

April 7, 2024

HOMEWORK 1 — Report

1 Question 1

Question 1.1 - Preliminaries

The partial derivative calculation steps for Tanh, Sigmoid, and ReLU activation functions are shown in Figure 1.

The figure shows handwritten calculations for the partial derivatives of three activation functions:

- Tanh:**
$$\frac{dy_1}{dx} = \frac{d}{dx} \left(\frac{e^{2x} - 1}{e^{2x} + 1} \right) = \frac{(2e^{2x})(e^{2x} + 1) - (2e^{2x})(e^{2x} - 1)}{(e^{2x} + 1)^2}$$
$$\Rightarrow \frac{(2e^{2x})[e^{2x} + 1 - e^{2x} + 1]}{(e^{2x} + 1)^2} = \frac{4e^{2x}}{(e^{2x} + 1)^2}$$
- Sigmoid:**
$$\frac{dy_2}{dx} = \frac{d}{dx} \left(\frac{1}{1 + e^{-x}} \right) = \frac{0 - (-1)e^{-x}(1)}{(1 + e^{-x})^2}$$
$$\Rightarrow \frac{e^{-x}}{(1 + e^{-x})^2}$$
- ReLU:**

We can calculate it from graphical point of view.

$$\frac{dy_3}{dx} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

A small graph of the ReLU function is shown, which is zero for $x < 0$ and increases linearly for $x \geq 0$.

Figure 1: Partial derivative calculation steps for Tanh, Sigmoid and ReLU activation functions.

Figure 2 illustrates the activation functions' response between -2 and 2. The plots are obtained using the matplotlib library as instructed.

Figure 3 indicates the gradients of those functions in the same range. The gradients are calculated using the partial derivatives derived in Figure 1.

Question 1.2

In this part, MLP with one hidden layer is implemented utilizing the given code template. The input-output pairs are fetched from the XOR data provided in utils.py. Note that for all networks, the learning

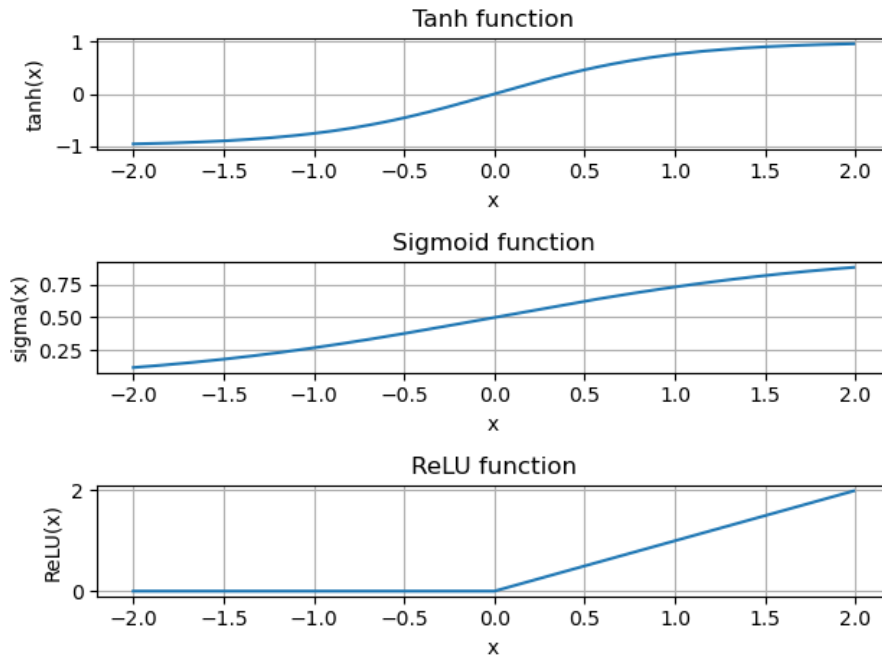


Figure 2: Activation functions plot.

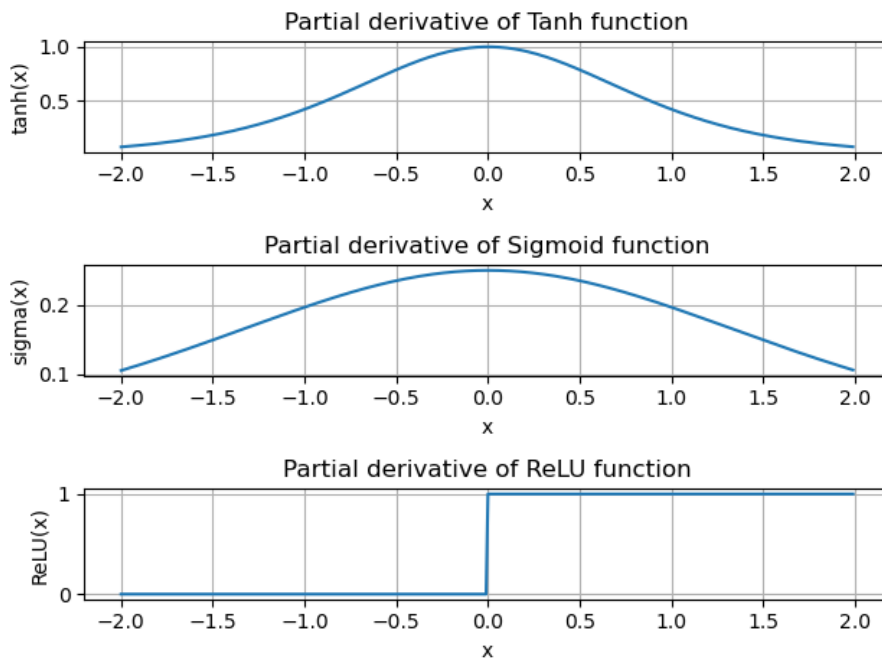


Figure 3: Gradients of the activation functions plot.

rate is fixed at 0.00001, and seed is utilized. Figure 4 shows the decision boundary for the sigmoid-activated network.

Similarly, Figure 5 is the decision boundary for the tanh-activated network, and Figure 6 is the decision boundary for the ReLU-activated network.

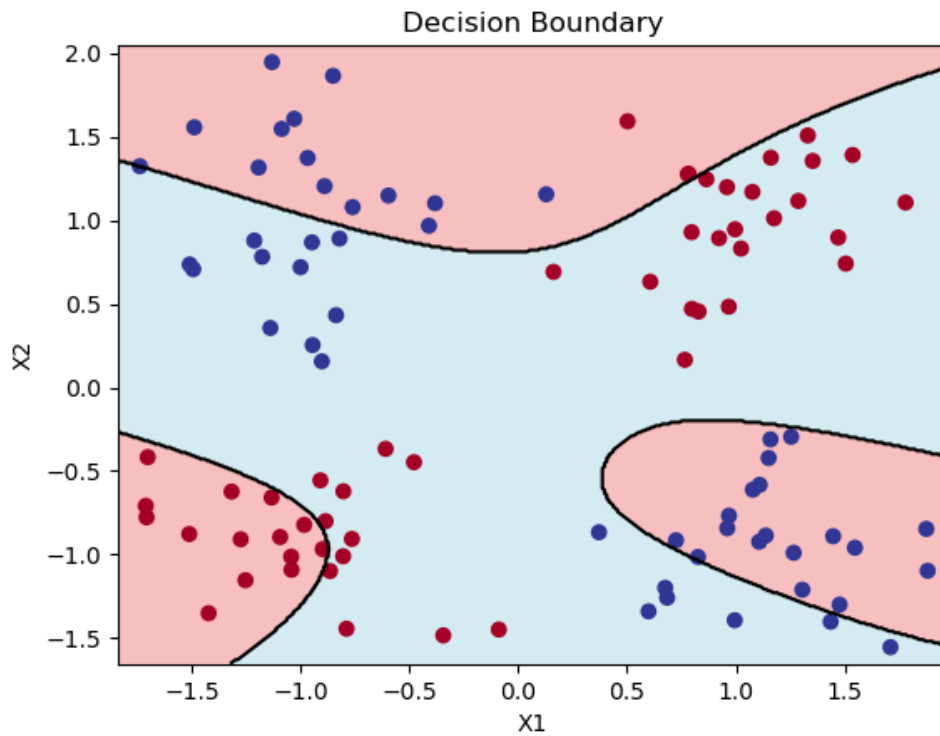


Figure 4: Sigmoid activated XOR problem output.

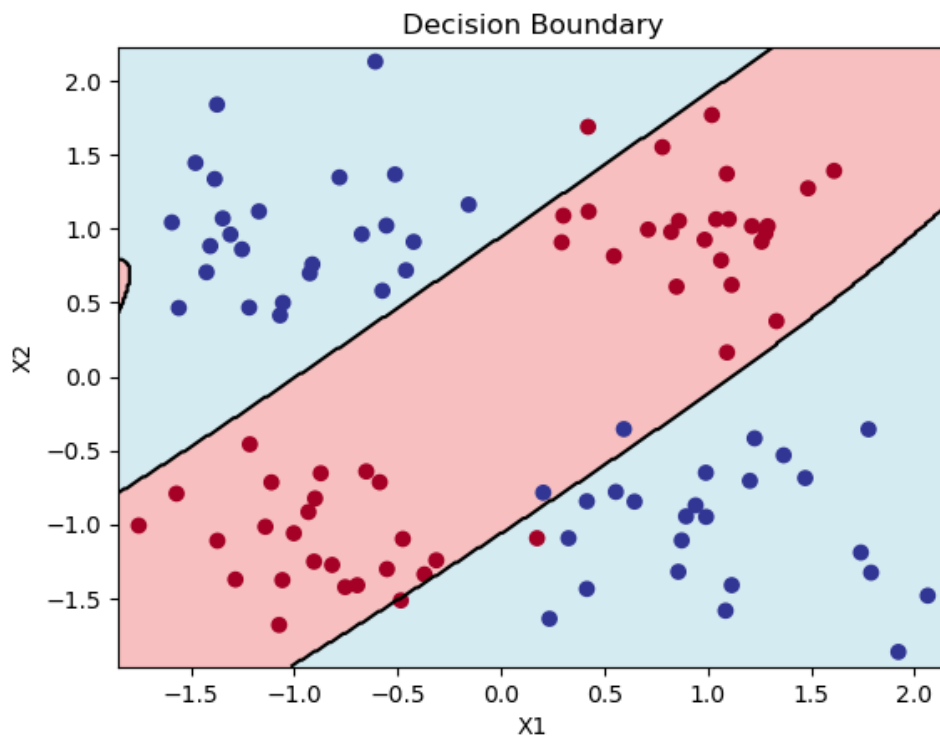


Figure 5: Tanh activated XOR problem output.

Question 1.3 - Discussions

1. All of those activation functions provide a smooth transition from one state to another. The advantage of Tanh and Sigmoid is they are limited in the range of -1 to 1 and 0 to 1, respectively. This property

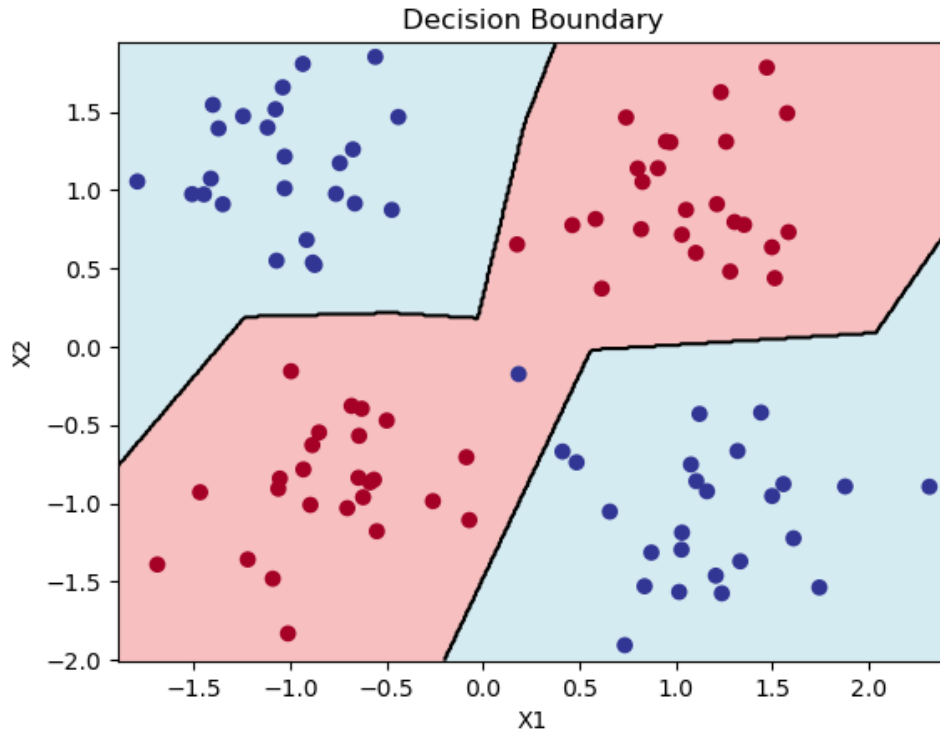


Figure 6: ReLU activated XOR problem output.

can be beneficial in some cases. However, the ReLU function is not limited in the range. It is also computationally cheaper than the other two. The disadvantage of ReLU is that it is not smooth at the origin. This can cause some problems in the optimization process. Another advantage of ReLU is negative gradients are zero. This may be helpful in some cases.

2. XOR problem is an input decision problem where inputs have to be different than each other to obtain 1. Since the output is always with respect to the state of two variables, the decision boundary is not linear. Therefore, a single-layer perceptron can not be solved since, in one step, two case evaluations can not be done. Yet, by adding a hidden layer, the problem is solved. The activation functions are used to introduce non-linearity to the network. The decision boundary of the XOR problem is shown in Figures 4, 5 and 6. As can be seen from the figures, the MLPs with activation functions can solve the XOR problem to some extent.

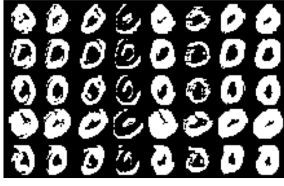
3. The boundaries change in each run since the initial weights and the data points are randomly generated. Therefore, the decision boundaries are dependent on the training process, where initial randomness leads to different outputs.

2 Question 2

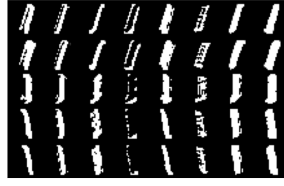
A convolution operator is implemented using a set of nested loops. The outputs are checked on a separate code so that they are identical to the output of `torch.conv2d`. Using the provided inputs, outputs provided in Figure 7 are obtained.

Question 2.2 - Discussions

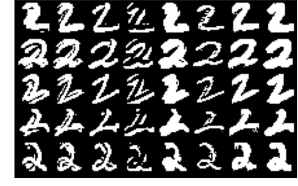
1. Two-dimensional convolution operation provides filtering in two dimensions via a kernel. The kernel is applied to the input image in a sliding window fashion. The output is obtained by element-wise



(a) Output 0



(b) Output 1



(c) Output 2



(d) Output 3



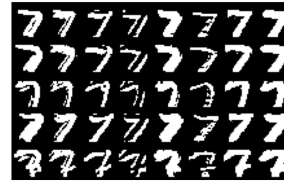
(e) Output 4



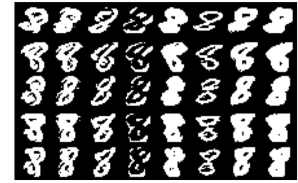
(f) Output 5



(g) Output 6



(h) Output 7



(i) Output 8

Figure 7: Convolution output

multiplication of the kernel and the input image. The kernel is then shifted by a stride, and the process is repeated. That is, two-dimensional convolution operations with learnable kernel entries are commonly used in image processing to extract features from the image. The kernel is learned during the training process. Those feature maps encode necessary information about distinct clues in the image. For example, in the case of object recognition, a specific kernel can be trained to detect bicycle rims. The kernel of a convolution layer corresponds to the filter function in one dimension. It is used to suppress or enhance certain information in the input image.

2. The sizes of the kernel correspond to the size of the filter. Therefore, the size list can be explained as follows (batch size, input channels, output channels, filter height, filter width). Batch size is the number of input sets in a batch if batching is used; we have not utilized it here. Input channels are the number of depth dimensions in the image; for example, in RGB frames, it is three. This might be different for

different sensor inputs and representations. The output channel number determines the depth dimension of the output tensor. The filter height and width are the dimensions of the filter.

3. To understand what exactly happened here, let us plot the kernel and input for the number zero. Figure 8 illustrates the plot.

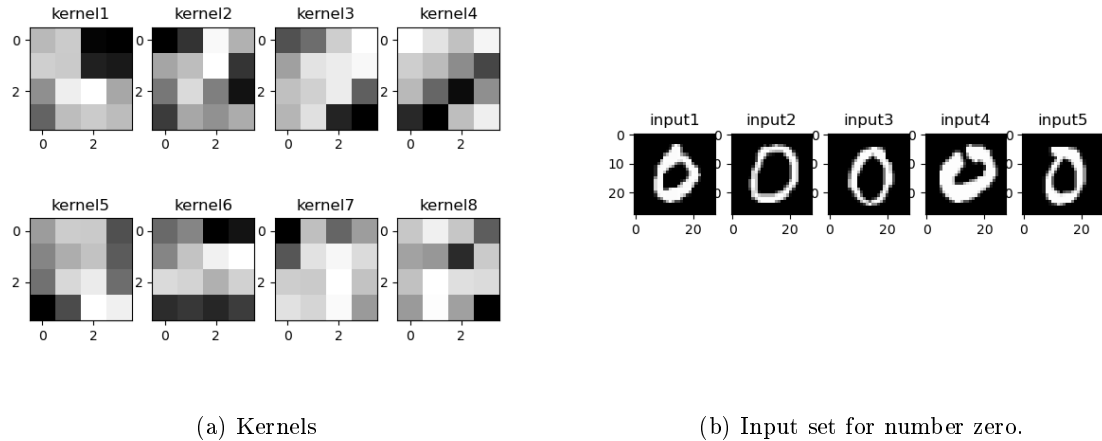


Figure 8: Kernel and input for number zero.

When we have a look at what the output image represents, on each row, an input image is convolved with a kernel. Different kernels are applied to the input image. The output image is the result of those convolutions. Figure 8 shows the input and kernel pairs in order.

4. Each convolution kernel embeds a certain feature of the image. That is, if we have a look at Figure 8.a, we can see that kernel 4 enhances the contours on the image, whereas kernel 6 enhances the filled part of those contours. This is the reason why we see similarly formatted outputs in the same column.

5. Similarly, since we "highlight" different properties on the image without filters, we see different patterns on the output side, even if the input is the same.

6. So, we can interpret from 4 and 5 that the output of the convolution layer is the result of the feature extraction process. The output is the result of the convolution of the input image with the learned filters. The output is the feature map that encodes the information about the input image. By post-processing those feature maps, we can obtain the necessary interpretations of the input image.

