Deep Learning - Zusammenfassung

Autoren: Linda Schneider und Julian Kotzur

Introduction

→ The importance of deep learning increased within the last years
→ Reasons are the huge possibilities arising out of higher computational power
→ Deep Learning is successfully used in Translating Software, Image Classification and Speech Assistance

Limitations of deep learning

- Image Captioning: Straight forward and fatal errors
- Deep learning often rely on manual made data sets
- End-to-end learning prohibits verification on parts
- Much time required to learn a deep learning network

Postulates for Pattern Recognition

1. Availability of a representative sample w of patterns $f^i(x)$ for the given field of problems Ω
2. A pattern has features, which characterize its membership to a certain class Ω_κ
3. Features of the same class should be compact in same domain: Domains of different classes should be separable
4. A complex pattern consists of simpler constituents, which have a certain relation to each other. A pattern can be decomposed into these constituents
5. A complex pattern has a certain structure. Important:not any arrangement of simple constituents is a valid pattern
6. Two patterns are similar if their features of simpler constituents differ only slightly

Rosenblatts Perceptron

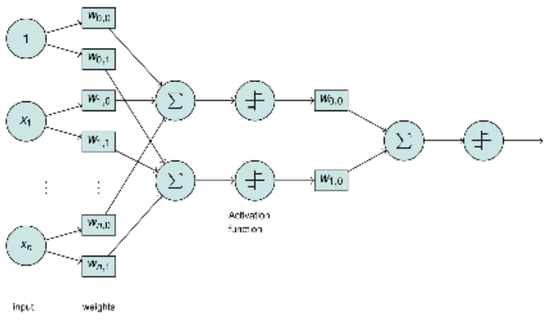
- Computes the function $y = sign(w^T x)$
- The set w are the weights with included bias at w_0
- The input vector x has also the bias with input 1 included
- The output has a binary classification $y \in \{-1, 1\}$
- Training: Process samples to perform weight update regularly and optimize them till convergence
- Task of the objective function: Find weights that minimize the distance of misclassified samples to the decision boundary

$$argmin\{D(w) = -\sum_{x_i \in M} y_i \cdot (w^T x_i)\}$$

  → M: Set of misclasified feature vectors
  ⇒ Iterative optimization
- Note: In each iteration, the cardinality and composition of M may change

Feedforward Networks

→ Multi-Layer Perceptrons enables non-linear decision boundaries
⇒ Example where it is needed: XOR Problem
→ Terms: Input, Hidden, Output Layer
→ A single layer can already be shown as an universal function approximator



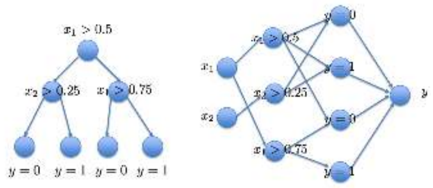
Universal Approximation Theorem

- Given is an arbitrary continuous function f on a compact subset of $\mathbb{R}^m$
- For any $\epsilon > 0$, you can find a feedfoward neural network with a non-linear activation function that can be trained to converge to f

$$|F(x) - f(x)| < \epsilon$$

- Is already fulfilled with a single hidden layer!
- BUT: It may be more efficient to use more hidden layer than just one to approximate a given function
- General problems: how many nodes, how to train ...

Example: Classification Tree into neural network



Softmax Function

- Used for multiclass problems
- For exclusive classes, y looks as follows:

$$y_k = \begin{cases} 1 & \text{if k is index of true class} \\ 0 & \text{otherwise} \end{cases}$$

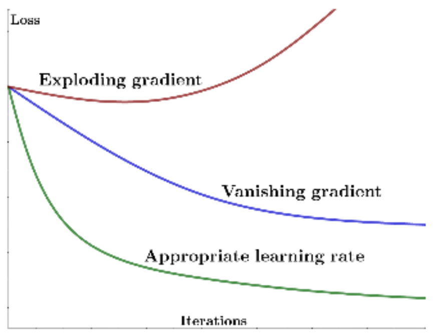
- *One-hot encoding*: Only one element is 1!
- Softmax function rescales a vector: $y_k = \frac{exp(x_k)}{\sum exp(x_j)}$
- Exponential function ensures positive values
- Allows to treat output as normalized probabilities
- Two properties: Output sums up to 1 and is always ≥ 0
- Loss function: How good is my prediction?
- Softmax loss: $L(y, x) = -log(\frac{exp(x_k)}{\sum exp(x_j)})|_{y_k=1}$
  → Combination with cross entropy loss
  ⇒ Naturally handles multiple class problems

Optimization of the network

- Goal: Find optimal weights for all layers
- Abstract whole network as loss function $L(w, x, y)$ and consider all training samples
- Gradient descent: $\min_w\{L(w, x, y)\}$
- Update of the weights:

$$w^{k+1} = w^k - \eta \nabla_w \frac{1}{M} \sum_{m=1}^{M} L(w, x, y)$$

- Choice of η is important:
  → Too high: Positive feedback which produced low grows without bounds
  → Too small: Negative feedback which produced a vanishing gradient



Vanishing and exploiding gradient

→ Problems get more worse for deeper networks

→ Reason for this are the multiplications

Example for vanishing gradient:

→ Sigmoid/tanh activation functions map large regions of x to a small range in y

⇒ Hence large changes in x only produce small changes in y

Calculating the derivative

- Finite Difference:
  → Easy to use
  → Computationally inefficient
  → Frequently used to check implementations
- Analytic gradient:
  → Chain rule and linearity enable to decompose functions
  → Computation more efficient
  → Analytic formulas have to be calculated manually

Backpropagation

- Forward pass: Compute activations
- Backward pass: Recursively apply chain rule
- Computationally very efficient using a dynamic programming approach
- Not a training algorithm, just a help for computing a gradient

Fully connected Layers

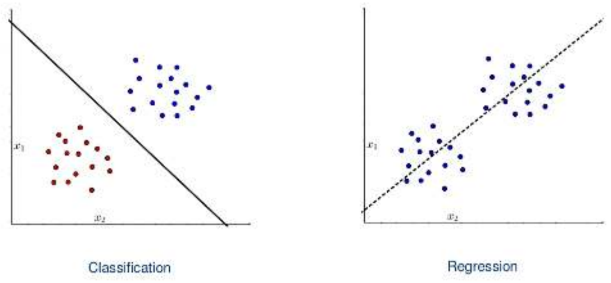
- Forward pass: $\hat{y} = Wx$
- Remember to add the bias as last entry of x and hence a further line for the weights
- Gradient with respect to weights: $\partial L/\partial W = \nabla_{\hat{y}} L x^T$
- Gradient with respect to input: $\partial L/\partial x = W^T \nabla_{\hat{y}} L$
- Example of an often used loss function (quadratic loss):

$$L(x, W, y) = 0.5\|Wx - y\|_2^2$$

⇒ Then simply: $\nabla_{\hat{y}} L = \hat{y} - y$

Loss functions

Distinguish between two different classes:



→ Classification: Estimate discrete variable for each input

→ Regression: Estimate continuous variable for each input

Last activation function vs. loss function

The last activation function:	The loss function
• is applied on each sample of the batch	• combines all samples and labels
• is used for training and testing	• is only used during training, not for testing
• produces output	• produces the loss
• is generally a vector	• is generally a scalar

Identification of suitable loss functions

→ We do not only want good results on the training set, but also for generalization

⇒ Hence, we want to develop a good estimator for our dataset

→ Instead of guessing a suitable estimator, we can use the Maximum Likelihood Estimation

Maximum Likelihood Estimator:

Given: Training set with observations X and labels Y

Assumptions:

- Probability for y_m given observation x_m is $p(y_m \mid x_m)$
- Input-Output-pairs are independent and identically distributed

⇒ Probability to observe Y is $\prod p(y_m \mid x_m)$

Likelihood function $L(w)$:

$$\max_w\{L(w)\} = \max_w\{\prod p(y_m \mid x_m, w)\} \Leftrightarrow \min_w\{-ln(L(w))\}$$

→ Using the negative log Likelihood is better because of the sum instead of the product!

How to find a suitable loss function

→ You should know the distribution of your dataset (nearly)

→ Then insert probability function inside negative log Likelihood and simplify it

⇒ For optimizing, you only need the parts depending on w!

Example: Regression with univariate Gaussian model

Probability function:

$$p(y \mid x, y, \beta) = \mathcal{N}(\hat{y}(x, w), \frac{1}{\beta}) = \sqrt{\frac{\beta}{2\pi}} e^{\beta \frac{-(y_m - \hat{y}(x_m, w))^2}{2}}$$

Calculation with log Likelihood:

$$\begin{aligned} L(w) &= \sum_{m=1}^{M} -ln(\sqrt{\frac{\beta}{2\pi}} e^{\beta \frac{-(y_m - \hat{y}(x_m, w))^2}{2}}) \\ &= \sum_{m=1}^{M} -ln(\sqrt{\frac{\beta}{2\pi}}) + \frac{\beta}{2}(y_m - \hat{y}(x_m, w))^2 \\ &= \frac{M}{2}(ln(2\pi) - ln(\beta)) + \underbrace{\frac{\beta}{2}\sum_{m=1}^{M}(y_m - \hat{y}(x_m, w))^2}_{depends\ on\ w} \end{aligned}$$

Filter relevant parts and derive L_2-loss:

$$\frac{1}{2}\sum_{m=1}^{M} \|y_m - \hat{y}(x_m, w)\|_2^2$$

→ L_2/L_1-loss can be applied for classification, but is generally used for regression

⇒ Minimizing the expected misclassification probability

→ BUT: Slow convergence, because there are no penalties for heavily wrong probabilities

⇒ Adventures choice for sets with extreme label noise

→ Classification of one class can be done with the Bernoulli distribution

$$\mathcal{B}(y \mid p) = \begin{cases} p^y(1-p)^{1-y} & if\ y \in \{0, 1\} \\ 0 & otherwise \end{cases}$$

→ One-hot encoded classification for multiple classes with the Multinoulli distribution

$$\mathcal{C}(y \mid p) = \begin{cases} \prod_{k=0}^{K} p_k^{y_k} & if\ y_k \in \{0, 1\} \\ 0 & otherwise \end{cases}$$

→ Inserted in the negative log Likelihood, we achieve:

$$-\sum_{m=1}^{M} \underbrace{\sum_{k=0}^{K} y_{k,m} ln(\hat{y}_{k,m})}_{Crossentropy} = -\sum_{m=1}^{M} ln(\hat{y}_k(x_m, w)) \mid_{y_{k,m}=1}$$

→ Cross-entropy can also be used for regression

→ We have to assure that the labels are between [0,1]

⇒ Easily done with the sigmoid activation function

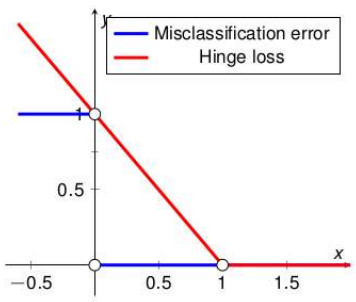
→ But: y is not longer one-hot encoded!

Subgradients

Motivation:

→ If we use the sign-function, we would just count the number of misclassifications

⇒ Hinge loss is a convex approximation of the misclassification loss

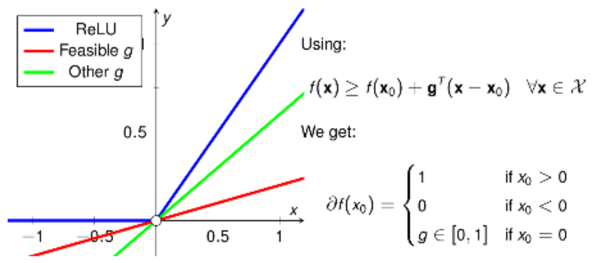


⇒ But what about the gradient?

