DAT300 CHAPTER 12 - Implementing a Multilayer Artificial Neural Network from Scratch

Jorid Holmen - Applied Robotics, Norwegian University of Life Sciences

Deep learning can be understood as a subfield of machine learning that is concerned with training artificial neural networks (NNs) with many layers efficiently.

Key words in this chapter is:

- Single-layer neural network
- Multilayer perceptron (MLP)
- Forward propagation
- Logisitic cost function
- Backpropagation
- convergence in neural networks
- Loss functions
- Activation functions

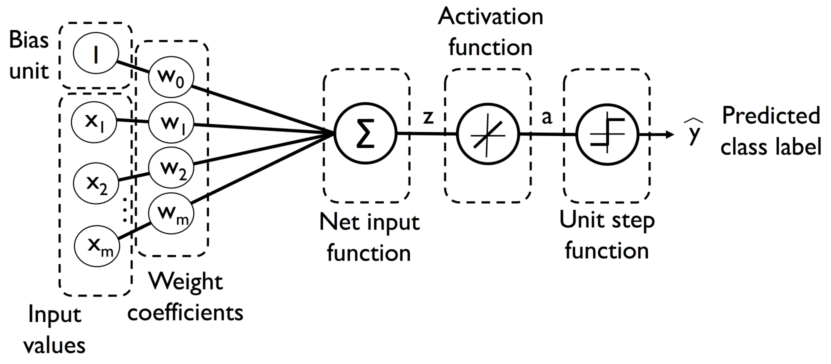
Important variables:

- w: weight vector
- $J(w)$: cost function
- η: learning rate
- $\Phi()$: activation function
- z: net input
- l: layer
- a: activation unit

1 Modeling complex functions with artificial NNs

Artificial neurons represent the building blocks of the multilayer artificial NNs that will be discussed in this chapter. The basic consept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problem tasks. NNs that are composed of many layers is called deep learning algorithms and architectures.

1.1 Single-layer neural network



To better understand multilayer NNs, you should understand single-layer neural networks such as Adaline and perceptron. Adaline has a net input

$$z = \sum_j w_j x_j = \mathbf{w^T x}. \tag{1}$$

For adaline the **activation function** simplifies the learning algorithm and allows for straightforward weight updates based on the linear relationship between the inputs and the target outputs. A bias unit is incorporated into the network to allow the activation function to shift left or right, facilitating more accurate function approximations. Adaline uses a linear function as activation function.

$$\phi(z) = z = a. \tag{2}$$

For reference, the activation function for logsistic regression is

$$\phi(z) = \frac{1}{1 + e^{-z}} = a. \tag{3}$$

The threshold for adaline is

$$\hat{y} = \begin{cases} 1 \text{ if } g(z) \geq 0 \\ -1 \text{ otherwise} \end{cases} \tag{4}$$

The weights are updated through gradient descent. The key feature of gradient descent is to find the minimum of a function. The gradient is based on the whole training set, and weights take a step opposite to the gradient.

$$\mathbf{w} := \mathbf{w} + \mathbf{\Delta w}, \text{where} \mathbf{\Delta w} = -\eta \nabla J(\mathbf{w}) \tag{5}$$

$$J(\mathbf{w}) = \mathbf{\frac{1}{2} \sum_{i=1}^{n} (y^{(i)} - \phi(z^{(i)}))^2} \tag{6}$$

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\mathbf{\sum_{i} (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}} \tag{7}$$

y: target label
x: sample
η: learning rate

In other words, we compute the gradient descent based on the whole training dataset and update the weights (the weight vector) of the model by taking a step in the opposite direction of the gradient. This is done in every epoch. In order to find the optimal weights of the model, we optimized an objective function **sum of squared errors (SSE)**, as the cost function $J(w)$. In addition we multiplied the gradient by the **learning rate** η. Furthermore we defined the **activation function** $\mathbf{\Phi()}$, that uses the **net input** z. Lastly we use the activation function to implement a threshold function to perform the binary classification.

Optimisation is an important part of adaline, and a type of optimization is gradient descent. We also use the **stochastic gradient descent (SDG)** optimization to accelerate the model learning. Gradient descent uses all training samples for each update, while stochastic gradient descent uses one training sample per update. In other words, SGD uses random, single sample update or random subset update, instead of using the entire dataset. It learns faster, but also has a noisy nature, that luckily is beneficial when training multilayer NNs with nonlinear activation functions. The added noise will help to escape the local cost minima. More about optimization can be found later.

