Letter Recognition with Machine Learning

Jc01663 – COM2028 Artificial Intelligence Coursework

Letter Recognition:

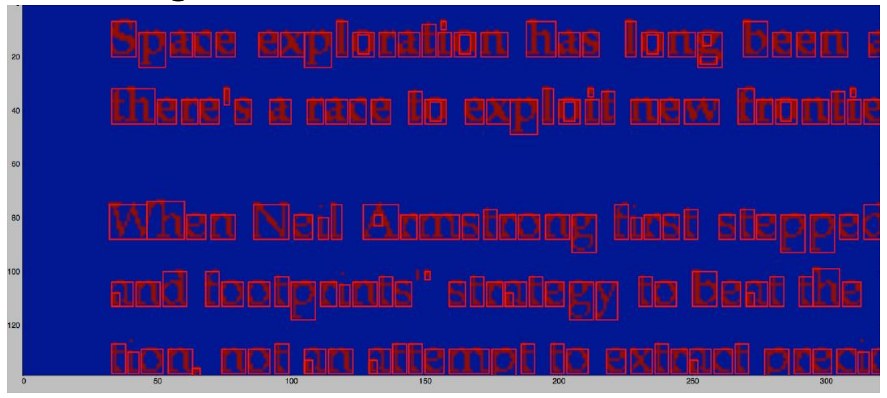
From our youngest age we are taught how to associate names and to the object we see in our world. Our process of learning which led us to where we are now all started by recognising letters and what sound they make as they're pronounced, and how to make meaningful words out of them.

Recently, machine learning has been progressing and evolving at a very impressive rate and it is already able achieve many different challenges and tasks either on par or even better than the average human being. It is excelling in the fields of image processing, classification, prediction and association.

Due to that, many Handwriting recognition (HWR) system were developed and are being used for commercial purposes. Letter/Character recognition acts as a key function in the translation of handwritten text to data that is understood and that be used by a machine learning application.

Our goal is to implement a functional system that is able to recognise input images of letters. Before we start talking about which machine learning technique we are using, we need to understand the pre-processing that needs to be done to the input data before it can be used to train our classification model.

Since the input we want to use is an image, we need to extract each character contained in the image. This is done by using the Extract_Letters class defined in the attached source code. The extractFile() function takes an image uri, seperates it into different regions based on pixel and neighbouring pixel values. Each region is then associated a bounding box, which defines a rectangular border around the region.

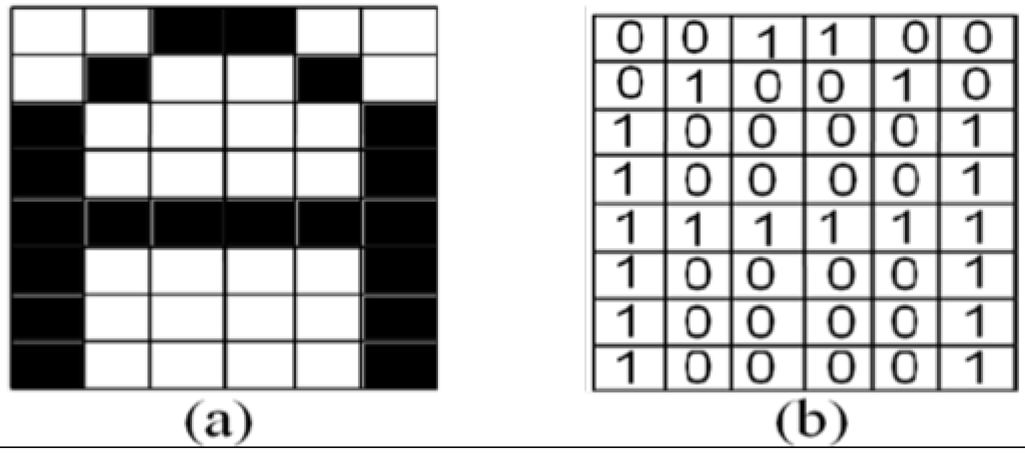


Space exploration has long been
there's a race to exploit new frontie

When Neil Armstrong first steppe
and footprints' strategy to beat the
tion, not an attempt to extract preci

The different regions in the input image represent the letters extracted. These are then sorted and returned as a list, ready to be used for training.

