University of Tübingen
Deep Learning Lecture Notes

Prof. Dr.-Ing. Andreas Geiger

Winter Term 2020/2021

Abstract

These lecture notes have been written collectively by the class of winter 2020/2021 and curated by the TAs of the deep learning lecture. If you find an error, please email the TA responsible for the section.

1 Introduction

1.1 Introduction

The introduction contained all organizational matters. The team was introduced and the contents, goal and organization of the course, the exercises, lecture notes, materials, credits and prerequisites were explained.

1.2 History of Deep Learning

Three waves of development
Deep Learning has been developed in three waves. The term "Deep Learning" didn't exist whet these ideas started to grow. During the first wave from 1940 to 1970, deep learning was called "Cybernetics". This wave was also called the "Golden Age", with a lot of great discoveries and following great hopes. At that time, people were using simple computational models of the brain to imitate biological learning and to learn simple rules. These rules could successfully classify simple patterns. But people realized, that these simple models were not powerful enough to solve complex tasks, which led to a decline in research.
During the second wave from 1980 to 2000, Deep Learning has been associated with "connectionism". In connectionism, it was assumed, that intelligent behavior was realized through a large number of simple units. Also backpropagation and sequence models have been developed. Despite being very important, these developments have been overshadowed by deployments in other research fields due to a lack of algorithms, ways of computing and data. This is why this period is also called the "Dark Age".
From 2006 until now, Deep Learning was actually called "Deep Learning". This can be called the "Revolution Age" where it was demonstrated, that with deeper networks, lager datasets and more compute, Deep Learning was leading to state-of-the-art results, dominating almost all leader boards across different fields and disciplines.

1943: McCullock and Pitts
McCullock and Pitts developed an early model for neural activation. It is called a linear threshold neuron because it is a threshold on a linear computation. It looks as follows:

$$f_{\mathbf{w}}(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}^T\mathbf{x} \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Here, x is the data and w are the weights. X could also be some features of the data, so we will call it features from now on. If you multiply there features with the weights and the product is bigger than zero, we assign +1 to that function. If it is smaller than 0, we assign -1. That's a binary decision, that's why it is called a linear threshold neuron. It has been demonstrated that his simple operation is more powerful that AND/OR gates, which are special cases of this computation. But at the time there was no procedure to effectively learn the weights. This has changed in 1958:

1958-1962: Rosenblatt's Perceptron
In 1958, Rosenblatt developed the famous perceptron, which was the first algorithm to train the single linear threshold neuron. He also developed a hardware implementation of this algorithm. He used the so-called

"perceptron algorithm", which was optimizing of perceptron criterion:

$$\mathcal{L}(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^T \mathbf{x}_n \, y_n$$

This is very different from the gradient based optimization with back propagation used today because the linear threshold unit is non-differentiable. Therefore, the auxiliary task of the perceptron criterion has been defined. The optimization of this criterion simply looks at all the results that have been wrongly labeled; so $\mathcal{M}$ is the set of wrongly labeled examples. It looks what the model does for those and what the true label is (+1 or -1) and it is minimizing a loss based on these incorrectly classified labels such that they are classified correctly in the next iteration of the algorithm. For this very simple model, this is an optimal thing to do because it converges to the right solution, if a solution exists, which was proven by Novikoff. It was a big success and there was also some resonance in the media, as for example a perceptron, that was trained to recognize the difference between males and females. While this provides a working example of such an algorithm, the perceptron was heavily overhyped: For instance, Rosenblatt claimed that the perceptron will lead to computers that walk, talk, see, write, reproduce themselves and are conscious of their own existence. So the hopes and expectations were really high and couldn't be fulfilled, which lead to mistrust in this technology.

1969: Minsky and Papert publish book
In 1969, Minsky and Papert published a book called "Perceptrons", which mathemalically showed several discouraging results for the model. For example, it showed that single-layer perceptrons cannot solve some very simple problems such as the XOR problem or counting. This led to less interest and funding in this area, as well as to the rise of symbolic AI research, which dominated the 70s.

1979: Fukushima's Neocognitron
In 1979, Fukushima proposed the neocognitron, which was a predecessor of models that are similar to convolutional neural network models that are still used today. The neocognitron was inspired by Hubel and Wiesels experiments in the 1950s. They studied the visual cortex V1 in cats and they found out that there are different types of neural cells in the V1, which are sensitive to orientation of edges but insensitive to their position. The so-called simple cells respond primarily to edges while the complex cells imlepement spatial invariance. Hubel and Wiesel received the Nobel price in 1981.
Fukushima proposed a computational model that mimics these simple and complex cells. It is a multi-layer processing network to create intelligent behavior. It is composed of simple (S) and complex (C) cells which implement convolution and pooling. However, there was no backpropagation algorithm at that time, so learning happened, using reinforcement based learning. But still, the model architecture was an inspiration for modern CNNs.

1986: Backpropagation Algorithm
In 1986, the backpropagation algorithm was reinvented by Rumelhart, Hinton and Williams. It was known since 1961, but had its first empirical success in 1986 and remains the main workhorse in deep learning today. It allows the efficient calculation of gradients in a deep neural network with respect to network weights by very efficiently updating the weights gradients. It enables application of gradient based learning to deep networks. This was a major breakthrough.

1997: Long Short-Term Memory
In 1991, Hochreiter demonstrated the problem of vanishing and exploding gradients in his Diploma Thesis. This led to the development of long-short term memory for sequence modeling. It uses a combination of feedback loops and forget or keep gates to effectively bridge this gradient flow over very long time horizons. This has revolutionized natural language processing (NLP). But only from 2015 on, it became very popular.

1998: Convolutional Neural Networks
In 1998, convolutional neural networks (CNN) were proposed, which are similar to the Neocognitron, but are trained end-to-end using backpropagation. It implements spatial invariance via convolutions and max-pooling and uses weight sharing to reduce the parameter space. It was demonstrated, that a CNN can lead to very good results in standard machine learning tasks such as the MNIST digit classification task or Tanh/Softmax activations. However, the results did not scale up (yet) to the complexity of computer vision problems.

2009-2012: ImageNet and AlexNet
Another major breakthrough was the demonstration of ImageNet and AlexNet. ImgageNet is a huge dataset and a recognition benchmark called the "Image Net Large Scale Visual Recognition Challenge" (ILSVRC) which features 10 million annotated images out of 1000 categories. This was the first time that such a huge amount

of data became available. AlexNet was the first neural network to win the ILSVRC via GPU training, deep models, better neural network architectures and huge datasets. This was the sparking point of the deep learning revolution, where people recognized that deep learning can really change the world.

2012-now: Golden Age of Datasets
From 2012 on, realized that big annotated datasets are a key to solving complex challenges, and so there was a large number of datasets developed subsequently, such as the KITTI dataset for self-driving in cityscapes. There are lots of datasets for various tasks, e.g. PASCAL and MS COCO for Recognition, ShapeNet and ScanNet for 3D Deep Learning, GLUE for Language understanding, Visual Genome for Vision and Language modeling, VisualQA for Question Answering and MITOS for the recognition of Breast cancer tissue.

2012-now: Synthetic Data
Annotating real data is expensive, which led to the surge of synthetic datasets. Creating 3D assets is also costly, but even very simple 3D datasets proved tremendously useful for pre-training deep neural models in particular e.g. in the case of optical flow.

2014: Generalization
This lead to the success story of deep learning in terms of generalization. It could be empirically demonstrated that deep representations generalize well despite a large number of parameters. A CNN can be pre-trained on large amounts of data on a generic task (e.g., ImageNet classification) and then only the last layers need to be fine-tuned (re-trained) on few data of a new task and still perform very well.

2014: Visualization
Zeiler and Fergus published a paper on visualization of that a deep neural network learns. The goal of this paper was to provide insights into what the network has learned, because the networks are like black boxes. They visualized image regions that most strongly activate various neurons at different layers of the network. And they found that higher levels capture more abstract semantic information.

2014: Adversarial Examples
Contrary to all these findings, in 2014 it was also demonstrated that accurate image classifiers can be easily fooled by imperceptible changes, using what is called the "Adversarial example":

$$x + \underset{\Delta x}{\operatorname{argmin}} \{\|\Delta x\|_2 : f(x + \Delta x) \neq f(x)\}$$

After applying this change to an image, the network recognizes all images as being classified as "ostrich".

2014: Domination of Deep Learning
From 2014 on, Deep Learning has developed really fast and deep learning has dominated many research fields. In 2014, there have been the first successfull deep models for machine translation e.g. Seq2Seq. Deep generative models (like VAEs and GANs) also started to produce compelling images, e.g. novel images of human faces. Major breakthroughs can also be observed e.g. in the prediction of molecular properties for generating novel materials, which have been revolutionized using Graph Neural Networks (GNNs). In summary, there have been dramatic gains in vision and speech which refers to Moore's Law of AI.

2015: Deep Reinforcement Learning
In 2015, DeepMind demonstrated that it is possible to learn a policy (a state→action mapping) through random exploration and reward signals (e.g., game score). This is called reinforcement learning, and this can be enriched by deep learning. It can play successfully a variety of Atari games without other supervision. However, some games remain hard, especially games that require reasoning, remembering or thinking long ahead.

2016: WaveNet
In 2016 it was demonstrated that deep generative models of raw audio waveforms like WaveNet can generate speech which mimics human voice, as well as music.

2016: Style Transfer
Networks trained on datasets such as ImageNet can learn powerful features that allow e.g. to manipulate photographs by adopting the style of a another image or painting. It uses a deep neural network pre-trained on ImageNet for disentangling the content from style. You can try it yourself on https://deepart.io/

2016: AlphaGo defeats Lee Sedol
In 2016, DeepMind developed AlphaGo, which combines deep learning with the Monte Carlo tree search. It

was the first computer program to defeat a professional Go player. AlphaZero, developed in 2017 even learns via self-play and masters multiple games.

2017: Mask R-CNN
Mask R-CNN is a deep neural network for joint object detection and instance segmentation which outputs "structured object", an entire pixel map and a label map, not only a single number or class label.

2017-2018: Transformers and BERT
Transformers demonstrated that attention can effectively replace recurrence and convolutions in neural networks. BERT showed that pre-training of language models on unlabeled text can be very effective. Once these model are fine-tuned, state-of-the-art results on very challenging tasks such as the GLUE benchmark can be obtained. On GLUE, algorithms achieve superhuman performance on some language understanding tasks such as paraphrasing and question answering. However, computers still fail in dialogue. It is easy to make these systems fail and make them not pass the Turing Test.

2018: Turing Award
In 2018, the "nobel price of computing" has been awarded to the "founding fathers of deep learning", Yoshua Bengio, Geoffrey Hinton and Yann LeCun.

2016-2020: 3D Deep Learning
From 2016 on, in the computer vision domain, first models to successfully output 3D representations were developed. They could effectively predict voxels, point clouds, meshes and implicit representations. Prediction of 3D models became even possible from a single 2D image. The models have been extended to properties such as geometry, materials, light and objects in motion.

2020: GPT-3
This year, GPT-3 came along, which is the first version of the language model by OpenAI. It is upscaling existing language models to 175 Billion parameters.It has a text-in / text-out interface and many use cases like coding, poetry, blogging, news articles and chatbots. There are also controversial discussions. It has been licensed exclusively to Microsoft on September 22, 2020.

Current Challenges
There remain still some challenges for the next generation, such as un- or self-supervised learning, interactive learning, accuracy (e.g. for self-driving), robustness and generalization, inductive biases, understanding and mathematics, memory and compute and, last but not least ethics and legal questions. And it also remains open, whether "Moore's Law of AI" will continue.

1.3 Machine Learning Basics

These basics are meant as a recap, but if things are going too fast of if you haven't seen any machine learning lecture before, you can have a look into Goodfellow et al.: Deep Learning, Chapter 5 (http://www.deeplearningbook.org/contents/ml.html).

1.3.1 Learning Problems

Supervised learning
In supervised learning, model parameters are learned using a dataset of data-label pairs $\{(x_i, y_i)\}_{i=1}^N$. Here x is the input and y is the output and we have N such pairs. Examples include classification, regression and structured prediction problems.

Unsupervised learning
Model parameters are learned using a dataset without labels $\{x_i\}_{i=1}^N$, so just with inputs. Examples include clustering, dimensionality reduction and generative models.

Self-supervised learning
Model parameters are learned using a dataset of data-data pairs $\{(x_i, x_i')\}_{i=1}^N$. Examples include self-supervised stereo and optical flow estimation and contrastive learning.

Reinforcement learning
Model parameters are learned using active exploration from sparse rewards instead of a fixed dataset. Examples

include deep q learning, gradient policy and actor critique.

In this lecture, we will cover the first three learning problems.

1.3.2 Supervised Learning

Classification / Regression:
In classification, the goal is to predict a discrete class label using a function f with input $\mathcal{X}$:

$$f : \mathcal{X} \to \mathbb{N}$$

In regression, we are interested in predicting from an arbitrary input $\mathcal{X}$ a continuous variable.

$$f : \mathcal{X} \to \mathbb{R}$$

Inputs $x \in \mathcal{X}$ can be any kind of objects images, text, a sequence of amino acids, ...
The output $y \in \mathbb{N}/y \in \mathbb{R}$ is either a discrete or real number, but only one single number. Examples are the label of an image as in classification, or regressing a single number for the stock market, a density estimation, ...

Structured Output Learning:
In structured output learning, both the input and the output are structured objects.

$$f : \mathcal{X} \to \mathcal{Y}$$

The inputs $x \in \mathcal{X}$ can be any kind of objects and the outputs $y \in \mathcal{Y}$ are complex (structured) objects such as images, text, parse trees, folds of a protein, computer programs, ...

Today we will focus on the regression problem. A model f always takes an input x and produces an output y. There are two main tasks, the learning task and the interference task.

Learning: Learning happens through estimating the parameters $\mathbf{w}$ from training data $\{(x_i, y_i)\}_{i=1}^N$
Inference: Given $\mathbf{w}$ from the learning task, novel predictions are made: $y = f_{\mathbf{w}}(x)$

Classification
An example is the mapping of an input image to the output label "Beach" or "No Beach", thus performing the task of classification:
Mapping: $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \to \{\text{"Beach"}, \text{"No Beach"}\}$

Regression
For a regression problem, a stock value prediction with N numbers as input and a single number as output aree an example:
Mapping: $f_{\mathbf{w}} : \mathbb{R}^N \to \mathbb{R}$

Structured Prediction
An example for structured prediction problems is an audio signal as input and a sequence of words, forming a sentence as output.
Mapping: $f_{\mathbf{w}} : \mathbb{R}^N \to \{1, \dots, L\}^M$

Semantic segmentation is another example for structured prediction, where the input is an image and the output is a label map.
Mapping: $f_{\mathbf{w}} : \mathbb{R}^{W \times H} \to \{1, \dots, L\}^{W \times H}$

A third example of structured prediction is 3D reconstruction, the input is a set of images and the output is a 3D reconstruction. Mapping: $f_{\mathbf{w}} : \mathbb{R}^{W \times H \times N} \to \{0, 1\}^{M^3}$

For this mapping, suppose we want to predict 32^3 voxels, with a binary variable per voxel (occupied/free) $2^{32^3} = 2^{32768}$ different reconstructions could be predicted by the model. This number is even larger than the number of atoms in the universe, which is $\sim 2^{273}$.

1.3.3 Linear Regression

In linear regression, we want to make a prediction for the model parameters in a linear sense. Formally, this means: Let $\mathcal{X}$ denote a dataset of size N and let $(\mathbf{x}_i, y_i) \in \mathcal{X}$ denote its elements ($y_i \in \mathbb{R}$). The goal is, to predict y for a previously unseen input $\mathbf{x}$. The input $\mathbf{x}$ may be multidimensional.
As an example, we try to fit a line as the "ground truth" over some noisy observations, which are samples from the model with added noise. This is what we provide to the learning algorithm. When we execute that model at a novel location $\mathbf{x}$, we get a sensible response $\mathbf{y}$. This is done by defining an error function.
The **error function** $E(\mathbf{w})$ measures the displacement along the y dimension between the data points and the model $f(\mathbf{x}, \mathbf{w})$ specified by the parameters $\mathbf{w}$. It aims at minimizing the overall error.

$$
\begin{aligned}
f(\mathbf{x}, \mathbf{w}) &= \mathbf{w}^\top \mathbf{x} \\
E(\mathbf{w}) &= \sum_{i=1}^{N} \left(f(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2 \\
&= \sum_{i=1}^{N} \left(\mathbf{x}_i^\top \mathbf{w} - y_i\right)^2 \\
&= \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2
\end{aligned}
$$

In linear regression, we have a function represented as a linear model. $\mathbf{w}^\top \mathbf{x}$ is the inner product of the weight vector $\mathbf{w}$ and the input features $\mathbf{x}$. The error function is the sum over all data points of the square of the difference in the $\mathbf{y}$ direction between the prediction of the model and the observation $\mathbf{y}$. This can be written as the squared ℓ_2 norm of the matrix $\mathbf{X}$ times the weight vector $\mathbf{w}$ minus the vector $\mathbf{y}$.

The **gradient of the error function** with respect to the parameters $\mathbf{w}$ is given by:

$$
\begin{aligned}
\nabla_{\mathbf{w}} E(\mathbf{w}) &= \nabla_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 \\
&= \nabla_{\mathbf{w}} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \\
&= \nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}\right) \\
&= 2\mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{X}^\top \mathbf{y}
\end{aligned}
$$

As $E(\mathbf{w})$ is quadratic and convex in $\mathbf{w}$, its minimizer (wrt. $\mathbf{w}$) is given in closed form. This means that we can set $\nabla_{\mathbf{w}} E(\mathbf{w})$ to zero to obtain a closed form solution:

$$
\nabla_{\mathbf{w}} E(\mathbf{w}) = 0 \Rightarrow \mathbf{w} = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}
$$

The matrix $\left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top$ is also called **Moore-Penrose inverse** or pseudoinverse because it allows us, to compute the inverse of a non-square matrix, which is the minimizer of this least-square problem.

Example: Line Fitting
A linear least squares fit of the model $f(\mathbf{x}, \mathbf{w}) = w_0 + w_1 x$ to the data points can be obtained with line fitting. The error function $E(\mathbf{w})$ wrt. parameter w_1 is a square function that produces a parabola. In order to minimize the error, we must find the lowest point of the parabola.

Example: Polynomial Curve Fitting
Let us choose a **polynomial of order M** to model dataset $\mathcal{X}$:

$$
f(x, \mathbf{w}) = \sum_{j=0}^{M} w_j x^j = \mathbf{w}^\top \mathbf{x} \qquad \text{with features} \qquad \mathbf{x} = (1, x^1, x^2, \ldots, x^M)^\top
$$

This function is linear in $\mathbf{w}$, but it is not linear in x.
There are two tasks, the training and the interference. In training, we want to estimate $\mathbf{w}$ from dataset $\mathcal{X}$ and during inference we want to predict y for novel x given estimated $\mathbf{w}$. Note that these features can be anything, including multi-dimensional inputs (e.g., images, audio), radial basis functions, sine/cosine functions, etc. In this example of polynomial curve fitting we have monomials.

For estimating $\mathbf{w}$ from $\mathcal{X}$, we define a squared error function, e.g.:

$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(f(x_i, \mathbf{w}) - y_i\right)^2$$

The goal is, to optimize error function wrt. the parameters $\mathbf{w}$.
The error function is linear in $\mathbf{w}$ but not in x. This means, that wrt. $\mathbf{w}$ we obtain the same problem as before; we have a quadratic problem with a closed form solution:

$$E(\mathbf{w}) = \sum_{i=1}^{N} \left(f(x_i, \mathbf{w}) - y_i\right)^2 = \sum_{i=1}^{N} \left(\mathbf{w}^\top \mathbf{x}_i - y_i\right)^2 = \sum_{i=1}^{N} \left(\sum_{j=0}^{M} w_j x_i^j - y_i\right)^2$$

It can be rewritten in the **matrix-vector notation** (i.e., as linear regression problem).

$$E(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

with feature matrix $\mathbf{X}$, observation vector $\mathbf{y}$ and weight vector $\mathbf{w}$:

$$\mathbf{X} = \begin{pmatrix} \vdots & \vdots & \vdots & & \vdots \\ 1 & x_i & x_i^2 & \dots & x_i^M \\ \vdots & \vdots & \vdots & & \vdots \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} \vdots \\ y_i \\ \vdots \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} w_0 \\ \vdots \\ w_M \end{pmatrix}$$

Example for Polynomial Curve Fitting Results

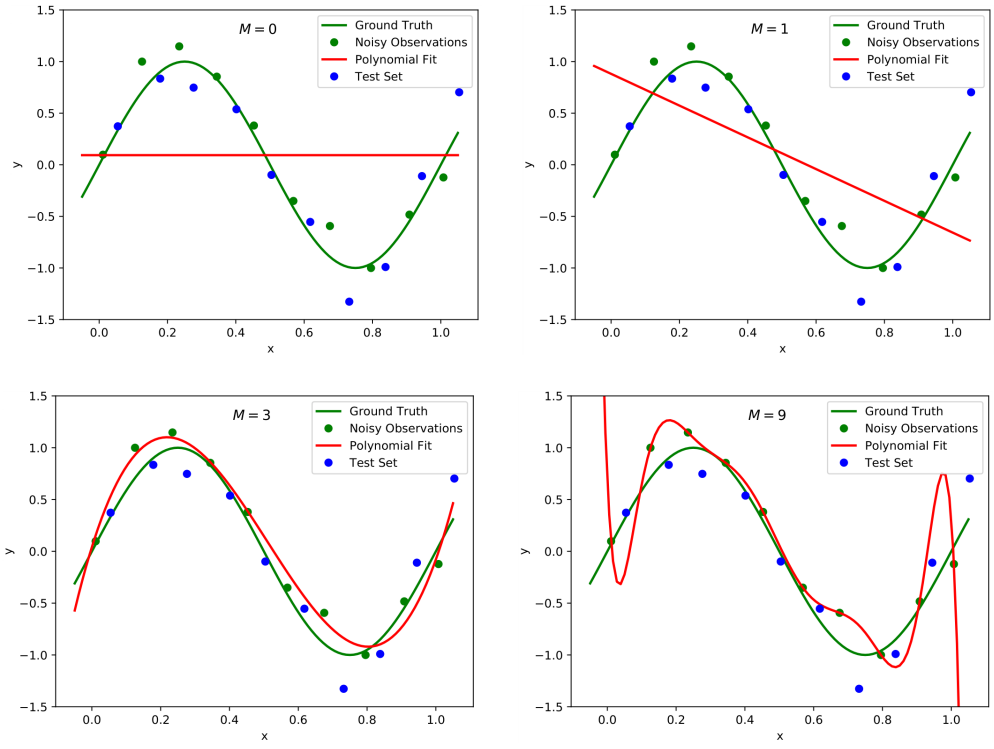


Figure 1: **Polynomial Curve fitting.** Plots of polynomials of various degrees M (red) fitted to the data (green). We observe underfitting ($M = 0/1$) and overfitting ($M = 9$). Choosing the best fitting degree M is a model selection problem.

Capacity, Overfitting and Underfitting
In summary, the goal of polynomial curve fitting is to perform well on new, previously unseen inputs (test set, blue in Fig. 1), not only on the training set (green in Fig. 1). This is called **generalization** and separates

machine learning from optimization. In optimization, we are just interested in fitting a model to observations while in machine learning, we are always interested in a model that generalizes well.
The assumption that is often made in statistical learning theory, is that the training and test data are independent and identically (i.i.d.) drawn from the data distribution $p_{data}(x, y)$. This is important, because this assures that we can make certain statements about the learning problem. In the case of Fig. 1 the data distribution has been chosen as follows:

$$p_{data}(x) = \mathcal{U}(0, 1)$$
$$p_{data}(y|x) = \mathcal{N}(\sin(2\pi x), \sigma)$$

We have an underlying sine curve with x-locations between 0 and 1. On this basis we sample a y-location by adding some gaussian noise with the standard deviation σ to the sine curve.

Clarification of the used terminology:
Capacity refers to the complexity of functions which can be represented by the model f. In Fig. 1, for $M = 0/1$ the capacity is too low, for $M = 3$ the capacity is about right, and for $M = 9$ the capacity is too high.
Underfitting refers to models being too simple, so they don't achieve low error, not even on the training set.
Overfitting refers to a scenario where the training error is very small, but test error (= generalization error) is large. So the model overfits the training data and doesn't generalize well to the test set. We want neither an underfitting nor an overfitting model, but a model that performs well on the test set and thus generalizes well.
As in the example of the generalization error for various polynomial degrees M, the model selection problem is as follows: we want to select the model with the smallest generalization error.
In general, we split the dataset into a training, a validation and a test set in order to chose the best model. We choose hyperparameters (e.g., degree of polynomial, learning rate in neural net, ..) using the validation set. It is important to evaluate only once on test set (true labels are typically not available). When the dataset is small, use (k-fold) cross validation instead of a fixed split.

1.3.4 Ridge Regression

Ridge Regression is another way of reducing the model complexity, but not by discretely changing the degree or the order of the polynomial, but by adding a regularization term. The error function of the ridge regression problem looks as follows:

$$E(\mathbf{w}) = \sum_{i=1}^{N} (f(x_i, \mathbf{w}) - y_i)^2 + \lambda \sum_{j=0}^{M} w^2$$
$$= \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2$$

The idea is, to discourage large parameters by adding a regularization term with strength λ. Since this problem is quadratic in $\mathbf{w}$, it also has a closed form solution: $\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$.

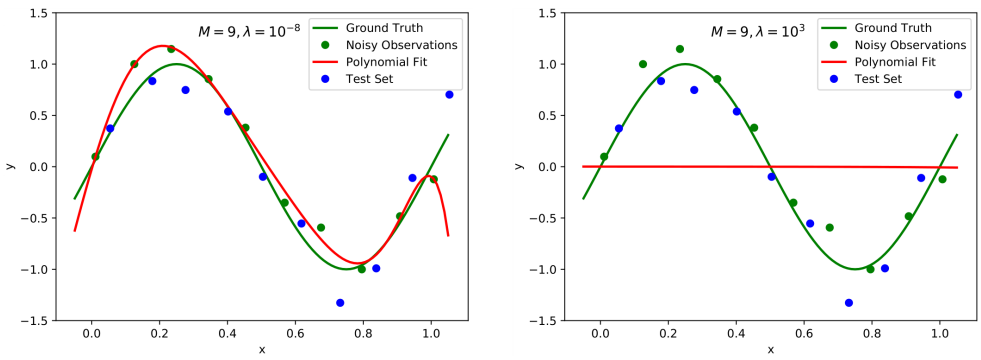


Figure 2: **Ridge Regression.** Plots of polynomial with degree $M = 9$ fitted to 10 data points using ridge regression. Left: weak regularization ($\lambda = 10^{-8}$). Right: strong regularization (right, $\lambda = 10^{3}$).

In Fig. 2 we can see, that for the formerly overfitting model with $M = 9$ we can obtain a much better fit with

a mild regularizer. With a strong regularizer the model remains constantly 0, which refers to the underfitting situation. If we choose a very small regularizer, parameters or model weights can become very large and the problem becomes ill-conditioned. If we increase the regularizer, the weights become smaller. Again, we want to select a model with the smallest generalization error on the validation set.

1.3.5 Estimators, Bias and Variance

Point Estimator:
A point estimator $g(\cdot)$ is a function that maps a dataset $\mathcal{X}$ to model parameters $\hat{\mathbf{w}}$: $\hat{\mathbf{w}} = g(\mathcal{X})$. We denote $\hat{\mathbf{w}}$ with a hat to explicitly mark that this is an estimate of the point estimator. An example is the estimator of the ridge regression model: $\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$. A good estimator is a function that returns a parameter set close to the true one. We assume that the data $\mathcal{X} = \{(x_i, y_i)\}$ is drawn from a random process $(x_i, y_i) \sim p_{data}(\cdot)$, thus, any function of the data is random and $\hat{\mathbf{w}}$ is a random variable. This leads us to the terminology of bias and variance and the so-called bias and variance dilemma.

Properties of Point Estimators:
The bias of a point estimator is the expected value of the point estimate over all data sets that can be drawn from the data distribution minus the true value of the parameters:

$$\text{Bias}(\hat{\mathbf{w}}) = \mathbb{E}(\hat{\mathbf{w}}) - \mathbf{w}$$

This is how, in expectation, the point estimator deviates from the true parameters. This expectation is over all possible datasets $\mathcal{X}$. $\hat{\mathbf{w}}$ is unbiased $\Leftrightarrow \text{Bias}(\hat{\mathbf{w}}) = 0$. A good estimator has little bias.

The variance of a point estimator is the variance over $\hat{\mathbf{w}}$ over all datasets $\mathcal{X}$:

$$\text{Var}(\hat{\mathbf{w}}) = \mathbb{E}(\hat{\mathbf{w}}^2) - \mathbb{E}(\hat{\mathbf{w}})^2$$

The square root of the variance $\sqrt{\text{Var}(\hat{\mathbf{w}})}$ is called "standard error". A good estimator also has low variance.

Bias-Variance Dilemma:
Statistical learning theory tells us that we can't have a little bias as well as a low variance. There is a trade-off that we have to make.

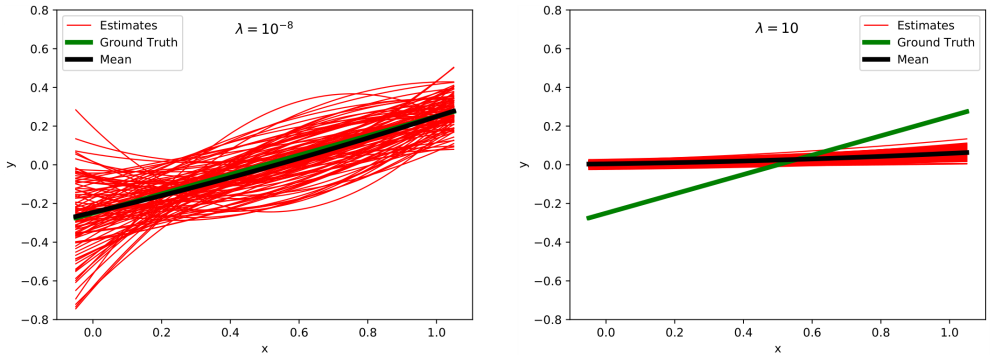


Figure 3: **Bias-Variance Dilemma.** Ridge regression with weak ($\lambda = 10^{-8}$) and strong ($\lambda = 10$) regularization. Green: True model. Black: Plot of model with mean parameters $\bar{\mathbf{w}} = \mathbb{E}(\mathbf{w})$. Red: Estimates, with high variance at the right and low variance at the left.

In Fig. 3 we can see that if we have a weak regularization, we have a large variance but the mean is a good fit to the ground truth, so the bias is small. With a strong regularization, we obtain less variance in the predictions, however, the model has a strong bias. The mean of the parameters (the red estimates) deviates strongly from the ground truth. So there is a bias-variance tradeoff which can be expressed mathematically: $\mathbb{E}[(\hat{\mathbf{w}} - \mathbf{w})^2] = \text{Bias}(\hat{\mathbf{w}})^2 + \text{Var}(\hat{\mathbf{w}})$. But we might not have to deal with this in all situations. E.g. in deep neural networks the test error decreases with network width. For further information see the blogpost at https://www.bradyneal.com/bias-variance-tradeoff-textbooks-update.

1.3.6 Maximum Likelihood Estimation

We now reinterpret our results by taking a probabilistic viewpoint. Let $\mathcal{X} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ be a dataset with samples drawn i.i.d. from the data distribution p_{data}. And let the model $p_{model}(y|\mathbf{x}, \mathbf{w})$ be a parametric family of probability distributions. Then the conditional maximum likelihood estimator for $\mathbf{w}$ is given by

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}}\ p_{model}(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\
&\overset{\text{iid}}{=} \underset{\mathbf{w}}{\operatorname{argmax}}\ \prod_{i=1}^{N} p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Log-Likelihood}}
\end{aligned}$$

Example: If we assume that the model distribution is a Gaussian, where the mean is a linear function, and the variance is σ: $p_{model}(y|\mathbf{x}, \mathbf{w}) = \mathcal{N}(y|\mathbf{w}^\top \mathbf{x}, \sigma)$, we obtain

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}}\ \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ \sum_{i=1}^{N} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{1}{2\sigma^2}\left(\mathbf{w}^\top \mathbf{x}_i - y_i\right)^2} \right] \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ -\sum_{i=1}^{N} \frac{1}{2} \log(2\pi\sigma^2) - \sum_{i=1}^{N} \frac{1}{2\sigma^2} \left(\mathbf{w}^\top \mathbf{x}_i - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmax}}\ -\sum_{i=1}^{N} \left(\mathbf{w}^\top \mathbf{x}_i - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmin}}\ \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2
\end{aligned}$$

We see that choosing $p_{model}(y|\mathbf{x}, \mathbf{w})$ to be Gaussian causes maximum likelihood to yield exactly the same least squares estimator derived before:

$$\hat{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}}\ \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

There are various variations possible here. If we were choosing e.g. the model distribution $p_{model}(y|\mathbf{x}, \mathbf{w})$ as a Laplace distribution, we would obtain an estimator that minimizes the ℓ_1 norm: $\hat{\mathbf{w}} = \operatorname{argmin}_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_1$. Or otherwise assuming a Gaussian distribution over the parameters $\mathbf{w}$ and performing a maximum a-posteriori (MAP) estimation yields to ridge regression:$\operatorname{argmax}_{\mathbf{w}} p(\mathbf{w}|y, \mathbf{x}) = \operatorname{argmax}_{\mathbf{w}} p(y|\mathbf{x}, \mathbf{w})p(\mathbf{w})$.
This shows that there is a connection between maximum likelihood estimation and the linear least squares problems that have been explained earlier. This is important, because maximum likelihood estimators are a very important tool in statistical learning theory because they are known to be consistent under mild assumptions. This means, that as the number of training samples approaches infinity $N \to \infty$, the maximum likelihood (ML) estimate converges to the true parameters. They are also very efficient: The ML estimate converges most quickly as N increases These theoretical considerations make ML estimators appealing.

2 Computation Graphs

2.1 Logistic Regression

We've already got to know the maximum likelihood estimator and more precisely the Conditional **Maximum Likelihood Estimator** for $\mathbf{w}$:

$$\hat{\mathbf{w}}_{ML} = \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \tag{1}$$

because we're conditioning on some input x we're not just trying to model a distribution y but we are trying to model a conditional distribution y given x. We can see here the Maximum Loglikehood estimator, we have the log of the product, which is the sum of the logarithm. However, the estimate that we obtain - argmax, is the same as the Maximum Likelihood Estimator would return as the logarithm is a monotonic function. In practice we prefer to use loglikelihood estimator both for numerical reasons when implementing these algorithms because this is resulting in more stable computation, but also for mathematical reasons and for connection to the concepts in information theory. We'll mostly use loglikelihood from now, it doesn't return the same value, but it returns the same maximum, the same parameters w. We denote the estimate as $\hat{\mathbf{w}}_{ML}$. This hat indicates that this is an estimate of the true parameters.Now we want to perform a binary classification: $y_i \in \{0, 1\}$. So the output can take only two possible discrete labels - 0 and 1.The question is - how should we choose $p_{model}(y|\mathbf{x}, \mathbf{w})$ in this case? Gaussian model isnt a good choice, because it's a continuous distribution. Bernoulli distribution is good for this case, it models binary classification problem:

$$p_{model}(y|\mathbf{x}, \mathbf{w}) = \hat{y}^y \left(1 - \hat{y}\right)^{(1-y)} \tag{2}$$

where $\hat{y}$ is a prediction of some model and y is the true label from our dataset. So $\hat{y}$ is some prediction that must depend on the input $\mathbf{x}$ and it must also depend on some trainable parameter $\mathbf{w}$, we denote that function as: $\hat{y} = f_{\mathbf{w}}(\mathbf{x})$.
We are working with the discrete distribution, so we have the next requirement $f_{\mathbf{w}}(\mathbf{x}) \in [0, 1]$. We can use a sigmoid function: $f_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ where σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

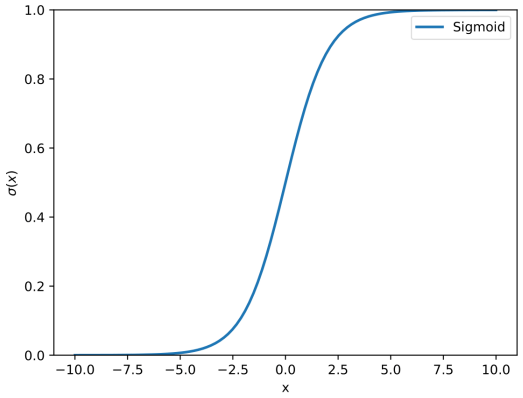


Figure 4: **Sigmoid.** An illustration of the Sigmoid function.

We take the linear combination of the weights and then we do the non-linear transformation sigma to it. Sigmoid takes an unbounded range of the real numbers and transform this domain of real numbers to the domain of from 0 to 1 interval. That's why this function is called squashing function. And that's what we want - we want the output of this function between 0 and 1 to be a proper probability. That's also a reason why is this called a logistic regression, because this is logistic transfer function.

Lets put it together:

$$\hat{\mathbf{w}}_{ML} = \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \tag{4}$$

$$= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \left[\hat{y}_i^{y_i} \left(1-\hat{y}_i\right)^{(1-y_i)}\right] \tag{5}$$

$$= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{-y_i \log \hat{y}_i - (1-y_i) \log(1-\hat{y}_i)}_{\text{Binary Cross Entropy Loss } \mathcal{L}(\hat{y}_i, y_i)} \tag{6}$$

Here for the model distribution we plug in the Bernoulli distribution, where

$$\hat{y}_i = \sigma(\mathbf{w}^\top \mathbf{x}_i)$$

. Because we have a logarithm, we can transform it, we split the product with a sum. And then instead of maximizing we can minimize the last expression by writing a minus in front. In machine learning we often don't maximize likelihoods, but we rather minimize loss functions and that's why we are writing it this way here and this this term in the sum has a specific name in the machine learning community, it is called a **binary cross-entropy loss**. It's a loss between the predicted y and the true y and as the name loss indicates a high loss is bad and a low loss is desirable, so we want to minimize the loss, we want to compute the overall minimum over the data set of all these individual losses. In machine learning we use often the more general term loss rather than what we have used before in linear regression the error function and the reason is that this loss term is more general, it's not necessarily an error between observations,it could also be for instance an inductive bias that we want to encode. Now this formula has the following interpretation, we minimize the dissimilarity between the empirical data distribution p_{data} (defined by the training set) and the model distribution p_{model}. We want to find a parameter for our model, such that the prediction of our model is most similar to our data. Lets look at the Binary cross entropy loss:

$$\mathcal{L}(\hat{y}_i, y_i) = -y_i \log \hat{y}_i - (1-y_i) \log(1-\hat{y}_i) \tag{7}$$

$$= \begin{cases} -\log \hat{y}_i & \text{if } y_i = 1 \\ -\log(1-\hat{y}_i) & \text{if } y_i = 0 \end{cases} \tag{8}$$

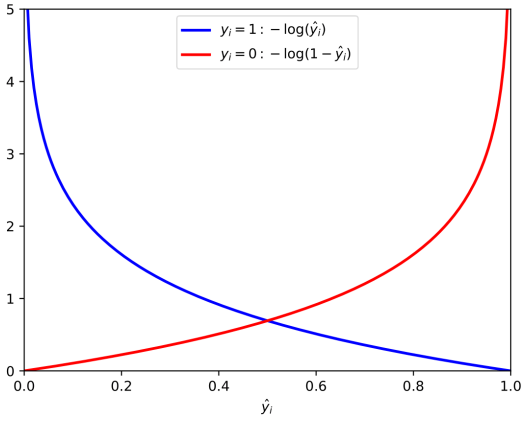


Figure 5: **Curves.** An illustration of the those two curves.

We want to minimize those curves. For $y_i = 1$ the loss $\mathcal{L}$ is minimized if $\hat{y}_i = 1$. For $y_i = 0$ the loss $\mathcal{L}$ is minimized if $\hat{y}_i = 0$.Thus, $\mathcal{L}$ is minimal if $\hat{y}_i = y_i$. This loss function can be extended to > 2 classes.

2.1.1 1D example

Lets look at this visually.
We have on the x-axis our features, this is the input 1d and we want to classify between positive and negative. So we want to tell for a particular x value is it in the positive class or in a negative class. We have set of positive samples in green and negative in red. We want to fit to dataset $\mathcal{X}$ logistic regression model $f_{\mathbf{w}}(x) = \sigma(w_0 + w_1 x)$

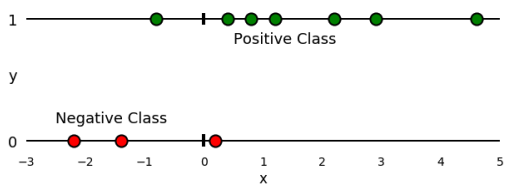


Figure 6: **1D example.** An illustration of the simple 1D example.

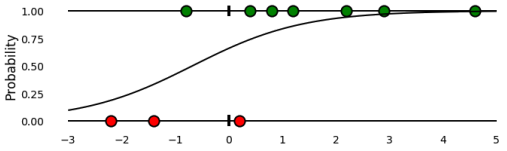


Figure 7: **Model fitting.** An illustration of fitting the log regression model.

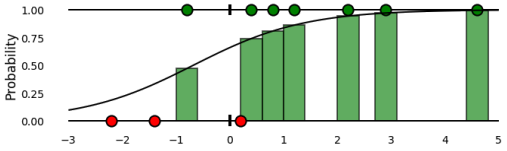


Figure 8: **Probabilistic interpretation.** Probabilities of classifier $f_{\mathbf{w}}(x_i)$ for positive samples ($y_i = 1$)

How do we interpret this in terms of probabilities and the loss function?
The probabilities of the classifier for the positive samples are shown with green bars here where each of these green bars is basically extending all the way up to the curve because this is the probability that the model assigns to these positive values. What we can see is already the criterion that we want to optimize, for all data points we want to maximize these probabilities being correctly assigned.
And these are the probabilities for the negative points Now we can put all of these probabilities together and

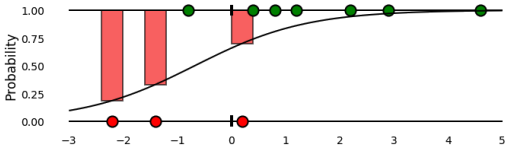


Figure 9: **Probability of the negative class.** Probabilities of classifier $f_{\mathbf{w}}(x_i)$ for negative samples ($y_i = 0$)

we"ll just put them to the bottom, because it makes more sense.

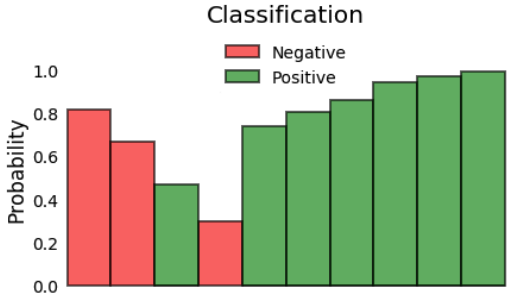


Figure 10: **Probability of the both classes.** Probabilities of classifier for the both classes

We want to maximize those probabilities or equivalently we want to minimize the negative logarithm.
So then we minimize the mean or the sum over all these negative log probabilities.

2.1.2 Optimizing w.

In contrast to linear regression, the loss $\mathcal{L}(\hat{y}_i, y_i)$ is **not quadratic** in $\mathbf{w}$. $\mathbf{w}$ appears inside the sigmoid which is a nonlinear function and then we take the logarithm of this nonlinear function, which is also a non-linear

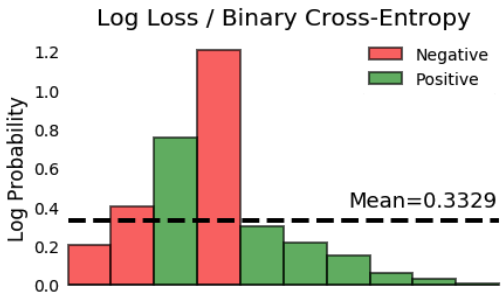


Figure 11: **Log loss.** Log loss of classifier for both classes

function, so this is highly non-linear transformation of linear model and we don't have an easy expression as we had it in the quadratic case for the linear regression task. If there is no closed form analytic solution, we need to apply some iterative optimization technique which is based on gradients. Luckily we can compute gradients and so we can use a gradient based optimizer, that slowly follows the gradient to a local minimum starting from some initialization point.Despite this not being a quadratic function we, it's still a convex function.It can be shown that this problem is a convex problem, so we know that we don't fall into a local optimum. If we apply a gradient based optimizer, we end up with the global optimum.But we can't reach it in a single step or analytically, we need to apply an iterative gradient-based algorithm. And for all the gradient-based optimization algorithms, where the grading is needed, we need to compute this gradient which is the nabla operator. The nice thing about this particular objective function is that despite in the in the first step the gradient looks quite complex, it simplifies a lot when you do the analytical derivation and this is the final form of the gradient of the binary cross entropy loss of this logistic regression model:

$$\nabla_{\mathbf{w}} \mathcal{L}(\hat{y}_i, y_i) = (\hat{y}_i - y_i)\mathbf{x}_i \tag{9}$$

Since we are able to compute this gradient analytically, we can apply it inside a iterative gradient based optimizer, which tries to go stepwise towards the optimum. The simplest optimizer that we can use and one that's actually used nowadays heavily even in deep learning is called gradient descent. We pick the step size η and tolerance ϵ and Initialize $\mathbf{w}^0$. And then we repeat until $\|\mathbf{v}\| < \epsilon$. We iterate the procedure:

$$\mathbf{v} = \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{i=1}^{N} \nabla_{\mathbf{w}} \mathcal{L}(\hat{y}_i, y_i) \tag{10}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \mathbf{v} \tag{11}$$

We optimize this until this gradient becomes small and converges to an extreme value of this function.

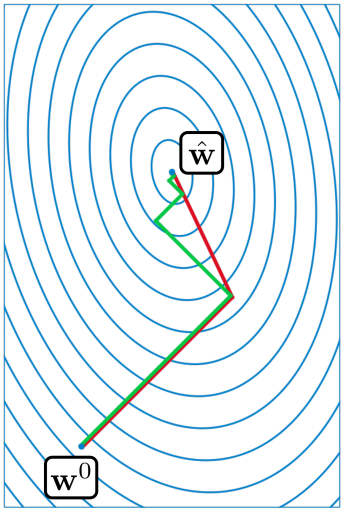


Figure 12: **Gradient descent.** Example of the converging variants of gradient descent.

But there's several variants that work a little bit better, one is line search, that's the green curve, where you're going into the direction of the gradient,but you search for the smallest value. You can do even better for some problems by using the conjugate gradient method (red), where you're not going into the direction of the gradient, but into some conjugate direction, that brings you more directly to the optimal value.

2.1.3 Connections to the Information Theory

Maximizing the **Log-Likelihood** is equivalent to minimizing **Cross Entropy** or **KL Divergence:**

$$\hat{\mathbf{w}}_{ML} = \underset{\mathbf{w}}{\text{argmax}} \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Log-Likelihood}} \tag{12}$$

$$= \underset{\mathbf{w}}{\text{argmax}} \ \mathbb{E}_{p_{data}} [\log p_{model}(y|\mathbf{x}, \mathbf{w})] \tag{13}$$

$$= \underset{\mathbf{w}}{\text{argmin}} \underbrace{-\mathbb{E}_{p_{data}} [\log p_{model}(y|\mathbf{x}, \mathbf{w})]}_{\text{Cross Entropy } H(p_{data}, p_{model})} \tag{14}$$

$$= \underset{\mathbf{w}}{\text{argmin}} \ \mathbb{E}_{p_{data}} [\log p_{data}(y|\mathbf{x}) - \log p_{model}(y|\mathbf{x}, \mathbf{w})] \tag{15}$$

$$= \underset{\mathbf{w}}{\text{argmin}} \underbrace{D_{KL}(p_{data} \| p_{model})}_{\text{KL Divergence}} \tag{16}$$

We can rewrite log likelihood using an expectation, because what if we would write 1 over N, which would be the mean, then we would look at the empirical estimate over the data distribution,but one over N is a constant with respect to w so it gets absorbed into the argmax operator. But equation having a one over N in the beginning would exactly correspond to the definition of the expectation operator, which is basically expectation over the probability of the data points. So each of the data points is a sampling based estimate of the $\log p_{model}$. We can write argmax in terms of argmin, if we replace plus with a minus. Now what we can do also is we can take this term and add the $\log p_{data}(y|\mathbf{x})$.We can do that because the data probability in this data distribution does not depend on w, so it's changing the value of this function, but it's constant with respect to w.It means that we have the cross-entropy minus the entropy and this in information theory terms is called the KL divergence. It's one of many divergences, but it's a measure of the similarity of two distributions.This is a nice intuition here, by computing the maximum likelihood estimate of the parameters we're trying to minimize the distance between two distributions,more precisely between the data distribution given by the data set - the empirical data distribution and the distribution of our model given its parameters w.

2.2 Computation graphs.

In the second unit of this lecture we will introduce computation graphs, which is a fundamental concept to understand larger changes of computations and also the computation of gradients in this large chains of computations. Unfortunately it's not true that these gradients are so simple to compute in general.For more complex models that we're interested in this lecture, in particular for deep neural networks, where you simply can't derive the gradients with respect to any of these gigantic number of parameters just using pen and paper. So how can we basically efficiently compute the gradients in the general?

2.2.1 Key idea.

The key idea of computation graphs is to **decompose** complex computations into sequences of very simple, more atomic assignments. We call this sequence of assignments a **computation graph** or **source code**. The **forward pass** takes a training point $(\mathbf{x}, y)$ as input and computes a loss, e.g.:

$$\mathcal{L} = -\log p_{model}(y|\mathbf{x}, \mathbf{w})$$

As we will see, gradients $\nabla_{\mathbf{w}}\mathcal{L}$ can be computed using a **backward pass**. Both, the forward pass and the backward pass are **efficient** due to the use of dynamic programming, i.e., storing and reusing intermediate results.This decomposition and reuse of computation is key to the success of the **backpropagation algorithm**, the primary workhorse of deep learning. It would be unthinkable to optimize deep networks with millions or billions of parameters without this decomposition and reuse of computation. That's why the backpropagation algorithm which adjusts these millions and billions of parameters in our deep models is still the primary workhorse for deep learning today.

2.2.2 Computation graph.

A computation graph has three kinds of nodes: input nodes in green, parameter notes in orange and compute nodes in red. Also the loss function is a compute node in our setting. The input nodes don't have parameters they are the input of our learning problem, they are basically the dataset,this is where the data set goes into the computation graph. The parameter nodes is where the parameters of the model are stored and this is what we're interested in when backpropagating gradients. We need to backpropagate gradients for all nodes, but

what we're really interested in the end is the gradients for these parameters, because this is where the updates are applied to. Finally we have the compute nodes which could be aver the final node (the loss function) or intermediate compute nodes that take inputs, for instance input nodes or previous compute nodes or parameters and produce a result.

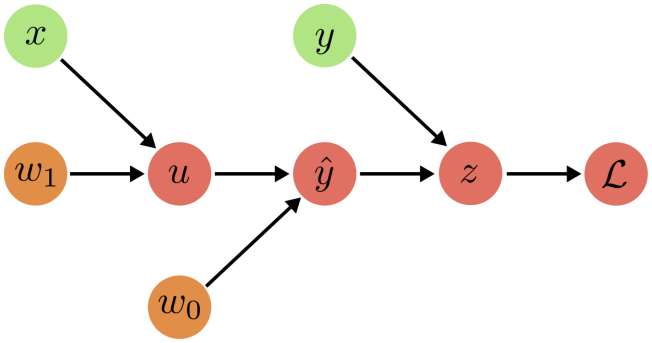


Figure 13: **Computation graph.** Example of the computation graph for linear regression.

We're looking at a very specific example of linear regression.This is the source code here:

$$
\begin{aligned}
&(1) \quad u = w_1 x \\
&(2) \quad \hat{y} = w_0 + u \\
&(3) \quad z = \hat{y} - y \\
&(4) \quad \mathcal{L} = z^2
\end{aligned}
$$

This is how we write the linear regression problem, which we could also write in a single line, we write it in as a sequence of atomic operations. Now there is multiple levels of granularities that we can use. This is a very fine level of granularity, but we can use more coarse-grained levels of granularity as well and what we want to choose depends on how big we want to make these atomic units, which level of atomic units we still can handle. So in this case here we could combine (1) and (2) , then we would change this computation graph.

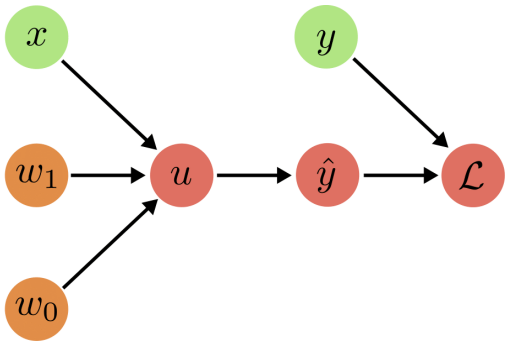


Figure 14: **Computation graph with another level of granularity.** Example of the computation graph with another level of granularity for linear regression.

Steps (1) and (2) have collapsed into one step, which directly computes this affine transformation.
We can also for instance collapse the loss, we can compute not these two things here independently but we don't want to compute them jointly:
So now we have only two steps:

$$
\begin{aligned}
&(1) \quad \hat{y} = w_0 + w_1 x \\
&(2) \quad \mathcal{L} = (\hat{y} - y)^2
\end{aligned}
$$

Here is the example for the logistic regression:

$$
\begin{aligned}
&(1) \quad u = w_0 + w_1 x \\
&(2) \quad \hat{y} = \sigma(u) \\
&(3) \quad \mathcal{L} = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})
\end{aligned}
$$

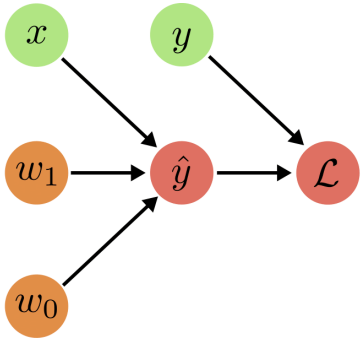


Figure 15: **Computation graph with another level of granularity.** Example of the computation graph with another level of granularity for linear regression.

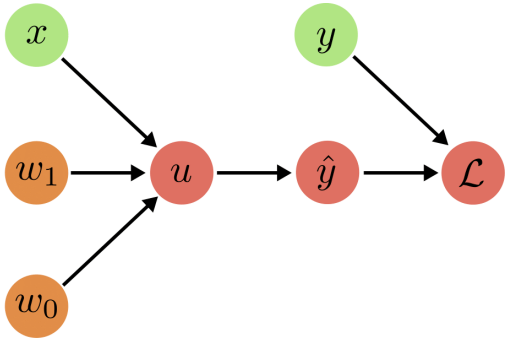


Figure 16: **Computation graph of logistic regression.** Example of the computation graph for logistic regression.

We could also write this in terms of vectors $\mathbf{w}$ and $\mathbf{x}$:

$$
\begin{aligned}
&(1) \quad u = \mathbf{w}^\top \mathbf{x} \\
&(2) \quad \hat{y} = \sigma(u) \\
&(3) \quad \mathcal{L} = -y \log \hat{y} - (1-y) \log(1-\hat{y})
\end{aligned}
$$

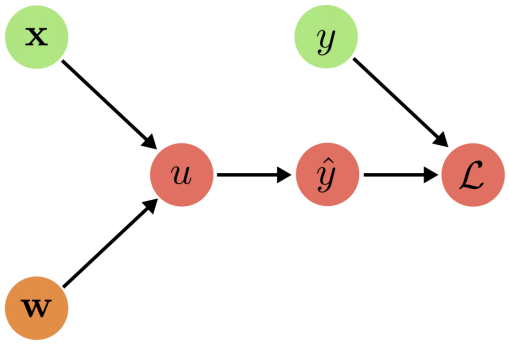


Figure 17: **Computation graph with vector representation.** Example of the computation graph with vector representation for logistic regression.

What we can also do is we can stack two of these operations behind each other:

$$
\begin{aligned}
&(1) \quad \mathbf{h} = \sigma(\mathbf{W}_1^\top \mathbf{x}) \\
&(2) \quad \hat{y} = \sigma(\mathbf{w}_2^\top \mathbf{h}) \\
&(3) \quad \mathcal{L} = -y \log \hat{y} - (1-y) \log(1-\hat{y})
\end{aligned}
$$

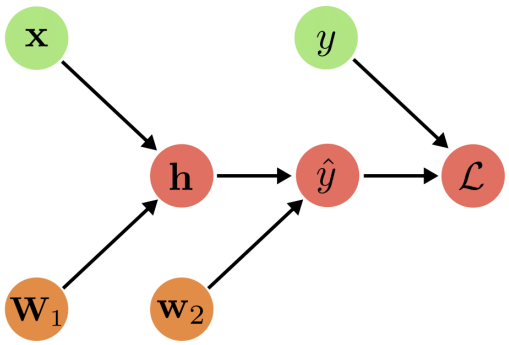


Figure 18: **Multi-Layer Perceptron.** Example of the computation graph for Multi-Layer Perceptron.

2.3 Backpropagation.

Our goal is to find gradients of negative log likelihood:

$$\nabla_{\mathbf{w}} \sum_{i=1}^{N} \underbrace{-\log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\mathcal{L}(y_i, \mathbf{x}_i, \mathbf{w})} \tag{17}$$

or more generally of a loss function

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{y}, \mathbf{X}, \mathbf{w}) = \nabla_{\mathbf{w}} \sum_{i=1}^{N} \mathcal{L}(y_i, \mathbf{x}_i, \mathbf{w}) = \sum_{i=1}^{N} \nabla_{\mathbf{w}} \mathcal{L}(y_i, \mathbf{x}_i, \mathbf{w}) \tag{18}$$

given a dataset $\mathcal{X} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ with N elements. In the following, we consider the computation of gradients wrt. a single data point: $\nabla_{\mathbf{w}} \mathcal{L}(y_i, \mathbf{x}_i, \mathbf{w})$. The gradient with respect to the entire dataset $\mathcal{X}$ is obtained by summing up all individual gradients.

2.3.1 Chain rule.

In order to understand the backpropagation algorithm there's basically just one rule that we have to understand and that's the chain rule:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g(x)) = \frac{\mathrm{d}f}{\mathrm{d}g} \frac{\mathrm{d}g}{\mathrm{d}x} \tag{19}$$

We're also going to need the so-called multivariate chain rule:4

$$\frac{\mathrm{d}}{\mathrm{d}x} f(g_1(x), \ldots, g_M(x)) = \sum_{i=1}^{M} \frac{\partial f}{\partial g_i} \frac{\mathrm{d}g_i}{\mathrm{d}x} \tag{20}$$

The multivariate chain rule is concerned with compositions of functions, where in the argument of a function we have a set of functions and each of these functions here depends on the same variable x.

2.3.2 Backpropagation algorithm.

Let's look at the most simple form of backpropagation algorithm, one of the most basic examples, which is basically just the chain rule in a slightly different form.For now we're going to not make any distinction between node types.
We're interested in the gradient of this loss with respect to the loss itself or the variable y or the variable x.

Forward Pass:

$$\begin{aligned} (1) \quad & y = y(x) \\ (2) \quad & \mathcal{L} = \mathcal{L}(y) \end{aligned}$$

We're going to after running this forward pass, actually calculating the value of $\mathcal{L}$ for a particular input x, we are going to run a so called backward pass, that's where the name backpropagation is coming from. **Backward**

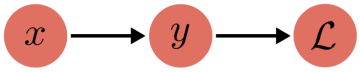


Figure 19: **Simple computation graph.**

Pass:

$$(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial y}$$

$$(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \frac{\partial y}{\partial x}$$

We will indicate forward pass with the black arrows and backward pass with the blue. We backpropagate gradients from the output node, which is always $\mathcal{L}$ in our case, to each individual node, such that at each individual node we can read off the gradients with respect to each of these nodes. We're interested in gradients of $\mathcal{L}$ with respect to any variable in this computation graph and we're going to use two different colors here for indicating two different types of quantities. Red color for backpropagated gradients and blue color for local gradients, that are locally computed based on this assignment formulas (1) and (2).

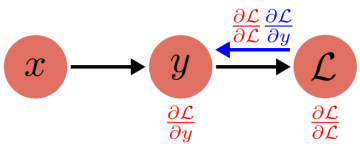


Figure 20: **Simple computation graph with backprop.**

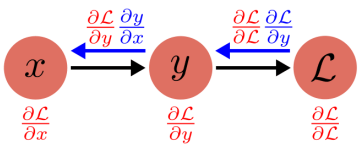


Figure 21: **Simple computation graph with backprop, step 2.**

The backward pass starts at the output node $\mathcal{L}$ and it computes gradients backwards.The first thing it does it computes the gradient of $\mathcal{L}$ with respect to itself which is obviously 1. In order to compute the gradient of $\mathcal{L}$ with respect to y we need to apply the chain rule. We need to compute the gradient of $\mathcal{L}$ with respect to the gradient of $\mathcal{L}$ times the gradient of $\mathcal{L}$ with respect to the gradient of y. Now we can do one step further,now

we're calculating the first expression the gradient with respect to the first variable, which we have in the forward pass computed first. Now we compute the gradient of $\mathcal{L}$ with respect to x, this is exactly what we wanted to compute. Again the chain rule, it's the gradient of $\mathcal{L}$ with respect to y times the gradient of y with respect to x.What we're ultimately interested in is to backpropagate gradients of course to the weights of a parametric function such as logistic regressor or a neural network. For all the values at a particular iteration we know the values , so we will we'll always insert the actual gradients and not the symbolic gradients.

Let's look at a slightly more difficult example.This is an example where we have a fan out bigger than one situation.Fan out means from a particular node there are multiple outgoing connections. In case y has multiple connections.

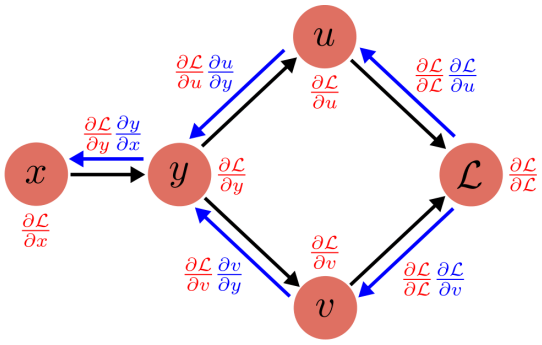


Figure 22: **Backpropagation: Fan-Out.**

Forward pass looks as follows:

$$
\begin{aligned}
&(1) \quad y = y(x) \\
&(2) \quad u = u(y) \\
&(2) \quad v = v(y) \\
&(3) \quad \mathcal{L} = \mathcal{L}(u, v)
\end{aligned}
$$

Lets compute the backward pass now:

$$
\begin{aligned}
&(3) \quad \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}}\frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial u} \\
&(3) \quad \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}}\frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial v} \\
&(2) \quad \frac{\partial \mathcal{L}}{\partial y} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial \mathcal{L}}{\partial v}\frac{\partial v}{\partial y} \\
&(1) \quad \frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial x}
\end{aligned}
$$

It's important to remember whenever we have such a fan out situation, whenever one variable sends to multiple variables we need to in the back propagation path is to sum up all the gradients and this is what has happened here.

2.3.3 Implementation.

Now we are basically ready to implement the backpropagation algorithm. A convenient way to do so is to consider each variable or node as an object that has certain attributes, for instance the value ($x.value$) and the gradient ($x.grad$). Values are computed in the forward pass. Here is the sequence of assignments in python:

$$
\begin{aligned}
&\texttt{x.value} = \texttt{Input} \\
&\texttt{y.value} = y(\texttt{x.value}) \\
&\texttt{u.value} = u(\texttt{y.value}) \\
&\texttt{v.value} = v(\texttt{y.value}) \\
&\texttt{L.value} = \mathcal{L}(\texttt{u.value}, \texttt{v.value})
\end{aligned}
$$

Tha value of x is an Input, value of y is function y evaluated at x value and then u value is the function u evaluated at y value and then v is also evaluated at y value. And then finally we have a Loss function.

Similarly we can look at the backward pass. First of all we would set the gradients to zero, because we'll gonna additively update.

$$
\begin{aligned}
&\texttt{x.grad} = \texttt{y.grad} = \texttt{u.grad} = \texttt{v.grad} = 0 \\
&\texttt{L.grad} = 1 \\
&\texttt{u.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial u)(\texttt{u.value}, \texttt{v.value}) \\
&\texttt{v.grad} \mathrel{+}= \texttt{L.grad} * (\partial \mathcal{L}/\partial v)(\texttt{u.value}, \texttt{v.value}) \\
&\texttt{y.grad} \mathrel{+}= \texttt{u.grad} * (\partial u/\partial y)(\texttt{y.value}) \\
&\texttt{y.grad} \mathrel{+}= \texttt{v.grad} * (\partial v/\partial y)(\texttt{y.value}) \\
&\texttt{x.grad} \mathrel{+}= \texttt{y.grad} * (\partial y/\partial x)(\texttt{x.value})
\end{aligned}
$$

In order to evaluate gradient we need the values that have been computed in the forward pass. The gradient itself is a function.
Let's look at a slightly more real example in the sense of machine learning. This is a logistic regression example. BCE denotes Binary Cross Entropy Loss.

Forward Pass:

$$
\begin{aligned}
(1) \quad & u = w_0 + w_1 x \\
(2) \quad & \hat{y} = \sigma(u) \\
(3) \quad & \mathcal{L} = \underbrace{-y \log \hat{y} - (1-y)\log(1-\hat{y})}_{\text{BCE}_{(\hat{y},y)}}
\end{aligned}
$$

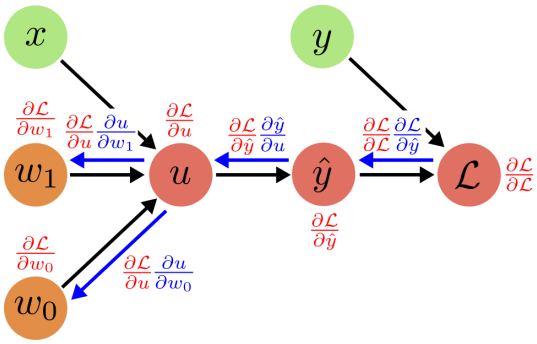


Figure 23: **Backpropagation: Log regression**

We want to calculate the gradients, but we're not interested in the gradients with respect to any variable in this graph, for instance we're not interested in the gradients with respect to these green input variables here, we could calculate them and sometimes we really do, like in the case of style transfer.That's the beauty about the back propagation algorithm that it can compute the gradients with respect to any variable can even compute the second derivative by executing it twice. But in this case here when we're interested in training the parameters of the logistic regression model. We're really just interested in back propagating the gradients from $\mathcal{L}$ to w_1 and w_0 so that we obtain the gradients of $\mathcal{L}$ with respect to w_1 and w_0. Which are then used in the gradient based optimization in an iteration loop in order to update the gradient the parameters of the model.Let's apply the back propagation algorithm to this example:

$$
\begin{aligned}
(3) \quad & \frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}}\frac{\partial \mathcal{L}}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \\
(2) \quad & \frac{\partial \mathcal{L}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial u} = \frac{\partial \mathcal{L}}{\partial \hat{y}}\sigma(u)(1-\sigma(u)) \\
(1) \quad & \frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial u} \\
(1) \quad & \frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial u}x
\end{aligned}
$$

2.3.4 Summary

We can write mathematical expressions as a computation graph, that decomposes complex expressions into much simpler computations that are tractable and where we can apply efficient dynamic programming to solve

for both - the values in the forward pass and the gradients in the backward pass.
The values are efficiently computed in the forward pass and the gradients are computed in the backward pass.
Multiple incoming gradients are summed up (multivariate chain rule).
Modularity: Each node must only "know" how to compute gradients wrt. its own arguments.
One fw/bw pass per data point:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{y}, \mathbf{X}, \mathbf{w}) = \sum_{i=1}^{N} \underbrace{\nabla_{\mathbf{w}} \mathcal{L}(y_i, \mathbf{x}_i, \mathbf{w})}_{\text{Backpropagation}} \tag{21}$$

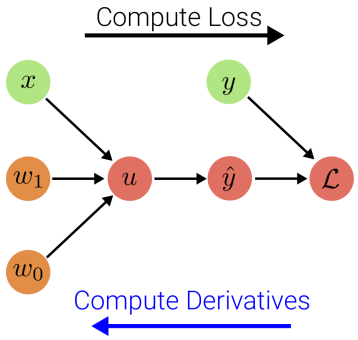


Figure 24: **Backpropagation: forward-backward**

2.4 Educational Framewotk.

Now we're going to introduce the educational framework, which is 160 lines of Python-NumPy code that implement a deep learning framework. It has been established in the context of a class taught by David McAllester, one of the pioneers of artificial intelligence at TTI Chicago. This educational framework allows us to understand the inner workings of a deep learning framework in depth. When listening to a lecture or a talk it is often easy to miss the details that are required to get a good understanding of how the discussed concepts actually work and only by implementing these concepts oneself one gets a good understanding of what has actually been missed.The other reason why we're using this educational framework is that it's very compact it allows us to understand each single line, it is a little bit less than 150 lines of Python-NumPy code. Let's look at this educational framework a little bit in more detail. It's using python, which means that variables are bound to objects. So for instance we have variables that describe the input or the labels or that are parents simply of particular compute nodes in that computation graph or we have values, which are attributes of these variables. The different types of nodes in the computation graph are implemented as classes. We have input classes parameter classes and computational node classes. So for instance from the compute node we can inherit a class that's called sigmoid and that implements the functionality of the sigmoid non-linearity within a logistic regression model or deep neural network.

```
class Input:
        def __init__(self):
                pass

        def addgrad(self, delta):
                pass

class Parameter:
        def __init__(self,value):
                self.value = DT(value)
                Parameters.append(self)

        def addgrad(self,delta):
                self.grad += np.sum(delta, axis = 0)

        def UpdateParameters(self):
                self.value -= learning_rate*self.grad

class CompNode:
        def addgrad(self, delta):
                self.grad += delta
```

Here we can see the abstract definition of the base classes. The input class has an initialization function and a function for adding the gradients, which are empty, because we are not interested in adding gradients to the input variables and also because input variables don't depend on any other variables. The parameter class has a initialization function that copies and stores the value of that parameter in the class itself and appends these parameters to a parameter vector, because later on we'll need to sum up these parameters in order to get the computation result. It also has a function for adding up gradients, this function simply sums up all the gradients along the data dimension. This model computes the values in the forward pass and the gradients in the backward pass for all the data points simultaneously and this is for efficiency reasons, because it allows to utilize efficient matrix operations in python. The Parameter class also has an UpdateParameter function that takes a step into the gradient direction, this is what is utilized during gradient descent. The compute node function also has a function for adding gradients, because we also need to update the gradients of the compute nodes themselves. In order to execute the computation graph we need to define a forward and a backward function.

```
def Forward():
        for c in CompNodes: c.forward()

def Backward(loss):
        for c in CompNodes + Parameters:
        c.grad = np.zeros(c.value.shape, dtype = DT)
        loss.grad = np.ones(loss.value.shape)/len(loss.value)
        for c in CompNodes[::-1]:
                c.backward()

def UpdateParameters():
        for p in Parameters: p.UpdateParameters()
```

Forward function takes the inputs x and y and the current state of the parameters w and goes left to right through the computation graph in order to compute all the intermediate values all the way until the head node $\mathcal{L}$. We have stored all these computation nodes in a list CompNodes and we have stored them in a way such that they are stored left to right, which means that when we arrive at any particular node in the sorted list, then we know that all the previous nodes have already been computed, so we can utilize their values. Thus in the forward pass of the backpropagation algorithm we simply go through the sorted list of computation nodes and for each computation node apply the forward function that is implemented inside that computation node. For the backward pass we first set all the gradients of all the computation nodes and all the parameters to zero and then we go backwards. We go backwards starting from $\mathcal{L}$ and propagate the gradients backward iteratively calling the backward function of each compute node in order to send gradient updates to its parents. Then finally we have an UpdateParameters function that loops through a list of parameters and calls the UpdateParameter function for each parameter.

