

ONE LATE HOUR

1 Deep Learning Principles [35 Points]

Relevant materials: lectures on deep learning

Problem A [5 points]: Backpropagation and Weight Initialization Part 1

Solution A.: *The second model doesn't learn because the weights are initialized to zero. This means during the forward pass all the activations for all the nodes will be zero due to the nature of the ReLU function. Thus, during back propagation one of the partial derivative $\frac{\partial s^{(l)}}{\partial W^{(l)}} = x^{(l-1)T} = 0$. Thus the model never learns so the test error stays at .508 which is error given by just guessing. The model where the weights are initialized randomly don't have that problem so it is able to learn and the test error drops rapidly to about .01.*

Problem B [5 points]: Backpropagation and Weight Initialization Part 2

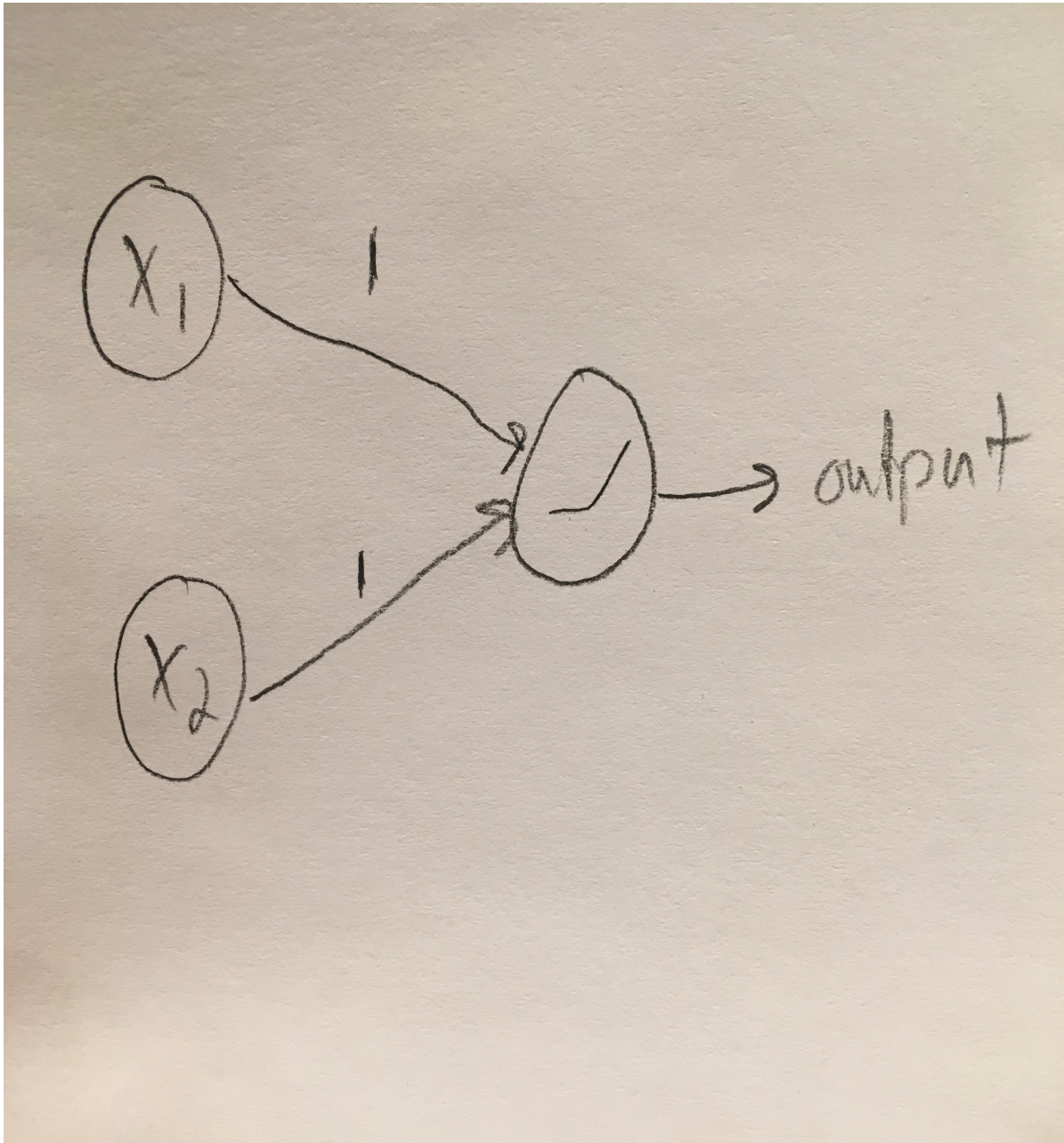
Solution B.: *The models trained using the sigmoid function take much longer to learn. This is in part due to the vanishing gradient problem as well as the derivative in general is always pretty low and is never 1 like ReLU. The vanishing gradient problem is when an input gets to large or small the derivative of the sigmoid goes to zero so learning stops occurring to an extent. With the randomly initialized weights it takes around 500 epochs to start seeing a significant decrease in test error. Again this is because sigmoid function leads to slower learning due to its derivative and vanishing gradients. The model with weights starting at 0 actually learn in this case because activations are always non zero with sigmoids but it is still very slow and takes almost 3200 epochs to learn. When this model does learn the weights all end up the same so we get a linear decision boundary because all the weights undergo the same update procedure so they always stay the same and you don't get the learning desired. As mentioned before the backpropagation doesn't result in no learning because the sigmoid always results in a non zero activation and is every where differentiable with a non zero derivative.*