What we see is simply an image with boxed around letters, but for the underlying system this is treated as a matrix of the pixel values contained in the image. In a black in white image, the darker the pixel is the higher its value is, white being 0 and black being 1; as shown in the figure below, where (a) is an input image and (b) is the matrix of pixel values.



0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
1	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	0	1
1	0	0	0	0	1

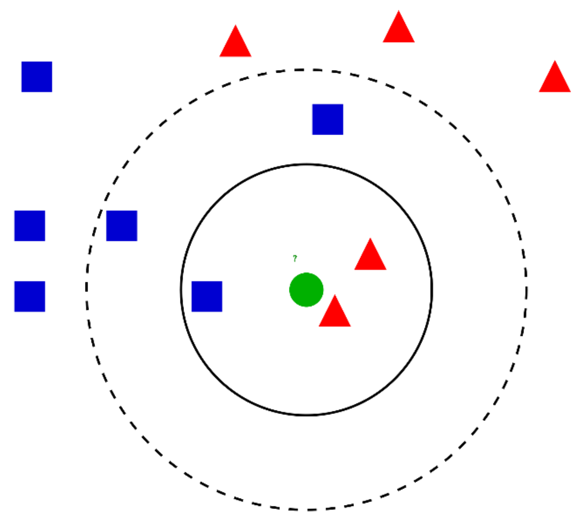
Classification techniques:

Now that we have seen what the input data consists and it is processed, we can look at how they are grouped into different clusters/classes using various classification techniques as well as their benefits and drawbacks.
Classification is achieved through the use of feature extraction and performing pattern recognition on extracted features to find similarities between each input data. The feature extraction, in our case, is done as explained in the first section; the feature vector is made by creating a matrix of pixel values for each letter extracted.

The following are widely use techniques used for classification in handwriting recognition:

- K-Nearest Neighbours (KNN)

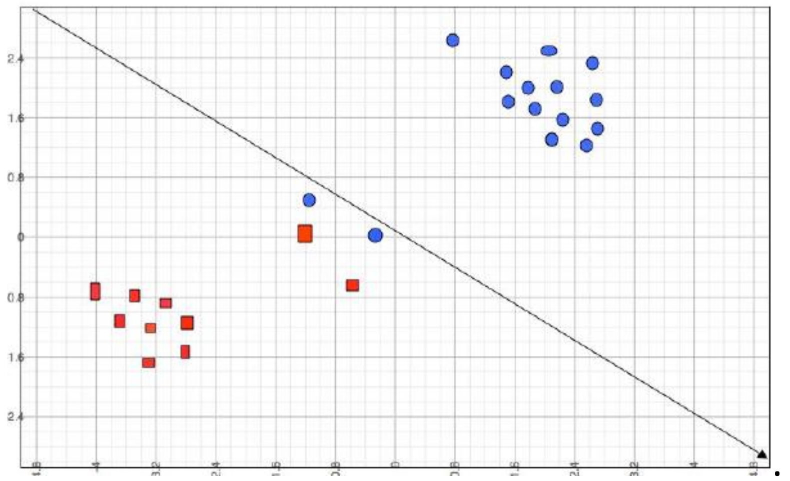
  In KNN, an object is classified by considering the k-nearest neighbour vectors and returning the class with the majority of objects whose vectors belong to.
  A vector is chosen as a neighbour if it is one of the top k vectors to have similarities with the input vector. This is calculated by using a distance function such as Euclidean distance or cosine similarity.



