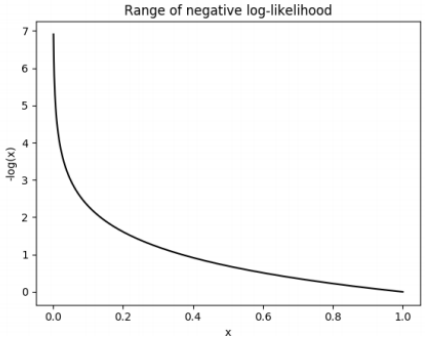
Example



Loss function for softmax: Negative log-likelihood

- $L(\boldsymbol{y}) = \sum_{k=1}^{K} -\log(y)$
- The negative logarithm of the probability of the correct decision classes are summed up
- It is small, if the confidence of a good decision was high for a certain class
- Large, when the confidence is low
- Partial derivative of a softmax layer with negative log-likelihood:

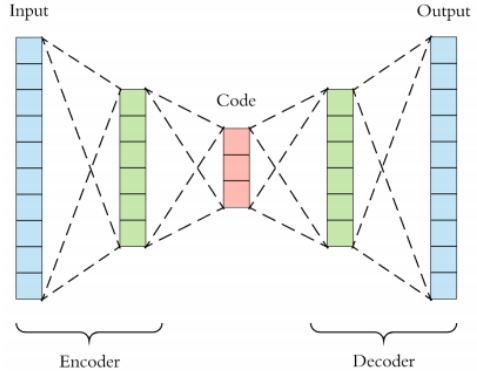
$$\frac{\partial C}{\partial v_j} = y_j - 1$$



---

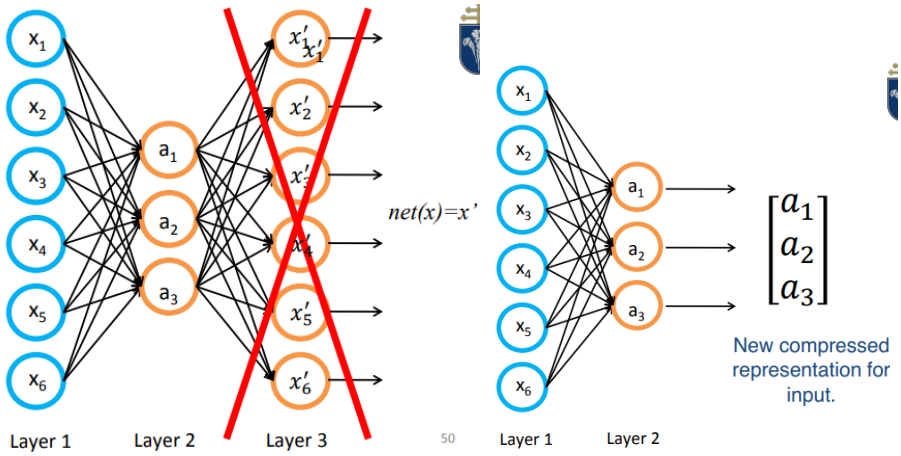
c. Autoencoders

- Neural network used for efficient data coding
- Uses the same vector for the input and the output
  - No labelled data set is needed
  - Unsupervised learning
- Two parts
  - Encoder: reduces data dimension
  - Decoder: reconstructs data
  - Middle layer: code

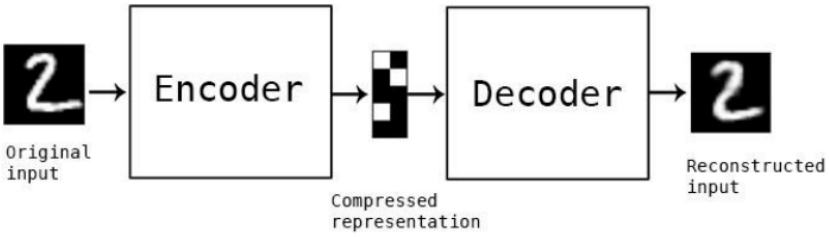


Operation

- The network is trained with the same input-output pairs
- Loss function:
  - MSE
  - Cross Entropy
- After network is trained, remove decoder part



Example



- Coding MNIST data base
- 28x28 (784 dimensions) → 2x5 (10 dimensions)
- 78 times compression

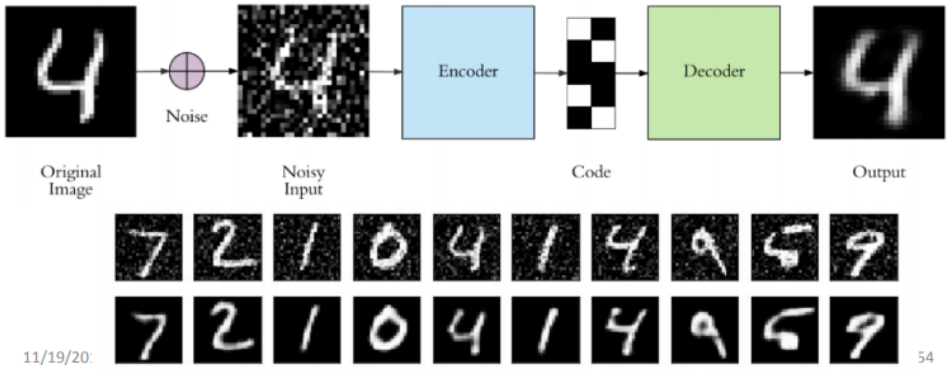
Autoencoder vs PCA

- Undercomplete autoencoder with
  - one hidden layer
  - linear output function
  - MSE loss
- Projects data on subspace of first K principal components

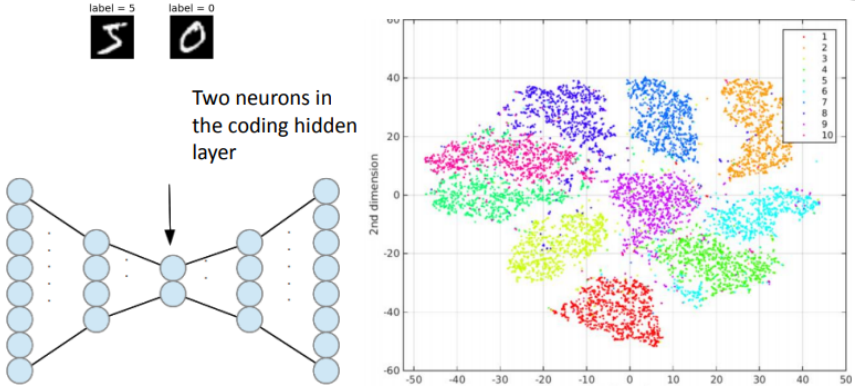
Undercomplete: width (dimension) of hidden layer is smaller than width input/output layer

Denoising

- Trick:
  - Adding noise to the input
  - The desired output is the original input



Autoencoder + t-SNE



---

d. Graph unrolling and parameter sharing in recurrent NN

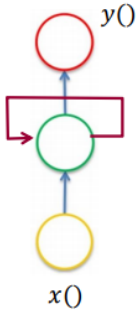
Még nézd át az RNN-t külön, mert ez homály

Introducing feedback loop

$$h(0) = \begin{bmatrix} h_1(0) \\ \cdot \\ \cdot \\ h_l(0) \end{bmatrix} \qquad x(1) = \begin{bmatrix} x_1(1) \\ \cdot \\ \cdot \\ x_k(1) \end{bmatrix}$$

concatenation →

$$c(1) = \begin{bmatrix} h_1(0) \\ \cdot \\ h_l(0) \\ x_1(1) \\ \cdot \\ x_k(0) \end{bmatrix}$$



$$h(1) = f\big(h(0), x(1)\big) = \mathbf{W_h} c(1)$$

w: $l \times (k + l)$ *sized weight matrix*

f() can be defined as a more complex function not only a matrix vector multiplication.

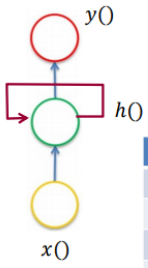
$$h(0) = 0$$

Timing of the RNN

- Discrete time steps are used
- Input vector sequence to apply
- Signals are calculated in a node, when all inputs exist
- State machine

Time	Input	State	output
t=1	$x(1)$	$h(1) = f\big(h(0), x(1)\big)$	$y(1) = g\big(h(1)\big)$
t=2	$x(2)$	$h(2) = f\big(h(1), x(2)\big)$	$y(2) = g\big(h(2)\big)$
t=3	$x(3)$	$h(3) = f\big(h(2), x(3)\big)$	$y(3) = g\big(h(3)\big)$
t=4	$x(4)$	$h(4) = f\big(h(3), x(4)\big)$	$y(4) = g\big(h(4)\big)$
		. . .	

How to calculate back propagation?

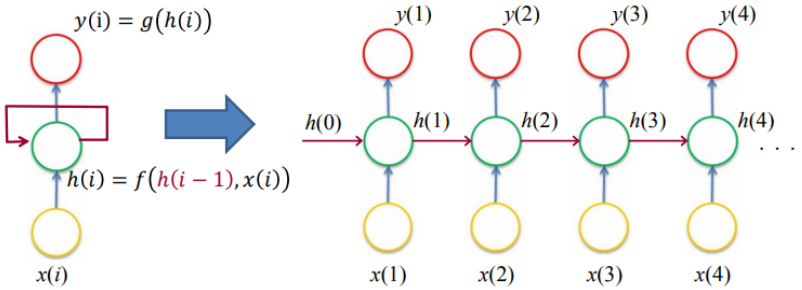


$h(0) = 0$

input
$x(1)$
$x(2)$
$x(3)$
$x(4)$
…

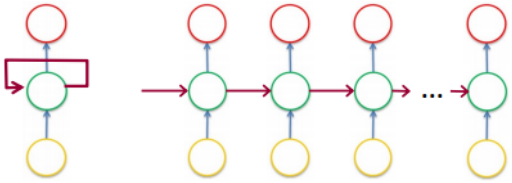
11/19/2019

Unrolling



Unrolling

- Unrolling generates an acyclic directed graph from the original cyclic directed graph structure
- It generates a final impulse response (FIR) filter from the original infinite impulse response (IIR) filter
- Dynamic behavior

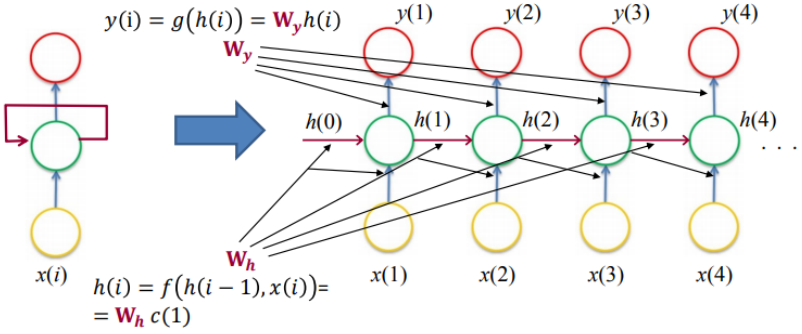


IIR filters may response to any finite length input with a infinite (usually decaying) response, due to their internal loop.

FIR filters response to any finite length input with a final response.

Weight matrix sharing

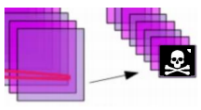
RNN re-uses the same weight matrix in every unrolled steps.



!!! EZ NEM RNN:

Properties of Convolutional Neural Networks II:

- Parameter sharing
  - Same parameters everywhere in the layer
  - Contribution to the gradient of a weight from many positions
  - Reduces the risk of overfitting
  - Reduces the risk of dying RELU (dying cell)
    - When it happens, an entire feature extractor on a layer is dying



EZ STACKOVERFLOW:

Parameter Sharing

Being able to efficiently process sequences of varying length is not the only advantage of parameter sharing. As you said, you can achieve that with padding. The main purpose of parameter sharing is a reduction of the parameters that the model has to learn. This is the whole purpose of using a RNN.

If you would learn a different network for each time step and feed the output of the first model to the second etc. you would end up with a regular feed-forward network. For a number of 20 time steps, you would have 20 models to learn. In Convolutional Nets, parameters are shared by the Convolutional Filters because when we can assume that there are similar interesting patterns in different regions of the picture (for example a simple edge). This drastically reduces the number of parameters we have to learn. Analogously, in sequence learning we can often assume that there are similar patterns at different time steps. Compare `'Yesterday I ate an apple'` and `'I ate an apple yesterday'`. These two sentences mean the same, but the `'I ate an apple'` part occurs on different time steps. By sharing parameters, you only have to learn what that part means once. Otherwise, you'd have to learn it for every time step, where it could occur in your model.

There is a drawback to sharing the parameters. Because our model applies the same transformation to the input at every time step, it now has to learn a transformation that makes sense for all time steps. So, it has to remember, what word came in which time step, i.e. `'chocolate milk'` should not lead to the same hidden and memory state as `'milk chocolate'`. But this drawback is small compared to using a large feed-forward network.

---

e. MobileNet

MobileNet

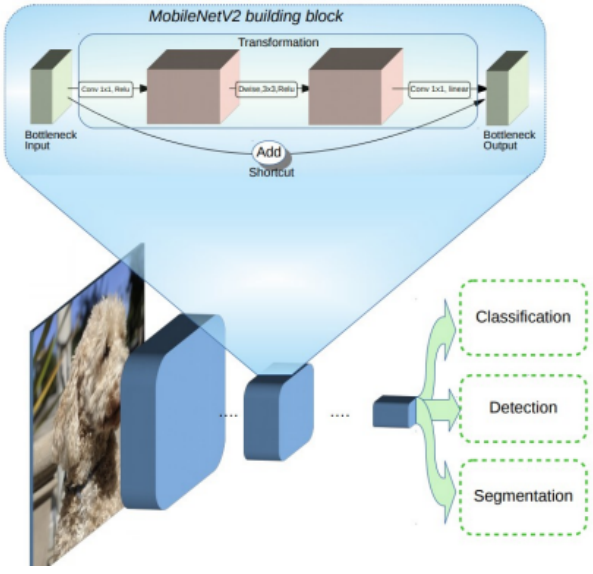
Scaling in feature map depths.

In this arhcitecture feature depths are squeezed before each operation

In a squeezed architecture we will use downscale the 128 feature maps to 16, using a linear combination (1x1 convolution)

After the 3x3 covolutions, we expanded back to 128 layers by 1x1 convolution again

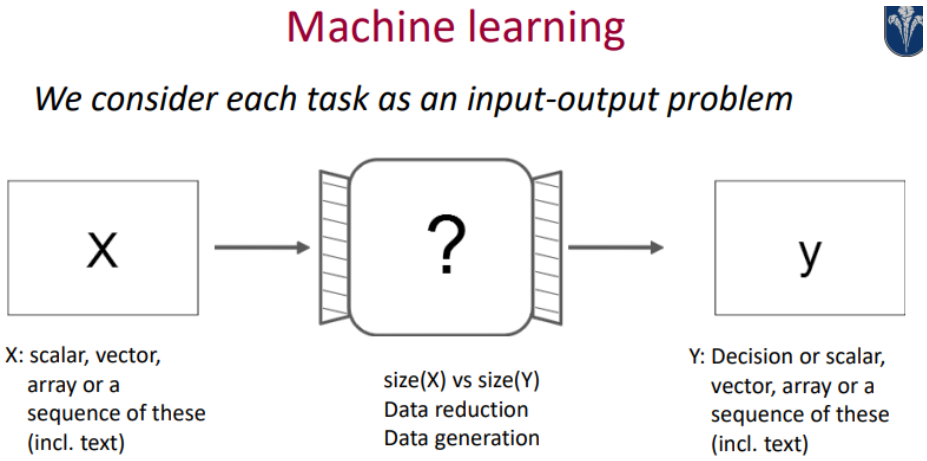
From the linear combination of these elements the new maps are created



No. 6 Topic

a. Machine learning problem definition

Nem vágom ez alatt mit ért…



Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. **Machine learning focuses on the development of computer programs** that can access data and use it learn for themselves.

The process of learning begins with observations or data, such as examples, direct experience, or instruction, in order to look for patterns in data and make better decisions in the future based on the examples that we provide. **The primary aim is to allow the computers learn automatically** without human intervention or assistance and adjust actions accordingly.

Problem Definition Framework

I use a simple framework when defining a new problem to address with machine learning. The framework helps me to quickly understand the elements and motivation for the problem and whether machine learning is suitable or not.

The framework involves answering three questions to varying degrees of thoroughness:

- **Step 1: What is the problem?** Describe the problem informally and formally and list assumptions and similar problems.
- **Step 2: Why does the problem need to be solve?** List your motivation for solving the problem, the benefits a solution provides and how the solution will be used.
- **Step 3: How would I solve the problem?** Describe how the problem would be solved manually to flush domain knowledge.

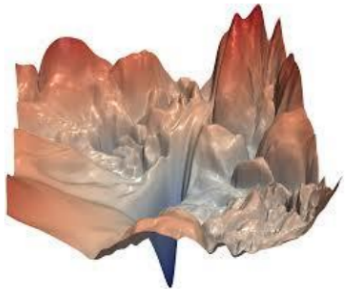
---

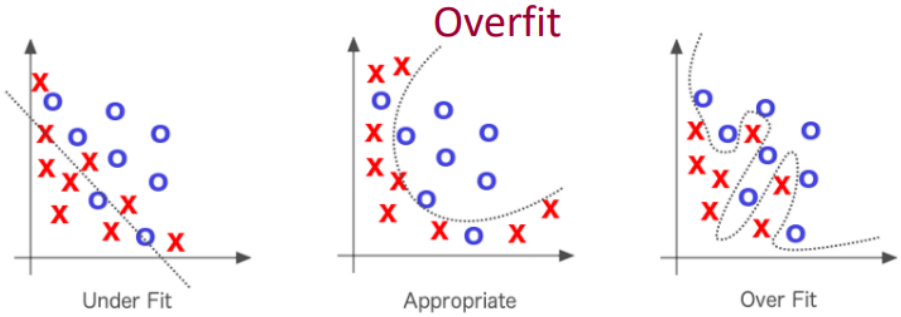
b. Newton optimizer -> See: No. 3 Topic

c. Effects and relationship of model capacity and complexity - overfitting, underfitting

Do we have to reach the global minimum?

- Not really
- Global minimum means: **Overfitting**
- Overfitting: The network exactly learned the training vectors
- However, it loses the generalization capabilities

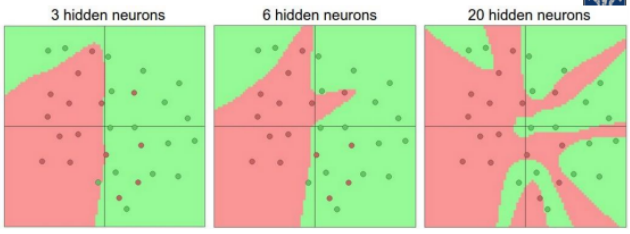




Overfitting occurs when a model with high capacity fits the noise in the data instead of the (assumed) underlying relationship

Losing the generalization capabilities!!!

Network complexity vs. capacity



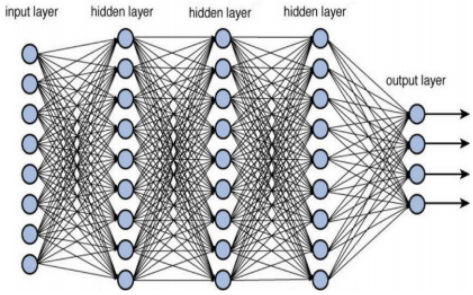
- In general, the more layers we have, and the more neurons there are, the larger the capacity.
- There is no adequate method to predict the required complexity.
- Even if a network is capable to learn a task, it is not guaranteed that it will.

Now we understand

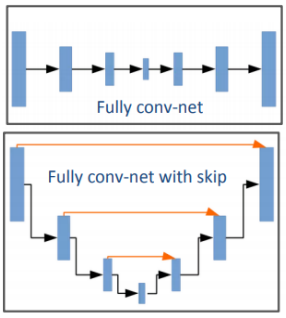
- Architecture of the multilayer fully connected neural networks
- Operation of these networks
- Derivation of the parameters
- Arbitrary function can be approximated if the neural network is complex enough

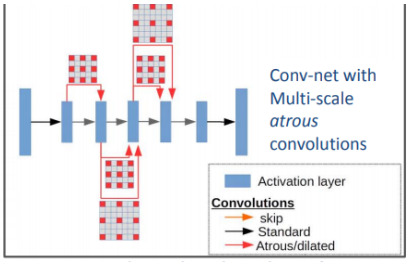
How to increase complexity on a smart way?

10/8/2019.



Semantic segmentation CNN arrangements





- How to solve reduced resolution?
  - Do not downsample !!!
    - Convolution on large images ⇒ Small FOV Enlarge kernel
    - Size $O(n^2)$ more parameters ⇒ getting close to fully
    - Connected layer, slow training, overfitting
  - Atrous Convolution.
    - Large FOV with little parameters → Kill two birds with one stone!

11/12/2019

How could we create deeper networks?

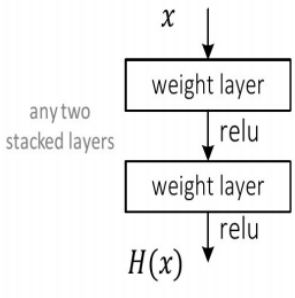
A deeper network always have the potential to perform better, but training becomes difficult

How could we ensure that additional layers will not decrease accuracy (might even increase it)?

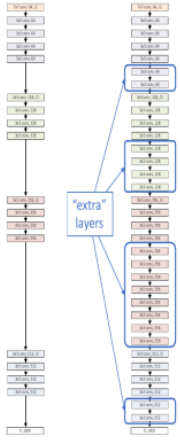
Let's start with a shallow model (18 layers) and add some extra layers (which we hope could increase accuracy)

Our aim is to add "useful" operations H(x)

The problem is that H(x) can ruin our accuracy because vanishing gradients, overfit - extra parameters



a shallower model (18 layers)



Overfitting in Machine Learning

Overfitting refers to a model that models the training data too well.

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns.

For example, decision trees are a nonparametric machine learning algorithm that is very flexible and is subject to overfitting training data. This problem can be addressed by pruning a tree after it has learned in order to remove some of the detail it has picked up.

Underfitting in Machine Learning

Underfitting refers to a model that can neither model the training data nor generalize to new data.

An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data.

Underfitting is often not discussed as it is easy to detect given a good performance metric. The remedy is to move on and try alternate machine learning algorithms. Nevertheless, it does provide a good contrast to the problem of overfitting.

A Good Fit in Machine Learning

Ideally, you want to select a model at the sweet spot between underfitting and overfitting.

This is the goal, but is very difficult to do in practice.

To understand this goal, we can look at the performance of a machine learning algorithm over time as it is learning a training data. We can plot both the skill on the training data and the skill on a test dataset we have held back from the training process.

Over time, as the algorithm learns, the error for the model on the training data goes down and so does the error on the test dataset. If we train for too long, the performance on the training dataset may continue to decrease because the model is overfitting and learning the irrelevant detail and noise in the training dataset. At the same time the error for the test set starts to rise again as the model's ability to generalize decreases.

The sweet spot is the point just before the error on the test dataset starts to increase where the model has good skill on both the training dataset and the unseen test dataset.

You can perform this experiment with your favorite machine learning algorithms. This is often not useful technique in practice, because by choosing the stopping point for training using the skill on the test dataset it means that the testset is no longer "unseen" or a standalone objective measure. Some knowledge (a lot of useful knowledge) about that data has leaked into the training procedure.

There are two additional techniques you can use to help find the sweet spot in practice: resampling methods and a validation dataset.

How To Limit Overfitting

Both overfitting and underfitting can lead to poor model performance. But by far the most common problem in applied machine learning is overfitting.

Overfitting is such a problem because the evaluation of machine learning algorithms on training data is different from the evaluation we actually care the most about, namely how well the algorithm performs on unseen data.

There are two important techniques that you can use when evaluating machine learning algorithms to limit overfitting:

1. Use a resampling technique to estimate model accuracy.
2. Hold back a validation dataset.

The most popular resampling technique is k-fold cross validation. It allows you to train and test your model k-times on different subsets of training data and build up an estimate of the performance of a machine learning model on unseen data.

A validation dataset is simply a subset of your training data that you hold back from your machine learning algorithms until the very end of your project. After you have selected and tuned your machine learning algorithms on your training dataset you can evaluate the learned models on the validation dataset to get a final objective idea of how the models might perform on unseen data.

Using cross validation is a gold standard in applied machine learning for estimating model accuracy on unseen data. If you have the data, using a validation dataset is also an excellent practice.

d. t-distributed Stochastic Neighbor Embedding

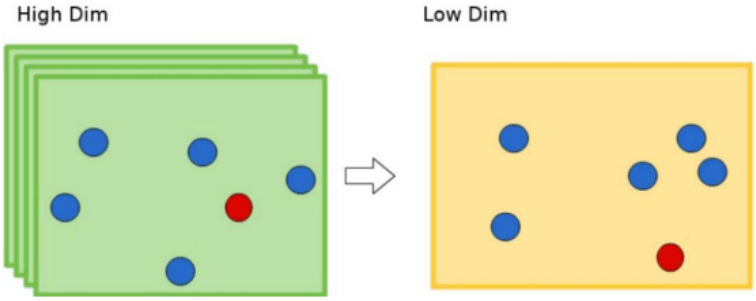
Unsupervised learning technique:

t-Distributed Stochastic Neighbor Embedding (t-SNE)

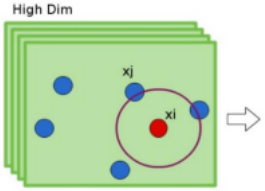
- Introduced by Laurens Van Der Maaten (2008)
- Generates a low dimensional representation of the high dimensional data set iteratively
- Aims to minimize the divergence between two distributions
  - Pairwise similarity of the points in the higher-dimensional space
  - Pairwise similarity of the points in the lower-dimensional space
- Output: original points mapped to a 2D or a 3D data space
  - similar objects are modeled by nearby points and
  - dissimilar objects are modeled by distant points with high probability
- Unlike PCA, it is stochastic (probabilistic)

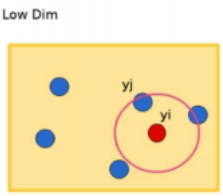
Step 1: *Generate the points in the low dimensional data set (2D or 3D)*

- random initialization
- First two or three components of PCA



Step 2: *Calculate the pair-wise similarities measures between data pairs (probability measure)*





The similarity of datapoint x_j to datapoint x_i means the conditional probability p_{ji} that x_i would pick x_j as its nearest neighbor.

$$p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq l} exp(-||x_l - x_k||^2/2\sigma^2)}$$

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$$

Exponential normalization of the Euclidian distances are needed due to the high dimensionality. (Curse of dimensionality)

Step 3: *Define the cost function*

- Similarity of data points in High dimension: $p_{ij} = \frac{exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq l} exp(-||x_l - x_k||^2/2\sigma^2)}$
- Similarity of data points in Low dimension: $q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l}(1 + ||y_k - y_l||^2)^{-1}}$
- Cost function (called Kullback-Leiber divergence between the two distributions): $C = KL(P||Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$
  - Large p_{ji} modeled by small q_{ji} → Large penalty
  - Large p_{ji} modeled by large q_{ji} → Small penalty
  - Local similarities are preserved

Step 4: *Minimize the cost function using gradient descent*

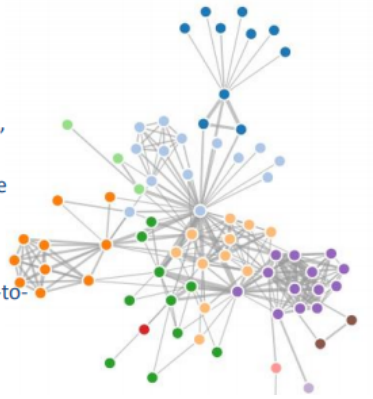
- Gradient has a surprisingly simple form:

$$\frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij})(1 + ||y_i - y_j||^2)^{-1}(y_i - y_j)$$

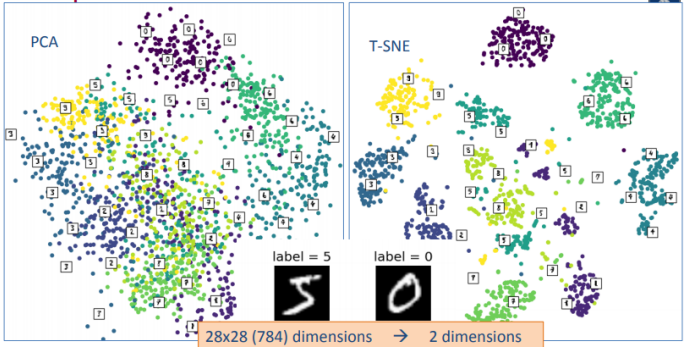
- Optimization can be done using momentum method

Physical analogy

- Our map points are all connected with springs in the low dimensional data map
- Stiffness of the springs depends on $p_{j|i}$ - $q_{j|i}$
- Let the system evolve according to the laws of physics
  - If two map points are far apart while the data points are close, they are attracted together
  - If they are nearby while the data points are dissimilar, they are repelled.
- Illustration (live)
  - https://www.oreilly.com/learning/an-illustrated-introduction-to-the-t-sne-algorithm

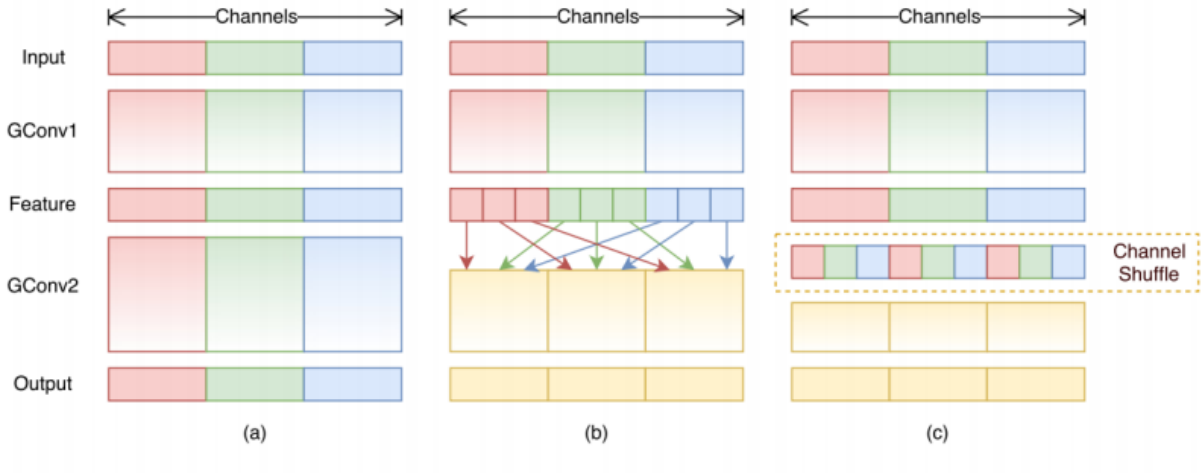


Comparison of PCA and t-SNE on MNIST database



e. ShuffleNet

ShuffleNet



Neten találtam:
We introduce an extremely computation-efficient CNN architecture named ShuffleNet, which is designed specially for mobile devices with very limited computing power (e.g., 10-150 MFLOPs). The new architecture utilizes two new operations, pointwise group convolution and channel shuffle, to greatly reduce computation cost while maintaining accuracy. Experiments on ImageNet classification and MS COCO object detection demonstrate the superior performance of ShuffleNet over other structures, e.g. lower top-1 error (absolute 7.8%) than recent MobileNet on ImageNet classification task, under the computation budget of 40 MFLOPs. On an ARM-based mobile device, ShuffleNet achieves ~13x actual speedup over AlexNet while maintaining comparable accuracy.

Maga a cikk:
https://zpascal.net/cvpr2018/Zhang_ShuffleNet_An_Extremely_CVPR_2018_paper.pdf

...To overcome the side effects brought by group convolutions, we come up with a novel channel **shuffle operation** to help the information flowing across feature channels. Based on the two techniques, we build a highly efficient architecture called **ShuffleNet**. Compared with popular structures like [31, 9, 41], for a given computation complexity budget, our ShuffleNet allows **more feature map channels**, which helps to **encode more information** and is especially critical to the performance of very small networks. We evaluate our models on the challenging ImageNet classification [4, 30] and MS COCO object detection [24] tasks. A series of controlled experiments shows the **effectiveness** of our design principles and the **better performance over other structures.** Compared with the state-of-the-art architecture MobileNet [12], ShuffleNet achieves **superior performance by a significant margin**, e.g. absolute 7.8% lower ImageNet top-1 error at level of 40 MFLOPs. We also examine the speedup on real hardware, i.e. an off-the-shelf ARM-based computing core. The ShuffleNet model achieves **~13× actual speedup over AlexNet** [22] while **maintaining comparable accuracy.**

No. 7 Topic

a. **Credit approval problem** -> na hogy itt mire gondolt a költő????

Neten találtam:

The last decade has seen an important rise of data gathering, especially in the financial sectors. Banks are indeed one of the biggest producers of big data, as a matter of fact no other company than the bank has so much data gathered on its customers. Gathering and analyzing this data is a key feature for decision making, particularly in banking sector. One of the most important and frequent decision banks has to make, is loan approval. The challenge is to know how to build a proactive, powerful, responsible and ethical exploitation of personal data, to make loan applicant proposals more relevant and personalized. Machine learning is a promising solution to deal with this problem. Therefor, in the last years, many algorithms based on machine learning have been proposed to solve loan approval issue. However, these algorithms have not taken into consideration Real-time paradigm during processing. In this paper, we propose a Real-Time Binary classification model to deal with loan approval. Our proposed model is based on a deep neural network, and it permits to classify loan applicant as good or bad risk. Experimental results prove that our proposed Real-Time model, based on deep neural network, outperforms typical binary classifiers, in terms of precision recall and accuracy.

Indiai bácsi a youtube-on:
https://www.youtube.com/watch?v=HNj8dzw3H_E
1. It is used in Distributed Systems
2. This can be divided into Temporal Credit Assignment Problem (Credit or blame to Outcome of internal Decisions) and Structural Credit Assignment Problem (Credit or blame to actions of internal decisions).
3. By these two, we can train the learning machine easily

---

b. **Objective functions in neural networks**

Optimization

- Given an ***Objective function*** to optimize
  - Also called: Error function, Cost function, Loss function, Criterion
  - Derived from the network topology and the input/output pairs
- Function types:
  - Quadratic, in case of regression (stochastic process)

$$R_{emp}(\mathbf{w}) = \frac{1}{K}\sum_{k=1}^{K}\left(d_k - Net(\mathbf{x}_k, \mathbf{w})\right)^2$$

  - Conditional log-likelihood, in case of classification (classification process)
    - The sum of the negative logarithmic likelihood (probability) is minimized

$$\Theta(\mathbf{w}) = \sum_{k=1}^{K} -logP(\mathbf{y_k}|\mathbf{x_k};\mathbf{w})$$

9/30/2019

Optimizations

- *Here we always minimize the objective function*
  - *Parametric equation*
    - **x** *are the variables*
    - **w** *are the parameters*
- *Optimization targets to find the optimal weights*

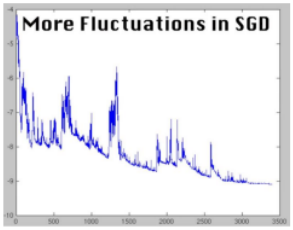
$$\mathbf{w}_{opt} = min\, f(\mathbf{x}, \mathbf{d}, Net(\mathbf{x},\mathbf{w}))$$

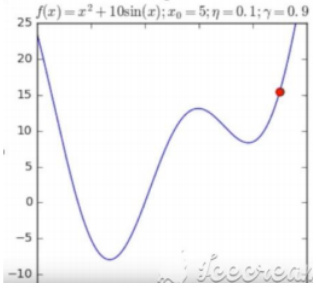
goals:

- *Acceptable error level*
- *Acceptable computational time assuming reasonable computational effort*

As we discussed …

- Stocastic Gradient Descent (SGD) Method
  - Uses a random subset of the training vectors (mini batches)
    - *One update is fast to calcualte*
  - The objective function changes stocastically with the minibatch selection
    - *More fluctuation in the objective function than in case of Gradient Descent*
    - *It helps to come out from local minima and saddles*
  - Decreases the learning rate during the training time to reduce overshoot
  - Still very slow! (Many update steps are needed)
- More advanced optimization methods required!





ez talán már nem is ide tartozik:

Ensemble methods

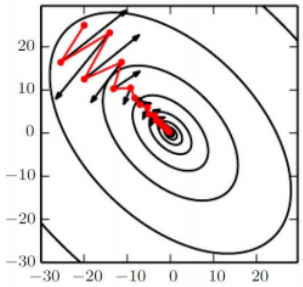
- Idea of ensemble methods:
  - Generate multiple copies of your net
    - Same or slightly modified architectures
  - Train them separately
    - Using different subsets of the training sets
    - Different objective functions
    - Different optimization methods
  - The different trained models have independent error characteristics
  - Averaging the results will lead to smaller error
- Requires more computation and memory both in training and inferencing (testing) phase

---

c. Nesterov momentum optimizer

Momentum I

- Introduced in 1964
- Physical analogy
- The idea is to simulate a unity weight mass
- It flows through on the surface of the error function
- Follows Newton's laws of dynamics
- Having *v* velocity
- Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.



