Introduction to deep learning and neural networks; building your own neural network code

Morten Hjorth-Jensen[1]

Department of Physics, University of Oslo[1]

October 10, 2024

Material for the lecture October 10, 2024

1. Neural Networks, setting up the basic steps, from the simple perceptron model to the multi-layer perceptron model.
2. Building our own Feed-forward Neural Network

Readings and Videos:

1. See books below
2. Neural Networks demystified at https://www.youtube.com/watch?v=bxe2T-V8XRs&list=PLiaHhY2iBX9hdHaRr6b7XevZtgZRa1PoU&ab_channel=WelchLabs
3. Building Neural Networks from scratch at https://www.youtube.com/watch?v=Wo5dMEP_BbI&list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3&ab_channel=sentdex
4. Video on Neural Networks at https://www.youtube.com/watch?v=CqOfi41LfDw
5. Video on the back propagation algorithm at

Introduction to 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.

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

Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_i, \dots, x_n$

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.

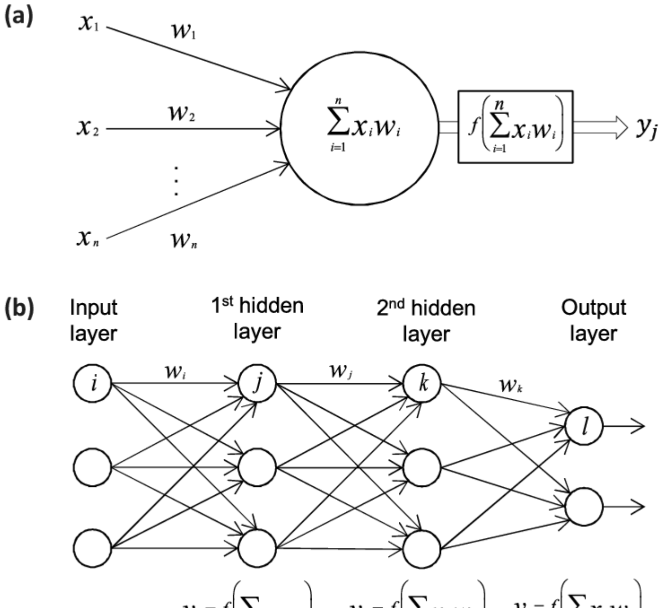
Multilayer perceptrons

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.

Illustration of a single perceptron model and a multi-perceptron model



Mathematics of deep learning

Two recent books online

1. The Modern Mathematics of Deep Learning, by Julius Berner, Philipp Grohs, Gitta Kutyniok, Philipp Petersen, published as Mathematical Aspects of Deep Learning, pp. 1-111. Cambridge University Press, 2022
2. Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory, Arnulf Jentzen, Benno Kuckuck, Philippe von Wurstemberger

Books with hands-on material and codes

- Sebastian Rashcka et al, Machine learning with Sickit-Learn and PyTorch

Reading recommendations

1. Rashkca et al., chapter 11, jupyter-notebook sent separately, from GitHub
2. Goodfellow et al, chapter 6 and 7 contain most of the neural network background.

Mathematics of deep learning and neural networks

Neural networks, in its so-called feed-forward form, where each iterations contains a feed-forward stage and a back-propgagation stage, consist of series of affine matrix-matrix and matrix-vector multiplications. The unknown parameters (the so-called biases and weights which deternine the architecture of a neural network), are uptaded iteratively using the so-called back-propagation algorithm. This algorithm corresponds to the so-called reverse mode of automatic differentation.

Basics of an NN

A neural network consists of a series of hidden layers, in addition to the input and output layers. Each layer l has a set of parameters $\boldsymbol{\Theta}^{(l)} = (\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)})$ which are related to the parameters in other layers through a series of affine transformations, for a standard NN these are matrix-matrix and matrix-vector multiplications. For all layers we will simply use a collective variable $\boldsymbol{\Theta}$.

It consist of two basic steps:

1. a feed forward stage which takes a given input and produces a final output which is compared with the target values through our cost/loss function.
2. a back-propagation state where the unknown parameters $\boldsymbol{\Theta}$ are updated through the optimization of the their gradients. The expressions for the gradients are obtained via the chain rule, starting from the derivative of the cost/function.

These two steps make up one iteration. This iterative process is continued till we reach an eventual stopping criterion.

Overarching view of a neural network

The architecture of a neural network defines our model. This model aims at describing some function $f(\boldsymbol{x}$ which represents some final result (outputs or tagrget values) given a specific inpput $\boldsymbol{x}$. Note that here $\boldsymbol{y}$ and $\boldsymbol{x}$ are not limited to be vectors.

The architecture consists of

1. An input and an output layer where the input layer is defined by the inputs $\boldsymbol{x}$. The output layer produces the model ouput $\tilde{\boldsymbol{y}}$ which is compared with the target value $\boldsymbol{y}$
2. A given number of hidden layers and neurons/nodes/units for each layer (this may vary)
3. A given activation function $\sigma(\boldsymbol{z})$ with arguments $\boldsymbol{z}$ to be defined below. The activation functions may differ from layer to layer.
4. The last layer, normally called **output** layer has normally an activation function tailored to the specific problem
5. Finally we define a so-called cost or loss function which is used to gauge the quality of our model.

The optimization problem

The cost function is a function of the unknown parameters $\boldsymbol{\Theta}$ where the latter is a container for all possible parameters needed to define a neural network
If we are dealing with a regression task a typical cost/loss function is the mean squared error

$$C(\boldsymbol{\Theta}) = \frac{1}{n}\left\{\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)^T\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\theta}\right)\right\}.$$

This function represents one of many possible ways to define the so-called cost function. Note that here we have assumed a linear dependence in terms of the paramters $\boldsymbol{\Theta}$. This is in general not the case.

Parameters of neural networks

For neural networks the parameters $\boldsymbol{\Theta}$ are given by the so-called weights and biases (to be defined below).
The weights are given by matrix elements $w_{ij}^{(l)}$ where the superscript indicates the layer number. The biases are typically given by vector elements representing each single node of a given layer, that is $b_j^{(l)}$.

Other ingredients of a neural network

Having defined the architecture of a neural network, the optimization of the cost function with respect to the parameters $\boldsymbol{\Theta}$, involves the calculations of gradients and their optimization. The gradients represent the derivatives of a multidimensional object and are often approximated by various gradient methods, including

1. various quasi-Newton methods,
2. plain gradient descent (GD) with a constant learning rate η,
3. GD with momentum and other approximations to the learning rates such as
   - Adapative gradient (ADAgrad)
   - Root mean-square propagation (RMSprop)
   - Adaptive gradient with momentum (ADAM) and many other
4. Stochastic gradient descent and various families of learning rate approximations

Other parameters

In addition to the above, there are often additional hyperparamaters which are included in the setup of a neural network. These will be discussed below.

Universal approximation theorem

The universal approximation theorem plays a central role in deep learning. Cybenko (1989) showed the following:

Let σ be any continuous sigmoidal function such that

$$\sigma(z) = \begin{cases} 1 & z \to \infty \\ 0 & z \to -\infty \end{cases}$$

Given a continuous and deterministic function $F(\boldsymbol{x})$ on the unit cube in d-dimensions $F \in [0,1]^d$, $x \in [0,1]^d$ and a parameter $\epsilon > 0$, there is a one-layer (hidden) neural network $f(\boldsymbol{x}; \boldsymbol{\Theta})$ with $\boldsymbol{\Theta} = (\boldsymbol{W}, \boldsymbol{b})$ and $\boldsymbol{W} \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b} \in \mathbb{R}^n$, for which

$$|F(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\Theta})| < \epsilon \, \forall \boldsymbol{x} \in [0,1]^d.$$

Some parallels from real analysis

For those of you familiar with for example the Stone-Weierstrass theorem for polynomial approximations or the convergence criterion for Fourier series, there are similarities in the derivation of the proof for neural networks.

The approximation theorem in words

Any continuous function $y = F(\boldsymbol{x})$ supported on the unit cube in d-dimensions can be approximated by a one-layer sigmoidal network to arbitrary accuracy.
Hornik (1991) extended the theorem by letting any non-constant, bounded activation function to be included using that the expectation value

$$\mathbb{E}[|F(\boldsymbol{x})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x})|^2 p(\boldsymbol{x}) d\boldsymbol{x} < \infty.$$

Then we have

$$\mathbb{E}[|F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2] = \int_{\boldsymbol{x}\in D} |F(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\Theta})|^2 p(\boldsymbol{x}) d\boldsymbol{x} < \epsilon.$$

More on the general approximation theorem

None of the proofs give any insight into the relation between the number of of hidden layers and nodes and the approximation error ϵ, nor the magnitudes of $\boldsymbol{W}$ and $\boldsymbol{b}$.
Neural networks (NNs) have what we may call a kind of universality no matter what function we want to compute.

