Project #2: Building a Convoluted Neural Network to Classify a Noisy Dataset of Handwritten Digits

ECE 566: Machine & Deep Learning

Due: December 4, 2021

Daniel Marten

A20394276

I. <u>Introduction</u>

The introduction of a convoluted neural network (CNN) and extensive usage of the TensorFlow-Keras package has allowed for significant advancements over the manually coded neural network classifier as seen in Project #1. The CNN designed for this project, with code seen in **Appendix 1**, allows us to explore the classification of a dataset of the ten handwritten arabic numerals (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) using a number of different methods. The impact of adding different layers and changing their parameters is also explored, namely the depth and dimensionality of CNN's namesake 2D convolution layers and batch size, as well as the implementation of batch normalization. The latter was hypothesized to be necessary due to this dataset's increased noise over the set in Project #1. The usage of the Keras option *EarlyStopping* to prevent unnecessary runtimes on plateaus or after a maximum validation accuracy is heavily implemented. The major comparison in this report concerns the usage of Keras' three optimizers: Adadelta, Adam, and the gradient descent method SGD. As a broad conclusion, the usage of Adam or SGD with batch normalization allows for the best results by way of test accuracy, with results approximately at or above 90% testing accuracy for a notably noisy dataset.

The project is coded in Python and extensively uses numpy, matplotlib, and TensorFlow-Keras. The code was run on Google Colabs Pro, as privately purchased by myself, and using the GPU acceleration option with my 2019 Asus Zenbook. Further, the code provided by Professor Jovan Brankov, notes from class, and starting suggestions for CNN designs online (as cited in the **References**) were indispensable in this project.

II. <u>Summary</u>

The CNN models designed in **Appendix 1** from *dmModel* obtained test accuracy levels of 90.38% with SGD (batch size of 4) and 90.30% with Adam optimization methods, both using batch normalization and the set of weights with the lowest validation accuracy value to predict the testing data. The SGD setup required approximately 6 minutes for a batch size of 4, with Adam requiring 7.1 minutes with the same batch size. Comparatively, Adadelta methods took greater amounts of time to return lower test accuracy values with larger batch values. Further, parameter investigations revealed the tradeoff for batch size — that lower values take longer to run but return better validation accuracy values for the same amount of epochs — as well as for 2D convolution layer depths — where larger values take longer to run but return improved validation accuracy

metrics. Considering the advanced degree of noise in our dataset, which is difficult to classify using just the human eye, returning above a 90% testing accuracy for our SGD method in under 10 minutes can be considered a modest success, though there is room for improvement. It is important to note that, because these numbers are stochastic, exact values for runtimes and testing accuracy may change between trials, but overall trends and results ought to remain consistent.

III. <u>Body</u>

<u>Part I: Dataset Concerns</u>

When comparing the results of this project to Project #1 — which boasted a testing accuracy of 99% for two-digit classification — it is important to consider that this dataset includes added noise which makes classification significantly more difficult. The dataset is the MNIST dataset with noise added, containing 60,000 images of size 28-by-28 pixels.Below in **Figure 1** examples of the digits 0, 5, 7, and 9 are shown, and it can be seen that the image is not clear.

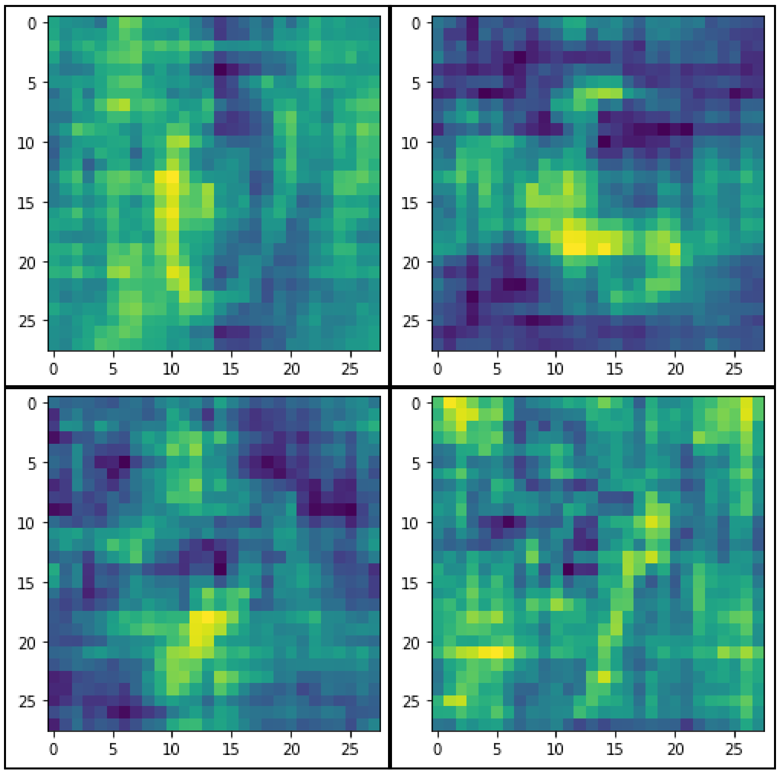


Figure 1: *Examples of the digits 0, 5, 7, and 9 from the dataset with noise added. While the center "hole" of the 0, the approximate shape of 5, the "tail" of 7, and the approximate shape of 9 can all be seen, the noise and unclearness of shape may pose a challenge to CNN. Digits are to be read from left to right. Compare to the original MNIST dataset in* ***S.1****.*

To account for this added noise, batch normalization as part of TensorFlow-Keras was imported and utilized. According to the documentation, batch normalization returns the input mean at approximately 0 while converting the standard deviation to approximately 1. While the specifics of the method are beyond the immediate scope of this problem, it is important to note that pre-processing and data transformations before (or during and after) machine learning processes can be some of the most subtle yet effective ways to improve results without drastically increased runtime. Consider them before increasing epochs or lowering learning rate automatically. For the remainder of this report, unless otherwise noted, all methods and tests employ batch utilization. Proof of their effectiveness on model performance and accuracy is included in Part III: Model Evaluation in **Figure 6**.

