Advanced machine learning and data analysis for the physical sciences

Morten Hjorth-Jensen[1]

Department of Physics and Center for Computing in Science Education,
University of Oslo, Norway[1]

January 29, 2026

Overview of second week, January 26-30, 2026

- Mathematics of neural networks
- Writing own code (bring back to life your NN code if you have one)
- Discussion of project alternatives
- Permanent Zoom link for the whole semester is `https://uio.zoom.us/my/mortenhj`

Videos on Neural Networks

- Video on Neural Networks at
  https://www.youtube.com/watch?v=CqOfi41LfDw
- Video on the back propagation algorithm at
  https://www.youtube.com/watch?v=Ilg3gGewQ5U

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

Reminder on books with hands-on material and codes

- Sebastian Rashcka et al, Machine learning with Sickit-Learn and PyTorch
- David Foster, Generative Deep Learning with TensorFlow
- Bali and Gavras, Generative AI with Python and TensorFlow 2

All three books have GitHub addresses from where one can download all codes. We will borrow most of the material from these three texts as well as from Goodfellow, Bengio and Courville's text Deep Learning

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 $f(\boldsymbol{x}$ that is meant to describe some final result (outputs or target values bmy) given a specific input $\boldsymbol{x}$. 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 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

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

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.

Why Feed Forward Neural Networks (FFNN)? Classical approximation theorem

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**. This statement is essentially a correct summary of the classical Universal Approximation Theorem as first proven by Cybenko (1989) based on activation functions like the **sigmoid** or **tanh** functions.

Digression, more updated variant

However, the formulation in the statement, while basically technically correct, omits some details and is not the most general form of the theorem. A more precise phrasing is: for any continuous function f on a compact domain in $\mathbb{R}^n$ and any error $\varepsilon > 0$, there exists a feedforward network with one hidden layer (finite number of neurons) and a suitable activation σ (non-constant, bounded, continuous, monotonic) such that the network's output uniformly approximates f within error ε .

Getting verbose

The classical theorem's conditions on the activation function (namely **non-constant, bounded, and monotonically-increasing continuous**) are sufficient but not necessary. Cybenko chose a sigmoidal activation in his proof, and Hornik et al. (1989) independently showed a similar result for so-called **squashing** functions (which are likewise bounded and monotonic). These conditions were convenient for those proofs, often using the Stone–Weierstrass theorem or functional analysis techniques.

Updates

Later research clarified that the essential requirement is that the activation function be non-polynomial. In 1991, Hornik showed that it is not a special property of sigmoids per se, but the general two-layer network architecture that confers universality, in fact, a wide range of activation functions can lead to universal approximation.
A famous result by Leshno, Lin, Pinkus, and Schocken (1993) proved that a feedforward network is a universal approximator if and only if the activation function is not a polynomial (on any interval).

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.

