Deep Learning (2IMM10) - Introduction

Vlado Menkovski

Spring 2020

1 Introduction

Learning outcomes of this chapter:

- Understand when to use Machine Learning
- Be able to motivate when to use Deep Learning
- Understand and be able to communicate what are the advantages of learning representations with neural network models

1.1 Motivation - ML

Many of the systems we aim to develop have the goal of enabling automation and in turn efficiency. Examples of such developments exist in domains such as computer vision, speech recognition, and natural text understanding. Building such solutions can be done by carefully designing a set of rules that guarantee the outcome will satisfy the given goal. To be able to do this we need to understand the domain sufficiently well. In many cases, however, such solutions can become increasingly complex.

One example would be the launch of a satellite in orbit. The mechanics involved in achieving orbit and placing the satellite in a particular location and velocity require highly complex calculations over sensory information. The control of the vehicles is also very complex and requires a large set of rules to achieve its desired operation. However complex, this problem does not necessitate a learning algorithm because there is sufficient understanding that allows for a successful solution that meets the desired goal. Any algorithm that would rather learn from observation would not exceed the performance of an expertly designed solution.

So, why are many problems that are much easier for humans so much harder to solve algorithmically using expert knowledge — instead requiring learning from examplesfig. 1?

Let us take two such problems as examples: driving a car and translating a text from one language to another. What are the unique properties of these two problems that make them well-suited for Machine Learning approaches?

It basically boils down to our inability to express the goals of these task with rules sufficiently precisely. This can either stem from our lack of understanding or from the underlying complexities in the system. In many such cases the concepts that we use to describe the system do not fully explain the state and evolution of it.

Many such examples can be found in biological systems. For example we can measure the genetic

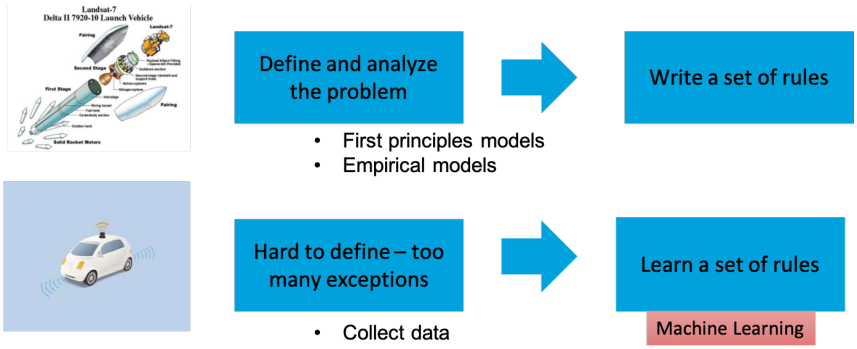


Figure 1: Traditional modeling versus machine learning.

code of a cell, we can also measure many other molecules present in it. However, these measurements typically destroy the cell so we cannot measure the evolution of the values in time precisely enough to develop a sufficiently accurate model of this system.

In other cases the sheer number of rules is so large that we cannot hope to express them fully. Imagine a scenario of detecting cats and dogs in images by looking at pixel values. Expressing all the ways in which combinations of pixel values determine whether a picture contains a cat or a dog, in rules is a very challenging task.

Nevertheless, even without full understanding from observing the system we may be able to make predictions. For example, we may learn that when a gene in the cell is expressed a particular behaviour can be expected, or the statistics of the pixel value of an image may be indicative of whether we are looking at airoplaes or zebras.

Such examples, motivate the use of Machine Learning (ML). ML gives us tools to build models from examples (observations) rather than data.

Question: What are the challenges self driving cars and translation pose that motivate ML?

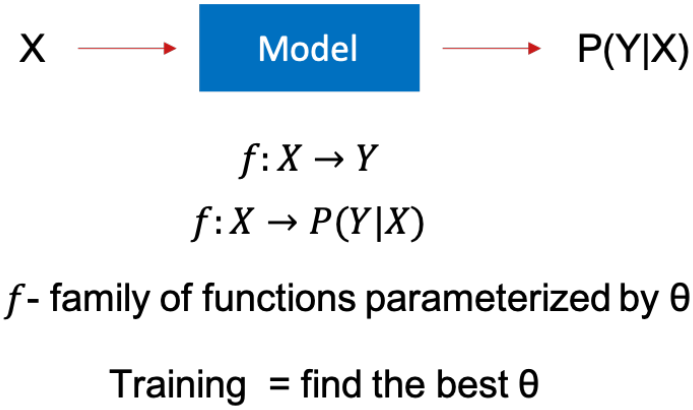


Figure 2: Formulation of a machine learning model as a parameterized function.

Motivation - ML - Definition

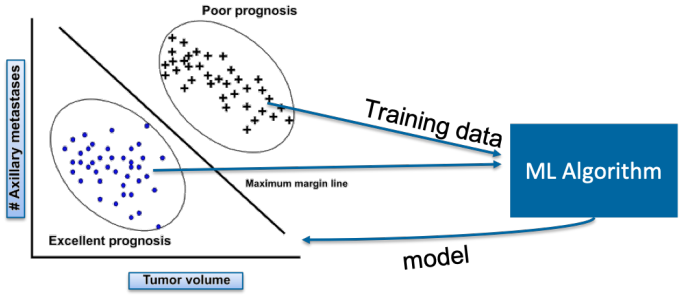
A Machine Learning model can be formulated as a function that maps its input X to an output Y and is parameterized by a parameters θ. In this formulation we assume that there is a set of rules that can determine the value of Y fully from the value of X.

Note: We typically use probability theory to express uncertainties about such maps. A ML model would express the probability of Y receiving certain values given a value for X.

ML algorithms are then developed to find good parameters θ. Typically this is done through an optimization process. If the model maps some input X to some output $\hat{Y}$, and the correct values would have been Y, some error, or loss, between $\hat{Y}$ and Y is computed. The algorithm tries to minimize this error by adjusting the values of θ.

Motivation - ML - Limitations

These algorithms have their own limitations. When the number of input variables is small, and there is a strong correlation between the variables and the correct outcomes, ML algorithms tend to perform well. However, as dimension of the input grows, and the output becomes less dependent on specific variables, ML algorithms often have a hard time finding good parameters for the model. For example, for large images the value of a single pixel is only very weakly related to whether the image contains a cat or a dog. Consequently ML algorithms have much more difficulty classifying large images than small images.



$$f_c = P(y = c|x)$$

$$f_c(y = c|x) = \frac{1}{1 + e^{-(w^T x + b)}}$$

- Logistic regression

Training Data

Tumor volume	# Metastasis	Prognosis
3	0	good
4	1	good
12	4	not good
...	...	...

Figure 3: Data set for the prognosis of cancer patients. Blue circles indicate an 'Excellent' prognosis and black plus-signs indicate a 'poor' prognosis.

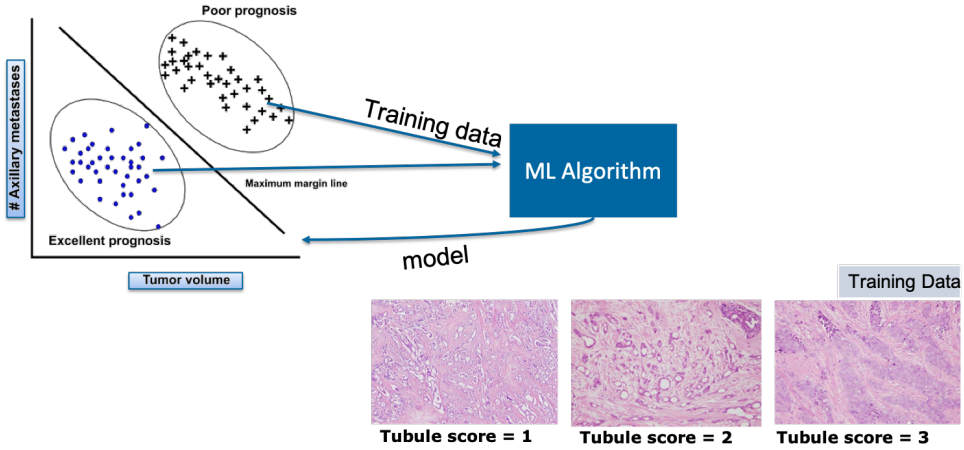


Figure 4: The data sets belonging to the two different examples.

Motivation - ML - Example

Let's look at a fictional dataset of cancer patients. Figure 3 illustrates a dataset with two features:

- Tumor volume
- Number of auxiliary metastasis

The target variable for this dataset is the prognosis for the patient. This is a discrete variable that can have two values:

- Excellent
- Poor

In the figure each datapoint is labelled with blue circle for 'Excellent' and black plus sign for 'Poor'. This dataset has a strong correlation between the two features and the target variable. Specifically, the datapoints with different labels form clear clusters that can be linearly separated, i.e. we can draw a straight line between the data points with different labels. In other words, we can define a linear combination of the two features that determine the value of the target variable.

The parameters of that linear combination (that is our model) are what the ML algorithms need to determine from the available data.

Question: Why can't we specify rules from which we can determine the prognosis given the values of the features for this problem?

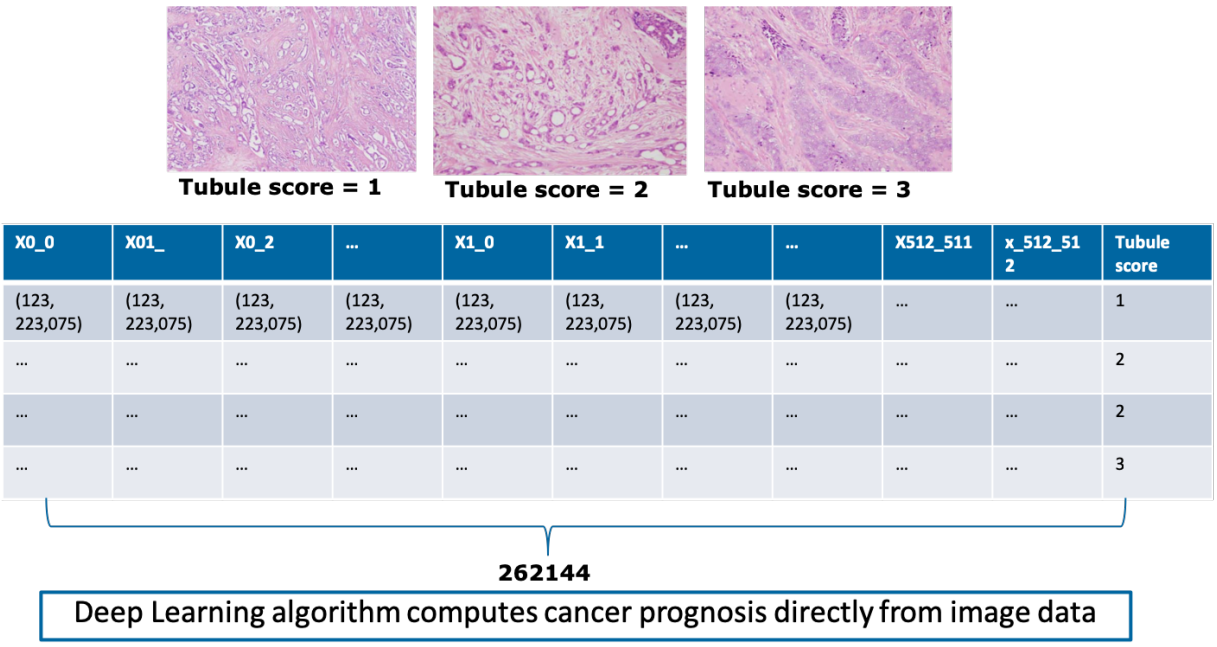


Figure 5: The input to our model consists of the values for all pixels. That makes 262144 three-dimensional variables.

Motivation - ML - Example

In another setting, we are facing a similar task of determining the prognosis of cancer patients. However, in this case the data has significantly different properties. The dataset in this case consists of optical images of biopsy samples. The images of the samples show different structures of the tissue that reflect the severity of the disease. Specifically, the samples are from breast tissue and our aim is to determine the prognosis of breast cancer. In the bottom row of Figure 4 you can see three different images. Each of the images shows tissue with a different 'tubule score' which will be our target variable. The 'tubular score' is part of the diagnostic process and involved in determining the prognosis.

In contrast to the previous example where there were two input variables that were strongly correlated with the target variable, now each of the pixels in the image is a variable that we need to consider. As illustrated in Figure 5, given in a table format we can observe 262144 variables.

It is no surprise that the values of each pixel are very weakly correlated with our target, the tubule score. Therefore it is evident that training a model that maps each parameter value to the target value would be exceedingly difficult.

As our main challenge is that these features are far removed from the target variable, can we do something to improve this? In other words, can we engineer features that would be more useful for predicting the target variable?

Motivation - ML - Feature Engineering

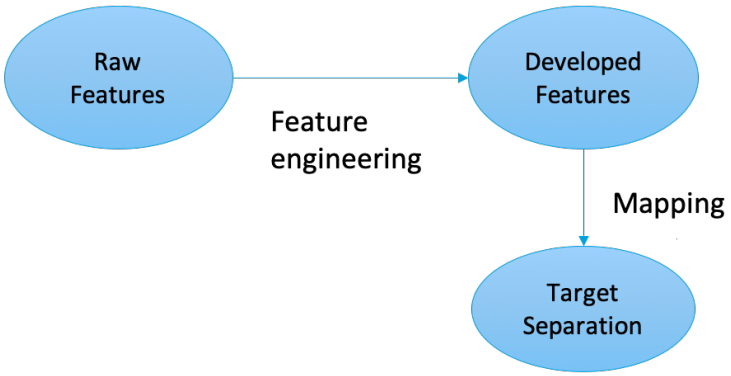


Figure 6: Feature engineering

One approach to address the challenge of using 'low-level' or 'raw' features for building ML models is to refine them in some way or to build more informative features, or 'high-level' features, from them. This approach is referred to as *Feature Engineering*.

The goal of feature engineering is to develop features that are more informative and for which a ML algorithm can develop a sufficiently good model. The assumption here is that we can use some kind of knowledge or expertise that will allow us to develop such 'high-level' features from the 'low-level' features. This is in a way analogous to the expertly designed rules, however now we are designing rules to develop features. The next step is to use a ML algorithm to develop a model using those features (fig. 6). In contrast to the expert systems the ML algorithms can actually indicate the quality of the developed features. If a feature has a weak relation to the target value, it will end up not being used in the model or the model will have very low sensitivity to that feature. So, in principle we can use ML tools to improve the feature engineering process (fig. 7).

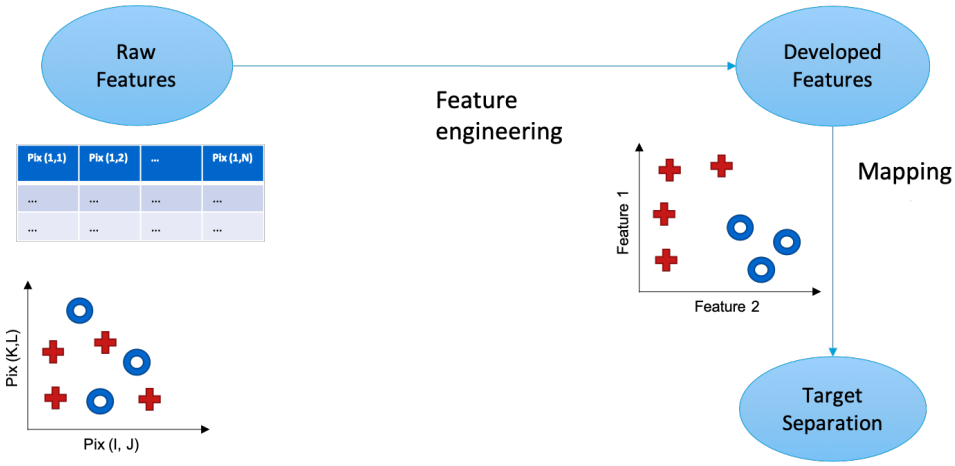
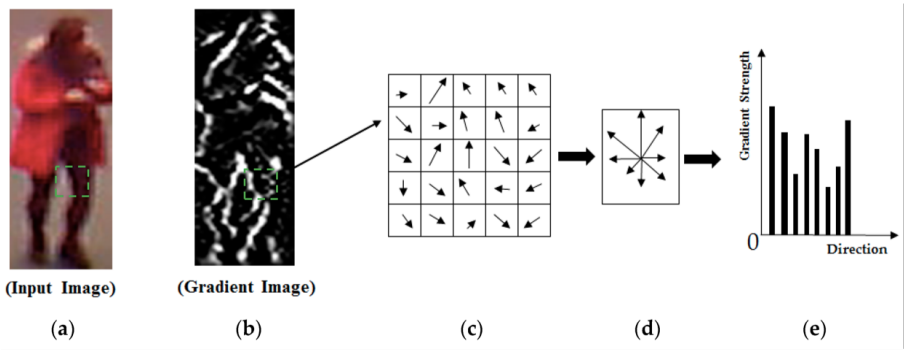


Figure 7: A schematic overview of feature engineering.

Question: What would be a possibility for a feature engineering target given our tubule score example? Do you see any patterns in the images for which we can develop feature engineering algorithms?



Nguyen, Dat Tien, et al. "Person recognition system based on a combination of body images from visible light and thermal cameras." *Sensors* 17.3 (2017): 605.

Figure 8: An illustration of a Histogram of Gradients

One problem with feature engineering is that in many cases the 'low-level' features are very high dimensional[1] and domain knowledge is not available (or at least not to a level that would allow for developing hard-coded rules).

One example of feature engineering using in general purpose image analysis such as the histogram of oriented gradients (HOG) fig. 8. This technique counts the occurrences of gradient orientations in small patches of the image. These are technique is very useful in detecting objects from background and is helpful in describing some objects, but falls short of capturing the high level concepts that we aim for in analysing images of various downstream tasks. Therefore, it is usually the case that we need to custom engineer features for specific tasks.

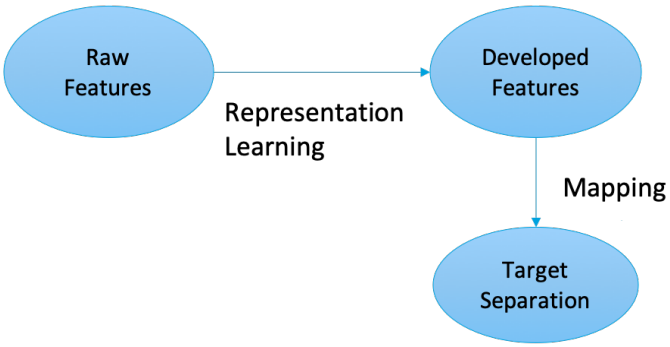


Figure 9: A schematic overview of representation learning

1.2 Motivation - ML - Representation Learning

As mentioned earlier, when feature engineering, we often face similar challenges as when trying to develop rule-based models. High quality features are difficult to design, and the process takes time. Moreover, features developed for specific tasks are not necessarily useful for other tasks. Finally,

[1]For example, with the pictures of biopsy samples we have 512 by 512 pixels all taking 3 values. That makes our input $512 \cdot 512 \cdot 3 = 786432$ dimensional.

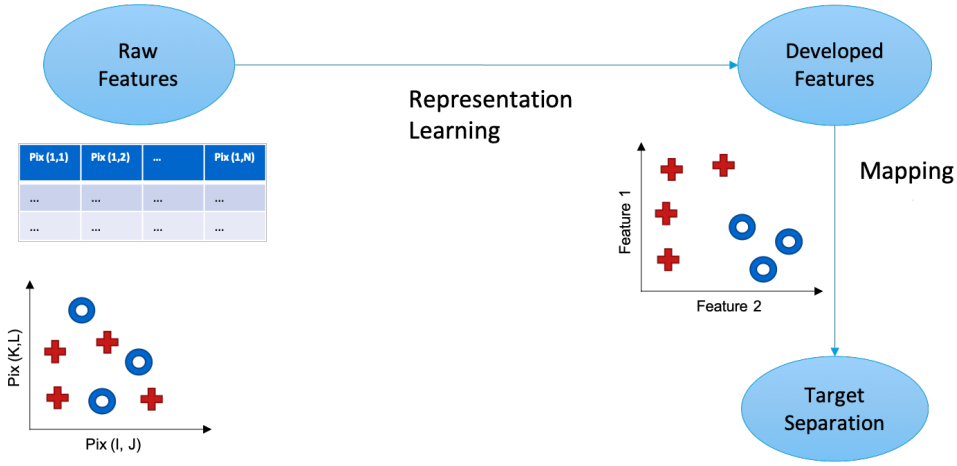


Figure 10: Representation learning end-to-end

in many cases (e.g. speech recognition) feature engineering has never managed to deliver features that were informative enough to enable the building of models with sufficient level of accuracy.

This invites the question of whether we can use ML to learn features in the same way that we learn models (or learn the parameters of models). Approaches trying to do this are referred to as *Representation Learning* (fig. 9). Much of this course revolves around representation learning, specifically learning representations of high-dimensional data for different tasks.

The setting is is the following: in many problems we face high-dimensional (low-level) data for which individual variables are far removed from a target variable and hence for which finding a map from the features to the target is difficult.

We study how to use artificial neural networks (ANN) to efficiently learn such maps. ANN models consists of a sequence of 'layers' that apply non-linear transformations to their inputs. This structure allows for learning hierarchical features with increasing level of complexity. This way, the neural network model can compose features of features of features in a hierarchical way such that sufficiently informative high-level features can then have a simple (even linear) map to the target variable (fig. 10).

This type of representation learning can also be referred to as *end-to-end representation learning* as the loss function[2] from the end of the model is used to learn the features at all levels back to the beginning (fig. 11). Moreover, as these ANN models can be deep (consisting of many layers) for certain types of data and tasks, the family of methods using them is referred to as *Deep Learning*.

1.3 Course overview

In this course we study the following topics (fig. 12 [3]):

[2] That is the function we try to minimize in order to train the model. Often this function is a measure of the error between predictions and correct outcomes.

[3] orange boxes

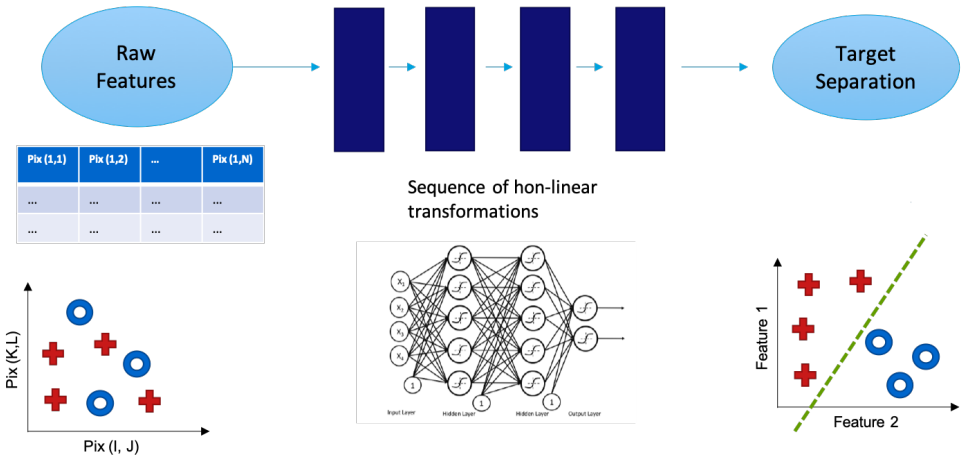


Figure 11: Representation learning with a neural network

- Artificial Neuron
- Stochastic gradient descent and backpropagation
- Multilayered perceptron (MLP)
- Convolutional Neural Networks (CNN)
- Recurrent Neural Networks (RNN)
- (Deep) metric learning
- Generative models
  - Variational Autoencoders (VAE)
  - Autoregressive models
  - Generative Adversarial Networks

In addition to these topics that follow techniques for building models, in the course we also cover solutions for downstream tasks using these techniques (fig. 12[4]). These include:

- Word embeddings
- Models for spatially distributed data
- Models for sequential data
- One shot learning
- Density estimation
- Image generation

[4]blue boxes

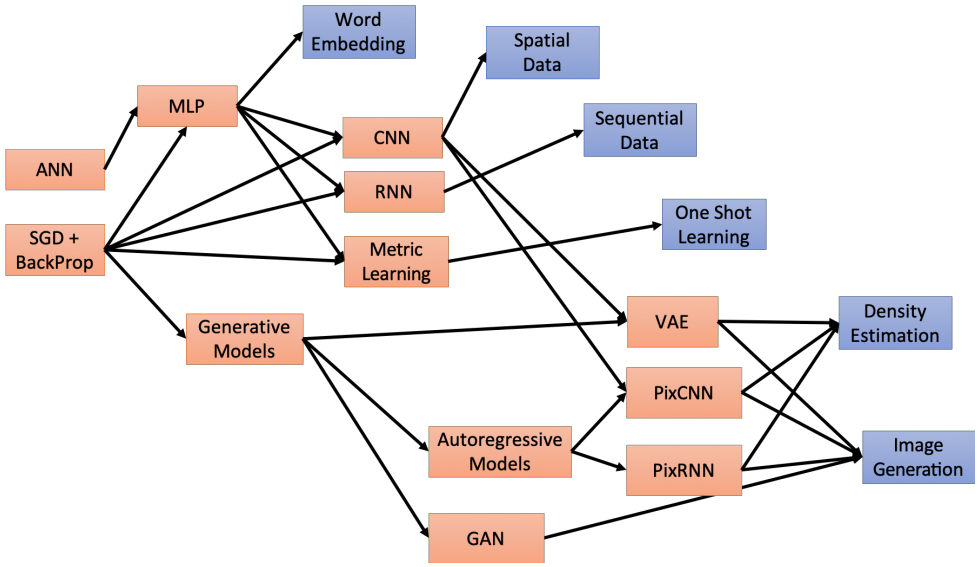


Figure 12: Course overview

The dependencies of the course topics are given in fig. 12. The course is organized in five content chapters:

- Primer of Neural networks
- Word embedding
- Models for spatially distributed data
- Models for sequential data
- Generative models

The assignment of each of the topics to the course chapters is presented in fig. 13

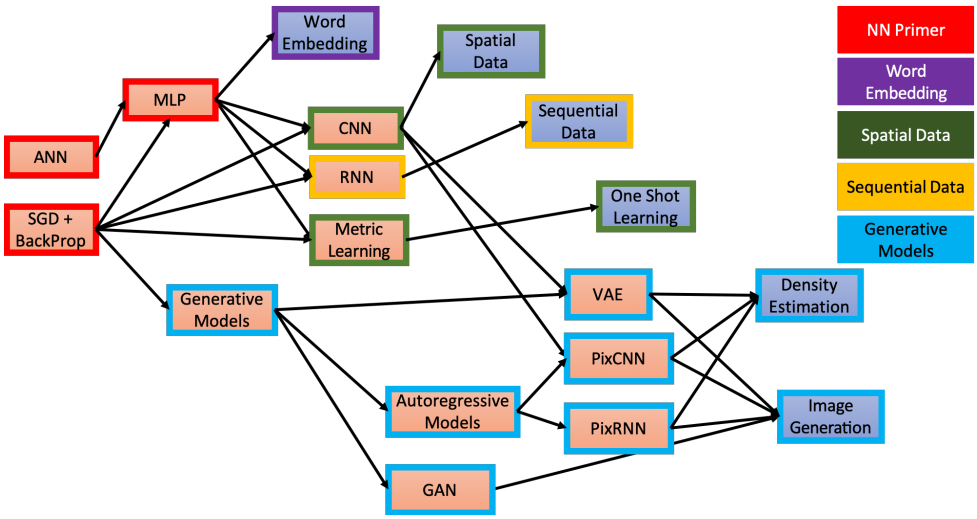


Figure 13: Caption

Deep Learning (2IMM10) - NN Primer

Vlado Menkovski
Spring 2020

The learning outcomes of this chapter:

- You are able to design and train a multi-layer percepton model

The main theme of this course is developing representations of high dimensional data. In the introduction chapter we discussed the motivation and utility of having such representations, in this chapter we will go over the underlying methods for developing such representations.

1 The Artificial Neuron

The main component of the artificial neural network is the artificial neuron. You can see a graphical representation in Figure 1. To understand what an artificial neuron is, let us look at an example of a neuron taking four inputs.

- The inputs to the neuron are given by x_0 to x_4, denoted as an input vector $\mathbf{x}$
- Edges between nodes have parameters $\mathbf{w}$, called weights, and a bias term b. The weights are used to calculate a weighted sum of the inputs, which is then offset by the bias. The weights and bias together are the parameters, denoted as θ, of the neuron.
- Finally the neuron applies some — usually non-linear — 'activation' function to the weighted sum.
- Putting everything together, this means a single neuron implements computes its output $o_\theta(x)$ as:

$$o_\theta(x) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T\mathbf{x} + b).^1$$

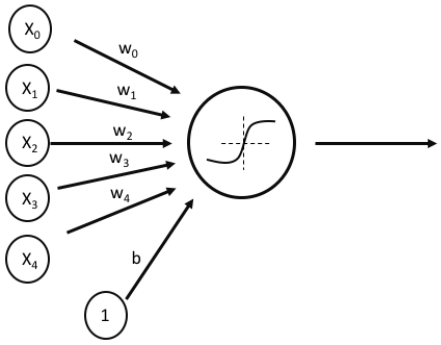


Figure 1: Artificial neuron

[1] In order to make notation a bit more concise, we will often omit writing the bias, giving to the notation $\phi(\mathbf{w}^\top\mathbf{x})$. The addition of the bias is implied. This convention is also quite common in the literature.

For example, if we have an artificial neuron taking three input variables, and if the input to the artificial neuron is a vector $\mathbf{x} = [1, 2, 3]^\top$ and the values of the parameters are $\mathbf{w} = [1, 0.5, 1]^\top$ and $b = 0.5$, the artificial neuron give the following output:

$$\begin{aligned} o_\theta(\mathbf{x}) &= \phi(\mathbf{w}^\top \mathbf{x} + b) \\ &= \phi(1 \cdot 1 + 2 \cdot 0.5 + 3 \cdot 1 + 0.5) \\ &= \phi(1 + 1 + 3 + 0.5) \\ &= \phi(5.5) \end{aligned}$$

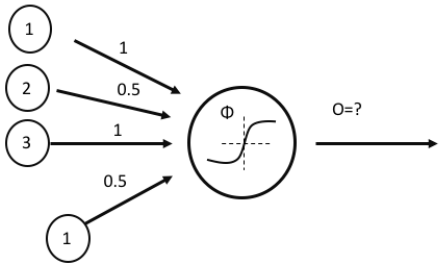


Figure 2: Artificial neuron activations

where ϕ is the activation function of the artificial neuron. The activation function can be any differentiable[2] function such as the hyperbolic tangent ($\phi(x) = \tanh(x)$) or simply a linear activation ($\phi(x) = x$).

If in our simple example we choose as an activation function $\phi(x) = x$, then for the given input, $\mathbf{x} = [1, 2, 3]^\top$, the artificial neuron will produce the value 5.5 — see Figure 2 for a schematic overview.

1.1 Linear Regression

If we choose the activation to be linear, $\phi(x) = x$, our artificial neuron becomes a linear regression model:

$$o_\theta(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

Let us look at how we can use this linear neuron to model a data set. As an example we will use the dataset from Figure 3.

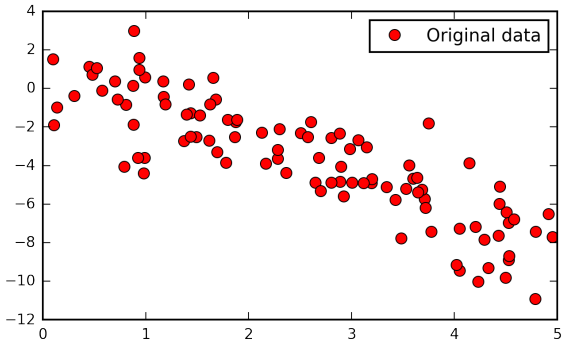


Figure 3: A dataset for which we might want to use a linear regression model.

Empirical Risk minimization

To use our neuron to model a dataset, we want to find the "best" values for our parameters $\mathbf{w}$ and b. Our dataset consists of pairs $(\mathbf{x}^{(i)}, y^{(i)})$ where $\mathbf{x}^{(i)}$ is an input vector, and $y^{(i)}$ is the corresponding value we wish to predict. Given any values for our parameters $\theta = (\mathbf{w}, b)$, our model makes predictions $\hat{y}^{(i)} = o_\theta(\mathbf{x}^{(i)}) = \mathbf{w}^\top \mathbf{x}^{(i)} + b$ for the inputs $\mathbf{x}^{(i)}$. In order to determine how good our parameters are, we can define a "loss function", $L\left(y^{(i)}, \hat{y}^{(i)}\right)$, that measures how far

[2] In practice it is also possible to use activation functions that are almost everywhere differentiable, such as the "Rectified Linear Unit", ReLU: $x \mapsto \max(x, 0)$.

our predictions are off, and see what the average
loss on the data set is:

$$\frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right).$$

This averaged loss is called the *empirical risk*. We can then define the "best" values for our paramaters to be those for which the empirical risk is minimal, and "training" the model then becomes the optimization task of minimizing this empirical risk:

$$\begin{aligned}\theta_{\text{best}} &= \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right) \\ &= \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(o_\theta(\mathbf{x}^{(i)}), y^{(i)}\right). \end{aligned} \tag{1}$$

This process is called *Empirical Risk minimization*. The most common loss function for linear regression models is the "mean squared error" which in our case comes down to taking

$$L(y, \hat{y}) = (y - \hat{y})^2.$$

2 Gradient Descent & Backpropagation

The minimization in eq. (1) is something that we actually need to implement with an algorithm. There are many more efficient implementations of the optimization of linear regression models, however for our purposes we will study gradient descent as it allows us to scale this to the much more complex and non-linear models that we aim for in this course.

The idea behind gradient descent as an optimization algorithm is quite simple. Suppose we have a continuously differentiable function $f : \mathbb{R}^2 \to \mathbb{R}$ of two variables, whose graph looks like some hilly landscape. We start out on some hill side, and want to get to a (local[3]) minimum. What we could do is see what direction the hill goes down most steeply in, and set a step in that direction. After that step we again look at what direction the hill is steepest in and take a step down, and so on and so on. This is gradient descent: the gradient of f, $\nabla f(u, v) = \left(\frac{\partial}{\partial u} f(u, v), \frac{\partial}{\partial v} f(u, v)\right)$, at a point $(u, v) \in \mathbb{R}^2$ points in the direction in which f has the steepest increase, and the negative of the gradient of f points in the direction of steepest descent.

To apply gradient descent to our machine learning problems, we view the empirical risk, or average loss function, as a function of the model parameters. We let the model make predictions for the dataset, calculate the loss and calculate the derivatives of that loss with respect to the model parameters. Then we update the parameters. In the following subsections, we will look at gradient descent in detail.

[3]If the function that we want to minimize is a convex function with Lipschitz gradient, and if we make the right choices for our step size, gradient descent can be proven to converge to a global minimum. Our linear regression model with mean square error satisfies these conditions. However in general in machine learning the best we can hope for is a useful local minimum.

The Gradient Descent Algorithm

- Given a set of n examples in a data set $D : \{(\mathbf{x}, y)\}$.
- GD Update rule:
  - repeat until convergence

$$\begin{aligned} w &\leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \\ b &\leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \end{aligned} \tag{2}$$

where α is a parameter referred to as the learning rate

2.1 Gradient Descent Optimization

Gradient descent (GD) optimization is a first-order iterative optimization algorithm. The algorithm updates the parameters of the model in an iterative fashion until it converges, i.e. until the loss value does not decrease any more. The update values of the parameters is given by eq. (2). So, the new values of the parameters we subtract from the old values a value proportional to the gradient of the loss with respect tothe parameters.

Intuitively, the gradient of the loss with respect to the parameters of the model indicates how the loss value changes by changing the parameter values. If the loss increases with an increase of a parameter, the gradient will have a positive value. As we aim at decreasing the loss we should then decrease the value of that parameter, hence the minus sign in the expression and the 'descent' in the name of the algorithm.

Because the gradient indicates how much every parameter influences the loss function, we take the amount of change to the parameters to be proportional to the gradient, multiplied by a "learning rate". The learning rate parameter α allows us to scale the step size . The value of the learning rate has a strong impact of the efficiency of the training. Think about how this happens:

- How would a large learning rate affect the training?
- How would a small learning rate affect the training?
- Does gradient descent optimization guarantee finding the parameters that minimize the loss?

To be able to implement GD optimization we need to be able to compute the gradient of the loss with respect to the parameters. As the loss is computed based on the output of our model, all components of our model need to be differentiable with respect to the parameters. We will come back to this requirement as we increase the complexity of the neural network models and include different components.

In general when developing these models we follow the following steps:

Requirements:

- Define the model and its parameters:
  - $o_\theta = \mathbf{w}^\top x + b$
  - $\theta : \{\mathbf{w}, b\}$
- Define the Loss function:
  - $L(\mathbf{x}, y; \mathbf{w}, b) = \frac{1}{N} \sum_{i=0}^{n-1} (o_\theta - y)^2$
- Specify the gradient of L with respect to the parameters $\mathbf{w}$ and b:
  - $\frac{\partial}{\partial w} L(.)$
  - $\frac{\partial}{\partial b} L(.)$

As the complexity of the models increases we will use tools such as Tensorflow to automate some of these steps and their calculations. However, to understand how this works, we will now look at how the gradients are computed.

2.2 Computing the gradient of the loss

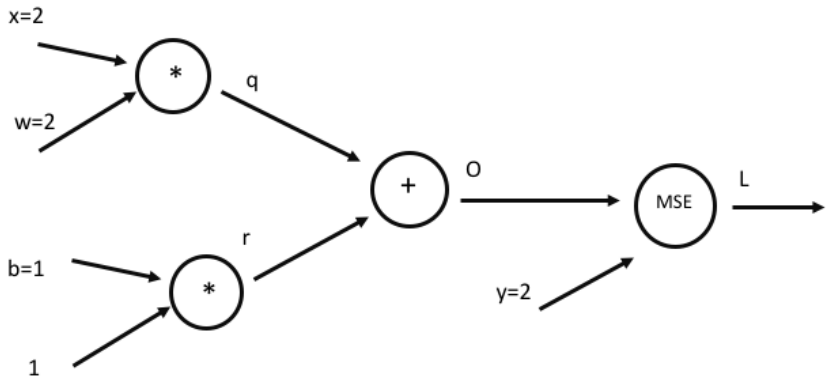


Figure 4: *Computation graph* - For a single datapoint $\{x = 2, y = 2\}$

Let us compute the gradient for our artificial neuron given a single datapoint. To help us follow the computation of the gradient we depict our model as a directed, acyclic graph: the *computation graph*. The computation graph has edges that carry values, labeled with variable names, and nodes that specify the operations on those values. Figure 4 shows the computation graph for our artificial neuron from Section 1.1 for a single datapoint and one-dimensional input.

Computing the gradient of the loss — The forward pass

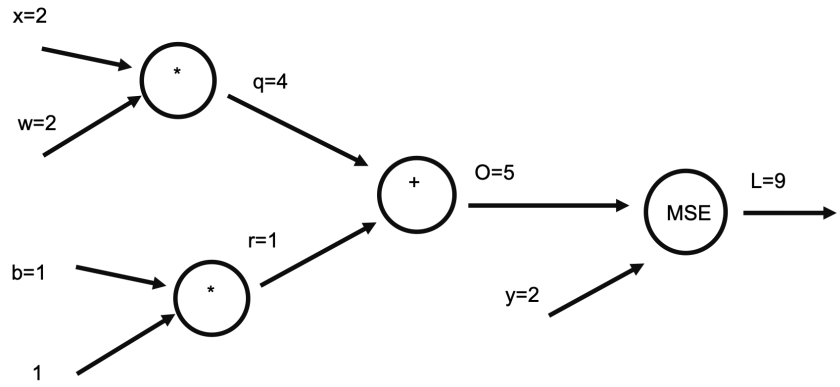


Figure 5: *Computation graph* — The forward pass

To compute the gradient of the loss with respect to to all the parameters in our model, we need to have all the intermediate values in our computation graph. We compute those in a *forward pass* over the compute graph — see Figure 5. The computations are as follows:

- $o = wx + b = 2 \cdot 2 + 1 = 5$,
- $L = \frac{1}{1}(o - y)^2 = (5 - 2)^2 = 3^2 = 9$.

