Deep Learning Lecture Notes

Vlado Menkovski, Simon Koop, Koen Minartz

May 2023

Chapter 1

Introduction

1.1 Learning outcomes

The learning outcomes of this chapter are:

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

1.2 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 automation 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 of requiring learning from examples (see Figure 1.1)?

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

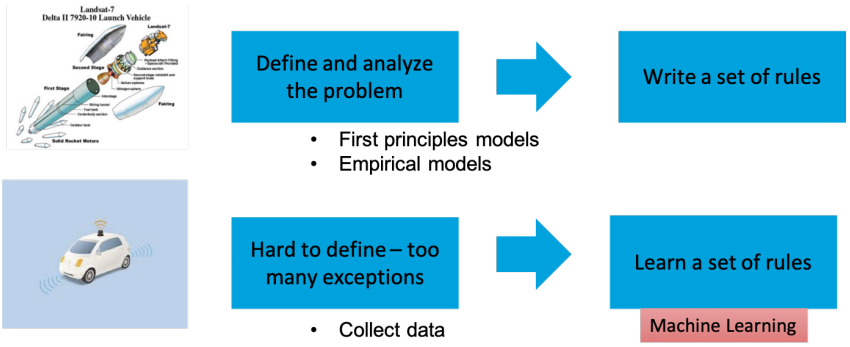


Figure 1.1: Traditional modeling versus machine learning.

It basically boils down to our inability to express the goals of these tasks with rules sufficiently precisely. This can either stem from our lack of understanding or from the underlying complexities in the system. In many 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 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, by observing the system we may be able to make predictions. For example, we may learn that when a certain gene in the cell is expressed, a particular behavior can be expected, and the statistics of the pixel value of an image may be indicative of whether we are looking at airplanes or zebras.

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

Question: What are the challenges self-driving cars and translation pose that motivate the use of ML for them?

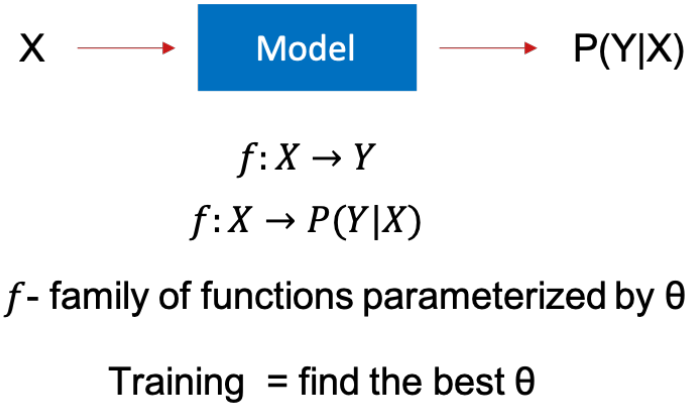


Figure 1.2: Formulation of a machine learning model as a parameterized function.

1.2.1 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 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 taking 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.

1.2.2 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 the 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.

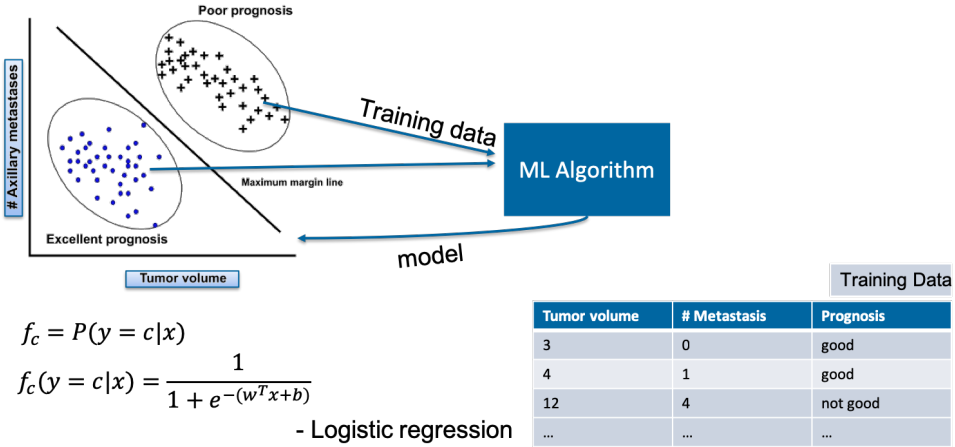


Figure 1.3: Dataset for the prognosis of cancer patients. Blue circles indicate an 'Excellent' prognosis and black plus-signs indicate a 'poor' prognosis.

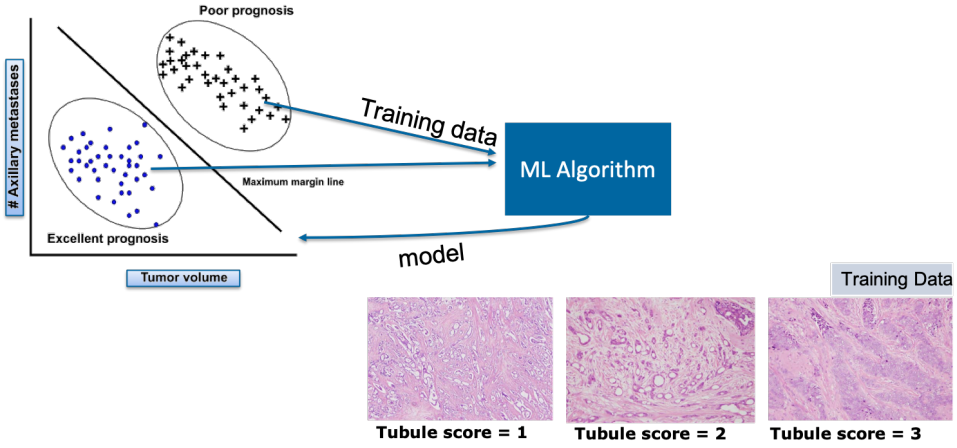


Figure 1.4: The data sets belonging to the two different examples.

Motivation - ML - Example

Let's look at a fictional dataset of cancer patients. Figure 1.3 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 data point is labeled with a blue circle for 'Excellent' and a black plus sign for 'Poor'. This dataset has a strong correlation between the two features and the target variable. Specifically, the data points 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?

1.2.3 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

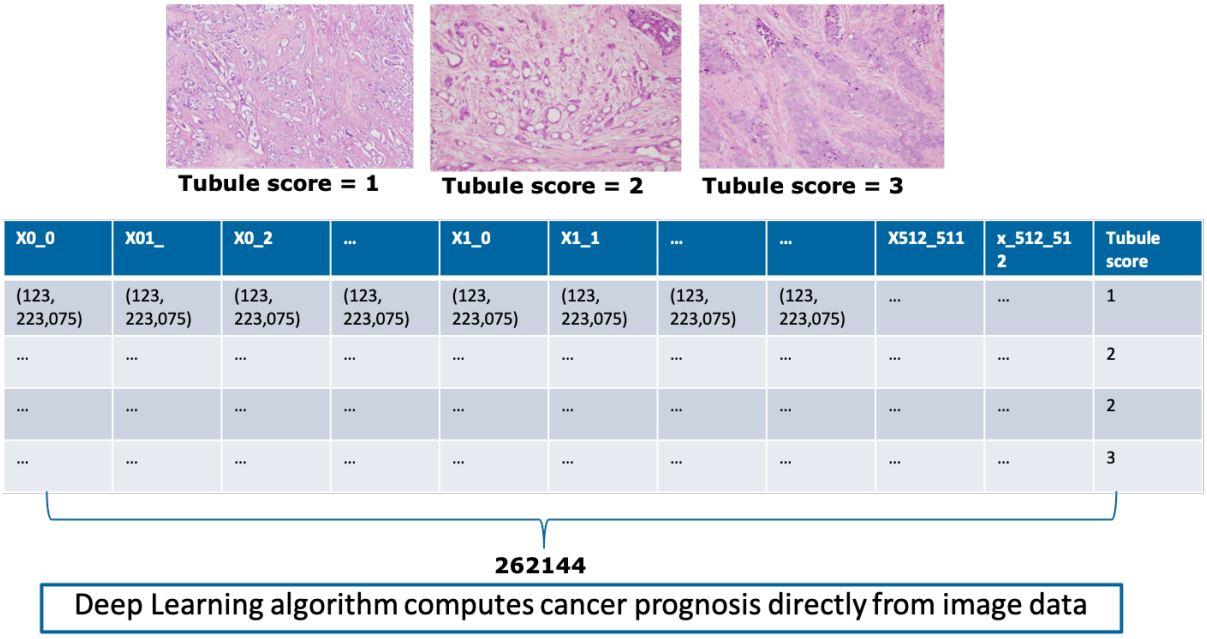


Figure 1.5: The input to our model consists of the values for all pixels. That makes 262144 three-dimensional variables.

the breast tissue and our aim is to determine the prognosis of breast cancer. In the bottom row of Figure 1.4 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 1.5, 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?

1.3 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, 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 an 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

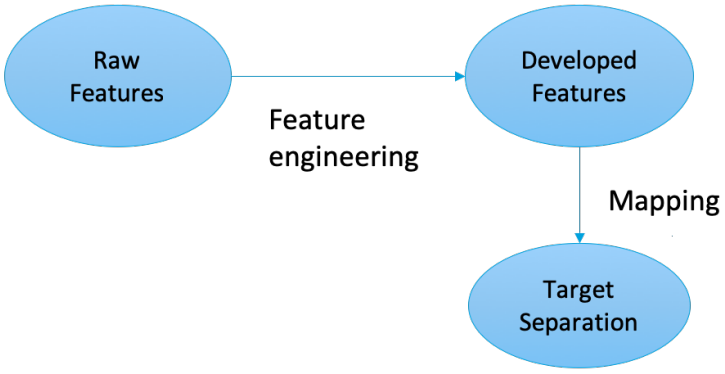


Figure 1.6: Feature engineering

now we are designing rules to develop features. The next step is to use an ML algorithm to develop a model using those features (fig. 1.6, fig. 1.7). 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.

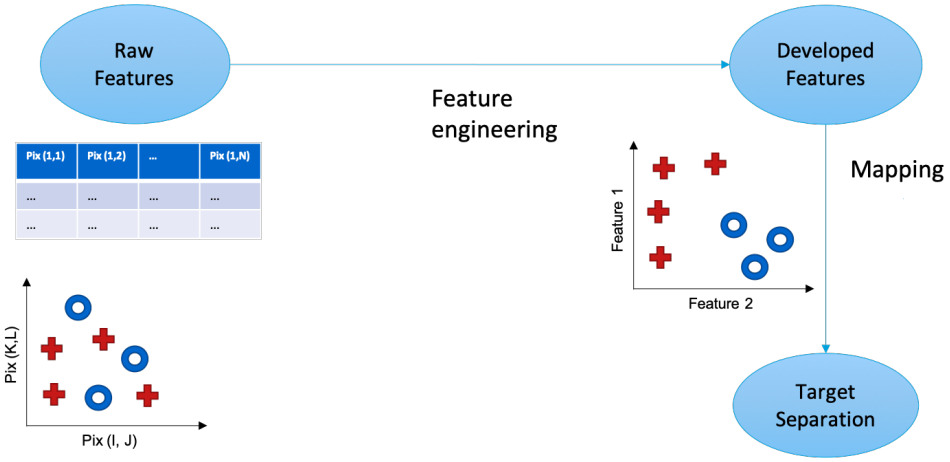


Figure 1.7: A schematic overview of feature engineering.

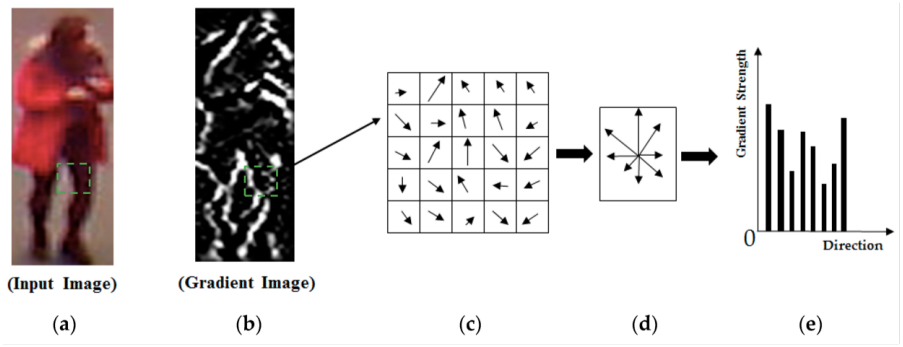
Question: What could be a feature engineering target in our tubule score example? Do you see any patterns in the images for which we can develop feature engineering algorithms?

One problem with feature engineering is that in many cases the 'low-level' features are very high dimensional[1] and domain knowledge is either not available or difficult to express in hard-coded rules.[2]

One example of feature engineering using in general-purpose image analysis such as the histogram of oriented gradients (HOG) fig. 1.8. This technique counts the occurrences of gradient orientations in small patches of the image. This technique is very useful for distinguishing objects from the background and is helpful in describing some objects but falls short of capturing the high-level concepts that we aim for in analyzing images for various downstream tasks.

[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.

[2]For example, for facial recognition, the shape of someone's nose might be a useful feature, but it is hard to define in terms of pixel values of a photo.



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 1.8: An illustration of a Histogram of Gradients

Therefore, it is usually the case that we need to engineer custom features for specific tasks.

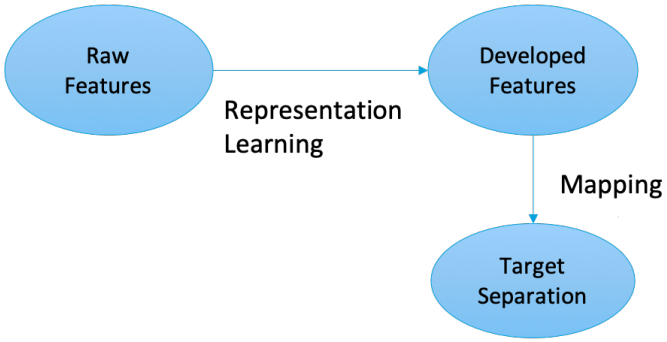


Figure 1.9: A schematic overview of representation learning

1.4 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, 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 levels 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. 1.9). Much of this course revolves around representation learning, specifically learning representations of high-dimensional data for various tasks.

The setting 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 consist of a sequence of 'layers' that apply non-linear transformations to their inputs. This structure allows for learning hierarchical features with increasing levels of complexity.

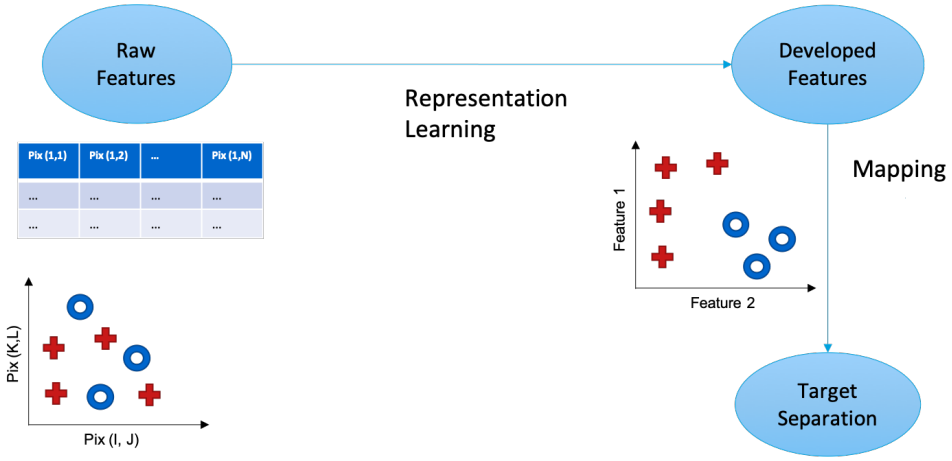


Figure 1.10: Representation learning end-to-end

This way, the neural network model composes features of features of features so that sufficiently informative high-level features can then have a simple (even linear) map to the target variable (fig. 1.10).

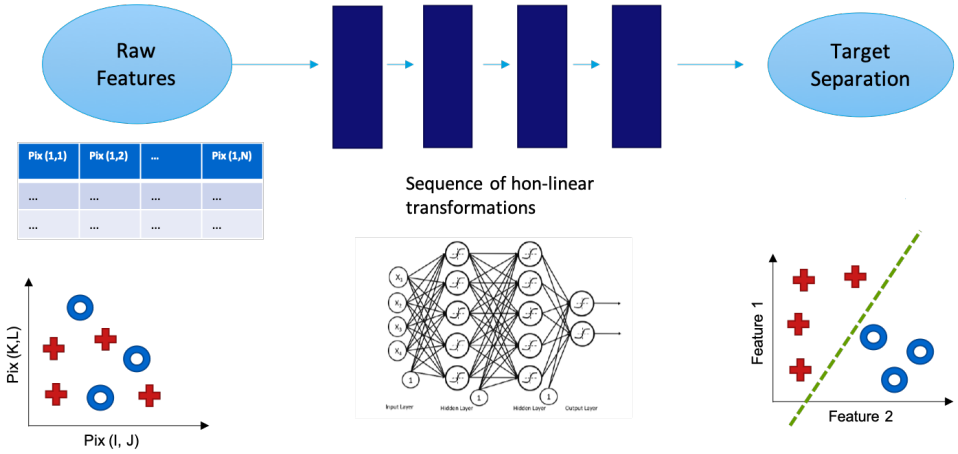


Figure 1.11: Representation learning with a neural network

This type of representation learning can also be referred to as *end-to-end representation learning* as the loss function[3] from the end of the model is used to learn the features at all levels back to the beginning (fig. 1.11). 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*.

[3]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.

1.5 Course overview

In this course we study the following topics:

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

In addition to these topics, which are techniques for building models, in the course, we also cover solutions for downstream tasks using these techniques (**??**[4]). These include:

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

The dependencies of the course topics are shown in **??**. The course is organized into six content chapters:

3. Introduction to Neural Networks
4. Word embedding
5. Models for spatially distributed data
6. Models for sequential data
7. Graph Neural Networks
8. Generative models

[4] blue boxes

Chapter 2

An introduction to Neural Networks

The learning outcomes of this chapter:

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

The main theme of this course is developing representations of high dimensional data. In Chapter 1, we discussed the motivation and utility of having such representations; in this chapter, we will go over the underlying methods for developing such representations.

2.1 The Artificial Neuron

The main component of the artificial neural network is the artificial neuron. You can see a graphical representation of such an artificial neuron in Figure 2.1. To understand what an artificial neuron is, let us look at an example of a neuron taking five 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 computes its output $o_\theta(x)$ as:

$$o_\theta(x) = \phi(\sum_i w_i x_i + b) = \phi(\mathbf{w}^T\mathbf{x} + b).^1 \quad (2.1)$$

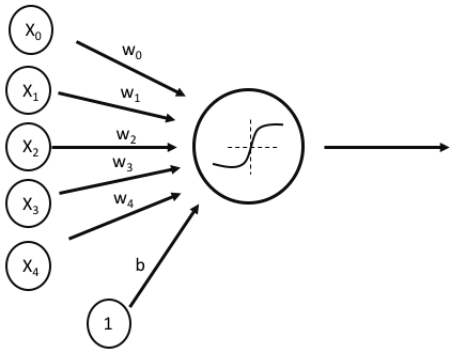


Figure 2.1: Artificial neuron

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

$$
\begin{aligned}
o_\theta(\mathbf{x}) &= \phi(\mathbf{w}^\top \mathbf{x} + b) && (2.2)\\
&= \phi(1 \cdot 1 + 2 \cdot 0.5 + 3 \cdot 1 + 0.5) && (2.3)\\
&= \phi(1 + 1 + 3 + 0.5) && (2.4)\\
&= \phi(5.5) && (2.5)\\
& && (2.6)\\
& && (2.7)
\end{aligned}
$$

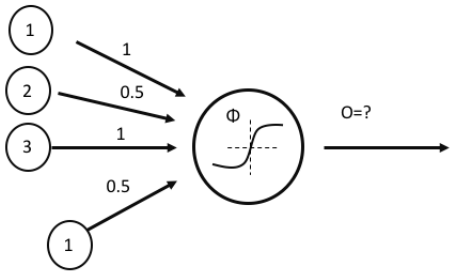


Figure 2.2: Artificial neuron activations

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, $\mathbf{x} = (1, 2, 3)$, the artificial neuron will produce the value 5.5 — see Figure 2.2 for a schematic overview.

2.1.1 Linear Regression

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

$$o_\theta(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b \tag{2.8}$$

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 2.3.

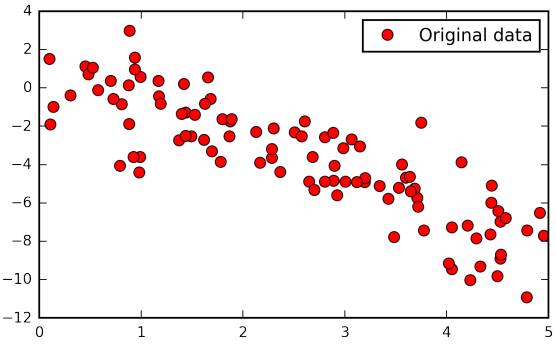


Figure 2.3: A dataset for which we might want to use a linear regression model.

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 our pre-

[1] In order to make the 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.

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

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

$$\frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right). \tag{2.9}$$

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

$$\theta_{\text{best}} = \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(y^{(i)}, \hat{y}^{(i)}\right) \tag{2.10}$$

$$= \arg\min_{\theta} \frac{1}{N}\sum_{i=0}^{N-1} L\left(o_\theta(\mathbf{x}^{(i)}), y^{(i)}\right). \tag{2.11}$$

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

$$L(y, \hat{y}) = (y - \hat{y})^2. \tag{2.12}$$

2.2 Gradient Descent & Backpropagation

The minimization in eq. (2.11) 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 hillside and want to get to a (local[3]) minimum. What we could do, is see in what direction the hill goes down most steeply, and take a step in that direction. After that step, we again look at in what direction the hill is steepest 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

$$\begin{aligned} w &\leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \\ b &\leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \end{aligned} \tag{2.13}$$

where α is a parameter referred to as the learning rate

2.2.1 Gradient Descent Optimization

Gradient descent (GD) optimization is an iterative optimization algorithm for finding minima of differentiable functions. The algorithm updates the parameters of the model in an iterative fashion until it converges, i.e. until the loss value does not decrease anymore. The update values of the parameters of our linear regression model are given by eq. (2.13). So, to get the new values of the parameters we subtract from the old values a value proportional to the gradient of the loss with respect to the 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 on 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 model increases, we will use tools such as Pytorch 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.