Remark: Forward() and Backward() functions compute the forward and backward passes respectively over the

entire data set and the reason for this is that matrix operations are very cheap, because they are implemented very efficiently in python, while if we would go over the data set with loops we would be much slower. This vectorization is very important to exploit, furthermore if we would have GPUs hardware available, then we could even parallelize this computation, because the forward pass of each individual data point is independent of each other data point and the backward pass of each individual data point is independent of the backward pass of our data points.

Lets look at the concrete example of the computation node of sigmoid function:

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

```
class Sigmoid(CompNode):
        def __init__(self,x):
                CompNodes.append(self)
                self.x = x

        def forward(self):
                bounded = np.maximum(-10,np.minimum(10,self.x.value))
                self.value = 1 / (1 + np.exp(-bounded))

        def backward(self):
                self.x.addgrad(self.grad * self.value * (1-self.value))
```

Here we can see the python definition of the sigmoid class, which inherits from the class computation node. It has three functions: an initialization function, a forward function and a backward propagation function. At initialization time we simply add the node itself to the list of computation nodes and we store the parent of that node in that class itself. $self.x = x$ means take the parent which is an input that could be another computation node let's say an affine computation node that does some computation and then inputs to the sigmoid function. In the forward function we calculate the sigmoid expression. So first we compute a bounded value of the value itself in order to avoid numerical problems and then we implement the sigmoid function. For the backward pass we implement derivative of sigmoid function, but what we do actually is we implement that function and multiply that function with the back propagated gradient, the gradient at the node itself and pass this as a message further on to the parent.
Remark: It's important to note that in this backward pass the gradient is sent to the parent node self.x.
Let's execute a concrete minimal example.

Execution Example:

- Load data $\mathbf{X}$ and labels $\mathbf{y}$
- Initialize parameters $\mathbf{w}^0$
- Define computation graph
- For all iterations do
  - Forward Pass
    $\mathcal{L}(\hat{y}_i = f_{\mathbf{w}}(\mathbf{x}_i), y_i)$
  - Backward Pass
    $\nabla_{\mathbf{w}}\mathcal{L}(\hat{y}_i, y_i)$
  - Gradient Update
    $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \sum_{i=1}^{N} \nabla_{\mathbf{w}}\mathcal{L}(\hat{y}_i, y_i)$

```
import edf

# data loading
edf.clear_compgraph()
x = edf.Input()
y = edf.Input()
x.value = Load(data)
y.value = Load(labels)

# initialization of parameters
params_1 = edf.AffineParams(nInputs,nHiddens)
params_2 = edf.AffineParams(nHiddens,nLabels)

# definition of computation graph
h = edf.Sigmoid(edf.Affine(params_1, x))
p = edf.Softmax(edf.Affine(params_2, h))
L = edf.CrossEntropyLoss(p, y)

# gradient descent
for i in range(iterations):
        edf.Forward()
        edf.Backward(L)
        edf.UpdateParameters()
```

3 Deep Neural Networks

3.1 Backpropagation with Tensors

So far, we have only discussed how the backpropagation algorithm works on functions of scalars, that is, functions like $y = \sigma(w_1 x + w_0)$. But what if we want to optimize parameters in a function like $\mathbf{y} = \sigma(\mathbf{A}\mathbf{x} + \mathbf{b})$, where we have to deal with matrices and vectors? Technically, this function can be completely broken down such that we have a computational graph consisting only of scalar operations. However, it is often much simpler to deal with vectors, matrices and tensors (n-dimensional matrices) directly. If we implement a computational graph composed of tensor-valued nodes, each node still has the attributes `value` and `grad`. The `value`-attribute of a node contains an array with the components of the tensor. Because the loss function $\mathcal{L}$ is still a scalar, all that is needed for backpropagation are the partial derivatives of the loss with respect to each component of a tensor. The `grad`-attribute thus simply contains the gradient, so `b.grad` stores $\nabla_{\mathbf{b}}\mathcal{L}$. Note that this means that `A.value` and `A.grad` have the same shape (similarly for all other nodes).

3.1.1 Backpropagation on Loops

How do the values and gradients get computed then? Let us take a look at the following example:

$$\mathbf{y} = \sigma(\underbrace{\mathbf{A}\mathbf{x}}_{=\mathbf{u}} + \mathbf{b})$$

Here, the intermediate variable $\mathbf{u}$ has been indicated. As already stated, the computational graph might also consist of scalar operations only and that is certainly one way to implement the computation of the `value` and `grad` attributes. The forward pass would look like this:

$$\begin{aligned}
\text{for i} \quad & \texttt{u.value}[i] = 0 \\
\text{for i,j} \quad & \texttt{u.value}[i] \texttt{ += } \texttt{A.value}[i,j] * \texttt{x.value}[j] \\
\text{for i} \quad & \texttt{y.value}[i] = \sigma(\texttt{u.value}[i] + \texttt{b.value}[i])
\end{aligned}$$

In the backward pass, first the gradients of the loss with respect to $\mathbf{y}$ (so $\nabla_{\mathbf{y}}\mathcal{L}$) are computed and stored in `y.grad`. Then, the gradients of $\mathbf{u}$ and $\mathbf{b}$ can be computed:

$$\begin{aligned}
\text{for i} \quad & \texttt{u.grad}[i] \texttt{ += } \texttt{y.grad}[i] * \sigma'(\texttt{u.value}[i] + \texttt{b.value}[i]) \\
\text{for i} \quad & \texttt{b.grad}[i] \texttt{ += } \texttt{y.grad}[i] * \sigma'(\texttt{u.value}[i] + \texttt{b.value}[i])
\end{aligned}$$

Finally, the gradients of $\mathbf{x}$ and $\mathbf{A}$ are evaluated:

$$\begin{aligned}
\text{for i,j} \quad & \texttt{A.grad}[i,j] \texttt{ += } \texttt{u.grad}[i] * \texttt{x.value}[j] \\
\text{for i,j} \quad & \texttt{x.grad}[j] \texttt{ += } \texttt{u.grad}[i] * \texttt{A.value}[i,j]
\end{aligned}$$

Here, the back-propagated gradients are shown in red and the locally computed gradients in blue.

In deep learning, this kind of implementation remains possible for higher order tensors. If the forward pass of a computational graph is defined as follows:

$$\begin{aligned}
\text{for h,i,j,k} \quad & \texttt{U.value}[h,i,j] \texttt{ += } \texttt{A.value}[h,i,k] * \texttt{B.value}[h,j,k] \\
\text{for h,i,j} \quad & \texttt{Y.value}[h,i,j] = \sigma(\texttt{U.value}[h,i,j]),
\end{aligned}$$

then the backward pass can also be implemented in terms of loops over indices:

$$\begin{aligned}
\text{h,i,j} \quad & \texttt{U.grad} \texttt{ += } \texttt{Y.grad}[h,i,j] * \sigma'(\texttt{U.value}[h,i,j]) \\
\text{h,i,j,k} \quad & \texttt{A.grad} \texttt{ += } \texttt{U.grad}[h,i,j] * \texttt{B.value}[h,j,k] \\
\text{h,i,j,k} \quad & \texttt{B.grad} \texttt{ += } \texttt{U.grad}[h,i,j] * \texttt{A.value}[h,i,k]
\end{aligned}$$

In practice, this is usually not how the training of a deep network is implemented. Instead, a method called *minibatching* is used.

3.1.2 Minibatching

In addition to too many indices being messy, writing loops manually can be very computationally inefficient. The languages that are mainly used for writing training loops for neural networks, such as Python or MATLAB, are not designed for efficient loops over arrays and can be very slow (mostly for type checking reasons). A technique called *vectorization* is often advantageous: Instead of looping over an array and applying an operation to each

scalar element, apply the operation to a whole vector (or matrix) at once. There are special libraries that provide this functionality (such as NumPy) and they can speed up the computation dramatically. Some libraries also add support for computation on a GPU. The goal is, that the bulk of the computation time should go to the actual calculation of floating-point numbers and not to things such as type checking.

In our context this means not taking a single observation $\mathbf{x}$ for an optimization step (forward pass, backward pass, parameter update), but a *minibatch* of N observations, stacked together in a matrix $\mathbf{X}$. The equation from above then becomes:

$$\mathbf{Y} = \sigma(\underbrace{\mathbf{X}\mathbf{A}}_{=\mathbf{U}} + \mathbf{B})$$

Here, each row in $\mathbf{X} \in \mathbb{R}^{N \times D}$ is one observation $\mathbf{x} \in \mathbb{R}^D$ and the bias vector $\mathbf{b} \in \mathbb{R}^M$ is broadcast to $\mathbb{R}^{N \times M}$. All values that depend on the input $\mathbf{x}$ now also have a batch index b:

$$\begin{aligned}
\text{for b,i} \quad & \texttt{U.value}[b,i] = 0 \\
\text{for b,i,j} \quad & \texttt{U.value}[b,i] \texttt{ += X.value}[b,j] * \texttt{A.value}[j,i] \\
\text{for b,i} \quad & \texttt{Y.value}[b,i] = \sigma(\texttt{U.value}[b,i] + \texttt{B.value}[i])
\end{aligned}$$

For the actual parameter update step, the computed gradients are averaged over one batch.

In a real implementation, vectorization would be used:

```
def forward(self):
    self.value = np.matmul(self.x.value,self.w.A.value) + self.w.b.value

def backward(self):
    self.x.addgrad(np.matmul(self.grad,self.w.A.value.transpose()))
    self.w.b.addgrad(self.grad)
    self.w.A.addgrad(self.x.value[:,:,np.newaxis] * self.grad[:,np.newaxis,:])
```

This is easier to read than for-loops with many indices, but might be harder to implement for NumPy-novices.

3.2 The XOR Problem

Recall now the logistic regression model from earlier:

$$\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x}) \quad \text{with} \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

Which problems could be solved with this classifier? Logistic regression defines a linear *decision boundary*:

$$\mathbf{w}^\top \mathbf{x} + w_0 = 0,$$

(see Fig. 25). Each point $\mathbf{x}$ is classified according to the side of this linear boundary that it lies on:

- Decide for class 1 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} > -w_0$
- Decide for class 0 $\Leftrightarrow \mathbf{w}^\top \mathbf{x} < -w_0$

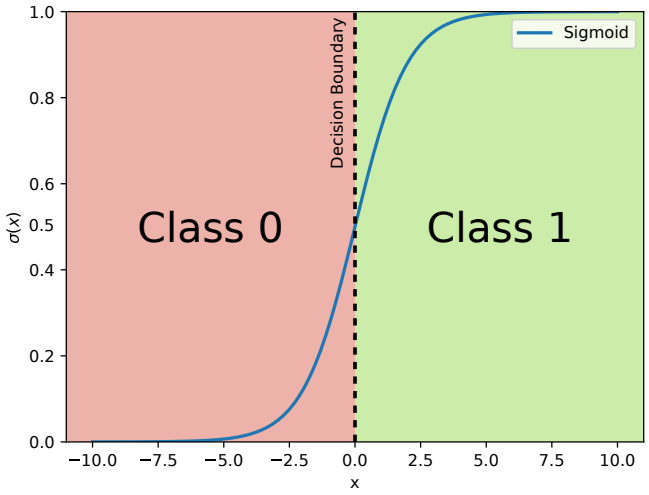


Figure 25: Decision boundary of a logistic regression model

Because the decision boundary is linear, logistic regression can only solve classification problems with *linearly separable* datasets. Some simple examples include the OR and AND functions:

x_1	x_2	OR(x_1,x_2)
0	0	0
0	1	1
1	0	1
1	1	1

x_1	x_2	AND(x_1,x_2)
0	0	0
0	1	0
1	0	0
1	1	1

For these two functions (datasets) optimal parameters can easily be found:

OR

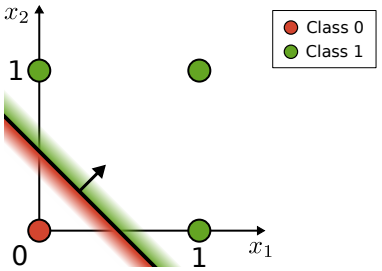
$$\underbrace{\begin{pmatrix}1 & 1\end{pmatrix}}_{\mathbf{w}^\top}\underbrace{\begin{pmatrix}x_1\\x_2\end{pmatrix}}_{\mathbf{x}} > \underbrace{0.5}_{-w_0}$$

AND

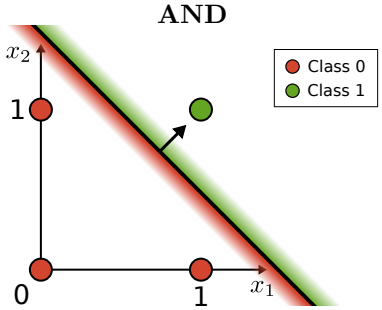
$$\underbrace{\begin{pmatrix}1 & 1\end{pmatrix}}_{\mathbf{w}^\top}\underbrace{\begin{pmatrix}x_1\\x_2\end{pmatrix}}_{\mathbf{x}} > \underbrace{1.5}_{-w_0}$$

From the following figures, it can be seen that these decision boundaries perfectly separate all four data points in both cases:

OR



AND



However, there are many datasets that are not linearly separable. One very simple example is the XOR-function:

x_1	x_2	XOR(x_1,x_2)
0	0	0
0	1	1
1	0	1
1	1	0

The dataset can be seen in Fig. 26. Visually, it is obvious that there exists no linear decision boundary that cleanly separates the two classes. Formally, this can be proven using convex sets.

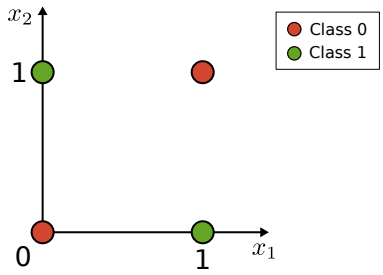


Figure 26: The XOR Dataset

Definition 1 (Convex Set) *A set S is called convex, if for any two points x, $y \in S$ the line segment connecting the two points also completely lies in S:*

$$\forall x,\, y \in S\ \forall \theta \in [0,1]\colon \theta x + (1-\theta)y \in S$$

The linear decision boundary divides the plane into two half-spaces (the decision regions). These half-spaces are convex. If a feasible hypothesis, i.e. a correct decision boundary, were to exist, the two red points must be in the red region, while the green points must lie in the green region. Because the decision regions are convex, the connecting line of the two green points must completely lie in the green decision region and the connecting line of the two red points must completely lie in the red decision region. However, as can be seen in Fig. 26,

the point $(0.5, 0.5)$ lies on both of these connecting lines (they intersect here). Thus it must lie in both decision regions, which is not possible. Hence, no solution can exist.

There are, of course, many problems that cannot be solved with a linear classifier. The XOR-Problem is particularly well known because of its simplicity. It is partially responsible for the decline of interest in perceptrons and neural networks in the 1970s.

The trick we need to use to solve the XOR-Problem (or other non-linear problems) using a linear classifier such as logistic regression, is to employ basis functions. Analogously to how polynomial basis functions allow a linear regression model to fit nonlinear datasets, the inputs of a classification problem can be transformed using basis functions in such a way that the dataset becomes linearly separable. The decision boundary no longer splits the input space of datapoints $\mathbf{x}$ in half, but the feature space of feature vectors $\boldsymbol{\psi}(\mathbf{x})$. One such feature mapping is defined in Tab. 1.

x_1	x_2	$\psi_1(\mathbf{x})$	$\psi_2(\mathbf{x})$	$\psi_3(\mathbf{x})$	XOR
0	0	0	0	0	0
0	1	0	1	0	1
1	0	1	0	0	1
1	1	1	1	1	0

Table 1: Feature mappings for the XOR problem

The decision boundary can then be defined as follows:

$$\mathbf{w}^\top \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_1 x_2 \end{pmatrix}}_{\boldsymbol{\psi}(\mathbf{x})} > -w_0$$

The feature space and the new decision boundary are visualized in Fig. 27.

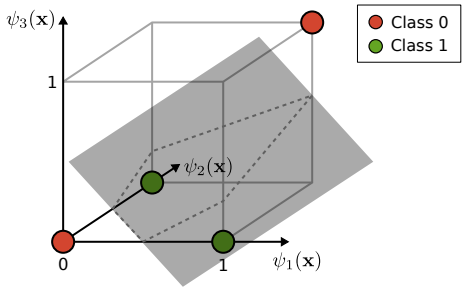


Figure 27: Feature space and decision boundary for the XOR problem

A different set of features that also transform the input into a linearly separable form are $\psi_1(\mathbf{x}) = \text{OR}(x_1, x_2)$ and $\psi_2(\mathbf{x}) = \text{NAND}(x_1, x_2)$. These features can then be combined using $\text{XOR}(x_1, x_2) = \text{AND}(\psi_1(\mathbf{x}), \psi_2(\mathbf{x}))$. So even though all of the functions OR, AND and NAND are linearly classifiable, they can compute XOR when composed together. The final computational graph for XOR can then be written as follows:

$$\begin{aligned} h_1 &= \sigma(\mathbf{w}_{OR}^\top \mathbf{x} + w_{OR}) \\ h_2 &= \sigma(\mathbf{w}_{NAND}^\top \mathbf{x} + w_{NAND}) \\ \hat{y} &= \sigma(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND}) \end{aligned}$$

Instead of going directly from input to output, the input gets transformed nonlinearly into the feature $\mathbf{h}$. Here, $\mathbf{h}$ is called a *hidden layer*. The equations can also be written more compactly:

$$\begin{aligned} \mathbf{h} &= \sigma\left(\underbrace{\begin{pmatrix} \mathbf{w}_{OR}^\top \\ \mathbf{w}_{NAND}^\top \end{pmatrix}}_{\mathbf{W}} \mathbf{x} + \underbrace{\begin{pmatrix} w_{OR} \\ w_{NAND} \end{pmatrix}}_{\mathbf{w}}\right) \\ \hat{y} &= \sigma(\mathbf{w}_{AND}^\top \mathbf{h} + w_{AND}) \end{aligned}$$

A visual depiction of this graph is shown in Fig. 28.

Because we now have two layers (one hidden and one output), this is called a *Multi-Layer Perceptron*. The real power of deep learning is that these intermediate features can be learned automatically using backpropagation. This is called *representation learning*.

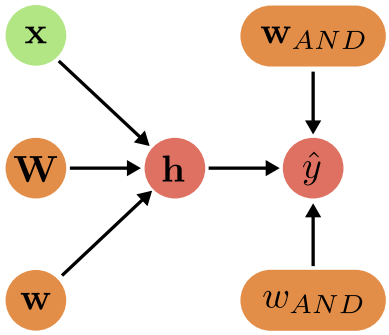


Figure 28: Solving the XOR Problem with a hidden layer

3.3 Multi-Layer Perceptrons

Multi-Layer Perceptrons (MLPs) are feedforward neural networks. This means there are no feedback signals flowing from later to earlier layers. MLPs compose several non-linear functions $\mathbf{f}(\mathbf{x}) = \hat{\mathbf{y}}(\mathbf{h}_3(\mathbf{h}_2(\mathbf{h}_1(\mathbf{x}))))$ where $\mathbf{h}_i(\cdot)$ are called *hidden layers* and $\hat{\mathbf{y}}(\cdot)$ is the *output layer*. The data with which a MLP is trained does not specify the representations learned by the hidden layers, only the behaviour of the output layer. Each layer i in a MLP comprises multiple neurons j which are implemented as affine transformations $(\mathbf{a}^\top\mathbf{x}+\mathbf{b})$ followed by non-linear activation functions (g):

$$h_{ij} = g(\mathbf{a}_{ij}^\top \mathbf{h}_{i-1} + \mathbf{b}_{ij})$$

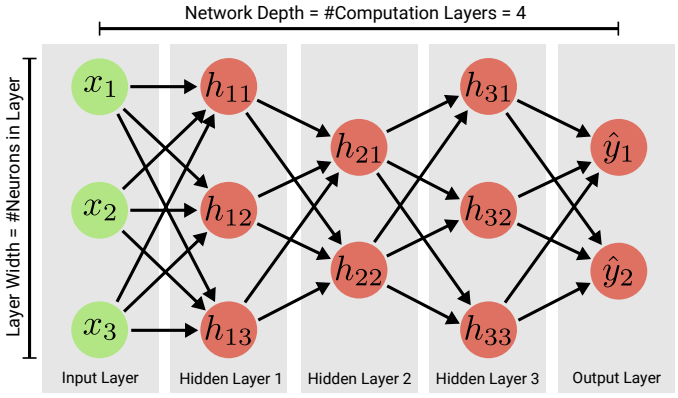


Figure 29: A multi layer perceptron with 4 layers

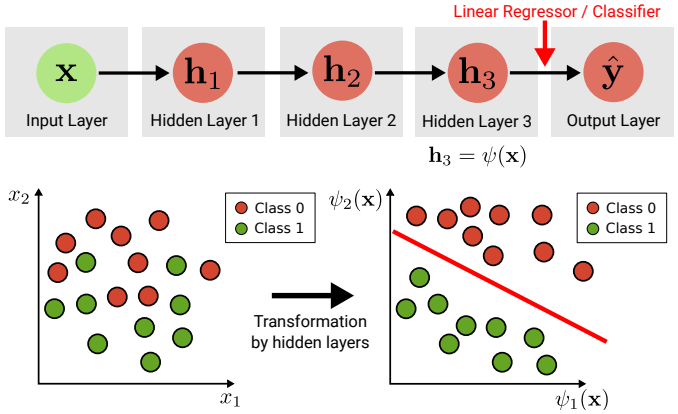


Figure 30: Hidden layers transform the input into better features

Each neuron in a MLP is *fully connected* to all neurons in the previous layer. The total number of layers is also called the *depth* of the model, this is where the name *deep learning* comes from. Technically, the neurons in a MLP are not perceptrons. Perceptrons use a linear threshold activation function which cannot be trained using backpropagation and is instead trained using the perceptron algorithm. MLPs use (mostly) differentiable

activation functions and are trained using backpropagation. In Fig. 29, a complete MLP is shown. The only function of the hidden layers is to transform the input vectors into features that can be processed by the simple linear regression or logistic regression output layer, as was done manually in the last section. This is illustrated in Fig. 30. Some different activation functions that can be used are shown in Fig. 31.

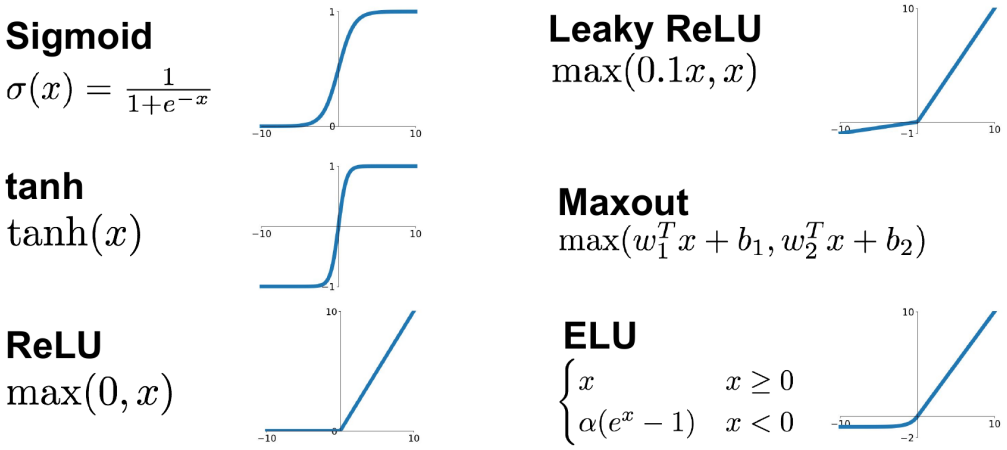


Figure 31: Activation functions that provide the needed nonlinearity

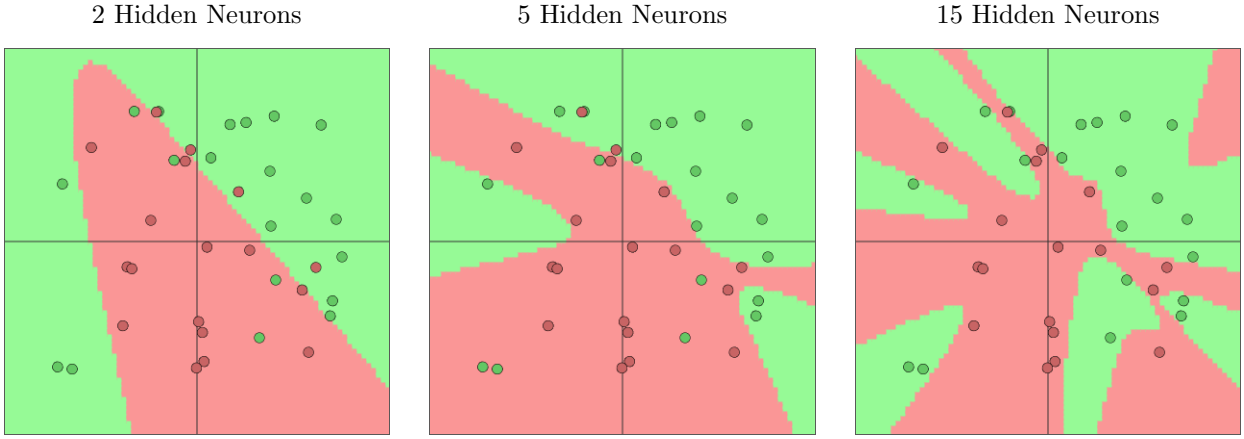
Artificial neural networks are loosely inspired by biological neural networks. Neurons in the brain are also structured in layers where one neuron has many inputs and computes one output. Even the sigmoid activation function can be found in biological neurons. However, brains are very different from Multi-Layer Perceptrons and the goal of deep learning is not to model the brain, but to build models that achieve good statistical generalization in many different settings.

MLPs can be trained using the backpropagation algorithm and (stochastic) gradient descent as follows:

1. Initialize weights $\mathbf{w}$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw (random) minibatch $\mathcal{X}_{\text{batch}} \subseteq \mathcal{X}$
3. For all elements $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}_{\text{batch}}$ of minibatch (in parallel) do:
   (a) Forward propagate $\mathbf{x}$ through network to calculate $\mathbf{h}_1, \mathbf{h}_2, \ldots, \hat{\mathbf{y}}$
   (b) Backpropagate gradients through network to obtain $\nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{|\mathcal{X}_{\text{batch}}|} \sum_{(\mathbf{x},\mathbf{y}) \in \mathcal{X}_{\text{batch}}} \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$
5. If validation error decreases, go to step 2, otherwise stop

Typically $|\mathcal{X}_{\text{batch}}| < |\mathcal{X}|$, because large datasets often do not fit into GPU memory.

The number of hidden layers and neurons per hidden layer are hyperparameters that can be selected to adjust the complexity of the model. In a two-layer MLP, the number of neurons in the hidden layer is simply the dimensionality of the feature space. The higher this number is, the more complex the function that the network represents can be. This is illustrated in the following figure[1]:



[1] https://cs.stanford.edu/people/karpathy/CNNjs/demo/classify2d.html

Finally, let us discuss what would happen if there are no non-linear activation functions. This following two-layer MLP

$$\mathbf{h} = g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = g(\mathbf{A}_2\mathbf{h} + \mathbf{b}_2)$$

can be written as

$$\mathbf{y} = g(\mathbf{A}_2\, g(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2).$$

If we used a linear activation function, such as $g(\mathbf{x}) = \mathbf{x}$, we can rewrite this equation:

$$\mathbf{y} = \mathbf{A}_2\,(\mathbf{A}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \mathbf{A}_2\mathbf{A}_1\mathbf{x} + \mathbf{A}_2\mathbf{b}_1 + \mathbf{b}_2 = \mathbf{A}\mathbf{x} + \mathbf{b}$$

Thus, with linear activations, a multi-layer network can only express linear functions.

3.4 Universal Approximation

If we use non-linear activation functions, what functions can we actually represent with a multi-layer perceptron? One important result tells us that we can approximate any continuous function arbitrarily well on a given bounded domain with only a 2-layer feedforward neural network.

Theorem 1 (Universal Approximation Theorem) *Let σ be any continuous discriminatory function. Then finite sums of the form*

$$G(\mathbf{x}) = \sum_{j=1}^{N} \alpha_j \sigma(\mathbf{a}_j^\top \mathbf{x} + b_j)$$

are dense in the space of continuous functions $C(I_n)$ on the n-dimensional unit cube I_n. In other words, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(\mathbf{x})$ for which

$$|G(\mathbf{x}) - f(\mathbf{x})| < \epsilon \quad \textit{for all} \quad \mathbf{x} \in I_n$$

Remark: This theorem has been proven for various activation functions (e.g., Sigmoid, ReLU).

If we restrict ourselves to the space of binary functions $f\colon \{0,1\}^D \to \{0,1\}$, it is not difficult to see why this result holds. Here, any function f is really just a table, such as the following:

x_1	x_2	x_3	y
⋮	⋮	⋮	⋮
0	1	0	0
0	**1**	**1**	**1**
1	0	0	0
⋮	⋮	⋮	⋮

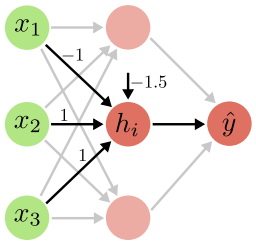


Figure 32: Linear threshold network

If we now take a linear threshold function as the activation function for each neuron, we can construct a 2-layer MLP with the output

$$\hat{y} = \sum_i \underbrace{[\mathbf{a}_i^\top \mathbf{x} + b_i > 0]}_{h_i}.$$

See Fig. 32 for an illustration. Here, each hidden neuron h_i recognizes exactly one combination of inputs, see Fig. 32 and the table above to see how the parameters of one such neuron are implemented. If we now do this for all 2^D combinations of inputs, we can construct a 2-layer MLP that can be equivalent to any binary function. We used a linear threshold function, which is not continuous. However, we can easily approximate this activation function using sigmoid functions, see Fig. 33

Thus, every binary function can be approximated arbitrarily well with a 2-layer MLP using 2^D number of hidden neurons. This is an important theoretical result, but an exponential number of neurons means that memory and computation time will also increase exponentially with the size of the input. Another problem with the approach from above is that the network only memorizes the input-output pairs and thus will not be able to generalize at all. By increasing the number of layers, a deep network needs far fewer parameters to learn complex functions. This is because a deep architecture introduces an inductive bias: it assumes that the

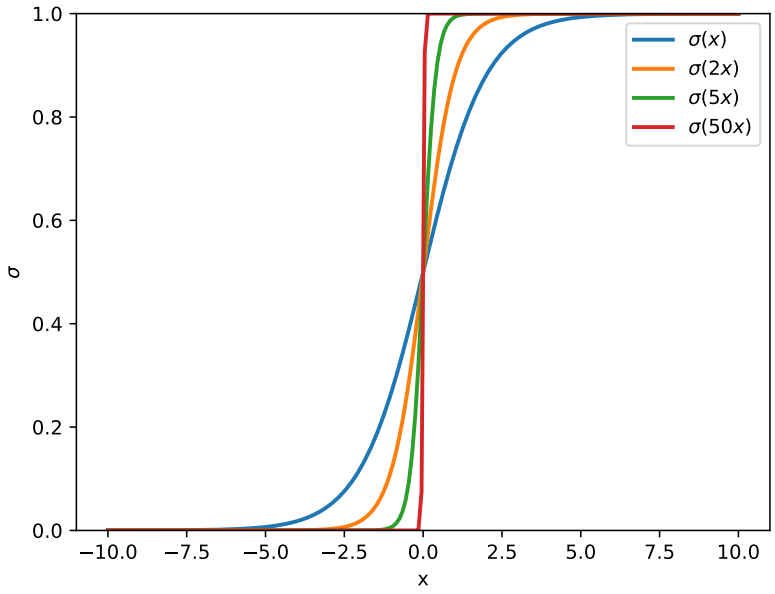


Figure 33: Sigmoid functions can approximate the Heaviside function

complex function learned is a composite of simpler functions. This leads to more compact models and better generalization performance. As an example, the parity function

$$f(x_1, \ldots, x_D) = \begin{cases} 1 & \text{if } \sum_i x_i \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

requires an exponentially large shallow network but can be computed using a deep network whose size is linear in the number of inputs D.

If the neurons of a MLP are activated with the *absolute value rectification function*, there is a intuitive geometric analog to how the layers of the network transform the input. The weights and biases of a layer define a hyperplane in the feature space which defines a "mirror". This means that the inductive bias of such networks is the assumption, that complex functions arise as mirrored images of simpler patterns. This is illustrated in Fig. 34. This image also intuitively conveys the exponential advantage of more layers over more parameters.

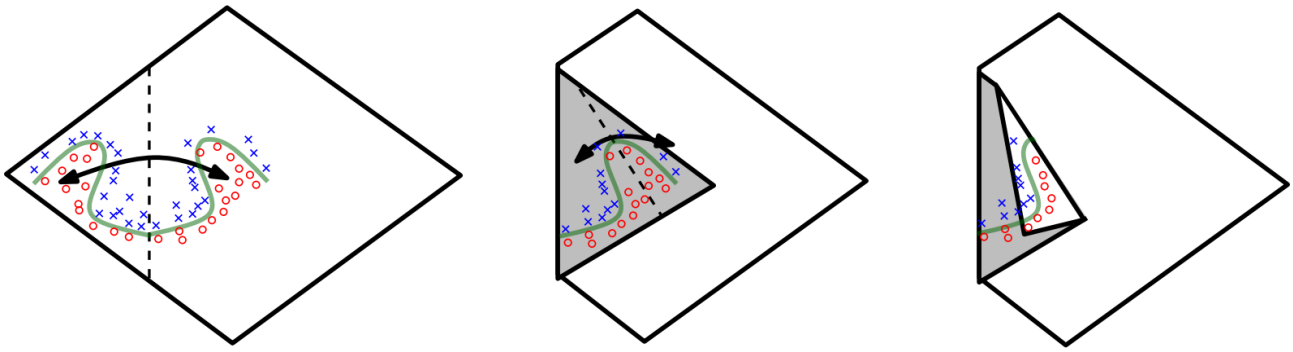


Figure 34: Space folding intuition for deep networks

It has been shown multiple times that deeper networks generalize better and that introducing more layers is often more effective than introducing more parameters, see Fig. 35. This means, that the inductive bias of compositionality is a very useful prior over the space of hypothesis functions that a model can learn.

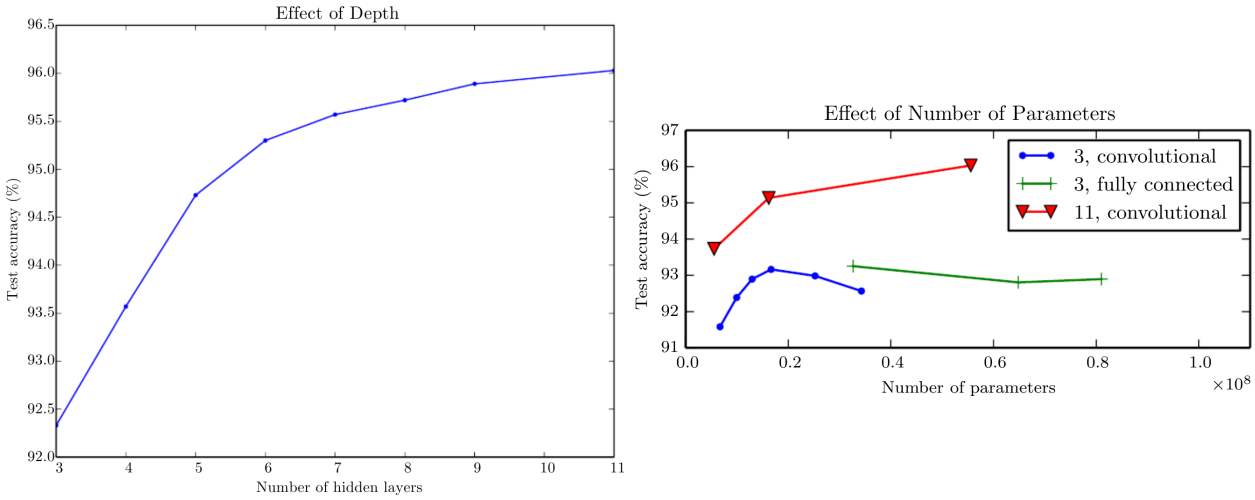


Figure 35: Deeper networks often perform better than shallow networks when using the same number of parameters

4 Deep Neural Networks II

4.1 Output and Loss Functions

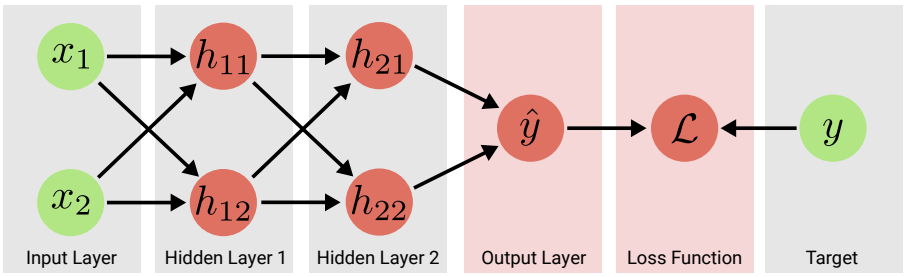
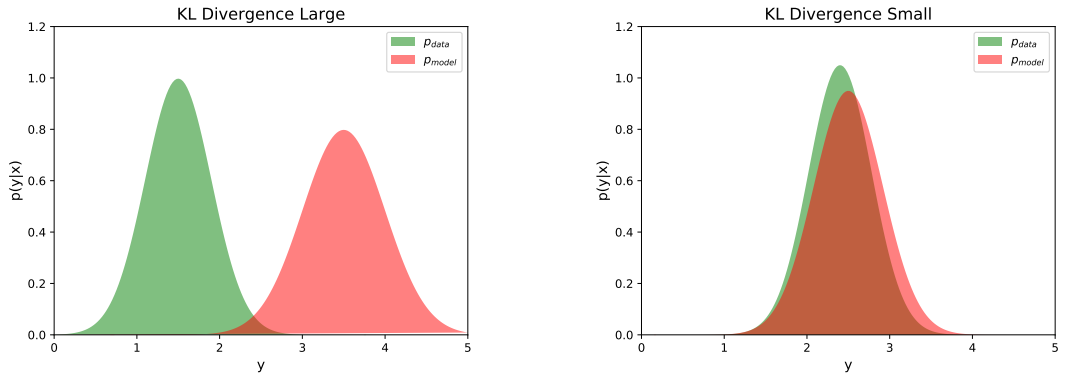


Figure 36: A simple neural network containing a one dimensional output layer and loss function.

Besides the input and hidden layers, an artificial neural network has an output layer containing the prediction $\hat{y}$ and a loss function $\mathcal{L}$ that compares the output with a given target y (see Fig. 36). The choice of the output layer depends on the task (discrete or continuous predictions, classification or regression problem).

4.1.1 Loss Function



(a) Large KL-divergence leads to a large loss value. (b) Small KL-divergence leads to a small loss value.

Figure 37: Quantifying the divergence of distributions.

The loss function of a neural network quantifies the divergence of model output (=prediction) and the target value. It evaluates the quality of a prediction by boiling it down to a single or few numbers. Predictions that are similar to the desired target value should get a small loss value, while strongly diverging predictions should be assigned a large loss value. In this sense you can think of the loss function as a measure of distance or cost being paid for a prediction. In this way the loss function enforces similarity between predictions and target values. Note that the values don't necessarily have to be single points, but might also be parameters of a probability distribution as visualized in Fig. 37. In the second case, the Kullback-Leibler divergence (KL divergence) serves as loss as it measures to divergence of to probability distributions.
But how to design a good loss function? Basically, a loss function can be any differentiable function that we wish to optimize. But instead of designing a loss function by hand, it is often preferable to derive the cost function from the **maximum likelihood principle**. To do so, consider the output of the neural network as **parameters of a distribution** over outputs y_i. The maximum likelihood principle is used to find the optimal values for the parameters by maximizing the likelihood function derived from the training data. So using this approach, we try to find model parameters $\mathbf{w}$ that maximize the likelihood of the model for a given training

dataset $\mathbf{X}$.

$$\begin{aligned}\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\text{argmax}}\ p_{model}(\mathbf{y}|\mathbf{X}, \mathbf{w}) \\ &\overset{\text{iid}}{=} \underset{\mathbf{w}}{\text{argmax}} \prod_{i=1}^{N} p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\ &= \underset{\mathbf{w}}{\text{argmax}} \underbrace{\sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w})}_{\text{Log-Likelihood}}\end{aligned}$$

For example, let a neural network $f_{\mathbf{w}}(\mathbf{x})$ predicts mean μ of Gaussian distribution over y, that is $p(y|\mathbf{x}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-f_{\mathbf{w}}(\mathbf{x}))^2}{2\sigma^2}\right)$. The goal is, to maximize the probability of the target y under this distribution which would result in the mean being shifted toward y. This setting is also visualized in Fig. 38.
As the loss function depends on the specific task at hand, the following two subsections will show how to derive the loss function for different regression and classification problems. We will also discuss the design of the output layer as both considerations are mutually dependent.

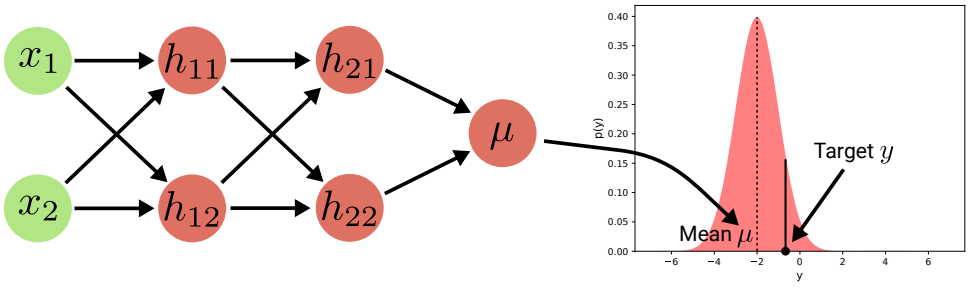


Figure 38: Neural network predicting model parameters μ, σ of a Gaussian distribution.

4.1.2 Regression Problems

The Gaussian distribution:

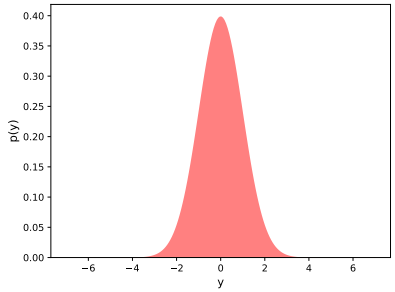


Figure 39: The Gaussian Distribution

The Gaussian Distribution (see Fig. 39 is parameterized by a mean μ and a standard deviation σ:

$$p(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$$

Its characteristics are thin tails: $p(y) \to 0$ quickly as $y \to \infty$. This means outliers are strongly penalized, wrong datapoints can impact the distribution significantly.

The $L2$ Loss
The formula of the $L2$ Loss is similar to the Maximum Likelihood in the first lecture, but now the parameter μ is the prediction of a neural network. We assume the model distribution is a Gaussian distribution where the Multi-Layer-Perceptron predicts the mean of that distribution, and σ is constant. When this is plugged into $\hat{\mathbf{w}}_{ML}$ (the Maximum Likelihood objective), the first expression can be removed, as it is constant with

respect to w. By reformulating the last expression by removing the minus, we are now minimizing instead of maximizing, so we have arrived at our loss function (see equation below).

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y-f_{\mathbf{w}}(\mathbf{x}))^2}{2\sigma^2}\right)$ be a **Gaussian distribution**. We obtain:

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\text{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\text{argmax}} -\sum_{i=1}^{N} \frac{1}{2} \log(2\pi\sigma^2) - \sum_{i=1}^{N} \frac{1}{2\sigma^2} \left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\text{argmax}} -\sum_{i=1}^{N} \left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2 \\
&= \underset{\mathbf{w}}{\text{argmin}} \sum_{i=1}^{N} \underbrace{\left(f_{\mathbf{w}}(\mathbf{x}_i) - y_i\right)^2}_{L_2 \text{ Loss}}
\end{aligned}$$

This loss function is called the squared loss or $L2$ Loss. As this loss is strongly affected by outliers, often a different distribution is used - the Laplace Distribution.

The Laplace distribution:

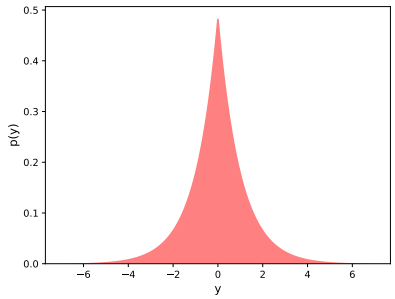


Figure 40: The Laplacian Distribution

The Laplace Distribution (see Fig. 40) has a similar form to the Gaussian distribution, except now in exponential expression the absolute difference between y and μ is used:

$$p(y) = \frac{1}{2b} \exp\left(-\frac{|y-\mu|}{b}\right)$$

The normalization constant $\frac{1}{2b}$ has changed slightly as well due to this. In the formula, μ denotes the location of the distribution, while b represents the scale, similar to the standard deviation in the Gaussian distribution determining the width of the distribution.
This distribution has heavier tails than the Gaussian: $p(y) \to 0$ more slowly as $y \to \infty$. This means more probability mass is at the tails, so outliers are penalized less strongly, which often makes it the preferred choice for regression problems in practice.

When deriving the loss here we do the same calculations as before, but now using the Laplacian Distribution. Here, the location parameter μ is predicted by the Feed-Forward Neural Network, with b being an arbitrary constant scale parameter.

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = \frac{1}{2b} \exp\left(-\frac{|y-f_{\mathbf{w}}(\mathbf{x})|}{b}\right)$ be a **Laplace distribution**. We obtain:

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} \log(2b) - \sum_{i=1}^{N} \frac{1}{b}|f_{\mathbf{w}}(\mathbf{x}_i) - y_i| \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} |f_{\mathbf{w}}(\mathbf{x}_i) - y_i| \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{|f_{\mathbf{w}}(\mathbf{x}_i) - y_i|}_{L_1 \text{ Loss}}
\end{aligned}
$$

Refactoring the equation in a similar way to the Gaussian, we arrive at a similar loss function too (see above). The difference is, that the square has been replaced with the absolute value, meaning that the **absolute loss** (called the $L1$ Loss)is being minimized. This loss is more robust than the $L2$ loss.

It is also possible to predict more than one parameter. Consider the Laplace Distribution again, but this time predicting both the location parameter μ and the scale b with a neural network. Technically these parameters are predicted by different neural networks $f_{\mathbf{w}}(\mathbf{x}_i)$ and $g_{\mathbf{w}}(\mathbf{x})$, but these are typicaly the same except for different output layer for each network.

Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = \frac{1}{2\, g_{\mathbf{w}}(\mathbf{x})} \exp\left(-\frac{|y - f_{\mathbf{w}}(\mathbf{x})|}{g_{\mathbf{w}}(\mathbf{x})}\right)$ be a **Laplace distribution**. We obtain:

$$
\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmax}} -\sum_{i=1}^{N} \log(2\, g_{\mathbf{w}}(\mathbf{x})) - \sum_{i=1}^{N} \frac{1}{g_{\mathbf{w}}(\mathbf{x})}|f_{\mathbf{w}}(\mathbf{x}_i) - y_i|
\end{aligned}
$$

The resulting expression is not as simple anymore, as both expressions depend on w and thus cant be removed. This also results in a more complicated loss function. This loss function is again derived from the maximum likelihood principle by assuming a certain distribution over the data. So if there exists knowledge of the distribution of the data, a loss function that fits can be derived that would be hard to specify otherwise.
Predicting both parameters allows for estimating the aleatoric uncertainty (observation noise) with the neutral network itself. This can be helpful for example in situations where an observation is not clear (for example an image where part of the lens was covered, that is too dark, etc.), the uncertainty about the prediction can be predicted too (larger or smaller b). The parameters can then be adjusted based on how certain the model is for a given prediction. Another example would be predicting the category/pixel depth of an image, where it is hard to make predictions at boundaries (foreground or background), resulting in higher uncertainty in these regions (see Fig. 41).

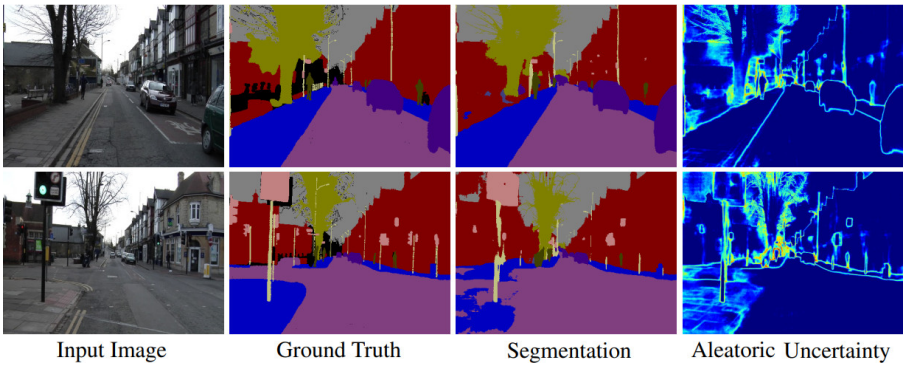


Figure 41: Example for predicting the aleatoric uncertainty

Mixture Density Networks All of the distributions considered so far were unimodal. In some cases like the depth prediction above, we may not know exactly if a pixel belongs to foreground or background. This is hard to model with a Laplacian or Gaussian distribution with a single peak. To represent multi-modal distributions,

we can also model **mixture densities** using a mixture model:

$$p_{model}(y|\mathbf{x}, \mathbf{w}) = \sum_{m=1}^{M} \pi_m \frac{1}{2\, g_{\mathbf{w}}^{(m)}(\mathbf{x})} \exp\left(-\frac{|y - f_{\mathbf{w}}^{(m)}(\mathbf{x})|}{g_{\mathbf{w}}^{(m)}(\mathbf{x})}\right)$$

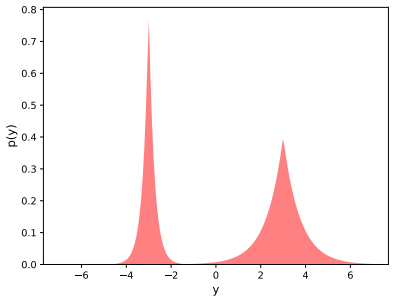


Figure 42: Laplacian Mixture Model

Fig. 42 above is a mixture model of two components using a simple Laplacian distribution that is summed over the number of components, resulting in a prediction for the location μ_m and scale b_m for all the m modes (in this case two). Here $\pi_m \in [0, 1]$ denotes the weight for each component, as all componentes are weighted relative to each other. A constraint for this parameter is further that $\sum_m \pi_m = 1$.
This model is called a mixture density network, with a mixture of Laplace distributions in this case.

The Output Layer for Regression Problems
It has been previously discussed, that the hidden layers of a neural network represent a simpl eaffine transformation plus an activation function (e.g. the Sigmoid).
The output layer is also a combination of its inputs, so we also have an affine transformation. This time we do not have an activation function though, as for most outputs (e.g., $\mu \in \mathbb{R}$), they are already going into the correct space. In this case the output is just a linear layer. For some outputs (e.g., $b \in \mathbb{R}^+$), we need a squashing function such as ReLU or softplus on top of the linear layer, to squash the otuput layer to $\mathbb{R}^+$.

Summary: Loss Functions for Regression Problems
In summary, when assuming a Gaussian or Laplacian model distribution, the loss function corresponds to the L2/L1 loss. It is also possible to predict uncertainty (variance/scale) or multiple modes using mixture density networks (MDN).

4.1.3 Classification Problems

Compared to regression problems, in classification a discrete outcome is predicted. This outcome can be binary or multi-class. The MNIST dataset for example, one of the most popular datasets in machine learning, contains handwritten digits, so ten different output classes. Each sample consists of one image of a single digit with 28x28 pixels each. In total the dataset contains 60k training samples with labels and 10k test samples.
The Curse of Dimensionality: For images with 28x28 pixels, there exist $10^2$36 different possible binary images, when using gray-scale such as in MNIST, there are even $256^7$84 different combinations. This means the image space is etremely large and impossible to enumerate. So how is this classification task with just 60k labeled images even possible then?
The answer is, that the images are concentrated on a low-dimensional manifold of this high-dimensional space. Only a tiny fraction of the possible images actually look like hand-written digits, so not the whole space has to be searched.

Bernoulli distribution: The Bernoulli distribution is a distribution over 2 classes, so it is only applicable for binary classification problems (e.g. classifying cats vs. dogs). It can be written as:

$$p(y) = \mu^y \, (1 - \mu)^{(1-y)}$$

Here, the parameter μ denotes the probability for $y = 1$. As before, the probabilities of both classes must sum to 1.
Similar as before, we can assume our model distribution as the Bernoulli distribution now and put this in the Maximum-Likelihood estimator to derive the loss function. Doing this we again obtain the **binary cross-entropy (BCE)** loss function (see below). Let $p_{model}(y|\mathbf{x}, \mathbf{w}) = f_{\mathbf{w}}(\mathbf{x})^y \, (1 - f_{\mathbf{w}}(\mathbf{x}))^{(1-y)}$ be a **Bernoulli**

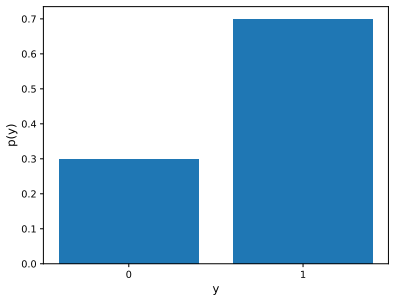


Figure 43: Bernoulli Distribution

distribution. We obtain:

$$\begin{aligned}\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\text{argmax}} \sum_{i=1}^{N} \log p_{model}(y_i|\mathbf{x}_i, \mathbf{w}) \\ &= \underset{\mathbf{w}}{\text{argmax}} \sum_{i=1}^{N} \log \left[f_{\mathbf{w}}(\mathbf{x}_i)^{y_i} \left(1 - f_{\mathbf{w}}(\mathbf{x}_i)\right)^{(1-y_i)} \right] \\ &= \underset{\mathbf{w}}{\text{argmin}} \sum_{i=1}^{N} \underbrace{-y_i \log f_{\mathbf{w}}(\mathbf{x}_i) - (1-y_i) \log(1 - f_{\mathbf{w}}(\mathbf{x}_i))}_{\text{BCE Loss}}\end{aligned}$$

In other words, maximizing the Log-Likelihood with the Bernoulli distribution as the model distribution is equivalent to minimizing the BCE loss. The last layer of $f_{\mathbf{w}}(\mathbf{x})$ can be a sigmoid function (or any other squeezing function) such that $f_{\mathbf{w}}(\mathbf{x})^y \in [0, 1]$ to get probabilities. In this case, unlike for regression problems, this is required as we have seperate classes.

Categorical distribution:
So how can this approach be scaled up to multiple classes, such as in MNIST?

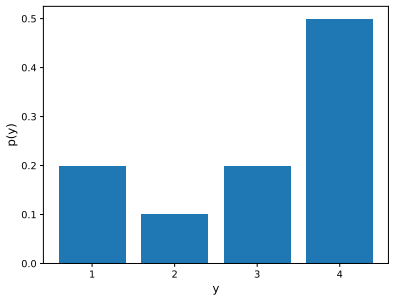


Figure 44: Categorical Distribution

In the case of multiple classes, we again use the Maximum-Likelihood principle, but this time with the Categorical distribution:

$$p(y = c) = \mu_c$$

Here, the probability of y taking any of the classes c is μ_c. Again, all probabilities have to sum to 1.
Discrete distributions such as this one by definition accomodate multiple modes.

Alternative notation:

$$p(\mathbf{y}) = \prod_{c=1}^{C} \mu_c^{y_c}$$

Consider the distribution not over one-dimensional categorical labels, but over a vector $\mathbf{y}$, which is a one-hot vector with $y_c \in \{0, 1\}$, where the length of $\mathbf{y}$ is the number of classes. Each element of the vector is either 1 or 0, but the sum over the whole vector is 1, so only one element can be 1 (the true class). For example $\mathbf{y} = (0, \ldots, 0, 1, 0, \ldots, 0)^\top$.

With this definition the probability can be rewritten as a product, where we have $\mu_c^{y_c}$. Only where $y_c = 1$ the term is not 1, but μ_c.
In the table below we see an example for this representation:

class	y	$\mathbf{y}$
	1	$(1,0,0,0)^\top$
	2	$(0,1,0,0)^\top$
	3	$(0,0,1,0)^\top$
	4	$(0,0,0,1)^\top$

In this example fouranimals are representing the diferent classes. Each class is represented by a one-hot vector $\mathbf{y}$ with binary elements $y_c \in \{0,1\}$, with an index c where $y_c = 1$ determines the correct class and with $y_k = 0$ for $k \neq c$.
This vector can also be interpreted as a discrete distribution, with all the probability mass at the true class.

The Categorical Distribution / CE Loss
Now we use the Categorical distribution as model distribution in the Maximum-Likelihood formulation:

Let $p_{model}(\mathbf{y}|\mathbf{x},\mathbf{w}) = \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x})^{y_c}$ be a **Categorical distribution**. We obtain:

$$\begin{aligned}\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log p_{model}(\mathbf{y}_i|\mathbf{x}_i,\mathbf{w}) \\ &= \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{i=1}^{N} \log \prod_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)^{y_{i,c}} \\ &= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \underbrace{\sum_{c=1}^{C} -y_{i,c} \log f_{\mathbf{w}}^{(c)}(\mathbf{x}_i)}_{\text{CE Loss}}\end{aligned}$$

Again, we can reformulate to turn the maximization problem into a minimization problem and arrive at the so called Cross-Entropy loss, which goes across all classes. We compare the target for class c (the entry in the one-hot vector) with the prediction for that c. This means we need a prediction for every c now. In other words, we minimize the **cross-entropy (CE)** loss.
The target $\mathbf{y} = (0,\ldots,0,1,0,\ldots,0)^\top$ is a one-hot vector with y_c its c'th element.

The Softmax
How can we ensure that $f_{\mathbf{w}}^{(c)}(\mathbf{x})$ predicts a **valid Categorical (discrete) distribution?**
The requirement for that is, that (1) each element $f_{\mathbf{w}}^{(c)}(\mathbf{x}) \in [0,1]$ and (2) $\sum_{c=1}^{C} f_{\mathbf{w}}^{(c)}(\mathbf{x}) = 1$, so the distribution has to sum to 1.
Using the element-wise Sigmoid for example, we would ensure (1), but not (2). This problem can be solved by defining a so called **softmax function** on top of the affine predictions (scores) for each class, which guarantees (1) and (2):

$$\text{softmax}(\mathbf{x}) = \left(\frac{\exp(x_1)}{\sum_{k=1}^{C} \exp(x_k)}, \ldots, \frac{\exp(x_C)}{\sum_{k=1}^{C} \exp(x_k)} \right)$$

So the exponential of each individual element is divided by the sum of the exponentials of each individual element.
Let $\mathbf{s}$ denote the network output after the last affine layer (=scores). Then:

$$f_{\mathbf{w}}^{(c)}(\mathbf{x}) = \frac{\exp(s_c)}{\sum_{k=1}^{C} \exp(s_k)} \quad \Rightarrow \quad \log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

So for each class we have the output of the neural network as input to the softmax. Taking the logarithm of the softmax we arrive at the Log Softmax function above. We already see that s_c is a direct contribution to the loss function, i.e., so it does not saturate.

The Log Softmax:

Intuition: Assume c is the correct class. Our goal is to maximize the log softmax:

$$\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k)$$

The first term here encourages the score s_c for the correct class c to increase. The second term encourages that all scores in $\mathbf{s}$ jointly decrease. It can be approximated by $\log \sum_{k=1}^{C} \exp(s_k) \approx \max_k s_k$ as $\exp(s_k)$ is insignificant for all $s_k < \max_k s_k$. Therefore, the loss always strongly penalizes the most active incorrect prediction. If it is the correct prediction (i.e., $s_c = \max_k s_k$), it is not penalized because both terms roughly cancel each out out, but it it's the wrong term then it is penalized.

Below we have an example for this:

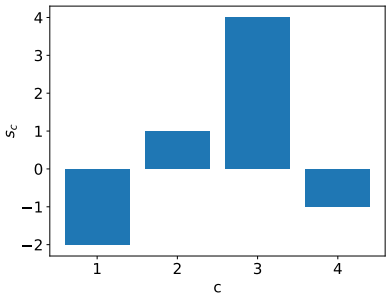


Figure 45: Scores s_c

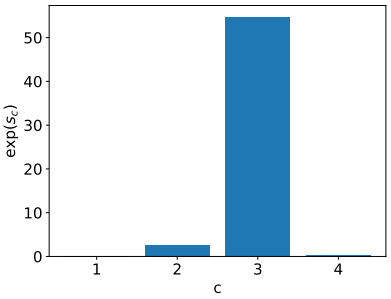


Figure 46: Exponential scores $\exp(s_c)$

On the left we see four classes and their scores predicted by the neural network. On the right we see the exponentials of these scores. All classes that are not top score classes are almost insignificant, when taking their exponentials. For example, the second term becomes: $\log \sum_{k=1}^{C} \exp(s_k) = 4.06 \approx s_3 = \max_k s_k$, so the value is very similar to the maximum.
Assuming $c = 2$ is the correct class, we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 1 - 4.06 \approx -3$. For $c = 3$ we obtain: $\log f_{\mathbf{w}}^{(c)}(\mathbf{x}) = s_c - \log \sum_{k=1}^{C} \exp(s_k) = 4 - 4.06 \approx 0$ So we get a much larger value, if the correct class corresponds to the class where we have assigned the highest score with our neural network, and a much lower value if we didn't assign the correct class.

The Relation between Softmax and Sigmoid:
When requiring the constraint that the used distribution sums to one is effectively removing one degree of freedom. Only $C - 1$ parameters are necessary, because the last one can be calculated as one minus the sum over the first ones.
Example: Consider $C = 2$ and fix one degree of freedom ($x_2 = 0$):

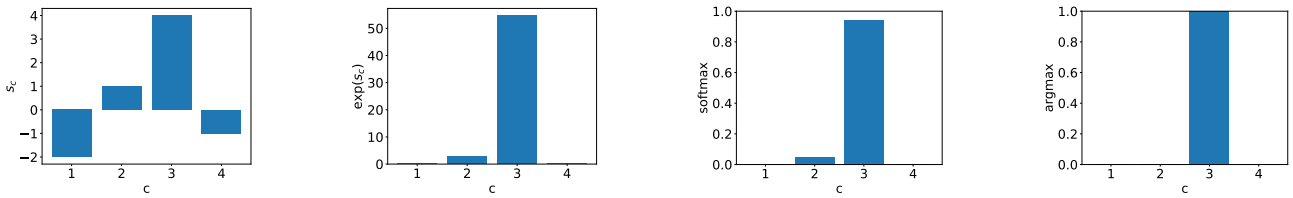
$$\begin{aligned}
\text{softmax}(\mathbf{x}) &= \left(\frac{\exp(x_1)}{\exp(x_1) + \exp(x_2)}, \frac{\exp(x_2)}{\exp(x_1) + \exp(x_2)} \right) \\
&= \left(\frac{\exp(x_1)}{\exp(x_1) + 1}, \frac{1}{\exp(x_1) + 1} \right) \\
&= \left(\frac{1}{1 + \exp(-x_1)}, 1 - \frac{1}{1 + \exp(-x_1)} \right) \\
&= (\sigma(x_1), 1 - \sigma(x_1))
\end{aligned}$$

As seen above, the softmax can be rewritten, such that the first expression is the expression of the Sigmoid function. This means that the Softmax is effectively a multiclass generalization of the sigmoid function.
In practice, the overparameterized version where all scores are predicted is often used, as it is simpler to implement and doesn't make a big difference.

The name Softmax is somewhat confusing, with "soft argmax" being a more precise name, as it is a continuous and differentiable version of the argmax function in one-hot representation:

$$\text{softmax}(\mathbf{s}) = \left(\frac{\exp(s_1)}{\sum_{k=1}^{C} \exp(s_k)}, \ldots, \frac{\exp(s_C)}{\sum_{k=1}^{C} \exp(s_k)} \right)$$

Example with 4 classes:



Here we first see the scores on the left, then the exponentials of these scores, then the softmax and then the argmax or one-hot encoding for category 3. As we can see, the softmax and the argmax are almost the same, so the softmax function can be regarded as an approximation of the argmax.

We have seen that the softmax responds to differences between inputs. It is also invariant to adding the same scalar to all it's inputs:

$$\text{softmax}(\mathbf{x}) = \text{softmax}(\mathbf{x} + c)$$

We can therefore derive a numerically more stable variant:

$$\text{softmax}(\mathbf{x}) = \text{softmax}(\mathbf{x} - \max_{k=1..L} x_k)$$

For all inputs, we subtract the maximum of all the inputs/scores of the softmax. This allows for accurate computation even with limited fixed precision, even when the x become large. It illustrates again, that the softmax depends only on the differences between individual scores and not on some global term that we add or subtract.

Cross Entropy Loss with Softmax Example:
The Cross Entropy Loss for a single training sample $(\mathbf{x}, \mathbf{y}) \in \mathcal{X}$ is:

$$\text{CE Loss: } \sum_{c=1}^{C} -y_c \log f_{\mathbf{w}}^{(c)}(\mathbf{x})$$

Example: Suppose we have 4 classes, so $C = 4$ and 4 training samples $\mathbf{x}$ with labels $\mathbf{y}$ represented in one-hot encoding:

Input $\mathbf{x}$	Label $\mathbf{y}$	Predicted scores $\mathbf{s}$	softmax($\mathbf{s}$)	CE Loss
	$(1,0,0,0)^\top$	$(+3,+1,-1,-1)^\top$	$(0.85,0.12,0.02,0.02)^\top$	0.16
	$(0,1,0,0)^\top$	$(+3,+3,+1,+0)^\top$	$(0.46,0.46,0.06,0.02)^\top$	0.78
	$(0,0,1,0)^\top$	$(+1,+1,+1,+1)^\top$	$(0.25,0.25,0.25,0.25)^\top$	1.38
	$(0,0,0,1)^\top$	$(+3,+2,+3,-1)^\top$	$(0.42,0.16,0.42,0.01)^\top$	4.87

Suppose we have a model making predictions (third column): For the first input it does a good job at predicting the correct class, for the second one it is uncertain between 1 or 2, for the third it is uncertain about all possible classes and for the last one the model actually predicts the wrong class.
After computing the softmax of these scores, we can then compute the Cross-Entropy loss. We can see, that the CE-loss is relatively small for the first example, as the the model assigned the correct class with high probability. For the second example, the CE-loss decreases, as the uncertainty increases. In the third it's increasing even more, as the uncertainty is also bigger. In the final example the model predicts the wrong class, resulting in a very high CE-loss. If this would be a minibatch in our stochastic gradient descent optimization, then sample 4 would contribute most strongly to the loss function.

Summary: Output Layer and Loss Function for Classification
For the output layer for two classes we can either predict a single value and use the Sigmoid function, or use the overparameterized representation and predict two values, applying the softmax afterwards to normalize the distribution. For more than two classes we use the general cross-entropy loss.

4.2 Activation Functions

After having focused on the output layer and loss function, we will now move on to the hidden layers. The parameters for the hidden layers are the number of hidden layers and the number of nodes in each of these hidden layers, but also the activation function used for the hidden layer.
A hidden layer can be defined as $\mathbf{h}_i = g(\mathbf{A}_i\mathbf{h}_{i-1} + \mathbf{b}_i)$ with a non-linear **activation function** $g(\cdot)$ and weights $\mathbf{A}_i, \mathbf{b}_i$. The activation function is frequently applied **element-wise** to its input, but not always. The activation

functions must be **non-linear** to learn non-linear mappings. Some activation functions are not differentiable everywhere, while still suitable for training.

The Sigmoid Activation Function
The first activation function we consider is the Sigmoid (see Fig. 47), which we have already seen in the context of logistic regression. It is defined as

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

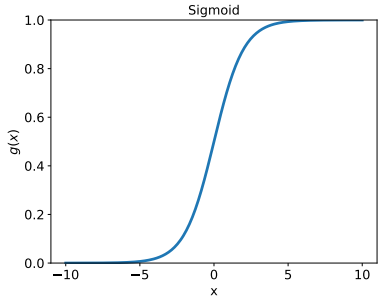


Figure 47: Sigmoid Activation Function

The Sigmoid maps the input to the range $[0, 1]$, which can be interpreted as a probability or analogously to the saturated "firing rate" of neurons in the brain.
It does come with some problems however: The saturation on the higher and lower ends "kills" the gradients, which leads to problems in backpropagation. The second problem is that the output is not zero-centered, but between 0 and 1, thus introducing a bias from the first layer on (positivity bias).

Problem 1: Saturation killing Gradients:
Consider the activation function in the context of its computation graph (see Fig. 48) with input x, activation

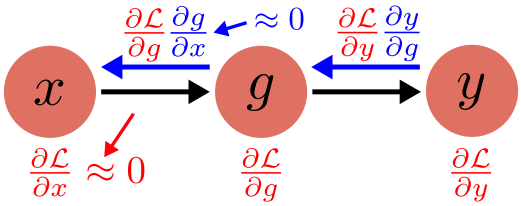


Figure 48: Sigmoid Problem 1

function g and output y.
When backpropagating we want to calculate the gradients of the loss function with respect to all nodes, so we are backpropagating gradients through the graph. If the input x to g is very small (< -10) or very large (> 10) we talk about the activation function being saturated. This means the gradient is almost zero: $g'(x) \approx 0$.
If this happens, the backpropagated gradient will stop, as the products will be close to 0.

Problem 2: The Sigmoid is not zero-centered:

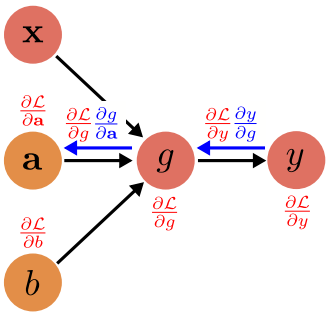


Figure 49: Sigmoid Problem 2

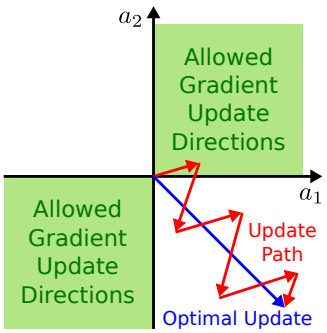


Figure 50: Sigmoid Problem 2: Inefficient Updates

Consider the sigmoid function, which is operating on a linear layer:

$$g(x) = \frac{1}{1+\exp(-x)} \quad x = \sum_i a_i x_i + b$$

We want to compute the loss function to the parameters a (see Fig. 49). If we have multiple hidden layers with a Sigmoid activation function for each of them, we know that the output of each hidden layer is always positive. Thus, the input to the next layer is also always positive. Furthermore, the gradient of the Sigmoid is also always positive. This means, that the gradient wrt. the parameters a_i is given by:

$$\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}\frac{\partial x}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i$$

As we know, all x_i are positive, and the gradient of the sigmoid is also always positive, the blue terms are all positive. Therefore, $\text{sgn}(\partial\mathcal{L}/\partial a_i) = \text{sgn}(\partial\mathcal{L}/\partial g)$, so all gradients have the same sign (+ or -).
The problem with this is, that it restricts the space in which gradient updates can happen and leads to effectively very inefficient optimization (see. Fig. 50). This problem becomes even bigger in higher-dimensional space, as the subset of possible directions to move into is even smaller relative to the full space. This problem can be somewhat alleviated by using minibatches.

The Tanh Activation Function:

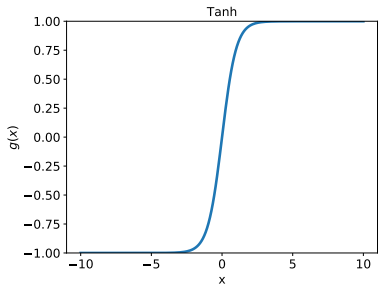


Figure 51: The Tanh Activation Function

To adress the problems of the Sigmoid function, the Tanh function has been proposed (see Fig. 51):

$$g(x) = \frac{2}{1+\exp(-2x)} - 1$$

It is generally pretty similar to the Sigmoid function, but the output range is fundamentally different. It maps the input to range $[-1, 1]$. This is an anti-symmetric mapping from the input domain to the output. The advantage of this is, that it makes the function zero-centered, so we have negative and positive output values. This alleviates problem 2 of the Sigmoid, but is also a saturating activation function, therefore also killing gradients.

The Rectified Linear Unit (ReLU) Activation Function:

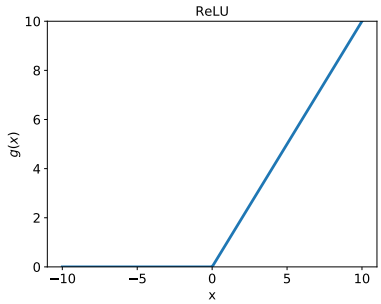


Figure 52: The ReLU Activation Function

Another activation function that has been propsed is the Rectified Linear Unit (see Fig. 52), which is the most commonly used activation function in practice. It is defined as:

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

It is called Rectified Linear Unit as we have a linear component, but all the values $x > 0$ are capped, so it does not saturate. It is not differentiable at 0, but that does not matter for training as we are not going to query that function exactly at 0.
This function typically leads to much faster convergence than the Sigmoid or Tanh activation functions, and is also computationally very efficient.
It is also not zero-centered however. Also, there is no learning for $x < 0 \Rightarrow$, so if the input becomes smaller than 0, then these ReLus become so called "dead ReLUs". It often happens in practice, that some neurons are dead and don't learn anymore.

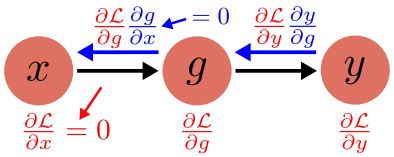


Figure 53: ReLU Problem: Dead ReLUs

As shown in Fig. 53, whenever we have an input value $x < 0$ this activation function is 0, so the downstream gradients are also going to be zero, so there is no learning. For this reason, we often initialize these ReLUs with a positive bias ($b > 0$).

The Leaky ReLU Activation Function:
One way to prevent dead ReLUs is to use the Leaky ReLU. It is defined as:

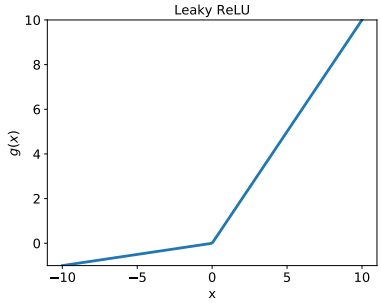


Figure 54: The Leaky Relu Activation Function

$$g(x) = \max(0.01x, x)$$

It is generally very similar to the normal ReLU, except that now there is a slope for all values $x < 0$ as well.
The advantage of this activation function is that it does not saturate as well, so the gradients won't die. It is

also closer to zero-centered outputs, leads to fast convergence and is computationally efficient.
The Parametric ReLU Activation Function;
There are more alternatives to the Leaky ReLU, for instance the Parametric ReLU as a generalization of the Leaky ReLU, where the factor α is itself a learnable parameter:

$$g(x) = \max(\alpha x, x)$$

It shares the advantages of the Leaky ReLU activation function.

The Exponential Linear Units (ELU) Activation Function:

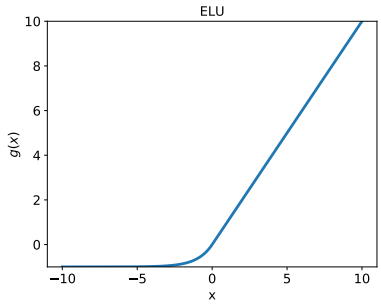


Figure 55: The Exponential Linear Units Activation Function

Another activation function that has been proposed is the Exponential Linear Units function (see Fig. 55). It is very similar to the Leaky ReLU again, with the difference that it is also differentiable at 0 and has a saturation for small x, which adds robustness to noise for some problems. Otherwise it shares the benefits with the leaky ReLU. Its default $\alpha = 1$.

The Maxout Activation Function:

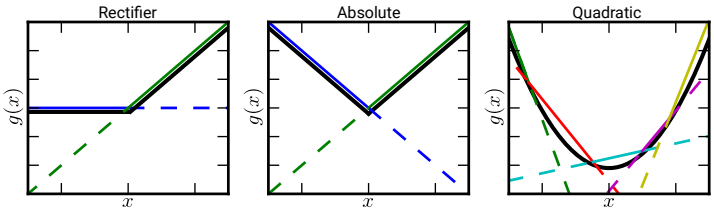


Figure 56: The Maxout Activation Function

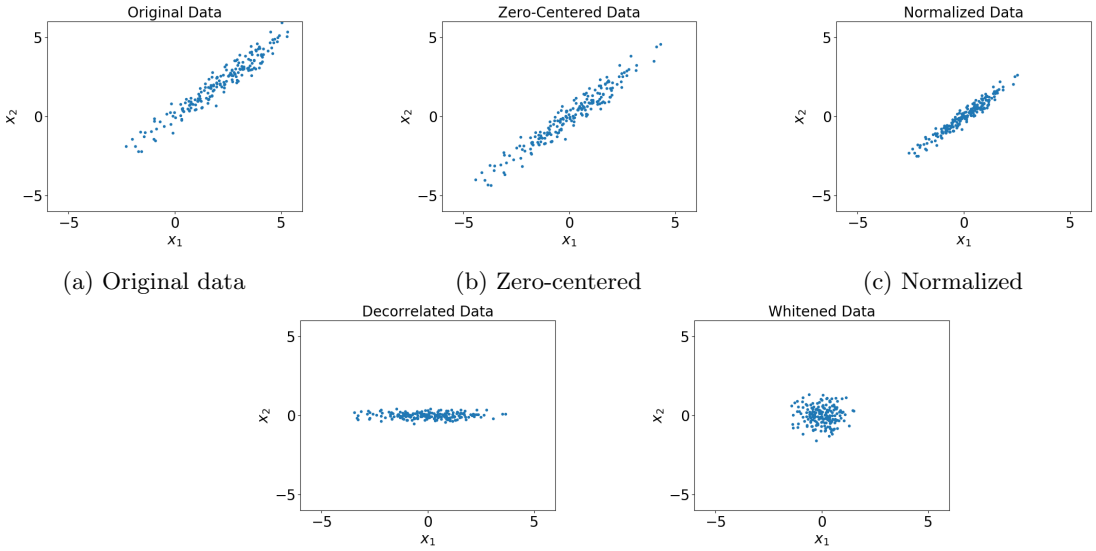
A generalization of the ReLU activation function is the Maxout activation function as proposed by Goodfellow et al.:

$$g(x) = \max(\mathbf{a}_1^\top \mathbf{x} + b_1, \mathbf{a}_2^\top \mathbf{x} + b_2)$$

The Maxout function is using multiple affine predictions as an input (for example two in the case above, $\mathbf{a}_1^\top$ and $\mathbf{a}_2^\top$. It thus increases number of parameters per function, which can be a disadvantage.

Summary Activation Functions:
There is no one-size-fits-all activation function, the choice highly depends on the problem. The activation functions discussed above only represent the most common ones, but there exist many more. The best activation function/model is often found using trial-and-error in practice. It is important to ensure a good "gradient flow" during optimization, so that the gradient flows backwards to all the parameters that we want to learn.

As a rule of thumb it is advisable to use ReLU with a sufficiently small learning rate by default, but Leaky ReLU, Maxout or ELU can be tried out for some small potential additional gain. In general, Tanh should be preferred over Sigmoid, and is often used in recurrent models.



(a) Original data (b) Zero-centered (c) Normalized

(d) Decorrelated (e) Whitened

Figure 57: The effect of different data preprocessing techniques.

Implementation of Activation Functions:
When implementing the backward pass of an activation, output or loss function it is important to ensure, that all the gradients are imlemented correctly. The default way to make sure the gradients are correct is to plot them and to verify them via computing numerical difference using Newton's difference quotient:

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Or the Symmetric difference quotient:

$$\frac{\partial f(x)}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

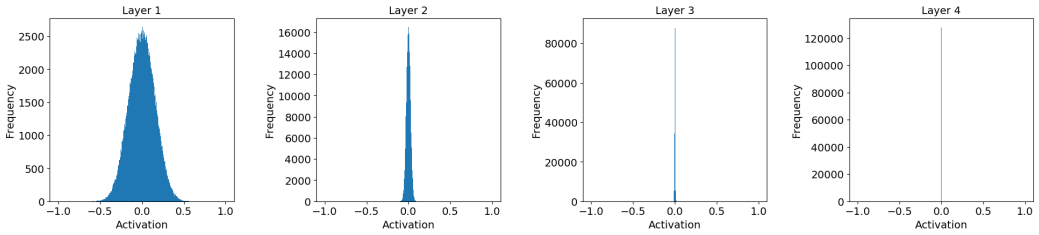
For a particular h we can evaluate this expression and compare it to the analytic gradient that we have implemented and verify it.
But how to choose h? For $h = 0$ the expression is undefined, so it has to be chosen bigger than 0 but small enough. When choosing h too small rounding errors may occur due to the finite precision of the data types. If its chosen too large we may get approximation errors because the approximation for the secant is wrong.
A good choice is usually $\sqrt[3]{\epsilon}$ with ϵ the machine precision. Examples are $\epsilon = 6 \times 10^{-8}$ for single precision (32 bit) and $\epsilon = 1 \times 10^{-16}$ for double precision (64 bit). (Example ommitted due to space constraints).

4.2.1 Data Preprocessing

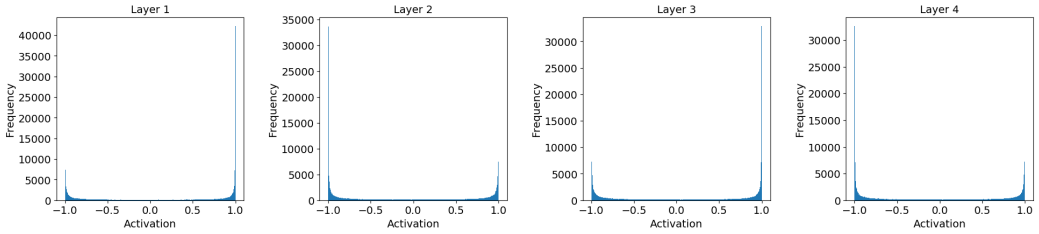
Data preprocessing is an integral part of the machine learning pipeline that aims to standardize the input data such that it is "well distributed" in the input space. This terms also refers to considerations about how to handle missing or inconsistent data and different data formats. Those aspects will not be discussed at this point. As we have seen before neural networks can be sensitive to differences in the magnitude of input features. Furthermore, preprocessing steps like centering serves numerical stability as it is the case for the softmax function.
The most common preprocessing techniques are zero-centering and normalization. **Zero-centering** refers to subtracting the feature-wise mean from all data points:

$$x_{i,j} \leftarrow x_{i,j} - \mu_j \text{ with } \mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

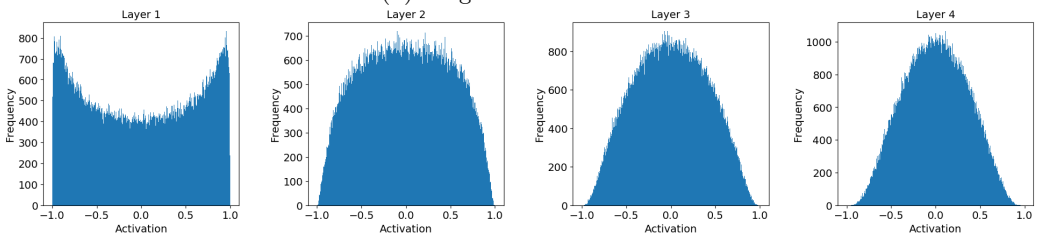
The result of zero-centering is visualized in Fig. 57b. The original data is shifted such that the mean is 0 for every feature. While being one of the most commonly applied preprocessing steps, note that the exact implementation of centering can vary slightly in practice: For AlexNet the overall mean image ($W \times H \times 3$ numbers) is subtracted. VGGNet and ResNet compute a per-channel mean (mean along each channel: 3 numbers) that is subtracted from the training data for preprocessing. In ResNet, data points are additionally



(a) Small Random Values



(b) Large Random Values



(c) Xavier

Figure 58: The effect of different initialization choices. Tahn was used as activation function for all examples.

normalized using a per-channel standard deviation. **Normalization** refers to transforming the data to have unit variance. This can be achieved by dividing the data points by the feature-wise standard variance:

$$x_{i,j} \leftarrow x_{i,j}/\sigma_j \text{ with } \sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

The effect is depicted in Fig. 57c. Two further techniques that require to compute the eigenvalues of the covariance matrix are decorrelation (Fig. 57d) and whitening (Fig. 57e). Decorrelation refers to multiplying with the eigenvectors of the covariance matrix. Dividing by the square root of the eigenvalues of the covariance matrix is called whitening. Both, decorrelation and whitening, are less commonly used.

4.2.2 Weight Initialization

The typical training process for neural networks comprises weight initialization as first step to start the training. This already suggests that the initialization plays a crucial role in the models performance. Indeed, there are several strategies, some of which will be presented in the following. Efficient parameter initialization remains an active field of research up until today.

Constant initialization Naively, initializing all weights with a constant value (e.g. 0) seems to be an easily and practicable idea. Unfortunately, any constant initialization scheme will perform very poorly as it hinders learning drastically. In the forward pass all weights will have the same influence and thus receive the same correction signal in the backward pass. This will be the case throughout training and causes all neurons to learn the same, thus significantly reducing the expressiveness of the neural network.

Random initialization An alternative approach is to initialize the weights at random. However, even with random numbers it is important to choose them appropriately.
Small Random Numbers: In order to achieve initial weights not equal but close to zero, weights could be drawn from a Gaussian with small standard deviation (e.g. $\sigma = 0.01$). Unfortunately, this leads to the problem of vanishing gradients, which refers to an exponential decrease of the activation with deeper layers (see Fig. 58a). Since in backpropagation the error signal is multiplied with the respective activation, the signal that will be passed on to subsequent layers will get small at an exponential scale and deeper layer will no longer receive

meaningful gradients. This causes learning to be very slow or even diverge. Using the chain rule, this can be seen by setting the activation x_i to 0, which also causes the gradient to be 0: $\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}0 = 0$
Large Random Numbers: Reacting to the problem of vanishing gradients the next obvious idea to try would be to initialize the weights with large random numbers, i.e. draw them independently from a Gaussian with large standard deviation (e.g. $\sigma = 0.2$). This however can cause exploding gradients as the backpropagated gradients will be multiplied with large values in every layer and can cause the network to oscillate. When using saturating activation functions large random weights cause all activation functions to saturate such that no meaningful gradient can be backpropagated which again hinders learning (see Fig. 58b). Looking at the chain rule, this becomes by setting the local gradient $\frac{\partial g}{\partial x}$ to 0 (saturated activation function): $\frac{\partial \mathcal{L}}{\partial a_i} = \frac{\partial \mathcal{L}}{\partial g}\frac{\partial g}{\partial x}x_i = \frac{\partial \mathcal{L}}{\partial g}\,0\,x_i = 0$

Xavier initialization A more sophisticated initialization strategy is Xavier initialization which was proposed by Glorot et al. [6]. The main idea is to adapt the magnitude of the initial weights to the input size of the respective layer. The authors propose to draw weights independently from a Gaussian distribution with $\sigma^2 = 1/D_{in}$, where D_{in} denotes the dimension of the input to the respective layer. Note that this may vary across layers as their input size varies. A result of this initialization is shown in figure Fig. 58c. The choice of $\sigma = 1/\sqrt{D_{in}}$ can be motivated theoretically. Let us consider $y = g(\mathbf{w}^\top \mathbf{x})$ and assume that all x_i and w_i are independent and identically (i.i.d.) distributed with zero mean. Let further $g'(0) = 1$. Then:

$$\begin{aligned}
\mathrm{Var}(y) \approx \mathrm{Var}(\mathbf{w}^\top \mathbf{x}) &= D_{in}\,\mathrm{Var}(x_i\,w_i) \\
&= D_{in}(\mathbb{E}[x_i^2 w_i^2] - \mathbb{E}[x_i w_i]^2) && \text{plug in formula} \\
&= D_{in}(\mathbb{E}[x_i^2]\,\mathbb{E}[w_i^2] - \mathbb{E}[x_i]^2\,\mathbb{E}[w_i]^2) && x_i, w_i \text{ independent} \\
&= D_{in}\mathbb{E}[x_i^2]\,\mathbb{E}[w_i^2] && \text{zero mean} \\
&= D_{in}\,\mathrm{Var}(x_i)\,\mathrm{Var}(w_i)
\end{aligned}$$

Thus:

$$\mathrm{Var}(w_i) = 1/D_{in} \Rightarrow \mathrm{Var}(y) = \mathrm{Var}(x_i)$$

It is important to note that the Xavier initialization assumes zero centered activation function. This is particularly visible when comparing the resulting activations when using tanh versus ReLU as activation function. For tanh the activations distribution appears to be well scaled across all layers. For ReLU and variants, collapsing activations are observable for deeper layers. Thus Xavier initialization should be used for zero-centered action function. For ReLU activation functions, a slightly adapted initialization strategy was proposed by He et al. [7].

He initialization As ReLU is restricted to the positive real line, He et al. [7] adapt the Xavier initialization by doubling the variance. Weights are now drawn from a Gaussian with $\sigma^2 = 2/D_{in}$. For ReLU activation functions this leads to a well scaled activation distribution across all layers.

5 Regularization

5.1 Problem Statement

The primary goal of machine learning techniques (for e.g. polynomial curve fitting, image classification using MLP's) is to learn a model from the training data which achieves low generalization error i.e. learn a model which performs well on new previously unseen samples (test set: assumed to be drawn i.i.d. - independently and identically from the true data distribution) and not just on the training set.
Now that we have formalized what we want our training model to do, the important question still remains what should the capacity/ complexity of our training model be to achieve such a low generalization error ? The best way to answer this is to look at the bias-variance trade-off curve plotted against model complexity and to think of generalization error in terms of bias/variance. (Fig. 59)

If we choose to train a model with low capacity (lower number of parameters/ weights, low variance) then we risk underfitting (Fig. 60) i.e. the model is too simple, has **high bias** and can not even fit our training data properly let alone perform well on the test set.

Takeaway: Low variance + High bias = High generalization error

On the other hand if we chose a model with very high capacity (higher number of parameters/ weights, high variance) then we risk overfitting (Fig. 61) i.e. the model has **high variance** and has fit too well to our training

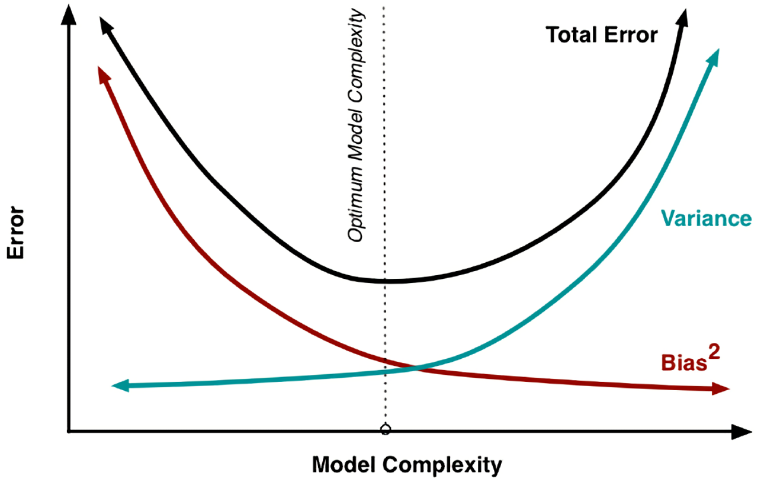


Figure 59: **Bias-Variance Trade-Off Curve**

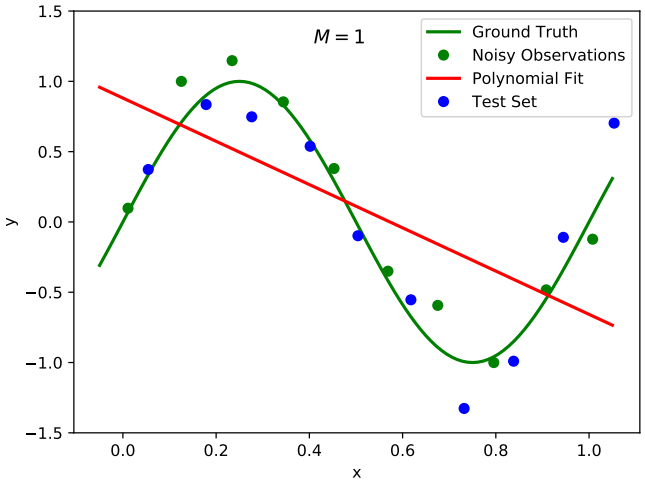


Figure 60: **Example of Underfitting** Generalization Error is High

data because of which it cannot generalize well to new unseen samples leading to high test error (=generalization error).

Takeaway: High variance + Low bias = High generalization error

From the curve(Fig. 59) we can see that the optimal model complexity for which our model would obtain the lowest total error is also when we have both **low bias** and **low variance**. Since at this optimal model capacity we have both low bias and low variance we can also be confident about obtaining low generalization error.

Takeaway: Low variance + Low bias = Low generalization error (desired)

For example: In the case of fitting polynomial functions we see in Fig. 62 that when the degree of the polynomial is 3 (optimal model complexity), both the variance and bias are low (because training set error is low) along with the lowest generalization error (test set error).

So how do we find this sweet-spot (optimum model complexity) empirically for which we have both low bias/-variance and low generalization error ? This is exactly where the technique of regularization figures in.

5.2 Intuition

We have two logical ways to empirically find the optimal model capacity. One way would be to start off with a low capacity model (high bias, high training error, underfitting regime) and iteratively increase its complexity (increase variance / reduce bias) till we obtain an optimal model which has the lowest generalization error and low training error. The other way would be the exact opposite, i.e. start off with a high capacity model (high

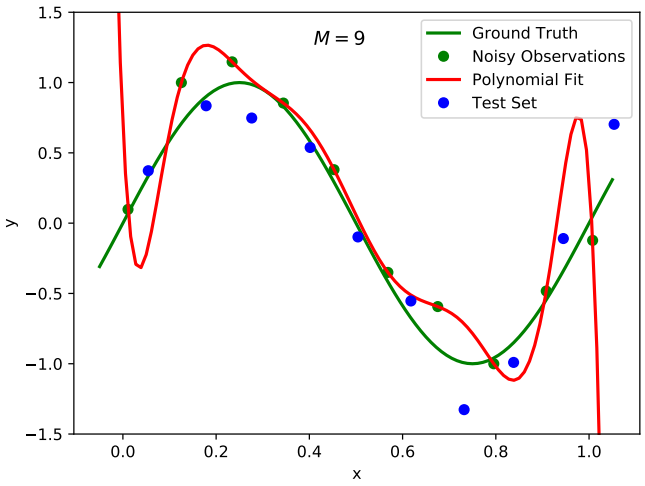


Figure 61: **Example of Overfitting** Training Error is Low but Generalization Error is High

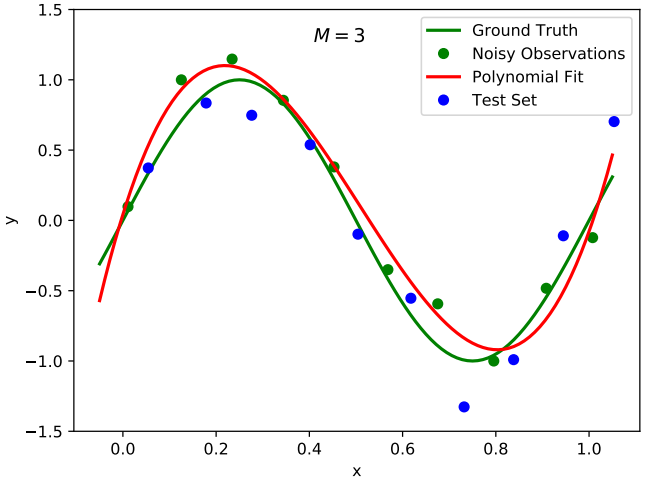


Figure 62: **Example of Good Fit** Generalization Error and Training Error are Low

variance, low training error, overfitting) and iteratively reduce its capacity (reduce variance / increase bias) to obtain an optimal model with lowest generalization error.
In practise, the second option is highly preferred purely for the fact that if we start with a low capacity model i.e. a model which underfits our data, we can never be sure about how much capacity needs to be added such that both training error and test error are low. Whereas in the second option we have one less variable to worry about i.e. the training error, as when we have a model which overfits our training data we can be very confident that reducing our model capacity by a bit will not affect the training error drastically.
Now that we have chosen the second option i.e. starting with a high capacity model, we have two ways to reduce the model capacity viz. either reduce the variance OR increase the bias of the model.
Important Note: As can be seen in the bias-variance trade-off curve(Fig. **??**), increasing bias would reduce variance and vice-versa. Our goal is to find the sweet spot where both variance and bias are low as not only would it lead to low total error but also low generalization error.
Again in practise, increasing the bias is highly preferred over reducing the variance of a model since there is no way to determine which and how many parameters/ weights need to be removed to decrease model capacity to the desired optimal capacity. Whereas there are many simple methods to increase the bias of a model. This is exactly what the concept of regularization is all about.(Fig. 63)

5.3 Concept

Regularization is a technique that trades increased bias for reduced variance of a model. When we apply regularization to a high capacity model which is overfitting the data (high variance, very low bias), the idea is that increasing the bias from a very low value to a slightly larger value would decrease the variance of the model as per the bias-variance trade-off i.e. reduce the model capacity without increasing its bias drastically.

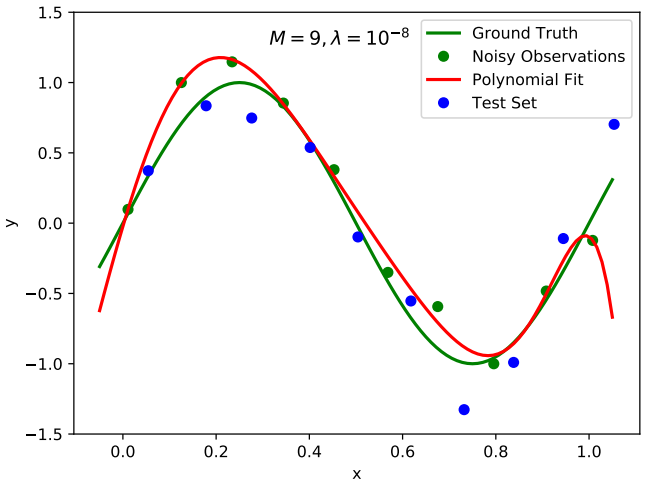


Figure 63: **A High Capacity Model after Regularization**

After applying regularization to an overfitting model, we hope to end up with an optimal model which has both low bias/ variance (=low generalization error) and low total error as is desired. In other words the goal of regularization is to minimize the generalization error when using large models (high complexity models).
Another good way to understand regularization is to visualize what it does in function space, as can be seen in the Fig. 64.

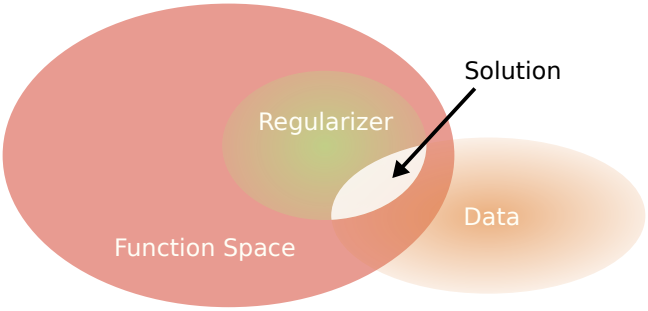


Figure 64: **Visualizing Regularization in Function Space**

1. In the figure, the set Function Space denotes the family of all possible functions that our model can represent.
2. The intersection between the Function Space and Data denotes the subset of functions within the function space which are constrained by the Data, i.e. the family of functions belonging to the function space of our model that maximize the likelihood (minimize negative log-likelihood loss) of the given data.

   (Note: Since we do not know what the true generative distribution of our data is, it need not completely overlap with the function space. Real-life data can be arbitrarily complex and it is very likely that it may not be able to be fully represented by even the most complex/ deep models. Our goal is to always find a good functional approximation which fits the data well.)
3. The Regularizer is an additional soft constraint on the function space and forces the model to represent functions which are close to the constraint boundary.
4. The solution that we want i.e. the optimal model lies within the intersection of the Function Space, Data and Regularizer.

In summary, both the Data and Regularizer should be viewed as constraints on the Function Space. The goal of introducing the regularizer is to encourage the model to learn a function which definitely fits the data well but most importantly is also not too complex.

5.4 Types of Regularization

There are many techniques to perform regularization, i.e. introduce bias into the model so that its variance can be reduced. Below are some of the most popular regularization techniques.

5.5 Parameter Penalties

Let $\mathcal{X} = (\mathbf{X}, \mathbf{y})$ denote the dataset and $\mathbf{w}$ the model parameters. (Note: In the case of a Multi-Layer Perceptron or a Deep Neural Network $\mathbf{w}$ is the vector which contains all the flattened weight matrices of each layer stacked together into a single column vector) We can **limit the end model capacity** by adding a parameter norm penalty $\mathcal{R}$ to the loss $\mathcal{L}$

$$\underbrace{\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w})}_{\text{Total Loss}} = \underbrace{\mathcal{L}(\mathcal{X}, \mathbf{w})}_{\text{Original Loss}} + \underbrace{\alpha \mathcal{R}(\mathbf{w})}_{\text{Regularizer}} \tag{22}$$

where $\alpha \in [0, \infty)$ controls the **strength of the regularizer.**

In the above loss, α is a hyper-parameter and has to be found empirically by performing cross-validation and then choosing a value which yields best generalization performance.
Important Note: The above loss is called the "*parameter*" penalty loss as the Regularizer $\mathcal{R}$ in the loss does not depend on the dataset $\mathcal{X}$ and penalizes only the parameter vector $\mathbf{w}$ i.e. the parameters of the model.

Some important points about the Parameter Penalty loss:

- $\mathcal{R}$ quantifies the size of the parameters / model capacity, as by penalizing the parameters the regularizer forces only a few parameters to be active effectively reducing the original model capacity.
- Minimizing $\tilde{\mathcal{L}}$ will decrease both $\mathcal{L}$ and $\mathcal{R}$. What this means is that we want to end up with a model which not only fits the data well (because $\mathcal{L}$ is minimized) but is also not too complex ((because $\mathcal{R}$ is minimized).
- Typically, $\mathcal{R}$ is applied only to the weights (not the bias) of the affine layers of the model. The two important reasons for doing this are
  - The bias term decides how far away the output of the affine transform layer should be from the origin. Adding the bias term to the regularizer would force the bias of each layer to be close to 0, effectively constraining the meaningful solution space and thus hampering the representational capacity of the model.
  - Since we typically only have one bias term per layer, the number of bias terms is minuscule when compared to the number of weight parameters, therefore not constraining the bias would make no significant difference to our optimization objective.
- Often, $\mathcal{R}$ drives weights closer to the origin (in absence of prior knowledge). Since typically no prior knowledge about the problem is available, constraining the weights to be close to 0 is the best possible practice. This is because smaller weights lead to less complicated solutions. Thus the regularizer $\mathcal{R}$ encourages the model to be as simple as possible.

5.5.1 Intuition behind Parameter Penalties

Why do we want the weights/inputs to be small?

- Suppose x_1 and x_2 are nearly identical.
  The following two networks make nearly the **same predictions:**

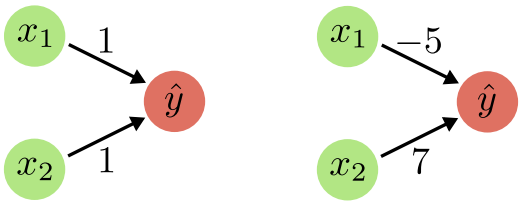


Figure 65: **Example to explain intuition behind penalizing weights to be small.**

- But the second network might predict wrongly if the test distribution is slightly different (x_1 and x_2 match less closely) ⇒ **Worse generalization**. The reasoning behind this statement is that even though

both networks give the same predictions the second network does this by using a complex model i.e. large weights, whereas the first network does the same using a less complex model i.e. small weights. Now if both these models were to see a sample which would be slightly different than the test distribution i.e. if the input varies by a bit, the difference in activations between the current sample and a sample from the test set would be extremely high for the second network given the large weights, possibly leading to wrong predictions. Whereas for the first network the difference in activations between the current sample and any sample from the test set would be lower given the small weights, very likely leading to a correct prediction. Thus, small weights also add some kind of robustness to small changes in the input to the first network consequently leading to better generalization performance.

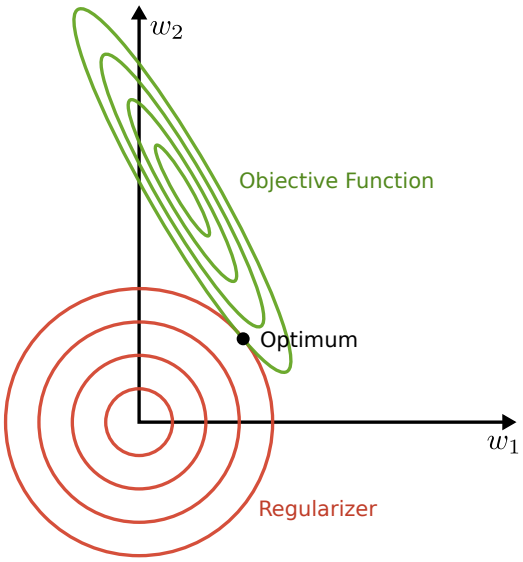


Figure 66: **Visualizing parameter penalties in parameter space.**

What does parameter penalization look like in parameter space? In the Fig. 66, the green contour plot is that of the objective function and the red contour plot is that of the regularizer. If there were no regularizer applied, ideally one would expect the Optimum point to be at the centre of the green contour plot, given how we would be minimizing just the original loss. However, when we minimize the loss $\underbrace{\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w})}_{\text{Total Loss}}$ in (22), we additionally also minimize the regularizer. Given this joint loss objective, the optimum now shifts to the current location so as to minimize the regularizer loss. As can be seen in Fig. 66, adding the regularizer forces the value of $w1$ to increase by a bit from the previous optimum at the centre of the green plot and the value of $w2$ to decrease significantly to the current location, so as to minimize the regularizer loss. Thus, these two competing losses i.e. the original loss and the regularizer force the optimum to shift to somewhere in-between the two plots.

5.5.2 L2 Regularization

Also known as **Weight decay** (=ridge regression, in the context of linear regression) uses an L_2 norm penalty $\mathcal{R}(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|_2^2$: (All variables below are the same as defined in (22))

$$\begin{aligned}\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w}) &= \mathcal{L}(\mathcal{X}, \mathbf{w}) + \alpha\, \mathcal{R}(\mathbf{w}) \\ &= \mathcal{L}(\mathcal{X}, \mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^\top \mathbf{w} \text{....(replacing } \mathcal{R}(\mathbf{w}) \text{ with the } L_2 \text{ norm)}\end{aligned}$$

The **parameter updates** during gradient descent are given by:

$$\begin{aligned}\mathbf{w}^{t+1} &= \mathbf{w}^t - \eta\, \nabla_{\mathbf{w}} \tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w}^t) \\ &= \mathbf{w}^t - \eta\, \left(\nabla_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w}^t) + \alpha\, \mathbf{w}^t\right) \\ &= (1 - \eta\, \alpha)\, \mathbf{w}^t - \eta\, \nabla_{\mathbf{w}} \mathcal{L}(\mathcal{X}, \mathbf{w}^t) \text{....(re-arranging terms)}\end{aligned}$$

,where
$\mathbf{w}^{t+1}$ are the updated weights obtained after performing a single gradient descent step on $\mathbf{w}^t$ and,
η is the step-size i.e. learning rate
Important Note: As can be observed from the final gradient descent update step, there is a slight difference here as compared to the normal gradient descent step done without L2 regularization. The first term in the L2 parameter update step is $(1-\eta\,\alpha)\mathbf{w}^t$ as opposed to just $\mathbf{w}^t$ used in the unregularized parameter updates. Now, since both $0 < \eta, \alpha < 1$, their product $\eta\,\alpha < 1$, therefore we know that $(1-\eta\,\alpha)$ is a number which is slightly lesser than 1. Since we are multiplying the weight vector $\mathbf{w}^t$ with a quantity lesser than 1 before each parameter update, we are effectively **decaying the weights** of the model at each training iteration before the gradient update while performing L2 regularization.
We have now seen what happens during a single gradient update step while using an L2 regularizer.

What happens while using an L2 regularizer over the the entire course of training?

Let $\mathbf{w}^* = \text{argmin}_{\mathbf{w}}\,\mathcal{L}(\mathcal{X}, \mathbf{w})$ denote the solution to the unregularized objective and consider a **quadratic approximation** $\hat{\mathcal{L}}$ of the unregularized loss $\mathcal{L}$ around $\mathbf{w}^*$. The quadratic approximation is a multi-variate Taylor series expansion around the optimum $\mathbf{w}^*$

$$\begin{aligned}\hat{\mathcal{L}}(\mathcal{X}, \mathbf{w}) &= \mathcal{L}(\mathcal{X}, \mathbf{w}^*) + \mathbf{g}^\top(\mathbf{w}-\mathbf{w}^*)_{\text{Linear Component}} + \frac{1}{2}(\mathbf{w}-\mathbf{w}^*)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}^*)_{\text{Quadratic Contribution}} \\ &= \mathcal{L}(\mathcal{X}, \mathbf{w}^*) + \frac{1}{2}(\mathbf{w}-\mathbf{w}^*)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}^*)\end{aligned}$$

with gradient vector $\mathbf{g} = \mathbf{0}$(since the gradient at the optimum $\mathbf{w}^*$ is 0) and semi-positive Hessian matrix $\mathbf{H}$ (second-order derivatives).
When including the **regularization term**, this approximation becomes:

$$\hat{\mathcal{L}}(\mathcal{X}, \mathbf{w}) = \mathcal{L}(\mathcal{X}, \mathbf{w}^*) + \frac{1}{2}(\mathbf{w}-\mathbf{w}^*)^\top\mathbf{H}(\mathbf{w}-\mathbf{w}^*) + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w}$$

The minimum $\tilde{\mathbf{w}}$ of the **regularized objective** $\hat{\mathcal{L}}(\mathcal{X}, \mathbf{w})$ is attained at $\nabla_{\mathbf{w}}\hat{\mathcal{L}}(\mathcal{X}, \mathbf{w}) = \mathbf{0}$:

$$\begin{aligned}\nabla_{\mathbf{w}}\hat{\mathcal{L}}(\mathcal{X}, \tilde{\mathbf{w}}) = \mathbf{H}(\tilde{\mathbf{w}}-\mathbf{w}^*) + \alpha\tilde{\mathbf{w}} &= \mathbf{0} \text{....}(\mathcal{L}(\mathcal{X}, \mathbf{w}^*) \text{ does not depend on } \mathbf{w}, \text{ thus its derivative wrt. } \mathbf{w} \text{ is 0}) \\ (\mathbf{H}+\alpha\mathbf{I})\tilde{\mathbf{w}} &= \mathbf{H}\mathbf{w}^* \\ \tilde{\mathbf{w}} &= (\mathbf{H}+\alpha\mathbf{I})^{-1}\mathbf{H}\mathbf{w}^*\end{aligned}$$

Thus, as α approaches 0, the regularized solution $\tilde{\mathbf{w}}$ approaches $\mathbf{w}^*$. Note: This is because when $\alpha = 0$, $\tilde{\mathbf{w}} = (\mathbf{H})^{-1}\mathbf{H}\mathbf{w}^* = \mathbf{I}\mathbf{w}^*$ This statement also intuitively makes a lot of sense, because as the influence of the regularizer approaches 0, the current optimum in Fig. 66 would start shifting back to the centre of the green contour plot i.e. the optimum of the unregularized objective.

What happens if α grows instead?
Consider the **eigen-decomposition $\mathbf{H} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ of the symmetric Hessian matrix** into
a diagonal matrix of eigenvalues $\mathbf{\Lambda}$ and an orthonormal basis of eigenvectors $\mathbf{Q}$
Note: The **Hessian matrix** is a symmetric matrix with real values, because according to Schwarz's theorem the partial derivatives of each element in the matrix can be swapped, making the matrix symmetric. Since the matrix is symmetric and real, we can apply spectral decomposition.

$$\begin{aligned}\tilde{\mathbf{w}} &= (\mathbf{H}+\alpha\mathbf{I})^{-1}\mathbf{H}\mathbf{w}^* \\ &= \left(\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top + \alpha\mathbf{I}\right)^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top\mathbf{w}^* \\ &= \left(\mathbf{Q}\left(\mathbf{\Lambda}+\alpha\mathbf{I}\right)\mathbf{Q}^\top\right)^{-1}\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top\mathbf{w}^* \text{.....(substitute } \mathbf{I} = \mathbf{Q}\mathbf{Q}^\top \text{ in above equation)} \\ &= \mathbf{Q}\left(\mathbf{\Lambda}+\alpha\mathbf{I}\right)^{-1}\mathbf{\Lambda}\mathbf{Q}^\top\mathbf{w}^* \text{.....(Apply } (\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1} \text{ in above equation, also } \mathbf{Q}^{-1} = \mathbf{Q}^\top)\end{aligned}$$

Since $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{H}$, we know that $(\mathbf{\Lambda}+\alpha\mathbf{I})^{-1}$ is also a diagonal matrix with its diagonal components being $\frac{1}{\lambda_i+\alpha}$ where λ_i is the i-th diagonal entry of $\mathbf{\Lambda}$, i.e. the i-th eigenvalue of the matrix $\mathbf{H}$.
With this in mind lets read the last equation from the R.H.S,
$\mathbf{Q}^\top\mathbf{w}^*$ signifies a change of basis of the vector $\mathbf{w}^*$ onto the eigen-basis of $\mathbf{H}$, i.e. every component of $\mathbf{w}^*$ is now aligned with the respective eigenvectors of $\mathbf{H}$.
Applying $(\mathbf{\Lambda}+\alpha\mathbf{I})^{-1}\mathbf{\Lambda}$ to $\mathbf{Q}^\top\mathbf{w}^*$ would thus mean that the component of $\mathbf{Q}^\top\mathbf{w}^*$ that is aligned with the i-th eigenvector of $\mathbf{H}$ is **rescaled** by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$.

Now multiplying this scaled vector by Q simply signifies a change of basis from the eigenbasis of $\mathbf{H}$ back to the original basis.
Thus, in summary what the additional term of L2 regularization does is that it takes every component of $\mathbf{w}^*$ that is aligned with the i-th eigenvector of $\mathbf{H}$ and **rescales** it by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$. From the previous term it is easy to see that Regularization affects directions with small eigenvalues $\lambda_i \ll \alpha$ the most.

Visual Representation of L2 Regularization

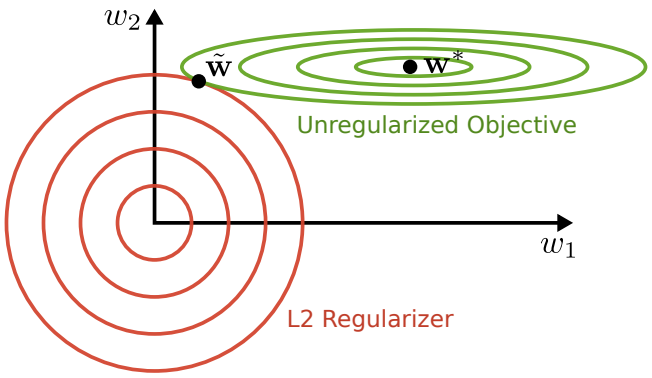


Figure 67: **Visualizing L2 regularization in parameter space.**

- Contours of unregularized objective $\mathcal{L}(\mathcal{X}, \mathbf{w})$ and L2 regularizer $\mathcal{R}(\mathbf{w})$
- At $\tilde{\mathbf{w}}$, the competing objectives **reach an equilibrium** (solution to regularized loss)
- Along w_1, eigenvalue of $\mathbf{H}$ is small (low curvature) $\Rightarrow$ **strong effect** of regularizer. Value changes much more in the direction of w_1 to reach $\tilde{\mathbf{w}}$
- Along w_2, eigenvalue of $\mathbf{H}$ is large (high curvature) $\Rightarrow$ **small effect** of regularizer. Value changes very less in the direction of w_2 to reach $\tilde{\mathbf{w}}$, as even small changes would incur large penalties.

5.5.3 L1 Regularization

The L1 regularization is another parameter penalization method to increase bias in a model. Unlike L2 regularization, the L1 regularizer $\mathcal{R}(\mathbf{w}) = \|\mathbf{w}\|_1$, tries to minimize the L1 norm of the weight vector $\mathbf{w}$ instead of minimizing its L2 norm. Since the regularization objectives for both regularizers are different, their contour plots are different as well as can be seen in Fig. 68 and Fig. 67.

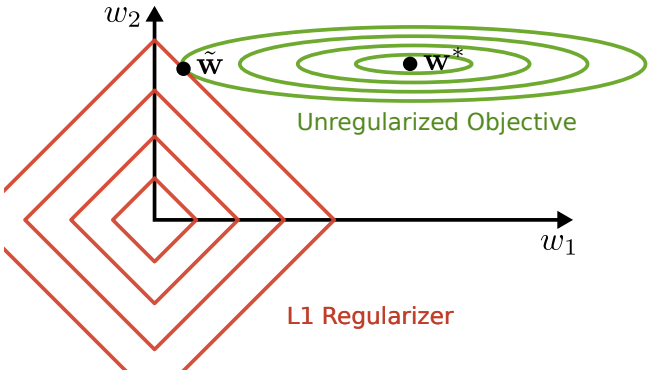


Figure 68: **Visualizing L1 regularization in parameter space.**

- Contours of unregularized objective $\mathcal{L}(\mathcal{X}, \mathbf{w})$ and L1 regularizer $\mathcal{R}(\mathbf{w})$
- At $\tilde{\mathbf{w}}$, the competing objectives **reach an equilibrium** (solution to regularized loss)
- L_1 regularized loss function: $\tilde{\mathcal{L}}(\mathcal{X}, \mathbf{w}) = \mathcal{L}(\mathcal{X}, \mathbf{w}) + \alpha\|\mathbf{w}\|_1$

- L_1 Regularization results in a solution which is more sparse (compared to L_2)
  Important Note: If we observe the contour plots of both the L1 regularizer (Fig. 68) and the L2 regularizer(Fig. 67) closely we can see that the optimum of the regularized objective $\tilde{\mathbf{w}}$ in the L1 case is closer to the origin along the direction of w_1 than it is in the case of L2. What this signifies is that the L1 regularizer tries to squeeze and set most of the parameters as close to 0 as possible (w_1 in the L1 example) and allots some amount of flexibility and freedom to the remaining parameters (w_2 in this example, which in-fact increased by a bit to reach $\tilde{\mathbf{w}}$ in the L1 case). Since the output weight vector after applying L1 regularization would consist of mostly 0's, it is said to encourage "**sparsity**" of solutions.

5.5.4 L2 vs. L1 Regularization

Example: Assume 3 input features: $\mathbf{x} = (1, 2, 1)^\top$
The following two **linear classifiers** $f_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ yield the **same result/loss:**

- $\mathbf{w}_1 = (0, 0.75, 0)^\top \Rightarrow$ ignores 2 features
- $\mathbf{w}_2 = (0.25, 0.5, 0.25)^\top \Rightarrow$ takes all features into account

But the L1 and L2 regularizer **prefer different solutions!**

L2 Regularization:

- $\|\mathbf{w}_1\|_2 = 0 + 0.75^2 + 0 = 0.5625$
- $\|\mathbf{w}_2\|_2 = 0.25^2 + 0.5^2 + 0.25^2 = \mathbf{0.375}$

L1 Regularization:

- $\|\mathbf{w}_1\|_1 = 0 + 0.75 + 0 = \mathbf{0.75}$
- $\|\mathbf{w}_2\|_1 = 0.25 + 0.5 + 0.25 = 1$

As we can see in the above example, the L2 regularizer prefers the model $\mathbf{w}_2$ where every feature is taken into account during activation computation, whereas the L1 regularizer prefers the model $\mathbf{w}_1$ which has sparse solutions, i.e. the model which gives importance to only one feature and ignores the other two.

A good way to visualize what the L1 and L2 regularizers do is to look at Fig. 69 and Fig. 70. In both these figures, we can think of the high level features in the orange box as being the output of the penultimate layer of our model. The model is now supposed to make a classification decision based on these high level features. The figures illustrate what solutions such a model trained with either an L2 or an L1 regularizer would prefer.

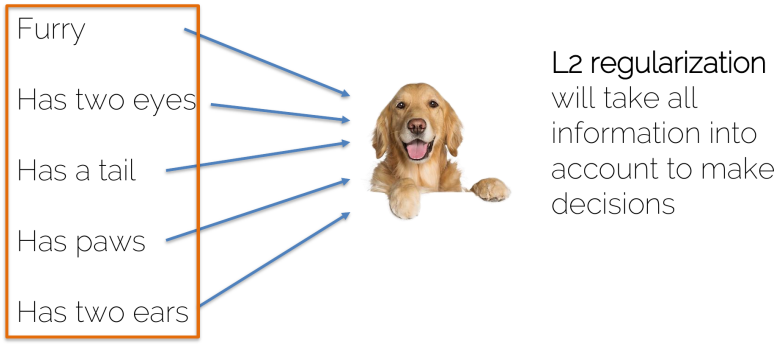


Figure 69: **Effect of L2 regularizer in high level feature space.** [9] The L2 regularizer prefers a model which takes all the input features into account, i.e. every feature will have a corresponding weight

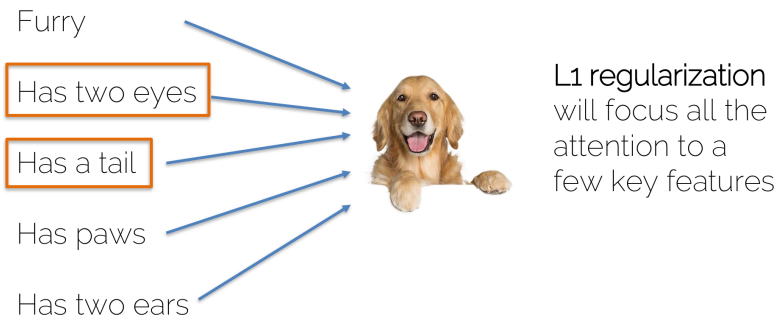


Figure 70: **Effect of L1 regularizer in high level feature space.** [9] Since the L1 regularizer prefers sparse solutions, it will prefer a model which pays attention to only a few key features, i.e. the weights for the other features are set to be close to 0 and thus ignored.

5.5.5 Interpretation of parameter penalties as MAP inference

L_2 regularization can be interpreted as **Bayesian maximum-a-posteriori (MAP) estimation** of the network parameters $\mathbf{w}$ with a Gaussian prior applied to $\mathbf{w}$:

$$
\begin{aligned}
\tilde{\mathbf{w}} &= \underset{\mathbf{w}}{\text{argmax}}\ p(\mathbf{w}|\mathbf{y},\mathbf{X}) \text{...Maximize apriori i.e. distribution of parameters given data} \\
&= \underset{\mathbf{w}}{\text{argmax}}\ p(\mathbf{y}|\mathbf{X},\mathbf{w})\, p(\mathbf{w}) \text{...After applying Bayes rule} \\
&= \underset{\mathbf{w}}{\text{argmax}}\ \log p(\mathbf{y}|\mathbf{X},\mathbf{w}) + \log p(\mathbf{w}) \text{...argmax does not change as log is a monotonous function} \\
&= \underset{\mathbf{w}}{\text{argmax}}\ \log p(\mathbf{y}|\mathbf{X},\mathbf{w}) + \log \mathcal{N}(\mathbf{w}|\mathbf{0},\alpha^{-1}\mathbf{I}) \text{...Choose such a gaussian distribution with 0 mean as prior} \\
&= \underset{\mathbf{w}}{\text{argmin}}\ -\log p(\mathbf{y}|\mathbf{X},\mathbf{w}) + \frac{\alpha}{2}\mathbf{w}^\top\mathbf{w} \text{...The choice of prior turns out to be equivalent to L2 penalty}
\end{aligned}
$$

With computation similar to the one done above, the L1 regularizer can also be interpreted as the **MAP inference** of $\mathbf{w}$ with a **Laplace distribution prior**.

5.5.6 How does the Computation Graph of a Parameter Penalty Regularizer look like ?

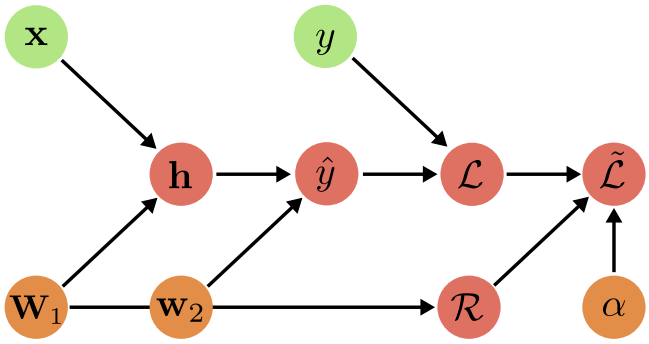


Figure 71: **Computation Graph for a Parameter Penalty**

5.6 Early Stopping

Important Note: An *iteration* means that we have iterated through a single mini-batch. An *epoch* means that we have iterated through all the mini-batches i.e. iterated through our complete training dataset one time. In Fig. 72,

- While training error decreases over time, validation error starts increasing again

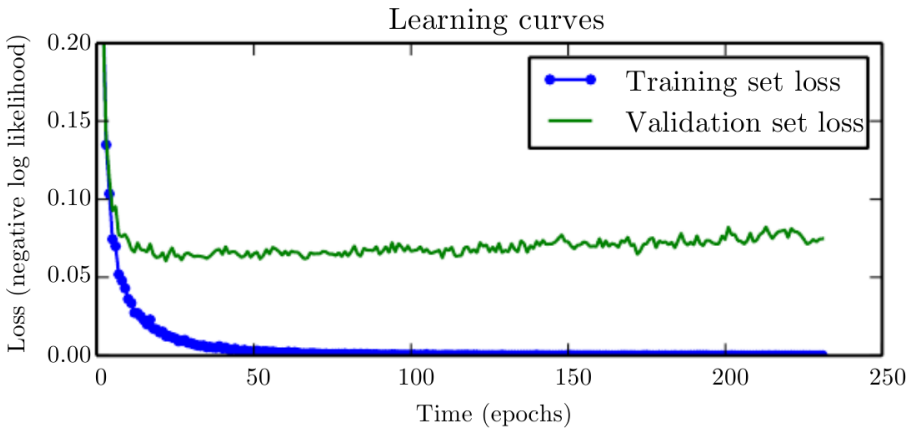


Figure 72: **Typical Loss Curve**

- The graph shows that training for long amounts of time does not always help in reducing validation error, for e.g. in this case, the validation error actually steadily increases with the number of epochs, even though the training error decreases slowly.
- Thus, the most logical approach is to: train for some time and **return parameters with lowest validation error** i.e. store model parameters every fixed number of epochs and at the end rollback to the saved parameters which give the lowest validation error.
- There is a small additional cost in terms of training resources and time, as to do early stopping the validation set needs to be forward propagated every fixed number of epochs so that we can keep track of the validation error.

5.6.1 Early Stopping vs. Parameter Penalties

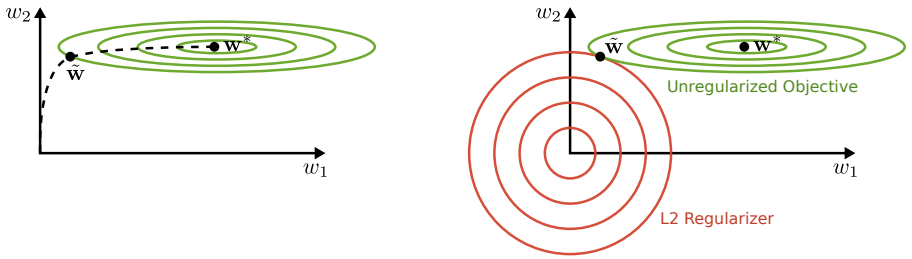


Figure 73: **Early Stopping (Left) vs. Parameter Penalties (Right)**

Early stopping:

- Dashed: Trajectory taken by SGD (Starts from near the origin as we initialize our weights to be close to 0)
- Trajectory stops at $\tilde{\mathbf{w}}$ before reaching the minimum $\mathbf{w}^*$

L2 Regularization:

- Regularize objective with L_2 penalty
- Penalty forces minimum of regularized loss $\tilde{\mathbf{w}}$ closer to origin

- Under some assumptions, both early stopping and parameter penalties can be considered equivalent, as can also be understood intuitively by looking the similar solutions that both approaches obtain in Fig. 73

5.6.2 Summary of Early Stopping

- Most commonly used form of regularization in deep learning
- Effective, simple and computationally efficient form of regularization
- Training time can be viewed as hyperparameter $\Rightarrow$ model selection problem

- Efficient as a single training run tests all hyperparameters (unlike weight decay)
- Only cost: periodically evaluate validation error on validation set
- To reduce this cost the Validation set can be made smaller than the training data (which is usually the case) OR/ AND the evaluation of the validation set can be done less frequently

Remark: If little training data is available, one can perform a second training phase where the model is retrained from scratch on all training data using the same number of training iterations determined by the early stopping procedure. It is important to note that the early stopping procedure was carried out on a training split (split the little training data into training/ validaton set as is done normally). We then use the number of training iterations obtained from this procedure to then train a model on the whole dataset (no splitting into sets unlike earlier).

5.7 Ensemble Methods

Idea:

- Train several models separately for the same task
- At inference time: average results
- Thus, often also called "model averaging"

Intuition:

- Different models make different errors on the test set
- By averaging we obtain a more robust estimate without a better model!
- Works best if models are maximally uncorrelated
- Winning entries of challenges are often ensembles (e.g., Netflix challenge), as empirically speaking it is very likely that using ensemble methods gives a 1-2% performance improvement in most tasks
- Drawback: requires evaluation of multiple models at inference time

5.7.1 Why are Ensembles a good idea ?

Consider K regression models, each of which has an error of $\epsilon_k \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$ with variances $\mathbb{E}[\epsilon_k^2] = v$ and covariances $\mathbb{E}[\epsilon_k \epsilon_l] = c$. The **expected square error of the ensemble predictor** (with each model having the same weight) is given as:

$$\mathbb{E}\left[\left(\frac{1}{K}\sum_k \epsilon_k\right)^2\right] = \frac{1}{K^2}\mathbb{E}\left[\sum_k \left(\epsilon_k^2 + \sum_{l\neq k} \epsilon_k \epsilon_l\right)\right] \text{ ....Expectation of square of sum has been broken into}$$

Expectation over sum of squares and sum of cross terms

$$= \frac{1}{K^2}\left(\sum_k \mathbb{E}\left[\epsilon_k^2\right] + \sum_k \sum_{l\neq k} \mathbb{E}\left[\epsilon_k \epsilon_l\right]\right) \text{ .....Linearity of Expectations}$$

$$= \frac{1}{K^2}\left(Kv + K(K-1)c\right) \text{ ....Plug in defined terms v and c}$$

$$= \frac{1}{K}v + \frac{K-1}{K}c \qquad \text{.....}\textbf{Ensemble Error}$$

- If errors are correlated ($c = v$), the ensemble error becomes $v \Rightarrow$ no gain (Since v is the variance of the error of a single model, there is no performance that has been gained by ensembling)
- If errors are uncorrelated ($c = 0$), the ensemble error reduces to $\frac{1}{K}v$ (We have gained performance by reducing the variance of the errors of each individual model in the ensemble from v to $\frac{1}{K}v$ when the errors are uncorrelated)

Thus:

- **Ensemble maximally effective if errors maximally uncorrelated** i.e. when $c = 0$

5.7.2 Different types of Ensemble Methods

- **Initialization:** Train networks starting from different random initialization on same dataset or using different minibatches (via stochastic gradient descent). This often already introduces some independence.
- **Model:** Use different models, architectures, losses or hyperparameters
- **Bagging:** Train networks on different random draws (with replacement) from the original dataset. Thus, each dataset likely misses some of the examples from the original dataset and contains some duplicates.

Example of Bagging

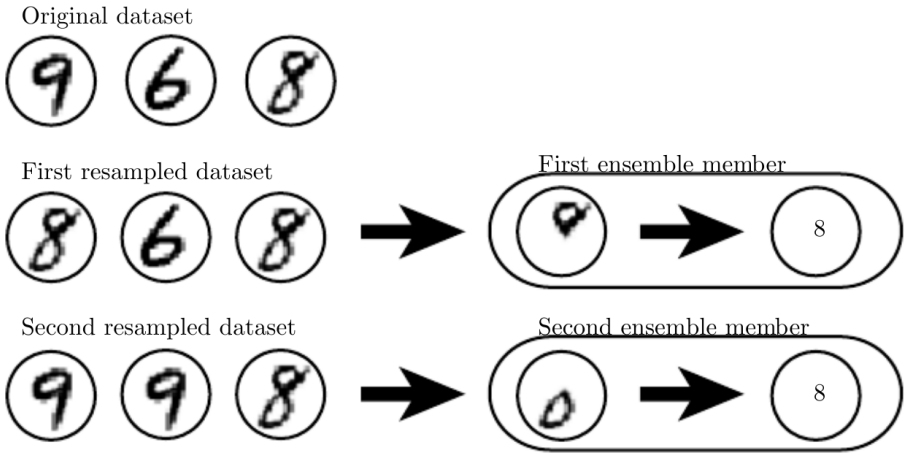


Figure 74: **Using bagging as an ensemble method**

- First model learns to detect top "loop", second model detects bottom "loop"
- The intuition behind using these models as part of an ensemble is that, each model learns to perform a simple task which is independent of each other (detecting top loop for the model trained on first resampled dataset and detecting the bottom loop for the model trained on the second resampled dataset. We can thus, average these models which learn to perform simple tasks into an ensemble which gives better performance by averaging over the predictions of each individual model.

5.8 Dropout

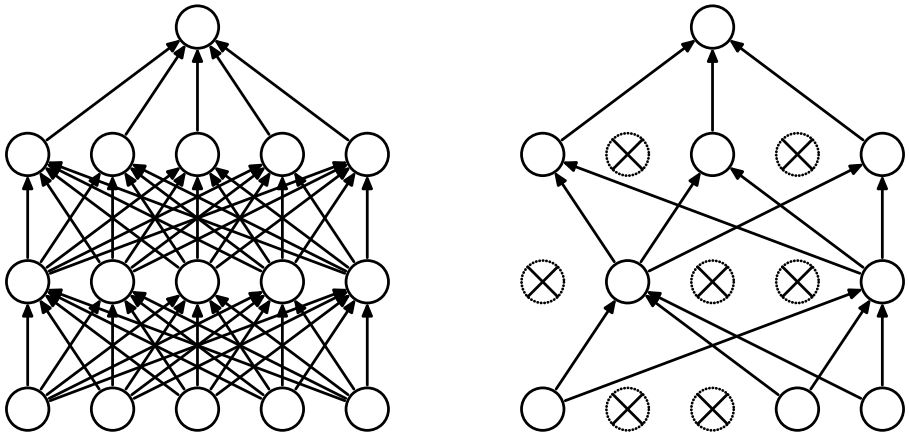


Figure 75: **Dropout** [11]

Idea:

- During training, **set neurons to zero** with probability μ (typically $\mu = 0.5$)
  Important Note: *Removing* a neuron or setting it to zero means that we remove all the incoming and outgoing connections of the neuron from the model architecture.
  Also this procedure of dropout is applied to all neurons i.e. every neuron in the network will be dropped out with probability μ.

- Each binary mask (after performing dropout we obtain a binary mask i.e. is a given neuron in the network active or not) is one model. This is because every different binary mask would give rise to a different model architecture.

- This probability-based dropout mask is applied in every iteration. Since in every iteration we flip a biased/ unbiased coin for every neuron to determine whether it will participate in training, the binary mask of the complete network changes randomly with every training iteration

- Creates **ensemble "on the fly"** from a single network with shared parameters. While performing dropout we get random binary masks in every iteration i.e. random models in every iteration. Thus, dropout is a technique which allows us to take advantage of ensemble methods i.e. to average predictions of different models (generated in each iteration) from just a single network without any of the computational overhead introduced in traditional ensemble methods like creating/ training multiple models from scratch, storing parameters of each model, etc.
  Important Note: The ensemble created by dropout is exponentially large and grows with the number of training iterations. This is because the probability of getting the same binary mask i.e. same model in any future iteration is extremely low. Thus, dropout effectively adds a new model to the ensemble in every iteration.