Part II: Parameter Concerns and Trade-Offs

Design blocking and structure was inspired by the pre-written code, the AlexNet example from lectures, and tutorials from the following source, though additions and changes needed to be made, as it did not work with such a noisy dataset [1]. Two blocks of a 2D convolution, batch normalization, pooling, and then a dropout layer comprise the first half of the model, before flattening and two density layers are added, as seen in the function *dmModel* in **Appendix 1**. For more notes on this, see **Recommendations and Improvements**. It is important to note that the dropout layers are included so as to prevent overfitting of the data. A crude drawing of the model can be seen in **Figure S.2**.

However, the depth of the 2D convolutions needed to be explored. To this end, the function *layerTesting* in **Appendix 1**, though it would more aptly be called something to the effect of a convolution depth testing method. For the two convolution layers in the model, a brief test was conducted to observe the relationship between convolution depth, validation accuracy, and the time it takes for an epoch to run. The results are seen below in **Table #1**:

From layerTesting, with batch=200			
Depth 1	Depth 2	Time per Epoch (s)	Val. Acc. after 3 Epochs
100	50	2-3	0.1637
150	50	3	0.1657
150	100	3-4	0.2125
200	50	3-4	0.1751
200	100	3-4	0.1914
200	150	4	0.2957
500	50	6	0.2193
500	100	6-7	0.2691
500	150	7-8	0.2904
500	200	7-8	0.309

Table 1: *The relationship between convolution depth, time, and validation accuracy after 3 epochs is shown in the table above. As depth increases, so does both time and accuracy.*

From this relationship, though it could potentially be further optimized, depths of 200 for the first convolution layer and 100 for the second convolution layer were chosen as a middle ground for both time and validation accuracy. It is important to note that the results for validation accuracy are stochastic and may not be repeated exactly if the same code is repeated.

Similarly, batch size was also investigated. The function *blockSizeTest* from the code in **Appendix 1** was used for this and produced the relationship seen in **Table 2**. The relationship of batch size to epoch timing and validation accuracy is opposite to that of convolution layer depth, as here larger values confer shorter runtimes but decreased accuracy. When run with a constant convolution depths of 200 and 100, the graphs for validation accuracy in **Figure 2** are produced.

From blockSizeTesting, depths of 200 and 100		
Batch Size	Time per Epoch (s)	Val Acc after 10 Epochs
8	26-28	0.6471
16	14-15	0.6094
32	9-10	0.5489
64	6-8	0.5051
128	4-5	0.4472

Table 2: *The relationship between batch size and timing and validation accuracy is opposite that of convolution depth.*

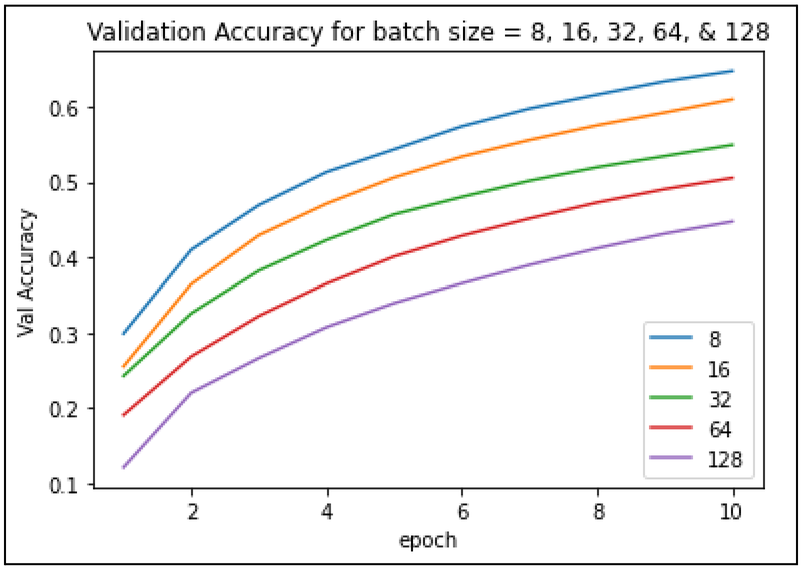


Figure 2: *The relationship above is graphed, with validation accuracy plotted over 10 epochs for the 5 chosen batch options. The code was produced in **Appendix 1**, and can be seen if the code is ran*

As seen in **Figure 2**, running analyses at lower batch sizes will confer increased validation accuracy. However, this comes at the cost of increased computation times. Potentially, it can be suggested to keep your batch sizes high when first designing or testing the construction of a model, then to use lower batch values when conducting your final testing and evaluations. This is why a batch size of 200 was used when this report conducted the prior investigation into convolution layer depth, so as to ensure the analyses were fast.

The additional TensorFlow-Keras callbacks *EarlyStopping* and *ReduceLRonPlateau* were also explored. EarlyStopping is included in *dmModel* in **Appendix 1** and is used when constructing all models for later visualization, so as to avoid unnecessary runtimes after a maximum validation accuracy has been met, as time and computational resources are at a premium. Though *ReduceLRonPlateau* could potentially be useful, it could not be employed to any meaningful extent in this project. Trials not included in this report were conducted with the default Adadelta method with *ReduceLRonPlateau*, which is prone to plateauing and significantly slowing down, but it never resulted in any meaningful increase in validation accuracy or any decrease in validation loss. With the above model design covered, evaluations of the actual models are included below.