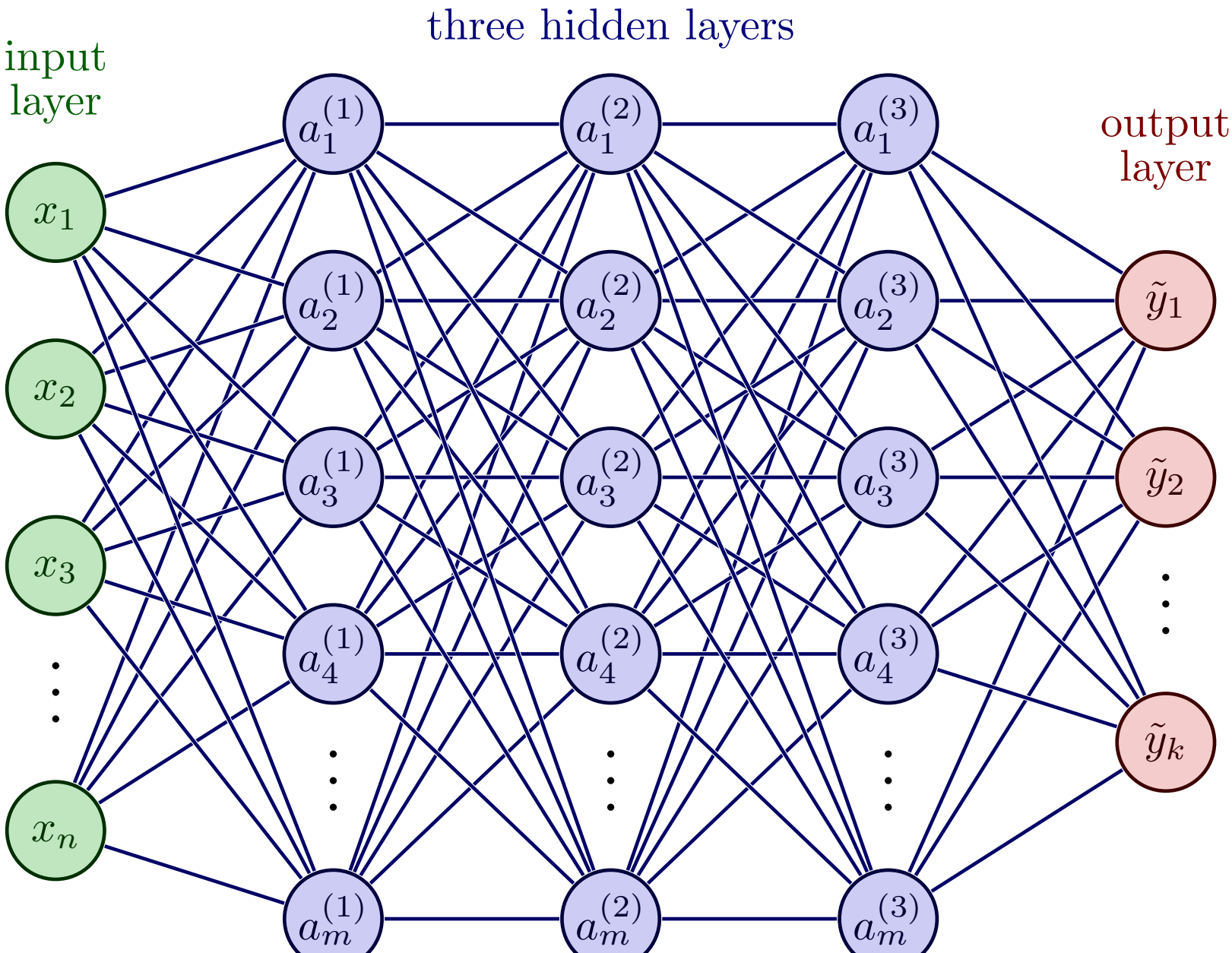
Setting up the equations for a neural network

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 $\tilde{\boldsymbol{y}}_i$.

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

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

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)

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

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

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

The derivative of this function with respect to the weights is

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

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

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

Bringing it together, first back propagation equation

We have thus

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

Defining

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

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

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

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 $f'(z_j^L)$. 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

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

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}, \tag{2}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{3}$$

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

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

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 $l = 1, 2, 3, \ldots, L$.

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

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

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

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

Setting up the Back propagation algorithm, part 3

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 1$ 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 $l = L-1, L-2, \ldots, 1$ as

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

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

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

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

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.

Insights from the paper by Glorot and Bengio

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/exploding gradients problems were in part due to a poor choice of activation function. Until then most people had assumed that if Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks, in particular the ReLU activation function, mostly because it does not saturate for positive values (and also because it is quite fast to compute).

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

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

GELU Activation Function

The **Gaussian Error Linear Unit** (GELU) activation function is defined as

$$\mathrm{GELU}(x) = x\,\Phi(x),$$

where $\Phi(x)$ is the cumulative distribution function (CDF) of the standard normal distribution,

$$\Phi(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right].$$

Approximation used in practice

In numerical implementations, GELU is often approximated by

$$\mathrm{GELU}(x) \approx \frac{x}{2}\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}\left(x + 0.044715\,x^3\right)\right]\right).$$

This form is computationally efficient and widely used in deep learning frameworks.

Key Properties

- Smooth and non-linear activation function.
- Weights inputs by their probability of being positive.
- Unlike ReLU, negative inputs are suppressed smoothly rather than set to zero.
- Empirically shown to improve performance in deep networks, especially Transformer-based models.

Activation Functions Related to GELU

These activation functions softly gate inputs rather than applying a hard threshold, improving gradient flow and optimization stability.

