Assignment 4: Neural Networks and Deep Learning - Machine Learning 6316

Jacob Dineen / JD5ED

November 5, 2019

1 Neural Network Playground

1.1 Hand-Crafted Feature Engineering

Below are the results of the experiments that I ran on Tensorflow Playground (or this variation of it). Please find my best solution screenshot for each dataset, along with a table specifying the hyperparameters use to garner each score. Some things to note: On the XOR task, we didn't have to use any kind of manual feature engineering in terms of feature expansion because we were just learning a linear decision boundary, which logistic regression is quite good at. In this case, I settled on just using the normal two input features. I didn't have to employ any regularization techniques for any dataset, as the time to convergence was relatively quick, On the Circle dataset, I used the two features labeled X12 and X22. I believe that these are really supposed to be superscripted so as to say that we're squaring each of the features. If we square the input space, intuitively we're allowing for more seperation for a decision boundary to be drawn in a higher dimensional space. When we map that linear n-dimensional decision boundary to a 2-D representation of the feature space, we get the circle shown above. For the case of the Gaussian data, we can get away with a single handcrafted feature, multiplying X1 and X2 together and actually shrinking down the input space and the number of connections between it and the first hidden layer.Intuitively, I imagine that this because we expect there to be some connection between the mapping of this transformation to the binary target label. E.g., the amount of information we get from this transformation is greater than when considering these features independently. For the spiral task, it was clear that we'd have to learn an incredibly complex decision boundary, so this is when I employed the power of 'deeper' networks, both in regard to node depth and hidden layers. The more of each of these we add to our network, the more complex functions we can learn to represent (Universal Approximation Theorem states something along the lines of, I believe, that an MLP with sufficient depth can learn to represent any function. I don't think this necessarily means that we are guaranteed to learn the function that correctly maps input to output, but it's interesting nonetheless). For the majority of this task, I went with the following architecture: 2 Hidden layers of 5 neurons a piece, No Regularization, sigmoid activation, batch size = 10, train/test split = 0.7 and different variations of feature engineering per the required task, which really just gleaned on insight into the likely feature space that would best model some specific task. See Figure 1 (may fall on a separate page depending on overleaf formatting).

Table 1: Table Showing Best Model for each dataset

Task	Train Test Split	Batch Size	Activation	Learning Rate	Regularization	Regularization Rate	Features	Hidden Layers	Neurons H1	Neurons H2	Epochs	Train Loss	Test Loss
Circle	0.7	10	Sigmoid	0.1	None	0	X12 & X22	2	5	5	306	0	0
Gaussian	0.7	10	Sigmoid	0.1	None	0	X1 & X2	2	5	5	70	0	0
XOR	0.7	10	Sigmoid	0.1	None	0	X1*X2	2	5	5	390	0	0.001
Spiral	0.7	10	Sigmoid	0.1	None	0	X1 & X2, Sin(X1) & Sin(X2)	3	5	5	700	0.017	0.051

Table 2: Experiments with Regularization

Task	Train Test Split	Batch Size	Activation	Learning Rate	Regularization	Regularization Rate	Features	Hidden Layers	Neurons H1	Neurons H2	Epochs	Train Loss	Test Loss	Features
XOR	0.7	10	Sigmoid	0.1	L1	0.01	All	2	5	5	250	0.011	0.01	X1 & X2
Circle	0.7	10	Sigmoid	0.1	L1	0.01	All	2	5	5	250	0.014	0.013	$X1^2$ & $X2^2$
Gaussian	0.7	10	Sigmoid	0.1	L1	0.01	All	2	5	5	250	0.013	0.019	X1*X2
Spiral	0.7	10	Sigmoid	0.1	L1	0.01	All	3	5	5	1000	0.324	0.404	Sin(X1) and Sin(X2)
XOR	0.7	10	Sigmoid	0.1	L2	0.01	All	2	5	5	250	0.016	0.015	X1 & X2, Sin(X1) and Sin(X2)
Circle	0.7	10	Sigmoid	0.1	L2	0.01	All	2	5	5	250	0.016	0.016	$X1^2$ & $X2^2$
Gaussian	0.7	10	Sigmoid	0.1	L2	0.01	All	2	5	5	250	0.027	0.039	X1*X2
Spiral	0.7	10	Sigmoid	0.1	L2	0.01	All	3	5	5	1000	0.02	0.02	Amalgamation

1.2 Regularization

Next, we look at the impact of regularization on the datasets, separately. We'll employ essentially the same architecture but see how the activations saturate throughout the network, and which features end up driving the learned decision boundary. The results from my experiments are noted below. I ran each architecture through L1 and L2 regularization with the reg. rate fixed to 0.01 throughout. I typically let each one for 250 epochs to see if I could close in on the test loss noted above - While the difference is negligible, the models shown in table 1 performed slightly better across the board and required less train time to do so. What was found was: The logic from above seems to hold. In the case of XOR, the regularization added by L1 or L2 squashed all feature importance side from the standard input of X1 and X2. Interstingly, L2 reg. allowed for Sin transformations to play a role in the decision boundary to some extent. Sin itself could be seen as a squashing function as the range is limited, so perhaps that's the reason why. For Circle, We just used the expansion formed by squaring the inputs - This was verified by both types of regularization. For Gaussian, again this was correctly noted above. Spiral was interesting, because in both cases the network thought unintuitive features were important, e.g. for L1 reg. the network was highly skewed by the Sin transformation, and for L2 the network decided that all features were almost equally important.See Figure 2 for a single example from each dataset(may fall on a separate page depending on overleaf formatting).

