Report INF265

Necessary libraries:

To run our code these libraries need to be installed, pytorch, numpy, plotly and pandas. Plotly and pandas are used to create the plots both here in the report and inside the Gradien_decient.ipynb file.

Division of labor:

Both worked on Backpropagation and report.

Christopher created train_manual_update()

Jonas Land fine tuned the code and created plots for relevant results.

Task 1.

Backpropagation:

The purpose of backpropagation is to compute the gradient of the loss function with respect to the parameters of the model. This is then used to update the parameters of the network/model.

We implemented backpropagation in a vectorized manner, with only one for loop to iterate over all the layers. We start by calculating dl_da as the derivative of the loss function with y_true and y_pred as inputs. Inside this for loop we calculate dL/dz and dL/da to update dL_dw and dL_db for each layer.

Task 2:

Gradient Descent:

To Load, analyze and preprocess the CIFAR-10 dataset, splitting and creating CIFAR-2 was already done in a previous weekly assignment so we used that code. We added a function to calculate the mean and standard deviation of the dataset for the normalization step. We then split the dataset into 75% training, 8,34% validation and 16,66% test datasets for CIFAR10, then took all pictures from cifar10_train with labels bird or plane into cifar2_train, and so on for validation and test. Resulting in a split of 75.14% for training, 8.19% for validation and 16.66% for test. There was a slightly different split probably due to a slight uneven split for bird and plane pictures in in cifar10_train/val/test.

When we implemented MyMLP we mostly used a previous weekly exercise where we only had to change the activation function to be ReLU for all the layers and change the dimensions of the input hidden and output layer(s) to match the tasks requirements (512, 128 and 32).

Implementing the “train()” function was also not necessary as this was done in a previous weekly assignment.

We added several Boolean inputs to “train_manual_update()” so our function would work for inputs with and without momentum, weight_decay and early stopping. We used if statements to determine what combinations of these were used and update the parameters accordingly. Doing it this way made it easier to perform model selection at the end. We did this by calling our function with hyperparameters in nested for loops and chose the model with the highest validation accuracy. Then we tested that model on our test dataset. We tried 4 different values for weight decay, momentum, and learning rate. We also tried with and without early stopping, so in total we trained and tested 128 models.

2. a)

There are three functions in PyTorch that correspond to tasks described in section 2. Optimizer.zerograd() which is used to clear gradiants of all optimized tensors to prepare for the next iteration. Loss.backward() which computes gradients of the loss function with respect to the model parameters. Optimizer.step() which updates the parameters of the model based on the computed gradients.

b)

We could do the same as the tests in backpropagation.ipynb. We would compute the gradient and compare it to autograds computation with a small tolerance for numerical precision difference. Check if the weights have been updated. Compare our results with a numerical technique for approximating derivatives. If all of these conditions pass then we can be pretty certain that our implementation is correct.

c)

optim.SGD(model_seq.parameters() is a method that takes different parameters such as learning rate, momentum and weight decay. This defines a stochastic gradient descent for the neural network. Which matches what we did manually in task 3,4. There are also other optimizers from pytorch such as optim.adam() and optim.Adagrad() which are more complex optimizers.

d)

Adding momentum to the algorithm almost always makes it more efficient. As this will make it take larger steps towards the goal. It is also more likely that it will overshoot smaller local minima, making it more likely to find large local minima or the global minima.

e)

Adding regularization punishes the model when it is about too overfit. It does this by adding a penalty for large weights or overly complex models.

f)

We trained 64 models with early stopping. Four parameters for weight decay, 0, 0.01, 0.1, 0.5. Four parameters for momentum, 0, 0.01,0.1, 0.8. Four parameters for learning rate, .1, 0.01,0.001,0.0001. Every combination of these parameters is being tested and validation accuracy is being saved. We also implemented early stopping where it will automatically stop training if validation loss increases 2 times in a row. This is done to avoid overfitting and to find the best number of epochs for each parameter combination.

We selected the model with the highest validation accuracy and tested this model using our test dataset. The model that performed the best on validation data had these hyperparameters, learning rate = 0,1, weight decay = 0, momentum = 0 and it got a test accuracy of 0.86.

g)

This is a respectable result as it can be difficult to spot the difference between a plane and a bird. The fact that weight decay and momentum was 0 for our best model seems a bit strange. But we also implemented early stopping, which is another form of regularization and might have made weight decay and momentum less effective. To test this, we ran all 64 models again without early stopping with 21 epochs. This resulted in the best model having weight decay = 0,01 momentum = 0 and a test accuracy of 0,85. This makes sense as when early stopping was off the model benefitted from some regularization.

