Introduction to Neural Networks

L. Graesser

July 26, 2016

What is a neural network?

Neural networks are a family of algorithms which excel at learning from data in order to make accurate predictions about unseen examples. The simplest characterization of a neural network is as a function. It maps a set of inputs to outputs. The inputs are called features and contain (hopefully) relevant information about the variable of interest, the output. For example, suppose I wanted a neural network to tell me if a person was in a photograph. Then the inputs would be the integer value representing the color of each pixel (one feature for every pixel), and the output variable would be a binary variable; 1 if a person is in the photo, 0 if it isn't. An alternative objective could be to predict the correct driving direction given images of a road in real time. In this case, the input features are still pixel values, but the output would be an angle corresponding to which direction the steering wheel should turn[1]. Outputs and inputs can be of many different types, including real numbers, categories (e.g. dog, cat, fish), probability distributions, or sequences of characters.

What makes neural networks so interesting is that they are formed from sums and compositions of very simple functions. This results in an algorithmic approach which is very flexible. Infinitely many different neural networks can be built from combinations of the simple building blocks. And it is extremely powerful; neural networks are capable of approximating highly complex functions.

The fundamental building block of a neural network is a node (also called a unit, or a neuron), which computes a function. The inputs for each node are vectors and the output is a scalar. Most nodes takes two sets of inputs, features and weights, and computes the output by performing two steps in sequence.

1. Compute the dot product of the features and weights and add a constant associated with the node, called the bias. This is an affine transformation[2] applied to the input features, controlled by the weights (the "learned parameters")[3].

[1] Off-Road Obstacle Avoidance through End-to-End Learning, LeCun et. al, 2005, NIPS, http://yann.lecun.com/exdb/publis/pdf/lecun-dave-05.pdf

[2] An affine transformation is a transformation which preserves points, straight lines and planes. See https://en.wikipedia.org/wiki/Affine_transformation for more details

[3] Deep Learning, ch 6, pg 171, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

2. Apply a non-linear function to the result of step 1. There are many possible options to choose from. Here, I will use the sigmoid function as this was the first function to be widely used and played an important role in the historical development of neural networks. The sigmoid function maps real numbers to the open interval, $(0, 1)$. The output is an "S" shaped graph (see image below) and the function has the effect of squashing the output of the dot product in step 1 to a value between 0 and 1. Today, the most commonly used non-linear function is a piecewise linear function called the rectified linear unit (ReLU, see image below) and its variants. The role of these non-linear functions, known as activation functions, will be explained in detail later on.

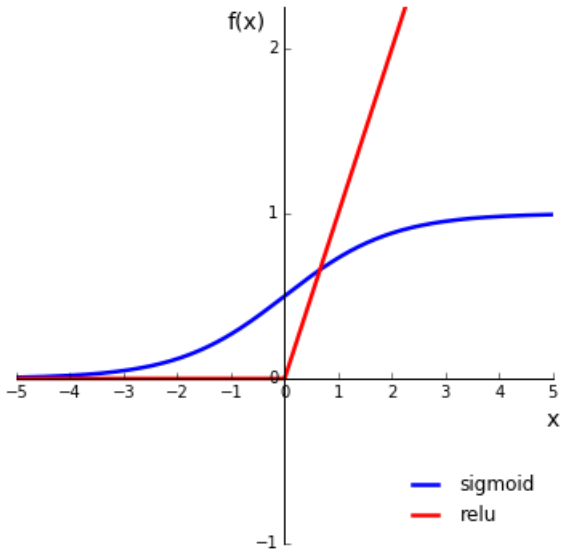

Figure 1: Sigmoid and ReLU activation functions

A node can be thought of as sending a signal, the strength of which is determined by the input features and the weights. A stronger signal corresponds to a large output value, a weak signal to a small output value. For a fixed set of values for the weights, different values of the input features will result in different outputs of the node. So a node computes a new feature and sends a signal about the extent to which this feature is present in the input features.

Let x be the vector of features, w be the vector of weights, b the bias, σ the non-linear sigmoid (activation) function, and a the output of a node. Then a node computes the following function.

$$a = \sigma(w^T x + b) \quad \text{, where} \quad \sigma(z) \quad = \frac{1}{1 + e^{-z}} \tag{1}$$

If you are familiar with logistic regression then this function will ring bells. With a sigmoid activation function each node computes the same function as a logistic regression in which x is a

vector of feature values, and w is the associated set of weights for each feature.

Nodes are connected to other nodes by directed edges. An edge leaves one node (the parent node) and connects to another node (the child node). Each edge is associated with a weight, a real valued scalar. There is at most one edge that connects a node to another node, but there may be no edge at all. For any child node, the outputs of its parent nodes make up the features, one of the two sets of inputs into the function computed by the child node. All of the weights of the incoming edges make up the weight vector for a child node and are the other set of inputs. Every weight corresponds to an input feature. The weight associated with the edge that connects a particular parent node to a child node is also associated with the output of that parent node. It is these two values, the output of the parent node, and its corresponding weight, that are multiplied together in the dot product computed by the child node.

In the example below, the node y, has four parents, a^1, a^2, a^3, a^4.

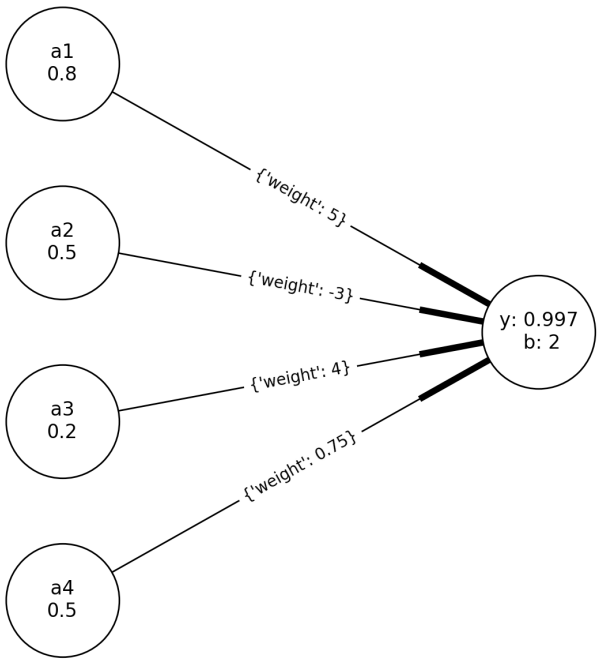

Figure 2: Child node (y) with four parent nodes (four inputs)