Usage of subgradients

→ Subgradients are a generalization of gradients for convex, non-smooth functions

⇒ For piecewise continuous functions you just choose a particular subgradient



Why (not) use Support Vector Machines

- SVM compute the optimal separating hyperplane
- If it is not linearly separable, we have to take misclassification
- Punished with help of Lagrangian function, but only linearly
- It's a composition of L2-regularizer and hinge loss
- With the correct loss function, one can use SVM and other restricted optimization formulations
- Same as hinge loss up to a multiplicative component

Optimization

→ Optimizing empirical risk with gradient descent

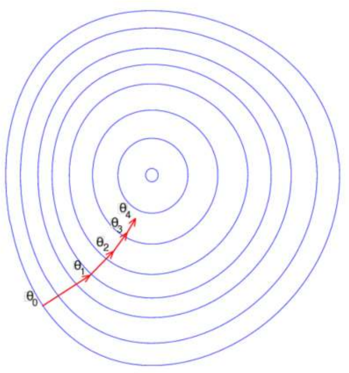
$$\mathcal{E}_{x,y\sim\hat{p}_{data}(x,y)}[L(w,x,y)] = \frac{1}{M}\sum L(w,x_m,y_m)$$

General gradient descent:

$$w^{(k+1)} = w^{(k)} - \eta\nabla L(w^{(k)},x,y)$$

→ Step size defined by learning rate η

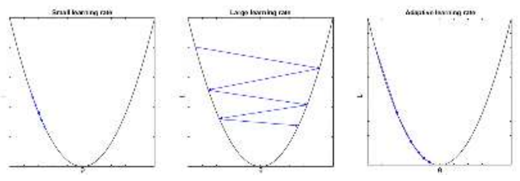
→ Calculate gradient with respect to every sample



Different versions of gradient descent

- Batch Gradient Descent: All M samples are used before updating the weights
  → Preferred option for convex problems
  → Guaranteed decrease of the error
  ⇒ Problems: Non-convexity and memory limitations
- Stochastic Gradient Descent (SGD): Use only one sample for updating the weights
  → Highly parallelizable
  → Not necessarily generate a decrease in every iteration
- Mini-Batch SGD: Use $B \ll M$ random samples
  → $\nabla L(w^{(k)}) = \frac{1}{B}\nabla\sum L(w^{(k)},x_b)$
  → Small batches offer regularization effect
  ⇒ Need a smaller η
  → Efficient method

Learning rate choice for SGD:



⇒ Practice: Adapt η gradually

Momentum for faster convergence

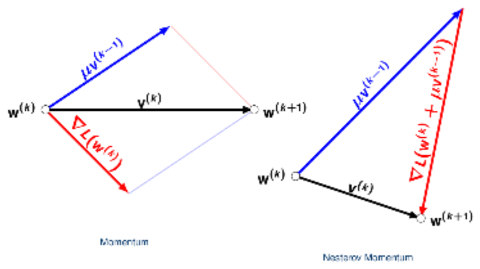
- Momentum: Parameter update based on current and past gradients, scaled with momentum μ

$$v^{(k)} = \mu v^{(k-1)} - \eta\nabla L(w^{(k)})$$
$$w^{(k+1)} = w^{(k)} + v^{(k)}$$

  + Dampened oscillations by overcomming poor Hessian and variance in SGD
  + Faster convergence
  - Adjusting of the learning rate still needed
- Nesterov Momentum: Compute the gradient in the direction we are going anyway

$$v^{(k)} = \mu v^{(k-1)} - \eta\nabla L(w^{(k)})$$
$$w^{(k+1)} = w^{(k)} - \mu v^{(k-1)} + (1+\mu)v^{(k)}$$

  + Good in situations of high variance



- Adaptive Gradient: Adaption based on all past squared gradients element-wise

$$g^{(k)} = \nabla L(w^{(k)})$$
$$r^{(k)} = r^{(k-1)} + g^{(k)} \odot g^{(k)}$$
$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{r^{(k)}}+\epsilon} \odot g^{(k)}$$

  + One can set individual learning rates for every parameter
  - Learning rates decrease too aggressively
- RMSProp: Improvement of AdaGrad by introducing an additional parameter ρ

$$g^{(k)} = \nabla L(w^{(k)})$$
$$r^{(k)} = \rho r^{(k-1)} + (1-\rho)g^{(k)} \odot g^{(k)}$$
$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{r^{(k)}}+\epsilon} \odot g^{(k)}$$

+ Aggressive decrease is fixed ($\rho = 0.9$)
- Still a learning rate to set

- Adadelta: Getting rid of the learning rate, but needs additional two formulas to RMSProp
- Adaptive Moment Estimation (Adam): Combination of all discussed methods, adds a bias correction

$$g^{(k)} = \nabla L(w^{(k)})$$
$$v^{(k)} = \mu v^{(k-1)} - (1-\mu) g^{(k)}$$
$$r^{(k)} = \rho r^{(k-1)} + (1-\rho) g^{(k)} \odot g^{(k)}$$
$$\hat{v}^{(k)} = \frac{v^{(k)}}{1-\mu^k} \quad \hat{r}^{(k)} = \frac{r^{(k)}}{1-\rho^k}$$
$$w^{(k+1)} = w^{(k)} - \eta \frac{\hat{v}^{(k)}}{\sqrt{\hat{r}^{(k)}} + \epsilon}$$

+ Robust optimization
+ Individual learning rate
- Loss curves are harder to interpret
- Empirically fail to converge to good solutions
$\rightarrow$ For ensure non-increasing step size:

$$\hat{v}^{(k)} = \max(\hat{v}^{(k-1)}, v^{(k)})$$

Activation functions

$\rightarrow$ Non-linear function enable the results of the universal approximation theorem
$\rightarrow$ Compared to the biological example, we can model all or nothing response with the sign function, but a time component is missing

Definition (Zero-centered).
A function is called zero-centered when it is symmetric in the origin.

$\rightarrow$ An activation function, that is not zero-centered causes the weights to have all the same sign, which may introduce undesirable zig-zagging in the gradient updates
$\rightarrow$ An activation of 0 produces an output bigger that 0 (co-variate shift of successive layers)
$\Rightarrow$ Layers constantly have to adapt the shifting distribution

---

Example: Assume we have two parameters w_1 and w_2 and we know $x_i > 0$. Compute the weight update:

$$f = \sum w_i x_i + b$$
$$\frac{\partial f}{\partial w_i} = x_i > 0$$
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial w_i} = \frac{\partial L}{\partial f} x_i$$

So the gradient has always the same sign as $\frac{\partial L}{\partial f}$, because it is the same for all weight updates.

$\rightarrow$ To solve this problem we can normalize the data in advance to be zero-centered (batch normalization)
$\Rightarrow$ Batch learning reduces the variance of the updates!

Different activation functions

- Sign:

$$f(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

+ Normalized output
+ Same like biological all or nothing response
- Gradient vanishes almost everywhere
- Mathematically undesirable

- Sigmoid Logistic function:

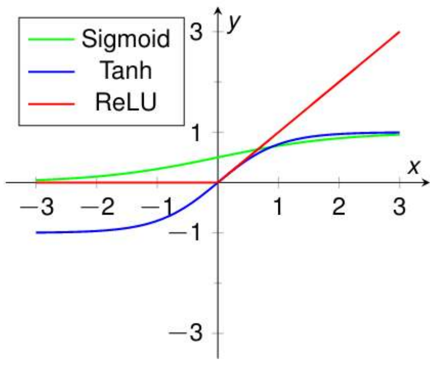
$$f(x) = \frac{1}{1 + exp(-x)}$$

+ Normalized/probabilistic output
+ Close to biological model, but differentiable
- Still the vanishing gradient problem for high/small values of x (saturates)
- Not zero-centered

- Tanh function:

$$f(x) = tanh(x)$$

+ Zero-centered
- Only a shifted version of sigmoid, hence still saturates



- Rectified Linear Unit (ReLU):

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

+ Good generalization due to piece-wise linearity
+ Speed up during learning
+ No vanishing gradient problem for positive values of x
- Not zero-centered
$\rightarrow$ Enabled training of deep supervised NN without unsupervised pre-training
$\rightarrow$ Dying ReLUs: If weights/bias trained negative values, that the ReLU performs only to 0 and than no more updates are possible.
$\Rightarrow$ Often related to a too high learning rate

- Leaky or parametric ReLU:

$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha x & else \end{cases}$$

$\rightarrow$ Leaky ReLU: $\alpha = 0.01$
$\rightarrow$ Parametric ReLU: learn α
+ Fixing the dying ReLU problem

- Exponential Linear Units:

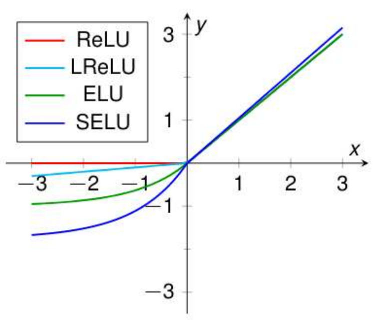
$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(exp(x) - 1) & else \end{cases}$$

+ No vanishing gradient
+ Reduces shift in activations
$\rightarrow$ Scaled version with additional lambda before, idea is self-normalization
$\rightarrow$ Theoretical properties great, but try ReLU first!

- Scaled ELU:

$$f(x) = \lambda \begin{cases} x & x > 0 \\ \alpha(exp(x) - 1) & else \end{cases}$$

$\rightarrow$ Idea: Self-normalizing
$\Rightarrow$ Alternative variant of ReLU



Finding optimal activation function

- Is a reinforcement learning problem
  $\Rightarrow$ Unfortunately that means training a network from scratch in every step
- Strategy:
  1. Define a search space
  2. Perform the searching using a recurrent neural network with reinforcement learning
  3. Use the best result
- Complicated activation functions did not perform well, even if they are the best functions
- The search often result in difficult, expensive optimization problems

Characterization of good activation functions

- They have almost linear areas to prevent vanishing gradients
- They have saturating areas to provide non-linearity
- They should be monotonic

Convolutional Neural Networks

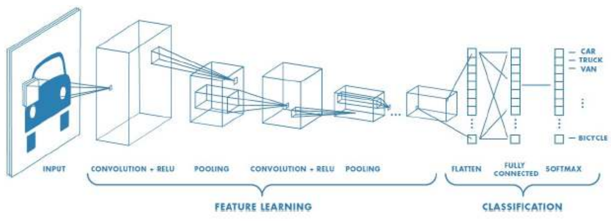
Motivation

- We can represent any kind of relationship between inputs with fully connected layers
- But images/videos/sounds have a very high number of input data
  $\rightarrow$ Example: Image with 512×512 pixels and 8 hidden neurons

$$(512^2 + 1) \cdot 8 > 2 \text{ million trainable weights}$$

- From a machine learning point of view, pixels are bad features, because they are
  $\rightarrow$ highly correlated
  $\rightarrow$ scale dependent
  $\rightarrow$ have intensity variations
- We can try to find the same macro features at different locations and build a hierarchy
  $\Rightarrow$ Composition matters!

Architecture of a convolutional neural network



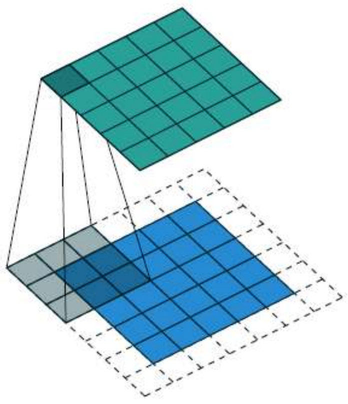
$\rightarrow$ Local connectivity can be extracted with filters, which are the same for the whole image
$\Rightarrow$ Secures translational equivariance
$\rightarrow$ Filters/kernels have to be learned!

Four essential building blocks

- Convolutional layer for feature extraction
- Activation function for introducing non-linearity
- Pooling layer for compressing and aggregating information
- Last Layer for classification (fully connected or flatten and 1×1 convolution)

Convolutional Layers

$\rightarrow$ Exploit structure by only connecting pixels in a neighborhood
$\Rightarrow$ Can be expressed as a fully connected layer, where nearly each entry in W is 0
$\rightarrow$ Effective filter size: $3 \times 3,\ 5 \times 5,\ 7 \times 7$
$\rightarrow$ Features that are important at one location are likely important anywhere in the image
$\Rightarrow$ Hence use shared weights and trainable features
$\rightarrow$ Remember: Cross-correlation is convolution with a flipped kernel – and vice versa
$\rightarrow$ Implementation: Cross-correlation is mostly used in the forward pass, because the weights are initialized randomly anyway



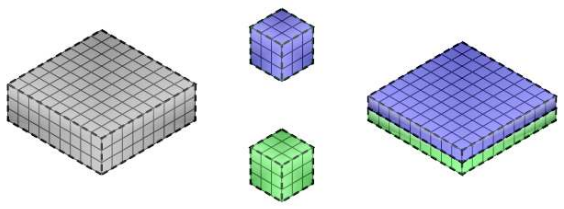
Padding

- Generally: Convolution without striding can reduce the image size by $2 \cdot \lfloor \frac{n}{2} \rfloor$
- 'Same'/Zero padding: Fill boarder with zeros, such that input and output have the same size
- 'Valid'/No padding: Output is smaller than the input

Forward Pass: Multi-channel convolution

- Input of size $X \times Y \times S$, where $X \times Y$ is the pixel dimension of the picture and S is the number of channels (eg. colors)

- H filters with size $M \times N \times S$
  $\Rightarrow$ Fully connected across the channels!
- Output dimension with zero padding: $X \times Y \times H$



Backward Pass: Multi-channel convolution

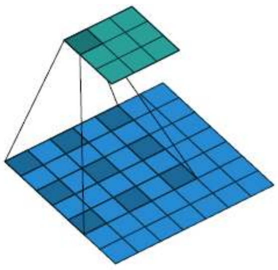
- We can use the same formulas as in a fully connected layer, but this needs a lot of rearranging to create correct weights and error matrices!
- Hence for implementation, we also do a convolution with flipped filters
- Instead of flipping the filters, we use cross-correlation in the forward and convolution in the backward pass
- For 3D operations, the channel dimension has to be flipped once more!

