Foundations of Mathematics
Part III: Neural Networks

Alexander Koller

January 25, 2022

1 Introduction

We will now put together everything you have learned from Parts I and II and look at the basics of neural networks and backpropagation.

Obviously, this is not a full class on neural networks, in that we are only looking at multilayer perceptrons (MLPs) and not at more complex architectures such as LSTMs; we are not training our own neural networks, and we are not looking at neural learning theory in any kind of detail. I strongly recommend that you take Dietrich Klakow's Neural Networks class and perhaps spend some time in the term break to install the Pytorch library and play around with training your own neural networks.

The reading materials for Part III are lecture notes from Roger Grosse's course CS 421 Neural Networks and Deep Learning at the University of Toronto. If you like, you can read the lecture notes for the rest of his course and try out his programming assignments.

2 Neural Networks **Feb 1**

- 3Blue1Brown: But what is a neural network?
- 3Blue1Brown: Gradient descent, how neural networks learn
- Grosse, Lecture 3: Multilayer Perceptrons
- Grosse, Lecture 7: Optimization

As you know, the use of neural networks has revolutionized many fields of artificial intelligence over the past few years, including natural language processing. Neural networks themselves have been around since the 1960s, but early architectures ("perceptrons") weren't expressive enough for interesting learning problems. The real usefulness of neural networks comes if multiple *layers* of computation can be stacked on top of each other; hence the name "deep learning". Since roughly 2010, the use of GPUs – the processors

of graphics cards – and specialized hardware such as Google's TPUs have made the efficient training of nontrivial neural networks feasible.

While neural networks look quite magical from the outside – they can learn to classify images, track people in videos, translate quite accurately from one language to another, play Go and Starcraft at human world-champion level, and many other things –, they are built from a few rather simple mathematical components. The explanation in this course is split into two sessions. First, we will look at what a neural network is and introduce the *gradient descent* algorithm, which trains a neural network by modifying the weights of the NN in order to minimize a loss function. As the name suggests, this involves computing gradients. Next time, we will look at the *backpropagation algorithm*, which computes the gradient of a loss function efficiently.

Neurons. As explained on page 2 of the Grosse notes, a neural network consists of *neurons*, which transform their inputs $\mathbf{x} = [x_1, \dots, x_n]$ into an output a, its *activation*:

$$a = \phi\left(\sum_j w_j x_j + b\right) \tag{1}$$

The inputs and the activation are arbitrary real numbers; of course these are implemented as floating-point numbers of varying precision on actual hardware. An NN can have multiple *layers*, each of which consists of any number of neurons. The activations of these neurons then serve as input to the next layer, as indicated in (1) on Grosse's p. 3.

Notice that the sum in (1) is simply the dot product between a vector $\mathbf{w} \in \mathbb{R}^n$ of weights and the vector $\mathbf{x} \in \mathbb{R}^n$ of inputs. Thus, one can write (1) more succinctly as

$$a = \phi(\mathbf{w}\mathbf{x} + b) \tag{2}$$

Furthermore, all activations on the same layer are computed in the same way, from the same inputs $\mathbf{x} \in \mathbb{R}^n$. Let's say that we have k neurons in the layer, with weight vectors $\mathbf{w}_1, \dots, \mathbf{w}_k$ and biases $b_1, \dots, b_k$. Then we can collect all the biases into a single column vector $\mathbf{b} = [b_1, \dots, b_k] \in \mathbb{R}^k$. We can further construct a $k \times n$-matrix W by reading each weight vector $\mathbf{w}$ as a row vector and stacking them on top of each other; that is, the i-th row of W is $\mathbf{w}_i^T$. Then we can simply write

$$\mathbf{a} = \phi(W\mathbf{x} + \mathbf{b}). \tag{3}$$

Notice that $\mathbf{a} \in \mathbb{R}^k$, as expected, and we have silently lifted the nonlinearity ϕ from a function $\phi : \mathbb{R} \to \mathbb{R}$ to a function $\phi : \mathbb{R}^k \to \mathbb{R}^k$ by applying it elementwise. When you read papers on neural networks, it is a really good idea to work out such details (and check that the shapes of the matrices and vectors all match) by hand.

Loss functions. In supervised learning, an NN is trained with respect to *labeled data*, which specifies the output that an NN is supposed to produce from its last (= output) layerfor any given input. The labels can be anything you like: In the classification of handwritten numbers as in 3B1B's example, you have the classes '0' to '9'; for a part-of-speech tagger, the classes are the parts of speech; and so on. An untrained, randomly initialized NN will produce junk output for most inputs. The goal of training is to get it to produce the correct output as frequently as possible.

This is done by defining a *loss function*, which indicates how much the predicted outputs of the NN differ from the gold-standard outputs in the labeled data. Grosse defines these as follows:

- A *loss function* $\mathcal{L}(y, \hat{y})$ takes the NN's output y and the gold label $\hat{y}$ as input and returns a real number. You can think of it as a judgment on how close to the truth the NN's prediction is on one individual input.
- A *cost function* $\mathcal{E}$ takes *all weights and biases* of the NN as its input and returns the mean loss of the NN over all training instances. It is a multivariate function that indicates how the mean loss changes when you manipulate the weights.

On page 11,[1] equations 3–6, Grosse shows how the cost function is determined from the losses of the individual training instances. The step from (3) to (4) may be a bit confusing. What he means is that the per-instance cost $\mathcal{E}_n$ is computed by running the NN on the input of the n-th training instance. The NN will produce an output $y^{(n)}$, which the loss function compares against the gold standard $\hat{y}^{(n)}$. Because the way in which the NN does this depends on its weights and biases, we can still see $\mathcal{E}_n$ as a value that was computed by taking those weights and biases as input and compute its gradient with respect to those.

A loss function can be any differentiable function, but there are two that are especially important.

- The *squared error* loss function, $(y - \hat{y})^2$. This loss function is practically useful in regression situations, where $\hat{y}$ is some numeric value and we want the NN's prediction to be as numerically close to it as possible. Its main appeal for pedagogical explanations as in 3B1B and Grosse is the simplicity of its derivative with respect to the NN output y, which is just $2(y - \hat{y})$.
- The *negative log-likelihood* loss function,

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{M} \hat{y}_j \log y_j. \tag{4}$$

  Here we assume that the NN predicts a discrete probability distribution over M possible classes (e.g. M part-of-speech tags), and that y_j is the probability of class

[1] Note that each chapter in the Grosse notes starts its page numbers at one. When I say "page 11", I mean the 11th page of the PDF, which happens to be marked as page 4 of Chapter 7 on the page itself.

j.[2] We also assume that $\hat{\mathbf{y}}$ is a *one-hot vector* encoding of the gold label; so for instance, if the correct output of the current training example is some number $1 \leq i \leq M$, then $\hat{\mathbf{y}} \in \mathbb{R}^M$ is a vector that is zero everywhere, except that it is one at position i.

If we think of $\hat{\mathbf{y}}$ as another probability distribution, the NLL loss captures the *cross-entropy* and the *KL divergence* between the probability distribution $\mathbf{y}$ and the probability distribution $\hat{\mathbf{y}}$.

In the case where $\hat{\mathbf{y}}$ is a one-hot encoding of class i, the negative log-likelihood loss becomes simply

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\log y_i, \tag{5}$$

i.e. minimizing this loss maximizes the log-likelihood of the correct class. This is why this loss is overwhelmingly popular for classification tasks, which constitute at least 90% of the machine learning tasks in NLP. However, it is less convenient to work with from a math perspective than the squared error loss.

Gradient descent. The goal of training a neural network is to minimize the value of the cost function, averaged over all training instances. The dominant method for doing this is *gradient descent*, defined by equation (2) in Grosse, p. 11. The basic idea is very straightforward if you understand multivariate calculus: The gradient gives you the direction of steepest ascent on the cost surface, so adding a negative multiple of the gradient to your current parameters will walk down the direction of steepest ascent. You continue this for a fixed number of training epochs, or until your cost no longer improves, i.e. you have found a local minimum of the cost surface.

I find the explanation in the Grosse notes very good and don't have much to add. Feel free to either stop reading on page 14, or continue for really useful insights into the things that can go wrong in NN training, how to identify them, and how to fix them.

3 Questions to think about **Feb 1**

(a) Draw a very small neural network and manually calculate its activations given two different inputs.

