Overfitting Analysis/Adjusting the hyperparameters

HAND-TUNING SOME HYPERPARAMETERS AND OVERFITTING ANALYSIS

1. My "first draft" of the model

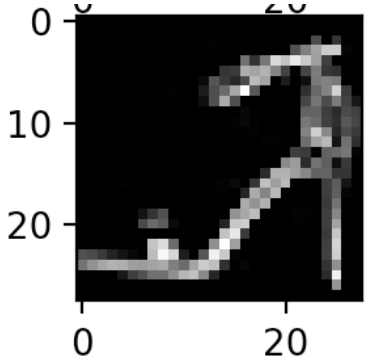
Here is a description of the model:
number of filters: 8
filter size: 3x3
pool size: 2x2
strides, padding, convolutional layer activation: not specified
epochs: 3

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1)),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8794
testing accuracy: .8706

- Since the training accuracy is slightly higher than the testing accuracy (after being run multiple times), this model may be overfitting a tiny bit
- **Analysis of the 87% accuracy:**
  - **When I randomly selected 9 images and tried to classify them, I classified 8/9 correctly (about 89% accuracy), which is higher than the model's accuracy (but the sample size is tiny and thus the results are not very generalizable). However, according to https://github.com/zalandoresearch/fashion-mnist, the human performance on this dataset from a large study was 83.5%. Thus, this model is outperforming human performance by around 4.5%, which is favorable, and this is when the model has the bare minimum architecture/specified hyper parameters for a Convolutional neural network. Right now, there is no activation specified for the Conv2D layer, which means that this layer linearly maps its inputs to its outputs (output= wx+b). There is room to improve the ~87% testing accuracy by using the ReLU activation function in the Conv2D layer, because in these images there is clearly**

a roughly linear relationship between neighboring pixels (if neighboring pixels did not have a linear relationship, sigmoid activation may be better)- a perfect example of the linear relationship between neighboring pixels is in this image:



- **Relu is also generally useful when dealing with image recognition, because it combats the issue of vanishing gradients during back propagation (however this neural network is probably not deep/ complex enough for the vanishing gradient problem to be an issue)**

2. Adding in ReLU activation to the convolutional layer

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu"),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8857
testing accuracy: .8713

- The accuracy is higher for both the training and testing sets(so I will keep this ReLU activation)
- **As stated above, it makes sense that ReLU activation increased the accuracy and is a good fit for this CNN, because the pixels in the images have linear relationships between them (the accuracy did not increase drastically with the addition of ReLU activation because the output of ReLU activation for some inputs is the same as the output when no activation was specified, because when no activation is specified, inputs would also be linearly mapped to outputs; ReLU's shape is very similar to that when the input is greater than 0). Using another activation like sigmoid or tanh would not give higher accuracies than ReLU because of**

the linear relationship between neighboring pixels in the image, and those other activations also are more computationally expensive and slower-converging than ReLU.

- The training accuracy is still slightly higher than the testing accuracy, so it seems that the CNN is still performing better on the training data than the testing data.

3. padding the images (with "same" padding) so that the edges of the images are more influential

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", padding="same"),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8880
testing accuracy: .8759

- The accuracy increased for both training and testing sets(so I will keep the same padding)
- The training accuracy is slightly higher than the testing accuracy, so the model may just slightly be overfitting (but less than it was before)
- **By padding the images with "same padding", the output of the convolutional layer is 28x28 (which is the same dimensions as the original image), instead of being 26x26, which is what it would be without padding. This padding increased both training and testing accuracy because it ensured that the output of the convolutional layer did not shrink, and more specifically the padding ensured that no information was lost on the corners of the image (without padding the filters only go over the corner pixels once, but with padding the corner pixels are gone over multiple times). While the images in the fashion mnist dataset are centered, there is hardly any empty space around the articles of clothing in the images, so there are important features in the corners of the images, and with padding the network was able to take into account those features and thus more accurately classify the clothes.**