Part III: Model Evaluation

Five different models of a potential CNN were tested: SGD with a batch size of 4, SGD with a batch size of 8, Adam with a batch size of 4, Adadelta with a batch size of 16, and Adadelta with a batch size of 16 but without batch normalization. The results of these models are shown in the remainder of the section, and all above trials used the previously stated convolution layer depth parameters of 200 and 100, a validation data fraction of 0.25, and *EarlyStopping*. As seen in **Table 3**, the results for both SGD models and the Adam model outperformed that of both Adadelta models. Graphs for accuracy and loss for both SGD models can be seen in **Figure 3**. Information on validation and training accuracy can be seen in **S.2.**

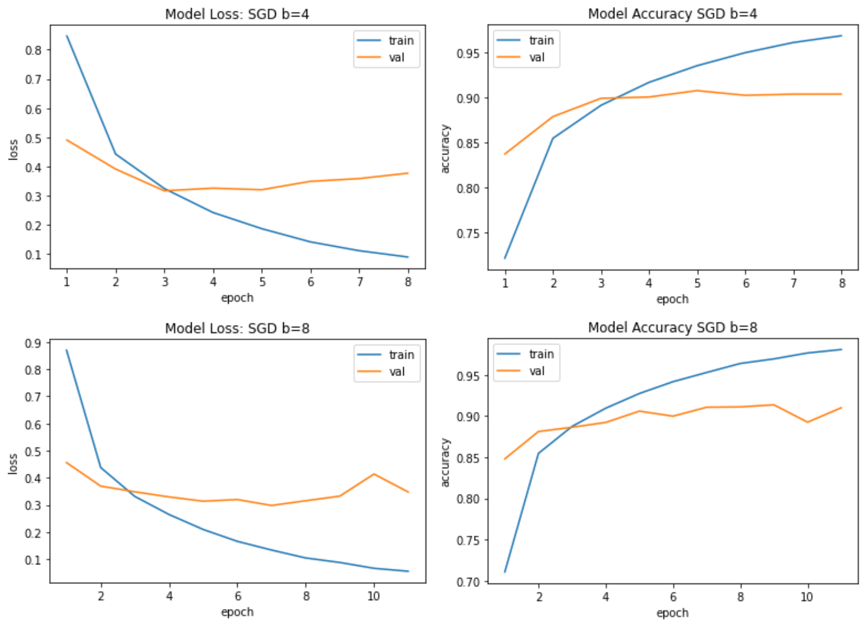


Figure 3: *Graphs of model loss and accuracy for SGD methods are shown above, with a batch size of 4 in the top row and a batch size of 8 in the bottom row. These graphs can also be seen and produced from the code in* ***Appendix 1****.*

From the graphs in **Figure 3**, the Keras' own model-fitting method selected the weights for the highest validation accuracy. This method reports a testing accuracy of 90.38% for a SGD with a batch size of 4 and 90.69% for a batch size of 8. A complete list of these testing accuracies can be seen in **Table 3**. **Figure 4** shows similar graphics for the optimizer Adam, with its weights selected in the same way.

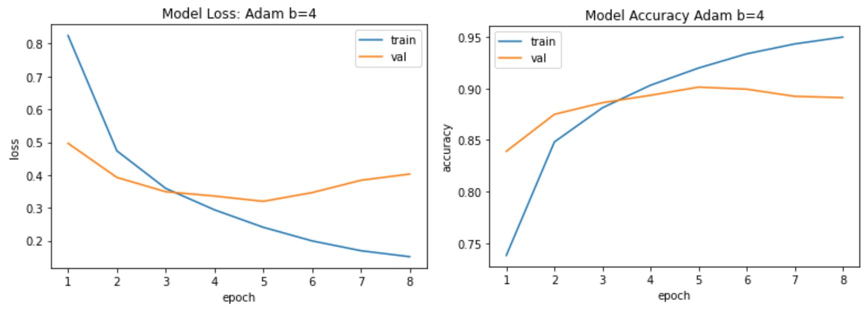


Figure 4: *Graphs of model loss and accuracy for Adam with a batch size of 4.*

The model using Adam returned a final testing accuracy of 90.30%, very similar to the testing accuracy of both SGD methods. All three methods are compared and graphed together below in **Figure 5**, plotting validation accuracy across epochs during training.

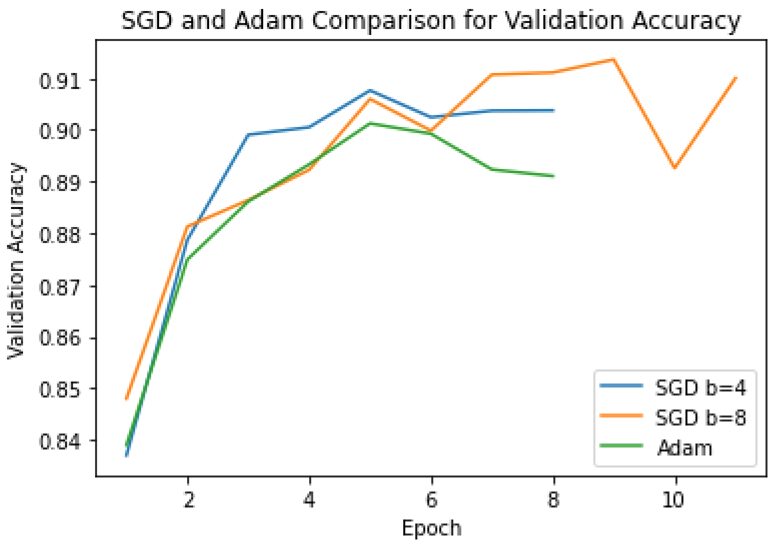


***Figure 5**: Comparison of validation accuracy across epochs for SGD (for batch size of 4 and 8) against Adam, as color-coded in the legend. Note the similar behavior of the three methods, as supported by their similar final testing accuracies.*

As seen above, both SGD models and the Adam model return final testing values approximately at or above 90%. At the same time, it is important to note that all methods used in this report are stochastic, and running the code multiple times may result in slightly different percentages. These results of these three models can be contrasted to those of Adadelta, with comparatively lower results for testing accuracy. Adadelta was run with and without batch normalization — to show the effect of the two instances of the processing layer — with a batch size of 16. It is important to note that these Adadelta methods took significantly longer to run than either SGD or Adam, as noted in **Table 3**. The results of these methods are seen below in **Figure 6.**

