1 Code explanations and design choices (Part 1)

NOTE: I (Kristian Mølbach Lian) did this course last year. Seeing as the assignment is pretty much the same, some code is changed, but is mostly the same. Further, I follow the notation that Andrew Ng uses.

1.1 Backpropagation

For the general backpropagation code, there are *two* cases to consider: The **output layer**, and the **hidden layers**. Doing the calculus of backpropagation, we arrive at an expression for each of the gradient's components. For simplicity of notation, we define a *local gradient* $\delta_i^{[l]}$ that is to be multiplied by $a_j^{[l-1]}$. This in turn gives a general description of the derivatives of the loss-function L with respect to their weights w.

$$\frac{\partial L}{\partial w_{i,j}^{[l]}} = \delta_i^{[l]} \times a_j^{[l-1]} \tag{1}$$

For the biases, it is simply:

$$\frac{\partial L}{\partial b_j^{[l]}} = \delta_i^{[l]} \times 1 \tag{2}$$

Hidden layers: Following the backpropagation-formulas, we can write a general code for the hidden layers. The *local gradients* for these hidden layers are:

$$\delta_i^{[l]} = \frac{\partial L}{\partial a_i^{[l]}} \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = \left(\sum_{k=1}^{n^{[l+1]}} \frac{\partial L}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial a_i^{[l]}} \right) \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}, \qquad l \in [1, \ldots, L-1] \tag{3}$$

If we now write this including the derivative of the activation functions in the l-th layer, we get:

$$\delta_i^{[l]} = \sum_{k=1}^{n^{[l+1]}} \delta_k^{[l+1]} w_{k,i}^{[l+1]} \times f_i^{'[l]}(z_i^{[l]}), \qquad l \in [1, \ldots, L-1] \tag{4}$$

Output layer: This very special case is dependent on what is chosen to be the loss function for the parameters. The local gradient for the output layer L is then:

$$\delta_i^{[L]} = \frac{\partial L}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial z_i^{[L]}} = e_i'(y_i, \hat{y}_i) \times f_i^{'[l]}(z_i^{[L]}), \qquad e_i'(y_i, \hat{y}_i) = -2 \times (y_i - \hat{y}_i) \tag{5}$$

Notice that the function $e_i(y_i, \hat{y}_i)$ comes from differentiating the term inside the sum in the loss function (we set the amount of training examples $m = 1$ in this formula for simplicity):

$$L(\theta) = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} e_i(y_i, \hat{y}_i) \tag{6}$$

Design: For the design of the code, this idea of sectioning the hidden layers from the output layers is a deliberate choice. The hidden layers are calculated in an appropriate

for-loop, where the loop steps backward in the network. The output layer is calculated outside this loop, where the *local gradient* is just slightly different from what it is defined to be in the hidden layers.

An argument can be made to have an if- or while-statement inside the loop, but these are nuances for the scope of this implementation. Also, following the hint, we include a `with torch.no_grad():`, although it runs fine without it.

```python
def backpropagation(model, y_true, y_pred):

    # Wrapping the backpropagation algorithm in a no_grad() torch-context manager
    with torch.no_grad():
        L = model.L

        # Derivatives for the output layer
        # ================================
        # Formula corresponding to formula (5) in the code explanations
        local_grad = -2*(y_true-y_pred) * model.df[L](model.z[L].data).data
        # The weight-derivatives between layer L and L-1 are calculated
        model.dL_dw[L] = local_grad.T * model.a[L-1].data
        # The bias-derivatives for the output layer is just local_grad itself.
        model.dL_db[L] = local_grad[0]

        # Derivatives for the hidden layers
        # =================================
        for l in range(L-1, 0,-1):

            # The (l+1)-th local gradient is dependent on previous ones, so the
            # local gradients are rewritten for each loop.
            previous_grad = local_grad

            # Inside the summation (formula (4)), the summation index is k.
            chain_rule_sum = torch.sum(previous_grad.T * model.fc[str(l+1)].weight.data, dim=0, keepdim=True)

            # Multiplying with the derivative of the activation function.
            local_grad = chain_rule_sum * model.df[l](model.z[l].data)

            # Like before, we calculate the weight- and bias-derivatives
            model.dL_dw[l] = local_grad.T * model.a[l-1].data
            model.dL_db[l] = local_grad[0]
    return None
```

Figure 1: Source code of the backpropagation-function.

