Artificial Neural Networks

Prof. Dr. Sen Cheng

Winter Semester 2019/20

Contents

1 Introduction

1.1 Deep neural networks in the news

Artificial intelligence creates perfumes without being able to smell them

How quickly can AI solve a Rubik's Cube? In less time than it took you to read this headline.

Top AI researchers race to detect 'deepfake' videos: 'We are outgunned'

AI is more powerful than ever. How do we hold it accountable?

Warning! Everything Is Going Deep: 'The Age of Surveillance Capitalism'

1.2 What is a neural network?

An artificial neural network is a universal function approximator.

So this means that an ANN can be mathematically described as

$$y = f(x). \tag{1}$$

What is a function? A function is a relation that maps an element in the *domain A* onto a unique element in the *codomain B* (Fig. 1).

$$f : A \longrightarrow B \tag{2}$$

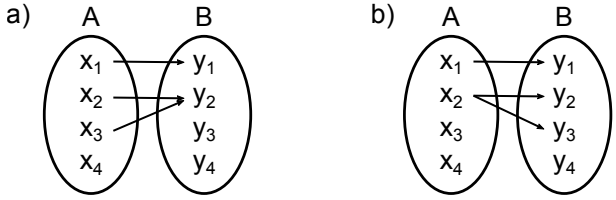


Figure 1: The mapping in a) is a function, the one in b) is not.

In real applications, relationships between two variables are almost never deterministic, i.e., there are unpredictable variations. We can model this with a *random variable* ε that doesn't indicate a fixed value, but a random value drawn from a *statistical distribution*. Distributions are characterized by their *probability density functions* (PDFs) (Fig. 2). With a random variable, we can model the uncertain relation between two variables as a *stochastic*

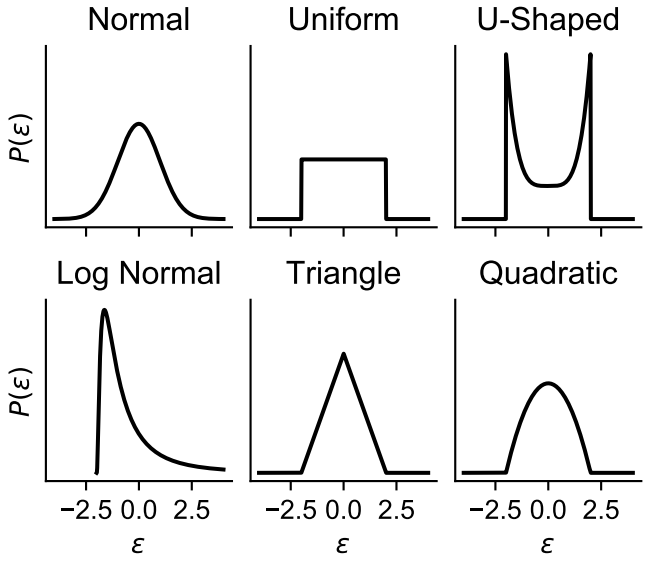


Figure 2: Different classes of distributions.

function

$$y = f(x) + \varepsilon. \tag{3}$$

ε is often called a *noise* term. Most commonly we assume that the noise is drawn from a *normal distribution* with mean μ and variance σ^2, a.k.a. *Gaussian distribution*, and denote this by

$$\varepsilon \sim N(\mu, \sigma^2). \tag{4}$$

Many real-world problems can be mapped onto the following mathematical problem: What function best describes the relationship between two variables in the data (X, Y)? Finding such a function is also called *fitting* a function to data. Only recently, it has been shown that ANN are particularly good function approximators. While this class is about ANN, it is important to understand the fundamental principles underlying function fitting. That's why we start with a very general statement of the problem and simple functions that have little to do with ANN.

1.3 Relationship between ANN and AI

artificial neural networks $\subset$ machine learning $\subset$ artificial intelligence (AI).

There are three fundamental paradigms in machine learning: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. This class focuses on supervised learning, however ANN are frequently used in the setting of unsupervised or reinforcement learning.

The literature on ANN is full of tricks and algorithms. We will try to bring out the underlying principles in the lectures. To this end a mathematical description is inevitable. The tutorials will deepen your understanding of the theoretical background by linking it to practical exercises and also enable you to apply the methods in practice.

1.4 Applications

Many real-world applications can be understood as function fitting.

- **Recognition**: Recognizing a visual object, a spoken word or other sensory stimuli is about mapping from a physical stimulus to a semantic label (Fig. 3).
- **Prediction**: predicting a future value (e.g. weather, stock prices, consumer demand, product failure, traffic) is a mapping from past values to future ones

$$(x_{t-a}, \ldots, x_t) \longrightarrow x_{t+b} \tag{5}$$

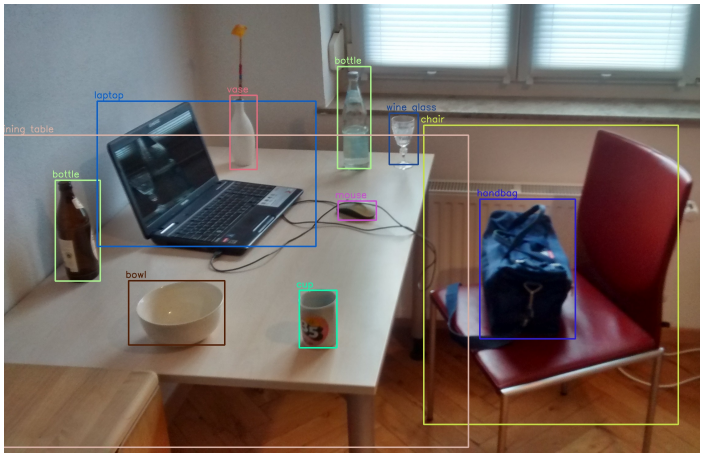
- **Recommender systems** (content-based filtering): mapping from content to like/dislike
- **Fraud/ Spam detection**: mapping from content to fraud/not-fraud or spam/ham (Fig. 3).
- and more

However, there are applications that cannot (easily) be modelled as function fitting.

- **Problem solving** (e.g. finding shortest path from A to B, playing games)
- **Recommender systems** (collaborative filtering)
- **Decision making** (avoiding accident in automated driving systems)
- and more

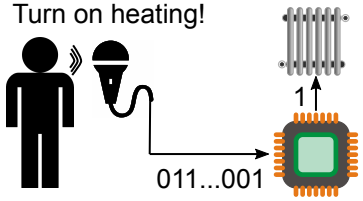
Note, even if a problem cannot be mapped onto function fitting, it turns out that ANN can be an essential component on a subtask, e.g. in deep reinforcement learning.

a)



Source: https://commons.wikimedia.org/wiki/
File:Detected-with-YOLO--Schreibtisch-mit-Objekten.jpg

b)



c)

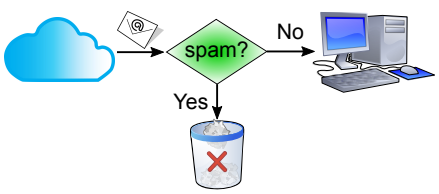


Figure 3: Examples of recognition tasks. a) Object recognition from an image of a scene. b) Speech recognition in the context of smart homes. c) Spam e-mail filtering.

1.5 Problem set

1. **Function**

   In the following you find two relations, f and g, that map x onto y. Determine which one is a function and why.

   (a) $f : x \to y$ such that $y = x^2$

   (b) $g : x \to y$ such that $y^2 = x$

   Also, plot y vs. x for both relationships in Python and try to visually determine which one is a function. Note that the range of x in your plot is limited and therefore you may not see the full characteristics of the functions, so choose the range carefully!

2. **Types of functions**

   Plot the following functions for $x \in [-10, 10]$. Be sure to plot them in a way that you can change the parameters easily:

   (a) $f(x) = 2x + 2$

   (b) $f(x) = 5x^2 + x$

   (c) $f(x) = 11x^3 + 2x^2 + 2x + 3$

   (d) $f(x) = e^x$

   Try the following adjustments:

   i. Take the linear function from above in its generalized form $f(x) = ax + b$. It has 2 parameters. Adjust each of them and plot the result. Observe how do they change the behavior of the function.

   ii. Take the quadratic function from above in its generalized form $f(x) = ax^2 + bx + c$. Can you tune the parameters in such a way that the minimum of the function lies at -2? What happens if you multiply the quadratic term with a large constant? Extra: can you explain why the linear factor does not contribute to the result?

2 Optimization problems

Fitting a function is an optimization problem and has the following components: data, model class, loss function, and optimization algorithm.

2.1 Data

Since we focus on supervised learning in this class, the data consists of *independent variables* and *dependent variables*. Variables can be continuous, binary or categorical. The independent variables can also be thought of as the inputs (of the function):

$$X = (x_1, x_2, \ldots, x_N) \tag{6}$$

Each x_i can be a vector, i.e., a sequence of numbers, x_{ij}, where $j = 1, \ldots, m$. This formulation is quite general and many types of information can be represented this way. For instance, an image can be represented as vector of numbers by dividing the image into $a \times b$ pixels (Fig. 4), calculating the average luminance in each pixel on a scale from 0 to 1, and then concatenating the b columns into a single vector of length ab. For future use, we define the mean as

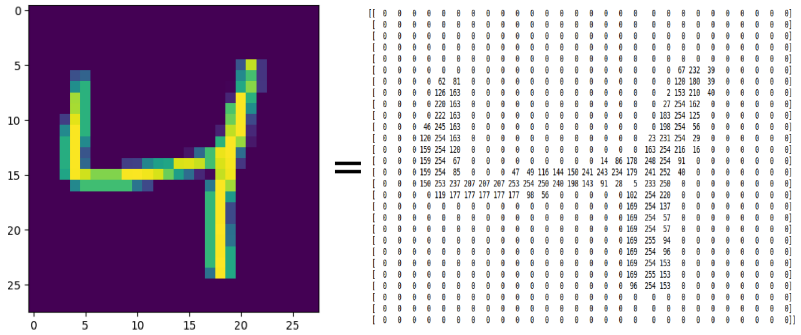


Figure 4: Representation of an image as vector. $18x18 = 324$ pixels or dimensions.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{7}$$

and variance

$$\sigma^2(X) = \frac{1}{N}\sum_{i=1}^{N} (x_i - \bar{x})^2 \tag{8}$$

The dependent variables are often thought of as the outputs (of the function):

$$Y = (y_1, y_2, \ldots, y_N) \tag{9}$$

Each y_i can be a vector, i.e., a sequence of numbers, y_{ij}, $j = 1, \ldots, n$.

2.2 Model class

The choice of the model class is key to function fitting. The success of the fitting depends on whether the model class is a good description of the data. For convenience, all parameters of the model class are summarized in the vector θ. The function is said to be *parametrized* and often denoted by f_θ.

Examples for functions:

class	**linear**	**polynomial**	**exponential**	**ANN**
$f(x)$	$mx+b$	$\sum a_n x^n$	$\exp(ax+b)$	$\cdots$
θ	(m,b)	$(a_0, \ldots, a_k)$	(a,b)	$\cdots$

2.3 Loss function

The *loss function* measures how well a function is describing the data. The lower the loss, the better.

$$L(\theta, X, Y) = \sum_{i=1}^{N} l\left(y_i, \hat{y}_i\right) \tag{10}$$

where i is the sample index and $\hat{y}_i = f_\theta(x_i)$.

A popular choice for the loss function is a class of functions called *l-norm* (or *Minkowski distance*), because for $k \geq 1$ these function can be used to measure distances:

$$l_k\left(|y_i - \hat{y}_i|\right) = \left(\sum_{j=1}^{n} |y_{ij} - \hat{y}_{ij}|^k\right)^{1/k}, \tag{11}$$

where j is the dimension index. The l_2-norm $\left(l_2 = \sqrt{\sum_j (y_{ij} - \hat{y}_{ij})^2}\right)$ is the most commonly used loss function. It is also known as the *Euclidian* norm or *Euclidian* distance. The l_1-norm $\left(l_1 = \sum_j |y_{ij} - \hat{y}_{ij}|\right)$ is also known as the *Manhattan* distance. In the limiting case $k \to \infty$,

$$l_\infty = \lim_{k\to\infty} \left(\sum_{j=1}^{n} |y_{ij} - \hat{y}_{ij}|^k\right)^{1/k} = \max_{j=1}^{n} |y_{ij} - \hat{y}_{ij}| \tag{12}$$

The choice of the loss function is important since it determines what it means for a function to describe the data well. In other words, the result of fitting strongly depends on the loss function. The loss function has to be appropriate for the task. For instance, the above loss functions work well for regression[1] type problems. We will encounter different types of loss functions for different tasks later.

2.4 Optimization

The objective of function fitting is to find $\hat{\theta}$ such that the total loss function Eq. 10 is minimal, i.e.,

$$\hat{\theta} = \arg\min_{\theta} L(\theta, X, Y). \tag{13}$$

Since the derivative at minima (and at maxima) vanish, we can find $\hat{\theta}$ by solving the equation

$$\frac{\mathrm{d}}{\mathrm{d}\theta} L(\theta, X, Y) = 0 \tag{14}$$

for θ. Note that Eq. 14 is a necessary condition, but not sufficient. The second derivative also has to be positive to ensure that we found a minimum. However, this step is often skipped since computing the second derivative is often very difficult. Fitting the parameters is often referred to as *training* the model, especially when the model is more complex. Equivalently, the model is sometimes said to *learn* from the data.

2.4.1 Analytical solution

In some simple cases Eq. 14 can be solved analytically. We will do this for linear regression in the next lecture.

2.4.2 Numerical solution: Newton's method

Most of the time, however, Eq. 14 cannot be solved analytically. We could use Newton's method to find the zeros of a function $g(x)$ numerically. Starting with a guess of the solution x_0, we approximate the function by a line going through the point $(x_0, g(x_0))$ (Fig. 5). We then find the root of that line

$$y = m x_1 + b \stackrel{!}{=} 0 \tag{15a}$$

$$x_1 = -\frac{b}{m} \tag{15b}$$

[1] We'll define this term later.

The y-offset b can be obtained by inserting the point into the equation for the line.

$$g(x_0) = g'(x_0)x_0 + b \tag{16a}$$
$$b = g(x_0) - g'(x_0)x_0 \tag{16b}$$

$$x_1 = -\frac{g(x_0) - g'(x_0)x_0}{g'(x_0)} \tag{17a}$$
$$= x_0 - \frac{g(x_0)}{g'(x_0)} \tag{17b}$$

This process is then iterated until it converges. For solving Eq. 14, we therefore use the iterative formula

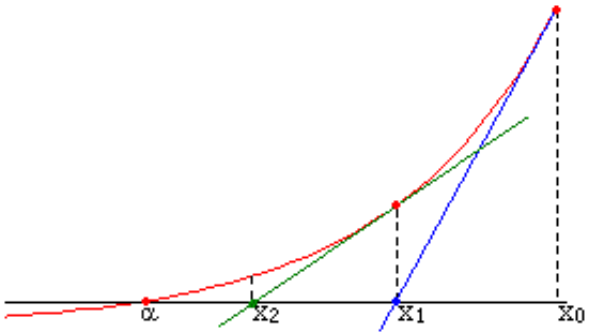


Figure 5: Newton's method for finding the root of a function. Source: `https://commons.wikimedia.org/wiki/File:Methode_newton.png`, User:Cham. Released into the public domain.

$$\theta_{n+1} = \theta_n - \frac{L'(\theta_n, X, Y)}{L''(\theta_n, X, Y)} \tag{18}$$

However, most of the time the second derivative is difficult to compute, and Newton's method only works well for convex problems.

2.4.3 Gradient descent

A versatile approach to find a minimum is *gradient descent*. It only requires that we are able to compute the *gradient* of the loss function $\frac{\mathrm{d}}{\mathrm{d}\theta}L(\theta, X, Y)$. To emphasize that the gradient is a vector the operator $\frac{\mathrm{d}}{\mathrm{d}\theta}$ is often written as ∇ ("nabla" operator) or as $\left(\frac{\partial}{\partial\theta_1}, \frac{\partial}{\partial\theta_2}, \ldots, \frac{\partial}{\partial\theta_k}\right)^T$, where k is the number of parameters of the model. The gradient indicates the direction of steepest increase of the function. By moving into the opposite direction, i.e., $-\nabla L$, we can find a local minimum of the loss function (Fig. 6). Interations:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t, X, Y). \tag{19}$$

η is the *learning rate*. Finding the appropriate learning rate for a given dataset is difficult (Fig. 7). Monitor the loss function in each iteration to spot potential issues.

A fundamental limitation of any optimization based on gradients is that the method can get stuck in *local minima*. While there are methods for avoiding this, there are never guarantees that they will find the global minimum.

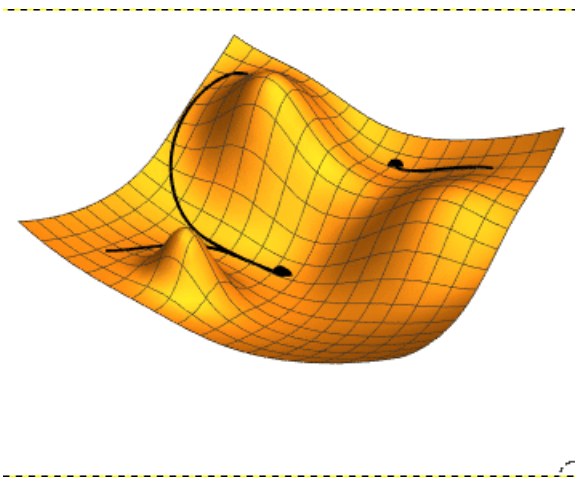


Figure 6: Gradient descent. Source: https://commons.wikimedia.org/wiki/File:Gradient_descent.gif. Author: Jacopo Bertolotti. Released into the public domain.

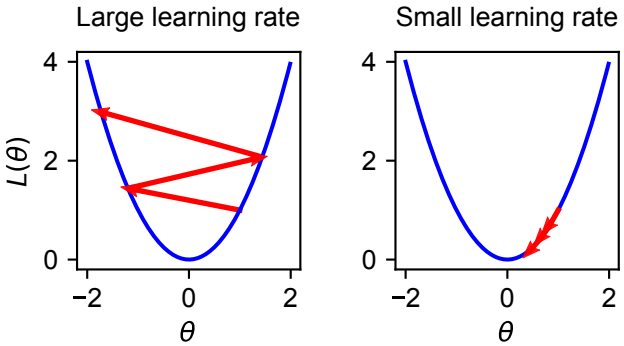


Figure 7: The effect of the learning rate. When it's too high, gradient descent may not converge (left) and when it is too small gradient descent requires much larger N (right).

2.4.4 Stochastic gradient descent, minibatch

The standard (or *batch*) gradient descent algorithm requires the evaluation of the gradients for all N datapoints before the parameter estimate can be updated

$$\theta_{t+1} = \theta_t - \eta \sum_{i=1}^{N} \nabla_\theta l\left(y_i, \hat{y}_i\right) \tag{20}$$

This is inefficient for very large datasets, because the parameters θ_t are kept fixed in the computation of all $\hat{y}_i$ and updated only after all N gradients are computed. This can take a long time, if the gradients are computationally costly. By contrast, if we updated the parameters after each gradient is computed, our parameter estimates would converge faster, but that would also make the fitting more sensitive to noise. That's why this method is called *stochastic gradient descent* (SGD).

$$\theta_i = \theta_{i-1} - \eta \nabla_\theta l\left(y_i, \hat{y}_i\right) \tag{21}$$

where i is iterated through the entire dataset. Typically multiple passes through the dataset are required. After each pass the data should be shuffled to prevent cycles.

To reduce the effect of the noise, but still speed convergence like in SGD, one can sum the gradients over mini-batches of size b

$$\theta_t = \theta_{t-1} - \eta \sum_{i=(t-1)b+1}^{tb} \nabla_\theta l\left(y_i, \hat{y}_i\right) \tag{22}$$

2.5 Convex optimization problems

Incremental optimization algorithms can always get stuck in a local minimum and therefore fail to find the globally optimal solution. However, for one class of functions convergence to the global optimal solution in guaranteed, if the learning rate is chosen appropriately.

Generally, an area is said to be *convex*, if a line connecting any two points within the area lies wholly within the area. A function is said to be convex, if the area above the graph of the function is convex. If a function $f(x)$ has a second derivative, $f(x)$ is convex, if and only if $f''(x) \geq 0$ everywhere in its domain. If f is a function of multiple (m) variables and is twice differentiable, then the second derivative $\nabla^2 f(x)$ is an $m \times m$ matrix, called the *Hessian*. In this case $f(x)$ is convex, if the Hessian is positive semi-definite everywhere in its domain. An $m \times m$ matrix M is positive semi-definite, if for any vector $z \in R^m$: $z^T M z \geq 0$. *For convex functions, a local minimum is also a global minimum.*

Therefore, in a *convex optimization problem*, i.e. one in which the loss function is convex, has special property. If gradient descent converges, the solution will be globally optimal.

2.6 Problem set

1. **Analytical minimization of a function**

   Study the minima of the function

   $$L(\theta) = \theta^4 - 4\theta^2 + 4 \tag{23}$$

   by following the steps below.

   (a) Use pen and paper to find the roots of $\dfrac{\mathrm{d}}{\mathrm{d}\theta} L(\theta)$.

   (b) Use again pen and paper to determine whether the roots of the derivative are minima or maxima of $L(\theta)$. What is the sufficient condition for minima and maxima?

   (c) Plot L against θ in Python for $\theta \in [-2, 2]$ and mark the minima (in red) and maxima (in green) in the plot.

2. **Numerical minimization of a function**

   Use gradient descent to find the minima of $L(\theta)$ from the problem above.

   (a) Implement gradient descent to find the minima of $L(\theta)$.

   (b) Starting the estimatation algorithm with different values, $\theta_0 \in [-2, 2]$, run the gradient descent algorithm with $\eta = 0.01$ and $N = 50$ (N, number of iterations). Observe how the starting value determines which local minimum is found.

(c) Set your initial guess $x_0 = 1$ and $N = 100$, use different learning rates $\eta \in [0.05, 0.15]$ (use `np.linspace`) to find the minimum. Calculate the precision of the estimation by computing the absolute value of the difference between the estimated and the actual minimum for each η. Plot the precision against η. How does η affect the precision of the estimation?

(d) Now set your initial guess $x_0 = 1$ and $\eta = 0.01$. Use different number of iterations, $N \in [1, 100]$. How does N affect the precision of the estimation?

3. **The role of the learning rate**

To better understand the implications of choosing an appropriate learning rate η in gradient descent, let us focus on a simpler loss function $L(\theta) = \theta^2$, because it has only one minimum.

(a) Use gradient descent to find the minimum of this function. To better understand the method you should keep track of the estimated minimum in each iteration, find the corresponding point on the function $L(\theta)$ and plot them together with the loss function.

(b) Set your initial guess $\theta_0 = 1$ and try few different $\eta \in [0.1, 1.1]$. Observe how the estimate depends on the learning rate. Find at least one η for which the estimate converges and one η for which it diverges.

3 Regression

3.1 Univariate linear regression

Simplest case of regression.

Data: one independent variable $x \in R$, and one dependent variable $y \in R$.
Model class: linear function, i.e.,

$$f(x) = mx + b \tag{24}$$

The parameters of the model are $\theta = (m, b)$.
Loss function: l_2-norm, or summed squared errors,

$$L(\theta, X, Y) = \sum_{i=1}^{N} (y_i - f(x_i))^2 \tag{25}$$

$$= \sum_{i=1}^{N} (y_i - mx_i - b)^2 \tag{26}$$

This is why this method is also known as the *minimum least square* (MLS) method.
Optimization: Linear regression is simple enough that we can derive an analytical solution, i.e., the parameters that will minimize the loss function.

The loss function is minimized for values of m and b such that $\frac{\partial}{\partial b} L = 0$ and $\frac{\partial}{\partial m} L = 0$. The first condition gives us

$$\sum_{i=1}^{N} -2(y_i - mx_i - b) = 0 \tag{27a}$$

$$\sum y_i - m \sum x_i - Nb = 0 \tag{27b}$$

which results in

$$b = \frac{1}{N} \sum y_i - m \frac{1}{N} \sum x_i \tag{28}$$

The second condition, $\frac{\partial}{\partial m} L = 0$, leads to

$$\sum_{i=1}^{N} -2x_i(y_i - mx_i - b) = 0 \tag{29}$$

substituting the value of b from Eq. 28 in here, we get

$$\sum x_i y_i - m\sum x_i^2 - \left(\sum x_i\right)\left(\frac{1}{N}\sum y_i - m\frac{1}{N}\sum x_i\right) = 0 \tag{30}$$

$$\sum x_i y_i - \frac{1}{N}\left(\sum x_i\right)\left(\sum y_i\right) = m\left[\sum x_i^2 - \frac{1}{N}\left(\sum x_i\right)^2\right] \tag{31}$$

$$m = \frac{\sum x_i y_i - \frac{1}{N}(\sum x_i)(\sum y_i)}{\sum x_i^2 - \frac{1}{N}(\sum x_i)^2} \tag{32}$$

Equation 28 and 32 give us the parameter values that will minimize the loss. A more memorable version of these equations is

$$m = \frac{\text{cov}(X,Y)}{\sigma^2(X)} \tag{33}$$

$$b = \bar{y} - m\bar{x} \tag{34}$$

where the covariance $\text{cov}(X,Y) = \frac{1}{N}\sum x_i y_i - \frac{1}{N^2}(\sum x_i)(\sum y_i)$.

3.2 Goodness of fit

Linear regression is guaranteed to find the linear function that best accounts for the data, but that doesn't mean much, if a linear function is not a good description of the data. How would you know? There are many methods. The simplest one is to look at the data and the fit line.

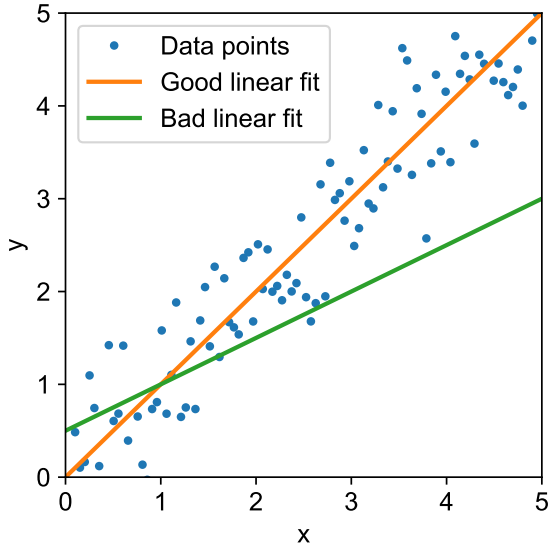


Figure 8: Good linear fit and bad linear fit

Explained variance R^2: A quantitative mesure of the quality of fit is the fraction of the variance in the data SS_{data} that can be accounted for by the model SS_{reg}.

$$SS_{data} = \sum_{i=1}^{N}(y_i - \bar{y})^2 \tag{35}$$

$$SS_{reg} = \sum_{i=1}^{N}(\hat{y}_i - \bar{y})^2 \tag{36}$$

$$R^2 = \frac{SS_{reg}}{SS_{data}} \tag{37}$$

R^2 is called the explained variance, because the total variance in the data can be partitioned into two components, the variance predicted by the regression estimates, SS_{reg}, and the deviation between model and data $SS_{err} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$

$$SS_{data} = SS_{reg} + SS_{err} \tag{38}$$

We can show that the sum of squares of the deviations can be partitioned as in Eq. 38 as follows :

$$y_i - \bar{y} = (\hat{y}_i - \bar{y}) + (y_i - \hat{y}_i) \tag{39a}$$

$$\underbrace{\sum_{i=1}^{N} (y_i - \bar{y})^2}_{SS_{data}} = \underbrace{\sum_{i=1}^{N} (\hat{y}_i - \bar{y})^2}_{SS_{reg}} + \underbrace{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}_{SS_{err}} + 2\sum_{i=1}^{N} (\hat{y}_i - \bar{y})(y_i - \hat{y}_i) \tag{39b}$$

Inserting $\hat{y} = mx + b$ and splitting up the last term, we obtain

$$\sum_{i=1}^{N} (\hat{y}_i - \bar{y})(y_i - \hat{y}_i) = \sum_{i=1}^{N} (mx_i + b)(y_i - mx_i - b) - \sum_{i=1}^{N} \bar{y}(y_i - mx_i - b) \tag{39c}$$

$$= m \underbrace{\sum_{i=1}^{N} x_i (y_i - mx_i - b)}_{=0 \quad \text{(Eq. 29)}} + (b - \bar{y}) \underbrace{\sum_{i=1}^{N} (y_i - mx_i - b)}_{=0 \quad \text{(Eq. 27a)}} \tag{39d}$$

Therefore

$$SS_{data} = SS_{reg} + SS_{err} \tag{39e}$$

Residual plot: calculate the residuals and plot them vs. the independent variable (Fig. 9).

3.3 Multiple linear regression

In many situations outcomes are influenced not by a single independent variable, but by multiple variables $(x_{\cdot 1}, \ldots, x_{\cdot m})$.

Model class:

$$y = \begin{bmatrix} x_{\cdot 0} & \cdots & x_{\cdot m} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_m \end{bmatrix} + \varepsilon \tag{40}$$

To make the notation simpler one of the variables, e.g., $x_{\cdot 0}$, is assumed to be equal to 1, so that the corresponding parameter θ_0 works out to be the y-offset, or the constant bias term. To simplify the notation when deriving and writing the solution, it is convenient to introduce the following notation that includes the relationship between all the observations in the dataset.

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} x_{10} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{N0} & \cdots & x_{Nm} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \vdots \\ \theta_m \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \vdots \\ \varepsilon_N \end{bmatrix} \tag{41}$$

or, more succinctly,

$$Y = X\theta + \varepsilon \tag{42}$$

The model prediction is

$$\hat{Y} = f(X) = X\theta \tag{43}$$

Similar to simple linear regression, we first define the summed square loss function as follows :

$$L(\theta, X, Y) = (Y - \hat{Y})^T (Y - \hat{Y}) \tag{44a}$$

$$= (Y - X\theta)^T (Y - X\theta) \tag{44b}$$

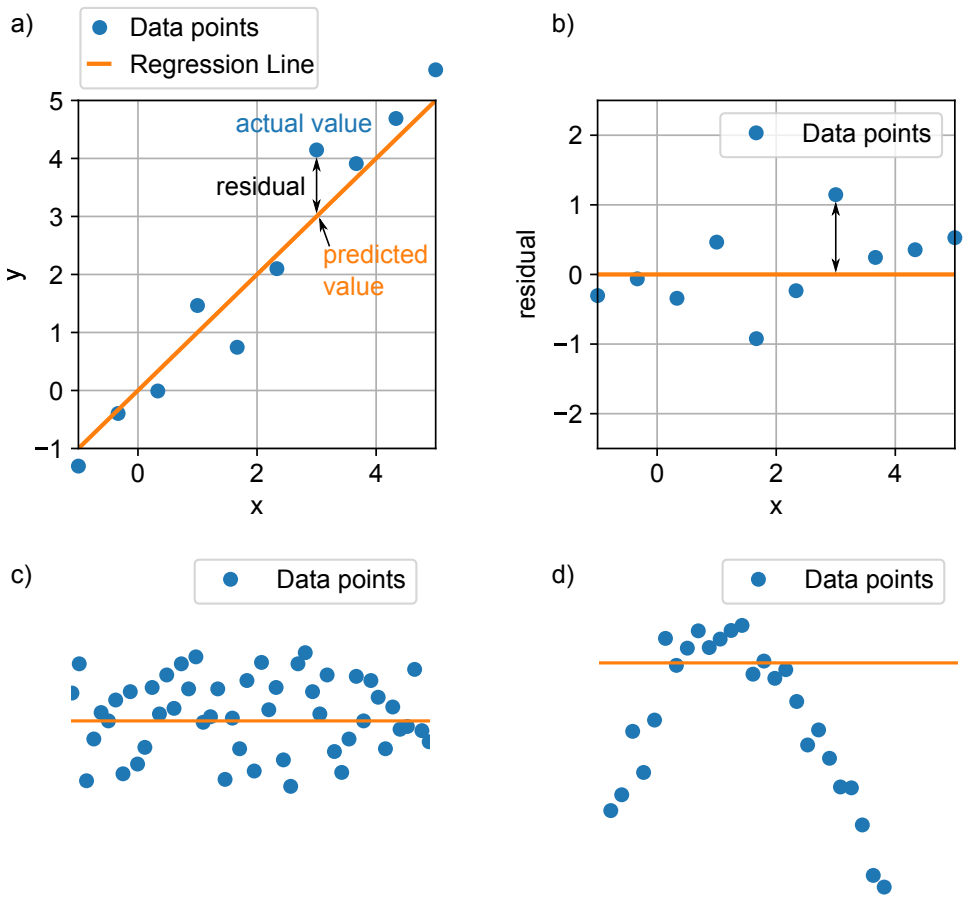


Figure 9: a) Residual on the regression plot and b) corresponding residual plot. Residual plots for good (c) and bad (d) fits.

Recall that for matrices A and B, $(AB)^T = B^T A^T$

$$= (Y^T - \theta^T X^T)(Y - X\theta) \tag{44c}$$
$$= Y^T Y - \theta^T X^T Y - Y^T X\theta + \theta^T X^T X\theta \tag{44d}$$

Since $\theta^T X^T Y$ is a scalar, it is equal to its transpose : $\theta^T X^T Y = (\theta^T X^T Y)^T = Y^T X\theta$

$$= Y^T Y - 2\theta^T X^T Y + \theta^T X^T X\theta \tag{44e}$$

The loss function is minimized when $\nabla_\theta L = 0$ This condition gives us

$$\nabla_\theta L(\theta, X, Y) = 0 \tag{45a}$$
$$-2X^T Y + 2X^T X\theta = 0 \tag{45b}$$
$$X^T X\theta = X^T Y \tag{45c}$$

Assuming that the matrix $X^T X$ is invertible, the solution is

$$\hat{\theta} = (X^T X)^{-1} X^T Y \tag{46}$$

3.4 Polynomial regression

What if the relationship between the input and output is not linear? A more general class are *polynomial functions*

$$f(x) = \sum_{j=0}^{m} \theta_j x^j \tag{47}$$

To fit a polynomial model to data, one can use a simple trick. Assume that each power of x is another variable, i.e., $x_{ij} = x_i^j$, and apply the multiple regression solution Eq. 46 to fit the coefficients $\hat{\theta}$ of the polynomial function.

3.5 Incremental linear regression

Even though an analytical solution for linear regression exists, we can still choose to apply an incremental algorithm. The loss per item is

$$l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2 = \left(y_i - x_i^T \theta\right)^2 \tag{48}$$

The gradient of the loss function is

$$\frac{\mathrm{d}}{\mathrm{d}\theta} l(y_i, \hat{y}_i) = -2(y_i - \hat{y}_i) x_i \tag{49}$$

Stochastic gradient descent applied to linear regression therefore yields the update equation

$$\hat{\theta}_i = \hat{\theta}_{i-1} + \eta (y_i - \hat{y}_i) x_i \tag{50}$$

3.6 The curse of dimensionality

If we can calculate multiple linear regression so easily, why do we mostly use few dimensions, i.e., small m? The reason is that the amount of data needed to generalize accurately grows exponentially as the number of features or dimensions grows.

The curse of dimensionality arises because the volume of a cube with length a and dimension d scales as a^d. To generalize accurately the model needs to have samples with a certain density. Let's say a point can inform us about its neighborhood up to a certain distance. For instance, in 1 dimensions we might need 10 data points to cover the length a (Fig. 10), then in 2-d we need 10^2 and in d dimensions, we'd need 10^d data points. For example, for a $10x10$ pixel image with 10 grey scale values, we would need 10^{100} data points! For comparison, the total number of atoms in the observable universe is estimated to be "only" 10^{80}.

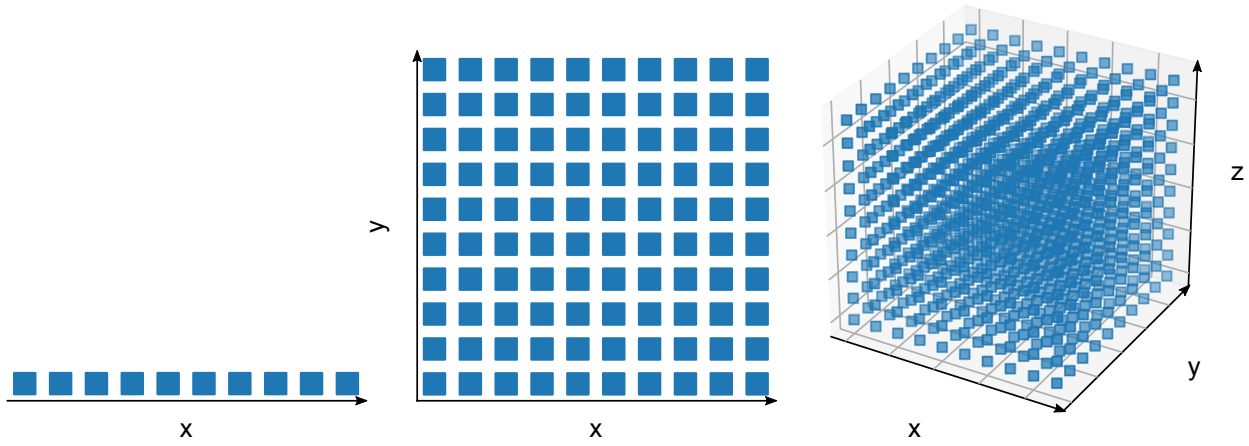


Figure 10: The curse of dimensionality.

3.7 Problem set

1. **Analytical solution of linear regression**

   Although in real-world problems you do not know the function that has generated the dataset, here we assume that we know the generating function known as "ground-truth". You will use the functions here to generate data and later fit a linear regression to the data, so you can see how close the regression is to the ground-truth.

   Let us start with the function

   $$y = 4x + 5 + \varepsilon \tag{51}$$

   as the ground-truth to generate a dataset. ε is a uniform random variable, which generates random numbers between -0.5 and 0.5.

   (a) Generate a vector x containing 100 linearly spaced numbers from -3 to 3 (use `numpy.linspace`) and a vector of 100 numbers for ε (use `numpy.random.uniform`). Now compute vector y according to Eq. 51 and make a scatter plot of y against x.

   (b) Use the solution for linear regression to estimate the parameters m and b from the data. Implement the equations yourself using only elementary operations.

   (c) Use the analytical solution for multiple linear regression to estimate the parameters. Make sure you have designed your X matrix correctly. Implement the equations yourself using only elementary matrix operations.

   (d) Compare the parameters you estimated to the ground-truth. Check how good the estimate is by plotting the the regression line along with the data points. Also compute the explained variance, and generate the residual plot.

   (e) Use the scikit-learn python package to perform linear regression. From `sklearn` import `linear_model` and use the method `LinearRegression`. Using the data you generated in the first step, fit a linear regressor. First look at the instructions provided by `help(linear_model.LinearRegression)`. Calculate the explained variance R^2 using `sklearn.metrics.r2_score`.

2. **Polynomial regression**

   Take a stochastic polynomial function

   $$y = -x^5 + 1.5x^3 - 2x^2 + 4x + 3 + \varepsilon, \tag{52}$$

   where $-2 < x < 2$ and ε is a random number uniformly distributed between -1 and 1.

   Follow the steps below:

   (a) Generate 100 data points using the polynomial function

(b) Fit a 5th-order polynomial with intercept to the dataset. Use the equation for multiple linear regression and a properly designed matrix X to find the coefficients of the polynomial.

(c) Compare the parameters you estimated to the ground-truth. Check how the estimated parameters vary with the number of data points by plotting the error in the estimates versus the number of data points N. Use the l_2-norm of the difference between true and estimated parameters as an error measure, and vary N in steps of 5 up to a range of 100.

3. **Incremental version of linear regression**

   Often in practice, datasets can be extremely large, which makes it infeasible to use the analytical solution. Use stochastic gradient descent (SGD) to compute the regression parameters as follows :

   (a) Program stochastic gradient descent update for linear regression using the l_2-loss function. Again, only use elementary operations. (Hint: don't forget to shuffle the data before iterating through it!)

   (b) Test this incremental form on the function you used in exercise 1.

   (c) Compare the parameters you estimated to the ground-truth. Plot both the data and predicted values for a visual check. Check how the number of data points affect the estimates by generating a plot as in Problem 2c. Use a constant learning rate of 0.01 for this and use 3 runs of SGD. Next, keep the number of data points constant at 100 and use 3 runs of SGD, and vary the learning rate in steps of 0.001 up to a range of 0.4. Generate a similar plot of error vs. learning rate.

4 Model selection/ Regularization

4.1 Generalization

The goal of curve fitting is to minimze the loss function. Most worry about *underfitting*, i.e., the model is not able to describe the data very well (Fig. 11).

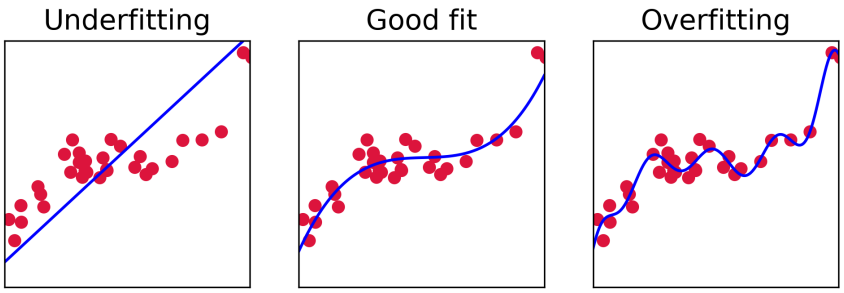


Figure 11: Example of underfitting and overfitting.

However, describing the data well that we already have is only a means to an end. The fit can be too good (*overfitting*) and start capturing the noise. The actual goal is to use the model to make predictions about future data. So what we actually want is that the model generalizes to data that the model was not trained on. If the model captures the noise in the *training data*, it will not generalize well to new data, also called *test data* (Fig. 12).

It is, therefore, a good practice to split the available dataset into *training data* and *test data*. The former is used to determine the parameters of the model in an optimization procedure, the latter is used to evaluate how well the model is doing.

4.2 Bias-variance tradeoff

When evaluating a fitted function on test data, there are three sources of error. First, the *bias* is the systematic error that the estimator $\hat{f}$ makes in describing the data, i.e., $\text{bias}[\hat{f}] = f - \text{E}\left[\hat{f}\right]$. It is the result of a mismatch between the

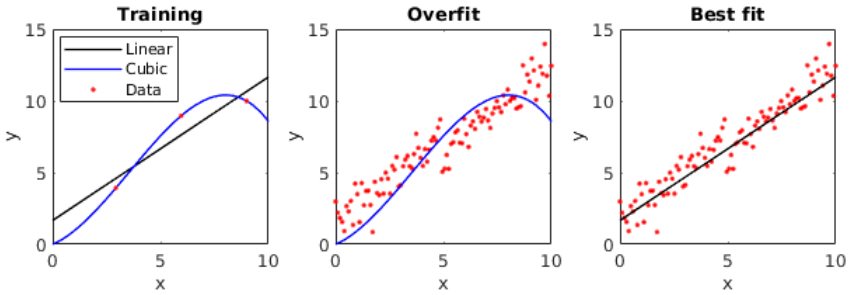


Figure 12: Overfitting leads to bad predicitons. A cubic model (blue) perfectly fits the training data (left), but it fails to generalize to new data (middle), which is better captured by the linear fit (right). Note that a cubic model has more parameters than the data points available for training in this case.

fitted function and the function underlying the data. Second, the *variance* is the fluctuation of the estimator from one dataset to another, i.e., $\mathrm{var}[\hat{f}] = \mathrm{E}\left[(\mathrm{E}\left[\hat{f}\right] - \hat{f})^2\right]$. It is the result of the sensitivity of the estimator to small fluctuations in the data. Third, the *irreducible error* is the error inherent in the data, i.e., $\mathrm{var}[y] = \sigma^2$. It turns out that the error of any estimator on any dataset is simply a sum of these three errors.

$$\text{error} = \mathrm{E}\left[\left(y - \hat{f}(x)\right)^2\right] = \mathrm{bias}[\hat{f}]^2 + \mathrm{var}[y] + \mathrm{var}[\hat{f}] \tag{53}$$

Proof:

Since $y = f(x) + \varepsilon$, the error can be expanded

$$\mathrm{E}\left[\left(f + \varepsilon - \hat{f}\right)^2\right] \tag{54a}$$

$$= \mathrm{E}\left[\left(f - \mathrm{E}\left[\hat{f}\right] + \varepsilon + \mathrm{E}\left[\hat{f}\right] - \hat{f}\right)^2\right] \tag{54b}$$

$$= \mathrm{E}\left[(f - \mathrm{E}\left[\hat{f}\right])^2 + \varepsilon^2 + (\mathrm{E}\left[\hat{f}\right] - \hat{f})^2 + 2(f - \mathrm{E}\left[\hat{f}\right])\varepsilon + 2(f - \mathrm{E}\left[\hat{f}\right])(\mathrm{E}\left[\hat{f}\right] - \hat{f}) + 2\varepsilon(\mathrm{E}\left[\hat{f}\right] - \hat{f})\right] \tag{54c}$$

The function f and the expectation value of a random variable $\mathrm{E}[\cdot]$ are deterministic, ε is independent of $\hat{f}$.

$$= (f - \mathrm{E}\left[\hat{f}\right])^2 + \mathrm{E}\left[\varepsilon^2\right] + \mathrm{E}\left[(\mathrm{E}\left[\hat{f}\right] - \hat{f})^2\right] + 2(f - \mathrm{E}\left[\hat{f}\right])\underbrace{\mathrm{E}[\varepsilon]}_{=0} + 2(f - \mathrm{E}\left[\hat{f}\right])\underbrace{\mathrm{E}\left[\mathrm{E}\left[\hat{f}\right] - \hat{f}\right]}_{\mathrm{E}[\hat{f}] - \mathrm{E}[\hat{f}] = 0} + 2\underbrace{\mathrm{E}[\varepsilon]}_{=0}\mathrm{E}\left[\mathrm{E}\left[\hat{f}\right] - \hat{f}\right] \tag{54d}$$

$$= (f - \mathrm{E}\left[\hat{f}\right])^2 + \mathrm{E}\left[\varepsilon^2\right] + \mathrm{E}\left[(\mathrm{E}\left[\hat{f}\right] - \hat{f})^2\right] \tag{54e}$$

$$= \mathrm{bias}[\hat{f}]^2 + \sigma^2 + \mathrm{var}[\hat{f}] \tag{54f}$$

Models of low complexity (which often, but by far not always, correlates with the number of parameters) tend to underfit, and models of high complexity tend to overfit (Fig. 13). Since the total error is the sum of bias and variance, the two can be traded-off. The optimum would be where the total is minimal.

4.3 Model selection

The selection of the model class should be based on the data. Fig. 13 tells us what model complexity would be optimal, but unfortunately it is a theoretical result assuming we knew the correct function and noise. In pratice, both are unknown. So we can only estimate the bias by using the training error and the variance using the generalization error.

In addition, most model classes have hyperparameters that need to be tuned. In the end, we choose the model class and hyperparameter that yield the lowest generalization error. However, now the generalization error cannot be used to gauge the performance of the model anymore, since we optimized the model and its hyperparameters to get as low a generalization error as possible. Therefore, we need to split the dataset into three subsets:

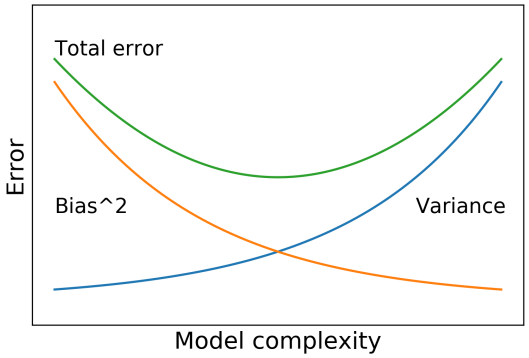


Figure 13: Bias-variance tradeoff.

training data	validation data	test data
↓	↓	↓
fitting	model selection	performance

The *validation data* is used for model selection and the test data for assessing the performance of the chosen model. Do not tweak the model or its hyperparameters to reduce the test error.

4.4 Cross validation

Often we don't have very large datasets available, so splitting the data into three subsets might not leave enough data. One way to solve this problem is to use *n-fold cross-validation* (Fig. 14). In this method, the data is partitioned into n blocks and the model is trained and tested n times. Each time or 'epoch', one of the blocks is used as the test set and the remaining blocks as the training set. The mean error on the test set across epochs is then used as a performance measure of the model, which can be used to compare it to alternatives.

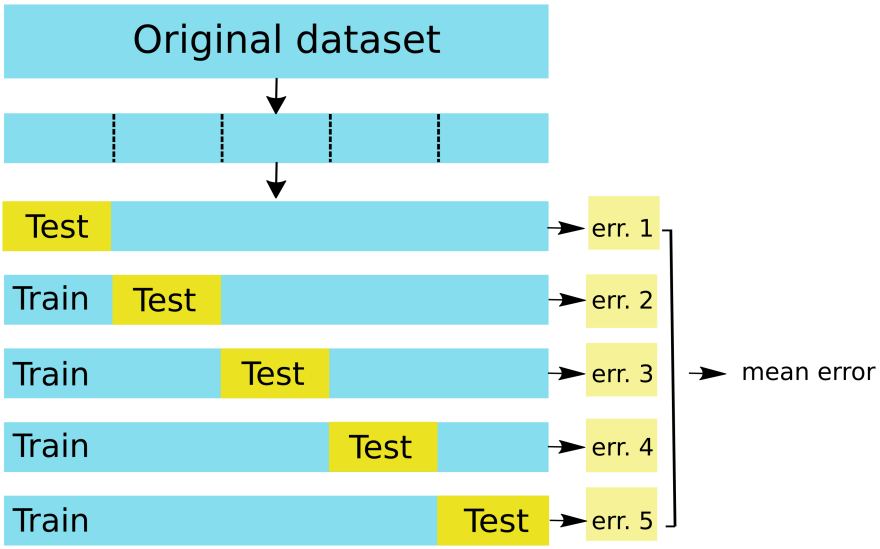


Figure 14: Example of 5-fold cross-validation

4.5 Regularization

Model selection is difficult. It would be nice to learn from the data, which parameters are needed and which are not. This can be accomplished by constraining the parameters during optimization and is called *regularization*. During

normal fitting, the optimization is very likely to assign large values to the parameters that are unimportant for describing the data. This counterintuitive result comes about because changing an unimportant parameter doesn't hurt model performance, but can account for some variance caused by noise. We can reduce this behavior, by including the values of the parameters themselves as an additional term in the loss function.

$$\hat{\theta} = \arg\min_{\theta}[L(\theta, X, Y) + \lambda R(\theta)] \tag{55}$$

The l-norm is frequently used for normalization, i.e., $R(\theta) = l_k(\theta)$. The regularizing term in the loss function forces the optimization to make the parameter values as small as possible, unless that would drive up the error in the first term. The hyperparameter λ controls the importance of the regularizing term. The larger λ, the more model complexity is penalized.