1.3 Automated Feature Engineering with Neural Network

Finally, we are to try to fit a model using only the standard features and allow the network to perform automatic feature engineering as we increase the complexity of the mode. I've noted my results in the table below (Some of them). It was generally pretty easy to see a quick time to convergence and receive a test loss of 0.001 or less. Some things I noticed: Increasing the network in size, either scaling down or out, helped the test loss to converge to a minima quicker. Wider and deeper networks all reached 0.001 or less loss much quicker than with regularization or hand crafted feature engineering. Relu performed much better than sigmoid as a general case. Sigmoid couldn't draw the decision bondary for the XOR task until I added more neurons and layers to the network. Relu always converged quicker tahn sigmoid, and tanh sometimes performed quicker than sigmoid, but never quicker than Relu. I think Tanh and Sigmoid both suffer from some form of vanishing and exploding

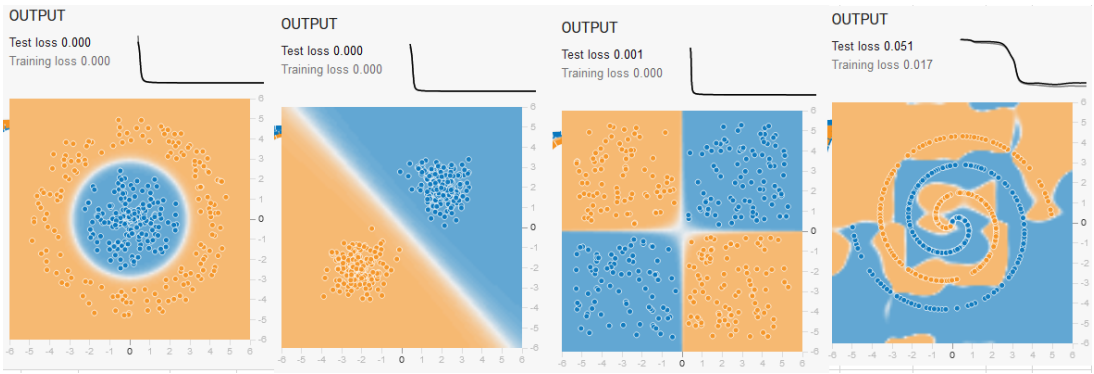


Figure 1: Results From Playground simulations for all four datasets using Sigmoid activation

Table 3: X1 and X2 Only - Auto Feature Engineering

Model #	Task	Train Test Split	Batch Size	Activation	Learning Rate	Regularization	Regularization Rate	Features	Hidden Layers	Neurons	Epochs	Train Loss	Test Loss
1	XOR	0.7	10	Relu	0.1	None	None	X1 and X2	4	5	40	0	0
2	XOR	0.7	10	Sigmoid	0.1	None	None	X1 and X2	2	5	300	0.5	0.5
1	XOR	0.7	10	Tanh	0.1	None	None	X1 and X2	2	5	64	0	0
2	XOR	0.7	10	Sigmoid	0.1	None	None	X1 and X2	2	10	28	0	0
1	Circle	0.7	10	Sigmoid	0.1	None	None	X1 and X2	2	5	760	0.001	0.001
2	Circle	0.7	10	Relu	0.1	None	None	X1 and X2	2	10	177	0	0.001
3	Circle	0.7	10	Relu	0.1	None	None	X1 and X2	3	10	177	0.001	0.001
4	Circle	0.7	10	Relu	0.1	None	None	X1 and X2	4	5	38	0	0

gradients to a larger extent than Relu, which is why Relu and all of the other variants (like leakyRelu) have kind of settled in as the standard. See figure 3 for a couple of examples of this convergence behavior given an arbitrarily larger network. These weren't reported on in the table, but paint a similar picture that as long as the data isn't too noisy (which it is in the real world) we can learn some function to represent these simple tasks. *Note: I didn't want to include too many images that didn't really add to the discussion. I think the three figures below show the general trend of network elasticity in terms of these parameters.