1.2 Introducing the multilayer neural network architecture

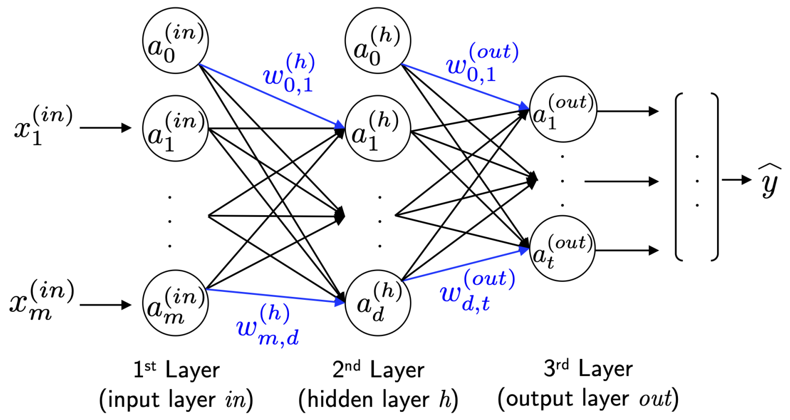
Training neural networks revolves around **layers**, the **input data** and the **corresponding data**. The **loss founction** defines the feedback signal used for learning and the **optimizer** determines how learning proceeds.

Layers are a fundemental data structure in neural networks. A layer is a data-processing module that takes one or more tensors as input and gives one or more tensors as output. A tensor is a container for data. The layers weights contain the networks knowledge.

A deep-learning model is a graph of layers, also called a network of layers. The most common instance is a linear stack of layers, mapping a single input to a single output. Some common network topologies are two-branch networks, multihead networks and inception blocks. The **topology** of network defines the **hypothesis space**. The hypothosis space is the range of functions and models the algorithm is capable of selecting as the solution to a leanring problem. By choosing a network topology, you constrain your space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data. Picking the right network architecture is more an art than a science. Although there are some practices and principles you can rely on, only practice can help you become a proper neural-network architect.

1.2.1 Multilayer perceptron

Multilayer perceptron (MLP) is a fully connected network with multiple single neurons connected to a multilayer feedforward neural network (NN). The primary advantage to using multilayer perceptron over single-layer perceptron, is the ability to model non-linear relationships.



$x_i^{(in)}$: the ith input feature value
$a_i^{(in)}$: the ith unit in the input layer.
$a_i^{(h)}$: the ith unit in the hidden layer.
$a_i^{(out)}$: the ith unit in the output layer.
$a_i^{(l)}$: the activation unit i in l-th layer.
$w_{i,j}^{(l)}$: weight connecting unit i in $l-1$-th layer with unit j in l-th layer. These will be grouped into a matrix $W^{(l)}$ later.

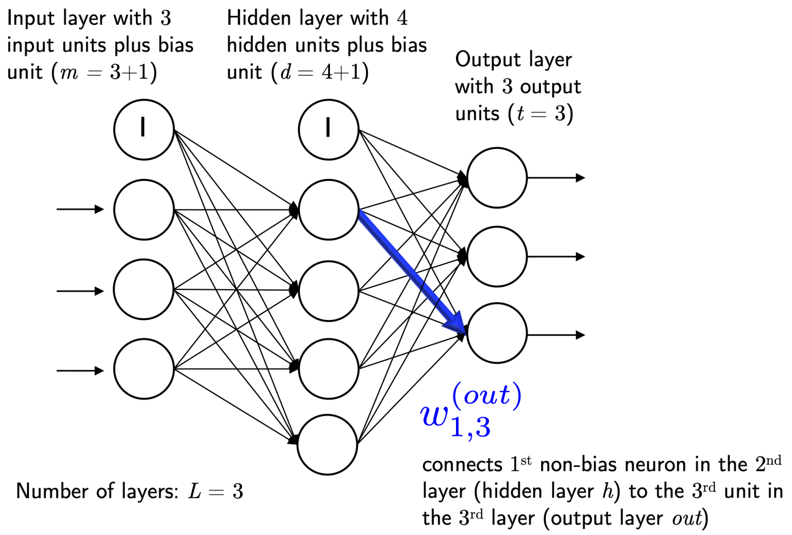
An MLP consist of one input layer, one or more hidden layers and an output layer. The units in the hidden layer is fully conncted to the input layer, and the ouput layer is fully connected to the hidden layer. If such a network has more than one hidden layer, we also call it **deep artificial NN**. A key characteristic of MLP is the multiple hidden layers. We can add any number of hidden layers to the MLP to create deeper network architectures. Practically we can think of the number of hidden layers and units in an NN as additional hyperparameters that we want to optimize for a given problem task.

More layers give higher flexibility, but also increases the possibility of overfitting, in addition to increasing the number of parameters to estimate. The error gradient, which we will calculate later using backpropagation, also becomes increasingly small as more layers are added to the network. This leads to the vanishing gradient problem. More on the vanishing gradient later.

Here, the activation units $a_0^{(in)}$ and $a_0^{(h)}$ are the bias units, which we set equal to 1. The bias unit allows the model neural network to represent functions that does not necessarily pass through the origin. The activiation of the units in the input layer is just input plus the bias unit:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix} \tag{8}$$

Each unit in layer l is connected to all units in layer $l+1$ via a weight coefficient. The connection between the kth unit in layer l to the jth unit in layer $l+1$ will be written as $w_{k,j}^{l}$. We write the matrix that connects the hidden layers to the output layer as $W^{(out)}$.