5.8.1 Why is Dropout a good idea ?

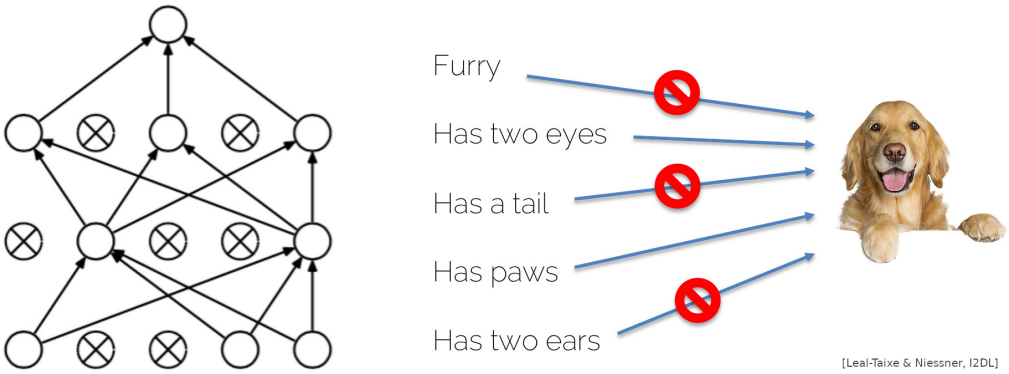


Figure 76: **Forward Pass with Dropout** [9]

- Forces the network to learn a **redundant representation** $\Rightarrow$ regularization.
  This is because the model understands that any neuron can be dropped randomly in a training iteration, therefore it learns to not rely heavily on the output of certain neurons/features for making its predictions given how these neurons could easily be dropped out in future iterations. This is exactly why dropout forces the model to represent information in a more distributed or redundant way, so that this randomness of neurons dropping out does not affect its final prediction drastically and thus also acts as a form of regularization.

- Reduces effective **model capacity** $\Rightarrow$ requires larger models, longer training
  Since dropout forces the model to learn redundant representations, it reduces the variance of the model i.e. reduces the capacity of the model. Thus, one downside of dropout is that because it reduces the capacity of a model we need to use bigger networks and train longer to achieve good training performance. However it is worth remembering, that we make this sacrifice to achieve very good generalization performance.

- Prevents **co-adaptation** of features (units can't learn to undo output of others)
  In classical deep learning, neurons can learn to undo the effect of other nearby neurons. For e.g. consider a scenario where one neuron learns to have a very high activation and another nearby neuron learns to have a very low activation. When the outputs of both these neurons are combined they effectively cancel out

the effect of each other. This phenomenon is called co-adaptation and is undesirable as in such a scenario the network does not learn any useful representation, given how it is busy cancelling out its own neurons. Whereas when using dropout since any neuron can be dropped out randomly, the network cannot learn to change the output of nearby neurons by relying on the output of a certain neuron. Thus dropout prevents co-adaptation of features from occurring.

- Requires only **one forward pass at inference** time (Explanation in Section 5.8.2)

5.8.2 Dropout at Inference Time

- Dropout makes the output random. Formally, we have:

$$\hat{y}_{\mathbf{z}} = f_{\mathbf{w}}(\mathbf{x}, \mathbf{z})$$

Important Note: The output of the neural network $\hat{y}$ now not only depends on the weight vector $\mathbf{w}$ but also on the binary mask $\mathbf{z}$ generated by dropout. Since the binary mask is random, the model architecture is random and thus the output with dropout is also random.
Here, $\mathbf{z}$ is a binary mask with one element per unit drawn i.i.d. from a Bernoulli $p(z_i) = \mu^{1-z_i}(1-\mu)^{z_i}$ where $z_i = 0$ if neuron i is removed from the network

- At inference time, we want to calculate the **ensemble prediction:**

$$\hat{y} = f_{\mathbf{w}}(\mathbf{x}) = \mathbb{E}_{\mathbf{z}}[f_{\mathbf{w}}(\mathbf{x}, \mathbf{z})] = \sum_{\mathbf{z}} p(\mathbf{z}) f_{\mathbf{w}}(\mathbf{x}, \mathbf{z})$$

Important Note: Every binary mask generated by dropout corresponds to a new model. To take benefit of the idea proposed in ensemble methods(Section 5.7), we then take the expectation of our individual model predictions over the generated ensemble i.e. the set of all models (binary masks) that have been generated by dropout.

- The number of ensembles generated by dropout is exponential in the number of neurons. That is, if we have M neurons in our original model, we have 2^M possible ensembles i.e. 2^M possible binary masks.
Important Note: We have to sum over all $\mathbf{z}$ (i.e. over all generated binary masks) to calculate the Ensemble Prediction using dropout.
Thus to obtain a single ensemble prediction we have to sum over 2^M possible terms (since we have 2^M possible binary masks), leaving the calculation intractable. (As the number of terms in the summation will rise exponentially with the number of neurons)

How do we solve the intractability of calculating an Ensemble Prediction while using dropout ?

Let us consider a simple **linear model:**

$$f_{\mathbf{w}}(\mathbf{x}) = w_1 x_1 + w_2 x_2$$
$$f_{\mathbf{w}}(\mathbf{x}, \mathbf{z}) = z_1 w_1 x_1 + z_2 w_2 x_2$$

Note: $f_{\mathbf{w}}(\mathbf{x}, \mathbf{z})$ is the dropout equivalent of making a prediction, where z_1 and z_2 are the binary masks of a neuron that can either be 0 (neuron has been dropped out) or 1 (neuron will remain active). Assuming $\mu = 0.5$, during training we optimize the **expectation over the ensemble:**

$$\begin{aligned}\mathbb{E}_{\mathbf{z}}[f_{\mathbf{w}}(\mathbf{x}, \mathbf{z})] &= \frac{1}{4}(0+0) + \frac{1}{4}(w_1 x_1 + 0) + \frac{1}{4}(0 + w_2 x_2) + \frac{1}{4}(w_1 x_1 + w_2 x_2) \\ &= \frac{1}{2}(w_1 x_1 + w_2 x_2) = \frac{1}{2} f_{\mathbf{w}}(\mathbf{x})\end{aligned}$$

Note: We have 4 terms in the summation as we have 4 possible combinations of z_1 and z_2. Each of the terms are weighted equally $(\frac{1}{4})$, since the probability of getting a given combination out of the 4 possible is $p(\mathbf{z}_1)p(\mathbf{z}_2) = 0.5 * 0.5 = 0.25$.
Important Note: In the above equation, we see that the Expected Prediction of the trained model is half (μ) times the output of the original network. This implies that the weights learnt by the model trained using dropout must be double $(1/\mu)$ times the weights of the original model, as only then would both the models give

the same prediction.
It is also important to note that dropout is only used during training. Therefore while performing inference we want our model trained using dropout to behave the same way as the original model.
Thus, at test time, we must **multiply the trained weights** (which are $1/\mu$ times the weights of the original model) by the dropout probability μ.
Remark: This weight scaling inference is only an approximation for non-linear models. However, this approximation still works really well empirically. (As can bee seen in Fig. 78)

5.8.3 Visualizing Effect of Dropout

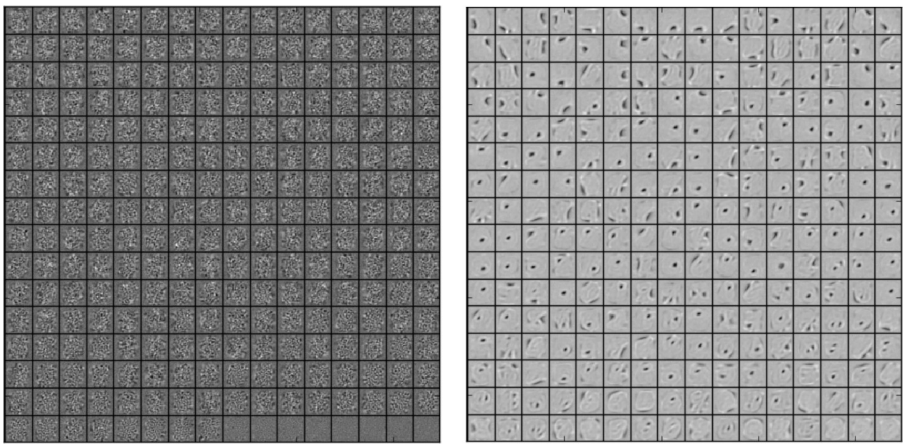


Figure 77: **Comparing the outputs of an autoencoder trained without (left) and with dropout**

- Features of an Autoencoder on MNIST with a single hidden layer of 256 ReLUs
- Left: Without dropout ⇒ more co-adaptation, as many units respond to cancel out the effect leading to noisy representation of the image
- Right: With dropout ⇒ less co-adaptation, as every unit learns to recognize information that appear in different places in the input image leading to more meaningful features and thus better generalization

Method	**Test Classification error %**
L2	1.62
L2 + L1 applied towards the end of training	1.60
L2 + KL-sparsity	1.55
Max-norm	1.35
Dropout + L2	1.25
Dropout + Max-norm	**1.05**

Table 9: Comparison of different regularization methods on MNIST.

Figure 78: **Dropout helps reduce generalization error**

5.9 Data Augmentation

Motivation:

- Deep neural networks must be **invariant** to a wide variety of input variations. We want our model to be invariant (not affected by) to input variations, as we want it to classify all inputs irrespective of their variations correctly. For e.g. we would want our model to correctly classify both an otter lying on its back and an otter sitting on a rock as an otter, despite the variations in these 2 images.

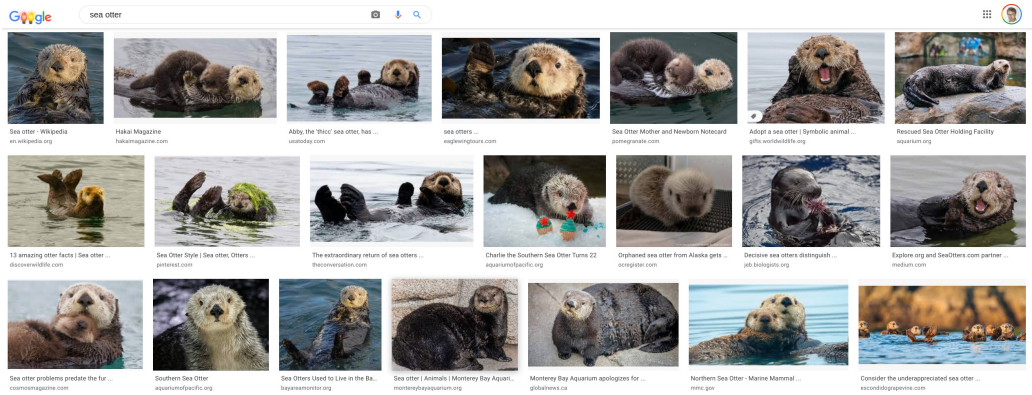
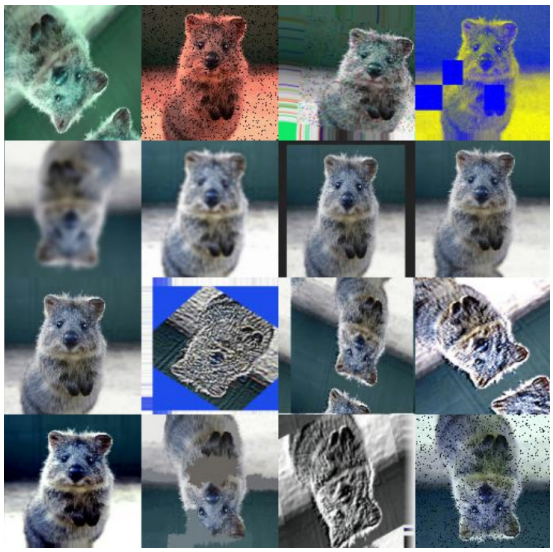


Figure 79: **Variations of images from a single class**

- Often in real-life data there exist **large intra-class variations** in terms of pose, appearance, lighting, etc. This makes image classification in particular an extremely hard task.

How do we tackle intra-class variations and improve generalization ?

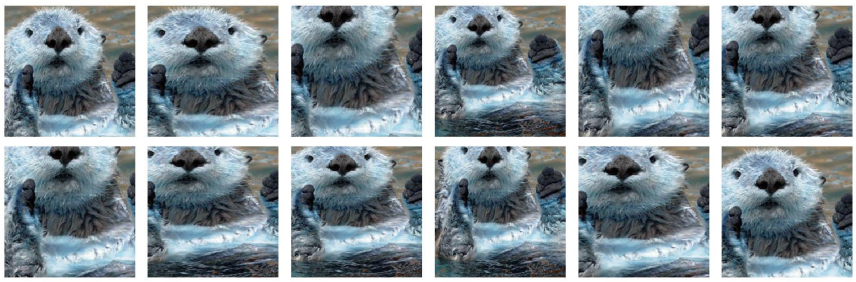
- Best way towards better generalization is to **train on more data**. However, data in practice often limited.
- Goal of data augmentation: create **"fake" data** from the existing data (on the fly) and add it to the training set. This process is "on the fly" as we do not store the augmented images. The augmented images are generated randomly (by applying random transformations) from each image in our batch at the beginning of every training iteration.
- New data must **preserve semantics** i.e. the augmented data of a class should not change the semantic meaning of the class category. For e.g. augmenting the images of a class of dogs to look like cats (with label as dogs) is not a good idea.
- Even **simple operations** like translation or adding per-pixel noise often already greatly improve generalization
- https://github.com/aleju/imgaug is a popular library on GitHub where implementations of different image augmentation techniques can be found.



5.9.1 Geometric Transformations

Image Cropping:

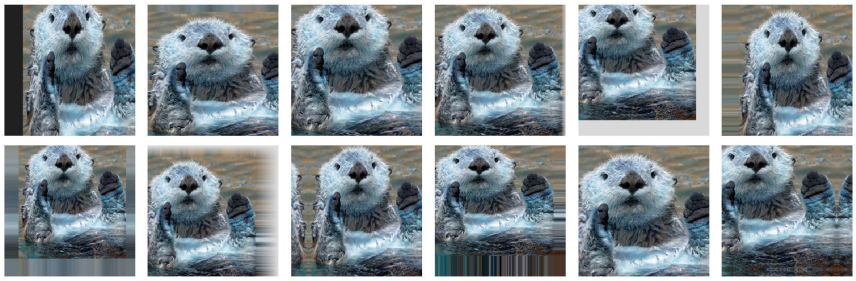
- Randomly crop and re-scale images to original image size
- Do not crop regions from the image which are too small, as then semantics of the image category could be lost



iaa.Crop(px=(1,64))

Image Cropping and Padding:

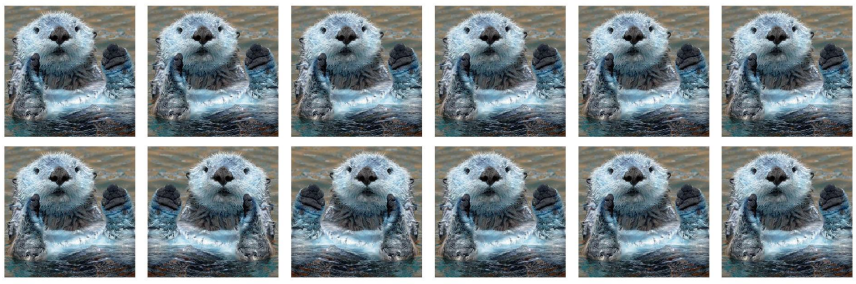
- Similar to Image Cropping the only difference being that the cropped images are not resized, but are instead padded with either a constant color or expanded image border colors etc. to reach the original image size.



iaa.CropAndPad(percent=(-0.2, 0.2),pad_mode=iaa.ia.ALL,pad_cval=(0, 255))

Horizontal Image Flipping:

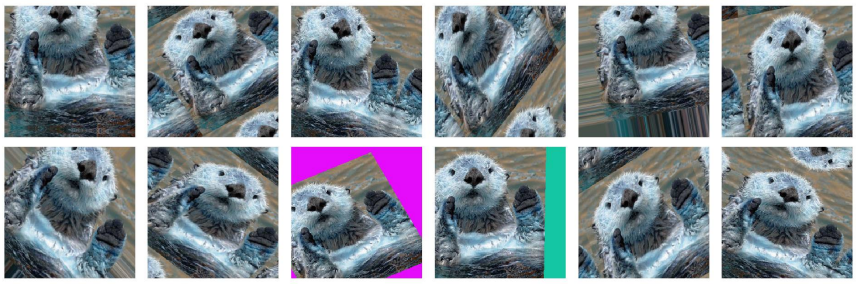
- Flip the given images horizontally based on the specified probability
- Not always good to use this augmentation naively, as it possible that semantic meaning of an image changes after flipping. For e.g. in street scene semantic segmentation even though horizontal flipping might be a good idea to generate more realistic data, the notion of traffic lanes would change from right sided traffic to left sided traffic and vice-versa leading to undesirable results
- Depending on the task images can also be flipped upside-down



iaa.Fliplr(0.5)

Affine Transformation:

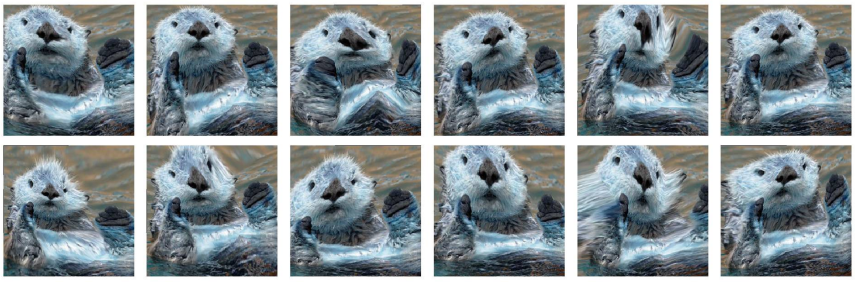
- Apply a single affine/ linear transformation to the 2D image space i.e. scale, rotate, shear an image etc.
- Blank spaces left after the transformation can be filled by using different strategies like using constant color, expand image colors, replicate colors, etc.



iaa.Affine()

Piecewise Affine Transformation:

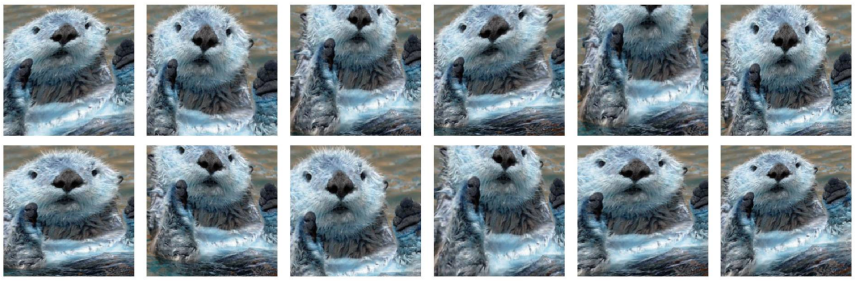
- Similar to Affine transformation albeit now a single transformation is not applied to the whole image
- The image is perceived as an underlying grid. A piece-wise affine transformation is applied independently to each grid cell. This causes different regions in the image to be distorted differently



`iaa.PiecewiseAffine(scale=(0.01, 0.1))`

Perspective Transformation:

- Similar to Affine transformation albeit a perspective transform has more degrees of freedom
- The perspective effect causes certain regions of the image to be squeezed and other regions to be enlarged

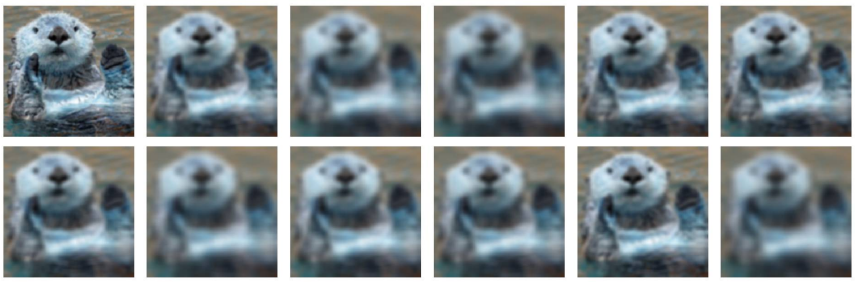


`iaa.PerspectiveTransform(scale=(0, 0.4))`

5.9.2 Local Filters

Gaussian Blur:

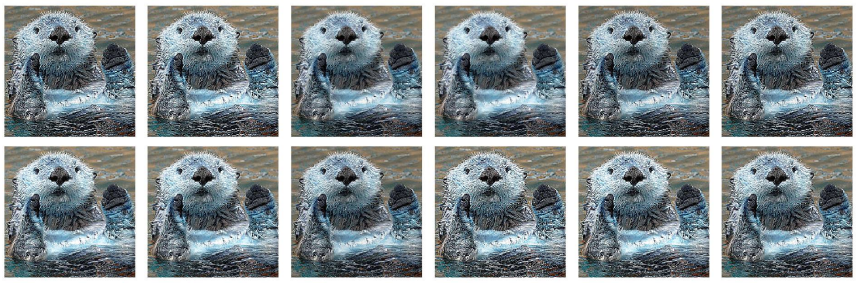
- Applies the gaussian blur filter onto the given image with the σ of the gaussian distribution being chosen randomly from an interval specified by the user
- Helps the model to recognize objects at different resolutions
- Also the process of capturing data using a camera induces a small amount of blur. Thus applying gaussian blur also helps the model to be robust to such camera-induced variations in captured images



`iaa.GaussianBlur(sigma=(0.0, 10.0))`

Image Sharpening:

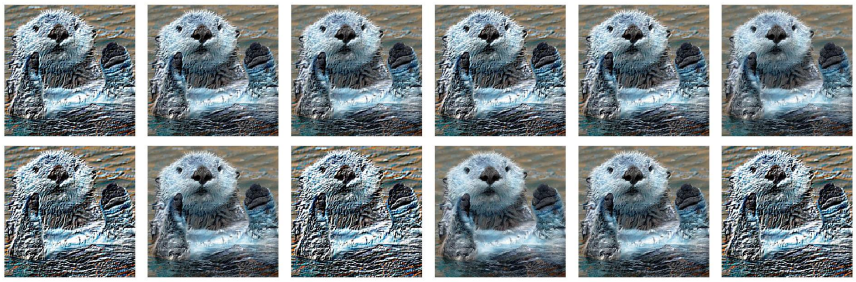
- Does the opposite of introducing blur i.e. introduces sharpness into input images



`iaa.Sharpen(alpha=(0, 0.5), lightness=(0.75, 1.25))`

Emboss Effect:

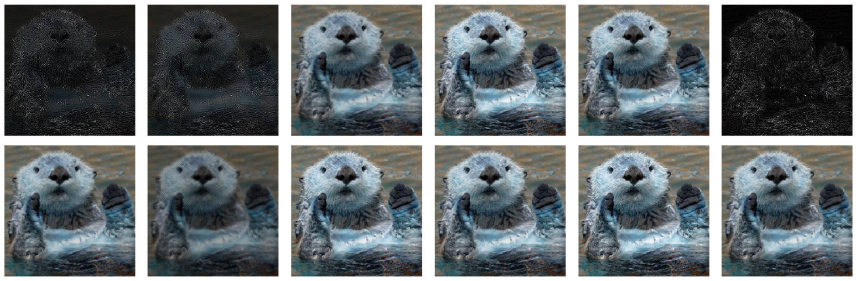
- Another effect which introduces pixel-wise light/shadow and could improve generalization performance



`iaa.Emboss(alpha=(0, 1.0), strength=(0, 2.0))`

Edge Detection:

- Uses the edge detected version of a given image
- Is important to verify whether the edge detected version are still reasonable and do not change / make it impossible to decipher semantics of the class



`iaa.EdgeDetect(alpha=(0, 1.0))`

5.9.3 Adding Noise

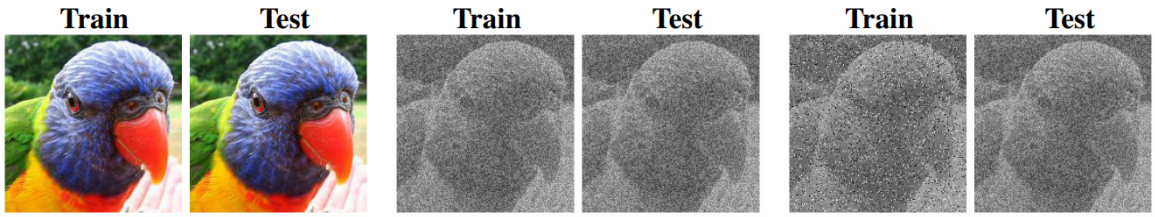
A popular data augmentation technique which typically involves introducing a more *structured* random per-pixel noise.

Why is studying noise important ?

- Deep Networks are highly sensitive to noise in images. Thus, it is very likely that adding even a small amount of noise (just to the test data) would lead to drastic reduction in performance, even though to the human eye the "noisy" test dataset would look almost similar to the original test dataset

  Important Note: In Fig. 80 we can see that when a deep network was trained on a dataset which added uniform noise to both the train and test dataset, it managed to achieve super-human performance. However, when the network was trained on the same dataset albeit this time with salt-and-pepper noise (a noise where pixels of the image are set to white or black based on a specified probability), the network managed to obtain only chance-level performance i.e. as good as random-guessing. This result is especially surprising given how both datasets (one with uniform noise and the other with salt-and-pepper noise look like they have similar noise distributions) and yet the difference in performance of the network between both these visually-similar datasets is drastic. This validates the point that networks are highly susceptible to noise.

- Noise is present everywhere while collecting data, for instance noise induced by camera sensors while capturing data, noise induced by the scene illumination conditions, etc.



(a) Super-human performance (b) Super-human performance (c) Chance level performance

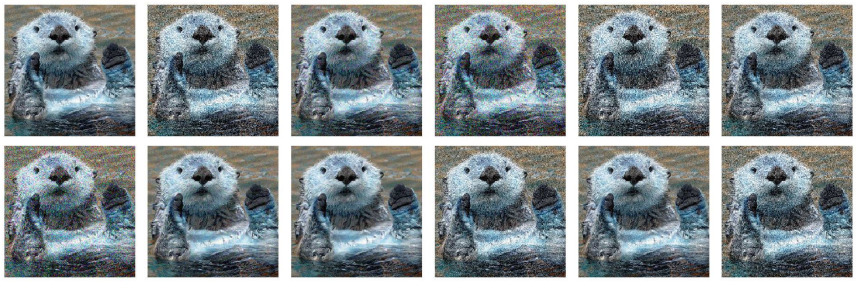
Figure 1: Classification performance of ResNet-50 trained from scratch on (potentially distorted) ImageNet images. **(a)** Classification performance when trained on standard colour images and tested on colour images is close to perfect (better than human observers). **(b)** Likewise, when trained and tested on images with additive uniform noise, performance is super-human. **(c)** Striking generalisation failure: When trained on images with salt-and-pepper noise and tested on images with uniform noise, performance is at chance level—even though both noise types do not seem much different to human observers.

Figure 80: **How is classification performance affected by different kinds of noise ?** [5]

Following are some of the most popular additive-noise augmentation techniques to improve generalization performance:

Gaussian Noise:

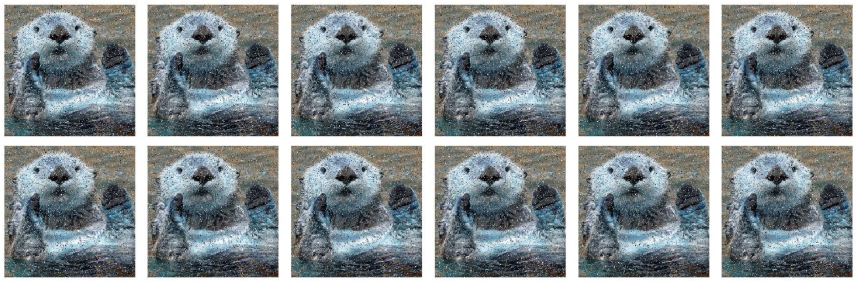
- Add a fixed Gaussian noise to each pixel in an image. The additive noise is chosen randomly for different images



`iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.2*255)`

Salt and Pepper Noise:

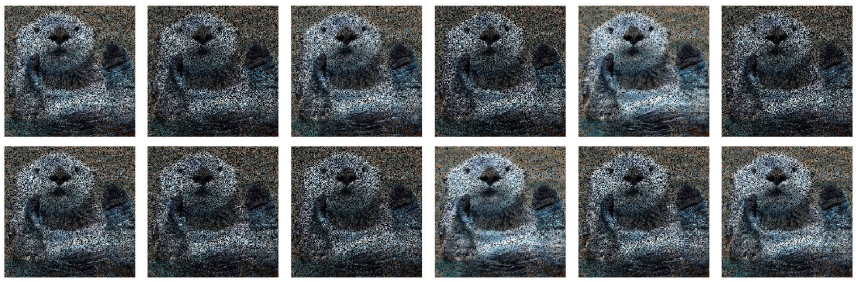
- Each pixel in an image is turned into either black or white depending on a specified probability



`iaa.SaltAndPepper(0.1)`

Dropout Noise:

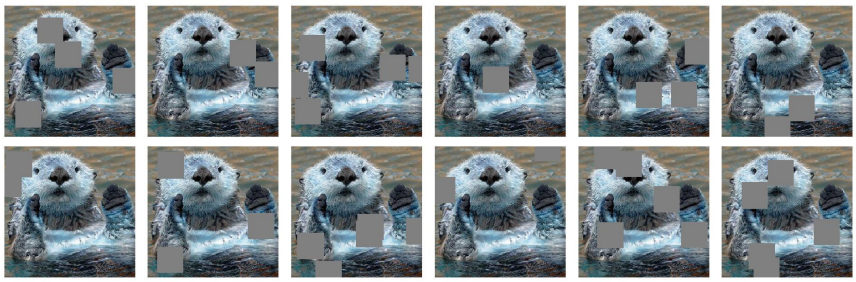
- Unlike salt-and-pepper noise which does not retain color, in dropout noise each pixel is only turned to black based on a specified dropout probability



`iaa.Dropout((0.01, 0.5))`

Cutout Noise:

- A more structured form of noise which involves cutting-out i.e. setting pixels to a specific color of regions in an image.



`iaa.Cutout(nb_iterations=(1, 5), size=0.2, squared=False)`

Can noise be added only to inputs ?

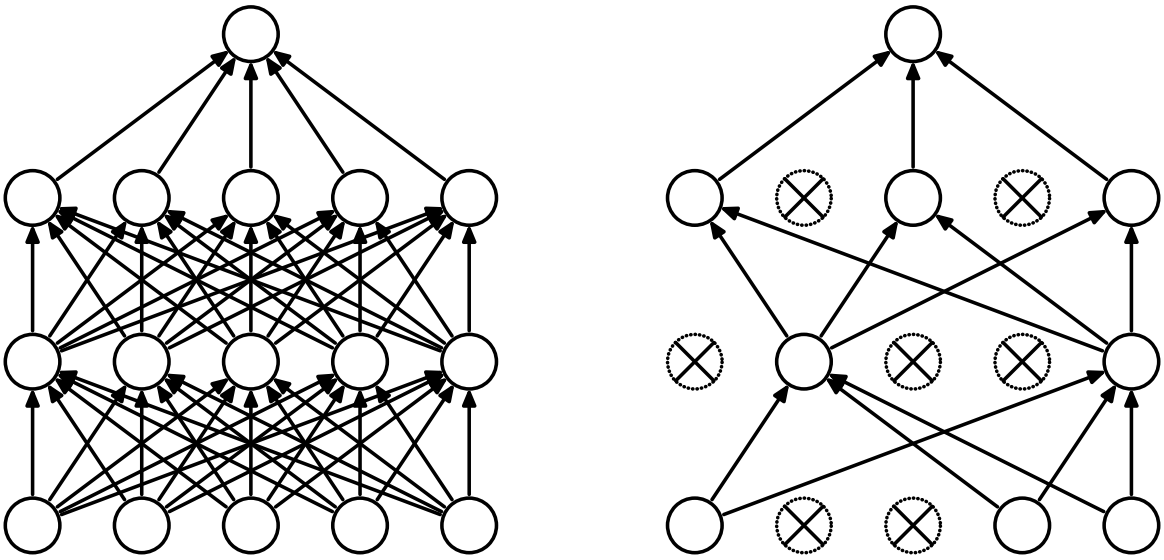


Figure 81: **Adding noise to hidden layers**

- Noise can also be applied to the **hidden units**, not only to the input
- Prominent example of applying noise to the hidden inputs is : **Dropout**. Another example would be adding random noise individually to each hidden unit
- The advantage of adding noise to the hidden units and not just the input is that we are adding robustness not just to input-level features but also at intermediate levels where higher level concepts and higher level representations of the neural network are established
- As seen in Section 5.8, adding noise to the hidden units of a network also works really well empirically to gain better generalization performance

5.9.4 Color Transformations

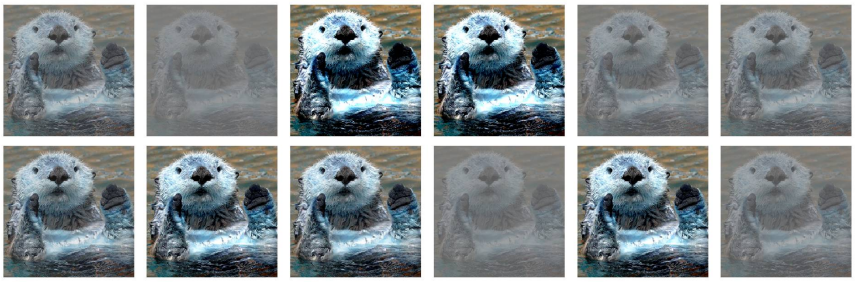
Why are Color Transformations important ?

- Color transformations have played a key-role in the success of most neural networks that work well on the Imagenet dataset
- Cameras produce different color spectrum's based on the type of sensors used and the kind of white-balancing performed. A more important reason behind this is the change in lighting conditions while capturing data, for e.g. the colors captured by a camera would change dramatically for the same scene depending on the time of the day i.e. sunset would have warmer colors, mid-day would have brighter colors, etc. We thus want our models to be invariant to such variations so that our generalization performance increases. This is exactly why color transformations are an essential form of data augmentation

Following are some of the most popular color transformations techniques to improve generalization performance:

Contrast:

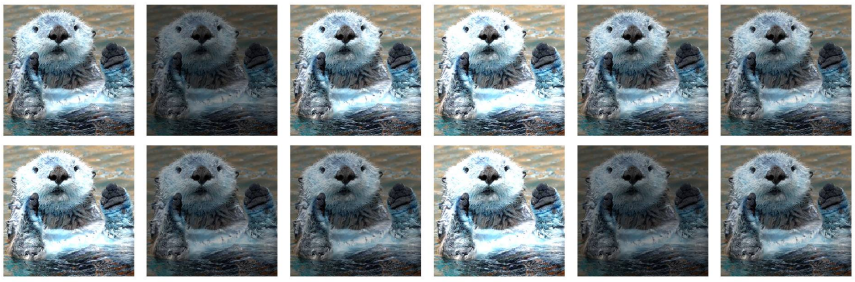
- Change the contrast of images to either become faint or have stronger contrast.



`iaa.LinearContrast((0.1, 2.0), per_channel=0)`

Brightness:

- Change the brightness of the whole images to make them either darker or brighter



`iaa.Multiply((0.5, 1.5), per_channel=0)`

Brightness per Channel:

- Instead of changing brightness of the whole image, change brightness for each channel instead

`iaa.Multiply((0.5, 1.5), per_channel=0.5)`

Local Brightness:

- Brightness are applied locally per channel, the locations of which are determined by a specified frequency noise

`iaa.FrequencyNoiseAlpha(exponent=(-4, 0), first=iaa.Multiply((0.5, 1.5), per_channel=True))`

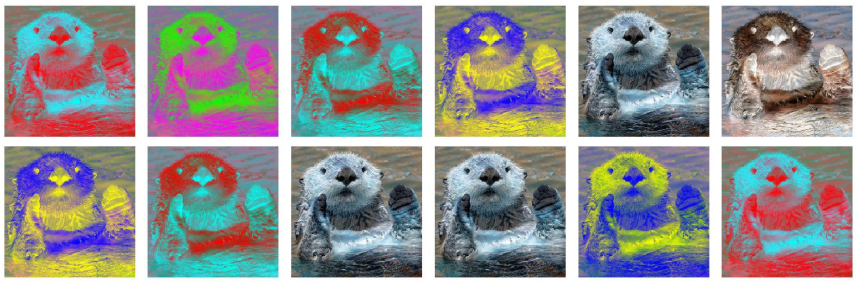
Hue and Saturation:

- Change hue and saturation of images

`iaa.AddToHueAndSaturation((-50, 50))`

Color Inversion:

- Even though color inverted images are not a realistic choice as they look nothing like natural images, depending on the task they could still be a good strategy to improve generalization performance



`iaa.Invert(0.5, per_channel=0.75)`

Grayscale:

- Convert the images into a spectrum between grayscale and colored images, the strength of which is detrmined by the specified α value

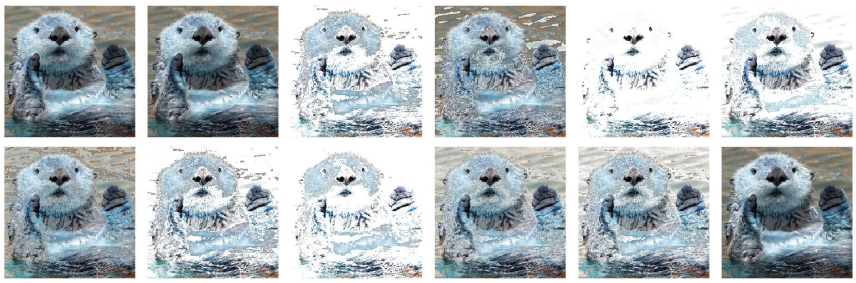
`iaa.Grayscale(alpha=(0.0, 1.0))`

5.9.5 Weathers

Even though weather effects might sound complex they are still very easy to compute and can be done on-the-fly without requiring any sophisticated graphics engine.

Snow:

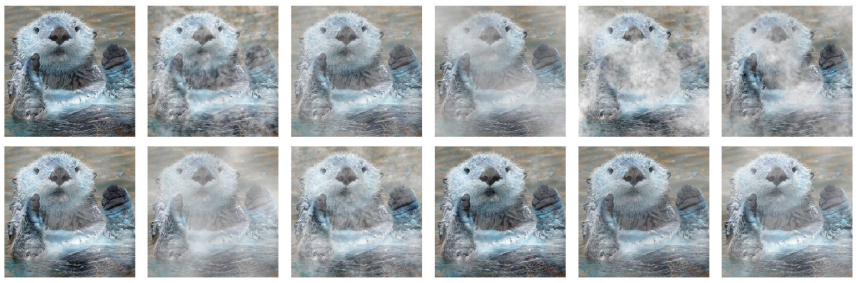
- Introduces a snow-like effect on the images



`iaa.FastSnowyLandscape(lightness_threshold=(100, 255),lightness_multiplier=(1.0, 4.0))`

Clouds:

- Introduces an effect on the images which look like clouds



`iaa.Clouds()`

Fogs:

- Introduces a fog effect on the images

`iaa.Fog()`

5.9.6 Random Combinations

In practice, all of the aforementioned augmentation techniques are combined and then applied randomly on-the-fly to different images drawn randomly from within a mini-batch.

5.9.7 Output Transformations

In some cases it is possible that after applying a transformation on the input, we have to apply transformations on the corresponding output/ class as well.

In Certain Classification Tasks:

- For some classification tasks, e.g., **handwritten letter recognition**, be careful to not apply transformations that would change the output class
- Example 1: Horizontal flips changes the interpretation of the letter 'd':

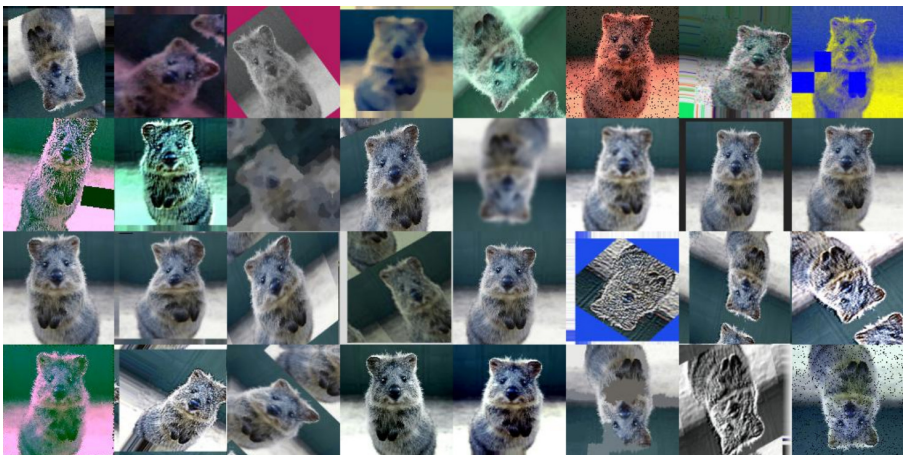


Figure 82: **Random Data Augmentation Combinations applied to a single input image**

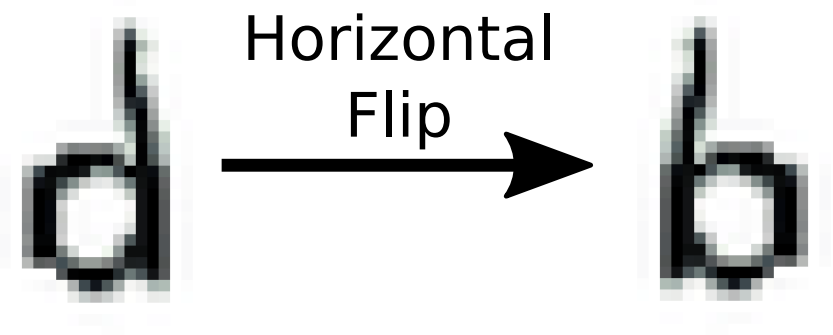


Figure 83: **Random Data Augmentation Combinations applied to a single input image**

- Example 2: 180° rotations changes the interpretation of the number '6':

- Remark: For **general object recognition**, flips and rotations can often be useful!

In other tasks with structured and more complex output

- For dense prediction tasks (depth/instance/keypoints), also **transform targets**. For e.g. in tasks like instance segmentation, depth prediction or stereo depth estimation, any affine/ warping/ translation transformations done to the input would affect the ground-truth predictions (whereas per-pixel noise would make no difference). Thus, in such dense prediction tasks, the output need to be transformed accordingly so that we obtain the valid ground-truth for the transformed input image

5.9.8 Important Remarks about Data Augmentation

- When comparing two networks, make sure you **use the same augmentation**. This is because data augmentation is an extremely powerful strategy and could significantly improve performancce. If the same augmentation strategy is not used in both networks, one could be misled into thinking that their idea was responsible was better performance, whereas in reality it was just a better data augmentation strategy
- Consider data augmentation as a **part of your network design**
- It is important to specify the right distributions (often done empirically). Not all transformations would be applicable or beneficial to improving performance in a given task. Thus the kind of transformations that would help in improving performance and their corresponding strength (hyperparameters) need to be chosen empirically
- Can also be **combined with ensemble idea:**

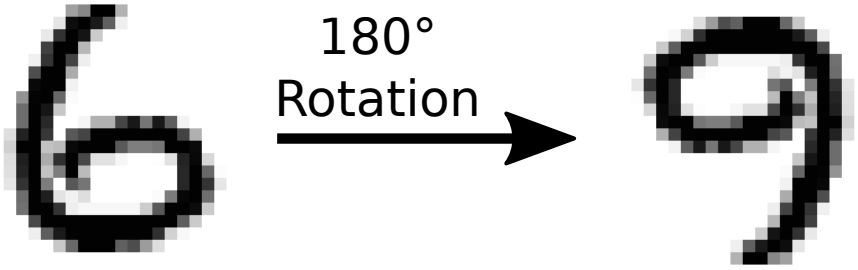


Figure 84: **Random Data Augmentation Combinations applied to a single input image**

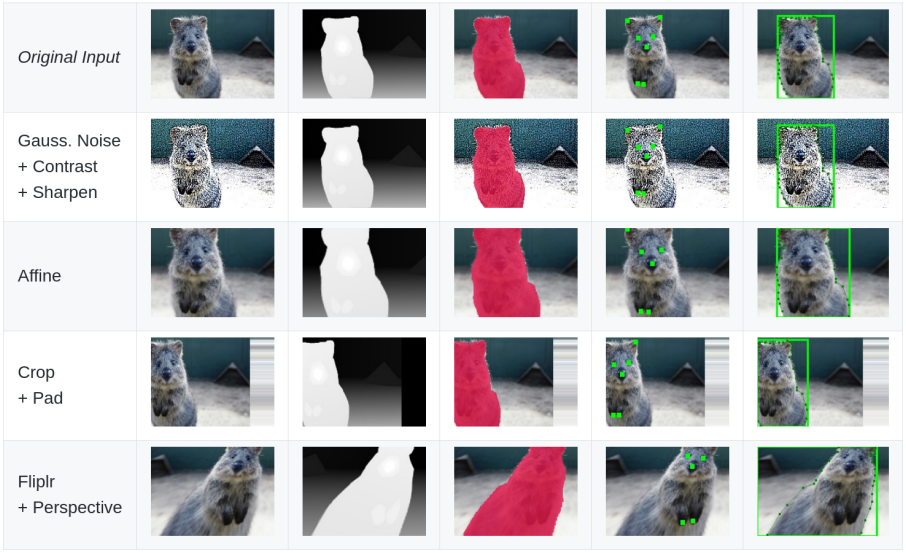


Figure 85: **Random Data Augmentation Combinations applied to a single input image**

  - At training time, sample random crops/scales and train one model
  - At inference time, average predictions for a fixed set of crops of the test image

- AutoAugment [4] uses reinforcement learning to **find strategies automatically:**

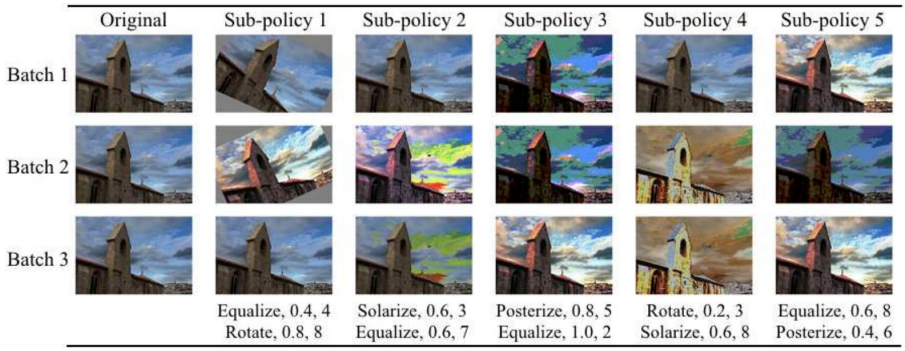


Figure 86: **Reinforcement Learning to find best Data Augmentation Strategies**

6 Optimization

We have previously already discussed the most basic form of optimization stochastic gradient descent and even simpler gradient descent. In the following we will learn about some of the variants of these. As well as some strategies on how to build up a deep learning project and on how to debug our model if there is some bug or expected behaviour.

6.1 Optimization Challenges

First, let's revisit gradient descent and in the following look at the most frequent challenges that we encounter while optimizing a deep learning model.

6.1.1 Gradient Descent

The standard formulation for gradient descent, that we've already seen before. The following algorithm describes the progress of optimization over time:
Let $\mathbf{w}$ be the parameters that we want to optimize, $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w})$ the gradients of an arbitrary loss function, η the learning rate and t the timestep/iteration.

$$\mathbf{w}^0 = \mathbf{w}^{\text{init}}$$
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^t)$$

First we initialize $\mathbf{w}^0$ at timestep 0 to some initial value, then in every timestep we update $\mathbf{w}^{t+1}$ by adding a fraction of the negative gradients to the weights $\mathbf{w}^t$ of the current timestep.

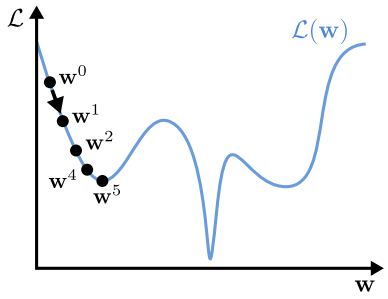


Figure 87: **Gradient Descent** A 1D example of a few gradient descent update steps.

In Fig. 87 we can see a 1D example loss function $\mathcal{L}(\mathbf{w})$ wrt. to the network parameters $\mathbf{w}$. The loss is a non-convex function, which is the case for most neural network losses. Therefore there are multiple local minima and one global minima, that we want to reach in the optimal case. Although we can only find one of those through optimization. The good news is that many local minima in deep neural networks are good ones. Here we can nicely see why we need to add the negative gradient to progress further. The gradient at $\mathbf{w}^0$ is negative, which means that if add it directly to the weight $\mathbf{w}$, the weight (x-axis) would get smaller, resulting in a new $\mathbf{w}^1$ that is to the left of $\mathbf{w}^0$, thus worse. By adding the negative gradient, in this case a positive number, to $\mathbf{w}^0$ we end up at the $\mathbf{w}^1$ as shown in the figure, thus closer to the local minima.

6.1.2 Learning rate

In the case of a convex loss function as seen in Fig. 88, we can see one of the challenges that we face in deep learning - choosing the right learning rate η. A learning rate too low leads to a smaller step size, which then

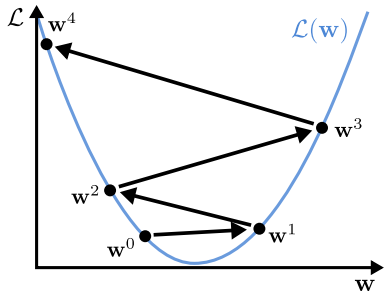


Figure 88: **Divergence** A 1D example of a few gradient descent update steps that diverge.

results in too many steps that are needed to converge in the minimum. Whereas a learning rate too high might lead the network to divergence. Fig. 88 shows this divergence behaviour, where the step size is too large such that we jump over the minimum, to a possibly steeper part of the loss function (higher gradient). This might repeat itself, leading to divergence.

6.1.3 Cliff

Another challenge regarding the gradients is a cliff (Fig. 89), where the updated weight $\mathbf{w}^t$ jumps over the minimum onto a steep cliff (very high gradient), which then catapults the next updated parameters $\mathbf{w}^{t+1}$ further away from the minimum. A common heuristic to counteract such effects is to clip the gradient to a a priori selected range, which introduces another hyperparameter we need to search.

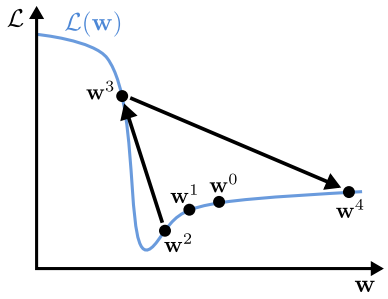


Figure 89: **Cliff** A 1D example of a few gradient descent update steps that jumps off a cliff.

6.1.4 Saddle point

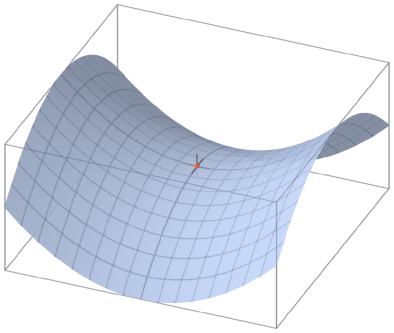


Figure 90: **Saddle point** An example for a saddle point of a simple loss function.

The saddle point (see Fig. 90) of loss functions is a tricky part, because the gradient $\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = 0$, although we are not at a minimum - not even a local minimum. In reality, this is mostly not a problem, even though there are many saddle points in DL, because the chance to exactly hit a saddle point is very low. For every parameter direction the gradient needs to be exactly zero, which is unlikely when working with millions of parameters.

6.1.5 Plateau

A region similar to the saddle point is the plateau (Fig. 91), where the gradients are close to zero (e.g. saturated sigmoid activation function, dead ReLUs).

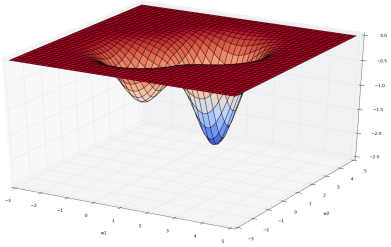


Figure 91: **Plateau** An example for a plateau of a loss function.

6.1.6 Ravine

A ravine (Fig. 92) is a very narrow valley with a small gradient along the slope of the valley. The ravine in the loss function leads to the minimum. Due to the narrow valley it is very hard easy to diverge, similar to Section 6.1.2 and Section 6.1.3. Furthermore due to the small gradient it is hard to follow the narrow path to the minimum.

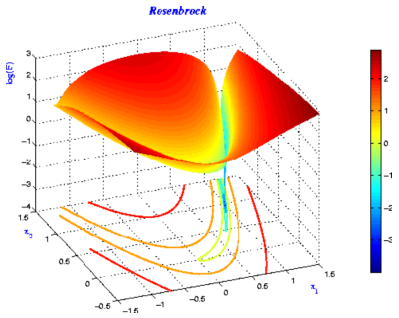


Figure 92: **Ravine** An example for a ravine of a loss function.

6.2 Optimization Algorithms

In this section we will take a look at some optimization algorithms that are more advanced than basic (stochastic) gradient descent and try to overcome some of the issue that we discussed in the previous Section 6.1.

6.2.1 Gradient Descent Algorithm

Let us first look at the basic gradient descent algorithm:

1. Initialize weights $\mathbf{w}^0$ and pick learning rate η
2. For all data points $i \in \{1, \dots, N\}$ do:
   (a) Forward propagate $\mathbf{x}_i$ through network to calculate prediction $\hat{\mathbf{y}}_i$
   (b) Backpropagate to obtain gradient $\nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i, \mathbf{w}^t)$
3. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t)$
4. If validation error decreases, go to step 2, otherwise stop

Be aware that we update the parameters by taking the average gradient over all training samples. This creates a problem when we use a lot of parameters and training points (e.g. 1 million or more), because the computation of one forward pass for every point is extremely expensive. Furthermore to compute it we need to hold all of our data in memory, which does often not fit.

6.2.2 Stochastic Gradient Descent

Luckily there is a solution for that problem - `Stochastic Gradient Descent`. This algorithm solves the problem of slow computation and large memory usage and thus is the basis for the subsequent algorithms.
Stochastic gradient descent uses the fact that the total loss over the entire training set can be expressed as an expectation:

$$\frac{1}{N} \sum_i \mathcal{L}_i(\mathbf{w}^t) = \mathbb{E}_{i \sim \mathcal{U}\{1,N\}} \left[\mathcal{L}_i(\mathbf{w}^t) \right]$$

where a sample is drawn uniformly from the entire training set. This means that we do not need the every training set to estimate the total loss.
We can approximate this expectation by a smaller subset - a `minibatch` $B \ll N$ (e.g. 8, 16,32,64,128 or as large as the (GPU) memory allows) of the data:

$$\mathbb{E}_{i \sim \mathcal{U}\{1,N\}} \left[\mathcal{L}_i(\mathbf{w}^t) \right] \approx \frac{1}{B} \sum_b \mathcal{L}_b(\mathbf{w}^t)$$

Thus the total loss can also be approximated by the following formula:

$$\frac{1}{N} \sum_i \nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) = \mathbb{E}_{i \sim \mathcal{U}\{1,N\}} \left[\nabla_{\mathbf{w}} \mathcal{L}_i(\mathbf{w}^t) \right] \approx \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$$

This represents a noisy approximation of the total loss, because training on a batch estimates the total loss, but adds a little noise as the smaller batch sizes lead to a larger variance in the gradients. These batches can either be chosen randomly or by partitioning the dataset. Either way they should be as independent as possible, therefore make sure to shuffle the training set. We also introduce some terminology here to make sure we talk about the same concept.

- Iteration = a single gradient update based on a single minibatch $\mathbf{w}^t \rightarrow \mathbf{w}^{t+1}$
- Epoch = complete pass through the training set ($= \frac{N}{B}$ iterations)

The algorithm for stochastic gradient descent now looks like this:

1. Initialize weights $\mathbf{w}^0$, pick learning rate η and minibatch size $|\mathcal{X}_{\text{batch}}|$
2. Draw random (shuffled) minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_B, \mathbf{y}_B)\} \subseteq \mathcal{X}$ (with $B \ll N$)
3. For all minibatch elements $b \in \{1, \ldots, B\}$ do:
   (a) Forward propagate $\mathbf{x}_b$ through network to calculate prediction $\hat{\mathbf{y}}_b$
   (b) Backpropagate to obtain batch element gradient $\nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t) \equiv \nabla_{\mathbf{w}} \mathcal{L}(\hat{\mathbf{y}}_b, \mathbf{y}_b, \mathbf{w}^t)$
4. Update gradients: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
5. If validation error decreases, go to step 2, otherwise stop

In the gradient update step we see that we changed the code to average over a minibatch and not the full dataset anymore.

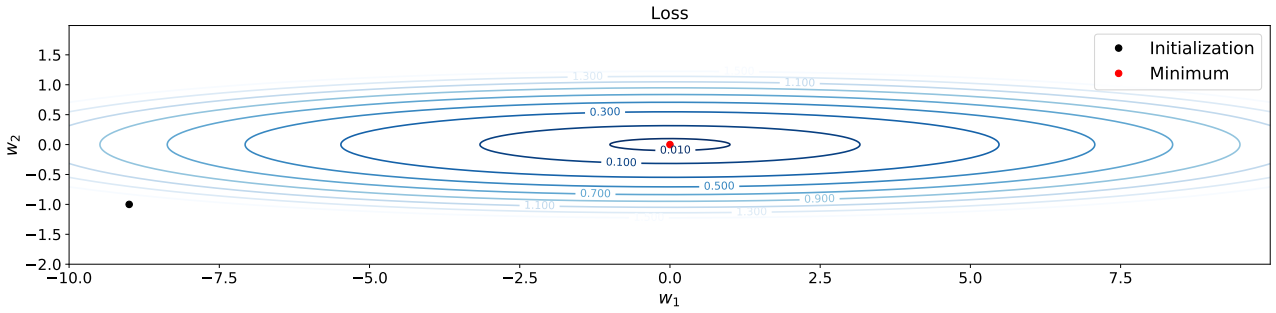


Figure 93: **SGD loss example** An example loss function for 2 parameters shown as contour plot.

Let's look at an example (Fig. 93) for a 2D parabola loss curve described by $\mathcal{L}(\mathbf{w}) = (0.1\, w_1)^2 + w_2^2$ and shown as a contour plot, where each ellipse represents a loss value. As we can see by looking at the loss the gradient slopes upward more quickly in the w_2 dimension and more slowly in the w_1 dimension (factors 1 and 0.1 respectively). The resulting gradient can be formulated by:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = (0.02\, w_1 \quad 2w_2)^\top + \mathcal{N}(0, 0.03)$$

with $\frac{\partial \mathcal{L}}{\partial w_1} = 0.02\, w_1$ and $\frac{\partial \mathcal{L}}{\partial w_2} = 2w_2$. To simulate the process of stochastic gradient descent of minibatches, we have added Gaussian noise to the gradient as well.
Although SGD helps with gradient descent's efficiency, the problem of how to choose the learning rate still exists:
In Fig. 94 we can observe the effect a learning rate that is chosen too low (very slow convergence - top) or too high (divergence - bottom) has on the convergence of the model. In the case of a learning rate that is too high, every update overshoots the previous w^2 which leads to oscillation and divergence, although it gets closer to the optimum in the w_1 dimension. We can also see the effect of the noise as the points along w_1 are not equally distributed.
When choosing a better learning rate, the effect is obvious (Fig. 95):
The update path is still oscillating and slow, but does not diverge anymore.

6.2.3 Finding the right Learning Rate

The question arises on how to find the right learning rate. One possible solution is the line search:

1. Compute minibatch gradient: $\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \equiv \frac{1}{B} \sum_b \nabla_{\mathbf{w}} \mathcal{L}_b(\mathbf{w}^t)$
2. Find optimal step size: $\eta^* = \text{argmin}_{\,\eta}\, \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t))$
3. Update weights: $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta^* \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$

In which we try to find the optimal learning rate for each update independently such that we get the best update for the parameters. But this creates another optimization problem on its own, because we need to solve a very large system at each step. Thus making it impractical for deep learning and not useful in practice.

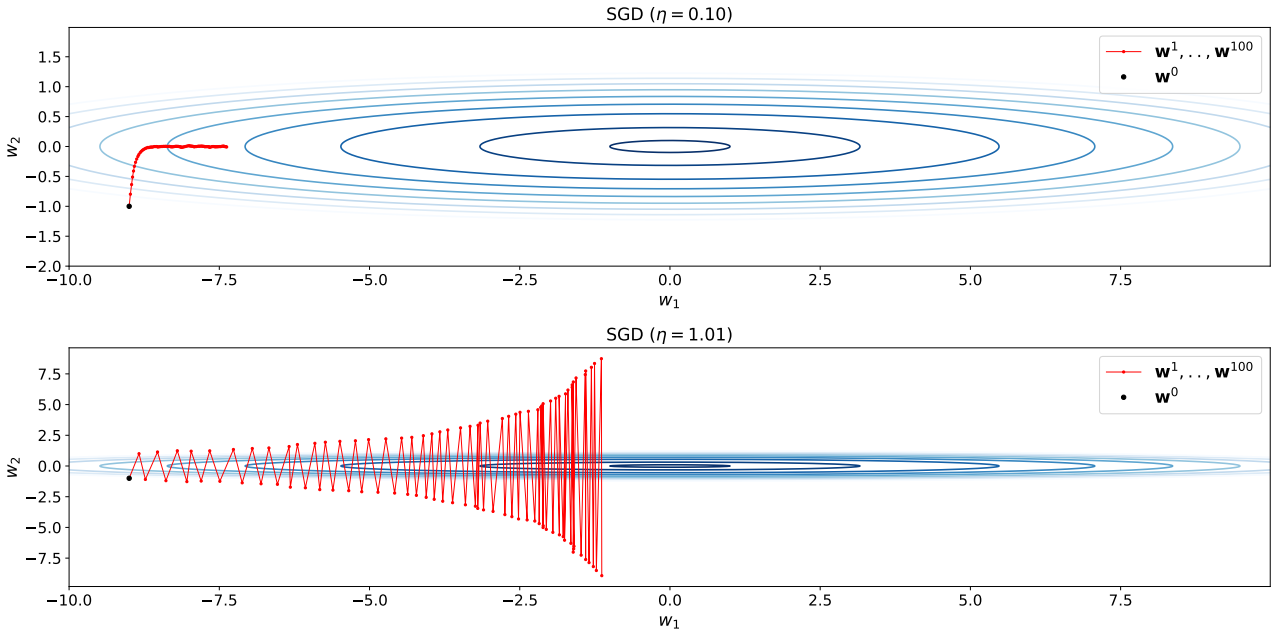


Figure 94: **SGD bad learning rates** Different learning rates on the previous toy example. Top: very low learning rate. Bottom: too high learning rate.

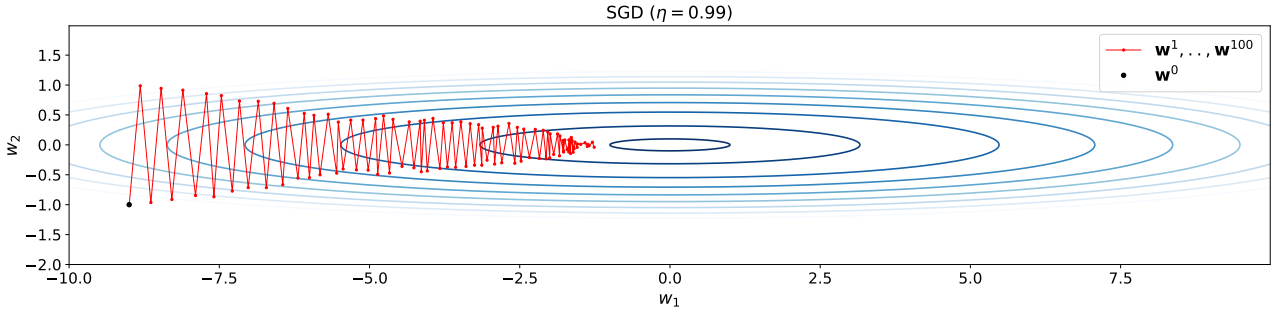


Figure 95: **SGD good learning rate** Good learning rate on the previous toy example.

6.2.4 Convergence of SGD

A general problem that SGD has over GD is that due to the stochasticity a fixed learning rate η will never lead to convergence, because of the noise that is added by using minibatches. This is nicely shown by Fig. 96 where the weights have been initialized at the optimum: $\mathbf{w}^0 = 0$. Stochastic gradient descent will always step over the optimum with a fixed learning rate, even when starting at the optimum. Let's look at this more in depth in the following.

In general, a **series** is the sum of terms of an infinite sequence of numbers $(a_1, a_2, \ldots)$

$$s_n = \sum_{k=1}^{n} a_k \qquad n \to \infty$$

A series is **convergent** if there exists a number s^* such that for every arbitrarily small positive number ϵ, there exists an integer N such that for all $n \geq N$:

$$|s_n - s^*| < \epsilon \tag{23}$$

Which means that if we go for long enough we will be arbitrarily close to the optimal solution.

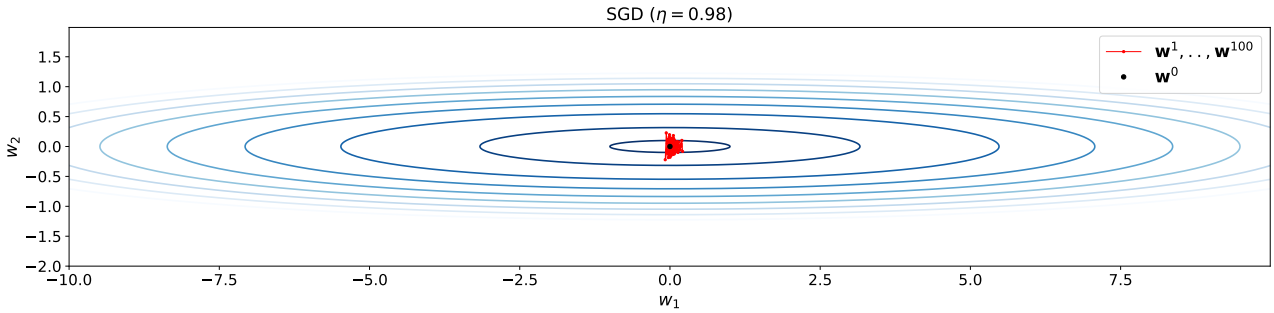


Figure 96: **SGD not converging** Good learning rate on the previous toy example shows non-convergence at optimum.

In the case of the SGD update step $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$, this results in:

$$\begin{aligned}
\mathbf{w}^0 &= \mathbf{w}^{\text{init}} \\
\mathbf{w}^1 &= \mathbf{w}^0 - \eta \nabla_{\mathbf{w}} \mathcal{L}_0 \\
\mathbf{w}^2 &= \mathbf{w}^1 - \eta \nabla_{\mathbf{w}} \mathcal{L}_1 = \mathbf{w}^0 - \eta \nabla_{\mathbf{w}} \mathcal{L}_0 - \eta \nabla_{\mathbf{w}} \mathcal{L}_1 \\
\mathbf{w}^3 &= \mathbf{w}^2 - \eta \nabla_{\mathbf{w}} \mathcal{L}_2 = \underbrace{\mathbf{w}^0 - \eta \nabla_{\mathbf{w}} \mathcal{L}_0}_{=\mathbf{a}_1} \underbrace{- \eta \nabla_{\mathbf{w}} \mathcal{L}_1}_{=\mathbf{a}_2} \underbrace{- \eta \nabla_{\mathbf{w}} \mathcal{L}_2}_{=\mathbf{a}_3}
\end{aligned}$$

with elements a_1, a_2 and a_3 being elements of a series. By applying the convergence criterion (23) we can conclude that optimization **converges** if there exists a vector $\mathbf{w}^*$ such that for every arbitrarily small positive number ϵ, there exists an integer T such that for all $t \geq T$:

$$\|\mathbf{w}^t - \mathbf{w}^*\| < \epsilon$$

The convergence theorem is given by [10]:

Let $(\eta_1, \eta_2, \dots)$ be a sequence of positive step sizes with

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty$$

and let $\mathbf{g}_t$ be an unbiased estimate of the gradient $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$, i.e., $\mathbb{E}[\mathbf{g}_t] = \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$.
Then, the series

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \mathbf{g}_t \qquad t \to \infty$$

converges to a local minimum of $\mathcal{L}(\mathbf{w})$.

This means on the one hand that we can guarantee that SGD converges to a local minimum when using a decaying learning rate (like $\eta_t = \frac{\eta}{t}$), on the other hand we cannot guarantee that it converges to a global minimum for a non-convex loss function. Although we can guarantee that it converges to the global minimum for a convex loss function, because the local and global minima are the same.

In conclusion, the problems of SGD are that the contribution of gradients to the update are scaled equally across all dimensions. SGD requires a conservative learning rate to avoid divergence, which then slows down the process. In general, finding a good learning rate is difficult.