Strided Convolutions

- Instead of multiplying the filter at each pixel position, we can skip some positions
- Stride s describes the offset, the output size is reduced by a factor of s
- Mathematically: Convolution + subsampling
- Allows trainable downsampling strategy

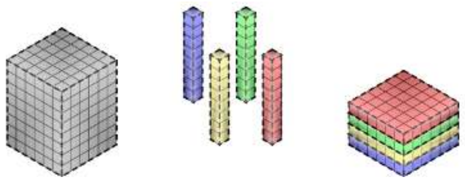
Dilated Convolutions

- Dilate convolution kernel: Skip certain pixels
- Goal: Wider receptive field with less parameters/weights



1×1 Convolution Concept

- Till now, filters are fully connected in depth direction S
- If we decrease the neighborhood to 1×1, we gain the fully connected property everywhere
- The dimensionality change from S channels to H features remains



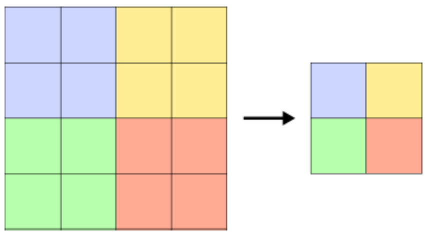
$\Rightarrow$ If we flatten the input, 1×1 convolutions are a fully connected layer

- 1×1 convolutions simply calculate inner products at each position
- 1×1 convolutions are a simple an efficient method to decrease depth size of a network
  $\Rightarrow$ Can accelerate the computation
- Additionally dimension reduction is learned
  $\Rightarrow$ Useful for reducing redundancy in feature maps

Pooling Layers

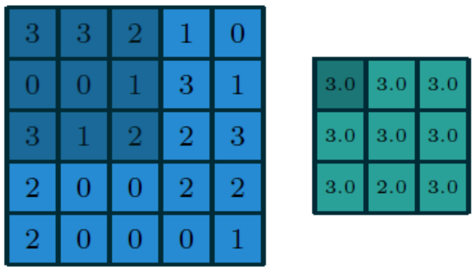
Idea behind Pooling Layers

- Fuses information of the input across spatial locations
  $\Rightarrow$ Decreases the number of parameters!
  $\Rightarrow$ Reduces computational costs and overfitting
- Assumptions:
  - Features are hierarchically structured
  - Regions can summarized
  - It's translational invariant
  - Exact location of a feature is not important



Max Pooling

- Forward: Propagate maximum value in a neighborhood to next layer
- Stride of pooling usually equals the neighborhood size (eg. 2×2)
- Don't forget: Maximum propagation adds additional non-linearity



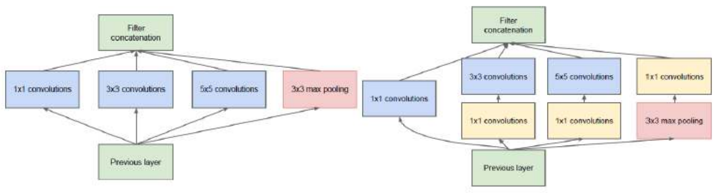
- Backward: Hence only one value contributes the error, only this weight has to be affected
- Error is propagated only along the path of the maximum value
- A subgradient is given by the colloquial rule 'Winner takes it all'
- In cases where the stride is smaller than the kernel size the error might be routed multiple times to the same location and therefore has to be summed up

Average Pooling

- Forward: Propagate average of the neighborhood
- Backward:Error is shared to equal parts
- Does not consistently perform better than max pooling

Inception model

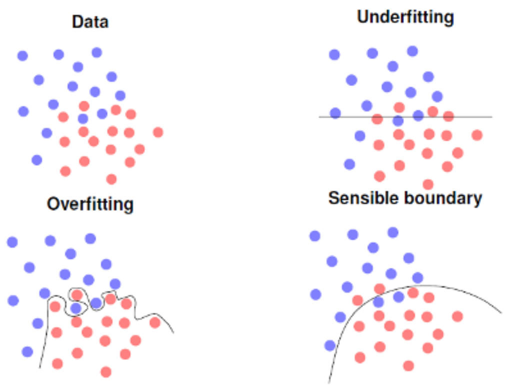
$\rightarrow$ Idea: Let the model learn which type of filter layer suits best
$\Rightarrow$ Construction of 'inception modules' that are stacked to form a large network



Regularization

The aim of regularization is the minimization of the generalization error, not of the training error. So we call every change of the learning algorithms regularization, which fulfills this goal.

Motivation: Finding a good boundary



$\Rightarrow$ Model should also be good in generalization, not just in training
$\Rightarrow$ Regularization helps to reduce overfitting!

Bias Variance Decomposition

- Is a decomposition for a regression problem
- A similar decomposition exists for classification using the zero-one loss
- Assume we have

$$h = f(x) + \epsilon \text{true model + noise}$$

$$y = f_i(x) \text{different predictors}$$

$$\bar{f}(x) = \frac{1}{m}\sum f_i(x) \text{average predictor}$$

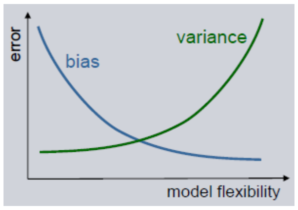
- Under the assumption that all upcoming cross-correlations are zero, the average prediction error can be decomposed in:

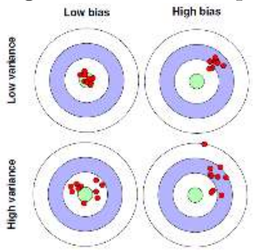
$$\begin{aligned}
&\frac{1}{m}\sum\frac{1}{T}[h_t - f_i(x_t)]^2 && \text{aver. prediction error}\\
=&\frac{1}{T}\sum[h_t - f(x_t)]^2 && \text{noise}\\
&+\frac{1}{T}\sum[f(x_t)-\bar{f}(x_t)]^2 && \text{bias}\\
&+\frac{1}{m}\sum\frac{1}{T}\sum[\bar{f}(x_t)-f_i(x_t)]^2 && \text{variance}
\end{aligned}$$

- We would like to minimize bias and variance

$\rightarrow$ Simultaneously optimizing bias and variance is impossible in general

$\Rightarrow$ Bias and variance can be studied together as model capacity





Model capacity

- Capacity of a model equals the variety of functions it can approximate
  - $\rightarrow$ Can be increased by increasing the number of parameters and epochs
- One can measure capacity with Vapnik-Chervonenkis (VC) dimension

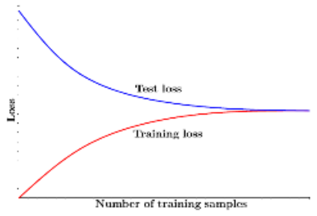
$\rightarrow$ It is based on counting how many points can be separated by a model

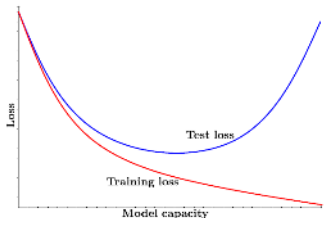
$\rightarrow$ Because of the high dimension of neural networks compared to classical methods, VC dimension is ineffective in judging the real capacity of neural networks

- Note: We can always reduce the bias by increasing model capacity, but the has the price of increasing variance

Influence of data size

- Observation: The variance of the model can be optimized by using more training data
  - $\rightarrow$ Model capacity has to match the size of the training set
- If no more data can be acquired, we should think about trading a higher bias for a lower variance



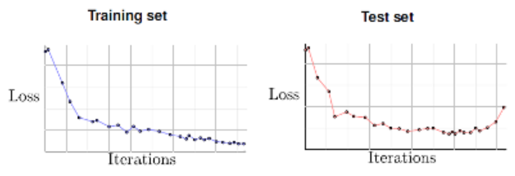


Classical Techniques

Usage of a validation set

$\rightarrow$ We are not allowed to use the test set for training

$\rightarrow$ To recognize overfitting, we split of a validation set from the training data!



$\rightarrow$ Here we can use early stopping to use parameters with minimum validation loss

$\rightarrow$ Example stopping criterion: Growing loss for the last x iterations

Data Augmentation for enlarging the dataset

$\rightarrow$ Every transformation should be invariant to the label

Common transformations:

- Random spatial transformations: affine, elastic, rotations ...
- Pixel transformations: changing resolution, random noise, changing pixel distribution ...

Regularization in the loss function

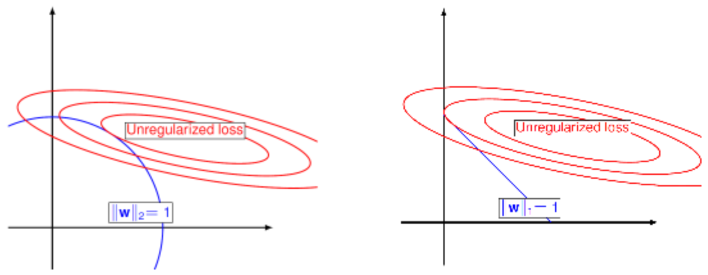
- General idea: restrict model capacity by using a parameter-penalty-term in the objective function ($\rightarrow$ loss function)
  - $\rightarrow$ This is weighted with a hyperparameter λ. Bigger values of λ lead to a stronger regularization
- L_2-Regularization: Add a penalty term for the weights in L_2-Norm (weight decay)

$$\tilde{L}(w,X,Y) = L(w,X,Y) + \lambda\|w\|_2^2$$

  - $\rightarrow$ Moves the weights closer to the origin, hence enforces a small norm
  - $\Rightarrow$ We have also to change the weight-parameter update:
  $$w^{(k+1)} = \underbrace{(1-\eta\lambda)w^{(k)}}_{Shrinkage} - \eta\frac{\partial L}{\partial w^{(k)}}$$
  - $\rightarrow$ Weights are constantly reduced by a certain factor
  - $\Rightarrow$ Directions in which the parameters contribute strongly to the reduction of the objective function remain relatively intact
  - $\rightarrow$ We trade increased bias for reduced variance
- L_1-Regularization: Add a penalty term for the weights in L_1-Norm

$$\tilde{L}(w,X,Y) = L(w,X,Y) + \lambda\|w\|_1$$

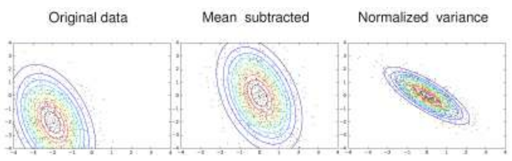
  - $\rightarrow$ Forces parts of the weights to be zero, hence enforces sparsity
  - $\rightarrow$ Can be used for feature selection
  - $\Rightarrow$ We have also to change the weight-parameter update:
  $$w^{(k+1)} = \underbrace{(w^{(k)}-\eta\lambda)sign(w^{(k)})}_{Shrinkage} - \eta\frac{\partial L}{\partial w^{(k)}}$$



Normalization

Data Normalization

- Only use training data to calculate normalization
- Normalization of the input data and within the network possible



Batch Normalization

- Add an extra layer to normalize the input for the following layer
- Calculate mean μ_B and standard variation σ_B from the mini-batch and produce normalized output

$$\tilde{x}_i = \frac{x_i - \mu_{B,i}}{\sqrt{\sigma_{B,i}^2 + \epsilon}}$$

- Use two trainable weights γ, β for scaling the normalized input
- For testing: replace μ_B, σ_B with μ, σ of the training set
- Convolutional layers are special, because batch normalization computes scalar μ, σ for every channel instead of vectors
- Nobody really knows why this is helpful, but practical evidence is overwhelming
- There exist many variants like calculating μ, σ over activations, layer, groups etc.