Input layer with 3 input units plus bias unit (m = 3+1)
Hidden layer with 4 hidden units plus bias unit (d = 4+1)
Output layer with 3 output units (t = 3)
1
1
$w_{1,3}^{(out)}$
connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer out)
Number of layers: L = 3

While one unit in the output layer would suffice for a binary classification task, we saw a more general form of NN in the preceding figure, which allows us to perform multiclass classification via generalization of the **one-versus-all (OvA)** technique. For example, we can encode three class labels like this:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{9}$$

This one-hot vector representation allows us to tackle classification with an arbitraty number of unique class labels present in the training set.

1.3 Activating a neural network via forward propagation

This section will describe the process of **forward propagation** to calculate the output of an MLP model. The MLP learning procedure can be summarized in three simple steps:

1. Starting at the input layer, we **forward propagate** the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the **error** that we want to minimize using a **cost function** that we will describe later.
3. We **backpropagate** the error, finds its derivative with respect to each weight in the network, and update the model.

Finally after we repeat these three steps for multiple epochs and learn the weights of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in a one-hot representation.

Forward propagation generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layer, we start by calculating the activiation function of the hidden layer a_1^h:

$$z_1^h = a_0^{in} w_{0,1}^h + a_1^{in} w_{1,1}^h + \cdot + a_m^{in} w_{m,1}^h = ainW^h \tag{10}$$

$$a_1^h = \phi(z_1^h) \tag{11}$$

Here z_1^h is the net input and $\phi(z_1^h)$ is the **activiation function**, which has to be differentiable to learn the weights that connect the neurons using a gradient based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP, for example the sigmoid (logisitic) activation function:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{12}$$

We can generalize the net input from the activiation function to all n examples in the training set:

$$Z^{(h)} = A^{(in)} W^h \tag{13}$$

Here $A^{(in)}$ is an n x m matrix, and the matrix-matrix multiplication will result in an n x m dimensional net input matrix $Z^{(h)}$, where d is the number of units in the hidden layer. This results in

$$A^{(h)} = \phi(Z^{(h)}) \tag{14}$$

Similarly, we can write the activation of the output layer in vectorized form for multiple examples:

$$Z^{(out)} = A^{(h)} W^{out} \tag{15}$$

Lastly we apply the sigmoid activation function to obtain the continuous values output of our network:

$$A^{(out)} = \phi(Z^{(out)}) \tag{16}$$

More on the choice of activation functions in a later section.

2 Training an artificial neural network

Here we will go deeper into the cost function, backpropagation and convergence.

2.1 Computing the logistic cost function

The logistic cost function that is often implemented is as follows:

$$J(w) = -\sum_{i=1}^{n} y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) \tag{17}$$

Here $a^{[i]}$ is the sigmoid activiation function of the ith sample in the dataset, which we compute in the forward propagation step.

$$a^{[i]} = \phi(z^{[i]}) \tag{18}$$

$[i]$ is an index for training examples, not layers.

Now lets add a regularization term, which allows us to reduce the degree of overfitting. Overfitting happens if we have large weights. It regularizes all weights expect for the bias units for the node. The L2 regularization term is added to the cost function to penalize large weights, and encouraging small weights. We can train without regularization, but we usually want to add it. It is defined as follows:

$$L2 = \lambda ||w||_2^2 = \lambda \sum_{i=1}^{m} w_j^2 \tag{19}$$

By adding the L2 regularization to the logistic cost function, we get:

$$J(w) = -\sum_{i=1}^{n} y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) + \frac{\lambda}{2}||w||_2^2 \tag{20}$$

We need to generalize the cost function to all t activation units in our network. t **is the elements in the output vector of the MLP**. We need to do this, because the output, y, is a vector and not just a single number, as it is in simple machine learning.

The cost function (without the regularization term) becomes the following:

$$J(w) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]})\right]. \tag{21}$$

The cost function with a generalized regularization term becomes:

$$J(w) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]})\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}(w_{j,i}^{(l)})^2 \tag{22}$$

l: the given layer
u_l: the number of units in a given layer l

The cost minimization will be performed using the partial derivative of the paramters of W with respect to each weight of the layer:

$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(W) \tag{23}$$

In the context of optimization algorithms (such as gradient descent), the term "cost function" is often used synonymously with the loss function, especially when referring to the objective that the algorithm aims to minimize. While technically the cost function can encompass additional components (like regularization terms), in practice, the primary focus is on minimizing the loss function during training.

2.2 Developing your understanding of backpropagation

In essence we can think of backpropagation as a very computationally efficient approach to compute the partial derivatives of a complex cost function in multilayer NNs, to find the error. The goal is to use those derivaties to learn the weight coefficients to find the parameters for such multilayer artificial NNs. The challenge in the parameterization of NNs is that we are typically dealing with very large numbers of weight coefficients in a high-dimensional feature space. Backpropagation is still one of the most popular algorithms to train ANNs.

In contrast to cost functions of single-layer NNs, such as Adaline or logistic regression, the error surface of an NN cost function is not convex or smooth with respect to the parameters. There are many bumps in this high-dimensional cost surface (local minima) that we have to overcome in order to find the global minimum of the cost function.

2.2.1 The chain rule

The chain rule is an approach to compute the derivative of a complex nested function:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx} \tag{24}$$

Lets assume we have five different functions and let F be the composition of these five functions:

$$F(x) = f(g(h(u(v(x))))) \tag{25}$$

Applying the chain rule, we can compute the derivative of this function as follows:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}[f(g(h(u(v(x)))))] = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx} \tag{26}$$

The derivative of the sigmoid function:

$$\begin{aligned} &\phi'(z) = \tfrac{\partial}{\partial z}\left(\tfrac{1}{1+e^{-z}}\right) = \tfrac{1}{1+e^{-z}} - \left(\tfrac{1}{1+e^{-z}}\right)^2 \\ &= \phi(z) - (\phi(z))^2 = \phi(z)(1-\phi(z)) = a(1-a) \end{aligned} \tag{27}$$

The chain rule is used for calculating the gradient for the weight updates.

2.2.2 Automatic differentiation

We can use the chain rule to compute the derivative of a complex, nested function. In the context of computer algebra, a set of techniques has been developed to solve such problems very efficiently, which is known as **automatic differentiation**. Automatic differentiation comes with two modes, the forward and the reverse modes. **Backpropagation is simply a special case of a reverse-mode automatic differentiation**. The key point is that applying the chain rule in the forward mode could be quite expensive since we could have to multiply large matrices for each layer that we could eventually multiply by a vector to obtain the output.

The trick of reverse mode is that we start from the right to the left: we multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix and so on. Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, and therefore backpropagation is the most popular algorithms used in NN training.

2.3 Training your neural network via backpropagation

In this section we will go through the math of backpropagation to understand how you can learn the weights in an NN very efficiently. In the code this is included in the `fit` method.

As we recall from the beginning of the chapter, we first need to apply forward propagation in order to obtain the activation of the output layer, which we formulated as follows:

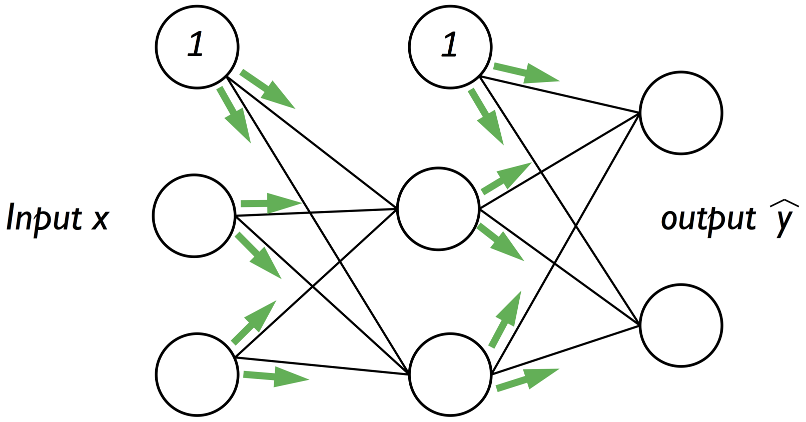
$$Z^{(h)} = A^{(in)}W^{(h)} \tag{28}$$

$$A^{(h)} = \phi(Z^{(h)}) \tag{29}$$

$$Z^{(out)} = A^{(h)}W^{(out)} \tag{30}$$

$$A^{(out)} = \phi(Z^{(out)}) \tag{31}$$

In other words, we just forward-propagate the input features through the connection in the network, as shown in the illustration:



In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

$$\delta^{(out)} = a^{(out)} - y \tag{32}$$

y: the vector of the true class labels (in `NeuralNetMLP`, the corresponding variable is `delta_out`).

Next, we calculate the $\delta^{(h)}$ (`delta_h`) layer error matrix of the hidden layer:

$$\delta^{(h)} = \delta^{(out)}(W^{(out)})^T \odot (a^{(h)} \odot (1 - a^{(h)})) \tag{33}$$

After obtaining the δ terms we can find the derivation of the cost function as follows:

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J(W) = a_j^{(h)} \delta_i^{(out)} \tag{34}$$

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J(W) = a_j^{(in)} \delta_i^{(h)} \tag{35}$$

Next we need to accumulate the partial derivative of every node in each layer and the error of the node in the next layer. Remember that we need to compute $\Delta_{i,j}^{(l)}$ for every sample in the training dataset. Thus it is easier to implement it as a vectorized version:

$$\Delta^{(h)} = (A^{(in)})^T \delta^{(h)} \tag{36}$$

$$\Delta^{(out)} = (A^{(h)})^T \delta^{(out)} \tag{37}$$

After we have accumulated the partial derivatives, we can add the following regularization term:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} W^{(l)} \tag{38}$$

