AutoEncoders

AutoEncoder is an unsupervised Artificial Neural Network that attempts to encode the data by compressing it into the lower dimensions (bottleneck layer or code) and then decoding the data to reconstruct the original input. The bottleneck layer (or code) holds the compressed representation of the input data.

In AutoEncoder the number of output units must be equal to the number of input units since we're attempting to reconstruct the input data. AutoEncoders usually consist of an encoder and a decoder. The encoder encodes the provided data into a lower dimension which is the size of the bottleneck layer and the decoder decodes the compressed data into its original form.

The number of neurons in the layers of the encoder will be decreasing as we move on with further layers, whereas the number of neurons in the layers of the decoder will be increasing as we move on with further layers. There are three layers used in the encoder and decoder in the following example. The encoder contains 32, 16, and 7 units in each layer respectively and the decoder contains 7, 16, and 32 units in each layer respectively. The code size/ the number of neurons in bottle-neck must be less than the number of features in the data.

Before feeding the data into the AutoEncoder the data must definitely be scaled between 0 and 1 using MinMaxScaler since we are going to use sigmoid activation function in the output layer which outputs values between 0 and 1.

When we are using AutoEncoders for dimensionality reduction we'll be extracting the bottleneck layer and use it to reduce the dimensions. This process can be viewed as **feature extraction**.

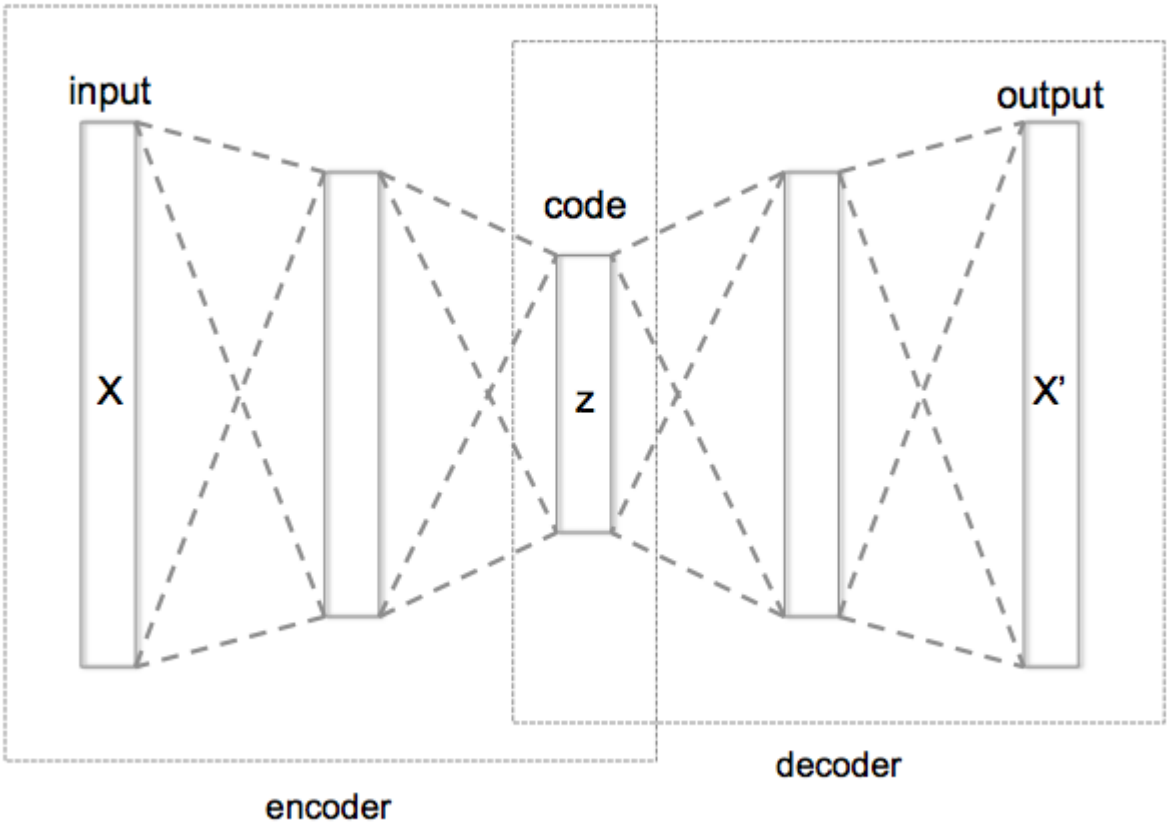
The type of AutoEncoder that we're using is **Deep AutoEncoder**, where the encoder and the decoder are symmetrical. The Autoencoders don't necessarily have a symmetrical encoder and decoder but we can have the encoder and decoder non-symmetrical as well.

