Data Analysis and Machine Learning: Logistic Regression, Gradient Methods and begin Neural Networks

Morten Hjorth-Jensen[1,2]

[1]Department of Physics and Center for Computing in Science Education, University of Oslo, Norway
partment of Physics and Astronomy and Facility for Rare Ion Beams and National Superconducting Cyclotron Laboratory, Michigan State University, U

Oct 11, 2021

Logistic Regression

In linear regression our main interest was centered on learning the coefficients of a functional fit (say a polynomial) in order to be able to predict the response of a continuous variable on some unseen data. The fit to the continuous variable y_i is based on some independent variables $\hat{x}_i$. Linear regression resulted in analytical expressions for standard ordinary Least Squares or Ridge regression (in terms of matrices to invert) for several quantities, ranging from the variance and thereby the confidence intervals of the parameters $\hat{\beta}$ to the mean squared error. If we can invert the product of the design matrices, linear regression gives then a simple recipe for fitting our data.

Classification problems

Classification problems, however, are concerned with outcomes taking the form of discrete variables (i.e. categories). We may for example, on the basis of DNA sequencing for a number of patients, like to find out which mutations are important for a certain disease; or based on scans of various patients' brains, figure out if there is a tumor or not; or given a specific physical system, we'd like to identify its state, say whether it is an ordered or disordered system (typical situation in solid state physics); or classify the status of a patient, whether she/he has a stroke or not and many other similar situations.

The most common situation we encounter when we apply logistic regression is that of two possible outcomes, normally denoted as a binary outcome, true or false, positive or negative, success or failure etc.

Optimization and Deep learning

Logistic regression will also serve as our stepping stone towards neural network algorithms and supervised deep learning. For logistic learning, the minimization of the cost function leads to a non-linear equation in the parameters $\hat{\beta}$. The optimization of the problem calls therefore for minimization algorithms. This forms the bottle neck of all machine learning algorithms, namely how to find reliable minima of a multi-variable function. This leads us to the family of gradient descent methods. The latter are the working horses of basically all modern machine learning algorithms.

We note also that many of the topics discussed here on logistic regression are also commonly used in modern supervised Deep Learning models, as we will see later.

Basics

We consider the case where the dependent variables, also called the responses or the outcomes, y_i are discrete and only take values from $k = 0, \ldots, K-1$ (i.e. K classes).

The goal is to predict the output classes from the design matrix $\hat{X} \in \mathbb{R}^{n \times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and $y_i = 1$. Our outcomes could represent the status of a credit card user that could default or not on her/his credit card debt. That is

$$y_i = \begin{bmatrix} 0 & \text{no} \\ 1 & \text{yes} \end{bmatrix}.$$

Linear classifier

Before moving to the logistic model, let us try to use our linear regression model to classify these two outcomes. We could for example fit a linear model to the default case if $y_i > 0.5$ and the no default case $y_i \leq 0.5$.

We would then have our weighted linear combination, namely

$$\hat{y} = \hat{X}^T \hat{\beta} + \hat{\epsilon}, \tag{1}$$

where $\hat{y}$ is a vector representing the possible outcomes, $\hat{X}$ is our $n \times p$ design matrix and $\hat{\beta}$ represents our estimators/predictors.

Some selected properties

The main problem with our function is that it takes values on the entire real axis. In the case of logistic regression, however, the labels y_i are discrete variables. A typical example is the credit card data discussed below here, where we can set the state of defaulting the debt to $y_i = 1$ and not to $y_i = 0$ for one the persons in the data set (see the full example below).

One simple way to get a discrete output is to have sign functions that map the output of a linear regressor to values $\{0, 1\}$, $f(s_i) = sign(s_i) = 1$ if $s_i \geq 0$ and 0 if otherwise. We will encounter this model in our first demonstration of neural networks. Historically it is called the "perceptron" model in the machine learning literature. This model is extremely simple. However, in many cases it is more favorable to use a "soft" classifier that outputs the probability of a given category. This leads us to the logistic function.

Simple example

The following example on data for coronary heart disease (CHD) as function of age may serve as an illustration. In the code here we read and plot whether a person has had CHD (output = 1) or not (output = 0). This ouput is plotted the person's against age. Clearly, the figure shows that attempting to make a standard linear regression fit may not be very meaningful.

```
# Common imports
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.model_selection import train_test_split
from sklearn.utils import resample
from sklearn.metrics import mean_squared_error
from IPython.display import display
from pylab import plt, mpl
plt.style.use('seaborn')
mpl.rcParams['font.family'] = 'serif'

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)

if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')

infile = open(data_path("chddata.csv"),'r')

# Read the chd data as  csv file and organize the data into arrays with age group, age, and chd
```

```
chd = pd.read_csv(infile, names=('ID', 'Age', 'Agegroup', 'CHD'))
chd.columns = ['ID', 'Age', 'Agegroup', 'CHD']
output = chd['CHD']
age = chd['Age']
agegroup = chd['Agegroup']
numberID  = chd['ID']
display(chd)

plt.scatter(age, output, marker='o')
plt.axis([18,70.0,-0.1, 1.2])
plt.xlabel(r'Age')
plt.ylabel(r'CHD')
plt.title(r'Age distribution and Coronary heart disease')
plt.show()
```

Plotting the mean value for each group

What we could attempt however is to plot the mean value for each group.

```
agegroupmean = np.array([0.1, 0.133, 0.250, 0.333, 0.462, 0.625, 0.765, 0.800])
group = np.array([1, 2, 3, 4, 5, 6, 7, 8])
plt.plot(group, agegroupmean, "r-")
plt.axis([0,9,0, 1.0])
plt.xlabel(r'Age group')
plt.ylabel(r'CHD mean values')
plt.title(r'Mean values for each age group')
plt.show()
```

We are now trying to find a function $f(y|x)$, that is a function which gives us an expected value for the output y with a given input x. In standard linear regression with a linear dependence on x, we would write this in terms of our model

$$f(y_i|x_i) = \beta_0 + \beta_1 x_i.$$

This expression implies however that $f(y_i|x_i)$ could take any value from minus infinity to plus infinity. If we however let $f(y|y)$ be represented by the mean value, the above example shows us that we can constrain the function to take values between zero and one, that is we have $0 \leq f(y_i|x_i) \leq 1$. Looking at our last curve we see also that it has an S-shaped form. This leads us to a very popular model for the function f, namely the so-called Sigmoid function or logistic model. We will consider this function as representing the probability for finding a value of y_i with a given x_i.

The logistic function

Another widely studied model, is the so-called perceptron model, which is an example of a "hard classification" model. We will encounter this model when we discuss neural networks as well. Each datapoint is deterministically assigned to a category (i.e $y_i = 0$ or $y_i = 1$). In many cases, and the coronary heart disease data forms one of many such examples, it is favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category

k. Logistic regression is the most common example of a so-called soft classifier. In logistic regression, the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the so-called logit function (or Sigmoid) which is meant to represent the likelihood for a given event,

$$p(t) = \frac{1}{1 + \exp -t} = \frac{\exp t}{1 + \exp t}.$$

Note that $1 - p(t) = p(-t)$.

Examples of likelihood functions used in logistic regression and nueral networks

The following code plots the logistic function, the step function and other functions we will encounter from here and on.

```
"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative scale.
The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')

plt.show()

"""Step Function"""
z = numpy.arange(-5, 5, .02)
step_fn = numpy.vectorize(lambda z: 1.0 if z >= 0.0 else 0.0)
step = step_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, step)
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('step function')

plt.show()

"""tanh Function"""
```

```
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.tanh(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, t)
ax.set_ylim([-1.0, 1.0])
ax.set_xlim([-2*mt.pi,2*mt.pi])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('tanh function')

plt.show()
```

Two parameters

We assume now that we have two classes with y_i either 0 or 1. Furthermore we assume also that we have only two parameters β in our fitting of the Sigmoid function, that is we define probabilities

$$p(y_i = 1|x_i, \hat{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}),$$

where $\hat{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

Note that we used

$$p(y_i = 0|x_i, \hat{\beta}) = 1 - p(y_i = 1|x_i, \hat{\beta}).$$

Maximum likelihood

In order to define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0, 1\}$ and where the data points are drawn independently, we use the so-called Maximum Likelihood Estimation (MLE) principle. We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i, that is

$$P(\mathcal{D}|\hat{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \hat{\beta})\right]^{y_i} \left[1 - p(y_i = 1|x_i, \hat{\beta}))\right]^{1-y_i}$$

from which we obtain the log-likelihood and our **cost/loss** function

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i \log p(y_i = 1|x_i, \hat{\beta}) + (1 - y_i) \log \left[1 - p(y_i = 1|x_i, \hat{\beta}))\right]\right).$$

The cost function rewritten

Reordering the logarithms, we can rewrite the **cost/loss** function as

$$\mathcal{C}(\hat{\beta}) = \sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to β. Since the cost (error) function is just the negative log-likelihood, for logistic regression we have that

$$\mathcal{C}(\hat{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right).$$

This equation is known in statistics as the **cross entropy**. Finally, we note that just as in linear regression, in practice we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization as we did for Ridge and Lasso regression.

Minimizing the cross entropy

The cross entropy is a convex function of the weights $\hat{\beta}$ and, therefore, any local minimizer is a global minimizer.

