1. Deep Learning Principles

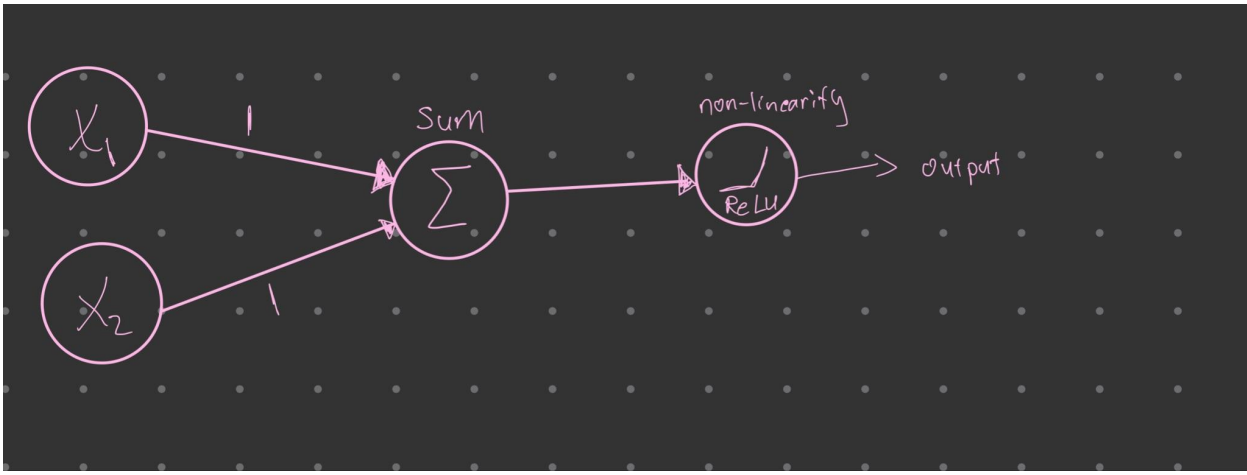
a. The first network initializes random weights allowing backpropagation to gradually adjust the weights in order to minimize the loss function as ReLU activation does not saturate for positive weights hence preventing gradients from vanishing in deep networks as discussed in lecture 7 (slide 58). Now, for the second network, all the weights are initialized to zero therefore their sum will be zero and ReLU activation saturates at zero, resulting in 'dead units' where the gradient is zero; as the gradient is zero at all these weights, then the backpropagation algorithm does not change the weights using a gradient of zero.

b. The first network learned at a bit better with the sigmoid (0.001 difference in test error compared to ReLU) however its loss function took a lot longer to plateau after around 2000 iterations compared to 197 for ReLu. The sigmoid model learned slower because as seen in lecture 7 slide on vanishing gradients with the sigmoid's saturating nonlinearities having small derivatives almost everywhere and in backpropagation, the product of many small terms goes to zero hence the weights are updated slowly compared to ReLu as in the positive region, ReLU does not saturate, preventing gradients from vanishing in deep networks.Also, the decision boundary looks a lot smoother for sigmoid (almost buble like) which makes sense as the sigmoid is differentiable everywhere compared to the ReLU (more hexagon like) which is differentiable only in the positive x. For the second network, it learned a lot better with both lower in sample and out of sample errors as it now developed a linear boundary for classification compared to no boundary at all with ReLU. However, this only started happening after over 3000 iterations as for the sigmoid the gradient is extremely small but not zero hence it is slowly but surely updating each weight hence it will take significantly more time, due to the

extremely small gradients, but the sigmoid would eventually learn the model even with the weights initialized at zero.

c. Looping through all the negative examples first will introduce a large negative bias in the weights as Stochastic Gradient Descent updates weights using the gradient at each point hence also resulting in the misclassification of positive labels as all the sum neurons will output negative values hence ReLU saturates at zero, resulting in 'dead units' where the gradient is zero hence the weights will no longer be updated, as seen in part A, for positive labels once the gradient is already zero hence the model stops learning in this case.

d.