Computing the gradient of the loss — The backward pass

Now that we have the intermediate values we can compute the gradient of the loss starting from the end of the computation graph and going backwards — see Figure 6.

$$
\begin{aligned}
\frac{\partial L}{\partial L} &= 1 \\
\frac{\partial L}{\partial o} &= 2(o - y) \cdot 1 = 2(o - y) \\
\frac{\partial L}{\partial q} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial q} = 2(o - y) \\
\frac{\partial L}{\partial r} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial r} = 2(o - y) \\
\frac{\partial L}{\partial w} &= \frac{\partial L}{\partial q}\frac{\partial q}{\partial w} = 2(o - y) \cdot x \\
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial b} = 2(o - y) \cdot 1
\end{aligned}
$$

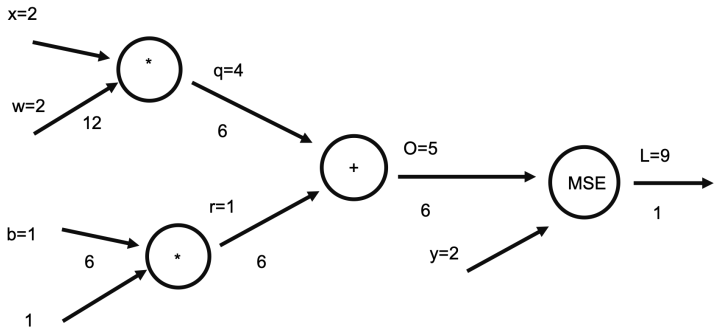


Figure 6: *Computation graph* — The backward pass

This algorithm is called backpropagation and it is basically a way to apply the derivative chain rule. The algorithm scales very well to large models. *The only requirement we have is that we need to compute the gradient of the output of any node in the compute graph with respect to its input variables.*

The Backpropagation Algorithm

- Compute forward pass
  - $o = x \cdot w$
- For each node compute the local derivatives
  - $\dfrac{\partial o}{\partial x}$
  - $\dfrac{\partial o}{\partial w}$
- Backward pass the derivative
  - apply the chain rule

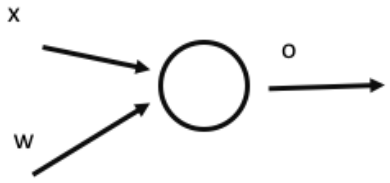


Figure 7: A node in a computation graph to perform Backpropagation on.

Computing the gradient of the loss — Updating the parameters

Now that we've computed the gradient of the loss through the backpropagation algorithm, we are ready to set a step in the gradient descent algorithm, i.e. to update the parameters. With the gradients and values from Figure 6 and learning rate $\alpha = 0.1$ this is done as:

$$\begin{aligned} w &\leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \\ &= 2 - 0.1 \cdot 12, \\ b &\leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \\ &= 1 - 0.1 \cdot 6. \end{aligned}$$

3 Classification

In Section 1.1 we looked at a model doing linear regression to predict continuous values, and in Section 2 we saw how we can train our linear regression model with gradient descent optimization using backpropagation. Next we aim to develop a model for classification. When doing classification, instead of predicting a continuous value for each data point, we want to predict discrete values coming from some fixed set of classes. To achieve this we specify our model's output as a discrete probability distribution over the set of target values.

$$f_c(\mathbf{x}) = P(y = c|\mathbf{x})$$

An example of binary classification

- Two input variables: x_0, x_1
- Two classes

$$c1 = \text{class 1}$$
$$c2 = \text{class 2}$$

- Output is

$$f_{c1}(\mathbf{x}) = P(y = c1|\mathbf{x})$$
$$f_{c2}(\mathbf{x}) = P(y = c2|\mathbf{x})$$

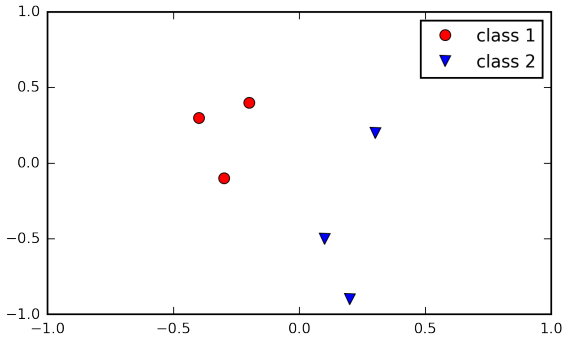


Figure 8: An example of a data set for which we might want to do classification.

To practically achieve this using the artificial neuron we use an activation function whose output has the properties of a probability distribution over a discrete set. In case of binary classification the activation typically used for this is the "logistic sigmoid" function, or "sigmoid" for short. In case of multiclass classification, where every data point belongs to only one class, the most common activation function is the "softmax"[4] function. Before we delve deeper into classification, let us first give a brief description of these two functions.

Logistic sigmoid function:

- is used for binary classification;
- is called "sigmoid" for short;
- is given by

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}};$$

- is the inverse of the "logit" function;
- can also be used when a single datapoint can belong to multiple classes;
- takes values between 0 and 1;
- its graph is shown in Figure 22e.

Softmax function:

- is used for multiclass classification;[5]
- takes a vector as its input;
- is given by

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}};$$

[4]The softmax function is not really an activation function for a single neuron, but takes the outputs of a bunch of neurons as its input. How this is done will become clear in Section 5.2.

- its components are positive and sum to 1.

3.1 Perceptron

To understand how classification can be done using Neural Networks, we will now try to do binary classification on a dataset with two input variable — see for example the data set in Figure 8 — using a single artificial neuron with sigmoid activation.

As inputs to our artificial neuron we have the two variables, x_0 and x_1, together denoted in vector notation as $\mathbf{x}$. As with the linear regression model, we first multiply our inputs by some weights $\mathbf{w}$, then sum them and add the bias b. Finally we feed the result of this into a sigmoid activation function. A schematic overview of this is given in Figure 9. As a whole the artificial neuron implements the following:

$$\begin{aligned} o_\theta(x) &= \text{sigmoid}(w_0 x_0 + w_1 x_1 + b) \\ &= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \\ &= \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}. \end{aligned}$$

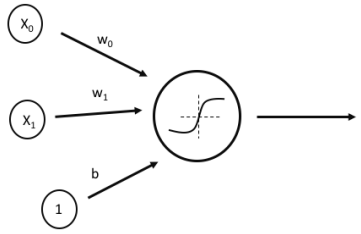


Figure 9: A schematic overview of our artificial neuron used for binary classification.

Binary classification

Now how do we use this neuron for binary classification? As mentioned earlier, we view its output as a probability distribution over our classes. More specifically, if we have classes 1 and 2, we see the output of our neuron as

$$\begin{aligned} P(y = 1 \mid \mathbf{x}) &= o_1 \\ &= o_\theta(\mathbf{x}) \\ &= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \\ P(y = 0 \mid \mathbf{x}) &= o_0 \\ &= 1 - o_1. \end{aligned}$$

Then as a loss function, we use the negative log-likelihood of this Bernoulli distribution, also known as "binary cross-entropy." The likelihood of an outcome $y \in \{0, 1\}$ for a Bernoulli random variable with success probability $o_\theta(x)$ can be written as

$$P(y) = o_\theta(\mathbf{x})^y (1 - o_\theta(\mathbf{x}))^{(1-y)},$$

so that the loss becomes

$$\begin{aligned} L(\mathbf{x}, y; \theta) &= -(y \log(o_\theta(\mathbf{x})) + (1 - y) \log(1 - o_\theta(\mathbf{x}))) \\ &= -\log(o_y). \end{aligned}$$

[5] Within the field of deep learning, the softmax function is also commonly used for so-called attention mechanisms.

As with the linear regression model, we now want to use Gradient Descent to minimize the empirical risk. To be able to do this we again use back propagation on a computation graph to get the gradients. The computation graph for this model is shown in Figure 10. Try to work out the gradient of the loss with respect to the model parameters for yourself — *hint: use case distinction between* $y = 0$ *and* $y = 1$.

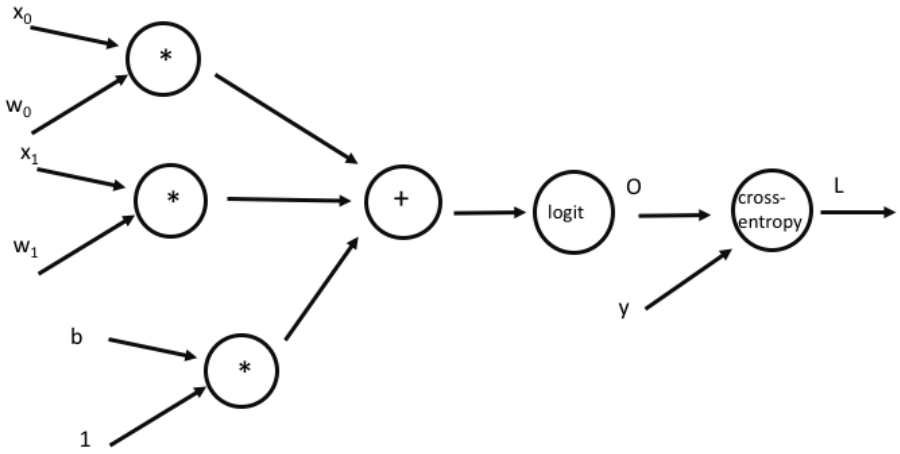


Figure 10: The computation graph for our single neuron binary classifier.

4 Multilayer Perceptron

4.1 The limitations of the Artificial Neuron

The artificial neuron model develops a linear combination of the input values to compute the output value. In the case of the classification task this linear combination forms a hyperplane boundary that separates the datapoints that are associated with a different label.

To address this limitation and enable the model to represent more complex, non-linear maps between the input and the output, we combine multiple neurons that have a non-linear activation function. First stack neurons in order to get a "hidden layer" and next stack layers on top of each other to get deeper models.

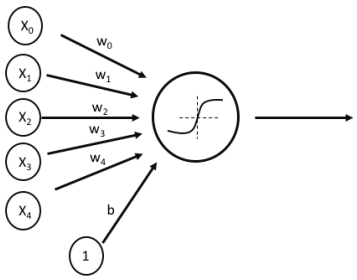


Figure 11: A single artificial neuron

Multilayer Perceptron — A hidden layer

Neurons can be stacked to form a layer of neurons. Every neuron in the layer gets the same input variables, and the outputs of the neurons together form the output vector of the layer. The neurons in the layer typically have the same non-linear[6]activation function. We can then use the output of the layer as input to an artificial neuron. Depending on the activation of the final neuron we can then use the model for non-linear regression, or for binary classification of data that is not linearly separable. This structure is an example of a "Multilayer Perceptron" (MLP), one with a single "hidden layer". A graphical representation is given in Figure 12.

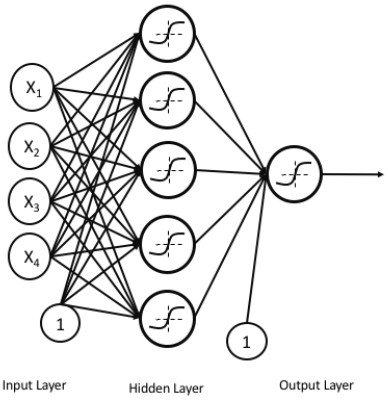


Figure 12: A Multi Layer Perceptron with a single hidden layer.

Note: This model consisting of a set of artificial neurons is an artificial neural network. The set of inputs in this model is referred to as *input layer*. The layer of neurons that we stacked on top of each other is referred to as the *hidden layer*, and the last layer that produces the output is the *output layer*.

Multilayer Perceptron — Stacking layers

In order to further increase the ability of our models to model non-linear data, we can stack layers of neurons on top of each other. The output of one hidden layer then becomes the input to the next. The number of hidden layer is called the "depth" of the network, and models with many layers stacked on top of each other are called "deep neural networks". The kind of structure that we get is shown in Figure 13.

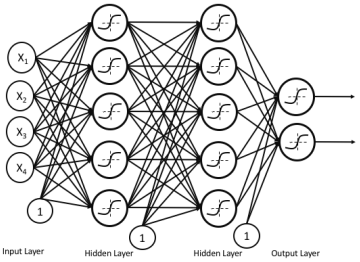


Figure 13: An MLP with multiple hidden layers

The advantage of having multiple layers is that a complex boundary between two assignments (decisions) about the input can be decomposed as a set of simpler boundaries that are then combined in the next layer. For deeper models this sequence of compositions can allow for developing highly complex maps from the input to the target variables.

Note: In principle such highly complex maps can also be achieved with a single hidden layer that has sufficiently many neurons as these neurons. However, this is significantly less efficient in terms of the number of parameters. Models that benefit from developing compositional features scale significantly better as for computing the higher level features the lower level features are re-used.

[6]Whereas with a single artificial neuron we could use linear activation in order to do regression, using a linear activation in a hidden layer does not make sense: we then have a composition of affine transformations which is itself an affine transformation, so we would not have gained any expressiveness for the model. Therefore a linear activation is typically only used in the output layer of an MLP.

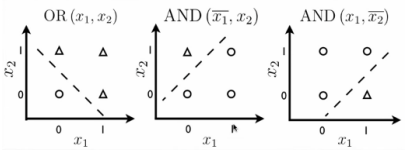


Figure 14: Dataset of binary operations that is linearly separable

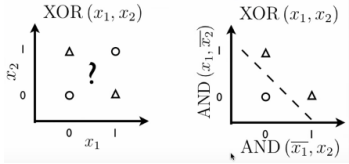


Figure 15: Data set of XOR binary operation that is not linearly separable

(The term features here refers to output values of neurons. During training neurons specialize in detecting specific pasterns of their input and can be referred to as feature detectors).

A simple example of where linearly separability is not sufficient is data generated by the binary xor operation (figs. 14 and 15).

Multilayer Perceptron — Summary so far

So far we have seen that we can stack neurons together to form layers, and that we can stack layers on top of each other to get deeper models. The models that we create this way are called Multilayer Perceptrons (MLPs). The following is a brief overview of what an MLP is:

- An MLP is a directed acyclic graph
  - where the nodes are artificial neurons
  - and the edges are parameterized connections between them.
- The nodes are organized in layers,
  - there are no connection between neurons within a layer
  - and all neurons in the same layer of the same type (have the same activation function).
- The set of inputs to the MLP is called the input layer,
- and the final layer is called the output layer.
- It is also referred to as feedforward Neural Network (meaning that there are no loops and information flows from the input directly to the output)

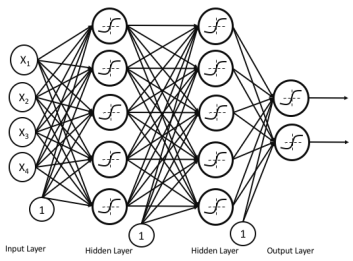


Figure 16: The structure of an MLP

We can also think of MLPs in terms of composition of functions. Each layer creates a new repre-

sentation of the input data:

$$\begin{aligned} h^{(0)} &= f^{(0)}(\mathbf{x}; \theta_0) \\ h^{(1)} &= f^{(1)}(\mathbf{h^{(0)}}; \theta_1) \\ y &= f^{(2)}(\mathbf{h^{(1)}}; \theta_2). \end{aligned}$$

The MLP as a whole is then a parameterised, composed, function:

$$f(x; \theta) = f^{(2)}(f^{(1)}(f^{(0)}(x; \theta_0); \theta_1); \theta_2),$$

where $\theta = (\theta_0, \theta_1, \theta_2)$. Here the $f^{(i)}$ are the layers of neurons, and the θ_i are their weights and biases.

MLP model

- Model:
  - $o_\theta = \phi_3(\mathbf{w_3}^\top \phi_1(\mathbf{w_2}^\top \phi_1(\mathbf{w_1}^\top x)))$
  - $\theta : \{\mathbf{W}\}$
- Loss function:
  - $L(\mathbf{x}, y; \mathbf{W}) = \frac{1}{2n} \sum_{i=0}^{n} (o_\theta - y)^2$
- Gradient of L wrt $\mathbf{W}$:
  - $\frac{\partial}{\partial W} L(.)$

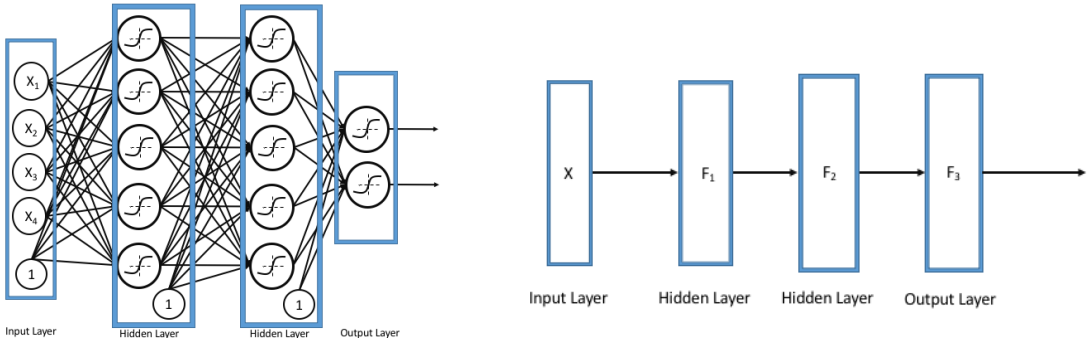
- biases omitted for brevity

Multilayer Perceptron — Layered representation

The MLP can be depicted in a more consolidated manner, where each layer is presented as a box and the connections between the layers as a single edge. This is shown in Figure 17. The advantage this gives us is that it helps us to more easily reason about complex models with many layers. Moreover, as we discuss in the following chapters we can have other kinds of layers of neurons such as convolutional layers.

Multilayer Perceptron — Backpropagation

We aim to train these models in the same way we trained the artificial neuron model: by gradient descent. In order to do this, we again compute the gradient of the empirical loss with respect to the model parameters through backpropagation. Here too it is useful to look at the model in a consolidated way: we write the computation graph in terms of the layers — see Figure 18. Now the edges carry vectors, and the notes are operations on those vectors. Since the layers consist of artificial neurons that are not connected to each other, we can calculate the total derivative of a layer the same way as we did for a single neuron back in Section 2.2. The only difference is that now our derivative is not a vector, but a matrix: the Jacobian matrix.



(a) An MLP with boxes around the neurons constituting the layers.

(b) The final consolidated representation of an MLP.

Figure 17: Looking at an MLP as a graph of layers provides a more abstract view of the model.

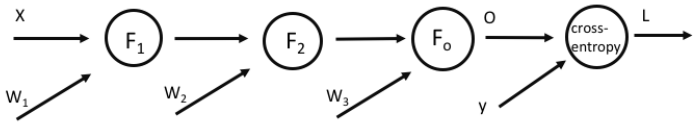


Figure 18: The computation graph in terms of layers.

Primer on the Jacobian matrix

A differentiable function $g : \mathbb{R}^n \to \mathbb{R}^m$ has at a point $\mathbf{x} \in \mathbb{R}^n$ a total derivative that is a linear map $dg_{\mathbf{x}} : \mathbb{R}^n \to \mathbb{R}^m$ that can be represented by its "Jacobian matrix"

$$J_g(\mathbf{x}) = \frac{\partial g}{\partial \mathbf{x}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_1}{\partial x_n}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_m}{\partial x_n}(\mathbf{x}) \end{pmatrix}.$$

These total derivatives and Jacobian matrices satisfy the following chain rule: if $g : \mathbb{R}^n \to \mathbb{R}^m$, and $h : \mathbb{R}^m \to \mathbb{R}^k$, then the total derivative of $h \circ g : \mathbb{R}^n \to \mathbb{R}^k$ and its Jacobian matrix at a point $\mathbf{x}$ are given by

$$\begin{aligned} d(h \circ g)_{\mathbf{x}} &= dh_{g(\mathbf{x})} \circ dg_{\mathbf{x}} \\ J_{h \circ g}(\mathbf{x}) &= J_h(g(\mathbf{x}))J_g(\mathbf{x}). \end{aligned}$$

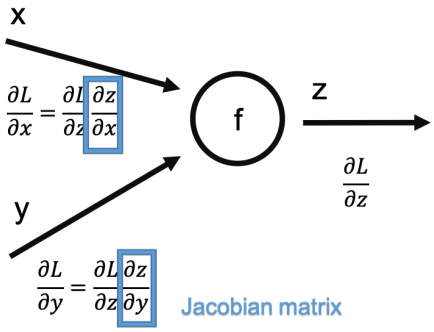


Figure 19: Derivatives of a layer.

If we now view a layer $F(\mathbf{x}; \theta)$ with n-dimensional input as an m-dimensional vector of functions

$$F(\mathbf{x}; \theta) = \begin{pmatrix} f_1(\mathbf{x}; \theta_1) \\ \vdots \\ f_m(\mathbf{x}; \theta_m) \end{pmatrix}$$

where $f_1, \ldots, f_m$ are the individual neurons with respective parameters $\theta_1, \ldots, \theta_n$ and $\theta_i = (w_{i,1}, \ldots, w_{i,n}, b_i)$, then its Jacobian matrix with respect to the parameters can be written as the following **block matrix**:

$$J_F = \frac{\partial F}{\partial \theta} = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{\partial f_2}{\partial \theta_2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{\partial f_3}{\partial \theta_3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \frac{\partial f_m}{\partial \theta_m} \end{pmatrix},$$

where $\frac{\partial f_i}{\partial \theta_i} = J_{f_i}$ is a $1 \times (n+1)$ matrix:

$$\frac{\partial f_i}{\partial \theta_i} = \begin{pmatrix} \frac{\partial f_i}{\partial w_{i,1}} & \cdots & \frac{\partial f_i}{\partial w_{i,n}} & \frac{\partial f_i}{\partial b_i} \end{pmatrix}.$$

Note that the off-diagonal blocks are 0-matrices because the neurons in our MLP do not share weights. This will be different when we get to convolutional layers. *Question: do we have the same for the Jaccobian matrix of the layer with repect to its input? Why or why not?*

Multilayer Perceptron — Backpropagation continued

Now lets see at what backpropagation looks like for an MLP with three hidden layers. Let L be our loss function, and let our MLP be given by $F_3 \circ F_2 \circ F_1$, where F_i is a layer with parameters θ_i (note that θ_i takes the role of θ in the primer on the Jacobian matrix above).As shown in Figure 20, we call the output of F_3 o, that of F_2 r, and that of F_1 q. We can then do back propagation as follows:

$$\begin{aligned}
\frac{\partial L}{\partial \theta_3} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial \theta_3}, \\
\frac{\partial L}{\partial r} &= \frac{\partial L}{\partial o}\frac{\partial o}{\partial r}, \\
\frac{\partial L}{\partial \theta_2} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial \theta_2}, \\
\frac{\partial L}{\partial q} &= \frac{\partial L}{\partial r}\frac{\partial r}{\partial q}, \\
\frac{\partial L}{\partial \theta_1} &= \frac{\partial L}{\partial q}\frac{\partial q}{\partial \theta_1},
\end{aligned}$$

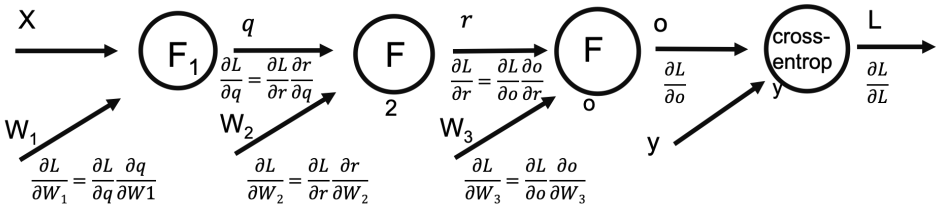


Figure 20: Backpropagation over the computation graph of an MLP with three hidden layers.

where we know how to compute the local gradients from Section 2.2 and the primer on the Jacobian matrix.

5 Model Design

So far we have seen how we can group artificial neurons together into layers, and stack layers on top of each other to get deeper neural networks. We have seen how we can build regression models and a model for binary classification, and we have seen how we can train these models by minimizing some loss function through gradient-descent. However, in the examples we have seen so far, we have brushed over some of the choices that were made. In this section we will zoom in on those choices that need to be made when developing a model. In later chapters you will learn about more techniques such as different types of layers, and stochastic regularization methods, that all come with their own design decisions, but for now we will focus on the following decisions and considerations:

- What loss function we are going to minimize during training
- What model design we are going to use (number of neurons and layers, activation function for the hidden layers)
- What activation the output layer will have
- What training parameters we will use for the gradient descent algorithm (e.g. learning rate)

To to be able to make these decisions in an informed way, we need to:

- Specify the input to our model

Figure 21: The MNIST dataset. Pixels with a higher value are depicted here as darker.

- Specify the output of our model (and how we are going to interpret it).

We will go over these decisions using classification on the MNIST dataset shown in Figure 21 as an example.

5.1 The MNIST dataset

We want to write a model that can take pictures of handwritten digits as its input and output what digit is in the picture. The dataset we are given, the MNIST dataset, consists of 28×28 gray-scale images. Each pixel has a single number associated to it indicating its brightness. These values are integers between 0 and 255. The images all belong to one of ten classes, an image belonging to class i meaning that the image contains the digit i. A selection of images from the MNIST dataset is shown in Figure 21.

If we want to write a neural network that can classify these images, a good start is to look at the data. Neural networks tend to perform better on data lying in a range of magnitude $O(1)$. Large numbers in the data make that a small difference in parameters in one layer will have a very large impact on the input the next layer receives. This is detrimental to its ability to be trained by gradient descent. On the other hand, when all data is very small, e.g. in a range of $(0, 10^{-3})$, the weights and biases of the network need to be very large to come to output of the right size, again negatively impacting the ability of the model to by trained by gradient descent. Keeping this in mind we will pre-process the data by transforming the integers to floating-point numbers, and scaling to the interval $[-1, 1]$ by the following map:

$$x \mapsto \frac{x - 127.5}{127.5}.$$

Besides that, we will reshape the 28×28 arrays to vectors of size 784 to feed to our neural network.

5.2 Output and Loss

Now that we have the input ready, let's look at the output. As we discussed in Section 3, doing classification with neural networks is typically done by outputting a probability distribution over the classes. To get matching labels, we will perform a so called *one-hot encoding*: we map an integer i to a vector e_i of length 10 where the i^{th} element is 1 and all other elements are 0, as shown in eq. (3) — here $\delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$

$$i \mapsto e_i = (\delta_{i,j})_{j=0,\dots,9}. \tag{3}$$

In order to output a probability distribution over our ten classes, we will use the softmax function. To do this, we take our last layer of neurons to have ten neurons, each of which will correspond to its own class. The output $(v_0, \dots, v_9)$ of these neurons will then be fed to the softmax function

$$\text{softmax} : (v_0, \dots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\dots,9}.$$

We want the output distribution for an image to give a lot of mass to the correct class, and very little to all other classes. That is, we want the likelihood of the correct answer to be as high as possible. To rephrase this as a minimization problem: we will aim to minimize the negative log-likelihood. Now if we have label y with one-shot encoding e_y, and we have a probability distribution $(p_0, \dots, p_9)$ over our classes (where all the p_i are positive, and their sum is 1), then the likelihood of y is

$$p(y) = \prod_{i=0}^{9} p_i^{e_{y,i}},$$

so the negative log-likelihood is the *categorical cross-entropy*[7]

$$\begin{aligned} L(y, p) &= -\sum_{i=0}^{9} e_{y,i} \log(p_i) \\ &= -\log(p_y). \end{aligned} \tag{4}$$

Let us take a step back and look at how we have come up with this loss function. Just like with the binary cross-entropy, we output a probability distribution, and we intend to maximize the likelihood of the true labels. This way, the loss function follows kind of naturally[8]from the fact that we are trying to predict a probability distribution, and from the type of distribution. When trying to predict continuous outcomes with a regression model, we based our loss simply on a distance between our predictions and the correct labels. We could however view this in the same light: if we view the outcome of the model not as a single value, but as the center for a normal distribution with fixed variance, the resulting average negative log-likelihood would (up to an additive constant which has no effect on training with gradient-descent) be a scaled version of the mean squared error.

In general there are many loss functions you can use to train a neural network. What loss function does work, and what doesn't depends strongly what the output of your network represents. It is

[7]Cross-entropy is really an information-theoretic measure of error between two probability distributions. The categorical cross-entropy in eq. (4) is a measure of how far we are off when using the incorrect probability distribution p instead of the true (deterministic) distribution e_y.

therefore important to consider what kind of values you are trying to predict, and how you are going to interpret the output of your neural network. Moreover, it is important to remember that if you want to train your network using gradient-descent or other gradient-based methods, the loss function needs to be differentiable. Additionally it is important to have some sort of monotonicity in that predictions that are further off target should give a higher loss to prevent getting stuck in low quality local minima. Most of the time one of the following three losses will be an appropriate choice:

- Mean Squared Error if you are dealing with a regression task;
- Binary Crossentropy if you are doing with binary classification;
- Categorical Crossentropy if you are doing multiclass classification.

Note also the role the final activation plays: with regression our final activation was linear, with binary classification it was a sigmoid function, and now with multiclass classification it is a softmax function. In general it is important to use an activation in the output layer that allows the model to give good predictions. If the values you are trying to predict lie in a small and fixed range, a linear activation will make it hard for your model to stay within that range. On the other hand, if there is no clear bound on the output — like when doing regression — a bounded activation function like a sigmoid will not allow your model to approximate the correct outcomes.

5.3 Network Design

We have pre-processed our data, we have decided upon the type of output we want our model to give, chosen an activation for the final layer, and defined a loss function. Now it is time to design the neural network itself. How many layers should we use? How many neurons should each layer have? And what activations are we going to use?

A large part of this is simply trial and error. Adding or removing a single layer can have large impacts on model performance, as can adding or removing neurons to/from layers. There are a couple of things however that you can keep in mind to help you make informed guesses.

Vanishing or Exploding Gradients

One of the main problems that can make training neural networks difficult is the gradients getting either too large (exploding) or too small(vanishing). This is because, as we saw in section 4, composition of layers results in a product of derivatives. If the derivatives are all smaller than 1, the product quickly vanishes to 0, but if they are all larger than 1 the product grows exponentially.

Activation Functions

Originally the go-to activation function for hidden layers were sigmoidal functions such as the logistic sigmoid and the hyperbolic tangent. These functions however suffered greatly from the

[8]The logarithm is really used here for numerical stability. Throughout this course you will see the theory of probability distributions being use in combination with neural networks time and again, and often the logarithm will play a role. This is not only because of that numerical stability, but also because it simplifies various expressions, and because many information-theoretic concepts related to probability distributions are defined using the logarithm.

vanishing gradient problem, making it hard to train deep networks with sigmoidal activations in the hidden layers. This shows that when choosing an activation function it is good to keep the vanishing/exploding gradient problem in mind. One approach to prevent this, is to take an activation function that is the identity on the positive half of its domain. The best known example of this is the Rectified Linear Unit (or ReLU) given by

$$\text{ReLU}: \ x \mapsto \max(0, x)$$

and depicted in Figure 22a. Because the derivative of the ReLU function on the positive half line is precisely 1, this helps against vanishing or exploding gradients. As a consequence, the advent of the ReLU allowed for much faster and better training of neural networks than before. However, the ReLU comes with its own problems when it comes to gradients: since it is constant for negative input, a neuron giving negative output to its ReLU will have a zero gradient from which it can never recover through gradient descent. This is called the "dying ReLU" problem and the neuron in question is said to be dead.

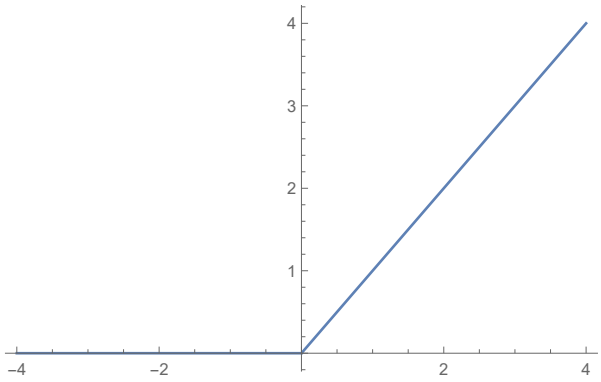
This dying ReLU problem lead to the introduction of a number of other activation functions such as the

- Leaky ReLU: $x \mapsto \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$
  - hyper parameter $\alpha \in (0, 1)$
  - called "parameterized ReLU" when α is learned through gradient-descent
- Exponential Linear Unit (ELU): $x \mapsto \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$
  - has several variations such as the Parameterized Exponential Linear Unit, and the Scaled Exponential Linear Unit.
- Softplus: $x \mapsto \log(1 + e^x)$
  - is a smooth approximation to the ReLU
  - its derivative is the logistic sigmoid which can be thought of as a smooth approximation to the Heaviside step function which is the derivative of the ReLU.

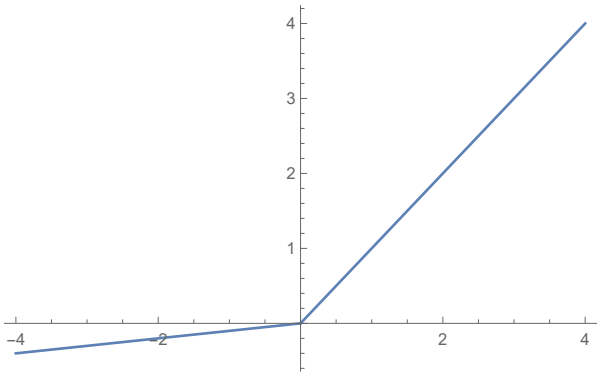
These all solve the dying ReLU problem, but for some of these the downside is that their gradients are more costly to compute.

Depth and Width of the Network

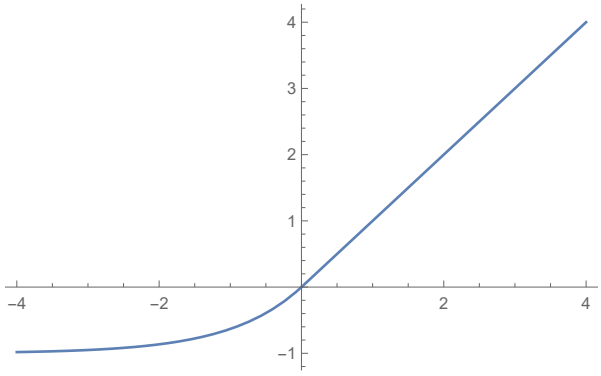
When you hear about neural networks it often seems that bigger is better. The state of the art on many tasks for many data sets consists of huge neural networks with millions of neurons arranged in a large number of layers. These models have managed to develop highly efficient representations of the data that in turn have allowed for breakthrough performance on many tasks. Does this mean, however, that we should also make our classifier on the MNIST set dozens of layers deep? The answer to that question surprisingly is a resounding *no* for several reasons.



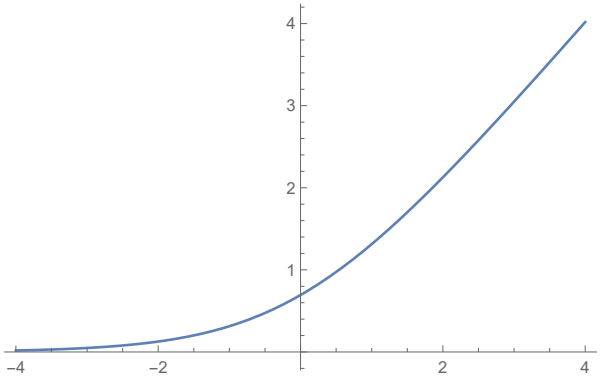
(a) The graph of a Rectified Linear Unit



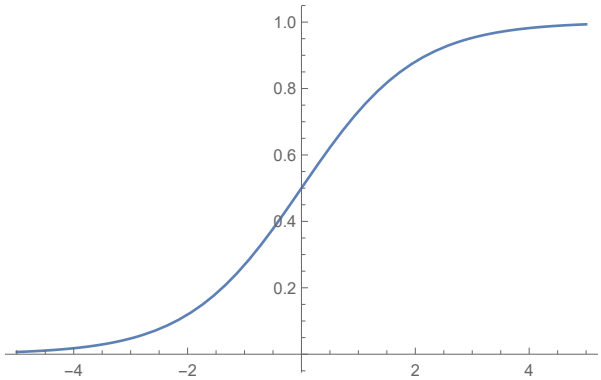
(b) The graph of Leaky-ReLU.



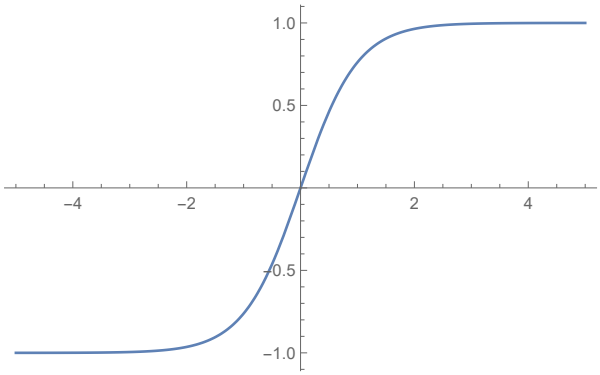
(c) The graph of an Exponential Linear Unit.



(d) The graph of the softplus function.



(e) The graph of the logistic sigmoid function.



(f) The graph of the hyperbolic tangent function.

Figure 22: The graphs of various commonly used activation functions.

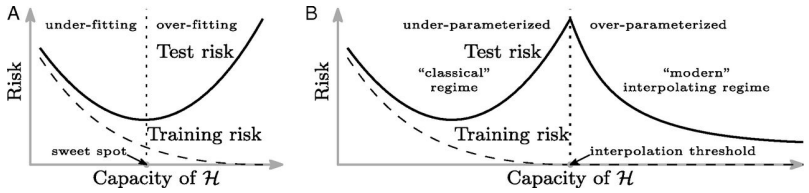


Figure 23: Bias-variance in overparametarize models
[9]

The first and probably most important reason lies in the layers we are using: the large neural networks you hear about typically combine different kinds of layers, such as convolutional layers, that allow for very deep models, and combine them with a myriad of techniques that help in training neural networks. Later in this course we will learn about some of these layers and techniques, but for now we are using only so-called "fully connected" layers where the neurons do not share any weights, and where the neurons within a layer have no connections to each other. These layers don't really allow for very deep networks and as mentions as recently as in 2017 many of the best performing networks of this kind were only four layers deep. Several techniques can help with training deeper networks but it is good to keep in mind that especially with fully connected networks more is not always better.

Another reason why more is not necessarily better is resources. Especially in image processing the state of the art consists of huge (convolutional) neural networks that have been trained over long periods of time on very expensive hardware. However, when you want to solve an actual problem using neural networks, you might not have similar resources available. This means you will likely have to make a trade-off between accuracy of the model and the cost of training (and even using) it.

Finally a problem that often occurs when using very powerful (deep) models, especially to predict relatively simple data, is that of *overfitting*. This is when a network is making predictions based on the peculiarities of the specific data it is trained on instead of on general patterns. The model might then perform extremely well on the data it is trained on, but will give much lower quality predictions on new data. This is a common problem in deep learning, and there is a large number of techniques that try to mitigate it, but one of the most effective (albeit rigorous) strategies is to simply reduce model complexity. Often that comes down to reducing the number of neurons in the model. Most well recognized deep neural network models are, however, overparameterized (have larger number of parameters than needed to capture the complexity of the map). The effects of the overparameterization of models in machine learning is an active field of research. Recent work indictes that the usual bias-variance trade-off has significantly different behaviour in these modelsfig. 23.

So, when designing deep neural networks we do not consider necessarily only the complexity of the map that the model needs to achieve, but also how the number of parameters affects the optimization process. In many cases an overparameterized model allows for the SGD optimization to be much more efficient in finding parameter values that yield good generalization properties. This does not mean that these models do not overfit, but on the contrary that controlling the overfitiing with well designed regularisation techniques is a key component of the design.