3 Question 3

Question 3.1

The implementations related to each architecture are completed as instructed. The code is given in the appendix. As a result, the plot shown in Figure 9 is obtained. Also, the first layer weights recorded are plotted and shown in Figure 10.

Question 3.2 - Discussions

1. The generalization performance of a classifier is the ability of the classifier to perform well on unseen data. That is, the classifier should be able to generalize the patterns in the training data to the test data. The overfitting phenomenon is the case where the model fits the training data too much and loses its generalization.

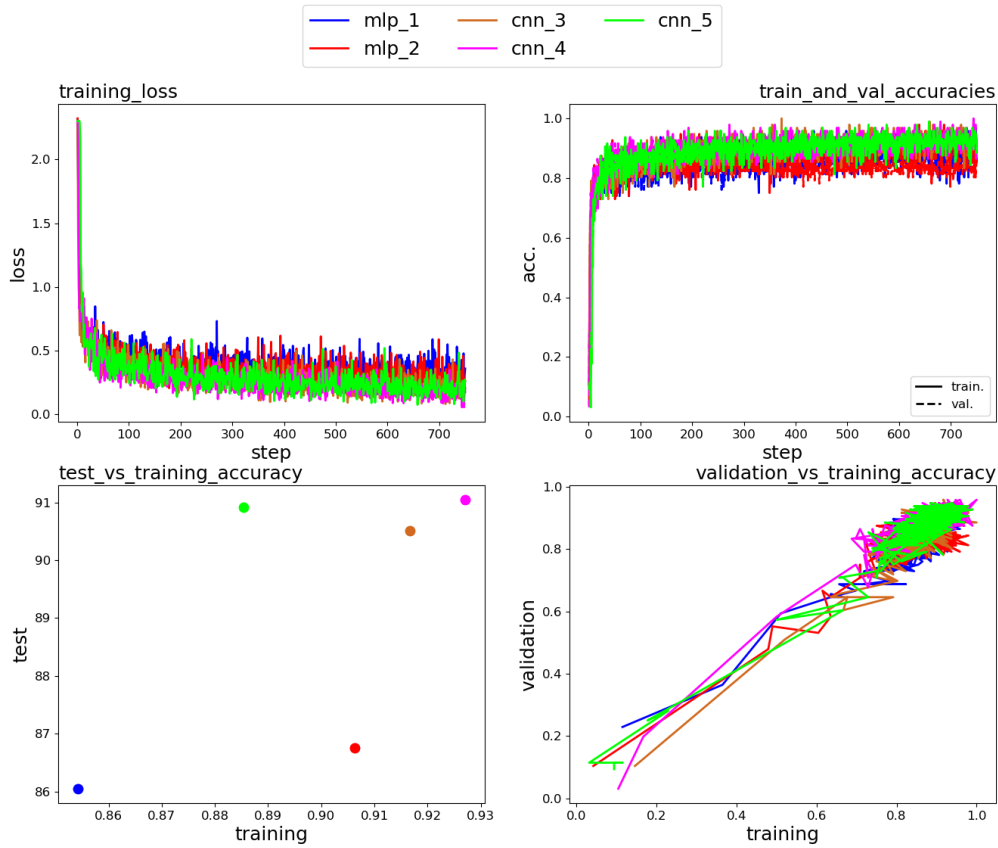


Figure 9: Benchmark of five different architectures.

2. The plots, test vs. training accuracy and validation vs. training accuracy are the most informative in this sense.

3. Somehow, the validation vs training accuracy plot is hard to read out. Therefore, deductions are made from test vs training accuracy plots. First of all, one can see right away that the plot that CNN-based models generalized better, whereas MLP-based ones have testing accuracy lower than training accuracy. On the other hand, amongst CNN-based ones, cnn4 has the best accuracy in terms of both training and test accuracy. However, it can be deduced that cnn5 has the best generalization performance since it has quite high test accuracy compared to its training accuracy. The test scores are even better than the training scores.

4.

Architecture	Trainable Params
mlp 1	25,450
mlp 2	27,818
cnn 3	22,626
cnn 4	14,962
cnn 5	10,986

Table 1: Trainable Parameters of Different Architectures

As the number of parameters increases, the number of points to fit the data increases. Therefore, the model can fit the data better in the most basic setting. The number of parameters for each architecture is given in Figure 1. So, classification performance increases as the number of parameters increases to a

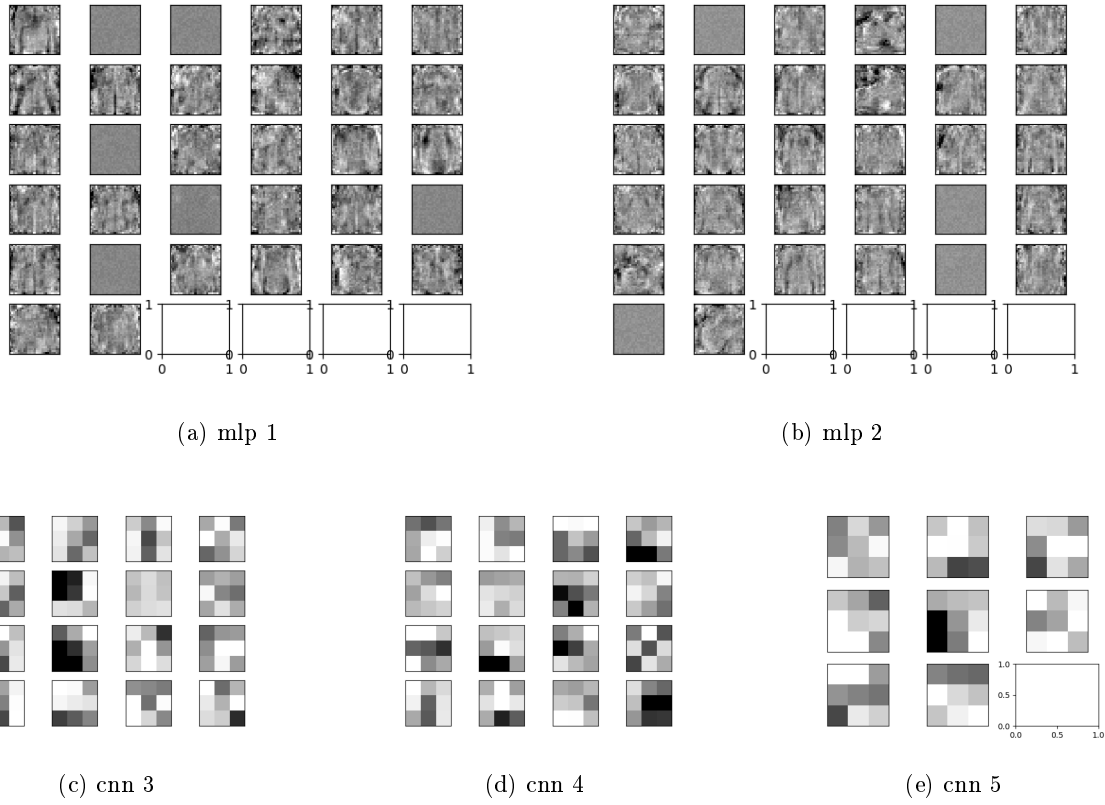


Figure 10: Weights of the first layers.

certain point, as we can observe from multi-layer perceptron training. From cnn3 to cnn4, both training and test accuracy increases, but a number of parameters are lower in cnn4. However, from cnn4 to cnn5, the test accuracy is almost the same, whereas the training accuracy decreases considerably. Therefore, it can be said that using only a number of parameters to describe generalization and training capabilities is not the best idea.

5. As the depth of the network increases, e.g., from mlp1 to cnn5 network depth increases, in general, both classification performance increases up to a certain level and then decreases. For generalization, we can say that it always increases according to the data we have right now in those examples.

6. The mlp weights are not interpretable. However, one may interpret what kind of filtering is done by the CNN weights by looking at the plots shown in Figure 10. The first layers, in general, encode the most basic features like corners, edges, and so on.

7. The limited experience of the report writer does not allow them to extract the information on whether the filters are specific to the cases by looking at them. However, principally, certain CNN filters are trained to detect certain features in the input image. Again, the first layers encode the most simplistic features like corners and edges. Therefore, the filters are specific to the classes to distinguish them better.

8. Similarly, it is hard to distinguish between the filters by looking at the weights. The most interpretable ones are from the cnn4 architecture. The filters are more clear in this case.

9. The mlp1 and mlp2 are the simplest architectures that are similar to each other. The mlp2 introduces one more layer. The cnn3, cnn4, and cnn5 are the convolutional neural network architectures. The cnn3 has three layers with three different spatial kernels. The cnn5 has six convolutional layers with fixed spatial dimensions, but it is the deepest of all five of the architectures. As done in the previous sections, mlp2 performs better than mlp1 by increasing the number of parameters. Cnn4 performs better than cnn3 while having similar number parameters and having higher depth. Cnn5 has a significantly

low number of parameters, so it underfits the training set but generalizes quite nicely.

10. I would pick cnn4 since it has the best test accuracy and generalization performance. It seems it can fit the data well and does not overfit in this setting.

4 Question 4

Question 4.1

As instructed in the homework documentation, the implementation for both ReLU and sigmoid activate models is done. Necessary statistics are recorded. As a result, the plot shown in Figure 11 is obtained. Note that the utils.py had to be modified (alpha value edited) to have a proper distinction. Also, Figure 12 illustrates the individual results for each architecture.

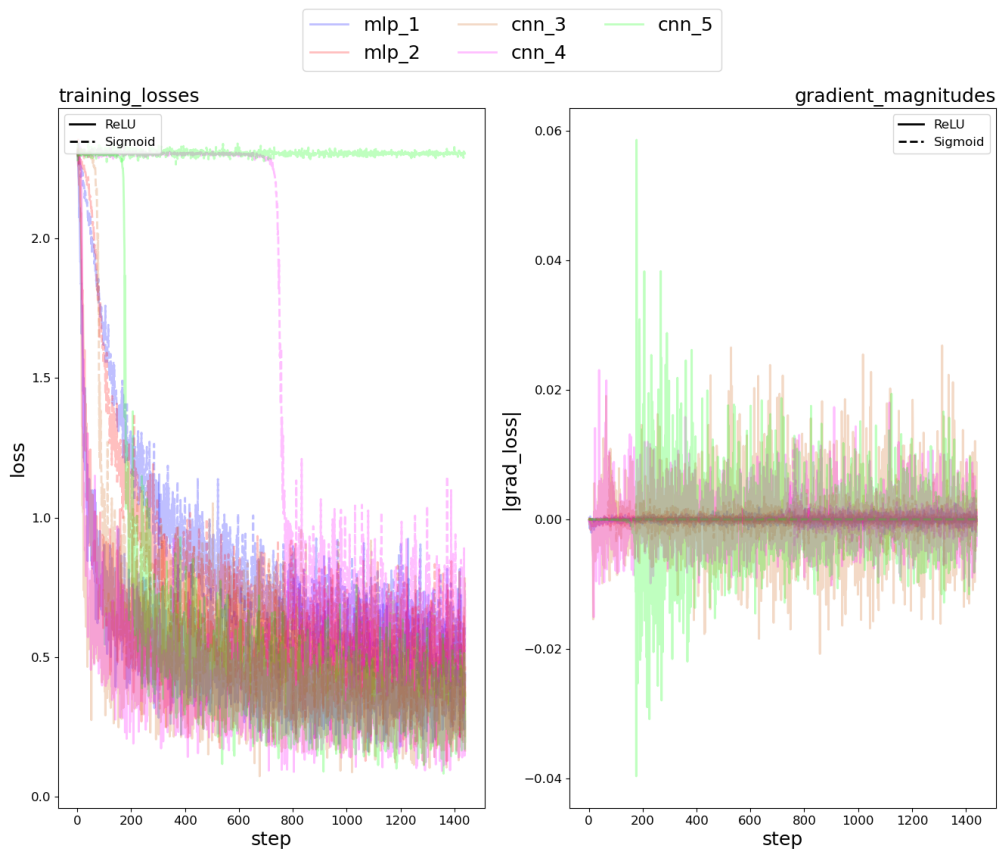


Figure 11: Benchmark of five different architectures trained using ReLU and Sigmoid activation functions.

Question 4.2 - Discussions

1. – 2. Gradient behavior is similar in each architecture if we focus on ReLU-activated ones. The values fluctuate around the 0 line as expected. However, the sigmoid-activated ones exhibit different behaviors. The gradients are quite small. Also, as the depth increases, the gradients for sigmoid-activated ones get smaller and quite close to 0. This is due to the vanishing gradient problem. The gradients are smaller and smaller as the depth increases in each layer pass. Furthermore, for the cnn5 case, it is almost always 0. It can be said that using the sigmoid activation function instead of ReLU is not the best idea for deep networks.

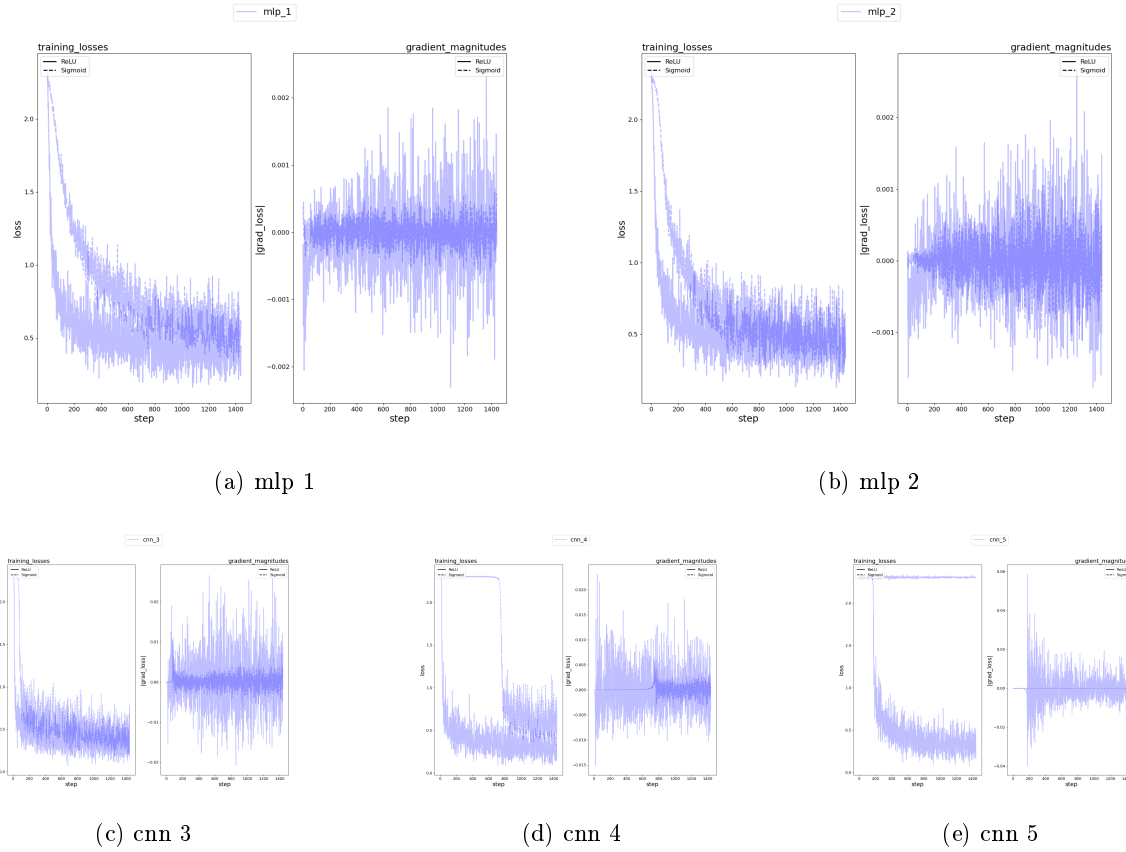


Figure 12: ReLU vs Sigmoid for each architecture.

3. In part 1.2. case, the network was only one layer, but the effect was similar, where the gradient values were smaller in the activated one. However, it is important to note that in order to obtain similar performance in both the ReLU and Sigmoid one, the learning rate needs to be adjusted where, overall, the weight update step yields similar scales. In this report, to have a fair comparison, the learning rate was fixed to a small value where ReLU did not blow up the gradients.

4. As pointed out in the previous paragraph, this time, Sigmoid could have performed better and avoided vanishing the gradients. On the other hand, depending on the rates, ReLU could have exploded and improperly updated the weights.

5 Question 5

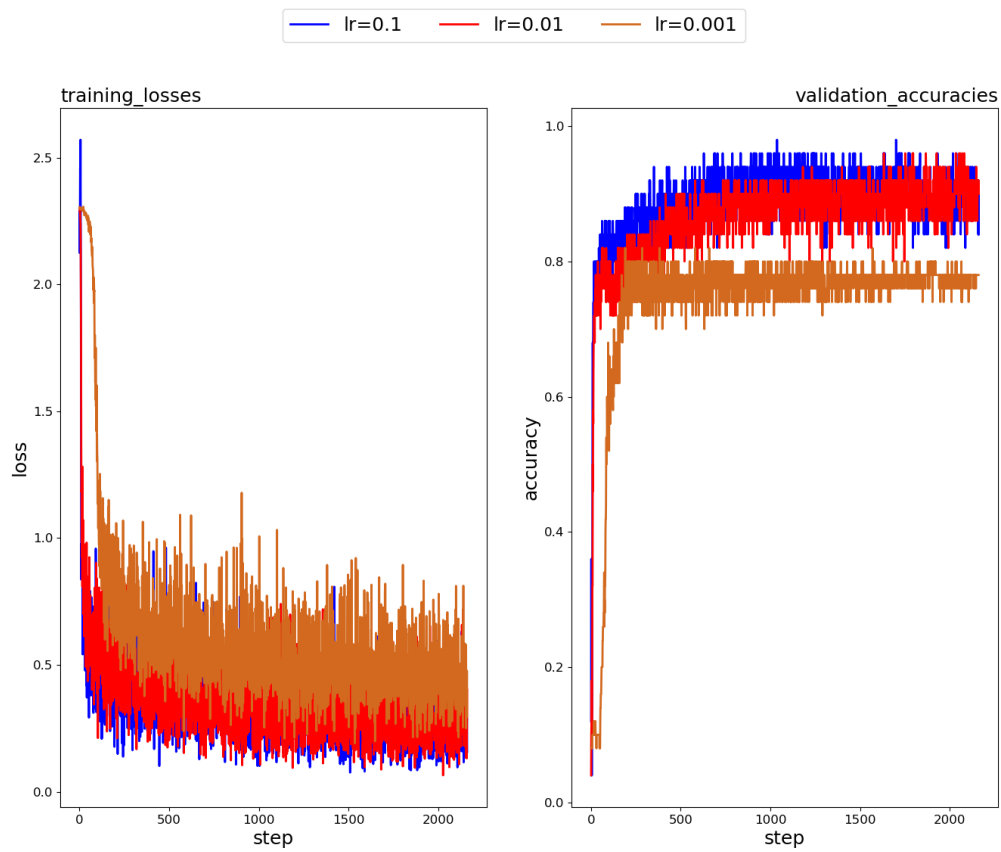
This time, the validation set is selected as ten percent of the training set.