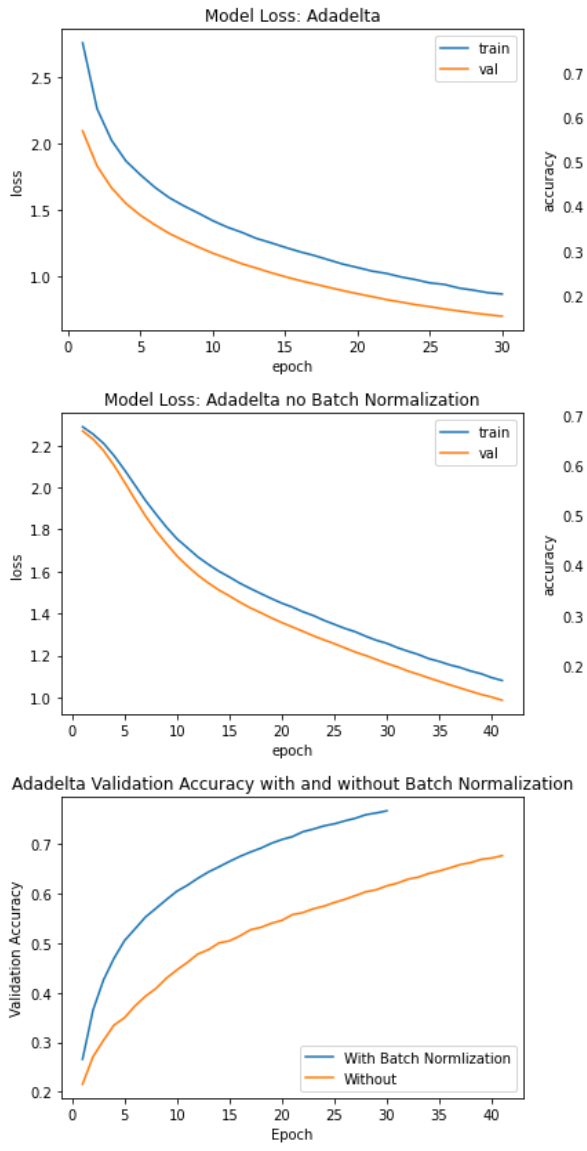


Figure 6: *Graphs for model loss and accuracy for Adadelta with and without batch normalization. Note the slower and more gradual slopes in these models compared to SGD and Adam. Also note the difference in validation accuracy at approximately 30 epochs between the method with versus without batch normalization.*

The Adadelta models report a testing accuracy of 76.3% with and 67.5% without batch normalization. Both methods also took between 7 and 8 minutes to run. A full list of model accuracies and run times can be seen in **Table 3.** See **Table S.1** for validation and training accuracies.

Model	Testing Accuracy	Runtime (min)
SGD b=4	90.38	6.0
SGD b=8	90.69	4.2
Adam	90.3	7.1
Adadelta	76.3	7.3
Adadelta w/o BN	67.54	8.0

Table 3: *Summary of final testing accuracy and runtimes for each method. Note that runtime is very approximate.*

From **Table 3**, note that accuracies for SGD and Adam models are all equal to or greater than 90.0% while Adadelta methods do not pass even 80% for equal or greater amounts of time. Though Adadelta may pass 90% given sufficient time, time and memory usage are significant factors when constructing neural networks, and the models' (as they are constructed in this report) comparative slowness with worse results are grounds for labelling the constructed models for SGD and Adam the superior models, at least as they are used in this report. While there may be ways or alternative processing methods for Adedelta to equal or surpass the others, it was not found or utilized for this project.

IV. Conclusion

In conclusion, the use of CNN models was able to produce three different models which predicted the class (or digit) of the testing data set (handwritten arabic numerals with noise) with an accuracy equal to or greater than 90.0%. These models utilized SGD with a batch size of 4, SGD with a batch size of 8, and Adam. Considering the significant amount of noise on the dataset, such a testing accuracy can be considered a modest success. Adadelta methods were not able to pass 80% testing accuracy within the same amount of time. This is not to say that Adadelta methods as a whole are inferior, just that the models constructed with Adadelta in this project were outperformed by those with SGD and Adam. Potential improvements could be made by investigating the impacts of other parameters and options, like the activation functions or arguments for density layers — see more in **Recommendations, Improvements, and Acknowledgements**

V. Recommendations, Improvements, and Acknowledgements

Preface: As in Project #1, this section may not be whole or presented as clearly as the remainder of this report. Rather, it may be a less formal list of future improvements or areas of concern about the current state of the project.

- More preprocessing, filtering, and data correction could be done in order to return both greater testing accuracies and faster processing times.
- By no means is ~90% the limit of testing accuracy due to the data's noisiness. Other methods could return significantly better results. Further, the fact that humans (see: myself) have a hard time distinguishing the digits does not mean that computers necessarily would. Computers and machine learning models can frequently out-perform humans at similar classification tasks, as discussed in class.

- Other methods and layers could have been added to the *dmodel* function to potentially improve its performance. However, it stuck fairly closely to the code written by the professor, provided in notes, and seen in the source [1]. Adding further layers could significantly increase performance.
- Similarly, the kernel sizes for *Conv2D* were not modified and neither were the parameters passed to *Dense*. Given more time and the inclination, these could be explored similar to *Conv2D* depth and batch size. Alternatives to the activation functions used could also be explored in the future.
- As mentioned previously, the results of this project are not enough to say that Adadelta AS A WHOLE is worse than SGD and Adam, just that they perform worse on this dataset.
- I look forward to using *ReduceLRonPlateau* in the future, and I am slightly disappointed and confused as to why it did not work with some Adadelta models that are not shown in the report.
- I would like to formally thank my girlfriend for all the support and help she gave me on this project, including cooking me a small dinner while I was finishing this report on the Saturday of December 4, 2021.
- I claim no competing interests or relevant outside funding for this project.