When using gradient descent, *early stopping* can also be used for regularization.

4.6 Ridge regression

Ridge regression is linear regression with l_2-norm regularization.

The predictors need to be standardized (same scale) before performing ridge regression.

$$\tilde{x}_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_{ij} - \bar{x}_{\cdot j})^2}} \tag{56}$$

Otherwise the parameters have very different scales and they would contribute very differently to the regularizing term.

Model class: $Y = X\theta + \varepsilon$

Loss function:

$$L(\theta, X, Y) = (Y - X\theta)^T(Y - X\theta) + \lambda\theta^T\theta \tag{57}$$

Optimization:

$$\begin{aligned} 0 &= \nabla_\theta L(\theta, X, Y) && (58a)\\ &= \nabla_\theta \left(Y^TY - 2\theta^TX^TY + \theta^TX^TX\theta + \lambda\theta^T\theta\right) && (58b)\\ &= -2X^TY + 2X^TX\theta + 2\lambda\theta && (58c)\\ &= -X^TY + (X^TX + \lambda I)\theta && (58d) \end{aligned}$$

$$\hat{\theta} = (X^TX + \lambda I)^{-1}X^TY \tag{59}$$

The solution is a minimum of the loss function since the Hessian

$$\nabla_\theta L(\theta, X, Y) = 2X^TX + 2\lambda I \tag{60}$$

is positiv semi-definite, since , $z^T\nabla_\theta L(\theta, X, Y)z \geq 0$ for any vector z. To see this note that $z^TX^TXz = (Xz)^T(Xz) = y^Ty \geq 0$ and $z^Tz \geq 0$ for any vector z

Regularization can also help in cases, in which the matrix X^TX is singlular or close to singular, i.e., not invertible or inversion is unstable. This occurs in particular when the colums of X, i.e., the observations, are linearly dependent.

4.7 Lasso regression

Lasso regression is linear regression with l_1-norm regularization. The acronym "LASSO" stands for Least Absolute Shrinkage and Selection Operator.

Model class: $Y = X\theta + \varepsilon$

Loss function:

$$L(\theta, X, Y) = (Y - X\theta)^T(Y - X\theta) + \lambda\sum_{j=1}^{m}|\theta_j| \tag{61}$$

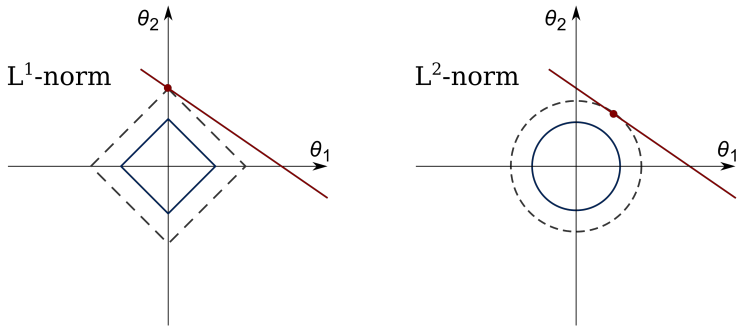


Figure 15: Constraints in ridge and lasso regression. Source (Adapted from): `https://commons.wikimedia.org/wiki/File:L1_and_L2_balls.svg`, User:Nicoguaro. Licensed under Creative Commons BY 4.0.

Optimization: There is no analytical solution for lasso regression. So, numerical methods have to be used to estimate the parameters.

Lasso is more likely than ridge regression to yield parameters that are zero. The resulting models are sparser (Fig. 15).

4.8 Problem set

1. **The holdout method**

   (a) Load the file *'04_model_selection_data.npy'*. The first and second columns of the data refer to the predictors (X) and targets (Y), respectively. Split the dataset into training and validation sets using the *train_test_split* function from the *sklearn.model_selection* package. Use a ratio of 80:20 and set the parameter *random_state* of the *train_test_split* function to a fixed value.

   (b) Expand the feature space of X using polynomials of N degrees (say $N \in [1,6]$). To generate the polynomial features, use the class *PolynomialFeatures* from *sklearn.preprocessing*.

   (c) Use *LinearRegression* from *sklearn.linear_model* to fit the data with polynomials of different degrees N.

   (d) Plot the regression lines along with the training data points.

   (e) Plot the *mean squared error* vs. the model complexity for the training and validation sets. Describe the resulting fits in terms of bias and variance. Which model seems to best fit the data? When using different random states do you always have the same best fit model? Why/Why not?

2. **k-fold cross-validation**

   (a) Use k-fold cross-validation to determine the optimal model for the data in exercise 1. You can use the *KFold* function from *sklearn.model_selection* package to obtain indices for the training and validation sets. Although cross-validation uses all available data for both training and validation, it doesn't use all possible combinations of the data points. Therefore, to check the stability of your estimation, shuffle the data for each split by setting the parameter shuffle in *KFold* function to True.

   (b) Run the regression for $k \in [2,4,5,10,20]$. For each k, plot the training and validation performances vs model complexity and select the model with the best performance. Repeat and see for which k do you get the most stable pattern of training and test errors.

3. **Regularization**

   (a) Run regression using the best polynomial degree that you've identified in the last exercise. Then, use ridge regression for 9-degree polynomial features with different regularization strengths (λ). Use *Ridge* and *Lasso* classes from *sklearn.linear_model*.

   Note: for the current data the range of λ should be very small: $\lambda \in [0, 0.003]$.

   (b) Plot the training and validation error vs λ for the ridge regression as well as the error from the best fit model for comparison. How does the regularization affect the model performance?

(c) Plot the the polynomial coefficients vs λ.

(d) Repeat (a)-(c) for Lasso regression. Does the λ have the same influence as in ridge regression? Is the number of nonzero coefficients what you would have expected?

5 Classification

Regression: predict values, continuous output

Classification: predict class, discrete output

An algorithm that makes classification judgements is called a *classifier*. In its simplest form we would like to determine whether an instance belongs to a class or not. This is called *binary classification*. It is more common than one might think. Examples: spam filters, predicting pass/fail on tests, credit card fraud detection, developing cancer or not, etc.

5.1 Logistic regression (logit regression)

The first binary classifier.

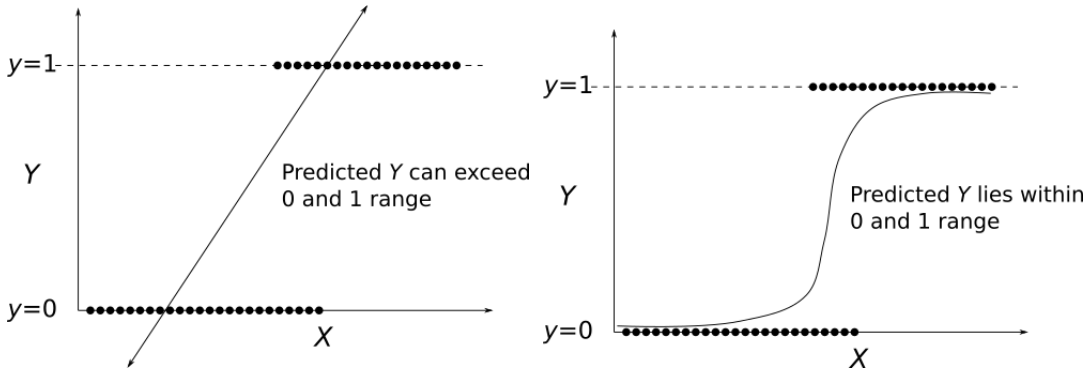
Data: continuous input X, discrete output Y

Model class: The model consists of two components: a function that computes a probability $\hat{p}(x)$ that a particular item x belongs to the class, and a threshold process that makes a decision based on the probability. The latter component is straight-forward and takes the form

$$\hat{y}(x) = \begin{cases} 0 & \hat{p} < p_0 \\ 1 & \hat{p} \geq p_0 \end{cases} \tag{62}$$

Usually $p_0 = 0.5$, but we can set it to other values to be more conservative or more liberal – depending on the problem at hand.

The first component, a function that calculates the probability, is more complicated. It would be nice if we could use a linear function $f(x) = \theta^T x$ (as we did in linear regression) to parametrize the predicted probability $\hat{p}(x)$ because it's simple and it takes into account and combines different factors or features, x_j, i.e. $\theta^T x = \sum \theta_j x_j$. However, the linear function takes on values from $-\infty$ to ∞ (Fig. 16a), and probabilities are constraint: $0 \leq \hat{p}(x) \leq 1$. That's why we relate another quantity of the stochastic process to $\theta^T x$.



(a) Linear regression.

(b) Logistic regression.

Assume we have a Bernoulli process with probability p, such that $0 < p < 1$. The *log odds ratio*, $\log\left(\frac{p}{1-p}\right)$, is

continuous and ranges from $-\infty$ to $+\infty$, so it can be equated to $\theta^T x$.

$$\log\left(\frac{p}{1-p}\right) = \theta^T x \tag{63a}$$

$$\frac{p}{1-p} = e^{\theta^T x} \tag{63b}$$

$$p = (1-p)e^{\theta^T x} \tag{63c}$$

$$p\left(1+e^{\theta^T x}\right) = e^{\theta^T x} \tag{63d}$$

$$p = \frac{e^{\theta^T x}}{1+e^{\theta^T x}} \tag{63e}$$

Hence, the probability in logistic regression is (Fig. 16b)

$$p(x) = \frac{1}{1+e^{-\theta^T x}} \tag{64}$$

That's why the function

$$\sigma(t) = \frac{1}{1+e^{-t}} \tag{65}$$

is called the *logistic function*. It's sometimes referred to as *the* sigmoid function, even though a *sigmoid function* is generally any s-shaped curve that is bounded.

To summarize the classification model in logistic regression:

$$\hat{y}(x) = \begin{cases} 0 & \sigma(\theta^T x) < p_0 \\ 1 & \sigma(\theta^T x) \geq p_0 \end{cases} \tag{66}$$

Later, we will need the derivatives of this function.

$$\frac{\mathrm{d}}{\mathrm{d}t}\sigma(t) = \frac{-1}{(1+e^{-t})^2}e^{-t}(-1) \tag{67a}$$

$$= \frac{1+e^{-t}-1}{(1+e^{-t})^2} \tag{67b}$$

$$= \frac{1}{1+e^{-t}} - \frac{1}{(1+e^{-t})^2} \tag{67c}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\sigma(t) = \sigma(t)\left(1-\sigma(t)\right) \tag{67d}$$

5.1.1 Cross entropy

The loss function should capture the difference between two probability distributions.

	in class	not in class
true	p_1	p_2
predicted	$\hat{p}_1$	$\hat{p}_2$

Since probabilities have to sum up to 1, we can simplify this to

	in class	not in class
true	p	$1-p$
predicted	$\hat{p}$	$1-\hat{p}$

The *cross entropy* is frequently used in *information theory*, and now machine learning, to quantify the difference between two statistical distributions. We don't have time to derive the equation for cross entropy.

$$H(p,\hat{p}) = -\sum_c p_c \log(\hat{p}_c) \tag{68}$$

Applied to binary classification of one item

$$H(p,\hat{p}) = -p_1 \log(\hat{p}_1) - p_2 \log(\hat{p}_2) \tag{69}$$
$$= -y \log(\hat{p}) - (1-y)\log(1-\hat{p}) \tag{70}$$

The cross entropy works better for fitting probabilities because it increases much more steeply for deviations than the

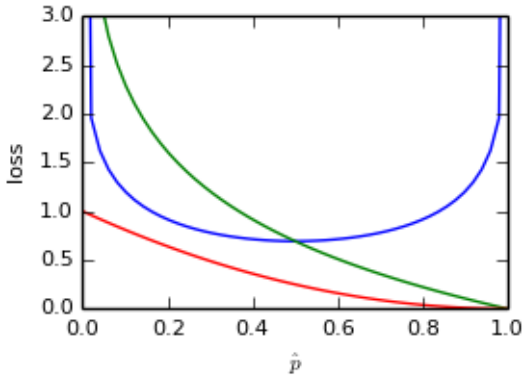


Figure 16: Loss functions as a function of estimated probability. Binary cross entropy, $p = 0.5$ (blue), $p = 1$ (green). Square loss, $p = 1$ (red).

square loss, which is rather small in the interval $[0,1]$ (Fig. 16).

Since the cross entropy is calculated for individual items, we have to sum the cross entropies over all items to get the **loss function** for the entire data:

$$L(\theta, X, Y) = -\frac{1}{N}\sum_{i=1}^{N} [y_i \log(\hat{p}(x_i)) + (1-y_i)\log(1-\hat{p}(x_i))] \tag{71}$$

5.1.2 Incremental algorithm

Optimization: Unlike in linear regression, no closed form solution is known for the parameters $\hat{\theta}$ that will minimize the loss function in logistic regression. So we have to use a numerical method, such as gradient descent, to find the global minimum. The gradient of the loss function is

$$\nabla_\theta L(\theta, X, Y) = \frac{1}{N}\sum_{i=1}^{N} \left(\sigma(\theta^T x_i) - y_i\right) x_i \tag{72}$$

The derivaiton of Eq. 72 is left as a homework excercise.

If the loss function Eq. 71 is convex, gradient descent is guaranteed to converge to the global minimum, if the learning rate is chosen appropriately. The loss function is convex, if the second derivative, the Hessian matrix, is positive semi-definite.

$$\nabla^2_\theta L(\theta, X, Y) = \frac{1}{N}\sum_{i=1}^{N} \sigma(\theta^T x_i)\left(1 - \sigma(\theta^T x_i)\right) x_i x_i^T \tag{73}$$

where we have used Eq. 67d and the chain rule. Since the sigmoid is constrained between 0 and 1, the first two terms are positive. The matrix xx^T is positive semi-definite because for any vector z: $z^T xx^T z = (x^T z)^2 \geq 0$. Therefore, the Hessian is positive semi-definite.

5.2 Analyzing performance

The performance of a classifier is more difficult to assess than that of a regression algorithm. In linear regression, the square loss function immediately gives you an intuitive sense of how far off your estimates are. The cross entropy is

		predicted	
		in class	**not in class**
true	**in class**	true positive (TP)	false negative (FN)
	not in class	false positive (FP)	true negative (TN)

Table 1: Types of correct responses and errors in binary classsification. False positives are also known as type 1 error, and false negatives as type 2 error.

somewhat opaque. Also, the deviations in the probability distribution that it captures might be far from the needs in an application. To understand this, note that there are two types of errors in binary classification (Table 1). The two different types of errors (false positives and false negatives) can be traded off, i.e., one can choose which error rate to minimize, but that will increase the rate of the other error. This choice is introduced by the parameter p_0. The larger p_0, the harder it is to classify an item as in the class, so FP goes down and FN goes up.

In some cases, it might be better to put as many item as possible in the class, i.e. minimize the rate of FN. However, that will come at the price of raising the rate of FP. For instance, if your test detects cancer, patients prefer that you catch all real instances of cancer, even if that means that there are many false alarms (FP), since early treatment increases survival rates. In some other cases, one might prefer to only put items in the class if they truely belong there (minimize FP), at the expense of missing a few (increasing FN). This might be the case if you decide whether to perform a heart transplant. The procedure is so risky and expensive that you don't want to perform it unnecessarily, even if that means that in a few cases you don't perform a surgery that could have helped the patient. Finally, in yet other cases, it might be best to find lowest overall error rate, irrespective of which type the error is.

It is useful to have a way to measure how well our classifier describes the data regardless of our choice of which error to minimize. We turn to such methods next. They are also useful to select an appropriate threshold.

5.2.1 Precision-Recall curve

The *precision* is defined as the fraction of "yes" responses that are correct, i.e.,

$$\text{precision} = \frac{TP}{TP+FP} \tag{74}$$

Recall is defined as the fraction of class items that are correctly classified, i.e.,

$$\text{recall} = \frac{TP}{TP+FN} \tag{75}$$

This is also known as the *hit rate*. For both precision and recall, higher values are better.

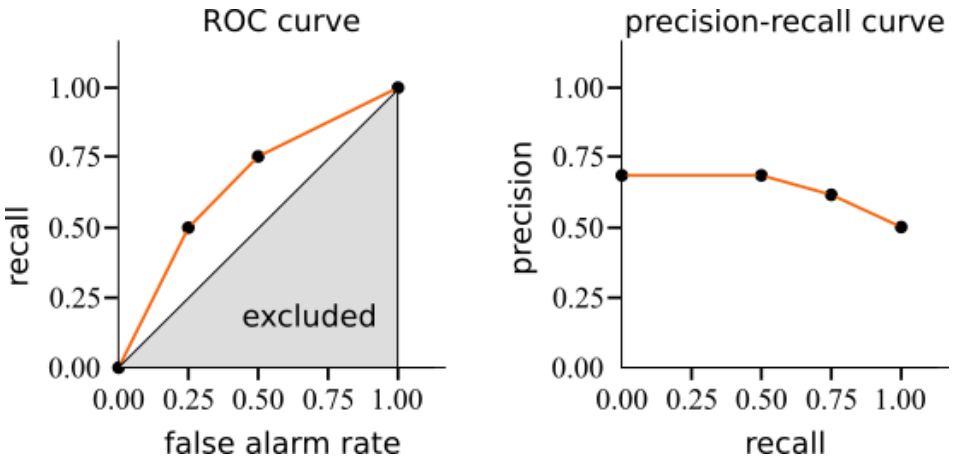


Figure 17: Precision-Recall and ROC curve.

The precision-recall curve is a plot of precision against recall that is obtained by running the classification with different decision thresholds (Fig. 17). A given value for the decision threshold leads to a certain point on the precision-recall curve. Since the curve is monotonically decreasing, you can either have high precision or high recall, but not

both at the same time. This is known as the *precision-recall tradeoff* and it is a direct consequence of the tradeoff between FN's and FP's. One strategy to pick a decision threshold that is a good tradeoff is the following method: start on the left most point (highest precision), move along the curve, and stop just before the curve descends steeply. Precision and and recall are sometimes combined into a single F_1 score, which is the harmonic mean of the two variables;

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{76}$$

Hence, the F_1 score takes into account both error types. One could pick a decision threshold to maximize F_1.

5.2.2 ROC curves

The receiver operating characteristics (ROC) curve is a plot of the recall rate (hit rate) against the rate of false positives (false alarm rate), which is

$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{77}$$

The ROC curve is monotonically increasing due to the tradeoff between the two types of errors. The identity line ($y = x$) indicates random guessing. The further away the ROC curve is from the identity line, the better the performance of the classifier. The lower triangle in Fig. 17 is excluded, because performance in this area would indicate that out-of-class items were predicted to be in-class more often than in-class items were. This would only be possible, if the classifier knew what the right answers were, but for some reason gave the wrong answer instead.

Picking a decision threshold selects a point along the ROC curve, but to move to a different curve requires changing the data or the classifier. That's why you can compare the performance of different classifiers by looking at the *area under the curve (AUC)*. The higher the AUC, the better. The maximum is AUC=1.

Rule of thumb: Use the precision-recall curve, when the positive class is rare, or when you care more about the FP than the FN. Otherwise use the ROC curve.

5.3 Multiclass classification

In many applications, the feature vectors can be classified into more than two classes (Fig. 18). We will discuss

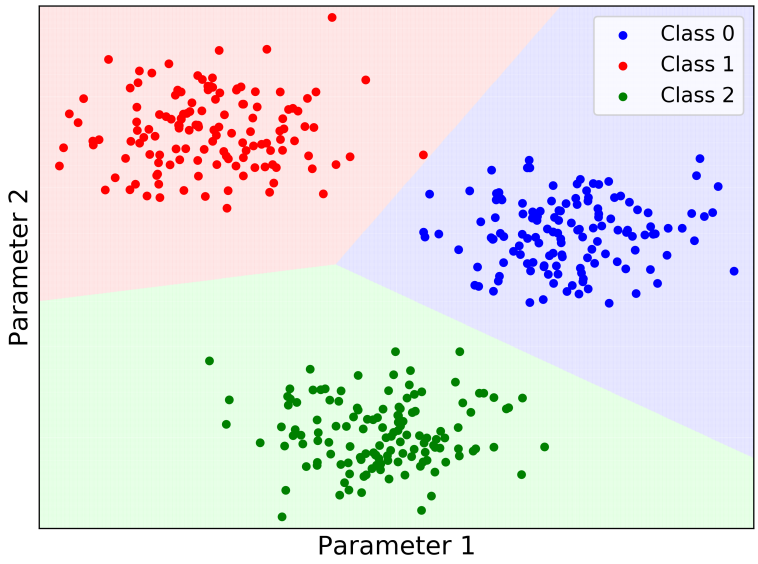


Figure 18: Classification with multiple classes.

true multiclass classifiers later. However, multiclass classification can be achieved simply by using multiple binary classifiers. There are two strategies: one-vs-all (a.k.a. one-vs-all-other, one-vs-rest) and one-vs-one.

One-vs-all: One binary classifier is trained for each class $c_i, i = 1, \ldots, k$. Training includes the entire dataset, where all items in c_i belong to the class and all items in other classes do not. This method only works for binary classifiers that can return a real value that represents how well a given item matches the characteristics of the class. For instance, one can extract a probability from logistic regression. Given a feature vector x, the output of each binary classifier $\hat{p}_i(x)$ is treated as a vote for the class c_i, which the classifier was trained on. The class receiving the highest vote is chosen.

$$\hat{y} = \underset{i}{\operatorname{argmax}}\, \hat{p}_i(x) \tag{78}$$

This method is frequently used. One of the biggest problems with this approach is that the training sets for the binary classifiers contain many more negative than positive items. This is not ideal for training binary classifiers.

One-vs-one: In this method, $k(k-1)/2$ binary classifiers are trained for multiclass classification with c classes. Each binary classifier is trained on items from a pair of classes to distinguish between these two. Given a feature vector x, each binary classifier gives a vote for one of the two classes it was trained on. The class receiving the highest number of votes is chosen as the final output. Unlike one-vs-all, this method works when the binary classifiers only ouptput binary decisions. One issue with this approach is that in some region of feature space, there might be ambiguity, i.e., more than one class receives the highest number of votes.

5.3.1 Confusion matrix

The *confusion matrix* M_{ij} indicates how often an item from class c_i is (mis-)classified by the algorithm as belonging to class c_j (Fig. 19). The confusion matrix helps visualize the overall performance of the multiclass classifier. If the performance is ideal, i.e. all items were assigned to the right class, the confusion matrix has nonzero entries along the diagonal, and zero entries everywhere else.

(a)

actual class \ predicted class	**1**	**2**	$\cdots$	**k**
1	M_{11}	M_{12}	$\cdots$	M_{1k}
2	M_{21}	M_{22}	$\cdots$	M_{2k}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
k	M_{k1}	M_{k2}	$\cdots$	M_{kk}

(b)

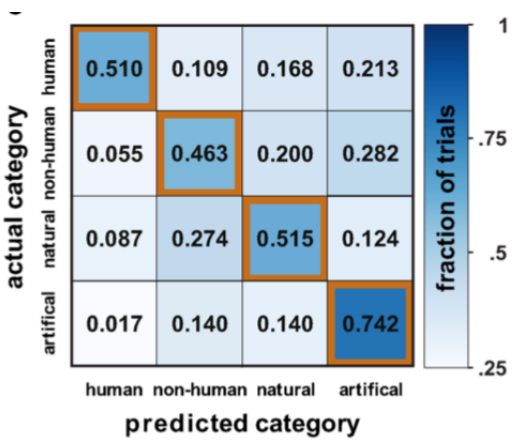


Figure 19: (a) Confusion matrix showing how many items from the acutal class c_i were (mis-)classified as class c_j by the classifier. (b) Example of a confusion matrix from Azizi et al. (2019).

5.4 Bayes classifier and Bayes error rate

Bayes' rule is very useful in cases where two things, e.g., A and B, depend on each other.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{79}$$

Each term in Bayes' rule has a name: $P(A|B)$ is called the *posterior*, meaning the probability of A *after* we have observed B. Accordingly, $P(A)$ is the *prior*, the probability of A before we have observed B. $P(B|A)$ is the *likelihood* of B given A. Finally, $P(B)$ is the *normalization constant* to ensure that the right-hand side of the equation sums/integrates

to one, as required for a probability distribution.

$$P(B) = \sum_{A's} P(B|A)P(A) \tag{80}$$

Bayes' rule can be applied to classification. This is then called the *naïve Bayes classifier*. Assume that we have a number of classes, denoted by c_i and feature vectors x. The classification problem could be solved if we knew $P(c_i|x)$ for all i. A given feature vector is assigned to the class c_i for which $P(c_i|x)$ is the highest. According to Bayes' rule

$$P(c_i|x) = \frac{p(x|c_i)P(c_i)}{p(x)} \tag{81}$$

Of course, writing the probability as a posterior only helps if the prior and the likelihood are known. This is sometimes the case, but much of the time it is not. Even though the name naïve Bayes classifier might suggest that it is a specific algorithm for classification, it is actually a general model of what a classification problem is about.

The *Bayes error rate* is a lower bound on the error rate in a classification problem, i.e., the lowest possible error rate that any classifier could theoretically achieve. It is analogous to the irreducible error in regression problems. The Bayes error rate is useful as a comparision to judge how close to optimal a given classifier is performing.

$$E_{\text{Bayes}} = 1 - \sum_i \int_{C_i} p(x|c_i)P(c_i)dx \tag{82}$$

where C_i is the feature region where class c_i has the highest posterior $P(c_i|x)$ (Fig. 18, shaded regions). So the Bayes error rate specifies how often the naïve Bayes classifier does not assign the correct class to a feature vector.

In practice, the Bayes error rate can be calculated only for very simple toy problems because for real data neither the distribution $p(x|c_i)$ nor the regions C_i in Eq. 82 are known. However, the concept is still useful since there are a number of methods to calculate an approximation of the Bayes error rate.

5.5 Problem set

1. Derive the gradient of the loss function of logistic regression using paper and pencil.

2. Implement logistic regression model using only elementary programming operations. For your guidance, see the steps below:

   (a) Load the file '05_log_regression_data.npy' in numpy. It contains a hypothetical dataset, where the first two columns reflect the features: length of current residency and yearly income, and the last column contains the labels, i.e., whether a bank loan was granted to an individual or not.

   (b) Because the scale of the two features differs considerably, it is advised to standardize them before fitting. Import the *zscore* function from *scipy.stats* to standardize each of the features.

   (c) Using the *train_test_split* function from *sklearn.model_selection* and a training to validation ratio of 80:20, split the data into training and validation sets.

   (d) Use a scatter plot to visualize the training data set.

   (e) Implement the gradient descent to minimize the loss function.
   - Set the initial parameters $\theta_0 = 0$ or to a small random number.
   - Run the fitting process for 15,000 epochs and a learning rate $\eta = 0.001$.

3. Once the gradient descent algorithm is completed, plot the decision boundary together with the data. Remember that the decision boundary is given by $\hat{p}(x) = \sigma(\theta^T x) = p_0$. If $p_0 = \frac{1}{2}$, the previous condition is equivalent to

$$\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0 \tag{83}$$

4. Plot the training loss vs the epochs. Why does the loss saturate? Why is the asymptotic value not zero?

5. The decision boundary can be shifted by adjusting p_0 to trade off the likelihood of the different errors. Apply different classification thresholds p_0 to obtain the precision-recall curve, F_1 score and ROC curve for the validation set. Based on these measures assess the model performance and determine what would be a reasonable classification criterion.

7 Artificial neural networks

7.1 Components of artifical neural networks

ANN were originally inspired by the architecture and functioning of the brain. Hence the name. The basic building blocks of ANN are neuron-like units. A unit i is connected to multiple input units j's via connections that have different weights w_{ij} (Fig. 20). [Note that the meaning of the indices i and j here are different from the previous usage.] The input units send a signal h_j^{l-1}. The unit sums the weighted inputs

$$a_i^l = \sum_j w_{ij} h_j^{l-1} + b_i^{l-1} \tag{84}$$

and applies an *activation function* ϕ to compute its activation $h_i^l = \phi(a_i)$.

$$h_i^l = \phi^l \left(\sum_j w_{ij}^{l-1} h_j^{l-1} + b_i^{l-1} \right) \tag{85}$$

b_i^{l-1} is the bias of unit i. All weights and biases in all the layers are considered the parameters θ of the neural network,

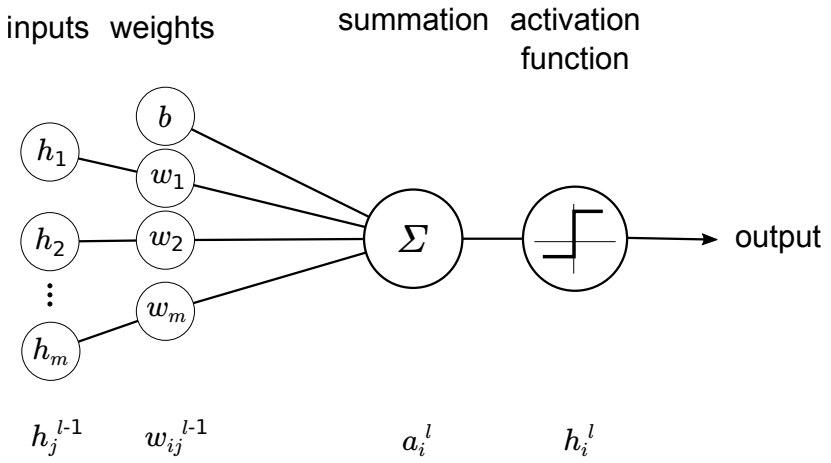


Figure 20: Units used in neural networks.

which are learned during optimization. They are essential for the function that the network represents.

The meaning of the index l differs depending on the architecture of the network. Taking another inspiration from the brain, many units are connected to each other in a network. There are two general classes of neural networks: *feedforward* and *recurrent* networks (Fig. 21). In feedforward networks, all units can be arranged in layers such that information processing proceeds strictly from layer $l-1$ to l (Fig. 21). This means that the activity propagates through the network in one pass, i.e., all units perform the computation in Eq. 85 once for each input. The first layer $l = 1$ is called the *input layer*, their activity is set to the input $h^1 = x$. The intermediate layers are called the *hidden layers*. The last layer $l = m$ is called the *output layer*, their activity represents the prediction of the network. The number of layers m and neurons n^l as well as the activation functions ϕ^l are considered hyperparameters.

In contrast to feedforward networks, information processing in recurrent networks can return to the same unit. In that case, the index l represents time and Eq. 85 describes how activity in one time step determines the network activity in the next. Inputs can be provided at the beginning and the network will continue its computation by itself. It is not so clear when the computations should stop. Generally the computations proceed until either 1. the network itself reaches a state that indicates "stop" or 2. the network reaches a steady state, in which either the activity of the units stops changing, or it might still change in individual units, but the mean remains constant. We'll discuss recurrent networks later and focus on feedforward networks first.

7.2 Activation function

The activation function is inspired by the threshold behavior of biological neurons, but also plays an essential role in ANN. It must be nonlinear in any complex neural network (Fig. 22). More on that below. The model with a threshold function

$$\tau(t) = \begin{cases} 0 & t \leq 0 \\ 1 & t > 0 \end{cases} \tag{86}$$

as activation function, $\phi(t) = \tau(t)$ is called a McCulloch–Pitts neuron (McCulloch & Pitts, 1943).

7.2.1 Nonlinearity

It doesn't make sense to have more than one layer with a linear activation function. Since each layer applies a function f^l on its inputs, we can write the computation of the entire network as:

$$\hat{y} = f(x) = f^m(f^{m-1}(\cdots f^2(x)\cdots)). \tag{87}$$

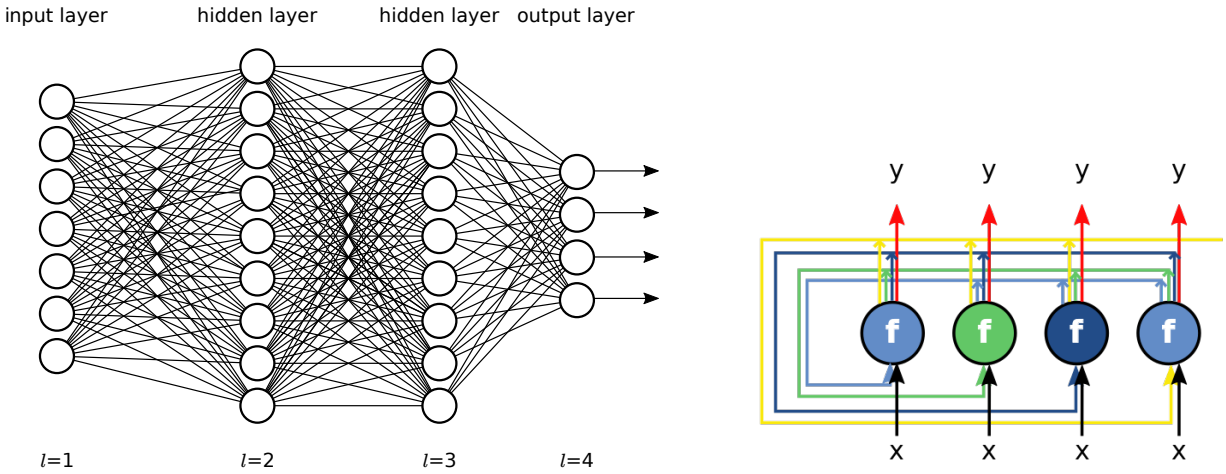
If two activation functions ϕ^l and ϕ^{l-1} were linear, $f^l \circ f^{l-1}$ would also be a linear function, since a linear function of a linear function is also a linear function. In that case, the two layers could be replaced by a single layer without losing any functionality. If all activation functions, were linear then the composition $f^m \circ f^{m-1} \circ \cdots \circ f^2$ would be a linear function of the input x. That means that the network would simply represent a linear function $y = f(x)$. That would be equivalent to linear regression, which we solved before. That's why any meaningful neural network has to use nonlinear activation functions. However, that nonlinearity means that the optimization problem to fit the model parameters is much harder than in linear regression. That's why even though the idea of neural networks sketched above had been around since the 1940's, nobody knew until the 1980's how to train such neural networks.

7.3 Comparison between biological neurons and ANN

Superficially, biological neurons and units in an ANN appear to be rather similar: they both integrate the signal arriving from other units, apply a non-linear activation function and send an output to other units in a network. This is, of course, not an coincidence since ANN were defined the way they were to model biological neurons. However, there are some fundamental differences between them, so that ANN are no longer thought to be good models of the brain.

1. ANN units perform computations and transmit information instantaneously, whereas neurons use complex dynamical processes.

2. ANN units are synchronous, whereas neurons are asynchronous.



(a) Feedforward network.

(b) Recurrent network.

Figure 21: Examples of artifical neural networks.

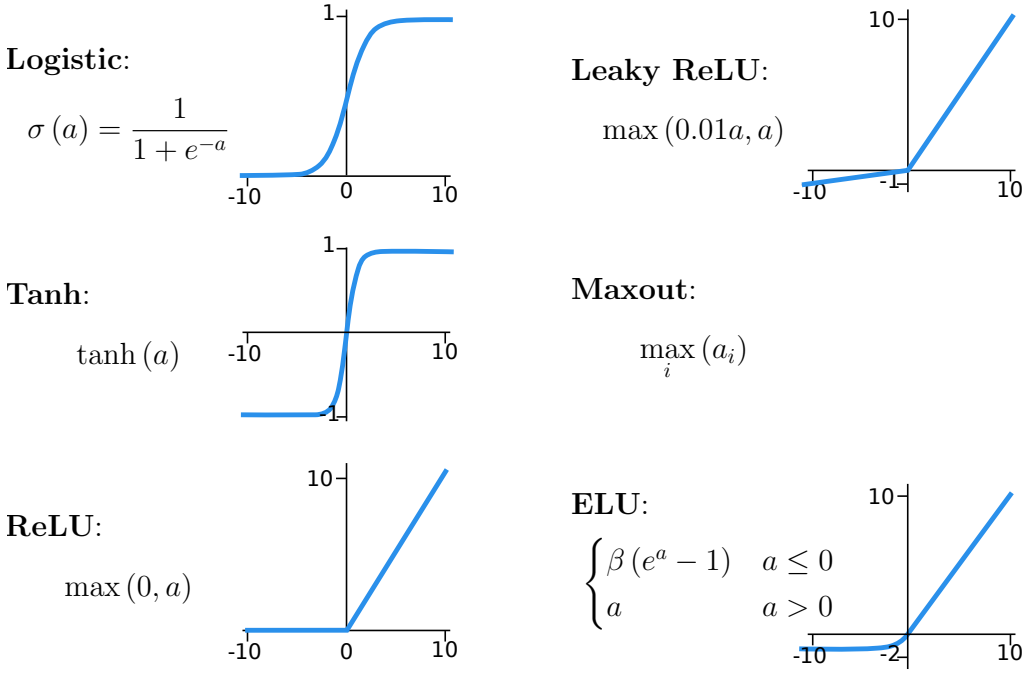


Figure 22: Activation functions. ReLU: Rectified Linear Unit. ELU: Exponential Linear Unit.

3. Summation of inputs in ANN is uniform and linear, whereas it is neither uniform nor linear in biological neurons (*dendritic processing*).
4. The output of an ANN unit is continuous, whereas neurons generate discrete spikes.
5. The weights in an ANN can be changed arbitrarily, e.g. gradient descent, whereas synapses can use only local information and are constrained by biological processes.
6. ANN usually have separate training and test phases, where weight are adjusted and constant, respectively. Plasticity cannot be switched off in a biological neuron.
7. The input-output response of biological neurons change in time (*adaptation*) and is changed by neuromodulators.

Summary:

		Artificial NN	**Biological NN**
1	Computations, Transmission	Instantaneous	Complex dynamics
2	Synchroneous	Yes	No
3	Input summation	Homogeneous, linear	Inhomogeneous, nonlinear
4	Output	Continuous variable	Discrete spikes
5	Weight changes	Arbitrary	Local, biological constraints
6	Training/Testing	Separate	Concurrent
7	Stability over time	Yes	No: adaptation, neuromodulation

7.4 Universal approximation theorem

Why should we bother training a feedforward neural network? Because they are powerful function approximators! In fact, they can approximate any continuous functions, under some mild conditions.

Theorem 1 *Let $\phi : R \to R$ be a nonconstant, bounded, and continuous function (the activation function). Let I_m denote a compact subset of R^m. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given*

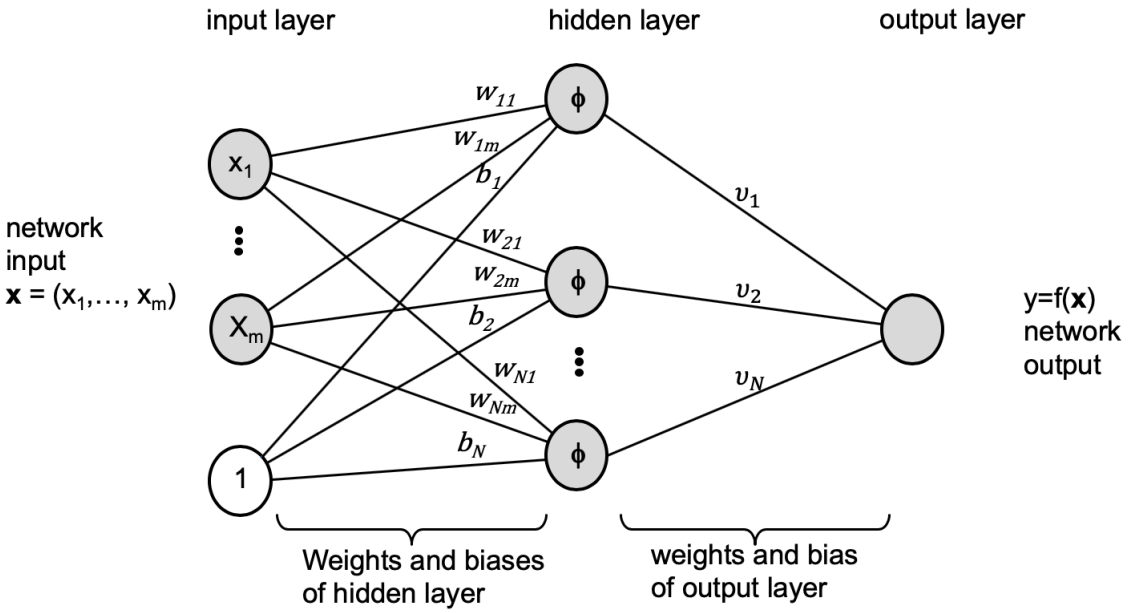


Figure 23: Neural network in the universal approximation theorem with one hidden layer. Note that the activation function is only applied to the hidden layer, the output layer computes only a weighted sum.

any $\varepsilon > 0$ *and any function* $g \in C(I_m)$*, there exist an integer N, real constants* $v_i, b_i \in R$ *and real vectors* $w_i \in R^m$ *for* $i = 1, \ldots, N$*, such that we may define:*

$$f(x) = \sum_{i=1}^{N} v_i \phi\left(w_i^T x + b_i\right) \tag{88}$$

as an approximation of the function g; that is,

$$|f(x) - g(x)| < \varepsilon \tag{89}$$

for all $x \in I_m$*.*

It is hard to believe that a simple 2-layer network with one hidden layer and linear output layer (Fig. 23) is so powerful, but this theorem has been proven mathematically. Unfortunately, the various proofs are not constructive, i.e., they show that the approximation is always possible, but don't tell us how to find the right network parameters. Here, we will only sketch an intuitive proof of the theorem.

1. An ANN with a threshold function as activation function, $\phi(x) = \tau(x + b_i)$, can approximate any function $g(x)$. Two elements can be summed to produce a localized bar Fig. 24. Then pairs can be added to produce multiple bars of different heights at different locations.

2. Any sigmoid function $\phi(x) = \sigma(m_i x + b_i)$ can approximate a threshold function by choosing very large m_i (Fig. 25).

3. Therefore replacing the threshold activation function in the network by a sigmoid function also approximates the function $g(x)$. This substituion increases the error, but since it's bounded, the new network still provides an appropriate approximation.

This argument works in 2 or more dimensions as well (Fig. 26), just the construction will become more and more complex. The above illustration only works for sigmoid-type activation functions, but other constructions are possible for nonsigmoid activation functions.

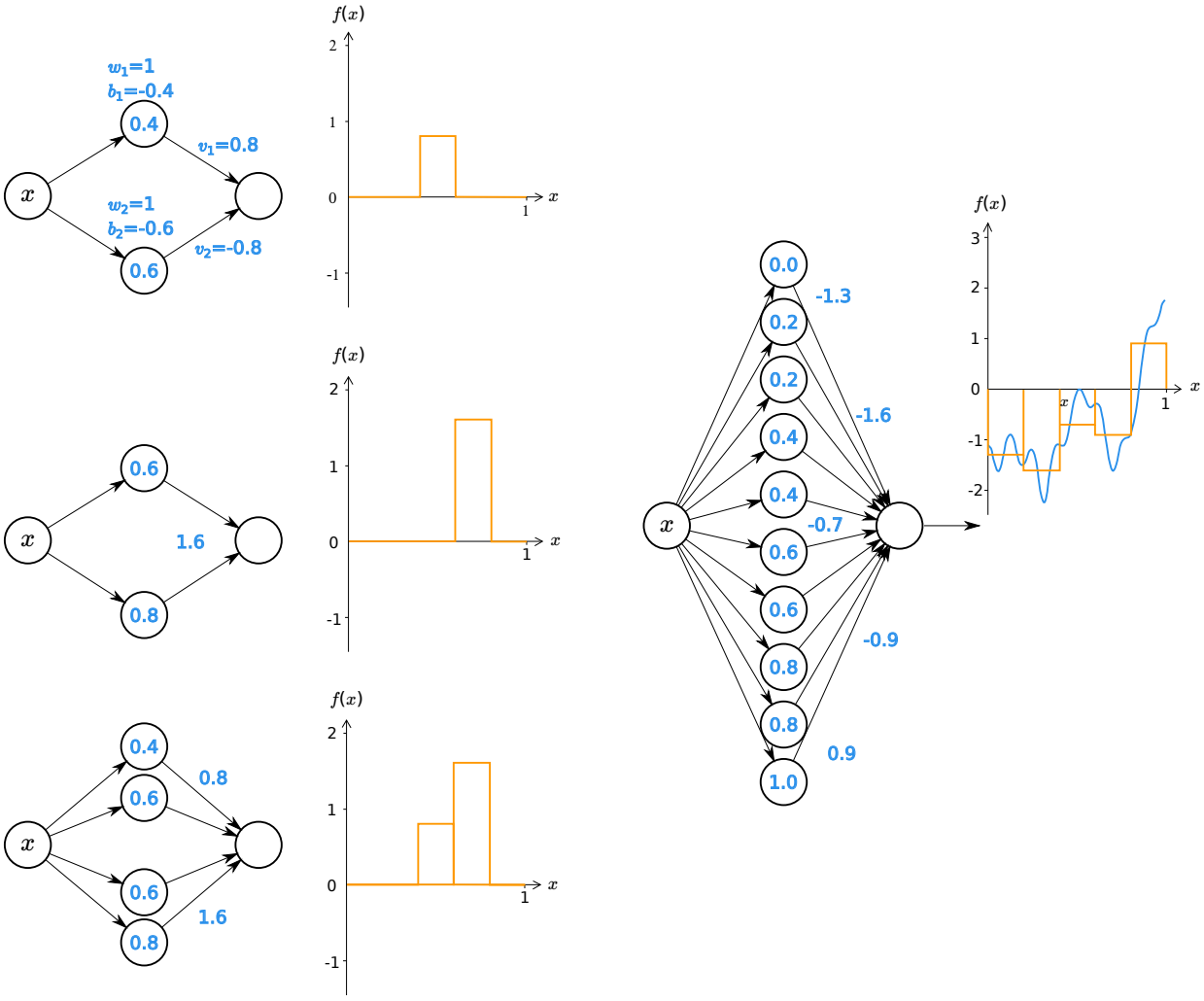


Figure 24: Function approximation in 1-d using splines.

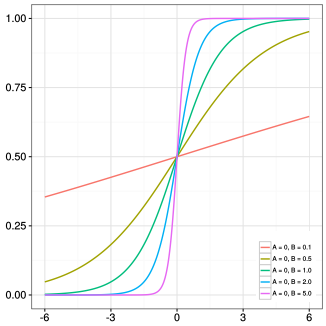


Figure 25: Logistic function with different slopes. Source (Adapted from): `https://commons.wikimedia.org/wiki/File:GeneralizedLogisticB.svg`, User:Bbanerje. Licensed under Creative Commons Attribution-Share Alike 3.0.

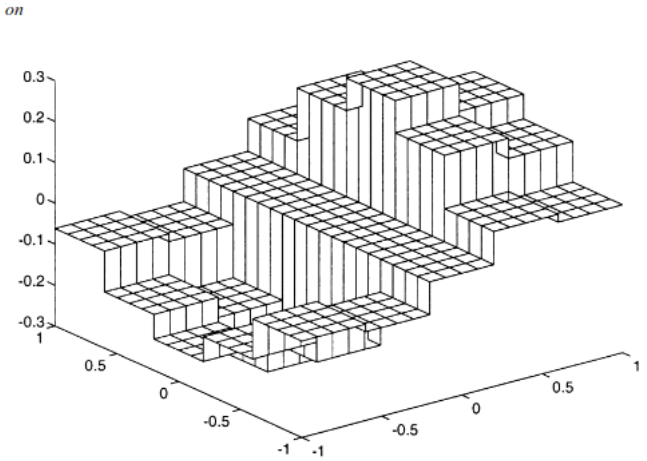


Figure 26: Function approximation in 2 dimensions.

7.5 Problem set

1. The universal approximation theorem states that any function can be approximated by a neural network with one hidden layer.

$$f(x) = \sum_{i=1}^{N} \nu_i \phi\left(w_i^T x + b_i\right) \tag{90}$$

   Implement this network in a Python function using only elementary programming operations. For the activation function $\phi(\cdot)$, use the sigmoid function $\sigma(\cdot)$. For the latter, use the `expit` function from `scipy.special`.

2. Using the previously implemented function $f(x)$, manually set the parameters ν_i, w_i, b_i, N in your program to replicate the output of $f(x)$ shown in the Figures 27a, 27b, and 27c.

   Useful applet that visualizes the impact of the network parameters: `http://neuralnetworksanddeeplearning.com/chap4.html`

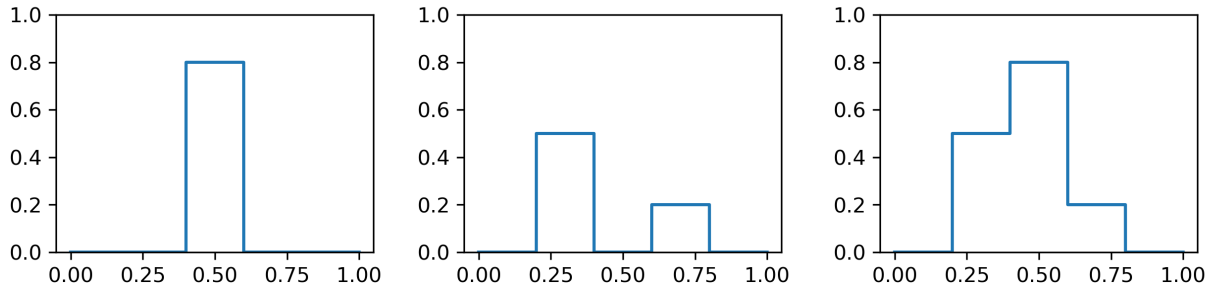


Figure 27: Sample outputs of $f(x)$.

3. Given $g(x) = \sin(2\pi x)$ on the domain $[0; 1]$.

   (a) Approximate $g(x)$ with $f(x)$ using N=10, by computing ν_i, w_i, b_i in a program. Plot the functions $g(x)$ and $f(x)$ in a single figure.

   (b) Compute the residual error $|f(x) - g(x)|$ using elementary programming operations. Repeat the approximation for several larger values of N. Plot the residual error against N.

8 Perceptron

8.1 Photos

See slides for historical photos of hardware implementation and news coverage.

8.2 Model class

The *perceptron* was invented by Frank Rosenblatt (Rosenblatt, 1958). It is a very simple neural network with n input units, and $m = 1$ output unit. The goal of the perceptron is to decide which of two classes $\{-1, 1\}$ the input belongs to. These class labels are chosen here for notational convenience. It is quite common to use 0 for the out-of-class label instead, but that doesn't change the way the perceptron works. The perceptron function maps a high-dimensional input to a binary variable and is therefore useful for solving classification problems. Like logistic regression, the perceptron uses a *linear decision boundary* (Fig. 28).

