Lydia Jin, Xi Qian,Yuyang Wang
DATS 6203-12
Group Final Report

Handwriting Recognition - IAM Handwriting

Introduction

This project addresses an image classification problem, and further application of the image classification explored in class. The IAM Handwriting Dataset chosen for this project is comprised of a collection of handwritten passages by various writers. The goal of this project is to use deep learning to classify the handwriting images by writing styles. The results of this project can be used in future applications, such as identifying criminals by signature in fraudulent cases. This report will discuss the dataset, network and training algorithm, experiment setup, results, and conclusions drawn from this analysis.

Dataset Description

The dataset utilized for this project is the IAM Handwriting Dataset taken from the Kaggle website. The database contains 1539 (unique case) pages of scanned text sentences written by 657 writers. The data was grouped by writer, with each writer given a set of sentences to transcribe from different passages. The sentences were taken separately and preprocessed. We only used 50 writer's handwritten images in our project. In total, this dataset contains 4899 images placed in a data_subset folder. Each image file is in png format and of different pixel size.

Network and Training Algorithm Description

This project will utilize a Convolutional Neural Network (CNN). In neural networks, convolutional neural networks one of the main categories for performing image recognition and classification. CNN image classification takes an input image, processes it and classifies it under certain categories. Technically, the CNN model passes the input images through a series of layers containing different filters, pooling layers, fully connected layers; and applies the Softmax function to classify objects based on probabilistic values ranging from 0 to 1. The following figure shows a general process of how CNN model classifies images.

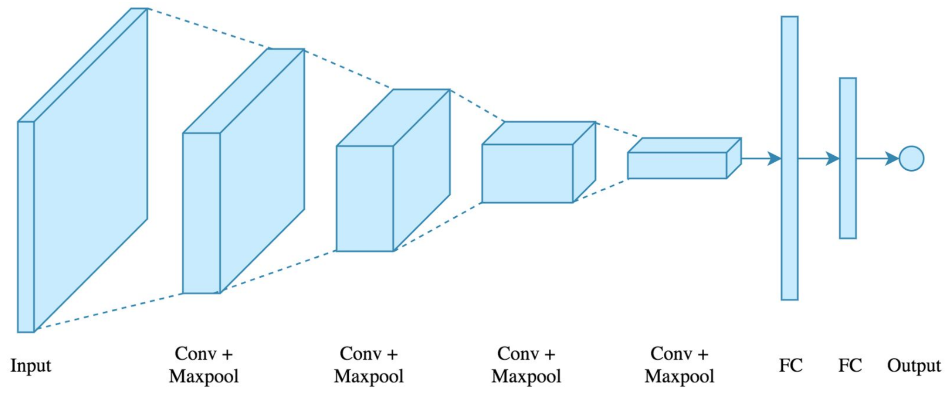


Fig1. Example of CNN architecture

In CNN the convolution operation is applied on the input data, using a convolution filter to produce a feature map. CNN has a hierarchical processing pipeline, where the features in lower layers are primitive, while those in upper layers are high-level abstract feature made from combinations of lower-level features. Therefore, more kernels are utilized in the upper layers, because they contain richer information than previous layers.

Experimental Setup

Data Preprocessing:

As mentioned above in the dataset description, the image files are of different sizes. For the images to be used as input in the Convolutional Neural Network, they must be of suitable pixel size. Therefore, the first pre-processing step was to create a target list that included the writer's number according to their handwriting scripts. Then, the target names (e.g., 000 to 0, 001 to 1) were label encoded to allow for simpler processing.

Training the Network:

The dataset was split into training, validation and testing sets, with a ratio of size 7:1.5:1.5. Next, the images were resized and cropped to receive randomly equal sized patches from each image file. After several rounds of testing, the chosen pixel size was 113*113. Parts of the code used for pre-processing were influenced by reference code found online. The generator function (see Appendix A) has multiple functions, such as scanning through each sentence and generating random patches of size 113*113. It is also used to get the pixel values of the images in the training,validation and testing sets and perform a grayscale normalization to reduce the effect of illumination's differences. Dividing the X_train images files by 255 allows the CNN to converge faster on [0..1] data compared to the original [0..255] data.

