**Arabic Handwritten Characters**

CSE 140 Project Report

Vishal Damojipurapu  
 Department of Computer Science and Engineering   
 UC Santa Cruz  
 Santa Cruz, CA, USA  
 vdamojip@ucsc.edu

Rishab Jain  
 Department of Computer Science and Engineering   
 UC Santa Cruz  
 Santa Cruz, CA, USA  
 rjain9@ucsc.edu

Sanat Sangamalli  
 Department of Computer Science and Engineering   
 UC Santa Cruz  
 Santa Cruz, CA, USA  
 ssangama@ucsc.edu

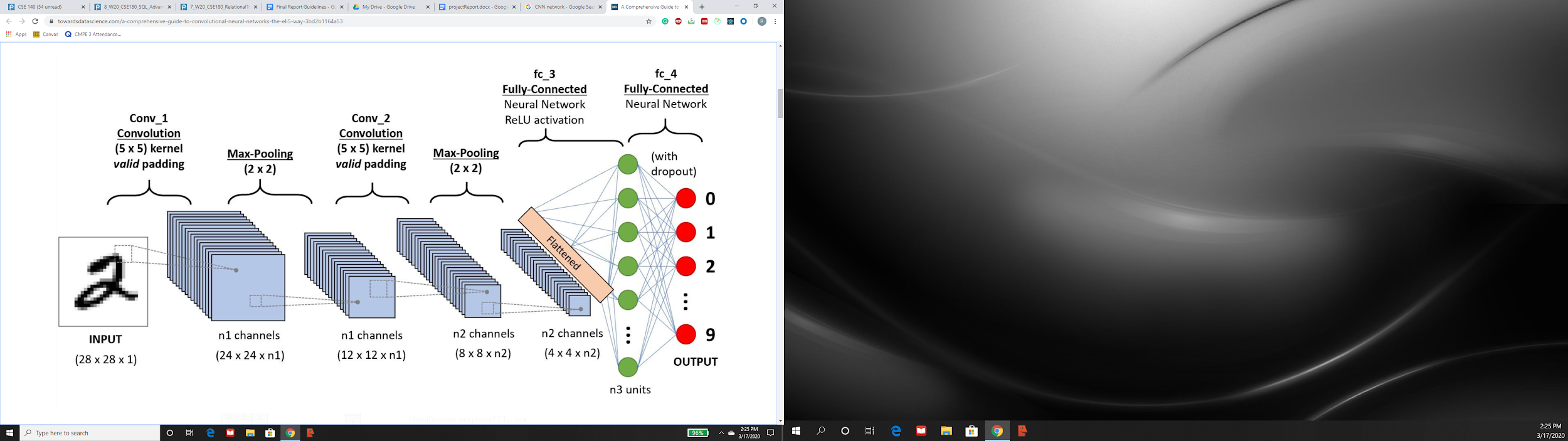
**ABSTRACT**

Our project involves Arabic handwriting recognition. Because of the practically infinite possible variations in Arabic handwriting, distinguishing and differentiating characters can prove to be quite difficult. One key issue is that there are different handwriting styles in the Arabic language. We have been given a dataset of many characters and their respective labels. In total, there are 28 different types of Arabic letters. With this data, we are going to build a classifier model that will be able to correctly distinguish these characters. We want to use a convolutional neural network (CNN) as the model because it is most commonly used in image classification. In fact, this project is very similar to digit classification, which uses the mnist database. it as per the instructions provided in previous sections. In this report we will discuss the dataset we were given, the details of our CNN network, and our results. We were able to correctly classify around 93% of the test images using our CNN. This was a good accuracy because it was similar to the Kaggle link, which had a 94% accuracy.

**MOTIVATION AND OBJECTIVE**

The motivation of this project is to create a deep learning system that can correctly identify Arabic characters. A system like this needs a large number of images which is provided to us through Kaggle. Given a dataset with different styles of handwriting, our goal was to create a neural network to classify Arabic characters with an accuracy around 94%.

Since our problem has to do with image classification and dealing with variation in human handwriting, we know to use a convolutional neural network (CNN) because it is most commonly used for image classification. Our main problem with trying to build a network like this is to correctly pre-process our data before feeding it into the model. Then our model must be deep enough and have enough parameters to correctly pick the Arabic character while dealing with varying types of handwriting styles. Other models aren’t used for image classification because a CNN network is able to learn at a better speed and accuracy when analyzing images in chunks while other models look at the image pixel by pixel which results in a lower accuracy for image recognition.