Momentum II: velocity considerations

The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$.
The larger α is relative to ϵ, the more previous gradients affect the current direction.

Terminal velocity is applied when it finds descending gradient permanently:

$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}$$

9/30/2019 37

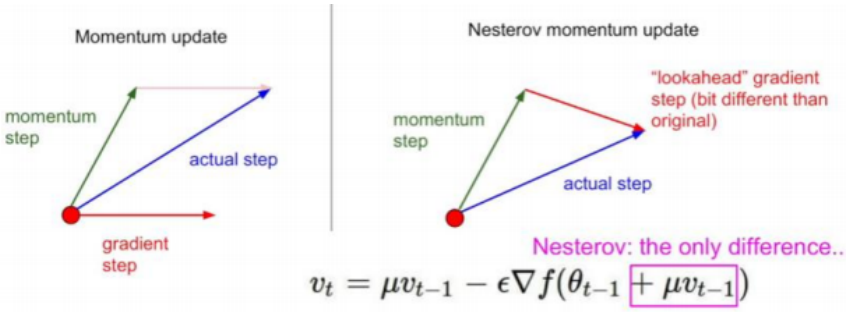
Momentum III

Algorithm Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Nesterov Momentum update:

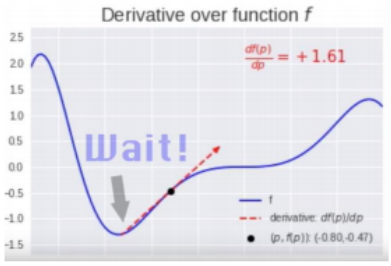
- It calculates the gradient not in the current point, but in the next point, and correct the velocity with the gradient over there (look ahead function)
- It does not runs through a minimum, because if there is a hill behind a minimum, than it starts decreasing the speed in time.



Nesterov: the only difference..

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$



What if we make the learning rate adaptive as well, not just the velocity?

9/30/2019 40

d. Decomposition of convolutional kernels

Decomposition of large kernels I

- Convolution is associative

$$f * (g * h) = (f * g) * h$$

Example:

$$\begin{bmatrix} 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.3 & 0.4 & -0.7 & -1.3 & -0.7 & 0.4 & 0.3 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \end{bmatrix} = \begin{bmatrix} 0.2 & 0.5 & 0.2 \\ 0.5 & -3.1 & 0.5 \\ 0.2 & 0.5 & 0.2 \end{bmatrix} * \begin{bmatrix} 0 & 0.2 & 0.3 & 0.2 & 0 \\ 0.2 & 0.6 & 0.8 & 0.6 & 0.2 \\ 0.3 & 0.8 & 1.2 & 0.8 & 0.3 \\ 0.2 & 0.6 & 0.8 & 0.6 & 0.2 \\ 0 & 0.2 & 0.3 & 0.2 & 0 \end{bmatrix}$$

Laplacian of Gaussian kernel $(g * h)$ — Laplacian (g) — Gaussian kernel (h)

Number of operations: $49*N_{pix}$ $9*N_{pix} + 25*N_{pix} = 34*N_{pix}$

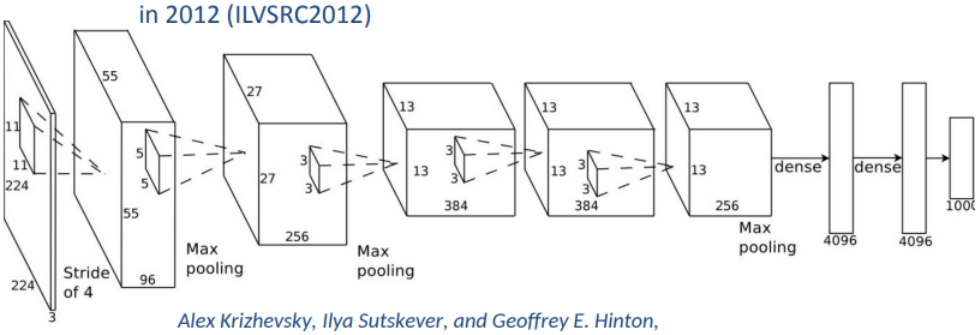
15% reduction of computational demand!!!

- Decomposition is not exact in most cases
  - In general case, it approximates the kernels with a limited accuracy only
- Neural nets does not sensitive for inaccurate decomposition
- Decomposition of larger kernels leads to higher savings!
- Wildly used!

e. Alexnet + ILSVRC

Alexnet: for image classification

- First fully trained deep convolutional neural network
  - Won the ImageNet Large Scale Visual Recognition (ILVSRC) Challenge in 2012 (ILVSRC2012)



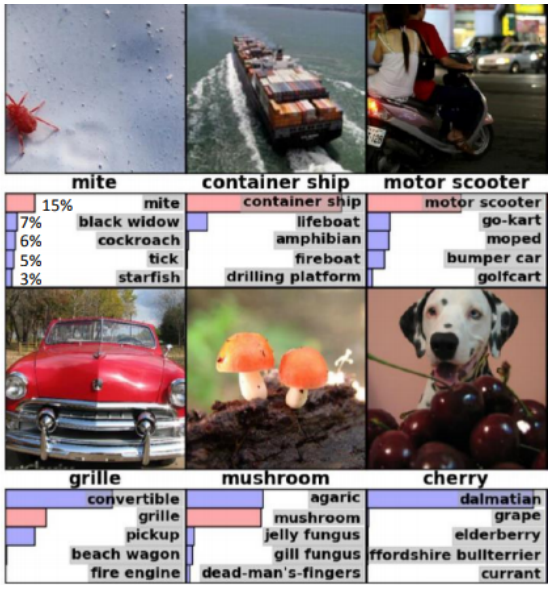
Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, "Imagenet classification with deep convolutional neural networks", Advances in neural information processing systems, 2012

ImageNet Large Scale Visual Recognition Challenge I

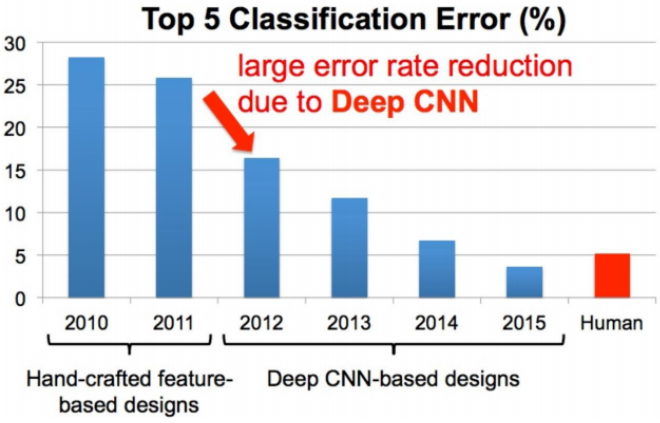
- ImageNet:
  - 15+ million labeled high-resolution images
  - 22000 categories
- ILSVRC uses a subset of ImageNet:
  - 1000 categories
  - ~1000 images per category
  - 1.2 million training images | 50 000 validation images | 150 000 testing images

ImageNet Large Scale Visual Recognition Challenge II

- Each image should be classified
  - Probability distribution
- Top 1 error rate:
  - What percentage was wrongly classified as highest probability? (38,9%)
- Top 5 error rate:
  - What percentage was not in the first five? (18.9%)



ILVSRC results



Alexnet

Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton (2012)

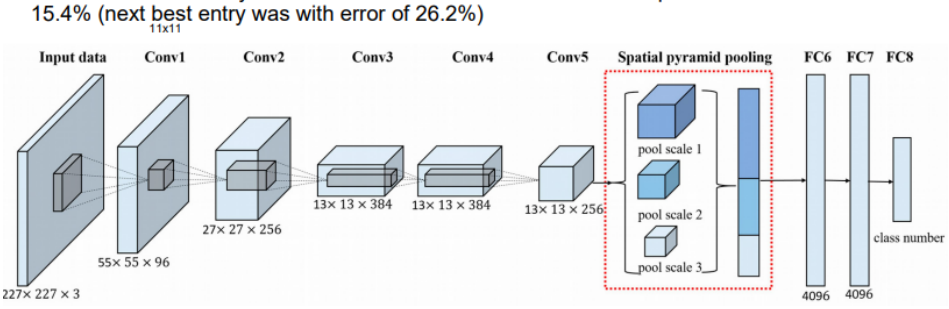
Trained whole ImageNet (15 million,22,000 categories)

Used data augmentation (image translations, horizontal reflections, and patch extractions)

Used ReLU for the nonlinearity functions (Decreased training time compared to tanh) - Trained on two GTX 580 GPUs for six days

Dropout layers

2012 marked the first year where a CNN was used to achieve a top 5 test error rate of 15.4% (next best entry was with error of 26.2%)



No. 8 Topic

a. Delta learning rule

Sequential back propagation

- Adapting the weights of the FFNN (recursive algorithm)

$$w_{ij}^{(l)}(k+1) = w_{ij}^{(l)}(k) + \Delta w_{ij}^{(l)}(k)$$

$$\Delta w_{ij}^{(l)}(k) = ?$$

- The weights are modified towards the differential of the error function (delta rule):

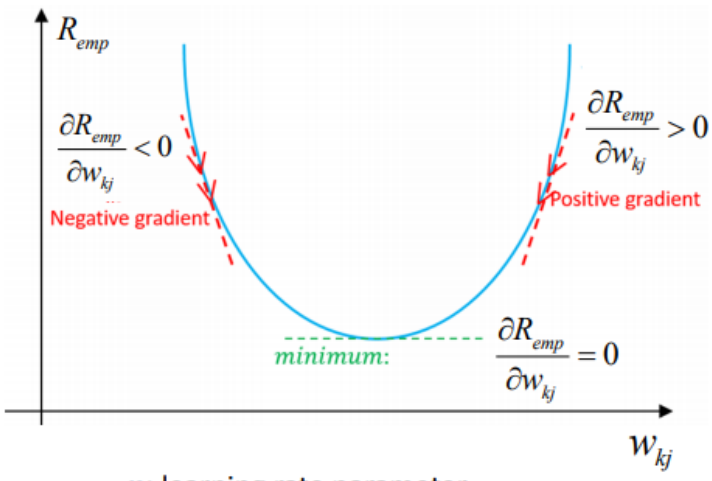
$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial R_{emp}}{\partial w_{ij}^{(l)}}$$

- The elements of the training set adapted by the FFNN sequentially

$$R_{emp} = R_{emp}(y(\mathbf{x}), d)$$

Delta (learning) rule

- If $\frac{\partial R_{emp}}{\partial w_{kj}} < 0$ than we have to increas w_{kj}, to get closer to the minimum.
  - $\Delta w_{kj} = -\eta \frac{\partial R_{emp}}{\partial w_{kj}}$
- If $\frac{\partial R_{emp}}{\partial w_{kj}} > 0$ than we have to decrease w_{kj}, to get closer to the minimum.
  - $\Delta w_{kj} = -\eta \frac{\partial R_{emp}}{\partial w_{kj}}$



η: learning rate parameter

---

b. Batch normalization

- In very deep networks the distribution of the input vectors changes from layer to layer
  - The first layer got normalized input
  - The second layer somewhat shifts and twists on this normalization
  - And it goes on, and the (originally normalized) data propagating trough the layers will be lose its normalized properties (called *„covariance shift"*)
  - This will shift the neuron out of its zero centered position, where the activation function performs well (where the nonlinearity is)
- Solution: **normalization on each layers**!
- It also introduce a noise (loss function increase), which helps to avoid local minima and avoids overfitting

Batch Normalization

- Done on layer level like softmax
- Training:
  - Done on minibatch level
- Inferencing:
  - Do the normalization with the pre-calculated parameters of the entire training set
- Batch normalization is differenciable via chain rule
  - Back propagation can be applied for batch normalized layers
- Rewriting the normalization using probability terms:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

E: *the expectation*
Var: the variance

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β ← bias
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$ ↑ weights

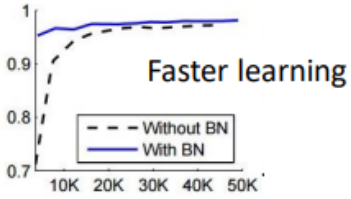
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

ϵ: avoid zero

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

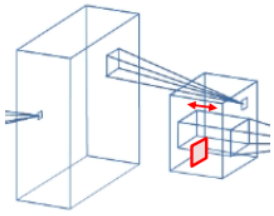


Faster learning

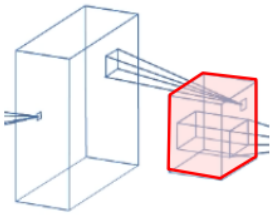
Local response norm. vs batch norm.

Both work within one convolutional layer

- Normalization either through the feature maps or within one feature map
- Normalization is done for one input image



- Normalization done for all the pixels in all the feature maps within a layer
- Normalization is done for the entire batch

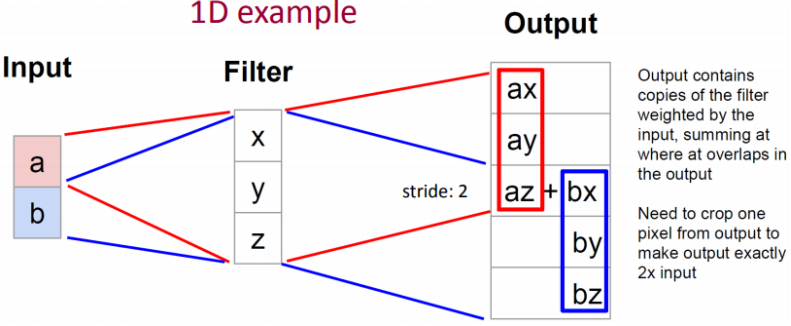


---

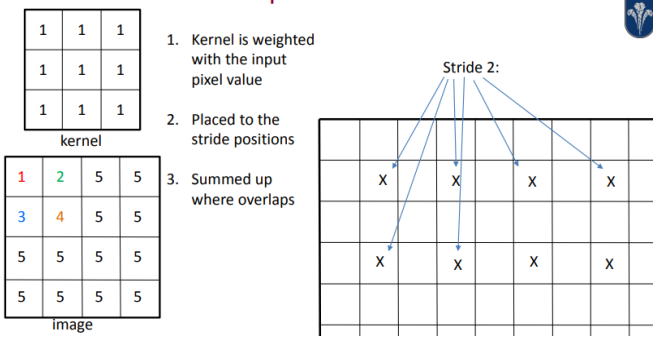
c. Transposed conv., atrous conv.

Upsampling II: "transpose convolution"

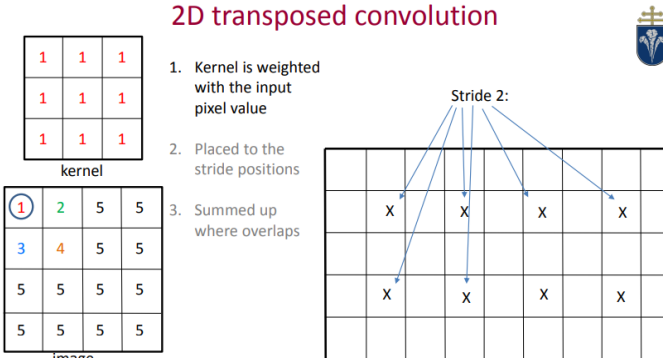
1D example



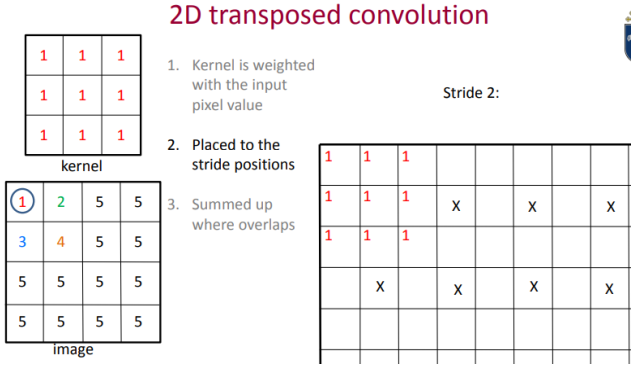
2D transposed convolution



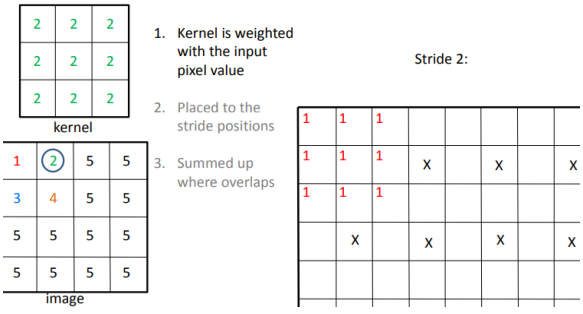
2D transposed convolution



2D transposed convolution

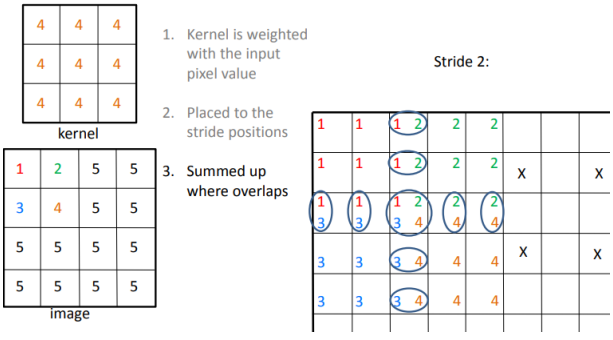


2D transposed convolution

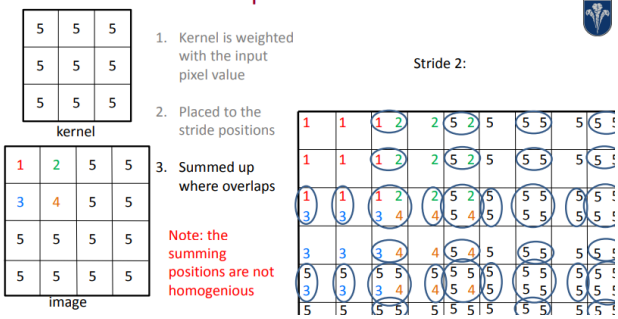


stb.stb. (**3**x3-assal is)

2D transposed convolution



2D transposed convolution

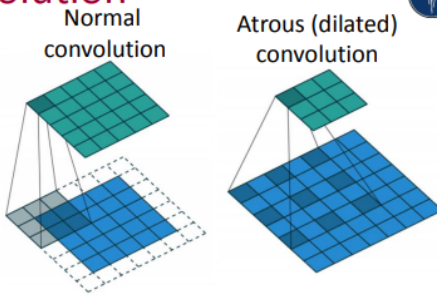


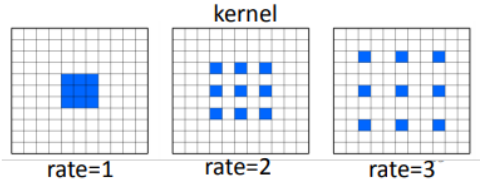
- Semantic Image Segmentation
  - U-Net
  - DeConvNet
  - SegNet
    - Resolution controlling
    - Atrous convolutions, sub-pixel image combination

Avoiding resolution loss but no high computational load:

Atrous convolution

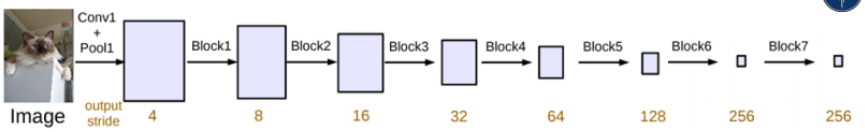
- How it works?
  - Blows up the kernel
  - Filling up the holes with zeros
    - Atrous means very dark (like the wholes between the values)
- Properties
  - Not doing downsampling
  - Not increasing computational load
  - But reaches larger neighborhood
  - Combines information from larger neighborhood



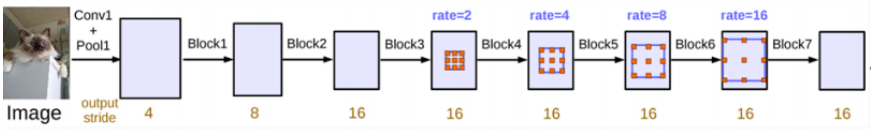


11/12/2019

Depth-to-Space

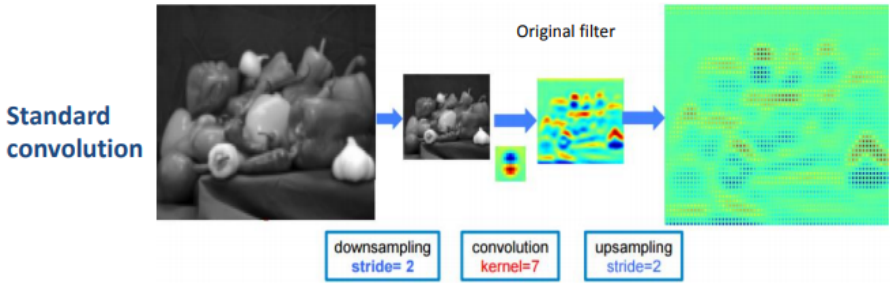


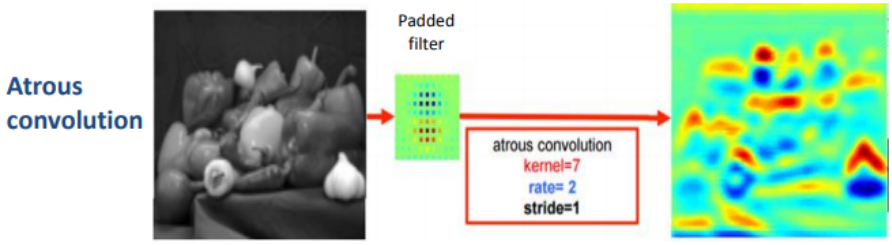
Normal convolution goes deeper with reducing resolution



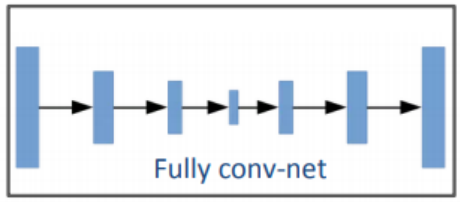
Atrous convolution goes deeper without further reducing resolution

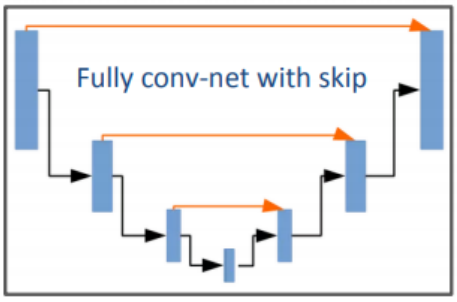
Visualizing atrous convolution

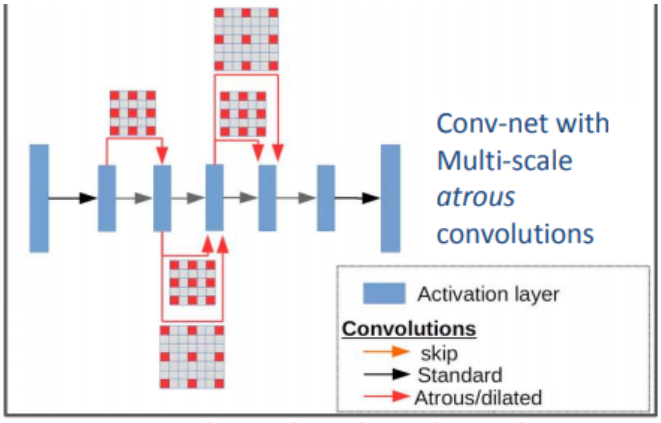




Semantic segmentation CNN arrangements







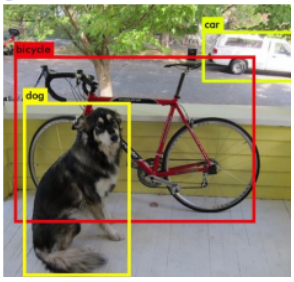
- How to solve reduced resolution?
  - Do not downsample !!!
    - Convolution on large images ⇒ Small FOV Enlarge kernel
    - Size $O(n^2)$ more parameters ⇒ getting close to fully
    - Connected layer, slow training, overfitting
  - Atrous Convolution.
    - Large FOV with little parameters → Kill two birds with one stone!

11/12/2019

d. Object classification, localization VS object detection, semantic segmentation VS instance segmentation -> ez kicsit megfoghatatlan a diasor alapján…

- How to do image classification
  - Alexnet
  - One decision per image (*classification*)

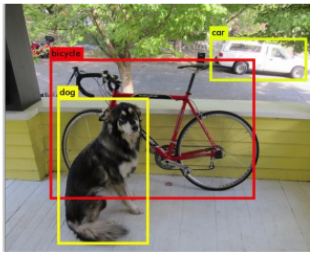
- Detection and Localization is more complex
  - Multiple (few) decision per image
    - *Regressions* for localization
    - *Classification* for detection
- Pixel level Segmentation
  - Very high number of decisions (*classification*) per image



2/2019

Object detection/localization and classification

- Chicken and egg problem
  - You need to know that it is a bicycle before able to say that both a wheel part and a pipe segment belongs to the same object
  - You need to know that the red box contains an object before you can recognize it. (Cannot recognize a bicycle if you try it from separated parts)
- Our brain does it parallel
- How neural nets can solve it?
  - Detection by regression?
  - Detection by classification?

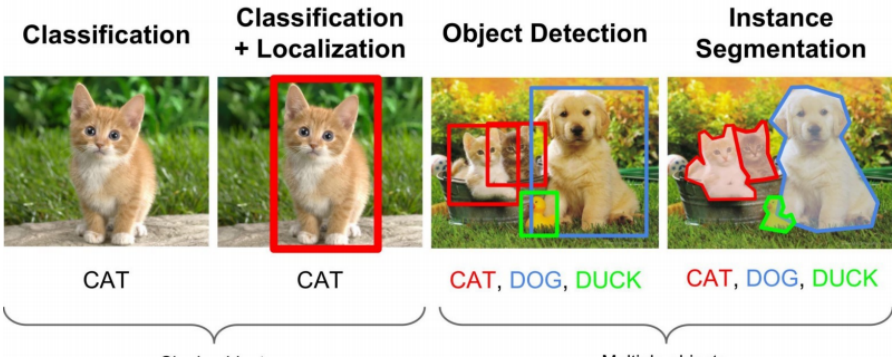


Neural networks for regression

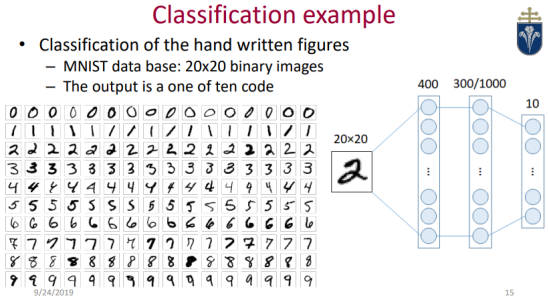
Multiple object detection on a single image

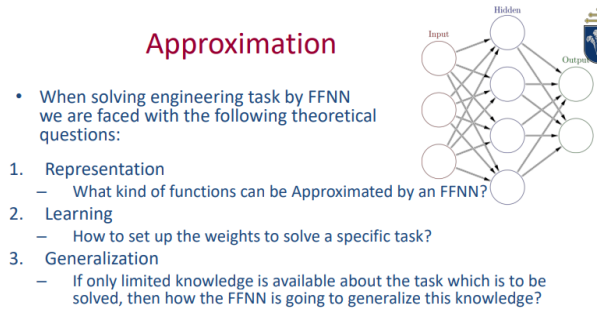
Classification is good for a single object (can be extended for k objects – top k candidates)

How could we detect objects in general, when the number of objects is unknow



Object classification:





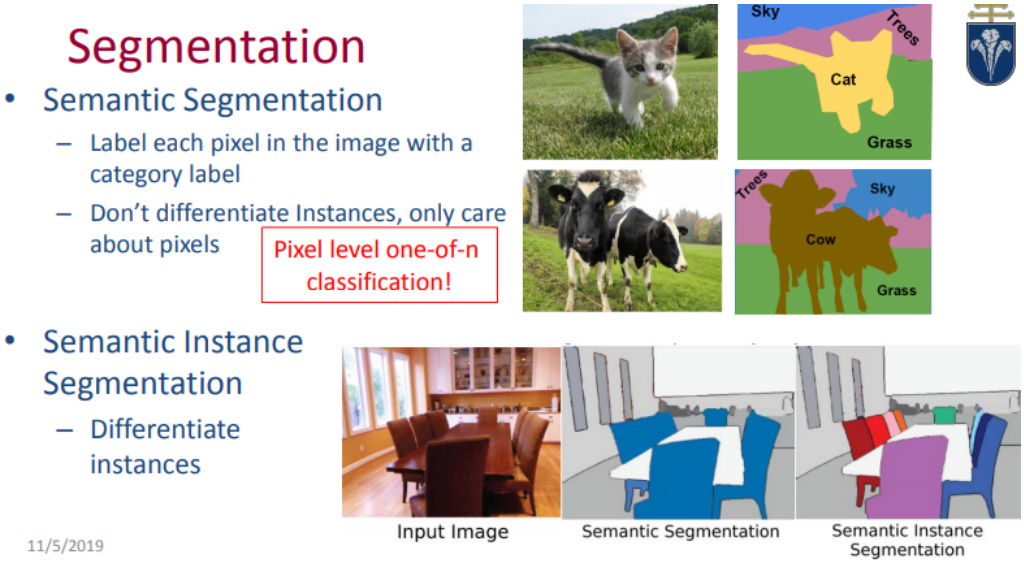
fullos link:
https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81

Object detection:

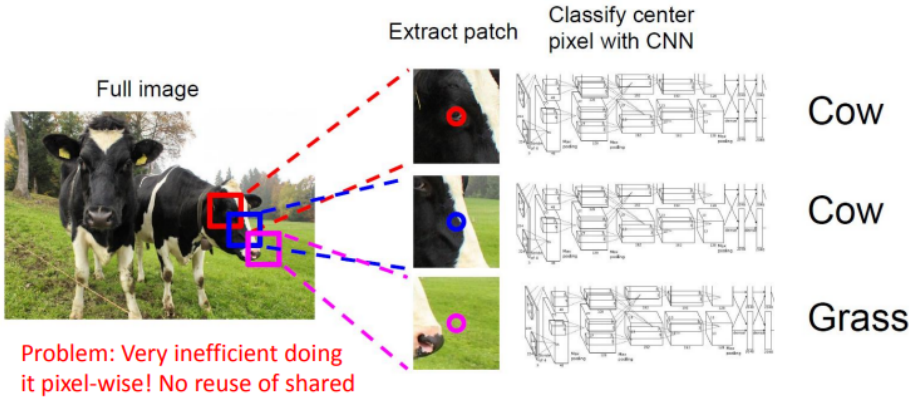
Annotated image database:

- Detection (squared objects)
- Segmentations (segmented objects)

Segmentations:



Semantic Segmentation Idea I: Sliding Window

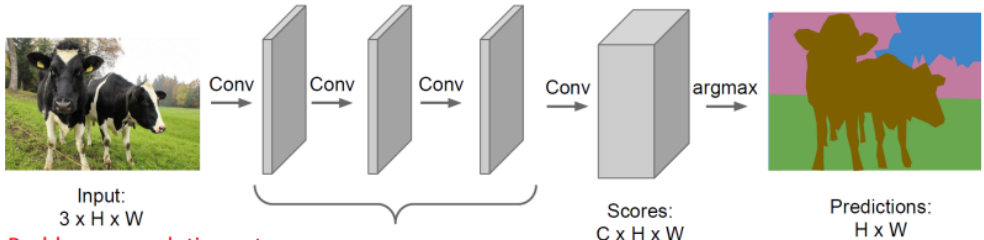


Problem: Very inefficient doing it pixel-wise! No reuse of shared features between overlapping Patches.

Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013
Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", I

Semantic Segmentation Idea II: Fully Convolutional

Design a network as a bunch of convolutional layers to make predictions for pixels all at once!



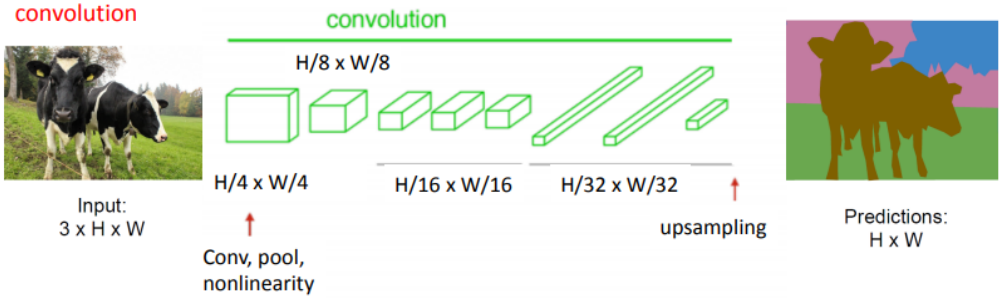
Problem: convolutions at original image resolution will be very expensive ...

Semantic Segmentation Idea III: Fully Convolutional

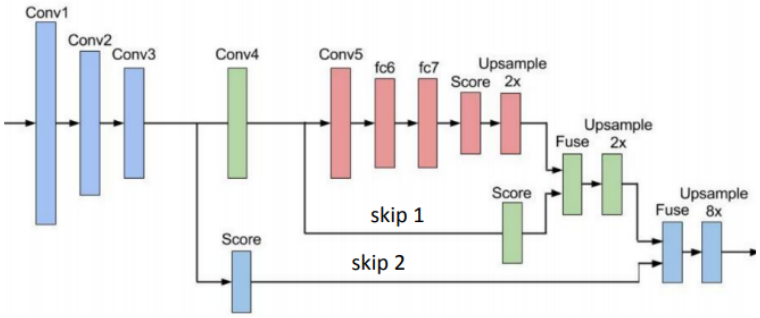
Downsampling: Pooling, strided convolution

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

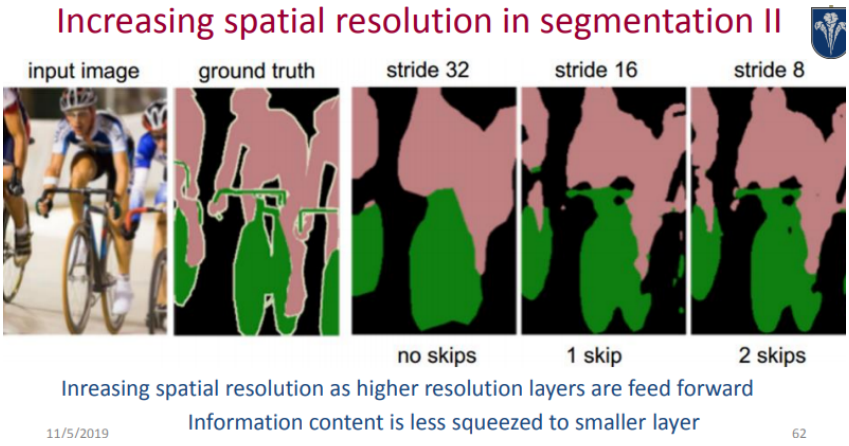
Upsampling: ???



