

Summary of the Deep Learning Lecture at the University of Bonn

Tim Nogga

July 15, 2024

Contents

1	Introduction	3
1.1	Task T	3
1.2	Experience E	3
1.2.1	Supervised Learning	3
1.2.2	Unsupervised Learning	4
1.3	Performance Measure P	4
1.4	Training Error, Test Error and Generalization	4
1.5	Capacity	4
1.6	Capacity vs Error	4
1.7	Hyperparameters	5
1.8	Cross Validation	5
2	Recap Statistics	5
2.1	Terms and Definitions	5
2.2	Multivariate Statistics	5
2.3	Marginalization in a Discrete Case	5
2.4	Marginalization in a Continuous Case	6
2.5	Marginalization in a Mixed Case	6
2.6	Marginalization in Multiple Dimensions	6
2.7	Conditional Probability	6
2.8	Independence	6
2.9	independent and Identically Distributed	6
2.10	Bayes Rule	6
2.11	Expectation	7
2.12	Variance and Standard Deviation	7
2.13	Shannon Entropy	7
2.14	Cross Entropy	7
2.15	Kullback-Leibler Divergence	7
3	Estimators	8
3.1	Point Estimation	8
3.2	Bias	8
3.3	Mean Squared Error	8
3.4	Bias-Variance Tradeoff	8
3.5	Point Estimation and the Consistency of the Estimator	9

3.6	Maximum Likelihood Estimation	9
3.7	(Negative) Log Likelihood	9
3.8	Negative Log-Likelihood and Cross Entropy	9
3.9	Conditional Negative Log-Likelihood	10
3.10	Cross Entropy for Classification/Segmentation	10
3.11	Segmentation and Pixel Wise Classification	10
3.12	Mean Squared Error for Regression	11
3.13	Mean Absolute Error	11
3.14	Maximum A Posteriori Estimation	11
4	Error functions	11
4.1	Local and Global Minima	12
4.2	Convex Functions	13
4.3	Different approaches to find Minima	13
4.3.1	Least Squares	13
4.3.2	Grid Search	13
4.3.3	Evoluationary Algorithms	13
4.3.4	Gradient Based Optimization	14
5	Gradient Descent	14
5.1	Error Surface	14
5.2	Formal Definition of Gradient Descent	14
5.3	Gradient - Taking the Derivative in Multiple Dimensions	15
5.4	Direcional Derivative	15
5.5	Learning Rate ϵ	15
5.6	Jacobi Matrix	15
5.7	The Derivative of second order	16
5.8	Hessian Matrix	16
5.9	Approximate Second-Order Methods	16
6	Stochastic Gradient Descent	16
6.1	Mini-Batch Gradient as an unbiased Estimator	17
6.2	Batch Size	17
6.3	Algorithm for Stochastic Gradient Descent	18
6.4	Learning Rate ϵ	18
6.5	Momentum	18
6.6	Algorithm for Stochastic Gradient Descent with Momentum	18
7	Adaptive Learning Rates	18
7.1	AdaGrad Algorithm	19
8	Feed-Forward Networks	20
8.1	Single-Neuron	20
8.2	One Layer and multiple Neurons	20
8.3	Multiple Layers	20
8.4	Fun Facts for Linear Activation	21
8.5	Activation Functions	21
8.6	Symmetries in Parameter Space	22
8.7	Universal Approximation Theorem	22

9	Backpropagation	22
9.1	Why do we need Backpropagation?	22
9.2	Lets look at the Chain Rule	22
9.3	Direcional Derivative with the Chain Rule	23
9.4	Backpropagation Algorithm Intuitively	24
9.5	Backpropagation Algorithm Formally	24
9.6	TODO explain this with jacobi matrices (i think before should work to understand the concept though)	27
10	Regularization	27
10.1	Gradient Clipping	27
10.2	Parameter Initialization	27
10.2.1	Very Small Weights	27
10.2.2	Very Large Weights	28
10.3	Scaling the Input Values	28
10.4	Weight Initialization	28
10.5	Parameter Initialization	28
10.6	Scaling of the Weights as an Hyperparameter	29
10.7	Overfitting and Underfitting quick comeback in Context	29
10.8	Definition of Regularization	29
10.9	Regularization in Deep Learning	29
10.10	Parameter Norm Penalties	29
10.11	L2 Regularization	30
10.11.1	TODO There is an intuition slide in the slides, which i cant explain well	30
10.12	L1 Regularization	30
10.13	Sparse Networks	30
10.14	Representation Sparsity	31
10.15	Regularization as Maximum A Posteriori Estimation	31
11	Reduction of Modelcapacity	31
11.1	Early Stopping	31