Minimizing this cost function with respect to the two parameters β_0 and β_1 we obtain

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right),$$

and

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \beta_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i\frac{\exp\left(\beta_0 + \beta_1 x_i\right)}{1 + \exp\left(\beta_0 + \beta_1 x_i\right)}\right).$$

A more compact expression

Let us now define a vector $\hat{y}$ with n elements y_i, an $n \times p$ matrix $\hat{X}$ which contains the x_i values and a vector $\hat{p}$ of fitted probabilities $p(y_i|x_i, \hat{\beta})$. We can rewrite in a more compact form the first derivative of cost function as

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T\left(\hat{y} - \hat{p}\right).$$

If we in addition define a diagonal matrix $\hat{W}$ with elements $p(y_i|x_i, \hat{\beta})(1 - p(y_i|x_i, \hat{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}\partial \hat{\beta}^T} = \hat{X}^T\hat{W}\hat{X}.$$

Extending to more predictors

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log \frac{p(\hat{\beta}\hat{x})}{1-p(\hat{\beta}\hat{x})} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p.$$

Here we defined $\hat{x} = [1, x_1, x_2, \ldots, x_p]$ and $\hat{\beta} = [\beta_0, \beta_1, \ldots, \beta_p]$ leading to

$$p(\hat{\beta}\hat{x}) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}{1+\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}.$$

Including more classes

Till now we have mainly focused on two classes, the so-called binary system. Suppose we wish to extend to K classes. Let us for the sake of simplicity assume we have only two predictors. We have then following model

$$\log \frac{p(C=1|x)}{p(K|x)} = \beta_{10} + \beta_{11} x_1,$$

and

$$\log \frac{p(C=2|x)}{p(K|x)} = \beta_{20} + \beta_{21} x_1,$$

and so on till the class $C = K-1$ class

$$\log \frac{p(C=K-1|x)}{p(K|x)} = \beta_{(K-1)0} + \beta_{(K-1)1} x_1,$$

and the model is specified in term of $K-1$ so-called log-odds or **logit** transformations.

More classes

In our discussion of neural networks we will encounter the above again in terms of a slightly modified function, the so-called **Softmax** function.

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression (also known as softmax regression), multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks. Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the predicted probability for the k-th class given a sample vector $\hat{x}$ and a weighting vector $\hat{\beta}$ is (with two predictors):

$$p(C=k|\mathbf{x}) = \frac{\exp\left(\beta_{k0} + \beta_{k1} x_1\right)}{1+\sum_{l=1}^{K-1}\exp\left(\beta_{l0} + \beta_{l1} x_1\right)}.$$

It is easy to extend to more predictors. The final class is

$$p(C = K|\mathbf{x}) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_{l1}x_1)},$$

and they sum to one. Our earlier discussions were all specialized to the case with two classes only. It is easy to see from the above that what we derived earlier is compatible with these equations.

To find the optimal parameters we would typically use a gradient descent method. Newton's method and gradient descent methods are discussed in the material on optimization methods below in these slides.

Wisconsin Cancer Data

We show here how we can use a simple regression case on the breast cancer data using Logistic regression as our algorithm for classification.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
#now scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Logistic Regression
logreg.fit(X_train_scaled, y_train)
print("Test set accuracy Logistic Regression with scaled data: {:.2f}".format(logreg.score(X_test_
```

Using the correlation matrix

In addition to the above scores, we could also study the covariance (and the correlation matrix). We use **Pandas** to compute the correlation matrix.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression
cancer = load_breast_cancer()
import pandas as pd
```

```
# Making a data frame
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)

fig, axes = plt.subplots(15,2,figsize=(10,20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]
ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:,i], bins =50)
    ax[i].hist(malignant[:,i], bins = bins, alpha = 0.5)
    ax[i].hist(benign[:,i], bins = bins, alpha = 0.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["Malignant", "Benign"], loc ="best")
fig.tight_layout()
plt.show()

import seaborn as sns
correlation_matrix = cancerpd.corr().round(1)
# use the heatmap function from seaborn to plot the correlation matrix
# annot = True to print the values inside the square
plt.figure(figsize=(15,8))
sns.heatmap(data=correlation_matrix, annot=True)
plt.show()
```

Discussing the correlation data

In the above example we note two things. In the first plot we display the overlap of benign and malignant tumors as functions of the various features in the Wisconsing breast cancer data set. We see that for some of the features we can distinguish clearly the benign and malignant cases while for other features we cannot. This can point to us which features may be of greater interest when we wish to classify a benign or not benign tumour.

In the second figure we have computed the so-called correlation matrix, which in our case with thirty features becomes a 30×30 matrix.

We constructed this matrix using **pandas** via the statements

```
cancerpd = pd.DataFrame(cancer.data, columns=cancer.feature_names)
```

and then

```
correlation_matrix = cancerpd.corr().round(1)
```

Diagonalizing this matrix we can in turn say something about which features are of relevance and which are not. This leads us to the classical Principal Component Analysis (PCA) theorem with applications.

Other measures in classification studies: Cancer Data again

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
```

```
from sklearn.datasets import load_breast_cancer
from sklearn.linear_model import LogisticRegression

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
#now scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Logistic Regression
logreg.fit(X_train_scaled, y_train)
print("Test set accuracy Logistic Regression with scaled data: {:.2f}".format(logreg.score(X_test_


from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
#Cross validation
accuracy = cross_validate(logreg,X_test_scaled,y_test,cv=10)['test_score']
print(accuracy)
print("Test set accuracy with Logistic Regression  and scaled data: {:.2f}".format(logreg.score(X_


