INF265
Project 1: Backpropagation and Gradient Descent
Alina Artemiuk

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

backpropagation.ipynb

`backpropagation:` This function calculates the gradients of the loss function with respect to the weights and biases of a neural network. The algorithm starts with computing the derivative of the loss function with respect to the output activation of the last layer da_L_dz_L (equation 5) and the delta for the last layer delta_L (equation 4). Then, the gradients of the loss function with respect to the weights and biases of the last layer dL_dw_L and dL_db_L are computed using delta_L and the activations of the previous layer. The computed gradients are stored in the dL_dw and dL_db dictionaries of the model.
Next, the algorithm iterates over the remaining layers in reverse order, computing the delta for each layer and the gradients of the loss function with respect to the weights and biases of the current layer using the delta and the weights of the previous layer, and the derivative of the activation function with respect to the input. These gradients are stored in the dL_dw and dL_db dictionaries of the model.
Function returns the updated model with the computed gradients.

gradient_descent.ipynb

`load_cifar:` This function loads and preprocesses the CIFAR-10 dataset. It also splits the dataset into training, validation and test sets. Additionally, it prints some sizes of the datasets.

`MyMLP:` This class implements a multi-layer perceptron (MLP) in PyTorch. The MLP has an input dimension of 3072 and an output dimension of 2. It has three hidden layers with 512, 128 and 32 hidden units. All activation functions are ReLU except for the last layer, which has no activation function since the cross-entropy loss already includes a softmax activation function.

`train:` This function trains a given model for a specified number of epochs using a given optimizer and loss function. The function starts by setting the device and initializing some variables to store the training and validation losses. Then it sets the model to training mode and initializes the optimizer's gradients to zero. The training loop iterates over each epoch and iterates over each batch in the training data. For each batch, the function applies the preprocessing function to the input data and moves the data to the specified device. It then feeds the input data into the model and calculates the loss between the model output and the expected output. The loss is backpropagated through the model and the optimizer is used to update the model's parameters.
After each epoch, the function evaluates the model's performance on the validation data. It does this by iterating over each batch in the validation data, applying the preprocessing function, and calculating the loss between the model output and the expected output.

`train_manual_update`: This function is a modified version of the `train` function that trains a PyTorch neural network model using manual parameter updates instead of an optimizer. The function initializes the model, sets it to training mode, and initializes the losses for the training and validation sets. It then iterates over the number of epochs specified, and for each epoch, it loops over the batches in the training data loader. For each batch, the function performs those steps:

- Preprocesses the input data and moves it to the specified device.
- Performs a forward pass through the model to get the predicted outputs.
- Calculates the loss between the predicted outputs and the actual labels.
- Backpropagates the loss to calculate the gradients for each parameter in the model.
- Performs a manual parameter update for each trainable parameter in the model. If weight_decay is specified, L2 regularization is added to the update. If momentum is specified, momentum is added to the update.
- Zeros out the gradients for the next batch.

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

The PyTorch methods used in the tasks described in section 2 in the backpropagation function for calculating gradients are:

- `torch.sub`: calculates the element-wise difference between two tensors
- `torch.einsum`: calculates the tensor contraction of one or more tensors
- `torch.squeeze`: removes dimensions of size 1 from the shape of a tensor. This is used to remove the extra dimension that is created by the delta tensor.

In addition to these methods, the function also uses the attributes of the model object, such as `model.fc` (the dictionary of fully connected layers), `model.z` (the dictionary of activation values), `model.a` (the dictionary of values after activation function), model.df (the dictionary of activation function derivatives), and `model.dL_dw` and `model.dL_db` (the dictionaries of gradients).

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.

One method that can be used to check whether the computed gradient of a function seems correct is numerical gradient checking.

Here are the steps of this method:

1. Choose a small epsilon value.
2. For each model parameter, calculate the analytical gradient using the chain rule.
3. For each model parameter, calculate the numerical gradient using the finite difference method. To do this, add the epsilon to the parameter value and calculate the function output, then subtract the epsilon from the parameter value and calculate the function output. Then calculate the numerical gradient.
4. Compare the analytical and numerical gradients for each parameter. If the difference between the two gradients is smaller than the threshold, the computed gradient is likely correct.