It does not mean that an NN can be used to exactly compute any function. Rather, we get an approximation that is as good as we want.

Class of functions we can approximate

The class of functions that can be approximated are the continuous ones. If the function $F(\boldsymbol{x})$ is discontinuous, it won't in general be possible to approximate it. However, an NN may still give an approximation even if we fail in some points.

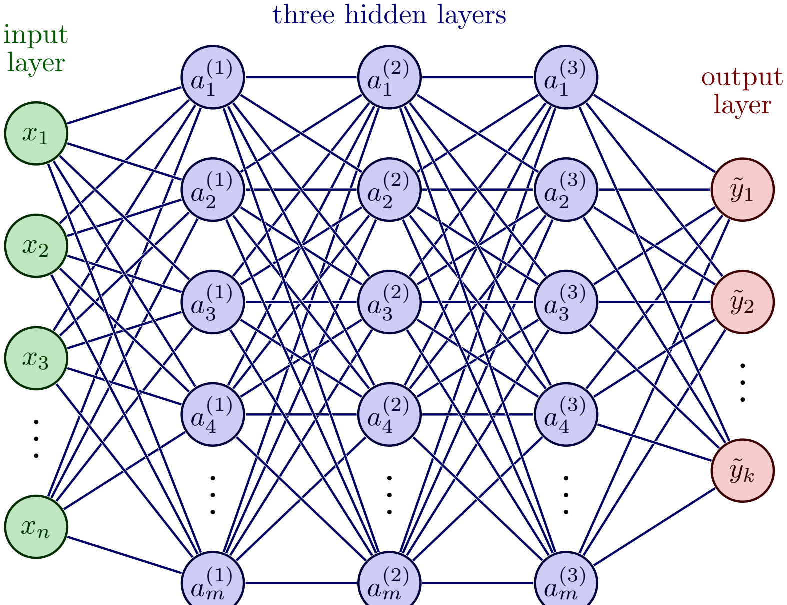
Setting up the equations for a neural network

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights and biases?
To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\boldsymbol{\Theta}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - \tilde{y}_i\right)^2,$$

where the y_is are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\boldsymbol{x}$ are given by $\boldsymbol{\tilde{y}}_i$.

Layout of a neural network with three hidden layers



Definitions

With our definition of the targets $\boldsymbol{y}$, the outputs of the network $\tilde{\boldsymbol{y}}$ and the inputs $\boldsymbol{x}$ we define now the activation z_j^l of node/neuron/unit j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs $\hat{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure in the whiteboard notes illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = \left(\hat{W}^l\right)^T \hat{a}^{l-1} + \hat{b}^l.$$

Inputs to the activation function

With the activation values $\boldsymbol{z}^l$ we can in turn define the output of layer l as $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = \sigma(z_j^l) = \frac{1}{1 + \exp -(z_j^l)}.$$

Derivatives and the chain rule

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1-a_j^l) = \sigma(z_j^l)(1-\sigma(z_j^l)).$$

Derivative of the cost function

With these definitions we can now compute the derivative of the cost function in terms of the weights.

Let us specialize to the output layer $l = L$. Our cost function is

$$\mathcal{C}(\boldsymbol{\Theta}^L) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(a_i^L - y_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial\mathcal{C}(\boldsymbol{\Theta}^L)}{\partial w_{jk}^L} = \left(a_j^L - y_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1 - a_j^L)a_k^{L-1}.$$

Simpler examples first, and automatic differentiation

In order to understand the back propagation algorithm and its derivation (an implementation of the chain rule), let us first digress with some simple examples. These examples are also meant to motivate the link with back propagation and automatic differentiation.

Reminder on the chain rule and gradients

If we have a multivariate function $f(x,y)$ where $x = x(t)$ and $y = y(t)$ are functions of a variable t, we have that the gradient of f with respect to t (without the explicit unit vector components)

$$\frac{df}{dt} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial t} \end{bmatrix} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}.$$

Multivariable functions

If we have a multivariate function $f(x, y)$ where $x = x(t, s)$ and $y = y(t, s)$ are functions of the variables t and s, we have that the partial derivatives

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial s},$$

and

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial t}.$$

the gradient of f with respect to t and s (without the explicit unit vector components)

$$\frac{df}{d(s, t)} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \end{bmatrix}.$$

Automatic differentiation through examples

A great introduction to automatic differentiation is given by Baydin et al., see `https://arxiv.org/abs/1502.05767`.
Automatic differentiation is a represented by a repeated application of the chain rule on well-known functions and allows for the calculation of derivatives to numerical precision. It is not the same as the calculation of symbolic derivatives via for example SymPy, nor does it use approximative formulae based on Taylor-expansions of a function around a given value. The latter are error prone due to truncation errors and values of the step size Δ.

Simple example

Our first example is rather simple,

$$f(x) = \exp x^2,$$

with derivative

$$f'(x) = 2x \exp x^2.$$

We can use SymPy to extract the pertinent lines of Python code through the following simple example

```
from __future__ import division
from sympy import *
x = symbols('x')
expr = exp(x*x)
simplify(expr)
derivative = diff(expr,x)
print(python(expr))
print(python(derivative))
```

Smarter way of evaluating the above function

If we study this function, we note that we can reduce the number of operations by introducing an intermediate variable

$$a = x^2,$$

leading to

$$f(x) = f(a(x)) = b = \exp a.$$

We now assume that all operations can be counted in terms of equal floating point operations. This means that in order to calculate $f(x)$ we need first to square x and then compute the exponential. We have thus two floating point operations only.

Reducing the number of operations

With the introduction of a precalculated quantity a and thereby $f(x)$ we have that the derivative can be written as

$$f'(x) = 2xb,$$

which reduces the number of operations from four in the orginal expression to two. This means that if we need to compute $f(x)$ and its derivative (a common task in optimizations), we have reduced the number of operations from six to four in total.
Note that the usage of a symbolic software like SymPy does not include such simplifications and the calculations of the function and the derivatives yield in general more floating point operations.

Chain rule, forward and reverse modes

In the above example we have introduced the variables a and b, and our function is

$$f(x) = f(a(x)) = b = \exp a,$$

with $a = x^2$. We can decompose the derivative of f with respect to x as

$$\frac{df}{dx} = \frac{df}{db}\frac{db}{da}\frac{da}{dx}.$$

We note that since $b = f(x)$ that

$$\frac{df}{db} = 1,$$

leading to

$$\frac{df}{dx} = \frac{db}{da}\frac{da}{dx} = 2x \exp x^2,$$

as before.

Forward and reverse modes

We have that

$$\frac{df}{dx} = \frac{df}{db}\frac{db}{da}\frac{da}{dx},$$

which we can rewrite either as

$$\frac{df}{dx} = \left[\frac{df}{db}\frac{db}{da}\right]\frac{da}{dx},$$

or

$$\frac{df}{dx} = \frac{df}{db}\left[\frac{db}{da}\frac{da}{dx}\right].$$

The first expression is called reverse mode (or back propagation) since we start by evaluating the derivatives at the end point and then propagate backwards. This is the standard way of evaluating derivatives (gradients) when optimizing the parameters of a neural network. In the context of deep learning this is computationally more efficient since the output of a neural network consists of either one or some few other output variables.
The second equation defines the so-called **forward mode**.

More complicated function

We increase our ambitions and introduce a slightly more complicated function

$$f(x) = \sqrt{x^2 + expx^2},$$

with derivative

$$f'(x) = \frac{x(1 + \exp x^2)}{\sqrt{x^2 + expx^2}}.$$

The corresponding SymPy code reads

```
from __future__ import division
from sympy import *
x = symbols('x')
expr = sqrt(x*x+exp(x*x))
simplify(expr)
derivative = diff(expr,x)
print(python(expr))
print(python(derivative))
```

Counting the number of floating point operations

A simple count of operations shows that we need five operations for the function itself and ten for the derivative. Fifteen operations in total if we wish to proceed with the above codes.
Can we reduce this to say half the number of operations?