The CNN in this project was implemented with the Keras framework. Keras allows the data to be run on GPU and is the best for deep learning models with stacked layers. The analysis in this report compares models with 3 and 4 layers and varying parameters and optimizers. The images and classification are complicated, so it is necessary to try different architectures with different number of convolution layers. For judging performance, Keras also has functions such as fit_generator that can return accuracy and loss values.

Minibatches can simplify the process of updating parameters for the CNN. Large batch methods tend to converge to sharp minimizers of the loss function, leading to sensitive loss values. In contrast, small batch methods tend to converge to flat minimizers, leading to nonsensitive loss values. After experimentation with various batch sizes (e.g., 5, 10, 16, 40), a batch size of approximately 16 was found to be the best fit for this model. Similar experimentation with learning rates for the models found that the default learning rates provided for each optimizer (e.g., 0.01 for SGD, 0.001 for Adam) returned the best results.

Overfitting can be detected in some cases by looking at training, validation, and testing accuracy and loss. If the training, validation, and testing sets have very different performances, then that can be a large indication of overfitting. The training set will almost always perform better than the validation and testing sets, but a good model is expected to generalize and reduce the gap between performances.

For this analysis, dropout layers were added in our model to prevent overfitting through regularization. Dropout randomly turns off neurons, based on an assigned probability, during the training process to reduce dependency on the training set. The dropout ratio used is about 0.4-0.5, which is commonly used in CNN.

Results

The most commonly used optimizers in CNN are SGD, Adam and RMSprop. SGD has the simplest computation formula:

$$\eta_t = \alpha \cdot g_t$$

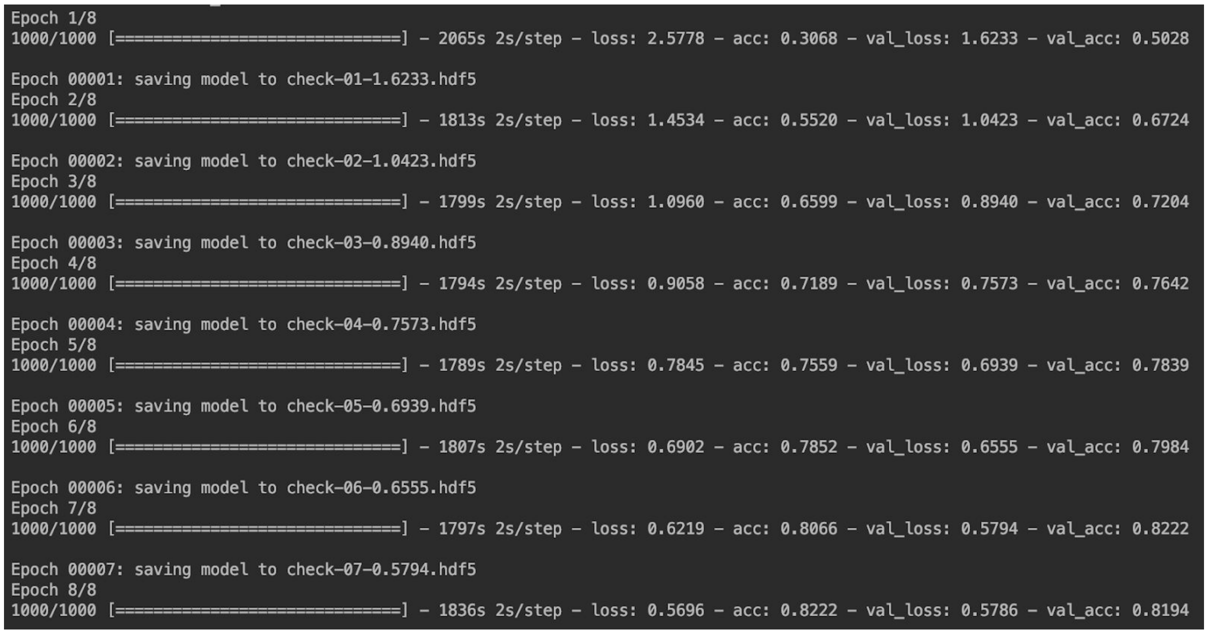
Because SGD updates frequently, sometimes significant fluctuations may occur; the model may get stuck at the local minima or saddle point, where it is difficult for SGD to escape. RMSProp optimizer is usually a good choice for recurrent neural network (RNN), and is not appropriate for use with image data. For Adam, it usually works well even with very little hyperparameter tuning. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. The update equations are shown as following:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$
$$V_t = \beta_2 \cdot V_{t-1} + (1 - \beta_2) g_t^2$$

