# Lecture 4: Neural Networks

Adityanarayanan Radhakrishnan

September 25, 2024

## 1 Introduction

Thus far, we have studied nonlinear models through the kernel regression framework, which involved transforming the features in a nonlinear way and then performing linear regression. An alternative strategy to building nonlinear models is to keep the features unchanged and instead, make the model nonlinear in parameters. In this lecture, we will introduce neural networks, which serve as one example of such a nonlinear model that has achieved impressive empirical results over the previous decade. While there is a vast literature on neural networks, we will focus on the following fundamentals:

1. Network architectures - fully connected networks, width, depth, and activation functions.
2. Training procedures - gradient descent, minibatch gradient descent.
3. Initialization schemes - zero initialization, standard initializations.

In addition, we will provide a list of guidelines for training neural networks to help avoid some common coding errors. We refer the reader to [2] for a broad, high-level overview of these models.

## 2 Fully Connected Neural Networks (FCNNs)

We will begin by defining fully connected neural networks (FCNNs).[1]

**Definition 1** (FCNN)**.** *A **fully connected neural network (FCNN)**, $f : \mathbb{R}^d \to \mathbb{R}$, of depth $L$ has the form:*

$$f(x) = W^{(L)}\phi(W^{(L-1)}\phi(\dots W^{(2)}\phi(W^{(1)}x)\dots);$$

*where $\phi : \mathbb{R} \to \mathbb{R}$ is an elementwise nonlinearity, $W^{(i)} \in \mathbb{R}^{k_i \times k_{i-1}}$ are weight matrices of width $k_i$ with $k_L = 1$ and $k_0 = d$.*

When we want to make dependence on the weight matrices $\{W^{(i)}\}_{i=1}^n$ explicit, we denote the network by $f_{\mathbf{w}}$ where $\mathbf{w} \in \mathbb{R}^{\sum_{i=1}^L k_i k_{i-1}}$ is a vector of all parameters in the weight matrices $\{W^{(i)}\}_{i=1}^n$.

**Commonly used nomenclature.** The intermediate outputs in a neural network (e.g. $\phi(W_1x)$, $\phi(W_2\phi(W_1x))$, etc.) are typically referred to as hidden layer representations. Pre-activation representations refer to those hidden layer representations without a nonlinearity applied (e.g. $W_1x$) and post-activation representations refer to those with nonlinearity applied (e.g. $\phi(W_1x)$). In this course, depth refers to the number of weight matrices. In the literature, it is common to also equate depth with the number of hidden layer representations (e.g. a depth 2 network in our course may be referred to as a 1-hidden layer network).

**Example 1** (Depth 2, Width $k$ ReLU FCNN)**.** *The rectified linear unit (ReLU) activation is defined by $\phi(z) = \max(z, 0)$ for $z \in \mathbb{R}$. A depth 2, width $k$ ReLU FCNN is given by*

$$f(x) = A\phi(Bx) = \sum_{i=1}^{k} A_i\phi(B_{i,:}x);$$

[1] We note that the reader may also see the term multi-layer perceptron (MLP) used to refer to such networks.

*where $A \in \mathbb{R}^{1\times k}$ and $B \in \mathbb{R}^{k\times d}$.*

**Standardly used activation functions.** While activation function selection is an area of active research, we recommend using the ReLU activation above or the Leaky ReLU variant given by

$$\phi(z) = \begin{cases} z & \text{if } z \geq 0 \\ cz & \text{if } z < 0 \end{cases} ;$$

where the constant $c > 0$ is typically chosen as $c = 0.01$. Practitioners sometimes grid search over activation function to see which one gives the best performance, and some other activation function choices can be found in [6].

# 3 Training Neural Networks with Gradient Descent

Now that we have defined a basic neural network architecture, we will train such networks on a dataset $(X, y) \subset \mathbb{R}^{d\times n} \times \mathbb{R}^{1\times n}$ using gradient descent. Recall, that the matrix $X$ consists of $n$ training samples $\{x^{(p)}\}_{p=1}^n$ and the vector $y$ consists of $n$ training labels $\{y^{(p)}\}_{p=1}^n$. Let $f_{\mathbf{w}}$ denote an $L$ hidden layer neural network with activation $\phi$. To train $f_{\mathbf{w}}$ on the dataset $(X, y)$, we first set up a loss function. In particular, we will use the following mean squared error for our loss function:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{n}(y^{(p)} - f_{\mathbf{w}}(x^{(p)}))^2.$$

Given an initial set of weights $\mathbf{w}^{(0)}$ and a learning rate $\eta$, we can use gradient descent to update the weights as follows:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}^{(t)}) = \mathbf{w}^{(t)} + \eta\sum_{p=1}^{n}(y^{(p)} - f_{\mathbf{w}^{(t)}}(x^{(p)}))\nabla f_{\mathbf{w}^{(t)}}(x^{(p)}) \ . \tag{1}$$

Training a neural network in this manner is also known as *back-propagation*. We provide an example below to make this computation explicit.

**Example 2** (Gradient descent for Depth 2, Width $k$ ReLU FCNN)**.** *Let $f(x) = A\phi(Bx)$ denote a ReLU FCNN of depth 2 and width $k$. We will next write the update rules explicitly by grouping the vector $\mathbf{w}$ into two components: one corresponding to the weights in $A$ and the other corresponding to weights in $B$. We will first compute the gradients of $f_{\mathbf{w}}$ with respect to the weights $A_i$ and then with respect weights $B_{ij}$. Suppressing the dependence on $\mathbf{w}$, recall that*

$$f(x) = \sum_{i=1}^{k} A_i\phi(B_{i,:}x).$$

*Thus, the partial derivative with respect to the weights is given by*

$$\frac{\partial f}{\partial A_i} = \phi(B_{i,:}x) \ ;$$
$$\frac{\partial f}{\partial B_{ij}} = A_i\phi'(B_{i,:}x)x_j \ .$$

*Using these expressions and substituting back into Eq.* (1)*, we find that:*

$$A_i^{(t+1)} = A_i^{(t)} + \eta\sum_{p=1}^{n}\left(y^{(p)} - f_{\mathbf{w}^{(t)}}(x^{(p)})\right)\phi(B_{i,:}^{(t)}x) \ ;$$

$$B_{ij}^{(t+1)} = B_{ij}^{(t)} + \eta\sum_{p=1}^{n}\left(y^{(p)} - f_{\mathbf{w}^{(t)}}(x^{(p)})\right)A_i^{(t)}\phi'(B_{i,:}^{(t)}x^{(p)})x_j^{(p)} \ .$$

*We can write these terms concisely through vectorization as follows:*

$$A^{(t+1)} = A^{(t)} + \eta \sum_{p=1}^{n} \left( y^{(p)} - f_{\mathbf{w}^{(t)}}(x^{(p)}) \right) \phi(B^{(t)} x)^T \ ;$$

$$B^{(t+1)} = B^{(t)} + \eta \sum_{p=1}^{n} \left( {A^{(t)}}^T \odot \phi'(B^{(t)} x^{(p)}) \right) \left( y^{(p)} - f_{\mathbf{w}^{(t)}}(x^{(p)}) \right) {x^{(p)}}^T \ .$$

**Mini-batch Gradient Descent.** In the formulation of gradient descent above, we update the weights based on the gradient of the loss over all training samples. In general, computing the gradient over all samples can lead to memory issues especially when using the GPU. To overcome this limitation, we can instead evaluate the loss for a handful of samples and update the weights based on this subset of gradients. This procedure is referred to as *mini-batch gradient descent*. Importantly, in addition to overcoming memory issues, mini-batch gradient descent can speed up training [1, 4].

**Definition 2** (Mini-batch Gradient Descent)**.** *Given a loss function $\mathcal{L}(w, x, y)$ for a sample $(x, y)$, an initialization $w^{(0)}$ and data $\{(x^{(i)}, y^{(i)}\}_{i=1}^{n}$,* ***mini-batch gradient descent*** *with step size $\eta > 0$ proceeds according to the following steps:*

*Step 1: Sample $m$ points $\{(x^{(j)}, y^{(j)}\}_{j=1}^{m}$.*

*Step 2:* $$w^{(t+1)} = w^{(t)} - \eta \sum_{j=1}^{m} \nabla_w \mathcal{L}(w^{(t)}, x^{(j)}, y^{(j)}) \ .$$

**Remarks.** When $m = 1$ in the definition above, mini-batch gradient descent is referred to as **stochastic gradient descent**. Note that we have been intentionally general about the sampling procedure in step 1 above. In particular, several sampling procedures exist, and we describe some standard ones below:

1. *Round robin sampling.* We partition the $n$ points into groups of $m$ with every example appearing in exactly one group.[2] We then iterate over all groups exactly once. A full iteration over this group is referred to as an **epoch**.
2. *Uniform random sampling.* We uniformly sample $m$ points from the total $n$ samples.
3. *Balanced sampling.* When there is an imbalance in the data (e.g. in classification, when one class has more examples than the other), it is often useful to use a sampling strategy which increases the likelihood of sampling from the minority class. This ensures that the model will not simply predict the majority class label for all samples.

**Other optimization methods.** In addition to gradient descent and mini-batch gradient descent described above, there are roughly thirteen different optimization methods (referred to as optimizers) available in PyTorch. Several of these optimizers are variants of gradient descent that incorporate additional techniques such as pre-conditioners or momentum to speed up training. How should one go about choosing an effective optimization method for neural networks? Unfortunately, the typical approach is to just grid search over many different optimization methods and select the one that gives best performance. Below, I provide some simple recommendations for optimizer selection based on my own experience training neural networks.

**My optimizer recommendations.**

In my experience, if I want to train a neural network quickly without necessarily squeezing out the best possible performance, I choose to use the `Adam` optimizer with learning rate of $10^{-4}$. While occasionally slower to train, I have empirically found mini-batch gradient descent (with learning rates of 0.01 or 0.1) to give slightly improved results over the Adam optimizer on select tasks (e.g. classifying images from the CIFAR-10 dataset [3]).

[2] When $n$ is not divisible by $m$, we have one group with fewer than $m$ samples.

**Initialization Schemes.** We now briefly discuss some standardly used initialization schemes and in the subsequent lectures, we will rationalize these schemes. First, unlike linear and kernel regression, we note that we cannot initialize all parameters of a neural network to start at 0. If we use a zero initialization scheme for more than 1 layer, then training will never update the weights. We examine this case below for the 2 layer ReLU FCNN below.

**Lemma 1** (Zero initialization for 2 layer ReLU FCNN)**.** *Let $f(x) = A\phi(Bx)$ denote a 2 layer ReLU FCNN. Suppose $A_i^{(0)}, B_{ij}^{(0)} = 0$ for all $i \in [k], j \in [d]$, then $A_i^{(t)} = 0, B_{ij}^{(t)} = 0$ for all timesteps $t$.*

*Proof.* From the gradient descent update rule, we have that:

$$A_i^{(1)} = A_i^{(0)} + \eta \sum_{p=1}^{n} \left(y^{(p)} - f_{\mathbf{w}^{(0)}}(x^{(p)})\right) \phi(B_{i,:}^{(0)} x) = 0 \ ;$$

$$B_{ij}^{(1)} = B_{ij}^{(0)} + \eta \sum_{p=1}^{n} \left(y^{(p)} - f_{\mathbf{w}^{(0)}}(x^{(p)})\right) A_i^{(0)} \phi'(B_{i,:}^{(0)} x^{(p)}) x_j^{(p)} = 0 \ .$$

We leave it to the reader to verify that the result then follows by induction on the time step $t$. □

The exercises will allow you to verify that for the ReLU activation, $B^{(0)}$ cannot be initialized at 0, but $A^{(0)}$ can be initialized at 0. The default initialization scheme in several deep learning libraries closely follows the so-called "standard initialization" scheme defined below.