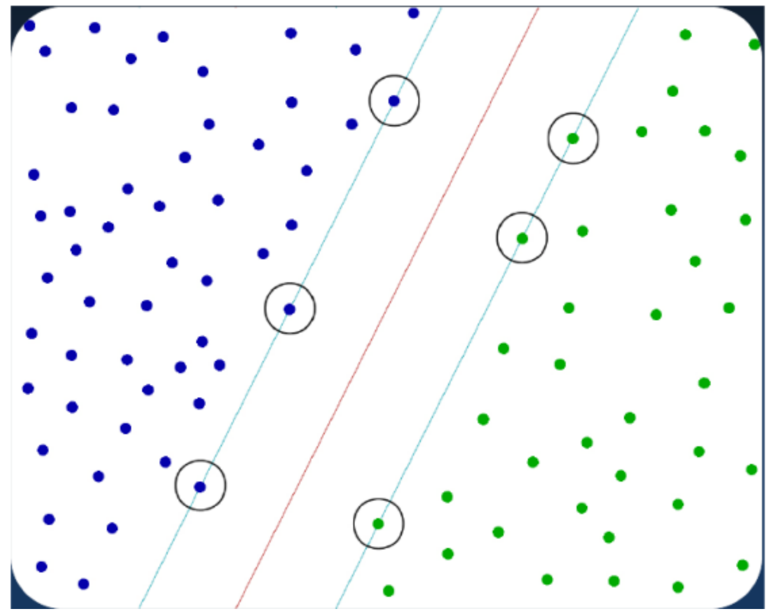
  As we can see in the figure above, if k = 3, the green input vector has 2 neighbours associated with red triangle class and 1 neighbour associated with blue, so it will be classified as red triangle is square class. But if k = 5, the input vector has 2 red triangle neighbours and 3 blue squares neighbours, so it will be classified as blue square.
  KNN requires no training time as it makes predictions from the training data whilst learning from it. A new data input will be classified but will also possible impact the next classifying decision. It is also very easy to implement. However, KNN doesn't perform well against large datasets or data that requires a high dimensional vector as it takes too much time to calculate the distance/similarity to every existing vector. KNN is also sensible to random noise and hence is might overfit our data.

- Support-Vector Machines (SVM)

  In order to determine to which class an input object belongs to, we take the dot-product distance of the object's vector and the average vector from the different classes. The dot product with the lowest value means that the object should be associated with that class. A dividing line gets calculated by the midpoint between the average point of two classes. However, this may lead to vectors being misclassified, as shown below



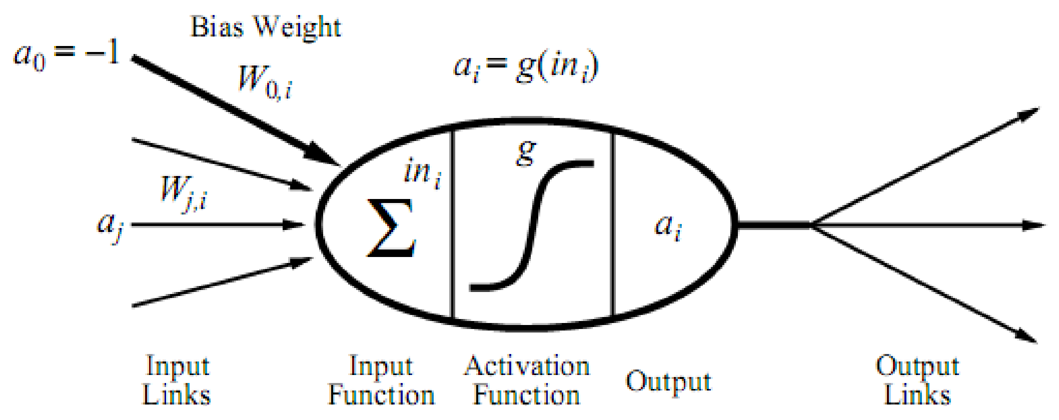
  SVM is a method used to maximise the margin between the points associated to a class and the diving line so that we reduce misclassification. The dividing line is chosen so that the parallel lines that touch the items from each class are as far as possible from it.