(b) Check that (3) and (1) are actually equivalent, by sketching W and trying it out for your small NN.

(c) For a training set of size one (invented by yourself), compute the gradient of a squared error loss with respect to the parameters of your very small NN. Perform a single step of gradient descent and double-check that the loss decreased.

(d) Implement your neural network with Numpy. How does the cost change if you change the weights?

[2] This is usually achieved by using the softmax function as the final operation of the NN.

4 Backpropagation

Feb 4

- 3Blue1Brown: What is backpropagation really doing?
- 3Blue1Brown: Backpropagation calculus
- Goodfellow et al., Deep Learning Book, Section 6.5

One question remains after our discussion of gradient descent for neural networks: How do you actually compute the gradient? Being able to do this efficiently is crucial; modern neural networks for NLP have billions of parameters, so any inefficiency in calculating the gradient will be punished very harshly in terms of runtime and energy consumption. The *backpropagation algorithm* solves this problem: It efficiently computes the gradient of the cost function with respect to the parameters (i.e., the weights and biases of the NN) through an elegant use of the chain rule for multivariate functions.

The main technical challenge here is that each layer of a neural network is a vector of values, and so we frequently have to compute the gradient of an entire vector of functions with respect to an entire vector of input variables. This can make it very messy and confusing to write down the backpropagation algorithm. Below, I will explain *Jacobian matrices*, which simplify the notation greatly, and then I will go through the backpropagation algorithm of a simple MLP step by step.

Jacobian matrix. Consider a situation where you have a multivariate function f, and each input to f is produced by a previous multivariate function g. More precisely, let $f : \mathbb{R}^n \to \mathbb{R}$ be a function with n inputs, and let $g : \mathbb{R}^m \to \mathbb{R}^n$ be a function with m inputs that returns an n-dimensional output vector. If you prefer, you can think of g as an n-tuple of functions from $\mathbb{R}^m$ to $\mathbb{R}$, whose outputs are then combined into a single vector.

$$\begin{array}{lll} \mathbf{x} & & \in \mathbb{R}^m\text{: inputs} \\ \mathbf{y} & = g(\mathbf{x}) & \in \mathbb{R}^n \\ z & = f(\mathbf{y}) & \in \mathbb{R} \end{array}$$

We would like to find out how the value of z changes with the values in $\mathbf{x}$, i.e. we want to compute the gradient $\nabla_{\mathbf{x}} z$. In the context of backpropagation, z might be the value of the cost function, and $\mathbf{x}$ might be some of the parameters of the NN.

By the multivariate chain rule, we have

$$\frac{\partial z}{\partial x_i} = \sum_{k=1}^{n} \frac{\partial z}{\partial y_k} \frac{\partial y_k}{\partial x_i} \tag{6}$$

That is, we need the partial derivative of every entry in $\mathbf{y}$ with respect to every entry in $\mathbf{x}$. Notice that the equation in (6) looks remarkably like the definition of the matrix-vector product, with the terms $\frac{\partial y_k}{\partial x_i}$ taking the role of the coefficients of a matrix. We can collect these terms in an actual matrix, which is called the *Jacobian*.

Definition 1. Let $\mathbf{y} = f(\mathbf{x})$ be a differentiable function. Then we call the matrix

$$J(\mathbf{y}, \mathbf{x}) = \left(\frac{\partial y_i}{\partial x_k} \right)_{ik}$$

the *Jacobian matrix* of $\mathbf{y}$ with respect to $\mathbf{x}$. If $\mathbf{y} \in \mathbb{R}^n$ and $\mathbf{x} \in \mathbb{R}^m$, the Jacobian is an $n \times m$-matrix.

Given this definition, we can rewrite (6) more succinctly as follows (check this):

$$\nabla_{\mathbf{x}} z = J(\mathbf{y}, \mathbf{x})^T \cdot \nabla_{\mathbf{y}} z. \tag{7}$$

Note that the dimensionalities come out right: The gradient of z with respect to a vector is a vector of the same shape (because we have a partial derivative with respect to each entry). $\mathbf{x}$ and $\mathbf{y}$ are in $\mathbb{R}^m$ and $\mathbb{R}^n$, respectively. $J(\mathbf{y}, \mathbf{x})^T$ is an $m \times n$-matrix, which transforms $\nabla_{\mathbf{y}} z \in \mathbb{R}^n$ into $\nabla_{\mathbf{x}} z \in \mathbb{R}^m$. Spell out the matrix-vector product in (7) to see that it is equivalent to (6) and understand where the transpose comes from.