Defining intermediate operations

We can indeed reduce the number of operation to half of those listed in the brute force approach above. We define the following quantities

$$a = x^2,$$

and

$$b = \exp x^2 = \exp a,$$

and

$$c = a + b,$$

and

$$d = f(x) = \sqrt{c}.$$

New expression for the derivative

With these definitions we obtain the following partial derivatives

$$\frac{\partial a}{\partial x} = 2x,$$

and

$$\frac{\partial b}{\partial a} = \exp a,$$

and

$$\frac{\partial c}{\partial a} = 1,$$

and

$$\frac{\partial c}{\partial b} = 1,$$

and

$$\frac{\partial d}{\partial c} = \frac{1}{2\sqrt{c}},$$

and finally

$$\frac{\partial f}{\partial d} = 1.$$

Final derivatives

Our final derivatives are thus

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} = \frac{1}{2\sqrt{c}},$$

$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b} = \frac{1}{2\sqrt{c}},$$

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial a} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} = \frac{1+\exp a}{2\sqrt{c}},$$

and finally

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = \frac{x(1+\exp a)}{\sqrt{c}},$$

which is just

$$\frac{\partial f}{\partial x} = \frac{x(1+b)}{d},$$

and requires only three operations if we can reuse all intermediate variables.

In general not this simple

In general, see the generalization below, unless we can obtain simple analytical expressions which we can simplify further, the final implementation of automatic differentiation involves repeated calculations (and thereby operations) of derivatives of elementary functions.

Automatic differentiation

We can make this example more formal. Automatic differentiation is a formalization of the previous example (see graph).
We define $\boldsymbol{x} \in x_1, \dots, x_l$ input variables to a given function $f(\boldsymbol{x})$ and $x_{l+1}, \dots, x_L$ intermediate variables.
In the above example we have only one input variable, $l = 1$ and four intermediate variables, that is

$$\left[x_1 = x \quad x_2 = x^2 = a \quad x_3 = \exp a = b \quad x_4 = c = a + b \quad x_5 = \sqrt{c} = d\right]$$

Furthemore, for $i = l + 1, \dots, L$ (here $i = 2, 3, 4, 5$ and $f = x_L = d$), we define the elementary functions $g_i(x_{Pa(x_i)})$ where $x_{Pa(x_i)}$ are the parent nodes of the variable x_i.
In our case, we have for example for $x_3 = g_3(x_{Pa(x_i)}) = \exp a$, that $g_3 = \exp()$ and $x_{Pa(x_3)} = a$.

Chain rule

We can now compute the gradients by back-propagating the derivatives using the chain rule. We have defined

$$\frac{\partial f}{\partial x_L} = 1,$$

which allows us to find the derivatives of the various variables x_i as

$$\frac{\partial f}{\partial x_i} = \sum_{x_j : x_i \in Pa(x_j)} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i} = \sum_{x_j : x_i \in Pa(x_j)} \frac{\partial f}{\partial x_j} \frac{\partial g_j}{\partial x_i}.$$

Whenever we have a function which can be expressed as a computation graph and the various functions can be expressed in terms of elementary functions that are differentiable, then automatic differentiation works. The functions may not need to be elementary functions, they could also be computer programs, although not all programs can be automatically differentiated.

First network example, simple percepetron with one input

As yet another example we define now a simple perceptron model with all quantities given by scalars. We consider only one input variable x and one target value y. We define an activation function σ_1 which takes as input

$$z_1 = w_1 x + b_1,$$

where w_1 is the weight and b_1 is the bias. These are the parameters we want to optimize. The output is $a_1 = \sigma(z_1)$ (see graph from whiteboard notes). This output is then fed into the **cost/loss** function, which we here for the sake of simplicity just define as the squared error

$$C(x; w_1, b_1) = \frac{1}{2}(a_1 - y)^2.$$

Layout of a simple neural network with no hidden layer

$$z = w_1 x + b_1$$

$$\begin{tikzcd}
x \arrow[r, "w_1"] & \boxed{\sigma(z)} \arrow[r] & a \arrow[r] & \boxed{C} & y \arrow[l] \\
& b_1 \arrow[u] & & &
\end{tikzcd}$$

$$C = C(a, y; \Theta)$$

$$\Theta = \{w_1, b_1\}$$

Optimizing the parameters

In setting up the feed forward and back propagation parts of the algorithm, we need now the derivative of the various variables we want to train.

We need

$$\frac{\partial C}{\partial w_1} \text{ and } \frac{\partial C}{\partial b_1}.$$

Using the chain rule we find

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1} = (a_1 - y)\sigma_1' x,$$

and

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial b_1} = (a_1 - y)\sigma_1',$$

which we later will just define as

$$\frac{\partial C}{\partial a_1}\frac{\partial a_1}{\partial z_1} = \delta_1.$$

Adding a hidden layer

We change our simple model to (see graph) a network with just one hidden layer but with scalar variables only.
Our output variable changes to a_2 and a_1 is now the output from the hidden node and $a_0 = x$. We have then

$$z_1 = w_1 a_0 + b_1 \ \wedge a_1 = \sigma_1(z_1),$$

$$z_2 = w_2 a_1 + b_2 \ \wedge a_2 = \sigma_2(z_2),$$

and the cost function

$$C(x; \boldsymbol{\Theta}) = \frac{1}{2}(a_2 - y)^2,$$

with $\boldsymbol{\Theta} = [w_1, w_2, b_1, b_2]$.

Layout of a simple neural network with one hidden layer

$$z_1 = w_1 x + b_1 \qquad z_2 = w_2 a_1 + b_2$$

$$\begin{tikzcd}
x \arrow[r, "w_1"] & \boxed{\sigma_1(z_1)} \arrow[r] & a_1 \arrow[r, "w_2"] & \boxed{\sigma_2(z_2)} \arrow[r] & a_2 \arrow[r] & \boxed{C} & y \arrow[l] \\
& b_1 \arrow[u] & & b_2 \arrow[u] & & &
\end{tikzcd}$$

$$C = C(a_2, y; \Theta)$$

$$\Theta = \{w_1, w_2, b_1, b_2\}$$

The derivatives

The derivatives are now, using the chain rule again

$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial w_2} = (a_2 - y)\sigma_2' a_1 = \delta_2 a_1,$$

$$\frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial b_2} = (a_2 - y)\sigma_2' = \delta_2,$$

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1} = (a_2 - y)\sigma_2' a_1 \sigma_1' a_0,$$

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2}\frac{\partial z_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial b_1} = (a_2 - y)\sigma_2' \sigma_1' = \delta_1.$$

Can you generalize this to more than one hidden layer?

Important observations

From the above equations we see that the derivatives of the activation functions play a central role. If they vanish, the training may stop. This is called the vanishing gradient problem, see discussions below. If they become large, the parameters w_i and b_i may simply go to infinity. This is referenced as the exploding gradient problem.

The training

The training of the parameters is done through various gradient descent approximations with

$$w_i \leftarrow w_i - \eta\delta_i a_{i-1},$$

and

$$b_i \leftarrow b_i - \eta\delta_i,$$

with η is the learning rate.
One iteration consists of one feed forward step and one back-propagation step. Each back-propagation step does one update of the parameters $\boldsymbol{\Theta}$.
For the first hidden layer $a_{i-1} = a_0 = x$ for this simple model.

Code example

The code here implements the above model with one hidden layer and scalar variables for the same function we studied in the previous example. The code is however set up so that we can add multiple inputs x and target values y. Note also that we have the possibility of defining a feature matrix $\boldsymbol{X}$ with more than just one column for the input values. This will turn useful in our next example. We have also defined matrices and vectors for all of our operations although it is not necessary here.

```python
import numpy as np
# We use the Sigmoid function as activation function
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def forwardpropagation(x):
    # weighted sum of inputs to the hidden layer
    z_1 = np.matmul(x, w_1) + b_1
    # activation in the hidden layer
    a_1 = sigmoid(z_1)
    # weighted sum of inputs to the output layer
    z_2 = np.matmul(a_1, w_2) + b_2
    a_2 = z_2
    return a_1, a_2

def backpropagation(x, y):
```

Try to increase the amount of input and target/output data. Try also to perform calculations for more values of the learning rates. Feel free to add either hyperparameters with an l_1 norm or an l_2 norm and discuss your results. Discuss your results as functions of the amount of training data and various learning rates.
Challenge: Try to change the activation functions and replace the hard-coded analytical expressions with automatic derivation via either **autograd** or **JAX**.