Aside: As the derivative of ReLu is 1 for any positive number hence we can use any weight (x, y) for all x, y >= 1 instead of 1, 1 as we did above.

e. The minimum number of fully-connected layers with ReLU units needed to implement XOR is two, all having ReLU activation as mentioned, with the first hidden layer containing two sum units separating our inputs into two linear outputs (0 or 1) and the second hidden layer containing a single sum unit to produce a our output feature. A network with fewer layers cannot be computed by XOR because XOR is a nonlinear function and each unit with ReLU activation computes only linear functions hence XOR needs to be split into two linear functions which is what our first layer is doing.

2. Inside a Neural Network

a. Torch: 1.13.1; Torchvision: 0.14.1

b. First, I dropped all of the "Missing"/"Unknown" deaths from the loaded data frame as we are doing classification. Next, I converted the date columns in our data into something we can process hence int64. Then, I transformed the other columns that are not dates into float representations. I chose to grab the first 11 columns, using list splicing, as my features because they were the easiest/most convenient to grab because there are 11 columns before our dependent variable. Also, choosing more input variables will allow us to better fit our data as we'll have a more complex model to work with. Then, I grabbed all of our death_yn dependent variables by indexing our data frame. Then, I used Sklearn model selection to randomly split out data into a training (80%) and test set (20%). Then I converted my data into torch sensors and created datasets and data loaders using Pytorch.

c. [redacted] (CODE FOR C & D) See Code for Prob 2

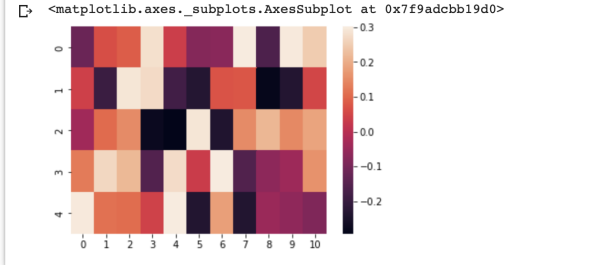
```
Test set: Average loss: 0.0090, Accuracy: 50600/51943 (97.4145)
```

▾ Weight Matrix Visualization

```
weights = model.state_dict()['0.weight']

sns.heatmap(weights)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9adcbb19d0>
```



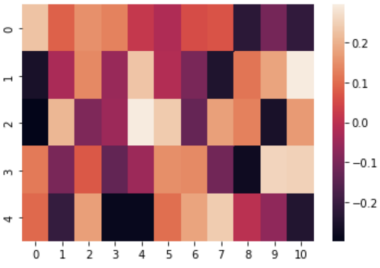
d.

```
Test set: Average loss: 0.0042, Accuracy: 50600/51943 (97.4145)
```

```
weights = model2.state_dict()['0.weight']

sns.heatmap(weights)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9ae3788430>
```



The 1-layer and 2-layer models have similar losses because our input size is small hence it can be classified/learned using 1-layer therefore adding another layer is unnecessary as seen above.

3. Depth vs Width on the MNIST Dataset

a. [redacted]

See Code for Prob3A

```
[ ]  print(len(train_dataset), type(train_dataset[0][0]), type(train_dataset[0][1]))
     print(len(test_dataset), type(test_dataset[0][0]), type(test_dataset[0][1]))
     print(train_dataset[0][0].size())
```

```
60000 <class 'torch.Tensor'> <class 'int'>
10000 <class 'torch.Tensor'> <class 'int'>
torch.Size([1, 28, 28])
```

The image height is 28 pixels. The image width is 28 pixels. The values in each array index represents if a part of the digit is present in that pixel. There are 60,000 images in the training set and 10,000 images in the testing set.

b. [redacted]

