Introduction

MNIST ("Modified National Institute of Standards and Technology") is the de facto “Hello World” dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike.

In this competition, we aim to correctly identify digits from a dataset of tens of thousands of handwritten images. Kaggle has curated a set of tutorial-style kernels which cover everything from regression to neural networks. They hope to encourage us to experiment with different algorithms to learn first-hand what works well and how techniques compare.

Approach

For this competition, we will be using Keras (with TensorFlow as our backend) as the main package to create a simple neural network to predict, as accurately as we can, digits from handwritten images. In particular, we will be calling the Functional Model API of Keras, and creating a 4-layered and 5-layered neural network.

Also, we will be experimenting with various optimizers: the plain vanilla Stochastic Gradient Descent optimizer and the Adam optimizer. However, there are many other parameters, such as training epochs which will we will not be experimenting with.

In addition, the choice of hidden layer units are completely arbitrary and may not be optimal. This is yet another parameter which we will not attempt to tinker with. Lastly, we introduce dropout, a form of regularisation, in our neural networks to prevent overfitting.

Result

Following our simulations on the cross validation dataset, it appears that a 4-layered neural network, using 'Adam' as the optimizer along with a learning rate of 0.01, performs best. We proceed to introduce dropout in the model, and use the model to predict for the test set.

The test predictions (submitted to Kaggle) generated by our model predicts with an accuracy score of 97.600%, which places us at the top 55 percentile of the competition.

Importing key libraries, and reading data

In [1]:

import pandas as pd

import numpy as np

np.random.seed(1212)

import keras

from keras.models import Model

from keras.layers import \*

from keras import optimizers

Using TensorFlow backend.

In [2]:

df\_train = pd.read\_csv('../input/train.csv')

df\_test = pd.read\_csv('../input/test.csv')

In [3]:

df\_train.head() *# 784 features, 1 label*

Out[3]:

|  | label | pixel0 | pixel1 | pixel2 | pixel3 | pixel4 | pixel5 | pixel6 | pixel7 | pixel8 | ... | pixel774 | pixel775 | pixel776 | pixel777 | pixel778 | pixel779 | pixel780 | pixel781 | pixel782 | pixel783 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

5 rows × 785 columns

Splitting into training and validation dataset

In [4]:

df\_features = df\_train.iloc[:, 1:785]

df\_label = df\_train.iloc[:, 0]

X\_test = df\_test.iloc[:, 0:784]

print(X\_test.shape)

(28000, 784)

In [5]:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_cv, y\_train, y\_cv = train\_test\_split(df\_features, df\_label,

test\_size = 0.2,

random\_state = 1212)

X\_train = X\_train.as\_matrix().reshape(33600, 784) *#(33600, 784)*

X\_cv = X\_cv.as\_matrix().reshape(8400, 784) *#(8400, 784)*

X\_test = X\_test.as\_matrix().reshape(28000, 784)

Data cleaning, normalization and selection

In [6]:

print((min(X\_train[1]), max(X\_train[1])))

(0, 255)

As the pixel intensities are currently between the range of 0 and 255, we proceed to normalize the features, using broadcasting. In addition, we proceed to convert our labels from a class vector to binary One Hot Encoded

In [7]:

*# Feature Normalization*

X\_train = X\_train.astype('float32'); X\_cv= X\_cv.astype('float32'); X\_test = X\_test.astype('float32')

X\_train /= 255; X\_cv /= 255; X\_test /= 255

*# Convert labels to One Hot Encoded*

num\_digits = 10

y\_train = keras.utils.to\_categorical(y\_train, num\_digits)

y\_cv = keras.utils.to\_categorical(y\_cv, num\_digits)

In [8]:

*# Printing 2 examples of labels after conversion*

print(y\_train[0]) *# 2*

print(y\_train[3]) *# 7*

[ 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]

[ 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

Model Fitting

We proceed by fitting several simple neural network models using Keras (with TensorFlow as our backend) and collect their accuracy. The model that performs the best on the validation set will be used as the model of choice for the competition.