Question 5.1

I picked the cnn3 architecture for learning rate experimentation. The learning rates are selected as 0.1, 0.01, 0.001, 0.0001 and 0.00001. The results are shown in Figure 13.

Then scheduling is applied to the learning rate. Figure 14,15,16 illustrates the scheduling applied cases. The learning rate is decreased by a factor of 0.1 at each selected epoch.

It should be indicated that an inconsistent behavior for dropping the learning rate only for one case is observed. For example, when the exact same setup is run on different CUDA setups, the results are quite different, yet this also can be observed from the initial part of the b). This is repeated multiple times in different computation units. This was assumed to be an unsolved problem related to random



training of <cn3_3> with different learning rates

Figure 13: Different learning rate settings on cn3.

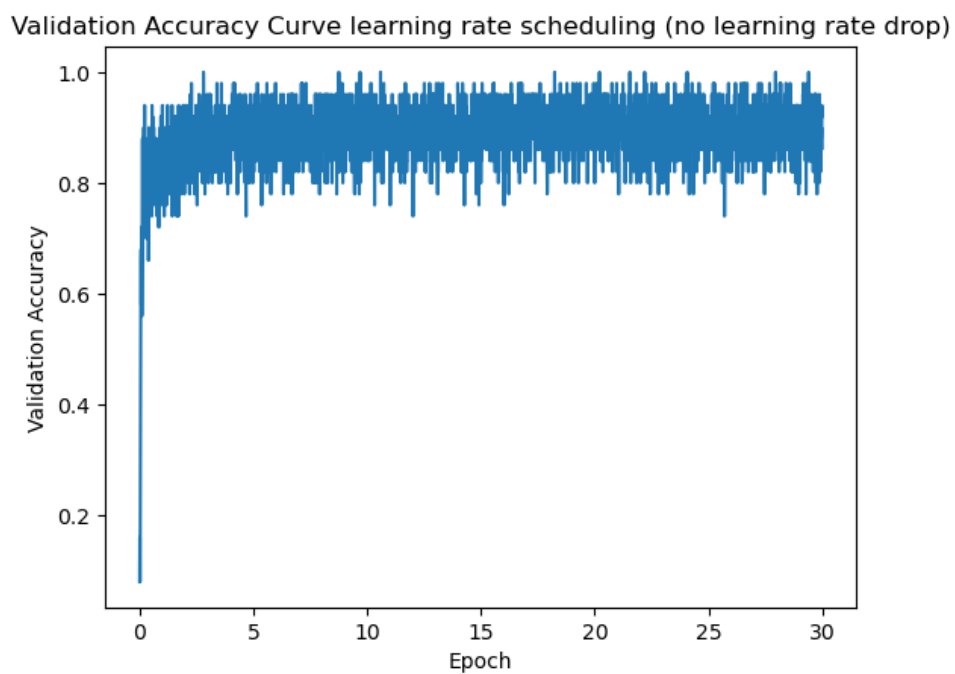


Figure 14: No scheduling

initializations and seeds.

Validation Accuracy Curve learning rate scheduling (rate drops at 7th epoch)

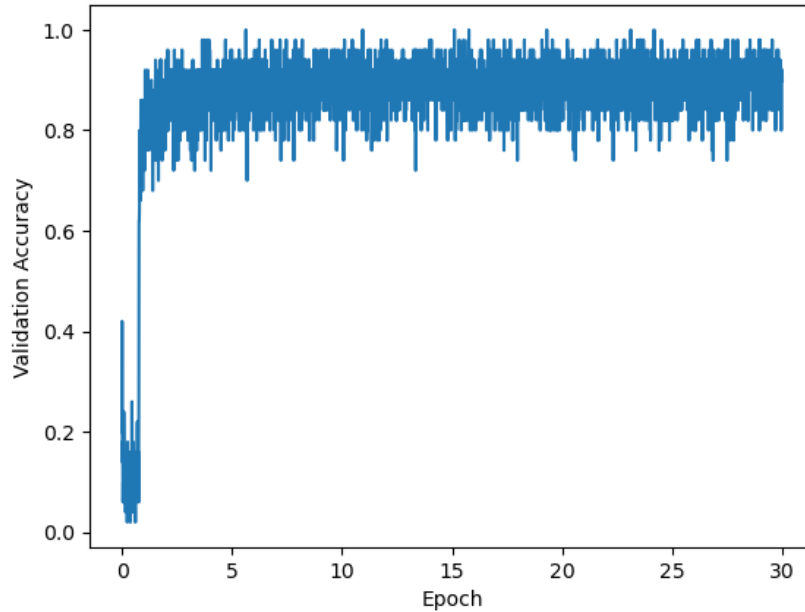


Figure 15: LR dropped at 7th epoch

Validation Accuracy Curve learning rate scheduling (rate drops at 7th and 14th epoch)

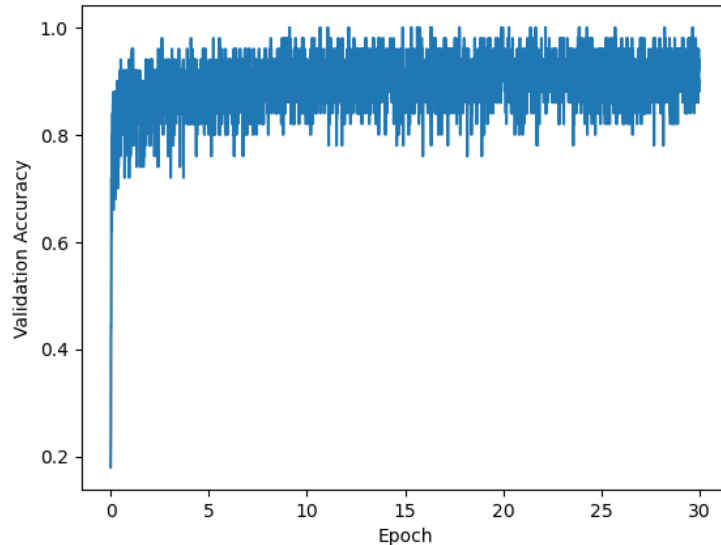


Figure 16: LR dropped at 7th and 15th epoch

Question 5.2 - Discussions

1. As can be observed from Figure 13, as the learning rate increases, the convergence speed also increases.

2. There are two points to consider when it comes to convergence to a better point. First, if the learning rate is too small, the weights may stuck at a suboptimal local minima. Also, even if the system convergence around the same minima, a small learning rate may lead to slow convergence. On the other hand, if the learning rate is too high, the weights may oscillate around the optimal point and may not go beyond a certain level. Therefore, the learning rate should be selected carefully. So, in order to find the best learning rate, one can start with a high learning rate and decrease it gradually.

3. In this case, the learning scheduling attempt worked out in terms of going to a better point. The oscillations around a convergence level are overcome by decreasing the learning rate.

4. When we compare these results with the one in question 3, where we trained the network with an ADAM optimizer, we can see that SGD yields a slightly better point. However, ADAM seems to converge faster than SGD. Yet, in this case, these two differences are quite subtle.

Appendix

The code set used throughout this homework is provided as follows. The code is written using VSCode and jupyter notebook infrastructure. The code is run on several different computational units that is why there are generic CUDA and CPU selection expressions. The code is converted from jupyter notebook to python script using the jupyter nbconvert command. At the end checkers used for q2 and a3 are also provided.

```
1 # %% [markdown]
2 # # Homework 1 Ahmet Akman
3 # This notebook consists the efforts led by homework 1 and done by Ahmet Akman in the
  scope of EE449 Course
4
5 # %% [markdown]
6 # ## Imports
7
8 # %%
9 import numpy as np
10 # import torch
11 import matplotlib.pyplot as plt
12 # import sklearn
13
14 # %% [markdown]
15 # ## Basic Neural Network Construction and Training
16 #
17
18 # %% [markdown]
19 # ### 1.1
20 #
21 #
22 #
23
24 # %%
25 x = np.arange(-2, 2, 0.01, dtype = float)
26 tanh_result = (np.exp(2*x)-1) / (np.exp(2*x)+1)
27 sigmoid_result = 1 / (np.exp(-x)+1)
28 ReLU_result = np.maximum(0,x)
29
30
31 fig, ax = plt.subplots(3)
32 ax[0].plot(x,tanh_result)
33 ax[0].set_title("Tanh function")
34 ax[0].set_xlabel("x")
35 ax[0].set_ylabel("tanh(x)")
36 ax[0].grid()
37
38 ax[1].plot(x,sigmoid_result)
39 ax[1].set_title("Sigmoid function")
40 ax[1].set_xlabel("x")
41 ax[1].set_ylabel("sigma(x)")
42 ax[1].grid()
43
44 ax[2].plot(x,ReLU_result)
```

```

45 ax[2].set_title("ReLU function")
46 ax[2].set_xlabel("x")
47 ax[2].set_ylabel("ReLU(x)")
48 ax[2].grid()
49 fig.tight_layout()
50
51
52 # %%
53 p_tanh_result = 1- (np.exp(x)-np.exp(-x))**2 / (np.exp(x) + np.exp(-x))**2
54 p_sigmoid_result = np.exp(-x) / (np.exp(-x)+1)**2
55 p_ReLU_result = x >= 0
56
57
58 fig, ax = plt.subplots(3)
59 ax[0].plot(x, p_tanh_result)
60 ax[0].set_title("Partial derivative of Tanh function")
61 ax[0].set_xlabel("x")
62 ax[0].set_ylabel("tanh(x)")
63 ax[0].grid()
64
65 ax[1].plot(x, p_sigmoid_result)
66 ax[1].set_title("Partial derivative of Sigmoid function")
67 ax[1].set_xlabel("x")
68 ax[1].set_ylabel("sigma(x)")
69 ax[1].grid()
70
71 ax[2].plot(x, p_ReLU_result)
72 ax[2].set_title("Partial derivative of ReLU function")
73 ax[2].set_xlabel("x")
74 ax[2].set_ylabel("ReLU(x)")
75 ax[2].grid()
76 fig.tight_layout()
77
78 # %% [markdown]
79 # ### 1.2
80 # - Sigmoid activated MLP.
81 #
82
83 # %%
84 np.random.seed(1234)
85 learning_rate_for_all_activation_functions = 0.00001
86
87 class MLP_sigmoid:
88     def __init__(self, input_size, hidden_size, output_size):
89         self.input_size = input_size
90         self.hidden_size = hidden_size
91         self.output_size = output_size
92
93         # Initialize weights and biases
94         self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
95         self.bias_hidden = np.zeros((1, self.hidden_size))
96         self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
97         self.bias_output = np.zeros((1, self.output_size))
98
99     def sigmoid(self, x):
100         return np.exp(x) / (np.exp(x)+1)
101     def sigmoid_derivative(self, x):
102         return np.exp(-x) / (np.exp(-x)+1)**2
103

```

```

104     def forward(self, inputs):
105         # Forward pass through the network
106         self.hidden_output = self.sigmoid(np.dot(inputs, self.weights_input_hidden) +
107         self.bias_hidden)
108         self.output = np.round(np.dot(self.hidden_output, self.weights_hidden_output) +
109         self.bias_output)
110         return self.output
111
112     def backward(self, inputs, targets, learning_rate):
113         # Backward pass through the network
114         # Compute error
115         output_error = targets - self.output
116         hidden_error = output_error * self.sigmoid_derivative(self.hidden_output)
117         # Compute gradients
118         output_delta = output_error * self.sigmoid_derivative(self.output)
119         hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
120         # Update weights and biases
121         self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(
122         self.hidden_output.T, output_delta)
123         self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis
124         =0, keepdims=True)
125         self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(
126         inputs.T, hidden_delta)
127         self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis
128         =0, keepdims=True)
129
130 from utils import part1CreateDataset, part1PlotBoundary
131 x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100,
132         std=0.4)
133
134 # Define neural network parameters
135 input_size = 2
136 hidden_size = 4
137 output_size = 1
138 learning_rate = learning_rate_for_all_activation_functions
139 # Create neural network
140 nn = MLP_sigmoid(input_size, hidden_size, output_size)
141 # Train the neural network
142 for epoch in range(10000):
143     # Forward propagation
144     output = nn.forward(x_train)
145     # Backpropagation
146     nn.backward(x_train, y_train, learning_rate)
147
148     # Print the loss (MSE) every 1000 epochs
149     if epoch % 1000 == 0:
150         loss = np.mean((y_train - output)**2)
151         print(f'Epoch {epoch}: Loss = {loss}')
152
153 # Test the trained neural network
154 y_predict = nn.forward(x_val)
155 print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
156
157 part1PlotBoundary(x_val, y_val, nn)
158
159 # %% [markdown]
160 # - Tanh activated MLP.

```

```

156
157 # %%
158 class MLP_tanh:
159     def __init__(self, input_size, hidden_size, output_size):
160         self.input_size = input_size
161         self.hidden_size = hidden_size
162         self.output_size = output_size
163
164         # Initialize weights and biases
165         self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
166         self.bias_hidden = np.zeros((1, self.hidden_size))
167         self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
168         self.bias_output = np.zeros((1, self.output_size))
169
170     def tanh(self, x):
171         return (np.exp(2*x)-1) / (np.exp(2*x)+1)
172     def tanh_derivative(self, x):
173         return 1- (np.exp(x)-np.exp(-x))*2 / (np.exp(x) + np.exp(-x))*2
174
175     def forward(self, inputs):
176         # Forward pass through the network
177         self.hidden_output = self.tanh(np.dot(inputs, self.weights_input_hidden) + self.bias_hidden)
178         self.output = np.round(np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output)
179         return self.output
180
181     def backward(self, inputs, targets, learning_rate):
182         # Backward pass through the network
183         # Compute error
184         output_error = targets - self.output
185         hidden_error = output_error * self.tanh_derivative(self.hidden_output)
186         # Compute gradients
187         output_delta = output_error * self.tanh_derivative(self.output)
188         hidden_delta = hidden_error * self.tanh_derivative(self.hidden_output)
189         # Update weights and biases
190         self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(self.hidden_output.T, output_delta)
191         self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis=0, keepdims=True)
192         self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(inputs.T, hidden_delta)
193         self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)
194
195 from utils import part1CreateDataset, part1PlotBoundary
196 x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100, std=0.4)
197
198 # Define neural network parameters
199 input_size = 2
200 hidden_size = 4
201 output_size = 1
202 learning_rate = learning_rate_for_all_activation_functions
203 # Create neural network
204 nn = MLP_tanh(input_size, hidden_size, output_size)
205 # Train the neural network
206 for epoch in range(10000):
207     # Forward propagation

```



```

208     output = nn.forward(x_train)
209     # Backpropagation
210     nn.backward(x_train, y_train, learning_rate)
211
212     # Print the loss (MSE) every 1000 epochs
213     if epoch % 1000 == 0:
214         loss = np.mean((y_train - output)**2)
215         print(f'Epoch {epoch}: Loss = {loss}')
216
217
218 # Test the trained neural network
219 y_predict = nn.forward(x_val)
220 print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
221
222
223 part1PlotBoundary(x_val, y_val, nn)
224
225 # %% [markdown]
226 # - ReLU Activated MLP
227 #
228
229 # %%
230 class MLP_ReLU:
231     def __init__(self, input_size, hidden_size, output_size):
232         self.input_size = input_size
233         self.hidden_size = hidden_size
234         self.output_size = output_size
235
236         # Initialize weights and biases
237         self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
238         self.bias_hidden = np.zeros((1, self.hidden_size))
239         self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)
240         self.bias_output = np.zeros((1, self.output_size))
241
242     def ReLU(self, x):
243         return np.maximum(0, x)
244     def ReLU_derivative(self, x):
245         return x >= 0
246
247     def forward(self, inputs):
248         # Forward pass through the network
249         self.hidden_output = self.ReLU(np.dot(inputs, self.weights_input_hidden) + self.bias_hidden)
250         self.output = np.round(np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output)
251         return self.output
252
253     def backward(self, inputs, targets, learning_rate):
254         # Backward pass through the network
255         # Compute error
256         output_error = targets - self.output
257         hidden_error = output_error * self.ReLU_derivative(self.hidden_output)
258         # Compute gradients
259         output_delta = output_error * self.ReLU_derivative(self.output)
260         hidden_delta = hidden_error * self.ReLU_derivative(self.hidden_output)
261         # Update weights and biases
262         self.weights_hidden_output = self.weights_hidden_output + learning_rate * np.dot(
            self.hidden_output.T, output_delta)

```

```

263     self.bias_output = self.bias_output + learning_rate * np.sum(output_delta, axis
264     =0, keepdims=True)
265     self.weights_input_hidden = self.weights_input_hidden + learning_rate * np.dot(
266     inputs.T, hidden_delta)
267     self.bias_hidden = self.bias_hidden + learning_rate * np.sum(hidden_delta, axis
268     =0, keepdims=True)
269
270 from utils import part1CreateDataset, part1PlotBoundary
271 x_train, y_train, x_val, y_val = part1CreateDataset(train_samples=1000, val_samples=100,
272     std=0.4)
273
274 # Define neural network parameters
275 input_size = 2
276 hidden_size = 4
277 output_size = 1
278 learning_rate = learning_rate_for_all_activation_functions
279 # Create neural network
280 nn = MLP_ReLU(input_size, hidden_size, output_size)
281 # Train the neural network
282 for epoch in range(10000):
283     # Forward propagation
284     output = nn.forward(x_train)
285     # Backpropagation
286     nn.backward(x_train, y_train, learning_rate)
287
288     # Print the loss (MSE) every 1000 epochs
289     if epoch % 1000 == 0:
290         loss = np.mean((y_train - output)**2)
291         print(f'Epoch {epoch}: Loss = {loss}')
292
293 # Test the trained neural network
294 y_predict = nn.forward(x_val)
295 print(f'{np.mean(y_predict==y_val)*100} % of test examples classified correctly.')
296
297 part1PlotBoundary(x_val, y_val, nn)
298
299 # %% [markdown]
300 # ### 1.3
301 #
302 # %% [markdown]
303 # ## Implementing a Convolutional Layer with NumPy
304 #
305 # %%
306
307 # implement a 2D convolutional layer using numpy
308 def my_conv2d(input, kernel):
309     # input shape: [batch size, input_channels, input_height, input_width]
310     # kernel shape: [output_channels, input_channels, filter_height, filter width]
311     batch_size, input_channels, input_height, input_width = input.shape
312     output_channels, input_channels, filter_height, filter_width = kernel.shape
313     # output shape: [batch size, output_channels, output_height, output_width]
314     output_height = input_height - filter_height + 1
315     output_width = input_width - filter_width + 1
316     output = np.zeros((batch_size, output_channels, output_height, output_width))
317     for b in range(batch_size):

