Summary of the Deep Learning Lecture at the University of Bonn

Tim Nogga

July 24, 2024

Contents

1 Introduction

1.1 Task T

Machine Learning Tasks describes how a machine learning system learns from data. Such data can be images(pixels) among many. Typical Tasks include

Klassification, Regression, and Density Function Estimation.

1.2 Experience E

Specifies the Information an Algorithm can use during the Learning Process.

1.2.1 Supervised Learning

The Algorithm is given a dataset with the correct answers. The Algorithm then tries to learn the mapping from the input to the output. This can be described with an Estimation $p_data(y|x)$.

1.2.2 Unsupervised Learning

The Algorithm is given a dataset without the correct answers. The Algorithm then tries to learn the underlying structure of the data. This can be described with an Estimation $p_data(x)$.

1.3 Performance Measure P

Learning Algorithms can be evaluated based on their Performance. In Classification Problems, the Accuracy is a common measure. The Perfomance Measure is calculated on a Test Set, which is not used during the Training Process. So there a are two sets a Test Set and a Training Set.

1.4 Training Error, Test Error and Generalization

Generalization is the ability of an algorithm to perform well on previously unseen data. During Training the Training Error should be reduced, but the Test Error should be reduced as well. If the Training Error is reduced but the Test Error is not, the algorithm is overfitting. Meaning the algorithm is learning the noise in the data instead of the underlying structure. But if the Test Error is not even reduced the algorithm is underfitting.

1.5 Capacity

Overfitting and underfitting are also dependent on the capacity of the model. Say you take a model with with a low capacity, it will underfit the data, as it is not able to learn the underlying structure. But if a model with high Capacity is used, it will just memorize the data and thus overfit, as it is not able to generalize.

1.6 Capacity vs Error

It becomes apparent that the Training Error decreases with increasing Capacity. But the Test Error decreases first and then increases again. This is due to the fact that the model is overfitting the data, thus not able to generalize anymore. This is also why it is called Generalization Gap.

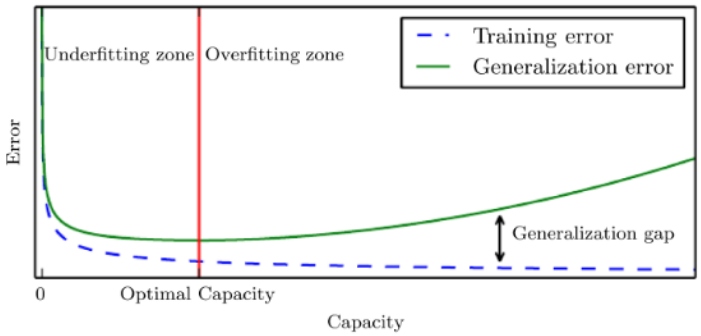


Figure 1:

1.7 Hyperparameters

Leraning Algorithms usually have Hyperparamters which are paramters not learned during Training, but set before. These could be things such as Model Capacity and Learning Rate. This poses the Question wether Capacity Hyperparameters can be derived from the Trainingset. The answer is no, since the Learning Algorithm would always choose the highest Capacity, as it would always reduce the Training Error.

1.8 Cross Validation

Cross Validation can be used to determine the best Hyperparameters. The Training Set is split into k parts. Then the Algorithm is trained on k-1 parts and tested on the remaining part. This is done k times, so that every part is used as a Test Set once. The average Test Error is then used to determine the best Hyperparameters.

2 Recap Statistics

2.1 Terms and Definitions

- **Sample Space** Ω: Set of all possible outcomes of an experiment. For example a dice roll has a Sample Space of $\Omega = \{1, 2, 3, 4, 5, 6\}$
- **Event** A: Subset of the Sample Space. For example the Event of rolling an even number is $A = \{2, 4, 6\}$
- **Sigma Algebra** Collection of Events. For example the Sigma Algebra of the dice roll is $\sigma(\Omega) = \{\emptyset, \{1, 2, 3, 4, 5, 6\}, \{1, 3, 5\}, \{2, 4, 6\}\}$
- **Measurable space** $(\Omega, \mathcal{A})$: Is a Measurable Space if Ω is a Sample Space and $\mathcal{A}$ is a Sigma Algebra.
- **Probability Density Function if Ω is Discrete** x $p : \Omega \to [0, 1]$ with $\sum_{\omega \in \Omega} p(\omega) = 1$.
- **Probability Density Function if Ω is Continuous** x $p : \Omega \to [0, \infty)$ with $\int_{\omega \in \Omega} p(\omega) d\omega = 1$.

2.2 Multivariate Statistics

It just means that there are more than one Random Variable. For example if we have two Random Variables X and Y we can define a Joint Probability Density Function $p(x, y)$.

2.3 Marginalization in a Discrete Case

We can get the PDF of one Random Variable by summing over the other. For example $p(x) = \sum_y p(x, y)$. or $p(y) = \sum_x p(x, y)$. In the Case of getting $p(x)$ this is just Summing over all possible values of y, which is rather Intuitive.

2.4 Marginalization in a Continuous Case

In the Continuous Case we have to integrate over the other Random Variable. For example $p(x) = \int p(x, y)dy$. or $p(y) = \int p(x, y)dx$.

2.5 Marginalization in a Mixed Case

In a Mixed case if $p(x)$ is Continuous and $p(y)$ is Discrete we have to sum over y and integrate over x. For example $p(x) = \int p(x, y)dy$ or $p(y) = \sum_x p(x, y)$.

2.6 Marginalization in Multiple Dimensions

If we for example want to know (p(x,y) given 4 parameters x,y,z,w) we have to marginalize over z and w. If w is Discrete we sum over w and if z is Continuous we integrate over z. This results in the following $p(x, y) = \sum_w \int p(x, y, z, w)dzdw$.

2.7 Conditional Probability

The Conditional Probability is defined as $p(x|y = y_1)$ and is the Probability of x given that y is y_1. $p(x|y = y^*) = \frac{p(x,y=y^*)}{p(y=y^*)}$ This is rather intuitiv aswell if you imagine that you are only interested in this one case where y is y^*. And then you will need to devide by the Probability of this case happening, independent of x.

This can be rewritten as $p(x|y) = \frac{p(x,y)}{p(y)}$.
Which in turn can be rewritten as $p(x, y) = p(x|y)p(y)$ and $p(x, y) = p(y|x)p(x)$.

2.8 Independence

Two Random Variables are independent if $p(x, y) = p(x)p(y)$. This means that the Probability of x and y happening is the same as the Probability of x happening times the Probability of y happening.

2.9 independent and Identically Distributed

As an example one could think of a Dice Roll. The Dice Roll is independent and identically distributed, as the Probability of rolling a 1 is the same for every roll and the Probability of rolling a 1 and a 2 is the same as the Probability of rolling a 1 times the Probability of rolling a 2. Formally this can be written $p(x_1, x_2, ..., x_n) = p(x_1)p(x_2)...p(x_n)$.

2.10 Bayes Rule

As previously discussed we have $p(x,y) = p(x|y)p(y) and p(x,y) = p(y|x)p(x)$ By setting these two equations equal we get $p(y|x)p(x) = p(x|y)p(y)$ This can be rewritten as $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$ which is the Bayes Rule. With the background knowledge that $p(x) = \sum_y p(x|y)p(y)$ for discrete and $p(x) = \int p(x|y)p(y)$ for Continuous cases we can rewrite the Bayes Rule as $p(y|x) = \frac{p(x|y)p(y)}{\sum_y p(x|y)p(y)}$ or $p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)dy}$. Here the Likelihood is the $p(x|y)$, the Prior is the $p(y)$ and the Posterior is the $p(y|x)$ and the Evidence is the $p(x)$. The Likelihood is the the Probability to observe a certain x given a certain y. The Prior is the Probability of y happening. The Posterior is the Probability of y happening given x. The Evidence is the Probability of x happening.

2.11 Expectation

The Expectation is the average value of a Random Variable. It is defined as $E[x] = \sum_x xp(x)$ for Discrete Random Variables and $E[x] = \int xp(x)dx$ for Continuous Random Variables.

2.12 Variance and Standard Deviation

The Variance is a measure of how much the values of a Random Variable differ from the mean. It is defined as $Var[x] = E[(x - E[x])^2] = E[x^2] - E[x]^2$. The Standard Deviation is the square root of the Variance. It is defined as $\sigma = \sqrt{Var[x]}$.

2.13 Shannon Entropy

The Shannon Entropy gives us an answer to the Question how many bits are needed to encode Samples from a Probability Distribution. It is defined as $H(x) = -\sum_x p(x) \log p(x) = E[-\log(p(X))]$ for Discrete Random Variables. This can be used to measure the Uncertainty of a Random Variable. If the Entropy is high, the Random Variable is uncertain. If the Entropy is low, the Random Variable is more certain. So for a Coin Flip the Entropy would be $H(x) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{2}\log\frac{1}{2} = 1$. This means that it takes 1 Bit to encode the Coin Flip.

2.14 Cross Entropy

The Cross Entropy is the same as the Shannon Entropy, but for two Probability Distributions. It is defined as $H(p,q) = -\sum_x p(x) \log q(x) = E_p[-\log(q(X))]$ for Discrete Random Variables. This can be used to measure the difference between two Probability Distributions. If the Cross Entropy is high, the Distributions are different. If the Cross Entropy is low, the Distributions are similar.

2.15 Kullback-Leibler Divergence

The Kullback-Leibler Divergence is a measure of how much one Probability Distribution differs from another. It is defined as $D_{KL}(p||q) = H(p,q) - H(p) =$

$\sum_x p(x) \log \frac{p(x)}{q(x)} = E_p[\log \frac{p(X)}{q(X)}]$ for Discrete Random Variables. This can be used to measure the difference between two Probability Distributions. If the Kullback-Leibler Divergence is high, the Distributions are different. If the Kullback-Leibler Divergence is low, the Distributions are similar, if the Divergence is 0 the Distributions are the same.

3 Estimators

3.1 Point Estimation

A Point Estimator is a function of the data that is used to estimate an unknown parameter $\hat{\theta}$ This can also be interpreted as an Learning Algorithm, where the Parameter is supposed to be learned. The Point Estimator is denoted as $\hat{\theta}$.

3.2 Bias

The Bias of an Estimator is the difference between the expected value of the Estimator and the true value of the parameter. It is defined as $Bias(\hat{\theta}) = E[\hat{\theta}] - \theta$. If the Bias is 0, the Estimator is unbiased.

3.3 Mean Squared Error

The Mean Squared Error is a measure of how well an Estimator performs. This was also discussed on one of the Tasks handed out. Here we had to Prove that the Mean Squared Error can be decomposed into the Variance of the Estimator and the Bias of the Estimator.

$$\begin{aligned}
&\text{MSE}(\hat{\theta}_m) = \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \\
\Leftrightarrow\ &\mathbb{E}[(\hat{\theta}_m - \theta)^2] = (\mathbb{E}[\hat{\theta}_m] - \theta)^2 + \mathbb{E}[\hat{\theta}_m^2] - \mathbb{E}[\hat{\theta}_m]^2 \\
\Leftrightarrow\ &\mathbb{E}[\hat{\theta}_m^2 - 2\hat{\theta}_m\theta + \theta^2] = \mathbb{E}[\hat{\theta}_m]^2 - 2\mathbb{E}[\hat{\theta}_m]\theta + \theta^2 + \mathbb{E}[\hat{\theta}_m^2] - \mathbb{E}[\hat{\theta}_m]^2 \\
\Leftrightarrow\ &\mathbb{E}[\hat{\theta}_m^2] - 2\mathbb{E}[\hat{\theta}_m]\theta + \theta^2 = \mathbb{E}[\hat{\theta}_m^2] - 2\mathbb{E}[\hat{\theta}_m]\theta + \theta^2
\end{aligned}$$

The Mean Squared Error can be decomposed into the Variance of the Estimator and the Bias of the Estimator. If the Mean Squared Error is low, the Estimator is good. The Goal of an Estimator is to minimize the Mean Squared Error.