6.2.5 SGD with Momentum

To help improve the update process of SGD, we introduce SGD with Momentum [12]. By applying a exponential moving average (Section 6.2.6) of gradients to the weight update, we intend to dampen the oscillation and accelerate the progress towards the optimum. The resulting improved path of weight updates is seen in Fig. 97.

The velocity term $\mathbf{m}$ in the update equations

$$\begin{aligned}
\mathbf{m}^{t+1} &= \beta_1 \, \mathbf{m}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \\
\mathbf{w}^{t+1} &= \mathbf{w}^t + \mathbf{m}^{t+1}
\end{aligned}$$

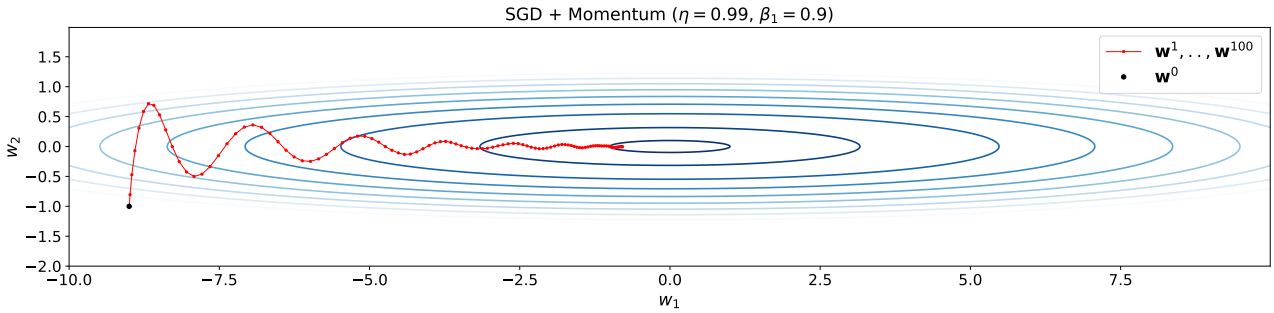


Figure 97: **SGD with Momentum** SGD with Momentum learning on the previous toy example.

is responsible for the improvement, by keeping a little bit (β_1; typically $\beta_1 = 0.9$) of the motion from the previous iteration $\mathbf{m}^t$ in the current update $\mathbf{m}^{t+1}$. By setting $\beta_1 = 0$, we can simulate the standard stochastic gradient descent. Although momentum is introduced as stated above, a better parameterization is the following linear combination

$$\mathbf{m}^{t+1} = \beta_1 \mathbf{m}^t + (1 - \beta_1) \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$$
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \mathbf{m}^{t+1}$$

as it decouples the momentum β_1 and learning rate η hyperparameters. We can now independently change the momentum and the learning rate and inspect their behaviour.

6.2.6 Exponential Moving Average

What both of the previous equations for $\mathbf{m}$ implement is effectively an exponential moving average of the gradient. Let's take a look at the following, to see why it is a exponential moving average.

Let us abbreviate the gradient at iteration t with $\mathbf{g}_t \equiv \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)$. We have:

$$\begin{aligned}
\mathbf{m}^{t+1} &= \beta_1 \mathbf{m}^t + (1 - \beta_1) \mathbf{g}_t \quad (\text{with } \mathbf{m}^0 = \mathbf{0}) \\
\mathbf{m}^1 &= \beta_1 \mathbf{m}^0 + (1 - \beta_1) \mathbf{g}_0 = (1 - \beta_1) \mathbf{g}_0 \\
\mathbf{m}^2 &= \beta_1 \mathbf{m}^1 + (1 - \beta_1) \mathbf{g}_1 \\
&= \beta_1 (1 - \beta_1) \mathbf{g}_0 + (1 - \beta_1) \mathbf{g}_1 \\
\mathbf{m}^3 &= \beta_1 \mathbf{m}^2 + (1 - \beta_1) \mathbf{g}_2 \\
&= \beta_1^2 (1 - \beta_1) \mathbf{g}_0 + \beta_1 (1 - \beta_1) \mathbf{g}_1 + (1 - \beta_1) \mathbf{g}_2
\end{aligned}$$

We see that the **weight decays exponentially**:

$$\mathbf{m}^t = (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1} \mathbf{g}_i$$

This shows that the contribution of gradients in the past is much smaller than the contribution of gradients that are closer to timestep t, as the $-i$ in the exponent of β_1^{t-i-1} marks. Although the contribution gets smaller and smaller for previous iterations, it's always there.

Depending on how β_1 is set, we either track the gradient more quickly (see Fig. 98 left) or slowly (Fig. 98 right) and sometimes not even reach the original value of the gradient.
We see that the average effectively dampens the behaviour of the function.

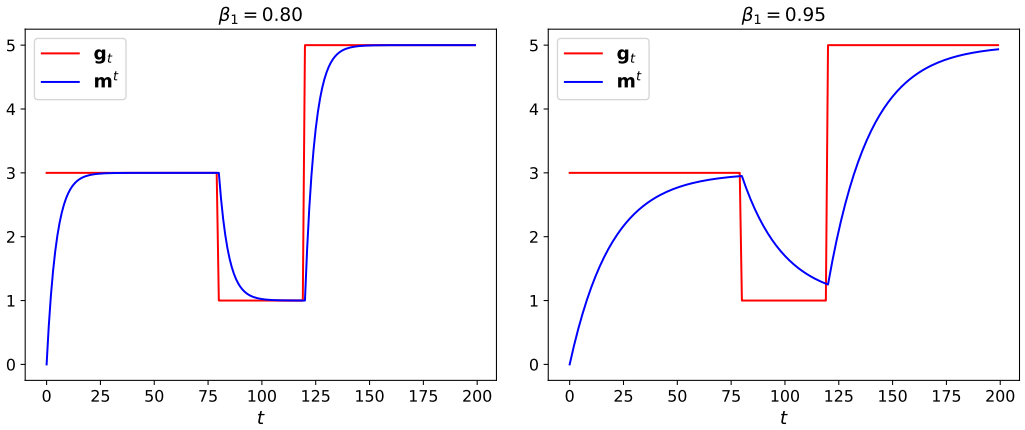


Figure 98: **Exponential Moving Average** An example that shows the dampening behaviour of EMA on the momentum.

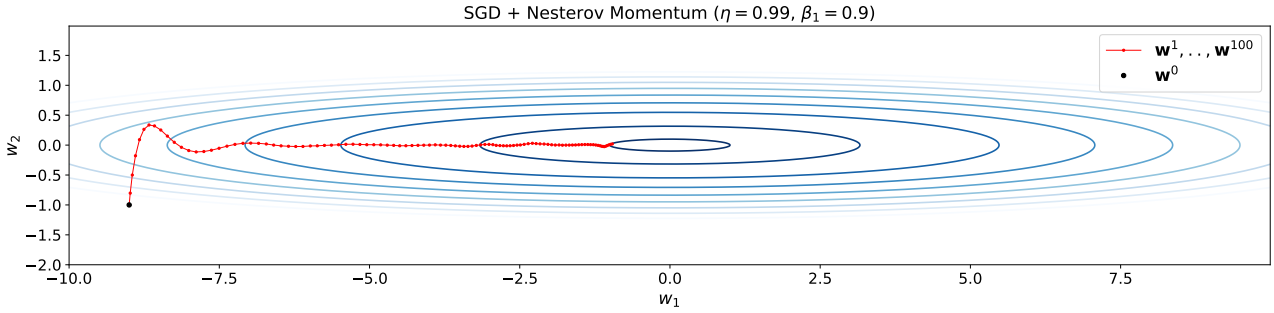


Figure 99: **SGD with Nesterov Momentum** An example that shows the improved dampening behaviour of SGD with Nesterov Momentum.

6.2.7 SGD with Nesterov Momentum

Another approach of adding momentum to SGD is with the Nesterov Momentum that builds upon the standard momentum, by looking one step ahead to calculate the gradient wrt. predicted parameters $\hat{\mathbf{w}}^{t+1}$:

$$\begin{aligned}\hat{\mathbf{w}}^{t+1} &= \mathbf{w}^t + \beta_1 \mathbf{m}^t \\ \mathbf{m}^{t+1} &= \beta_1 \mathbf{m}^t - \eta \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\hat{\mathbf{w}}^{t+1}) \\ \mathbf{w}^{t+1} &= \mathbf{w}^t + \mathbf{m}^{t+1}\end{aligned}$$

Again, this expression can be rewritten such that the hyperparameters are decoupled:

$$\begin{aligned}\hat{\mathbf{w}}^{t+1} &= \mathbf{w}^t - \eta \beta_1 \mathbf{m}^t \\ \mathbf{m}^{t+1} &= \beta_1 \mathbf{m}^t + (1 - \beta_1) \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\hat{\mathbf{w}}^{t+1}) \\ \mathbf{w}^{t+1} &= \mathbf{w}^t - \eta \mathbf{m}^{t+1}\end{aligned}$$

The predicted parameters $\hat{\mathbf{w}}^{t+1}$ estimate where we might end up by updating with the previous velocity $\mathbf{m}^t$, but without the current gradient. This alternative to the standard momentum increases the responsiveness of momentum, because we take a look into the future, thus it leads to faster dampening as shown in Fig. 99.

6.2.8 RMSProp

An approach with a the same motivation as momentum is RMSprop. We want to have an even distribution of gradients on each weight dimension, in contrast to the standard SGD (see Fig. 95) with an very uneven gradient distribution. The idea of RMSprop is to divide the learning rate by a moving average of **squared gradients**, which means that we change the learning rate per parameter, i.e. w_1 and w_2 in Fig. 95.
The moving average of squared gradients or the **running variance v** is used in this approach to adjust the

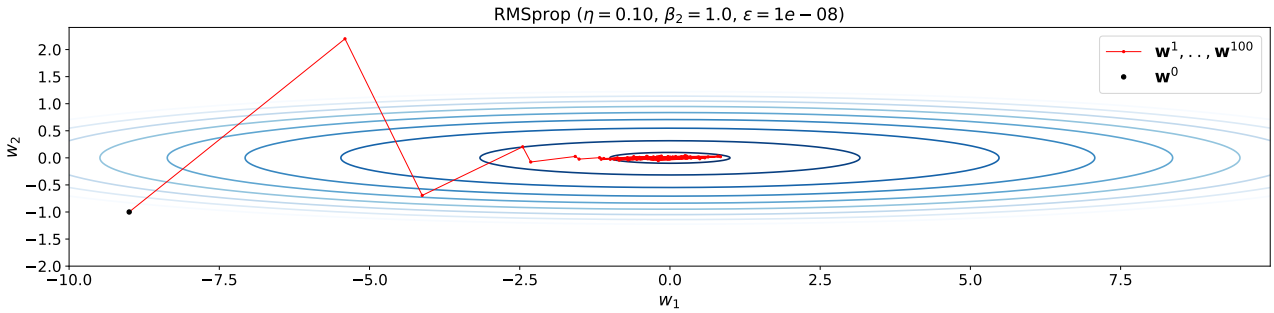


Figure 100: **RMSprop** An example that shows the update behaviour of RMSprop.

per-weight step size (e.g. division in w_2 direction will be larger, division in w_1 direction will be small).

$$\mathbf{v}^{t+1} = \beta_2 \mathbf{v}^t + (1 - \beta_2) \left(\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \odot \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \right) \tag{24}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \frac{\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t)}{\sqrt{\mathbf{v}^{t+1}} + \epsilon} \tag{25}$$

By only considering the squared gradient and not the mean, we end up with an uncentered variance of the gradient $\mathbf{v}$. To make sure that we only apply it per parameter, all operations are elementwise. The effect of this per-weight scaling can be seen in Fig. 100, which converges much faster to the optimum. However we see that in the first few iterations the updates show a jumping behaviour. That is due to a bias towards zero, because we divide by a very small number $\mathbf{v}$ at the beginning due to initialization of $\mathbf{v}^0 = 0$. This problem is solved with the next algorithm.

6.2.9 Adam

Adam is the most used and de facto default optimizer, due to its robustness, as it combines all previous mentioned ideas of Section 6.2.5 and Section 6.2.8. It is also possible to combine Adam with Nesterov's Momentum, although not many people use it. The benefits can be seen in Fig. 101 in contrast to Fig. 95.

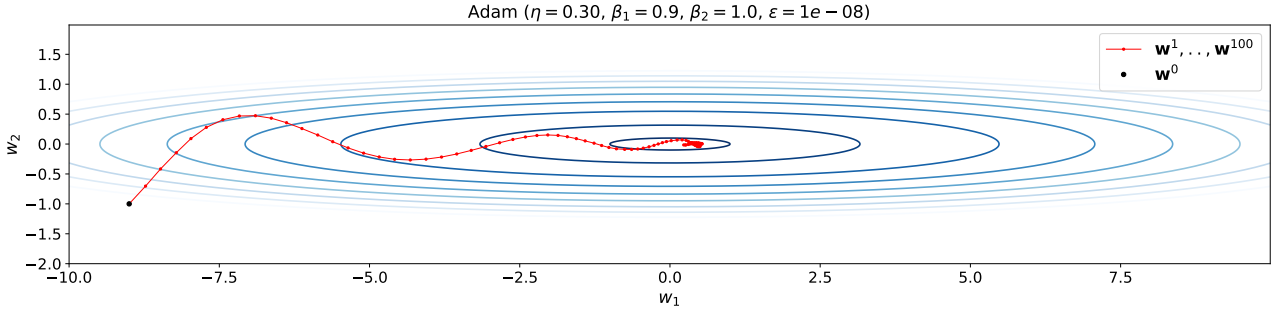


Figure 101: **Adam** An example that shows the update behaviour of Adam.

Given the update equation we can clearly see the influences of Momentum and RMSprop as both the first-moment velocity term $\mathbf{m}$ and the second-moment variance term $\mathbf{v}$ are used.

$$\begin{aligned}
\mathbf{m}^{t+1} &= \beta_1 \mathbf{m}^t + (1 - \beta_1) \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \\
\mathbf{v}^{t+1} &= \beta_2 \mathbf{v}^t + (1 - \beta_2) \left(\nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \odot \nabla_{\mathbf{w}} \mathcal{L}_{\mathcal{B}}(\mathbf{w}^t) \right) \\
\hat{\mathbf{m}}^{t+1} &= \frac{\mathbf{m}^{t+1}}{1 - \beta_1^{t+1}} \qquad \hat{\mathbf{v}}^{t+1} = \frac{\mathbf{v}^{t+1}}{1 - \beta_2^{t+1}} \\
\mathbf{w}^{t+1} &= \mathbf{w}^t - \eta \frac{\hat{\mathbf{m}}^{t+1}}{\sqrt{\hat{\mathbf{v}}^{t+1}} + \epsilon}
\end{aligned}$$

In addition, Adam uses a bias correction, namely $\hat{\mathbf{m}}^{t+1}$ and $\hat{\mathbf{v}}^{t+1}$. But why does this help remove the bias?

6.2.10 Bias Correction

Let $\mathbf{g}_t = \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^t)$ denote the gradient of the stochastic objective $\mathcal{L}(\mathbf{w}^t)$.
Let further $\mathbf{m}^0 = \mathbf{0}$. Then, the update $\mathbf{m}^{t+1} = \beta_1 \mathbf{m}^t + (1 - \beta_1)\mathbf{g}_t$ can be written as:

$$\mathbf{m}^t = (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1} \mathbf{g}_i$$

Thus, the expectation over $\mathbf{m}^t$ is given by:

$$\mathbb{E}[\mathbf{m}^t] = \mathbb{E}\left[(1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^{t-i-1} \mathbf{g}_i\right]$$

Because the expectation is a linear operation we can push it through into the sum. We can approximate the expectation of previous time steps by $\mathbb{E}[\mathbf{g}_t]$ and pull it out of the sum again. Also we switch the sum afterwards from counting downwards to counting upwards:

$$\approx \mathbb{E}\left[\mathbf{g}_t\right] \cdot (1 - \beta_1) \sum_{i=0}^{t-1} \beta_1^i$$

The sum is now a mathematical series for which a simple solution exists:

$$= \mathbb{E}\left[\mathbf{g}_t\right] \cdot (1 - \beta_1^t)$$

Here we can see that the term $(1 - \beta_1^t)$ is the factor under the fraction in the bias correction term $\hat{\mathbf{m}}$.

6.2.11 Second-order Methods

There are also some second-order methods that we've not looked into, like Newton, Gauss-Newton etc. These second-order methods are faster as they exploit the curvature of the loss function, however they are not applicable to mini-batches as the Hessian matrix estimates are too inaccurate if computed from small batches. Also they are not tractable as they require the inversion of a very large matrix, thus they are typically not applied in deep models.

6.3 Optimization Strategies

6.3.1 Learning Rate Schedules

We have already seen that the learning rate should decrease with time, although not too fast as we might get stuck in a region that is far from a good one, i.e. a local minimum. This is where learning rate schedules can help. The following are some learning rate schedules that we can make use of:

- Fixed learning rate (not a good idea: too slow in the beginning and fast in the end)
- Inverse proportional decay: $\eta_t = \eta/t$ (Robbins and Monro)
- Exponential decay: $\eta_t = \eta\alpha^t$
- Step decay: $\eta \leftarrow \alpha\eta$ (every K iterations/epochs, common in practice: $\alpha = 0.5$)

6.3.2 Monitoring the Training Process

Maybe just as important as choosing the right learning rate schedule is to recognize common training patterns that can indicate to us how the model is currently training. In the following we see multiple learning loss plot patterns that indicate some learning behaviour:

6.3.3 Hyperparameter Search

It is also important to choose the hyperparameters right, because they are parameters that are not optimized by gradient descent, e.g. parameters of the network are not hyperparameters. In general, it is good to have less hyperparameters to search the space of possible values efficiently as hyperparameter search is a difficult problem. Thus often still based on human intuition.
The most common methods of finding hyperparameters are:

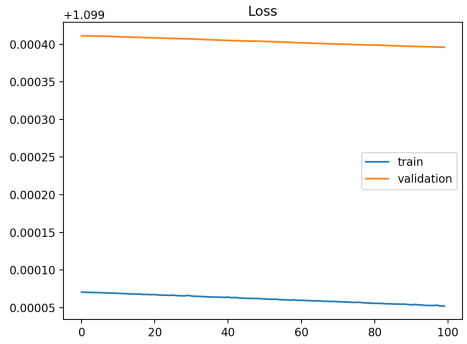


Figure 102: **Underfitting** Model does not have enough capacity to decrease losses. The typical steep decline at the start is missing.

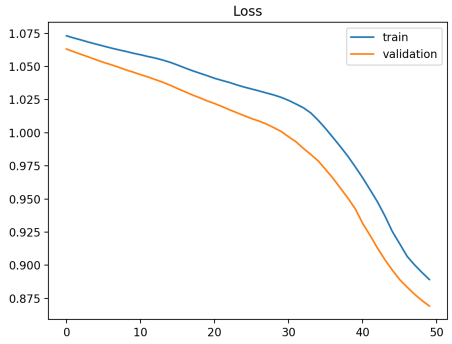


Figure 103: **Not converged** Model requires more iterations to converge. We have to run training for longer time.

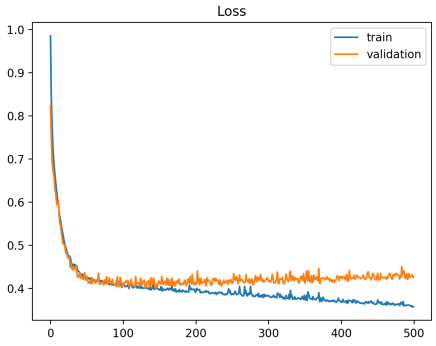


Figure 104: **Overfitting** Training loss decreases, but validation loss increases. We can use regularization (e.g. early stopping) or change the model capacity to avoid.

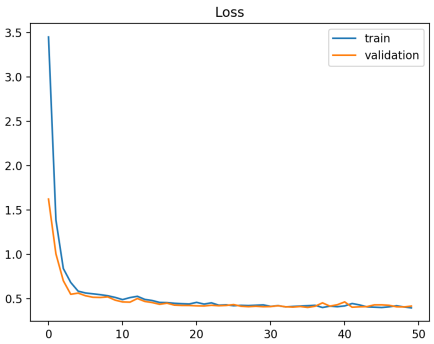


Figure 105: **Good example** Example of train and validation curves that show a good fit.

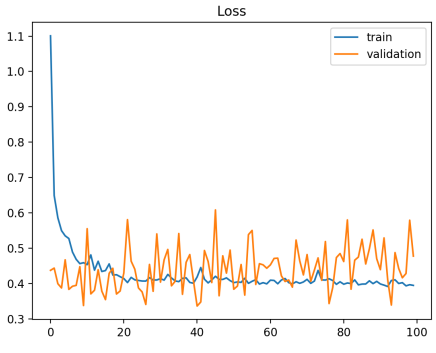


Figure 106: **Noisy validation curves** The validation set might be too small such that we cannot observe something meaningful. Increase the validation set size.

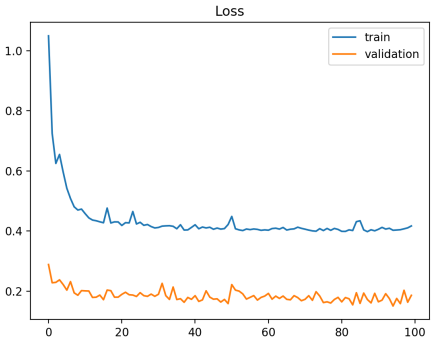


Figure 107: **Validation set easier** It might also happen that the validation set is easier to predict than the training set.

6.3.4 How to Start

When starting a new DL project and we want to make sure that our model is correct, we can go through the following steps:

1. Start with single training sample and use a small network

- First verify that the output is correct
- Then overfit, accuracy should be 100%, fast training/debug cycles

- **Manuel Search**
  It is still the most common. E.g. look in the neighborhood of your current hyperparameters.
- **Grid Search**
  Define ranges in which you want to systematically evaluate hyperparameters (use human intuition to define ranges). Works well, but is very expense, because we need to run a lots of full trainings.
- **Random Search** Like grid search, but hyperparameters selected based on random draws.

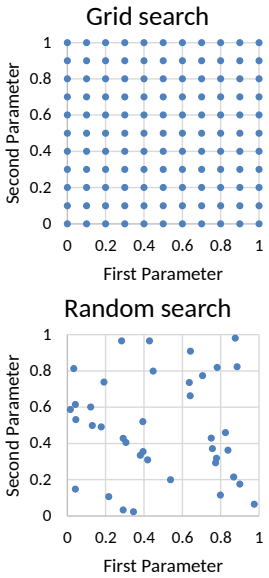


Figure 108: **Grid and Random search** An exmaple for hyperparameter search using grid and random search.

- Choose a good learning rate $(0.1, 0.01, 0.001, ..)$

2. Increase to 10 training samples

- Again, verify that the output is correct
- Measure time for one iteration $(< 1s) \Rightarrow$ identify bottlenecks (e.g., data loading)
- Overfit to 10 samples, accuracy should be near 100%

3. Increase to 100, 1000, 10000 samples and increase network size

- Plot train and validation error $\Rightarrow$ now you should start to see generalization
- Important: Make only one change at a time to identify causes

6.3.5 Improving Gradient Flow

One of the most important requirements of a deep neural network is that the gradient flow has to work correctly. If we don't have a proper gradient flow, there might be some parameters that are not or incorrectly updated. To make sure that this is not the case, we can visualize the gradient flow. There are multiple possible solutions to improve the gradient flow: Initialization, Batch Normalization and Residual Networks.

Initialization
The initialization of the network's weights is very important, because depending on the initialization the activation of layers might get worse layer after layer. Therefore we should consider using Xavier or He initialization to ensure that the activation distribution is a constant Gaussian across all layers.

Batch Normalization
The Batch Normalization normalizes **each channel** individually by mean and variance over the batch as shown in the follow equations by the indices b and c.

$$\mu_c = \frac{1}{B}\sum_{b=1}^{B} x_{b,c}$$

$$\sigma_c^2 = \frac{1}{B}\sum_{b=1}^{B}(x_{b,c} - \mu_c)^2$$

$$\hat{x}_{b,c} = \frac{x_{b,c} - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

$$y_{b,c} = \gamma_c \hat{x}_{b,c} + \beta_c$$

Similar to Momentum and RMSprop, we want to make sure that any bias is removed before applying the subsequent operations. Because we have removed any bias, we want to add a learnable bias β_c as well as a learnable scale parameter γ_c after that. While training the batch normalization layers, that are placed before the activation function, saves a running average of mean and variance that will be applied at test time to estimate the mean and variance during training.
Of course there are other forms of normalization that we can add to our DNN:

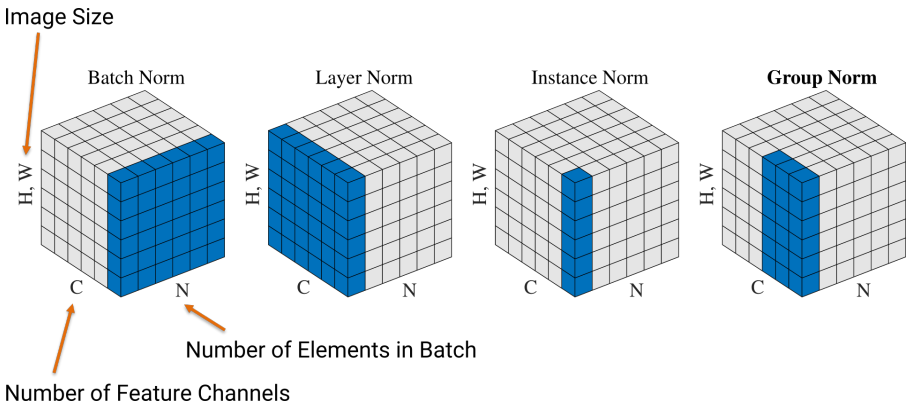


Figure 109: **Normalization Variations** An example of different normalization variations.

Residual Networks
Especiall deep networks have the problem that gradients are propagated very slowly and once they arrive at early layers they are very small (*vanishing gradient*-problem). The observation is that deeper networks often perform worse than more shallow ones, which is counter-intuitive, because if we have a shallow network and add identity layers/transformations, we get the same performance as the shallow network. So the deep model should be in theory at least as good in terms of accuracy as the shallow model, which is not the case. Therefore residual networks have been proposed to learn the residual mappings of a few layers i.e. 2, by introducing a skip connection across these layers (see Fig. 110), which helps the gradient flow. These layers now only have to learn a delta of the input towards their output.

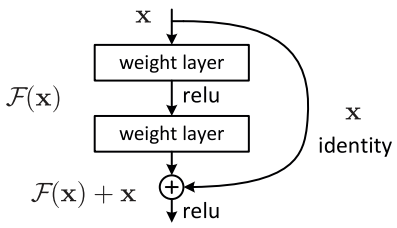


Figure 110: **Residual Network** An example of a residual network unit.

6.3.6 Training Schedules

Lastly, there are some common training schedules that are useful to know when dealing with DL.

Pretraining
Pretraining is most useful, when we don't have a large enough dataset to train a full network. We first pretrain our backbone of the DNN (e.g. convolution layers) on another task for which a large dataset with labels is available (e.g. ImageNet). Then we **finetune** the last layers on our target task/dataset, which should yield a significant improvement.

Self Supervision
Similar to pretraining, we want to pretrain the backbone on a task for which supervision is generated from the data itself (e.g. denoising, inpainting, contrastive learning).

Curriculum Learning
Curriculum Learning is the process of starting training on a easy dataset and then successively increasing the difficulty for the network. We want the network to work its way step-by-step to the original full dataset, such that very difficult samples don't take the training off course.

6.4 Debugging Strategies

In deep learning many things can go wrong, therefore we will discuss some common failure/errors cases and how to solve them in the following section about debugging strategies.

6.4.1 Emergency First Response

Sometimes it is hard to find the problems for unexpected outputs, especially when training with a large dataset, where one training might take days or weeks. For these cases there are the following points as a *emergency first response* to cover the most common problems that could occur. As a general advise, it is best to try one step at a time to see what each change results in.

- **Start with a simple model that is known to work for this type of data (for example, VGG for images). Use a standard loss if possible.** Make sure that we have something working to build up on.
- **Turn off all bells and whistles, e.g. regularization and data augmentation.** We want to make sure that we simplify complex systems to make it easier to debug. Also reduce the size of the dataset to increase debug cycles.
- **If finetuning a model, double check the preprocessing, for it should be the same as the original model's training.**
- **Verify that the input data is correct.** It is very easy to get the input wrong, try to visualize to double-check if it is correct.
- **Start with a really small dataset (1–10 samples).** Overfit on it and gradually add more data.
- **Start gradually adding back all the pieces that were omitted: augmentation/regularization, custom loss functions, try more complex models.**

6.4.2 Dataset Issues

Check your input data
Check if the input data you are feeding the network makes sense. Make sure to not mix up width and height of an image. Or that the image is actually filled with data, not all zeros. So print/display a couple of batches of input and target output and make sure they are OK.

Try random input
Try passing random numbers instead of actual data and see if the error behaves the same way. If it does, it's a sure sign that your net is turning data into garbage at some point. Try debugging layer by layer /op by op/ and see where things go wrong.

Check the data loader
Your data might be fine but the code that passes the input to the net (i.e. the data loader) might be broken. Print the input of the first layer before any operations and check it.

Make sure input is connected to output
Check if a few input samples have the correct labels. Also make sure shuffling input samples works the same way for output labels.

Verify noise in the dataset
I can happen that your dataset has a lot of bad labels, such that the network cannot learn. Check a bunch of input samples manually and see if labels seem off.

Shuffle the dataset
If your dataset hasn't been shuffled and has a particular order to it (ordered by label) this could negatively impact the learning, as nearby samples in the same minibatch might be highly correlated. Shuffle your dataset to avoid this. Make sure you are shuffling input and labels together.

Reduce class imbalance
Are there a 1000 class A images for every class B image? Then you might need to balance your loss function if both classes are equally important or try other class imbalance approaches.

Verify number of training examples
If you are training a net from scratch (i.e. not finetuning), you probably need lots of data. For image classification, people say you need a 1000 images per class or more. If this is not possible, try to use more data augmentation or pretraining to reduce the number of needed samples.

Make sure your batches don't contain a single label
This can happen in a sorted dataset (i.e. the first 10k samples contain the same class). Easily fixable by shuffling the dataset.

Use a standard dataset
When testing new network architecture or writing a new piece of code, use the standard datasets first, instead of your own data. This is because there are many reference results for these datasets and they are proved to be 'solvable'. There will be no issues of label noise, train/test distribution difference etc.

6.4.3 Data Normalization / Augmentation Issues

Standardize the features.
Make sure to standardize your input to have zero mean and unit variance.

Check for too much data augmentation
Augmentation has a regularizing effect. Too much of this combined with other forms of regularization (weight L2, dropout, etc.) can cause the network to underfit.

Check the preprocessing of your pretrained model
If you are using a pretrained model, make sure you are using the same normalization and preprocessing as the model was when training and testing. For example, should an image pixel be in the range [0, 1] or [0, 255]?

Check the preprocessing for train/validation/test set
Any preprocessing statistics (e.g. the data mean) must only be computed on the training data, and then applied to the validation/test data. E.g. computing the mean and subtracting it from every image across the entire dataset and then splitting the data into train/val/test splits would be a mistake.

6.4.4 Implementation Issues

Try solving a simpler version of the problem
This will help with finding where the issue is. For example, if the target output is an object class and coordinates, try limiting the prediction to object class only.

Make sure your training procedure is correct
It is easy to forget toggling train/validation mode. Make sure you are using the right inputs. It is also easy to forget setting the gradients to zero before backpropagation.

Check your loss function
If you implemented your own loss function, check it for bugs and add unit tests. Often, wrong losses hurt the performance of the network in a subtle way.

Verify loss input
If you are using a loss function provided by your framework, make sure you are passing to it what it expects. For example, in PyTorch I would mix up the NLLLoss and CrossEntropyLoss as the former requires a softmax input and the latter doesn't.

Adjust loss weights
If your loss is composed of several smaller loss functions, make sure their magnitude relative to each is correct. This might involve testing different loss weights.

Monitor other metrics
Sometimes the loss is not the best predictor of whether your network is training properly. If you can, use other metrics like accuracy. In general, it is desirable to use as little loss functions as possible to reduce the number of hyperparameters.

Test any custom layers
Did you implement any of the layers in the network yourself? Check and double-check to make sure they are working as intended. Make sure the output has the right format (e.g., did you pass a softmax to a loss that expects raw logits?)

Check for "frozen" layers or variables
Check if you unintentionally disabled gradient updates for some layers/variables that should be learnable.

Increase network size
Maybe the expressive power of your network is not enough to capture the target function. If you observe underfitting, try adding more layers or more hidden units in fully connected layers.

Check for hidden dimension errors
If your input looks like (k, H, W) = (64, 64, 64) it's easy to miss errors related to wrong dimensions. Use weird numbers for input dimensions (for example, different prime numbers for each dimension) and check how they propagate through the network and that all dimensions of intermediate/hidden layers are correct.

Explore gradient checking
If you implemented Gradient Descent by hand, gradient checking makes sure that your backpropagation works like it should.

6.4.5 Training Issues

Solve for a really small dataset
Overfit a small subset of the data and make sure it works. For example, train with just 1 or 2 examples and see if your network can learn to differentiate these. Move on to more samples per class.

Check weights initialization
If unsure, use Xavier or He initialization. Also, your initialization might be leading you to a bad local minimum, so try a different initialization and see if it helps.

Change your hyperparameters
Maybe you using a particularly bad set of hyperparameters. Try a grid search.

Reduce regularization
Too much regularization can cause the network to underfit badly. Reduce regularization such as dropout, batch norm, weight/bias L2 regularization, etc.

Give it time
Maybe your network needs more time to train before it starts making meaningful predictions. If your loss is steadily decreasing, let it train some more.

Switch from Train to Test mode
Some frameworks have layers like Batch Norm, Dropout, and other layers behave differently during training and testing. Switching to the appropriate mode might help your network to predict properly.

Visualize the training process

- Monitor the activations, weights, and updates of each layer. Make sure their magnitudes match. For example, the magnitude of the updates to the parameters (weights and biases) should be 1-e3.
- Consider a visualization library like Tensorboard and Crayon. In a pinch, you can also print weights/biases/activations.
- Be on the lookout for layer activations with a mean much larger than 0. Try Batch Norm, ELUs or other forms of activations.
- Weight histograms should have an approximately Gaussian (normal) distribution, after some time. For biases, these histograms will generally start at 0, and will usually end up being approximately Gaussian. Keep an eye out for parameters that are diverging or biases that become very large.

Try a different optimizer
Your choice of optimizer shouldn't prevent your network from training unless you have selected particularly

bad hyperparameters. However, the proper optimizer for a task can be helpful in getting the most training in the shortest amount of time.

Exploding / Vanishing gradients
Check layer updates, as very large values can indicate exploding gradients. Gradient clipping may help. Check layer activations. A good standard deviation for the activations is on the order of 0.5 to 2.0. Significantly outside of this range may indicate vanishing or exploding activations.

Increase/Decrease Learning Rate
A low learning rate will cause your model to converge very slowly. A high learning rate will quickly decrease the loss in the beginning but might have a hard time finding a good solution. Play around with your current learning rate by multiplying it by 0.1 or 10.

Overcoming NaNs

- Decrease the learning rate, especially if you are getting NaNs in the first 100 iterations.
- NaNs can arise from division by zero or natural log of zero or negative number.
- Try evaluating your network layer by layer and see where the NaNs appear.

7 Convolutional Neural Networks

The deep learning revolution started with a significant performance increase on the ImageNet competition by AlexNet in 2012 [8]. AlexNets' success is due to using many layers, using convolutional layers, and a decrease of features and the depth of the network. In the following section, techniques relating to convolutional neural nets (CNNs) are introduced.

7.1 Prerequisites

In the following, we use Einstein Notation. Capital letters denote tensor slices. One element of a matrix A is denoted by $A[i, j]$. The i'th row of matrix A is accessed with $A[i, J]$ and the j'th column by $A[I, j]$. A full matrix uses both capital letters such that $A[I, J]$ denotes A. The same holds for higher-order tensors, $H[i, j, k]$ denotes one element of the tensor H. Repeated capital letters in a product denote summation over those letters. The Einstein Notation of a matrix product $y = Ax$ is given by

$$\begin{aligned} \mathbf{y} = \mathbf{A}\mathbf{x} \quad &\equiv \quad y[i] = \sum_j A[i, j]x[j] \\ &\equiv \quad y[i] = A[i, J]x[J] \end{aligned}$$

Capital J indicates the summation over the J'th dimension. The analog case for $y = x^T A$ can be written as a summation over the I'th dimension

$$\begin{aligned} \mathbf{y} = \mathbf{x}^\top \mathbf{A} \quad &\equiv \quad y[j] = \sum_i A[i, j]x[i] \\ &\equiv \quad y[j] = A[I, j]x[I] \end{aligned}$$

7.2 Convolution Layers

In contrast to multi-layer perceptrons (MLP), a CNN layer adds a spatial extension. W and H denote the width and height of an image. C denotes the number of feature channels. In a colored picture, there are three feature channels corresponding to Red, Blue, and Green. b. B represents the batch size. Figure 111a shows the spatial dimensions of two succeeding feature maps H_i, H_{i+1}, with input channels C_{in} and output channels C_{out}. Using a gray-scale image as input results in $C_{in} = 1$. Consequently a colored image yields $C_{in} = 3$.
In a standard MLP (Figure 111b), the whole input is connected to a single feature in the output layer, yielding a fully connected layer. The number of weights for a fully connected layer is calculated by

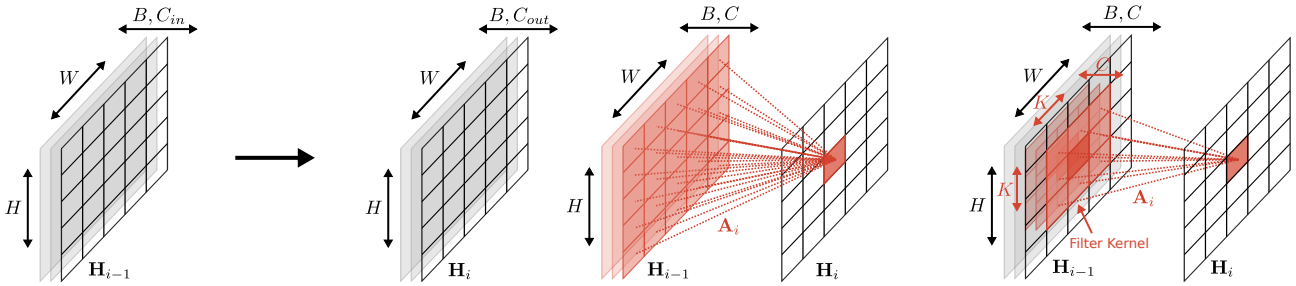
$$\#\text{Weights} = W \times H \times C_{out} \times (W \times H \times C_{in} + 1)$$

In contrast, the convolutional layer has fewer connections (Figure 111c). The weight matrix A defines a filter of a certain size. A feature in the output layer is computed by multiplying a subset of the input neurons with the weight matrix A. The subset of neurons that influence an output field is in the corresponding input field's local

neighborhood. The same filter operation is applied at every spatial location (weight sharing). A convolution filter has the size $K \times K$. The number of weights for a convolutional layer is calculated by

$$\#\text{Weights} = C_{out} \times (K \times K \times C_{in} + 1)$$

Usually, multiple convolution kernels are convolved with the input, each producing an output channel.



(a) (b) (c)

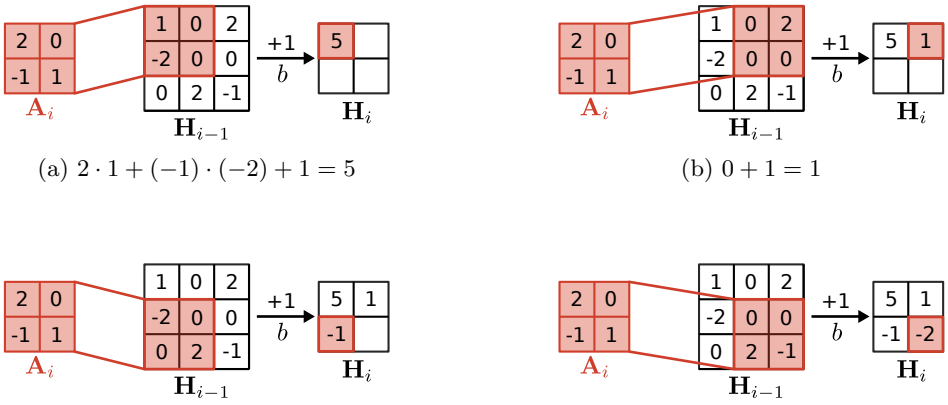
Figure 111: (a) spatial dimensions of two succeeding feature maps (b) a fully connected layer (c) convolutional layer. Only three input, one output channel and a single convolution are shown for clarity.

7.2.1 Convolution layer

The mathematical formulation for a convolutional layer is given by

$$\underbrace{H_i[b, x, y, c_{out}]}_{\text{Current Layer}} = g\left(\underbrace{A_i[\Delta X, \Delta Y, C_{in}, c_{out}]}_{\text{Weights}} \underbrace{H_{i-1}[b, x + \Delta X, y + \Delta Y, C_{in}]}_{\text{Prev. Layer}} + \underbrace{b_i[c_{out}]}_{\text{Bias}}\right)$$

where b is the batch index, x, y are the spatial locations and c_{in}, c_{out} the feature channels. The field x, y in hidden layer H_{i+1} is computed by applying the activation function g to the product of a subset of A with a subset of H_i added with the output channels bias term. The subset of A is a square, selected by $\Delta X, \Delta Y$, of a certain input channel C_{in} corresponding to a certain output channel c_{out}. The subset of H_i is a square spanned by the lengths $\Delta X, \Delta Y$, fixed around coordinates x, y of the same channel C_{in}. This operation is done for every batch b. A simple example using a filter size of two can be seen in Figure 112, where the resulting field in output map H_i is computed by point-wise multiplying the fields in A_i with the selected fields in H_{i-1} summed up and added to a bias term.



(a) $2 \cdot 1 + (-1) \cdot (-2) + 1 = 5$ (b) $0 + 1 = 1$

(c) $2 \cdot (-2) + 1 \cdot 2 + 1 = -1$ (d) $(-1) \cdot 2 + 1 \cdot (-1) + 1 = 2$

Figure 112: A convolution of input matrix A_i with input map H_{i-1} resulting in the output map H_i.

Technically, CNNs implement correlation and not convolution, which would be the case if the kernel was flipped in the above operations. This misnomer does not matter as a flipped kernel does not decrease the expressiveness of the CNN.

7.2.2 Convolution Operator

The star operator $(*)$ denotes the convolution operator. The formula of convolving feature map H with filter kernel A is given by

$$[\mathbf{A} * \mathbf{H}](\mathbf{x}) = \sum_{\Delta\mathbf{x} \in \mathbb{Z}^2} \mathbf{A}(\Delta\mathbf{x})\, \mathbf{H}(\mathbf{x} + \Delta\mathbf{x})$$

where A convolved with H and is evaluated at a particular position x. Moreover the convolution is translation equivariant. An operation $f(\cdot)$ is invariant to a transformation $\mathcal{T}_\theta$ if the functions output is the same for any transformed input $\mathcal{T}_\theta[\mathbf{H}]$

$$f(\mathbf{H}) = f(\mathcal{T}_\theta[\mathbf{H}])$$

Whereas an operation $f(\cdot)$ is equivariant if its output transforms as its input for some specific transformation type

$$\mathcal{T}_\theta[f](\mathbf{H}) = f(\mathcal{T}_\theta[\mathbf{H}])$$

CNNs are translation equivariant but not translation invariant. The CNN is equivariant because shifting the input results in a shifted output. Put in another way, by transforming the input, the feature maps are also transformed. If e.g., an image is shifted by a certain amount of pixels, the output feature maps are also shifted in the same way. The proof for the convolution operations translation equivariance is given by

$$\begin{aligned}
[\mathbf{A} * \mathcal{T}_\mathbf{t}[\mathbf{H}]](\mathbf{x}) &= \sum_{\Delta\mathbf{x} \in \mathbb{Z}^2} \mathbf{A}(\Delta\mathbf{x})\, \mathcal{T}_\mathbf{t}[\mathbf{H}](\mathbf{x} + \Delta\mathbf{x}) && \text{definition of convolution} \\
&= \sum_{\Delta\mathbf{x} \in \mathbb{Z}^2} \mathbf{A}(\Delta\mathbf{x})\, \mathbf{H}(\mathbf{x} + \Delta\mathbf{x} - \mathbf{t}) && \text{expanding translation operator} \\
&= \sum_{\Delta\mathbf{x} \in \mathbb{Z}^2} \mathbf{A}(\Delta\mathbf{x})\, \mathbf{H}((\mathbf{x} - \mathbf{t}) + \Delta\mathbf{x}) && \text{rearranging} \\
&= [\mathbf{A} * \mathbf{H}](\mathbf{x} - \mathbf{t}) = \mathcal{T}_\mathbf{t}[\mathbf{A} * \mathbf{H}](\mathbf{x}) && \text{definition of convolution}
\end{aligned}$$

.

The ConvLayer can be implemented as computation graph. Figure 113 shows the convolutional operations dependencies between input tensor (h_{11}, h_{12}, h_{13}), the output tensor (h_{21}, h_{22}) and the kernel (w_1, w_2), that is swiped along the input. The gradients get accumulated across locations due to the kernels weight sharing.

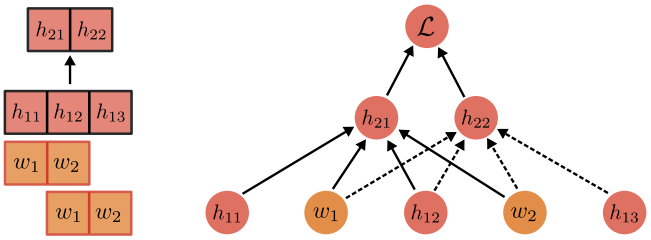


Figure 113: **Computation graph of a convolutional layer**. The arrows show dependencies in the summation. The weight w_1 appears in the summation for h_{21} and h_{22}. Therefore w_1 receives more gradients/information in back-propagation than it would have in a fully connected layer.

7.2.3 Padding

Applying a convolution kernel to all fields of the input map (Figure 114a) decreases the succeeding feature maps' size (Figure 114b). Convolutions can only be executed if the kernel lies entirely within the input domain. This decreasing in size undesirable as it couples architecture and input size. Padding adds a boundary of appropriate size with zeros around the input tensor (Figure 114d). Consequently solving the questions which values should be used for the bordering fields (Figure 114c). The convolution with a padded tensor $H'[\cdot]$ is given by

$$\underbrace{H_i[b, x, y, c_{out}]}_{\text{Current Layer}} = g\left(\underbrace{A_i[\Delta X, \Delta Y, C_{in}, c_{out}]}_{\text{Weights}}\, \underbrace{H'_{i-1}[b, x + \Delta X, y + \Delta Y, C_{in}]}_{\text{Prev. Layer}} + \underbrace{b_i[c_{out}]}_{\text{Bias}}\right)$$

. If the input is padded, then Δx and Δy are non-negative.

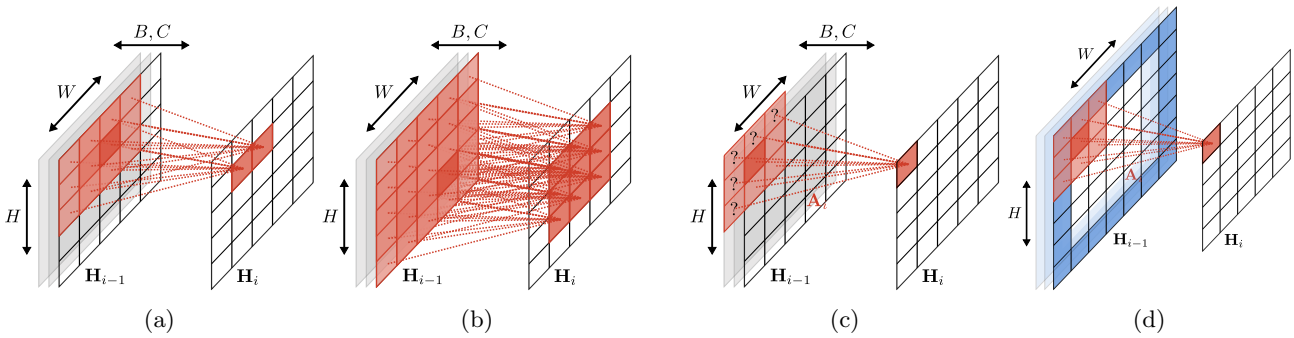


Figure 114: (a) A convolution (b) the resulting receptive field. (c) Fields with question-marks indiciate that it is unclear what values the kernel should be convolved with. (d) Padding: the input tensor is extended to the appropriate size with zeros (blue) in the rightmost image.

7.3 Downsampling

Convolution operations can sustain the resolution of the input; therefore extra downsampling operations are used to reduce the resolution of the spatial input. Downsampling is necessary when the network's input dimension is larger than the output dimensions. An example is CNNs that reduce images to few labels. Furthermore, reducing the spatial resolution increases a deep in the network located neurons receptive field.

7.3.1 Pooling

Reducing the spatial dimensions can be achieved by a pooling operation. Pooling requires no parameters and is typically one of a max, min, or mean operation. A pooling layer is defined as

$$\underbrace{H_i[b, x, y, c]}_{\text{Current Layer}} = \max_{\Delta x, \Delta y} \underbrace{H_{i-1}[b, s \cdot x + \Delta x, s \cdot y + \Delta y, c]}_{\text{Prev. Layer}}$$

where s is the stride, the steps a kernel takes. Typically a stride $s = 2$ and kernel size 2×2 are used, consequently reducing the spatial dimensions by a factor of two (Figure 115a and 115b). Pooling is applied to each channel separately, therefore retaining the number of channels from input to output. Furthermore, max-pooling provides invariance to small translations of the input. In practice, max pooling is often replaced with strided convolution (ResNet).

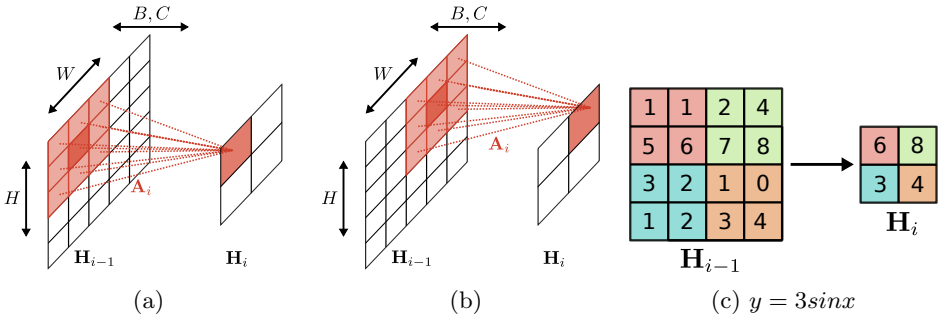


Figure 115: (a,b) A pooling operation with stride $s = 2$ for H_i's first two pixels. (c) Two consecutive feature maps H_{i-1} and H_i after applying a max-pooling operation to H_{i-1}.

7.3.2 Strided convolution

A convolutional filter is moved, similarly to pooling, over the feature map H_i with a stride s. The operation is given by the equation

$$\underbrace{H_i[b, x, y, c_{out}]}_{\text{Current Layer}} = g\left(\underbrace{A_i[\Delta X, \Delta Y, C_{in}, c_{out}]}_{\text{Weights}} \underbrace{H_{i-1}[b, s \cdot x + \Delta X, s \cdot y + \Delta Y, C_{in}]}_{\text{Prev. Layer}} + \underbrace{b_i[c_{out}]}_{\text{Bias}}\right)$$

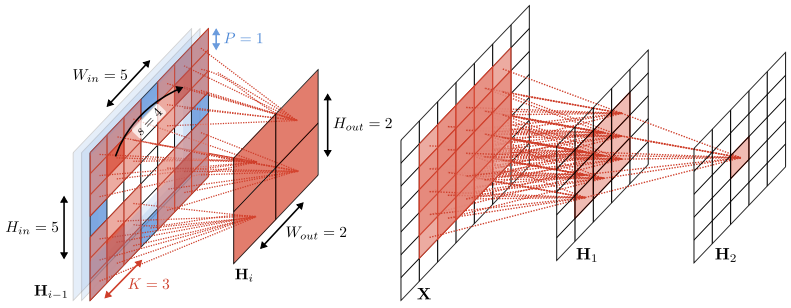
7.3.3 Receptive field and arithmetics

A neurons' receptive field is defined as all pixels in the input X that influence it, as shown in Figure 116b. Given an input image by width and height $W_{in} \times H_{in}$ padded by P zeros and a kernel $K \times K$ moving with a

stride of s, we can calculate the output tensors spatial dimensions with

$$\underbrace{\left(\left\lfloor \frac{W_{in}+2P-K}{s} \right\rfloor + 1\right)}_{W_{out}} \times \underbrace{\left(\left\lfloor \frac{H_{in}+2P-K}{s} \right\rfloor + 1\right)}_{H_{out}}$$

The equation to calculate W_{out} is similar to H_out. In the case of W_{out}, we apply twice the padding (both left and right). The kernel length is subtracted because the total width minus the kernel length is the maximal index the kernel can be positioned. The term $W_{in} + 2P - K$ is divided by the step size and rounded to the bottom. An example with parameters set can be seen in Figure 116a.



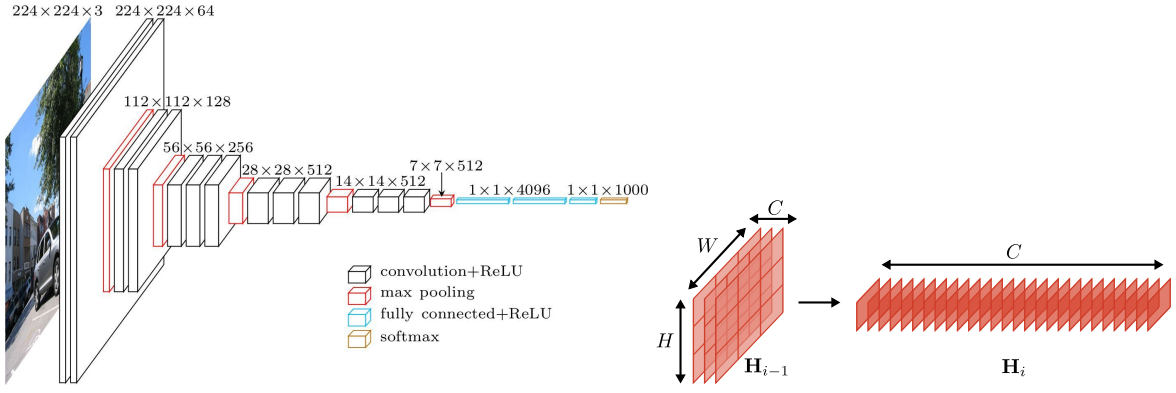
(a) (b)

Figure 116: (a) Strided convolution for an 5×5 image, a 3×3 kernel, 1 pixel zero padding and a stride of 2 resulting in a 2×2 output image. (b) The receptive field for the feature map H_2

7.3.4 Fully Connected Layers

In many CNN architectures, there exist fully connected (FC) layers. However, the most memory intensive part for, e.g., a VGG architecture are those FC layers. In order to apply a FC layer after a CNN, the spatial dimensions have to shrink. An example would be shrinking from 7×7 to 1×1 in Figure 117a. The tensor is reshaped, so that channel dimensions C absorb the width and the height dimensions, see Figure 117b. We can express a FC layer as

$$\underbrace{H_i[b, c_{out}]}_{\text{Current Layer}} = g\left(\underbrace{A_i[c_{out}, C_{in}]}_{\text{Weights}} \underbrace{H_{i-1}[b, C_{in}]}_{\text{Prev. Layer}} + \underbrace{b_i[c_{out}]}_{\text{Bias}}\right)$$



(a) (b)

Figure 117: (a) VGG16 (b) The reshape operation, transforming $H_{i-1}[B, X, Y, C]$ into $H_i[B, C]$

7.4 Upsampling

If pixel-level outputs are desired, it is necessary to upsample the features again. Downsampling is still required to provide good features with large receptive fields in the intermediate layers. Upsampling yields outputs at the same resolution as the input.

7.4.1 Nearest neighbor

Each channel is scaled using nearest-neighbor interpolation, as shown in Figure 118a.

7.4.2 Bilinear

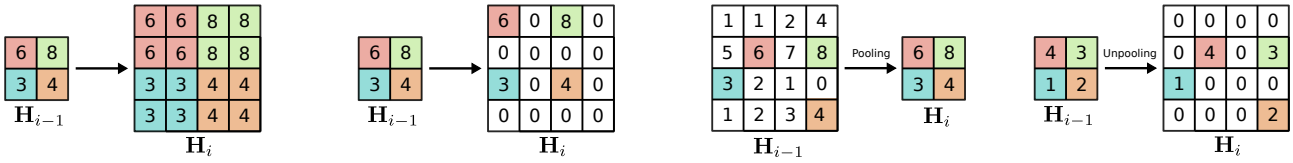
Each channel is scaled using bilinear neighbor interpolation.

7.4.3 Bed of Nails

Each element is inserted at a sparse location, and the in-between values are set to zero as shown in Figure 118b. The next layer often applies a convolution to fill up the values. Bed of nails sometimes produces "nails" artifacts.

7.4.4 Max-Unpooling

For unpooling, the indices of the maximum elements in the earlier corresponding pooling operation are stored. When applying the unpooling operation, the values are set at the locations from the previous pooling operation. As a result, corresponding pairs of downsampling and upsampling layers are required. An example can be seen in Figure 118c. The max-unpooling approach has been used in SegNet.



(a) (b) (c)

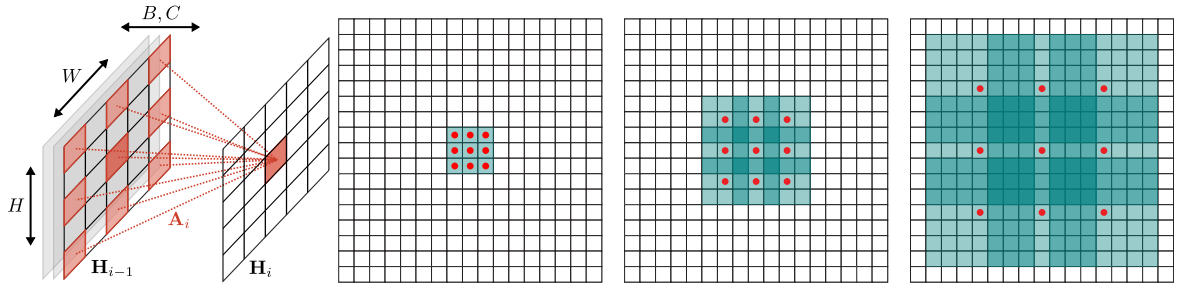
Figure 118: (a) nearest neighbor upsampling, (b) example bed of nails upsampling shows, (c) max-unpooling operation with the corresponding downsampling counterpart.

7.4.5 Dilated Convolution

The goal of dilated convolution is to increase the receptive field without increasing parameters or reducing spatial dimensions. Consequently, no pooling is required to increase the receptive field, and predictions can be produced at the same resolution as the inputs. The dilation factor d leads to a more "distributed" sampling of the input. A layer that implements dilated convolution is expressed with the formula

$$\underbrace{H_i[b, x, y, c_{out}]}_{\text{Current Layer}} = g\left(\underbrace{A_i[\Delta X, \Delta Y, C_{in}, c_{out}]}_{\text{Weights}} \underbrace{H_{i-1}[b, x + d \cdot \Delta X, y + d \cdot \Delta Y, C_{in}]}_{\text{Prev. Layer}} + \underbrace{b_i[c_{out}]}_{\text{Bias}}\right)$$

Dilated convolutions are used in, e.g., semantic segmentation, depth, optical flow.



(a) (b)

Figure 119: (a) Dilated convolution applied with a dilation factor $d = 2$. (b) The receptive fields exponential expansion without loss of resolution or coverage.

7.5 Architectures

In the following, we look at standard CNN architectures and paradigms they introduced.

7.5.1 LeNet-5

In 1998 the LeNet-5 was introduced using two convolution layers (5×5), two pooling layers (2×2) followed by two fully connected layers. In 1998 LeNet-5 achieved state-of-the-art accuracy on MNIST (before ImageNet). The architecture is shown in figure 120a.

7.5.2 AlexNet

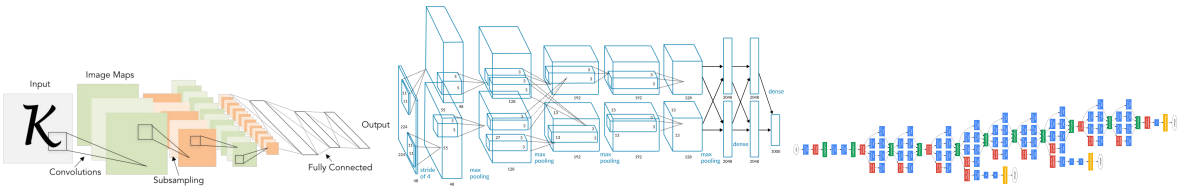
The deep learning revolution started in 2012 with the introduction of the AlexNet architecture. AlexNet consists of eight successive layers. It used ReLUs, dropout, data augmentation and was trained on 2 GTX 580 GPUs. The number of feature channels increases with depth, resulting in decreasing spatial resolution. AlexNet showed that CNNs work well in practice. The architecture is displayed in figure 120b.

7.5.3 VGG

In 2015 the VGG architecture was introduced. The novelty was using 3×3 convolutions everywhere but maintaining the same expressiveness with fewer parameters. The paper backed the paradigm that using small-sized kernels is better. A second variant has 19 instead of only 16 layers. The architecture is shown in figure 117a.

7.5.4 Inception

In 2015 the inception architecture was introduced. Inception consists of 22 layers. The modules utilize conv/pool operations with varying filter size. Backpropagating an error through 22 layers turns out to be very hard. To mitigate a vanishing gradient, multiple intermediate classification heads improve gradient flow. The use of 1×1 convolutions reduces the number of features and leads to higher efficiency.



(a) $y = x$ (b) $y = x$ (c) $y = x$

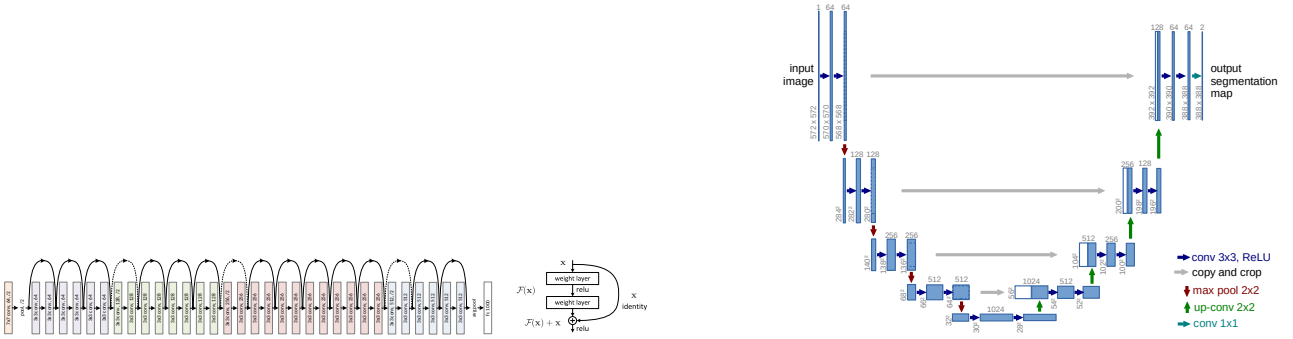
Figure 120: (a) LeNet-5 (b) AlexNet (c) Inception Network.

7.5.5 ResNet

ResNet was introduced in 2016. Residual connections allow for training deeper networks (up to 152 layers). The network uses a very simple and regular structure with 3×3 convolutions and strided convolutions for downsampling. ResNet and ResNet-like architectures are dominant today in computer vision.

7.5.6 U-Net

U-Net was introduced in 2015 and produced image-level segmentation by combining max-pooling as downscaling and up-convolution as upscaling operation. U-Net is the defacto standard for many tasks with image output (e.g., depth, segmentation).



(a) (b)

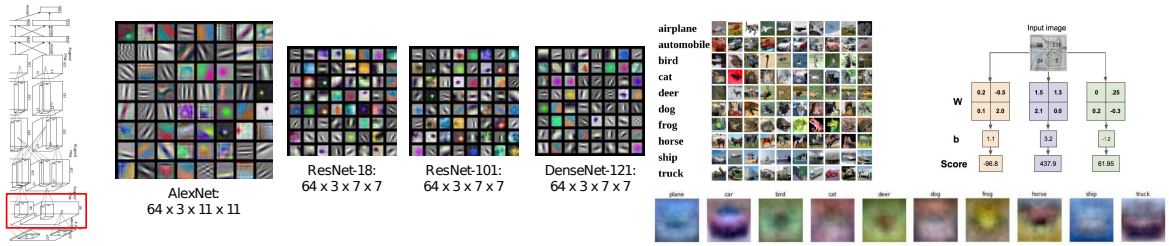
Figure 121: (a) ResNet (b) U-Net

7.6 Visualization

Visualization provides researchers with methods to open the black box of neural network classifiers. In the following section, a few visualization techniques are presented.

7.6.1 First Layers

For the first layers of a CNN, the convolutional filters can be directly displayed as shown in figure 122a. The first layers learn Gabor filters or simple gradients. Gabor filters for example extract spatial orientations. The interpretation of deeper layers is more difficult.



(a) (b)

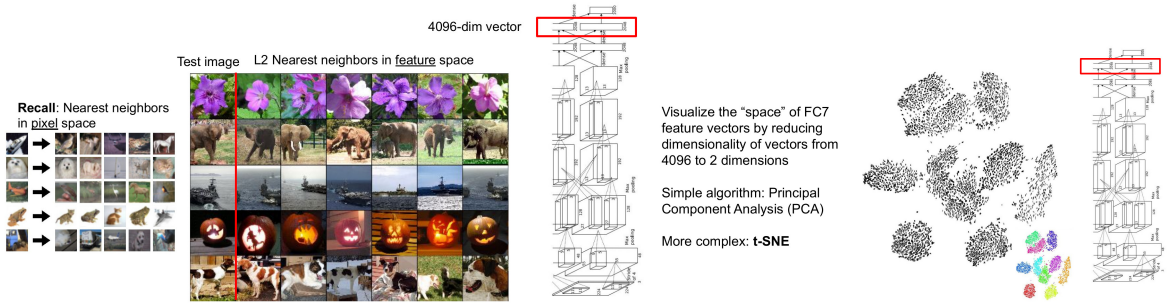
Figure 122: The right image shows weight matrix visualizations of a linear classifier. The left image shows the first layers learned filters of different deep learning architectures.

7.6.2 Last Layer

Nearest Neighbors
The last layer before the classification layer encodes an image representation for each image class. An image feature space is populated by recording the previous layers activation for all validation images, For one image, the k-nearest neighbors can be determined and displayed, showing semantic relationships between the visual embeddings. An example is displayed in figure 123a.

Dimensionality Reduction
Using dimensionality reduction techniques such as t-SNE or PCA, similar items can be identified or grouped. Figure 123b shows the FC7 architectures visual embedding of the MNIST dataset where similar numbers cluster in the same regions.

Maximally Activating Patches
Many images are passed through a network. The patches which maximally activate a specific neuron of interest are recorded. This way, we can determine what a neuron attends to.



(a) (b)

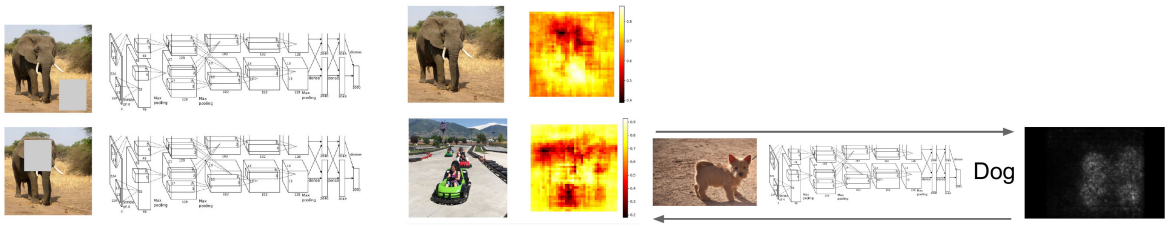
Figure 123: (a) Test images the k-nearest neighbors utilizing the visual embedding of the layer preceding the fully connected classification layer. (b) A lower dimensional embedding of the high dimensional visual embedding for the MNIST dataset.

7.6.3 Saliency

Saliency determines the relevant parts of an image for a particular classification decision and can uncover unwanted biases in data or the model.

Saliency via masking
An image of a particular class is passed through the network multiple times. Each time a different image region is masked, and the classification response is recorded. A saliency map can be constructed from these responses. A pixel in the saliency map defines how much the classifier is sensitive to an occlusion at that pixel wrt. to the class. An example is shown in figure 124a.

Saliency via backpropagation
Saliency via backpropagation computes the gradient of the class score for image pixels. The saliency map is generated by displaying the magnitude of the gradients. This shows how changing pixels of the input affects class scores the most. Saliency via backpropagation is cheaper than saliency via masking and is used to uncover biases in the data or the model.