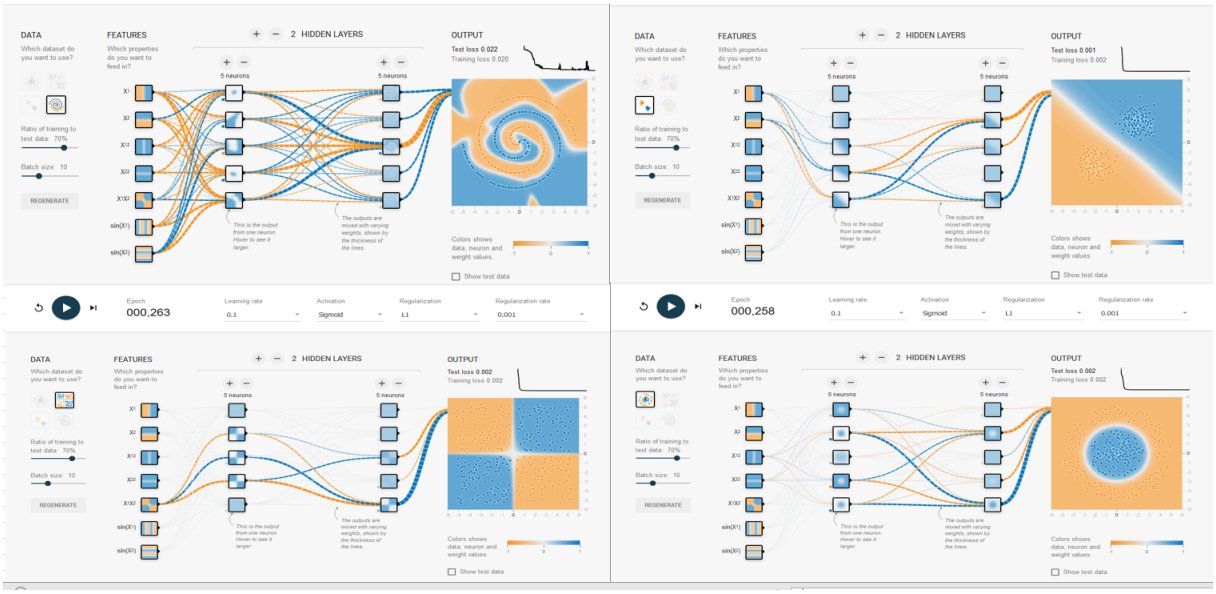


Figure 2: Regularization Results

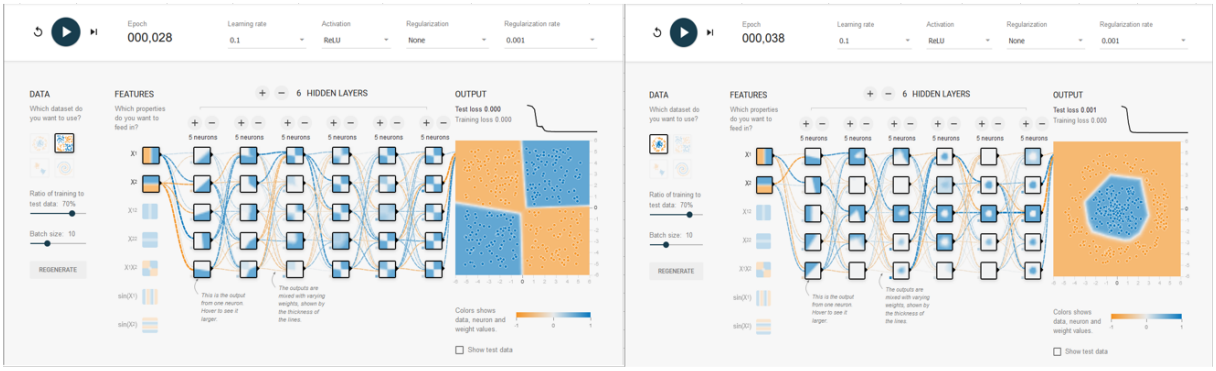


Figure 3: Deeper/Wider Network Results

1.4 Spiral Challenge (0.5 Extra credit)

I actually submitted this above, but please find an isolated attachment using a simpler model. Here I use only the original two features and 2 transformed feautures using sin. I also use sigmoid activation. The architecture has 2 hidden layers and 5 neurons in each, so it's a much simpler model than the ones constructing toward the end of this section.

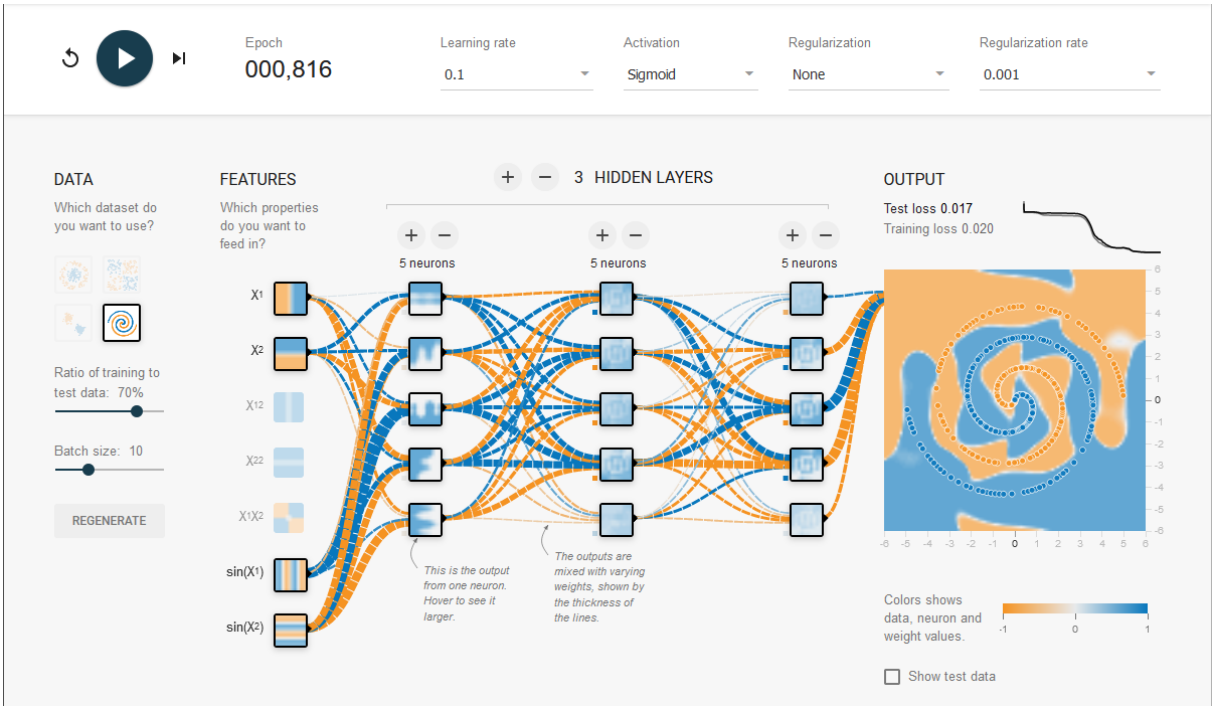


Figure 4: Solid Performing Net on Spiral

Table 4: Solid Performing Net on Spiral

Task	Train Test Split	Batch Size	Activation	Learning Rate	Regularization	Regularization Rate	Features	Hidden Layers	Neurons H2	Epochs	Train Loss	Test Loss
Spiral	0.7	10	Sigmoid	0.1	None	0	X1 & X2, Sin(X1) & Sin(X2)	3	5	700	0.017	0.02

2 Deep Learning with Keras

Now we move onto the coding portion of the assignment. The dataset we're working with is the Fashion MNIST datset, which contains 60k labeled pairs of photos and class companions in the train set. We're provided the testset without labels for submission in a similar Kaggle style competition. Is photo is a 28x28 image with a single color channel. We can think of this as each feature having 784 pixels representing the pixel density of each location in the image. I haven't worked on this dataset before, but it seems fairly similar to MNIST as far as categorical classification goes. The professor provided us with the data normalization/ read in code already, so we only really need to worry about finding an optimal set of parameters that allows for generalization. We are allowed to use Sklearn for model selection, so I'll likely utilize their grid searching and cross validation code, as it's a bit more clean than mine.

I have a couple of Nvidia GPUs in my current PC, and am atleast running a workable version of Tensorflow-GPU (Likely a very old version, as I haven't updated in a while for the specific reasons noted in the post, although

Conda install allows for a relatively painfree experience).

