Deeper Networks for Image Classification

Author: Bikash Kumar Tiwary

Student Id: 190778031

1. Introduction

Previous ways to deal with object recognition was to make essential use of machine learning strategies, by collecting large datasets, powerful models can be learned using better methods and prevent overfitting [1]. Deep convolutional neural networks [1] have gone through a progression of breakthroughs for image classification [2]. Deep convolution networks normally incorporate low/mid/high level highlights [3] and classifiers in an end to end multilayer style, and the "levels" of features can be advanced by the quantity of stacked layers (depth). Evidence [4] reveals that depth is of crucial significance, and the outcome on the challenging ImageNet dataset all exploit very deep models [4].

Despite the appealing characteristics of convolution networks, and efficiency of their architecture. They have still been restrictively costly to apply in large scale to high resolution images. Fortunately, current GPUs, paired with exceptionally optimized implementation of 2D convolution, are incredible enough to facilitate the training of large CNNs, and ongoing datasets, for example, ImageNet contain enough labeled examples to train models without serious overfitting [1].

The specific contribution on this report are as follows: Implementation of two deep networks which includes VGG and ResNet. Training the deep networks on the Datasets MNIST and CIFAR10. Section 2, review the critical analysis/ related work on the deep networks. Section 3, describes the Models implemented (I.e. VGG-16 and ResNet50) and their Architecture. Section 4, reviews the experiments done with the two models VGG and ResNet on MNIST dataset, where the Datasets and the testing results are discussed. Further evaluation is made with CIFAR10 dataset on the two deep networks.

2. Critical Analysis

2.1 VGG

VGG base concept.

VGG is a convolutional neural network model developed by Simonyan and Zisserman [4]. All VGG configuration follow a conventional configuration and only differ in the depth. As shown below in Figure 1, From 11 weight convolution layers (8 conv. And 3 FC layers) in network A (VGG-11) to 19 weight conv. Layers (16 conv. And 3 FC layers) in the network E (VGG-19). The width of conv layers (quantity of channels) is somewhat small, beginning from 64 in primary layer and then usually expanding by a factor of 2 after every maximum pooling layer, until it reaches 512 [4].

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

Figure1: VGG

Although depth increases, total parameters are loosely conserved compared to a shallower CNN with larger receptive fields.

Working of VGG.

The optimization of the proposed network is carried out by using stochastic gradient method(sgd) with batch size set as 128. In addition, a dropout rate of 0.5 is used to regularize the network parameters during the training process. The training start with default learning rate of 0.01 and momentum value of 0.9. An input image (Greyscale) is fed from the dataset (i.e. MNIST) as an input until the probability of output layer (last layer) of network is calculated.

The stochastic gradient decent method optimizes and finds the parameters of connected layers that minimize the prediction SoftMax-log-loss for image classification. Meanwhile the convolutional layers parameters are unchanged. In other words, fully connect layers are optimized to predict the image while not changing the parameters of convolutional layers which were trained and optimized for image classification.

Application of VGG.

VGG convolutional neural network is mainly used for image processing, image classification, segmentation and also for other auto correlated data.

Problem.

Problem: too many weight parameters, Models are very heavy which also leads to long inference time.

Main advantage is the architecture is simple and easy to implement.

2.2 ResNet

ResNet base concept.

ResNet, so-called Residual Neural Network by Kaiming He et al [5], introduced a novel architecture with "skip connections" and features heavy batch normalization. Such skip connections are known as gated recurrent units or gated units and has similarity to recent successful elements applied to RNNs. This technique can train neural network with 152 layers while still have lower complexity than VGGNet.

ResNet has residual connections.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\ 3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\ 3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\ 3\times3, 64\\ 1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\ 3\times3, 64\\ 1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\ 3\times3, 64\\ 1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\ 3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\ 3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\ 3\times3, 128\\ 1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\ 3\times3, 128\\ 1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\ 3\times3, 128\\ 1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\ 3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\ 3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\ 3\times3, 256\\ 1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\ 3\times3, 256\\ 1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\ 3\times3, 256\\ 1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\ 3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\ 3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\ 3\times3, 512\\ 1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\ 3\times3, 512\\ 1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\ 3\times3, 512\\ 1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure2: ResNet

As shown above, in figure 2 all the ResNet configuration follows a similar configuration, only differs in depth of building blocks (shown in brackets). Ranging from 18 layers (ResNet18) all the way to 152 layers (ResNet152).

Working of ResNet.

For residual block, adopt batch normalization (BN) right after each convolution and before activation. Initiate the weights and train all residual nets from scratch. Use Adam optimizer with batch size of 128 with default learning rate and momentum. Avoid using dropout. And, add shortcut.