See Code for Prob3B-D

```
[36]     # Track loss each epoch
         print('Train Epoch: %d  Loss: %.4f' % (epoch + 1,  loss.item()))
```

```
Train Epoch: 1  Loss: 0.1254
Train Epoch: 2  Loss: 0.2105
Train Epoch: 3  Loss: 0.1366
Train Epoch: 4  Loss: 0.0387
Train Epoch: 5  Loss: 0.1046
Train Epoch: 6  Loss: 0.0460
Train Epoch: 7  Loss: 0.0662
Train Epoch: 8  Loss: 0.0623
Train Epoch: 9  Loss: 0.0286
Train Epoch: 10  Loss: 0.0502
```

```
# Putting layers like Dropout into evaluation mode
model.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch l
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0028, Accuracy: 9757/10000 (97.5700)
```

```
model = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an im
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 100),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(100, 10)
)
print(model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=100, bias=True
  (2): ReLU()
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=100, out_features=10, bias=True)
)
```

```
[34] optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
     loss_fn = nn.CrossEntropyLoss()
```

c.

See Code for Prob3B-D

```
model2 = nn.Sequential(
    nn.Flatten(),
    nn.Linear(784, 148),
    nn.ReLU(),
    nn.Dropout(0.23),
    nn.Linear(148, 52),
    nn.ReLU(),
    nn.Dropout(0.23),
    nn.Linear(52, 10)
)
print(model2)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=148, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.23, inplace=False)
  (4): Linear(in_features=148, out_features=52, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.23, inplace=False)
  (7): Linear(in_features=52, out_features=10, bias=True)
)
```

```
Train Epoch: 1  Loss: 0.0176
Train Epoch: 2  Loss: 0.0041
Train Epoch: 3  Loss: 0.0026
Train Epoch: 4  Loss: 0.1498
Train Epoch: 5  Loss: 0.0247
Train Epoch: 6  Loss: 0.0029
Train Epoch: 7  Loss: 0.0269
Train Epoch: 8  Loss: 0.0231
Train Epoch: 9  Loss: 0.0023
Train Epoch: 10  Loss: 0.0207
```

```
# Putting layers like Dropout into evaluation mode
model2.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model2(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of th
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0026, Accuracy: 9807/10000 (98.0700)
```

d.

See Code for Prob3B-D

```
model3 = nn.Sequential(
    # In problem 2, we don't use the 2D structure of an image at all. Our net
    # takes in a flat vector of the pixel values as input.
    nn.Flatten(),
    nn.Linear(784, 500),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(500, 350),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(350, 150),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.Linear(150, 10)
)
print(model)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=100, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.5, inplace=False)
  (4): Linear(in_features=100, out_features=10, bias=True)
)
```

```
Train Epoch: 1  Loss: 0.0042
Train Epoch: 2  Loss: 0.0001
Train Epoch: 3  Loss: 0.0046
Train Epoch: 4  Loss: 0.0184
Train Epoch: 5  Loss: 0.1198
Train Epoch: 6  Loss: 0.0114
Train Epoch: 7  Loss: 0.0023
Train Epoch: 8  Loss: 0.0013
Train Epoch: 9  Loss: 0.0001
Train Epoch: 10  Loss: 0.0003
```

```
# Putting layers like Dropout into evaluation mode
model3.eval()

test_loss = 0
correct = 0

# Turning off automatic differentiation
with torch.no_grad():
    for data, target in test_loader:
        output = model3(data)
        test_loss += loss_fn(output, target).item()  # Sum up batch loss
        pred = output.argmax(dim=1, keepdim=True)  # Get the index of the
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(test_loader.dataset)

print('Test set: Average loss: %.4f, Accuracy: %d/%d (%.4f)' %
      (test_loss, correct, len(test_loader.dataset),
       100. * correct / len(test_loader.dataset)))
```

```
Test set: Average loss: 0.0029, Accuracy: 9853/10000 (98.5300)
```

4. Convolutional Neural Networks

a. A benefit of Zero-padding is that it preserves border information hence allowing for better learning as the full image is used. A drawback of zero-padding is that it increases computation and memory usage, and may result in overfitting as we are using the entire image.

b. The number of parameter = 8 filters * size = 8 * (5 * 5 * 3) + bias = 8 *5*5*3 + (1*8) = 608 = the number of parameters = weights. The shape of the output tensor is 28 X 28 X 8.

