Esmat Sahak
ECE324

ASSIGNMENT 2

Solve by Inspection

What are values of w_i and b that will make the classifier 100% correct for all possible inputs using the 0.5 decision point classification rule?

w_0, w_2, w_4, w_6, w_8 = 1 and w_1, w_3, w_5, w_7 = -1 and b = -4.5.

Is the answer to (i) unique? If your answer is yes, say why. If your answer is no, give a second answer that uses different weights and bias.

The answer is not unique.
w_0, w_2, w_4, w_6, w_8 = 0.5 and w_1, w_3, w_5, w_7 = -0.5 and b = -2.

How many unique inputs (that is, different instances of I = {I0, I1, ..., I8}) are possible for the 3x3 grid?

2^9 = 512 unique inputs.

Does your solution easily scale to solve finding a single X-type pattern in a 4x4 grid? Explain why or why not.

Yes, the same general convention can be used: input elements along the X receive weight 1 and the other elements receive weight -1. The only factor that needs adjusting is the bias which depends on the number of input elements that form the X pattern and the decision function limit. Note that this only works because there is only ONE configuration for the valid X-pattern. In the example below, if the X-pattern is the input with ones at array indices corresponding to the diagonals, then the necessary bias value is 0.5 - 8 = -7.5.

1	0	0	1
0	1	1	0
0	1	1	0
1	0	0	1

Suppose that, on a 5x5 grid, you had to match the 'X' as above, but, in addition, an 'X' shifted left by one, and shifted right by 1 position also had to be matched (in the shifted cases, part of the X would be missing). Could you as easily create the single-neuron parameters to solve that problem? Why or why not?

The approach in (iv) cannot be implemented if there is more than one inputs that satisfy the criteria of the X pattern. This is because some inputs can satisfy the decision function if there are sufficient elements of weight 1 have value 1 but do not form an X-pattern. An example of an input that would fall under this category is shown below. Hence, the analysis would be more complex.

1	0	0	1	1
1	1	1	1	0
0	1	1	0	0
1	1	1	1	0
1	0	0	1	1

Esmat Sahak
ECE324

Part 1

Learning rate = 0.1, activation function = rectified linear unit, seed = 10

Epochs	T_{acc}	V_{acc}	T_{loss}	V_{loss}
0	0.335	0.35	5.962786776737121	5.913201429964414
25	0.945	1.0	0.05973656774706246	0.05318154297107647
50	0.97	0.95	0.02600641478530398	0.027860492669099828
75	0.985	0.95	0.01725734722788328	0.022611332053890907
100	0.985	0.95	0.013701028908155334	0.019786095821936717
200	0.99	1.0	0.00895996206789775	0.014421496268395723
300	0.99	1.0	0.007106019201008822	0.011353434230923144
400	0.99	1.0	0.006006455857975831	0.008998822622017566
500	0.99	1.0	0.005216209437758075	0.007154118554173167
600	0.99	1.0	0.004614743963656527	0.005772731061210722
700	0.99	1.0	0.0041208299906332155	0.004751372507841446
800	0.995	1.0	0.0036896815557836025	0.00403527209838952
900	0.995	1.0	0.00330827252064156	0.003613454303982732
1000	0.995	1.0	0.002969600564842086	0.0034565135521732056
1250	1.0	1.0	0.0022756264492993046	0.003294462191652809
1500	1.0	1.0	0.0017515995640570775	0.0031434991935699757
1750	1.0	1.0	0.0013534801309948205	0.0030009373674439997
2000	1.0	1.0	0.00104971741087786	0.0028642505889893673

As number of epochs increases, both training and validation accuracy increases (towards 1) and the training and validation loss decreases (towards 0). Note that after some number of epochs, values are consistent indicating that runtime can be minimized without impacting the quality of the predictor by selecting the number of epochs appropriately.

Epochs = 200, activation function = linear, seed = 10

Learning Rate	T_{acc}	V_{acc}	T_{loss}	V_{loss}
0	0.335	0.35	5.962786776737121	5.913201429964414
0.0025	0.41	0.4	0.3034516330049272	0.26598912975788
0.005	0.855	0.8	0.17724470931048852	0.1577264390533794
0.075	0.98	0.9	0.05580884878945078	0.06033041881357254
0.01	0.905	1.0	0.09933907250930325	0.08654647580731951
0.025	0.965	0.95	0.061534818373363685	0.05898185772223632
0.05	0.98	0.9	0.05622550678405133	0.05966876221101075
0.075	0.98	0.9	0.05580884878945078	0.06033041881357254
0.1	0.98	0.9	0.055698852978373965	0.06055491146934686
0.125	0.98	0.9	0.05565612575243218	0.06068514155667848
0.15	0.98	0.9	0.05563882087822391	0.060775681856141336
0.175	0.98	0.9	0.055631798091640636	0.060838198094062976
0.2	0.98	0.9	0.055628952748631715	0.06088013562209923
0.25	0.98	0.9	0.055627340610664565	0.06092005933677107
0.3	0.665	0.65	8.597472884194297e+50	9.102405921294252e+50
0.5	0.665	0.65	2.941950013088329e+184	3.1147319194826933e+184