**Figure 1: An example of a CNN network for digit classification**

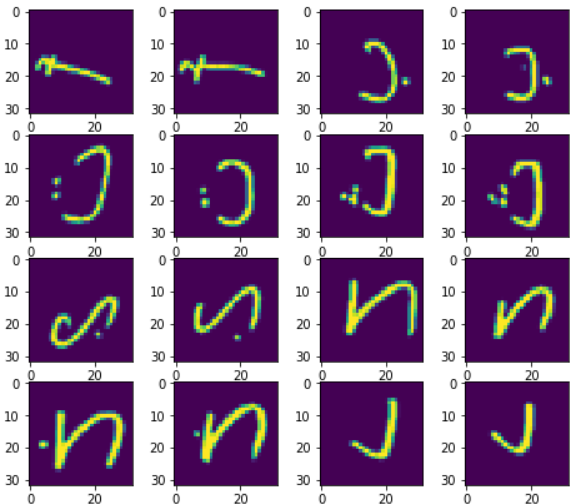
Recognizing Arabic handwritten characters is fundamentally

similar to digit classification since both are looking at grayscale images and are trying to classify them. Arabic characters are more complex because of the style of the characters and there are 28 characters to only 10 digits however fundamentally both involve pre-processing data and creating a CNN network to help correctly classify the images.

**DATASET**

This dataset was provided for use via a link to Kaggle by the professor. It is composed of 16,800 characters in Arabic by dozens of participants. The participants' age range is between 19-40 years and 90% of the participants are right-handed. These characters are written 10 times on 2 forms per participant. The blocks were then scanned at 300 dpi, and then segmented using MATLAB in order for the coordinates to be mapped. The database is partitioned into two sets: a training set (13,440 characters to 480 images per class) and a test set (3,360 characters to 120 images per class). Ordering of including writers to the test set was randomized. This was to ensure variability, as writers who come from the same institution generally tend to have similar handwriting traits and patterns. Since the data is coming from Kaggle, we didn’t need to worry about cleaning up the data too much.

For the data, we downgraded the images into grayscale format, where black is 0 and white is 255, and gray is somewhere in between. We also changed the labels of the data to make classification easier. The images training and test sets were converted to float32’s, and the labels training and test sets were converted to int32’s. When we divide by 255, the range of values from 0 to 255 from the grayscale will now be classified into either 0 and 1. We then partitioned the training data into training and validation sets. This allowed us to check if we have a higher validation set error than a training set error, which is a very textbook sign that our model is overfitting, as we have seen in class many times already. When we looked at the image folders on Kaggle, they were labeled as 32x32 images, so we did some reshaping to put the images into that format. Before we fit the model, we used an ImageDataGenerator function to augment our images so our CNN can learn from those images. Additionally, this augmentation helps to reduce overfitting.



**Figure 2: Test set data**

**MODELS AND ALGORITHMS**

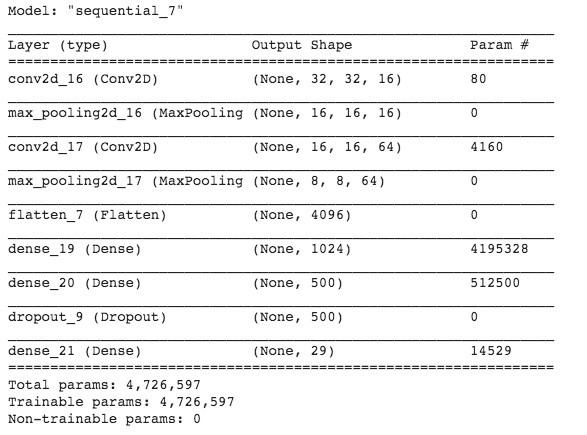
As we stated before, we will be using CNN’s, as the kaggle description said that CNN’s lead to significant improvements when it comes to classifications. We also know from the lectures that CNN’s are commonly used for image recognition. A good example we learned about was digit classification. Since our assignment is about image recognition, CNN’s are the perfect tool for the job.

We want to exploit three key details from the images we receive. We know that we don’t need to see the entire image to identify a character, the character can be written on different areas of the image, and that subsampling the image will not change the character. With that being said, we went to work on our CNN model. Our initial goal was to build a model that ran and to fine tune the parameters later on.

The first couple of models did a decent job of classification with a training accuracy around the 70% range, but this was extremely far off from our target goal of 94%. The first couple of models were also very simple CNNs with only two convolution layers, one max pool layer, one average pool layer. Then we flattened the model and had a dense layer and a layer for activation. Later, we realized we needed to build a deeper narrower network instead of a wide shallow network for two reasons. We learned from the TA’s that one popular school of thought when it comes to modeling in artificial intelligence is to start off with a very wide input layer. From there, as you traverse down the layers, you make them thinner and thinner. This allows the model to accept whatever input is needed, and then filter down in order to only get the most important features of the model. On top of that, we can learn smaller lower level features and combine them into complex features in a deep network. So that is what we decided to do. Keeping everything in mind, with the results from our trial and error phase, we were able to build a better model.

