Report - Project 01 - Gradient Descent

An explanation of your approach and design choices to help us understand how your particular implementation works

Backpropagation was implemented by following the formulas given in the project. Here we first started with the output lay which, vectorized, can be computed similarly to the formula. We then generalize the layers with a for loop, even though we only have one more layer to compute the gradients for. We don't have to care for the summation when computing delta but implement it equal to the formula. When vectorizing we had to make sure the dimensions of the matrices were right. Therefore we had to understand the difference between matrix multiplication and product wise binary operators. In the implementation we use them to achieve the right dimensions. Also we had to squeeze() the bias since pytorch expects it to be of dim(n^[l]). Following the formulas keeps the computational graph intact and is generalized to also work for the mnist example.

Gradient Descent part of the project.

We started by creating the load_cifar method. As this had been done in an earlier weekly exercise, this was trivial. We transform the datasets by using the PyTorch - transpose.Compose function. This allows us to both normalize and turn our objects into a tensor object.

Data is further split on a 90/10 ratio, where 90 percent is for training and 10 percent is for validation. We randomly split the train/validation data to not have any corruption of data.

The task requires only data of airplanes and birds, as this the prediction goal of our classifier model. Again, this was done in weekly exercise 03, so the implementation is very similar. Check code for comments. The end result is a train, validation and test set - containing tensor objects of airplanes and birds with their respective label.

With that out of the way, our approach and design can be more accurately described. We created the MyMLP class, with three hidden units, which means four layers respectively. This was in order to have the correct amount of hidden layers: 512, 128, 32 and 2 output dimensions should be 2. In our implementation that means fc1 layer to fc5 due to the input dimension. The model uses ReLU activation function for each layer (not the last layer - of course).

The train_manual_update function works for all of the given tasks. Our implementation supports you to test for both “naked” train functions without any optimizer parameters. You will find the function to support both momentum and weight decay also.

Finally, to visualize the results, we took inspiration for weekly exercise 06 for a graph representation of how the Cross Entropy Loss is for every model with different hyperparameters. The output experience will be discussed further down.

Which PyTorch method(s) corresponds to the tasks described in section 2?

loss.backwards(). Where loss = `loss_fn(prediction, labels)`

Cite a method used to check whether the computed gradient of a function seems correct. Briefly explain how you would use this method to check your computed gradients in section 2.

Using loss.backwards() then printing the gradients computed with autograd to see if they’re equal. Where loss = `loss_fn(prediction, labels)`

Which PyTorch method(s) correspond to the tasks described in section (3), question 4.

SGD.Optimizer(). Where we manually write the optimizing steps of the weights and biases using L2 regularization and momentum for stochastic gradient descent.

Briefly explain the purpose of adding momentum to the gradient descent algorithm

As Gradient Descent Algorithm is an algorithm in which we use to optimize the gradient of an objective function, momentum helps us speed up or accelerate this process. Momentum helps to speed up the gradients vectors in a “right” direction (space minimufm).

Briefly explain the purpose of adding regularization to the gradient descent algorithm.

Regularization is a concept we apply in order to penalize a cost on the optimization function. This can be applied to prevent overfitting and find a more optimal funcion. In this project we applied L2 - or ridge regression. This shrinks the size of all coefficients. Larger coefficients will be more penalized. In a more specific term, regularization tries to reduce variance while at the same time not over increasing the bias.

Report the different parameters used in section (3) question (8), the selected parameters in question (9), as well as the evaluation of your selected model.

Five models.

First model parameters = lr=1e-2, weight_decay = 0.0, momentum = 0.0, epoch=60
Second model parameters = lr = 1e-2, weight_decay = 1e-5, momentum =0.6, epoch=60
Third model parameters = lr = 1e-2, weight_decay = 1e-2, momentum=0.66, epoch=60
Fourth model parameters = lr=1e-1, weight_decay = 1e-1, momentum=0.8, epoch = 30

See selected model explanation below.

Comment your results. In case you do not get expected results, try to give potential reasons that would explain why your code does not work and/or your results differ.

The first instance is just a regular SGD without any momentum or regularization. Here we see that we get a pretty slow start with lots of potential for learning more from the training loss of 0.65 after 60 epochs. The accuracy is ~75% . Not really that good so let's see what happens when we add momentum and weight decay (regularization).

The second instance has "default" values for all the parameters which yields an accuracy of ~81% with training loss of 0.43. Here we still see potential to learn more though the curve for 60 epochs seems to show a steady rate of acquiring learning. Important here is that the instance does not seem to reach a plateau meaning that if we had trained it for longer it would have had a higher accuracy. The problem with the instance is that it has high variance between the training and validation set.

The next model therefore is a tweak of the momentum and weight decay to regularize more while trying to lower the bias. It archives an accuracy of ~82% which for 60 epochs, on the TEST data (shown in the last cell), is decent but not good at all but in terms of a neural network it's not overfitted making it better than the rest. Though for our small project we choose this as the best one which is displayed at the end. The reason why will be explored now.

So for the fourth instance we test if we can achieve the same bias with less epochs. For the model with parameters epochs 30, lr = 1e-1, mom=0.8, w=1e-1, the intention was to gain as low of a bias as possible, while maintaining a low variance. We get the resulting training accuracy of 0.793 and the validation accuracy of 0.793. With training loss of 0.505. Here we get a low variance but a high bias with lots of learning potential for the training loss, meaning that we should train for longer, change the architecture or increase the network size. We tried to train another instance with 300 epochs, while tuning the momentum, weight decay and learning rate but could only achieve an accuracy of ~81% with low variance and ~0.43 remaining training loss. This instance is NOT amongst the submitted ones, but the conclusion from the results is that we'd need a larger network to encapsulate more of the nuances to describe the difference between birds and planes. Or change the model architecture to achieve a lower bias.