4. specifying the strides=2 (also see "iteratively tuning the hyperparameters" section of this document)

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", padding="same", strides=2),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])


# Compile the model
```

training accuracy (3rd epoch): .8561
testing accuracy: .8479

- The accuracy decreased for both training and testing sets(so I will not specify strides=2)
- The testing accuracy is still lower than the training accuracy, but by more than before (almost by .01)
- **With a stride size of 2, the model is prone to underfitting because the convolutional layer would be connected less densely with a larger stride (compared to a stride size of 1 which is the default value), and this explains the decreased accuracy for both the training and testing sets when the stride was set to equal 2. With a stride size of 2, the output image of the convolutional layer would be 14.5x14.5, whereas with a stride size of 1, the output of the convolutional layer would be 28x28 (almost double the size of the 14.5x14.5 output!) and the layer would be more densely connected.**

5. changing the strides to be strides=1 (also see “iteratively tuning the hyperparameters” section of this document)

```
pool_size = 2

# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", \
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8875
testing accuracy: .8840

- The accuracy increased for both training and testing sets
- The testing accuracy is still lower than the training accuracy, but by less than

before

- The training accuracy was higher before any strides were specified, but the testing accuracy is higher with strides=1; therefore I will keep the strides to be equal to 1
- **The stride size of 1 (which is also the default stride in a Keras Conv2D layer) ensured that the convolutional layer is more densely connected than it was with a stride size of 2, and the image is 28x28 with a stride size of 1, compared with being 14.5x14.5 with a stride size of 2. The denser connections and increased amount of pixels in the image which resulted in changing the stride size to 1, ensured that the model is no longer underfitting, which is why the accuracy of the training and testing sets increased by a few percentage points. It is worth noting that in some cases a smaller stride size can make the model prone to overfitting, but in this case since the original images are only 28x28 pixels (as opposed to pixel dimensions in the hundreds or thousands), decreasing the stride size does not make the model overfit.**

6. Adding another convolutional layer and another max pooling layer

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", \
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Conv2D(num_filters, filter_size, activation="relu", \
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(10, activation='softmax'),
])


# Compile the model.
```

Computations of the image dimensions:

- original image: 28x28x1
- after 3x3 filtering with same padding: 28x28x1
- after 2x2 max pooling: 14x14x1
- after 3x3 filtering with same padding: 14x14x1
- after 2x2 max pooling: 7x7x1

training accuracy (3rd epoch): .8716
testing accuracy: .8660

- The accuracy decreased for both training and testing sets
- The training accuracy is still slightly higher than the testing accuracy, so the model may still be overfitting a tiny bit
- **It makes sense that both the training and testing accuracy decreased with the addition of another convolutional and max pooling layer because the extra layers effectively reduced the image down from 28x28 pixels to 7x7 pixels, and for many of the images they would have lost some identifying features and details when they went through the second round of max pooling, which would mean that the model has less information from which to classify them. The original 28x28 images are already blurry/not detailed, so by reducing them down to 1/4 of the size, they would have lost many of the identifying features that are needed to classify them. Thus the CNN classified more images incorrectly with the addition of these extra layers.**
- Because the training and testing accuracy both decreased (by 1-2%), I will go back to using 1 convolutional layer and 1 max pooling layer in my network

7. Adding in a fully connected layer directly before the output/softmax layer, with ReLU activation

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", \
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(64, activation="relu"),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8972
testing accuracy: .8938

- **Adding in a fully connected layer between the convolution/pooling layers and the output layer helped the model learn relationships between the high-level features that the convolutional layers extracted, and by learning these relationships, the CNN was able to better classify the images (that is why the accuracy increased after this layer was added).**
- The accuracy increased greatly for both the training and testing sets, I will keep this fully connected layer
- The testing accuracy is still slightly lower than the training accuracy
- Since my accuracy increased by almost a full percent after adding this fully

connected layer, I will try adding another one

8. Adding in another fully connected layer directly before the output/softmax layer, with ReLU activation

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu", \
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Flatten(),
  Dense(64, activation="relu"),
  Dense(64, activation="relu"),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .9041
testing accuracy: .8939

- The accuracy increased a bit for the training set, but only increased by .0001 for the testing set, so it looks like adding another fully-connected layer made the CNN more prone to memorizing the training data; therefore, I will take out this extra fully connected layer
- **Adding in this fully connected layer increased training accuracy but hardly changed testing accuracy, which indicates that the extra layer was overcomplicating the model (trying to find relationships within the training images that did not really exist/were unnecessary). This is why the model became more prone to memorizing the training data, and performed better on the training data, but did not perform better on the testing data.**

9. To try and prevent the tiny bit of overfitting that there is, I will add in a dropout layer with 0.5 dropout