2.2.2 Computing the gradient of the loss

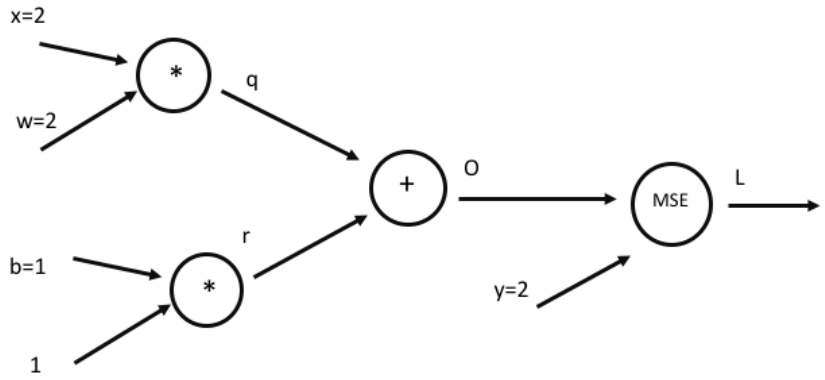


Figure 2.4: *Computation graph* - For a single data point $\{x = 2, y = 2\}$

Let us compute the gradient for our artificial neuron given a single data point. 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 2.4 shows the computation graph for our artificial neuron from Section 2.1.1 for a single data point and one-dimensional input.

Computing the gradient of the loss — The forward pass

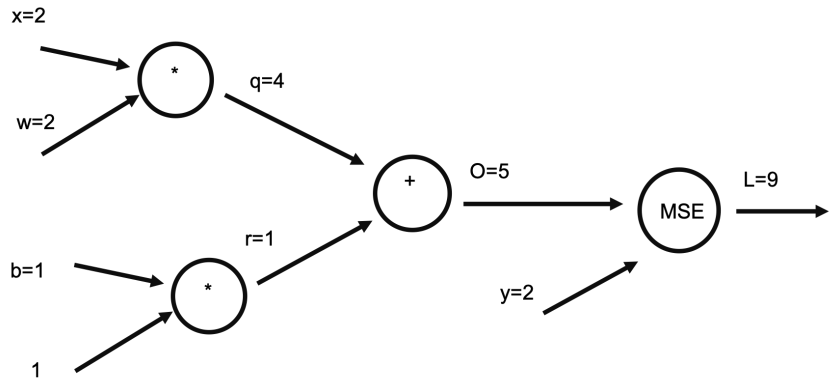


Figure 2.5: *Computation graph* — The forward pass

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 2.5. The computations are as follows:

$$o = wx + b = 2 \cdot 2 + 1 = 5 \tag{2.14}$$

$$L = \frac{1}{1}(o - y)^2 = (5 - 2)^2 = 3^2 = 9 \tag{2.15}$$

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 backward — see Figure 2.6.

$$\frac{\partial L}{\partial L} = 1 \tag{2.16}$$

$$\frac{\partial L}{\partial o} = 2(o - y) \cdot 1 = 2(o - y) \tag{2.17}$$

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial q} = 2(o - y) \tag{2.18}$$

$$\frac{\partial L}{\partial r} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial r} = 2(o - y) \tag{2.19}$$

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial w} = 2(o - y) \cdot x \tag{2.20}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial r}\frac{\partial r}{\partial b} = 2(o - y) \cdot 1 \tag{2.21}$$

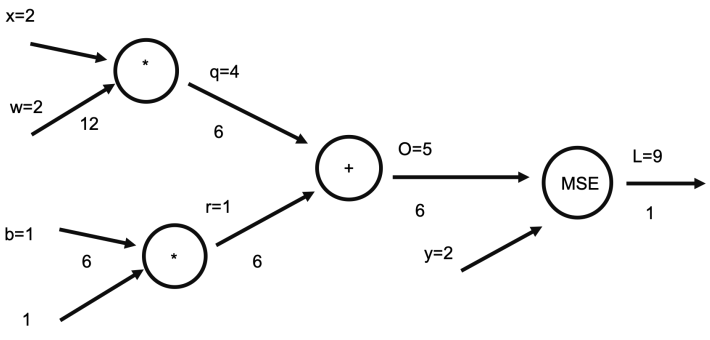


Figure 2.6: *Computation graph* — The backward pass

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

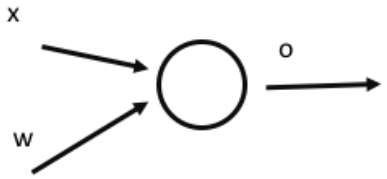


Figure 2.7: A node in a computation graph to perform Backpropagation on.

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 take a step in the gradient descent algorithm, i.e. to update the parameters. With the gradients and values from Figure 2.6 and learning rate $\alpha = 0.1$ this is done as:

$$w \leftarrow w - \alpha \frac{\partial}{\partial w} L(\mathbf{x}, y; \mathbf{w}, b) \tag{2.22}$$

$$= 2 - 0.1 \cdot 12, \tag{2.23}$$

$$b \leftarrow b - \alpha \frac{\partial}{\partial b} L(\mathbf{x}, y; \mathbf{w}, b) \tag{2.24}$$

$$= 1 - 0.1 \cdot 6. \tag{2.25}$$

2.3 Classification

In Section 2.1.1 we looked at a model doing linear regression to predict continuous values and in Section 2.2 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.

$$f_c(\mathbf{x}) = P(y = c|\mathbf{x}) \tag{2.26}$$

An example of binary classification

- Two input variables: x_0, x_1
- Two classes

$$c1 = \text{class 1} \tag{2.27}$$
$$c2 = \text{class 2} \tag{2.28}$$

- Output is

$$f_{c1}(\mathbf{x}) = P(y = c1|\mathbf{x}) \tag{2.29}$$
$$f_{c2}(\mathbf{x}) = P(y = c2|\mathbf{x}) \tag{2.30}$$

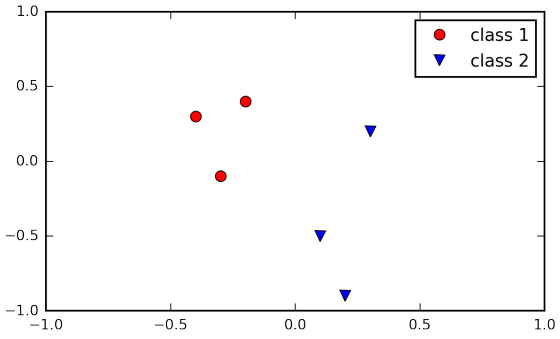


Figure 2.8: An example of a data set for which we might want to do classification.

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 the case of binary classification, the activation typically used for this is the "logistic sigmoid" function, or "sigmoid" for short. In the 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 data point can belong to multiple classes;
- takes values between 0 and 1;
- its graph is shown in Figure 2.21e.

[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 2.5.2.

Softmax function:

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

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

- its components are positive and sum to 1.

2.3.1 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 variables — see for example the data set in Figure 2.8 — 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 2.9. As a whole the artificial neuron implements the following:

$$o_\theta(x) = \text{sigmoid}(w_0 x_0 + w_1 x_1 + b) \tag{2.31}$$
$$= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \tag{2.32}$$
$$= \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}. \tag{2.33}$$

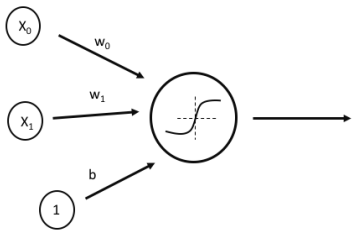


Figure 2.9: A schematic overview of our artificial neuron used for binary classification.

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

$$P(y = 1 \mid \mathbf{x}) = o_1 \tag{2.34}$$
$$= o_\theta(\mathbf{x}) \tag{2.35}$$
$$= \text{sigmoid}(\mathbf{w}^\top \mathbf{x} + b) \tag{2.36}$$
$$P(y = 0 \mid \mathbf{x}) = o_0 \tag{2.37}$$
$$= 1 - o_1. \tag{2.38}$$

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

$$P(y) = o_\theta(\mathbf{x})^y (1 - o_\theta(\mathbf{x}))^{(1-y)}, \tag{2.39}$$

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

so that the loss becomes

$$L(\mathbf{x}, y; \theta) = -(y \log(o_\theta(\mathbf{x})) + (1 - y) \log(1 - o_\theta(\mathbf{x}))) \tag{2.40}$$
$$= -\log(o_y). \tag{2.41}$$

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 backpropagation on a computation graph to get the gradients. The computation graph for this model is shown in Figure 2.10. 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* $y = 1$.

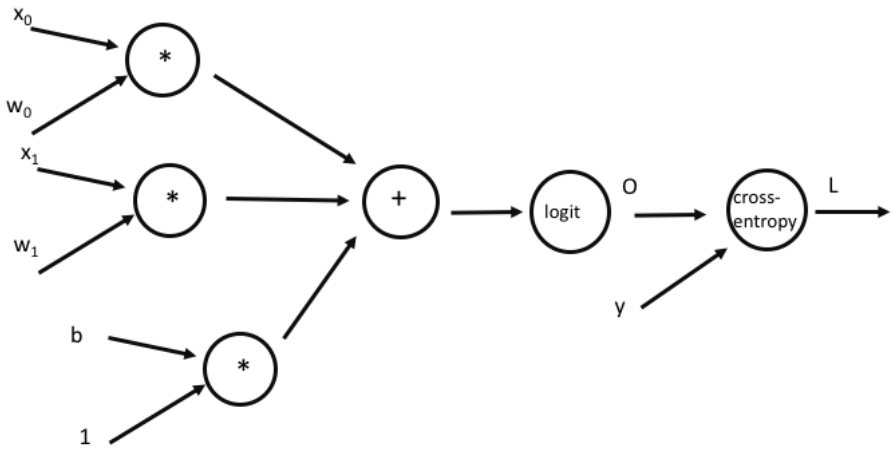


Figure 2.10: The computation graph for our single neuron binary classifier.

2.4 Multilayer Perceptron

2.4.1 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 data points that are associated with different labels.

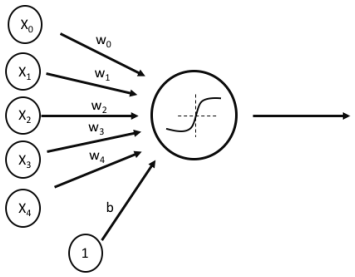


Figure 2.11: A single artificial neuron

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.

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 2.12.

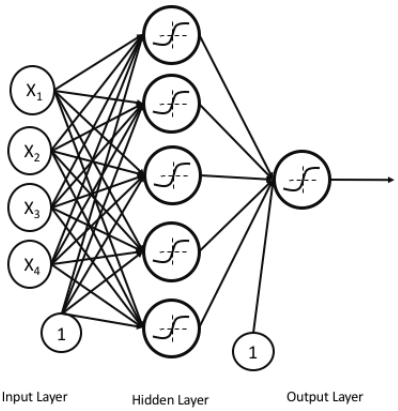


Figure 2.12: A Multi Layer Perceptron with a single hidden layer.

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 final layer that produces the output is the *output layer*.

[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.

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 the hidden layers 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 2.13.

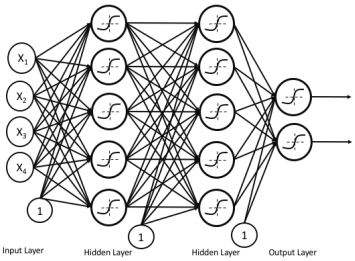
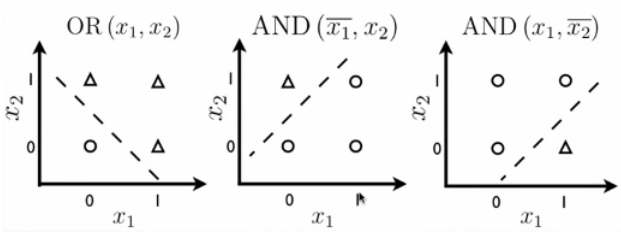


Figure 2.13: An MLP with multiple hidden layers

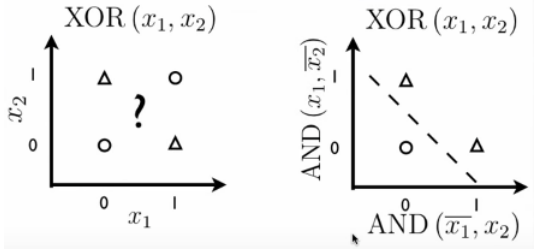
The advantage of having multiple layers is that a complex boundary between two assignments (decisions) about the input can be decomposed as a set of simpler boundaries that are then combined in the next layer. For deeper models, this sequence of compositions can allow for developing highly complex maps 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 composite features scale significantly better, as for computing the higher-level features, the lower-level features are re-used. The term features here refers to output values of neurons. During training, neurons specialize in detecting specific patterns of their input and can be referred to as feature detectors.

A simple example of where linear separability is not sufficient is data generated by the binary xor operation (figs. 2.14a and 2.14b).



(a) Dataset of binary operations that is linearly separable



(b) Dataset of XOR binary operation that is not linearly separable

Figure 2.14: Two datasets of binary operations. One is linearly separable, the other is not.

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 connections between neurons within a layer
  - and all neurons in the same layer are of the same type (have the same activation function).

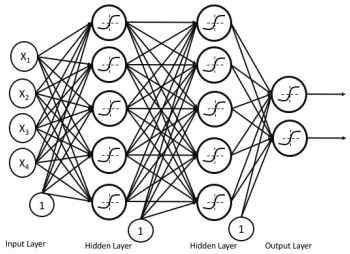


Figure 2.15: The structure of an MLP

- The set of inputs to the MLP is called the input layer.
- The final layer is called the output layer.
- It is also referred to as a feedforward Neural Network (meaning that there are no loops and information flows from the input directly to the output)

We can also think of MLPs in terms of composition of functions. Each layer creates a new representation of the input data:

$$h^{(0)} = f^{(0)}(x; \theta_0) \tag{2.42}$$
$$h^{(1)} = f^{(1)}(h^{(0)}; \theta_1) \tag{2.43}$$
$$y = f^{(2)}(h^{(1)}; \theta_2). \tag{2.44}$$
$$\tag{2.45}$$

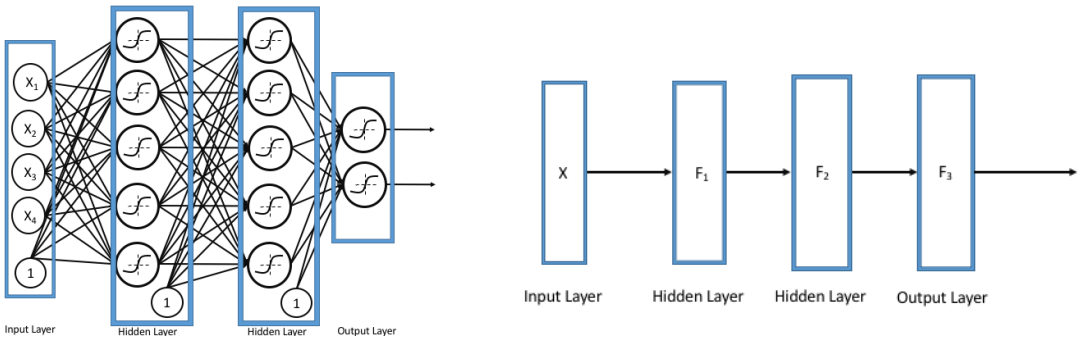
The MLP as a whole is then a parameterized, composed function:

$$f(x; \theta) = f^{(2)}(f^{(1)}(f^{(0)}(x; \theta_0); \theta_1); \theta_2), \tag{2.46}$$

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.

What we need in order to use an MLP model is:

- a model:
  - $o_\theta = \phi_3(\mathbf{w_3}^\top \phi_1(\mathbf{w_2}^\top \phi_1(\mathbf{w_1}^\top x)))$ (biases are omitted for brevity)
  - $\theta : \{\mathbf{W}\}$
- a loss function:
  - e.g. $L(\mathbf{x}, y; \mathbf{W}) = \frac{1}{n} \sum_{i=0}^{n} (o_\theta - y)^2$
- the gradient of L with respect to $\mathbf{W}$:
  - $\frac{\partial}{\partial W} L(.)$



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

(b) The final consolidated representation of an MLP.

Figure 2.16: Looking at an MLP as a graph of layers provides a more abstract view of the model.

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 are represented as a single edge. This is shown in Figure 2.16. The advantage this gives us is that it helps us to more easily reason about complex models with many layers. Moreover, as we will 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 2.17. Now the edges carry vectors, and the nodes 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 2.2.2. The only difference is that now our derivative is not a vector, but a matrix: the Jacobian matrix.

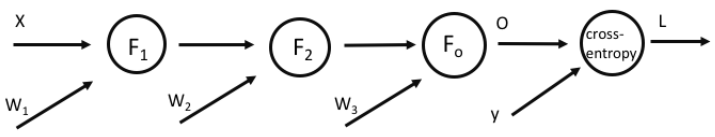


Figure 2.17: The computation graph in terms of layers.

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"

$$J_g(\mathbf{x}) = \frac{\partial g}{\partial \mathbf{x}}(\mathbf{x}) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_1}{\partial x_n}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1}(\mathbf{x}) & \cdots & \frac{\partial g_m}{\partial x_n}(\mathbf{x}) \end{pmatrix}. \quad (2.47)$$

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

$$d(h \circ g)_{\mathbf{x}} = dh_{g(\mathbf{x})} \circ dg_{\mathbf{x}} \quad (2.48)$$

$$J_{h \circ g}(\mathbf{x}) = J_h(g(\mathbf{x}))J_g(\mathbf{x}). \quad (2.49)$$

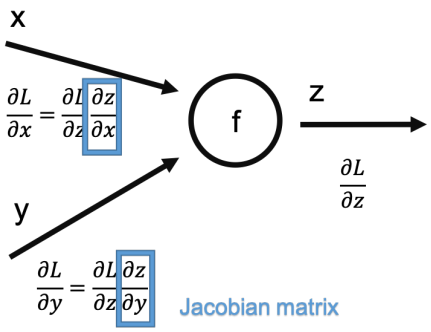


Figure 2.18: Derivatives of a layer.

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

$$F(\mathbf{x}; \theta) = \begin{pmatrix} f_1(\mathbf{x}; \theta_1) \\ \vdots \\ f_m(\mathbf{x}; \theta_m) \end{pmatrix} \quad (2.50)$$

where $f_1, \ldots, f_m$ are the individual neurons with respective parameters $\theta_1, \ldots, \theta_m$ 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**:

$$J_F = \frac{\partial F}{\partial \theta} = \begin{pmatrix} \frac{\partial f_1}{\partial \theta_1} & 0 & 0 & \cdots & 0 & 0 \\ 0 & \frac{\partial f_2}{\partial \theta_2} & 0 & \cdots & 0 & 0 \\ 0 & 0 & \frac{\partial f_3}{\partial \theta_3} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & \frac{\partial f_m}{\partial \theta_m} \end{pmatrix}, \quad (2.51)$$

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

$$\frac{\partial f_i}{\partial \theta_i} = \begin{pmatrix} \frac{\partial f_i}{\partial w_{i,1}} & \cdots & \frac{\partial f_i}{\partial w_{i,n}} & \frac{\partial f_i}{\partial b_i} \end{pmatrix}. \quad (2.52)$$

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 Jacobian matrix of the layer with respect 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 2.19, 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:

$$\frac{\partial L}{\partial \theta_3} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial \theta_3}, \tag{2.53}$$

$$\frac{\partial L}{\partial r} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial r}, \tag{2.54}$$

$$\frac{\partial L}{\partial \theta_2} = \frac{\partial L}{\partial r}\frac{\partial r}{\partial \theta_2}, \tag{2.55}$$

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial r}\frac{\partial r}{\partial q}, \tag{2.56}$$

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial q}\frac{\partial q}{\partial \theta_1}, \tag{2.57}$$

where we know how to compute the local gradients from Section 2.2.2 and the primer on the Jacobian matrix.

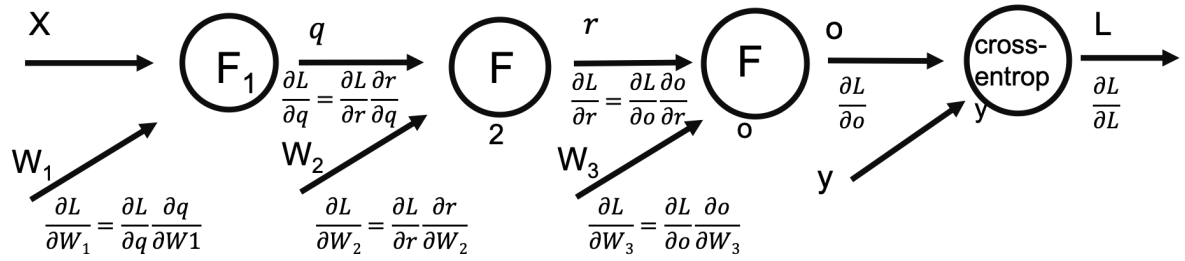


Figure 2.19: Backpropagation over the computation graph of an MLP with three hidden layers.

Figure 2.20: The MNIST dataset. Pixels with a higher value are depicted here as darker.

2.5 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 want to minimize during training;
- what model design we want to use — e.g. the number of neurons and layers, and the type of 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;
- 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 2.20 as an example.

2.5.1 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 with 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 2.20.

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 the output of the right size, again negatively impacting the ability of the model to be 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:

$$x \mapsto \frac{x - 127.5}{127.5}. \tag{2.58}$$

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

2.5.2 Output and Loss

Now that we have the input ready, let's look at the output. As we discussed in Section 2.3, 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. (2.59) — here $\delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$

$$i \mapsto e_i = (\delta_{i,j})_{j=0,\dots,9}. \tag{2.59}$$

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

$$\text{softmax} : (v_0, \dots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\dots,9}. \tag{2.60}$$

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-hot 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

$$p(y) = \prod_{i=0}^{9} p_i^{e_{y,i}}, \tag{2.61}$$

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

$$L(y, p) = -\sum_{i=0}^{9} e_{y,i} \log(p_i) \tag{2.62}$$

$$= -\log(p_y). \tag{2.63}$$

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 on what the output of your network represents. It is 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.

2.5.3 Network Design

We have pre-processed our data, we have decided upon the type of output we want our model to give, choose 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

[7]Cross-entropy is really an information-theoretic measure of error between two probability distributions. The categorical cross-entropy in eq. (2.62) is a measure of how far we are off when using the incorrect probability distribution p instead of the true (deterministic) distribution e_y.

[8]The logarithm is really used here for numerical stability. Throughout this course, you will see the theory of probability distributions being used 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.

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 2.4, the 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 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

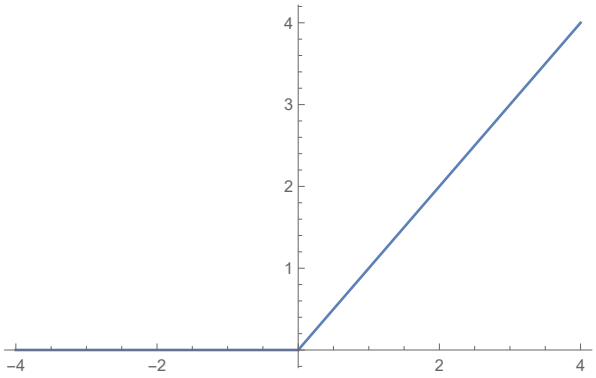
$$\text{ReLU}: \ x \mapsto \max(0, x) \tag{2.64}$$

and depicted in Figure 2.21a. 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.