Simple neural network and the back propagation equations

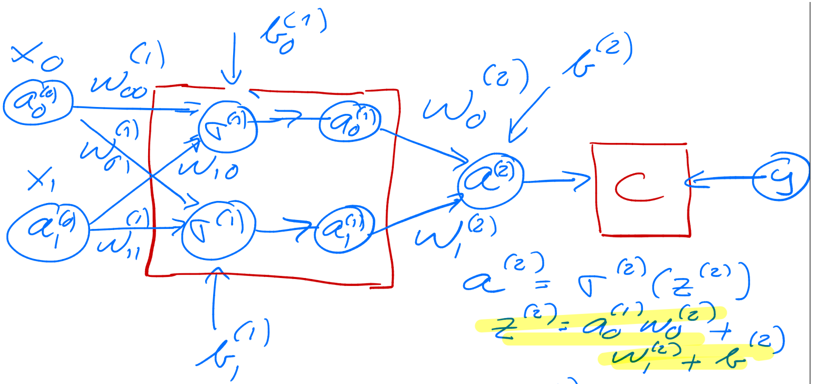
Let us now try to increase our level of ambition and attempt at setting up the equations for a neural network with two input nodes, one hidden layer with two hidden nodes and one output layer with one output node/neuron only (see graph)..
We need to define the following parameters and variables with the input layer (layer (0)) where we label the nodes x_0 and x_1

$$x_0 = a_0^{(0)} \wedge x_1 = a_1^{(0)}.$$

The hidden layer (layer (1)) has nodes which yield the outputs $a_0^{(1)}$ and $a_1^{(1)}$) with weight $\boldsymbol{w}$ and bias $\boldsymbol{b}$ parameters

$$w_{ij}^{(1)} = \left\{w_{00}^{(1)}, w_{01}^{(1)}, w_{10}^{(1)}, w_{11}^{(1)}\right\} \wedge b^{(1)} = \left\{b_0^{(1)}, b_1^{(1)}\right\}.$$

Layout of a simple neural network with two input nodes, one hidden layer and one output node



$$a^{(2)} = \sigma^{(2)}(z^{(2)})$$

$$z^{(2)} = a_0^{(1)} w_0^{(2)} + w_1^{(2)} + b^{(2)}$$

$$a_0^{(1)} = \sigma^{(1)}(z_0^{(1)}) \wedge z_0^{(1)} = w_{00}^{(1)} a_0^{(0)} + w_{01}^{(1)} a_1^{(0)} + b_0^{(1)}$$

$$a_1^{(1)} = \sigma^{(1)}(z_1^{(1)}) \wedge z_1^{(1)} = w_{10}^{(1)} a_0^{(0)} + w_{11}^{(1)} a_1^{(0)} + b_1^{(0)}$$

The ouput layer

Finally, we have the ouput layer given by layer label (2) with output $a^{(2)}$ and weights and biases to be determined given by the variables

$$w_i^{(2)} = \left\{w_0^{(2)}, w_1^{(2)}\right\} \wedge b^{(2)}.$$

Our output is $\tilde{y} = a^{(2)}$ and we define a generic cost function $C(a^{(2)}, y; \boldsymbol{\Theta})$ where y is the target value (a scalar here). The parameters we need to optimize are given by

$$\boldsymbol{\Theta} = \left\{w_{00}^{(1)}, w_{01}^{(1)}, w_{10}^{(1)}, w_{11}^{(1)}, w_0^{(2)}, w_1^{(2)}, b_0^{(1)}, b_1^{(1)}, b^{(2)}\right\}.$$

Compact expressions

We can define the inputs to the activation functions for the various layers in terms of various matrix-vector multiplications and vector additions. The inputs to the first hidden layer are

$$\begin{bmatrix} z_0^{(1)} \\ z_1^{(1)} \end{bmatrix} = \begin{bmatrix} w_{00}^{(1)} & w_{01}^{(1)} \\ w_{10}^{(1)} & w_{11}^{(1)} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \end{bmatrix},$$

with outputs

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \end{bmatrix} = \begin{bmatrix} \sigma^{(1)}(z_0^{(1)}) \\ \sigma^{(1)}(z_1^{(1)}) \end{bmatrix}.$$

Output layer

For the final output layer we have the inputs to the final activation function

$$z^{(2)} = w_0^{(2)} a_0^{(1)} + w_1^{(2)} a_1^{(1)} + b^{(2)},$$

resulting in the output

$$a^{(2)} = \sigma^{(2)}(z^{(2)}).$$

Explicit derivatives

In total we have nine parameters which we need to train. Using the chain rule (or just the back-propagation algorithm) we can find all derivatives. Since we will use automatic differentiation in reverse mode, we start with the derivatives of the cost function with respect to the parameters of the output layer, namely

$$\frac{\partial C}{\partial w_i^{(2)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial w_i^{(2)}} = \delta^{(2)} a_i^{(1)},$$

with

$$\delta^{(2)} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}$$

and finally

$$\frac{\partial C}{\partial b^{(2)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial b^{(2)}} = \delta^{(2)}.$$

Derivatives of the hidden layer

Using the chain rule we have the following expressions for say one of the weight parameters (it is easy to generalize to the other weight parameters)

$$\frac{\partial C}{\partial w_{00}^{(1)}} = \frac{\partial C}{\partial a^{(2)}}\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial z_0^{(1)}}\frac{\partial z_0^{(1)}}{\partial w_{00}^{(1)}} = \delta^{(2)}\frac{\partial z^{(2)}}{\partial z_0^{(1)}}\frac{\partial z_0^{(1)}}{\partial w_{00}^{(1)}},$$

which, noting that

$$z^{(2)} = w_0^{(2)}a_0^{(1)} + w_1^{(2)}a_1^{(1)} + b^{(2)},$$

allows us to rewrite

$$\frac{\partial z^{(2)}}{\partial z_0^{(1)}}\frac{\partial z_0^{(1)}}{\partial w_{00}^{(1)}} = w_0^{(2)}\frac{\partial a_0^{(1)}}{\partial z_0^{(1)}}a_0^{(1)}.$$

Final expression

Defining

$$\delta_0^{(1)} = w_0^{(2)} \frac{\partial a_0^{(1)}}{\partial z_0^{(1)}} \delta^{(2)},$$

we have

$$\frac{\partial C}{\partial w_{00}^{(1)}} = \delta_0^{(1)} a_0^{(1)}.$$

Similarly, we obtain

$$\frac{\partial C}{\partial w_{01}^{(1)}} = \delta_0^{(1)} a_1^{(1)}.$$

Completing the list

Similarly, we find

$$\frac{\partial C}{\partial w_{10}^{(1)}} = \delta_1^{(1)} a_0^{(1)},$$

and

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \delta_1^{(1)} a_1^{(1)},$$

where we have defined

$$\delta_1^{(1)} = w_1^{(2)} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \delta^{(2)}.$$

Final expressions for the biases of the hidden layer

For the sake of completeness, we list the derivatives of the biases, which are

$$\frac{\partial C}{\partial b_0^{(1)}} = \delta_0^{(1)},$$

and

$$\frac{\partial C}{\partial b_1^{(1)}} = \delta_1^{(1)}.$$

As we will see below, these expressions can be generalized in a more compact form.

Gradient expressions

For this specific model, with just one output node and two hidden nodes, the gradient descent equations take the following form for output layer

$$w_i^{(2)} \leftarrow w_i^{(2)} - \eta \delta^{(2)} a_i^{(1)},$$

and

$$b^{(2)} \leftarrow b^{(2)} - \eta \delta^{(2)},$$

and

$$w_{ij}^{(1)} \leftarrow w_{ij}^{(1)} - \eta \delta_i^{(1)} a_j^{(0)},$$

and

$$b_i^{(1)} \leftarrow b_i^{(1)} - \eta \delta_i^{(1)},$$

where η is the learning rate.

We extend our simple code to a function which depends on two variable x_0 and x_1, that is

$$y = f(x_0, x_1) = x_0^2 + 3x_0x_1 + x_1^2 + 5.$$

We feed our network with $n = 100$ entries x_0 and x_1. We have thus two features represented by these variable and an input matrix/design matrix $\boldsymbol{X} \in \mathrm{R}^{n\times 2}$

$$\boldsymbol{X} = \begin{bmatrix} x_{00} & x_{01} \\ x_{00} & x_{01} \\ x_{10} & x_{11} \\ x_{20} & x_{21} \\ \dots & \dots \\ \dots & \dots \\ x_{n-20} & x_{n-21} \\ x_{n-10} & x_{n-11} \end{bmatrix}.$$

Write a code, based on the previous code examples, which takes as input these data and fit the above function. You can extend your code to include automatic differentiation.
With these examples, we are now ready to embark upon the writing of more a general code for neural networks.

