Assignment 1: Implementing an MLP using Numpy

Goal: Implement an MLP (i.e. a fully-connected neural network) and its learning algorithm in plain python and in numpy.

Submission: Write your solutions to each question in a PDF prepared in LaTeX. Include the relevant parts of your code in the report. Make sure everything is neatly typeset. Do not include screenshots of math and code, but include it properly using the latex packages provided.

You are required to use the provided LaTeX template. See the document for some extra instructions.

Changelog

Introduction

In this assignment, we will 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. Collect your answers in a PDF as detailed in the LaTeX template linked above.

Preliminaries:

- Make sure you can do simple python programming. If not, https://www.learnpython.org/ is a 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 l is the loss and the logarithm 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.

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.

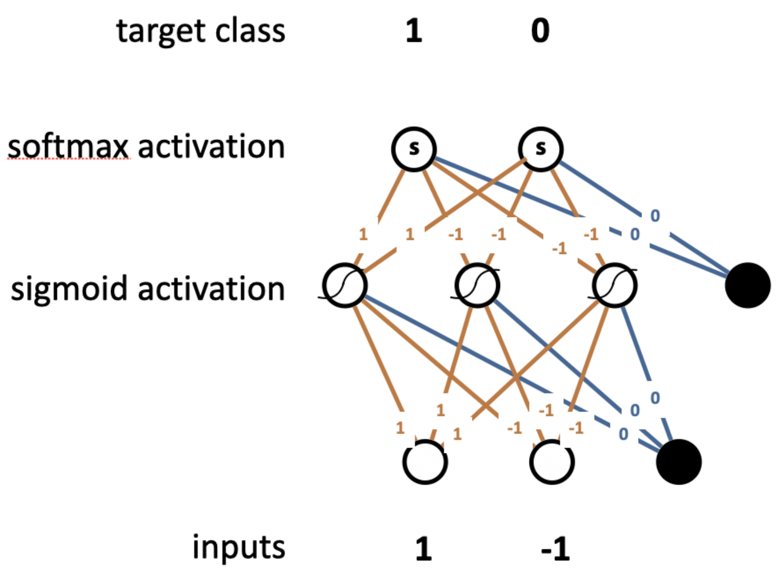
If your calculus is a little rusty, have a look at the suggested reading on Canvas, for some resources to help you catch up. We don't recommend taking shortcuts here, since there's much more calculus coming up.

Question 2*[1]: 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?

1.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 3. Implement the network drawn 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 (including biases). 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:

[1] Starred questions can be skipped if you're in a hurry to meet the deadline, or if you are struggling with the material. You will still score a passing grade without the starred questions, but if you want full marks you should do these questions as well. We suggest doing the assignment without the starred questions first, and then coming back to them if you have time.

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

You should get the following values for your derivatives of the loss wrt to the parameters. For the first layer:

```
derivatives wrt W, b:
   [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]    [0.0, 0.0, 0.0]
```

For the second layer:

```
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.

Load the synthetic data with the following command:

```
    (xtrain, ytrain), (xval, yval), num_cls = load_synth()
```

`xtrain` is a numpy matrix containing the data, and train is a vector containing the labels (as integers.

Question 4. Implement a training loop for your network and show that the training loss drops as training progresses.

Make sure to print some of the data before you start, so see the range of values it gives you. If the values are not in a controlled range like [0, 1] or [-1, 1], you should normalize them first.

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.

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 5. Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10.

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.
- Make sure to check if the data needs to be normalized.
- 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.

Question 6*. Work out the vectorized version of a *batched* forward and backward. That is, work out how you can feed your network a batch of images in a single 2-tensor, and still perform each multiplication by a weight matrix, the addition of a bias vector, computation of the loss, etc. in a single numpy call.

Tip: Numpy's matmul takes care of batched matrix multiplication automatically, but you'll have to study the documentation to make sure that you understand how this works. Try a quick test with matrix batches and multiply them (in a python console of a notebook).

Part 4: Analysis

Question 7. 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. Train the neural network from a random initialization 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., 0.001, 0.003, 0.01, 0.03). Analyze how the learning rate value influences the final performance.
4. Based on these experiments, choose a final set of hyperparameters.[3] Then load the full training data with the canonical test set, train your model with the chosen hyperparameters and report the accuracy you get.

Grading key (10pt)

- question 1: 1pt
- question 2: 1pt
- question 3: 1.5pt
- question 4: 1.5pt
- question 5: 1.5pt
- question 6: 1pt
- question 7: 2.5pt

[2] Either the loss or the accuracy. You can also plot both to give yourself a fuller picture.
[3] In this case the learning rate is the main hyperparameter to try. Feel free to play around with other hyperparameters like the batch size, the number of hidden nodes, or even the type of activation function, but you don't have to.