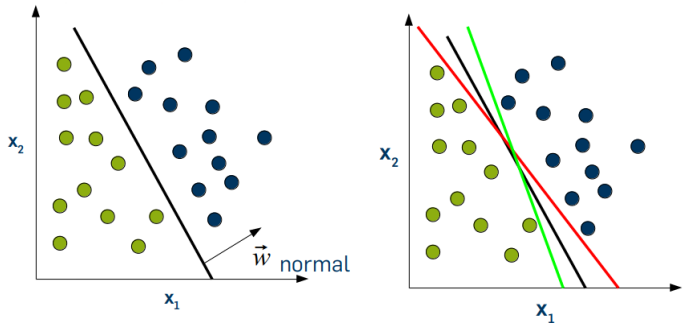


Figure 28: The perceptron implements a linear decision boundary (left). The boundary is given by the equation $w_0 + w_1 x_1 + w_2 x_2 = 0$ or $w^T x = 0$ in vector notation. Solutions found by the perceptron are not unique (right).

$$\hat{y} = f(x) = \begin{cases} -1 & w^T x \leq 0 \\ +1 & w^T x > 0 \end{cases} \tag{91}$$

The parameters of the perceptron are therefore the vector w, which is orthogonal to the decision boundary (Fig. 28). Note that in this formulation w has $n+1$ components, so that w_0 is the bias term and $x_0 = 1$ has to be set in the input.

8.2.1 Example: Boolean AND function

x_1	x_2	output
0	0	-1
0	1	-1
1	0	-1
1	1	1

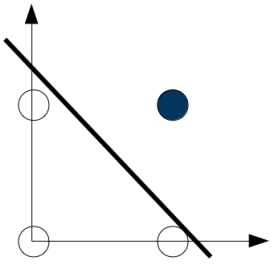


Figure 29: The AND function and a linear decision boundary.

An an exercise, we manually determine a perceptron solution that implements the AND function (Fig. 29). We

can define a decision boundary by

$$x_2 = 1.5 - x_1 \tag{92a}$$
$$0 = 1.5 - x_1 - x_2 \tag{92b}$$
$$= \begin{bmatrix} 1.5 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \tag{92c}$$

Note that we can multiply the weight vector by any scalar and the equation would still hold, i.e., there is an infinite number of solutions to describe the boundary in Fig. 28. However, not all weight vector that generate the right boundary also yields a correct solution. For instance, the one above does not, because for $(1,1)$ it predicts the incorrect output -1. The solution is therefore to multiply the weight with a negative scalar:

$$w = \begin{bmatrix} -1.5 \\ 1 \\ 1 \end{bmatrix} \tag{92d}$$

Note that this solution is not unique. Any

$$w = \begin{bmatrix} z \\ 1 \\ 1 \end{bmatrix} \tag{93}$$

where $-2 < z \leq -1$ is also a solution. In addition, αw with $\alpha > 0$ is also a solution.

8.3 The Perceptron algorithm

The perceptron training algorithm (Alg. 8.1) was the first one suggested for any neural network.

Algorithm 8.1 Perceptron Algorithm (Rosenblatt, 1958)

Require: X, Y
1: initialize parameters w with 0 or small random values
2: **repeat**
3: **for all** data pairs i **do**
4: Calculate the perceptron output: $\hat{y}_i(x_i)$
5: Update weights using the rule: $w \leftarrow w + \eta\,(y_i - \hat{y}_i)\,x_i$
6: **end for**
7: **until** $\frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i| \leq \mu$ or predetermined number of iterations is reached

If classification is already correct, $y_i = \hat{y}_i$ and the weight update term vanished. In other words, the weight are only updated if there is an error. This property is sometimes called error-driven learning.

8.4 Geometric interpretation

To further analyze the perceptron learning algorithm, it is convenient to implement the algorithm a little differently. First, $y_i\hat{y}_i = 1$ if the classification by the perceptron is correct, and $y_i\hat{y}_i = -1$ otherwise. Second, if weights need updating, $y_i - \hat{y}_i = 2y_i$. We can therefore rewrite the perceptron algorithm in the following form:

Algorithm 8.2 Perceptron Algorithm - Variant

Require: X, Y

initialize parameters w with 0 or small random values
initialize errors=0
repeat
  for all data pairs i **do**
    if $y_i \hat{y}_i = -1$ **then**
      Update weights using the rule: $w \leftarrow w + 2\eta y_i x_i$
      errors $\leftarrow$ errors +1
    end if
  end for
until errors$/N \leq \mu$ or predetermined number of iterations is reached

The corrections term always points in the direction of either x_i (if $y_i = 1$), or $-x_i$ (if $y_i = -1$). To illustrate how this helps the algorithm find the category boundary consider the following example in Fig. 30. Since the vector w is

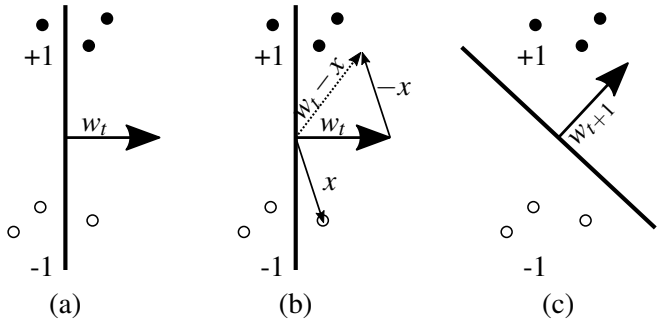


Figure 30: One update in the perceptron algorithm.

normal to the decision boundary, it should point in the direction of the items in the $+1$ class (Fig. 30, right). If this is not the case, then it needs to be rotated towards the class. If the item under analysis x_i is a member of the class, rotating w towards it is a pretty good bet. If the item under analysis x_i is not a member of the class ($y_t = -1$), w is rotated away from the item, i.e. towards $-x_i$.

8.5 Convergence – first steps

Since the perceptron algorithm updates the weights incrementally, an important question is whether the algorithm ever converges, i.e., do the weight updates stop at some point by themselves, if we didn't stop it? That would occur if and only if all items are classified correctly.

Before we proof convergence in general, we make a small first step. Given that a certain item x is misclassified, how many times in a row could the perceptron misclassify the same item, if testing and updating are applied repeatedly to the same item?

For simplicity, we denote the initial weight vector as w_0 and assume that $w_0 \neq 0$. $x \neq 0$ because of the bias component. If the item is misclassified, the weights are updated such that $w_1 = w_0 + 2\eta yx$, then $w_2 = w_1 + 2\eta yx = w_0 + 4\eta yx$, and so on. After k updates, the weight vector $w_k = w_0 + 2k\eta yx$.

If $w \neq 0$ and $yw^T x > 0$, then the item is classified correctly, because the sign of $w^T x_i$ determines the classification $\hat{y}$. So, as long as x is misclassified,

$$yw_k^T x = y\left(w_0 + 2k\eta yx\right)^T x \tag{94a}$$

$$= yw_0^T x + 2k\eta \underbrace{y^2}_{1} x^T x < 0 \tag{94b}$$

Solving for k

$$k < -\frac{yw_0^T x}{2\eta x^T x} \tag{95}$$

This proofs that a given feature vector can be misclassied by the perceptron only a finite number of times. So, if we had only one feature vector in our training set, the perceptron would converge for sure.

8.6 Convergence proof

Minsky & Papert (1969) proofed that the Perceptron will find a solution after a finite number of updates (converge), if the data are linearly separable. The number of updates depends on the data set.

Preliminaries: Assume that data is *linearly separable*, i.e., there exist a solution $\hat{w}$ such that $y_i \hat{w}^T x_i \geq 0 \ \forall i$. Without loss of generality, assume $\|\hat{w}\| = 1$, $\|x_i\| \leq 1$, and $\mu = 0$. Define the *margin* as the shortest distance between the decision boundary and any feature vector, i.e.,

$$\gamma = \min_i |\hat{w}^T x_i| \tag{96}$$

That is, $\gamma \leq |\hat{w}^T x_i| \ \forall i$. Due to Cauchy-Schwarz inequality

$$|\hat{w}^T x_i| \leq \underbrace{\|\hat{w}\|}_{1} \underbrace{\|x_i\|}_{\leq 1} \leq 1 \tag{97}$$

Hence $\gamma \leq |\hat{w}^T x_i| \leq 1 \ \forall i$. We also know that $|\hat{w}^T x_i| = y_i \hat{w}^T x_i$ and $\gamma \geq 0$ because of linear separability. Hence, altogether

$$0 \leq \gamma \leq y_i \hat{w}^T x_i \leq 1 \ \forall i \tag{98}$$

Step 1: Show that with each update w become more similar to $\hat{w}$, i.e., $w^T \hat{w}$ is monotonically increasing with each update. Limit analysis to cases when an error occurs. If no error ocurrs, the algorithm is done.

$$w^T \hat{w} \leftarrow (w + 2\eta y_i x_i)^T \hat{w} = w^T \hat{w} + 2\eta \underbrace{y_i x_i^T \hat{w}}_{\geq \gamma} \geq w^T \hat{w} + 2\eta\gamma \tag{99}$$

Step 2: The previous could be accomplished trivially by increasing $\|w\|$ without getting closer to $\hat{w}$. So, we need to look at how $w^T w$ is changing in each update.

$$\begin{aligned} w^T w \leftarrow (w + 2\eta y_i x_i)^T (w + 2\eta y_i x_i) &= w^T w + 2\eta \underbrace{y_i w^T x_i}_{\leq 0} + 2\eta \underbrace{y_i x_i^T w}_{= y_i w^T x_i} + 4\eta^2 \underbrace{y_i^2}_{=1} \underbrace{x_i^T x_i}_{\leq 1} && (100) \\ &\leq w^T w + 4\eta^2 && (101) \end{aligned}$$

Step 3: Starting with $w = 0$, after a finite number of updates M, in each of which a misclassified element is encountered:

$$\begin{aligned} w^T \hat{w} &\geq 2\eta\gamma M && (102) \\ w^T w &\leq 4\eta^2 M && (103) \end{aligned}$$

Step 4: Derive relationship between the two constraints.

$$2\eta\gamma M \leq w^T \hat{w} \leq \underbrace{|w^T \hat{w}| \leq \|w\| \, \|\hat{w}\|}_{\text{Cauchy-Schwarz inequality}} = \|w\| = \sqrt{w^T w} \leq \sqrt{4\eta^2 M} \tag{104}$$

Hence,

$$M \leq \frac{1}{\gamma^2} \tag{105}$$

This result indicates that convergence is faster if the margin is larger.

8.7 Limitations of the perceptron

- Solutions found by the perceptron are not unique (Fig. 28).
- If the data is not linearly separable, convergence is not guaranteed. Counterexamples are simple functions such as the Boolean XOR function. It is not linearly separable (Fig. 31) and the perceptron algorithm will not converge, if applied to this data. This has become known as the *XOR problem*. If the data is not linearly separable, there will be oscillation (which could be detected automatically).

x_1	x_2	output
0	0	-1
0	1	1
1	0	1
1	1	-1

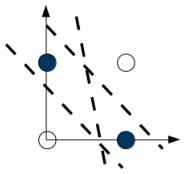


Figure 31: The XOR function (left) is not linearly separable (right) and cannot be learned by the perceptron.

8.8 Problem set

1. Load the Iris dataset from *sklearn.datasets* (at sci-kit learn's website you can find an example for how to load and display the data). The dataset contains measures of the flowers of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured for each sample: the length and the width of the sepals and petals respectively, in centimeters. Extract the first two features (septal length and width) and make a scatter plot from them. Use two different colors to distinguish Iris setosa samples from the rest. Is the data linearly separable?

2. Implement the perceptron algorithm from scratch. You can follow the pseudocode suggested below. Note that, unlike on the version described in the lecture notes, we are recalculating the overall classification error after every weight update. This is somewhat inefficient, but for the sake of this tutorial, it is interesting to see the evolution of the error at a finer scale. Then, use your perceptron to classify Iris setosa based on septal length and width. Plot the resulting decision boundary together with the data points.

Algorithm 8.3 Perceptron Algorithm - Tutorial variant

Require: X, Y
initialize $w = 0$ or small random values
errors = count total number of misclassified items in training set
for epoch = 0 **to** maximum allowed epochs **do**
  for all data pairs i **do**
    if $y_i \hat{y}_i = -1$ **then**
      update weights using the rule: $w \leftarrow w + 2\eta y_i x_i$
      errors = count total number of misclassified items in training set
    end if
    if errors $\leq$ maximum allowed errors **then**
      return solution found!
    end if
  end for
end for

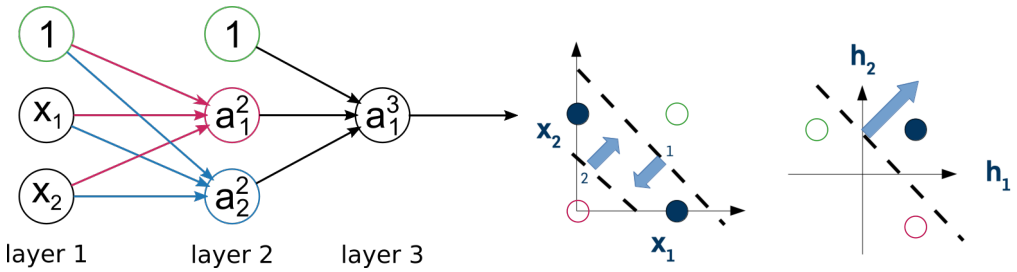
3. Plot the training error after each iteration of the algorithm. Think about what could happen if your design employed a limited number of iterations.

4. Explore the effect of the learning rate on the speed of convergence when you start with $w = 0$. Draw a conclusion.

5. Extract the first and fourth features (septal length and petal width) and make a scatter plot showing Iris virginica vs the rest. What would happen if a perceptron tried to classify Iris virginica?

6. Implement the pocket algorithm, which returns the best set of weights encountered during training instead of the last solution. For each iteration, plot the training error as well as the error for the best solution seen so far. In what way is this modification of the algorithm helpful?

7. (Bonus Question) In the dataset, the samples are ordered by species. However, convergence can be achieved faster if the samples are shuffled. Try running the algorithm after shuffling the data points once in the beginning,



(a) 3-layer network.

(b) Activations of first layer.

(c) Activations of second layer.

Figure 32: A solution of XOR problem using MLP.

and/or shuffling at the beginning of each epoch. Compare the number of iterations it takes for the basic perceptron algorithm to find a solution in each of these cases, and contrast them to the results you got before. You can also look at the effect of shuffling on the pocket perceptron by plotting the evolution of the best classification error over iterations. In order to get more meaningful results, train the classifiers a number of times and plot the mean and standard deviations of the indicated variables.

9 Training multilayer networks

9.1 Multilayer perceptrons (MLP)

Multilayer perceptrons (MLPs) can solve problems that are not linearly separable. For instance, an MLP can solve the XOR problem (Fig. 32). The network consists of two perceptrons in the second layer that both assign the off-diagonal elements to class +1, but also misclassifies one of the diagonal elements as +1 (Fig. 32b).

$$w^1_{1.} = \begin{bmatrix} 1.5 & -1 & -1 \end{bmatrix} \tag{106a}$$

$$w^1_{2.} = \begin{bmatrix} -0.5 & 1 & 1 \end{bmatrix} \tag{106b}$$

By adding an output perceptron in the third layer, we can combine the outputs of the two perceptrons in the second layer. Only if both perceptrons in the second layer return +1, the output perceptron should return +1, otherwise it should return -1 (Fig. 32c).

$$w^2_{1.} = \begin{bmatrix} -1 & 1 & 1 \end{bmatrix} \tag{106c}$$

Even though solutions to the XOR problem, such as the one above, were known since the 1940s, nobody knew how to train these networks until the 1970s and, even then, the solutions weren't widely publicized until Rumelhart et al. (1986). For pratical problems, it is not feasible to construct a solution by hand, as we did for the very simple XOR-function.

9.2 Backpropagation of errors

Data: Inputs and outputs are quite flexible and can be continuous, categorical or binary. Inputs are generally of much higher dimensionality than outputs. To simplify the math, we break with our previous notation here and use x and y without the indices to denote inputs and outputs, respectively. Nevertheless, x, y are implied to take on different values as we iterate through the data.

There are several ways to represent categorical variables: string labels, numerical labels and *one-hot encoding*. String labels are difficult to handle in algorithms, numerical labels lead to unwanted potential correlations. So often the preferred method is one-hot encoding. The class membership in every class c_j is represented by one feature of x, i.e., $x_i \in \{0, 1\}$. So, if the item is an element in class c_j:

$$x_i = \begin{cases} 1 & \text{, if } i = j \\ 0 & \text{, if } i \neq j \end{cases} \tag{107}$$

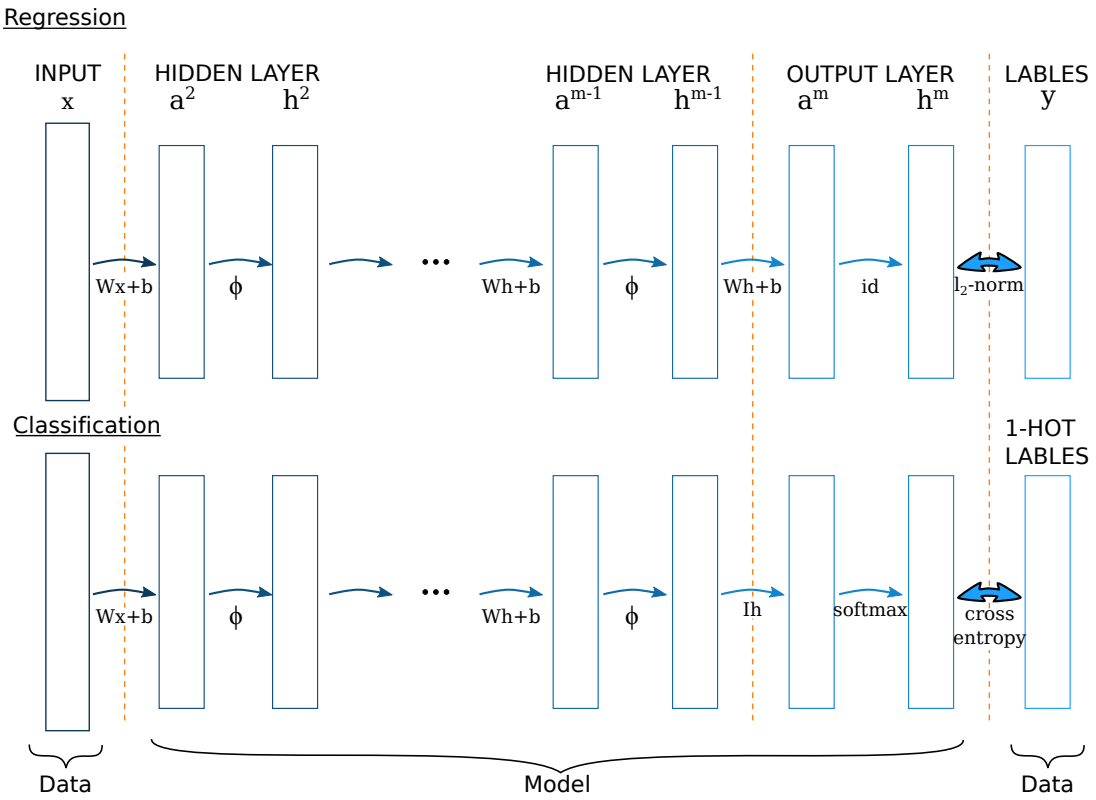


Figure 33: Multilayer networks for regression and classification problems. Note that the two components of each layer are shown separately: the weighted summation leading to a^l and the activation function leading to h^l.

Model class: multilayer neural network (Fig. 33)

$$h_i^{l+1} = \phi^{l+1}\left(\sum_j w_{ij}^l h_j^l + b_i^l\right) = \phi^{l+1}\left(a_i^{l+1}\right) \tag{108}$$

where

$$a_i^{l+1} = \sum_j w_{ij}^l h_j^l + b_i^l \tag{109}$$

The activation functions can differ between the layers. In practice, they are the same for the hidden layers, and a different activation function is used in the last layer depending on the task. For regression, the identity function $\phi^m(a) = a$ is used (Fig. 33, top). For multiclass classification, the *softmax* function is used (Fig. 33, bottom).

$$\phi_i^m(a^m) = \frac{e^{a_i^m}}{\sum_{j=1}^n e^{a_j^m}} \tag{110}$$

In addition, the last layer does not sum over its inputs, instead it simply applies the activation function to each of the inputs, i.e., $a^m = h^{m-1}$. Since the outputs are bounded between 0 and 1, and sum to 1, the outputs can be interpreted as a probability distribution. If the last layer has n^m units that each represent a different class, $\phi_i^m(a^m)$ gives the probability that the input x is a representative of class c_i. The class predicted by the network is simply the class for which the output value is the highest, i.e., $\hat{y} = \arg\max_i \phi_i^m(a)$.

The parameters of the model are all the weights and biases. The hyperparameters of the model include the number of layers m and the number of neurons n^l in each layer l, as well as the activation functions ϕ^l. Additional hyperparameters will be added by the optimization procedure.

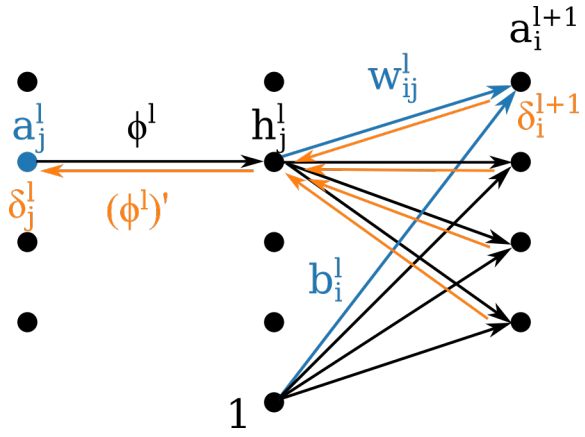


Figure 34: Variables involved in forward and backward pass.

Loss function: $L(\theta, x, y)$ depends on the task. l_2-norm for regression, cross entropy for classification. To simplify the math, we break with our previous notation here and use L to denote the loss for a single data pair x, y.

Optimization: Gradient descent on the loss functions. We'll derive the algorithm in general for any differentiable loss function. To update the weights, we need the derivative of the loss function w.r.t. to the weights: $\frac{\partial L}{\partial w_{ij}^l}$. The weight w_{ij}^l affects the loss function only through the activity of the downstream unit a_i^{l+1} (Fig. 34). That's why, using the chain rule, the gradient w.r.t. the weights can be written as

$$\frac{\partial L}{\partial w_{ij}^l} = \underbrace{\frac{\partial L}{\partial a_i^{l+1}}}_{\delta_i^{l+1}} \underbrace{\frac{\partial a_i^{l+1}}{\partial w_{ij}^l}}_{h_j^l} = \delta_i^{l+1} h_j^l \tag{111a}$$

The gradient w.r.t. the biases is calculated similarly (Fig. 34):

$$\frac{\partial L}{\partial b_i^l} = \frac{\partial L}{\partial a_i^{l+1}} \underbrace{\frac{\partial a_i^{l+1}}{\partial b_i^l}}_{1} = \delta_i^{l+1} \tag{111b}$$

The *error* term δ_j^l is common to both gradients. It affects the loss function only through the activation h_j^l (Fig. 34).

$$\delta_j^l = \frac{\partial L}{\partial a_j^l} = \frac{\partial L}{\partial h_j^l} \underbrace{\frac{\partial h_j^l}{\partial a_j^l}}_{(\phi^l)'(a_j^l)} = \left(\phi^l\right)'\left(a_j^l\right) \frac{\partial L}{\partial h_j^l} \tag{111c}$$

At the output layer $l = m$, $y = h^m$, the loss function is defined in terms of the y's. If the loss function is differentiable w.r.t. y, then the gradient $\frac{\partial L}{\partial h_i^m}$ is known, and the error can be calculated directly:

$$\delta_j^m = (\phi^m)'\left(a_j^m\right) \frac{\partial L}{\partial h_j^m} \tag{111d}$$

For the hidden layers, it is more complicated, since their activations do not enter the loss function directly. But we can derive how the gradient $\frac{\partial L}{\partial h_i^m}$ depends on the errors of the next layer, and derive a *recursive equation*. The key is to note that the activation h_i^l affects all summed activity a_k^{l+1} in the next layer (Fig. 34)

$$\frac{\partial L}{\partial h_j^l} = \frac{\partial L(\ldots, a_i^{l+1}, \ldots)}{\partial h_j^l} = \sum_i \frac{\partial L}{\partial a_j^{l+1}} \underbrace{\frac{\partial a_i^{l+1}}{\partial h_j^l}}_{w_{ij}^l} = \sum_i \delta_i^{l+1} w_{ij}^l \tag{111e}$$

So, the recursive equation for the error is:

$$\delta_j^l = \left(\phi^l\right)' \left(a_j^l\right) \sum_i \delta_i^{l+1} w_{ij}^l \tag{111f}$$

To calculate all the errors, one starts in the output layer $l = m$, and computes the error according to Eq. 111d. Then one traverses the layers of the network in reverse order from $l = m-1$ to $l = 2$ and calculates the error iteratively according to Eq. 111f. Once the errors are known, the gradients can be computed.

Since the error terms are passed from the output layer back through the layers of the network, this algorithm is called *backward propagation of errors* or *backpropagation* for short – or simply *backprop*. Activations h_i^l are propagated forward through the network, and errors are passed backwards through the network.

9.2.1 Backpropagation algorithm

Algorithm 9.1 Backpropagation algorithm – stochastic gradient descent

Require: X, Y

1: Initialize weights w_{ij}^l
2: **repeat**
3: Randomly select a data pair (x, y)
4: *Forward pass*: calculate the network activations h_i^l
5: *Backward pass*: calculate the errors δ_i^l
6: Update all weights $w_{ij}^l \leftarrow w_{ij}^l - \eta \delta_i^{l+1} h_j^l$
7: **until** some stopping criterion

9.2.2 Vector/matrix notation and summary

The gradient for the bias term (Eq. 111b) is same as if we had a weight w_{i0}^l that connects a unit that outputs $h_0^l = 1$ in one layer to unit i in the next layer.
Forward pass

$$h^1 = \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{112a}$$

$$\vdots \tag{112b}$$

$$\tag{112c}$$

$$a^{l+1} = w^l h^l \tag{112d}$$

$$h^{l+1} = \begin{bmatrix} 1 \\ \phi^l(a^l) \end{bmatrix} \tag{112e}$$

$$\vdots \tag{112f}$$

$$\tag{112g}$$

$$a^m = w^{m-1} h^{m-1} \tag{112h}$$

$$h^m = \phi^m(a^m) \rightarrow \hat{y} \tag{112i}$$

Backward pass - errors:

$$\delta^m = (\phi^m)'(a^m) \odot \nabla L(h^m) \tag{113a}$$

$$\vdots \tag{113b}$$

$$\delta^l = \left(\phi^l\right)' \left(a^l\right) \odot \left[\left(w^l\right)^T \delta^{l+1}\right] \tag{113c}$$

$$\vdots \tag{113d}$$

$$\delta^2 = \left(\phi^2\right)' \left(a^2\right) \odot \left[\left(w^2\right)^T \delta^3\right] \tag{113e}$$

$$\tag{113f}$$

	function	**derivative**
logistic	$\sigma(a) = \dfrac{1}{1+e^{-a}}$	$\sigma(a)\,(1-\sigma(a))$
hyperbolic tangent	$\tanh(a) = \dfrac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$\tanh'(a) = 1-\tanh^2(a)$
ReLU	$\phi(a) = \begin{cases} 0 & a \le 0 \\ a & a > 0 \end{cases}$	$\phi'(a) = \begin{cases} 0 & a \le 0 \\ 1 & a > 0 \end{cases}$
leaky ReLU	$\max(\beta a, a);\, 0 < \beta < 1$	$\phi'(a) = \begin{cases} \beta & a \le 0 \\ 1 & a > 0 \end{cases}$
ELU	$\phi(a) = \begin{cases} \beta\,(e^{a}-1) & a \le 0 \\ a & a > 0 \end{cases}$	$\phi'(a) = \begin{cases} \phi(a)+\beta & a \le 0 \\ 1 & a > 0 \end{cases}$
maxout	$\phi(a) = \max_i (a_i)$	$\dfrac{\partial \phi(a)}{\partial a_i} = \delta_{ik}$
softmax	$\phi_i(a) = \dfrac{e^{a_i}}{\sum_{j=1}^{n} e^{a_j}}$	$\dfrac{\partial \phi_i(a)}{\partial a_j} = \phi_i(a)\,(\delta_{ij} - \phi_j(a))$

Table 2: Common activation functions and their derivatives. δ_{ij} is the Kronecker delta function. In the derivative of the Maxout activation function, k is the index of the largest element, i.e., $k = \arg\max_i x_i$.

where we use the Hadamard product $\odot$ to multiply the two vectors component-wise.

Backward pass - derivatives w.r.t. weights:

$$\delta w^l \quad = \quad \delta^{l+1} \times h^l \tag{113g}$$

for $l = m-1, \ldots, 1$. The operator $\times$ is the outer product between two vectors and generates a matrix.

If using stochastic gradient descent (SGD), the forward and backward pass are iterated for each data pair, and the weights are updated for each data pair. If using batch gradient descent (BGD), the weight updates are summed for the entire data set before the weights are changed.

$$w^l \leftarrow w^l - \eta \sum_{\text{batch}} \delta w^l \tag{114}$$

The difference between the two is that in BGD the activations and errors are calculated with the same network parameters for all data pairs, whereas in the former case the network parameters are different each time because they are constantly updated.

9.2.3 Activation function

The backpropagation algorithm derived above works for any loss function as long as it is differentiable. The logistic function $\sigma(t)$ was popular for a long time because its derivative is particularly simple $\sigma' = \sigma(1-\sigma)$. So, the activations calculated during the forward pass $h^m = \sigma(a^m)$ can be saved and used to compute the errors. However, many other activation functions have been, and are being, used (Table 2). Depending on the application, they all have advantages and disadvantages. For instance, sigmoid activation functions are based on the exponential function, which is computationally expensive. More severely, they suffer from the vanishing gradient problem (see below).

9.3 Vanishing/Exploding gradient problems

As the recursive algorithm Eq. 111f propagates the errors down the layers, the error tends to become smaller and smaller. If the network has many layers, the *gradient vanishes*. As a result, the weights in the lower layers are not changed and learning does not converge to a good solution. In some other cases, the errors might increase with every iteration through the layers. As a result, the *gradients explode* and the algorithm diverges. Both scenarios are undesirable.

activation function	uniform distribution $[-r, r]$	normal distribution	name
tanh	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$	Xavier
logistic	$r = \sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$	Xavier
ReLU	$r = \sqrt{\frac{6}{n_{\text{inputs}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{inputs}}}}$	He et al. (2015)

Table 3: Different weight initialization schemes for different activation function.

The source of this problem is a combination of using the logistic activation function and initialzing weights with random normal values with zero mean and standard deviation of 1 (Glorot & Bengio, 2010). Activation keeps increasing with every layer. The logistic function saturates for very large negative or positive values and, therefore, the derivative is nearly zero. Once the network reaches this point, the weights cannot be reduced any more.

9.3.1 Xavier and He initialization

One solution is to make sure that the activations and the gradients do not die out or explode when passing through the network layers. This could be achieved by making sure that the output magnitude is roughly equal to the input magnitude, which in turn can be achieved by carefully initializing the weights. When using the logistic activation function, Glorot & Bengio (2010) suggested to initialize weights randomly drawn from a:

1. normal distribution with zero mean and standard deviation $\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}}$, or
2. uniform distribution between $\pm\sqrt{\frac{6}{n_{\text{inputs}} + n_{\text{outputs}}}}$

The parameters n_{inputs} and n_{outputs} are also known as *fan-in* and *fan-out* factors. This method is named after the first name of the first author *Xavier initialization*. The optimal weight initialization depends on the activation function (Table 3).

9.3.2 Nonsaturating activation functions

The ReLu function does not suffer as much from vanishing gradients, because it doesn't saturate for positive input values. A disadvatage is that the ReLU function is not differentiable at zero. In practice, the derivative is simply set to zero, i.e., $\phi'(0) = 0$, which works remarkably well in practical applications. It also generates many zero values in the activations, leading to *representational sparsity*. However, the ReLu function saturates for negative inputs, in which case both the activation and the derivative are zero. Because of the zero derivative, the incoming weights will not be updated and therefore the inputs to the unit are not going to become stronger. If this happens for all inputs, the unit is effectively removed from the network. This effect is called the *dying ReLU problem*.

To avoid both the vanishing gradient and the dying ReLU problems, an alternative is to use activation functions that do not saturate. The leaky ReLU or the ELU activation function use the same activation function when the input is positive. However, for negative inputs, they take on small negative values and the derivatives are non-zero. For $\beta = 1$, ELU is differentiable at zero and therefore avoids large jumps in the gradient around zero. The disadvantage of ELU is that is uses the costly exponential function.

9.3.3 Gradient clipping

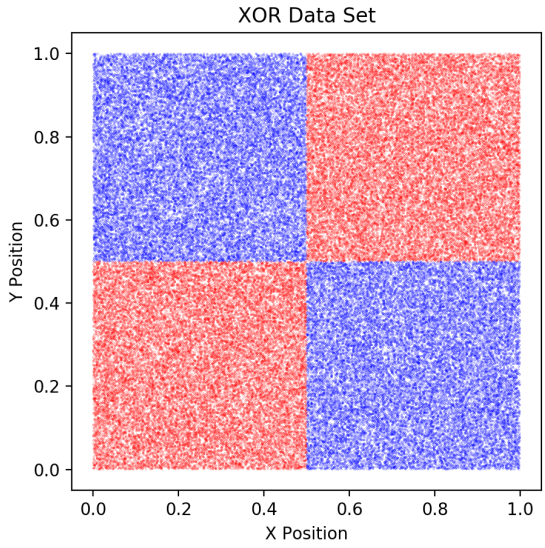
Using careful weight initialization avoids the vanishing/exploding gradient problem at the beginning of the training, however, it might reappear later during training. To avoid these issues during training one can use *gradient clipping*. In this method, the gradients are limited to a certain range $[-g, g]$. If any gradient exceeds g, the gradient is set to g. Likewise, if the gradient is smaller than $-g$, it is set to $-g$.

9.4 Problem set

1. **Backpropagation algorithm**

   A skeleton of a class that models a multi-layer perceptron (MLP) with one hidden layer is provided in *prob_mlp.py*. The *main* function of the provided code already loads the data set and splits it into training and test sets (Note: While you are actively developing and testing your code, use a smaller training set to speed things up.)

   (a) Implement the function *train* that trains the MLP's weights with backpropagation using stochastic gradient descent.



   The first two columns of the data set contain x and y position of the data point while the third column represents the class identiy. Class identities are coded with $+1$ and -1. Use the following parameters for your network:

   - Hyperbolic tangent as activation function for hidden and output layers.
   - Use the squared error (SE) as loss function (Note: for a classification problem, one would normally use the cross-entropy loss, but here we will stick to the squared error as it is easier to compute.)
   - Hidden layer with 64 units. Also add a bias unit to the hidden layer.

   (b) Train the MLP on the provided XOR data set (*xor.npy*) using a learning rate of 0.005.

   (c) Implement a *predict* function in the MLP class. This function should compute the mean squared error (MSE) on the test set and predict the class labels to compute the misclassified points (using 0 as the threshold).

   (d) Repeat the training for one epoch 100 times (reinitializing the weights in between runs).

      i. Record the MSE for each run and compute the mean and standard deviation of the error across all the runs.

      ii. Also record the misclassified points for each run using the test function. This will be useful for Problem 3.

2. **Xavier initialization**

   The MLP class provided initializes weights by drawing them from a normal distribution with zero mean and standard deviation of 1. Extend the class so that weights can be optionally initialized using Xavier initialization. Test the implementation identically to Problem 1, changing only the initialization type. Plot the mean and SD of the error together with those recorded from Problem 1. Which initialization performs better? Is that expected?

3. **Misclassifications**

   (a) Count the misclassifications that you recorded from Problems 1. and 2. in separate 100x100 bins and make heatplots. How does the choice of weight initialization method affect the distribution of the MLP's misclassifications?

   (b) Instead of the hyperbolic tangent use ReLU activations for the hidden layer and repeat the previous tasks. Also change the number of hidden units to 20. How does the MSE for the weight initialization methods compare to when using the hyperbolic tangent as activation function? How does the distribution of the MLP's misclassifications differ qualitatively?

4. **Training ANN in Keras**

   The above tasks can also be solved using standard machine learning libraries. For example, you could use `sklearn.neural_network.MLPClassifier` to construct an identical architecture. We will concentrate on using `keras` in this exercise. The file *prob_keras.py* contains code which creates an MLP identical to Problem 1. The code is briefly explained below :

   - We will use the Sequential model API, which enables the creation of a stack of layers (from `keras.models` import `Sequential`)
   - Layers are added to a keras model using the method `model.add`. The type of layer we will use is the *Dense* layer, which means that each unit is connected to *all units* in the previous layer, as in the case of an MLP. The input layer is specified directly as an argument, `input_dim` in the hidden layer. All layers in keras are available from `keras.layers`
   - The activation function is specified using *Activation* layers in the code. It is also possible to use the argument `activation` directly instead. We must also specify the optimizer to use for training (SGD) which can be imported from `keras.optimizers`.
   - Finally, before training, the model must be configured for training by calling `model.compile`. The model is trained for one epoch using `model.fit`.
   - Once the model has been trained, we can evaluate the model's performance on the test set. In the code provided, we have used `test_on_batch`. You can also use the `evaluate` function to do this.

   (a) Modify the above code to directly perform classification: the dataset provided uses class labels -1 and +1. Modify the class labels of the training set by using a one-hot encoding instead. The built-in function `keras.utils.to_categorical` can be used to accomplish this. Next, change the output layer to have one neuron per class, and use a softmax activation to compute the class probabilities. Repeat the training using the cross-entropy loss instead of MSE.

   (b) Evaluate the loss and predict the classes on the test set. Also compute the MSE and plot it as before. Compare it to the MSE in assignment 1. Why don't we compare the cross-entropy loss directly with the MSE?

10 Deep neural networks

A deep neural network is simply a neural network with many layers. There is no standard definition of when a network counts as deep. Sometimes a network with $m = 4$ layers is already considered deep. The main challenges with deep neural networks, as compared to shallow neural networks, is that 1. they have a very large number of parameters that need to be trained, and 2. the gradients can vanish/explode as errors are progagated through the many layers. Therefore, special techniques have been developed to deal with these specific issues. These techniques can mostly be applied to shallow networks as well to make training more efficient.

10.1 Convolutional neural network

The more layers there are in a network and the more units there are in a layer, the more complex a function the network can represent. However, one cannot train multiple, fully-connected layers because there are simply too

many parameters. Hence, *convolutional neural networks (CNN)* were developed to address these issues. This type of network was inspired by two properties of visual information processing in the brain since these networks are mostly used for image processing.

First, lower layers of the visual processing hierarchy process lower-level representations that are highly variable and hence less informative. For instance, when an image shows a line, the pixel values change with illumination and the specific values of the pixels don't really matter. By contrast, higher layers of the visual processing hierarchy represent higher-level features that are more abstract and more invariant such as the orientation of bar, shapes or semantic categories. Second, units in higher layers process information from a larger visual field, i.e., their receptive fields are larger. In CNN, these two properties, feature extraction and integration, are modelled by *convolution* and *max pooling*, respectively. Together these two operations can reduce network complexity and extract higher-order features that are useful for image recognition. An example of these increasingly complex and abstract features can be seen in Fig. 35.

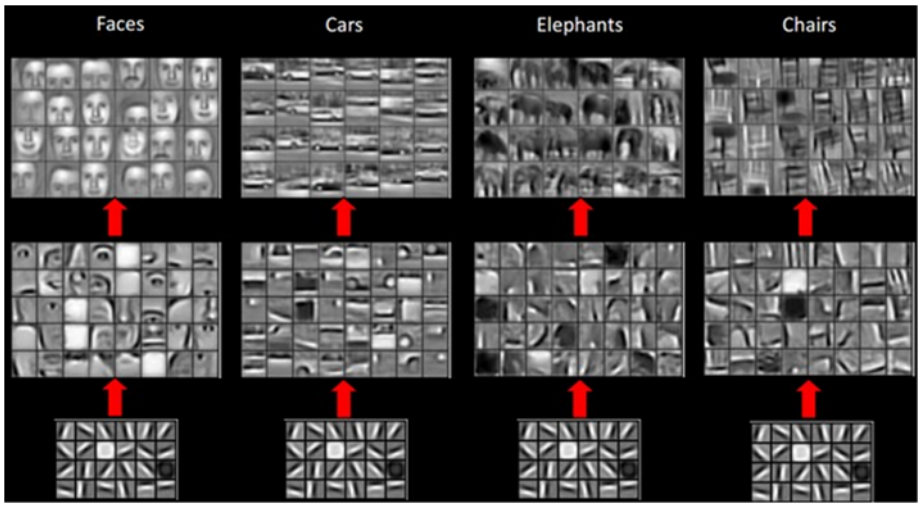


Figure 35: Network represents increasingly complex and abstract features (Siegel et al., 2016, Fig.3).

10.1.1 Convolution

Mathematically, a *convolution* between two one-dimensional functions f and g is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau \tag{115}$$

One of the functions, e.g. f, is usually considered a *filter* and is a localized function. Discrete variant

$$(f * g)_i = \sum_{i'=0}^{n-1} f_{i'} g_{i-i'} \tag{116}$$

where n is the size of the convolution filter. For instance, calculating a moving average of a time series g (Fig. 36) is like convolving it with a box filter

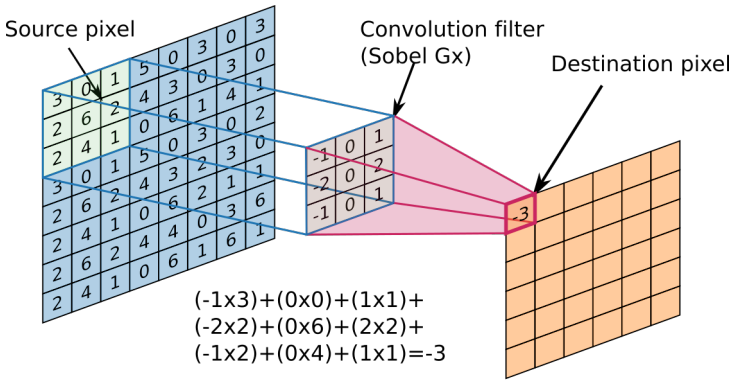
$$f_i = \begin{cases} \frac{1}{n} & 0 \le i < n \\ 0 & i \ge n \end{cases} \tag{117}$$

Definition of convolution in 2-d:

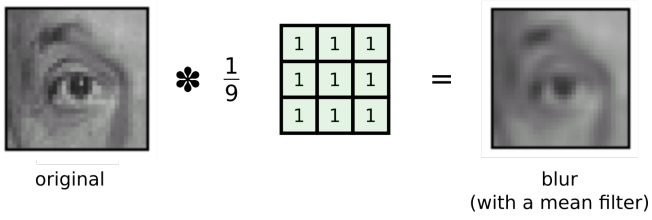
$$(f * g)_{i,j} = \sum_{i'=0}^{n_v-1} \sum_{j'=0}^{n_h-1} f_{i',j'} g_{i-i',j-j'} \tag{118}$$

where n_v and n_h are the sizes of the convolution filter in the vertical and horizontal direction, respectively. Example of 2-d convolution Fig. 37a. Just like in 1-d, convolution can be used for smoothing in 2-d, although in image processing it's frequently called blurring instead (Fig. 37b).

Figure 36: Moving average in 1-d.



(a) Illustration of 2-d convolution.



Adapted from D. Lowe

(b) Blurring convolution.

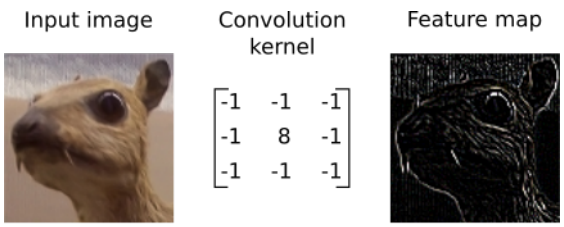


Image credits: Michael Plotke;
License: https://creativecommons.org/licenses/by-sa/3.0/deed.en

(c) Outline convolution.

Figure 37: 2-d convolutions in image processing.

Convolutions can be used to extract more abstract features, such as outlines (Fig. 37c) or edges. The latter is called *edge detection*. This is useful for image recognition because these features are more abstract, invariant and informative. In addition, the number of parameters is reduced from the number of pixels in the image to the number of elements in the filter.

Usually, there are multiple, n_f^l sets of features that are calculated in one convolutional layer, e.g., to represent a color image in RGB space the red, green and blue channels are separately encoded a two dimensional matrix. Each color channel contributes a feature map.

Note that the convolution is defined with a minus sign $i - i', j - j'$. As a result, there is a disconnect between the visualization of the convolution filter $f_{i',j'}$ and its effect in the convolution. In mathematics, the minus sign is required so that the convolution operation is symmetric in f and g. However, in image processing, this creates a lot of confusion. That's why we use a plus sign in *convolutional layers* in an ANN, even though the unit doesn't, strictly speaking, perform a convolution operation. It is called a *correlation* operation. In 2-d, the difference between the kernels in convolution and correlation is a rotation by 180°.

$$a_{i,j,k}^{l+1} = b_k^l + \sum_{i'=0}^{n_i^l-1} \sum_{j'=0}^{n_j^l-1} \sum_{k'=1}^{n_f^l} w_{i',j',k,k'}^l h_{is_v+i',js_h+j',k'}^l \tag{119a}$$

To generate the output, an activation function is applied to each unit.

$$h_{i,j,k}^l = \phi^l \left(a_{i,j,k}^l \right) \tag{119b}$$

- $w_{i',j',k,k'}^l$ convolution kernel. Note that the indices i and j have very different meaning from earlier use; they index the units within the input layer/ kernel in the vertical and horizontal dimension, respectively. Earlier, they referred to units in the input and output layers, respectively. This change is necessary because the weights in CNNs are not uniquely connecting pairs of units (see below).
- n_v^l, n_h^l: size of filter in vertical and horizontal direction, respectively
- n_f^l: number of feature maps
- s_v^l, s_h^l: strides in vertical and horizontal direction, respectively
- b_k^l: overall brightness of the feature map, bias
- m_v^l, m_h^l: size of image/feature map in vertical and horizontal direction, respectively

In Eq. 119a, there is a summation across the feature dimension, no convolution.

The size of the resulting feature map. Another difference to strict convolution is when the strides s_i^l and s_j^l are larger than 1. This reduces the size of the image and is useful when memory during training is limited. This is equivalent to computing the entire convolution, but skipping $s_i^l - 1$ (respectively $s_j^l - 1$) output units when generating the output. That's why the size of the output layer is roughly $s_i^l s_j^l$-times smaller than the full convolution. Image reduction with pooling layers (see below) appear to work better than using larger strides to reduce the size of the image representation. It is therefore more common to use $s_i^l = s_j^l = 1$.

Even when using 1 for the stride, the size of the feature map is smaller than the size of its input: $m_v^{l+1} = m_v^l - n_v^l + 1$ and $m_h^{l+1} = m_h^l - n_h^l + 1$. That is because the convolutional filter is not allowed to stick out. This reduction is undesirable in image processing because that means that the image representation will shrink in every layer and it's always the information around the edges that is lost. To avoid this loss of information, *padding* is used. The image in the input, or the feature map in the hidden layer, is enlarged by $(n_v^l - 1)/2$ and $(n_h^l - 1)/2$, respectively, around the edges. The additional pixels or the additional hidden units are either set to 0 *zero padding* or the new pixels are filled with copies of the neighboring edge. Padding the input preserve the size of the image representation after the convolution and the image information around the edges.

Weight sharing. In a fully connected network, all the units in layer l are connected to all the units in layer $l+1$. Hence, there are $n^l \times n^{l+1}$ weights that need to be estimated. Each weight connects exactly one pair of input and output units. The reason why CNN use dramatically fewer parameters is *weight sharing*. There is one convolution kernel $w_{i',j',k,k'}^l$ that is shared between all units to compute the features in each layer. In addition, the kernel is usually much smaller than the input image itself, further saving on parameters.

Additional material:

- "Convolutional Neural Network – In a Nut Shell" by Muhammad Rizwan
- "An intuitive guide to Convolutional Neural Networks" by Daphne Cornelisse

10.1.2 Backpropagation in convolutional neural networks

Because of weight sharing, each entry in the weight matrix, i.e. convolution kernel, $w^l_{i',j',k,k'}$ influences every unit's activity in the next layer $h^{l+1}_{i,j,k}$, therefore the gradient has to sum over all these influences.

$$\frac{\partial L}{\partial w^l_{i',j',k,k'}} = \underbrace{\sum_{i=1}^{m_v^{l+1}} \sum_{j=1}^{m_h^{l+1}}}_{\text{weight sharing}} \underbrace{\frac{\partial L}{\partial a^{l+1}_{i,j,k}}}_{\delta^{l+1}_{i,j,k}} \underbrace{\frac{\partial a^{l+1}_{i,j,k}}{\partial w^l_{i',j',k,k'}}}_{h^l_{is_v+i',js_h+j',k'}} = \sum_{i=1}^{m_v^{l+1}} \sum_{j=1}^{m_h^{l+1}} \delta^{l+1}_{i,j,k} h^l_{is_i+i',js_j+j',k'} \tag{120a}$$

$$h^l_{i^*,j^*,k'} \qquad w_{i',j',k,k'} \qquad a^{l+1}_{i,j,k} \tag{120b}$$

If the loss function is differentiable w.r.t. y, then the gradient

$$\delta^m_{i,j,k} = \frac{\partial L}{\partial h^m_{i,j,k}} \frac{\partial h^m_{i,j,k}}{\partial a^m_{i,j,k}} = \frac{\partial L}{\partial h^m_{i,j,k}} (\phi^m)' \left(a^m_{i,j,k}\right) \tag{120c}$$

is defined. For the hidden layers, we have to sum over all activations in layer $a^{l+1}_{i,j,k}$ through which the activation $a^l_{i^*,j^*,k'}$ can propagate to the output. Eq. 119a shows that this occurs if the indices satisfy certain relationships:

$$i^* = is_v + i' \qquad , i' = 0, \ldots, n_v - 1 \tag{120d}$$

$$j^* = js_h + j' \qquad , j' = 0, \ldots, n_h - 1 \tag{120e}$$

Because the convolution limits the spatial range through which one pixel can affect the next layer, we only have to sum over the width of the filter:

$$\delta^l_{i^*,j^*,k'} = \frac{\partial L}{\partial a^l_{i^*,j^*,k'}} = \sum_{i'=0}^{n_v^l-1} \sum_{j'=0}^{n_h^l-1} \sum_{k=1}^{n_f^{l+1}} \frac{\partial L}{\partial a^{l+1}_{i,j,k}} \frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} \tag{120f}$$

$$= \sum_{i'=0}^{n_v^l-1} \sum_{j'=0}^{n_h^l-1} \sum_{k=1}^{n_f^{l+1}} \delta^{l+1}_{i,j,k} \frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} \tag{120g}$$

The last derivative is:

$$\frac{\partial a^{l+1}_{i,j,k}}{\partial a^l_{i^*,j^*,k'}} = \frac{\partial a^{l+1}_{i,j,k}}{\partial h^l_{i^*,j^*,k'}} \frac{\partial h^l_{i^*,j^*,k'}}{\partial a^l_{i^*,j^*,k'}} \tag{120h}$$

$$= \left(\phi^l\right)' \left(a^l_{i^*,j^*,k'}\right) w_{i',j',k,k'} \tag{120i}$$

$$\delta^l_{i^*,j^*,k'} = \left(\phi^l\right)' \left(a^l_{i^*,j^*,k'}\right) \sum_{i'=0}^{n_v^l-1} \sum_{j'=0}^{n_h^l-1} \sum_{k=1}^{n_f^{l+1}} \delta^{l+1}_{(i^*-i')/s_v,(j^*-j')/s_h,k} w_{i',j',k,k'} \tag{120j}$$

Notice that the equations for backpropagation in CNN are very similar to the one for backpropagation in fully connected networks. This is because both use only linear operations. The differences between the networks consists in how units are indexed and how inputs are summed. That's why the differences in the backpropagation equations differ in precisely these aspects.

