Neural networks

LING83600
Kyle Gorman
Graduate Center, City University of New York

Simple discriminative linear classifiers like *logistic regression* and *support vector machines* have proved—and continue to be—highly effective tools for a wide variety of speech & language processing problems. However, they are increasingly outperformed by *neural networks*, a family of discriminative, **non-linear** classifiers. As Manning (2015) wrote a few years ago, we are living through a "deep learning tsunami".

Outline

- Simple linear classifiers
- Multilayer perceptrons
- Learning
- Tricks
- Neural network hardware and software

Simple linear classifiers

Text classification

Let us suppose that we wish to predict the genre of various text documents. Suppose further that for each document we have a vector of raw token counts x (or TF-IDF counts).

Simple linear text classification

We wish to predict a label $\hat{y} \in \mathcal{Y}$ given x. To do this we use a scoring function $\Psi(x, y)$ such that

$$\Psi(x, y) = \theta \cdot f(x, y)$$

according to weights $\theta \in \mathbb{R}^{VK}$ where $K = |\mathcal{Y}|$ and V is the size of the vocabulary and a feature function f. Inference is performed by selecting the label $\hat{y} \in \mathcal{Y}$ which maximizes $\Psi(x, y)$.

The XOR problem

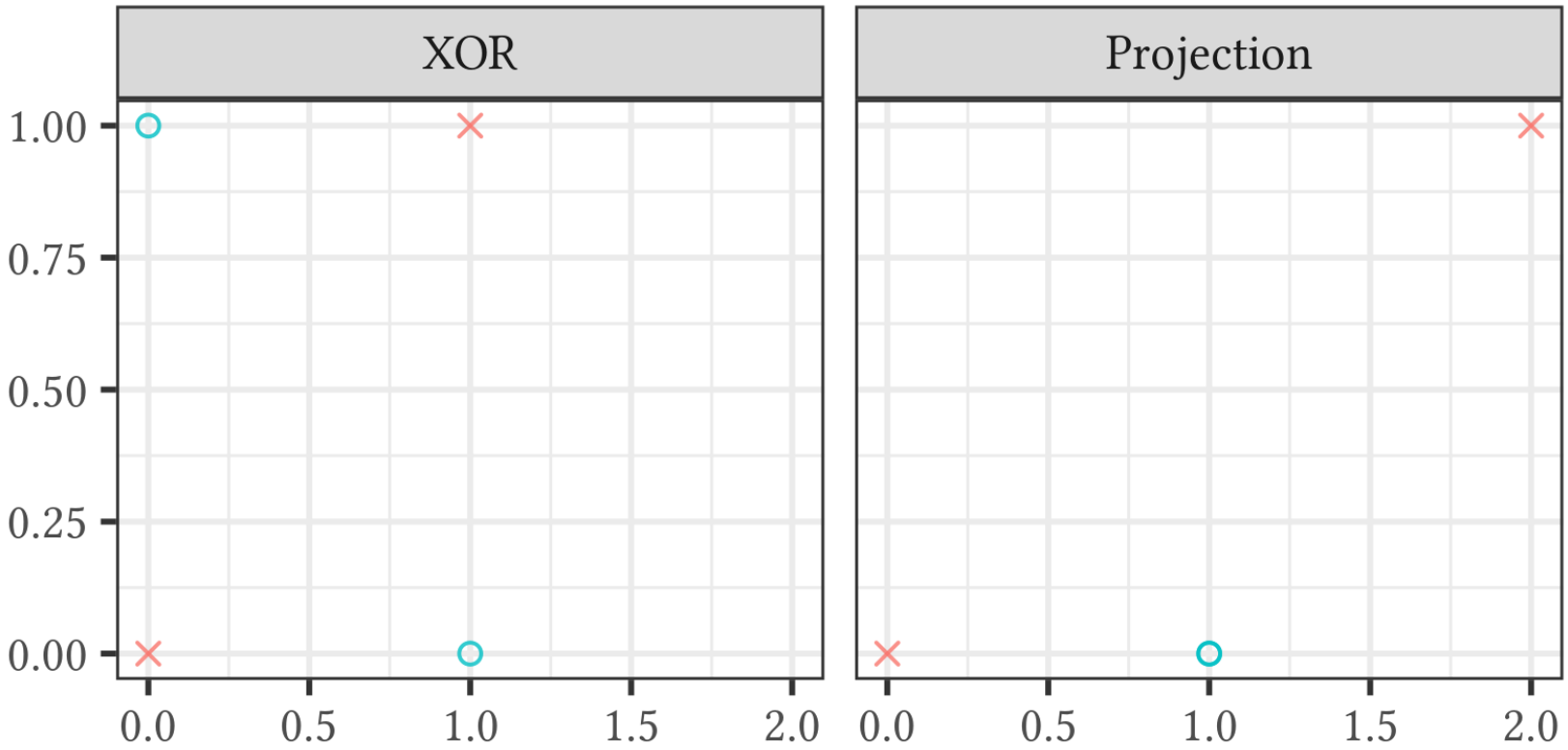
However, the hypothesis class of simple linear models is severely restricted, in the sense that many useful functions cannot be described by a linear model. One well-known example is the xor ("exclusive or") function:

$$\mathrm{xor}(0, 0) = 0$$
$$\mathrm{xor}(1, 0) = 1$$
$$\mathrm{xor}(0, 1) = 1$$
$$\mathrm{xor}(1, 1) = 0$$

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.

Non-linear transformations

However, there may exists non-linear transformations that make the data linearly separable, such as $\phi(x_1, x_2) = [x_1 x_2, x_1 + x_2]$ (Goldberg, p. 17).



Multilayer perceptrons

Introducing the multilayer perceptron

Neural network classifiers are cascades of one or more linear models with a trainable non-linearity. Such models are sometimes known—not very informatively—as *multilayer perceptrons* (MLPs).

Motivations

Imagine that we have raw observations x, features extracted from those observations $z = f(x, y)$, and labels $y \in \mathcal{Y}$. Then, one could construct a "two-step" classifier as follows:

- Use one logistic classifier to predict z according to $p(z_k \mid \boldsymbol{x})$.
- Use a second logistic classifier to predict y according to $p(y \mid \boldsymbol{z})$.

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

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

where σ is the non-linear *sigmoid function*

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

Motivations

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

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

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

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

in which each element $j \in \mathcal{Y}$ is computed as above.

Formalization

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

$$\boldsymbol{z} = \sigma(\boldsymbol{\theta}^{(x \rightarrow z)} \boldsymbol{x})$$

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

Early history

- McCulloch and Pitts (1943) proposed *nerve nets*, a simple model of computation loosely inspired by neurons.
- This inspired Rosenblatt (1958) to propose *perceptrons*, a form of simple linear model.
- Minsky and Papert (1969) noted the XOR problem for simple linear models in their 1969 book *Perceptrons*.
- Various authors noted at the time that a cascade of linear models, one feeding into another, could get around the XOR problem, but lacked an algorithm for learning the weights of such a cascade.
- Rumelhart et al. (1986) popularized the *backpropagation* algorithm for learning these weights.

Learning

Expressivity

According to the *universal approximation theorem*, any continuous function over (a compact subset of) real numbers can be approximated (to arbitrary precision) by an MLP with a single hidden layer and a finite number of neurons. However, this theorem doesn't tell us

- how many neurons we need,
- whether we'd be better off with two (or three, or four) hidden layers,
- how to learn the weights of that network from a finite sample.