To determine the optimizer that will return the best performance, this analysis will explore two 3-layer CNN models with the optimizers Adam and SGD. To make comparisons on the effects of the two optimizers, the number of convolution layers will be controlled at 3, and the same values for the various parameters (e.g., number of filters, kernel size) will be used.

This analysis found the 3-layer CNN with Adam optimizer to be the best model. Using the Keras framework function, fit_generator (Appendix B), the output returns a History object with records of training and validation accuracy and loss values at successive epochs. The output for this model is shown below in Figure 2.

Also shown below as Figure 3 is the subplot displaying a visualization of the validation accuracies for each of the 4 models. There is a clear increase in accuracy for the models with the Adam optimizer as the epochs increase. In contrast, the SGD optimizer model remains at a constant 20-30% accuracy regardless of the number of epochs. For the image classification of this IAM dataset, the Adam optimizer is the clear choice for better classification accuracy for a 3 or 4-layer CNN.



```
Epoch 1/8
1000/1000 [==============================] - 2065s 2s/step - loss: 2.5778 - acc: 0.3068 - val_loss: 1.6233 - val_acc: 0.5028

Epoch 00001: saving model to check-01-1.6233.hdf5
Epoch 2/8
1000/1000 [==============================] - 1813s 2s/step - loss: 1.4534 - acc: 0.5520 - val_loss: 1.0423 - val_acc: 0.6724

Epoch 00002: saving model to check-02-1.0423.hdf5
Epoch 3/8
1000/1000 [==============================] - 1799s 2s/step - loss: 1.0960 - acc: 0.6599 - val_loss: 0.8940 - val_acc: 0.7204

Epoch 00003: saving model to check-03-0.8940.hdf5
Epoch 4/8
1000/1000 [==============================] - 1794s 2s/step - loss: 0.9058 - acc: 0.7189 - val_loss: 0.7573 - val_acc: 0.7642

Epoch 00004: saving model to check-04-0.7573.hdf5
Epoch 5/8
1000/1000 [==============================] - 1789s 2s/step - loss: 0.7845 - acc: 0.7559 - val_loss: 0.6939 - val_acc: 0.7839

Epoch 00005: saving model to check-05-0.6939.hdf5
Epoch 6/8
1000/1000 [==============================] - 1807s 2s/step - loss: 0.6902 - acc: 0.7852 - val_loss: 0.6555 - val_acc: 0.7984

Epoch 00006: saving model to check-06-0.6555.hdf5
Epoch 7/8
1000/1000 [==============================] - 1797s 2s/step - loss: 0.6219 - acc: 0.8066 - val_loss: 0.5794 - val_acc: 0.8222

Epoch 00007: saving model to check-07-0.5794.hdf5
Epoch 8/8
1000/1000 [==============================] - 1836s 2s/step - loss: 0.5696 - acc: 0.8222 - val_loss: 0.5786 - val_acc: 0.8194
```

