Summary

Introduction to Deep Learning (Technische Universität München)

Contents

1 Machine Learning Basics

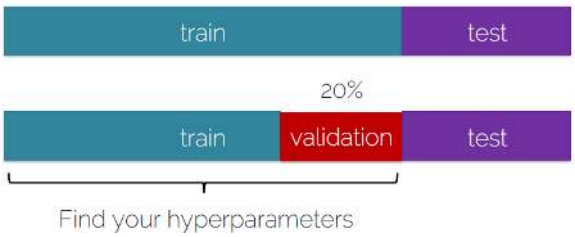
1.1 Machine Learning

Unsupervised vs. Supervised learning

- Unsupervised learning
  - No label or target class
  - Find out properties of the structure of the data
  - e.g. Clustering (k-means, PCA)
- Supervised learning: Labels or target classes
- Reinforcement learning: Agent learns by reward

1.2 A simple classifier

- k-NN classifier that looks at distance of k neighbors
  - Hyperparameters: Distance (e.g. L1, L2) and k (number of neighbors)
  - Use cross validation to tune hyperparameters: Different split of training data into train and validation set for different test runs



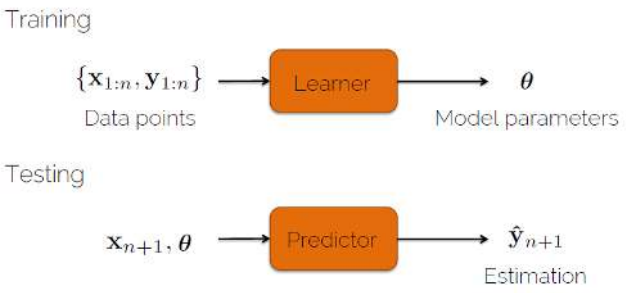
- 1-NN classifier is perfect for training data, but classifier that includes more neighbors generalizes better

1.3 Linear Regression

Linear regression basics

- Supervised learning
- Goal: find a linear model that explains a target y given the inputs X

- Training and Testing:



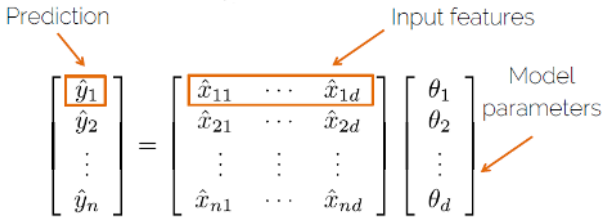
Linear prediction

- Form of a linear model:

$$\hat{y}_i = \sum_{j=1}^{d} x_{ij}\theta_j = x_{i1}\theta_1 + x_{i2}\theta_2 + ... + x_{id}\theta_d$$

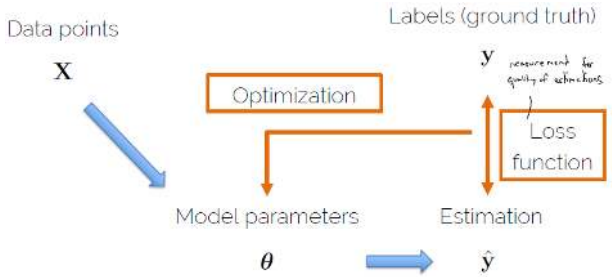
  with input dimension d, input data x_{ij} and weights θ

- First term $x_{i1}1$ is usually set to 1 and θ_1 is called bias
- Matrix notation: $\hat{\boldsymbol{y}} = \boldsymbol{X}\theta$



How to obtain the model?

- Loss function: measures how good my estimation/model is and tells the optimization method how to make it better
- Optimization: changes the model in order to improve the loss function (i.e. the model)



Linear regression: loss function - least square estimate

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Optimization: Linear least squares

- Goal: Minimize Loss function:

$$\min_{\boldsymbol{\theta}} J(\theta) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i\theta - y_i)^2$$

- Matrix notation:

$$\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = (\boldsymbol{X\theta} - \boldsymbol{y})^T(\boldsymbol{X\theta} - \boldsymbol{y})$$

- Convex problem: there exists a closed-form solution
- Calculate derivative and set it equal to 0:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 2\boldsymbol{X}^T\boldsymbol{X\theta} - 2\boldsymbol{X}^T\boldsymbol{y} = 0$$

$$\boldsymbol{\theta} = (\boldsymbol{X^T X})^{-1}\boldsymbol{X^T y}$$

- There are also other optimizations available

Optimization: Maximum Likelihood Estimate

- A method of estimating the parameters of a statistical model given observations by finding the parameter values that maximize the likelihood of making the observations given the parameters
- Approach

$$\boldsymbol{\theta}_{ML} = arg\max_{\theta}\prod_{i=1}^{n} p_{model}(y_i|\boldsymbol{x_i}, \boldsymbol{\theta})$$

- We can replace the product by applying the logarithmic property $log_c(ab) = log_c(a) + log_c(b)$:

$$\boldsymbol{\theta}_{ML} = arg\max_{\theta}\sum_{i=1}^{n} \log p_{model}(y_i|\boldsymbol{x_i}, \boldsymbol{\theta})$$

- Assuming Gaussian distribution, we get the same result for the optimization as for Linear least squares

$$\boldsymbol{\theta} = (\boldsymbol{X^T X})^{-1}\boldsymbol{X^T y}$$

1.4 Logistic Regression

Regression vs. Classification

- Regression: predict a continuous output value (e.g. temperature of a room)
- Classification: predict a discrete value
  - Binary classification: output is either 0 or 1 (e.g. logistic regression)
  - Multi-class classification: set of N classes

Sigmoid function for binary predictions

- Can be interpreted as a probability

- Formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Logistic regression basics

- Probability of a binary output (with Bernoulli trial):

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) = \hat{\boldsymbol{y}} = \prod_{i=1}^{n} \hat{y}_i^{y_i} \left(1 - \hat{y}_i\right)^{(1-y_i)}$$

Loss and cost function

- Loss function (basic expression for one sample): cross-entropy loss

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - y_i)$$

- Cost function (covering all samples):

$$\mathcal{C}(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} y_i \log \hat{y}_i + (1 - y_i) \log(1 - y_i)$$

- There is no closed form solution for optimization. We therefore use gradient descent

1.5 Gradient descent

Basics

- To minimize the function we want to follow the slope of the derivative
- We therefore step in the direction of the negative gradient:

$$x' = x - \epsilon \nabla_x f(x)$$

- ϵ is called learning rate and defines the step size
- With gradient descent, we might not find the global optimum but end in a local optimum (depends e.g. on initialization)
- Local optima might be good enough to solve our problem

Gradient descent for least squares

- easy to calculate, because we have an analytic solution for the gradient

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \left(2\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\theta} - 2\boldsymbol{X}^T \boldsymbol{y}\right)$$

- convex and therefore always converges to the same solution

Gradient descent for logistic regression

- Harder, because there is no analytical solution
- Solved with computational graphs and backpropagation of gradients

1.6 Regularization

Basic idea

- Add a regularization term $\lambda R(\boldsymbol{\theta})$ to the loss (example for linear least squares):

$$J(\boldsymbol{\theta}) = (\boldsymbol{X\theta} - \boldsymbol{y})^T(\boldsymbol{X\theta} - \boldsymbol{y}) + \lambda R(\boldsymbol{\theta})$$

- Goal: Make the model work better for test data by preventing overfitting (better generalization of the model by increasing training error and decreasing validation error)

L1 regularization

- Focus the attention to a few key features
- Formula:

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{n} |\theta_i|$$

L2 regularization

- Take all information into account to make decisions
- Formula:

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{n} \theta_i^2$$

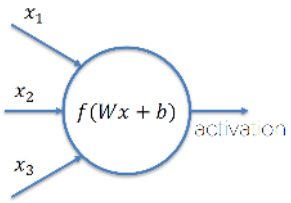
2 Neural Networks

2.1 Basics

Neural Network

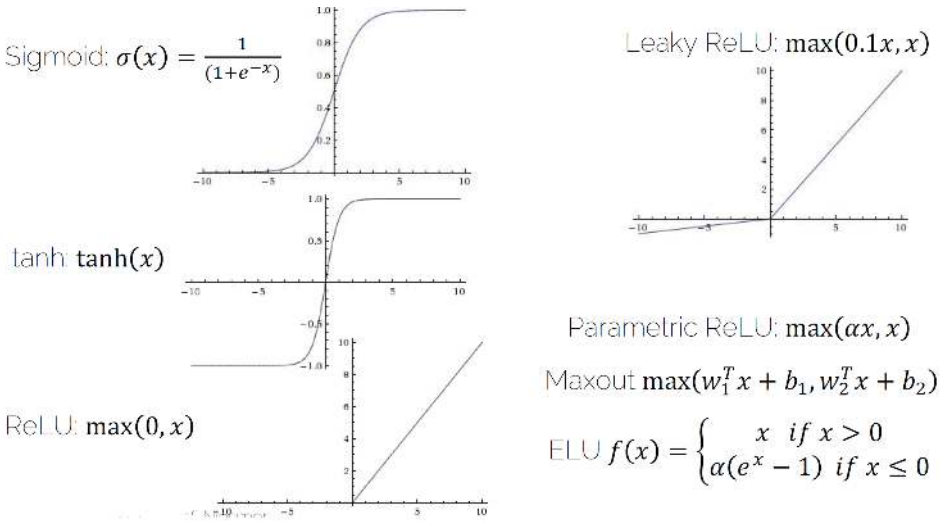
- Linear score function $f = Wx$
- Functions are nested and combined with non-linearity:
  - 2-layers: $f = W_2 \max(0, W_1 x)$
  - 3-layers: $f = W_3 \max(0, W_2 \max(0, W_1 x))$
  - ...
- Non-linearity allows to learn more complex functions ("more powerful")
- Short summary of a Neural Network:
  - Given a dataset with ground truth training pairs $[x_i, y_i]$
  - Find optimal weights $\boldsymbol{W}$ using stochastic gradient descent, such that the loss function is minimized:
    - Compute gradients with backpropagation
    - Iterate many times over training set

Neurons



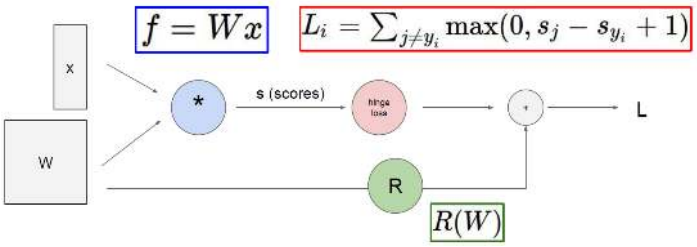
- Linear function: $Wx + b$
- Non-linearity: $f(x)$
- Every neuron computes $f(Wx + b)$

Activation Functions

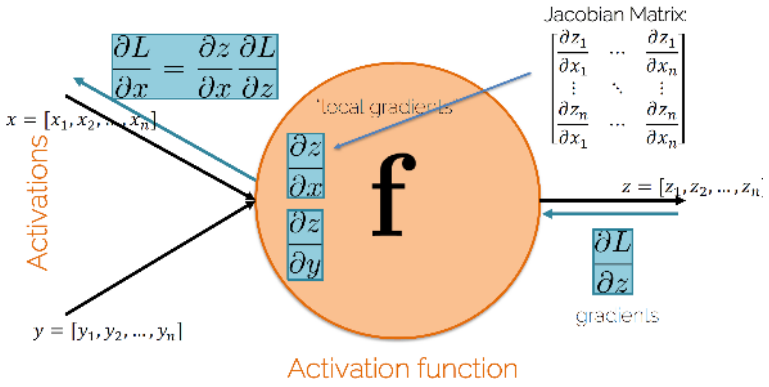


2.2 Backpropagation

- Calculate gradients numerically if function becomes too complex for analytical solutions
- Analytically solutions would at least partly be possible, but can't e.g. be parallelized
- Possible problems and solution:
  - Solution for multiple outputs in a compute node: take average
  - Solution for loops: No loops in computational graphs
  - Values of forward pass required for backward pass: Cache results of forward pass to reach faster runtime in backward pass
- Computational graph for Linear activation with hinge loss and regularization:



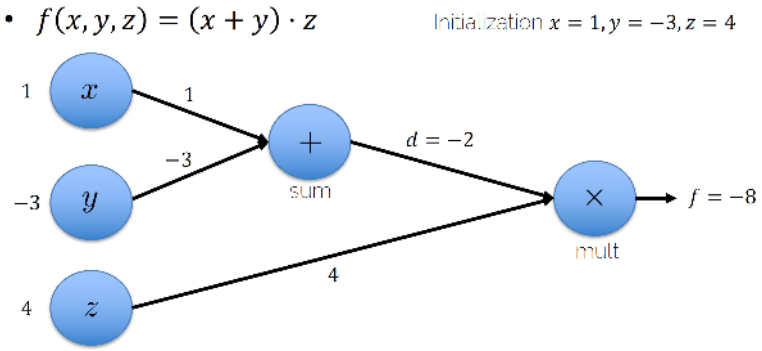
- If input and output values of a function are vectors:
  - Gradients also become vectors
  - We need to calculate the derivative of every output element w.r.t. every input element
  - Hence, we receive a Jacobian Matrix
  - Example:



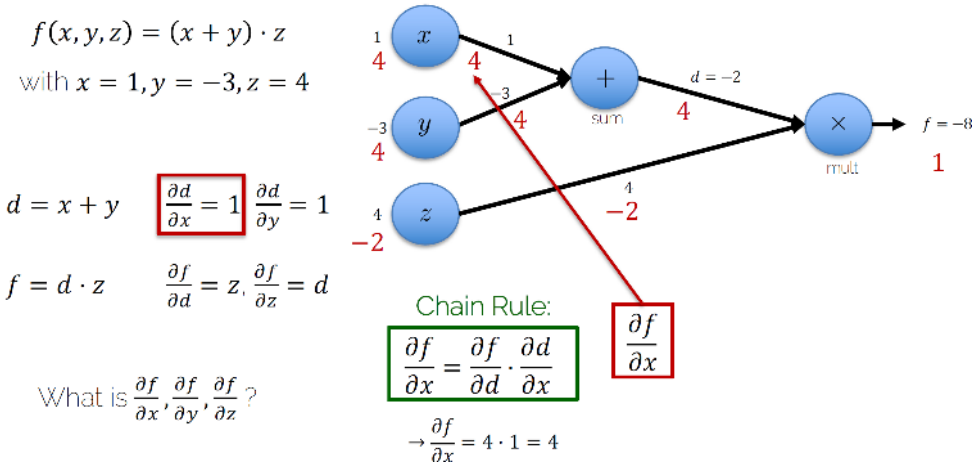
2.3 Computational Graphs and Neural Networks

Computational Graphs

- Directed graph where nodes correspond to operations or variables
- Neural networks can be interpreted as a computational graph
- To compute output we can do a forward pass:



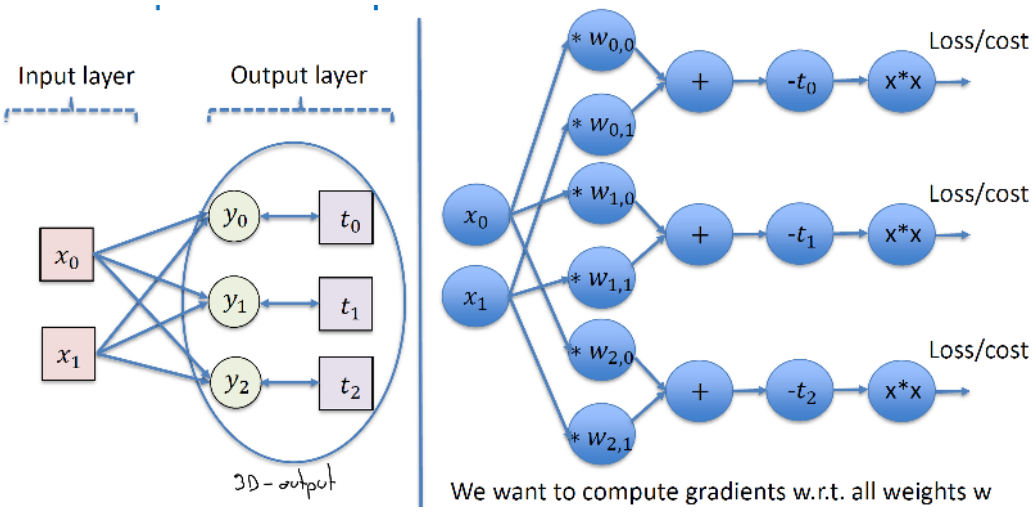
- To compute the gradients we can do a backward pass:



Compute Graphs to Neural Networks - Notation

- x_k input variables
- $w_{l,m,n}$ network weights
  - l denotes the layer
  - m the neuron within the layer
  - n denotes weights in neuron
- y_i computed output
- t_i ground truth targets
- L is loss function

Neural Network as compute graph - Example



- The example uses an L2 loss function. Therefore we subtract t_0 and finally square the output x.

- Formula for L2 loss:

$$L_i = (y_i - t_i)^2, L = \sum_i L_i$$

- To optimize our net, we need to compute gradients w.r.t. to all weights w and biases b:

$$\frac{\partial L}{\partial w_{i,k}} = \frac{\partial L}{\partial y_i} * \frac{\partial y_i}{\partial w_{i,k}}$$

2.4 Optimization in Neural Networks

2.4.1 Vanilla Gradient descent

Calculate gradients for gradient descent

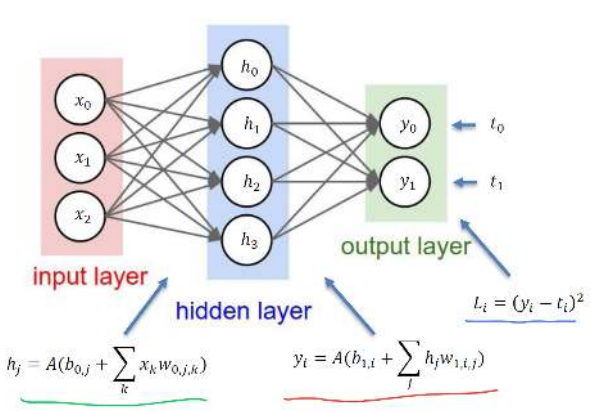
- Given training pair x, t
- We want to update all weights: Calculate derivatives w.r.t to all weights (and biases)
- Gradient:

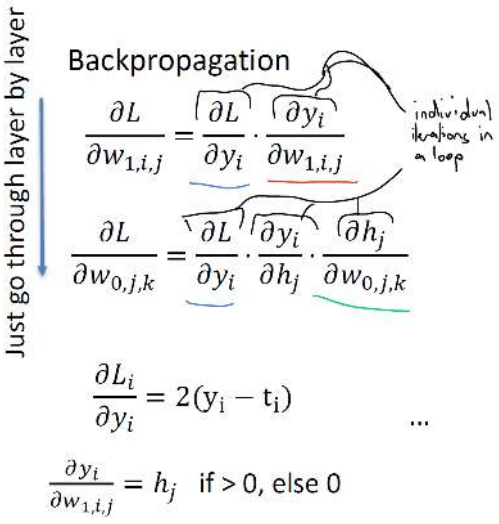
$$\nabla_w f_{x,t}(w) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \dots \\ \dots \\ \frac{\partial f}{\partial w_{l,m,n}} \\ \dots \\ \dots \\ \frac{\partial f}{\partial b_{l,m}} \end{bmatrix}$$

- Gradient step:

$$w' = w - \epsilon \nabla_w f_{x,t}(w)$$

- Example:





  - with $A(x) = max(0, x)$ (ReLu)

  - Amount of unknown weights:
    - Output layer: $2 * 4 + 2 = 10$
      (current-neurons * neurons-prev-layer + biases-current-layer)
    - Hidden layer: $4 * 3 + 4 = 16$

- MAYBE ADD: DERIVATIVES OF CROSS ENTROPY

Convergence of Gradient Descent

- Neural networks are non-convex
- There are different local minima
- No practical way to look for the global optimum (local optima might be good enough)
- Small learning rate yields small steps towards the minimum, big learning rates large steps

Gradient descent for multiple training samples

- Given:
  - Loss function $\boldsymbol{L}$
  - $\boldsymbol{n}$ training samples $\boldsymbol{(x_i, y_i)}$
- Goal: Find best model parameters $\boldsymbol{\theta = (w, b)}$

$$\theta = arg\min L$$

- Cost ($\frac{1}{n}$ can be left out and just means rescaling the learning rate):

$$L = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta, x_i, y_i)$$

- For a gradient step we get:

$$\theta^{k+1} = \theta^k - \alpha\nabla_\theta L\left(\theta^k, x_{(1..n)}, y_{(1..n)}\right)$$

$$\nabla_\theta L\left(\theta^k, x_{(1..n)}, y_{(1..n)}\right) = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta L_i\left(\theta^k, x_i, y_i\right)$$

- We get the gradient $\nabla_\theta L_i\left(\theta^k, x_i, y_i\right)$ from backpropagation

Boundaries of vanilla Gradient Descent

- Gradient is very expensive to compute for large networks
- Assume a dimension $\mathbb{R}^{4096}$ for input x and output z for a neuron
- Dimension of Jacobian $dim(J) = 4096 \times 4096$
- That will lead to 64 MB for one Jacobian and for a mini-batch of 16 we will get $dim(J) = (16 * 4096) \times (16 * 4096) = 4295$ mio. and therefore about 16GB of memory

2.4.2 Stochastic Gradient Descent (SGD)

Difference to Gradient Descent

- Instead of using all $\boldsymbol{n}$ training samples for one update pass, we only use a smaller number of samples (Minibatch) to approximate the gradient
- We choose a subset of the trainset containing $\boldsymbol{m} << \boldsymbol{n}$ samples as a Minibatch
- The size of the Minibatches is a Hyperparameter (Usually power of 2)
- Epoch: complete pass through training set
- For a gradient step, we now only need to calculate gradients for $\boldsymbol{m}$ samples:

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L\left(\theta^k, x_{(1..m)}, y_{(1..m)}\right)$$

$$\nabla_\theta L = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L_i$$

Strategies for learning rate

- Start high and reduce over time
- Reduce every iteration or have fixed training schedule
- There are many decreasing strategies for learning rate

Problems of SGD

- Gradient is scaled equally across all dimension
- Gradient therefore cannot independently scale directions
- min learning rate needs to be conservative to avoid divergence
- Slower than 'necessary'

2.4.3 Gradient Descent with Momentum

- We accumulate Gradients over time and take a velocity $\boldsymbol{v}$ with same dimension as gradient into account
- Formula:

$$v^{k+1} = \beta * v^k + \nabla_\theta L(\theta^k)$$
$$\theta^{k+1} = \theta^k - \alpha * v^{k+1}$$

- α and β (often set to 0.9) are hyperparameters
- Step will be largest when a sequence of gradients all point in the same direction

2.4.4 Nesterov's Momentum

- Look-ahead momentum: Compute gradient 1 step in the future

$$\hat{\theta}^{k+1} = \theta^k + v_k$$
$$v^{k+1} = \beta * v^k + \nabla_\theta L(\hat{\theta}^{k+1})$$
$$\theta^{k+1} = \theta^k - \alpha * v^{k+1}$$

2.4.5 Root Mean Squared Prop (RMSProp)

- Utilizes second momentum by using the variance of gradients
- Divide the learning rate by an exponentially-decaying average of squared gradients

$$s^{k+1} = \beta * s^k + (1 - \beta)[\nabla_\theta L \circ \nabla_\theta L]$$

$$\theta^{k+1} = \theta^k - \alpha * \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \epsilon}$$