Typically, ResNet models are implemented with double or triple layer skips that contains ReLU and batch normalization in between.

Application of ResNet.

Residual network model is the winner of ImageNet challenge in 2015. Mainly used to image classification, image processing.

Problem.

Problem: (i) Vanishing gradient, cost function (error function) gradient value backpropagation diminishing. (ii) Degradation problem.

Advantage: accelerate speed of training, reduce effect of vanishing gradient problem, increasing the depth of network results in less extra parameters.

3. Method/Model Description

3.1 Model Architecture

(i) VGG-16

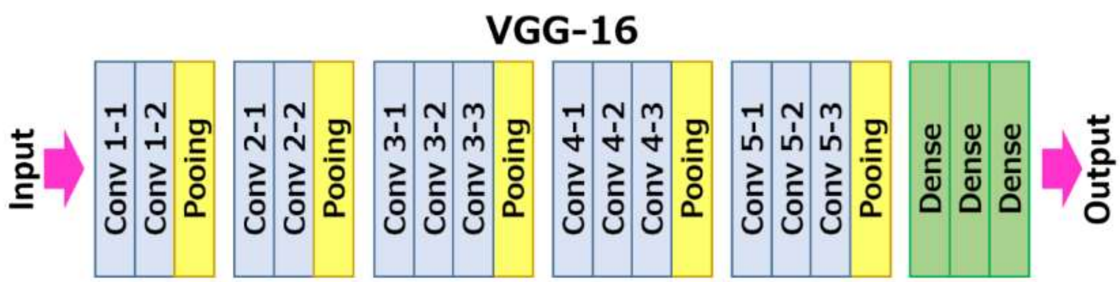


Figure 3

VGG-16 is a convolutional neural network architecture, it consists of 16 layers. Its layers consist of convolutional layers, Max Pooling layers, Fully Connected layers. There are 13 convolutional layers, 5 Max pooling and 3 dense layers (Shown in Figure3).

VGG16 CNN Model

```
[ ] model = Sequential()

    # Convolutional Layer1 and Pooling Layer1
    model.add(ZeroPadding2D((1,1), input_shape=(IMG_ROWS, IMG_COLS, IMG_CHANNELS)))
    model.add(Conv2D(filters = 64, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 64, kernel_size = 3, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    # Convolutional Layer2 and Pooling Layer2
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 128, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 128, kernel_size = 3, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    # Convolutional Layer3 and Pooling Layer3
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 256, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 256, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 256, kernel_size = 3, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    # Convolutional Layer4 and Pooling Layer4
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

    # Convolutional Layer5 and Pooling Layer5
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    model.add(ZeroPadding2D((1,1)))
    model.add(Conv2D(filters = 512, kernel_size = 3, activation='relu'))
    #model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))  # Uncomment this when u
    model.add(MaxPooling2D(pool_size=(1,1), strides=(2,2)))

    # FC Layers
    model.add(Flatten())
    #top layer of the VGG net
    model.add(Dense(units = 4096, activation='relu')); model.add(Dropout(0.5))
    model.add(Dense(units = 4096, activation='relu')); model.add(Dropout(0.5))
    model.add(Dense(NB_CLASSES, activation='softmax'))
    model.summary()
```

Shown above is a VGG-16 code snippet for a sequential model, Two Convolution layer 1(Conv1) with 64 filters, Two Conv2 with 128 filters, Three Conv3 has 256filters, Three Conv4 layers with 256 filters and Three Conv5 layers with 516 filters. Followed by FC layers.

Kernel Size of 3x3, ReLU Activation and similar paddings. MaxPooling with pool_size and Stride of 2.