Our final CNN was a sequential model that started off with a convolution layer (16 filters, with a kernel size of 2, no padding, and a relu activation function). We then needed to either add a max pooling layer or use an alternative, average pooling. After a bit of research and trial and error, we learned that max pooling was more effective because it extracts the most important features from an image, such as edges. Average pooling considers everything equally and then just averages it, so we chose to use a max pool layer of size 2. Then, we added another convolution layer (64 filters, with a kernel size of 2, with no padding, and a relu activation function). Then, we added a max pooling layer of size 2 and flattened out the model. Here, we made it a fully connected network with a dense layer of 1024 neurons that all use relu activation. We have 1024 neurons because our image size is 32x32, which gives us one neuron for each part of the image(32\*32=1024). Relu activation is used because we have multiple layers and want to avoid having a vanishing or exploding gradient. It also has the advantage of being easy to compute. Moreover, we wanted to stick to a deeper network, so we added another dense layer with 500 neurons and relu activation to build more complex features. After considering the common issue of overfitting, we added a dropout layer of 50%. We then have a softmax layer with sigmoid activation and 28 different neurons to identify the correct character. Softmax is used because it is easier to distinguish the probability of getting a certain character when the numbers are more far apart. Also, we used 28 neurons because there are 28 different types of characters in the Arabic language. Lastly, we used sigmoid activation because this is the final output and we do not need to pass this number forward. We should have the correct character where only 1 of the 28 neurons results in a classification, which should give an output of 1.

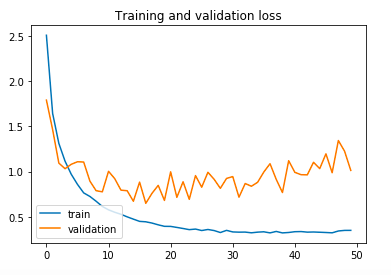
Pictured below is our model summary, which gives a physical representation of how the model is structured. You can see that there are multiple layers, each with numerous parameters. You can also tell how the model is structured to be deep and narrows down towards the ending, which was a goal of ours, as mentioned earlier.



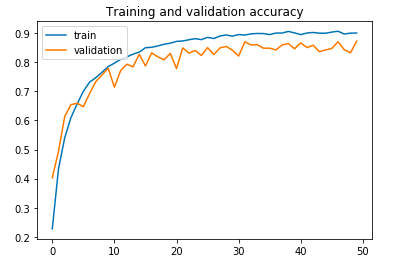
**Figure 3: CNN layer summary**

After building this model, we had to decide on what method we were going to use in order to compile it. After reading up online, we settled on using categorical cross entropy for our loss function because it was preferential over the simple squared loss function when it comes to multi-classification models. Choosing between stochastic gradient descent and RMSProp for our optimizer was also difficult, but we settled on RMSProp because we ended up getting more promising numbers for it. We later realized this was because RMSProp is helpful in dealing with both the exploding gradient problem as well as the vanishing gradient problem by normalizing the gradient. Lastly, after some more experimentation, we fit the model with a batch size of 100 and for 50 epochs. We were getting the best accuracy for these numbers, but we also thought that this was a good batch size to split our training data because it seemed the most optimal to get more updates. 50 epochs were chosen because we saw a gradual decline in loss as we increased the number of epochs, but this loss plateaued eventually. For the amount of time that running our model for 50 epochs was taking, we collectively decided that we would stop trying to improve our model just by increasing the number of epochs unless we finished and fine tuned everything else, and had nothing else to work on within the model.

**RESULTS AND ANALYSIS**



**Figure 4: Graph showing training and validation loss**

**Figure 5 : Training and Validation Accuracy**



**Figure 6: Shows Test set accuracy result**

As seen from the images above, we were able to correctly classify about 93% of the images in the test set, and 90% of the images in our training and validation sets. We are pleased with our test set accuracy, because the kaggle link stated their model was able to achieve 94% accuracy on the test set data. Therefore, we came pretty close to our initial goal that we set out. Our training data loss gradually decreased as the model underwent more and more epochs, eventually reaching a limit that roughly approached 0. However, the validation set loss started to stop decreasing around a loss value of 1.0 after about 10 epochs, and kept oscillating around that value for the remaining 40 epochs that we ran the model.