$\Rightarrow$ Original motivation: BN reduces the Internal Covariate Shift

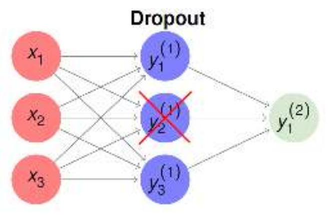
Internal Covariate Shift

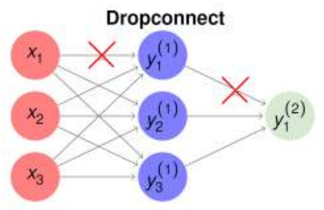
- ReLU is not zero-centered
- Initialization and input distribution might not be normalized

  $\rightarrow$ Input distribution shifts over time

- Effect amplifies for deeper networks, which leads to a slow learning rate

Dropout and Dropconnect



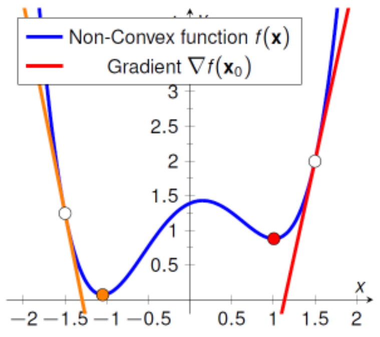


- Randomly set activations to zero with prob. 1 - p
- Test-time: multiply activation with p

- Generalizes Dropout with setting connections to zero
- Less efficient implementation

Initialization

- Initialization matters for non-convex functions, which we can generally assume in context of neural networks



Basic initialization

- Bias
  - Generally, bias units can simply be initialized to zero
  - When using ReLU, a small positive constant is better because of the dying ReLU issue
- Weights
  - Need to be random to break symmetry
  - It is not a good idea to initialize them with zeros because than the resulting gradient is zero!
  - Similar to the learning rate their variance influences the stability of learning

  $\Rightarrow$ Small uniform/Gaussian values work

Xavier initialization

- Assumption: We have linear neurons
- Then we can calibrate the variances for the forward pass by initializing with a zero mean Gaussian $\mathcal{N}(0, \frac{1}{fan\_in})$ where fan_in is the input dimension of the weights
- For the backward-pass we need $\mathcal{N}(0, \frac{1}{fan\_out})$ where fan_out is the output dimension of the weights

$\Rightarrow$ Hence we average those two to $\mathcal{N}(0, \sqrt{\frac{2}{fan\_in + fan\_out}})$

He initialization

- Assumption of linear neurons is a problem

$\Rightarrow$ For ReLU it's better to use $\mathcal{N}(0, \sqrt{\frac{2}{fan\_in}})$

Transfer Learning

- Some datasets are very small
- Reuse models trained
  - for a different task on the same data
  - on different data for the same task
  - on different data for a different task
- For weight transfer in convolutional layers (extract features) we expect the less task-specific weights to be in earlier leayers

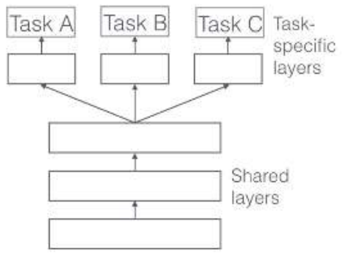
  $\rightarrow$ We cut the network at some depth of the feature learning part

  $\Rightarrow$ Those parts can be fixed with $\eta = 0$ or fine-tuned
- Also found to be beneficial if we transfer from RGB images to X-ray (sufficient to simply copy the input three times)

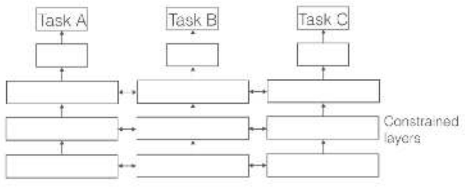
$\Rightarrow$ Always use transfer learning!

Multi-Task Learning (MTL)

- So far we had one network for one task and discussed hoow to reuse them
- Learning them simultaneously can provide a better understanding of the shared underlying concepts
- Idea: Train a network simultaneously on multiple related tasks
- Adapt loss function to assess performance for multiple tasks
- Reduces risk of overfitting on one particular task
- Hard parameter sharing



  - Several hidden layers shared between all tasks
  - Additional task-specific output layers
  - Multi-task learning of N tasks reduces chance of overfitting by an order of N
- Soft parameter sharing



  - Each model has its own parameters
  - Instead of forcing equality, distance between parameters is regularized as part of the loss function

Auxiliary tasks

- Additional tasks have own purpose or are just auxiliary to the original task
- Example: Facial Landmark Detection

$\rightarrow$ Simultaneously learn to estimate landmarks and "subtly" related task like face pose, glasses, smiling, gender

- Certain features difficult to learn for one task but easy for a related one
- Auxiliary tasks can help to "steer" training in a specific direction
- Include prior knowledge by choosing appropriate auxiliary tasks
- Tasks can have different convergence rates (task-based early stopping possible)

Common Practice

1. Check your implementation before training: Gradient, initialization, ...
2. Monitor training process continuously: training/validation loss, weights, activations
3. Stick to established architectures before reinventing the wheel
4. Experiment with few data sets, keep your evaluation data safe until evaluation
5. Decay the learning rate over time
6. Do random search (not grid search) for hyperparameters
7. Perform model ensembling for better performance
8. Check for significance when comparing classifiers

Class Imbalance

- Different classes are available with very different frequencies in the dataset

→ Hard challenge for machine learning algorithms

- Sometimes, learning to always assign to the same class can lead to an extreme high accuracy
- Problem: Measures like accuracy, L2 norm, cross-entropy do not show imbalance
- Different resampling strategies:
  - Undersampling: Just use a subset of the overrepresented class such that all classes are now presented equally often

  → Disadvantage: Not all data is used for training, which may lead to underfitting
  - Oversampling:Use sample from underrepresented class multiple times such that you can use all data

  → Disadvantage: Can lead to overfitting
- Underfitting caused undersampling can be reduced by taking a different subset after each epoch
- Data augmentation can help to reduce overfitting for underrepresented class

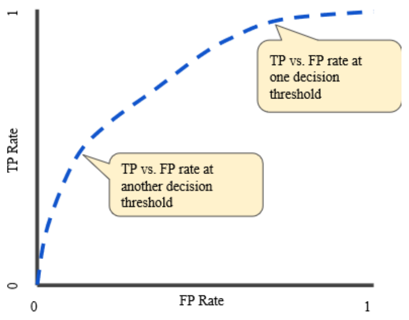
Evaluation

- Remember: Labels are set by human, hence it does not have to be correct
- Different ways of measuring performance:
  - Accuracy: $ACC = \frac{\#CorrectPredictions}{TotalPrediction}$

  → Fraction of predictions our model got right
  - Precision: $PRE = \frac{\#CorrectPositives}{TotalPositives}$

  → What proportion of positive identifications was actually correct?
  - Recall: $REC = \frac{\#CorrectPositives}{TotalrealPositives}$

  → What proportion of actual positives was identified correctly?
  - ROC Curve: Graph showing the performance of a classification model at all classification thresholds



- Multiclass classification: Top-K error, True class label is not in the K classes with the highest prediction score
- k-fold cross validation: split data in k folds, train and test, repeat k times
- Measuring Significant Difference: Run training for each method/network multiple times and determine whether performance is significantly different e.g. Student's t-test

Architectures

Different interesting Architectures

- LeNet-5 (1998): First convolutional deep network
- AlexNet (2012): Winner of the ImageNet 2012 challenge, used GPU to reduce training time, Introduced ReLU
- Visual Geometry Group: Used small kernel sizes, but is hard to train
- GoogleNet: 22 layers, use inception modules to decide between convolution and pooling
- ResNeXt: Aggregated residual transformations
- DenseNets: Layer input: feature-maps of all preceding layers, hence one can reuse features
- SENet: Explicitly model channel interdependencies : channels have different relevance depending on content

Bottleneck layer A bottleneck layer is a layer that contains few nodes compared to the previous layers. It can be used to obtain a representation of the input with reduced dimensionality.

- Features are often correlated

→ Redundancy can be removed by 1×1 filters

- Reduced the number of feature maps that have to be convolved

⇒ Less multiply and add operations

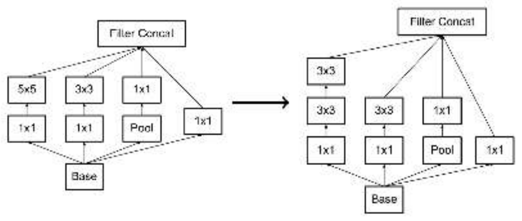
Deeper Models

→ Observation: Top 5 Error can be decreased with the help of deeper networks

→ Advantage of going deeper: Increasingly abstract features and exponential feature reuse

The evolution of Inception

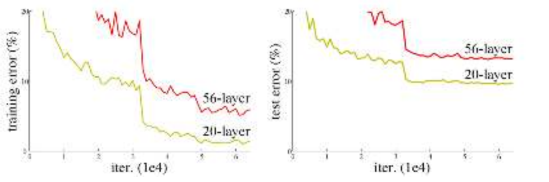
- Change basic inception layer: Replace 7×7 and 5×5 filters by multiple 3×3 convolutions



- Efficient grid size reduction with introducing additional striding
- Improvements: RMSProp ,Bach-normalization also in the FC layers of auxiliary classifiers and Label-smoothing regularization (exchange label distribution)

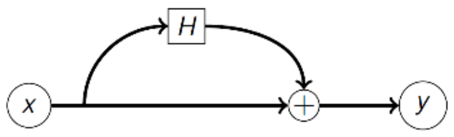
So why not going deeper

- Disadvantage: Deeper models tend to have higher training and test error than shallower models, which cannot only be caused by overfitting
- Possible reasons:
  - Vanishing gradient problem which can tried to be solved with ReLU or a proper initialization
  - Internal co-variate shift which can tried to be solved with batch normalization or the use of ELU/SELU
  - Degradation problem: poor propagation of activations and gradients



Solution for deeper networks: residual units

- Non-residual nets learn to map $F(x)$
- Instead we could learn residual mapping $F(x) = H(x) + x$



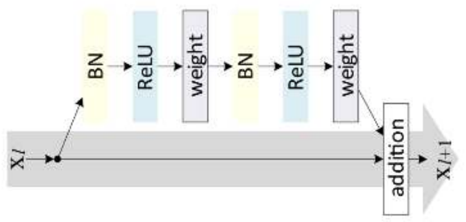
- Sometimes it can be useful to skip parts of the network

→ Motivation: Avoid the problem of vanishing gradient

→ Having skip connections allows the network to more easily learn identity-like mappings

- Can be used as bottleneck that utilises 1 × 1 convolutions
- Idea is to make residual blocks as thin as possible to increase depth and have less parameters

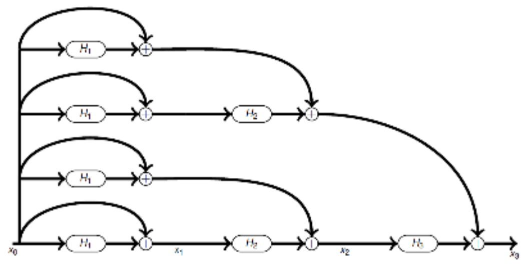
→ Specially useful for deep networks and large input image size



⇒ Training / validation error of deeper nets is now lower

Ensemble view of ResNets

- ResNets behave like ensemble of shallow networks
- $x_{l+1} = x_l + H_{l+1}(x_l)$
- Residual networks allow removal of connections without significant drop in performance



Classical Feed-forward Network vs. Residual Networks

Classical feed-forward network:	Residual networks:
• At layer level: one single path	• At layer level: 2^n different paths
• At neuron level: many different paths of same length	• At neuron layer: many different paths of varying length
→ Exponential in # layers	
• Can compute entirely new representations	• Do not compute entirely new representations

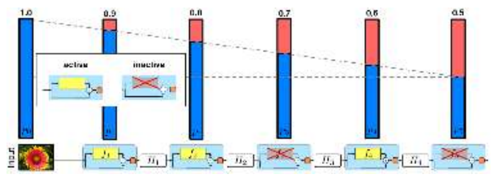
Deep Networks with Stochastic Depth

- Stochastic depth: layer-wise dropout, i.e., drop random layers and bypass with identity

→ Ensemble of exponentially many small networks

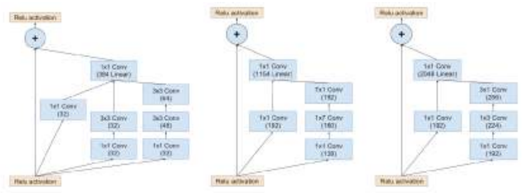
- Network are short during training

→ Decreased training time



Connection of Inception and ResNet

- Faster convergence and better performance than without residual connections



Wide residual networks

- Decrease depth, increase width of ResNet blocks
- Use dropout in residual block
- Power not from depth but from residual connections

Learning Architectures

- Network should be optimized with respect to accuracy and FLOPs

→ Can be done with Grid-search, but this is too time-consuming

- Alternative: Use reinforcement learning with the help of RNNs to generate model descriptions of networks

Recurrent Neural Networks

Motivation

- So far we have feedforward neural networks, which can handle one input at once
- But lots of tasks are sequential or time-dependent

→ Snapshots are often not informative enough

⇒ Temporal context is important

- Simple approach would be to feed the whole sequence into a big network. Bad idea because of inefficient memory usage, difficult training and still no real-time dimension
- Better approach is to model a sequential behavior within the architecture