- Element-wise multiplication for gradients $\nabla_\theta L$ to calculate s^{k+1}
- Hyperparameters: α (needs tuning), β (usually 0.9) and ϵ (usually 10^{-8})
- RMSProp dampens the oscillations for high-variance directions
- Therefore it's possible to use higher learning rate because it is less likely to diverge

2.4.6 Adaptive Moment Estimation (Adam)

- Combines Momentum and RMSProp
- First momentum: mean of gradients

$$m^{k+1} = \beta_1 * m^k + (1 - \beta_1)\nabla_\theta L(\theta^k)$$

- Second momentum: variance of gradients

$$v^{k+1} = \beta_2 * v^k + (1 - \beta_2)[\nabla_\theta L(\theta^k) \circ \nabla_\theta L(\theta^k)]$$

- Weight update:

$$\theta^{k+1} = \theta^k - \alpha * \frac{m^{k+1}}{\sqrt{v^{k+1}} + \epsilon}$$

- m^{k+1} and v^{k+1} are initialized with 0
- Typically, bias-corrected moment updates:

$$\hat{m}^{k+1} = \frac{m^k}{1 - \beta_1}$$

$$\hat{v}^{k+1} = \frac{v^k}{1 - \beta_2}$$

- This leads to following weight updates:

$$\theta^{k+1} = \theta^k - \alpha * \frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \epsilon}$$

2.4.7 Different strategies

Shape of derivatives

- Derivative: $f : \mathbb{R} \mapsto \mathbb{R}$ $\quad \frac{df(x)}{dx}$
- Gradient: $f : \mathbb{R}^m \mapsto \mathbb{R}$ $\quad \nabla_x f(x)$
- Jacobian: $f : \mathbb{R}^m \mapsto \mathbb{R}^n$ $\quad \boldsymbol{J} \in \mathbb{R}^{n \times m}$
- Hessian: $f : \mathbb{R}^m \mapsto \mathbb{R}$ $\quad \boldsymbol{H} \in \mathbb{R}^{m \times m}$ (Second derivative)

Newton's method

- Exploits the curvature to take a more direct route to the minimum
- Approximate our function by a second-order Taylor series expansion
- Differentiate and equate to zero
- Problems:
  - Number of parameters of Hessian is k^2 for k network parameters
  - Computational complexity of inversion of Hessian $\mathcal{O}(k^3)$

BFGS and L-BFGS

- Belongs to family of quasi-Newton methods
- Uses an approximation of the inverse of the Hessian
- Limited complexity ($\mathcal{O}(n^2)$)
- Limited memory for L-BFGS ($\mathcal{O}(n)$)

Gauss-Newton

- Approximate the Hessian with the Jacobian $H_f \approx 2J_F^T J_F$

Levenberg

- "damped" version of Gauss-Newton
- additional damping factor λ that is adjusted each iteration
- "interpolation" between Gauss-Newton (small λ) and Gradient Descent (large λ)

Levenberg-Marquardt

- Instead of a plain Gradient Descent for large λ scale each component of the gradient according to the curvature
- Avoids slow convergence in components with a small gradient

2.4.8 Conclusion

Which method to use for Neural Networks?

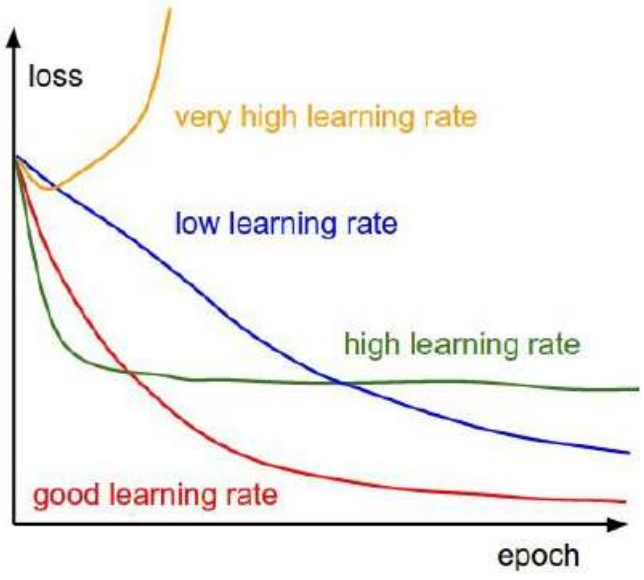
- Standard: Adam
- Fallback option: SGD with momentum
- Newton, L-BFGS, GN and LM only if you can do full batch updates, because it doesn't work well for minibatches

General approach

- Linear Systems:
  - LU, QR, Cholesky, Jacobi, Gauss-Seidel, CG, PCG, etc.
  - GD or SGD not for linear systems, as Gradient is only a constant!
- Non-linear Systems (gradient-based):
  - second order: Newton, Gauss-Newton, LM, (L-)BFGS (Faster, when applicable)
  - first order: Gradient Descent SGD (Work well for minibatches)
- Others
  - Genetic algorithms, MCMC, Metropolis-Hastings, ...
  - Constrained and convex solvers

2.5 Learning Rate, Training and Learning

2.5.1 Importance of Learning Rate



2.5.2 Learning Rate Decay

- We need a high learning rate when far away from minimum
- We need a low learning rate when close

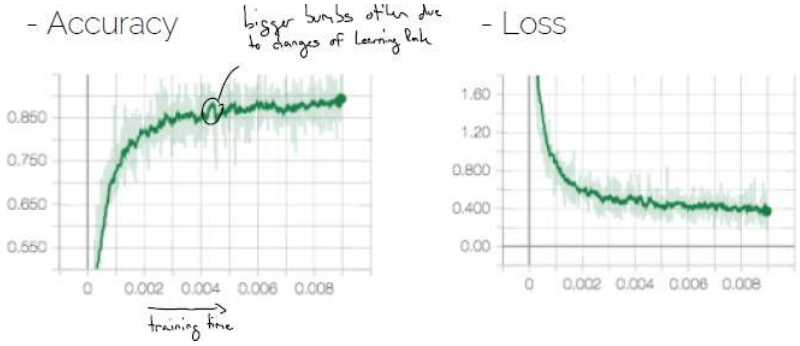
- Possible variants:
  - $\alpha = \frac{1}{1+decayrate*epoch} * a_0$ (a_0 initial LR)
  - Step decay (with rate t): $\alpha = \alpha - t * \alpha$
  - Exponential decay $\alpha = t^{epoch} * \alpha_0$
  - $\alpha = \frac{t}{\sqrt{epoch}} * \alpha_0$
- It's also possible to manually set the learning rate every n-epochs

2.5.3 Training, Learning and Datasets

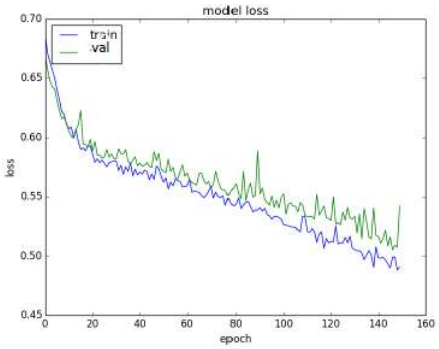
Training Given ground dataset with ground labels

- x_i, y_i as training pair
- Take network f and its parameters w, b
- Use SGD and variations to find optimal parameters w, b

Training graph:



Validation graph:

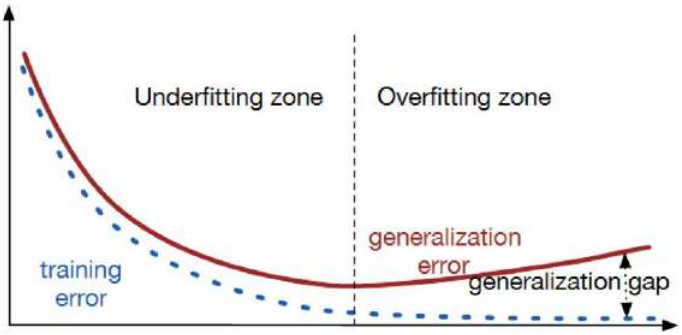


Learning

- Generalization to unknown dataset
- Assumption: Optimized parameters will give similar results on different data (i.e. test data)

