CIS5200: Machine Learning Spring 2023

Lecture: Neural Networks

Date: March 28, 2023 *Author: Eric Wong*

Acknowledgements. These notes are heavily inspired by lectures 3-5 of the Deep Learning Systems Course by Zico Kolter and Tianqi Chen from Carnegie Mellon University.

Disclaimer. These notes have not been subjected to the usual scrutiny reserved for formal publications. If you notice any typos or errors, please reach out to the author.

1 Neural Networks

Up until this point, we've been studying ways to solve empirical risk minimization (ERM) on a variety of machine learning tasks. For example, in the classification setting, we formalized this problem as the following:

Data: Training set $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ where $y_i \in \{-1, 1\}$

Function class: Function class $\mathcal{F}$, the set of all functions over which we aim to minimize the risk.

Goal: Output a classifier f that gets small training error ϵ.

For a loss function ℓ that captures the notion of error, this is the standard empirical risk minimization problem that we've seen before in this course:

$$\min_{f \in \mathcal{F}} \frac{1}{m} \sum_{i=1}^{m} \ell(f(x_i), y_i) = \min_{f \in \mathcal{F}} \hat{R}(f)$$

where $\hat{R}(f)$ is the empirical risk of f. The two central questions behind neural networks is to ask the following two questions:

1. What is the function class $\mathcal{F}$? This is often referred to as the *architecture* of the neural network.
2. How do we learn f? That is, given a fixed architecture, how do we train the model parameters to minimize the risk?

Over-parameterization. Up this point, we've discussed parametric models with a fixed number of parameters (e.g. linear models) and non-parametric models with a variable number of parameters (e.g. decision trees). We referred to parametric models as having high bias and low variance, and non-parametric models as having low bias and high variance. Neural networks are a parametric family of model in that they have a fixed number of parameters, with one key difference: instead of having a small number of parameters (and thus low bias, high variance), neural networks have a large number of parameters. This is called *over*-parameterization, and results in neural networks having low bias and high variance similar to non-parametric models.

Bias-variance trade-off. There is a belief (with some supporting empirical results) that neural networks generalize in a fundamentally different way than traditional machine learning models. Typically, we see that as model complexity increases, test error decreases initially but eventually increases when the model is too complex and *overfits* and memorizes the training data. However, it is argued that neural networks generalize past the point of interpolation, resulting in a different generalization trend—larger models often result in lower test error, without seeming to overfit. This behavior is known as double descent (i.e. Figure 1 from https://www.pnas.org/doi/10.1073/pnas.1903070116).

1.1 Feed-Forward Networks

In these notes, we'll consider a general formulation of neural network architectures known as feed-forward networks. These networks are named after their composition structure: inputs are passed "forward" through a string of computatios. Specifically, a feed-forward network is represented as a sequence of k operations:

$$f_\theta(x) = (f_k \circ \cdots \circ f_1)(x) \tag{1}$$

where $\theta = (\theta_1, \ldots, \theta_k)$ are the parameters of each operation, i.e. θ_i is the parameters for f_i. Then, all that remains to instantiate the functions f_i with some function. In principle, these functions could be anything, however there are certain choices that the community has converged upon for general usage. These are typically categorized into two main groups: core building blocks and activation functions.

Remark: Other presentations of feed-forward networks will present it as an architecture with alternating linear and non-linear functions. For our purposes, we will abstract this away and just consider a sequence of arbitrary functions, which will work nicely with the computational graphs for auto-differentiation down the road.

1.1.1 Core building blocks

The core building blocks are the main workhorse of the neural network, and contain the majority of the parameters. Several classic and recently popular examples include:

- Linear: $f(x) = Wx + b$, where $\theta = (W, b)$. These linear models that we studied early on in the course, can be used as a building block in neural networks.
- Convolutions: $f(x) = w * x$ where $*$ is the convolve operator. This can be defined for 1D, 2D, or even 3D inputs. A 1D convolution is defined as $f(x)_i = \sum_k w_k \cdot x_{i-k}$, while a 2D convolution is defined as $f(x)_{ij} = \sum_{kl} w_{kl} x_{i-k,j-l}$.
- Self attention: $f(x) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V$ where $(Q, K, V) = (XW_Q, XW_K, XW_V)$

1.1.2 Activation functions

Another group of building blocks are known as activation functions. Historically, when the core building blocks were all linear functions, activations were a way to inject non-linearities into the function class. These functions perform a non-linear function element-wise to the input.