**Standard Initialization.** Given a depth $L$ FCNN with activation $\phi : \mathbb{R} \to \mathbb{R}$ and weight matrices $\{W^{(i)}\}_{i=1}^{L}$ where $W^{(i)} \in \mathbb{R}^{k_i \times k_{i-1}}$, we initialize according to:

$$W_{j\ell}^{(i)} \overset{i.i.d}{\sim} \mathcal{N}\left(0, \frac{c}{k_i}\right) \ ,$$

where $c$ satisfies $c\mathbb{E}_z[\phi(z)^2] = 1$ with $z \sim \mathcal{N}(0,1) = 1$ (for ReLU activation, $c = 2$).

In the following lectures, we will explain why this is a sensible initialization scheme that is stable upon increasing network width. In addition, we will derive another initialization scheme known as the Neural Tangent Kernel (NTK) initialization, which will allow us to train infinitely wide neural networks.

**Complexity in training neural networks.** Unlike the case for linear and kernel regression, training neural networks now involves solving a non-convex optimization problem in the weights $\mathbf{w}$. The difficulty with such problems is that we no longer have simple, closed form solutions nor do we have theoretical guarantees that we will converge to a solution that minimizes the training loss. Moreover, architecture choice, initialization scheme, and training procedure all effect the solution learned through training, and we want to ensure that we choose these aspects to get the best solution possible. Thus, while neural networks offer a lot of flexibility, such flexibility comes at a cost of increased complexity in modeling choice.

**Tips for training FCNNs.** Given the several steps involved in training neural networks, it can be difficult to identify any problems that may arise during implementation. To mitigate such difficulties, we provide some tips below for successfully training fully connected neural networks.

**Training tips for fully connected neural networks.**

1. Start with a 2 layer (1-hidden layer) ReLU network of width 128. Make sure that the bias terms are not being used (in PyTorch, set `bias=False`).
2. Train on 1 training example using mean squared error for 1000 epochs using either: (1) the `Adam` optimizer with learning rate $10^{-4}$ or (2) gradient descent with learning rate of 0.01.
3. If the loss does reach values of lower than $10^{-4}$, then there is likely a bug in the code and one should verify the following:
   (a) Check that the gradients are zeroed out during every step of training before the loss is computed.
   (b) Check the data point and the output of the network at each step of gradient descent. Some networks, such as those without bias terms, will always map 0 inputs to 0.
   (c) Try shrinking the learning rate further to see if the loss decreases.
   (d) Try increasing the number of epochs to see if the loss continues to decrease.
4. If the loss converges to near zero (i.e. $10^{-4}$ or less), then try training on the entire dataset.
5. Try introducing bias terms (`bias=True`) and re-training.
6. Increase the width from 128 to higher values and track how training and test loss change.
7. Increase the depth from 1 to higher values and track how training and test loss change.

**Important:** Please do not start with a model that uses more complex layers such as BatchNorm or Dropout until you have verified that the above works.

**Essential:** Ensure that your network is placed in inference mode for test samples and training mode for training samples. In PyTorch, this is done using the `model.eval` and `model.train` flags respectively. This is necessary when using BatchNorm or Dropout layers as the behavior of these layers changes during inference time.

# 4 Autoencoders: Neural Networks for Unsupervised Learning

We now provide a concrete example to illustrate how varying architecture and initialization scheme can drastically impact the solution learned through training. Namely, we will turn to fully connected autoencoders, which are neural networks commonly used for unsupervised learning problems such as clustering and anomaly detection. Autoencoders form the backbone of several popular deep learning applications today including state-of-the-art models for image generation. We begin with the formulation of autoencoders below.

**Definition 3** (Fully Connected Autoencoder)**.** *Given a dataset $X \in \mathbb{R}^{d\times n}$ of samples $\{x^{(p)}\}_{p=1}^n$, a **fully connected autoencoder** is a fully connected network $f_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}^d$ that is trained to minimize the loss*

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{n} \|x^{(p)} - f_{\mathbf{w}}(x^{(p)})\|_2^2 \ .$$

For simplicity, in the remainder of the section, let us assume that $f_{\mathbf{w}}$ is given by a 2 layer, width $k$ network, i.e.,

$$f(x) = A\phi(Bx);$$

where $A \in \mathbb{R}^{d\times k}$, $B \in \mathbb{R}^{k\times d}$ and $\phi$ is an elementwise nonlinearity.