Esmat Sahak
ECE324

1	0.665	0.65	inf	inf
2	0.665	0.65	inf	inf

The accuracy and loss values are appropriate when the learning rate is within a certain range (dependent on the activation function). In the linear case, an appropriate learning rate ranged from 0.01-0.25. If the learning rate is too low, progress is slow, and an excessive number of epochs is needed to produce desirable results. If the learning rate is too high, the loss function diverges (shown in table).

Epochs = 200, learning rate = 0.1, seed = 10

Activation Function	T_{acc}	V_{acc}	T_{loss}	V_{loss}
Linear	0.98	0.9	0.055698852978373965	0.06055491146934686
ReLU	0.99	1.0	0.00895996206789775	0.014421496268395723
Sigmoid	0.965	0.95	0.0990695435737731	0.1127002400365859

Note that the performance of the activation function is dependent on the learning rate and number of epochs. Generally, it was observed that ReLU served as the better activation function. Issues with sigmoid activation functions include the vanishing gradient problem as the sigmoid functions grows very slowly (derivative of a sigmoid function, s', bounded from 0 to 1 satisfies $0 < s' < 0.25$). Note that parameters (weights, bias) can diverge with linear functions upon application of the gradient descent method. ReLU mitigates this problem (to some extent) as its gradient is either zero or positive.

Epochs = 200, learning rate = 0.005, activation function = linear

Seed	T_{acc}	V_{acc}	T_{loss}	V_{loss}
0	0.49	0.35	0.18486027250229328	0.22152522257651358
1	0.885	0.65	0.13589656272720796	0.20053852797296878
5	0.48	0.45	0.2309844104993518	0.2753840091039139
10	0.855	0.8	0.17724470931048852	0.1577264390533794
100	0.83	0.65	0.2062775545674016	0.3315125003692077

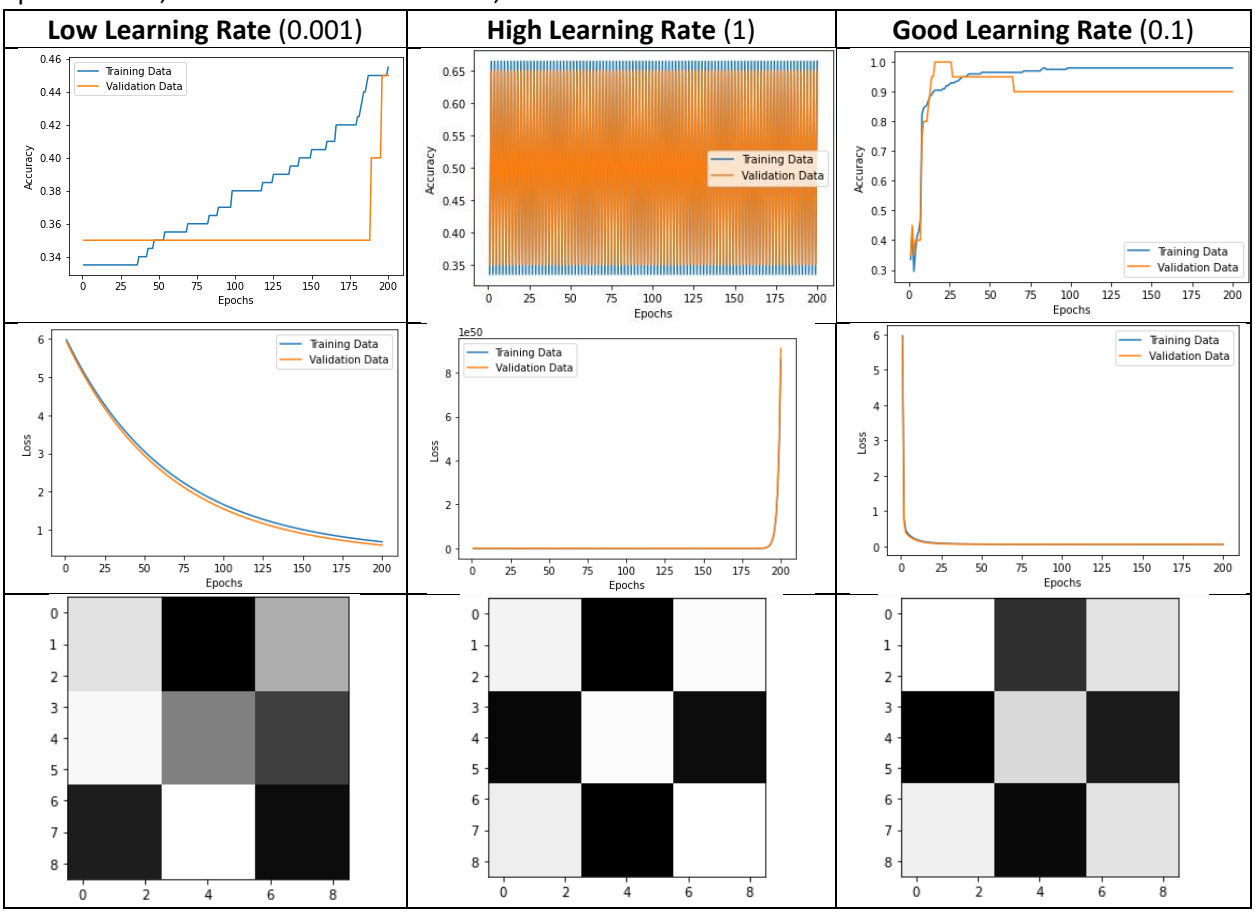
Different seeds randomize initial values of the parameters. This helps avoid consistently reaching poor local minima (i.e. there exist other parameter values which further minimize loss).