Suppose the outputs from the parent nodes and corresponding weights are as follows.

$$a = \begin{pmatrix} 0.8 \\ 0.5 \\ 0.2 \\ 0.5 \end{pmatrix}, \quad w = \begin{pmatrix} 5 \\ -3 \\ 4 \\ 0.75 \end{pmatrix} \quad b = 2$$

Then, y will output ≈ 0.997, which is computed in vector form below.

$$\begin{aligned} y &= \sigma(w^T a + b) \\ a^T w &= 3.675 \\ y &= \sigma(5.675) \\ &= \frac{1}{1 + e^{-5.675}} \\ &\approx 0.997 \end{aligned} \tag{2}$$

In a neural network nodes are organized into ordered layers. It is helpful to think of the layers as organized from low to high, where the lowest layer is the input layer, which receives the original input features, and the highest layer is the output layer, which contains a neural networks' output. All layers in the middle are referred to as hidden layers, since the output values of these nodes are hidden from the user. A neural network with one or more hidden layers is a deep neural network.

Nodes, edges, and layers can be combined in a variety of ways to produce different types of neural networks, designed to perform well on a particular family of problems. Feedforward, convolutional and recurrent neural networks are the most common. This post focuses on feedforward neural networks since they are the simplest to understand, and were developed first. Fortunately, many of the techniques for training[4] feedforward networks also apply to convolutional and recurrent networks. Once you understand feedforward networks, it will be relatively easy to understand the others.

In deep feedforward neural networks, every node in a layer is connected to every node in the layer above it by an edge. A node in layer x is a parent of every node in layer $x+1$. And each node in layer x is the child of every node in layer $x-1$. No nodes within a layer are connected to each other[5]. Layers organized in this way are known as fully connected layers. If you are familiar with graphs then a deep feedforward network is a directed acyclic k-partite graph, where the nodes in each layer form a partition of the graph and where k equals the number of layers in the network.

The input and output layers are special. Nodes in the input layers do not compute any function, instead they just hold the values of the original input features for a particular example in a

[4] Process of finding the best values for the parameters of a neural network so as to produce good results

[5] In more complex feedforward neural networks there may be additional edges which skip layers, connecting nodes to higher layers not in the layer above. However, edges will never connect nodes to lower layers. A node can never be the parent of a node in a lower layers

dataset[6]. The output of these nodes is exactly the same as the input. Nodes in the output layers always compute the first of the two steps that a node typically takes, but may not compute the second step, i.e. they may not apply a non-linear function to the result of step 1.

Finally, computations are executed sequentially. Starting with the input layer, every node in a layer must complete its computation before any node in the layer above can start its computation, since the inputs into a node in layer $x+1$ are the outputs of all of the nodes in layer x. Computations flow through a network, layer by layer, until the output layer is reached. This process is called the ***feedfoward*** step, since information only every flows forward through the network from the input layer to the output layer. This is how a neural network computes an estimate (or prediction) of the correct output value, given a particular set of input features.

Below is an example of a simple deep feedforward network with three layers, the input layer, one hidden layer, and the output layer. There are two nodes in the first layer, x_1^1, and x_2^1, three nodes in the hidden layer, h_1^2, h_2^2, and h_3^2, and one node in the output layer, y_1^3. Typically, weights are indexed by three indices. The superscript refers to the layer of the parent nodes, the first subscript refers to the parent node number, and the second subscript refers to the child node number. The value inside each node is the output value. For simplicity the biases are all assumed to be zero. A vectorized version of the computation steps is below.

[6]A dataset is a set of examples. Each example consists of a number of features, which are usually represented by real numbers or integers. Typically a single example will be represented as a row vector, each element of which represents an individual feature. For example, an integer representing the color intensity of an individual pixel. Or the average monthly spend for a particular customer. Examples are stacked vertically to formed a dataset, a 2D matrix in which the rows represent examples and the columns represent features.

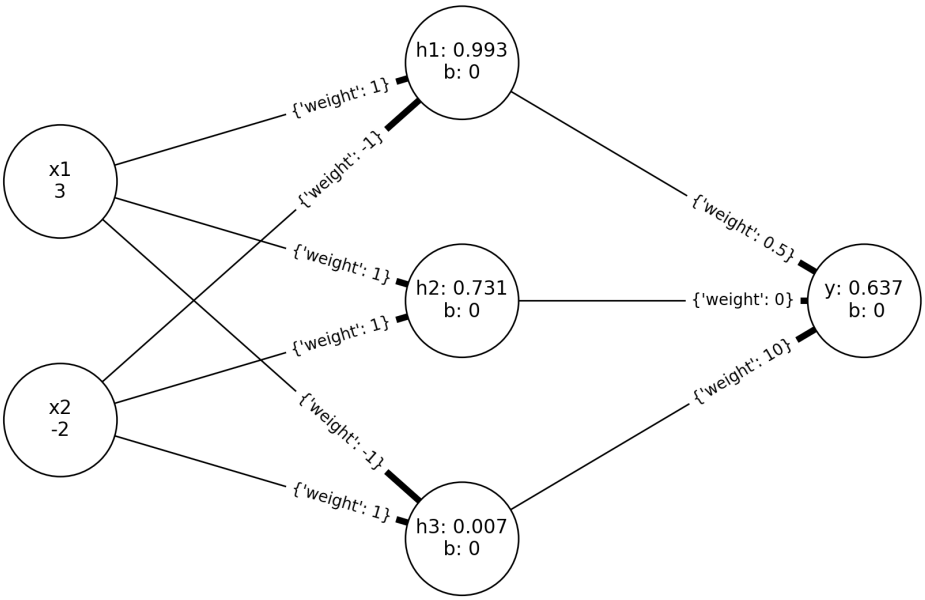

Figure 3: Simple deep feedforward network with 3 layers

$$x = \begin{pmatrix} x^1 \\ x^2 \end{pmatrix} = \begin{pmatrix} 3 \\ -2 \end{pmatrix}, \quad h = \begin{pmatrix} h^1 \\ h^2 \\ h^3 \end{pmatrix}, \quad b_1 = \begin{pmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad b_2 = 0$$

$$W_1 = \begin{pmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & -1 \\ -1 & 1 & 1 \end{pmatrix}, \quad W_2 = \begin{pmatrix} w_{11}^2 \\ w_{21}^2 \\ w_{31}^2 \end{pmatrix}, \quad = \begin{pmatrix} 0.5 \\ 0 \\ 10 \end{pmatrix}$$

$$\begin{aligned} h &= \sigma(W_1^T x + b_1) = \begin{pmatrix} 0.993 \\ 0.731 \\ 0.007 \end{pmatrix} \quad \text{, where } \sigma(z) = \frac{1}{1+e^{-z}} \\ y &= \sigma(W_2^T h + b_2) \\ &= 0.637 \end{aligned} \tag{3}$$

The flexibility of neural networks is clear. Using very simple building blocks it is possible to build deep feedforward networks which can process an arbitrary number of inputs and compute an arbitrary number of outputs. And they can have any number of hidden layers and nodes per layer. However, at this point many things have been left unexplained.

1. Why are activation functions applied at nodes?
2. Why must they be non-linear?
3. Why are networks organized into layers?
4. How are the weights determined so that a neural network makes accurate predictions for different values of the input features?
5. Which activation function to choose?
6. How many layers should a network have?
7. How many nodes should each of the hidden layers have?

And it is not yet clear why structuring an algorithm according to these principles is so powerful. To answer these questions, some historical context explaining the motivations for neural networks will be helpful.

Historical motivations for neural networks

When I think about neural networks, I start by thinking about the problem that linear classifiers are trying to solve when applied to a two class classification problem, a positive class and a negative class. For example, given a set of features about a person applying for a loan (e.g. monthly income, whether they have ever defaulted on a loan before, credit card debt, monthly rent, credit rating, requested loan amount), will they default on the loan or not? Given a set of examples $\in \mathbb{R}^n$ (think of this as there being n features per example) and output $\in \mathbb{Z}$ (each example has an output value of 0 or 1) the objective is to find a separating function that defines a hyperplane $\in \mathbb{R}^{n-1}$ such that the points on one side of the hyperplane are in the positive class, and the points on the other side are in the negative class. If the datapoints are $\in \mathbb{R}^2$, then the separating function will be a straight line. If the datapoints are $\in \mathbb{R}^3$, then it will be a plane.

A simple linear classifier has the following functional form: $b = w^T x$, where x is a vector of features, w is a vector of real valued numbers corresponding to the weight of each feature, and b is a scalar representing the output score. The optimal solution $\hat{b} = \hat{w}^T x$ defines a hyperplane which will separate the data into two classes. Given a new data point, $\tilde{x}$, then $\tilde{b} = \hat{w}^T \tilde{x}$. If $\tilde{b} > \hat{b}$ then $\tilde{x}$ is predicted to belong to the positive class. If not, then it is predicted to belong to the negative class. Linear algebra tells us that the hyperplane defined by $\hat{b}$ can only ever be a linear combination of the initial features. More generally, any algorithm than only makes use of linear or affine transformations (matrix-vector or matrix-matrix multiplication for example), no matter how complex the sequence of operations, will only be able to define a separating boundary that is linear. So unless the classes are linearly separable (i.e. can be separated by drawing a line, plane or hyperplane), this approach cannot produce good results. And unfortunately, most interesting

classification problems involve complex non-linear relationships between the input features, so the classes are not linearly separable.

Linear functions have nice properties[7] which makes it relatively easy to solve for the values of w through either an analytical solution (for example, the normal equations to solve linear least squares regression) or by convex optimizaton. Non-linear functions are much, much more difficult to solve. One solution to this problem is to find an appropriate transformation, or set of transformations to apply to the input features such that they are linearly separable in the new feature space they occupy.[8] This has been the historical approach taken by machine learning researchers and practitioners, and until the recent success of deep neural networks, there were two dominant techniques[9]. The first was to try to specify the correct transformation by manually transforming the input features, known as feature engineering. However this is typically very time consuming and domain specific. Good transformations for solving one problem are not necessarily good for solving another. The alternative was to specify an extremely general transformation. If the transformed features are in a high enough dimension, then it is always possible to linearly separate the dataset. But this approach tended to perform poorly when tested on unseen examples. Neural networks instead try to learn the correct transformation[10]. This approach generalizes much better to unseen data and does not have to be manually engineered. How a neural network achieves this is through the activation function and the hidden layers.

Role of the activation function

A key insight in the development of neural networks was the addition of non-linear[11] activation functions in the hidden nodes to introduce non-linearities into the network. Recall that the first computation that a node computes is an affine transformation, a dot product and addition of a bias. This is a linear transformation of a node's inputs. If this was all that a node did, and since linear transformations of linear transformations are still linear functions of the original input features, then neural networks as a whole would only ever be able to compute linear functions of their inputs, no matter how many layers they had. Nodes that apply a non-linear function to the outcome of a dot product and addition of a bias change this.

Suppose we have a network with three layers, an input layer, one hidden layer, and an output layer, and non linear activation functions in the hidden nodes. Then, the nodes in the hidden layer compute non-linear functions of their inputs. The hidden layer can be interpreted as computing a new set of features, one per hidden node, each of which is a different non-linear transformation of the original features.

For deeper networks, each hidden layer computes a new, more complex, set of features, which are non-linear functions of the features computed by the previous layer. When deep neural networks

[7] Deep Learning; Goodfellow, Bengio, Courville; ch 6, page 168, http://www.deeplearningbook.org/contents/mlp.html

[8] Christopher Olah has some very helpful diagrams and animations which demonstrate how this works in two dimensions. You can find them in this post. http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/. The post itself is excellent and well worth reading.

[9] Deep Learning; Goodfellow, Bengio, Courville; ch 6, page 169, http://www.deeplearningbook.org/contents/mlp.html

[10] Deep Learning; Goodfellow, Bengio, Courville; ch 6, page 169, http://www.deeplearningbook.org/contents/mlp.html

[11] A non-linear function is one in which the output is not directly proportional to the inputs

are used for object recognition in images, we can actually visualize the increasing complexity of the features. Pixel values, the inputs, are the simplest possible type of feature. Then, lower layers learn simple features, to distinguish colors, light and dark, and edges at different orientations. Middle layers learn moderately complex features, simple components of objects such as eyes, beaks, or complex textures, such as feathers. The highest layers learn complex features, how to distinguish between objects, such as people, dogs, and cats[12]. Each hidden layer in the network occupies a different feature space because of the chain of non-linear transformations. As the layers get closer to the output layer, the feature space hopefully becomes closer to one in which the data is linearly separable. It is the last layer, the output layer, with one node per category, which represents the linear separation of datapoints into classes. Now we can see why it isn't necessary to apply the non-linear activation function to the nodes in the output layer. The previous layers are responsible for transforming the original features so that they are linearly separable. The role of the last layer is to find the linear separating boundary. In practice, a non-linear function is often applied to the output nodes after step 1 so that outputs can be more easily interpreted, for example by transforming the maximum valued output node to a 1, and the rest to 0s, or so that the interpretations are more expressive, for example as a probability distribution over the output categories.

It is the activation functions, then, that gives deep networks their power. Without them, the new features computed at each hidden layer would be equivalent to the original features, in that they occupy the same feature space. The depth of the network adds nothing.

By adjusting the values of the weights and biases, a neural networks learns to compute the most useful transformations to produce intermediary features in the hidden nodes which are tailored to the task that it was presented with. The hidden layers give a neural network the flexibility to compute different transformations for different tasks. This corresponds to a different value of the weights and biases, and is why a neural network is said to learn the transformation. Programmers do not specify the values of the parameters in advance.

Introducing non-linearity in this way is more powerful than you might imagine. A deep feedforward neural network with just one hidden layer is a universal approximator, provided there are enough nodes in the hidden layer[13]. It can approximate any function to any degree of accuracy required. Even though this is a theoretical result, it is an amazing property to think about. Originally proven for "squashing functions," such as the sigmoid function, which saturate for small and large values of the input, it has now been extended to all non polynomial activations[14,15]. However,

[12]The Keras blog visualized a selection of features that nodes at different layers in a neural network compute http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html. For more information on the types of features learned in different layers by neural networks applied to computer vision, see Understanding Neural Networks Through Deep Visualization, Yosinski et. al, http://yosinski.com/media/papers/Yosinski__2015__ICML_DL__Understanding_Neural_Networks_Through_Deep_Visualization__.pdf, and Visualizing and Understanding Convolutional Networks , Zeiler and Fergus, https://arxiv.org/abs/1311.2901

[13]Multilayer Feedforward Networks are universal approximator, Hornik et. al, 1989, http://deeplearning.cs.cmu.edu/pdfs/Kornick_et_al.pdf

[14]Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function, Leshno et. al, 1993, http://www2.math.technion.ac.il/~pinkus/papers/neural.pdf

[15]If you are interested in getting more intuition for why this is the case, chapter 4 of Michael Nielsen's, *Neural Networks and Deep learning* is helpful http://neuralnetworksanddeeplearning.com/chap4.html. The whole book is superb, particularly the explanation of backpropagation, and is well worth reading in its entirety

being able to represent a function doesn't necessarily mean that a network can learn it. There are two potential pitfalls. During training, a network may choose to approximate wrong function (for example by overfitting the training set) or the wrong values for the parameters (by getting stuck in local minima or flat regions)[16]. A further challenge is that the number of hidden nodes required to approximate the desired function may result in a network that is intractably large. Depth helps to mitigate this problem by making networks more computationally efficient. A function of the same complexity may often be approximated using a deeper network with many fewer nodes when compared to a network with one hidden layer[17]. In practice then, depth makes it easier for neural networks to learn more complex transformations. This suggests that a network designer should just keep adding layers to the network in order to improve performance. Unfortunately, the deeper a network is the harder it is to train the network. A topic we will return to later.

One final interpretation of building networks with multiple layers is that the network designer believes that solving the task at hand is best approached by breaking the problem into smaller, simpler subproblems, and solving them first. More formally, it expresses a strong prior that the function we want to approximate to map inputs to outputs is composed of many simpler functions. That there is a hierarchical and sequential component to the function being computed[18].

There are many possible candidates for activations functions since there are many non-linear functions. However the choice of activation function has a significant effect on how well and how fast a network learns. Good activations functions need to be:

1. Non-linear
2. Continuously, or almost continuously differentiable.
3. Have a wide range with a non zero derivative so as not to saturate.

To understand why the last two points matter, it is necessary to dive into how neural networks learn.

How a neural network learns

When a neural network is described as "learning", this refers to the process of finding the correct values of the weights and biases (the variables, or parameters) in the network. Correct in this context means that when an example is fedforward through a network, the output is the correct class, probability distribution or number, even if the program has never seen this example before.

A neural network learns by processing labelled examples, known as the training dataset. This is supervised learning. Each example, x, is associated with a correct output y. The network processes many examples, often repeatedly, in the training set. For each example, the network checks how

[16]Deep Learning, ch 6, pg 197, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

[17]Deep Learning, ch 6, pg 198, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

[18]Deep Learning, ch 6, pg 200, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

close its output is to the correct value. Over time, the parameters of the network are adjusted so that the outputs the network produces for the training examples are as close as possible to the correct answer. The goal of the network is to correctly predict y for previously unseen examples. After the network has been trained, it is tested on a different data set, which contains examples with output labels that the network has never seen before. How well the network performs on this data, the % of examples it gets correct, is the most important measure of a network's performance since it simulates the situation in which a network is tested on unseen examples. The only difference being that in this case, we know what the correct answer is[19].

More formally, a loss (or cost) function is defined which the network has to minimize. There are many possible functions to choose from, but they need to satisfy certain properties. The most important is that for a particular example the loss is low, ideally 0, when the network outputs the same result as the label associated with that example. And the more different the result is from the correct answer, the higher the loss should be. Two popular loss functions are Quadratic Loss and Cross Entropy Loss. Let y be the correct output vector for an example, x. And let a be the output vector of the network given x. Then the cost functions are defined as follows:

$$\begin{aligned} \text{Quadratic Loss: } & \frac{1}{2} \parallel y - a \parallel_2^2 \quad \text{, alternatively } \frac{1}{2}(y-a)^T(y-a) \\ \text{Cross Entropy Loss: } & -\left(y^T \ln(a) + (1-y)^T \ln(1-a)\right) \end{aligned} \tag{4}$$

To give some concrete examples, let's take a problem for which there is only one output value that the network needs to estimate, and that variable is binary, it has a value of 0 or 1. If the correct answer for a particular example is 1, then below are the losses for different values of a for each cost function.

1. $a = 0.99$
   (a) Quadratic loss $= (1 - 0.99)^2 = 0.0001$
   (b) Cross Entropy loss $= -(1 * \ln(0.99) + 0 * \ln(0.01)) \approx 0.01$
2. $a = 0.5$
   (a) Quadratic loss $= (1 - 0.5)^2 = 0.25$
   (b) Cross Entropy loss $= -(1 * \ln(0.5) + 0 * \ln(0.5)) \approx 0.69$
3. $a = 0.01$
   (a) Quadratic loss $= (1 - 0.01)^2 = 0.9801$
   (b) Cross Entropy loss $= -(1 * \ln(0.01) + 0 * \ln(0.99)) \approx 4.61$

[19] Alternatively a network can learn through unsupervised learning. In this case, there are only examples, x, with no associated output or label. The goal of the network is to learn something about the structure of the data, without being presented with a target variable or variables to estimate. In practice this tends to mean transforming the dataset from a high dimensional feature representation to a lower dimensional representation which is both meaningful and easier to interpret than the original set of features.

The closer the network output is to the correct answer, the smaller the loss, and the further away, the more wrong the network is, the higher the loss. So, these functions are both good candidates for loss functions. One practical consideration for the Cross-Entropy loss is that ln(0) is undefined. Implementations which make use of this function therefore need to modify the output units and/or the cost function such that ln(0) is never computed.

Suppose we are training a network to correctly classify just a single example. Once the loss for this example has been computed, the network parameters need to be modified so as to reduce the loss the next time the example is fedforward through the network. This has the effect of changing the output so that it is closer to the correct value. The goal is to find a value to change each parameter by so that the loss decreases as much as possible in the next iteration, then update each parameter accordingly. The process of finding the optimal value to adjust each parameter by is known as backpropagation. Invented in 1986 by Rumelhart, Hinton and Williams[20], it is an elegant application of the chain rule of calculus and dynamic programming.

Backpropagation finds the partial derivative (also called the gradient) of each of the weights and biases with respect to the loss function, evaluated for each training example. This process can be thought of as propagating the error of the output nodes backwards through the network, layer by layer. Hence the name. Once the the gradient has been found for each parameter, it is multiplied by a small constant known as the learning rate[21], and subtracted from the current value for that parameter. In practice, the process of feeding an example forward through the dataset, computing the loss and associated gradient, is carried out for each example in the dataset. The gradient per parameter is then averaged over all of the examples, and the parameters are updated using this average value. Known as *gradient descent*, this process is repeated many many times until the loss converges to its minimum value[22].

Intuitively, the partial derivative of the output of a function with respect to one of its inputs is a measure of the extent to which the output will change in response to a small change in that particular input. The loss is a function of the weights and biases (the parameters), so by changing a parameter in a neural network in proportion to the negation of its gradient, and provided that the change is small, the loss will decrease[23]. To picture the changes that the parameters go through, imagine a surface in parameter space plus an extra dimension for the loss. A point on this surface is the size of the loss (in the loss' dimension) evaluated at a particular value for each

[20] Learning representations by back-propagating errors, Nature, 1986, David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams

[21] Hyperparameter who's value is specified by the network designer

[22] Some additional factors are also used to decide when gradient descent is stopped to prevent the algorithm from overfitting the training set. A common heuristic is to stop training when the loss evaluated on the validation data set has stopped decreasing for a few iterations

[23] Assuming all the other parameters are held constant, and assuming the rate of change of the gradient (2nd order partial derivatives) does not change too significantly in the local region of the parameter's current value. In practice all of the parameters are updated at once, so the final effect on the loss function is one in which all of the parameters are interacting. Furthermore, second order effects might be significant. This adds complexity to the process of training neural networks since the actual change in the loss function will be different to the effect implied by the gradient. The trick is to find an appropriately sized learning rate such that the loss function is relatively well behaved in the region of the update, but not so small that the network takes too long to train

of the weights and biases, and a specific example. It can be helpful to think of mountains and valleys in 3-dimensions. In this case the parameter space is in 2-dimensions and the loss is the 3rd dimension. As the two parameter values are changed (moving around on the surface of the earth), the loss changes (how high you are from sea level), with high loss represented by peaks, and low loss by valleys. The goal is to change the weights and the biases to move to the point on the surface which has the lowest cost. Each adjustment to the parameters can therefore be thought of as the network taking a small step downhill in the direction of the steepest slope. By taking many, many of these small steps, the network will hopefully follow a path down to a global minimum, the point on the surface where the loss is lowest[24].

There are a few important implications of this approach.

- The size of the gradient and the learning rate determine the rate at which the network learns, since together they determine the change in the parameter values.

- The learning rate hyperparameter is as a constraint on the size of the steps that a network takes. It has a significant effect on how fast a network learns, or if it learns at all. Too big and the steps are too large for the local region of the update to be well behaved. This causes the loss to chaotically jump about, never converging to a low value. The network learns nothing. Too small and the network will take very small steps and take too long to train. Network designers need to specify the size of this parameter, and if it should change during training. Typically, the initial learning rate is in the range from 10^{-1} to 10^{-6} is selected experimentally through hyper-parameter tuning.

- The initial values of the parameters matter. They affect how long it takes for the network to reach the point where the loss is minimized. It is quite easy to imagine why. In the extreme case, if the parameters were all initially set (initialized) to the correct values then there is no work for the network to do. A more plausible scenario is to suppose that the parameters are initialized close to the correct values. Then, all other things being equal (the learning rate and gradient), the network will have to take fewer steps to reach the correct answer than if the parameters are initialized far from the correct values. Alternatively, suppose there two paths to the minimum loss. One path has a very steep slope, the other a shallow slope. Parameter initializations which place the starting point on the steep path as opposed to the shallow path will mean a neural networks has larger gradients (corresponding to the steeper slope) and can make faster progress towards the minimum loss. Finally, the initial values of the parameters may also affect the quality of the final result, how small a loss the network reaches before it stops learning. This is not immediately obvious but follows from the fact that non-linear activation functions cause the loss function to be non-convex. There may be many flat regions and local minima, which makes it difficult for a neural network to find the global minimum. And there is no guarantee that there will be a path from your starting point to the point of lowest cost.

- Ideally, the updates to the parameters should be zero or very close to zero when the network is very accurate. Since the learning rate is constant, this implies that the gradient should

[24]For a more in depth treatment of backpropagation, read chapter 2 of Michael Nielsen's *Neural Networks and Deep Learning*, http://neuralnetworksanddeeplearning.com/chap2.html. It is superb!

be near zero. If a network is producing excellent results with a particular configuration of its parameters then it should not change significantly from those settings. Conversely, the gradient should be large when the network is very wrong since this means that the parameters are far from their optimal values, and require significant updates. A large gradient allows the network to make relatively large changes to the weights.

The gradient is a function of the loss and activation functions. Much of the research on these functions has been motivated by trying to guarantee the last property above. It is not straightforward. To gain some insight as to why, let's define the *error* of a node as the gradient of the loss with respect to the input to the activation function of a node. The error can be thought of as how sensitive the loss is to a small change in the value of a node, before applying the activation function. The formula for the error in the output layer nodes sheds some light on the issues that can arise with loss and activation functions since it is these errors which are propagated backwards through the network and are used to compute the final gradients.

Suppose there are n output nodes. Let the output of output layer node i, be a_i, let z_i be the value of the node before the activation function is applied, and δ_i be the error of that node. So a is the vector of outputs for all the nodes in the output layer, z is the vector of values pre-activation, and δ is the vector of errors. Let C be the loss function. Let $f(x)$ be the activation function for each of the nodes in the output layer, and let $f\,'(x)$ be the derivative of this function. Finally, I will use the nabla operator, ∇, to represent the following vector, since this is the symbol used in vector calculus.

$$\nabla_a C = \left[\frac{\partial C}{\partial a_1}, \frac{\partial C}{\partial a_2}, ..., \frac{\partial C}{\partial a_n}\right]$$

So, $\nabla_a C$ is a vector containing the partial derivatives of the cost function with respect to the output of each of the output layer nodes. $\odot$ is the element wise multiplication operator. Then, the error vector for the output layer is given by.

$$\begin{aligned}\delta &= \nabla_a C \odot f\,'(z) \\ &= \left[\frac{\partial C}{\partial a_1} f\,'(z_1),\ \frac{\partial C}{\partial a_2} f\,'(z_2),\ ....,\ \frac{\partial C}{\partial a_n} f\,'(z_n)\right] \\ &= \left[\frac{\partial C}{\partial a_1}\frac{\partial a_1}{\partial z_1},\ \frac{\partial C}{\partial a_2}\frac{\partial a_2}{\partial z_2},\ ....,\ \frac{\partial C}{\partial a_n}\frac{\partial a_n}{\partial z_n}\right]\end{aligned} \tag{5}$$

The last line shows that this step is an application of the chain rule. From this, it is clear that if any element of $\nabla_a C$ or $f\,'(z)$ is small then the error for the corresponding element of δ will also be small. Typically, $\nabla_a C$ will be small when the network is very accurate, and large when it is not. This is a desirable property, one which both the quadratic loss[25] and cross-entropy loss[26] have.

$$\begin{aligned}&\nabla_a C^Q = (a - y) \\ &a \approx y \Rightarrow \nabla_a C^Q \approx 0\end{aligned} \tag{6}$$

[25] Michael Nielsen, Neural Networks and Deep Learning, Ch 2, http://neuralnetworksanddeeplearning.com/chap2.html

[26] Michael Nielsen, Neural Networks and Deep Learning, Ch 3, http://neuralnetworksanddeeplearning.com/chap3.html

$$\begin{aligned}\nabla_a C^{CE} &= -\left(\frac{y}{a} - \frac{(1-y)}{(1-a)}\right) \\ a \approx y \Rightarrow \frac{y}{a} &\approx 1, \frac{(1-y)}{(1-a)} \approx 1 \\ \Rightarrow \nabla_a C^{CE} &\approx -(1-1) = 0\end{aligned} \tag{7}$$

It is with $f\,'(z)$, that problems tend to arise. Let's consider three different activation functions, the sigmoid, σ, the rectified linear unit, $ReLU$ and the leaky rectified linear unit, $LReLU$. $f(x)$ and $f\,'(x)$ for the three functions are given below.

$$\begin{aligned}&\sigma(x) = \frac{1}{1+e^{-x}} && \sigma\,'(x) = \sigma(x)(1-\sigma(x)) \\ &ReLU(x) = max(0,x) && ReLU\,'(x) = \begin{cases} 0 & if\ x \leq 0 \\ 1 & if\ x > 0 \end{cases} \\ &LReLU(x) = max(0,x) + min(0.1x, 0) && LReLU\,'(x) = \begin{cases} 0.1 & if\ x \leq 0 \\ 1 & if\ x > 0 \end{cases}\end{aligned} \tag{8}$$