[8] Here too the vanishing and exploding gradient problems play a large role: the more layers you have, the more factors there are in the overall derivative, thus the more opportunity it has to vanish or explode.

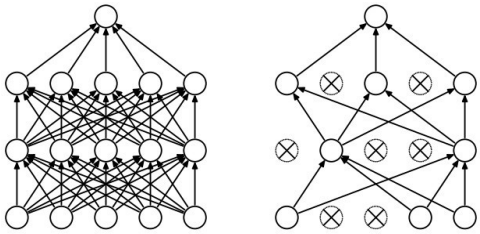


Figure 24: Dropout regularization

5.4 Regularization

Regularization techniques are one of the key breakthroughs that allowed for the success of deep learning. They both allow for significantly larger number of parameters as well as training the model longer. Both of these aspects of developing deep neural network models are key to achieve the performance that they have on high dimensional data.

Regularization can be applied to both the values of the weights as well as the values of the activation in neural networks.

Well known techniques for regularization in machine learning such as L_1 and L_2 regularization also apply in neural networks. The L_1 (LASSO) results in sparse values. This type of regularization may be particularly desirable on the activations of the neurons if we would like to have a representations of our data that is sparse. We revisit this idea when we discuss sparse autoencoders in the following chapters. On the other hand L_2 (rigde) regularization will induce lower values. When applied to the weights, it limits the capacity of the model and hence it can control for overfitting.

One of the most successful regularization techniques in deep learning is dropoutfig. 24. Dropout is typically applied to the activations of the neurons. Dropout is applied during training of the model by randomly setting a set fractions of the activations of neurons in a particular layer to zero. The idea behind dropout is to prevent the neurons in one layer to specifically depend (or co-adapt) to the activations of some of the neurons in the previous layer. In general, we would like to avoid store the datapoints in a form of memory and then recall them during inference, but rather to form an efficient representation of the data that will allow for generalization on the test set. When neurons are prevented from co-adapting to each other[10], unique pathways of activation to specific training datapoints are disabled. In contrast the neural network, starts to develop a distributed representation of the data. The distributed representation[11] allows for compositionality of the representation. Features from the lower layers can be composed to form high level features in the subsequent layers that results in highly efficient representation.

Note: Many other commonly used regularization methods are ommitted due to the space in these lecture notes. The "Deep Learning" book (Ian Goodfellow and Yoshua Bengio and Aaron Courville (2016), chapter 7 is on regularzation. It is work to further highlight a the more recent method, the BatchNormalization[12] that has enabled significant success on many modern neural networks.

[10]Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." The journal of machine learning research 15.1 (2014): 1929-1958.

[11]Hinton, Geoffrey E. "Distributed representations." (1984).

[12]Ioffe, Sergey, and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." International Conference on Machine Learning. 2015.

This method particularly improves the training process, by controlling how the distribution of the activations of the layer changes as they are exposed to new data.

5.5 Initialization

The initialization of the parameters of the neural networks are also an important aspect of the model design. As the training of these models is a stochastic process, the initial values of the parameters can result in significantly different outcomes. This is also an active area of research and out of the scope of this document.

Note: For further reading on initialization please refer to Grolot[13] and He[14].

6 Training

The training process of neural networks is stochastic and comes with its one set of configuration parameters and design decisions. One of the most salient parameters is already in the update step of gradient descent, the learning rate eq. (5).

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\mathbf{x}; \theta) \tag{5}$$

Recent methods actually adapt the learning rate parameter rather than keeping it at a constant rate. Specifically, with the introduction of the momentum technique eq. (6). Momentum allows the model to take updates with larger steps if we have been going in the same direction for a while, and smaller steps when we are changing directions, which corresponds to the notion of momentum in mechanics.

$$\begin{aligned} v &\leftarrow \gamma v - \alpha \nabla_\theta L(\mathbf{x}; \theta) \\ \theta &\leftarrow \theta - v \end{aligned} \tag{6}$$

Without going into further details, some of the commonly used variations of the SGD algorithm that use such techniques are: Nestorov momentum, AdaGrad, AdaDelta Adam and RMSProp.

One other key parameter for which a decision has to be made is the batch size. The subsampling of the dataset that SGD does, introduces a type of noise in the updates that helps avoiding local minima during training. The more we subsample, or the smaller the batch size, the larger the amount of noise. So, choosing the right batch size has significant influence on the training. One other consideration is that for many high dimensional cases and large model the computational resources needed can actually limit the batch size. To implement the update step, we have to hold in memory the activations of the forward pass to be able to compute the backward pass. We do this typically in parallel for the whole batch. Using large batches can then lead to out of memory conditions.

[13] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

[14] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." Proceedings of the IEEE international conference on computer vision. 2015.

Deep Learning (2IMM10) - Word embedding (word2vec)

Vlado Menkovski

Spring 2020

3 Word embedding

The learning outcomes of this chapter:

- Able to develop representation of large (but countable) sets using neural network models

3.1 Introduction

In this chapter we consider tasks where the input data can only take discrete values — that is, our input variables x take values in some set V where this set has a fixed size $|V| = N$. There are many examples where the data has such discrete structure. For example, if we were to use a neural network to process DNA or RNA sequences, the input is a sequence that consists of only four different elements corresponding to the nucleotides. Similarly if we would process data that encodes the genes of proteins, we would have a fixed set of different components, the aminoacids.[1] Analogously, when dealing with natural language, the sentences we process are made up of a (large but) finite set of different words, and these words in turn are built by a finite set of different characters.

In each such case, we need to develop a way to represent or encode the data such that it can be processed by our model. For example, we can enumerate the discrete elements and encode them as integers or binary numbers.

However, in cases where the number of elements in the set, N, is very large, particularly compared to the number of datapoints in the dataset, it is very useful to have a representation that allows for better generalization. More specifically, we would like to have an encoding that reflects the relationships between the elements in the set, because in that case the model can learn to behave similarly for similar elements in the set. This in turn would allow us to be much more efficient with the amount of training data that we need.

In this chapter we discuss how neural network models can be used for learning representations of large sets of elements such as words in a natural language.

[1] A set of only 22 different "proteinogenic" amino acids form the building blocks of all proteins in all known lifeforms.

3.2 Example: Text prediction

Let us consider the task of predicting the next word in a sentence. Our model would need to take a sequence of words as its input and produce a suggestion for the most likely next word — see Figure 1.
Each of the words needs to be encoded and presented to our model, such that the model can produce the next word . For simplicity we can assume that the input is a fixed-length window of words rather than a variable-length sequence of words.
Suppose we would encode the words with sparse vectors. Specifically, with binary vectors with length $|V|$ as in Figure 2.
If we make the model do predictions for

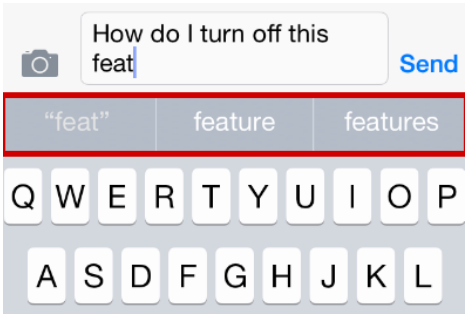


Figure 1: Predict the next word

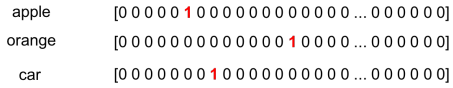


Figure 2: Sparse words

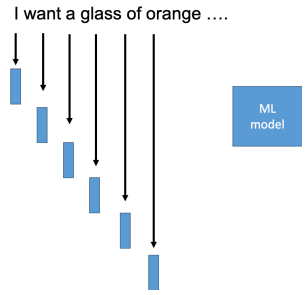
- "I want a glass of orange ..."

we can depict the process as in Figure 3. What happens if we instead want to do predictions for

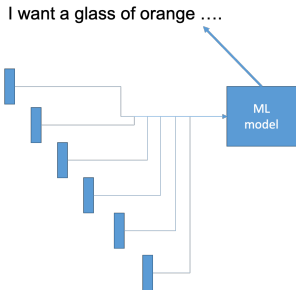
- "I want a glass of apple ..."

The current representation of the data as sparse encoding does not provide any possibilities for model to re-use what it has learned for the word 'orange' on the input with the word 'apple'. This is not very efficient as we would need to have all sentences where both apply double in the training set — once with "apple" and once with "orange".

In contrast if we would have a representation of 'apple' and 'orange' that allows for encoding the commonalities between the two words (e.g. both of them are fruits, but also types of juice) our model could benefit from this as it can learn to actually map the common properties to a target rather than the individual words. For example, all fruits that can be made into a tasty glass of juice could share a specific part of their encoding. So, if we have an example of a sentence with one, then the model would be able to generalize what it learns from that to the other fruits.



(a) The model takes the sentence as a sequence of one-hot vectors.



(b) The model processes these sparse vectors.

Figure 3: Predicting the next word using sparse encoding.

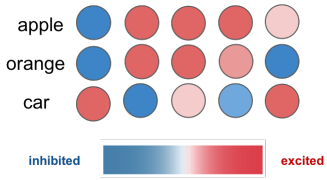


Figure 4: Distributed word representation

3.3 Distributed word representations

When developing a word embedding we aim at an encoding that captures different features, or aspects, of the words. This is referred to as *Distributed word representations*. Such a representation can describe a large number of words efficiently.

Let us consider the following example:

- king ← male
- queen ← female
- man ← male
- woman ← female

Here the representation of 'king' and 'man' would share the property of 'man' as would the representation of 'queen' and 'woman' share the property of 'female'.

If we had such a representation then after learning about the word 'aunt', we would be able to learn 'uncle' by adjusting the 'aunt' representation as

$$\text{'uncle'} = \text{'aunt'} - \text{'female'} + \text{'male'}.$$

To be able to achieve such distributed representation we first represent the words in an **embedding space**, such that each word is represented by a d dimensional vector in $\mathbb{R}^d$ space. Here each of the axes in the space can take different roles, see Figure 4.

If we manage to create such a representation, the meaning of the individual words will be distributed over the different dimensions of the representation. Furthermore, the Euclidean distance in this space will capture the semantic distance of the words.
How can be develop such a representation?
One important property of natural text is that the meaning of a word is related to its context, so that words that appear next to similar words have similar, or related, meaning . [2]In the work by Benio et al. this property was used to develop the "Neural probabilistic model".

[1]Take the orange and apple example from earlier. Both words occur in similar contexts such as "I am going to eat a ..." and "she was drinking ... juice". If another word systematically pops up near words such as "fresh", "eat", "juice", "drink", "grow", and "sweat", chances are it's a word for some kind of fruit.

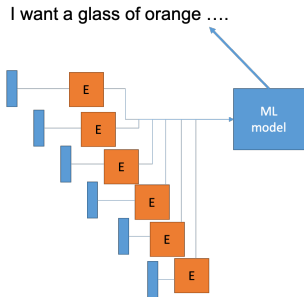


Figure 5: The model first uses an encoding map E to map the one-hot vectors to their dense representations, and then uses these dense representations to make predictions.

The approach is as follows: we try to learn a representation that captures semantics based on context by learning a probability distribution over sentences factorized as

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}),$$

where w_t is the t-th word in the sequence, and $w_i^j = (w_i, \ldots, w_j)$. We simplify this by only looking at the last $n-1$ words:

$$\hat{P}(w_t | w_1^{t-1}) \approx \prod_{t=1}^{T} \hat{P}(w_t | w_{t-n+1}^{t-1}).$$

The idea is then to write this as a function in the following way:

$$\begin{aligned} \hat{P}(w_t = i | w_{t-n+1}^{t-1}) &= f(i, w_{t-n+1}, \ldots, w_{t-1}) \\ &= g(i, E(w_{t-n+1}), \ldots, E(w_{t-1})), \end{aligned} \tag{1}$$

where $E(w)$ is the embedding of a word w into our embedding space — see Figure 5 for a schematic representation. In practice we use an embedding matrix that maps the one-hot encoding of the words to a dense representation of the data. For example, if our vocabulary V has size $|V| = 10^4$, and our embedding space is 300-dimensional, the map E is given by a 300×10^4 matrix — see Figure 6.

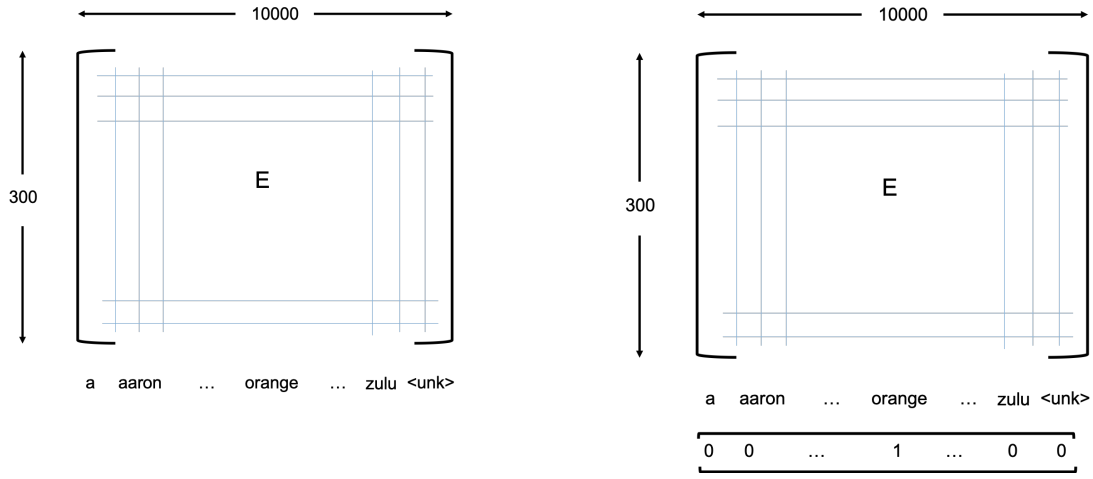
In the work of Bengio et al.[3] the function g was implemented as a feed-forward neural network with, since we want the output to be a probability distribution, a softmax function for its output layer. Since E is a differentiable function, we can train E and g simultaneously using back propagation and a gradient-based optimization algorithm such as SGD.

3.4 Further developments

The model we discussed above can be used on its own, but far more importantly, the word vectors it has learned can be used for downstream tasks. I.e. we can use the (trained) embedding matrix as the first layer to another model[4]. If we keep in mind that our real goal is not to use the model

[3] Bengio, Yoshua, et al. "A neural probabilistic language model." Journal of machine learning research 3.Feb (2003): 1137-1155.

[4] This is an example of *transfer learning*, a topic we will discuss more thoroughly in the next chapter.



(a) If our vocabulary has size 10 000, and our embedding space is 300 dimensional, our embedding matrix E is of size $300 \times 10\,000$.

(b) We can use the embedding matrix to map a one-hot encoded word to the corresponding dense representation.

Figure 6: Embedding matrix

we train directly, but just to learn a word embedding for a later task, what kind of model do we ideally want for g in eq. (1)?

The cheaper g is to train, the larger the vocabulary we can embed and the more data we can handle with a limited amount of resources. Let us therefore look at two techniques that simplified the model, allowing us to train word embeddings for much larger vocabularies, or on much more data: *Continuous Bag Of Words* (CBOW) and *Skipgram*, both introduced in "Efficient Estimation of Word Representations in Vector Space", Mikolov et al. (2013).

These two models consist of two matrices, a word embedding E and a context embedding, C. The context embedding, C is essentially the weights matrix of the output layer of g. In other words, these models remove all hidden layers from g. Graphical representations of both the original Neural Probabilistic Language Model and the two improvements we will discuss in this subsection can be found in Figure 7, Figure 8, and Figure 9 at the end of this chapter.

Continuous Bag of Words

The training objective with CBOW is similar to the one in Section 3.3: we try to predict words based on their context. However, instead of using only previous words, we try to predict based on a window around the word. That is, we try to learn

$$P(w_t \mid w_{t-L}, \ldots, w_{t-1}, w_{t+1}, \ldots w_{t+L}).$$

In order to do this, we embed all the context words using the same word embedding

$$u_i = Ew_i$$

where E is a matrix as before. Now instead of putting $(u_{t-L}, \ldots, u_{t-1}, u_{t+1}, u_{t+L})$ through a hidden layer of a neural network, we simply compute their average

$$\hat{u} = \frac{u_{t-L} + \ldots + u_{t-1} + u_{t+1} + \ldots + u_{t+L}}{2L}$$

to get some kind of summary of the context.

This context vector is then transformed to a score vector

$$z = C\hat{u},$$

which is then put through a softmax function in order to obtain our estimated probability distribution

$$\hat{P}(w_t \mid w_{t-L}, \ldots, w_{t-1}, w_{t+1}, \ldots w_{t+L}) = \text{softmax}(z).$$

Question: what loss function should we use to train this?

Skipgram

With Skipgram, the training objective is in a way of reversed compared to CBOW: instead of guessing a word based on its context, we try to learn the context based on a word. That is, we try to learn the probability distributions of words close to a given word. The way we do this is by setting up our training data in a specific way.

To generate the training data, we specify a maximum window size L_{max} and then we go through our available sentences:

1. for a word w_t in the sentence we pick a window size $1 \leq L \leq L_{max}$
2. we add the the word pairs $(w_t, w_{t-L}), \ldots, (w_t, w_{t-1}), (w_t, w_{t+1}), \ldots, (w_t, w_{t+L})$ to the dataset.

Our training set then consists of a (large) number of word couples (w_i, w_j) and our training objective is to predict w_j based on w_i. *Question: how does this force the model to learn the probability distribution of the context of w_i?*

The network itself is now even further simplified compared to CBOW: we only get one word as the input so the averaging is no longer necessary. This means we do prediction in the following way:

1. We embed the input word using our embedding matrix
$$u = E\, w_i$$
2. Based on the embedded word, we calculate a score vector for the context, i.e.
$$z = C\, u$$
3. We apply the softmax function to the score vector to get our estimated probability distribution
$$\hat{P}(w_j \mid w_i) = \text{softmax}(z).$$

This setup forces our word embedding E to give similar output for words that appear in the same contexts since such words will need to result in similar context distributions. For example, "orange" and "pear" both occur frequently near words like "fresh", "juice", "eat", "drink", "tasty", "buy", and "healty" but much less frequently near words like "headphone", "war", "guitar", or "chair". Since our way of training forces the model to give high probability to frequently occurring combinations, and low probability to rarer combinations, the output distribution for "orange" and "pear" will need to be similar, forcing the word embeddings for these words to be similar too.

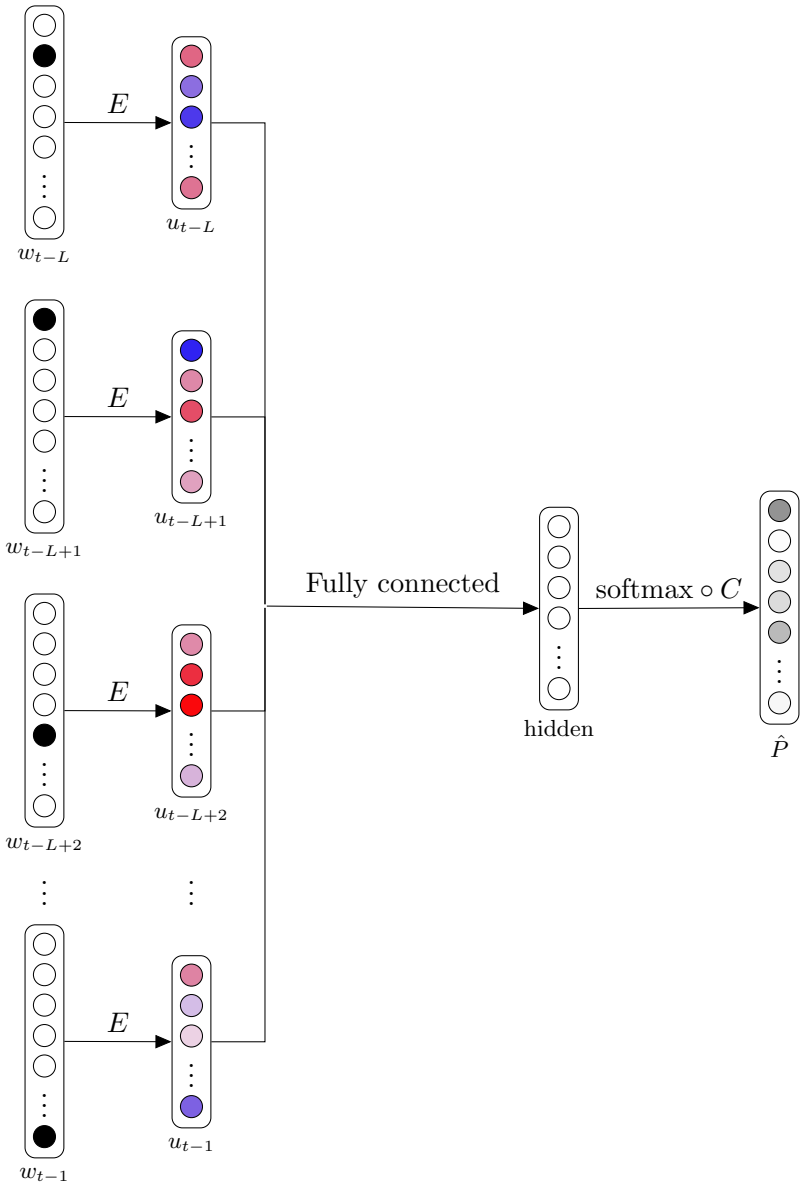


Figure 7: Neural Probabilistic Language Model. *Question: what probability measure is being estimated?*

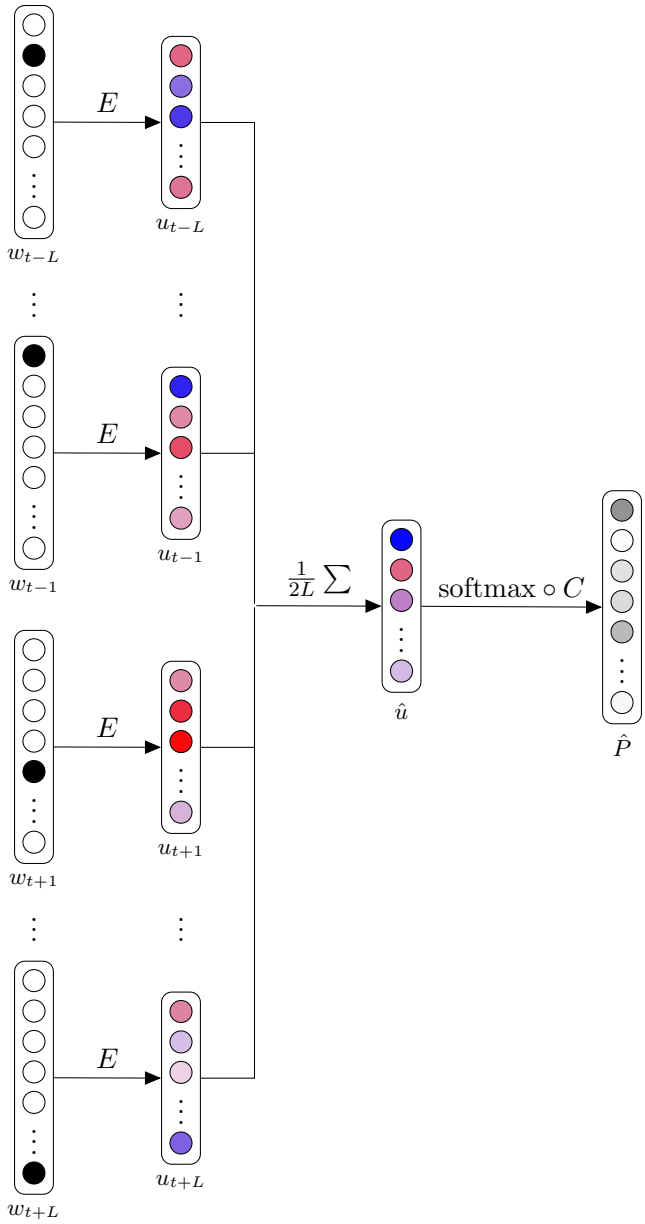
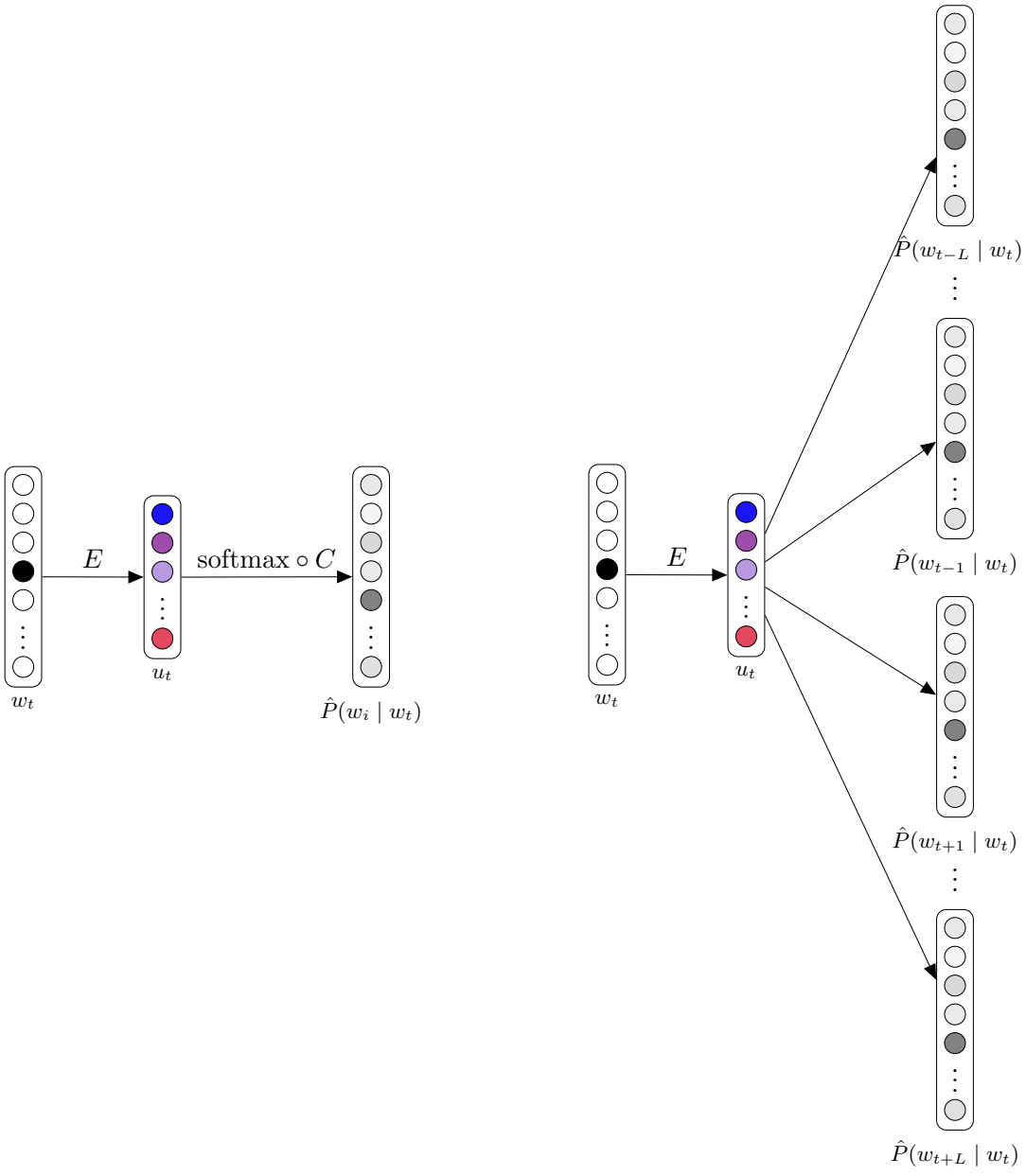


Figure 8: Continuous Bag of Words. *Question: what are the differences with Figure 7? And what are the similarities?*



(a) Skipgram

(b) Skipgram

Figure 9: Two representations of Skipgram. *Question: why are these both accurate representations of the same model?*

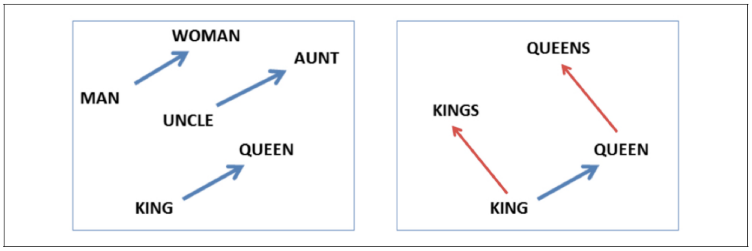


Figure 10: Properties of word embedding

3.5 Properties of the embedding

Assuming that our model has succeeded to develop a distributed representation of the words in the vocabulary we can expect certain properties of this representation. One such property is that we can compute analogies such as: "man to a woman" is as a "king to a queen" fig. 10.

In principle we can use these analogies to search for a word that should have the desired properties. Again in the example: "man to a woman" is as a "king to a ?", using the embedding space we would like to find the word queen. Using the embedding vectors for "man", "woman", and "king" (e_{man}, e_{woman}, e_{king}). We can search for a word that is closest to the algebraic expression: "king" - "man" + "woman" as in eq. (2).

$$w = \arg\min L(e_w, e_{king} - e_{man} + e_{woman}) \tag{2}$$

where L is a distance metric in the embedding space.

Deep Learning (2IMM10)
Spatially distributed data (CNN)

Vlado Menkovski

Spring 2020

4 Models for spatially distributed data (CNN)

The learning outcomes of this chapter:

- Be able to develop models for data with local spatial correlations using convolutional neural networks

4.1 Spatially distributed data

We refer to data as spatially distributed when the data manifests with useful correlations of the features based on their position relative to the other features. For example if we take image data where the features are the pixel color and intensities. We can observe patterns such as strokes and edges based on locally correlated groups of pixels (Figure 1).

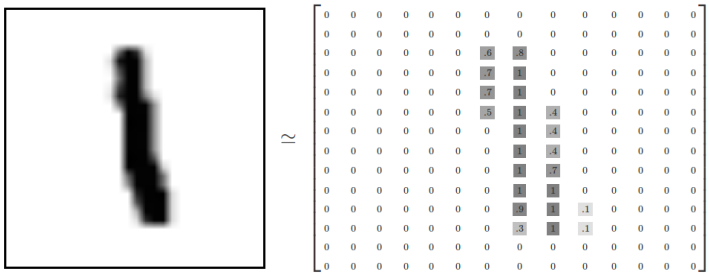


Figure 1: MNIST image with an array representation

For such data it is important to preserve this information and hence we structure it as an image. In contrast you can think about a table as a data structure, where each column is a feature. In a table we typically to not expect that information will be lost if the column are shuffled, but in an image we do.

Images can have different dimesionality. Photographs are typically two-dimensional, however there are images of one dimension, such as speech (Figure 7) as well as higher dimensional image (such as Magnetic Resonance Image or Computer Tomography medical scans).

Note: a color 2D image is typically organized as a 3D array where the third axis holds the encoding for the different colors of the pixels. In an RGB encoded image, we would have one 2D image for the red color of the pixels one for the green color and one for the blue color. Even though the image is represented in a 3D array we still refer to it as a 2D image because the spatial information is distributed in 2 dimensions (i.e. we can re-arrange the color planes without losing information).

Spatially distributed data often also has correlations that are localized in small regions of the image. For example to detect a bicycle in an image it would be useful to have a detector for a wheel. This detector could then be used to detect both of the wheels and combined with a detector for the frame could be combined in a detection mechanisms for bicycles. Note, however, that a wheel can take a small part of an image and can appear in different locations. Neural networks are particularly well suited for building such hierarchical representations, however the MLP architectures discussed so far are not efficient at detecting local patterns in images. In this chapter we study the convolutional neuron the the convolutional neural network models that use this mechanisms to learn to detect local patterns more efficiently.

4.2 Convolutional neural networks

Convolutional neuron

Let us consider the following task. We develop a model that detects the sequence '1011' in a 1D image (Figure 2). We can formulate this as binary classification. With a single artificial neuron (Figure 3) we could develop a model to detect the pattern. However, since this pattern is shorter than the length of the image it could in principle appear in different locations in the image. The problem with our single neuron is that it looks at the whole image. We also refer to this type of neurons as 'fully connected'. To detect the pattern at different locations we would need more neurons, each focusing on a different location. Then would need a second layer with a neuron combining the output of the neurons or the first layer. The second layer neuron would basically need to implement the binary 'OR' operation to test whether any of first layer neurons detecting the pattern.

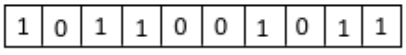


Figure 2: An input sequence in which we want to detect and locate the sub-sequence 1011.

This model would work as expected, but it will also have a number of parameters that is higher than what we could have optimally. It would be much more efficient to have a neuron that does not look at the whole image but has a narrower 'field of view', ideally as wide as the pattern.

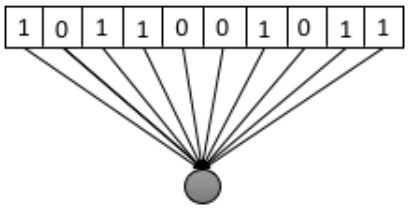


Figure 3: The way a regular artificial neuron sees its input.

In fig. 4 such a neuron is depicted. This neuron sees only a part of the image. For a model as this to process the whole image we would need to slide it across the image. As such we can re-use it on different locations. Such as neuron is referred to as a convolutional neuron. The output would then not be a single activation, but rather an activation at each location. This 'activation map' would then need to be processed by the second layer that would make the final decision. In our case we can acutally use the same second layer neuron as before. A single neuron that implements the binary OR operation on the output of the convolutional neuron.

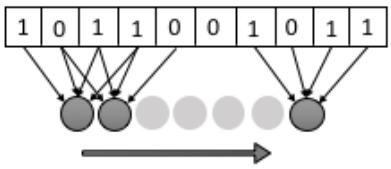


Figure 4: A schematic overview of our neurons field of view sliding along the input sequence.

The important aspect is that the convolutional neuron re-uses the paramters at each location and as a consequence the new model will have significantly less parameters than the fully connected model.

As with the original artificial neuron, the convolutional neuron has a weight for every input in its

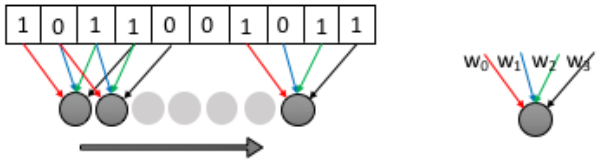


Figure 5: An illustration of the neuron operation described in eq. (2).

field of view. These weights together are called the *kernel* of the neuron, and the size of its field of view referred to as its kernel-size. The operation the neuron implements can be described in terms of the mathematical operation of "convolution". Convolution of two sequences is typically[1]defined as follows:

$$\begin{aligned}(a * b)_n &= \sum_{i=-\infty}^{\infty} a_i b_{n-i} \\ &= \sum_{i=-\infty}^{\infty} a_{n-i} b_i\end{aligned} \tag{1}$$

where finite sequences are padded with zeros, and convolution of higher dimensional arrays is defined analogously. A visual representation of this can be seen in Figure 5. Note that the second equality means that convolution is a commutative operation. If the neuron from our example has kernel $k = (w_3, w_2, w_1, w_0)$, and has bias b and activation function ϕ, then if we apply it to an input sequence $x = (x_0, \ldots, x_N)$, the output is

$$o_\theta(x) = \phi(x * k + \mathbf{b}), \tag{2}$$

where $\mathbf{b}$ is the vector with b at every index. Note that in order for the weights to be applied as in Figure 5 we have arranged them in reversed order to form the kernel.

The main goal of using convolutions is to re-use parameters of the model which in turn makes the model significantly more efficient both for training and inference. As depicted in Figure 5 the parameters w_0, w_1, w_2, and w_3 on the corresponding colored edges (red, blue, green and black) are re-used at each position where the neuron is placed during the convolution operation. In contrast, if we used a different neuron for each location, or one neuron taking the whole image as input, the number of parameters needed to achieve the same detection would be significantly larger as the parameters cannot be re-used.

This has a very significant impact on the success of the training such models. Not only because the model is less efficient, and we need to train more parameters, but we would also need to have examples of the pattern appearing in all locations in our training dataset. This is a key challenge in many settings, as we cannot expect to have training data available that covers the entire natural distribution of the data. Therefore, one of the main challenges of Deep Learning is to develop methods that are efficient in generalizing from a small number of examples.

[1]In physics, mathematics, and engineering, multiplication is typically denoted by $\cdot$ (`\cdot` in LaTeX) and $*$ is used for convolution. In some computer-science literature one can also find $\star$ being used to denote convolution. Outside of computer-science this $\star$ operation is usually used to denote the related cross-correlation operation, which is essentially convolution with the order of weights in the kernel swapped. Fun fact: TensorFlow actually performs cross-correlation instead of convolution.

Figure 6: A selection of images form the MNIST dataset. Ask yourself the following questions: What kind of patterns might be present in various locations in an image from the MNIST set? What kind of patterns could help you classify these images?

Figure 7: A sound fragment of someone speaking. Besides phonemes, what might be some local structures that could help a network identify what is being said?

Padding

There is one problem with our description in eq. (2) of what the neuron does. We said that for the theoretical definition of convolution to work, we simply pad our sequences with zeros. This however begs the question how long our output sequence should be. Clearly we cannot have an infinitely long output sequence. Far away from the actual (non-zero) input sequence the output is zero, so these outputs can be ignored and obviously these zeros are not computed — see Figure 8 for a visual representation of this. On the interior of the original input sequence, i.e. where the neurons field of view lies entirely within the original input sequence, we are certainly interested in the output, so there the output clearly should be computed — see Figure 9.

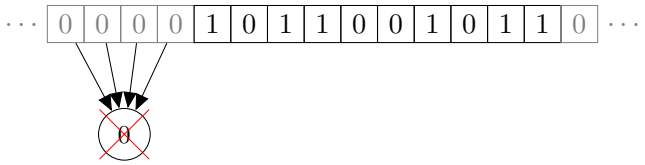


Figure 8: Theoretical zeros in (1) far away from the input are ignored and outputs there are not computed. The padded zeros are drawn in gray.

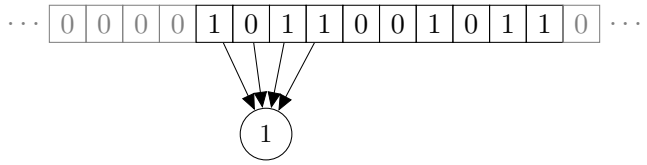


Figure 9: On the interior of the original input sequence outputs clearly need to be calculated. The padded zeros are drawn in gray.

However, the question remains what we should do at the edge of our input sequence — see Figure 10. This depends on the task at hand. In our current example it doesn't make much sense to compute the corresponding outputs since we are looking for locations of the entire pattern. On the other hand, suppose we are training our neuron[2]to recognize bicycles in pictures. In that case we would very well be interested in bicycles that are only partly in the picture, and we would want to pad our picture so that the output of our convolutional neuron has the same size as the original picture.

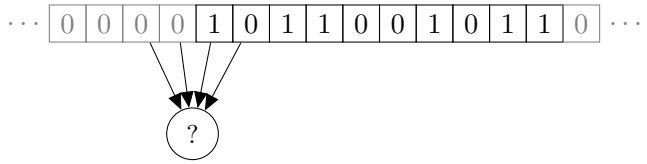


Figure 10: What should happen at the edge of the input sequence?

For one-dimensional convolution, there are three common ways of padding:

Valid padding: No padding happens at all, i.e. all potential outputs for which the field of view does not lie entirely within the input sequence, are discarded.

Same padding: The input sequence is padded in such a way that the output sequence has the same size as the original input sequence.