  Classification based on dot-product is a basic linear classifier, which just finds a diving line between different classes. This means that If the data cannot be linearly separable, the classifier will underfit the data and make so incorrect predictions.
  This can be overcome by using the kernel trick. A technique that involves replacing the dot-product function with a new function that returns what the dot-product would have returned if the data had first been transformed to a higher dimensional space using some mapping function.

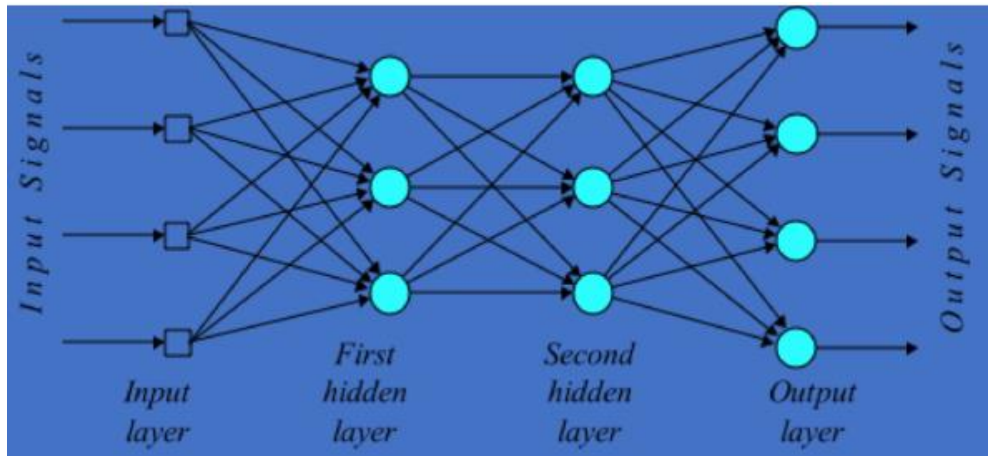
- MultiLayer Perceptron (MLP)

  The MLP architecture consists of multiple layers of perceptrons. Each perceptron can be used as a linear classifier due to its activation function, a cost function that triggers if cost is low enough and doesn't if cost is too high. Perceptron are connected using input and output links that carry different weights and that allows the input data to be processed by each perceptron until we reach an output value.



  A perceptron learns by recursively making small adjustments in the weights to reduce the difference between the actual and desired outputs of the perceptron.

  By having network of perceptron (neural network) we can classify non-linear data. A traditional neural network consists of an input layer, which introduces input values into the network. At least one hidden layer, which performs classification of features. And an output layer, which functions like a hidden layer but passes the outputs to the outside world.

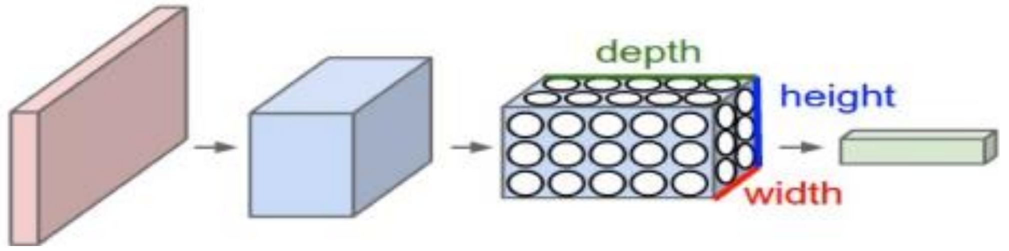


A neural network learns through the use of back-propagation. Which can be described in a few words as comparing the predicted output with the desired output and feeding an error signal back through the network so that each perceptron can adjusts its weights accordingly.