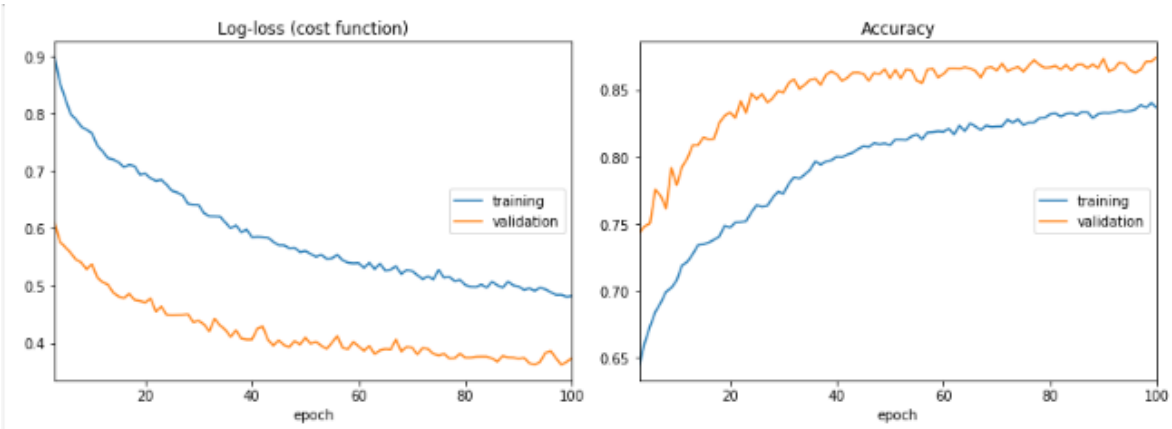
2.1 Multilayer Perceptron

After training a few models via keras, I've realized that intense hyperparameter tuning / cross validation might be prohibitive because these models do take a while to train. I imagine convnets will train quicker, as they are able to utilize (atleast, better I think) cuda cores. After some extensive tuning, I was able to create a model that scored 87.4 on the 10 percent validation set heldout from training.

Layer (type)	Output Shape	Param #
dense_65 (Dense)	(None, 256)	200960
dropout_30 (Dropout)	(None, 256)	0
dense_66 (Dense)	(None, 128)	32896
dropout_31 (Dropout)	(None, 128)	0
dense_67 (Dense)	(None, 50)	6450
batch_normalization_50 (Batc	(None, 50)	200
dense_68 (Dense)	(None, 50)	2550
dense_69 (Dense)	(None, 25)	1275
dropout_32 (Dropout)	(None, 25)	0
dense_70 (Dense)	(None, 10)	260
activation_32 (Activation)	(None, 10)	0

Total params: 244,591
Trainable params: 244,491
Non-trainable params: 100

Figure 5: MLP Architecture




```
Log-loss (cost function):
training   (min:    0.481, max:    1.680, cur:    0.482)
validation (min:    0.362, max:    0.826, cur:    0.374)

Accuracy:
training   (min:    0.345, max:    0.840, cur:    0.837)
validation (min:    0.681, max:    0.874, cur:    0.874)
```

Figure 6: MLP Metrics

The standard architecture that I went with is shown in figure 5. In summary, there's 6 dense layers, one instance

of batchnorm (reducing the covariance shift), and 3 instances of dropout (turning off neurons with probability to prevent overfitting). I've also implemented some callbacks via Keras for early stopping and model checkpointing. Early stopping will shut off the model if there's a continuous period of time where there's no improvement in validation accuracy (defined by patience). Checkpoint will ensure that our final model is the model taken at the epoch where the validation accuracy was the highest, e.g. if our model performs well in epoch 10, but suffers in epoch 100, we will take the learned parameters from epoch 10's fit on the data. This has the added advantage of saving the model directly to disk, so we can simply call the load model method via Keras to avoid retraining for submission, which should substantially reduce runtime on your end (I'll submit the h5 files for both the MLP and convnet in Collab). Softmax is used at the output layer to give us a probability distribution over the 10 classes (categorical MLE, for the probabilistic interpretation). Categorical cross entropy is used as the less. I used relu activations throughout simply because I've found that variants of relu perform the best in practice, coupled with an Adam Optimizer, which I've also found performs slightly better than SGD. So the optimization techniques involves adam (This was changed to SGD with Nesterov Momentum), which is an adaptive optimizer, and we use mini batch learning = 256 to perform len(dataset)/256 weight updates during a single epoch. I set this up to run 100 epochs under the above parameters and plateued around the 50th iteration - The best model was stored with a validation accuracy of about 96.7 percent. Not bad. See figure 6 below. Had we have taken the final model, we would have been overfitting to an extent on the train data. Because we took the model around the 30th epoch, we're likely ok. Early stopping, from the papers I've read, is a form of regularization in this regard.

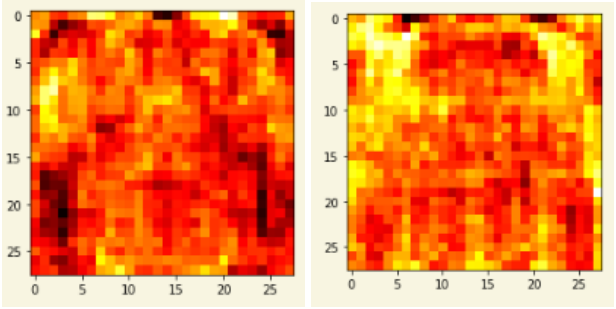


Figure 7: Visualizing Weights

In Figure 7, we try to visualize what exactly the network is learning as we feed from the input layer to the first hidden later. I've commonly heard that, in vision tasks, the lower level layers are more responsible for generating such features as shapes and edges, so if we look at the above, we can maybe understanding that we're trying to draw some boundary around the region of interest. As this propagates further through the network, we'll have a more clear understanding of which regions in the network are proving important for our predictions. There's some cool work in deep learning explainability that explores such things as occlusion and marginal contributions that also help to show which features are activated toward the tail end of a model.

I anticipate that a basic CNN will outperform this MLP. I'll submit both h5 files but point only to the best performing (valacc) model when submitting.

2.2 Convolutional Neural Network

I take much of the same skeleton code from above to create a convnet. The only real difference occurs in the layers toward the front where we define convolutional layers - In essence, we're having a filter/kernel stride across the feature space while computing dot products with the learnable weights. This works well because there's quite a bit of autocorrelation present in images. We then feed the output of the conv layer to a pooling layer to reduce dimensionality (I use max pooling here), and finally feed this feature space through, essentially, an MLP (dense layers). I note my (relatively unnecessarily large) network below. I've aggregated a number of conv layers, batch norm layers, max pooling layers, activation layers, and fully connected layers. The model will still be trained using categorical cross entropy with SGD optimizer, and will still employ the same validation split as the MLP.

Early stopping and checkpointing will be used as well. The number of parameters is exceedingly large, so hopefully my GPU configuration is correct to send these operations through for parallelization (it's been a while since I've used Keras). I set this with max epochs = 100. The data shape that we feed into the first layer of the network is a bit different here. I believe because we don't have 3 color channels, the input looks a bit different than what I'm accustomed to seeing. I also used a batch size = 256 during training to speed up runtime.