3.4 Bias-Variance Tradeoff

As Depicted in the Figure, the Bias and the Variance are inversely proportional. If the Bias is high, the Variance is low and vice versa. The Goal is to find the sweet spot where the Bias and the Variance are both low. This is called the Bias-Variance Tradeoff.

3.5 Point Estamation and the Consinstency of the Estimator

A Point Estimator is consistent if it converges in Probability to the true value of the parameter. This means that the Estimator is getting better and better with more data. Formally this is written as $\lim_{m\to\infty} P(|\hat{\theta}_m - \theta| > \epsilon) = 0$.

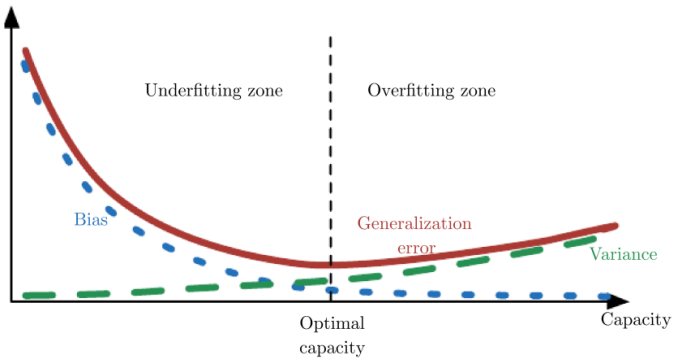


Figure 2:

3.6 Maximum Likelihood Estimation

The Maximum Likelihood Estimation is a method to estimate the parameter previously discussed. It is defined as $\hat{\theta}_{ML} = \arg\max_\theta p_{model}(X|\theta) = \arg\max_\theta \prod_{i=1}^{m} p_{model}(x_i|\theta)$. This means that the Maximum Likelihood Estimation maximizes $\hat{\theta}_{ML}$ given the data X. With no poof what so ever we will assume, that the Maximum Likelihood Estimation is consistent, when the true distrubution p_{data} is in the model class and p_{data} is a Value of θ

3.7 (Negative) Log Likelihood

Since it is numerically Problematic to take the Product, due to overflow/underflow we can take the Logarithm of the Likelihood. This is called the Log Likelihood. It is defined as

$$\begin{aligned}
\arg\max_\theta \log p_{model}(X|\theta) &= \arg\max_\theta \log \prod_{i=1}^{m} p_{model}(x_i|\theta) \\
&= \arg\max_\theta \sum_{i=1}^{m} \log p_{model}(x_i|\theta) \\
\arg\min_\theta \left(-\log p_{model}(X|\theta)\right) &= \arg\min_\theta \left(-\sum_{i=1}^{m} \log p_{model}(x_i|\theta)\right)
\end{aligned}$$

The last Equation is the Negative Log Likelihood. It is the same as the Log Likelihood, but with a minus sign in front. This is done to make it a minimization Problem, as most Optimization Algorithms are designed to minimize a Function.

3.8 Negative Log-Likelihood and Cross Entropy

Negative Log-Likelihood and Cross Entropy are equivalent when m approaches infinity. This is due to the fact that the Negative Log-Likelihood is the average

Cross Entropy. This can be shown as follows: Given that the samples x_i are drawn from the true data-generating distribution $p_{data}(x)$, and the negative log-likelihood (NLL) is divided by the number of samples m, the expression for $m \to \infty$ approximates the cross-entropy between the modeled distribution $p_{model}(x|\theta)$ and the true data-generating distribution $p_{data}(x)$:

$$\begin{aligned}\frac{1}{m}\sum_{i=1}^{m} -\log p_{model}(x_i|\theta) &\approx \mathbb{E}_{x\sim p_{data}}[-\log p_{model}(x|\theta)] \\ &= -\int p_{data}(x)\log p_{model}(x|\theta)\,dx \\ &\approx H(p_{data}, p_{model})\end{aligned}$$

As $m \to \infty$, the average NLL converges to the cross-entropy between p_{data} and p_{model}.

3.9 Conditional Negative Log-Likelihood

We can generalize the Negative Log-Likelihood to the Conditional Negative Log-Likelihood. This is done by conditioning the Negative Log-Likelihood on the input. It is defined as $\arg\min_\theta \sum_{i=1}^{m} -\log p_{model}(y_i|x_i,\theta)$. This is particularly useful, because it can be used in Classification, Segmentation, among many other Tasks, where we want to Intepret the conditional Probability of the output given the input.

3.10 Cross Entropy for Classification/Segmentation

In Classification and Segmentation Problems a Neural Network usually learns Probability Function right away lets call it $p_\theta(y|x)$ This leads to the following Negative Log Likelihood: $-\log p(Y|X,\theta) = \sum_i -\log p(y_i|x_i,\theta)$. Thus follows the Cross Entropy Loss: $H = \frac{1}{m}\sum_i -\log p_\theta(y_i|x_i)$

3.11 Segmentation and Pixel Wise Classification

In Segmentation and Pixel Wise Classification the Cross Entropy Loss is calculated for every Pixel. This is done by calculating the Cross Entropy Loss for every Pixel and then averaging over all Pixels. This is done to get a single Value for the Loss. The Figure is a visualization of the Cross Entropy Loss for a Pixel Wise Classification Task. Every Layer here represents a Class. So here the average for all Classes is calculated and then compared to get a Classification Result.

3.12 Mean Squared Error for Regression

If we consider the Normal Distribution as the Model Distribution, the mean of that is learned by the Neural Networks, considering most Regression Problems. This leads to the Likelihood being defined as $p(Y|X,\theta,\sigma) = \prod_i \mathcal{N}(y_i, \mu_\theta(x_i), \sigma)$ Thus follows the Negative Log Likelihood: $-\log p(Y|X,\theta,\sigma) = \sum_i \frac{1}{2}2(\pi\sigma^2) + \frac{1}{2}\frac{(y_i-\mu_i(x_i))^2}{\sigma^2}$. If we assume that the Variance is constant, we can drop the first term, also the Denominator can be dropped, as it is constant aswell. This leads

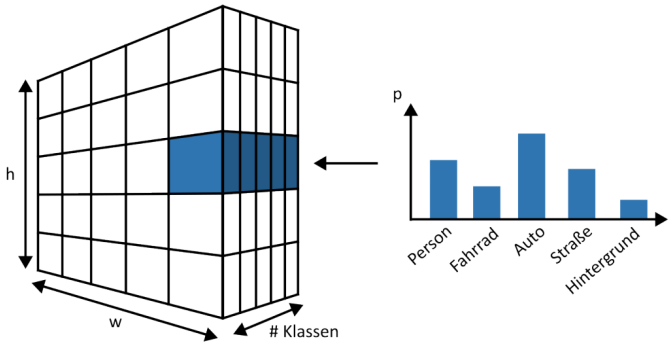


Figure 3:

to the Mean Squared Error: $MSE = \frac{1}{m}\sum_i (y_i - \mu_i(x_i))^2$ The minimum of the Mean Squared Error is if $\mu(x_i$ is the mean of the samples.)

3.13 Mean Absolute Error

For the MAE or $L1$ a Laplace Distribution is assumed. This leads to the Likelihood being defined as $p(Y|X, \theta, \sigma) = \prod_i \mathcal{L}(y_i, \mu_\theta(x_i), \sigma)$ With the NLL being defined as $-\log p(Y|X, \theta, \sigma) = \sum_i log(2\sigma) + \frac{|y_i - \mu_i(x_i)|}{\sigma}$. This leads to the Mean Absolute Error: $MAE = \frac{1}{m}\sum_i |y_i - \mu_i(x_i)|$ The Major difference is the missing squared making it more robust to outliers. But also the minimum is not the mean of the samples, but the median.

3.14 Maximum A Posteriori Estimation

An alternative to the Maximum Likelihood Estimation is the Maximum A Posteriori Estimation. It is defined as $\hat{Theta}_{MAP} = \arg\max_\theta p(\theta|X)$ Assuming θ is a random variable, we can use Bayes Rule to rewrite this as $\hat{\theta}_{MAP} = \arg\max_\theta p(X|\theta)p(\theta)$ To further simplify the operations, we can take the log as per usual. This leads to $\hat{\theta}_{MAP} = \arg\max_\theta \log p(X|\theta) + \log p(\theta)$ Here we can incoperate a sum again thus follows $\hat{\theta}_{MAP} = \arg\max_\theta \sum_i \log p(x_i|\theta) + \log p(\theta)$ The is as the Name already hints, the Maximum Likelihood Estimation with a Prior. This is useful if we have some prior knowledge about the parameter. With backround knowledge we can push θ to a realistic value. Say we have knowledge X we can have $P(Y|X, \theta)$ here the same with x follows $\hat{\theta}_{MAP} = \arg\max_\theta \sum_i \log p(x_i|x_i\theta) + \log p(\theta)$ This is a key Concept later on for Regularization of Weights for Neural Networks.

4 Error functions

The NLL $\hat{\theta}_{ML} = \arg\min_\theta(-\sum_{i=1}^m \log p(x_i|\theta)) = \arg\min_\theta f(\theta)$ the Maximum Likelihood Estimator will now be viewed as the minimum of a Function. One might notice that this does not look too different from minimizing or maximizing a function during Training in Machine-Learning. This would in that Context

be reffered to as Objective Function, cost function, error function or loss function. The Variable θ that is to be Optimized would then be equivalent to the Parameters of the Model. Thus the Maximum Likelihood Point Estimation is equivalent to a Machine Learning Algorithm. The obvious question now is, how do we minimize this function?

4.1 Local and Global Minima

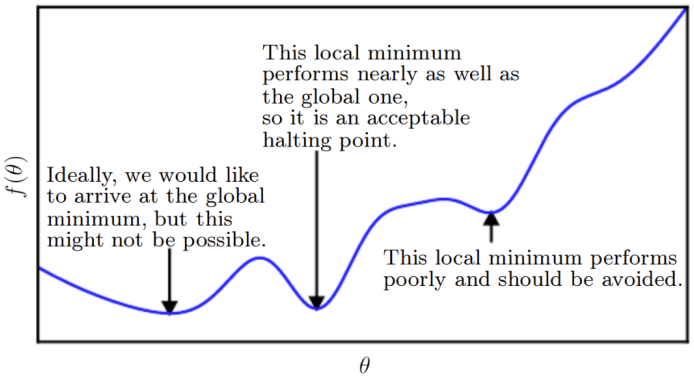


Figure 4:

As explained before the Local Minima is the one found by Gradient Descent. This might not be too bad if it is approximitely the same as the Global Minima. But if it is not, the Algorithm will get stuck in the Local Minima. A good rule of thumb is to compare the Local Minima to low values of the cost function, if they are not far off, the Local Minima is good enough.

4.2 Convex Functions

Convex function if $\forall x, y \in D \text{and} t \in [0,1] \rightarrow f(tx+(1-t)y) \leq tf(x)+(1-t)f(y)$ hold. This basically just means that every point is below the average of the two points. Convex Functions have the nice property that they have only one minimum. However this is rather rare in the context of Machine Learning.

