Assignment 1: Implementing MLP using Numpy

Due: November 6, 2020

Goal: Implement MLP (aka fully-connected neural network) and its learning algorithm (stochastic gradient descent) in Numpy.

Submission: The assignment consists of two parts: implementation and an analysis. You are supposed to present results for both parts in the following manner:

1. Upload your code.
2. Prepare a report with an analysis of results obtained.

The code and the report must be uploaded due to the deadline to Canvas.
UPLOAD A **SINGLE FILE** (a zip file) containing your code and the report. Name your file as follows: [vunetid]_[assignment number].

changelog

4 Nov: Add note on initialization in Q3.
3 Nov: Add note on normalization.
31 Oct: Add use of random package.
30 Oct: Clarify page limit and report structure. Fix phrasing in final part.
28 Oct: Add changelog. Clarify base of logarithm.
27 Oct: Clarified requirements in Part 1. Fixed gradients given at the end of part 1.
26 Oct: Published.

Introduction

In this assignment we are going to implement backpropagation for a simple feed-forward network, with a cross-entropy loss function. We will first implement it entirely from scratch, using only basic python primitives. Then, we will vectorize the forward and backward passes using numpy.

The first three parts of this exercise consist of simple exercises. The final part is a small experimental investigation. You should hand in a small research report presenting your findings of part 4. The answers to the questions can be placed throughout your report, or collected in a special section.[1] If you do the former, please make sure that the answers are clearly marked, so that the TAs can easily find them.

We suggest a length of around 2-4 pages for the report. The length is ultimately up to you, but please go easy on the TAs.

Preliminaries:

- Make sure you can do simple python programming. If not, https://www.learnpython.org/ is good place to quickly brush up.
- Make sure you have a working knowledge of numpy. Follow this notebook to brush up.
- Make sure you've watched the videos or read the slides for the first and second lectures.
- You can do the whole assignment within a single python script, but you may want to work in a notebook environment so you can more easily see what's going on, and plot particular values. It's up to you.

Part 1: Working out the local gradients.

The slides provide derivations of most of the parts of a feedforward network. With two exceptions

- The softmax activation.
- The cross entropy loss.

Call the linear (non-activated) outputs of the network o_i and call the corresponding softmax-activated nodes y_i (where i ranges over the number of classes). For a given instance x with a true class c, we then have

$$y_i = \frac{\exp o_i}{\sum_j \exp o_j}$$

$$l = -\log y_c$$

where the log is base e.

Question 1. Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.

[1] Please use a section of the main report rather than an appendix.

Tips: We're looking for the derivatives $\frac{\partial l}{\partial y_i}$ and $\frac{\partial y_i}{\partial o_j}$. Note that because of the sum in the softmax formula, y_i depends on all o_j, not just on o_i. It may be helpful to work out $\frac{\partial y_i}{\partial o_i}$ and $\frac{\partial y_i}{\partial o_j}$ separately (where in the last case $i \neq j$).

Note also that the true class c is given as an integer, so it may be helpful to write the loss as:

$$\text{loss} = \sum_i l_i$$

$$l_i = \begin{cases} \ldots & \text{if } c = i \\ 0 & \text{otherwise.} \end{cases}$$

In the simple example network in the slides, we didn't need the *multivariate* chain rule. It may be helpful in this part, and it's required in the network coming up.

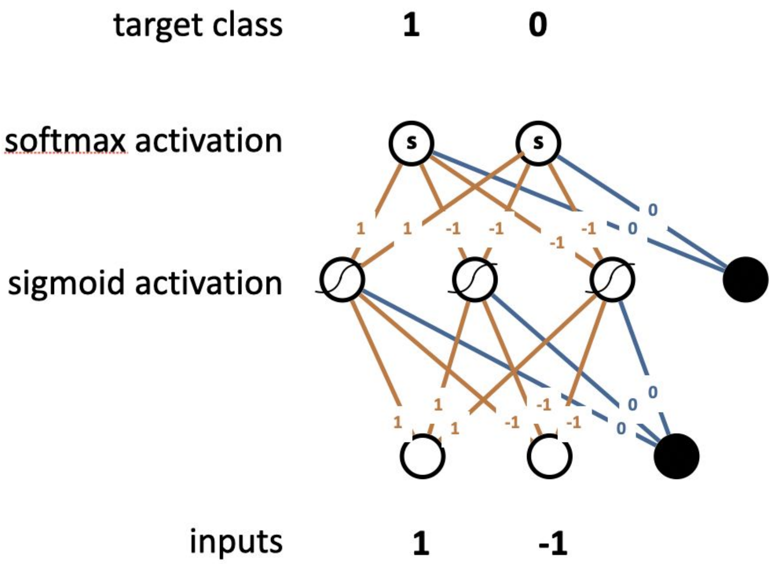
If your calculus is a little rusty, have a look at the suggested reading on Canvas, for some resources to help you catch up. We don't recommend taking shortcuts here, since there's much more calculus coming up.

Bonus task: Work out the derivative $\frac{\partial l}{\partial o_i}$. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?