We end up with a final accuracy on the 10 percent validation set of 93.6 percent, which is a vast improvement from the best MLP model. We also have quite a few more learned parameters, so it's difficult to assess the tradeoff between accuracy and disc space/ real time inference. Figure 9 shows the loss and accuracy plots for this CNN throughout training. What's a bit interesting here is that we were underfitting our train set for almost the entire duration of training. I think this has to do with too much regularization in the form of dropout pushed into the network. Nevertheless, the generalization ability proved better than the MLP and I'll likely submit this as the best final model. In the py file that I'm submitting, the structure of main looks like - I'm submitting with the grading mode flag on:
Fetch system arguments (file locations):
Get data for train and test sets.
reshape train set. Call Keras Load Model to get the best performing convnet (located in directory).
Run predict classes on the test set.
Run output predictions to store the categorical response in a text file for submission
If grade mode is off, we'll run through and train the MLP and CNN from scratch, which will likely take over 30 minutes.

To answer the remaining questions from the homework template: I run model.layers[idx].output and get the shape to see what kind of transformations are happening to my data at each layer. After the first convolutional layer, the data changes from being 60000 28x28x1 numpy arrays to a tensor of shape: 28x28x200. The size of the conv filter applied in the first layer is 200, so this makes sense. After the data goes through a max pooling layer (2,2), the shape changes to 28x28x100 to 14x14x100 (2 conv layers before the first max pooling layer). This is accounting for autocorrelation and taking only the most dense pixel from a region defined by the stride. See fig 8.

A few additional things that could have been done that would have likely eeked out a slight performance increase, based on what I've seen in the past: Data augmentation - Keras has a pretty user friendly image generator class that allows for a pipeline to be created in which data is rotated, shifted, brightened, etc.. Basically, we expand the size of the trainset and hope that makes our model more generalizing to unseen data because we are slightly modifying the images to an extent that almost makes them new training data altogether. Nesterov Momentum / Adaptive Learning Rates. Because the performance was already really good right out of the box, I didn't play around with this. Adaptive learning rates (cos annealing) follow some function to change throughout training. Perhaps we start off with an initial learning rate of .1 but as training progresses through each epoch we kick this up to try to avoid some of the local minimas that occur in highly non-convex loss surfaces. I believe you can use the K class from keras to wrap up a Keras classifier in scikit learn functions like grid search. Because the training took quite a long time as is, I opted to avoid extensive parameter searching, instead just relying on the size of the network to automatically generate features and find some decision boundary in a high-dimensional space. Hyperas is a cool python module for param searching, however. The thing with these nets, however, is there's so many hyperparameters to tune that the search space is incredibly large. I believe there's a whole line of research on this - e.g., bayesian methods for search.

*Note I went back and retrained this with less dropout, learning rate decay, and a few additional tweaks (LR was set to 0.01 for all experiments above. Here it is set to 0.01 with decay). I increased the batch size to utilize GPU resources more, and increased the max epochs. Results of this run can be seen in figure 10. Final validation accuracy was around 95 percent, while final train accuracy was closer to 99 percent (.1 val data taken out of this round, .2 taken in previous rounds). I made a mistake that I luckily caught before submission when using the get data method. I had to turn shuffling off prior to inference in order to preserve the order. The output predictions file generates a fairly uniform distribution of class responses, so I assume that my model generalized well to unseen data.

Layer (type)	Output Shape	Param #
conv2d_21 (Conv2D)	(None, 28, 28, 200)	2000
activation_59 (Activation)	(None, 28, 28, 200)	0
conv2d_22 (Conv2D)	(None, 28, 28, 100)	180100
activation_60 (Activation)	(None, 28, 28, 100)	0
max_pooling2d_15 (MaxPooling	(None, 14, 14, 100)	0
dropout_45 (Dropout)	(None, 14, 14, 100)	0
conv2d_23 (Conv2D)	(None, 14, 14, 50)	45050
activation_61 (Activation)	(None, 14, 14, 50)	0
max_pooling2d_16 (MaxPooling	(None, 7, 7, 50)	0
conv2d_24 (Conv2D)	(None, 7, 7, 50)	22550
activation_62 (Activation)	(None, 7, 7, 50)	0
max_pooling2d_17 (MaxPooling	(None, 3, 3, 50)	0
flatten_7 (Flatten)	(None, 450)	0
dense_86 (Dense)	(None, 256)	115456
batch_normalization_67 (Batc	(None, 256)	1024
activation_63 (Activation)	(None, 256)	0
dense_87 (Dense)	(None, 50)	12850
activation_64 (Activation)	(None, 50)	0
dropout_46 (Dropout)	(None, 50)	0
dense_88 (Dense)	(None, 25)	1275
activation_65 (Activation)	(None, 25)	0
dropout_47 (Dropout)	(None, 25)	0
dense_89 (Dense)	(None, 10)	260
activation_66 (Activation)	(None, 10)	0
dropout_48 (Dropout)	(None, 10)	0
dense_90 (Dense)	(None, 10)	110
activation_67 (Activation)	(None, 10)	0