VI. Supplemental Figures

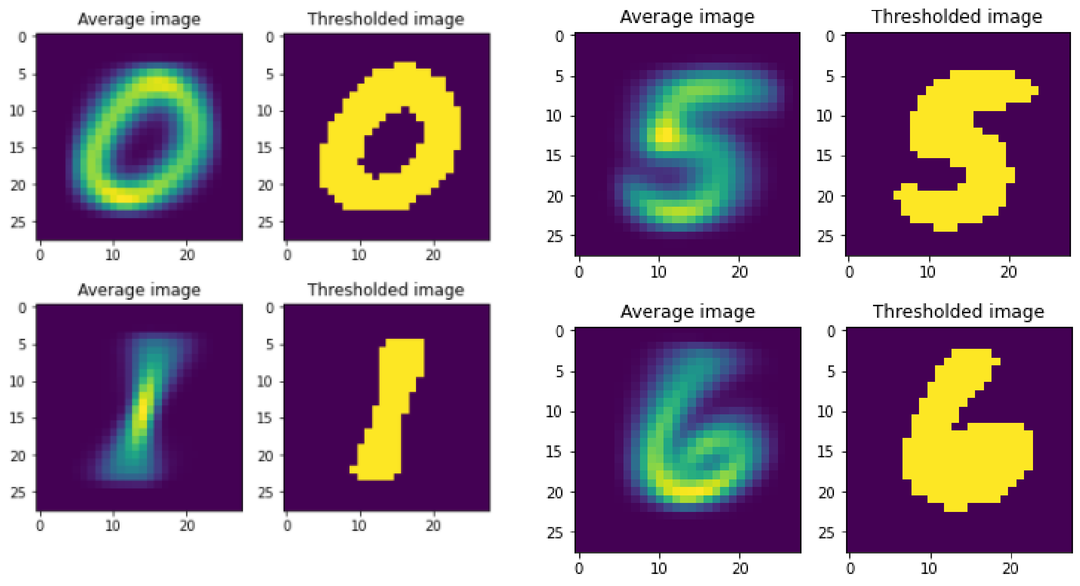
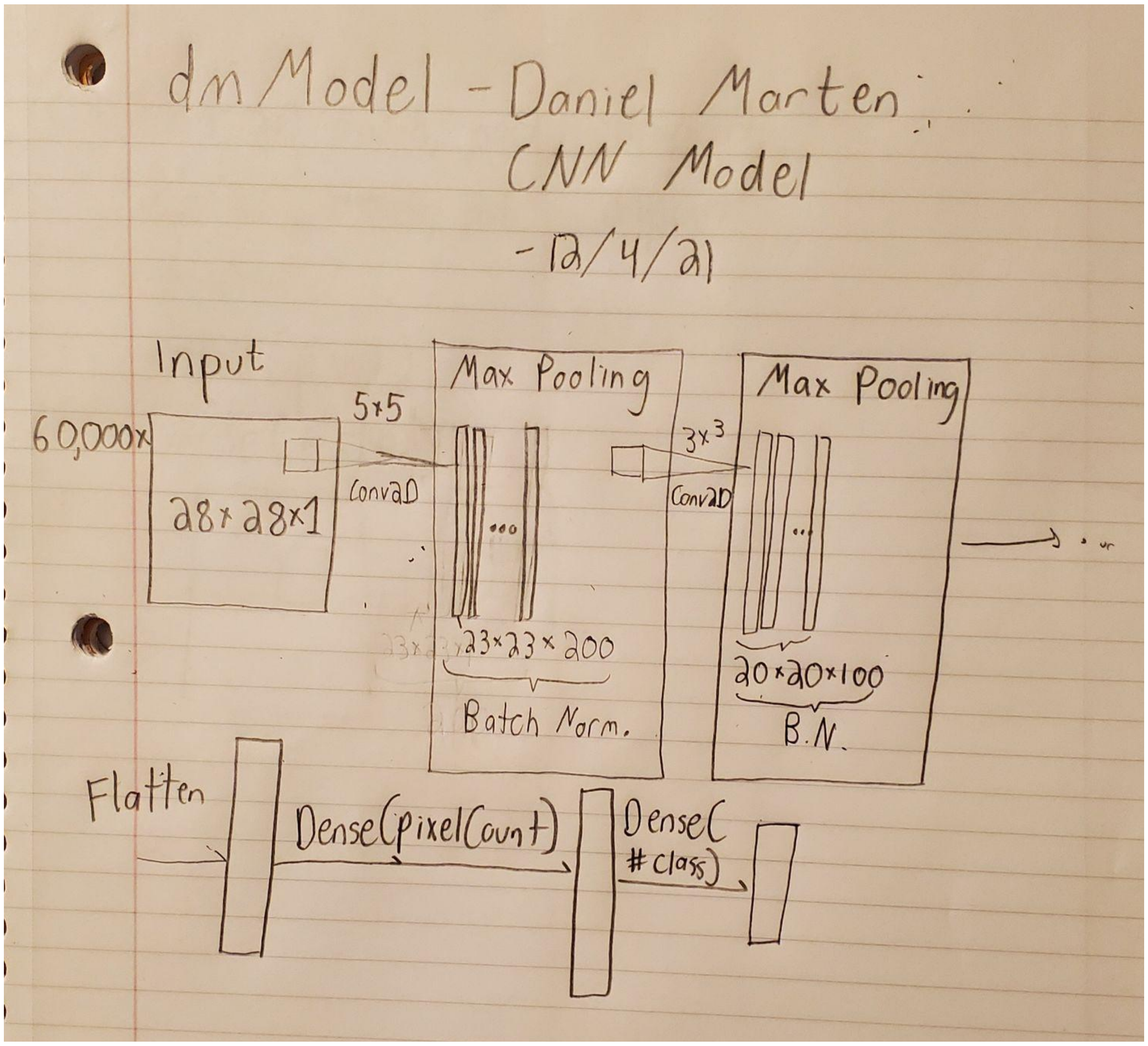