**Classical intuition behind autoencoding.** At first, it would appear that autoencoding simply involves learning a map from each data point to itself. In particular, the identity function ($f_{\mathbf{w}} = I_{d\times d}$) is a simple map that maps each sample to itself and would yield $\mathcal{L}(\mathbf{w}) = 0$. What is thus the advantage of training a model to learn such a map?

The original idea behind autoencoders was to enforce that the width $k$ is less than the dimension so that the model cannot learn the identity map. Instead, the model would be forced to learn a low dimensional hidden representation of the data that can be used to accurately reconstruct the data. To see this mathematically, let us consider the case of a linear autoencoder, i.e., $\phi(z) = z$.

**Example 3** (Linear Autoencoder)**.** *A linear autoencoder is a network of the form:*

$$f(x) = ABx \ . \tag{2}$$

*If $k < d$, then the operator $AB$ has rank at most $d$ while the identity map has rank equal to $d$. Hence, such a model cannot learn the identity map. Training a linear autoencoder thus learns an embedding, $z = Bx$, for each point $x$ such that $Az \approx x$.*

*On the other hand, if $k = d$, we can easily set $A = B = I$, and so such an autoencoder can learn the identity map. An analogous statement holds for $k > d$.*

Thus, since their inception around 40 years ago in 1986, autoencoders have been introduced with the widely believed claim that layer width should be less than data dimension so that these models can learn low dimensional representations of data.

Note that in Example 3 above, when $k \geq d$, we showed that it was possible to initialize $A, B$ to be the identity. Yet, we never actually demonstrated that if we initialize $A, B$ differently, then training will recover the identity function for both $A, B$. For example, we can initialize $A = (XX^T)^\dagger$ and $B = (XX^T)$. We leave it to the reader to check that $ABX = X$, and so we minimize the squared loss on the training data. On the other hand, $Bx$ for any test sample $x$ is now a projection onto the span of the training data. If the data $X$ is not full rank, then this is quite a useful representation of data, in particular, for applications like anomaly detection. For example, if a test sample does not lie in the span of the training data (i.e., it is an anomaly), then its reconstruction under this autoencoder would simply be 0. On the other hand, any sample lying in the span of the training data would be reconstructed perfectly.

Moreover, as is discussed in [5], remarkably more is true for nonlinear over-parameterized autoencoders, i.e. those that have width larger than the input. Namely, while such models again *can* learn the identity map, training finds a solution that is contractive around training examples.

# 5 Discussion

In this lecture, we provided a brief introduction to neural networks. We defined the fully connected architecture, depth, width, and activation function. We then demonstrated how to use gradient descent (and mini-batch gradient descent) to train neural networks. We provided some commonly used initialization schemes for network weights for running gradient descent. We then provided some tips for effectively training fully connected networks. Lastly, we used the example of autoencoders to demonstrate how different choices in neural network modeling (in particular, initialization scheme) can drastically effect the solution learned through training.

Overall, we demonstrated that neural networks provide significant modeling flexibility and indeed, such flexibility has been leveraged to successfully empower various applications. Yet, such flexibility unfortunately also leads to increased complexity, and it remains difficult to provide a set of guiding principles for understanding which network and hyper-parameters will work best for a given application.

On the other hand, by connecting neural networks with the kernel regression framework, we can combine the flexibility of neural networks with the simplicity of kernel machines in order to build simple and effective models. In the next lecture, we will provide our first connection between neural networks and kernel machines by considering the infinite width limit of neural networks where only the last layer is trained.

# References


[1] R. Bassily, M. Belkin, and S. Ma. On exponential convergence of sgd in non-convex over-parametrized learning. *arXiv:1811.02564*, 2018.

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*, volume 1. MIT Press, 2016.

[3] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[4] S. Ma and M. Belkin. Diving into the shallows: a computational perspective on large-scale shallow learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.

[5] A. Radhakrishnan, M. Belkin, and C. Uhler. Overparameterized neural networks implement associative memory. *Proceedings of the National Academy of Sciences*, 44(117):27162–27170, 2020.

[6] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. In *International Conference on Learning Representations*, 2017.