Total params: 380,675
Trainable params: 380,163
Non-trainable params: 512

Figure 8: CNN Architecture

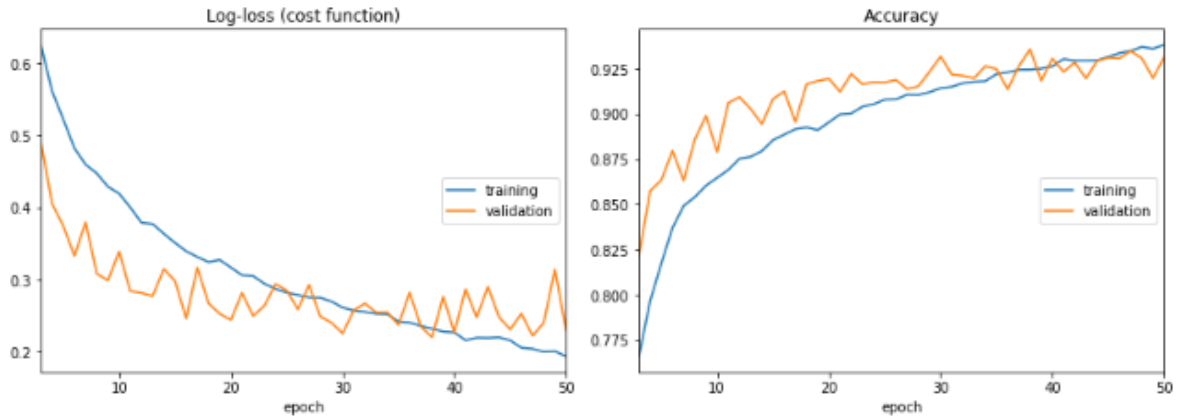
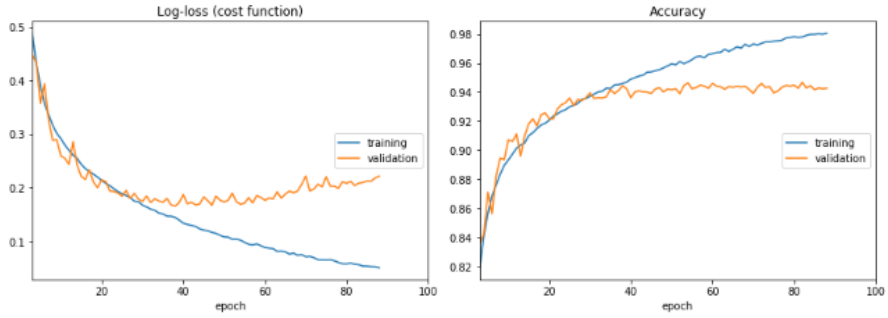


Figure 9: CNN results




```
Log-loss (cost function):
training   (min:    0.050, max:    0.992, cur:    0.050)
validation (min:    0.166, max:    0.649, cur:    0.221)

Accuracy:
training   (min:    0.633, max:    0.981, cur:    0.981)
validation (min:    0.755, max:    0.947, cur:    0.943)
Epoch 00088: early stopping
```

Figure 10: CNN results - v2

Layer (type)	Output Shape	Param #
conv2d_84 (Conv2D)	(None, 28, 28, 32)	320
batch_normalization_82 (Batc	(None, 28, 28, 32)	128
conv2d_85 (Conv2D)	(None, 26, 26, 64)	18496
batch_normalization_83 (Batc	(None, 26, 26, 64)	256
max_pooling2d_66 (MaxPooling	(None, 13, 13, 64)	0
dropout_95 (Dropout)	(None, 13, 13, 64)	0
conv2d_86 (Conv2D)	(None, 13, 13, 128)	73856
batch_normalization_84 (Batc	(None, 13, 13, 128)	512
conv2d_87 (Conv2D)	(None, 11, 11, 128)	147584
batch_normalization_85 (Batc	(None, 11, 11, 128)	512
max_pooling2d_67 (MaxPooling	(None, 5, 5, 128)	0
dropout_96 (Dropout)	(None, 5, 5, 128)	0
conv2d_88 (Conv2D)	(None, 5, 5, 256)	295168
batch_normalization_86 (Batc	(None, 5, 5, 256)	1024
conv2d_89 (Conv2D)	(None, 3, 3, 256)	590080
batch_normalization_87 (Batc	(None, 3, 3, 256)	1024
max_pooling2d_68 (MaxPooling	(None, 1, 1, 256)	0
flatten_16 (Flatten)	(None, 256)	0
dense_117 (Dense)	(None, 1024)	263168
dropout_97 (Dropout)	(None, 1024)	0
dense_118 (Dense)	(None, 512)	524800
dense_119 (Dense)	(None, 10)	5130

Total params: 1,922,058
Trainable params: 1,920,330
Non-trainable params: 1,728

Figure 11: CNN results - v2

2.3 Extra Credit: PCA and Logistic Regression

This wasn't required, but I thought it would be interesting as I head into the PCA section. I ran the test data through a PCA + Kmeans clustering pipeline. When reducing the dimensionality to 2 PCAs, we end with a cluster plot of the ten classes that looks like the following figure. There appears to be significant overlap between class boundaries -I'm thinking that the loss of variance is too great at this low of a dimension.

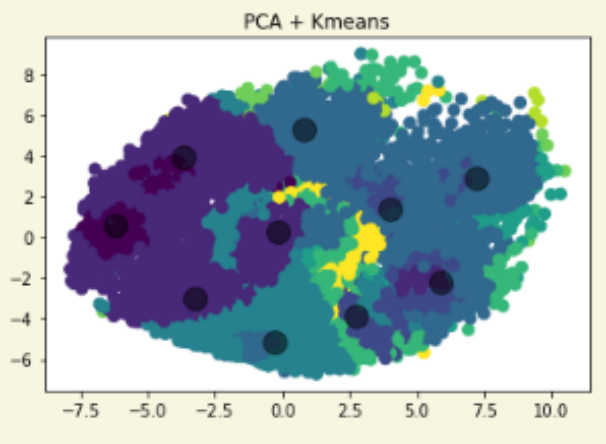


Figure 12: PCA (2) + K means results

For this extra credit portion, we have to construct various logistic regression classifiers after fit transforming our train set per the number of PCs we desire. PCA is a dimensionality reduction algorithm. If we can remove superfluous feature from our feature space, or simply dampen them, we can reduce retain the same variance that allows for us to draw a decision boundary in a high dimensionality space while also experiencing vast speedups in runtime (Train). The design for this was relatively straightforward. The piepline is shown below. First we fit and transform the train set per the specified number of PCs, then we run each new train set through a cross validation schema - I use 5 fold CV here.

```
train_file = 'fashion_train.csv'
test_file = 'fashion_test.csv'
x_train, y_train = get_data(train_file)
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate

num_comp = [2,5,10,25,50,100]
score_1 = {}
for i in num_comp:
    train = PCA(n_components=i).fit_transform(x_train)
    model = LogisticRegression()
    score_1[i] = cross_validate(model, train, y_train,
                      cv=5)
```

Listing 1: PCA + Logit

The results seen in Figure 12 show that as we increase the number of PCs, our mean CV accuracy on the holdout fold for each iteration of train data increases, however there is an inflection point of sorts, where there isn't much added benefit between the cost (in seconds) to train the model and the perceived value in terms of increased accuracy. I've included this code in my final submission file, but turned a flag off so it won't run. All noticeable results are included here anyways.