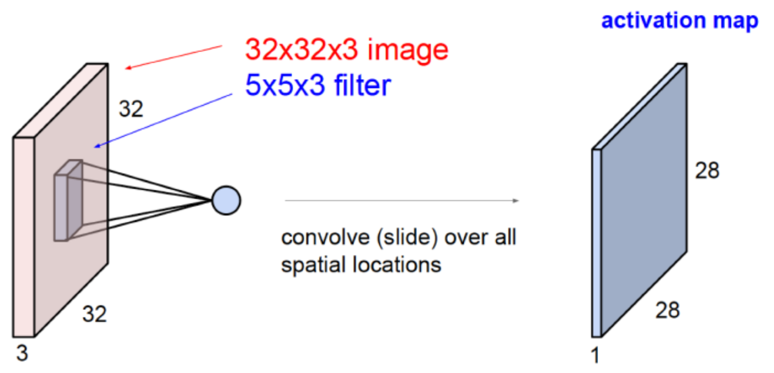
- Convolutional Neural Network(CNNs)

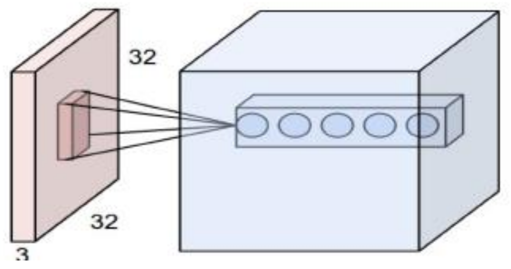
  Similar to ordinary neural networks due to having connected neurons with learnable weights and activation function. Since a fully connected network doesn't scale well for large input size, CNN uses convolutional layers to reduce the size of the input data.

  A Layer of ConvNet has neurons arranged in 3 dimensions: width, height and depth. Neurons in a layer will only be connected to a small region of the previous layer. Each layer of ConvNet transforms the original image layer by layer from the original input values to final class scores



  The transformation from a layer to another is done by convolving a filter across the input image and taking the dot product for each region with the filter, creating activation maps which are then stacked to make a new input for the next layer.



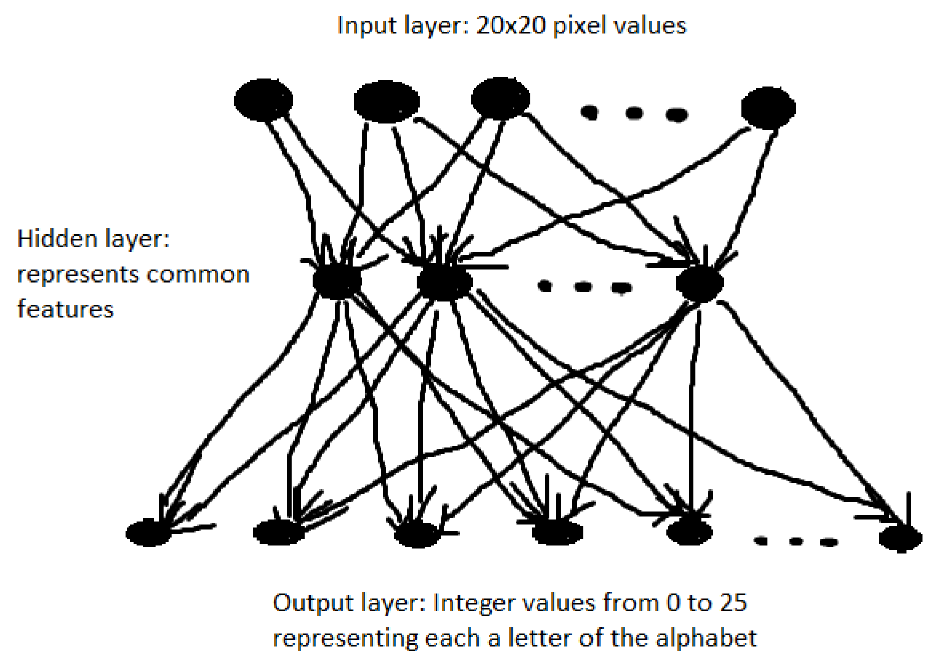


  CNN are often accompanied by other layers, like pooling which is used to down sample the input data, or activation functions like ReLu.

Once the convolutional layers are finished, we can use a fully connected layer to compute the class scores and return a prediction.