Getting serious, the back propagation equations for a neural network

Now it is time to move away from one node in each layer only. Our inputs are also represented either by several inputs.
We have thus

$$\frac{\partial\mathcal{C}((\boldsymbol{\Theta}^L)}{\partial w_{jk}^L} = \left(a_j^L - y_j\right) a_j^L(1-a_j^L)a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1-a_j^L)\left(a_j^L - y_j\right) = \sigma'(z_j^L)\frac{\partial\mathcal{C}}{\partial(a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\boldsymbol{\delta}^L = \sigma'(\hat{z}^L)\circ\frac{\partial\mathcal{C}}{\partial(\boldsymbol{a}^L)}.$$

Analyzing the last results

This is an important expression. The second term on the right handside measures how fast the cost function is changing as a function of the jth output activation. If, for example, the cost function doesn't depend much on a particular output node j, then δ_j^L will be small, which is what we would expect. The first term on the right, measures how fast the activation function f is changing at a given activation value z_j^L.

More considerations

Notice that everything in the above equations is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it is only a small additional overhead to compute $\sigma'(z_j^L)$. The exact form of the derivative with respect to the output depends on the form of the cost function. However, provided the cost function is known there should be little trouble in calculating

$$\frac{\partial \mathcal{C}}{\partial (a_j^L)}$$

With the definition of δ_j^L we have a more compact definition of the derivative of the cost function in terms of the weights, namely

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}.$$

Derivatives in terms of z_j^L

It is also easy to see that our previous equation can be written as

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L},$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, namely

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

Bringing it together

We have now three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{2}$$

and

$$\delta_j^L = \sigma'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}, \tag{3}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{4}$$

Final back propagating equation

We have that (replacing L with a general layer l)

$$\delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l+1$.

Using the chain rule and summing over all k entries

We obtain

$$\delta_j^l = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

with M_l being the number of nodes in layer l, we obtain

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l),$$

This is our final equation.
We are now ready to set up the algorithm for back propagation and learning the weights and biases.

Setting up the back propagation algorithm

The four equations provide us with a way of computing the gradient of the cost function. Let us write this out in the form of an algorithm.

First, we set up the input data $\hat{x}$ and the activations $\hat{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^1$.

Secondly, we perform then the feed forward till we reach the output layer and compute all $\hat{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\hat{a}^l$ for $l = 1, 2, 3, \dots, L$.

Notation: The first hidden layer has $l = 1$ as label and the final output layer has $l = L$.

Setting up the back propagation algorithm, part 2

Thereafter we compute the ouput error $\hat{\delta}^L$ by computing all

$$\delta_j^L = \sigma'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \dots, 1$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l).$$

Setting up the Back propagation algorithm, part 3

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \dots, 1$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta\delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta\frac{\partial\mathcal{C}}{\partial b_j^l} = b_j^l - \eta\delta_j^l,$$

with η being the learning rate.

Updating the gradients

With the back propagate error for each $l = L-1, L-2, \dots, 1$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} sigma'(z_j^l),$$

we update the weights and the biases using gradient descent for each $l = L-1, L-2, \dots, 1$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

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 sc
The value z is the weighted sum of parameters involved in the learning

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()
```

Setting up a Multi-layer perceptron model for classification

We are now gong to develop an example based on the MNIST data base. This is a classification problem and we need to use our cross-entropy function we discussed in connection with logistic regression. The cross-entropy defines our cost function for the classificaton problems with neural networks.

In binary classification with two classes $(0, 1)$ we define the logistic/sigmoid function as the probability that a particular input is in class 0 or 1. This is possible because the logistic function takes any input from the real numbers and inputs a number between 0 and 1, and can therefore be interpreted as a probability. It also has other nice properties, such as a derivative that is simple to calculate.

For an input $\boldsymbol{a}$ from the hidden layer, the probability that the input $\boldsymbol{x}$ is in class 0 or 1 is just. We let θ represent the unknown weights and biases to be adjusted by our equations). The variable x represents our activation values z. We have

$$P(y = 0 \mid \boldsymbol{x}, \boldsymbol{\theta}) = \frac{1}{1 + \exp{(-\boldsymbol{x})}},$$

Defining the cost function

Our cost function is given as (see the Logistic regression lectures)

$$\mathcal{C}(\boldsymbol{\theta}) = -\ln P(\mathcal{D} \mid \boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \ln[P(y_i = 0)] + (1 - y_i) \ln[1 - P(y_i = 0)] =$$

This last equality means that we can interpret our *cost* function as a sum over the *loss* function for each point in the dataset $\mathcal{L}_i(\boldsymbol{\theta})$. The negative sign is just so that we can think about our algorithm as minimizing a positive number, rather than maximizing a negative number.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$y = 5 \quad \rightarrow \quad \boldsymbol{y} = (0,0,0,0,0,1,0,0,0,0)$, and

$y = 1 \quad \rightarrow \quad \boldsymbol{y} = (0,1,0,0,0,0,0,0,0,0)$,

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset (numbers from 0 to 9)..

If $\boldsymbol{x}_i$ is the i-th input (image), y_{ic} refers to the c-th component of the i-th output vector $\boldsymbol{y}_i$. The probability of $\boldsymbol{x}_i$ being in class c will be given by the softmax function:

Example: binary classification problem

As an example of the above, relevant for project 2 as well, let us consider a binary class. As discussed in our logistic regression lectures, we defined a cost function in terms of the parameters β as

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

where we had defined the logistic (sigmoid) function

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

and

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

The parameters $\boldsymbol{\beta}$ were defined using a minimization method like gradient descent or Newton-Raphson's method.

Now we replace x_i with the activation z_i^l for a given layer l and the outputs as $y_i = a_i^l = f(z_i^l)$, with z_i^l now being a function of the weights w_{ij}^l and biases b_i^l. We have then

$$a_i^l = y_i = \frac{\exp{(z_i^l)}}{}$$

The Softmax function

In case we employ the more general case given by the Softmax equation, we need to evaluate the derivative of the activation function with respect to the activation z_i^l, that is we need

$$\frac{\partial f(z_i^l)}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial f(z_i^l)}{\partial z_j^l} a_k^{l-1}.$$

For the Softmax function we have

$$f(z_i^l) = \frac{\exp{(z_i^l)}}{\sum_{m=1}^K \exp{(z_m^l)}}.$$

Its derivative with respect to z_j^l gives

$$\frac{\partial f(z_i^l)}{\partial z_j^l} = f(z_i^l)\left(\delta_{ij} - f(z_j^l)\right),$$

which in case of the simply binary model reduces to having $i = j$.

Developing a code for doing neural networks with back propagation

One can identify a set of key steps when using neural networks to solve supervised learning problems:

1. Collect and pre-process data
2. Define model and architecture
3. Choose cost function and optimizer
4. Train the model
5. Evaluate model performance on test data
6. Adjust hyperparameters (if necessary, network architecture)

Collect and pre-process data

Here we will be using the MNIST dataset, which is readily available through the **scikit-learn** package. You may also find it for example here. The *MNIST* (Modified National Institute of Standards and Technology) database is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST dataset consists of 70 000 images of size 28×28 pixels, each labeled from 0 to 9. The scikit-learn dataset we will use consists of a selection of 1797 images of size 8×8 collected and processed from this database.

To feed data into a feed-forward neural network we need to represent the inputs as a design/feature matrix $X = (n_{inputs}, n_{features})$. Each row represents an *input*, in this case a handwritten digit, and each column represents a *feature*, in this case a pixel. The correct answers, also known as *labels* or *targets* are represented as a 1D array of integers $Y = (n_{inputs}) = (5, 3, 1, 8, ...)$.

As an example, say we want to build a neural network using supervised learning to predict Body-Mass Index (BMI) from measurements of height (in m) and weight (in kg). If we have measurements of 5 people the design/feature matrix could be for

Train and test datasets

Performing analysis before partitioning the dataset is a major error, that can lead to incorrect conclusions.
We will reserve 80% of our dataset for training and 20% for testing.
It is important that the train and test datasets are drawn randomly from our dataset, to ensure no bias in the sampling. Say you are taking measurements of weather data to predict the weather in the coming 5 days. You don't want to train your model on measurements taken from the hours 00.00 to 12.00, and then test it on data collected from 12.00 to 24.00.