Increasing spatial resolution in segmentation I

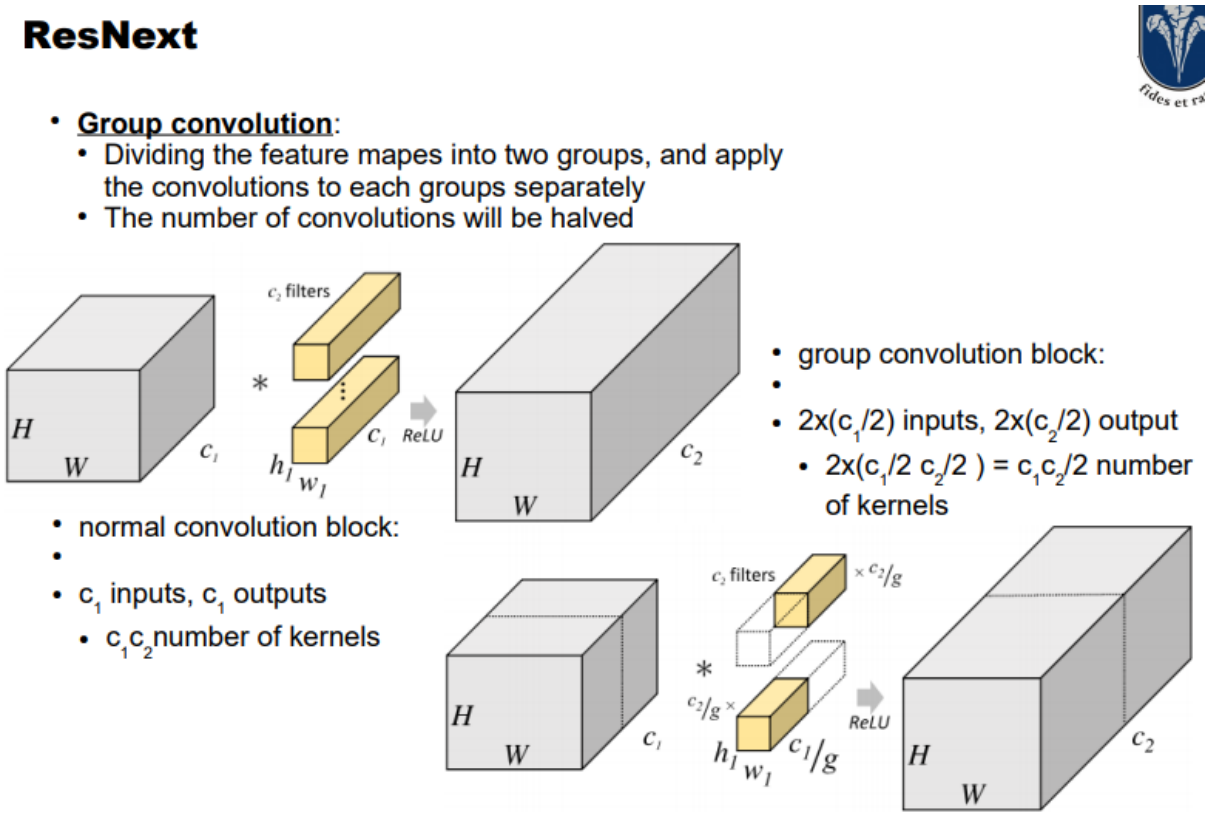


Higher resolution layers directly forwarded to transfer finer spatial information
Called "Skipping". It skips using the coarser (more downsampled) layers
Can be considered of an ensembling of three networks



---

e. ResNext



The model name, ResNeXt, contains Next. It means the next dimension, on top of the ResNet. This next dimension is called the “cardinality” dimension. And ResNeXt becomes the 1st Runner Up of ILSVRC classification task.

No. 9 Topic

a. ADAM optimizer

ADAM algorithm (2014)

- The name "Adam" derives from the phrase "adaptive moments."
- In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions.
- in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient.
- Adam includes bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin

ADAM algorithm

s estimates the gradient from the history (moment)

r estimates the curvature of the gradient

Booth of them are biased to reduce anomalies at the initialization

9/30/2019

Algorithm The Adam algorithm

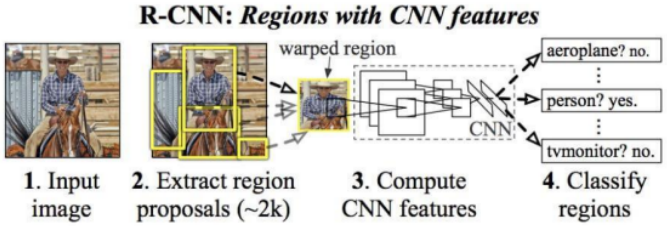
Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

b. The softmax function -> lasd in No. 5 Topic

c. R-CNN architectures: R-CNN, Fast R-CNN, Faster R-CNN

- Detection and Localization
  - PASCAL Database and Competion
  - R-CNN
    - Region proposal, Classification
    - Support Vector Machine (SVN), Bounding box refinement
  - Fast R-CNN
  - Faster R-CNN

R-CNN in a Glance



1. Input image
2. Region proposals
3. Compute CNN features with warped images
4. Classification with Support Vector Machine (SVM)
5. Ranking/selecting/merging → detections
6. Bounding box regression

R-CNN: Region Proposal

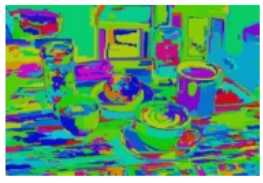
- Requirements:
  - Propose a large number (up to 2000) of regions (boxes) with different sizes
  - Still much better than exhausting search with multi-scale sliding window (brute force)
  - Boxes should contain all the candidate objects with high probability
- R-CNN works with various Region proposal methods:
  - Objectness
  - Constrained Parametric Min-Cuts for Automatic Object Segmentation
  - Category Independent Object Proposals
  - Randomized Prim
  - Selective Search
- Selective Search is the fastest and provides best regions

R-CNN: Selective Search I

- Graph based segmentation (Felzenszwalb and Huttenlocher method)
  - cannot be used in this form, because one object is covered with multiple segments, moreover regions for occluded objects will not be covered
- Idea: oversegment it and apply scaled similarity based merging

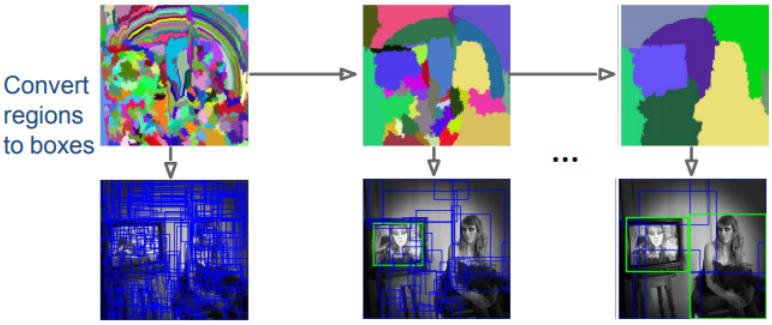

Input image


Segmented image


Oversegmented image

R-CNN: Selective Search II

Step-by-step merging regions at multiple scales based on similarities

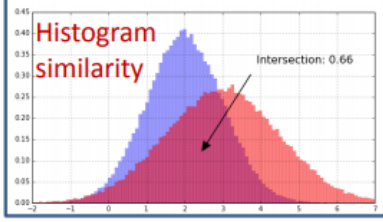


Similarity measures I

Color Similarity

- Generate color histogram of each segment (descriptor)
  - 25 bins/ color channels
  - Descriptor vector (c_i^k)size: 3x25=75
- Calculate histogram similarity for each region pair

$$s_{color}(r_i, r_j) = \sum_{k=1}^{75} \min(c_i^k, c_j^k)$$



c_i^k *is the histogram value for the k^th bin in color descriptor*

Texture Similarity

- Texture features: Gaussian derivatives at 8 orientations in each pixel
  - 10 bins/color channels
  - Descriptor vector (t_i^k)size: 3x10x8=240
- Each region will have a texture histogram
- Calculate histogram similarity for each region pair

$$s_{texture}(r_i, r_j) = \sum_{k=1}^{240} \min(t_i^k, t_j^k)$$

t_i^k *is the histogram value for the k^th bin in texture descriptor*

Size Similarity

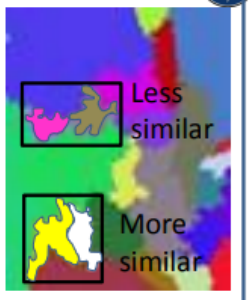
- Helps merging the smaller sized objects
- Since we do bottom up merging, the small segments will be merged first, because their size similarity score is higher

$$s_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(image)}$$

$size(image)$ *is the size of the entire image in pixels*

Shape Similarity

- Measures how well two regions are fit
  - How close they are
  - How large is the overlap



$$s_{fill}(r_i, r_j) = 1 - \frac{size(BB_{ij}) - size(r_i) - size(r_j)}{size(image)}$$

$size(BB_{ij})$ *is the size of the bounding box of* r_i *and* r_j

Final Similarity

- Linear combination of the four similarities

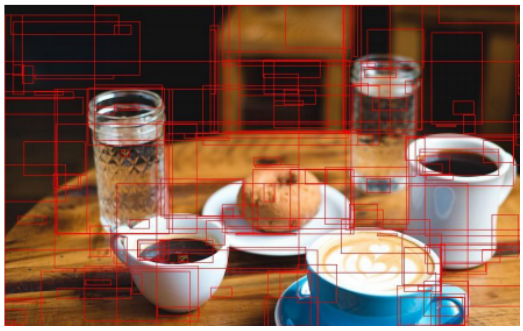
$$s_{final}(r_i, r_j) = a_1 s_{color}(r_i, r_j) + a_2 s_{texture}(r_i, r_j) + a_3 s_{shap}(r_i, r_j) + a_4 s_{fill}(r_i, r_j)$$

List or proposed region

1. Initial oversegmentation
2. Calculation the similarities
3. Merge the similar regions
4. The formed regions are added to the region list (*this ensures that there will be smaller and larger regions in the list as well*)
5. Goto 2

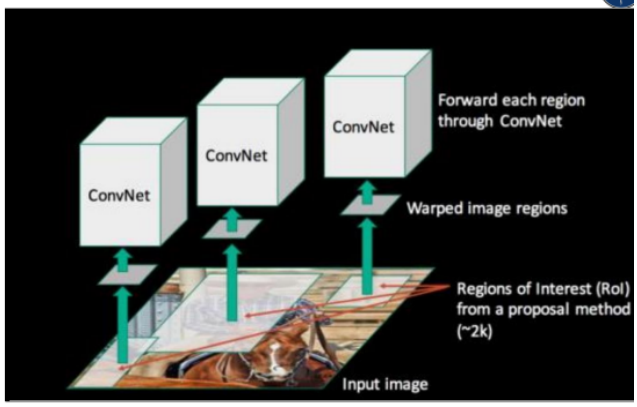
Proposed regions

- Few hundreds or few thousand boxes
- Includes all the objects with high probability
- Number of the boxes are much smaller than with brute force method
- C and python functions exist



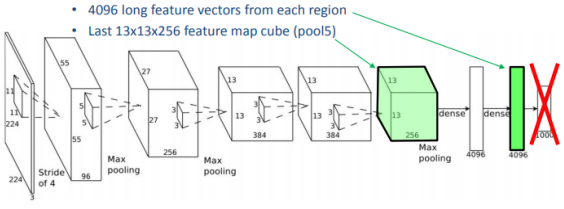
Computing the features of the regions

- Cut the regions one after the other
- Resize (warp) the regions to the input size of the ConvNet
- Calculate the features of the individual regions



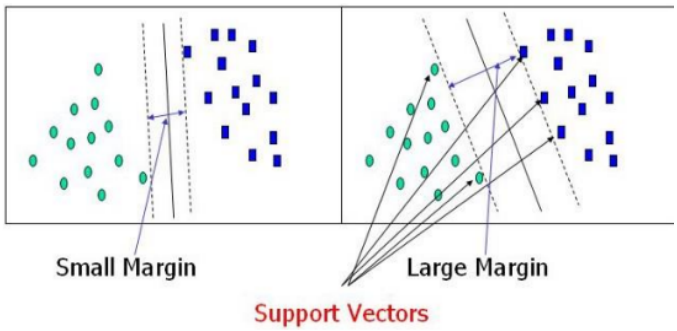
Convolution network

- Pre-trained AlexNet, later VGGNet
- The decision maker SoftMax layer was cut
  - Outputs:
    - 4096 long feature vectors from each region
    - Last 13x13x256 feature map cube (pool5)



Linear Support Vector Machine

- **Idea:** Separate the data point in the data space with a boundary surface (hyperplane) with maximum margin
- Vectors pointing to the data points touching the margins are the support vectors
- The parameters of the optimal hyperplane is calculated with regression



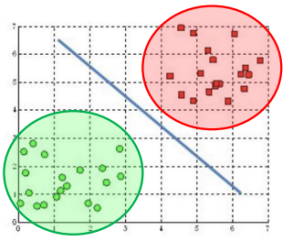
- Similar to single layer perceptron, but optimized for maximum margin

Why SVM?

- Why not use simple the classification output of the AlexNet?
- During the training, the AlexNet/VGGNet is not trained
- Only SVM is trained
- The number of category is much smaller
  - Designed for 20-200 categories rather than 1000

Decision with SVM

As many separate SVM as many category we have



Feature vector of the category to be detected e.g.: Cat

Feature vector of all the other categories plus the background e.g.: No Cat

The result: Each region is categorized in every image classes.

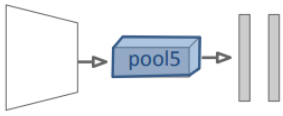
Ranking, selecting, merging

- Greedy non-maximum suppression
  - Regions with low classification probabilities are rejected
  - Regions with high Intersection over Union values (within the same category) are merged

- *Result:* **localized and classified object**

Bounding Box Regression

- Linear regression model
- One per object category
- Input: last feature map cube of the conv net (pool5)
- Output: size and position modification to the bounding box:
  - dx, dy, dw, dh



Training image regions

Input:
Cached feature map cube (pool5)

Regression targets:
(dx, dy, dw, dh)
(normalized)

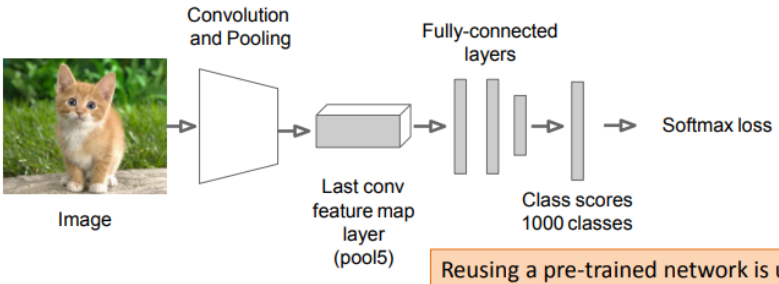
(0, 0, 0, 0)
Proposal is good

(.25, 0, 0, 0)
Proposal too far to left

(0, 0, -0.125, 0)
Proposal too wide

R-CNN Training

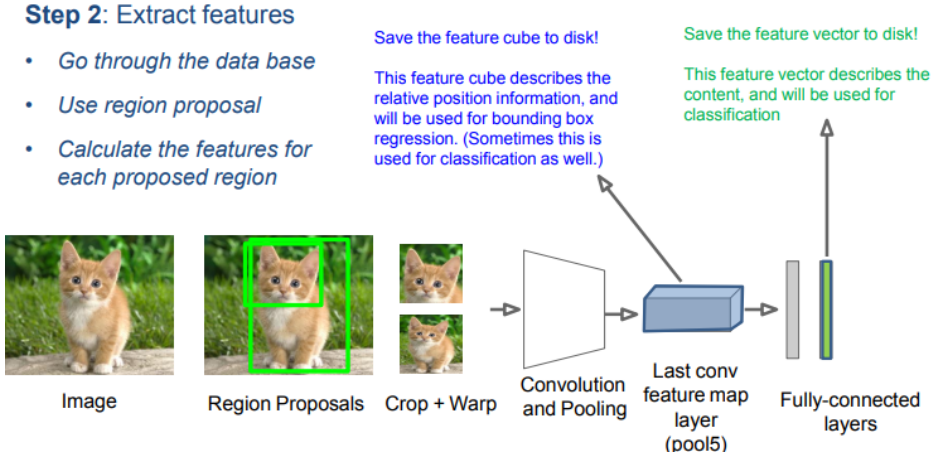
Step 1: Take a pretrained Convolutional Neural Network (e.g. AlexNet)



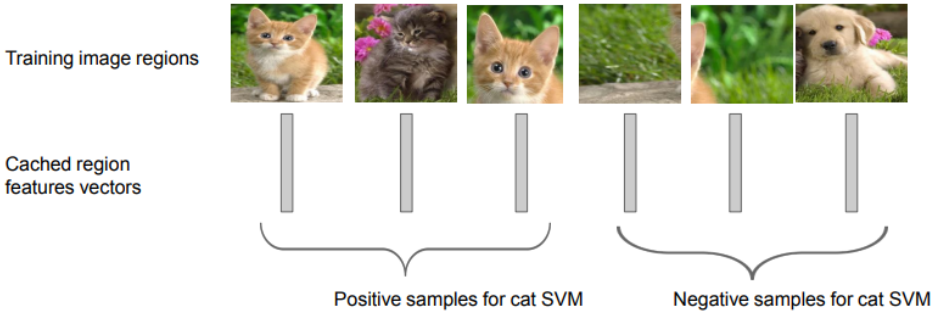
Reusing a pre-trained network is useful, if there is not enough data to train or if it provides good enough result. Fine tuning typically needed!

Step 2: Extract features

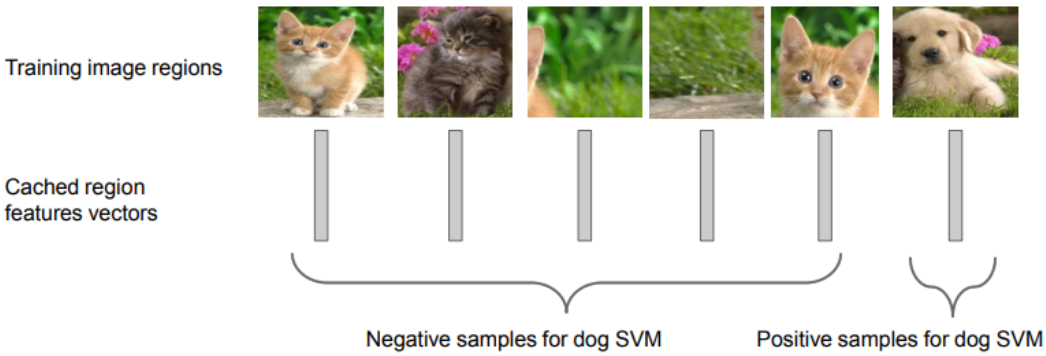
- *Go through the data base*
- *Use region proposal*
- *Calculate the features for each proposed region*



Step 4: Train one SVM per class to classify region features



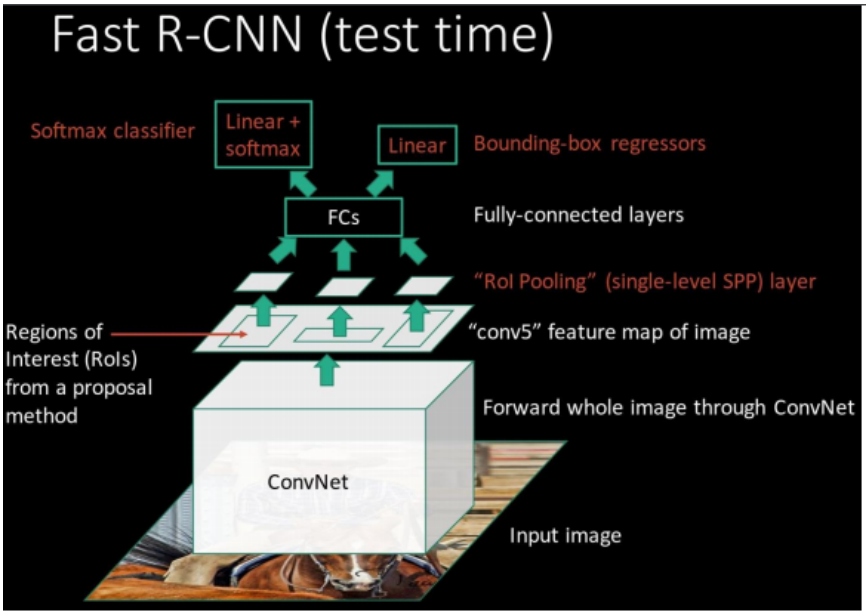
Step 4: Train one SVM per class to classify region features



Step 5 (bbox regression): For each class, train a linear regression model to map from cached features cubes to offsets/size of the boxes to fix "slightly wrong" position proposals

R-CNN Problems

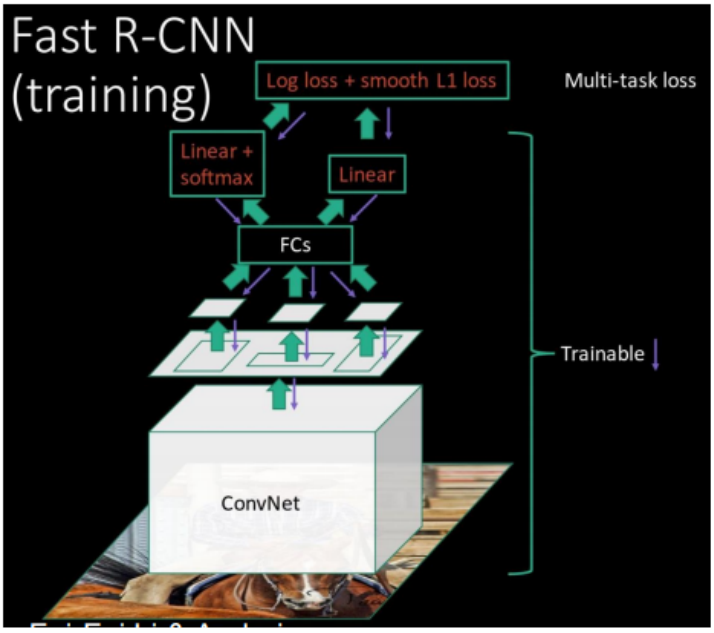
1. Slow at test-time: need to run full forward pass of CNN for each region proposal
   - Recalculate the features again-and-again in the overlapping regions
2. SVMs and bbox regressors are post-hoc:
   - CNN features not updated in response to SVMs and regressors
3. Complex multistage training pipeline
   - Calculate the features for all the regions for all the training image first
   - Then train for SVM and bbox regressor separately



R-CNN Problem #1: Slow at test-time due to independent forward passes of the CNN

Solution: Share computation of convolutional layers between proposals for an image

Girschick, "Fast R-CNN", ICCV 2015



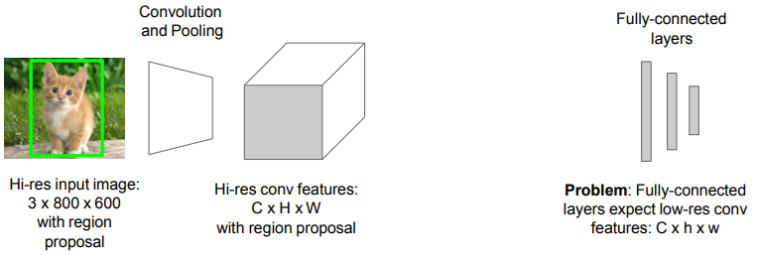
R-CNN Problem #2:
Post-hoc training: CNN not updated in response to final classifiers and regressors

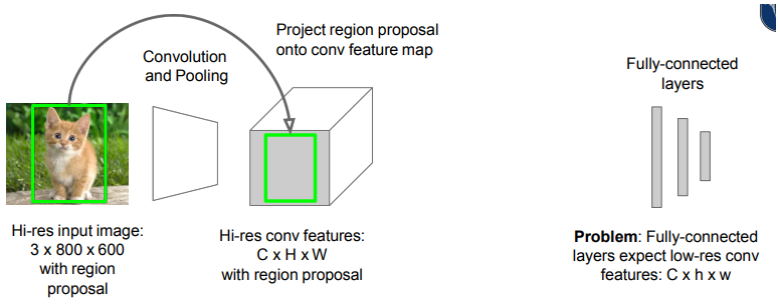
R-CNN Problem #3:
Complex training pipeline

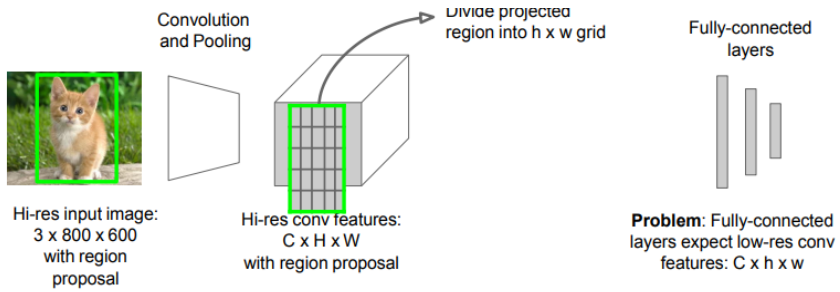
Solution:
Just train the whole system end-to-end all at once!

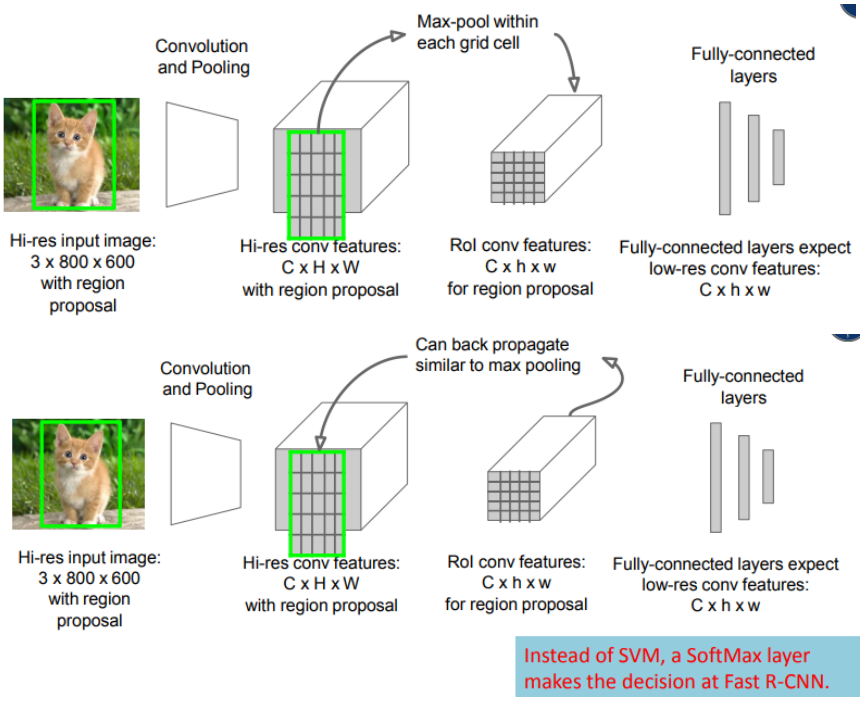
Slide credit: Ross Girschick

Fast R-CNN: Region of Interest Pooling





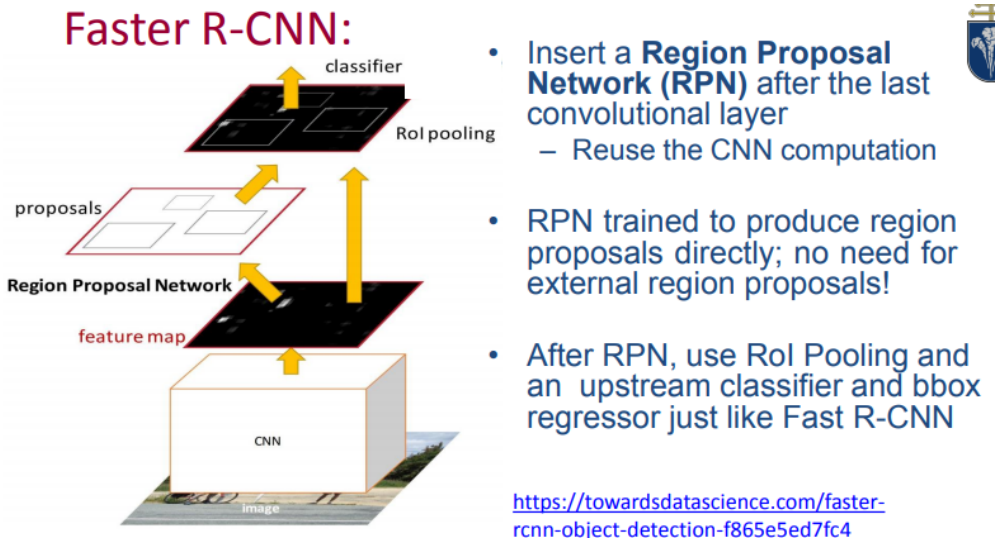




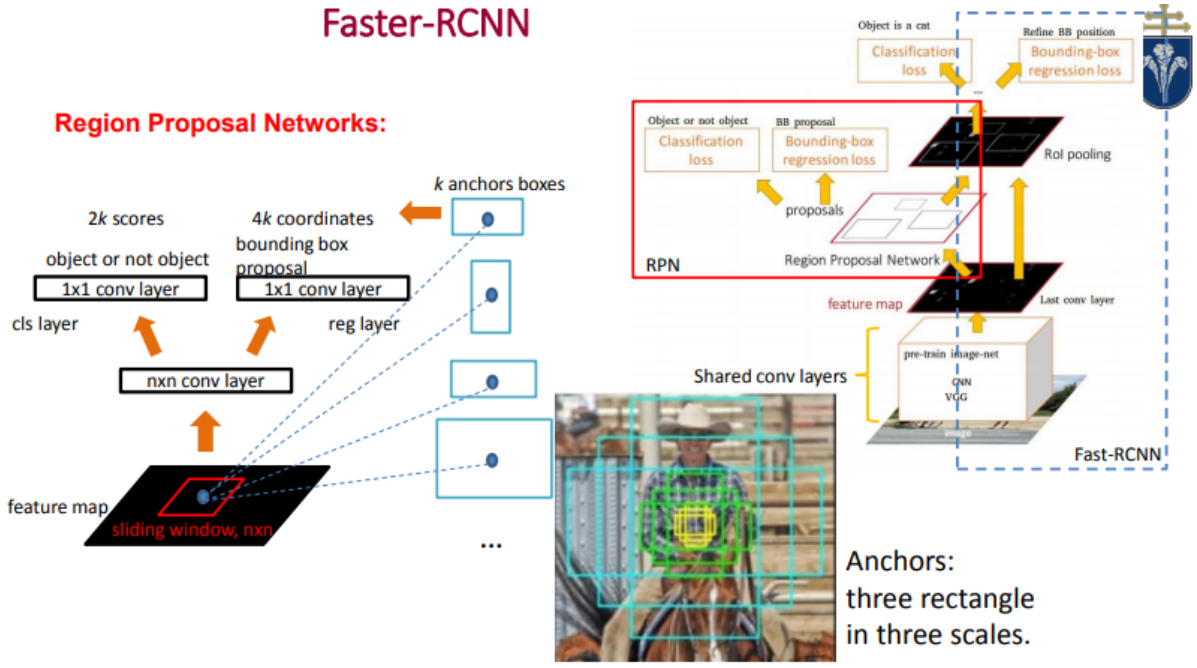
ennyi a lényeg: (training time)

R-CNN -> Fast R-CNN

84 hours -> 9.5 hours



Faster-RCNN



Anchors:
three rectangle
in three scales.

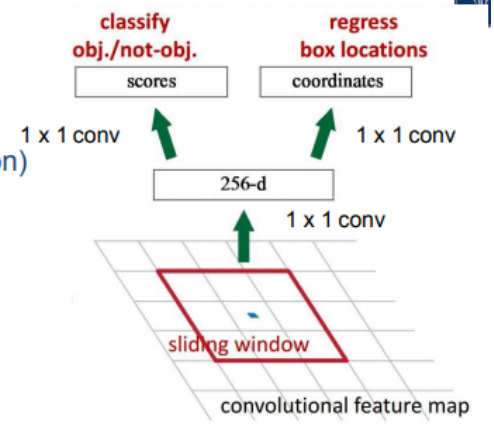
Faster R-CNN: Region Proposal Network

Slide a small window on the feature map
(very small computational effort per position)

Build a small network for:
- classifying object or not-object, (Binary decision)
- regressing bbox locations

Position of the sliding window provides localization information with reference to the image

Box regression provides finer localization information with reference to this sliding window

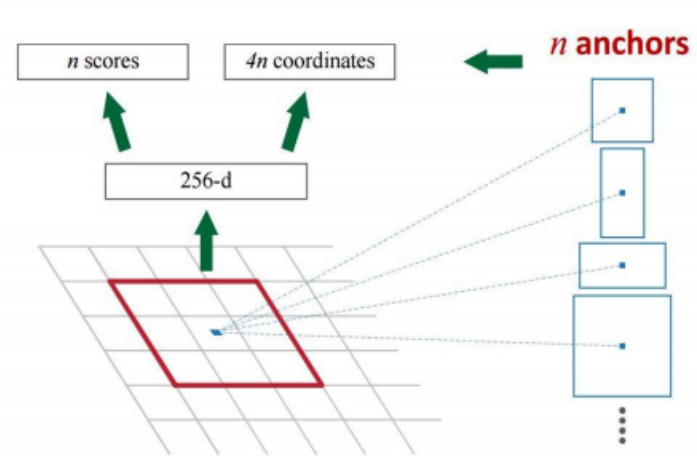


Use **N anchor boxes** at each location

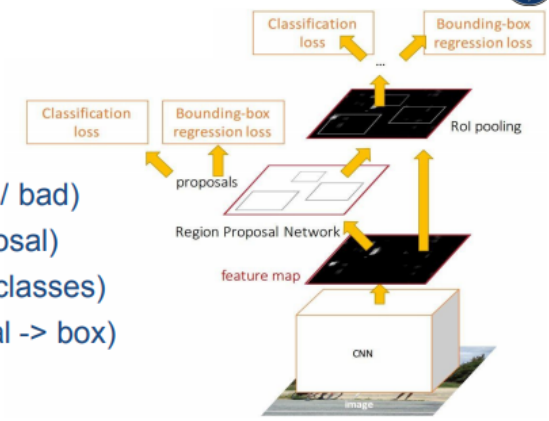
Anchors are **translation invariant**: use the same ones at every location

Regression gives offsets from anchor boxes

Classification gives the probability that each (regressed) anchor shows an object



Faster R-CNN: Training



One network, four losses

- RPN classification (anchor good / bad)
- RPN regression (anchor -> proposal)
- Fast R-CNN classification (over classes)
- Fast R-CNN regression (proposal -> box)

le credit: Ross Girschick

	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image (with proposals)	50 seconds	2 seconds	**0.2 seconds**
(Speedup)	1x	25x	**250x**
mAP (VOC 2007)	66.0	**66.9**	**66.9**

d. Supervised VS unsupervised learning

- **Supervised Learning**
  - Learning from labeled examples (for which the answer is known)
- **Unsupervised Learning**
  - Learning from unlabeled examples (for which the answer is unknown)
- **Reinforcement Learning**
  - Learning by trial and feedback, like the "child learning" example

- Supervised learning
  - We have prior knowledge of the desired output
    - Always have data set with ground truth (like image data sets with labels)
  - Typical tasks
    - Classification
    - Regression

- Unsupervised learning
  - No prior knowledge of the desired output
    - Received radio signals from deep space
  - Typical tasks
    - Clustering
    - Representation learning
    - Density estimation

We wish to learn the inherent structure of (patterns in) our data.

