# RECOGNISING MNIST HANDWRITTEN DIGITS USING CNN MODELS

- SIDDHARTHA SHANMUHAVEL

# Table of Contents

## 1. Tasks for Basic Autoencoder Assignment

A convolution sweeps the window through images then calculates its input and filter dot product pixel values. This allows convolution to emphasize the relevant features. The more filters deployed, the more features that CNN will extract. This allows more features to be found but with the cost of more training time.

There is a sweet spot for the number of layers. Smaller filters collect as much local information as possible, bigger filters represent more global, high-level and representative information. Essentially, the convolution layers promote weight sharing to examine pixels in kernels and develop visual context to classify images.

Unlike Neural Network (NN) where the weights are independent, CNN's weights are attached to the neighboring pixels to extract features in every part of the image. CNN uses max pooling to replace output with a max summary to reduce data size and processing time. This allows you to determine features that produce the highest impact and reduces the risk of overfitting.

Too many neurons, layers, and training epochs promote memorization and inhibit generalize. The more you train your model, the more likely it becomes too specialized. To counter this, you could reduce the complexity by removing a few hidden layers and neurons per layer.

Alternatively, you could also use regularization techniques such as Dropout to remove activation unit in every gradient step training. Each epoch training deactivates different neurons. Since the number of gradient steps is usually high, all neurons will averagely have same occurrences for dropout. Intuitively, the more you drop out, the less likely your model memorizes.

*Values of some variables used in part (1 and 2) -> epochs=50, batch size=128*
**Model Numbers Referenced to the Model Numbers in the Jupyter Notebook*
***Running only one of the 7 given models at a time for training the model!!*
****The Best model is Highlighted in Red*

### 1.1 CNN Models

Given CNN Model with 6 Layers of Conv2D with (16, (3, 3)), (8, (3, 3)), (8, (3, 3)) and
3 MaxPooling2D((2, 2)) and UpSampling2D((2, 2))

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function.*

| Model Number* | Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|---|
| 1 | 6L,16,8,8F(3X3) | 3+3 | 0 | 0 | 0.0910/ 0.0901 |

## 1.2 Complex CNN Models

Why do we use batch normalization? We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get more improvement in the training speed.

Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). To explain covariance shift, let's have a deep network on cat detection. We train our data on only black cats' images. So, if we now try to apply this network to data with colored cats, it is obvious; we're not going to do well.

The training set and the prediction set are both cats' images but they differ a little bit. In other words, if an algorithm learned some X to Y mapping, and if the distribution of X changes, then we might need to retrain the learning algorithm by trying to align the distribution of X with the distribution of Y. Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers. It reduces overfitting because it has a slight regularization effects.

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function.*

| Model Number | Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|---|
| 2** | 6L,16F (3X3) | 0 | 0 | 0 | 0.0586/ 0.0583*** |
| 3 | 6L,16F (3X3) | 0 | 0 | 2 | 0.0588/ 0.0584 |
| 4 | 14L,16F (3X3) | 1 | 2 (p=0.2) | 6 | 0.0618/ 0.0617 |

***Model 2 : Training Loss: 0.0586     Validation Loss: 0.0583 Optimizer= 'adam'***

```
##(..2..)## CNN complex model 1: ###Best Model   loss: 0.0586 - val_loss: 0.0583       optimizer='adam', loss='binary_crossentropy'

input_img = tf.keras.layers.Input(shape=(28, 28, 1)) # adapt this if using `channels_first` image data format

x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
encoded = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)

x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = tf.keras.layers.Conv2D(16, (3, 3), activation='relu', padding = 'same')(x)
decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = tf.keras.models.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.summary()
```

We can compare the above two models (model 1 & model 2) and conclude that MaxPooling2D() did not improve the accuracy of the our model. What is Max Pooling? Max Pooling is a convolution process where the Kernel extracts the maximum value of the area it convolves.