Problem C: [10 Points]

Solution C: *When all the negative examples are looped through first, the model learns a heavy negative bias so all the negative points are classified correctly. Thus, when the positive points come around, the negative bias is so large that the positive points aren't labeled as such. Due to the nature of the ReLU function, all of the points being classified as negative means they will go to zero and "die" and thus we have the dying ReLU problem as described by the hint because learning won't happen after the death of the nodes.*

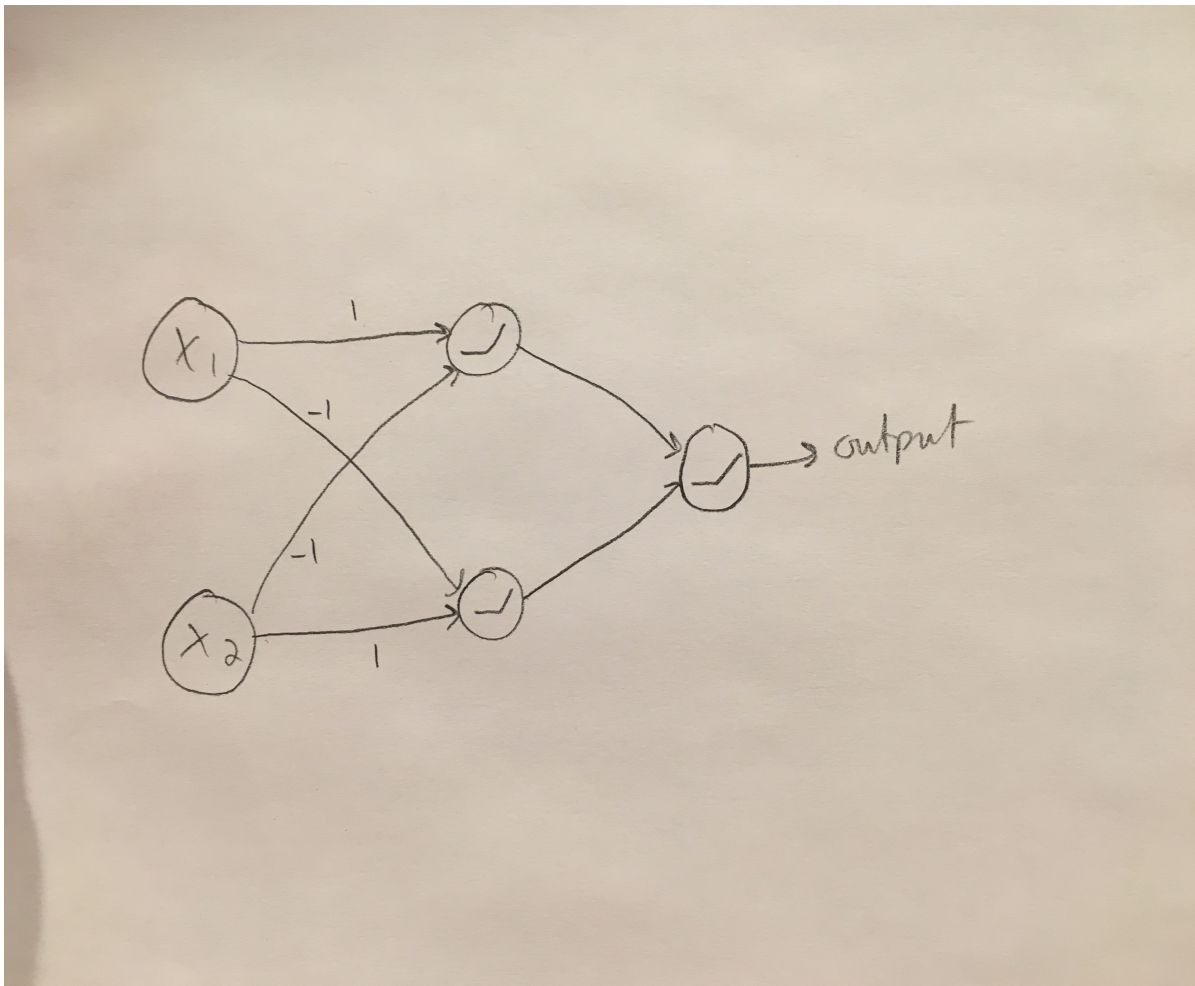
Problem D: Approximating Functions Part 1 [7 Points]

Solution D.:



Problem E: Approximating Functions Part 2 [8 Points]

Solution E.: A network with fewer layers would only be able to describe linearly separable data and as we saw from lecture we know XOR must be able to deal with non linearly separable data. Thus, we need one hidden layer instead of just an input and an output layer as we see in the OR network.



2 Depth vs Width on the MNIST Dataset [25 Points]

Problem A: Installation [2 Points]

Solution A:

Keras: 2.2.4

Tensorflow: 1.12.0

Problem B: The Data [1 Point]