(ii) ResNet50

ResNet50 is a convolutional neural network architecture, it consists of 50 layers. Its layers consist of convolutional layers, Average Pooling layers, Fully Connected layer.

```
# Residual Block
class ResidualBlock(Model):
    def __init__(self, channel_in = 64, channel_out = 256):
        super().__init__()

        channel = channel_out // 4
        # Batch Normalization after Concolution layer
        self.conv1 = Conv2D(channel, kernel_size = (1, 1), padding = "same")
        self.bn1 = BatchNormalization()
        self.av1 = Activation(tf.nn.relu)
        self.conv2 = Conv2D(channel, kernel_size = (3, 3), padding = "same")
        self.bn2 = BatchNormalization()
        self.av2 = Activation(tf.nn.relu)
        self.conv3 = Conv2D(channel_out, kernel_size = (1, 1), padding = "same")
        self.bn3 = BatchNormalization()
        self.shortcut = self._shortcut(channel_in, channel_out)
        self.add = Add()
        self.av3 = Activation(tf.nn.relu)

    def call(self, x):
        h = self.conv1(x)
        h = self.bn1(h)
        h = self.av1(h)
        h = self.conv2(h)
        h = self.bn2(h)
        h = self.av2(h)
        h = self.conv3(h)
        h = self.bn3(h)
        shortcut = self.shortcut(x)
        h = self.add([h, shortcut])
        y = self.av3(h)
        return y

    # Shortcut
    def _shortcut(self, channel_in, channel_out):
        if channel_in == channel_out:
            return lambda x : x
        else:
            return self._projection(channel_out)

    def _projection(self, channel_out):
        return Conv2D(channel_out, kernel_size = (1, 1), padding = "same")
```

The above code snippet, describes the residual block of Residual Network. This residual block consists of 3 convolutional layers followed by Batch Normalization, before Activation with shortcut. No dropout is used.

```python
# ResNet
class ResNet50(Model):
    def __init__(self, input_shape, output_dim):
        super().__init__()

        self._layers = [
            # convolution Layer 1 (conv1)
            Conv2D(64, input_shape = input_shape, kernel_size = (7, 7), strides=(2, 2), padding = "same"),
            BatchNormalization(),
            Activation(tf.nn.relu),
            # convolution Layer 2 (conv2_x)
            MaxPool2D(pool_size = (3, 3), strides = (2, 2), padding = "same"),
            ResidualBlock(64, 256),
            [
                ResidualBlock(256, 256) for _ in range(2)
            ],
            # convolution Layer 3 (conv3_x)
            Conv2D(512, kernel_size = (1, 1), strides=(2, 2)),
            [
                ResidualBlock(512, 512) for _ in range(4)
            ],
            # convolution Layer 4 (conv4_x)
            Conv2D(1024, kernel_size = (1, 1), strides=(2, 2)),
            [
                ResidualBlock(1024, 1024) for _ in range(6)
            ],
            # convolution Layer 5 (conv5_x)
            Conv2D(2048, kernel_size = (1, 1), strides=(2, 2)),
            [
                ResidualBlock(2048, 2048) for _ in range(3)
            ],
            # Average Pooling and FC1000 layer
            GlobalAveragePooling2D(),
            Dense(1000, activation = tf.nn.relu),
            Dense(output_dim, activation = tf.nn.softmax)
        ]

    def call(self, x):
        for layer in self._layers:
            if isinstance(layer, list):
                for l in layer:
                    x = l(x)
            else:
                x = layer(x)
        return x

model = ResNet50((28, 28, 1), 10)
model.build(input_shape = (None, 28, 28, 1))
model.summary()
```

3

4

6

3

7×7, 64, stride 2	
3×3 max pool, stri	
1×1, 64 3×3, 64 1×1, 256	×3
1×1, 128 3×3, 128 1×1, 512	×4
1×1, 256 3×3, 256 1×1, 1024	×6
1×1, 512 3×3, 512 1×1, 2048	×3

The above code snippet describes the ResNet50 model, the first layer is a conv with 64 filters, 7x7 kernel, stride 2. After, MaxPool is applied with pool size=3 and stride=2.

Conv 2_x consists of 3 residual blocks. And with 256 Filter.

Conv3_x consists of 1 Convolutional layer and 4 residual blocks. And with 512 Filters.

Conv4_x consists of 1 Convolutional layer and 6 residual blocks. And with 1024 Filters.

Conv5_x consists of 1 Convolutional layer and 3 residual blocks. And with 2048 Filters.

And Followed by Average Pooling, FC-1000 and softmax. As shown in Figure 2.

4. Experiments

4.1 Datasets

MNIST Dataset.

This was the Primary dataset used while implementing VGG-16 and ResNet50 models. MNIST database consists of handwritten digits, it consists of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digit has been size-normalized and centered in a fixed-size image.

```
X_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

MNIST consist of 28x28 pixels images, 1 channel (Greyscale) and 10classes.

CIFAR10 Dataset.

CIFAR10 is an extra dataset used to evaluate the models. CIFAR10 consists of 60,000 RGB images. 50,000 training images and 10,000 test image examples.

```
X_train shape: (50000, 32, 32, 3)
50000 train samples
10000 test samples
```

CIFAR10 consist of size 32x32 pixel images, 3 channels and 10 classes.

4.2 Testing results

<u>VGG-16 (MNIST dataset):</u>

LOAD DATA:

```
IMG_CHANNELS = 1; IMG_ROWS = 28; IMG_COLS = 28;

#constant
BATCH_SIZE = 128; NB_EPOCH = 50; NB_CLASSES = 10; VERBOSE = 1; VALIDATION_SPLIT = 0.2;

#load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], IMG_ROWS, IMG_COLS, 1)
X_test = X_test.reshape(X_test.shape[0], IMG_ROWS, IMG_COLS, 1)
```

TRANING SETTINGS:

```
start_time = time.time()

#==============================================================================================
## Training

model.compile(loss= tf.keras.losses.CategoricalCrossentropy(), optimizer='sgd', metrics=['accuracy'])


history = model.fit(X_train, Y_train, batch_size=BATCH_SIZE,
                    epochs=NB_EPOCH, validation_split=VALIDATION_SPLIT, verbose=VERBOSE)

#==============================================================================================
```

TRAINING Accuracy:

```
Train on 48000 samples, validate on 12000 samples
Epoch 1/50
48000/48000 [==============================] - 44s 919us/step - loss:
2.2995 - accuracy: 0.1151 - val_loss: 2.2944 - val_accuracy: 0.1060
Epoch 2/50
48000/48000 [==============================] - 36s 759us/step - loss:
2.2376 - accuracy: 0.2045 - val_loss: 1.4387 - val_accuracy: 0.5782
Epoch 3/50
48000/48000 [==============================] - 36s 760us/step - loss:
0.7218 - accuracy: 0.7568 - val_loss: 0.2183 - val_accuracy: 0.9333
.

.

.

Epoch 49/50
48000/48000 [==============================] - 36s 758us/step - loss:
2.6556e-05 - accuracy: 1.0000 - val_loss: 0.0797 - val_accuracy: 0.9891
Epoch 50/50
48000/48000 [==============================] - 36s 758us/step - loss:
2.4017e-05 - accuracy: 1.0000 - val_loss: 0.0789 - val_accuracy: 0.9894
```

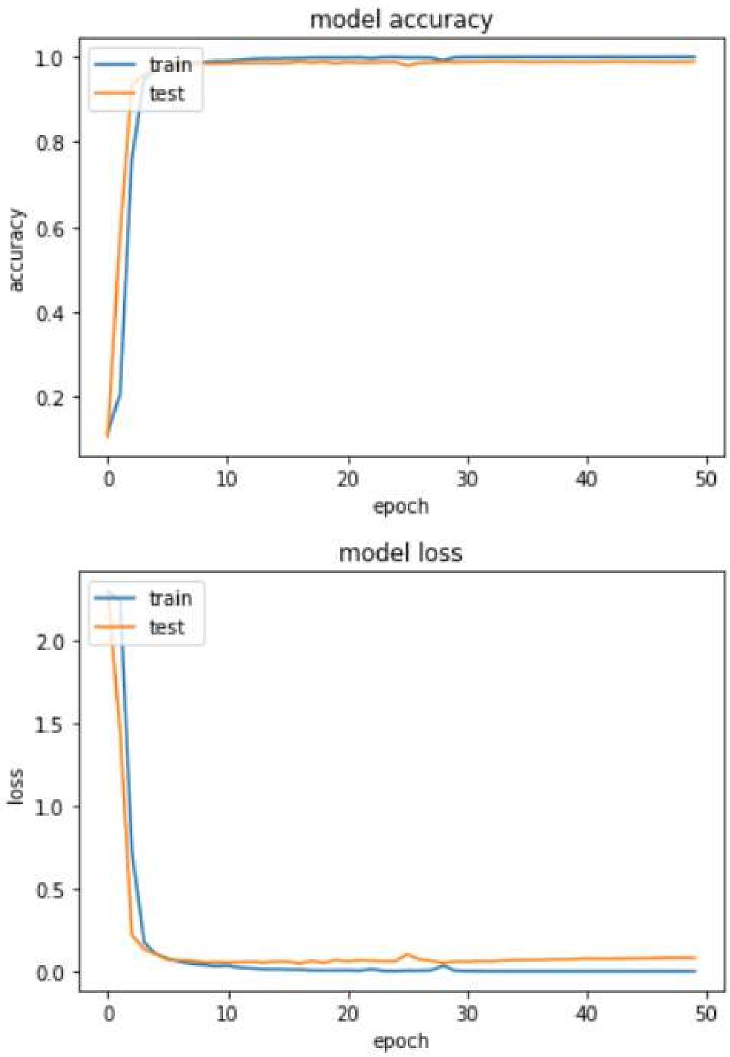
TEST SETTINGS:

```
#==============================================================================================
## Testing
print('Testing...')
score = model.evaluate(X_test, Y_test, batch_size=BATCH_SIZE, verbose=VERBOSE)
print("\nTest score:", score[0]); print('Test accuracy:', score[1]);
```

TEST Accuracy:

```
Testing...
10000/10000 [==============================] - 2s 191us/step

Test score: 0.06720940882283283
      Test accuracy: 0.989799976348877
```

PLOT:




```
Execution Time 1861.9940462112427 seconds:
```

As shown by the environment setting and the execution output, VGG-16 model has achieved training accuracy of 98.94% (validation accuracy) and Test accuracy of 98.97% on Epoch 50.

<u>ResNet50(MNIST Dataset)</u>

LOAD DATA:

```python
# Load MNIST Dataset
dataset, info = tfds.load('mnist', as_supervised = True, with_info = True)
dataset_test, dataset_train = dataset['test'], dataset['train']
```

Loss and Optimizer:

```python
# Categorial CrossEntropy
loss_object = tf.keras.losses.SparseCategoricalCrossentropy()
# Adam
optimizer = tf.keras.optimizers.Adam()
```

TRANING and TEST SETTINGS:

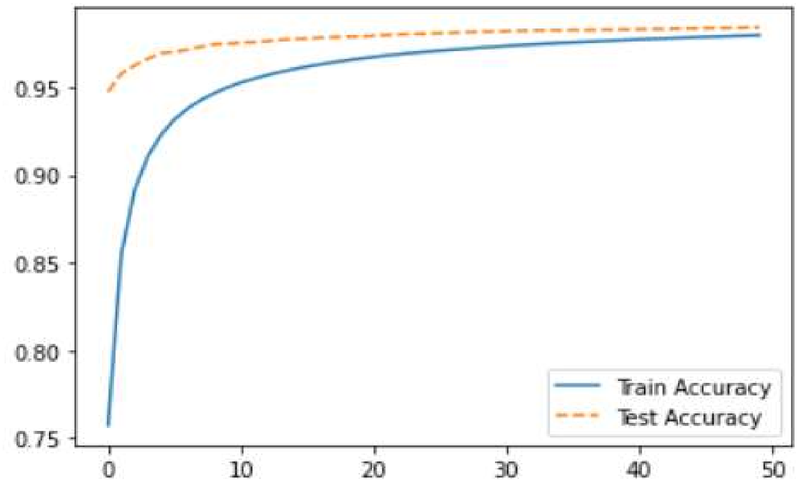
```python
# Train Loss and Accuracy
train_loss = tf.keras.metrics.Mean(name = 'train_loss')
train_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name = 'train_accuracy')
# Test Loss and Accuracy
test_loss = tf.keras.metrics.Mean(name = 'test_loss')
test_accuracy = tf.keras.metrics.SparseCategoricalAccuracy(name = 'test_accuracy')


@tf.function
def train_step(image, label):
    with tf.GradientTape() as tape:
        predictions = model(image)
        loss = loss_object(label, predictions)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

    train_loss(loss)
    train_accuracy(label, predictions)

@tf.function
def test_step(image, label):
    predictions = model(image)
    loss = loss_object(label, predictions)

    test_loss(loss)
    test_accuracy(label, predictions)
```

```
num_epoch = 50
start_time = time.time()

train_accuracies = []
test_accuracies = []

for epoch in range(num_epoch):
    for image, label in dataset_train:
        for _image, _label in datagen.flow(image, label, batch_size = batch_size):
            train_step(_image, _label)
            break

    for test_image, test_label in dataset_test:
        test_step(test_image, test_label)

    train_accuracies.append(train_accuracy.result())
    test_accuracies.append(test_accuracy.result())
```

TRAINING and TEST Accuracy:

```
Epoch 1, Loss: 0.6807350516319275, Accuracy: 75.73833465576172, Test
Loss: 0.17914921045303345, Test Accuracy: 94.76000213623047,
spent_time: 0.8261510252952575 min

Epoch 2, Loss: 0.4209822118282318, Accuracy: 85.50833129882812, Test
Loss: 0.14059126377105713, Test Accuracy: 95.79499816894531,
spent_time: 1.4815698663393657 min

Epoch 3, Loss: 0.3227335810661316, Accuracy: 89.14888763427734, Test
Loss: 0.12302588671445847, Test Accuracy: 96.26000213623047,
spent_time: 2.137596619129181 min
.

.

.

Epoch 49, Loss: 0.06740095466375351, Accuracy: 97.95609283447266, Test
Loss: 0.054914094507694244, Test Accuracy: 98.41510009765625,
spent_time: 32.57619661887487 min

Epoch 50, Loss: 0.0666566714644432, Accuracy: 97.97980499267578, Test
Loss: 0.05460898578166962, Test Accuracy: 98.42240142822266,
spent_time: 33.23585059245428 min
```

PLOT:



As shown by the environment setting and the execution output, ResNet50 model has achieved training accuracy of 97.97% and Test accuracy of 98.42% on Epoch 50.

<u>Comparison between VGG-16 and ResNet50 accuracy results (MNIST Dataset):</u>

Metric	**Training**	**Testing**
VGG-16	98.94%	98.97%
ResNet50	97.97%	98.42%

5. Conclusion

The outcome shows that both VGG-16 and ResNet50 deep convolutional neural networks are capable of achieving High accuracy results on mnist dataset using purely supervised learning. To keep the experiment simple, both the models were tested with mnist datasets to check the model performance.

Compare to VGG-16 model results, ResNet50 model has an accelerated speed of training and better early epochs results.

For future work, it will be interesting to check the performance of both vgg-16 and resnet50 model on a highly challenging dataset like ImageNet. Alternatively, increasing the layers to upgrade to vgg-19 and ResNet152 models and observing the performance with different types of datasets. Orelse, checking model performance by introducing or removing a layer.

6. Reference

[1] Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

[2] Rawat, W. and Wang, Z., 2017. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, *29*(9), pp.2352-2449.

[3] Zeiler, M.D. and Fergus, R., 2014, September. Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818-833). Springer, Cham.

[4] Simonyan, K. and Zisserman, A., 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[5] He, K., Zhang, X., Ren, S. and Sun, J., 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

7. Appendices:

<u>Residual Block:</u>

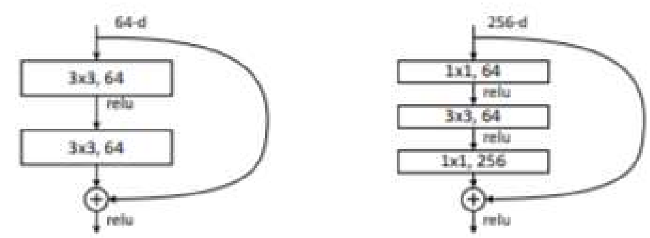


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a "bottleneck" building block for ResNet-50/101/152.

<u>VGG-16 on CIFAR10 dataset:</u>

LOAD DATA

```python
# CIFAR_10 is set of 60K images 32x32 pixels, 3 channels
IMG_CHANNELS = 3; IMG_ROWS = 32; IMG_COLS = 32;

#constant
BATCH_SIZE = 128; NB_EPOCH = 50; NB_CLASSES = 10; VERBOSE = 1; VALIDATION_SPLIT = 0.2;

#load dataset
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

```python
model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))  # Uncomment this when using CIFAR10 DataSet and comment the MaxPooling below
#model.add(MaxPooling2D(pool_size=(1,1), strides=(2,2)))
```

Because for 28x28 mnist data pixel size, the maxpooling of 5th conv layer with pool size = 2 returns negative, so for mnist the last max pooling has poolsize =1. In case of Cifar10 pool size = 2 is used.

TRANING SETTINGS

```python
start_time = time.time()

#==================================================================================================
## Training

model.compile(loss= tf.keras.losses.CategoricalCrossentropy(), optimizer='sgd', metrics=['accuracy'])


history = model.fit(X_train, Y_train, batch_size=BATCH_SIZE,
                    epochs=NB_EPOCH, validation_split=VALIDATION_SPLIT, verbose=VERBOSE)

#==================================================================================================
```

TEST SETTINGS

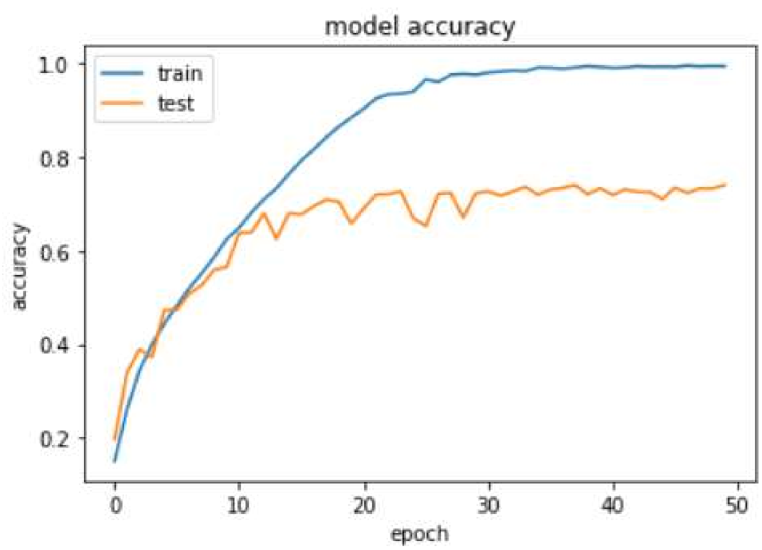
```python
#==================================================================================================
## Testing
print('Testing...')
score = model.evaluate(X_test, Y_test, batch_size=BATCH_SIZE, verbose=VERBOSE)
print("\nTest score:", score[0]); print('Test accuracy:', score[1]);
```

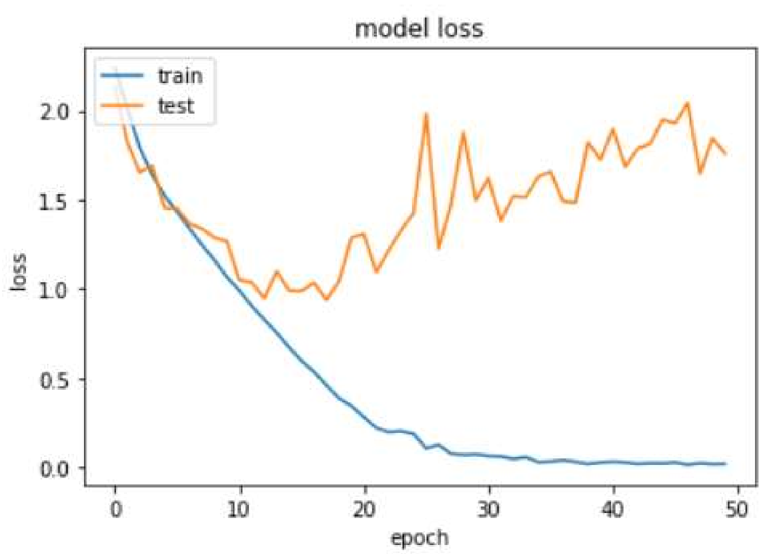
TRAINING and TEST Accuracy

```
Train on 40000 samples, validate on 10000 samples
Epoch 1/50
40000/40000 [==============================] - 15s 386us/step - loss:
2.2409 - accuracy: 0.1509 - val_loss: 2.1434 - val_accuracy: 0.1974
Epoch 2/50
40000/40000 [==============================] - 15s 375us/step - loss:
2.0155 - accuracy: 0.2613 - val_loss: 1.8248 - val_accuracy: 0.3392
Epoch 3/50
40000/40000 [==============================] - 15s 374us/step - loss:
1.7930 - accuracy: 0.3437 - val_loss: 1.6543 - val_accuracy: 0.3895
.
```

```
.

.

Epoch 49/50
40000/40000 [==============================] - 15s 385us/step - loss:
0.0189 - accuracy: 0.9940 - val_loss: 1.8455 - val_accuracy: 0.7324
Epoch 50/50
40000/40000 [==============================] - 15s 377us/step - loss:
0.0200 - accuracy: 0.9937 - val_loss: 1.7610 - val_accuracy: 0.7399


Testing...
10000/10000 [==============================] - 1s 120us/step

Test score: 1.8258602352142335
Test accuracy: 0.7268999814987183
```

PLOT





VGG-16 Model summary:

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
zero_padding2d_53 (ZeroPaddi (None, 30, 30, 1)         0
_________________________________________________________________
conv2d_53 (Conv2D)           (None, 28, 28, 64)        640
_________________________________________________________________
zero_padding2d_54 (ZeroPaddi (None, 30, 30, 64)        0
_________________________________________________________________
conv2d_54 (Conv2D)           (None, 28, 28, 64)        36928
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 14, 14, 64)        0
_________________________________________________________________
zero_padding2d_55 (ZeroPaddi (None, 16, 16, 64)        0
_________________________________________________________________
conv2d_55 (Conv2D)           (None, 14, 14, 128)       73856
_________________________________________________________________
zero_padding2d_56 (ZeroPaddi (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_56 (Conv2D)           (None, 14, 14, 128)       147584
_________________________________________________________________
max_pooling2d_22 (MaxPooling (None, 7, 7, 128)         0
_________________________________________________________________
zero_padding2d_57 (ZeroPaddi (None, 9, 9, 128)         0
_________________________________________________________________
conv2d_57 (Conv2D)           (None, 7, 7, 256)         295168
_________________________________________________________________
zero_padding2d_58 (ZeroPaddi (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_58 (Conv2D)           (None, 7, 7, 256)         590080
_________________________________________________________________
zero_padding2d_59 (ZeroPaddi (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_59 (Conv2D)           (None, 7, 7, 256)         590080
_________________________________________________________________
max_pooling2d_23 (MaxPooling (None, 3, 3, 256)         0
_________________________________________________________________
zero_padding2d_60 (ZeroPaddi (None, 5, 5, 256)         0
_________________________________________________________________
conv2d_60 (Conv2D)           (None, 3, 3, 512)         1180160
_________________________________________________________________
zero_padding2d_61 (ZeroPaddi (None, 5, 5, 512)         0
_________________________________________________________________
conv2d_61 (Conv2D)           (None, 3, 3, 512)         2359808
_________________________________________________________________
zero_padding2d_62 (ZeroPaddi (None, 5, 5, 512)         0
_________________________________________________________________
conv2d_62 (Conv2D)           (None, 3, 3, 512)         2359808
_________________________________________________________________
max_pooling2d_24 (MaxPooling (None, 1, 1, 512)         0
_________________________________________________________________
zero_padding2d_63 (ZeroPaddi (None, 3, 3, 512)         0
_________________________________________________________________
```

```
conv2d_63 (Conv2D)           (None, 1, 1, 512)         2359808
_________________________________________________________________
zero_padding2d_64 (ZeroPaddi (None, 3, 3, 512)         0
_________________________________________________________________
conv2d_64 (Conv2D)           (None, 1, 1, 512)         2359808
_________________________________________________________________
zero_padding2d_65 (ZeroPaddi (None, 3, 3, 512)         0
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 1, 1, 512)         2359808
_________________________________________________________________
max_pooling2d_25 (MaxPooling (None, 1, 1, 512)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 512)               0
_________________________________________________________________
dense_13 (Dense)             (None, 4096)              2101248
_________________________________________________________________
dropout_9 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_14 (Dense)             (None, 4096)              16781312
_________________________________________________________________
dropout_10 (Dropout)         (None, 4096)              0
_________________________________________________________________
dense_15 (Dense)             (None, 10)                40970
=================================================================
Total params: 33,637,066
Trainable params: 33,637,066
Non-trainable params: 0
_________________________________________________________________
```

<u>ResNet50 Model Summary:</u>

```
Model: "res_net50"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              multiple                  3200
_________________________________________________________________
batch_normalization (BatchNo multiple                  256
_________________________________________________________________
activation (Activation)      multiple                  0
_________________________________________________________________
max_pooling2d (MaxPooling2D) multiple                  0
_________________________________________________________________
residual_block (ResidualBloc multiple                  75904
_________________________________________________________________
residual_block_1 (ResidualBl multiple                  71552
_________________________________________________________________
residual_block_2 (ResidualBl multiple                  71552
_________________________________________________________________
conv2d_11 (Conv2D)           multiple                  131584
_________________________________________________________________
residual_block_3 (ResidualBl multiple                  282368
_________________________________________________________________
residual_block_4 (ResidualBl multiple                  282368
```

```
_________________________________________________________________
residual_block_5 (ResidualBl multiple                  282368
_________________________________________________________________
residual_block_6 (ResidualBl multiple                  282368
_________________________________________________________________
conv2d_24 (Conv2D)           multiple                  525312
_________________________________________________________________
residual_block_7 (ResidualBl multiple                  1121792
_________________________________________________________________
residual_block_8 (ResidualBl multiple                  1121792
_________________________________________________________________
residual_block_9 (ResidualBl multiple                  1121792
_________________________________________________________________
residual_block_10 (ResidualB multiple                  1121792
_________________________________________________________________
residual_block_11 (ResidualB multiple                  1121792
_________________________________________________________________
residual_block_12 (ResidualB multiple                  1121792
_________________________________________________________________
conv2d_43 (Conv2D)           multiple                  2099200
_________________________________________________________________
residual_block_13 (ResidualB multiple                  4471808
_________________________________________________________________
residual_block_14 (ResidualB multiple                  4471808
_________________________________________________________________
residual_block_15 (ResidualB multiple                  4471808
_________________________________________________________________
global_average_pooling2d (Gl multiple                  0
_________________________________________________________________
dense (Dense)                multiple                  2049000
_________________________________________________________________
dense_1 (Dense)              multiple                  10010
=================================================================
Total params: 26,313,218
Trainable params: 26,267,778
Non-trainable params: 45,440
```