Use cases for unsupervised learning

- Exploratory analysis of a large data set
  - Clustering by data similarity
  - Enables verifying individual hypothesizes after analyzing the clustered data

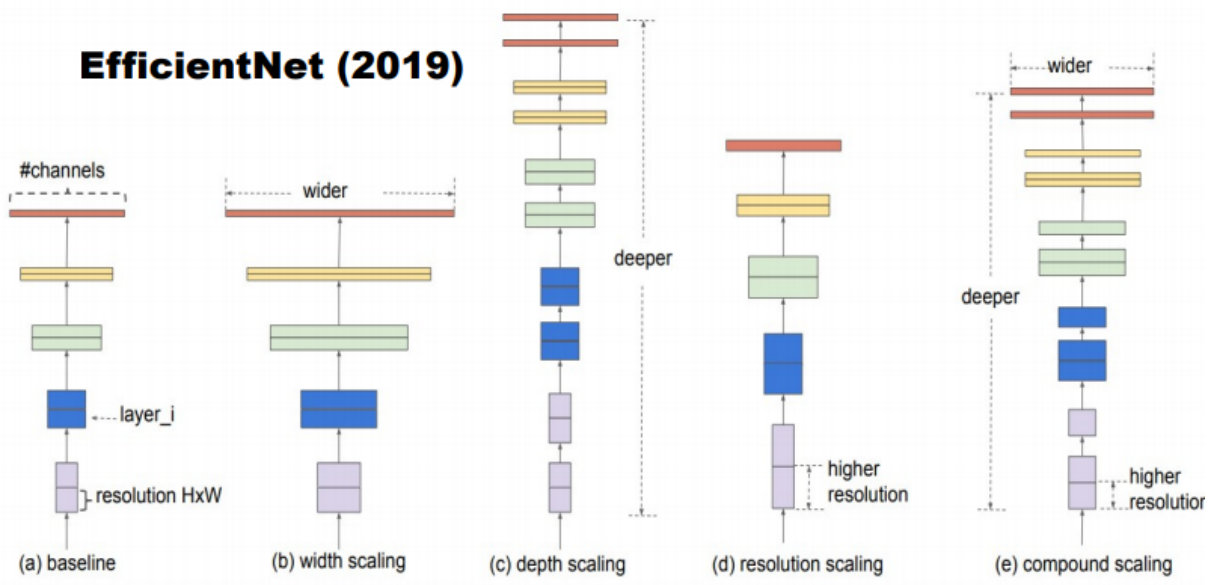
- Dimensionality reduction
  - Represents data with less columns
  - Allows to present data with fewer features
  - Selects the relevant features
  - Enables less power consuming data processing, and/or human analysis

Unsupervised learning techniques

- Curse of dimensionality
- Principal component analysis (PCA)
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Autoencoder

---

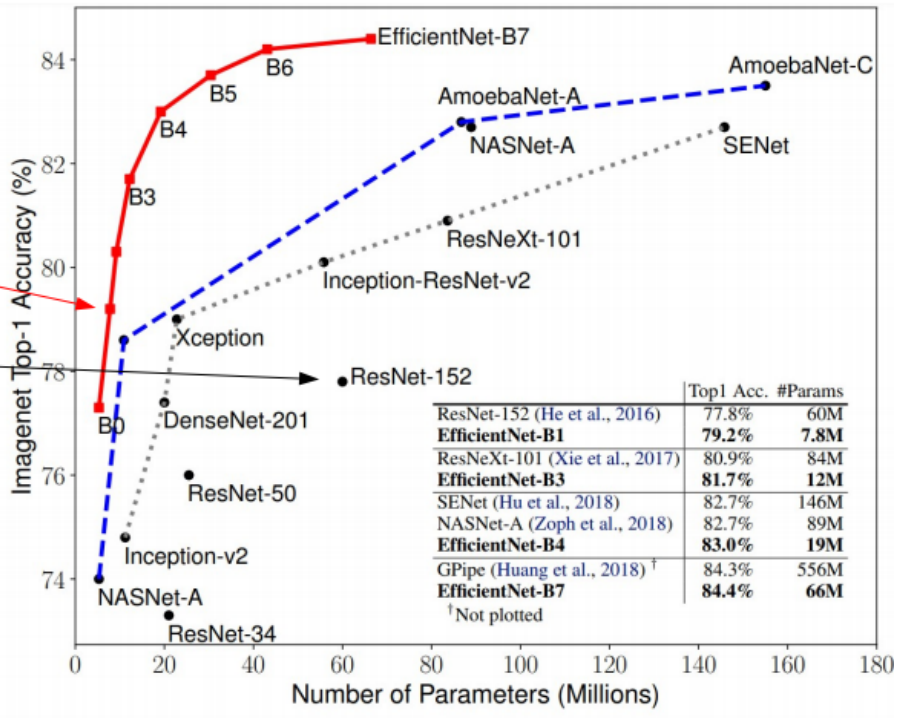
e. EfficientNet



- Scale the width, the depth, and the resolution uniformly!
- Can be used for any existing architecture, and the efficiency will be significantly better with the same performance
  - EfficientNet-B7 achieves stateof-the-art 84.4% top-1 / 97.1% top-5 accuracy on ImageNet, while being 8.4x smaller (number of parameters) and 6.1x faster on inference than the best existing ConvNet.
- Best performance can be reached by using NN to generate the optimal baseline ConvNet.

7 different scaled version of EfficientNet. (B0, B1, … B7)

EfficientNet-B1 is 7.6x smaller and 5.7x faster than ResNet-152.



	Top1 Acc.	#Params
ResNet-152 (He et al., 2016)	77.8%	60M
EfficientNet-B1	**79.2%**	**7.8M**
ResNeXt-101 (Xie et al., 2017)	80.9%	84M
EfficientNet-B3	**81.7%**	**12M**
SENet (Hu et al., 2018)	82.7%	146M
NASNet-A (Zoph et al., 2018)	82.7%	89M
EfficientNet-B4	**83.0%**	**19M**
GPipe (Huang et al., 2018) †	84.3%	556M
EfficientNet-B7	**84.4%**	**66M**

†Not plotted

No. 10 Topic

a. **Optimization problem of objective functions of NN** - tessék??

https://www.youtube.com/watch?v=AoJQS10Ewn4

Optimization problems are commonly written in the form minimize f(x)
Here, f is the objective function

- Objective Function
  - The value you are trying to optimize
  - Minimized or maximized
- Decision Variables
  - The values the optimizer can change
  - Also called design or manipulated variables

Learning

- The questions are the following
  - What is the relationship of these optimal weights?

$$\mathbf{w}_{\text{opt}} \overset{???}{\Leftrightarrow} \mathbf{w}_{\text{opt}}^{(K)}$$

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \left(d_k - Net\left(\mathbf{x}_k, \mathbf{w}\right)\right)^2$$

  - How this new objective function should be minimized as quickly as possible?

Learning – in practice

- Learning based on the training set:

$$\tau^{(K)} = \left\{\left(\mathbf{x}_k, d_k\right); k = 1, \ldots, K\right\}$$

- Minimize the empirical error function (R_{emp})

$$\mathbf{w}_{\text{opt}}^{(K)} : \min_{\mathbf{w}} \frac{1}{K} \sum_{k=1}^{K} \underbrace{\left(d_k - Net\left(\mathbf{x}_k, \mathbf{w}\right)\right)^2}_{E_k} = \min_{\mathbf{w}} R_{emp}\left(\mathbf{w}\right)$$

- Learning is a multivariate optimization task

b. AdaGrad optimizer

- The AdaGrad algorithm (2011) individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values
- The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate
- The net effect is greater progress in the more gently sloped directions of parameter space
- AdaGrad performs well for some but not all deep learning models

AdaGrad algorithm

Algorithm The AdaGrad algorithm

Remembers the entire history evenly

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

RMSP algorithm

- The RMSProp algorithm (2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average
- In each step AdaGrad reduces the learning rate, therefore after a while it stops entirely!
- AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure
- RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl

Video comparing adaptive and non-adaptive:

Three optimizer types are compared:

- SGD
- Momentum types
  - Momentum
  - Nesterov AG
- Adaptív
  - AdaGrad
  - AdaDelta
  - RmsProp

Adaptive ones are the fastest

c. Input vector normalization

Data regularization techniques

- Modification of the input vectors and internal data and internal parameters of the net
- Targeting to perform better in generalization
- Increases the loss during training phase
- Puts the parameters further away from a minimum with an expectation of it will find a deeper minimum
- In many cases these are heuristic methods with mostly experimental and partial mathematical proof

Input vector normalization

- When the input vector contains high and small mean values in different vector positions it is usefull to normalize them
- Squeezes the number to the same range
- Speeds up the training process

Input vector: $x = \begin{pmatrix} 0{,}45 \\ 1589{,}2 \\ 0{,}00143 \end{pmatrix}$

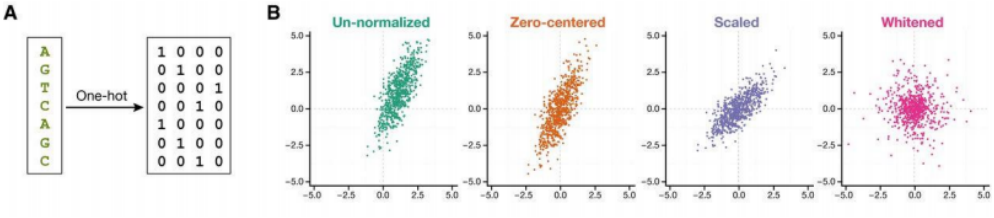
mean: $\bar{x} = \begin{pmatrix} 0{,}32 \\ 1423{,}2 \\ 0{,}00132 \end{pmatrix}$

deviation: $\sigma = \begin{pmatrix} 0{,}11 \\ 155{,}2 \\ 0{,}00042 \end{pmatrix}$

normalized input vector: $x_{normed} = \frac{x - \bar{x}}{\sigma} = \begin{pmatrix} 1{,}18 \\ 1{,}06 \\ 0{,}26 \end{pmatrix}$

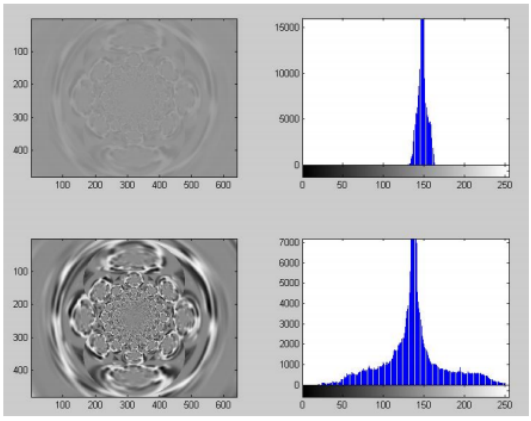
Input Normalization

- Different normalization strategies exists for different input types
- Showing it in two dimension, it shapes the input vector



Once you trained you net with a normalized training set, you have to apply normalization when a previously unseen vector (a new observation) is appled during inference.
OK, but how do you know the statistics?

Input normalization ezample



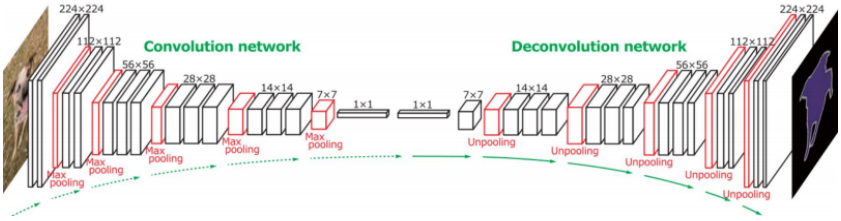
d. DeconvNet, U-Net

Semantic Image Segmentation

- U-Net
- DeConvNet
- SegNet
  - Resolution controlling
  - Atrous convolutions, sub-pixel image combination

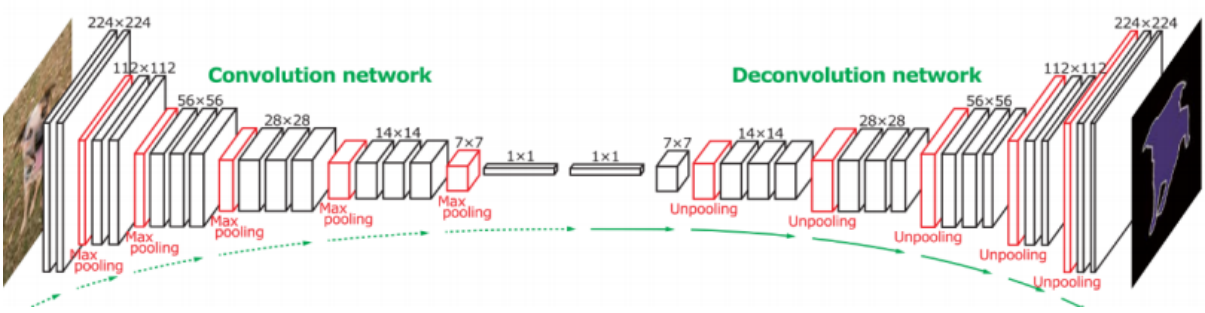
DeconvNet

- Instance-wise segmentation
- Two-stage training:
  - train on easy examples (cropped bounding boxes centered on a single object) first and
  - then more difficult examples

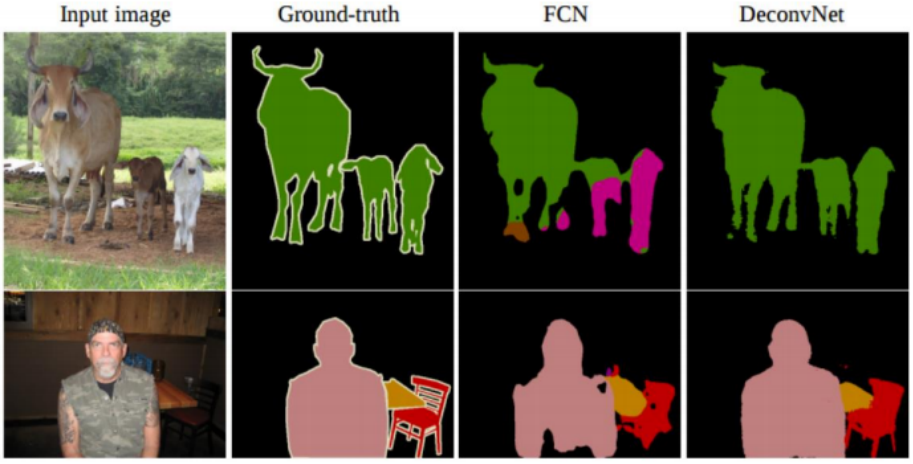


Deconvnet: Extreme segmentation I

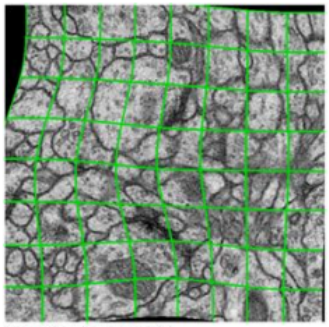
- Fully symmetrical convolutional network
  - All convolution and pooling layers are reversed
- Two stage training (first side trained for classification first)
- Takes 6 days to train on titan GPU
- Output probability map same size as input



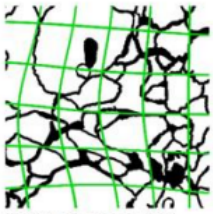
Deconvnet: Extreme segmentation II



U-Net



resulting deformed image
(for visualization: no rotation, no shift, no extrapolation)

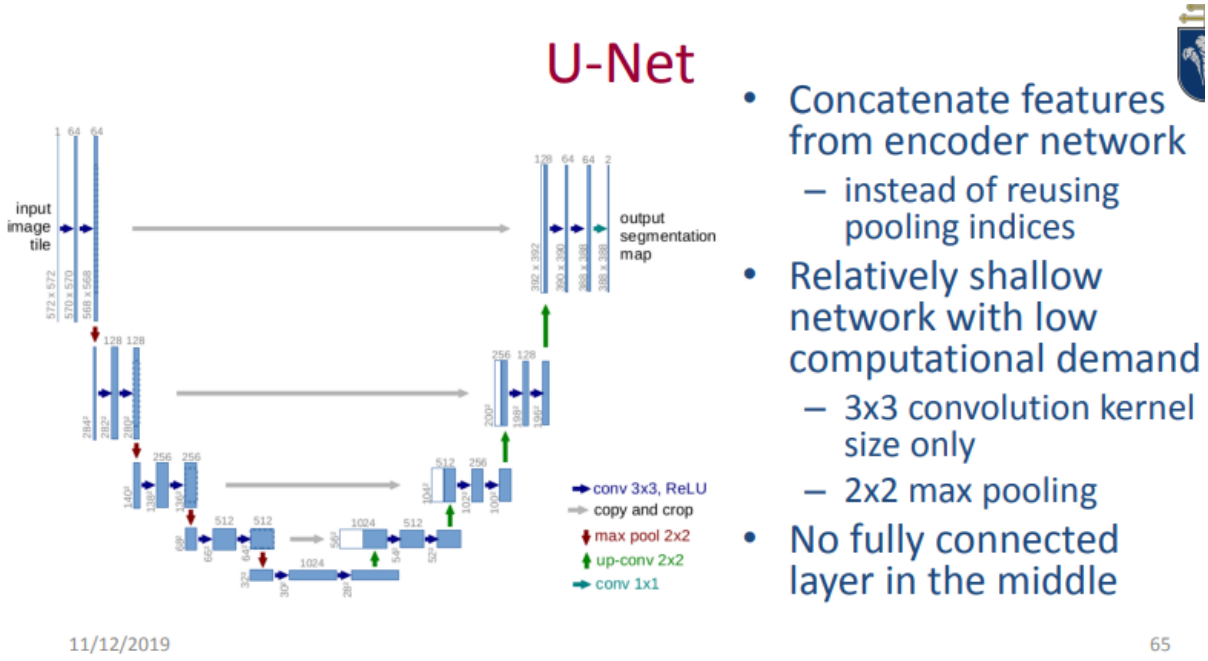


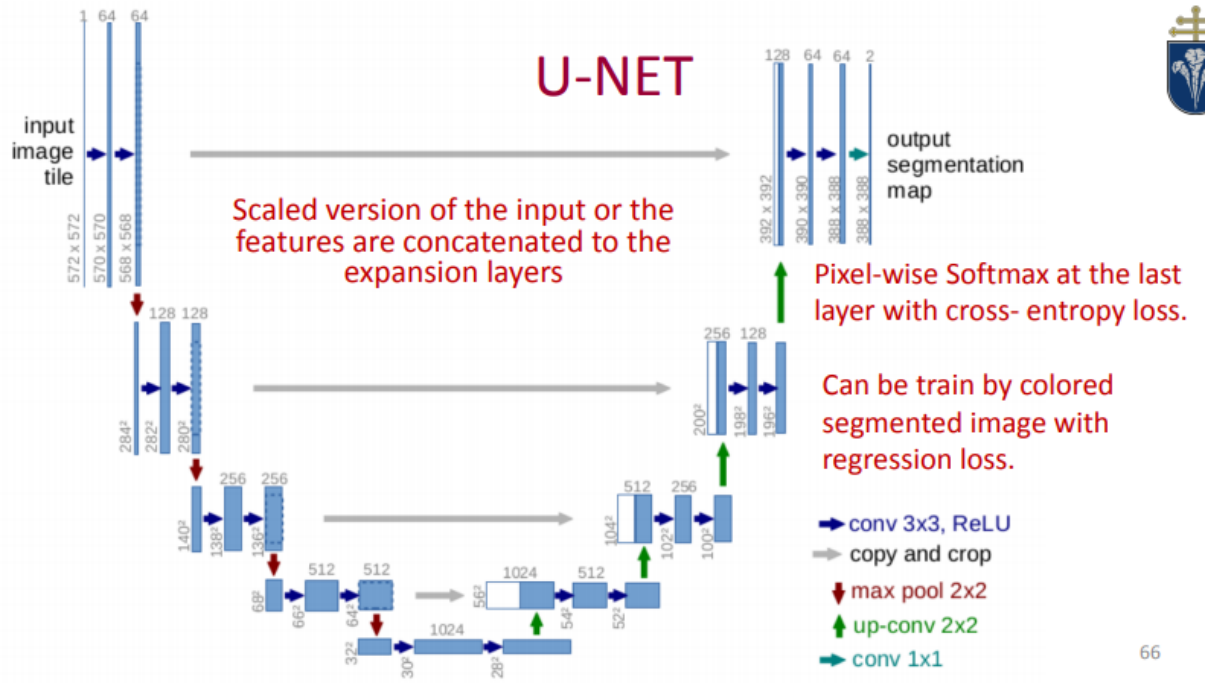
correspondingly deformed
manual labels

- Designed for biomedical image processing: cell segmentation
- Data augmentation via applying elastic deformations,
  - Natural since deformation is a common variation of tissue
  - Smaller dataset is enough

From [3]

64





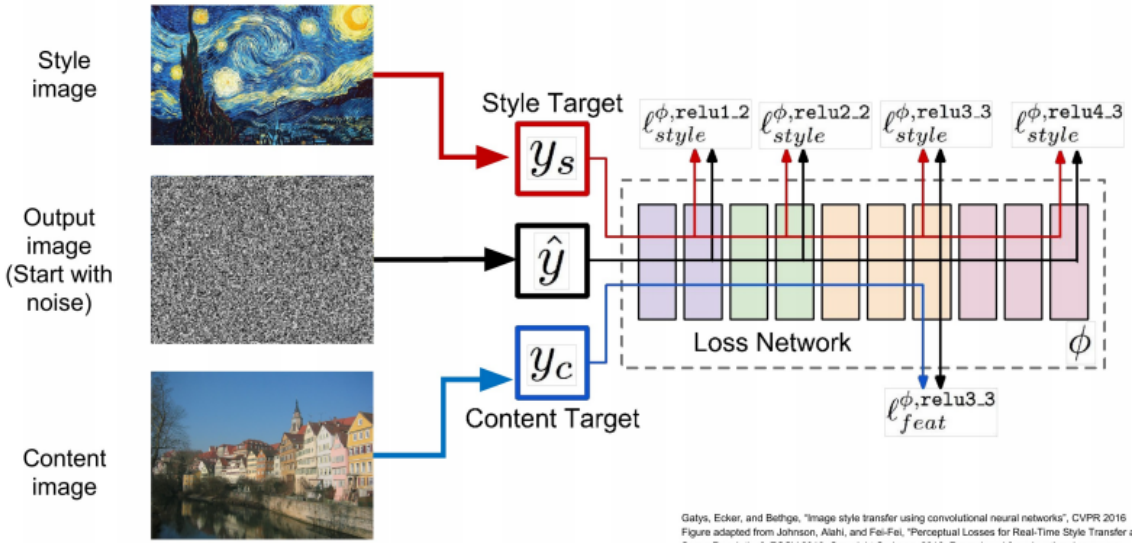
---

e. Neural style transfer

An interesting application of the gradient ascent method is neural style transfer Could we use an input image and transform it into the style of an other input image?

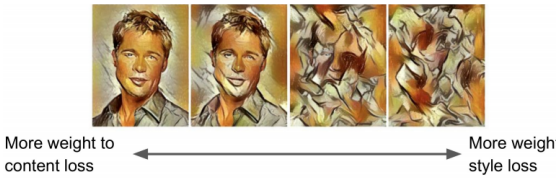
Could we use an input image and transform it into the style of an other input image?
Gradient ascent transforms the image according to a loss function. Can we find a loss function, which would preserve objects and another which preserves features connected to style?

Style transfer works, but It requires a lot of time, to generate an image. Many forward and backward passes are needed.
We could train a network that learns the result of this iterative transformation, and tries to predict it. Only a single pas is needed.



We have a loss function for content:
Can the same objects be found on both images? Content loss, Perceptual loss: this is a distance between the two embedded image vectors in the last features layers
Style loss:
Can the same low level features, edges structures, simple patterns be found on both images
Style loss: Distances between lower level representations of the images

Could we use an input image and transform it into the style of an other input image?
Gradient ascent transforms the image according to a loss function. Can we find a loss function, which would preserve objects and another which preserves features connected to style?



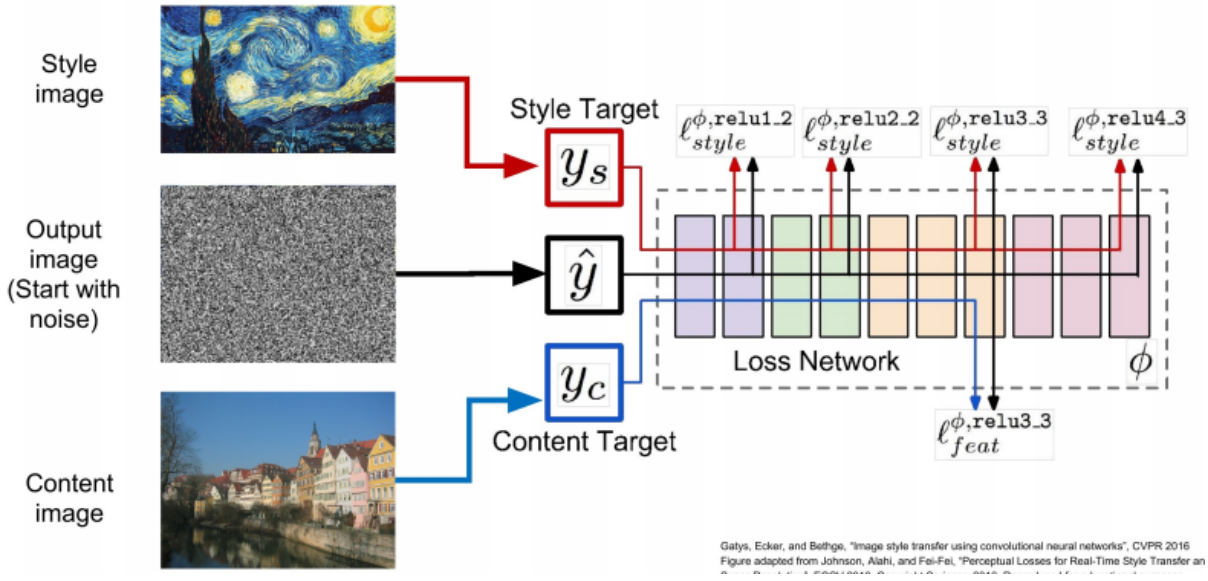
Neural style transfer with Cycle Consistent GANs

Fast Neural Style Transfer

Could we use an input image and transform it into the style of an other input image?

Gradient ascent transforms the image according to a loss function.

Can we find a loss function, which would preserve objects and another which preserves features connected to style?

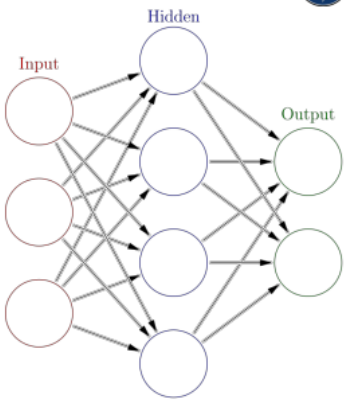


No. 11 Topic

a. Multilayer perceptron

Multilayer perceptron

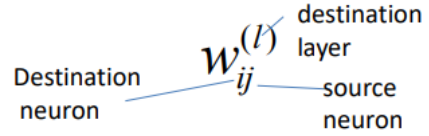
- Different names of Multilayer perceptron
  - Feed forward neural networks (FFNN)
  - Fully connected neural networks
- Multilayer neural network
  - Input layer
  - Hidden layers (one or multiple)
  - Output layer
  - The outputs are the inputs of the next layer
  - Many hidden layers → deep network
- Multiple inputs, multiple outputs
- The output is typically not binary
- Used practically in all deep neural networks!

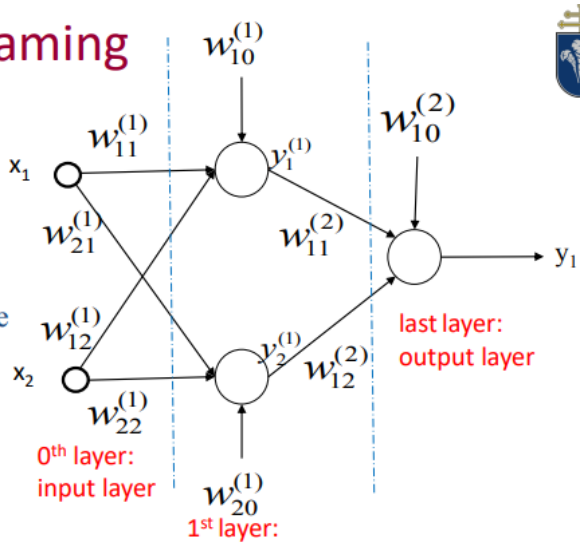


Can solve linearly non-separable problems!

Topology and naming

- Weights: $w_{ij}^{(l)}$
  - Arrives to the l^{th} layer
  - Comes from the j^{th} neuron from the $(l-1)^{th}$ layer
  - Arrives to the i^{th} neuron of the l^{th} layer



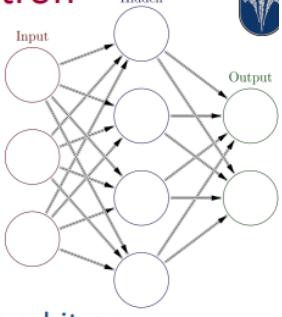


9/24/2019.

6

Usage of Multilayer Perceptron

- Multilayer perceptrons are used for
  - Classification
    - Supervised learning for classification
    - Given inputs and class labels
  - Approximation
    - Approximate an arbitrary function with arbitrary precision



b. Early stopping

Regularization and optimization methods

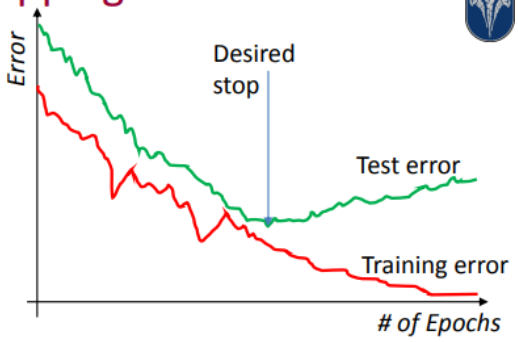
Different methods to increase the loss in the learning phase, but reduce overfitting and increase generalization capabilities

- Local response normalization
- Batch normalization
- Data augmentation (Enriching the data set)
- Early stopping
- Ensemble methods
  - Network duplication
  - Bagging
  - Dropout

Ian Goodfellow: regularization is *"any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error"*

Early stopping

- **Idea:**
  - Split data into training and test sets
  - At the end of each epoch (or, every N epochs):
    - evaluate the network performance on the test set
    - if the network outperforms the previous best model: save a copy of the network parameters at the current epoch



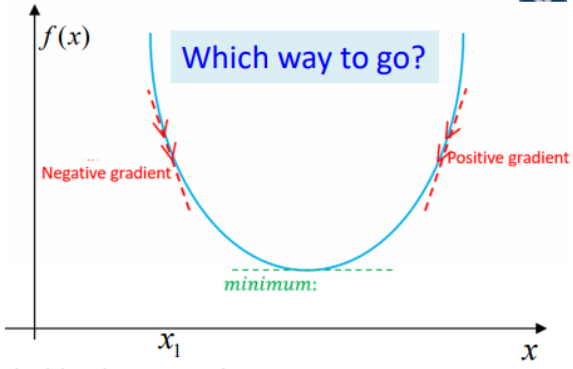
  - The best suboptimum is selected finally
  - Since the error function is not necessarily monotonic, the optimization goes on, but the suboptima are saved

---

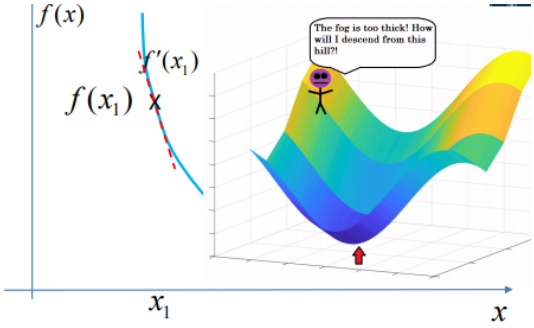
c. Gradient descent (multidimensional cases as well)

Basic idea of Gradient Descent

- There is a function, where $f(x)$ and $f'(x)$
  - can be calculated at any points, but $f'(x)=0$
  - cannot.
- Therefore the trace of the light blue line is not known.
- We have to start out from one point (say x_1) and with an iterative method, we need to go towards the minimum



- We do not know where the curve is
- We know the value at $f(x_1)$
- We know the derivative at x_1 $f'(x_1)$
- Which way to go?
- Idea: follow the descending gradient!



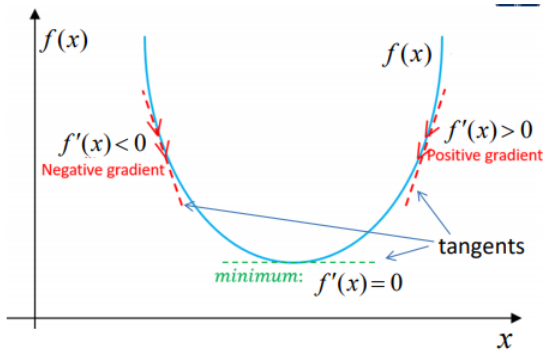
Derivative means for small ε

$$f(x+\varepsilon) \approx f(x) + \varepsilon f'(x)$$

therefore

$$f(x - \varepsilon\, sign(f'(x))) \leq f(x)$$

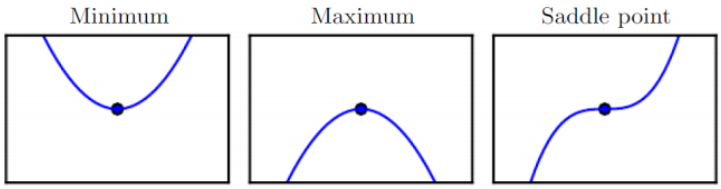
This technique is called **Gradient Descent (Cauchy, 1847).**



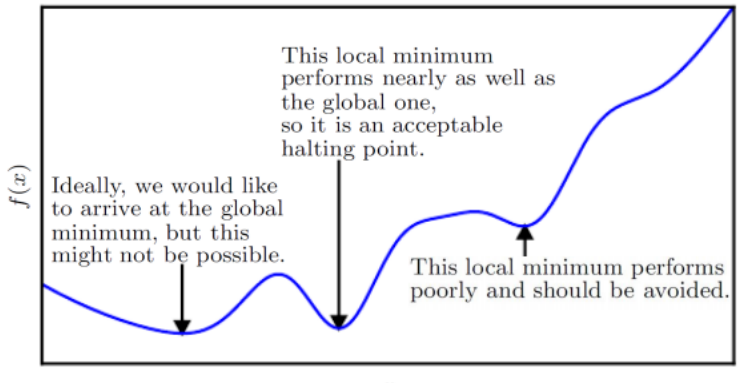
Optimization goal is to find the $f'(x) = 0$ position.
(Critical or stationary points)

Stationary points

- Local minimum, where *f`(x)=0*, and *f(x)* is smaller than all neighboring points
- Local maximum, where *f`(x)=0*, and *f(x)* is larger than all neighboring points
- Saddle points, where *f`(x)=0*, and neither minimum nor maximum



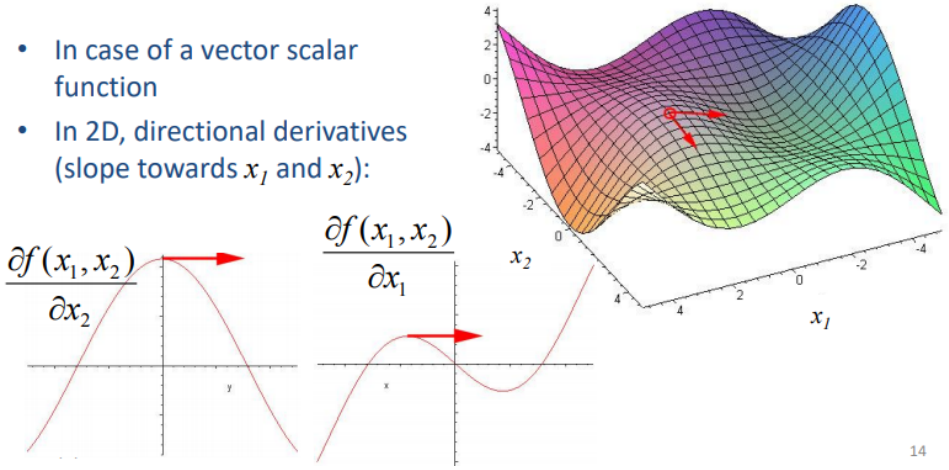
Local and global minimum



In neural network parameter optimization we usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense.