1.2 Gradient Descent

1.2.1 Task 3.1.1

The CIFAR-10 dataset is downloaded and preprocessed differently from last year. Last year, the mean and std values were based on the entire CIFAR-10 dataset, whereas now it they are based on exactly the training data, namely CIFAR-2. The dataset has followingly been normalized for faster training. The code can be found in the segment named **1. Loading data**.

1.2.2 Task 3.1.2

The init-method from `nn.Module` is inherited, and the layers and activation functions are set. Furthermore, the forward-propagation for `MyMLP` is defined in the method `forward(self, x)`. Setting the network architecture could have been done in a for-loop, but since this is a very small network the attributes were set manually for a visually understandable code. The code can be found in the segment named **2. Defining a Multi-Layer Perceptron in PyTorch**.

1.2.3 Task 3.1.3

The `train()` function is inspired by the notebook from *Week 06 - Machine learning pipeline and MLP*. Note that one for-loop defines one *epoch* of the training data, whereas another for-loop loops over each mini-batch specified by the `train_loader`. These loops cannot be vectorized because the purpose of the content in the loop is of an iterative nature. The code can be found in the segment named **3. train()-function**.

1.2.4 Task 3.1.4

For this function, there are minor changes to the code, but it ultimately follows the same structure as the `train()`-function from the previous task. The major difference here is the weight update, following formula (3) from the PROJECT1-description. We also follow the hints from section 3.2 for the gradient descent calculation. The code can be found in the segmend named **4. Manual training function**.

1.2.5 Task 3.1.5

In this section, we train the two models, and hope that they pose exactly the same results. The result of it is in the notebook, and shows exactly the same training losses. This verifies that the two functions produce the same models after training. Note that for reproducibility, a randomizer seed is set before each instantiation of `model = MyMLP()`. The code can be found in the segment named **5. Training 2 instances of MyMLP**.

IMPORTANT NOTE: *For structure in the answers, we have decided to make new train-functions that adds on things like L2-regularization and momentum, as separate train-functions with their own fitting names. This is done instead of going back to previously written train-functions, and then updating them.*

1.2.6 Task 3.1.6

As the note suggests, a new function has been written: `train_manual_update_with_L2()`. This pretty much follows the same structure as the `train()`-function, but now a weight decay parameter can be adjusted. Beware that the term $\lambda/2$ has been substituted by `weight_decay` instead. A comparison has been done between the SGD-optimizer from PyTorch, and our manual `train_manual_update_with_L2()`-function. This comparison reveals the *same* training loss. The code can be found in the segment named **6. Adding regularization/weight decay to the manual train function**.

1.2.7 Task 3.1.7

We now add the momentum for the gradient descent. Note that the $(1 - \beta)$ term (from Andrews videos) has been omitted (in the code, `momentum_coeff` is β). When this was decided upon, the model gave exactly the same results as the SGD-optimizer from PyTorch. Omitting this was also noted by Andrew in the videos.

A thing to mention is the storing of the gradients being done in the code. For the first if- and else statements, this is to store the first gradients of the first mini-batch. We are not doing bias-correction here, so the training for the first epoch is noticeably slower than

other epochs. This is because the momentum calculation needs to iteratively build upon on previous gradients to reveal its significance. So these if- and else statements are needed to store the gradients of the first mini-batch, so that the momentum-calculation can build upon this. The code can be found in the segment named **7. Adding momentum to the manual training function**.

1.2.8 Task 3.1.8

Following the recommendation of random search (recommended from Andrew's videos), the hyperparameters for the *learning rate*, *weight decay* and the *momentum coefficient* are randomly set. Note that a seed has been set for reproducibility of these hyperparameters. The different models are also named, such that they can be evaluated with the validation and test data. The code can be found in the segment named **8. Training multiple instances with different parameters**.

1.2.9 Task 3.1.9

For this task, the code from *Week 06 - Machine learning pipeline and MLP* has been pulled to select a model based on best validation accuracy. This model with the best validation accuracy is then the choice for a final model, where this model will be evaluated by the next section of code with the test data. The code can be found in the segment named **9. Selecting the best model**.