Over- and Underfitting

- Overfitting: function resembles training data to closely and doesn't generalize well (Variance)
- Underfitting: function doesn't describe the data well (Bias)



Datasets

- Training set: Used for training $(60 - 80\%)$
- Validiation set: for hyperparameter optimization and checking of generalization $(10 - 20\%)$
- Test set: Only for the final test at the very end $(10 - 20\%)$

2.5.4 Hyperparameters and Hyperparameter tuning

Hyperparameters

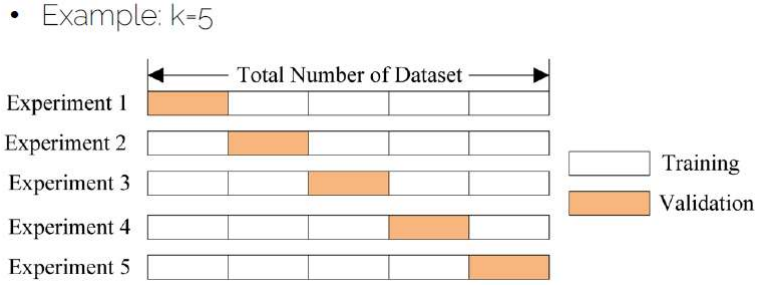
- Network architecture
- Number of iterations
- Learning rate(s)
- Regularization
- Batch size
- ...

Hyperparameter Tuning

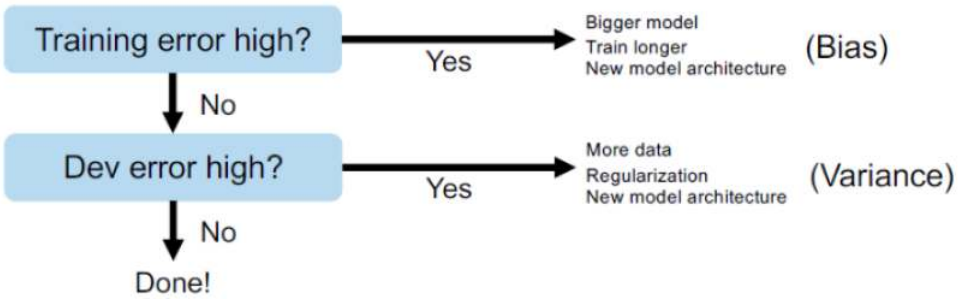
- Manual search: most common
- Grid search:Define ranges for all parameter spaces and select points. Iterate over all possible configurations
- Random search: Pick points at random in the predefined ranges

Cross Validation

- Used when data set is extremely small and/or method of choice has low training times
- Partition data into k subsets, train on k-1 and evaluate performance on the remaining subset



2.5.5 Basic recipe for machine learning



2.6 Loss, output and activation functions

2.6.1 Going deep in Neural Networks

Approach

- Start simple: Overfit to a single and then to several training samples
- Always try a small architecture first to verify that you are learning something
- Estimate timings (how long takes one epoch?)
- Try to optimize data loading

Problem of going deeper

- Vanishing gradients: Multiplication of already small gradients will eventually lead to very small or 0 gradients
- Impact of small decisions (architecture, activation functions, ...)

2.6.2 Output Functions

Sigmoid

- For binary predictions
- Formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Result can be interpreted as a probability

Softmax

- For predicting multiple classes
- Formula:

$$p(y_i|x, \theta) = \frac{e^{x\theta_i}}{\sum_{k=1}^{n} e^{x\theta_k}}$$

2.6.3 Loss Functions

L1 loss

- Sum of absolute differences
- Costly to compute
- Optimum is the median
- Formula:

$$L^1 = \sum_{i=1}^{n} |y_i - f(x_i)|$$

L2 loss

- Sum of squared differences
- Compute efficient
- Prune to outliers
- Optimum is the mean
- Formula:

$$L^2 = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Cross-Entropy loss (Softmax loss)

- Formula:

$$L_i = -\ln\left(\frac{e^{s_{y_i}}}{\sum_k e^{s_k}}\right)$$

- score function $s = f(x_i, W)$, e.g. $f(x_i, W) = W * [x_o, x_1, ..., x_N]^T$
- Calculate final loss:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

- Softmax Loss always wants to improve no matter how small the loss is

Hinge Loss (SVM Loss)

- Formula:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

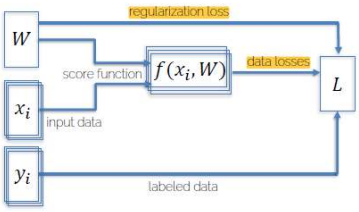
- score function s like for Softmax loss
- Calculate final loss:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

- Hinge Loss saturates. Small losses become 0.

Loss, Compute graphs and regularization

- The final loss is calculated based on data loss and regularization loss
- Regularization loss:
  - L^1-reg: $R^1(W) = \sum_{i=1}^{N} |w_i|$
  - L^2-reg: $R^2(W) = \sum_{i=1}^{N} w_i^2$
- Full loss: $L = \frac{1}{N} \sum_{i=1}^{N} L_i + R^{1|2}(W)$
- Usually, L^2 reg is used for weight regularization
- Loss in a compute graph:



2.6.4 Activation Functions

Sigmoid

- Sigmoid can be used as an activation function (Formula: see above)
- Problems:
  - Vanishing gradients: Saturated neurons kill the gradient flow
  - This can be overcome by good initialization which is very hard for big networks
  - Output is always positive
- Problem of positive output:
  - Shrinks down the gradient space
  - We can't go in any direction that would lead to a minimum faster
    $\rightarrow$ It is not zero-centered

tanh

- Simply calculate with tanh function: $\sigma(x) = \tanh(x)$
- Zero-centered
- Problem: Still saturates
- Used for recurrent neural networks

Rectified Linear Units (ReLU)

- $\sigma(x) = \max(0, x)$
- Fast convergence
- Does not saturate

- Large and consistent gradients
- Problem: Neurons with negative input don't contribute → die out over time
- By initializing ReLU neurons with slightly positive biases (0.1) makes it likely that they stay active for most inputs
- Standard choice for neural networks

Leaky ReLU

- $\sigma(x) = \max(0.01x, x)$
- Does not die

Parametric ReLU

- $\sigma(x) = \max(\alpha x, x)$
- α additional parameter to backprop
- Does not die

Maxout units

- Pick maximal output:
$$Maxout = \max(w_1^T x + b_1, w_2^T x + b_2)$$
- Generalization of ReLUs
- Does not die, does not saturate
- Problem: Increases the number of parameters

Data pre-processing

For images, subtract the mean image (AlexNet) or per-channel mean (VGG-Net)

2.7 Weight initialization

All weights to zero

- Set all weights to 0
- Problems:
  - Hidden units are all going to compute the same function, gradients are all going to be the same
    → All layers predict the same features

Small random numbers

- Gaussian with zero mean and standard deviation 0.01
- Problem: Gradients vanish

Big random numbers

- Gaussian with zero mean and standard deviation 0.01
- Problem: Saturates fast

Xavier initialization

- Ensure that input variance is equal to output variance
- Calculate Variance of score function:

$$Var(s) = ... = (nVar(w))Var(x)$$

- $Var(x)$ should be 1:

$$Var(w) = \frac{1}{n}$$

- Problem with ReLU: Kills half of the data
- We can mitigate that effect with Xavier/2 initialization:

$$Var(w) = \frac{2}{n}$$

- But: Doesn't scale for very high number of layers, as output layers are killed at some point

Batchnorm

- Goal: Activations should not die out
- Solution: Unit Gaussian activations
- Batchnorm can be included in Neural networks as separate layer:
  - Insert after fully connected layer
  - but before non-linear activation
- Approach:
  1. Normalize mean and variance of the inputs to 0 and 1, respectively:

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

  2. Allow the network to change the range

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

- The network can learn to undo the normalization:

$$\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$$

$$\beta^{(k)} = E[x^{(k)}]$$

- It's empirically shown that treating the dimensions separately still leads to faster convergence
- Biases before BN layers can all be set to 0 as they will be canceled out by BN
- Problem for test time: We might only process one input, so we can't compute a meaningful mean and variance
  → Compute a mean (μ_{test}) and variance (σ^2_{test}) by running an exponentially weighted average across training mini-batches
- Benefits of BN:
  - Makes very deep nets easier to train due to more stable gradients
  - A much larger range of hyperparameters works similarly
- Drawback: Doesn't work well for small batch sizes (about < 16)

2.8 Regularization

- Any strategy that aims to:
  - Lower validation error
  - Increase training error

2.8.1 Weight decay

- L^2 regularization:
$$\theta_{k+1} = \theta_k - \epsilon \nabla_\theta L(\theta_k, x^i, y^i) - \lambda \theta_k^T \theta_k$$
- Note: This is directly subtracted during the update step (unlike L^2 regularization loss)
- Penalizes large weights and improves generalization

2.8.2 Data augmentation

- A classifier has to be invariant to a wide variety of transformations
- We can help the classifier by generating fake data simulating plausible transformations
- Examples for images:
  - Flip augmentation: Flip images
  - Crop augmentation: Only use sections of images
  - Random brightness and contrast changes
- Data augmentation should be considered as a part of the network design
  → Use same augmentation when comparing two networks

2.8.3 Early stopping

- Take model with lowest validation error (before overfitting sets in)
  → Optimize training time as a hyperparameter

2.8.4 Bagging and ensemble methods

Ensemble

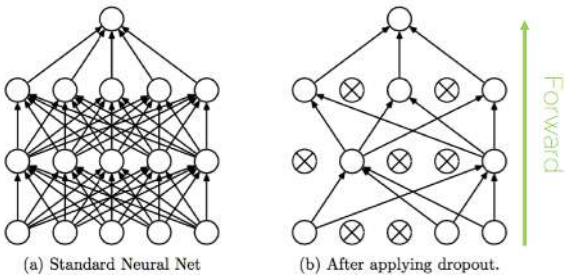
- Train three models and average their results
- Change a different algorithm of change the objective function
- If errors are uncorrelated, the expected combined error will decrease linearly with the ensemble size

Bagging

- Use k different datasets to train the different models

2.8.5 Dropout

- Disable a random set of neurons (typically 50%):



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

- Intuition: Using half the network = half capacity
  - Disable redundant representations
  - Base scores on more features
  - "Two models in one" → can be considered as model ensemble
- Dropout is not used during test time
  - Different conditions at train and test time
  - We need to scale the weights during training accordingly: Weight scaling inference rule

$$\frac{1}{p}(Wx + b)$$

- Summary:
  - Efficient bagging method with parameter sharping
  - Use dropout!
  - Dropout reduces the effective capacity of a model
    → larger models, more training time

2.9 Transfer Learning

- Use what have been learned on a task P1 for another setting P2
  → Train on large dataset, apply on smaller dataset
- Intuition: Convolutional layers extract features and final decision layers (e.g. FC for classification) can use those to make decisions → only need to train the possibly adapted decision layer (Finetuning)
- Allows usage of pretrained models, e.g. ImageNet for images
- When Transfer learning makes sense:
  - When tasks have the same input (e.g. an RGB image)
  - When you have more data for task P1 than for task P2
  - When low-level features of task P1 could be useful to learn P2

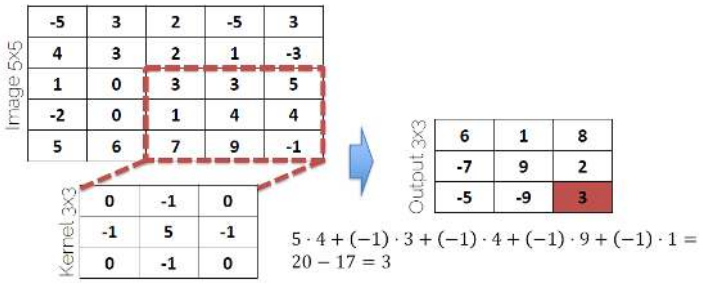
3 Convolutional Neural Network (CNN)

3.1 Processing images with FC layers

- Small images already result in a lot of parameters: e.g. $5 \times 5 \times 3$ image that is processed by a layer with 3 neurons
  $\rightarrow 5 * 5 * 3 = 75$ weights per neuron, $75 * 3 = 225$ weights for the layer
- Larger images impractical: An image with dimension $1000 * 1000 * 3$ and a layer with 1000 neurons would already have 3 billion weights
- Approach:
  - We need a layer with structure
  - Weight sharing
    $\rightarrow$ using same weights for different parts of the image

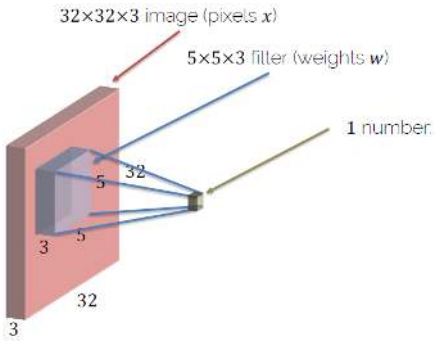
3.2 Convolutions

- Convolutions vs. Fully-Connected:
  - Convolutions restrict the degrees of freedom
    - FC is somewhat brute force
    - Convolutions are structured
  - Sliding window with the same filter parameters to extract image features
    - Concept of weight sharing
    - Extract same features independent of location
- Types of convolutions:
  - Valid convolution: Using no padding
  - Same convolution: output size equal to input size (either with padding or 1×1 convolution)
- Idea: Apply a filter (called kernel) to a function
- Slide the filter kernel through the image
- Example:



Convolutions on RGB Images

- Depth dimension (channels) of input image ($width \times height \times depth$) and Kernel must match: $32 \times 32 \times \mathbf{3}$ input image $\rightarrow 5 \times 5 \times \mathbf{3}$ filter



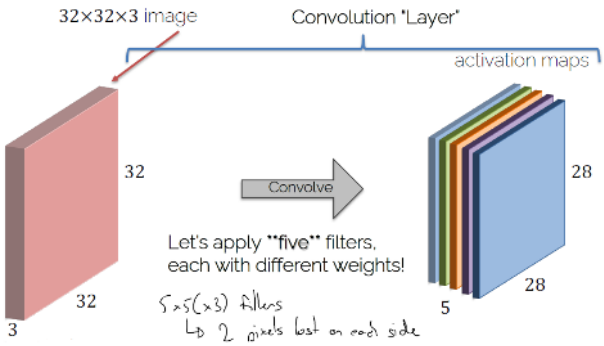
- Can be simply calculated with a dot product (with x_i i-th chunk of the image):

$$z_i = w^T x_i + b$$

- For the above example, we would have $5 \times 5 \times 3 = 75$ weights and input values and therefore 75 operations (+ operations for the bias)
- The result of applying one filter is an activation map (also: feature map) with depth 1

3.3 Convolutional Layer

- Instead of applying only one filter, we can apply multiple Filters
- A basic layer is defined by:
  - Filter width and height (depth is implicitly given by the input image)
  - Number of different filter banks (#weight sets)
  - Each filter captures a different image characteristic



Dimensions of a Convolutional Layer

- Parameters:
  - Input: $W_{in} \times H_{in} \times D_{in}$
  - Number of Filters: K
  - Filter: $F \times F$

  - Stride: S
  - Padding P

- Output Dimension:

$$\left(\frac{W_{in} + 2 * P - F}{S} + 1\right) \times \left(\frac{H_{in} + 2 * P - F}{S} + 1\right) \times K$$

  $\rightarrow$ Depth of the output when applying one filter is always 1 (in general: equal to the number of filters)

- Amount of weights per Filter:

$$F \times F \times D_{in}$$

- Total amount of weights:
  $(F \times F \times D_{in}) * K$ weights + K biases

Padding

- Reasons for padding:
  - Sizes get small too quickly
  - Corner pixels are only used once
- Padding size:

$$P = \frac{F - 1}{2}$$

- Usually, zero padding is used

Receptive field

- Spatial extent of the connectivity of a convolutional filter
- Synonymous to filter/kernel size
- Each output pixel is connected to following amount of input pixels:

$$F \times F \times D_i n$$

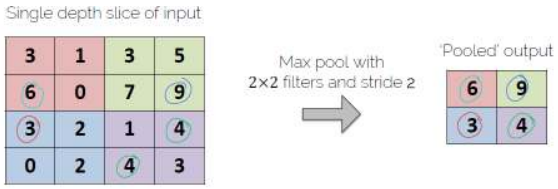
- Equal to the weights of each Filter!

