Homework 01 - Implementing and Training an MLP

IANNwTF

November 2, 2022

Homework Timeframe:

- Starting 1.11.
- Submission: **Sunday November 12, 23:59**

Submit your homework via https://forms.gle/9XFB4ryCCR7zgmfQ9

Notice: HOmework 1 probably represents the most challenging homework in this class, however does not require learning anything about TensorFlow yet! Make use of the QnA and flipped Classroom, which are meant to support your work on this submission directly.

Contents

1 General Information

- Homework is to be submitted by each student
- You are allowed and **recommended** to collaborate with other students on your submission
- You are required to completely understand your submission when collaborating with other students
- You can get feedback for your homework submission in the QnA sessions, however, we are not able to provide general feedback for every submission by default due to tutoring hour constraints
- You can submit an outstanding homework submission. This is generally not expected from you but rewards students going *above and beyond*. Outstanding submissions are required to stand-alone work as sample solutions for other students. This includes (1) the submission providing a competitive solution to the task at hand, (2) being implemented based on the tools recommended, (3) running without warnings and errors, (4) the code being well documented with comments, (5) including content explanations via comments or markdown text blocks for .ipynb submissions, (6) quality visualizations wherever graphs are recommended or required. Finally (7) it is important the submitting student joins both flipped Classroom meetings following the submission date, where they might be asked to present elements from their submission to the class.

Remember to make use of the Coding Support / QnA sessions and the flipped classroom to support your work on this homework assignment

2 Assignment: Multi-Layer Perceptron

In this section, you are implementing a Multi-Layer Perceptron using Numpy (i.e. no deep learning library is allowed!). For this first homework, we are also providing comparatively much detail to help you along.

The following description will walk you through an implementation of an MLP, and implementing Backpropagation for it. You are not required to follow this explanation, as long as you do not make use of only pre-existing Deep Learning packages (e.g. TF, PyTorch, sklearn, JAX, etc.), or automatic differentiation packages. Outstanding submissions should follow the guide in a reasonably close fashion.

2.1 Data

We start with the data, as it helps you understand what to expect for your network: You will work with a comparatively small and simple handwritten digit dataset, which you can load from sklearn:

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html

- The dataset contains small $8x8$ greyscale images of digits as inputs, along with their respective target digits (0-9).
- Extract the data into (input, target) tuples
- Make sure your input images seem correct by plotting them as images respectively!
- Reshape the $8x8$ images into vectors of shape (64) or $(1, 64)$ depending on your preference
- Make sure the images are represented as float32 values within either the [0 to 1] or [-1 to 1] range, if necessary rescale them respectively
- One-hot encode the target digits (e.g. the target digit 2 would be represented as $[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$, 9 as $[0, 0, 0, 0, 0, 0, 0, 0, 0, 9]$). Onehot encoded vectors should have the shape (10) or $(1, 10)$ up to your preference
- Write a generator function, which shuffles the (input, target) pairs (keeping the respective input and target together)
- Adjust your generator function to create minibatches: Combine *minibatchsize* many inputs into a ndarray of shape $minibatch\_size, 64$, and targets into a ndarray of shape $minibatch\_size, 10$ respectively. Make sure you can adjust your minibatchsize as an argument to this generator, and also that respective (input-target) pairs match with respect to their index in the minibatch

2.2 Sigmoid Activation Function

Implement the Sigmoid Activation Function as an object with a *call* function (we will later add a *backwards* function for backpropagation, this being the reason its implemented as an object). It should expect inputs as ndarrays of shape $minibatchsize, num\_units$ (where *num*)

2.3 Softmax activation function

The softmax activation function is used in the final layer of a classification model, **replacing the sigmoid** activation function. It is given by $\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K$, where z_i are the pre-activaitons (MLP activations before applying an activation function), and K is the size of the input vector, in our case 10 (as there are 10 different digits).

Notice how the softmax converts the input vector into a probability distribution, i.e. the input vector is positive at every entry and sums up to one.

Implement the the softmax in Numpy, again as an object with a *call* function, which expects inputs of size $minibatchsize, 10$, where each $1, 10$ subelement represents a vector as discussed above.

2.4 MLP weights

Implement a class representing an MLP layer. The constructor should specify the activation function (Sigmoid or Softmax as implemented above), as well as the number of units (Perceptrons) in this layer, and the input size (number of units in the preceding layer). It should create attributes for the weight matrix and a bias vector (or absorb the bias vector into the weight matrix as discussed in the lecture, depending on your preference). Initialize the weights as small random values (e.g. from a normal distribution with $\mu = 0., \sigma = 0.2$), and bias values set to zero.

Implement a forward function, which accepts an input of shape $minibatchsize, input\_size$, and outputs an ndarray of shape $minibatchsize, num\_units$ after applying the weight matrix, the bias and the activation function.