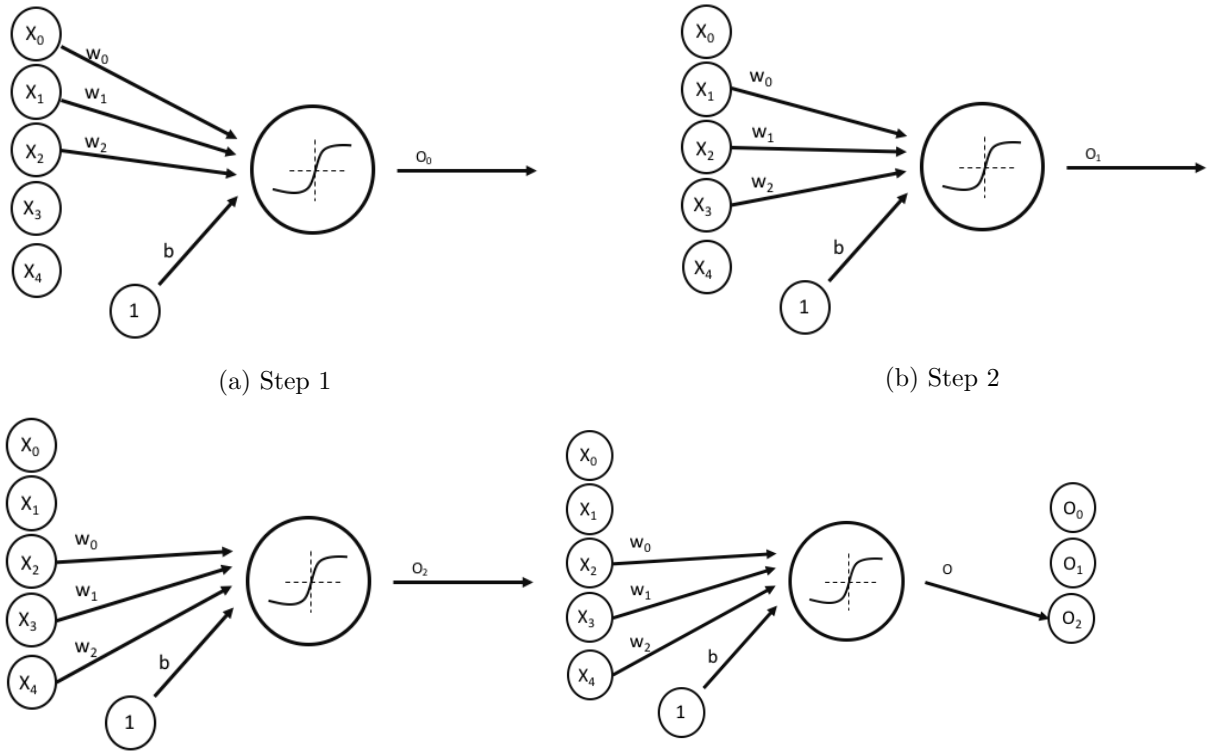
Causal padding: Padding is applied in such a way that the output at index i does not depend on input variables at any later index.

For higher-dimensional convolution, valid padding and same padding are the two common ways of padding.

Backpropagation

To be able to use a new component in a neural network model we need to make sure that it will not prevent us from training our model. To do that the new component needs to allow us to compute the gradient of its output with respect to the inputs. Specifically for the convolutional neuron in addtion to computing the gradient of the output with respect to its input we also need to compute the gradient of the its output with respect to its parameters such that we can update these parameters during training.

[2]Of course a single neuron is a bit too simple for this task, but we just use this to make a clear point.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 3 with all outputs

Figure 11: Forward pass of convolutional neuron

As before we take a forward step with the neuron and a backward step to compute the output values. In this case the difference is that both the forward and the backward steps involve a number of convolutional steps.

In Figure 11 we can observe the forward steps. The neuron processes an input with a a size of 5 $(x_0, ..x_4)$, with a kernel of length 3 (w_0, w_1, w_2). In the first step at the initial position the neuron computes the ouput o_0 Figure 11a. It the second and third steps outputs O_1 Figure 11b and o_2 Figure 11c are computed.

We can represent these outputs as a activation map vector Figure 11d

We assume that the component under investigation is part of a larger model and as such during the backwards pass it receives gradients from the end of the model back to its output (Figure 12).

Let us consider the compute graph of the convolutional neuron at each step of the convolution to compute the backward pass values Figure 13. In this case it is more effective to look at separate compute graphs for each step in the convolution Figure 13a and Figure 13b, and Figure 13c. For each of these steps we can compute a gradient update to a selected parameter (in the figure for w_1) or for an input compute coming from a previous layer (x_1 in Figure 13d and Figure 13e).

The same operations can be expresses in a condensed way using vector representations Figures 14a

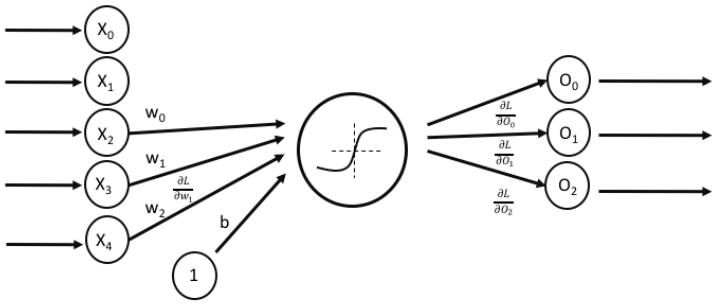


Figure 12: Convolutional neuron receives gradient updates from subsequent layers for each output

and 14b. We can notice in the figures that computing the backward pass can be achieved also by a convolution. In this case we convolve the transposed kernel with the gradients coming from the end of the model.

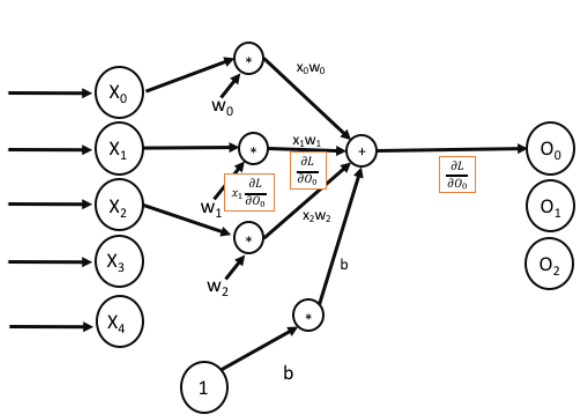
To express these operations more formally, the details of how this works exactly depend a on the type of padding we perform. However, for a theoretical analysis, the type of padding comes down to how much you shift the indices of the output sequence, and at what points you cut the output sequence off. Therefore, if our neuron has kernel

$$\begin{aligned}\mathbf{k} &= (k_0, \dots, k_K) \\ &= (w_K, \dots, w_0),\end{aligned}$$

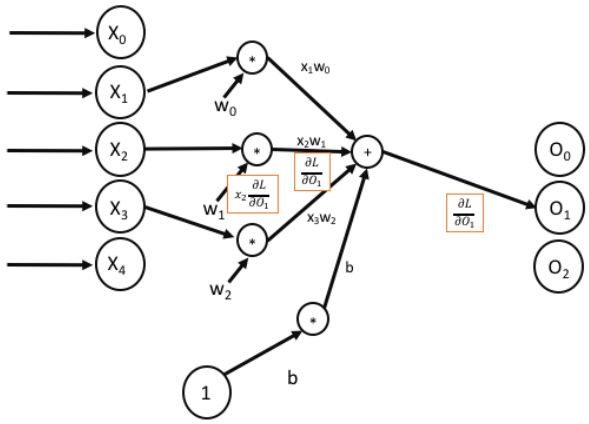
and bias b, and is applied to some input $\mathbf{x} = (x_0, \dots, x_N)$, we can without loss of generality say that the output of our neuron is given by

$$\begin{aligned}o_i &= \phi(y_i + b) && \text{if an output is required, otherwise 0,} \\ y_i &= \sum_{j=-\infty}^{\infty} x_{i-j} k_j \\ &= \sum_{j=-\infty}^{\infty} x_j k_{i-j}.\end{aligned}$$

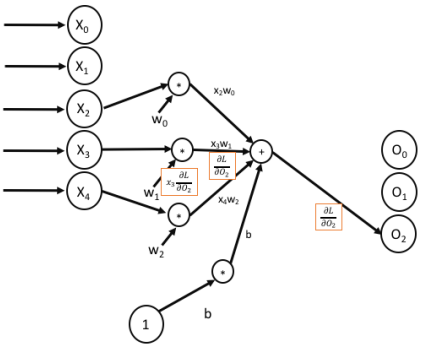
Now if we have a loss function L, which is a function of $\mathbf{o}$, then we can calculate the gradient of L with respect to the parameters of the neuron, and with respect to the input to our neuron as follows:



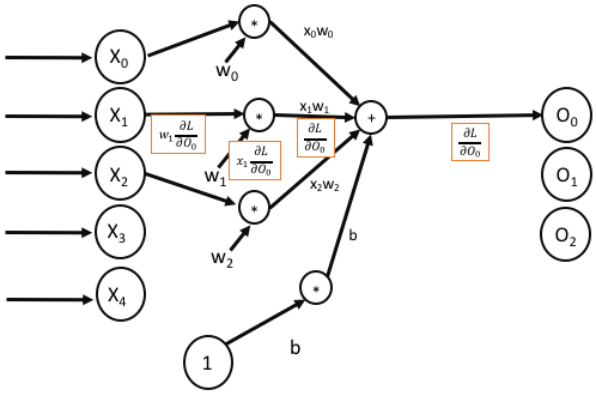
(a) Backpropagating the update value for w_1 at step 1



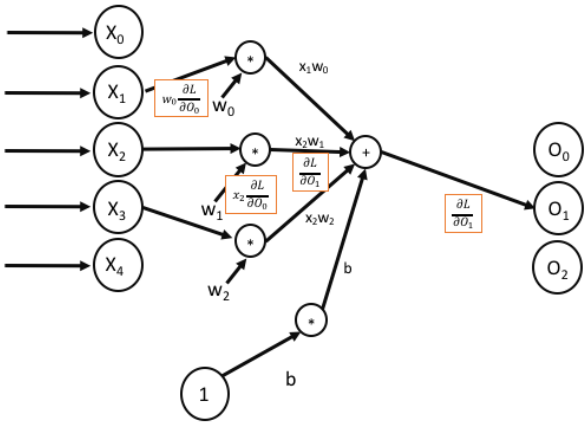
(b) Backpropagating the update value for w_1 at step 2



(c) Backpropagating the update value for w_1 at step 3

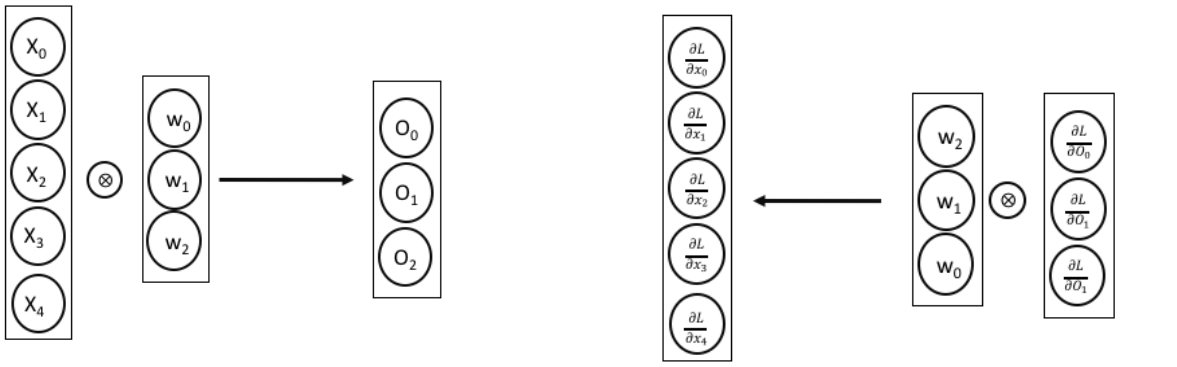


(d) Backpropagating the update value for x_1 at step 1



(e) Backpropagating the update value for x_1 at step 2

Figure 13: Computing the backpropagation of the convolutional neuron for the parameter w_1 and the signal x_1 using its computation graphs



(a) Forward pass

(b) Backward pass (notice the transposed kernel)

Figure 14: Backpropagation convolutional neuron (vector form)

$$
\begin{aligned}
\frac{\partial L}{\partial b} &= \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial b} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}) \\
\frac{\partial L}{\partial \mathbf{y}} &= \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}) \\
\frac{\partial y_i}{\partial k_j} &= x_{i-j} \\
\frac{\partial L}{\partial k_j} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial k_j} \qquad (3) \\
\frac{\partial y_i}{\partial x_l} &= k_{i-l} \\
\frac{\partial L}{\partial x_l} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_l}. \qquad (4)
\end{aligned}
$$

where ϕ' is applied element-wise. Here equations (3) and (4) can be summarized as

$$
\begin{aligned}
\frac{\partial L}{\partial k_j} &= \sum_i \frac{\partial L}{\partial y_i} x_{i-j} \\
\frac{\partial L}{\partial x_l} &= \sum_i \frac{\partial L}{\partial y_i} k_{i-l}
\end{aligned}
$$

which is essentially just convolution with the order of the right sequence ($\mathbf{x}$ or $\mathbf{k}$) reversed[3] (and some shifting of indices).

Question: the indexing in all of this can be a bit confusing. Can you work out the formulas if $\mathbf{x} = (x_0, x_1, x_2)$, $\mathbf{k} = (k_0, k_1) = (w_1, w_0)$, *padding is of the "same" type,* $b = 0$, ϕ *is the identity, and the output is indexed as* $\mathbf{o} = (o_0, o_1, o_2)$*?*

[3]This is called cross-correlation.

Channels

So far we have looked at the case where the input is an array of numbers over which we want to do convolution. As discussed earlier we may have an image with different components that exist in the image in parallel such as different colors is an image. We do not want to do convolution over that dimension as the data is not spatially distributed in that dimension. We refer to the different values in that axis as channels. To be able to use a convolutional neuron in such an image we need the kernel to also have this additional axis such that we can learn unique parameters for each of the different channels.

In practice the tensors that represent the color images are organized according to two main conventions for channels, the most common being "channels last", and the other one being "channels first". This means the input to a convolutional neuron has the following shape:

1-D convolution: $L \times C$ for channels last, or $C \times L$ for channels first;

2-D convolution: $L \times M \times C$ for channels last or $C \times L \times M$ for channels first;

3-D convolution: $L \times M \times N \times C$ for channels last or $C \times L \times M \times N$ for channels first.

If we take the channels last format, we can view all of this as a 1, 2, or 3-dimensional array of vectors, and our kernel should be such an array too. The product in eq. (1) then becomes an inner product, so in the case of 1-dimensional convolution this becomes

$$\begin{aligned}(\mathbf{a} * \mathbf{b})_n &= \sum_{i=-\infty}^{\infty} \mathbf{a}_i^\top \mathbf{b}_{n-i} \\ &= \sum_{i=-\infty}^{\infty} \sum_{j=0}^{C-1} a_{i,j} b_{n-i,j}.\end{aligned}$$

The analysis of backpropagation for a convolutional neuron holds with the necessary changes.

Convolutional layers

In the same way as we benefited from using multiple neurons and forming layers in the MLP model we can create layers of convolutional neurons to create model with high capacity. Such layers of convolutional neurons are refrred to as convolutional layers.

In this context, a convolutional neuron is typically refered to as a "filter". A convolutional layer has multiple filters with the same activation function, the same kernel-size, and the same kind of padding. The outputs of the filters are stacked together so that if a 2-dimensional filter transforms its input to an $N \times M$ tensor, a layer with K filters will transform the input to an $N \times M \times K$ tensor.[4]The output of

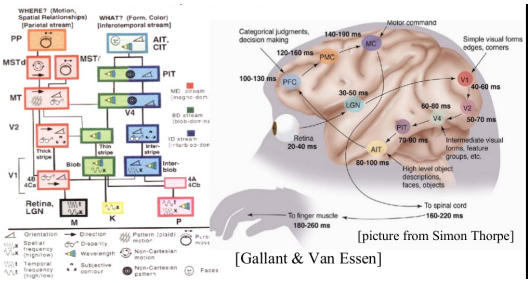


Figure 15: Processing visual information in a sequence of detections of low level concepts to high level concepts is also observed in biological systems as the human visual system.

one convolutional layer can be used as the input for another, allowing us to stack these layers. For a graphical representation of how convolutional neurons are grouped together into layers which can be stacked, see Figure 16.

These models that contain layers of convolutional neurons are referred to as convolutional neural networks (CNN). Typically CNN models have a number of hidden convolutional layers. Each layer detects localized patterns that are combined into more complex features in the subsequent layers. Since we know how to compute the gradient of a convolutional neuron, we can use back propagation and gradient-based optimization algorithms to train these models.

CNN Models

The architecture of the of a CNN as an extension of the MLP model includes a number of convolutional layers typically as hidden layers. These layers detect spatially localized features in the images. The sequence of convolutional layers combines these patterns in increasingly complex patterns, or higher level features. For many tasks these geographically distributed patterns need to be combined and mapped to a target variable. Suppose for example we would like to assign a label to an image. To achieve this we *flatten* the output of our last convolutional layer and feed that to a (sequence of) *dense* or *fully connected* layers. In other words the convolutional layers are followed by an MLP model — see fig. 17a for an example. For the output layer of the CNN the same principles hold as for the MLP. Both convolutional layers and dense layers have an activation function. These activation functions can also be specified and implemented as a separate layer, see fig. 17b for an example of this.

Sub-sampling When a CNN processes a large image, the activation map at each layer is proportionally large. This can place a significant computational complexity burden both on training and inference — especially when each convolutional layer has more filters than the previous. These activation maps give a precise location of where a feature has been detected. In many cases, however, we do not need to know the precise location, but rather only if the feature has been located or not. Therefore, we can benefit from decreasing the size of the activation maps, which will result in improved the efficiency of our models. There are two ways of doing this: the first is using *strided convolutions*[5], the second is using *pooling layers*.

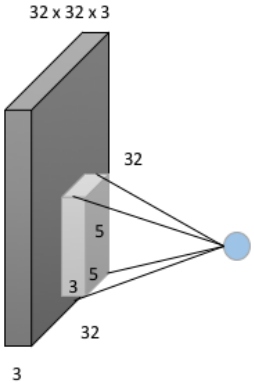
Sub-sampling with Strided convolution

Especially when the field of view of a neuron is large, two neighboring outputs will have largely overlapping inputs. One way of reducing our output size could therefore be to only pick every n^{th} output for some n. This is called strided convolution and n is the size of our strides. For higher dimension, we could also specify different stride sizes for the different dimensions. A graphical representation of strided 1-D convolution is given in Figure 18.

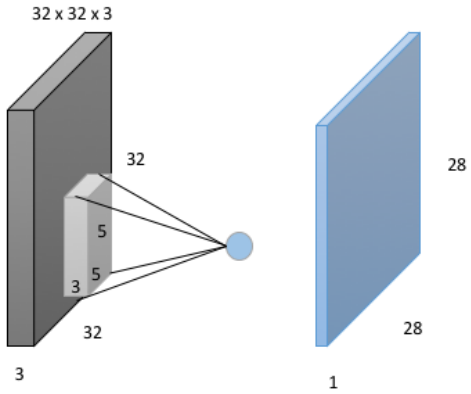
To see what the output size will be for a 2-D convolutional layer with padding and strides we can use the following overview:

[4]The order actually depends on choice of channels convention. If we use the channels first convention we get a $K \times N \times M$ tensor.

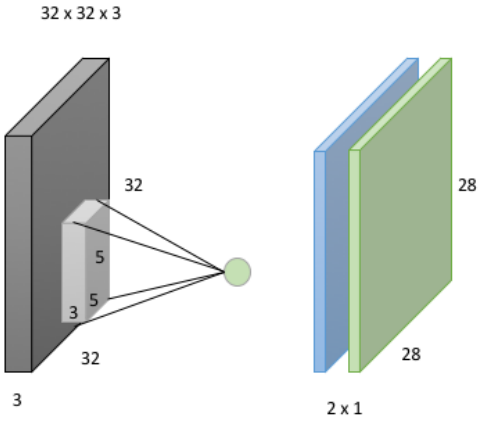
[5]Not to be confused with fractionally strided convolutions, which is a related but different concept also referred to as "transposed convolution".



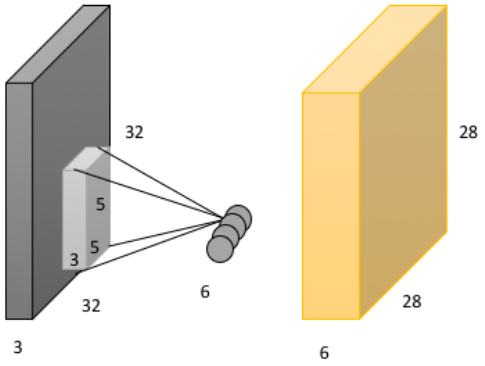
(a) We want to apply a neuron with a $5 \times 5(\times 3)$ kernel to a $32 \times 32 \times 3$ image.



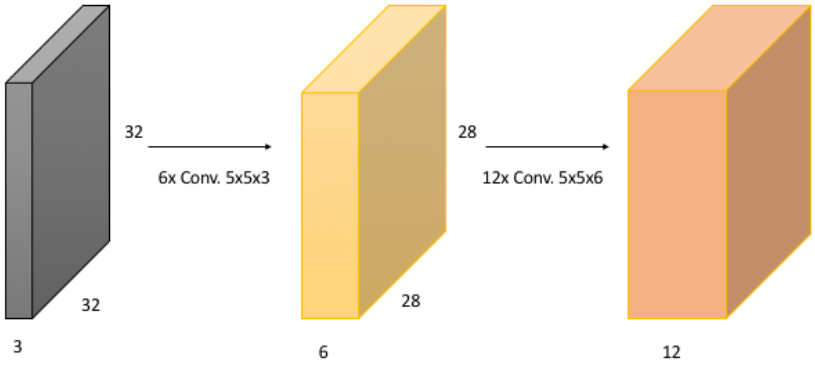
(b) Convolving the neuron over the entire image using "valid" padding, we get a $28 \times 28 \times 1$ output.



(c) We can convolve another neuron with the same dimensions over the image and stack the outputs to get a $28 \times 28 \times 2$ output.

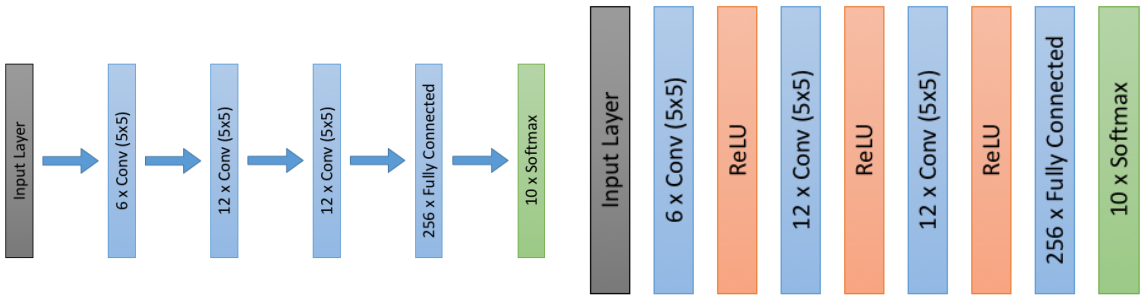


(d) We can stack more neurons this way to get a convolutional layer with 6 filters, giving a $28 \times 28 \times 6$ output.



(e) We can in turn stack such convolutional layers to get a deeper model.

Figure 16: A graphical representation of how we stack convolutional neurons to form convolutional layers.



(a) Depiction of an example CNN architecture

(b) Depiction of an example CNN architecture with activations explicitly mentioned

Figure 17: Two depictions of an example CNN architecture.

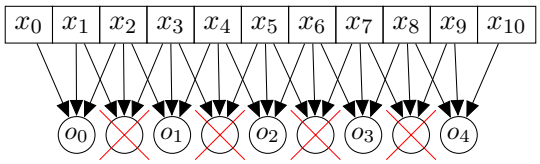


Figure 18: Strided convolution with a stride size of two,and a kernel of size three.

- Accepts:
  - $W_1 \times H_1 \times D_1$
- Outputs:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$
- Where:
  - F is the filter size
  - P is the padding size
  - S is the stride
  - K is the layer depth (number of neurons)

Software packages implementing these convolutional layers can usually infer the output shape of a layer given its input shape, and you can access these details if you need to know what the shapes of your tensors at some point in your network are.

Sub-sampling with Pooling layers

The other way we can reduce the output size is by using a *pooling* layer. Pooling layers 'pool' the values of a local region in an image into a single value, such that the dimensionality of their output is reduced. There are different pooling layers such as 'Average Pooling', 'Maximum Pooling'

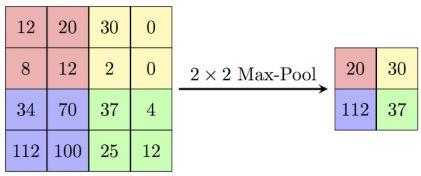


Figure 19: Subsampling - maxpooling

and 'Minimum Pooling'. The 'Maximum Pooling' or maxpooling layer is commonly used in image processing. This layer takes a specified windows size (e.g. 2 by 2) and produces as output for that location the maximum of the values in that window — see Figure 19.

Adding a pooling layer to the model requires that we can also backpropagate using that layer. Specifically we need to be able to compute the gradient of the output with respect to the inputs. For this we have the following formulas:

$$a(x) = \max_{i \in \{0,\ldots,m-1\}} x_i \qquad \text{has derivatives}^6$$

$$\frac{\partial a(x)}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise,} \end{cases}$$

and

$$a(x) = \frac{1}{m} \sum_{i=0}^{m-1} x_i \qquad \text{has derivatives}$$

$$\frac{\partial a(x)}{\partial x_i} = \frac{1}{m}.$$

Question: The derivative for an average pooling layer is smaller than one. Do you expect using average pooling in combination with convolutional layers to cause problems with vanishing gradients? Why or why not?

In Figure 20 we can see an example architecture of a CNN model with maxpool layers. In this example a maxpool layer is used after every convolution. For deeper and larger models it is also common to apply maxpool after every second or third convolutional layer instead.

Note that the components we have discussed so far are already sufficient to build a broadly applicable class of CNN models consisting of a bunch of convolutional layers with pooling layers in between, and a fully connected network on top of that.

4.3 Image Analysis with CNN

Digit classification with MNIST

Now that we have the basic building blocks of Convolutional Neural Networks, let us look at how we can use them to do image analysis. We first consider the classification task on the MNIST dataset.

[6] This is in the case of a unique maximum. If the maximum is attained at two or more distinct indices we only have one-sided derivatives for those indices.

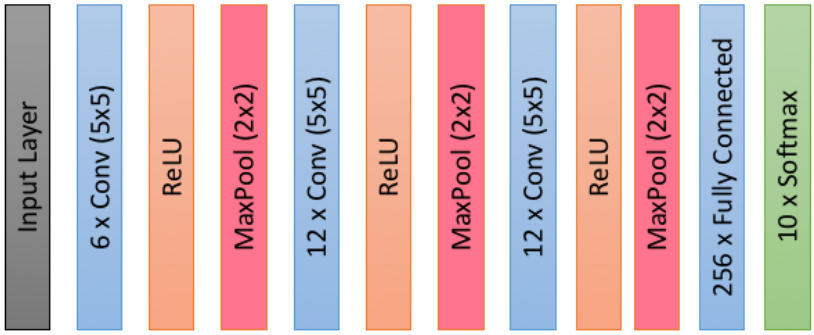


Figure 20: CNN - architecture with maxpool layers

Then we look at what kind of other image analysis tasks we might encounter, and finally we look at some existing image analysis models and how we can use those to solve specific problems.

In **??** we talked about how the sharing of weights allows the network to recognize learn a pattern in one place and recognize it everywhere.

In order to do classification we again make the labels into a one-hot encoding

$$y \mapsto e_y = (\delta_{y,j})_{j=0,\ldots,9}.$$

We use a Softmax layer to interpret the output of our network as a probability distribution:

$$\text{softmax} : (v_0, \ldots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\ldots,9}.$$

and for our loss we use

$$\begin{aligned} L(y, \mathbf{p}) &= -\sum_{i=0}^{9} e_{y,i} \log(p_i) \\ &= -\log(p_y). \end{aligned}$$

we can view as either the negative log-likelihood of the correct label according to the probability distribution, $\mathbf{p}$, that our network gives as its output, or as the cross-entropy of the probability distribution that our network gives as its output relative to the correct (empirical) distribution represented by the one-hot encoding of the label.

Let us consider the architecture shown in Figure 21.

As an input we take $28 \times 28 \times 1$ tensors and we pass them to a two dimensional convolutional layer with 6 filters and a 5×5 kernel and valid padding. The output of this first convolutional layer then is a $24 \times 24 \times 6$ tensor. As activation we apply the ReLU function to every entry of that tensor. Next we send this tensor to the second convolutional layer with 12 filters, a 5×5 kernel, valid padding, and again ReLU activation. This gives us $20 \times 20 \times 12$ tensor on which we perform maxpooling with a 2×2 window to obtain a $10 \times 10 \times 12$ tensor. Next we send this tensor to the third and final convolutional layer which again has 12 filters, a 5×5 kernel, valid padding, and a ReLU activation which gives s a $6 \times 6 \times 12$ tensor to which we again apply maxpooling with a 2×2 pool to get a $3 \times 3 \times 12$ tensor. We flatten this into a 108-dimensional vector which we send to a fully connected layer with 256 neurons and ReLU activation. To obtain an output we send this 256-dimensional vector to a fully connected layer with 10 units whose output is fed to a softmax function in order to obtain a probability distribution.

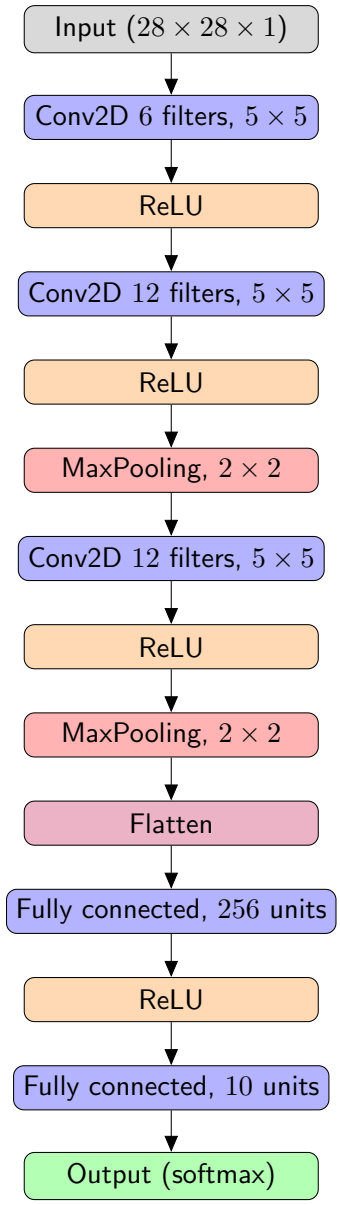


Figure 21: The architecture performing classification on the MNIST dataset.

Note: Convolutional layers give us a natural way to develop model that are invariant translation of the local patterns. You can imagine that our model can benefit from invariances to other transformations as well. We can also achieve this by modifying the data we train the model on. Before feeding the data to the network in our training loop we can randomly shift, rotate, flip, shear deform, and zoom in on the images. This essentially increases the variations that our training data covers from the input space. This approach is referred to as *data augmentation* and is particularly valuable in regimes with limited amount of data available. For this we do need to understand well what kind of transformations should our model be invariant to. For example we can not use a horizontal flip transformation on a task where we would like to distinguish the left from the right hand in an x-ray image, unless we also change the label accordingly.

State-of-the-art CNN classifiers

Beyond the foundational aspects of CNN models discussed so far, state-of-the-art models include many further developments. We discuss this evolution by highlighting a number of impactful solutions.
Historically the CNN model was introduced with the LeNet architecture[7]. In Figure 22, you can

[7]Though earlier versions were already published as early as 1989 (LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W.; Jackel, L. D. (1989). "Backpropagation Applied to Handwritten Zip Code Recognition". Neural Computation. MIT Press - Journals. 1 (4): 541–551.) with the core concepts published even earlier (Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position")

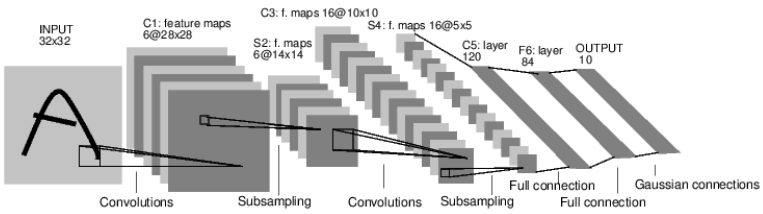


Figure 22: LeNet model. LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

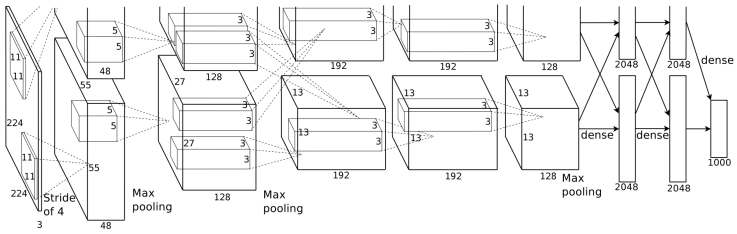


Figure 23: The AlexNet: Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

see on depiction of LeNet where rather than layers and neurons you can see the activation maps and the kernels. You can work out the architecture based on this activation maps. For example you can see that the first convolutional layer has 6 neurons based on the dimensionality of the first activation map.

A more recent model and arguably one of the models that is most responsible for the increased attention to this field is AlexNet. AlexNet won the ImageNet Large Scale Visual Recognition Challenge in 2012 with a very large margin and set the stage for further developments in deep learning. The ImageNet dataset consists of color images with resolution of 224 by 224 labeled with 1000 different classes. The model is depicted in fig. 23 with the activation maps.

Following the success of AlexNet a number of models the rate of development of new and improved models rapidly increased, particularly on the ImageNet task. One of these models is VGGNet Creffig:vgg-network. VGGNet introduced models with much larger complexity. The model also reduced the number of decisions that you would need to take by standardizing the size of the kernels to 3 by 3. This model (or variations) of it is still very useful today. For many tasks that do not present complexity as large as ImageNet a variation of VGGNet (typically with fewer blocks) can deliver satisfactory performance. Due to its simplicity in implementation it can be a good choice as a first attempt at a suitable task.

One of the main drawbacks of VGGNet was the large number of parameters. Relative to its depth VGGNet uses many parameters, particularly after the last convolutional layer, the dense layer that follows uses a large about of parameters to process that last activation map.

The GoogleNet model (Figure 25) that relies on the inception block addressed this challenge. This architecture is deeper and uses less parameters that VGGNet. It also delivers superior performance.

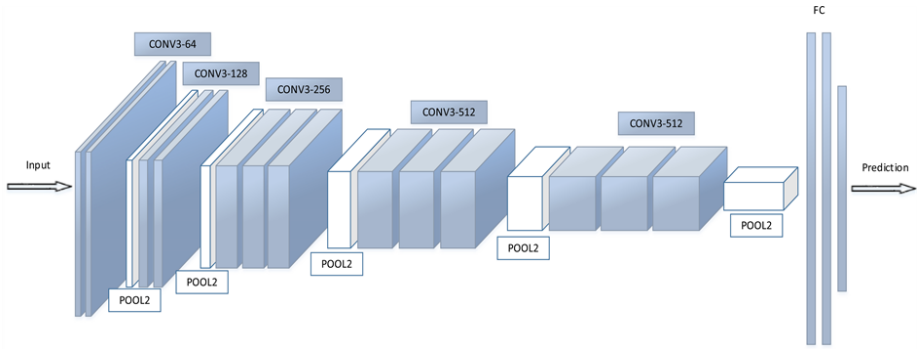


Figure 24: VGGNet: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

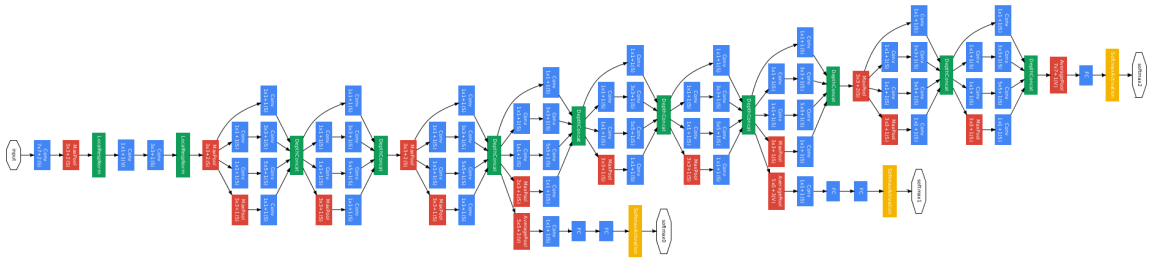


Figure 25: GoogleNet: Szegedy, Christian, et al. "Going deeper with convolutions." Cvpr, 2015.

The architecture consists of a number of inception blocks that are combined together. The model also uses multiple output layers to improve the flow of the gradient updates deeper in the model and in turn improve the training time.

The inception block, deals with the choice of the size of the kernel by actually using multiple kernel sizes in parallel. Specifically, the inception model uses 1×1 or 3×3 or 5×5 kernels Figure 26.

Note: What is the point of using a 1×1 kernel? Such a filter certainly can not learn spatial patterns. However, with such a kernel we can combine all the activation on a particular location, or specifically we can learn how to combine them. One other way to think of the 1×1 kernel is that it offers a way to change the depth of the activation map. This is very closely related to the computional complexity of the model. A 1×1 kernel can enable us to change the depth of activation maps. This trade off between capacity and complexity enables us to optimize for different settings.

Even though a number of advances were introduced in the models so far (ReLU activations, multiple output layers), the vanishing gradient effects still limits the performance of very deep models. The ResNet model has brought a new innovation that enabled much deeper models. The ResNet architecture introduces the residual block (Figure 28). The residual block introduces the skip connection, an identity map between the input and the output. The map allows for gradients to flow without passing through the non-linear layers as they are positioned in parallel to the skip connection.

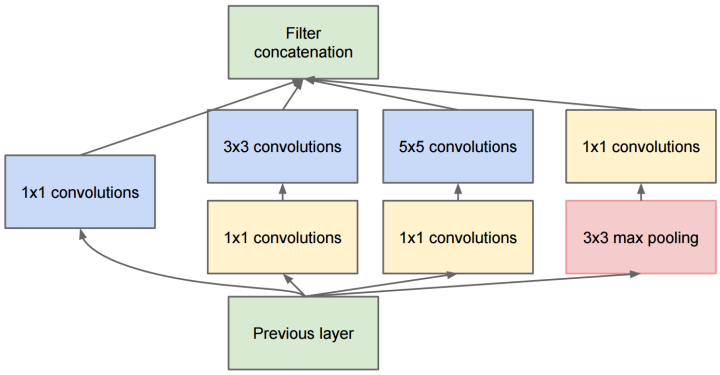
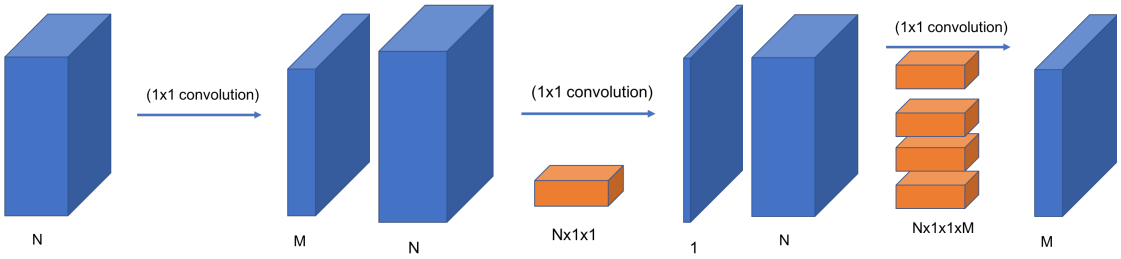


Figure 26: Inception module



(a) $1x1$ convolution (b) $1x1$ convolution (c) $1x1$ convolution

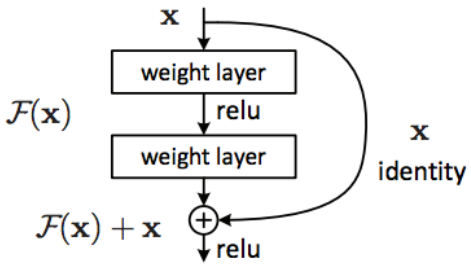


Figure 28: Residual block

8

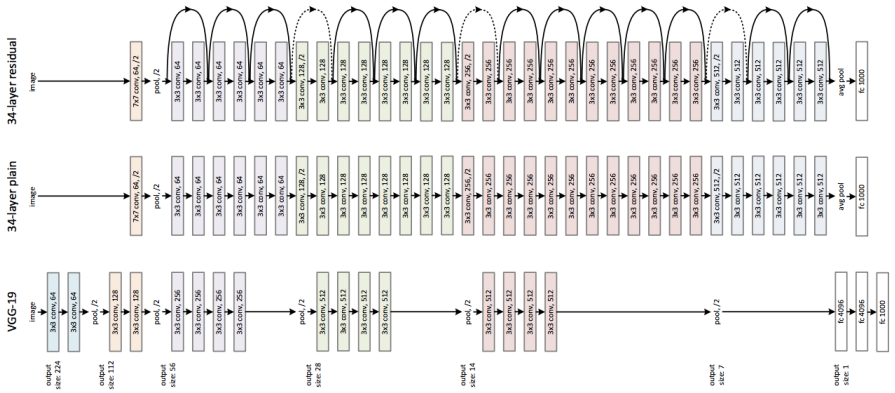


Figure 29: Residual Network
9

Using the residual block the ResNet model can achieve improved performance by using a deep architecture. In Figure 29 a 34-layer deep architecture is depicted in comparison to a 19-layer deep architecture of VGGNet.

