Hw3 Coding Solutions

3.1.1 Linear layer (6pts) First, you need to implement the linear layer of the MLP by implementing three python functions in `class linear_layer`. This layer has two parameters `W` and `b`.

```
self.params['W'] = np.random.normal(0, 0.1, size=(input_D, output_D))
self.params['b'] = np.random.normal(0, 0.1, size=(1, output_D))


########################################################################################
# TODO: Initialize the following two (gradients) with zeros
#   - self.gradient['W']
#   - self.gradient['b']
########################################################################################
self.gradient['W'] = np.zeros(shape=(input_D, output_D))
self.gradient['b'] = np.zeros(shape=(1, output_D))
```

```
#############################################################################
# TODO: Implement the linear forward pass. Store the result in forward_output  #
#############################################################################
forward_output = X @ self.params['W'] + self.params['b']
```

```
#############################################################################################
# TODO: Implement the backward pass (i.e., compute the following three terms)
#   - self.gradient['W'] (input_D-by-output_D numpy array, the gradient of the mini-batch loss w.r.t. self.params['W'])
#   - self.gradient['b'] (1-by-output_D numpy array, the gradient of the mini-batch loss w.r.t. self.params['b'])
#   - backward_output (N-by-input_D numpy array, the gradient of the mini-batch loss w.r.t. X)
# only return backward_output, but need to compute self.gradient['W'] and self.gradient['b']
#############################################################################################
self.gradient['W'] = X.T @ grad
self.gradient['b'] = np.sum(grad, axis=0)
backward_output = grad @ self.params['W'].T
```

3.1.2 ReLU (4pts) Next, you need to implement the ReLU activation by implementing 2 python functions in `class relu`. There are no parameters to be learned in this module.

```
##########################################################################
# TODO: Implement the relu forward pass. Store the result in forward_output    #
##########################################################################
self.mask = (X > 0).astype(int)
forward_output = X * self.mask
```

```
##########################################################################################
# TODO: Implement the backward pass
# You can use the mask created in the forward step.
##########################################################################################
backward_output = grad * self.mask
```

3.1.4 Backpropagation (4pts) Your network is almost ready to be trained. The only function you need to implement in this part is `backward_pass`, which calls the backward pass of each layer in the correct order and with the correct inputs.

```
# 4. Model backward pass
def backward_pass(model, x, a1, h1, a2, y):
    # grad_a2 = model['loss'].backward(a2, y)
    ################################################################################
    # TODO: Call the backward methods of every layer in the model in reverse order.
    # We have given the first and last backward calls (above and below this TODO block).
    ################################################################################
    grad_a2 = model['loss'].backward(a2, y)
    grad_h1 = model['L2'].backward(h1, grad_a2)
    grad_a1 = model['nonlinear1'].backward(a1, grad_h1)
    grad_x = model['L1'].backward(x, grad_a1)
```

3.1.5 Gradient checker (4pts) As you probably realized as you wrote the code for the previous parts, backpropagation is a subtle algorithm and it is easy to make mistakes. In this part, you will implement gradient checking to give you confidence that you are calculating the gradients of your model correctly. The idea behind gradient checking is as follows. As mentioned in the lecture, one naive way to approximate gradient is by

$$\frac{dF(w)}{dw} = \lim_{\epsilon \to 0} \frac{F(w+\epsilon) - F(w-\epsilon)}{2\epsilon}$$

```
model[layer_name].params[param_name] += epsilon
_, _, _, f_w_add_epsilon = forward_pass(model, x, y)

################################################################################
# TODO: Estimate the gradient of parameters from the loss function F(w + epsilon) and F(w - epsilon).
# Take one forward pass with w - epsilon
# Refer to the lecture notes for the exact equation for computing the approximate gradient
################################################################################
model[layer_name].params[param_name] -= 2 * epsilon
_, _, _, f_w_minus_epsilon = forward_pass(model, x, y)
approximate_gradient = (f_w_add_epsilon - f_w_minus_epsilon) / (2 * epsilon_value)
model[layer_name].params[param_name] += epsilon
print("Check the gradient of %s in the %s layer from backpropagation: %f and from approximation: %f"
      % (param_name, layer_name, grad, approximate_gradient))
```

3.2.3 Analysis (6pts) Answer the following question based on the previous plots and results:

(i). Does smaller batch size guarantee faster training? Why do you think this is the case?

(ii). Does larger batch size imply higher test accuracy? Why do you think this is the case?

(iii). Does larger batch size imply less gradient updates to converge? Why do you think this is the case?

(i). No, **more gradient updates** are needed to converge, which takes time. Also, because of the way computer memory is organized, it is often as **efficient to take the gradient with respect to a few samples** as it is to take the gradient with respect to a single sample.