```

```

318         for oc in range(output_channels):
319             for ic in range(input_channels):
320                 for i in range(output_height):
321                     for j in range(output_width):
322                         output[b, oc, i, j] = np.sum(input[b, ic, i:i+filter_height, j:j+
filter_width] * kernel[oc, ic])
323         return output
324
325 # input shape: [batch size, input_channels, input_height, input_width]
326 kernel=np.load('data/kernel.npy')
327
328 for i in range(10):
329     input=np.load('data/samples_{}.npy'.format(i))
330     # input shape: [output_channels, input_channels, filter_height, filter width]
331     out = my_conv2d(input, kernel)
332     out_check = torch.conv2d(torch.tensor(input).float(), torch.tensor(kernel).float())
333     np.save('outputs/out_{}.npy'.format(i), out)
334
335     from utils import part2Plots
336     part2Plots(out = out, save_dir='outputs', filename='out_{}'.format(i) )
337     part2Plots(out = out_check, save_dir='outputs', filename='out_{}_check'.format(i) )
338
339 # %% [markdown]
340 # ## 2.2
341 #
342
343 # %%
344 %reset -f
345
346 # %% [markdown]
347 # ## Experimenting ANN Architectures
348
349 # %%
350 # Load fashion MNIST dataset
351 import torchvision
352
353 #training set
354 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=True,
transform= torchvision.transforms.ToTensor(), shuffle=True)
355
356 #test set
357 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=True,
transform= torchvision.transforms.ToTensor())
358
359
360 # %%
361 import torch
362 # divide training data into training and validation sets of 0.8 and 0.2 respectively
363
364 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
365
366 train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
367 val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
368 test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)
369
370
371 # %% [markdown]
372 # ### mlp_1 case
373 # mlp_1 corresponds to FC-32, ReLU + FC10

```

```

374
375 # %%
376
377 from tqdm import tqdm
378
379 # example mlp classifier
380 class mlp_1(torch.nn.Module):
381     def __init__(self, input_size, hidden_size, num_classes):
382         super(mlp_1, self).__init__()
383         self.input_size = input_size
384         self.FC = torch.nn.Linear(input_size, hidden_size)
385         self.prediction_layer = torch.nn.Linear(hidden_size, num_classes)
386         self.relu = torch.nn.ReLU()
387     def forward(self, x):
388         x = x.view(-1, self.input_size)
389         hidden = self.FC(x)
390         relu = self.relu(hidden)
391         output = self.prediction_layer(relu)
392         return output
393
394 # initialize your model
395 model_mlp_1 = mlp_1(784,32,10)
396 if torch.cuda.is_available():
397     device = torch.device("cuda:0")
398     print("CUDA to be used")
399 else:
400     device = "cpu"
401     print("CPU to be used.")
402 model_mlp_1.to(device)
403 # create loss: use cross entropy loss
404 loss = torch.nn.CrossEntropyLoss()
405 # create optimizer
406 optimizer = torch.optim.Adam(model_mlp_1.parameters(), lr=0.001)
407 # transfer your model to train mode
408 model_mlp_1.train()
409
410
411 mlp_1_dict = {"name": "mlp_1", "loss_curve": [], "train_acc_curve": [], "val_acc_curve":
412             [], "test_acc": 0, 'weights': []}
413
414
415 # train the model and save the training loss and validation loss for every 10 batches
416 # using model eval mode.s
417 for epoch in tqdm(range(15)):
418     for batch, (x, y) in enumerate(train_generator):
419         x = x.to(device)
420         y = y.to(device)
421         # forward pass
422         output = model_mlp_1(x)
423         # compute loss
424         loss_val = loss(output, y)
425         # zero gradients
426         optimizer.zero_grad()
427         # backward pass
428         loss_val.backward()
429         # optimize
430         optimizer.step()
431         # print loss

```

```

431         if batch % 10 == 0:
432             model_mlp_1.eval()
433             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
434         )
435         mlp_1_dict['loss_curve'].append(loss_val.item())
436         training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
437         mlp_1_dict['train_acc_curve'].append(training_accuracy.item())
438
439         # validation loss
440         validation_accuracy_per_batch = torch.tensor([])
441         with torch.no_grad():
442             for val_x, val_y in val_generator:
443                 val_x = val_x.to(device)
444                 val_y = val_y.to(device)
445
446                 val_output = model_mlp_1(val_x)
447                 #val_loss = loss(val_output, val_y)
448                 val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
449 float())
450
451                 try:
452                     validation_accuracy_per_batch = torch.cat((
453 validation_accuracy_per_batch, val_accuracy))
454
455                 except:
456                     validation_accuracy_per_batch = val_accuracy
457
458                 #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
459
460             mlp_1_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
461 ())
462
463             model_mlp_1.train()
464
465
466 # test the model
467 correct = 0
468 total = 0
469 for x, y in test_generator:
470     x = x.to(device)
471     y = y.to(device)
472
473     output = model_mlp_1(x)
474     _, predicted = torch.max(output, 1)
475     total += y.size(0)
476     correct += (predicted == y).sum().item()
477
478 print('Test Accuracy: {} %'.format(100 * correct / total))
479 mlp_1_dict['test_acc'] = 100 * correct / total
480
481 # save the model as a pty file
482 torch.save(model_mlp_1.state_dict(), 'q3_models/model_mlp_1.pt')
483 print("model saved as 'q3_models/model_mlp_1.pt'")
484
485 # get the parameters 784x32 layer as numpy array
486 weights_first_layer = model_mlp_1.FC.weight.data.cpu().numpy()
487 mlp_1_dict['weights'] = weights_first_layer
488
489
490 # %%
491 # save mlp_1_dict as a pickle file
492 import pickle
493 with open('q3_models/mlp_1_dict.pkl', 'wb') as f:
494     pickle.dump(mlp_1_dict, f)
495

```

```

485
486 # %% [markdown]
487 # ### mlp_2 case
488 #
489
490 # %%
491 %reset -f
492
493 # Load fashion MNIST dataset
494 import torchvision
495
496 from tqdm import tqdm
497
498 #training set
499 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
        transform= torchvision.transforms.ToTensor(), shuffle=True)
500
501 #test set
502 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
        transform= torchvision.transforms.ToTensor())
503
504
505 import torch
506 # divide training data into training and validation sets of 0.8 and 0.2 respectively
507
508 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
509
510 train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
511 val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
512 test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)
513
514
515
516
517
518 # example mlp classifier
519 class mlp_2(torch.nn.Module):
520     def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
521         super(mlp_2, self).__init__()
522         self.input_size = input_size
523         self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
524         self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
525         self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
526         self.relu = torch.nn.ReLU()
527     def forward(self, x):
528         x = x.view(-1, self.input_size)
529         hidden1 = self.FC1(x)
530         relu = self.relu(hidden1)
531         hidden2 = self.FC2(relu)
532         output = self.prediction_layer(hidden2)
533         return output
534
535 # initialize your model
536 model_mlp_2 = mlp_2(784 ,32 ,64 ,10)
537
538 if torch.cuda.is_available():
539     device = torch.device("cuda:0")
540     print("CUDA to be used")
541 else:

```

```

542     device = "cpu"
543     print("CPU to be used.")
544 model_mlp_2.to(device)
545
546 # create loss: use cross entropy loss
547 loss = torch.nn.CrossEntropyLoss()
548 # create optimizer
549 optimizer = torch.optim.Adam(model_mlp_2.parameters(), lr=0.001)
550 # transfer your model to train mode
551 model_mlp_2.train()
552
553
554 mlp_2_dict = {"name": "mlp_2", "loss_curve": [], "train_acc_curve": [], "val_acc_curve":
    [], "test_acc": 0, 'weights': []}
555
556
557
558 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
559 for epoch in tqdm(range(15)):
560     for batch, (x, y) in enumerate(train_generator):
561         x = x.to(device)
562         y = y.to(device)
563
564         # forward pass
565         output = model_mlp_2(x)
566         # compute loss
567         loss_val = loss(output, y)
568         # zero gradients
569         optimizer.zero_grad()
570         # backward pass
571         loss_val.backward()
572         # optimize
573         optimizer.step()
574         # print loss
575         if batch % 10 == 0:
576             model_mlp_2.eval()
577             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
578         )
579
580         mlp_2_dict['loss_curve'].append(loss_val.item())
581         training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
582         mlp_2_dict['train_acc_curve'].append(training_accuracy.item())
583
584         # validation loss
585         validation_accuracy_per_batch = torch.tensor([])
586         with torch.no_grad():
587             for val_x, val_y in val_generator:
588                 val_x = val_x.to(device)
589                 val_y = val_y.to(device)
590
591                 val_output = model_mlp_2(val_x)
592                 #val_loss = loss(val_output, val_y)
593                 val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
594                 float())
595
596                 try:
597                     validation_accuracy_per_batch = torch.cat((
598                     validation_accuracy_per_batch ,val_accuracy))
599                 except:
600                     validation_accuracy_per_batch = val_accuracy

```

```

596         #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
597         mlp_2_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
598         model_mlp_2.train()
599
600
601
602 # test the model
603 correct = 0
604 total = 0
605 for x, y in test_generator:
606     x = x.to(device)
607     y = y.to(device)
608
609     output = model_mlp_2(x)
610     _, predicted = torch.max(output, 1)
611     total += y.size(0)
612     correct += (predicted == y).sum().item()
613 print('Test Accuracy: {} %'.format(100 * correct / total))
614 mlp_2_dict['test_acc'] = 100 * correct / total
615 # save the model as a pty file
616 torch.save(model_mlp_2.state_dict(), 'q3_models/model_mlp_2.pt')
617 print("model saved as 'q3_models/model_mlp_2.pt'")
618 # get the parameters 784x32 layer as numpy array
619 weights_first_layer = model_mlp_2.FC1.weight.data.cpu().numpy()
620 mlp_2_dict['weights'] = weights_first_layer
621
622
623 # save mlp_2_dict as a pickle file
624 import pickle
625 with open('q3_models/mlp_2_dict.pkl', 'wb') as f:
626     pickle.dump(mlp_2_dict, f)
627
628 # %% [markdown]
629 # ### CNN3
630 # Where Conv - 3x3x16 Relu
631 # Conv - 5x5x8 Relu, MaxPool 2x2
632 # Conv - 7x7x16 MaxPool 2x2
633 # Fully connected layer of 10.
634
635 # %%
636 %reset -f
637
638 # Load fashion MNIST dataset
639 import torchvision
640
641 from tqdm import tqdm
642
643 #training set
644 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
        transform= torchvision.transforms.ToTensor(), shuffle=True)
645
646 #test set
647 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
        transform= torchvision.transforms.ToTensor())
648
649
650 import torch

```



```

651 # divide training data into training and validation sets of 0.8 and 0.2 respectively
652
653 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
654
655 train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
656 val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
657 test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)
658
659
660
661
662
663 # example cnn_3 classifier
664 class cnn_3(torch.nn.Module):
665     def __init__(self, input_size, num_classes):
666         super(cnn_3, self).__init__()
667         self.input_size = input_size
668         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
669         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
670         self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
671         self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
672         self.prediction_layer = torch.nn.Linear(1296, num_classes)
673         self.relu = torch.nn.ReLU()
674     def forward(self, x):
675         x = x.view(-1, 1, 28, 28)
676         hidden1 = self.Conv1(x)
677         relu1 = self.relu(hidden1)
678         hidden2 = self.Conv2(relu1)
679         relu2 = self.relu(hidden2)
680         pool = self.MaxPool(relu2)
681         hidden3 = self.Conv3(pool)
682         flattened = hidden3.view(96, -1)
683         output = self.prediction_layer(flattened)
684         return output
685
686 # initialize your model
687 model_cnn_3 = cnn_3(784, 10)
688 if torch.cuda.is_available():
689     device = torch.device("cuda:0")
690     print("CUDA to be used")
691 else:
692     device = "cpu"
693     print("CPU to be used.")
694 model_cnn_3.to(device)
695
696 # create loss: use cross entropy loss
697 loss = torch.nn.CrossEntropyLoss()
698 # create optimizer
699 optimizer = torch.optim.Adam(model_cnn_3.parameters(), lr=0.001)
700 # transfer your model to train mode
701 model_cnn_3.train()
702
703
704 cnn_3_dict = {"name": "cnn_3", "loss_curve": [], "train_acc_curve": [], "val_acc_curve":
705              [], "test_acc": 0, 'weights': []}
706
707

```

```

708 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
709 for epoch in tqdm(range(15)):
710     for batch, (x, y) in enumerate(train_generator):
711
712         x = x.to(device)
713         y = y.to(device)
714
715         # forward pass
716         output = model_cnn_3(x)
717         # compute loss
718         loss_val = loss(output, y)
719         # zero gradients
720         optimizer.zero_grad()
721         # backward pass
722         loss_val.backward()
723         # optimize
724         optimizer.step()
725         # print loss
726         if batch % 10 == 0:
727             model_cnn_3.eval()
728             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
729             cnn_3_dict['loss_curve'].append(loss_val.item())
730             training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
731             cnn_3_dict['train_acc_curve'].append(training_accuracy.item())
732
733             # validation loss
734             validation_accuracy_per_batch = torch.tensor([])
735             with torch.no_grad():
736                 for val_x, val_y in val_generator:
737
738                     val_x = val_x.to(device)
739                     val_y = val_y.to(device)
740
741                     val_output = model_cnn_3(val_x)
742                     #val_loss = loss(val_output, val_y)
743                     val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
744                 try:
745                     validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch, val_accuracy))
746                 except:
747                     validation_accuracy_per_batch = val_accuracy
748                 #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
    )
749                 cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
    ())
750                 model_cnn_3.train()
751
752
753
754 # %%
755
756
757 # test the model
758 correct = 0
759 total = 0
760 for x, y in test_generator:

```

```

761     if y.size(dim = 0) < 96:
762         break
763     x = x.to(device)
764     y = y.to(device)
765
766     output = model_cnn_3(x)
767     _, predicted = torch.max(output, 1)
768     total += y.size(0)
769     correct += (predicted == y).sum().item()
770 print('Test Accuracy: {} %'.format(100 * correct / total))
771 cnn_3_dict['test_acc'] = 100 * correct / total
772 # save the model as a pty file
773 torch.save(model_cnn_3.state_dict(), 'q3_models/model_cnn_3.pt')
774 print("model saved as 'q3_models/model_cnn_3.pt'")
775 # get the parameters 784x32 layer as numpy array
776 weights_first_layer = model_cnn_3.Conv1.weight.data.cpu().numpy()
777 cnn_3_dict['weights'] = weights_first_layer
778
779 # save cnn_3 as a pickle file
780 import pickle
781 with open('q3_models/cnn_3_dict.pkl', 'wb') as f:
782     pickle.dump(cnn_3_dict, f)
783
784 # %% [markdown]
785 # ### CNN 4
786 #
787
788 # %%
789 %reset -f
790
791 # Load fashion MNIST dataset
792 import torchvision
793
794 from tqdm import tqdm
795
796 #training set
797 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
798         transform= torchvision.transforms.ToTensor(), shuffle=True)
799
800 #test set
801 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
802         transform= torchvision.transforms.ToTensor())
803
804 import torch
805
806 # divide training data into training and validation sets of 0.8 and 0.2 respectively
807
808 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
809
810 train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
811 val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
812 test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)
813
814
815
816 # example cnn_4 classifier
817 class cnn_4(torch.nn.Module):

```

```

818     def __init__(self, input_size, num_classes):
819         super(cnn_4, self).__init__()
820         self.input_size = input_size
821         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
822         self.Conv2 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
823         self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
824         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
825         self.Conv4 = torch.nn.Conv2d(16, 16, 5, stride=1, padding=1)
826         self.prediction_layer = torch.nn.Linear(400, num_classes)
827         self.relu = torch.nn.ReLU()
828     def forward(self, x):
829         x = x.view(-1, 1, 28, 28)
830         hidden1 = self.Conv1(x)
831         relu1 = self.relu(hidden1)
832         hidden2 = self.Conv2(relu1)
833         relu2 = self.relu(hidden2)
834         hidden3 = self.Conv3(relu2)
835         relu3 = self.relu(hidden3)
836         pool1 = self.MaxPool(relu3)
837         hidden4 = self.Conv4(pool1)
838         pool2 = self.MaxPool(hidden4)
839         flattened = pool2.view(96, -1)
840         output = self.prediction_layer(flattened)
841         return output
842
843 # initialize your model
844 model_cnn_4 = cnn_4(784, 10)
845
846 if torch.cuda.is_available():
847     device = torch.device("cuda:0")
848     print("CUDA to be used")
849 else:
850     device = "cpu"
851     print("CPU to be used.")
852 model_cnn_4.to(device)
853
854
855 # create loss: use cross entropy loss
856 loss = torch.nn.CrossEntropyLoss()
857 # create optimizer
858 optimizer = torch.optim.Adam(model_cnn_4.parameters(), lr=0.001)
859 # transfer your model to train mode
860 model_cnn_4.train()
861
862
863 cnn_4_dict = {"name": "cnn_4", "loss_curve": [], "train_acc_curve": [], "val_acc_curve":
864              [], "test_acc": 0, 'weights': []}
865
866
867 # train the model and save the training loss and validation loss for every 10 batches
868 # using model eval mode.s
869 for epoch in tqdm(range(15)):
870     for batch, (x, y) in enumerate(train_generator):
871
872         x = x.to(device)
873         y = y.to(device)
874
875         # forward pass

```

```

875     output = model_cnn_4(x)
876     # compute loss
877     loss_val = loss(output, y)
878     # zero gradients
879     optimizer.zero_grad()
880     # backward pass
881     loss_val.backward()
882     # optimize
883     optimizer.step()
884     # print loss
885     if batch % 10 == 0:
886         model_cnn_4.eval()
887         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
888     )
889     cnn_4_dict['loss_curve'].append(loss_val.item())
890     training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
891     cnn_4_dict['train_acc_curve'].append(training_accuracy.item())
892
893     # validation loss
894     validation_accuracy_per_batch = torch.tensor([])
895     with torch.no_grad():
896         for val_x, val_y in val_generator:
897
898             val_x = val_x.to(device)
899             val_y = val_y.to(device)
900
901             val_output = model_cnn_4(val_x)
902             #val_loss = loss(val_output, val_y)
903             val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
904 float())
905
906             try:
907                 validation_accuracy_per_batch = torch.cat((
908 validation_accuracy_per_batch ,val_accuracy))
909             except:
910                 validation_accuracy_per_batch = val_accuracy
911             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
912         )
913         cnn_4_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
914 ())
915         model_cnn_4.train()
916
917 # test the model
918 correct = 0
919 total = 0
920 for x, y in test_generator:
921     if y.size(dim = 0) < 96:
922         break
923     x = x.to(device)
924     y = y.to(device)
925
926     output = model_cnn_4(x)
927     _, predicted = torch.max(output, 1)
928     total += y.size(0)
929     correct += (predicted == y).sum().item()
930 print('Test Accuracy: {} %'.format(100 * correct / total))
931 cnn_4_dict['test_acc'] = 100 * correct / total
932 # save the model as a pty file