- Sigmoid: $f(x)_i = \sigma(x_i) = \frac{1}{1+e^{-x_i}}$. The sigmoid function that we've seen before is one of the most classic activation functions, though it is not commonly used today.
- ReLU (Rectified Linear Unit): $f(x)_i = \max(0, x_i)$. This is a simple function that sets all negative values to zero, and is widely used in today's architectures.
- Sign: $f(x)_i = \text{sign}(x_i)$. The sign function is not useful in practice (it's gradient is always zero), but the set of neural networks with sign activations is a simple enough function class that can be theoretically analyzed.
- Normalization: $f(x)_i = (x_i - \mu)\sigma$. This activation estimates the mean and variance of inputs over the whole dataset and normalizes the inputs. This keeps the values in the network centered around zero without too much spread.

1.1.3 Layers

Common compositions of these building blocks and activation functions are often grouped together in what is sometimes referred to as a layer. Here are some (simplified) layers that you'll see commonly used today or in the past:

- Traditional layer: One linear or convolutional layer followed by an activation function, i.e. $f(x) = \max(0, Wx + b)$.
- Residual layer: $f(x) = x + g(x)$ where $g(x)$ is another function. g could be another layer or possibly another neural network, such as $g(x) = \text{Conv}(\text{ReLU}(\text{Normalize}(\text{Conv}(x))))$. This would a residual layer with 2 convolutionals.
- Multihead attention: $f(x) = \text{Linear}(\text{Concat}(\text{SA}_1(x_1), \ldots, \text{SA}_k(x_k)))$ where SA_i is a self attention block applied to a subset of x.

1.1.4 Architectures

Common combinations of layers create even larger classes of functions referred to as architectures. Here are several common ones:

- Residual Networks (ResNet): A convolutional layer followed by a series of residual layers with k convolutional or linear sub-blocks. k can range anywhere from 18 to 152.
- Transformers: A combination of (1) residual layers with multihead self attention subblocks and (2) residual layers with two linear sub-blocks, where (1) and (2) are repeated k times. k can range anywhere from 2 to 8.

1.2 Neural network theory

Most of the early theory around neural networks focus on the class of neural networks with 2 linear layers separated by a sign activation function. This setting is sometimes referred two as 1 hidden

layer (where hidden layers are sandwiched between inputs and outputs). Specifically, this is the set of functions f where

$$f_{\text{sign}}(x) = \text{sign}(W_2 \cdot \text{sign}(W_1 x + b_1) + b_2$$

Some sample results include:

1. Representation: $f_{\text{sign}}(x)$ can represent every boolean function $g : \{\pm 1\}^d \to \{\pm 1\}$.
2. VC dimension: If $w_1 \in \mathbb{R}^{k \times d}$, $b_1, w_2 \in \mathbb{R}^k$ and $b_2 \in \mathbb{R}$, then the VC dimension of f_{sign} is $O(kd \log kd)$.
3. Learning: Solving $\min_\theta \hat{R}(f_{\text{sign}})$ is NP-hard.

For proofs, see the reading. We only covered the first one in class. In general, similar proofs for networks with more than one hiden layer, other building blocks and activation functions, and non-Boolean target functions are hard. This is a theoretically challenging space!

2 Training Neural Networks

Despite its theoretical challenges, we are going to try to solve the ERM problem regardless. As it turns out, with enough compute power and data, we can make substantial progress.

2.1 Stochastic Gradient Descent

ERM for neural networks is an unconstrained optimization problem. So, a natural approach is to use our gradient descent approaches that we learned earlier in this course. Recall that the gradient descent algorithm repeated the following iteration:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{m} \sum_i \nabla_\theta \ell(f_\theta(x_i), y_i)$$

where α was a step size hyperparameter. The challenge with deep learning is that, because the architectures and datasets are so large, calculating this gradient is extremely expensive. With a large number of parameters and examples, the memory requirements to even store such a gradient can easily exceed any normal computer or GPU. This has led to the adoptation of more memory efficient variations of gradient descent, such as *stochastic gradient descent* (SGD):

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{1}{|B|} \sum_{i \in B} \nabla_\theta \ell(f_\theta(x_i), y_i)$$

where at every time step t, we sample a random minibatch of example indices B where $|B| << m$. This allows us to compute the gradient with respect to only a very small subset of the data as opposed to the entire dataset.