```
# Build the model.
model = Sequential([
  Conv2D(num_filters, filter_size, input_shape=(28, 28, 1), activation="relu",
  padding="same", strides=1),
  MaxPooling2D(pool_size=pool_size),
  Dropout(0.5),
  Flatten(),
  Dense(64, activation="relu"),
  Dense(10, activation='softmax'),
])
```

training accuracy (3rd epoch): .8737
testing accuracy: .8840

- **Adding in this dropout layer decreased training and testing accuracy, however now testing accuracy is higher than training accuracy, which indicates that the dropout layer did help combat any overfitting or memorization of the training data. As mentioned before, the output of the MaxPooling layer is 14x14x1. The original images were 28x28 pixels and they held just enough information/features from which to classify them. Thus, when Max Pooling is done to decrease complexity/reduce dimensionality of the images, the output holds even less information than the original 28x28 images. In this case, adding in the Dropout means that a few of the important relationships in the 14x14 output of the Max Pooling layer, are no longer considered, and this is why the accuracy decreased by about 2%.**
- Since the model's training accuracy was just barely higher than its testing accuracy before the dropout layer was added in, and since adding in the dropout layer decreased both the training and testing accuracies, I am leaving out this dropout layer
- I also tried Dropout(0.25), and got a similar result as the Dropout(0.5), with decreased accuracies
- I also tried adding in a dropout layer between the last 2 layers, and my accuracies decreased then as well

10. Decreasing number of epochs from 3 to 1 (also see "iteratively tuning the hyperparameters" section of this document)

```
# Train the model.
model.fit(
  train_images,
  to_categorical(train_labels),
  epochs=1,
#   validation_data=(test_images, to_categorical(test_labels)),
```

training accuracy- .8370
testing accuracy- .8685

- **By reducing the number of epochs, the accuracy dropped 2-3% on average. The images in the mnist fashion dataset are simple, but not simple enough that the model can learn all the relationships in the images in 1 epoch. Because of this, the model was underfitting and not capturing enough relationships/information from the images to be able to classify many of them correctly; this explains the decrease in accuracy that is associated with the decrease in number of epochs.**
- Decreasing the number of epochs made the testing accuracy greater than the training accuracy, which indicates that the model did not overfit in this case, however, decreasing the number of epochs dramatically decreased both the training and testing accuracy, so I will change the epochs back to 3

11. Increasing the number of epochs from 3 to 5 to 20 (also see “iteratively tuning the hyperparameters” section of this document)

```
# Train the model.
model.fit(
  train_images,
  to_categorical(train_labels),
  epochs=5,
#   validation_data=(test_images, to_categorical(test_labels)),
```

training accuracy- .9177
testing accuracy- .9000

- Increasing the number of epochs increased the training accuracy by about a percentage point, and it increased the testing accuracy by about 1/3 to 1/2 a percentage point
- Increasing the number of epochs increased the gap between the training and testing accuracies
- However, since the testing accuracy did go up with the increased number of epochs, I will try increasing the number of epochs a little more to try to increase the testing accuracy even more

When I increased the number of epochs to 20, the training accuracy went up to .9727 and the testing accuracy went up to .7063. The model at this point was memorizing the training set(and taking a very long time to run) and its testing accuracy was hardly changing. Therefore, I decided to keep the number of epochs at 5 because it was the best combination of increasing the testing accuracy (because the model was iterating over the training dataset enough times to learn all the relevant information/relationships in it) and being time efficient.

12. Increasing the number of filters from 8 to 12 (also see "iteratively tuning the hyperparameters" section of this document)

