Homework 1

An Introduction to Neural Networks

11-785: Introduction to Deep Learning (Spring 2019)

OUT: **Jun __, 2019**

DUE: **August 10, 2019, 11:59 PM**

Start Here

- **Overview:**
  - **Part 1**: All of the problems in Part 1 are graded locally by the provided autograder. This assignment has 8 points total.
  - **Part 2**: Your score on this section of the homework depends on the accuracy of the predictions of your model. A local script is provided for you to see the accuracy of your model.

  The handout also contains:

  - **Appendix**: This contains information and formulas about functions you have to implement as part of the homework
  - **Glossary**: This contains basic definitions to most of the technical vocabulary used in the handout

  In addition, you will find the page:

  `http://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2019/www/hwnotes/HW1p1.html`

  extremely helpful. It contains detailed information about the key topics needed to complete the homework.

  Bear in mind that all of the concepts mentioned will be covered in detail during the course. This is only an introduction to help you get a head start for the course.

- **Submission:**
  - **Part 1**: The compressed `handout` folder contains a directory named `test` which contains a directory named `hw1`, which contains the file, `hw1.py`. This contains a template for solving all of the problems and includes some useful helper functions. Make sure that all class and function definitions originally specified (as well as class attributes and methods) in `hw1.py` are fully implemented to the specification provided in this writeup and in the docstrings or comments in `hw1.py`.

    To test your code, you can use the local autograder: run the `runner.py` file located in the main directory (`part1`). You will see your score for each of the functions you must implement printed out. You do not need to submit the code anywhere.
  - **Part 2**: You need to generate a csv file containing your model's predictions on the test set. The csv must have one column (no heading) which contains the labels generated by your model for the test data. E.g you can use `np.savetxt(filename, pred_labels, delimiter=',')`)

    Then call `python test_mnist.py prediction.csv` from the command line to evaluate the accuracy of your predictions.

HW Part 1: A Simple Neural Network

A neural network is a "computer system modelled on the human brain and nervous system". The purpose of having such a system is to try and have a computer emulate the learning processes that occur in the human brain. Intuitively, having fixed mathematical functions is not enough to replicate the processes in the human brain. This led to the creation of neural networks.

For part 1 of the homework you will write your own implementation of the backpropagation algorithm for training your own neural network, as well as a few other features such as activation and loss functions. You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc) - you are only permitted and recommended to vectorize your computation using the Numpy library.

In the file `hw1.py` we provide you with classes whose methods you have to fill. The autograder tests will compare the outputs of your methods and attributes of your classes with a reference solution. Therefore we enforce for a large part the design of your code ; however, you still have a lot of freedom in your implementation . We recommend that you look through all of the problems before attempting the first problem. However we do recommend you complete the problems in order as the difficulty increases, and questions often rely on the completion of previous questions.

Loss and Activation Functions

Here are some links to articles to help you better understand loss and activation functions:

- Introduction to loss functions
- Understanding activation functions

1 ReLU [4 points]

You will implement the ReLU activation class. It comes in the form of a class which has a forward and a derivative method. Both methods must be implemented. The Identity activation has been implemented for you as an example. Both the Identity anf ReLU classes derive from Activation which acts as an abstract base class / interface.

You only need to implement the two methods for ReLU. No helper functions or additional classes should be necessary. Keep your code as concise as possible and leverage Numpy as much as possible.

Be mindful that you are working with matrices which have a given shape as well as a set of values.

The output of the activation should be stored in the `self.state` variable of the class. The `self.state` variable should be further used for calculating the derivative during the backward pass.

You can refer to the appendix for more details on the function

2 SoftmaxCrossEntropy [4 points]

For this homework, we will be using the softmax cross entropy loss detailed in the appendix of this writeup. Use the `LogSumExp` trick (see appendix) to ensure numerical stability.

Implement the methods specified in the class definition `SoftmaxCrossEntropy`. This class inherits the base `Criterion` class.

Again no helper methods should be needed and you should stride to keep your code as simple as possible. Use 1e-8 for eps.

You can find helpful information on implementing the function here: `https://deepnotes.io/softmax-crossentropy`

You can also refer to the appendix for more details on the function.

2.0.1 `Forward`

Implement the softmax cross entropy operation on a batch of output vectors.

Hint: Add a class attribute to keep track of intermediate values necessary for the backward computation.

- Input shapes:
  - x: (batch size, 10)
  - y: (batch size, 1)
- Output Shape:
  - out: (batch size, 1)

2.0.2 `Backward`

Perform the 'backward' pass of softmax cross entropy operation using intermediate values saved in the forward pass.

- Output shapes:
  - out: (batch size, 10)

HW Part 2: MNIST Classification

This is for the summer 2019 version of 11-785 Homework 1 Part 2.

In this task, you need to implement a multi-layer perceptron to classify handwritten digits (0-9).

You will be using the MNIST dataset, which contains a collection of hand-written digits. Your task is to classify each entry as one of the 9 digits.