Model: 18

Hyperparameters:

Learning rate:	0.01
Weight decay:	0.01
Momentum:	0
Early stopping:	Off

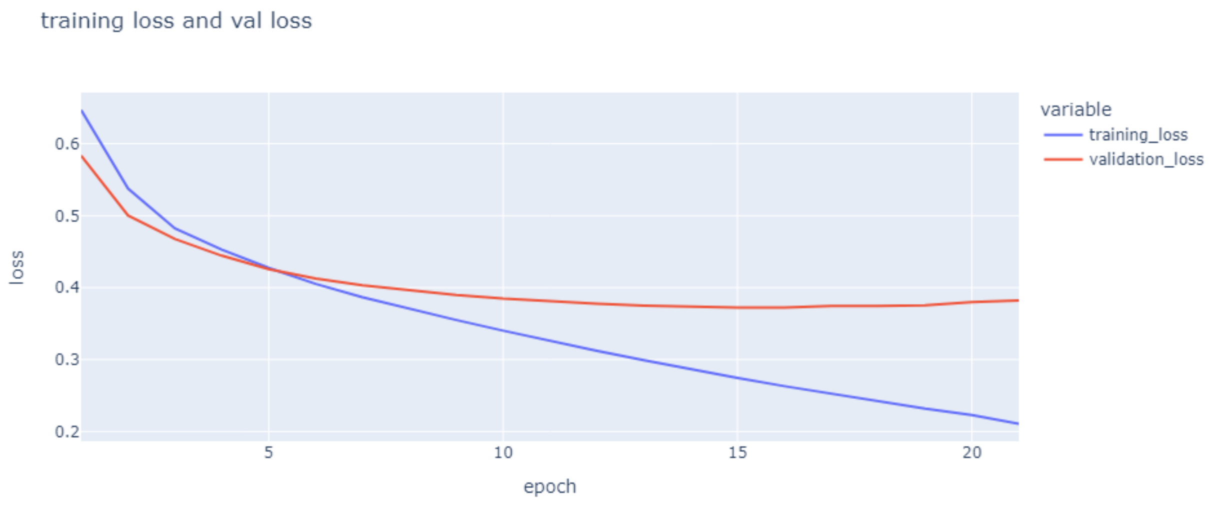

FIGURE 1: TRAINING AND VALIDATION LOSS PER EPOCH WITHOUT EARLY STOPPING

Performance:

Training accuracy:	0.94
Validation accuracy:	0.85
Test accuracy:	0.85

Model: 2

Hyperparameters:

Learning rate: 0.01

Weight decay: 0

Momentum: 0

Early stopping: On

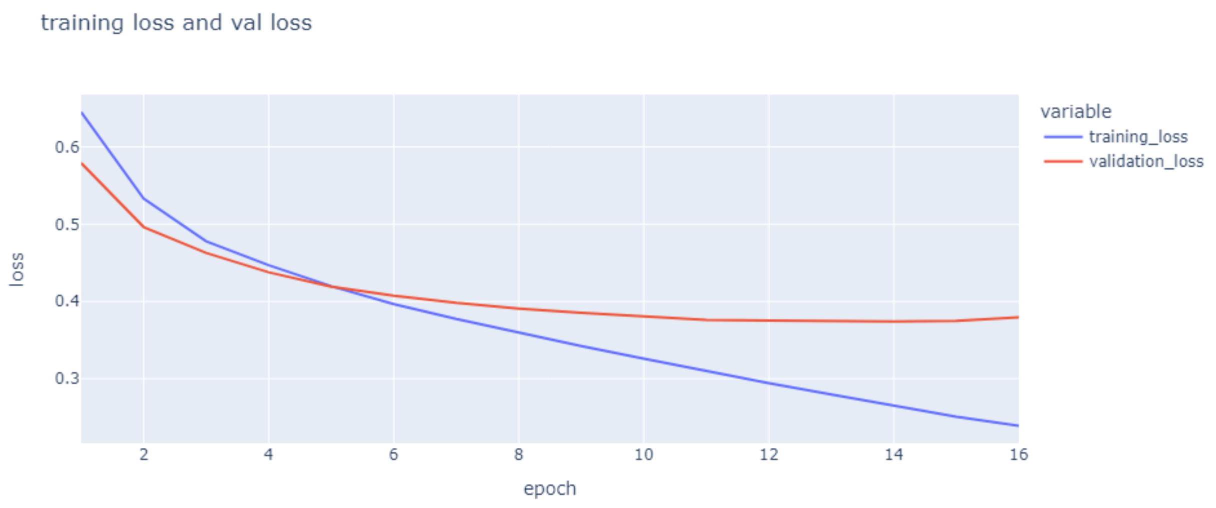

FIGURE 2: TRAINING AND VALIDATION LOSS PER EPOCH WITH EARLY STOPPING

Performance:

Training accuracy: 0.93

Validation accuracy: 0.85

Test accuracy: 0.86

These are the two models that performed best on validation data. Looking at the difference between train and validation accuracy, the models does not seem too overfit or underfit. We tested the model on the test set as well, and the result were somewhat the same as validation accuracy.

The code seem quite efficient considering it trained 128 models in approximately 2 hours and 10 minutes.

NB: As stated above we trained 128 different models, you can find all the results on validation data at the bottom of our gradient_descent.ipyb file.