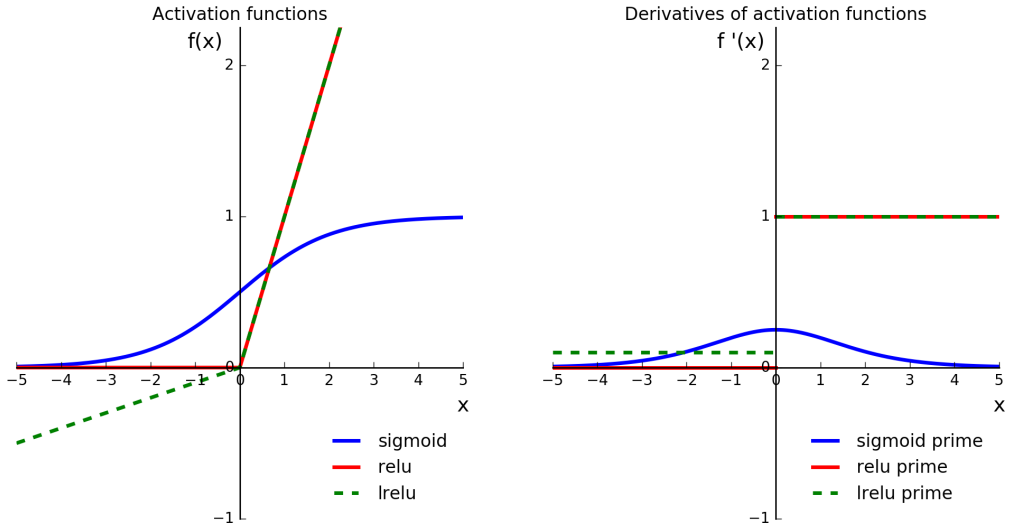

Figure 4: Common activations functions and their derivatives

For most of the domain of z, the range of $\sigma\,'(z)$ is approximately 0, and the nodes are either not firing (low values of z) or are saturated (high values of z). Only when $-4 \leq z \leq 4$

is $\sigma\,'(z)$ somewhat larger. The maximum value that the function can take is 0.25. Recall that $\delta = \nabla_a C\ f\,'(z) = \nabla_a C\ \sigma\,'(z)$. Whatever the value of z, $\sigma\,'(z)$, will shrink δ. And often $\sigma\,'(z)$ will shrink δ to almost 0. This can be very problematic when the network is wrong and the z vector associated with the incorrect output causes $\sigma\,'(z)$ to be almost 0. In this case $\nabla_a C$ will be large, but the error, δ will be very small. This causes the updates to the parameters to be very small when in fact they should be large. For deep networks this issue is amplified. The error for each node is calculated by taking the dot product of the errors in the layer above and the relevant weights. Then the result is multiplied by the derivative of the activation function for that node. The multiplicative presence of the derivative of the sigmoid activation function further shrinks the errors, weakening the signal in the network and making it difficult for it to learn.

The ReLU does not saturate for positive values of z which is a significant advantage. Instead, the derivative is 1 for all positive values of z. When the network is producing accurate results, δ will be approximately 0 because $\nabla_a C$ will be very small. However, when the network is very wrong, and z is positive, then $\delta = \nabla_a C$. The error has not been diluted and so the updates to the parameters will be much larger when compared with the sigmoid function. Consequently, networks with ReLU activations tend to learn much faster, and achieve better results than networks with sigmoid activations. It is somewhat surprising then that the ReLU has only recently become popular, since 2009[27]. A variant of the ReLU was used in early neural networks but tended to be replaced by the sigmoid function in the 1980s. Partly because sigmoid units appeared to perform better on smaller networks but also because it was widely believed that activation functions had to be continuously differentiable[28] in order for the backpropagation algorithm to work. In practice, activation functions that are almost continuously differentiable work just fine. Whilst the probability of encountering the exact value of x where the derivative of $ReLU(x)$ is undefined (when $x = 0$) is very low, implementations get around this issue by picking either the left or right derivative at 0 rather than causing an error. The benefits of a non saturating derivative seem to significantly outweigh any issues that this might cause.

Currently, a number of variants of the ReLU are becoming popular and which aim to address the issue that a ReLU node cannot learn anything if $z \leq 0$ since $ReLU\,'(z) = 0$. The leaky ReLU ($LReLU$) is one example. $LReLU(z) = \alpha z\ \forall z \leq 0$, where α is small, 0.1 for example. Then $LReLU\,'(z) = 0.1\ \forall z \leq 0$. The $LReLU$ is the same as the ReLU function for $z > 0$. Others include the parametric ReLU and the exponential ReLU[29]. They all introduce some small non-zero derivative $\forall z \leq 0$ which corresponds to allowing a weak signal to propagate through the network for negative values of z.

It is worth spending some time on the cross entropy loss function. Suppose the output nodes

[27] Deep Learning, ch 6, pg 225, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

[28] Deep Learning, ch 6, pg 226, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/mlp.html

[29] For more information see Empirical Evaluation of Rectified Activations in Convolution Network, 2015, Lu, Wang, Chen, Li, https://arxiv.org/pdf/1505.00853.pdf, and Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016, Clevert, Unterthiner, Hochreiter, http://arxiv.org/pdf/1511.07289v5.pdf

have sigmoid activations, then δ is given by

$$\delta = \nabla_a C^{CE} \sigma\,'(x) = -\left(\frac{y}{a} - \frac{(1-y)}{(1-a)}\right)\sigma'(z)$$

Recall that a are the final output values of the output nodes, so a can be re-written as $\sigma(z)$. So, we have,

$$\begin{aligned}
\delta &= -\left(\frac{y}{\sigma(z)} - \frac{(1-y)}{(1-\sigma(z))}\right)\sigma\,'(z) \\
&= -\left(\frac{y(1-\sigma(z) - (1-y)\sigma(z)}{\sigma(z)(1-\sigma(z))}\right)\sigma\,'(z) \\
&= -\left(\frac{y - \sigma(z)y - \sigma(z) + y\sigma(z)}{\sigma(z)(1-\sigma(z))}\right)\sigma\,'(z) \\
&= -\left(\frac{y - \sigma(z)}{\sigma(z)(1-\sigma(z))}\right)\sigma\,'(z) \\
&= \left(\frac{\sigma(z) - y}{\sigma(z)(1-\sigma(z))}\right)\sigma\,'(z) \\
&= \sigma(z) - y \quad , \textit{ since } \sigma\,'(z) = \sigma(z)(1-\sigma(z))
\end{aligned} \tag{9}$$

When sigmoid activation functions in the output layer are combined with the cross-entropy loss function, the error term, δ, does not have any $\sigma\,'(z)$ element so δ will not be shrunk at all. The cross-entropy loss "undoes" the activation function derivative. The quadratic loss function has no such effect. Learning is likely to be much slower if quadratic loss is used along with sigmoid functions in the output layer nodes when compared with cross-entropy loss. Sigmoid functions will still cause issues in the hidden layers, even when the cross-entropy loss is used, since $\sigma\,'(z)$ features as a multiplicative element in the partial derivative calculation for the weights and biases in these layers. In contrast, use of the ReLU activation function in the output layer causes problems when combined with the cross-entropy loss function[30] since it results in divisions by 0 when $z \leq 0$.

