Reading guide for Eisenstein (2019) ch. 3 and Goldberg (2017) ch. 3–5

LING83600

1 Introduction

Eisenstein (2019: 47) observes that simple linear classifiers are shockingly effective for many problems—possibly because words are inherently meaningful even in isolation, making them powerful features on their own, and there are often more unique words than we know what to do with—but that non-linear methods (of which neural networks are the prime example) are increasingly successful. This is thanks to

- advances in deep learning,
- their ability to leverage embeddings, and
- their ability to leverage custom hardware in the form of graphics processing units (GPUs).

Improvements in optimization, and corporate-fueled momentum, play an important role in the ongoing neural network revolution.

Goldberg (2017: 37f.) notes that the "hypothesis class of linear (and log-linear) models is severely restricted", in the sense that many functions one may want to predict are non-linear. One famous example, due to Marvin Minsky, is the so-called xor ("exclusive or") function:

$$\begin{aligned}
\text{xor}(0,0) &= 0\\
\text{xor}(1,0) &= 1\\
\text{xor}(0,1) &= 1\\
\text{xor}(1,1) &= 0
\end{aligned}$$

For this, there is no set of linear weights $\mathbb{R}^2$ and bias $b \in \mathbb{R}$ such that the score of $(0,0)$ and $(1,1)$ are ≤ 0 but the scores of $(1,0)$ and $(0,1)$ are ≥ 0. Alternatively, one can plot the four points and confirm that one cannot separate the 0s and 1s with a line. However, it is possible to transform the inputs so that the data is *linearly separable*. Goldberg suggests the transformation $\varphi(x_1, x_2) = [x_1 x_2, x_1 + x_2]$; this is illustrated in Figure 1.

Kernel methods, often used with support vector machines, are a well-studied approach for non-linear classification in which the experimenter provides a non-linear transformation function—e.g., the polynomial mapping $\varphi(x) = (x)^d$ where $d \in \mathbb{N}$ is as hyperparameter—which is applied using the *kernel trick*.

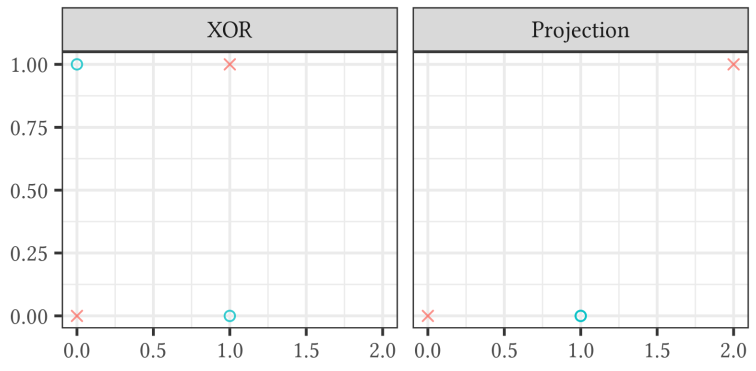

Figure 1: The XOR problem, and a non-linear projection (after Goldberg: 37-38).

2 Motivations

Neural network classifiers offer an alternative in which the non-linearity is itself trainable. Eisenstein motivates this imagining that we have a set of raw observations x, features extracted from those observations z, and labels $y \in \mathcal{Y}$. Given this, one could construct a "two-step" classifier as follows:

1. Use one logistic classifier to predict z according to $p(z_k \mid x)$.
2. Use a second logistic classifier to predict y according to $p(y \mid z)$.

If we assume that each feature is binary, the first step can be written

$$p(z_k = 1 \mid x; \theta^{(x \to z)}) = \sigma(\theta_k^{(x \to z)} \cdot x)$$

where σ is the *sigmoid function*

$$\sigma(x) = \frac{1}{\exp(-x)}.$$

We form $\theta^{(x \to z)}$ by concatenating (i.e., stack) the weight vectors for each z_k to form Given a set of labels $\mathcal{Y}$, the probability for a given label $y \in \mathcal{Y}$ is given by

$$p(y = j \mid z; \theta^{(z \to y)}, b) = \frac{\exp(\theta_j^{(z \to y)} \cdot z + b_j)}{\sum_{j' \in \mathcal{Y}} \exp(\theta_{j'}^{(z \to y)} \cdot z + b_{j'})} \tag{1}$$

where b_j is a bias term for label $j \in \mathcal{Y}$, and $\theta^{(z \to y)}$ is once again formed by concatenation. The vector of probabilities over each possible value of y, the *softmax*, is denoted by

$$p(y \mid z; \theta^{(z \to y)}, b) = \mathrm{softmax}(\theta^{(z \to y)} z + b)$$

in which each element $j \in \mathcal{Y}$ is computed as per eq. 1.