Figure S.1: *Refer to these averaged and thresholded images from Project #1 to observe their lack of noise, compared to the noisy digits used in this project.*

Model	Training Accuracy	Runtime (min)	Final Training Accuracy	Final Validation Accuracy
SGD b=4	90.38	6.0	96.9	90.4
SGD b=8	90.69	4.2	98.1	91.0
Adam	90.3	7.1	95.7	90.3
Adadelta	76.3	7.3	71.0	76.7
Adadelta w/o BN	67.54	8.0	64.2	67.6

Table S.1: Testing accuracy data, along with the final values for training accuracy and validation accuracy. This can be considered supplemental to **Table 3**. It can also be noted that the final values for training and validation accuracy may not correspond with the lowest validation accuracy, which is used to determine the parameter weights, which determine the testing accuracy.

Figure S.2: A pencil-and-paper drawing of the CNN



SOURCES:

[Throughout]: Professor Jovan Brankov, *Class Notes in ECE 566*, Fall 2021, Illinois Institute of Technology.

[1]: Jason Brownless, *Handwritten Digit Recognition using Convolutional Neural Networks in Python with Keras*, Machine Learning Mastery, June 27, 2016, https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/?fbclid=IwAR1hlsS8DOv9pN2L3rOgIaHyrv8gkNCTkkzMVD8ICQG4XvN02qS3tYjbQj0

```
## APPENDIX 1
## Daniel Marten - Submitted: 12/4/2021
## Project 2 - ECE 566

# Importing packages, standard from code
# tensorflow.keras
# additionally imported Batch Normalization which helps noisey data

%matplotlib inline
import os
import numpy as np
import pickle
import matplotlib.pyplot as plt

from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ReduceLROnPlateau, ModelCheckpoint, EarlyStopping
from tensorflow.keras.optimizers import Adadelta, Adam, SGD
from tensorflow.keras.layers import Input, Conv2D, Dense, MaxPooling2D, Dropout, Flatten, Ave
from tensorflow.keras.models import Sequential
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.layers import BatchNormalization

from google.colab import drive
drive.mount("/content/gdrive", force_remount=True)
os.chdir("/content/gdrive/My Drive/Colab Notebooks") # Points to my own Google Drive Folder
# may correct later
```

```
     Mounted at /content/gdrive
```

```
data = np.load('./MNIST_CorrNoise.npz') # Importing the dataset which includes noise
# this is a LOT of noise oh my goodness


x_train = data['x_train'] # Defining the training data
y_train = data['y_train']

num_cls = len(np.unique(y_train)) # Defining the number of classes
print('Number of classes: ' + str(num_cls))

print("x train shape as: " + str(x_train.shape))
pixelCount = x_train.shape[1]*x_train.shape[2] # Calculates size of the total number of pixel
# this is used for the first Density layer, seen below
print("pixel count as: " + str(pixelCount))
# pixelCount = x_train.shape[1]**2 ## ASSUMES SQUARE

print('Examples of handwritten digit with correlated noise: \n')

kImg = [500*x for x in range(2,12)] # 10 handwritten digits to show the extent of the noise
```

```
#k = 3001 these are actually very hard to tell sometimes
# SHOWING said noise

for k in kImg:
  plt.imshow(np.squeeze(x_train[k,:,:]))
  plt.show()
  print('Class: '+str(y_train[k])+'\n')

# RESHAPE and standarize
x_train = np.expand_dims(x_train/255,axis=3) # standardizing to make working easier

# convert class vectors to binary class matrices
y_train = to_categorical(y_train, num_cls)

print('Shape of x_train: '+str(x_train.shape))
print('Shape of y_train: '+str(y_train.shape))

# print(x_train[k])
# print(y_train[k])




# Defining a model
# As a function, ONLY works with our dataset
# Pick from Adadelta, Adam, SGD -- CASE SENSITIVE

input_shape = x_train.shape[1:4] #(28,28,1) # Set-up work, defining shape

if not os.path.exists('./weights'):
  os.mkdir('./weights') # making sure there is a place for weights to be stores


# THE BELOW IS WHERE A MAJORITY OF WORK IN THIS PROJECT IS DONE
# IT CONTAINS THE DESIGN OF THE CONVOLUTED NEURAL NETWORK
# dmModel for Daniel Marten's Model
def dmModel(layer1num,layer2num,bVal,eVal,vVal,verboseVal=1,modelName="testName",batch=True,o

  #print("we are initializing!!!")
  pweight='./weights/weights_'+modelName+'.hdf5' # defining where the weights will be stores
  model = Sequential()

  # START layer design
  model.add(Conv2D(layer1num, kernel_size=(5, 5), activation='relu', input_shape=input_shape)
  if (batch==True):
    model.add(BatchNormalization()) # reduces noise
  model.add(MaxPooling2D()) # pooling

  model.add(Dropout(0.2)) # reduces overfitting

  # Repeat the block of code above
```

```
    model.add(Conv2D(layer2num, kernel_size=(3,3), activation='relu',input_shape=input_shape))
    if (batch==True):
      model.add(BatchNormalization())
    model.add(MaxPooling2D())

    model.add(Dropout(0.2)) # further reduces overfitting

    model.add(Flatten())
    model.add(Dense(pixelCount, activation='relu')) # could optimize these values, did not have
    model.add(Dense(num_cls, activation='softmax'))
    #END layers

    # THREE optimizers, of which SGD and Adam are preferred over the default Adadelta
    if (optimizer == 'SGD'):
      model.compile(loss=categorical_crossentropy,