Max Pooling simply says to the Convolutional Neural Network that we will carry forward only that information, if that is the largest information available amplitude wise. But in our case, each pixel is important for us as the image size is already very small(27 x 27) and hence Max Pooling is not beneficial here. In general, we use MaxPooling2D() when the image size is very large.

## 1.3 Dense multi-layer model

| Model Number | Dense | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|
| 5 | 6L(28*28)(28*28*28) | 4(p=0.2) | 4 | 1.8511/1.8399 |
| 6 | 6L(28*28) | 4(p=0.2) | 4 | 1.1832/1.1326 |
| 7 | 10L(28*28) | 8(p=0.2) | 8 | 1.8631/1.8760 |

It seems that the point of saturation for the number of layers has been crossed and now instead of increasing, the accuracy is decreasing.

Why CNNs? The CNNs have several different filters/kernels consisting of trainable parameters which can convolve on a given image spatially to detect features like edges and shapes. These high number of filters essentially learn to capture spatial features from the image based on the learned weights through back propagation and stacked layers of filters can be used to detect complex spatial shapes from the spatial features at every subsequent level.

Hence, they can successfully boil down a given image into a highly abstracted representation which is easy for predicting. In Dense networks we try to find patterns in pixel values given as input for eg. if pixel number 25 and 26 are greater than a certain value it might belong to a certain class and a few complex variations of the same.

This might easily fail if we can have objects anywhere in the image and not necessarily centered like in the MNIST or to a certain extent also in the Fashion-MNIST data. As we increase the number of hidden layers, the accuracy first increases and then after some point it start decreasing. This is the case happened to our above model.

## Training of Model 2: ## Best Model

```
Epoch 35/50
469/469 [==============================] - 135s 288ms/step - loss: 0.0586 - val_loss: 0.0582
Epoch 36/50
469/469 [==============================] - 135s 288ms/step - loss: 0.0586 - val_loss: 0.0582
Epoch 37/50
469/469 [==============================] - 135s 287ms/step - loss: 0.0587 - val_loss: 0.0583
Epoch 38/50
469/469 [==============================] - 134s 287ms/step - loss: 0.0586 - val_loss: 0.0583
Epoch 39/50
469/469 [==============================] - 135s 288ms/step - loss: 0.0586 - val_loss: 0.0583
Epoch 40/50
469/469 [==============================] - 135s 287ms/step - loss: 0.0586 - val_loss: 0.0585
Epoch 41/50
469/469 [==============================] - 134s 286ms/step - loss: 0.0586 - val_loss: 0.0582
Epoch 42/50
469/469 [==============================] - 134s 287ms/step - loss: 0.0585 - val_loss: 0.0582
Epoch 43/50
469/469 [==============================] - 136s 290ms/step - loss: 0.0586 - val_loss: 0.0582
Epoch 44/50
469/469 [==============================] - 136s 289ms/step - loss: 0.0585 - val_loss: 0.0582
Epoch 45/50
469/469 [==============================] - 135s 287ms/step - loss: 0.0585 - val_loss: 0.0582
Epoch 46/50
469/469 [==============================] - 135s 287ms/step - loss: 0.0585 - val_loss: 0.0582
Epoch 47/50
469/469 [==============================] - 135s 287ms/step - loss: 0.0591 - val_loss: 0.0587
Epoch 48/50
469/469 [==============================] - 140s 298ms/step - loss: 0.0588 - val_loss: 0.0584
Epoch 49/50
469/469 [==============================] - 136s 289ms/step - loss: 0.0587 - val_loss: 0.0583
Epoch 50/50
469/469 [==============================] - 136s 289ms/step - loss: 0.0586 - val_loss: 0.0583
<tensorflow.python.keras.callbacks.History at 0x7fe1203c2358>
```

## 2. Denoising Autoencoder

### 2.1 CNN Models

Given CNN Model with 6 Layers of Conv2D with (16, (3, 3)), (8, (3, 3)), (8, (3, 3)) and
3 MaxPooling2D((2, 2)) and UpSampling2D((2, 2))