The `grad_check` function implements a numerical gradient check. It computes the loss gradients for each model parameter using both back-propagation and finite differences and compares the results. The function returns a flag indicating whether the gradients are consistent and the error between the gradients.
To use this function, you first need to define a neural network model and a loss function. The `grad_check` function takes as input an instance of this class along with input data, labels, and a loss function. Once we have defined model and loss function, we can call `grad_check` to check the gradients.

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

Instead of using an optimizer, the function directly updates the trainable parameters of the model using the learning rate `lr` and the gradient grad. The PyTorch method to implement the function is `p.data -= p.grad * lr`. Additionally, after each iteration, the gradients zeroed out using `model.zero_grad()`.

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

The purpose of adding momentum to the gradient descent algorithm is to accelerate convergence and improve the stability of the learning process.

In standard gradient descent, the weights are updated at each iteration based on the gradient of the loss function with respect to the weights. However, this can lead to oscillations or slow convergence.
On the other hand, momentum addresses these issues by using an exponentially weighted moving average of past gradients to update the weights. This means that the update at each iteration is influenced not only by the current gradient, but also by the previous gradients, thereby smoothing out oscillations and speed up convergence, especially in areas where the gradient changes direction frequently.

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

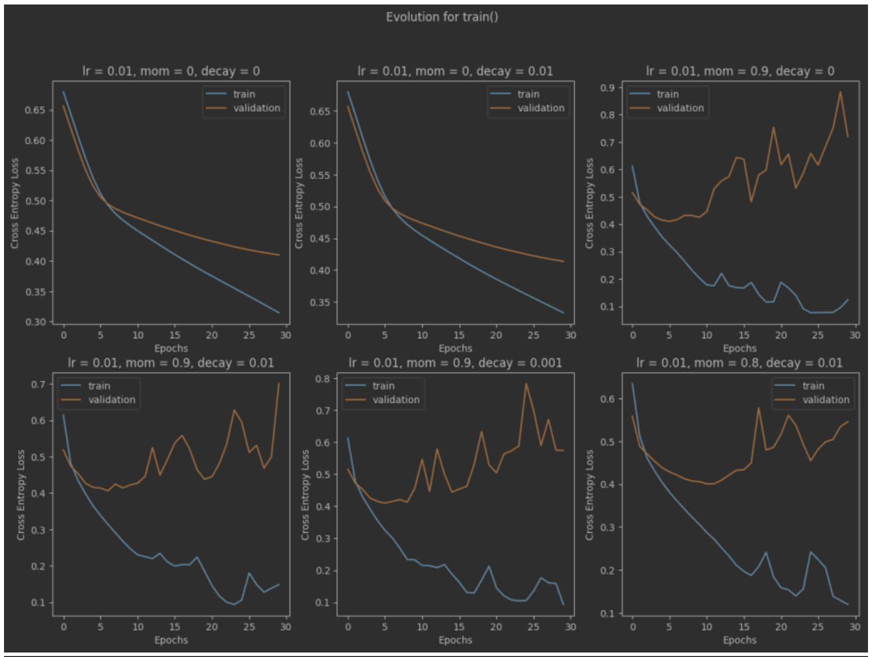
The purpose of adding regularization to the gradient descent algorithm is to prevent overfitting and improve the generalization performance of the model.