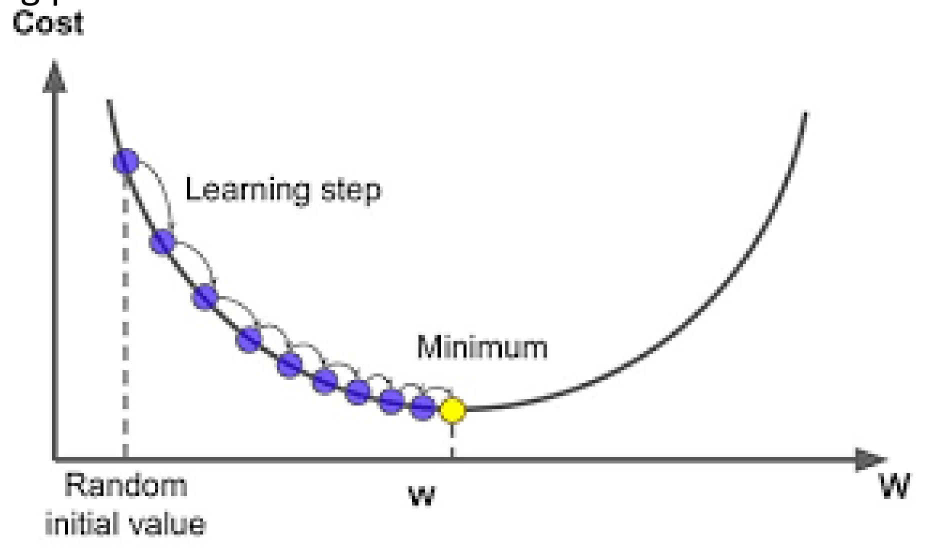
Implementation:

I used the TensorFlow library in order to build my neural network model easily. First, we need configure the layers of our model.
In the implementation we will look at, a common multilayer perceptron has been used. It is made of an input layer of size 20*20, because every extracted letter is resized to fit into a 20 by 20 pixel bounding box. One hidden layer of size 100 as it shouldn't be more than the input layer or less than the output layer. And an output layer of size 26 as each possible output should represent a character from the English alphabet.


Input layer: 20x20 pixel values
Hidden layer: represents common features
Output layer: Integer values from 0 to 25 representing each a letter of the alphabet

Before the model is ready, we also need to define the loss function, the optimizer and how we will evaluate it.
The loss function measures the error margin for each training data. This is done using SoftMax cross entropy function which penalizes models that estimate a low probability for the target class. The optimizer's role is to gradually modify the weights of the network to minimise the loss function. A gradient descent technique is often used to calculate how to best reduce any loss. Finally, we will evaluate the model by taking an accuracy reading of the valid predictions make against the invalid ones.

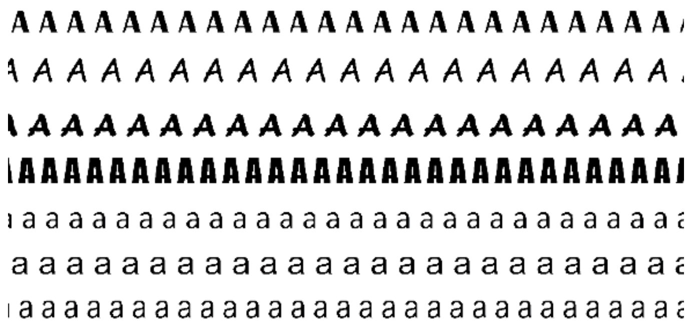
Now that the model is ready, we can start using our extracted letters to train it. There are 3 main gradient descent techniques used to train a model. Stochastic gradient descent calculates the error and updates the model for each piece of training data. Batch gradient descent calculates the error for each piece of training data but only updates the model after all of them have been processed. We are using Minibatch gradient descent, which is combines the best of both techniques. It splits the training set into a number of batches and updates the model for each batch being processed.