Fig2: Loss and accuracy for 3-layer CNN model with Adam Optimizer

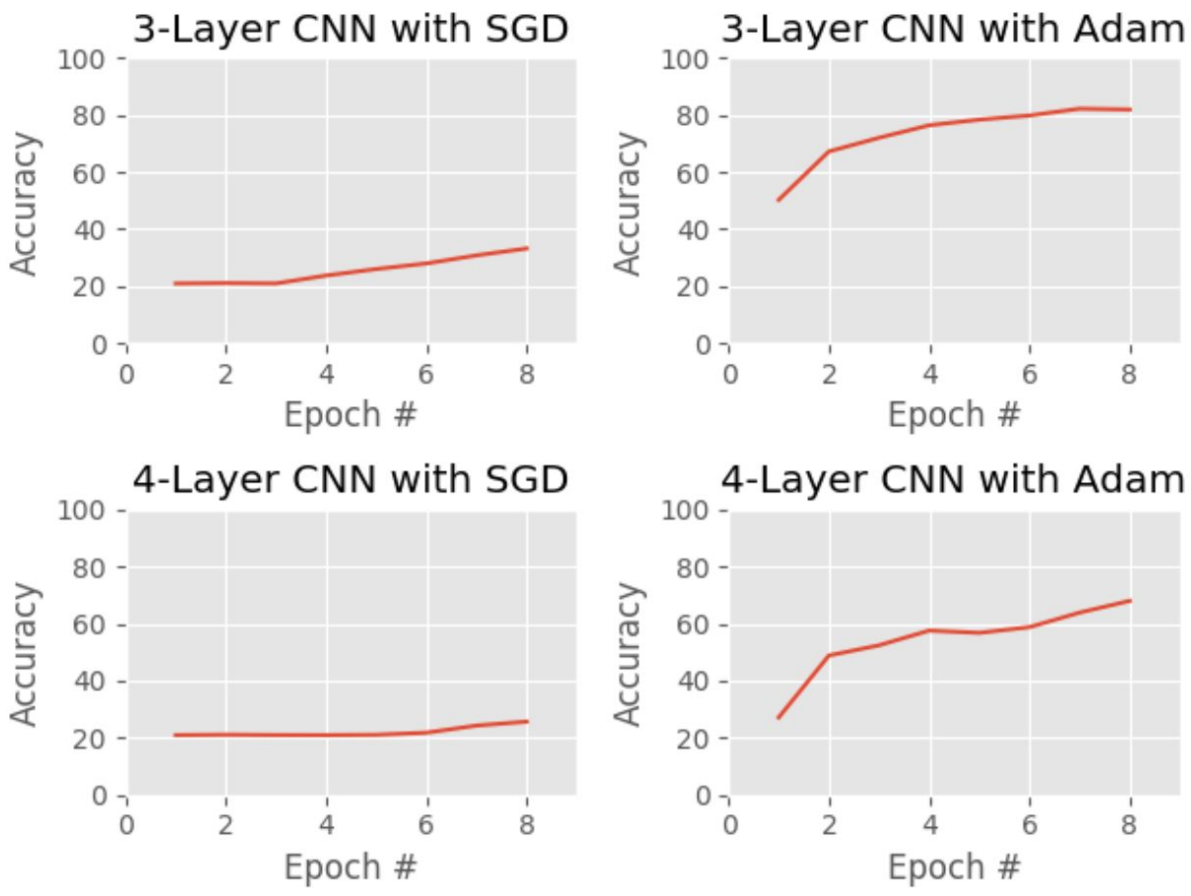
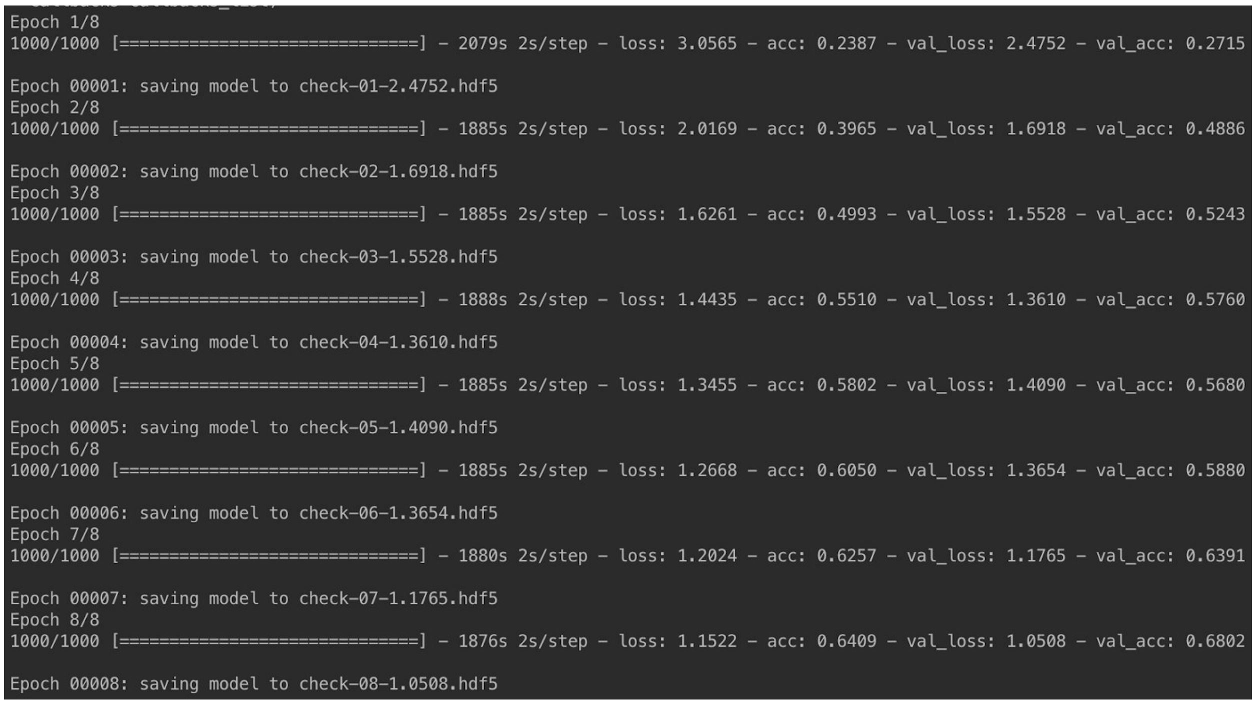


