CS337: AIML
The Neural Network

Scribes: Pavuluri Sohith Kumar*, Siddharth Bhuva*, Karanam Lokesh,
Krishna Narasimhan Agaram*, Aditya Raj, Aadish Sethiya
Edited by: Govind Saju

September 12, 2023

We introduce possibly the most influential modern machine learning innovation - **neural networks**. Imagine you're teaching a computer to recognize patterns, like recognizing whether a photo contains a cat or a dog. In traditional machine learning, we might manually define specific rules or features to help the computer make decisions. For example, we could tell it to look for specific shapes or colors.

Now, think of a neural network as an approach inspired by the human brain. Just like our brain has interconnected neurons, a neural network has layers of 'artificial neurons' or 'nodes.' These nodes work together to automatically learn patterns from the data without us having to specify explicit rules.

What's fascinating is that during training, the neural network adjusts the connections between these neurons (just like our brain strengthens or weakens connections) to get better at recognizing patterns. We show it many examples of cats and dogs, and it gets better over time at distinguishing them on its own. Note however, that analogy between NNs and the human brain stops here - the way the brain works is very different from the way neural networks learn.

Neural nets are especially powerful for tasks like image recognition, natural language processing, and many other complex problems where traditional machine learning techniques struggle. It all started in the 1940s with...

1 The McCulloch-Pitts Neuron model

This is one of the simplest models of an object that learns from data. McCulloch and Pitts proposed this model in 1943, calling it an **artificial neuron** (we will henceforth simply call it a neuron). Data $\mathbf{x} = (x_1, \ldots, x_n)$ is passed to the neuron, which has weights $\mathbf{w} = (w_1, \ldots, w_n) \in \mathbb{R}^n$, $b \in \mathbb{R}$ on its incoming edges. The neuron computes the quantity $f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$, where $f_{\mathbf{w},b}$ is called the aggregate function. The neuron then passes the result through a function g called the **activation function**. The output of the neuron is defined to be $g(f_{\mathbf{w},b}(\mathbf{x}))$. Note that the activation of a neuron on input $\mathbf{x}$ is simply its output on input $\mathbf{x}$.

Remark. Consider the case where g is the sign function. This neuron model is precisely the perceptron model! Indeed, in 1957, Rosenblatt proposed the perceptron model inspired by the McCulloch-Pitts neuron model. And thus, one neuron can be used to perform binary classification.

[*] Larger credit to these scribes for the final notes.

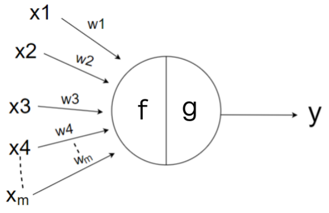


Figure 1: The McCulloch-Pitts model of an artificial neuron.

2 A brief history of neural networks

- 1943: McCulloch and Pitts propose the McCulloch-Pitts neuron model.
- 1957: Rosenblatt proposes the perceptron model.
- 1960: The idea of backpropagation is proposed by a controls engineer, Kelley. It was actually developed for flight path control.
- 1969: Minsky and Papert show that the perceptron model cannot learn the XOR function.
- 1986: Rumelhart, Hinton, and Williams show that backpropagation can be used to train neural networks with hidden layers.
- 1989: The convolutional neural network is proposed by LeCun.
- 1997: The long short-term memory (LSTM) model is proposed by Hochreiter and Schmidhuber.
- 2009: Deep learning for speech recognition is proposed by Dahl.
- 2012: AlexNet wins the ImageNet competition.
- 2014: Generative adversarial networks (GANs) are proposed by Goodfellow.
- 2016: AlphaGo defeats Go world champion Lee Sedol.

Today, deep learning has taken over the machine learning community by storm by being able to solve really hard problems, often better than humans. The advent of deep (many hidden layers) neural networks has been made possible thanks to the following factors:

1. Vast amounts of data.
2. Faster computation: specialized hardware like GPUs (and recently TPUs) for matrix computation.
3. Better algorithms: better initialization, better activation functions, better optimization techniques, toolkits, libraries like TensorFlow, PyTorch, etc.

3 The Multi-Layer Perceptron

Okay, back to the neuron. The output of one neuron would serve quite well for regression where we need a scalar output (indeed, a single neuron with g being the identity is linear regression). Supposing we have a classification problem, say classify an input x into one of the classes $0, 1, \dots, 9$? A simple way to use a network for this is to have 10 neurons, pass the same input x to all of them, and then predict the class to be the one corresponding to the neuron with the highest output value. Indeed, this is not a bad idea, and is called (multinomial) logistic regression (see this for details).

Key Idea 1

We can use outputs from neurons to make a prediction for classification/regression problems.

In general, each neuron is expected to 'do its bit', that is, capture something interesting about the data, some pattern, etc. One can think of them as mini-perceptrons, each with its own set of weights and activations, finding a separator of their inputs. Even when they receive the same input, these neurons can perceive different aspects of the data.

To illustrate, imagine two inputs that seem similar to one neuron, as their outputs are closely related, but appear quite distinct to another neuron. This diversity arises from the different weightings applied to the input in each of these neurons. Essentially, different weights capture different separations of the data.

Supposing we could use these varied outputs in some way - give them to someone to make sense of the different separations these neurons have come up with. Here is the key idea:

Key Idea 2

The outputs of neurons can be passed as input to another neuron to possibly recognize more intricate features in the separations.

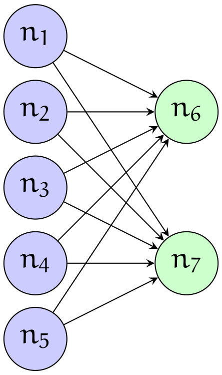


Figure 2: Connecting the output of neurons to the input of two other neurons.

Passing the output of the d neurons as the input to another neuron forms a more complicated network of interconnected neurons - the neural network. On input x, this 'second-level' or second-layer neuron now has different details about the input x fed to it by all the 'first-layer' neurons - and these new details are, crucially, non-linear. The second-layer neuron can now separate the data on non-linear boundaries. Two second-layer neurons, with their different weights, lead to two non-linear decision boundaries.

But before we revel in glory, let's see how the final output $\hat{y}$ looks like:

$$\hat{y} = g_2(\mathbf{w}^T \mathbf{x}_1 + \mathbf{b}).$$

Here $\mathbf{x}_1$ is the vector comprising the outputs of the first layer of neurons. That is, $(\mathbf{x}_1)_i = g_1(\mathbf{w}_i^T \mathbf{x} + b_i)$, where $\mathbf{w}_i$ is the weight vector of the i-th neuron in the first layer, and b_i is the bias of the i-th neuron in the first layer. If we suppose that the activations of the first layer are all identical (this typically does not hurt performance, but allows vectorization, which greatly speeds up training), we can write this in vectorized form as:

$$\mathbf{x}_1 = g_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

where $\mathbf{W}_1$ is the matrix whose i-th column is $\mathbf{w}_i$, and $\mathbf{b}_1$ is the vector whose i-th entry is b_i.

Thus, we can write:

$$
\begin{aligned}
\mathbf{x}_0 &= \mathbf{x} \\
\mathbf{x}_1 &= g_1(\mathbf{W}_1\mathbf{x}_0 + \mathbf{b}_1) \\
\hat{\mathbf{y}} = \mathbf{x}_2 &= g_2(\mathbf{W}_2\mathbf{x}_1 + \mathbf{b}_2)
\end{aligned}
$$

But why stop here? Perhaps we can use the outputs of the second layer to feed into a third layer, and so on. This is the idea behind **feedforward neural networks**, also called **multi-layer perceptrons** (MLPs) (the reason for the name is fairly obvious).

Key Idea 3
We can stack multiple layers of neurons next to each other to form a *deep* neural network.

4 Designing a 2 hidden layered network

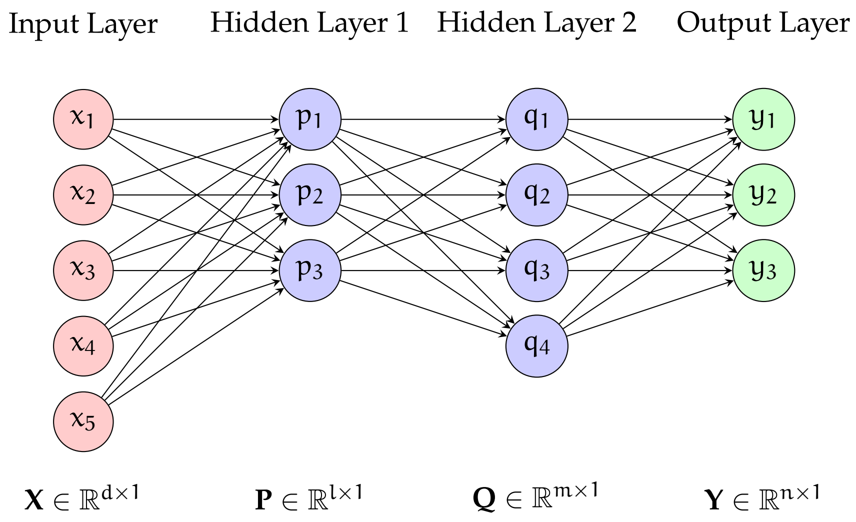

Input Layer
Hidden Layer 1
Hidden Layer 2
Output Layer
x_1
x_2
x_3
x_4
x_5
p_1
p_2
p_3
q_1
q_2
q_3
q_4
y_1
y_2
y_3
$\mathbf{X} \in \mathbb{R}^{d\times 1}$
$\mathbf{P} \in \mathbb{R}^{l\times 1}$
$\mathbf{Q} \in \mathbb{R}^{m\times 1}$
$\mathbf{Y} \in \mathbb{R}^{n\times 1}$

Figure 3: A (tiny) deep neural network. Note the dimensions of the input and output of each layer.

This is an example of a fully-connected network, i.e., all nodes in one layer is connected to every node in the next layer. The weights in the 3 layers are of the dimensions:

The weights in the 3 layers are of the dimensions:

$$
\begin{aligned}
\mathbf{W}^{xp} &\in \mathbb{R}^{l\times d} \\
\mathbf{W}^{pq} &\in \mathbb{R}^{m\times l} \\
\mathbf{W}^{qy} &\in \mathbb{R}^{n\times m}
\end{aligned}
$$

where $\mathbf{W}^{xp}_{ij}$ refers to the weight of the connection from the i^{th} node of the input layer to the j^{th} node of the hidden layer.

Let the activation function across all the layers be $\mathbf{g}(\cdot)$.

The relation between the layers and weights can be written as:

$$
\begin{aligned}
\mathbf{P} &= g(\mathbf{W}^{xp}\mathbf{X} + \mathbf{B}^{xp}) \\
\mathbf{Q} &= g(\mathbf{W}^{pq}\mathbf{P} + \mathbf{B}^{pq}) \\
\mathbf{Y} &= g(\mathbf{W}^{qy}\mathbf{Q} + \mathbf{B}^{qy})
\end{aligned}
$$

Here, **X** is our input, and **Y** is the corresponding classified output. **B** denotes the biases of the nodes in each layer where $\mathbf{B}^{xp} \in \mathbb{R}^{l\times 1}$, $\mathbf{B}^{pq} \in \mathbb{R}^{m\times 1}$ and $\mathbf{B}^{qy} \in \mathbb{R}^{n\times 1}$.

5 The loss function

Before we get too excited about deep neural networks, we need to figure out how to train them. That is, we need to figure out how to choose the weights and biases of the neurons so that the network does what we want it to do. To do this, as before, we need to define a loss function to tell the network how well it is doing. The loss function is a function of the weights and biases of the network, and measures how well the network is doing. The goal is to minimize the loss function.

5.1 Typical loss functions

For regression problems (the output vector of the MLP is a scalar, aka there is only one output neuron), we typically use the squared error loss. Suppose the true output is y, and the output of the MLP upon input x is $\hat{y}$. Then, the squared error loss is defined by

$$\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2 \implies \mathcal{L}(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$$

It is convex in the weights and biases and so behaves well w.r.t gradient descent.

For classification problems, the cross-entropy loss - a generalization of the binary cross-entropy from logistic regression - is typically used. Suppose we have K classes, and so the output vector of the MLP is K-dimensional. Let y be the true label of the input x, and let $\hat{y} \in \mathbb{R}^K$ be the output of the MLP on input x. The vector $\hat{y}$ represents the probability distribution over the classes predicted for input x. The cross-entropy loss is defined as:

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \implies \mathcal{L}(\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}(y^{(i)})_k \log(\hat{y}^{(i)})_k$$

How do we go from the neural network outputs to a probability distribution $\hat{y}_k = P(y = k|x)$ over the classes? We used the sigmoid for binary classification. Here we use a generalization, called the softmax function. The softmax function takes a K-dimensional vector z and outputs a K-dimensional vector $\sigma(z)$, where:

$$\sigma(z)_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_j}}$$

We will see how to compute the gradient of these neural networks with respect to the different weights (and thus, complete the training of neural networks) in the next lecture. We will finish this lecture with a note on the activation functions of the different neurons.

6 Activation functions

This is another crucial aspect that can determine whether your model succeeds or remains stuck at a minimum. Activation functions must be chosen carefully to ensure rapid learning.

If every neuron in the network had a linear activation function, the final output would be a linear function of the input. This would essentially reduce the neural network to a single neuron with too many weights – not very useful (for an example of a linearly activated neural network that learns non-linearities for different reasons, you can refer to this).

Nonlinear activation functions are where neural networks derive their power. Neural networks with just one hidden layer and a nonlinear activation can approximate *any* function with high accuracy. You can gain a deeper (visual) understanding of why this is true by checking this.

We thus need to use non-linear activation functions. Here are some common ones:

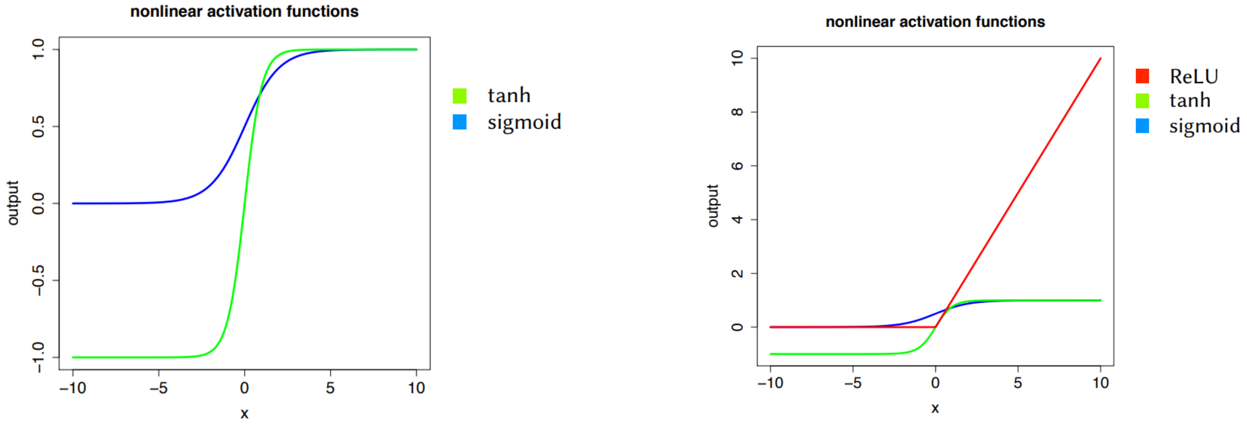
1. **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$. A common choice for the activation function. It is smooth, and bounded. However, it suffers from the vanishing gradient problem: the gradient of the sigmoid function is very small for large positive values of x, and so the weights corresponding to neurons with large activations do not get updated much. This is a problem because the neurons with large activations are the ones that are most important in the network. This slows down learning significantly. Note also that the output of the sigmoid is always positive.

2. **Hyperbolic tangent**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This is similar to the sigmoid function, but is **zero-centered**. This property helps with learning because the output values are centered around zero, which means that both positive and negative values can be learned effectively. However, like the sigmoid function, the hyperbolic tangent still suffers from the **vanishing gradient problem**.

   The hyperbolic tangent function $\tanh(x)$ can be related to the sigmoid function $\sigma(x)$ by $\tanh(x) = 2\sigma(2x) - 1$.

3. **ReLU (Rectified Linear Unit)**: The ReLU activation function is a very popular choice in neural networks due to its simplicity and the fact that it converges extremely quickly during training. It is defined as follows:

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$



(a) tanh and sigmoid suffer from the vanishing gradient problem.

(b) The three activations together.

4. **Many more!**: Leaky ReLU, ELU, SELU, etc.