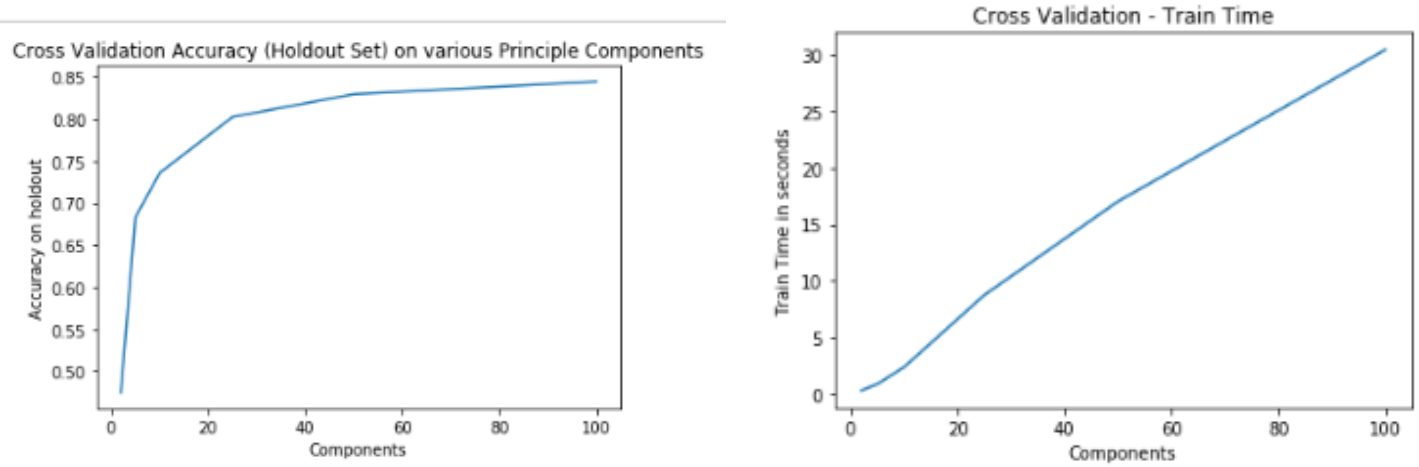


Figure 13: PCA + Logit results

For comparison, a simple logistic regression model without PCA (right out of the scikit box) runs with CV = 5 mean holdout accuracy of right around 85 percent, but takes around 90 seconds to fit each fold. So even if we shrink the feature space down from 784 to 100 we can produce very similar accuracy at a third of the computational cost to train.

```
{'fit_time': array([90.03111982, 91.35583448, 90.8130126 , 89.75264072, 90.3546879
3]),
 'score_time': array([0.04687572, 0.04884529, 0.04986691, 0.04787207, 0.04485536]),
 'test_score': array([0.85170374, 0.85584535, 0.84466667, 0.8551546 , 0.85071268])}
```

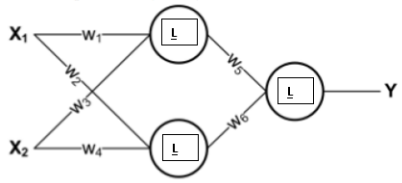
Figure 14: Logit results

3 Sample Questions:

Question 1

Question one asks us to express each unit of the given graph in terms of linear or nonlinear interactions for neural networks that simulate linear regression and binary logistic regression.

(a) Assign proper activation functions (**S** or **L**) to each unit in the following graph so this neural network simulates a linear regression: $Y = \beta_1 X_1 + \beta_2 X_2$.

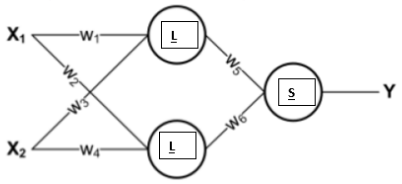


In the linear regression case, we simply have three series of linear combinations, which we could vectorize according to learned weight matrices/vectors. Because we just have a single hidden layer here, the notation z_i is just referring to the i^{th} neuron of the hidden layer.
$z_1 = X_1 w_1 + X_2 w_3$ (L)
$z_2 = X_1 w_2 + X_2 w_4$ (L)
$\hat{y} = z_1 w_5 + z_2 w_6$ (L)

In the second case, where we are building a binary logistic classifier via our neural net configuration, we are running through the same series of linear combinations to produce the input to our $\hat{y}$, but will be introducing non-linearity via the sigmoid function, which maps the value of a after the linear combinations to a bounded interval between 0 and 1. The way I have studied neural networks in the past doesn't necessarily break the linear and nonlinear interactions down into node-wise operations, but instead assumes that the input to each neuron is the linear combination of the weights and previous input values, while the ouput is the nonlinear function applied on top of that, whether it be ReLu/Tanh/Sigmoid, etc.. I don't believe we'd apply the sign function unless we were trying to generate accuracy, as the result of the nonlinearity would still have to be differentiable in order for us to perform backprop/weight updates. See fig. 15 below.

$z_1 = X_1 w_1 + X_2 w_3$ (L)
$z_2 = X_1 w_2 + X_2 w_4$ (L)
$z_{3in} = z_1 w_5 + z_2 w_6$
$\hat{y} = \sigma(z_{3in})$ (S)

(b) Assign proper activation functions (**S** or **L**) to each unit in the following graph so this neural network simulates a binary logistic regression classifier: $Y = argmax_y P(Y = y|X)$, where $P(Y = 1|X) = \frac{exp(\beta_1 X_1 + \beta_2 X_2)}{1+exp(\beta_1 X_1 + \beta_2 X_2)}$, $P(Y = -1|X) = \frac{1}{1+exp(\beta_1 X_1 + \beta_2 X_2)}$.



(c) Following (b), derive β_1 and β_2 in terms of $w_1, \ldots, w_6$

With the sigmoid equation provided in question two, we can formulate the derivation of the β matrices as follows.

We can use the same general idea as noted in problem b.