Overall, we are very pleased with our effort and the results we were able to achieve during the course of this quarter. We had spent the first week getting our environment set up, and we had to get help from our TA’s on how to get our collab notebook(the software we were using in order to run our python model code in) to work. When we finally got our notebook to properly compile our code, we found out we had to modify our images in order to do any classification training on them. In the kaggle link, all the images were in folders labeled 32x32, so we needed to resize our images into that format. In addition, there were many steps involved in that process. That is just one of the things we had to do in order to “clean up” our data. All this preprocessing took up a couple of weeks, because initially we had no idea what to do, as none of us had done any models on image classification. Our next step was to build a model that simply ran, no matter how bad the accuracy was. We got a model to run pretty quickly, however we had to improve it. When we ran our model, the accuracy was absolutely dreadful. It obtained an accuracy of about 5%. This was definitely not going to be good enough to build off of, and we each had our own ideas for the model. Ultimately, we had to completely scrap our initial model because we were not going to get very far with it. After a couple more weeks of trying out different ideas, we collectively agreed on a model that would serve as our base. We spent the last few weeks fine-tuning our model and getting it to the accuracy we wanted.

I think the main problems we faced in the beginning were the setup and preprocessing for the data given. Since we couldn’t actually even run any code on the environment we were given, we were hindered in terms of the effort we could put into our model. Those first 3 weeks roughly, were spent just trying to put everything into place so we could actually do the work. In addition, each member of our group also had several other responsibilities during the quarter, including their other classes and project work to do. Not to mention other circumstances which strongly discouraged and even outright prevented us from coming on to campus in order to get help. Overall, our group is pleased with the level or work that went into our model and what we were able to accomplish in this somewhat short period of time.

**CONTRIBUTION**

Our approach to this project was to experiment on our own when it came to building the model. This way everyone was exposed to the process of building a CNN and fine tuning the parameters. Eventually, we narrowed down to choosing a model that was giving the best accuracy and improving it.

Both Rishab and Sanat were in charge of taking care of cleaning up the data. They worked on preprocessing the data that came from kaggle. This was very difficult for us as we spent a large amount of time trying to figure out how to clean up the data. However, the folders labeled the images in a 32x32 format, so that gave us the direction we needed on how our images needed to be formatted.

Rishab also helped with the visual aspects of our project. He worked on the formatting of our poster, as well as formatting this project report, adding visuals to help justify our explanations.

In addition, Sanat spent a lot of time researching different ways to compile our model. He even focused on how to deal with overfitting. This was a problem we had to deal with around the time of the progress report.

Vishal was mostly in charge of fine tuning the parameters for the model. When we started actually testing our models, we each decided to make a copy that we would modify on our own, since working on one model at a time seemed very inefficient. With the way collab software functioned, it was simply not a practical solution. Then based on whichever model did the best, we would use theirs as the final model. We ultimately decided to use Vishal’s model as the base since his model was giving the best accuracy, and we tried to improve it from there.

We also all equally contributed to the documentation aspect of the project. Rishab mainly focused on the abstract and introduction to the project as well as the dataset preprocessing. Vishal wrote about the model, while Sanat explained the results and our conclusions. We all were able to help each other out since we took part in all aspects of the project.

**FUTURE WORK**

We are still going to continue using CNN’s, as they are ideal for image recognition and classification. As of this moment, we have found no viable alternatives for models that do image classification. Surprisingly, our test set accuracy was better than our training and validation accuracy. This is something we could look into for our future work. In addition, we could do further experimentation with increasing our number of epochs. At the time we were writing our progress report, we were only running our model for 20 epochs. At that time, we were achieving roughly a 70% accuracy on our test data. Our main goal at the time was to get the highest possible accuracy within the shortest possible time, so we wanted our model to fully train within a few minutes at most. I realize that we made several structural changes to our CNN, but part of the reason we were able to increase our accuracy so much was due to the fact that we increased the number of epochs to 50. Running the model for this amount of epochs definitely helped, but it also was very time consuming. Each run of our model takes roughly 20 minutes to run. Within the scope of this quarter in terms of time, we could not run our model that many times in order to meet all the different requirements and deadlines posed to us. However, in the future, we would have significantly more time to explore this avenue. We could probably run our model for at least 100 epochs, perhaps even more.

We found that for certain numbers of epochs greater than 50, the decline in loss eventually stagnated, particularly for the validation loss. However, we did limited experimentation, as we were trying to finalize our results, and that gave us limited time to explore this avenue. One thing we could also do to try to fix this stagnation for the validation loss is to try out a different loss function. We didn’t experiment with different ones too much as we were quite satisfied with the accuracy results we achieved using categorical cross entropy. If we invested more time in that, we could possibly find a better loss value function that would fit our model’s needs better.

**REFERENCES**

[1] Mohamed Loey. 2019. Arabic Handwritten Characters Dataset. (July 2019). Retrieved March 18, 2020 from https://www.kaggle.com/mloey1/ahcd1?fbclid=IwAR1LknswRpZZJ5UI85rN9b-EaCaqWNdVT6rOp-0mJydfkqdj1vCM0rzq3wk