To minimize the number of epochs, consider accuracies of 0.95 to be acceptable.
Fewest epochs = 15, activation function = ReLU, learning rate = 0.145, seed = 40 → $T_{acc} = 0.95$, $V_{acc} = 1$, $T_{loss} = 0.04879350696910327$, $V_{loss} = 0.04253585316259109$

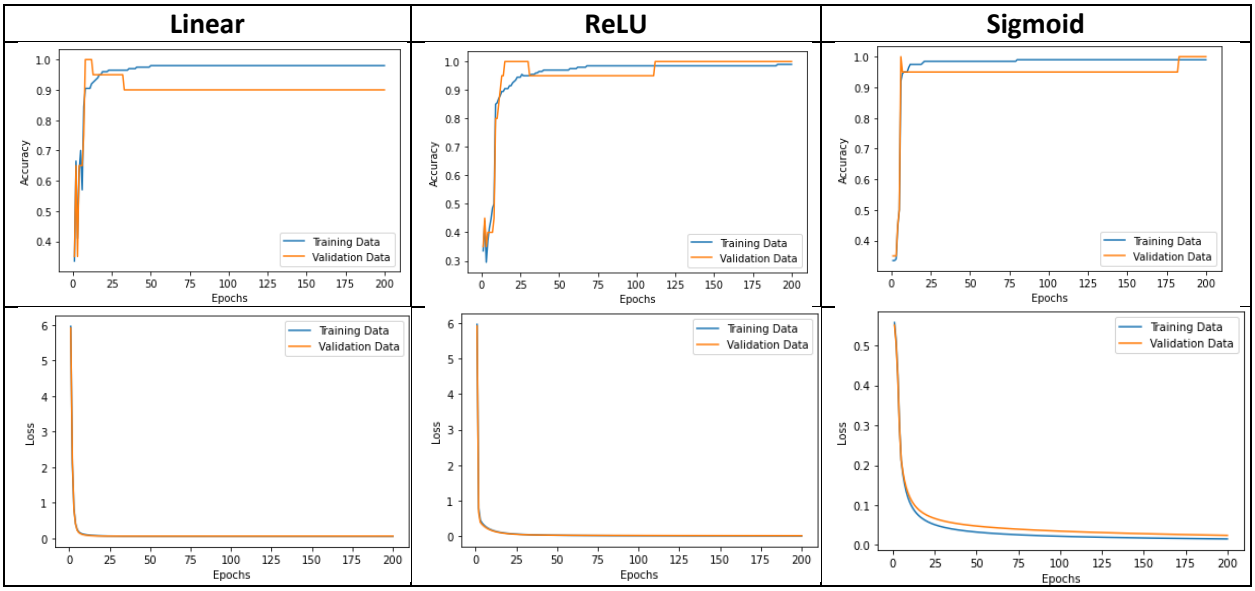
A low learning rate implies that more epochs are required for the loss and accuracy to converge to appropriate values. Given that the learning rate is low, progress per epoch is slow (i.e. parameters adjust very slightly) which is problematic in terms of runtime. The plots below demonstrate that 200 epochs are insufficient to attain the desirable convergence behaviour. A high learning rate implies adjustments to parameters per epoch are too high, this introduces complications such as potential divergent loss behaviour if an excessive number of epochs is applied. Lowering the number of epochs may help avoid this divergence, but subpar results are still achieved as the optimizing step is too large. A good learning rate optimizes both performance and efficiency – accuracy and loss converge to 1 and 0 respectively in the fewest number of epochs.

