CS 614 – Machine Learning

Michael Tripp

3/22/2023

Lab 3 Neural Network Project Discussion

For this project, I used the machine learning package Keras to build my models. In this project, I built a variety of dense and convolutional models with various hyperparameters and compared their accuracies. We will start by examining the dense models.

Dense Models

For the first three models, I focused on the effects of the loss and activation functions. Note that I bold which parameters I am changing/emphasizing from model to model.

dense_msquared_relu.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'mean_squared_error'
learningRate = 0.01
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train accuracy = 0.9558, 0.8729, 0.9636 ($\bar{x}$ = 0.93076)
test accuracy = 0.9221, 0.8386, 0.9332 ($\bar{x}$ = 0.89796)

dense_xentropy_relu.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.9765, 0.9799, 0.9825 ($\bar{x}$ = 0.97963)
test_accuracy = 0.9477, 0.9460, 0.9560 ($\bar{x}$ = 0.9499)

dense_tanh.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'tanh'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.9377, 0.9542, 0.9555 ($\bar{x}$ = 0.94913)
test_accuracy = 0.9104, 0.9277, 0.9238 ($\bar{x}$ = 0.92063)

From these three, I found that categorical cross entropy generally gives better results for this data than mean squared error as a loss function. Although we still had relatively good results using tanh as the activation function, relu still gave more accurate results. As such, I generally stuck with relu and categorical cross entropy in the following models as they produced the best average accuracies.

dense1.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 1
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.0984, 0.0984, 0.0994 ($\bar{x}$ = 0.09873)
test_accuracy = 0.0991, 0.0991, 0.0985 ($\bar{x}$ = 0.0989)

This first model showcases one of the issues I was having at the start of this project. I set my learning rate to 1 as a default, thinking back to previous examples of learning rates in class and assuming this was a good place to start. This, however, caused my model to train very badly as it turns out this learning rate was too large, most likely causing my model to converge too quickly and spit out a suboptimal solution.

dense2.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.1
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.0984
test_accuracy = 0.0991

dense3.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.02
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.3837, 0.6830, 0.1012
test_accuracy = 0.3767, 0.6600, 0.0996

As seen by the above 2 models, even learning rates as low as 0.1 or 0.02 were too large, although we do see the accuracy start to get a little better with a learning rate of 0.02. I did not record the model for a learning rate of 0.01, but I did run it a few times and found 0.01 worked really well. It seems going from 0.02 to 0.01 is the tipping point for a good learning rate, although I was sort of surprised to see this change so suddenly. I would think 0.02 would result in accuracies closer to 0.01.

dense4.h5
hiddenLayers = 3
layerSize = 64
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.9833, 0.9749, 0.9770
test_accuracy = 0.9488, 0.9304, 0.9449

dense4.h5 demonstrates the best learning rate I found, 0.001. Although 0.01 worked very well, I found this model to work slightly better with this learning rate.

dense5.h5
hiddenLayers = 10
layerSize = 32 ←**seemingly negligible**
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.9160, 0.8567, 0.8928
test_accuracy = 0.8681, 0.8125, 0.8381

Here, I modified the number of hidden layers as well as layer size. It is clear to see here that having too many hidden layers can greatly reduce the accuracy of a model. I experimented with a few other sizes as well, but generally found 3 or less to be the best number of hidden layers for this data. I also experimented with a few different layer sizes but did not really see any significant difference in accuracy.

dense5a.h5
hiddenLayers = 10
layerSize = 32
activationFunction = 'relu'
lossFunction = 'mean_squared_error'
learningRate = 0.001
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.1525, 0.1677, 0.1316
test_accuracy = 0.1486, 0.1619, 0.1319

Just to see how well mean squared error would do with an increased number of hidden layers, as well as the new learning rate and layer size, I plugged it in again as the loss function. Unsurprisingly, however, this tanked the accuracy.

dense6.h5
hiddenLayers = 5
layerSize = 32
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.9438, 0.9506, 0.9511
test_accuracy = 0.9082, 0.9165, 0.9032

Again, we see here that too many hidden layers can reduce accuracy. Although not bad accuracies here, I found using 3 hidden layers worked better.

dense7.h5
hiddenLayers = 3
layerSize = 16
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 5
batch_size_val = 10

train_accuracy = 0.7727, 0.9278, 0.9438
test_accuracy = 0.7201, 0.8776, 0.8820

Here, I tried a much lower layer size than previously, and this time found it significantly lowered the accuracy, even once getting accuracies below 80%. As such, it seems some layer sizes may not make a difference, but a layer size too low (and I'm sure too high) can cause problems. I'd guess this is because we are trying to fit too much data in too few connections.

dense8.h5
hiddenLayers = 3
layerSize = 32
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 50
batch_size_val = 10

train_accuracy = 0.9953, 0.9963, 0.9922, 0.9961
1s 3ms/step, 1s 2ms/step, 0s 2ms/step, 1s 2ms/step
test_accuracy = 0.9521, 0.9482, 0.9572, 0.9471
0s 2ms/step, 0s 3ms/step, 0s 2ms/step, 0s 2ms/step
Early stopping: 36/50, 50/50, 42/50, 31/50