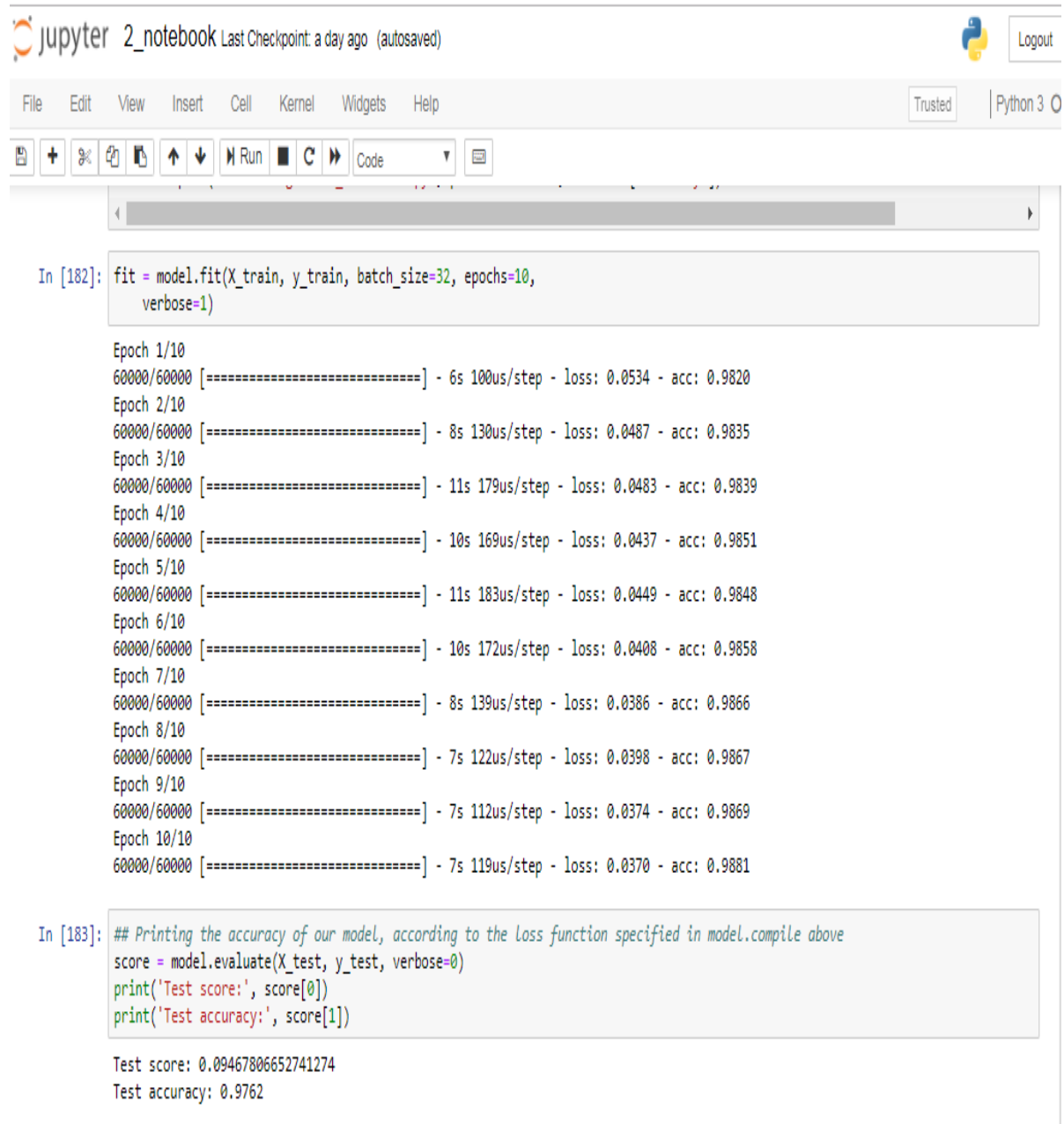
Solution B.: *The images are 28 pixels by 28 pixels. The values in each array index indicate the pixel intensity at the pixel which has a position corresponding to its' position in the array.*

Problem C: One-Hot Encoding [2 Points]

Solution C.: *The new shape of the training input is (60000, 784) because there are 60000 instances of hand-written digits and a 28 x 28 image has 784 pixels so when the image is made into a single vector it becomes of length 784.*

Problem D: Modeling Part 1 [8 Points]

Solution D: See code for neural net structure.



The screenshot shows a Jupyter Notebook titled "2_notebook" with a "Last Checkpoint: a day ago (autosaved)" status. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, running, and other actions. The code is written in Python 3. The first cell (In [182]) contains the training code, which prints the progress for each of the 10 epochs. The second cell (In [183]) contains the evaluation code, which prints the test score and accuracy.

```
In [182]: fit = model.fit(X_train, y_train, batch_size=32, epochs=10,
    verbose=1)

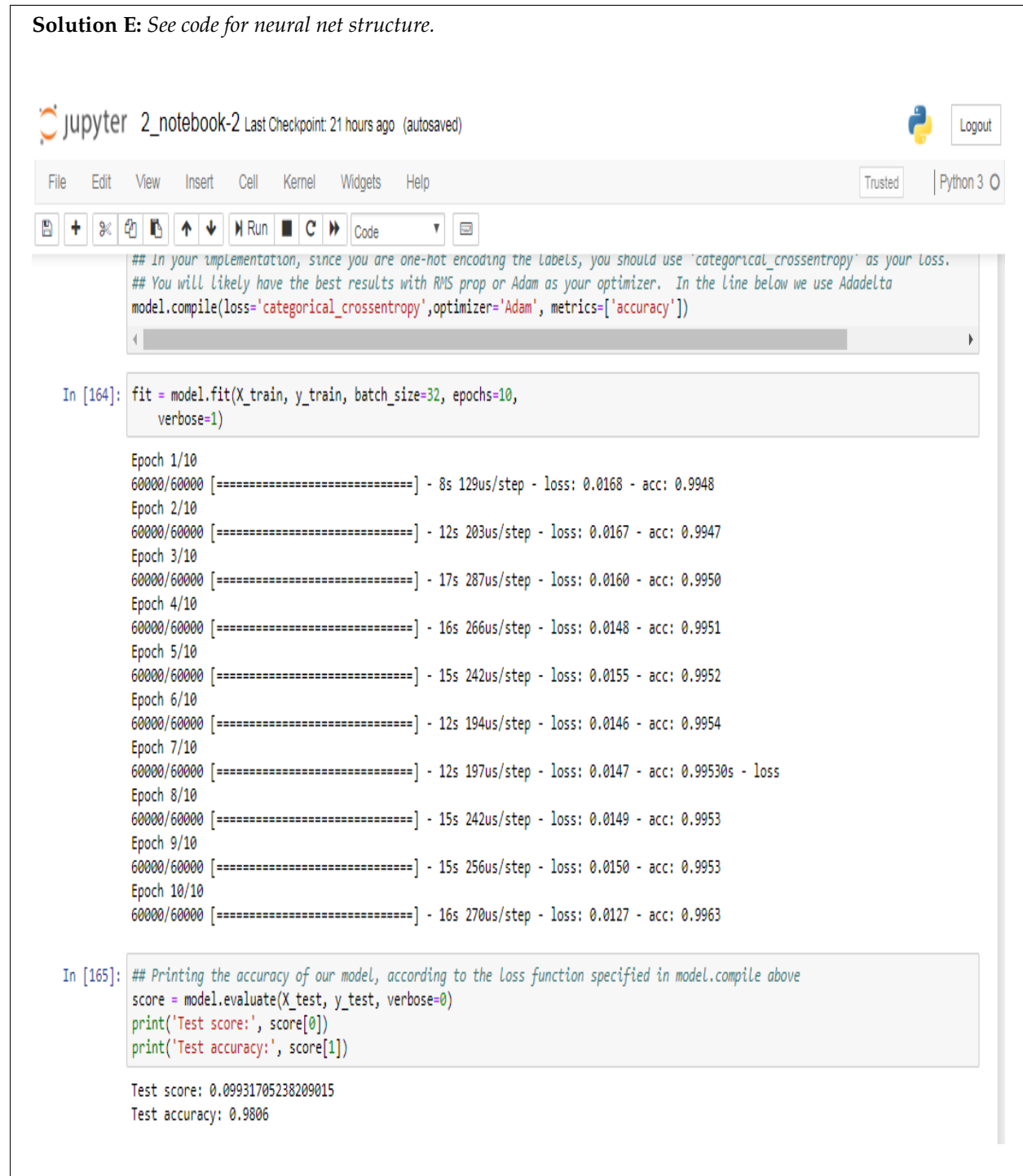
Epoch 1/10
60000/60000 [=====] - 6s 100us/step - loss: 0.0534 - acc: 0.9820
Epoch 2/10
60000/60000 [=====] - 8s 130us/step - loss: 0.0487 - acc: 0.9835
Epoch 3/10
60000/60000 [=====] - 11s 179us/step - loss: 0.0483 - acc: 0.9839
Epoch 4/10
60000/60000 [=====] - 10s 169us/step - loss: 0.0437 - acc: 0.9851
Epoch 5/10
60000/60000 [=====] - 11s 183us/step - loss: 0.0449 - acc: 0.9848
Epoch 6/10
60000/60000 [=====] - 10s 172us/step - loss: 0.0408 - acc: 0.9858
Epoch 7/10
60000/60000 [=====] - 8s 139us/step - loss: 0.0386 - acc: 0.9866
Epoch 8/10
60000/60000 [=====] - 7s 122us/step - loss: 0.0398 - acc: 0.9867
Epoch 9/10
60000/60000 [=====] - 7s 112us/step - loss: 0.0374 - acc: 0.9869
Epoch 10/10
60000/60000 [=====] - 7s 119us/step - loss: 0.0370 - acc: 0.9881

In [183]: ## Printing the accuracy of our model, according to the loss function specified in model.compile above
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

Test score: 0.09467806652741274
Test accuracy: 0.9762
```