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function*

We can see from CNN Models that as we increase the number of filters, it reduces our loss and thus increases our accuracy. Also, as we decrease the number of hidden layers, till some point our accuracy increases but further decreasing the layers in turn decreases our accuracy.

| Model Number | Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|---|
| 1 | 6L,16,8,8F(3X3) | 3+3 | 0 | 0 | 0.0816/ 0.0811 |

## 2.2 Complex CNN Models

Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). To explain covariance shift, let's have a deep network on cat detection. We train our data on only black cats' images. So, if we now try to apply this network to data with colored cats, it is obvious; we're not going to do well.

The training set and the prediction set are both cats' images but they differ a little bit. In other words, if an algorithm learned some X to Y mapping, and if the distribution of X changes, then we might need to retrain the learning algorithm by trying to align the distribution of X with the distribution of Y. Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers. It reduces overfitting because it has a slight regularization effect.

### ###CNN Denoising Autoencoder Best Model with Loss: 0.0023

```
##(.3..)## Complex CNN Model 2          optimizer='adam', loss='mean_squared_error'
# Lets' define our autoencoder now
def build_autoenocder():
    input_img = Input(shape=(28,28,1), name='image_input')

    x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding = 'same')(input_img)
    x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding = 'same')(x)
    x = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding = 'same')(x)
    x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
    autoencoder = Model(inputs=input_img, outputs=decoded)
#   loss = 'mean_squared_error' , 'binary_crossentropy', 'mean_absolute_error'
    autoencoder.compile(optimizer='adam', loss='mean_squared_error')
    return autoencoder
autoencoder = build_autoenocder()
autoencoder.summary()
```

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function*

| Model Number | Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|---|
| 2 | 10L,16F (3X3) | 1 | 2 (p=0.2) | 6 | 0.1521/ 0.1498 |
| 3 | 4L,32,32,64,128F (3X3) | 0 | 0 | 0 | 0.0023/ 0.0026 |
| 4 | 4L,16F (3X3) | 2+2 | 0 | 4 | 0.0086/0.00745 |

Here we see that without using convolution volume our accuracy decreases significantly because Conv2D models have inbuilt property to store the image features and there is not much loss of information layer after layer in using convolutions. But when we use Dense layers, they are not that much capable of extracting image features

## 2.3 Dense multi-layer model

CNN Complex Models can successfully boil down a given image into a highly abstracted representation which is easy for predicting. In Dense networks we try to find patterns in pixel values given as input for eg. if pixel number 25 and 26 are greater than a certain value it might belong to a certain class and a few complex variations of the same.

This might easily fail if we can have objects anywhere in the image and not necessarily centered like in the MNIST or to a certain extent also in the Fashion-MNIST data. As we increase the number of hidden layers, the accuracy first increases and then after some point it start decreasing. This is the case happened to our above model.

| Model Number | Dense | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|
| 5 | 3L(2000, 2000, 226800) | 2(p=0.2) | 2 | 0.0756/0.1722 |

### Best Complex CNN Model Architecture 4L,32,32,64,128F (3X3)

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
image_input (InputLayer)     [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 28, 28, 32)        9248
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 28, 28, 64)        18496
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 28, 28, 128)       73856
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 28, 28, 1)         1153
=================================================================
Total params: 103,073
Trainable params: 103,073
Non-trainable params: 0
_________________________________________________________________
```

## 3. Text Reconstruction Application

Applying this autoencoder approach (as just described) to the text data provided as noted

below The data has two sets of images, train and test

```
! git clone https://github.com/iamsivab/Deep-Learning-CNN-AutoEncoder
```

These images contain various styles of text, to which synthetic noise has been added to simulate real-world, messy artifacts. The training set includes the test without the noise (train_cleaned)

I created an algorithm to clean the images in the test set, and report the error as RMSE (root-mean-square error).

Models:
Dense, multi-layer model;
CNN basic model;
CNN complex models.

RMSE Value obtained from my model **is 0.0014 with 25 Epochs**

RMSE Value Obtained from my model is **3.7641e-04 with 100 Epochs and Batch Size=8**

### Preprocessing the data