import scikitplot as skplt
y_pred = logreg.predict(X_test_scaled)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
plt.show()
y_probas = logreg.predict_proba(X_test_scaled)
skplt.metrics.plot_roc(y_test, y_probas)
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
plt.show()
```

Optimization, the central part of any Machine Learning algortithm

Almost every problem in machine learning and data science starts with a dataset X, a model $g(\beta)$, which is a function of the parameters β and a cost function $C(X, g(\beta))$ that allows us to judge how well the model $g(\beta)$ explains the observations X. The model is fit by finding the values of β that minimize the cost function. Ideally we would be able to solve for β analytically, however this is not possible in general and we must use some approximative/numerical method to compute the minimum.

Revisiting our Logistic Regression case

In our discussion on Logistic Regression we studied the case of two classes, with y_i either 0 or 1. Furthermore we assumed also that we have only two parameters β in our fitting, that is we defined probabilities

$$p(y_i = 1|x_i, \boldsymbol{\beta}) = \frac{\exp(\beta_0 + \beta_1 x_i)}{1 + \exp(\beta_0 + \beta_1 x_i)},$$
$$p(y_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta}),$$

where $\boldsymbol{\beta}$ are the weights we wish to extract from data, in our case β_0 and β_1.

The equations to solve

Our compact equations used a definition of a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\boldsymbol{X}$ which contains the x_i values and a vector $\boldsymbol{p}$ of fitted probabilities $p(y_i|x_i, \boldsymbol{\beta})$. We rewrote in a more compact form the first derivative of the cost function as

$$\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p}).$$

If we in addition define a diagonal matrix $\boldsymbol{W}$ with elements $p(y_i|x_i, \boldsymbol{\beta})(1 - p(y_i|x_i, \boldsymbol{\beta})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}.$$

This defines what is called the Hessian matrix.

Solving using Newton-Raphson's method

If we can set up these equations, Newton-Raphson's iterative method is normally the method of choice. It requires however that we can compute in an efficient way the matrices that define the first and second derivatives.

Our iterative scheme is then given by

$$\boldsymbol{\beta}^{\rm new} = \boldsymbol{\beta}^{\rm old} - \left(\frac{\partial^2 \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^T}\right)^{-1}_{\boldsymbol{\beta}^{\rm old}} \times \left(\frac{\partial \mathcal{C}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}}\right)_{\boldsymbol{\beta}^{\rm old}},$$

or in matrix form as

$$\boldsymbol{\beta}^{\rm new} = \boldsymbol{\beta}^{\rm old} - \left(\boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}\right)^{-1} \times \left(-\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{p})\right)_{\boldsymbol{\beta}^{\rm old}}.$$

The right-hand side is computed with the old values of β.

If we can compute these matrices, in particular the Hessian, the above is often the easiest method to implement.

Brief reminder on Newton-Raphson's method

Let us quickly remind ourselves how we derive the above method.

Perhaps the most celebrated of all one-dimensional root-finding routines is Newton's method, also called the Newton-Raphson method. This method requires the evaluation of both the function f and its derivative f' at arbitrary points. If you can only calculate the derivative numerically and/or your function is not of the smooth type, we normally discourage the use of this method.

The equations

The Newton-Raphson formula consists geometrically of extending the tangent line at a current point until it crosses zero, then setting the next guess to the abscissa of that zero-crossing. The mathematics behind this method is rather simple. Employing a Taylor expansion for x sufficiently close to the solution s, we have

$$f(s) = 0 = f(x) + (s-x)f'(x) + \frac{(s-x)^2}{2}f''(x) + \dots .$$

For small enough values of the function and for well-behaved functions, the terms beyond linear are unimportant, hence we obtain

$$f(x) + (s-x)f'(x) \approx 0,$$

yielding

$$s \approx x - \frac{f(x)}{f'(x)}.$$

Having in mind an iterative procedure, it is natural to start iterating with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Simple geometric interpretation

The above is Newton-Raphson's method. It has a simple geometric interpretation, namely x_{n+1} is the point where the tangent from $(x_n, f(x_n))$ crosses the x-axis. Close to the solution, Newton-Raphson converges fast to the desired result. However, if we are far from a root, where the higher-order terms in the series are important, the Newton-Raphson formula can give grossly inaccurate results. For instance, the initial guess for the root might be so far from the true root as to let the search interval include a local maximum or minimum of the function. If an iteration places a trial guess near such a local extremum, so that the first derivative nearly vanishes, then Newton-Raphson may fail totally

Extending to more than one variable

Newton's method can be generalized to systems of several non-linear equations and variables. Consider the case with two equations

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0, \end{aligned}$$

which we Taylor expand to obtain

$$\begin{aligned} 0 = f_1(x_1 + h_1, x_2 + h_2) = &\quad f_1(x_1, x_2) + h_1 \partial f_1/\partial x_1 + h_2 \partial f_1/\partial x_2 + \dots \\ 0 = f_2(x_1 + h_1, x_2 + h_2) = &\quad f_2(x_1, x_2) + h_1 \partial f_2/\partial x_1 + h_2 \partial f_2/\partial x_2 + \dots \end{aligned}.$$

Defining the Jacobian matrix $\mathbf{J}$ we have

$$\mathbf{J} = \begin{pmatrix} \partial f_1/\partial x_1 & \partial f_1/\partial x_2 \\ \partial f_2/\partial x_1 & \partial f_2/\partial x_2 \end{pmatrix},$$

we can rephrase Newton's method as

$$\begin{pmatrix} x_1^{n+1} \\ x_2^{n+1} \end{pmatrix} = \begin{pmatrix} x_1^n \\ x_2^n \end{pmatrix} + \begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix},$$

where we have defined

$$\begin{pmatrix} h_1^n \\ h_2^n \end{pmatrix} = -\mathbf{J}^{-1} \begin{pmatrix} f_1(x_1^n, x_2^n) \\ f_2(x_1^n, x_2^n) \end{pmatrix}.$$

We need thus to compute the inverse of the Jacobian matrix and it is to understand that difficulties may arise in case $\mathbf{J}$ is nearly singular.

It is rather straightforward to extend the above scheme to systems of more than two non-linear equations. In our case, the Jacobian matrix is given by the Hessian that represents the second derivative of cost function.

Steepest descent

The basic idea of gradient descent is that a function $F(\mathbf{x})$, $\mathbf{x} \equiv (x_1, \cdots, x_n)$, decreases fastest if one goes from $\mathbf{x}$ in the direction of the negative gradient $-\nabla F(\mathbf{x})$.

It can be shown that if

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k),$$

with $\gamma_k > 0$.

For γ_k small enough, then $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum.

More on Steepest descent

The previous observation is the basis of the method of steepest descent, which is also referred to as just gradient descent (GD). One starts with an initial guess $\mathbf{x}_0$ for a minimum of F and computes new approximations according to

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \gamma_k \nabla F(\mathbf{x}_k), \quad k \geq 0.$$

The parameter γ_k is often referred to as the step length or the learning rate within the context of Machine Learning.

The ideal

Ideally the sequence $\{\mathbf{x}_k\}_{k=0}$ converges to a global minimum of the function F. In general we do not know if we are in a global or local minimum. In the special case when F is a convex function, all local minima are also global minima, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement. However the method in this form has some severe limitations:

In machine learing we are often faced with non-convex high dimensional cost functions with many local minima. Since GD is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good intial guess. This also implies that the scheme is sensitive to the chosen initial condition.

Note that the gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$ which makes it expensive to compute numerically.

The sensitiveness of the gradient descent

The gradient descent method is sensitive to the choice of learning rate γ_k. This is due to the fact that we are only guaranteed that $F(\mathbf{x}_{k+1}) \leq F(\mathbf{x}_k)$ for sufficiently small γ_k. The problem is to determine an optimal learning rate. If the learning rate is chosen too small the method will take a long time to converge and if it is too large we can experience erratic behavior.

Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD), see below.

Convex functions

Ideally we want our cost/loss function to be convex(concave).

First we give the definition of a convex set: A set C in $\mathbb{R}^n$ is said to be convex if, for all x and y in C and all $t \in (0, 1)$, the point $(1 - t)x + ty$ also belongs to C. Geometrically this means that every point on the line segment connecting x and y is in C as discussed below.

The convex subsets of $\mathbb{R}$ are the intervals of $\mathbb{R}$. Examples of convex sets of $\mathbb{R}^2$ are the regular polygons (triangles, rectangles, pentagons, etc...).

Convex function

Convex function: Let $X \subset \mathbb{R}^n$ be a convex set. Assume that the function $f : X \to \mathbb{R}$ is continuous, then f is said to be convex if

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

for all $x_1, x_2 \in X$ and for all $t \in [0, 1]$. If $\leq$ is replaced with a strict inequaltiy in the definition, we demand $x_1 \neq x_2$ and $t \in (0, 1)$ then f is said to be strictly convex. For a single variable function, convexity means that if you draw a straight line connecting $f(x_1)$ and $f(x_2)$, the value of the function on the interval $[x_1, x_2]$ is always below the line as illustrated below.

Conditions on convex functions

In the following we state first and second-order conditions which ensures convexity of a function f. We write D_f to denote the domain of f, i.e the subset of R^n where f is defined. For more details and proofs we refer to: S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press.

First order condition. Suppose f is differentiable (i.e $\nabla f(x)$ is well defined for all x in the domain of f). Then f is convex if and only if D_f is a convex set and

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

holds for all $x, y \in D_f$. This condition means that for a convex function the first order Taylor expansion (right hand side above) at any point a global under estimator of the function. To convince yourself you can make a drawing of $f(x) = x^2 + 1$ and draw the tangent line to $f(x)$ and note that it is always below the graph.

Second order condition. Assume that f is twice differentiable, i.e the Hessian matrix exists at each point in D_f. Then f is convex if and only if D_f is a convex set and its Hessian is positive semi-definite for all $x \in D_f$. For a single-variable function this reduces to $f''(x) \geq 0$. Geometrically this means that f has nonnegative curvature everywhere.

This condition is particularly useful since it gives us an procedure for determining if the function under consideration is convex, apart from using the definition.

More on convex functions

The next result is of great importance to us and the reason why we are going on about convex functions. In machine learning we frequently have to minimize a loss/cost function in order to find the best parameters for the model we are considering.

Ideally we want the global minimum (for high-dimensional models it is hard to know if we have local or global minimum). However, if the cost/loss function is convex the following result provides invaluable information:

Any minimum is global for convex functions. Consider the problem of finding $x \in \mathbb{R}^n$ such that $f(x)$ is minimal, where f is convex and differentiable. Then, any point x^* that satisfies $\nabla f(x^*) = 0$ is a global minimum.

This result means that if we know that the cost/loss function is convex and we are able to find a minimum, we are guaranteed that it is a global minimum.

Some simple problems

1. Show that $f(x) = x^2$ is convex for $x \in \mathbb{R}$ using the definition of convexity. Hint: If you re-write the definition, f is convex if the following holds for all $x, y \in D_f$ and any $\lambda \in [0, 1]$ $\lambda f(x) + (1 - \lambda)f(y) - f(\lambda x + (1 - \lambda)y) \geq 0$.

2. Using the second order condition show that the following functions are convex on the specified domain.

   - $f(x) = e^x$ is convex for $x \in \mathbb{R}$.
   - $g(x) = -\ln(x)$ is convex for $x \in (0, \infty)$.

3. Let $f(x) = x^2$ and $g(x) = e^x$. Show that $f(g(x))$ and $g(f(x))$ is convex for $x \in \mathbb{R}$. Also show that if $f(x)$ is any convex function than $h(x) = e^{f(x)}$ is convex.

4. A norm is any function that satisfy the following properties

   - $f(\alpha x) = |\alpha| f(x)$ for all $\alpha \in \mathbb{R}$.
   - $f(x + y) \leq f(x) + f(y)$
   - $f(x) \leq 0$ for all $x \in \mathbb{R}^n$ with equality if and only if $x = 0$

Using the definition of convexity, try to show that a function satisfying the properties above is convex (the third condition is not needed to show this).

Standard steepest descent

Before we proceed, we would like to discuss the approach called the **standard Steepest descent** (different from the above steepest descent discussion), which again leads to us having to be able to compute a matrix. It belongs to the class of Conjugate Gradient methods (CG).

The success of the CG method for finding solutions of non-linear problems is based on the theory of conjugate gradients for linear systems of equations. It belongs to the class of iterative methods for solving problems from linear algebra of the type

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

In the iterative process we end up with a problem like

$$\boldsymbol{r} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x},$$

where $\boldsymbol{r}$ is the so-called residual or error in the iterative process.

When we have found the exact solution, $\boldsymbol{r} = 0$.

Gradient method

The residual is zero when we reach the minimum of the quadratic equation

$$P(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b},$$

with the constraint that the matrix $\boldsymbol{A}$ is positive definite and symmetric. This defines also the Hessian and we want it to be positive definite.

Steepest descent method

We denote the initial guess for $\boldsymbol{x}$ as $\boldsymbol{x}_0$. We can assume without loss of generality that

$$\boldsymbol{x}_0 = 0,$$

or consider the system

$$\boldsymbol{A}\boldsymbol{z} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0,$$

instead.

Steepest descent method

One can show that the solution $\boldsymbol{x}$ is also the unique minimizer of the quadratic form

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{x}, \quad \boldsymbol{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\boldsymbol{r}_1$ (see below for definition) to be the gradient of f at $\boldsymbol{x} = \boldsymbol{x}_0$, which equals

$$\boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b},$$

and $\boldsymbol{x}_0 = 0$ it is equal $-\boldsymbol{b}$.

Final expressions

We can compute the residual iteratively as

$$\boldsymbol{r}_{k+1} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{k+1},$$

which equals

$$\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_k + \alpha_k\boldsymbol{r}_k),$$

or

$$(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k) - \alpha_k \boldsymbol{A}\boldsymbol{r}_k,$$

which gives

$$\alpha_k = \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{r}_k^T \boldsymbol{A}\boldsymbol{r}_k}$$

leading to the iterative scheme

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \boldsymbol{r}_k,$$

Steepest descent example

```
import numpy as np
import numpy.linalg as la

import scipy.optimize as sopt

import matplotlib.pyplot as pt
from mpl_toolkits.mplot3d import axes3d

def f(x):
    return 0.5*x[0]**2 + 2.5*x[1]**2

def df(x):
    return np.array([x[0], 5*x[1]])

fig = pt.figure()
ax = fig.gca(projection="3d")

xmesh, ymesh = np.mgrid[-2:2:50j,-2:2:50j]
fmesh = f(np.array([xmesh, ymesh]))
ax.plot_surface(xmesh, ymesh, fmesh)
```

And then as countor plot

```
pt.axis("equal")
pt.contour(xmesh, ymesh, fmesh)
guesses = [np.array([2, 2./5])]
```

Find guesses

```
x = guesses[-1]
s = -df(x)
```

Run it!

```
def f1d(alpha):
    return f(x + alpha*s)

alpha_opt = sopt.golden(f1d)
next_guess = x + alpha_opt * s
guesses.append(next_guess)
print(next_guess)
```

What happened?

```
pt.axis("equal")
pt.contour(xmesh, ymesh, fmesh, 50)
it_array = np.array(guesses)
pt.plot(it_array.T[0], it_array.T[1], "x-")
```

Conjugate gradient method

In the CG method we define so-called conjugate directions and two vectors $\boldsymbol{s}$ and $\boldsymbol{t}$ are said to be conjugate if

$$\boldsymbol{s}^T\boldsymbol{A}\boldsymbol{t} = 0.$$

The philosophy of the CG method is to perform searches in various conjugate directions of our vectors $\boldsymbol{x}_i$ obeying the above criterion, namely

$$\boldsymbol{x}_i^T\boldsymbol{A}\boldsymbol{x}_j = 0.$$

Two vectors are conjugate if they are orthogonal with respect to this inner product. Being conjugate is a symmetric relation: if $\boldsymbol{s}$ is conjugate to $\boldsymbol{t}$, then $\boldsymbol{t}$ is conjugate to $\boldsymbol{s}$.

Conjugate gradient method

An example is given by the eigenvectors of the matrix

$$\boldsymbol{v}_i^T\boldsymbol{A}\boldsymbol{v}_j = \lambda\boldsymbol{v}_i^T\boldsymbol{v}_j,$$

which is zero unless $i = j$.

Conjugate gradient method

Assume now that we have a symmetric positive-definite matrix $\boldsymbol{A}$ of size $n \times n$. At each iteration $i + 1$ we obtain the conjugate direction of a vector

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i.$$

We assume that $\boldsymbol{p}_i$ is a sequence of n mutually conjugate directions. Then the $\boldsymbol{p}_i$ form a basis of R^n and we can expand the solution $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ in this basis, namely

$$\boldsymbol{x} = \sum_{i=1}^{n}\alpha_i\boldsymbol{p}_i.$$

Conjugate gradient method

The coefficients are given by

$$\mathbf{A}\mathbf{x} = \sum_{i=1}^{n}\alpha_i\mathbf{A}\mathbf{p}_i = \mathbf{b}.$$

Multiplying with $\boldsymbol{p}_k^T$ from the left gives

$$\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{x} = \sum_{i=1}^{n}\alpha_i\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_i = \boldsymbol{p}_k^T\boldsymbol{b},$$

and we can define the coefficients α_k as

$$\alpha_k = \frac{\boldsymbol{p}_k^T\boldsymbol{b}}{\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k}$$

Conjugate gradient method and iterations

If we choose the conjugate vectors $\boldsymbol{p}_k$ carefully, then we may not need all of them to obtain a good approximation to the solution $\boldsymbol{x}$. We want to regard the conjugate gradient method as an iterative method. This will us to solve systems where n is so large that the direct method would take too much time.

We denote the initial guess for $\boldsymbol{x}$ as $\boldsymbol{x}_0$. We can assume without loss of generality that

$$\boldsymbol{x}_0 = 0,$$

or consider the system

$$\boldsymbol{A}\boldsymbol{z} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0,$$

instead.

Conjugate gradient method

One can show that the solution $\boldsymbol{x}$ is also the unique minimizer of the quadratic form

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{x}, \quad \boldsymbol{x} \in \mathbf{R}^n.$$

This suggests taking the first basis vector $\boldsymbol{p}_1$ to be the gradient of f at $\boldsymbol{x} = \boldsymbol{x}_0$, which equals

$$\boldsymbol{A}\boldsymbol{x}_0 - \boldsymbol{b},$$

and $\boldsymbol{x}_0 = 0$ it is equal $-\boldsymbol{b}$. The other vectors in the basis will be conjugate to the gradient, hence the name conjugate gradient method.

Conjugate gradient method

Let $\boldsymbol{r}_k$ be the residual at the k-th step:

$$\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k.$$

Note that $\boldsymbol{r}_k$ is the negative gradient of f at $\boldsymbol{x} = \boldsymbol{x}_k$, so the gradient descent method would be to move in the direction $\boldsymbol{r}_k$. Here, we insist that the directions $\boldsymbol{p}_k$ are conjugate to each other, so we take the direction closest to the gradient $\boldsymbol{r}_k$ under the conjugacy constraint. This gives the following expression

$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_k - \frac{\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{r}_k}{\boldsymbol{p}_k^T\boldsymbol{A}\boldsymbol{p}_k}\boldsymbol{p}_k.$$

Conjugate gradient method

We can also compute the residual iteratively as

$$\boldsymbol{r}_{k+1} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{k+1},$$

which equals

$$\boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_k + \alpha_k\boldsymbol{p}_k),$$

or

$$(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k) - \alpha_k \boldsymbol{A}\boldsymbol{p}_k,$$

which gives

$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \boldsymbol{A}\boldsymbol{p}_k,$$

Revisiting our first homework

We will use linear regression as a case study for the gradient descent methods. Linear regression is a great test case for the gradient descent methods discussed in the lectures since it has several desirable properties such as:

1. An analytical solution (recall homework set 1).
2. The gradient can be computed analytically.
3. The cost function is convex which guarantees that gradient descent converges for small enough learning rates

We revisit an example similar to what we had in the first homework set. We had a function of the type

```
x = 2*np.random.rand(m,1)
y = 4+3*x+np.random.randn(m,1)
```

with $x_i \in [0, 1]$ is chosen randomly using a uniform distribution. Additionally we have a stochastic noise chosen according to a normal distribution $\mathcal{N}(\iota, \infty)$. The linear regression model is given by

$$h_\beta(x) = \boldsymbol{y} = \beta_0 + \beta_1 x,$$

such that

$$\boldsymbol{y}_i = \beta_0 + \beta_1 x_i.$$

Gradient descent example

Let $\mathbf{y} = (y_1, \cdots, y_n)^T$, $\mathbf{y} = (\boldsymbol{y}_1, \cdots, \boldsymbol{y}_n)^T$ and $\beta = (\beta_0, \beta_1)^T$

It is convenient to write $\mathbf{y} = X\beta$ where $X \in \mathbb{R}^{100\times 2}$ is the design matrix given by (we keep the intercept here)

$$X \equiv \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{100} \end{bmatrix}.$$

The cost/loss/risk function is given by (

$$C(\beta) = \frac{1}{n}||X\beta - \mathbf{y}||_2^2 = \frac{1}{n}\sum_{i=1}^{100}\left[(\beta_0 + \beta_1 x_i)^2 - 2y_i(\beta_0 + \beta_1 x_i) + y_i^2\right]$$

and we want to find β such that $C(\beta)$ is minimized.

The derivative of the cost/loss function

Computing $\partial C(\beta)/\partial\beta_0$ and $\partial C(\beta)/\partial\beta_1$ we can show that the gradient can be written as

$$\nabla_\beta C(\beta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100}(\beta_0+\beta_1 x_i - y_i) \\ \sum_{i=1}^{100}(x_i(\beta_0+\beta_1 x_i) - y_i x_i) \end{bmatrix} = \frac{2}{n}X^T(X\beta - \mathbf{y}),$$

where X is the design matrix defined above.

The Hessian matrix

The Hessian matrix of $C(\beta)$ is given by

$$\boldsymbol{H} \equiv \begin{bmatrix} \frac{\partial^2 C(\beta)}{\partial\beta_0^2} & \frac{\partial^2 C(\beta)}{\partial\beta_0\partial\beta_1} \\ \frac{\partial^2 C(\beta)}{\partial\beta_0\partial\beta_1} & \frac{\partial^2 C(\beta)}{\partial\beta_1^2} \end{bmatrix} = \frac{2}{n}X^T X.$$

This result implies that $C(\beta)$ is a convex function since the matrix $X^T X$ always is positive semi-definite.

Simple program

We can now write a program that minimizes $C(\beta)$ using the gradient descent method with a constant learning rate γ according to

$$\beta_{k+1} = \beta_k - \gamma\nabla_\beta C(\beta_k), \ k = 0, 1, \cdots$$

We can use the expression we computed for the gradient and let use a β_0 be chosen randomly and let $\gamma = 0.001$. Stop iterating when $||\nabla_\beta C(\beta_k)|| \leq \epsilon = 10^{-8}$. **Note that the code below does not include the latter stop criterion**.

And finally we can compare our solution for β with the analytic result given by $\beta = (X^T X)^{-1}X^T\mathbf{y}$.

Gradient Descent Example

Here our simple example

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys

# the number of datapoints
n = 100
x = 2*np.random.rand(n,1)
y = 4+3*x+np.random.randn(n,1)
```

```
X = np.c_[np.ones((n,1)), x]
# Hessian matrix
H = (2.0/n)* X.T @ X
# Get the eigenvalues
EigValues, EigVectors = np.linalg.eig(H)
print(EigValues)

beta_linreg = np.linalg.inv(X.T @ X) @ X.T @ y
print(beta_linreg)
beta = np.random.randn(2,1)

eta = 1.0/np.max(EigValues)
Niterations = 1000

for iter in range(Niterations):
    gradient = (2.0/n)*X.T @ (X @ beta-y)
    beta -= eta*gradient

print(beta)
xnew = np.array([[0],[2]])
xbnew = np.c_[np.ones((2,1)), xnew]
ypredict = xbnew.dot(beta)
ypredict2 = xbnew.dot(beta_linreg)
plt.plot(xnew, ypredict, "r-")
plt.plot(xnew, ypredict2, "b-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Gradient descent example')
plt.show()
```

And a corresponding example using scikit-learn

```
# Importing various packages
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor

n = 100
x = 2*np.random.rand(n,1)
y = 4+3*x+np.random.randn(n,1)

X = np.c_[np.ones((n,1)), x]
beta_linreg = np.linalg.inv(X.T @ X) @ (X.T @ y)
print(beta_linreg)
sgdreg = SGDRegressor(max_iter = 50, penalty=None, eta0=0.1)
sgdreg.fit(x,y.ravel())
print(sgdreg.intercept_, sgdreg.coef_)
```

Gradient descent and Ridge

We have also discussed Ridge regression where the loss function contains a regularized term given by the L_2 norm of β,

$$C_{\text{ridge}}(\beta) = \frac{1}{n}||X\beta - \mathbf{y}||^2 + \lambda||\beta||^2, \ \lambda \geq 0.$$

In order to minimize $C_{\rm ridge}(\beta)$ using GD we only have adjust the gradient as follows

$$\nabla_\beta C_{\rm ridge}(\beta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100} (\beta_0+\beta_1 x_i - y_i) \\ \sum_{i=1}^{100} (x_i(\beta_0+\beta_1 x_i) - y_i x_i) \end{bmatrix} + 2\lambda \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = 2(X^T(X\beta - \mathbf{y}) + \lambda\beta).$$

We can easily extend our program to minimize $C_{\rm ridge}(\beta)$ using gradient descent and compare with the analytical solution given by

$$\beta_{\rm ridge} = \left(X^T X + \lambda I_{2\times 2}\right)^{-1} X^T \mathbf{y}.$$

Program example for gradient descent with Ridge Regression

```
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import sys

# the number of datapoints
n = 100
x = 2*np.random.rand(n,1)
y = 4+3*x+np.random.randn(n,1)

X = np.c_[np.ones((n,1)), x]
XT_X = X.T @ X

#Ridge parameter lambda
lmbda  = 0.001
Id = lmbda* np.eye(XT_X.shape[0])

beta_linreg = np.linalg.inv(XT_X+Id) @ X.T @ y
print(beta_linreg)
# Start plain gradient descent
beta = np.random.randn(2,1)

eta = 0.1
Niterations = 100

for iter in range(Niterations):
    gradients = 2.0/n*X.T @ (X @ (beta)-y)+2*lmbda*beta
    beta -= eta*gradients

print(beta)
ypredict = X @ beta
ypredict2 = X @ beta_linreg
plt.plot(x, ypredict, "r-")
plt.plot(x, ypredict2, "b-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Gradient descent example for Ridge')
plt.show()
```

Using gradient descent methods, limitations

- **Gradient descent (GD) finds local minima of our function**. Since the GD algorithm is deterministic, if it converges, it will converge to a local minimum of our cost/loss/risk function. Because in ML we are often dealing with extremely rugged landscapes with many local minima, this can lead to poor performance.

- **GD is sensitive to initial conditions**. One consequence of the local nature of GD is that initial conditions matter. Depending on where one starts, one will end up at a different local minima. Therefore, it is very important to think about how one initializes the training process. This is true for GD as well as more complicated variants of GD.

- **Gradients are computationally expensive to calculate for large datasets**. In many cases in statistics and ML, the cost/loss/risk function is a sum of terms, with one term for each data point. For example, in linear regression, $E \propto \sum_{i=1}^{n} (y_i - \mathbf{w}^T \cdot \mathbf{x}_i)^2$; for logistic regression, the square error is replaced by the cross entropy. To calculate the gradient we have to sum over *all* n data points. Doing this at every GD step becomes extremely computationally expensive. An ingenious solution to this, is to calculate the gradients using small subsets of the data called "mini batches". This has the added benefit of introducing stochasticity into our algorithm.

- **GD is very sensitive to choices of learning rates**. GD is extremely sensitive to the choice of learning rates. If the learning rate is very small, the training process take an extremely long time. For larger learning rates, GD can diverge and give poor results. Furthermore, depending on what the local landscape looks like, we have to modify the learning rates to ensure convergence. Ideally, we would *adaptively* choose the learning rates to match the landscape.

- **GD treats all directions in parameter space uniformly.** Another major drawback of GD is that unlike Newton's method, the learning rate for GD is the same in all directions in parameter space. For this reason, the maximum learning rate is set by the behavior of the steepest direction and this can significantly slow down training. Ideally, we would like to take large steps in flat directions and small steps in steep directions. Since we are exploring rugged landscapes where curvatures change, this requires us to keep track of not only the gradient but second derivatives. The ideal scenario would be to calculate the Hessian but this proves to be too computationally expensive.

- GD can take exponential time to escape saddle points, even with random initialization. As we mentioned, GD is extremely sensitive to initial condition since it determines the particular local minimum GD would eventually reach. However, even with a good initialization scheme, through

the introduction of randomness, GD can still take exponential time to escape saddle points.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The underlying idea of SGD comes from the observation that the cost function, which we want to minimize, can almost always be written as a sum over n data points $\{\mathbf{x}_i\}_{i=1}^n$,

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta).$$

Computation of gradients

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

Stochasticity/randomness is introduced by only taking the gradient on a subset of the data called minibatches. If there are n data points and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

SGD example

As an example, suppose we have 10 data points $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two data points. In particular we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if you choose $M = 1$ you have only a single batch with all data points and on the other extreme, you may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$.

The idea is now to approximate the gradient by replacing the sum over all data points with a sum over the data points in one the minibatches picked at random in each gradient descent step

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \rightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta).$$

The gradient step

Thus a gradient descent step now looks like

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

where k is picked at random with equal probability from $[1, n/M]$. An iteration over the number of minibathces (n/M) is commonly referred to as an epoch. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches, as exemplified in the code below.

Simple example code

```
import numpy as np

n = 100 #100 datapoints
M = 5   #size of each minibatch
m = int(n/M) #number of minibatches
n_epochs = 10 #number of epochs

j = 0
for epoch in range(1,n_epochs+1):
    for i in range(m):
        k = np.random.randint(m) #Pick the k-th minibatch at random
        #Compute the gradient using the data in minibatch Bk
        #Compute new suggestion for
        j += 1
```

Taking the gradient only on a subset of the data has two important benefits. First, it introduces randomness which decreases the chance that our opmization scheme gets stuck in a local minima. Second, if the size of the minibatches are small relative to the number of datapoints ($M < n$), the computation of the gradient is much cheaper since we sum over the datapoints in the $k - th$ minibatch and not all n datapoints.

When do we stop?

A natural question is when do we stop the search for a new minimum? One possibility is to compute the full gradient after a given number of epochs and check if the norm of the gradient is smaller than some threshold and stop if true. However, the condition that the gradient is zero is valid also for local minima, so this would only tell us that we are close to a local/global minimum. However, we could also evaluate the cost function at this point, store the result and continue the search. If the test kicks in at a later stage we can compare the values of the cost function and keep the β that gave the lowest value.