Additional material:

- "Backpropagation in Convolutional Neural Networks" by Jefkine Kafunah

10.1.3 Pooling

A convolution layer extracts features from the input image, however, the size of the feature map remains as large as the input image itself, if padding is used. Since we want the network to learn more abstract, lower-dimensional representations of the input, we have to reduce the size of the image representation. A *pooling layer* accomplishes this by reducing the inputs in a rectangluar, mostly square, receptive field to a single number. Most commonly, the receptive fields are fairly small such as 2×2 with a stride of 2 because usually multiple pooling layers are used in a deep network and the image size reduces exponentially. The input-output function that appears to work the best in most situation is the maximum. This is a so-called *max pooling* layer (Fig. 38a). Other choices include averaging over the inputs (*average pooling*.

(a) Max pooling

(b) Average pooling

Figure 38: Example of pooling operations.

Another desirable result of pooling is *translational invariance*. The identity of objects is invariant to translation (and rotation and many other image transformations). Humans and other animals have learned these invariances and can recognize objects under these transformations. However, the mathematical representation of a shifted image I' is very different from the original image I, i.e., a distance measure such as $l_2(I, I')$ would be quite large – in fact in many cases it would be larger than changing the identity of the object. The ANN has to learn these invariances, i.e. map the very different input images I and I' to the same label. Pooling helps create these invariances because the output of the pooling unit is not sensitive to the spatial location of the information within the receptive field.

Since a pooling layer has no parameters, there is nothing to be learned. The pooling layer does have to keep track of from which input units the output came from in the forward pass and route the error to those input units during the backward pass.

- **max pooling**: only the winning unit contributes to the output, so the error is passed onto the winning unit, the other inputs units receive zero error.
- **average pooling**: the error is divided by the size of the receptive field and this error is routed to all input units alike

10.1.4 Putting it all together

A deep neural network generally consists of multiple instances of convolutional layers and pooling layers (Fig. 39). Because of how the activations are defined in the forward pass and how the errors are computed in the backward pass, each layer l in a deep network interacts only with the preceding layer $l-1$ and the following layer $l+1$. That's why ANN are modular and any type of layer can be combined with any other type of layer, as long as the fan-in and fan-out factors match.

For image recognition, the architecture of the deep network is inspired by the architecture of the visual systems in the brain. The lower layers consist of convolutional and pooling layers that generate feature maps that are local, compressed and abstract representations of the input image. The highest layers consist of fully connected layers that integrate all feature information to generate the classification (Fig. 39).

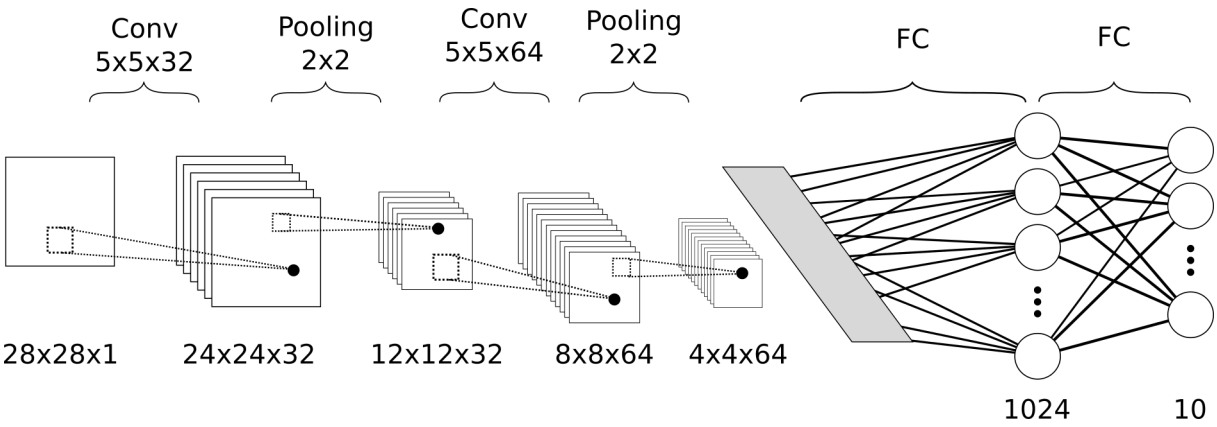


Figure 39: Example of convolutional neural network (CNN) with seven layers.

10.2 Recent technological breakthroughs

CNN have become hugely popular only in 2012. However, the ideas behind CNN have been known since the 1990's (Lecun et al., 1998). What happened in that time? Earlier CNN were used, but not very successful in solving image recognition problems. Their performance was far from human performance (Fig. 40). This changed because of three technical advances. First, CNN have to be trained on massive amounts of data. These were generated by Fei-Fei Li

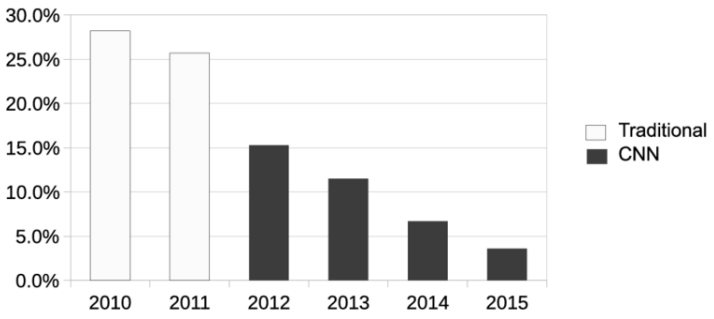


Figure 40: Performance on ImageNet competition over the years (Bottou et al., 2018, Fig. 2.4). A human achieves an error rate of roughly 5%.

and colleagues (Deng et al., 2009) in their ImageNet database. Using Amazon Mechanical Turk they had more than 14 million images hand-annotated by people to indicate what objects are pictured in more than 20,000 categories. The ImageNet competition started 2010. The second technical advance were graphical processing units (GPUs), which made it possible to train deep neural networks on massive amounts of data. In 2012, a deep convolutional neural net called AlexNet (Krizhevsky et al., 2012) trained using GPUs achieved an error rate of 16%, which was a huge improvement over the 2nd best-performing algorithm. That achievement sparked huge interest in CNNs. Nevertheless, AlexNet was not the first deep CNN trained using GPUs. That distinction goes to DanNet (Ciresan et al., 2011), which AlexNet is very similar to.

The wide-spread popularity of deep learning would not be possible without the third technological advance: the availability of software frameworks that automate the otherwise tedious implementation, training and testing of deep networks. Several frameworks have been developed over the years and include: Caffe, CNTK, TensorFlow, Theano, and Torch. To make the usage even simpler, Keras provides another abstraction layer on top of CNTK, TensorFlow, or Theano.

10.3 Regularization

Large neural nets and small datasets usually lead to overfitting. The model learns the statistical noise in the training data and cannot generalize to new data. Earlier in this class regularization was introduced as a method to avoid overfitting. A regularization term was introduced into the loss function to penalize large values of the model parameters.

$$L(\theta, X, Y) = L_0(\theta, X, Y) + \lambda R(\theta_i) \tag{121}$$

This is also known as *shrinkage*. Some regularization techniques discussed below are quite different from shrinkage. Why are these techniques then considered forms of regularization? Because they impose features onto the model that we know that the model should posses: in the case of shrinkage we "know" that the parameter values should be fairly small. In a similar vein, we can introduce other types of knowledge to constrain the model. Goodfellow et al. (2016) therefore propose:

> Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

In other words, regularization reduces model complexity.

How to detect that your network is overfitting? Traditional methods for model selection are difficult to use with deep networks because they have so many parameters and take a long time to train. Instead, to catch overfitting in a deep network, look at the accuracy of the model on the training data (training accuracy) and the accuracy on the test data (test accuracy) over the training steps Fig. 41. The accuracy on the training set will always increase, when you use gradient descent. For stochastic gradient descent that will only be true on average. If the network starts to overfit the data, the test accuracy will diverge from the training accuracy and might even start to drop, if overfittig is severe.

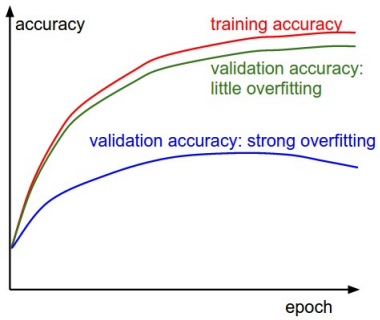


Figure 41: Detecting overfitting in your dataset.

10.3.1 Early stopping

A very simple strategy to avoid overfitting the data, is to stop training process when the test accuracy starts to drop. One way to implement this is to compute test accuracy every T training steps. If the current test accuracy is lower than the previous one, then return the previous snapshot of the network and stop the optimization. If not, store the current test accuracy and snapshot of the network and repeat.

10.3.2 Weight decay

Weight decay sounds like a new regularization method, but it is mathematically equivalent to gradient descent on an l_2 regularization term. The term describes the update equation that is used: after each update, the weights are multiplied by a factor $\beta < 1$, which means that the weights "decay" with time and, hence, prevents the weights from getting too large.

$$w \leftarrow \beta w \tag{122}$$

Assuming that the original loss function that is being minimized is L_0, one can add an l_2 regularization term to form a new loss function:

$$L = L_0 + \frac{1}{2}\lambda \sum_i \theta_i^2 \tag{123}$$

This new loss function will penalize large values of the parameters and therefore limit overfitting (see earlier lecture). The regularization parameter λ determines how the original loss function L_0 is traded off with the regularization term.

The gradient of the new loss function is:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L_0}{\partial \theta_i} + \lambda \theta_i \tag{124}$$

When using this gradient in gradient descent, the second term means that the weights should decrease proportionally to its value, which is equivalent to multiplying it by a factor $\beta < 1$.

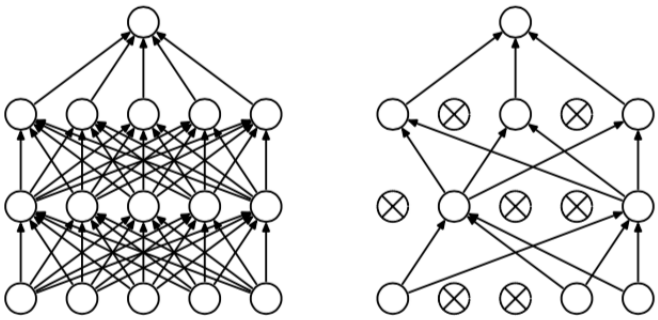
l_2 regularization, or weight decay, effectively reduces the capacity of the network. So deeper and wider networks might have to be used to compensate for the reduced capacity.

10.3.3 Dropout

Overfitting occurs because all items in the training set are burned into the network, so that the network doesn't generalize well to new data. One problem is complex co-adaptation in the network, i.e., units depend on fine differences between the activity of other units to generate their outputs. Complex *co-adaptation* are sensitive to small changes in the input and therefore do not generalize well. The second problem is that all units in the network are trained on the training data. So all units adapt to the idiosyncrasies in the training data (Hinton et al., 2012; Srivastava et al., 2014).

A solution to the first problem is to add stochasticity, a solution to the second problem is to train different networks on different subsets of the training data and then to average the predictions across the ensemble of networks. Even though every network might overfit the dataset it was trained on, each network will learn a different input-output relationship and therefore generate different predictions on the test data. A problem with this approach is that many networks have to be trained and stored. If the networks are large, this required a long time.

Dropout is a regularization method that adds stochasticity and approximates the training of a large number of different neural networks in parallel. During training, each unit is randomly ignored, or dropped, with a probability p. Typical values for the dropout rate are 0.5 and 0.2.



(a) Standard Neural Net (b) After applying dropout.

Figure 42: Illustration of dropout in a neural network (Srivastava et al., 2014, Fig. 1).

Dropout introduces noise into the training process and forces nodes to probabilistically take on more or less responsibility for the inputs, hence averting complex co-adaptation. Each iteration of dropout creates a different network with different numbers of units in the layers and different connectivity. Each update to the weights is applied only to the units that are not dropped. The weights are shared between the different configurations of the network.

The number of different effective networks that can be generated by dropout is computed as follows

units dropped out	**number of conf.**
1	n^l
2	$n^l(n^l-1)/2$
$\vdots$	$\vdots$
k	$\binom{n^l}{k}$

Since the number of configurations reaches a maximum at $k = n^l/2$, a dropout rates of $p = 0.5$ is the most common choice. However, for a given problem other rates may yield a better performance.

Dropout reduces the effective capacity of the model, because the effective size of the network is smaller than the number of available units. Therefore a deeper and wider architectures has to be used to compensate for the lower capacity.

10.3.4 Data augmentation

Visual object recognition should be *invariant* under several transformations of the image, because the identity of objects does not change, if the object is translated, rotated, scaled, stretched (to a certain extend), discolored (to a certain extend), damaged (to a certain extend), or occluded. In addition, the object identity is invariant to changes in our view of the object, i.e., whether we are near to or far from the object, how it's lighted, noise in our perception (e.g. smoke or fog) or changes of the background behind the object. Humans and other animals have learned these invariances and can recognize objects under all these transformations. However, the mathematical representation of a transformed image I' is very different from the original image I, i.e., a distance measure such as $l_2(I, I')$ would be quite large – in fact in many cases it would be larger than changing the identity of the object. The ANN has to learn these invariances, i.e. map the very different input images I and I' to the same label. In the early days of computer vision, it was tried to built models that explicitly took into account these invariances (see pooling above for an example).

In deep learning, these invariance are trained into the network by *data augmentation*. All images are presented as they are, but also transformed in all the ways that one would like while leaving object identity, i.e. the label, intact. The network then learned to associate very different input images I and I' with the same label. It has been shown that deep networks *generalize* the invariance to novel objects that were not shown before.

Another advantage of data augmentation is that it increases the size of the dataset tremendously, because each transformed image I' and its label form a new item that the network learns from.

10.4 Speeding up convergence

10.4.1 Transfer learning

When training a very large neural network, it is helpful to start with an existing network that was trained on a similar task to the one at hand, and to reuse the lower layers of that network in the new network. This so-called *transfer learning* works because the lower layers in ANN generally learn representations that are closer to the inputs, and these representations are generally more generic, i.e., they are useful for a number of different tasks. So, the lower layers can be transferred to another network to perform a similar tast. The closer to the output layer a layer is, the more specific its representation is for the task the network is performing. That's why the upper layers have to be trained specifically.

Transfer learning will speed up training and require less training data. Since the weights in the lower layer are already close to correct, there are fewer network parameters that still need to be trained. In addition, since backpropagation starts at the output layer, the training signal is always the most informative in the upper layers and the weights there converge the fastest. It takes longer for the weights in the lower layers to converge. In transfer learning, those weights are nearly correct already and so the more time-consuming part of training can be dramatically shortened.

10.4.2 Learning rate schedule

Training the network critically depends on the learning rate (see earlier lecture). If it is much too high, the optimization diverges Fig. 43. If it is somewhat high, optimization will converge very fast initially, but fail to reach the minimum. If it is too low, convergence will be very slow. A good itermediate learning rate achieves a medium convergence speed initially and a low asympototic loss. With a *learning rate schedule*, one can have the best of both worlds. A fast initial convergence with an high initial learning rate, and good asymptotic performance in later update steps. So, the learning rate changes during the course of training.

$$\eta_t = r(t, \xi) \tag{125}$$

There are several different choices for the learning rate schedule:

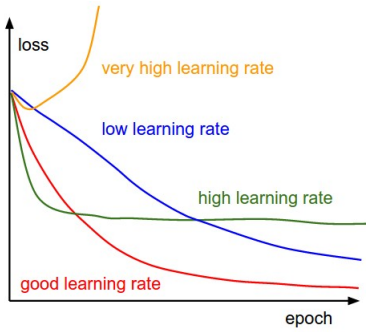


Figure 43: The effect of choosing different learning rates on network performance.

- predetermined piecewise constant learning rate: learning rate stays constant until certain predetermined epochs, in which the learning rate is reduced
- performance scheduling: every T steps measure the validation error and reduce the learning rate by a factor β when the error changes $< \varepsilon$.
- expotential scheduling: $\eta_t = \eta_0 d^{-t/r}$
- power scheduling: $\eta_t = \eta_0 (1 + t/r)^{-c}$

10.4.3 Momemtum optimization

Gradient descent can be very slow, especially when the some activation values are very small. In addition, each update step is independent of the previous ones. To speed up convergence, a metaphor from physics can be adopted for optimization. Imagine that you are describing a ball rolling down a hill. The parameter values θ correspond to the position of the ball. The loss function corresponds to the potential energy of the ball. Then, gradient descent equates the velocity of the ball to the gradient of the loss function, i.e., the gradient describes how the position of the ball changes.

$$\theta \leftarrow \theta - \eta \nabla L(\theta) \tag{126}$$

If instead we assume that the gradient describes the acceleration of the ball, then the gradient describes how the velocity changes:

$$v \leftarrow \mu v - \eta \nabla L(\theta) \tag{127a}$$

and the velocity then describes how the position, i.e.,

$$\theta \leftarrow \theta + v \tag{127b}$$

This method is called *momentum optimization*. The parameter μ is often called the *momentum* because it describes the tendency of the optimization to continue in the same direction as previously. For $\mu = 0$, there is no momentum and the optimization forgets the previous direction immediately. That is the normal gradient descent. For $\mu = 1$, the optimization never forgets any previous step. The only way to change the direction, is when the gradient changes direction. This is a bad idea since around a local minimum of the loss function, the gradient vanishes, so the velocity will only change once the optimization has overshot. This leads to oscillations. A good trade-off between these two extremes is $\mu = 0.9$, but the optimal value is different in every application.

A small modification improves the performance of momentum optimization, because in Eq. 127a the gradient is evaluated at θ, even though the gradient is added to the parameters at $\theta + \mu v$. Nesterov Accelerated Gradient (NAG)

$$v \leftarrow \mu v - \eta \nabla L(\theta + \mu v) \tag{128a}$$
$$\theta \leftarrow \theta + v \tag{128b}$$

There are many other optimization methods that exploit certain properties of the gradient around the local minimum to speed up convergence. We cannot mention then all here. One that is used and mentioned quite frequently is

Adam, which stands for *adaptive moment estimation*. It combines two different techniques and works very well for many problems. However, in some cases it might lead to solutions that generalize poorly.

In addition, some optimization methods make use of the second derivative, i.e., the Hessian. However, those are expensive and for a network with millions of parameters, the Hessians cannot even be stored in computer memory.

10.5 Problem set 1

1. **Fully connected (dense) networks.** Build and train a fully connected network with 4 layers. Note that *keras* does not have an explicit representation of the input layer. The input is passed directly to the first hidden layer. Make use of the Keras documentation (keras.io).

   (a) Import the 'fashion_mnist' dataset from the 'keras' library and load it. The data set contains 60 000 images with a dimension of 28x28 pixels. Visualize a few images to get a feel for the data. Vectorize the images for the training and test set.

   (b) Note that the images are in greyscale. Scale of the pixel values appropriately to the range [0.0;1.0]. Why is this necessary? (Hint: Think about the backward pass and what it does to the weights.)

   (c) Build a Sequential network. Add two `Dense` layers, with 10 units each and ReLU activation functions. Add another `Dense` layer with 10 units and a softmax activation function. The sizes of the hidden layers are flexible, but the size of the output layer must remain at 10 units. Why?

   (d) Calculate by hand how many trainable parameters are in your network (include the bias terms). Note it down for later.

   (e) Compile (`.compile()`) the network using Stochastic Gradient Descent (SGD) as an optimization method, the `sparse_categorical_crossentropy` loss function, and `accuracy` as metrics.

   (f) Train the network for 3 epochs. Use the test set for validation. Note down the accuracy metric on the training and test sets. Define and compile new networks and train them a few times to get a feel for the training process.

   (g) If you did not code it explicitly, find out what was the value that you used in the learning rate. Change the learning rate to 0.005 and train the network for a few times.

   (h) Restore the learning rate to the default. Increase the number of neurons in the hidden layers to 64. Calculate the number of trainable parameters and compare it to the number of parameters in the models above. Also compare the accuracies to those of the smaller models.

   (i) Reduce the training data to 20.000 items. Train a new network and look at its accuracy. Why does the test set accuracy diverge from the training set accuracy? Does it matter for us?

   (j) Using the trained network, find all elements of the test set that the model predicts to be ankle boots (label= 9). Visualize the retrieved items. Use a display loop to do that. Stop the loop whenever an item has been classified incorrectly. Visually inspect the incorrectly classified items. Give a reason for those items being mis-classified as ankle boots.

2. **Convolutions.**

   (a) Given two 'image' matrices:

$$\mathbf{I}_A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

   and two filter (kernel) matrices:

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{K}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

manually compute the 2d convolutions $\mathbf{K}_i * \mathbf{I}_i$ for $i = x, y$ and $j = A, B$ on paper. Assume that the image matrices are padded with the appropriate number of additional rows and colums, which are filled by repeating their current border (you can compare with 'nearest' method in 'scipy.ndimage.convolve').

(b) Inspect the results of the convolutions: What kind of features do the filter kernels extract?

3. **Convolutional Neural Networks (CNN).** Build and train a 6-layer CNN using the *keras* library.

   (a) Proceed as in problem 1, except for the following two points:

      i. Change the dimensionality of your data appropriately so that they can serve as input for a CNN, i.e. $28 \times 28 \times 1$.

      ii. Instead of the two hidden dense layers, use two consecutive pairs of 2-d convolution (`Conv2D`) and max-pooling (`MaxPooling2D`) layers. Use `Conv2D` with 16 and 8 filters of size of 3x3 respectively, and ReLU activation function, and `MaxPooling2D` layers with a 2x2 receptive field.

   (b) Compile and train the network as in problems 1f. Compare the final accuracies. Does the discrepancy between training and test error remain? What are the reasons for this?

   (c) Calculate the number of trainable parameters. Compare with the number of parameters for the fully connected network above.

4. **BONUS.** Modify the network from problem 3 according to the following instructions and observe how the training time, the training accuracy and the test accuracy change.

   (a) Remove the two `MaxPooling2D` layers and add a stride of 2 to the `Conv2D` layers. Train this network and compare epoch speed. Why the speedup?

   (b) Use the ADAM optimizer in place of the SGD. Why the increase in accuracy?

10.6 Problem set 2

1. **Learning rate in DNN.** Build a deep convolutional neural network similar to the one in problem set 10, problem 3, but insert a new dense layer with ReLU activation function and 64 units between the last max pooling layer and the output layer. As optimizer use `keras.optimizers.SGD`.

   (a) Train models with varying learning rates $\eta \in \{1, 0.1, 0.01, 0.001, 0.0001\}$ on the first $20,000$ samples of the fashion MNIST training data set. Train each model for a total of 10 epochs. After each epoch evaluate the accuracy and loss on the training set. **Note:** Make sure to rebuild your model and compile it with the new learning rate. Just calling *model.compile* is not enough, since it does not reinitialize the weights of your model.

   (b) Plot accuracy and loss for each model as a function of trained epochs. How do the different learning rates affect the speed of learning? Can the choice of learning rate prevent the model from learning?

2. **Overfitting in DNN.** Reuse the model architecture of the previous problem and increase the number of units of the last hidden layer to 128. You may use *keras.optimizers.Adam* as optimizer.

   (a) Train models on varying training set sizes $N \in \{30000, 15000, 2500, 500\}$ of the fashion MNIST data set. Train each model for 10 epochs. Evaluate accuracy and loss on both the training set and the test set.

   (b) Plot accuracy and loss for each model as a function of training set size. How does the size of the training set affect accuracy and loss on the test set?

3. **Data Augmentation.** The images in the fashion MNIST dataset are standardized (crop, scale and orientation) to make it easier to train neural networks. Of course, real images are not standardized. To simulate a more realistic dataset, we use the *ImageDataGenerator* (from *keras.preprocessing.image*), a preprocessing tool that generates augmented data by applying various transformations (rotation, zoom, deformations, etc.) to your original data set. Normally, it is used for data augmentation to increase the size of and variability in your training data set.

(a) Initialize an *ImageDataGenerator* object and use its *flow* function to create an iterator from the first $10,000$ samples of the fashion MNIST training set. During initialization of the *ImageDataGenerator* object use the following parameters:

i. rotation_range: 20
ii. width_shift_range: 0.1
iii. height_shift_range: 0.1
iv. shear_range: 0.5
v. zoom_range: (0.9, 1.1)
vi. fill_mode: 'constant'

Iterate over the iterator object to acquire pretend data until you have generated roughly $20,000$ samples. Split the data $50/50$ into training $Train_{pret}$ and test $Test_{pret}$ sets. Save the pretend data set for later use. **Note:** Iterator objects can be iterated over indefinitely and require manual stopping.

(b) Reuse the model architecture of problem 1 and use *keras.optimizers.Adam* as optimizer with a learning rate of $\eta = 0.01$. Train the network on the training data in the original fashion MNIST dataset $Train_{orig}$ for one epochs and save it. (**Note:** You can save your model by calling its *save* function.) By saving the model you save the architecture, the weights as well as the optimizer and its hyperparameters. Let's call this model M_{orig}.

(c) Evaluate M_{orig} on the original test set $Test_{orig}$ and the pretend test set $Test_{pret}$. We will refer to these accuracies as $M_{orig}(Test_{orig})$ and $M_{orig}(Test_{pret})$. Why does the accuracy differ between the two sets?

(d) Train the same model architecture as above on $Train_{pret}$ pretending that the pretend data was real data. Evaluate the accuracy of this model M_{pret} on $Test_{pret}$. Compare $M_{pret}(Test_{pret})$ to $M_{orig}(Test_{pret})$. What did you expect?

(e) Train the model on $Train_{pret}$ with data augmentation for one epoch. To do so, call *model.fit_generator* with the *ImageDataGenerator* object you created and *steps_per_epoch* set to 4000 (for more information check the Keras documentary). We'll call this model M_{aug}. Compare $M_{aug}(Test_{pret})$ to $M_{pret}(Test_{pret})$. What did you expect?

4. **Transfer learning.** To illustrate transfer learning, we reused the previously trained network M_{orig}.

(a) Load the trained network from problem 3. (**Note:** To load your previously saved model you have to import the *load_model* function from *keras.models*.) Also load the pretend data set you generated in problem 3.

(b) Train the loaded model M_{orig} for one additional epoch on the first 500 samples of the pretend training set $Train_{pret}$. Also train a network of the same type *de novo* on the same 500 samples. Call this model M_{novo} Evaluate the test accuracy for both models. Repeat this for the full pretend training set $Train_{pret}$. How do accuracies differ in these two cases?

12 Recurrent neural networks

Feedforward (ffw) networks process information one item at a time. This mode of processing assumes that each item (x_i, y_i) is independent of other items (x_j, y_j) in the dataset, where $i \neq j$. This assumption does not work well, if the data are sequential, such that an item (x_t, y_t) at one time point t is strongly correlated with the preceding or following item(s). These sequential dependencies occur quite often in data:

1. *Time-series data*. Almost all variables that change in time have strong sequential correlations, e.g., stock prices, gross domestic product, blood pressure, atmospheric temperature, etc. The task is usually to predict some future value(s) in the time-series.

2. Text data. The specific sequence, in which words are arranged, matters a lot for the meaning of a sentence, e.g. "I see everything that I eat." is quite different from "I eat everything that I see." Computer code can be considered text data for our purposes. Networks are used for many different kinds of tasks such as: predicting the next character/word, translation, speech-to-text, handwriting-recognition, figure captioning, sentiment analysis.

3. Biological data such as DNA sequences.

Figure 44: Ripples on water surface. Foto by Brocken Inaglory. Licensed under Creative Commons BY-SA 3.0.

There are several ways, in which sequential data can be processed with a ffw network. One could simply concatenate the entire sequence into one single input and process that input in a ffw network. However, that would dramatically increase the dimensionality of the input and, thus, the size of the network required to model the data. If we knew that the sequential correlations had a certain finite distance, e.g., one item only affects the next τ items in the sequence, then we could concatenate only the $\tau+1$ elements that affect each other

$$\begin{bmatrix} x_{t-\tau} \\ \vdots \\ x_{t-1} \\ x_t \end{bmatrix} \tag{129}$$

and provide them simultaneously as inputs. (Note, sequences with $\tau = 1$ are called *Markov chains*). However, the concatenation approach works only if one knows how many timesteps τ into the future are affected by the current state, τ is finite, and if τ is fixed for the entire sequence. These conditions are not satified in many applications. For example, if you wanted to predict the next word in a sentence, and you are presented the words one-by-one in a sequence. The next word does not only depend on the current word, but also on many preceeding words. Exactly how many is not predetermined and can change a lot based on the sentence, context, speaker, etc.

Finally, one could ignore the information that is contained in the sequence all together and extract some summary information from the data. For instance, instead of representing a text as a sequence of words, one could represent it as a frequency distribution over words, i.e., the input is a vector whose elements represent words and the value indicates the number of times a word occurs in the text. This approach is known as *bag of words*.

While these workarounds might work well enough in some cases, they don't work in general. That's why we need *recurrent neural networks (RNN)*. They are defined structurally by having loops in their connections, this enables information to flow from one unit back to the same unit, which in turn enables the network to sustain activity/ store information in the absence of external inputs. This is the property that allows RNN to integrate information over time. Similar to ripples on a water surface recording a history of rain drops falling in (Fig. 44), RNNs can be thought of as a *short-term memory* that stores a history of its recent inputs. Ffw networks lack this short-term memory. Note, however, that all neural networks, including ffw networks, store memory of training inputs in their weights. This can be thought of as *long-term memory*.

RNNs have been shown to be *Turing complete*.

12.1 Backpropagation through time

A recurrent network evolves from timestep t to $t+1$

$$h_t = \phi\left(w_{fw}x_t + w_{rec}h_{t-1}\right) \tag{130}$$

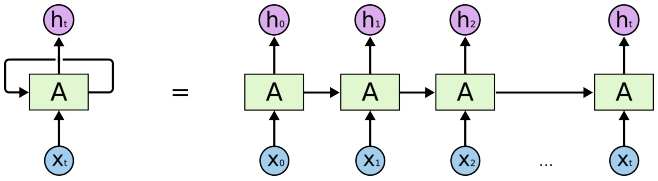
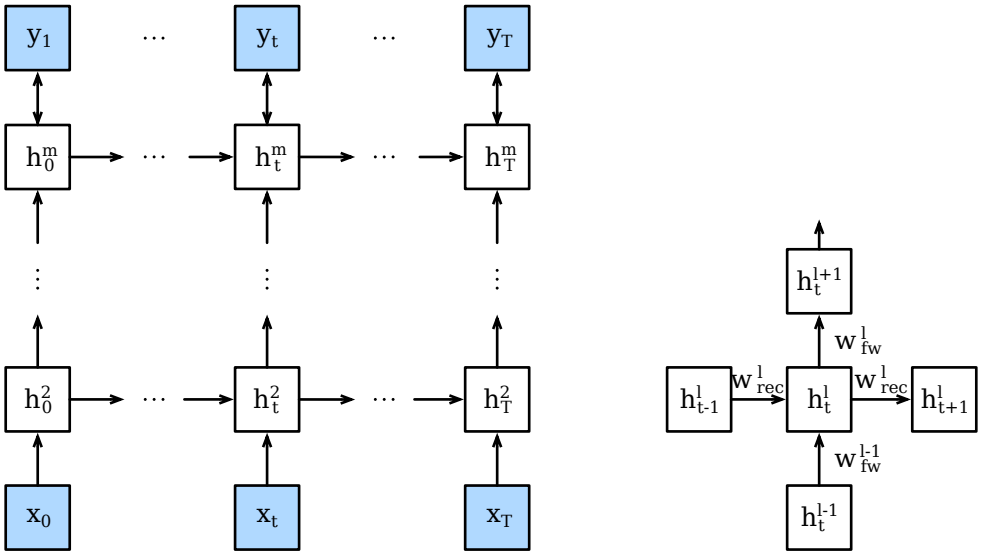


Figure 45: RNN unrolled in time.



(a) Network architecture.

(b) Information flow for one module.

Figure 46: Unrolled deep RNN.

This corresponds to propagating activity from layer $l-1$ to l if the network is unrolled in time (Fig. 45). We can pretend that the network state at a particular timepoint t constitutes a layer in a network, which is connected to the next layer with weights w_t. Of course all the $w_t = w_{rec}$, but we can view this as weight sharing in a network.

We can therefore apply the backpropation algorithm to this unrolled network. This is called *Backpropagation Through Time (BPTT)*. It is like regular backpropagation through a ffw network, except that the recurrent weights are shared. Therefore, the weight changes computed iteratively for every layer Δw_t should not be applied immediately. Instead, they are stored and all applied at the end: $\Delta w_{rec} = \sum_t \Delta w_t$.

12.2 Deep RNN

In addition, RNN architecture can be deep by adding hidden layers to the processing hierarchy (Fig. 46). We denote the activity of the l-th layer at timestep t as

$$h_t^l = \phi\left(w_{fw}^{l-1} h_t^{l-1} + w_{rec}^l h_{t-1}^l\right) \tag{131}$$

That means that the input comes from two sources, the ffw pathway (Fig. 46, vertical arrows) and the recurrent pathway (Fig. 46, horizontal arrows).

Therefore, in the backward pass errors have to flow back through both pathways.

$$\delta_t^l = \phi'\left(a_t^l\right) \odot \left[\left(w_{fw}^l\right)^T \delta_t^{l+1} + \left(w_{rec}^l\right)^T \delta_{t+1}^l\right] \tag{132a}$$

The ffw error in the output layer $l = m$ is determined by the gradient of the loss function.

$$\delta_t^m = \phi'\left(a_t^m\right) \odot \left[\nabla L(h_t^m) + \left(w_{rec}^m\right)^T \delta_{t+1}^m\right] \tag{132b}$$

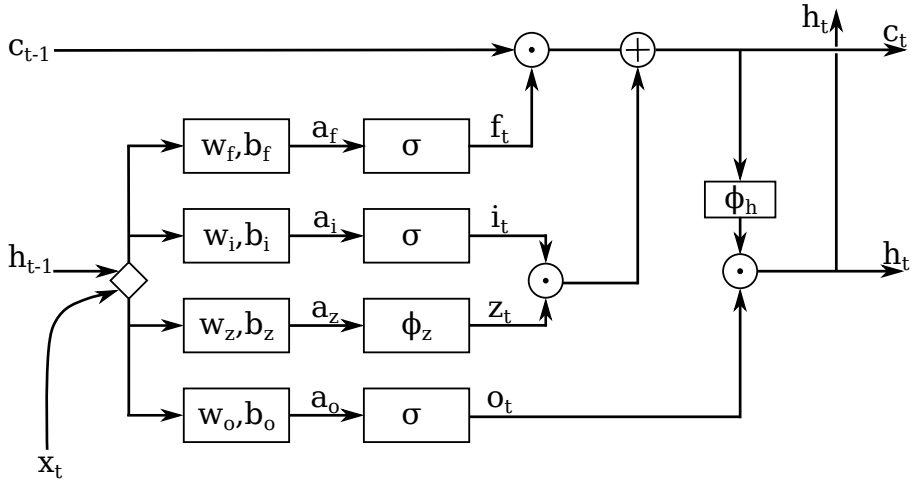


Figure 47: Long short-term memory (LSTM) unit.

The network potentially generates output in multiple time steps, and not just in the last one. Depending on the problem, the external inputs or training labels might be missing at some timepoints in the sequence. If that is the case, the missing input or output terms in Eq. 132 are ignored.

Since the length T of the sequences fed into the network are often quite long, unrolled RNN are usually deep networks and therefore suffer even more than ffw networks from the vanishing and exploding gradients problems. The gradient vanishes or explodes, depending on the largest eigenvalue λ of w_{rec} ($\lambda < 1$ vanishing; $\lambda > 1$ exploding). Therefore, learning in RNNs is particular unstable, and RNNs are difficult to train.

12.3 Long short-term memory (LSTM)

The vanishing/exploding gradient problem can be quite severe in RNN. That means that even though the network maintains short-term memory of the inputs, the errors do not propagate backwards through the network. The training of the recurrent network cannot take into account a long history of inputs, thus limiting the utility of the short-term memory. The solution is to increase the duration of the short-term memory like in the *long short-term memory (LSTM)* network (Hochreiter & Schmidhuber, 1997). A regular unit in an ANN performs a weighted summation of its inputs and then applies a nonlinear activation function. The unit maintains no memory of its previous activities. The short-term memory is maintained in a recurrent network because the previous activity of the layer is provided back to the layer as input. This activity has to serve two purposes, it has to maintain memory of the previous states and it has to signal some information to the other units in the network. All this is controlled by a single weight matrix w^l_{rec}. There is little control over the short-term memory process.

In an LSTM unit, short-term memory is much better controlled by separating the output of the unit from the memory maintanance. Memory of previous states is maintained in an internal state c_t (called a "memory cell"). This internal state is controlled by a forgetting "gate" and updated by external inputs, which is in turn controlled by the input gate (Fig. 47). The internal state passes through an activation function and an output gate to generate the output. All three "gates" are actually neural networks.

12.3.1 Forward pass

Variables

- m, n: the number of input features and number of hidden units
- $x_t \in R^m$: input vector to the LSTM unit/ could be the output of a hidden layer below
- $h_t \in R^n$: hidden state vector also known as output vector of the LSTM unit
- $c_t \in R^n$: cell state vector
- $z_t \in R^n$: the candidate vector

- $f_t \in R^n$: forget gate's activation vector
- $i_t \in R^n$: input/update gate's activation vector
- $o_t \in R^n$: output gate's activation vector
- $w \in R^{n\times(n+m)}$, and $b \in R^n$: weight matrices and bias vector parameters, which need to be learned during training

Activation functions

- σ: activation function of gates, must be logistic function.
- ϕ_z: activation function of inputs, usually hyperbolic tangent function.
- ϕ_h: activation function of outputs, usually hyperbolic tangent function, but could also be identity function.

The cell performs the usual weighted summation and activation function.

$$z_t = \phi_z\left(w_z\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_z\right) \tag{133}$$

where we use the notation x_t for the input to the LSTM unit (in a multi-layer network, the input would be the output of the lower layer h_t^{l-1}). However, the activity is not directly passed on to the output. Instead, it is gated by the input gate i_t before it is integrated into the new cell state. In addition, the new cell state, includes the previous cell state c_{t-1} gated by a forgetting gate f_t. Taken together, the new cell state is given by:

$$c_t = \underbrace{f_t \odot c_{t-1}}_{\text{memory}} + \underbrace{i_t \odot z_t}_{\text{new info.}} \tag{134a}$$

This internal state leaks into the output, controlled by the output gate o_t.

$$h_t = o_t \odot \phi_h(c_t) \tag{134b}$$

The gates are themselves neural networks that are learned.

$$f_t = \sigma\left(w_f\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_f\right) \tag{134c}$$

$$i_t = \sigma\left(w_i\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_i\right) \tag{134d}$$

$$o_t = \sigma\left(w_o\begin{bmatrix}h_{t-1}\\ x_t\end{bmatrix} + b_o\right) \tag{134e}$$

$$\tag{134f}$$

The initial values are $c_0 = 0$ and $h_0 = 0$.

Additional material:

- "Understanding LSTM Networks" by Christopher Olah

12.3.2 Backward pass

We define

$$\delta k_t = \frac{\partial L}{\partial a_k^t} \tag{135}$$

$$\delta h_t = \frac{\partial L}{\partial h_t^l} \tag{136}$$

$$\delta c_t = \frac{\partial L}{\partial c_t^l} \tag{137}$$

where k stands for one of the three gates $k = f, i, o$ or the inputs $k = z$. Given all δ's and gate values from layer l at time $t+1$, and from the next layer $l+1$ at time t, the errors in the current layer are

$$\delta c_t = o_t \odot \phi_h'(c_t) \odot \delta h_t + f_{t+1} \odot \delta c_{t+1} \tag{138a}$$

$$\delta h_t = \delta x_t^{l+1} + M_h \left[(w_f^l)^T \delta f_{t+1} + (w_i^l)^T \delta i_{t+1} + (w_z^l)^T \delta z_{t+1} + (w_o^l)^T \delta o_{t+1} \right] \tag{138b}$$

where $M_h = [I_{n\times n}, 0_{n\times m}]$ and δx_t^{l+1} are the errors from the next layer without the recurrent errors. The errors can be propagated further back towards the activations of the gates and the input.

$$\delta f_t = \sigma'(a_f) \odot c_{t-1} \odot \delta c_t \tag{138c}$$

$$\delta i_t = \sigma'(a_i) \odot z_t \odot \delta c_t \tag{138d}$$

$$\delta z_t = \phi_z'(a_z) \odot i_t \odot \delta c_t \tag{138e}$$

$$\delta o_t = \sigma'(a_o) \odot \phi_h(c_t) \odot \delta h_t \tag{138f}$$

Finally, the error propagated back to the lower layer is

$$\delta x_t^l = M_x \left[(w_f^l)^T \delta f_t + (w_i^l)^T \delta i_t + (w_z^l)^T \delta z_t + (w_o^l)^T \delta o_t \right] \tag{138g}$$

where $M_x = [0_{m\times n}, I_{m\times m}]$.

Once all the δ's have been computed, the updates for the weight and biases are

$$\delta w_k = \sum_{t=0}^{T} \delta k_t \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{138h}$$

$$\delta b_k = \sum_{t=0}^{T} \delta k_t \tag{138i}$$

where $k = f, i, z, o$ and $h_{-1} = 0$.

12.3.3 Applications

Example by Andrej Karpathy: LSTM-network trained on Leo Tolstoy's "War and Peace" to predict the next character.

Sample text self-generated by the network:

Iteration 100:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

Iteration 300:

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

Iteration 500:

```
we counter. He stutn co des. His stanted out one ofler that concossions and was
to gearang reay Jotrets and with fre colt otf paitt thin wall. Which das stimn
```

Iteration 700:

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

Iteration 1200:

```
"Kite vouch!" he repeated by her
door. "But I would be done and quarts, feeling, then, son is people...."
```