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 Klassifikation, 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 Performance Measure is calculated on a Test Set, which is not used during the Training Process. So there 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 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

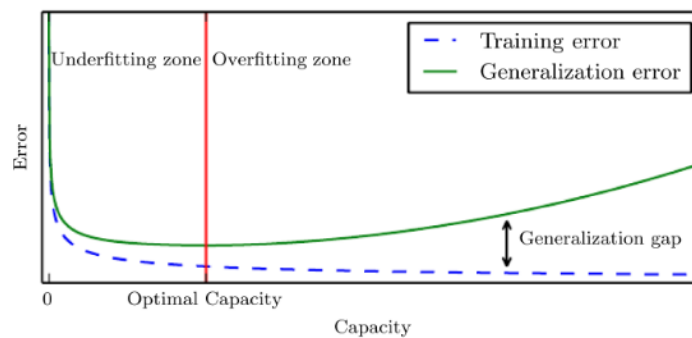


Figure 1:

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.

1.7 Hyperparameters

Learning Algorithms usually have Hyperparameters which are parameters not learned during Training, but set before. These could be things such as Model Capacity and Learning Rate. This poses the Question whether 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** (Ω, \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** $\times p : \Omega \rightarrow [0, 1]$ with $\sum_{\omega \in \Omega} p(\omega) = 1$.
- **Probability Density Function if Ω is Continuous** $\times p : \Omega \rightarrow [0, \infty)$ 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 $p(x, y) = \sum_w \int p(x, y, z, w) dz dw$.

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 $p(x_1, x_2, \dots, x_n) = p(x_1)p(x_2)\dots p(x_n)$.

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 back-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 a 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.

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

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

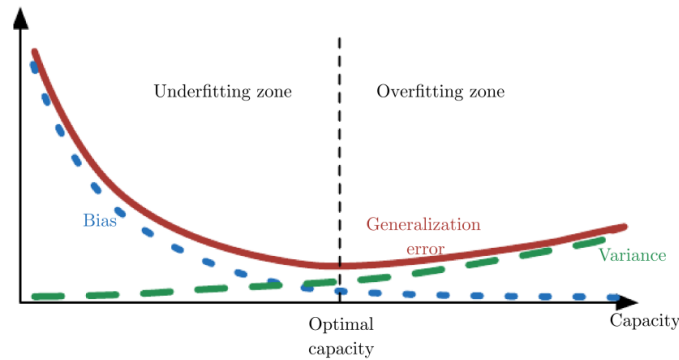


Figure 2:

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 Estimation and the Consistency 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 $\lim_{m \rightarrow \infty} P(|\hat{\theta}_m - \theta| > \epsilon) = 0$.

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 proof what so ever we will assume, that the Maximum Likelihood Estimation is consistent, when the true distribution 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

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

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 $m \rightarrow \infty$ approximates the cross-entropy between the modeled distribution $p_{model}(x|\theta)$ and the true data-generating distribution $p_{data}(x)$:

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

As $m \rightarrow \infty$, 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 $\arg \min_{\theta} \sum_{i=1}^m -\log p_{\text{model}}(y_i|x_i, \theta)$. This is particularly useful, because it can be used in Classification, Segmentation, among many other Tasks, where we want to Interpret 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

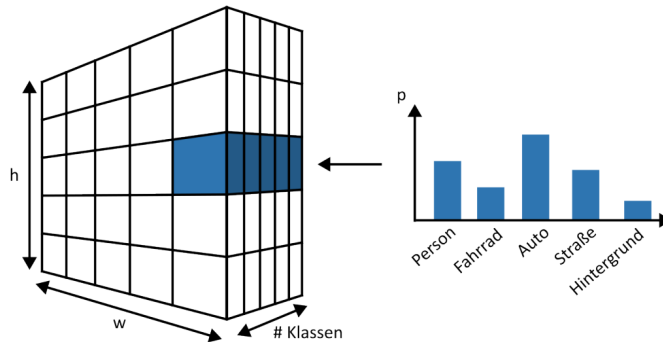


Figure 3:

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 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 $\theta_{MAP} = \arg \max_\theta p(\theta|X)$. 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 incorporate a sum again thus follows $\hat{\theta}_{MAP} = \arg \max_\theta \sum_i \log p(x_i|\theta) + \log p(\theta)$. This 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 background 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 referred 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

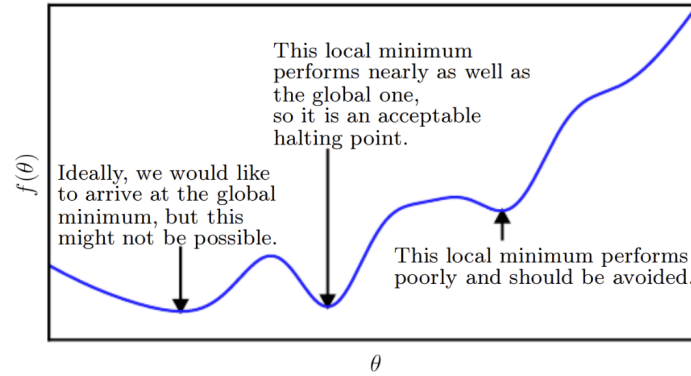


Figure 4:

As explained before the Local Minima is the one found by Gradient Descent. This might not be too bad if it is approximately 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 $\forall x, y \in D$ 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.

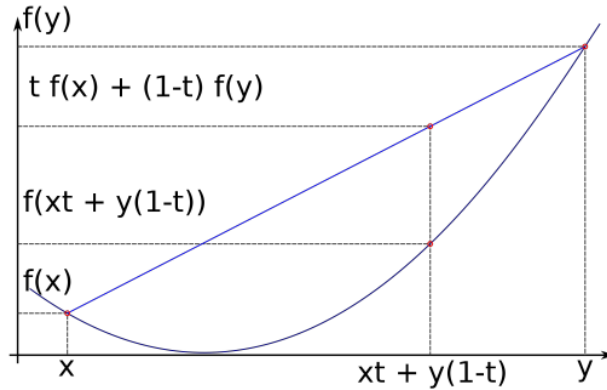


Figure 5:

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 Evolutionary 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 as well, as it is a brute force method. It works fine for 10-1000 Parameters but becomes inefficient for more. It can be useful 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

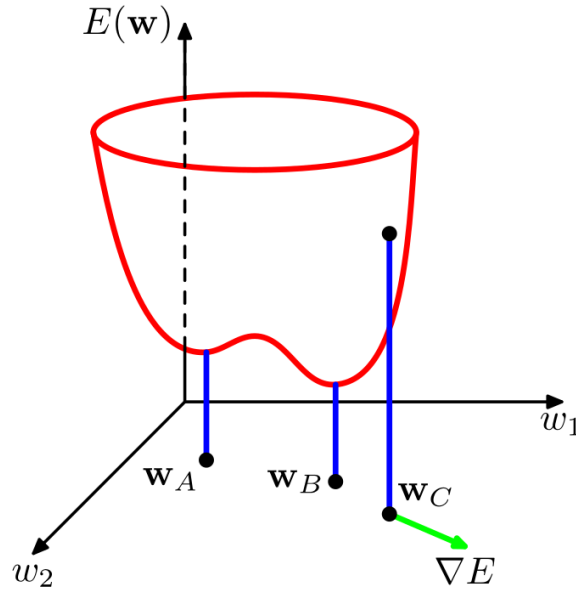


Figure 6:

This is a Geometric Representation of the Error 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 $\mathbb{R}^n \rightarrow \mathbb{R}$. 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 $\nabla_u f(\theta) = \lim_{\alpha \rightarrow 0} \frac{f(\theta + \alpha u) - f(\theta)}{\alpha}$ 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 oscilates between -1 and 1. Thus the minimum 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

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

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 between the smallest and the largest Eigenvalue of H , because it is a weighted sum, it must be constrained 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 disadvantages though, as the Hessian Matrix calculation as well as the inversion can be rather expensive for large disadvantages.

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 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 Variance 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 $\{x^{(1)}, \dots, x^{(m)}\}$
 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 constant factor until it reaches a certain iteration, where it 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.

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

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 $\{x^{(1)}, \dots, x^{(m)}\}$
 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 $\{x^{(1)}, \dots, x^{(m)}\}$
 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 Multiplication and the Division are applied element wise, the update is done for every parameter and so every parameter that was previously 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 described 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:

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

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 Neurons, those are fed with the same input and combined in one layer. The output of such a layer can be calculated by

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

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

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

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.

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

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

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

Combining weights into a single matrix W' :

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

Finally, we get:

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

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

Some of the Output Layer Activation Functions are:

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

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 calculate the Difference quotient being defined by

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

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 $\frac{\partial}{\partial x} g(f(x)) = \frac{\partial g}{\partial f} \frac{\partial f}{\partial x}$. 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

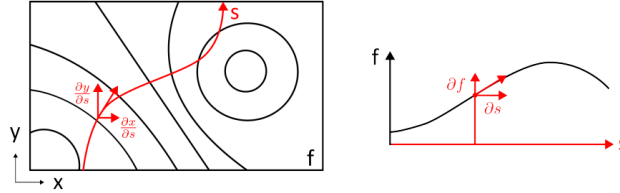
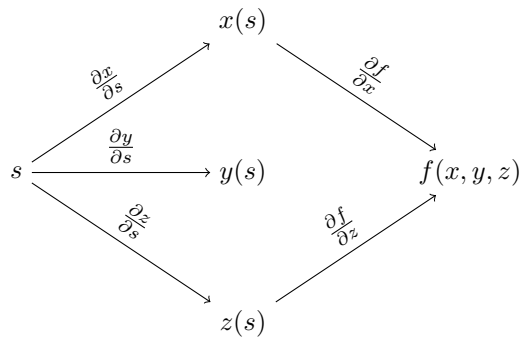


Figure 7:

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

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

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 classifying 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

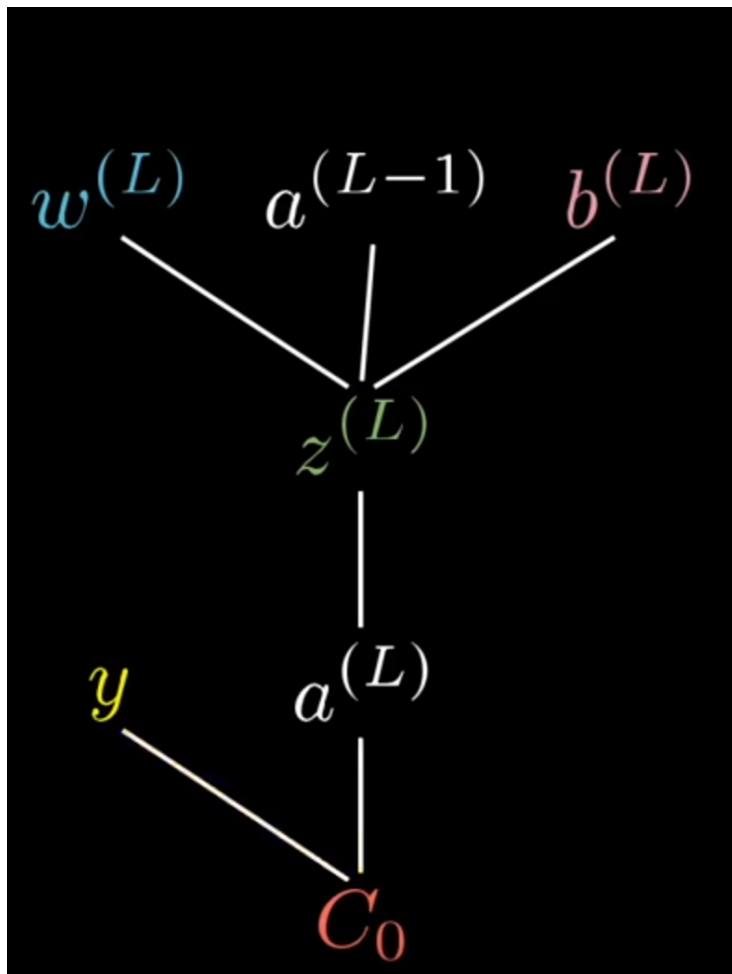


Figure 8: 3B1B Backpropagation

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!

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

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 how 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 successfully 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 obviously 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 index to keep track which neuron talking about lets call it activation of the k th neuron in the $L-1$ layer with $a_k^{(L-1)}$ thus the cost function with respect to the activation of the k th 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

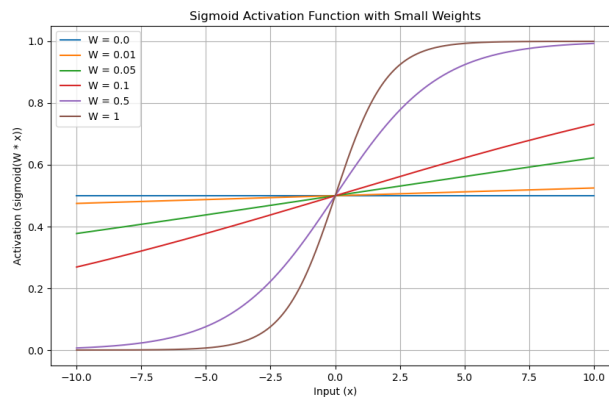
10.1 Gradient Clipping

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 exceeds 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 collapse 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.

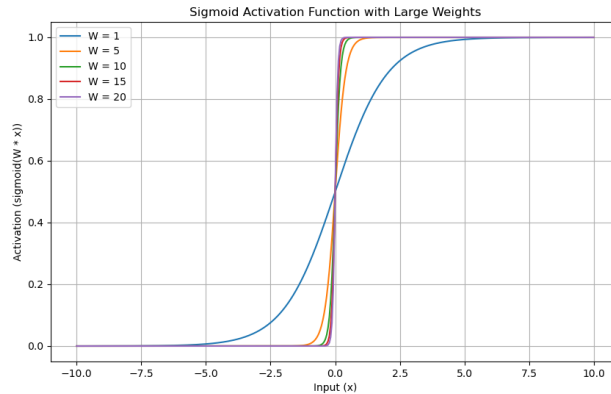


Figure 9: Sigmod with Large Weights

10.3 Scaling 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 activation 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 whether 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 volatile 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 Variance 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 Penalty 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 loss 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

$$\begin{array}{c} \begin{bmatrix} 18 \\ 5 \\ 15 \\ -9 \\ -3 \end{bmatrix} \\ \mathbf{y} \in \mathbb{R}^m \end{array} = \begin{array}{c} \begin{bmatrix} 4 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -1 & 0 & 3 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & -4 \\ 1 & 0 & 0 & 0 & -5 & 0 \end{bmatrix} \\ \mathbf{A} \in \mathbb{R}^{m \times n} \end{array} \begin{array}{c} \begin{bmatrix} 2 \\ 3 \\ -2 \\ -5 \\ 1 \\ 4 \end{bmatrix} \\ \mathbf{x} \in \mathbb{R}^n \end{array}
 \end{array}
 \quad
 \begin{array}{c} \begin{bmatrix} -14 \\ 1 \\ 19 \\ 2 \\ 23 \end{bmatrix} \\ \mathbf{y} \in \mathbb{R}^m \end{array} = \begin{array}{c} \begin{bmatrix} 3 & -1 & 2 & -5 & 4 & 1 \\ 4 & 2 & -3 & -1 & 1 & 3 \\ -1 & 5 & 4 & 2 & -3 & -2 \\ 3 & 1 & 2 & -3 & 0 & -3 \\ -5 & 4 & -2 & 2 & -5 & -1 \end{bmatrix} \\ \mathbf{B} \in \mathbb{R}^{m \times n} \end{array} \begin{array}{c} \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \\ -3 \\ 0 \end{bmatrix} \\ \mathbf{h} \in \mathbb{R}^n \end{array}$$

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 $1/\alpha$ 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.