```
! git clone https://github.com/iamsivab/Deep-Learning-CNN-AutoEncoder
```

```
Cloning into 'Deep-Learning-CNN-AutoEncoder'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 16 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (16/16), done.
```

```
from zipfile import ZipFile
with ZipFile("/content/Deep-Learning-CNN-AutoEncoder/train.zip", 'r') as zip:
    zip.extractall()

from zipfile import ZipFile
with ZipFile("/content/Deep-Learning-CNN-AutoEncoder/test.zip", 'r') as zip:
    zip.extractall()

from zipfile import ZipFile
with ZipFile("/content/Deep-Learning-CNN-AutoEncoder/train_cleaned.zip", 'r') as zip:
    zip.extractall()
```

```python
for img in os.listdir('/content/train'):
    img = load_img('/content/train' +'/'+ img, grayscale=True,target_size=(420,540))
    img = img_to_array(img).astype('float32')/255.
    X.append(img)

for img in os.listdir('/content/train_cleaned'):
    img = load_img('/content/train_cleaned' +'/'+ img, grayscale=True,target_size=(420,540))
    img = img_to_array(img).astype('float32')/255.
    Y.append(img)

for img in os.listdir('/content/test'):
    img = load_img('/content/test' +'/'+ img, grayscale=True,target_size=(420,540))
    img = img_to_array(img).astype('float32')/255.
    Z.append(img)
```

## 3.1 CNN models

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function*

| Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|
| 1L, 8F(3X3) | 0 | 0 | 0 | 0.0423/0.0431 |
| 3L, 8F(3X3) | 0 | 0 | 0 | 0.004/0.0036 |
| 3L, 16F(3X3) | 0 | 0 | 0 | 0.0039/0.0034 |
| 3L, 32F(3X3) | 0 | 0 | 0 | 0.0024/0.0021 |
| 5L, 1F(3X3) | 0 | 0 | 0 | 0.1832/0.1852 |
| 5L , 8F(3X3) | 0 | 0 | 0 | 0.005/0.0045 |
| 5L , 8F(1X1) | 2+2 (2X2) | 0 | 0 | 0.0466/0.0412 |
| 5L , 8F(3X3) | 2+2 (2X2) | 0 | 0 | 0.0054/0.0056 |
| 5L , 8F(5X5) | 2+2 (2X2) | 0 | 0 | 0.009/0.008 |
| 5L , 16F(3X3) | 2+2 (2X2) | 0 | 0 | 0.0023/0.0023 |
| 5L, 16,16,8,8,8F(3X3) | 2+2(2X2) | 0 | 0 | 0.0121/0.0118 |
| 8L , 8F(3X3) | 4+4 (2X2) | 0 | 0 | 0.0257/0.0289 |
| 5L, 16,8,8, 4,4F(3X3) | 2+2 (2X2) | 0 | 0 | 0.0166/0.0172 |

| | | | | |
|---|---|---|---|---|
| 5L, 16,8,8, 4,4F(5X5) | 2+2 (2X2) | 0 | 0 | 0.0185/0.0195 |
| 5L, 16,8,8, 8,8F(3X3) | 2+2 (2X2) | 0 | 0 | 0.0180/0.0163 |

We can see from the above table that as we increase the number of filters, it reduces our loss and thus increases our accuracy. Also, as we decrease the number of hidden layers, till some point our accuracy increases but further decreasing the layers in turn decreases our accuracy.

As we increase the size of the filter our accuracy decreases because when we use large filters, we miss the minute details in the image and hence poor accuracy. Size of filters is purely model dependent. The more the number of channels, more the number of filters used, more are the features learnt, and more is the chances to over-fit and vice-versa

## 3.2 Dense multi-layer model

| Dense | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|
| 3L(2000, 2000, 226800) | 0 | 0 | 0.4011/0.3972 |
| 3L(2000, 2000, 226800) | 2 (p=0.2) | 0 | 0.2453/0.1368 |
| 3L(2000, 2000, 226800) | 2 (p=0.4) | 2 | 0.0791/0.0661 |
| 3L(2000, 2000, 226800) | 2 (p=0.2) | 2 | 0.0556/0.1722 |

Here we see that without using convolution volume our accuracy decreases significantly because Conv2D models have inbuilt property to store the image features and there is not much loss of information layer after layer in using convolutions. But when we use Dense layers, they are not that much capable of extracting image features.

## 3.3 Complex CNN models

*****In all the below tables the last Conv2D layer has been excluded as I have assumed it in all the models. The last Conv2D layer uses sigmoid function while the rest Conv2D layers use relu function*

| Conv2D | MaxPooling2D+ UpSampling2D | Dropout | BatchNormalization | Mean_squared_error Train/Validation |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 4L, 16F(3X3) | 2+2 (2X2) | 1(p=0.3) | 4 | 0.0145/0.0745 |
| 4L, 16F(3X3) | 0 | 0 | 4 | 0.0075/0.0642 |
| 4L, 16,16,32,128F(3X3) | 0 | 0 | 4 | 0.0034/0.0658 |
| 4L, 32,32,64,128F(3X3) | 0 | 0 | 1 | 0.0024/0.0273 |
| 4L, 32,32,64,128F(3X3) | 0 | 0 | 0 | 0.0014/0.0015 |

L = no. of layers
F = no. of filters, e.g. 4F means 4 filters
epoch = 25
batch_size = 16

### Best Model Complex CNN Model 1 with Loss 0.0014