Problem E: Modeling Part 2 [6 Points]

Solution E: See code for neural net structure.



The screenshot shows a Jupyter Notebook titled "2_notebook-2" with a last checkpoint 21 hours ago. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, and running code. The code in the notebook is as follows:

```
## In your implementation, since you are one-hot encoding the labels, you should use 'categorical_crossentropy' as your loss.
## You will likely have the best results with RMS prop or Adam as your optimizer. In the line below we use Adadelta
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

In [164]: fit = model.fit(X_train, y_train, batch_size=32, epochs=10,
                        verbose=1)

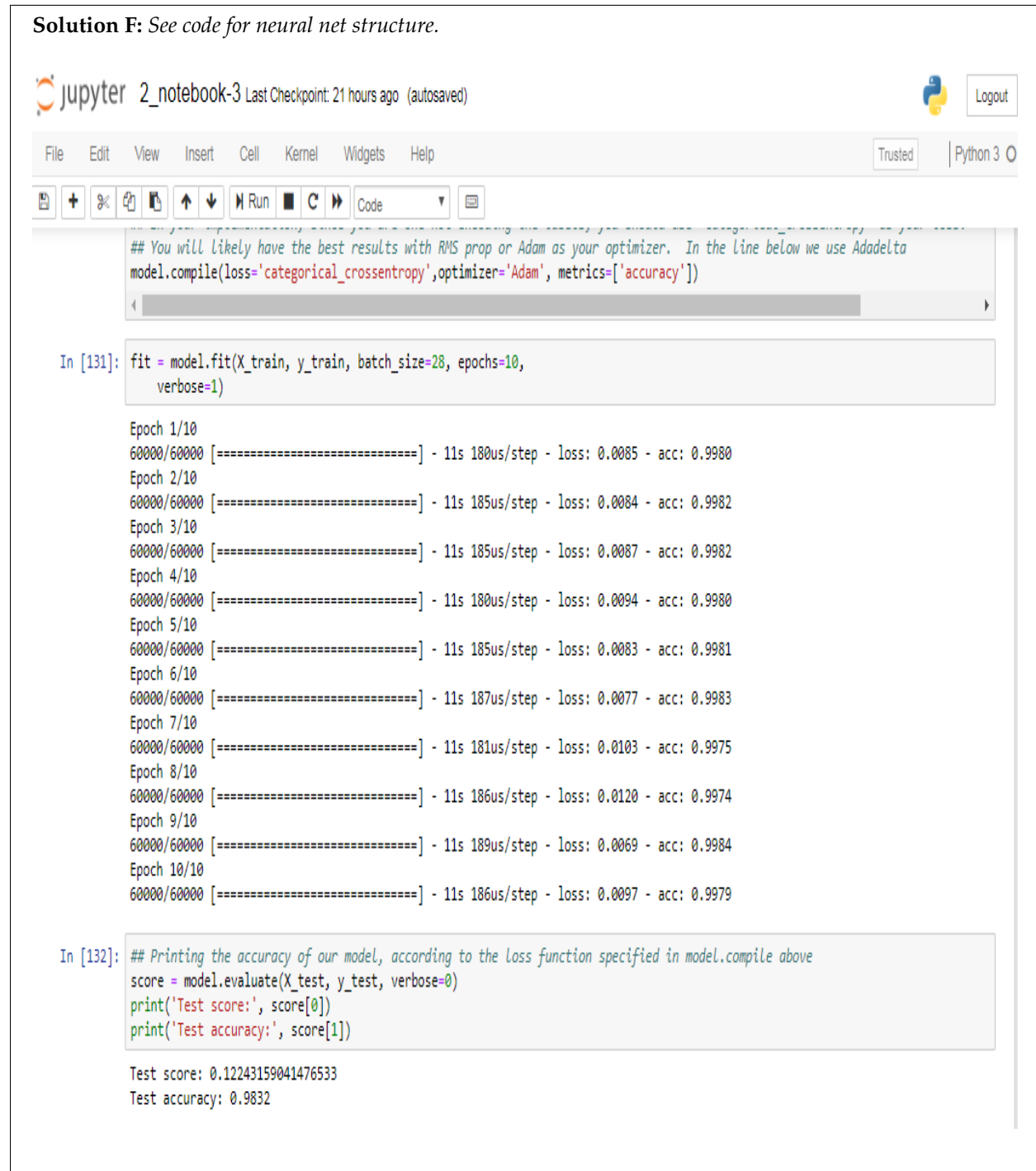
Epoch 1/10
60000/60000 [=====] - 8s 129us/step - loss: 0.0168 - acc: 0.9948
Epoch 2/10
60000/60000 [=====] - 12s 203us/step - loss: 0.0167 - acc: 0.9947
Epoch 3/10
60000/60000 [=====] - 17s 287us/step - loss: 0.0160 - acc: 0.9950
Epoch 4/10
60000/60000 [=====] - 16s 266us/step - loss: 0.0148 - acc: 0.9951
Epoch 5/10
60000/60000 [=====] - 15s 242us/step - loss: 0.0155 - acc: 0.9952
Epoch 6/10
60000/60000 [=====] - 12s 194us/step - loss: 0.0146 - acc: 0.9954
Epoch 7/10
60000/60000 [=====] - 12s 197us/step - loss: 0.0147 - acc: 0.99530s - loss
Epoch 8/10
60000/60000 [=====] - 15s 242us/step - loss: 0.0149 - acc: 0.9953
Epoch 9/10
60000/60000 [=====] - 15s 256us/step - loss: 0.0150 - acc: 0.9953
Epoch 10/10
60000/60000 [=====] - 16s 270us/step - loss: 0.0127 - acc: 0.9963

In [165]: ## Printing the accuracy of our model, according to the loss function specified in model.compile above
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

Test score: 0.09931705238209015
Test accuracy: 0.9806
```

Problem F: Modeling Part 3 [6 Points]

Solution F: See code for neural net structure.



The screenshot shows a Jupyter Notebook titled "2_notebook-3" with a last checkpoint 21 hours ago. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for file operations, running, and code execution. The code is written in Python and uses Keras for model compilation and training. The first cell shows the model compilation with categorical crossentropy loss, Adam optimizer, and accuracy metric. The second cell shows the training process for 10 epochs, with detailed output for each epoch including time, loss, and accuracy. The third cell shows the evaluation of the model on test data, printing the test score and accuracy.

```
## You will likely have the best results with RMS prop or Adam as your optimizer. In the line below we use Adadelta
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

In [131]: fit = model.fit(X_train, y_train, batch_size=28, epochs=10,
                        verbose=1)

Epoch 1/10
60000/60000 [=====] - 11s 180us/step - loss: 0.0085 - acc: 0.9980
Epoch 2/10
60000/60000 [=====] - 11s 185us/step - loss: 0.0084 - acc: 0.9982
Epoch 3/10
60000/60000 [=====] - 11s 185us/step - loss: 0.0087 - acc: 0.9982
Epoch 4/10
60000/60000 [=====] - 11s 180us/step - loss: 0.0094 - acc: 0.9980
Epoch 5/10
60000/60000 [=====] - 11s 185us/step - loss: 0.0083 - acc: 0.9981
Epoch 6/10
60000/60000 [=====] - 11s 187us/step - loss: 0.0077 - acc: 0.9983
Epoch 7/10
60000/60000 [=====] - 11s 181us/step - loss: 0.0103 - acc: 0.9975
Epoch 8/10
60000/60000 [=====] - 11s 186us/step - loss: 0.0120 - acc: 0.9974
Epoch 9/10
60000/60000 [=====] - 11s 189us/step - loss: 0.0069 - acc: 0.9984
Epoch 10/10
60000/60000 [=====] - 11s 186us/step - loss: 0.0097 - acc: 0.9979

In [132]: ## Printing the accuracy of our model, according to the loss function specified in model.compile above
score = model.evaluate(X_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])

Test score: 0.12243159041476533
Test accuracy: 0.9832
```

3 Convolutional Neural Networks [40 Points]

Problem A: Zero Padding [5 Points]

Solution A: *One benefit of padding is that it allows us to preserve the original size of the image. This means we can keep information about the border as well as make deeper networks. Both of these things are ideal. One downside of zero padding is that it takes much more time and space to do the computations necessary to learn.*

5 x 5 Convolutions

Problem B [2 points]:

Solution B.: *The number of parameters is $8 \text{ filters} * 5 * 5 * 3 + 8 \text{ biases} = 608$. Each filter has a bias.*

Problem C [3 points]:

Solution C.: The output tensor has the shape $28 \times 28 \times 8$. The 8 is explained by the number of filters. The three channels of the filter line up with those of the image. The other can be explained by the equation for the size of one of the output dimensions which is given by $\text{output size} = \frac{W - K + 2P}{S} + 1$ Where W is the input size, K is the filter size, P is the padding size and S is the stride. Since $P = 0$ and $S = 1$ we get $\text{output size} = W - K + 1$ which will turn the height and width from 32 to 28 in our case.

Max/Average Pooling

Problem D [3 points]:

Solution D.:

$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.25 \end{bmatrix}, \begin{bmatrix} 0.5 & 1 \\ 0.25 & 0.5 \end{bmatrix}, \begin{bmatrix} 0.25 & 0.5 \\ 0.5 & 1 \end{bmatrix}, \begin{bmatrix} 0.5 & 0.25 \\ 1 & 0.5 \end{bmatrix}$$

Problem E [3 points]:

Solution E.:

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

Problem F [4 points]:

Solution F: Pooling would be advantageous because by grouping pixels into regions we have mitigated the effect of distortions like a missing pixel because they will be accounted for by the pixels surrounding them in their region. Similarly the different angles and locations are a form of noise and that problem can be mitigated because pooling makes it such that the location of features is less important than their existence.

Keras implementation

Problem G [20 points]:

Solution G.: See code for structure of the CNN.

The loss and accuracy for the dropouts ranging from 0 to 0.9 are [[0.9714], [0.9694], [0.97], [0.9688], [0.9578], [0.9524], [0.9474], [0.9279], [0.8897], [0.8479]]. Thus, we can see 0.2 gives us the best non zero dropout. Changing the dropout was the most effective for of regularization. Having a dropout that is too high greatly affected learning as it caused underfitting. Batch normalization also proved useful as it leads to a more stable distribution of inputs which allows for accelerated learning. Testing hyperparameters by just learning over one epoch can lead to problems with types of regularizaiton that improve your model over each epoch or that don't work in a dramatic fashion. Some forms of reularization require multiple passes to be really effective so this way of testing won't always be indicative of final performance with more epochs. I couldn't find a layerwise regularization that performed significantly better than the model without it so I didn't include it.