Introduction to Neural Networks

Computation unit

Consider a supervised learning problem where we have access to labeled training examples $(x^{(i)}, y^{(i)})$. Neural networks give a way of defining a complex, non-linear form of hypotheses $h_\Theta(x)$, with parameters Θ (also called weights) that we can fit to our data. To describe neural networks, we will begin by describing the simplest possible neural network, one which comprises a single *neuron*. We will use the following diagram (Fig. 1) to denote a single neuron:

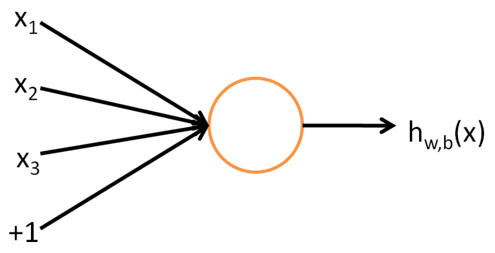


Figure 1: Computation unit

This *neuron* is a computational unit that takes as input $x = (x_0, x_1, x_2, x_3)$ ($x_0 = +1$, called bias), and outputs $h_\theta(x) = f(\theta^\intercal x) = f(\sum_i \theta_i x_i)$, where $f : \mathbb{R} \mapsto \mathbb{R}$ is called the **activation function**. In these notes, we will choose $f(\cdot)$ to be the sigmoid function:

$$f(z) = \frac{1}{1 + e^{-z}}$$

Although these notes will use the sigmoid function, it is worth noting that another common choice for f is the hyperbolic tangent, or `tanh`, function:

$$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The `tanh(z)` function is a rescaled version of the sigmoid, and its output range is $[-1, 1]$ instead of $[0, 1]$.

Finally, one identity that will be useful later: If $f(z) = 1/(1 + e^z)$ is the sigmoid function, then its derivative is given by $f'(z) = f(z)(1 - f(z))$. If f is the `tanh` function, then its derivative is given by $f'(z) = 1 - (f(z))^2$. You can derive this yourself using the definition of the sigmoid (or `tanh`) function.

In modern neural networks, the default recommendation is to use the rectified linear unit or ReLU defined by the activation function $f(z) = \max\{0, z\}$ (Fig. 2). However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient based methods. They also preserve many of the properties that make linear models generalize well.

Historically, the sigmoid was the mostly used activation function since it is differentiable and allows to keep values in the interval $[0, 1]$. Nevertheless, it is problematic since its gradient is very close to 0 when $|x|$ is not close to 0. With neural networks with a high number of layers (which is the case for deep learning), this causes troubles for the backpropagation algorithm to estimate the parameter (backpropagation is explained in the following). This is why the sigmoid function was supplanted by the rectified linear function. This function is not differentiable in 0 but in practice this is not really a problem since the probability to have an entry equal to 0 is generally null. The ReLU function also has a sparsification effect. The ReLU function and its derivative are equal to 0 for negative values, and no information can be obtain in this case for such a unit, this is why it is advised to add a small positive bias to ensure that each unit is active.

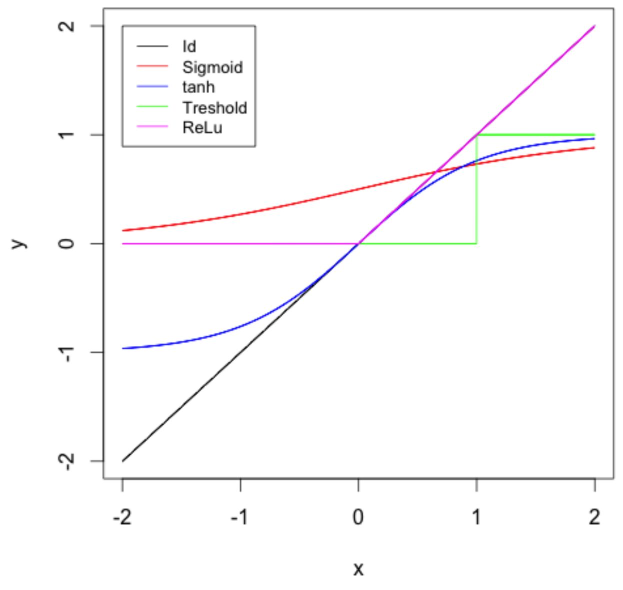


Figure 2: Activation functions

Neural network formulation

A neural network is put together by hooking together many of our simple *neurons*, so that the output of a *neuron* can be the input of another. For example, here (Fig.3) is a small neural network.

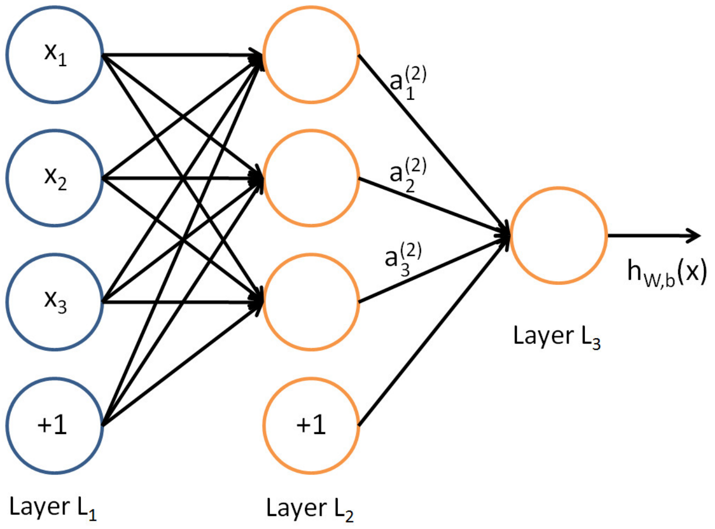


Figure 3: Small neural network

In this figure, we have used circles to also denote the inputs to the network. The circles labeled +1 are called bias units, and correspond to the intercept term. The leftmost layer of the network is called the input layer, and the rightmost layer the output layer (which, in this example, has only one node). The middle layer of nodes is called the hidden layer, because its values are not observed in the training set. We also say that our example neural network has 3 input units (not counting the bias unit), 3 hidden units, and 1 output unit.

Remark

Observe that (Figure 3), if we use sigmoid function as activation in the output node:

- From input layer to layer 2 we implement a non-linear transformation, getting a new set of **complex features**.
- From layer 2 to output layer we implement a logistic regression on the set of **complex features**.

Then, the ouput of the neural network is of the form:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}}$$

Recall that, in logistic regression, we use the logit model

$$\log \frac{p(Y=1|x)}{1 - p(Y=1|x)} = \theta^\intercal x$$

We can isolate $p(Y=1|x)$. Taking exponential in both sides, we have:

$$\frac{p(Y=1|x)}{1-p(Y=1|x)} = e^{\theta^\intercal x}$$

Thus

$$p(Y=1|x) = \frac{e^{\theta^\intercal x}}{1+e^{\theta^\intercal x}} = \frac{1}{1+e^{-\theta^\intercal x}}$$

Observe that, when the activation function of the output node is the sigmoid activation function, the output coincides with a logistic regression on complex features which result from passing the input vector through all layers until it reaches the output node.

Then, with $h_\theta(x)$, the output of the NN, we are estimating $p(Y=1|x)$.

We will let n_l denote the number of layers in our network, thus $n_l = 3$ in our example. We label layer l as L_l, so layer L_1 is the input layer, and layer $L_{n_l} = L_3$ the output layer. Our neural network has parameters $\Theta = (\Theta^{(1)}, \Theta^{(2)})$, where we will write $\theta_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$. Thus, in our example, we have $\Theta^{(1)} \in \mathbb{R}^{3\times 4}$, and $\Theta^{(2)} \in \mathbb{R}^{1\times 4}$, Note that bias units don't have inputs or connections going into them, since they always output the value $+1$. We also let s_l denote the number of nodes in layer l (not counting the bias unit).

We will write $a_i^{(l)}$ to denote the activation (meaning output value) of unit i in layer l. For $l=1$, we also use $a_i^{(1)} = x_i$ to denote the i-th input.

Given a fixed setting of the parameters Θ, our neural network defines a hypothesis $h_\Theta(x)$ that outputs a real number.

Specifically, the computation that this neural network represents is given by:

$$\begin{aligned}
a_1^{(2)} &= f(\theta_{10}^{(1)} + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3) & (1)\\
a_2^{(2)} &= f(\theta_{20}^{(1)} + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) & (2)\\
a_3^{(2)} &= f(\theta_{30}^{(1)} + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) & (3)\\
h_\Theta(x) &= a_1^{(3)} = f(\theta_{10}^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) & (4)
\end{aligned}$$

In the sequel, we also let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \theta_{i0}^{(1)} + \theta_{i1}^{(1)} x_1 + \theta_{i2}^{(1)} x_2 + \theta_{i3}^{(1)} x_3$), so that $a_i^{(l)} = f(z_i^{(l)})$.

Note that this easily lends itself to a more compact notation. Specifically, if we extend the activation function $f(\cdot)$ to apply to vectors in an elementwise fashion (i.e., $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$), then we can write Equations (1-4) more compactly as:

$$
\begin{aligned}
z^{(2)} &= \Theta^{(1)} x \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= \Theta^{(2)} a^{(2)} \\
h_{\Theta}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

More generally, recalling that we also use $a^{(1)} = x$ to also denote the values from the input layer, then given layer l's activations $a^{(l)}$, we can compute layer $l + 1$'s activations $a^{(l+1)}$ as:

$$
\begin{aligned}
z^{(l+1)} &= \Theta^{(l)} a^{(l)} && (5) \\
a^{(l+1)} &= f(z^{(l+1)}) && (6)
\end{aligned}
$$

In matrix notation

$$
z^{(l+1)} = \begin{bmatrix} z_1^{(l+1)} \\ z_2^{(l+1)} \\ \vdots \\ z_{s_{l+1}}^{(l)} \end{bmatrix} = \begin{bmatrix} \theta_{10}^{(l)} & \theta_{11}^{(l)} & \theta_{12}^{(l)} & \dots & \theta_{1s_l}^{(l)} \\ \theta_{20}^{(l)} & \theta_{21}^{(l)} & \theta_{22}^{(l)} & \dots & \theta_{2s_l}^{(l)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \theta_{s_{l+1}0}^{(l)} & \theta_{s_{l+1}1}^{(l)} & \theta_{s_{l+1}2}^{(l)} & \dots & \theta_{s_{l+1}s_l}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{s_l}^{(l)} \end{bmatrix}
$$

The activation

$$
a^{(l+1)} = \begin{bmatrix} a_1^{(l+1)} \\ a_2^{(l+1)} \\ \vdots \\ a_{s_{l+1}}^{(l)} \end{bmatrix} = f(z^{(l+1)}) = \begin{bmatrix} f(z_1^{(l+1)}) \\ f(z_2^{(l+1)}) \\ \vdots \\ f(z_{s_{l+1}}^{(l)}) \end{bmatrix}
$$

By organizing our parameters in matrices and using matrix-vector operations, we can take advantage of fast linear algebra routines to quickly perform calculations in our network. This process is call forward propagation.

We have so far focused on one example neural network, but one can also build neural networks with other architectures (meaning patterns of connectivity between neurons), including ones with multiple hidden layers. The most common choice is a n_l-layered network where layer 1 is the input layer, layer n_l is the output layer, and each layer l is densely connected to layer $l + 1$. In this setting, to compute the output of the network, we can successively compute all the activations in layer L_2, then layer L_3, and so on, up to layer L_{nl} , using Equations (5-6). This is one example of a feedforward neural network (FNN), since the connectivity graph does not have any directed loops or cycles.

Neural networks can also have multiple output units. For example, in (Fig. 4) we can see a network with two hidden layers layers L_2 and L_3 and four output units in layer L_4, where bias of each layer were omited.

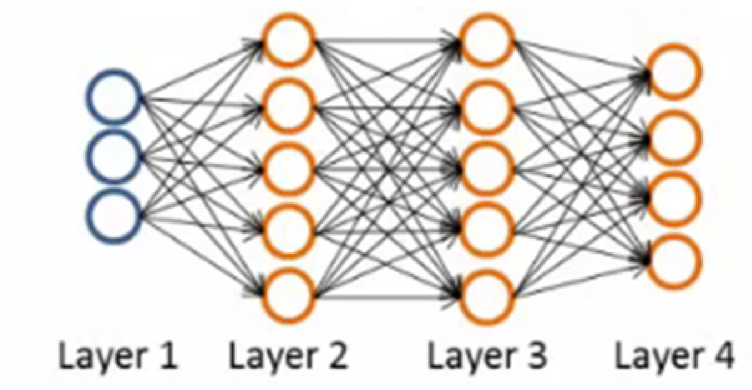


Figure 4: Neural network

To train this network, we would need training examples $(x^{(i)}, y^{(i)})$ where $y^{(i)} \in \mathbb{R}^4$. This sort of network is useful if there're multiple outputs that you're interested in predicting. For example, in a medical diagnosis application, the vector x might give the input features of a patient, and the different outputs y_i's might indicate presence or absence of different diseases.

The binary cross-entropy loss function

As we can say previously, when the activation function of the output node is the sigmoid activation function, the output of the NN is of the form:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}}$$

We need to use a proper (convex) loss function to fit this kind of output values. We can not use the squared error loss, because the minimization of

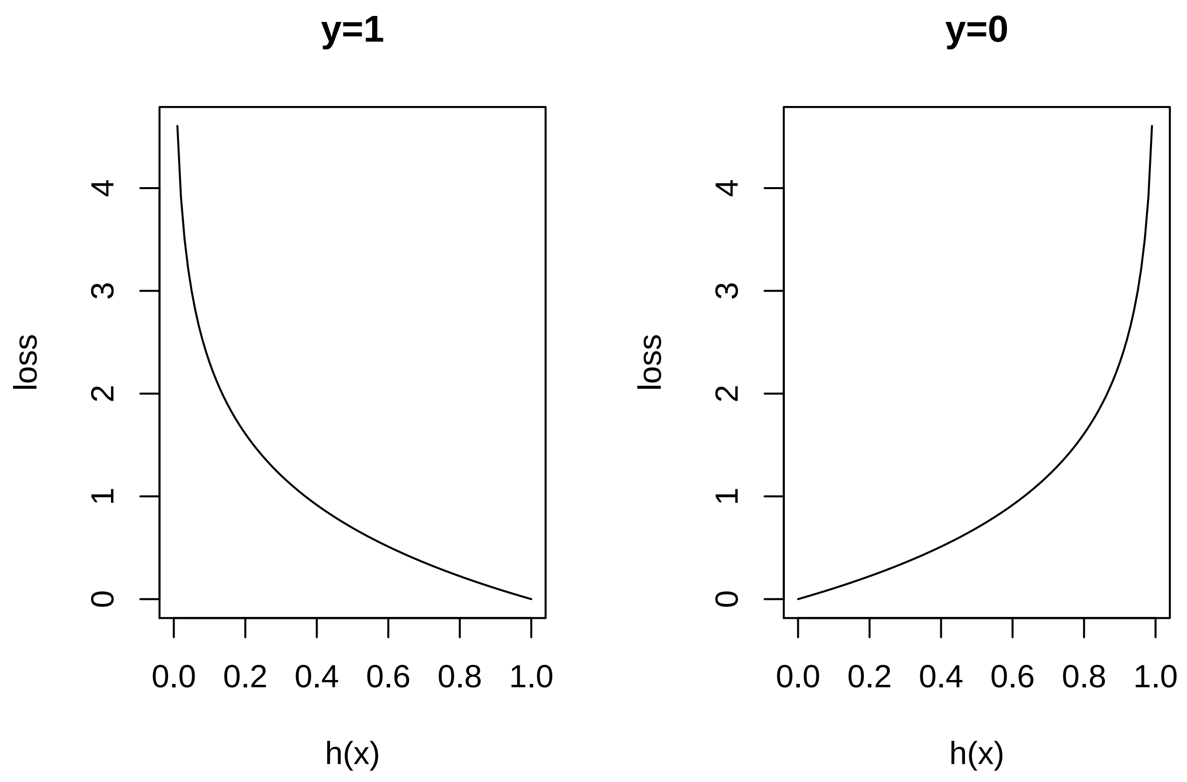
$$l(h_\theta(x), y) = (y - \frac{1}{1 + e^{-\theta^\intercal x}})^2$$

is not a convex problem.

Alternativelly, we use the loss function

$$l(h_\theta(x), y) = \begin{cases} -\log h_\theta(x) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

We can take a look on the graphical representation of the loss function.



We can write the loss function in a compact formulation

$$l(h_\theta(x), y) = -y \log h_\theta(x) - (1-y)\log(1-h_\theta(x))$$

This loss is called binary cross-entropy loss.

And, using the cross-entropy loss, the cost function is of the form:

$$J(\Theta) = -\frac{1}{n}\Big[\sum_{i=1}^{n} y_i \log h_\theta(x_i) + (1-y_i)\log(1-h_\theta(x_i))\Big]$$

And is a convex optimization problem.

Softmax Transfer and Cross Entropy Error

When a classification task has more than two classes, it is standard to use a softmax output layer. The softmax function provides a way of predicting a discrete probability distribution over the classes. We again use the cross-entropy error function, but it takes a slightly different form. The softmax activation of the k-th output unit is

$$\Big(h_\theta(x)\Big)_k = \frac{e^{z_k}}{\sum_j^K e^{z_j}}$$

and the categorical cross entropy cost function for multi-class output is

$$J(\Theta) = -\sum_j^K y_j \log\Big(\Big(h_\theta(x)\Big)_j\Big)$$

Regression task

When we are addressing a regression problem, a convenient activation function on the output node is linear, here we can use the squared error loss function.

Regularization

To conclude, let us say a few words about regularization.

L2 penalization

Let us suppose a multilabel problem (see Fig. 4). In a neural network ($h_\theta(x) \in \mathbb{R}^K$, and $(h_\theta(x))_k$ denotes the k-th output), the cost function (called binary cross-entropy) is of the form

$$J(\Theta) = -\frac{1}{n}\Big[\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(h_\theta(x_i))_k + (1-y_{ik})\log\Big(1-(h_\theta(x_i))_k\Big)\Big] + \lambda\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\theta_{ji}^{(l)})^2$$

Dropuout

For deep learning, the mostly used method is the dropout (see Fig. 5). It was introduced by Hinton et al. (2012). With a certain probability p, and independently of the others, each unit of the network is set to 0. The probability p is another hyperparameter. It is classical to set it to 0.5 for units in the hidden layers, and to 0.2 for the entry layer. The computational cost is weak since we just have to set to 0 some weights with probability p. This method improves significantly the generalization properties of deep neural networks and is now the most popular regularization method in this context. The disadvantage is that training is much slower (it needs to increase the number of epochs). Ensembling models (aggregate several models) can also be used. It is also classical to use data augmentation or adversarial examples.

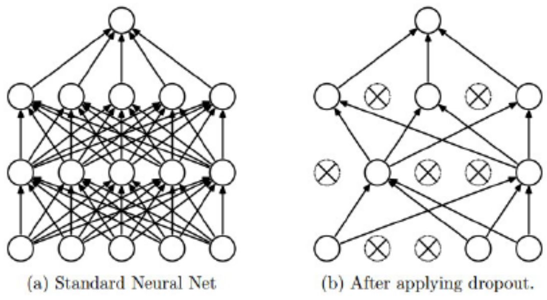


Figure 5: Dropout

Universal Approximation Properties and Depth

Hornik (1991) showed that any bounded and regular function $\mathbb{R}^d \to \mathbb{R}$ can be approximated at any given precision by a neural network with one hidden layer containing a finite number of neurons, having the same activation function and one linear output neuron. This result was earlier proved by Cybenko (1989) in the particular case of the sigmoid activation function. More precisely, Hornik's theorem can be stated as follows.

Theorem. Let ϕ be a bounded, continuous and non decreasing (activation) function. Let K_d be some compact set in $\mathbb{R}^d$ and $C(K_d)$ the set of continuous functions on K_d. Let $f \in C(K_d)$. Then for all $\epsilon > 0$, there exists $N \in \mathbb{N}$, real numbers v_i, b_i and $\mathbb{R}^d$-vectors w_i such that, if we define

$$F(x) = \sum_{i=1}^{N} v_i \phi\Big(w_i^T x + b_i\Big)$$

then we have

$$\forall x \in K_d, |F(x) - f(x)| \leq \epsilon.$$

This theorem is interesting from a theoretical point of view. From a practical point of view, this is not really useful since the number of neurons in the hidden layer may be very large.

The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. In summary, a feedforward network with a single hidden layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.

In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error. There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d, but which require a much larger model if depth is restricted to be less than or equal to d. In many cases, the number of hidden units required by the shallow model is exponential in p (input space dimension).

The strength of deep learning lies in the deep (number of hidden layers) of the networks.

References

Aggarwal, Charu C. Neural networks and deep learning. Berlin, Germany. Springer, 2018.