Model 1: Simple Neural Network with 4 layers (300, 100, 100, 200)

In our first model, we will use the Keras library to train a neural network with the activation function set as ReLu. To determine which class to output, we will rely on the SoftMax function

In [9]:

*# Input Parameters*

n\_input = 784 *# number of features*

n\_hidden\_1 = 300

n\_hidden\_2 = 100

n\_hidden\_3 = 100

n\_hidden\_4 = 200

num\_digits = 10

In [10]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

In [11]:

*# Our model would have '6' layers - input layer, 4 hidden layer and 1 output layer*

model = Model(Inp, output)

model.summary() *# We have 297,910 parameters to estimate*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_1 (Dense) (None, 300) 235500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_2 (Dense) (None, 100) 30100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_3 (Dense) (None, 100) 10100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_4 (Dense) (None, 200) 20200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Output\_Layer (Dense) (None, 10) 2010

=================================================================

Total params: 297,910

Trainable params: 297,910

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [12]:

*# Insert Hyperparameters*

learning\_rate = 0.1

training\_epochs = 20

batch\_size = 100

sgd = optimizers.SGD(lr=learning\_rate)

In [13]:

*# We rely on the plain vanilla Stochastic Gradient Descent as our optimizing methodology*

model.compile(loss='categorical\_crossentropy',

optimizer='sgd',

metrics=['accuracy'])

In [14]:

history1 = model.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

verbose = 2,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

4s - loss: 1.8541 - acc: 0.4983 - val\_loss: 1.0046 - val\_acc: 0.7602

Epoch 2/20

4s - loss: 0.6480 - acc: 0.8295 - val\_loss: 0.4635 - val\_acc: 0.8729

Epoch 3/20

4s - loss: 0.4092 - acc: 0.8834 - val\_loss: 0.3616 - val\_acc: 0.8980

Epoch 4/20

4s - loss: 0.3373 - acc: 0.9026 - val\_loss: 0.3121 - val\_acc: 0.9100

Epoch 5/20

4s - loss: 0.2979 - acc: 0.9139 - val\_loss: 0.2893 - val\_acc: 0.9169

Epoch 6/20

4s - loss: 0.2684 - acc: 0.9227 - val\_loss: 0.2651 - val\_acc: 0.9238

Epoch 7/20

4s - loss: 0.2453 - acc: 0.9296 - val\_loss: 0.2557 - val\_acc: 0.9257

Epoch 8/20

4s - loss: 0.2272 - acc: 0.9352 - val\_loss: 0.2322 - val\_acc: 0.9336

Epoch 9/20

3s - loss: 0.2101 - acc: 0.9378 - val\_loss: 0.2175 - val\_acc: 0.9363

Epoch 10/20

4s - loss: 0.1952 - acc: 0.9438 - val\_loss: 0.2053 - val\_acc: 0.9399

Epoch 11/20

4s - loss: 0.1828 - acc: 0.9468 - val\_loss: 0.1954 - val\_acc: 0.9427

Epoch 12/20

4s - loss: 0.1708 - acc: 0.9504 - val\_loss: 0.1850 - val\_acc: 0.9445

Epoch 13/20

4s - loss: 0.1613 - acc: 0.9529 - val\_loss: 0.1806 - val\_acc: 0.9458

Epoch 14/20

3s - loss: 0.1516 - acc: 0.9562 - val\_loss: 0.1764 - val\_acc: 0.9471

Epoch 15/20

4s - loss: 0.1432 - acc: 0.9588 - val\_loss: 0.1658 - val\_acc: 0.9502

Epoch 16/20

4s - loss: 0.1354 - acc: 0.9605 - val\_loss: 0.1596 - val\_acc: 0.9531

Epoch 17/20

4s - loss: 0.1290 - acc: 0.9621 - val\_loss: 0.1547 - val\_acc: 0.9533

Epoch 18/20

4s - loss: 0.1217 - acc: 0.9645 - val\_loss: 0.1479 - val\_acc: 0.9569

Epoch 19/20

4s - loss: 0.1157 - acc: 0.9670 - val\_loss: 0.1472 - val\_acc: 0.9560

Epoch 20/20

3s - loss: 0.1101 - acc: 0.9683 - val\_loss: 0.1409 - val\_acc: 0.9580

Using a 4 layer neural network with:

1. 20 training epochs
2. A training batch size of 100
3. Hidden layers set as (300, 100, 100, 200)
4. Learning rate of 0.1

Achieved a training score of around 96-98% and a test score of around 95 - 97%.

Can we do better if we were to change the optimizer? To find out, we use the Adam optimizer for our second model, while maintaining the same parameter values for all other parameters.