```
from sklearn.model_selection import train_test_split

# one-liner from scikit-learn library
train_size = 0.8
test_size = 1 - train_size
X_train, X_test, Y_train, Y_test = train_test_split(inputs, labels, tr
                                                    test_size=test_siz

# equivalently in numpy
def train_test_split_numpy(inputs, labels, train_size, test_size):
    n_inputs = len(inputs)
    inputs_shuffled = inputs.copy()
    labels_shuffled = labels.copy()

    np.random.shuffle(inputs_shuffled)
```

Define model and architecture

Our simple feed-forward neural network will consist of an *input* layer, a single *hidden* layer and an *output* layer. The activation y of each neuron is a weighted sum of inputs, passed through an activation function. In case of the simple perceptron model we have

$$z = \sum_{i=1}^{n} w_i a_i,$$

$$y = f(z),$$

where f is the activation function, a_i represents input from neuron i in the preceding layer and w_i is the weight to input i. The activation of the neurons in the input layer is just the features (e.g. a pixel value).

The simplest activation function for a neuron is the *Heaviside* function:

$$f(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Layers

- Input

Since each input image has 8x8 = 64 pixels or features, we have an input layer of 64 neurons.

- Hidden layer

We will use 50 neurons in the hidden layer receiving input from the neurons in the input layer. Since each neuron in the hidden layer is connected to the 64 inputs we have 64x50 = 3200 weights to the hidden layer.

- Output

If we were building a binary classifier, it would be sufficient with a single neuron in the output layer, which could output 0 or 1 according to the Heaviside function. This would be an example of a *hard* classifier, meaning it outputs the class of the input directly. However, if we are dealing with noisy data it is often beneficial to use a *soft* classifier, which outputs the probability of being in class 0 or 1.

For a soft binary classifier, we could use a single neuron and interpret the output as either being the probability of being in class 0 or the probability of being in class 1. Alternatively we could use 2

Weights and biases

Typically weights are initialized with small values distributed around zero, drawn from a uniform or normal distribution. Setting all weights to zero means all neurons give the same output, making the network useless.

Adding a bias value to the weighted sum of inputs allows the neural network to represent a greater range of values. Without it, any input with the value 0 will be mapped to zero (before being passed through the activation). The bias unit has an output of 1, and a weight to each neuron j, b_j:

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j.$$

The bias weights $\boldsymbol{b}$ are often initialized to zero, but a small value like 0.01 ensures all neurons have some output which can be backpropagated in the first training cycle.

```
# building our neural network

n_inputs, n_features = X_train.shape
n_hidden_neurons = 50
n_categories = 10
```

Feed-forward pass

Denote F the number of features, H the number of hidden neurons and C the number of categories. For each input image we calculate a weighted sum of input features (pixel values) to each neuron j in the hidden layer l:

$$z_j^l = \sum_{i=1}^{F} w_{ij}^l x_i + b_j^l,$$

this is then passed through our activation function

$$a_j^l = f(z_j^l).$$

We calculate a weighted sum of inputs (activations in the hidden layer) to each neuron j in the output layer:

$$z_j^L = \sum_{i=1}^{H} w_{ij}^L a_i^l + b_j^L.$$

Finally we calculate the output of neuron j in the output layer using the softmax function:

Matrix multiplications

Since our data has the dimensions $X = (n_{inputs}, n_{features})$ and our weights to the hidden layer have the dimensions $W_{hidden} = (n_{features}, n_{hidden})$, we can easily feed the network all our training data in one go by taking the matrix product

$$XW^h = (n_{inputs}, n_{hidden}),$$

and obtain a matrix that holds the weighted sum of inputs to the hidden layer for each input image and each hidden neuron. We also add the bias to obtain a matrix of weighted sums to the hidden layer Z^h:

$$\boldsymbol{z}^l = \boldsymbol{X}\boldsymbol{W}^l + \boldsymbol{b}^l,$$

meaning the same bias (1D array with size equal number of hidden neurons) is added to each input image. This is then passed through the activation:

$$\boldsymbol{a}^l = f(\boldsymbol{z}^l).$$

This is fed to the output layer:

Choose cost function and optimizer

To measure how well our neural network is doing we need to introduce a cost function. We will call the function that gives the error of a single sample output the *loss* function, and the function that gives the total error of our network across all samples the *cost* function. A typical choice for multiclass classification is the *cross-entropy* loss, also known as the negative log likelihood.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$$y = 5 \quad \rightarrow \quad \boldsymbol{y} = (0,0,0,0,0,1,0,0,0,0),$$

$$y = 1 \quad \rightarrow \quad \boldsymbol{y} = (0,1,0,0,0,0,0,0,0,0),$$

i.e. a binary bit string of length C, where $C = 10$ is the number of classes in the MNIST dataset.

Let y_{ic} denote the c-th component of the i-th one-hot vector. We define the cost function $\mathcal{C}$ as a sum over the cross-entropy loss for each point $\boldsymbol{x}_i$ in the dataset.

In the one-hot representation only one of the terms in the loss

Optimizing the cost function

The network is trained by finding the weights and biases that minimize the cost function. One of the most widely used classes of methods is *gradient descent* and its generalizations. The idea behind gradient descent is simply to adjust the weights in the direction where the gradient of the cost function is large and negative. This ensures we flow toward a *local* minimum of the cost function. Each parameter θ is iteratively adjusted according to the rule

$$\theta_{i+1} = \theta_i - \eta \nabla \mathcal{C}(\theta_i),$$

where η is known as the *learning rate*, which controls how big a step we take towards the minimum. This update can be repeated for any number of iterations, or until we are satisfied with the result. A simple and effective improvement is a variant called *Batch Gradient Descent*. Instead of calculating the gradient on the whole dataset, we calculate an approximation of the gradient on a subset of the data called a *minibatch*. If there are N data points and we have a minibatch size of M, the total number of batches is N/M.

Regularization

It is common to add an extra term to the cost function, proportional to the size of the weights. This is equivalent to constraining the size of the weights, so that they do not grow out of control. Constraining the size of the weights means that the weights cannot grow arbitrarily large to fit the training data, and in this way reduces *overfitting*.

We will measure the size of the weights using the so called *L2-norm*, meaning our cost function becomes:

$$\mathcal{C}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) \quad \rightarrow \quad \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i(\theta) + \lambda||\boldsymbol{w}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(\theta) + \lambda\sum_{ij} w$$

i.e. we sum up all the weights squared. The factor λ is known as a regularization parameter.

In order to train the model, we need to calculate the derivative of the cost function with respect to every bias and weight in the network. In total our network has $(64+1)\times 50 = 3250$ weights in the hidden layer and $(50+1)\times 10 = 510$ weights to the output layer (+1 for the bias), and the gradient must be calculated for

Matrix multiplication

To more efficently train our network these equations are implemented using matrix operations. The error in the output layer is calculated simply as, with $\boldsymbol{t}$ being our targets,

$$\delta_L = \boldsymbol{t} - \boldsymbol{y} = (n_{inputs}, n_{categories}).$$

The gradient for the output weights is calculated as

$$\nabla W_L = \boldsymbol{a}^T \delta_L = (n_{hidden}, n_{categories}),$$

where $\boldsymbol{a} = (n_{inputs}, n_{hidden})$. This simply means that we are summing up the gradients for each input. Since we are going backwards we have to transpose the activation matrix.

The gradient with respect to the output bias is then

$$\nabla \boldsymbol{b}_L = \sum_{i=1}^{n_{inputs}} \delta_L = (n_{categories}).$$

The error in the hidden layer is

Improving performance

As we can see the network does not seem to be learning at all. It seems to be just guessing the label for each image. In order to obtain a network that does something useful, we will have to do a bit more work.
The choice of *hyperparameters* such as learning rate and regularization parameter is hugely influential for the performance of the network. Typically a *grid-search* is performed, wherein we test different hyperparameters separated by orders of magnitude. For example we could test the learning rates $\eta = 10^{-6}, 10^{-5}, ..., 10^{-1}$ with different regularization parameters $\lambda = 10^{-6}, ..., 10^{-0}$.
Next, we haven't implemented minibatching yet, which introduces stochasticity and is though to act as an important regularizer on the weights. We call a feed-forward + backward pass with a minibatch an *iteration*, and a full training period going through the entire dataset (n/M batches) an *epoch*.
If this does not improve network performance, you may want to consider altering the network architecture, adding more neurons or hidden layers. Andrew Ng goes through some of these considerations in this video. You can find a summary of the video

Full object-oriented implementation

It is very natural to think of the network as an object, with specific instances of the network being realizations of this object with different hyperparameters. An implementation using Python classes provides a clean structure and interface, and the full implementation of our neural network is given below.