You are only allowed to use MLPs, no RNNs or ConvNets. You may use auto differentiation libraries such as PyTorch and Tensorflow

Note: we recommend Pytorch as this is what TA's will be able to provide most support for during the semester

We suggest that you begin with a simple MLP with the following specifications:

- Linear layer: in_features = 784, out_features = 512
- ReLU layer
- Linear layer: in_features = 512, out_features = 10

Note that 784 corresponds to the number of input features of the data, and 10 corresponds the number of output classes (see descriptions of data below)

Once you have the above model implemented and working, start playing around with the number of layers, their sizes, the activation function, using dropout, ... whatever you would like!

Data

- **train.npy**: The training data. Array of shape (40,000, 784) i.e. array containing 50,000 entries each with 784 features. Note that the 784 features correspond to the 784 pixels of the 28 * 28 images used
- **train_labels.npy**: The training labels. Array of shape (50,000) i.e. array containing the labels corresponding to the entries in train.npy
- **dev.npy**: The dev set. Array of shape (10,000, 784) i.e. array containing 10,000 entries each with 784 features. Note that the 784 features correspond to the 784 pixels of the 28 * 28 images used
- **dev_labels.npy**: The dev labels. Array of shape (10,000) i.e. array containing the labels corresponding to the entries in train.npy
- **test.npy**: The testing data. Array of shape (10,000, 784) i.e. array containing 10,000 entries, of the same format as in train.npy.

Please refer to the submission section for details on seeing the accuracy of your predictions.

Appendix

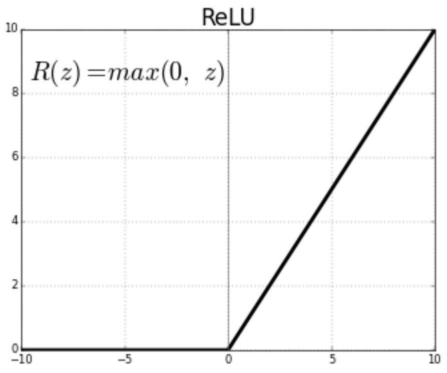
A ReLU

The ReLU (Rectified Linear Unit) function is one of the most widely used activation functions.

The formula for the forward function is as follows:

$$ReLU(x) = max(0, x)$$

Which is given by the following graph:



The derivative is equally as simple.

By the basic rules of calculus, the derivative of the identity function is 1, and the derivative of a horizontal line is 0. This gives us the following formula for the derivative of the ReLU function:

$$\begin{cases} 1 \text{ if x } \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

B Softmax Cross Entropy

Let's define softmax, a useful function that offers a smooth (and differentiable) version of the max function with a nice probabilistic interpretation. We denote the softmax function for a logit x_j of K logits (outputs) as $\sigma(x_j)$.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

Notice that the softmax function properly normalizes the k logits, so we can interpret each $\sigma(x_j)$ as a probability and the largest logit x_j will have the greatest mass in the distribution.

$$\sum_{j=1}^{K} \sigma(x_j) = \frac{1}{\sum_{k=1}^{K} e^{x_k}} \sum_{j=1}^{K} e^{x_j} = 1$$

For two distributions P and Q, we define cross-entropy over discrete events X as

$$CE = \sum_{x \in X} P(x) \log \frac{1}{Q(x)} = -\sum_{x \in X} P(x) \log Q(x)$$

Cross-entropy comes from information theory, where it is defined as the expected information quantified as $\log \frac{1}{q}$ of some subjective distribution Q over an objective distribution P. It quantifies how much information we (our model Q) receive when we observe true outcomes from Q – it tells us how far our model is from the true distribution P. We minimize Cross-Entropy when $P = Q$. The value of cross-entropy in this case is known simply as the entropy.

$$H = \sum_{x \in X} P(x) \log \frac{1}{P(x)} = -\sum_{x \in X} P(x) \log P(x)$$

This is the irreducible information we receive when we observe the outcome of a random process. Consider a coin toss. Even if we know the Bernoulli parameter p (the probability of heads) ahead of time, we will never have absolute certainty about the outcome until we actually observe the toss. The greater p is, the more certain we are about the outcome and the less information we expect to receive upon observation.

We can normalize our network output using softmax and then use Cross-Entropy as an objective function. Our softmax outputs represent Q, our subjective distribution. We will denote each softmax output as $\hat{y}_j$ and represent the true distribution P with output labels $y_j = 1$ when the label is for each output. We let $y_j = 1$ when the label is j and $y_j = 0$ otherwise. The result is a degenerate distribution that will aim to estimate P when averaged over the training set. Let's formalize this objective function and take the derivative.

$$\begin{aligned}
L(\hat{y}, y) &= CE(\hat{y}, y) \\
&= -\sum_{i=1}^{K} y_i \log \hat{y}_i \\
&= -\sum_{i=1}^{K} y_i \log \sigma(x_i) \\
&= -\sum_{i=1}^{K} y_i \log \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \\
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= \frac{\partial}{\partial x_j} \left(-\sum_{i=1}^{K} y_i \log \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \right) \\
&= \frac{\partial}{\partial x_j} \left(-\sum_{i=1}^{K} y_i (\log e^{x_i} - \log \sum_{k=1}^{K} e^{x_k}) \right) \\
&= \frac{\partial}{\partial x_j} \left(\sum_{i=1}^{K} -y_i \log e^{x_i} + \sum_{i=1}^{K} y_i \log \sum_{k=1}^{K} e^{x_k} \right) \\
&= \frac{\partial}{\partial x_j} \left(\sum_{i=1}^{K} -y_i x_i + \sum_{i=1}^{K} y_i \log \sum_{k=1}^{K} e^{x_k} \right)
\end{aligned}$$