Multidimensional input functions I

- In case of a vector scalar function
- In 2D, directional derivatives (slope towards x_1 and x_2):

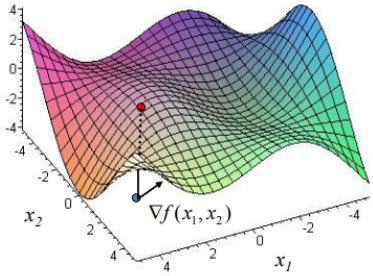


14

- In case of a vector scalar function
- Gradient definition in 2D

$$f : R^2 \to R$$

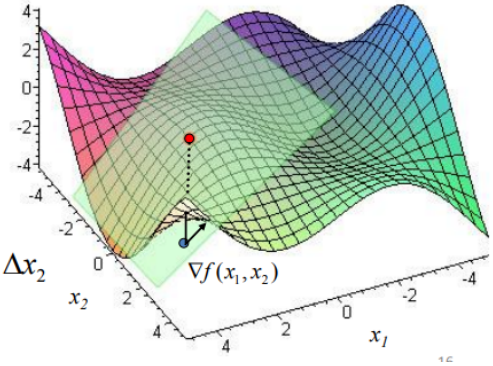
$$\nabla f(x_1,x_2) := \left(\frac{\partial f}{\partial x_1} \quad \frac{\partial f}{\partial x_2} \right)$$



A vector in the in the x_1 - x_2 plane

- The gradient defines (hyper) plane approximating the function infinitesimally at point $\mathbf{x}\ (x_1, x_2)$

$$\Delta z = \frac{\partial f(x_1,x_2)}{\partial x_1} \cdot \Delta x_1 + \frac{\partial f(x_1,x_2)}{\partial x_2} \cdot \Delta x_2$$



- Directional derivative to an arbitrary direction **u** (**u** is unit vector) is the slope of f in that direction at point $\mathbf{x}\ (x_1, x_2)$:

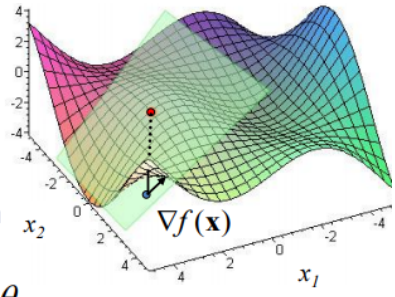
$$\mathbf{u}^{\mathrm{T}} \nabla f(\mathbf{x})$$

- f decreases the fastest:

Not changing with u

$$\min_{u,u^{\mathrm{T}}u=1} \mathbf{u}^{\mathrm{T}} \nabla f(\mathbf{x}) = \min_{u,u^{\mathrm{T}}u=1} \|\mathbf{u}\|_2 \|\nabla f(\mathbf{x})\|_2 \cos\theta$$

minimum at 180



- **u** is opposite to the gradient!!!

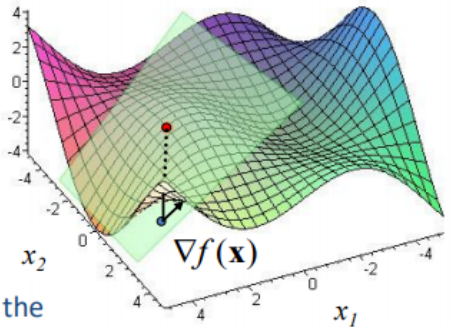
New points towards steepest descent:
$\mathbf{x}' = \mathbf{x} - \varepsilon \nabla f(\mathbf{x})$

Gradient Descent in multidimensional input case

- Steepest gradient descent iteration

$$\mathbf{x}(n+1) = \mathbf{x}(n) - \varepsilon \nabla f(\mathbf{x}(n))$$

- ε is the learning rate
- Choosing ε:
  - Small constant
  - Decreases as the iteration goes ahead
  - **Line search**: checked with several values, and the one selected, where $f(\mathbf{x})$ is the smallest
- Stopping condition of the gradient descent iteration
  - When the gradient is zero or close to zero



Egy kis extra ide:

Jacobean Matrix

- Partial derivative of a vector → vector function
- Specifically, if we have a function $\mathbf{f}: \Re^m \to \Re^n$ then the Jacobian matrix $\mathbf{J} \in \Re^{n \times m}$ of $\mathbf{f}$ is defined such that: $\mathbf{J}_{i,j} = \frac{\partial}{\partial x_j} f(x_i)$

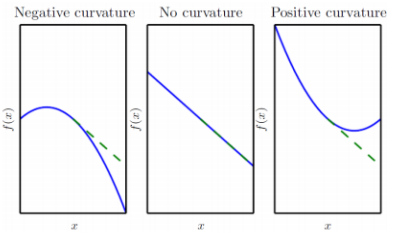
$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

2nd derivatives

- 2nd derivative determines the curvature of a line in 1D
- In nD, it is described by the Hessian Matrix

$$H(f(x_{i,j})) = \frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x)$$

- The Hessian is the Jacobian of the gradient.



2nd order gradient descent method I

- 2nd derivative in a specific direction: $\mathbf{u}^T \mathbf{H} \mathbf{u}$
- Second-order Taylor series approximation to the function *f(x) around the current point* $\mathbf{x}_0$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \mathbf{H} (\mathbf{x} - \mathbf{x}_0)$$

where:
g: gradient at x_0
H: Hessian at x_0

- stepping towards the largest gradient:

$$\mathbf{x}_0 - \varepsilon \mathbf{g} \approx \mathbf{x} \quad \to \quad \mathbf{x} - \mathbf{x}_0 \approx -\varepsilon \mathbf{g}$$

$$f(\mathbf{x}) \approx f(\mathbf{x}_0 - \varepsilon \mathbf{g}) \approx f(\mathbf{x}_0) - \varepsilon \mathbf{g}^T \mathbf{g} + \frac{1}{2} \varepsilon^2 \mathbf{g}^T \mathbf{H} \mathbf{g}$$

2nd order gradient descent method II

- Analyzing: $f(\mathbf{x}_0 - \varepsilon \mathbf{g}) \approx f(\mathbf{x}_0) - \varepsilon \mathbf{g}^T \mathbf{g} + \frac{1}{2} \varepsilon^2 \mathbf{g}^T \mathbf{H} \mathbf{g}$

Original value | Expected improvement | Correction due to curvature

- When the third term is too large, the gradient descent step can actually move uphill.
- When it *is zero or negative,* the Taylor series approximation predicts that increasing ε *forever will decrease f* forever.
- In practice, the Taylor series is unlikely to remain accurate for large ε, *so* one must resort to more heuristic choices of ε *in this case.*
- *When it is positive,* solving for the optimal step

$$\varepsilon^* = \frac{\mathbf{g}^T \mathbf{g}}{\mathbf{g}^T \mathbf{H} \mathbf{g}}$$

(+itt már folyt a newton opt.)

d. Weight regularization (L1, L2)

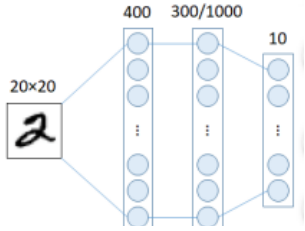
L1 and L2 regularization

- L1, L2, regularization modifies the weights
  - Rather than using MSE cost function
  - An extra term, the MSE of the weights is added (biases excluded)
  - Done on minibatch level
- Can be used with other cost function type as well
- Differenciable: back propagation works
- Why is it good?
  - Network preffers smaller weights
  - If a few large weights dominate the decision the network will lose fine generalization properties
  - In case of large weigths, the decisions are less distributed, the network is less error tolerant

$$C_0 = \frac{1}{2n}\sum_x \|y - a^L\|^2$$

$$C_{L_1} = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{n}\sum_w |w|$$

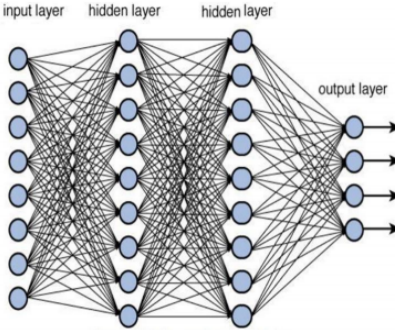
$$C_{L_2} = \frac{1}{2n}\sum_x \|y - a^L\|^2 + \frac{\lambda}{2n}\sum_w w^2$$



10/8/2019

Data regularization techniques

- Modification of the input vectors or the internal data composition of the network
  - Input normalization
  - Batch normalization
- Modification of the cost function (involving the weight magnitudes)
  - L1 and L2 regularization (weight penalty)
- Temporal Modification of the net architecture in training phase
  - Dropout



10/22/2019

e. Pooling

Methods of data size reduction:

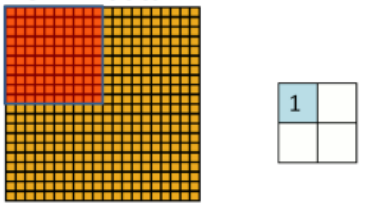
- Pooling
- Convolution with strides
- Convolution without padding

Data aggregation:

- Stride convolution, pooling

Pooling

- Pooling summarizes statistically the extracted features from the same location on a feature map
- Mathematically, it is a local function over 1D or 2D data
  - input:
    - Segment of a vector in 1D
    - rectangular neighborhood in 2D
  - Function
    - Statistical (maximum: max-pool)
    - L2 norm
    - Weighted average (weights proportional of the distance of the central element)
- In most cases: stride > 1
  - This leads to downsampling
- Pooling introduces some shift invariancy



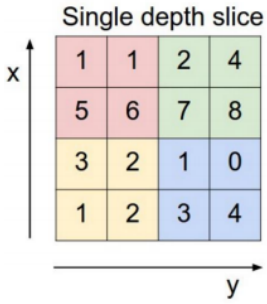
Convolved feature

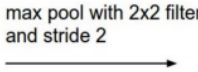
Pooled feature

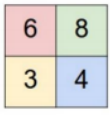
$s = 10$

Max pooling

- Max pooling is the most used pooling in CNN
- Picks the largest value from a neighborhood
- Non-linear
- Statistical filter
- Downsampling depends on the stride

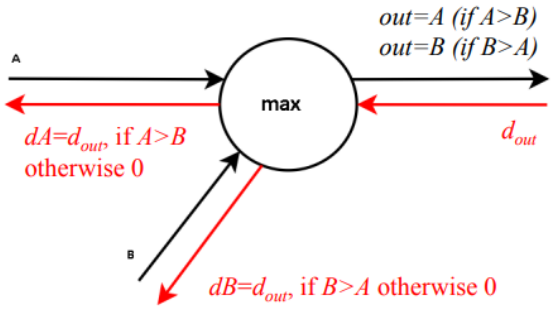


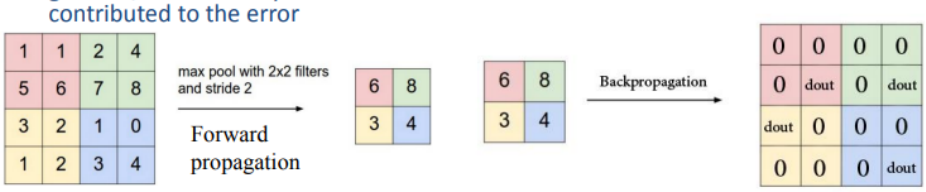




Backpropagation through max-pooling layer

- Maximum node acts as a router
- The d_{out} gradient is given to the input node, which has contributed (which was the biggest)
- The remaining positions will get zero, because they did not contributed to the error

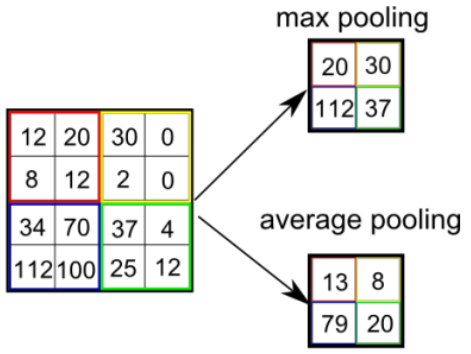


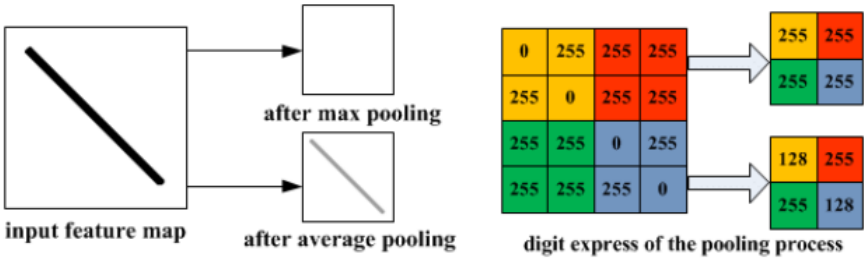


The maximum positions are stored

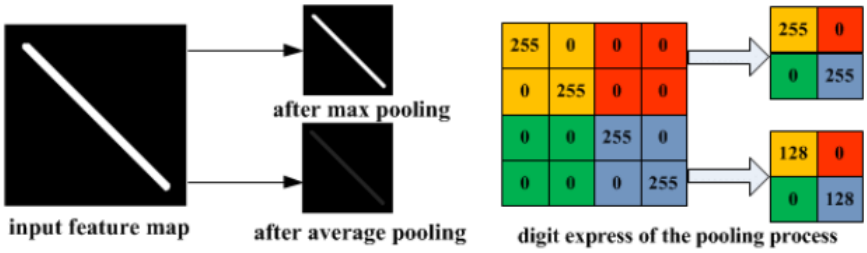
Average pooling

- Similar to max pooling, but uses the average



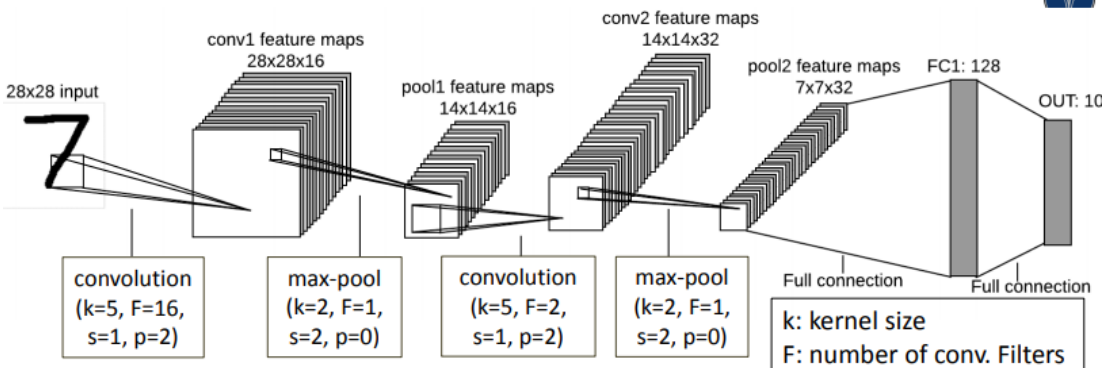


(a) Illustration of max pooling drawback



(b) Illustration of average pooling drawback

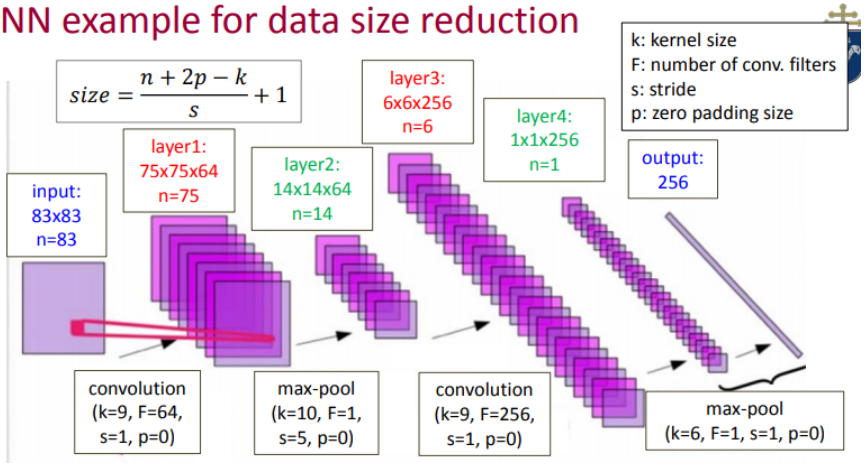
Architecture of a typical Convolution Neural Network



- Input
- Parallel feature extractors (convolution layers w. RELU)
- Data reduction (pooling)
- Combination of the features – aggregating information (fully connected layer)
- Decision (fully connected layer with soft-max activation)

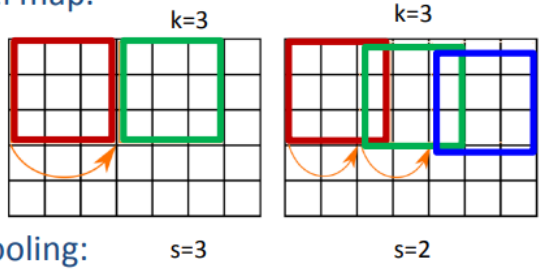
47

CNN example for data size reduction

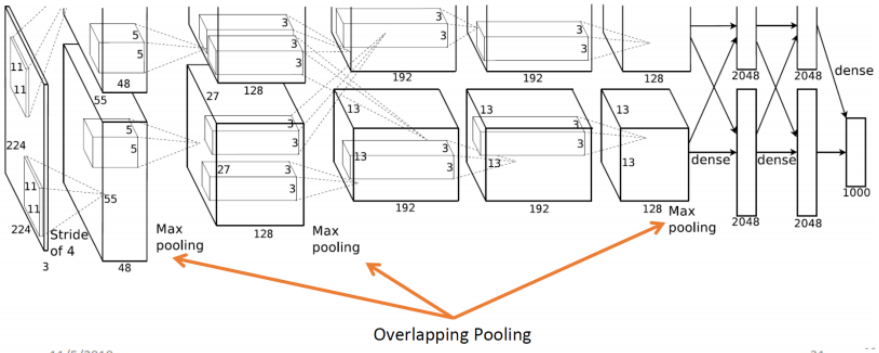


Overlapping Pooling I

- Pooling layers summarize the outputs of neighboring neurons in the same kernel map.
  - Overlapping pooling
  - s < k



- Improvement using MaxPooling:
  - Top 1 error rate by 0.4%
  - Top 5 error rates by 0.3%



11/5/2019 31

Upsampling I: "Unpooling"

Nearest Neighbor

1	2
3	4

→

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Input: 2 x 2 Output: 4 x 4

"Bed of Nails"

1	2
3	4

→

1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

Input: 2 x 2 Output: 4 x 4

Upsampling I: "Unpooling"

Max Pooling
Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

→

5	6
7	8

→ . . . → Rest of the network

Input: 4 x 4 Output: 2 x 2

Max Unpooling
Use positions from pooling layer

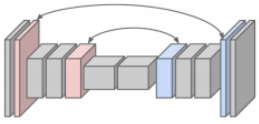
1	2
3	4

→

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

Input: 2 x 2 Output: 4 x 4

Corresponding pairs of downsampling and upsampling layers



11/5/2019 45

No. 12 Topic

a. **Perceptron convergence theorem (no proof)** - nem tudom hol kezdodik a bizonyitas.hahaha

Perceptron Convergence theorem (1)

Assumptions:

- ***w(0)=0***
- *the input space is linearly separable, therefore* $\mathbf{w}_o$ *(stands for* $\mathbf{w}_{optimal}$*) exists:*

$$x \in X^+: \quad w_o^T x > 0: \; d = 1$$
$$x \in X^-: \quad w_o^T x < 0: \; d = 0$$

- *Let us denote* $\tilde{x} = -x$

$$\tilde{x} \in \tilde{X}^-: \quad w_o^T \tilde{x} > 0: \; d = 1$$

For the proof, see also: Simon Haykins: Neural Networks and Learning Machines, *Section 1.3:* http://dai.fmph.uniba.sk/courses/NN/haykin.neural-networks.3ed.2009.pdf

Perceptron Convergence theorem (2)

- Idea:
  - During the training, the network will be activated with those input vectors (one after the other), where the decision is wrong, hence non zero adaptation is needed:

$$x(j) \in X^+: \quad w^T(j)x(j) < 0, \quad y = 0, \; d = 1$$
$$x(j) \in \tilde{X}^-: \quad w^T(j)x(j) < 0, \quad y = 0, \; d = 1$$

  - Note: The error function is always positive ($\varepsilon = 1$)

Perceptron Convergence theorem (3)

- According to the learning method:
- $w(n+1)=w(0)+\eta x(0)+\eta x(1)+\eta x(2)+\eta x(3)+\ldots+\eta x(n)$
  - where

$$x(j) \in X^+: \quad w^T(j)x(j) < 0, \quad y = -1, \; d = 1$$

or

$$x(j) \in \tilde{X}^-: \quad w^T(j)x(j) < 0, \quad y = -1, \; d = 1$$

  - The decision boundary will be:

$$\eta w^T x = 0$$

which means that η is a scaling factor, therefore it can be choosen for any positive number.

Let us use $\eta=1$, *therefore* $\eta\varepsilon=1$

- We will calculate $\|w(n+1)\|^2$ in two ways, and give an upper and a lower boundary, and it will turn out that an n_{max} exists, and beyond that the lower boundary is higher than the upper boundary (squeeze theorem, sandwitch lemma (*közrefogási elv, rendőr elv*))

lower limit (1)

According to the learning method, the presented input vectors are added up:

$$w(n+1) = w(0) + x(0) + x(1) + \ldots + x(n)$$ w(0)=0

Multiply it with w_o^T from the left:

$$w_o^T w(n+1) = w_o^T x(0) + w_o^T x(1) + \ldots + w_o^T x(n)$$

$$0 < \alpha \le w_o^T x(j)$$ Because each input vector (or its opposite) were selected that way.

$$0 < \alpha = \min_{x(n) \in \{X^+, \tilde{X}^-\}} w_o^T x(n)$$

$$w_o^T w(n+1) \ge n\alpha$$

lower limit (2)

$$w_o^T w(n+1) \ge n\alpha$$ We apply Cauchy Schwarty inequality $\|a\|^2 \|b\|^2 \ge \|a^T b\|^2$

$$\|w_0^T\|^2 \|w(n+1)\|^2 \ge \|w_o^T w(n+1)\|^2 \ge n^2\alpha^2$$

Lower limit:

$$\|w(n+1)\|^2 \ge \frac{n^2\alpha^2}{\|w_0^T\|^2}$$ Lower limit proportional with n^2

upper limit (1)

Let us have a different synthetization approach of *w(n+1)*:

$$w(k+1) = w(k) + x(k)$$ for k= 0 … n

Squared Euclidian norm:

$$\|w(k+1)\|^2 = \|w(k)\|^2 + \|x(k)\|^2 + 2w(k)^T x(k)$$

$$w(k)^T x(k) < 0$$ Because each input vector (or its opposite) were selected that way.

$$\|w(k+1)\|^2 \le \|w(k)\|^2 + \|x(k)\|^2$$

$$\|w(k+1)\|^2 - \|w(k)\|^2 \le \|x(k)\|^2$$ for k= 0 … n

upper limit (2)

$$\|w(k+1)\|^2 - \|w(k)\|^2 \le \|x(k)\|^2$$

Telescoping sum: $\sum_{i=1}^{n}(a_{i+1} - a_i) = a_{n+1} - a_1$

Example: $\sum_{i=1}^{4}(a_{i+1} - a_i) = \cancel{a_2} - a_1 +$
$+\cancel{a_3} - \cancel{a_2} +$
$+\cancel{a_4} - \cancel{a_3} +$
$+a_5 - \cancel{a_4} =$
$= \boldsymbol{a_5 - a_1}$

Summing up the upper term:

$$\sum_{k=0}^{n}\left(\|w(k+1)\|^2 - \|w(k)\|^2\right) \le \sum_{k=0}^{n}\|x(k)\|^2$$

Note that there is a telescoping sum in the left hand side.

$\|w(n+1)\|^2 - \|w(0)\|^2$:

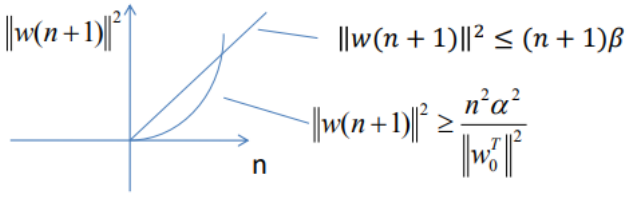
$\|w(0)\|^2 = 0$

$$\|w(n+1)\|^2 \le (n+1)\beta$$

$$0 < \beta = \max_{x(k)\in\{X^+, X^-\}}\|x(k)\|^2$$

Upper limit linearly proportional with n

comparing upper and lower limits



Linear upper limit and squared lower limit cannot grow unlimitedly

n_{max} should exist

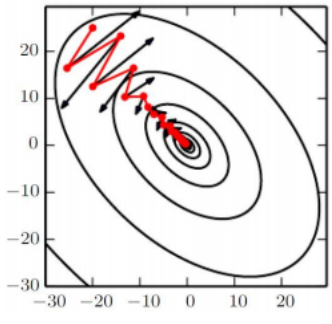
$$n_{max} = \frac{\beta\|w_0\|^2}{\alpha^2}$$

---

b. Momentum optimizer

Momentum I

- Introduced in 1964
- Physical analogy
- The idea is to simulate a unity weight mass
- It flows through on the surface of the error function
- Follows Newton's laws of dynamics
- Having *v* velocity
- Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon.



Momentum II: velocity considerations

The update rule is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$.
The larger α is relative to ϵ, the more previous gradients affect the current direction.

Terminal velocity is applied when it finds descending gradient permanently:

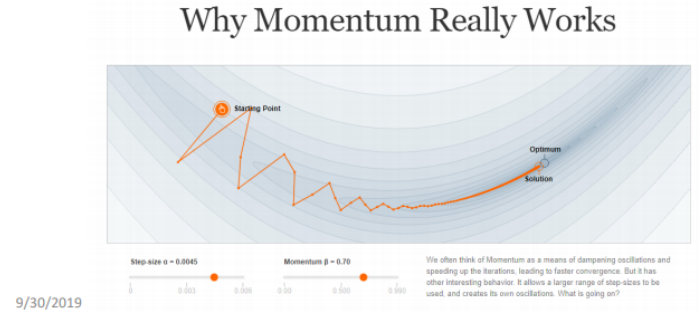
$$\frac{\epsilon ||\boldsymbol{g}||}{1 - \alpha}$$

Algorithm Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Momentum demo

- What does the parameter of the momentum method means, and how to set them?
  - https://distill.pub/2017/momentum/



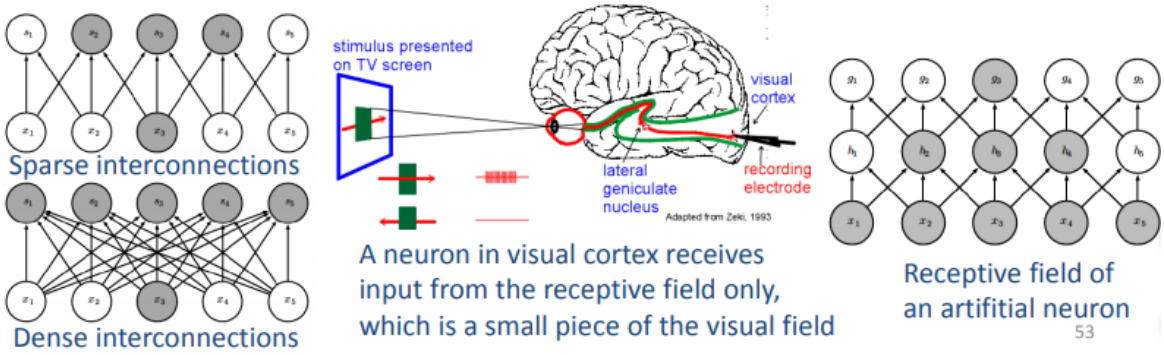
Algorithms with changing but not adaptive learning rate:
– Stochastic Gradient Descent algorithm
– Momentum algorithm
– Nesterov momentum update

itt tuti van valami hasznos még:
https://mlfromscratch.com/optimizers-explained/#/

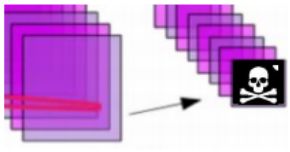
c. **Properties of CNN: sparsity, parameter sharing, equivariance, invariance to shifting** -> mi a kutyafule

Properties of Convolutional Neural Networks I:

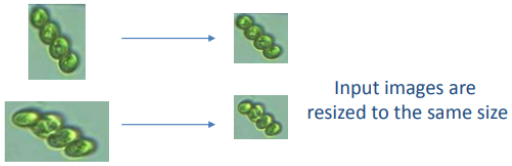
- **Sparsity**
  - The interconnection weights are just a fraction of the fully connected NN (the weight matrix between two layers are sparse)
  - A few dozen free parameter describes the operation of a layer
  - Receptive field organization similar to natural neural vision systems



- **Parameter sharing**
  - Same parameters everywhere in the layer
  - Contribution to the gradient of a weight from many positions
  - Reduces the risk of overfitting
  - Reduces the risk of dying RELU (dying cell)
    - When it happens, an entire feature extractor on a layer is dying



- Variable input size
  - The input image is either resized or padded



Equivalent representation:

- Equvariance to translation
  - The output shifts with an input shift
- In a fully connected neural network, each input is a dedicated channel for a certain input parameter-therefore the inputs cannot be swapped
  - Like bank example, one cannot replace the age input with the salary input
- In CNN, the image can be shifted, because the inputs are not dedicated and the features are identified anywhere

Space invariance means here that the functionality of a 2D function is not changing in space. This enables the detection of a certain image feature anywhere on the image.

---

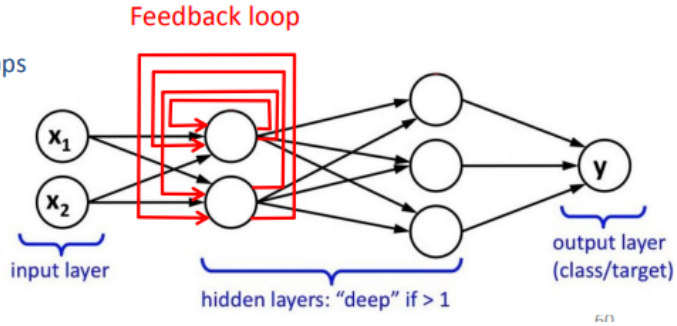
d. RNN examples: predicting the next letter, image captioning

Recurrent networks (RNN)

- Unlike traditional neural networks, the output of the RNN depends on the previous inputs
  - State
- RNN contains feedback
- Theoretically:
  - Directed graph with cyclic loops
- From now, time has a role in execution
  - Time steps, delays

Jürgen lives in Berlin.

He speeks .....



11/19/2019

60

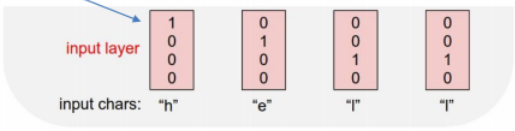
Simple RNN Training Example: Predicting the next letter

Example:
Character-level
Language Model

Vocabulary:
[h,e,l,o]

One-hot encoding

Example training sequence:
"hello"

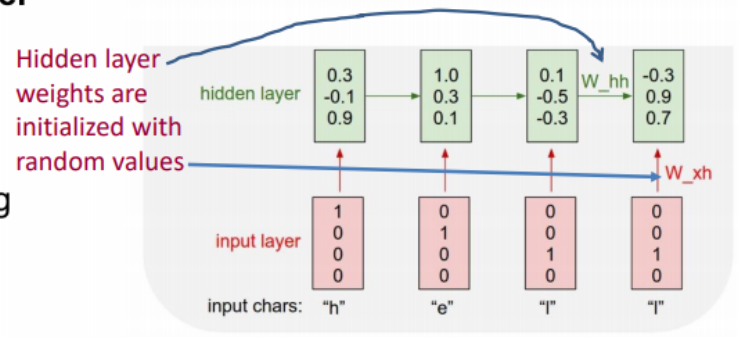


Example: Character-level Language Model

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Vocabulary: [h,e,l,o]

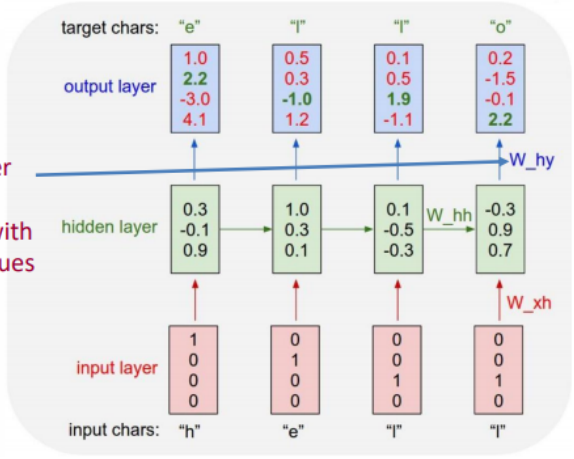
Example training sequence: **"hello"**



Example: Character-level Language Model

Vocabulary: [h,e,l,o]

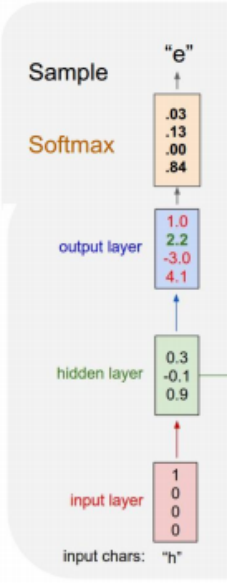
Example training sequence: **"hello"**



Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

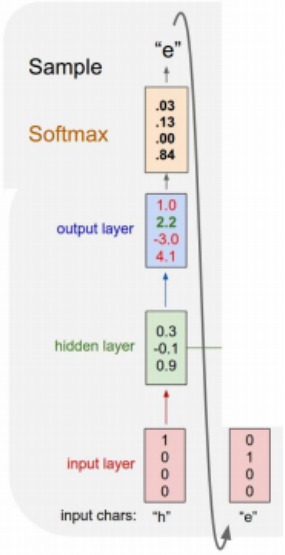
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

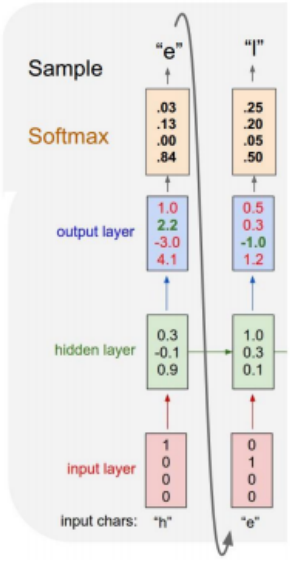
At test-time sample characters one at a time, feed back to model



Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model

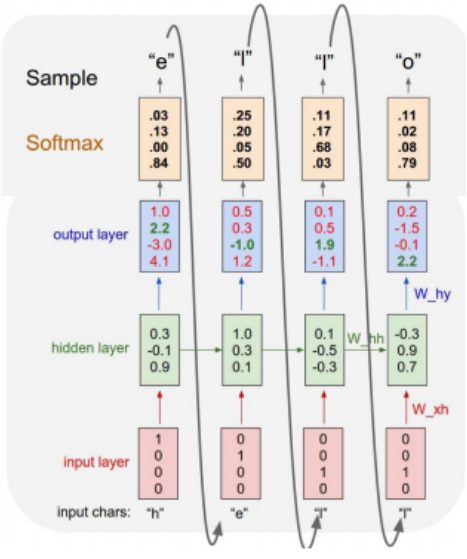


Example: Character-level Language Model Sampling

Backpropagation can be started using negative log likelihood cost function

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



2019-11-25

Image captioning example

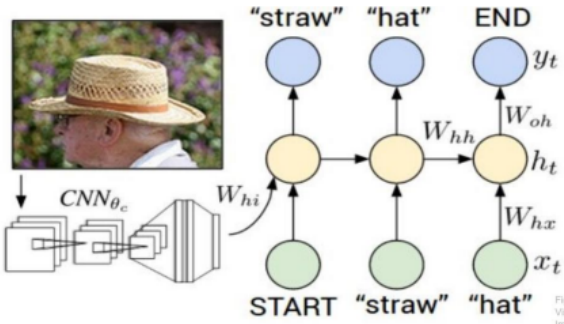
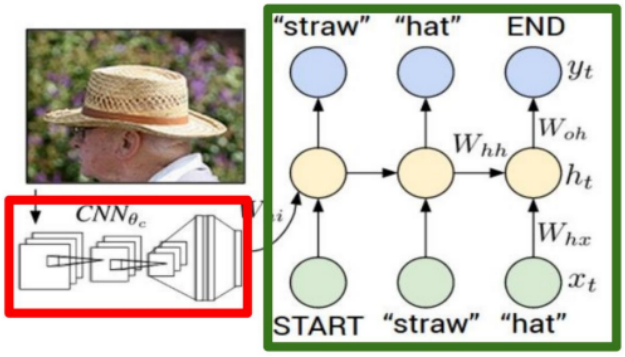


Figure from Karpathy et a, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015. Reproduced for educational purposes.

Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Image captioning example

Recurrent Neural Network



Convolutional Neural Network

2019-11-25

Alexnet: scored 5 best guesses

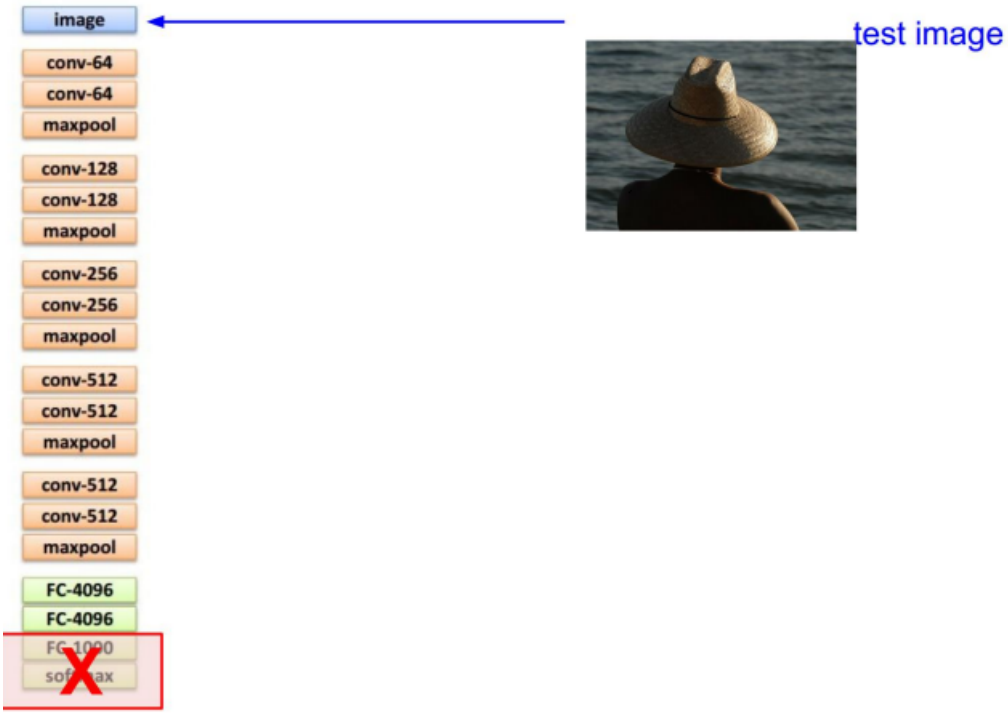


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
FC-1000
softmax
test image

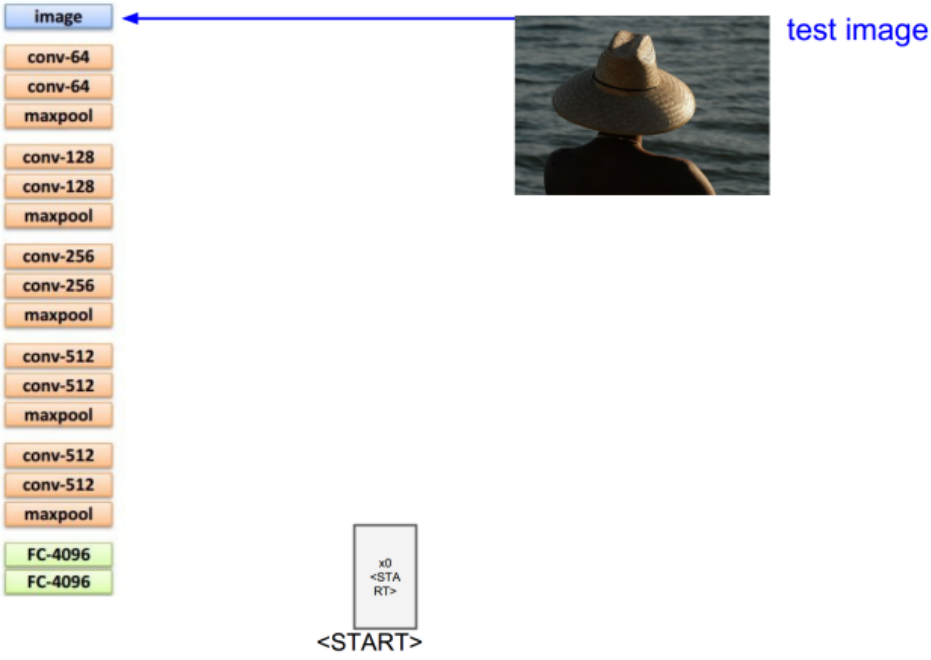


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image

x0
<STA
RT>

<START>

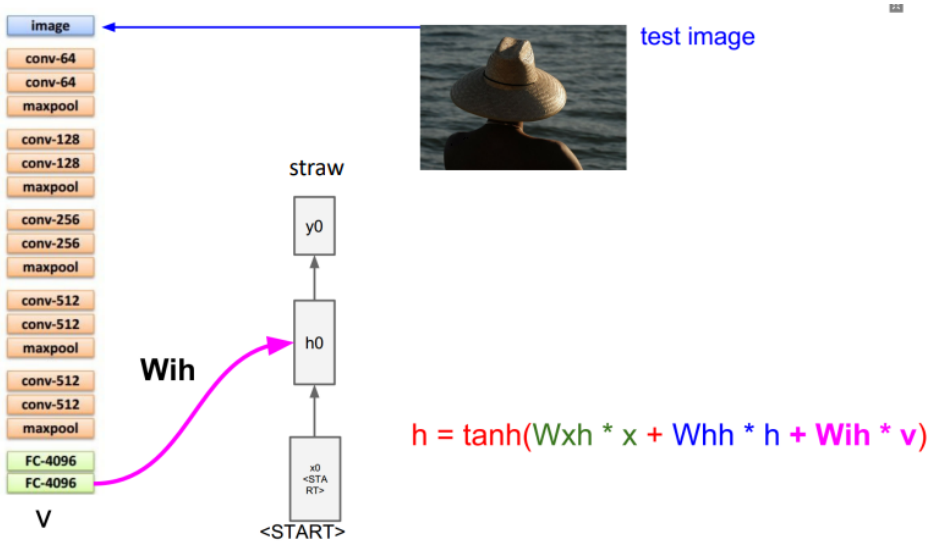


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
V
Wih
straw
y0
h0
x0
<STA
RT>
<START>
test image
h = tanh(Wxh * x + Whh * h + Wih * v)

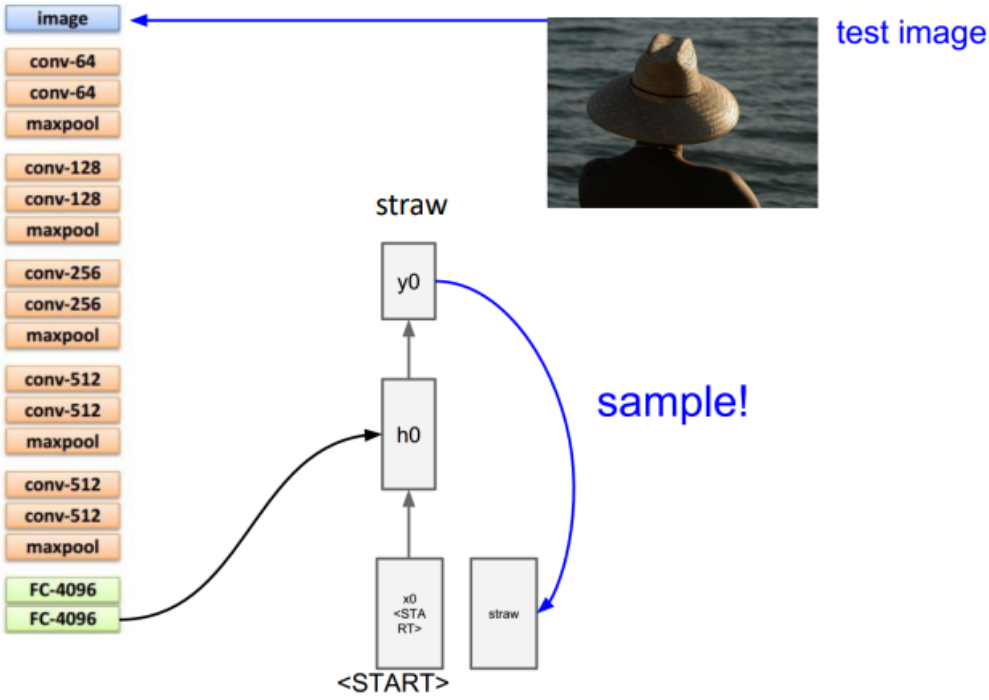


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
straw
y0
h0
x0
<STA
RT>
<START>
straw
sample!
test image

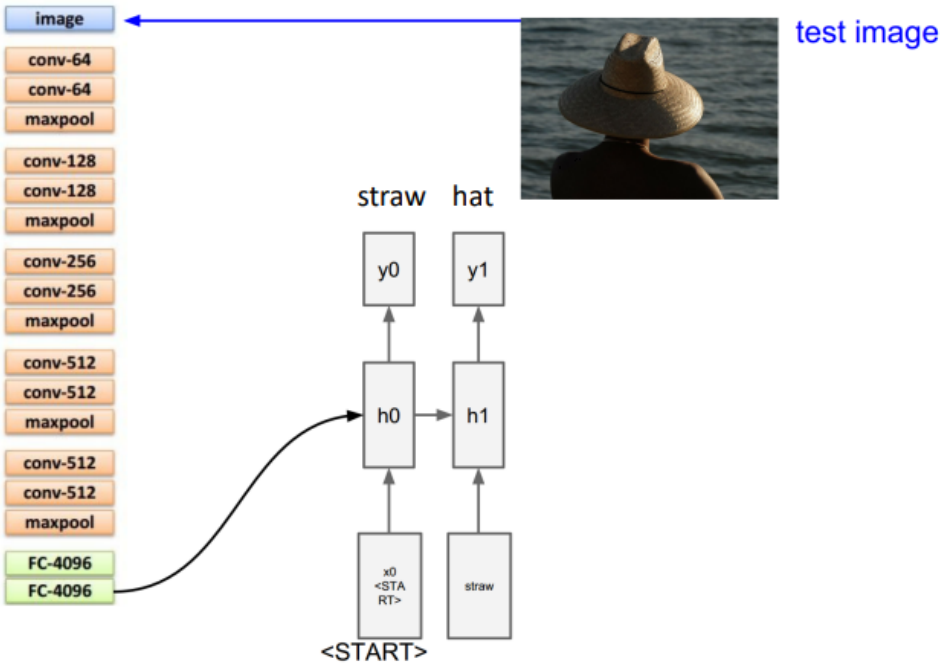


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
y0
y1
h0
h1
x0
<STA
RT>
straw
<START>

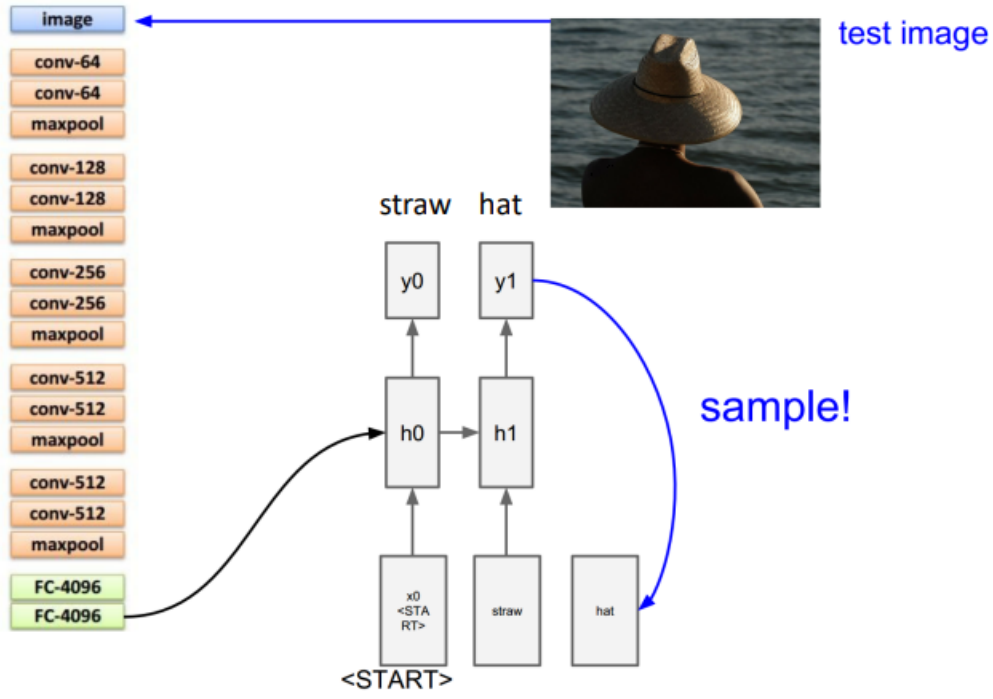


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
y0
y1
h0
h1
sample!
x0
<STA
RT>
straw
hat
<START>

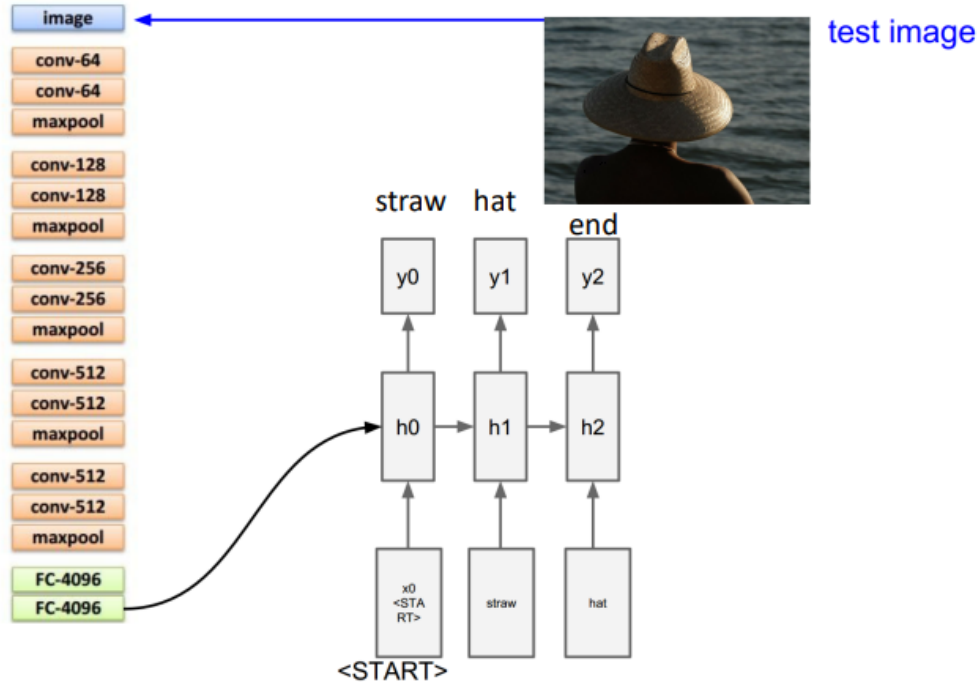


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
end
y0
y1
y2
h0
h1
h2
x0 <STA RT>
straw
hat
<START>

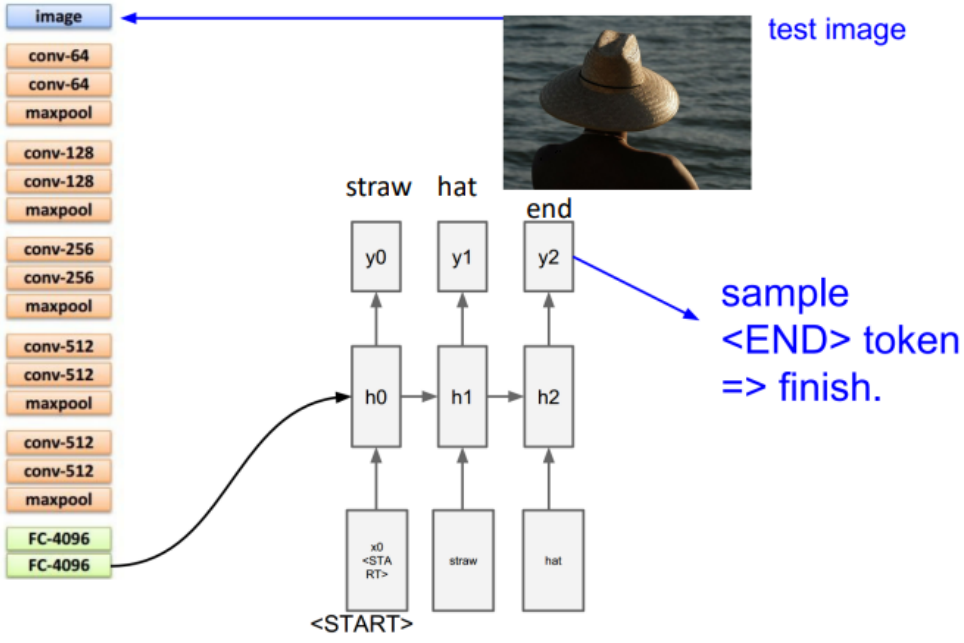
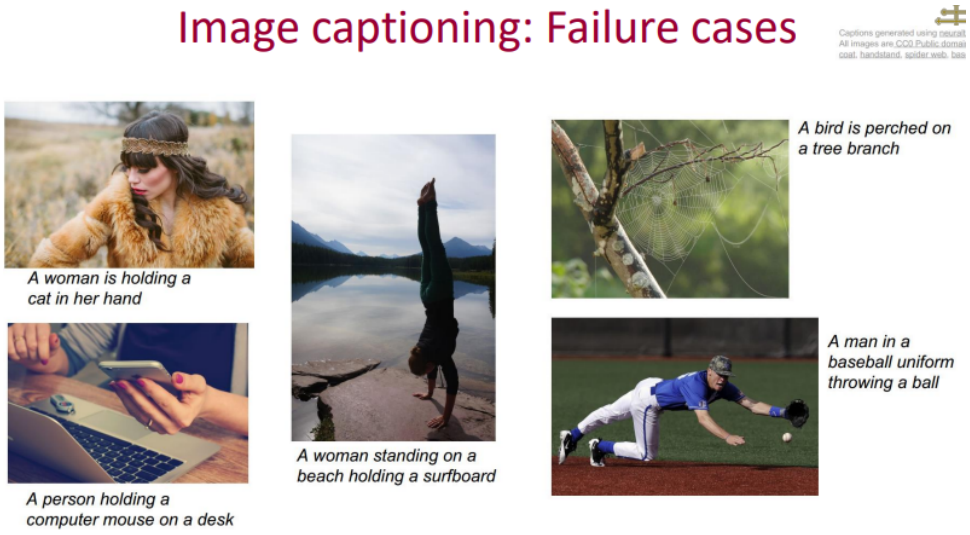


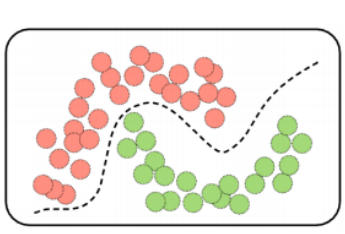
image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
straw
hat
end
y0
y1
y2
h0
h1
h2
x0 <STA RT>
straw
hat
<START>
sample
<END> token
=> finish.

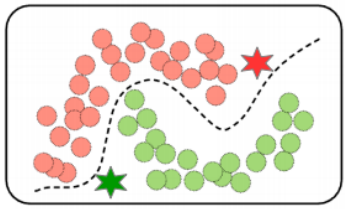


---

e. Adversarial attacks

We have a high number of parameters to be optimized
An even higher-dimensional input
The network works well in practice, but can not cover all the possible inputs

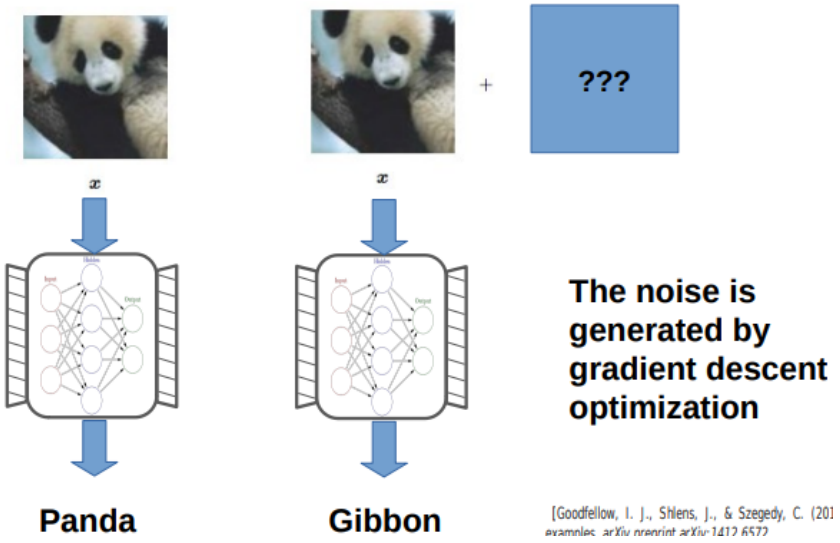




One can exploit that there will be regions in the input domain, which were not seen during training

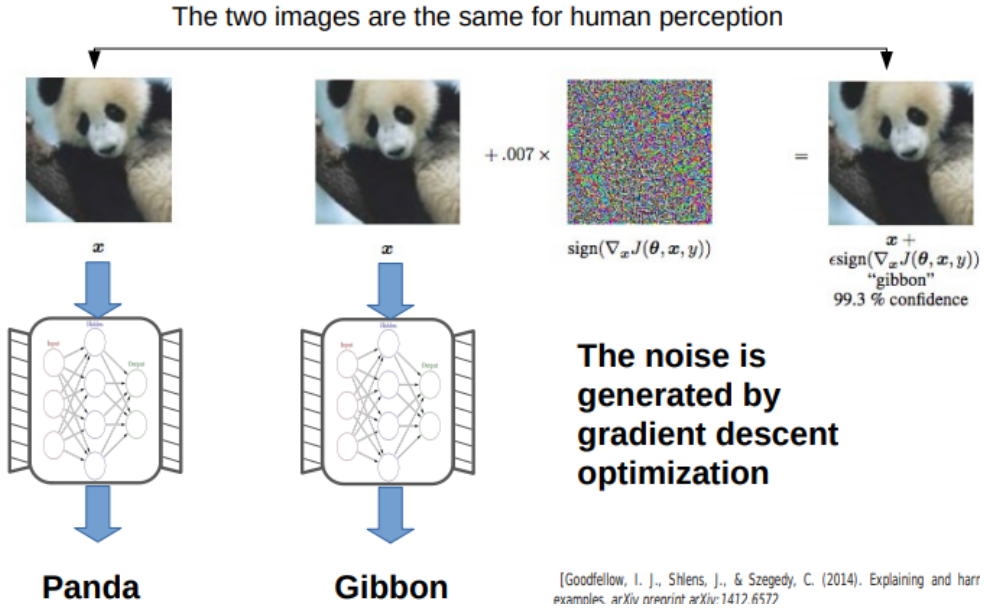
Adversarial noise:

- I have a working well-trained classifier:

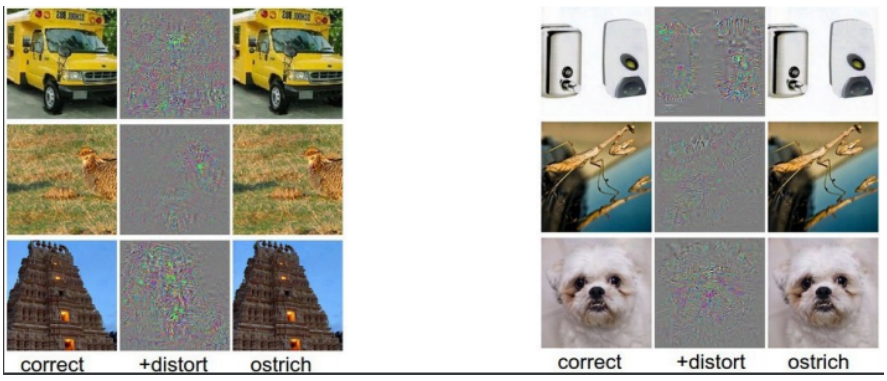


What should I add to the input to cause misclassification:

A special, low amplitude additive noise:



Knowing a trained network one can identify modifications (which does not happen in real life), which change the network output completely

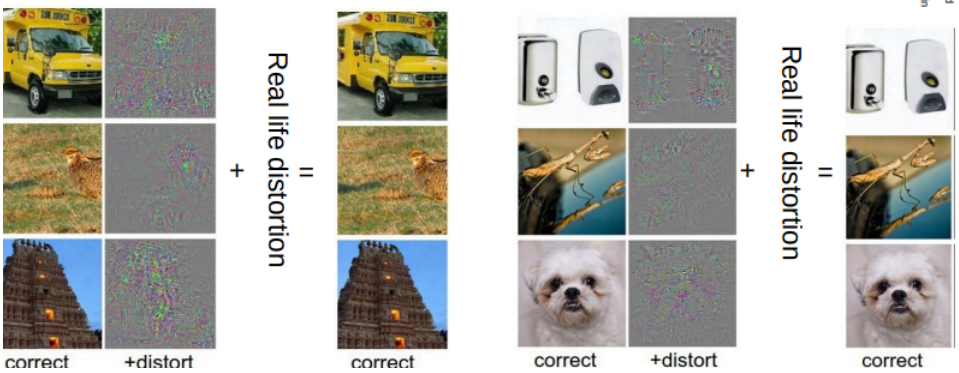


Adversarial noise – does not work in practice

Knowing a trained network one can identify modifications (which does not happen in real life), which change the network output completely

Luckily this low amplitude noise is not robust enough in real life (lens distortion and other additive noises)

Jiajun Lu, Hussein Sibai, Evan Fabry, and David Forsyth. No need to worry about adversarial examples in object detection in autonomous vehicles. 2017. URL https://arxiv.org/abs/1707.03501.g



Sticker based adversarial attacks

High intensity noise concentrated on a small region of the input image:

$$C_d = N\left(I + \sum_{i=1}^{k} St_i(x_i, y_i, w_i, h_i) + \sum_{j=1}^{l} St_j(x_j, y_j, w_j, h_j)\right)$$

Parameters are the positions (x,y) and size (w,h) of the stickers

It was shown that these attacks are **robust** enough to be applied in practical applications

Does this mean that convolutional neural networks can not be used in critical problem in practice anymore?

No. 13 Topic

a. Elementary set separation by a single neuron

Elementary set separation by a single neuron (1)

- Let us use φ(.) step nonlinear function for siplicity:

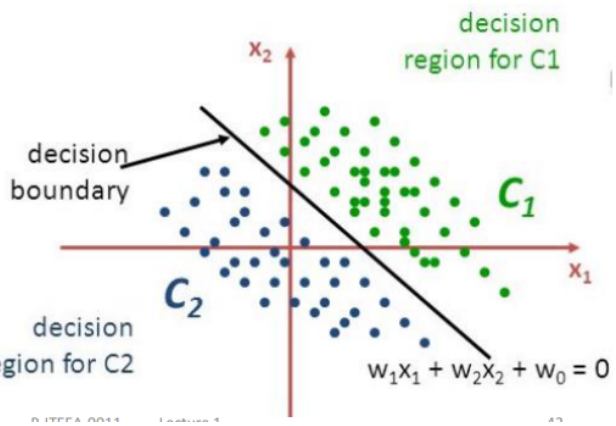
$$y = \varphi(u) = \frac{sign(u)}{2} + \frac{1}{2} = \begin{cases} 1, \text{ if } u \geq 0 \\ 0, \text{ else} \end{cases}$$

- The output of the neuron will be binary:

$$y = \varphi(u) = \frac{sign(\mathbf{w}^T\mathbf{x})}{2} + \frac{1}{2} = \begin{cases} 1, \text{ if } \mathbf{w}^T\mathbf{x} \geq 0 \\ 0, \text{ else} \end{cases}$$

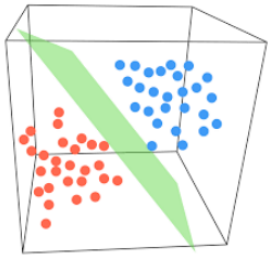
DECISION!

- in a 2-D input space, the hyper plane is a straight line.
- Above the line is classified: +1 (C1: yes)
- Below the line is classified : 0 (C2: no)



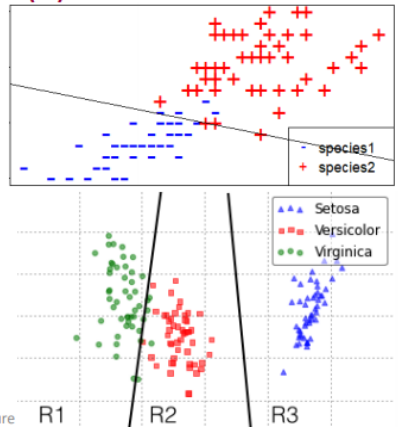
- Neuron with *m* inputs has an *m* dimensional input space
- Neuron makes a linear decision for a 2 class problem
- The decision boundary is a hyperplane defined:

$$\mathbf{w}^T\mathbf{x} = 0$$



Why it is so important to use set separation by hyper plane? (1)

- Most logic functions has this complexity (OR, AND)
- There are plenty of mathematical and computational task which can be derived to a set separation problem by a linear hyper plane
- Application of multiple hyper plane provides complex decision boundary

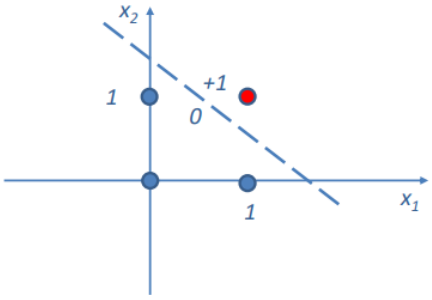


Implementation of a single logical function by a single neuron (1)

AND

x_1	x_2	$x_1 x_2$
0	0	0
0	1	0
1	0	0
1	1	1

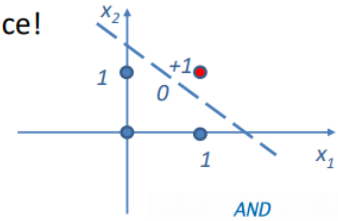
- The truth table of the logical AND function.



- 2-D AND input space and decision boundary

- We need to figure out the separation surface!
- Mathematically is the following equation:

$$-1.5 + x_1 + x_2 = 0$$



w_0=-1.5; w_1=1; w_2=1;

- The weight vector is:

w = (−1.5, 1, 1).

$$y = \frac{sign(u)}{2} + \frac{1}{2} = \begin{cases} 1, & if\ u \geq 0 \\ 0, & else \end{cases}$$

$$u = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

$$x_0 = 1$$

AND

x_1	x_2	$x_1 x_2$
0	0	0
0	1	0
1	0	0
1	1	1

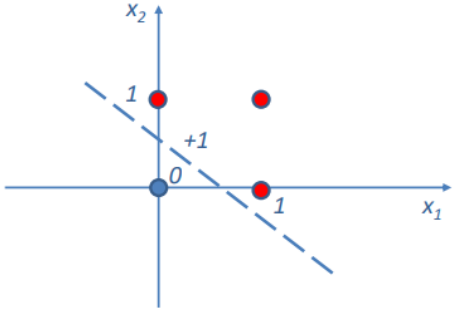
- Furthermore instead of 2D, we can actually come up with the *R* dimensional AND function.
- The weights corresponding to the inputs are all 1 and threshold should be *R* − 0.5. As a result the actual weights of the neuron are the following:

$$\mathbf{w}^{\mathrm{T}} = \left(-(R-0.5), 1, \ldots, 1\right)$$

OR

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

- The truth table of the logical OR function.



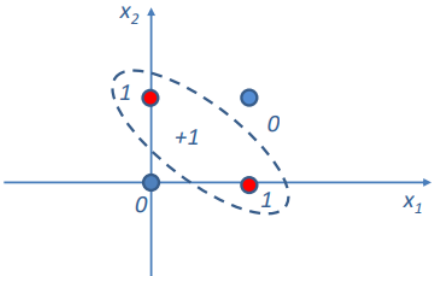
- 2-D OR input space and decision boundary

w =(−0.5, 1, 1).

- However we cannot implement every logical function by a linear hyper plane.
- Exclusive OR (XOR) cannot be implemented by a single neuron (linearly not separable)

XOR

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0



---

b. Local response normalization

- Regularization and normalization methods
  - Local response normalization
  - Data augmentation
  - Early stopping
  - Ensembling

Regularization and optimization methods

Different methods to increase the loss in the learning phase, but reduce overfitting and increase generalization capabilities
- Local response normalization
- Batch normalization
- Data augmentation (Enriching the data set)
- Early stopping
- Ensemble methods
  - Network duplication
  - Bagging
  - Dropout

Ian Goodfellow: regularization is *"any modification we make to the learning algorithm that is intended to reduce the generalization error, but not its training error"*

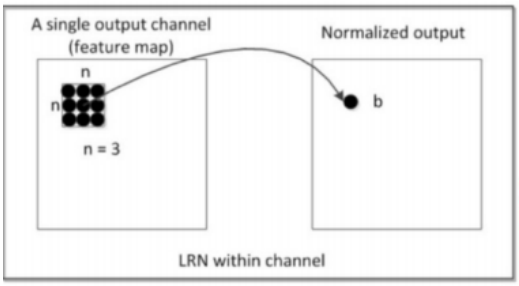
Local response normalization I

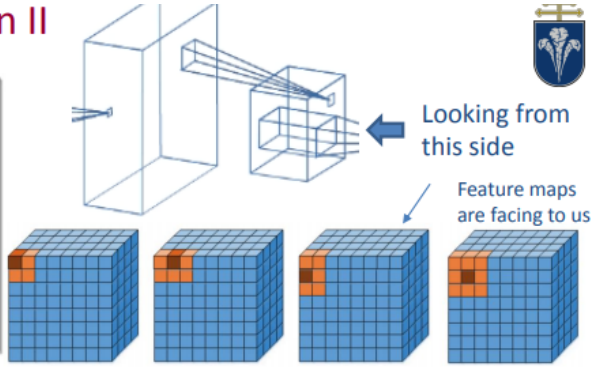
- Implementation of the Lateral inhibition from neurobiology
  - If a neuron starts spiking strongly in a layer it inhibits (suppresses) the of the neighboring cells
  - Winner take all (have a strong decision)
  - Balances the asymmetric responses of neurons in different areas of the layer
- Useful when we are dealing with ReLU neurons
  - Normalizes the unbounded activation of the ReLU neurons
    - Avoids concentrating and delivering large values through layers
  - It enhances high spatial frequencies by suppressing the local neighbors of the strongest neuron

11/5/2019

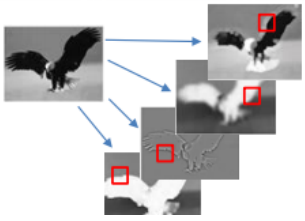
https://towardsdatascience.com/difference-between-local-response-normalization-and-batch-normalization-272308c034ac

Local response normalization II



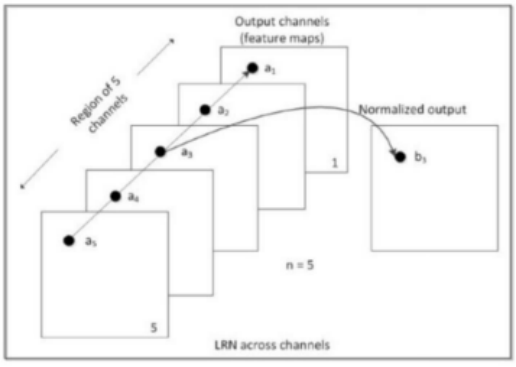


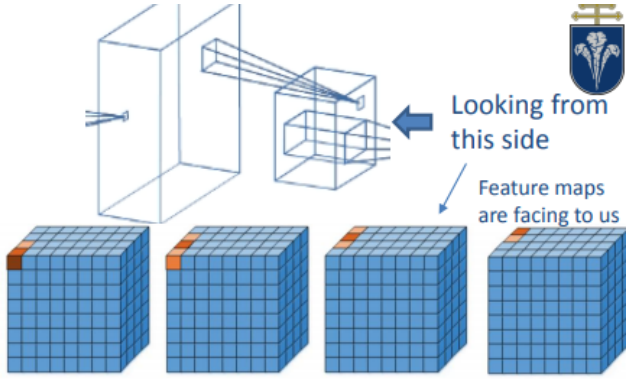
- Intra map normalization
  - 2D normalization within the same feature map
  - Balancing for different areas
  - Winner-take-all for neigbouring neurons in the same feature map (strongest response to the same transformation should win)



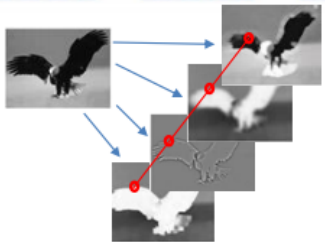
Which particular pattern responds the largest for the same transformation?