Our data is divided into a few different sections. X_train which contains most of the extracted data and will be used to train the model. y_train, which consist of the targets/labels from the corresponding data in X_train. X_valid is used to give an estimate of the model's predictions. These predictions are evaluated against the y_valid data which holds the target data of X_valid. When we get to the testing phase, new data should be used than the one from the training set, these are stored into X_test, and are used by the model to make predictions.

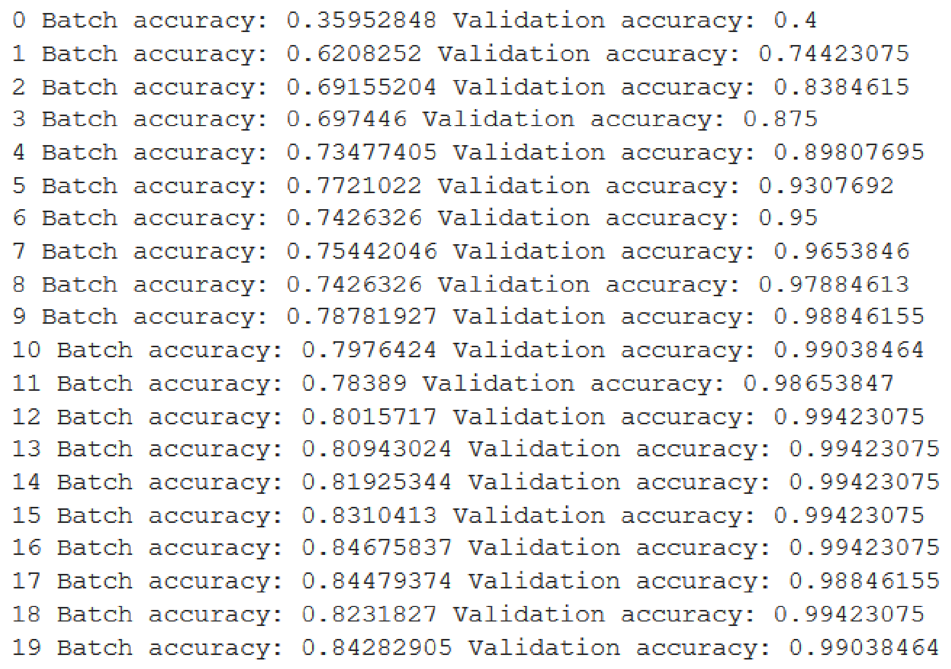
Evaluation:

The training data used is a collection of png files, where each file is filled with the same letter in different fonts and in both uppercase and lowercase. Below is a small part of the training data.



In total, the model was trained using 18354 letters (17834 for training and 520 for validation) extracted from the 26 files.

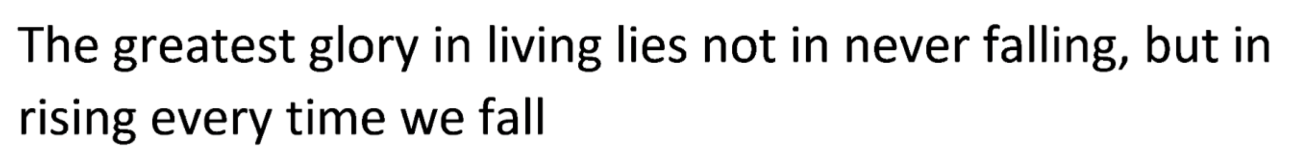
By using a batch size of 500 and a training phase of 20 epochs, the training using minibatch gradient descent seems to work well.



```
0 Batch accuracy: 0.35952848 Validation accuracy: 0.4
1 Batch accuracy: 0.6208252 Validation accuracy: 0.74423075
2 Batch accuracy: 0.69155204 Validation accuracy: 0.8384615
3 Batch accuracy: 0.697446 Validation accuracy: 0.875
4 Batch accuracy: 0.73477405 Validation accuracy: 0.89807695
5 Batch accuracy: 0.7721022 Validation accuracy: 0.9307692
6 Batch accuracy: 0.7426326 Validation accuracy: 0.95
7 Batch accuracy: 0.75442046 Validation accuracy: 0.9653846
8 Batch accuracy: 0.7426326 Validation accuracy: 0.97884613
9 Batch accuracy: 0.78781927 Validation accuracy: 0.98846155
10 Batch accuracy: 0.7976424 Validation accuracy: 0.99038464
11 Batch accuracy: 0.78389 Validation accuracy: 0.98653847
12 Batch accuracy: 0.8015717 Validation accuracy: 0.99423075
13 Batch accuracy: 0.80943024 Validation accuracy: 0.99423075
14 Batch accuracy: 0.81925344 Validation accuracy: 0.99423075
15 Batch accuracy: 0.8310413 Validation accuracy: 0.99423075
16 Batch accuracy: 0.84675837 Validation accuracy: 0.99423075
17 Batch accuracy: 0.84479374 Validation accuracy: 0.98846155
18 Batch accuracy: 0.8231827 Validation accuracy: 0.99423075
19 Batch accuracy: 0.84282905 Validation accuracy: 0.99038464
```