The next step is a little bit more subtle, but recall there is only a single true label for each example and therefore only a single y_i is equal to 1; all others are 0. Therefore we can imagine expanding the sum over i in the second term and only one term of this sum will be 1 and all the others will be zero.

$$\frac{\partial L(\hat{y}, y)}{\partial x_j} = \frac{\partial}{\partial x_j} \left(\sum_{i=1}^{K} -y_i x_i + \log \sum_{k=1}^{K} e^{x_k} \right)$$

Now we take the partial derivative and remember that the derivative of a sum is the sum of the derivative of its terms and that any term without x_j can be discarded.

$$
\begin{aligned}
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= \frac{\partial}{\partial x_j}(-y_j x_j) + \frac{\partial}{\partial x_j} \log \sum_{k=1}^{K} e^{x_k} \\
&= -y_j + \frac{1}{\sum_{k=1}^{K} e^{x_k}} \cdot \left(\frac{\partial}{\partial x_j} \sum_{k=1}^{K} e^{x_k} \right) \\
&= -y_j + \frac{1}{\sum_{k=1}^{K} e^{x_k}} \cdot (e^{x_j}) \\
&= -y_j + \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}
\end{aligned}
$$

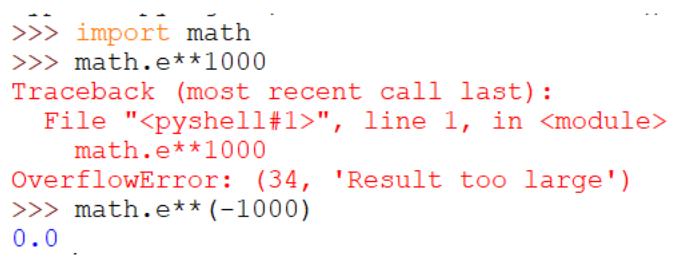
That second term looks very familiar, huh?

$$
\begin{aligned}
\frac{\partial L(\hat{y}, y)}{\partial x_j} &= -y_j + \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \\
&= -y_j + \sigma(x_j) \\
&= \sigma(x_j) - y_j \\
&= \hat{y}_j - y_j
\end{aligned}
$$

After all that work we end up with a very simple and elegant expression for the derivative of softmax with cross-entropy divergence with respect to its input. What this is telling us is that when $y_j = 1$, the gradient is negative and thus the opposite direction of the gradient is positive: it is telling us to increase the probability mass of that specific output through the softmax.

C LogSumExp

The LogSumExp trick is used to prevent numerical underflow and overflow which can occur when the exponent is very large or very small. For example, look at the results of trying to exponentiate in python shown in the image below:



```
>>> import math
>>> math.e**1000
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    math.e**1000
OverflowError: (34, 'Result too large')
>>> math.e**(-1000)
0.0
```

As you can see, for exponents that are too large, python throws an overflow error, and for exponents that are too small, it rounds down to zero.

We can avoid these errors by using the LogSumExp trick:

$$\log \sum_{i=1}^{n} e^{x_i} = a + \log \sum_{i=1}^{n} e^{x_i - a}$$

You can read more about the derivation of the equivalence here and here

Glossary

Activation function:

An activation function defines a threshold which specifies when a given neuron in our network fires, i.e. what combinations of weights and biases lead the neuron to output a signal. They are modelled based on activation potentials in the brain, which mean neurons only ”fire” when the signals are above the threshold.

Auto-differentiation libraries:

Libraries such as PyTorch and Tensorflow which allow internal calculations of partial derivatives.

A comprehensive explanation of auto-differentiation can be found here

Bias:

Biases act the same way as weights in a neural network, except unlike weights, biases are not dependent on activity, but rather are always ”on”.

Backpropagation:

Short for ”backwards propagation”, backpropagation is an algorithm for learning using gradient descent. Backpropagation is a specific type of autodifferentiation.

More details can be found here

Forward:

The forward pass of a neural network refers to the calculations of outputs, and the comparison of these to the desired outputs (in supervised learning)

Gradient descent:

Gradient descent is an optimization algorithm. Optimizers are introduced on the supplementary page, but you'll have to wait for class to go more in depth.

Loss:

A loss function measures the difference between the output of the neural network (prediction) and the actual desired output (label). The aim is to reduce the value of the loss function.

Multi-Layer Perceptron (MLP):

An MLP is a network of multiple perceptrons, which, as a combination of linear classifiers can be used to classfy more complex shapes.

Perceptron:

A perceptron is a linear classifier. Meaning it can be used to distinguish which side of a linear function a given point lies.

Weight:

Weights define the strength of the ”connections” between neurons in the network. A weight is a multiplier, and defines how much of a signal is transmitted through the corresponding connection.