To take advantage of the strengths of difference activation and loss functions, and avoid their weaknesses, a common neural network structure is to use *ReLU* activations (or its variants) for the hidden layers. This avoids diluting in the gradient as it is propagated backwards. The sigmoid or softmax activation function (a generalization of the sigmoid function for more than two output nodes) is then used for the output nodes, combined with the cross-entropy loss.

The final topic to discuss is where to start? How to initialize the weights and the biases? Recall that a neural network can only take small steps in parameter space and that the loss function is unlikely to be convex. The initial choice of variables is therefore important since it partly determines the final outcome. Similarly to the way in which the layered structure of a neural network expresses a prior belief about the function being approximated, the initial values of the parameters in the network express a prior belief about their correct values. Since little is known about what the values should be, the initial values of weights are typically drawn from a Gaussian or uniform distribution

[30] http://stats.stackexchange.com/questions/166595/how-to-apply-cross-entropy-on-rectified-linear-units

with mean 0 and a small variance. It is important that the values are non zero, otherwise nodes will not send any signals and the network will not be able to learn anything. Let w be any weight in the network, and let m be the total number of inputs to the child node associated with w. Then a common initialization strategies is:

$$w \sim U\left(\frac{-1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

The $\sqrt{m}$ term is to help prevent the signal from nodes increasing with the number of inputs (size of layers)[31]. This approach has two additional advantages:

- It is unlikely that two weights in the same layer will be set to the same value, known as breaking symmetry. In a deep feedforward neural network, and more generally for deterministic models, if two hidden units in the same layer have the same inputs then they must have different weights, otherwise they will compute the same function, thus duplicating effort and introducing redundancy into the network[32]. Either one of the two nodes could be removed with no effect, or it is a wasted opportunity. One of the nodes could learn to compute a different feature which is useful to the network
- Small initial values can help to prevent overfitting. They can be thought of as expressing a prior belief that most parameters are not important. This means that for a parameter to end up with a large value, it must significantly contribute to reducing the loss.

Neural networks are less sensitive to the initial values of the biases. The number of inputs per node is typically in the hundreds or thousands, so the dot product of the inputs and weights tends to dominate the scalar bias. However it is can be helpful to set them to a non zero value such as 0.1.

Conclusion

This post has introduced the fundamental characteristics of neural networks. It has focused on deep feedforward neural networks to illustrate how a neural network makes a prediction, how it learns, and the types of choices that network designers need to make. It has discussed the role of the activation function in detail and showed how different activations functions interact with the loss functions to affect the learning process. Neural networks are a generalization of previous machine learning algorithms. Their structure, combined with backpropagation, enables a neural network to learn feature transformations that were once manually engineered, resulting in more power and flexibility.

Five of the seven questions posed in section one have been discussed: *Why are activation functions applied at nodes? Why must they be non-linear? Why are networks organized into layers? How are the weights determined so that a neural network makes accurate predictions for different values of the input features? And which activation function to choose?*

[31] Deep Learning, ch 8, pg 303, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/optimization.html

[32] Deep Learning, ch 8, pg 301, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org/contents/optimization.html

One important topic is missing from this post: regularization. Neural networks are powerful algorithms with the capacity to approximate highly complex functions. Consequently they are vulnerable to overfitting the training data and not generalizing well to unseen examples. There are many strategies that a network designer can make use of to address this issue and they fall under the umbrella term of regularization. My next post will explain regularization and give an overview of the main techniques.

The remaining two questions, *How many layers should a network have?* and *How many nodes should each of the hidden layers have?*, will be discussed in an upcoming post, ***A Neural Network program in Python***. Neural network design is an art. There are no rules, only heuristics for how many layers, and nodes per layer there should be. I will walk through the code for a vectorized general neural network program and demonstrate how it works with examples. This is the ideal time to discuss choices about network design.

I hope that you've enjoyed this post and found it useful. If you have any feedback please send me an email at contact@learningmachinelearning.org

Bibliography

- Neural Networks and Deep Learning, Michael Nielsen, 2016, http://neuralnetworksanddeeplearning.com/
- Deep Learning, Book in preparation for MIT Press, Ian Goodfellow, Yoshua Bengio and Aaron Courville, 2016, http://www.deeplearningbook.org
- Neural Networks, Manifolds, and Topology, Christopher Olah, http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
- A Neural Network Playground, TensorFlow, playground.tensorflow.org
- AI, Deep Learning, and Machine Learning: A Primer, Frank Chen, http://a16z.com/2016/06/10/ai-deep-learning-machines/
- The Revolutionary Technique that Quietly Changed Machine Learning Forever, MIT Technology Review, Sept 2014, https://www.technologyreview.com/s/530561/the-revolutionary-technique-that-quietly-changed-machine-vi
- How convolutional neural networks see the world, The Keras Blog, Francois Chollet, Jan 2016, http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html
- Understanding Neural Networks Through Deep Visualization, Yosinski et. al, http://yosinski.com/media/papers/Yosinski2015ICMLDLUnderstandingNeuralNetworksThroughDeepVisualization.pdf
- Visualizing and Understanding Convolutional Networks , Zeiler and Fergus, https://arxiv.org/abs/1311.2901
- Learning representations by back-propagating errors, Nature, 1986, David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams

- Empirical Evaluation of Rectified Activations in Convolution Network, 2015, Lu, Wang, Chen, Li, https://arxiv.org/pdf/1505.00853.pdf
- Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016, Clevert, Unterthiner, Hochreiter, http://arxiv.org/pdf/1511.07289v5.pdf
- https://theclevermachine.wordpress.com/2014/09/08/derivation-derivatives-for-common-neural-network-ac
- http://stats.stackexchange.com/questions/166595/how-to-apply-cross-entropy-on-rectified-linear-units
- http://stackoverflow.com/questions/9782071/why-must-a-nonlinear-activation-function-be-used-in-a-backp
- https://en.wikipedia.org/wiki/Activation_function
- https://en.wikipedia.org/wiki/Affine_transformation
- https://en.wikipedia.org/wiki/Artificial_neuron
- https://en.wikipedia.org/wiki/Nonlinear_system