In [15]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

*# We rely on ADAM as our optimizing methodology*

adam = keras.optimizers.Adam(lr=learning\_rate)

model2 = Model(Inp, output)

model2.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

In [16]:

history2 = model2.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

verbose = 2,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

5s - loss: 0.3434 - acc: 0.8972 - val\_loss: 0.1616 - val\_acc: 0.9510

Epoch 2/20

5s - loss: 0.1260 - acc: 0.9610 - val\_loss: 0.1195 - val\_acc: 0.9645

Epoch 3/20

5s - loss: 0.0825 - acc: 0.9743 - val\_loss: 0.1021 - val\_acc: 0.9685

Epoch 4/20

5s - loss: 0.0603 - acc: 0.9807 - val\_loss: 0.1212 - val\_acc: 0.9639

Epoch 5/20

5s - loss: 0.0455 - acc: 0.9855 - val\_loss: 0.1224 - val\_acc: 0.9660

Epoch 6/20

5s - loss: 0.0394 - acc: 0.9864 - val\_loss: 0.1034 - val\_acc: 0.9700

Epoch 7/20

5s - loss: 0.0313 - acc: 0.9892 - val\_loss: 0.1193 - val\_acc: 0.9700

Epoch 8/20

5s - loss: 0.0275 - acc: 0.9914 - val\_loss: 0.0979 - val\_acc: 0.9735

Epoch 9/20

5s - loss: 0.0247 - acc: 0.9922 - val\_loss: 0.0980 - val\_acc: 0.9752

Epoch 10/20

5s - loss: 0.0192 - acc: 0.9938 - val\_loss: 0.0926 - val\_acc: 0.9769

Epoch 11/20

5s - loss: 0.0186 - acc: 0.9938 - val\_loss: 0.1209 - val\_acc: 0.9738

Epoch 12/20

5s - loss: 0.0240 - acc: 0.9920 - val\_loss: 0.1114 - val\_acc: 0.9736

Epoch 13/20

5s - loss: 0.0132 - acc: 0.9959 - val\_loss: 0.1084 - val\_acc: 0.9767

Epoch 14/20

5s - loss: 0.0156 - acc: 0.9949 - val\_loss: 0.1421 - val\_acc: 0.9692

Epoch 15/20

5s - loss: 0.0169 - acc: 0.9940 - val\_loss: 0.1213 - val\_acc: 0.9735

Epoch 16/20

5s - loss: 0.0147 - acc: 0.9951 - val\_loss: 0.1094 - val\_acc: 0.9760

Epoch 17/20

4s - loss: 0.0090 - acc: 0.9975 - val\_loss: 0.1130 - val\_acc: 0.9780

Epoch 18/20

5s - loss: 0.0102 - acc: 0.9968 - val\_loss: 0.1276 - val\_acc: 0.9763

Epoch 19/20

5s - loss: 0.0152 - acc: 0.9954 - val\_loss: 0.1146 - val\_acc: 0.9763

Epoch 20/20

5s - loss: 0.0107 - acc: 0.9963 - val\_loss: 0.1192 - val\_acc: 0.9767

As it turns out, it does appear to be the case that the optimizer plays a crucial part in the validation score. In particular, the model which relies on 'Adam' as its optimizer tend to perform 1.5 - 2.5% better on average. Going forward, we will use 'Adam' as our optimizer of choice.

What if we changed the learning rate from 0.1 to 0.01, or 0.5? Will it have any impact on the accuracy? Model 2A

In [17]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

learning\_rate = 0.01

adam = keras.optimizers.Adam(lr=learning\_rate)

model2a = Model(Inp, output)

model2a.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

In [18]:

history2a = model2a.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

verbose = 2,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

5s - loss: 0.3403 - acc: 0.8973 - val\_loss: 0.1920 - val\_acc: 0.9423

Epoch 2/20

5s - loss: 0.1242 - acc: 0.9618 - val\_loss: 0.1203 - val\_acc: 0.9645

Epoch 3/20

5s - loss: 0.0819 - acc: 0.9749 - val\_loss: 0.1090 - val\_acc: 0.9663

Epoch 4/20

5s - loss: 0.0612 - acc: 0.9802 - val\_loss: 0.0993 - val\_acc: 0.9704

Epoch 5/20

5s - loss: 0.0459 - acc: 0.9857 - val\_loss: 0.1067 - val\_acc: 0.9681

Epoch 6/20

5s - loss: 0.0352 - acc: 0.9880 - val\_loss: 0.1074 - val\_acc: 0.9705

Epoch 7/20

5s - loss: 0.0278 - acc: 0.9906 - val\_loss: 0.0970 - val\_acc: 0.9736

Epoch 8/20

5s - loss: 0.0242 - acc: 0.9920 - val\_loss: 0.1112 - val\_acc: 0.9736

Epoch 9/20

5s - loss: 0.0225 - acc: 0.9926 - val\_loss: 0.1273 - val\_acc: 0.9664

Epoch 10/20

5s - loss: 0.0245 - acc: 0.9923 - val\_loss: 0.1239 - val\_acc: 0.9706

Epoch 11/20

5s - loss: 0.0212 - acc: 0.9934 - val\_loss: 0.1088 - val\_acc: 0.9770

Epoch 12/20

5s - loss: 0.0180 - acc: 0.9943 - val\_loss: 0.1090 - val\_acc: 0.9745

Epoch 13/20

5s - loss: 0.0134 - acc: 0.9956 - val\_loss: 0.1138 - val\_acc: 0.9750

Epoch 14/20

5s - loss: 0.0131 - acc: 0.9956 - val\_loss: 0.1173 - val\_acc: 0.9746

Epoch 15/20

5s - loss: 0.0183 - acc: 0.9944 - val\_loss: 0.1064 - val\_acc: 0.9771

Epoch 16/20

5s - loss: 0.0145 - acc: 0.9953 - val\_loss: 0.1169 - val\_acc: 0.9743

Epoch 17/20

5s - loss: 0.0106 - acc: 0.9968 - val\_loss: 0.1041 - val\_acc: 0.9788

Epoch 18/20

5s - loss: 0.0057 - acc: 0.9984 - val\_loss: 0.1258 - val\_acc: 0.9757

Epoch 19/20

5s - loss: 0.0147 - acc: 0.9954 - val\_loss: 0.1085 - val\_acc: 0.9776

Epoch 20/20

5s - loss: 0.0124 - acc: 0.9960 - val\_loss: 0.1187 - val\_acc: 0.9749

Model 2B

In [19]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

learning\_rate = 0.5

adam = keras.optimizers.Adam(lr=learning\_rate)

model2b = Model(Inp, output)

model2b.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

In [20]:

history2b = model2b.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

33600/33600 [==============================] - 5s - loss: 0.3354 - acc: 0.9026 - val\_loss: 0.1438 - val\_acc: 0.9570

Epoch 2/20

33600/33600 [==============================] - 5s - loss: 0.1231 - acc: 0.9619 - val\_loss: 0.1276 - val\_acc: 0.9612

Epoch 3/20

33600/33600 [==============================] - 5s - loss: 0.0809 - acc: 0.9749 - val\_loss: 0.0969 - val\_acc: 0.9719

Epoch 4/20

33600/33600 [==============================] - 5s - loss: 0.0561 - acc: 0.9819 - val\_loss: 0.0867 - val\_acc: 0.9744

Epoch 5/20

33600/33600 [==============================] - 5s - loss: 0.0475 - acc: 0.9849 - val\_loss: 0.0882 - val\_acc: 0.9743

Epoch 6/20

33600/33600 [==============================] - 5s - loss: 0.0363 - acc: 0.9887 - val\_loss: 0.1015 - val\_acc: 0.9727

Epoch 7/20

33600/33600 [==============================] - 5s - loss: 0.0288 - acc: 0.9908 - val\_loss: 0.1124 - val\_acc: 0.9713

Epoch 8/20

33600/33600 [==============================] - 5s - loss: 0.0318 - acc: 0.9901 - val\_loss: 0.1221 - val\_acc: 0.9694

Epoch 9/20

33600/33600 [==============================] - 5s - loss: 0.0214 - acc: 0.9935 - val\_loss: 0.0882 - val\_acc: 0.9769

Epoch 10/20

33600/33600 [==============================] - 5s - loss: 0.0210 - acc: 0.9934 - val\_loss: 0.1410 - val\_acc: 0.9670

Epoch 11/20

33600/33600 [==============================] - 5s - loss: 0.0198 - acc: 0.9933 - val\_loss: 0.1121 - val\_acc: 0.9756

Epoch 12/20

33600/33600 [==============================] - 5s - loss: 0.0176 - acc: 0.9944 - val\_loss: 0.1084 - val\_acc: 0.9730

Epoch 13/20

33600/33600 [==============================] - 5s - loss: 0.0170 - acc: 0.9945 - val\_loss: 0.1243 - val\_acc: 0.9736

Epoch 14/20

33600/33600 [==============================] - 5s - loss: 0.0157 - acc: 0.9950 - val\_loss: 0.1084 - val\_acc: 0.9770

Epoch 15/20

33600/33600 [==============================] - 5s - loss: 0.0212 - acc: 0.9932 - val\_loss: 0.1142 - val\_acc: 0.9746

Epoch 16/20

33600/33600 [==============================] - 5s - loss: 0.0138 - acc: 0.9953 - val\_loss: 0.1150 - val\_acc: 0.9762

Epoch 17/20

33600/33600 [==============================] - 5s - loss: 0.0090 - acc: 0.9971 - val\_loss: 0.1187 - val\_acc: 0.9761

Epoch 18/20

33600/33600 [==============================] - 5s - loss: 0.0143 - acc: 0.9957 - val\_loss: 0.1204 - val\_acc: 0.9763

Epoch 19/20

33600/33600 [==============================] - 5s - loss: 0.0120 - acc: 0.9962 - val\_loss: 0.0988 - val\_acc: 0.9779

Epoch 20/20

33600/33600 [==============================] - 4s - loss: 0.0142 - acc: 0.9954 - val\_loss: 0.1172 - val\_acc: 0.9768

The accuracy, as measured by the 3 different learning rates 0.01, 0.1 and 0.5 are around 98%, 97% and 98% respectively. As there are no considerable gains by changing the learning rates, we stick with the default learning rate of 0.01.

We proceed to fit a neural network with 5 hidden layers with the features in the hidden layer set as (300, 100, 100, 100, 200) respectively. To ensure that the two models are comparable, we will set the training epochs as 20, and the training batch size as 100.

In [21]:

*# Input Parameters*

n\_input = 784 *# number of features*

n\_hidden\_1 = 300

n\_hidden\_2 = 100

n\_hidden\_3 = 100

n\_hidden\_4 = 100

n\_hidden\_5 = 200

num\_digits = 10

In [22]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

x = Dense(n\_hidden\_5, activation='relu', name = "Hidden\_Layer\_5")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

In [23]:

*# Our model would have '7' layers - input layer, 5 hidden layer and 1 output layer*

model3 = Model(Inp, output)

model3.summary() *# We have 308,010 parameters to estimate*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_5 (InputLayer) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_1 (Dense) (None, 300) 235500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_2 (Dense) (None, 100) 30100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_3 (Dense) (None, 100) 10100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_4 (Dense) (None, 100) 10100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_5 (Dense) (None, 200) 20200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Output\_Layer (Dense) (None, 10) 2010

=================================================================

Total params: 308,010

Trainable params: 308,010

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [24]:

*# We rely on 'Adam' as our optimizing methodology*

adam = keras.optimizers.Adam(lr=0.01)

model3.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

In [25]:

history3 = model3.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

33600/33600 [==============================] - 5s - loss: 0.3611 - acc: 0.8924 - val\_loss: 0.1889 - val\_acc: 0.9433

Epoch 2/20

33600/33600 [==============================] - 5s - loss: 0.1267 - acc: 0.9604 - val\_loss: 0.1305 - val\_acc: 0.9625

Epoch 3/20

33600/33600 [==============================] - 5s - loss: 0.0850 - acc: 0.9738 - val\_loss: 0.1177 - val\_acc: 0.9660

Epoch 4/20

33600/33600 [==============================] - 5s - loss: 0.0598 - acc: 0.9815 - val\_loss: 0.1045 - val\_acc: 0.9713

Epoch 5/20

33600/33600 [==============================] - 5s - loss: 0.0463 - acc: 0.9855 - val\_loss: 0.0926 - val\_acc: 0.9757

Epoch 6/20

33600/33600 [==============================] - 5s - loss: 0.0369 - acc: 0.9884 - val\_loss: 0.1009 - val\_acc: 0.9732

Epoch 7/20

33600/33600 [==============================] - 5s - loss: 0.0339 - acc: 0.9895 - val\_loss: 0.0982 - val\_acc: 0.9760

Epoch 8/20

33600/33600 [==============================] - 5s - loss: 0.0292 - acc: 0.9905 - val\_loss: 0.1002 - val\_acc: 0.9751

Epoch 9/20

33600/33600 [==============================] - 5s - loss: 0.0254 - acc: 0.9917 - val\_loss: 0.1109 - val\_acc: 0.9737

Epoch 10/20

33600/33600 [==============================] - 5s - loss: 0.0212 - acc: 0.9936 - val\_loss: 0.1147 - val\_acc: 0.9714

Epoch 11/20

33600/33600 [==============================] - 5s - loss: 0.0189 - acc: 0.9942 - val\_loss: 0.1076 - val\_acc: 0.9750

Epoch 12/20

33600/33600 [==============================] - 5s - loss: 0.0220 - acc: 0.9930 - val\_loss: 0.1046 - val\_acc: 0.9750

Epoch 13/20

33600/33600 [==============================] - 5s - loss: 0.0225 - acc: 0.9928 - val\_loss: 0.0935 - val\_acc: 0.9773

Epoch 14/20

33600/33600 [==============================] - 5s - loss: 0.0170 - acc: 0.9948 - val\_loss: 0.1237 - val\_acc: 0.9721

Epoch 15/20

33600/33600 [==============================] - 5s - loss: 0.0162 - acc: 0.9950 - val\_loss: 0.1258 - val\_acc: 0.9723

Epoch 16/20

33600/33600 [==============================] - 5s - loss: 0.0128 - acc: 0.9960 - val\_loss: 0.1347 - val\_acc: 0.9738

Epoch 17/20

33600/33600 [==============================] - 5s - loss: 0.0146 - acc: 0.9958 - val\_loss: 0.1049 - val\_acc: 0.9773

Epoch 18/20

33600/33600 [==============================] - 5s - loss: 0.0141 - acc: 0.9958 - val\_loss: 0.1000 - val\_acc: 0.9792

Epoch 19/20

33600/33600 [==============================] - 5s - loss: 0.0063 - acc: 0.9982 - val\_loss: 0.1228 - val\_acc: 0.9745

Epoch 20/20

33600/33600 [==============================] - 5s - loss: 0.0175 - acc: 0.9943 - val\_loss: 0.1135 - val\_acc: 0.9762

Compared to our first model, adding an additional layer did not significantly improve the accuracy from our previous model. However, there are computational costs (in terms of complexity) in implementing an additional layer in our neural network. Given that the benefits of an additional layer are low while the costs are high, we will stick with the 4 layer neural network.

We now proceed to include dropout (dropout rate of 0.3) in our second model to prevent overfitting.

In [26]:

*# Input Parameters*

n\_input = 784 *# number of features*

n\_hidden\_1 = 300

n\_hidden\_2 = 100

n\_hidden\_3 = 100

n\_hidden\_4 = 200

num\_digits = 10

In [27]:

Inp = Input(shape=(784,))

x = Dense(n\_hidden\_1, activation='relu', name = "Hidden\_Layer\_1")(Inp)

x = Dropout(0.3)(x)

x = Dense(n\_hidden\_2, activation='relu', name = "Hidden\_Layer\_2")(x)

x = Dropout(0.3)(x)

x = Dense(n\_hidden\_3, activation='relu', name = "Hidden\_Layer\_3")(x)

x = Dropout(0.3)(x)

x = Dense(n\_hidden\_4, activation='relu', name = "Hidden\_Layer\_4")(x)

output = Dense(num\_digits, activation='softmax', name = "Output\_Layer")(x)

In [28]:

*# Our model would have '6' layers - input layer, 4 hidden layer and 1 output layer*

model4 = Model(Inp, output)

model4.summary() *# We have 297,910 parameters to estimate*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_6 (InputLayer) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_1 (Dense) (None, 300) 235500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 300) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_2 (Dense) (None, 100) 30100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_2 (Dropout) (None, 100) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_3 (Dense) (None, 100) 10100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_3 (Dropout) (None, 100) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hidden\_Layer\_4 (Dense) (None, 200) 20200

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Output\_Layer (Dense) (None, 10) 2010

=================================================================

Total params: 297,910

Trainable params: 297,910

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [29]:

model4.compile(loss='categorical\_crossentropy',

optimizer='adam',

metrics=['accuracy'])

In [30]:

history = model4.fit(X\_train, y\_train,

batch\_size = batch\_size,

epochs = training\_epochs,

validation\_data=(X\_cv, y\_cv))