```

```

929 torch.save(model_cnn_4.state_dict(), 'q3_models/model_cnn_4.pt')
930 print("model saved as 'q3_models/model_cnn_4.pt'")
931 # get the parameters 784x32 layer as numpy array
932 weights_first_layer = model_cnn_4.Conv1.weight.data.cpu().numpy()
933 cnn_4_dict['weights'] = weights_first_layer
934
935 # save cnn_4 as a pickle file
936 import pickle
937 with open('q3_models/cnn_4_dict.pkl', 'wb') as f:
938     pickle.dump(cnn_4_dict, f)
939
940 # %% [markdown]
941 # ### CNN 5
942 #
943
944 # %%
945 %reset -f
946
947 # Load fashion MNIST dataset
948 import torchvision
949
950 from tqdm import tqdm
951
952 #training set
953 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
954     transform= torchvision.transforms.ToTensor(), shuffle=True)
955
956 #test set
957 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
958     transform= torchvision.transforms.ToTensor())
959
960
961 import torch
962 # divide training data into training and validation sets of 0.8 and 0.2 respectively
963
964 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
965
966 train_generator = torch.utils.data.DataLoader(train_data, batch_size=96, shuffle=True)
967 val_generator = torch.utils.data.DataLoader(val_data, batch_size=96, shuffle=False)
968 test_generator = torch.utils.data.DataLoader(test_data, batch_size=96, shuffle=False)
969
970
971
972 # example cnn_5 classifier
973 class cnn_5(torch.nn.Module):
974     def __init__(self, input_size, num_classes):
975         super(cnn_5, self).__init__()
976         self.input_size = input_size
977         self.Conv1 = torch.nn.Conv2d(1, 8, 3, stride=1, padding=1)
978         self.Conv2 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
979         self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
980         self.Conv4 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
981         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
982         self.Conv5 = torch.nn.Conv2d(16, 16, 3, stride=1, padding=1)
983         self.Conv6 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
984         self.prediction_layer = torch.nn.Linear(392, num_classes)
985         self.relu = torch.nn.ReLU()

```

```

986     def forward(self, x):
987         x = x.view(-1, 1, 28, 28)
988         hidden1 = self.Conv1(x)
989         relu1 = self.relu(hidden1)
990         hidden2 = self.Conv2(relu1)
991         relu2 = self.relu(hidden2)
992         hidden3 = self.Conv3(relu2)
993         relu3 = self.relu(hidden3)
994         hidden4 = self.Conv4(relu3)
995         relu4 = self.relu(hidden4)
996         pool1 = self.MaxPool(relu4)
997         hidden5 = self.Conv5(pool1)
998         relu5 = self.relu(hidden5)
999         hidden6 = self.Conv6(relu5)
1000        relu6 = self.relu(hidden6)
1001        pool2 = self.MaxPool(relu6)
1002        flattened = pool2.view(96, -1)
1003        output = self.prediction_layer(flattened)
1004        return output
1005
1006 # initialize your model
1007 model_cnn_5 = cnn_5(784 ,10)
1008
1009 if torch.cuda.is_available():
1010     device = torch.device("cuda:0")
1011     print("CUDA to be used")
1012 else:
1013     device = "cpu"
1014     print("CPU to be used.")
1015 model_cnn_5.to(device)
1016
1017
1018 # create loss: use cross entropy loss
1019 loss = torch.nn.CrossEntropyLoss()
1020 # create optimizer
1021 optimizer = torch.optim.Adam(model_cnn_5.parameters(), lr=0.001)
1022 # transfer your model to train mode
1023 model_cnn_5.train()
1024
1025
1026 cnn_5_dict = {"name": "cnn_5", "loss_curve": [], "train_acc_curve": [], "val_acc_curve":
1027              [], "test_acc": 0, 'weights': []}
1028
1029
1030 # train the model and save the training loss and validation loss for every 10 batches
1031 # using model eval mode.s
1032 for epoch in tqdm(range(15)):
1033     for batch, (x, y) in enumerate(train_generator):
1034
1035         x = x.to(device)
1036         y = y.to(device)
1037
1038         # forward pass
1039         output = model_cnn_5(x)
1040         # compute loss
1041         loss_val = loss(output, y)
1042         # zero gradients
1043         optimizer.zero_grad()

```

```

1043     # backward pass
1044     loss_val.backward()
1045     # optimize
1046     optimizer.step()
1047     # print loss
1048     if batch % 10 == 0:
1049         model_cnn_5.eval()
1050         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1051     )
1052     cnn_5_dict['loss_curve'].append(loss_val.item())
1053     training_accuracy = torch.mean((torch.argmax(output, dim=1) == y).float())
1054     cnn_5_dict['train_acc_curve'].append(training_accuracy.item())
1055
1056     # validation loss
1057     validation_accuracy_per_batch = torch.tensor([])
1058     with torch.no_grad():
1059         for val_x, val_y in val_generator:
1060             val_x = val_x.to(device)
1061             val_y = val_y.to(device)
1062
1063             val_output = model_cnn_5(val_x)
1064             #val_loss = loss(val_output, val_y)
1065             val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
1066
1067             try:
1068                 validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch, val_accuracy))
1069             except:
1070                 validation_accuracy_per_batch = val_accuracy
1071             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
)
1072             cnn_5_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
())
1073             model_cnn_5.train()
1074
1075
1076 # test the model
1077 correct = 0
1078 total = 0
1079 for x, y in test_generator:
1080     if y.size(dim = 0) < 96:
1081         break
1082     x = x.to(device)
1083     y = y.to(device)
1084
1085     output = model_cnn_5(x)
1086     _, predicted = torch.max(output, 1)
1087     total += y.size(0)
1088     correct += (predicted == y).sum().item()
1089 print('Test Accuracy: {} %'.format(100 * correct / total))
1090 cnn_5_dict['test_acc'] = 100 * correct / total
1091 # save the model as a pty file
1092 torch.save(model_cnn_5.state_dict(), 'q3_models/model_cnn_5.pt')
1093 print("model saved as 'q3_models/model_cnn_5.pt'")
1094 # get the parameters 784x32 layer as numpy array
1095 weights_first_layer = model_cnn_5.Conv1.weight.data.cpu().numpy()
1096 cnn_5_dict['weights'] = weights_first_layer

```



```

1097
1098 # save cnn_5 as a pickle file
1099 import pickle
1100 with open('q3_models/cnn_5_dict.pkl', 'wb') as f:
1101     pickle.dump(cnn_5_dict, f)
1102
1103 # %%
1104 %reset -f
1105
1106 import pickle
1107
1108 results = []
1109 with open('q3_models/mlp_1_dict.pkl', 'rb') as f:
1110     data = pickle.load(f)
1111
1112 results.append(data)
1113 print(data.keys())
1114
1115 with open('q3_models/mlp_2_dict.pkl', 'rb') as f:
1116     data = pickle.load(f)
1117 results.append(data)
1118 print(data.keys())
1119
1120
1121 with open('q3_models/cnn_3_dict.pkl', 'rb') as f:
1122     data = pickle.load(f)
1123 results.append(data)
1124 print(data.keys())
1125 with open('q3_models/cnn_4_dict.pkl', 'rb') as f:
1126     data = pickle.load(f)
1127 results.append(data)
1128 print(data.keys())
1129
1130 with open('q3_models/cnn_5_dict.pkl', 'rb') as f:
1131     data = pickle.load(f)
1132 results.append(data)
1133 print(data.keys())
1134
1135
1136
1137 from utils import part3Plots, visualizeWeights
1138 # To plot the curves
1139 part3Plots(results, save_dir='q3_models', filename='q3_results')
1140
1141 # To plot the weights
1142 for i in range(len(results)):
1143     weights = results[i]['weights']
1144     visualizeWeights(weights, save_dir='q3_models', filename='weights_'+results[i]['name']
1145                      ])
1146
1147 # %% [markdown]
1148 # ## Experimenting Activation Functions
1149
1150 # %% [markdown]
1151 # ### This part will consist of several activation function comparison trainings.
1152 # First element will be the training of mlp_1 with SGD and ReLU
1153 #
1154 # %%

```

```

1155 %reset -f
1156
1157 # Load fashion MNIST dataset
1158 import torchvision
1159
1160 from tqdm import tqdm
1161
1162 #training set
1163 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
1164                                                transform= torchvision.transforms.ToTensor())
1165
1166 #test set
1167 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
1168                                                transform= torchvision.transforms.ToTensor())
1169
1170 import torch
1171 # divide training data into training and validation sets of 0.8 and 0.2 respectively
1172 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
1173
1174 train_generator = torch.utils.data.DataLoader(train_data, batch_size=50, shuffle=True)
1175 val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)
1176 test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)
1177
1178
1179
1180
1181 # ReLU mlp classifier
1182 class mlp_1_ReLU(torch.nn.Module):
1183     def __init__(self, input_size, hidden_size, num_classes):
1184         super(mlp_1_ReLU, self).__init__()
1185         self.input_size = input_size
1186         self.FC = torch.nn.Linear(input_size, hidden_size)
1187         self.prediction_layer = torch.nn.Linear(hidden_size, num_classes)
1188         self.relu = torch.nn.ReLU()
1189     def forward(self, x):
1190         x = x.view(-1, self.input_size)
1191         hidden = self.FC(x)
1192         relu = self.relu(hidden)
1193         output = self.prediction_layer(relu)
1194         return output
1195
1196 # Sigmoid mlp classifier
1197 class mlp_1_Sigmoid(torch.nn.Module):
1198     def __init__(self, input_size, hidden_size, num_classes):
1199         super(mlp_1_Sigmoid, self).__init__()
1200         self.input_size = input_size
1201         self.FC = torch.nn.Linear(input_size, hidden_size)
1202         self.prediction_layer = torch.nn.Linear(hidden_size, num_classes)
1203         self.sigmoid = torch.nn.Sigmoid()
1204     def forward(self, x):
1205         x = x.view(-1, self.input_size)
1206         hidden = self.FC(x)
1207         sigmoid = self.sigmoid(hidden)
1208         output = self.prediction_layer(sigmoid)
1209         return output
1210
1211

```

```

1212
1213
1214 # initialize the relu model
1215 model_mlp_1_relu = mlp_1_ReLU(784,32,10)
1216 # initialize the sigmoid model
1217 model_mlp_1_sigmoid = mlp_1_Sigmoid(784,32,10)
1218
1219
1220 if torch.cuda.is_available():
1221     device = torch.device("cuda:0")
1222     print("CUDA to be used")
1223 else:
1224     device = "cpu"
1225     print("CPU to be used.")
1226
1227 model_mlp_1_relu.to(device)
1228 model_mlp_1_sigmoid.to(device)
1229
1230 # create loss: use cross entropy loss
1231 loss = torch.nn.CrossEntropyLoss()
1232
1233 # create optimizer
1234 optimizer_relu = torch.optim.SGD(model_mlp_1_relu.parameters(), lr=0.01)
1235 optimizer_sigmoid = torch.optim.SGD(model_mlp_1_sigmoid.parameters(), lr=0.01)
1236
1237
1238 # transfer your model to train mode
1239 model_mlp_1_relu.train()
1240 model_mlp_1_sigmoid.train()
1241
1242 mlp_1_dict = {"name": "mlp_1", "relu_loss_curve": [], "sigmoid_loss_curve": [], "
               relu_grad_curve": [], "sigmoid_grad_curve": []}
1243
1244
1245
1246 # train the model and save the training loss and validation loss for every 10 batches
   using model eval mode.s
1247 for epoch in tqdm(range(15)):
1248     for batch, (x, y) in enumerate(train_generator):
1249         x = x.to(device)
1250         y = y.to(device)
1251         # forward pass
1252         output = model_mlp_1_relu(x)
1253         # compute loss
1254         loss_val = loss(output, y)
1255         # zero gradients
1256         optimizer_relu.zero_grad()
1257         # backward pass
1258         loss_val.backward()
1259         # optimize
1260         optimizer_relu.step()
1261         # print loss
1262         if batch % 10 == 0:
1263             model_mlp_1_relu.eval()
1264             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
   )
1265             mlp_1_dict['relu_loss_curve'].append(loss_val.item())
1266             mlp_1_dict['relu_grad_curve'].append(model_mlp_1_relu.FC.weight.grad.mean().
   item())

```

```

1267
1268         model_mlp_1_relu.train()
1269
1270
1271
1272
1273 # save the model as a pty file
1274 torch.save(model_mlp_1_relu.state_dict(), 'q4_models/model_mlp_1_relu.pt')
1275 print("model saved as 'q4_models/model_mlp_1_relu.pt'")
1276
1277 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
1278 for epoch in tqdm(range(15)):
1279     for batch, (x, y) in enumerate(train_generator):
1280         x = x.to(device)
1281         y = y.to(device)
1282         # forward pass
1283         output = model_mlp_1_sigmoid(x)
1284         # compute loss
1285         loss_val = loss(output, y)
1286         # zero gradients
1287         optimizer_sigmoid.zero_grad()
1288         # backward pass
1289         loss_val.backward()
1290         # optimize
1291         optimizer_sigmoid.step()
1292         # print loss
1293         if batch % 10 == 0:
1294             model_mlp_1_sigmoid.eval()
1295             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
        )
1296             mlp_1_dict['sigmoid_loss_curve'].append(loss_val.item())
1297             mlp_1_dict['sigmoid_grad_curve'].append(model_mlp_1_sigmoid.FC.weight.grad.
        mean().item())
1298
1299         model_mlp_1_sigmoid.train()
1300
1301
1302
1303
1304 # save the model as a pty file
1305 torch.save(model_mlp_1_sigmoid.state_dict(), 'q4_models/model_mlp_1_sigmoid.pt')
1306 print("model saved as 'q4_models/model_mlp_1_sigmoid.pt'")
1307
1308
1309
1310 # save mlp_1_dict as a pickle file
1311 import pickle
1312 with open('q4_models/part4_mlp_1_dict.pkl', 'wb') as f:
1313     pickle.dump(mlp_1_dict, f)
1314
1315 # %% [markdown]
1316 # ### MLP2
1317
1318 # %%
1319 %reset -f
1320
1321 # Load fashion MNIST dataset
1322 import torchvision

```

```

1323
1324 from tqdm import tqdm
1325
1326 #training set
1327 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
1328         transform= torchvision.transforms.ToTensor())
1329
1330 #test set
1331 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
1332         transform= torchvision.transforms.ToTensor())
1333
1334 import torch
1335 # divide training data into training and validation sets of 0.8 and 0.2 respectively
1336
1337 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
1338
1339 train_generator = torch.utils.data.DataLoader(train_data, batch_size=50, shuffle=True)
1340 val_generator = torch.utils.data.DataLoader(val_data, batch_size=50, shuffle=False)
1341 test_generator = torch.utils.data.DataLoader(test_data, batch_size=50, shuffle=False)
1342
1343
1344
1345 # ReLU mlp classifier
1346 class mlp_2_ReLU(torch.nn.Module):
1347     def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
1348         super(mlp_2_ReLU, self).__init__()
1349         self.input_size = input_size
1350         self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
1351         self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
1352         self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
1353         self.relu = torch.nn.ReLU()
1354     def forward(self, x):
1355         x = x.view(-1, self.input_size)
1356         hidden1 = self.FC1(x)
1357         relu = self.relu(hidden1)
1358         hidden2 = self.FC2(relu)
1359         output = self.prediction_layer(hidden2)
1360         return output
1361
1362
1363 # Sigmoid mlp classifier
1364 class mlp_2_Sigmoid(torch.nn.Module):
1365     def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
1366         super(mlp_2_Sigmoid, self).__init__()
1367         self.input_size = input_size
1368         self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
1369         self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
1370         self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
1371         self.sigmoid = torch.nn.Sigmoid()
1372     def forward(self, x):
1373         x = x.view(-1, self.input_size)
1374         hidden1 = self.FC1(x)
1375         sigmoid = self.sigmoid(hidden1)
1376         hidden2 = self.FC2(sigmoid)
1377         output = self.prediction_layer(hidden2)
1378         return output
1379

```

```

1380
1381
1382 # initialize the relu model
1383 model_mlp_2_relu = mlp_2_ReLU(784 ,32 ,64 ,10)
1384 # initialize the sigmoid model
1385 model_mlp_2_sigmoid = mlp_2_Sigmoid(784 ,32 ,64 ,10)
1386
1387
1388 if torch.cuda.is_available():
1389     device = torch.device("cuda:0")
1390     print("CUDA to be used")
1391 else:
1392     device = "cpu"
1393     print("CPU to be used.")
1394
1395 model_mlp_2_relu.to(device)
1396 model_mlp_2_sigmoid.to(device)
1397
1398 # create loss: use cross entropy loss
1399 loss = torch.nn.CrossEntropyLoss()
1400
1401 # create optimizer
1402 optimizer_relu = torch.optim.SGD(model_mlp_2_relu.parameters(), lr=0.01)
1403 optimizer_sigmoid = torch.optim.SGD(model_mlp_2_sigmoid.parameters(), lr=0.01)
1404
1405
1406 # transfer your model to train mode
1407 model_mlp_2_relu.train()
1408 model_mlp_2_sigmoid.train()
1409
1410 mlp_2_dict = {"name": "mlp_2", "relu_loss_curve": [], "sigmoid_loss_curve": [], "
               relu_grad_curve": [], "sigmoid_grad_curve": []}
1411
1412
1413
1414 # train the model and save the training loss and validation loss for every 10 batches
   using model eval mode.s
1415 for epoch in tqdm(range(15)):
1416     for batch, (x, y) in enumerate(train_generator):
1417         x = x.to(device)
1418         y = y.to(device)
1419         # forward pass
1420         output = model_mlp_2_relu(x)
1421         # compute loss
1422         loss_val = loss(output, y)
1423         # zero gradients
1424         optimizer_relu.zero_grad()
1425         # backward pass
1426         loss_val.backward()
1427         # optimize
1428         optimizer_relu.step()
1429         # print loss
1430         if batch % 10 == 0:
1431             model_mlp_2_relu.eval()
1432             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
   )
1433             mlp_2_dict['relu_loss_curve'].append(loss_val.item())
1434             mlp_2_dict['relu_grad_curve'].append(model_mlp_2_relu.FC1.weight.grad.mean().
   item())