Although memory efficient, SGD has substantially worse theoretical properties than classic gradient descent. While it maintains the $O(\frac{1}{\sqrt{t}})$ convergence rate for convex functions, this rate does not improve with Lipschitz functions. Even worse, strongly convex functions only achieve $O(\frac{1}{t})$ convergence rate as opposed to exponential. Nonetheless, as our only option, this will be our tool for training neural networks.

2.2 Automatic Differentiation

As the SGD update is fairly simple, the only challenge is how to compute the gradient. However, neural networks can be quite complex and be composed of hundreds of functions. Calculating these gradients by hand is an onerous task. There are, however, tools such as PyTorch and Tensorflow that can calculate these gradients automatically for you, also known as automatic-differentiation. These frameworks rely on repeated application of the chain rule on the computational graph, where the precise ordering of the computations determines what type of automatic-differentiation you get.

2.2.1 Chain Rule

We'll exclusively use the chainrule to derive gradients for neural networks. The simple, univariate chain rule is

$$\frac{\partial f(g(\theta))}{\partial \theta} = \frac{\partial f(g(\theta))}{\partial g(\theta)} \cdot \frac{\partial g(\theta)}{\partial \theta}$$

while the general multivariate form is

$$\frac{\partial f(g(\theta))}{\partial \theta} = \sum_{i=1}^{k} \frac{\partial f(g_1(\theta), \ldots, g_k(\theta))}{\partial g_i(\theta)} \cdot \frac{\partial g_i(\theta)}{\partial \theta}$$

A classic algorithm for efficiently computing the chain rule for feed-forward networks consisting of alternating linear and sigmoid activations with a single forward and backward pass is sometimes referred to as the backpropagation algorithm. If you are interested, see https://dlsyscourse.org/slides/manual_neural_nets.pdf for an example of how to derive this manually, and the original backpropagation algorithm (mainly for pedagogical reasons, as we are not expecting you to derive this manually).

2.2.2 Computational graphs

We'll derive the automatic differentiation rules for general computation graphs. These are graphs that represent the sequence of operations computed in the neural network. For a feedforward neural network, it is easy to represent this as a straight line of operations f_1 through f_k with leaves for parameters θ_i that connect to f_i respectively. For an example of a computational graph of a simple function, and an example of these algorithms in practice, see https://dlsyscourse.org/slides/4-automatic-differentiation.pdf.

The forward/reverse-mode auto-differentiation frameworks are much simpler, modular, and extendable than the original backpropagation algorithm. Reverse-mode auto-differentiation is also how nearly all deep learning frameworks are built in practice.

Forward-mode auto-differentiation. For forward mode, we define $\dot{v}_i = \frac{\partial v_i}{x_1}$. We can then compute $\dot{v}_i$ iteratively in the forward topological order of the computational graph. It is called forward because we start from an input x_1, and then propagate gradients forward until we get to the gradient of the output with respect to the input, $\frac{\partial y}{\partial x_1}$.

Note that if we had multiple outputs $y_1, \ldots, y_k$ then this algorithm would compute all the gradients $\frac{\partial y_i}{\partial x_1}$ for all i in a single pass through the network, but only with respect to a specific input x_1. If

we had multiple inputs, we would need to re-run forward-mode auto-differentiation multiple times to get gradients with respect to each input.

Reverse-mode auto-differentiation. For backward mode, we define $\bar{v}_i = \frac{\partial y}{\partial v_i}$. We can then compute $\bar{v}_i$ iteratively in the reverse topological order of the computational graph. It is called reverse because we start from an output y and then propagate gradients backward until we get the gradient of the output with respect to the input, $\frac{\partial y}{\partial x_1}$.

Note that if we had multiple inputs $x_1, \ldots, x_k$ then this algorithm would compute all the gradients $\frac{\partial y}{\partial x_i}$ for all i in a single pass through the network, but only with respect to a specific output y. If we had multiple outputs, we would need to re-run reverse-mode auto-differentiation multiple times to get gradients with respect to each output.

In deep learning, we usually have computational graphs with only one output (the final loss) and a lot of inputs (every parameter in the network). So reverse-mode auto-differentiation is by far the most efficient and is what is used in practice.