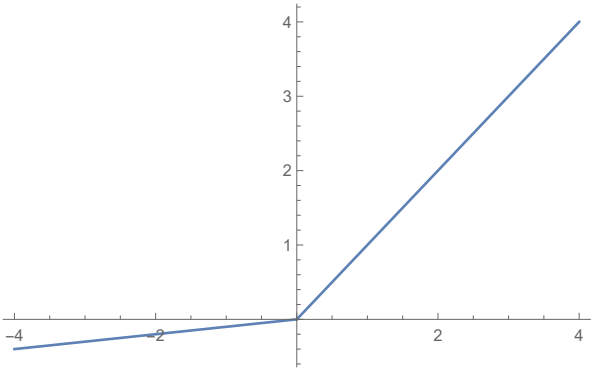
This dying ReLU problem lead to the introduction of a number of other activation functions such as the

- 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.

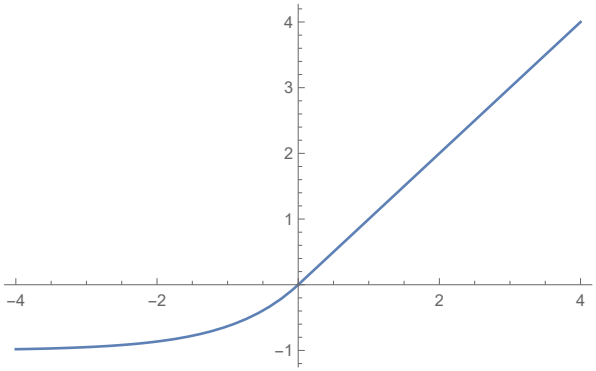
These all solve the dying ReLU problem, but for some of these, the downside is that their gradients are more costly to compute.



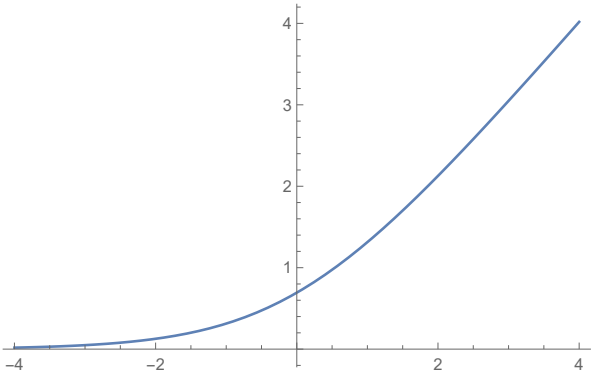
(a) The graph of a Rectified Linear Unit



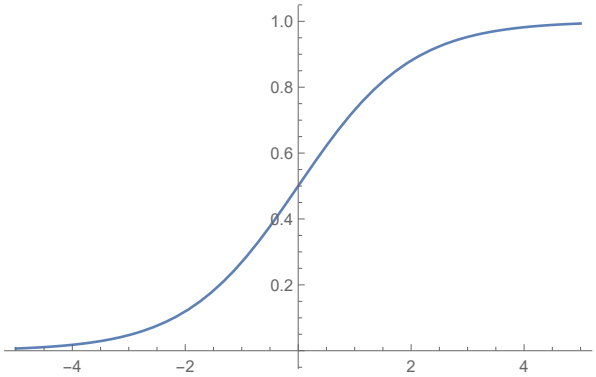
(b) The graph of Leaky-ReLU.



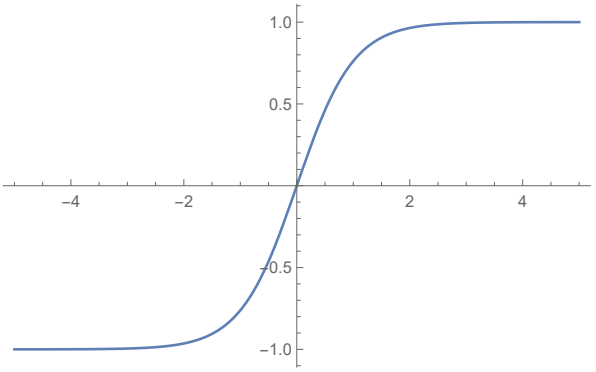
(c) The graph of an Exponential Linear Unit.



(d) The graph of the softplus function.



(e) The graph of the logistic sigmoid function.



(f) The graph of the hyperbolic tangent function.

Figure 2.21: The graphs of various commonly used activation functions.

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.

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 the 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 a larger number of parameters than needed to capture the complexity of the map). The effect of the overparameterization of models in machine learning is an active field of research. Recent work indicates that the usual bias-variance trade-off has significantly different behavior in these modelsfig. 2.22.

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 overfitting 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.

[9] "Reconciling modern machine-learning practice and the classical bias-variance trade-off" Mikhail Belkin, Daniel Hsu, Siyuan Ma, Soumik Mandal; Proceedings of the National Academy of Sciences Aug 2019, 116 (32)

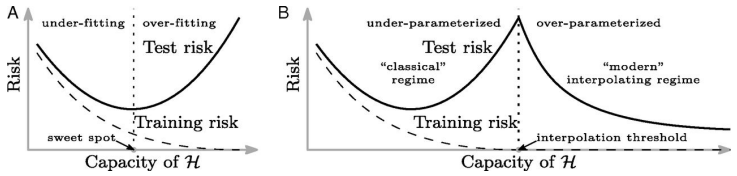


Figure 2.22: Bias-variance in over-parametarized models[9]

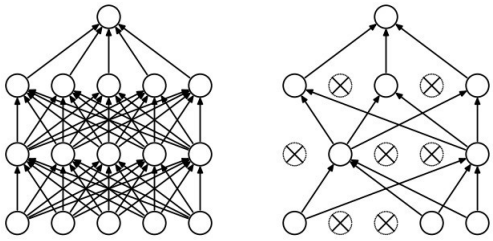


Figure 2.23: Dropout regularization

2.5.4 Regularization

Regularization techniques are one of the key breakthroughs that allowed for the success of deep learning. They both allow for a 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 representation 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 (ridge) 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. 2.23. Dropout is typically applied to the activations of the neurons. Dropout is applied during training of the model by randomly setting a set of 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) on the activations of some of the neurons in the previous layer. In general, we would like to avoid store the data points 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 the compositionality of the representation. Features from the lower layers can be composed to form high-level features in the subsequent layers that result in highly efficient representation.

15849-15854

[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).

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

2.5.5 Initialization

The initialization of the parameters of the neural networks is 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].

2.6 Training

The training process of neural networks is stochastic and comes with 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. (2.65).

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\mathbf{x}; \theta) \tag{2.65}$$

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. (2.67). 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.

$$v \leftarrow \gamma v - \alpha \nabla_\theta L(\mathbf{x}; \theta) \tag{2.66}$$

$$\theta \leftarrow \theta - v \tag{2.67}$$

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 to avoid 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 a significant influence on the training. One other consideration is that for many high-dimensional cases and large models 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

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

[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.

backward pass. We do this typically in parallel for the whole batch. Using large batches can then lead to out-of-memory conditions.

Chapter 3

Word embedding

The learning outcomes of this chapter:

- You are able to develop representations 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 a 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 amino acids.[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 data points 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.

3.1.1 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 3.1.
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 3.2.
If we make the model do predictions for

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

we can depict the process as in Figure 3.3. What happens if we instead want to do predictions for

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

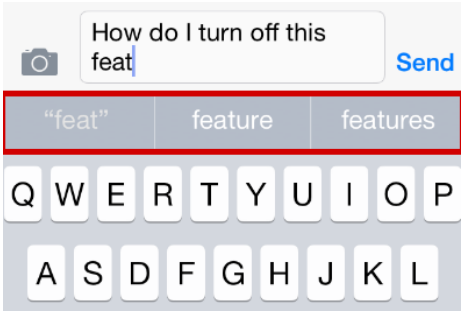


Figure 3.1: Predict the next word

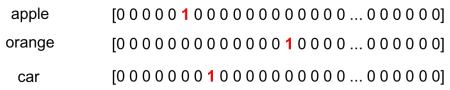
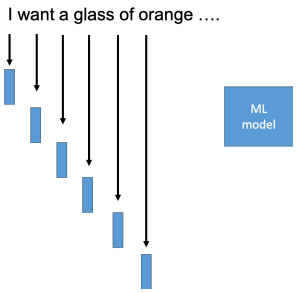


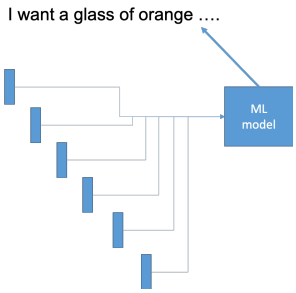
Figure 3.2: Sparse words

The current representation of the data as sparse encoding does not provide any possibilities for the 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 to 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 3.3: Predicting the next word using sparse encoding.

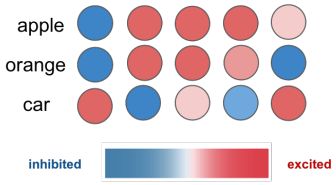


Figure 3.4: Distributed word representation

3.2 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

$$\text{'uncle'} = \text{'aunt'} - \text{'female'} + \text{'male'}. \tag{3.1}$$

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 3.4.

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 we 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".

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

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}), \tag{3.2}$$

[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.

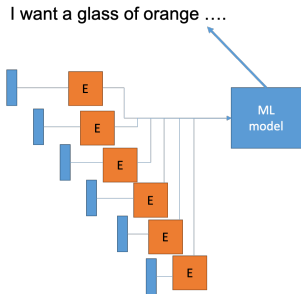


Figure 3.5: The model first uses an encoding map E to map the one-hot vectors to their dense representations, and then uses these dense representations to make predictions.

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:

$$\hat{P}(w_t|w_1^{t-1}) \approx \hat{P}(w_t|w_{t-n+1}^{t-1}). \tag{3.3}$$

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

$$\begin{aligned} \hat{P}(w_t = i|w_{t-n+1}^{t-1}) &= f(i, w_{t-n+1}, \ldots, w_{t-1}) && (3.4)\\ &= g(i, E(w_{t-n+1}), \ldots, E(w_{t-1})), && (3.5) \end{aligned}$$

where $E(w)$ is the embedding of a word w into our embedding space — see Figure 3.5 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 3.6.

In the work of Bengio et al.[3] the function g was implemented as a feed-forward neural network 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 backpropagation and a gradient-based optimization algorithm such as SGD.

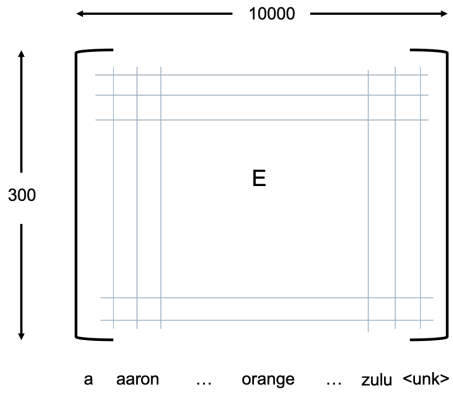
3.3 Further developments

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 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. (3.5)?

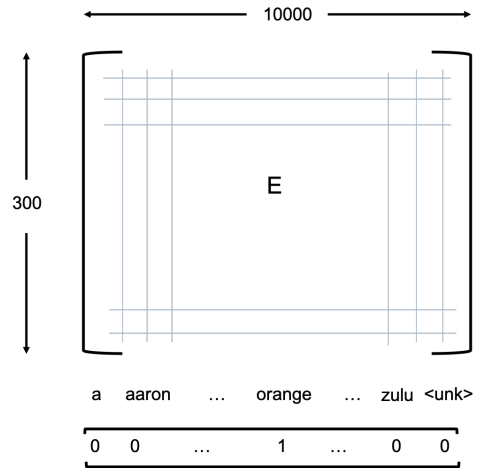
The cheaper g is to train, the larger the vocabulary we can embed and the more data we can handle with a limited amount of resources. Let us, therefore, look at two techniques that simplified the model, allowing us to train word embeddings for much larger vocabularies, or on much more data: *Continuous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

[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 3.6: Embedding matrix

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 3.7, Figure 3.8, and Figure 3.9 at the end of this chapter.

3.3.1 Continuous Bag of Words

The training objective with CBOW is similar to the one in Section 3.2: 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

$$P(w_t \mid w_{t-L}, \ldots, w_{t-1}, w_{t+1}, \ldots w_{t+L}). \tag{3.6}$$

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

$$u_i = Ew_i \tag{3.7}$$

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

$$\hat{u} = \frac{u_{t-L} + \ldots + u_{t-1} + u_{t+1} + \ldots + u_{t+L}}{2L} \tag{3.8}$$

to get some kind of summary of the context.

This context vector is then transformed into a score vector

$$z = C\hat{u}, \tag{3.9}$$

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

$$\hat{P}(w_t \mid w_{t-L}, \ldots, w_{t-1}, w_{t+1}, \ldots w_{t+L}) = \text{softmax}(z). \tag{3.10}$$

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

3.3.2 Skipgram

With Skipgram, the training objective is in a way 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
$$u = E\, w_i \tag{3.11}$$
2. Based on the embedded word, we calculate a score vector for the context, i.e.
$$z = C\, u \tag{3.12}$$
3. We apply the softmax function to the score vector to get our estimated probability distribution
$$\hat{P}(w_j \mid w_i) = \text{softmax}(z). \tag{3.13}$$

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 hear words like "fresh", "juice", "eat", "drink", "tasty", "buy", and "healthy" 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.

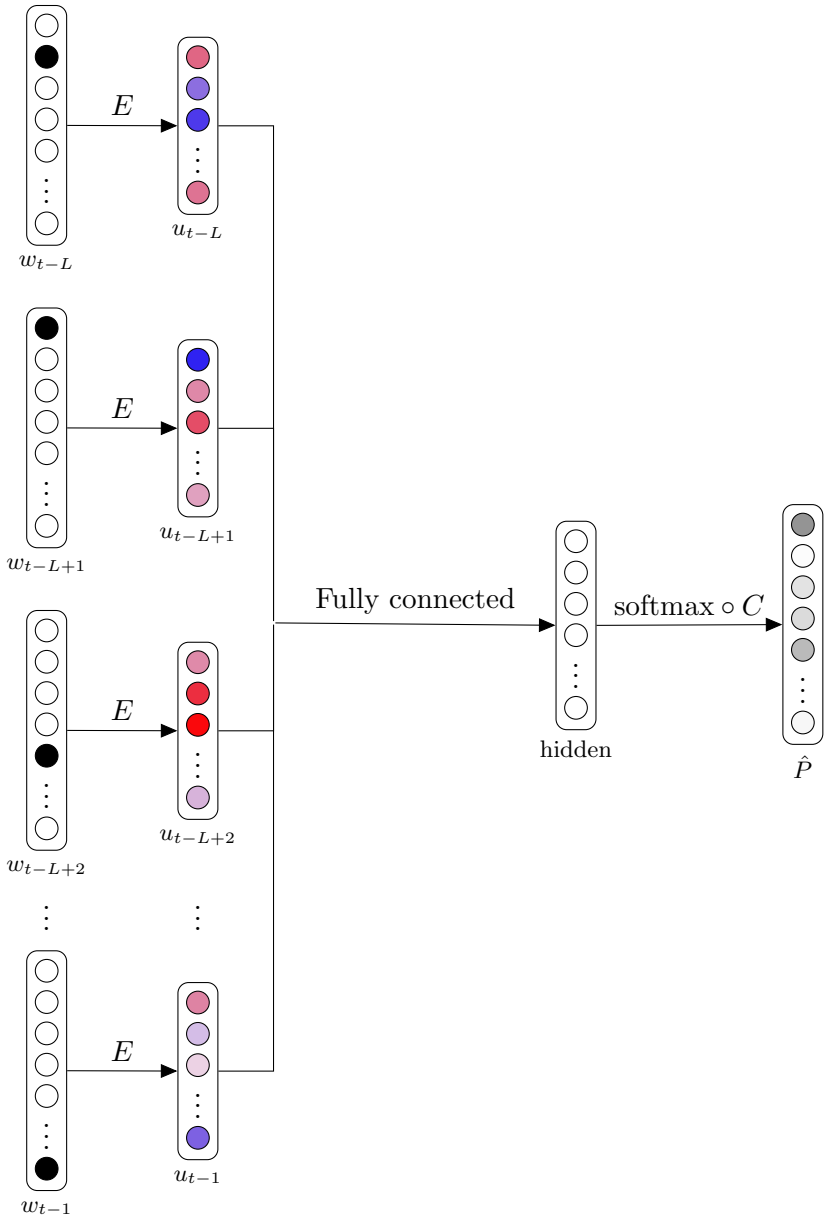


Figure 3.7: Neural Probabilistic Language Model. *Question: what probability measure is being estimated?*

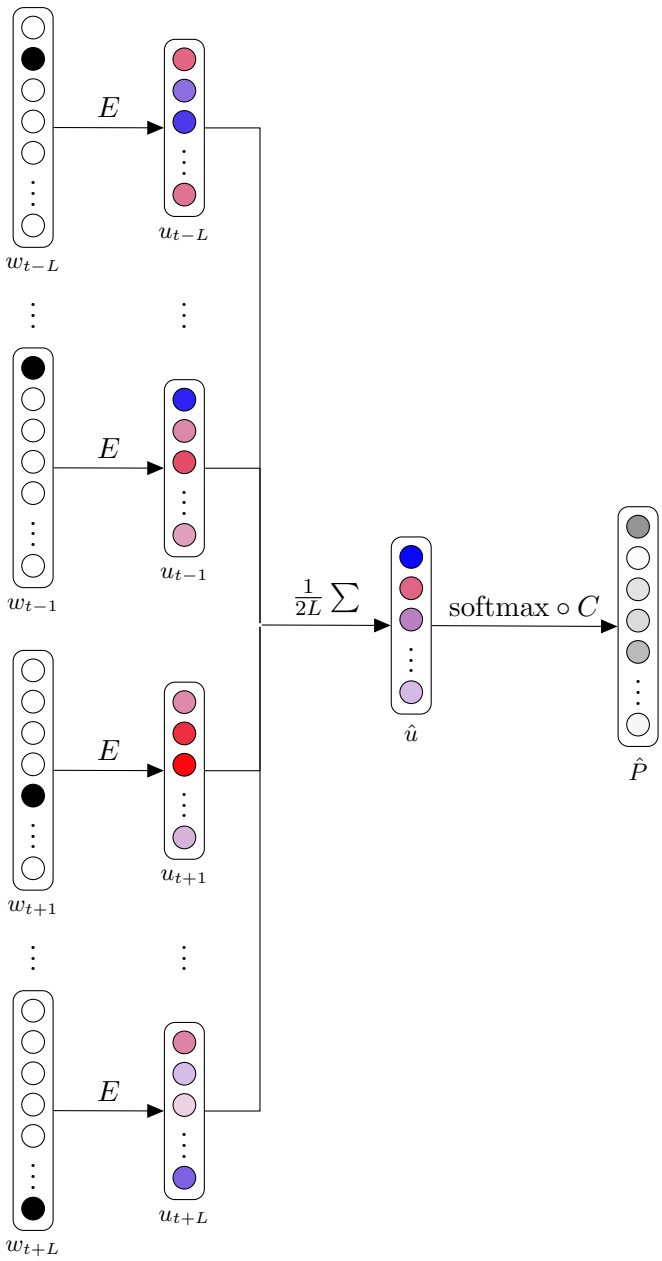
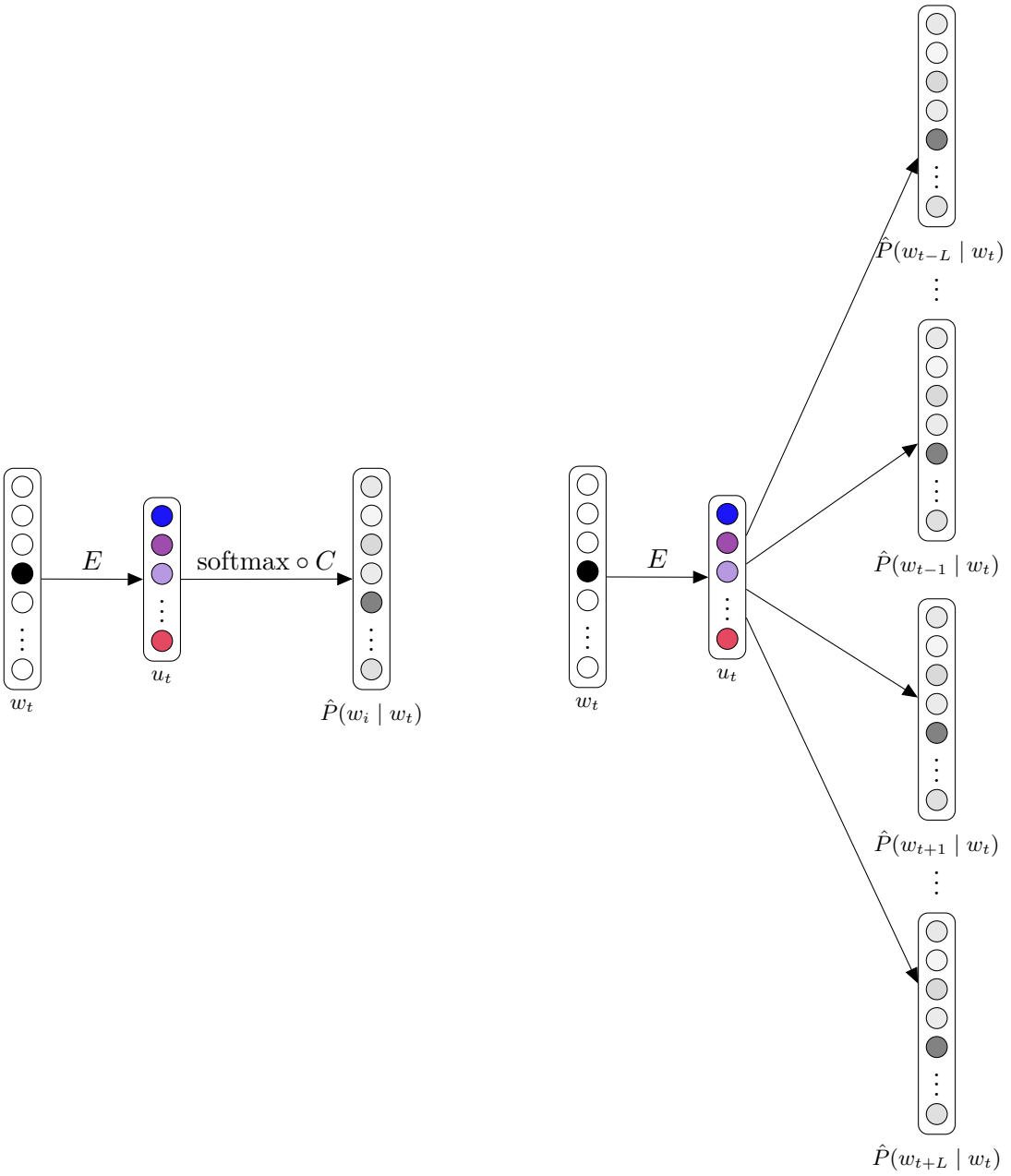


Figure 3.8: Continuous Bag of Words. *Question: what are the differences with Figure 3.7? And what are the similarities?*



(a) Skipgram

(b) Skipgram

Figure 3.9: Two representations of Skipgram. *Question: why are these both accurate representations of the same model?*

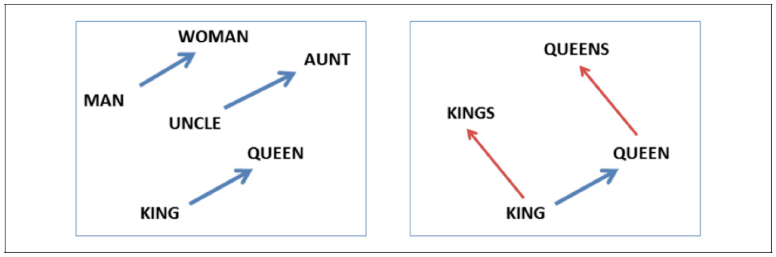


Figure 3.10: Properties of word embedding

3.4 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. 3.10.