```

```

1435
1436         model_mlp_2_relu.train()
1437
1438
1439
1440
1441 # save the model as a pty file
1442 torch.save(model_mlp_2_relu.state_dict(), 'q4_models/model_mlp_2_relu.pt')
1443 print("model saved as 'q4_models/model_mlp_2_relu.pt'")
1444
1445 # train the model and save the training loss and validation loss for every 10 batches
1446     using model eval mode.s
1447 for epoch in tqdm(range(15)):
1448     for batch, (x, y) in enumerate(train_generator):
1449         x = x.to(device)
1450         y = y.to(device)
1451         # forward pass
1452         output = model_mlp_2_sigmoid(x)
1453         # compute loss
1454         loss_val = loss(output, y)
1455         # zero gradients
1456         optimizer_sigmoid.zero_grad()
1457         # backward pass
1458         loss_val.backward()
1459         # optimize
1460         optimizer_sigmoid.step()
1461         # print loss
1462         if batch % 10 == 0:
1463             model_mlp_2_sigmoid.eval()
1464             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1465             )
1466             mlp_2_dict['sigmoid_loss_curve'].append(loss_val.item())
1467             mlp_2_dict['sigmoid_grad_curve'].append(model_mlp_2_sigmoid.FC1.weight.grad.
1468             mean().item())
1469
1470         model_mlp_2_sigmoid.train()
1471
1472
1473
1474 # save the model as a pty file
1475 torch.save(model_mlp_2_sigmoid.state_dict(), 'q4_models/model_mlp_2_sigmoid.pt')
1476 print("model saved as 'q4_models/model_mlp_2_sigmoid.pt'")
1477
1478
1479
1480 # save mlp_2_dict as a pickle file
1481 import pickle
1482 with open('q4_models/part4_mlp_2_dict.pkl', 'wb') as f:
1483     pickle.dump(mlp_2_dict, f)
1484
1485
1486 # %% [markdown]
1487 # ### CNN3 Activation Function Experiment
1488
1489 # %%
1490 %reset -f
1491
1492 # Load fashion MNIST dataset
1493 import torchvision

```

```

1491
1492 from tqdm import tqdm
1493
1494 #training set
1495 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
1496         transform= torchvision.transforms.ToTensor())
1497
1498 #test set
1499 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
1500         transform= torchvision.transforms.ToTensor())
1501
1502
1503
1504 batch_size = 50
1505
1506
1507 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
1508
1509 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
1510         True)
1511 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
1512         False)
1513 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
1514         False)
1515
1516
1517
1518 # cnn_3 classifier
1519 class cnn_3_relu(torch.nn.Module):
1520     def __init__(self, input_size, num_classes):
1521         super(cnn_3_relu, self).__init__()
1522         self.input_size = input_size
1523         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
1524         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
1525         self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
1526         self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
1527         self.prediction_layer = torch.nn.Linear(1296, num_classes)
1528         self.relu = torch.nn.ReLU()
1529     def forward(self, x):
1530         x = x.view(-1, 1, 28, 28)
1531         hidden1 = self.Conv1(x)
1532         relu1 = self.relu(hidden1)
1533         hidden2 = self.Conv2(relu1)
1534         relu2 = self.relu(hidden2)
1535         pool = self.MaxPool(relu2)
1536         hidden3 = self.Conv3(pool)
1537         flattened = hidden3.view(batch_size, -1)
1538         output = self.prediction_layer(flattened)
1539         return output
1540
1541
1542 # sigmoid cnn_3 classifier
1543 class cnn_3_sigmoid(torch.nn.Module):
1544     def __init__(self, input_size, num_classes):
1545         super(cnn_3_sigmoid, self).__init__()
1546         self.input_size = input_size
1547         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
1548         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)

```



```

1545     self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
1546     self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
1547     self.prediction_layer = torch.nn.Linear(1296, num_classes)
1548     self.sigmoid = torch.nn.Sigmoid()
1549     def forward(self, x):
1550         x = x.view(-1, 1, 28, 28)
1551         hidden1 = self.Conv1(x)
1552         sigmoid1 = self.sigmoid(hidden1)
1553         hidden2 = self.Conv2(sigmoid1)
1554         sigmoid2 = self.sigmoid(hidden2)
1555         pool = self.MaxPool(sigmoid2)
1556         hidden3 = self.Conv3(pool)
1557         flattened = hidden3.view(batch_size, -1)
1558         output = self.prediction_layer(flattened)
1559         return output
1560
1561
1562 # initialize cnn_3 relu model
1563 model_cnn_3_relu = cnn_3_relu(784, 10)
1564 # initialize cnn_3 sigmoid model
1565 model_cnn_3_sigmoid = cnn_3_sigmoid(784, 10)
1566
1567
1568 if torch.cuda.is_available():
1569     device = torch.device("cuda:0")
1570     print("CUDA to be used")
1571 else:
1572     device = "cpu"
1573     print("CPU to be used.")
1574
1575 model_cnn_3_relu.to(device)
1576 model_cnn_3_sigmoid.to(device)
1577
1578 # create loss: use cross entropy loss
1579 loss = torch.nn.CrossEntropyLoss()
1580
1581 # create optimizer
1582 optimizer_relu = torch.optim.SGD(model_cnn_3_relu.parameters(), lr=0.01)
1583 optimizer_sigmoid = torch.optim.SGD(model_cnn_3_sigmoid.parameters(), lr=0.01)
1584
1585
1586 # transfer your model to train mode
1587 model_cnn_3_relu.train()
1588 model_cnn_3_sigmoid.train()
1589
1590 cnn_3_dict = {"name": "cnn_3", "relu_loss_curve": [], "sigmoid_loss_curve": [], "
1591              relu_grad_curve": [], "sigmoid_grad_curve": []}
1592
1593
1594 # train the model and save the training loss and validation loss for every 10 batches
1595 # using model eval mode.s
1596 for epoch in tqdm(range(15)):
1597     for batch, (x, y) in enumerate(train_generator):
1598         x = x.to(device)
1599         y = y.to(device)
1600         # forward pass
1601         output = model_cnn_3_relu(x)
1602         # compute loss

```

```

1602     loss_val = loss(output, y)
1603     # zero gradients
1604     optimizer_relu.zero_grad()
1605     # backward pass
1606     loss_val.backward()
1607     # optimize
1608     optimizer_relu.step()
1609     # print loss
1610     if batch % 10 == 0:
1611         model_cnn_3_relu.eval()
1612         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1613     )
1614     cnn_3_dict['relu_loss_curve'].append(loss_val.item())
1615     cnn_3_dict['relu_grad_curve'].append(model_cnn_3_relu.Conv1.weight.grad.mean
1616     ().item())
1617
1618     model_cnn_3_relu.train()
1619
1620
1621 # save the model as a pty file
1622 torch.save(model_cnn_3_relu.state_dict(), 'q4_models/model_cnn_3_relu.pt')
1623 print("model saved as 'q4_models/model_cnn_3_relu.pt'")
1624
1625 # train the model and save the training loss and validation loss for every 10 batches
1626 # using model eval mode.s
1627 for epoch in tqdm(range(15)):
1628     for batch, (x, y) in enumerate(train_generator):
1629         x = x.to(device)
1630         y = y.to(device)
1631         # forward pass
1632         output = model_cnn_3_sigmoid(x)
1633         # compute loss
1634         loss_val = loss(output, y)
1635         # zero gradients
1636         optimizer_sigmoid.zero_grad()
1637         # backward pass
1638         loss_val.backward()
1639         # optimize
1640         optimizer_sigmoid.step()
1641         # print loss
1642         if batch % 10 == 0:
1643             model_cnn_3_sigmoid.eval()
1644             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1645         )
1646             cnn_3_dict['sigmoid_loss_curve'].append(loss_val.item())
1647             cnn_3_dict['sigmoid_grad_curve'].append(model_cnn_3_sigmoid.Conv1.weight.grad
1648             .mean().item())
1649
1650             model_cnn_3_sigmoid.train()
1651
1652
1653 # save the model as a pty file
1654 torch.save(model_cnn_3_sigmoid.state_dict(), 'q4_models/model_cnn_3_sigmoid.pt')
1655 print("model saved as 'q4_models/model_cnn_3_sigmoid.pt'")

```

```

1656
1657
1658 # save cnn_3_dict as a pickle file
1659 import pickle
1660 with open('q4_models/part4_cnn_3_dict.pkl', 'wb') as f:
1661     pickle.dump(cnn_3_dict, f)
1662
1663 # %% [markdown]
1664 # ### CNN4 Activation function experiment
1665
1666 # %%
1667 %reset -f
1668
1669 # Load fashion MNIST dataset
1670 import torchvision
1671
1672 from tqdm import tqdm
1673
1674 #training set
1675 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
1676     transform= torchvision.transforms.ToTensor())
1677
1678 #test set
1679 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
1680     transform= torchvision.transforms.ToTensor())
1681
1682 batch_size = 50
1683
1684 import torch
1685 # divide training data into training and validation sets of 0.8 and 0.2 respectively
1686
1687 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
1688
1689 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
1690     True)
1691 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
1692     False)
1693 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
1694     False)
1695
1696 # relu cnn_4 classifier
1697 class cnn_4_relu(torch.nn.Module):
1698     def __init__(self, input_size, num_classes):
1699         super(cnn_4_relu, self).__init__()
1700         self.input_size = input_size
1701         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
1702         self.Conv2 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1703         self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
1704         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
1705         self.Conv4 = torch.nn.Conv2d(16, 16, 5, stride=1, padding=1)
1706         self.prediction_layer = torch.nn.Linear(400, num_classes)
1707         self.relu = torch.nn.ReLU()
1708     def forward(self, x):
1709         x = x.view(-1, 1, 28, 28)
1710         hidden1 = self.Conv1(x)
1711         relu1 = self.relu(hidden1)
1712         hidden2 = self.Conv2(relu1)
1713         relu2 = self.relu(hidden2)
1714         hidden3 = self.Conv3(relu2)

```

```

1710     relu3 = self.relu(hidden3)
1711     pool1 = self.MaxPool(relu3)
1712     hidden4 = self.Conv4(pool1)
1713     pool2 = self.MaxPool(hidden4)
1714     flattened = pool2.view(batch_size, -1)
1715     output = self.prediction_layer(flattened)
1716     return output
1717
1718
1719 # sigmoid cnn_4 classifier
1720 class cnn_4_sigmoid(torch.nn.Module):
1721     def __init__(self, input_size, num_classes):
1722         super(cnn_4_sigmoid, self).__init__()
1723         self.input_size = input_size
1724         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
1725         self.Conv2 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1726         self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
1727         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
1728         self.Conv4 = torch.nn.Conv2d(16, 16, 5, stride=1, padding=1)
1729         self.prediction_layer = torch.nn.Linear(400, num_classes)
1730         self.sigmoid = torch.nn.Sigmoid()
1731     def forward(self, x):
1732         x = x.view(-1, 1, 28, 28)
1733         hidden1 = self.Conv1(x)
1734         sigmoid1 = self.sigmoid(hidden1)
1735         hidden2 = self.Conv2(sigmoid1)
1736         sigmoid2 = self.sigmoid(hidden2)
1737         hidden3 = self.Conv3(sigmoid2)
1738         sigmoid3 = self.sigmoid(hidden3)
1739         pool1 = self.MaxPool(sigmoid3)
1740         hidden4 = self.Conv4(pool1)
1741         pool2 = self.MaxPool(hidden4)
1742         flattened = pool2.view(batch_size, -1)
1743         output = self.prediction_layer(flattened)
1744         return output
1745
1746
1747
1748
1749 # initialize cnn_4 relu model
1750 model_cnn_4_relu = cnn_4_relu(784, 10)
1751 # initialize cnn_4 sigmoid model
1752 model_cnn_4_sigmoid = cnn_4_sigmoid(784, 10)
1753
1754
1755 if torch.cuda.is_available():
1756     device = torch.device("cuda:0")
1757     print("CUDA to be used")
1758 else:
1759     device = "cpu"
1760     print("CPU to be used.")
1761
1762 model_cnn_4_relu.to(device)
1763 model_cnn_4_sigmoid.to(device)
1764
1765 # create loss: use cross entropy loss
1766 loss = torch.nn.CrossEntropyLoss()
1767
1768 # create optimizer

```

```

1769 optimizer_relu = torch.optim.SGD(model_cnn_4_relu.parameters(), lr=0.01)
1770 optimizer_sigmoid = torch.optim.SGD(model_cnn_4_sigmoid.parameters(), lr=0.01)
1771
1772
1773 # transfer your model to train mode
1774 model_cnn_4_relu.train()
1775 model_cnn_4_sigmoid.train()
1776
1777 cnn_4_dict = {"name": "cnn_4", "relu_loss_curve": [], "sigmoid_loss_curve": [], "
1778               relu_grad_curve": [], "sigmoid_grad_curve": []}
1779
1780
1781 # train the model and save the training loss and validation loss for every 10 batches
1782   using model eval mode.s
1783 for epoch in tqdm(range(15)):
1784     for batch, (x, y) in enumerate(train_generator):
1785         x = x.to(device)
1786         y = y.to(device)
1787         # forward pass
1788         output = model_cnn_4_relu(x)
1789         # compute loss
1790         loss_val = loss(output, y)
1791         # zero gradients
1792         optimizer_relu.zero_grad()
1793         # backward pass
1794         loss_val.backward()
1795         # optimize
1796         optimizer_relu.step()
1797         # print loss
1798         if batch % 10 == 0:
1799             model_cnn_4_relu.eval()
1800             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1801             )
1802             cnn_4_dict['relu_loss_curve'].append(loss_val.item())
1803             cnn_4_dict['relu_grad_curve'].append(model_cnn_4_relu.Conv1.weight.grad.mean
1804             ().item())
1805
1806             model_cnn_4_relu.train()
1807
1808 # save the model as a pty file
1809 torch.save(model_cnn_4_relu.state_dict(), 'q4_models/model_cnn_4_relu.pt')
1810 print("model saved as 'q4_models/model_cnn_4_relu.pt'")
1811
1812 # train the model and save the training loss and validation loss for every 10 batches
1813   using model eval mode.s
1814 for epoch in tqdm(range(15)):
1815     for batch, (x, y) in enumerate(train_generator):
1816         x = x.to(device)
1817         y = y.to(device)
1818         # forward pass
1819         output = model_cnn_4_sigmoid(x)
1820         # compute loss
1821         loss_val = loss(output, y)
1822         # zero gradients
1823         optimizer_sigmoid.zero_grad()

```

```

1823     # backward pass
1824     loss_val.backward()
1825     # optimize
1826     optimizer_sigmoid.step()
1827     # print loss
1828     if batch % 10 == 0:
1829         model_cnn_4_sigmoid.eval()
1830         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
1831     )
1832     cnn_4_dict['sigmoid_loss_curve'].append(loss_val.item())
1833     cnn_4_dict['sigmoid_grad_curve'].append(model_cnn_4_sigmoid.Conv1.weight.grad
1834     .mean().item())
1835
1836     model_cnn_4_sigmoid.train()
1837
1838
1839 # save the model as a pty file
1840 torch.save(model_cnn_4_sigmoid.state_dict(), 'q4_models/model_cnn_4_sigmoid.pt')
1841 print("model saved as 'q4_models/model_cnn_4_sigmoid.pt'")
1842
1843
1844
1845 # save cnn_4_dict as a pickle file
1846 import pickle
1847 with open('q4_models/part4_cnn_4_dict.pkl', 'wb') as f:
1848     pickle.dump(cnn_4_dict, f)
1849
1850 # %% [markdown]
1851 # ### CNN5 Activation Function Experiment
1852
1853 # %%
1854 %reset -f
1855
1856 # Load fashion MNIST dataset
1857 import torchvision
1858
1859 from tqdm import tqdm
1860
1861
1862
1863 #training set
1864 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
1865         transform= torchvision.transforms.ToTensor())
1866
1867 #test set
1868 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
1869         transform= torchvision.transforms.ToTensor())
1870
1871
1872 batch_size = 50
1873 import torch
1874 # divide training data into training and validation sets of 0.8 and 0.2 respectively
1875
1876 train_data, val_data = torch.utils.data.random_split(train_data, [0.8, 0.2])
1877
1878 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
1879     True)

```

```

1877 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
      False)
1878 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
      False)
1879
1880 # relu cnn_5 classifier
1881 class cnn_5_relu(torch.nn.Module):
1882     def __init__(self, input_size, num_classes):
1883         super(cnn_5_relu, self).__init__()
1884         self.input_size = input_size
1885         self.Conv1 = torch.nn.Conv2d(1, 8, 3, stride=1, padding=1)
1886         self.Conv2 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
1887         self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1888         self.Conv4 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
1889         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
1890         self.Conv5 = torch.nn.Conv2d(16, 16, 3, stride=1, padding=1)
1891         self.Conv6 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1892         self.prediction_layer = torch.nn.Linear(392, num_classes)
1893         self.relu = torch.nn.ReLU()
1894     def forward(self, x):
1895         x = x.view(-1, 1, 28, 28)
1896         hidden1 = self.Conv1(x)
1897         relu1 = self.relu(hidden1)
1898         hidden2 = self.Conv2(relu1)
1899         relu2 = self.relu(hidden2)
1900         hidden3 = self.Conv3(relu2)
1901         relu3 = self.relu(hidden3)
1902         hidden4 = self.Conv4(relu3)
1903         relu4 = self.relu(hidden4)
1904         pool1 = self.MaxPool(relu4)
1905         hidden5 = self.Conv5(pool1)
1906         relu5 = self.relu(hidden5)
1907         hidden6 = self.Conv6(relu5)
1908         relu6 = self.relu(hidden6)
1909         pool2 = self.MaxPool(relu6)
1910         flattened = pool2.view(batch_size, -1)
1911         output = self.prediction_layer(flattened)
1912         return output
1913
1914 # sigmoid cnn_5 classifier
1915 class cnn_5_sigmoid(torch.nn.Module):
1916     def __init__(self, input_size, num_classes):
1917         super(cnn_5_sigmoid, self).__init__()
1918         self.input_size = input_size
1919         self.Conv1 = torch.nn.Conv2d(1, 8, 3, stride=1, padding=1)
1920         self.Conv2 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
1921         self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1922         self.Conv4 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
1923         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
1924         self.Conv5 = torch.nn.Conv2d(16, 16, 3, stride=1, padding=1)
1925         self.Conv6 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
1926         self.prediction_layer = torch.nn.Linear(392, num_classes)
1927         self.sigmoid = torch.nn.Sigmoid()
1928     def forward(self, x):
1929         x = x.view(-1, 1, 28, 28)
1930         hidden1 = self.Conv1(x)
1931         sigmoid1 = self.sigmoid(hidden1)
1932         hidden2 = self.Conv2(sigmoid1)
1933         sigmoid2 = self.sigmoid(hidden2)