Optimization

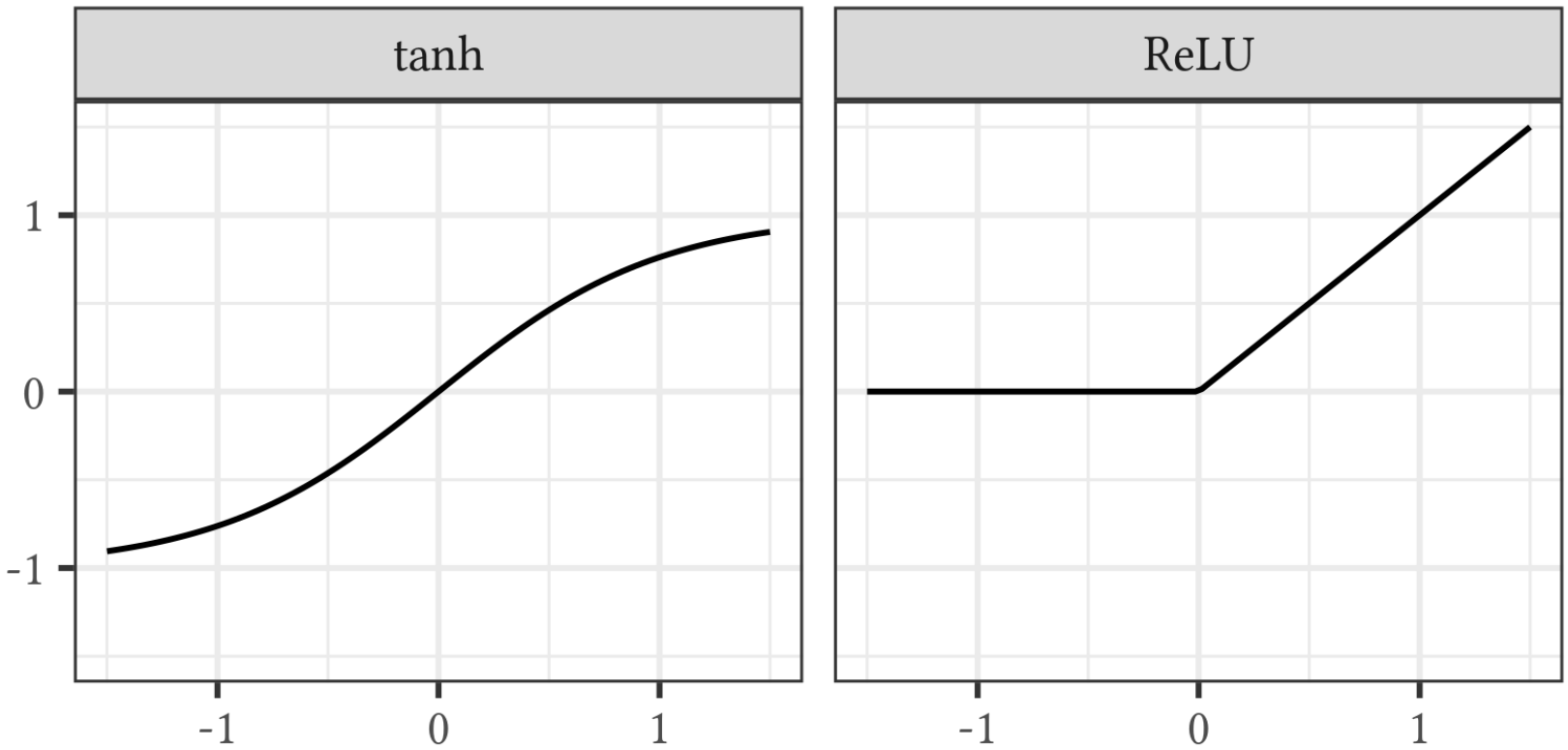
Training simple linear models are *convex* optimization problems, and as such, forms of stochastic gradient descent are guaranteed to converge on an optimal solution under a wide variety of scenarios. In contrast, MLPs are in non-convex and may have multiple *local optima*. There are no general guarantees that neural network training will converge nor are there any bounds on error.

Tricks

Non-linearities

Early work in neural networks mostly used the sigmoid function σ as a non-linearity, whereas modern work tends to use either

- the *hyperbolic tangent function* $\tanh$ or
- the *rectified linear unit* (ReLU).