2.5 Putting together the MLP

Implement a class representing a full MLP. You should be able to specify the number of layers, as well as the size of each layer separately in the constructor. Remember the initial input will be shaped $minibatchsize, 64$, the final output will be shaped $minibatchsize, 10$. The MLP should have an attribute, which is a list of all the MLP layers it contains.

2.6 CCE Loss function

Implement a categorical cross-entropy loss function (see e.g. https://www.google.com/search?channel=fs&client=ubuntu&q=cce+loss+function) for your network. Again, implement it as an object with a call function for the function itself (which again is extended with a backwards function later).

3 Backpropagation

In the following you are implementing Backpropagation. As the first (and most important step) you are extending your existing implementation by implementing the respective backwards functions (which essentially compute the respective Jacobian matrices for optimizing the network).

Since Backpropagation happens from back to front, we will also work through follow this idea here:

3.1 CCE Backwards

Since the last step is Computing the loss, it is the first for which we have to handle. Essentially we have to Implement

$$\frac{\mathrm{d}L_{CCE}}{\mathrm{d}\hat{y}}$$

, where $\hat{y}$ is the prediction, i.e. the output of the last layer after the softmax activation. Try to calculate this yourself, but you can find support (beyond QnA meetings and flipped Classroom) also e.g. here: https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1 (notice they are replacing $\hat{y}$ with s in their notation however).

Implement the backwards function for your CCE object: It expects at least the inputs prediction (our $\hat{y}$) of shape *(minibatch_size,10)*, and loss (our L_{CCE} of shape *(minibatch_size,1)*, i.e. one loss per minibatch element. It should return a respective matrix of shape *(minibatch_size,1)* representing the error signal (think in Leibnitz notation: The derivative calculated up to this step).

3.2 Sigmoid Backwards

Implement the backwards function for the sigmoid. It calculates

$$\frac{\mathrm{d}L}{\mathrm{d}pre\_activation} = \frac{\mathrm{d}activation}{\mathrm{d}pre\_activation}\frac{\mathrm{d}L}{\mathrm{d}activation}$$

and applies it to the error signal. The inputs for this function should include the preactivation (shape *(minibatch_size,num_units)*), the activation, and the error signal (i.e. $\frac{\mathrm{d}L}{\mathrm{d}activation}$). Notice how the Jacobian for this step is a diagonal matrix (i.e. all entries not on the diagonal are zero!), so it might be easier for you not to explicitly calculate the Jacobian.

3.3 MLP Layer Weights backwards

Notice how the MLP layer requires three backpropagation steps:

1. From layer output (i.e. activation) to the pre-activation, which we already got (these are the activation function backwards steps above!).
2. From pre-activation to derivative w.r.t. weights, i.e. $\frac{\mathrm{d}L}{\mathrm{d}W}$ where W is the weight matrix of this layer
3. From pre-activation to derivative of the input of this layer (required, as it is the error signal for the layer from which we get this input!), i.e. $\frac{\mathrm{d}L}{\mathrm{d}input}$

The weights_backward step should be a function of your MLP layer, with inputs including the error signal so far $\frac{\mathrm{d}L}{\mathrm{d}pre\_activation}$, as well as the pre-activations.

It should return $\frac{\mathrm{d}L}{\mathrm{d}W}$ and $\frac{\mathrm{d}L}{\mathrm{d}input}$ (both for the respective layer of course), calculated as:

- $\frac{\mathrm{d}L}{\mathrm{d}W} = \frac{\mathrm{d}L}{\mathrm{d}pre\_activations}\frac{\mathrm{d}pre\_activations}{\mathrm{d}W}$
- $\frac{\mathrm{d}L}{\mathrm{d}input} = \frac{\mathrm{d}L}{\mathrm{d}pre\_activations}\frac{\mathrm{d}pre\_activations}{\mathrm{d}input}$

3.4 MLP Layer backwards

The MLP backwards simply combines the activation backwards and weights_backwards steps for the MLP layer object

3.5 Gradient Tape and MLP backward

Create a MLP backward function, which combines the CCE Loss backwards and the backwards function of the included MLP layers.

In this full MLP (full network) backward function, it is recommended to create a list of empty dictionaries with equal length as your respective number of layers in the MLP. During the forward pass, include all the activations and pre_activations at the respective index in the respective dictionary. For the backward step, also include the weight gradients in the respective dictionary at the respective index. You only need the error signal to compute these, there is no further reason to keep it saved anywhere. Finally use the weight gradients to update the weights of all your included MLP layers

3.6 Training

Create a training function, which for multiple epochs iterates your mini batched training data set, and trains the network based on the code you have written so far. Plot the average loss vs. the epoch number.

For outstanding submissions, also calculate the accuracy (percentage of the maximum of the prediction equaling the target) at each epoch and include it in this plot.