```
##(..1..)## Complex CNN Model 1  ###***The Best***                loss: 0.0012 - val_loss: 0.0014          optimizer='adam', loss='mean_squared_error'
###   After epochs = 100, batch_size = 8 we get::=>  loss:  0.00041 - val_loss: 0.00047
# Lets' define our autoencoder now
def build_autoenocder():
    input_img = Input(shape=(420,540,1), name='image_input')

    x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding = 'same')(input_img)
    x = tf.keras.layers.Conv2D(32, (3, 3), activation='relu', padding = 'same')(x)
    x = tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding = 'same')(x)
    x = tf.keras.layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    decoded = tf.keras.layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
    autoencoder = Model(inputs=input_img, outputs=decoded)
#   loss = 'mean_squared_error' , 'binary_crossentropy', 'mean_absolute_error'
    autoencoder.compile(optimizer='adam', loss='mean_squared_error')
    return autoencoder
autoencoder = build_autoenocder()
autoencoder.summary()
```

For the last / best model for complex CNN model (4L, 32,32,64,128F(3X3)) in last table, if we use epoch =100 and batch_size = 8, then we get **loss = 3.7641e-04, val_loss: 4.5381e-04**

## 4. Results of Text Reconstruction Application

The Best Model is the CNN Complex Model which has (4L, 32,32,64,128F(3X3)) with no MaxPooling2D+UpSampling2D , no Dropout and no BatchNormalization.

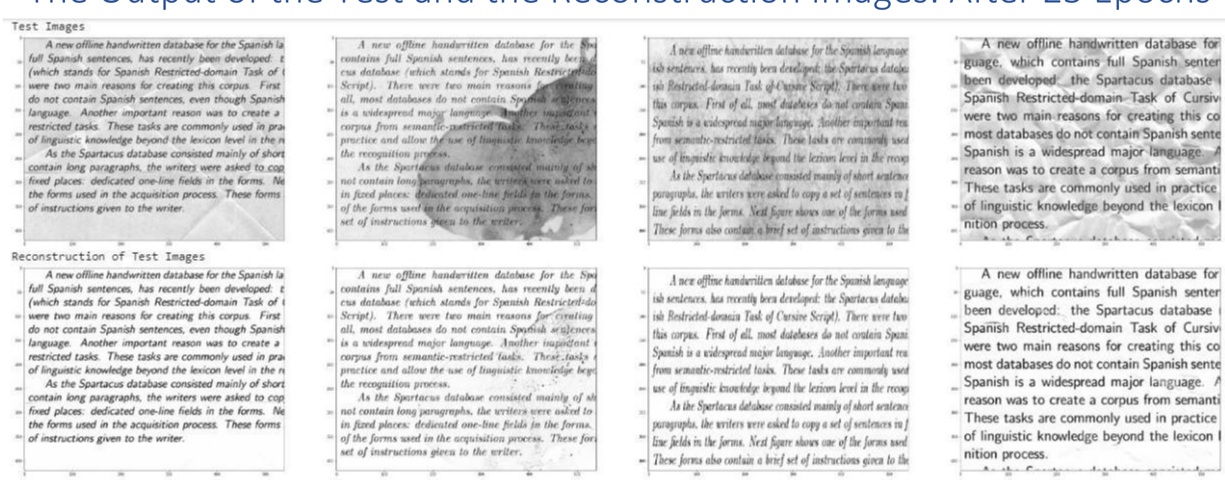
## Architecture of Text Reconstruction Application Model

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
image_input (InputLayer)     [(None, 420, 540, 1)]     0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 420, 540, 32)      320
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 420, 540, 32)      9248
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 420, 540, 64)      18496
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 420, 540, 128)     73856
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 420, 540, 1)       1153
=================================================================
Total params: 103,073
Trainable params: 103,073
Non-trainable params: 0
_________________________________________________________________
```