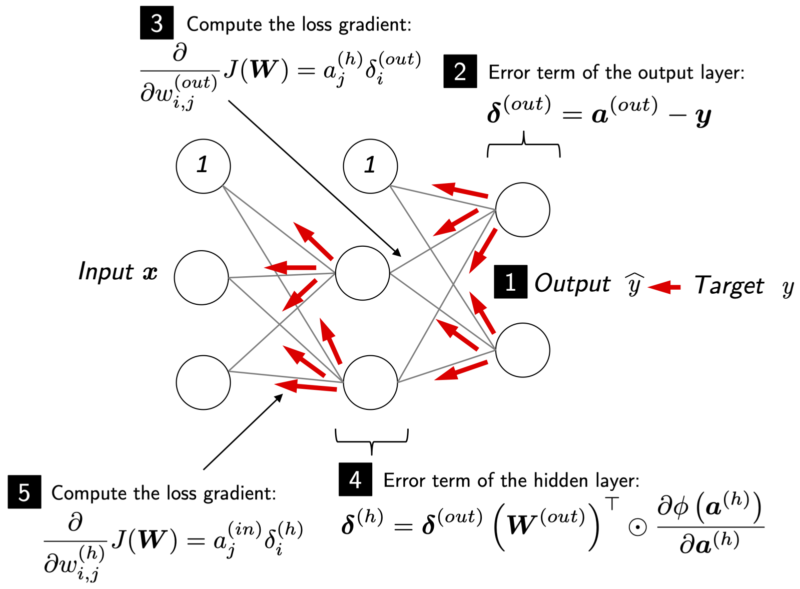
Lastly, after computing the gradients, we can update the weights by taking an opposite step towards the gradient for each layer l:

$$W^{(l)} := W^{(l)} - \eta \Delta^{(l)} \tag{39}$$

The only thing that really matters is that we remember that when we are doing backpropagation we take the derivative of the non-linear activation function.

Gradients represents how much a small change in a parameter (such as weights) affects the output of a function (such as the loss function). When you calculate the gradient during backpropagation, it involves multiplying the error (a measure of how far the networks prediction is from the actual target) by the weights. This multiplication represents how much the weights contributed to the error in the output.

To bring everything together, backpropagation can be summarized in to one figure:

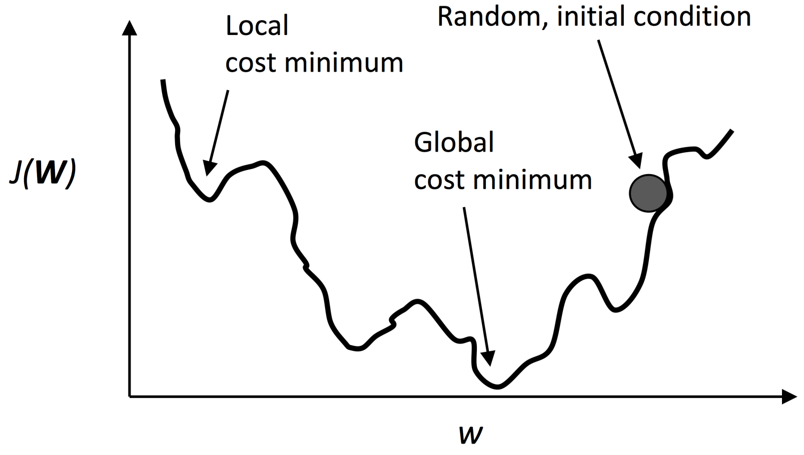

3 Compute the loss gradient:
$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\boldsymbol{W}) = a_j^{(h)} \delta_i^{(out)}$
2 Error term of the output layer:
$\boldsymbol{\delta}^{(out)} = \boldsymbol{a}^{(out)} - \boldsymbol{y}$
1
1
Input $\boldsymbol{x}$
1 Output $\hat{y}$
Target y
5 Compute the loss gradient:
$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\boldsymbol{W}) = a_j^{(in)} \delta_i^{(h)}$
4 Error term of the hidden layer:
$\boldsymbol{\delta}^{(h)} = \boldsymbol{\delta}^{(out)} \left(\boldsymbol{W}^{(out)}\right)^{\top} \odot \frac{\partial \phi\left(\boldsymbol{a}^{(h)}\right)}{\partial \boldsymbol{a}^{(h)}}$

1. During forward propagation, inputs are passed through the network, and the output is calculated.
2. The loss function measures how far the network's output is from the actual target.
3. During backpropagation, gradients are calculated by taking derivatives of the loss function with respect to each weight.
4. The gradient indicates the direction and magnitude of the change needed in each weight to reduce the loss.
5. To compute these gradients, the chain rule is applied, involving the multiplication of error terms and weights as the derivative components. Multiplying the error by the weights is part of calculating the gradient and helps distribute the contribution of each weight to the error.