1×1 convolutions

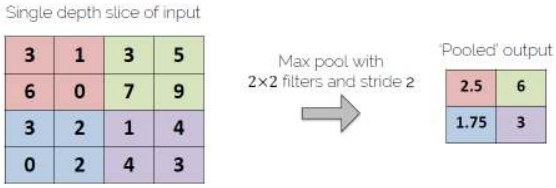
- Kernel size of 1×1
- Output size: Equal to input size (channels adapted according to number of filters)
- The input is just scaled with a number
- Usage:
  - Use it to shrink the number of channels
  - Further adds a non-linearity
    $\rightarrow$ allows to learn more complex functions

3.4 Pooling

- Used to downsample images
- Types:
  - Max Pooling: Apply max operation on specified kernel size



  - Average Pooling: Apply average operation on specified kernel size



- Convolution vs. Pooling:
  - Conv Layer: Feature extraction
    → computes a feature in a given region
  - Pooling Layer: Feature selection
    → Pick strongest activation in a region

Dimensions of a Pooling Layer

- Parameters:
  - Input: $W_{in} \times H_{in} \times D_{in}$
  - Filter: $F \times F$
  - Stride: S
- Output Dimension:

$$\left(\frac{W_{in} - F}{S} + 1\right) \times \left(\frac{H_{in} - F}{S} + 1\right) \times D_{in}$$

- Pooling is a fixed function and doesn't contain parameters

3.5 Fully convolutional network

- Fully connected layers are converted to convolutional layers
- A final upsampling layer resizes the output image to the required size
- E.g. used for semantic segmentation: pixelwise prediction of images

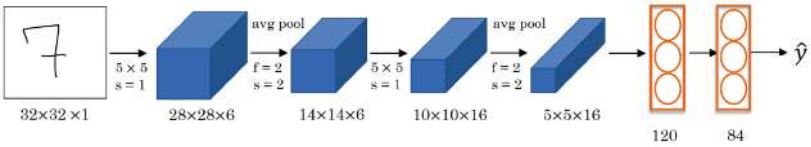
Upsampling

- Interpolation
  - Nearest neighbor interpolation
  - bilinear interpolation
  - bicubic interpolation
- Transposed conv
  - Unpooling
  - Convolution filter (learned)
  - Also called: up-convolution
- Autoencoders are a cascade of unpooling + conv operations

3.6 Architectures

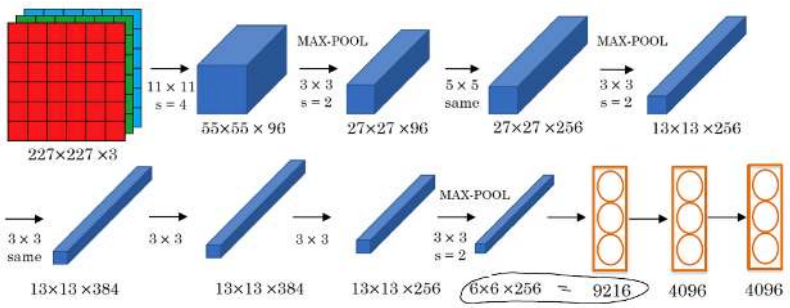
3.6.1 LeNet

- Digit recognition: 10 classes
- Uses average pool (today: usually max pool) and tanh/sigmoid (today: ReLu)
- General approach: Width and height decrease with the number of layers, number of filters increases
- 60k parameters



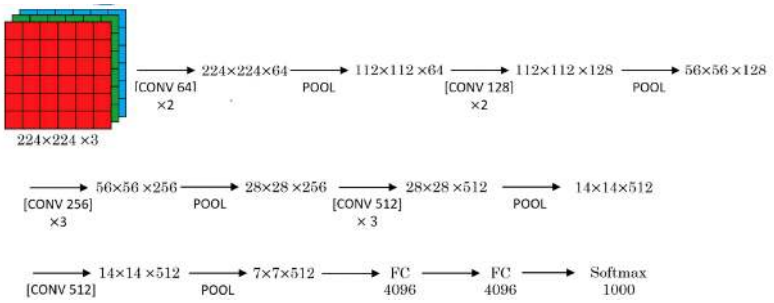
3.6.2 AlexNet

- First filter with stride 4 to reduce size significantly
- Higher number of filters
- Uses ReLU instead of tanh/sigmoid, max pool instead of average pool
- 60 million parameters



3.6.3 VGGNet

- Strives for simplicity
- 138 million parameters



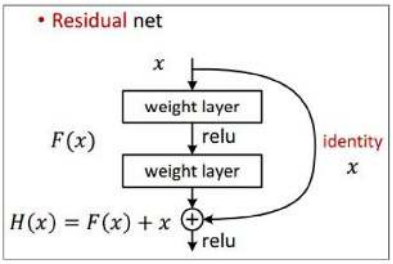
3.6.4 ResNet

Problem of Depth

- Training becomes harder with more and more layers
- Vanishing and exploding gradients

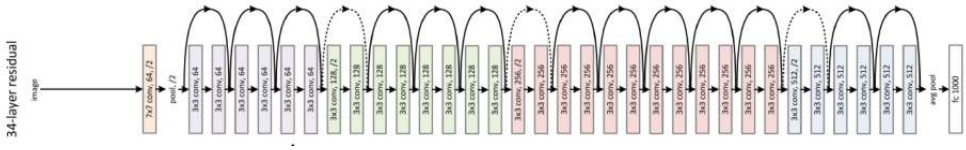
Residual block

- Introduces skip connection ("alternative path")
- Skip one or several layers during training
- Example:



Properties of ResNet

- Xavier/2 initialization
- SGD - Momentum
- Learning rate 0.1, divided by 10 when plateau
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout



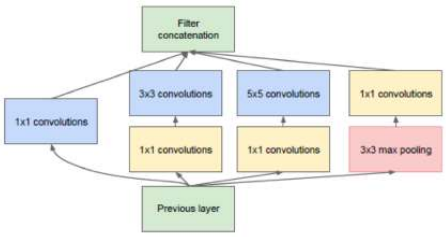
Why do ResNets work?