Iteration 2000:

```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

12.4 Gated recurrent unit (GRU)

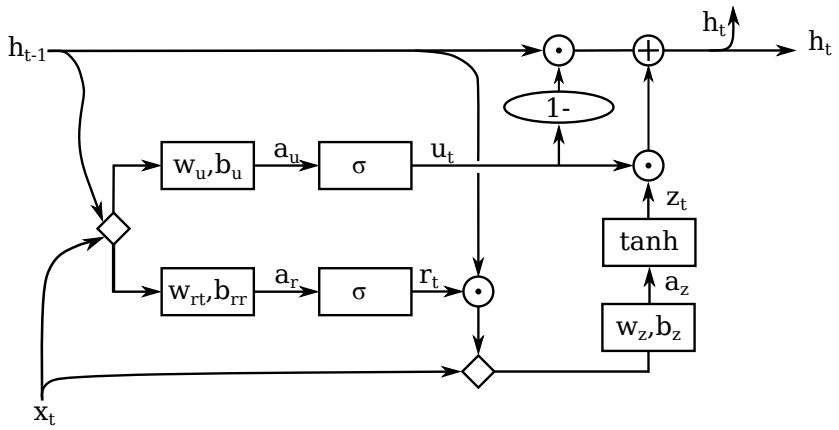


Figure 48: Gated recurrent unit (GRU).

The *gated recurrent unit (GRU)* was suggested by Cho et al. (2014) and is simpler than the LSTM. It maintains no internal cell state and combines the input gate and forget gate into a single update gate u_t (Fig. 48). It also introduces a new reset gate r_t that selectively acts on the previous output h_{t-1}. GRU has fewer parameters that need to be trained and appears to perform better in some cases, but in other cases LSTM perform better. So, it is not clear at the moment when to prefer GRU or LSTM.

12.4.1 Forward pass

$$u_t = \sigma\left(w_u \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_u\right) \tag{139a}$$

$$r_t = \sigma\left(w_r \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + b_r\right) \tag{139b}$$

$$z_t = \tanh\left(w_z \begin{bmatrix} r_t \odot h_{t-1} \\ x_t \end{bmatrix} + b_z\right) \tag{139c}$$

$$h_t = (1-u_t) \odot h_{t-1} + u_t \odot z_t \tag{139d}$$

Variables (same symbols as in LSTM mean the same things)

- $u_t \in R^n$: update gate's activation vector
- $r_t \in R^n$: reset gate's activation vector

12.4.2 Backward pass

Given δ's at $t+1$ and in layer $l+1$.

$$\delta h_t = (1-u_{t+1})(\delta x_t^{l+1} + \delta h_{t+1}) + M_h \left[w_u^T \delta u_{t+1} + w_r^T \delta r_{t+1} \right] + r_{t+1} \odot M_h w_z^T \delta z_{t+1} \tag{140}$$

where $M_h = [I_{n\times n}, 0_{n\times m}]$.

$$\delta z_t = \tanh'(a_z) \odot u_t \odot \delta h_t \tag{141a}$$
$$\delta u_t = \sigma'(a_u) \odot (z_t - h_{t-1}) \odot \delta h_t \tag{141b}$$
$$\delta r_t = \sigma'(a_r) \odot h_{t-1} \odot M_h w_z^T \delta z_t \tag{141c}$$
$$\tag{141d}$$

Finally, the error propagated back to the lower layer is

$$\delta x_t^l = M_x \left[w_u^T \delta u_t + w_r^T \delta r_t + w_z^T \delta z_t \right] \tag{141e}$$

where $M_x = [0_{m\times n}, I_{m\times m}]$.

Once all the δ's have been computed, the updates for the weight and biases are

$$\delta w_k = \sum_{t=0}^{T} \delta k_t \times \begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} \tag{141f}$$
$$\delta b_k = \sum_{t=0}^{T} \delta k_t \tag{141g}$$

where $k = r, u, z$ and $h_{-1} = 0$.

12.5 Problem set

1. **Predicting time series data.** Use Keras' *SimpleRNN* layer to predict data points of a ramping sine wave.

$$f(x) = sin(x) + 0.1x \tag{142}$$

   (a) In Keras, recurrent neural networks require a specific input shape (batch size, time steps, input length). Write a function *prepareData* that takes the number of time steps as a parameter, and prepares inputs x and labels $y = f(x)$ with appropriate shapes. Generate a dataset of size $N = 1000$ (with x linearly spaced in the range $[0, 100]$).

   (b) Build a sequential model with a hidden *SimpleRNN* layer (32 units, ReLU activation) and a fully connected output layer (1 unit, linear activation). Use mean squared error (MSE) as your loss function and Adam as your optimizer.

   (c) Prepare your data using one time step as input x. Split the data set $70/30$ into training and test sets. Train the model for 20 epochs. Plot the prediction for training and test sets.

   (d) Calculate and plot the squared errors for each data point. How does the squared error depend on the phase of the sine wave? Why does it show this behavior? Are there MSE differences between training and test set?

   (e) Repeat the previous two tasks using three time steps instead. Does the MSE differ? How does the squared error depend on the phase of the sine wave?

2. **Learning long-term dependencies.** In this exercise, we will compare a simple RNN to LSTMs in a simple text generation task. Train a network on the text file containing Goethe's *Faust* to predict the *next character* given a sequence of characters, i.e., a character-level RNN. The model is only predicting single characters and does not have any knowledge that words are units of language, or that words are combined into sentences.

(a) Load the text file in lowercase and print out the length of the whole text. To get an idea of the dataset, also print the number of individual characters present in the text after sorting them in order (use the default python functions `sorted` and `set`).

(b) To prepare the data, create two dictionaries that map characters to integers and vice-versa. Next, prepare the training data: split the data into sequences of length 50 that start at every third character of the text. The labels are the next character after each sequence.

(c) To encode the characters use one-hot encoding. Use the dictionaries from the previous step to generate the one-hot encodings of the labels and also of the input sequences. That is, each input sequence is a sequence of the one-hot encodings of the individual characters in that sequence.

(d) Build a Sequential model using an LSTM layer with 256 units as the hidden layer followed by an ouput Dense layer. How many units should the ouput layer have? What are the activation function and loss function that you would use, and why? Use Adam as the optimizer again and compile.

(e) Callbacks are a useful function in Keras that show relevant aspects of the model during training. The file `print_callback.py` provides a keras callback that prints the network output for a random sample after every epoch. Pass this callback to the fit function to see how the model does after every epoch. Train the model for 20 epochs. Save the weights after training. (Note : There is no train/test split. We train the model on the entire dataset to learn a probability distribution of characters in a sequence.)

(f) Test the model using a random input sample and look at the next 500 characters generated (approximately a paragraph). Look at the callback function provided for hints on how to do this. Analyze the features of the generated text.

(g) Repeat the training and test after replacing the LSTM layer with a SimpleRNN of 256 units. What are the qualitative differences you notice in the text generated by the two models?

13 Hopfield network

All models discussed so far map some inputs X to qualitatively different outputs Y, and have been viewed as approximating some kind of function $Y = f(X)$. The goal has been to generalize from the training dataset to a novel test dataset. An alternative goal of training neural networks is that we want to store the training dataset (X,Y), and retrieve the y that was associated with an item x. This is called *hetero-associative memory* and is an important function of the brain. This kind of memory is easy to implement using a computer algorithm and without using neural networks. However, another kind of memory that is not so easy to implement is *auto-associative memory*, or *content-addressable memory*, where an item x has to be retrieved based on a partial, or noisy, version x' of the item. The *Hopfield net* is a recurrent neural network that performs this function (Hopfield, 1982). In a way, memory networks severely overfit the training data, but that's exactly what we want them to do.

13.1 Attractor states and attractor networks

We are familiar with gradient descent to find the parameters of a model that minimize the loss function. In the Hopfield net, the same idea is applied to the network states instead, i.e., there is an *energy function* over the states of a network that is minimized by the dynamics of the network (Fig. 49). The local minima are also called *attractors*, and the Hopfield net is called an attractor network. If we could setup the network such that the minima corresponded to the pattern x that we want to store in the network, then the dynamics of the network could retrieve these patterns, if cued with slightly different patterns x'.

13.2 Definition of the Hopfield Network

Given n data vectors $X = (x^1, x^2, \ldots, x^n)$, where each x^i is a binary vector, i.e., $x^i_j \in \{-1, 1\}$, $j = 1, \ldots, m$. The goal of the Hopfield net is to store these patterns in a neural network and retrieve a stored pattern from the network based on a cue x'. This cue could be a, more-or-less, corrupted version of a patterns that was previously stored in the network. The Hopfield net consists of m binary units, i.e., $h_j \in \{-1, 1\}$, that are connected via weights w_{ij}. Two elements are

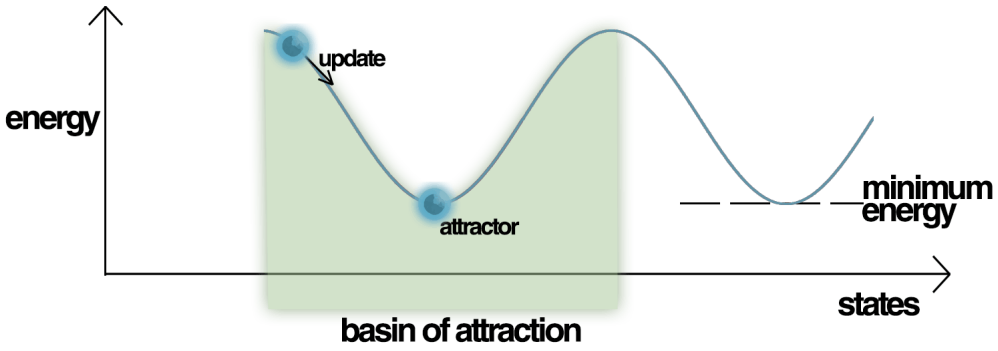


Figure 49: Attractor in energy landscape. The attractor state corresponds to the pattern x that is stored in the network. If the retrieval is initiated with a partial, or noisy, pattern x', which lies within the basin of attraction, the network will evolve towards the attractor state x. Source: Wikimedia Commons, user:Mrazvan22. Licensed under Creative Commons BY-SA 3.0.

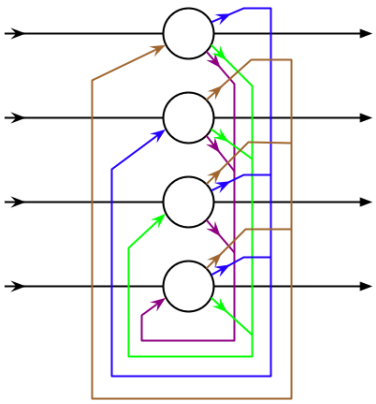


Figure 50: Hopfield net. Source: Wikimedia Commons, user:Zawersh. Licensed under Creative Commons BY-SA 3.0.

essential in the definition of the network: how the weights are set depending on the inputs and the update rule, which gives rise to the dynamics of the network.

13.2.1 Weights

The weights of the recurrent network (Fig. 50) are given by

$$w_{jk} = \begin{cases} 0 & \text{if } j = k \\ \frac{1}{n}\sum_{i=1}^{N} x_j^i x_k^i & \text{if } j \neq k \end{cases} \tag{143}$$

which is equivalent to writing the following matrix identity:

$$w = \frac{1}{n}\sum_{i=1}^{n} x^i \times x^i - I \tag{144}$$

Eq. 143 can be viewed as the learning rule since it encodes the patterns x into the weight matrix w. It has two desirable properties that are biologically plausible:

- Local: The learning rule is local because the weight update uses only information available to neurons that are connected by the weight.
- Incremental: New patterns are added to the weights as they are encountered and previous patterns do not have to be stored for later stages of training.

13.2.2 Update Rule

To start retrieval we initialize the network with the retrieval cue, i.e. $h = x'$, and iterate the update rule Alg. 13.1.

Algorithm 13.1 Hopfield Net – asynchronous update

Require: w, partial cue h

1: **repeat**

2: Randomly select a unit j

3: Update state of unit j according to

$$h_j \leftarrow \begin{cases} -1 & \text{if } \sum_k w_{jk} h_k + b_j \leq 0 \\ 1 & \text{if } \sum_k w_{jk} h_k + b_j > 0 \end{cases} \tag{145}$$

4: **until** predetermined number of iterations is reached

This update rule changes the state of the network until it reaches a local minimum of the energy function (see below). The bias term can be thought of as some constant external input that is provided to the network during the updating. Most of the times, it will be zero.

For simplicity, we define the function

$$Q(t) = \begin{cases} -1 & \text{if } t \leq 0 \\ 1 & \text{if } t > 0 \end{cases} \tag{146}$$

So, the update equation becomes

$$h_j \leftarrow Q\left(\sum_k w_{jk} h_k + b_j\right) \tag{147}$$

The asynchronous update rule robustly leads to convergence, but is also slow because in each step the activity of only a single unit is updated. To speed the computations, one can update all units at the same time (synchronous update).

Algorithm 13.2 Hopfield Net – synchronous update

Require: w, partial cue h

1: **repeat**

2: Update network state according to

$$h \leftarrow Q(wh+b) \tag{148}$$

3: **until** predetermined number of iterations is reached

From a computational perspective, the disadvantage of the synchronous update rule is that it can lead to oscillations, which do not occur when the asynchronous update rule is used. From a biological perspective, the synchronous update rule seems less realistic since it requires a central clock that determines when the units should be updated and there is little evidence for such a mechanism in the brain.

13.2.3 Examples:

Define a Hopfield net that stores one pattern x:

$$x^1 = \begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix} \quad \rightarrow \quad w = \begin{bmatrix} 0 & -1 & +1 \\ -1 & 0 & -1 \\ +1 & -1 & 0 \end{bmatrix} \tag{149a}$$

Start with x and $b_j = 0$, and iterate the update algorithm Alg. 13.1 for the pre-set number of iterations. Verify that $x = x^1 \rightarrow x^1$. In the following the box around an element indicates that that element is being updated in the next step.

$$\begin{bmatrix} \boxed{-1} \\ +1 \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ +1 \\ \boxed{-1} \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ \boxed{+1} \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ +1 \\ 1 \end{bmatrix} \tag{149b}$$

Example for pattern completion:

$$\begin{bmatrix} \boxed{-1} \\ -1 \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ \boxed{-1} \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix} \tag{149c}$$

Example for clean-up:

$$\begin{bmatrix} +1 \\ \boxed{+1} \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} \boxed{+1} \\ -1 \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ \boxed{-1} \\ -1 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix} \tag{149d}$$

Example for *spurious attractor*:

$$\begin{bmatrix} +1 \\ \boxed{+1} \\ +1 \end{bmatrix} \rightarrow \begin{bmatrix} +1 \\ -1 \\ \boxed{+1} \end{bmatrix} \rightarrow \begin{bmatrix} \boxed{+1} \\ -1 \\ +1 \end{bmatrix} \rightarrow \begin{bmatrix} +1 \\ \boxed{-1} \\ +1 \end{bmatrix} \rightarrow \begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix} \tag{149e}$$

13.2.4 Spurious attractors

Spurious attractors are local minima of the energy of a Hopfield network that do not correspond to one of the patterns stored in the network. There are different types of spurious attractors:

1. The inverse pattern. If x is an attractor then $Q(Wx) = x$. If we define $y = -x$ then

$$Q(Wy) = Q(W(-x)) = Q(-Wx) = -Q(Wx) = -x = y \tag{150}$$

   So, if x is an attractor of the Hopfield net, then so is $-x$. Since they were not intended to be attractors of the network, they are called spurious attractors.

2. Linear combination of an odd number of attractor states. (Linear combinations of an even number of attractor states are not attractor states, because the elements ($= \pm 1$) can sum to zero.)

3. For large n, local minima that are not linear combinations of stored patterns.

13.3 Convergence

The energy of a Hopfield network is given by

$$E(w,h) = -\frac{1}{2}\sum_{j,k} w_{jk}h_j h_k - \sum_j b_j h_j \tag{151}$$

$$= -\frac{1}{2}h^T w h - b^T h \tag{152}$$

In analogy to physics, this function is called the energy because every update of the Hopfield network decreases it. Consider what happens to the energy when the network state is updated: $h \to h'$.

$$\Delta E = E(w,h') - E(w,h) \tag{153a}$$

In the asynchroneous update rule only one unit's activity is updated, while all other units remain the same. Say unit v is updated, then $h'_v \neq h_v$ and $h'_j = h_j$ for all $j \neq v$. Therefore, terms in Eq. 152 that do not include either h'_v or h_v appear identically in both $E(w,h')$ and $E(w,h)$ and therefore cancel out. In other words, only terms where either $j = v$ or $k = v$ survive the substraction. Therefore,

$$\Delta E = -\frac{1}{2}\sum_j w_{jv}h'_j h'_v - \frac{1}{2}\sum_k w_{vk}h'_v h'_k - b_v h'_v + \frac{1}{2}\sum_j w_{jv}h_j h_v + \frac{1}{2}\sum_k w_{vk}h_v h_k + b_v h_v \tag{153b}$$

Since the summation index is arbitrary and the weight matrix is symmetrical ($w_{jk} = w_{kj}$), some of the sums are identical to each other:

$$= -\sum_k w_{vk}h'_v h'_k - b_v h'_v + \sum_k w_{vk}h_v h_k + b_v h_v \tag{153c}$$

Since $w_{vv} = 0$, we can restrict the sums to $k \neq v$

$$= -\sum_{k\neq v} w_{vk}h'_v h'_k - b_v h'_v + \sum_{k\neq v} w_{vk}h_v h_k + b_v h_v \tag{153d}$$

For $k \neq v$, $h'_k = h_k$ and so

$$= -\sum_{k\neq v} w_{vk}h'_v h_k - b_v h'_v + \sum_{k\neq v} w_{vk}h_v h_k + b_v h_v \tag{153e}$$

$$= -(h'_v - h_v)\left(\sum_{k\neq v} w_{vk}h_k + b_v\right) \tag{153f}$$

If the activity of unit v (h_v) didn't change, $(h'_v - h_v) = 0$. Otherwise h'_v and h_v have opposite signs, which means that the first factor $(h'_v - h_v)$ has the same sign as h'_v. The second factor is the summed input to unit v, and is either zero or has the same sign as h'_v (as per Eq. 145). Therefore, ΔE is either zero or both factors in ΔE have the same sign, which means $\Delta E < 0$. Hence,

$$\Delta E \leq 0 \tag{153g}$$

This means that the energy function monotonically decreases. Since there is only a finite number of states, the dynamics of state changes must eventually converge. Note that this proof does not say which pattern the dynamics converges to. We will see in the capacity derivation below that the stored patterns are in fact attractor states of the network.

13.4 Capacity

The first step is to state clearly what is meant by the *memory capacity* of the Hopfield net. It is the maximum number of patterns n that can be stored in the network. This statement in turn has to be made explcit. To say that the network has stored n patterns, at the very least, the following has to be true of all the n patterns: When initialized with a stored pattern x^μ, the updates of the Hopfield net converge to a pattern that is very similar to that stored pattern. We therefore study the probability that a given unit's activity remains faithful, i.e. $h_j = x_j^\mu$. Then we will only state the results for the retrieval of the entire pattern and not derive it because the derivation is quite involved and outside the scope of this class.

We assume n patterns are stored in the network with m units. Since the considerations below are based on statistical arguments, m has to be large. We define the *load* parameter $\alpha = n/m$. Assume that there is no biasing input, i.e., $b = 0$. Use x^μ as input, then the update rule is

$$h_j = Q\left(\sum_k w_{jk} x_k^\mu\right) \tag{154a}$$

$$= Q\left(\frac{1}{n}\sum_{k\neq j}\sum_{i=1}^{n} x_j^i x_k^i x_k^\mu\right) \tag{154b}$$

We split the inner sum into two terms and note that $x_j^i x_k^i x_k^\mu = x_j^\mu$ for $i = \mu$

$$= Q\left(\frac{1}{n}\sum_{k\neq j}\left(x_j^\mu + \sum_{i\neq\mu} x_j^i x_k^i x_k^\mu\right)\right) \tag{154c}$$

$$= Q\left(\frac{m-1}{n} x_j^\mu + \frac{1}{n}\sum_{k\neq j}\sum_{i\neq\mu} x_j^i x_k^i x_k^\mu\right) \tag{154d}$$

We define the cross term:

$$C_j^\mu = \sum_{k\neq j}\sum_{i\neq\mu} x_j^i x_k^i x_k^\mu \tag{154e}$$

If the cross term has the same sign as x_j^μ, then h_j has the same sign, too. If the cross term has the opposite sign to x_j^μ, and $|C_j^\mu| > m-1$, then the cross term will flip the sign of h_j. Depending on the sign of x_j^μ, the probability of that flip is equal to

$$P(C_j^\mu > m-1) \text{ or } P(C_j^\mu < -(m-1)) \tag{155}$$

which are equal since C_j^μ is symmetric. To compute this probability, we will assume that $x_j^i x_k^i x_k^\mu \in \{-1, 1\}$ is a random variable.

Aside: Central Limit Theorem Given random variables v_i with mean μ and variance σ^2, and large n, then the random variable

$$u = \frac{1}{n}\sum_{i=1}^{n} v_i \tag{156}$$

is normally distributed with mean μ and variance σ^2/n, i.e.,

$$u \sim N\left(\mu, \frac{\sigma^2}{n}\right) \tag{157}$$

Approximating Eq. 154e as

$$C = mn\frac{1}{mn}\sum_{j=1}^{mn} x_j \tag{158}$$

where $x_j \in \{-1, 1\}$, $\mu(x_j) = 0$ and $\sigma^2 = 1$, and using the central limit theorem, we obtain

$$C \sim N(0, mn) \tag{159}$$

where the variance is the product of $(mn)^2$ due to the coefficient, and $\frac{1}{mn}$ as per central limit theorem.

$$C' = \frac{1}{\sqrt{mn}} C \sim N(0,1) \tag{160}$$

$$P(C_j^\mu > m-1) \approx P(C > m) = P\left(C' > \sqrt{\frac{m}{n}}\right) = P\left(C' > \frac{1}{\sqrt{\alpha}}\right) = \int_{\frac{1}{\sqrt{\alpha}}}^{\infty} N(x)dx \tag{161}$$

So, the probability of a flip depends only on the load parameter α. The probability is plotted in Fig. 51. The flip

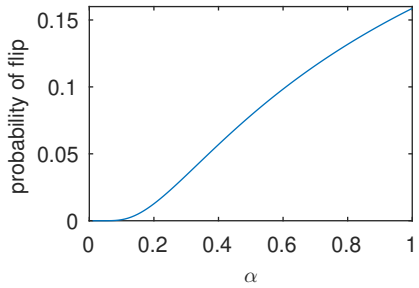


Figure 51: Probability that one unit is flipped incorrectly in one update step of the Hopfield net.

probability is quite low even for high loads of the network. However, the flip probability only looks at one update of an individual unit. Retrieval of the entire pattern requires many updates to different units and errors can accumulate quickly. So, our intuition is that the flip probability of a single unit should be very low, so that the pattern can be retrieved with a high probability. From Fig. 51 we might guess that a load of $\alpha \approx 0.15$ might be the maximum.

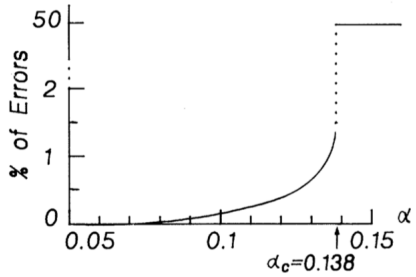


Figure 52: Probability of error in Hopfield net as a function of load (α). Source: Fig. 1, Amit et al. (1985)

Unfortunately, the derivation of the retrieval probability of the entire pattern is rather difficult to follow. We therefore only report the final result in Fig. 52 and note that the theoretical result for the critical value is $\alpha_c = 0.138$ – quite close to our intuitive guess based on the flip probability.

The steep increase of the error probability to 50% in Fig. 52 for $\alpha > \alpha_c = 0.138$ indicates *catastrophic interference*: adding just a few more patterns beyond its capacity will render the network unable to retrieve any patterns correctly. So, the maximum number of patterns that can be stored in a Hopfield net is $n_{max} = 0.138m$. Note that the capacity estimate is based on stochastic arguments, so stable solutions might exist for $\alpha > \alpha_c$ for certain sets of patterns.

13.5 Problem set

1. **Asynchrounous updates in Hopfield network.** We stored one input pattern in a Hopfield network, which yielded the following weight matrix:

$$w = \begin{bmatrix} 0 & -1 & -1 & +1 \\ -1 & 0 & +1 & -1 \\ -1 & +1 & 0 & -1 \\ +1 & -1 & -1 & 0 \end{bmatrix}$$

Based on this, answer the questions below.

(a) How many elements did the pattern have?

(b) Starting with any valid pattern you like, use algorithm 13.1 from the lecture notes using pen and paper to find the pattern the network converges to. Is the pattern you have found necessarily the one the network was trained on?

(c) Now write a Python script that automatizes this process and compare your results.

(d) Modify your code so as to calculate and store the energy of the network for each step in (c). How does it vary across multiple repetitions of your script?

2. **Pattern storage and retrieval in Hopfield network.** There are seven images stored in `images.npy`. Follow the tasks below to complete this exercise.

(a) Load the data. You will observe that the shape of the data is 1150×7. The first dimension corresponds to the number of pixels in each image (46×25 pixels stacked column-wise into vectors). The second dimension corresponds to the number of images. The content of each pixel is either -1 or 1, where -1 corresponds to black and 1 to white.

(b) Reshape the image vectors according to the information provided in 2a and visualize the images.

(c) Store these images in a Hopfield network using Eq. 2 from the lecture notes.

(d) Add noise to the images by randomly choosing a subset of pixels and flipping their values from -1/1 to 1/-1. Generate noisy images by setting the subset size to 200 and provide them as input to the network. Run the algorithm for 100 iterations to retrieve the stored images. Visualize this process by showing the original, noisy and retrieved images side by side.

(e) For each image compute the correlation coefficient (`numpy.corrcoef`) of the original against the retrieved images as a measure of retrieval quality. The correlation coefficient (cc) outputs a number between -1 and 1, $cc \in [-1, 1]$. Interpret what these values mean and which value(s) correspond(s) to lossless retrieval.

(f) During retrieval, even with 0 noise, one image is confused with another one. Compute the correlation coefficient across all pairs of images. Do you find any relationship between the computed correlation coefficients and the incorrectly retrieved image?

(g) Vary systematically the number of flipped pixels from 0 to 1150 with a step size of 25 and record the correlations between original and retrieved images. Plot these correlations for all images and noise levels, e.g., using `pyplot.matshow`. Describe the effect of noise in the retrieval process.

3. **Capacity of Hopfield network.** To study the memory capacity of the Hopfield network, generate and store random patterns in a network of 100 units. Then, initialize the network with each of the stored patterns and let it evolve for 100 iterations. After this, count the number of pixels which have deviated from the original value (`numpy.count_nonzero` might be of help here). Repeat this process for different numbers of patterns and plot the error rate as a function of the network load (= number of stored patterns / network size). Focus on network loads between 0 and 0.5. At which point does performance start to degrade steeply?

14 Boltzmann machine

Boltzmann machines are stochastic neural networks that represent statistical distributions. The units in the Boltzmann machine are divided into visible units, v, and hidden units, h (Fig. 53). The visible units receive inputs, i.e. the training set is a set of binary vectors over the set V. The hidden variables represent *latent variables* that influence the activity of the visible variables, but are not observed directly.

Boltzmann machines are generalizations of the Hopfield net and have several advantages over the Hopfield net or ffw networks.

1. They are *generative*, i.e., once they are trained, they can generate different instances of the data according to a certain distribution.

2. They can assign probabilites to states they have never seen before. In many situations, e.g., for a nuclear power plant, we cannot sample all possible states, because some states are dangerous. In these cases, supervised learning of desirable and undesirable states doesn't work.

3. The Hopfield net gets stuck in local minima quite easily. Stochasticity helps to move on to a better local minimum.

Boltzmann machines use the same energy function as the Hopfield net.

$$E(\theta, s) \quad = \quad -\frac{1}{2}\sum_{j,k} w_{jk} s_j s_k - \sum_j b_j s_j \tag{162}$$

where s_i is the state of the i-th unit, w_{ij} the symmetric weight between unit i and j, and b_i the bias.

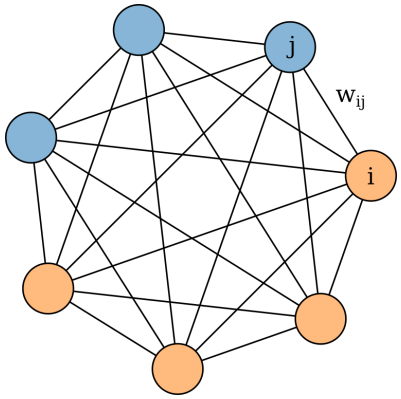


Figure 53: Boltzmann machine. Orange: visible units; blue: hidden units.

Additional material:

- Video of lecture by Geoffrey Hinton.

14.1 Stochastic neural network

Unlike all the networks we have discussed before, the units in a Boltzmann machine are stochastic. Describe the state as the vector s. Then each unit is either active ($s_i = 1$) or inactive ($s_i = 0$). The probability of finding the system in the state s is given by the *Boltzmann distribution*:

$$P(s) = \frac{\exp(-E(s))}{\sum_{s'} \exp(-E(s'))}. \tag{163}$$

In many treatments of the Boltmann machine, the energy is divided by a scalar T, the temperature of the system. The temperature has an important meaning in statistical physics and influences the networks dynamics. However, in practice it turns out that the temperature is not very relevant for Boltzmann machines and we therefore set the temperature to $T = 1$. The Hopfield network could be viewed as a Boltzmann machine at temperature $T = 0$. The denominator in Eq. 163 is known as the *partition function*:

$$Z = \sum_s \exp(-E(s)), \tag{164}$$

The summation is over all possible states of the system, of which there is a very large number in any reasonably sized system. Therefore, the partition function is very costly to calculate and the probability of observing a particular state is not practicably computable.

The trick is to find ways to compute meaningful quantities without having to compute the partition function. For instance, the differences in energy ΔE_i that results from a single unit i being off ($s_i = 0$) versus on ($s_i = 1$). Assuming a symmetric matrix of weights:

$$\Delta E_i = E(s_i = 0 | s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) - E(s_i = 1 | s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n) \tag{165}$$

To simplify the notation, we will write $i = \text{off}$ for $s_i = 0 | s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$ and $i = \text{on}$ for $s_i = 1 | s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n$

$$\Delta E_i = E(i = \text{off}) - E(i = \text{on}) \tag{166}$$

Since the energy is given by Eq. 162, we can compute the energy difference from the network activity (see also the derivation of a similar equation in the Hopfield lecture)

$$\Delta E_i = \sum_{j \neq i} w_{ij} s_j + b_i \tag{167}$$

Next, we would like to also set the network activity based on the energy difference. To this end, we substitute the energy of each state in Eq. 166 with its relative probability according to the Boltzmann factor (the property that the energy of a state is proportional to the negative log probability of that state) gives:

$$\Delta E_i = -\log P(i = \text{off}) + \log P(i = \text{on}) \tag{168}$$

We then rearrange terms and consider that the probabilities of the unit being on and being off must sum to one, and $p_i = P(i = \text{on})$:

$$\Delta E_i = \log p_i - \log(1 - p_i) \tag{169}$$

$$\Delta E_i = \log \left(\frac{p_i}{1 - p_i} \right) \tag{170}$$

$$-\Delta E_i = \log \left(\frac{1 - p_i}{p_i} \right) \tag{171}$$

$$-\Delta E_i = \ln \left(\frac{1}{p_i} - 1 \right) \tag{172}$$

$$\exp(-\Delta E_i) = \frac{1}{p_i} - 1 \tag{173}$$

Solving for p_i, the probability that the i-th unit is on gives:

$$p_i = \frac{1}{1 + \exp(-\Delta E_i)} = \sigma(\Delta E_i) \tag{174}$$

This relation is the source of the logistic function found in probability expressions in variants of the Boltzmann machine. Plugging in Eq. 167 yields the final equation:

$$p_i = \sigma \left(\sum_{j \neq i} w_{ij} s_j + b_i \right) \tag{175}$$

Eq. 175 defines a neural network, in which each unit calculates a linear weighted sum of its inputs from other units and then applies a stochastic activation function.

14.2 Thermal equilibrium

An important concept in the study of stochastic systems, is the concept of *equilibrium*, or *thermal equilibrium*. It doesn't mean that the network state is constant. This cannot occur since the state of the network is stochastic and therefore constantly changing. However, if the network is left alone and updates its state a sufficient number of times, then it will reach a point in which the states generated by the network will be distributed according to the Boltzmann distribution Eq. 163. If the network is disturbed, e.g. by external inputs, then the previous statement will not be true. The distribution of states will depend on the initial state from which the process was started, and not just on T.

Running the network beginning from a high temperature, its temperature gradually decreases until reaching a thermal equilibrium at a lower temperature. It then may converge to a distribution where the energy level fluctuates around the global minimum. This process is called simulated annealing.

To train the network so that the chance it will converge to a global state is according to an external distribution over these states, the weights must be set so that the global states with the highest probabilities get the lowest energies. This is done by training.

14.3 Learning rule

We define the loss function such that minimizing the the loss function corresponds to maximizing the likelihood of observing the data vectors v, i.e.,

$$L(\theta) = -\log \prod_v p(v) \tag{176}$$

where we use the log-function because it greatly simplifies the math and since the log function is strictly monotonically increasing, it does not change the solution.

$$L(\theta) = -\sum_v \log p(v) \tag{177}$$

We define

$$l(\theta, v) = \log p(v) \tag{178}$$

To find the weights, we need the gradient:

$$\Delta w_{ij} = -\eta \frac{\partial L(\theta)}{\partial w_{ij}} = \eta \sum_v \frac{\partial l(\theta, v)}{\partial w_{ij}} \tag{179}$$

$$l(\theta, v) = \log p(v) = \log \sum_h p(v, h) \tag{180a}$$

At thermal equilibrium, the probability is given by the Boltzman equation Eq. 163

$$= \log \sum_h e^{-E(v,h)} - \log \sum_{v,h} e^{-E(v,h)} \tag{180b}$$

$$\frac{\partial l(\theta, v)}{\partial w_{ij}} = -\frac{1}{\sum_h e^{-E(v,h)}} \sum_h e^{-E(v,h)} \frac{\partial E(v,h)}{\partial w_{ij}} + \frac{1}{\sum_{v,h} e^{-E(v,h)}} \sum_{v,h} e^{-E(v,h)} \frac{\partial E(v,h)}{\partial w_{ij}} \tag{180c}$$

The gradient of the energy function is $\frac{\partial E(v,h)}{\partial w_{ij}} = -\frac{1}{2} s_i s_j$. Using the relationship $e^{-E(v,h)} = Z p(v,h)$ and $Z = \sum_{v,h} e^{-E(v,h)}$

$$= \frac{1}{2} \frac{1}{\sum_h Z p(v,h)} \sum_h Z p(v,h) s_i s_j - \frac{1}{2} \frac{1}{Z} \sum_{v,h} Z p(v,h) s_i s_j \tag{180d}$$

Since Z is a constant

$$= \frac{1}{2} \frac{1}{\sum_h p(v,h)} \sum_h p(v,h) s_i s_j - \frac{1}{2} \sum_{v,h} p(v,h) s_i s_j \tag{180e}$$

Using $\sum_h p(v,h) = p(v)$ and $p(h|v) = p(v,h)/p(v)$

$$= \frac{1}{2} \sum_h \frac{p(v,h)}{p(v)} s_i s_j - \frac{1}{2} \sum_{v,h} p(v,h) s_i s_j \tag{180f}$$

$$= \frac{1}{2} \sum_h p(h|v) s_i s_j - \frac{1}{2} \sum_{v,h} p(v,h) s_i s_j \tag{180g}$$

$$= \frac{1}{2} \mathrm{E}\left[s_i s_j\right]_{p(h|v)} - \frac{1}{2} \mathrm{E}\left[s_i s_j\right]_{p(v,h)} \tag{180h}$$

These summations are pratically impossible to calculate because the number of states is immense. Therefore, we can only sample these terms as an approximation of the true gradient.

$$\frac{\partial l(\theta, v)}{\partial w_{ij}} \approx \frac{1}{2} \left\langle s_i s_j \right\rangle_{p(h|v)} - \frac{1}{2} \left\langle s_i s_j \right\rangle_{p(v,h)} \tag{181a}$$

So, the weight updates are

$$\Delta w_{ij} = \eta \sum_v \left[\langle s_i s_j \rangle_{p(h|v)} - \langle s_i s_j \rangle_{p(v,h)} \right] \tag{181b}$$

$$= \eta \sum_v \left[\langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model} \right] \tag{181c}$$

Two phases are required to sample $\langle s_i s_j \rangle_{data}$ and $\langle s_i s_j \rangle_{model}$. The positive phase, where the visible units are clamped to the data, and a negative phase, where the visible units are free to change. The algorithm is summarized in Alg. 14.1. The first term stems from the numerator of the Boltzmann distribution Eq. 163. It increases the weight between two units that are strongly correlated in the data-driven distribution. That is how the network learns correlations between units. By contrast, the second term stems from the partition function, i.e., the denominator in Eq. 163, and reduces weights between units that are strongly correlated in the model distribution.

The update rule for the bias terms are equivalent:

$$\Delta b_i = \frac{\eta}{T} \sum_v \left[\langle s_i \rangle_{data} - \langle s_i \rangle_{model} \right] \tag{181d}$$

The learning rule for the Boltzmann machine is far more biologically plausible than backpropagation because the only information needed to change the weights is provided by "local" information. That is, the connection does not need information about anything other than the two units it connects. Nevertheless, the Boltzmann machine learning rule is impractical because the sampling takes a very long time due to the large number of states, and because of the time required to reach thermal equilibrium.

Algorithm 14.1 Training a Boltzmann Machine

repeat
  Permute the order of data points
  for all mini-batches (size around $10-100$) **do**
    {Sample from data distribution (positive phase)}
    repeat
      Randomly select a data point i
      Initialize visible units to x^i (and do not update visible units)
      Initialize hidden units to random values $\sim B(0.5)$
      repeat {burn-in phase}
        Update probabilities of hidden units according to Eq. 175 and sample activities
      until thermal equilibrium is reached
      repeat {sampling phase}
        Update probabilities of hidden units according to Eq. 175 and sample activities {visible units are clamped to input}
        Save state as s_{data}
      until enough samples have been collected
    until sufficient data points have been sampled
    {Sample from model distribution (negative phase)}
    Initialize visible and hidden units to random values $\sim B(0.5)$
    repeat {burn-in phase}
      Update probabilities of visible and hidden units according to Eq. 175 and sample activities
    until thermal equilibrium is reached
    repeat {sampling phase}
      Update probabilities of visible and hidden units according to Eq. 175 and sample activities
      Save state as s_{model}
    until enough samples have been collected
    Calculate $\langle s_i \rangle_{data}$ and $\langle s_i s_j \rangle_{data}$ for the mini-batch
    Calculate $\langle s_i \rangle_{model}$ and $\langle s_i s_j \rangle_{model}$ for the mini-batch
    Update weights and biases
  end for
until predetermined number of epochs is reached

14.4 Restricted Boltzmann machine

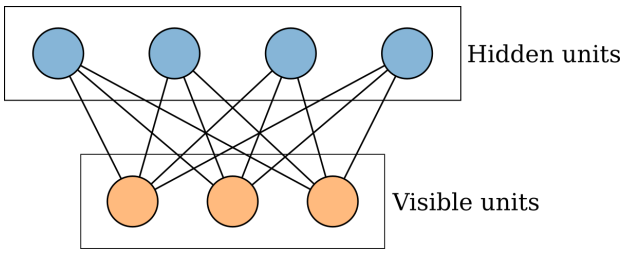


Figure 54: Restricted Boltzmann Machine (RBM).

Although learning is impractical in general Boltzmann machines, it can be made quite efficient in an architecture called the Restricted Boltzmann Machine or RBM, which does not allow intralayer connections.

Therefore, when the visible units are known, the activity of the hidden units can be updated in one step, and no Gibbs sampling is required. This greatly speeds up the learning process.

$$P(h_j = 1|v) = \sigma\left(\sum_i w_{ij} v_i + b_j^h\right) \tag{182}$$

$$P(v_i = 1|h) = \sigma\left(\sum_j w_{ij} h_j + b_i^v\right) \tag{183}$$

Algorithm 14.2 Training an RBM

repeat
  Permute the order of data points
  for all mini-batches **do**
    {Sample from data distribution (positive phase)}
    for all data points i in mini-batch **do** {data distribution}
      Initialize visible units to x^i (and do not update visible units)
      Update probabilities of hidden units according to Eq. 182 and sample activities
      Save state as s_{data}
    end for
    {Sample from model distribution (negative phase)}
    Initialize visible and hidden units to random values $\sim B(0.5)$
    repeat {burn-in phase}
      Update probabilities of hidden units according to Eq. 182 and sample activities
      Update probabilities of visible units according to Eq. 183 and sample activities
    until thermal equilibrium is reached
    repeat {sampling phase}
      Update probabilities of hidden units according to Eq. 182 and sample activities
      Update probabilities of visible units according to Eq. 183 and sample activities
      Save state as s_{model}
    until enough samples have been collected
    Calculate $\langle s_i \rangle_{data}$ and $\langle s_i s_j \rangle_{data}$ for the mini-batch
    Calculate $\langle s_i \rangle_{model}$ and $\langle s_i s_j \rangle_{model}$ for the mini-batch
    Update weights and biases
  end for{mini-batches}
until predetermined number of epochs is reached

14.4.1 Contrastive divergence

The difference between the positive and negative phases, is simply the number of iterations through the network. With every iteration $\langle s_i s_j \rangle_{model}$ becomes more and more different from $\langle s_i s_j \rangle_{data}$. This is called *contrastive divergence*. We could speed up Alg. 14.2 further by giving up on thermal equilibrium before we sample from the model distribution. So, in the contrastive divergence algorithm (Alg. 14.3), a fixed number of iterations k are used in the negative phase. Particularly in the beginning of training, the energy function is not correct and the gradient is very inexact. So, little

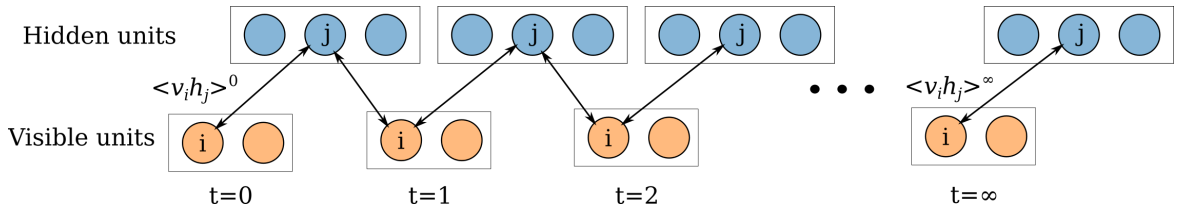


Figure 55: Contrastive divergence.

is gained by spending a lot of time to reach thermal equilibrium, which is needed to calculate the gradient precisely. Therefore, in the beginning of training, only one step of contrastive divergence, i.e., $k = 1$, is good enough to compute a gradient that moves the weights towards a better solution. As learning progresses and gradient descent approaches a good solution, more steps of contrastive divergence are needed to compute the gradient more exactly. There are several different schedules for increasing k with the number of gradient descent steps.

Algorithm 14.3 Constrastive divergence CD_k for RBM

repeat
  Permute the order of data points
  for all mini-batches **do**
    for all data points i in mini-batch **do**
      {Sample from data distribution (positive phase)}
      Initialize visible units to x^i (and do not update visible units)
      Update probabilities of hidden units according to Eq. 182 and sample activities
      Save state as s_{data}
      for k times **do**
        Update probabilities of visible units according to Eq. 183 and sample activities
        Update probabilities of hidden units according to Eq. 182 and sample activities
      end for
      Save state as s_{model}
    end for
    Calculate $\langle s_i \rangle_{data}$ and $\langle s_i s_j \rangle_{data}$ for the mini-batch
    Calculate $\langle s_i \rangle_{model}$ and $\langle s_i s_j \rangle_{model}$ for the mini-batch
    Update weights and biases
  end for{mini-batches}
until predetermined number of epochs is reached

Applications

- Pre-training a ffw network (unsupervised feature learning) Hinton (2006).
- Unsupervised learning of feature representations when little labelled data is available.

Additional material:

- "A Practical Guide to Training Restricted Boltzmann Machines" by Geoffrey Hinton

Figure 56: Reconstructing images of hand-written digits. 1. row: original image. 2. row: reconstruction by RBM. 3. row: reconstruction by logistic PCA. 4. row: reconstruction by regular PCA. Source: Hinton (2006), Fig. 2B.

14.5 Problem set

1. **Contrastive Divergence.** You are provided with an unfinished implementation of a restricted Boltzmann machine (RBM) in *prob_rbm.py*. Implement a function *train* that trains the RBM using the 1-step contrastive divergence (CD) algorithm. Your function should make it possible to train the RBM using either the hidden units' activation probabilities or binary states during the positive phase.

2. **Training RBMs.** Load the *digits* data set provided by *sklearn*. Use the first 1200 examples as your training set and the rest as your test set (Note: Make sure to normalize the data). Do the following:

   (a) Train two RBMs, M_{binary} and M_{prob}, for 100 epochs, which use binary activations and activation probabilities during the positive phase, respectively. Use 16 hidden units and a learning rate of 0.01.

   (b) Save the trained RBMs.

   (c) Implement a function *reconstruction* that reconstructs the visible units for a given input (Note: During the positive phase do not use binary hidden units). Visualize the reconstructions of the first 32 examples of the test set for both RBMs. For comparison also visualize the ground truths.

   (d) Compute the mean squared error error (MSE) E_{rec} over all image pixels on the reconstructions of the test set. Plot E_{rec} as a function of training epoch for both RBMs. How do the error curves for the two RBMs compare?

3. **Pattern completion.** RBMs are generative models and are also able to repair or denoise corrupted input. Use the digits dataset and the two RBMs, M_{binary} and M_{prob}, from the previous problem. Load the networks you saved in problem 2b.

   (a) Apply increasing levels of Gaussian noise with $\sigma = \{0., 0.05, 0.1, ..., 0.45, 0.5\}$ to the images in the test set and let the RBMs reconstruct the input.

   (b) Compute E_{rec} for the reconstructions and plot E_{rec} as a function of noise level σ. How do the two RBMs differ in their ability to cope with increasing levels of noise?

4. **Data confabulation.** Generative models RBMs can confabulate new data. Use the two RBMs, M_{binary} and M_{prob}, from the previous problem and load the networks you saved in problem 2b.

   (a) Generate 16 random binary images. The images should be sparse with only 5%-10% units being active. Feed the random images to both RBMs and let them "reconstruct" the images for multiple iterations ($20-30$ iterations should suffice). Make sure to store the "reconstructed" images after each iteration.

   (b) Visualize the images confabulated by both RBMs at each iteration step. How do the images confabulated by the two RBMs, M_{binary} and M_{prob}, differ? How do they change over the course of iterations?

15 Bibliography

Amit, D. J., Gutfreund, H., & Sompolinsky, H. (1985). Storing Infinite Numbers of Patterns in a Spin-Glass Model of Neural Networks. *Phys. Rev. Lett.*, *55*(14), 1530–1533.

Azizi, A. H., Pusch, R., Koenen, C., Klatt, S., Bröker, F., Thiele, S., Kellermann, J., Güntürkün, O., & Cheng, S. (2019). Emerging category representation in the visual forebrain hierarchy of pigeons (Columba livia). *Behavioural Brain Research*, *356*, 423–434.

Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.*, *60*(2), 223–311.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv*, *cs.CL*, 1406.1078.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., & Schmidhuber, J. (2011). Flexible, High Performance Convolutional Neural Networks for Image Classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, volume 2, (pp. 1237–1242)., Barcelona, Spain.

Deng, J., Dong, W., Socher, R., Li, L.-j., Li, K., & Fei-fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *In CVPR*.

Glorot, X. & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, (pp. 249–256).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. The MIT Press.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, (pp. 1026–1034).

Hinton, G. E. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, *313*(5786), 504–507.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv*, *cs*, 1207.0580.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc Natl Acad Sci U A*, *79*(8), 2554–2558.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proc. IEEE*, *86*(11), 2278–2324.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*(4), 115–133.

Minsky, M. & Papert, S. (1969). *Perceptrons*. Perceptrons. Oxford, England: M.I.T. Press.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychol. Rev.*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Siegel, C., Daily, J., & Vishnu, A. (2016). Adaptive Neuron Apoptosis for Accelerating Deep Learning on Large Scale Systems. *arXiv*, *cs.NE*, 1610.00790.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.*, *15*, 1929–1958.

Appendix A: Introduction to scientific computing in python

Problem set: Python Programming

1. **Make a Q-Q-plot to check visually whether a sample distribution is normal.**

   A Q-Q-plot can be applied to visually judge whether a given sample was drawn from a certain theoretical distribution. This can be useful for instance before choosing a statistical test that requires normally distributed samples. Q-Q stands for quantile-quantile because each point in the scatter plot is a quantile of one sample distribution plotted against the same quantile of another sample distribution. If the points deviate systematically from a straight line, then the sample was probably not drawn from the theoretical distribution. In our case, we want to compare a sample to a normal distribution.

   The task is to first generate a sample as well as normal distributed data, then plot each distribution separately, and, finally, the Q-Q-plot for the two distributions (Fig. 57). Below you find step-by-step instructions to help you with this task.

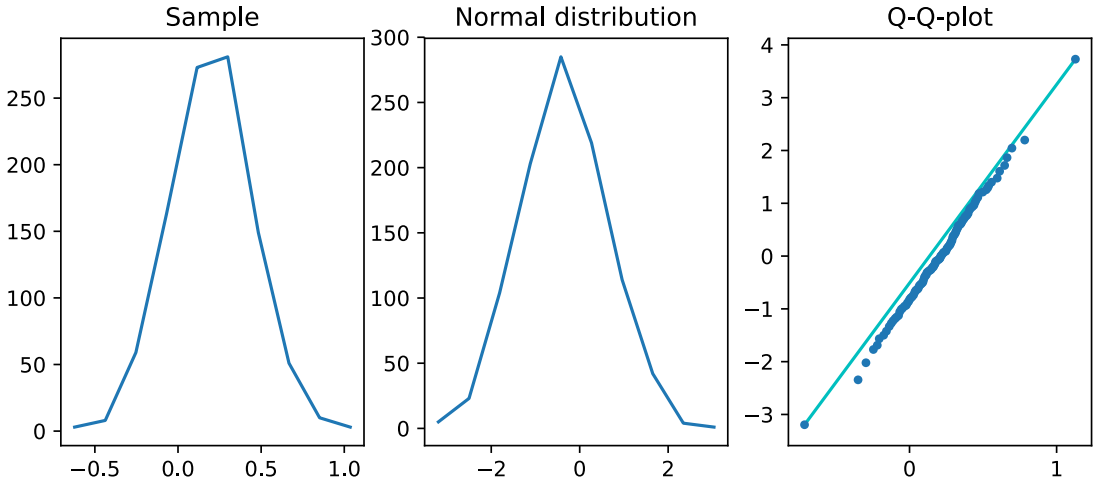


Figure 57: Example figure output

   (a) Obtain sample data. This can be any series of single measurements. You can be creative here, or use the following suggestion.

   Generate a set of points uniformly distributed within a square. Compute your data as the Euclidean distance of each point to the center of the square.

   - Generating $N = 1000$ data points should work well.
   - With numpy you can generate arrays with random values.
     The function `numpy.random.rand()` produces uniformly distributed values between 0 and 1 in the given shape. For instance, the call `numpy.random.rand(10)` generates a 10-element vector and the call `numpy.random.rand(5,4)` generates a 5-by-4 matrix containing random values.
     By multiplying the result with a constant you can scale the number range. Also, you can add a constant to shift.
   - What is the Euclidean distance?
     The Euclidean distance d from q to p is the "length" of the vector pointing from q to p.

$$d = \sqrt{(q-p)\cdot(q-p)} = ||q-p|| \tag{184}$$

     Calculating this requires several steps, which can at the end be integrated in a single line.

     (i) *Difference.* With numpy you can use operators between arrays of identical shape to perform element-wise operations. However, you can also use operators between arrays with a different number of dimensions when the shared dimensions are of the same size. For instance, you can add a 3-element vector to a 5-by-3 matrix. This is called "broadcasting", try it out!

With that in mind you can calculate the difference between each each point and the center of the square using only one operator.

(ii) *Scalar product with itself.* You can do this in two steps: First, square each element. Second, sum the entries of each vector up. You can do this for all vectors at the same time by using the `numpy.sum()` function with the `axis` argument (see the slides or the numpy docs).

(iii) *Square root.* You can use the `numpy.sqrt()` function.

(b) Plot the sample distribution. You do not need to normalize the distribution. See Fig. 57 (left) for an example. Your distribution will look different.

- The non-normalized distribution is a histogram. You can have a quick look at it by using the function `pyplot.hist()`. This function generates a bar plot. However, what we want is a line plot to make it look like a distribution function. The function `numpy.histogram()` returns the histogram data (values and bin edges) without generating a plot. You can look it up in the numpy docs.
- The function `pyplot.plot()` needs an array for each the x- and y-coordinate of the data points. Conveniently, the y's are the histogram values. For x, though, you cannot use the bin edges because they have one too many data points. If you want to be precise, calculate the histogram bin centers from the edges and use those as x.
- You have to use subplots to generate several plots next to each other in one figure. You can use `pyplot.subplot()`. Check the slides or the pyplot docs.

(c) Generate normal distributed data for comparison and plot it next to the sample distribution.

- The function `numpy.random.randn()` generates a sample of normal distributed values in a given shape, similar to `numpy.random.rand()` that was introduced in (a).

(d) Make the Q-Q-plot. Generate an ascending range of numbers q_i between 0 and 1 and calculate the q_i-quantiles. Make a scatter plot. You can add the diagonal line from the 0- to the 1-quantile. Note that this is not necessarily the best line to reference the data to.

- What is a quantile?
  The 0.25-quantile, for instance, is the x-value of the distribution such that 25% of the observations have a lower value (= are to the left of the quantile). Similarly, the 0-quantile is always the lowest value of the distribution and the 1-quantile the highest.
- You can use the function `numpy.quantile()`. You can look it up in the numpy docs.

2. **Bonus: Test whether a number is prime**

Write a function that tests whether a given integer $x >= 2$ is a prime number and returns `True` or `False` accordingly. In that function test whether any whole number in an appropriate range divides x.

- How to test whether a number divides another number?
  `9//4` yields `2`, but `9/4` yields `2.25`.
  `9//3` yields `3`, and `9/3` also yields `3`.
- What is an appropriate range?
  Every number (even a prime number) is divisible by itself and by 1.
  What is the smallest, and what is the largest number (potential divisor) we need to test?

3. **Bonus: Calculate π by random sampling**

"Construct" a square that contains a circle. The radius of the circle is equal to the side length of the square. Randomly sample uniformly distributed points within the square. Observe the number of points C that appear within the circle. See Fig. 58 for a visualization of that sampling process.

(a) Do not use loops, but make use of the functionality of numpy.

- How to calculate π from C and N, the total number of points?
  Start with the ansatz that the ratio $\frac{C}{N}$ equals the ratio of the surface areas of the circle and the square. Solve for π.

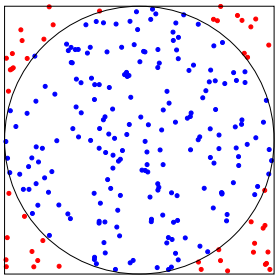


Figure 58: Sampling points within the square

- $N = 10^6$ should work well.

(b) To improve the approximation, put your sampling into a function that returns its approximation for π. Average over multiple (e.g. 1000) calls of the function. Here you can use a loop.

Solutions to set: Python Programming

1. **Make a Q-Q-plot to visualize "how Gaussian" a sample distribution is**

```
import numpy as np
import matplotlib.pyplot as plt

N = 1000
BINS = 20

pts = np.random.rand(N, 2)*2 - 1
ct = np.zeros(2)

dists = np.sqrt(np.sum((pts - ct) ** 2, axis=1))  # Euclidean distance

hist, hedges = np.histogram(dists, BINS)
hx = hedges[:-1] + (hedges[1]-hedges[0])/2

plt.subplot(1,3,1)
plt.plot(hx, hist)
plt.title("Data")

norm = np.random.randn(N)

hist_norm, hedges_norm = np.histogram(norm)
hx_norm = hedges_norm[:-1] + (hedges_norm[1]-hedges_norm[0])/2

plt.subplot(1,3,2)
plt.plot(hedges_norm[:-1], hist_norm)
plt.title("Normal distribution")

quants = np.arange(0, 1.01, 0.01)

qx = np.quantile(dists, quants)
qy = np.quantile(norm, quants)

plt.subplot(1,3,3,)
plt.plot([qx[0], qx[-1]], [qy[0], qy[-1]], 'c', zorder=1)
```

```
35  plt . scatter (qx, qy, s=10, zorder=2)
36  plt . title ("Q-Q-plot")

38  plt .show()
```

2. **Test whether a number is prime**

   - Fixed range - runtime $O(x)$

```
1  def prime(x):
2      for i in range(2, x//2+1):
3          if x// i == x/i:
4              return False
5      return True
```

   - Dynamic range - runtime $O(\sqrt{x})$ (much faster)

```
1  def prime(x):
2      i = 2
3      while True:
4          stop = x// i
5          if i > stop:
6              break
7          if stop == x/i:
8              return False
9          i += 1
10     return True
```

3. **Calculate π by random sampling**

```
1  import numpy as np

3  N2 = 10**3
4  N = 10**6
5  r = 1   # radius of the circle / side length of the square

7  def pi_appr(N, r):
8      points = np.random.rand(N, 2) * r * 2 - r
9      C = len(np.nonzero(r**2 > (points [:,0]**2 + points [:,1]**2))[0])
10     pi_appr = (4*C) / N
11     return pi_appr

13 appr = []
14 for i in range(N2):
15     appr.append(pi_appr(N,r))

17 pi = np.mean(appr)
18 print("Approximation of pi: {:.6f}".format(pi))
19 print("Actual value of pi: {:.6f}".format(np.pi))
```

Appendix B: Solutions to problem sets

Solutions to set 1: Introduction

1. **Definition of functions**

   f is a function because every x is mapped onto exactly one y (left panel, Fig. 59). g, however, is not a function. If we rewrite $y^2 = x$ as $y = \pm\sqrt{x}$, it is clear that all $x > 0$ are mapped onto two y (right panel, Fig. 59).

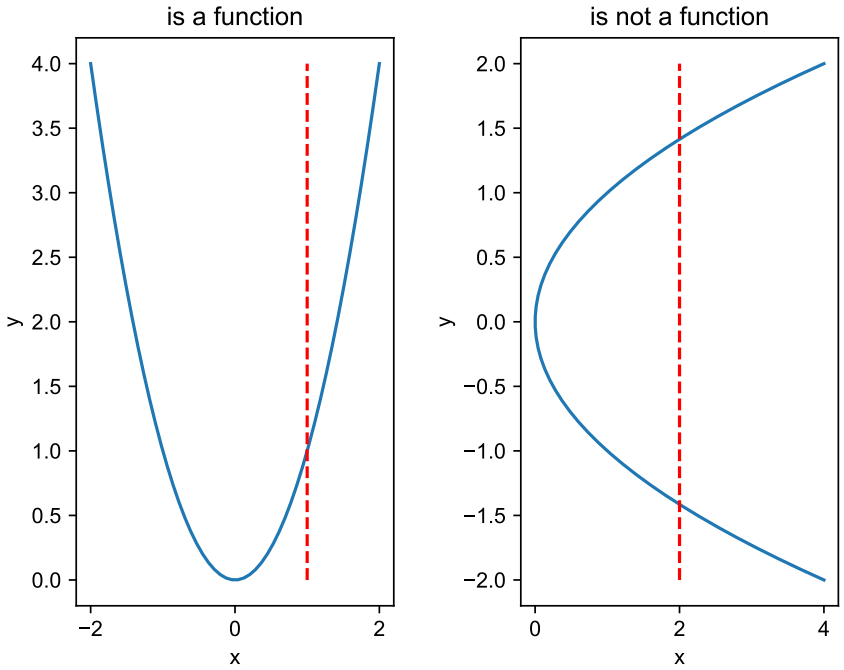


Figure 59: Plotting the mappings f (left panel) and g (right panel) vs. x. Any line parallel to the vertical axis has only one intersection with f, so $f(x)$ is a function. Any vertical line for $x > 0$ has two intersections with g and hence g is not a function.