Slightly different approach

Another approach is to let the step length γ_j depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all.

As an example, let $e = 0, 1, 2, 3, \cdots$ denote the current epoch and let $t_0, t_1 > 0$ be two fixed numbers. Furthermore, let $t = e \cdot m + i$ where m is the number of minibatches and $i = 0, \cdots, m - 1$. Then the function

$$\gamma_j(t; t_0, t_1) = \frac{t_0}{t + t_1}$$

goes to zero as the number of epochs gets large. I.e. we start with a step length $\gamma_j(0;t_0,t_1) = t_0/t_1$ which decays in *time* t.

In this way we can fix the number of epochs, compute β and evaluate the cost function at the end. Repeating the computation will give a different result since the scheme is random by design. Then we pick the final β that gives the lowest value of the cost function.

```
import numpy as np

def step_length(t,t0,t1):
    return t0/(t+t1)

n = 100 #100 datapoints
M = 5   #size of each minibatch
m = int(n/M) #number of minibatches
n_epochs = 500 #number of epochs
t0 = 1.0
t1 = 10

gamma_j = t0/t1
j = 0
for epoch in range(1,n_epochs+1):
    for i in range(m):
        k = np.random.randint(m) #Pick the k-th minibatch at random
        #Compute the gradient using the data in minibatch Bk
        #Compute new suggestion for beta
        t = epoch*m+i
        gamma_j = step_length(t,t0,t1)
        j += 1

print("gamma_j after %d epochs: %g" % (n_epochs,gamma_j))
```

Program for stochastic gradient

```
# Importing various packages
from math import exp, sqrt
from random import random, seed
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor

m = 100
x = 2*np.random.rand(m,1)
y = 4+3*x+np.random.randn(m,1)

X = np.c_[np.ones((m,1)), x]
theta_linreg = np.linalg.inv(X.T @ X) @ (X.T @ y)
print("Own inversion")
print(theta_linreg)
sgdreg = SGDRegressor(max_iter = 50, penalty=None, eta0=0.1)
sgdreg.fit(x,y.ravel())
print("sgdreg from scikit")
print(sgdreg.intercept_, sgdreg.coef_)


theta = np.random.randn(2,1)
eta = 0.1
Niterations = 1000
```

```
for iter in range(Niterations):
    gradients = 2.0/m*X.T @ ((X @ theta)-y)
    theta -= eta*gradients
print("theta from own gd")
print(theta)

xnew = np.array([[0],[2]])
Xnew = np.c_[np.ones((2,1)), xnew]
ypredict = Xnew.dot(theta)
ypredict2 = Xnew.dot(theta_linreg)


n_epochs = 50
t0, t1 = 5, 50
def learning_schedule(t):
    return t0/(t+t1)

theta = np.random.randn(2,1)

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T @ ((xi @ theta)-yi)
        eta = learning_schedule(epoch*m+i)
        theta = theta - eta*gradients
print("theta from own sdg")
print(theta)

plt.plot(xnew, ypredict, "r-")
plt.plot(xnew, ypredict2, "b-")
plt.plot(x, y ,'ro')
plt.axis([0,2.0,0, 15.0])
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'Random numbers ')
plt.show()
```

Challenge: try to write a similar code for a Logistic Regression case.

Momentum based GD

The stochastic gradient descent (SGD) is almost always used with a *momentum* or inertia term that serves as a memory of the direction we are moving in parameter space. This is typically implemented as follows

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t, \end{aligned} \tag{2}$$

where we have introduced a momentum parameter γ, with $0 \leq \gamma \leq 1$, and for brevity we dropped the explicit notation to indicate the gradient is to be taken over a different mini-batch at each step. We call this algorithm gradient descent

with momentum (GDM). From these equations, it is clear that $\mathbf{v}_t$ is a running average of recently encountered gradients and $(1-\gamma)^{-1}$ sets the characteristic time scale for the memory used in the averaging procedure. Consistent with this, when $\gamma = 0$, this just reduces down to ordinary SGD as discussed earlier. An equivalent way of writing the updates is

$$\Delta\boldsymbol{\theta}_{t+1} = \gamma\Delta\boldsymbol{\theta}_t - \ \eta_t \nabla_\theta E(\boldsymbol{\theta}_t),$$

where we have defined $\Delta\boldsymbol{\theta}_t = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$.

More on momentum based approaches

Let us try to get more intuition from these equations. It is helpful to consider a simple physical analogy with a particle of mass m moving in a viscous medium with drag coefficient μ and potential $E(\mathbf{w})$. If we denote the particle's position by $\mathbf{w}$, then its motion is described by

$$m\frac{d^2\mathbf{w}}{dt^2} + \mu\frac{d\mathbf{w}}{dt} = -\nabla_w E(\mathbf{w}).$$

We can discretize this equation in the usual way to get

$$m\frac{\mathbf{w}_{t+\Delta t} - 2\mathbf{w}_t + \mathbf{w}_{t-\Delta t}}{(\Delta t)^2} + \mu\frac{\mathbf{w}_{t+\Delta t} - \mathbf{w}_t}{\Delta t} = -\nabla_w E(\mathbf{w}).$$

Rearranging this equation, we can rewrite this as

$$\Delta\mathbf{w}_{t+\Delta t} = -\frac{(\Delta t)^2}{m + \mu\Delta t}\nabla_w E(\mathbf{w}) + \frac{m}{m+\mu\Delta t}\Delta\mathbf{w}_t.$$

Momentum parameter

Notice that this equation is identical to previous one if we identify the position of the particle, $\mathbf{w}$, with the parameters $\boldsymbol{\theta}$. This allows us to identify the momentum parameter and learning rate with the mass of the particle and the viscous drag as:

$$\gamma = \frac{m}{m+\mu\Delta t}, \qquad \eta = \frac{(\Delta t)^2}{m+\mu\Delta t}.$$

Thus, as the name suggests, the momentum parameter is proportional to the mass of the particle and effectively provides inertia. Furthermore, in the large viscosity/small learning rate limit, our memory time scales as $(1-\gamma)^{-1} \approx m/(\mu\Delta t)$.

Why is momentum useful? SGD momentum helps the gradient descent algorithm gain speed in directions with persistent but small gradients even in the presence of stochasticity, while suppressing oscillations in high-curvature directions. This becomes especially important in situations where the landscape is shallow and flat in some directions and narrow and steep in others. It has

been argued that first-order methods (with appropriate initial conditions) can perform comparable to more expensive second order methods, especially in the context of complex deep learning models.

These beneficial properties of momentum can sometimes become even more pronounced by using a slight modification of the classical momentum algorithm called Nesterov Accelerated Gradient (NAG).

In the NAG algorithm, rather than calculating the gradient at the current parameters, $\nabla_\theta E(\boldsymbol{\theta}_t)$, one calculates the gradient at the expected value of the parameters given our current momentum, $\nabla_\theta E(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1})$. This yields the NAG update rule

$$\begin{aligned} \mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t + \gamma \mathbf{v}_{t-1}) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \mathbf{v}_t. \end{aligned} \tag{3}$$

One of the major advantages of NAG is that it allows for the use of a larger learning rate than GDM for the same choice of γ.

Second moment of the gradient

In stochastic gradient descent, with and without momentum, we still have to specify a schedule for tuning the learning rates η_t as a function of time. As discussed in the context of Newton's method, this presents a number of dilemmas. The learning rate is limited by the steepest direction which can change depending on the current position in the landscape. To circumvent this problem, ideally our algorithm would keep track of curvature and take large steps in shallow, flat directions and small steps in steep, narrow directions. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. However, this is very computationally expensive for extremely large models. Ideally, we would like to be able to adaptively change the step size to match the landscape without paying the steep computational price of calculating or approximating Hessians.

Recently, a number of methods have been introduced that accomplish this by tracking not only the gradient, but also the second moment of the gradient. These methods include AdaGrad, AdaDelta, RMS-Prop, and ADAM.

RMS prop

In RMS prop, in addition to keeping a running average of the first moment of the gradient, we also keep track of the second moment denoted by $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. The update rule for RMS prop is given by

$$\begin{aligned} \mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \\ \mathbf{s}_t &= \beta \mathbf{s}_{t-1} + (1-\beta)\mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}, \end{aligned} \tag{4}$$

where β controls the averaging time of the second moment and is typically taken to be about $\beta = 0.9$, η_t is a learning rate typically chosen to be 10^{-3}, and $\epsilon \sim 10^{-8}$ is a small regularization constant to prevent divergences. Multiplication and division by vectors is understood as an element-wise operation. It is clear from this formula that the learning rate is reduced in directions where the norm of the gradient is consistently large. This greatly speeds up the convergence by allowing us to use a larger learning rate for flat directions.