- Same values are kept and a non-lineartiy is added
  → Identity function
- The identity is easy to learn for the residual block
- Guaranteed to not hurt the performance
  → can only improve performance

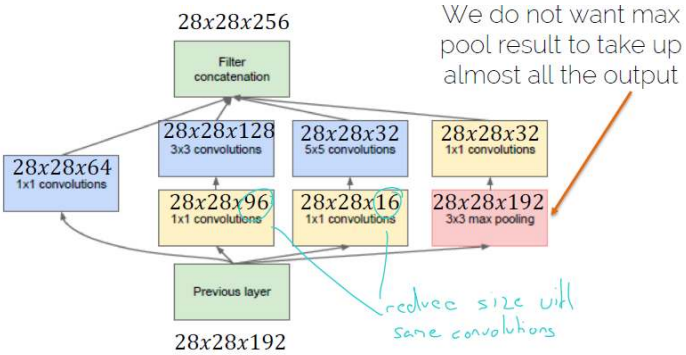
3.6.5 GoogLeNet

Inception layer

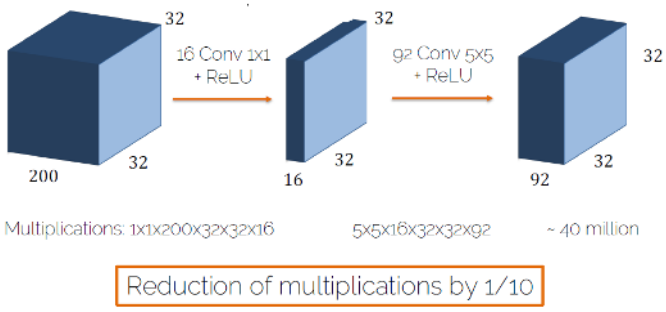
- Combines usage of several filters
- max pooling with stride 1



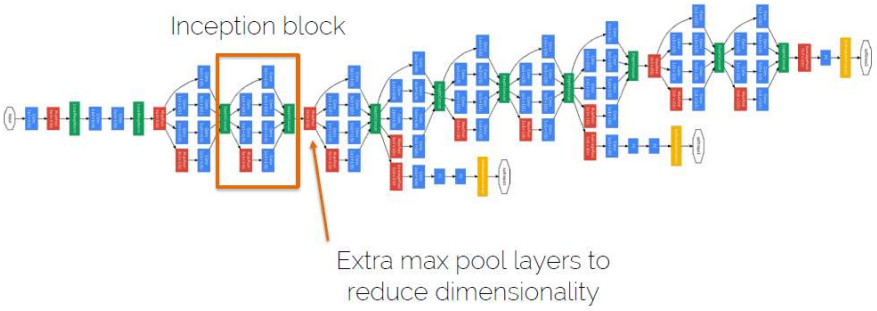
- They output includes the concatenation of all the individual filters
- Example:



- Computational cost:
  - Reduce number of multiplications by adding same convolution layer in the middle:



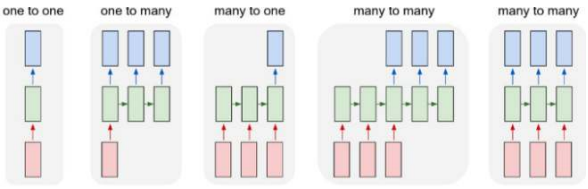
Architecture



4 Recurrent Neural Network (RNN)

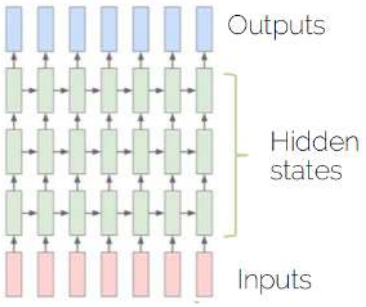
4.1 Context and Structure

- Allows to predict time series
- Different shapes of RNNs:



  - one to one: Classic Neural Networks for image classification (no RNN)
  - one to many: image captioning
  - many to one: Language recognition
  - many to many: Machine translation
  - many to many: Event classification

- Structure of a Multi-layer RNN:



4.2 Basic RNN

Output calculation

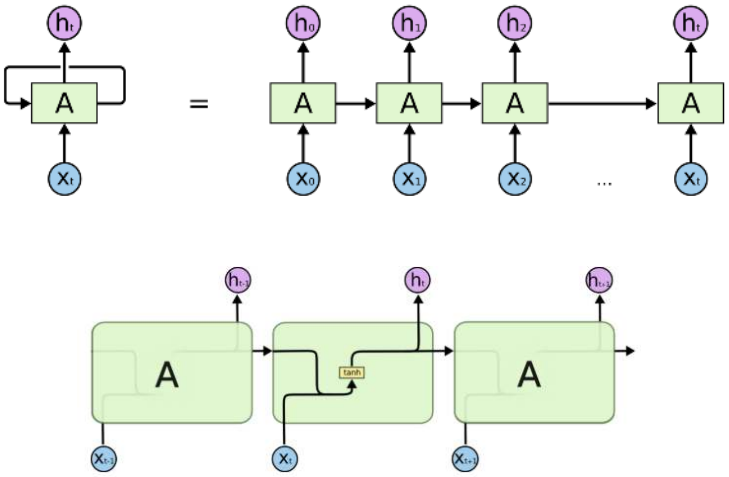
- New hidden state h_t is updated based on the previous hidden state h_{t-1} and the input x_t:

$$h_t = \tanh\left(\theta_c h_{t-1} + \theta_x x_t + b\right)$$

- Output is calculated based on the hidden state:

$$y = \theta_y h_t$$

- Weights θ_c, θ_x and θ_y are parameters to be learned
  → Same for each time step (generalization)
- Example of a Network and it's "internals":



  → Hidden states A are the same

Problems of RNNs

- With RNNs, we also want to model long-term dependencies
- If we just look at the step of updating the hidden state:

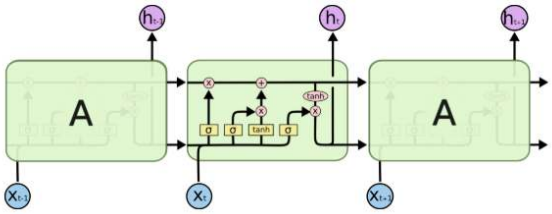
$$h_t = \theta^t h_0$$

- Same weights are multiplied over and over again
  → Vanishing or exploding gradients
  - Vanishing gradients: If magnitude of eigenvalues of θ is smaller than one
    → Can be (partially) solved with LSTMs
  - Exploding gradients: If magnitude of eigenvalues of θ is bigger than one
    → Can be solved with gradient clipping
- Gradients will also vanish due to tanh, because it's derivative outputs a value smaller or equal to 1

4.3 Long Short Term Memory

Structure of LSTM

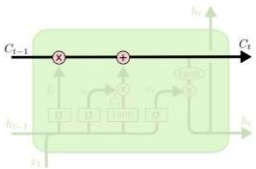
- Repeating module of LSTMs contains four interacting layers

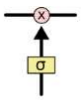


- Legend:

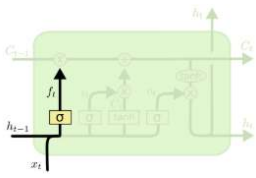
Individual components of LSTM repeating modules

- Basic building blocks:
  - Cell state that transports information and gates to add or remove information to/from the cell state:



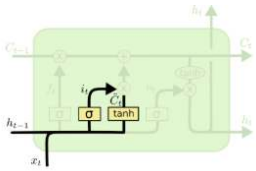


  - Gates use sigmoid function to determine the output: zero means "let nothing through (forget)", 1 means "let everything through (keep)"
- Individual steps:
  - Forget gate f_t: Decide what information to throw away from the cell state



$$f_t = \sigma(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)$$

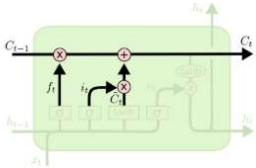
- Input gate i_t: Decides which values will be updated and cell update $\tilde{C}$:



$$i_t = \sigma(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i)$$

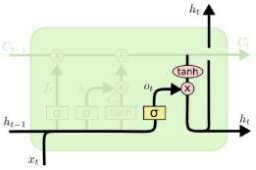
$$\tilde{C} = \tanh\left(\theta_{xg} x_t + \theta_{hg} h_{t-1} + b_g\right)$$

- Update cell state C_{t-1} to new cell state C_t:



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

- Output gate o_t: Decides which values to output and final output calculation



$$o_t = \sigma\left(\theta_{xo} x_t + \theta_{ho} h_{t-1} + b_o\right)$$

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

- Further information:
  - All weight parameters θ are learned through backpropagation
  - Dimensions of input and hidden state need same dimension
  - How do LSTMs solve vanishing gradients?
    - they create a connection between the forget gate activations and the gradients computation
    - this connection creates a path for information flow through the forget gate for information the LSTM should not forget