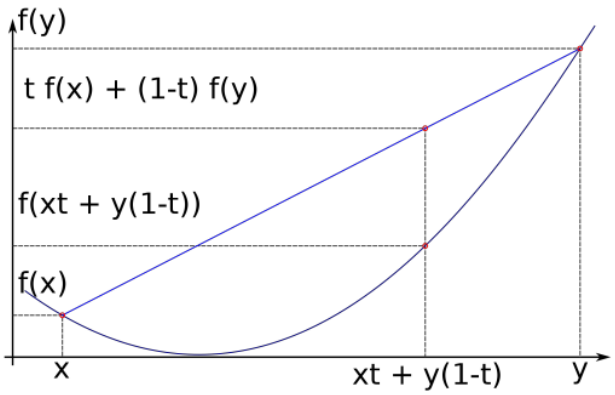


Figure 5:

4.3 Different approaches to find Minima

4.3.1 Least Squares

Least Squares is a method to find $\hat{\theta}$ for a linear Model this is done by taking $\arg\min_\theta ||A\theta - y||^2 = (A^*A)^{-1}A^*y$ This only works for least squares though and Machine Learning Models are usually not linear.

4.3.2 Grid Search

Grid Search is a method to find the minimum of a function. It is done by evaluating the function at every point on a grid. This is rather slow, as the number of points grows exponentially with the number of dimensions.

4.3.3 Evoluationary Algorithms

Evolutionary Algorithms are a class of algorithms that are inspired by the process of natural selection. They work by creating a population of solutions and then using genetic operators such as mutation and crossover to evolve the population over time. This is rather slow aswell, as it is a brute force method. It works fine for 10-1000 Parameters but becomes inneficient for more. It can be usefull for HyperParameters though.

4.3.4 Gradient Based Optimization

If the Model and the Error Function are differentiable, Gradient Based Optimization can be used. This is done by calculating the Gradient of the Error Function and then following the Gradient to the minimum.

5 Gradient Descent

5.1 Error Surface

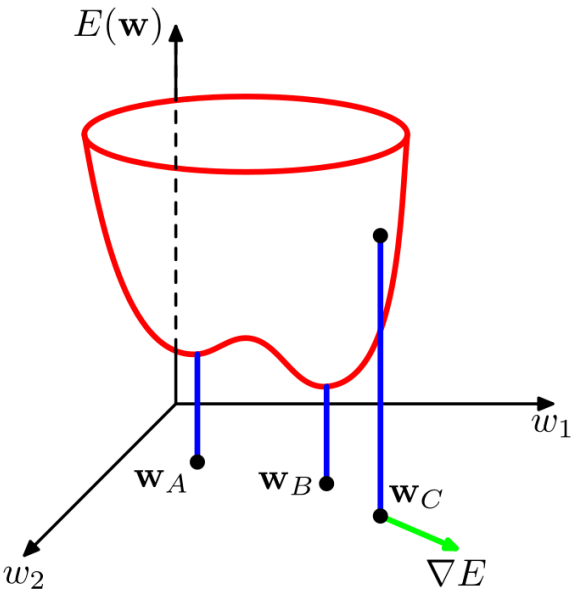


Figure 6:

This is a Geometric Representation of the Errror Function. w_A, w_B is a local and a Global minimum. Here Gradient Descent comes in handy. If we start at a random point, here w_C we can follow the Gradient to the minimum. So basically we just find the derivative of the function, follow the Gradient to the opposite direction and repeat this until we reach the minimum. The opposite direction is taken, as the Gradient points to the steepest ascent, and for minimization we want to go in the opposite direction.

5.2 Formal Definition of Gradient Descent

We are looking for θ that minimizes the Error Function $f(\theta)$ Since we can not do this analytically we will use a numerical method. For that we choose a starting value θ_0 and then update it iteratively. The update rule is defined as $\theta_{i+1} = \theta_i + \Delta\theta_i$ The usual way to do this is to introduce a learning rate ϵ and then update the parameter as follows $\theta_{i+1} = \theta_i - \epsilon\nabla f(\theta_i)$

5.3 Gradient - Taking the Derivative in Multiple Dimensions

The Error Function of a Neural Net is usually dependent on multiple Parameters. This is why it becomes necessary to talk about partial derivatives, thus the concept of Gradients is introduced. Let $f(\theta)$ be a skalar Function. Meaning it mapps from $\mathbb{R}^n \to \mathbb{R}$. The Gradient of f is defined as $\nabla f(\theta)_i = \frac{\partial f(\theta)}{\partial \theta_i} f(\theta)$ This is just the partial derivative of f with respect to θ_i. If u write this in vector form it becomes $\nabla f(\theta) = \begin{pmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \frac{\partial f(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_n} \end{pmatrix}$ So every entry of the Gradient is the partial derivative of f with respect to the corresponding parameter in our cases usually the weight. A stationary point is given if every entry of the Gradient is 0.

5.4 Direcional Derivative

The Directional Derivative describes the rate in which a function changes in a certain direction. It is defined as $\nabla_u f(\theta) = \lim_{\alpha \to 0} \frac{f(\theta + \alpha u) - f(\theta)}{h}$ This is just the derivative of f in the direction of u. This can be rewritten as $\nabla_u f(\theta) = \nabla f(\theta) u^T$ This is just the dot product of the Gradient and the direction. To minimize f, we must find the direction u, where f decreases the fastest, this means $min_{u,u^T u=1}\ u^T \nabla f(\theta) = min_{u,u^T u=1} ||u||_2 \cdot ||\nabla f(\theta)||_2 \cdot \cos(\phi)$. where ϕ is the angle between the Gradient and the direction. Since u is a unit vector, the minimum can be simplified to $min_u \cos(\phi)$, since the Gradient is independent of u. This is due to the fact that cos oxcilates between -1 and 1. Thus the minimun is -1 and $\cos(\pi) = -1$

5.5 Learning Rate ϵ

For now we will have just take ϵ as a small constant. The danger when ϵ is too large is that may not converge to the minimum. If ϵ is too small, the convergence will be slow. This is why it is important to choose ϵ wisely.

5.6 Jacobi Matrix

If we now have vectors as inputs and outputs it is handy to define a Matrix called Jacobi-Matrix. This matrix conatins the partial derivative of f at the position x. It is defined as

$$(J_f(\theta))_{i,j} = \frac{\partial}{\partial \theta_j} f(\theta)_i$$

This can be written in the following Matrix Form

$$J_f(\theta) = \begin{pmatrix} \frac{\partial f_1(\theta)}{\partial \theta_1} & \frac{\partial f_1(\theta)}{\partial \theta_2} & \cdots & \frac{\partial f_1(\theta)}{\partial \theta_n} \\ \frac{\partial f_2(\theta)}{\partial \theta_1} & \frac{\partial f_2(\theta)}{\partial \theta_2} & \cdots & \frac{\partial f_2(\theta)}{\partial \theta_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(\theta)}{\partial \theta_1} & \frac{\partial f_m(\theta)}{\partial \theta_2} & \cdots & \frac{\partial f_m(\theta)}{\partial \theta_n} \end{pmatrix}$$

5.7 The Derivative of second order

The derivative of second order is important for us as well since it can imply the improvement that is expected of one iteration of the Gradient Descent.

5.8 Hessian Matrix

The Hessian Matrix is the Matrix of second order partial derivatives of a function. It is defined as

$$H_f(\theta)_{i,j} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} f(\theta)$$

The directional Derivative of second order to a Directionvector u is given by $u^T H_f(\theta) u$. If u is an Eigenvektor of H, then the derivative of second order is equivalent to the corresponding Eigenvalue. For other directions, the derivative of second order is the weighted average of the Eigenvalues. Where Directions which correspond better with the Eigenvalues are more important. This also implies that the second directional derivative is beween the smallest and the largest Eigenvalue of H, because it is a weighted sum, it must be contrained by the smallest and the largest Eigenvalue.

This is also equivalent to the Jacobi Matrix of the Gradient.

5.9 Approximate Second-Order Methods

Let's have a look at the Newton Method. The Newton Method is used to find the minimum of a function. It is defined as $\theta_{i+1} = \theta_i - H_f(\theta_i)^{-1} \nabla f(\theta_i)$ This is just the Gradient Descent with the Hessian Matrix. I mean it makes sense though, since we basically exchanged our shitty ϵ to a more dynamic way of choosing the learning rate. It brings some disatvantages though, as the Hessian Matrix calculation as well as the inversion can be rather expensive for large disatvantages.

6 Stochastic Gradient Descent

If the calculation of the Gradient is done for the entire Dataset, it might be slow, due to the immense size. Also the data during an iteration is saved in RAM or Graphicsmemory, this can be become impossible to handle for large Datasets. This is why we introduce Stochastic Gradient Descent. Which takes Batches of the Data and calculates the Gradient for these Batches. Lets assume that the Trainingsample is i.i.d. then the log likelihood can be written as $L_\theta(X) = \frac{1}{n} \sum_{i=1}^{n} L(f_\theta(x_i), y_i)$ which implicates that the Gradient can be written as $\nabla_\theta L_\theta(X) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(f_\theta(x_i), y_i)$ This is just the average of the Gradients of the Batches.

6.1 Mini-Batch Gradient as an unbiased Estimator

The Approximation of the Gradient with the help of Mini-Batches can be seen as an unbiased Estimator of the Gradient. In math language we can write

$$\begin{aligned}
\mathbb{E}\left[\nabla_\theta L_\theta(\hat{X})\right] &= \mathbb{E}\left[\frac{1}{k}\sum_{x\in\hat{X}} \nabla_\theta L_\theta(x)\right] \\
&= \frac{1}{|\mathcal{X}_k|}\sum_{\hat{X}\in\mathcal{X}_k} \frac{1}{k}\sum_{x\in\hat{X}} \nabla_\theta L_\theta(x) \\
&= \frac{1}{k}\sum_{x\in X} \frac{\binom{n-1}{k-1}}{\binom{n}{k}} \nabla_\theta L_\theta(x) \\
&= \frac{1}{k}\sum_{x\in X} \frac{k}{n} \nabla_\theta L_\theta(x) \\
&= \frac{1}{n}\sum_{x\in X} \nabla_\theta L_\theta(x) \\
&= \nabla_\theta L_\theta(X)
\end{aligned}$$

The given equations demonstrate that the expected gradient of the loss function, when computed over a randomly sampled subset of the data, equals the gradient of the loss function over the entire dataset. This result shows that using mini-batches in stochastic gradient descent provides an unbiased estimate of the full gradient. As a result, SGD can efficiently and effectively optimize the model parameters by using only small subsets of data at each iteration, making it scalable and practical for large datasets. (rewritten by chatgpt so it becomes more coherent lol)

6.2 Batch Size

The Batch size is as per usual a trade off, either we choose large Batches. This would lead to a more accurate Gradient, but needs to more computation power. If we choose small Batches, the Gradient is less accurate, but the computation is faster. This makes it necessary to choose a smaller learning rate so the training remains stable. For the Varianz of the Gradient $Var(\frac{1}{|I|}\sum_{i\in I} g_i) = \frac{1}{|I|^2}\sum_{i\in I} Var(g_i) = \frac{\sigma^2}{|I|}$ I is a set of indices of the Patch and the Gradients are g_i This makes sense as it exactly depicts the relation we established before the equation. The choice of the Batch size, since it is a Tradeoff depends on several Factors

1. If the Batches are processed in parallel, the Batch size should be chosen to fit the memory of the GPU.
2. Multiprozessor Systems can't be used to their full potential if the Batch size is too small. Since the Batches have fixed cost for the computation at a certain size.
3. Hardware is optimized for certain Batch sizes. For GPUs this is usually 2^n.
4. Small Batches have a Regularization effect, as the Gradient is more noisy.