Local response normalization III





- Inter-map normalization
  - Normalization between the neighboring feature maps
  - Winner-take-all for the largest response with different transformation for the same input location

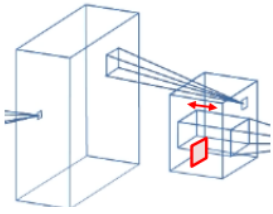


Which particular tranformation responds the largest for the same pattern?

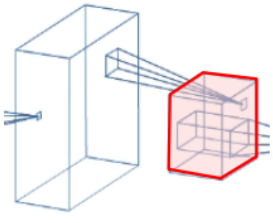
Local response norm. vs batch norm.

Both work within one convolutional layer

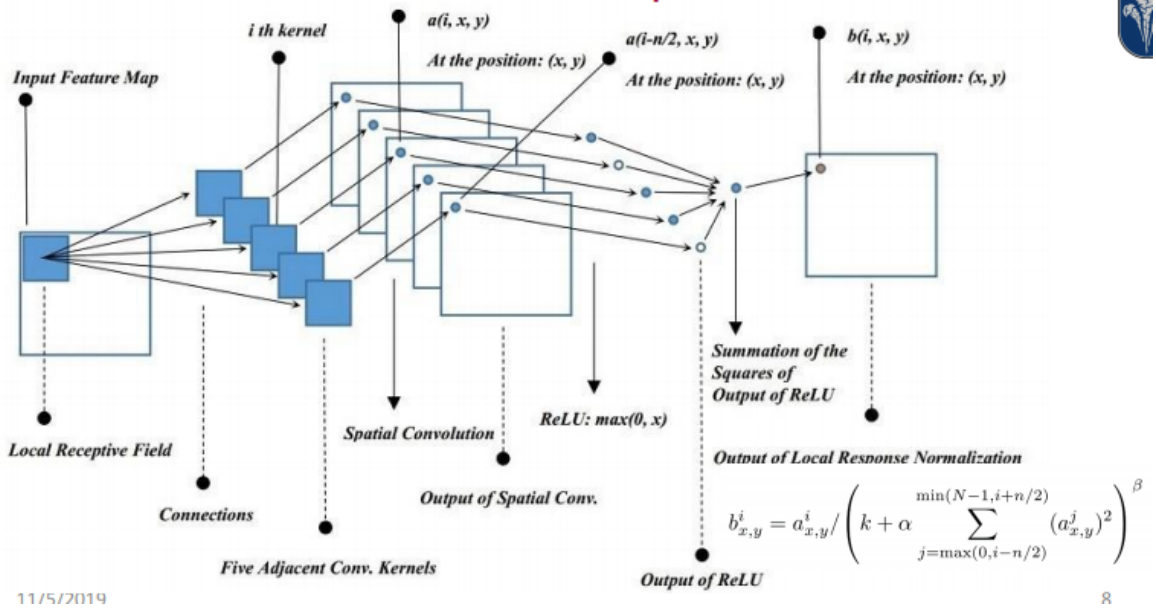
- Normalization either through the feature maps or within one feature map
- Normalization is done for one input image



- Normalization done for all the pixels in all the feature maps within a layer
- Normalization is done for the entire batch



Calculation method of local response normalization



11/5/2019 8

---

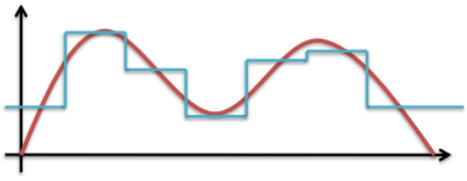
c. Unpooling -> lasd No. 11 Topic

---

d. Representations: Blum and Li theorem, construction

Representation – Blum and Li theorem

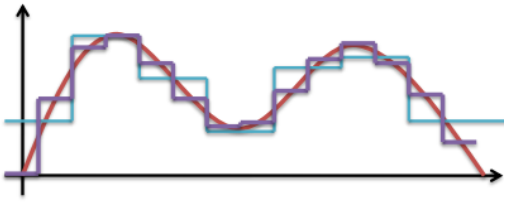
- Theorem: $F(x) \in L^2$
  $\forall \varepsilon > 0, \exists \mathbf{w}$
- Proof: $\int \cdots \int_{\mathbf{x}} \left(F(\mathbf{x}) - Net(\mathbf{x}, \mathbf{w}) \right)^2 \mathbf{d}x, \ldots \mathbf{d}x_N < \varepsilon$
  - Using the step functions: S
  - From elementary integral theory it is clear every function can be approximated by appropriate step function sequence



9/24/2019

Representation – Blum and Li theorem

- From elementary integral theory it is clear every function can be approximated by appropriate step function sequence
  - The step function can have arbitrary narrow steps
  - For example each step could be divided into two sub-steps
  - Therefore we can synthetize a function with arbitrary precision

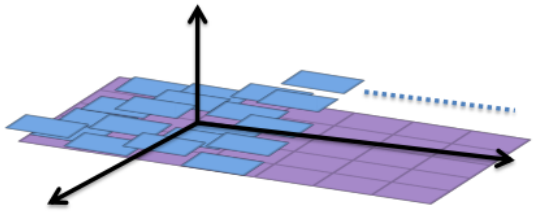


$$I(X) = \begin{cases} 1 & \text{if } \mathbf{x} \in X \\ 0 & \text{else} \end{cases}$$

$$F(x) \cong \underbrace{\sum_i F(x_i) I(x_i)}_{s(x)}$$

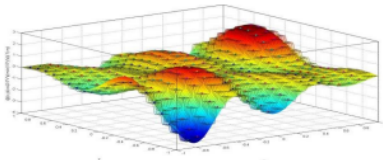
Representation – Blum and Li construction

- This construction …
  - … has no dimensional limits
  - … has no equidistance restrictions on tiles (partitions)
  - … can be further fined, and the approximation can be any precise
- 2 dimensional example
  - The tiles are the top of the columns for each approximation cell



Blum and Li – Limitations

- The size of the FFNN constructed via this method is quite big
- Consider the task on the picture, where there are 1000 by 1000 cell to approximate the function
- General case:
  ~2 Million neurons are needed
- Smoother approximation needs more
- The network architecture is synthetized (constracted), the weights are generated
- We are after to find a less complicated architectures



No. 14 Topic

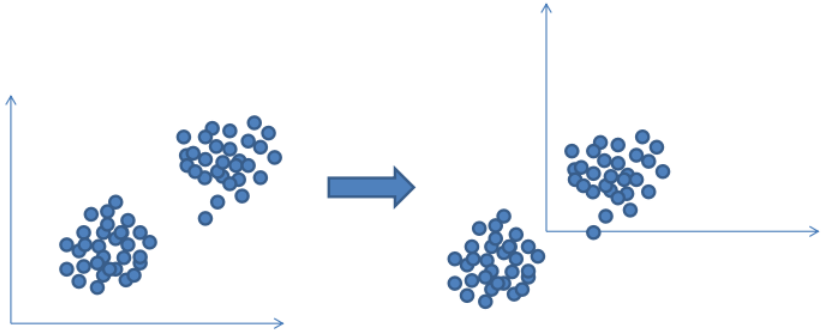
a. Principal component analysis (PCA)

Principal component analysis (PCA)

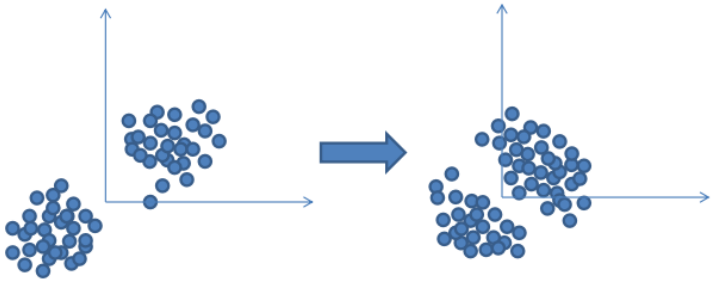
- Technique for dimensionality reduction
- Invented by Karl Pearson (1901)
- Linear coordinate transformation
  - converts a set of observations of possibly correlated variables
  - into a set of values of linearly uncorrelated orthogonal variables called principal components
- Deterministic algorithm

PCA algorithm

1. Mean normalization: For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.



2. Standardization (optional): Do it, if you want to have each of your features the same variance.



3. Covariance matrix: Calculate the covariance matrix

Covariance (formal definition)

- Assume that $\mathbf{x}$ are random variable vectors
- We have n vectors

$$\text{Variance(x)} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$$= \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})$$

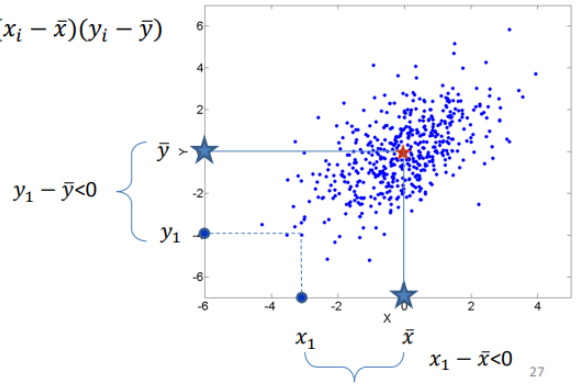
$$\text{Covariance(x, y)} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- $\text{Covariance}(x, x) = \text{var}(x)$
- $\text{Covariance}(x, y) = \text{Covariance}(y, x)$

Covariance example for 2D

$$\text{Covariance}(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

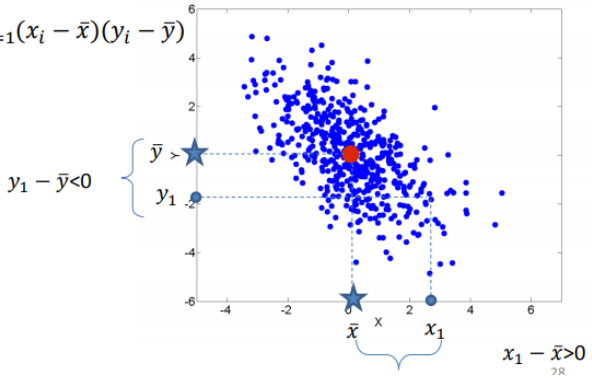
- Positive covariance between the two dimensions



Covariance example for 2D

$$\text{Covariance}(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

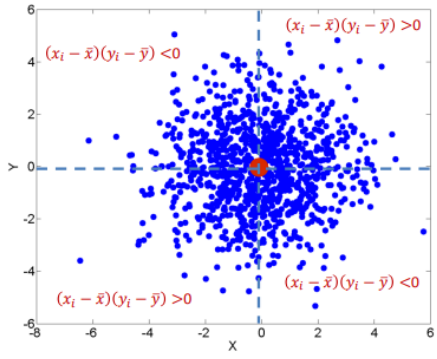
- Negative covariance between the two dimensions



Covariance example for 2D

$$\text{Covariance}(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

- No covariance between the two dimensions



Covariance matrix

- Diagonal elements are variances, i.e. $\mathrm{Cov}(x,x)=var(x)$
  - n is the number of the vectors
  - m is the dimension

$$Cov\ (\sum) = \begin{bmatrix} cov(x_1,x_1) & cov(x_1,x_2) & \cdots & cov(x_1,x_m) \\ cov(x_2,x_1) & cov(x_2,x_2) & \cdots & cov(x_2,x_m) \\ \vdots & \vdots & \vdots & \vdots \\ cov(x_m,x_1) & cov(x_m,x_2) & \cdots & cov(x_m,x_m) \end{bmatrix}$$

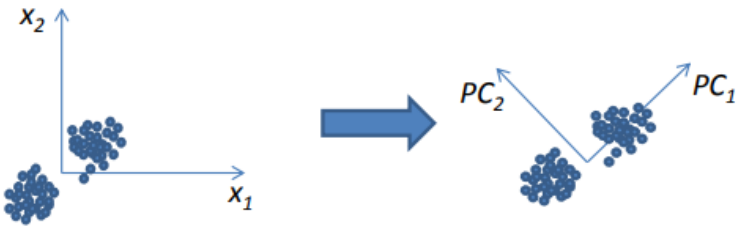
$$Cov\ (\sum) = \frac{1}{n}(X-\bar{X})(X-\bar{X})^T; where\ X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

- Covariance Matrix is symmetric
  - commutative

$$Cov\ (\sum) = \begin{bmatrix} var(x_1,x_1) & cov(x_1,x_2) & \cdots & cov(x_1,x_m) \\ cov(x_2,x_1) & var(x_2,x_2) & \cdots & cov(x_2,x_m) \\ \vdots & \vdots & \vdots & \vdots \\ cov(x_m,x_1) & cov(x_m,x_2) & \cdots & var(x_m,x_m) \end{bmatrix}$$

PCA algorithm

1. Mean normalization: For every value in the data, subtract its mean dimension value. This makes the average of each dimension zero.
2. Standardization (optional): Do it, if you want to have each of your features the same variance.
3. Covariance matrix: Calculate the covariance matrix
4. Eigenvectors and eigenvalues of the covariance matrix
   - Note: Each new axis (PC) is an eigenvector of the data. The standard deviation of the data variance on the new axis is the eigenvalue for that eigenvector.



Principal components will be orthogonal. Uncorrelated, independent!

5. Rank eigenvectors by eigenvalues
6. Keep top k eigenvectors and stack them to form a feature vector
7. Transform data to PCs:
   - New data = feature vectors (transposed) * original data

From k original variables: $x_1,x_2,\ldots,x_k$:

Produce k new variables: $y_1,y_2,\ldots,y_k$:

$y_1 = a_{11}x_1 + a_{12}x_2 + \ldots + a_{1k}x_k$

$y_2 = a_{21}x_1 + a_{22}x_2 + \ldots + a_{2k}x_k$

...

$y_k = a_{k1}x_1 + a_{k2}x_2 + \ldots + a_{kk}x_k$

y_k's are **Principal Components**

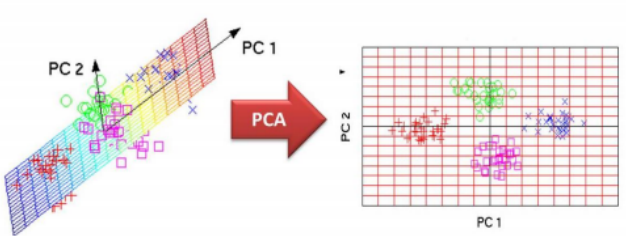
$\{a_{11},a_{12},\ldots,a_{1k}\}$ is 1st **Eigenvector** of of first principal component

$\{a_{21},a_{22},\ldots,a_{2k}\}$ is 2nd **Eigenvector** of of 2nd principal component

$\{a_{k1},a_{k2},\ldots,a_{kk}\}$ is kth **Eigenvector** of of kth principal component

Principal Component Analysis (PCA)

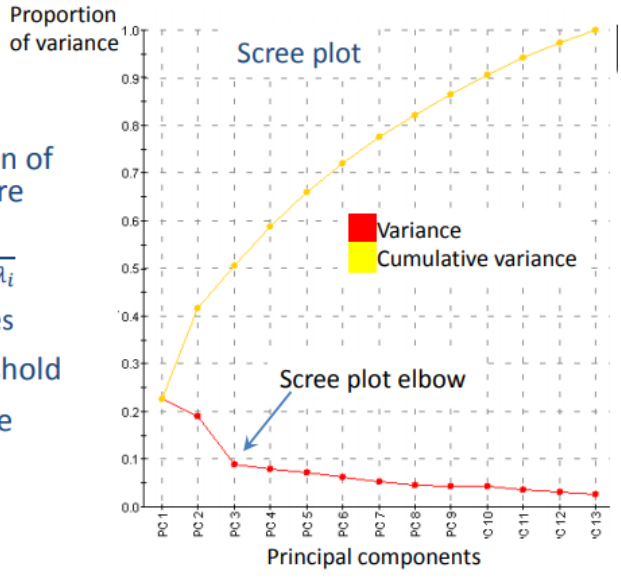
- The idea is to project the data onto a subspace which compresses most of the variance in as little dimensions as possible.
- Each new dimension is a **principle component**
- The principle components **are ordered** according to how much **variance in the data** they capture
  - Example:
    - PC1 – 55% of variance
    - PC2 – 22% of variance
    - PC3 – 10% of variance
    - PC4 – 7% of variance
    - PC5 – 2% of variance
    - PC6 – 1% of variance
    - PC7 - ....



We have to choose how many PCs to use from the top

How many PCs to use?

- Calculate the proportion of variance for each feature
  - $prop.of\ var. = \frac{\lambda_i}{\sum_{i=1}^{n}\lambda_i}$
  - λ_i are the eigen values
- Rich a predefined threshold
- Or find the elbow of the Scree plot



PCA Example

- Weekly food consumption of the four countries
  - food types: variables
  - countries: observations
- Clustering the countries:
  - Needs visualization in 17 dimension
- PCA: reduce dimensionality

http://www.sdss.jhu.edu/~szalay/class/2016-oldold/SignalProcPCA.pdf

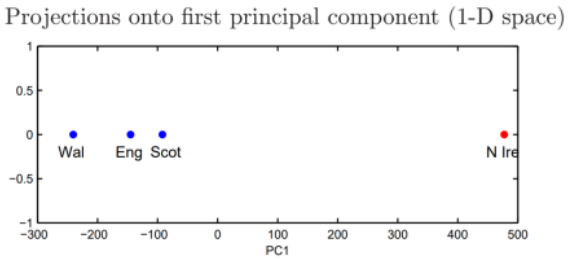
11/19/2019

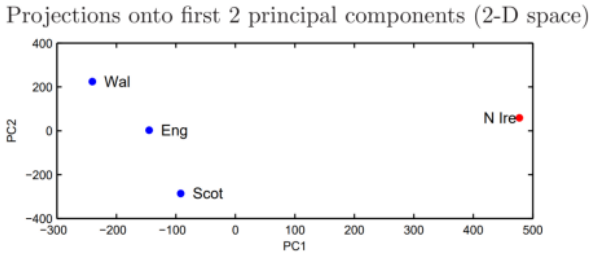
	England	Wales	Scotland	N Ireland
Cheese	105	103	103	66
Carcass meat	245	227	242	267
Other meat	685	803	750	586
Fish	147	160	122	93
Fats and oils	193	235	184	209
Sugars	156	175	147	139
Fresh potatoes	720	874	566	1033
Fresh Veg	253	265	171	143
Other Veg	488	570	418	355
Processed potatoes	198	203	220	187
Processed Veg	360	365	337	334
Fresh fruit	1102	1137	957	674
Cereals	1472	1582	1462	1494
Beverages	57	73	53	47
Soft drinks	1374	1256	1572	1506
Alcoholic drinks	375	475	458	135
Confectionery	54	64	62	41

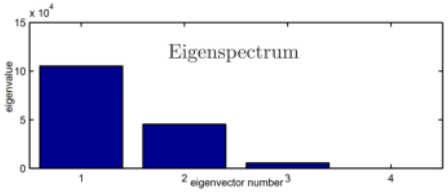
UK food consumption in 1997 (g/person/week). Source: DEFRA

PCA Example

- From PC1, two clusters are well separable
- Including PC2, the four clusters can be well separated

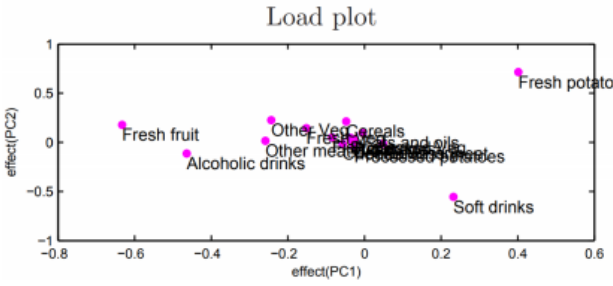


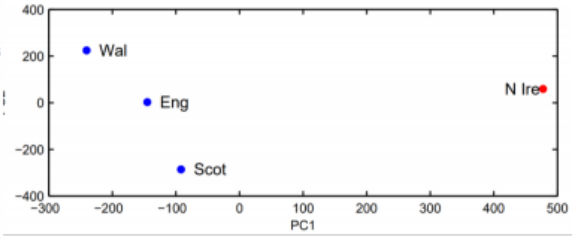




11/19/2019 37

Coefficients of the Principal Components



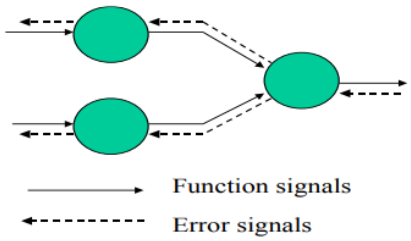


Load plot shows the coefficients of the original feature vectors to the principal components

b. **Back-propagation through time** -> shit ezt csak most vettem eszre, lent a megoldas

Learning

- The Rosenblatt algorithm is inapplicable,
  - the error and desired output in the hidden layers of the FFNN **is unknown**
- Someway the error of the whole network has to be distributed to the internal neurons, in a feedback way



Forward propagation of function signals and back-propagation of errors signals

Sequential back propagation

- Adapting the weights of the FFNN (recursive algorithm)

$$w_{ij}^{(l)}(k+1) = w_{ij}^{(l)}(k) + \Delta w_{ij}^{(l)}(k)$$

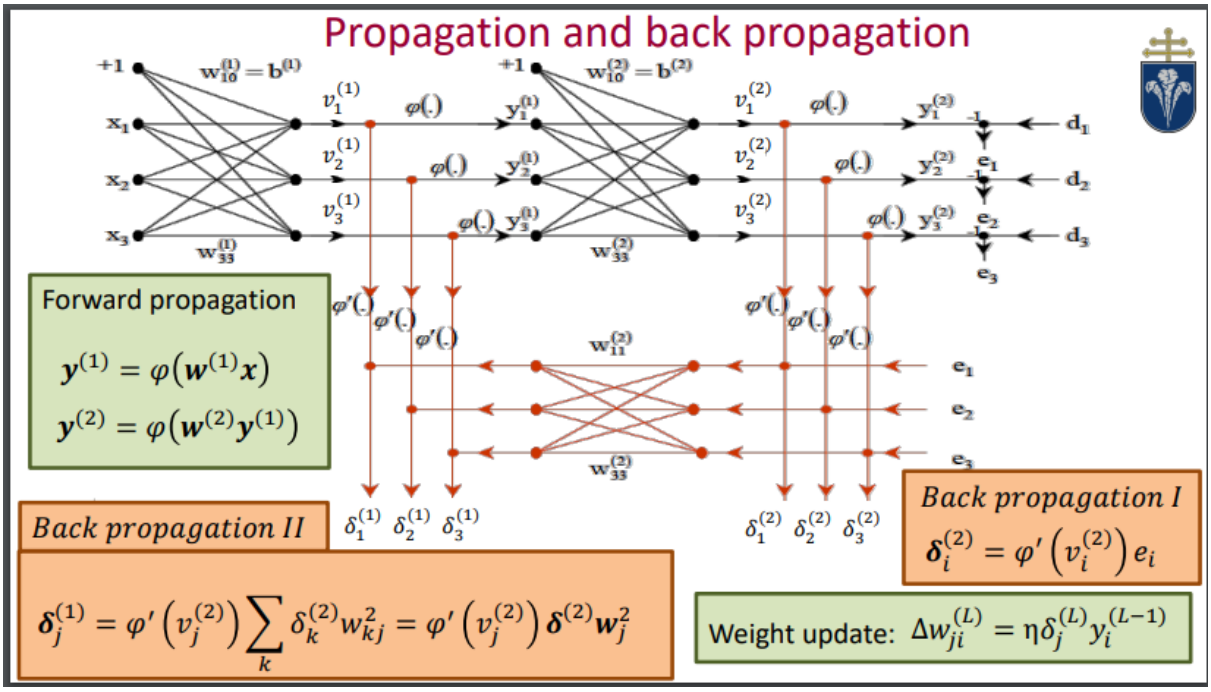
$$\Delta w_{ij}^{(l)}(k) = ?$$

- The weights are modified towards the differential of the error function (delta rule):

$$\Delta w_{ij}^{(l)} = -\eta \frac{\partial R_{emp}}{\partial w_{ij}^{(l)}}$$

- The elements of the training set adapted by the FFNN sequentially

$$R_{emp} = R_{emp}(y(\mathbf{x}), d)$$



- Though we showed how to modify the weights with back propagation, its most important value that it can calculate the gradient
- The weight updates can be calculated with different optimization methods, after the gradients are calculated
- Various optimization method can drastically speed up the training (100x, 1000x)

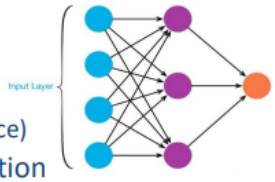
Conclusion

- For known functions (according to Blum-Li)
  – One can define a Neural Network architecture
  – And generate the weights
  – That it can represent the known function with arbitrary precision
- For unknown but existing function defined by IO pairs (according to statistic learning)
  – One can find a Neural Network architecture
  – And train the network (optimize the weights)
  – Reach arbitrary precision with high number of IO pairs
  – The trained network will be able to well predict previously unknown IO pairs (generalization)

Back propagation

- We have seen last time how to calculate the gradient in a multilayer fully connected network using back propagation
  – The introduced method was based on gradient descent method
- However, being able to calculate gradient, we might select any of the above methods, which leads to orders of magnitude faster convergence

We discussed…



- How to construct an Artifitial Neural Network
  - Architecture, parameters, signal propagation, recall (inference)
- How to calculate the local gradient from the error function
  - Error back propagation
- Update strategies
  - Batch approach: Error function based on all the training vectors
    (*K: Number of all the training vectors*) $e = \frac{1}{K}\sum_{k=1}^{K}(d_k - Net(x_k, w))^2$
  - Instant update: Error function based on one training vector
    $e = (d_k - Net(x_k, w))^2$
  - Mini batch approach: Error function based on a random subset of the training vectors
    ($m_b \approx 200$) $e = \frac{1}{m_b}\sum_{k=1}^{m_b}(d_k - Net(x_k, w))^2$

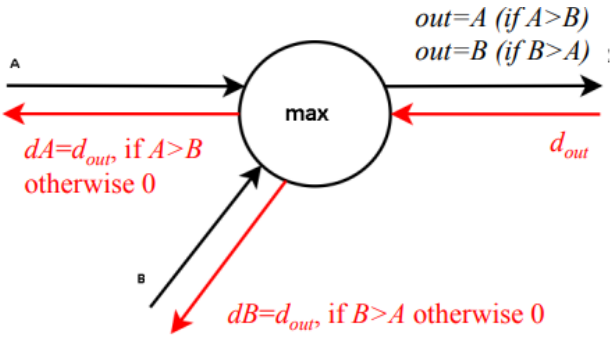
Epoch: *One Epoch is when the ENTIRE training set is passed forward and backward through the neural network only ONCE.*

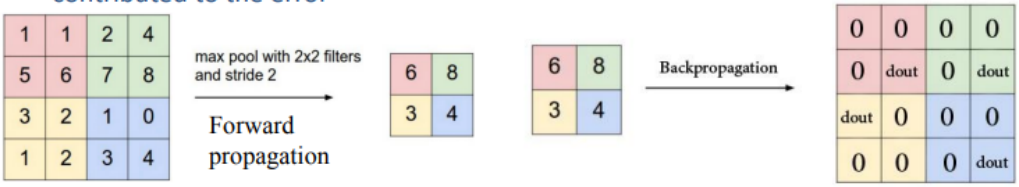
Epoch: time period (korszak in Hungarian)

Back propagation can be applied for batch normalized layers

Backpropagation through max-pooling layer

- Maximum node acts as a router
- The d_{out} gradient is given to the input node, which has contributed (which was the biggest)
- The remaining positions will get zero, because they did not contributed to the error





The maximum positions are stored

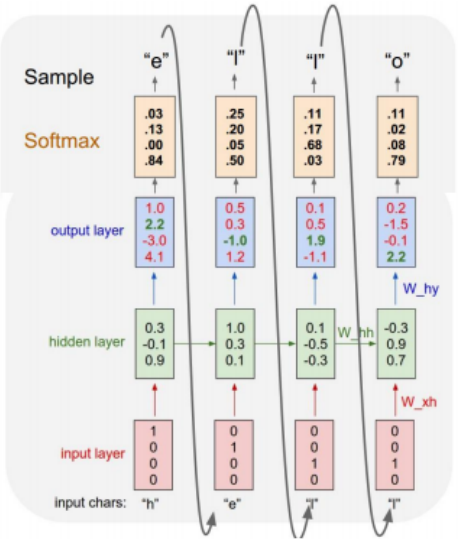
Simple RNN Training Example: Predicting the next letter

Example: Character-level Language Model Sampling

Vocabulary: [h,e,l,o]

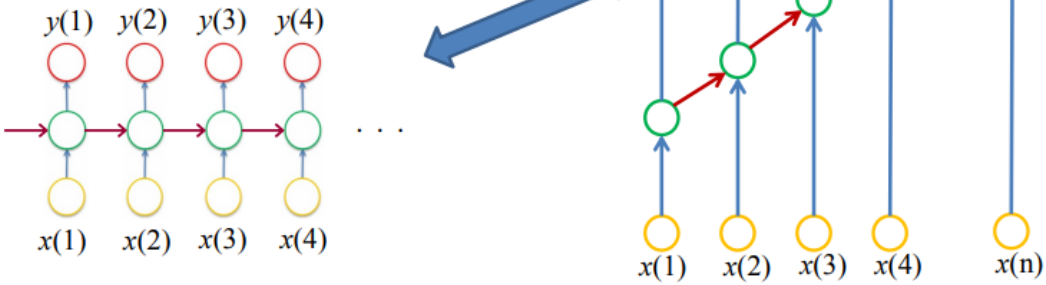
At test-time sample characters one at a time, feed back to model

Backpropagation can be started using negative log likelihood cost function



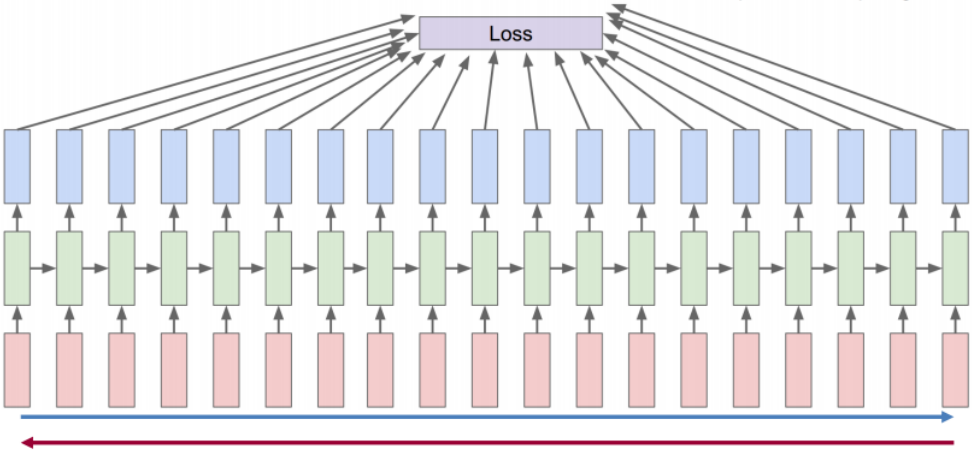
Back propagation through time

- Assuming that the length of the input vector sequence is limited
- It became a feedforward neural net
- Possible to apply back propagation
- We need multiple vector sequences to train!

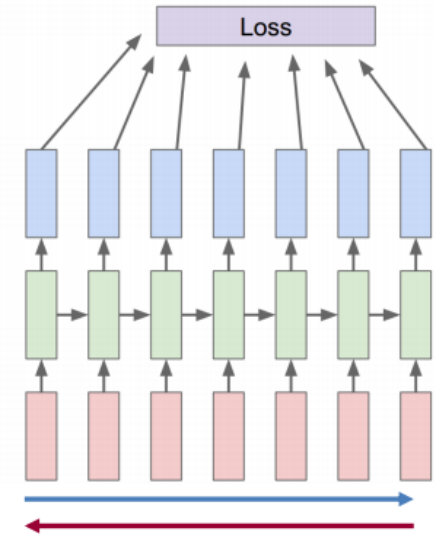


Backpropagation through time

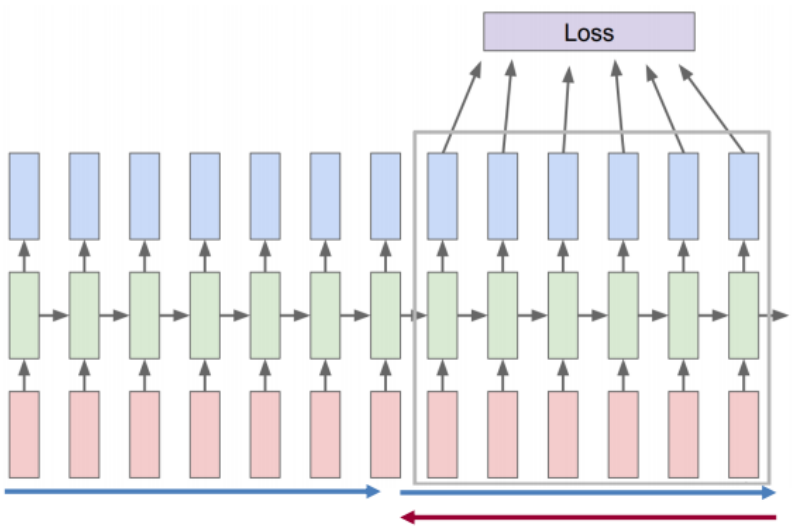
Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient



Truncated Backpropagation through time

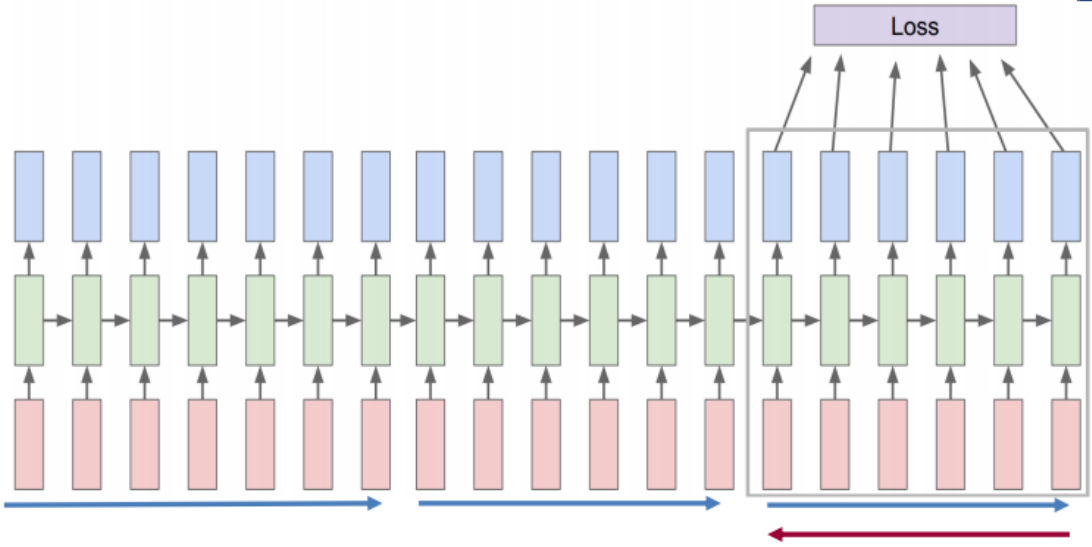


Run forward and backward through chunks of the sequence instead of whole sequence



Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps!

Truncated Backpropagation through time



-> en ebbol nem ertettem meg, de azért beraktam a diakat

c. Stochastic gradient descent optimizer

Stochastic process is a process, where we cannot observe the exact values. In these processes, our observations are always corrupted with some random noise.