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. (3.14).

$$w = \arg\min L(e_w, e_{king} - e_{man} + e_{woman}) \tag{3.14}$$

where L is a distance metric in the embedding space.

Chapter 4

Data Symmetries

The learning outcomes of this chapter:

- Understand the notion of a data domain
- Understand the notions of symmetry, invariance, and equivariance
- Learn how to recognize the data domain of a data structure
- Learn how to recognize the symmetries of a data domain
- Understand which NN layers incorporate which symmetries
- Learn how to select the NN layers to make the model equivariant or invariant to the symmetries present in the data domain

This chapter serves as an introduction to the topic of data symmetries and geometric deep learning. For a more comprehensive introduction to the field that goes into more mathematical detail, please refer to [1].

4.1 Motivation

Deep Learning models develop maps between data in high-dimensional space and targets in lower-dimensional space (Fig. 4.1). As discussed in the previous chapters, Deep Neural Network models do this by learning how to capture the low-level patterns of the data in the initial layers and to compose higher-level patterns by combining patterns of the prior layers. This hierarchical composition of patterns allows neural networks to build efficient high-level representations of the data that are finally mapped to the target lower dimensional space.

Still, learning these maps is a challenging task as the data space can be very high dimensional. In this setting, the data space has very many degrees of freedom and as such we typically need a large number of training data points, which can be expensive to get, and many parameters in the model, which comes with risks of overfitting and increases the training time.

This chapter discusses the possibilities to effectively decrease the degrees of freedom of the data space by incorporating the available understanding of the structure of the data. This has been a critical component in the success of Deep Learning in a number of fields such as image analysis and speech recognition. After reading this chapter you should have a deeper understanding of the motivation for using various neural network layers depending on the structure of the data.

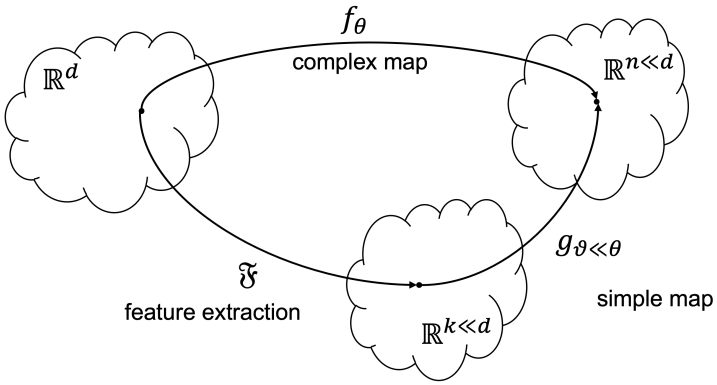


Figure 4.1: Deep Learning mapping

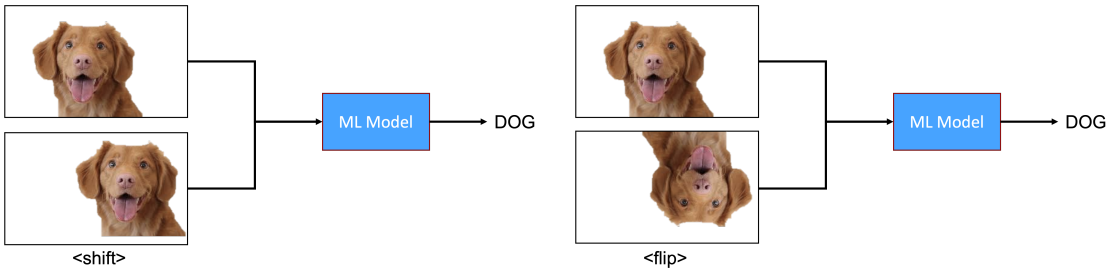
4.2 Introduction

Let us look at a particular example of a model that recognizes an object in an image. If we present the model with an image of a dog, we expect the model to produce a probability distribution over all possible classes in the training data where the largest value is set to the class of a dog. If the dog in the image is slightly translated to the left we expect that this will have no effect on the model's output and the highest class will remain to be the dog. In other words, we expect that our model's output remains *invariant* to the translation of the object in the image.

A **symmetry** is a feature or property that does not change under certain transformations. In this context, we can state that the model's output is symmetric to translation. As we aim to build a model that can learn the high-level concept of a dog, we want our model to be symmetric to any transformations to the data that leave the object still being a dog. As such, the task of learning how to classify dogs in images is exactly the same as learning all the symmetries of dogs in images – if we have a model that is symmetric to all these transformations, any dog that it will see will still be classified as a dog.

Looking at the data space, we can imagine a subspace in that data space where all images of dogs exist. Moving in that space is akin to applying transformations to one instance of a dog and arriving at another (Fig 4.3). However, not all transformations are equal.

While for all transformations the images are changed, some of the transformations do not change the instance of the dog, while other transformations give us a different dog. We refer to this type of symmetry as *semantic symmetry*. The goal of developing a model to recognize the content



(a) Image shifted

(b) Image flipped along the vertical axis

Figure 4.2: Transformations to the image that do not change the image label

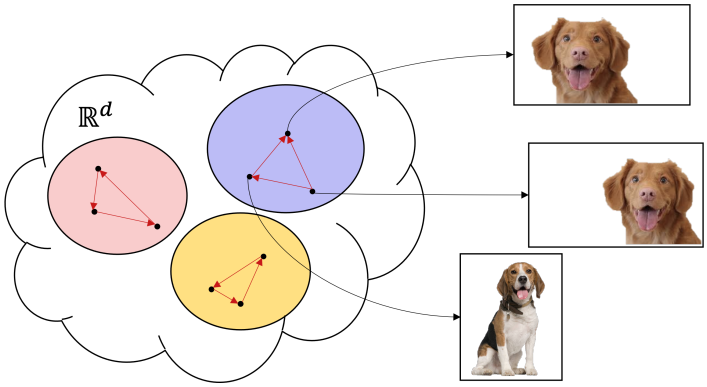


Figure 4.3: Images of dogs in abstract $\mathbb{R}^d$ dataspace

of the images is basically learning this semantic symmetry. As the transformations between the different dogs are highly complex, we aim to learn this symmetry by training the model with many examples of dogs. However, transformations such as translation are much simpler geometric transformations. If we can incorporate this transformation directly in our model (i.e. make the model symmetric to translation *by design*) we effectively make the learning task much easier as the model needs to learn fewer transformations to achieve semantic symmetry.

More generally, we can identify a set of transformations that are related to the structure of the data (or more specifically the structure of the *data domain*). If we can incorporate this set into the model, it would not have to learn these symmetries anymore, making the learning process significantly more efficient.

You can think of the data domain as a set of variables with specific structures or relationships. For example, they can be organized in a grid, where each variable has a fixed number of neighbors, or as nodes in a graph, where the edges represent the relationships between the nodes. A data point is then defined as an assignment of a value to the variables in the domain.

More formally, we denote the data domain as Ω, and the set of possible values the domain can take as $\mathcal{C}$. For example, in the case of images, Ω is a regular grid, and $\mathcal{C}$ is $\{0, \ldots, 255\}^3$, the set of possible pixel values in each of the three color channels. One data point, e.g. a specific image, is then denoted as a *function* $x : \Omega \to \mathcal{C}$. Sometimes, x is referred to as the *signal* on the domain. We can then denote our dog classification model as a function $f : \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is the set of all possible signals (e.g. images), and $\mathcal{Y}$ is the set of classes. So, in this perspective, a data point is a function x that maps each point $u \in \Omega$ to a value $c \in \mathcal{C}$ (e.g. RGB values), and a model maps a function $x \in \mathcal{X}$ to a class $y \in \mathcal{Y}$.

In the case of image data, any local pattern that exists on the grid when translated to another location does not change (Fig. 4.5). The symmetry exists because if we translate the domain, i.e. the grid, by a fixed amount, by shifting the coordinates the information on the domain does not change as the grid has a homogeneous topology. As such, the grid domain itself is invariant to translation (Fig. 4.6). For a two-dimensional grid, this symmetry exists for translating along the two spatial dimensions. For a one-dimensional grid or a chain, the translation symmetry of local patterns exists along one spatial axis as we can see in the figure of the ECG data (Fig 4.7).

When the data exists on a more general (i.e. less structured) data domain such as a graph, the translation symmetry is not present as we cannot translate (reposition) the data on the graph without changing the relative relationship between the node values as the graph does not have homogeneous topology. On the other hand, the permutation transformation (or reindexing)

that does not change the topology of the graph leaves the graph unchanged (Fig 4.8).

Before we go forward let us formalize the concept of invariance and introduce and formalize the concept of equivariance. Given a transformation g (e.g. translation) and a data point x. The function f is *invariant* to the transformation g if $f(g \cdot x) = f(x) = c$ (Fig 4.9). We define the function f to be equivariant to the transformation g if $f(g \cdot x) = g \cdot f(x)$ (Fig 4.10). For example, a model that processes images would be equivariant to translation if first applying the translation to the input data and then processing this with the model would result in the same outcome as first processing the original input and then applying the transformation to the output.

4.3 Equivariant neural networks

Now that we know that image data resides on the grid domain and features on the grid are symmetric to translation, we would like to incorporate this symmetry in our model. To achieve this we need to use a neural network layer that is equivariant to translation when operating on the grid. In this way, the layer can learn to recognize the patterns in the images regardless of their location. For example, the equivariant 'star point detection' model of Fig. 4.10 identifies the locations of the points regardless of the star's translation. In the case of classification, to achieve the invariance in the overall detection of objects we would need to follow up the equivariant layers with an invariant aggregation layer that would enable this.

The way we introduce equivariance to a transformation in a layer is by incorporating the transformation in the layer mechanics and with parameter re-use. Let us examine this statement in more detail by looking at the following example.

Figure 4.11 shows how data on a grid, e.g. an image, is processed by a fully connected neuron, versus a convolutional neuron. Each line in the image stands for a learned weight parameter of the neuron. In the fully connected case, let's assume the model has learned a different weight for each line. Now, if we translate the grid, the lines attach to different nodes, meaning that the learned weights get multiplied by different input values. As such, a fully connected neuron is neither invariant nor equivariant to translations and consequently does not respect the grid's symmetries.

Let us now look at the case of a convolutional neuron. Convolutional neural networks will be explained in more detail later in these lecture notes, but for now, a convolutional layer processes

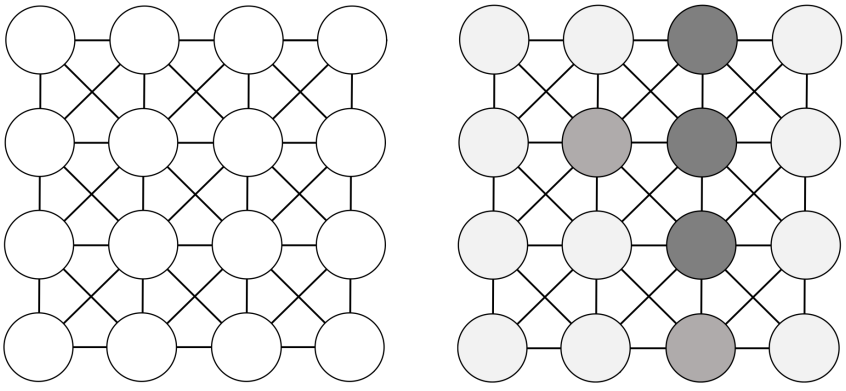


Figure 4.4: The gird data domain

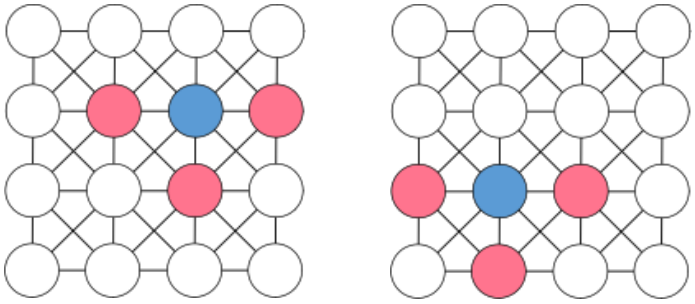


Figure 4.5: Translation invariance of local patterns on the grid domain

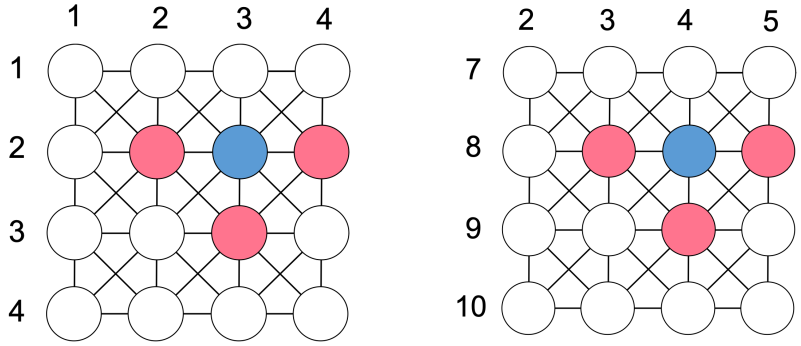


Figure 4.6: Invariance to coordinate translation on the grid domain

an image by sliding (i.e. translating) a small patch of weights over the input grid, instead of processing all nodes at once. As such, the same weights (lines in Fig. 4.11) are re-used by connecting them to different nodes for each translation of the patch, as illustrated in Fig. 4.12.

Now, what would happen if we translated the input grid? Since the same set of weights are re-used over the entire grid to produce the activation map of the neuron, a translation of the grid will result in the same activation map, but translated by the same translation! To visualize this, let's apply the convolutional neuron of Fig. 4.12 to a grid with some data, illustrated in Fig. 4.13. Here, we have trained the neuron to detect a feature of one blue node surrounded by a specific pattern of pink and white nodes. If it detects such a pattern, it will produce a 'yellow' activation; if not, it produces a 'blue' activation. We see that translating the input grid by one unit to the left and one unit downwards results in the same activation map as before, but also translated by one unit left and one unit downwards, meaning the convolutional neuron

Figure 4.7: Example of 1D signal on a grid

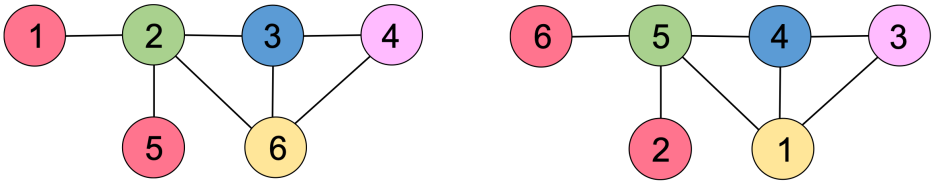


Figure 4.8: Permutation of the nodes on a graph does not change the features

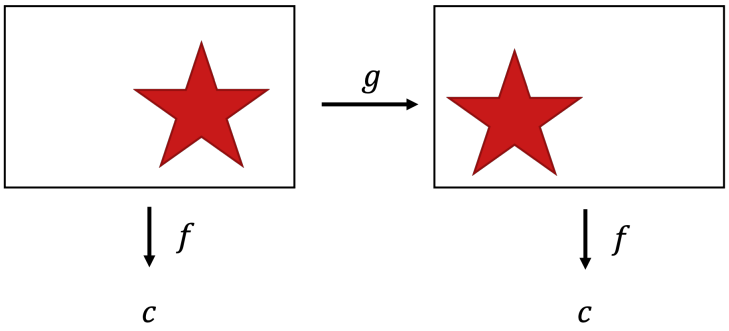


Figure 4.9: Illustration of invariance

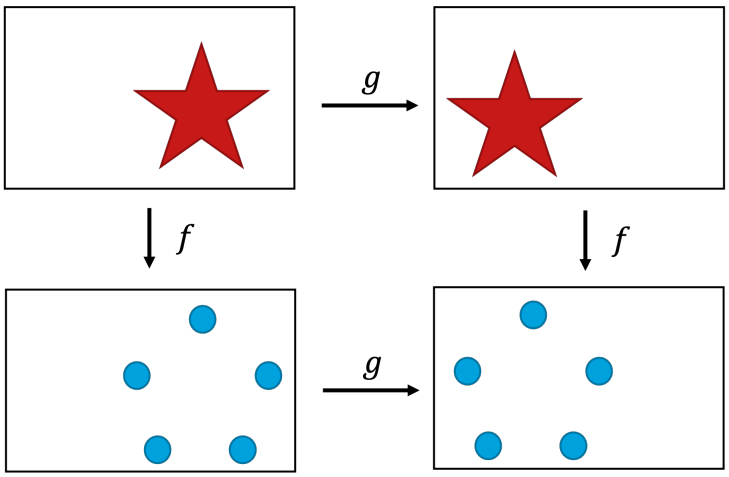
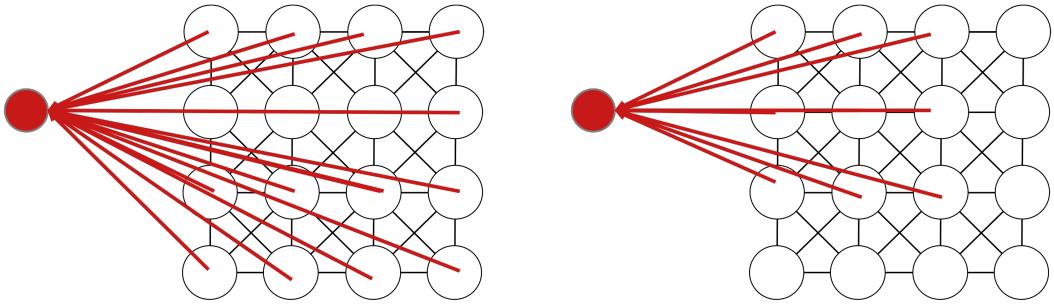


Figure 4.10: Illustration of equivariance



(a) Fully connected neuron (b) Convolutional neuron

Figure 4.11: Comparison of the connectivity of a fully connected and convolutional neuron

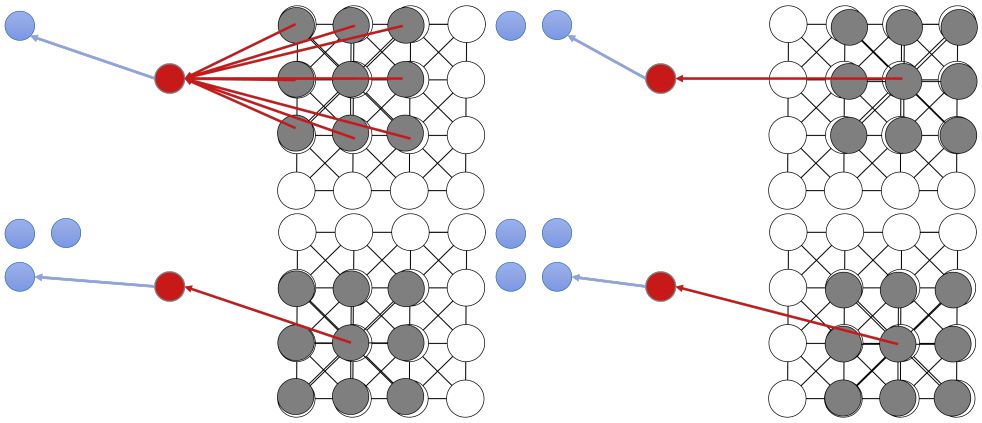
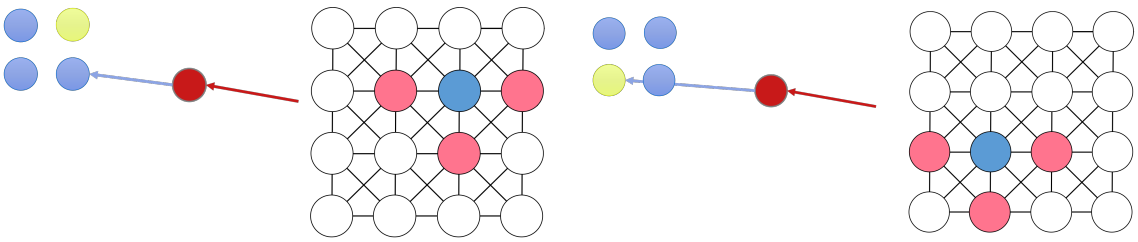


Figure 4.12: Illustration of how a convolutional neuron processes a grid.

is equivariant to translations!

4.4 Geometric Deep Learning framework

Now that we understand that we can design neural network layers that respect the symmetries of the data, the question is how we can use these layers to build an entire model that respects these symmetries. For this, we can refer to a general Geometric Deep Learning blueprint.