6.3 Algorithm for Stochastic Gradient Descent

Algorithm 1 Stochastic gradient descent (SGD) update at training iteration k

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

Lets go through the Algorithm step by step. First we give a learning rate and an initial parameter. Then we sample a minibatch of m examples from the training set. We compute the gradient estimate by taking the average of the gradients of the loss function over the minibatch. We then apply the update rule to the parameter by subtracting the learning rate times the gradient estimate. We repeat this process until a stopping criterion is met, which could be a maximum number of iterations, a convergence criterion, or another condition.

6.4 Learning Rate ϵ

The first Parameter was the Learning Rate ϵ. The Learning Rate needs to go to Infinity, so $\sum_{k=1}^{\infty} \epsilon_k = \infty$ and $\sum_{k=1}^{\infty} \epsilon = \infty$ Usually the learning rate is being reduced over time. This is done by a learning rate schedule. For example a linear decay, where the learning rate is reduced by a contant factor until it reaches a certain iteration, where is remains constant.

6.5 Momentum

6.6 Algorithm for Stochastic Gradient Descent with Momentum

Momentum is a method to accelerate the convergence of Gradient Descent. Momentum aggregates the Gradients of previous Iterations.

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

This is just the Gradient Descent with the addition of the Momentum. The Momentum is a Hyperparameter and is usually set to 0.9, 0.99, or 0.5. The Momentum is then used to update the parameter as follows $\theta \leftarrow \theta + v$ This is just the Gradient Descent with the addition of the Momentum. The Momentum is a Hyperparameter and is usually set to 0.9, 0.99, or 0.5. The Momentum is then used to update the parameter as follows $\theta \leftarrow \theta - v$

7 Adaptive Learning Rates

This basically just means that we can adapt the learning rate to the Parameters. There are several methods to do this.

Algorithm 2 Stochastic gradient descent (SGD) with momentum

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

1. AdaGrad scales a learning rate by the inverse of the square root of the sum of all historical squared gradients. This means that the learning rate is reduced for parameters that have received large gradients in the past.

2. RMSProp is similar to AdaGrad, but uses a moving average of squared gradients instead of the sum of squared gradients. This allows the learning rate to adapt more quickly to changes in the gradients.

3. Adam combines the ideas of momentum and RMSProp. It uses a moving average of gradients and a moving average of squared gradients to adapt the learning rate for each parameter.

7.1 AdaGrad Algorithm

Algorithm 3 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter θ
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $r = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{x^{(1)}, \ldots, x^{(m)}\}$ with corresponding targets $y^{(i)}$
Compute gradient: $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
Accumulate squared gradient: $r \leftarrow r + g \odot g$
Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$ (Division and square root applied element-wise)
Apply update: $\theta \leftarrow \theta + \Delta\theta$
end while

So the steps added here are the Accumulation of the squared Gradient, and the update. As mentioned in the short description before, since the Multiplikation and the Division are applied element wise, the update is done for every parameter and so every parameter that was previosuly large will have a reduced learning rate. The other algorithms for other adaptive learning rates are similar to this one, but with the steps I descibed before.

8 Feed-Forward Networks

8.1 Single-Neuron

The output of single Neuron with D input values x_i, weights w_i and Bias b can be calculated as follows:

$$a = \sigma(\sum_{i=1}^{D} w_i x_i + b)$$

where σ is the activation function. The activation function is usually a non-linear function, such as the sigmoid function, the tanh function, or the ReLU function. The output of the neuron is then passed through the activation function to produce a non linear output. This is done, so that the Model can learn non-linear relationships between the input and the output, otherwise the Model would just be a linear regression.

8.2 One Layer and multiple Neurons

Often a layer is defined by Multiple Neruons, those are fed with the same input and combined in one layer. The output of such a layer can be calculated by

$$a = \sigma(\sum_{i=1}^{D} w_{ji} \cdot x_i + b_j)$$

The Compact form of this would be $a = \sigma(Wx + b)$ where W is the Matrix of the weights and b is the Bias.

8.3 Multiple Layers

Well, now that we have the definition of one Layer down, why not do Multiple and thus create a neural Net. The output of a Neural Net with L layers can be calculated as follows:

$$\begin{aligned} a^{(1)} &= \sigma(W^{(1)}x + b^{(1)}) \\ a^{(2)} &= \sigma(W^{(2)}a^{(1)} + b^{(2)}) \\ &\vdots \\ a^{(L)} &= f^{out}(W^{(L)}a^{(L-1)} + b^{(L)}) \end{aligned}$$

Now we can consider writing this as a parameterized function

$$\mathrm{nn}_\theta(x) = f_{\text{out}}\left(W^{(N)} \cdot f^{(N-1)}\left(\dots W^{(2)} \cdot f^{(1)}\left(W^{(1)} \cdot x + b^{(1)}\right) + b^{(2)} \dots\right) + b^{(N)}\right)$$

With the parameters:

$$\theta = \left((W^{(1)}, b^{(1)}), (W^{(2)}, b^{(2)}), \dots, (W^{(N)}, b^{(N)})\right)$$

The f I use is just a placeholder for the activation function.

How many layers exist is something we will define with the amount of layers being the amount of weight matrices used for Multiplication.

8.4 Fun Facts for Linear Activation

Lets say we decide to use a linear function, lets show by its properties that it is not a good choice.

$$\text{nn}_\theta(x) = f_{\text{out}}\left(W^{(N)} \cdot f^{(N-1)}\left(\cdots f^{(2)}\left(W^{(2)} \cdot f^{(1)}\left(W^{(1)} \cdot x\right)\right)\right)\right)$$

This can be simplified by grouping the weight matrices and activation functions:

$$\text{nn}_\theta(x) = f_{\text{out}}\left(W^{(N)} \cdot f^{(N-1)}\left(\cdots f^{(2)}\left(W^{(2)} \cdot W^{(1)} \cdot f^{(1)}(x)\right)\right)\right)$$

Combining weights into a single matrix W':

$$\text{nn}_\theta(x) = f_{\text{out}}\left(W^{(N)} \cdots W^{(2)} \cdot W^{(1)} \cdot f^{(N-1)} \circ f^{(2)} \circ f^{(1)}(x)\right)$$

Finally, we get:

$$\text{nn}_\theta(x) = f_{\text{out}}\left(W' g(x)\right)$$

8.5 Activation Functions

The Activation Function is a non-linear function that is applied to the output of a neuron. As shown before we do need those non-linear Activation Functions to learn non-linear relationships. We can differentiate beween two Sorts of here, one being for Hidden Layers (So the Layers between the Input and the Output) and the other for the Output Layer. Some of the Hidden Layer Activation Functions are:

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Tanh: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU: $ReLU(x) = \max(0, x)$
- Hard Tanh: $HardTanh(x) = \max(-1, \min(1, x))$
- Softplus: $Softplus(x) = \log(1 + e^x)$
-

Some of the Output Layer Activation Functions are:

- Linear: $f(x) = x$
- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Softmax: $Softmax(x) = \frac{e^{x_i}}{\sum_j e^{x_j}}$

For the output Functions it is maybe important to note that you can use the Sigmoid Function for Binary Classification and the Softmax Function for Multiclass Classification.

8.6 Symmetries in Parameter Space

There are many ways to achieve the same mapping of input to output values, even though the weight matrices and biases are different. One way is to permute the neurons in a layer, this leads to $M!$ equivalent solutions. Some activation functions are also point reflections, meaning that $f(x) = f(-x)$, this leads to 2^M equivalent solutions. An Example for this would be $tanh(-x) = -tanh(x)$ If we combine those symmetries, we get $2^M \cdot M!$ equivalent solutions. This also there are as many equivalent local Minima. Which is ok though, as large networks local Minimas are usually quite good.

8.7 Universal Approximation Theorem

This Theorem states that if we have 2 layers, with enough hidden units, we can approximate any Continuous Function.

9 Backpropagation

9.1 Why do we need Backpropagation?

As discussed in Chapter 5 we use gradient descent to minimize the error function. The error function is a function of the parameters of the model, and we need to compute the gradient of the error function with respect to the parameters in order to update them. Perhaps a rather simple approach would be to to calculate the Difference quotient being defined by

$$\frac{\partial L}{\partial W_{ji}} = \frac{L(W_{ji} + \epsilon) - L(W_{ji} - \epsilon)}{2\epsilon} + \underbrace{O(\epsilon^2)}_{residual\ corrections}$$

This is a rather simple approach, but it is not very efficient, since you need to evaluate the Loss-Function twice for every Parameter. As this is why we use the Backpropagation Algorithm.

9.2 Lets look at the Chain Rule

For Backpropagation we first need to understand the chain rule. The chain rule in 1D is defined as $(g \circ f)'(x) = g'(f(x)) \cdot f'(x)$. This is also written as $\frac{x}{\partial x} g(f(x)) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial x}$. If our function depends on multiple variables, for example x and y then we can look at the partial derivatives. Meaning the partial derivative of $f(x, y)$ with respect to x can be written as $\frac{\partial f}{\partial x}$. And with respect to y as $\frac{\partial f}{\partial y}$.

9.3 Direcional Derivative with the Chain Rule

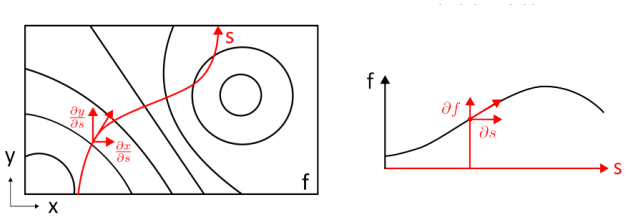
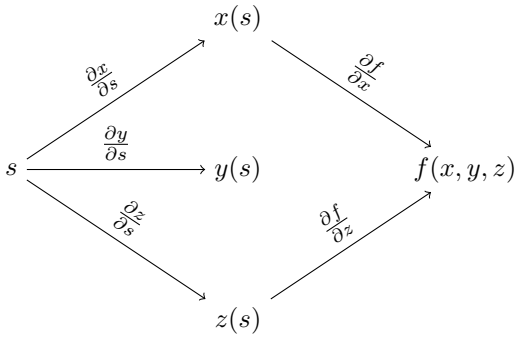


Figure 7:

Ok lets say our goal is to calculate the partial derivative of f to s, on the left graph. Well to analyze we take the chain rule, meaning we can express f as $f(x(s), y(s))$ and can then write the partial derivative of f with respect to s so $\frac{\partial f}{\partial s} = \frac{\partial f}{x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s}$. This is just the chain rule applied to the partial derivatives. So if more Dimensions follow, we can use the same pattern.



$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial s}$$

This concept can easily be expanded to more functions, lets say we also have g we then need to calculate g in respect to s, to do this we can say

$$\frac{g}{\partial s} = \frac{\partial g}{\partial f}\frac{\partial f}{\partial s}$$

Since I had trouble understanding it, lets make a quick example for g and f. If g and f are dependent on one variable s.

$$f(s) = s^2$$

$$g(f) = f + 1$$

We want to find $\frac{\partial g}{\partial s}$.

1. Compute $\frac{\partial f}{\partial s}$:
$$\frac{\partial f}{\partial s} = \frac{\partial (s^2)}{\partial s} = 2s$$
2. Compute $\frac{\partial g}{\partial f}$:
$$\frac{\partial g}{\partial f} = \frac{\partial (f+1)}{\partial f} = 1$$
3. Apply the chain rule:
$$\frac{\partial g}{\partial s} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial s} = 1 \cdot 2s = 2s$$

Thus, the derivative of g with respect to s is:
$$\frac{\partial g}{\partial s} = 2s$$

With more variables and more functions this becomes increasingly annoying to write down, which is why it looks so complicated in the Slides. But with Neural Nets we typically have many layers, thus making these "graphs" unsuitable for the task. Now this where Backpropagation finally comes in handy.

