Assignment 7: Pytorch, Vanishing Gradients, and Convolutional Neural Networks

Machine Learning

Fall 2019

Learning Objectives

- Learn the basics of Pytorch
- Understand the vanishing gradient problem and how to fix it
- Learn about convolutional neural networks from a conceptual perspective

Prior Knowledge Utilized

- Multilayer perceptron
- Backpropagation

1 Pytorch and Autograd Algorithms

In this assignment you will be learning about another new Python library: pytorch. Pytorch is an open-source library designed for high-performance (meaning fast, rather than necessarily high accuracy) machine learning. There are a number of other similar frameworks out there. We chose pytorch due to its straightforward data handling, flexibility, and support for debugging. Below, we'll list some other frameworks and why we ultimately decided to go with pytorch.

- Keras also lets you construct high-performance neural networks in Python. The biggest downside in our eyes is that it hides away a lot of the details of how data moves through the network. This makes it a bit too abstract for our purposes (e.g., harder to debug and less flexible).
- TensorFlow is another package that is similar to pytorch. Tensorflow is more popular than pytorch, but it is a bit more complicated and has a steeper learning curve.
- sklearn (which you've gotten to use) has support for Multilayer Perceptrons (as we saw in the last notebook). That said, it doesn't support that many different types of networks, is hard to customize, and doesn't support high performance computation using GPUs (graphics processing units).
- numpy could also be an option. It is very flexible, but it is too low-level. Like sklearn it doesn't support high performance computation with GPUs.

Before you go through some resources on how to use pytorch, let's discuss what we are hoping to get from pytorch.

- The ability to automatically compute gradients of our loss function with respect to parameters of a neural network!
- The ability to train neural networks on a GPU (for performance boosts of up to 20 times). Luckily, Google is nice enough to let you use their GPUs through Colab!
- Tools for debugging common problems with neural networks.

- A module called torchvision for working with images in neural networks.

Go through the following tutorials for pytorch. This will give you the barebones. Later in this document you'll explore pytorch again via a companion notebook.

External Resource(s) (15 minutes)

We recommend you go through these with Colab open so you can try the blocks of code yourself.

- Tensor Tutorial
- Autograd Tutorial This second tutorial mentions a pre-trained network. In our assignments, we're focusing on the initial training of neural networks. This gives you a comprehensive understanding of how they work. But, to save time, sometimes people will use pre-trained networks to speed things up. We'll learn about that later in class.

External Resource(s) (60 minutes)

Now that you have the barebones basics for Pytorch, let's look back at our old friend the logistic regression model. In this companion notebook you'll play with an implementation of logistic regression using pytorch and compare it with sklearn's implementation of logistic regression. In the notebook we'll use the Titanic data as our example.

2 Backpropagation and the Problem of Vanishing (or Exploding) Gradients

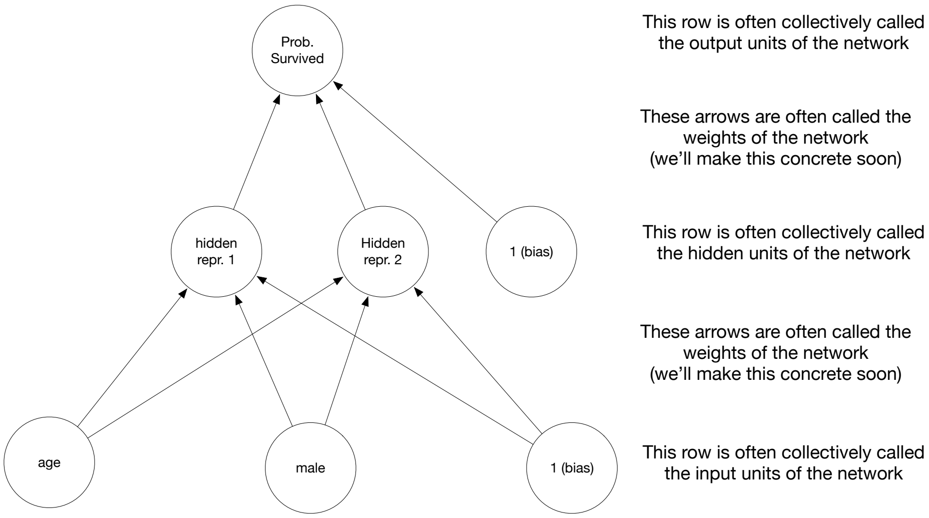
One possible reaction to reading about the autograd capabilities in pytorch is to think "Why didn't Paul and Sam tell me about this before? Why have we been wasting our time doing all this math to compute gradients?" There are two really good (in our opinion) answers to this question.

1. Knowing how backpropagation works gives you a more powerful mental model of how these networks are trained, how to diagnose failures in training, and potentially how to fix these failures.
2. The math you learned along the way is useful even outside of machine learning.

One particularly important way in which knowing backpropagation helps you understand how to train a better network is when confronting something known as the *vanishing gradient problem* (or its more dramatic sounding counterpart the *exploding gradient problem*). You'll be reading all about it in just a bit, but before we dive in let's review the multi-layer perceptron (MLP). If you feel good about the MLP, skip the next box.

Recall: Multi-layer perceptron

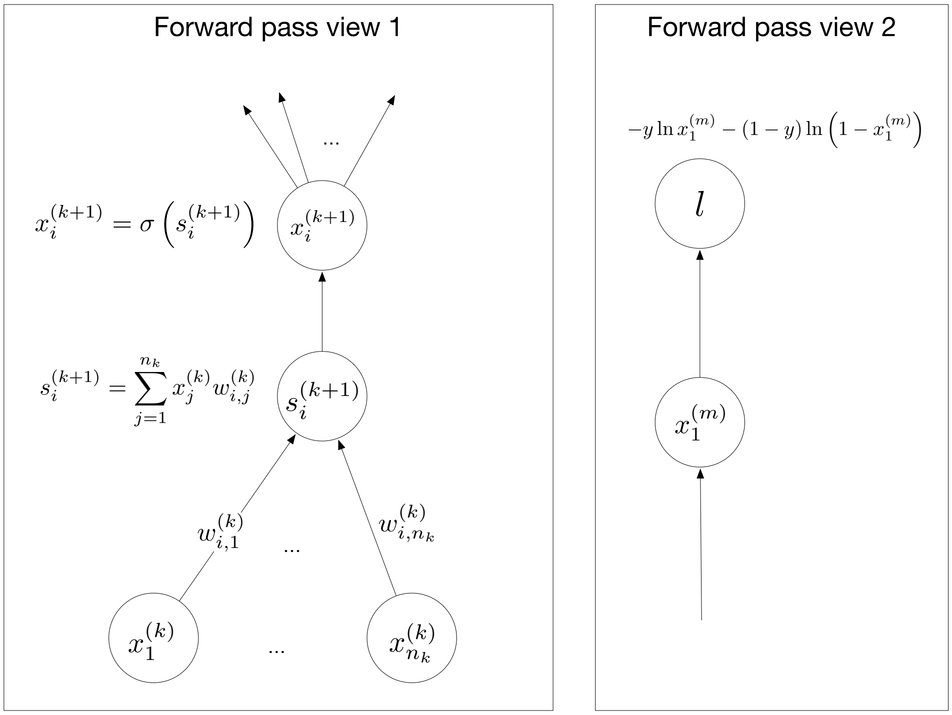
The multi-layer perceptron consists of a bunch of single-layer perceptrons stacked on top of each other. Last assignment we saw a particular special case of the MLP where we take logistic regression models and stack them. A cartoon version of the network applied to the titanic dataset is shown below.


Prob.
Survived
hidden
repr. 1
Hidden
repr. 2
1 (bias)
age
male
1 (bias)
This row is often collectively called
the output units of the network
These arrows are often called the
weights of the network
(we'll make this concrete soon)
This row is often collectively called
the hidden units of the network
These arrows are often called the
weights of the network
(we'll make this concrete soon)
This row is often collectively called
the input units of the network

We saw in the companion notebook that such networks are capable of learning useful internal representations for prediction. To make this figure more precise, we can zoom in on how data propagates from lower layers to higher layers in the network (view 1 in the figure below) and ultimately to the loss function (view 2 in the figure below).

Before presenting the figure, for your convenience here is a notation guide for the figure.

l	the loss function for the network (log loss in this case)
m	the number of layers in the network
n_k	the number of nodes in the kth layer
$x_j^{(k)}$	The jth node in the kth layer of the network ($k = 1$ is the input and $k = m$ is the output)
$\mathbf{x}^{(\mathbf{k})}$	The nodes at the kth layer in vector form
$s_i^{(k)}$	the ith summation node in the kth layer
$w_{i,j}^{(k)}$	the weight connecting the jth node in layer k to the ith node in layer $k + 1$
$\mathbf{w}_{\mathbf{i}}^{(\mathbf{k})}$	the vector of weights to the ith summation node in layer $k + 1$.