        optimizer=SGD(), # explore other optimizers: Adam, SGD
        metrics=['accuracy'])
    elif (optimizer == 'Adam'):
      model.compile(loss=categorical_crossentropy,
        optimizer=Adam(), # explore other optimizers: Adam, SGD
        metrics=['accuracy'])
    else:
      model.compile(loss=categorical_crossentropy,
        optimizer=Adadelta(), # explore other optimizers: Adam, SGD
        metrics=['accuracy'])

    model.summary()

    checkpointer = ModelCheckpoint(filepath=pweight, verbose=verboseVal, save_best_only=True, m
    callbacks_list = [checkpointer] # explore adding other callbacks such as ReduceLROnPlateau,

    if (reduce==True):
      lr = ReduceLROnPlateau(monitor='val_accuracy',patience=1,factor=0.1,min_delta=0.01,verbos
      # lowers learning rate at platuea
      # SEEMS incredibly useful if learning rate is the 'dependent variable'
      # but it seems this problem has other limiting variables on accuracy
      callbacks_list.append(lr)
    elif (earlyStop==True):
      es = EarlyStopping(monitor='val_accuracy',patience=eStopPatience,min_delta=eStopVal,verbo
      # stops at plateau
      # INCREDIBLY useful for new or long processes
      # however, over time, if running the same process twice, the user can just adjust the epo
      # ... accordingly and not use earlyStop
      callbacks_list.append(es)

    # actual fitting!
    historyIn=model.fit(x_train, y_train,
                          epochs=eVal,
                          batch_size=bVal,
                          verbose=verboseVal,
                          shuffle=True,
                          validation_split = vVal,
                          callbacks=callbacks_list)

      # some reporting
      valAccArr = historyIn.history['val_accuracy']
      # print("Val_Accuracy array: " + str(valAccArr))
      print("Final val accuracy: " + str(valAccArr[-1]))
      delta = valAccArr[-1]-valAccArr[-2]
      print("Difference in last two: " + str(delta))

      print('CNN weights saved in ' + pweight)

      return historyIn, valAccArr[-1], delta

    def modelPrinting(input,titleStr):
      # fairly self explanatory, code for printing as provided by professor
      # Plot loss vs epochs
      #plt.plot(input.history['loss'])
      #plt.plot(input.history['val_loss'])
      plt.plot(range(1,len(input.history['loss'])+1),input.history['loss'])
      plt.plot(range(1,len(input.history['val_loss'])+1),input.history['val_loss'])

      plt.title('Model Loss: '+(titleStr))
      plt.ylabel('loss')
      plt.xlabel('epoch')
      plt.legend(['train', 'val'], loc='upper right')
      plt.show()

      # Plot accuracy vs epochs
      plt.plot(range(1,len(input.history['accuracy'])+1),input.history['accuracy'])
      plt.plot(range(1,len(input.history['val_accuracy'])+1),input.history['val_accuracy'])
      plt.title('Model Accuracy ' + (titleStr))
      plt.ylabel('accuracy')
      plt.xlabel('epoch')
      plt.legend(['train', 'val'], loc='upper left')
      plt.show()




    def layerTesting(layerListIn):
      # function modelled to test for optimal layer depths
      # used to find (200,100) as optimal arguments
      # assumes layerListIn is sorted, kind of an oversigh

      i = 1
      minDelta = 0
      minDString = ""
      minValAcc = 0
      minVString = ""

      for l1 in range(1,len(layerListIn)):
```

```
      lnum1 = layerListIn[l1]
      for l2 in range(0,l1):
        # print("ayylmao")
        lnum2 = layerListIn[l2]
        pairStr = (str(i) + "- Layer Pairing: ("+str(lnum1)+","+str(lnum2)+")")
        print(pairStr)
        # workingModel = dmModelTwoLayers(lnum1,lnum2)
        #model = Sequential()
        # test with
        # modelTest, pRet = dmTwoLayersBatchNorm(lnum1,lnum2)
        modelTest2, val, delVal = dmModel(lnum1,lnum2,200,3,0.25)
        #print("Val as: " + str(val))
        #print("DelVal as: " + str(delVal))
        if val > minValAcc:
          minValAcc = val
          minVString = pairStr
        if delVal > minDelta:
          minDelta = delVal
          minDString = pairStr
        i += 1

    dt1 = [minDelta, minDString, minValAcc, minVString]

    print("DEPTH TEST")
    print("Max Delta of: " + str(dt1[0]) + " at: " + str(dt1[1]))
    print("Max Acc of: " + str(dt1[2]) + " at: " + str(dt1[3]))

    return [minDelta, minDString, minValAcc, minVString]

  def blockSizeTest(blockListIn):
    # function modelled to test for optimal batch size
    # investigated said relationship, no 'best' value
    # how much time and RAM do you have?
    #
    i = 1
    minDelta = 0
    minDString = ""
    minValAcc = 0
    minVString = ""

    for bIter in blockListIn:
      print("Testing at block value of: " + str(bIter))
      modelTest3, val, delVal = dmModel(200,100,bIter,3,0.25)
      print("Val as: " + str(val))
      print("DelVal as: " + str(delVal))
      if val > minValAcc:
        minValAcc = val
        minVString = str(bIter)
      if delVal > minDelta:
        minDelta = delVal
        minDString = str(bIter)
      # i += 1

    bst = [minDelta, minDString, minValAcc, minVString]

    print("BLOCK TEST TEST")
    print("Max Delta of: " + str(bst[0]) + " at: " + str(bst[1]))