Part 2: Scalar backpropagation

We now have everything we need to implement a neural network. The slides of the "scalar backpropagation" video give you some pseudocode to take inspiration from.

Question 2. Implement the network in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights W, b , V and c. Do not use anything more than plain python and the math package.



Tips: You'll need to break up this diagram into more fine-grained modules (for instance, the first layer linear outputs k_i and their sigmoid activated versions h_i). Then write down the symbolic definition of all these modules, and their local derivatives. Most of this is already done in the slides. The two missing modules, you've worked out in the first part.

It's easiest to represent each layer as a list containing float values, like so:

```
	k = [0., 0., 0.]
```

and the weights as lists of lists:

```
	w = [[1., 1., 1.], [-1., -1., -1.]]
```

You should get the following values for your derivatives of the loss wrt to the parameters:

```
derivatives wrt W, b:
   [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]    [0.0, 0.0, 0.0]
derivatives wrt V, c:
   [[-0.4404, 0.4404], [-0.4404, 0.4404], [-0.4404, 0.4404]]   [-0.5, 0.5]
```

Training on a dataset

Download the following code and add it to your own

https://gist.github.com/pbloem/bd8348d58251872d9ca10de4816945e4

Note the two functions `load_synth(...)` and `load_mnist(...)`. We will use the first now, and the second in the next part.

Question 3. Load the synthetic data. Implement a training loop for your network and show that the loss drops as training progresses.

Some tips:

- How you initialize the weights is an important choice. For now, you can set the regular weights to some normally distributed random value, and *the bias weights to 0*. We'll delve into this question more in later lectures.
- You can use the python `random` package to generate normally distributed random values, or generate some values online and hardcode them.
- Our data loaders provide the target classes as integer values. It's easiest to work out the derivative in terms of these values directly, but you can also convert them to one-hot vectors as shown in the image.
- You can use stochastic gradient descent, calculating the loss over *one* instance at a time.
- ~~You'll need to keep a low learning rate (like 1e-10) and train for several epochs.~~
- For this dataset, the normal initialization sometimes leads to divergent networks and sometimes to convergent networks. If you initialize your weights to the values given in the image above (for question 2), and you set your learning rate to 1e-4 you should see the error drop to around 0.002 over the first 30 epochs.

Part 3: Tensor backpropagation

In this part, we will *vectorize* the operation of our neural network using `numpy`. The last slide of the "tensor backpropagation" video gives you some pseudocode to build from (but note that we are using a different loss function).

Question 4. Implement a neural network for the MNIST data. Use the following architecture:

```
784 (input) -> Linear(784, 300) -> Sigmoid -> Linear(300, 10) -> Softmax
```

Tips and pointers.

- MNIST is a famous dataset of handwritten digits. It's given to you as just a large vector of numbers. If you'd like to visualize the data (always a good idea), there's some examples in the repository of the data loader we used.
- ~~The inputs for both of our datasets are pretty well normalized. That means that the values are (largely) in a controlled range like [0, 1] or [-1, 1]. This is extremely important for neural networks. If your data is *not* normalized, you should normalize it before you feed it to your network.~~
- **Contrary to earlier statements, the MNIST data is not normalized. You should make sure to normalize your data before you feed it to the neural network.**
- To stabilize your learning in a simple way, you can add up or average the gradients for a few instances before you take a gradient descent step. This is equivalent to minibatch learning, but a bit slower, since it doesn't take advantage of numpy's ability to parallelize over the batch dimension.
- As a **bonus task**, you can work out the vectorized version of a *batched* forward and backward (i.e. multiple images at a time) This will stabilize and speed up your training.

Part 4: Analysis

Question 5. Train the network on MNIST and plot the loss of each batch or instance against the timestep. This is called a *learning curve* or a *loss curve*. You can achieve this easily enough with a library like matplotlib in a jupyter notebook, or you can install a specialized tool like tensorboard. We'll leave that up to you.

If you set the switch `final` on `load_mnist()` to `False` (the default), you will get part of the training set, and the withheld training data will be returned as the test set (i.e. you get a *validation set*). If you set the switch to `True`, you will get the full training data and the canonical test set. Make sure you know the difference.

We will investigate how well this network can learn if we limit it to 5 epochs. Do the following experiments:

1. Compare the training loss per epoch to the validation loss per epoch. What does the difference tell you?
2. Run the SGD method multiple times (at least 3) and plot an average and a standard deviation of the objective value[2] in each iteration (e.g., see : here). What does this tell you?
3. Run the SGD with different learning rates (e.g., 0.0001, 0.01, 0.05). Analyze how the learning rate value influences the final performance.
4. Based on these experiments, choose a final set of hyperparameters, load the full training data with the canonical test set, train your model with the chosen hyperparameters and report the accuracy you get.

Notes:

- If you'd like to try your network on other image datasets, look for data that is similar in size: there's "fashion-mnist", "emnist", "color mnist", "mnist-rot" and so on. You'll get **bonus points** for this, but you must report your results on the original MNIST as well.

[2] Either the loss or the accuracy. You can plot both to give yourself a full picture.