```
# Python packages required for this assignment

import numpy as np
import matplotlib.pyplot as plt

# Declared functions

def fx(x):
    return x**2

# For g: y^2=x it is easier to think of computing x for different y and
# then plot the computed x on the horizontal axis and y on vertical axis.
def ginv(gy):
    return gy**2

# Main body of script the


# Costumizing the fonts used in plots
```

```
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = 'Arial'
plt.rcParams['font.size'] = 10

x = np.linspace( start =-2, stop=2, num=50)#Generate 50 linearly-spaced
                                            #numbers in [-2, 2]
fy = fx(x)
gy = x
gx = ginv(gy)  #Computing x for different values of g(x)

#make figure with two subplots
fig, ax = plt.subplots(nrows=1, ncols=2,
                       gridspec_kw={'wspace':0.4})

ax[0].plot(x, fy)  #Choosing the first subplot and plot the function
ax[0].plot([1, 1], [0, 4], '--r')
ax[0].set_xlabel('x')        #Set x label for the first subplot
ax[0].set_ylabel('y')  #Set y label ...
ax[0].set_title('is a function')

ax[1].plot(gx, gy)  #Choosing the second subplot and ...
ax[1].plot([2, 2], [-2, 2], '--r')
ax[1].set_xlabel('x')
ax[1].set_ylabel('y')
ax[1].set_title('is not a function')
# Save the figure
fig.savefig('a1-e1-function.pdf', format='pdf')
```

2. **Types of functions**

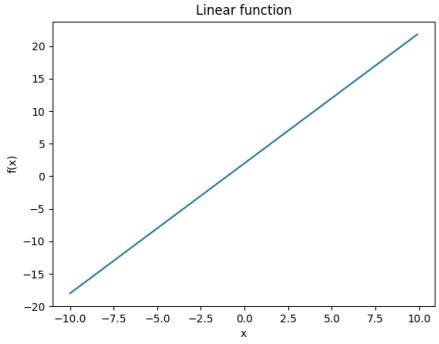
The graphs of the functions can be found in Fig. 60.

i. **Parametrized functions:** It is easy to determine what the parameters do in a linear function by changing them by hand (Fig. 61). Changing the bias causes the linear function to be offset in the y-axis. Changing the scaling factor in the linear function (the a in $ax+b$) will cause the slope of the line to change.

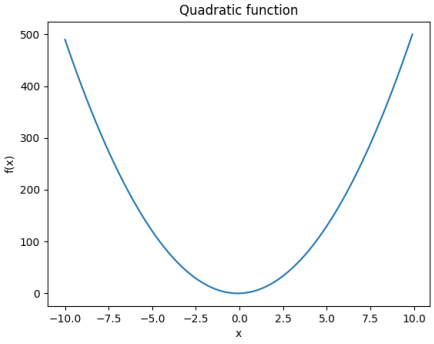
ii. In quadratic functions, changing the bias causes the function to be offset in the y-axis and changing the quadratic term (the a in ax^2+bx+c) causes the parabola to become narrower (Fig. 62).

Extra: The quadratic function (c) includes also a linear term that somehow does not contribute to the output. The reason is that the quadratic term has a higher multiplier, and moreover grows faster than the linear term. If you adjust the values the other way around and plot(x^2+5x) you will see a different story. Could you explain why the function has become imbalanced?

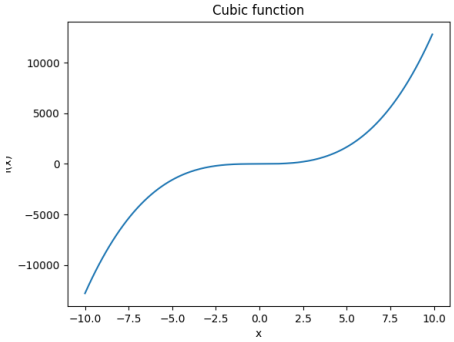
```
# python libraries required for the assignment
import numpy as np
import matplotlib.pyplot as plt


# define exponential function
def exponential(x, bias=0):
    return np.exp(x)+bias
```



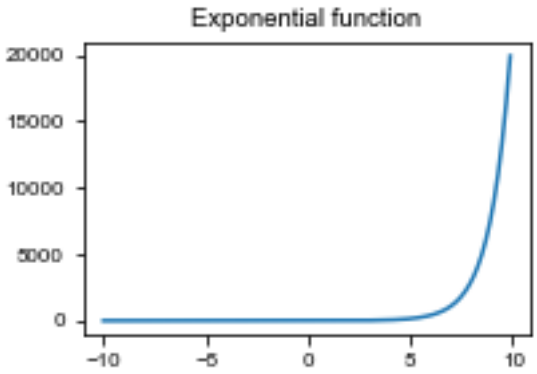
(a) Linear function



(b) Quadratic function

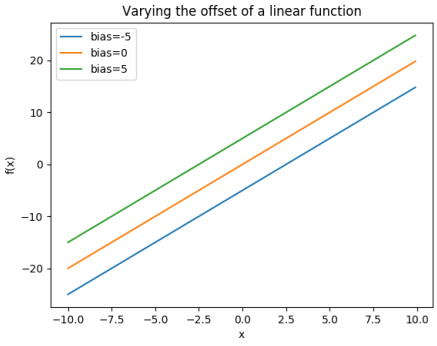


(c) Cubic function

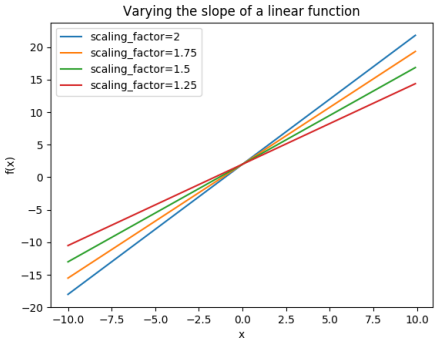


(d) Exponential function

Figure 60: Functions

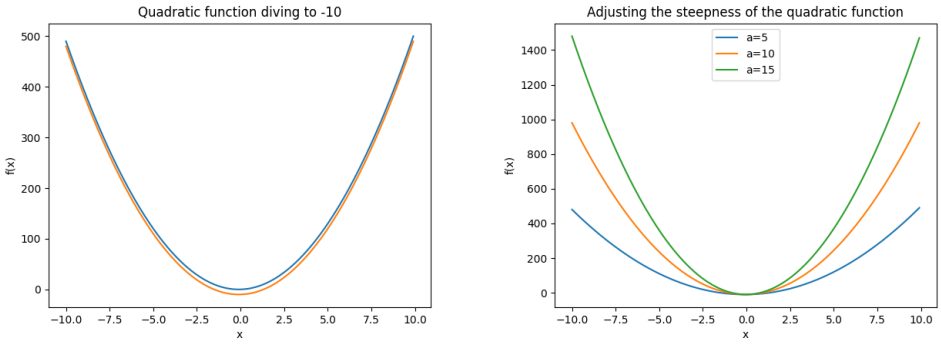


(a) Changing the bias



(b) Changing the scaling factor

Figure 61: Parametrization of linear function.



(a) Changing the bias

(b) Changing the quadratic term

Figure 62: Parametrization of quadratic function.

```
# define linear function
def linear(x, scaling_factor=2, bias=2):
    return scaling_factor*x + bias

# define quadratic function
def quadratic(x, scaling_factors=[11,2], bias=0):
    # making sure that this function is used correctly
    assert isinstance(scaling_factors, list), \
        "scaling factors should be a list of two elements"
    assert len(scaling_factors)==2, \
        "scaling factors should be a list of two elements"

    # unpack scaling factors
    a, b = scaling_factors

    # actual definition
    return a*x**2 + b*x + bias

# define polynomial with x^3
def cubic(x, scaling_factors=[11, 2, 2], bias=0):
    # sanity checks
    assert isinstance(scaling_factors, list), \
        "scaling factors should be a list of three elements"
    assert len(scaling_factors)==3, \
        "scaling factors should be a list of three elements"

    # unpack scaling factors
    a, b, c = scaling_factors

    # actual definition
    return a*x**3 + b*x**2 + c*x**3 + bias

# define x as an interval between -10 and 10
x=np.arange(start=-10,step=0.1,stop=10)

# plot the functions

# Costumizing the fonts used in plots
```

```
48  plt.rcParams['font.family'] = 'sans-serif'
49  plt.rcParams['font.sans-serif'] = 'Arial'
50  plt.rcParams['font.size'] = 8

52  # exponential
53  fig=plt.figure(figsize=(3,2)) # set up figure
54  plt.title('Exponential function')
55  plt.plot(x, exponential(x, bias=0)) # try the parameter here!

57  # linear
58  fig2=plt.figure() # set up figure
59  plt.title('Linear function')
60  plt.plot(x, linear(x, scaling_factor=2, bias=2)) # try the parameters here!


63  # quadratic
64  fig3=plt.figure() # set up figure
65  plt.title('Quadratic function')
66  plt.plot(x, quadratic(x, scaling_factors=[5,1], bias=0)) # try the parameters here!

68  # cubic
69  fig4=plt.figure() # set up figure
70  plt.title('Cubic function')
71  plt.plot(x,cubic(x, scaling_factors=[11,2,2], bias=0)) # try the parameters here!


74  # changing offset of linear function
75  fig5=plt.figure() # set up figure
76  plt.title('Varying the offset of a linear function')
77  plt.plot(x, linear(x, scaling_factor=2, bias=-5))
78  plt.plot(x, linear(x, scaling_factor=2, bias=0))
79  plt.plot(x, linear(x, scaling_factor=2, bias=5))
80  plt.legend(labels=['b=-5', 'b=0', 'b=5'])

82  # changing scaling factor
83  fig6=plt.figure() # set up figure
84  plt.title('Varying the slope of a linear function')
85  plt.plot(x, linear(x, scaling_factor=2, bias=2))
86  plt.plot(x, linear(x, scaling_factor=1.75, bias=2))
87  plt.plot(x, linear(x, scaling_factor=1.5, bias=2))
88  plt.plot(x, linear(x, scaling_factor=1.25, bias=2))
89  plt.legend(labels=['a=2', 'a=1.75',
90                     'a=1.5', 'a=1.25'])


93  # changing offset of quadratic function
94  fig7=plt.figure() # set up figure
95  plt.title('Quadratic function diving to -10')
96  plt.plot(x, quadratic(x, scaling_factors=[5,1], bias=-10))

98  # changing steepness of quadratic function
99  fig8=plt.figure() # set up figure
100 plt.title('Adjusting the steepness of the quadratic function')
```

```
plt.plot(x, quadratic(x, scaling_factors=[5,1], bias=-10))
plt.plot(x, quadratic(x, scaling_factors=[10,1], bias=-10))
plt.plot(x, quadratic(x, scaling_factors=[15,1], bias=-10))
plt.legend(labels=['a=5', 'a=10','a=15'])
```

Solutions to set 2: Optimization

1. **Analytical minimization of a function**

(a) $\frac{d}{d\theta}L(\theta) = 0$:

$$\begin{aligned} \frac{d}{d\theta}L(\theta) &= 4\theta^3 - 8\theta = 4\theta(\theta^2-2) = 4\theta(\theta-\sqrt{2})(\theta+\sqrt{2}) \stackrel{!}{=} 0 \\ \theta &= -\sqrt{2}, 0, \sqrt{2} \end{aligned}$$

(b) To distinguish minima (desired in minimizing cost functions) from maxima we must look at the second derivative $\frac{d^2}{dx^2}L(\theta)$ at θ's where $\frac{d}{d\theta}L(\theta) = 0$. For the minima $\frac{d^2}{dx^2}L(\theta) > 0$ and for maxima $\frac{d^2}{dx^2}L(\theta) < 0$.

$$\begin{aligned} \frac{d^2}{d\theta^2}L(\theta) &= L''(\theta) = 12\theta^2 - 8 \\ L''(-\sqrt{2}) &= 16 > 0 \text{ (minimum)} \\ L''(0) &= -8 < 0 \text{ (maximum)} \\ L''(\sqrt{2}) &= 16 > 0 \text{ (minimum)} \end{aligned}$$

(c) The visualization of the loss function and its minima and maximum (Fig. 63).

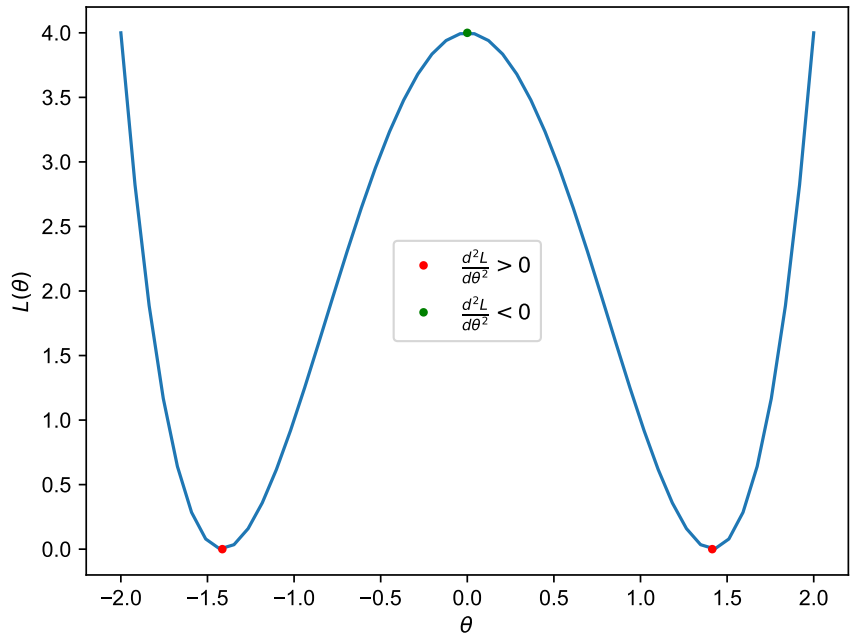


Figure 63: The loss function $L(\theta)$ is in blue, the minima are in red and the maximum is in green.

```
# Python packages required for this assignment

import numpy as np
import matplotlib.pyplot as plt

# Declared functions

def L(x):                #Cost function
    return x**4-4*x**2+4

def d2L(x):              #Second derivative of the cost function
    return 12*x**2-8


# Costumizing the fonts used in plots
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = 'Arial'
plt.rcParams['font.size'] = 10

x = np.linspace( start =-2, stop=2, num=50)
l = L(x)

fig, ax = plt.subplots()

#roots of the first derivative
root_dldx = np.array([-np.sqrt(2), 0, np.sqrt(2)])
#Points on L(x) that corresponds to the roots of the first derivative of L
lroot = L(root_dldx)

ax.plot(x, l)
ax.set_xlabel(r'$\theta$')
ax.set_ylabel(r'$L(\theta)$')
d2l = d2L(root_dldx)
ax.plot(root_dldx[d2l>0], lroot[d2l>0], '.r',
        label=r'$\frac{d^2L}{d\theta^2}>0$')
ax.plot(root_dldx[d2l<0], lroot[d2l<0], '.g',
        label=r'$\frac{d^2L}{d\theta^2}<0$')
ax.legend()
# Saving the figure
fig.savefig('a2-e1-lossfunction-extrema.pdf',
            format='pdf')
```

2. **Numerical minimization of a function**

(a)

```
import numpy as np

def grad_descent(l_rate, num_iter, init_min, dL):
    '''
    Inputs
    l_rate: learning rate
    num_iter: number of iterations
    init_guess: initial guess for the minimum
```

```
    dL: derivative of the loss function

    Outputs
    mins: An array of the estimated minimum for each iteration .
          mins[-1] gives the final estimation .
    '''
    minima = init_min*np.ones(num_iter+1)
    for iter in range(num_iter):
        min_ = init_min - l_rate*dL(init_min)
        init_min = min_
        mins[it+1] = min_
    return mins
```

(b) Gradient descent can only find one minimum based on the initial guess that was chosen. When the initial guess is negative gradient descent finds the leftmost minimum and when the guess is positive gradient descent finds the rightmost minimum (Fig. 64).

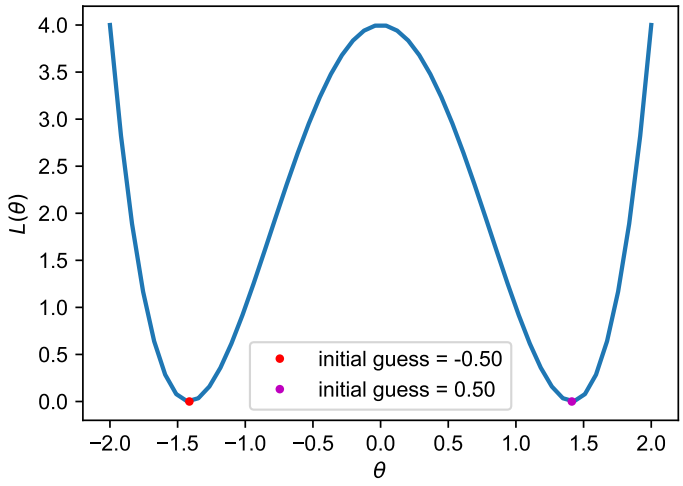


Figure 64: The role of the the initial guess on the estimation of the minimum.

(c) For the given learning rate (η) and initial value, as η increases the estimation becomes less precise (Fig. 65a).

(d) For the given number of iterations (N) and initial value, as N increases the estimation becomes more precise (Fig. 65b).

```
# Python packages required for this assignment

import numpy as np
import matplotlib.pyplot as plt


def L(x):                    #Loss function
    return x**4-4*x**2+4

def dLdx(x):                 #Analytical derivative of loss function
    return 4*x**3-8*x
```

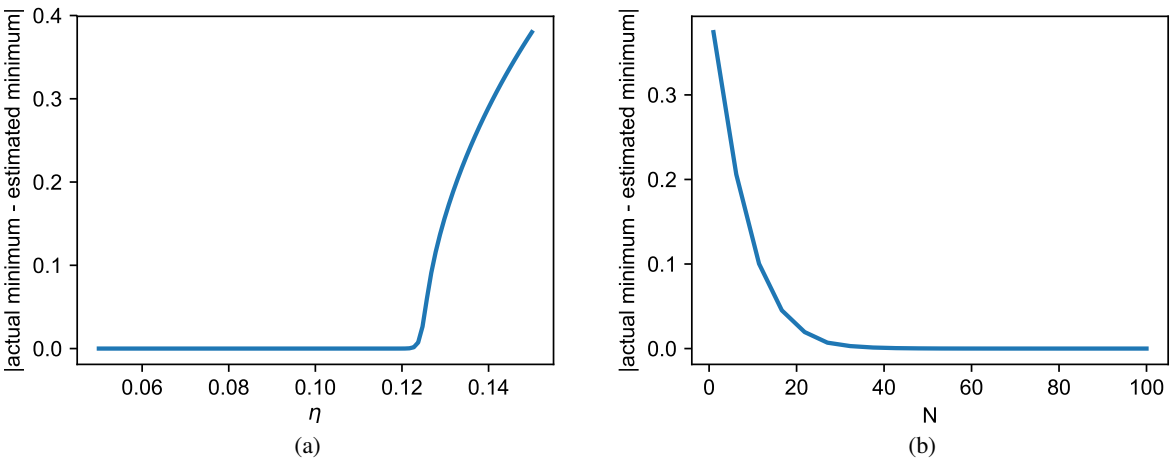


Figure 65: The role of η (a) and N (b) on the precision of the estimation.

```
13 def dL_num(x):     #Numerical implementation of derivation
14     dx=1e-4#Precision of neighboring point for computation of the derivative
15     dif = (L(x+dx)-L(x))/dx
16     return dif

18 def grad_descent( l_rate , num_iter, init_min , dL):
19     mins = init_min *np.ones(num_iter+1)
20     for it in range(num_iter):
21         min_ = init_min - l_rate *dL(init_min)
22         init_min = min_
23         mins[it+1] = min_
24     return mins


27 # Costumizing the fonts used in plots
28 plt.rcParams['font.family'] = 'sans-serif'
29 plt.rcParams['font.sans-serif'] = 'Arial'
30 plt.rcParams['font.size'] = 10

32 '''
33 PART B
34 '''

36 init_min = -.5          # Initial guess
37 learning_rate = 0.01
38 num_iter = 50          #Number of iterations

40 est_min = grad_descent( learning_rate , num_iter, init_min , dLdx)
41 min_ = est_min[-1]
42 l_min = L(min_)

44 x = np.linspace( start =-2, stop=2, num=50)
45 l = L(x)

47 fig , ax = plt.subplots( figsize =(5,3.5))
```

```
49  ax.plot(x, l, linewidth=2)
50  ax.set_xlabel(r'$\theta$')
51  ax.set_ylabel(r'$L(\theta)$')
52  ax.plot(min_, l_min, '.r', label='initial guess = %.2f' %init_min)
53  ax.legend()


56  init_min = .5                # Initial guess

58  est_min = grad_descent( learning_rate , num_iter, init_min , dLdx)
59  min_ = est_min[-1]
60  l_min = L(min_)

62  ax.plot(min_, l_min, '.m', label='initial guess = %.2f' %init_min)
63  ax.legend()
64  fig.savefig('a2-e2-initguess.pdf', format='pdf')


67  '''
68  PART C
69  '''

71  init_min = 1              # Initial guess
72  learning_rate = np.linspace(0.05, 0.15, 100)
73  num_iter = 100            #Number of iterations
74  mins = np.zeros_like( learning_rate )

76  for ind, l_ in enumerate( learning_rate ):
77      est_min = grad_descent(l_, num_iter, init_min , dLdx)
78      mins[ind] = est_min[-1]

80  fig , ax = plt.subplots( figsize =(4,3))
81  ax.plot( learning_rate , np.abs(mins-np.sqrt(2)), linewidth=2)
82  ax.set_xlabel(r'$\eta$')
83  ax.set_ylabel('|actual minimum - estimated minimum|')
84  fig.savefig('a2-e2-error-lrate.pdf', format='pdf')

86  '''
87  PART D
88  '''

90  init_min = 1              # Initial guess
91  learning_rate = 0.01
92  num_iters = np.linspace(1,100, 20)          #Number of iterations
93  mins = np.zeros_like(num_iters)

95  for ind, num_iter in enumerate(num_iters):
96      est_min = grad_descent( learning_rate , int(num_iter), init_min , dLdx)
97      mins[ind] = est_min[-1]

99  fig , ax = plt.subplots( figsize =(4,3))
100 ax.plot(num_iters , np.abs(mins-np.sqrt(2)), linewidth=2)
```

```
101  ax. set_xlabel ('N')
102  ax. set_ylabel ('| actual minimum - estimated minimum|')
103  fig . savefig ('a2-e2-error-N.pdf', format='pdf')
```

3. **The role of the learning rate**

   (a) The minimum of $L(\theta)$ is 0 (Fig. 66).

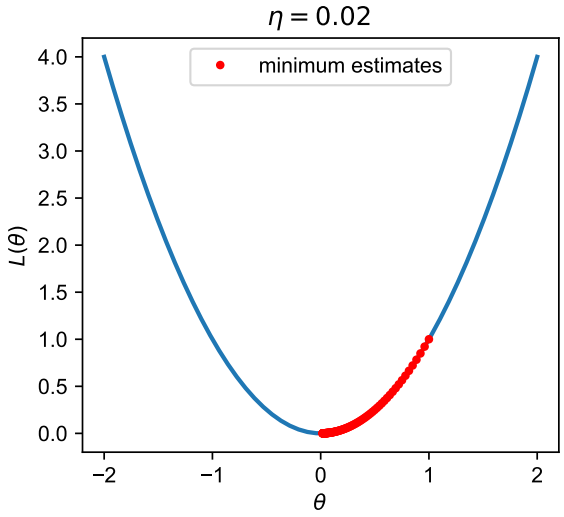


Figure 66: The estimate gets closer and closer as the gradient descent reaches its last iterations.

   (b) When $\eta = 1.1$ the estimate diverges and in this case more iterations will make the estimate worse (Fig. 67a). When $\eta = 0.1$ the estimate converges, but the number of iterations in this case, $N = 4$, is not sufficient for the estimate to reach the minimum (Fig. 67b).

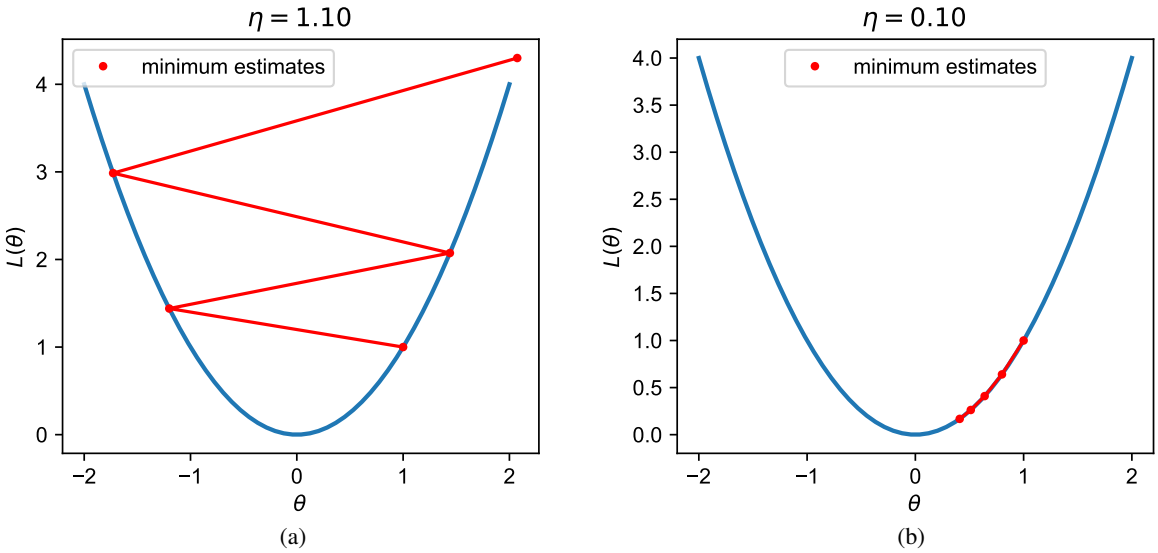


Figure 67: Diverging (a) and converging (b) estimates.

```
import numpy as np
import matplotlib.pyplot as plt

def grad_descent(l_rate, num_iter, init_min, dL):
    mins = init_min*np.ones(num_iter+1)
    for it in range(num_iter):
        min_ = init_min - l_rate*dL(init_min)
        init_min = min_
        mins[it+1] = min_
    return mins

def L(x):                    #The simple loss function
    return x**2

def dL(x):
    return 2*x

def dL_num(x):   #Numerical implementation of derivation
    dx=1/10000#Precision of neighboring point for computation of the derivative
    dif = (L(x+dx)-L(x))/dx
    return dif

def regressor(a, x):
    return a*x

def loss(y, a, x):
    return np.sum(np.abs(y - regressor(a, x)))

def plot_grad_descent(learning_rate, num_iter, init_min, filename):
    min_vec = grad_descent(learning_rate, num_iter, init_min, dL)
    x = np.linspace(start=-2, stop=2, num=50)

    fig, ax = plt.subplots(figsize=(4,3.5))

    ax.plot(x, L(x), linewidth=2)
    ax.plot(min_vec, L(min_vec), color='red')
    ax.plot(min_vec, L(min_vec), '.r', label='minimum_estimates')
    ax.set_ylabel(r'$L(\theta)$')
    ax.set_xlabel(r'$\theta$')
    ax.set_title(r'$\eta=%.2f$' % learning_rate)
    ax.legend()
    fig.savefig(filename, format='pdf')

# Costumizing the fonts used in plots
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = 'Arial'
plt.rcParams['font.size'] = 10


# PART A
plot_grad_descent(0.02, 100, 1, 'a2-e3-minimum.pdf')

# PART B
```

```
plot_grad_descent (1.1, 4, 1, 'a2-e3-diverging.pdf')
plot_grad_descent (.1, 4, 1, 'a2-e3-converging.pdf')
```

Solutions to set 3: Regression

1. **Analytical solution of linear regression**

   (a) Scatter plot of y against x (Fig. 68)

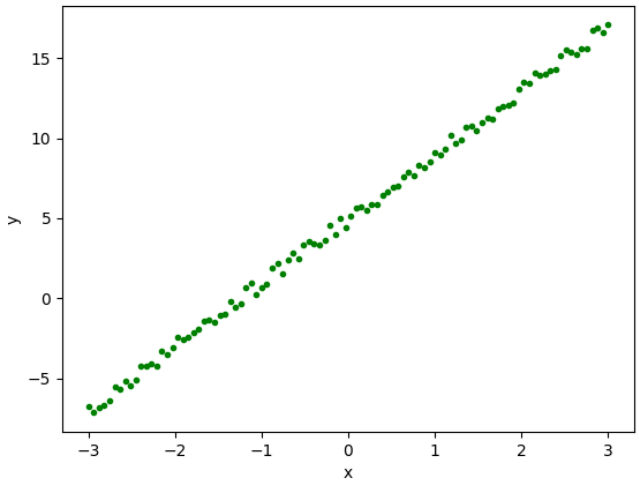
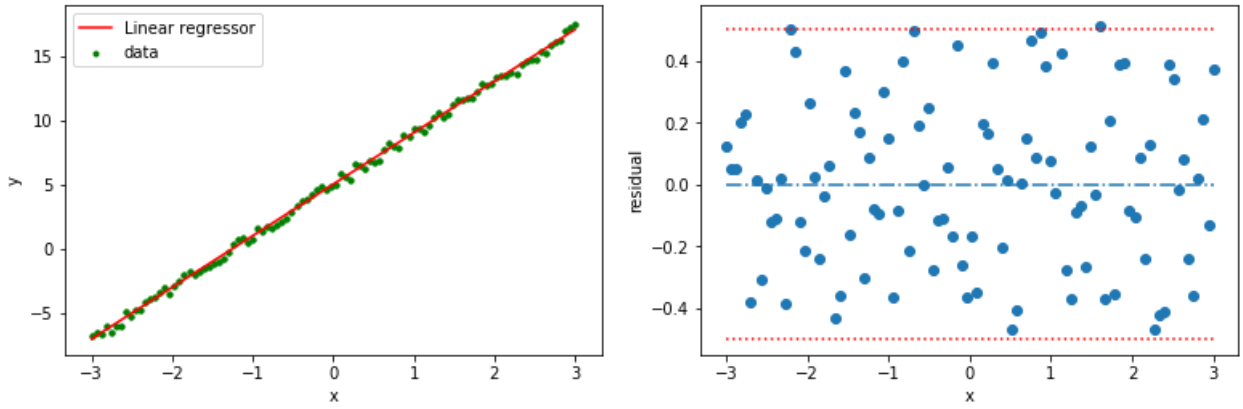


Figure 68: Data

   (b) The estimation of parameters with 4 floating point precision are: $m = 4.0057$ and $b = 4.9547$. You will most probably get different estimates due to the difference in the random number generator used to generate ε. In other words, you will get a different y vector due to a different ε.

   (c) Parameters estimated using the analytical solution for multiple linear regression: $m = 4.0057$ and $b = 4.9547$. The parameters estimated in this part must be identical to solution 1b, when the dataset is identical.

   (d) Plotting regression line together with the data (Fig. 69a) and its corresponding residual plot (Fig. 69b). Explained variance (R2) is 0.998353.



(a) (b)

Figure 69: (a) Data and the linear regressor. (b) The residual plot of the linear regressor.

(e) The estimated parameters for multiple linear regression using sklearn: $m = 4.0057$ and $b = 4.9547$ and R^2=0.998353.

```
import numpy as np
import matplotlib.pyplot as plt

#.....Solution 1a.....#
size = 100
true_params = [5,4]
x = np.linspace(−3, 3, size)
eps = np.random.uniform(−0.5, 0.5, size) #noise
Y = true_params[0] + true_params[1]∗x + eps

fig, ax = plt.subplots()
ax.scatter(x, Y, c='g', marker='.')
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.savefig('1_a.png', format='png')

#.....Solution 1b.....#
m = (np.sum(x∗Y) − np.sum(x)∗np.sum(Y) / size) / (np.sum(x∗x) −
      ((np.sum(x))∗∗2) / size)
b = np.mean(Y) − m∗np.mean(x)

print('Estimated intercept is %.4f and slope is %.4f' %(b,m))

#.....Solution 1c.....#
bias = np.ones(size)
X = np.vstack((bias, x)).T #design matrix

params = np.linalg.pinv((np.dot(X.T,X)))
params = np.dot(params,X.T)
params = np.dot(params,Y)

print('Estimated intercept is %.4f and slope is %.4f' %tuple(params))

#.....Solution 1d.....#
#Plot data with regression line
fig, ax = plt.subplots()

ax.scatter(x, Y, c='g', marker='.')
ax.set_xlabel('x')
ax.set_ylabel('y')

y_pred = params[0] + params[1] ∗ x

ax.scatter(x, Y, c='g', marker='.', label='data')
ax.plot(x, y_pred, c='r', label='Linear regressor')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('')
ax.legend()
fig.savefig('1_d.png', format='png')
```

```
residues = y_pred − Y

y_mean = np.mean(Y)
ss_data = np.sum((Y − y_mean)**2)
ss_reg= np.sum((y_pred − y_mean)**2)
exp_variance = ss_reg / ss_data
print('Explained variance is = %f' %exp_variance)
#Generate residual plot
fig, ax = plt.subplots()

ax.scatter(x,residues)
ax.plot(x,np.zeros(len(x)), linestyle='−.')
ax.plot(x, 0.5*np.ones(len(x)),c="r", linestyle=':')
ax.plot(x,−0.5*np.ones(len(x)),c="r", linestyle=':')
ax.set_xlabel('x')
ax.set_ylabel('residual')
fig.savefig('1_d2.png', format='png')

#.....Solution 1e.....#

from sklearn import linear_model
from sklearn.metrics import r2_score

lin_reg = linear_model.LinearRegression(fit_intercept=True)
lin_reg.fit(x.reshape(−1,1), Y)
slope = lin_reg.coef_
intercept = lin_reg.intercept_
y_pred_sk = lin_reg.predict(x.reshape(−1,1))
r2_sk = r2_score(Y,y_pred_sk)
print('Estimated intercept is %.4f and slope is %.4f' %(intercept, slope))
print('R2−score (using sklearn) = %f' %r2_sk)
```

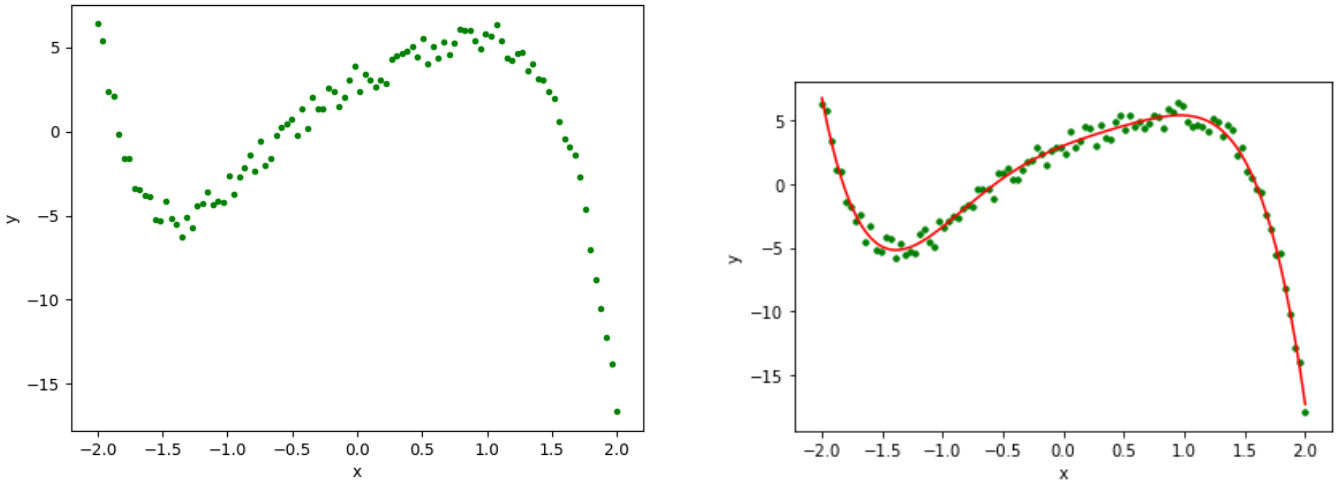
2. **Polynomial regression**

   (a) Scatter plot of y against x (Fig. 70a)

   (b) Applying multiple linear regression to the data generated using the polynomial function (Fig. 70b).

   (c) Increasing the number of data points generally reduces the l_2-norm of the difference between the true and estimated parameters (Fig. 71).

```
import numpy as np
import matplotlib.pyplot as plt

#.....Solution 2a.....#
def generate_data(size) :
    x = np.linspace(−2,2,size)
    eps = np.random.uniform(−1,1,size)
    true_params = [3,4,−2,1.5,0,−1]
```



(a) (b)

Figure 70: (a) Scatter plot of the data (b) Data and the linear regressor

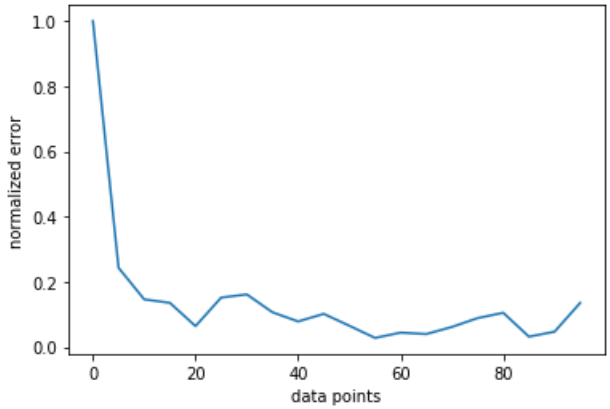


Figure 71: The role of the number of data point on the normalized error.

```
    Y = (true_params[0] + true_params[1]*x + true_params[2]*(x**2) +
         true_params[3]*(x**3) + true_params[4]*x**4 + true_params[5]*x**5) + eps
    bias = np.ones(size)
    X = np.vstack((bias,x,x**2,x**3,x**4,x**5)).T #design matrix

    return x,X,Y

size = 100
x, X, Y = generate_data(size)
fig, ax = plt.subplots()
ax.scatter(x, Y, c='g', marker='.')
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.savefig('2_a.png', format='png')

#.....Solution 2b.....#
params = np.linalg.pinv((np.dot(X.T,X)))
params = np.dot(params,X.T)
params = np.dot(params,Y)
```

```
fig, ax = plt.subplots()
ax.scatter(x, Y, c='g', marker='.')

y_pred = np.dot(params, X.T)
#OR y_pred = (params[0] + params[1]*x + params[2]*(x**2) +
# params[3]*(x**3) + params[4]*x**4 + params[5]*x**5)
ax.scatter(x, Y, c='g', marker='.')
ax.plot(x, y_pred, c='r', linewidth=2)
ax.set_xlabel('x')
ax.set_ylabel('y')
fig.savefig('2_b.png', format='png')

#.....Solution 2c.....#
error = []
N = np.arange(0,100,5)
true_params = [3,4,-2,1.5,0,-1]
for n in N :
    x,X,Y = generate_data(n)
    params = np.linalg.pinv((np.dot(X.T,X)))
    params = np.dot(params,X.T)
    params = np.dot(params,Y)
    err = np.linalg.norm(true_params - params, 2)/np.linalg.norm(true_params,2)
    error.append(err)

fig, ax = plt.subplots()
ax.plot(N,error,linewidth=2)
ax.set_xlabel('data_points')
ax.set_ylabel('normalized_error')
fig.savefig('2_c.png', format='png')
```

3. **Incremental form of regression**

(a) For the implementation look at the `sgd_update` function in the python code below.

(b) Test the above function on data from Problem 1.

(c) Plotting the data points and the linear regression of the data (Fig. 72). Increasing the number of data point decreases the error (Fig. 73a). In general, smaller learning rates lead to a lower error when compared to larger learning rates. However, when the learning rate is too small and the number of iterations is limited ($N = 100$), the gradient descent does not reach the minimum and the error can be large (Fig. 73b). Thus, the choice of learning rate has to be made after careful consideration.

```
import numpy as np
import matplotlib.pyplot as plt

#.....Solution 3a.....#
def sgd_update(X,Y,lr=0.001,runs=1):
    ''' X : X matrix like in examples 1 and 2
        Y : observed values of dependent variable
        lr : learning rate
```

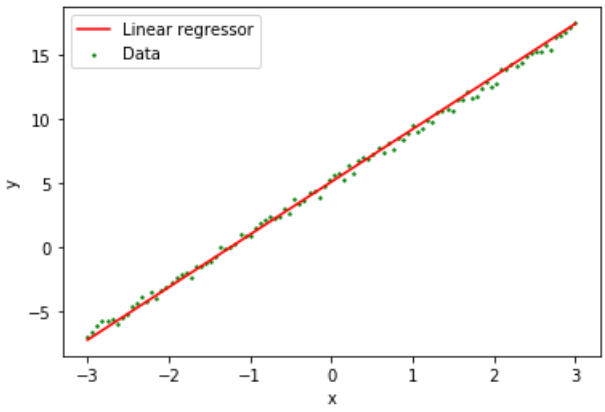
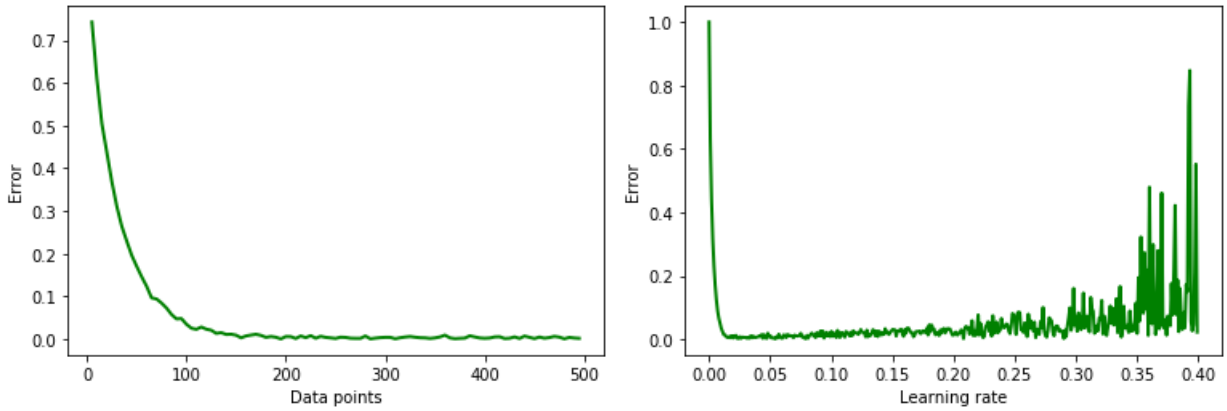


Figure 72: (a) Data and the linear regressor.



(a) (b)

Figure 73: The impact of the number of data points (a) and the learning rate (b; $N = 100$) on the error.

```
        runs : number of iterations of stochastic gradient descent
    '''
    theta = np.zeros(X.shape[1])
    for j in range(runs):
# np.random.seed(30)
        p = np.random.permutation(X.shape[0])
        X = X[p]
        Y = Y[p]
        for i in range(X.shape[0]) :
            theta = theta − lr∗(np.dot(X[i], theta)−Y[i])∗X[i] #eq. 22
    return theta

#.............Solution 3b.....................................................#
size = 100
true_params = [5,4]
x = np.linspace(−3, 3, size)
eps = np.random.uniform(−0.5, 0.5, size) #noise
Y = true_params[0] + true_params[1]∗x + eps
bias = np.ones(100)
X = np.vstack((bias, x)).T

theta = sgd_update(X, Y, 0.1, 5)
```

```
#.....Solution 3c.....#
y_pred = theta[0] + theta[1] * x
fig, ax = plt.subplots()
ax.scatter(x, Y, s=8, c='g', marker='.',label='Data')
ax.plot(x, y_pred, c='r',label='Linear regressor')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
# effect of data points
error_n = []
N = np.arange(5,500,5)
for n in N :
    x = np.linspace(-3, 3, n)
    eps = np.random.uniform(-0.5, 0.5, n) #noise
    Y = true_params[0] + true_params[1]*x + eps
    bias = np.ones(n)
    X = np.vstack((bias, x)).T
    params = sgd_update(X,Y,0.01,3)

    err = np.linalg.norm(true_params - params, 2)/np.linalg.norm(true_params,2)
    error_n.append(err)

fig, ax = plt.subplots()
ax.plot(N, error_n, linewidth=2, c='g')
ax.set_xlabel('Data_points')
ax.set_ylabel('Error')
# effect of learning rate
error_lr = []
lr = np.arange(0,0.4,0.001)

x = np.linspace(-3, 3, 100)
eps = np.random.uniform(-0.5, 0.5, 100) #noise
Y = true_params[0] + true_params[1]*x + eps
bias = np.ones(100)
X = np.vstack((bias, x)).T

for l in lr :
    params = sgd_update(X,Y,l,3)
    err = np.linalg.norm(true_params - params, 2)/np.linalg.norm(true_params,2)
    error_lr.append(err)

fig, ax = plt.subplots()
ax.plot(lr, error_lr, linewidth=2, c='g')
ax.set_xlabel('Learning_rate')
ax.set_ylabel('Error')
```

Solutions to set 4: Model Selection/ Regularization

Core steps needed for all exercises:

```
2 import numpy as np
3 import matplotlib.pyplot as plt
```

```
# Load the data and separate the predictors and targets
data=np.load('data.npy')
X,Y=data[:,0], data[:,1]

# Generate polynomial features of a given degree
def poly_features(x,degree):
    from sklearn.preprocessing import PolynomialFeatures as poly
    polynomial=poly(degree)
    x_poly=polynomial.fit_transform(x.reshape(len(x),1))
    return x_poly

# Calculate the mean squared error
def mean_squared_error(y, y_predicted):
    return np.mean((y-y_predicted)**2)

# Plot the fit
def plot_fit_line(ax,x,model,degree):
    x1=np.linspace(min(x),max(x),100)
    x_poly= poly_features(x1,degree)
    ax.plot(x1,model.predict(x_poly))

# Make aspect ratio equal
def set_aspect(ax):

    x0,x1 = ax.get_xlim()
    y0,y1 = ax.get_ylim()
    ax.set_aspect(abs(x1-x0)/abs(y1-y0))

# Initialize the class for linear regression

from sklearn.linear_model import LinearRegression
model=LinearRegression()
```

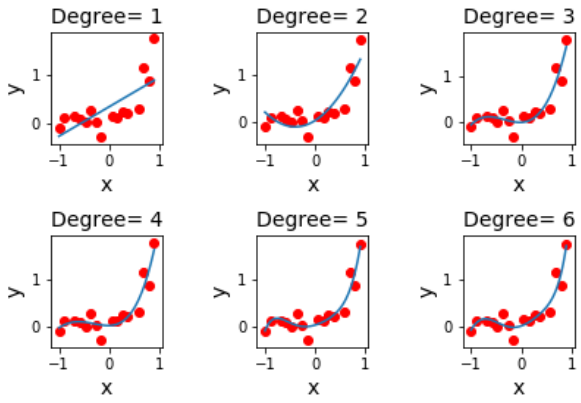
1. **The holdout method**

   The holdout method is the simplest kind of cross validation. The data is split into training and validation dataset. The model is fit to the training data, and evaluated on the validation data. The complete code for (a)-(e) is given below.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X,Y, test_size=0.2)
degrees=[1,2,3,4,5,6]
train_error=[]
test_error=[]
fig,axes=plt.subplots(2,3)
axes=axes.flatten()

for ind, degree in enumerate(degrees):
    #Get the polynomial features
    train_poly = poly_features(x_train,degree)
    test_poly = poly_features(x_test,degree)

    #Fit the data and assess the error
```

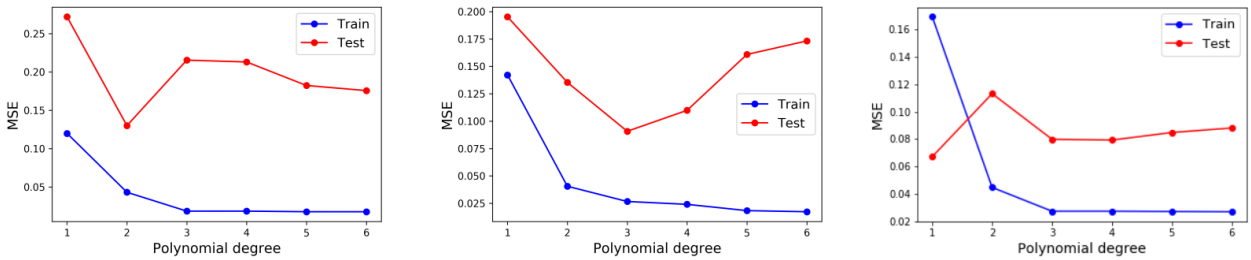
```
15        model. fit ( train_poly , y_train )
16         train_error  .append(mean_squared_error( y_train ,model. predict ( train_poly  )))
17         test_error  .append(mean_squared_error( y_test ,model. predict ( test_poly  )))

19        # Visualize  the  data  points  and  fitting   lines
20        ax=axes[ind]
21          plot_fit_line  (ax, x_train ,model,degree)
22        ax.  set_title  ('Degree=␣%s' %degree,fontsize=14)
23        ax. scatter ( x_train , y_train , color='r')
24        ax. set_xlabel ('x', fontsize =14)
25        ax. set_ylabel ('y', fontsize =14)
26         set_aspect (ax)

28   fig . tight_layout ()

30   # Plot  the  training  and  validation  errors
31   fig1 ,ax1=plt . subplots (1,1)
32   ax1. plot (degrees ,  train_error  ,'bo-',label='Train')
33   ax1. plot (degrees ,  test_error  ,'ro-',label='Test')
34   ax1.  set_xlabel ('Polynomial_degree', fontsize =14)
35   ax1.  set_ylabel ('MSE',fontsize=14)
36   ax1.legend(prop={'size':13})
```

(d) Fits with polynomials of different degrees



A simple linear model underfits the data and increasing the polynomial degree reduces the bias.