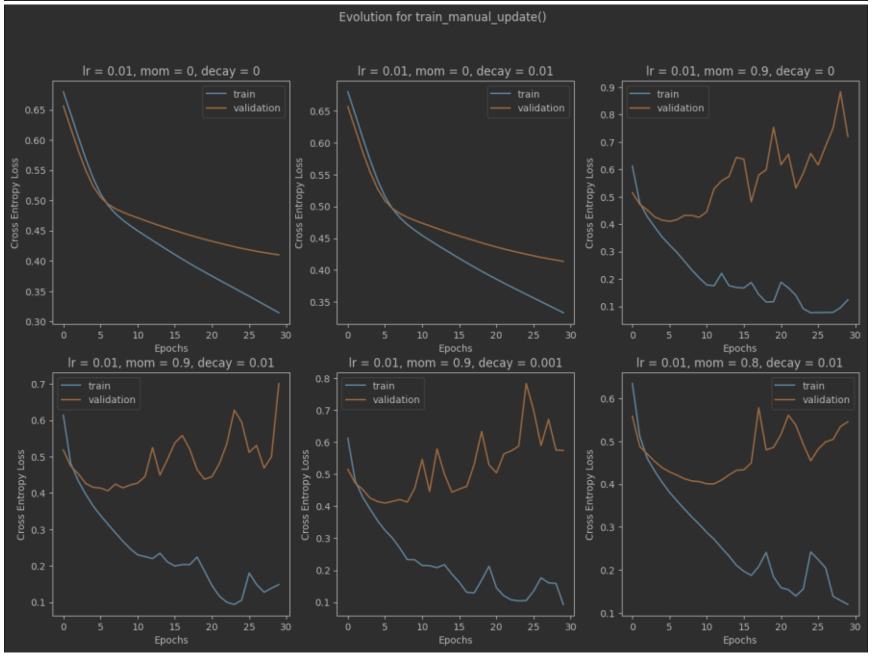
There are several types of regularization techniques that can be used, including L1 and L2 regularization, dropout, and early stopping. Where:

- L1 and L2 regularization add a penalty term to the loss function that encourages the weights to be small, which can help to prevent overfitting by reducing the complexity of the model.
- Dropout randomly sets to zero some of the neurons in the network during training, which can help to prevent overfitting by forcing the network to learn more robust features.
- Early stopping involves monitoring the validation loss during training and stopping the training when the validation loss stops improving. This can help to prevent overfitting by avoiding the point at which the model starts to fit the noise in the training data.

2.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.

I chose the parameters and the same number of models as in the file `gradient_descent_output.txt`. To choose the best model among the trained ones in terms of accuracy, I used the arrays of training and validation accuracy of the 6 models found with `train()` only, because `train()` and `train_manual_update()` give identical results, which can be seen in these training curves:





2.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.

Based on those results:

```
Model [1] (lr = 0.01, mom = 0, decay = 0)

Training accuracy:     0.87081
Validation accuracy:   0.84107
--------------------------------------------------
Model [2] (lr = 0.01, mom = 0, decay = 0.01)

Training accuracy:     0.86236
Validation accuracy:   0.83514
--------------------------------------------------
Model [3] (lr = 0.01, mom = 0.9, decay = 0)

Training accuracy:     0.93724
Validation accuracy:   0.84107
--------------------------------------------------
Model [4] (lr = 0.01, mom = 0.9, decay = 0.01)

Training accuracy:     0.88673
Validation accuracy:   0.78282
--------------------------------------------------
Model [5] (lr = 0.01, mom = 0.9, decay = 0.001)

Training accuracy:     0.98286
Validation accuracy:   0.85489
--------------------------------------------------
Model [6] (lr = 0.01, mom = 0.8, decay = 0.01)

Training accuracy:     0.95549
Validation accuracy:   0.82823
```

Model 1 and 2 have similar training and validation accuracies, with Model 1 performing slightly better. This suggests that adding weight decay=0.01 did not help improve the performance of the model.
Model 3 achieved the highest training accuracy among all the models, indicating that the addition of momentum=0.9 helped improve the model's performance on the training set.
Model 4 has a lower validation accuracy compared to the other models, indicating that the addition of weight decay=0.01 and momentum=0.9 may have negatively impacted the model's generalization performance.
Model 5 achieved the highest validation accuracy among all the models, indicating that the combination of momentum=0.9 and weight decay=0.001 helped improve the model's generalization performance.
Model 6 achieved a relatively high training accuracy and a moderate validation accuracy, suggesting that the combination of momentum=0.8 and weight decay=0.01 may have helped improve the model's performance on the training set but did not significantly improve its generalization performance.

Based solely on the validation accuracy metric, Model 5 appears to be the best among all the models as it achieved the highest validation accuracy of 0.85489. However, it is important to remember that choosing the best model may also depend on other factors. The test performance of 0.855 indicates that the model achieved a good level of accuracy on the test dataset.