Advanced Machine Learning **Fall 2020**

Lecture 10: Deep Learning

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

Disclaimer: *These notes are adapted from ETH's Advanced Machine Learning Course, EPFL's CS433 Course, Stanford's CS231N Course and "Neural Networks and Deep Learning" book.*

10.1 Neural Networks

Let us look at the structure of a neural network. It is shown in Figure 10.1. This is a neural net with one input layer of size **D**, **L** hidden layers of size **K**, and one output layer. It is a feedfoward network: the computation performed by the network starts with the input from the left and flows to the right. There is no feedback loop. As always, we assume that our input is a **D**-dimensional vector. We see that there is a node drawn in Figure 1 for each of the D components of x. We denote these nodes by $x_i^{(0)}$, where the superscript (0) specifies that this is the input layer.

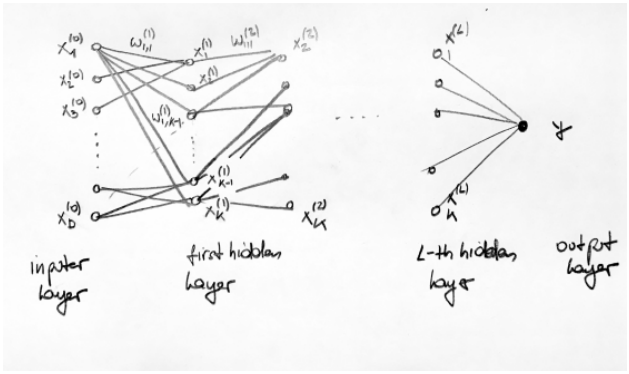


Figure 10.1: A neural network with one input layer, L hidden layers, and one output layer.

Let us assume that there are **K** nodes in each hidden layer, where K is a hyper-parameter. Each node in the hidden layer l, $l = 1, \ldots, L$, is connected to all the nodes in the previous layer via a weighted edge. We denote the edge from node i in layer $l-1$ to the node j in layer l by $w_{i,j}^{(l)}$. The super-script (l) indicates that these are the weights of edges that lead to layer l. The output at the node j in layer l is denoted by $x_j^{(l)}$ and it is given by:

$$x_j^{(l)} = \phi\Big(\sum_i w_{i,j}^{(l)} x_i^{(l-1)} + b_j^{(l)}\Big)$$

In simple words, in order to compute the output we first compute the weighted sum of the inputs and then apply a function ϕ to this sum.

A few remarks:

- The constant term $b_j^{(l)}$ is called the **bias term** and is a parameter like any of the weights $w_{i,j}^{(l)}$. The learning part will consist of choosing all these parameters appropriately for the task.
- The function ϕ is called the **activation function**. It is crucial that this function is **non-linear**. Why is this? Assume not, then the whole neural-net would just be a highly factorized linear function of the input data and there would be no gain compared to standard linear regression/classification.

10.1.1 Representation Power

How "powerful" are neural nets? More precisely, what functions $f(x)$ can they represent, or better, what functions can they approximate? Before we get into our heuristic argument let us state the main theorem of the paper by Barron: "Universal approximation bounds for superpositions of a sigmoidal function". This gives you a flavor of what kind of results can be proved.

Lemma 10.1 *Let $f : \mathbb{R}^D \to \mathbb{R}$ be a function such that*

$$\int_{\mathbb{R}^D} |w||\tilde{f}(w)|dw \leq C$$

where

$$\tilde{f}(w) = \int_{\mathbb{R}^D} f(x)e^{-jw^T x}dx$$

is the Fourier transform of $f(x)$

Then for all $n \geq 1$, there exists a function f_n of the form

$$f_n(x) = \sum_{j=1}^{n} c_j \phi(x^T w_j + b_j) + c_0$$

i.e., a function that is representable by a NN with one hidden layer with n nodes and "sigmoid-like" activation functions so that

$$\int_{|x|\leq r} (f(x) - f_n(x))^2 dx \leq \frac{(2Cr)^2}{n}$$

Let's discuss this result:

- First note that the condition on the Fourier transform is a "smoothness condition." E.g., functions so that $\int_{\mathbb{R}^D} |w||\tilde{f}(w)|dw \leq \infty$ can be shown to be continuously differentiable.
- Second note that the lemma only guarantees a good approximation in a **bounded domain**. The larger the domain, the more nodes we need in order to approximate a function to the same level (see the term r^2, where r is the radius of the ball where we want the approximation to be good, in the upper bound).
- Third, this is an approximation "in average", more precisely in $\mathbb{L}_2$-norm.
- Fourth, the approximation f_n with n terms corresponds exactly to our model of a neural net with one hidden layer containing n nodes and sigmoids as activation functions.

- Fifth, the theorem applies to all activation functions that are "sigmoid-like," i.e., all activation functions whose left limit is 0, whose right limit is 1, and that are sufficiently smooth.

In simple words, the lemma says that a sufficiently "smooth" function can be approximated by a neural net with one hidden layer and the approximation error goes down like one over the number of nodes in the hidden layer. Note that this is a very fast convergence.

10.1.2 Approximation in Average

We start with a scalar function $f(x) : \mathbb{R} \to \mathbb{R}$ on a bounded domain. Recall that if this function is continuous then it is Riemann integrable, i.e., it can be approximated arbitrarily closely by "upper" and "lower" sums of rectangles, see Figure 10.2. Of course, we might need a lot of such rectangles to approximate the area with an error of at most ϵ, but for every $\epsilon > 0$ we can find such an approximation.

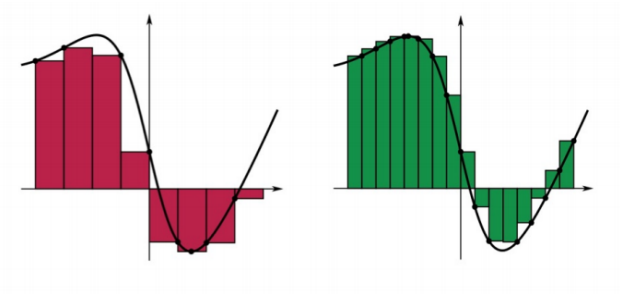


Figure 10.2: A lower and an upper Riemann sum.

We will now show that if we do not limit the weights, then with two hidden nodes (of a neural network with one hidden layer) we can construct a function which is arbitrarily close to a given rectangle. But since, as we have just seen, a finite number of rectangles suffices to approximate a bounded continuous function arbitrarily closely, it follows that with a finite number of hidden nodes of a neural network with one hidden layer we can approximate any such function arbitrarily closely.

Let $\phi(x) = \dfrac{1}{1+e^{-x}}$ be the sigmoid function. Consider the function $f(x) = \phi(w(x-b))$, where w is the weight of a particular edge and $-wb$ is the bias term.
If we want to create a rectangle that jumps from 0 to 1 at $x = a$ and jumps back to 0 at $x = b$, $a < b$, then we can accomplish this by taking

$$\phi(w(x-a)) - \phi(w(x-b))$$

and taking w very large (see Figure 10.3).

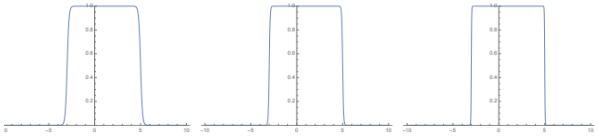


Figure 10.3: An approximate rectangle of the form $\phi(w(x-a)) - \phi(w(x-b))$ with $w = 10$, 20, and 50, respectively.

Note that these "rectangles" have a very simple representation in form of a neural network. This is shown in Figure 10.4. There is one input node which contains the value x. This value is multiplied by some large

weight (in the figure it is 50) and it is then forwarded to the two hidden nodes. One of these hidden nodes has a bias of 150 the other one has a bias of -250, so that the sums at these two hidden nodes are $50(x+3)$ and $50(x-5)$, respectively. Each node applies the sigmoid function and forwards the result to the output layer. The edge from the top hidden node to the output has weight 1 and the one from the bottom hidden node to the output has weight -1. The output node adds the two inputs. The result is $\phi(50(x+3)) - \phi(50(x-5))$, which is approximately a unit-height rectangle from -3 to 5.

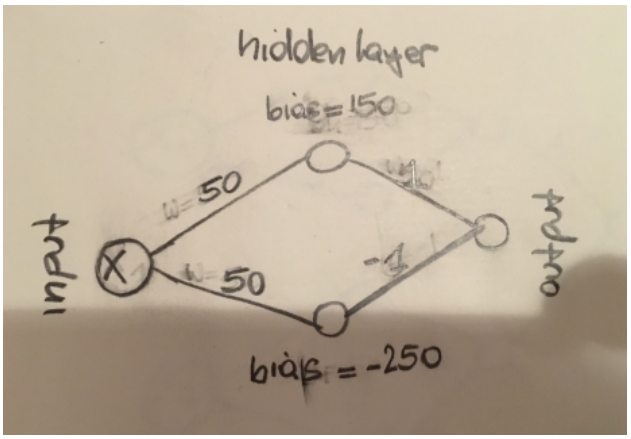


Figure 10.4: A simple NN implementation of a unit-height rectangle from -3 to 5

It is hopefully clear at this point why any continuous function on a bounded domain can be approximated via a neural network with one hidden layer. Let us summarize in telegram style: Take the function. Approximate it in the Riemann sense. Approximate each of the rectangles in the Riemann sum by means of two nodes in the hidden layer of a neural net. Compute the sum (with appropriate sign) of all the hidden layers at the output node. If we are using a Riemann sum with **K** rectangles we get therefore a neural network approximation with one hidden layer containing **2K** nodes.

10.1.3 Activation Functions

There are many activation functions that are being used in practice. Let us list here some of them and briefly discuss their merits.

Sigmoid We start with the sigmoid $\phi(x)$, which we have encountered already several times. Just to summarize, it is defined by:

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

and a plot is shown in Figure 10.5. Note that the sigmoid is always positive (not really an issue) and that it is bounded.
Further, for $|x|$ large, $\phi'(x) \approx 0$. This can cause the gradient to become very small (which is known as the "**vanishing gradient problem**"), sometimes making learning slow.

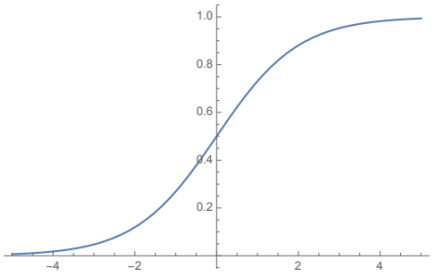


Figure 10.5: The sigmoid function $\phi(x)$.

Tanh Very much related to the sigmoid is $tanh(x)$. It is defined by

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\phi(2x) - 1$$

and a plot is shown in Figure 10.6.

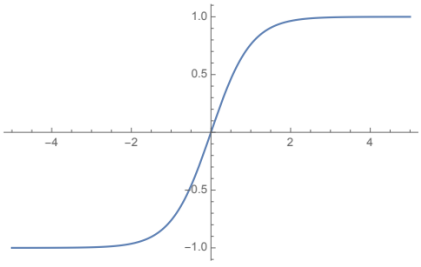


Figure 10.6: The function $\tanh(x)$.

Note that $tanh(x)$ is "balanced" (positive and negative) and that it is bounded. But it has the same problem as the sigmoid function, namely for $|x|$ large, $tanh'(x) \approx 0$. As mentioned before, this can cause the gradient to become very small, sometimes making learning slow.

Rectified linear Unit – ReLU Very popular is the rectified linear unit (ReLU) , which is defined by

$$(x)_+ = max\{0, x\}$$

and a plot is shown in Figure 10.7.

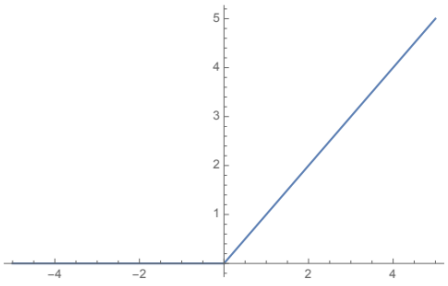


Figure 10.7: The ReLU $(x)_+$.

Note that the ReLU is always positive and that it is unbounded. One nice property of the ReLU is that its derivative is 1 (and does not vanish) for positive values of x (it has 0 derivative for negative values of x though).

Leaky ReLU In order to solve the 0-derivative problem of the ReLU (for negative values of x) one can add a very small slope α in the negative part. This gives rise to the leaky rectified linear unit (LReLU), which is defined by

$$f(x) = max\{\alpha x, x\}$$

and a plot is shown in Figure 10.8. The constant α is of course a hyper-parameter that can be optimized.

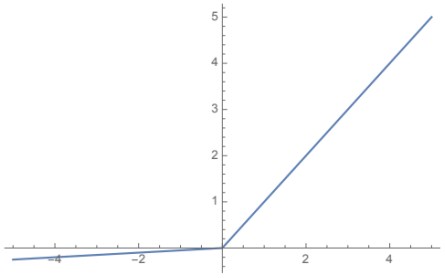


Figure 10.8: LReLU with $\alpha = 0.05$.

10.1.4 Compact Description of Output

Let us start by writing down the output as a function of the input explicitly in compact form. It is natural and convenient to describe the function that is implemented by each layer of the network separately at first. The overall function is then the composition of these functions.

Let $\mathbf{W}^{(l)}$ denote the weight matrix that connects layer $l-1$ to layer l. The matrix $W^{(1)}$ is of dimension $D \times K$, the matrices $W^{(l)}$, $2 \leq l \leq L$, are of dimension $K \times K$, and the matrix $W^{(L+1)}$ is of dimension $K \times 1$. The entries of each matrix are given by

$$W_{i,j}^{(l)} = w_{i,j}^{(l)}$$

where we recall that $w_{i,j}^{(l)}$ is the weight on the edge that connects node i on layer $l-1$ to node j on layer l. Further, let us introduce the bias vectors $\mathbf{b}^{(l)}$, $1 \leq l \leq L+1$, that collect all the bias terms. All these vectors are of length K, except the term $b^{(L+1)}$, that is a scalar.

With this notation we can describe the function that is implemented by each layer in the form:

$$x^{(l)} = f^{(l)}(x^{(l-1)}) = \phi((W^{(l)})^T x^{(l-1)} + b^{(l)})$$

where the (generic) activation function is applied point-wise to the vector. The overall function $y = f(x^{(0)})$ can then be written in terms of these functions as the composition:

$$f(x^{(0)}) = f^{(L+1)} \circ \ldots \circ f^{(2)} \circ f^{(1)}(x^{(0)}).$$

10.1.5 The Backpropagation Algorithm

The cost function can be written as:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (y_i - f^{(L+1)} \circ \ldots \circ f^{(2)} \circ f^{(1)}(x_i))^2$$

Note that this cost function is a function of all weight matrices and bias vectors and that it is a composition of all the functions describing the transformation at each layer.
Note also that the specific form of the loss does not really matter for the workings of the back propagation algorithm that we now discuss. Just to be specific we stick to the square loss. Only the initialization of the back recursion changes if we pick a different loss function.

In SGD we compute the gradient of this function with respect to one single sample. Therefore, we start with the function:

$$\mathcal{L}_n = (y_n - f^{(L+1)} \circ \ldots \circ f^{(2)} \circ f^{(1)}(x_n))^2$$

Recall that our aim is to compute:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}},\ l = 1, \ldots, L+1$$
$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}},\ l = 1, \ldots, L+1$$

It will be convenient to first compute two preliminary quantities. The desired derivatives are then easily expressed in terms of those quantities. Let:

$$z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$$

where $x^{(0)} = x_n$ and $x^{(l)} = \phi(z^{(l)})$. In simple words, $z^{(l)}$ is the input at the l-th layer before applying the activation function. These quantities are easy to compute by a **forward pass** in the network.
More precisely, start with $x^{(0)} = x_n$ and then apply this recursion for $l = 1, \dots, L+1$, first always computing $z^{(l)}$ and then computing $x^{(l)} = \phi(z^{(l)})$.

Further, let

$$\delta_j = \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}$$

Let $\delta^{(l)}$ be the corresponding vector at level l. Whereas the quantities $z^{(l)}$ were easily computed by a forward pass, the quantities $\delta^{(l)}$ are easily computed by a **backwards pass**:

$$\begin{aligned}\delta_j^{(l)} &= \frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\ &= \sum_{k=1}^{K} \delta_k^{(l+1)} W_{j,k}^{(l+1)} \phi'(z_j^{(l)})\end{aligned}$$

The sum comes from the fact the loss $\mathcal{L}_n$ at the layer $l+1$ is the sum of the losses evaluated at each of the neurons (z_k) in that layer, then we applied the chain rule. In vector form, we can write this as:

$$\delta^{(l)} = (W^{(l+1)} \delta^{(l+1)}) \odot \phi'(z^{(l)})$$

where $\odot$ denotes the Hadamard product. Now that we both have $z^{(l)}$ and $\delta^{(l)}$ let us get back to our initial goal:

$$\frac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial w_{i,j}^{(l)}} = \overbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}^{\delta_j^{(l)}} \overbrace{\frac{\partial z_j^{(l)}}{\partial w_{i,j}^{(l)}}}^{x_i^{(l-1)}} = \delta_j^{(l)} x_i^{(l-1)}$$

Why could we drop the sum in the above expression? When we change the weight $w_{i,j}^{(l)}$ then it only changes the sum $z_j^{(l)}$. All other sums at level l stay unchanged.
In a similar manner,

$$\frac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}_n}{\partial z_k^{(l)}} \frac{\partial z_k^{(l)}}{\partial b_j^{(l)}} = \overbrace{\frac{\partial \mathcal{L}_n}{\partial z_j^{(l)}}}^{\delta_j^{(l)}} \overbrace{\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}}^{1} = \delta_j^{(l)}$$

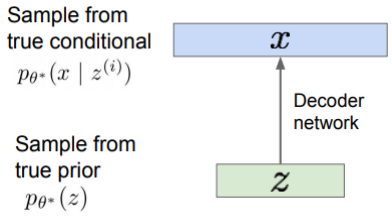
Now we can outline the steps of the training algorithm:

- **Forward Pass**: Set $x^{(0)} = x_n$. Compute for $l = 1, \dots L+1$: $z^{(l)} = (W^{(l)})^T x^{(l-1)} + b^{(l)}$ and $x^{(l)} = \phi(z^{(l)})$
- **Backward Pass**: Set $\delta^{(L+1)} = -2(y_n - x^{(L+1)}a)\phi'(z^{(L+1)})$. Compute for $l = L, \dots 1$: $\delta^{(l)} = (W^{(l+1)}\delta^{(l+1)}) \odot \phi'(z^{(l)})$
- **Final Computation**: For all parameters compute $\dfrac{\partial \mathcal{L}_n}{\partial w_{i,j}^{(l)}} = \delta_j^{(l)} x_i^{(l-1)}$ and $\dfrac{\partial \mathcal{L}_n}{\partial b_j^{(l)}} = \delta_j^{(l)}$

Now that we have the gradient with respect to all parameters, the SGD algorithm makes a small step in the direction opposite to the gradient, then picks a new sample (x_n, y_n), and repeats.