Esmat Sahak
ECE324

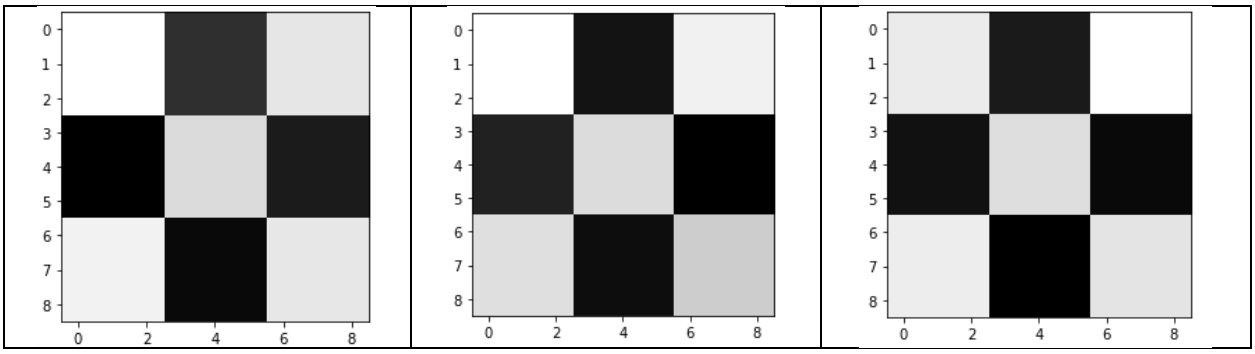
Epochs = 200, activation function = linear, seed = 10



Epochs = 200 and seed = 10 for all plots.
Learning rate: linear = 0.2, ReLU = 0.1, sigmoid = 2



Esmat Sahak
ECE324



Part 2

For the single-neuron classifier that you instantiate, what is the full name of the tensor object that contains the weights, and what is the name of the object that contains the bias?

smallNN.fc1.weight and smallNN.fc1.bias are the names of the tensor objects that contains the weight and bias respectively.

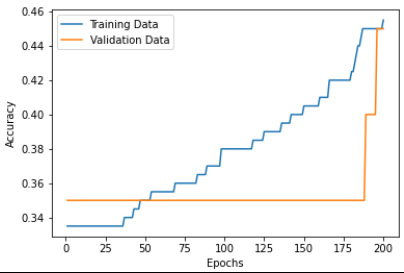
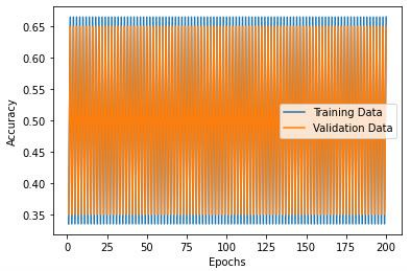
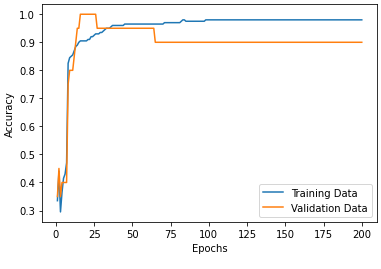
What is the name of the tensor object that contains the calculated gradients of the weights and the bias? (This was not covered in class).

smallNN.fc1.weight.grad and smallNN.fc1.bias.grad are the names of the tensor objects that contains the gradients of the weights and bias respectively.

Which part of your code computes gradients (i.e. give the line of code that causes the gradients to be computed). Explain, in a general way, what this line must cause to happen to compute the gradients, and how PyTorch 'knows' how to compute the gradients.

The line that computes the gradient is loss.backward(). This line induces backpropagation and computes the gradients with respect to the parameters (using the chain rule) that have the attribute requires_grad = True. PyTorch knows how to do this as Torch tensors maintain a record of all operations applied to them.

Epochs = 200, seed = 10

Low Learning Rate (0.001)	**High Learning Rate** (0.3)	**Good Learning Rate** (0.1)
		

Esmat Sahak
ECE324