Learning from data: Neural networks, from the simple perceptron to deep learning

Christian Forssén[1]

Morten Hjorth-Jensen[2,3]

[1]Department of Physics, Chalmers University of Technology, Sweden
[2]Department of Physics, University of Oslo
[3]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Oct 15, 2019

1 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.

1.1 Terminology

Each time we describe a neural network algorithm we will typically specify three things.

Architecture: The architecture specifies what variables are involved in the network and their topological relationships – for example, the variables involved in a neural net might be the weights of the connections between the neurons, along with the activities of the neurons.

Activity rule: Most neural network models have short time-scale dynamics: local rules define how the activities of the neurons change in response to each other. Typically the activity rule depends on the weights (the parameters) in the network.

Learning rule: The learning rule specifies the way in which the neural network's weights change with time. This learning is usually viewed as taking place on a longer time scale than the time scale of the dynamics under the activity rule. Usually the learning rule will depend on the activities of the neurons. It may also depend on the values of target values supplied by a teacher and on the current value of the weights.

1.2 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.

$$y = f\left(\sum_{i=1}^{n} w_j x_j + b\right) = f(z), \tag{1}$$

where the bias b is sometimes denoted w_0. Here, the output y of the neuron is the value of its activation function, which have as input a weighted sum of signals $x_1, \dots, x_n$ received by n other neurons.

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

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

In natural science, DNNs and CNNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum physics. Various quantum phase transitions can be detected and studied using DNNs and CNNs: topological phases, and even non-equilibrium many-body localization.

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

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

1.3 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.

1.4 Multilayer perceptrons

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

1. a neural network with one or more layers of nodes between the input and the output nodes.
2. the multilayer network structure, or architecture, or topology, consists of an input layer, one or more hidden layers, and one output layer.
3. the input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

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

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

The hidden layers have their name from the fact that they are not linked to observables and as we will see below when we define the so-called activation $\boldsymbol{z}$, we can think of this as a basis expansion of the original inputs $\boldsymbol{x}$.

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**. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. It is the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators.

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

Mathematical model. The output y is produced via the activation function f

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

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

First, for each node j in the first hidden layer, we calculate a weighted sum z_j^1 of the input coordinates x_i,

$$z_j^1 = \sum_{i=1}^{n} w_{ji}^1 x_i + b_j^1 \tag{2}$$

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

$$y_j^1 = f(z_j^1) = f\left(\sum_{i=1}^{n} w_{ji}^1 x_i + b_j^1\right), \tag{3}$$

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

$$y_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right), \tag{4}$$

where N_{l-1} is the number of nodes in layer $l-1$. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{5}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{6}$$

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

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{7}$$

$$= f^3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{8}$$

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

$$y_i^{L+1} = f^{L+1}\left[\sum_{j=1}^{N_L} w_{ij}^L f^L\left(\sum_{k=1}^{N_{L-1}} w_{jk}^{L-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^{L-1}\right) + b_1^L\right] \tag{9}$$

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

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

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

$$f(x) = c_1 f(c_2 x + c_3) + c_4, \tag{10}$$

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

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

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

We have that $\boldsymbol{W}_l$ is an $N_{l-1} \times N_l$ matrix, while $\boldsymbol{b}_l$ and $\boldsymbol{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 (assuming three nodes for simplicity) as

$$\boldsymbol{y}_2 = f_2(\boldsymbol{W}_2\boldsymbol{y}_1 + \boldsymbol{b}_2) = f_2\left(\left[\begin{array}{ccc} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{array}\right] \cdot \left[\begin{array}{c} y^1_1 \\ y^1_2 \\ y^1_3 \end{array}\right] + \left[\begin{array}{c} b^2_1 \\ b^2_2 \\ b^2_3 \end{array}\right]\right). \tag{11}$$

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

$$y^2_i = f_2\Big(w^2_{i1}y^1_1 + w^2_{i2}y^1_2 + w^2_{i3}y^1_3 + b^2_i\Big) = f_2\left(\sum_{j=1}^3 w^2_{ij}y^1_j + b^2_i\right). \tag{12}$$