## Training of Text Reconstruction Application Model

```
Epoch 12/25
9/9 [==============================] - 230s 26s/step - loss: 0.0028 - val_loss: 0.0026
Epoch 13/25
9/9 [==============================] - 234s 26s/step - loss: 0.0027 - val_loss: 0.0029
Epoch 14/25
9/9 [==============================] - 231s 26s/step - loss: 0.0024 - val_loss: 0.0019
Epoch 15/25
9/9 [==============================] - 231s 26s/step - loss: 0.0019 - val_loss: 0.0018
Epoch 16/25
9/9 [==============================] - 232s 26s/step - loss: 0.0019 - val_loss: 0.0023
Epoch 17/25
9/9 [==============================] - 230s 26s/step - loss: 0.0019 - val_loss: 0.0016
Epoch 18/25
9/9 [==============================] - 233s 26s/step - loss: 0.0018 - val_loss: 0.0023
Epoch 19/25
9/9 [==============================] - 230s 26s/step - loss: 0.0019 - val_loss: 0.0015
Epoch 20/25
9/9 [==============================] - 232s 26s/step - loss: 0.0015 - val_loss: 0.0014
Epoch 21/25
9/9 [==============================] - 230s 26s/step - loss: 0.0015 - val_loss: 0.0015
Epoch 22/25
9/9 [==============================] - 230s 26s/step - loss: 0.0014 - val_loss: 0.0013
Epoch 23/25
9/9 [==============================] - 232s 26s/step - loss: 0.0014 - val_loss: 0.0017
Epoch 24/25
9/9 [==============================] - 230s 26s/step - loss: 0.0014 - val_loss: 0.0012
Epoch 25/25
9/9 [==============================] - 233s 26s/step - loss: 0.0014 - val_loss: 0.0015
<tensorflow.python.keras.callbacks.History at 0x7f5d492a0b70>
```

## The Output of the Test and the Reconstruction Images: After 25 Epochs



## 5. References


1. CNN Architectures, a Deep-dive
2. The Quest of Higher Accuracy for CNN Models
3. Home - Keras Documentation
4. [TensorFlow 2.0] Load Images to tensorflow - A Ydobon
5. Keras Autoencoders: Beginner Tutorial
6. Understanding CNN (Convolutional Neural Network)