(a) Input data (b) Translated input data

Figure 4.13: Visualization of the activation of convolutional neuron before and after translation

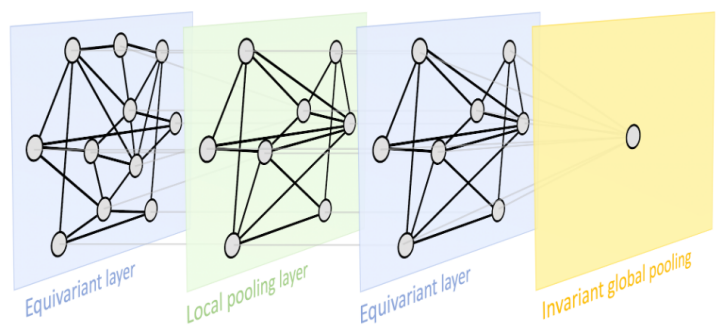


Figure 4.14: Deep Geometric Learning framework. Image source: Bronstein, Michael M., et al. "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges." arXiv preprint arXiv:2104.13478 (2021 [1]

Let Ω be a data domain that is symmetric with respect to a transformation g. We define the following building blocks to design geometric deep learning models:

- Linear equivariant layer: a function $L : \mathcal{X} \to \mathcal{X}'$, such that $L(g \cdot x) = g \cdot L(x)$, which equivariantly maps activations $x \in \mathcal{X}$ of the previous layer to output $x' \in \mathcal{X}'$
- Element-wise nonlinearity: a function $\sigma : \mathcal{C} \to \mathcal{C}'$, which applies a nonlinear activation function to the previous layer's output function values $x \in \mathcal{X}$ to get the activations $x' \in \mathcal{X}'$. The nonlinearity is applied elementwise, so $\sigma(x)$ really means calculating $\sigma(x(u))$ for all $u \in \Omega$.
- Local pooling layer: a coarsening function $P : \mathcal{X} \to \mathcal{X}'$ that locally aggregates x to produce a coarser representation of the data. Note that in this case, the coarses output x' is a function defined on a coarser domain Ω'.
- Invariant global pooling layer: a function $A : \mathcal{X} \to \mathcal{C}$, such that $A(g \cdot x) = A(x)$, which is an aggregation of $x \in \mathcal{X}$ that is invariant to the relevant symmetry g.

Using these building blocks, we can design models f that are invariant to a transformation g as follows:

$$f = \sigma_n \circ A \circ B_n \circ P_n \circ \sigma_{n-1} \circ B_{n-1} \circ P_{n-1} \ldots \circ \sigma_1 \circ P_1 \circ B_1. \tag{4.1}$$

Notably, as long as we have the right tools (equivariant layers, pooling operators, and aggregation layers) at our disposal, we can use the blueprint to design symmetry-respecting networks for a wide range of data domains, such as grids, graphs, sets, sequences, and even more exotic domains like spheres. An illustration of the geometric learning blueprint applied to the graph case can be found in Fig 4.14.

Chapter 5

Learning on grids

The learning outcomes of this chapter:

- Learn to recognize the symmetries of the grid data domain
- Understand the mechanics of the convolutional layer
- Learn the component of the convolutional neural network
- Learn the effects of the filter size, stride, and stacking convolutional layers on the learned features in convolutional neural networks
- Familiarize with a set of successful CNN models and their architectural characteristics
- Learn to identify different image analysis problem formulations from desired goals
- Learn how to embed high dimensional data using Metric Learning

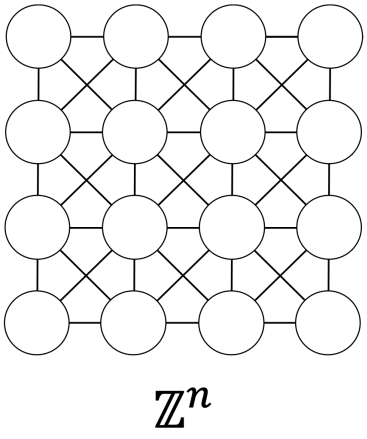
5.1 Introduction

The focus of this chapter is models for data on the grid domain. Many types of high-dimensional data sources exist on the grid domain. Some of the most significant early successes of deep learning on tasks such as image analysis or speech recognition involve data that is well represented on the grid data domain. Therefore, a plethora of model architectures, specifically convolutional neural networks have been devised for this domain. These models are capable of developing very efficient high-level representations of the content of the images.

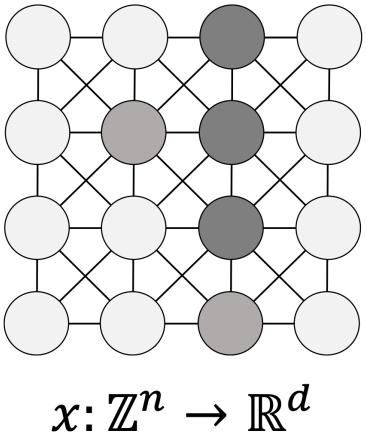
As discussed in Ch. 4 we consider a data domain a set of variables with a specific structure. A graph is an object that allows us to specify this structure such that the nodes of the graph define the variables and the edges of the graph define the relationships between the variables such as their adjacency. In this context, grids are graphs with specific adjacency, such that each node has a fixed number of neighbor nodes (Fig. 5.1). In grids the order of the nodes is fixed so permutational invariance is not required from the model. As a result, we can define a *localized* feature on a graph as a set of connected nodes (Fig. 5.2).

We can define specific transformations on this domain, such as translation and rotation. Based on how this feature is defined, applying these transformations leave the feature unchanged. In other words, the feature is invariant to the given transformations (Fig 5.3).

Following our established goal of learning efficient representations, we aim at identifying low-level features that can be combined in a hierarchical manner into high-level features that would



(a) Grid data domain without signal



(b) Grid data domain with signal

Figure 5.1: Grid data domain

form an overall efficient representation of the data.

To achieve this we need to introduce feature learning mechanisms that are equivariant to these transformations of the data. Therefore, our approach is as follows: identify the data domain, identify the transformations that leave features invariant on that domain, and incorporate pattern detectors that are equivariant to these transformations.

In the case of a 2D grid if the definition of the feature takes into account the adjacency of the variables, transformations such as translation along the horizontal and vertical axis as well as discrete rotations of the domain would leave the feature unchanged. Therefore, developing a feature detection layer that is equivariant to these transformations would be a highly efficient way of learning patterns on this domain.

Contrary to this we could also achieve the same goal by applying these transformations to the training data set and as such exposing our model to the transformed variations of these features with the expectation that the model will develop feature detectors for all the transformed versions of the features. This approach is referred to as *data augmentation* and requires a higher capacity of our model as well as longer training. Nevertheless, incorporating some transformations such as the rotations in this case may introduce significant complexity in our model that we may choose to avoid by using data augmentation. Incorporating rotations would typically increase the activation map of the layer by a factor proportional to all possible rotation val-

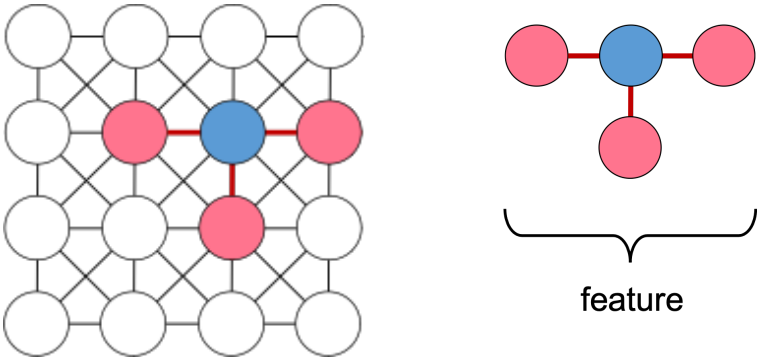
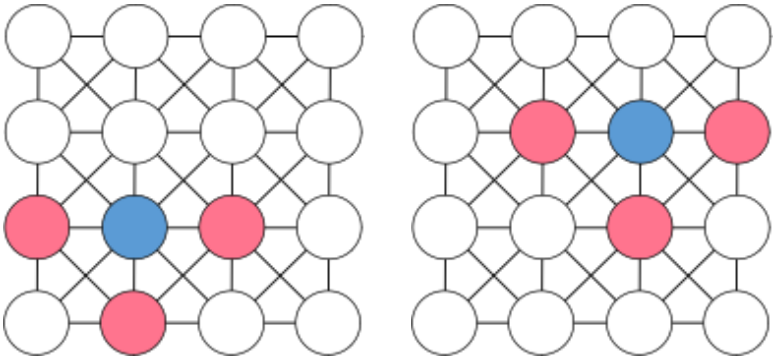


Figure 5.2: Feature on a grid



(a) Original feature

(b) Translated feature

(c) Translated and Rotated by 90 degrees

(d) Translated and Rotated by 180 degrees

Figure 5.3: Feature on a grid with various transformations

ues. Therefore, it is common to balance between incorporating symmetries in the model with equivariant layers and allowing for model capacity and possibly including data augmentations for the remaining data symmetries.

As we aim to develop hierarchical models that learn high-level features, we also need to consider how the detected features in the lower layers can be combined to form higher-level features in the deeper layers of the model. Typically a patterns to a specific level of complexity would be equivariant to the transformations of the domain. Therefore we would aim to include a number of equivariant layers consecutively to learn feature detectors for these higher-level patterns. As we will see in the model architectures in the rest of this chapter typically a sequence of equivariant layers is used to achieve efficient feature learning capability.

5.2 Convolutional neural networks

5.2.1 Convolutional neuron

Let us consider the following task. We develop a model that detects the sequence '1011' in a 1D image (Figure 5.4). We can formulate this as a binary classification. With a single artificial neuron (Figure 5.5) 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 neuron 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 the first layer neurons detect the pattern.

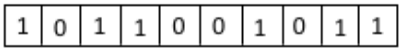


Figure 5.4: An input sequence in which we want to detect and locate the sub-sequence 1011.

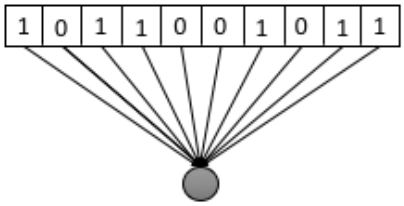


Figure 5.5: The way a regular artificial neuron sees its input.

This model would work as expected, but it will also have a number of parameters that are 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.

In fig. 5.6 such a neuron is depicted. This neuron sees only a part of the image. For a model like this to process the whole image, we would need to slide it across the image. As such we can re-use it in 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 actually use the same second-layer neuron as before. A single neuron that implements the binary OR operation on the output of the convolutional neuron.

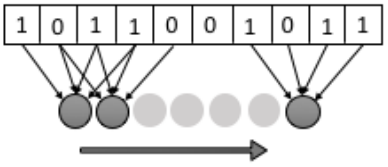


Figure 5.6: A schematic overview of our neuron's field of view sliding along the input sequence.

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

As with the original artificial neuron, the convolutional neuron has a weight for every input in its field of view. These weights together are called the *kernel* of the neuron, and the size of its field of view is referred to as its kernel size. The operation the neuron implements can be described in terms of the mathematical operation of "convolution". The convolution of two sequences is typically[1]defined as follows:

$$(a * b)_n = \sum_{i=-\infty}^{\infty} a_i b_{n-i} \tag{5.1}$$

$$= \sum_{i=-\infty}^{\infty} a_{n-i} b_i \tag{5.2}$$

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 5.7. Note that the

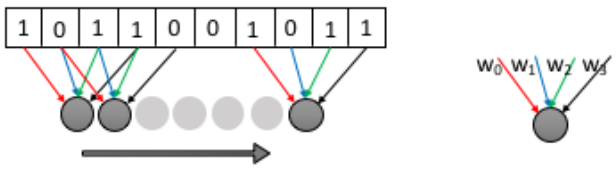


Figure 5.7: An illustration of the neuron operation described in eq. (5.3).

Figure 5.8: A selection of images from 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?

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

$$o_\theta(x) = \phi(x * k + \mathbf{b}), \tag{5.3}$$

where $\mathbf{b}$ is the vector with b at every index. Note that in order for the weights to be applied as in Figure 5.7 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 5.7 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.

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

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 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.

5.2.2 Padding

There is one problem with our description in eq. (5.3) 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 of 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 5.10 for a visual representation of this. On the interior of the original input sequence, i.e. where the neuron's 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 5.11.

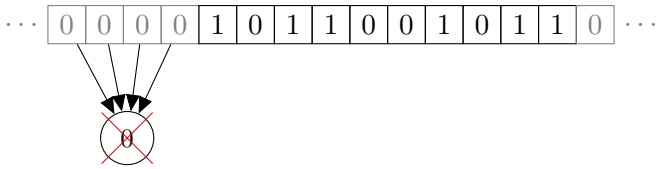


Figure 5.10: Theoretical zeros in (5.1) far away from the input are ignored and outputs there are not computed. The padded zeros are drawn in gray.

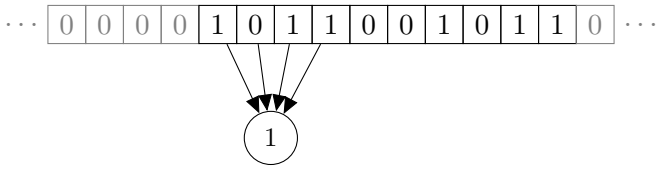


Figure 5.11: On the interior of the original input sequence outputs clearly need to be calculated. The padded zeros are drawn in gray.

However, the question remains what we should do at the edge of our input sequence — see Figure 5.12. 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.

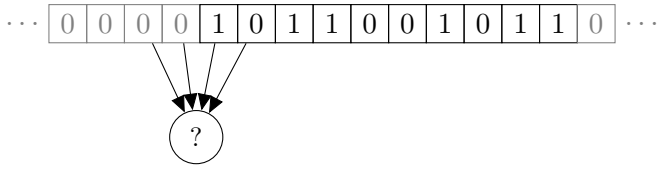


Figure 5.12: What should happen at the edge of the input sequence?

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 the same padding are the two common ways of padding.

5.2.3 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 addition to computing the gradient of the output with respect to its input, we also need to compute the gradient of its output with respect to its parameters such that we can update these parameters during training.

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.

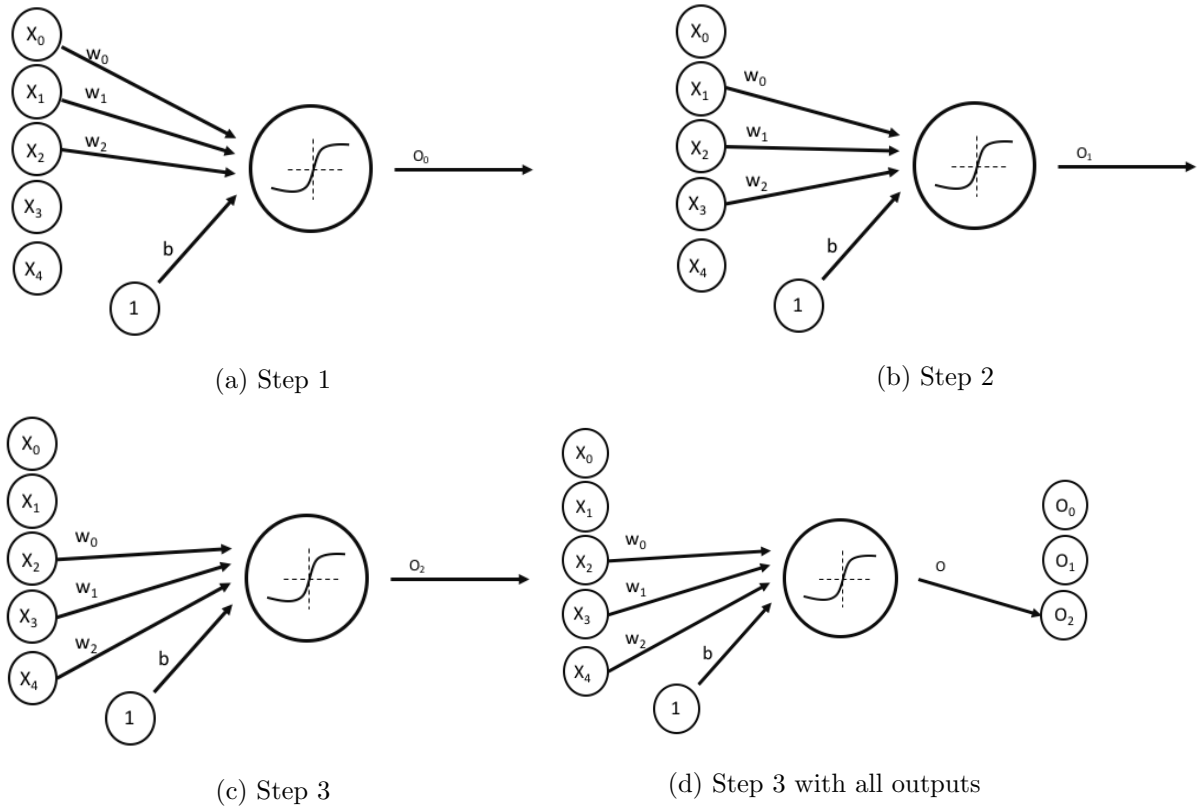
In Figure 5.13 we can observe the forward steps. The neuron processes an input with a size of 5 $(x_0, ..x_4)$, with a kernel of length 3 (w_0, w_1, w_2). In the first step at the initial position, the neuron computes the output o_0 Figure 5.13a. It the second and third steps outputs O_1 Figure 5.13b and o_2 Figure 5.13c are computed.

We can represent these outputs as an activation map vector Figure 5.13d

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 5.14).

Let us consider the compute graph of the convolutional neuron at each step of the convolution to compute the backward pass values Figure 5.15. In this case it is more effective to look at separate compute graphs for each step in the convolution Figure 5.15a and Figure 5.15b, and Figure 5.15c. 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 5.15d and Figure 5.15e).

[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 5.13: Forward pass of convolutional neuron

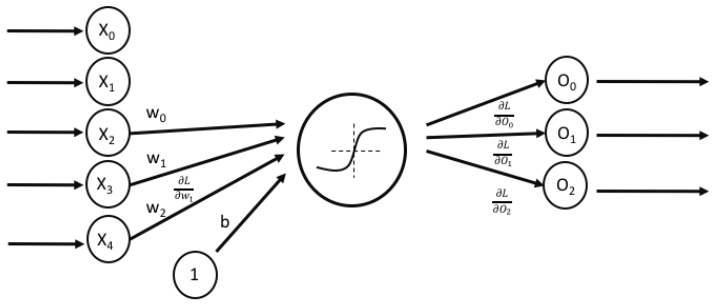
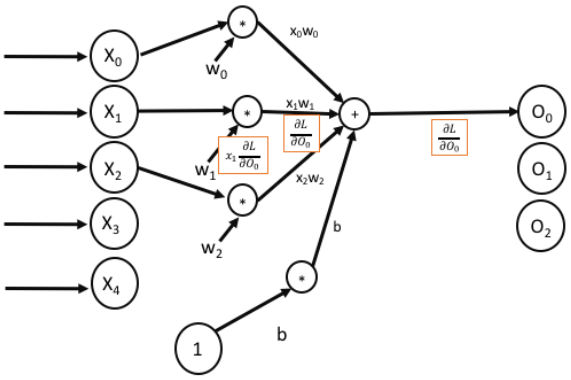
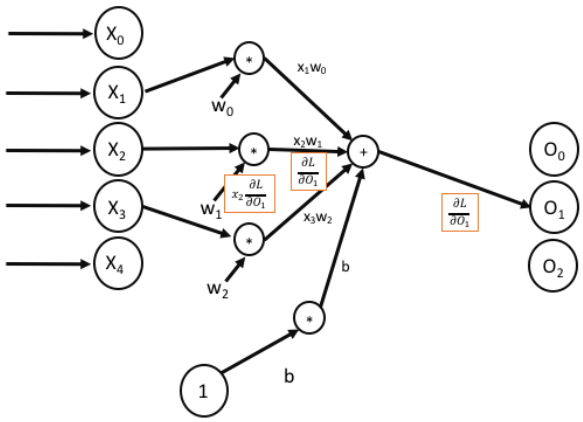


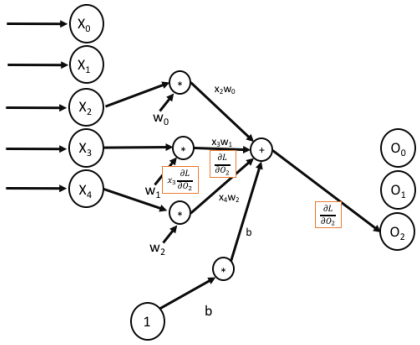
Figure 5.14: Convolutional neuron receives gradient updates from subsequent layers for each output



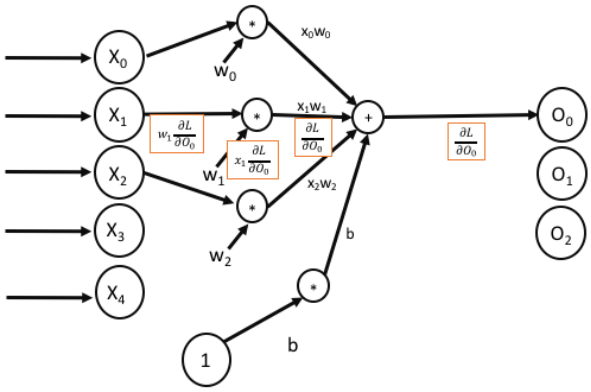
(a) Backpropagating the update value for w_1 at step 1



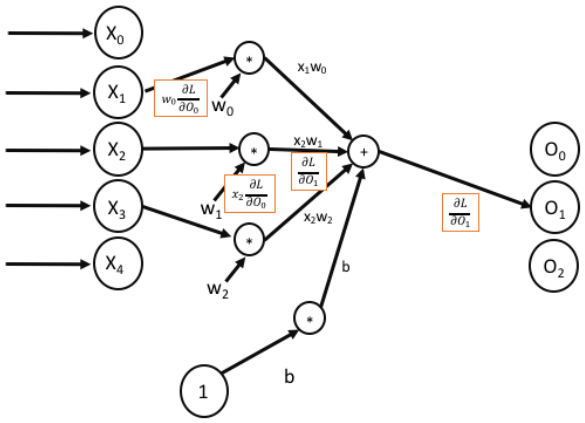
(b) Backpropagating the update value for w_1 at step 2



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

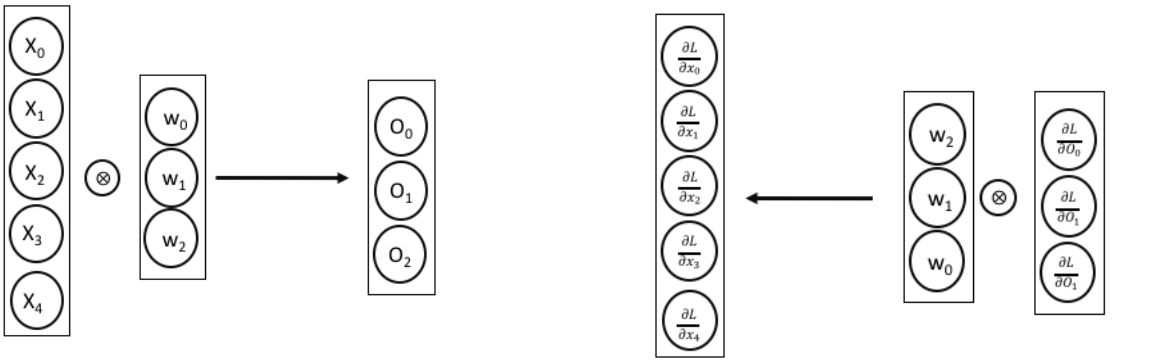


(d) Backpropagating the update value for x_1 at step 1



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

Figure 5.15: 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 5.16: Backpropagation convolutional neuron (vector form)

The same operations can be expresses in a condensed way using vector representations Figures 5.16a and 5.16b. 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 on the type of padding we perform. However, for 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 a kernel

$$\begin{aligned} \mathbf{k} &= (k_0, \ldots, k_K) && (5.4)\\ &= (w_K, \ldots, w_0), && (5.5) \end{aligned}$$

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

$$\begin{aligned} o_i &= \phi(y_i + b) \qquad \text{if an output is required, otherwise 0,} && (5.6)\\ y_i &= \sum_{j=-\infty}^{\infty} x_{i-j} k_j && (5.7)\\ &= \sum_{j=-\infty}^{\infty} x_j k_{i-j}. && (5.8) \end{aligned}$$

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:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial b} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}), \tag{5.9}$$

$$\frac{\partial L}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{y}} = \frac{\partial L}{\partial \mathbf{o}} \phi'(\mathbf{y} + \mathbf{b}), \tag{5.10}$$

$$\begin{aligned} \frac{\partial y_i}{\partial k_j} &= x_{i-j}, \\ \frac{\partial L}{\partial k_j} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial k_j}, \end{aligned} \tag{5.11}$$

$$\begin{aligned} \frac{\partial y_i}{\partial x_l} &= k_{i-l}, \\ \frac{\partial L}{\partial x_l} &= \sum_i \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x_l}, \end{aligned} \tag{5.12}$$

where ϕ' is applied element-wise. Here equations (5.11) and (5.12) can be summarized as

$$\frac{\partial L}{\partial k_j} = \sum_i \frac{\partial L}{\partial y_i} x_{i-j}, \tag{5.13}$$

$$\frac{\partial L}{\partial x_l} = \sum_i \frac{\partial L}{\partial y_i} k_{i-l}, \tag{5.14}$$

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* $\mathbf{o} = (o_0, o_1, o_2)$*?*

5.2.4 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 in 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. (5.1) then becomes an

[3] This is called cross-correlation.

inner product, so in the case of 1-dimensional convolution this becomes

$$(\mathbf{a} * \mathbf{b})_n = \sum_{i=-\infty}^{\infty} \mathbf{a}_i^\top \mathbf{b}_{n-i} \tag{5.15}$$

$$= \sum_{i=-\infty}^{\infty} \sum_{j=0}^{C-1} a_{i,j} b_{n-i,j}. \tag{5.16}$$

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

5.2.5 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 models with high capacity. Such layers of convolutional neurons are referred to as convolutional layers.