(e) MSE as a function of polynomial degree for three different random states in the function *train_test_split* (left to right).



In all cases, the bias (training error) is reduced with higher polynomial degrees, because the optimization minimizes the MSE and polynomials of higher degree are more flexible to fit the data. The model complexity that yields the lowest variance (validation error) should be chosen, in the example on the left $N = 2$ is the optimal choice.

A different random state in the function *train_test_split* generates a different split (left to right). Since the

variance fluctuates widely in this dataset, the minimum variance varies across different runs ($N = 3$ in the middle, $N = 1$ on the right). This issue arises because the model performance depends on how the data is split between training and validation sets. Especially if the dataset is small, the exact configuration of the limited training points determines the outcome. The holdout method is thefore a viable option only in case of large datasets and limited computational resources.

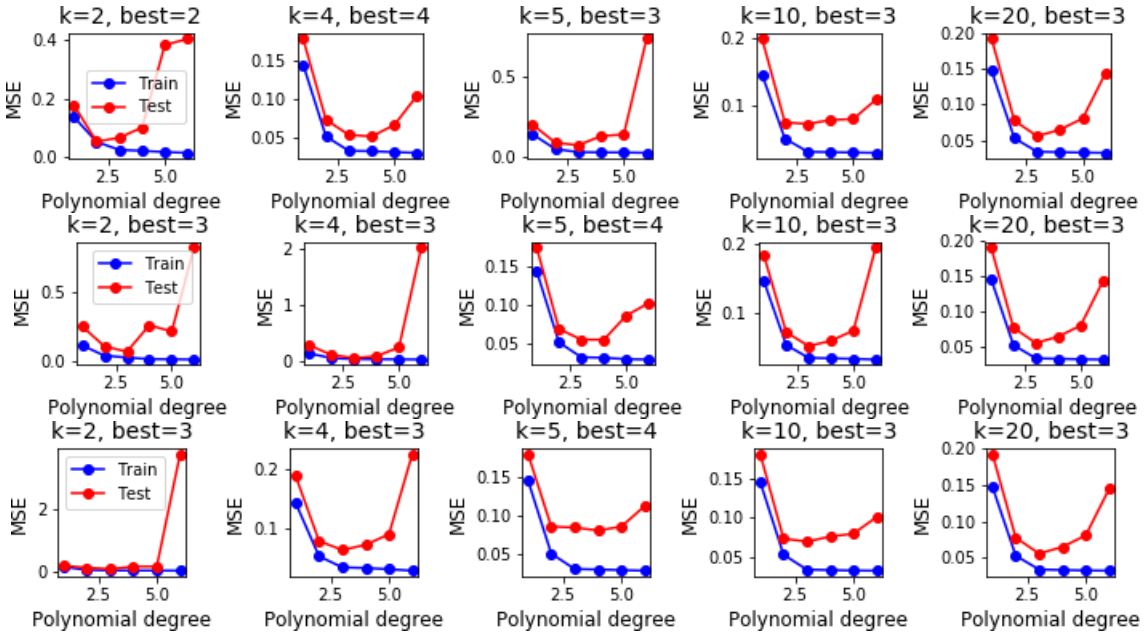
2. **k-fold cross-validation**

Code for (a) and (b):

```
from sklearn.model_selection import KFold
k_folds =[2,4,5,10,20]
degrees =[1,2,3,4,5,6]
fig,axes=plt.subplots(1,len(k_folds), figsize =(10,10))
for ind,k_fold in enumerate(k_folds):
    #get an instance of KFold class
    kf = KFold(n_splits =k_fold, shuffle =True)
    kf.get_n_splits(X)

    #allocated to store the error
     train_fold_err =[[] for j in range(k_fold)]
     test_fold_err =[[] for j in range(k_fold)]
    count=0


    for train_index, test_index in kf.split(X):
        #get the training and test data for consecutive folds
         x_train, x_test = X[train_index], X[test_index]
         y_train, y_test = Y[train_index], Y[test_index]

        #perform regression for different polynomial degrees

        for degree in degrees:
             train_poly = poly_features(x_train,degree)
             test_poly = poly_features(x_test,degree)
            model.fit(train_poly, y_train)
             train_fold_err[count].append(\
            mean_squared_error(y_train,model.predict(train_poly)))
             test_fold_err[count].append(\
            mean_squared_error(y_test,model.predict(test_poly)))
        count+=1

    #average error across folds and plot
     train_error =np.mean(train_fold_err,axis=0)
     test_error =np.mean(test_fold_err,axis=0)
    ax=axes[ind]
    ax.plot(degrees, train_error,'bo-',label='Train')
    ax.plot(degrees, test_error,'ro-',label='Test')
    ax.set_xlabel('Polynomial_degree', fontsize =12)
    ax.set_ylabel('MSE',fontsize=12)
    ax.set_title('Best=%s'%(np.argmin(test_error)+1), fontsize =14)
     set_aspect(ax)
    ax.legend(prop={'size':10})
```

A big advantage of cross-validation method is that no data is wasted, and all data points are used for both training and testing. A rule of thumb for cross-validation is to use 5 or 10 folds. However, you should always keep in mind that the stability of the model decreases if the dataset is small.

Examples of the training and validation errors vs model complexity for different folds. Each row represents a run with different training/validation-split.



Most of the time, the 3rd-degree polynomial is selected; however, the training and test errors exhibit the most stable pattern when $k = n = 20$.

With very small datasets, it is recommended to use $k = n$ folds, where n refers to the size of the dataset. It is known as *leave-one-out cross-validation*. This method allows to effectively use the data for training since in each step only one data point is taken for testing. However, for large datasets, this method may be computationally expensive, since a large number of iterations will be needed.

3. **Regularization**

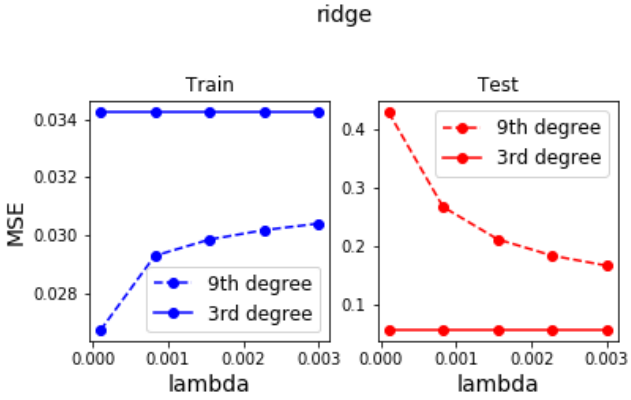
Code for (a)-(d):

```
# get instances of regularizing models
 regularization ='ridge' #'lasso' or 'ridge'
if regularization =='ridge':
    from sklearn.linear_model import Ridge
    reg_model=Ridge()
elif regularization =='lasso':
    from sklearn.linear_model import Lasso
    reg_model=Lasso()


lambdas=np.linspace(0.0001,.003,5)
 overfit_degree =9
best_degree =3

# get an instance of KFold class
k=20
kf = KFold( n_splits =k, shuffle =True)

# allocate for storing
 train_error =[]
```

```
test_error =[]
train_error_reg =[[] for i in range(len(lambdas))]
test_error_reg =[[] for i in range(len(lambdas))]
coefs=[[] for i in range(len(lambdas))]

for train_index, test_index in kf.split(X):
    # get the training and test data for consecutive folds
    x_train, x_test = X[train_index], X[test_index]
    y_train, y_test = Y[train_index], Y[test_index]
    # fit the best-fit polynomial without regularization
    model.fit(poly_features(x_train, best_degree), y_train)
    train_error.append(mean_squared_error(y_train, model.predict \
    (poly_features(x_train, best_degree))))
    test_error.append(mean_squared_error(y_test, model.predict \
    (poly_features(x_test, best_degree))))

    #use regularization with different strengths for overfitting model
    for ind, Lambda in enumerate(lambdas):
        reg_model.alpha=Lambda
        reg_model.fit(poly_features(x_train, overfit_degree), y_train)
        train_error_reg[ind].append(mean_squared_error(y_train, \
                    reg_model.predict(poly_features \
                    (x_train, overfit_degree))))
        test_error_reg[ind].append(mean_squared_error(y_test,\
                    reg_model.predict(poly_features \
                    (x_test, overfit_degree))))
        coefs[ind].append(reg_model.coef_)

# average across folds
train_error=np.mean(train_error)
test_error=np.mean(test_error)
train_error_reg=np.mean(train_error_reg,axis=1)
test_error_reg=np.mean(test_error_reg,axis=1)
coefs=np.mean(coefs,axis=1)
#
# plot the errors vs lambda
fig,axes=plt.subplots(1,2)
axes[0].plot(lambdas, train_error_reg,'bo--',label='9th_degree')
axes[1].plot(lambdas, test_error_reg,'ro--',label='9th_degree')
axes[0].plot(lambdas,[train_error]*len(lambdas),'bo-',label='3rd_degree')
axes[1].plot(lambdas,[test_error]*len(lambdas),'ro-',label='3rd_degree')
axes[0].set_xlabel('lambda', fontsize=14)
axes[1].set_xlabel('lambda', fontsize=14)
axes[0].set_ylabel('MSE',fontsize=14)
axes[0].set_xlabel('lambda', fontsize=14)
axes[0].legend(prop={'size':12})
axes[1].legend(prop={'size':12})
axes[0].set_title('Train')
axes[1].set_title('Test')
set_aspect(axes[0])
set_aspect(axes[1])
fig.suptitle(regularization, fontsize=14)
```

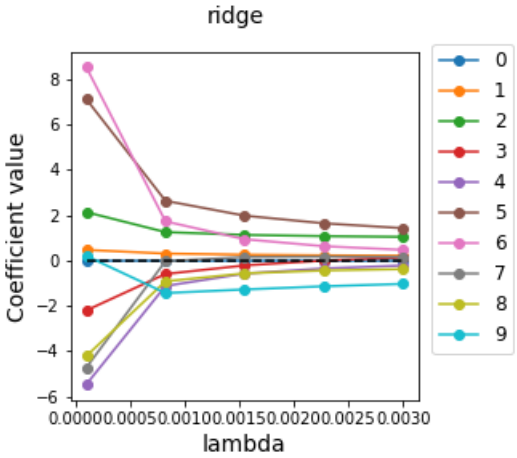
```
# plot coefficients vs lambda
fig1 ,ax1=plt. subplots (1,1)
for ind,coef in enumerate(coefs.T):
    ax1. plot (lambdas,coef,'o-',label=ind)
ax1. plot (lambdas,[0]*len(lambdas),'k--')
 set_aspect (ax1)
ax1. set_xlabel ('lambda', fontsize =14)
ax1. set_ylabel (' Coefficient value', fontsize =14)
ax1.legend(prop={'size':12}, loc='upper right', bbox_to_anchor =(1.3, 1.05))
fig1 . suptitle ( regularization , fontsize =14)
```

(b) MSE for ridge regression using polynomial of degree 9 with different regularization parameters $\lambda \in [0, 0.003]$:

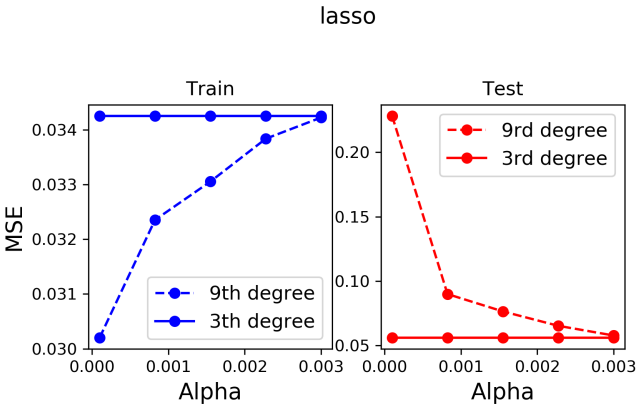


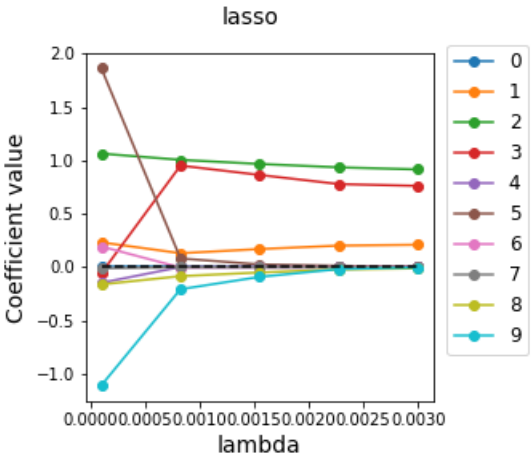
When regularization is applied, the training error, i.e. bias, of the model increases with λ, while the validation error, i.e. variance, decreases. Both errors approach the values of the best-fit, showing that overfitting can be avoided using regularization.

(c) In ridge regression, the coefficients approach zero but don't become zero.



(d) MSE for lasso regression and estimated coefficients:





Unlike ride regression, lasso regression performs feature selection by setting irrelevant coefficients to zero, leaving 3 non-zero coefficients, which was expected given that the data is best represented by a 3rd-degree polynomial (problem 2). Since the contribution of higher coefficients is eliminated, lasso regression approaches the best fit in the current dataset and therefore performs much better than ridge regression.

The function used to generate the data is $f(x) = x^3 + x^2 + \varepsilon$, where ε is drawn from a normal distribution with a mean of 0 and standard deviation of 0.4. So, the estimated parameters are quite accurate.

Solutions to set 5: Logistic Regression/ Classification

1. Derivation of the gradient.

$$L(\theta, X, Y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{p}_i) + (1-y_i) log(1-\hat{p}_i)]$$

$$\begin{aligned}\frac{\partial L}{\partial \theta_j} &= -\frac{1}{N}\sum_{i=1}^{N}\frac{\partial}{\partial \theta_j}[y_i log(\hat{p}_i) + (1-y_i) log(1-\hat{p}_i)] \\ &= \frac{1}{N}\sum_{i=1}^{N} x_i(y_i - \sigma(\theta^T x_i))\end{aligned}$$

As per the lecture notes:

$$\frac{d}{dt}\sigma(t) = \sigma(t)(1-\sigma(t))$$

Let $t = \theta^T x_i$, then

$$\begin{aligned}\frac{\partial}{\partial \theta}\sigma(t) &= \frac{\partial}{\partial t}\sigma(t).\frac{\partial t}{\partial \theta} \\ &= \sigma(t)(1-\sigma(t)).x_i \\ &= x_i\sigma(\theta^T x_i)(1-\sigma(\theta^T x_i))\end{aligned}$$

Let,

$$L(\theta, X, Y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\hat{p}_i) + (1-y_i) log(1-\hat{p}_i)],$$

where $\hat{p}_i = \sigma(\theta^T x_i)$.

Then,

$$L(\theta, X, Y) = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\sigma(\theta^T x_i)) + (1-y_i) log(1-\sigma(\theta^T x_i))]$$

And thus,

$$\begin{aligned}
\frac{\partial L}{\partial \theta} &= -\frac{1}{N}\sum_{i=1}^{N}[y_i \frac{\partial}{\partial \theta}[log(\sigma(\theta^T x_i))] + (1-y_i)\frac{\partial}{\partial \theta}[log(1-\sigma(\theta^T x_i))]] \\
&= -\frac{1}{N}\sum_{i=1}^{N}[y_i [\frac{1}{\sigma(\theta^T x_i)}\frac{\partial}{\partial \theta}(\sigma(\theta^T x_i))] + (1-y_i)[\frac{1}{1-\sigma(\theta^T x_i)}\frac{\partial}{\partial \theta}(1-\sigma(\theta^T x_i))]] \\
&= -\frac{1}{N}\sum_{i=1}^{N}[y_i [\frac{1}{\sigma(\theta^T x_i)} x_i \sigma(\theta^T x_i)(1-\sigma(\theta^T x_i))] + (1-y_i)[\frac{1}{1-\sigma(\theta^T x_i)}(-x_i \sigma(\theta^T x_i)(1-\sigma(\theta^T x_i))]] \\
&= -\frac{1}{N}\sum_{i=1}^{N}[y_i x_i (1-\sigma(\theta^T x_i) + (1-y_i)(-x_i \sigma(\theta^T x_i))] \\
&= -\frac{1}{N}\sum_{i=1}^{N}[y_i x_i - y_i x_i \sigma(\theta^T x_i) - x_i \sigma(\theta^T x_i) + x_i y_i \sigma(\theta^T x_i)] \\
&= -\frac{1}{N}\sum_{i=1}^{N} x_i (y_i - \sigma(\theta^T x_i))
\end{aligned}$$

2. Code for 2.a – 2.e:

```
# Load packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

# equal aspect ratio for plots
def set_aspect(ax):
    x0,x1 = ax.get_xlim()
    y0,y1 = ax.get_ylim()
    ax.set_aspect(abs(x1-x0)/abs(y1-y0))

# scatter plot of data points
def plot_data(data):
    fig,ax=plt.subplots(1,1, figsize=(8,8))
    approved=train[train[:,-1]==0]
    denied=train[train[:,-1]==1]
    ax.plot(approved[:,0], approved[:,1], 'go',alpha=0.5)
    ax.plot(denied[:,0], denied[:,1], 'ro',alpha=0.5)
    ax.set_xlabel('Residence duration', fontsize=16)
    ax.set_ylabel('Yearly income', fontsize=16)
    return fig,ax

class LogRegression:
    def __init__(self):
```

```
25            self.loss=[]

27        def prepare_data(self,x):
28            # add a column for the bias term
29            x=np.hstack((np.ones((x.shape[0], 1)), x))
30            return x

32        def loss_function(self,y,p_hat):
33            return -np.mean(y*np.log(p_hat)+(1-y)*np.log(1-p_hat))

35        def prediction(self,x):
36            try:
37                value=np.dot(x, self.theta)
38            except:
39                # relevant when predictions are done for test set
40                x=self.prepare_data(x)
41                value=np.dot(x, self.theta)
42            return 1/(1+np.exp(-value))

44        def fit(self,num_epochs, learn_rate,X,Y):
45                X=self.prepare_data(X)
46                self.theta=np.zeros(X.shape[1]) # initialize the weights
47                for iteration in range(num_epochs):
48                    p_hat=self.prediction(X)
49                    gradient = np.dot(X.T, (p_hat-Y))/Y.size
50                    self.theta -= learn_rate * gradient # update the weights
51                    self.loss.append(self.loss_function(Y,p_hat))

53        def plot_decision_boundary(self,ax,X):
54            y_intercept =-self.theta[0]/self.theta[2]
55            slope=-self.theta[1]/self.theta[2]
56            x=np.arange(min(X[:,0]), max(X[:,0]))
57            y=slope*x+y_intercept
58            ax.plot(x,y,'k-')

60        def plot_loss(self,num_epochs):
61            fig,ax=plt.subplots(1,1)
62            ax.plot(range(num_epochs),self.loss)
63            ax.set_xlabel('Epochs', fontsize=14)
64            ax.set_ylabel('Loss', fontsize=14)
65    #          fig.savefig('loss.png', dpi=100)

67    class Classification:
68        def __init__(self):
69            self.N_thr=8
70            self.thresholds=np.linspace(0.0,1, self.N_thr)[::-1]
71            self.precision=[]
72            self.recall=[]
73            self.F1=[]
74            self.FP_rate=[]

76        def classify(self, true_label, prediction):
77            for threshold in self.thresholds:
```

```
78                  response=[1 if item>threshold else 0 for item in prediction ]
79                  accuracy= true_label -response
80                  FN=len(np.where(accuracy==1)[0])
81                  FP=len(np.where(accuracy==-1)[0])
82                  TP=np.sum(true_label)-FN
83                  TN=len( true_label )-(FN+FP+TP)
84                  prec=TP/(TP+FP)
85                  rec=TP/(TP+FN)

87                  self . precision .append(prec)
88                  self . recall .append(rec)
89                  self .F1.append (2/((1/ prec )+(1/ rec )))
90                  self . FP_rate .append(FP/(FP+TN))


93          def plot_measures ( self ):
94              fig ,axes=plt . subplots (1,3, figsize =(8,8))
95              colors=plt .rcParams['axes . prop_cycle ' ]. by_key ()[ ' color ' ]
96              [axes [0]. plot ( self . recall [ i ], self . precision [ i ], 'o' ,
97              color=colors [ i ]) for i in range( self . N_thr )]
98              axes [0]. plot ( self . recall , self . precision , '-',alpha=.5)
99              axes [0]. set_xlabel ( ' Recall ' , fontsize =16)
100             axes [0]. set_ylabel ( ' Precision ' , fontsize =16)

102             [axes [1]. plot ( self . thresholds [ i ], self .F1[ i ], 'o' ,
103             color=colors [ i ]) for i in range( self . N_thr )]
104             axes [1]. plot ( self . thresholds , self .F1,'-',alpha=.5)
105             axes [1]. set_xlabel ('Threshold' , fontsize =16)
106             axes [1]. set_ylabel ('F1', fontsize =16)

108             [axes [2]. plot ( self . FP_rate [ i ], self . recall [ i ], 'o' ,
109             color=colors [ i ]) for i in range( self . N_thr )]
110             axes [2]. plot ( self . FP_rate , self . recall , '-',alpha=.5)
111             axes [2]. plot ([0,1],[0,1], 'k--')
112             axes [2]. set_xlabel ( ' False_alarms ' , fontsize =16)
113             axes [2]. set_ylabel ( ' Recall ' , fontsize =16)
114             axes [2]. set_xlim (-0.05,1)
115             axes [2]. set_ylim (0,1.05)
116             [ set_aspect (ax) for ax in axes]
117             fig . tight_layout ()
118 #           fig . savefig ('performance.png', dpi=100)

120 # Routine for executing the steps
121 data=np.load('05 _log_regression_data .npy')

123 # standardize the features
124 from scipy . stats import zscore
125 data [:,0]= zscore ( data [:,0])
126 data [:,1]= zscore ( data [:,1])

128 # split into train and test ( validation )
129 train , test = train_test_split (data)
```

```
131  train_data , train_label =train [:,:-1], train [:,-1]
132  test_data , test_label =test [:,:-1], test [:,-1]

134  # visualize the training data
135  fig ,ax= plot_data ( train_data )

137  reg=LogRegression() # get an instance of the regression model
138  num_epochs=15000
139  reg. fit (num_epochs,0.001, train_data , train_label )
140  reg. plot_decision_boundary (ax, train_data )
141  #fig . savefig (' classification .png',dpi=200)
142  reg. plot_loss (num_epochs)

144  # measure the performance of the model on the validation set
145   test_predictions =reg. prediction ( test_data )
146  decision= Classification ()
147  decision . classify ( test_label , test_predictions )
148  decision . plot_measures ()
```

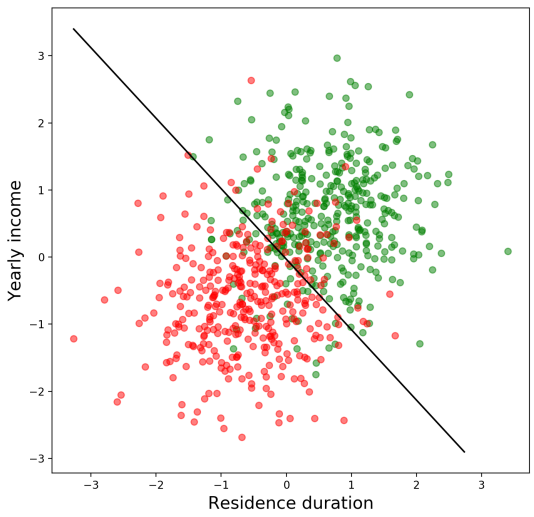


Figure 74

3. **Notes about plotting the decision line**
   The slope-intercept form of a line is

$$y = mx + b \tag{185}$$

If the decision threshold $p_0 = \frac{1}{2}$, the decision boundary is given by:

$$\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0 \tag{186}$$

If we compare the two equations, $\theta_2 x_2$ is equivalent to y, $\theta_1 x_1$ is equivalent to x and θ_0 is equivalent to b. Therefore, to express equation 2 in the slope-intercept form, we need to simply rearrange the terms:

$$\theta_2 x_2 = -\theta_1 x_1 - \theta_0 \tag{187}$$

$$x_2 = -\frac{\theta_1}{\theta_2} x_1 - \frac{\theta_0}{\theta_2} \tag{188}$$

The line is plotted in Fig.74 together with the data.

4. The loss function:

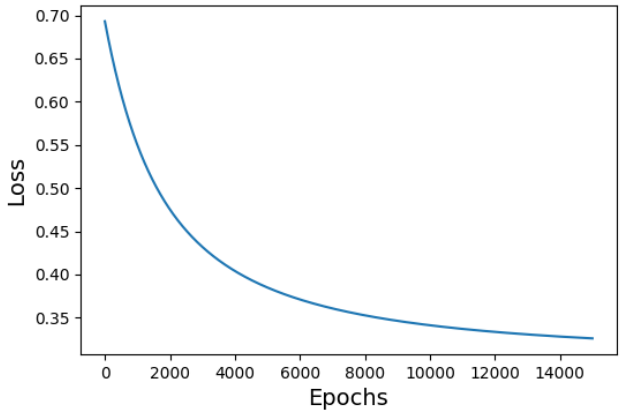


Figure 75

The loss function in logistic regression is a convex function, therefore a global maximum is reached and the loss saturates. However, since the provided data is not linearly separable (Fig.74), the loss never becomes zero (Fig.75).

5. The performance measures yield a reasonable performance for the validation set as well (see the ROC-curve in Fig.76). For this particular test-train split the maximum value of F1 score is reached for a threshold equal to 0.43. Note that this is also the region where the precision-recall curve starts dropping more abruptly and the false alarm rate increases on the ROC-curve (see the purple dots).

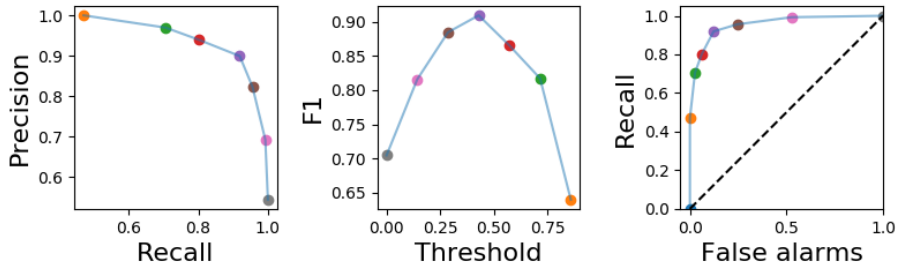


Figure 76

Solutions to set 7: Artifical Neural Networks

1. The definition of $\phi(\cdot) = \sigma(x)$ can be done as follows:

```
from scipy.special import expit as sigmoid
```

With that, 'sigmoid(x)' represents the function $\sigma(x)$ that can be used as activation function $\phi(\cdot)$.
Note: do not forget to import other necessary libraries:

```
import numpy as np
import matplotlib.pyplot as graph
```

In your program define $f(x)$, taking into account the necessary adaptation of w_i and b_i:

```
def f(x,w,b,v,N):
    output=0.0
    for i in range(N):
        output+=v[i]*sigmoid(w[i]*(x+b[i]))
    return output
```

2. In your program, define the x-domain:

```
x=np.linspace(0.0,1.0,1e4)
```

Generate output shown in Fig. 27a:

```
# number of units
N=2
# vector of weights (hidden)
w=np.array([1e4,1e4])
# vector of biases (hidden)
b=np.array([-0.4,-0.6])
# vector of weights (output)
v=np.array([0.8,-0.8])

# plot the function
graph.plot(x,f(x,w,b,v,N))
# display the plot
graph.show()
```

Generate output shown in Fig. 27b:

```
N=4
w=np.array([1e4,1e4,1e4,1e4])
b=np.array([-0.2,-0.4,-0.6,-0.8])
v=np.array([0.5,-0.5,0.2,-0.2])

graph.plot(x,f(x,w,b,v,N))
graph.show()
```

Generate the output shown in Fig. 27c:

```
N=6
w=np.array([1e4,1e4,1e4,1e4,1e4,1e4])
b=np.array([-0.2,-0.4,-0.4,-0.6,-0.6,-0.8])
v=np.array([0.5,-0.5,0.8,-0.8,0.2,-0.2])
```

```
graph.plot(x,f(x,w,b,v,N))
graph.show()
```

3. In your program, define $g(x) = \sin(2\pi x)$:

```
def g(x):
    return np.sin(x*2.0*np.pi)
```

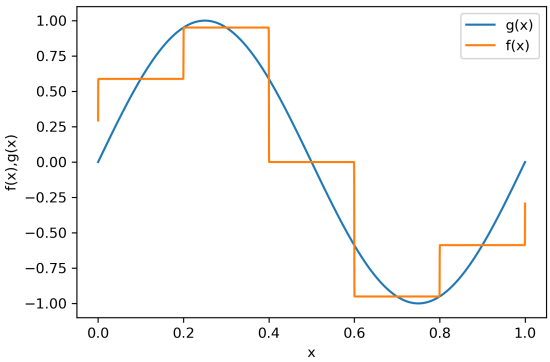
(a) Approximate $g(x)$ with $f(x)$, using $N = 10$, by choosing v_i, w_i, b_i automatically.
For that, first program the function:

```
def setParameters(N):
    w=np.ones((N),dtype=float)*1e4
    b=np.zeros((N),dtype=float)
    v=np.zeros((N),dtype=float)

    intervalDomain=np.linspace(0.0,1.0, N/2+1)

    for index in range(int(N/2)):
        leftBorder=intervalDomain[index]
        rightBorder=intervalDomain[index+1]
        b[index*2]=-leftBorder
        b[index*2+1]=-rightBorder
        mid=(rightBorder-leftBorder)/2.0+ leftBorder
        v[index*2]=g(mid)
        v[index*2+1]=-g(mid)

    return w,b,v
```

Then use the programmed function:

```
N=10
w,b,v=setParameters(N)

graph.plot(x,g(x))
graph.plot(x,f(x,w,b,v,N))
graph.show()
```

Output:



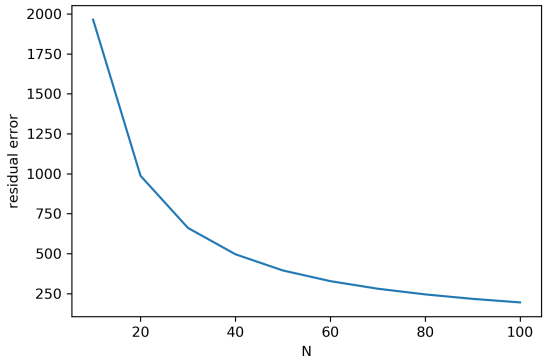
(b) Compute the residual error using elementary programming operations.

```
residualError =np.sum(np.abs(f(x,w,b,v,N)-g(x)))
```

Using $N = 10$, the residual error is ≈ 1963.96. To plot the residual error against N:

```
errors =np.zeros ((10,1), dtype=float)
N=np.linspace(10,100,10)
for nIndex in range(N.shape[0]):
    n=int(N[nIndex])
    w,b,v=setParameters(n)
    residualError =np.sum(np.abs(f(x,w,b,v,n)-g(x)))
    errors [nIndex]= residualError

graph.plot(N,errors)
graph.show()
```

Output:



Solutions to set 8: Perceptron

Core packages and the code for implementing the basic perceptron algorithm and the pocket algorithm:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from sklearn import datasets


6  class Perceptron:
7      def __init__(self):
8          self.weights = None
9          self.current_training_error = None
10         self.training_errors = None

12     def train(self, X, Y, learning_rate, max_epochs, max_error, shuffle=True):
13         self.weights = np.zeros(X.shape[1])  # Initialize weights
14         self.current_training_error = self.count_errors(X, Y)
15         self.training_errors = []

17         # Run the perceptron algorithm
18         for epoch_num in range(max_epochs):
19             if shuffle:
20                 shuffled_indices = np.random.permutation(len(X))
21                 X = X[shuffled_indices]
22                 Y = Y[shuffled_indices]
23             for x, y in zip(X, Y):
24                 self.training_step(X, Y, x, y, learning_rate)
25                 if self.current_training_error <= max_error:
26                     print(f"Solution found at epoch {epoch_num}")
27                     return

29     def training_step(self, X, Y, x, y, learning_rate):
```

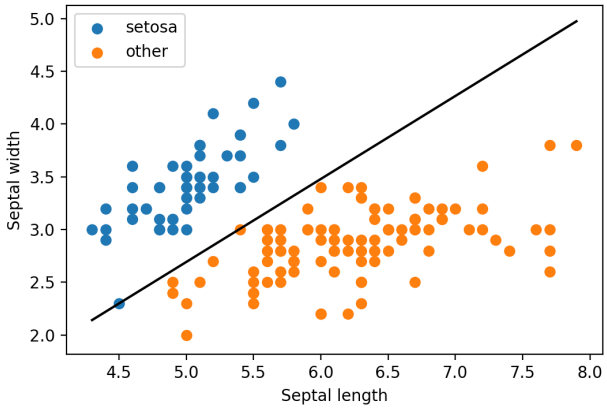
```
30            y_predicted = 1 if np.dot(self.weights, x) > 0 else -1
31            if y * y_predicted == -1:
32                self.weights += 2 * learning_rate * y * x
33                self.current_training_error = self.count_errors(X, Y)
34            self.training_errors.append(self.current_training_error)

36        def count_errors(self, X, Y):
37            Y_predicted = np.where(np.dot(X, self.weights) > 0, 1, -1)
38            return np.count_nonzero(Y*Y_predicted == -1)

40        def plot_training_errors(self):
41            fig, ax = plt.subplots()
42            ax.plot(self.training_errors)
43            ax.set_xlabel("Iteration")
44            ax.set_ylabel("Error")
45            return ax

47        def plot_2D_decision_boundary(self, ax, X):
48            x1 = np.linspace(X[:, 1].min(), X[:, 1].max(), 2)
49            ax.plot(x1, -self.trained_weights()[0] / self.trained_weights()[2]
50                    - self.trained_weights()[1] / self.trained_weights()[2] * x1, 'k-')

52        def trained_weights(self):
53            return self.weights


56    class PocketPerceptron(Perceptron):
57        def __init__(self):
58            super().__init__()
59            self.weights_in_pocket = None
60            self.smallest_training_error = np.inf
61            self.smallest_training_errors = []

63        def training_step(self, X, Y, x, y, learning_rate):
64            super().training_step(X, Y, x, y, learning_rate)
65            if self.training_errors[-1] < self.smallest_training_error:
66                self.weights_in_pocket = np.copy(self.weights)
67                self.smallest_training_error = self.training_errors[-1]
68            self.smallest_training_errors.append(self.smallest_training_error)

70        def trained_weights(self):
71            return self.weights_in_pocket

73        def plot_training_errors(self):
74            ax = super().plot_training_errors()
75            ax.plot(self.smallest_training_errors, 'r', label='Best solution')
76            ax.legend()
```

Classify Iris setosa vs the rest:

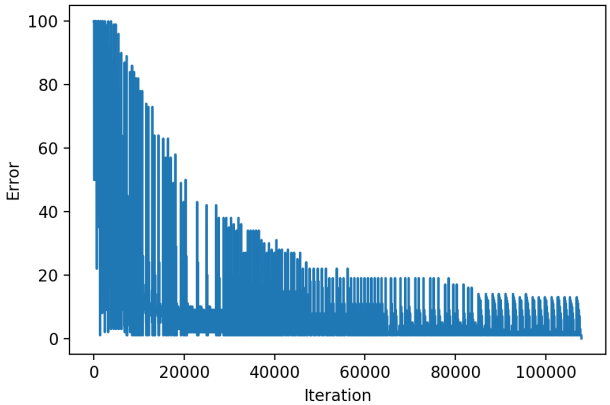
```
1     import os
2     from Perceptron import *


5     # Create a folder for saving the figures
6     if not os.path.exists("figures"):
7         os.makedirs("figures")

9     # Load the dataset
10    iris_dataset = datasets.load_iris()
```

```
X = iris_dataset.data[:, :2]  # we only take the first two features
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 0, 1, -1) # reassign labels

# Display the dataset
fig, ax = plt.subplots()
ax.scatter(X[Y == 1, 1], X[Y == 1, 2], label="setosa")
ax.scatter(X[Y == -1, 1], X[Y == -1, 2], label="other")
ax.set_xlabel("Septal length")
ax.set_ylabel("Septal width")
ax.legend()

# Train the basic perceptron
model = Perceptron()
model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
print(f"{model.count_errors(X,Y)} classification errors in training set")
model.plot_2D_decision_boundary(ax, X)
fig.savefig("figures/setosa-scatter.png", dpi=200)
model.plot_training_errors()
plt.savefig("figures/setosa-error.png", dpi=200)
plt.show()
```

1-2. The data is linearly separable, so the perceptron is guaranteed to converge. The scatter plot of septal width vs length, together with the resulting decision boundary should look like this:



3. If you terminate the classic perceptron before convergence, it will return the last solution. However, because the training error oscillates across iterations, you are not guaranteed to have the best solution encountered during training if you do so.

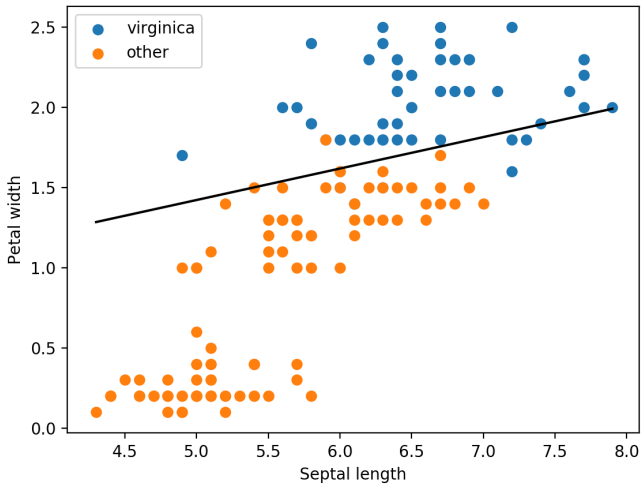


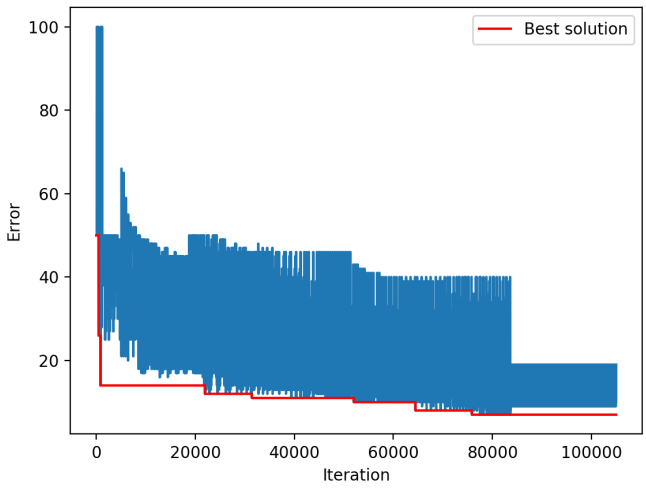
4. If weights are initialized at zero, the learning rate has no effect since only the direction of the weight vector counts.

Classify Iris Virginica vs the rest:

```
from Perceptron import *


# Load the dataset
iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, (0, 3)]
X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 2, 1, -1)

# Display the dataset
fig, ax = plt.subplots()
ax.scatter(X[Y == 1, 1], X[Y == 1, 2], label="virginica")
ax.scatter(X[Y == -1, 1], X[Y == -1, 2], label="other")
ax.set_xlabel("Septal length")
ax.set_ylabel("Petal width")
ax.legend()

# Train the pocket perceptron
model = PocketPerceptron()
model.train(X, Y, learning_rate=1, max_epochs=700, max_error=0, shuffle=False)
print(f"{model.smallest_training_errors[-1]} errors in the training set")
model.plot_2D_decision_boundary(ax, X)
fig.savefig("figures/virginica-scatter.png", dpi=200)
model.plot_training_errors()
plt.savefig("figures/virginica-error.png", dpi=200)
plt.show()
```

5-6. Iris virginica cannot be linearly separated from the other two types of Iris when using the septal length and petal width. However, since there are only a few outliers affecting linear separability, we would still like to be able to classify correctly most of the points.
If we used the classic perceptron, the algorithm would never converge, and if we terminated the algorithm after some maximum number of iterations, we could encounter the problem discussed in exercise 3.
The pocket algorithm can help us in this case by returning the best solution encounterd after a fixed number of iterations:



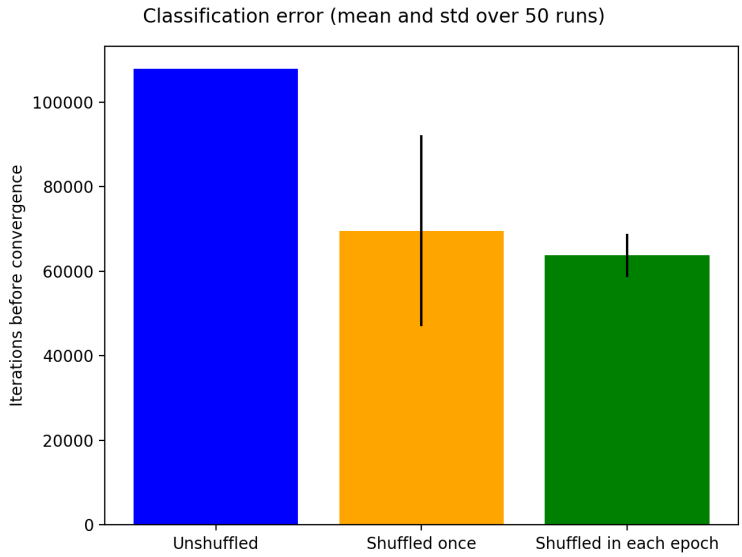


7. With this code we compare the convergence of the basic and the pocket perceptron algorithms without shuffling, shuffling once before training, or shuffling before each training epoch.

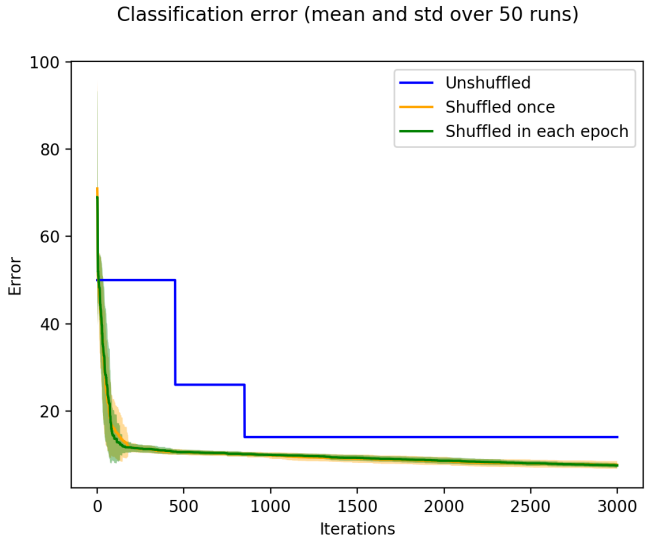
```
from Perceptron import *


# Study the effect of shuffling on the basic perceptron

iris_dataset = datasets.load_iris()
X = iris_dataset.data[:, :2] # we only take the first two features
X = np.hstack((np.ones((X.shape[0], 1)), X)) # Add x0 = 1 for bias term
Y = np.where(iris_dataset.target == 0, 1, -1) # reassign labels

model = Perceptron()
model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
conv_unshuffled = (len(model.training_errors))
conv_shuffled = []
conv_shuffled_once = []
N = 50

for i in range(N):
    print(f"Run number {i}")
    model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=True)
    conv_shuffled.append(len(model.training_errors))
    shuffled_indices = np.random.permutation(len(X))
    X = X[shuffled_indices]
    Y = Y[shuffled_indices]
    model.train(X, Y, learning_rate=1, max_epochs=1000, max_error=0, shuffle=False)
    conv_shuffled_once.append(len(model.training_errors))

fig, ax = plt.subplots(constrained_layout=True)
ax.bar(1, conv_unshuffled, color='b')
ax.bar(2, np.mean(conv_shuffled_once), yerr=np.std(conv_shuffled_once), color='orange')
ax.bar(3, np.mean(conv_shuffled), yerr=np.std(conv_shuffled), color='g')
ax.set_xticks(np.arange(1, 4))
ax.set_xticklabels(['Unshuffled', 'Shuffled once', 'Shuffled in each epoch'])
ax.set_ylabel('Iterations before convergence')
fig.suptitle(f"Classification error (mean and std over {N} runs)")
fig.savefig("figures/shuffled-basic.png", dpi=200)


# Study the effect of shuffling on pocket perceptron
```

```
41  iris_dataset = datasets.load_iris()
42  X = iris_dataset.data[:, (0, 3)]
43  X = np.hstack((np.ones((X.shape[0], 1)), X))  # Add x0 = 1 for bias term
44  Y = np.where(iris_dataset.target == 2, 1, -1)

46  N = 50
47  num_epochs = 20

49  model_unshuffled = PocketPerceptron()
50  model_unshuffled.train(X, Y, learning_rate=1, max_epochs=num_epochs,
51                         max_error=0, shuffle=False)
52  unshuffled = model_unshuffled.smallest_training_errors

54  shuffled = []
55  shuffled_once = []

57  for i in range(N):
58      print(f"Run number {i}")
59      model_shuffled = PocketPerceptron()
60      model_shuffled_once = PocketPerceptron()
61      model_shuffled.train(X, Y, learning_rate=1, max_epochs=num_epochs, max_error=0)
62      shuffled.append(model_shuffled.smallest_training_errors)
63      shuffled_indices = np.random.permutation(len(X))
64      X = X[shuffled_indices]
65      Y = Y[shuffled_indices]
66      model_shuffled_once.train(X, Y, learning_rate=1, max_epochs=num_epochs,
67                                max_error=0, shuffle=False)
68      shuffled_once.append(model_shuffled_once.smallest_training_errors)

70  shuffled_mean = np.mean(shuffled, axis=0)
71  shuffled_std = np.std(shuffled, axis=0)
72  shuffled_once_mean = np.mean(shuffled_once, axis=0)
73  shuffled_once_std = np.std(shuffled_once, axis=0)
74  fig, ax = plt.subplots()
75  ax.plot(unshuffled, color='b', label="Unshuffled")
76  ax.fill_between(np.arange(shuffled_once_mean.size),
77                  shuffled_once_mean - shuffled_once_std,
78                  shuffled_once_mean + shuffled_once_std,
79                  facecolor='orange', alpha=0.4)
80  ax.plot(shuffled_once_mean, color='orange', label="Shuffled once")
81  ax.fill_between(np.arange(shuffled_mean.size), shuffled_mean - shuffled_std,
82                  shuffled_mean + shuffled_std,
83                  facecolor='g', alpha=0.4)
84  ax.plot(shuffled_mean, color='g', label="Shuffled in each epoch")
85  ax.set_xlabel(" Iterations ")
86  ax.set_ylabel("Error")
87  ax.legend()
88  fig.suptitle(f" Classification error (mean and std over {N} runs)")
89  fig.savefig(" figures/shuffled-pocket.png", dpi=200)

91  plt.show()
```

For the basic perceptron algorithm, we plot the number of iterations before a solution is found for the unshuffled case, and the mean and standard deviation of 50 runs for the cases where the training set is shuffled once at the beginning, or before each training epoch. As you can see, shuffling the data leads to faster convergence.



For the pocket perceptron, we plot the evolution of the best classification error found over iterations. Again, we compare the unshuffled case with the mean and standard deviation of 50 runs for the two shuffling scenarios.



Solutions to set 9: Multi-Layer Perceptron

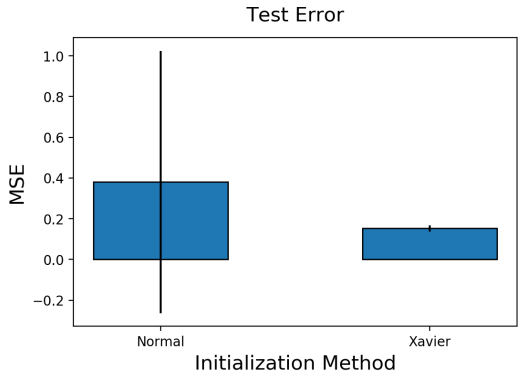
The full code solution for problems $1-3$ is available in the file *sol_mlp.py*.

1. **Backpropagation algorithm**

   The train function should implement the forward and backward pass using Algorithm 9.1 from the lecture notes. The prediction function computes the output using the forward pass and the mean squared error.

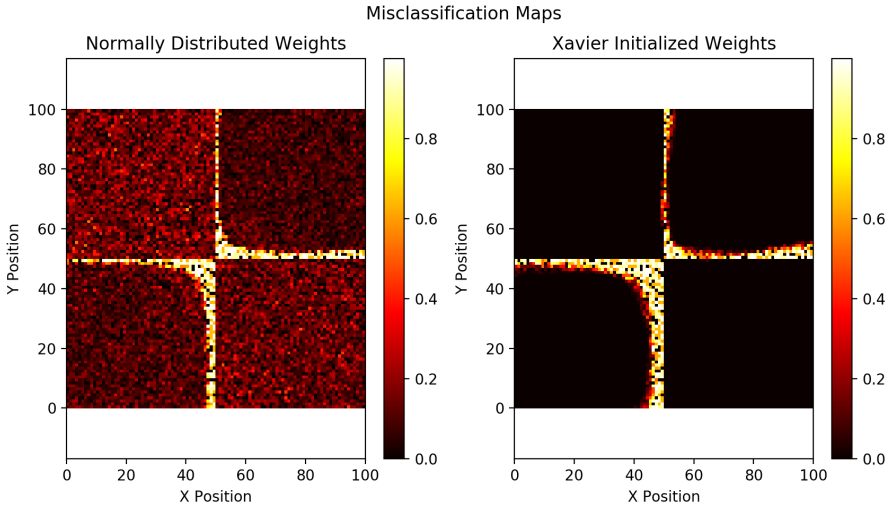
2. **Xavier initialization**

   Weights drawn from a normal distribution with unit variance lead to a large MSE with high variance. If Xavier initialization is used, the MSE is small with low variance.

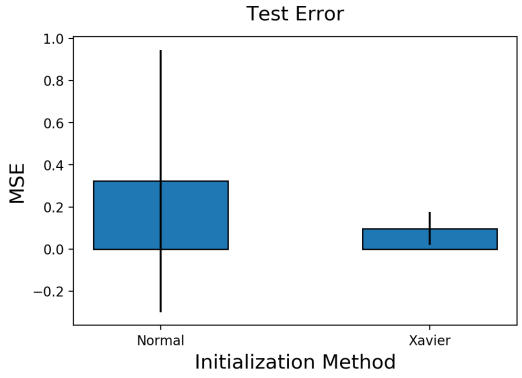


3. **Misclassifications**

(a) The distribution of misclassified points using different weight initialization schemes. For both initialization schemes, misclassifications mostly occur at the class boundaries. If weights are normally distributed, the MLP is not able to properly separate the data in some runs, which is reflected by off-boundary misclassifications covering the misclassification map (resembling background noise). When using Xavier initialization, the off-boundary misclassifications disappear almost completely, showing the superiority of Xavier initialization.

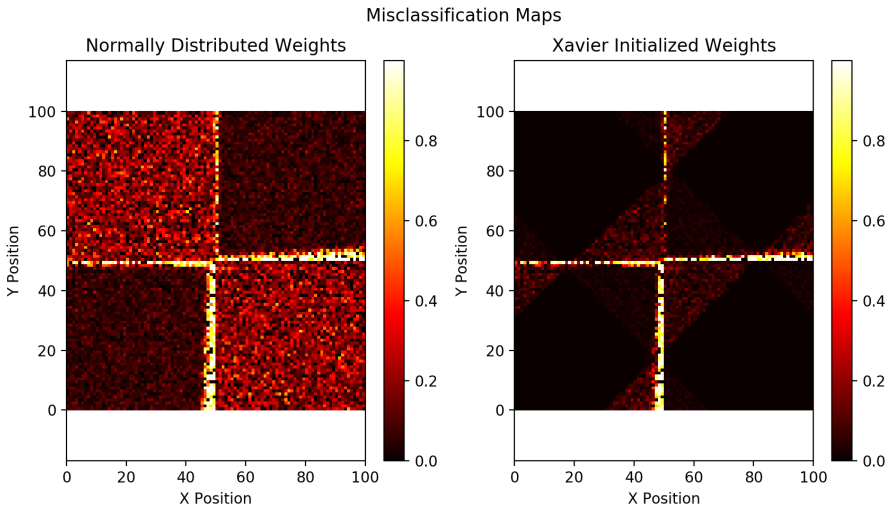


(b) Prediction errors using ReLU activations.



Compared to using the hyperbolic tangent as activation function, the MSE is similiar for weights initalized in the standard way and slightly lower for Xavier-initialized weights. However, the variance of of MSE when using Xavier-initialized weights is higher.

Distribution of misclassified points using ReLU activation.



Misclassifications also occur mostly at the class boundaries. Off-boundary misclassifications behave similarly compared to when using the hyperbolic tangent activation. Noteworthy is the linear pattern of off-boundary misclassifications produced by the ReLU activation function (you will find a pattern reminiscent of curves when using the hyperbolic tangent)

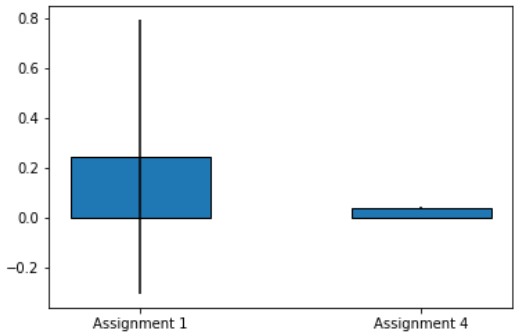
4. **Training ANN in Keras**

(a)

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD
from keras.utils import to_categorical
from keras import backend
import numpy as np
import matplotlib.pyplot as plt
#load and split dataset
dataset = np.load('xor.npy')
X, y = dataset[:,:2], dataset[:,2]
y = np.where(y==-1,0,1) #convert class labels to 0 & 1
y_enc= to_categorical(y)
X_train, X_test, y_train, y_test = X[:80000], X[80000:], y_enc[:80000], y_enc[80000:]


#create the network
mse_keras = []
for run in range(100) :
    model = Sequential()
    model.add(Dense(64,input_dim=2))
    model.add(Activation('tanh'))
    model.add(Dense(2))
    model.add(Activation('softmax'))

    #add optimizer, compile and train
    sgd = SGD(lr=0.005)
    model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['mse'])

    #evaluate loss and predict classes on test set
    model.fit(X_train, y_train, batch_size=1, epochs=1)
    loss = model.evaluate(X_test,y_test)
    mse_keras.append(loss[1])

    backend.clear_session()

mse_mean_keras = np.mean(mse_keras)
```

```
mse_std_keras = np.std(mse_keras)

mse_mlp = np.load('stats_ex_1.npy')
mse_mean_mlp = np.mean(mse_mlp)
mse_std_mlp = np.std(mse_mlp)

plt.bar(x=np.array([1, 2]), height=np.array([mse_mean_mlp,
        mse_mean_keras]), width=0.5, tick_label=np.array(['Assignment_1', 'Assignment_4']),
    yerr=np.array([mse_std_mlp, mse_std_keras]), edgecolor='k')
```

(b) We do not compare MSE with the Cross-entropy loss as it is not possible to directly compare the values of two different loss functions which are computed differently. Instead, we compute the MSE as an additional metric in order to compare it with the MSE from Problem 1.