```
num_filters = 12
filter_size = 3
pool_size = 2
```

training accuracy- .9251
testing accuracy- .9026

training accuracy- .9220
testing accuracy- .8998

- **Increasing the amount of filters increases the number of characteristics learned by the model. I originally set the number of filters at 8 because the testing accuracy increased as I added on more and more filters from the starting point of 1 filter, but as I got to around 8 filters the increase in accuracy seemed mostly negligible with the addition of each new filter. The images in this dataset are quite non-complex and the vast majority of them can be correctly classified by looking at the outlines of the clothes (the edges) and the shape of the article of clothing. There are not really any small, important details that need to be detected in order to make the model more accurate. Thus, adding 4 extra filters makes the model identify and memorize insignificant details of the images in the training set, which accounts for the increase in training accuracy. However, these extra filters don't have a significant impact on the testing accuracy; these extra filters just unnecessarily complicate the model. I think that if I increased the number of filters dramatically from 12 filters, the training accuracy would increase a lot, but the testing accuracy would actually decrease because the model would be greatly prone to overfitting.**
- I ran this code twice, and got testing accuracies both higher and lower than before I increased the number of filters. The training accuracies were consistently higher (by 0.5 to 1%) than before I increased the number of filters.
- I decided to test out 10 filters (which is between 8 and 12) to see if that

number would be more time efficient and give more consistent results than running the model with 12 filters.

13. Changing the number of filters to 10 (also see “iteratively tuning the hyperparameters” section of this document)

```
num_filters = 10
filter_size = 3
pool_size = 2
```

training accuracy- .9219
testing accuracy- .9075

training accuracy- .9233
testing accuracy- .9054

- I ran this code twice, and my testing accuracy was consistently higher than it was using 8 or 12 filters, so I am keeping 10 filters
- **Earlier, I had identified 8 filters as a good amount for this CNN, and I found that 12 filters made it overfit. When I tested out 10 filters, the CNN consistently had higher accuracy than using 8 filters, so I determined that using 10 filters captured just the right amount of details/ characteristics of the images, without overfitting to the training data.**

14. Changing the filter size from 3x3 to 2x2 (also see “iteratively tuning the hyperparameters” section of this document)