Some of these advancements have later on been combined, such the Inception blocks with skip connections that further improved the performance of the models. These particular architectures are currently no longer state-of-art and have been surpassed by even newer architectures, but have certainly been influential and very commonly used on many tasks. We leave even more recent developments out of the scope of this paper.

Besides classifying images there are other tasks we can solve with CNN models in the domain of image analysis. Each such task comes with decisions regarding the input and output encoding, the model output, the loss function(s), model architecture and training configuration. We continue with a overview that is by no means complete, but should give you an idea of what kind of tasks you may encounter and what kind of choices you could make.

Image localization

The task of image localization, extends the task of object detection into object detection and localization in the image. Typically, we formulate the task of object detection as a machine learning classification task as given in the MNIST example above. For image localization we have the following formulation; detecting the type of object is formulated as classification and identifying the location of that object is well suited for the regression formulation. As such, the model has multiple outputs that have different roles and accordingly the loss function has multiple terms corresponding to each output and its role.

On a conceptual level we define the notion of a *bounding box*. A bounding box can be represented by four numbers: an x-coordinate, a y-coordinate, its width w, and its height h. The content of the bounding box can again be encoded as a one-hot vector, and the prediction is again a distribution over the possible classes. We then use a bounding box to localize an object in the image (Figure 30). As such this task requires that we have corresponding annotations for each image. For each object in the image we need an annotation of its bounding box. Our model needs to be able to output a

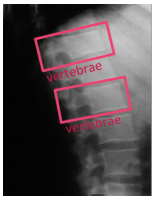
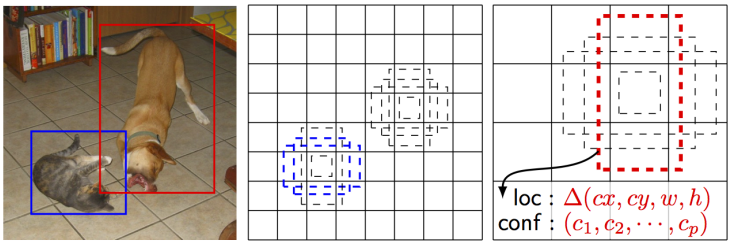


Figure 30: Localization of objects



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 31: YOLO - you only look once

desired number of bounding boxes.

We can then use the following loss functions for each bounding box:

- MSE for the location and size of the boxes;
- Cross-entropy for the content of the boxes.

In practice we would have a CNN model with four outputs for each bounding box that do regression and one output that does classification.

Question: How can we use these losses to deal with a shortage or surplus of bounding boxes?

State-of-the-art CNN localization

The "You only look once" (YOLO) architecture[10] develops a solution for image localization by splitting the image in cells and assign each cell an number of bounding boxes. Each bounding box has (x, y, w, h, l) and distribution over the type of objects present (**??**).

The details of the architecture are given in Figure 32.

Image segmentation

Image segmentation means partitioning the pixels of an image into segments. This can be used for object detection, but is also useful for other recognition tasks. It comes down to classifying each pixel, so our output is a width $\times$ height $\times$ c tensor of values between 0 and 1 where c is the number of classes. In order to get the right output format (a probability distribution per pixel) we can

[10]Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

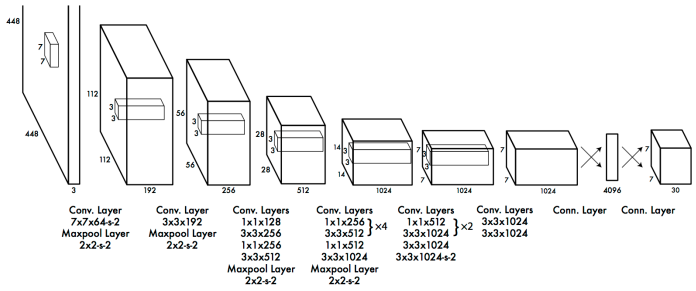


Figure 32: YOLO - you only look once

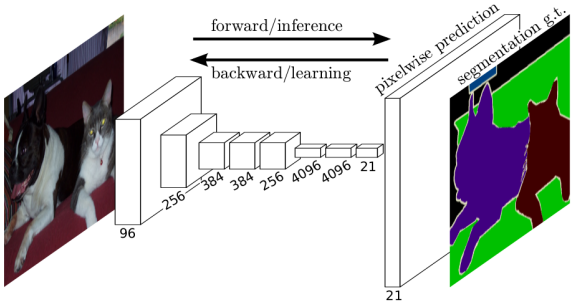


Figure 33: Image Segmentation

apply the softmax function over the last axis of this tensor. A good loss function for training a model for image segmentation is pixel-wise categorical cross-entropy (Figure 33.

State-of-the-art CNN for segmentation

One highly successful architecture of image segmentation is UNet (Figure 35). The difficulty in image segmentation is that information about the regions that need to be segmented in the image are present both globally in the image and locally in small patches. The global context brings information about the type of object with respect to other objects in the image and the local information gives the ability to precisely separate the borders of the object. Looking at a medical imaging example in Figure 34. The different organs are segmented in the image. The type of the object is needs to be determined from the global human anatomy present is such images, while for the border of each image we need precise pixel information such that we can delineate the edges.

This challenge is addressed by the UNet architecture. The model processes the input image in stages by iterativelly bringing the information into a more compact representation. This series of compatifications form a global context at different levels. This forms the bottom U-shaped channel of the model. To carry the local information needed for detecting the edges of the object the model introduces horizontal 'skip' connections where context and different levels is being mixed the high level information.

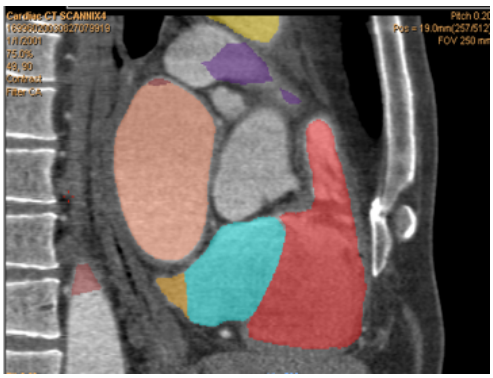


Figure 34: Image Segmentation

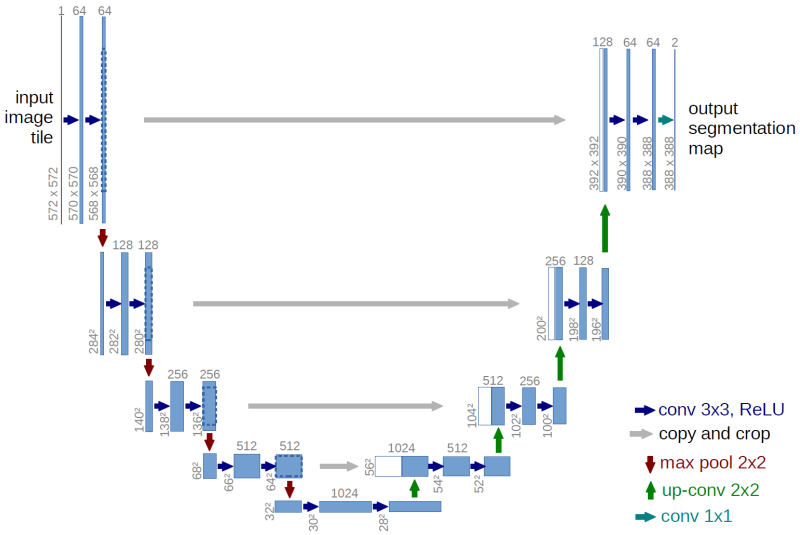


Figure 35: UNet CNN for image segmentation: Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.

Figure 36: Input image → output (transformed) image (Gatys, Leon A., et al. "A neural algorithm of artistic style. arXiv 2015." arXiv preprint arXiv:1508.06576 (2015).)

Filtering

There are multiple approaches to removing noise from images with neural networks. One simple approach is to have a network that gives an output the same size as the input, and for which the loss during training is the mean squared error between the predicted image and the original noiseless image. A particularly efficient architecture for such a task is the fully convolutional model introduced before. The concept of introducing a noise to the input data and training a model to reconstruct the orignial image (without noise) is also related to the denoising autoencoders, which we discuss in more detail later in the chapter. Filtering, nevertheless, has also a broader set of applications. We do not go in more details about such applications here, but we introduce one architecture that should illustrate the broad range of possibilities.

In the work Gatys et al. a model is presented that can copy the artistic style of a given image an apply it to another image (Figure 36). This method is a good illustration of how a CNN model can be used in a task where we need to generate an image.

Transfer learning

Some of the models that we have discussed so far are very large and have been trained extensively in order to get very high levels of accuracy. When we want to do image analysis in practice training these models from start is often not feasible for several reasons:

- We have much less data than what is typically used to get these state of the art results.
- We don't have the expensive hardware or the large amount of time required to train these models from start.
- The task we want to perform is slightly different, e.g. we want to do classification with a different number of classes.

Fortunately, it is possible to re-use (parts of) models trained on one dataset and one task on a different task or dataset. This is called transfer learning.

For CNNs, this shouldn't come as a surprise if we think about our motivation for using convolutional layers: the aim was for filters to pick up on increasingly high level local patterns that are relevant in a broader context. Thus the features learned by the convolutional layers in a CNN trained on image classification should be useful for doing image segmentation as well.

In Section 4.1 we talked about how many CNN architectures consist roughly of two parts: a convolutional part and an MLP stacked on top of that[11]. A basic way of doing transfer learning with such models is the following. If we have trained a full CNN with such a two-part architecture on a dataset D_1 with task T_1, and we want to use what it has learned to perform another task T_2 on a dataset D_2, we can simply take the (trained) convolutional part of the model, and build a new fully connected network on top of that. The new network can than be trained for a relatively short period of time on D_2 to perform T_2. This way the representations the first CNN has learned on the first task are transferred to the new model and can be used for the second task. Alternatively, in many cases it is possible to even keep most of the original MLP and only change the final (output) layer.

When training the new model we can either freeze (parts of) the transferred layers or train the full model, so called *fine tuning*. Fine tuning can help get better results, but when there is little data available in D_2 it can also increase the risk of overfitting. Moreover, when the model is very large you have insufficient computational resources, freezing the transferred part of the model can make training it much cheaper because you don't have to do back-propagation over those layers. *Question: why is this? And under what circumstances does this not hold?*

4.4 Representation learning of images

In chapter 1 we talked about how we want to automatically learn meaningful features from complex data so that we can use those features for various (machine learning) tasks. Then in Section 4.3 we discussed how we can use layers of a CNN trained on one task to build models that perform some other task. This strongly suggests that a CNN indeed learns meaningful, or at least useful, features. In this section we will explore this property of CNNs further.

Autoencoding

So far the tasks that we considered fall into the category of supervised learning. In a broader context we can imagine a scenario where we do not have annotations for part or all of the datapoints. Even for such a scenario there are task for which need more efficient representation of the data. For example an image retrieval task would benefit from an efficient representation of the data where computing the distance between the datapoints is more meaningful (but also more efficient) then computing the distance in the image space. Furthermore, in light of the transfer learning discussion from above, we may have a scenario where we would like to develop features in unsupervised setting that we can transfer in a supervised setting and therefore improve the efficiency of the supervised training specifically with respect to the amount of annotations.

Such reprsentations can be learning with the autoencoder architectures. The autoencoder learns a compressed representation of the data by learning how to reconstruct the data after passing it though some kind of inofmration bottleneck. Specifically, an autoencoder consists of two parts: an encoder f and a decoder g (fig. 37) . These two parts are trained such that the decoder is an approximate inverse of the encoder on the target data, i.e.

$$g(f(x)) = x$$

[11] Of course this second part doesn't necessarily have to be an MLP, or in general even a fully connected network. The point is mainly that the second part is either non-local, or task specific.

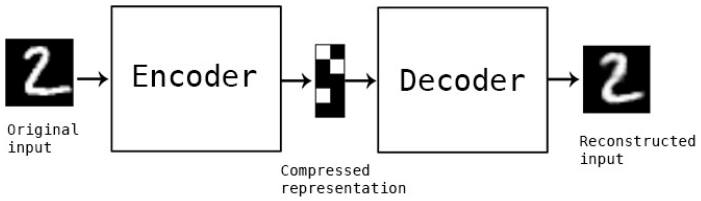


Figure 37: A conceptual representation of an autoencoder (https://blog.keras.io/building-autoencoders-in-keras.html)

for all x in the data set. The encoded data,

$$h = f(x),$$

is then the representation learned by the autoencoder. However, we don't want $g \circ f$ to be the identity map on all possible input data, otherwise it would not have to learn a useful representation. Therefore we need to introduce some kind of restriction to the autoencoder.

An autoencoder architecture was introduced in fig. 38. A convolutional version of an autoencoder also inclused convolutional layers fig. 39.

To train the autoencoder model we need to use a loss function that computes the precision of the reconstruction compared to the input. As a basic loss function we can use $L(x, g(f(x)) = \|x - g(f(x))\|^2$, or scale this by the number of pixels to get the mean squared pixel-wise error.

Using a different loss function allows us to bias the model towards a particular representation. We can represent our domain knowledge about what in the dataset is signal and what is noise via the loss function. For example, we may only care about the intensity of the pixel values, but not for their color. We can then define a loss function that first converts both images into black and white encoding and then computes the mean square error of the black and white values.

There are different approaches to introduce a bottleneck in the autoencoder models. The most direct approach is to reduce the number of neurons in one or more layers in the autoencoder. In this approach the narrowest layer defines the bottleneck of the capacity of the model. The capacity of the learned representation is determined by this layer, which is typically the representation layer. This models reduces the dimensionality of the input to the dimensionality of the representation layer. Note: In the chapter on generative models we discuss at another technique for regularizing autoencoders called the "variational autoencoder". There we use the encoder and decoder to parameterize conditional probability distributions on the spaces of data, and on the space of representations, the so called "latent space". Even though, such a model also fits the definition of an autoencoder, it is more suitable to study this model in the context of generative models.

Sparse Autoencoder We can induce particular properties of the representation of the autoencoder model by introducing specific a loss function on the latent space. This gives us regularized autoencoders with a loss function of the form

$$\text{Loss}(x, g, f) = L(x, g(f(x)) + \Omega(f(x)) = \|x - g(f(x))\|^2 + \Omega(f(x))$$

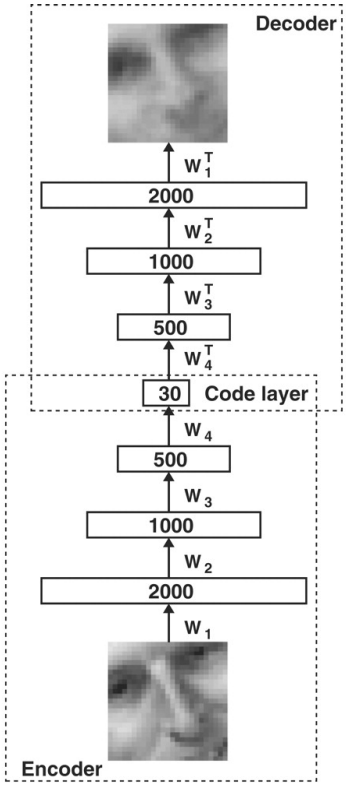


Figure 38: MLP Autoencoder: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

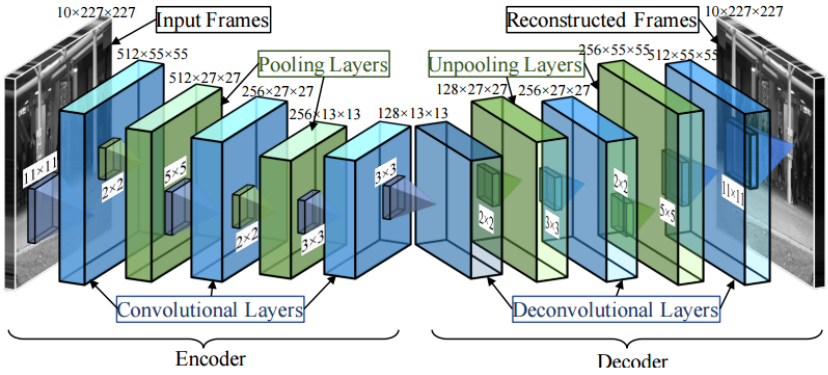


Figure 39: Convolutional Autoencoder

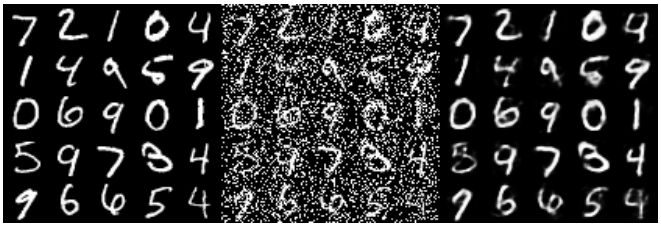


Figure 40: De-noising Autoencoder - Bengio, Yoshua, et al. "Generalized denoising auto-encoders as generative models." Advances in Neural Information Processing Systems. 2013.

For example, we may aim to develop a sparse representation[12] of our data. Such a sparse autoencoder can be achieved by introducing L^1 regularization on the output of the representation layer:

$$\Omega(h) = \sum_i |h_i|.$$

De-noising Autoencoder Another way to avoid learning the identify function is to introduce noise to the input — see Figure 40. The noise acts as a bottleneck as the capacity of the model is now determined by the signal-to-noise ratio. In other words, the variance of the noise determines how close two values in the signal can be discerned from each other. If we introduce through some (stochastic) function $n(x)$, the loss we try to minimize during training is given by

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

Moreover, the denoising autoencoder can also be interpreted as a generative model, as the noise forces the f and g to implicitly learn $p_{data}(x)$. We will look at generative models in more detail in the last chapter of this course.

One-shot/metric learning

Metric learning techniques develop a distance metric between the data points in the dataset. There are a number of tasks that can benefit from a distance metric that captures the semantic properties of the data. We already mentioned the retrieval task in the context of the autoencoder. However, the autoencoder as an unsupervised model can only offer a compressed representation of the data. Metric learning techniques allow us to learn various distance metrics based on available supervised information about the data. A model that produces such a metric enables solving various targeted retrieval or recommendation tasks. Moreover, these techniques closely correspond to the one/few-shot classification task. The typical classification task includes a fixed number of classes and a sufficient amount of examples per class. In contrast to this we can have a task that has a large number of classes and only a few (or even one) example per class. In this section we study in details the methods for this types of tasks.

One-shot learning In the supervised learning setting, the model eventually learns the most salient features that predict the values for the target variables. When we only have very few (even one) example datapoint per class, there is little opportunity for the right patterns to be distinguished from the rest of the information in the data. Take for example the task of facial recognition. For

[12]Sparse representations are representations where the vast majority of entries are zero.

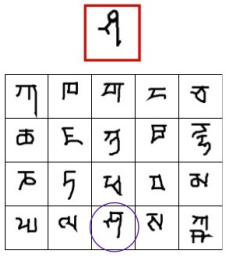


Figure 41: One-shot learning

this task we may have a large number of training points, but we also have a large number of classes, as we may only have one photo per person. Furthermore, this task will likely require a solution that can adapt to adding new people without the high computational cost of re-training the whole model.

So, now rather than developing a model that can detect specific classes, we are better off training a model that can learn the characteristic features for faces and then introduce the downstream task of detection. One technique that allows developing more general representations of data is metric learning. Metric learning methods develop a model that maps the data to a metric space where the distance captures a given semantic property. Here we will be looking at deep metric learning, where the mapping is given by some deep neural network.

In our face detection example, the goal of the model would be to represent the data in a space where images of faces belonging to the same person would be closer together than images of different people.

In this setup, the model has a better chance to develop features that distinguish faces of people and use them to generalize even when given a single example of the face of a person. Furthermore, adding a new person to the dataset may require little or even no additional training.

Metric learning The goal is to learn a distance metric d and some value τ such that for images of the same class we have

$$d(I^{(1)}_{c=1}, I^{(2)}_{c=1}) < \tau$$

and for images of different classes we have

$$d(I^{(1)}_{c=1}, I^{(2)}_{c\neq 1}) > \tau.$$

One method to develop such a metric is based on the Siamese network model — see Figure 42. Here we learn a distance metric d of the form

$$d(x_1, x_2) = \|f(x_1) - f(x_2)\|^2 \tag{5}$$

for some function f parameterized by a deep neural network. The goal is thus to train the network f such that

- if x_i and x_j are from the same class $\|f(x_i) - f(x_j)\|^2$ is small;

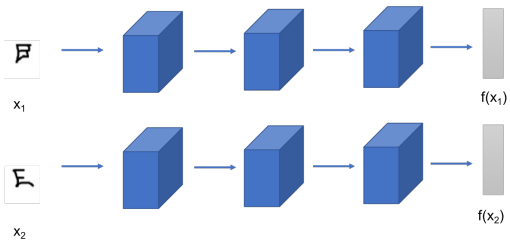


Figure 42: Siamese Network: Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

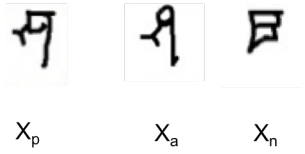


Figure 43: Triplet Network[15]

- if x_i and x_j are not from the same class $\|f(x_i) - f(x_j)\|^2$ is large.

[13] To achieve a better robustness to the margin parameter, τ, the triplet network method introduces a more robust training setup including three datapoints and a three component loss function.

Triplet Network The idea behind the triplet network — see Figure 43 — is to show three images to the same network: two images from the same class x_p and x_a, and an image from a different class x_n. We call x_a the anchor (or baseline) input, and x_p and x_n the positive and negative input respectively. The goal is then to learn a metric such that the distance between the positive input and the anchor is less then that between the negative input and the anchor. I.e.

$$d(x_p, x_a) \leq d(x_n, x_a)$$

or in terms of a learned metric (5)

$$\|f(x_p) - f(x_a)\|^2 \leq \|f(x_n) - f(x_a)\|^2 .$$

We can rewrite this as

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 \leq 0.$$

Now in order to build in more robustness, we want this inequality to hold by a margin $\alpha > 0$, i.e.

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0.$$

In order to achieve this we use the *triplet loss* given by:

$$L(x_p, x_n, x_a) = \max(\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha, 0).$$

[13] Hoffer, Elad, and Nir Ailon. "Deep metric learning using triplet network." International Workshop on Similarity-Based Pattern Recognition. Springer, Cham, 2015.

[14]Selecting the triplets is important and has a significant impact on the efficiency of the metric learning. This is an active research area and out of the scope of this document.

Triplet Selection

The number of all possible triplets is k combination of n, where k is 3, and n is the number of datapoints in the dataset. Training with all of the triplets is typically not feasible. Nevertheless, to achieve a good embedding (i.e. learning a discriminative distance metric), we need only a small fraction of all possible triplets. However, the quality of the embedding depends significantly on the selected triplets as not all triplets as equally informative.

Based on the definition of the loss, there are three categories of triplets.[15]

easy triplets: triplets which have a loss of 0, because:

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0.$$

hard triplets: triplets where the negative is closer to the anchor than the positive, i.e.

$$\|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2$$

semi-hard triplets: triplets where the negative is not closer to the anchor than the positive, but which still have positive loss:

$$\|f(x_p) - f(x_a)\|^2 < \|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2 + \alpha$$

Each of these definitions depends on where the negative is, relatively to the anchor and positive. We can, therefore, extend these three categories to the negatives: hard negatives, semi-hard negatives or easy negatives. The figure below shows the three corresponding regions of the embedding space for the negative as depicted in Figure 44.

Easy negatives are less informative and will contribute no gradients to our training. Hard negatives contribute most to training but sometimes this selection method might result that too many triplets are consisting of mislabeled and poorly imaged samples, specifically it might lead to a collapsed model (i.e. $f(x) = 0$). Semi-hard negatives selection, as a trade-off option, select relatively more informative triples as well as avoid model collapsing.

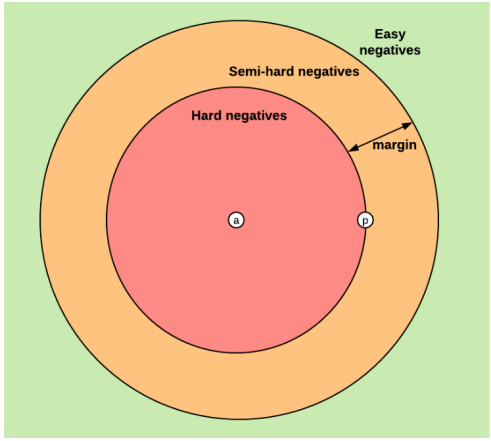


Figure 44: The three types of negatives, given an anchor and a positive

[14]Hermans, Alexander, Lucas Beyer, and Bastian Leibe. "In defense of the triplet loss for person re-identification." arXiv preprint arXiv:1703.07737 (2017).

[15]Schroff, Florian, Dmitry Kalenichenko, and James Philbin. "Facenet: A unified embedding for face recognition and clustering." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

Models for sequential data (RNN)

Deep Learning (2IMM10)
Spring 2020

5 Models for sequential data (RNN)

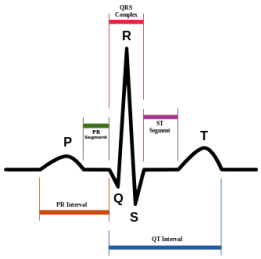
The learning outcomes of this chapter:

- Able to develop models for sequential data using recurrent neural networks

6 Temporal correlations

In this chapter we study models for tasks where the data is typically sequential with long distance correlations. Let us examine the following example:

We have data collected over a period of time from senor measurements of the electrical activity of the human hearth. These measurements form an Electro Cardiogram, or ECG, depicted in Figure 1.



(a) The QRS complex observed in an ECG.

(b) Multiple QRS complices.

Figure 1: Electro cardiogram (ECG)

The QRS complex observed in the ECG, consist of the Q wave, the R wave and the S wave — see Figure 1a. Without going into any medical details as this is not the purpose of the example, we only define our task as detecting the length of the complex as this is relevant diagnostic information.

To be able to estimate the length of the QRS, basically we need to detect the individual components and measure the time from the beginning to the end of the complex.

As the Q-wave, R-wave and S-wave have a particular shape, one can safely assume that a CNN model can develop detectors for those shapes — see Figure 2 for this.

When dealing with sequential data, we often encounter very long signals, particularly during execution of the model. In general we can often assume that our model does not need to see the whole signal to produce the output. In this particular case, we use a sliding window approach, where our

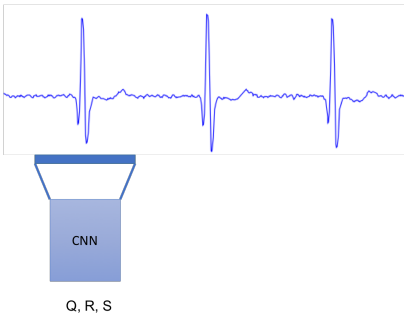
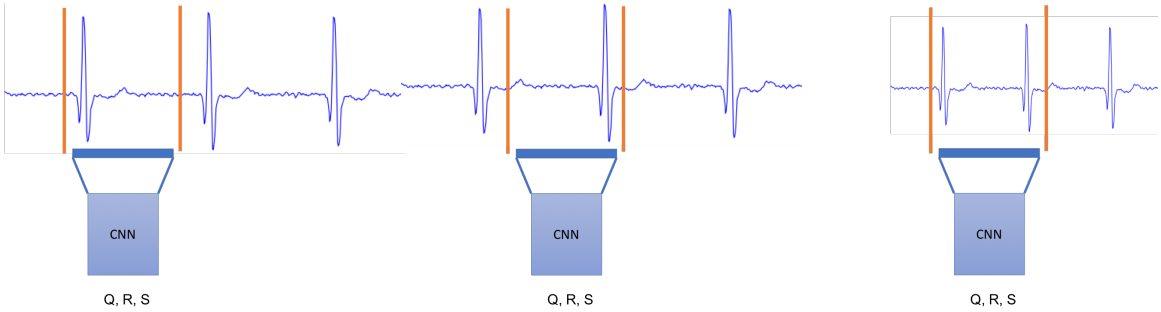


Figure 2: CNN model for detecting the QRS complex



(a) The QRS complex located at the beginning of the window

(b) The QRS Object at the end of the window

(c) Two QRS objects within the window

Figure 3: The QRS complex can be present in various spots of the sliding window, or even at multiple points in the same window.

model looks at a window of data that is slid over the data — see Figure 3a. Detecting the complex with such a CNN model involves a number of decisions, one of which is the size of the window.

The task for our model is to estimate the length of the QRS complex. To be able to achieve this using a sliding window approach, the window size must be able to fit the maximum length of QRS complex that we expect to encounter (or aim to detect).

As that would be the upper limit, most of the QRS complexes in the signal will have a shorter length. This means we can expect the QRS patterns to appear in different locations in the window — see Figures 3b and 3c.

We can also expect that for a select range of QRS-length and frequency of appearance of the pattern, no QRS-objects may land in a window — see fig. 4.

Because we have so many different cases, when preparing the training data we have to make sure that all of them are represented. Assuming that our original training dataset consist of long sequences, first we need to split the data into window length images add specify a label for each image. We expect that the labels are difficult to compute automatically with rule based algorithm,[1] so we rely on an expert to provide the labels. To achieve good generalization, our dataset should

[1]Otherwise there's really no need to train an ML model.

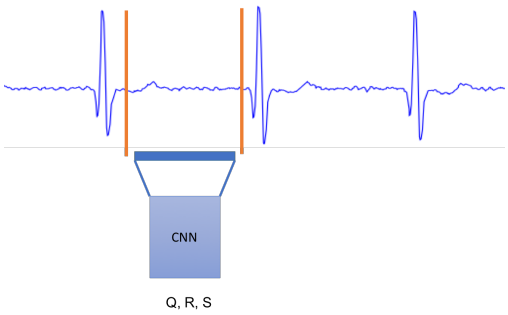


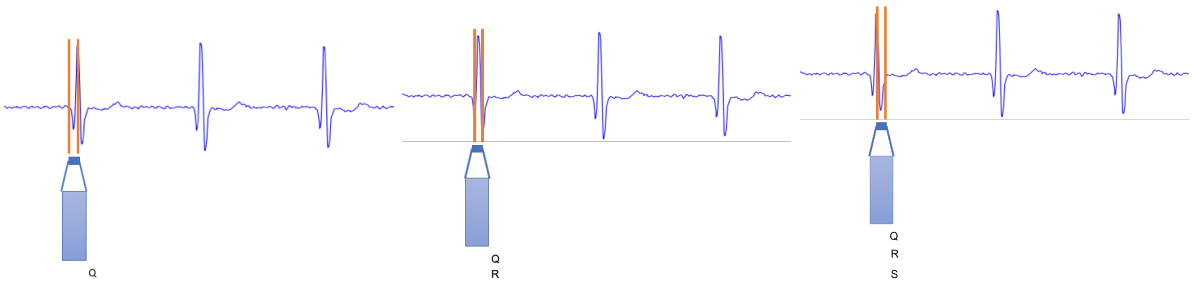
Figure 4: A window containing no QRS-complexes.

have a good coverage of cases with different lengths — from a typically shortest to the longest QRS complex. However, as we also expect that the object can appear at every location in the image, this adds another degree of freedom to the range of variations that need to be covered with training data. In other words, our training data should contain different length examples should in different locations of the window.

This need to cover all possible cases in the training data can become a significant drawback. Particularly in task where the correlations need to be uncovered from events that happen over longer distances. This means we need a large window size, that in turn adds computational complexity and increases the number of parameters significantly in the dense layers. Moreover, the need to cover all cases can mean we need to create a very large training set which can be laborious and costly.

6.1 A different type of model

In contrast, imagine a different type of model. One that looks at a very small window, detects the Q-wave. Then takes a number of steps until it reaches the R-wave. It then detects that one, moves forward until it reaches the S-wave and detects this last part of the pattern — see Figure 5. Can this model perform the same task that the CNN model did? What would this model need to be able to achieve the same task?



(a) The model first detects the Q-wave.

(b) It then continues to detect the R-wave.

(c) Finally it detects the S-wave.

Figure 5: A different type of model for sequential data.

Basically for such a model to detect the length of the QRS complex, it needs to be able to perform

the following tasks:

- Detect Q, and R, and S patterns
- Remember the the point of Q
- Remember the point of R
- Remember the point of S
- Compute the distance from Q to R to S

One requirement that is not met by the CNN is that this model needs *memory*. As we have seen so far all feed-forward models process the input directly to produce output and are not capable of storing information in between consecutive inputs. To meet these requirements we introduce the recurrent neural network (RNN).

7 Recurrent neural networks

7.1 Introduction

One type of network that can learn to perform the tasks above, is the *Recurrent Neural Network* (RNN). This type of network can

- process a sequence of datapoints one at a time;
- produce an output at each step[2];
- produce a memory state at each step;
- combine information from datapoints at different times to compute an output.

To achieve these feature the RNN model introduces a internal 'hidden state' and links with a delay, so that the hidden state at the end of one time step is used to do computations in the next — see Figure 6a.

On a high level the model now has the following components

- the input x_t
- the hidden state h_t
- the output y_t

each of these is indexed with the time step t. Here h_t is defined as a function of both x_t and h_{t-1}, that is

$$h_t = f_h(x_t, h_{t-1}),$$

and y is defined as in the MLP as a function of the hidden state of the model, so

$$y_t = f_y(h_t).$$

You can think of the connection between h_t and h_{t-1} as a delay node, as indicated in Figure 6a.

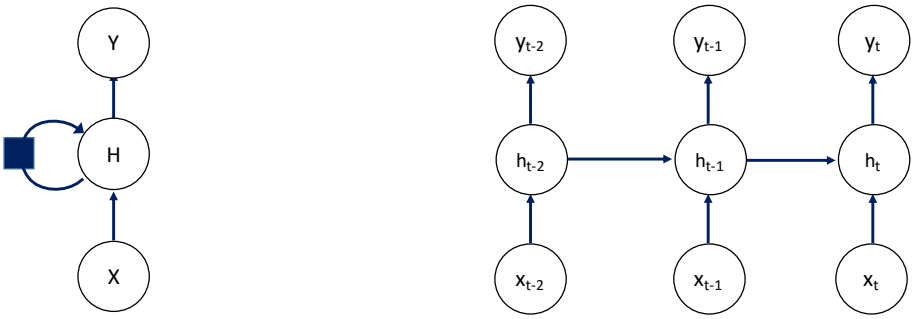
Another way that we look at the RNN models is in an unrolled form — see Figure 6b. This form is useful to look at the computations during training in detail. Note that here the edges represent the maps

$$f_h : x_t,\, h_{t-1} \mapsto h_t,$$
$$f_y : h_t \mapsto y_t.$$

These maps themselves do not depend on t, which means that the model reuses the parameters[3] for each time step.

[2] Although it doesn't have to.

[3] To prevent cluttering of the notation we don't include the parameters as subscript, but of course, as in the previous chapters, these functions are parameterized and the goal will be to find the right parameters again.



(a) The square in the connection from the hidden state to itself indicates a delay.

(b) We can display the same architecture in an "unrolled" way.

Figure 6: A schematic overview of an RNN.

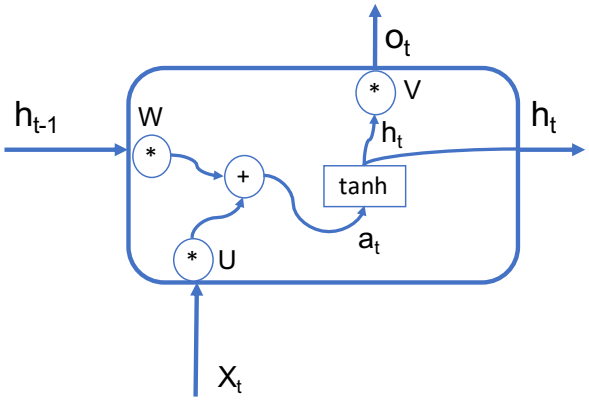


Figure 7: Vanilla RNN

We implement the conceptual model given in Figure 6a as follows:

$$h_t = \phi_1(Wh_{t-1} + Ux_t + b),$$
$$y_t = \phi_2(Vh_t + c),$$

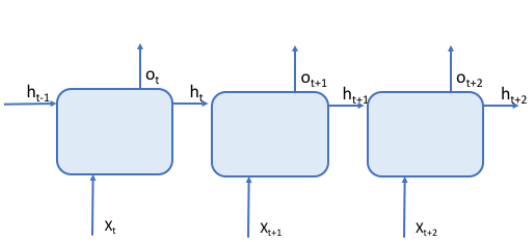
where W, U, b, V, c are parameters of the model, and ϕ_1 and ϕ_2 are activation functions. Figure 7 depicts the implementation diagram of the model, which we specifically refer to as the RNN cell.

If we use $\phi_1 = \tanh$ and $\phi_2 = \mathrm{id}$, we get the cell shown in Figure 7, where

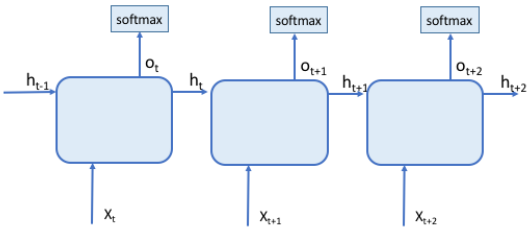
$$\left.\begin{aligned} a_t &= Wh_{t-1} + Ux_t + b, \\ h_t &= \tanh(a_t), \\ o_t &= Vh_t + c. \end{aligned}\right\} \tag{1}$$

Figure 8a depicts the unrolling of this model.

Let us look at some tasks for which we can use an RNN. For a task of sequence classification, we have the following training data: $D : \{(x_i)_0^{N-1}, (y_i)_0^{N-1}\}$. The the input sequence x is of length N and the target variable y is a assigned a value (label) from a discrete set of values $y \in S$.



(a) Unrolled RNN



(b) Unrolled RNN for sequence classification.

Figure 8: The RNN unrolled.

We can the use the softmax output function to model a discrete probability distribution over the set of classes in S, like in Figure 8b. This means that on top of eq. (1), we have

$$y^{(t)} = \text{softmax}(o^{(t)})$$

for the output.

A simple example of such a task would be to take the running sum of a sequence of integers modulo 10. In this case $(x_i)_0^{N-1}$ is a sequence of integers and $(y_i)_0^{N-1}$ is a sequence of elements from $S = \{0, \ldots, 9\}$.

Note: This task is a great example of a task that is not well motivated for a Machine Learning solution. It is no surprise that a ML model will do a substandard job at this task compared to any calculator as the calculator can easily achieve full accuracy.