This means that in order to compute a gradient of the output with respect to more "distant" variables, we can compute the gradient with respect to variables that are "closer" to the output and multiply it with the transpose of the Jacobian. Carrying this step out repeatedly for all computations of the NN yields the backpropagation algorithm.

A simple neural network. In the remainder of this section, we will apply the backpropagation algorithm to compute the gradients for a simple neural network. The network is an MLP with one hidden layer, defined as follows:

$$\begin{array}{lll}
\mathbf{x} & & \in \mathbb{R}^r \text{: inputs} \\
\mathbf{l} & = V\mathbf{x} + \mathbf{b} & V \in \mathbb{R}^{m \times r}, \mathbf{b} \in \mathbb{R}^m \\
\mathbf{a} & = \phi_1(\mathbf{l}) & \in \mathbb{R}^m \\
\mathbf{s} & = W\mathbf{a} + \mathbf{c} & W \in \mathbb{R}^{n \times m}, \mathbf{c} \in \mathbb{R}^n \\
\mathbf{y} & = \phi_2(\mathbf{s}) & \in \mathbb{R}^n \\
\mathcal{E} & = \sum_{i=1}^n (y_i - \hat{y}_i)^2 & \in \mathbb{R}
\end{array}$$

The parameters of this NN are the entries of $V, W, \mathbf{b}, \mathbf{c}$, and we want to compute the gradients of $\mathcal{E}$ with respect to all of these parameters so we can train the NN using gradient descent as explained earlier. The backpropagation algorithm does this by repeatedly applying (7) to compute gradients with respect to the parameters and to other intermediate values. Note that $\phi_1, \phi_2 : \mathbb{R} \to \mathbb{R}$ are arbitrary differentiable activation functions, which are applied elementwise to $\mathbf{l}$ and $\mathbf{s}$ to obtain $\mathbf{a}$ and $\mathbf{y}$, respectively.

A note on training. The formulas above describe the behavior of the neural network on a single input $\mathbf{x}$, predicting a single output $\mathbf{y}$. In reality, the training data would consist of N instances, each consisting of an input $\mathbf{x}^{(i)}$ and a gold output $\hat{\mathbf{y}}^{(i)}$, where $1 \leq i \leq N$. The neural network predicts an output $\mathbf{y}^{(i)}$ for each instance, and we can therefore compute a cost $\mathcal{E}^{(i)}$ that measures the distance between the prediction and the gold output. The overall cost of the training data is $\mathcal{E} = \sum_{i=1}^{n} \mathcal{E}^{(i)}$.

We want to minimize $\mathcal{E}$ by performing gradient descent, and thus we want to compute the gradient of $\mathcal{E}$ with respect to all parameters, e.g. $\nabla_V \mathcal{E}$. In practice, we compute these gradients instance-wise; they will then sum to the gradient of the whole corpus because of the sum rule for derivatives:

$$\nabla_V \mathcal{E} = \nabla_V \left(\sum_{i=1}^{N} E^{(i)} \right) = \sum_{i=1}^{N} \nabla_V E^{(i)} \tag{8}$$

Below, we pretend there is only a single training instance and its cost is simply $\mathcal{E}$. But the entire discussion generalizes to the case of larger training data sets through equation (8).