```
num_filters = 10
filter_size = 2
pool_size = 2
```

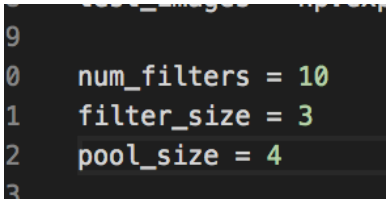
training accuracy- .9161
testing accuracy- .8940

training accuracy- .9140
testing accuracy- .9005

- I ran this code 2 times, with similar results both times
- Both the training and testing accuracies decreased by about .005 or more, so I decided to keep the filter size as 3x3
- **I think that decreasing the filter size decreased accuracy (although only by a small amount), because in order to correctly classify the fashion mnist images, it is more important for the model to identify the larger,**

overall shapes than small, local features. Thus a larger filter size is the better choice for this model, and the accuracies reflect that. Also, decreasing the filter size means connecting the convolutional layer less densely, so less relationships are learned by the model.

15. Changing the pool size from 2x2 to 4x4 (also see "iteratively tuning the hyperparameters" section of this document)



training accuracy- .8968
testing accuracy- .8936

- Both the training and testing accuracies decreased by about .01, so I decided to keep the pool size 2x2
- **Increasing the pool size most likely decreased the accuracy because it condensed down some of the images too much, and important features may have been lost, leading the model to misclassify some images. This is because the images in the dataset were small and not very detailed to begin with, so a 4x4 pooling would decrease the image dimensionality too much for it to be useful.**
- I also tried running the model with a pool size of 1 (which is effectively not doing any pooling at all), and not only did it take longer to run, but the

Description of my model after all these adjustments:
number of filters: 10
filter size: 3x3
pool size: 2x2
strides=1
padding="same"
convolutional layer activation: relu
epochs: 5
number of Conv2d and MaxPooling2d layers: 1 each
extra fully connected layer added in before softmax output layer

My final testing accuracy was .9075 (it varied slightly each time I ran the code), compared to my initial testing accuracy of .8706. Most of the adjustments I made increased the testing accuracy by very small amounts, but all the adjustments

added together to overall bring my accuracy up about 3.5%.

While making most of my adjustments, I noticed the testing and training accuracies went up together. All throughout this process, my training accuracy was about 1% higher than my testing accuracy. The model performed better on the training set than the testing set, however since the difference was so small, I don't think overfitting is a problem in this case.

ITERATIVELY TUNING THE HYPERPARAMETERS

Although I hand-tuned some of the hyperparameters, for the numerical hyper parameters I also used a nested for-loop (5 for-loops nested within each other) to tune them

- using a nested for-loop, I adjusted the hyperparameters for:
  - the number of filters (I tried up to 14 filters)
  - the size of the filters (I tried up to a 6x6 filter)
  - the pool size(I tried up to a 6x6 pool size)
  - the number of strides(I tried up to 4 strides)
  - the number of epochs (I tried up to 12 epochs)

Attached is the document in which I printed the results of all my iterations (as I was running different models I intermittently saved my results to a new file, so the pdf is just all those json files put together). The format of the document is a list, with each iteration of running the model having its own dictionary, and the "key" refers to a list which is [number of filters, size of filters, pool size, number of strides, number of epochs], the "value" refers to a list which is [loss of this model, accuracy of this model].

My final hyperparameters that maximized accuracy are:
number of filters: 10
size of filters: 5
pool size: 2
stride size: 1
number of epochs: 7

The final maximized accuracy was: 0.9077000021934509

COMPARISON OF ITERATIVELY AND HAND TUNING THE HYPERPARAMETERS

Interestingly, I got the accuracy-maximizing number of filters (10), pool size (2x2), and stride size(1x1) when hand-tuning the hyperparameters. However, in iteratively

tuning the hyperparamters the size of the filters that maximized accuracy was 5x5 (as compared to the 3x3 I got when hand tuning the hyper parameters) and the number of epochs that maximized accuracy was 7 (as compared to the 5 epochs I got when hand tuning the hyperparameters).

Addressing the disparity in results and analyzing why the iteratively tuned hyper parameters work better than the hand tuned ones:

- When hand tuning the hyperparameters, I did not try increasing the filter size, I only decreased it. It makes sense that the optimal size of the filters is 5x5, not 3x3 because perhaps the 3x3 filters were trying to detect important details from too-small sections of the images (which is not surprising because the original images were not detailed at all, they were more like vague shapes of the articles of clothing). Thus, using a 5x5 filter was the large enough to detect important features from the images (edges, etc.), but the 5x5 filters were not too big that it was preventing the convolutional layer from extracting important edges and shapes from the images. Also, using a larger filter makes the convolutional layer more densely connected to it's neighboring layer in the network, and perhaps those extra connections helped the model learn important relationships that increased the CNN's accuracy.

- When hand tuning the hyperparameters, I did not try running a model with 7 epochs (I only tried 1, 3, 5, and 20 epochs). The CNN's accuracy was maximized with 7 epochs (rather than any number above or below 7 epochs) because with any less than 7 epochs, the CNN was not learning enough relationships/information from the training data, and thus the accuracy was not maximized. With any more than 7 epochs, the CNN was learning too many relationships from the training data (overfitting to the training data) and thus its performance on the testing data was lower than ideal because the extra relationships found in the training images may not have been present in the testing images.