c. Each sublist is a row and the full list is our matrix in the list of lists:

   i. (1st matrix):
   [[(1+1+1+1)/4), ((1+1+0+0)/4)], [(1+1+0+0)/4), ((1+0+0+0)/4)]] =
   [[1, 0.5], [0.5, 0.25]]
   now repeat process for remaining 3 matrices to get:
   (2nd matrix):
   [[0.5, 1], [0.25, 0.5]]
   (3rd matrix):
   [[0.25, 0.5], [0.5, 1]]
   (4th matrix):
   [[0.5, 0.25], [1, 0.5]]

   ii. The max in each 2x2 subgrid is always 2:
   (1st matrix):
   [[1, 1], [1, 1]]
   (2nd matrix):
   [[1, 1], [1, 1]]
   (3rd matrix):
   [[1, 1], [1, 1]]
   (4th matrix):
   [[1, 1], [1, 1]]

d. Pooling also helps in reducing the number of parameters in the network by summarizing the information in each region of the feature map and discarding spatial information making the network less prone to overfitting on small distortions in the dataset and allowing it to scale to larger images. Furthermore, pooling layers operates on small

regions in our feature map, and summarizing the information in these regions can make the network invariant to small translations and rotations in the input image resulting in a more robust CNN that can handle small distortions in the dataset.

e. Compared to grid search, random search is more efficient as it doesn't need to test all possible combinations of hyperparameters which is important when there are many hyperparameters or when the search space is very large. Random search can also more efficiently explore the hyperparameter space as unlike grid search, which may get stuck in local minima, random search can jump around the search space, which can help find global minima in large datasets. However, random search may not perform well on small search spaces or when the hyperparameters are tightly correlated due to its jumping around hence it may miss some features.. In these cases, a grid search may be more effective as it will explore the entire search space in a systematic way.

f. [redacted]

See Code for Prob4

```python
# sample model
import torch.nn as nn

model = nn.Sequential(
    nn.Conv2d(1, 32, kernel_size=(3,3)),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(p=0.2),

    nn.Conv2d(32, 32, kernel_size=(3,3)), nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Dropout(p=0.2),

    nn.Flatten(),
    nn.Linear(25*32, 64),
    nn.ReLU(),
    nn.Linear(64, 10)
    # PyTorch implementation of cross-entropy loss includes softmax layer
)
```

```python
# why don't we take a look at the shape of the weights for each layer
for p in model.parameters():
    print(p.data.shape)
```

```
torch.Size([32, 1, 3, 3])
torch.Size([32])
torch.Size([32, 32, 3, 3])
torch.Size([32])
torch.Size([32])
torch.Size([32])
torch.Size([64, 800])
torch.Size([64])
torch.Size([10, 64])
torch.Size([10])
```

```python
        print(f', val loss: {validation_loss_history[epoch,0]:0.4f}, val acc: {validation_accuracy_history[epoch,0]:0.4f}')
```

```
Epoch 1/10:..........
        loss: 0.4016, acc: 0.9193, val loss: 0.0928, val acc: 0.9707
Epoch 2/10:..........
        loss: 0.1021, acc: 0.9705, val loss: 0.1269, val acc: 0.9619
Epoch 3/10:..........
        loss: 0.0868, acc: 0.9746, val loss: 0.0790, val acc: 0.9789
Epoch 4/10:..........
        loss: 0.0779, acc: 0.9768, val loss: 0.0518, val acc: 0.9843
Epoch 5/10:..........
        loss: 0.0760, acc: 0.9786, val loss: 0.0726, val acc: 0.9794
Epoch 6/10:..........
        loss: 0.0722, acc: 0.9794, val loss: 0.0591, val acc: 0.9814
Epoch 7/10:..........
        loss: 0.0729, acc: 0.9796, val loss: 0.0488, val acc: 0.9858
Epoch 8/10:..........
        loss: 0.0681, acc: 0.9811, val loss: 0.0405, val acc: 0.9880
Epoch 9/10:..........
        loss: 0.0665, acc: 0.9820, val loss: 0.0443, val acc: 0.9885
Epoch 10/10:..........
        loss: 0.0664, acc: 0.9818, val loss: 0.0466, val acc: 0.9871
```