```
class NeuralNetwork:
    def __init__(
            self,
            X_data,
            Y_data,
            n_hidden_neurons=50,
            n_categories=10,
            epochs=10,
            batch_size=100,
            eta=0.1,
            lmbd=0.0):

        self.X_data_full = X_data
        self.Y_data_full = Y_data

        self.n_inputs = X_data.shape[0]
        self.n_features = X_data.shape[1]
        self.n_hidden_neurons = n_hidden_neurons
        self.n_categories = n_categories

        self.epochs = epochs
```

Evaluate model performance on test data

To measure the performance of our network we evaluate how well it does it data it has never seen before, i.e. the test data. We measure the performance of the network using the *accuracy* score. The accuracy is as you would expect just the number of images correctly labeled divided by the total number of images. A perfect classifier will have an accuracy score of 1.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n},$$

where I is the indicator function, 1 if $\tilde{y}_i = y_i$ and 0 otherwise.

```
epochs = 100
batch_size = 100

dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmbd, epoch
                    n_hidden_neurons=n_hidden_neurons, n_categories=n_
dnn.train()
test_predict = dnn.predict(X_test)

# accuracy score from scikit library
print("Accuracy score on test set: ", accuracy_score(Y_test, test_pred

# equivalent in numpy
def accuracy_score_numpy(Y_test, Y_pred):
    return np.sum(Y_test == Y_pred) / len(Y_test)
```

Adjust hyperparameters

We now perform a grid search to find the optimal hyperparameters for the network. Note that we are only using 1 layer with 50 neurons, and human performance is estimated to be around 98% (2% error rate).

```
eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
# store the models for later use
DNN_numpy = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

# grid search
for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = NeuralNetwork(X_train, Y_train_onehot, eta=eta, lmbd=lmb
                            n_hidden_neurons=n_hidden_neurons, n_categ
        dnn.train()

        DNN_numpy[i][j] = dnn

        test_predict = dnn.predict(X_test)

        print("Learning rate  = ", eta)
        print("Lambda = ", lmbd)
        print("Accuracy score on test set: ", accuracy_score(Y_test, t
        print()
```

Visualization

```
# visual representation of grid search
# uses seaborn heatmap, you can also do this with matplotlib imshow
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_numpy[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
        test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()

fig, ax = plt.subplots(figsize = (10, 10))
```

scikit-learn implementation

scikit-learn focuses more on traditional machine learning methods, such as regression, clustering, decision trees, etc. As such, it has only two types of neural networks: Multi Layer Perceptron outputting continuous values, *MPLRegressor*, and Multi Layer Perceptron outputting labels, *MLPClassifier*. We will see how simple it is to use these classes.

scikit-learn implements a few improvements from our neural network, such as early stopping, a varying learning rate, different optimization methods, etc. We would therefore expect a better performance overall.

```
from sklearn.neural_network import MLPClassifier
# store models for later use
DNN_scikit = np.zeros((len(eta_vals), len(lmbd_vals)), dtype=object)

for i, eta in enumerate(eta_vals):
    for j, lmbd in enumerate(lmbd_vals):
        dnn = MLPClassifier(hidden_layer_sizes=(n_hidden_neurons), act
                            alpha=lmbd, learning_rate_init=eta, max_it
        dnn.fit(X_train, Y_train)

        DNN_scikit[i][j] = dnn

        print("Learning rate  = ", eta)
```

Visualization

```
# optional
# visual representation of grid search
# uses seaborn heatmap, could probably do this in matplotlib
import seaborn as sns

sns.set()

train_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))
test_accuracy = np.zeros((len(eta_vals), len(lmbd_vals)))

for i in range(len(eta_vals)):
    for j in range(len(lmbd_vals)):
        dnn = DNN_scikit[i][j]

        train_pred = dnn.predict(X_train)
        test_pred = dnn.predict(X_test)

        train_accuracy[i][j] = accuracy_score(Y_train, train_pred)
        test_accuracy[i][j] = accuracy_score(Y_test, test_pred)


fig, ax = plt.subplots(figsize = (10, 10))
sns.heatmap(train_accuracy, annot=True, ax=ax, cmap="viridis")
ax.set_title("Training Accuracy")
ax.set_ylabel("$\eta$")
ax.set_xlabel("$\lambda$")
plt.show()
```

Testing our code for the XOR, OR and AND gates

Last week we discussed three different types of gates, the so-called XOR, the OR and the AND gates. Their inputs and outputs can be summarized using the following tables, first for the OR gate with inputs x_1 and x_2 and outputs y:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

The AND and XOR Gates

The AND gate is defined as

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

And finally we have the XOR gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Representing the Data Sets

Our design matrix is defined by the input values x_1 and x_2. Since we have four possible outputs, our design matrix reads

$$
\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix},
$$

while the vector of outputs is $\boldsymbol{y}^T = [0, 1, 1, 0]$ for the XOR gate, $\boldsymbol{y}^T = [0, 0, 0, 1]$ for the AND gate and $\boldsymbol{y}^T = [0, 1, 1, 1]$ for the OR gate.

Setting up the Neural Network

We define first our design matrix and the various output vectors for the different gates.

```python
"""
Simple code that tests XOR, OR and AND gates with linear regression
"""

# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

def sigmoid(x):
    return 1/(1 + np.exp(-x))

def feed_forward(X):
    # weighted sum of inputs to the hidden layer
    z_h = np.matmul(X, hidden_weights) + hidden_bias
    # activation in the hidden layer
    a_h = sigmoid(z_h)

    # weighted sum of inputs to the output layer
    z_o = np.matmul(a_h, output_weights) + output_bias
    # softmax output
    # axis 0 holds each input and axis 1 the probabilities of each cat
    probabilities = sigmoid(z_o)
    return probabilities
```

The Code using Scikit-Learn

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
import seaborn as sns

# ensure the same random numbers appear every time
np.random.seed(0)

# Design matrix
X = np.array([ [0, 0], [0, 1], [1, 0],[1, 1]],dtype=np.float64)

# The XOR gate
yXOR = np.array( [ 0, 1 ,1, 0])
# The OR gate
yOR = np.array( [ 0, 1 ,1, 1])
# The AND gate
yAND = np.array( [ 0, 0 ,0, 1])

# Defining the neural network
n_inputs, n_features = X.shape
n_hidden_neurons = 2
n_categories = 2
n_features = 2

eta_vals = np.logspace(-5, 1, 7)
lmbd_vals = np.logspace(-5, 1, 7)
```

Building an object-oriented neural network code

Here we present a flexible object oriented codebase for a feed forward neural network, along with a demonstration of how to use it. Before we get into the details of the neural network, we will first present some implementations of various schedulers, cost functions and activation functions that can be used together with the neural network.

Learning rate methods. The code below shows object oriented implementations of the Constant, Momentum, Adagrad, AdagradMomentum, RMS prop and Adam schedulers. All of the classes belong to the shared abstract Scheduler class, and share the update$_c$*hange()andreset()methodsallowingforanyoftheschedulerstobeseam*
and returns the change which will be subtracted from the weights. The reset() function takes no parameters, and resets the desired variables. For Constant and Momentum, reset does nothing.

```
import autograd.numpy as np

class Scheduler:
    """
    Abstract class for Schedulers
    """
```

Testing the XOR gate and other gates

Let us now use our code to test the XOR gate.

```
X = np.array([ [0, 0], [0, 1], [1, 0],[1, 1]],dtype=np.float64)

# The XOR gate
yXOR = np.array( [[ 0], [1] ,[1], [0]])

input_nodes = X.shape[1]
output_nodes = 1

logistic_regression = FFNN((input_nodes, output_nodes), output_func=si
logistic_regression.reset_weights() # reset weights such that previous
scheduler = Adam(eta=1e-1, rho=0.9, rho2=0.999)
scores = logistic_regression.fit(X, yXOR, scheduler, epochs=1000)
```

Not bad, but the results depend strongly on the learning reate. Try different learning rates.

Building neural networks in Tensorflow and Keras

Now we want to build on the experience gained from our neural network implementation in NumPy and scikit-learn and use it to construct a neural network in Tensorflow. Once we have constructed a neural network in NumPy and Tensorflow, building one in Keras is really quite trivial, though the performance may suffer.
In our previous example we used only one hidden layer, and in this we will use two. From this it should be quite clear how to build one using an arbitrary number of hidden layers, using data structures such as Python lists or NumPy arrays.

Tensorflow

Tensorflow is an open source library machine learning library developed by the Google Brain team for internal use. It was released under the Apache 2.0 open source license in November 9, 2015. Tensorflow is a computational framework that allows you to construct machine learning models at different levels of abstraction, from high-level, object-oriented APIs like Keras, down to the C++ kernels that Tensorflow is built upon. The higher levels of abstraction are simpler to use, but less flexible, and our choice of implementation should reflect the problems we are trying to solve. Tensorflow uses so-called graphs to represent your computation in terms of the dependencies between individual operations, such that you first build a Tensorflow *graph* to represent your model, and then create a Tensorflow *session* to run the graph.
In this guide we will analyze the same data as we did in our NumPy and scikit-learn tutorial, gathered from the MNIST database of images. We will give an introduction to the lower level Python Application Program Interfaces (APIs), and see how we use them to build our graph. Then we will build (effectively) the same graph in Keras, to see just how simple solving a machine learning problem

Using Keras

Keras is a high level neural network that supports Tensorflow, CTNK and Theano as backends. If you have Anaconda installed you may run the following command

```
conda install keras
```

You can look up the instructions here for more information. We will to a large extent use **keras** in our examples..

Collect and pre-process data

Let us look again at the MINST data set.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn import datasets


# ensure the same random numbers appear every time
np.random.seed(0)

# display images in notebook
%matplotlib inline
plt.rcParams['figure.figsize'] = (12,12)


# download MNIST dataset
digits = datasets.load_digits()

# define inputs and labels
inputs = digits.images
labels = digits.target

print("inputs = (n_inputs, pixel_width, pixel_height) = " + str(inputs
print("labels = (n_inputs) = " + str(labels.shape))


# flatten the image
```

The Breast Cancer Data, now with Keras

```
import tensorflow as tf
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Sequential      #This allows appen
from tensorflow.keras.layers import Dense           #This allows defin
from tensorflow.keras import optimizers             #This allows using
from tensorflow.keras import regularizers           #This allows using
from tensorflow.keras.utils import to_categorical   #This allows using
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split as splitter
from sklearn.datasets import load_breast_cancer
import pickle
import os


"""Load breast cancer dataset"""

np.random.seed(0)        #create same seed for random number every tim

cancer=load_breast_cancer()      #Download breast cancer dataset

inputs=cancer.data                  #Feature matrix of 569 rows (sa
outputs=cancer.target               #Label array of 569 rows (0 for
labels=cancer.feature_names[0:30]

print('The content of the breast cancer dataset is:')      #Print info
print(labels)
```

Fine-tuning neural network hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network topology (how neurons/nodes are interconnected), but even in a simple FFNN you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, the stochastic gradient optmized and much more. How do you know what combination of hyperparameters is the best for your task?

- You can use grid search with cross-validation to find the right hyperparameters.

However,since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space.

- You can use randomized search.
- Or use tools like Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly.

Hidden layers

For many problems you can start with just one or two hidden layers and it will work just fine. For the MNIST data set you ca easily get a high accuracy using just one hidden layer with a few hundred neurons. You can reach for this data set above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time.
For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task.

Which activation function should I use?

The Back propagation algorithm we derived above works by going from the output layer to the input layer, propagating the error gradient on the way. Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent (GD) step.

Unfortunately for us, the gradients often get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the GD update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is known in the literature as **the vanishing gradients problem**.

In other cases, the opposite can happen, namely the the gradients can grow bigger and bigger. The result is that many of the layers get large updates of the weights the algorithm diverges. This is the **exploding gradients problem**, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients, different layers may learn at widely different speeds

Is the Logistic activation function (Sigmoid) our choice?

Although this unfortunate behavior has been empirically observed for quite a while (it was one of the reasons why deep neural networks were mostly abandoned for a long time), it is only around 2010 that significant progress was made in understanding it.
A paper titled Understanding the Difficulty of Training Deep Feedforward Neural Networks by Xavier Glorot and Yoshua Bengio found that the problems with the popular logistic sigmoid activation function and the weight initialization technique that was most popular at the time, namely random initialization using a normal distribution with a mean of 0 and a standard deviation of 1.
They showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation function saturates at the top layers. This is actually made worse by the fact that the logistic function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the logistic function in deep networks).

The derivative of the Logistic funtion

Looking at the logistic activation function, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

In their paper, Glorot and Bengio propose a way to significantly alleviate this problem. We need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction.

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to

The RELU function family

The ReLU activation function suffers from a problem known as the dying ReLUs: during training, some neurons effectively die, meaning they stop outputting anything other than 0.
In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. During training, if a neuron's weights get updated such that the weighted sum of the neuron's inputs is negative, it will start outputting 0. When this happen, the neuron is unlikely to come back to life since the gradient of the ReLU function is 0 when its input is negative.
To solve this problem, nowadays practitioners use a variant of the ReLU function, such as the leaky ReLU discussed above or the so-called exponential linear unit (ELU) function

$$ELU(z) = \begin{cases} \alpha\left(\exp\left(z\right) - 1\right) & z < 0, \\ z & z \geq 0. \end{cases}$$

Which activation function should we use?

In general it seems that the ELU activation function is better than the leaky ReLU function (and its variants), which is better than ReLU. ReLU performs better than tanh which in turn performs better than the logistic function.
If runtime performance is an issue, then you may opt for the leaky ReLU function over the ELU function If you don't want to tweak yet another hyperparameter, you may just use the default α of 0.01 for the leaky ReLU, and 1 for ELU. If you have spare time and computing power, you can use cross-validation or bootstrap to evaluate other activation functions.

More on activation functions, output layers

In most cases you can use the ReLU activation function in the hidden layers (or one of its variants).
It is a bit faster to compute than other activation functions, and the gradient descent optimization does in general not get stuck.
For the output layer:

- For classification the softmax activation function is generally a good choice for classification tasks (when the classes are mutually exclusive).
- For regression tasks, you can simply use no activation function at all.

Batch Normalization

Batch Normalization aims to address the vanishing/exploding gradients problems, and more generally the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.
The technique consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). In other words, this operation lets the model learn the optimal scale and mean of the inputs for each layer. In order to zero-center and normalize the inputs, the algorithm needs to estimate the inputs' mean and standard deviation. It does so by evaluating the mean and standard deviation of the inputs over the current mini-batch, from this the name batch normalization.

Dropout

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily dropped out, meaning it will be entirely ignored during this training step, but it may be active during the next step.
The hyperparameter p is called the dropout rate, and it is typically set to 50%. After training, the neurons are not dropped anymore.
It is viewed as one of the most popular regularization techniques.

Gradient Clipping

A popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for recurrent neural networks).
This technique is called Gradient Clipping.
In general however, Batch Normalization is preferred.

A very nice website on Neural Networks

You may find this website very useful.

A top-down perspective on Neural networks

The first thing we would like to do is divide the data into two or three parts. A training set, a validation or dev (development) set, and a test set. The test set is the data on which we want to make predictions. The dev set is a subset of the training data we use to check how well we are doing out-of-sample, after training the model on the training dataset. We use the validation error as a proxy for the test error in order to make tweaks to our model. It is crucial that we do not use any of the test data to train the algorithm. This is a cardinal sin in ML. Then:

- Estimate optimal error rate
- Minimize underfitting (bias) on training data set.
- Make sure you are not overfitting.

If the validation and test sets are drawn from the same distributions, then a good performance on the validation set should lead to similarly good performance on the test set.
However, sometimes the training data and test data differ in subtle ways because, for example, they are collected using slightly different methods, or because it is cheaper to collect data in one way versus another. In this case, there can be a mismatch between the training

Limitations of supervised learning with deep networks

Like all statistical methods, supervised learning using neural networks has important limitations. This is especially important when one seeks to apply these methods, especially to physics problems. Like all tools, DNNs are not a universal solution. Often, the same or better performance on a task can be achieved by using a few hand-engineered features (or even a collection of random features).

Here we list some of the important limitations of supervised neural network based models.

- **Need labeled data**. All supervised learning methods, DNNs for supervised learning require labeled data. Often, labeled data is harder to acquire than unlabeled data (e.g. one must pay for human experts to label images).
- **Supervised neural networks are extremely data intensive.** DNNs are data hungry. They perform best when data is plentiful. This is doubly so for supervised methods where the data must also be labeled. The utility of DNNs is extremely limited if data is hard to acquire or the datasets are small (hundreds to a few thousand samples). In this case, the