Deep Learning Primer

Sami Abu-El-Haija

Spring 2019

Note: This is work-in-progress. It is published early so that students can have a look before it is complete, as completing it might take some time

1 Introduction

This document is written to aid students understand Deep Learning, especially as it pertains to Natural Language Processing (NLP) for the course CSCI 544. This primer gives more breadth than depth. Each section can become its own book, but our goal is to give a complete picture rather than fill all the details. In addition, this primer is not meant to replace the TensorFlow coding lectures.

In general, Deep Learning (DL) is a sub-field of Machine Learning (ML) which is a sub-field of Artificial Intelligence (AI). At a high level, AI is concerned in making intelligent decisions especially on daughnting tasks e.g. classify thousands or millions of articles or images. ML "*automates*" this process and roughly-speaking, frees the algorithm designer from manually customizing the intelligence of the system. In ML, the system is automatically learned from training data. Finally, DL is when the ML model is architectued as an artificial neural network with multiple layers.

Suppose your friend does Machine Learning for her job. If you are curious about her job, then I would expect you to collect **four core ML ingredients** to satisfy your curiosity:

1. **Data:** What dataset do you train your Machine Learning model on? Where do you get the data from?
2. **Model:** What is your model? How do you model the transformation process from the input (e.g. photo of someone's face) to predicted output (e.g. name the person in the photo)?
3. **Objective Function:** What is the objective that you tying to maximize (or minimize)?
4. **Training Algorithm:** How do you train your model? Specifically, what is the procedure for searching for the model that maximizes/minimizes the value of the objective function on the training data?

1.1 Notation

We will use the following notation throughout:

- We denote a dataset $\mathcal{D}$ of n items $\mathcal{D} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}\}$ where $\mathbf{x}^{(i)}$ is the ith training example, and its label (if available) is denoted $y^{(i)}$. It is also common to denote the dataset explicitly

with labels e.g. $\mathcal{D} = \left\{(\mathbf{x}^{(i)}, y^{(i)})\right\}_{i=1}^{n}$. In general, we use lower-case boldface to indicate a vector, lower-case non-boldface to indicate a scalar, and upper-case to indicate a matrix.

- Upper-case X usually denotes a batch of training examples. In extreme cases (known as Batch Gradient Descent), $X \in \mathbb{R}^{n \times m}$ corresponds to the "*design matrix*" with row i containing M-dimensional features for example $\mathbf{x}^{(i)} \in \mathbb{R}^m$.

- Super-script (i) in $\mathbf{x}^{(i)}$ indicates the example ID. In addition, if $\mathbf{x}$ is a sequence with length T, then we denote $\mathbf{x}_j$ to indicate the features of the term $j \in \{1, \ldots, T\}$.

- The model is denoted h and we write $h(\mathbf{x}; \theta)$ to indicate the output of the model operating on example $\mathbf{x}$ using parameter values θ. The parameters are made explicit because they are placed after the semicolon. This is consistent with notation of Andrew Ng. It is also common to write $h_\theta : \mathcal{X} \to \mathcal{Y}$, as in, the function h (with parameters θ) transforms from the input space to the output space. We do not use the last form but it is worth noting.

- We denote the loss as $\mathcal{L} \in \mathbb{R}_+$. Our objective functions are defined as minimizing the loss $\mathcal{L}$. Loss means "error": minimizing the error is equivalent to maximizing accuracy on the training data. Loss is usually defined as sum of losses: over all training examples, and optionally (though very likely) a over the parameters θ for purpose of model regularization.

$$\begin{aligned} \mathcal{L} &= \mathcal{L}_{\text{classification}} &&+ \mathcal{L}_{\text{regularization}} \\ &= \mathcal{L}_{\text{c}} &&+ \mathcal{L}_{\text{reg}} \end{aligned} \tag{1}$$

With

$$\mathcal{L}_{\text{c}} = \sum_i L_i = \sum_i L(y^{(i)}, h(\mathbf{x}^{(i)}; \theta)), \tag{2}$$

$$\mathcal{L}_{\text{reg}} = \lambda \sum_{W \in \theta} R(W), \tag{3}$$

where L measures the loss on an individual example: we will see some examples soon.

1.2 Training Algorithm

The training algorithm refers to the process of finding the parameters θ that give the least possible error. Formally, the purpose of the training algorithm is to do the minimization:

$$\min_\theta \mathcal{L} = \min_\theta \mathcal{L}_{\text{c}} + \mathcal{L}_{\text{reg}} = \min_\theta \sum_i L_i + \mathcal{L}_{\text{reg}}. \tag{4}$$

"Optimization" is very active area of research that spans multiple disciplines (e.g. Computer Science, Mathematics, Electrical Engineering, Industrial Engineering, just to name a few). Nonetheless, we will take the optimization algorithm as a "blackbox". We will only discuss the most popular optimization algorithm: Gradient Descent.