```

```

1934         hidden3 = self.Conv3(sigmoid2)
1935         sigmoid3 = self.sigmoid(hidden3)
1936         hidden4 = self.Conv4(sigmoid3)
1937         sigmoid4 = self.sigmoid(hidden4)
1938         pool1 = self.MaxPool(sigmoid4)
1939         hidden5 = self.Conv5(pool1)
1940         sigmoid5 = self.sigmoid(hidden5)
1941         hidden6 = self.Conv6(sigmoid5)
1942         sigmoid6 = self.sigmoid(hidden6)
1943         pool2 = self.MaxPool(sigmoid6)
1944         flattened = pool2.view(batch_size, -1)
1945         output = self.prediction_layer(flattened)
1946         return output
1947
1948
1949
1950 # initialize cnn_5 relu model
1951 model_cnn_5_relu = cnn_5_relu(784, 10)
1952 # initialize cnn_5 sigmoid model
1953 model_cnn_5_sigmoid = cnn_5_sigmoid(784, 10)
1954
1955
1956 if torch.cuda.is_available():
1957     device = torch.device("cuda:0")
1958     print("CUDA to be used")
1959 else:
1960     device = "cpu"
1961     print("CPU to be used.")
1962
1963 model_cnn_5_relu.to(device)
1964 model_cnn_5_sigmoid.to(device)
1965
1966 # create loss: use cross entropy loss
1967 loss = torch.nn.CrossEntropyLoss()
1968
1969 # create optimizer
1970 optimizer_relu = torch.optim.SGD(model_cnn_5_relu.parameters(), lr=0.01)
1971 optimizer_sigmoid = torch.optim.SGD(model_cnn_5_sigmoid.parameters(), lr=0.01)
1972
1973
1974 # transfer your model to train mode
1975 model_cnn_5_relu.train()
1976 model_cnn_5_sigmoid.train()
1977
1978 cnn_5_dict = {"name": "cnn_5", "relu_loss_curve": [], "sigmoid_loss_curve": [], "
1979              relu_grad_curve": [], "sigmoid_grad_curve": []}
1980
1981
1982 # train the model and save the training loss and validation loss for every 10 batches
1983 # using model eval mode.s
1984 for epoch in tqdm(range(15)):
1985     for batch, (x, y) in enumerate(train_generator):
1986         x = x.to(device)
1987         y = y.to(device)
1988         # forward pass
1989         output = model_cnn_5_relu(x)
1990         # compute loss
1991         loss_val = loss(output, y)

```



```

1991     # zero gradients
1992     optimizer_relu.zero_grad()
1993     # backward pass
1994     loss_val.backward()
1995     # optimize
1996     optimizer_relu.step()
1997     # print loss
1998     if batch % 10 == 0:
1999         model_cnn_5_relu.eval()
2000         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
2001     )
2002     cnn_5_dict['relu_loss_curve'].append(loss_val.item())
2003     cnn_5_dict['relu_grad_curve'].append(model_cnn_5_relu.Conv1.weight.grad.mean
2004     ().item())
2005
2006     model_cnn_5_relu.train()
2007
2008
2009 # save the model as a pty file
2010 torch.save(model_cnn_5_relu.state_dict(), 'q4_models/model_cnn_5_relu.pt')
2011 print("model saved as 'q4_models/model_cnn_5_relu.pt'")
2012
2013 # train the model and save the training loss and validation loss for every 10 batches
2014     using model eval mode.s
2015 for epoch in tqdm(range(15)):
2016     for batch, (x, y) in enumerate(train_generator):
2017         x = x.to(device)
2018         y = y.to(device)
2019         # forward pass
2020         output = model_cnn_5_sigmoid(x)
2021         # compute loss
2022         loss_val = loss(output, y)
2023         # zero gradients
2024         optimizer_sigmoid.zero_grad()
2025         # backward pass
2026         loss_val.backward()
2027         # optimize
2028         optimizer_sigmoid.step()
2029         # print loss
2030         if batch % 10 == 0:
2031             model_cnn_5_sigmoid.eval()
2032             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
2033         )
2034             cnn_5_dict['sigmoid_loss_curve'].append(loss_val.item())
2035             cnn_5_dict['sigmoid_grad_curve'].append(model_cnn_5_sigmoid.Conv1.weight.grad
2036             .mean().item())
2037
2038             model_cnn_5_sigmoid.train()
2039
2040
2041 # save the model as a pty file
2042 torch.save(model_cnn_5_sigmoid.state_dict(), 'q4_models/model_cnn_5_sigmoid.pt')
2043 print("model saved as 'q4_models/model_cnn_5_sigmoid.pt'")
2044

```

```

2045
2046 # save cnn_5_dict as a pickle file
2047 import pickle
2048 with open('q4_models/part4_cnn_5_dict.pkl', 'wb') as f:
2049     pickle.dump(cnn_5_dict, f)
2050
2051 # %% [markdown]
2052 # ## Part 4 plotting part.
2053
2054 # %%
2055 %reset -f
2056 from utils import part4Plots
2057 import pickle
2058
2059 # load the pickle files
2060 with open('q4_models/part4_mlp_1_dict.pkl', 'rb') as f:
2061     mlp_1_dict = pickle.load(f)
2062
2063 with open('q4_models/part4_mlp_2_dict.pkl', 'rb') as f:
2064     mlp_2_dict = pickle.load(f)
2065
2066 with open('q4_models/part4_cnn_3_dict.pkl', 'rb') as f:
2067     cnn_3_dict = pickle.load(f)
2068
2069 with open('q4_models/part4_cnn_4_dict.pkl', 'rb') as f:
2070     cnn_4_dict = pickle.load(f)
2071
2072 with open('q4_models/part4_cnn_5_dict.pkl', 'rb') as f:
2073     cnn_5_dict = pickle.load(f)
2074 results = [mlp_1_dict]
2075
2076 part4Plots(results, "q4_models", "part4_alpha_mlp_1")
2077
2078 results = [mlp_2_dict]
2079
2080 part4Plots(results, "q4_models", "part4_alpha_mlp_2")
2081
2082 results = [cnn_3_dict]
2083
2084 part4Plots(results, "q4_models", "part4_alpha_cnn_3")
2085
2086 results = [cnn_4_dict]
2087
2088 part4Plots(results, "q4_models", "part4_alpha_cnn_4")
2089
2090 results = [cnn_5_dict]
2091
2092 part4Plots(results, "q4_models", "part4_alpha_cnn_5")
2093
2094
2095 # %% [markdown]
2096 # ## Experimenting Learning Rate
2097 #
2098 # My favorite architecture is CNN3.
2099 #
2100 #
2101
2102 # %%
2103 %reset -f

```

```

2104
2105 # Load fashion MNIST dataset
2106 import torchvision
2107
2108 from tqdm import tqdm
2109
2110 #training set
2111 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
        transform= torchvision.transforms.ToTensor())
2112
2113 #test set
2114 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
        transform= torchvision.transforms.ToTensor())
2115
2116 batch_size = 50
2117 import torch
2118 # divide training data into training and validation sets of 0.9 and 0.1 respectively
2119
2120 train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])
2121
2122 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
        True)
2123 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=
        False)
2124 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
        False)
2125
2126
2127
2128 # cnn_3 classifier
2129 class cnn_3(torch.nn.Module):
2130     def __init__(self, input_size, num_classes):
2131         super(cnn_3, self).__init__()
2132         self.input_size = input_size
2133         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
2134         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
2135         self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
2136         self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
2137         self.prediction_layer = torch.nn.Linear(1296, num_classes)
2138         self.relu = torch.nn.ReLU()
2139     def forward(self, x):
2140         x = x.view(-1, 1, 28, 28)
2141         hidden1 = self.Conv1(x)
2142         relu1 = self.relu(hidden1)
2143         hidden2 = self.Conv2(relu1)
2144         relu2 = self.relu(hidden2)
2145         pool = self.MaxPool(relu2)
2146         hidden3 = self.Conv3(pool)
2147         flattened = hidden3.view(batch_size, -1)
2148         output = self.prediction_layer(flattened)
2149         return output
2150
2151
2152 # initialize cnn_5 relu model
2153 model_cnn_3 = cnn_3(784, 10)
2154
2155
2156 if torch.cuda.is_available():
2157     device = torch.device("cuda:0")

```

```

2158     print("CUDA to be used")
2159 else:
2160     device = "cpu"
2161     print("CPU to be used.")
2162
2163 model_cnn_3.to(device)
2164
2165
2166 # create loss: use cross entropy loss
2167 loss = torch.nn.CrossEntropyLoss()
2168
2169 # create optimizer
2170 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)
2171
2172
2173 # transfer your model to train mode
2174 model_cnn_3.train()
2175
2176
2177 cnn_3_dict = {"name": "cnn_3", "loss_curve_1": [], "loss_curve_01": [], "loss_curve_001":
2178               [], "val_acc_curve_1": [], "val_acc_curve_01": [], "val_acc_curve_001": []}
2179
2180
2181 # train the model and save the training loss and validation loss for every 10 batches
2182 # using model eval mode.s
2183 for epoch in tqdm(range(20)):
2184     for batch, (x, y) in enumerate(train_generator):
2185
2186         x = x.to(device)
2187         y = y.to(device)
2188
2189         # forward pass
2190         output = model_cnn_3(x)
2191         # compute loss
2192         loss_val = loss(output, y)
2193         # zero gradients
2194         optimizer.zero_grad()
2195         # backward pass
2196         loss_val.backward()
2197         # optimize
2198         optimizer.step()
2199         # print loss
2200         if batch % 10 == 0:
2201             model_cnn_3.eval()
2202             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
2203
2204             cnn_3_dict['loss_curve_1'].append(loss_val.item())
2205
2206             # validation loss
2207             validation_accuracy_per_batch = torch.tensor([])
2208             with torch.no_grad():
2209                 for val_x, val_y in val_generator:
2210
2211                     val_x = val_x.to(device)
2212                     val_y = val_y.to(device)
2213
2214                     val_output = model_cnn_3(val_x)
2215                     #val_loss = loss(val_output, val_y)

```

```

2214         val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
2215         try:
2216             validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch ,val_accuracy))
2217         except:
2218             validation_accuracy_per_batch = val_accuracy
2219             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean())
)
2220             cnn_3_dict['val_acc_curve_1'].append(validation_accuracy_per_batch.mean().
item())
2221             model_cnn_3.train()
2222
2223
2224
2225
2226 # initialize cnn_5 relu model
2227 model_cnn_3 = cnn_3(784 ,10)
2228
2229 model_cnn_3.to(device)
2230
2231
2232 # create loss: use cross entropy loss
2233 loss = torch.nn.CrossEntropyLoss()
2234
2235 # create optimizer
2236 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
2237
2238
2239 # transfer your model to train mode
2240 model_cnn_3.train()
2241
2242 # train the model and save the training loss and validation loss for every 10 batches
using model eval mode.s
2243 for epoch in tqdm(range(20)):
2244     for batch, (x, y) in enumerate(train_generator):
2245
2246         x = x.to(device)
2247         y = y.to(device)
2248
2249         # forward pass
2250         output = model_cnn_3(x)
2251         # compute loss
2252         loss_val = loss(output, y)
2253         # zero gradients
2254         optimizer.zero_grad()
2255         # backward pass
2256         loss_val.backward()
2257         # optimize
2258         optimizer.step()
2259         # print loss
2260         if batch % 10 == 0:
2261             model_cnn_3.eval()
2262             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
)
2263             cnn_3_dict['loss_curve_01'].append(loss_val.item())
2264
2265         # validation loss
2266         validation_accuracy_per_batch = torch.tensor([])

```

```

2267         with torch.no_grad():
2268             for val_x, val_y in val_generator:
2269
2270                 val_x = val_x.to(device)
2271                 val_y = val_y.to(device)
2272
2273                 val_output = model_cnn_3(val_x)
2274                 #val_loss = loss(val_output, val_y)
2275                 val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
2276
2277                 try:
2278                     validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch ,val_accuracy))
2279                 except:
2280                     validation_accuracy_per_batch = val_accuracy
2281                     #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()
))
2282                     cnn_3_dict['val_acc_curve_01'].append(validation_accuracy_per_batch.mean()
item())
2283                     model_cnn_3.train()
2284
2285
2286
2287 # initialize cnn_5 relu model
2288 model_cnn_3 = cnn_3(784 ,10)
2289
2290 model_cnn_3.to(device)
2291
2292
2293 # create loss: use cross entropy loss
2294 loss = torch.nn.CrossEntropyLoss()
2295
2296 # create optimizer
2297 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)
2298
2299
2300 # transfer your model to train mode
2301 model_cnn_3.train()
2302
2303 # train the model and save the training loss and validation loss for every 10 batches
using model eval mode.s
2304 for epoch in tqdm(range(20)):
2305     for batch, (x, y) in enumerate(train_generator):
2306
2307         x = x.to(device)
2308         y = y.to(device)
2309
2310         # forward pass
2311         output = model_cnn_3(x)
2312         # compute loss
2313         loss_val = loss(output, y)
2314         # zero gradients
2315         optimizer.zero_grad()
2316         # backward pass
2317         loss_val.backward()
2318         # optimize
2319         optimizer.step()
2320         # print loss

```

```

2321         if batch % 10 == 0:
2322             model_cnn_3.eval()
2323             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
2324         )
2325
2326         cnn_3_dict['loss_curve_001'].append(loss_val.item())
2327
2328         # validation loss
2329         validation_accuracy_per_batch = torch.tensor([])
2330         with torch.no_grad():
2331             for val_x, val_y in val_generator:
2332
2333                 val_x = val_x.to(device)
2334                 val_y = val_y.to(device)
2335
2336                 val_output = model_cnn_3(val_x)
2337                 #val_loss = loss(val_output, val_y)
2338                 val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
2339 float())
2340
2341             try:
2342                 validation_accuracy_per_batch = torch.cat((
2343 validation_accuracy_per_batch ,val_accuracy))
2344             except:
2345                 validation_accuracy_per_batch = val_accuracy
2346             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
2347         )
2348         cnn_3_dict['val_acc_curve_001'].append(validation_accuracy_per_batch.mean().
2349 item())
2350         model_cnn_3.train()
2351
2352
2353
2354
2355
2356
2357 # save cnn_3_dict as a pickle file
2358 import pickle
2359 with open('q5_models/part5_cnn_3_dict.pkl', 'wb') as f:
2360     pickle.dump(cnn_3_dict, f)
2361
2362
2363 # %%
2364 from utils import part5Plots
2365 import pickle
2366
2367 # load the pickle files
2368 with open('q5_models/part5_cnn_3_dict.pkl', 'rb') as f:
2369     cnn_3_dict = pickle.load(f)
2370
2371 part5Plots(cnn_3_dict, "q5_models")
2372 print(len(cnn_3_dict['val_acc_curve_1']))
2373
2374 # %%
2375 %reset -f
2376
2377 # Load fashion MNIST dataset
2378 import torchvision
2379
2380 from tqdm import tqdm
2381
2382 #training set
2383 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
2384 transform= torchvision.transforms.ToTensor())

```

```

2374 #test set
2375 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
        transform= torchvision.transforms.ToTensor())
2376
2377 batch_size = 50
2378 import torch
2379 # divide training data into training and validation sets of 0.9 and 0.1 respectively
2380
2381 train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])
2382
2383 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
        True)
2384 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True
        )
2385 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
        False)
2386
2387
2388
2389 # cnn_3 classifier
2390 class cnn_3(torch.nn.Module):
2391     def __init__(self, input_size, num_classes):
2392         super(cnn_3, self).__init__()
2393         self.input_size = input_size
2394         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
2395         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
2396         self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
2397         self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
2398         self.prediction_layer = torch.nn.Linear(1296, num_classes)
2399         self.relu = torch.nn.ReLU()
2400     def forward(self, x):
2401         x = x.view(-1, 1, 28, 28)
2402         hidden1 = self.Conv1(x)
2403         relu1 = self.relu(hidden1)
2404         hidden2 = self.Conv2(relu1)
2405         relu2 = self.relu(hidden2)
2406         pool = self.MaxPool(relu2)
2407         hidden3 = self.Conv3(pool)
2408         flattened = hidden3.view(batch_size, -1)
2409         output = self.prediction_layer(flattened)
2410         return output
2411
2412
2413 # initialize cnn_5 relu model
2414 model_cnn_3 = cnn_3(784, 10)
2415
2416
2417 if torch.cuda.is_available():
2418     device = torch.device("cuda:0")
2419     print("CUDA to be used")
2420 else:
2421     device = "cpu"
2422     print("CPU to be used.")
2423
2424 model_cnn_3.to(device)
2425
2426
2427 # create loss: use cross entropy loss
2428 loss = torch.nn.CrossEntropyLoss()

```



```

2429
2430 # create optimizer
2431 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)
2432
2433
2434 # transfer your model to train mode
2435 model_cnn_3.train()
2436
2437
2438 cnn_3_dict = {"name": "cnn_3_scheduled", "loss_curve": [], "val_acc_curve": []}
2439
2440
2441
2442 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
2443 for epoch in tqdm(range(30)):
2444     #decrease learning rate to 0.01 after 7 epochs
2445     if epoch == 8:
2446         optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
2447     for batch, (x, y) in enumerate(train_generator):
2448
2449         x = x.to(device)
2450         y = y.to(device)
2451
2452         # forward pass
2453         output = model_cnn_3(x)
2454         # compute loss
2455         loss_val = loss(output, y)
2456         # zero gradients
2457         optimizer.zero_grad()
2458         # backward pass
2459         loss_val.backward()
2460         # optimize
2461         optimizer.step()
2462         # print loss
2463         if batch % 10 == 0:
2464             model_cnn_3.eval()
2465             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
        )
2466             cnn_3_dict['loss_curve'].append(loss_val.item())
2467
2468         # validation loss
2469         validation_accuracy_per_batch = torch.tensor([])
2470         with torch.no_grad():
2471             for val_x, val_y in val_generator:
2472
2473                 val_x = val_x.to(device)
2474                 val_y = val_y.to(device)
2475
2476                 val_output = model_cnn_3(val_x)
2477                 #val_loss = loss(val_output, val_y)
2478                 val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
2479                 try:
2480                     validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch ,val_accuracy))
2481                 except:
2482                     validation_accuracy_per_batch = val_accuracy