9.4 Backpropagation Algorithm Intuitively

Well lets talk about Backpropagation in a more intuitiv manner before discussing the calculus involved. We first run the forward pass so we pass in Training examples and then calculate the output of the Neural Net. If we think of classifiying a number for example, this will most likely not be the correct output. So we calculate the error of that output thus giving us the loss, so how far off our predictions are from the actual values. Ok now we can apply the backward pass. We basically go back through the layers and check where mistakes were made. Thus we calculate how much each weight in the network contributed to the error. So now we can adjust the weights to minimize the loss in the Neural Net. We can do this over and over again for each Training example, until the loss is minimized. Chapter 3 of the 3B1B Video Series on Neural Networks might be worth to check out.

9.5 Backpropagation Algorithm Formally

Ok lets first look at simple example and then expaldn it, lets say we only have 1 Neuron per layer now lets introduce some notations, as shown in the image. Lets focus on the connection of the last 2 Neurons first. a^L being the activation of the last neuron, and $a^{(L-1)}$ being the activation of the previous neuron. So how do we determine the last activation well as we discussed it is determined by a weight $w^{(L)}$ the previous action $a^{(L-1)}$ and a bias $a = b^{(L)}$ and then an activation function. We also dicussed that we define a name for that here being $z^{(L)} = w^{(L)} \cdot a^{(L-1)} + b^{(L)}$ and $a^{(L)} = \sigma(z^{(L)})$. Now that we computed a as shown in the image we can compute the cost with a target value y and a cost function C This can be expanded to substituting the a^{L-1} function as well of course. But lets focus on the last layer for now. The first thing that might be interesting here is how changes in the weight effect our cost function. In mathematical

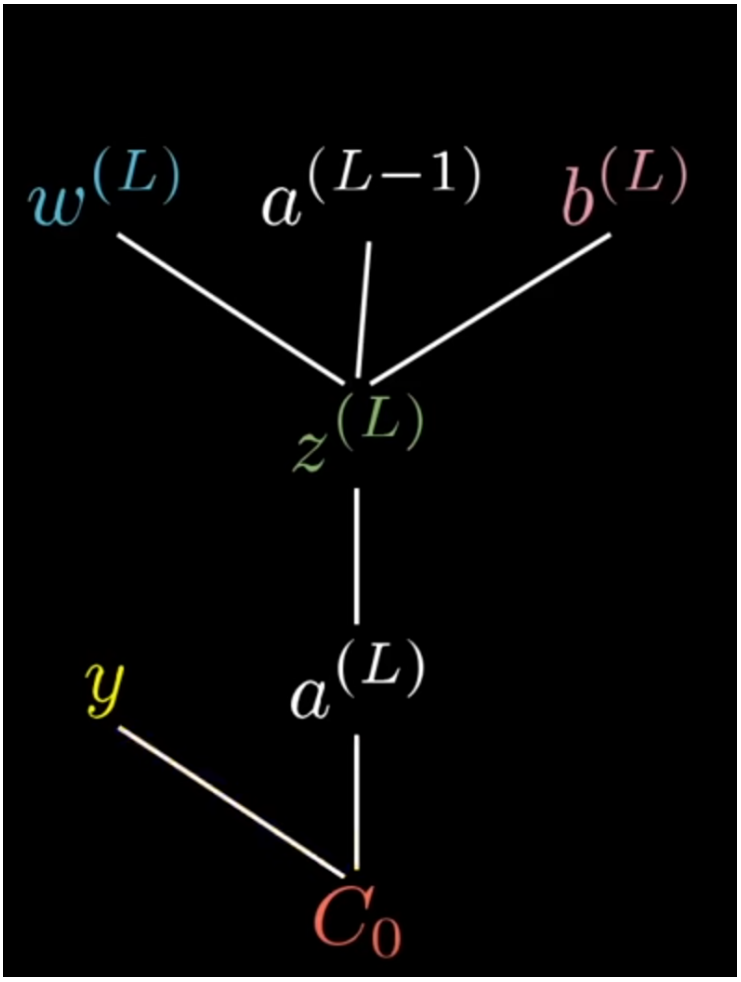


Figure 8: 3B1B Backpropagation

Terms we are interested in the derivative of the cost function with respect to the weight. $\frac{\partial C_0}{\partial W^{(L)}}$ But how do we get there again with intuition the chain rule makes perfect sense here because what is important for that ratio to know. Well first $w^{(L)}$ effects $z^{(L)}$ which in turn effects $a^{(L)}$ which in turn effects the cost function. So lets write this down mathematically using the chain rule!

$$\frac{\partial C_0}{\partial W^{(L)}} = \underbrace{\frac{\partial z^{(L)}}{\partial w^{(L)}}}_{\substack{\text{effect of } w \\ \text{with respect to } z}} \quad \underbrace{\frac{\partial a^{(L)}}{\partial z^{(L)}}}_{\substack{\text{now that } z \text{ changed} \\ \text{let's calculate the effect} \\ \text{that has on } a}} \quad \underbrace{\frac{\partial C_0}{\partial a^{(L)}}}_{\substack{\text{effect of } a \\ \text{on the cost } C_0}}$$

Lets break down the terms now. So first the effect of a on the term C is just $2(a^{(L)} - y)$ The derivative of the activation function is just the derivative of the activation function. And the derivative of z with respect to w is just $a^{(L-1)}$. Because this relation just depends on ow strong the previous neuron was activated. Now this is just for a single training example if we wanted to compute this for all the Training examples we would compute an average meaning

$$\frac{\partial C}{\partial W^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial W^{(L)}}$$

Now we have succesfully computed one component of the gradient vector C We would now need to do the same for the bias and the previous layers. For the bias the computation is almost the same, because we can just swap out every w for a b. But if we consider the derivative of $\frac{\partial z^{(L)}}{\partial b^{(L)}}$ we see that this is just 1. Because obviosuly a change in the Bias would result in a change of the activation exactly equivalent to the change in the Bias. Ok now lets look at how we go backward. The sensitivity of the cost function with respect to the activation of the last layer. In the formula it would look like this:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \cdots$$

which in turn would just be the weight. Now since we kept track of this we can trace back how sensitive the cost function is to the activation of the previous layer. This is the basic idea of Backpropagation. Now this can be expanded to multiple neurons, however the Idea stays the same. One thing that changes is the cost function with respect to the activation of the previous layer. Lets introduce a new indeci to keep track which neuron talking about lets call it activation of the kth neuron in the L-1 layer with $a_k^{(L-1)}$ thus the cost function with respect to the activation of the kth neuron in the L-1 layer would be $\frac{\partial C_0}{\partial a_k^{(L-1)}}$ This can be computed by the chain rule as well, but we need to consider the entire layer L so we compute the sum which leads to the following equation

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L - 1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

I personally found the slides to be rather confusing that is why i mainly used 3B1B 4th video of his Neural Network Series to explain this.

9.6 TODO explain this with jacobi matrices (i think before should work to understand the concept though)

10 Regularization

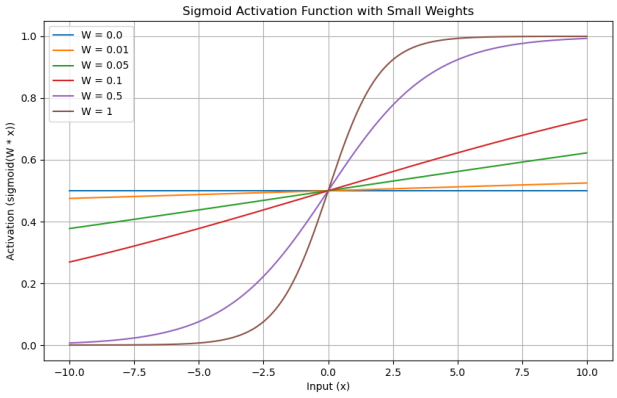
10.1 Gradient Clipping

Sometimes Deep Neural Networks, have the issue of exploding Gradients. This Problem occurs if Multiplying large weights. Visually this is explodingly steep parts in a Loss Function. The Problem is that with our current approach the Parameters of the Model would be updated by a large amount, making our previous progress in Optimizing useless. This is why we introduce Gradient Clipping. This is done by setting a Threshold for the Gradient. Here 2 Methods are common, the first one is to clip the Gradient if an Entry of the Gradient exeeds a certain Threshold, this is refered to as Value-Clipping. The second one would be Norm-Clipping if the Gradient exceeds a certain Norm, the Gradient is scaled down to the Threshold, this is refered to as Norm-Clipping.

10.2 Parameter Initialization

10.2.1 Very Small Weights

Lets talk about what happens with certain weights, starting with small weights. If the weights are too small, the Network might collaps to a linear Modell, which as previosuly discussed is not what we want. If we use the Sigmoid Activation Function and apply it to $W \cdot x$ with W being very small the Function becomes linear.



10.2.2 Very Large Weights

Problems also arise with Very Large weights. With very large weights the Sigmoid Function becomes very steep, which makes the Gradient very small. This is a problem because the Gradient Descent would be very slow.

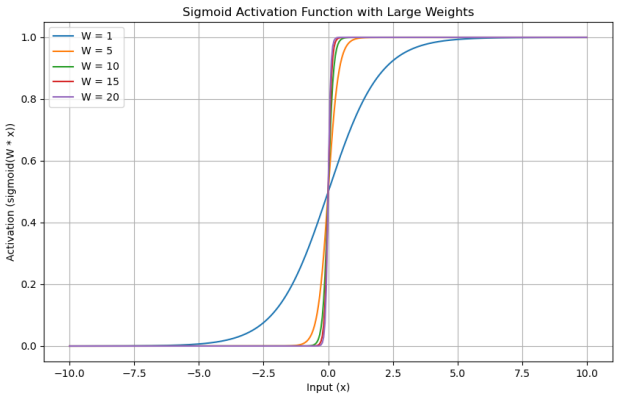


Figure 9: Sigmod with Large Weights

10.3 Scalng the Input Values

The range of Inputs determines the range of values for the Weights, thus making it important to scale the Input Values, because as we have seen this determines the quality of the Neural Net. A good rule of thumb is to scale the Inputs to have a mean of 0 and a standard deviation of 1.

10.4 Weight Initialization

Usually the weights are initialized randomly with small values. So the network starts linear, but this will not remain the case when the weights are updated during training. Some might think it would make sense to initialize the weights with 0, but this would be problematic with the simplest reason being that the input for second layer would be the same for all Nodes, the output would be the same as well and would then be multiplied by the same weights. This would lead to the same output for all Nodes in the second layer, this goes on until the output layer, where it will still be zero. Another reason would be that backpropagation would not work, as the weights will be multiplied by a value determined by backpropagation but since the weights are zero they will remain zero, making our whole progress useless.

10.5 Parameter Initialization

We want to initialize our parameter so that the Eigenvalues of the Jacobi Matrix and the Weights are close to 1. If the Eigenvalues are larger than 1, the Gradient will explode, if they are smaller than 1, the Gradient will vanish. This means that for the Jacobi Matrix of the actiavtion function with $J_{f^{(i)}}(a^{(i)})$ we

want the derivative of the activation function to be close to 1. This is different from activation activation function to activation function though.