- Swish / SiLU: $\mathrm{Swish}(x) = x\,\sigma(x)$ (logistic-gated variant; widely used in CNNs and modern architectures)
- Mish: $x \tanh(\ln(1 + e^x))$ (smooth, non-monotonic; retains small negative values)

Related Smooth or Gated Activations

- ELU / SELU: exponential smoothing of negative inputs
- Softplus: smooth approximation to ReLU, common in probabilistic models
- PReLU: learnable slope for negative inputs (piecewise linear)

Transformer-Oriented Gated Variants

- GLU: explicit learned gating via $\sigma(Wx)$
- GeGLU: GELU-based gated linear unit
- SwiGLU: Swish-based gated linear unit (used in modern LLMs)

Why GELU-like Activations?

- Smooth derivatives (important for deep networks and PINNs)
- Probabilistic or learned gating of information
- Improved optimization compared to hard-threshold activations (ReLU)

Which activation function should we use?

In general it seems that the GELU/ELU activation functions are 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 GELU family 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 say ELU.

More on activation functions, output layers

In most cases you can use the ReLU activation function or (preferred for PINNs the GELU family) 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.

More top-down perspectives

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 and test data. This can lead to the neural network overfitting these small differences between the test and training sets, and a poor performance on the test set despite having a good performance on the validation set. To rectify this, Andrew Ng suggests making two validation or dev sets, one constructed from the training data and one constructed from the test data. The difference between the performance of the algorithm on these two validation sets quantifies the train-test mismatch. This can serve as another important diagnostic when using DNNs for supervised learning.

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).

Limitations of NNs

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 performance of other methods that utilize hand-engineered features can exceed that of DNNs.

Homogeneous data

- **Homogeneous data.** Almost all DNNs deal with homogeneous data of one type. It is very hard to design architectures that mix and match data types (i.e. some continuous variables, some discrete variables, some time series). In applications beyond images, video, and language, this is often what is required. In contrast, ensemble models like random forests or gradient-boosted trees have no difficulty handling mixed data types.

More limitations

- **Many problems are not about prediction.** In natural science we are often interested in learning something about the underlying distribution that generates the data. In this case, it is often difficult to cast these ideas in a supervised learning setting. While the problems are related, it is possible to make good predictions with a *wrong* model. The model might or might not be useful for understanding the underlying science.

Some of these remarks are particular to DNNs, others are shared by all supervised learning methods. This motivates the use of unsupervised methods which in part circumvent these problems.

Codes

We present here three different possibilities. The first one is based on own codes, the second alternative uses TensorFlow with Keras, while the last version is a simple PyTorch implementation.

Building a 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.

The codes here were developed by Eric Reber and Gregor Kajda during spring 2023.

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-change() and reset() methods allowing for any of the schedulers to be seamlessly used during the training stage, as will later be shown in the fit() method of the neural network. Update-change() only has one parameter, the gradient, 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

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.

Using Pytorch with the full MNIST data set

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

# Device configuration: use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# MNIST dataset (downloads if not already present)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # normalize to mean=0.5, std
])
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
test_dataset  = torchvision.datasets.MNIST(root='./data', train=False,

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=6
test_loader  = torch.utils.data.DataLoader(test_dataset, batch_size=64


class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 100)   # first hidden layer (784 -
        self.fc2 = nn.Linear(100, 100)    # second hidden layer (100 -
        self.fc3 = nn.Linear(100, 10)     # output layer (100 -> 10 cl
    def forward(self, x):
        x = x.view(x.size(0), -1)          # flatten images into vector
```

And a similar example using Tensorflow with Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers

# Check for GPU (TensorFlow will use it automatically if available)
gpus = tf.config.list_physical_devices('GPU')
print(f"GPUs available: {gpus}")

# 1) Load and preprocess MNIST
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data(
# Normalize to [0, 1]
x_train = (x_train.astype("float32") / 255.0)
x_test  = (x_test.astype("float32") / 255.0)

# 2) Build the model: 784 -> 100 -> 100 -> 10
l2_reg = 1e-4  # L2 regularization strength

model = keras.Sequential([
    layers.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(100, activation="relu",
                 kernel_regularizer=regularizers.l2(l2_reg)),
    layers.Dense(100, activation="relu",
                 kernel_regularizer=regularizers.l2(l2_reg)),
    layers.Dense(10, activation="softmax")  # output probabilities for
])

# 3) Compile with SGD + weight decay via L2 regularizers
```