Now instead suppose that we never observe z, and instead of using predicted values of z, we feed the second layer the probabilities $\sigma(\theta_k \cdot \mathbf{x})$. We can write the resulting model as

$$z = \sigma(\theta^{(x \to z)} x)$$
$$p(y \mid x; \theta^{(z \to y)}, b) = \text{softmax}(\theta^{(z \to y)} z + b)$$

This model is known as a single-layer *feedforward neural network*, or somewhat inaccurately, as a *multilayer perceptron.*

3 Learning

Neural networks' parameters are usually trained to minimize the *cross-entropy*, the negative conditional log-likelihood (Eisenstein: 52-53). Software packages like PyTorch or TensorFlow use an abstraction known as the *computation graph* (Goldberg: 51f.) to automatically compute and cache the derivatives, and to *backpropagate* gradients to earlier components in the network.

4 Learning theory

It is known (Goldberg: 44-45) that single-layer MLPs with a non-linear activation function are a *universal approximator*, meaning that it can approximate any continuous real-valued function, and any discrete function, so long as they are bounded. This may suggest that we don't need multiple hidden layers; however, some caution is in order. The above theorems have little to say about how large the hidden layer has to be, and in fact there exist functions such that two small hidden layers can approximate an arbitrary large single layer.

Whereas the linear models we reviewed last week are all convex, neural networks are generally non-convex and may have many local optima. However, empirical studies suggest that neural network critical points—points at which the gradient is zero, halting gradient-based learning—are often *saddle points* rather than true local minima (Eisenstein: 58). However, there is no general guarantee of convergence nor any bounds on error. It may be the case that the assiduous use of multiple hidden layers also speeds convergence.

5 Tricks

"Tricks" used to train neural networks include choice of non-linearity, clever random initializations, gradient clipping, custom optimization routines, learning rate scheduling, and novel forms of regularization. All of these are supported by common neural network software.

5.1 Non-linearities

Early work in neural networks primarily used the sigmoid function σ as a non-linearity (or *activation function*), as we did above. Modern work tends to use the *hyperbolic tangent function* $\tanh$, which has the range $(-1, +1)$. Another commonly used non-linearity is the *rectified linear unit* (ReLU) given by

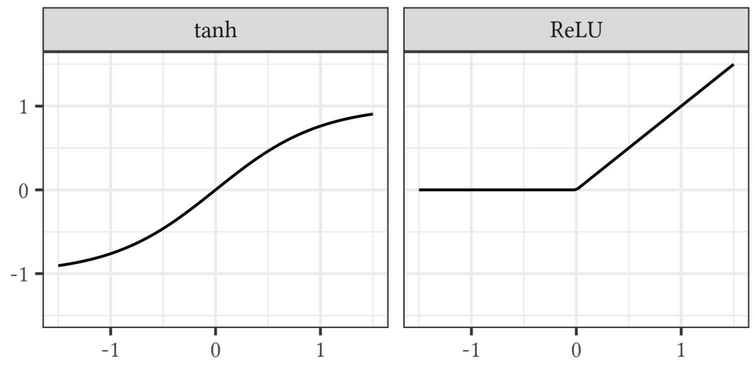

Figure 2: Two commonly-used non-linearities: $\tanh$ and ReLU (after Eisenstein, figure 4.3).

$$\text{ReLU}(a) = \begin{cases} a \text{ where } a \geq 0 \\ 0 \text{ otherwise} \end{cases} .$$

These non-linearities are visualized in Figure 2.

5.2 Initialization

Weights in a neural network model are often initialized randomly. Ideally, one should use several random initializations. *Xavier initialization*, named for one of its creators, is claimed to be optimal with a $\tanh$ nonlinearity; alternatives exist for other nonlinearities (Eisenstein: 59, Goldberg: 59).

The initialization of embeddings is somewhat unique. One can either initialize them by sampling from a random uniform distribution, or initialize them from external embeddings created with tools like Word2Vec or GloVe. In both cases, one may train them as part of network training; in the case the embeddings are initialized from some external tool, this is known as *fine-tuning*.

5.3 Gradient clipping

The problem of exploding gradients is a poorly-understood hazard in neural network training. To avoid this, a simple but effective solution (Eisenstein: 60, Goldberg: 60) is to "clip" gradients (i.e., bound) whose L_2 norm exceeds a certain threshold.

5.4 Optimization

Neural networks are often trained using minibatches of examples (Goldberg: 61); sometimes minibatch size is bounded solely by the GPU. Today, few neural networks are trained using vanilla minibatch stochastic gradient descent; rather, they use *adaptive* variants which use separate learning rates for each parameter, such as AdaGrad (Eisenstein: 38f.) or Adam.

5.5 Learning rate scheduling

It is often desirable to automatically decrease the learning rate according to some fixed schedule as the model approaches convergence (Goldberg: 61). It may also be desirable to *ramp up* learning rate at the start of training.

5.6 Dropout

Traditional regularization can be applied to neural networks (Eisenstein: 56f.); in this context they are usually referred to as *weight decay*. A more-common method is the use of *dropout*, in which, with some probability p, we set input and/or hidden unit activations to 0. As Eisenstein writes, this "prevents the network from learning to depend too much on any one feature or hidden node and prevents feature co-adaptation, in which a hidden unit is only useful in combination with one or more other hidden units" (57).

References

Eisenstein, Jacob. 2019. *Introduction to Natural Language Processing*. MIT Press.

Goldberg, Yoav. 2017. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool.