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
- Apply convolutional neural networks to a real dataset.

Prior Knowledge Utilized

- Multilayer perceptron
- Backpropagation

1 Pytorch and Autograd Algorithms

External Resource(s)

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

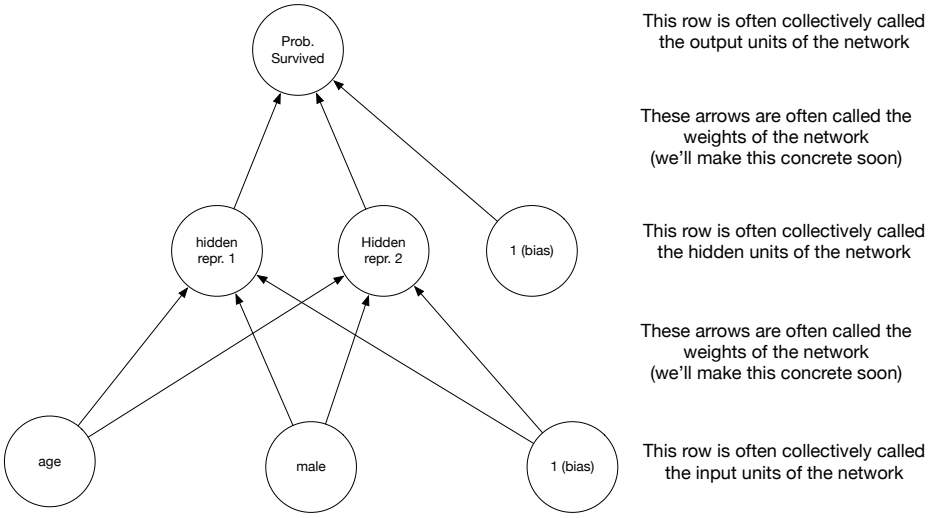
- Tensor Tutorial
- Autograd Tutorial

1.1 Titanic III

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

Recall: Multi-layer perceptron

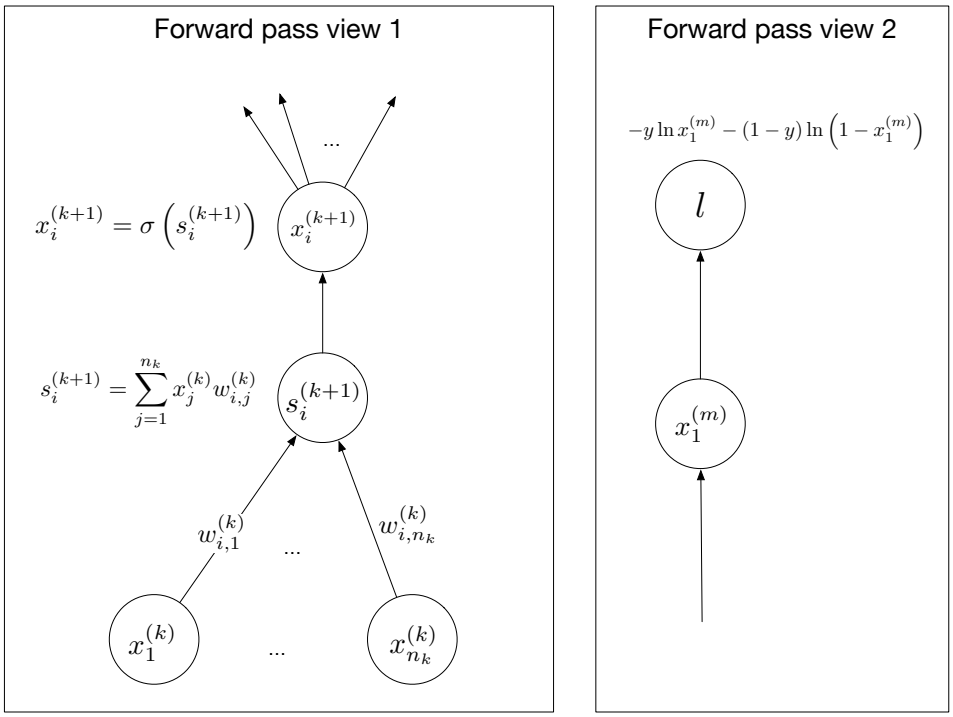
The multi-layer perceptron consists of a bunch of single-layer perceptrons stacked on top of each other. Last assignment we saw a particular special case of this where we just a few logistic regression models and stack them. A cartoon version of the network applied to the titanic dataset is shown below.



We saw in the companion notebook that the network that networks of this type are capable of. We can make things even more precise by zooming in on how data propagates from lower layers to higher layers (view 1 in the figure below) and ultimately to the loss function (view 2 the figure below), which helps us understand how to tune our weights to the data.

Notation Guide:

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)
$\mathbf{x^{(k)}}$	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 $k+1$
$\mathbf{w_i^{(k)}}$	the vector of weights to the ith summation node in layer $k+1$.



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 pitfalls 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 back 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 (30-90 minutes)

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
- Deep Learning Simplified: An Old Problem
- Vanishing Gradient Problem on Wikipedia (we recommend you stop reading once you get to the section marked *Solutions*.

Last assignment we derived the following backpropagation equations.

$$(\nabla_{\mathbf{w_i^{(k)}}})l = \mathbf{x^{(k)}}\sigma\left(s_i^{(k+1)}\right)\left(1-\sigma\left(s_i^{(k+1)}\right)\right)\frac{\partial l}{\partial x_i^{(k+1)}} \tag{1}$$

$$\frac{\partial l}{\partial x_i^{(k)}} = \sum_{j=1}^{n_{k+1}} w_{j,i}^{(k)}\sigma\left(s_j^{k+1}\right)\left(1-\sigma\left(s_j^{k+1}\right)\right)\frac{\partial l}{\partial x_j^{(k+1)}} \tag{2}$$

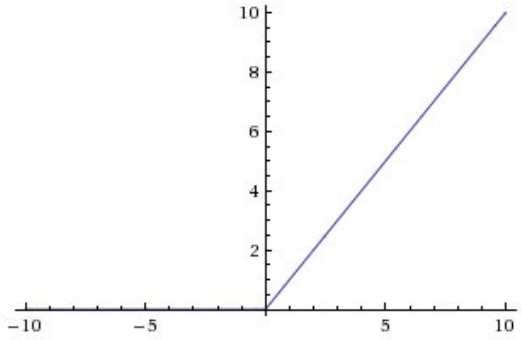
$$\frac{\partial l}{\partial x_1^{(m)}} = -y\frac{1}{x_1^m} + (1-y)\frac{1}{1-x_1^{(m)}} \tag{3}$$

(a) If we examine these equations, we can understand much better where the vanishing gradient problem comes from. In particular, Equation **??** 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 **??**, 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 eat each layer of the network (n_k).

(b) It turns out that the use of the sigmoid function to transform the summation units when calculating the values at the next layer is not the only viable choice. It turns out that many different types of 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 **??** 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 does choosing the ReLu as an activation function help with the vanishing gradient problem versus choosing the sigmoid.

(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 file 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.

3 Convolutions and Image Filtering

Depending on your math background you may have seen the convolution operation before.

External Resource(s) (30 minutes)

- Image Filtering
- Video: How blurs and filters work

4 Convolutional Neural Networks

Quick blurb here.

External Resource(s) (60 minutes)

- Intuitive Explanation of Convnets
- Beautiful interactive visualization of a Convnet trained to recognize handwritten digits (don't miss this one!)

- Convnet.js
- Convolutional Neural Networks Explained
- Lecture 5 from CS231 at Stanford (it starts getting a bit too low-level around the 35 minute mark)

5 Training Your First Convnet

TODO: NINJAs to work on this.

6 Something on Context and Ethics and Convnets (if we pushed project out another day, I would vote for this

7 Getting Ready for the Project

Sam to work on this.

7.1 Goals

7.2 Datasets