In fig. 8b we can see this model unrolled. This model implements the conditional probability distribution over S given in eq. (2).

$$P(y_t|x_t, x_{t-1}, ..., x_0) \tag{2}$$

The output of such models is determined by selecting the most likely value for y_t at time t as eq. (3)

$$\arg\max_{s_t \in S} P(y_t = s_t|x_t, x_{t-1}, ..., x_0) \tag{3}$$

where s_t are is an element of $s_t \in S$.

In this case we select the value for y_t independently of other values y_t as shown in eq. (4).

$$P(y_0, ..y_N) = P(y_0)P(y_1)...P(y_N) \tag{4}$$

Architectures for other tasks

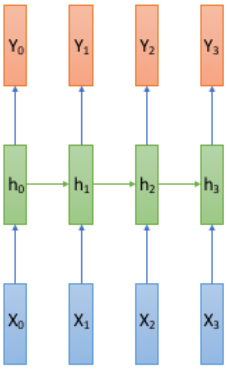
We give a schematic representation of the architecture we have discussed so far in Figure 9a. Other commonly used architectures with an RNN cell are depicted in Figures 9b to 9d.

The architecture shown in Figure 9b implements the model given in eq. (5), where we want to determine what class the entire sequence belongs to. In this case we have a single label y for the sequence of input $(x_t)_0^N$. We implement this by ignoring all except the last output of the RNN cell.

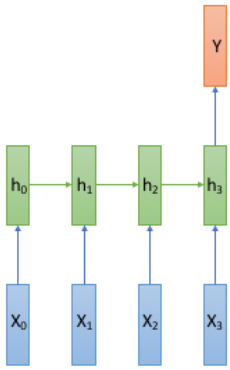
$$P(y|x_N, x_{N-1}, ..., x_0) \tag{5}$$

In contrast to this the architecture given in Figure 9c takes a single input x and produces a sequence of outputs $(y_t)_0^N$. An example of a task for which this model is well suited is the captioning of images. To implement this we can feed the (processed) image to the RNN in the first step, and in subsequent steps we can feed it an empty image or we can just feed it the same image in each time step.

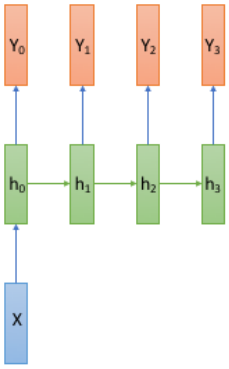
Another set of tasks fit into the description of sequence-to-sequence mapping. In these cases both the input and the output are sequences, but they are of different length. A solution using a single RNN cell is depicted in Figure 9d. One example for such a task is translation of a sentence from one language to another. Typically for such tasks a more complex architecture with multiple components (including more than one RNN cell) is better suited. We will discuss these sequence-to-sequence models in more detail later in this chapter.



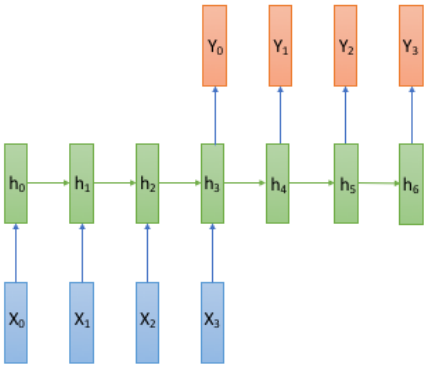
(a) RNN used for aligned sequence classification



(b) RNN used for sequence classification

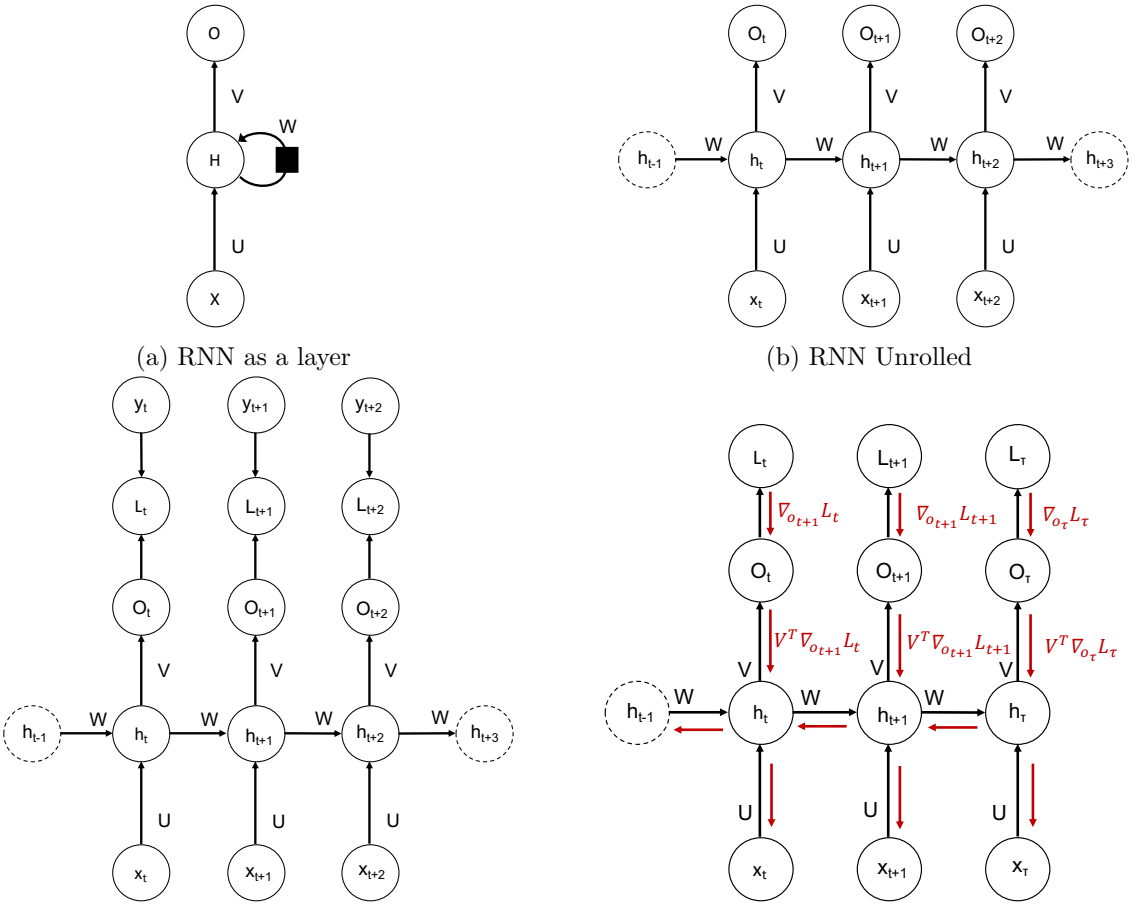


(c) RNN used for sequence generation



(d) RNN used for the sequence to sequence model

Figure 9: RNN architectures for various common types of tasks.



(a) RNN as a layer

(b) RNN Unrolled

(c) RNN including loss function unrolled.

(d) Backpropagation through time[4]

Figure 10: The training of an RNN unrolled.

7.2 Backpropagation through time

As with the other neural network models we have discussed so far, we use SGD to train the parameters of the RNN models. However, backprogatation through the delay node of the RNN cell cannot happen directly — from Figure 10a there is no obvious way to calculate the gradients directly. Rather, training of RNN models first requires that the model is unrolled as in Figure 10b. This variation of backpropagation is called *backpropagation through time*.

For simplicity, let us first look at the case where we only have a loss at the final time step, i.e.

$$L = L_\tau.$$

We will walk through this process for the RNN cell described in eq. (1).

We start the backpropagation from the end of the model as usual, and we go backward to the beginning.

[4]N.B. for a function $f : \mathbb{R}^n \to \mathbb{R} : \mathbf{x} \mapsto f(\mathbf{x})$ the gradient is the transposed of the total derivative, i.e. $\nabla_{\mathbf{x}} f = J_f(\mathbf{x})^\top = \left(\frac{\partial f}{\partial \mathbf{x}}\right)^\top$.

Then we take the gradient of each signal with respect to previous signals as given in eq. (7).

$$\begin{aligned}
\frac{\partial L}{\partial h_\tau} &= \frac{\partial L_\tau}{\partial O_\tau} V \\
\frac{\partial L}{\partial h_t} &= \frac{\partial L}{\partial h_\tau}\frac{\partial h_\tau}{\partial h_t} && \text{for all } t < \tau \\
\frac{\partial h_\tau}{\partial h_{\tau-1}} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_{\tau-1} + b))W \\
\frac{\partial h_\tau}{\partial h_t} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_{\tau-1} + b))W \cdots \operatorname{diag}(\phi'(W h_t + U x_t + b))W && \text{for all } t < \tau \quad (6) \\
\frac{\partial h_\tau}{\partial W} &= \frac{\partial h_\tau}{\partial W^{(\tau)}} + \frac{\partial h_\tau}{\partial h_{\tau-1}}\frac{\partial h_{\tau-1}}{\partial W} && (7)
\end{aligned}$$

Here by $\frac{\partial h_\tau}{\partial W^{(\tau)}}$ we mean the derivative we get when we view the W in every time step separately.[5]This becomes

$$\begin{aligned}
\frac{\partial h_\tau}{\partial W_{ij}^{(\tau)}} &= \operatorname{diag}(\phi'(W h_{\tau-1} + U x_{\tau-1} + b))(h_{\tau-1,j}\,\mathbf{e}^i), \\
&= \phi'(W h_{\tau-1} + U x_{\tau-1} + b)_i \cdot h_{\tau-1,j}\,\mathbf{e}^i,
\end{aligned}$$

where $\mathbf{e}^i$ is the vector with all zeros except at index i where it has a one.

In contrast to the feedforward networks, we see in that in the unrolled RNN, depicted in Figure 10d, that the parameters are shared at each time step. Therefore, the parameters will receive updates proportional to the gradients from the current time step and all future time steps.

To illustrate this let us look at how W is updated. For the model we are looking at, the derivative of the loss with respect to W is computed as

$$\begin{aligned}
\frac{\partial L}{\partial W} &= \frac{\partial L}{\partial h_\tau}\frac{\partial h_\tau}{\partial W} \\
&= \frac{\partial L}{\partial h_\tau}\left(\frac{\partial h_\tau}{\partial W^{(\tau)}} + \frac{\partial h_\tau}{\partial h_{\tau-1}}\frac{\partial h_{\tau-1}}{\partial W}\right) \\
&= \frac{\partial L}{\partial h_\tau}\sum_t \frac{\partial h_\tau}{\partial h_t}\frac{\partial h_t}{\partial W^{(t)}}.
\end{aligned}$$

If instead of only having a loss at the final timne step, we have a loss based on the output at every time step, i.e.

$$L = \sum_t L_t,$$

the gradient of the loss with respect to W becomes a sum over the gradients with respect to all of

[5]What we mean here is that we can see h_τ a function of the form $h_\tau(W, h_{\tau-1}(W))$ and compute the derivative wrt W as $\frac{\partial h_\tau(W,h_{\tau-1}(W))}{\partial W} = \frac{\partial h_\tau(A,h_{\tau-1}(B))}{\partial A} + \frac{\partial h_\tau(A,h_{\tau-1}(B))}{\partial B}$ where we set A and B to W. So with $\frac{\partial h_\tau}{\partial W^{(\tau)}}$ we mean $\frac{\partial h_\tau(A,h_{\tau-1}(B))}{\partial A}$ with A and B set to W.

those losses. That means we get

$$\begin{aligned}\frac{\partial L}{\partial W} &= \sum_{t\leq\tau} \frac{\partial L_t}{\partial W} \\ &= \sum_{t\leq\tau} \frac{\partial L_t}{\partial h_t} \sum_{s\leq t} \frac{\partial h_t}{\partial h_s} \frac{\partial h_s}{\partial W^{(s)}}. \end{aligned} \tag{8}$$

If we rearrange these terms, we can see how updates from future losses determine how the weights should be changed at a certain time step:

$$(8) = \sum_{s\leq\tau} \left(\sum_{t=s}^{\tau} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_s} \right) \frac{\partial h_s}{\partial W^{(s)}}. \tag{9}$$

Note that, as shown in eq. (6), $\frac{\partial h_t}{\partial h_s}$ is a product of $t-s$ terms, so the further in the future the loss is, the harder it becomes to get updates from it due to vanishing (or exploding) gradients.

To make this a bit clearer, let us look at the RNN unrolled, and let us say that instead of having the same weights W at every time step, we have different weight matrices $W^{(t)}$ at different times t. The term

$$\frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_s} \frac{\partial h_s}{\partial W^{(s)}}$$

then tells us how the weights $W^{(s)}$ at time s should be updated to decrease the loss L_t at time t. If all the matrices in the product $\frac{\partial h_t}{\partial h_s}$ have norm less than 1, this product will be smaller the more factors it has,[6]i.e. the further t and s are apart, and so the less effect the loss at time t will have on how we update the weights at time s.

The difference when we do share the parameters between time steps is that to get the gradient with respect to W we sum over all the gradients with respect to $W^{(s)}$, as shown in eq. (9). But since for every s we get very little information from losses far from s, this means our RNN has a hard time learning long range dependencies.

To recap, we can draw two conclusions. As the model is feeding forward to the update as well as to future steps, we get update values both from the loss at the current step as well as from all future losses. Secondly, as the parameters (in our example W) are re-used at each time step the full update for them is the sum over all time steps — see eq. (9).

This realization has important consequences. For one, we need to think differently about depth when using RNN models. Not only is depth in these models proportional to the number of layers, but it is also proportional to the number of steps in the input sequence. This implies that the same challenges we face when dealing with deeper networks — such as vanishing gradients — also come up when we try to use RNNs to model long sequences. Note that, in the computation of the gradients during backpropagation, the parameters of the model are a factor of a product with a length of up to the length of the sequence. This can lead to exploding or vanishing gradients. This limits the ability of this 'vanilla' version of the RNN cell to be used on longer sequences and to model longer term dependencies[7].

[6]Note that if ϕ is the hyperbolic tangent function, its derivative ϕ' is less than, or equal to 1, with equality only in 0.

[7]Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks 5.2 (1994): 157-166.

Different innovations in the RNN cell have been developed to add address these difficulties. Most notably the introduction of the Long short-term memory (LSTM) model[8]. This model introduces a protected cell that allows for gradients to flow freely across the different time steps. Before we go into the details of the LSTM we go over the gating mechanism that allows for developing protected cells.

7.3 Gating Mechanism

In the vanilla RNN we have discussed so far, the hidden state is computed as

$$\begin{aligned} a^{(t)} &= Wh^{(t-1)} + Ux^{(t)} + b \\ h^{(t)} &= \tanh(a^{(t)}). \end{aligned}$$

The hidden state at time t are directly affected by its values at the previous time, $t-1$, and the current input x_t. As such, to be able to store information in h needed to produce a specific output at a later time, the parameters in W need to be very well-tuned. They are responsible for making sure that information will be stored for a given input (and previous hidden values) as well as protected once stored until it is needed to produce the specific output. Finding such parameters during training is challenging. (You can read about the limitations in a theoretical study by Benio et al. 1994).

One of the main ways to overcome this problem is to *protect the state of the RNN*. That is, rather than updating the state with each datapoint, we learn

- when to update, given the input and the previous hidden state,
- what to update given the input and the previous state,
- even more so, what to remove (forget),
- and what to add into the memory

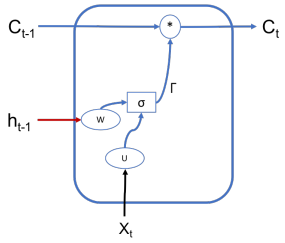
separately. We do this using "gates".

A gate is a mechanism that allows the model to select which information passes through and which is removed. In Figure 11a we introduce a memory cell C into the model. We can observe here how the information in C is controlled based on the values of the hidden state h and the input x — see eq. (10). Specifically, the multiplicative[9] gate in Figure 11a will let information pass if the sigmoidal activations are 1 and will remove information if the activations are 0. Therefore, the parameters in this gate, W_Γ and U_Γ, determine how the hidden state and input remove information from the memory cell C.

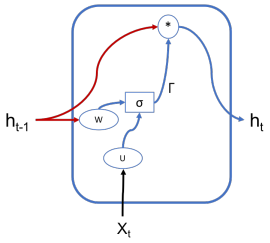
$$\left.\begin{aligned} \Gamma &= \sigma(W_\Gamma h_{t-1} + U_\Gamma x_t + b) \\ C_t &= \Gamma \cdot C_{t-1} \end{aligned}\right\} \tag{10}$$

[8]Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[9]The multiplication is performed element-wise.



(a) We can use a gate to remove information from the memory cell, C. See eq. (10) for details.



(b) We can also apply a gating mechanism to the hidden state directly. See eq. (11) for details.

Figure 11: Gating mechanisms

Such mechanism can be even applied to the hidden state itself without introducing a memory cell as shown in eq. (11) and Figure 11b.

$$\left.\begin{aligned} \Gamma &= \sigma(W_\Gamma h_{t-1} + U_\Gamma x_t + b) \\ h_t &= \Gamma \cdot h_{t-1} \end{aligned}\right\} \tag{11}$$

Throughout this chapter, σ denotes a logistic sigmoid function.

7.4 Long short-term memory

The Long short-term memory, or LSTM, architecture introduces a memory cell and a number of gates to control the access to this cell and to how the output is produced. The memory cell can flow directly from one step to another, see Figure 12a. Its value is affected by the forget gate and the add gate.

The forget gate multiplies the cell values element-wise with vector values in the range $[0, 1]$. The values of the gate are specified by the hidden state h_{t-1} and the input x_t as in eq. (12).

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{12}$$

The add gate determines the information that is added to the cell state. The LSTM cell determines the candidate value C'_t based on the values of the hidden state h_{t-1} and the input x_t as shown in eq. (13). These candidate values are passed through the add gate as in eq. (14).

$$C'_t = \tanh(W_C[h_{t-1}, x_t] + b_C) \tag{13}$$

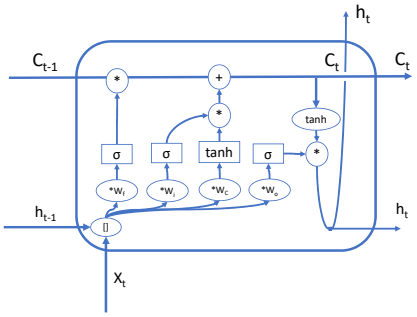
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{14}$$

This makes that the new cell state becomes

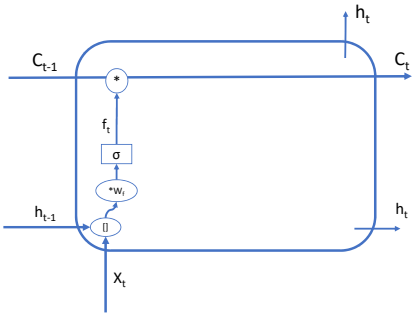
$$C_t = f_t \cdot C_{t-1} + i_t \cdot C'_t.$$

The output of the LSTM is computed based on the cell state, however, it also gated by the values of the hidden state and the input as shown in Figure 12d and eq. (15).

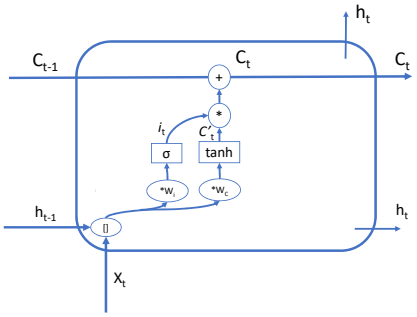
$$\left.\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \cdot \tanh(C_t) \end{aligned}\right\} \tag{15}$$



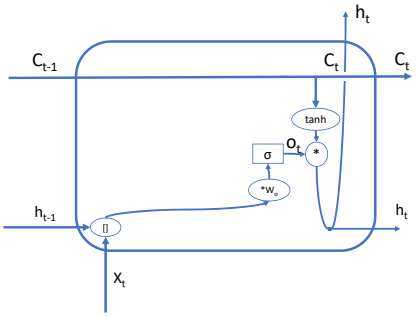
(a) The full cell



(b) *Forget gate*



(c) *Add gate*



(d) *Output gate*

Figure 12: The LSTM cell with its gates.

In summary in the LSTM RNN cell, in contrast to the 'vanilla' RNN cell, the hidden state and the input control the flow of information in and out of the cell rather than carrying the information that the model need to produce the output. Analogously when training this model the updates coming from the gradients of the loss can flow back through the cell values. This in turn allows for training parameters of all the gates such that the model can learn long term dependencies in the data much more efficiently. Note that in this LSTM architecture, the hidden state h_t is used as the output of the cell.

7.5 Gated Recurrent Unit

Since its introduction there have been many variations proposed on the LSTML model (including some by the original authors). Nevertheless, the LSTM model still maintained its good standing when it comes to perfomance. In this subsection we consider one variation of the LSTM model that showed some improvements in performance on certain datasets. This, however, is not the motivation to introduce this model, but rather to illustrate how the hidden state can take the role of the cell, while the gating mechanisms still provide the advantage of being able to capture long term dependence. We briefly mentioned before that the gate can be applied to the hidden state. This approach is introduced in the gated recurrent unit (GRU) model shown in Figure 13. In that sense, the GRU model is a simplificaiton of the LSTM and has fewer parameters.

The GRU has two gates: a "reset gate" r and an "update gate" z. The update gate determines what information in the hidden state should be updated, and the reset gate determines what information

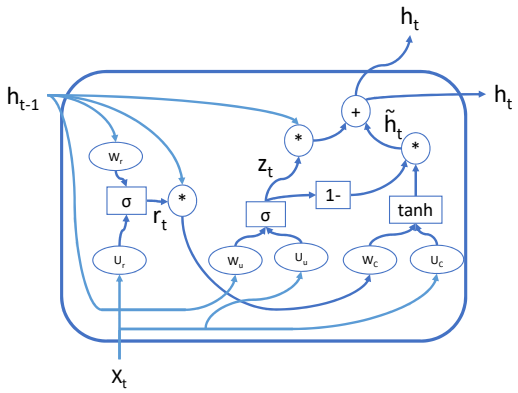


Figure 13: Gated Recurrent Unit

in the old hidden state should be ignored when computing the update. The gates are computed as

$$z_t = \sigma(W_u h_{t-1} + U_u x_t + b_u),$$
$$r_t = \sigma(W_r h_{t-1} + U_r x_t + b_r),$$

and the proposed update is computed as

$$\tilde{h}_t = \tanh(W_c(r_t \cdot h_{t-1}) + U_c x_t + b_c).$$

Finally the new value for the hidden state is then

$$h_t = z_t \cdot h_{t-1} + (1 - z_t) \cdot \tilde{h}_t.$$

8 RNN models

We look at a number of models that include RNN cells for different tasks.

8.1 Sequence classification

For a task of sequence classification, where a variable length sequence needs to be classified or regressed to a single value we can often use the model in Figure 14. This model is especially suitable for situations where the input comes from a fixed-size set such as the characters in the alphabet.

The RNN cell can also be used as a component of more complex models depending on the structure of the input data. For example, let us look at the motivating example at the beginning of this chapter again: the QRS complex. There we have both tasks based on localized correlations — detecting the separate Q, R, and S waves — as well as a task based on the longer term correlations of following the sequence of those 3 events and calculating the length of the complex. With this in mind, instead of only using an RNN, as in Figure 15a, we might want to use a combination of convolutional layers and an RNN cell as in Figure 15b.

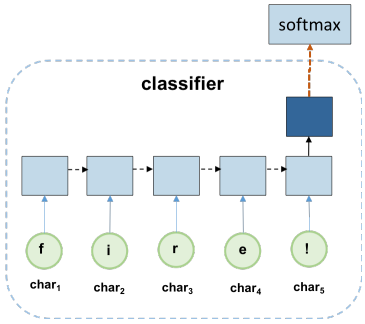
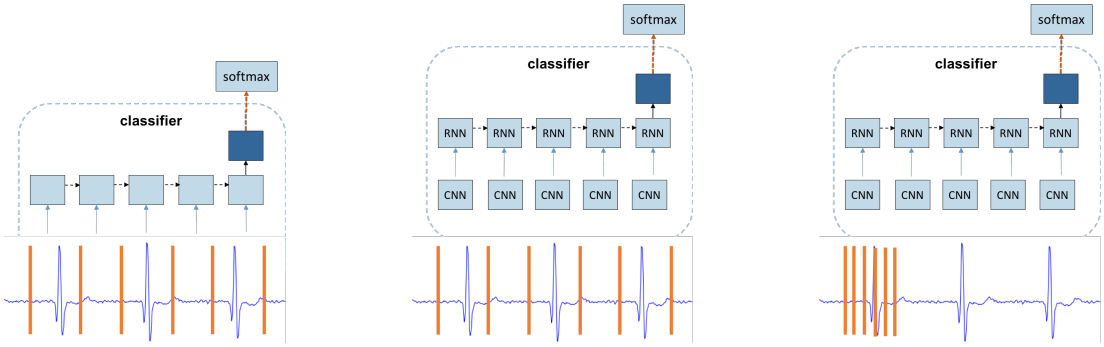


Figure 14: Sequence classifier - Classify variable length sequences



(a) Using just an RNN to measure the length of the QRS-complex.

(b) We can combine the RNN with convolutional layers.

(c) Because the convolutional layers only need to detect highly localized phenomena, the window size can be small.

Figure 15: Models for measuring the length of the QRS-complex.

Note: as we now have an RNN cell to take over the job of detecting the full QRS complex, the responsibility of the convolutional layers is reduced to only detecting the individual waves and hence the convolutional window can be much smaller making the model more efficient. This is depicted in Figure 15c.

Let us look at the problem of sentiment detection in natural text as another example. This is where a short text is classified with a sentiment labels, like positive or negative. Instead of learning representations from scratch, we can use an embedding layer that is pre-trained using the CBOW or Skipgram model. This pre-training does not need labeled data, so we can train the embedding layer on a much larger corpus, especially when training data for the classification task is limited. Alternatively we can make use of a pre-trained embedding layer that is publicly available. Both options are examples of transfer learning. The overall architecture would come down to the one shown in Figure 16.

8.2 Bi-directional RNN

In sequence classification tasks the model processes the whole sequence and produces the class estimation. In such cases it may be useful to process the sequence in both directions before producing

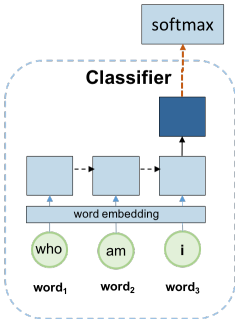
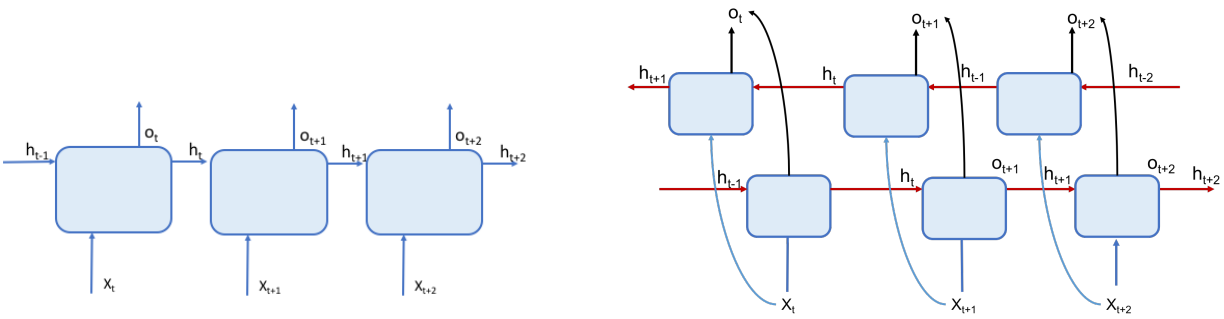


Figure 16: Text classification with RNN and embedding



(a) The RNN we have discussed so far processes the data in one direction.

(b) A bi-directional RNN on the other hand processes the sequence in both directions simultaneously

Figure 17: A regular RNN versus a bi-directional RNN.

the estimation. As a variation of RNN, the bi-directinal RNN[10] actually processes the data in both directions to produce an estimate — see Figures 17a and 17b.

8.3 Sequence to sequence models

So far we have looked at models where either every element of a sequence needs to be classified, or where a sequence needs to be classified as a whole. But sometimes we want to map variable length sequences to other sequences that might not be aligned and have different lengths. Think for example of translating sentences from one language to another. The translated sentences might have very different lengths, and the word order might be completely different. For these kinds of tasks we have sequence to sequence (seq2seq) models.

In general, we would like to have a model of the joint distribution of a sequence $y_0, ...y_n$, given a sequence $x_0, ..., x_k$, as shown in eq. (16). Having such a model allows us to find the most likely sequence $y_0, ...y_n$ for a given input sequence $x_0, ..., x_k$ — i.e. to find eq. (17). From the point of view of the implementation typically there are two components: a model that estimates the joint probability distribution and a search algorithm that produces the sequence that maximizes this probability.

[10] Graves, Alex, Santiago Fernández, and Jürgen Schmidhuber. "Bidirectional LSTM networks for improved phoneme classification and recognition." 2005

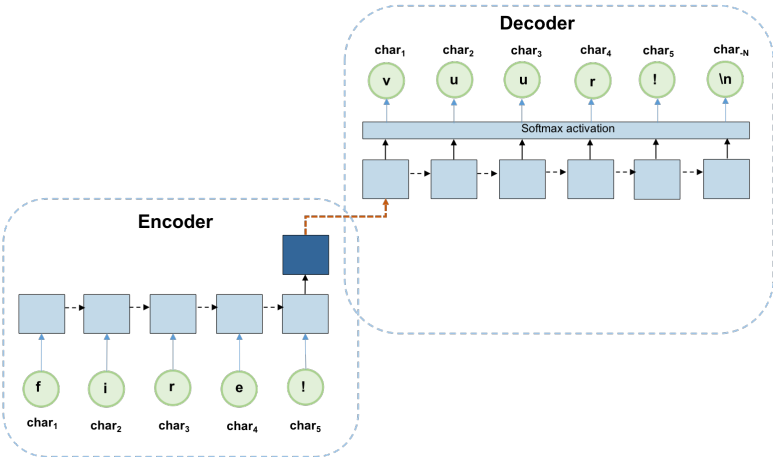


Figure 18: Seq2seq model

$$P(y_0, y_1, ..y_n | x_0, x_1, ...x_k) \tag{16}$$

$$\arg\max_{y_0,...y_n} P(y_0, y_1, ..y_n | x_0, x_1, ...x_k) \tag{17}$$

Figures 9d and 18 depict the conceptual view of a seq2seq model. The seq2seq models are typically implemented with two RNN cells, where one cell acts as an encoder of the input and the other acts as decoder that produces the output.

The encoder's output or its hidden state at the end of the sequence can be used as the *encoding* of the input sequence. This *code* is then used as an initialization of the hidden state of the decoder. In such a configuration the decoder models eq. (18) where h_k is the code.

$$P(y_0, y_1, ..y_n | h_k) \tag{18}$$

Notice the similarities with the autoencoders from Chapter 4 in both terminology and structure. Both models have an encoding and a decoding part. We have a representation of the input produced by the encoder. If we train the seq2seq model with the same input as the output it does take on the role of a sequence autoencoder and can be used for representation learning.

There are variations on the seq2seq model changing e.g. what is used as the code and how the decoder is initialized (conditioned) by the code. There may be specific reasons to use such a variation for some applications, but here we limit the discussion to the given setup.

As the goal of our task is to maximize the joint probability of the output sequence, eq. (17), the model configuration given in Figure 18 actually assumes that the next output symbol is independent of the previously emitted symbols in the sequence, see eq. (19).

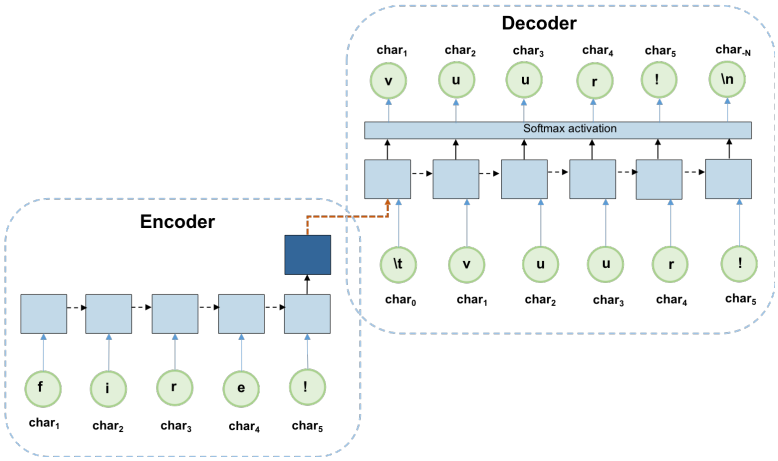


Figure 19: Seq2seq model without independence assumption on the output sequence

$$P(y_0, ...y_n|x_0, ...x_k) = P(y_0|x_0, ...x_k)...P(y_n|x_0, ...x_k) \tag{19}$$

In this case, computing the target, eq. (17), is reduced to greedily finding the y assignment that maximizes the distribution at that time, as shown in eq. (20).[11]

$$\underset{(y_0,...y_n)}{\arg\max}\, P(y_0, y_1, ..y_n|x_0, x_1, ...x_k) = \left(\underset{y_i}{\arg\max}\, P(y_i|x_0, x_1, ...x_k)\right)_{i=0,...,n} \tag{20}$$

In a more general setting, we cannot assume that the output sequence consist of independent symbols. We therefore condition the decoder on the previous selected output as shown in Figures 19 and 20b.

In this case, however, a greedy solution will not guarantee finding the sequence that maximizes the joint probability since selecting the most likely symbols early on can lead us to a path of unlikely symbols further in the sequence. To guarantee the maximum joint probability we would need to search through the whole space of output sequences. For long sequences this search would be computationally expensive.

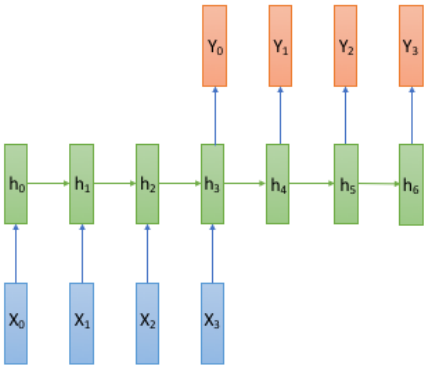
Sequence to sequence models — Beam Search

For each step in generating the output sequence we can branch-out in a number of branches equal to the number of possible output symbols. As such the search can well be structured as a tree, and tree search algorithms such as depth first or bread first search are applicable for solving eq. (17).

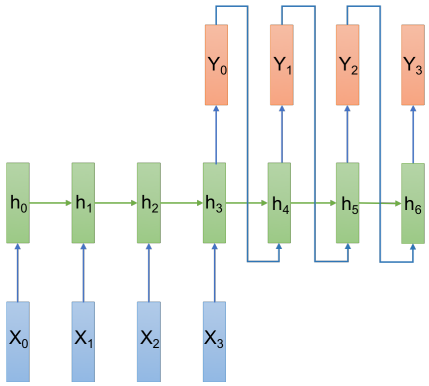
To avoid the high computational complexity we may relax the requirement for a maximum guarantee and use a heuristic approach that comes with a significantly lower computational cost as the sequence length increases. One commonly used search algorithm is Beam Search.[12]

[11] We assume independence of previous values, but not of index, so this optimal y needn't be the same at every index.

[12] Furcy, David, and Sven Koenig. "Limited discrepancy beam search." IJCAI. 2005



(a) Seq2seq model assuming independent outputs.



(b) Seq2seq model without independence assumption on the output sequence.

Figure 20: Seq2seq with and without the independence assumption.

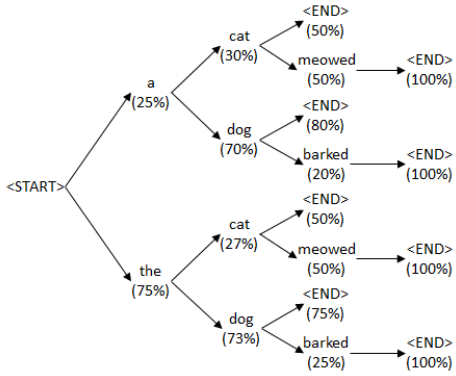


Figure 21: Generating a sequence - Beam Search

Beam search maintains a list of candidate sequences (a beam) that maximize the given metric and expands the search from that list rather that doing an exhaustive search — see Figure 21 for an illustration of this. In most practical applications the distribution of the solutions is such that Beam Search finds the most likely solution. That is why the algorithm is commonly used in combination with seq2seq models.

Sequence to sequence models — Example

To get a better intuition of the seq2seq models, let us consider a practical setting. The task is translation of a sentence from one language to another. We are given training data $D : (x, y)$ where x and y are sequences of words with different lengths in two different languages. In order to learn efficiently, we want to use meaningful representations of the words, so we use pre-trained word-embeddings. Alternatively we can train word embedding matrices in the model directly. Overall this gives us the architecture shown in Figure 22.

There are a number of decisions highlighted in this solution. The *encoding* produced by the encoder

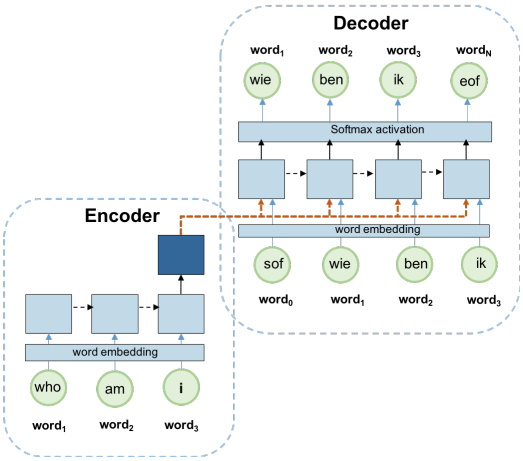


Figure 22: Seq2seq model for translation

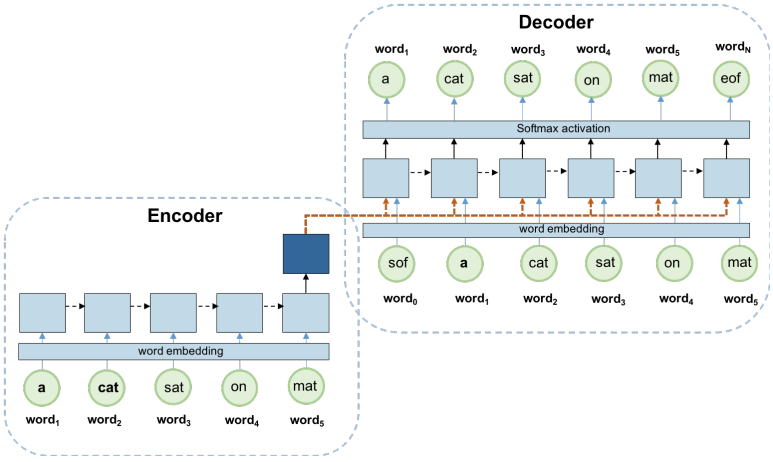


Figure 23: Seq2seq autoencoding

is presented to the decoder at each time step rather than only as initialization of its hidden state. The decoder is presented with the previous symbol (element) of the output at each decoding step. An embedding layer is present both for the encoder and decoder and is used to represent the words (symbols). Given that the words come from different vocabularies the embedding matrices are different.

Note: During inference the model provides a probability distribution over the possible sequences and we choose to use Beam Search to find a list of likely solutions. However, during training we typically only present the model with the solution given in the training (assuming we are given one target sequence per input sequence). Therefore, during training we do not run Beam Search for the decoder. Instead we compute the negative log-likelihood of the target sequence according to the computed distribution, and try to minimize this. [13]

An autoencoder for sequences using the same architectural choices is shown in Figure 23

[13]Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le. "Sequence to sequence learning with neural networks." Advances in neural information processing systems. 2014.

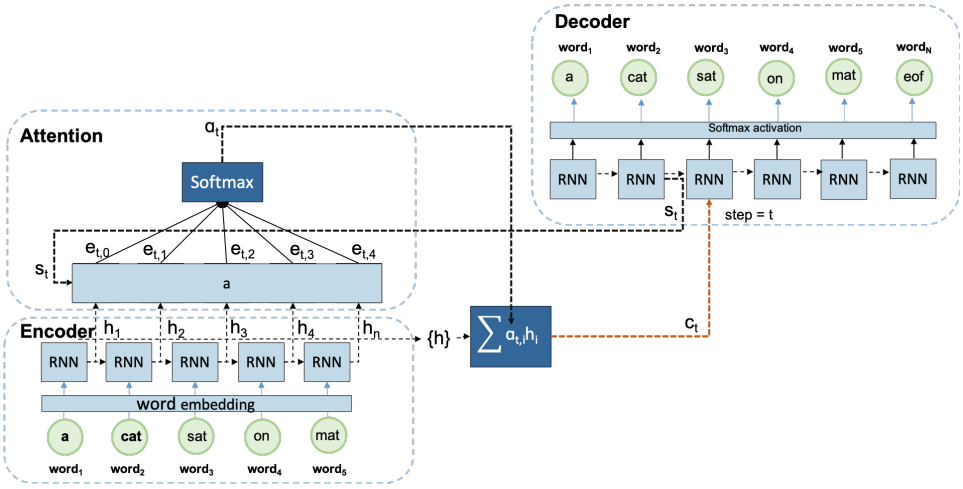


Figure 24: Seq2seq with attention

8.4 Attention Mechanism

Even though the seq2seq model is an efficient architecture for developing a sequence representation and mapping a sequence to another sequence, work in the field has brought up arguments that encoding the whole sequence to a single vector, as shown in Figure 22, may be challenging. Or in other words, finding the parameters for a seq2seq model with SGD and backprop may be difficult given the structure of their architecture.

Instead of using a single *code*, we can use a model with an "attention mechanism"[14]. The high-level idea of these models is that rather than using a code to store all the information about the input sequence, we should use it as a context and enable the decoder to have a look-back mechanism that gives it access to the input data.

Let use consider the following illustrative example. For the task of text summarizing, we are given one longer text sequence for which we need to produce a shorter text sequence. The shorter version of the text should capture the meaning of the longer one with fewer words. Suppose we develop a model for this task with the same architecture as in Figure 22. With this architecture the decoder model needs to be able to generate summarizing sequences of text based on the given *encoding* of the long sequence. However, when writing a summary, one can often copy certain words or short phrases from the original text. If the decoder had access to the original text, it could develop a mechanism for copying in addition to generation, which could make it more efficient or more precise.

In general the idea is that if the decoding model can 'attend' to the input sequence, this model can become more efficient, or we can become have more efficient training of such models in terms of the amount of training data or complexity of the model. This feature is referred to as the attention mechanism.

In Figure 24 we can see a seq2seq model with attention. In this model the decoder receives an

[14]Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." (2014).

additional input c_t at each time step. The vector c_t is a convex combination of the representation (or in this case the hidden state) of the input sequence produced by the encoder at each time step. So, now the decoder not only has access to the representation of the whole sequence but also to the individual elements of the input. The model also learns how to 'attend' to the individual inputs with different weights. In Figure 24, this is done using the α (attention) vector. The attention vector α is indexed by the time step t, meaning that the decoder 'attends' differently to the input at each time step.

The α is normalized to sum up to one, so we can view it as a probability distribution over the input. It is formed by a dense layer that inputs the encoding vectors h and a decoder state vector s (typically based on the hidden state of the decoder). The hidden layer produces the logit values of the distribution of the attention that is then normalized with a softmax activation function.

Let us look at how a model using this kind of attention works in detail.[15] We have an input sequence $\{x_0, x_1, ..., x_k\}$, and want to model $P(y_1, \ldots, y_n \mid x_1, \ldots, x_k)$. To do this, we first process the input sequence using a word embedding and a GRU[16] to come to hidden states $h_1, \ldots, h_k$, so

$$h_t = f(x_t, h_{t-1})$$

where f is the combination of a GRU cell and an embedding function.

The decoder too uses a GRU with hidden state s_k where s_0 is initialized as

$$s_0 = \tanh(W_s h_1).$$

The difference is that at every time step i, the decoding GRU is presented with a (different) context vector c_k. This context vector is computed as follows. First we compute the logits for the weights of the attention vector

$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

from which we compute the attention vector as

$$\alpha_k = \text{softmax}(e_k).$$

This attention vector is then used to compute the context vector as

$$c_i = \sum_j \alpha_{ij} h_j.$$

We can then feed the context vector, together with the hidden state and the previous output to the decoding GRU to get the next hidden state, and thu the next context vector

$$s_i = g(y_{i-1}, s_{i-1}, c_i).$$

To get the output, we can now feed the previous output, the hidden state, and the context vector to an MLP with softmax output:[17]

$$P(y_i \mid x_1, \ldots, x_k, y_1, \ldots, y_{i-1}) = \text{softmax}(W_o s_i + C_o c_i + U_o E y_{i-1})$$

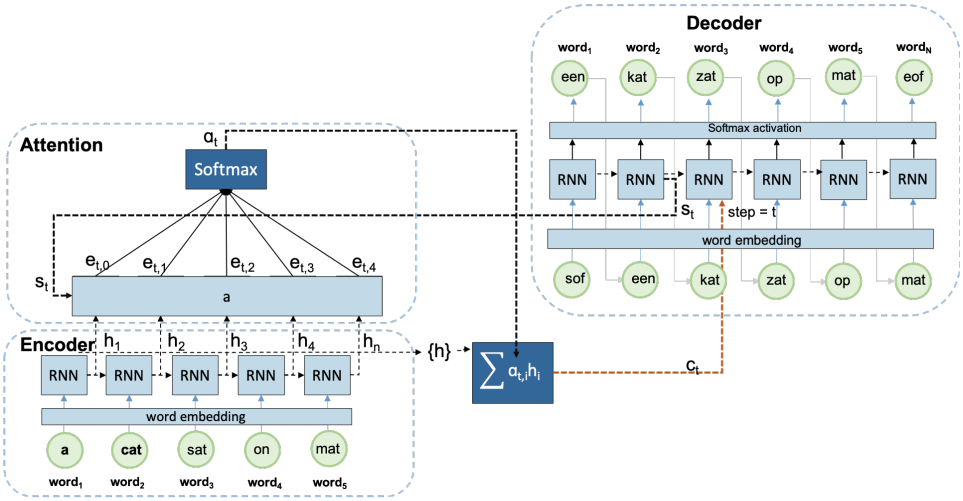


Figure 25: Attention mechanism

where E is an embedding matrix. This architecture is shown in Figure 25 for the different task of translating sentences from English to Dutch.

With this architecture, the decoder has a mechanism that allows it to access the (processed) input as an external memory. It can use the s_t signal as an addressing mechanism. In this way the model does not need to store all information needed for the decoding in the *encoding* of the input sequence, which can be particularly effective if the decoder needs to copy information from the input sequence.

Note: The weights of the attention vector α can also be informative of the operations of the model. Specifically, it can be interesting to look at what parts of the input was attended to for what parts of the output. This may be particularly useful in translation (or generally transducing) tasks, where a decision regarding an output word can be 'explained' by to what the model is attending to. Another perspective of the attention is as a means of alignment. The seq2seq models generally map sequences without indicating alignment. In other words, the model does not indicate how each output signal is aligned to an input symbol. For example in a event detection task, the input sequence may be a high dimensional signal that needs to be mapped to a sequence of events (e.g. ECG signal that is mapped to a sequence of Q, R, S-wave detections). In such a setting the attention can indicate alignment of the input to the output symbols that may be useful for diagnosing the model or even as a feature of the model.

[15] For simplicity we have omited writing biases.

[16] In the original paper on attention, a bi-directional GRU was used, but we want to keep the example simple.

[17] In the original paper some more processing was done to come to an output distribution, but this changes nothing to the working of the attention mechanism, so this is besides the point.

Generative models

Deep Learning (2IMM10)

Vlado Menkovski, Simon Koop

Spring 2020

6 Deep generative models

6.1 Motivation

Point estimate models So far we covered different models developed from data. Specifically for different tasks we developed a formulation where the model maps an input x to the an output y ($f_\theta(x) = y$) or more generally, models that describe a distribution over the values of y given a value for x ($f_\theta(x) = P(y|x)$). In this setting, we reduced the task as finding the parameters θ that minimize some error $L(y, \hat{y}) = L(y, f_\theta(x))$ given a training dataset $D : \{x^{(i)}, y^{(i)}\}_{i=0}^{N}$. We then solve this task $\theta = \arg\min_\theta L(y, f_\theta(x))$ with the SGD algorithm. We refer to the $f_\theta(x) = P(y|x)$ models as discriminative models. An example of such a model is given in fig. 1.

For a binary classification task such as this we define the model as in eq. (1).

$$P(y = c|X = (x_0, x_1)) = \frac{1}{1 + e^{(w_0 x_0 + w_1 x_1 + b)}} \tag{1}$$

In contrast to such a model we can also develop a model of the distribution of the data itself fig. 2.

$$P(X, Y)$$

- describes the relationship between X and Y. This means that we would like our model to express the likelihood of observing the different values of x and y This can be defined as $P(x, y)$ (probability of x and y rather than probability of x given y), the joint probability of x and y.

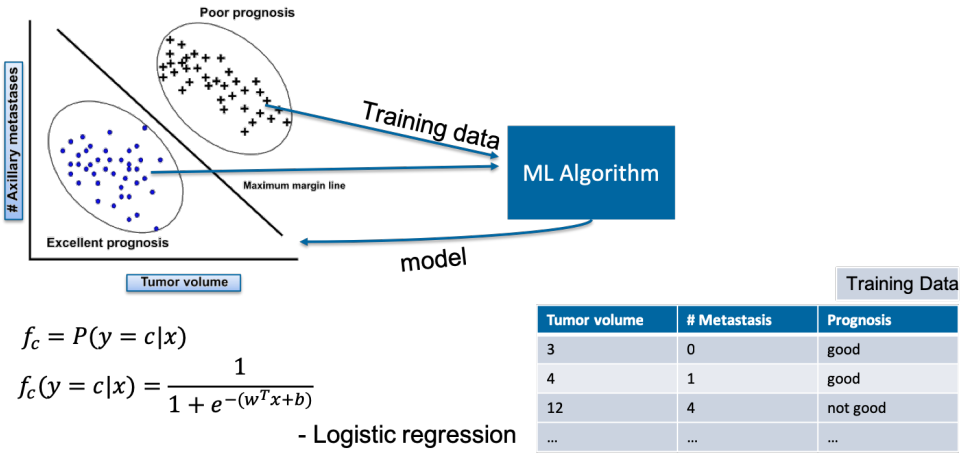


Figure 1: Discriminative model

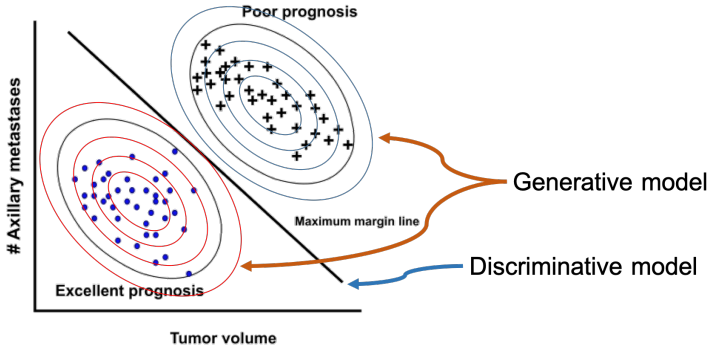


Figure 2: Generative models intuition

Let us for example take the following model:

$$P(X|Y=0) \sim \mathcal{N}(\mu_0, \Sigma_0)$$

$$P(X|Y=1) \sim \mathcal{N}(\mu_1, \Sigma_1)$$

Given such a model, we can compute the most likely class assignment for a datapoint with the following expression:

$$\hat{y} = \arg\max_c P(y=c) P_\theta(X|y=c)$$

Where θ are the parameters for the distribution.

Important: In the rest of the lecture notes we will use terminology and notation from probability theory and Bayesian inference that you need to be familiar with. Please refer to section .1 to familiarize/refresh yourself with the needed background.

6.2 Training a probabilistic model

An important question we have to ask ourselves when making a generative model, is how are we going to measure its performance, and how are we going to train it. We will discuss two commonly used frameworks for doing this:

- Maximizing the (log-)likelihood of the model given the data.
- Minimizing a divergence (or other notion of distance) between our model distribution and the (hypothetical) true or empirical distribution.

These two frameworks are not mutually exclusive. In fact, the cross-entropy loss we have seen popping-up in many instances, fits perfectly in both frameworks.

Maximizing the log-likelihood

The likelihood of the data given the model is an often used to select the best parameters. The parameters maximizing this likelihood are called the maximum likelihood estimate (MSE). The way this works is that we have a model for our data, $X = \{x^{(1)}, \ldots, x^{(n)}\}$, in the form of a parameterized

Figure 3: Normal distribution

probability distribution P_θ with density p_θ, and we assume all data points are independent and identically distributed. The joint density is then simply the product of the density for a single random variable, so that the likelihood of the total data set is given by

$$p_\theta(X) = \prod_{i=1}^{n} p_\theta(x^{(i)}).$$

In order to maximize this, we take the logarithm to obtain

$$\log p_\theta(x) = \sum_{i=1}^{n} \log(p_\theta(x^{(i)}))$$

and try to maximize this *log-likelihood*. We illustrate how this works for a simple distribution in the following subsection.

Illustrative Example

To see how this likelihood maximization traditionally works, let's look at a classical statistics exercise. We are given a set of numbers,

$$X = \{x^{(1)}, ..., x^{(m)}\} \subset \mathbb{R}$$

and assume that these numbers are realizations of random variables that are independent and identically distributed (i.i.d.). We define a parametric family of densities $(p_\theta)_{\theta \in \mathbb{R}^d}$ and want to find the parameters of the distribution that maximize the likelihood of our data:

$$\hat{\theta} \leftarrow \arg\max_{\theta \in \mathbb{R}^d} p_\theta(X) \tag{2}$$

For our family of distributions we choose the normal distribution, $\mathcal{N}(\mu, \sigma^2)$, so that

$$p_\theta(x) = \mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

where $\theta = \{\mu, \sigma\}$. With this choice for our family of distributions, our objective, eq. (2) boils down

to

$$\begin{aligned}\hat{\theta} &= \arg\max_{\theta\in\mathbb{R}^2} \prod_{i=1}^{m} p_\theta(x^{(i)}) \\ &= \arg\max_{\theta\in\mathbb{R}^2} \frac{1}{m}\prod_{i=1}^{m} p_\theta(x^{(i)}) \\ &= \arg\max_{\theta} \frac{1}{m}\sum_{i=1}^{m} \log p_\theta(x^{(i)}) \\ &= \arg\max_{\mu,\sigma} \frac{1}{m}\sum_{i=1}^{m} \frac{-(x^{(i)}-\mu)^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma^2). \end{aligned} \tag{3}$$

To find this maximum, we first take the derivative with respect to μ:

$$\frac{\partial}{\partial\mu}(3) = \frac{-1}{m\,\sigma^2}\sum_{i=1}^{m}(x^{(i)}-\mu).$$

This value is 0 if and only if

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)},$$

so to maximize the log-likelihood of our data set, we must take

$$\hat{\mu} = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}.$$

Similar procedure (differentiating to σ^2) gives us

$$\hat{\sigma}^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\hat{\mu})^2.$$

Maximizing the log-likelihood — continuation

The generative models we deal with in this course typically have a much more complex relation between their parameters and the value of the log-likelihood. Instead of solving things by setting derivatives to 0, we use gradient descent and backpropagation to maximize the log-likelihood (or minimize the negative log-likelihood). Moreover, instead of computing the log-likelihood for the full dataset, we approximate it using smaller batches of datapoints.

Minimizing the distance between distributions

When we have a probabilistic generative model and a dataset, we can see this as having two distributions: a model distribution, and the true distribution we are trying to model. The goal is to get these two distributions to match. To do this, we could try to minimize some kind of distance between the two distributions.

There are many notions of distance between probability distributions available to us, all with their own strengths and weaknesses. Some notions of distance that are popular in machine learning are

Kullback-Leibler divergence, Jensen-Shannon divergence, the Weierstrass metrics, and Maximum Mean Discrepancies (MMD). Here we shall focus on the Kullback-Leibler divergence and the Jensen-Shannon divergence.

If we have two discrete probability distributions Q and P on the sample space S, the *Kullback-Leibler divergence* from Q to P is defined as

$$D_{KL}(Q \mid\mid P) = \sum_{s \in S} Q(s) \log \left(\frac{Q(s)}{P(s)} \right),$$

For distributions corresponding to continuous random variables with range $S \subset \mathbb{R}$, the Kullback-Leibler divergence is given by

$$D_{KL}(Q \mid\mid P) = \int_S q(s) \log \left(\frac{q(s)}{p(s)} \right) \mathrm{d}s,$$

where p and q are the densities of P and Q. The definition for joint distributions is analogous.

If Q is the true distribution, and P is an approximation of this distribution, the Kullback-Leibler divergence from P to Q can be interpreted as the amount of information lost by using the approximation over the true distribution. It has the following properties:

- The divergence is non-negative — i.e. $D_{KL}(Q \mid\mid P) \geq 0$ — with equality if and only if $P = Q$;
- the divergence is not symmetric, i.e. $D_{KL}(P \mid\mid Q) \neq D_{KL}(Q \mid\mid P)$ in general;
- it sums over dimensions, i.e. if $P(x, y) = P_1(x)P_2(y)$ and $Q(x, y) = Q_1(x)Q_2(Y)$, then

  $$D_{KL}(Q \mid\mid P) = D_{KL}(Q_1 \mid\mid P_1) + D_{KL}(Q_2 \mid\mid P_2)$$

  and similar for continuous distributions.

As we will see in Section 6.2, maximizing the log-likelihood is equivalent to minimizing the Kullback-Leibler divergence between the model distribution and the empirical distribution of the data. The Kullback-Leibler divergence also plays an important role in the Variational Auto-Encoder discussed in Section 6.6.

Another divergence is the *Jensen-Shannon divergence*. This divergence can be seen as a symmetrized version of the Kullback-Leibler divergence, and is given by

$$D_{JS}(P \mid\mid Q) = \frac{1}{2} D_{KL}(P \mid\mid M) + \frac{1}{2} D_{KL}(Q \mid\mid M)$$
$$M = \frac{P + Q}{2}.$$

This divergence has the following properties:

- it is symmetric, i.e. $D_{JS}(P \mid\mid Q) = D_{JS}(Q \mid\mid P)$;
- it is bounded by $0 \leq D_{JS}(P \mid\mid Q) \leq \log(2)$;

- it is 0 if and only if the distributions mach, i.e. $D_{JS}(P \mid\mid Q) \iff P = Q$.

The training objective of the original GAN can be seen as minimizing the Jensen-Shannon divergence.

When we train a generative model by minimizing some notion of distance, we usually see the dataset, $X = \{x^{(1)}, \ldots, x^{(n)}\}$ as drawn from independent random variables that all have the same true probability distribution[1], $q(x)$, often called the empirical distribution. Our generative model implements an approximation $p_\theta(x)$ to that distribution and we want to pick our model-parameters θ such that some distance, e.g. the Kullback-Leibler divergence, or the Jensen-Shannon divergence, is minimal.

Cross-entropy

So far we have discussed two common frameworks for training a generative model:

1. maximizing the log-likelihood of the data according to our model distribution;
2. minimizing some distance between our model distribution, p_θ and the empirical distribution q.

Let's have a quick look at how these two relate. We assume we have a dataset $X = \{x^{(1)}, \ldots, x^{(N)}\}$ which we view as realizations of i.i.d. random variables distributed according to $q(x)$, the empirical distribution. We try to approximate this using a model distribution $p_\theta(x)$ for which we want to find the right parameters.

We can do this by trying to maximizing the log-likelihood of the data, or equivalently, to minimize the negative log-likelihood:

$$\hat{\theta} = \arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})).$$

When the model has a complex distribution, we often want to use gradient descent to do this optimization. In that case, instead of looking at the log-likelihood of the full dataset, we take batches of datapoints and use those for computing the gradient:

$$I \subset \{1, \ldots, n\}$$
$$\sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) \approx \frac{n}{|I|} \sum_{i \in I} -\log(p_\theta(x^{(i)})).$$

Changing the factor from $\frac{n}{|I|}$ to $\frac{1}{|I|}$ doesn't change the direction of the gradient, only its size. Since we can freely choose the learning rate of gradient descent this need not be of any consequence, and we can use

$$\frac{1}{|I|} \sum_{i \in I} -\log(p_\theta(x^{(i)}))$$

[1]Notation: we will use $q(x)$ to denote both the distribution and its density. Similarly $p_\theta(x)$ denotes both the distribution and the density.

instead, which comes down to replacing a sum by an average. But due to the law of large numbers, this expression is a good approximation of

$$\frac{1}{|I|}\sum_{i\in I} -\log(p_\theta(x^{(i)})) \approx \mathbb{E}_{q(x)}[-\log(p_\theta(x))]$$

(at least when $|I|$ is large). Put more concisely, *minimizing the negative log-likelihood of the dataset is equivalent*[2] *to minimizing the expected negative log-likelihood:*

$$\arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) \approx \arg\min_\theta \mathbb{E}_{q(x)}[-\log(p_\theta(x))].$$

This expected negative log-likelihood is called the **cross entropy** of the model distribution $p_\theta(x)$ to the true distribution $q(x)$, and is denoted by

$$H(q(x), p_\theta(x)) = \mathbb{E}_{q(x)}[-\log(p_\theta(x))].$$

We can rewrite this cross entropy as follows:

$$\begin{aligned} H(q(x), p_\theta(x)) &= H(q(x)) + D_{KL}(q(x) \,||\, p_\theta(x)) \\ H(q(x)) &= \mathbb{E}_{q(x)}[-\log(q(x))] \end{aligned}$$

where the entropy of $q(X)$, $H(q(x))$, is a constant with respect to the model parameters θ. Since $H(q(x))$ is constant, minimizing the cross-entropy is equivalent to minimizing the Kullback-Leibler divergence:

$$\begin{aligned} \arg\min_\theta \sum_{i=1}^{n} -\log(p_\theta(x^{(i)})) &\approx \arg\min_\theta H(q(x), p_\theta(x)) \\ &= \arg\min_\theta D_{KL}(q(x) \,||\, p_\theta(x)). \end{aligned}$$

In conclusion we can see maximizing the log-likelihood of the data-set as a special case of minimizing a distance between the model distribution and the empirical distribution, where we use the Kullback-Leibler divergence as our notion of distance. Algorithmically we try to solve them in the same way.

6.3 Some challenges with probabilistic models

So far we have discussed why we might want to use generative models to attempt to approximate the distribution of our dataset, and what kind of loss functionals we can use to train them. However, there are some challenges we face when trying to model data this way. These challenges come from the problem of dimensionality: it is hard and costly to represent very high dimensional probability distributions, especially when the distributions are complex. Moreover, the integrals that show up in the various loss functions we want to use become impossible to calculate exactly, and even become difficult to approximate in a deterministic way. Many calculations that can be done cheaply for low dimensional data become very expensive when the number of dimensions becomes large.

[2] Or at least they are equivalent in so far as that we would solve both minimization problems algorithmically in the same way and in that the quantities that we try to minimize are, after scaling, good approximations of each other.

Representations of distributions

The data we try to model is often very high dimensional. Images of only 100×100 pixels already are 10 000 dimensional. Suppose we have very small black-and-white pictures of 10×10 pixels which have value 1 if they are white, and 0 if they are black. If we want to specify a probability distribution over this set as an arbitrary joint distribution, that would mean we have to specify $2^{100} - 1$ parameters.

For continuous distributions we have various families of parameterized probability distributions such as the family multivariate normal distributions. These make it possible to easily specify very high dimensional distributions, but are generally unfit for modelling the complex data we try to represent.

Clearly we need better ways to represent probability distributions. We will discuss three ways of doing this here, and later in the chapter we will see these in action. For those who are not familiar with conditional probability we have added a short summary of basic probability theory in Section .1.

The first way to factorize the joint distribution using the chain rule of probability:

$$\begin{aligned} p_\theta(\mathbf{x}) &= \prod_{i=1}^{n} p_\theta(x_i \mid x_1, \ldots, x_{i-1}) \\ &= \prod_{i=1}^{n} p_\theta(x_i \mid x_{<i}). \end{aligned}$$

— think of $x_1, \ldots, x_k$ as for example the pixels of an image. As is this chain rule doesn't help much, but if we implement $p_\theta(x_i \mid x_{<i})$ using some RNN, we can learn very complex and high dimensional probability distributions using relatively few parameters. Models using this technique are called auto-regressive models. In Section 6.4 we will look at how this works in detail and see how this is used by PixelRNN to generate images.

The second way we want to discuss is to use a latent variable. We describe the rationale behind this in Section 6.5. This is the approach taken in the Variational Auto-Encoder. With these models we assume that our data $\mathbf{x}$ can best be explained using an unobserved *latent* variable $\mathbf{z}$. We model the joint distribution of $\mathbf{x}$ and $\mathbf{z}$ as

$$p_\theta(\mathbf{x}, \mathbf{z}) = p_\theta(\mathbf{z}) p_\theta(\mathbf{x} \mid \mathbf{z})$$

which makes that our data distribution is given by

$$p_\theta(\mathbf{x}) = \int_Z p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) \mathrm{d}\mathbf{z}.$$

In order to sample from $p_\theta(\mathbf{x})$ we can simply sample z from $p_\theta(\mathbf{z})$ and then sample x from $p_\theta(\mathbf{x} \mid \mathbf{z})$. Here $p_\theta(\mathbf{z})$ is can often be chosen to be a relatively simple distribution such as a multivariate standard normal distribution. We can then choose $p_\theta(\mathbf{x} \mid \mathbf{z})$ to be some parameterized family of distributions, where the parameter is determined based on $\mathbf{z}$ using a neural network. The classical example from a VAE is

$$\begin{aligned} p_\theta(\mathbf{z}) &= \mathcal{N}(0, I), \\ p_\theta(\mathbf{x} \mid \mathbf{z}) &= \mathcal{N}(f_\theta(\mathbf{z}), \lambda I), \end{aligned}$$

	Autoregressive	VAE	GAN
Advantages	- simple and stable training - good log likelihood estimation	- efficient inference with latent variables	- sharp sampling of datapoints
Drawbacks	- inefficient sampling	- blurry reconstruction	- unstable training

Figure 4: Generative models comparison

where f_θ is a neural network with weights and biases θ, and λ is a parameter mainly important for the training of the model.

This idea of drawing from a simple distribution and using a neural network to add complexity is the basis of the third approach too: using a push-forward measure as our model distribution. We take a neural network f_θ, and a simple distribution such as a multivariate normal distribution $\mathcal{N}(0, I)$.We then very easily sample from our model distribution by sampling some

$$\epsilon \sim \mathcal{N}(0, I)$$

and computing our x as

$$x = f_\theta(\epsilon).$$

This is how the model distribution is represented in the original GAN. A downside to this is that it is very hard to do density estimation using this representation of a probability distribution.

We will illustrate these techniques by discussing three algorithms that use them: PixelRNN, VAE, and GAN. Many more recent algorithms use combinations of these techniques or variations on them to model the probability distribution of a dataset. In Figure 4 we give a short comparison of Autoregressive models, VAE, and GAN.

6.4 Autoregressive models

There are a number of solutions to the challenges discussed in Section 6.3 in developing generative models for high dimensional data. In continuation we take an in-depth look of three approaches for developing generative models. The first type of generative models we will look into are the autoregressive models of which PixelRNN is a great example. Autoregressive models in deep learning are models where we specify the distribution through factorization:

$$\begin{aligned} p_\theta(\mathbf{x}) &= \prod_{i=1}^{n} p_\theta(x_i \mid x_1, \ldots, x_{i-1}) \\ &= \prod_{i=1}^{n} p_\theta(x_i \mid x_{<i}). \end{aligned}$$

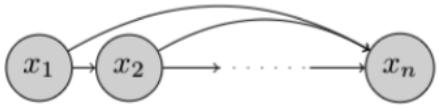


Figure 5: Autoregressive model - graphical representation

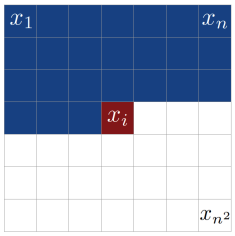


Figure 6: Sample generation with the PixelRNN method
[3]

The factorization allows us to reformulate the joint probability modeling problem into a sequence problem fig. 5. You can reflect back to the seq2seq models discussed in Chapter 5, where the output of the decoder at time t is made available for decoding at time $t + 1$.

PixelRNN models the joint distribution of the pixels in an image and allows us to sample from that distribution. In other words, the model allows us to generate images form the same distribution as the training data. PixelRNN views an image as a sequence of pixels and applies the factorization of the distribution to predict the next pixel from all previous pixels. To accomplish this it uses a variation of the LSTM inspired RNN cell.

In principle the PixelRNN model produces one pixel at the time. Starting from the first pixel the model produces a probability distribution over the values of the color intensities for that pixel. This values are discretized such that the model can model the output with a discrete distribution over the colors fig. 6. The decision for the value of the pixel at the location is then presented as input the the RNN cell such that the model can condition the probability of the next pixel value based on the decision that was made for the previous pixel value fig. 7.

Training

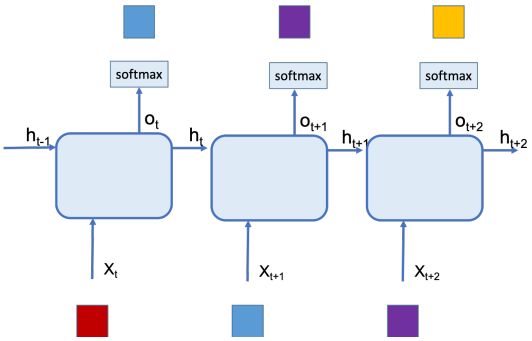


Figure 7: Sample generation PixelRNN with an RNN cell

Figure 8: PixelRNN example

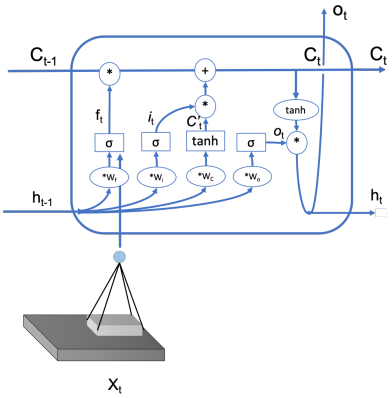


Figure 9: LSTM with convolutional filters attached to the input

To train the pixelRNN model we start by selecting an image from the training dataset. Starting from the first pixel we now use the model to generate each pixel of the image. As we did with the autoregressive seq2seq models, during training we present the ground truth to the model as a the previous output. Specifically, at time t we present to the RNN input the value x_{t-1} from training data rather than from the model's output. The loss is then compute from the output at each pixel location by computing the crossentropy between the model's output and the one-hot encoding of pixel values at each location in the training image. We apply this training process to cycling through all the images in the dataset.

A trained PixelRNN model can then be used to generate new images or part of images (fig. 8). This capability can be useful in tasks such as reconstruction of corrupted image or inference of occluded areas. The method also allows for estimating the likelihood that an image is a sample from the distribution of images given our training data. This can be useful to solving tasks such as anomaly detection.

PixelRNN - architecture

The major limitation of the PixelRNN model in its most simple form presented so far is that the conditional dependencies of the pixel values have to be propagated by the hidden state of the RNN cell. Even though different variation of the RNN cell have made improvements to the capability of this model to store longer term dependencies in general this is still a limitation of the RNN architecture. In the context of this particular task the different PixelRNN architectures that improve the performance were introduced.

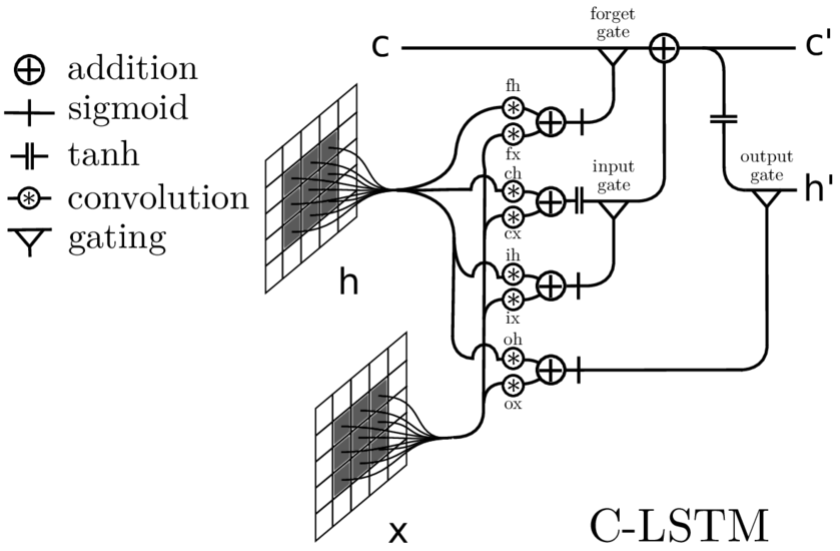


Figure 10: PixelRNN architectures. Shows convolutions over the input image and the previous hidden states.

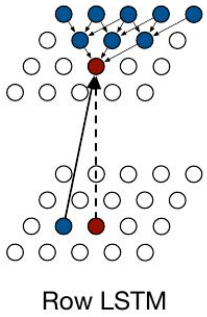


Figure 11: RowLSTM (looking at a row at the time)

One immediate improvement is to use the characteristics of the data to improve the expressiveness of the model. Specifically for image data, this means to use the localized correlations that are present. We can do that by using convolutional layers to process the image rather than processing one pixel at the time (fig. 9).

Further improvements are achieved by processing of the first layer of 2D convolutions with different strategies within the RNN cell it self. The PixelRNN approach actually extend the RNN cell to include convolutions both of the pixel values and the hidden states computed in previous steps (fig. 10. Two different strategies of processing the previous states are introduced. The RowLSTM model develops a probability distribution of a pixel at particular location based on the estimated value of neighboring pixels in the previous row (fig. 11 and the DiagonalBiLSTM model develops a conditional distribution of the pixels on an even wider area of neighbours (fig. 12). These improvements allow for more effective training of these models as the patterns present in the data are more explicit during training. The full technical details are available in the publication.

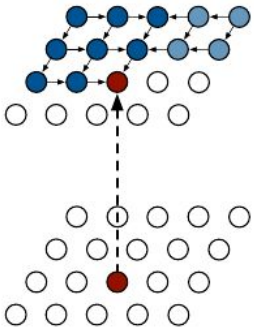


Figure 12: DiagonalBiLSTM

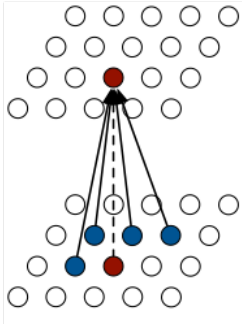


Figure 13: PixelCNN model

In this work the the we are also introduced to the PixelCNN model. This model is a fully convolutional neural networks that without a RNN cell and effectively limits the conditioning of each pixel values only to the immediate neighborhood (fig. 13). This limitation may be significant for images where a larger context may be required to form a good representation of the data. But in cases where global patterns are not present or they have little influence on value of the pixels the PixelCNN model presents a significantly more efficient solutions.

6.5 Latent Variable Models

A second strategy to develop generative models for high dimensional data is to introduce latent variables with simple distribution on which we can condition the observed variables these models are referred to as latent variable models. Imagine expressing the distribution of possible pixel values in a dataset of images. We assume that there is a simple description of the data in a lower dimensional space. In this space we can express the data with a simple distribution.

For example imaging a dataset of image of one chair (fig. 14). Each of the images in this dataset is a picture of the same chair, but at a different angle. Even though this dataset is high dimensional as the images contain many pixels, we can easily see that there is only one factors that describes the image and that is the orientation of the chair. This is a latent factor and we cannot observe it directly. The goal of the latent variable models is to develop a such descriptions of the data. Typically this is not an easy task as with most datasets we need more factors to develop an efficient representation of the data (fig. 15). Furthremore, we typically do not know the number of factors that we need and the most suited distribution to assign to them. Nevertheless the latent variable

Figure 14: A dataset of images of the same chair at a different angle

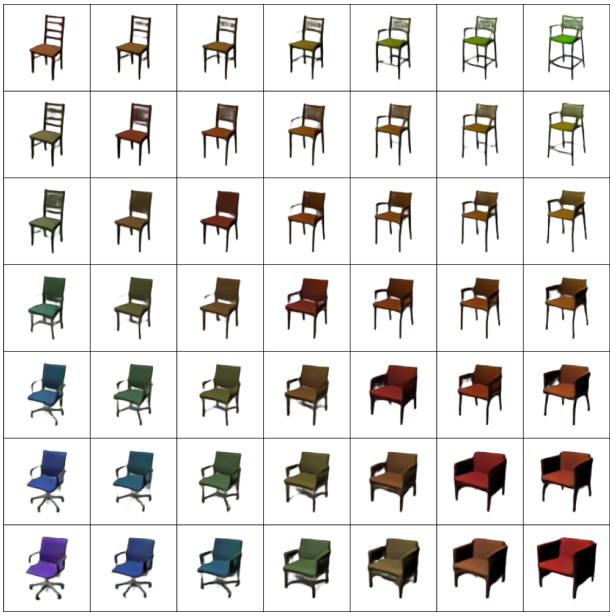


Figure 15: A dataset of images of different chairs

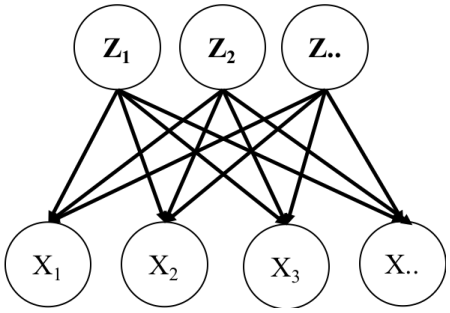


Figure 16: Graphical model representation of a Latent Variable Model

model give us the flexibility that eventually allows us develop generative models.

As a graphical representation the latent variable model can be depicted as in (fig. 16).

The joint distribution of the observed variables is

$$P(X) = \sum_{Z} P(X|Z)P(Z)$$

The Variational Auto-Encoder (VAE) is an of latent variable model that is a generative model for high dimensional data.

6.6 Variational Auto-Encoder

The VAE uses a latent variable z to describe the data x. That means it describes a distribution

$$p_\theta(x, z) = p_\theta(x \mid z)p_\theta(z),$$

which in turn gives a distribution over the data of the form

$$\begin{aligned} p_\theta(x) &= \int p_\theta(x, z)\mathrm{d}z \\ &= \int p_\theta(x \mid x)p_\theta(z)\mathrm{d}z. \end{aligned}$$

This integral however is itself intractable. We want the distribution we learn to match the true[4] distribution, which we will denote by $q(x)$, over the data as closely as possible. We can go at this in two ways, which as we will see, result in the same loss function. In both cases we introduce an extra conditional probability distribution $q_\phi(z \mid x)$ that we want to approximate $p_\theta(z \mid x)$ because the latter is intractable:

$$q_\phi(z \mid x) \approx p_\theta(z \mid x) = \frac{p_\theta(x \mid z)p_\theta(z)}{p_\theta(x)}.$$

We will write $q_\phi(x, z)$ for $q_\phi(z \mid x)q(x)$ which is, as a whole, a parameterized joint distribution on x and z.

[4]Also called the empirical distribution.

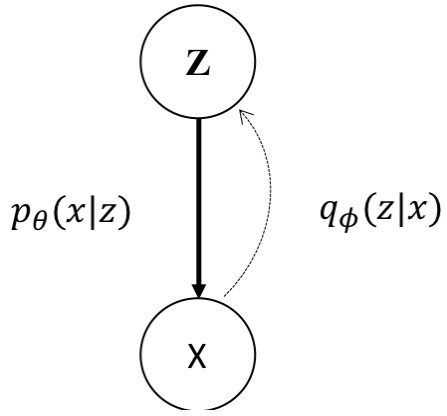


Figure 17: The graphical model for the Variational Auto-Encoder. The solid arrow represents the decoding distribution $p_\theta(x \mid z)$ and the dashed arrow represents the encoding distribution $q_\phi(z \mid x)$.

The first perspective is that we want to maximize the (log-)likelihood of the observed data according to our model distribution, i.e.

$$\arg\max_\theta \mathbb{E}_{q(x)} \left[\log p_\theta(x)\right].$$

In order to do this, we rewrite the expected log-likelihood as

$$\begin{aligned}
\mathbb{E}_{q(x)} \log p_\theta(x) &= \int_x 1 \cdot \log(p_\theta(x))\, q(x) \mathrm{d}x \\
&= \int_X \int_Z q_\phi(z \mid x) \mathrm{d}z \log(p_\theta(x))\, q(x) \mathrm{d}x \\
&= \int_{X \times Z} \log(p_\theta(x))\, q_\phi(x, z) \mathrm{d}x\, \mathrm{d}z \\
&= \int_{X \times Z} \log\left(\frac{p_\theta(x \mid z) p_\theta(z) q_\phi(z \mid x)}{p_\theta(z \mid x) q_\phi(z \mid x)}\right) q_\phi(x, z) \mathrm{d}x\, \mathrm{d}z \\
&= \int_{X \times Z} \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z \mid x)}\right) q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x - \int_{X \times Z} \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x \\
&\quad + \int_{X \times Z} \log(p_\theta(x \mid z))\, q_\phi(z \mid x) \mathrm{d}z\, q(x) \mathrm{d}x \\
&= \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z \mid x)\right) - \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z)\right) \\
&\quad + \mathbb{E}_{q(x)} \mathbb{E}_{q_\phi(z|x)} \log p_\theta(x \mid z).
\end{aligned}$$

Seeing as the Kullback-Leibler divergence in the intractable first term is always non-negative, we can bound the average log-likelihood from below by

$$\begin{aligned}
\mathbb{E}_{q(x)} \log p_\theta(x) \geq & - \mathbb{E}_{q(x)} D_{KL}\left(q_\phi(z \mid x) \,||\, p_\theta(z)\right) \\
& + \mathbb{E}_{q(x)} \mathbb{E}_{q_\phi(z|x)} \log p_\theta(x \mid z).
\end{aligned}$$

We can then try to maximize this "variational lower bound", also called the ELBO, in hopes of maximizing the log-likelihood. This is shown in Figure 18.

Another perspective is that q_ϕ and p_θ are both models for the same data with different properties: p_θ allows us to easily generate data but $p_\theta(x)$ is just an approximation to the real distribution and

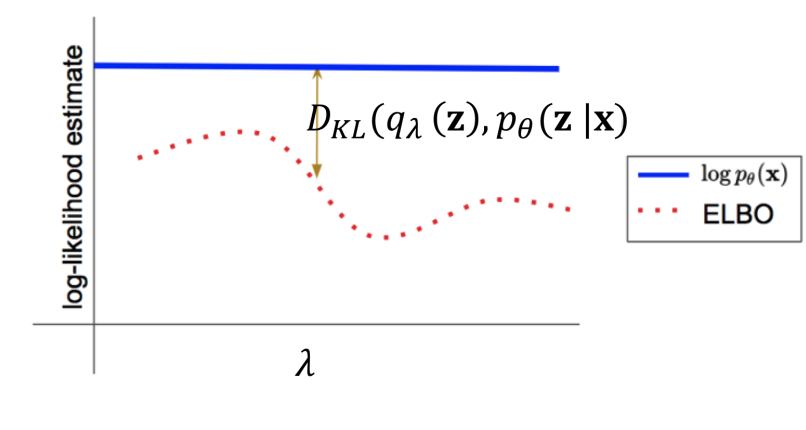


Figure 18: We try to maximize the log-likelihood by maximizing the ELBO. Source: Grover and Ermon (2018) DGM tutorial.

we have no easy way of knowing what latent variable corresponds to what data point; on the other hand, $q(x)$ is the correct distribution over the data, and $q_\phi(z \mid x)$ tells us what latent variable corresponds to a data point, but we have no easy way of generating data using this description of the distribution.

Now suppose that these models agree, i.e. $p_\theta(x, z) = q_\phi(x, z)$. In that case we have a correct description of the data distribution that allows us to generate new data points and that allows us to relate the latent variable to the data in both directions.

To achieve this we measure the difference between the two distributions using the Kullback-Leibler divergence and jointly train θ and ϕ using gradient descent and backpropagation to minimize this divergence:

$$\arg\min_{\theta,\phi} D_{KL}(q_\phi(x, z) \,||\, p_\theta(x, z)).$$

We rewrite this divergence as

$$\begin{aligned}
D_{KL}(q_\phi(x, z) \,||\, p_\theta(x, z)) &= \int_{X\times Z} \log\left(\frac{q_\phi(x, z)}{p_\theta(x, z)}\right) q_\phi(x, z)\mathrm{d}x\,\mathrm{d}z \\
&= \int_{X\times Z} \log\left(\frac{q(x)q_\phi(z \mid x)}{p_\theta(z)p_\theta(x \mid z)}\right) q(x)q_\phi(z \mid x)\mathrm{d}x\,\mathrm{d}z \\
&= \int_X \log(q(x))q(x)\mathrm{d}x \qquad (4) \\
&\quad + \int_X \int_Z \log\left(\frac{q_\phi(z \mid x)}{p_\theta(z)}\right) q_\phi(z \mid x)\mathrm{d}z\, q(x)\mathrm{d}x \qquad (5) \\
&\quad - \int_X \int_Z \log(p_\theta(x \mid z))q_\phi(z \mid x)\mathrm{d}z\, q(x)\mathrm{d}x. \qquad (6)
\end{aligned}$$

Here the first term, (4), is the negative (differential) entropy of the true distribution. Since this term contains no parameters, its gradients in training are 0 and we can ignore it for our minimization

problem. This leaves us with (5) and (6) making our loss function

$$L_{VAE} = \mathbb{E}_{q(x)}\left[D_{KL}(q_\phi(z \mid x) \mid\mid p_\theta(z))\right] \tag{7}$$

$$- \mathbb{E}_{q(x)}\left[\mathbb{E}_{q_\phi(z|x)}\big(\log p_\theta(x \mid z)\big)\right], \tag{8}$$

which is the earlier variational lower bound with its sign swapped — i.e. both perspectives lead to the same optimization objective.

As in Section 6.2, we can approximate the expectations with respect to $q(x)$ by using a random batch of data points, say $x^{(1)}, \ldots, x^{(m)}$. We approximate as

$$L_{VAE} = \frac{1}{m}\sum_{i=1}^{m} D_{KL}(q_\phi(z \mid x = x^{(i)}) \mid\mid p_\theta(z)) \tag{9}$$

$$- \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}_{q_\phi(z|x=x^{(i)})}\big(\log p_\theta(x = x^{(i)} \mid z)\big), \tag{10}$$

We then still have two integrals to either compute or approximate: the expectation in (10) and the Kullback-Leibler divergence in (9). In some cases, most notably when both $q_\phi(z \mid x)$ and $p_\theta(z)$ are multivariate normal distributions, we can use an exact formula for the Kullback-Leibler divergence. Otherwise we can approximate it in the same way as we approximate the expectation in (10): by sampling from $q_\phi(z \mid x = x^{(i)})$.

In general, if we have some probability distribution $\rho(y)$ and an integrable function f, we can approximate

$$\mathbb{E}_{\rho(y)}[f(y)] \tag{11}$$

by sampling a number of independent drawings $y_1, \ldots, y_l \sim \rho(y)$, and computing

$$\mathbb{E}_{\rho(y)}[f(y)] \approx \frac{1}{l}\sum_{j=1}^{l} f(y_j).$$

Due to the law of large numbers, if l is large enough this will likely be a good approximation. This way of approximating integrals is a basic example of a *Monte Carlo method*.

Both the Kullback-Leibler divergence from (9) and the expected value in (10) are integrals like (11) where $\rho = q_\phi(z \mid x = x^{(i)})$. So if for very $x^{(i)}$, we can draw $z_1^{(i)}, \ldots, z_l^{(i)} \sim q_\phi(z \mid x = x^{(i)})$ independently, then we can approximate (9) and (10) as

$$(9) = \frac{1}{m \cdot l}\sum_{i=1}^{m}\sum_{j=1}^{l} \log(q_\phi(z = z_j^{(i)} \mid x = x^{(i)})) - \log(p_\theta(z = z_j^{(i)}))$$

$$(10) = \frac{1}{m \cdot l}\sum_{i=1}^{m}\sum_{j=1}^{l} \log(p_\theta(x = x^{(i)} \mid z = z_j^{(i)})).$$

In practice, l is usually chosen as $l = 1$.

One difficulty we have to address is that to be able to train p_θ and q_ϕ jointly using backpropagation and gradient-descent, we need both of our approximations[5]to be differentiable with respect to ϕ.

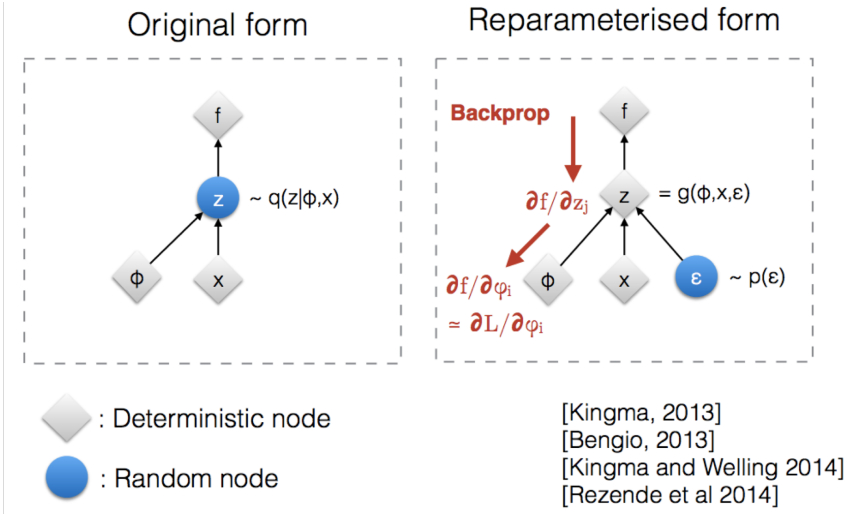


Figure 19: The reparameterization trick

More specifically, we want our samples $z_j^{(i)}$ to be differentiable with respect to ϕ. In order to do this, we can use the reparameterization trick.

The *reparameterization trick* consists of drawing a stochastic variable ϵ from some fixed distribution, and applying a differentiable[6] function g_ϕ such that

$$g_\phi(\epsilon, x^{(i)}) \sim q_\phi(z \mid x = x^{(i)}).$$

For example, if $q_\phi(z \mid x = x^{(i)}) = \mathcal{N}(\mu, \sigma^2)$, then if $\epsilon \sim (0, 1)$, we have $\mu + \sigma\epsilon \sim q_\phi(z \mid x = x^{(i)})$. Using this trick we can draw $z_j^{(i)}$ as

$$z_j^{(i)} = g_\phi(\epsilon_j^{(i)}, x^{(i)})$$

where $\epsilon_j^{(i)}$ are independent samples from the same fixed distribution. The original paper on the VAE mentions three basic approaches to finding a transformation g_ϕ:

1. For any location-scale family of distributions, such as Gaussian distributions, we can sample ϵ from the distribution with location=0 and scale = 1, and set $g_\phi(\epsilon, x) = \text{location}_\phi(x) + \text{scale}_\phi(x)\epsilon$.

2. If the distribution has a tractable inverse cumulative distribution function, we can let $\epsilon \sim U(0, \mathbf{1})$ (where $\mathbf{1} = (1, \ldots, 1)$), and let g_ϕ be the inverse CDF.

3. Many distributions can be drawn from by drawing from a different distribution and applying a standard transformation, e.g. a log-normal distribution can be drawn from by drawing from a normal distribution and exponentiating the result.

[5]Really it's not just that we want the approximations to be differentiable with respect to ϕ, but also that we want the resulting derivative is a good approximation of the derivative of the terms we are approximating.

[6]With respect to ϕ

Figure 20: The MNIST dataset

Let us look at a more concrete example. Suppose we want to use a VAE with d-dimensional latent space to develop a generative model for the MNIST dataset in Figure 20. We can choose the following distributions:

$$\begin{aligned} q_\phi(z \mid x) &= \mathcal{N}\left(\mu_\phi(x), \mathrm{diag}(\sigma_\phi^2(x))\right), \\ p_\theta(z) &= \mathcal{N}(0, I_{\mathbb{R}^d}), \\ p_\theta(x \mid z) &= \mathcal{N}\left(m_\theta(z), s\, I\right), \end{aligned}$$

where μ_ϕ and σ_ϕ are neural networks[7]with parameters ϕ, and m_θ is a neural network with parameters θ, and where s is some fixed number. To train this we can take a batch of data points $x^{(1)}, \ldots, x^{(N)}$ and for each data point compute $\mu_\phi(x^{(i)})$ and $\sigma_\phi(x^{(i)})$. Next we can, for each data point, draw a random vector $\epsilon^{(i)} \sim \mathcal{N}(0, I)$ to use in the parameterization trick, as shown in Figure 21. The resulting value for the latent variables then becomes

$$z^{(i)} = \mu_\phi(x^{(i)}) + \sigma_\phi(x^{(i)}) \cdot \epsilon^{(i)},$$

where the multiplication is element-wise. We can compute the first loss term, (7), using

$$\begin{aligned} D_{KL}(q_\phi(z \mid x = x^{(i)}) \,||\, p_\theta(z)) &= D_{KL}\left(\mathcal{N}(\mu_\phi(x^{(i)}), \mathrm{diag}(\sigma_\phi^2(x^{(i)}))) \,||\, \mathcal{N}(0, I)\right) \\ &= \frac{1}{2}\sum_{j=1}^{d}\left(\sigma_\phi^2(x^{(i)})_j + \mu_\phi(x^{(i)})_j - \log(\sigma_\phi^2(x^{(i)})_j) - 1\right), \end{aligned}$$

and approximate the second, (8), using a one sample Monte-Carlo approximation and using

$$-\log p_\theta(x = x^{(i)} \mid z^{(i)}) = C + \frac{1}{2s}||x^{(i)} - \mu_\theta(z^{(i)})||^2$$

[7]For σ_ϕ we typically use an exponential function as the final activation to ensure we get a positive number.

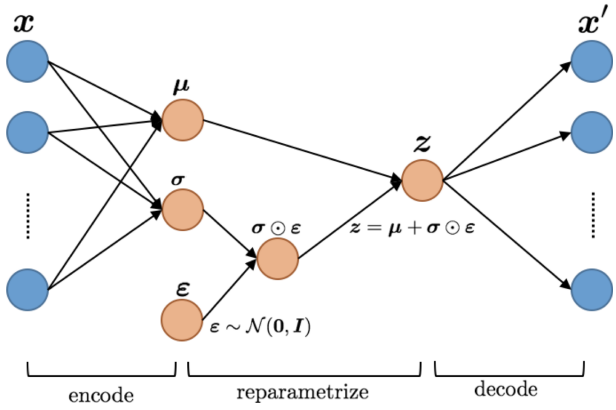


Figure 21: Reparameterization trick with Gaussian distributions with diagonal covariance.

where C is a constant (and thus of no influence on the training). This loss term is usually referred to as the *Reconstruction loss*. We could also have used a different probability distribution. For example, if we had modelled $p(x \mid z)$ as a bunch of independent Bernoulli random variables with success probabilities based on z, we would have gotten binary cross-entropy as the reconstruction loss.

Note that if we ignore the first loss term, (7), and make $q_\phi(z \mid x)$ deterministic (e.g. setting $\sigma_\phi = 0$), this model is simply an auto-encoder as discussed in Chapter 3. Let us look at the full model from this perspective too. If we have a regular auto-encoder, it is generally very hard to, without encoding real data, come up with codes that correspond to realistic data. The way in which data and code relate to each other can often be very irregular, and only small portions of the space of possible codes correspond to realistic data. By making the encoder stochastic with a distribution that covers a portion of the latent space with relatively high probability, we ensure that all points in the latent space that are close to $\mu_\phi(x)$ for some x will correspond to realistic data.

This however is not yet enough to ensure that we can use the model to generate new, realistic looking data. The encoder might simply learn to make $\sigma_\phi(x)$ very small, and to send $\mu_\phi(x)$ for different x very far apart (compared to $\sigma_\phi(x)$). The Kullback-Leibler divergence loss on the latent space prevents this, forcing the encoder to cover the latent space as similar to a normal distribution as it can without seriously sacrificing the quality of reconstruction. This makes that we can use the VAE to generate new data points, and even interpolate between two data points, as shown in Figure 22.

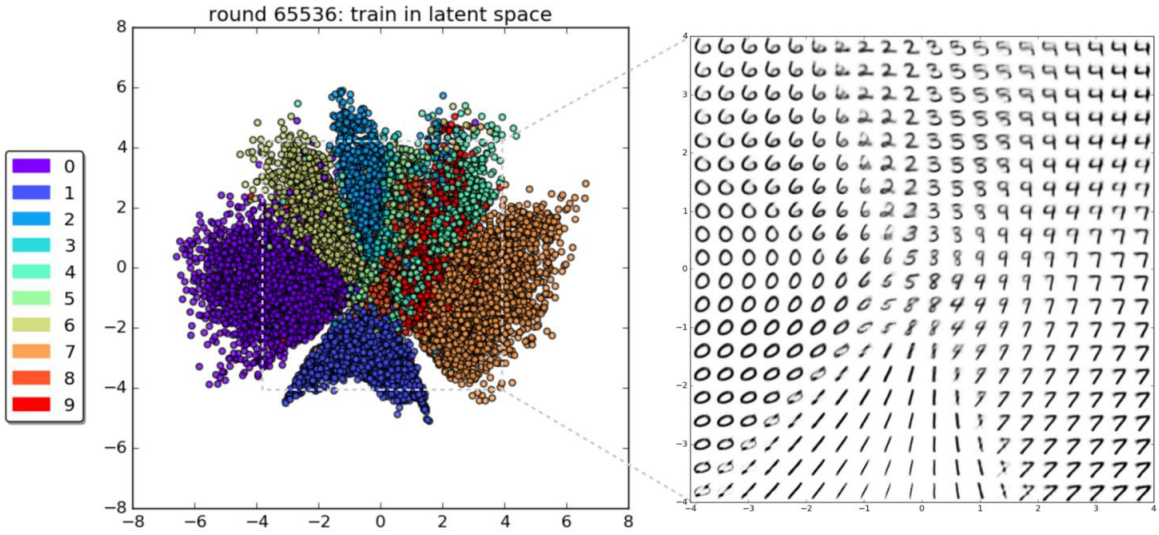


Figure 22: The 2-dimensional latent space of a VAE trained on MNIST

6.7 Generative Adversarial Networks

The main idea behind the Generative Adversarial Networks (GAN) is that the performance metric against which we train the generative model should be learned itself so as to prevent over-fitting to a specific loss function, and because no simple loss function really accurately captures how humans sees e.g. whether two pictures look similar. This is done by having a generative model G and a discriminative model D which we train jointly to try to achieve[8]

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim P(x)}[\log(D(x))] + \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))]. \tag{12}$$

In words, we try to train a discriminator D to be able to decide whether its input comes from the data set (maximizing the first term) or is generated (maximizing the second term). In the meanwhile we try to train a generator G to fool the discriminator (minimizing the second term by picking the best G). In terms of loss functions this comes down to

$$\begin{aligned}\text{Loss}_D &= -\mathbb{E}_{x\sim P(x)}[\log(D(x))] - \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))],\\ \text{Loss}_G &= -\text{Loss}_D\end{aligned}$$

or equivalently for gradient descent

$$\text{Loss}_{G,\,equivalent} = \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))].$$

The generative model G learns the distribution of the data in an implicit way. We sample from it by sampling a noise variable z from some noise distribution $p(z)$ which can be e.g. a uniform distribution, or a Gaussian distribution, and by applying a function G to z:

$$\begin{aligned}z &\sim p(z)\\ x &= G(z).\end{aligned}$$

Here G is a neural network, and we optimize (12) by training the parameters of the network. The discriminator D is a neural network too, and again, optimizing is done by training its parameters.

We can train the model by doing the following every training step:

1. Take a batch of data points $x^{(i)}$ and sample a batch of noise variables $z^{(i)}$ and use them to compute the discriminator loss and gradients
2. Update the discriminator using those gradients
3. (Optionally) repeat step 1 and 2 a number of times
4. Take a batch of noise variables and use them to compute the generator loss
5. Update the generator using those gradients

[8] This is the learning objective for the original GAN. Since then a number of variations on this have been developed, some of which we will discuss in short.

This adversarial training is often compared to real world examples such as criminals innovating to avoid getting caught by police, and police innovating to become better at catching criminals. Think of the generator as some art forger trying to create pictures such that they pass for creations by some great artist, and think of the discriminator as some expert trying to distinguish the forgeries from the masterpieces.

The objective in eq. (12) is very useful for theoretical analysis of the GAN, but has a major drawback: whenever the discriminator manages to distinguish well between real data and generated examples, the gradient of the loss function for the generator is rather flat, making it hard for the generator to learn. To overcome this a heuristic loss is often used for the generator instead. This heuristic loss is given by

$$\text{Loss}_{G,\,heuristic} = -\mathbb{E}_z \log D(G(z)). \tag{13}$$

Instead of trying to minimize the probability of the discriminator being right, this loss encourages the generator to maximize the probability of the discriminator being wrong.

Training a GAN can be difficult for several reasons. Training can often be unstable where the discriminator and the generator simply keep undoing each others progress by focusing on irrelevant details, or where the generator is unable to learn at all due to the discriminator being too good. The latter problem can be reduced by using the heuristic loss for the generator given in eq. (13). Another trick for stabilizing training is *one-sided label smoothing*. This means that instead of

$$\begin{aligned}\text{Loss}_D &= \sum_{x^{(i)}} -\log(D(x^{(i)})) + \sum_{z^{(i)}} -\log(1 - D(G(z^{(i)}))) \\ &= \sum_{x^{(i)}} H_{bin}(1, D(x^{(i)})) + \sum_{z^{(i)}} H_{bin}(0, D(G(z^{(i)})))\end{aligned}$$

we replace the labels for the true samples by slightly lower values:

$$\text{Loss}_{D,\,smooth} = \sum_{x^{(i)}} H_{bin}(1 - \epsilon, D(x^{(i)})) + \sum_{z^{(i)}} H_{bin}(0, D(G(z^{(i)})))$$

where H_{bin} denotes binary cross-entropy

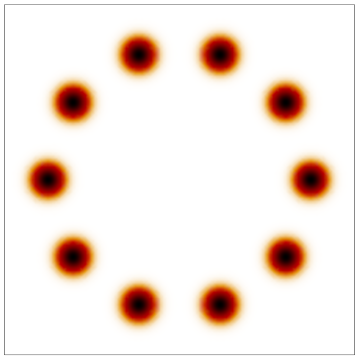
$$H_{bin}(y, x) = y \log(x) + (1 - y) \log(1 - x),$$

and $0 < \epsilon \ll 1$ can either be fixed or random. This is done to prevent overly confident classification through interpolation by the discriminator.

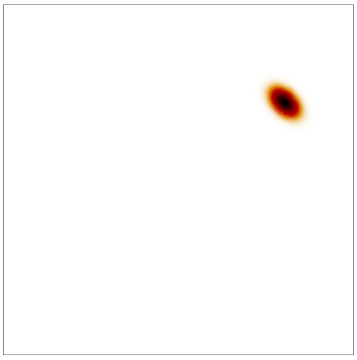
Another recurring problem with the training of GANs is that of mode collapse. This is where the true distribution as a number of "modes" or most likely outcomes and the generator only produces examples corresponding to one of the modes. Think e.g. of images from different classes where the different modes correspond to the different classes, the generator might only create examples from one class, or of a few of the classes, ignoring the others. This is illustrated in Figure 23.

In theory optimizing (12) is equivalent to minimizing the Jensen-Shannon divergence between the learned distribution and the real distribution.[9] By modifying (12) or putting some extra restrictions on the discriminator, the corresponding divergence can be changed to various other notions of

[9] Due to the need to represent the functions by neural networks and having to learn parameters instead of functions directly, this equivalence doesn't really fully hold. It is still useful for theoretical analysis of GANs.



(a) The target distribution that we want to learn has ten modes.



(b) The GAN only learns one of the modes.

Figure 23: Mode collapse in a GAN model

distance. In attempts to address the problems with training GANs various such modifications have been proposed, including WGAN (Wasserstein-1 or Earth-Mover metric) and f-GAN (any f-divergence including Jensen-Shannon and Kullback-Leibler, the latter of which theoretically would allow a GAN to be used for likelihood maximization).

Finally, if labels are present for (part of) the data set, they can be used in various ways to significantly improve the quality of the generated samples and to stabilize the training process and prevent mode collapse.

Figure 24: An example from `thispersondoesnotexist.com`, generated by StyleGAN2. A link to the corresponding article can be found on that website.

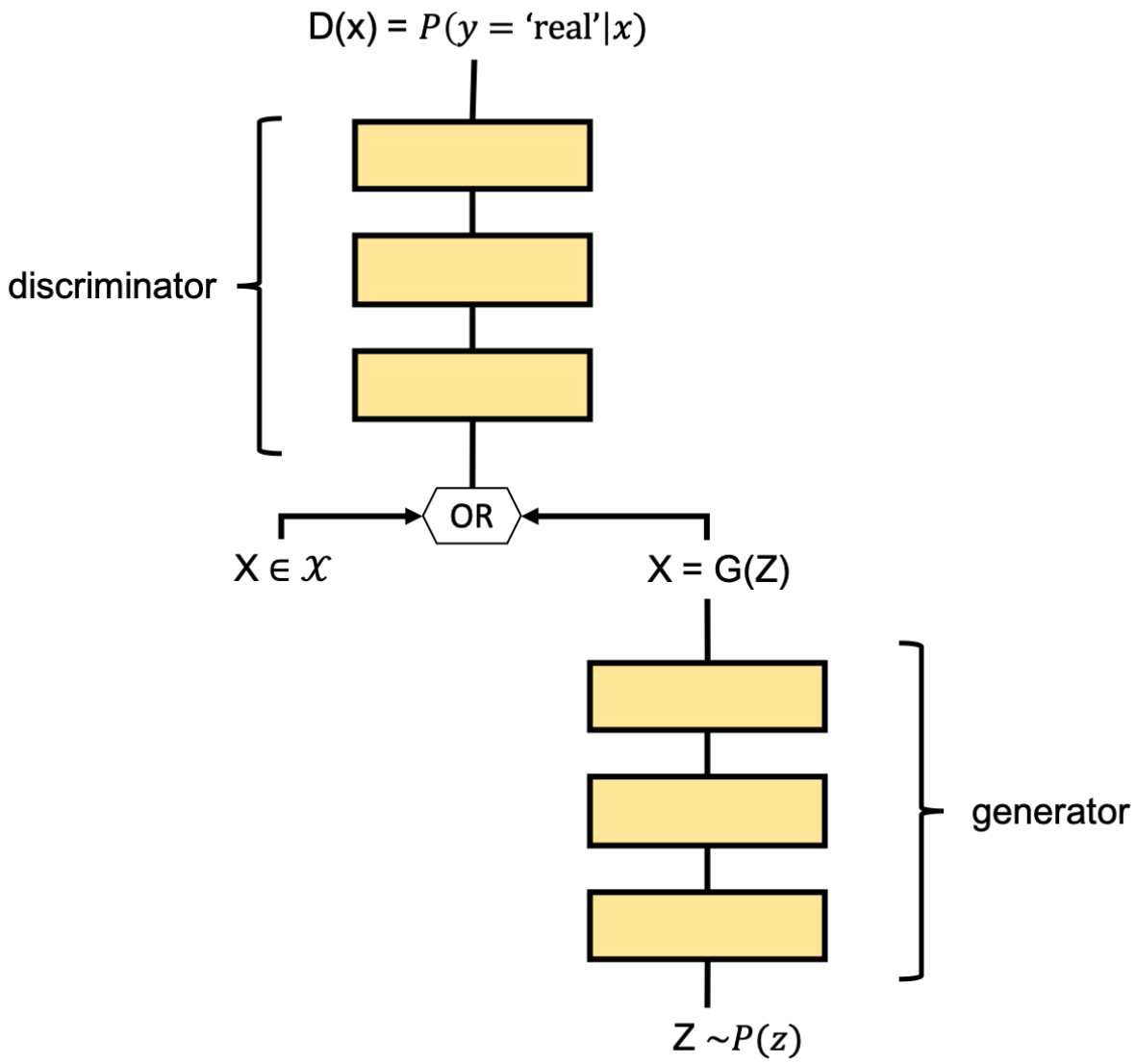


Figure 25: A visual representation of a Generative Adversarial Network

Appendices

.1 Short recap of probability theory

Throughout this chapter we will be working with a number of concepts from probability theory. For those who haven't worked with probability theory for a long time, we give a short description of these concepts. Moreover, we will talk a bit about the notation we use throughout this chapter. You do not need to know all of the technical details behind these concepts to understand the material in this chapter, so do not worry if you don't understand some of these technicalities.

Probability distributions and random variables

The most commonly used framework for probability theory is built upon the concept of a probability space and a probability measure. The details of these are often not needed for practicing probability theory, but a very basic understanding can help you understand how things relate to each other.

In this framework we have a set of possible outcomes called the **sample space**, often denoted by Ω. Think of this set as follows: if we run an experiment where we throw seven fair six-sided dice in a row, this space would consist of all 6-tuples of possible outcomes:

$$\Omega = \{\omega = (\omega_1, \ldots, \omega_7) \mid \omega_1, \ldots, \omega_7 \in \{1, \ldots, 6\}\}$$

On top of this, we have a set of **events**, often denoted by $\mathcal{F} \subset 2^\Omega$, which is a set of subsets of Ω. One event in our experiment with six dice might be that the second die was a three. The corresponding element of $\mathcal{F}$ would be

$$\{\omega \in \Omega \mid \omega_2 = 3\} \in \mathcal{F}.$$

The final ingredient of a probability space is the **probability measure**. This probability measure tells us how likely an event is, and is a function from $\mathcal{F}$ to the interval $[0, 1]$, in this case

$$P : \mathcal{F} \to [0, 1]$$
$$P : A \mapsto \frac{\#A}{6^7},$$

where $\#A$ denotes the number of elements in A. E.g. for the event $A = \{\omega \in \Omega \mid \omega_2 = 3\}$ we have 6^6 different outcomes that belong to A, so the probability of A is $P(A) = 6^6/6^7 = 1/6$, which is precisely what we expect intuitively.

The sample space, Ω, doesn't have to be finite, or even countable. Say for example its raining and we have a square of 1×1 meter on the ground. If we look at just the next rain drop, we could say that our sample space is just a square

$$\Omega = [0, 1] \times [0, 1]$$

and our events are regions of the square[10]where the rain drop might fall

$$\mathcal{F} \subset 2^\Omega,$$

with the probability of an event being the area of that region

$$P(A) = \text{area}(A).$$

This model of the situation is manageable if we want to model just one rain drop, but if we want to do more — e.g. multiple rain drops and the times in between them — it easily becomes impractical. In practice, the **probability space** $(\Omega, \mathcal{F}, P)$ is kept abstract[11], and all calculations are done using *random variables*.

A **random variable**, X is a measurable function

$$X : \Omega \to \mathbb{R},$$

or more generally

$$\mathbf{X} : \Omega \to \mathbb{R}^n.$$

Such a random variable comes with its own events of the form[12]

$$\{X \in A\} = X^{-1}(A) = \{\omega \in \Omega \mid X(\omega) \in A\}$$

and with its own probability measure on its range:

$$P_X = P_{\#}X \ : \ A \mapsto P(X^{-1}(A)).$$

This probability measure is called the **distribution**, or the **law** of X.

In this course all our random variables are either discrete, where the range of X is some countable discrete set — e.g. $\{1, \ldots, 10\}$, $\mathbb{N}$, or $\mathbb{Z}$ — or continuous, where the range of X is some interval of $\mathbb{R}$ (or some Cartesian product thereof).

In the case of a discrete random variable, we can characterize the whole random variable by the probability of it taking specific values, i.e. we can describe X using the function

$$f_X : R \to [0, 1] : r \mapsto P(X = r).$$

This function, f_X is called the **probability mass function (PMF)** of X.

For a distribution to be continuous, there is an additional requirement besides what the range of X is: the distribution must have a **probability density function**[13] **(PDF)**, $f_X \geq 0$, so that

$$P(X \in A) = \int_A f_X(x) \mathrm{d}x.$$

Throughout this course all PDFs will be assumed to be continuous and strictly positive.

When the range of X is some ordered set such as $\mathbb{R}$, $\mathbb{N}$, $\mathbb{Z}$, or $[0, 1]$, we also have a **cumulative distribution function (CDF)** given by

$$F_X : x \mapsto P(X \leq x).$$

For continuous random variables, the PDF is the derivative of the CDF.

[10] Whether $\mathcal{F}$ can be all of 2^Ω in this case depends on your axioms of set theory. For the most commonly used axioms the answer is *no*. This however is beyond the scope of this course.

[11] Such an abstract probability space needs to satisfy some conditions: $\mathcal{F}$ needs to contain the full sample space, it needs to be closed under taking complements, and under taking countable unions of its elements. The probability measure P must be a function on $\mathcal{F}$ such that $P(\Omega)$=1, and such that for any *countable* collection of *disjoint* elements of $\mathcal{F}$, $(A_i)_{i=1}^\infty$, we have $P(\bigcup_{i=1}^\infty A_i) = \sum_{i=1}^\infty P(A_i)$.

[12] Notation like $X \in A$ or $X \leq x$ is often used to denote the corresponding events $\{\omega \in \Omega \mid X(\omega) \in A\}$ and $\{\omega \in \Omega \mid X(\omega) \leq x\}$. This way direct references to Ω are seldom needed.

[13] With respect to the Lebesgue measure. You can have densities with respect to other measures too, e.g. a PMF is a density with respect to the counting measure supported on the range of the random variable, but that's besides the point.

Conditional probability, joint distributions, and independence.

When we have multiple random variables, $X_1, \ldots, X_n$ we can look at them together. Suppose two people, person 1 and person 2, are living together. Say that on a given day the probability of one of them having the flue is around 2%, i.e. $P(X_i = 1) = .02$, where $X_i = 1$ means person i has the flue, and $X_i = 0$ means the student doesn't have it. Because the students have a lot of contact with each other, if one student has the flue, others are likely to get it as well, so the probability that student 1 has the flue given the fact that student 2 has it, is much larger than the a-priori probability that student 1 has it without extra information. This leads us to conditional probability and to joint distributions for random variables.

If we have two events, A and B, the probability of A **conditioned** on B is given by

$$P(A \mid B) = \frac{P(A \cap B)}{B}.$$

In the example above, the contagiousness of the flue makes that

$$P(X_1 = 1 \mid X_2 = 1) > P(X_1 = 1).$$

In general $P(A \mid B)$ can be larger than, less than, or equal to $P(A)$. The case $P(A \mid B) = P(A)$ is special: if

$$P(A \mid B) = A,$$

or equivalently[14]

$$P(A \cap B) = P(A) \cdot P(B),$$

we say the events A and B are **independent**, denoted by $A \perp B$. Two random variables, X and Y, are called independent, denoted by $X \perp Y$ if all their associated events are independent mutually independent, i.e.

$$X \perp Y \iff \forall_{A,B \text{ measurable}} \ : \ X^{-1}(A) \perp Y^{-1}(B).$$

An important theorem for dealing with conditional probabilities is **Bayes' theorem**, which says that for any two events, A and B, we have

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}.$$

Especially when two or more random variables are not independent, it is interesting to look at their joint distribution. For simplicity we will look at continuous random variables taking values in $\mathbb{R}$, but all of this can easily be generalized to other cases. We can view a collection of random variables $X_1, \ldots, X_n$ as function

$$\mathbf{X} \ : \ \Omega \to \mathbb{R}^n \ : \ \omega \mapsto (X_1(\omega), \ldots, X_n(\omega)).$$

[14]Here and in the definition of conditional probability we assume that $P(B) > 0$. Conditioning on events of probability 0 is tricky, and we will only do so when the events come from continuous random variables with continuous and nowhere vanishing densities.

This **random vector** again has associated events of the form

$$\{\mathbf{X} \in A\} = \mathbf{X}^{-1}(A) = \bigcap_{i=1}^{n} X_i^{-1}(A),$$

and an associated probability measure on $\mathbb{R}$ given by

$$P_{\mathbf{X}} = \mathbf{X}_{\#}P \ : \ A \mapsto P(\mathbf{X}^{-1}(A)).$$

This measure is called the joint distribution of the random variables. Throughout this chapter, all our joint distributions will be assumed to be continuous (at least if needed) so that this joint distribution again has a **joint density** $f_{\mathbf{X}} : \mathbb{R}^n \to [0, 1]$ such that

$$P(\mathbf{X} \in A) = \int_A f_{\mathbf{X}}(x_1, \ldots, x_n) \mathrm{d}x_1 \cdots \mathrm{d}x_n.$$

From the joint density we obtain the density of a single random variable through the process of **marginalization**, for example if we have two random variables X and Y with joint density $f_{X,Y}$, then:

$$f_X(x) = \int_{\mathbb{R}} f_{X,Y}(x, y) \mathrm{d}y.$$

From the joint density and marginal densities we can get the **conditional distribution**. If we have two random variables X and Y with joint density $f_{X,Y}$, then the conditional distribution is given by

$$f_{Y|X=x}(y) = f_{Y|X}(y \mid x) = \frac{f_{X,Y}(x, y)}{f_X(x)}.$$

Note that this means we can **factorize** the density of the joint distribution as

$$\begin{aligned} f_{X,Y}(x, y) &= f_X(x) f_{Y|X}(y \mid x) \\ &= f_Y(y) f_{X|Y}(x \mid y). \end{aligned}$$

Expectation

When we are dealing with random phenomena, we often want to make predictions, or comparisons. Although we usually can't make claims about the precise outcomes, we can often say something about the expected outcome, or the average outcome if we have a large number of independent and identically distributed random variables. For example, when throwing fair six sided dice, we don't know in advance what the outcome will be, but we do expect that if we throw a lot of these dice, the average outcome will be around 3.5. This idea that we can expect a certain average outcome is formalized by the **expected value**, and the idea that we will obtain this value if we do a lot of independent experiments is formalized by the **Law of Large Numbers**. The expected value of a discrete random variable X is simply a weighted average of the possible outcomes where the weights are the probabilities of the outcomes — i.e. if the set of possible outcomes is S, and X has probability mass function f_X

$$\mathbb{E}[X] = \sum_{s \in S} s \cdot f_X(s).$$

We generalize this to continuous random variables by replacing the sum by an integral, and the mass function by a density:

$$\mathbb{E}[X] = \int_S s \cdot f_X(s) \mathrm{d}s,$$

where again S is the range of the random variable. Whenever this integral, or in case of discrete random variables the sum, is not well-defined, we say that the random variable does not have an expected value.

Sometimes we want to know what the expected value of some function of a random variable instead of the random variable itself. For this we have two useful results: the first is the **law of the unconscious statistician** which tells us that for a random variable X and a function g, we have

$$\mathbb{E}[g(X)] = \sum_{s \in S} g(s) \cdot f_X(s) \qquad \text{if } X \text{ is discrete,}$$

and

$$\mathbb{E}[g(X)] = \int_S g(s) \cdot f_X(s) \mathrm{d}s \qquad \text{if } X \text{ is continuous,}$$

provided that the expressions are well-defined.

The second useful result is **Jensen's inequality**, which tells us that if a function g is **convex** we have

$$g(\mathbb{E}[X]) \leq \mathbb{E}[g(X)],$$

and consequently if g is **concave** we get

$$g(\mathbb{E}[X]) \geq \mathbb{E}[g(X)].$$

Whenever there might be confusion against what random variable, or what probability distribution, we are taking the expected value, we will indicate the distribution in subscript.