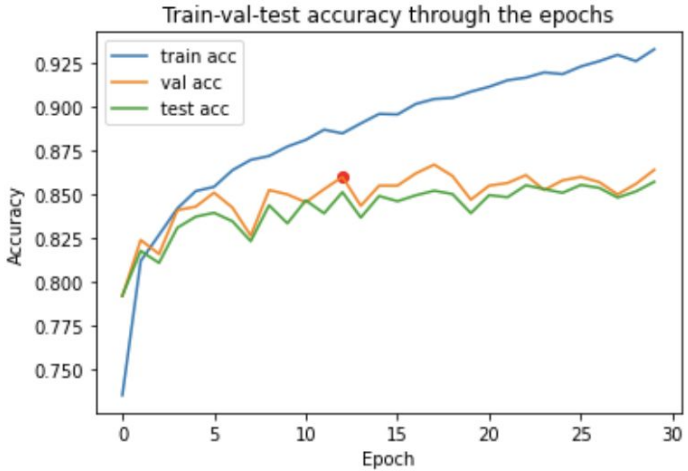
(ii). No, the **stochasticity** in smaller batch sizes **helps with generalization**. See https://wandb.ai/ayush-thakur/dl-question-bank/reports/What-s-the-Optimal-Batch-Size-to-Train-a-Neural-Network---VmlldzoyMDkyNDU and https://stats.stackexchange.com/questions/164876/what-is-the-trade-off-between-batch-size-and-number-of-iterations-to-train-a-neu.

(iii). Yes, all are **unbiased estimates** of the gradient but larger batch size implies **smaller variance**.

3.3 The effect of early-stopping (6pts) [3.5min] We provided built-in early-stopping in the starter code, but did not explore the effect that it has on training. In this question, we will explore the effect of early-stopping. To do this, we evaluate the training, validation and test accuracy for each training epoch. We then plot the training, validation and test accuracy throughout the training process to see how early-stopping works. We train 30 epochs on a batch size of 5 without early-stopping.

(ii). What is the trend of training and test accuracy after the early-stopped point? (2pts)

(iii). Based on the plot, what do you think could go wrong if the patience parameter for early-stopping is too small? (Recall that if the patience parameter is set to k epochs, then training will terminate if there is no improvement in the validation accuracy for k epochs in a row.) (2pts)



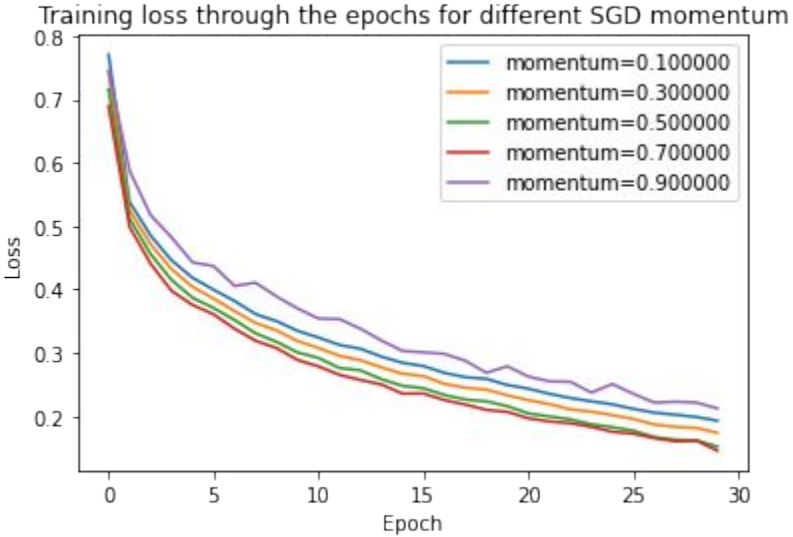
(ii). Training curve goes up, but validation and test curve fluctuate.
(iii). **Stop at suboptimal** points due to small fluctuations.

3.4 SGD with momentum (6pts) [14min] We mentioned in class that adding a "momentum" term, which encourages the model to continue along the previous gradient direction helps the network to converge. Concretely, with an initial velocity $\boldsymbol{v} = \mathbf{0}$, we update the gradient by

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} + \nabla F(\boldsymbol{w})$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{v}$$

where η is the learning rate and α is the factor describing how much weight we put on the previous gradients. $\alpha = 0$ is equivalent to gradient update without momentum.

(ii). Based on this, what is a suitable value of α? Therefore, how should training ideally rely on previous gradients for better convergence? (2pts)



(ii). 0.5 or 0.7, both are good. Relying on previous gradients **helps convergence**, but **too much is harmful**.

4.1 Training the Model (10pts) Now, we will train our model. In this section, you are not expected to implement anything.

(i). Before training the model, make it generate some sequence. Then, you will train the model with tiny Shakespeare dataset. We provided you the initial hyper-parameters. With Colab, the training session should take around 10 minutes. After training the model, make it generate some sequence again. Compare it with the first generation. Also, compare the generation with training data. Do they look similar? Explain the reason why the generation is not like a Wikipedia article, for example. (3pts)

(ii). Provide the loss curve for train and validation during training. (1pt)

(iii). For different values of block-size (sequence-length/context-length) [2, 16, 64], train the model from scratch. Provide the train-validation loss curves and model's generations for different sequence-lengths. Discuss how the sequence-length changes the generation of the model. (3pts)

(iv). For different number of heads [2, 4, 8], train the model from scratch. Provide the train-validation loss curves and model's generations. Discuss how the number of heads changes the train-validation loss curves. (3pts)