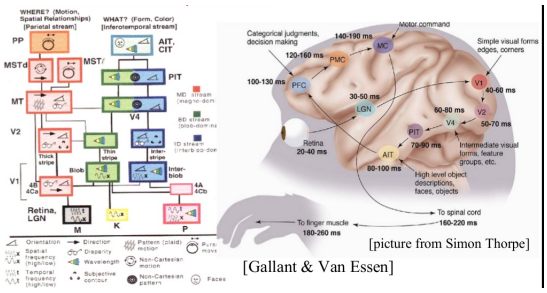


Figure 5.17: Processing visual information in a sequence of detections of low level concepts to high level concepts is also observed in biological systems as the human visual system.

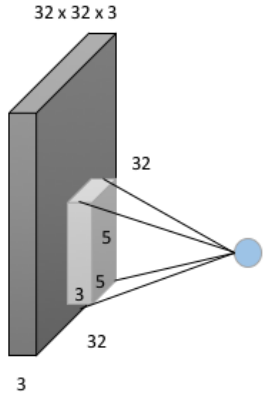
In this context, a convolutional neuron is typically referred to as a "filter". A convolutional layer has multiple filters with the same activation function, the same kernel size, and the same kind of padding. The outputs of the filters are stacked together so that if a 2-dimensional filter transforms 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 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 that can be stacked, see Figure 5.18.

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 backpropagation and gradient-based optimization algorithms to train these models.

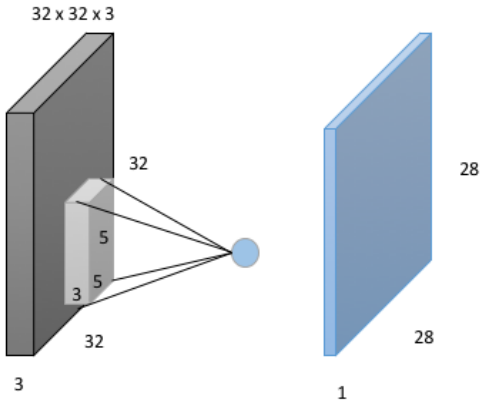
5.2.6 CNN Models

The architecture 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. 5.19a 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. 5.19b for an example of this.

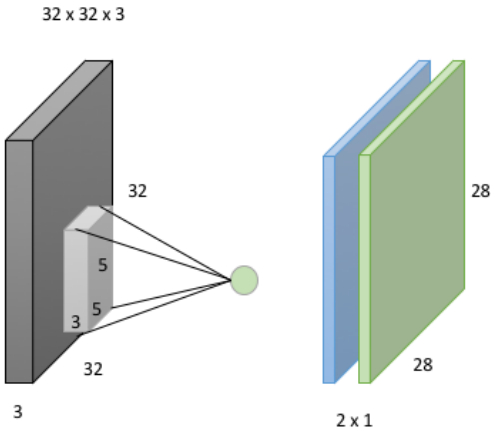
[4]The order actually depends on the choice of channels convention. If we use the channels first convention we get a $K \times N \times M$ tensor.



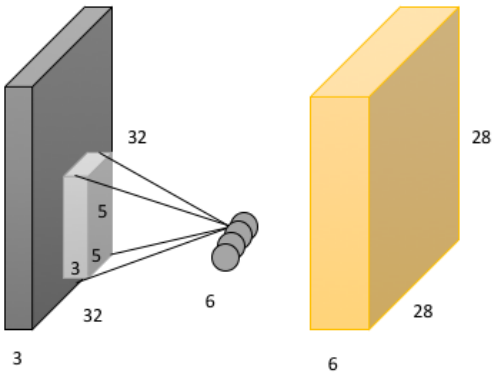
(a) We want to apply a neuron with a $5 \times 5 (\times 3)$ kernel to a $32 \times 32 \times 3$ image.



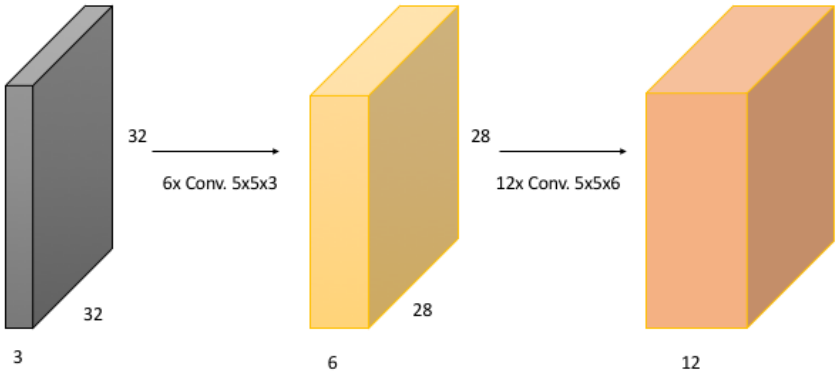
(b) Convolving the neuron over the entire image using "valid" padding, we get a $28 \times 28 \times 1$ output.



(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.

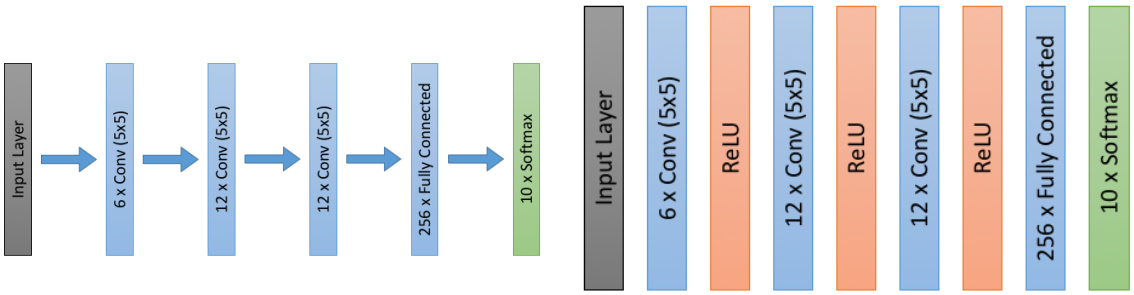


(d) We can stack more neurons this way to get a convolutional layer with 6 filters, giving a $28 \times 28 \times 6$ output.



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

Figure 5.18: 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 5.19: Two depictions of an example CNN architecture.

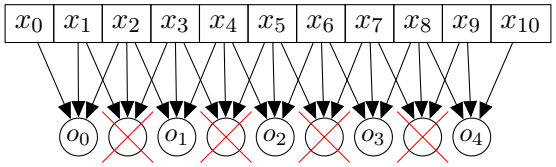


Figure 5.20: Strided convolution with a stride size of two,and a kernel of size three.

5.2.7 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 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 dimensions, we could also specify different stride sizes for the different dimensions. A graphical representation of strided 1-D convolution is given in Figure 5.20.

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

- Accepts:
  - $W_1 \times H_1 \times D_1$
- Outputs:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$

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

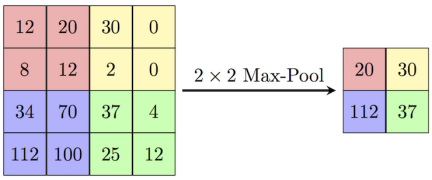


Figure 5.21: Subsampling - maxpooling

- 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' 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 5.21.

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:

$$a(x) = \max_{i \in \{0,\ldots,m-1\}} x_i \qquad \text{has derivatives}^6 \tag{5.17}$$

$$\frac{\partial a(x)}{\partial x_i} = \begin{cases} 1 & \text{if } x_i = \max(x) \\ 0 & \text{otherwise,} \end{cases} \tag{5.18}$$

and

$$a(x) = \frac{1}{m} \sum_{i=0}^{m-1} x_i \qquad \text{has derivatives} \tag{5.19}$$

$$\frac{\partial a(x)}{\partial x_i} = \frac{1}{m}. \tag{5.20}$$

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 5.22 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.

[6] This is in the case of a unique maximum. If the maximum is attained at two or more distinct indices we only

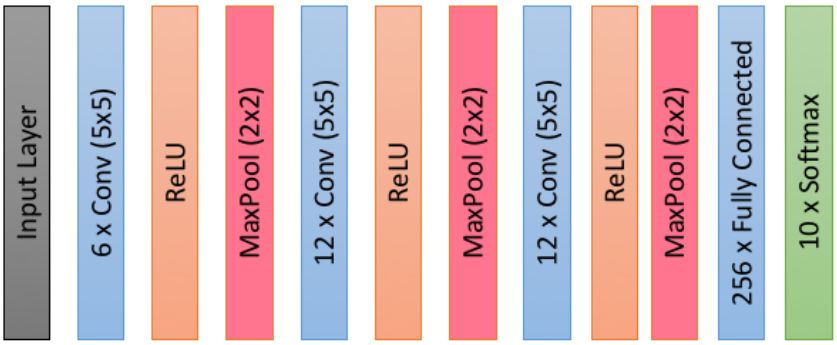


Figure 5.22: CNN - architecture with maxpool layers

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.

have one-sided derivatives for those indices.

5.3 Image Analysis with CNN

5.3.1 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. 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 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:

$$y \mapsto e_y = (\delta_{y,j})_{j=0,\ldots,9}. \tag{5.21}$$

We use a Softmax layer to interpret the output of our network as a probability distribution, $\mathbf{p}$:

$$\text{softmax} : (v_0, \ldots, v_9) \mapsto \left(\frac{e^{v_j}}{\sum_{i=0}^{9} e^{v_i}} \right)_{j=0,\ldots,9}, \tag{5.22}$$

and for our loss we use

$$L(y, \mathbf{p}) = -\sum_{i=0}^{9} e_{y,i} \log(p_i) \tag{5.23}$$

$$= -\log(p_y). \tag{5.24}$$

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 5.23.

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.

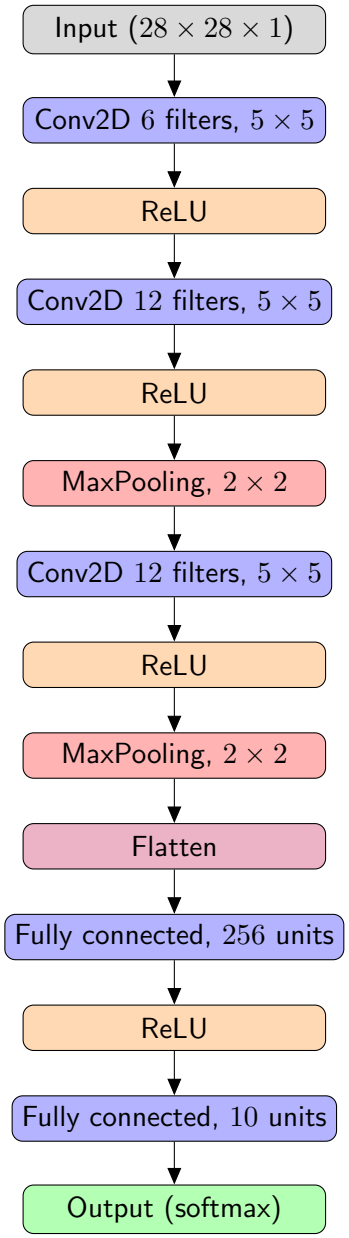


Figure 5.23: The architecture performing classification on the MNIST dataset.

Note: Convolutional layers give us a natural way to develop models that are invariant to the 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 a 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.

5.3.2 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 5.24, you can see on the 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 these activation maps. For example, you can see that the first convolutional layer has 6 neurons based on the dimensionality of the first activation map.

[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")

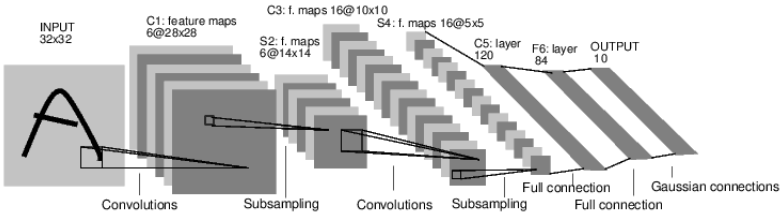


Figure 5.24: LeNet model. LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

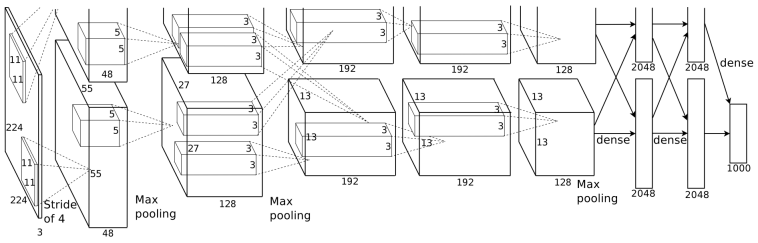


Figure 5.25: The AlexNet: Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

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 a resolution of 224 by 224 labeled with 1000 different classes. The model is depicted in fig. 5.25 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 is the large number of parameters. Relative to its depth

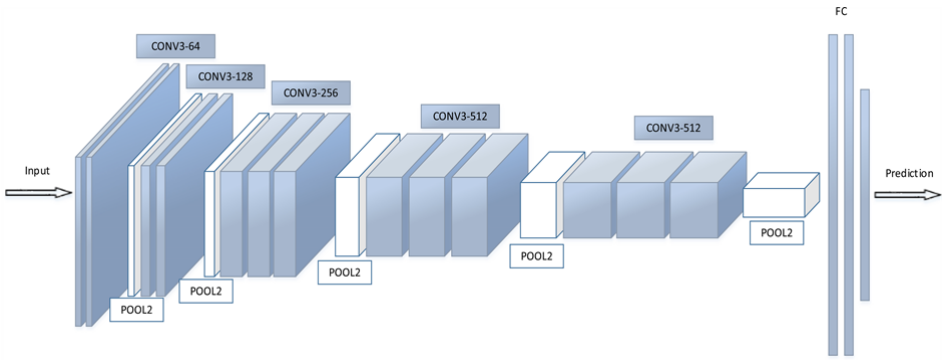


Figure 5.26: VGGNet: Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

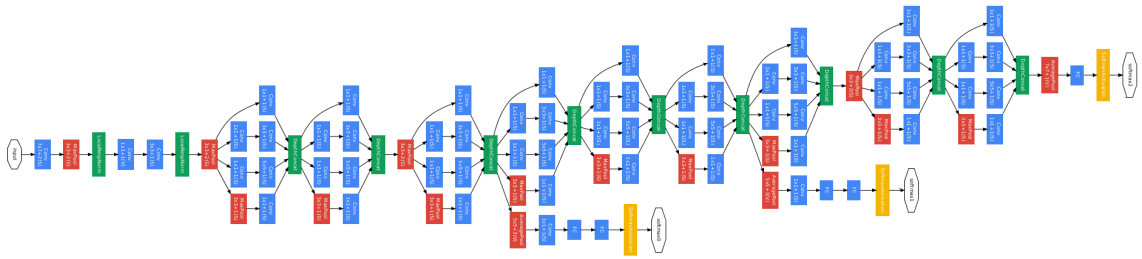


Figure 5.27: GoogleNet: Szegedy, Christian, et al. "Going deeper with convolutions." Cvpr, 2015.

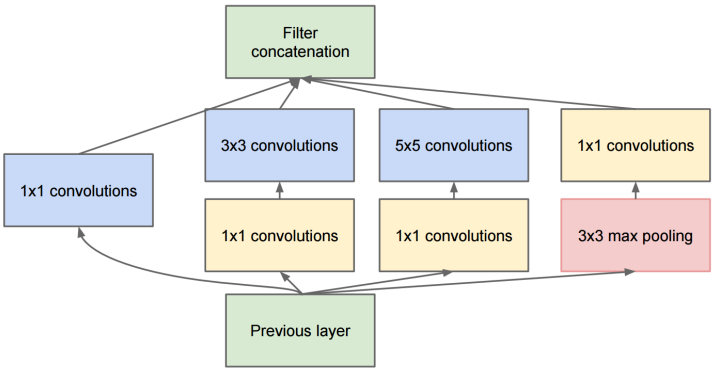


Figure 5.28: Inception module

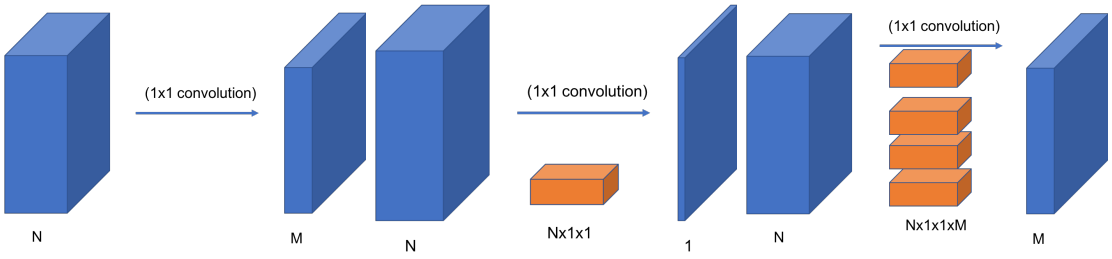
VGGNet uses many parameters, particularly after the last convolutional layer, the dense layer that follows uses a large about of parameters to process that last activation map.

The GoogleNet model (Figure 5.27) that relies on the inception block addressed this challenge. This architecture is deeper and uses fewer parameters than VGGNet. It also delivers superior performance. 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 5.28.

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 computational 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 limit the performance of very deep models. The ResNet model has brought a new innovation that enabled much deeper models. The



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

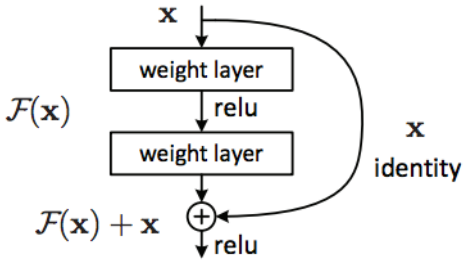


Figure 5.30: Residual block
8

ResNet architecture introduces the residual block (Figure 5.30). 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.

Using the residual block the ResNet model can achieve improved performance by using a deep architecture. In Figure 5.31 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 as 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.

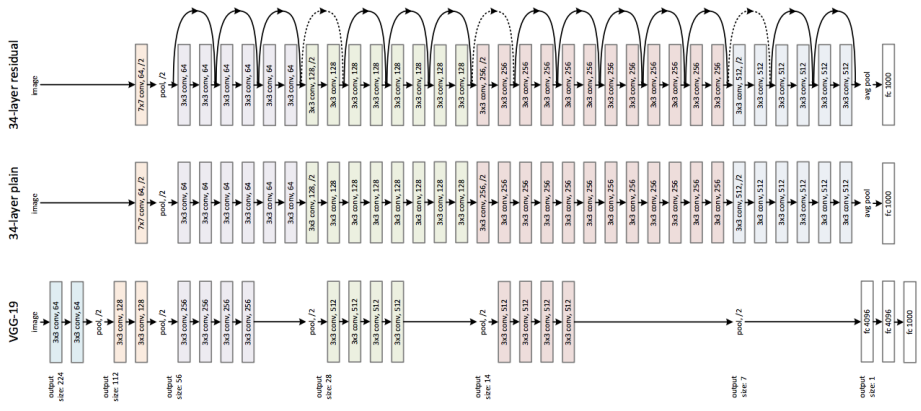


Figure 5.31: Residual Network
9

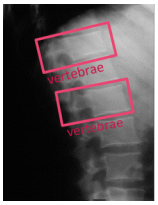


Figure 5.32: Localization of objects

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 an 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.

5.3.3 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 5.32). 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 the 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 does regression and one output that does classification.

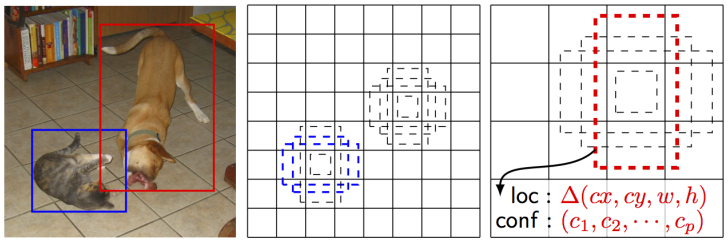
Question: How can we use these losses to deal with a shortage or surplus of bounding boxes?

State-of-the-art CNN localization

The "You only look once" (YOLO) architecture[10] develops a solution for image localization by splitting the image into 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 5.34.

[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.



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

Figure 5.33: YOLO - you only look once

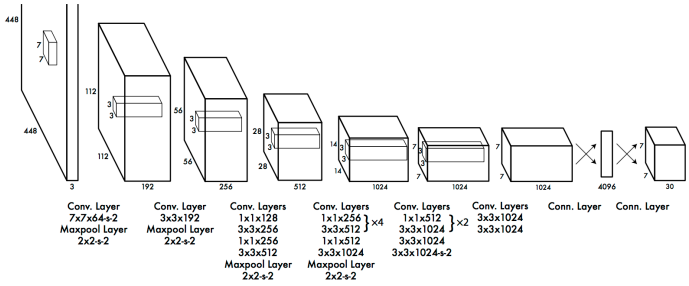


Figure 5.34: YOLO - you only look once

5.3.4 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 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 5.35.

State-of-the-art CNN for segmentation

One highly successful architecture of image segmentation is UNet (Figure 5.37). 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 5.36. The different organs are segmented in the image. The

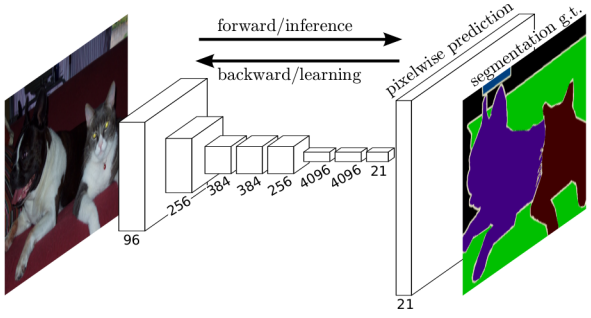


Figure 5.35: Image Segmentation

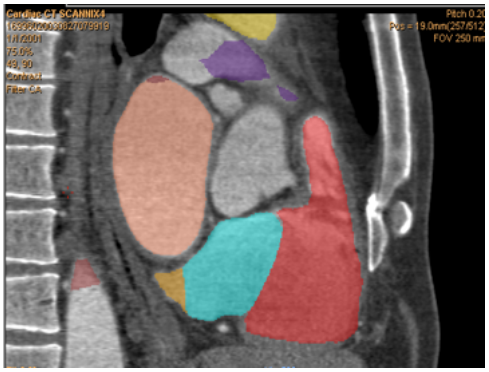


Figure 5.36: Image Segmentation

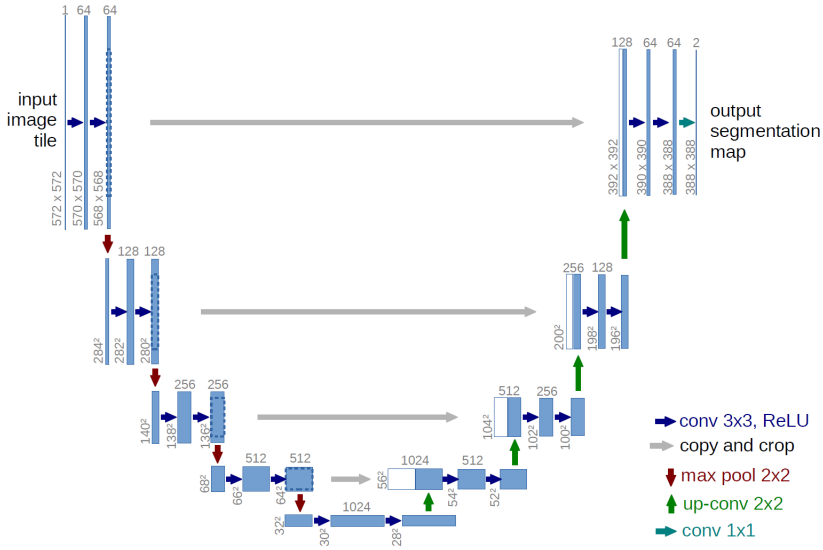


Figure 5.37: UNet CNN for image segmentation: Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015.

type of the object is needed to be determined from the global human anatomy present in 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 iteratively bringing the information into a more compact representation. This series of compactifications 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 are being mixed the high-level information.

5.3.5 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

Figure 5.38: Input image → output (transformed) image (Gatys, Leon A., et al. "A neural algorithm of artistic style. arXiv 2015." arXiv preprint arXiv:1508.06576 (2015).)

convolutional model introduced before. The concept of introducing noise to the input data and training a model to reconstruct the original 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 into more detail about such applications here, but we introduce one architecture that should illustrate the broad range of possibilities.

In the work of Gatys et al. a model is presented that can copy the artistic style of a given image and apply it to another image (Figure 5.38). This method is a good illustration of how a CNN model can be used in a task where we need to generate an image.

5.4 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 a 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 5.2.6 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 then 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?*

[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.

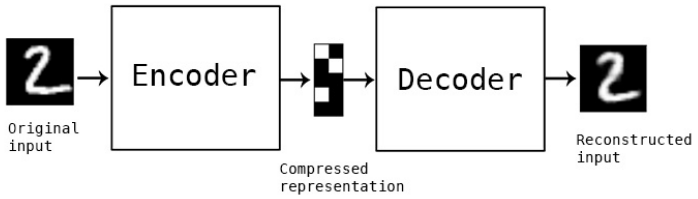


Figure 5.39: A conceptual representation of an autoencoder (https://blog.keras.io/building-autoencoders-in-keras.html)

5.5 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 5.4 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.

5.5.1 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 data points. Even for such a scenario, there are tasks for which need a 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 data points is more meaningful (but also more efficient) than 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 an unsupervised setting that we can transfer in a supervised setting and therefore improve the efficiency of the supervised training specifically with respect to the number of annotations.

Such representations 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 through some kind of information bottleneck. Specifically, an autoencoder consists of two parts: an encoder f and a decoder g (fig. 5.39) . These two parts are trained such that the decoder is an approximate inverse of the encoder on the target data, i.e.

$$g(f(x)) = x \tag{5.25}$$

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

$$h = f(x), \tag{5.26}$$

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. 5.40. A convolutional version of an autoencoder also inclused convolutional layers fig. 5.41.

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.

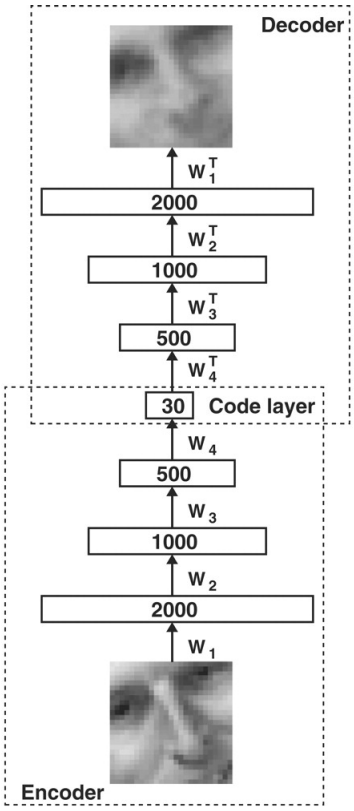


Figure 5.40: MLP Autoencoder: Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

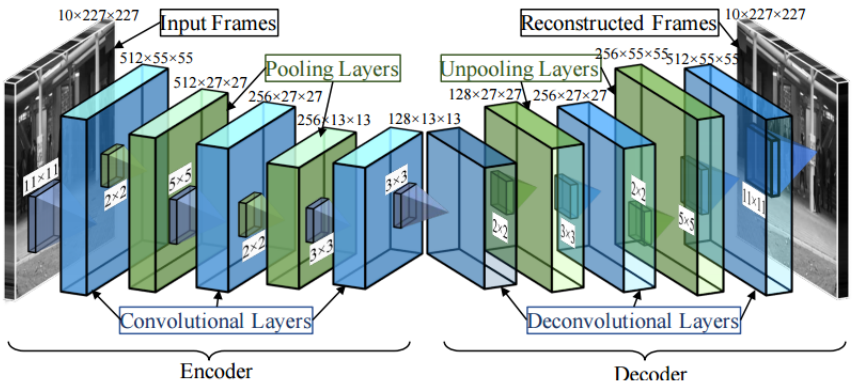


Figure 5.41: Convolutional Autoencoder

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 model reduces the dimensionality of the input to the dimensionality of the representation layer. Note: In the chapter on generative models we discuss 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

$$\text{Loss}(x, g, f) = L(x, g(f(x)) + \Omega(f(x)) = \|x - g(f(x))\|^2 + \Omega(f(x)) \tag{5.27}$$

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:

$$\Omega(h) = \sum_i |h_i|. \tag{5.28}$$

De-noising Autoencoder

Another way to avoid learning the identify function is to introduce noise to the input — see Figure 5.42. 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

$$L(x, g(f(n(x))). \tag{5.29}$$

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.

5.5.2 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

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

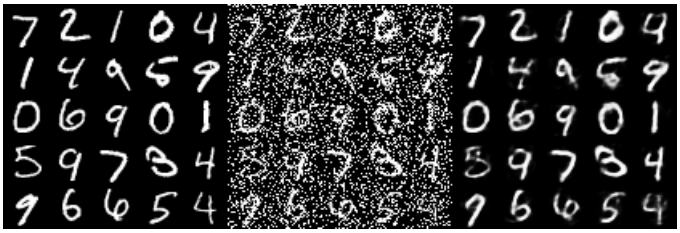


Figure 5.42: De-noising Autoencoder - Bengio, Yoshua, et al. "Generalized denoising auto-encoders as generative models." Advances in Neural Information Processing Systems. 2013.

Figure 5.43: One-shot learning

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 detail the methods for these 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 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.

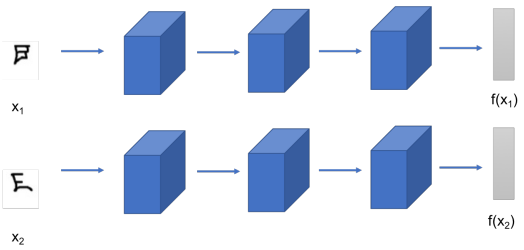


Figure 5.44: Siamese Network: Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." Proceedings of the IEEE conference on computer vision and pattern recognition. 2014.

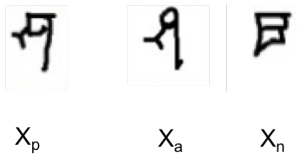


Figure 5.45: Triplet Network[15]

In this setup, the model has a better chance to develop features that distinguish the 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

$$d(I^{(1)}_{c=1}, I^{(2)}_{c=1}) < \tau \tag{5.30}$$

and for images of different classes we have

$$d(I^{(1)}_{c=1}, I^{(2)}_{c\neq 1}) > \tau. \tag{5.31}$$

One method to develop such a metric is based on the Siamese network model — see Figure 5.44. Here we learn a distance metric d of the form

$$d(x_1, x_2) = \|f(x_1) - f(x_2)\|^2 \tag{5.32}$$

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

[13] To achieve 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.

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

Triplet Network

The idea behind the triplet network — see Figure 5.45 — 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 than that between the negative input and the anchor. I.e.

$$d(x_p, x_a) \leq d(x_n, x_a) \tag{5.33}$$

or in terms of a learned metric (5.32)

$$\|f(x_p) - f(x_a)\|^2 \leq \|f(x_n) - f(x_a)\|^2 . \tag{5.34}$$

We can rewrite this as

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 \leq 0. \tag{5.35}$$

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

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0. \tag{5.36}$$

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

$$L(x_p, x_n, x_a) = \max(\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha, 0). \tag{5.37}$$

[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 data points 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:

$$\|f(x_p) - f(x_a)\|^2 - \|f(x_n) - f(x_a)\|^2 + \alpha \leq 0. \tag{5.38}$$

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

$$\|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2 \tag{5.39}$$

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

$$\|f(x_p) - f(x_a)\|^2 < \|f(x_n) - f(x_a)\|^2 < \|f(x_p) - f(x_a)\|^2 + \alpha \tag{5.40}$$

[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.

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 5.46.

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, selects relatively more informative triples as well as avoid model collapsing.

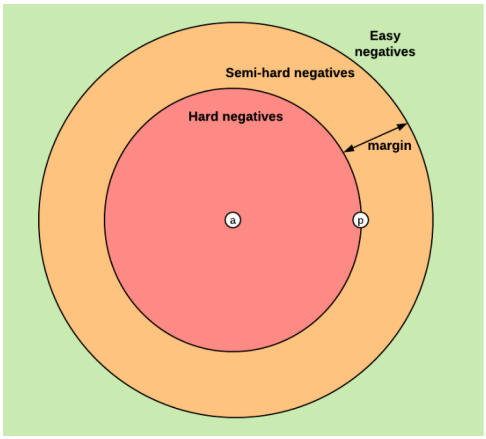


Figure 5.46: The three types of negatives, given an anchor and a positive

Chapter 6

Models for Graph data

The learning outcomes of this chapter:

- Learn to recognize the symmetries of graph and set data
- Learn how to develop Machine Learning problem formulations for graph and set data
- You are able to design models for graph data using graph neural networks
- You are able to design models for set data using DeepSets

6.1 Introduction

In this chapter, we study graphs and sets as data structures and the corresponding models that are suitable for processing such data.

A *graph* is an abstract mathematical structure that expresses the relationships between objects (Figure 6.1). Formally, a graph $\mathcal{G}$ consists of a finite set of nodes (also called vertices) $\mathcal{V}$ and a set of either ordered or unordered pairs of vertices referred to as edges $\mathcal{E}$:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$
$$\mathcal{E} \subseteq \{(i, j) \mid i, j \in \mathcal{V}\}$$

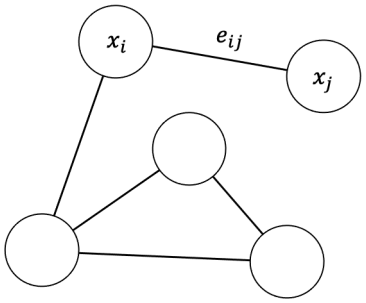


Figure 6.1: A depiction of a small graph

In contrast to this, the *set* structure does not contain edges as there are no relationships between the objects in the set beyond the fact that they belong to the same set.

As a data structure, graphs are highly expressive. They can be used to develop a representation of a broad range of systems. For example, a social network of people can be represented by a graph, where individuals are represented as nodes and their relationships as the edges between

the nodes. Graphs are commonly used as representations in applications such as telecommunication networks, biological networks, protein-protein interactions, the brain connectome, and many more. There are many variations in the definition of a graph. These are typically designed to align with a given problem formulation. For example, in the graph representation for the social network, we could add node properties to store information about each person such as their age or gender. Or we could encode information about the type or length of the relationships between people as information on the edges of the graph. Graphs can and have been extended in many ways. Beyond just adding properties to the edges and nodes, graphs can have edges between multiple nodes, or there could be different types of edges and nodes. Going forward, we use as a base structure an undirected (simple) graph with properties only on the nodes and explicitly state any extensions to this base structure.

The set structure has no notion of ordering. For example, given $A = \{1, 2, 3\}$ and $B = \{2, 1, 3\}$, A and B are exactly the same set. As a graph is defined as a combination of a set of nodes and a set of edges, there is no intrinsic ordering to a graph domain either. However, in order to store graph data in an array in computer memory so that it can be processed by neural networks, we are forced to choose an order to store the elements. Still, we would like a neural network architecture that operates on the graph data to be agnostic to this ordering, so that any ordering we happened to store the data in would still result in the same outcome. As such, the neural network needs to be symmetric to *permutations* of the ordering in which the data is stored.

More formally, assume we have a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and let each node $v \in \mathcal{V}$ have some associated node features $x_v \in \mathbb{R}^d$. To store this data in computer memory, all x_v can be stored in a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, where $n = |\mathcal{V}|$, and $\mathcal{E}$ is stored as an $n \times n$ adjacency matrix $\mathbf{A}$, such that $a_{ij} = 1$ if $(i, j) \in \mathcal{E}$, and 0 otherwise. We can then say that a function f, for example, a neural network, is permutation invariant if for any permutation matrix $\mathbf{P}$ it holds that

$$f\left(\mathbf{PX}, \mathbf{PAP}^T\right) = f\left(\mathbf{X}, \mathbf{A}\right). \tag{6.1}$$

Similarly, a function $\mathbf{f}$ that maps each node to output $y \in \mathbb{R}^{d_y}$ is equivariant if

$$\mathbf{f}\left(\mathbf{PX}, \mathbf{PAP}^T\right) = \mathbf{Pf}\left(\mathbf{X}, \mathbf{A}\right). \tag{6.2}$$

In order to design neural networks that respect permutation symmetry, we can use the geometric deep learning blueprint (Section 4.4) by concatenating equivariant layers, element-wise activation functions, and local and global pooling layers. The key challenge here is to develop the equivariant layers, of which we will see examples later in this chapter.

6.2 Machine Learning on graph data

As with other data structures, a broad range of problems can be solved by running computations on graph data. You can think of summarizing information of a node and its neighborhood, searching for similar nodes or subgraphs in a large graph, or any number of transformations where graph data is converted into a graph or another data structure. As with other tasks we have seen so far, there are many cases where an explicit set of algorithmic steps for performing a task is hard to define, and where a data-driven formulation is well motivated. This chapter discusses the use of Machine Learning on graph data and specifically in cases where the data is high dimensional and where traditional feature extraction on graph data may be difficult. Towards this goal, as in all preceding chapters we look at neural network-based models that can learn representations of the data automatically.

Let us take the example of a Recommender System. This is a system that has two types of data objects, users and items. Users interact with certain items and the systems' role is to identify items that would be of interest to the users and recommend them. For example, in an online video streaming service, the users interact with the videos. The systems' role is to identify the preferences of the users such that it can deliver useful recommendations. You can think of numerous such examples as books, clothing, and travel destinations. One of the advantages

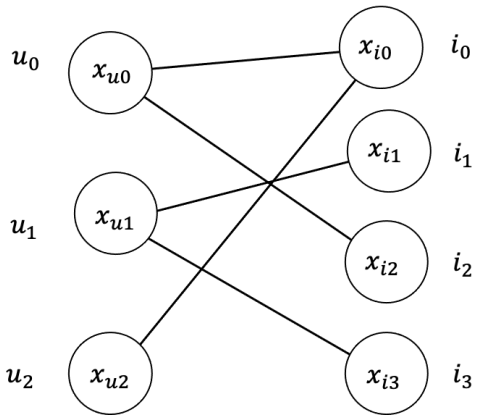


Figure 6.2: Graph structure for recommender system

of using a graph, in this case, is that we can encode with one data structure both the content information (user profiles, item descriptions) and the relationships between them. To achieve this, a graph is defined with two types of nodes. One for the users and one for the items (Figure 6.2). As such, all information about the user profiles can be encoded in the user nodes and all information about the items in the item nodes. The interactions between the users and the nodes can be expressed in the edges between the user and item nodes.

The achieve the goal of delivering useful recommendations, recommender systems need to develop an accurate representation of their users. These representations are then used to find similar users and provide recommendations based on how other users have interacted with the items. In the example of an online video streaming service, the target user would receive recommendations for videos that other users have watched who have also watched similar videos as the ones the target user has watched. These representations are built based on the user profile and the user-item interactions.

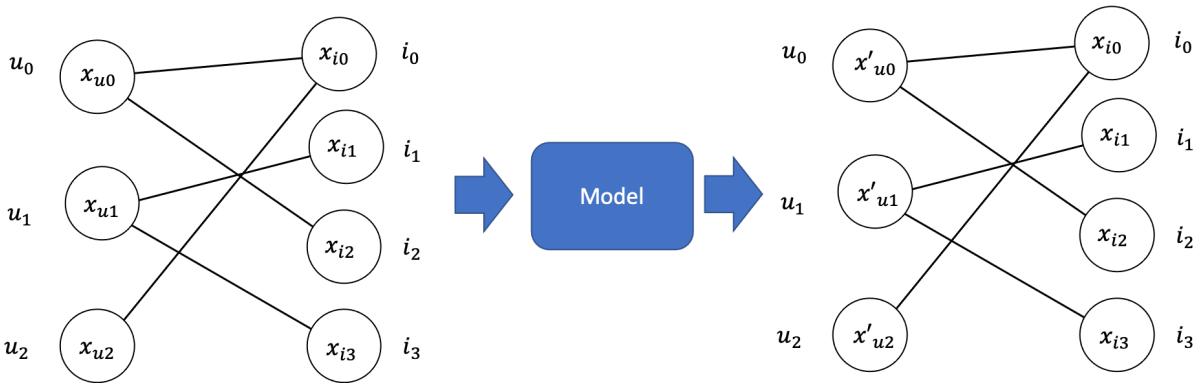


Figure 6.3: Graph recommender system using Machine Learning

Such user-item recommender systems have been studied before and many solutions exist. Some solutions are solely focused on the interactions between the users and the items without taking into account the content (i.e. properties of the users or items). Some use a combination of both

content and interactions. Similar to many other domains, one can carefully design features and algorithms that may solve a specific data-rich problem effectively. However, in a growing number of cases, a Machine Learning approach can address challenges such as scalability and improve accuracy by developing better representations of the data (Figure 6.3).

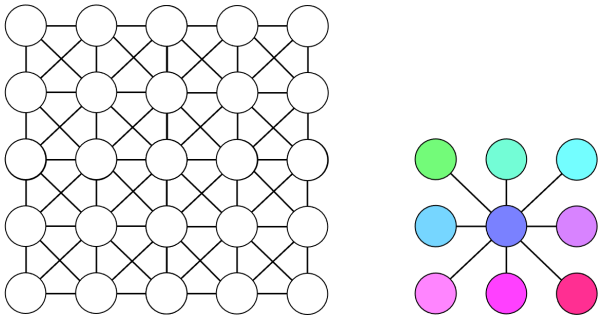
The family of models that are suitable for such a task and which we further discuss in this chapter is referred to as Graph Neural Networks (GNN).

6.3 Parameter re-use on graphs

To learn efficient representations of graph data we need to be able to utilize the structure of the data in our model. As an analogy, we can use the convolutional layer that operates on spatially distributed data (e.g. 2D images, 3D images, 1D signals) and learns localized correlations. This machinery allows us to develop feature detectors for these localized patterns. Such feature detectors make the model highly efficient as they can be reused on different spatial locations in the data. In other words, our model as such was able to utilize the structure in the data to learn efficient representations.

In chapter 5, we considered reusing parameters for data on grids as a way to incorporate equivariance in our model. When looking at image data for the grid domain, we think of features as pixels and their pixel neighborhoods. Analogously, on the grid domain, we can think of features as nodes and their neighborhood of connected nodes.

The goal of our model is then to build representations of nodes, given their properties and the properties of their neighbors.