Initialization

Weights in a neural network are usually initialized by sampling values randomly from some reference distribution. When working with a $\tanh$ non-linearity, for example, *Xavier initialization* (Glorot and Bengio 2010)

$$W_{i,j}^{[l]} = \mathcal{N}(0, \frac{1}{n^{[l-1]}})$$

where l is the layer index and n is the number of neurons in that layer, is claimed to be optimal. Similar alternatives exist for other non-linearities; see here for a demonstration.

Embedding initialization

For networks that involve an embedding layer, there are three possibilities:

- Randomly initialize the embeddings and *tune* them during training.
- Use pre-trained embeddings (e.g., from Word2Vec, GloVe, fastText) but *fine-tune* them during training.
- Use pre-trained embeddings but do not fine-tune.

Gradient clipping

Occasionally training fails to the poorly-understood problem of *exploding gradients*. A simple but effective solution to this problem is to "clip" (i.e., bound) gradients when their overall magnitude (measured using the L2 norm) exceeds a certain hyperparameter threshold $C \in \mathbb{R}_+$.

Optimizers

Neural networks are trained with batched stochastic gradient descent (SGD). Most researchers prefer *adaptive* variants of SGD like Adam (Kingma and Ba 2015), which keep track of separate learning rates for each parameter. Batch size itself is an important hyperparameter, and interacts—often in unpredictable ways—with other hyperparameters like learning rate, regularization coefficients, etc.

Learning rate scheduling

It is usually desirable to gradually reduce the learning rate. Some popular *scheduling* techniques involve decaying the learning rate

- learning rate linearly (i.e., dividing it by the number of updates),
- according to the inverse square root law (i.e., dividing it by the inverse root square of the number of updates), or
- every time development set performance plateaus.

A fast linear *ramp-up* of the learning rate at the start of training from some low rate is also recommended, particularly when training with randomly-initialized embeddings.

Regularization

- *Weight decay* is a form of L2 regularization that can be applied to neural networks; it imposes a penalty on large-magnitude weights.
- An alternative is *dropout* (Srivastava et al. 2014). For each neuron, at each training step, we sample a number $p \in [0, 1]$. If $p < P$ (where P is a hyperparameter in the range $[0, 1]$), we "mask" (i.e., temporarily ignore) the output of said neuron during that step.

Assiduous use of weight decay and/or dropout is essential to prevent neural network overfitting.

Hardware and software

Hardware

The “tsunami” might never have arrived without Nvidia’s CUDA toolkit, which enables power-efficient, massively-parallel floating-point number computations—the type of computation that neural network training and inference requires—to be run on Graphics Processing Units (GPUs) chips originally designed for video gaming.

- The GC CL lab has six Nvidia GTX 1080 GPUs.
- Additional GPU capacity can be obtained from the High Performance Computing Center at the College of Staten Island’s PENZIAS cluster.
- Google sells training/inference time with proprietary ASICs called Tensor Processing Units (TPUs).

Neural networks can also be trained on conventional CPUs, but GPUs are generally more efficient and much faster.

Software

Rather than programming in CUDA itself, developers use Python extension module libraries which dispatch the operations of the computation graph to CUDA. The two best-known libraries are

- Facebook's PyTorch and
- Google's TensorFlow.

Python libraries like AllenNLP, FairSeq, HuggingFace, Keras, NeMo, and PyTorch Lightning provide friendlier, task-specific interfaces to PyTorch and/or TensorFlow.

References I

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

D. P. Kingma and J. Ba. Adam: a method for stochastic optimization. In *3rd International Conference on Learning Representations: Conference Track Proceedings*, 2015.

C. D. Manning. Computational linguistics and deep learning. *Computational Linguistics*, 41(4):701–707, 2015.

W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

M. Minsky and S. Papert. *Perceptrons*. MIT Press, 1969.

References II

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6): 386–408, 1958.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.