Handwritten Character Recognition Using SVM, NN and CNN

By Jason Cho(bc454), Hannah Lee(hl838)

Link to GitHub repo : https://github.coecis.cornell.edu/hl838/MNIST

Introduction

Our goal for this project was to develop an image recognition AI using various supervised learning models. We initially tried to come up with a model that would distinguish different types of pizzas. (as we were craving for a pizza when we were writing the proposal). We were planning on using Convolutional Neural Network (CNN) a popular deep learning technique for image classification tasks to develop our pizza classification model.

However, we came upon a problem right off the bet: preprocessing. To train the model, we needed a good amount of data, in this case images of pizzas. Our strategy was to scrape images of pizzas through web scraping using google or Instagram api, resize the image into certain resolutions, vectorize the image, and finally use those data to train our model. The problem with using Instagram or google images were that many of them included images unrelated to what we are searching. For example, when we searched for “sausage pizzas” on Instagram searching api, a lot of images included people drinking beer, watching televisions, and pictures of their wife or husband etc. To make sure that our dataset only contains relevant data, we decided to label the data manually. However, we realized that even the ones that were picture of pizzas had wide variations within the same categories due to various lighting and angles as you can see above. We realized that we would need extensive amount of time just to preprocess the images, so we decided to develop a image recognition model using more widely used and popular data. We were able to easily obtain large sets of handwritten image of alphabets from EMNIST. Models that we covered in this project include Support Vector Machine model, Feed-forward neural network, and Convolutional Neural Network.

Datasets

As we experienced difficulties with collecting an appropriate dataset to train our pizza classifier, we realized that data collection could easily be one of the most challenging part of any real-world machine learning task. For the sake of our project, we decided to opt for an already existing dataset that is already 1) sufficiently large, 2) properly labeled and 3) normalized, so that we can focus more on experimenting with different machine learning algorithms.

As a result, we used a dataset provided by extended MNIST for both handwritten uppercase and lowercase English letter alphabets (A - Z). Each image is 28 by 28 pixels. We used the vectorized and normalized the image. Each row has total of 28*28 (784) features. We

had two sets of data, one for testing and one for training the models. The figures below represent recreated images of the datasets.

Overfitting

As extensively covered in class, overfitting is arguably the most common pitfall in machine learning. We tried to capture overfitting by separating data sets into three different categories, **training set, validation set** and **test set.** Training set is dataset that is used to train the model. For the case of training a neural network, we would use training set to find the most optimal weights for the backpropagation.Validation set is dataset that is not used for training, but it would be used to tune our model. Based on the performance of our model against the validation set, we would decide the number of epochs, number of nodes in each layer, number of layers, number of the training set. By having a validation set that is independent to the training set, we could prevent the model from overfitting the training set. As part of the cross validation method, we used holdout method. Holdout method separates the training set into two parts, one for training set and another one for validation set. We used last 10 percent of the training set as the validation set. This would be a problem if our training data set is not randomly spread out throughout the training dataset, (ex: alphabet 'A' is the first 800 data points and so on). This was not the case for our dataset so we used the holdout method. Testing set is the data set to estimate the accuracy of the model overall. Testing set was separately provided by EMNIST as a separate file. This test set was only used once per model to assess the accuracy of the model after all the fine tuning was finished based on the results from validation and training set. Once our model was assessed with the testing set, we did not improve our model any further.

SVM(Support Vector Machine)

General Structure

A Support Vector Machine is an algorithm that is mainly used for a classification problem. The algorithm constructs a set of hyper-planes that best distinguish different classes. Support Vector Machine is a kernel method. A kernel function measures the distance between two observations. We tried different kernel functions provided by scikit package in python to train our model.

Advantages

According to the documentation of Support Vector Machine, the algorithm is effective in high dimensional spaces and it utilizes so called "support vector" a subset of training datasets

which makes this algorithm memory efficient as well as a strong algorithm against overfitting. The algorithm also allows us to use different kernel functions including linear, polynomial, radial basis function (RBF) and etc; the algorithm also allows for custom kernel function which provides flexibility in using the algorithm.

Disadvantages

One of the difficulties in using the Support Vector Machine that we encountered was choosing the right kernel function. Because algorithm is sensitive to the kernel function, finding the most effective and "appropriate" kernel function for this particular problem was difficult. We tried different kernel functions to see how well our model perform depending on the kernel functions.

Choosing parameters - kernel function

```
# train data with SVM rbf
from sklearn.model_selection import cross_val_score
clf = svm.SVC(kernel = 'rbf')
clf.fit(data["images"], data["labels"])

# predict against the validation set and training sets
predict_train = clf.predict(data["images"])
scores = cross_val_score(clf, data["images"], data["labels"], cv=5)
print(scores)
```

We tested three different kernel functions: rbf (radial basis function), linear and sigmoid functions.We initially trained our model using 10,001 data points. We used different kernel functions including radial basis function, linear, and sigmoid to see which kernel function bests fits our datasets. We fitted our model using the functions mentioned above and calculated the accuracy against the validation sets. We calculated the accuracy over 5 times and took the average of the accuracy. For the **rbf**, we obtained **68 percent** of accuracy, with **sigmoid** as the kernel function. obtained **63 percent**, for **linear function** we obtained **71** percent of accuracy. At first we thought that linear function would be the best kernel function.

accuracy score = 0.985835694051
report =

	precision	recall	f1-score	support
1	1.00	1.00	1.00	235
2	1.00	1.00	1.00	216
3	1.00	1.00	1.00	246
4	1.00	1.00	1.00	230
5	1.00	1.00	1.00	215
6	1.00	1.00	1.00	206
7	1.00	0.99	0.99	222
8	1.00	1.00	1.00	254
9	0.82	0.85	0.83	231
10	1.00	0.99	0.99	255
11	1.00	1.00	1.00	221
12	0.85	0.83	0.84	238
13	1.00	1.00	1.00	240
14	1.00	1.00	1.00	221
15	1.00	1.00	1.00	229
16	1.00	1.00	1.00	236
17	0.99	1.00	0.99	215
18	1.00	1.00	1.00	219
19	1.00	1.00	1.00	248
20	1.00	1.00	1.00	240
21	1.00	1.00	1.00	254
22	1.00	1.00	1.00	254
23	1.00	1.00	1.00	230
24	1.00	1.00	1.00	202
25	1.00	1.00	1.00	228
26	1.00	1.00	1.00	216
avg / total	0.99	0.99	0.99	6001

accuracy score = 0.684807692308
report =

	precision	recall	f1-score	support
1	0.51	0.61	0.56	800
2	0.71	0.74	0.73	800
3	0.71	0.82	0.76	800
4	0.68	0.61	0.64	800
5	0.79	0.69	0.74	800
6	0.73	0.66	0.69	800
7	0.65	0.38	0.48	800
8	0.58	0.73	0.65	800
9	0.50	0.70	0.58	800
10	0.65	0.79	0.71	800
11	0.66	0.67	0.66	800
12	0.45	0.38	0.41	800
13	0.82	0.91	0.86	800
14	0.66	0.65	0.65	800
15	0.79	0.85	0.82	800
16	0.76	0.81	0.78	800
17	0.61	0.57	0.59	800
18	0.68	0.58	0.63	800
19	0.91	0.76	0.83	800
20	0.53	0.55	0.54	800
21	0.73	0.75	0.74	800
22	0.69	0.77	0.73	800
23	0.82	0.86	0.84	800
24	0.84	0.64	0.72	800
25	0.65	0.63	0.64	800
26	0.88	0.71	0.79	800
avg / total	0.69	0.68	0.68	20800

accuracy score = 0.662889518414
report =

	precision	recall	f1-score	support
1	0.53	0.62	0.57	235
2	0.66	0.58	0.62	216
3	0.67	0.77	0.72	246
4	0.65	0.58	0.61	230
5	0.80	0.64	0.71	215
6	0.76	0.62	0.68	206
7	0.66	0.38	0.48	222
8	0.59	0.74	0.66	254
9	0.40	0.78	0.53	231
10	0.64	0.79	0.71	255
11	0.66	0.64	0.65	221
12	0.33	0.16	0.21	238
13	0.82	0.88	0.85	240
14	0.61	0.62	0.62	221
15	0.73	0.83	0.78	229
16	0.71	0.78	0.74	236
17	0.65	0.61	0.63	215
18	0.67	0.57	0.61	219
19	0.89	0.71	0.79	248
20	0.52	0.53	0.52	240
21	0.74	0.73	0.74	254
22	0.69	0.76	0.72	254
23	0.82	0.86	0.84	230
24	0.76	0.64	0.70	202
25	0.65	0.65	0.65	228
26	0.90	0.71	0.79	216
avg / total	0.67	0.66	0.66	6001

Figure above represents the precision, recall and f1-score of the model against the training sets. From left to right each figure represents linear, rbf and sigmoid functions. It is

important to notice that in all the cases above, our model performs much better against the training set than the testing set. We believe this is because in all cases, the kernel functions are not appropriate for training the model for this particular problem. Especially, if you look at the result from linear kernel function, the model has relatively high accuracy for the testing dataset, however it also has extremely high accuracy against the training dataset of 98 percent, which means the SVM modeled using linear kernel function seriously overfits the training set. Out of those three different kernel functions, we decided to incorporate rbf into our final SVM model.

And the following result represents model performance of the SVM model with rbf function as the kernel function with 20,001 training dataset tested against the 20800 total tests sets(on the right) and against the training datasets.

Conclusion

accuracy score = 0.768961551922
report =

	precision	recall	f1-score	support
1	0.63	0.70	0.66	764
2	0.77	0.81	0.79	714
3	0.83	0.84	0.84	764
4	0.73	0.74	0.74	764
5	0.82	0.78	0.80	759
6	0.80	0.78	0.79	771
7	0.72	0.54	0.62	754
8	0.72	0.80	0.75	780
9	0.60	0.62	0.61	774
10	0.76	0.85	0.80	794
11	0.74	0.75	0.75	757
12	0.56	0.66	0.61	784
13	0.87	0.91	0.89	779
14	0.72	0.76	0.74	776
15	0.84	0.89	0.86	770
16	0.84	0.86	0.85	794
17	0.68	0.71	0.69	758
18	0.77	0.69	0.73	730
19	0.93	0.83	0.88	797
20	0.69	0.66	0.67	773
21	0.83	0.80	0.82	813
22	0.82	0.80	0.81	795
23	0.90	0.87	0.89	768
24	0.81	0.77	0.79	779
25	0.78	0.75	0.77	756
26	0.92	0.81	0.86	734
avg / total	0.77	0.77	0.77	20001

accuracy score = 0.753798076923
report =

	precision	recall	f1-score	support
1	0.61	0.68	0.65	800
2	0.78	0.80	0.79	800
3	0.83	0.83	0.83	800
4	0.73	0.70	0.72	800
5	0.82	0.79	0.81	800
6	0.79	0.75	0.77	800
7	0.71	0.49	0.58	800
8	0.67	0.75	0.71	800
9	0.61	0.63	0.62	800
10	0.73	0.82	0.77	800
11	0.72	0.74	0.73	800
12	0.54	0.67	0.60	800
13	0.85	0.91	0.88	800
14	0.71	0.73	0.72	800
15	0.82	0.90	0.86	800
16	0.81	0.85	0.83	800
17	0.68	0.65	0.66	800
18	0.76	0.66	0.71	800
19	0.93	0.83	0.87	800
20	0.66	0.65	0.65	800
21	0.80	0.82	0.81	800
22	0.77	0.81	0.79	800
23	0.88	0.88	0.88	800
24	0.83	0.75	0.79	800
25	0.75	0.71	0.73	800
26	0.91	0.79	0.84	800
avg / total	0.76	0.75	0.75	20800

The performance of our models were measured with following metrics.

The **precision** is the ratio of true positive divided by (true positive + false positive). The precision measures how well model label positive when a specific data represents negative. The **recall** is the ratio true positive divided by true positive plus false negative. The recall measures how well the model finds the “right label” for the specific classification. The **F-beta** scores can be interpreted as a weighted harmonic mean of the precision and recall, where an F-beta score reaches its best value at 1 and worst score at 0. (According to the documentation of this package). We used F1 score when we talked about the performance of our model for this section. Based on Support Vector Machine algorithm, we were able to obtain the model accuracy, (based on f1 score) of 76 percent**.** We concluded that this model is not a good model because 1) it is still overfitting the data 2) does not have overall high accuracy. We believe this is due to the fact that our kernel function is not appropriate for this particular problem and we

would guess that coming up with better kernel function would increase the accuracy of the model.

Feed-Forward Neural Network

Introduction

In a multilayered feed-forward neural network, the information moves from the input layer to hidden layer(s) to an output layer.

Advantage

Neural networks have shown successes in image recognition tasks. Indeed, neural networks are good for nonlinear data with large number of inputs, images are a good example of such data.

Disadvantage

Disadvantages of neural networks include its empirical nature of model development and proneness to overfitting. Training a multi-layer neural network is difficult in that it requires a lot of parameter tuning.

Constructing a baseline neural network architecture

1. General layout

Since our data is not linearly separable, we used a multi-layer neural network. The input layer consists of 784 neurons, which matches the number of features - 28*28 pixels - of any input image. The output layer consists of 27 neurons.

On a side note, in processing our labels data(y_train and y_test), we used np_utils.to_categorical to convert the original data (where ‘A’ is represented as a label of 1, and ‘Z’ is represented as a label of 26) to one-hot vectors. Because np_utils.to_categorical expects a label of 0 to start with, the first one-hot vector created (namely, [1 0 0 0 0 …. 0]) represents nothing, while [0 1 0 0 0 ….0] represents ‘A’, [0 0 0 0 0 … 1] representing ‘Z’ and so on. As a result, we have 27 neurons in the output layer.

In practice, two layers are said to be sufficient for almost any application since one layer is supposed to approximate most functions. We started out with one hidden layer in our baseline model. We compared its performance with a model with two hidden layers.

2. Number of neurons in hidden layers

It seems that the most commonly relied rule of thumb for deciding an optimal size of hidden neurons is picking a number between the size of input neurons and the size of output neurons. We chose the size of input neurons as our starting point.

We adjusted this number by trial and error. Using too few hidden neurons would lead to underfitting, where there are too few neurons to adequately detect the many signals in a dataset. Using too many hidden neurons would lead to overfitting.

3. Activation function

We used a ReLu: y(x)=max(0,x) activation for hidden layers, considering the following advantages:

- It is a simple way to inject nonlinearity into the model.
- It is a popular choice in practice.
- It doesn't face a gradient vanishing problem.

We used a Softmax activation for the output layer. A Softmax activation outputs a vector of probabilities for multiple classes.

4. Loss function

We used a cross-entropy loss function. It aims only to maximize the model's confidence in the correct class, and is not concerned with the distribution of probabilities for other classes.

5. Optimization algorithm

We choose the 'adam' optimizer, which is said to perform well for most cases in practice. Known advantages for choosing adam are:

- Repeated weight updates in a particular direction build up momentum, so that the algorithm achieves faster convergence.
- It adaptively selects an appropriate learning rate.

6. Evaluation

We selected 10% of our training dataset (validation data) to see after each epoch to see the performance of the model against dataset that it hasn't seen during training.

We used a separate set of 20800 samples (test data) to test how well our final model predicts on an unseen dataset.

7. Addressing the overfitting problem

As a regularization method, we relied on the Dropout feature of keras to randomly drop a certain percentage of neurons from training. In a fully connected layer, neurons tend to develop co-dependency amongst each other. This leads to overfitting of training data since the co-dependency restrains the effects of individual neurons. We can prevent this by randomly selecting nodes to completely 'drop' from the calculations. We set Dropout=0.2 in our model, so neurons are randomly selected and dropped out with a probability of 0.2. We will further adjust this probability to see how it affects the performance of our model.

The Dropout rate being too low will result in minimal effect of regularization, and the rate being too high will result in under-learning by the model.

As another way to deal with overfitting, we increased the number of epochs, starting off with 20 epochs, until we find a point where performance on a validation set starts to decrease.

Optimizing the model

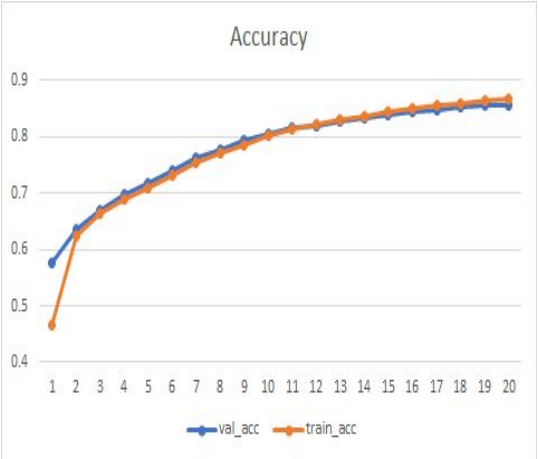
1. 1st model - Baseline Model

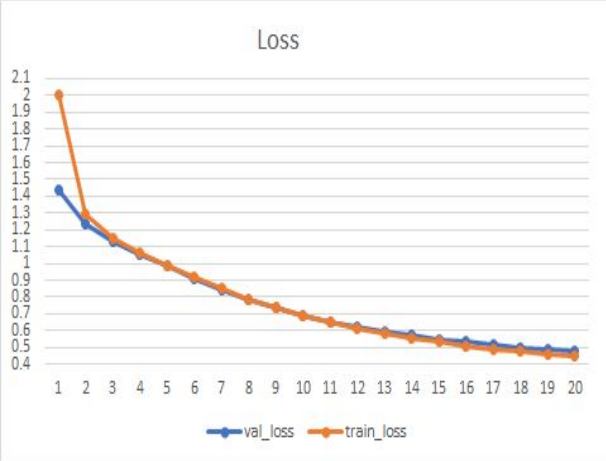
Our baseline model is a feed-forward neural network with one hidden layer with the same number of neurons as input neurons. A ReLu function is used for activation in the hidden layers, and a softmax function is used for activation in the output layer. Dropout rate is set to

0.2. Categorical_crossentropy is used as the loss function, and Adam is used to update the weights. This model is fit over 20 epochs with weight updates for every image input. For each epoch, the performance of the trained model is evaluated with the validation set. The final model is evaluated using the test dataset.

The final model obtains an accuracy of 0.86 and a loss of 0.47. Validation loss keeps decreasing to 0.4800 and validation accuracy keeps increasing to 0.8554.

We conclude that we can train our model further to make it better.





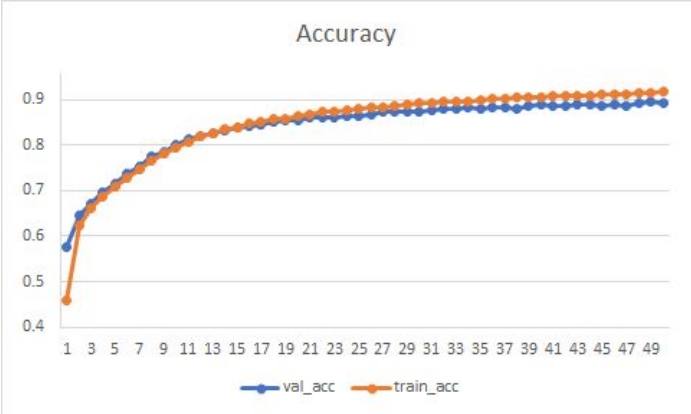
2. 2nd model - increasing epochs

For the second model, We trained the same baseline model as above, but increased the number of epochs to 50 times. Our goal for this was to find the cutoff epoch in which the overfitting occurs: the accuracy of the validation dataset stagnates while the accuracy of the training set increases or the loss function for the validation set stagnates while the loss of the training set decrease.

The final model we obtained had an accuracy of 0.89 and a loss of 0.36 against the validation set. Since the only difference from the previous model is that we increased the number of epochs, we concluded that the model improved due to the increased number of epochs.

We also noticed that the validation loss keeps decreasing to 0.3628 until epoch 44, and validation accuracy keeps increasing to 0.8734 only until epoch 28. We noticed an improvement in both metrics however we also could not rule out the problem of overfitting here.

Therefore, we choose a smaller number of hidden neurons in our next modification of the model to reduce overfitting.

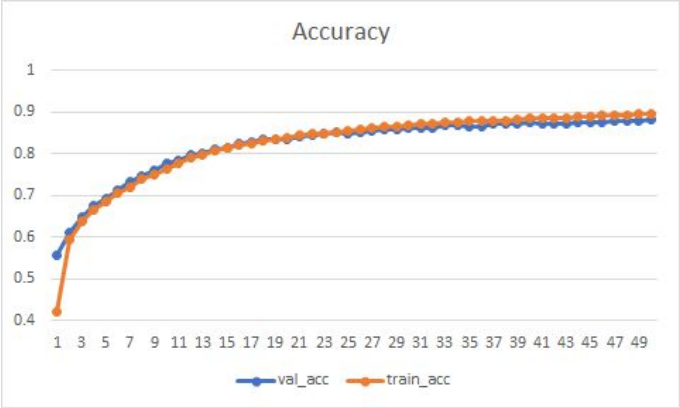


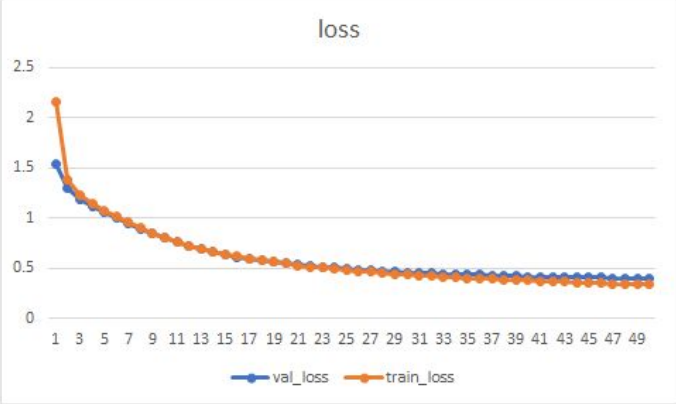
3. 3rd model - decreasing the number of hidden neurons

For our third model, we decreased the number of hidden neurons to be around 550 neurons. Number of epochs is still kept to 50.

The final model obtains an accuracy of 0.88 and a loss of 0.38. Performance on the test set does not look that different from before. Validation loss keeps decreasing to 0.3918, but validation accuracy keeps increasing to 0.8712 until epoch 37. We notice a possible decrease in performance associated with a smaller number of hidden neurons.

In our next modification, we add another hidden layer and see how that affects the performance of the model. Because another layer will be added, we again lower the number of epochs to 20 to leverage overfitting.



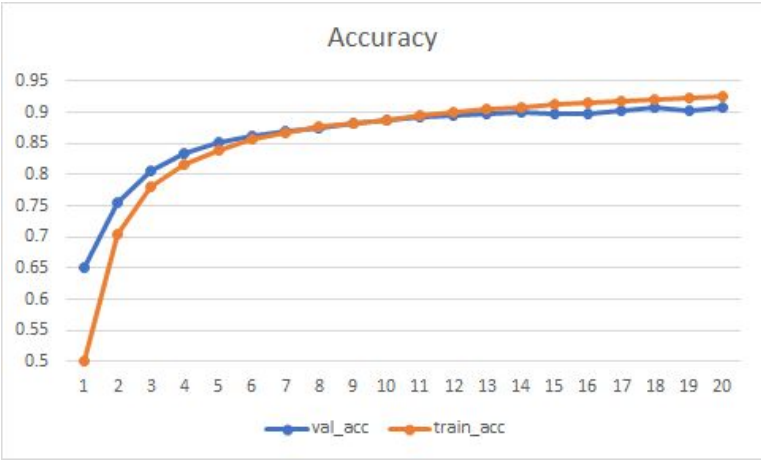


4. 4th model - adding another hidden layer

For the 4th model, we added another hidden layer; thus we had total of 2 hidden layers with 784 neurons each. We also set the number of epochs to 20.

The final model obtained an accuracy of 0.91 and a loss of 0.30. It seems that there has been an improvement in the model. Validation loss kept decreasing until it reached 0.3142, while validation accuracy kept increasing to 0.9010 both around epoch 14. It seems that there has been an improvement in the model, but we notice that the train accuracy exceeds the validation accuracy around this point which leads us to believe that there is a chance of overfitting.

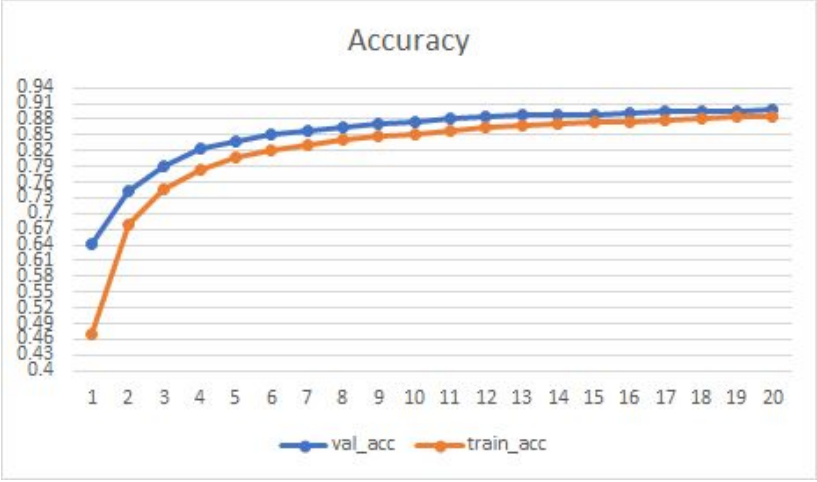
To combat the problem of overfitting, we increased the Dropout rate for the next model.



5. 5th model - increasing the Dropout rate

This model had the same structure as the 4th model except that we increased the dropout rate of the first hidden layer to 0.5 and the the second with a dropout rate of 0.3.

The final model obtained an accuracy of 0.90 and a loss of 0.31 against the validation set. The Validation loss kept decreasing to 0.3162, while validation accuracy peaked at epoch 18 with 0.8965 and fluctuating after that.



Conclusion

We have constructed a baseline neural network model and further modified different hyperparameters to improve the model performance. As a result, we were able to obtain a model with a test accuracy of 0.9058 and a validation accuracy of 0.9010.

Convolutional Neural Network (CNN)

Introduction

Convolutional Neural Networks are very similar to Neural Networks that we implemented above. It has the same structure, neurons with biases and learnable weights, and other similar architectures of ordinary Neural network. Convolutional neural network, however, explicitly assumes that the inputs are images.

We were unable to study this algorithm in detail. Since this method is directly tailored towards image recognition, we thought it would be a good idea to build a basic model using this algorithm and compare with other two models we built above.

Advantages

The main advantage of CNN is its increased accuracy in image recognition tasks.

Disadvantages

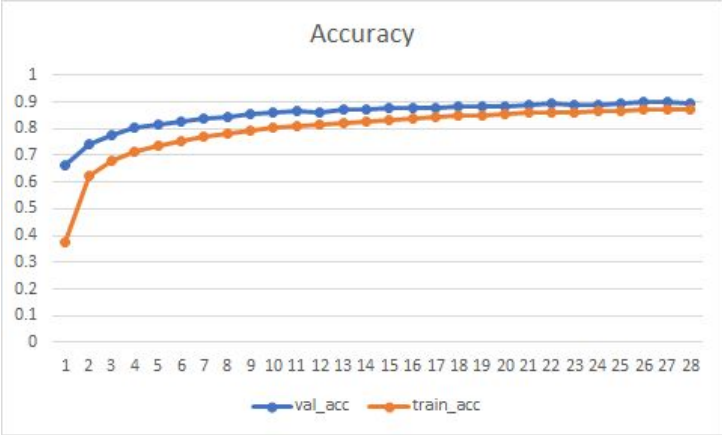
CNN is computationally heavy so requires a good GPU for sufficient training and parameter tuning.

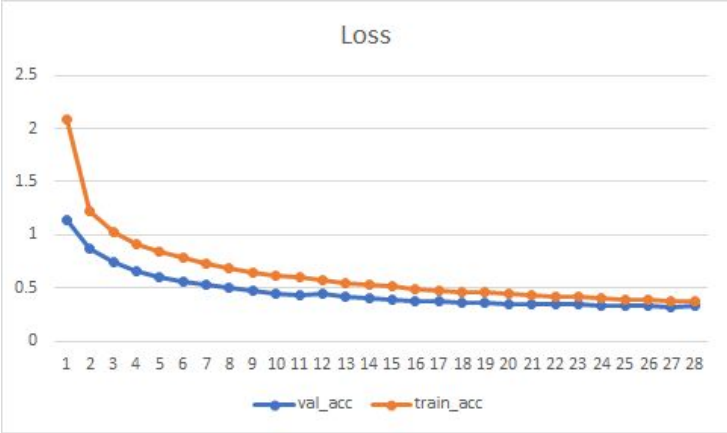
Baseline model

```
model.add(Convolution2D(32, (3, 3), activation='relu', padding = 'valid',input_shape=(1,28,28), data_format='channels_first'))

model.add(Convolution2D(32,(3,3),activation='relu'))
# maxpooling is a way to reduce the number of parameters
model.add(MaxPooling2D(pool_size=(2,2)))
# so far, we've added 2 convolution layers
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(27, activation='softmax'))
```

Convolutional Neural Network incorporates three different layers: convolutional layers, pooling layer and fully connected layer. For our initial model, we added a convolutional layer with 3 by 3 filters, and pooling layer filter of 2 by 2 and tested against the validation sets. The below represent the accuracy and the loss for each epoch. As the graph suggests, the model does not improve its validation accuracy after 23rd epoch, which has the accuracy of around 88 percent and 40 percent in loss function.

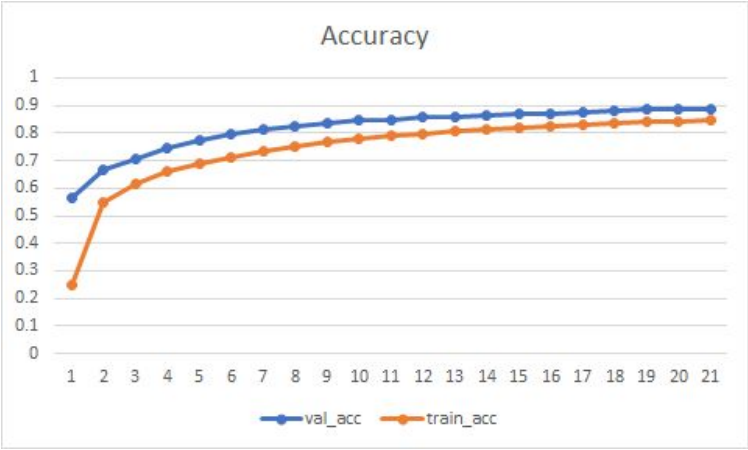




We realized that the performance of the model is not so vastly improved from the model developed from above so we decided to change other parameters, such as dimension of the filter for convolutional layer or number of the nodes in the hidden layer.

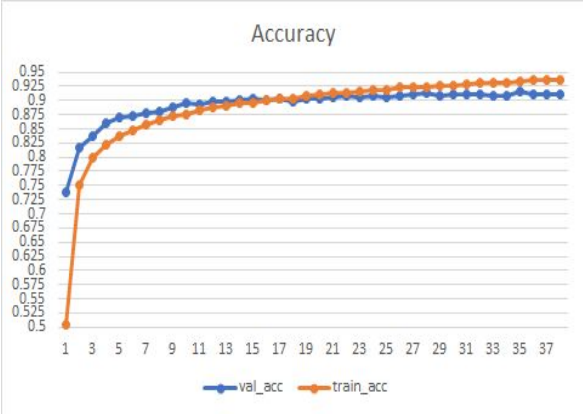
Changing the parameter for the Convolutional Layer

For the second model of Convolutional Neural Network, we decided to change the size of the convolutional layer from 3 by 3 to 6 by 6. And had the following results.



The above model had validation accuracy of 0.8845 and validation error of 0.357 which are not significant improvements from the previous model.

Changing the number of nodes to 512



For our third model, we change the number of nodes in the hidden layer from 128 to 512. We thought this would definitely improve the accuracy . The below represents the accuracy and for each iteration. As you can see, overfitting problem becomes apparent around 13 to 15 epochs. We trained our model until the 14th epoch, when it had the lowest validation error of 0.3122 and highest accuracy of 0.9042. Since this had the best result out of three models, we decided to use this model to see how well this model perform against the test data. This model had loss of 0.313174440099 and accuracy= 0.913942307692 which is the highest among the other 2 models.

Conclusion

Even though our convolutional neural network only had two convolutional layers with basic setup, we were able to obtain improved accuracy and loss; we believe that CNN would definitely lead us to develop the most accurate model in predicting the classification of alphabets. Unfortunately, we were unable to study and carefully implement the model using

Convolutional Neural Network due to time constraint. One of the downfall of the Convolutional Neural Network is that it is quite computational heavy. On average each epoch takes about 300 seconds to 600 seconds. On top of that, we believe that carefully calibrating each layers, nodes, number of epochs and other hyperparameters for this particular problem would be another project on its own.

Overall Conclusion

The purpose of this project was to familiarize ourselves with various machine learning algorithms for image recognition. We were able to develop various models using 3 different algorithms such as Support Vector Machine, Feed Forward Neural Network and Convolutional Neural Network. Through this project, my partner and I were able to gain deeper understanding on this topic.

References

We referred to the book cited below in writing our code.

Hikouzkue, K. 파이썬을 이용한 머신러닝, 딥러닝 실전 개발 입문. Paju: Wikibooks, 2017. Print