(a) Image as a graph

(b) Specially localized feature

Figure 6.4: A graph representation of image data

However, nodes in a grid and nodes in a graph have many differences. The nodes of the grid have a fixed number of neighbors based on their location on the grid. The neighborhood of the node in the graph is determined by its edges. The graph (in general) does not express information about the geometry or the distances[1] of each node in terms of a fixed coordinate system. So, a graph is invariant (i.e. does not change) to any transformation that does not change the topology of the graph (i.e. which nodes are connected to each other and how). Any changes to the ordering of the nodes[2] or any other spatial representation that does not have

[1] One can design a graph with edge properties that expresses such distance.

[2] Again, this holds for our base type of graph. Graphs can be simply extended to incorporate the ordering of nodes

an effect on the graph. To utilize the properties of the graph data structure, our model should perceive two graphs where the order of the nodes is changed as equal.

> Note: You can contrast Figure 6.1 and Figure 6.4. In the image data given in Figure 6.4 (Left) we can use the regular structure to develop feature detectors for localized patterns as given in Figure 6.4 (Right). However, for graph data, such spatial regularity is not present. So, the localized feature detectors need to take a different form.

In principle, for the GNN to utilize the properties of graph data, it should learn to form features related to the properties of a node and its neighborhood. It should be able to develop these features in any node in the graph regardless of how many edges it has. Similarly, many GNN models need to be able to process a variable number of nodes. To achieve permutational symmetry, following the principle of parameter re-use, the GNN layer re-uses its feature detectors on each node and each edge in the graph. Such layer can then be used as a component of models that operate on graph data. There are a number of approaches that develop GNN models that meet these goals. In the rest of this chapter, we will delve into a number of them through the lens of the Neural Message Passing framework.

6.4 Neural Message Passing

Neural Message Passing algorithm is a framework that formalizes a number of GNN models and a solid foundation for conceptualizing such models [2]. The algorithm defines a notion of a message that a node sends to its neighbors. These messages allow the information from the nodes to be propagated through the graph and hence node representations to be developed.

The algorithm consists of the following steps:

- Emit a message from each node consisting of the content of each node.
- The messages from each node are processed at the edges in the graph using the *edge transfer function*, and the output reaches the neighboring nodes.
- The messages are then aggregated at each node using the *aggregation function*.
- The new node value is computed by the *node update function* which combines the old node value and the value of the aggregation.

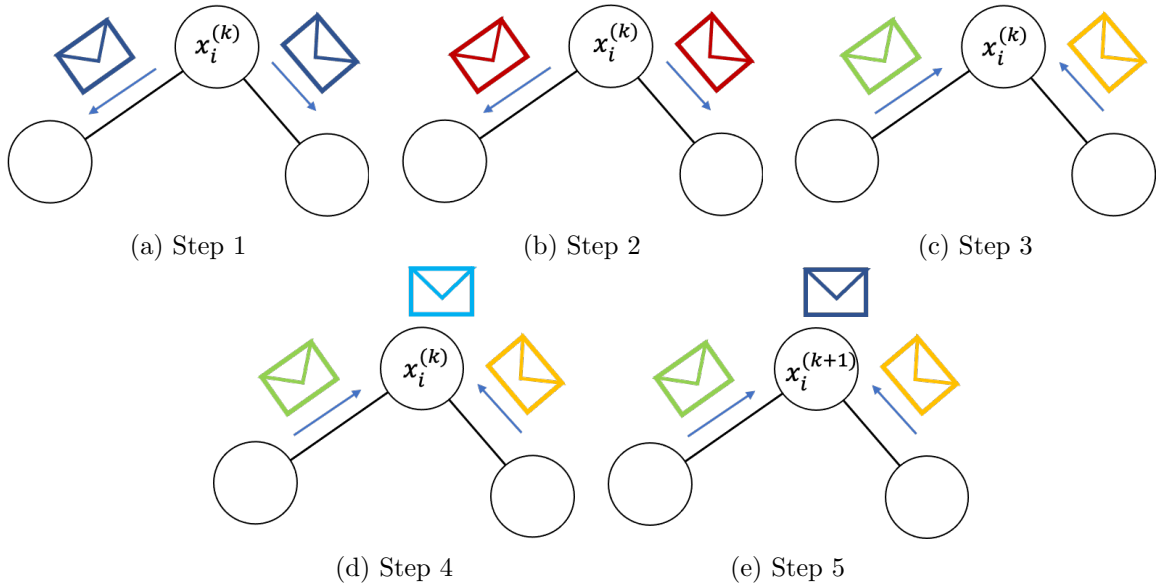
Based on the task at hand, the model may also include a *node read-out function* or a *graph read-out function* that transforms the node representation into a node output or representation of all the nodes in the graph into a graph output.

Given a representation (properties) x_i of a node i in a graph. The update rule of the message passing algorithm from iteration $k-1$ to iteration k is given as:

$$x_i^{(k)} = \gamma^{(k)}(x_i^{(k-1)}, \cup_{j \in \mathcal{N}(i)} \phi^{(k)}(x_i^{(k-1)}, x_j^{(k-1)}, e_{j,i})) \tag{6.3}$$

where ϕ is the *edge transfer function*, $\cup$ is the *aggregation function* on the node, γ is the *node update function*, $e_{j,i}$ is the edge from node j to node i, and $\mathcal{N}(i)$ is the neighborhood of node i. These three functions are differentiable functions, such as an MLP.

The algorithm is iterative and in each iteration, the representation of a node incorporates a larger and larger neighborhood. In the first step, the message of the neighbors of node i con-



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5

Figure 6.5: Neural Message Passing algorithm

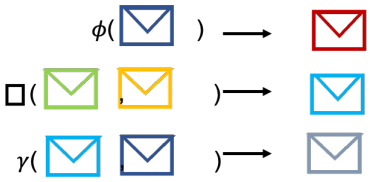


Figure 6.6: Neural Message Passing functions

tains information only about their properties. In the second step, the messages also contain information from their neighbors and so on. You can think of this as applying multiple convolutional layers to an image. When convolving over the activation map on the first layer, the second layer combines information from a larger neighborhood. So, GNN achieves depth by implementing multiple iterations. As you can see in the update rule given above, the message passing function ϕ and the node update function γ are also indexed by k. This allows the GNN to use different parameters (read: you can think of these as functions as layers or even MLP) at different iteration steps k.

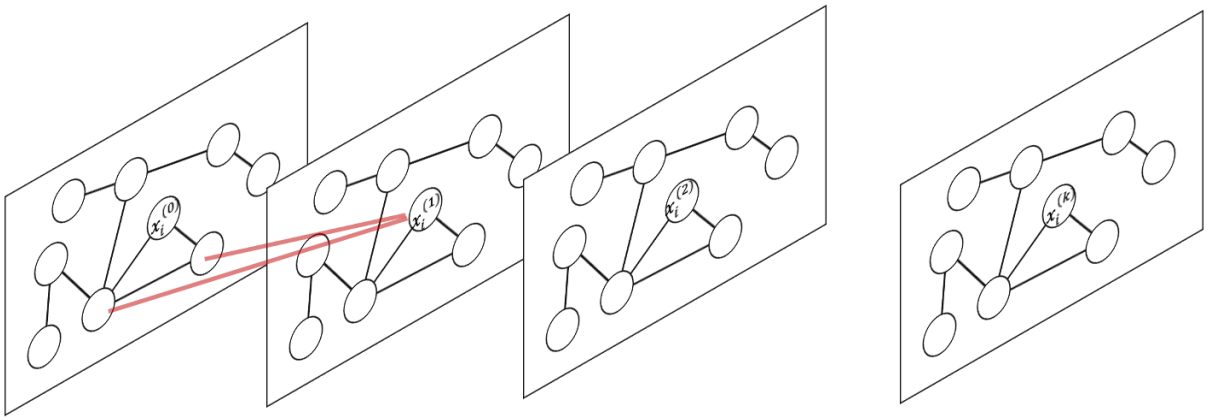
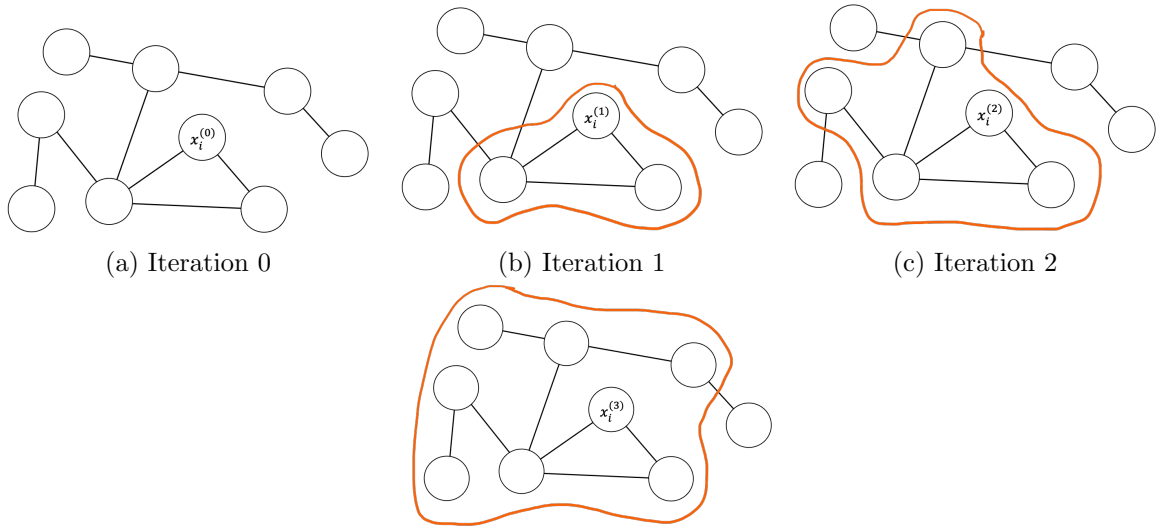


Figure 6.7: Neural Message Passing functions

In order for neural message passing layers to be eligible for designing invariant or equivariant networks, we need the layer to be equivariant to permutations. Indeed, this is the case, as the

node update function is re-used on each node. So, permuting the nodes would result in an equivariant permutation of the updated node values. Moreover, as there is no ordering in the neighborhood of each node, the node update function needs to be invariant to the order of the nodes. This is achieved by using a permutationally invariant aggregation function $\cup$.



(a) Iteration 0 (b) Iteration 1 (c) Iteration 2

(d) Iteration 3

Figure 6.8: Neural Message Passing algorithm size of neighborhood after each iteration

6.5 GNN architectures

A number of GNN architectures have been proposed in the literature that can be formulated through the neural message passing framework. Each of these proposes a variety of choices for the edge transfer function, the node aggregation, the node update function, and the node (or graph) readout function. All of these choices as well as the variations in the graph structure are typically developed in the context of a particular task on graph data. To get a better intuition for the type of tasks on graph data, let us look at some concrete examples.

Let us first take a look at the Cora dataset [3]. This is a dataset consisting of papers, which are represented by nodes in a graph, and citations between the papers, which are represented by directed edges. One task on this dataset is to recognize the topic of each paper based on the papers it cites and its content. For this task, some of the papers are labeled with a topic from a set of topics. The goal is to assign topics to the nodes that are not labeled. This problem can be defined as node classification. Specifically, based on each node's and its neighbors' properties a GNN model will produce a probability distribution over the set of possible topics. Based on this, we know that our node readout function needs to be a softmax.

Note that for this task, the whole Cora dataset is represented with a single graph. As such, all of the nodes are available during training, while only the ones that are labeled can be used to compute the loss. Furthermore, in this task, it is expected that the graph evolves over time as new papers are added to the Cora dataset. In other words, our model is required to work on a node level, as the graph representing the dataset can change over time both in the node properties as well as the topology. This formulation where both labeled - training and unlabelled - testing data is available during the training phase has a resemblance to the semi-supervised formulation of machine learning.

Let us contrast this with another example. The dataset consists of a number of graphs. Each graph represents a molecule where the nodes represent the atoms, and the edges represent bonds between the atoms. One task on this data is to predict the biological functions of molecules based on these graphs. This task can be formulated as graph classification. In this case, a GNN model needs to develop a representation of the whole graph and use that representation to produce a probability distribution over a fixed set of biological functions. Here a training set would consist of a variety of graphs of known molecules and their biological functions as the labels. A test set would then consist of molecules the model hasn't seen yet (with appropriate labels) and the final goal would be to have the model predict what the biological function is or would be, of molecules we ourselves do not yet know the function of. This formulation is more synonymous with the supervised learning formulation where the unlabelled data - test data is not available during training.

These examples are just two out of a vast number of formulations that can be developed on graph data. Nevertheless, they give us a certain amount of context in which we can discuss different GNN architectures.

6.5.1 GraphSAGE

The GraphSAGE (SAmple and aggreGatE) model [4] is proposed as a solution for the task of classifying scientific publications using the Cora dataset discussed above. The authors specifically propose this model as a flexible solution that can scale well to a very large graph dataset and deal with changes in the graph topology and content.

The GraphSAGE is a GNN that uses the identity function as an edge transfer function. The aggregation function is given as a set of choices consisting of:

- computing the mean of all incoming messages
- LSTM model processing a variable-length list of neighboring representations. The LSTM model is not permutationally invariant as it can use the order in the sequence as information to process a sequence to deal with them during training the messages from the neighbors of a node are randomly permuted.
- a non-linear transformation using a dense layer of a neural network for each message followed by a maximum aggregation function.

If we select the aggregate function to be mean we get the following expression for the node update:

$$x_i^{(k)} = W_1 x_i^{(k-1)} + \frac{1}{|\mathcal{N}(i)|} \sum_{j \in \mathcal{N}(i)} x_j^{(k-1)}$$

In this method, the authors also introduced an additional loss term in the loss that incorporates domain knowledge. Specifically, adjacent nodes should have a similar representation[3]. This is very specific to this application, however, one can imaging that such terms in different applications can also be very useful.

6.5.2 Gated GNN

The Gated GNN model [5], introduces a GRU as the update function. Taking multiple message-passing steps allows the node representation to combine information from a wider neighborhood of nodes. In general a GNN model as specified Equation (6.3) has distinctive parameters for

[3]for full details on the solution please go to [4]

each update step (read: the update function ϕ is indexed by k the iteration number). In this method, the node updates are considered as a sequence and as such an RNN model is used that can re-use its parameters over the course of the sequence.

$$x_i^{(k)} = GRU(x_i^{(k-1)}, \sum_{j \in \mathcal{N}(i)} e_{j,i} W x_j^{(k-1)})$$

Where $e_{j,i}$ denotes an edge weight of an edge between nodes j and i, and W a set of learnable parameters of the model's linear edge transfer function.

6.5.3 Graph Attention Networks

Due to the innate property of graph data to have an arbitrary number of (unordered) edges between the nodes GNN architectures typically incorporate an aggregation function that is invariant to permutations of the order of the edges. This limits the possibility to develop specific feature detectors for different size neighborhoods or for specific topologies of the graph.

In models for sequential data, more specifically in seq2seq models, the attention mechanism was introduced to alleviate the bottleneck of the encoder-decoder architecture and allow the decoder to look back at the input sequence when computing its output. In this case, the attention weights are computed based on the current state of the decoder and the sequence of hidden states of the encoder. A modification of the attention mechanism referred to as *self-attention* as introduced in [6] where the attention is computed without the decoder mechanism. In this case, the attention is computed using different parts of the sequence to refer back to the whole sequence. This allowed for a number of computational advantages in modeling sequences.

Such a solution based on this mechanism named Graph Attention Networks (GAT) was introduced in [7] to enable GNN models to generalized the aggregation to learn specific feature detectors based on the size and the content of the neighborhood of a node. The GAT model 'pays a specific amount of attention to each incoming message based on the content of that message and the content of all other incoming messages. This self-attention mechanism differs from the original formulation of self-attention in that the mechanism is permutationally invariant to the order of the messages.

The update computation in the GAT is as follows:

$$x_i^{(k)} = \alpha_{i,i} W x_i^{(k-1)} + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} W x_j^{(k-1)}$$

where the $\alpha_{i,j}$ is the attention coefficient and it is computed as:

$$\alpha_{i,j} = \frac{e^{a(x_i, x_j)}}{\sum_{l \in \mathcal{N}(i)} \alpha_{i,l} e^{a(x_i, x_l)}}$$

and a is given as:

$$a(x_i, x_j) = \sigma(W_a[W x_i, W x_j])$$

where $[,]$ is a concatenation operator and W_a and W are learnable parameters.

With this mechanism first a linear edge transfer is applied with the parameters W. Then attention is computed, where each node can attend to each other node. The attention coefficient is computed by a. These attention coefficients are normalized by the softmax computation in α.

6.5.4 Spectral Graph Neural Networks

Another class of GNN models takes a different perspective from the Neural Information Passing and works with a spectral representation of the graph [8, 9]. These methods work on the full graph and the models can be defined as:

$$f(X, A)$$

where X is a matrix of node feature vectors and A is the adjacency matrix of the graph. An adjacency matrix is a square matrix that represents the edges of the graph. In a simple graph, an element in row i and column j of the adjacency matrix has a value of 1 when there exists an edge between nodes i and j in the graph. The adjacency matrix can also represent the direction and weights of the edges.

These models develop a generalization of the convolutional operator for non-Euclidean data. Nevertheless, as these models are trained on a particular graph structure they cannot be directly applied to graphs with different structures or to graphs with evolving structures.

Appendix A

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.

A.1 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:

$$\Omega = \{\omega = (\omega_1, \ldots, \omega_7) \mid \omega_1, \ldots, \omega_7 \in \{1, \ldots, 6\}\} \tag{A.1}$$

On top of this, we have a set of **events**, often denoted by $\mathcal{F} \subseteq 2^\Omega$, 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

$$\{\omega \in \Omega \mid \omega_2 = 3\} \in \mathcal{F}. \tag{A.2}$$

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

$$P : \mathcal{F} \to [0, 1] \tag{A.3}$$

$$P : A \mapsto \frac{\#A}{6^7}, \tag{A.4}$$

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

$$\Omega = [0, 1] \times [0, 1] \tag{A.5}$$

and our events are regions of the square[1]where the rain drop might fall

$$\mathcal{F} \subseteq 2^{\Omega}, \tag{A.6}$$

with the probability of an event being the area of that region

$$P(A) = \text{area}(A). \tag{A.7}$$

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[2], and all calculations are done using *random variables*.

A **random variable**, X is a measurable function

$$X : \Omega \to \mathbb{R}, \tag{A.8}$$

or more generally

$$\mathbf{X} : \Omega \to \mathbb{R}^n. \tag{A.9}$$

Such a random variable comes with its own events of the form[3]

$$\{X \in A\} = X^{-1}(A) = \{\omega \in \Omega \mid X(\omega) \in A\} \tag{A.10}$$

and with its own probability measure on its range:

$$P_X = P_{\#}X \ : \ A \mapsto P(X^{-1}(A)). \tag{A.11}$$

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

$$f_X : R \to [0, 1] : r \mapsto P(X = r). \tag{A.12}$$

This function, f_X is called the **probability mass function (PMF)** of X.

[1]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.

[2]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 $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$.

[3]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.

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**[4] **(PDF)**, $f_X \geq 0$, so that

$$P(X \in A) = \int_A f_X(x)\mathrm{d}x. \tag{A.13}$$

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

$$F_X : x \mapsto P(X \leq x). \tag{A.14}$$

For continuous random variables, the PDF is the derivative of the CDF.

A.2 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

$$P(A \mid B) = \frac{P(A \cap B)}{B}. \tag{A.15}$$

In the example above, the contagiousness of the flue makes that

$$P(X_1 = 1 \mid X_2 = 1) > P(X_1 = 1). \tag{A.16}$$

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

$$P(A \mid B) = A, \tag{A.17}$$

or equivalently[5]

$$P(A \cap B) = P(A) \cdot P(B), \tag{A.18}$$

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.

$$X \perp Y \iff \forall_{A,B \text{ measurable}} \ : \ X^{-1}(A) \perp Y^{-1}(B). \tag{A.19}$$

[4] 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.

An important theorem for dealing with conditional probabilities is **Bayes' theorem**, which says that for any two events, A and B, we have

$$P(B \mid A) = \frac{P(A \mid B)P(B)}{P(A)}. \tag{A.20}$$

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

$$\mathbf{X} \ : \ \Omega \to \mathbb{R}^n \ : \ \omega \mapsto (X_1(\omega), \ldots, X_n(\omega)). \tag{A.21}$$

This **random vector** again has associated events of the form

$$\{\mathbf{X} \in A\} = \mathbf{X}^{-1}(A) = \bigcap_{i=1}^{n} X_i^{-1}(A), \tag{A.22}$$

and an associated probability measure on $\mathbb{R}$ given by

$$P_{\mathbf{X}} = \mathbf{X}_{\#}P \ : \ A \mapsto P(\mathbf{X}^{-1}(A)). \tag{A.23}$$

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

$$P(\mathbf{X} \in A) = \int_A f_{\mathbf{X}}(x_1, \ldots, x_n) \mathrm{d}x_1 \cdots \mathrm{d}x_n. \tag{A.24}$$

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:

$$f_X(x) = \int_{\mathbb{R}} f_{X,Y}(x, y) \mathrm{d}y. \tag{A.25}$$

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

$$f_{Y|X=x}(y) = f_{Y|X}(y \mid x) = \frac{f_{X,Y}(x, y)}{f_X(x)}. \tag{A.26}$$

Note that this means we can **factorize** the density of the joint distribution as

$$f_{X,Y}(x, y) = f_X(x) f_{Y|X}(y \mid x) \tag{A.27}$$

$$= f_Y(y) f_{X|Y}(x \mid y). \tag{A.28}$$

[5]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.

A.3 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

$$\mathbb{E}[X] = \sum_{s \in S} s \cdot f_X(s). \tag{A.29}$$

We generalize this to continuous random variables by replacing the sum by an integral, and the mass function by a density:

$$\mathbb{E}[X] = \int_S s \cdot f_X(s) \mathrm{d}s, \tag{A.30}$$

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

$$\mathbb{E}[g(X)] = \sum_{s \in S} g(s) \cdot f_X(s) \qquad \text{if } X \text{ is discrete,} \tag{A.31}$$

and

$$\mathbb{E}[g(X)] = \int_S g(s) \cdot f_X(s) \mathrm{d}s \qquad \text{if } X \text{ is continuous,} \tag{A.32}$$

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

$$g(\mathbb{E}[X]) \leq \mathbb{E}[g(X)], \tag{A.33}$$

and consequently if g is **concave** we get

$$g(\mathbb{E}[X]) \geq \mathbb{E}[g(X)]. \tag{A.34}$$

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.

Bibliography

[1] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," 2021.

[2] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning*, pp. 1263–1272, PMLR, 2017.

[3] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, "Automating the construction of internet portals with machine learning," *Information Retrieval*, vol. 3, no. 2, pp. 127–163, 2000.

[4] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 1025–1035, Curran Associates Inc., 2017.

[5] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[7] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[9] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.