1.2.10 Task 3.1.10

The best model selected from the previous section is being put to the test with the test data. As the general rule of thumb states in machine learning, this test data is *only* used for this final model. The code can be found in the segment named **10. Evaluating the best model**.

2 Answers to the questions (Part 2)

2.1 a)

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

Answer: The PyTorch method corresponding to our backpropagation function in section 2 is the `backward()`-method; they compute the gradient $\nabla L(\theta)$ of the loss-function.

2.2 b)

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.

Answer: To see if the computed gradients seem correct, we can compare the analytical gradients computed with the chain rule, and a numerically computed gradients with the

central difference scheme for each layer in the pipeline. Given J is the cost function and $\theta^{[l]} = \left(W^{[l]}, b^{[l]}\right)$, then:

$$d\theta^{[i]}_{approx} = \frac{J\left(\theta^{[1]}, \theta^{[2]}, \ldots, \theta^{[i]} + \varepsilon, \ldots, \theta^{[L]}\right) - J\left(\theta^{[1]}, \theta^{[2]}, \ldots, \theta^{[i]} - \varepsilon, \ldots, \theta^{[L]}\right)}{2\varepsilon} \tag{7}$$

Usually we set $\varepsilon = 10^{-7}$. Now we compare this approximated derivative with $d\theta^{[i]} = \frac{\partial J}{\partial \theta^{[i]}}$. We then check quotient:

$$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} \tag{8}$$

If this quotient is in the ballpark of 10^{-7}, we conclude that the gradients are fine. If it is in the ballpark of 10^{-5}, there may be a bug somewhere in the code. If it is in the ballpark of 10^{-3}, then we should worry.

2.3 c)

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

Answer:
The `manual_train_update` function written in the task replaces the `optimizer.step()` and `optimizer.zero_grad()` methods with a manual implementation of parameters updating and zeroing the gradient after each iteration.

2.4 d)

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

Answer: The normal gradient descent algorithm is prone to oscillate around the search space based on the gradients. This will lead the search for the minima to take a high number of steps and more time before reaching it. Using momentum with gradient descent will make the search for the minima usually faster as it uses the exponentially weighted average of the previous gradients to accelerate the search in the same direction.

2.5 e)

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

Answer: Adding the regularization term penalizes parameters by multiplying them by $\left(1 - \frac{\alpha\lambda}{m}\right)$ (from Andrew's video on regularization). So, whenever some parameter is updated to become significantly large, it will increase the value of the cost function by the regularization term. As a result, it will be penalized and updated to a small value. Reducing the parameter-values. will result in reduced input values for the activation function ($z = w^T x + b$). A range of small input values (z) will simplify the model and discourage learning a more complex or flexible model, avoiding the risk of overfitting. This, however, depends on the activation function.

2.6 f)

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.

Answer: Following the recommendations from the videos, we perform a random search (seed has been set for reproducibility) with certain ranges in mind for the different hyperparameters. These ranges were set according to values that usually perform well with neural networks, so this in turn makes our decision of hyperparameters a fine, random search. For scaling, the logarithmic scale has been used for fine searches.

The selected parameters seem fine. The learning rate might look a bit low, but is in a common ballpark. As far as the regularization is concerned, the value is somewhat high, but the momentum might be a bit too low.

Using the test set on the best model given the hyperparameters, we achieved a test accuracy of $75,4\%$.

2.7 g)

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.

Answer: The results have been unexpected! Although all the manual training functions output the same as what the training functions that use the optimizer from PyTorch do, we get a sincere downgrade in accuracy from last year! This might be attributed to how the dataset was normalized, as this was the only major change in the code.

UPDATE (23:56 24th Feb.): I have mistakenly not preprocessed the training set after it was collected. The mistake has been corrected in the Jupyter Notebook, but has not been run due to time constraints. I am very aware of this, and I am trying to run the Jupyter Notebook as of writing this.

3 Division of labor

3.0.1 Kristian:

Last year: Commenting and *cleaning* the code, wrote some of the code. Did most of the debugging. Answered section 4.1.

This year: Read through all the code, some of it written originally by my lab partner Muhammad, fixing it and tidying it up for this year's hand-in.

3.0.2 Andreas:

Has to the best of his ability followed along and reading the code.