Gradient with respect to $\mathbf{y}$ and $\mathbf{s}$. We start with the gradient of $\mathcal{E}$ with respect to itself, which is trivial:

$$\nabla_{\mathcal{E}} \mathcal{E} = 1 \quad \in \mathbb{R} \tag{9}$$

In order to backpropagate this gradient to $\nabla_{\mathbf{y}} \mathcal{E}$, we need the Jacobian of $\mathcal{E}$ with respect to $\mathbf{y}$. Taking the partial derivatives of $\mathcal{E}$ with respect to each y_i, we obtain this matrix:

$$J(\mathcal{E}, \mathbf{y}) = \left(\frac{\partial \mathcal{E}}{\partial y_i} \right)_i = \left(2(y_i - \hat{y}_i) \right)_i \quad \in \mathbb{R}^{1 \times n} \tag{10}$$

Using (7), we obtain

$$\nabla_{\mathbf{y}} \mathcal{E} = J(\mathcal{E}, \mathbf{y})^T \cdot \nabla_{\mathcal{E}} \mathcal{E} = J(\mathcal{E}, \mathbf{y})^T \quad \in \mathbb{R}^n \tag{11}$$

As expected, the gradient for $\mathbf{y} \in \mathbb{R}^n$ is a vector in $\mathbb{R}^n$, which tells us how to update each component of $\mathbf{y}$ during gradient descent. Notice that we are switching as needed between different ways of looking at isomorphic objects: $\nabla_{\mathcal{E}} \mathcal{E}$ could be a real number or a column vector in $\mathbb{R}^1$; $J(\mathcal{E}, \mathbf{y})^T$ could be an $n \times 1$-matrix or an n-dimensional column vector in $\mathbb{R}^n$; and so on.

Finally, $\mathbf{y}$ is computed from $\mathbf{s}$ by elementwise application of the activation function ϕ_2, so the Jacobi matrix is very simple:

$$J(\mathbf{y}, \mathbf{s}) = \left(\delta_{ik} \, \phi_2'(s_i) \right)_{i,k} \quad \in \mathbb{R}^{n \times n}, \tag{12}$$

where δ_{ik} is the *Kronecker delta*, i.e. $\delta_{ik} = 1$ if $i = k$ and $\delta_{ik} = 0$ otherwise. Thus, $J(\mathbf{y}, \mathbf{s})$ is a diagonal matrix with the derivatives of ϕ_2 on the main diagonal. We obtain

$$\nabla_{\mathbf{s}} \mathcal{E} = J(\mathbf{y}, \mathbf{s})^T \cdot \nabla_{\mathbf{y}} \mathcal{E} = \left(2\phi_2'(s_i)(y_i - \hat{y}_i) \right)_i \quad \in \mathbb{R}^n \tag{13}$$

Gradient with respect to a. Now that we have the gradient with respect to $\mathbf{s}$, we can backpropagate into multiple objects. We can backpropagate into the parameters W and $\mathbf{c}$; we will do this in a moment. We can also backpropagate into $\mathbf{a}$. This gives us the gradient $\nabla_{\mathbf{a}} \mathcal{E}$, from which we can then backpropagate into the previous layer of the neural network and its parameters.

Given the definition of $\mathbf{a}$, we have the partial derivative $\frac{\partial s_i}{\partial a_k} = W_{ik}$ (check this by spelling out the matrix-vector product $W\mathbf{a}$). Thus the Jacobian is simply

$$J(\mathbf{s}, \mathbf{a}) = W \quad \in R^{n \times m}, \tag{14}$$

and we get the gradient

$$\nabla_{\mathbf{a}} \mathcal{E} = J(\mathbf{s}, \mathbf{a})^T \cdot \nabla_{\mathbf{s}} \mathcal{E} = W^T \cdot \nabla_{\mathbf{s}} \mathcal{E} \quad \in \mathbb{R}^m. \tag{15}$$

This construction can be continued layer by layer, until we have computed gradients for the entire neural network.

Gradient with respect to W and $\mathbf{c}$. Finally, let's compute the gradients with respect to the parameters in the second layer, the weight matrix W and the bias $\mathbf{c}$. These are the gradients we actually need in order to make a gradient descent update to the parameters.