Types of AutoEncoders available are,

- Deep Autoencoder
- Sparse Autoencoder
- Under complete Autoencoder
- Variational Autoencoder
- LSTM Autoencoder

Hyperparameters of an AutoEncoder

- Code size or the number of units in the bottleneck layer
- Input and output size, which is the number of features in the data
- Number of neurons or nodes per layer
- Number of layers in encoder and decoder.
- Activation function
- Optimization function



Source: https://commons.wikimedia.org/wiki/File:Autoencoder_structure.png

Applications of AutoEncoders

- Dimensionality reduction
- Anomaly detection
- Image denoising
- Image compression
- Image generation

In this post let us dive deep into dimensionality reduction using autoencoders.

Dimensionality Reduction using AutoEncoders

Import the required libraries and split the data for training and testing.

```
import math
import pandas as pd
import tensorflow as tf
import kerastuner.tuners as kt
import matplotlib.pyplot as plt
from tensorflow.keras import Model
from tensorflow.keras import Sequential
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from tensorflow.keras.losses import MeanSquaredLogarithmicError

# data in google colab
TRAIN_DATA_PATH = '/content/sample_data/california_housing_train.csv'
TEST_DATA_PATH = '/content/sample_data/california_housing_test.csv'
TARGET_NAME = 'median_house_value'

train_data = pd.read_csv(TRAIN_DATA_PATH)
test_data = pd.read_csv(TEST_DATA_PATH)

x_train, y_train = train_data.drop(TARGET_NAME, axis=1),
train_data[TARGET_NAME]
x_test, y_test = test_data.drop(TARGET_NAME, axis=1),
test_data[TARGET_NAME]
```

Scale the dataset using MinMaxScaler.

```
from sklearn.preprocessing import MinMaxScaler

def scale_datasets(x_train, x_test):
  """
  Standard Scale test and train data
  """
  standard_scaler = MinMaxScaler()
  x_train_scaled = pd.DataFrame(
```

```
      standard_scaler.fit_transform(x_train),
      columns=x_train.columns
  )
  x_test_scaled = pd.DataFrame(
      standard_scaler.transform(x_test),
      columns = x_test.columns
  )
  return x_train_scaled, x_test_scaled

x_train_scaled, x_test_scaled = scale_datasets(x_train, x_test)
```

Train the autoencoder with the training data.

```
class AutoEncoders(Model):

  def __init__(self, output_units):

    super().__init__()
    self.encoder = Sequential(
        [
          Dense(32, activation="relu"),
          Dense(16, activation="relu"),
          Dense(7, activation="relu")
        ]
    )

    self.decoder = Sequential(
        [
          Dense(16, activation="relu"),
          Dense(32, activation="relu"),
          Dense(output_units, activation="sigmoid")
        ]
    )

def call(self, inputs):

  encoded = self.encoder(inputs)
  decoded = self.decoder(encoded)
```

```
  return decoded

auto_encoder = AutoEncoders(len(x_train_scaled.columns))

auto_encoder.compile(
    loss='mae',
    metrics=['mae'],
    optimizer='adam'
)

history = auto_encoder.fit(
    x_train_scaled,
    x_train_scaled,
    epochs=15,
    batch_size=32,
    validation_data=(x_test_scaled, x_test_scaled)
)
```

Here we have defined the autoencoder model by subclassing the Model class in Tensorflow and we compile the AutoEncoder model with mean absolute error and adam optimization function. We split the data into batches of 32 and we run it for 15 epochs.

```
Epoch 1/15
532/532 [==============================] - 2s 2ms/step - loss: 0.1551 - mae: 0.1551 - val_loss: 0.0459 - val_mae: 0.0459
Epoch 2/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0417 - mae: 0.0417 - val_loss: 0.0277 - val_mae: 0.0277
Epoch 3/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0273 - mae: 0.0273 - val_loss: 0.0229 - val_mae: 0.0229
Epoch 4/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0187 - mae: 0.0187 - val_loss: 0.0142 - val_mae: 0.0142
Epoch 5/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0138 - mae: 0.0138 - val_loss: 0.0137 - val_mae: 0.0137
Epoch 6/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0128 - mae: 0.0128 - val_loss: 0.0127 - val_mae: 0.0127
Epoch 7/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0121 - mae: 0.0121 - val_loss: 0.0116 - val_mae: 0.0116
Epoch 8/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0116 - mae: 0.0116 - val_loss: 0.0116 - val_mae: 0.0116
Epoch 9/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0113 - mae: 0.0113 - val_loss: 0.0114 - val_mae: 0.0114
Epoch 10/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0113 - mae: 0.0113 - val_loss: 0.0116 - val_mae: 0.0116
Epoch 11/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0110 - mae: 0.0110 - val_loss: 0.0114 - val_mae: 0.0114
Epoch 12/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0108 - mae: 0.0108 - val_loss: 0.0109 - val_mae: 0.0109
Epoch 13/15
532/532 [==============================] - 1s 1ms/step - loss: 0.0107 - mae: 0.0107 - val_loss: 0.0111 - val_mae: 0.0111
Epoch 14/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0108 - mae: 0.0108 - val_loss: 0.0110 - val_mae: 0.0110
Epoch 15/15
532/532 [==============================] - 1s 2ms/step - loss: 0.0104 - mae: 0.0104 - val_loss: 0.0111 - val_mae: 0.0111
```

Get the encoder layer and use the method predict to reduce dimensions in data. Since we have seven hidden units in the bottleneck the data is reduced to seven features.

```
encoder_layer = auto_encoder.get_layer('sequential')
reduced_df = pd.DataFrame(encoder_layer.predict(x_train_scaled))
reduced_df = reduced_df.add_prefix('feature_')
```

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6
0	0.611391	0.233921	1.362540	0.874464	1.767208	0.907174	0.975969
1	0.680573	0.201547	1.322693	0.838978	1.662543	0.735694	0.975585
2	0.993378	0.230347	2.349445	0.938575	2.724568	1.590461	1.249937
3	0.873322	0.036029	1.855744	0.891263	2.467980	1.349111	0.981628
4	0.891184	0.183360	1.930577	0.881793	2.523278	1.286392	1.212686
...	...	...	...	...	...	...	...
16995	4.121347	0.157185	4.347832	0.354799	1.121877	0.422405	1.921186
16996	3.639772	0.147412	3.923980	0.545723	0.600747	0.614113	1.386678
16997	3.559852	0.091495	3.983143	0.716093	0.125736	0.817908	0.960197
16998	3.561868	0.203386	4.058132	0.732039	0.120612	0.812090	1.086610
16999	4.196994	0.082328	4.433511	0.331889	1.223260	0.476652	1.894219

17000 rows × 7 columns

In this way, AutoEncoders can be used to reduce dimensions in data.

References

[1] Applications of Autoencoders
[2] Intro to Autoencoders

[3] Types of Autoencoders

Alexnet won the Imagenet large-scale visual recognition challenge in 2012. The model was proposed in 2012 in the research paper named Imagenet

Classification with Deep Convolution Neural Network by Alex Krizhevsky and his colleagues.

The Alexnet has eight layers with learnable parameters. The model consists of five layers with a combination of max pooling followed by 3 fully connected layers and they use Relu activation in each of these layers except the output layer.

They found out that using the relu as an activation function accelerated the speed of the training process by almost six times. They also used the dropout layers, that prevented their model from overfitting. Further, the model is trained on the Imagenet dataset. The Imagenet dataset has almost 14 million images across a thousand classes.

Let's see the architectural details in this article.

Alexnet Architecture

One thing to note here, since Alexnet is a deep architecture, the authors introduced padding to prevent the size of the feature maps from reducing drastically. The input to this model is the images of size 227X227X3.

Layer	# filters / neurons	Filter size	Stride	Padding	Size of feature map	Activation function
Input	-	-	-	-	227 x 227 x 3	-
Conv 1	96	11 x 11	4	-	55 x 55 x 96	ReLU
Max Pool 1	-	3 x 3	2	-	27 x 27 x 96	-
Conv 2	256	5 x 5	1	2	27 x 27 x 256	ReLU
Max Pool 2	-	3 x 3	2	-	13 x 13 x 256	-
Conv 3	384	3 x 3	1	1	13 x 13 x 384	ReLU
Conv 4	384	3 x 3	1	1	13 x 13 x 384	ReLU
Conv 5	256	3 x 3	1	1	13 x 13 x 256	ReLU
Max Pool 3	-	3 x 3	2	-	6 x 6 x 256	-
Dropout 1	rate = 0.5	-	-	-	6 x 6 x 256	-

Convolution and Maxpooling Layers

Then we apply the first convolution layer with 96 filters of size 11X11 with stride 4. The activation function used in this layer is relu. The output feature map is 55X55X96.

In case, you are unaware of how to calculate the output size of a convolution layer

output= ((Input-filter size)/ stride)+1

Also, the number of filters becomes the channel in the output feature map.

Next, we have the first Maxpooling layer, of size 3X3 and stride 2. Then we get the resulting feature map with the size 27X27X96.

After this, we apply the second convolution operation. This time the filter size is reduced to 5X5 and we have 256 such filters. The stride is 1 and padding 2. The activation function used is again relu. Now the output size we get is 27X27X256.

Again we applied a max-pooling layer of size 3X3 with stride 2. The resulting feature map is of shape 13X13X256.

Now we apply the third convolution operation with 384 filters of size 3X3 stride 1 and also padding 1. Again the activation function used is relu. The output feature map is of shape 13X13X384.

Then we have the fourth convolution operation with 384 filters of size 3X3. The stride along with the padding is 1. On top of that activation function used is relu. Now the output size remains unchanged i.e 13X13X384.

After this, we have the final convolution layer of size 3X3 with 256 such filters. The stride and padding are set to one also the activation function is relu. The resulting feature map is of shape 13X13X256.

So if you look at the architecture till now, the number of filters is increasing as we are going deeper. Hence it is extracting more features as we move deeper into the architecture. Also, the filter size is reducing, which means the initial filter was larger and as we go ahead the filter size is decreasing, resulting in a decrease in the feature map shape.

Next, we apply the third max-pooling layer of size 3X3 and stride 2. Resulting in the feature map of the shape 6X6X256.

Fully Connected and Dropout Layers

Layer	# filters / neurons	Filter size	Stride	Padding	Size of feature map	Activation function
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
Dropout 1	rate = 0.5	-	-	-	6 x 6 x 256	-
Fully Connected 1	-	-	-	-	4096	ReLU
Dropout 2	rate = 0.5	-	-	-	4096	-
Fully Connected 2	-	-	-	-	4096	ReLU
Fully Connected 3	-	-	-	-	1000	Softmax

After this, we have our first dropout layer. The drop-out rate is set to be 0.5.

Then we have the first fully connected layer with a relu activation function. The size of the output is 4096. Next comes another dropout layer with the dropout rate fixed at 0.5.

This followed by a second fully connected layer with 4096 neurons and relu activation.

Finally, we have the last fully connected layer or output layer with 1000 neurons as we have 10000 classes in the data set. The activation function used at this layer is Softmax.

This is the architecture of the Alexnet model. It has a total of 62.3 million learnable parameters.

End Notes

To quickly summarize the architecture that we have seen in this article.

- It has 8 layers with learnable parameters.
- The input to the Model is RGB images.
- It has 5 convolution layers with a combination of max-pooling layers.
- Then it has 3 fully connected layers.
- The activation function used in all layers is Relu.

- It used two Dropout layers.
- The activation function used in the output layer is Softmax.
- The total number of parameters in this architecture is 62.3 million.

So this was all about Alexnet. If you have any questions let me know in the comments below!

The Architecture of Lenet-5

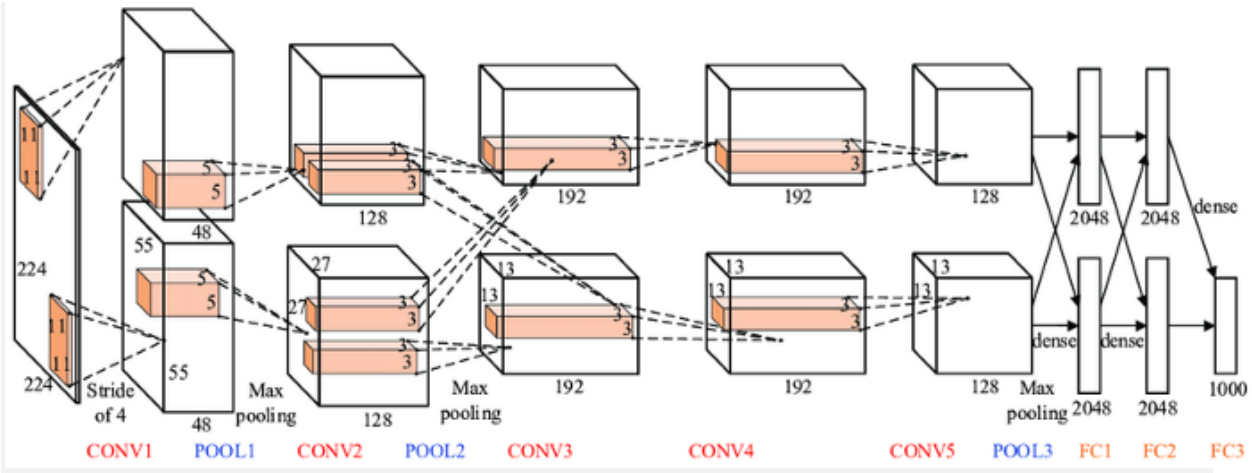
Build VGG Net from Scratch with Python!

Introduction to Convolutional Neural Networks (CNN)

Architecture of Alexnet

2. AlexNet

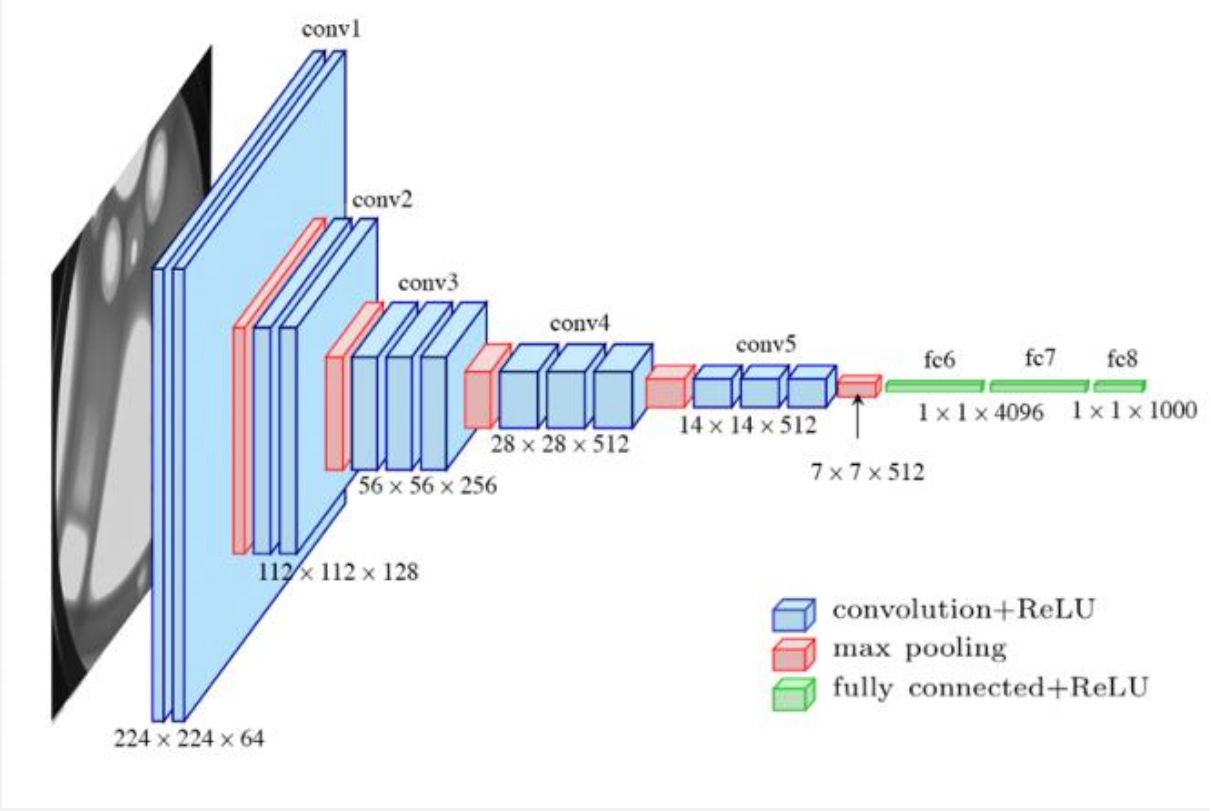
This network was very similar to LeNet-5 but was deeper with 8 layers, with more filters, stacked convolutional layers, max pooling, dropout, data augmentation, ReLU and SGD. AlexNet was the winner of the ImageNet ILSVRC-2012 competition, designed by Alex Krizhevsky, Ilya Sutskever and Geoffery E. Hinton. It was trained on two Nvidia Geforce GTX 580 GPUs, therefore, the network was split into two pipelines. AlexNet has 5 Convolution layers and 3 fully connected layers. AlexNet consists of approximately 60 M parameters. A major drawback of this network was that it comprises of too many hyper-parameters. A new concept of Local Response Normalization was also introduced in the paper. Refer to the original paper.



AlexNet Architecture

3. VGG-16 Net

The major shortcoming of too many hyper-parameters of AlexNet was solved by VGG Net by replacing large kernel-sized filters (11 and 5 in the first and second convolution layer, respectively) with multiple 3×3 kernel-sized filters one after another. The architecture developed by Simonyan and Zisserman was the 1st runner up of the Visual Recognition Challenge of 2014. The architecture consist of 3*3 Convolutional filters, 2*2 Max Pooling layer with a stride of 1, keeping the padding same to preserve the dimension. In total, there are 16 layers in the network where the input image is RGB format with dimension of 224*224*3, followed by 5 pairs of Convolution(filters: 64, 128, 256,512,512) and Max Pooling. The output of these layers is fed into three fully connected layers and a softmax function in the output layer. In total there are 138 Million parameters in VGG Net.



VGG-16 Architecture

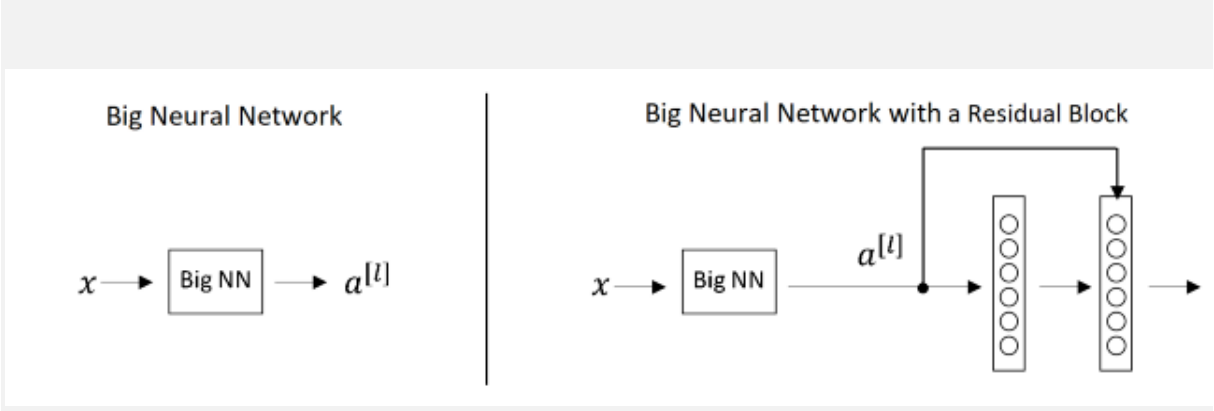
Drawbacks of VGG Net:

1. Long training time
2. Heavy model
3. Computationally expensive
4. Vanishing/exploding gradient problem

4. ResNet

ResNet, the winner of ILSVRC-2015 competition are deep networks of over 100 layers. Residual networks are similar to VGG nets however with a sequential approach they also use “Skip connections” and “batch normalization” that helps to train deep layers without hampering the performance. After VGG Nets, as CNNs were going deep, it was becoming hard to train

them because of vanishing gradients problem that makes the derivate infinitely small. Therefore, the overall performance saturates or even degrades. The idea of skips connection came from highway network where gated shortcut connections were used.



Normal Deep Networks vs Networks with skip connections

For the above figure for network with skip connection, a[l+2]=g(w[l+2]a[l+1]+ a[l])

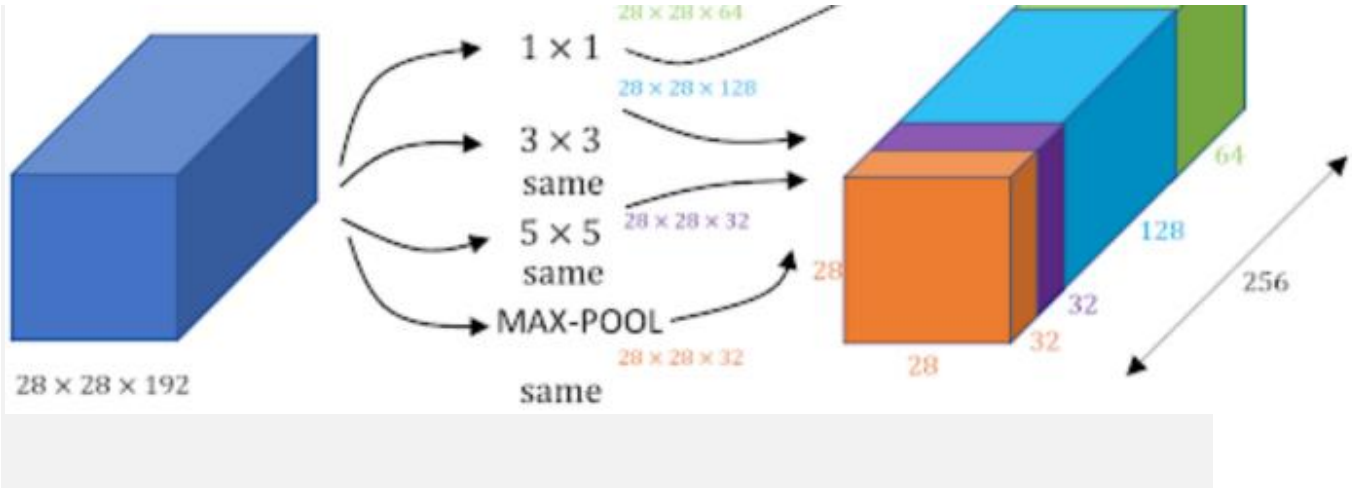
Lets say for some reason, due to weight decay w[l+2] becomes 0, therefore, a[l+2]=g(a[l])

Hence, the layer that is introduced doesnot hurt the performance of the neural network. This the reason, increasing layers doesn't decrease the training accuracy as some layers may make the result worse. The concept of skip connections can also be seen in LSTMs.

5. Inception Net

Inception network also known as GoogleLe Net was proposed by developers at google in "Going Deeper with Convolutions" in

2014. The motivation of InceptionNet comes from the presence of sparse features Salient parts in the image that can have a large variation in size. Due to this, the selection of right kernel size becomes extremely difficult as big kernels are selected for global features and small kernels when the features are locally located. The InceptionNets resolves this by stacking multiple kernels at the same level. Typically it uses 5*5, 3*3 and 1*1 filters in one go. For better understanding refer to the image below:

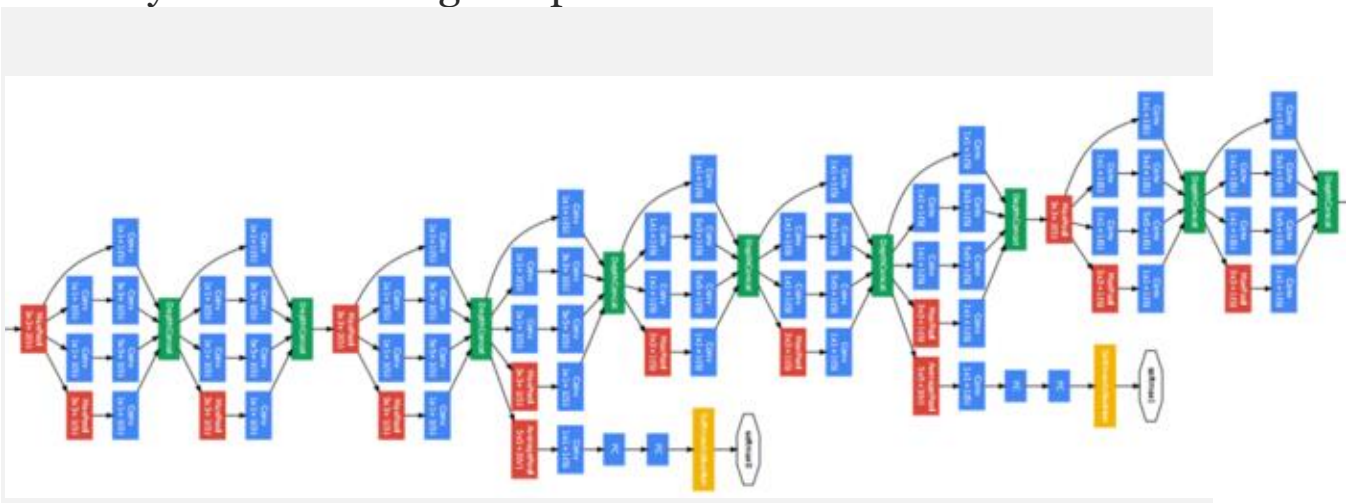


Inception Module of GoogleLe Net

Note: Same padding is used to preserve the dimension of the image.

As we can see in the image, three different filters are applied in the same level and the output is combined and fed to the next layer. The combination increases the overall number of channels in the output. The problem with this structure was the number of parameter (120M approx.) that increases the computational cost. Therefore, 1*1 filters were used before feeding the image directly to these filters that act as a bottleneck

and reduces the number of channels. Using 1*1 filters, the parameter were reduced to 1/10 of the actual. GoogLeNet has 9 such inception modules stacked linearly. It is 22 layers deep (27, including the pooling layers). It uses global average pooling at the end of the last inception module. Inception v2 and v3 were also mentioned in the same paper that further increased the accuracy and decreasing computational cost.



Several Inception modules are linked to form a dense network

Side branches can be seen in the network which predicts output in order to check the shallow network performance at lower levels.

A gentle introduction to batch normalization

In the rise of deep learning, one of the most important ideas has been an algorithm called ***batch normalization*** (also known as ***batch norm***).