Difference between RNNs and FFNNs

- Feedforward networks only feed information forward
- With recurrent neural networks, we can:
  - model loops
  - model memory and experience
  - learn sequential relationships
  - provide continuous predictions as data comes in
- RNNs allows us to model real-time structure

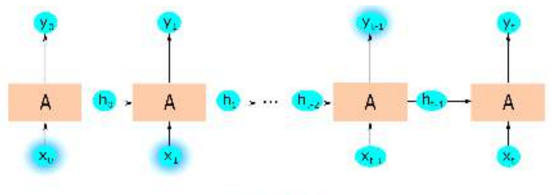
The Elman network

- Current input x_t multiplied by weight matrix
- New: Additional input h_{t-1} (hidden state) of the unit

→ Can be seen as feedback loop that uses information from present and recent past to compute the output y_t

- Unfolded RNN unit: sequence of copies of the same unit (= same weights)

→ Previous input can influence current output



Basic RNN unfolded

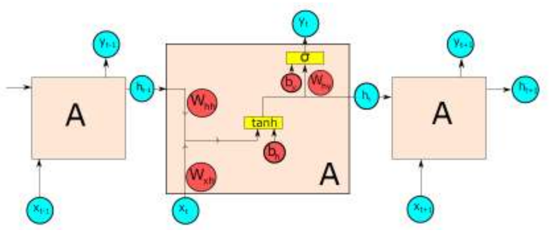
- Central questions: How to compute the update of the hidden state and how to combine hidden state and input to produce the output?
- The hidden state is updated according to the following formula:

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

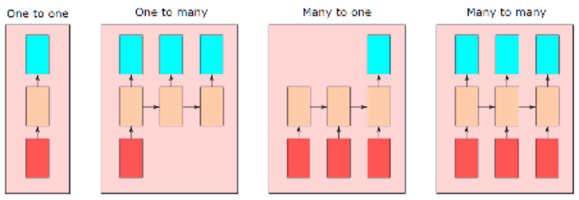
- The output is calculated according to the following formula:

$$y_t = \sigma(W_{hy} \cdot h_t + b_y)$$

→ Remember: $\sigma \equiv$ Sigmoid activation function



Basic Architectures



Examples:

- One to one: Image classification (classic feed-forward)
- One to many: Image captioning
- Many to one: Sentiment analysis
- Many to many: Video classification

→ For deep RNNs, stack multiple units!

Backpropagation through time (BPTT)

- Variant of backpropagation to train unfolded networks
- Compute the forward pass for the full sequence to calculate the loss

Algorithm 1 Forward pass for RNNs

Input: Sequence $X = \{x_1, \dots, x_T\}$

```
1: for t from 1 to T do
2:     $u_t = W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h$
3:     $h_T = tanh(u_t)$
4:     $o_t = W_{hy} \cdot h_t + b_y$
5:     $y_t = \sigma(o_t)$
```

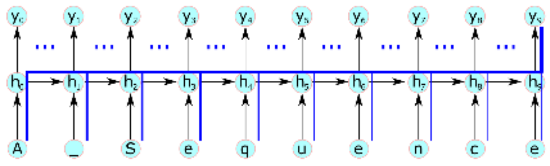
- Compute backward pass through full sequence to get gradients for performing the weight update

Algorithm 2 Backpropagation trough time

```
1: for t from T to 1 do
2:     $\nabla o_t = \sigma'(o_t) \cdot \frac{\partial L}{\partial y_t}(\hat{y}_t, y_t)$
3:     $\nabla W_{hy,t} = \nabla o_t h_t^T$
4:     $\nabla b_{y,t} = \nabla o_t$
       ▷ For t = 0,T we only need the 2. part of the following sum
5:     $\nabla h_t = (\frac{\partial h_{t+1}}{\partial h_t})^T \nabla h_{t+1} + (\frac{\partial o_t}{\partial h_t})^T \nabla o_t$
6:     $\nabla W_{hh,t} = \nabla h_t \cdot tanh'(u_t) \cdot h_{t-1}^T$
7:     $\nabla W_{xh,t} = \nabla h_t \cdot tanh'(u_t) \cdot x_t^T$
8:     $\nabla b_{h,t} = \nabla h_t \cdot tanh'(u_t)$
9: Update weights by summing them up for all time steps
```

- We have to sum up all calculated derivatives because unrolled units are a nwetwork with shared weights!
- Often used loss function: cross-entropy loss

$$L(\hat{y}, y) = \sum L(\hat{y}_t, y_t)$$



- BPTT: One update requires backpropagation through a complete sequence

→ Single parameter update is very expensive!

- Naive solution: Split long sequences into batches of smaller parts

⇒ Might work ok in practice, but blind to long-term dependencies

- Better solution: Truncated backpropagation through time (TBPTT)
  - Main idea: keep processing sequence as a whole and adapt frequency and depth of update

  → Every k_1 time steps, run BPTT for k_2 time steps

  ⇒ Parameter update cheap if k_2 small
- Short term dependencies work fine now, Contextual information nearby and can be encoded in hidden state easily
- Still a problem: Long-Term dependency's

Long-Term Dependency Problem with Basic RNNs

- Harder to connect relevant past and present inputs for longer time spans
- Old acquaintances: vanishing and exploding gradients
  - Layers and time steps of deep RNNs are related through multiplication

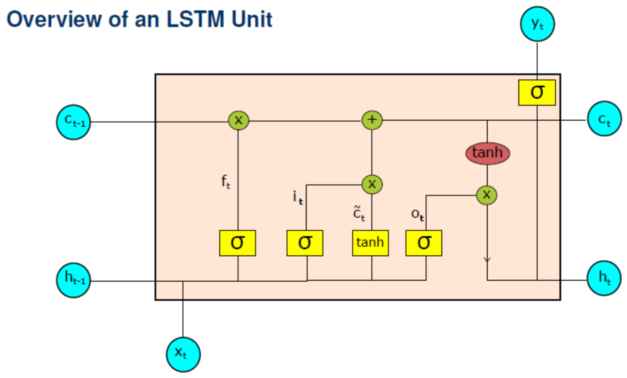
  → As usual, gradients tend to vanish or explode

  ⇒ Exploding gradient relatively easy to solve by truncating gradient
- Additional problem: memory overwriting, because hidden state is overwritten each time step

→ Detecting long-term dependencies even more difficult

Long Short-Term Memory Units (LSTMs)

- Designed in 1997 to solve vanishing gradient and learning long-term dependencies
- Main idea: introduction of gates that control writing and accessing "memory" in additional cell state



What is the LSTM cell state

- Models forgetting and memorizing
- Undergoes only linear changes: no activation function!
- c_t can flow through a unit unchanged

Update steps:

1. Forget gate: Forgetting old information in cell state
   - Key idea: forgetting and memorizing information in separate steps
   - f_t controls how much of the previous cell state is forgotten:

   $$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
2. Input gate: Deciding on new input for cell state
   - Combination of input and hidden state on two paths:

   $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)\tilde{c}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
3. Computing the updated cell state
   - New cell state: Sum of remaining information and new information from input and hidden state
   - Calculated with element-wise multiplication

   $$c_t = f_t \odot c_{t-1} + i_t \tilde{c}_t$$
4. Computing the updated hidden state
   - Important: Cell state and hidden state are updated separately
   - The output y_t directly depends on the hidden state h_t

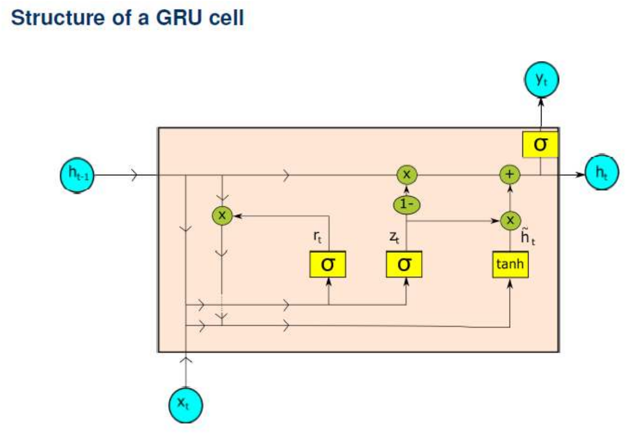
   $$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
   $$h_t = o_t \odot tanh(c_t)$$
   $$y_t = \sigma(h_t)$$

Gated Recurrent Units (GRU)

- Variant of the LSTM unit, but simpler and fewer parameters
- LSTM great idea, but many parameters and difficult to train
- As with LSTM, you have more control over the memory of the hidden states using gates

→ Main difference: no additional cell state

- Gates allow capturing diverse time scales and remote dependencies



Update steps

1. Reset gate: Determines the influence of the previous hidden state
$$r_t = \sigma(W_r[h_{t-1}, s_t] + b_r)$$
2. Update Gate: Determines the influence of an update proposal on the new hidden state
$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$
$\rightarrow z_t$ close to 0: ignore new input
3. Proposing an updated hidden state
   - Combination of input and reset hidden state
   - If r_t is close to 0: ignore previous hidden state
$$\tilde{h}_t = tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h)$$
4. Computing updated hidden state: Update gate controls combination of old state and proposed update
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

→ Units learning short-term dependencies have restrictive reset gates

→ Units learning long-term dependencies have restrictive update gates

Comparison between the different units

- Simple RNNs still have many problems like difficult gradient-based training, overwritten hidden states and no modeling of long-term dependencies
- LSTM and GRU are more advanced structures with similar performance
- Similarities of the advanced structures:
  - Control information via gates
  - Can capture dependencies of different time scales
  - Additive calculation of states preserves errors during backpropagation

  → More efficient training possible
- Differences

LSTM	GRU
Seperate hidden and cell state	Combined hidden and cell state
Controlled exposure of memory content through output gate	Controlled exposure of memory content through output gate
Independent input and forget gate → New memory content independent of current memory	Common update gate for the hidden state → New memory content depends on current memory

Sampling strategies for RNNs

- RNN actually computes probability distribution of the next element
- Greedy search:
  - Concept: At each point, pick the most likely element
  - Generates exactly one sample sequence per experiment
  - BUT: No lookahead possible
- Beam search:
  - Concept: Select k most likely elements
  - Out of all possible sequences that have one of these k elements as a prefix, take k most probable ones
  - Can generate k sequences in one go
- Random sampling:
  - Idea: Sample next word according to output probability distribution
  - Creates very diverse results, can look too random

  → Use temperature sampling

Visualization and Attention Mechanisms

Motivation

- Neural networks do not have to be black boxes
- We need visualization to communicate results between researchers, identify issues and faulty data during training and to understand how our network is learning

Network Architecture Visualization

- It is important to present the architecture of a network, because it can be the main reason for good performance
- Node-link diagrams:
  - Neurons are the nodes, weights are edges
  - Detailed representation, focus on connectivity
  - Only for small (sub-)networks, building blocks
- Block diagrams:
  - Each layer is a solid block and we have just a single connection between layers
  - Blocks represent layers or mathematical operations
  - Arrows show the direction of flow
  - Blocks can have different granularity – often hierarchical descriptions
  - Recommendation: Pick one that clearly represents what you want to show

Visualization of Training

- Most DL libraries provide tools to record and monitor training - use them!

Visualization of Parameters

Motivation

- Networks learn representation of the training data
- We should take a look at it to investigate unexpected/unintuitive behavior

→ Potential causes: Focus on “wrong” features, different noise properties

⇒ (Anecdotal) example: Identification of tanks in photos (Confounds example)

Confounds

- Problem: Networks may learn correlated features instead of identify the correct task
- Important: Not a fault in the learning algorithm, but in the data!

→ ML will focus on the most discriminative features!

- Example: Detects whether instead of tanks
- Good news: If confounder is known, we may be able to correct for it

→ Best strategy: Be aware and avoid!