    print("Max Acc of: " + str(bst[2]) + " at: " + str(bst[3]))

    return [minDelta, minDString, minValAcc, minVString]




  # Opserving the impact of layers on depth on Conv2D

  # layerList = [1,2,5,10,15,20,25,50,100, 200]#,75,100]
  layerList02 = [50,100,150,200,500]
  # fakeList = [10,50,200]

  layerTesting(layerList02)

  # Observing the impact of batch size
  batchList = [8,16,32,64,128]
  epoch = range(1,11)

  batch01, b, c = dmModel(200,100,batchList[0],10,0.25) # five different batch sizes
  batch02, b, c = dmModel(200,100,batchList[1],10,0.25) # one major player is timing, which is
  batch03, b, c = dmModel(200,100,batchList[2],10,0.25)
  batch04, b, c = dmModel(200,100,batchList[3],10,0.25)
  batch05, b, c = dmModel(200,100,batchList[4],10,0.25)

  plt.plot(epoch,batch01.history['val_accuracy'])
  plt.plot(epoch,batch02.history['val_accuracy'])
  plt.plot(epoch,batch03.history['val_accuracy'])
  plt.plot(epoch,batch04.history['val_accuracy'])
  plt.plot(epoch,batch05.history['val_accuracy'])
  plt.title('Validation Accuracy for batch size = 8, 16, 32, 64, & 128')
  plt.ylabel('Val Accuracy')
  plt.xlabel('epoch')
  plt.legend(['8', '16', '32', '64', '128'], loc='lower right')
  plt.show()





  # Early Stopping Example
  # sgdStop003, ssb003, ssc003 = dmModel(200,100,4,50,0.25,optimizer='SGD',earlyStop=True,model
  # modelPrinting(sgdStop003,'SGD EarlyStop Misuse') # this is OVERLY judicious

  sgdStop004, ssb004, ssc004 = dmModel(200,100,4,50,0.25,optimizer='SGD',earlyStop=True,modelNa
  modelPrinting(sgdStop004,'SGD b=4') # good results! Or so we believe at least
```

```
sgdStop005b8, ssb005, ssc005 = dmModel(200,100,8,50,0.25,optimizer='SGD',earlyStop=True,model
modelPrinting(sgdStop005b8,'SGD b=8') # SGD, but with a batch of 8



# Adam Example

adam001, ab001, ac001 = dmModel(200,100,4,50,0.25,optimizer='Adam',earlyStop=True,modelName='
modelPrinting(adam001,'Adam b=4') # good results! Or so we believe at le


# Default AdaDelta example

#default001, db001, dc001 = dmModel(200,100,16,50,0.25,earlyStop=True,modelName='DEFAULT001',
modelPrinting(default001,'Adadelta') # using b=16 here to due time considerations for adaDelt

#noBN, noBNb, noBNc = dmModel(200,100,16,50,0.25,earlyStop=True,modelName='NOBN',eStopPatienc
modelPrinting(noBN,'Adadelta no Batch Normalization')



## Evaluating with Testing Data

from tensorflow.keras import models

## LOAD DATA
data = np.load('./MNIST_CorrNoise.npz') #already evaluated

def testingResults(modelIn,modelNameIn):

  x_test = data['x_test']
  y_test = data['y_test']

  num_cls = len(np.unique(y_test))
  # print('Number of classes: ' + str(num_cls))

  # RESHAPE and standarize
  x_test = np.expand_dims(x_test/255,axis=3)

  # print('Shape of x_train: '+str(x_test.shape)+'\n')

  ## Define model parameters
  model_name = modelNameIn # To compare models, you can give them different names
  pweight='./weights/weights_' + model_name  + '.hdf5'
  print(pweight)

  model = models.load_model(pweight)

  # y_pred = model.predict_class(x_test)
  predict_x=model.predict(x_test)
  y_pred=np.argmax(predict_x,axis=1)
  Acc_pred = sum(y_pred == y_test)/len(y_test)

  print('Accuracy in test set is: '+str(Acc_pred))

  return Acc_pred

# sgdAcc002 = testingResults(sgdStop,'SGD002')
#adamAcc = testingResults(adam,'Adam')
#adaDeltaAcc = testingResults(default,'Adadelta')
sgd004Acc = testingResults(sgdStop004,'SGDSTOP004')
adamResults = testingResults(adam001,'ADAM001')
sgd003Acc = testingResults(sgdStop003,'SGDSTOP003')
sgdStop005b8Acc = testingResults(sgdStop005b8,'SGDSTOP005')

defaultAcc = testingResults(default001,'DEFAULT001')
defaultNoBnAcc = testingResults(noBN,'NOBN')

# SGD vs Adam vs Default across 10 Epochs

plt.title('SGD and Adam Comparison for Validation Accuracy')
plt.plot(range(1,len(sgdStop004.history['val_accuracy'])+1),sgdStop004.history['val_accuracy'
plt.plot(range(1,len(sgdStop005b8.history['val_accuracy'])+1),sgdStop005b8.history['val_accur
plt.plot(range(1,len(adam001.history['val_accuracy'])+1),adam001.history['val_accuracy'])
plt.ylabel('Validation Accuracy')
plt.xlabel('Epoch')
plt.legend(['SGD b=4','SGD b=8','Adam'])
plt.show()
# plt.plot(range(10),default001.history['val_accuracy'][0:10])

# AdaDelta with and without Batch Normalization

plt.title('Adadelta Validation Accuracy with and without Batch Normalization')
plt.plot(range(1,len(default001.history['val_accuracy'])+1),default001.history['val_accuracy'
plt.plot(range(1,len(noBN.history['val_accuracy'])+1),noBN.history['val_accuracy'])
plt.ylabel('Validation Accuracy')
plt.xlabel('Epoch')
plt.legend(['With Batch Normlization','Without'])
plt.show()



print(adam001.history['accuracy'][-1])
print(adam001.history['val_accuracy'][-1])
print(sgdStop004.history['accuracy'][-1])
print(sgdStop004.history['val_accuracy'][-1])
print(sgdStop005b8.history['accuracy'][-1])
print(sgdStop005b8.history['val_accuracy'][-1])
print(default001.history['accuracy'][-1])
print(default001.history['val_accuracy'][-1])
print(noBN.history['accuracy'][-1])
print(noBN.history['val_accuracy'][-1])
```

✓ 0s completed at 11:19 PM ● ✕