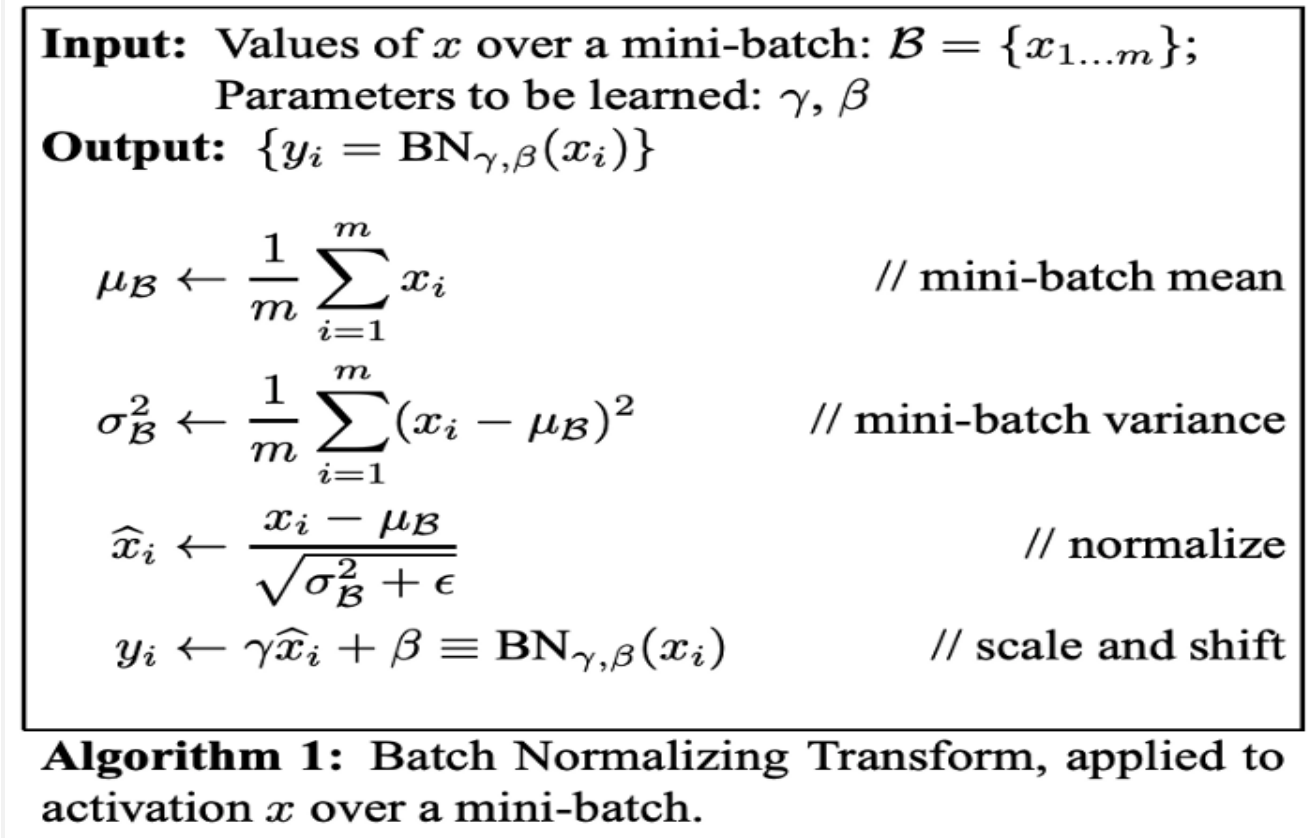
Batch normalization is a technique for training very deep neural networks that standardizes the

inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

By Jason Brownlee

Batch normalization can be implemented during training by calculating the mean and standard deviation of each input variable to a layer per mini-batch and using these statistics to perform the standardization.

Formally, the batch normalization algorithm [1] is defined as:



Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

From the original paper: [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift]

In the algorithm, `B` is used to denote a mini-batch of size `m` of the entire training set. The mean and variance of `B` could thus be calculated as:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \text{ and } \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

For a layer with *d*-dimensional input, `x = (x_1, …, x_d)`, each dimension of its input can be normalized (re-centered and rescaled) separately. Thus, the normalization for a *d*-dimensional input can be calculated as:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)^2} + \epsilon}}, \text{ where } k \in [1, d] \text{ and } i \in [1, m];$$

ε is added in the denominator for numerical stability and is an arbitrarily small constant.

And finally, to restore the representation power of the network, a transformation step is defined as:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

where the parameter β and γ are subsequently learned in the optimization process.

The benefits of batch normalization are [2]:

- **A deep neural network can be trained faster:** Although each training iteration will be slower because of the extra normalization calculation during the forward pass and the additional hyperparameters to train during backpropagation, it should converge much more quickly; thus, training should be faster overall.
- **Higher learning rate:** Gradient descent generally requires small learning rates for the network to converge. As networks become deeper, gradients become smaller during backpropagation and thus require even more iterations. Using batch normalization allows much higher learning rates, thereby increasing the speed of training.
- **Easier to initialize weight:** Weight initialization can be difficult, particularly when creating deeper networks. Batch normalization reduces the sensitivity to the initial starting weights.