- Input Signals are usually scaled to have a mean of 0 and a standard deviation of 1.
- Since the standard deviation of the output of a neuron is 1, the weights should be initialized and multiplied $\frac{1}{\sqrt{n}}$ with n being the number of inputs. This ensures ensures that the variance of the weighted sum remains 1, because the deviation of the weights is σ^2.
- The bias changes the mean of the output, usually it remains 0 sometimes it makes sense to initialize it differently however. An example for that would be ReLU, where the bias should be initialized with a small positive value, so values lower than 0 are passed through the ReLU function.

10.6 Scaling of the Weights as an Hyperparameter

If we have enough computational power, we can use the scaling of the weights as a hyperparameter. This means that we can try different values for the scaling of the weights and see which one works best we can just experiment on Mini Batches though and check wether the Hidden Units are getting lower each layer.

10.7 Overfitting and Underfitting quick comeback in Context

While Input and Output are usually fixed, the number of Hidden Units is a Capacity Parameter, we need to figure out. If we use too many neurons, the network will overfit, if we use too few neurons, the network will underfit. Deep Neural Networks are usually volotile to Overfitting, due to its high capacity. To prevent that Regularization is used.

10.8 Definition of Regularization

Regularization can be defined as any modification of a learning algorithm with the purpose of reducing the generalization error without significantly increasing the training error. Common strategies to achieve this are adding more Terms to the Loss and Extra Constrains to the Modelparameters. Those strategies can use prior knowledge about the problem, but can also try to find a simple model for good generalization.

10.9 Regularization in Deep Learning

Most Regularization strategies want to find a good Bias Varianz Tradeoff. Which we have also talked about at the beginning.

10.10 Parameter Norm Penalties

One way too force our Model to have a certain complexity is a so called Penality Term. This Term is added to the Loss Function. The Term is controlled by a

Hyperparameter α and the Norm of the Parameters.

$$\tilde{L}(\theta; X, y) = \underbrace{L(\theta; X, y)}_{\text{Original Loss Function}} + \underbrace{\alpha \cdot \Omega(\theta)}_{\text{Penalty Term}}$$

The Parameter Norm Penalty usually only applies to large Values in the weights, but not in the Biases, since the Bias only Influences a single Variable, meaning it needs little Data to be learned. If we would Introduce a Penalty Term for the Bias, underfitting becomes our Problem.

10.11 L2 Regularization

A common and simple approach is the L2 Regularization. The L2 Regularization is defined as the sum of the squares of the weights.

$$\tilde{L}(\theta; X, y) = \underbrace{L(\theta; X, y)}_{\text{Original Loss Function}} + \underbrace{\alpha \frac{1}{2} ||w||^2}_{\text{L2 Norm Penalty Term}}$$

This is also known as weight decay, because the weights are decaying to zero. In the Context of Backpropagation this means that the Gradient of the Penalty Term is added to the Gradient of the Loss Function.

10.11.1 TODO There is an intuition slide in the slides, which i cant explain well

10.12 L1 Regularization

We have had the L2 Regularization, now lets talk about the L1 Regularization. The L1 Regularization is defined as the sum of the absolute values of the weights. For the loos function this means The loss Function now is defined as:

$$\tilde{L}(w; X, y) = L(w; X, y) + \alpha \cdot \|w\|_1$$

and the Gradient:

$$\nabla_w \tilde{L}(w; X, y) = \nabla_w L(w; X, y) + \alpha \cdot \text{sign}(w)$$

10.13 Sparse Networks

The advantage of L1 Regularization is that it leads to sparse Networks. This means that the Network has many weights that are zero, so not Unit every unit in one layer is connected to every unit in another layer. This is useful because it saves computation time.

10.14 Representation Sparsity

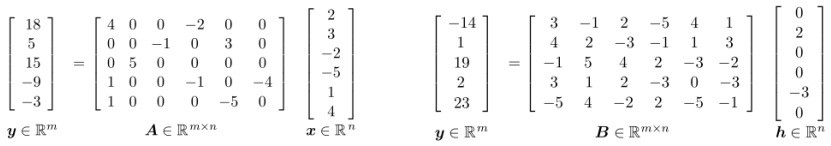


Figure: Gewichts vs. Repräsentations Spärlichkeit, von Goodfellow

Figure 10: Different Sparsity

As visible in the figure in the Second way of writing the Vector the Vector multiply just forces Values to be sparse, since the Vector has Zero Values, while the weight Sparsity sets values of the Martix to 0.

10.15 Regularization as Maximum A Posteriori Estimation

The L1 and L2 Regularization can also be interpreted as Maximum A Posteriori Estimation. This means that we are looking for the most probable parameters given the data. The L2 Regularization can be interpreted as a Gaussian Prior, while the L1 Regularization can be interpreted as a Laplace Prior. This is not carried out in the slides, but I guess if you leave out the constant erm of the gaus prior we see that α is equivalent to $\frac{1}{2\sigma^2}$. This surely works pretty similar for the Laplace distrubution.

11 Reduction of Modelcapacity

11.1 Early Stopping

One way to reduce the Modelcapacity is to stop the training early. This is done by monitoring the Validation Set and stopping the training when the Validation Error starts to increase. This is done because the Model is starting to overfit. The problem is that the Model might overfit faster, due to it recognizing patterns fast, it shouldnt have by training with the Validation Set.

11.2 Exponential Moving Average

We have learned about SGD already, here and issue might be the noise from the random selection of training data, this could lead to instable updating, therefore we introduce EMA which is defined as

$$\theta_{EMA,t+1} = (1 - \lambda)\theta_{EMA,t} + \lambda\theta_t$$

If $\lambda = 1$ then the EMA is just the parameter itself. If $\lambda = 0$ then the EMA is just the previous EMA.

11.3 Parameter Sharing

In Regularization we assumed that we know, what the Model should look like, and thus can know which value will make sense and nudge the model in that direction, for example 0. This is not always the case, but sometimes we have knowldege the knowledge that for example two Models are close together. So lets say we have a Model A and a Model B with their respective parameters, we could apply a Parameter Penality, to force the Models to be similar, since we know they are. This would look like this

$$\Omega(\theta_A, \theta_B) = ||\theta_A - \theta_B||_2^2$$

We could also force the Parameters to be equivalent, which is done by Parameter Sharing. By Sharing parameters we essenstially tie weights together, meaning updates to the weights from one part of the Model will directly effect the others. If we think in text of Convolutional Neural Networks, we can think of the same filter being applied to different parts of the image. (This will be discussed later)

11.4 Data Augmentation

Data Augmentation is a technique to increase the size of the training set. This is done by applying transformations to the training data. (The one Python task from the practical Worksheet with the CIFAR Dataset)

11.5 Injecting Noise

Well we use Data Augmentation to make the Data a bit noisy so that the Neural Net is able to deal with Noise. We can also add noise to hidden units. We can even add noise to the weights.

11.6 Noisy Outputs

We can make the Outputs Noisy as well to make the Model robust to the Training Label being incorrect. This is done by saying our training label is just correct with a certain probability $1 - \epsilon$

12 Bagging and Dropout

12.1 Bagging

Bagging comes from the term Bootstrap Aggregating. This is basically the Introduction of Democracy. We train several models separately and then average the results, so every Model has the same vote in the final decision. This can be written as

$$p_{bagging} = \frac{1}{k} \sum_{i=1}^{k} p_i(y|x)$$

12.2 Expected Error of Bagging

If the test samples all do the same error, the expected error of the bagging is the same as the expected error of the single model. If the test samples do different errors, the expected error of the bagging is smaller than the expected error of the single model.

12.3 Creating the Models for Bagging

The Models for Bagging are created by training the Model on different subsets of the training data. This is done by sampling the training data with replacement. This means that the same sample can be selected multiple times. This quite computationaly expensive, therefore we take a closer look at Dropout.

12.4 Dropout

Dropout is also to reduce complexity and overfitting, but it is an efficient way of performing Bagging, it just omits data during training. This is done by setting the output of a neuron to 0 with a certain probability. This is done for every neuron in the network. It just trains ensembles of all possible sub networks.(An ensemble refers to a combination of multiple models to solve a particular problem.) TODO why is this more efficient?

12.5 Inference in Dropout

First talk about how we calculate the Dropout. The Dropout $p_dropout$ is calculated by summing over all binary masks μ averaging over all possible sub networks, which can be formed by applying different binary masks. Now this is not computationally feasible, so we approximate this by randomly sampling binary masks, which turns out to be quite a good approximation.

12.5.1 Weight Scaling Inference Rule

Ok so during Dropout we set the Neurons to zero with a certain Probabiliy p, as discussed before. But now we would like to use the entire Network, so we use something called Weight Scaling. During Inference we scale the Network Weights to $1 - p$, this scaling accounts for the fact that during training we set the Neurons to zero with a certain probability.

12.6 Bagging vs Dropout

One major difference is that Bagging trains every model independently, while dropouts Subnetworks share Parameters, thus making it possible to compute even regarding the exponential number of possible Models. In Bagging every single Model is trained until converging this is not the case for Dropout. Due to the sheer Size of Dropout, we just train a fraction of the possible Models, then sharing is caring comes into play and we share the Parameters across the Models. All differences asside both methods are quite similar as Dropout Subnetworks can also be seen as Subsets of the original Network, which also are created when performing Bagging.

12.7 Effectivity of Dropout

Dropout is very effective in Regularization, and can be combined with other Regularization Techniques, we were Previosuly discussed. Dropout computes a Binary Mask per Training Sample in $O(n)$, and needs $O(n)$ Memory, to store the Binary Mask for Backpropagation. Weight Scaling is applied once for all Modelweights, thus making it Computationally irrelevant for Dropout. For Dropout it is important to note that we need large Models, since we reduce the Capacity of the Network by Dropout.

12.8 Dropout in the Context of Noise Augmentation

We used and learned about Augmentation, some might have noticed that Dropout adds Noise to the Network, as we loose out on some Information, and the Network is forced to learn with missing Information.

13 Basics of Convolutions

NOTE: The following Introcution to Groups is brought to you by chatgpt and the slides as i am too lazy to Tex this.

Let G be a set and $\cdot : G \times G \to G$ a binary operation. Then $(G, \cdot)$ is a group if:

1. There exists an identity element $e \in G$ such that $\forall a \in G : e \cdot a = a \cdot e = a$.
2. Every element $a \in G$ has an inverse element $a^{-1} \in G$ such that $a \cdot a^{-1} = a^{-1} \cdot a = e$.
3. Associativity holds: $\forall a, b, c \in G : (a \cdot b) \cdot c = a \cdot (b \cdot c)$.

Group Operation

Let $(G, \cdot)$ be a group and X a set. Then $\triangleright : G \times X \to X$ is a (left) group operation if:

1. Identity: $\forall x \in X : e \triangleright x = x$, where e is the identity element of G.
2. Compatibility: $\forall a, b \in G, x \in X : (a \cdot b) \triangleright x = a \triangleright (b \triangleright x)$.

Examples: Group Operations

Transformations can be elegantly described using group operations. Let $X = [0, 1]^{n \times n}$ be the space of images of size $n \times n$.

1. $G = \{\mathrm{Id}, \mathrm{Ref}_y\}$ forms a group containing the identity element (Id) and a reflection over the y-axis (Ref_y). That is, $\mathrm{Id} \triangleright x = x$ and $\mathrm{Ref}_y \triangleright x$ corresponds to the mirrored image of x.
2. $G = \{\mathrm{Id}, \mathrm{Rot}_{90}, \mathrm{Rot}_{180}, \mathrm{Rot}_{270}\}$ forms a group containing all 90° rotations of the image. What are the inverse elements of Rot_{90}, Rot_{180}, and Rot_{270}?

3. $G = \{\text{Rot}_\theta : \theta \in \mathbb{R}\}$ forms a group containing all rotations. What problems could arise for a discrete and n-limited image space?

4. $G = \{T(x, y) : x, y \in \mathbb{R}\}$ forms a group containing all translations by x, y. What problems could arise for a discrete and n-limited image space?

5. $G = S(M)$ is the group of all permutations of a set M. This group plays an important role, for example, in point cloud data.

Equivariance and Invariance

Let $f : X \to Y$ be a mapping (for example, a neural network), $(G, \cdot)$ a group, and $\triangleright : G \times X \to X$ and $\square : G \times Y \to Y$ group operations on X and Y respectively.

f is called equivariant with respect to G if:

$$\forall g \in G, x \in X : f(g \triangleright x) = g \square f(x)$$

f is called invariant with respect to G if:

$$\forall g \in G, x \in X : f(g \triangleright x) = f(x)$$

13.1 Convolution/Cross Correlation

Ok now after the gpt Introduction to Groups, lets talk about Convolutions and Cross Correlations.

For this I recommend the 3B1B Video on Convolutions, but lets try to explain it here as well. To quote the Video "The formulaic definition ... can look quite Intemidating".

But lets start slow and easy, imagine we have two lists of numbers (1,2,3) (4,5,6) and we can want to calculate the Convolution here. We would flip around list 2. So we would have (6,5,4) and then we would slide it over list 1. We would then multiply the corresponding elements and sum them up. So we have $(4 \cdot 1, 1 \cdot 5 + 2 \cdot 4, 1 \cdot 6 + 2 \cdot 5 + 3 \cdot 4, 2 \cdot 6 + 3 \cdot 5, 3 \cdot 6) = (4, 13, 28, 27, 18)$. This is the Convolution of the two lists. Ok now lets look at the Formula. The convolution of a function $x(t)$ with a function $y(t)$ is defined as:

$$(x * y)(t) = \int_{-\infty}^{\infty} x(\tau)y(t - \tau)\, d\tau = \int_{-\infty}^{\infty} x(t - \tau)y(\tau)\, d\tau$$

Now the τ is basicallty the flip to our list 2, just to give some intuition to the integral, considering our example the last equivalence in the definition makes sense as well, we could also reverse the first list and slide it over the second list, to achieve the same result. Here we can introduce the cross correlation, which is defined as:

$$(x \star y)(t) = \int_{-\infty}^{\infty} x(\tau)y(t + \tau)\, d\tau = \int_{-\infty}^{\infty} x(\tau - t)y(\tau)\, d\tau = (x(-\tau) * y(\tau))(t)$$

This would be like multiplying the first list with the second list, but not flipping the second list, which would be the same as flipping the first list and sliding it

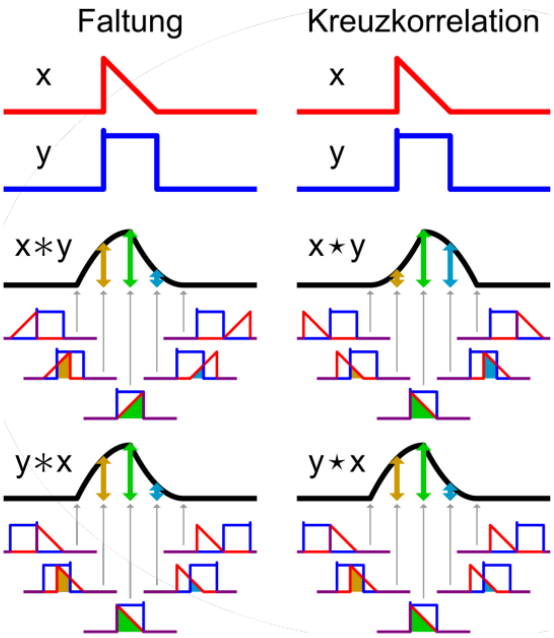


Figure 11: Convolution

over the second list. Ok lets visualize this with an example from the Slides. Ok lets break down Convolution first, lets take the function y and flip like we have done to the list, and then we slide it over function x. Except the script image is explaining it with the second equivalence thus we flip x and slide it over y. The peak of the new function is basically in our example of the lists, where both lists are multiplied and added together. So imagine the first function coming closer to y and then we start multiplying the corresponding components, so the value begins to increase and reaches its peak when both function overlap completly and then it decreases again. Lets have a closer look at the Slope of the Convolution here. The Slope is rather steep at the start and then decreases again, this is because the start of function y is steeper (imagine a larger number in the list) decreasing it is not as steep, since the numbers are smalle for y. Now lets have a look at cross correlation, we just slide x over y, so the slope is not as steep at the start because the smaller part of x slides over y first. To put all of this into the two lists context, we can imagine the list y being (6,5,4) so the Bump at the start is our largest value at the start, try to think about this and watch the 3B1B Video. This example makes everyhting more intuitiv for me personally, how ever after reading this I do see if the explanation is a bit messy and all over the place, sorry for that.

Important Properties of and Cross Correlation

- **Commutative:** $x * y = y * x$
- **Associative:** $x * (y * z) = (x * y) * z$
- **Distributive:** $x * (y + z) = x * y + x * z$

- **Convolution Theorem:** $\mathcal{F}(x * y) = (2\pi)^{n/2}\mathcal{F}(x) \cdot \mathcal{F}(y)$, where

$$\mathcal{F}(y)(f) = (2\pi)^{-n/2} \int_{\mathbb{R}^n} y(t)e^{-ift}\, dt$$

  is the Fourier transform.

- Equivariant with respect to translations of x and y.

Important Properties of Cross-Correlation

- **Distributive:** $x \star (y + z) = x \star y + x \star z$
- Equivariant with respect to translations of y.

13.2 Discrete Convolution/Cross Correlation

With all my brambling about the lists, discrete Convolution should be rather intuitiv, we just sum up the products of the corresponding elements. The Discrete Convolution is defined as:

$$(x * y)[n] = \sum_{k \in \mathbb{Z}^n} x[k]y[n-k] = \sum_{k \in \mathbb{Z}^n} x[n-k]y[k]$$

The Discrete Cross Correlation is defined as:

$$(x \star y)[n] = \sum_{k \in \mathbb{Z}^n} x[k]y[n+k] = \sum_{k \in \mathbb{Z}^n} x[k-n]y[k]$$

Here if you go over the list and stay within the bounds of the list that k provides, you are even good to go to see this as the formula from the first example.

Ok but all of this kind of begs the question of why, why would we do that? In Visual Computing we often have images we refer here as a signal which is just a matrix of Pixel values(Probably a vector containing RGB or something), and we then want to apply a filter to the image this filter, which would be our function y is called a kernel. The Kernel is just a Small Matrix, which we slide over the image(yes like the 2 list of numbers) and then we apply the Convolution. (To get great visuals on this, i recommend the 3B1B Video on Convolutions).

14 Convolutional Layers

14.1 Convolutional Layers in 1D

We are going to use cross correlation now, lets say we have an Input Signal y, with the Kernel x(which is as mentioned before just a matrix) we are going to calculate the cross correlation $s = x \star y$. After calculating the cross correlation we apply an Activation Function to the result. This is the basic idea of a Convolutional Layer. It is also important to mention that we use weight sharing meaning we apply the same Kernels to every for example part of the image. This reduces the amount of Parameters when compared to fully connected layers. Also the connection are less, because we only apply our Kernels to a small part of the image. This allows for the Sparsity discussed previously in connection to L1 Regularization. An example of this can be seen in the following figure from the Slides.

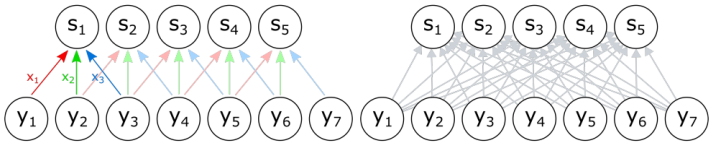


Figure 12: Convolutional Layer

14.2 Receptive Field

Again considering images as signals, we can think of the Receptive Field as the part of the image that the Kernel is applied to. The Receptive Field is the area of the input image that has an effect on the output of the layer. This makes sense since the pixels in this case close to each other holds for other examples as well though are usually correlated and thus it makes sense to consider the once close to each other. This can also be seen in this figure from the slides. Here g3 is effected by h2, h3, h4, which are in turn also always influenced by the

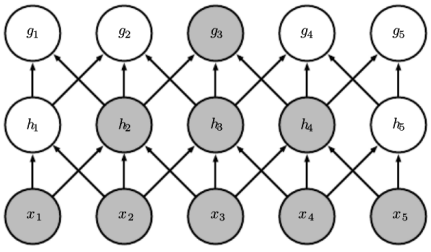


Figure 13: Receptive Field

previous layer. Which would be x1, x2, x3, x4, x5. This means the receptive field grows large in Multilayer Networks, even if the Kernel is small.

14.3 Dilation

Another way to make the Receptive Field rather large is Dilation. Again in the context of pixels and images(imho the most intuitiv). In a traditional convolution, a filter slides over the input image, and at each position, it computes the dot product between the filter weights and the image pixels. The dilation introduces gaps between the filter weights. This means that the filter will skip over certain pixels in the input, effectively increasing the distance between the pixels it considers. This means that we can maybe get a broader context.

14.4 Padding

Padding also makes sense lets again as always consider a Grid of Pixels, if we apply a Kernel of say size 3. If we apply the Kernel to the first Pixel of the image on a side we would loose that pixel, this would also happen if our Kernel reaches the last pixel of the image. This means that we would loose Every Pixel in the left pixel row, as well as the right, making our output smaller or less Dimensional. But this is not what we want we want to keep the Dimensionality of the Image, this is where Padding comes into play. Padding is just adding zeros around the image, so that the Kernel can be applied to every Pixel as we would basically just loose the zeros, instead of ours actual Pixels. This would then be called 0-Padding. There are also other types of Padding, such as Mirroring the Signal at the Edges or Peridodic Padding.(Not explained further on the slides, so I wont do it either.)

14.5 Stride

We can reduce the Dimensionality of the Output by using a Stride. The Stride is the amount by which the Kernel is moved over the Image. If we have a Stride of 2, the Kernel is moved by 2 Pixels. This also means that the Receptive Field Grows as our "context" grows larger. If we reduce Dimensionality it is also called Downsampling.

14.6 Pooling

In Pooling we can look at neighbouring Pixels and take an average or the maximum value. This also lets us assess context better, after we have done that we can for example do a stride operation again.

14.7 Example Max Pooling

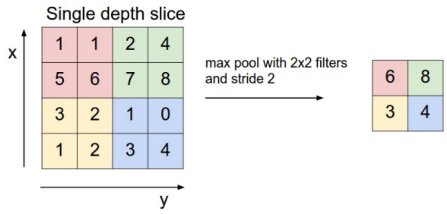


Figure 14: Max Pooling

This is an example of Max Pooling, here we just divide our image into 2x2 squares and take the maximum value of the 4 Pixels, since we use stride 2 there are no overlaps. This reduces the Dimensionality as can be seen in the output.

14.8 Convolutional Layers with Several Channels

Ok this is where the Mind Fuck starts a bit. Lets talk about images. Our image has its set of pixels now and we always naively assumed the Pixels to have a certain value. But usually they might have multiple Values, for example RGB etc. This poses a Problem though, how we do we compute those with our Kernel Matrix. The answer is we dont we need a bit of a more sophisticated approach. So lets introduce 4 Dimensional Tensors.