3 About the convergence in neural networks

The purpose of gradient descent is to update the network weights based on the error. You can use mini-batch learning instead of regular gradient descent to train an NN. Mini-batch learning is a special form of stochastic gradient descent (SGD) where we compute the gradient based on a subset k of the n training examples with $1 < k < n$, instead of computing the gradient descent based on a single training example at a time. Mini-batch makes use of the vectorized implementation to improve computational efficiency. Imagine a presidential election where we ask a representative of a subset of the population to vote, instead of asking everyone.

Multilayer NNs are much more harder to train than simpler algorithms such as Adaline, logisistic regression or support vector machines. In multilayer NNs we typically have hundreds, thousands or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface and the optimization algorithm can easily become trapped in local minima, as shown in the figure.



Note that this representation is extremely simplified since our NN has many dimensions, and that makes it impossible to visualize for the human eye. This is the cost surface for a single weight on the x-axis. The point is that we do not want our algorithm to be trapped on a local minima, which we can avoid by **increasing the learning rate**. However, if the learning rate is to big we risk overshooting the global optimum.

Deep learning brings adaptive learning rate, learning rate with decay, learning rate with restarts, amongst other to help with this problem.

4 Loss functions

A loss function, also known as an error function or objective function, measures how well a machine learning model's predictions match the true (target) values of the data. The goal of a loss function is to quantify the difference between the predicted output of the model and the actual target output for a given input example. Typically, a lower loss value indicates better performance of the model on the training data. The loss function is used during both training (to optimize the model's parameters) and evaluation (to assess the model's performance).

NB! In the context of training a Multi-Layer Perceptron (MLP), typically only the term "loss function" is commonly used to refer to the function that *measures the error or discrepancy between the model's predictions and the true target values for a given set of input data. The concept of a "cost function" may not be explicitly distinguished as a separate entity in the terminology related to MLPs, but the ideas behind these terms are closely related. - ChatGPT*

Choosing the right loss/objective function for the right problem is extremely important: the network will take any shortcut it can, to minimize the loss. If the objective does not fully correlate with the success for the task at hand, your network will end up doing things you may not have wanted.

For problems such as classification, regression and sequence predicition, there are simple guidelines one can follow to choose the correct loss function:

- Binary crossentropy for a two-class classification problem
- Categorical cross entropy for a many-class classification problem
- Mean squared error for a regression problem
- Connection temporal classification (CTC) for sequence-learning problems
- etc.

4.1 Cross entropy

Cross entropy loss, or log loss, measures the performance of a classification model whose output represents a probability, that is, a value between 0 and 1. Cross entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of for example .017 when the actual observation label is 1 would result in a high loss value. A perfect model would have a log loss of 0.

You can choose binary cross entropy, for binary problems, or categorical entropy, for multiclass problems.

Binary cross entropy:

$$D_{binary}(\hat{y}, y) = -(y \log(\hat{y}) + (1-y)\log(1-\hat{y})) \tag{40}$$

Categorical cross entropy:

$$D_{categorical}(\hat{y}, y) = -\sum_{i=1}^{K} y_i \log(\hat{y}_i) \tag{41}$$

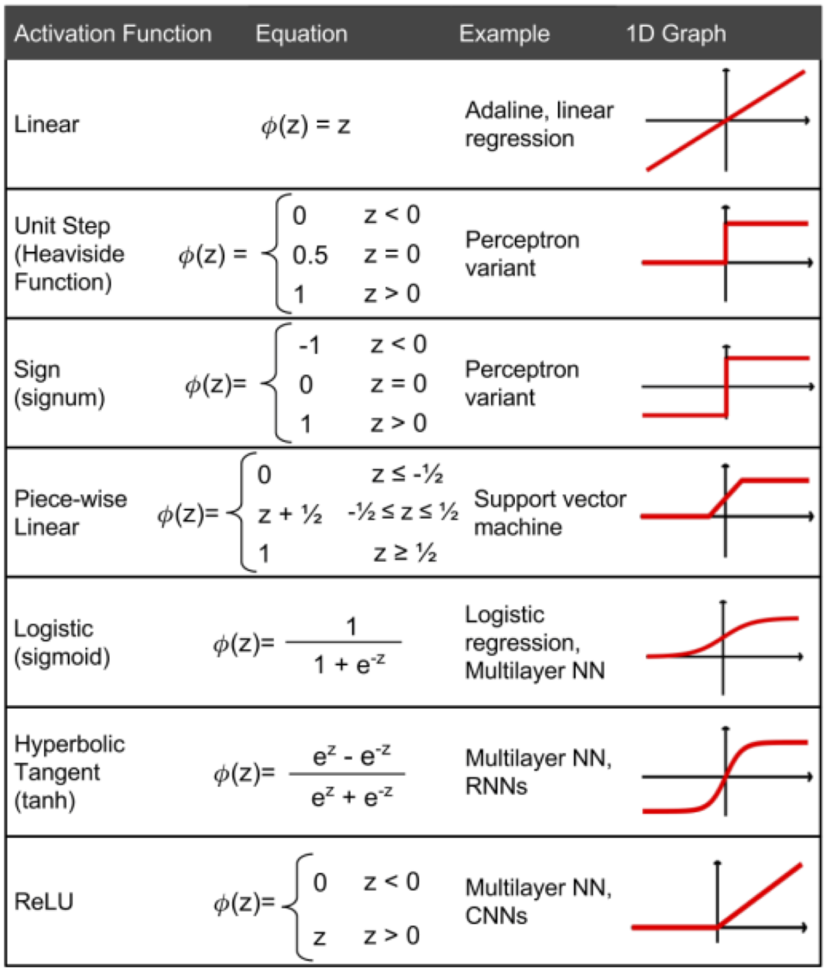
5 Choosing activation functions for multilayer neural networks