(a) (b)

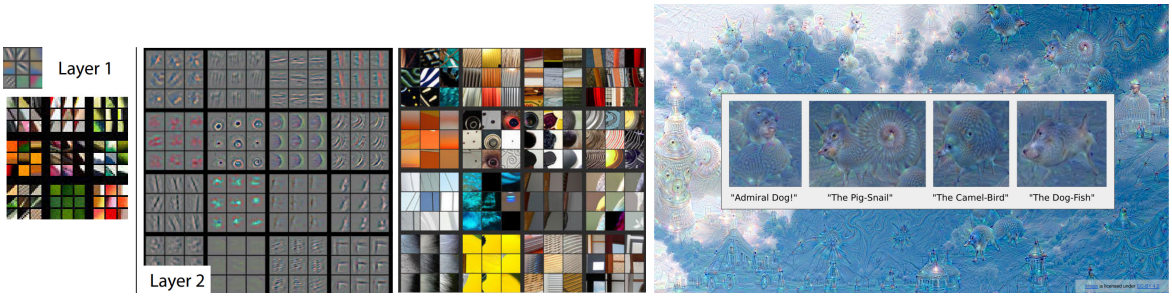
Figure 124: (a) Saliency via masking w.r.t to the classification of an elephant. The saliency map reveals that the elephant's head was most important for the classification decision. (b) A saliency map generated wrt. to gradients.

7.6.4 Deconvolution

The idea of deconvolution is to reconstruct the CNN in reverse order starting from one neuron. Figure 125a shows the deconvolution of two layers. The first layers display simple edge detectors, whereas the second layer shows contours or image parts up to the last layers that display certain objects.

7.6.5 Deep dream

Amplify activation of a chosen layer wrt. to a given image. The backpropagated gradient is set to the neuron's activation itself, leading to increased activation of neurons that are already activated. The results can be seen in figure 125b.



(a) (b)

Figure 125: (a) Deconvolution (b) Deep Dream.

8 Sequence Models

In real-life, our input and output might not always fit into a tensor of a fixed size. It might expand or shrink continuously across time. Speech, for example, might consist of a single word or a whole sentence. However, when we start speaking the model has no way of knowing whether our sentence might be 3 words or 30 words long. Sequence models have been developed to handle such input and output. The most popular type of sequencing models are recurrent networks which will be explained in section 8.1. Next, we will discuss different applications of such models to demonstrate their flexibility before moving on to discuss an improved version of the classical recurrent networks, i.e. gated recurrent networks. In the last part of this unit, auto-regressive models will be discussed.

A little remark on how computation graphs will be displayed in this section. For better clarity, the weight input nodes will be consider implicit for all hidden and output layer nodes. Furthermore, output nodes will be coloured in blue from now on.

8.1 Recurrent Networks (RNNs)

The main idea of RNNs is so-called **feedback connections** which feed information from one node back to itself. Thus, the **hidden state** h is updated based on the current input and the previous hidden state using the same parameters at each **time step**. This update is indicated by the arrow looping back to the compute node h in the central computation graph of Fig. 126. This procedure can be repeated t times. To keep track of how many recurrences already took place and to see the forward pass more clearly, one can "unroll" the RNN over time (cf. Fig. 126, left).
Although the input feature maps might change over time, the computations in $h_1, h_2, h_3, \dots$ always stay the same. Assuming the RNN is meant to translate from a language A into a language B, the inputs $x_1, x_2, x_3, \dots$ as characters or words of language A which are translated into the characters or words $\hat{y}_1, \hat{y}_2, \hat{y}_3, \dots$ of language B.
RNNs allow processing of **sequences of variable length** and have – at least in theory – an infinite memory since h is a function of all previous inputs, i.e., has long-term dependencies.

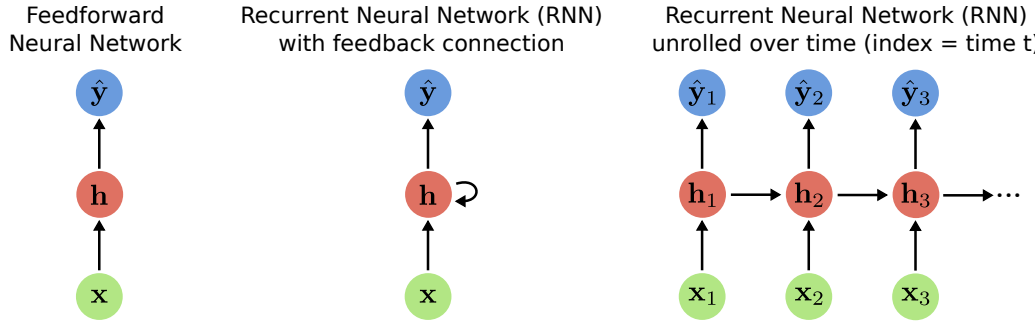


Figure 126: Feed-forward neural network vs. Recurrent neural network computation graphs.

In RNNs, computation nodes like h are called **cells**. RNN cells like the one in Fig. 127 receive input from the previous time-step $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ and output the tensor $\mathbf{h}_t$ and $\hat{\mathbf{y}}_t$. After concatenating the two input tensors, some transformations are applied to the resulting tensor to compute $\mathbf{h}_t$. However, the actual translation $\hat{\mathbf{y}}$ of our input needs one final affine transformation of the previously computed $\mathbf{h}_t$.

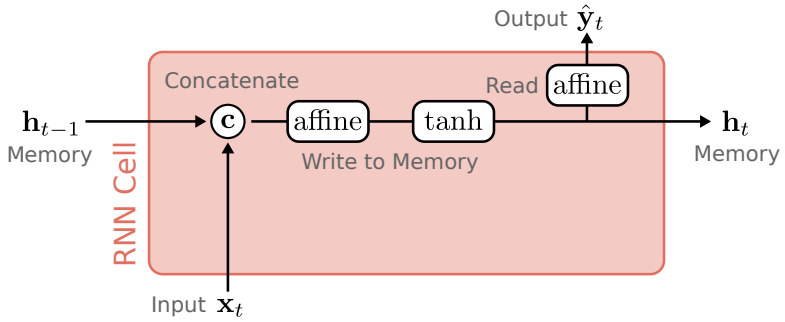


Figure 127: A basic RNN cell.

Mathematically, these steps are described by

$$\begin{aligned} \mathbf{h}_{t-1} &= f_h(\mathbf{h}_{t-1}, \mathbf{x}_t) \\ \hat{\mathbf{y}}_t &= f_y(\mathbf{h}_t) \end{aligned} \tag{26}$$

This general formula does not specify the form of the output mappings. Furthermore, **neither** f_h **nor** f_y **change over time**, unlike in layers of feed-forward networks.
The single layer RNN consisting of a cell like the one in Fig. 127, makes this more concrete:

$$\begin{aligned} \mathbf{h}_{t-1} &= \tanh(\mathbf{A}_h \mathbf{h}_{t-1} + \mathbf{A}_x \mathbf{x}_t + \mathbf{b}) \\ \hat{\mathbf{y}}_t &= \mathbf{A}_y \mathbf{h}_t \end{aligned} \tag{27}$$

This hidden state of this single layer (vanilla) RNN $\mathbf{h}_t$ is the linear combination of the input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. The output is the linear prediction based on the current hidden state $\mathbf{h}_t$. The $\tanh(\cdot)$ activation function used here, is the standard for most RNNs. It both takes and returns data in the range $[-1, 1]$ and comes with all the benefits (e.g., zero-centred) and disadvantage (e.g., saturation and vanishing gradient) (cf. section 4.2.2. on Tanh function). As mentioned before, the weight parameters $\mathbf{A}_h$, $\mathbf{A}_x$, $\mathbf{A}_y$, and $\mathbf{b}$ are constant over time, although the number of time steps might vary due to differences in the sequence length. Formulated in Einstein-Notation the RNN equation for a single layer RNN will look like this:

$$\begin{aligned} H_t[b, c_{out}] &= \tanh(A_h[c_{out}, C_{in}]H_{t-1}[b, C_{in}] + A_x[c_{out}, C_{in}]X_t + b[c_{out}]) \\ &= \tanh\left(A_h[c_{out}, C_{in}]\begin{bmatrix} H_{t-1}[b, C_{in}] \\ X_t[b, C_{in}] \end{bmatrix} + b[c_{out}]\right) \\ \hat{Y}_t[b, c_{out}] &= A_y[c_{out}, C_{in}]H_t[b, C_{in}] \end{aligned} \tag{28}$$

By stacking the input tensors are concatenated to $\begin{bmatrix} H_{t-1}[b, C_{in}] \\ X_t[b, C_{in}] \end{bmatrix}$. The concatenation operation of the two inputs is represented by the **c** node in Fig. 127.
Since, RNNs allow for processing of inputs and outputs of **variable length**, they can be categorised into four types based on their input-output mappings:

- **One to Many**, e.g., image captioning (image to sentence)
- **Many to one**, e.g., action recognition (video to action)
- **Many to Many**, e.g., machine translation (sentence to sentence)
- **Many to Many**, e.g., object tracking (video to object location per frame)

The for types are visualised in Fig. 128.
To allow the model to determine the output's length, we can predict a **stop symbol**. For languages, this could be a full stop at the end of a sentence.

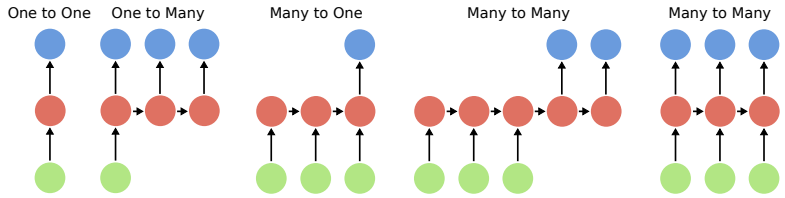


Figure 128: Types of input and output mappings.

Backpropagation. RNNs return an output at multiple times-steps with the number of time-steps not being fixed. At the same time, each output generates a loss. Thus, each of the losses at a time-step need to be taken into account when back-propagating, i.e. **gradients are back-propagate through time** (cf.).

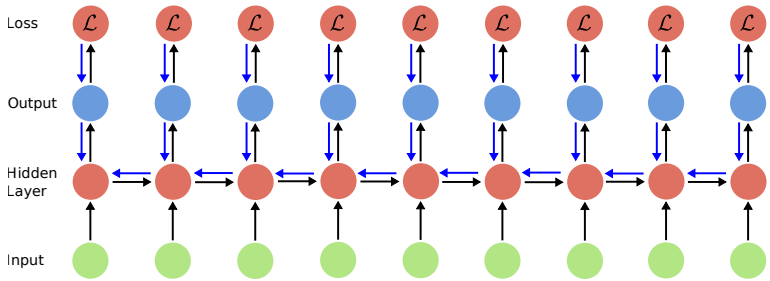


Figure 129: Backpropagation through time in an single layer RNN.

The gradient derived from each loss at a time-step is back-propagated to the previous time-step. There, it is added to the gradient of that time-step and so on. This process is repeated until the initial time-step is reached. Here, the weight parameters are updated. All of the hidden RNN cells share their parameters; hence, the gradients are accumulated. However, gradients are getting intractable very quickly – in terms of memory capacity – for more extensive sequences such as Wikipedia because all previously calculated time-steps had to be stored along the way.

A practical solution is **truncated backpropagation through time**. While hidden states continue to be carried forward forever, backpropagation is stopped before reaching the first time-step. In this way, only a limited number of time-steps is back-propagated. For example, given $t = 9$ time-steps, backpropagation could be split into three chunks, i.e., time-steps 1 to 3, 4 to 6, and 7 to 9. This truncation eliminates long-term dependencies as their gradients are no longer dependent on each other (cf.).

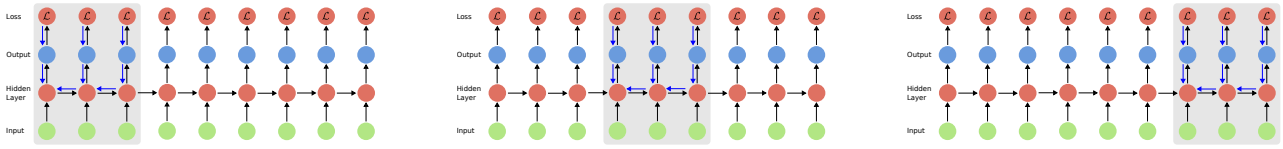


Figure 130: Truncated backpropagation through time in an single layer RNN.

Unfortunately, it will also cause the RNN to lose its memory. Thus, the truncation can not simply be set to a random value. Instead, it has to be chosen such that necessary context is preserved. In general, the truncation ranges between 50 - 60.[2]
Although it is not uncommon to find RNNs consisting of only a single layer, they can be built with multiple layers. RNNs are often kept very shallow in practice, i.e., only a couple of layers deep. Multi-layer RNNs are constructed by adding a second cell/hidden layer after the first one, like in typical feed-forward architectures. As both of the hidden layers in the RNN feedback to themselves, a computation graph would look like the diagram in Fig. 131 if unrolled.
Alternatively, the cell itself could be made deeper. In other words, instead of having a single affine transformation and activation function f_h inside a cell, you can stack multiple layers inside of it. Such an approach is often combined with residual connections in the vertical direction, i.e., between the layers.

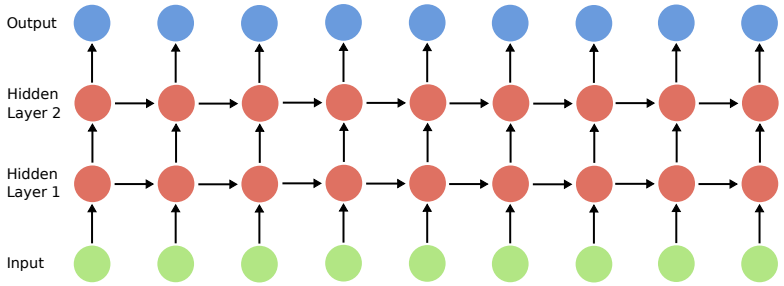


Figure 131: Schema of a multi-layer RNNs with 2 cells/layers.

8.2 Recurrent Network Applications

One task that can be addressed using RNNs is **multiple object recognition**. If you are given a house number and want to find the right house in a street, you need to recognize the correct sequence of digits when it appears. Given an image of a house number, an RNN would also have to recognize the sequence of numbers in the picture to make the correct classification. At each time-step, it would get a **glimpse** at an image region and predict a **saccade**. A saccade is the location where the model should shift its attention to in the next time-step. An example of how an RNN would operate on such a sequential task is presented in Fig. 132a).
RNNs can also be used for **recurrent instance segmentation**. First, an image gets transformed into a feature map by a fully-convolutional network. Then, an RNN **produces one segment of the image at each time-step**. To know which parts of an object have not yet been segmented, it relies on the segmented parts' union fed back into the model. In Fig. 132b) the colours indicate which part of an object was segmented at which time-step.
RNNs can also carry out **object tracking** of one or multiple objects in a video. Therefore, it updates each object's hidden state for each video frame (cf. Fig. 132c)). The updated information could consist of the location, the shape, the size, or the velocity of the object.
An interesting ability of RNNs is that they can solve tasks that are not sequential by nature. For instance, we can use them for **image generation**. An example is given in Fig. 133a) where a red rectangle indicates the attended region where the model can draw.
Instead of creating a single digit or sequences of digits, an RNN can also learn how to generate/complete images based on partially occluded inputs as demonstrated in Fig. 133Fig: 90 b). The occluded input is shown on

[2]However, it is task-dependent. In our language example, the truncation might differ, depending on whether the sequences are characters, words, or sentences.

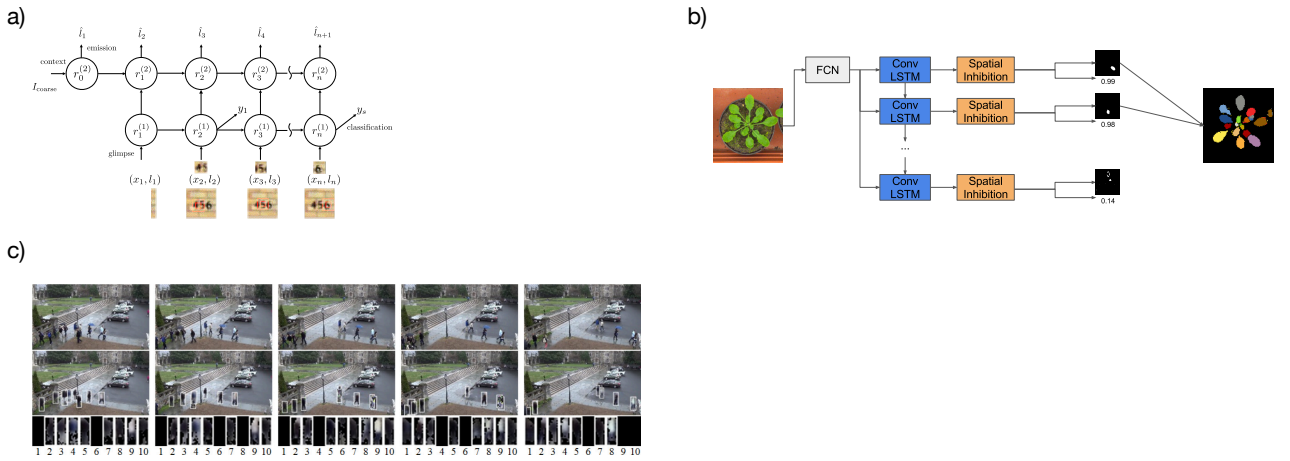


Figure 132: (a) Sequence recognition in multiple object recognition, (b) recurrent instance segmentation, (c) object tracking in video frames (right).

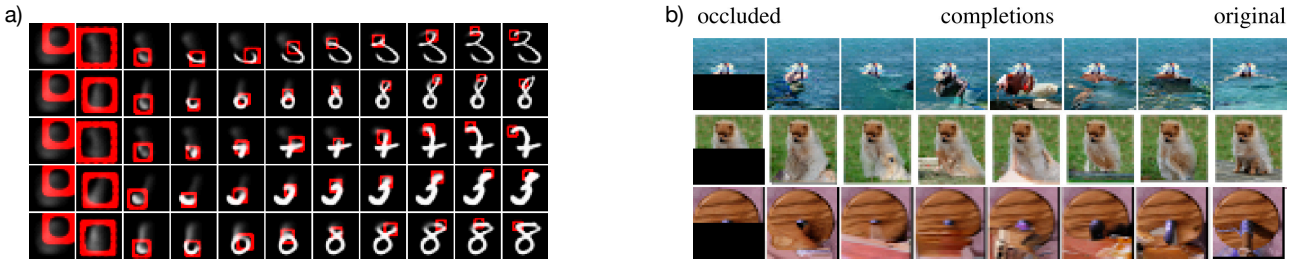


Figure 133: Examples of sequential image generation. a) Generation of handwritten digits; b) Generation of images based on partially occluded inputs.

the left and the original on the right. All pictures in between are completions generated by the model. This prediction is based on pixel intensities in the non-occluded part of the input. Some of the resulting images are more plausible than others.

The wide range of application possibilities also includes **image annotation**. The **Polygon-RNN** is an interactive object annotation tool that iteratively annotates the outline of a 2D object instance (e.g., a car) in an image with polygons (cf. Fig. 134a)). The annotator can manually correct the suggested outlines. Such tools are beneficial when creating an annotated data set to train an object detection model as they significantly reduce the amount of time spend on annotation.

Figure 134: Examples of object annotation (left) and road layout modeling (right)

Similar to the inference of object outlines in the image annotation, RNNs can model or generate **road layouts**. The resulting spatial graphs are based on an image input. An example of road layout generation and aerial road parsing can be seen in (cf. Fig. 134b)).

With the help of RNNs, we are even able to transform an image into a sentence that describes the picture content (cf. Fig. 135Fig: 92). For this task, an RNN is trained to sequentially generate words based on a condensed feature representation of the input image. A CNN previously computed this representation. In Fig. 135 the attention shifts from one region in the image to another one. The Figure also demonstrates how the attended regions correspond to the words that are generated.

Although this is quite impressive and it works (cf. Fig. 134c)), there is a lot that can go wrong (cf. Fig. 134d)). For instance, taking the second picture to the left in the first row of Fig. 134d) indicates that the clock-like shape identified by the model, is in fact, printing on the woman's hoodie. Thus, the errors made by such a

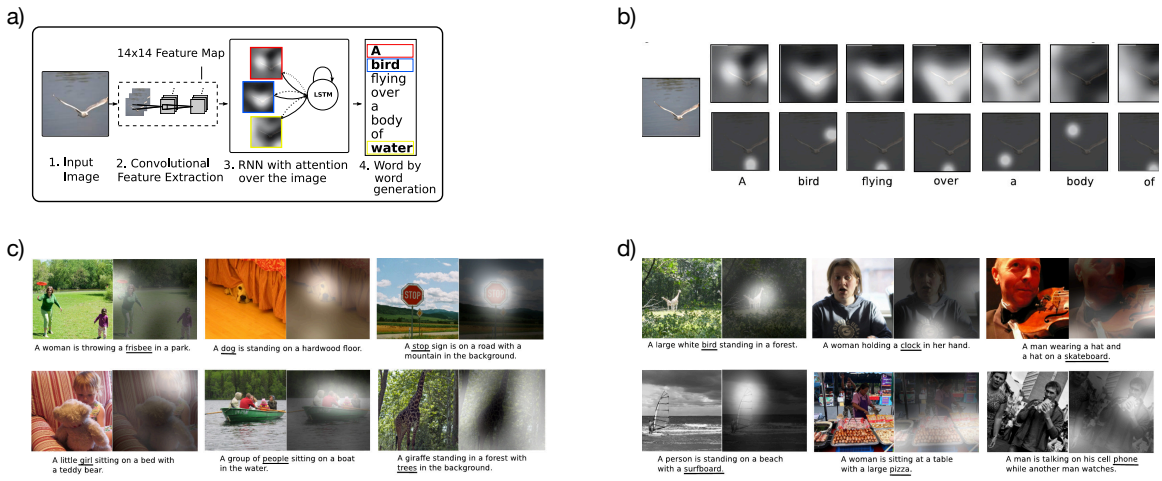


Figure 135: Examples of image captioning. a) general process flow; b) Shifted attention of the RNN (top: soft attention, bottom: hard attention); c) Successful image captioning examples; d) Unsuccessful image captioning examples.

model can be partially attributed to the regions with high attention. Analyzing these patterns of attention allow improvements in the model's performance.
A more recently proposed task for the usage of RNNs is **visual question answering**. The model receives an image and a question (and multiple possible answers) as input and predicts the correct answer. For instance, given the image of a Teddy Bear and the question "What kind of stuffed animal is shown?" the model should answer: "Teddy Bear".

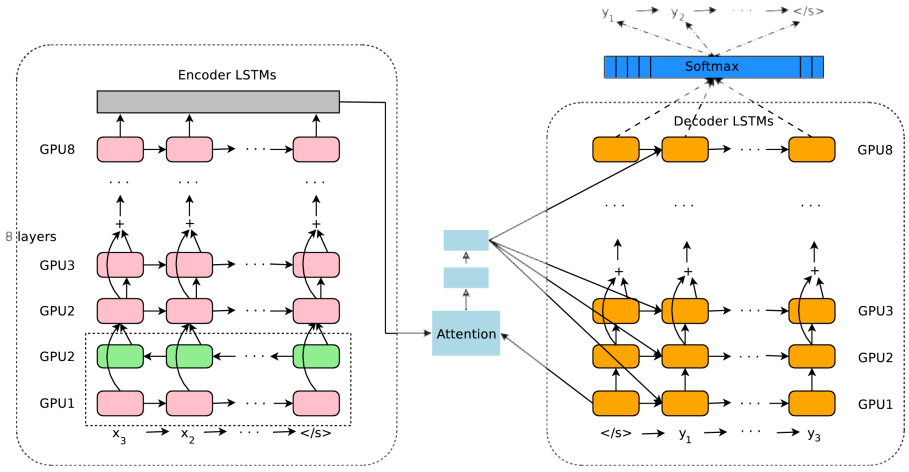


Figure 136: Google's neural machine translation system: An Encoder-Decoder model for machine translation.

One of the more common non-image related fields is **neural machine translation**. When translating from one language to another, many to many mapping is required as it is not possible to translate sentences word by word. For example, a German metaphor like "Ìch verstehe nur Bahnhof" would be translated to "I only understand train station." However, this is not what the model should return. A more appropriate translation would be something like "I don't understand what you just said.". For this reason, translation models use an encoder that builds a meaning representation of the original sentence and a decoder to construct a sentence with the corresponding meaning in the target language (cf. Fig. 136).
RNNs for language generation. In his blog post from 2015 Andrej Karpathy demonstrated the effectiveness of simple **character-level language models** to generate natural language character-by-character. To be able to generate the word "hello", an alphabet containing four characters: "h", "e", "l" and "o" is needed. Each of the character is represented by a **1 hot vector**, e.g., the letter "h" is represented by the vector $(1, 0, 0, 0))^T$. If fed with the "h"-vector, the model predicts a **distribution over the next character** via a Softmax function. The character drawn from the distribution is then in turn **fed as an input** to the RNN at the next time-step. However, this little toy example would have to be extended to work on more than just the simple word "hello". Thus, a three layer RNN with 512 hidden nodes was trained on William Shakespeare's works, which consisted of 4.4 million characters in total. Although the meaning and grammar of the produced sentences are debatable,

this simple model makes surprisingly few orthographic errors, as can be seen from Fig. 137a). It can also understand that there are some underlying structures in natural language such as spacing, punctuation, main and sub-clauses, and dialogue structure without being told that such things exist.
A closer look at the training process sheds light on what the model learns and at which stage it learns what. It starts by detecting that space separates words. At the next stage, the model starts to get the hang of some principles of sentence structuring. It also caught on to some words like “I”, “here”,“in”, “at”, “on”. Then, the model constantly improves and fine-tunes until it produces actual words and non-sense sentences.

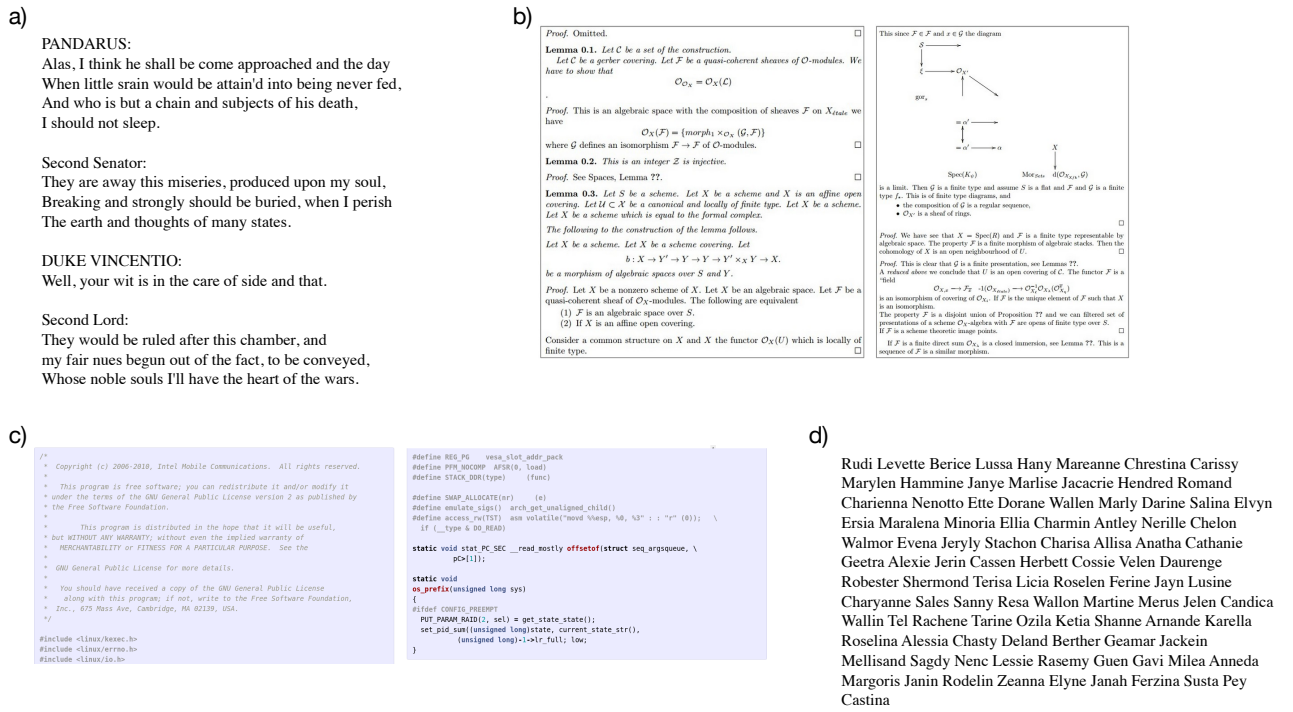


Figure 137: Examples of text generation with character-level language model. a) Output from model trained on Shakespeare text; b) Output from model trained on Latex source code file for mathematics book; c) Output from model trained on programming source code; d) Output from model trained on baby names

Moving to a more practical application of such a model, it is also possible to train a model to create output in latex format, source code, or baby names (Fig. 137b) - d)).
When looking at what the single neurons in such a model have learned, one can find neurons that detect a line’s end or the beginning and end of a quote. In the source code example, there are neurons trained to the structure if-statements. However, there are way more cells whose function is not so easily explained. Overall the behavior of only $\sim 5\%$ of the hidden neurons is logical and can be interpreted by a human.

8.3 Gated Recurrent Networks

RNNs can be quite difficult to train. To improve trainability, almost all modern RNNs use gates – hence the name **gated recurrent networks**. Most of the applications mentioned in the previous section use such gated RNNs.
But why is it so difficult to train a vanilla RNN? Assuming a 1D hidden state $h_t \in \mathbb{R}$ with $h_t = \tanh(a_h h_{t-1} + a_x x_t + b)$ and its gradient $\frac{\partial h_t}{\partial h_{t-1}} = \tanh' a_h$ with $\tanh' a_h = \frac{\partial \tanh(x)}{x}$ a longer chain of time-steps would result in

$$\frac{\partial h_t}{\partial h_{t-k}} = \frac{\partial h_t}{\partial h_{t-1}} + \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_{t-k+1}}{\partial h_{t-k}} = \left(\prod_{i=t-k+1}^{t} \tanh'_i \right) a_h^k \tag{29}$$

during backpropagation where k refers to the number of earlier time-steps. Thus, the gradient vanishes as soon as $\tanh(\cdot)$ saturates. Hence, the initialization of the network needs to be chosen carefully to avoid saturation.
Unfortunately, the gradient might still not behave as it should. Assuming the weights have been initialised properly, it is possible that the activation functions do not saturate at all. The hidden state $h_t = \tanh(a_h h_{t-1} + a_x x_t + b)$ would be $\approx a_h h_{t-1} + a_x x_t + b \in [-1, 1]$. If the gradient for the hidden state h_t is calculated, i.e. $\frac{\partial h_t}{\partial h_{t-1}}$, it would be approximately a_h. A backpropagation over k previous hidden states would then return a gradient

$$\frac{\partial h_t}{\partial h_{t-k}} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{t-k+1}}{\partial h_{t-k}} = a_h^k \tag{30}$$

where the weights are multiplied to the power of k previous time-steps. This becomes a problem if $a_h > 1$. In this case the gradients will **explode**, i.e. become very large and lead to divergence. Consider, for example, $a_h = 1.1$ and $k = 100$. The resulting gradient would be $\frac{\partial h_t}{\partial h_{t-k}} = a_h^k = 13781$.
Gradient clipping is often applied to prevent exploding gradients in RNNs. First, think about what happens in case $a_h < 1$. Given $a_h = 0.9$ and $k = 100$, the gradient will become really really small, $\frac{\partial h_t}{\partial h_{t-k}} = a_h^k = 0.0000266$. Such **vanishing** gradients will prevent the model from learning at earlier time-steps. This issue can not be solved by simply clipping the gradient. Instead, the architecture of the model needs to be changed. Unfortunately, introducing residual connections will not work as the parameters are shared across time. On top of that, the input and desired output at each time-step are different.
The same difficulties will arise when extending the RNN to vector- or tensor-valued

$$\begin{aligned} \mathbf{h}_t = \tanh(\mathbf{A}_h \mathbf{h}_{t-1} + \mathbf{A}_x \mathbf{x}_t + \mathbf{b}) \approx \mathbf{A}_h \mathbf{h}_{t-1} + \mathbf{A}_x \mathbf{x}_t + \mathbf{b} \\ \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \approx \mathbf{A}_h \Rightarrow \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-k}} \approx \mathbf{A}_h^k \end{aligned} \tag{31}$$

Let $\mathbf{A}_h = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$ be the eigendecomposition of the square matrix $\mathbf{A}_h$. Thus, the gradient is $\mathbf{A}_h^k = (\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1})^k = \mathbf{Q}\mathbf{\Lambda}^k\mathbf{Q}^{-1}$ with a diagonal eigenvalue matrix $\mathbf{\Lambda}$. Similar to the 1D hidden states before, the model will encounter exploding gradients for components with eigenvalues > 1 and vanishing gradient for components with eigenvalue < 1. Again the weight matrix $\mathbf{A}_h$ is shared across time.
Now, gradient clipping can be used to dampen the effects of exploding gradients. Gradient clipping refers to a simple heuristic which clips the gradient to a fixed value τ, before applying the gradient update during SGD. If the norm of the gradient vector, i.e. $||\texttt{A.grad}||_2$, is smaller then the threshold τ, nothing happens. Should the norm exceed τ, the gradient vector will be divided by its norm and the result will be multiplied with τ. In other word, first the norm is "reset" to 1 before it is set to τ, i.e. the maximum gradient value. Thus,

$$\texttt{A.grad} = \begin{cases} \texttt{A.grad} & \text{if } ||\texttt{A.grad}||_2 \leq \tau \\ \tau \frac{\texttt{A.grad}}{||\texttt{A.grad}||_2} & \text{otherwise} \end{cases} \tag{32}$$

The maximal gradient magnitude τ is a hyperparameter of the model and often ranges between $[1, 10]$.
Avoiding vanishing gradients is more complicated than that. Here, **gates** come into play. There are several types of gates. The first, most complex, and most influential one, the **Long- Short-Term Memory** (LSTM), was introduced back in 1997 by Hochreiter. There are two more recent and simpler types of gated RRNs, namely the **Gated Recurrent Unit** (GRU, by Cho 2014) and the **Update Gated Recurrent Neural Network** (UGRNN, by Collins 2017).
All three gate architectures use their gates to filter information. However, the number of gates and the way they work is different (σ in the cell architectures in Fig. 138).

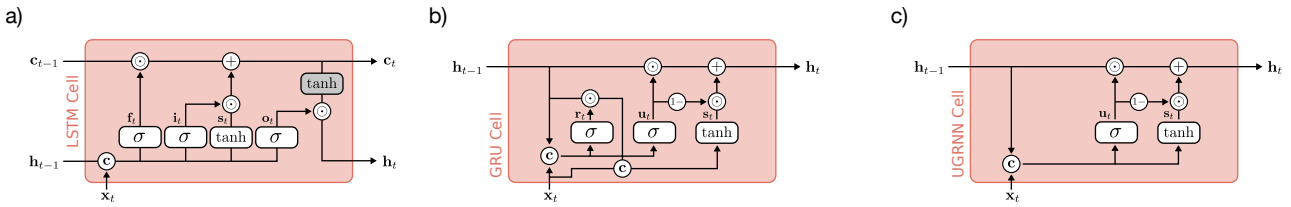


Figure 138: Three types of gated RNNs: a) LSTM, b) GRU, c) UGRNN.

8.3.1 Update Gate Recurrent Neural Network (UGRNN)

Starting with simplest gated architecture, the UGRNN, Fig. 138c) shows that it only uses a single gate to control the gradient flow. Like the vanilla RNN cell, a UGRNN cell receives input from the previous hidden state H_{t-1} and the current data input X_t. These two are again concatenated to a single matrix at c.
From there, the concatenated tensor is passed to two linear transformations. One is the **update gate** U_t which determines if the hidden state H is updated or not. The other is the next **target state** S_t that is added to H_{t-1} with element wise weights U_t.

At the first gate a Sigmoid function is computed. In addition to the concatenated input matrix, it uses two weight matrices A_{uh} and A_{xx} – the former is applied to the previous hidden state H_{t-1}, the later to data input X_t – and a bias b_u all of which are shared across time-steps. Thus,

$$U_t[b, c_{out}] = \sigma(A_{uh}[c_{out}, C_{in}]H_{t-1}[b, C_{in}] + A_{ux}[c_{out}, C_{in}]X_t[b, C_{in}] + b_u[c_{out}]) \tag{33}$$

If the output of the Sigmoid function is 1, then the information from the previous time-step is passed on and used to update the hidden state. If it is 0, the information accumulated in the previous time-step(s) is discarded. The concatenated input matrix is also passed into the tanh activation function at the target gate. Like the Sigmoid function, it also receives two weight matrices A_sh and A_sx for H_{t-1} and X_t respectively and a bias b_s. Hence,

$$S_t[b, c_{out}] = \tanh(A_{sh}[c_{out}, C_{in}]H_{t-1}[b, C_{in}] + A_{sx}[c_{out}, C_{in}]X_t[b, C_{in}] + b_s[c_{out}]) \tag{34}$$

One last calculation is necessary to arrive at the output of the current state H_t. Using the output of the update gate U_t and the target gate S_t, the following computation takes place

$$H_t[b, c_{out}] = U_t[b, C_{out}]H_{t-1}[b, C_{out}] + (1 - U_t[b, C_{out}])S_t[b, C_{out}] \tag{35}$$

The first term of the equation is an element-wise multiplication of the Sigmoid function's output and the previous hidden state. It controls how much information of the previous time step should be part of the current hidden state output. In places where the Sigmoid is 0, the previous state's information is erased and kept otherwise. The second term decides how much information from the current state should be passed into the output. By element-wise subtracting the Sigmoid function's output from 1 before element-wise multiplying the result with the target gate, the output information is kept where the memory is erased and deleted where the memory is preserved. In other words, U_t is a linear weighting between the previous state and the new target state.
In the schematic cell, this equation is indicated by the four operations in circles that follow after calculating the Sigmoid and Tanh function. The element-wise multiplication is indicated by $\odot$. This symbol denotes the **Hardamad product**. The Hardamad product means that the first element in matrix U_t is multiplied with the first element in matrix H_{t-1}, then the second element in matrix U_t with the second element in matrix H_{t-1} and so on. We can drop the Einstein notation of the Equation above as it does not have to be stated explicitly. However, one can also make it explicit by using standard vector notation and writing the function as

$$H_t[b, c_{out}] = U_t[b, C_{out}] \odot H_{t-1}[b, C_{out}] + (1 - U_t[b, C_{out}]) \odot S_t[b, C_{out}] \tag{36}$$

After understanding how a gate works, the question of what does one gain from them remains. Let's compare the backpropagation in a regular RNN with that of a gated UGRNN (cf. Fig. 139).

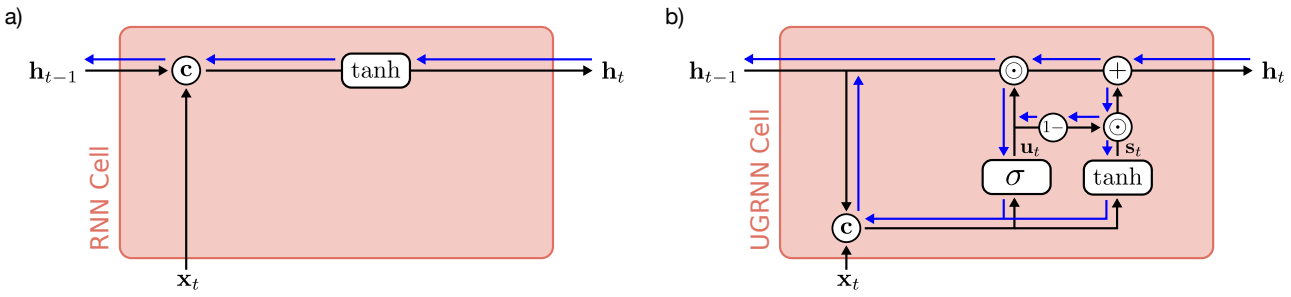


Figure 139: Backpropagation in a regular RNN cell (left) and a gated UGRNN cell (right).

Expressed mathematically, the backpropagation of the two RNN types are:

$$\begin{aligned}
& & \mathbf{u}_t &= \sigma(\mathbf{A}_{uh}\mathbf{h}_{t-1} + \mathbf{A}_{ux}\mathbf{x}_t + \mathbf{b}_u) \\
& & \mathbf{s}_t &= \sigma(\mathbf{A}_{sh}\mathbf{h}_{t-1} + \mathbf{A}_{sx}\mathbf{x}_t + \mathbf{b}_s) \\
\mathbf{h}_t &= \tanh(\mathbf{A}_h\mathbf{h}_{t-1} + \mathbf{A}_x\mathbf{x}_t + \mathbf{b}) & \mathbf{h}_t &= \mathbf{u}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{u}_t) \odot \mathbf{s}_t \\
\mathbf{h}'_t &= \tanh'\mathbf{A}_h \approx \mathbf{A}_h \text{ with } \mathbf{h}'_t = \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} & \mathbf{h}'_t &= \mathbf{u}'_t \odot \mathbf{h}_{t-1} + \mathbf{u}_t + (1 - \mathbf{u}'_t) \odot \mathbf{s}_t + \dots
\end{aligned} \tag{37}$$

with the backpropagation of a regular RNN on the left and a UGRNN on the left. As seen previously, when discussing vanishing and exploding gradients in RNNs, the gradients can be approximated by the weight matrix $\mathbf{A}_h$ which gets multiplied with itself according to the number of time-steps. In a gated RNN cell, as the UGRNN referred to here, this is no longer the case. Instead, it can maintain gradient flow despite small $\mathbf{A}_h$ by setting its gate to $u \approx 1$. Put differently; the cell can determine how much of the gradient is passed backward (similar to the forward pass). It can learn to ignore some of the information of a state vector for a very long time and thus pass it backward for a very long time, too. In the equation this can be seen from $\mathbf{u}_t$. No matter how small any of the other terms in $\mathbf{h}'_t$ get, we always multiply the backward propagated gradient with a value close to 1, i.e., the already computed gradient will just be passed through.
Thus, an UGRNN is able to **keep the state** of a variable **over a long time horizon** ($u \approx 1$). Going back to the character-level language models, such a model can, for instance, keep track of being inside a quote/if-statement/etc. or not.

8.3.2 Gated Recurrent Unit (GRU)

Turning to Gated Recurrent Unit (GRU) as the one in Fig. 138b), it becomes apparent that the GRU cell relies on two gates. Although the right part of the cell that computes the update gate and the target state has the same behavior as in the UGRNN cell, the GRU cell's input must pass through a so-called **reset get**. The reset gate decides which parts of the previous state are used to compute the current state.
Put into an equation, a GRU cell computes a new current state as follows:

$$\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{W}_{rx}\mathbf{x}_t + \mathbf{b}_r) \\
\mathbf{u}_t &= \sigma(\mathbf{W}_{uh}\mathbf{h}_{t-1} + \mathbf{W}_{ux}\mathbf{x}_t + \mathbf{b}_u) \\
\mathbf{s}_t &= \tanh(\mathbf{W}_{sh}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{W}_{sx}\mathbf{x}_t + \mathbf{b}_s) \\
\mathbf{h}_t &= (1 - \mathbf{u}_t) \odot \mathbf{h}_{t-1} + \mathbf{u}_t\mathbf{s}_t
\end{aligned} \tag{38}$$

8.3.3 Long Short-Term Memory (LSTM)

The Long Short-Term Memory makes the current hidden state's processing even more complicated by passing along an additional cell state $\mathbf{c}$ and using three gates. The first gate, the **forget gate**, determines which information is erased from the cell state. The second gate, the **input gate**, decides which values of a cell state should be updated. The third gate, the **output gate**, is in charge of picking the cell state elements that are revealed at a time t.
More formally, the operations in a LSTM cell can be formulated as

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{W}_{fx}\mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_{ih}\mathbf{h}_{t-1} + \mathbf{W}_{ix}\mathbf{x}_t + \mathbf{b}_i) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{oh}\mathbf{h}_{t-1} + \mathbf{W}_{ox}\mathbf{x}_t + \mathbf{b}_o) \\
\mathbf{s}_t &= \tanh(\mathbf{W}_{sh}\mathbf{h}_{t-1} + \mathbf{W}_{sx}\mathbf{x}_t + \mathbf{b}_s) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{s}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned} \tag{39}$$

8.3.4 UGRNN vs. GRU vs. LSTM

A comparison of the three types:

UGRNN	GRU	LSTM
• 1 gate	• 2 gates	• 3 gates
• expose entire state	• expose entire state	• control exposure
• single update gate	• single update gate	• input/forget gates
• few parameters	• medium parameter	• many parameters

In their systematic study, Collins et al. 2017 [3] find that GRUs are the most learnable of the three types of gated RNNs when it comes to shallow architectures. It is followed by the UGRNN. This finding supports the observations that the simpler and more recently developed gated RNNs perform as well as (or even better) as the more complex ones. Thus, GRU and UGRNN can be thought of as improved and simplified versions of the original LSTM cell.

8.4 Auto-regressive Models

RNNs have long been believed to be the only model capable of processing sequential input. Recently, a class of feed-forward neural networks has be found that are at least on par with RNNs. Such models are called **Auto-regressive Models**. A k'th order auto-regressive model predicts the next variable $\mathbf{x}_t$ in a time series based on the k previous variables $\mathbf{x}_{t-1}, \mathbf{x}_{t-2}, \cdot$. In the example in Fig. 140, one can see that $\mathbf{x}_3$, $\mathbf{x}_4$ and $\mathbf{x}_5$ are connected to the previous two inputs, i.e. $\mathbf{x}_5$ on $\mathbf{x}_3$ and $\mathbf{x}_4$, $\mathbf{x}_4$ on $\mathbf{x}_2$ and $\mathbf{x}_1$, and $\mathbf{x}_3$ on $\mathbf{x}_2$ and $\mathbf{x}_1$. Thus, k would be 2.

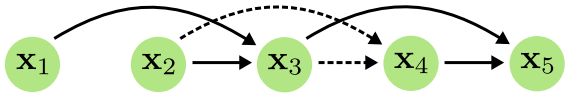


Figure 140: Example of input dependencies of a a 2^{nd} order autoregressive model

Like RNNs, **parameters are shared** across time, i.e. same function $f(\cdot)$ at each t. What makes them different from RNNs is that they make a strong **conditional independence** assumption, e.g. $\mathbf{x}_5$ is independent of $\mathbf{x}_2$ given $\mathbf{x}_3$ and $\mathbf{x}_4$.
The simple concept of autoregressive model can be extended to situations with **varying or different input and output lengths**. For instance, as illustrated in this single-layer model in Fig. 141. Here an output $\hat{\mathbf{y}}_t$ is dependent on the k-last inputs, i.e. $\{\mathbf{x}_i | t-k \leq i \leq t\}$, and independent of everything that came before, i.e. $\{\mathbf{x}_i | i < t-k\}$. Thus, $\hat{\mathbf{y}}_t = f(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k})$.

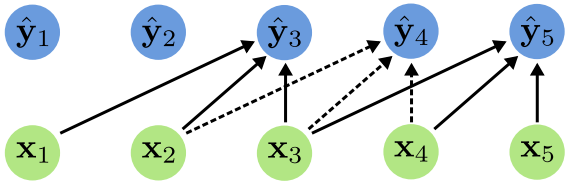


Figure 141: Example of a single-layer auto-regressive model

From this most basic form of an auto-regressive model, it becomes evident that the past information is not summarised in a hidden state $\mathbf{h}$. Thus, such a model does not have infinite memory. However, it is **easier to train** because there is no need for backpropagation through time.
Like all other neural networks, auto-regressive models can be extended to deep models with **multiple layers** by adding some hidden layers between the input and output layers (cf. Fig. 142). Thus,

$$\begin{aligned} \mathbf{h}_t &= f_1(\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k}) \\ \hat{\mathbf{y}}_t &= f_2(\mathbf{h}_t, \mathbf{h}_{t-1}, \ldots, \mathbf{h}_{t-k}) \end{aligned} \tag{40}$$

where each hidden layer h_t takes into account k previous inputs $\mathbf{x}_t, \mathbf{x}_{t-1}, \ldots, \mathbf{x}_{t-k}$ and each output at the current time $\hat{\mathbf{y}}_t$ k previous $\mathbf{h}_t, \mathbf{h}_{t-1}, \ldots, \mathbf{h}_{t-k}$ hidden layer outputs.
Such a model would effectively perform multiple causal **temporal convolutions**. The weights are shared across time, which means we are swiping a kernel filter over the sequential inputs to arrive at the outputs. These convolutions are causal because they depend only on past and present inputs but not on future inputs.[3]
Since autoregressive models are similar to any other deep neural network (e.g. ConvNets), residual connections and dilated convolutions can be integrated into its architecture.
The first successful model of that type is **WaveNet** by Oord et al. 2016 [14] (cf. Fig. 143 a)). WaveNet is a generative model for **raw audio wave-forms** which generates realistic speech. It consists of four hidden layers.

[3]Theoretically, it is possible in both autoregressive models and RNNS to also include future information into the computation process. Those models would then be **bi-directional**.

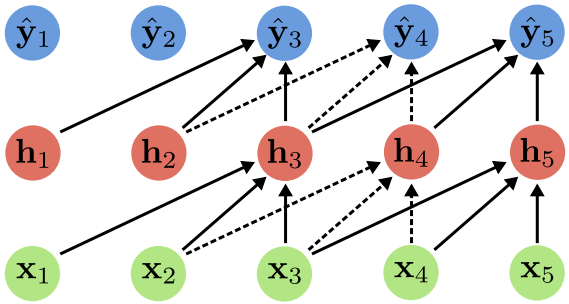


Figure 142: Example of a multi-layer auto-regressive model

At each hidden layer, we consider a larger amount of past input information – similar to a growing receptive field in ConvNets – due to the usage of **dilated convolutions**. The model architecture also contains **residual and skip connections**.
The success of WaveNet was surprising as the generation of audio signals is considered hard as it needs 16k samples per second and structure at **multiple time scales**. RNNs usually avoid raw audio wave-forms for this reason. However, WaveNet was able to do this and outperform traditional LSTMs on speech synthesis, as demonstrated for North American English and Mandarin Chinese by the authors.

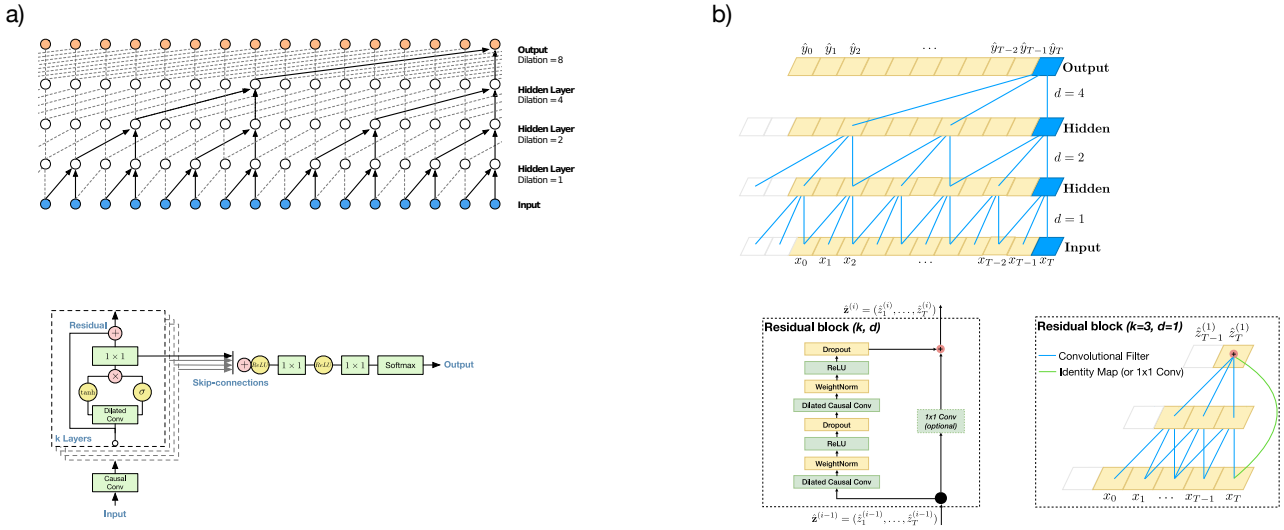


Figure 143: Example of auto-regressive models. a) WaveNet; b) TCNs.

This idea has been developed into a simpler type of autoregressive models called **Temporal Convolution Networks** (TCNs) by Bai et al. 2018 [1] (cf. Fig. 143 b)). These networks use **zero-padding** to handle sequences of arbitrary length. Aside from the **residual layers**, the rest of the **deep multi-layer** architecture is kept simple. Like the WaveNet, it also uses dilated convolutions to increase the receptive field size, also called "context". TCNs have been shown to perform better than LSTMs, GRUs, and RNNs on **several different sequence tasks**.
How come that these simpler models that do not have a memory like an RNN, are working so well? This blogpost concludes that – at least in theory – a stable RNN can be approximated by a feed-forward network for both inference as well as training. Based on current literature, it states that the strength of RNNs, namely their "infinite memory", has no chance to unfold its power in practice. Furthermore, the unlimited context offered by RNNs is not strictly necessary in some tasks (e.g., language modeling). Models that truncate at a sequence length of 13 or 25 are often competitive with infinite memory models.

9 Natural Language Processing

9.1 Language Models

A **language model** models the probability distribution over a sequence of **discrete tokens** $\mathbf{x} = (x_1, \ldots, x_T)$. Each of these tokens can take a value from a **vocabulary** $\mathcal{V}$ ($x_t \in \mathcal{V}$) and it can be for example a word, a character or a byte, depending on the model. The last token of such a sequence is a special <EOS> token to

indicate the end of a sentence. Therefor: $x_T = \texttt{<EOS>}$. This means that whenever we predict the next word or token, we can either predict a word form the vocabulary or we can predict the end of sentence token.

$$\begin{aligned} p(\mathbf{x}) = p(x_1, \ldots, x_T) &= \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1}) \\ &= p(x_1)\, p(x_2|x_1)\, p(x_3|x_1, x_2) \ldots \end{aligned}$$

We can model a probability distribution over a sequence $\mathbf{x}$ of discrete tokens $(x_1, \ldots, x_T)$. The joint distribution over this sequence decomposes into a product of t conditional distributions p of x_t given all of the x with an index smaller than x_t which is a consequence of the **product rule** or **chain rule of probability**. This is not an approximation. We simply can rewrite or decompose any joint distribution in such a form. By decomposing it that way each token in a sequence only depends on the previous tokens in that sequence by iteratively applying the product rule.
We have already seen in the lecture about recurrent networks and feed forward models that there are models that operate on the character level but there's also language models that operate on the word level as we will see in this lecture. The difficulty of operating on the word level is that the vocabularies are much larger. But it is also easier to operate on the word level since it is easier to model long-term dependencies while at the character level long-term dependencies are hard. At character level vocabularies are typically small, like 10, 20 or 30 characters, while at the word level typical vocabularies are of size 10.000, 20.000 or 30.000, depending on the language. So therefor we have to predict distributions over a very large state space.

Word Language Model Example:

$$\begin{aligned} p(\text{The dog ran away } \texttt{<EOS>}) = {} & p(\text{The})\, p(\text{dog}|\text{The})\, p(\text{ran}|\text{The dog}) \\ & p(\text{away}|\text{The dog ran})\, p(\texttt{<EOS>}|\text{The dog ran away}) \end{aligned}$$

As we can see in the example above, the sentence 'The dog ran away <EOS>' decomposes based on the product rule into the conditional distributions p of 'The', times p of 'dog' given 'The', times p of 'ran' given 'The' and 'dog' and so forth. We can see that language models in general but also in particular the given word language model are **autoregressive** models that predict the next token given all the previous tokens in this sentence or sequence. If a model is good then it has a high probability of predicting likely next words.

9.1.1 Applications

Language Recognition:
One of the most straightforward applications of language models is language recognition. Assume we have trained two language models p and p' which assign probabilities to sentences.

$$p(\mathbf{x}) = p(x_1, \ldots, x_T) \qquad p'(\mathbf{x}) = p'(x_1, \ldots, x_T)$$

Since they are different models they might assign a different probability to the same sentence. Lets assume that p has been trained on a large text corpora of English sentences and p' on French sentences.
We can then determine which sentence a language is from by simply classifying according to:

$$\text{Language}(\mathbf{x}) = \begin{cases} \text{English} & \text{if } p(\mathbf{x}) > p'(\mathbf{x}) \\ \text{French} & \text{otherwise} \end{cases}$$

Generative Model:
We can also use a language model to sample new sequences. Assume a language model over sentences $\mathbf{x}$:

$$p(\mathbf{x}) = p(x_1, \ldots, x_T)$$

And $p(\mathbf{x})$ has already been trained. By using the **decomposition into conditional distributions**

$$p(x_1, \ldots, x_T) = p(x_1)\, p(x_2|x_1)\, p(x_3|x_1, x_2) \ldots$$

we can efficiently **sample new sentences** from the model distribution. Sow e start by sampling x_1, the first word, and then we sample the second word conditioned on the first word, then we sample the third world conditioned on the first two words and so forth. By doing this we can sample from this model in linear time which is a sampling of words based on all the previous words.

Bayesian Inference: (Machine Translation)
We can also use these language models to do bayesian inference and in this case we have an example for machine

translation. Assume we have a language model trained p of $\mathbf{x}$ and lets consider this as a **prior** over possible sentences $\mathbf{x}$ in the form of a language model:

$$p(\mathbf{x}) = p(x_1, \dots, x_T)$$

Now assume a **likelihood** or proposal mechanism that tells us how likely sentence $\mathbf{x}$ translates to sentence $\mathbf{y}$:

$$p(\mathbf{y}|\mathbf{x}) = p(y_1, \dots, y_{T'}|x_1, \dots, x_T)$$

Which tells us according to this model how likely this translation is. This is how machine translation have worked in the past. These were rule based and very complicated systems that propose sentences and assign likelihoods to the translated sentences given the input sentence. It therefor assigns a probability to $\mathbf{y}$ given $\mathbf{x}$. The original sentence $\mathbf{x}$ and the translated sentence $\mathbf{y}$ might not have the same length. This is indicated by the last indice of $\mathbf{x}$ being T and for $\mathbf{y}$ being T'. If we have $p(\mathbf{x})$ and $p(\mathbf{y}|\mathbf{x})$ we can use Bayes rule to infer the **posterior** over translated sentences:

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})\, p(\mathbf{x})}{p(\mathbf{y})}$$

We multiply these two, $p(\mathbf{x})$ and $p(\mathbf{y}|\mathbf{x})$, together to get the posterior probability of $\mathbf{x}$ in the target language given the sentence in the source language.
Modern machine translations don't work like this. In this example we are using a generative model to infer a discriminative decision rule. In modern machine translation systems the probability of $\mathbf{x}$ given $\mathbf{y}$ is directly modeled by conditioning on the source sentence and modeling a distribution for the output sentence. We will be looking at some examples of these type of models in the last units.

9.1.2 Training

So far we have considered just generic distributions but now to make it more precise we use $p_{model}(\mathbf{x}|\mathbf{w})$ to distinguish the model from the data distribution and to indicate that the model has some parameter $\mathbf{w}$ that we want to train. So now let $\mathcal{X} = \{\mathbf{x}_i\}_{i=1}^N$ denote a training set with sentences $\mathbf{x}_i = \{x_1^i, \dots, x_{T_i}^i\}$.
We train the unconditional language model $p_{model}(\mathbf{x}_i|\mathbf{w})$ via **maximum likelihood:**

$$\begin{aligned}
\hat{\mathbf{w}}_{ML} &= \underset{\mathbf{w}}{\text{argmax}} \prod_{i=1}^{N} p_{model}(\mathbf{x}_i|\mathbf{w}) \\
&= \underset{\mathbf{w}}{\text{argmax}} \sum_{i=1}^{N} \log p_{model}(\mathbf{x}_i|\mathbf{w}) \\
&= \underset{\mathbf{w}}{\text{argmin}} \; -\mathbb{E}_{p_{data}} \left[\log p_{model}(\mathbf{x}|\mathbf{w})\right]
\end{aligned}$$

This is a standard formulation that we have seen before where the maximum likelihood prediction is the argmax over the parameters of the product of the model distribution for the entire dataset, where N is the size of the dataset. By applying the logarithm, which is a monotonic function, we can turn the product into a sum. This on the other hand corresponds to a minimization problem of the negative expectation of p_{data} of the logarithm of p_{model} $\mathbf{x}$ given $\mathbf{w}$. So we **minimize the cross entropy** between the data and the model distribution. If we manage to minimize this then the model is as similar as possible to the data distribution.

9.1.3 Evaluation

We want to be able to measure the performance of such a trained language model. **Character language models** typically measure performance in bits per character.

In order to do that we first need to introduce some basic quantities from information theory:
Shannon Information:
Given a character sequence $\mathbf{x}$ of length T with probability $p(\mathbf{x})$, the so called **surprise**, or shannon information, normalized by the sequence length T is the normalized negative log-likelihood of $\mathbf{x}$:

$$I(\mathbf{x}) = -\frac{1}{T} \log_2 p(\mathbf{x}) = -\frac{1}{T} \sum_{t=1}^{T} \log_2 p(x_t|x_1, \dots, x_{t-1}) \quad [bits]$$

The $-\frac{1}{T}$ is a normalizing constant to normalize with respect to the sequence length and thereby getting independent of the sequence length. Since $p(\mathbf{x})$ can be decomposed into this sequence of conditionals we then have

the sum of t from 1 to t over this decomposition. The unit of this is bits since we are using the basis of 2 for the logarithm.
Intuitively if we consider a distribution over $\mathbf{x}$. If we observe an $\mathbf{x}$ and the probability for that $\mathbf{x}$ is low under that distribution then the negative logarithm of that is large, therefor the surprise is large. If we are observing something that is unlikely then the information or the surprise is large. Conversely if we observe something that has a high probability of occurring then the surprise is low. We are not surprised to see that because it is likely to occur under our model.

Cross Entropy:
The **expected surprise** of the model under the data distribution (for sequences of length T) is thus given by the (normalized) cross entropy:

$$H(p_{data}, p_{model}) = \mathbb{E}_{p_{data}}\left[-\frac{1}{T}\log_2 p_{model}(\mathbf{x})\right]$$

Here we took the expression from the shannon information and took the expectation over the data distribution, we average over the entire dataset. In other words this is the expected surprise given our trained model p_{model}. A model is better if the expected surprise is low. So if the model models the data distribution well then the expected surprise is low and better models have a smaller number of bits.

Let us now consider sequences of arbitrary length by taking the limit $T \to \infty$.
By the Shannon-McMillan-Breiman theorem, the cross-entropy simplifies further

$$\begin{aligned} H(p_{data}, p_{model}) &= \mathbb{E}_{p_{data}}\left[-\frac{1}{T}\log_2 p_{model}(\mathbf{x})\right] \\ &\approx -\frac{1}{T}\log_2 p_{model}(\mathbf{x}) \quad [bits] \end{aligned}$$

as each sequence occurs in proportion to its probability anyways if we consider long enough sequences (think of an infinite text $\mathbf{x}$ generated from the data distribution). Because this model distribution factorizes into conditional probabilities where a word depends on all the previous words. So one can think of this with the logarithm as a large sum. So we have to sum over the entire dataset of for example sentences and then we have to sum over each sentence. Now if we let the sequence or sentence length go to infinity then the expectation does not matter anymore because each sequence occurs in proportion to its probability anyways. So each of these terms that are added together anyways occur in proportion to their probability. So for long sequence lengths this is a reasonable approximation to make. Therefor in practice this is typically computed when cross entropy in bits is reported, which is the measure for character level language models. Additionally in practice, $H(p_{data}, p_{model})$ is evaluated on a test or validation sequence $\mathbf{x}$. It is not evaluated on the training sequence because that might just measure overfitting, but we are measuring generalization performance by evaluating on a test or validation sequence $\mathbf{x}$.

Example 1:
Consider a vocabulary $\mathcal{V} = \{A, B\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$. So the probability distribution over sequences of length t factorizes completely into the probability of each individual token in a sequence. In other words in this simple example the probability of the next word is completely independent of the previous words. Let's further assume that the probability for each of the symbols in the vocabulary is fifty percent: $p(x_t) = \frac{1}{2}$ for both data and model distribution.
Then

$$H(p_{data}, p_{model}) = -\frac{1}{10}\log_2\left(\frac{1}{2}\right)^{10} = \log_2 2 = 1 \; bit$$

The amount of information needed to predict the next character is 1 bit with this simple (unigram) model as the next character is either A or B with equal probability. In other words, we can't find a better encoding of this language than using 1 bit per character.
Remark: A uniform distribution always maximizes the entropy ($\Rightarrow$ upper bound for $|\mathcal{V}|$).

Example 2:
Consider a vocabulary $\mathcal{V} = \{A, B\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 1$ and $p(x_t = B) = 0$
for both data and model distribution.
Then

$$H(p_{data}, p_{model}) = -\frac{1}{10} \log_2 1^{10} = \log_2 1 = 0 \; bits$$

In this case, the amount of information needed to predict the next character is 0 bits as the next character is always A. In other words, we don't need any capacity to transmit this language through some channel, it contains no information.
Remark: 0 bits is the minimal value for the entropy or cross-entropy ($\Rightarrow$ lower bound).

Example 3:
Consider a vocabulary $\mathcal{V} = \{A, B\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 0.1$ and $p(x_t = B) = 0.9$
for both data and model distribution.
Then

$$H(p_{data}, p_{model}) = -\frac{1}{10} \log_2 \left(\frac{1}{10}\right)^1 \left(\frac{9}{10}\right)^9 = 0.47 \; bits$$

We need 0.47 bits now as we sometimes observe A, but most often B. Thus, the information conveyed in this language is larger than 0 bits (lower bound) and smaller than 1 bit (upper bound).

Example 4:
Consider a vocabulary $\mathcal{V} = \{A, B\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 0.1$ and $p(x_t = B) = 0.9$
for the model distribution and $p(x_t = A) = 1$ for the data distribution.
Then

$$H(p_{data}, p_{model}) = -\frac{1}{10} \log_2 \left(\frac{1}{10}\right)^{10} = \log_2 10 = 3.32 \; bits$$

Because in the model we have a probability of 0.1 of observing A but in the data we only observe A.
We need more than 1 bit now as the model fits the data badly. We need 0.47 bits to encode any possible outcome of p_{data} using the code optimized for p_{data} and 2.85 bits to encode any possible outcome of p_{data} using the code optimized for p_{model}:

$$H(p_{data}, p_{model}) = \underbrace{H(p_{data})}_{\geq 0} + \underbrace{D_{KL}(p_{data} || p_{model})}_{\geq 0}$$

In other words, and this is universally correct, the cross entropy of the model distribution with respect to the data distribution is th entropy of the data distribution plus the KL divergence between the data and the model distribution. Because this KL divergence must always be bigger or equal to zero we know that the entropy of p_{data} is a lower bound on the cross entropy of p_{model} with respect to p_{data}. So the cross entropy that we obtain, because our model is always imperfect, must always be bigger or equal to the entropy of the data distribution.

Evaluating Word Language Models
It would be natural to measure **word language models** in bits per word.
However, word language models are traditionally measured in **perplexity**. Which is a measure that is very related to bits per weight to the cross entropy:

$$\begin{aligned}
\text{Perplexity}(p_{data}, p_{model}) &= 2^{H(p_{data}, p_{model})} \\
&\approx 2^{-\frac{1}{T} \log_2 p_{model}(\mathbf{x})} \\
&= p_{model}(\mathbf{x})^{-\frac{1}{T}} \\
&= \left(\prod_{t=1}^{T} p_{model}(x_t | x_1, \ldots, x_{t-1}) \right)^{-\frac{1}{T}}
\end{aligned}$$

Thus, perplexity can be interpreted as the **inverse probability of the test set**, normalized by the sequence length T which acts as a geometric mean.
Again, Perplexity(p_{data}, p_{model}) is evaluated on a test or validation sequence $\mathbf{x}$.

Example 1:
Consider a vocabulary $\mathcal{V} = \{A, B, C\}$ and sequences of length $T = 10$.
Again we assume unigrams: $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t) = \frac{1}{3}$ for both data and model distribution.
Then

$$\text{Perplexity}(p_{data}, p_{model}) = \left(\left(\frac{1}{3}\right)^{10}\right)^{-\frac{1}{10}} = 3$$

We see that the perplexity models the number of possible next tokens to choose from (i.e., here the model is maximally confused which of the 3 tokens A, B or C to pick). Because they all occur independently with equal probability of one third.
Thus, perplexity is often also called the **average weighted branching factor.** Because we have three branching opportunities here that are all equally likely, so the model has a average branching factor of three.
Remark: A uniform distribution maximizes the perplexity ($\Rightarrow$ upper bound for $|\mathcal{V}|$).

Example 2:
Consider a vocabulary $\mathcal{V} = \{A, B, C\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 1$ and $p(x_t \in \{B, C\}) = 0$,
for both data and model distribution.
Then

$$\text{Perplexity}(p_{data}, p_{model}) = \left(1^{10}\right)^{-\frac{1}{10}} = 1$$

We see that the perplexity reduces in this case as the next choice is certain. The average branching factor is one, we know we have to choose A. The model is not surprised to see the test set as it is able to predict the test set exactly (all A's).
Remark: 1 is the minimal value for the perplexity measure ($\Rightarrow$ lower bound). However, this is only achievable for languages that contain only a single token.

