§1 Homework 6: Convolutions and Transformers

"I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

§1.1 Problem I

(a) We have $(4 \cdot 1 + 1) \cdot 8 = 40$ parameters. The output tensor has size $\frac{(200-4+2\cdot 0)}{2} \times 1 = 98 \times 1$.

(b) We have $(4 \cdot 4 \cdot 3 + 1) \cdot 8 = 392$. parameters. The size of the output tensor is given by

$$\frac{(28 - 4 + 2 \cdot 0)}{2} \times \frac{(28 - 4 + 2 \cdot 0)}{2} \times 3 = 12 \times 12 \times 3.$$

(c) The result of average pooling for each matrix is given by

$$\begin{bmatrix} \frac{8}{9} & \frac{2}{3} \\ \frac{2}{3} & \frac{5}{9} \end{bmatrix}, \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} \end{bmatrix}.$$

The result of max pooling for each matrix is given by

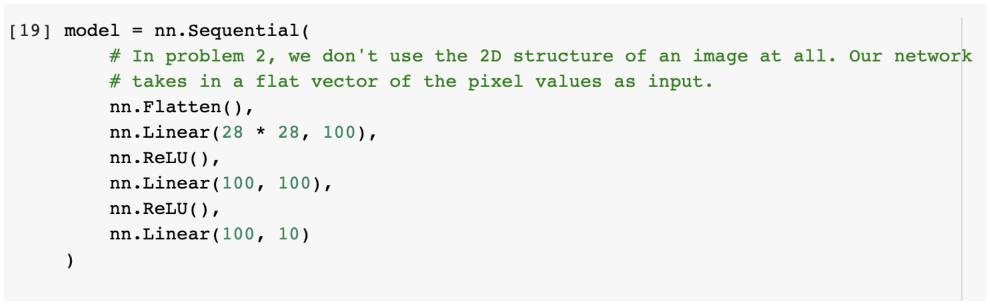
$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

(d) In the case of max pooling, since we have a small amount of noise, we can perform analysis on the pooled where each cell corresponds to the maximum value in a corresponding block. We also take advantage of the convolutional structure which has translational invariance and spatial locality. The case with average pooling is similar - with small distortions, we can take the average within blocks and perform analysis on the corresponding pooled tensor.

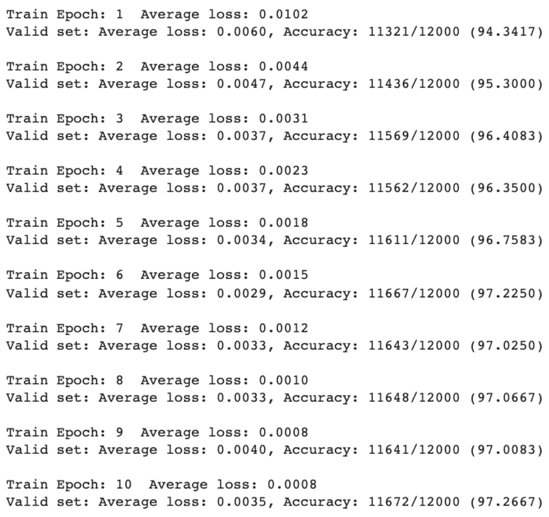
§1.2 Problem II

(a) The images are 28×28. The values are in the range $[0, 1]$ and correspond to the gray-scale opacity of the corresponding pixel - 0.0 corresponds to black, 1.0 corresponds to white, and they scale continuously in between these values. There are 48000 images in the training set. There are 10000 images in the test set.

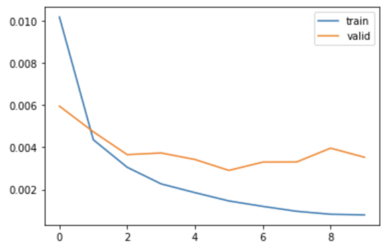
(b)



```
[19] model = nn.Sequential(
            # In problem 2, we don't use the 2D structure of an image at all. Our network
            # takes in a flat vector of the pixel values as input.
            nn.Flatten(),
            nn.Linear(28 * 28, 100),
            nn.ReLU(),
            nn.Linear(100, 100),
            nn.ReLU(),
            nn.Linear(100, 10)
        )
```



```
Train Epoch: 1  Average loss: 0.0102
Valid set: Average loss: 0.0060, Accuracy: 11321/12000 (94.3417)

Train Epoch: 2  Average loss: 0.0044
Valid set: Average loss: 0.0047, Accuracy: 11436/12000 (95.3000)

Train Epoch: 3  Average loss: 0.0031
Valid set: Average loss: 0.0037, Accuracy: 11569/12000 (96.4083)

Train Epoch: 4  Average loss: 0.0023
Valid set: Average loss: 0.0037, Accuracy: 11562/12000 (96.3500)

Train Epoch: 5  Average loss: 0.0018
Valid set: Average loss: 0.0034, Accuracy: 11611/12000 (96.7583)

Train Epoch: 6  Average loss: 0.0015
Valid set: Average loss: 0.0029, Accuracy: 11667/12000 (97.2250)

Train Epoch: 7  Average loss: 0.0012
Valid set: Average loss: 0.0033, Accuracy: 11643/12000 (97.0250)

Train Epoch: 8  Average loss: 0.0010
Valid set: Average loss: 0.0033, Accuracy: 11648/12000 (97.0667)

Train Epoch: 9  Average loss: 0.0008
Valid set: Average loss: 0.0040, Accuracy: 11641/12000 (97.0083)

Train Epoch: 10  Average loss: 0.0008
Valid set: Average loss: 0.0035, Accuracy: 11672/12000 (97.2667)
```



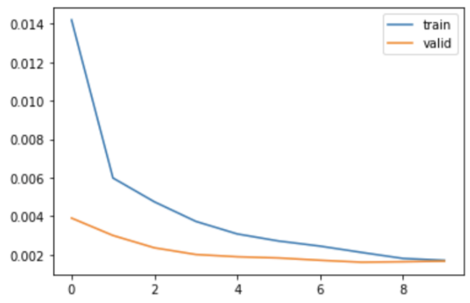
```
Test set: Average loss: 0.0030, Accuracy: 9760/10000 (97.6000)
(0.0029631294265225505, 0.976)
```

The first model I used worked using the default batch and epoch size. I essentially stacked two of the default linear models together, using a larger number of hidden layers.

```
[27] model = nn.Sequential(
         nn.Conv2d(1, 12, kernel_size=3, padding=1, stride=1),
         nn.ReLU(),
         nn.MaxPool2d(kernel_size=2),

         nn.Conv2d(12, 24, kernel_size=3, padding=1, stride=1),
         nn.ReLU(),
         nn.MaxPool2d(kernel_size=2),

         nn.Flatten(),
         nn.Linear(24 * 7 * 7, 32),
         nn.ReLU(),
         nn.Dropout(p=0.2),
         nn.Linear(32, 10)
     )

     # make sure the # of parameters is under 50,000
     count = 0
     for p in model.parameters():
         n_params = np.prod(list(p.data.shape)).item()
         count += n_params
     print(f'total params: {count}')
```

(c)

```
     total params: 40730

Train Epoch: 1  Average loss: 0.0142
Valid set: Average loss: 0.0039, Accuracy: 11537/12000 (96.1417)

Train Epoch: 2  Average loss: 0.0060
Valid set: Average loss: 0.0030, Accuracy: 11656/12000 (97.1333)

Train Epoch: 3  Average loss: 0.0048
Valid set: Average loss: 0.0024, Accuracy: 11732/12000 (97.7667)

Train Epoch: 4  Average loss: 0.0037
Valid set: Average loss: 0.0020, Accuracy: 11769/12000 (98.0750)

Train Epoch: 5  Average loss: 0.0031
Valid set: Average loss: 0.0019, Accuracy: 11793/12000 (98.2750)

Train Epoch: 6  Average loss: 0.0027
Valid set: Average loss: 0.0018, Accuracy: 11805/12000 (98.3750)

Train Epoch: 7  Average loss: 0.0025
Valid set: Average loss: 0.0017, Accuracy: 11816/12000 (98.4667)

Train Epoch: 8  Average loss: 0.0021
Valid set: Average loss: 0.0016, Accuracy: 11821/12000 (98.5083)

Train Epoch: 9  Average loss: 0.0018
Valid set: Average loss: 0.0016, Accuracy: 11828/12000 (98.5667)

Train Epoch: 10  Average loss: 0.0017
Valid set: Average loss: 0.0017, Accuracy: 11828/12000 (98.5667)
```

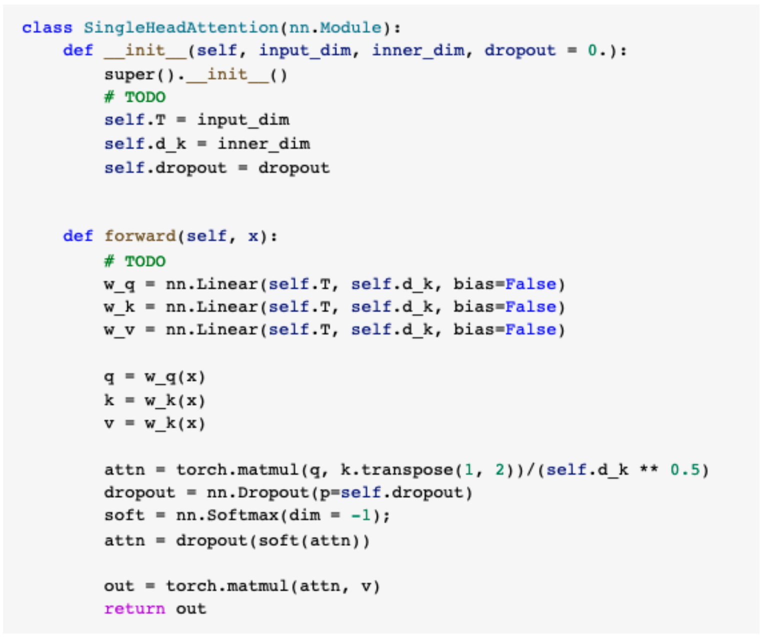


```
Test set: Average loss: 0.0012, Accuracy: 9880/10000 (98.8000)
(0.0012302720687239344, 0.988)
```

The first model I used worked using the default batch and epoch size. I stacked two of the default convolutional models together.

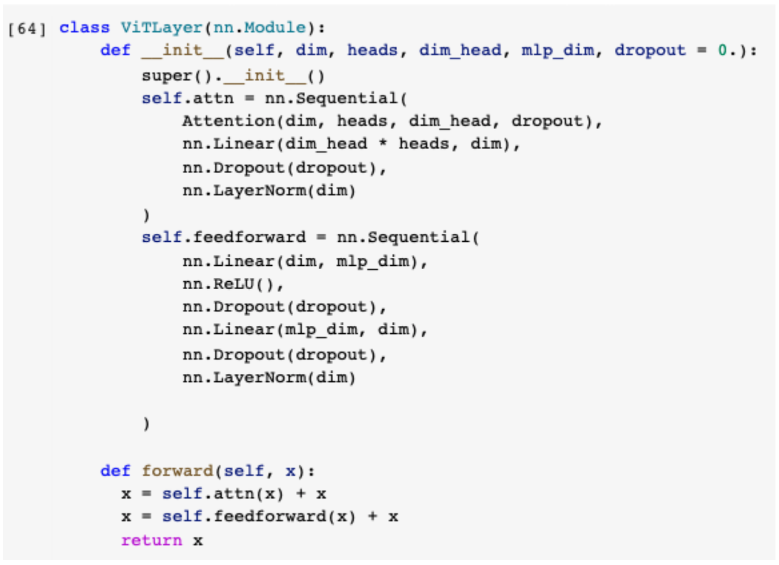
§1.3 Problem III

(a)



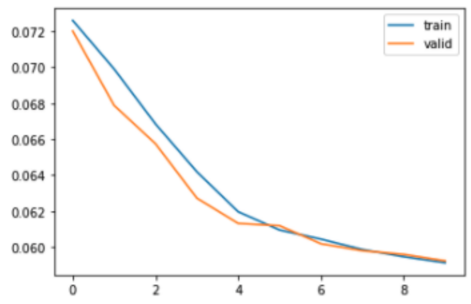
```
class SingleHeadAttention(nn.Module):
    def __init__(self, input_dim, inner_dim, dropout = 0.):
        super().__init__()
        # TODO
        self.T = input_dim
        self.d_k = inner_dim
        self.dropout = dropout


    def forward(self, x):
        # TODO
        w_q = nn.Linear(self.T, self.d_k, bias=False)
        w_k = nn.Linear(self.T, self.d_k, bias=False)
        w_v = nn.Linear(self.T, self.d_k, bias=False)

        q = w_q(x)
        k = w_k(x)
        v = w_k(x)

        attn = torch.matmul(q, k.transpose(1, 2))/(self.d_k ** 0.5)
        dropout = nn.Dropout(p=self.dropout)
        soft = nn.Softmax(dim = -1);
        attn = dropout(soft(attn))

        out = torch.matmul(attn, v)
        return out
```

(b)



```
[64] class ViTLayer(nn.Module):
         def __init__(self, dim, heads, dim_head, mlp_dim, dropout = 0.):
             super().__init__()
             self.attn = nn.Sequential(
                 Attention(dim, heads, dim_head, dropout),
                 nn.Linear(dim_head * heads, dim),
                 nn.Dropout(dropout),
                 nn.LayerNorm(dim)
             )
             self.feedforward = nn.Sequential(
                 nn.Linear(dim, mlp_dim),
                 nn.ReLU(),
                 nn.Dropout(dropout),
                 nn.Linear(mlp_dim, dim),
                 nn.Dropout(dropout),
                 nn.LayerNorm(dim)

             )

         def forward(self, x):
           x = self.attn(x) + x
           x = self.feedforward(x) + x
           return x
```

```
Train Epoch: 1  Average loss: 0.0726
Valid set: Average loss: 0.0720, Accuracy: 1425/12000 (11.8750)

Train Epoch: 2  Average loss: 0.0699
Valid set: Average loss: 0.0679, Accuracy: 2096/12000 (17.4667)

Train Epoch: 3  Average loss: 0.0668
Valid set: Average loss: 0.0657, Accuracy: 2531/12000 (21.0917)

Train Epoch: 4  Average loss: 0.0642
Valid set: Average loss: 0.0627, Accuracy: 2771/12000 (23.0917)

Train Epoch: 5  Average loss: 0.0620
Valid set: Average loss: 0.0613, Accuracy: 3065/12000 (25.5417)

Train Epoch: 6  Average loss: 0.0610
Valid set: Average loss: 0.0612, Accuracy: 2954/12000 (24.6167)

Train Epoch: 7  Average loss: 0.0604
Valid set: Average loss: 0.0602, Accuracy: 3117/12000 (25.9750)

Train Epoch: 8  Average loss: 0.0599
Valid set: Average loss: 0.0598, Accuracy: 3101/12000 (25.8417)

Train Epoch: 9  Average loss: 0.0595
Valid set: Average loss: 0.0596, Accuracy: 3125/12000 (26.0417)

Train Epoch: 10  Average loss: 0.0591
Valid set: Average loss: 0.0592, Accuracy: 3122/12000 (26.0167)
```



(c)

```
Test set: Average loss: 0.0582, Accuracy: 2691/10000 (26.9100)
(0.058205325961112976, 0.2691)
```