The activation function is applied in every layer, except for the input layer, in the NN and the task is to introduce non-linearity.

Technically, we can use any activation function in multilayer NNs as long as it is differentiable. We could also use linear acitivation functions, but it would not be very usefull to use on both hidden and output layers, since we want nonlinearity in typical artificial NN to be able to tackle complex problems. After all, the sum of linear functions yields a linear function.

Activation Function	**Pros**	**Cons**
Linear	Simple, good for regression	No non-linearity, cannot handle complex data
Unit Step	Simple, useful in binary classification	Not differentiable, poor for gradient-based learning
Sign	Easy to compute	Non-differentiable, limits model complexity
Piece-wise Linear	Used in specific algorithms (SVMs)	More complex, limited use in neural networks
Logistic (Sigmoid)	Smooth output, good for probabilities	Vanishing gradient problem, slow convergence
Tanh (Hyperbolic Tangent)	Zero-centered output, stronger gradient	Suffers from vanishing gradients
ReLU	Efficient, fast convergence, avoids vanishing gradients	Can "die" for negative inputs, not zero-centered

Table 1: Comparison of Activation Functions



Activation Function	Equation	Example	1D Graph
Linear	$\phi(z) = z$	Adaline, linear regression	
Unit Step (Heaviside Function)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Sign (signum)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Perceptron variant	
Piece-wise Linear	$\phi(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multilayer NN	
Hyperbolic Tangent (tanh)	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multilayer NN, RNNs	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Multilayer NN, CNNs	

5.1 The logistic function

The logisitic activation function, often called sigmoid function, probably mimics the concept of a neuron in the brain most closely. Think of it as the probability of wether a neuron fires or not. However, logistic activation functions can be problematic when the net input z is hugely negative, $\phi(s)$ would be close to zero. If $\phi(s)$ is close to zero the neural netowrk would learn very slowly. More slowly learning could lead to the neural network getting trapped in local minima during training. We can use a logistic function to model the probability that sample x belongs to the positive class in a binary classification.

The logistic (sigmoid) function will compute the following:

$$\phi_{logisitc}(z) = \frac{1}{1 + e^{-z}}, \tag{42}$$

where z is the net input of the weights and samples.

An output layer constisting of multiple logisitic activation units does not produce meaningful, interpretable probability values. The reason for this is that they do not sum up to 1. Usually this is not of concern when we use the model to predict class membership. One way to predict class membership is to assign a sample to the maximum value of z.

5.2 The softmax function

The `softmax` function is a generalisation of the logisitc function; instead of giving a single class index, it provides the probability of each class. Therefore, it allows us to compute meaningful class probabilities in multiclass settings.

If you have the scores 2.0, 1.0 and 0.1, the softmax will convert into the propability, which is 0.7, 0.2 and 0.1, which adds up to 1. We pick the highest probability, and one hot encode into 1, 0 and 0.

In `softmax`, the probability of particular sample with net-input z belonging to the ith classs can be computed with a normalization term in the denominator, that is, the sum of the exponentially weighted linear functions:

$$p(z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}} \tag{43}$$

It may help to think of the results of the `softmax` function as a *normalized* output that is useful for obatining meaningful class-membership predictions in multiclass settings.

5.3 The hyperbolic tangent (tanh)

Another sigmoidal function that is often used in the hidden layers of artificial NNs is the hyperbolic tangent (tanh), which can be interpreted as a rescaled version of the logistic function:

$$\phi_{tanh}(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{44}$$

The hyperbolic tangent has an output in the interval (-1, 1), while the logistic function only range in the interval (0,1). This means hyberbolic tangent can improve the convergence of the backpropagating algorithm.

5.4 Rectified linear unit activation (ReLU)

Rectified linear unit (ReLU) is another activation function that is often used in deep NNs.

Tanh and logistic activations has a problem called **vanishing gradient**, which will soon be discussed.