- Algorithms with changing but not adaptive learning rate
  - Stochastic Gradient Descent algorithm
  - Momentum algorithm
  - Nesterov momentum update

Stochastic Gradient Descent (SGD) algorithm:

• Introduced in 1945

• Gradient Descent method, plus:

– Applying mini batches

– Changing the learning rate during the iteration

Learning rate at SGD

- Sufficient conditions to guarantee convergence of SGD:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

ϵ is the learning rate, also marked with η sometimes

- In practice:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \qquad \alpha = \frac{k}{\tau}$$

- After iteration τ, *it is common to leave* ε *constant*

Stochastic Gradient Descent algorithm

Algorithm Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.

Require: Initial parameter $\boldsymbol{\theta}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\hat{\boldsymbol{g}}$

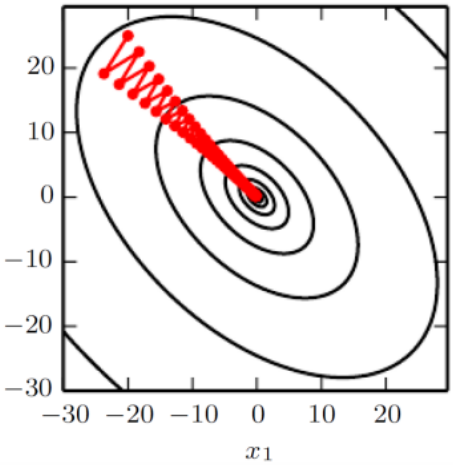
end while

where: L is the cost function

θ is the total set of $w_{i,j}^{(l)}$ (and all other parameters to optimize)

Stochastic Gradient Descent algorithm

- This very elongated quadratic function resembles a long canyon.
- Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature.
- Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration.



Training I

- Stochastic Gradient Descent (with momentum)
  - ADAM method was introduced in 2014 only (2 years later)
- Minimizing the negative log-likelihood (cross-entropy) loss function
- With L2 regularization (weight penalty):

$$L(w) = \sum_{i=1}^{N} \sum_{c=1}^{1000} -y_{ic} \log f_c(x_i) + \epsilon \|w\|_2^2$$

predicted probability of class c for image x

indicator that example i has label c

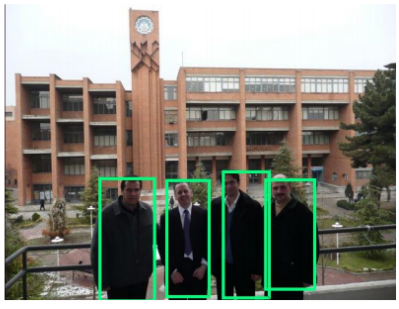
---

d. Object detection problem explained -> itt nem tudom mirol maradtam le, de en ezt se latom a diaban...

Object detection

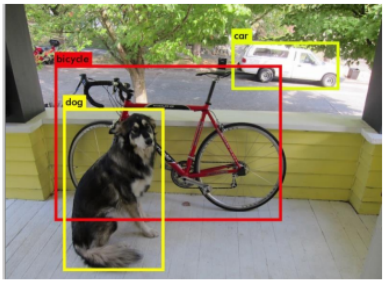
- One or Multiple object per image
  - Task:
    - Find the objects
    - Identify them with bounding boxes

Area or pixel level one-of-two decision!



Object detection/localization and classification

- Chicken and egg problem
  - You need to know that it is a bicycle before able to say that both a wheel part and a pipe segment belongs to the same object
  - You need to know that the red box contains an object before you can recognize it. (Cannot recognize a bicycle if you try it from separated parts)
- Our brain does it parallel
- How neural nets can solve it?
  - Detection by regression?
  - Detection by classification?

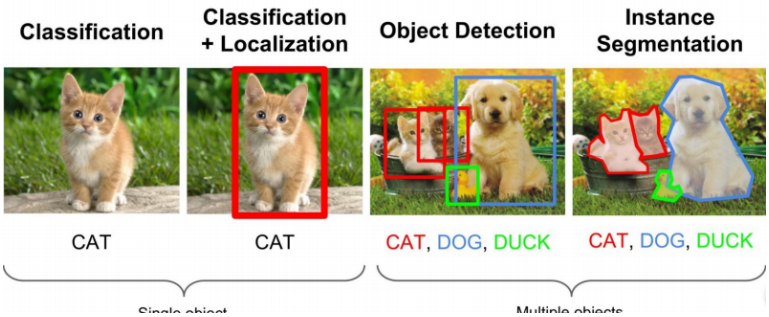


Neural networks for regression

Multiple object detection on a single image

Classification is good for a single object (can be extended for k objects – top k candidates)

How could we detect objects in general, when the number of objects is unknow



Dunno what I'm doin:

Object detection as regression

RCNN

Single Shot Object Detector (SSD) (2016 March)

You Only Look Once YOLO (2016 May)

R-CNN

Region proposal CNN network

Separate the problem of object detection and calssification

It consists of three modules.

The first generates category-independent region proposals. These proposals define the set of candidate detection avail-able to detector.

The second module is a large convolutional neural network that extracts a fixed-length feature vector from each region.

The third module is a set of class- specific linear SVMs

Ezek jók lehetnek:

- https://towardsdatascience.com/5-significant-object-detection-challenges-and-solutions-924cb09de9dd
- https://www.coursera.org/lecture/deep-learning-in-computer-vision/object-detection-problem-tvLPq

+ a szövege:

Object detection: probably the key problem in computer vision.
The goal of object detection is to detect the presence of object from a certain set of classes, and locate the exact position in the image.
We can informally divide all objects into two big groups: things and stuff. Things are objects of certain size and shape like cars, bicycles, people, animals, planes. We can specify where object is located in image with a bounding box. Stuff is more likely a region of image which correspond to objects like road, or grass, or sky, or water. It is easier to specify the location of a sky by marking the region in an image, not by a bounding box.

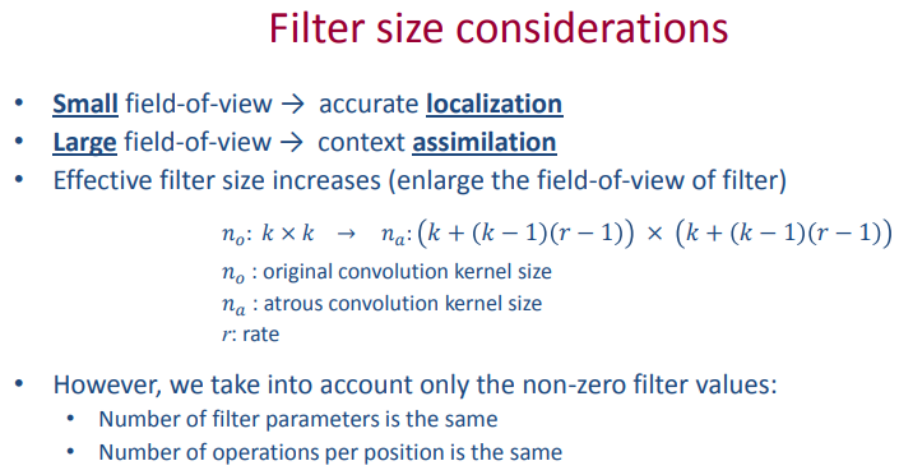
Now, we will talk mostly about **detection of things**. To detect a stuff, it is better to use semantic image segmentation methods, which we will discuss during week five. Compared to image classification, output of the detector is structured. Each object is usually marked with a bounding box and class label. Bounding box is described by position of one of the corners, and by width and size of the box. Object position and class are annotated in ground truth data. If only part of this information is annotated, we call it the "weak" annotation.
For example, only the presence of object in image can be annotated without a bounding box. There is a lot of research into object detection with weak annotation, but performance of such algorithms is lower compared to algorithms trained to use full annotation. To check whether the detection is correct, we compare the predicted bounding box with ground truth bounding box. The metric is intersection over union or IoU. It is the ratio of area of intersection of predicted in ground truth bounding boxes to the area of the union on these boxes as shown on the slide. Either IoU is larger than the threshold, then the detection is correct. The larger the threshold, the more precisely detector should localize objects. Currently, the threshold is usually set to 0.5. The detector output is a set of detection proposals. Usually, for each proposal the detector also gives a score as a measure of confidence in the detection. So, we can rank all proposals according to the score. Each proposal is considered. If IoU is larger than the threshold, then it is the true positive detection. If IoU is lower, then it's false positive detection. If some ground truth object is not detected, then it is marked as misdetection or false negative. On the whole dataset, you can measure the precision and the recall of the detector. The precision is the ratio between the number of true detections and the number of all detections. The recall is the ratio of number of true detection to the number of objects annotated in the ground truth data. By varying some parameter in the detector, usually the threshold on detection score, you can simultaneously change precision and recall. Then you can plot the precision-recall curve. To compare two detectors correctly, you should compare their precision-recall curves. If one curve is generally higher than the other curve, then the first detector is better than the other. To measure the overall quality of the detector with one number, we compute average precision. It is the mean of 11 points on the curve for recalls from zero to one, by 0.1

intervals. If you have multi-class detector, then you can compute mean average precision by averaging of average precision across classes. On the plot, we can select the working point, the precision, and the recall are best suited for the task at hand. Know that for production algorithm, the point is selected so that precision is closer one by sacrificing the recall. Sometimes, another charting metric are used to measure the quality of the detector. It is the plot of a miss rate with false detections per image? The curve is constructed similar to the precision-recall curve by varying the threshold on the detection score. For the object detection, the creation of ground truth annotation is very important. It has been demonstrated in the recent papers that annotations are usually different across datasets and annotators. **This lead to significant error in training of detector and its evaluation**. A lot of objects can be missed in ground truth data. This is especially true for the small objects or objects with very similar appearance to the target.

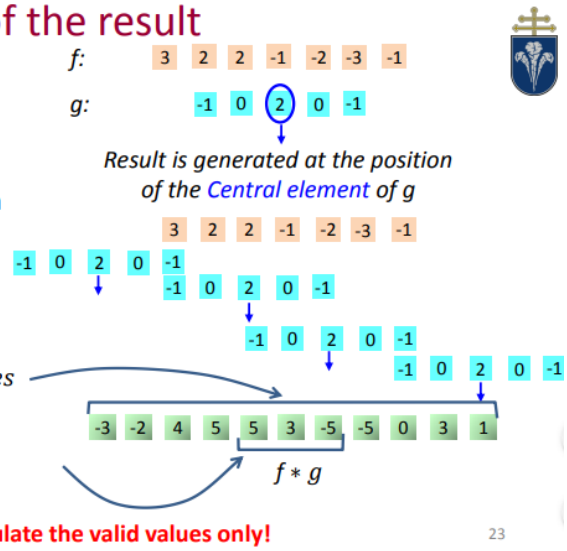
For example, if you want to detect faces, then detectors can detect not only real faces but for example prints on clothes or portraits. It is much better to annotate all such objects, and then mark some of them as objects to ignore during training and evaluation. It can also use several annotators for verification of annotation. The higher localization precision you want to obtain, the higher are usually the requirements on annotation precision. It is very important to have strict annotation protocol with clear definition how objects should be marked in difficult cases. For example, in a recent city person data set, each pedestrian should be marked with a line from toe to the head, to the middle point between the legs, and then bounding boxes width to height aspect ratio is placed on top of this annotation. Some objects are more important for us than other, so detection of such object classes as faces, pedestrian, or cars has a lot of practical applications. Thus, a lot of algorithms has been proposed for the detection of specific class of object. Such detectors reach top performance on their classes compared to the multi-class detectors. Practical multi-class detectors repeat only this development of deep learning methods. There are several peripheral data sets for multi-class detection. ImageNet is the first example of such data sets. It has objects of 1000 classes same as classification on ImageNet. Each image has annotation of one class and at least one bounding box. There are 800 training images per class. Detector should produce five guesses per image. Detection on the is correct if at least one guess has correct class and the corresponding bounding box is close to correct.

---

e. Effects of filter size on convolution



Size of the result



- In practice, convolution is used as a *filter*, where
  - f is the measrurement data, g is the filter function descriptor (*kernel*)
  - size(f) ≫ size(g)

size(f)= n size(g)= k n≥k

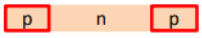
$$\text{size}(f * g) = \begin{cases} n + k - 1 & (if\ all\ values\ counted) \end{cases}$$

$f * g$

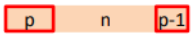
In CNN, we calculate the valid values only!

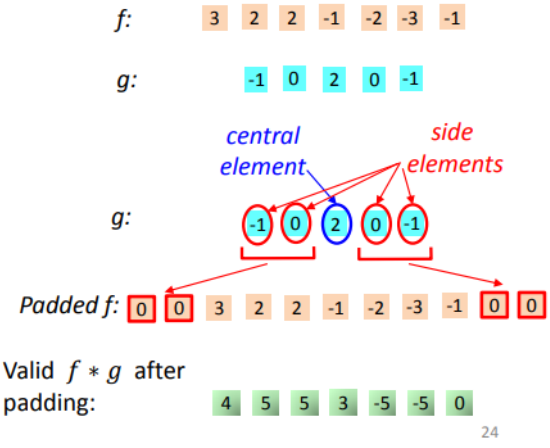
Padding in 1D

- In many case, we use a sequence of convolution filters on the measured data blocks
- We do not want size changes on the data blocks
- To avoid size changes, we have to pad the data block with zeros at the boundaries
  - *k=size(g)* is odd: k=2p+1



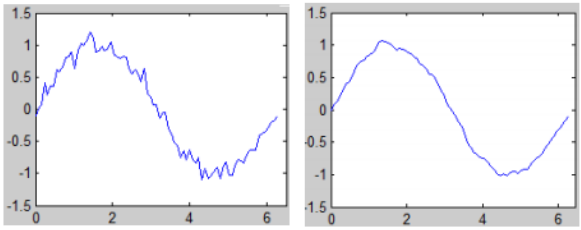
  - *k=size(g)* is even: k=2p
    - Padding is asymmetric:



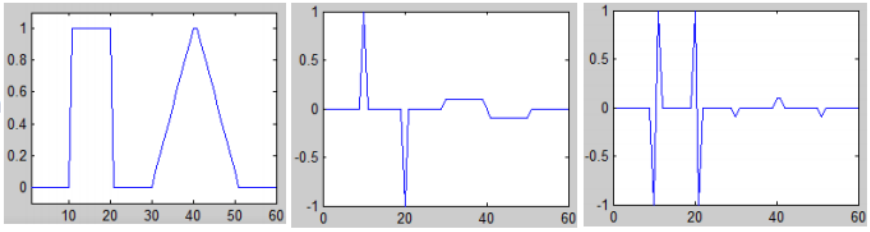


Usage of convolution I : 1D filtering

Smoothing noisy signal
Data lengths: 80 points
kernel: $\frac{1}{5}[\,1\ \ 1\ \ 1\ \ 1\ \ 1\,]$



Signal differentiation



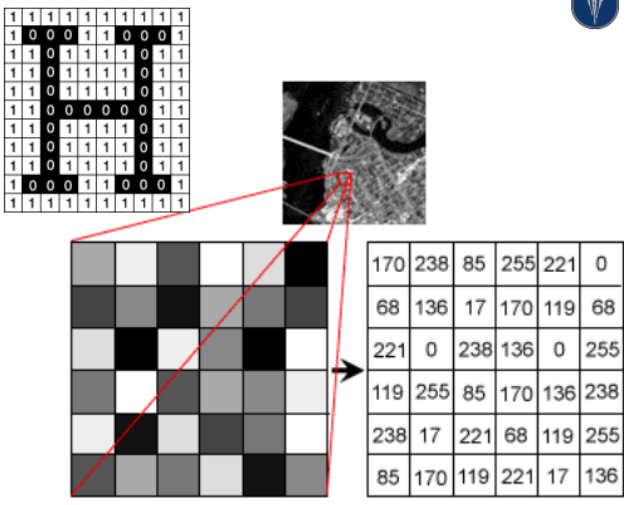
Data lengths: 60 points

$\frac{dy}{dx}$ kernel: $\frac{1}{2}[1\ \ -1]$

$\frac{d^2y}{dx^2}$ kernel: $\frac{1}{4}[-1\ \ 2\ \ -1]$

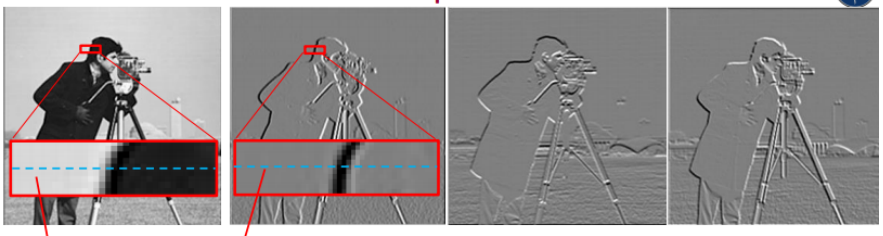
2D convolution: image filtering

- *What is a digital image?*
  - One-to-one mapping of a matrix and the pixels
  - Black-and-white image
    - Binary matrix
    - 0: black
    - 1: white
  - Monochrome (grayscale)
    - Matrix of (typically) 8 bit numbers
    - Values representing the brightness of the pixel
  - Color image
    - 3 matrices (R,G,B)



170	238	85	255	221	0
68	136	17	170	119	68
221	0	238	136	0	255
119	255	85	170	136	238
238	17	221	68	119	255
85	170	119	221	17	136

Usage of convolution II : 2D filtering
Sobel operation



Cameraman

First derivative (horizontal gradient)

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

First derivative (vertical gradient)

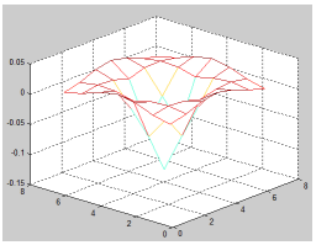
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & 1 \end{bmatrix}$$

First derivative (diagonal gradient)

$$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

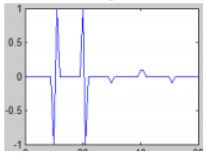
Usage of convolution III : 2D filtering

7x7 Laplacian of Gaussian kernel



$$\begin{bmatrix} 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.3 & 0.4 & -0.7 & -1.3 & -0.7 & 0.4 & 0.3 \\ 0.2 & 0.11 & -0.3 & -0.7 & -0.3 & 0.11 & 0.2 \\ 0.09 & 0.13 & 0.11 & 0.4 & 0.11 & 0.13 & 0.09 \\ 0.02 & 0.09 & 0.2 & 0.3 & 0.2 & 0.09 & 0.02 \end{bmatrix}$$

Second derivative of an image



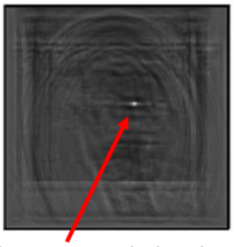
10/22/2019

Usage of convolution IV : 2D filtering

- Seeking for a known patter
- Large convolution kernel is applied
- Kernel size is equivalent with the size of the sought pattern

$*$

$=$



- Sensitive for rotation
- Scale variant

Filter responeded with a strong white peek in the matching position

oszinten nem tudom mi kene meg ide..

No. 15 Topic

a. Rosenblatt perceptron training algorithm

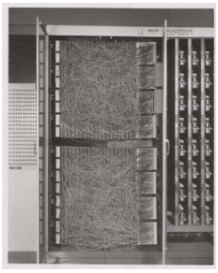
The learning algorithm: Recursive algorithm

- We have to develop a recursive algorithm called learning, which can learn the weight step by step, based on observing
  - the **(i) input,**
  - the **(ii) weight vector**,
  - the **(iii) desired output,** and
  - the **(iv) actual output** of the system.
- This can be described formally as follows:

$$\mathbf{w}(k+1) = \Psi\big(\mathbf{x}(k), \mathbf{w}(k), d(k), y(k)\big) \quad \rightarrow \quad \mathbf{w}_{opt}$$

The learning algorithm: Perceptron Learning Algorithm

- In a more ambitious way it can be called intelligent, because
  - perceptron can learn through examples (adapt),
  - even the function parameters are fully hidden.
- Perceptron learning was introduced by Frank Rosenblatt 1958
  - Built a 20x20 image sensor
  - With analog perceptron
  - 400 weights controlled by electromotors

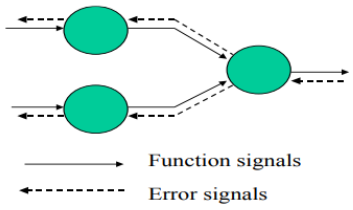


The learning algorithm: Recursive steps

1. *Initialization.*
   Set **w**(0)=**0** or **w**(0)=**rand**
2. *Activation.*
   Select a $\mathbf{x}_k \rightarrow d_k$ pair
3. *Computation of actual response*
   $y(k) = sign\big(w^T(k)x(k)\big)$
4. *Adaptation of the weight vector*
   $\mathbf{w}(k+1) = \Psi\big(\mathbf{x}(k), \mathbf{w}(k), d(k), y(k)\big)$
5. *Continuation*
   Until all responses of the perceptron are OK

Learning

- The Rosenblatt algorithm is inapplicable,
  - the error and desired output in the hidden layers of the FFNN **is unknown**
- Someway the error of the whole network has to be distributed to the internal neurons, in a feedback way



Forward propagation of function signals and back-propagation of errors signals

b. Back-propagation -> tok random lasd No. 14 Topic b.

Batch Normalization

- Done on layer level like softmax
- Training:
  - Done on minibatch level
- Inferencing:
  - Do the normalization with the pre-calculated parameters of the entire training set
- Batch normalization is differenciable via chain rule
  - Back propagation can be applied for batch normalized layers
- Rewriting the normalization using probability terms:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

E: *the expectation*
Var: the variance

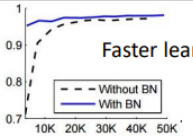
Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β (bias, weights)
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \epsilon\text{: avoid zero} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$



Faster learning

Back propagation

- We have seen last time how to calculate the gradient in a multilayer fully connected network using back propagation
  - The introduced method was based on gradient descent method
- However, being able to calculate gradient, we might select any of the above methods, which leads to orders of magnitude faster convergence

---

c. Curse of dimensionality

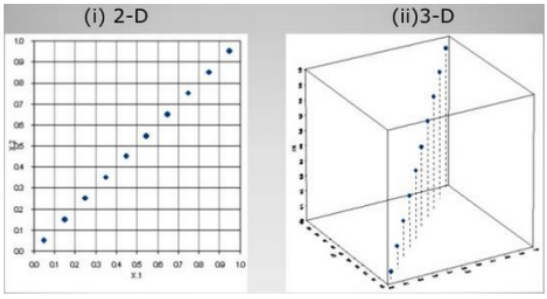
- What is it?
  - A name for **various problems that arise** when analyzing data in high dimensional space.
  - Dimensions = independent features in ML
    - Input vector size (different measurements, or number of pixels in an image)
  - Occurs when d (# dimensions) is large in relation to n (number of samples).
- Real life examples:
  - Genomics
    - We have ~20k genes, but disease sample sizes are often in the 100s or 1000s.

So what is this curse?

- **Sparse data:**
  - When the dimensionality d increases, the volume of the space increases so fast that the available data becomes **sparse, i.e. a few points in a large space**
  - Many features are not balanced, or are 'rarely occur' – sparse features
- **Noisy data**: More features can lead to increased noise → it is harder to find the true signal
- **Less clusters**: Neighborhoods with fixed k points are less concentrated as d increases.
- **Complex features**: High dimensional functions tend to have more complex features than low-dimensional functions, and hence harder to estimate

Data becomes sparse as dimensions increase

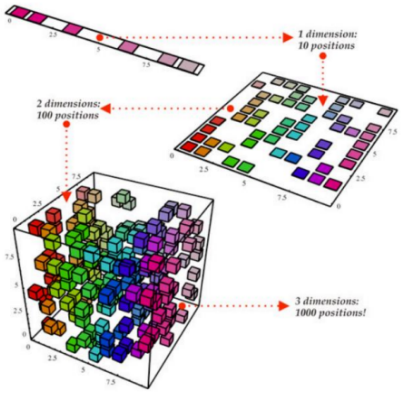
- A sample that maps 10% of the 1x1 squares in 2D represent only 1% of the 1x1x1 cubes in 3D



- There is an exponential increase in the search-space

Data sample number increase to avoid sparsity

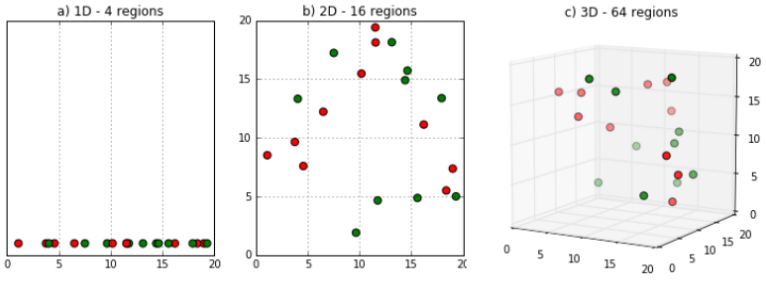
- e.g. 10 observations /dimension
  - 1D: 10 observations
  - 2D: 100 observations
  - 3D: 1000 observations
  - …



Curse of dim - Running complexity

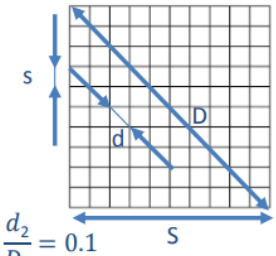
- Many data points (labeled measurements) are needed
- Complexity (running time) increase with dimension ***d***
- A lot of methods have at least $O(n*d^2)$ complexity, where ***n*** is the number of samples
- As ***d*** becomes large, this complexity becomes very costly.
  - Compute = $

Sparisty increase: More regions with the same number of data points



Distances in high dimension

- Assume, we have a unit side (2D) square, what we divided to 100 equal small squares
  - Calculate the ratio of the largest distance in a small square and the largest distance of the big square (in 2D)



$S_2=1$

$s_2=\sqrt[2]{\frac{1}{100}}=0.1$

$D_2=\sqrt{2}$

$d_2=0.1\sqrt{2}$

$$R_2=\frac{d_2}{D_2}=0.1$$

- Assume, we have a unit side 100D cube, what we divided to 100 equal small 100D cubes
  - Calculate the ratio Ratio of the largest distance in a small cube and the largest distance of the big cube (in 100D)
  - The average nearest neighbor distance is 95% of the largest distance!!!
  - Euclidian distance becomes meaningless, most two points are "far" from each others

$S_{100}=1$ $\quad s_{100}=\sqrt[100]{\frac{1}{100}}=0.95$

$D_{100}=\sqrt{100}=10$ $\quad d_{100}=\sqrt{100*0.95^2}=9.5$

$$R_{100}=\frac{d_{100}}{D_{100}}=0.95$$

Curse of dim - Some mathematical (weird) effects

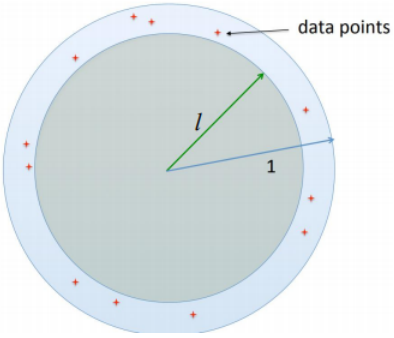
- Ratio between the volume of a sphere and a cube for **d**=3: $\frac{(\frac{4}{3})\pi r^3}{(2r)^3} \approx \frac{4r^3}{8r^3} \approx 0.5$
- When **d** tends to infinity the volume of the sphere (this ratio) tends to zero

d	3	5	10	20	30	50
ratio	0.52	0.16	0.0025	2.5E-08	2.0E-14	1.5E-28

- Most of the data is in the corner of the cube
  - Thus, Euclidian distance becomes meaningless, most two points are "far" from each others
- Very problematic for methods such as k-NN classification or k-means clustering because most of the neighbors are equidistant

The nearest neighbor problem in a sphere

- Assume randomly distributed points in a sphere with a unit diameter
- The median of the nearest neighbors is *l*
- As dimension tends to infinity
  - The median of the nearest neighbors converges to 1



"The Curse of Dimensionality" by Raúl Rojas
https://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/dimensionality.pdf

How to calculate dimensionality?

observations (*d*)

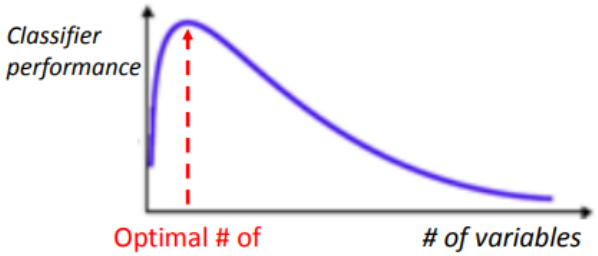
	feature vectors (*x*)			
	x_1	x_2	x_3	x_4
d_1	1	2	1	1
d_2	2	4	3.5	1
d_3	3	6	17	1

- How many dimensions does the data intrinsically have here? (How many independent coordinates?)

  - Two!
    - x1 = ½ * x2 (no additional information, correlated, not independent)
    - x4 is constant (carries no information at all!)

How to avoid the curse?

- Reduce dimensions
  - **Feature selection** - Choose only a subset of features
  - Use algorithms that transform the data into a lower dimensional space (example – PCA, t-SNE)
    **Both methods often result in information loss*
- Less is More
  - In many cases the information that is lost by discarding variables is made up for by a more accurate mapping/sampling in the lower-dimensional space



---

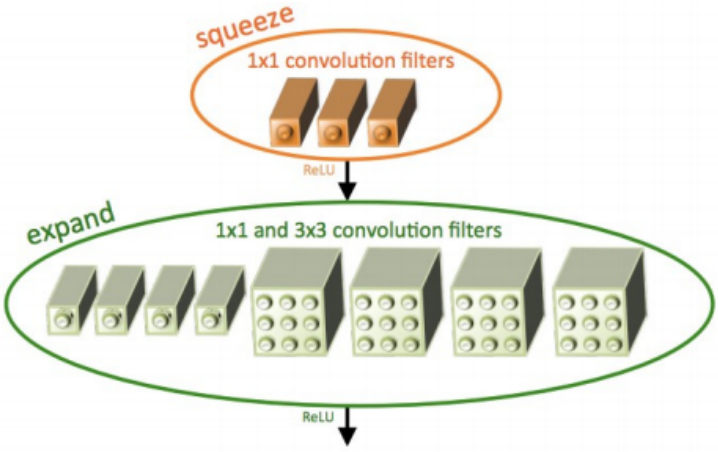
d. SqueezeNet

SqueezeNet

In this arhcitecture depths are squeezed before each operation

The expand is done by the concatenation of the 1x1 and the 3x3 convolutions.

Advantage: the expand layer is saved.



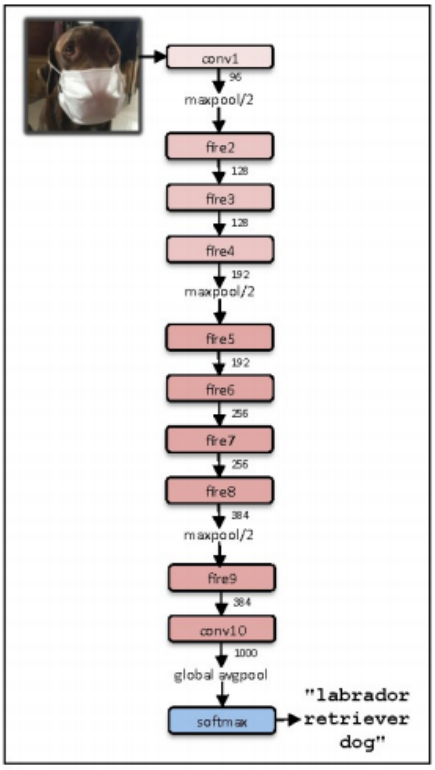
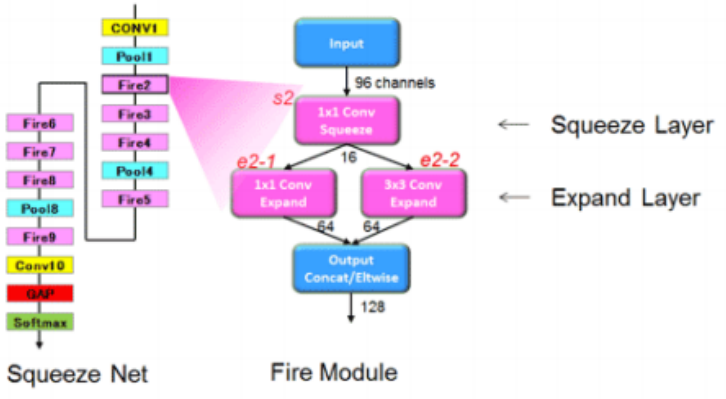


Figure 2. The SqueezeNet architecture

In this arhcitecture depths are squeezed before each operation



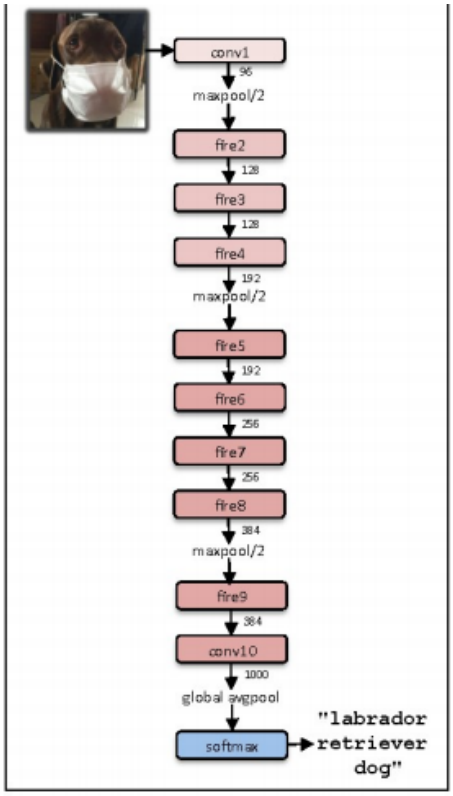


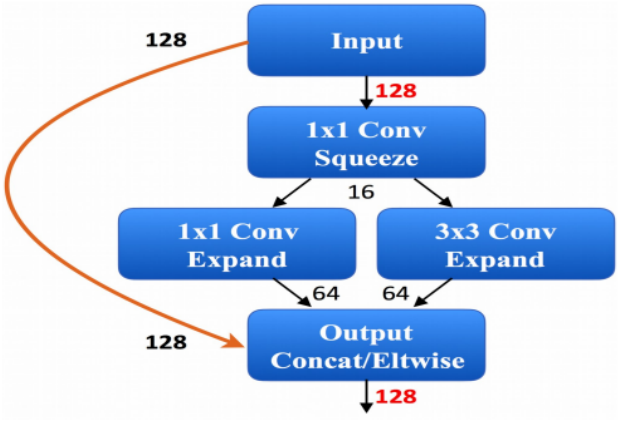
Figure 2. The SqueezeNet architecture

SqueezeNext

In this arhcitecture depths are squeezed before each operation

In a SqueezeNext architecture we will use a linear approximatine of 128 feature maps, using 16 independent feature maps

From the linear combination of these elements the new maps are created



e. Back-propagation and gradient-based optimizers

Simple gradient based optimizers:

- 1st and 2nd order optimizers

Quora - TL;DR:
The backpropagation algorithm is an instruction set for computing the gradient of a multi-variable function.
The Adam optimizer is a specialized gradient-descent algorithm that uses the computed gradient, its statistics, and its historical values to take small steps in its opposite direction inside the input parameter space as a means of minimizing a function. It is used for optimization in neural network training.
In other words, the Adam optimizer would need to use an algorithm like the backpropagation algorithm to first compute the gradient of the function. Then the Adam optimizer would use this computation to perform gradient-descent in a specialized manner.

Másik jó oldalka:
https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

Optimization Algorithm falls in 2 major categories -

1. First Order Optimization Algorithms — These algorithms minimize or maximize a Loss function E(x) using its Gradient values with respect to the parameters. Most widely used First order optimization algorithm is Gradient Descent.The First order derivative tells us whether the function is decreasing or increasing at a particular point. First order Derivative basically give us a line which is Tangential to a point on its Error Surface.

2. Second Order Optimization Algorithms — Second-order methods use the second order derivative which is also called Hessian to minimize or maximize the Loss function.The Hessian is a Matrix of Second Order Partial Derivatives. Since the second derivative is costly to compute, the second order is not used much .The second order derivative tells us whether the first derivative is increasing or decreasing which hints at the function's curvature.Second Order Derivative provide us with a quadratic surface which touches the curvature of the Error Surface.