1.2.1 Gradient Descent

The Gradient descent operates in epochs. Each epoch is an iteration over the training data. It can be described with the following pseudo-code:

1. Initialize θ to Random.
2. Repeat:
   (a) Sample $X \subset \mathcal{D}$
   (b) Compute Gradient of loss w.r.t. paremeters θ using sample X: $G \leftarrow \frac{\partial \mathcal{L}}{\partial \theta}\big|_{X,\theta}$
   (c) Update: $\theta \leftarrow \theta - \eta G$, where $\eta \in \mathbb{R}_+$ is known as "step-size" or "learning rate".

1.2.2 Stochastic Versus Full-batch Gradient Descent

In step 2.(a) of the above Gradient Descent Psuedo-code, we repeatedly sample matrix X from $\mathcal{D}$. In one extreme version, X contains all n (m-dimensional) examples from $\mathcal{D}$ i.e. $X \in \mathbb{R}^{n \times m}$. In the other extreme version, X contains only a single example i.e. $X = \mathbf{x}^{(i)\top}$. The first version is known as full-batch gradient descent (or simply, gradient descent), and the second version is known as Stochastic Gradient Descent (SGD). If the right step-size η is chosen, both algorithms will provably converge to the global optimum if the $\mathcal{L}$ is convex and to a local optimum if $\mathcal{L}$ is non-convex.

Between these two versions lie the most popular alternative: mini-batch gradient descent, where every time X is sampled, it is chosen to be a number of b examples (e.g. b =16 to 500 examples). Here, $1 < b \ll n$.

Studying convergence is outside the scope of this primer, but you should know that if $\mathcal{L}$ is non-convex, then full-batch GD is worse than SGD in terms of finding a more optimal local minima. The rough explanation is as follows: Every example defines a different loss surface (in terms of θ) and very likely the minimas will on these surfaces will be different than the minimas on the average surface. The average surface is the one optimized by GD. In essence, SGD will allow the optimization algorithm to escape these local minimas defined by the average loss surface.

1.3 Linear Models

Before we present "Deep Learning", we review a "shallow model" that is used in a ton of places (including perceptron, logistic regression, linear regression, SVMs, and others). The linear model is defined as:

$$h(\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{x}^\top \mathbf{w} + b), \tag{5}$$

Where the model parameters $\theta = \{\mathbf{w}, b\}$ are found to maximize accuracy (minimize loss), and σ is an activation function i.e. $\sigma : \mathbb{R} \to \mathbb{R}$. You can recover Perceptron if you use σ to be the sign function, recover logistic regression if σ is the sigmoid logistic, and recover SVM if σ was identity i.e. $\sigma(z) = z$.

Throughout the writing, we will interchange the input of h to operate on single example i.e. $h(\mathbf{x})$ or group of examples $h(X)$. When applying on a group of examples, we will write the multiplication as XW as to indicate that every row of X contains an row-vector example $\mathbf{x}^\top$.

1.4 Loss Functions

The most common loss function to use in classification is cross entropy. It is defined as:

$$\mathcal{L}_{\text{c}} = \sum_{i=1}^{n} -y^{(i)} \log h(\mathbf{x}^{(i)}; \theta) - (1 - y^{(i)}) \log(1 - h(\mathbf{x}^{(i)}; \theta)). \tag{6}$$

We will later explain as to where this loss came from. But for now, you should have a pictorial understanding of why this function make sense. First of all, this cross entropy function will only work if both h and y values are between 0 and 1. In fact, for single-class classification, almost always: $y^{(i)} \in \{0, 1\}$ and the (final) activation of h is sigmoid. For multi-class classification (many labels can be "on" or "of") then $y^{(i)} \in \{0, 1\}^c$, where c is the number of classes. In which case, $h : \mathbb{R}^m \to (0, 1)^c$, the log function is applied element-wise, and the multiplication of $y^{(i)}$ with the output of log is dot-product or hadamard product. Let us focus our pictorial understanding to the case that $c = 1$ and if $c > 1$, the argument below applies independently to each class parameters. Specifically, only one of the terms is active for some example i: One of $y^{(i)}$ and $1 - y^{(i)}$ is 1 and the other is 0. Let's assume a positive example i.e. with $y^{(i)} = 1$. For that example, we are minimizing $-\log h(\mathbf{x}^{(i)}; \theta)$. We plot $-\log(z)$ below:

TODO Plot

For positive labeled-examples, we want our model to give a score closer to 1. The way to minimize the error (i.e. $-log(h(\mathbf{x}^{(i)}; \theta))$) is to find the θ that makes h outputs a number as-close-as-possible to 1. You should think on your own, what happens when the example is negative i.e. when the term $(1 - y^{(i)})$ is the active term.

There are many other loss functions that we won't discuss in this primer. One notable one is the one used for regression with $L_i = \frac{1}{2}(y^{(i)} - h(\mathbf{x}^{(i)}; \theta))^2$, which minimizes the squared difference between the model output and the label. This minimization is also known as SSE (sum-of-squared errors), and is equivalent to the minimizations MSE (mean-squared-error) and RMSE (root-mean-squared-error).