In this model, I experimented with the number of epochs. Changing the size from 5 to 50, I found 5 was too low. Over 50 epochs, I was able to get significantly better accuracies than any of the other models I have made thus far. Some of the iterations of dense8.h5 were stopped by early stopping, but one even made it to the end. As such, I feel 50 is a good number of epochs to train with, as it gives my model the freedom to use all 50 epochs to train if needed or stop early otherwise.

dense9.h5
hiddenLayers = 3
layerSize = 32
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 50
batch_size_val = 50

train_accuracy = 0.9898, 0.9916, 0.9874
0s 2ms/step, 0s 2ms/step, 1s 2ms/step
test_accuracy = 0.9572, 0.9510, 0.9455
0s 2ms/step, 0s 2ms/step, 0s 3ms/step

In this last model, I modified the batch size to be 50 instead of 10, as it was. This changed seemed to noticeably speed up the time it took to go through each epoch and was done training very quickly even after 50 epochs. I compared the time per step to dense8.h5 and it does seem this one ran a little faster. Although a small difference, this model's accuracies were, however, a little worse on average than dense8.h5.

I did experiment with some other hyperparameters not listed here, although I did not find them important enough to list. Momentum, for example, did not seem to make a huge difference, but I did find 0.8 to work nicely compared to other values between 0 and 1. Overall, I found dense8.h5 to be my best model.

Convolutional Models

conv1.h5
hiddenLayers = 1
filterSize_input = 16
filterSize_hidden = 32
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 10
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9655, 0.9699, 0.9932
test_accuracy = 0.9299, 0.9438, 0.9538

I will be using this model as the base model. We can see these parameters already give good accuracies. We will see which parameters can be changed to make a significant difference from here. Notice that there are some new hyperparameters being introduced, such as kernel and pooling size, and the layer size has been replaced by number of filters (filterSize).

conv2.h5
hiddenLayers = 1
filterSize_input = 64
filterSize_hidden = 128
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 10
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9979, 0.9979, 0.9979
test_accuracy = 0.9800, 0.9800, 0.9755

For this model, I multiplied the filter sizes by a multiple of 4 and it turned out to have consistently very high accuracies for both the test and training accuracies. It is interesting that the train_accuracy came back the same three times in a row, and the test_accuracy twice, so I'm wondering if it is reaching some upper bound for accuracy with these parameters. I'm also wondering as to what point increasing the number of filters will not increase accuracy, as well as how this will change given more hidden layers or more epochs.

conv3.h5
hiddenLayers = 1
filterSize_input = 64
filterSize_hidden = 128
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 50
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9982, 0.9984, 0.9979
test_accuracy = 0.9783, 0.9761, 0.9761
early stopping: 17/50, 17/50, 17/50

To test if these filter sizes can be even more accurate given increased number of epochs, I created this model. According to the accuracies, we can see that this change actually does improve our accuracy very slightly. For subsequent models with these hyperparameters, I would say no more than 20 epochs is necessary since it often stops at epoch 17 due to early stopping.

conv4.h5
hiddenLayers = 1
filterSize_input = 64
filterSize_hidden = 128
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.001
momentum = 0.8
numEpochs = 50
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9979, 0.9979, 0.9982
test_accuracy = 0.9783, 0.9750, 0.9777
early stopping: 33/50, 33/50, 34/50

Although changing learning rate here to 0.001 does not seem to impact our accuracies at all as it did with our dense models, I wanted to include this model as it took much longer to arrive at these answers. While the previous model always stopped early around epoch 17, these all got as far as 33 or 34 epochs before stopping. This, of course, makes sense since a smaller learning rate would slow down the rate at which the weights in our neural network are updated, which would slow down the overall training process.

conv5.h5
hiddenLayers = 1
filterSize_input = 64
filterSize_hidden = 32
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 15
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9974, 0.9969, 9636
test_accuracy = 0.9694, 0.9789, 0.9360
early stopping: 15/15, 14/15, 12/15

For this model, I wanted to do some more testing on the filter size. I'm not entirely sure what makes a good filter size, just that 64 and 128 did give me good accuracies. As such, I wanted to try a hidden layer filter size that is smaller than the input filter size instead. This one generally performed very well, although not as consistent as other models.

conv6.h5
hiddenLayers = 1
filterSize_input = 64
filterSize_hidden = 128
activationFunction = 'relu'
lossFunction = 'categorical_crossentropy'
learningRate = 0.01
momentum = 0.8
numEpochs = 20
batch_size_val = 10
kernel_size = (3, 3)
pooling_size = (2, 2)

train_accuracy = 0.9979, 0.9982, 0.9982
test_accuracy = 0.9761, 0.9777, 0.9783
early stopping: 17/20, 19/20, 17/20

This last model is simply conv3.h5 but with 20 epochs instead of 50. I'd say out of all of these, conv3.h5 was the most accurate, but I did not think 50 epochs was necessary, so I made this one. As such, I would say this model, conv6.h5, is my most accurate model.