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

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

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

Logistic and Hyperbolic activation functions 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*

$$f_{\text{sigmoid}}(z) = \frac{1}{1+e^{-z}},$$

and the *hyperbolic tangent* function

$$f_{\tanh}(z) = \tanh(z)$$

Rectifier activation functions The Rectifier Linear Unit (ReLU) uses the following activation function

$$f_{\mathrm{ReLU}}(z) = \max(0, z).$$

To solve a problem of dying ReLU neurons, practitioners often use a variant of the ReLU function, such as the leaky ReLU or the so-called exponential linear unit (ELU) function

$$f_{\mathrm{ELU}}(z) = \left\{ \begin{array}{cc} \alpha\left(\exp{(z)}-1\right) & z < 0,\\ z & z \ge 0. \end{array} \right.$$

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. It has become the most popular for *deep neural networks*

1.5 Deriving the back propagation code for a multilayer perceptron model

Note: figures will be inserted later!

As we have seen the final output of a feed-forward network can be expressed in terms of basic matrix-vector multiplications. The unknowwn quantities are our weights w_{ij} and we need to find an algorithm for changing them so that our errors are as small as possible. This leads us to the famous back propagation algorithm.

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?

To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\boldsymbol{W}) = \frac{1}{2}\sum_{i=1}^{n}\left(y_i - t_i\right)^2,$$

where the t_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 y_i. Other cost functions can also be considered.

Definitions. With our definition of the targets $\boldsymbol{t}$, the outputs of the network $\boldsymbol{y}$ 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 $\boldsymbol{a}^{l-1}$ 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_j^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 here illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

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

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 the logistic regression lecture. 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 (note that this function depends only on z_j^l)

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

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

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

The derivative of this function with respect to the weights is

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

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

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

Defining

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

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

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

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.

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

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

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.

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

The starting equations

$$\frac{\partial \mathcal{C}(\boldsymbol{W^L})}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{13}$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)}, \tag{14}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{15}$$

An interesting consequence of the above equations is that when the activation a_k^{L-1} is small, the gradient term, that is the derivative of the cost function with respect to the weights, will also tend to be small. We say then that the weight learns slowly, meaning that it changes slowly when we minimize the weights via say gradient descent. In this case we say the system learns slowly.

Another interesting feature is that is when the activation function, represented by the sigmoid function here, is rather flat when we move towards its end values 0 and 1. In these cases, the derivatives of the activation function will also be close to zero, meaning again that the gradients will be small and the network learns slowly again.

We need a fourth equation and we are set. We are going to propagate backwards in order to determine the weights and biases. In order to do so we need to represent the error in the layer before the final one $L-1$ in terms of the errors in the final output layer.

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 have

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

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.

1.6 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.

Summary

- First, we set up the input data $\boldsymbol{x}$ and the activations $\boldsymbol{z}_1$ of the input layer and compute the activation function and the outputs $\boldsymbol{a}^1$.
- Secondly, perform the feed-forward until we reach the output layer. I.e., compute all activation functions and the pertinent outputs $\boldsymbol{a}^l$ for $l = 2, 3, \ldots, L$.
- Compute the ouput error $\boldsymbol{\delta}^L$ by
$$\delta_j^L = f'(z_j^L) \frac{\partial \mathcal{C}}{\partial (a_j^L)}.$$
- Back-propagate the error for each $l = L-1, L-2, \ldots, 2$ as
$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$
- Finally, update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 2$ and update the weights and biases according to the rules
$$w_{jk}^l \leftarrow w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$
$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

The parameter η is the learning rate. Here it is convenient to use stochastic gradient descent with mini-batches and an outer loop that steps through multiple epochs of training.

1.7 Learning challenges

The back-propagation algorithm works by going from the output layer to the input layer, propagating the error gradient. The learning algorithm uses these gradients to update each parameter with a Gradient Descent (GD) step.

Unfortunately, the gradients often get smaller and smaller as the algorithm progresses down to the first hidden layers. As a result, the GD update step 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 that the gradients grow bigger and bigger. The result is that many of the layers get large updates of the weights and the learning 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 identified 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 proposed a way to significantly alleviate this problem. The signal must 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 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 Rectifier Linear Unit (ReLU) uses the following activation function

$$f(z) = \max(0, z).$$

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.

1.8 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 training set is used for learning and adjusting the weights.
- The dev/validation set is a subset of the training data. It is used 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, e.g. changing hyperparameters such as the learning rate.

- The test set will be used to test the performance of or predictions with the final neural net.

It is crucial that we do not use any of the test data to train the algorithm. This is a cardinal sin in ML. T 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.

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

- **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.

- **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.