The gradient $\nabla_{\mathbf{c}} \mathcal{E}$ with respect to the bias is standard by now. We have the particularly simple situation that $J(\mathbf{s}, \mathbf{c}) = I_n$, the $n \times n$ identity matrix (check this). Thus we get

$$\nabla_{\mathbf{c}} \mathcal{E} = J(\mathbf{s}, \mathbf{c})^T \cdot \nabla_{\mathbf{s}} \mathcal{E} = \nabla_{\mathbf{s}} \mathcal{E} \quad \in \mathbb{R}^n. \tag{16}$$

The gradient $\nabla_W \mathcal{E}$ with respect to the weight matrix is a bit more complicated, because now we need to look at all the partial derivatives $\frac{\partial s_i}{\partial W_{jk}}$. These don't fit into a neat two-axis Jacobian matrix. We could define a *three-axis Jacobian tensor* whose entries are indexed by i, j, k, but we are almost done with this class and I don't want to define yet another concept. So let's do it by hand. First, we have (check this)

$$\frac{\partial s_i}{\partial W_{jk}} = \delta_{ij}\, a_k. \tag{17}$$

That is, the partial derivative is nonzero only if $i = j$; the j-th row of the weight matrix has no influence on the i-th row of the column vector $\mathbf{s}$ for $i \neq j$. We can then spell out

the partial derivatives of $\mathcal{E}$ using a direct application of the multivariate chain rule:

$$\begin{aligned}
\frac{\partial \mathcal{E}}{\partial W_{jk}} &= \sum_{i=1}^{n} \frac{\partial \mathcal{E}}{\partial s_i} \frac{\partial s_i}{\partial W_{jk}} \\
&= \sum_{i=1}^{n} \frac{\partial \mathcal{E}}{\partial s_i} \, \delta_{ij} \, a_k \quad \text{(spell out } \frac{\partial s_i}{\partial W_{jk}}\text{)} \\
&= \frac{\partial \mathcal{E}}{\partial s_j} \, a_k \qquad \text{(terms with } i \neq j \text{ are zero)}
\end{aligned} \tag{18}$$

Thus we obtain the gradient

$$\nabla_W \, \mathcal{E} = \left(a_k \cdot \frac{\partial \mathcal{E}}{\partial s_j} \right)_{jk} \quad \in \mathbb{R}^{n \times m}. \tag{19}$$

Backpropagation in practice. At this point, you are able to carry out the backpropagation algorithm by hand and perform gradient descent on a small neural network. You could also implement it in Numpy for a specific neural network, and this is a really good exercise to gain a deeper understanding of the algorithm. Note that you will need access to the intermediate results of the calculation, such as $\mathbf{s}$ and $\mathbf{a}$, in order to compute all parts of the gradient. Thus if you have packaged the forward evaluation of the neural network into a function in question (d) above, you will need to find a way to make these intermediate results accessible to your gradient calculation.

Of course, in the daily practice of developing, debugging, and training neural networks, it would be far too labor-intensive if you had to do backpropagation by hand, or even if you had to change your Numpy implementation of backprop every time you changed the neural network. We therefore use specialized libraries for neural networks, most prominently Tensorflow (by Google) and Pytorch (by Facebook).

These libraries provide two core functions: They have an *autograd* module which automatically computes derivatives, and they have very efficient implementations for matrix multiplication. They also predefine many functions that we use to define layers and losses, and provide support for running the neural network on various kinds of hardware, including the GPUs in graphics cards and TPUs, specialized hardware developed by Google specifically for neural network operations. This allows all gradient computations in one minibatch to be carried out at physically the same time, which greatly speeds up training.

If you have some time over the term break, I warmly recommend that you implement and train a few simple neural networks. I personally prefer Pytorch over Tensorflow because it doesn't require you to define a static computation graph, which means that you can write code that mostly just looks like Numpy code, and then Pytorch will do all the heavy lifting regarding backpropagation for you behind the scenes. But of course Tensorflow is also a more than legitimate alternative.

You should also strongly consider taking Dietrich Klakow's course on "Neural Networks: Implementation and Application", which continues where we are leaving off and explains the theory and practice behind neural networks. With the math background you acquired here, you should be well prepared to handle all the technical intricacies.

5 Questions to think about **Feb 4**

(a) Carry out the rest of the backpropagation algorithm to obtain the gradients $\nabla_V \mathcal{E}$ and $\nabla_{\mathbf{b}} \mathcal{E}$.

(b) Compute the gradients for your simple NN from Section 3 and perform one gradient descent step with a learning rate of your choice.

(c) Why don't we need gradients $\nabla_{\mathbf{x}} \mathcal{E}$ for the inputs?

(d) Implement backpropagation for a small neural network in Numpy and train the NN on some data.