10.2 Variational Autoencoders

Assume that the training data $\{x^{(i)}\}_{i=1}^N$ is generated from an underlying unobserved (latent) representation $\mathbf{z}$. For example, imagine that $\mathbf{x}$ is an image and $\mathbf{z}$ are the latent factors used to generate $\mathbf{x}$: attributes, orientation, etc.



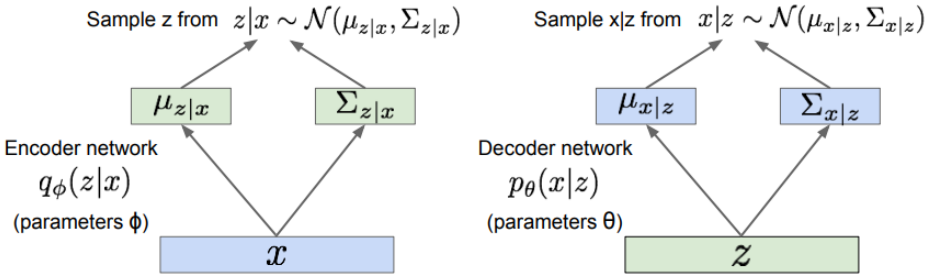
We want to estimate the true parameters θ^* of this generative model:

- We choose the prior $p(z)$ to be simple, e.g. Gaussian.
- The conditional distribution $p(x|z)$ on the other hand is complex. We can model it with a neural network.
- We train the model to maximize the likelihood of the training data:

$$p_\theta(x) = \int \overbrace{p_\theta(z)}^{\text{Gaussian Prior}} \overbrace{p_\theta(x|z)}^{\text{Decoder NN}} dz$$

However, the problem with this approach is that the **integral is intractable**: it is not feasible to compute $p(x|z)$ for every z.

To overcome this, we can define an additional encoder network $q_\phi(z|x)$ that approximates $p_\theta(z|x)$ (note that $p_\theta(z|x)$ is intractable as well because of the integral computation).



Now, we can find a lower bound (Evidence Lower Bound) for the likelihood of the training data:

$$\log p_\theta(x^{(i)}) = \mathbb{E}_{z\sim q_\phi(z|x^{(i)})}\Big[\log p_\theta(x^{(i)})\Big] \qquad (p_\theta(x^{(i)}) \text{ does not depend on } z)$$

$$= \mathbb{E}_z\Big[\log \frac{p_\theta(x^{(i)}|z)p_\theta(z)}{p_\theta(z|x^{(i)})}\Big] \qquad \text{(Bayes Rule)}$$

$$= \mathbb{E}_z\Big[\log \frac{p_\theta(x^{(i)}|z)p_\theta(z)}{p_\theta(z|x^{(i)})}\frac{q_\phi(z|x^{(i)})}{q_\phi(z|x^{(i)})}\Big] \qquad \text{(Multiply and divide by q)}$$

$$= \mathbb{E}_z\Big[\log p_\theta(x^{(i)}|z)\Big] - \mathbb{E}_z\Big[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z)}\Big] + \mathbb{E}_z\Big[\log \frac{q_\phi(z|x^{(i)})}{p_\theta(z|x^{(i)})}\Big] \qquad \text{(Logarithms property)}$$

$$= \overbrace{\mathbb{E}_z\Big[\log p_\theta(x^{(i)}|z)\Big] - D_{KL}\Big(q_\phi(z|x^{(i)}) \,||\, p(z)\Big)}^{= \mathcal{L}_{\theta,\phi} \text{ (ELBO)}} + \overbrace{D_{KL}\Big(q_\phi(z|x^{(i)}) \,||\, p_\theta(x^{(i)}|z)\Big)}^{\geq 0}$$

Finally, we can find the optimal parameters optimizing the ELBO:

$$\theta^*, \phi^* = argmax_{\theta,\phi} \sum_{i=1}^{N} \mathcal{L}(x^{(i)}, \theta, \phi)$$