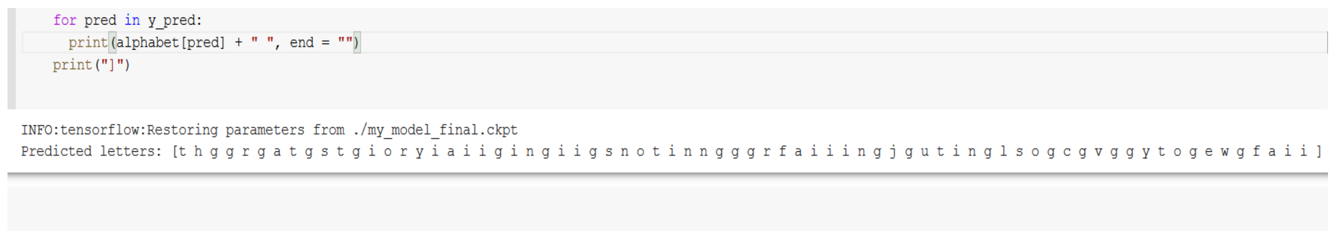
As we can see in the screenshot above, the model starts with a low batch and validation accuracy which is to be expected since it hasn't been properly trained yet. The more batches we train the model with, the more the model increases its accuracy. With batch accuracy reaching up to 84% and validation accuracy reaching up to 99%.

Let's now see how our trained model does when it faces new test data. The test file I used for this run contains only 72 characters, but this is enough to give us an idea of how well the model performs on new data.



The greatest glory in living lies not in never falling, but in rising every time we fall

If we compare the predicted letters with the letters actually being inputted into the system, we notice that a lot of letters have been wrongly predicted. We cannot really tell what the predicted set of letters are supposed to form a readable English sentence.



```
for pred in y_pred:
  print(alphabet[pred] + " ", end = "")
print("]")
```

```
INFO:tensorflow:Restoring parameters from ./my_model_final.ckpt
Predicted letters: [t h g g r g a t g s t g i o r y i a i i g i n g i i g s n o t i n n g g g r f a i i i n g j g u t i n g l s o g c g v g g y t o g e w g f a i i ]
```

The rate of correctly predicted letters against incorrect ones is about 61% and 39% respectively, which is quite a poor accuracy rate.

I believe this is due to the following reasons:
It seems like the model tried to overfit the training data too closely and ended up not fitting other data that could be slightly different.
Even though I tried to make the training data as diverse as possible, the test data still differs from it. In the training data, letters are nicely spaced out in a line. Whereas, in the testing data, letters are put into a sentence which means letters are in the same words will be much closer to each other. This may have affected the extracted result of each letter and hence impact the prediction of the model.
Perhaps using a multilayer perceptron is not needed for this problem, as it is a lot more complex than it needs to be. Too many neuron connections mean that a model can learn data that has a high dimensional vector, which is not the case here. Using the simpler SVM or KNN techniques could have resulted in a better model that didn't overfit the training data and gave better predictions for new data.