$z_{2_1} = z_{1_1}w_5 + z_{1_2}w_6$
$z_{1_1} = X_1w_1 + X_2w_3$
$z_{1_2} = X_1w_2 + X_2w_4$
$z_{2_1} = (X_1w_1 + X_2w_3)(w_5) + (X_1w_2 + X_2w_4)(w_6)$
$z_{2_1} = (X_1w_1w_5 + X_2w_3w_5) + (X_1w_2w_6 + X_2w_4w_6)$
$z_{2_1} = X_1(w_1w_5 + w_2w_6) + X_2(w_3w_5 + w_4w_6)$
$P(Y = 1 \mid X) = \frac{e^{X_1(w_1w_5+w_2w_6)+X_2(w_3w_5+w_4w_6)}}{1+e^{X_1(w_1w_5+w_2w_6)+X_2(w_3w_5+w_4w_6)}}$

$\beta_1 = w_1w_5 + w_2w_6$
$\beta_2 = w_3w_5 + w_4w_6$

Which makes intuitive sense if you look at the paths of the flow from input to output.

Note: This is superfluous to follow. I mistook this as a backprop derivation question initially, but adjusted it to suit the specific question at hand. We can think about the composition of functions as per the following:

$z_{1_1} = X_1w_1 + X_2w_3$
$z_{1_2} = X_1w_2 + X_2w_4$
$z_{2_1} = z_{1_1}w_5 + z_{1_2}w_6$
$\hat{y} = \sigma(z_{2_1})$
We need to differentiate w.r.t each of the individual weights.

Our error function is binary cross entropy: $-y_i log(\hat{y}) - (1-y)(log(1-\hat{y})$The derivative of our error function is $\frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$ *Re: Jack's Neural Net slides. We're using sigmoid as our squashing function, which can be represented by: $\frac{e^z}{1+e^z}$ with a derivative of: $\frac{e^z}{1+e^z} * (1 - \frac{e^z}{1+e^z})$, or simply $\sigma(z) * (1 - \sigma(z))$.

Let's start with the weights closest to the output layer, because we can use a number of the gradients found in these steps as we get closer to the input layer of our network.

$\frac{\partial E}{\partial w_6} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial w_6} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * z_{1_2}$

$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial w_5} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * z_{1_1}$

$\frac{\partial E}{\partial w_4} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial z_{1_2}} * \frac{\partial \hat{z1_2}}{\partial w_4} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * w_6 * x_2$

$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial z_{1_2}} * \frac{\partial \hat{z1_2}}{\partial w_3} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * w_6 * x_1$

$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial z_{1_1}} * \frac{\partial \hat{z1_1}}{\partial w_2} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * w_5 * x_2$

$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_{2_1}} * \frac{\partial \hat{z_{2_1}}}{\partial z_{1_1}} * \frac{\partial \hat{z1_1}}{\partial w_1} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} * \frac{e^{z_{2_1}}}{1+e^{z_{2_1}}} * w_5 * x_1$

z_{2_1} represents the input into the output layer, before squashing, while z_{1_1} and z_{1_2} represent the summation/dot product of all of the features and weight being fed into each of the two neurons.

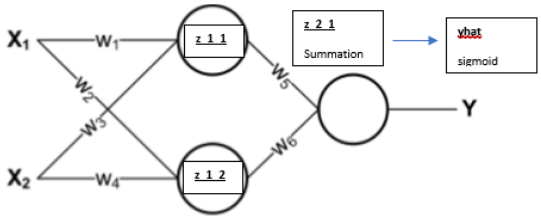


Figure 15:

Question 2 : Neuron Decision Boundaries I think this question is up to a bit of interpretation. We use the slides from the neural network session to understand that the linear combination of neuron outputs can act to soften the respective decision boundary at the next layer.

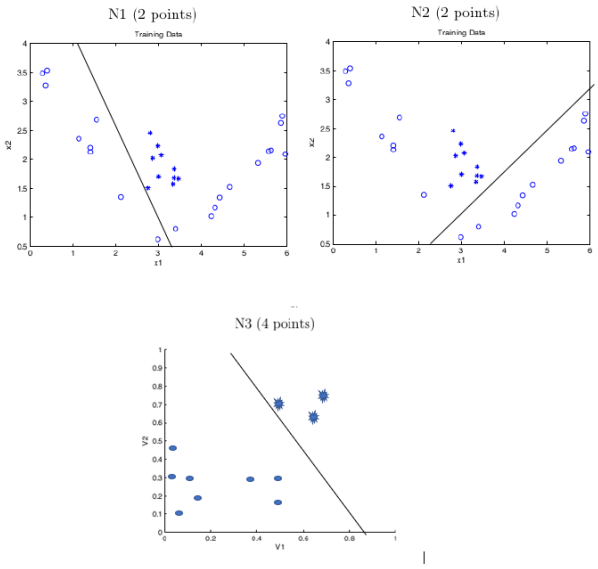


Figure 16: Neuron Decision Boundaries

I was initially thinking that we'd end with a nonlinear decision boundary at the output layer after inducing nonlinearity, but I'm not sure if that is explicitly what this question is looking for.

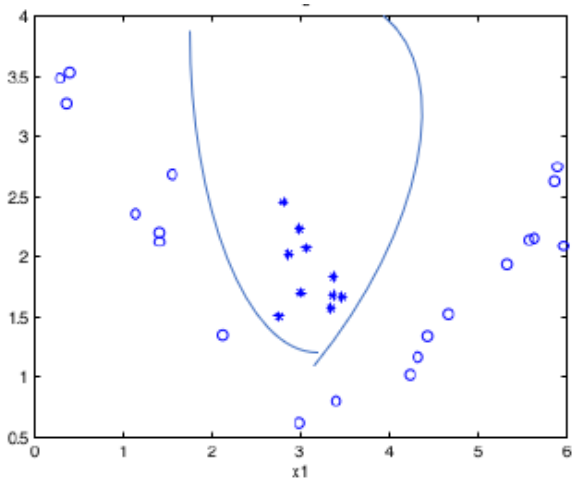


Figure 17: Neuron Decision Boundaries

I missed a submission question in the MLP section, but didn't want to disrupt the placement of the images above. The question asks why MSE is not a good choice for classification tasks. When we use cross entropy, whether it be binary or categorical, we are minimizing the negative log likelihood (MLE) for bernoulli/multinomial distributed variables. We are in essence trying to minimize the KL Divergence between two probability distributions for classification tasks, in which the probability is generated via the softmax output and the one hot encoded vector

pertaining to class responses. There is more of a penalty accrued (via loss) for predictions that are further away from the target.