Train on 33600 samples, validate on 8400 samples

Epoch 1/20

33600/33600 [==============================] - 5s - loss: 0.5867 - acc: 0.8112 - val\_loss: 0.1921 - val\_acc: 0.9439

Epoch 2/20

33600/33600 [==============================] - 5s - loss: 0.2318 - acc: 0.9332 - val\_loss: 0.1373 - val\_acc: 0.9599

Epoch 3/20

33600/33600 [==============================] - 5s - loss: 0.1728 - acc: 0.9491 - val\_loss: 0.1213 - val\_acc: 0.9640

Epoch 4/20

33600/33600 [==============================] - 5s - loss: 0.1415 - acc: 0.9591 - val\_loss: 0.1064 - val\_acc: 0.9696

Epoch 5/20

33600/33600 [==============================] - 5s - loss: 0.1245 - acc: 0.9636 - val\_loss: 0.1047 - val\_acc: 0.9710

Epoch 6/20

33600/33600 [==============================] - 5s - loss: 0.1071 - acc: 0.9678 - val\_loss: 0.0935 - val\_acc: 0.9743

Epoch 7/20

33600/33600 [==============================] - 5s - loss: 0.1000 - acc: 0.9710 - val\_loss: 0.0938 - val\_acc: 0.9737

Epoch 8/20

33600/33600 [==============================] - 6s - loss: 0.0872 - acc: 0.9748 - val\_loss: 0.0912 - val\_acc: 0.9748

Epoch 9/20

33600/33600 [==============================] - 5s - loss: 0.0805 - acc: 0.9768 - val\_loss: 0.0928 - val\_acc: 0.9743

Epoch 10/20

33600/33600 [==============================] - 6s - loss: 0.0766 - acc: 0.9768 - val\_loss: 0.0906 - val\_acc: 0.9756

Epoch 11/20

33600/33600 [==============================] - 5s - loss: 0.0647 - acc: 0.9809 - val\_loss: 0.0903 - val\_acc: 0.9763

Epoch 12/20

33600/33600 [==============================] - 5s - loss: 0.0640 - acc: 0.9803 - val\_loss: 0.0913 - val\_acc: 0.9781

Epoch 13/20

33600/33600 [==============================] - 6s - loss: 0.0630 - acc: 0.9812 - val\_loss: 0.0885 - val\_acc: 0.9782

Epoch 14/20

33600/33600 [==============================] - 6s - loss: 0.0587 - acc: 0.9825 - val\_loss: 0.0857 - val\_acc: 0.9777

Epoch 15/20

33600/33600 [==============================] - 6s - loss: 0.0556 - acc: 0.9828 - val\_loss: 0.0904 - val\_acc: 0.9776

Epoch 16/20

33600/33600 [==============================] - 6s - loss: 0.0524 - acc: 0.9841 - val\_loss: 0.0899 - val\_acc: 0.9783

Epoch 17/20

33600/33600 [==============================] - 6s - loss: 0.0520 - acc: 0.9856 - val\_loss: 0.0897 - val\_acc: 0.9779

Epoch 18/20

33600/33600 [==============================] - 6s - loss: 0.0479 - acc: 0.9857 - val\_loss: 0.0896 - val\_acc: 0.9777

Epoch 19/20

33600/33600 [==============================] - 6s - loss: 0.0482 - acc: 0.9859 - val\_loss: 0.0948 - val\_acc: 0.9762

Epoch 20/20

33600/33600 [==============================] - 5s - loss: 0.0430 - acc: 0.9877 - val\_loss: 0.0864 - val\_acc: 0.9786

With a validation score of close to 98%, we proceed to use this model to predict for the test set.

In [31]:

test\_pred = pd.DataFrame(model4.predict(X\_test, batch\_size=200))

test\_pred = pd.DataFrame(test\_pred.idxmax(axis = 1))

test\_pred.index.name = 'ImageId'

test\_pred = test\_pred.rename(columns = {0: 'Label'}).reset\_index()

test\_pred['ImageId'] = test\_pred['ImageId'] + 1

test\_pred.head()

Out[31]:

|  | ImageId | Label |
| --- | --- | --- |
| 0 | 1 | 2 |
| 1 | 2 | 0 |
| 2 | 3 | 9 |
| 3 | 4 | 9 |
| 4 | 5 | 3 |

In [32]:

test\_pred.to\_csv('mnist\_submission.csv', index = False)

Using this model, we are able to achieve a score of 0.976, which places us at the top 55th percentile!