ADAM optimizer

A related algorithm is the ADAM optimizer. In ADAM, we keep a running average of both the first and second moment of the gradient and use this information to adaptively change the learning rate for different parameters. In addition to keeping a running average of the first and second moments of the gradient (i.e. $\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t]$ and $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$, respectively), ADAM performs an additional bias correction to account for the fact that we are estimating the first two moments of the gradient using a running average (denoted by the hats in the update rule below). The update rule for ADAM is given by (where multiplication and division are once again understood to be element-wise operations below)

$$\begin{aligned}
\mathbf{g}_t &= \nabla_\theta E(\boldsymbol{\theta}) \qquad (5)\\
\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
\mathbf{m}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t} \\
\mathbf{s}_t &= \frac{\mathbf{s}_t}{1-\beta_2^t} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta_t \frac{\mathbf{m}_t}{\sqrt{\mathbf{s}_t}+\epsilon}, \qquad (6)
\end{aligned}$$

where β_1 and β_2 set the memory lifetime of the first and second moment and are typically taken to be 0.9 and 0.99 respectively, and η and ϵ are identical to RMSprop.

Like in RMSprop, the effective step size of a parameter depends on the magnitude of its gradient squared. To understand this better, let us rewrite this expression in terms of the variance $\boldsymbol{\sigma}_t^2 = \mathbf{s}_t - (\mathbf{m}_t)^2$. Consider a single parameter θ_t. The update rule for this parameter is given by

$$\Delta\theta_{t+1} = -\eta_t \frac{\boldsymbol{m}_t}{\sqrt{\sigma_t^2 + m_t^2} + \epsilon}.$$

Practical tips

- **Randomize the data when making mini-batches**. It is always important to randomly shuffle the data when forming mini-batches. Otherwise, the gradient descent method can fit spurious correlations resulting from the order in which data is presented.

- **Transform your inputs**. Learning becomes difficult when our landscape has a mixture of steep and flat directions. One simple trick for minimizing these situations is to standardize the data by subtracting the mean and normalizing the variance of input variables. Whenever possible, also decorrelate the inputs. To understand why this is helpful, consider the case of linear regression. It is easy to show that for the squared error cost function, the Hessian of the cost function is just the correlation matrix between the inputs. Thus, by standardizing the inputs, we are ensuring that the landscape looks homogeneous in all directions in parameter space. Since most deep networks can be viewed as linear transformations followed by a non-linearity at each layer, we expect this intuition to hold beyond the linear case.

- **Monitor the out-of-sample performance.** Always monitor the performance of your model on a validation set (a small portion of the training data that is held out of the training process to serve as a proxy for the test set. If the validation error starts increasing, then the model is beginning to overfit. Terminate the learning process. This *early stopping* significantly improves performance in many settings.

- **Adaptive optimization methods don't always have good generalization.** Recent studies have shown that adaptive methods such as ADAM, RMSPorp, and AdaGrad tend to have poor generalization compared to SGD or SGD with momentum, particularly in the high-dimensional limit (i.e. the number of parameters exceeds the number of data points). Although it is not clear at this stage why these methods perform so well in training deep neural networks, simpler procedures like properly-tuned SGD may work as well or better in these applications.

Geron's text, see chapter 11, has several interesting discussions.

Automatic differentiation

Automatic differentiation (AD), also called algorithmic differentiation or computational differentiation,is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be

computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

Automatic differentiation is neither:

- Symbolic differentiation, nor
- Numerical differentiation (the method of finite differences).

Symbolic differentiation can lead to inefficient code and faces the difficulty of converting a computer program into a single expression, while numerical differentiation can introduce round-off errors in the discretization process and cancellation

Python has tools for so-called **automatic differentiation**. Consider the following example

$$f(x) = \sin\left(2\pi x + x^2\right)$$

which has the following derivative

$$f'(x) = \cos\left(2\pi x + x^2\right)(2\pi + 2x)$$

Using **autograd** we have

```
import autograd.numpy as np

# To do elementwise differentiation:
from autograd import elementwise_grad as egrad

# To plot:
import matplotlib.pyplot as plt


def f(x):
    return np.sin(2*np.pi*x + x**2)

def f_grad_analytic(x):
    return np.cos(2*np.pi*x + x**2)*(2*np.pi + 2*x)

# Do the comparison:
x = np.linspace(0,1,1000)

f_grad = egrad(f)

computed = f_grad(x)
analytic = f_grad_analytic(x)

plt.title('Derivative computed from Autograd compared with the analytical derivative')
plt.plot(x,computed,label='autograd')
plt.plot(x,analytic,label='analytic')

plt.xlabel('x')
plt.ylabel('y')
plt.legend()

plt.show()

print("The max absolute difference is: %g"%(np.max(np.abs(computed - analytic))))
```

Using autograd

Here we experiment with what kind of functions Autograd is capable of finding the gradient of. The following Python functions are just meant to illustrate what Autograd can do, but please feel free to experiment with other, possibly more complicated, functions as well.

```
import autograd.numpy as np
from autograd import grad

def f1(x):
    return x**3 + 1

f1_grad = grad(f1)

# Remember to send in float as argument to the computed gradient from Autograd!
a = 1.0

# See the evaluated gradient at a using autograd:
print("The gradient of f1 evaluated at a = %g using autograd is: %g"%(a,f1_grad(a)))

# Compare with the analytical derivative, that is f1'(x) = 3*x**2
grad_analytical = 3*a**2
print("The gradient of f1 evaluated at a = %g by finding the analytic expression is: %g"%(a,grad_a
```

Autograd with more complicated functions

To differentiate with respect to two (or more) arguments of a Python function, Autograd need to know at which variable the function if being differentiated with respect to.

```
import autograd.numpy as np
from autograd import grad
def f2(x1,x2):
    return 3*x1**3 + x2*(x1 - 5) + 1

# By sending the argument 0, Autograd will compute the derivative w.r.t the first variable, in th
f2_grad_x1 = grad(f2,0)

# ... and differentiate w.r.t x2 by sending 1 as an additional arugment to grad
f2_grad_x2 = grad(f2,1)

x1 = 1.0
x2 = 3.0

print("Evaluating at x1 = %g, x2 = %g"%(x1,x2))
print("-"*30)

# Compare with the analytical derivatives:

# Derivative of f2 w.r.t x1 is: 9*x1**2 + x2:
f2_grad_x1_analytical = 9*x1**2 + x2

# Derivative of f2 w.r.t x2 is: x1 - 5:
f2_grad_x2_analytical = x1 - 5

# See the evaluated derivations:
print("The derivative of f2 w.r.t x1: %g"%( f2_grad_x1(x1,x2) ))
```

```
print("The analytical derivative of f2 w.r.t x1: %g"%( f2_grad_x1(x1,x2) ))

print()

print("The derivative of f2 w.r.t x2: %g"%( f2_grad_x2(x1,x2) ))
print("The analytical derivative of f2 w.r.t x2: %g"%( f2_grad_x2(x1,x2) ))
```

Note that the grad function will not produce the true gradient of the function. The true gradient of a function with two or more variables will produce a vector, where each element is the function differentiated w.r.t a variable.

More complicated functions using the elements of their arguments directly

```
import autograd.numpy as np
from autograd import grad
def f3(x): # Assumes x is an array of length 5 or higher
    return 2*x[0] + 3*x[1] + 5*x[2] + 7*x[3] + 11*x[4]**2

f3_grad = grad(f3)

x = np.linspace(0,4,5)

# Print the computed gradient:
print("The computed gradient of f3 is: ", f3_grad(x))

# The analytical gradient is: (2, 3, 5, 7, 22*x[4])
f3_grad_analytical = np.array([2, 3, 5, 7, 22*x[4]])

# Print the analytical gradient:
print("The analytical gradient of f3 is: ", f3_grad_analytical)
```

Note that in this case, when sending an array as input argument, the output from Autograd is another array. This is the true gradient of the function, as opposed to the function in the previous example. By using arrays to represent the variables, the output from Autograd might be easier to work with, as the output is closer to what one could expect form a gradient-evaluting function.

Functions using mathematical functions from Numpy

```
import autograd.numpy as np
from autograd import grad
def f4(x):
    return np.sqrt(1+x**2) + np.exp(x) + np.sin(2*np.pi*x)

f4_grad = grad(f4)

x = 2.7

# Print the computed derivative:
print("The computed derivative of f4 at x = %g is: %g"%(x,f4_grad(x)))

# The analytical derivative is: x/sqrt(1 + x**2) + exp(x) + cos(2*pi*x)*2*pi
f4_grad_analytical = x/np.sqrt(1 + x**2) + np.exp(x) + np.cos(2*np.pi*x)*2*np.pi

# Print the analytical gradient:
print("The analytical gradient of f4 at x = %g is: %g"%(x,f4_grad_analytical))
```

More autograd

```
import autograd.numpy as np
from autograd import grad
def f5(x):
    if x >= 0:
        return x**2
    else:
        return -3*x + 1

f5_grad = grad(f5)

x = 2.7

# Print the computed derivative:
print("The computed derivative of f5 at x = %g is: %g"%(x,f5_grad(x)))
```

And with loops

```
import autograd.numpy as np
from autograd import grad
def f6_for(x):
    val = 0
    for i in range(10):
        val = val + x**i
    return val

def f6_while(x):
    val = 0
    i = 0
    while i < 10:
        val = val + x**i
        i = i + 1
    return val

f6_for_grad = grad(f6_for)
f6_while_grad = grad(f6_while)

x = 0.5

# Print the computed derivaties of f6_for and f6_while
print("The computed derivative of f6_for at x = %g is: %g"%(x,f6_for_grad(x)))
print("The computed derivative of f6_while at x = %g is: %g"%(x,f6_while_grad(x)))

import autograd.numpy as np
from autograd import grad
# Both of the functions are implementation of the sum: sum(x**i) for i = 0, ..., 9
# The analytical derivative is: sum(i*x**(i-1))
f6_grad_analytical = 0
for i in range(10):
    f6_grad_analytical += i*x**(i-1)

print("The analytical derivative of f6 at x = %g is: %g"%(x,f6_grad_analytical))
```

Using recursion

```
import autograd.numpy as np
from autograd import grad
```

```
def f7(n): # Assume that n is an integer
    if n == 1 or n == 0:
        return 1
    else:
        return n*f7(n-1)

f7_grad = grad(f7)

n = 2.0

print("The computed derivative of f7 at n = %d is: %g"%(n,f7_grad(n)))

# The function f7 is an implementation of the factorial of n.
# By using the product rule, one can find that the derivative is:

f7_grad_analytical = 0
for i in range(int(n)-1):
    tmp = 1
    for k in range(int(n)-1):
        if k != i:
            tmp *= (n - k)
    f7_grad_analytical += tmp

print("The analytical derivative of f7 at n = %d is: %g"%(n,f7_grad_analytical))
```

Note that if n is equal to zero or one, Autograd will give an error message. This message appears when the output is independent on input.

