# Explanation of approach and design

## Backpropagation

In the backpropagation method we iterate through the network backwards starting from the output-layer until we reach the first hidden layer while keeping track of the local gradient. For each iteration we update the weights and biases for layer .

### Calculating gradient:

For the output layer we compute the gradient using:

with

When we update the gradient for the for the hidden layers we use:

### Calculating Bias

For calculating the bias we simply use:

## Gradient Descent

### Loading dataset

TODO

### MyMLP class

MyMLP class has been implemented to fit the structure given to us in 3.1.2. The input dimension is 3 072 , the hidden layers has respectively 512, 128, 32 hidden units with ReLU as activation function on all layers with exception of the output layer.

### Training Function

TODO

### Manual Training Function

The manual training function have been implemented with the same structure as we find in the original Training function with exception of the optimizer step.

### Learning step

First step is wrapping the

For each image and label we go through the weights and biases.

# Questions

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

* In section 2 we implement a backpropagation algorithm that computes the gradient of the output with regards to the input to update the weights. In PyTorch this is done with the loss.backward().

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

* We could use torch.autograd.gradcheck to check if a computed gradient of a functions seems correct. Using gradcheck we could input out function and inputs to check if the computed gradients in section 2 were correct.

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

* In section 3 we implemented optimizer step that multiplies the learning rate with the gradient and updates the weights. In PyTorch this is done with optimizer.step()

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

* The purpose of momentum is to reduce the oscillations of gradient descent. By using momentum we’re able to constantly against the local minimum without just depending on the current result but also previous results that has pushed us in a certain direction.

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

* The purpose of adding regularization to the gradient descent algorithm is that we’re trying to avoid overfitting on the dataset. With regularization we regularize the incoming data so that our models are not affected be extreme values that might occur.

#### 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.

**Global parameters:**

Batch\_size = 256

N\_epochs = 100

Loss\_fn = nn.CrossEntropyLoss()

Seed = 256

**Model based parameters:**

Model 1: lr=0.01, momentum=0, decay=0

Model 2: lr=0.01, momentum=0, decay=0.01

Model 3: lr=0.01, momentum=0.9, decay=0

Model 4: lr=0.01, momentum=0,.9 decay=0.01

#### 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.

* From this we got that model 2 was the best with training Accuracy: 0.98, and validation Accuracy: 0.85. From the plot below we see that the instances where we add momentum is very unstable. All of the models overfit, but instance 2 is overfits least, it also has the best validation accuracy.