Fig3: Validation Accuracy for 4 models of interest

Figure 4 displays the output after training the 4-layer CNN model with Adam optimizer. This output is useful for comparing the differences that adding another convolution layer can make. The overall validation accuracy has dropped and the overall validation loss has increased compared to the 3-layer model.



```
Epoch 1/8
1000/1000 [==============================] - 2079s 2s/step - loss: 3.0565 - acc: 0.2387 - val_loss: 2.4752 - val_acc: 0.2715

Epoch 00001: saving model to check-01-2.4752.hdf5
Epoch 2/8
1000/1000 [==============================] - 1885s 2s/step - loss: 2.0169 - acc: 0.3965 - val_loss: 1.6918 - val_acc: 0.4886

Epoch 00002: saving model to check-02-1.6918.hdf5
Epoch 3/8
1000/1000 [==============================] - 1885s 2s/step - loss: 1.6261 - acc: 0.4993 - val_loss: 1.5528 - val_acc: 0.5243

Epoch 00003: saving model to check-03-1.5528.hdf5
Epoch 4/8
1000/1000 [==============================] - 1888s 2s/step - loss: 1.4435 - acc: 0.5510 - val_loss: 1.3610 - val_acc: 0.5760

Epoch 00004: saving model to check-04-1.3610.hdf5
Epoch 5/8
1000/1000 [==============================] - 1885s 2s/step - loss: 1.3455 - acc: 0.5802 - val_loss: 1.4090 - val_acc: 0.5680

Epoch 00005: saving model to check-05-1.4090.hdf5
Epoch 6/8
1000/1000 [==============================] - 1885s 2s/step - loss: 1.2668 - acc: 0.6050 - val_loss: 1.3654 - val_acc: 0.5880

Epoch 00006: saving model to check-06-1.3654.hdf5
Epoch 7/8
1000/1000 [==============================] - 1880s 2s/step - loss: 1.2024 - acc: 0.6257 - val_loss: 1.1765 - val_acc: 0.6391

Epoch 00007: saving model to check-07-1.1765.hdf5
Epoch 8/8
1000/1000 [==============================] - 1876s 2s/step - loss: 1.1522 - acc: 0.6409 - val_loss: 1.0508 - val_acc: 0.6802

Epoch 00008: saving model to check-08-1.0508.hdf5
```

Fig4: Loss and accuracy for 4-layer CNN model with Adam Optimizer

The output has various lines stating "saving model to check…". This is a result of the filepaths created that are necessary for the Keras ModelCheckpoint function (Appendix B). The main focus of this output are the validation accuracies, of which we see a validation accuracy of approximately 68% in the 8th epoch.