1.5 Regularization

The coefficient $\lambda \in \mathbb{R}_+$ in Equation 2 indicates "how much regularization to apply". Intuitively, low numbers imply higher degrees of freedom for the parameters $W \in \theta$. The regularization function R is a statistical measure. One frequently used R, is the squared-Frobenius norm (equivalent to squared L2-norm, if W was a vector $\mathbf{w}$). For all practical purposes, and theoritically provable: If $\mathcal{L}_c$ is convex, then jointly optimizing $\mathcal{L}_{\text{reg}}$ **will always hurt** $\mathcal{L}_c$. In addition, smaller (i.e. better) values of $\mathcal{L}_c$ implies high classification accuracy. It might seem counter-intuitive: why jointly optimize $\mathcal{L}_{\text{reg}}$ if it is guaranteed to give us worse classification accuracy due to higher training loss $\mathcal{L}_c$? The answer is: Regularization helps generalization! The goal of Machine Learning algorithms is **not** to

do a perfect job on training examples. The goal is to do as good as possible, on data points that were never seen during training (i.e. test data). Unless you have a strong reason not to, you should always regularize multiplicative[1] parameters as:

$$\mathcal{L}_{\text{reg}} = \lambda ||\theta||_F^2 \tag{7}$$

A usual choice of λ is usually a small value e.g. 0.000001 to 0.001. The mathematical justification is outside the scope of this primer[2]. However, you can reason about the effect of regularization: it encourages individual values of parameters to be small. Small values are nice, because they make our function h more smooth (or "less abrupt"), in the sense that, if you slightly change the its input $\mathbf{x}$, then you would expect a small change in its output. To summarize:

- Regularization encourages smaller parameter values.
- Regularization always makes the accuracy worse on training data examples, but it makes it better for unseen test data examples.
- Regularization is not part of the model, but is part of the training process. Specifically, if someone gives you a trained model h with its optimized parameters θ, then you can very easily you can compute the classification loss on the data $\mathcal{L}_{\text{c}}$, but in no-way you would be able to calculate $\mathcal{L}_{\text{reg}}$, unless she told you exacly how she did it (e.g. specified the value of λ)

1.6 Putting it altogether

Lets fully write-out the optimization of Logistic Regression: a regularized-linear model on a dataset can be trained by the following minimization:

$$\min_{\theta} \mathcal{L}_{\text{reg}} + \mathcal{L}_{\text{c}} \tag{8}$$

$$= \min_{\theta} \lambda ||\mathbf{w}||_2^2 + \sum_{i=1}^{n} -y^{(i)} \log \left(\frac{1}{1 + e^{-(\mathbf{x}^{(i)\top} \mathbf{w} + b)}} \right) - (1 - y^{(i)}) \log \left(1 - \frac{1}{1 + e^{-(\mathbf{x}^{(i)\top} \mathbf{w} + b)}} \right). \tag{9}$$

The fraction inside the log is the equation for the standard sigmoid logistic function. The parameters here are $\theta = \{\mathbf{w}, b\}$. It is important to notice that the bias term should not be regularized (though some implementations regularize them).

Everything after this point is incomplete

[1] for very good reasons, you should not regularize your bias terms, if possible. Only regularize values that get "multiplied" rather than added

[2] Nonetheless, this form can be derived by linear regression when assuming that training data-points are generated from a distribution containing gaussian noise centered around 0.

2 How to create x

Mathtematically speaking, $\mathbf{x}$ represents numerical values, but all we have are documents and their corresponding classes.

3 Multi-Layer Perceptron

Going from one layer (e.g. as in perceptron, or logistic regression above) to multiple layers neural network, is simply acheived by applying the transformation multiple times. For example, one-layer neural network can be defined by: $h_{\text{(1-layer)}}(\mathbf{x}; W, \mathbf{b}) = \sigma(\mathbf{x}^\top W + \mathbf{b})$. Let us make the dimensions explicit. If the number of classes is c and the dimensionality of $\mathbf{x}$ is m. Therefore, we would like the input-output dimensions of h to be such that $h : \mathbb{R}^m \to \mathbb{R}^n$. In addition, if we want to train this model using cross-entropy loss (Equation 6), then range of σ must be between 0 and 1 and therefore $h : \mathbb{R}^m \to (0, 1)^n$. Dimension-wise, W must have m rows and c columns i.e. $W \in \mathbb{R}^{m \times c}$ and also $\mathbf{b} \in \mathbb{R}^c$. In contrast, a three layer neural network can be written as:

$$h_{\text{(3-layers)}}(\mathbf{x}; \theta) = \sigma(\sigma(\sigma(\mathbf{x}^\top W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2)W_3 + b_3) \tag{10}$$

with $\theta = \{W_1, \mathbf{b}_1, W_2, \mathbf{b}_2, W_3, \mathbf{b}_3\}$

3.1 Representational Capacity

4 Recurrent Neural Networks (RNNs)