Forward pass view 1
...
$x_i^{(k+1)} = \sigma\left(s_i^{(k+1)}\right)$
$x_i^{(k+1)}$
$s_i^{(k+1)} = \sum_{j=1}^{n_k} x_j^{(k)} w_{i,j}^{(k)}$
$s_i^{(k+1)}$
$w_{i,1}^{(k)}$
...
$w_{i,n_k}^{(k)}$
$x_1^{(k)}$
...
$x_{n_k}^{(k)}$
Forward pass view 2
$-y \ln x_1^{(m)} - (1-y)\ln\left(1 - x_1^{(m)}\right)$
l
$x_1^{(m)}$

In the previous assignment you derived the backpropagation algorithm, which can be used to compute the gradient of the weights in an MLP with respect to the loss function. What's beautiful about this formula is that it works for networks of arbitrary depth. You may have heard of deep learning, which is where networks of dozens or even hundreds of layers are trained on a particular task. Despite their complexity, these networks are typically trained using the backpropagation algorithm you met in the last assignment!

Despite the relative simplicity of the backpropagation equations, there are some dangers lurking when applying the algorithm to deep networks (m very large). The most common issues that one encounters are **vanishing and exploding gradients**. A vanishing gradient is when the gradient of the loss with respect to the weights in a layer becomes vanishingly small as you move from the output layer towards the input layer. An exploding gradient is the opposite problem (when the gradient of the loss with respect to the weights in a layer becomes exceedingly large as you move from the output layer towards the input layer).

Exercise 1 (45 minutes) or (105 minutes including optional part)

Before starting this exercise, go and learn about the vanishing gradient problem. Here are a few resources to look at. You do not need to read all of them, so please pick the ones that seem the most inline with how you learn.

External Resource(s)

- Rohan Kapur's Intuitive Explanation of the Vanishing Gradient Problem
- Video resource: Deep Learning Simplified: An Old Problem

Last assignment we derived the following backpropagation equations.

$$(\nabla_{\mathbf{w_i^{(k)}}})l = \mathbf{x^{(k)}}\sigma\left(s_i^{(k+1)}\right)\left(1-\sigma\left(s_i^{(k+1)}\right)\right)\frac{\partial l}{\partial x_i^{(k+1)}} \qquad \text{gives us the gradient of the weights going into a unit} \tag{1}$$

$$\frac{\partial l}{\partial x_i^{(k)}} = \sum_{j=1}^{n_{k+1}} w_{j,i}^{(k)}\sigma\left(s_j^{(k+1)}\right)\left(1-\sigma\left(s_j^{(k+1)}\right)\right)\frac{\partial l}{\partial x_j^{(k+1)}} \qquad \text{recursively gives the partial of the loss w.r.t. the units} \tag{2}$$

$$\frac{\partial l}{\partial x_1^{(m)}} = -y\frac{1}{x_1^{(m)}} + (1-y)\frac{1}{1-x_1^{(m)}} \qquad \text{provides the base case of the recursion} \tag{3}$$

(a) If we examine these equations, we can understand where the vanishing gradient problem comes from. In particular, Equation 2 provides the crucial recursive formula for defining the partial derivatives of the loss with respect to nodes in the network in terms of the same partial derivatives for nodes deeper in the network (i.e., closer to the output). Based on Equation 2, what aspects of the network might contribute to or counteract the vanishing gradient problem? Consider things such as the values of the summation units $s_j^{(k+1)}$, the values of the weights, and the number of units at each layer of the network (the n_k's).

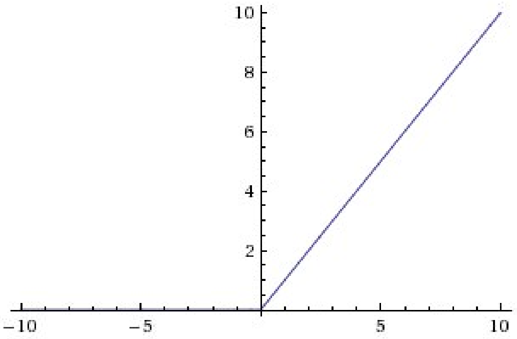
☆ Solution

- The product of $\sigma(s_j^{(k+1)})(1-\sigma(s_j^{(k+1)}))$ is at most $\frac{1}{4}$. Further, if the value of $s_j^{(k+1)}$ is either very positive or very negative, the product will be very, very small. Thus $\sigma(s_j^{(k+1)})(1-\sigma(s_j^{(k+1)}))$ will tend to decrease gradients as they go through the network.
- The weight term $w_{j,i}^{(k)}$ could act to either increase gradients or decrease gradients depending on whether, on average, its magnitude is bigger or smaller than 1.
- Since we are summing over n_{k+1} units, having more units will tend to increase the values of the gradient as you move backward down the layers of the network.