Example 3:
Consider a vocabulary $\mathcal{V} = \{A, B, C\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 0.1$, $p(x_t = B) = 0.9$, and $p(x_t = C) = 0$ for both data and model distribution.
Then

$$\text{Perplexity}(p_{data}, p_{model}) = \left(\left(\frac{1}{10}\right)^{1}\left(\frac{9}{10}\right)^{9}\right)^{-\frac{1}{10}} = 1.38$$

In this case, the perplexity is slightly larger than 1 as the model is quite certain to predict B as the next character, but sometimes it should predict A.

Example 4:
Consider a vocabulary $\mathcal{V} = \{A, B, C\}$ and sequences of length $T = 10$.
Assume $p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t)$ with $p(x_t = A) = 0.1$, $p(x_t = B) = 0.9$, and $p(x_t = C) = 0$ for the model distribution and $p(x_t = A) = 1$ for the data distribution.
Then

$$\text{Perplexity}(p_{data}, p_{model}) = \left(\left(\frac{1}{10}\right)^{10}\right)^{-\frac{1}{10}} = 10$$

In this case the perplexity is larger than 3 as the model fits the data badly.

9.1.4 Summary

- For character language models, current performance is roughly 1 bit per character
- For word language models, perplexities of about 60 were typical until 2017
- According to Quora, there are 4.79 letters per word (excluding spaces)
- Assuming 1 bit per character, we have a perplexity of $2^{5.79} = 55.3$
- State-of-the-art models (GPT-2, Megatron-LM) yield perplexities of $10 - 20$
- Be careful: Metrics not comparable across vocabularies or datasets

Additional Resources:
https://thegradient.pub/understanding-evaluation-metrics-for-language-models/
https://towardsdatascience.com/the-relationship-between-perplexity-and-entropy-in-nlp-f81888775ccc

9.2 Traditional Language Models

In this section we're going to be talking about n-grams which have been the default language model for several decades. A quick recap about **language models**:

$$\begin{aligned} p(\mathbf{x}) = p(x_1, \ldots, x_T) &= \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1}) \\ &= p(x_1)\, p(x_2|x_1)\, p(x_3|x_1, x_2) \ldots \end{aligned}$$

- Probability distribution over a sequence of **discrete tokens** $\mathbf{x} = (x_1, \ldots, x_T)$ where each token can take a value from a **vocabulary** $\mathcal{V}$ ($x_t \in \mathcal{V}$)
- Decomposes into a sequence of conditional distributions
- The history (conditioning variables/tokens) is called **context** in NLP
- We can represent each conditional distribution with a **probability table** and learn the entries of these tables, but this becomes intractable as T grows
- The probability table for $p(x_T|x_1, \ldots, x_{T-1})$ comprises $|\mathcal{V}|^T$ entries where the vocabulary size $|\mathcal{V}|$ of word language models is roughly $|\mathcal{V}| \approx 30.000$
- Huge **memory** and **training sets** needed (many long sentences are very rare)

9.2.1 N-gram Models

- An n-gram is a sequence of n tokens (e.g. 2-grams or bigrams: "The dog", "dog ran", "ran away")
- n-gram models **approximate the history context** by the last $n-1$ tokens, they shorten the context. So we are cutting off the first words and only keeping the last n minus one words in the history context to predict the current word
- In other words, they make a **Markov assumption** (the model is **memoryless**), which makes this much more tractable, in particular if we consider small ns
- This idea is similar to the autoregressive models that we have introduced, except that each conditional $p(x_t|x_{t-n+1}, \ldots, x_{t-1})$ is represented by a probability table and not a neural network
- Early language models considered bigrams ($n = 2$) and trigrams ($n = 3$)

Word Language Model Examples:

- **bigram:** ($n = 2$, or in other words: history length $= 1$)

$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_1)\, p(x_2|x_1)\, p(x_3|x_2)\, p(x_4|x_3) \\ p(\text{The dog ran away}) &= p(\text{The})\, p(\text{dog}|\text{The})\, p(\text{ran}|\text{dog})\, p(\text{away}|\text{ran}) \end{aligned}$$

- **trigram:** ($n = 3$, history length $= 2$)

$$\begin{aligned} p(x_1, x_2, x_3, x_4) &= p(x_1, x_2)\, p(x_3|x_1, x_2)\, p(x_4|x_2, x_3) \\ p(\text{The dog ran away}) &= p(\text{The dot})\, p(\text{ran}|\text{The dog})\, p(\text{away}|\text{dog ran}) \end{aligned}$$

9.2.2 Training of n-gram Models

The **conditional probability** can be written as:

$$p(x_t|x_{t-n+1}, \ldots, x_{t-1}) = \frac{p(x_{t-n+1}, \ldots, x_t)}{p(x_{t-n+1}, \ldots, x_{t-1})} = \frac{p(x_{t-n+1}, \ldots, x_t)}{\sum_{x_t} p(x_{t-n+1}, \ldots, x_t)}$$

For a **bigram** model this would yield:

$$p(x_t|x_{t-1}) = \frac{p(x_{t-1}, x_t)}{p(x_{t-1})} = \frac{p(x_{t-1}, x_t)}{\sum_{x_t} p(x_{t-1}, x_t)}$$

We see that we simply need to **count** the number of n-grams and (n-1)-grams in the training set to populate the probability table of the n-gram model. We simply need to count um how often is x_{t-1} followed by x_t divided by the number of occurrences of x_{t-1}.
Smoothing: For large n, the n-gram probabilities are often zero as they haven't been observed in the training set but they might occur in the test set which means that we are overfitting. A simple heuristic is to add one to all n-gram counts.

9.2.3 Sampling from n-gram Models

- As n-gram models are autoregressive models, **sampling is easy**
- We just need to iteratively draw tokens from $p(x_t|x_{t-n+1}, \ldots, x_{t-1})$ until we reach the end of sentence symbol <EOS>

Example of a random sentence drawn from a Jane Austen trigram model:

> "You are uniformly charming!" cried he, with a smile of associating and now and then I bowed and they perceived a chaise and four to wish for."

As as one can read, this might not make a lot of sense but it's still natural language as it could potentially occur.

Samples from a Shakespeare language model:

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less f rst you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Figure 144: **Samples from a Shakespeare language model.** https://web.stanford.edu/~jurafsky/slp3/

One can see that as we increase the context size from a unigram, to a bigram, to a trigram, to a 4-gram, that the text becomes much more realistic.

9.2.4 Summary

- n-gram models are simple models that make a **Markov assumption** to model a distribution over sequences via probability tables
- However, they have **limited history context**, they cannot model long-term dependencies, and **parameters grow exponentially**
- **Smoothing heuristics** are required to deal with the resulting sparsity
- They can't directly model conditional distributions (e.g., for translating sentences)
- In contrast to neural language models, they can be considered as **local non-parametric** predictors (thus suffering the curse of dimensionality)
- Tokens are encoded as **discrete items**
  - Large vocabularies are typically reduced to a shortlist (removing infrequent words)
  - Any two words have the same distance ($\sqrt{2}$ in one-hot vector space)
  - Thus, n-gram models **can't share information** between related (=close) words

9.3 Neural Language Models

The motivation for this unit is the fact that we cannot represent semantically related words with n-gram models, which would be beneficial for a machine learning model. Neural language models now can utilize neural networks for predicting the next word.

9.3.1 Local Word Representations

- **n-gram** models, that effectively are just conditional probability tables, suffer from the **curse of dimensionality**
- Modeling the joint distribution of $n = 10$ consecutive words with a vocabulary of size $|\mathcal{V}| = 10000$ results in intractable $|\mathcal{V}|^n = 10^{40}$ parameters, even if we limit the history context to 10.
- However, and this is where the idea for this type of neural language models came from, when modeling continuous variables, we obtain generalization more easily than in the discrete case, because the function to be learned is expected to behave **locally smoothly**
- This has been observed in other contexts but they haven't been used for NLP and its usage here has made a tremendous impact on the performance of NLP algorithms
- Discrete word representations assume the same **distance between words:**

$$\left\| \underbrace{(1,0,0,\ldots,0)^\top}_{\mathbf{w}_1} - \underbrace{(0,1,0,\ldots,0)^\top}_{\mathbf{w}_2} \right\|_2 = \sqrt{2}$$

- Remark: In this unit we use $\mathbf{w}$ to denote a one-hot encoding of a word

9.3.2 Word Representations

However, **some words are more similar than others**. Consider the sentences:

- "The cat is walking in the bedroom"
- "The dog was running in a room"

The sentences at the one-hot-vector level are very different but semantically they are quite related and having seen the first sentence (in the training set) should help to generalize making the second sentence very likely as

- "cat" and "dog",
- "walking" and "running",
- "the" and "a", ...

have similar semantic roles. But this cannot be captured in purely discrete models.

9.3.3 Distributed Word Representations

Therefor the idea is to use a different representation, a so-called distributed representation:

- **Distributed representations** map one-hot vectors $\{0,1\}^{|\mathcal{V}|}$ of high dimensions (e.g., $|\mathcal{V}| = 10000$) to word embeddings $\mathbb{R}^M$ of lower dimensions (e.g., $M = 30$)
- Thus, they distribute the representation along all dimensions (of the embedding vector), so we now have a real valued vector where there is a different real number for each of these M dimensions, so the semantics of that word is now distributed in that representation and we are able to **model similarity between words**
- They distribute probability mass where it matters rather than uniformly in all directions around each training point
- They therefore allow **generalization across sentences**
- Word embeddings are often realized through a simple **linear mapping** (it's basically just a matrix multiplication where the one hot encoder encoding goes in and out comes this lower dimensional vector that is a matrix) which can then be further processed by a feedforward or recurrent neural network

Remark: Unlike in this illustration, we do not attach actual labels to words. The depiction in Fig. 145 is only about semantic relatedness. For example student, colleges and schoolhouse are all related to academic, and academic is related to papers, so student, colleges and schoolhouse are more likely to be followed by papers, than by bill.

Example: Assume a training set with the following 3 sentences:

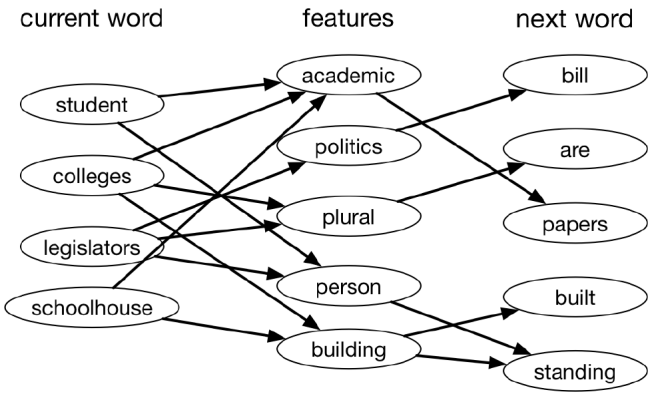


Figure 145: **Distributed Word representations.** Depiction of distributed word representations.

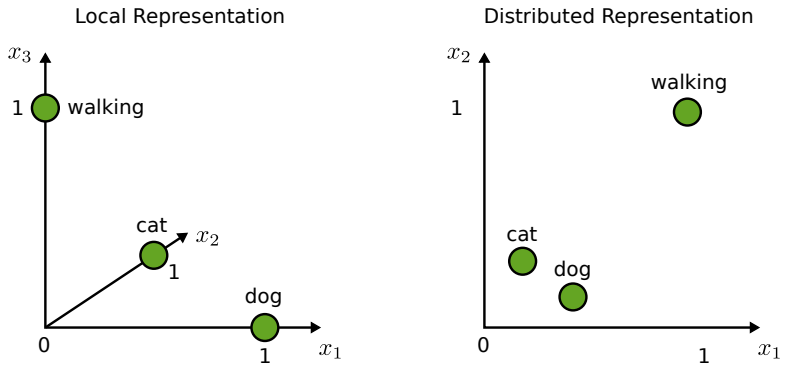


Figure 146: **Local vs Distributed Word representations.**

- "The cat is walking"
- "The dog is walking"
- "The cat is sitting"

A distributed representation learns to embed "cat" and "dog" nearby. So the distance between those two words should be low.
Given that p(The cat is sitting) is high, and "cat" and "dog" are related

- "The dog is sitting"

is also likely despite not being part of the training set.
Through this similarity we can model sentences to be more likely that are actually not part of the training set. An n-gram model can't generalize in this setting.
As we can see in Fig. 146, we have a local representation on the left of walking, cat and dog. This is a one hot encoding so they are on the axis and the distance between all of these is the same: $\sqrt{2}$. The goal is to transform this into a lower dimensional representation, a distributed representation. In this particular case we go from three-dimensional to two-dimensional, where now cat and dog are closer to each other than they are to walking, as can be seen in Fig. 146 on the right.

Once we have learned these word embeddings, we can also visualize these word embeddings. An example of a two-dimensional t-SNE visualization of a word embedding can be seen on Fig. 147. The shown regions are zoomed in because the vocabulary is much larger. When zooming into particular regions we see that in the region on the left we see countries and in the region on the right we observe dates or years. So the model has learned to relate the semantic entities to each other by learning them from text.
But we always have to be careful with 2D visualizations - Geoffrey Hinton once said "In a 30-dimensional grocery store, anchovies can be next to fish and next to pizza toppings." This illustrates how difficult it is to imagine and how easy it is to be misled by high dimensional spaces. In high dimensions everything can be next to everything. Even if we visualize word embeddings we can really just do it in 2D, everything else is beyond the scope of our imagination.

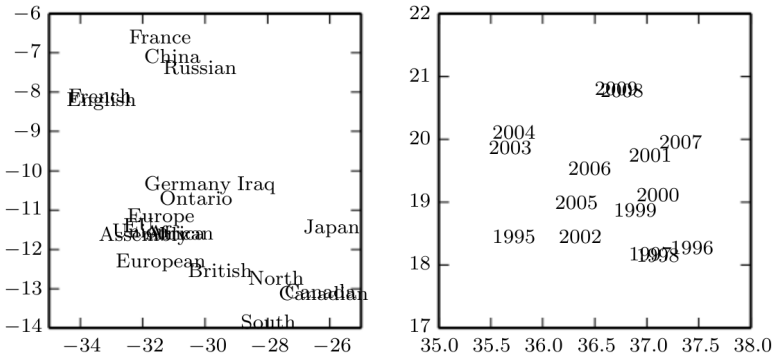


Figure 147: **Word embeddings.** Two-dimensional t-SNE visualization of a word embedding model.

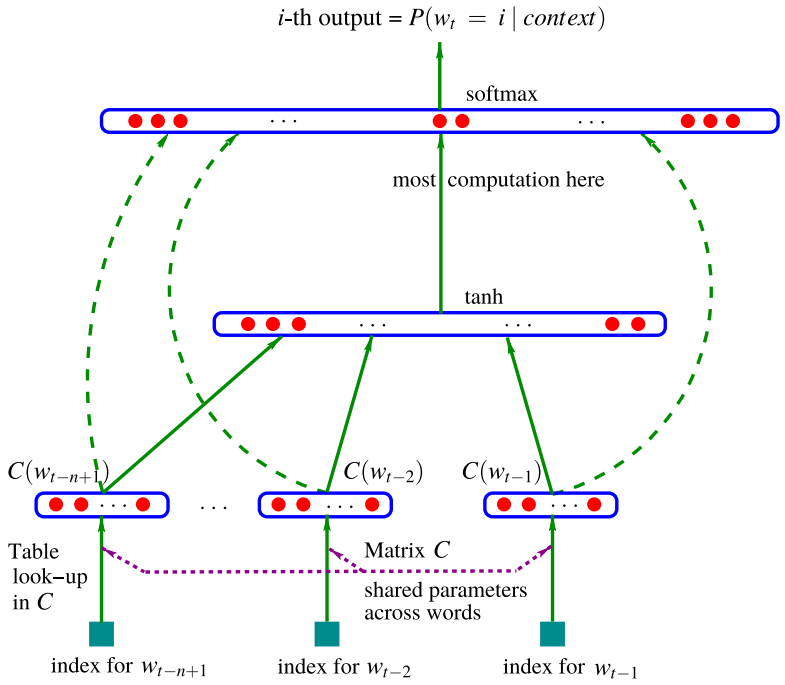


Figure 148: **Neural Probabilistic Language Model**

9.3.4 Neural Probabilistic Language Model

The neural probabilistic language model [2] was the first language model that used a distributed representation which have been used before in connectionism (Hinton, Elman) and symbolic reasoning (Paccanaro), but not for language, which was a breakthrough for NLP.

Key Ideas:

- Associate with each word in the vocabulary a distributed **word feature vector** (a real-valued vector in $\mathbb{R}^M$)
- Express the **joint probability function** of word sequences in terms of the feature vectors of these words in the sequence, and
- Learn simultaneously the **word feature vectors** and the parameters of that **probability function**

So we have an end-to-end training process. This was the first end-to-end model that simultaneously learned the word feature vectors and the parameters of the probability function.

Feedforward Model:

- Input: Sequence of words
- Output: Prob. of next word
- 3 Layers:
  1. Fully connected ⇒ embedding
  2. Fully connected + tanh
  3. Fully connected + softmax
- Input to 2^{nd} layer: vector of concatenated word embeddings
- Optional: direct connections

Formal description of the Feedforward Model:

$$P(\mathbf{w}_t|\mathbf{w}_{t-1},\dots,\mathbf{w}_{t-n+1}) = \frac{e^{y_{\text{idx}(\mathbf{w}_t)}}}{\sum_i e^{y_i}}$$
$$\mathbf{y} = \mathbf{b} + \mathbf{W}\mathbf{x} + \mathbf{U}\tanh(\mathbf{d} + \mathbf{H}\mathbf{x})$$
$$\mathbf{x} = (\mathbf{C}\,\mathbf{w}_{t-n+1},\dots,\mathbf{C}\,\mathbf{w}_{t-1})$$

where $\mathbf{C} \in \mathbb{R}^{M\times|\mathcal{V}|}$. Thus the model

- .. scales linearly with $|\mathcal{V}|$
- .. scales linearly with n

The model is shown in Fig. 148. The model has some inputs at the bottom of the figure which are the indices of the words, where the words are one-hot-vectors. This builds a sequence of words which then is transformed by the following matrix $\mathbf{C}$ from a one-hot-vector to a word embedding. So for each word we get a high-dimensional word embedding and then all of these word embeddings get concatenated into a vector. Which builds the first layer, which is then fully connected and followed by a tanh as a hidden second layer. Next this is fully connected and followed by a softmax as the third layer. This softmax is then outputting the probability of the next word $\mathbf{w}_t$ in the sequence given all the previous words $\mathbf{w}_{t-n+1}$ until $\mathbf{w}_{t-1}$.
There also exists a small variation of this model which is visualized with the dotted lines in Fig. 148. These lines show direct connections from the first layer to the last layer
In the formal description of this feedforward model the probability for the next word $\mathbf{w}_t$ is given by a softmax. $y_{idx(\mathbf{w}_t)}$ in the softmax is used to get the index of the one-hot-vector which is then normalized by all the others. The $\mathbf{y}$'s then are linear predictions of the direct connections followed by a tanh-nonlinearity from $\mathbf{x}$. $\mathbf{x}$ is a concatenation of all the one-hot-vectors for each previous word multiplied with the embedding matrix $\mathbf{C}$. Now $\mathbf{C}$ is in $\mathbb{R}^{M\times|\mathcal{V}|}$. We can see that the model scales linearly with the vocabulary size because if we increase the vocabulary then $\mathbf{C}$ also has to increase. But different to before now the model also scales linearly with n. Before we had $|\mathcal{V}|^n$ and but now the model scales linearly with n because we are increasing the length of the concatenated vector that we input to the second layer and so the number of parameters in that second layer increases linearly, not exponentially.
This is actually a common trick that's used when working with neural networks that instead of representing some conditioning explicitly, the conditioning is an input to a neural network and therefore, in terms of complexity, the model becomes more tractable. This is also something that's for instance used in the context of reinforcement learning. This way there is a lot more sharing of information and parameters that takes place.

- Neural probabilistic language models lead to significantly better results in comparison with the best of the n-grams, with a test perplexity difference of about 24% on Brown and about 8% on AP News (which are both datasets), when taking the MLP versus the n-gram that worked best on the validation set.
- The results also suggests that the neural network was able to take advantage of more context (on Brown, going from 2 words of context to 4 words brought improvements to the neural network, not to the n-grams). This is because of the similarities that are learned and the generalization that is implied by it.
- Also showed that the hidden units are useful (MLP3 vs MLP1 and MLP4 vs MLP2), and that mixing the output probabilities of the neural network with the interpolated trigram always helps to reduce perplexity.

9.3.5 Word2Vec / Skip-Grams

There are a number of papers called Word to Vec - efficient estimation of word representation vector space.

- Fitting language models is hard (large $|\mathcal{V}|$ ⇒ **large softmax**, memory intensive and computation intensive)
- **Skip-grams** predict a word in their surrounding context, which is kind of an unsupervised training task
- Instead of predicting a distribution over words,
  switch to a **binary prediction problem**
- The model is given pairs of words and needs to distinguish if the words occur next to each other in the training corpus or they are sampled randomly

- Logistic regression on inner product of word embeddings
- Can be **trained very efficiently** with lots of data

Using this we can some word vector arithmetics:
Word Vector Arithmetics

Expression	*Nearest token*
Paris - France + Italy	Rome
bigger - big + cold	colder
sushi - Japan + Germany	bratwurst
Cu - copper + gold	Au
Windows - Microsoft + Google	Android
Montreal Canadiens - Montreal + Toronto	Toronto Maple Leafs

Figure 149: **Word Vector Arithmetics**

In Fig. 149 we can see some expression on the left and the nearest most likely token on the right. For example, we can take the word embedding vector of Paris and subtract the word embedding of France and add Italy and we get Rome as an outcome. This also shows that the semantics that this word embeddings have learned is quite meaningful.

9.4 Neural Machine Translation

Neural machine translation is one of the landmark applications of neural language models.

9.4.1 Sequence to Sequence Learning

Sutskever et al. [13] proposed a very simple end-to-end trainable model for machine translation that shocked the community because it was the first paper to produce results that outperformed all of the previously developed hand-engineered rule-based systems for machine translation. The key behind this was a very well engineered but simple model in combination with a lot of compute and data.
Sequence to Sequence Learning Model

- **Two 4-Layer LSTMs** for encoding/decoding the source/target sentence
- Encoding operates in **reverse order** of the input sentence to introduce short-term dependencies. Because we end with the first word and then the decoder starts to decode with the first word in the target language. By reversing the input sentence you get shorter dependencies because the first word in the input sentence is likely related to earlier words in the output sentence.

- Otherwise this is a simple LSTM that processes the reverse ordered words, so that the words are embedded in a distributed word representation. Also there is a hidden state that is carried forward through this recurrent neural network and then there's an intermediate representation:
- Intermediate representation produced by the encoder is called **thought vector**. This is the representation that stores the meaning of that sentence that shall be translated. So once we reach the end of sentence symbol we have the thought vector as the hidden representation. Which is a global representation of the input sentence and then that fourth vector is decoded. The reason why we have this interface of the

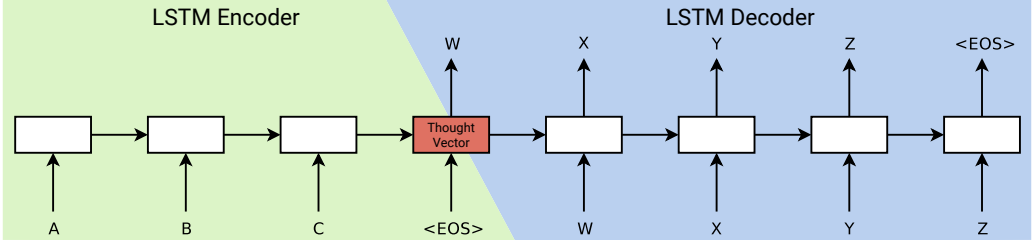


Figure 150: **Sequence to sequence modelling**

forward vector is that the input sentence and the output sentence may have different length, also the word order may be very different in different languages depending on the grammar. So we create a global representation of the input sentence default vector that is passed to decoder.

- Encoding using 1000 dim. word embeddings, decoding via **beam search**
- First end-to-end system that outperforms rule-based models and also the first system that demonstrated that this is really possible at large scale which led to deployment.

9.4.2 Decoding

Let $\mathbf{w}_1, \ldots, \mathbf{w}_T$ denote the target sentence and let $\mathbf{v}$ denote the thought vector. **Sampling** a translation from the LSTM decoder is simple, because we have the autoregressive nature that we can just sample a new word from the distribution that is predicted by the LSTM, given the thought vector and the previous words. Where the previous words are summarized in this hidden state in the case of an LSTM:

$$\mathbf{w}_t \sim p(\mathbf{w}_t|\mathbf{v}, \mathbf{w}_1, \ldots, \mathbf{w}_{t-1})$$

But this is not what we want to do in translation, in that case we like to compute the **most probable translation:**

$$\mathbf{w}_1, \ldots, \mathbf{w}_T = \underset{\mathbf{w}_1,\ldots,\mathbf{w}_T}{\operatorname{argmax}} \; p(\mathbf{w}_1, \ldots, \mathbf{w}_T|\mathbf{v})$$

This is costly because there is a large number of possibilities. So searching that exhaustively is intractable. But often a **greedy algorithm** often works well in practice:

$$\mathbf{w}_t = \underset{\mathbf{w}_t}{\operatorname{argmax}} \; p(\mathbf{w}_t|\mathbf{v}, \mathbf{w}_1, \ldots, \mathbf{w}_{t-1})$$

So we only take the argmax over $\mathbf{w}_t$ instead of over the sequence $\mathbf{w}_1, \ldots, \mathbf{w}_T$. Also the probability is calculated for $\mathbf{w}_t$ given the thought vector $\mathbf{v}$ and $\mathbf{w}_1, \ldots, \mathbf{w}_T$.

9.4.3 Beam Search

Failure of Greedy Algorithm:

- p(Apples are good) > p(Those apples are good), because "apples are good" occurs more frequently in the training corpora.
- However there are more sentences in the training corpora that start with "those" compared to "apples": p(Those) > p(Apples)
- So if we do greedy decoding then we would start with "those" even though the probability of "apples are good" is bigger than the probability of "those apples are good".

This is where beam search comes in. It is better than a purely greedy algorithm but it's still only an approximation to the true solution that searches over the entire search space.

Idea of Beam Search:

- At each time step, **maintain a list** of the K best words and hidden vectors
- This can be used to produce a list of K best decodings for the following word, which can then be compared to select the most likely one

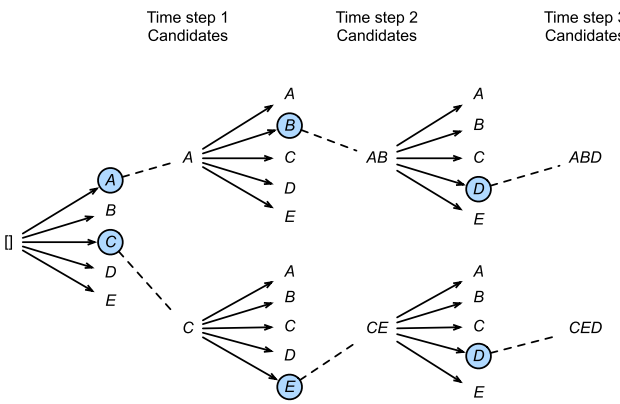


Figure 151: **Illustration of beam search.** https://d2l.ai/chapter_recurrent-modern/beam-search.html

In Fig. 151 an illustration of beam search can be seen. We start with an empty sequence and then we have A, B, C, D, E as candidates and A and C are most likely so with a beam of size two we we keep track of A and C.Next for A B is most likely and for C E is most likely. Then we continue with AB and CE. We always maintain this short list of two hypotheses. If in the first step AB and AC would both be more likely than CE then we would of course continue with those two possibilities.

9.4.4 The Transformer

In 2017 there was another big step and it was the transformer paper [15] that completely changed the architecture of previous neural machine translation models into a purely attention-based architecture.

- **Attention based** model which doesn't rely on recurrence or convolution.
- The advantage of an attention-based model over an RNN for instance is that there is no sequential dependencies. One doesn't need to wait for the previous words to be computed until you can compute the probability for the current word, all the tokens can be processed in parallel. This is advantageous because nowadays all the computational gains are through parallelism.
- Leads to significant speed-ups when using modern GPU clusters
- **Self-attention** relates all tokens in a layer with each other
- Thus can more easily capture **long-distance dependencies** compared to an RNN
- Transformer-like architectures have now replaced RNNs in NLP applications,
  Defacto standard for all state-of-the-art models (e.g., on SuperGLUE benchmark)

Architecture of the Transformer

- Each **layer** in the Transformer has shape $L[T, J]$ where t ranges over the position in the input sequence and j ranges over features at that position. L is basically a matrix where for each position for each word we have a vector, a word embedding.
- When processing sentences of words, T is the sentence length
- This is the same shape as in an RNN – a sequence of vectors $L[t, J]$
- However, unlike in RNNs, in the Transformer we can compute the layer $L_{\ell+1}[T, J]$ from $L_\ell[T, J]$ in **parallel**
- In this respect, the transformer is more similar to a CNN than to an RNN

Self-Attention

- The fundamental innovation of the Transformer is the **self-attention layer**
- For each position t in the sequence we compute an attention over the other positions in the sequence
- The transformer uses **multiple heads** (because multiple heads are empirically better), i.e., it computes the attention operation multiple times ($K = 8$ in the original implementation)
- **Self-attention** then constructs a tensor $A[k, t_1, t_2]$ – the strength of the attention weight from t_1 to t_2 for head k. So the attention that t_1 pays to t_2 in a particular layer of the transformer for a particular head k.
- In the paper, an **embedding dimension** of $D_J = 512$ is chosen per token
- Using $K = 8$ heads, this results in a dimension of $D_Q = D_K = D_V = 64$ for the query, key and value embeddings that are used for each token (can be different). So they do computations at 64 dimensions but they do it eight times and then they concatenate them again to get 512 dimensions.

Multi-Headed Self-Attention:
In the following we see the equations for the (multi-headed) self-attention-layer, the most fundamental layer of The Transformer. Note that there are more layers which are not described here.
For each head k and word position t, we compute a **key**, **query** and **value** vector. The queries Q and the keys

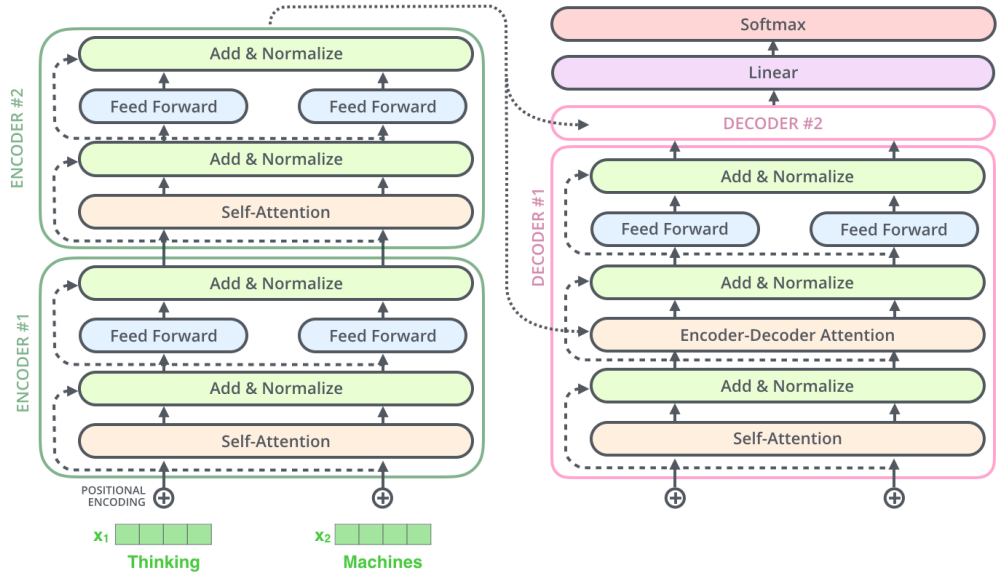


Figure 152: **The Transformer.** http://jalammar.github.io/illustrated-transformer/

K are used to compute the self-attention matrix A for head k. A is then multiplied with the values V to yield embedding vectors H that are concatenated.

$$Q_{\ell+1}[k,t,i] = W^Q_{l+1}[k,i,J]\, L_\ell[t,J] \tag{41}$$

$$K_{\ell+1}[k,t,i] = W^K_{l+1}[k,i,J]\, L_\ell[t,J] \tag{42}$$

$$V_{\ell+1}[k,t,i] = W^V_{l+1}[k,i,J]\, L_\ell[t,J] \tag{43}$$

$$A_{\ell+1}[k,t_1,t_2] = \underset{t_2}{\text{softmax}}\left[\frac{1}{\sqrt{D_Q}} Q_{\ell+1}[k,t_1,I]\, K_{\ell+1}[k,t_2,I]\right] \tag{44}$$

$$H_{\ell+1}[k,t,i] = A_{\ell+1}[k,t,T]\, V_{\ell+1}[k,T,i] \tag{45}$$

$$L_{\ell+1}[t,j] = W^L_{\ell+1}[j,I]\, (H_{\ell+1}[1,t,I],\ldots,H_{\ell+1}[K,t,I]) \tag{46}$$

(1)-(3): So we start at the bottom where we have just the words represented as one-hot-vectors or word embeddings of one-hot-vectors. Then we multiply this first layer with a matrix W to get a vector. This multiplication runs over i and yields a 64 dimensional vector for each head and each word. This is done three times, once for the query vector, once for the key vector and once for the value vector.

(4): For computing the attention matrix A, we first take the dot product of the query and the key vector. The divison by the square root of D_Q serves as a normalization factor. All of this is then put into a softmax over t_2. As can be seen in the equation, Q goes over t_1 and K goes over t_2 and so we get this matrix A were t_1 attends to t_2.

(5): Next we multiply this attention matrix A to the value vector V, where we now sum over the T dimension of this vector V and the t_2 dimension of matrix A.

(6): So now we have these 64 dimensional vectors H that we now concatenate over the different heads and then we compute a matrix product with W^L in oder to get the output of that layer.

In summary we compute queries, we compute keys and we compute values for each head and for each time step, or word, and then from the queries and keys we compute attention and that attention is multiplied with the values so we take the values where the model attends to and these are then concatenated and fed into the next layer. Fig. 152 shows an illustration of the transformer that includes the here described self-attention layer which is the core of the transformer.
If one wants to use the transformer for translation you will have a sequence of input words that are going to be encoded which are depicted on the bottom left of Fig. 152, underneath the encoder. Then information from this encoder is given to the decoder. In particular the key and the queries are given to the decoder and they are combined with the values of each word in the sequence encoded through this encoder that's part of the decoder. The output sentence is successfully established through the decoder by combining keys and queries from the encoder with values from the decoder and the predicting the next word.

10 Graph Convolution Networks

10.1 Machine Learning on Graphs

This section gives a motivation why GCNs have a better performance compared to other network architectures and illustrates some areas of application of GCNs.

10.1.1 Motivation

Multi layer perceptrons (MLPs) are very flexible function approximators. Theoretically a MLP with only one layer can already act as a universal function approximator, given that this layer can grow infinitely wide. However, MLPs do not scale well. If the input to an MLP is large the amount of model parameters grows large as well. Furthermore, the more parameters a neural network has, the more it tends to overfit: Its generalization capability decreases. For structured signal grids like 2D images or 1D time series a CNN addresses this issue and allows good generalization due to its convolution operation over the the regular grid of signals, which reduces the number of signals. Unfortunately a lot of signals cannot be described in such a structured way such as molecules or natural language. These signals may be better described with the help of graphs. In order to exploit graph structured data, a model class is necessary that scales better than an MLP when receiving large input data without loosing its predictive power. Further on this model has to be more flexible than a CNN in order to exploit the local connectivity structure of any graph - not only structured ones - as prior information.

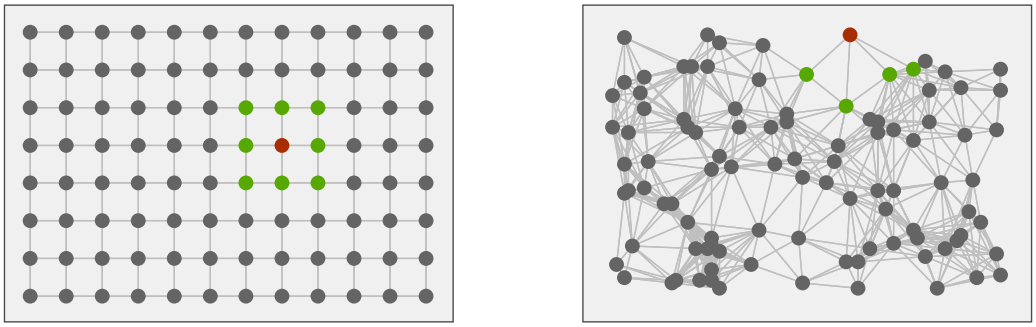


Figure 153: **Comparison of graph structures.** An illustration of a regular structured graph (left) and an unstructured graph (right). The red dots marks the point of interest that is calculated by exploiting the graph structure and the neighboring nodes (green).

Graphs are descriptors of a signal structure, where the signals are described as nodes (points) and the similarity between signals with edges (lines between signals). Fig. 153 shows a comparison between a structured graph (left) and an unstructured graph (right). On both graphs a convolution operation is applied in order to calculate the red dot by exploiting the graph structure and the neighboring nodes (green). On the structured graph on the left a convolution filter is applied, that computes a value for the red dot by calculating the dot product between the elements in green and red. The equivalent operation is done on the right by using a convolution that exploits the locality in the graph. These convolutions on unstructured graphs are polynomials conditioned on the graph structure with the graph structure being encoded in a matrix derived from the graph.

10.1.2 Applications

Since different types of data can be represented in form of a graph, GCNs can be applied in several tasks in multiple domains. Some of these applications are scene graph generation, multi-object tracking, authorship attribution, recommendation systems, learning molecular fingerprints, protein interface prediction, interaction networks, learnable physics engines for control, decentralized control of robot swarms.

10.2 Graphs

GCNs operate on graphs. Therefore some basic knowledge of graphs and operations on graphs is presented in this section.
A graph can be represented as a triplet $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ with vertices $\mathcal{V}$, edges $\mathcal{E}$ and weights $\mathcal{W}$. Vertices or nodes are defined as a set of N numerical labels $\mathcal{V} = \{1, \dots, N\}$. Edges are ordered pairs of these numerical labels (i, j), where $(i, j) \in \mathcal{E}$ is interpreted as "node i is influenced by node j". Weights $w_{ij} \in \mathbb{R}$ are numbers associated to edges (i, j) that determine the strength of the influence that node j has on node i. Depending on these basic properties, a graph can be classified as a directed graph or as a symmetric graph.

In the case of a **directed graph** Fig. 154 the edge (i, j) differs from edge (j, i). Thus, a connection between between two nodes can be one-way only, meaning that $(i, j) \in \mathcal{E}$ and $(j, i) \notin \mathcal{E}$. Furthermore, if a connection between two nodes is bidirectional $\{(i, j), (j, i)\} \subseteq \mathcal{E}$, their weights can be different $w_{ij} \neq w_{ji}$, e.g. the connections between nodes 3 and 5 in Fig. 154 could have different weights.
The **undirected** or **symmetric graph** is a directed graph, with the special property that its edge set and its weights are symmetric. Therefore, if the edge set $\mathcal{E}$ contains (i, j) it implies that $(j, i) \in \mathcal{E}$ as well. This is illustrated in Fig. 155 with the red line with arrows on both sides. For the weights of these connections (i, j) that means that $w_{ij} = w_{ji}$ for all $(i, j) \in \mathcal{E}$.
A special case in directed and symmetric graphs is the **unweighted graph**, where all existing connections $(i, j) \in \mathcal{E}$ have weights $w_{ij} = 1$. Since the weights of a graph convey valuable information as a prior for GCNs, most graphs that are encountered in the context of deep learning are weighted.

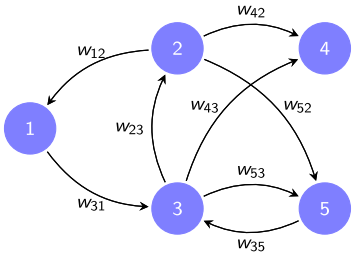


Figure 154: **Directed graph.** Nodes are illustrated by numerical labelled purple dots, edges are denoted by arrow lines and weights with a description in the form w_{ij}.

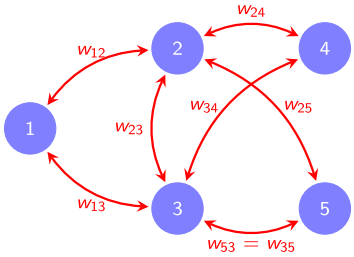
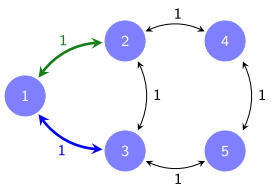
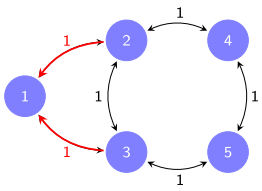


Figure 155: **Symmetric graph.** The illustration is similar to Fig. 154. Due to the symmetry of the graph the edges are directed in both directions and there is exactly one weight per edge.

Graph Matrix Representation. The visual graph with its connections can be mathematically represented with the help of different matrices. A common representation of the edges of a graph $\mathcal{G}$ is the **adjacency matrix A**. In this sparse $N \times N$ matrix, each row and each column represent a vertex and each entry A_{ij} contains the weight w_{ij} of all connections $i, j \in \mathcal{E}$. Furthermore if the graph is symmetric its holds that $\mathbf{A} = \mathbf{A}^\top$ as seen in Fig. 156.



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$



$$\mathbf{D} = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Figure 156: **Unweighted graph and corresponding matrices.** The unweighted graph is transformed into respective matrices **A** and **D**. Colors mark the corresponding location of the edges in the matrices

Another representation is the **degree matrix D** that contains the degree of a vertex on its diagonal axis (see Fig. 156). The **degree** d_i of node i is the sum of weights of its incident edges:

$$d_i = \sum_{j \in \mathcal{N}(i)} w_{ij}, \tag{47}$$

where $\mathcal{N}(i)$ - the **neighborhood** - is the set of nodes that influence node i:

$$\mathcal{N}(i) = \{j | (i, j) \in \mathcal{E}\}. \tag{48}$$

The diagonal D_{ii} therefore contains the degree d of vertex i: $D_{ii} = d_i$. **D** can be expressed in terms of adjacency matrix as $\mathbf{D} = \text{diag}(\mathbf{A1})$[4]
The **Laplacian matrix L** combines matrices **D** and **A** as follows:

$$\mathbf{L} = \mathbf{D} - \mathbf{A}. \tag{49}$$

It therefore contains **D** on its diagonal and **A** negated on its off-diagonals. Written explicitly in terms of graph weights the entries consist of $L_{ij} = -A_{ij} = -w_{ij}$ and $L_{ii} = d_i = \sum_{j \in \mathcal{N}(i)} w_{ij}$. One can think of this matrix as a Laplacian in regular domain. It therefore measures the smoothness of a graph.

[4] diag converts the vector $\mathbf{A1}^N$ into a $N \times N$ square matrix with the vector on its diagonal.

Both the Laplacian and the adjacency matrix can be normalized to achieve a more homogeneous representation of a graph. That is especially helpful for asymmetric graphs, where some nodes have a lot of neighbors and/or a high degree and some nodes have only a few neighbors and/or a low degree. The **normalized adjacency matrix** expresses weights relative to node degrees:

$$\bar{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} \quad \Rightarrow \quad \bar{A}_{ij} = \frac{w_{ij}}{\sqrt{d_i d_j}}. \tag{50}$$

The **normalized Laplacian matrix** is similarly defined as:

$$\bar{\mathbf{L}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{D} - \mathbf{A})\mathbf{D}^{-\frac{1}{2}} = \mathbf{I} - \bar{\mathbf{A}}. \tag{51}$$

The Laplacian matrix, the adjacency matrix and the normalized forms of both are in the following represented by the **Graph Shift Operators S**. That is, because for the theoretical analysis of GCNs the specific graph matrix representation is irrelevant. Nevertheless during deployment the specific representation matters and leads to different results. It holds that if $\mathcal{G}$ is symmetric $\mathbf{S} = \mathbf{S}^\top$.

Graph Signal Diffusion. Given the mathematical representation of a graph $\mathcal{G}$ as $\mathbf{S}$ that captures the structure of this graph, a **graph signal** can be defined as a vector $\mathbf{x} \in \mathbb{R}^N$ that assigns a value $x_i \in \mathbb{R}$ to every node i. In that way $\mathbf{S}$ encodes the expected proximity or similarity between components of $\mathbf{x}$. A multiplication $\mathbf{S}\mathbf{x}$ yields to a diffused signal $\mathbf{y}$ over $\mathcal{G}$:

$$\mathbf{y} = \mathbf{S}\,\mathbf{x} \tag{52}$$

If $\mathbf{S}$ is the adjacency matrix that would yield to $y_i = \sum_j w_{ij} x_j$. The operation mixes the neighboring values of x_i and diffuses its signal along the edges over $\mathcal{G}$. Since one application of $\mathbf{S}$ moves the signal of $\mathbf{x}$ only one edge at a time, a **diffusion sequence** over k steps can be defined as

$$\mathbf{x}_{k+1} = \mathbf{S}\,\mathbf{x}_k \quad \text{with} \quad \mathbf{x}_0 = \mathbf{x} \tag{53}$$

or equivalently in form of a power sequence

$$\mathbf{x}_k = \mathbf{S}^k\,\mathbf{x}. \tag{54}$$

(53) should be implemented in practice since leads to faster processing. For the following description of Graph Convolution Filters (54) is used.

10.3 Graph Convolution

Graph convolution filters are the basic building block of a GCN. Given $\mathbf{S}$ and filter coefficients h_k, a graph convolution is a polynomial on $\mathbf{S}$

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{\infty} h_k\,\mathbf{S}^k \tag{55}$$

The result of applying the filter $\mathbf{H}(\mathbf{S})$ to the signal $\mathbf{x}$ is the signal $\mathbf{y} = \mathbf{H}(\mathbf{S})\,\mathbf{x} = \sum_{k=0}^{\infty} h_k\,\mathbf{S}^k\,\mathbf{x}$. As a short hand notation this is written as $\mathbf{y} = \mathbf{h} \star_{\mathbf{S}} \mathbf{x}$, where $\star_{\mathbf{S}}$ denotes the **graph convolution** operation of filter $\mathbf{h} = \{h_k\}_{k=0}^{\infty}$ on signal $\mathbf{x}$ and graph shift operator $\mathbf{S}$. The filter coefficient h_k determines the importance of a particular diffusion state $k = 0, \infty$ that is multiplied by $\mathbf{x}$. Since in practice only a a finite amount of diffusion states $K - 1$ is observed, it yields that

$$\mathbf{y} = \mathbf{h} \star_{\mathbf{S}} \mathbf{x} = h_0\,\mathbf{S}^0\,\mathbf{x} + h_1\,\mathbf{S}^1\,\mathbf{x} + \cdots + h_{K-1}\,\mathbf{S}^{K-1}\,\mathbf{x} = \sum_{k=0}^{K-1} h_k\,\mathbf{S}^k\,\mathbf{x}. \tag{56}$$

The convolution therefore successively aggregates information from local to global neighborhoods. This is done by a linear combination of the elements of the diffusion sequence $\mathbf{S}^k\,\mathbf{x}$ weighted by the filter coefficients $\mathbf{h}$. Fig. 157 shows an exemplary application of a graph convolution.

Time Convolutions as Graph Convolutions. Time can be represented in form of a 1D line graph of infinite length, where an input signal is shifted infinite many times. A resulting graph shift operator could therefore be an infinitely big adjacency matrix $\mathbf{A}$ with ones only one left the diagonal. Now, given an input signal $\mathbf{x}$ and $\mathbf{S} = \mathbf{A}$, time convolution can be thought of as a polynomial on the adjacency matrix of the line graph: $\mathbf{y} = \sum_{k=0}^{\infty} h_k\,\mathbf{S}^k\,\mathbf{x}$.

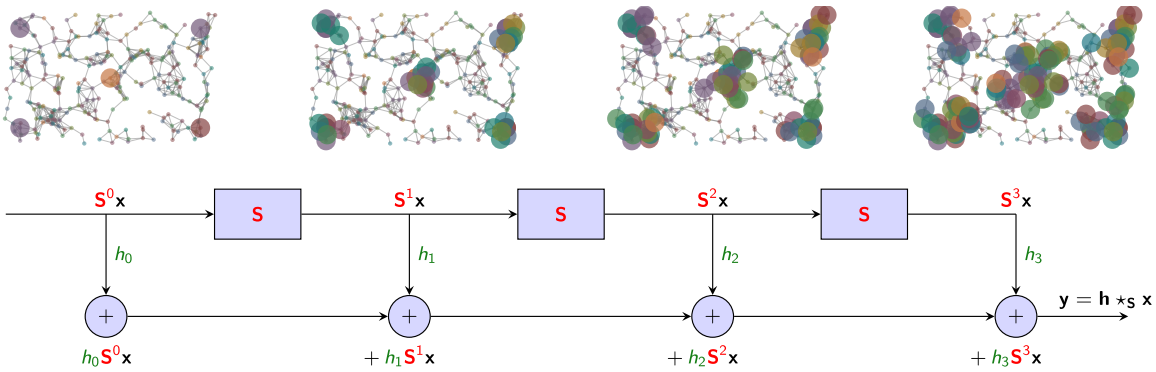


Figure 157: **Graph convolutions as diffusion operators.** A graph convolution can be visualized in a block diagram. Iteratively a shift of $\mathbf{S}$, a weighting of $\mathbf{S}^k\mathbf{x}$ with h_k and a summation of the weighted terms is applied.

Graph Fourier Transforms. Graph Fourier transforms (GFT) are equivalent to standard Fourier transforms in the regular domain but on graphs. Given the eigendecomposition of the graph shift operator $\mathbf{S} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^*$, the GFT of a graph signal is given by

$$\tilde{\mathbf{x}} = \mathbf{V}^*\mathbf{x}. \tag{57}$$

The GFT is thus a projection onto the eigenspace of $\mathbf{S}$. The GFT is a helpful tool for analyzing graph information processing systems, since in the GFT domain, graph convolutions are only pointwise operations $\tilde{y}_i = \sum_{k=0}^{\infty} h_k \lambda_i^k \tilde{x}_i$. Further information on GFTs can be found at https://GCN.seas.upenn.edu/.

10.4 Graph Convolution Networks

Learning Graph Filters. After knowing what graph convolutions are, an algorithm for learning the filter coefficients $\mathbf{h}$ of this graph convolution operation can be defined. This is equivalent to training a GCN with a single layer only. Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i, \mathbf{S}_i)\}$ denote a dataset of input signals $\mathbf{x}_i$, output signals $\mathbf{y}_i$ and graph shift operators $\mathbf{S}_i$ of a graph. It is important to note that a dataset can contain *various* graph shift operators $\mathbf{S}_i$. Thus, a single graph filter can be trained on multiple graphs in order to become more robust during deployment even for unseen graph representations. The graph convolution (56) is denoted by $f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$. A prediction of an output signal $\hat{\mathbf{y}}$ can be calculated using the following equation:

$$\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = \sum_{k=0}^{K-1} h_k \, \mathbf{S}^k \, \mathbf{x}. \tag{58}$$

During the training a loss $\mathcal{L}$ is minimized between the predicted output signal $\hat{y} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S})$ and the original output signal y in order to find the optimal values for the filter coefficients $\mathbf{h}$:

$$\mathbf{h}^* = \underset{\mathbf{h}}{\operatorname{argmin}} \sum_{(\mathbf{x},\mathbf{y},\mathbf{S}) \in \mathcal{D}} \mathcal{L}\left(f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}), \mathbf{y}\right) \tag{59}$$

If the inference problem at hand has a distinct input and output dimension, a **readout layer** is needed. Let N denote the number of vertices in our graph, thus the input dimension of $\mathbf{x}^N$ and M the output dimension of $\hat{\mathbf{y}}^M$ with $N \neq M$. The readout layer is defined as:

$$\hat{\mathbf{y}} = f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = \mathbf{R} \sum_{k=0}^{K-1} h_k \, \mathbf{S}^k \, \mathbf{x} \tag{60}$$

with the readout matrix $\mathbf{R} \in \mathbb{R}^{M \times N}$ that matches the input and output dimension. Typically, $\mathbf{R}$ is not learned but a design choice of the programmer. $\mathbf{R}$ could be such that it reads out only the value of node i by using a unit vector at location i $\mathbf{R} = \mathbf{e}_i^\top$. Another exemplary readout matrix could be the summation over all N elements of all the graph $\mathbf{R} = \mathbf{1}^\top$. This is useful for classification purposes.

Graph Perceptron. The graph filters as defined above have limited expressive power as they can only learn linear mappings. In order to achieve a higher expressive power, the graph filters are combined with pointwise non-linearities $g(\cdot)$, such as sigmoid, tanh or ReLU activation functions. This function is named **graph perceptron**, since it introduces the same features for GCN as the perceptron for general neural networks. The graph perceptron can be expressed as:

$$f_{\mathbf{h}}(\mathbf{x}, \mathbf{S}) = g\left(\sum_{k=0}^{K-1} h_k \, \mathbf{S}^k \, \mathbf{x}\right). \tag{61}$$

Because of the introduced non-linearity, the graph perceptron is able to express a larger function class.

Graph Convolution Networks. A GCN can now be defined by stacking multiple layers of graph perceptrons on top of each other. Similar to a MLP, the GCN is recursively composed

$$\mathbf{x}_\ell = g\left(\sum_{k=0}^{K-1} h_{\ell k}\, \mathbf{S}^k\, \mathbf{x}_{\ell-1}\right), \tag{62}$$

where it is assumed that the input to the first layer is set to the input signal $\mathbf{x}_0 = \mathbf{x}$. Note that compared to (61) $\mathbf{x}$ and h_k have index ℓ now to denote the layer. A short hand notation for the recursive application of the graph perceptron over L layers is:

$$\hat{\mathbf{y}} = f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}) = \mathbf{x}_L, \tag{63}$$

where $\mathcal{H}$ denotes a set of L vectors of trainable filter coefficients $\mathcal{H} = \{\mathbf{h}_1, \ldots, \mathbf{h}_L\}$. The recursive composition of a GCN is illustrated in Fig. 161. Empirical evidence suggests that, similar to CNNs deeper GCNs have a better prediction performance due to more layers of non-linearities.

Learning Graph Convolution Networks. The filter parameters of a GCN are learned in same way as in the single-layer-case but with the filter coefficients of all layers $\mathcal{H}$ as the optimization objective:

$$\mathcal{H}^* = \underset{\mathcal{H}}{\operatorname{argmin}} \sum_{(\mathbf{x},\mathbf{y},\mathbf{S})\in\mathcal{D}} \mathcal{L}\left(f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}), \mathbf{y}\right). \tag{64}$$

Comparing a Multi-Layer Perceptron and a Graph Convolution Network. A comparison between a MLP and a GCN shows that the GCN can be regarded as a special case of a MLP. This becomes clear when looking at the trainable parameters of both network architectures. In (62) the learnable parameter in each layer is the vector $\mathbf{h}_\ell$, whereas the MLP with $\mathbf{x}_\ell = g\left(\mathbf{W}_\ell\, \mathbf{x}_{\ell-1}\right)$ has a weight matrix $\mathbf{W}_\ell$. Under the assumption that $\mathbf{S}$ is constant in the dataset, it is obvious that the MLP is more flexible during training due to having more parameters and thus attains lower training cost:

$$\min_{\mathcal{W}} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \mathcal{L}\left(f_{\mathcal{W}}(\mathbf{x}), \mathbf{y}\right) \leq \min_{\mathcal{H}} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \mathcal{L}\left(f_{\mathcal{H}}(\mathbf{x}, \mathbf{S}), \mathbf{y}\right) \tag{65}$$

Although GCNs are a special case of MLPs, their generalization capabilities to unseen input signals are better. That is because GCNs are able to exploit *a priori* information about the symmetries of a graph that are carried with the graph shift operator $\mathbf{S}$. GCNs learn the structure of a graph as seen in Fig. 158. Furthermore, unlike MLPs, GCNs can be trained on different graphs by passing different graph shift operators to them.

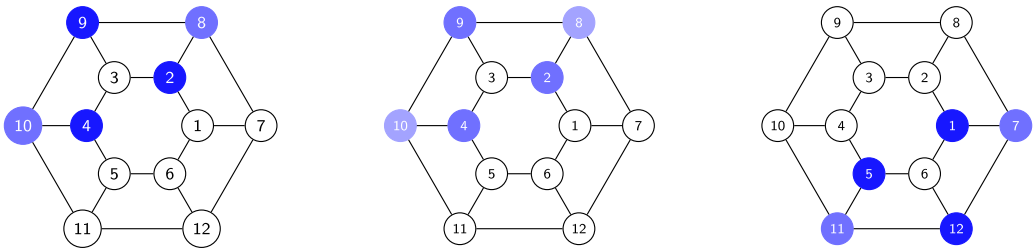


Figure 158: **Comparing the generalization of MLPs and GCNs.** The three images depict an output signal on a graph, where the color indicates the signal strength. An MLP and a GCN receive the same input and output pairs during training and produce a similar output signal (left). During inference time an unseen signal is passed into the model and the MLP predicts an output signal similar to the predictions during training (center). The GCN in contrast generalizes better since it knows the underlying structure of the graph. Therefore it is able to predict an output signal that has a structure similar to the structures predicted during training but on another part of the graph (right). This is due to the translation equivariance of such a network.

Multiple-Input-Multiple-Output GCNs. The definition of a standard GCN in (62) is limited to an input vector $\mathbf{x}$. Thus, for each vertex of the graph, only a single scalar value can be used as an input and as an intermediate representation. In a Multiple-Input-Multiple-Output GCN (MIMO GCN) this restriction is eased by introducing filter banks:

Filter banks take a graph signal and output a graph signal matrix $\mathbf{Z} = (\mathbf{z}^1, \ldots, \mathbf{z}^G)$, where Z represents G features per node. This is achieved by transforming a graph signal with G convolution filters separately (see Fig. 159).

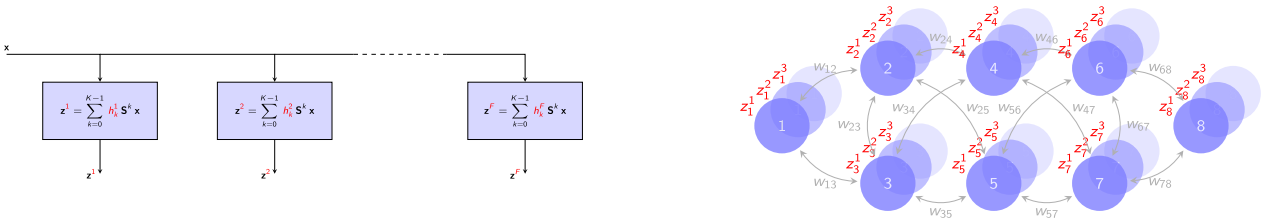


Figure 159: **Filter banks.** The block diagram on the left receives a signal $\mathbf{x}$ of the graph and applies this signal separately with 3 different graph filters ($G = 3$). This produces a graph signal matrix $\mathbf{Z} = (\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3$ with each vecotr containing one feature per node (right).

If now the input to a MIMO GCN is multidimensional with F features per node, the feature vector x^f is processed through G filters weighted by coefficients h_k^{fg}:

$$\mathbf{u}^{fg} = \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \mathbf{x}^f \tag{66}$$

The **MIMO Graph Filter**, thus generates an output with $F \times G$ features per node. Applying filter banks over multiple layers would therefore lead to an unwanted exponential growth of features and parameters. For this reason, in each layer the output of each graph filter G is summed up. This reduces the number of features to G (see Fig. 160:

$$\mathbf{z}^g = \sum_{f=1}^{F} \mathbf{u}^{fg} = \sum_{f=1}^{F} \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \mathbf{x}^f \tag{67}$$

In matrix notation the MIMO graph filter can simply be expressed as

$$\mathbf{z}^g = \sum_{f=1}^{F} \sum_{k=0}^{K-1} h_k^{fg} \mathbf{S}^k \mathbf{x}^f \quad \Leftrightarrow \quad \mathbf{Z} = \sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{X} \mathbf{H}_k, \tag{68}$$

where

$$\underbrace{\begin{pmatrix} \mathbf{z}^1 & \cdots & \mathbf{z}^G \end{pmatrix}}_{=\mathbf{Z} \in \mathbb{R}^{N \times G}} = \sum_{k=0}^{K-1} \underbrace{\mathbf{S}^k}_{\in \mathbb{R}^{N \times N}} \underbrace{\begin{pmatrix} \mathbf{x}^1 & \cdots & \mathbf{x}^F \end{pmatrix}}_{=\mathbf{X} \in \mathbb{R}^{N \times F}} \underbrace{\begin{pmatrix} h_k^{11} & \cdots & h_k^{1G} \\ \vdots & & \vdots \\ h_k^{F1} & \cdots & h_k^{FG} \end{pmatrix}}_{=\mathbf{H}_k \in \mathbb{R}^{F \times G}}.$$

A **MIMO GCN** is now defined by stacking multiple MIMO Graph Filters on top of each other for L layers. This yields to the following equation:

$$\mathbf{X}_\ell = g\left(\sum_{k=0}^{K-1} \mathbf{S}^k \mathbf{X}_{\ell-1} \mathbf{H}_{\ell k} \right) \tag{69}$$

with $\mathbf{X}_0 = \mathbf{X}$ being the input signal matrix to the first layer. (69) is expressed as a short-hand notation as:

$$\hat{\mathbf{Y}} = f_{\mathcal{H}}(\mathbf{X}, \mathbf{S}) = \mathbf{X}_L \tag{70}$$

with parameters $\mathcal{H} = \{\mathbf{H}_1, \ldots, \mathbf{H}_L\}$, where $\mathbf{H}_\ell = \{\mathbf{H}_{\ell 0}, \ldots, \mathbf{H}_{\ell, K-1}\}$.
Compared to the standard GCN the MIMO GCN is handling matrices as an input and output (see Fig. 162).

11 Autoencoders

11.1 Latent Variable Models

Latent variable models capture the structure of the data space using *latent* variables that can be modelled using unsupervised learning without the need for data-label pairs. Formally, a latent variable model relates the observation space $\mathbf{x} \in \mathbb{R}^D$ and the latent space $\mathbf{z} \in \mathbb{R}^Q$ with $N << Q$. The mapping can be $f_w : \mathbf{x} \to \mathbf{z}$ referred as *encoder* or $g_w : \mathbf{z} \to \mathbf{x}$ referred as *decoder*. Latent variable models always have a decoder but may or may not have an encoder. Furthermore, the mapping can either be linear or non-linear and deterministic or probabilistic. Table 2 cites one example for each of mapping types. Latent variable models capture the underlying structure and semantics of the data manifold usually in a lower dimensional space.

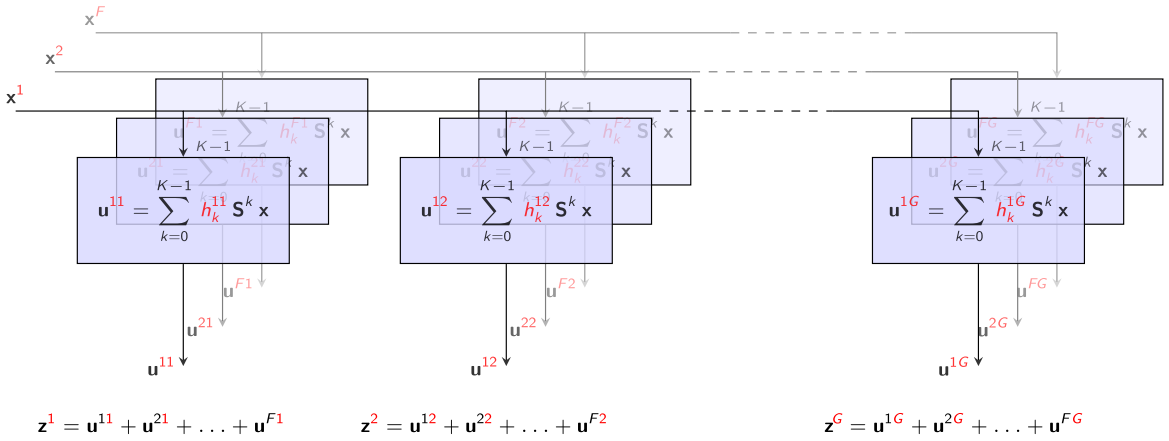


Figure 160: **Filter Banks with multiple inputs.** Each of the F feature vectors $\mathbf{x}^f$ is applied to G graph filters and produces the outputs u^{fg} that are then summed up to z^g.

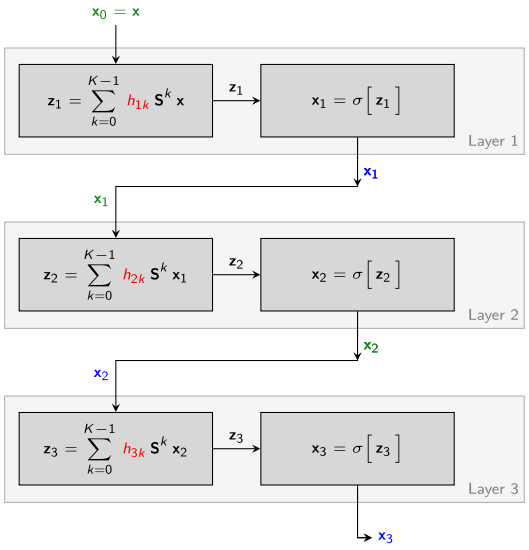


Figure 161: **Block diagram of a GCN.** The input vector $\mathbf{x}_0$ is passed into the first graph perceptron in the first layer $\ell = 1$. The output x_1 is equally passed into layer $\ell = \ell + 1$ until the final output x_3 is is produced.

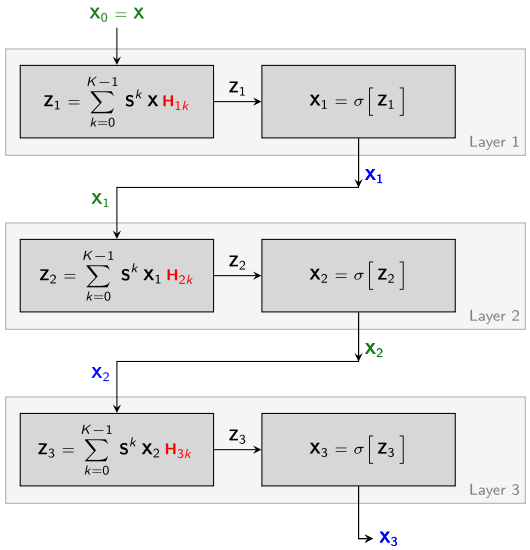


Figure 162: **Block diagram of a MIMO GCN.** In contrast to the GCN, the MIMO GCN receives an input matrix $\mathbf{X}$ and produces with the parameter coefficient matrix $\mathbf{H}$ the output matrix $\mathbf{Z}$ containing the values from the filter bank.

11.1.1 Generative Latent Variable Models

Broadly, generative models tries to capture the underlying data distribution $p(\mathbf{x})$ from which the data points $\{\mathbf{x}_i\}_{i=1}^N$ are supposedly sampled from. They are primarily used to sample random instances from the modelled distribution. Generative latent variable models capture the structure and semantics of the data using latent variables. Generally, they employ a *Bayesian* model and the data distribution $p(\mathbf{x})$ is defined as:

$$p(\mathbf{x}) = \int_z p(\mathbf{z})p(\mathbf{x}|\mathbf{z}) = \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[p(\mathbf{x}|\mathbf{z})],$$

where $p(\mathbf{z})$ is the **prior** probability over the latent variable $\mathbf{z} \in \mathbb{R}^Q$ and $p(\mathbf{x}|\mathbf{z})$ represents the likelihood of $\mathbf{x}$ given $\mathbf{z}$. Given a dataset $\mathcal{X}$, generative latent variable models aim to maximize $p(\mathbf{x})$ where $\mathbf{x} \in \mathcal{X}$ by learning a prior $p(\mathbf{z})$ and the likelihood $p(\mathbf{x}|\mathbf{z})$. Each ata point $\mathbf{x}$ is associated to a unique latent variable $\mathbf{z}$. Fig. 163 shows graphical representation of generative latent variable model.

	Deterministic	Probabilistic
Linear	Principle Component Analysis(PCA)	Probablistic PCA
Non-Linear with Encoder	Autoencoder	Variational Autoencoder
Non-Linear without Encoder		Generative Adversarial Networks

Table 2: **Taxonomy of latent variable models**

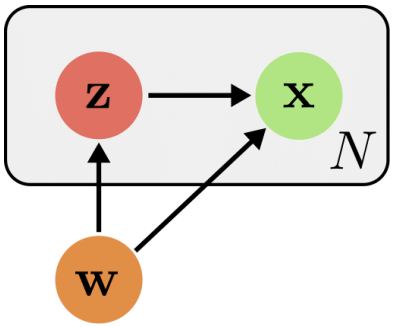


Figure 163: **Plate notation of generative latent variable models**. The model parameters **w** are fixed for all the **N** data points. However, each input is associated with one distinct latent variable

11.2 Principal Component Analysis

Standard principal component analysis(PCA) is a deterministic and linear latent variable model that is primarily used in data analysis and as a dimensionality reduction technique. PCA learns a *bidirectional* linear mapping between the input space $\mathbf{x}$ and the latent space $\mathbf{z}$. Given a dataset $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)^T \in \mathbb{R}^{N \times D}$, PCA assumes a linear mapping between data points $\mathbf{X}$ and the corresponding latent variables $\mathbf{Z} = (\mathbf{z}_1, \ldots, \mathbf{z}_N) \in \mathbb{R}^{N \times Q}$ as follows:

$$\hat{\mathbf{x}}_i = \overline{\mathbf{x}} + \sum_{j=1}^{Q} z_{ij} \mathbf{v}_j$$

where $\hat{x}_i$ is the reconstruction/prediction of the data point $\mathbf{x}_i$ and $\overline{x} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i$ is **data mean**, $V = (\mathbf{v}_1, \ldots, \mathbf{v}_Q)$ is an **orthonormal** basis of the latent space. Essentially, PCA constitutes a **decoder** given by $\mathbf{x} = \overline{\mathbf{x}} + \mathbf{V}\mathbf{z}$ (obtained by rewriting the above equation in matrix form) and an **encoder** $\mathbf{z} = \mathbf{V}^T(\mathbf{x} - \overline{\mathbf{x}})$ obtained by inverting the above linear mapping. The goal of PCA is to minimize the L_2 reconstruction loss wrt latent variables $\mathbf{Z}$ and the orthogonal basis $\mathbf{V}$. We formalize the reconstruction loss below:

$$\begin{aligned}
\mathcal{L}(\mathbf{Z}, \mathbf{V}) = \sum_{i=1}^{N} \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|^2 &= \sum_{i=1}^{N} \|\underbrace{\overline{\mathbf{x}} + \sum_{j=1}^{Q} z_{ij} \mathbf{v}_j}_{\hat{\mathbf{x}}_i} - \mathbf{x}_i\|^2 \\
&= \sum_{i=1}^{N} \|\sum_{j=1}^{Q} z_{ij} \mathbf{v}_j + \overline{\mathbf{x}} - \mathbf{x}_i\|^2 \\
&= \sum_{i=1}^{N} \left[\sum_{j=1}^{Q} z_{ij}^2 + 2 \sum_{j=1}^{Q} z_{ij} \mathbf{v}_j^T (\overline{\mathbf{x}} - \mathbf{x}_i) + \|\overline{\mathbf{x}} - \mathbf{x}_i\|^2 \right]
\end{aligned}$$

The **optimization** objective of PCA is given by

$$(\mathbf{Z}^*, \mathbf{V}^*) = \underset{\mathbf{Z}, \mathbf{V}}{\operatorname{argmin}} \ \mathcal{L}(\mathbf{Z}, \mathbf{V}),$$

11.2.1 Solving PCA: Reconstruction Error Minimization

One can obtain the minimizers $(\mathbf{Z}^*, \mathbf{V}^*)$ of the reconstruction loss in *closed-form*. We first solve for the latent variables $\mathbf{Z}$ by setting the partials of the reconstruction loss wrt $\mathbf{Z}$ to zero:

$$\frac{\partial \mathcal{L}(\mathbf{Z}, \mathbf{V})}{\partial \mathbf{z}_{ih}} = 2z_{ij} + 2\mathbf{v}_j^T(\overline{\mathbf{x}} - \mathbf{x}_i) \stackrel{!}{=} 0$$
$$\implies \boxed{z_{ij}^* = -\mathbf{v}_j^T(\overline{\mathbf{x}} - \mathbf{x}_i)}$$

We plugin the solution for $\mathbf{Z} = \mathbf{Z}^*$ into the reconstruction loss $\mathcal{L}(\mathbf{Z}, \mathbf{V})$ which can be simplified and rewritten as

$$\begin{aligned}\mathcal{L}(\mathbf{Z}^*, \mathbf{V}) &= \sum_{i=1}^{N}\left[-\sum_{j=1}^{Q} z_{ij}^{*2} + \|\overline{\mathbf{x}} - \mathbf{x}_i\|^2\right] \\ &= -\sum_{j=1}^{Q}\mathbf{v}_j^T\mathbf{S}\mathbf{v}_j + \sum_{i=1}^{N}\|\overline{\mathbf{x}} - \mathbf{x}_i\|^2\end{aligned}$$

where $\mathbf{S} = \sum_{i=1}^{N}(\overline{\mathbf{x}} - \mathbf{x}_i)(\overline{\mathbf{x}} - \mathbf{x}_i)^T$ is the **scatter matrix** of the dataset $\mathbf{X}$.
We then proceed to find the orthogonal basis $\mathbf{V}$ using the simplified form. Since, $\mathbf{V}$ is *constrained* to be the orthonormal basis, the solution $\mathbf{V}^*$ that minimizes $\mathcal{L}(\mathbf{Z}^*, \mathbf{V})$ is subjected to an *equality* constraint $\mathbf{v}_j^T\mathbf{v}_j = 1$ for all $j \in 1, \ldots, Q$. We employ **Lagrange multiplier** to solve for $\mathbf{V}$ and the Lagrange expression for the same is given by:

$$\mathcal{L}(\mathbf{Z}^*, \mathbf{V}, \boldsymbol{\lambda}) = -\sum_{j=1}^{Q}\mathbf{v}_j^T\mathbf{S}\mathbf{v}_j + \sum_{i=1}^{N}\|\overline{\mathbf{x}} - \mathbf{x}_i\|^2 + \sum_{j=1}^{Q}\lambda_j(\mathbf{v}_j^T\mathbf{v}_j - 1)$$

We find $\mathbf{V}^*$ by setting the partials of the above Lagrange expression wrt $\mathbf{V}$ to zero

$$\frac{\partial \mathcal{L}(\mathbf{Z}^*, \mathbf{V}, \boldsymbol{\lambda})}{\partial \mathbf{v}_j} = -2\mathbf{S}\mathbf{v}_j + 2\lambda_j\mathbf{v}_j \stackrel{!}{=} 0$$
$$\implies \boxed{\mathbf{S}\mathbf{v}_j = \lambda_j\mathbf{v}_j}$$

The possible solutions for $(\boldsymbol{\lambda}, \mathbf{V})$ are the eigen values and vectors of the scatter matrix $\mathbf{S} \in \mathbb{R}^{D\times D}$. There can exist upto D eigen vectors. For latent space with dimensions $\mathbf{Q} < \mathbf{D}$ we choose the eigen vectors with **top** Q largest eigen values. This ensure the reconstruction loss is minimized as the loss is proportional to the sum of pruned eigen values. $\sum_{j=1}^{Q}\mathbf{v}_j^T\mathbf{S}\mathbf{v}_j = \sum_{j=1}^{Q}\lambda_j$.

11.2.2 Solving PCA: Latent Variance Maximization

: An alternative motivation for PCA is **maximizing** the variance in the latent space. Formally, the variance of latent variables can be written as,

$$\begin{aligned}Var(\mathbf{z}) &= \mathbb{E}\left[(\mathbf{v}^T(\mathbf{x} - \overline{\mathbf{x}}) - \mathbb{E}\left[\mathbf{v}^T(\mathbf{x} - \overline{\mathbf{x}})\right])^2\right] \\ &= \mathbb{E}\left[(\mathbf{v}^T(\mathbf{x} - \overline{\mathbf{x}})(\mathbf{x} - \overline{\mathbf{x}})^T\mathbf{v}\right] \\ &\propto \mathbf{v}^T\mathbf{S}\mathbf{v}\end{aligned}$$

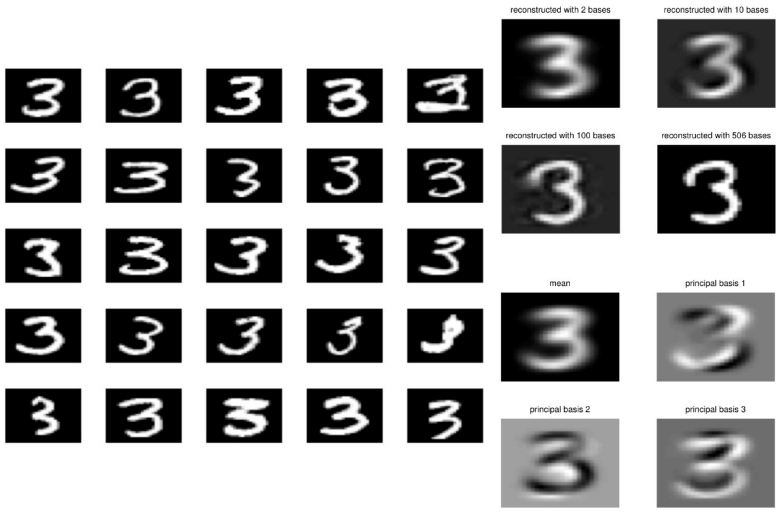
which shows maximizing the variance is proportional to maximizing the term $\mathbf{v}^T\mathbf{S}\mathbf{v}$. Thus, one can solve the below optimization objective wrt $\mathbf{V}$ for variance maximization of the latent variables where $\mathbf{V}$ is constrained to be a orthonormal basis like before.

$$(\mathbf{V}^*, \boldsymbol{\lambda}^*) = \underset{\mathbf{V},\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{j=1}^{Q}\mathbf{v}_j^T\mathbf{S}\mathbf{v}_j + \sum_{j=1}^{Q}\lambda_j(\mathbf{v}_j^T\mathbf{v}_j - 1)$$

The above objective can be maximized by the Q largest eigen values and their corresponding eigenvectors.

11.2.3 Applications

Despite being a linear model, PCA yields good reconstructions with very low dimensional latent spaces. Besides dimensionality reduction, PCA can also be used to explore the latent properties of the datasets. Fig. 164 showcases two applications of PCA on image manifold.



(a)

(b)

Figure 164: **Applications of PCA to Image ManiFold** a) Application of principal component analysis on MNIST digit dataset. PCA is applied on the images with digit 3. The reconstruction becomes better as the number of basis are increased. b) shows reconstruction of face images using only **3** eigen components achieving significant dimenstionality reduction.

11.3 Autoencoder

Autoencoders generally constitutes a deterministic non-linear encoder $f_{\mathbf{w}}$ and a decoder $g_{\mathbf{w}}$. They are usually learnt to predict the input $\mathbf{x} \in \mathbb{R}^D$ as output $\hat{\mathbf{x}} \in \mathbb{R}^D$. The encoder $f_{\mathbf{w}} : \mathbf{x} \to \mathbf{z}$ maps the input to a latent code $\mathbf{z} \in \mathbb{R}^Q$ while decoder $g_{\mathbf{w}} : \mathbf{z} \to \mathbf{x}$ tries to reconstructs the input back from the latent code $\mathbf{z}$. Hence, autoencoders are trained to minimize the reconstruction loss such as squared error, $\|\hat{\mathbf{x}} - \mathbf{x}\|^2$. Fig. 165 presents a schematic illustration of auto encoders.

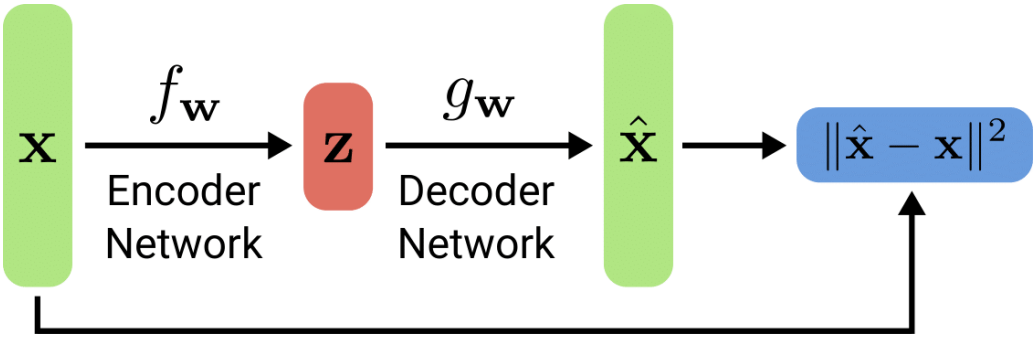


Figure 165: **Schema of an autoencoder**. Autoencoder takes the input $\mathbf{x}$ and encodes into the latent representation $\mathbf{z}$ using the encoder network $f_{\mathbf{w}}(\mathbf{x})$. The latent code $\mathbf{z}$ is then reconstructed back using the decoder network $g_{\mathbf{w}}(\mathbf{z})$ which outputs the prediction $\hat{\mathbf{x}}$. To learn the model parameters $\mathbf{w}$ squared reconstruction error is used.

The non-linear mappings $f_{\mathbf{w}}$ and $g_{\mathbf{w}}$ are parameterized by **neural networks** with weights $\mathbf{w}$. One can choose an appropriate class of the neural networks(Convolutional Neural Networks, Multi Layer Perceptron, etc.) based on the nature of the input data.

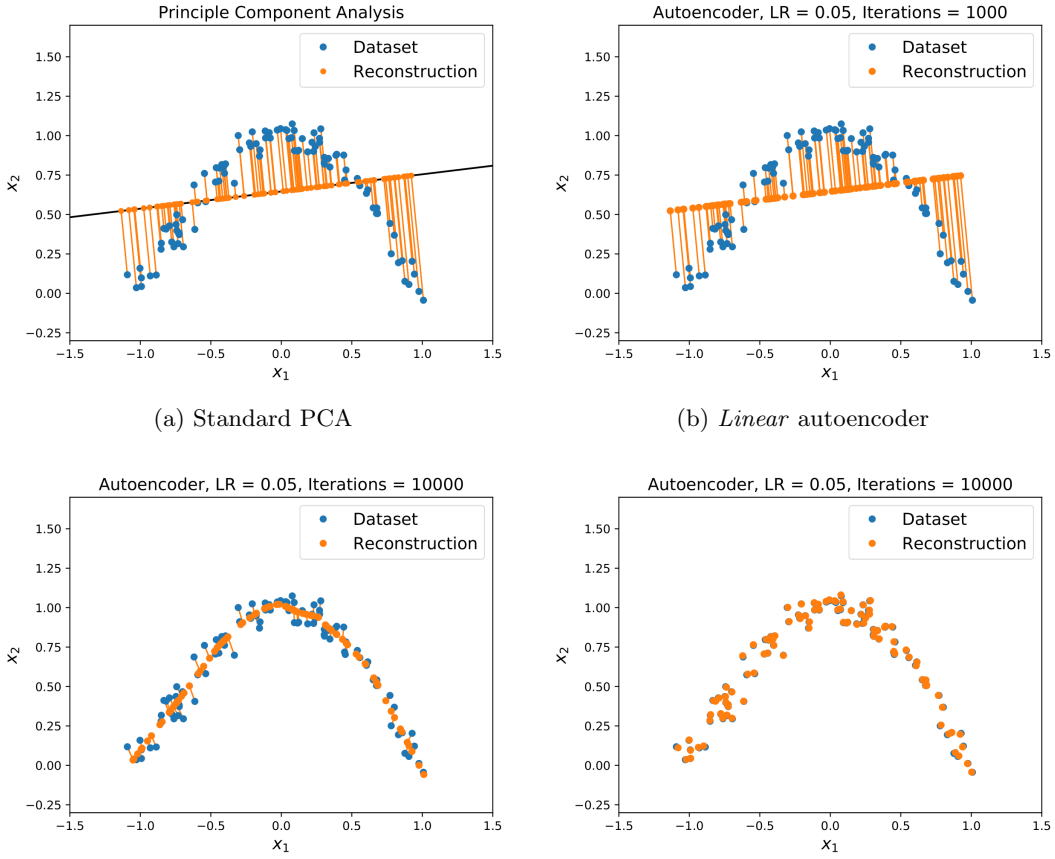
11.3.1 PCA:Special Case of Autoencoders

When the mappings $f_{\mathbf{w}}$ and $g_{\mathbf{w}}$ are assumed to be **linear** using **identity** activation functions, autoencoders interestingly reduce to standard PCA. The encoder is of form $f_w(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{a}$ which computes the latent variable $\mathbf{z}$ and the decoder is given by $g_w(\mathbf{z}) = \mathbf{B}\mathbf{z} + \mathbf{b}$ which outputs the reconstruction $\hat{\mathbf{x}}$ of input $\mathbf{x}$. Since,

the goal of autoencoders is to minimize the reconstruction error the optimization objective can be written as:

$$
\begin{aligned}
\mathbf{w}^* &= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \|g_{\mathbf{w}}(f_{\mathbf{w}}(\mathbf{x}_i)) - \mathbf{x}_i\|^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \|(\mathbf{B}(\mathbf{A}\mathbf{x}_i + \mathbf{a}) + \mathbf{b}) - \mathbf{x}_i\|^2 \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \|(\underbrace{\mathbf{C}\mathbf{x}_i + \mathbf{c}}_{\hat{\mathbf{x}}_i}) - \mathbf{x}_i\|^2
\end{aligned}
$$

The optimal weights $\mathbf{w}^*$ can be obtained by PCA as we have seen before.
Autoencoders can learn non-linear mappings that makes them powerful and often have small reconstruction errors compared to PCA. In Fig. 166 we juxtapose the reconstructions of PCA and non-linear autoencoders which suggests that autoencoders easily achieve very small reconstruction errors.



(a) Standard PCA

(b) *Linear* autoencoder

(c) Autoencoder with 1D latent space

(d) Autoencoder with 2D latent space

Figure 166: **Comparison of PCA and autoencoder on cosine manifold**a) PCA applied on cosine data with 1 dimensional latent space. b) Linear autoencoder with same latent space as PCA. It is interesting to see that after sufficient number of iterations the reconstructions of PCA and linear autoencoder are almost indistinguishable. c) Non-linear autoencoder with 1 dimensional latent space. With the non-linear functions autoencoder produces much better reconstructions than standard PCA. d) With increase in the latent space the reconstruction completely overlaps with the data points.

11.3.2 Denoising Autoencoder

Denoising autoencoders take noisy input data and learns to reconstruct the original undistorted input. In this way the model learns to stable high level representations and increases its robustness to input corruptions.

11.4 Variational Auto Encoders(VAE)

11.4.1 Intractability of Learning Generative Latent Variable Models

Generative latent variable models disccussed in Section11.1.1 essentially captures the underlying data distribution $p(\mathbf{x})$ by considering a simple Bayesian model with prior probability $p(\mathbf{z})$ over the latent space and conditional probability $p(\mathbf{x}|\mathbf{z})$ over the input space given $\mathbf{z}$. However, learning optimal **parameterized** distributions is often intractable. Consider $p_{\mathbf{w}}(x)$ to be a distribution parameterized by some weights $\mathbf{w}$. The goal of generative latent variable models is to maximize $p(\mathbf{x})$ for the data points $\mathbf{x} \in \mathcal{X}$ in our dataset. To find the *optimal* weights $\mathbf{w}^*$ that maximizes $p(\mathbf{x})$ one can use the familiar minimization objective stated below:

$$\begin{aligned}
\mathbf{w}^* &= \underset{\mathbf{w}}{\operatorname{argmin}} \ \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log p(x)\right] \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \ \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{w}}(\mathbf{z})}[p_{\mathbf{w}}(\mathbf{x}|\mathbf{z})]\right] \\
&= \underset{\mathbf{w}}{\operatorname{argmin}} \ \frac{1}{N}\sum_{i=1}^{N} -\log \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{w}}(\mathbf{z})}[p_{\mathbf{w}}(\mathbf{x}_i|\mathbf{z})]
\end{aligned}$$

The objective requires computing expectation over the latent variable $\mathbf{z}$ with distribution $p(\mathbf{z})$. One is forced to draw **huge** number of samples from $p(\mathbf{z})$ to approximate the expectation and this number exponentially grows with increase in latent dimensions making computation of the objective intractable.

11.4.2 Variational Auto encoders

Variational Auto encoders tackles the above intractability issue by **approximating** the posterior distribution $p_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$ with a so called **recognition model** $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$. Using this **approximation** one can seek a **tractable lower bound** for the marginal distribution $p_{\mathbf{w}}(\mathbf{x})$ and optimize the lower bound to in turn maximize the marginal distribution. We discuss the lower bound in detail in the next section.
To provide an intuitive explanation for recognition model $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$ alleviating the intractability problem, consider an observation space to be sound waves and latent space to be the word sequences. We wish to maximize the likelihood of the sound waves in our dataset. Given a song(sound waves) from our dataset it might be highly difficult to infer the lyrics(word sequences) from a vast space of words sequences. However, it becomes easier to realize the song when someone can provide us some approximate of what the lyrics could be for the given song. Similarly, the recognition model $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$ provides an *approximation* of the true posterior $p(\mathbf{z}|\mathbf{x})$. This helps in restricting the search space from which we the sample latent variable $\mathbf{z}$ compared to the prior probability $p_{\mathbf{w}}(\mathbf{z})$. Fig. 167 pictorially showcases an example where drawing three samples from prior probability $p_{\mathbf{w}}(\mathbf{z})$ assigns near-zero probability to the datapoint $\mathbf{x}_i$ while with an approximate recognition model $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$ realized significant probability to the input.

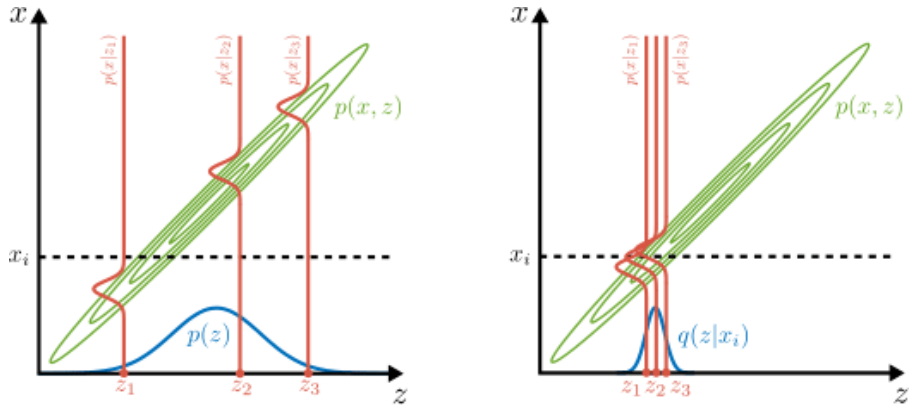


Figure 167: Illustration of intractability with prior probability $p(\mathbf{z})$. We wish to estimate the probability of $p(\mathbf{x}_i)$ by drawing three samples from $p(\mathbf{z})$ For all the three samples $p(\mathbf{x}_i|\mathbf{z})$ is almost zero leading to draw more samples from $p(\mathbf{z})$. However, using the recognition model $q(\mathbf{z}|\mathbf{x}_i)$ the search space for $\mathbf{z}$ is reduced and thereby leading to significant estimate of $p(\mathbf{x}_i)$ with just three samples.

11.4.3 The Evidence Lower Bound

The log likelihood $p_{\mathbf{w}}(\mathbf{x})$ can be rewritten as follows using the recognition model $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$. We drop the parameter $\mathbf{w}$ in below equations as the bound holds in general.

$$\begin{aligned}\log p(x) &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{x})p(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \\ &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] + \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})}\right] \\ &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right] + \underbrace{\mathcal{KL}(q(\mathbf{z}|\mathbf{x}), p(\mathbf{z}|\mathbf{x}))}_{\geq 0}\end{aligned}$$

Note that the above reformulation maintains the equality. Furthermore, the $\mathcal{KL}$ divergence term in the above equation is intractable as it relies on the true posterior $p(\mathbf{z}|\mathbf{x})$. However, $\mathcal{KL}$ divergence is **non-negative** distance measure. Thus, we eliminate the intractable $\mathcal{KL}$ divergence and obtain a lower bound for the log likelihood defined by:

$$\log p(x) \geq \underbrace{\boxed{\mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{p(\mathbf{x},\mathbf{z})}{q(\mathbf{z}|\mathbf{x})}\right]}}_{ELBO} \implies -\log p(x) \leq \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{x},\mathbf{z})}\right]$$

Alternatively, one can *upper bound* the **negative** log likelihood with the *negative* ELBO. For better intrepretation we expand the *negative* ELBO into:

$$\begin{aligned}\mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{x},\mathbf{z})}\right] &= \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log \frac{q(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} - \log(p(\mathbf{x}|\mathbf{z}))\right] \\ &= \mathcal{KL}(q(\mathbf{z}|\mathbf{x}), p(\mathbf{z})) + \mathbb{E}_{\mathbf{z}\sim q(\mathbf{z}|\mathbf{x})}\left[\log p(\mathbf{x}|\mathbf{z})\right]\end{aligned}$$

One can interpret the above bound as enforcing $\mathcal{KL}$ measure to inhibit the divergence of the approximate **recognition** model term from the prior distribution $p(\mathbf{x})$ while the other term measures the reconstruction error wrt to the input $\mathbf{x}$.

11.4.4 Learning Objective

Given a dataset $\mathcal{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ and $\mathbf{w}$ the model parameters, VAE minimizes the negative log likelihood objective as follows:

$$\begin{aligned}\mathbf{w}^* &= \underset{\mathbf{w}}{\text{argmin}}\ \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log p_{\mathbf{w}}(\mathbf{x})\right] \\ &= \underset{\mathbf{w}}{\text{argmin}}\ \sum_{i=1}^{N}\left[-\log p_{\mathbf{w}}(\mathbf{x}_i)\right] \\ &\approx \underset{\mathbf{w}}{\text{argmin}}\ \sum_{i=1}^{N}\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{w}}(\mathbf{z}|\mathbf{x}_i)}\left[\log \frac{q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})}{p_{\mathbf{w}}(\mathbf{x}_i,\mathbf{z})}\right] && \text{ELBO approximation} \\ &= \underset{\mathbf{w}}{\text{argmin}}\ \sum_{i=1}^{N}\underbrace{\mathcal{KL}(q_{\mathbf{w}}(\mathbf{z}|\mathbf{x}_i), p(\mathbf{z}))}_{\text{Approx Posterior = Prior}} + \underbrace{\mathbb{E}_{\mathbf{z}\sim q_{\mathbf{w}}(\mathbf{z}|\mathbf{x}_i)}\left[-\log p_{\mathbf{w}}(\mathbf{x}_i|\mathbf{z})\right]}_{\text{Reconstruction Term}} && \text{By expansion}\end{aligned}$$

11.4.5 Neural Network Parameterization

Essentially VAEs comprises of a *recognition* model $q_{\mathbf{w}}(\mathbf{z}|\mathbf{x})$ that approximates the *true* posterior $p(\mathbf{z}|\mathbf{x})$ and a *likelihood* model $p_{\mathbf{w}}(\mathbf{x}|\mathbf{z})$ To ensure tractability of the $\mathcal{KL}$ divergence term in the learning objective, the recognition model is chosen to be a **multi-variate Gaussian distribution** parameterized by the neural network. However, the prior of the latent space is generally as standard Gaussian distribution. Formally, the Gaussian recognition model can be expressed as:

$$q_{\mathbf{w}}(\mathbf{z}|\mathbf{x}) = \frac{1}{(2\pi)^{Q/2}}\frac{1}{|\boldsymbol{\Sigma}_{\mathbf{w}}(\mathbf{x})|^{1/2}}\exp\left(-\frac{1}{2}(\mathbf{z}-\boldsymbol{\mu}_{\mathbf{w}}(\mathbf{x}))^T\boldsymbol{\Sigma}_{\mathbf{w}}(\mathbf{x})^{-1}(\mathbf{z}-\boldsymbol{\mu}_{\mathbf{w}}(\mathbf{x}))\right)$$

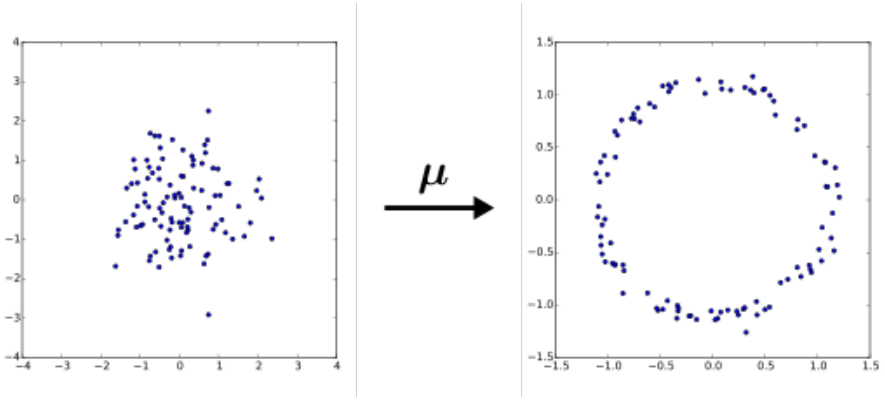


Figure 168: In this example the random variable $\mathbf{z}$ on the left takes a standard Gaussian distribution. However, by finding an approximate function the random variable can evoke a different distribution. Here when $g(\mathbf{z}) = \mathbf{z}/10 + z/\|z\|$ the normally distributed random variable can generate the distribution on the right using $g(\mathbf{z})$. This is how VAEs are expressive despite simple prior probability distribution.

where $\mathbf{\Sigma}_\mathbf{w}(\mathbf{x})$ and $\boldsymbol{\mu}_\mathbf{w}(\mathbf{x})$ are parameterized by neural network with weights $\mathbf{w}$. Typically, $\mathbf{\Sigma}_\mathbf{w}(\mathbf{x}) = \text{diag}(\boldsymbol{\sigma}^2_\mathbf{w}(\mathbf{x}))$ a diagonal matrix and shares the same backbone with $\boldsymbol{\mu}_\mathbf{w}(\mathbf{x})$. The reason for restricting the recognition model and the prior to be Gaussian distributions such that $\mathcal{KL}$ divergence term in the learning objective is tractable and can be computed directly using an **analytical solution**.
One final problem with the current framework is that the gradients for reconstruction term wrt $\mathbf{w}$ needs to backpropagate through *sampling* operation that is non-differentiable. This makes training the parameters with stochastic gradient methods challenging. VAE resolves this problem by invoking an alternate method for sampling from $q_\mathbf{w}(\mathbf{z}|\mathbf{x})$ which is called as the **reparameterization** trick.
Reparameterization trick moves the sampling step to an input layer and avoids directly sampling from $q_\mathbf{w}(\mathbf{z}|\mathbf{x})$. Given an input $\mathbf{x}_i$, $\mathbf{\Sigma}_\mathbf{w}(\mathbf{x}_i)$ and $\boldsymbol{\mu}_w(\mathbf{x}_i)$ can be *deterministically* computed. Alternative to sampling directly from the distribution $\mathcal{N}(\boldsymbol{\mu}_\mathbf{w}(\mathbf{x}_i), \boldsymbol{\sigma}_\mathbf{w}(\mathbf{x}_i),)$ one can sample a standard Gaussian distribution by $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ and construct the latent sample as

$$\mathbf{z} = \boldsymbol{\mu}_\mathbf{w}(\mathbf{x}_i) + \boldsymbol{\sigma}_\mathbf{w}(\mathbf{x}_i) \bigodot \epsilon$$

In this way one can sample from the latent variables by first sampling the error variable ϵ and computing z using deterministic functions. With the above trick the reconstruction term can be written as

$$\mathbb{E}_{\mathbf{z}\sim q_\mathbf{w}(\mathbf{z}|\mathbf{x}_i)}\left[-\log p_\mathbf{w}(\mathbf{x}_i|\mathbf{z})\right] = \mathbb{E}_{\epsilon\sim\mathcal{N}(0,1)}\left[-\log p_\mathbf{w}(\mathbf{x}_i|\mathbf{z} = \boldsymbol{\mu}_\mathbf{w}(\mathbf{x}_i) + \boldsymbol{\sigma}_\mathbf{w}(\mathbf{x}_i) \bigodot \epsilon)\right]$$

. A schematic illustration of *reparameterized* VAE is shown in Fig. 169

11.4.6 Applications

Although, variational autoencoders assume prior distribution of the latent space $p_\mathbf{w}(\mathbf{z})$ to be a standard normal distribution, with powerful neural networks they can be quite expressive. VAE can learn a mapping from the standard normal distribution to any distribution of the latent variables using the first few layers of the decoder networks. See Fig. 168 for an example. A broad range of recent demonstrations of VAEs are illustrated in Fig. 170.

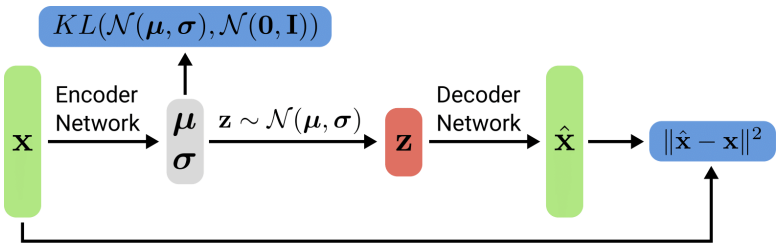
12 Generative Adversarial Networks

12.1 Generative Adversarial Networks

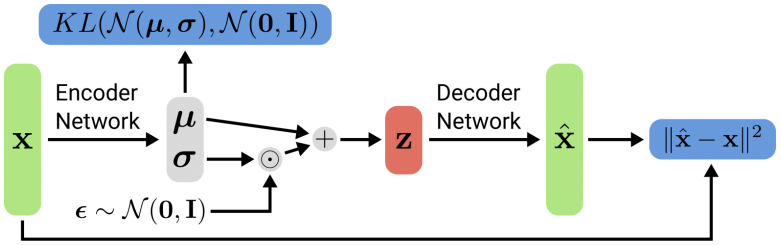
12.1.1 Generative Models

Generative models

- are probabilistic, they learn a probability distribution p_{model} to represent the true data distribution p_{data}
- are able to generate samples from p_{model}
- can in some cases estimate p_{model} explicitly and therefore allow to evaluate the (approximate) likelihood $p_{model}(\mathbf{x})$ of a sample $\mathbf{x}$
- are called implicit if they do not allow for evaluating the likelihood, explicit otherwise



(a) VAE regular



(b) VAE reparameterized

.

Figure 169: In a VAE we apply KL divergence measure on the recognition model and reconstruction loss on the likelihood model. With a regular VAE the latent variable $\mathbf{z}$ is sampled directly from the recognition model $q_{\mathbf{w}}(z|x)$. With reparameterization trick the latent variable is obtained by using deterministic estimates $\boldsymbol{\sigma}_{\mathbf{w}}, \boldsymbol{\mu}_{\mathbf{w}}$ and sampling ϵ from $\mathcal{N}(0,1)$

12.1.2 GANs

Generative Adversarial Networks are currently the most popular generative models. Just like (Variational) Autoencoders they are non-linear, but GANs are implicit models, do not have an encoder and are trained differently.
GANs use an adversarial process in which two models are trained simultaneously. Given a prior (e.g. gaussian) $p(\mathbf{z})$ over latent variables $\mathbf{z} \in \mathbb{R}^Q$ as well as observations $\mathbf{x} \in \mathbb{R}^D$ the two models can be described as:

- a generator $G_{\mathbf{w}_G} : \mathbb{R}^Q \mapsto \mathbb{R}^D$ that gets a sample $\mathbf{z} \sim p(\mathbf{z})$ as input and generates $\hat{\mathbf{x}} \sim p_{model}$ with p_{model} approximating p_{data}
- a discriminator $D_{\mathbf{w}_D} : \mathbb{R}^D \mapsto [0,1]$ that estimates the probability that a sample comes from p_{data}

The goal of G is to maximize the probability of D making a mistake while the goal of D is to make as little mistakes as possible. Due to the two models playing against each other, both of them need to improve constantly. This can be described as a two-player minimax game with a value function $V(G,D)$:

$$G^*, V^* = \underset{G}{\operatorname{argmin}} \ \underset{D}{\operatorname{argmax}} \ V(D,G) \tag{71}$$

$$V(D,G) = \underbrace{\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})]}_{\text{recognize true samples}} + \underbrace{\mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]}_{\text{reject samples generated by } G} \tag{72}$$

$V(G,D)$ describes the expectation of D recognizing samples from p_{data} and rejecting samples generated by G. The parameters of D are optimized towards maximising this expectation while the parameters of G are optimized towards minimizing $V(G,D)$. The optimization of the models is attained via Backpropagation.

Generator and discriminator can be implemented with different types of neural networks (e.g. MLPs, CNNs, RNNs). After training only the generator is kept in order to sample from p_{model}. The discriminator can be seen as a learned loss function on $\hat{\mathbf{x}}$ that is only necessary for training the generator. Fig. 171 shows an overview of the functionality of GANs.

12.1.3 Algorithm

In order to train GANs, the discriminator D is updated first. For k iterations an equal amount of real samples $\mathbf{x}$ as well as latent samples $\mathbf{z}$ are drawn. The latent samples are used to generate samples $\hat{\mathbf{x}}$. In each iteration the weights pf the discriminator are updated by stochastic gradient ascent of $V(G,D)$.
Afterwards, the generator is updated. After drawing new latent samples $\mathbf{z}$, the weights of G are updated via

stochastic gradient descent. Here, only the second part of $V(G, D)$ is used since the first part of the sum is independent of the generator.
D is optimized k times (typically with $k \in \{1, \dots, 5\}$) for each optimization of G in order to maintain it near its optimal solution, while preventing overfitting if the dataset is finite. For this to work, the generator has to be changed slowly, i.e. its learning rate must be small. The algorithm stops when the value function converges to some extreme.

. **While** not converged **do**

1. **For** k steps **do**

   1.1 Draw B training samples $\{\mathbf{x}_1, \dots, \mathbf{x}_B\}$ from $p_{data}(\mathbf{x})$

   1.2 Draw B latent samples $\{\mathbf{z}_1, \dots, \mathbf{z}_B\}$ from $p(\mathbf{z})$

   1.3 Update the **discriminator** D by **ascending** its stochastic gradient:

$$\nabla_{\mathbf{w}_D} \frac{1}{B} \sum_{b=1}^{B} \log D(\mathbf{x}_b) + \log(1 - D(G(\mathbf{z}_b)))$$

2. Draw B latent samples $\{\mathbf{z}_1, \dots, \mathbf{z}_B\}$ from $p(\mathbf{z})$

3. Update the **generator** G by **descending** its stochastic gradient:

$$\nabla_{\mathbf{w}_G} \frac{1}{B} \sum_{b=1}^{B} \log(1 - D(G(\mathbf{z}_b)))$$

12.1.4 The gradient trick

Early in the training, the generated samples $\hat{\mathbf{x}}$ are rather random, so the discriminator can easily distinguish between true and generated samples. This leads to a low gradient $\log(1 - D(G(\mathbf{z}))) = \frac{\log 1}{\log D(G(\mathbf{z}))}$. In contrast, $\log(D(G(\mathbf{z})))$ provides stronger gradients and can therefore be maximized instead of minimizing $\log(1 - D(G(\mathbf{z})))$. See Fig. 172 for visualization.

12.1.5 Expressiveness

GANs are very expressive. Fig. 173 shows an one-dimensional example of a gaussian p_{model} distribution shifting its mean and increasing the variance in order to adapt to p_{data}. It can be seen how with increasingly overlapping distributions p_{model} and p_{data} the discriminator becomes increasingly uncertain until it returns a probability of about 0.5 everywhere.

12.1.6 Theoretical results

Proposition 1. Optimal discriminator
For any given generator G, the optimal discriminator D is:

$$D_G^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_{model}(\mathbf{x})}$$

Proof. The training criterion for the discriminator D is to maximize (wrt. D):

$$V(G, D) = \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log(D(\mathbf{x}))\, d\mathbf{x} + \int_{\mathbf{z}} p(\mathbf{z}) \log(1 - D(G(\mathbf{z})))\, d\mathbf{z} \tag{73}$$

$$= \int_{\mathbf{x}} p_{data}(\mathbf{x}) \log(D(\mathbf{x})) + p_{model}(\mathbf{x}) \log(1 - D(\mathbf{x}))\, d\mathbf{x} \tag{74}$$

$$\tag{75}$$

$$f = a \log(y) + b \log(1 - y) \qquad (a, b) \in \mathcal{R}^2 \setminus \{0, 0\} \tag{76}$$

$$\Rightarrow f' = \frac{a}{y} + \frac{b}{(1 - y)} \cdot -1 \tag{77}$$

Set the derivative to 0 to calculate the maximum:

$$0 = \frac{a}{y} - \frac{b}{(1-y)} \Leftrightarrow \frac{a}{y} = \frac{b}{(1-y)} \tag{78}$$

$$\Leftrightarrow a - ay = by \Leftrightarrow a = by + ay \tag{79}$$

$$\Leftrightarrow y = \frac{a}{a+b} \tag{80}$$

This proposition shows that the optimal discriminator will return the probability of an input x being part of the data distribution p_{data}.

Theorem 1. Global Optimality
The global minimum of the virtual training criterion

$$V(G, D_G^*) = \mathbb{E}_{\mathbf{x}\sim p_{data}} [\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_{model}} [\log(1 - D_G^*(\mathbf{x}))] \tag{81}$$

$$= \mathbb{E}_{\mathbf{x}\sim p_{data}} \log \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_{model}(\mathbf{x})} + \mathbb{E}_{\mathbf{x}\sim p_{model}} \log \frac{p_{model}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_{model}(\mathbf{x})} \tag{82}$$

is achieved for $p_{model} = p_{data}$ where $D_G^* = \frac{1}{2}$ and $V(G, D_G^*) = -\log 4 \approx -1.386$.

Proof. Reformulation in terms of the non-negative Jensen-Shannon divergence yields:

$$V(G, D_G^*) = KL(p_{data}, p_{data} + p_{model}) + KL(p_{model}, p_{data} + p_{model}) \tag{83}$$

$$= -\log 4 + KL\left(p_{data}, \frac{p_{data} + p_{model}}{2}\right) + KL\left(p_{model}, \frac{p_{data} + p_{model}}{2}\right) \tag{84}$$

$$= -\log 4 + JSD(p_{data}, p_{model}) \tag{85}$$

since the Jensen-Shannon divergence $JSD(p_{data}, p_{model})$ is 0 if $p_{model} = p_{data}$ and $-\log 4$ is thus the minimal value that $V(G, D_G^*)$ can assume.

Here, it is proved that for the value function $V(G, D_G^*)$ to be minimal even though the discriminator is optimal, p_{data} has to be equal to p_{model}.

Proposition 2. Convergence
If G and D have enough capacity, and at each update step the discriminator D is allowed to reach $D = D_G^*$, and p_{model} is updated to improve

$$V(p_{model}, D_G^*) = \mathbb{E}_{\mathbf{x}\sim p_{data}} [\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_{model}} [\log(1 - D_G^*(\mathbf{x}))] \tag{86}$$

$$\propto \sup_D \int_{\mathbf{x}} p_{model}(\mathbf{x}) \log(1 - D(\mathbf{x}))\, d\mathbf{x} \tag{87}$$

then p_{model} converges to p_{data}.

Proof. The argument of the supremum is convex in p_{model}. The supremum doesn't change convexity, thus $V(p_{model}, D_G^*)$ is also convex in p_{model} with global optimum $p_{model} = p_{data}$ as shown in Theorem 1.

This proposition shows that with G and D large and complex enough and the discriminator being allowed to reach optimality each time it is updated, the generator will achieve to adapt p_{model} exactly to p_{data}.

These theoretical results are based on three assumptions.

- G has enough capacity to represent the data distribution and D has enough capacity to optimally discriminate between samples from p_{data} and p_{model}.To fulfill this assumption the models must be very large.
- For each update of G the discriminator D is updated until reaching the optimum D_G^*. In practice, this would be computationally too expensive and lead to overfitting when using finite datasets. Instead, Algorithm 12.1.3 is used.
- Optimize the model distribution p_{model}. Since the model distribution is represented by parameters of G, this is not possible.

Due to the assumptions not holding in practice, GANs might not converge to p_{data} or even oscillate. Nevertheless, GANs are empirically quite successful.

12.1.7 Mode Collapse

A common failure in GANs is mode collapse. If the generator learns to cover a part of p_{data} very well, the gradients regarding this part are low while the gradients regarding the rest of p_{data} are high. This can lead to the generator repeatedly learning different parts of the data distribution while unlearning the previous parts. Thus, the generator is always capable of producing high-quality samples but with very low variability.
There exist different strategies for avoiding mode collapse.

- encourage diversity: in minibatch discrimination the discriminator outputs a probability regarding the whole batch. As a result, a broader range of samples have to be generated closely to p_{data}
- anticipate counterplay: stabilize the training of the generator via anticipating the response of the discriminator. Fig. 174 shows an example of anticipate counterplay. This strategy requires backpropagating the generator gradient through the anticipation steps.
- experience replay: minimizes hopping back and forth between modes
- train multiple GANs: approximate the whole data distribution with multiple GANs covering a range of modes
- optimization objective: use different optimization strategies in order to improve the learning

12.1.8 Advantages and Disadvantages

An advantage of GANs is its flexibility: a wide variety of functions and distributions can be modeled. It is easier than other generative models since only backpropagation is required for training - the sampling happens before, when choosing an input - and unless VAEs, there is no approximation of the likelihood required. Nevertheless, GANs generally produce more realistic samples than VAEs.
The missing explicit representation of p_{model} might in contrast be an disadvantage since the sample likelihood can thus not be evaluated and used e.g. for inference or performance evaluations. Additionally, the discriminator and generator must be balanced well during training to ensure convergence to p_{data} and to avoid mode collapse. It is therefore quite hard to train GANs.

12.2 GAN Developments

12.2.1 Architecture Guidelines for stable Deep Convolution GANs (DCGAN)

In the DCGAN paper, a systematic search for a suitable architecture for Deep Convolutional GANs has been conducted. The following ideas have been found to work well in order to produce high quality images:

- Replace any pooling layers with **strided convolutions** (discriminator) and fractional strided convolutions for upsampling (generator)
- Use **batch normalization** in both the generator and the discriminator
- Remove fully connected hidden layers for deeper architectures
- Use **ReLU** activations in the generator except for the output which uses tanh
- Use **Leaky ReLU** activations in the discriminator for all layers

The resulting model was able to produce images of complex scenes like bedrooms and to produce reasonable interpolations between scenes. Comparable to the vector arithmetics of Word2Vec ([**?**]), DCGAN can perform arithmetic operations on z vectors in order to create new samples as shown in Fig. 175. The model thus seems to learn semantically meaningful representations.

12.2.2 Evaluating the performance of GANs

Evaluating the performance of implicit generative models is difficult. Therefore, a metric called Fréchet inception distance (FID) is often used to assess the quality of images created by a GAN. The idea behind FID is to take a pretrained Inception v3 network, to input real images and to calculate a distribution of these images by means of deeper, more semantically meaningful features computed inside the Inception network. In the following, GAN produced images are inputted and their distribution is computed based on the same features. This leads to multivariate distributions $\mathcal{N}(\boldsymbol{\mu}_d, \boldsymbol{\Sigma}_d)$ for the real images and $\mathcal{N}(\boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$ for the GAN images. FID then computes a distance between these to distributions:

$$FID = \|\boldsymbol{\mu}_m - \boldsymbol{\mu}_d\|_2^2 + \mathrm{Tr}(\boldsymbol{\Sigma}_m + \boldsymbol{\Sigma}_d - 2(\boldsymbol{\Sigma}_m\boldsymbol{\Sigma}_d)^{1/2})$$

FID could be proven to increase when manipulating images or using images from different distributions. While it can detect mode collapse within a class of images, it is not suitable for measuring mode collapse over the whole data distribution - unless knowing where to find modes and testing each of them separetely.

12.2.3 Gradient Penalties and Convergence

One mayor advance for making GAN training more stable and less dependent on hyperparameters was the introduction of regularizers such as Gradient Penalties, e.g.

$$V(D,G) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}\left[\log D(\mathbf{x}) - \lambda\|\nabla_{\mathbf{x}} D(\mathbf{x})\|^2\right] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}\left[\log(1 - D(G(\mathbf{z})))\right]$$

The regularizer penalizes large gradients of the discriminator with respect to x. This has been shown to lead to converging instead of oscillating parameters (Fig. 176). When applying gradient penalties to DCGANs, high quality images can be produced without needing to rely on ideas like the ones mentioned in 12.2.1.

12.2.4 CycleGAN

CycleGANs are used for Image-to-Image translation, i.e. they map images from one domain X to another domain Y and back. A cycle consistency loss is used to check how well an image x is reconstructed after being mapped to another dimension and back. See Fig. 177 for visualization. Possible use cases are the creation of paintings out of photographs or photographs out of line drawings.

12.2.5 Progressive Growing of GANs

An idea to improve the performance of GANs is to start with small layers and low resolutions and to progressively add bigger layers used for creating images with higher resolution. This is one of the approaches used in state-of-the-art GANs today.

12.2.6 BigGANs

Really large image sets like ImageNet are difficult to handle for GANs due to their diversity. The idea of BigGANs is to create class-conditional GANs which receive the class label as an additional input. Instead of using regularizers which might hamper the performance of GANs, the authors monitored the singular values of the weight matrices of generator and discriminator. By means of these singular values they could determine when mode collapse started to happen and stop training at these locations.

12.3 Research at AVG

The goal of the research group is to create intelligent systems that can interact with a complex 3D environment. One task in order to achieve this goal is to understand the environment in terms of its geometry and appearance. This means, a 3D shape has to be predicted from images. While the input is well defined - it consists of one or multiple images - different types of output can be returned. In the past, Voxels, Points or Meshes have been used. As one can see in Fig. 178 these outputs discretize space and are thus rather poorly detailed.
The idea of the occupancy network paper was to not represent the 3D shape explicitly. Instead, the surface is considered implicitly as the decision boundary of a non-linear classifier such that the decision boundary separates the points inside an object from the points outside:

$$f_\theta : \mathbb{R}^3 \times \mathcal{X} \to [0,1]$$

The first input is a 3D-location. $\mathcal{X}$ inputs a condition, e.g. an image and the neural network then outputs an occupancy probability.
The representation can then be learned by a VAE in order to create new samples. Moreover, it can be extended to 4-dimensional space and used e.g. to model conditional surface light fields or Generative Radiance Fields.

References

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv.org*, 2018.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research (JMLR)*, 3:1137–1155, 2003.

[3] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capability and trainability in recurrent neural networks. *Proc. of the International Conf. on Learning Representations (ICLR)*, pages 1–17, 2017.

[4] Mané Vasudevan Le Cubuk, Zoph. Learning augmentation strategies from data. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[5] Rauber Schütt Bethge Geirhos, Temme and Wichmann. Generalisation in humans and deep neural networks. 2018.

[6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[9] Leal-Taixe and Niessner. I2dl.

[10] Herbert Robbins and Sutton Munro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 1951.

[11] Krizhevsky Sutskever Srivastava, Hinton and Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. In *Journal of Machine Learning Research (JMLR)*, 2014.

[12] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *Proc. of the International Conf. on Machine learning (ICML)*, volume 28, pages 1139–1147, 2013.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[14] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *ISCA Speech Synthesis Workshop*, pages 1–15, 2016.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

References

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv.org*, 2018.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research (JMLR)*, 3:1137–1155, 2003.

[3] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capability and trainability in recurrent neural networks. *Proc. of the International Conf. on Learning Representations (ICLR)*, pages 1–17, 2017.

[4] Mané Vasudevan Le Cubuk, Zoph. Learning augmentation strategies from data. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[5] Rauber Schütt Bethge Geirhos, Temme and Wichmann. Generalisation in humans and deep neural networks. 2018.

[6] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[9] Leal-Taixe and Niessner. I2dl.

[10] Herbert Robbins and Sutton Munro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 1951.

[11] Krizhevsky Sutskever Srivastava, Hinton and Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. In *Journal of Machine Learning Research (JMLR)*, 2014.

[12] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *Proc. of the International Conf. on Machine learning (ICML)*, volume 28, pages 1139–1147, 2013.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

[14] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *ISCA Speech Synthesis Workshop*, pages 1–15, 2016.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

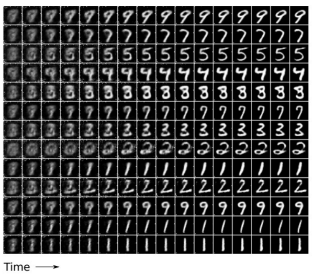


Figure 7. **MNIST generation sequences for DRAW without attention.** Notice how the network first generates a very blurry image that is subsequently refined.

Figure 8. **Generated MNIST images with two digits.**

Figure 9. **Generated SVHN images.**

(a)

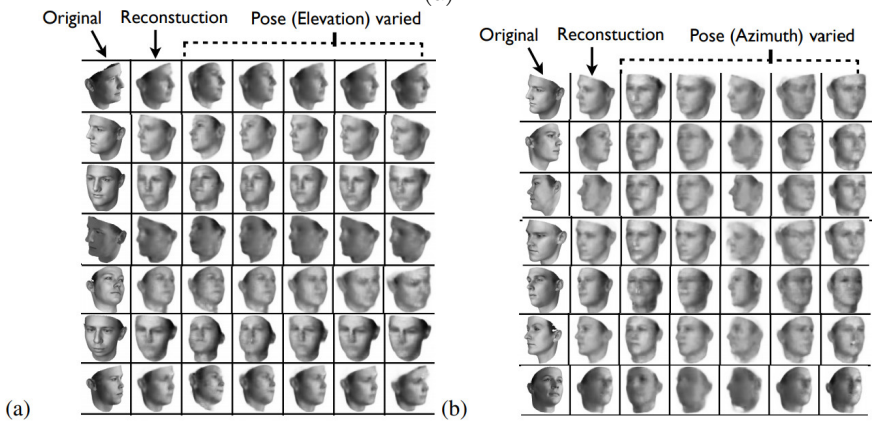


Figure 4: **Manipulating pose variables:** Qualitative results showing the generalization capability of the learned DC-IGN decoder to rerender a single input image with different pose directions.

(b)

Figure 1: Class-conditional 256x256 image samples from a two-level model trained on ImageNet.

(c)

Figure 170: Applications of variational autoencoders

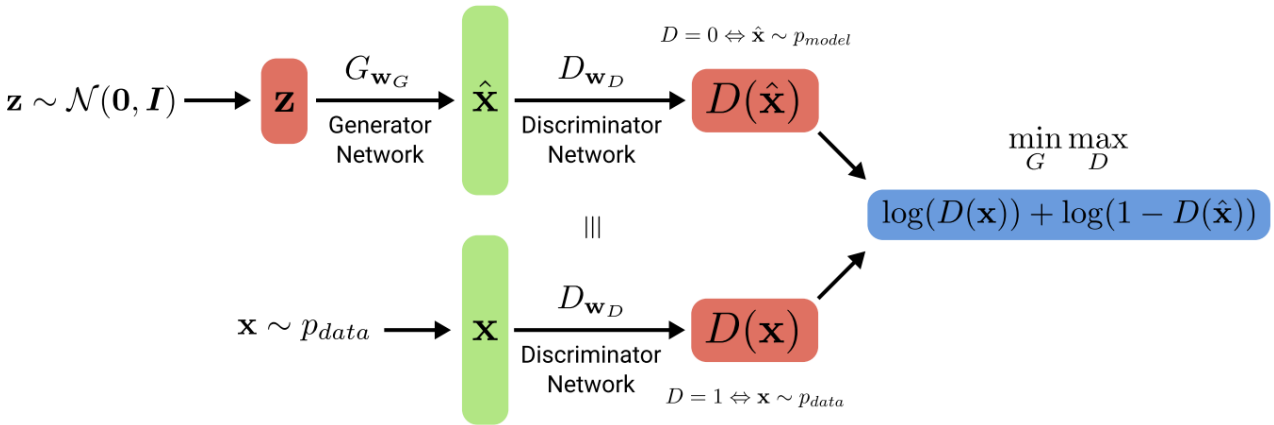


Figure 171: **Overview of the functionality of a GAN.**

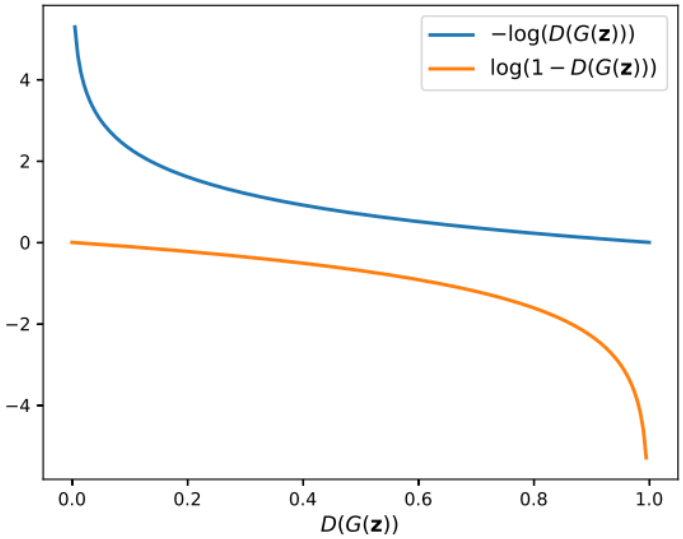


Figure 172: **The gradient trick.** While $\log(1 - D(G(\mathbf{z})))$ has a low gradient in the beginning, $\log(D(G(\mathbf{z})))$ has a high gradient and is thus more useful for training.

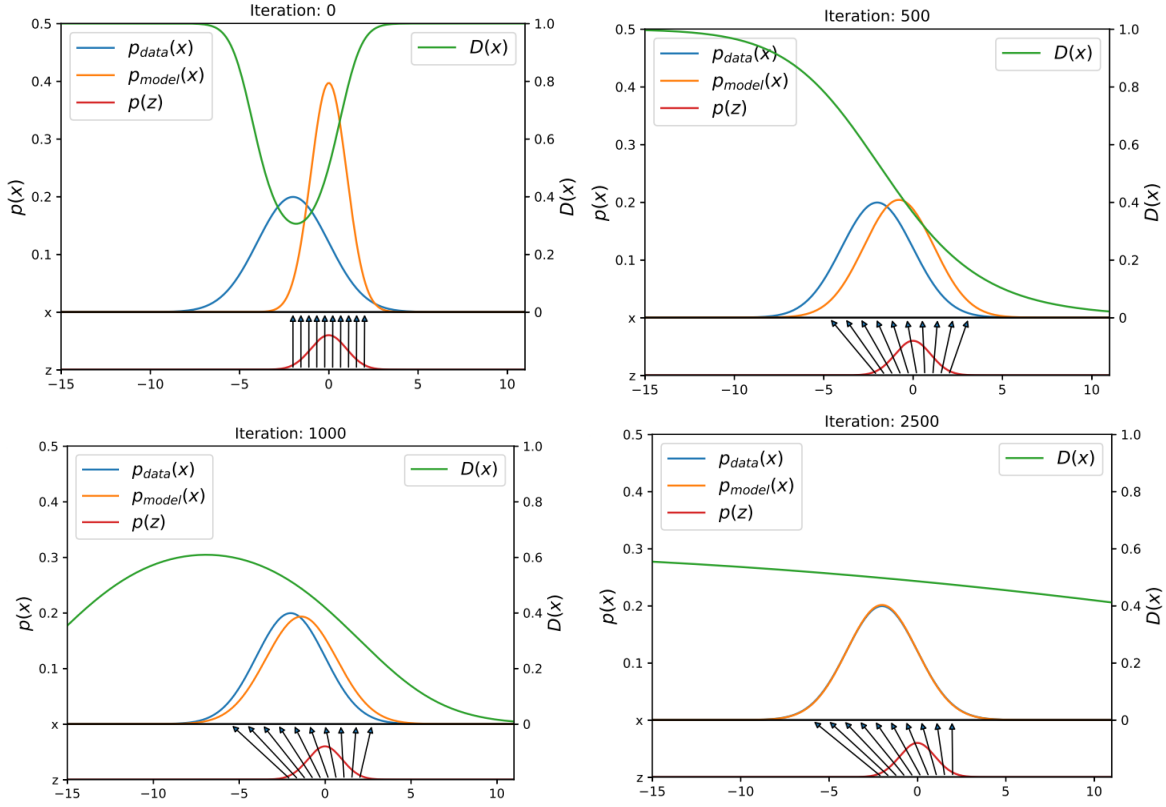


Figure 173: **Adapting p_{model} to p_{data}.**

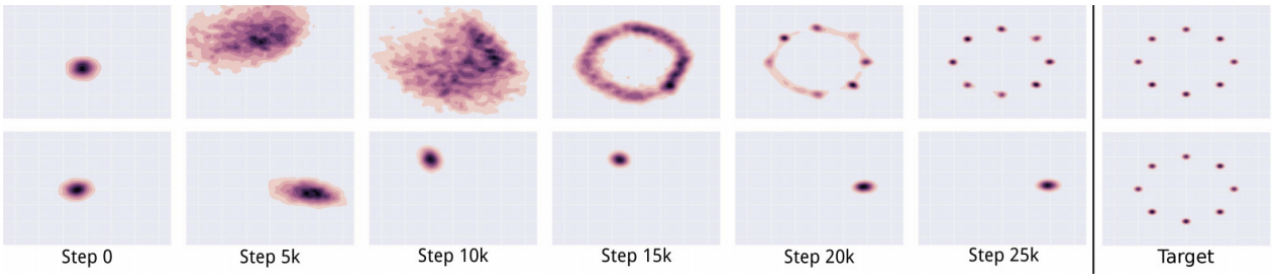


Figure 174: **Anticipating the response of the discriminator.** The first row shows the result of a GAN on a target distribution with multiple modes when using an anticipation strategy. The second row shows how the GAN jumps between modes if no anticipation strategy is used.

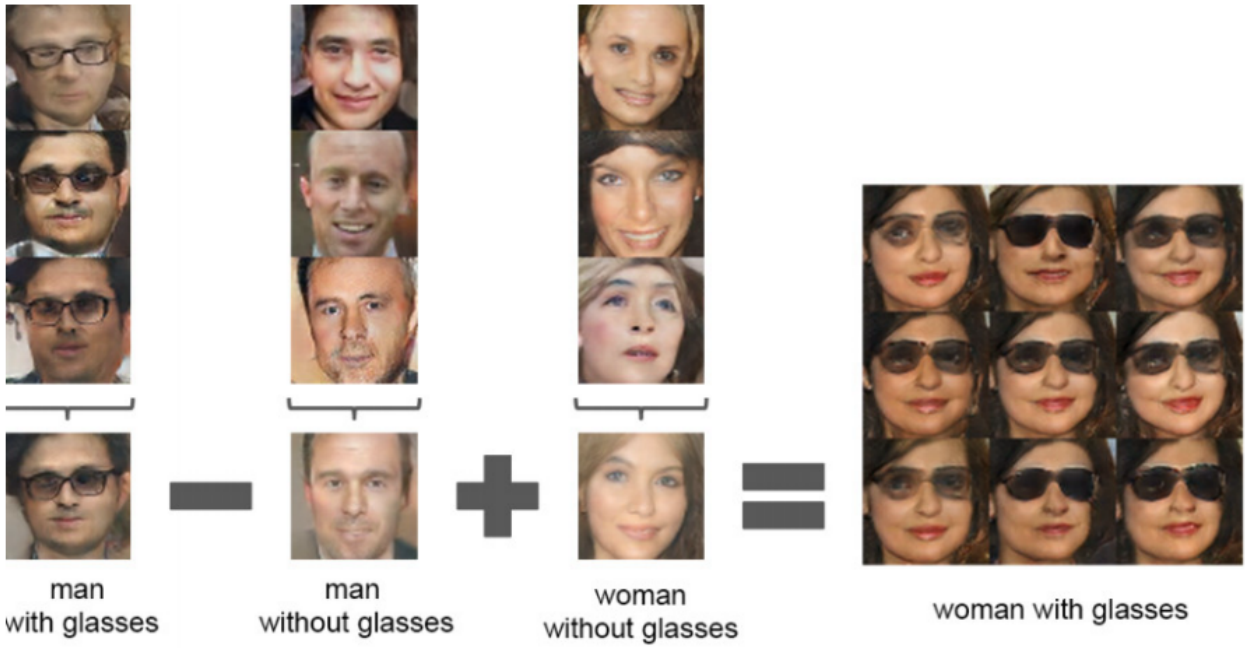


Figure 175: **Vector arithmetic on averaged z vectors of samples. z** vectors are combined via arithmetic operations and Gaussian noise is added to produce new samples.

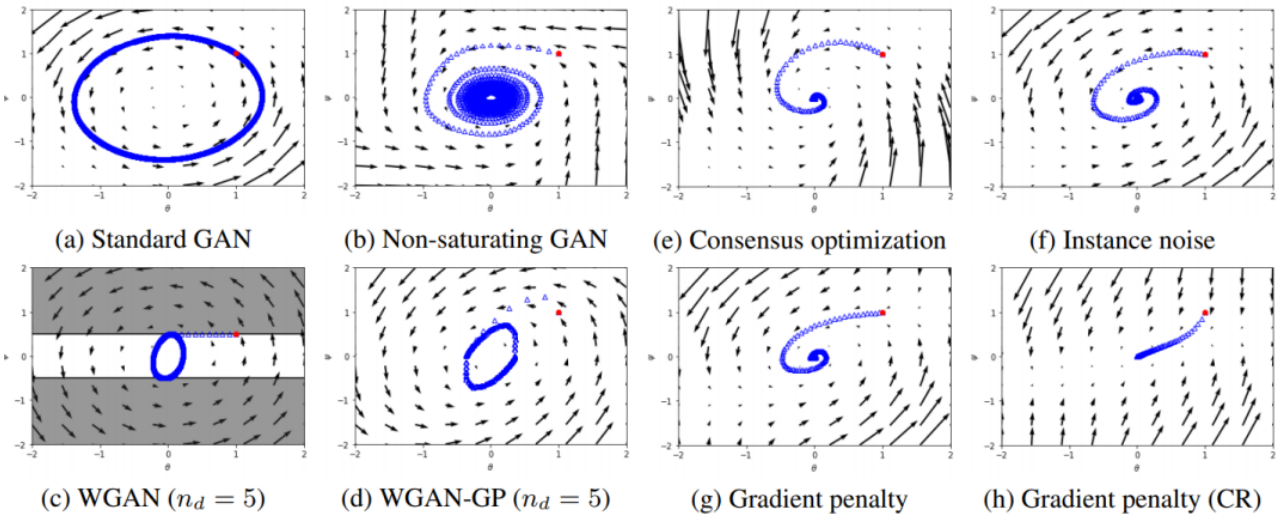


Figure 176: **Gradient penalties.** (a) the parameters oscillate. (g) and (h) the paramters converge to an equilibrium.

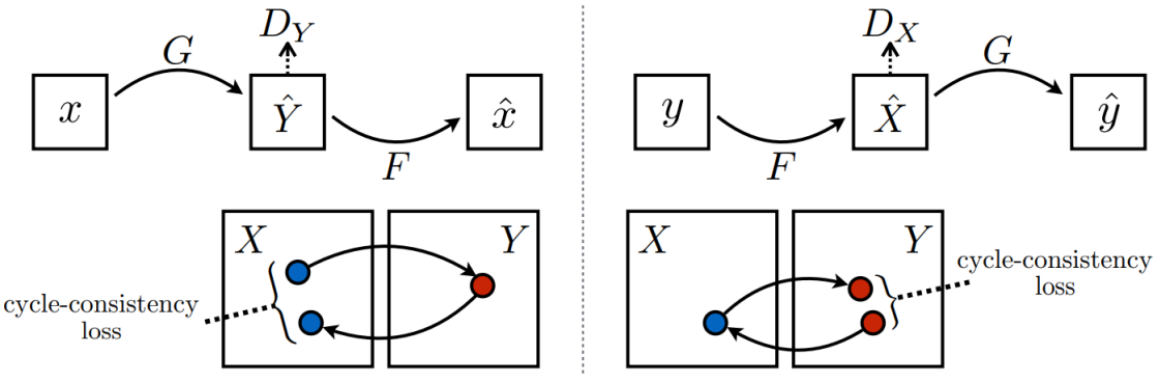


Figure 177: **Cycle GAN image-to-image translation.**

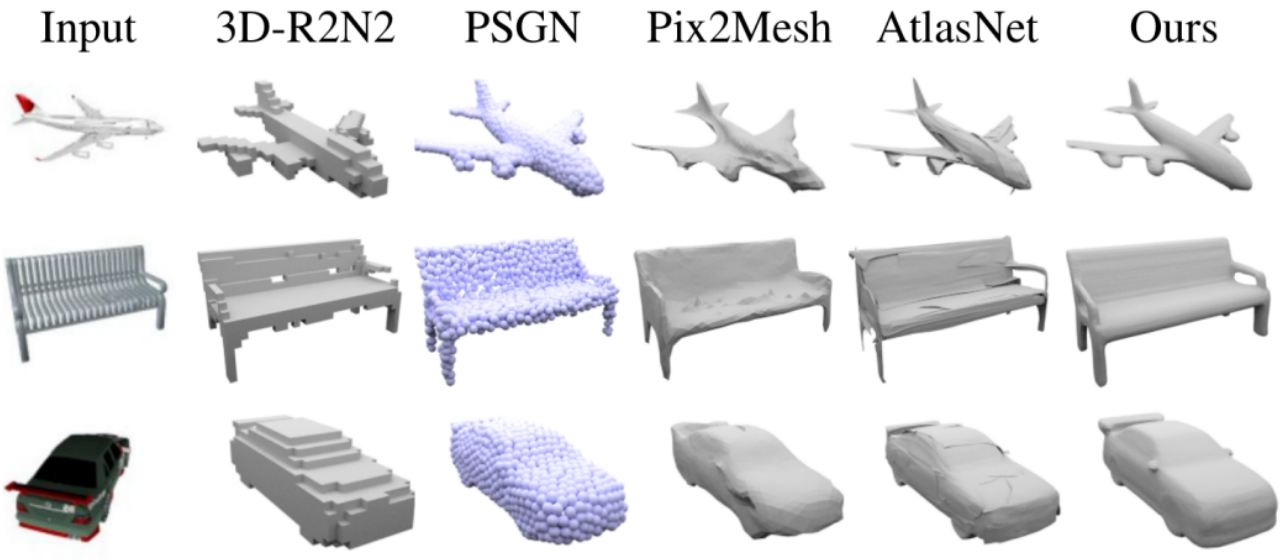


Figure 178: **3d representations of images.** 3D-R2N2 uses Voxels. PSGN uses Points. Pix2Mesh and AtlasNet use Meshes.