ReLu addresses this issue and is defined as follows:

$$\phi(z) = \max(0, z) \tag{45}$$

ReLu is a nonlinear function that is good for learning complex functions with NNs. Besides this, the derivative of ReLu, with respect to the input, is always 1 for positive input values. Therefore, it solves the problem of vanishing gradients, making it suitable for deep neural netowrks.

5.5 Swish activation

The Swish activation function is defined as follows:

$$\sigma(x) = x \cdot sigmoid(x) = \frac{x}{1 + e^{-x}} \tag{46}$$

where σ is the sigmoid activation function and β is a learnable parameter.

In mathematical terms, the sigmoid function introduces a smoothness to the Swish activation, and the β parameter controls the shape of the function. Swish has been reported to perform well in various deep learning tasks, often outperforming other activation functions like ReLU. It is believed to combine some of the favorable properties of both sigmoid and ReLU activations.

The Swish activation function tends to allow more expressive power in the network due to its non-monotonicity, which can help the optimization process during training. However, it's worth noting that the effectiveness of activation functions can depend on the specific characteristics of the task at hand, and empirical testing is often necessary to determine the most suitable activation function for a given scenario.

5.6 The vanishing gradient problem

Backpropagation worked well with relatively shallow networks (one or two layers of hidden units (1980s)), but as the networks got deeper, the networks either took an inordinate amount of time to train, or else they entirely failed to converge on a good set of weights. The use of backpropagation to update weights across many layers leads to vanishing gradient problem and the root of the problem is that the gradient of a given layer is the product of gradients at previous layers.

5.6.1 Chain rule leading to vanishing gradient problem

Fundamentally, the backpropagation algorithm is an implementation of the chain rule from calculus. The chain rule involves the multiplication of terms. Backpropagating an error from one neuron back to another can involve multiplying the error by a number terms with values less than 1. These multiplications by values less than 1 happen repeatedly as the error signal gets passed back through the network. This results in the error signal becoming smaller and smaller as it is backpropagated through the network. Indeed, the error signal often diminishes exponentially with respect to the distance from the output layer. The effect of this diminishing error is that the weights in the early layers of a deep network are often adjusted by only a tiny (or zero) amount during each training iteration. In other words: the early layers either train very, very slowly or do not move away from their random starting positions at all.

However, the early layers in a neural network are vitally important to the success of the network. It is the neurons in these layers that learn to detect the features in the input. Later layers of the network use detected features as the fundamental building blocks of the representations. These building blocks ultimately determine the output of the network. The error signal that is backpropagated through the network is in fact the gradient of the error of the network. This problem of the error signal rapidly diminishing to near zero is known as the vanishing gradient problem.

5.7 When to use which activation function

The **input layer** does not just holds the input data and no calculations is performed. Therefore, no activation function is used there.

In the **hidden layer** the typical activation functions are sigmoid, tanh and ReLu.

In the **output layer** there are different activation functions for different tasks.

For regression you want to predict a continuous value, and therefore you would typically use a linear activation function such as $f(x) = x$.

For a binary classification task where you only want to predict to classes, the output layer should hold a sigmoid activation function. This means the output value will be between 0 and 1, representing the probabilbity that the input belongs to 0 or 1.

For a multiclass classification where you want to predict among multiple classes (e.g., cat, dog, bird), the output layer typically uses a softmax activation function. Softmax normalizes the outputs into a probability distribution across multiple classes, ensuring that the sum of probabilities for all classes equals 1.

6 Mathematics in ANN

The main part of mathematics in a neural network is calculating the amount of paramters.

For example, if you have a neural network with 100 inputs, and three dense hidden layers with 64, 32 and 10 neurons, we will get

$$\text{amount of parameters} = 100 * 64 + 64 + 64 * 32 + 32 + 32 * 10 + 10 = 8874 \quad (47)$$

If you have 2000 samples and the batch size is 500, it will take 4 iterations to do one epoch. For 5 epochs it will be 20 iterations.

i is the amount of samples, and m is the amount of features. The dimension for $a^{(in)} is$m x 1 + bias.

d is the amount of units in a hidden layer. The dimension of the weight matrix for this hidden layer is m x d, and the bias unit is 1 x d.

t is the amount of units in the output layer. The dimension of bias going from hidden layer to the output layer is 1 x t, and the weights are d x t.

For n samples $z^{(h)}$ has the dimensions n x d, where d still is the amount of neurons in the hidden layer. The dimensions for $a^{(h)}$ is also n x d.

For n samples $z^{(out)}$ has the dimensions n x t, where t still is the amount of units in the output layer. The dimensions for $a^{(out)}$ is also n x t.

7 Usefull pages and videos

Neural Networks Explained in 5 minutes - IBM Technology

Playlist from StatQuest with Josh Starmer - video 1-13

What are MLPs (Multilayer Perceptrons)? - IBM Technology

Playlist from 3Blue1Brown - video 1-4

Dive into deep learning - online book