(b) It turns out that the sigmoid function is only one viable choice for transforming the summation units into the values of the units for the next layer. In fact, many different non-linear functions can be used (e.g., tanh). The function used in this capacity is called an *activation function*. If we let a refer to our activation function, then Equation 2 becomes

$$\frac{\partial l}{\partial x_i^{(k)}} = \sum_{j=1}^{n_{k+1}} w_{j,i}^{(k)} a'\left(s_j^{(k+1)}\right)\frac{\partial l}{\partial x_j^{(k+1)}} \ . \tag{4}$$

One very popular choice of activation function is called rectified linear (or ReLu for *rectified linear unit*). A graph of the ReLu activation function is shown below.



How might choosing the ReLu as an activation function help with the vanishing gradient problem in comparison to choosing the sigmoid as the activation function (use the backpropagation equations to arrive at your answer)?

☆ Solution

the derivative of the function $ReLu(s_j^{(k+1)})$ will be 1 if $s_j^{(k+1)}$ is greater than 0. The derivative will be 0 if $s_j^{(k+1)} \leq 0$. Looking back at our equation, this means that the gradients from upper layers will either be preserved (if $s_j^{(k+1)} > 0$) or zeroed out (if $s_j^{(k+1)} \leq 0$). As long as at least a few of the values $s_1^{(k+1)}, \ldots, s_{n_{k+1}}^{(k+1)}$ are positive, the partial derivative magnitudes won't be decreased to 0. In order to avoid exploding gradients, you would want to make sure the weight magnitudes are in the appropriate range to counteract exponential growth in the partial derivatives (this is why weight initialization is the focus of a lot of research in deep learning).

(c) **Going Beyond (optional)** Using pytorch, perform a computational investigation of the vanishing gradient problem. We suggest the following steps (note: we have our own version of this in the solutions if you want to check it out):

- Create an MLP where the number of layers is passed in as a parameter.
- Before doing any training of the network, feed an arbitrary input into the network, compute the loss, and then compute the gradient of the loss with respect to the weights at each level of the network.
- Summarize the partial derivatives of the weights at each level in the network (e.g., take the mean of the absolute values) and plot this quantity as a function of k (the layer number). It may be helpful to use a log scale for the y-axis.

☆ Solution

This is pretty rough, but here is our go of this. We'll update after class.

3 *Convolutions and Image Filtering*

Depending on your math background you may have seen the convolution operation before. There are various types of convolutions, including convolutions with continuous functions, discrete functions, in 1D, in 2D, or in higher dimensions. In this class we're only going to be talking about discrete, 2D convolutions. It turns out that such convolutions provide a compelling way to think about processing data with intrinsically 2D structure (such as images).

Check out at least one of the resources below to learn how convolutions can be used to filter and transform images.

External Resource(s) (30 minutes)

- Image Filtering
- Video resource: How blurs and filters work

Exercise 2 (10 minutes)

This is not an exercise per se. On the quiz you should mark whether or not you were able to engage with the readings and get a basic grasp of the topic. As a basic guide, you should have some familiarity with the following ideas.

- What a 2D convolutional kernel is and how it is applied to a 2D matrix (e.g., an image).
- What happens when applying the kernel around the edges of the 2D matrix?
- Understand how to reason about what a kernel does to an image based on the form of the kernel.

4 *Convolutional Neural Networks*

We now come to a particular type of neural network called a convolutional neural network. These networks supplement the perceptron layers we've seen thus far with convolutional layers that apply various filtering operations to the data. These networks are very well-suited for working with images since they are able to customize their filtering operations to the dataset in a way that maximizes their ability to solve the task. There are lots of great resources on convolutional neural networks out there. Please don't feel like you have to go through all of these.

External Resource(s) (60 minutes)

- Intuitive Explanation of Convnets (this one is a very well done high-level overview)
- Beautiful interactive visualization of a Convnet trained to recognize handwritten digits (this is great, especially for visual learners. Make sure to click on the activations at a layer to see the learned filter. Don't miss this one!)
- Convnet.js (convnets training in your browser?!? Another good one for visual learners)
- Lecture 5 from CS231 at Stanford (if you want a lecture, this one is good. it starts getting a bit too low-level around the 35 minute mark. there are some nice slides on neural network history at the beginning).

Exercise 3 (15 minutes)

Open up Convnet.js and try it on either MNIST or CIFAR-10. Watch the network train. Interpret as much of what you see as possible. If you don't understand something, ask about it on Discord.