Adversarial Examples

- With optimization, we can construct pictures without visible difference, but different classification

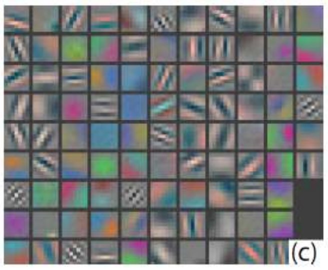
→ Humans: Optical Illusions

- Adversarial examples can be generated to cause a specific mistake
- Example: Toaster sticker

Direct Visualization of Learned Kernels

- Idea: Plot learned filter weights directly
- Easy to implement, easy to interpret for the first layer

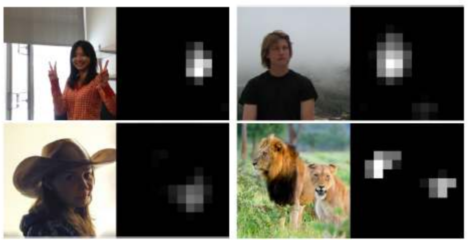
→ Apart from that, mostly uninteresting



First layer filters in AlexNet (11 × 11 conv)

Visualization of Activations

- Idea: Instead visualize activations generated by kernels
- Strong response: feature is present, weak response: feature is absent
- Possible for any layer/neuron in the network, with different resolutions
- Channels may correspond to specific features, e.g., faces:



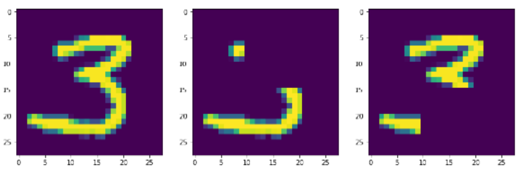
- Drawback: No insight into what exactly caused the response, coarse representation

Investigating Features via Occlusion

- Idea: Move a masking patch around the input image
- If occlusions cause a significant drop in prediction confidence: area important for classification

→ Can identify confounds, e.g. wrong focus

- Shift mask over input generates probability heatmap for a class

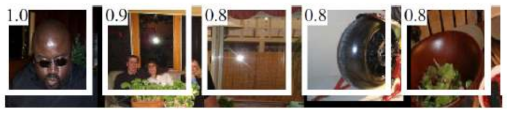


3 with 97.4% confidence 3 with 83% confidence 3 with 94% confidence

Investigating Features via Maximally Activating Images

- So far: Looking at activations for single images
- More general question: Which inputs cause high activations?
- Idea: Find input that activates a specific neuron the most
- Benefits: Easy to implement, "false friends" are comparatively easy to find
- Drawbacks: Neurons don't necessarily have semantic meaning by themselves, rather 'basis vectors' of a representation

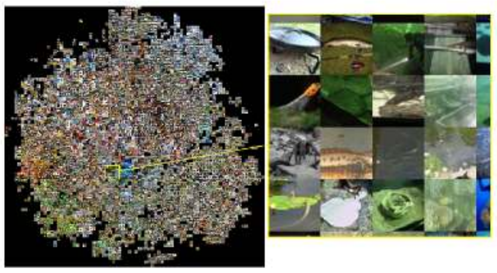
Red flowers (and tomato sauce?)



Specular highlights

t-SNE visualization of CNN

- t-SNE: t-Distributed Stochastic Neighbor Embedding
  - Performs dimensionality reduction for high-dimensional datasets
  - Result: 2-D embedding that respects high-dimensional distances of activations
- Idea: Understand which images the network regards as similar
- Compute activations of the last layer and group inputs with similar activations
- Can help to assess whether the network grasps the correct concept of similarity
- Drawback: 2-D embedding of very high dimensional space, difficult to interpret



Gradient-Based Visualization

Backpropagation for Visualization

- Question: Which pixels are most significant to a neuron? Which would have most affected neuron output, had they been different?

→ For which pixels x_i is $\frac{\partial neuron}{\partial x_i}$ large?

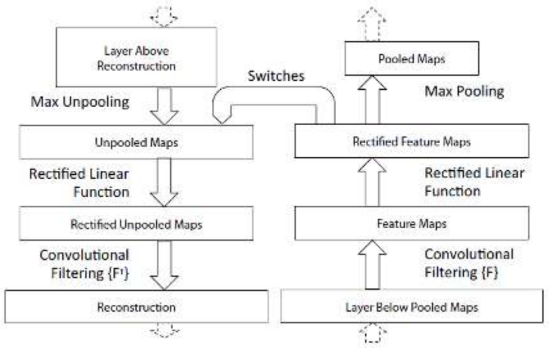
- Backpropagation can be used to compute this gradient for a specific neuron
- Remember: we need a loss that we can backpropagate, hence we use a pseudo loss $f_n(x)$

→ f_n is the activation of an arbitrary neuron of a layer

Feature Visualization – Deconvnet

- Uses a reversed network (deconvnet)
- Input is trained network and a image, than we choose one activation and set all others to zero

→ No training involved here!

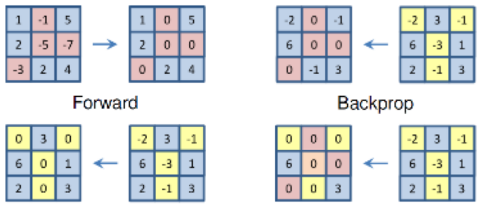


- May reveals feature map focuses on grass / background instead foreground object

Guided Backpropagation

- Mix between Deconvnet and Backpropagation
- Idea: Positive gradients = features the neuron is interested in / Negative gradients = features the neuron is not interested in

→ Set all negative gradients in the backpropagation process to zero



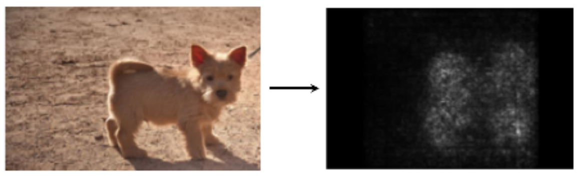
- Interesting for higher-layer neurons, because it can reveal neurons that focus on very abstract features

Visual comparison of the three visualizations:

Deconvnet
Backprop
Guided backprop

Saliency maps

- Instead of investigating what influences neurons, investigate impact of pixels on class score
- Pseudo loss is now unnormalized class score
- Interesting observation: Saliency map “localizes” dog in the image, even though the network was never trained on localization!



Parameter Visualization via Optimization

Google DeepDream / Inceptionism

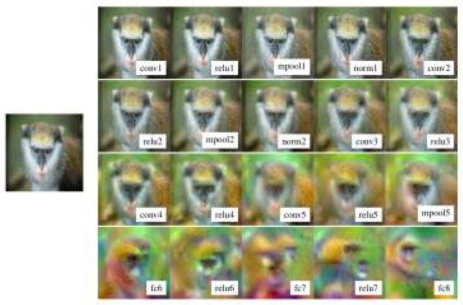
- Intuition: Attempt to understand the inner workings of the network: What it ’dreams’ about when presented with images.
- Idea: Use arbitrary image or noise as input. Now, instead of adjusting network parameters, change image towards high activations of a complete layer

→ Search for images, the layer finds interesting

- Abstract features emerge
- This can reveal hidden weaknesses in the NN classification process

Inversion

- Inversion attempts to construct an image from a given layer activation
- Clearly visible features in the reconstructed image correspond to features which matter most to the CNN
- Inversion can be made till layer 5-6



Attention Mechanisms

What is Attention?

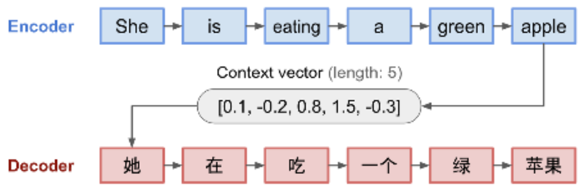
- Humans process data by actively shifting their focus
- Different parts of an image carry different information and can be context sensitive
- Remember specific, related events in the past
- Helps to focus on one task
- We saw: Saliency maps learn attention implicitly
- Attention alone very powerful (Transformer)

Sequence to sequence models (Seq2Seq)

- Encoder/decoder architecture, typically RNNs
- Encoder network receives a input sequence and computes hidden states h_t
- Decoder network receives context vector h_t, computes own hidden state and then generates the output sequence

→ Split allows different length in input/output

⇒ The encoding of complete content is difficult



- We can realize attention for Seq2Seq

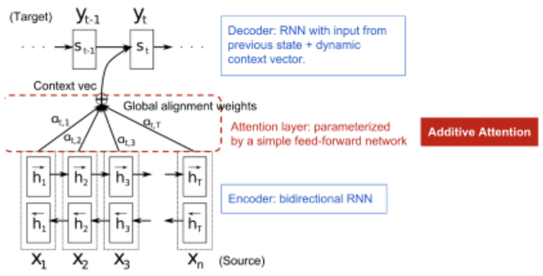
→ Problem: Context vector provides no access to earlier inputs

⇒ Solution: Define a dynamic context vector as combination of all previous hidden states

- Uses a score function represented by a trainable single layer FCN

→ Determines alignment: Which inputs are important for which outputs

⇒ Alignment scores allow interpretation



Soft Attention vs. Hard Attention

- So far: Attention is computed over all patches/inputs: Soft/global attention
  - + Model is fully differentiable
  - - Not effective/efficient for large inputs
- Alternative: Hard attention, fixed size glimpses on the input,
  - + Lower computation times
  - - Not differentiable, requires e.g. reinforcement learning techniques to train

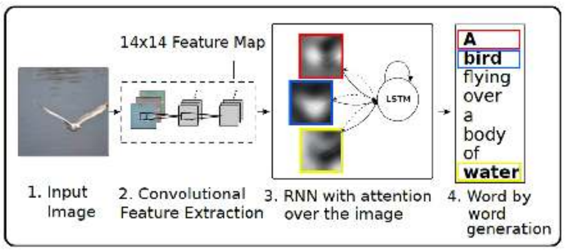
Show-Attend-Tell

- Task: Automatic generation of image captions

- Different elements and their relationship in the image trigger different words

→ Attention mechanism to improve caption quality

- Attention weighs CNN feature maps according to caption produced so far

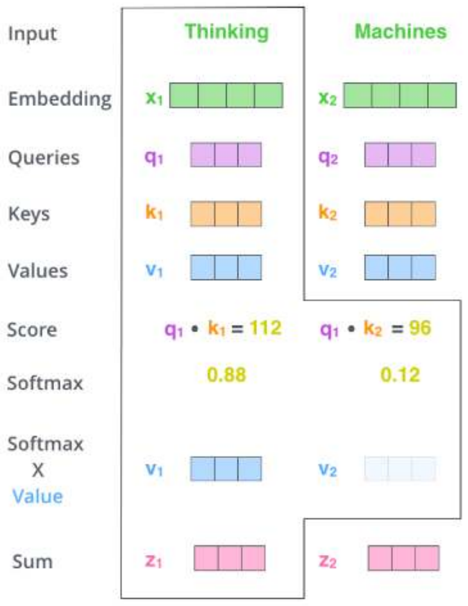

14x14 Feature Map
LSTM
A
bird
flying
over
a
body
of
water
1. Input Image
2. Convolutional Feature Extraction
3. RNN with attention over the image
4. Word by word generation

Self-Attention

- Computes attention of sequence to 'itself'
- Allows to enrich representation of tokens with context information
- Important for machine reading, question answering, reasoning ...

Attention is all you need (AIAYN)

- Machine translation based only on attention, no convolution or recurrence
- Core idea: Iteratively improve representation by self-attention
- Each input token is translated into vector of same length using an embedding algorithm
- Self attention: For each token, compute
  - query q: what a token is looking for
  - key k: description for query
  - value v: potential



- Multi-Head attention: Multiple attention vectors per token

→ Recombination uses a fully connected layer

- Allows for integration of knowledge independent of distance
- Positional encoding still allows to learn convolution-like steps
- State-of-the-art performance and faster training

Deep Reinforcement Learning

Sequential Decision Making

- Also called Multi-armed bandit problem
- Consists of three parts:
  - An Action a at time t which can be chosen from a set A
  - A reward r_t which is generated by an unknown probability density function $p(r \mid a)$
  - A policy $\pi(a)$ which formalizes how to choose an action a
- Aim: Find an action a that produces the maximum expected reward over time ($\max_a \mathbb{E}[p(r \mid a)]$)
- Difference to supervised learning: No feedback on what action to choose
- We can form a one-hot encoded vector r which reflects which action from a caused the reward

→ Estimate the joint pdf: $Q_t(a) = \frac{1}{t}\sum r_i$

- $Q_t(a)$ is called action-value function, which changes in every step
- Reward is maximized by a policy $\pi(a)$ which chooses $\max_a Q_t(a)$

→ Exploitation of a known good action

⇒ This deterministic policy is called greedy action selection

- Therefore we need to obtain samples r_a

→ This means we cannot follow the greedy action selection policy for learning

⇒ We need to explore by selecting other moves to be potentially better

- To sample discrete actions from $\pi(a)$, we can use different policies:
  - Uniform random for a complete random choice in each iteration

$$\pi(a) = \frac{1}{|A|}$$

  - Epsilon greedy with $\epsilon < 1$ to choose sometimes a new strategy

$$\pi(a) = \begin{cases} 1-\epsilon & a = \max_a Q_t(a) \\ \epsilon/(n-1) & else \end{cases}$$

  - Softmax with decreasing temperature parameter to decrease exploration over time

$$\pi(a) = \frac{e^{Q_t(a)/\tau_t}}{\sum e^{Q_t(a)/\tau_t}}$$

Reinforcement Learning

Motivation

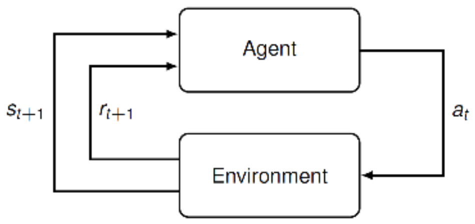
- Extension of the multi-armed bandits problem:
  - We introduces a state of the world at any time t: s_t

  → Rewards now additionally depend on the state s_t

  - In the full reinforcement learning problem, actions influence the following state

→ This setting is known as contextual bandit

Markov Decision Processes



- Generally, policies now depend on s_t

→ We can extend the uniform random policy to be independent from s_t

⇒ However there's no reason to believe that this policy is any good

What is a good policy?

- Preliminary we have to state two kinds of tasks, episodic and continuing tasks

→ Can be unified by using a terminal state in episodic tasks

- Goal is to maximize the future return

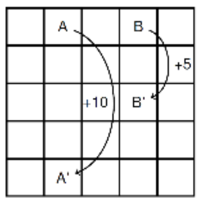
$$\max_{\pi(s_t,a_t)} g_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$$

→ γ is a discount reducing influence of rewards far in the future

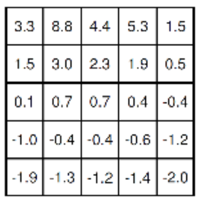
Policy Iteration

- We introduce the state-value function $V_\pi(s)$ to predict future reward

$$V_\pi(s) = \mathbb{E}_\pi[g_t \mid s_t]$$



The definition of the gridworld



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

$V_\pi(s)$ for the uniform random policy

- By using greedy action selection policy on this $V_\pi(s)$, we can get a better policy

→ $V_\pi(s)$ is used to guide our search for good policies

→ Extend the action-value function

$$Q_\pi(s,a) = \mathbb{E}_\pi[g_t \mid s_t, a_t]$$

- Note: There can only be one optimal V*(s), which can be obtained by maximizing is in terms of the policy
- Policies can now be ordered: $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)\ \forall s \in S$
- More than one optimal policy is possible, because every policy with $V_\pi = V^*$ is optimal

→ Given either V^* or Q^* an optimal policy is directly obtained by greedy action selection

⇒ Still need to compute $V^*(s), Q^*(s,a)$

- For this the Bellman equations can be utilized
- They are consistency conditions for the value functions
- The Bellman equations form a system of linear equations which can be solved for small problems
- Better: Iteratively solve, by turning the Bellman equations into update rules
- However if we use greedy action selection an update of $V_\pi(s)$ is simultaneously an update of π(s)
- Stop iterating if the policy stops changing
- According to the policy improvement theorem, the greedy action selection together with updating is guaranteed to work for improving the policy
- Both policy iteration and value iteration require using the updated policies during learning to obtain better approximations to $V^*(s)$

→ For this reason we call them on-policy algorithms

- Additionally we assumed the state-transition pdf and reward pdf are known

Other Solution Methods

- Monte Carlo Techniques: Off-policy - learns V*(s) by following any arbitrary π(s, a)
- Temporal Difference Learning: On-policy, which does not need information about dynamics of the environment
- Q Learning: Off-policy and a temporal difference type of method

→ None of those need information about dynamics of the environment

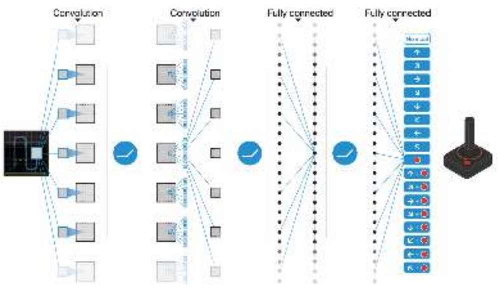
- Use the universal approximation theorem an parameterize the policy

→ Known as policy gradient

Deep Reinforcement Learning with Deep Q Learning

Atari Games

- Project of Google DeepMind in 2013
- Idea: Let a neural network play Atari games!
- Input: Current and three subsequent video frames from game
- Processed by network trained with reinforcement learning
- Goal: learn best controller movements



- Deep Q-network: Deep network that applies Q-learning
- 18 outputs associated with an action

→ Each output estimates optimal action value for its action given the input

- Instead of label and cost function, update to maximize reward
- ϵ-greedy policy with ϵ decreasing to a low value during training
- Uses mini-batches to accumulate weight updates
- Problem: The target is a function also containing w_t

→ Target changes simultaneously with the weights we want to learn!

⇒ Training can oscillate or diverge

- Idea: Use a second target network, where after k steps the weights of action-value network are copied to a duplicate network and fixed
- Use experience replay to reduce the correlation between updates
  - Memory accumulates experiences
  - To update the network, draw random samples from memory, instead of taking the most recent ones

  → Removes dependence on current weights and increases stability

Alpha Go

- Exhaustive search is infeasible!
- First improvement compared to a full tree search: Monte Carlo Tree Search

→ Idea: Run many Monte Carlo simulations of episodes (=entire Go games) to select action (=where to place a stone)

→ Starting from a root node representing the current state, MCTS iteratively extends the search tree

⇒ Problem: Estimation via MCTS not accurate enough for Go.

- Deep Neural Networks for Go with three different networks:
  - Policy network: Suggests the next move in leaf nodes for extension
  - Value network: Given the current board position, get chances of winning
  - Rollout policy network: Guide rollout action selection
- Alpha Go Zero: Solely trained with reinforcement learning and playing against itself

Unsupervised Deep Learning

Motivation

- We can have a huge number of variations between different datasets (e.g. size, complexity)
- Sometimes we do not have enough labeled data, hence we use unsupervised learning
- Applications: Clustering, network initialization, dimensionality reduction

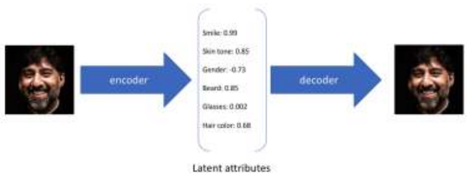
Autoencoder

Concept of AE

- Special case of feed-forward neural networks
- Consists of two parts: Encoder ($y = f(x)$) and Decoder ($\hat{x} = g(y)$)

→ We train for $\hat{x} = x$, AE tries to learn an approximation of the identity

- We can use known loss functions according to the distribution of your $\hat{x}$
- 'Traditional' AEs compute a deterministic feature vector describing the attributes of the input in latent space



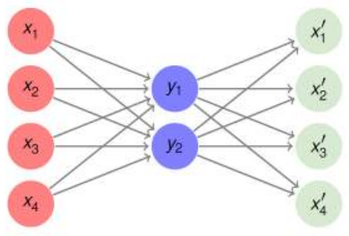
- We can enforce information compression by using undercomplete AE

→ Prevent network from simply copying the input

- We can also model sparse autoencoder, which includes more hidden units compared to the input units

→ You have to add a additional regularization to enforce sparsity in the activations

- AE acts as bottleneck layer
- Other variations: Convolutional AE, Denoising AE (regularization effect), stacked AE



Variational Autoencoders

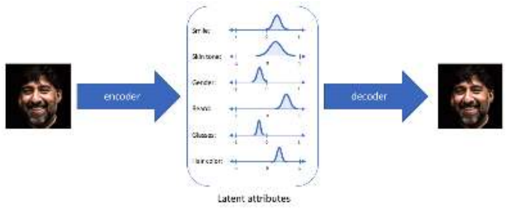
- Difference to traditional AEs: use a variational approach to learn the latent representation
- Linked to the content generation problem
- A variational autoencoder can be defined as being an autoencoder whose training is regularised to avoid overfitting and ensure that the latent space has good properties that enable generative process

→ It allows to describe the latent space in a probabilistic manner

→ Describe each latent attribute as probability distribution

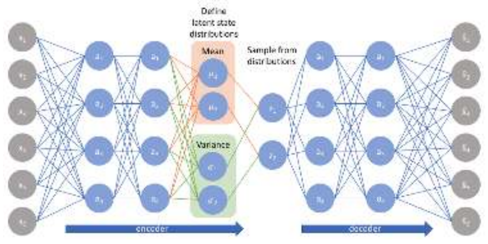
⇒ Allows to model uncertainty in the input data

- Representation as probability distribution enforces a continuous, smooth latent space representation

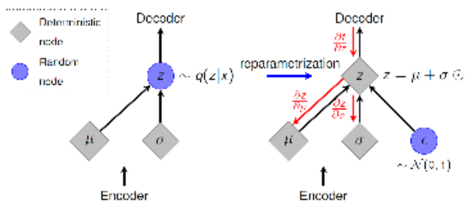


- Training variational AE is problematic, because the network contains a sampling operator
- We change the interpretation of the activations in the red and green box

→ Mean and variance of our latent distribution

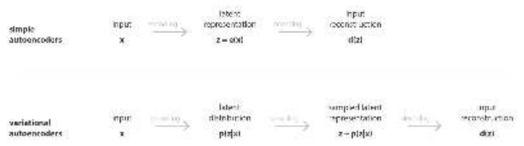


- Because of the randomness of the sampling operator, we cannot backpropagate at this position
- Instead of evaluating to a single point, we evaluate to a distribution and choose a point randomly for the decoder
- For using backpropagation, we have to add a path by reparametrization:



- This way, we can make it deterministic
- New data can be generated by sampling from distributions in the latent space with reconstructing them by decoder
- Pros: Principled approach to generative models and latent space representation can be useful for other tasks
- Cons: Only maximizes lower bound of likelihood and samples in standard models often of lower quality compared to GANs

Comparison of AE:



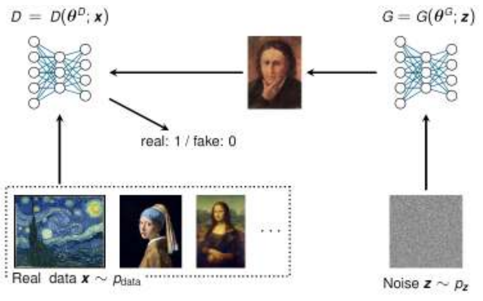
simple autoencoders
input x
latent representation z = e(x)
input reconstruction d(z)
variational autoencoders
input x
latent distribution p(z|x)
sampled latent representation z ~ p(z|x)
input reconstruction d(z)

Generative Adversarial Networks

GANs are generative models that use supervised learning to approximate an intractable cost function.

Principle of GANs

- We play a game between a generator and a discriminator
- Generator creates a picture and the discriminator has to tell whether the incoming data is real or fake



$D = D(\theta^D; \boldsymbol{x})$
$G = G(\theta^G; \boldsymbol{z})$
real: 1 / fake: 0
Real data $\boldsymbol{x} \sim p_{\text{data}}$
Noise $\boldsymbol{z} \sim p_{\boldsymbol{z}}$

- D is trained to distinguish real data samples from fake ones
- G is trained to fool D

→ Equilibrium is a saddle point of the discriminator loss

⇒ Summarize game with a value function specifying the discriminator's payoff and play minimax game

- We can only in theory calculate an optimal D and G
- Hence we use GANs as approximation mechanism which use supervised learning

- Loss function: G maximizes log-probability of D being mistaken

→ Heuristically motivated: fights vanishing gradient of G when D is 'too smart'

- Other popular loss functions:Feature matching loss, Wasserstein Loss
  - G trained to match expected value of features f (x) of intermediate layer of D

  → The expected output of the generator should be the same as the original input

  - Prevent 'overtraining' of G on current D
- GANs are pretty good at generating low-resolution images
- High-resolution images are much more difficult!

→ Subsequently increase resolution and add detail with pyramid of GANs

GANs in Comparison to Other Generative Models

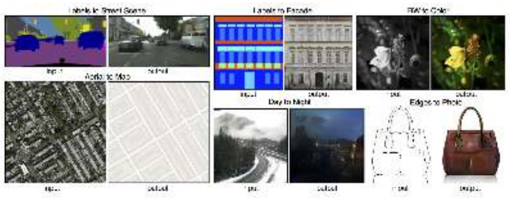
- Ability to generate samples in parallel
- Very few restrictions because e.g no markov chain needed
- No variational bound is needed
- GANs known to be asymptotically consistent since the model families are universal function approximators

Conditional GANs

- Problem: Generator creates a fake generic image : is not specific for a certain condition/characteristic
- Example: text to image generation – image should depend on the text

→ Idea: Provide additional vector y to networks to encode conditioning

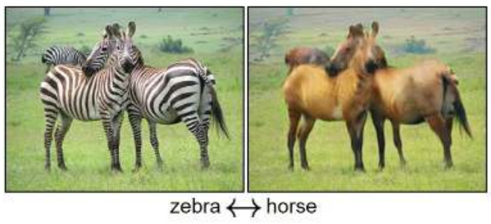
- Generator G receives the latent vector z and a conditioning vector y
- Discriminator D receives x and also y
- Example Face Generation: Generator/Discriminator learn to operate in modes:
  - Generator learns to generate a face with a certain attribute
  - Discriminator learns to decide whether the face contains attribute
- Image To Image is just a Conditional GAN!



Cycle Consistent GANs

- Image to Image GAN should generate plausible results w.r.t. input
- Paired data difficult/impossible to obtain
- Cycle consistency loss: Couple GAN with trainable inverse mapping
- Two discriminators DY and DX
- Cycle consistency ensures that both versions are not recognizable by D

→ We achieve better forth and back translation

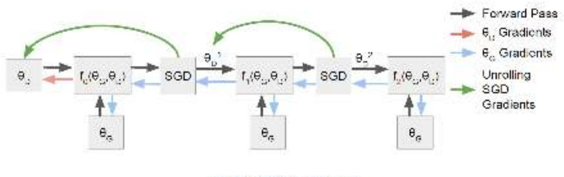


Deep Convolutional GANs (DCGAN)

- Replace any pooling layer with strided convolutions (D) and transposed convolution (G)
- Remove fully connected hidden layers for deeper architectures
- G: Use ReLU activation except for output layer which uses tanh
- D: LeakyReLU activation for all layers
- Use virtual batch normalization: Create reference batch R of random samples chosen and fixed once at the start of training

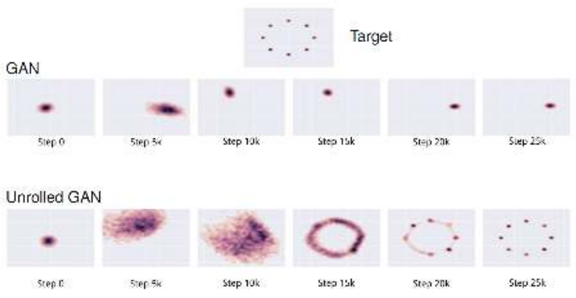
Mode Collapse

- G rotates through the modes of the data distribution
- Never converges to a fixed distribution
- Problem: min-max not exchangable!
- D in inner loop: convergence to correct distribution
- G in inner loop: place all mass on most likely point
- Solutions: Minibatch discrimination or unrolled GANs
- Minibatch discrimination:
  - Intuition: Allow D to look at multiple samples in combination to help G avoid collapsing
  - Extract features from intermediate layer
  - Add minibatch layer that computes for each feature a similarity to all other samples of the mini-batch and concatenate similarity vector to each feature
  - D still outputs 0/1 but now uses the similarity to all samples in the mini-batch as side information
- Unrolled GAN:



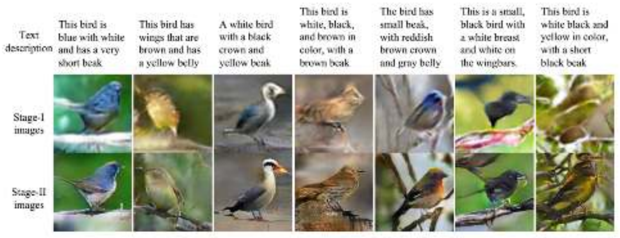
Unrolled GAN with $k = 3$

  - Ideally: $G^* = \min_G \max_D V(G, D)$
  - But essentially, we ignore the max operation when computing G's gradient
  - Build computational graph describing k steps of learning in D
  - Back-propagate through all k steps when computing G's gradient



StackGANs

- Task: Given some text, generate a fitting image
- Decompose problem: Sketch-refinement using a two-stage conditional GANs
- Stage-I GAN: Draws low resolution images
- Stage-II GAN: Add Generates high resolution images


Text description
This bird is blue with white and has a very short beak
This bird has wings that are brown and has a yellow belly
A white bird with a black crown and yellow beak
This bird is white, black, and brown in color, with a brown beak
The bird has small beak, with reddish brown crown and gray belly
This is a small, black bird with a white breast and white on the wingbars.
This bird is white black and yellow in color, with a short black beak
Stage-I images
Stage-II images

Segmentation and Object Detection

Segmentation

Motivation

- Goal: partition images into different segments
- Segments are regions that delineate meaningful objects
- Label regions with an object category label
- Each pixel gets a semantic class

→ Pixel-wise dense classification

- Applications: Medical Imaging, Autonomous driving, robotics

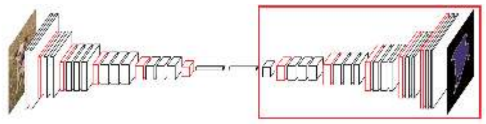
Evaluation Metrics

- Usefulness of segmentation depends on many factors: Execution time, memory footprint and quality
- Quality of a method can be assessed by different metrics
- Assumptions:
  - k + 1 classes including void or background
  - p_{ij} is the amount of pixels of class i inferred to belong to class j

  → p_{ij} represents the number of true positive
- Pixel Accuracy (PA):Ratio between the amount of correctly classified pixels and the total number of pixels
- Mean Pixel Accuracy (MPA): Average ratio of correctly classified pixels per-class basis
- Mean Intersection over Union (MIoU): Ratio between the intersection and the union of two sets

Fully Convolutional Networks for Segmentation

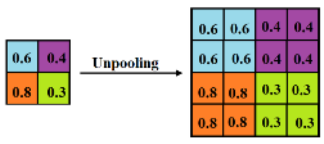
- Transform fully connected layers to convolutional layers
- Output independent of size
- Main problem: Segmentation is coarse
- Use encode/decoder network for higher resolution
- Network has to learn to decode (map) the output of the encoder to pixel-wise predictions



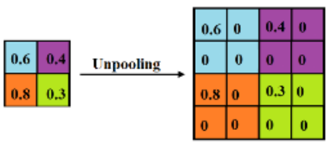
- Examples: SegNet, U-net

Upsampling

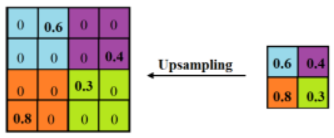
- Decoder requires upsampling to enable pixel-wise prediction
- Different options would be: unpooling, transpose convolution
- Nearest neighbor unpooling: Duplicate the value of the element for the region



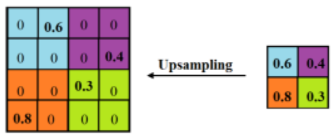
- Bed of Nails: Take the one value and make the others zero



- Using max pooling indices: Save location of max



- Transpose convolution: learnable upsampling, where filter moves 2 pixels in the output for every 1 pixel in the input



→ Transpose convolution is backward pass of normal convolution

- Transpose convolution has uneven overlap when the kernel size is not divisible by the stride
- In practice: avoid this case by choosing dividable kernel size

Integrating Context Knowledge

- Semantic segmentation requires integration of information from various spatial scales

→ Need to balance local and global information

⇒ Local information crucial to achieve good pixel-level accuracy

⇒ Global context of the image enables to resolve local ambiguities

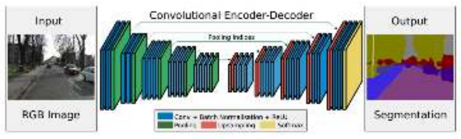
- CNNs struggle with this balance

Alternative methods:

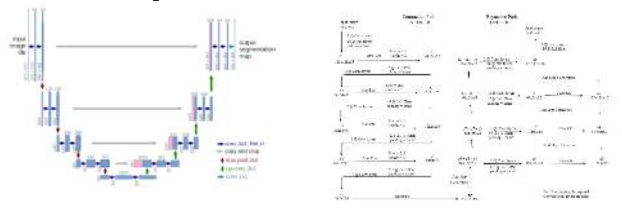
- Long et al.'s Fully Convolutional Networks
  - Idea: Add links combining the final prediction and previous (lower) layers with finer strides
  - Upsampling using learnable transposed convolutions
  - Additional 1 × 1 convolution after pooling layer, predictions are added up

  → Make local predictions with global structure
  - Refinement of coarse segmentation
- SegNet
  - Decoder: Upsampling and convolution layers followed by softmax
  - Each upsampling layer corresponds to a max-pooling layer in encoder

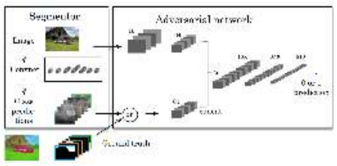
  → provides context information



- U-Net
  - Network consists of encoder and decoder
  - Encoder: A contracting path to capture context and follows a typical architecture of a CNN
  - Decoder: A symmetric expansion path for localization. It consits of unsampling of feature maps followed by a 2 × 2 convolution that halves the number of feature channels. In addition we have a concatenation with the corresponding cropped feature map from the contracting path
  - The training strategy relies on use of data augmentation
  - Include skip connections

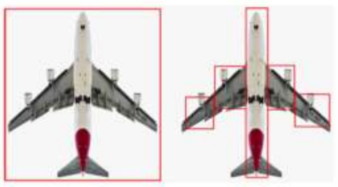


- Adversarial Networks for Segmentation
  - Optimize Segmentor with hybrid loss function with two terms
  - → Term 1: Usual pixel-wise multi-class cross-entropy for semantic segmentation
  - → Term 2: Loss based on min-max game with Discriminator
  - Segmentor is trained both on the segmentation and on fooling the discriminator
  - Multi-task learning with adversarial task



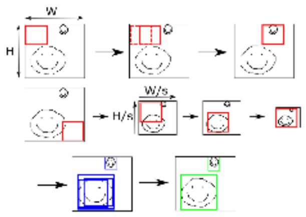
Object Detection

- Goal: Object localization and classification
- Classical Solution: Hypothesize bounding boxes, then Resample selected boxes an last apply classifier
- Bounding Box: Smallest box by some measure that fully contains the object in question

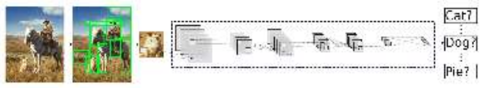


Modern Approaches

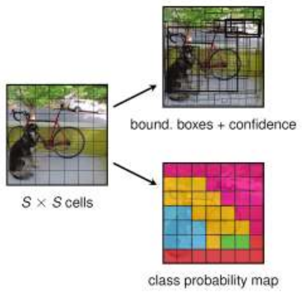
- Sliding window
  - Classify each possible window by CNN
  - Use multiple image scales to detect objects of different sizes
  - Problem: Large number of predictions, but many with low confidence
  - → Keep only high confidence areas
  - + Trained classification network can be used for object detection directly
  - - Computationally inefficient: One pass through the network for every patch!



- Region-based Detectors
  - First find interesting regions, then classify by CNN
  - We improve efficiency by only considering interesting regions (ROIs)
  - ROI pooling: Polling the regions of interest with max pooling
  - Candidate objects by grouping pixels of similar texture or color
  - Apply for different sized windows
  - Produces only a few object proposals in constrast to number of possible windows
  - Essentially a form of segmentation
  - For each region proposal window, wrap it to standard window size, use CNN for feature extraction and classify it
  - + Improved retrieval rate at that time (2013) by more than 30%
  - Much faster... but still slow and not end to end
  - Speed up: Pass full image through the network and use Feature maps. Apply region proposals to last conv layer, then use Spatial pyramid pooling layer to pool to fixed size using max-pooling



- Single-Shot Detectors
  - Joint detection and classification
  - You Only Look Once (YOLO) algorithm: Combined bounding box prediction and classification in one network
  - YOLO9000 is an improved version of YOLO advertised as Better, Faster, Stronger



  - Alternative: Single-Shot Multi-Box Detector
  - All single shot detectors evaluate many hypothesis locations, but most of them are easy negatives
  - This imbalance is not addressed by current training
  - By using Focal loss, we pay less attention to easy examples

Instance Segmentation

- Next step after semantic segmentation
- Main goal: detect different instances of the same class
- Number of instances initially unknown, pixel-wise prediction as in semantic segmentation not sufficient

→ Combination of object detection and semantic segmentation

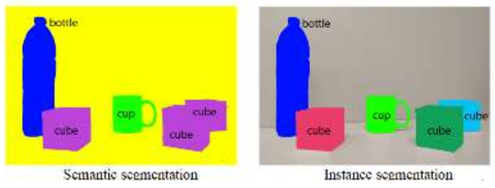
- Example: Mask R-CNN: Object detection solves instance separation, segmentation refines bounding boxes per instance

→ Step 1: Region proposal: proposes candidate object bounding boxes

→ Step 2:Classification, bounding-box regression and segmentation in parallel

Difference between semantic segmentation, instance segmentation and object detection

Semantic segmentation aims to predict a class label to each pixel of an image.Object detection usually predicts bounding boxes for each objects, while instance segmentation aims to dis-criminate different instances (samples) of the sample class by assigning a class label as well as associating an instance identifier for each pixel.



bottle
cup
cube
cube
cube
Semantic segmentation
bottle
cup
cube
cube
cube
Instance segmentation