This figure depicts a 4D Tensor, lets break down what we need the Dimensions for, so first of all like before we need height and width of our Matrix. So we are in 2D Now, now all of those need to be applied to red green and blue, so now we are in 3D. Now we want to apply m filters for example to detect things such as features like edges, textures, or colors, so for all of those we need different blocks of the 3D filters I just described. This is why we need the 4th Dimension.

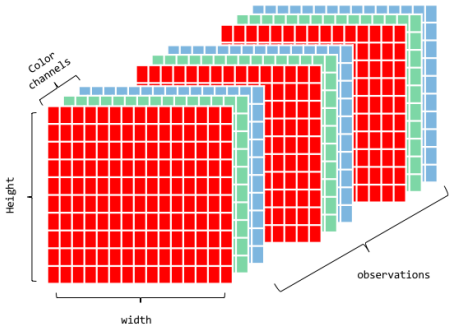


Figure 15: 4D Tensor

14.9 Transpoed Convolution Layer

This is the same process we have been discussing, for Convolution Layers, but in reverse. All the Concepts we learned can be applied to those as well in reverse. This is usefull for Decoders and Genrative Models.

15 Batch Normalization

We often have the issue that the input to the Activation Function is not centered around 0, this can lead to exploding or vanishing gradients. Now we learn about Batch Normalization in order to prevent that Problem. Batch Normalization can be applied to the Input but also the hidden layers. Ok if we take a look at the equation for Batch Normalization we see that we subtract the mean and divide by the standard deviation. from a Matrix H which contains the activations for a Mini Batch. So $H' = \frac{H-\mu}{\sigma}$ So what does that do, well substracting the mean we center the activations around 0, and dividing by the standard deviation we scale the activations to have a standard deviation of 1. Forcing the activations to have a mean of 0 and a standard deviation of 1. It should be noted that the standard deviation also adds a really small number to prevent divison by 0.

15.1 Why is this usefull?

If we consider activation Functions, such as the Sigmoid Function, if we input a very large number for exampe we get to a point where the function is almost flat, this means that the gradient is almost 0. This of course is problematic, thus normalizing makes perfect sense. In the Context of Inference, we need to keep track of a running average if the mean and standard deviation, so we can apply some kind of normalization to the test data.

15.2 How we do it in Practice

To my understanding this means that the Data might be more complex for us to adjust leave it normalized around zero, so we need to make it learn where it wants to be. This is done by introducing two new parameters, γ and β. γ is used to scale the normalized value and β is used to shift the normalized value. Both of those are learned during training, we can always leave out the Bias, since β is pretty much the same but a bit more flexible. So now we calculate $\gamma \cdot H' + \beta$

16 Convolutional Neural Networks Architectures

TODO if someones to list them all here, too lazy for that, would be a bit of a prank if this comes in the exam.

17 CNN visualization

Probably makes more sense to read the slides, a lot of images. Would be weird to quizz on this, as it was never remotely part of the tasks or the practicals, as far as i can remember.

18 Transfer Learning

We can do transfer learning if we for example have a classifier for cats, and now want to classify dogs. We cann use features of the cat network, to classify dogs. We could for example initialize the weights of the dog network with the weights of the cat network, this would probably give us a good starting point for gradient descent.

19 Autoencoders

An autoencoder is an architecture for neural networks designed to efficiently compress input data, reducing it to its essential features, and then reconstruct the original input from this compressed representation. Compression happens as the encoder takes the input data and transforms it into a usually lower-dimensional representation. The reconstruction occurs in the decoder, which attempts to restore the original input data.

The name was chosen because the model learns to automatically and independently encode and then decode the data. As this is just another Feed-Forward-Network, we can use all the techniques we have learned so far. This means we can train it using SGD and Backpropagation. Since we want the output to be close to the input we can minimize that giving us

$$L(x, g(f(x)))$$

where x is the input and $g(f(x))$ is the output. The error function can for example use the L1 or L2 loss function. $g(f(x))$ will be referred as the rekonstruction r moving forward.

19.1 Goal and Usecases

When we train an autoencoder, often the output itself is not what we are interested in, but rather the hidden representations, which are often the compressed lower dimensional version of the input data. This can be used for example for dimensionality reduction, or for example for denoising since it can learn to focus on the underlying structure of the data. It is also usefull for Self Supervised Learning ad Generative Models.(Not explained further on the slides, so I wont do it either.)

19.2 Undercomplete, Complete and Overcomplete Autoencoders

An undercomplete autoencoder is one where the hidden layer has fewer neurons than the input layer. This forces the autoencoder to learn a compressed representation of the data. A complete autoencoder is one where the hidden layer has the same number of neurons as the input layer. This means that the autoencoder can learn to store the input data in the hidden layer. An overcomplete autoencoder is one where the hidden layer has more neurons than the input layer. This allows the autoencoder to learn multiple ways to represent the input data.(To not get a perfect reconstruction, we would need to introduce some kind of regularization, so no trivial solution is found.)

20 Information Retrieval

20.1 Autoencoder vs PCA

PCA is Principal Component Analysis, and what is basically does is it checks for correlated features and then tries to find a new set of uncorrelated features, thus reducing Dimensionality. We do this by finding the Principal Components which are the directions in which the Data varies the most. The biggest difference is that Autoencoders can learn none linear representations , while PCA can only learn linear ones. We have to be carefull not giving the Autoencoder too much capacity, since it could just learn the Identity Function, effectivly copying the input to the output. Due to the universal approximation theorem, the Autoencoder can also learn noise and irrelevant features, which would overfit the data. If it doesnt have enough Capacity it wont have enough flexibility to learn the underlying structure of the data.

20.2 Information Retrieval

Information Retrieval has the goal of finding entries in a Database, which are relevant to a query. This can be implemented efficiently in lower dimensional space priovided by the Autoencoder. We can force our hidden Code to be binary code, which would make it easier to compare the entries in the Database. We could save it in a Hash-Table where the Code-Vectors are the Keys and the Entries are the Values. We could then Flip a Bit in the Query and then compare the Code-Vectors to find the most similar entries. This is called Semantic Hashing.

21 Regularization in Autoencoders

Ideally we want our Autoencoder to choose a Dimensionality for the Data, which represents the complexity of the Data. So for Autoencoders we can also introduce Regularization. Besides not wanting to copy the Data, we also want to force Sparsity of the Representation and make the Model robust to Noise. A well regularized Autoencoder should have bigger capacity, while also being able to learn the underlying structure of the Data.

21.1 Sparse Autoencoders

A sparse Autoencoder adds a Regularization term to the loss function,

$$L(x, r) + \Omega(h)$$

. This is commonly used to learn features for other tasks such as Classification. It has to be noted that this is not equivalent to the MAP Estimation. It can be seen as a preference for certain Functions. If we take a look we see some similarities to a generative Model with latent variables. Ok lets go through why that is. Given a model with visible variables x and latent variables h, the joint probability distribution is:

$$p_{\text{model}}(x, h) = p_{\text{model}}(x|h) \cdot p_{\text{model}}(h)$$

The log-likelihood of the observed data x can be found by marginalizing over the latent variables h:

$$\log p_{\text{model}}(x) = \log \sum_h p_{\text{model}}(x, h)$$

The autoencoder can be interpreted as an approximation by considering only the most probable value of h that maximizes:

$$\log p_{\text{model}}(x, h) = \underbrace{\log p_{\text{model}}(x|h)}_{-L(x,r)} + \underbrace{\log p_{\text{model}}(h)}_{-\Omega(h)}$$

The term $\log p_{\text{model}}(h)$ can for example be defined as the Laplace Prior.

21.2 Depth

As we just have a Feed-Forward-Network, we can just add more layers making the network more powerfull.

21.3 Denoising Autoencoders

We can build a Denoising Autoencoder by comparing the Loss with r. But this might lead to issues, since the Autoencoder (with enough capacity) will find the Identity Function. So we can add Noise to the Input, and then compare the Output to the original Input. This is called Denoising Autoencoder. So the loss $L(x, r)$ is now defined with $r = g(f(\hat{x}))$ with $\hat{x}$ being the Noisy Input.

21.4 Network Initialization with stacked denoising Autoencoders

We can use denoising Autoencoders to pre train deep networks. The deep network training can be subdivided into Multiple steps.

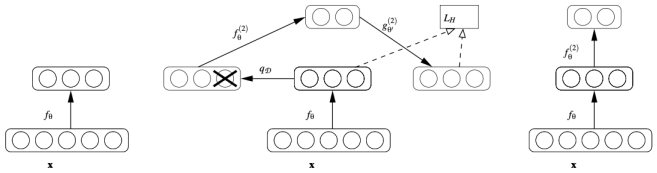


Figure 16: Stacked Autoencoder

As visible in the image we keep on Stacking the Autoencoders. And after each step calculate the loss, and then Stack again. At the last Autoencoder we train the layer(which is a decoder) supervised and then we can fine tune the whole network by training the whole network supervised. (NOTE: I am really unsure here)

22 Variational Autoencoders and Stochastic Autoencoders

22.1 Discrimative vs Generative Models

- Discriminative Models: Learn the conditional probability distribution $Y^* = \arg\max_Y p(y|x)$ directly.
- Generative Models: A generative method models the conditional probabilities for the classes $P(X|Y)$ and uses Bayes' theorem together with the prior $P(Y)$ to find the posterior $P(Y|X)$.

  The term generative refers to the fact that the prior $P(Y)$ and the likelihood $P(X|Y)$ can be used to generate new data by sampling from $P(X) = \sum_Y P(X|Y) \cdot P(Y)$.

  The class labels are determined by:

$$Y^* = \arg\max_Y P(Y|X) = \arg\max_Y \frac{P(X|Y) \cdot P(Y)}{P(X)}$$

  Since $P(X)$ is constant for all classes, this simplifies to:

$$Y^* = \arg\max_Y P(X|Y) \cdot P(Y)$$

23 Stochastic Autoencoders

Thus far we have only considered deterministic Autoencoders, but we can also introduce probabilistic Functions. We could say that we now have $p_{encoder}(h|x)$ and $p_{decoder}(x|h)$.

23.1 Stochastic Encoder and Decoders

Since we are using Neural Nets as before our strategy doesnt really change, the only difference in Terms of the Loss is that, x is not just our input, but also the target of the output. This is why we get the Decoder which tries to reconstruct the input given the hidden code and thus we need to minimize negative log likelihood of this reconstruction $-\log p_{decoder}(x|h)$. Regarding the output, if x follows a normal distribution, we can minimize the mean squared error, because the NN needs to predict the mean of the distribution. If x follows a Bernoulli distribution, which has 2 outcomes, Sigmoid makes sense, as you might remember it is just a value between 0 and 1. This is equivalent to minimizing the binary cross-entropy loss, which measures the difference between 2 probability distributions. And then we also have the Case of Multiple Class Labels, in which case we use Softmax, because the sum of all probabilities is 1. We again can minimize the cross entropy loss.

The likelihood L of the observed data given the predicted probabilities is:

$$L = \prod_{i=1}^{N} p_i^{y_i} (1 - p_i)^{1-y_i}$$

The log-likelihood $\log L$ is:

$$\log L = \sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

The negative log-likelihood is then:

$$-\log L = -\sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

The cross-entropy $H(p, q)$ for binary classification is given by:

$$H(p, q) = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)]$$

When comparing this to the negative log-likelihood, we see that they differ only by the constant factor $\frac{1}{N}$, which represents the average over all instances. Since this constant factor does not affect the minimization process, minimizing the cross-entropy is equivalent to minimizing the negative log-likelihood.