Solutions to set 10: Deep Neural Networks

1. **Fully connected (dense) networks.**

   Do all necessary imports.

```
import numpy as np
import keras
from keras.layers import Dense, Dropout, Flatten , Conv2D, MaxPooling2D
from matplotlib import pyplot as plotDisplay
```

   (a) Import fashion-mnist

```
### ex 1a
# load data
fashion_mnist_data =keras.datasets.fashion_mnist.load_data()

# unpack data
((train_x , train_y ), ( test_x , test_y )) = fashion_mnist_data

# vectorize
train_x.resize(train_x.shape[0], train_x.shape[1]**2)
test_x.resize(test_x.shape[0], test_x.shape[1]**2)
```

   (b) Normalize.

```
# normalize
train_x = train_x / 255.0
test_x = test_x / 255.0
```

   We normalize between 0 and 1 to aid the convergence process. Recall from gradient descend that if the error is too large, our models will not be able to find a minimum. Further, normalization helps with the vanishing and exploding gradients issues.

   (c) Build a fully connected feed-forward network:

```
fc = keras.models.Sequential()
fc.add(Dense(units=10, activation='relu', input_shape=(784,)))
fc.add(Dense(units=10, activation='relu'))
fc.add(Dense(units=10, activation='softmax'))
```

The dimensionality of the final layer is 10 because we want to classify items in 10 classes (and because our labels are one-hot encoded).

(d) Calculate the weights $(784 \times 10 + 10) + (10 \times 10 + 10) + (10 \times 10 + 10) = 8070$.

(e) Compile the fully connected feed-forward network:

```
fc.compile(optimizer='SGD', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

(f) Train the network for 3 epochs:

```
batch_size = 64
epochs = 3
fc.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y), batch_size=batch_size)
```

Training accuracy ≈ 0.768 and testing accuracy ≈ 0.771.

(g) Learning rate: The default learning rate for the SGD optimizer in Keras is 0.01.

```
### ex 1g
fc2 = keras.models.Sequential()
fc2.add(Dense(units=10, activation='relu', input_shape=(784,)))
fc2.add(Dense(units=10, activation='relu'))
fc2.add(Dense(units=10, activation='softmax'))
sgd = keras.optimizers.SGD(lr=0.005) # change the learning rate
fc2.compile(optimizer=sgd, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
fc2.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y), batch_size=batch_size)
```

Training accuracy ≈ 0.731 and testing accuracy ≈ 0.732.

(h) Increase number of hidden layer neurons The total number of parameters for this network is $(784 \times 64 + 64) + (64 \times 64 + 64) + (64 \times 10 + 10)) = 55050$! The following code produces an appropriate model:

```
### ex 1h
fc3 = keras.models.Sequential()
fc3.add(Dense(units=64, activation='relu', input_shape=(784,)))
fc3.add(Dense(units=64, activation='relu'))
fc3.add(Dense(units=10, activation='softmax'))
sgd = keras.optimizers.SGD(lr=0.01)
fc3.compile(optimizer=sgd, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
fc3.fit(x=train_x, y=train_y, epochs=epochs, validation_data=(test_x, test_y), batch_size=batch_size)
```

Training accuracy ≈ 0.818 and testing accuracy ≈ 0.818.

(i) Reduce training data

```
# reduce size of dataset
train_x = train_x[:20000]
train_y = train_y[:20000]
```

Then we compile a new model and train it on the data. The accuracy of the model on the test set should start falling below the training accuracy (e.g. training accuracy ≈ 0.767 vs test accuracy ≈ 0.752). This phenomenon is called overfitting, and it happens because the network is large enough that it can memorize the training examples. This is problematic, because we are interested in predicting items that are not in the training dataset.

(j) Visualize all items in the test set that the model predicts to be ankle boots in a display loop:

```
predictions = np.argmax(fc.predict(test_x), axis=1)

for i in range(predictions.shape[0]):
    if predictions[i] == 9:
        plotDisplay.imshow(test_x[i, :, :, 0])
        plotDisplay.pause(0.01)

        if test_y[i]!=9:
            print('WRONG classification! True class of this item is %d.' % test_y[i])
            plotDisplay.show()
        else:
            print('Item correctly classified!')
        plotDisplay.cla()
```

Items that are misclassified look somewhat similar to ankle boots.

2. **Convolutions.**

   (a)

$$\mathbf{K}_x * \mathbf{I}_A = \begin{bmatrix} 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \\ 0 & -4 & 0 & 4 & 0 \end{bmatrix} \qquad \mathbf{K}_y * \mathbf{I}_A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{K}_x * \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{K}_y * \mathbf{I}_B = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ -4 & -4 & -4 & -4 & -4 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 4 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

   (b) The filter kernels can be used to detect vertical (K_x) and horizontal (K_y) edges in images (see 'Sobel-Filter' for more information).

3. **Convolutional Neural Networks (CNN).**

```
# reshape for CNN
train_x = train_x.reshape(train_x.shape[0],28,28, 1)
test_x = test_x.reshape(test_x.shape[0],28,28, 1)
input_shape = (train_x[0].shape[0], train_x[0].shape[1], 1)

CNN = keras.models.Sequential([
Conv2D(filters=16, kernel_size=(3,3), activation='relu', input_shape=input_shape),
MaxPooling2D(pool_size=(2,2)),
Conv2D(filters=8, kernel_size=(3,3), activation='relu', input_shape=input_shape),
MaxPooling2D(pool_size=(2,2)),
Flatten(),
Dense(units=10, activation='softmax')
])
```

   (a) Compile the network:

```
CNN.compile(optimizer='SGD', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

   Train the network.

```
CNN.fit( train_x , train_y , epochs=3)
```

Indicatively our architecture achieved an accuracy of 0.8553 and a validation accuracy of 0.8577 with three epochs of training.

(b) Convolutional neural networks have an implicit architectural bias for local connections that comes from performing the convolution operation. This implicit bias seems to let them perform and generalize better on image processing tasks, where spatially close information is important.

(c) Calculate number of weights

Convolutional layers are light on parameters, and max pooling layers do not have any. The first layer has 16 filters of size 3×3 plus their biases for a total of $3x3x16 + 16 = 160$ parameters. The second layer receives an input from each of the 16 filters in the first layer and applies eight 3×3 filters, so it has $3 \times 3 \times 16 \times 8 + 8 = 1160$ parameters, which brings the total so far to 1320.

Most of the trainable parameters for this model actually come from the dense layer at the end. Calculating the dimensionality of the dense layer is tricky because we have to track the dimensionality of the input as it changes when it is propagated through the network. The first convolutional operation returns a $26 \times 26 \times 16$ output (downgraded from 28×28 because we did not use padding). The max pooling operation halves that to $13 \times 13 \times 16$ which is given as input to the next convolutional layer, which also doesn't use padding. So the dimensionality of the output becomes $11 \times 11 \times 8$. Max pooling reduces it to $5 \times 5 \times 8$ that we map on the 10 fashion-mnist classes. So the weights for the neural network are $200 \times 10 + 10 = 2010$. This brings the total number of trainable weights in this convolutional neural network to $1320 + 2010 = 3330$ in total.

4. **BONUS.**

(a) Strided convolutions

```
CNN2 = keras.models.Sequential([
Conv2D( filters=16, kernel_size =(3,3), activation ='relu', input_shape=input_shape, strides =(2,2)),
Conv2D( filters=8, kernel_size =(3,3), activation ='relu', input_shape=input_shape, strides =(2,2)),
Flatten (),
Dense(units=10, activation ='softmax')
])

CNN2.compile(optimizer='SGD', loss=' sparse_categorical_crossentropy ', metrics=['accuracy'])
CNN2.fit(x=train_x , y=train_y , epochs=epochs, validation_data =( test_x , test_y ), batch_size=batch_size )
```

The total number of trainable weights on this network is $3 \times 3 \times 16 + 16) + (3 \times 3 \times 16 \times 8 + 8) + (288 \times 10 + 10) = 4210$. The difference is because we round down odd-number dimensions one time less than in the previous model. Pooling is a cheaper operation than a convolution, as it only needs to make a comparison between the numbers in the current filter area. However, if we use a stride during the convolution operation then we do not have to do as many convolution operations, which results in faster performance of strided convolution despite the increase in the number of trainable parameters.

(b) ADAM optimizer

```
CNN3 = keras.models.Sequential([
Conv2D( filters=16, kernel_size =(3,3), activation ='relu', input_shape=input_shape, strides =(2,2)),
Conv2D( filters=8, kernel_size =(3,3), activation ='relu', input_shape=input_shape, strides =(2,2)),
Flatten (),
Dense(units=10, activation ='softmax')
])

CNN3.compile(optimizer='adam', loss=' sparse_categorical_crossentropy ', metrics=['accuracy'])
CNN3.fit(x=train_x , y=train_y , epochs=epochs, validation_data =( test_x , test_y ), batch_size=batch_size )
```

The ADAM optimizer uses an adaptive learning rate that starts off large as the error is large, and decreases with the error. At the same time, it stabilizes the gradient trajectory by discounting large perturbations

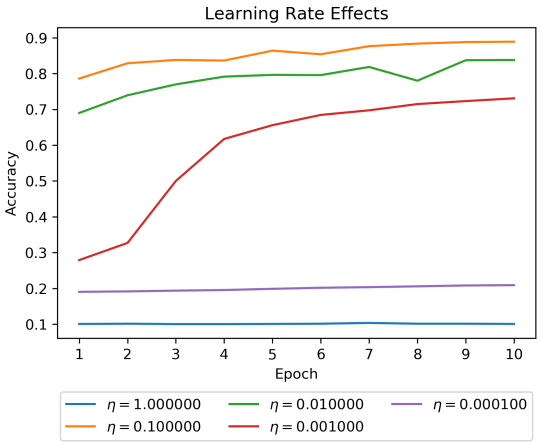
in the gradients, predominantly in the bias updates. This makes the learning process smoother and faster. The method is a weighted combination of two other gradient descend algorithms, Momentum SGD and RMSProp.

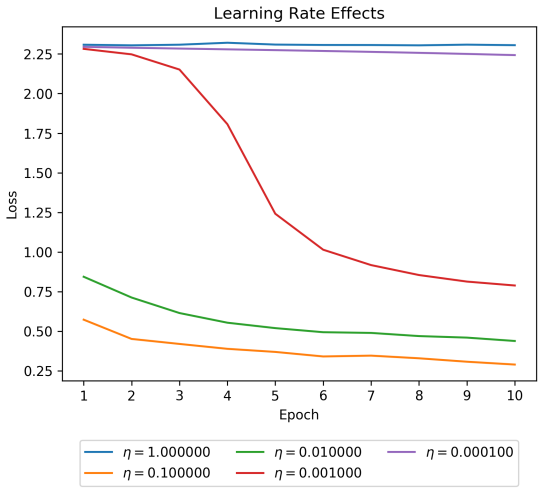
Solutions to set 11: Deep Neural Networks 2

1. **Learning rate in DNN.**

   (a) Please refer to *solution_learning_rate.py* for the code.

   (b) The learning rate mostly affects the speed of learning as should be expected. Smaller learning rates slow down learning and larger learning rates speed up learning. However, with too large a learning rate, such as $\eta = 1$, the model is not able to learn at all. Good performance is achieved with $\eta = 0.1$ and $\eta = 0.01$.

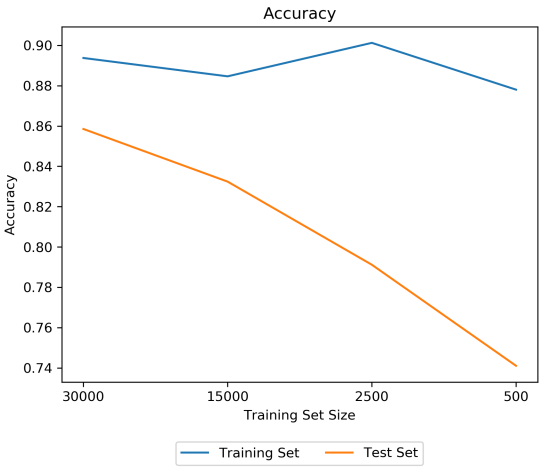


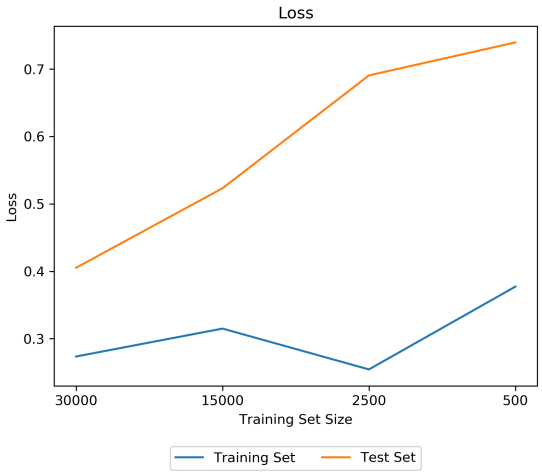


2. **Overfitting in DNN.**

   (a) Please refer to *solution_overfitting.py* for the code.

   (b) With decreasing training set size, the accuracy decreases and the loss increases on the test set. When decreasing the training set size from $N = 30000$ to $N = 500$, the test accuracy falls by almost 10% while the training accuracy stays roughly the same.





3. **Data Augmentation.**

   (a) Please refer to *solution_ data_augmentation.py* for the code.

   (b) Please refer to *solution_data_augmentation.py* for the code.

   (c) The trained network performs significantly worse on the augmented test set (*accuracy* ≈ 0.61) compared to the original test set (*accuracy* ≈ 0.85). This is because examples of the original data set are cropped,

scaled and orientated the same way. The network overfits this specific data distribution and does not generalize well to different types of images.

(d) A model trained on the pretend data set performs better on the pretend test set than the original network does (*accuracy* ≈ 0.72), since the network can learn to cope with the transformations introduced by the image generator.

(e) Training the network using the *fit_generator* method is functionally equivalent to training on the pretend data set obtained from the image generator. However, since now the network is trained using more samples ($steps\_per\_epoch \cdot batch\_size \approx 4000 \cdot 32 = 128000$), the performance is significantly better (*accuracy* $\approx$ 0.77).

4. **Transfer learning.**

(a) Please refer to *solution_transfer_learning.py* for the code.

(b) After training on the first 500 samples, the pretrained network outperforms the *de novo* network (*accuracy* $\approx$ 0.67 vs. 0.36). This is because the pretrained network can transfer what it learned to the new data set. The difference in test accuracy diminishes when training the networks on the complete augmented training set (*accuracy* ≈ 0.74 and 0.7 for the pretrained and *de novo* networks respectively). **Note:** Due to the stochastic nature of the training process, it is possible that sometimes you get a different result.

Solutions to set 12: Recurrent Neural Networks

1. **Predicting time series data.**

(a) Write prepareData and generate a dataset

```
# basic imports
import numpy as np
import matplotlib.pyplot as plt
# keras imports
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN


def prepareData(data, input_steps=1, output=1):
    '''
    This function prepares inputs and labels from raw data.

    | **Args**
    | data:                 Raw data.
    | input_steps:          The number of steps to be used as input.
    | output:               The number of steps to be used as label.
    '''
    # prepare inputs
    X = []
    for example in range(data.shape[0] - input_steps):
        X += [np.reshape(data[example:example + input_steps], (input_steps, 1))]
    # prepare labels
    y = []
    for example in range(input_steps, data.shape[0] - output + 1):
        y += [data[example:example + output]]

    return np.array(X), np.array(y)

# define data size
N = 1000 # data set size
start = 0 # linspace start
end = 100 # linspace end
```

```
34  # generate raw data
35  X = np.linspace( start , end, N)
36  sine = np.sin(X) + X*0.1
```

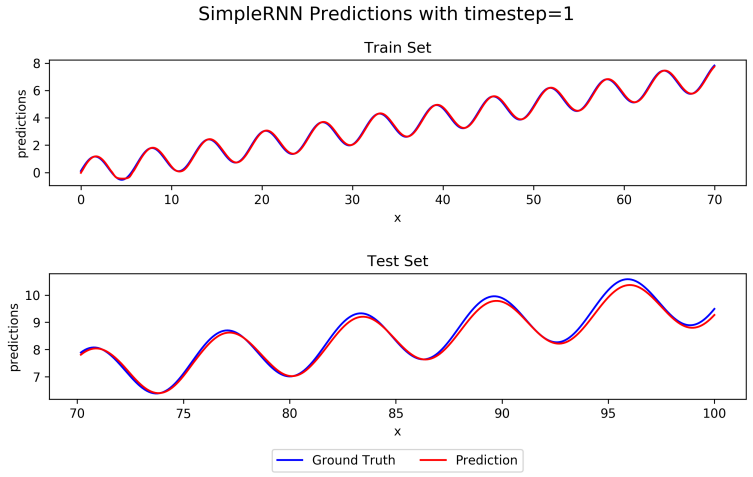
(b) Build a sequential model

```
1  # prepare model
2  model = Sequential ()
3  model.add(SimpleRNN(units=32, input_shape=( input_steps , 1), activation ="relu"))
4  model.add(Dense(output))
5  model.compile(loss='mean_squared_error' , optimizer='adam')
```

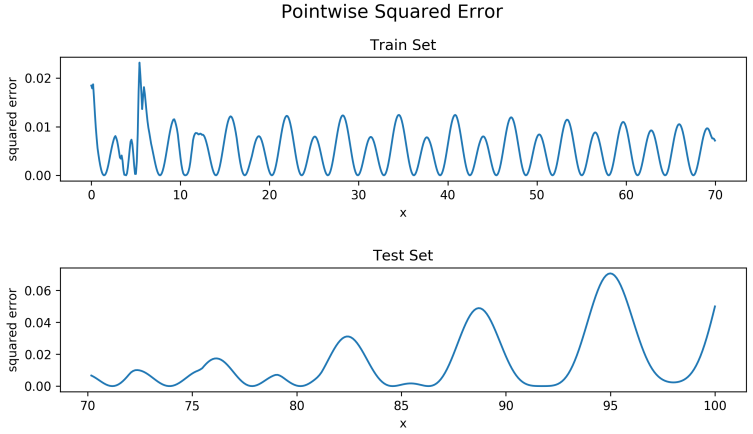
(c) Prepare the data with one timestep.

```
1  # prepare data
2  X, y = prepareData( sine , input_steps =1, output=output)

4  # split data
5  X_train , X_test , y_train , y_test = X[: N_split ], X[N_split :], y[: N_split ], y[ N_split :]

7  # fit model
8  model. fit ( X_train , y_train , epochs=20, batch_size =16, verbose=2)
```

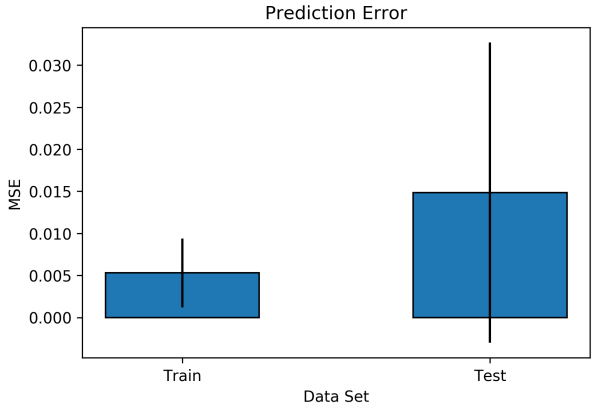
The RNN is reasonably effective in predicting the next values of the sinusoid.



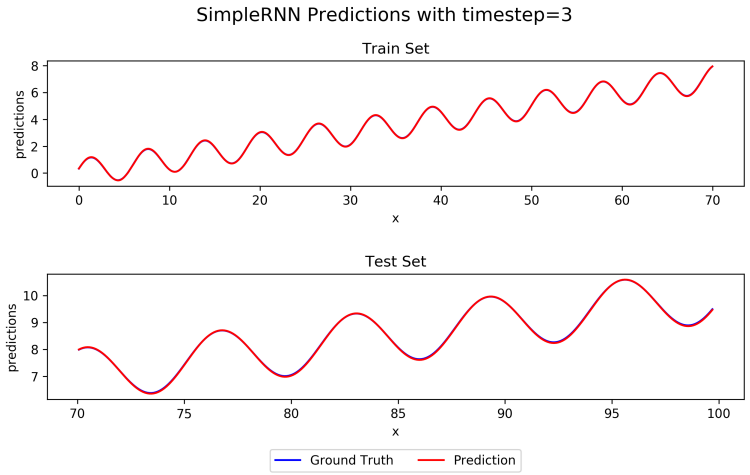
(d) Two things jump out in the pointwise squared errors of the model. Firstly, the error peaks right after the derivative of the function changes sign. Secondly, when the derivative changes from negative to positive, the error is larger that when the derivative changes from positive to negative. The reason for the latter is that the RNN has to combat both the change in the derivative of the sine component as well as the overall upwards trend.



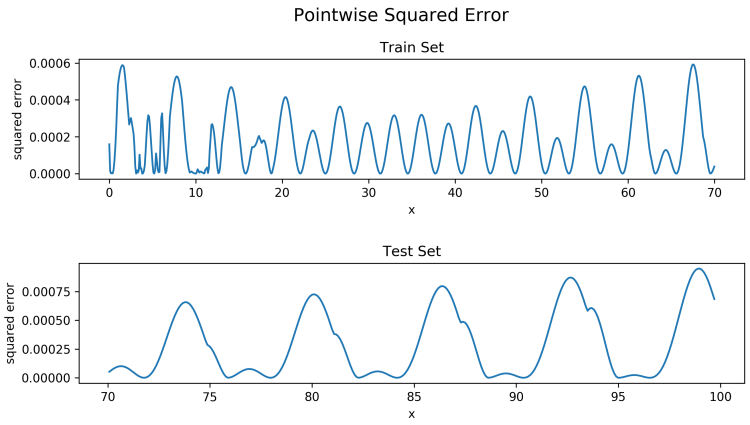
The MSE increases for the test set, as the RNN does not keep up with the general upwards trend.



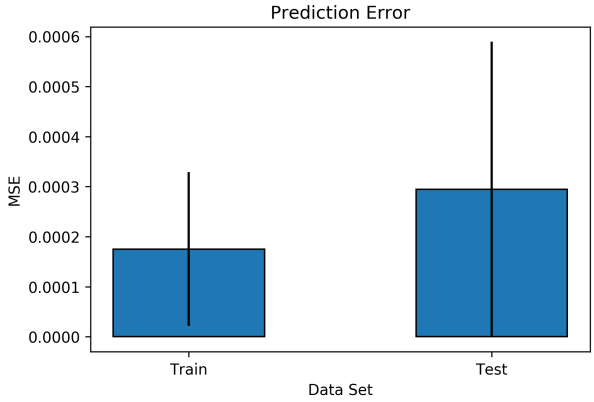
(e) On a first reading increasing the input length has produced better performance for the RNN. The network's generalization ability does not seem to deteriorate over time as much.



While the error is still dependent on the sign change of the derivative of the sinusoid, the squared errors are much smaller and don't increase overall.



The MSE is lower in general, and the test error is much closer to training error.


Prediction Error
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
MSE
Train
Test
Data Set

2. **Learning long-term dependencies.**

(a) Loading the text and looking at basic characteristics :
Text length : 523485
Number of unique characters in the text : 64

```
import io
path = 'faust.txt'
with io.open(path, encoding='utf-8') as f:
    text = f.read().lower()

print('Text length:', len(text))
chars = sorted(list(set(text)))
print('Number of unique characters in text:', len(chars))
```

(b) Create mapping dictionaries and split the data into sequences as follows :

```
char2int = dict((c, i) for i, c in enumerate(chars))
int2char = dict((i, c) for i, c in enumerate(chars))

len_seq = 50
step = 3
sequences = []
targets = []
for i in range(0, len(text) - len_seq, step):
    sequences.append(text[i: i + len_seq])
    targets.append(text[i + len_seq])
```

(c) Compute the one-hot encoding using the dictionaries created

```
x = np.zeros((len(sequences), len_seq, len(chars)))
y = np.zeros((len(sequences), len(chars)))

#compute one-hot encoding
for i, sentence in enumerate(sequences):
    for t, char in enumerate(sentence):
        x[i, t, char2int[char]] = 1
    y[i, char2int[targets[i]]] = 1
```

(d) The output layer should have as many units as the number of unique characters in our text, in this case, 64. Since this is a classification problem, the cross-entropy loss in combination with softmax activation in the output layer should be used.

```
model = Sequential()
model.add(LSTM(256, input_shape=(len_seq, len(chars))))
model.add(Dense(len(chars), activation='softmax'))

optimizer = Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

(e) Call `model.fit` to train the network. The weights can be saved using the keras function `model.save_weights`.

(f) After only 20 epochs, the LSTM output with a random seed looks like this :

```
ließ' es dem großen herren gut,
das arme rung,
und wie der geistern auf.

mephistopheles:
die pracht geschmückt' er soll erstaaben.
und mit der schlagen, wo sich der schlassen
```

```
die gewalt sich welt und gar zu regen;
und, die meiner weise schweigen schließen
und trete doch diese schaffen schallen,
in der feind, geleben auf.
das glauben frei, was ihr gedrängen,
wenn ich mich heran.

helena:
wie weiß die schärfe nacht die schlagen,
ich habe mich mit der menser gehn
die fester glücknen stürke,
```

Although the model is still far from producing meaningful text, we can see that the LSTM has learned many words as well as the structure of the text very well. It imitates the style of the text we provided it.

(g) The output of a simple RNN after 20 epochs still looks like this :

```
die früh sich einst dem trüben blick gest sterst stest
stest stis st stis sie sstest slust!
serst srürs sterstst ut ust irst sest sten isteitst set
 sterst ssestirst st stestist sieststst slistest.
dat st st st werstst sis stest stes sisst sist stst srerst
sterstest srerst ist tist st esst.
sest st st sustst diest sist st,
sist sers st steststst st sistüst stist sist sterst.
stist dist sis ststest,
sst.
stersest stes st stest stist sut stes sterst sstst
ststestersterst stestest ssts sisest.
ser striesterst sus stest sttest
```

While the LSTM generates *Faust*-like text, the RNN gets stuck in predictive loops and does not produce any meaningful words. This is because of the inability of the RNN to learn long-range dependencies.

Solutions to set 13: Hopfield Network

1. **Asynchrounous updates in Hopfield network.**

(a) The stored pattern had 4 elements. This can be inferred by counting how many rows (or columns) the weight matrix has.

(b) We should start with an arbitrary pattern x and iterate through the update algorithm until the pattern does not change. We initialize the pattern arbitrarily with $x = [+1,-1,+1,-1]^T$. We start updating from the first element:

$$\begin{bmatrix} \mathbf{0} & \mathbf{-1} & \mathbf{-1} & \mathbf{+1} \\ -1 & 0 & +1 & -1 \\ -1 & +1 & 0 & -1 \\ +1 & -1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} +1 \\ -1 \\ +1 \\ -1 \end{bmatrix} = \begin{bmatrix} \mathbf{-1} \\ -1 \\ +1 \\ -1 \end{bmatrix} \tag{189}$$

Now we continue with the second one:

$$\begin{bmatrix} 0 & -1 & -1 & +1 \\ \mathbf{-1} & \mathbf{0} & \mathbf{+1} & \mathbf{-1} \\ -1 & +1 & 0 & -1 \\ +1 & -1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} -1 \\ -1 \\ +1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ \mathbf{+1} \\ +1 \\ -1 \end{bmatrix} \tag{190}$$

Since the second element has changed, we should redo the procedure for the first element to see whether

it changes:

$$\begin{bmatrix} \mathbf{0} & \mathbf{-1} & \mathbf{-1} & \mathbf{+1} \\ -1 & 0 & +1 & -1 \\ -1 & +1 & 0 & -1 \\ +1 & -1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} -1 \\ +1 \\ +1 \\ -1 \end{bmatrix} = \begin{bmatrix} \mathbf{-1} \\ +1 \\ +1 \\ -1 \end{bmatrix} \tag{191}$$

Since it did not change, we can continue for the third element:

$$\begin{bmatrix} 0 & -1 & -1 & +1 \\ -1 & 0 & +1 & -1 \\ \mathbf{-1} & \mathbf{+1} & \mathbf{0} & \mathbf{-1} \\ +1 & -1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} -1 \\ -1 \\ +1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \\ \mathbf{+1} \\ -1 \end{bmatrix} \tag{192}$$

And for the last one:

$$\begin{bmatrix} 0 & -1 & -1 & +1 \\ -1 & 0 & +1 & -1 \\ -1 & +1 & 0 & -1 \\ \mathbf{+1} & \mathbf{-1} & \mathbf{-1} & \mathbf{0} \end{bmatrix} \times \begin{bmatrix} -1 \\ -1 \\ +1 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ +1 \\ +1 \\ \mathbf{-1} \end{bmatrix} \tag{193}$$

In this case we have found a stable solution at $[+1,-1,-1,+1]^T$. The pattern the network was trained on could be this or its inverse, $[-1,+1,+1,-1]^T$, since if x is an attractor of the network, so is $-x$.

(c) Refer to `ex1.py` for the code.

(d) The energy of the network always decreases or stays constant, and the stored pattern corresponds to the lowest energy level. Depending on the initial guess and the order in which units are updated, the solution is found more or less quickly.

2. **Pattern storage and retrieval in Hopfield network.** Refer to `ex2.py` for the code.

(b) Visualization of the input images (Fig. 77).

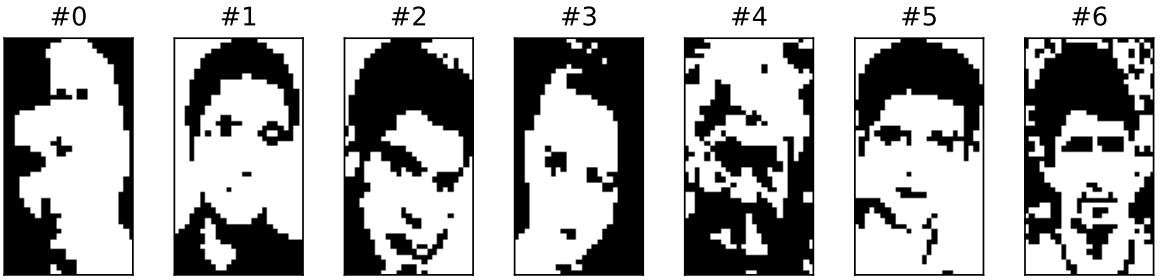


Figure 77: Input images

(d) Input images were stored in a Hopfield network. Noisy versions of the images were given as retrieval cues and the network was let to evolve for 100 iterations of the update algorithm (Fig. 78).

(e) We used correlation coefficients to measure the similarity between the stored and retrieved images (Fig. 78, r values in the lower row). If the correlation coefficient is 1, the retrieved image is identical to the stored one and we have a lossless retrieval. A correlation coefficient of 0 would imply that the retrieved image has no relation to the stored one. On the other hand, a correlation coefficient of -1 means that the retrieved image is the negative version of the stored image.

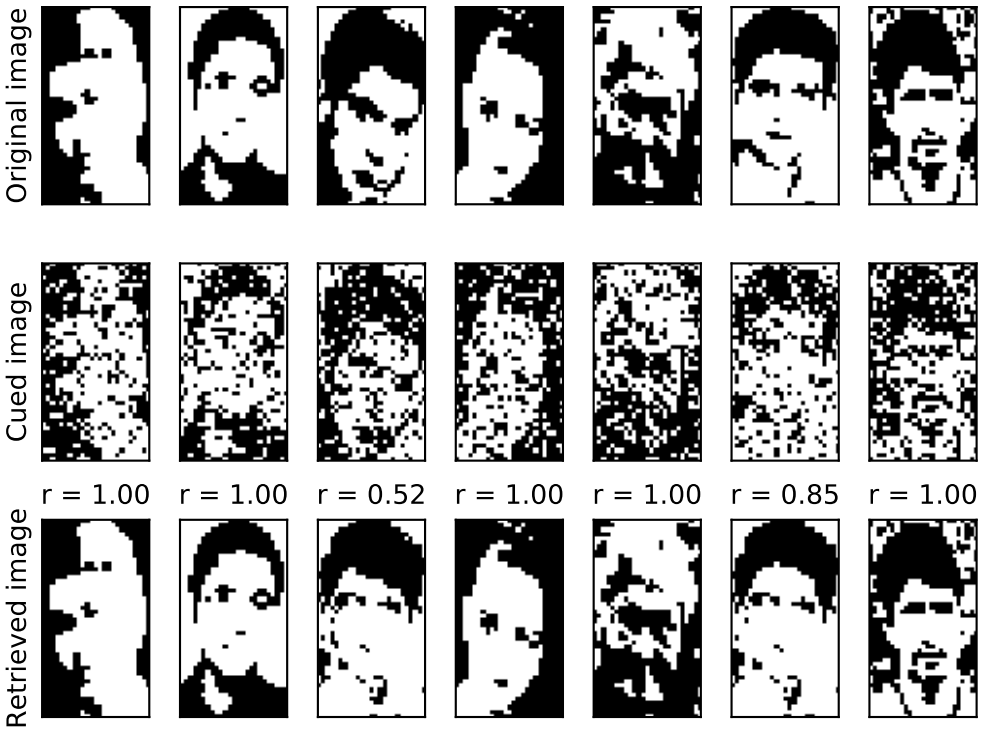


Figure 78: Retrieval process. Original images (upper row), cued noisy images (middle row) and the retrieved ones (lower row).

(f) The network confused image #2 with image #5 and by looking at the correlation coefficient computed for all pairs of images (Fig. 79) we see that these two images are more correlated than the other pairs.

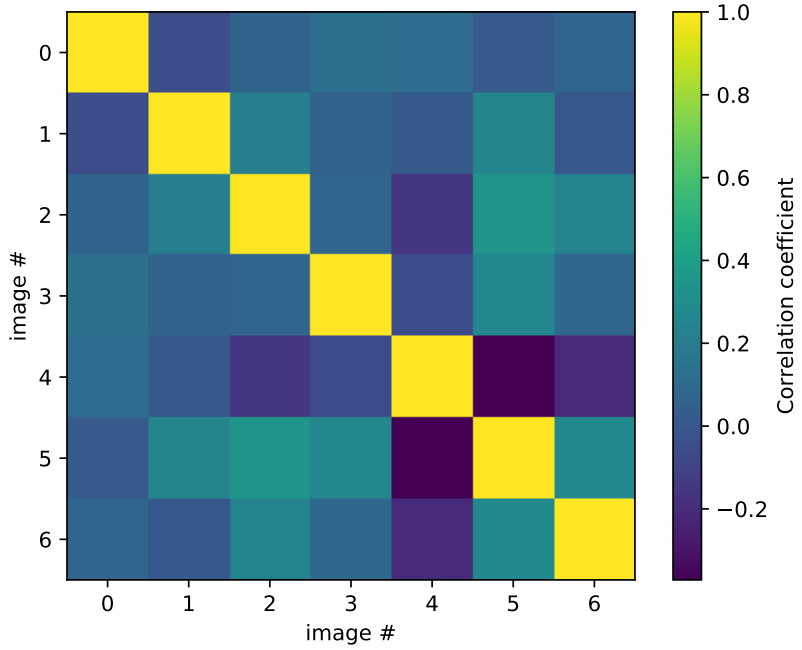


Figure 79: Correlation coefficient computed across all pairs of images.

(g) Our network proves very resistant to noise, being able to restore most noisy patterns, although the performance breaks down when approaching a noise level of 575 (1150/2, equivalent to flipping 50% of all pixels). For noise levels above this, the cued images become more similar to their negative versions, which are also spurious attractors of the network. The network then converges to those attractors leading to negative correlation values.

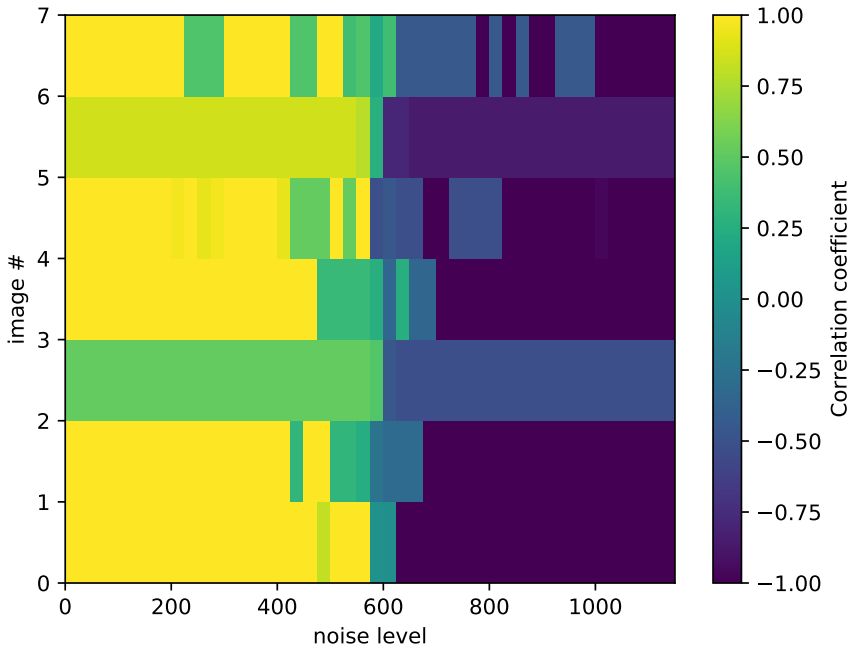


Figure 80: Correlation coefficient in the retrieval process was computed for all individual images with varying noise level.

3. **Capacity of Hopfield network.** Refer to `ex3.py` for the code.

   Under low network load conditions, the stored patterns are indeed stable attractors of the network, and thus the error rate remains close to 0. However, at a critical value of ≈ 0.14, catastrophic interference occurs and the network is unable to retrieve any patterns correctly. Therefore the maximum number of patterns that can be safely stored in a Hopfield network is approximately 0.14 times the network size.

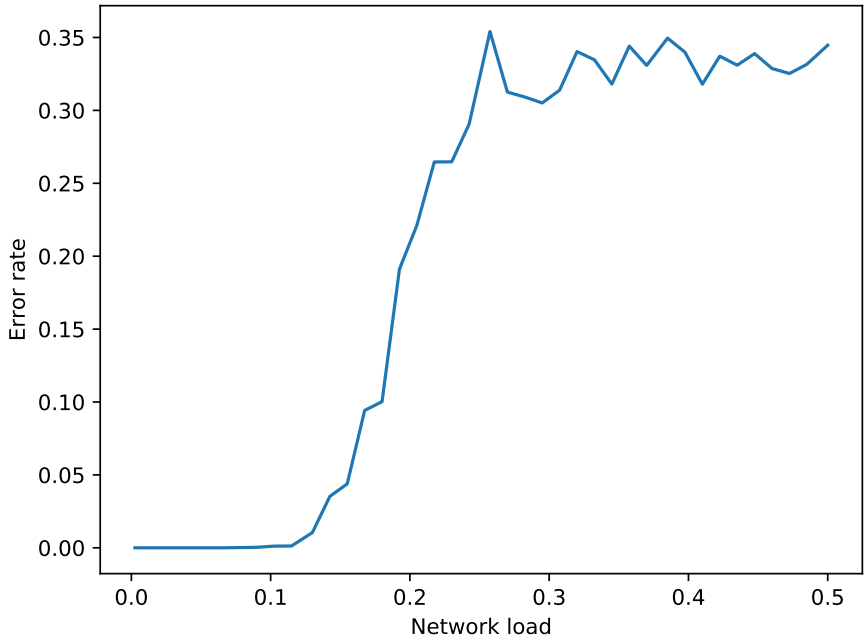


Figure 81: Probability of error in a Hopfield network as a function of the load.

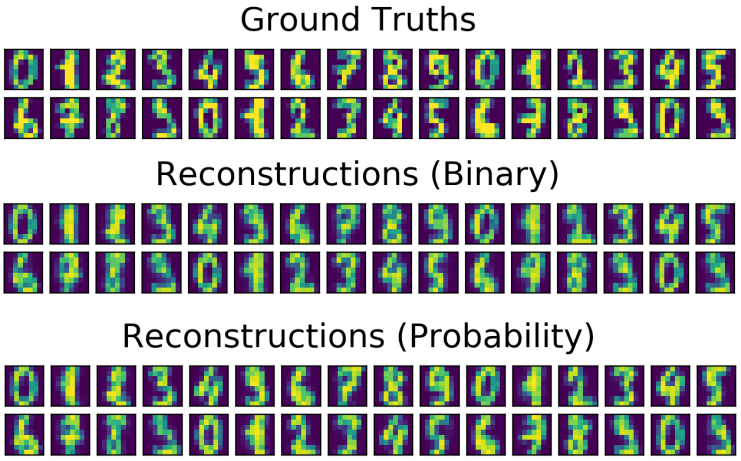
Solutions to set 14: Bolzmann Machines

1. **Contrastive Divergence.** Please refer to *sol_rbm.py* for the code.
2. **Training RBMs.**

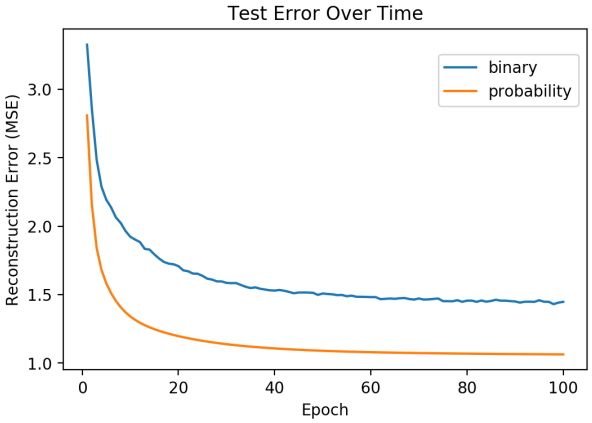
   (a) Please refer to *sol_learning.py* for the code.

   (b) dito

   (c) Please refer to *sol_reconstruction.py* for the implementation of *reconstruction*. Reconstructions:



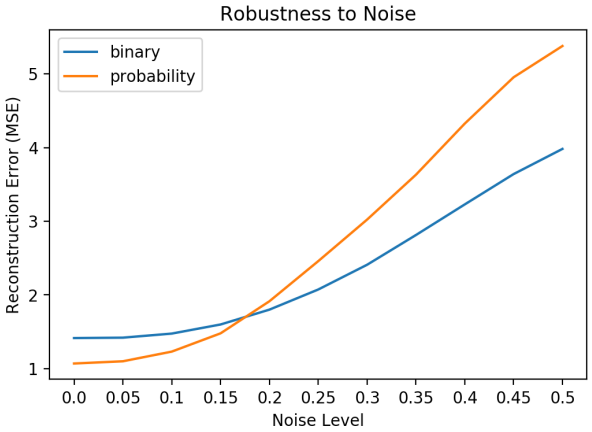
   (d) For both RBMs, MSE of the reconstructions of the test set quickly decreases and then plateaues. When using binary hidden units, the test error is slightly higher.



3. **Pattern completion.**

   (a) Please refer to *sol_noise.py* for the code.

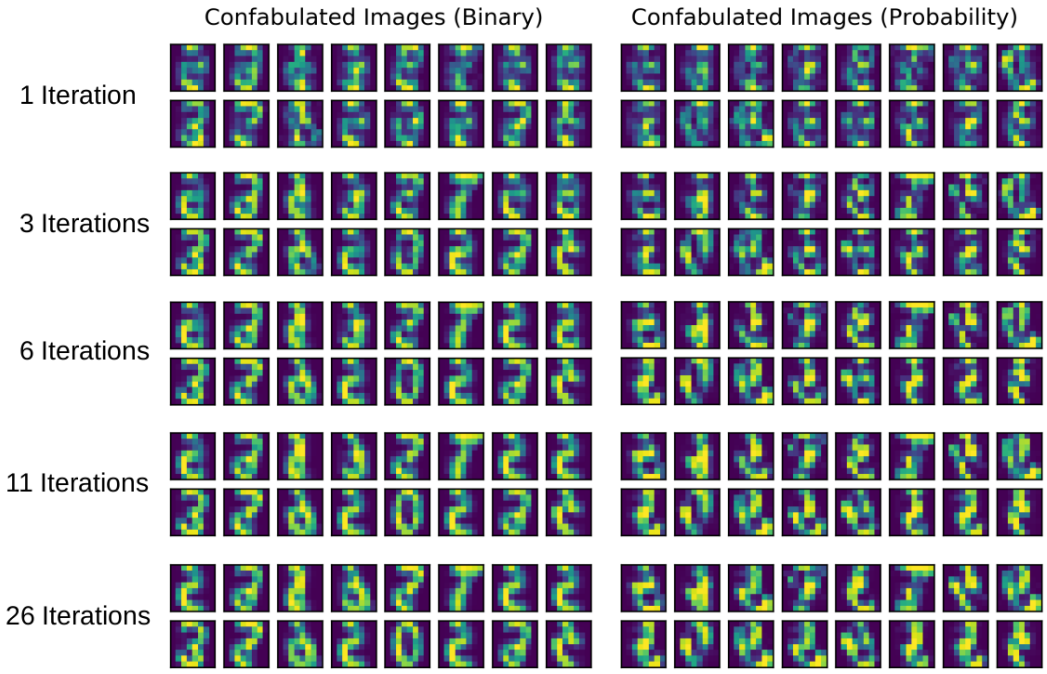
   (b) The binary RBM is more robust to noise due to the thresholding that its units perform.



4. **Data confabulation.**

   (a) Please refer to *sol_confabulation.py* for the code.

   (b) Both RBMs refine their confabulated images over the course of iterations. For example, gaps present in digits after the first iteration are slowly filled in by in later iterations.



While the binary RBM is able to confabulate images that look like actual digits, the probabilistic RBM is not. This property is consistent with the observation in the previous problem that the binary RBM has a lower reconstruction error than the probabilistic RBM at large noise levels. The binary RBM has a tendency to confabulate digits that look like a '2' or mirrored 'S' – possibly due to many digits sharing this pattern at least partially.