Summary and Conclusions

This analysis found that for this type of handwriting recognition problem, the Adam optimizer with default parameters is the best fit for configuring the model for training. Likewise, the analysis returns that a 3-layer convolution model is more accurate than a 4-layer convolution model. This may result from the sizes of the images used for this problem. This analysis utilizes a generator function that resizes and crops the image files to pixel size of 113x113. For datasets with larger or smaller pixel sized images, the model should be updated with more or less convolution layers as needed.

The process of building convolution models and compiling various ideas and conclusions into this report allowed for better overall understanding of CNN and the Keras framework. CNN is a very flexible algorithm and Keras provides multiple methods of manipulating layers efficiently. More importantly, this process allowed for a greater understanding of how to connect the various aspects of neural network architecture and applications learned in class.

For improving results, it may prove useful to run the models on a large number of epochs and number of samples per epoch. Further research may consider performing handwriting recognition for the top 10 classes, instead of the top 50 used in this report. This would allow for exploring more comprehensive visualizations, such as confusion matrixes or visualization of gradients for each layer. Also, the dataset should contain more sentences written by writers. Currently, there are only a few sentences written for each writer, and there will be some loss of accuracy this way.

References

Dataset: https://www.kaggle.com/tejasreddy/iam-handwriting-top50
https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/
Figure 1:
https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2
Checkpoint code:
https://machinelearningmastery.com/check-point-deep-learning-models-keras/
Generator code:
https://www.kaggle.com/tejasreddy/offline-handwriting-recognition-cnn/notebook
CNN Background Information:
https://www.learnopencv.com/image-classification-using-convolutional-neural-networks-in-keras/
Save/Load Keras models: http://faroit.com/keras-docs/2.0.2/models/about-keras-models/
Johnson, W. A., Faieta, B. A., Jellinek, H. D., & Smith III, Z. E. (1999). U.S. Patent No. 5,991,469. Washington, DC: U.S. Patent and Trademark Office.
Girija, S. S. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems.

Appendix

Appendix A:

```
def generate_data(samples, target_files, batch_size=batch_size, factor=0.1):
    num_samples = len(samples)
    from sklearn.utils import shuffle
    while 1:  # Loop forever so the generator never terminates
        for offset in range(0, num_samples, batch_size):
            batch_samples = samples[offset:offset + batch_size]
            batch_targets = target_files[offset:offset + batch_size]

            images = []
            targets = []
            for i in range(len(batch_samples)):
```

```
        batch_sample = batch_samples[i]
        batch_target = batch_targets[i]
        im = Image.open(batch_sample)
        cur_width = im.size[0]
        cur_height = im.size[1]

        height_fac = 113 / cur_height
        new_width = int(cur_width * height_fac)
        size = new_width, 113

        imresize = im.resize((size), Image.ANTIALIAS)
        now_width = imresize.size[0]

        avail_x_points = list(range(0, now_width - 113))
        pick_num = int(len(avail_x_points) * factor)
        random_startx = sample(avail_x_points, pick_num)

        for start in random_startx:
            imcrop = imresize.crop((start, 0, start + 113, 113))
            images.append(np.asarray(imcrop))
            targets.append(batch_target)

    X_train = np.array(images)
    y_train = np.array(targets)

    X_train = X_train.reshape(X_train.shape[0], 113, 113, 1)
    X_train = X_train.astype('float32')
    X_train /= 255

    #convert a class vector to binary class matric
    y_train = to_categorical(y_train, num_classes)
    yield shuffle(X_train, y_train)
```

Appendix B:

```
from keras.callbacks import ModelCheckpoint
filepath = "check-{epoch:02d}-{val_loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath=filepath, verbose=1, save_best_only=False)
callbacks_list = [checkpoint]

history_object = model.fit_generator(train_generator, samples_per_epoch=samples_per_epoch,
                        validation_data=validation_generator,
                        nb_val_samples=nb_val_samples, nb_epoch=nb_epoch, verbose=1,
                        callbacks=callbacks_list)
```