```

```

2483         #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
2484         )
2485         cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
2486     )
2487     model_cnn_3.train()
2488
2489     # #decrease learning rate to 0.001 after 15 epochs
2490     if epoch == 18:
2491         optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)
2492
2493     # plot validation accuracy curve where the x axis is epoch
2494     import matplotlib.pyplot as plt
2495     import numpy as np
2496     x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
2497     plt.plot(x_axis, cnn_3_dict['val_acc_curve'])
2498     plt.title("Validation Accuracy Curve learning rate scheduling (rate drops at 7th and 14th epoch)")
2499     plt.xlabel("Epoch")
2500     plt.ylabel("Validation Accuracy")
2501     plt.show()
2502     plt.savefig("q5_models/part5_cnn_3_scheduled_3.png")
2503
2504     # %%
2505
2506     %reset -f
2507
2508     # Load fashion MNIST dataset
2509     import torchvision
2510
2511     from tqdm import tqdm
2512
2513     #training set
2514     train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
2515                                                    transform= torchvision.transforms.ToTensor())
2516
2517     #test set
2518     test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
2519                                                    transform= torchvision.transforms.ToTensor())
2520
2521     batch_size = 50
2522     import torch
2523     # divide training data into training and validation sets of 0.9 and 0.1 respectively
2524
2525     train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])
2526
2527     train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
2528                                                    True)
2529     val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True
2530                                                  )
2531     test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
2532                                                    False)
2533
2534     # cnn_3 classifier
2535     class cnn_3(torch.nn.Module):
2536         def __init__(self, input_size, num_classes):
2537             super(cnn_3, self).__init__()

```

```

2534     self.input_size = input_size
2535     self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
2536     self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
2537     self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
2538     self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
2539     self.prediction_layer = torch.nn.Linear(1296, num_classes)
2540     self.relu = torch.nn.ReLU()
2541     def forward(self, x):
2542         x = x.view(-1, 1, 28, 28)
2543         hidden1 = self.Conv1(x)
2544         relu1 = self.relu(hidden1)
2545         hidden2 = self.Conv2(relu1)
2546         relu2 = self.relu(hidden2)
2547         pool = self.MaxPool(relu2)
2548         hidden3 = self.Conv3(pool)
2549         flattened = hidden3.view(batch_size, -1)
2550         output = self.prediction_layer(flattened)
2551         return output
2552
2553
2554 # initialize cnn_5 relu model
2555 model_cnn_3 = cnn_3(784, 10)
2556
2557
2558 if torch.cuda.is_available():
2559     device = torch.device("cuda:0")
2560     print("CUDA to be used")
2561 else:
2562     device = "cpu"
2563     print("CPU to be used.")
2564
2565 model_cnn_3.to(device)
2566
2567
2568 # create loss: use cross entropy loss
2569 loss = torch.nn.CrossEntropyLoss()
2570
2571 # create optimizer
2572 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)
2573
2574
2575 # transfer your model to train mode
2576 model_cnn_3.train()
2577
2578
2579 cnn_3_dict = {"name": "cnn_3_scheduled", "loss_curve": [], "val_acc_curve": []}
2580
2581
2582
2583 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
2584 for epoch in tqdm(range(7)):
2585     # decrease learning rate to 0.01 after 7 epochs
2586     # if epoch == 8:
2587     #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
2588     for batch, (x, y) in enumerate(train_generator):
2589
2590         x = x.to(device)
2591         y = y.to(device)

```

```

2592
2593     # forward pass
2594     output = model_cnn_3(x)
2595     # compute loss
2596     loss_val = loss(output, y)
2597     # zero gradients
2598     optimizer.zero_grad()
2599     # backward pass
2600     loss_val.backward()
2601     # optimize
2602     optimizer.step()
2603     # print loss
2604     if batch % 10 == 0:
2605         model_cnn_3.eval()
2606         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
2607     )
2608
2609     cnn_3_dict['loss_curve'].append(loss_val.item())
2610
2611     # validation loss
2612     validation_accuracy_per_batch = torch.tensor([])
2613     with torch.no_grad():
2614         for val_x, val_y in val_generator:
2615
2616             val_x = val_x.to(device)
2617             val_y = val_y.to(device)
2618
2619             val_output = model_cnn_3(val_x)
2620             #val_loss = loss(val_output, val_y)
2621             val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
2622 float())
2623
2624             try:
2625                 validation_accuracy_per_batch = torch.cat((
2626 validation_accuracy_per_batch ,val_accuracy))
2627             except:
2628                 validation_accuracy_per_batch = val_accuracy
2629             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
2630         )
2631
2632         cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
2633 ())
2634
2635         model_cnn_3.train()
2636
2637 for g in optimizer.param_groups:
2638     g['lr'] = 0.001
2639
2640 # transfer your model to train mode
2641 model_cnn_3.train()
2642
2643 # train the model and save the training loss and validation loss for every 10 batches
2644 using model eval mode.s
2645 for epoch in tqdm(range(23)):
2646     for batch, (x, y) in enumerate(train_generator):
2647
2648         x = x.to(device)
2649         y = y.to(device)
2650
2651         # forward pass
2652         output = model_cnn_3(x)
2653         # compute loss
2654         loss_val = loss(output, y)

```

```

2645     # zero gradients
2646     optimizer.zero_grad()
2647     # backward pass
2648     loss_val.backward()
2649     # optimize
2650     optimizer.step()
2651     # print loss
2652     if batch % 10 == 0:
2653         model_cnn_3.eval()
2654         #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item()))
2655     )
2656     cnn_3_dict['loss_curve'].append(loss_val.item())
2657
2658     # validation loss
2659     validation_accuracy_per_batch = torch.tensor([])
2660     with torch.no_grad():
2661         for val_x, val_y in val_generator:
2662
2663             val_x = val_x.to(device)
2664             val_y = val_y.to(device)
2665
2666             val_output = model_cnn_3(val_x)
2667             #val_loss = loss(val_output, val_y)
2668             val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
2669 float())
2670
2671             try:
2672                 validation_accuracy_per_batch = torch.cat((
2673 validation_accuracy_per_batch , val_accuracy))
2674             except:
2675                 validation_accuracy_per_batch = val_accuracy
2676             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
2677         )
2678         cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item
2679 ())
2680         model_cnn_3.train()
2681
2682
2683
2684
2685
2686
2687
2688
2689 # plot validation accuracy curve where the x axis is epoch
2690 import matplotlib.pyplot as plt
2691 import numpy as np
2692 x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
2693 plt.plot(x_axis , cnn_3_dict['val_acc_curve'])
2694 plt.title("Validation Accuracy Curve learning rate scheduling (rate drops at 7th epoch)")
2695 plt.xlabel("Epoch")
2696 plt.ylabel("Validation Accuracy")
2697 plt.show()
2698 plt.savefig("q5_models/part5_cnn_3_scheduled_2.png")
2699
2700
2701 # %%
2702 %reset -f
2703
2704
2705 # Load fashion MNIST dataset
2706 import torchvision
2707
2708
2709 from tqdm import tqdm
2710
2711 #training set

```

```

2698 train_data = torchvision.datasets.FashionMNIST(root='./data', train=True, download=False,
2699         transform= torchvision.transforms.ToTensor())
2700 #test set
2701 test_data = torchvision.datasets.FashionMNIST(root='./data', train=False, download=False,
2702         transform= torchvision.transforms.ToTensor())
2703
2704 batch_size = 50
2705 import torch
2706 # divide training data into training and validation sets of 0.9 and 0.1 respectively
2707 train_data, val_data = torch.utils.data.random_split(train_data, [0.9, 0.1])
2708
2709 train_generator = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=
2710     True)
2711 val_generator = torch.utils.data.DataLoader(val_data, batch_size=batch_size, shuffle=True
2712     )
2713 test_generator = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=
2714     False)
2715
2716 # cnn_3 classifier
2717 class cnn_3(torch.nn.Module):
2718     def __init__(self, input_size, num_classes):
2719         super(cnn_3, self).__init__()
2720         self.input_size = input_size
2721         self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
2722         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
2723         self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)
2724         self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
2725         self.prediction_layer = torch.nn.Linear(1296, num_classes)
2726         self.relu = torch.nn.ReLU()
2727     def forward(self, x):
2728         x = x.view(-1, 1, 28, 28)
2729         hidden1 = self.Conv1(x)
2730         relu1 = self.relu(hidden1)
2731         hidden2 = self.Conv2(relu1)
2732         relu2 = self.relu(hidden2)
2733         pool = self.MaxPool(relu2)
2734         hidden3 = self.Conv3(pool)
2735         flattened = hidden3.view(batch_size, -1)
2736         output = self.prediction_layer(flattened)
2737         return output
2738
2739 # initialize cnn_5 relu model
2740 model_cnn_3 = cnn_3(784, 10)
2741
2742
2743 if torch.cuda.is_available():
2744     device = torch.device("cuda:0")
2745     print("CUDA to be used")
2746 else:
2747     device = "cpu"
2748     print("CPU to be used.")
2749
2750 model_cnn_3.to(device)
2751

```

```

2752
2753 # create loss: use cross entropy loss
2754 loss = torch.nn.CrossEntropyLoss()
2755
2756 # create optimizer
2757 optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.1)
2758
2759
2760 # transfer your model to train mode
2761 model_cnn_3.train()
2762
2763
2764 cnn_3_dict = {"name": "cnn_3_scheduled", "loss_curve": [], "val_acc_curve": []}
2765
2766
2767
2768 # train the model and save the training loss and validation loss for every 10 batches
    using model eval mode.s
2769 for epoch in tqdm(range(30)):
2770     #decrease learning rate to 0.01 after 7 epochs
2771     # if epoch == 8:
2772     #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.01)
2773     for batch, (x, y) in enumerate(train_generator):
2774
2775         x = x.to(device)
2776         y = y.to(device)
2777
2778         # forward pass
2779         output = model_cnn_3(x)
2780         # compute loss
2781         loss_val = loss(output, y)
2782         # zero gradients
2783         optimizer.zero_grad()
2784         # backward pass
2785         loss_val.backward()
2786         # optimize
2787         optimizer.step()
2788         # print loss
2789         if batch % 10 == 0:
2790             model_cnn_3.eval()
2791             #print('Epoch: {}, Batch: {}, Loss: {}'.format(epoch, batch, loss_val.item())
    )
2792
2793             cnn_3_dict['loss_curve'].append(loss_val.item())
2794
2795             # validation loss
2796             validation_accuracy_per_batch = torch.tensor([])
2797             with torch.no_grad():
2798                 for val_x, val_y in val_generator:
2799
2800                     val_x = val_x.to(device)
2801                     val_y = val_y.to(device)
2802
2803                     val_output = model_cnn_3(val_x)
2804                     #val_loss = loss(val_output, val_y)
2805                     val_accuracy = torch.mean((torch.argmax(val_output, dim=1) == val_y).
float())
2806
2807                     try:
2808                         validation_accuracy_per_batch = torch.cat((
validation_accuracy_per_batch ,val_accuracy))

```

```

2807         except:
2808             validation_accuracy_per_batch = val_accuracy
2809             #print('Validation Accuracy: {}'.format(validation_accuracy_per_batch.mean()))
2810         )
2811         cnn_3_dict['val_acc_curve'].append(validation_accuracy_per_batch.mean().item())
2812     )
2813     model_cnn_3.train()
2814
2815     # # #decrease learning rate to 0.001 after 15 epochs
2816     # if epoch == 18:
2817     #     optimizer = torch.optim.SGD(model_cnn_3.parameters(), lr=0.001)
2818
2819 # plot validation accuracy curve where the x axis is epoch
2820 import matplotlib.pyplot as plt
2821 import numpy as np
2822 x_axis = np.linspace(0, 30, len(cnn_3_dict['val_acc_curve']))
2823 plt.plot(x_axis, cnn_3_dict['val_acc_curve'])
2824 plt.title("Validation Accuracy Curve learning rate scheduling (no learning rate drop)")
2825 plt.xlabel("Epoch")
2826 plt.ylabel("Validation Accuracy")
2827 plt.show()
2828 plt.savefig("q5_models/part5_cnn_3_scheduled_1.png")

```

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 # load the kernel from
6 kernel=np.load('hw1/data/kernel.npy')
7
8 input=np.load('hw1/data/samples_0.npy')
9
10 # plot kernels on the same figure there are eight kernels in total
11 plt.figure()
12 for i in range(8):
13     plt.subplot(2,4,i+1)
14     plt.imshow(kernel[i,0,:,:], cmap='gray')
15     plt.title('kernel'+str(i+1))
16
17 plt.show()
18
19 # plot input
20
21 plt.figure()
22 for i in range(5):
23     plt.subplot(1,5,i+1)
24     plt.imshow(input[i,0,:,:], cmap='gray')
25     plt.title('input'+str(i+1))
26
27 plt.show()

```

```

1 import torchinfo
2
3 import torch
4
5 # example mlp classifier
6 class mlp_1(torch.nn.Module):
7     def __init__(self, input_size, hidden_size, num_classes):
8         super(mlp_1, self).__init__()

```



```

9         self.input_size = input_size
10        self.FC = torch.nn.Linear(input_size, hidden_size)
11        self.prediction_layer = torch.nn.Linear(hidden_size, num_classes)
12        self.relu = torch.nn.ReLU()
13        def forward(self, x):
14            x = x.view(-1, self.input_size)
15            hidden = self.FC(x)
16            relu = self.relu(hidden)
17            output = self.prediction_layer(relu)
18            return output
19
20    # initialize your model
21    model_mlp_1 = mlp_1(784,32,10)
22
23
24    torchinfo.summary(model_mlp_1, input_size=(96, 784))
25
26    # Total params: 25,450
27    # Trainable params: 25,450
28
29
30
31
32
33    # example mlp classifier
34    class mlp_2(torch.nn.Module):
35        def __init__(self, input_size, hidden_size_1, hidden_size_2, num_classes):
36            super(mlp_2, self).__init__()
37            self.input_size = input_size
38            self.FC1 = torch.nn.Linear(input_size, hidden_size_1)
39            self.FC2 = torch.nn.Linear(hidden_size_1, hidden_size_2, bias=False)
40            self.prediction_layer = torch.nn.Linear(hidden_size_2, num_classes)
41            self.relu = torch.nn.ReLU()
42        def forward(self, x):
43            x = x.view(-1, self.input_size)
44            hidden1 = self.FC1(x)
45            relu = self.relu(hidden1)
46            hidden2 = self.FC2(relu)
47            output = self.prediction_layer(hidden2)
48            return output
49
50    # initialize your model
51    model_mlp_2 = mlp_2(784, 32, 64, 10)
52
53    torchinfo.summary(model_mlp_2, input_size=(96, 784))
54
55
56    # Total params: 27,818
57    # Trainable params: 27,818
58
59
60    # example cnn_3 classifier
61    class cnn_3(torch.nn.Module):
62        def __init__(self, input_size, num_classes):
63            super(cnn_3, self).__init__()
64            self.input_size = input_size
65            self.Conv1 = torch.nn.Conv2d(1, 16, 3, stride=1, padding=1)
66            self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
67            self.Conv2 = torch.nn.Conv2d(16, 8, 5, stride=1, padding=1)

```

```

68     self.Conv3 = torch.nn.Conv2d(8, 16, 7, stride=1, padding=1)
69     self.prediction_layer = torch.nn.Linear(1296, num_classes)
70     self.relu = torch.nn.ReLU()
71     def forward(self, x):
72         x = x.view(-1, 1, 28, 28)
73         hidden1 = self.Conv1(x)
74         relu1 = self.relu(hidden1)
75         hidden2 = self.Conv2(relu1)
76         relu2 = self.relu(hidden2)
77         pool = self.MaxPool(relu2)
78         hidden3 = self.Conv3(pool)
79         flattened = hidden3.view(96, -1)
80         output = self.prediction_layer(flattened)
81         return output
82
83 # initialize your model
84 model_cnn_3 = cnn_3(784 ,10)
85
86 torchinfo.summary(model_cnn_3, input_size=(96, 784))
87
88 # Total params: 22,626
89 # Trainable params: 22,626
90
91
92 # example cnn_4 classifier
93 class cnn_4(torch.nn.Module):
94     def __init__(self, input_size, num_classes):
95         super(cnn_4, self).__init__()
96         self.input_size = input_size
97         self.Conv1 = torch.nn.Conv2d(1 ,16 ,3, stride=1, padding=1)
98         self.Conv2 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
99         self.Conv3 = torch.nn.Conv2d(8, 16, 5, stride=1, padding=1)
100        self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
101        self.Conv4 = torch.nn.Conv2d(16, 16, 5, stride=1, padding=1)
102        self.prediction_layer = torch.nn.Linear(400, num_classes)
103        self.relu = torch.nn.ReLU()
104
105        def forward(self, x):
106            x = x.view(-1, 1, 28, 28)
107            hidden1 = self.Conv1(x)
108            relu1 = self.relu(hidden1)
109            hidden2 = self.Conv2(relu1)
110            relu2 = self.relu(hidden2)
111            hidden3 = self.Conv3(relu2)
112            relu3 = self.relu(hidden3)
113            pool1 = self.MaxPool(relu3)
114            hidden4 = self.Conv4(pool1)
115            pool2 = self.MaxPool(hidden4)
116            flattened = pool2.view(96, -1)
117            output = self.prediction_layer(flattened)
118            return output
119
120 # initialize your model
121 model_cnn_4 = cnn_4(784 ,10)
122
123 torchinfo.summary(model_cnn_4, input_size=(96, 784))
124
125 # Total params: 14,962
126 # Trainable params: 14,962

```

```

127 # example cnn_5 classifier
128 class cnn_5(torch.nn.Module):
129     def __init__(self, input_size, num_classes):
130         super(cnn_5, self).__init__()
131         self.input_size = input_size
132         self.Conv1 = torch.nn.Conv2d(1, 8, 3, stride=1, padding=1)
133         self.Conv2 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
134         self.Conv3 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
135         self.Conv4 = torch.nn.Conv2d(8, 16, 3, stride=1, padding=1)
136         self.MaxPool = torch.nn.MaxPool2d(2, stride=2)
137         self.Conv5 = torch.nn.Conv2d(16, 16, 3, stride=1, padding=1)
138         self.Conv6 = torch.nn.Conv2d(16, 8, 3, stride=1, padding=1)
139         self.prediction_layer = torch.nn.Linear(392, num_classes)
140         self.relu = torch.nn.ReLU()
141     def forward(self, x):
142         x = x.view(-1, 1, 28, 28)
143         hidden1 = self.Conv1(x)
144         relu1 = self.relu(hidden1)
145         hidden2 = self.Conv2(relu1)
146         relu2 = self.relu(hidden2)
147         hidden3 = self.Conv3(relu2)
148         relu3 = self.relu(hidden3)
149         hidden4 = self.Conv4(relu3)
150         relu4 = self.relu(hidden4)
151         pool1 = self.MaxPool(relu4)
152         hidden5 = self.Conv5(pool1)
153         relu5 = self.relu(hidden5)
154         hidden6 = self.Conv6(relu5)
155         relu6 = self.relu(hidden6)
156         pool2 = self.MaxPool(relu6)
157         flattened = pool2.view(96, -1)
158         output = self.prediction_layer(flattened)
159         return output
160
161 # initialize your model
162 model_cnn_5 = cnn_5(784, 10)
163
164
165 torchinfo.summary(model_cnn_5, input_size=(96, 784))
166 # Total params: 10,986
167 # Trainable params: 10,986

```

Submitted by Ahmet Akman 2442366 on April 7, 2024.