Unsupported functions

Autograd supports many features. However, there are some functions that is not supported (yet) by Autograd.

Assigning a value to the variable being differentiated with respect to

```
import autograd.numpy as np
from autograd import grad
def f8(x): # Assume x is an array
    x[2] = 3
    return x*2

f8_grad = grad(f8)

x = 8.4

print("The derivative of f8 is:",f8_grad(x))
```

Here, Autograd tells us that an 'ArrayBox' does not support item assignment. The item assignment is done when the program tries to assign x[2] to the value 3. However, Autograd has implemented the computation of the derivative such that this assignment is not possible.

The syntax a.dot(b) when finding the dot product

```
import autograd.numpy as np
from autograd import grad
def f9(a): # Assume a is an array with 2 elements
```

```
    b = np.array([1.0,2.0])
    return a.dot(b)

f9_grad = grad(f9)

x = np.array([1.0,0.0])

print("The derivative of f9 is:",f9_grad(x))
```

Here we are told that the 'dot' function does not belong to Autograd's version of a Numpy array. To overcome this, an alternative syntax which also computed the dot product can be used:

```
import autograd.numpy as np
from autograd import grad
def f9_alternative(x): # Assume a is an array with 2 elements
    b = np.array([1.0,2.0])
    return np.dot(x,b) # The same as x_1*b_1 + x_2*b_2

f9_alternative_grad = grad(f9_alternative)

x = np.array([3.0,0.0])

print("The gradient of f9 is:",f9_alternative_grad(x))

# The analytical gradient of the dot product of vectors x and b with two elements (x_1,x_2) and (
# w.r.t x is (b_1, b_2).
```

Recommended to avoid

The documentation recommends to avoid inplace operations such as

```
a += b
a -= b
a*= b
a /=b
```

Neural networks

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable.

Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain,

which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{7}$$

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons.

Conceptually, it is helpful to divide neural networks into four categories:

1. general purpose neural networks for supervised learning,
2. neural networks designed specifically for image processing, the most prominent example of this class being Convolutional Neural Networks (CNNs),
3. neural networks for sequential data such as Recurrent Neural Networks (RNNs), and
4. neural networks for unsupervised learning such as Deep Boltzmann Machines.

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs, topological phases, and even non-equilibrium many-body localization. Representing quantum states as DNNs quantum state tomography are among some of the impressive achievements to reveal the potential of DNNs to facilitate the study of quantum systems.

In quantum information theory, it has been shown that one can perform gate decompositions with the help of neural.

The applications are not limited to the natural sciences. There is a plethora of applications in essentially all disciplines, from the humanities to life science and medicine.

Neural network types

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons, or nodes or units. We will refer to these interchangeably as units or nodes, and sometimes as neurons.

It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical

functions between layers. A wide variety of different ANNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods we discussed earlier.

Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of ANNs that were devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable (figure to come). We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

Convolutional Neural Network

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation. (figure to come)

Convolutional neural networks emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

Recurrent neural networks

So far we have only mentioned ANNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where

each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

Other types of networks

There are many other kinds of ANNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer ("linear" here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a separate type of NN due the unusual activation functions.

Multilayer perceptrons

One uses often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs).

Why multilayer perceptrons?

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**.

Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

Mathematical model

The output y is produced via the activation function f

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z),$$

This function receives x_i as inputs. Here the activation $z = (\sum_{i=1}^{n} w_i x_i + b_i)$. In an FFNN of such neurons, the *inputs* x_i are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

Mathematical model

First, for each node i in the first hidden layer, we calculate a weighted sum z_i^1 of the input coordinates x_j,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{8}$$

Here b_i is the so-called bias which is normally needed in case of zero activation weights or inputs. How to fix the biases and the weights will be discussed below. The value of z_i^1 is the argument to the activation function f_i of each node i, The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{9}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f. In general, we could assume in the more general case that different layers have different activation functions. In this case we would identify these functions with a superscript l for the l-th layer,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{10}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

Mathematical model

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{11}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{12}$$

where we have substituted y_k^1 with the inputs x_k. Finally, the ANN output reads

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{13}$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{14}$$

Mathematical model

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right] \tag{15}$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n.

Mathematical model

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\hat{x} \in \mathbb{R}^n \to \hat{y} \in \mathbb{R}^m$.

Furthermore, the flexibility and universality of an MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{16}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

Matrix-vector notation. We can introduce a more convenient notation for the activations in an A NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\hat{b}_l$ and $\hat{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that W_l is an $N_{l-1} \times N_l$ matrix, while $\hat{b}_l$ and $\hat{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication,

and we can write the equation for the activations of hidden layer 2 (assuming three nodes for simplicity) as

$$\hat{y}_2 = f_2(\mathrm{W}_2\hat{y}_1 + \hat{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \quad (17)$$

Matrix-vector notation and activation. The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \quad (18)$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l\hat{y}_{l-1}$ we move forward one layer.

Activation functions. A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

Activation functions, Logistic and Hyperbolic ones. The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1 + e^{-x}},$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x)$$

Relevance. The *sigmoid* function are more biologically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown that the hyperbolic tangent performs better than the sigmoid for training MLPs. has become the most popular for *deep neural networks*

```
"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative scale.
The value z is the weighted sum of parameters involved in the learning algorithm."""

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')

plt.show()

"""Step Function"""
z = numpy.arange(-5, 5, .02)
step_fn = numpy.vectorize(lambda z: 1.0 if z >= 0.0 else 0.0)
step = step_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, step)
ax.set_ylim([-0.5, 1.5])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('step function')

plt.show()

"""Sine Function"""
z = numpy.arange(-2*mt.pi, 2*mt.pi, 0.1)
t = numpy.sin(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, t)
ax.set_ylim([-1.0, 1.0])
ax.set_xlim([-2*mt.pi,2*mt.pi])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sine function')

plt.show()
```

```
    """Plots a graph of the squashing function used by a rectified linear
    unit"""
    z = numpy.arange(-2, 2, .1)
    zero = numpy.zeros(len(z))
    y = numpy.max([zero, z], axis=0)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(z, y)
    ax.set_ylim([-2.0, 2.0])
    ax.set_xlim([-2.0, 2.0])
    ax.grid(True)
    ax.set_xlabel('z')
    ax.set_title('Rectified linear unit')

    plt.show()
```

The multilayer perceptron (MLP)

The multilayer perceptron is a very popular, and easy to implement approach, to deep learning. It consists of

1. A neural network with one or more layers of nodes between the input and the output nodes.

2. The multilayer network structure, or architecture, or topology, consists of an input layer, one or more hidden layers, and one output layer.

3. The input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

As a convention it is normal to call a network with one layer of input units, one layer of hidden units and one layer of output units as a two-layer network. A network with two layers of hidden units is called a three-layer network etc etc.

For an MLP network there is no direct connection between the output nodes/neurons/units and the input nodes/neurons/units. Hereafter we will call the various entities of a layer for nodes. There are also no connections within a single layer.

The number of input nodes does not need to equal the number of output nodes. This applies also to the hidden layers. Each layer may have its own number of nodes and activation functions.

The hidden layers have their name from the fact that they are not linked to observables and as we will see below when we define the so-called activation $\hat{z}$, we can think of this as a basis expansion of the original inputs $\hat{x}$. The difference however between neural networks and say linear regression is that now these basis functions (which will correspond to the weights in the network) are learned from data. This results in an important difference between neural networks and deep learning approaches on one side and methods like logistic regression or linear regression and their modifications on the other side.

From one to many layers, the universal approximation theorem

A neural network with only one layer, what we called the simple perceptron, is best suited if we have a standard binary model with clear (linear) boundaries between the outcomes. As such it could equally well be replaced by standard linear regression or logistic regression. Networks with one or more hidden layers approximate systems with more complex boundaries.

As stated earlier, an important theorem in studies of neural networks, restated without proof here, is the universal approximation theorem.

It states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. It is the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators.