Transfer learning and the art of using Pre-trained Models in Deep Learning

DISHASHREE GUPTA, JUNE 1, 2017 LOGIN TO BOOKMARK THIS ARTICLE

Introduction

Neural networks are a different breed of models compared to the supervised machine learning algorithms. Why do I say so? There are multiple reasons for that, but the most prominent is the cost of running algorithms on the hardware.

In today's world, RAM on a machine is cheap and is available in plenty. You need hundreds of GBs of RAM to run a super complex supervised machine learning problem – it can be yours for a little investment / rent. On the other hand, access to GPUs is not that cheap. You need access to hundred GB VRAM on GPUs – it won't be straight forward and would involve significant costs.

Now, that may change in future. But for now, it means that we have to be smarter about the way we use our resources in solving Deep Learning problems. Especially so, when we try to solve complex real life problems on areas like image and voice recognition. Once you have a few hidden layers in your model, adding another layer of hidden layer would need immense resources.

Thankfully, there is something called "Transfer Learning" which enables us to use pre-trained models from other people by making small changes. In this article, I am going to tell how we can use pre-trained models to accelerate our solutions.

To learn more about pre-trained models and transfer learning and their specific use cases, you can check out the following articles:

- Deep Learning for Everyone: Master the Powerful Art of Transfer Learning using PyTorch
- Top 10 Pretrained Models to get you Started with Deep Learning (Part 1 – Computer Vision)
- 8 Excellent Pretrained Models to get you Started with Natural Language Processing (NLP)

Note – *This article assumes basic familiarity with Neural networks and deep learning. If you are new to deep learning, I would strongly recommend that you read the following articles first:*

1. What is deep learning and why is it getting so much attention?
2. Deep Learning vs. Machine Learning – the essential differences you need to know!
3. 25 Must Know Terms & concepts for Beginners in Deep Learning
4. Why are GPUs necessary for training Deep Learning models?

Table of Contents

What is transfer learning?

Let us start with developing an intuition for transfer learning. Let us understand from a simple teacher – student analogy.

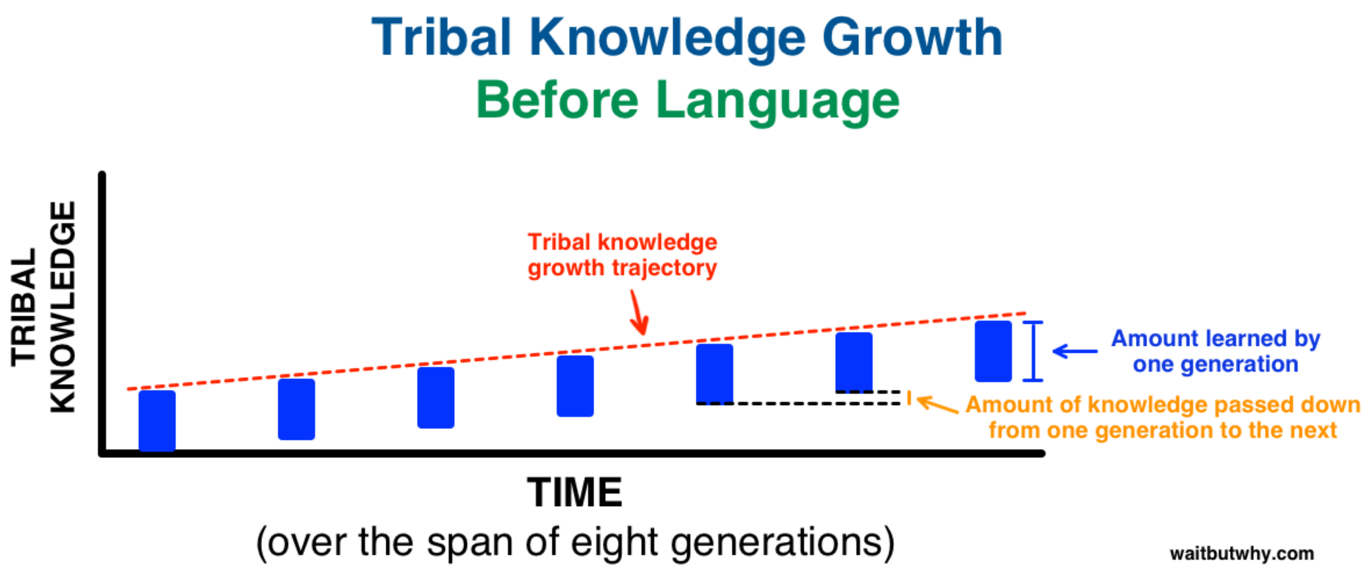
A teacher has years of experience in the particular topic he/she teaches. With all this accumulated information, the lectures that students get is a concise and brief overview of the topic. So it can be seen as a "transfer" of information from the learned to a novice.

TRANSFER OF LEARNING
The application of skills, knowledge, and/or attitudes that were learned in one situation to another learning situation (Perkins, 1992)

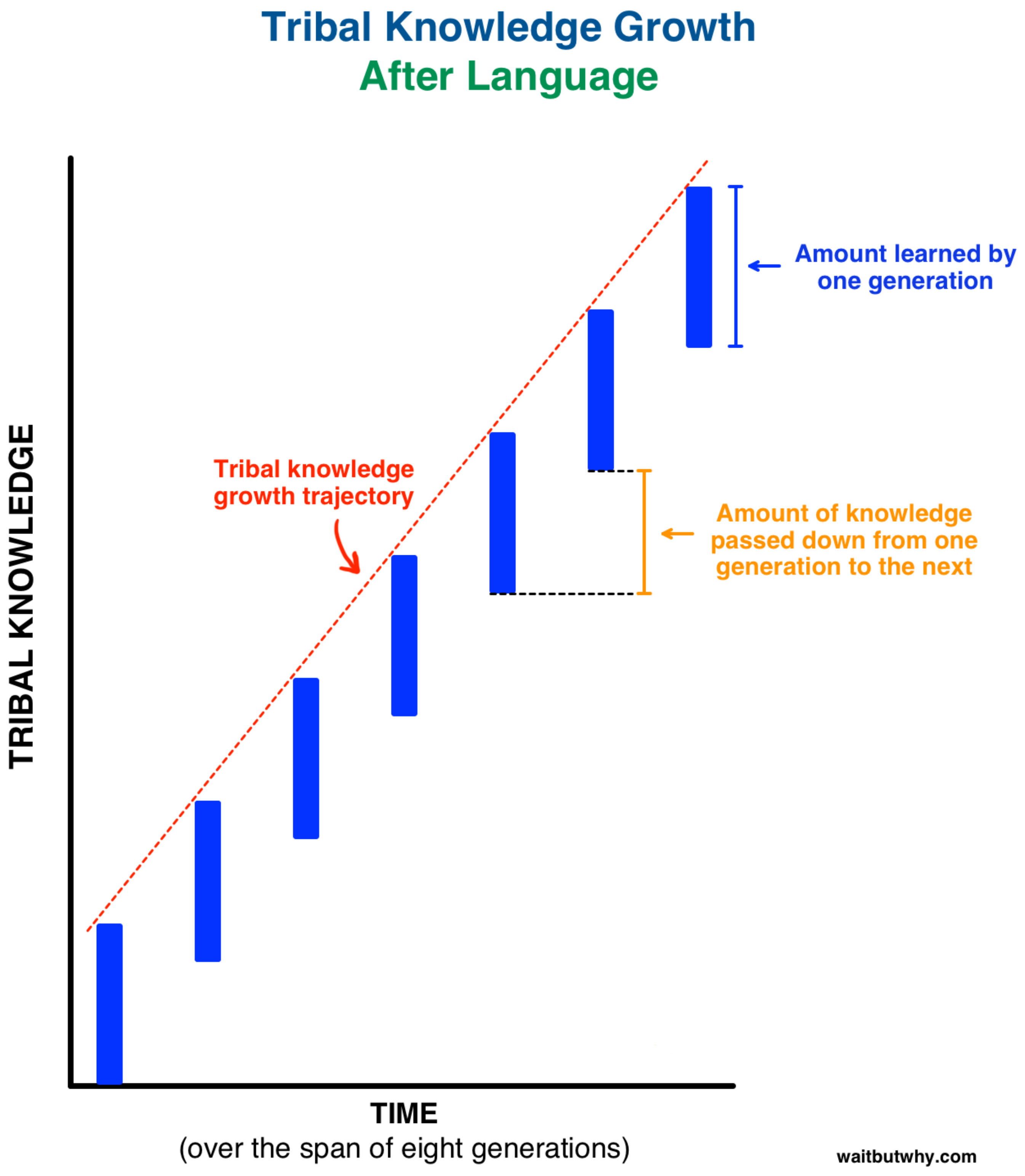
Keeping in mind this analogy, we compare this to neural network. A neural network is trained on a data. This network gains knowledge from this data, which is compiled as "weights" of the network. These weights can be extracted and then transferred to any other neural network. Instead of training the other neural network from scratch, we "**transfer**" the learned features.

Now, let us reflect on the importance of transfer learning by relating to our evolution. And what better way than to use transfer learning for this! So I am picking on a concept touched on by Tim Urban from one of his recent articles on waitbutwhy.com

Tim explains that before language was invented, every generation of humans had to re-invent the knowledge for themselves and this is how knowledge growth was happening from one generation to other:


Tribal Knowledge Growth
Before Language
TRIBAL KNOWLEDGE
Tribal knowledge growth trajectory
Amount learned by one generation
Amount of knowledge passed down from one generation to the next
TIME
(over the span of eight generations)
waitbutwhy.com

Then, we invented language! A way to transfer learning from one generation to another and this is what happened over same time frame:



Isn't it phenomenal and super empowering? So, transfer learning by passing on weights is equivalent of language used to disseminate knowledge over generations in human evolution.

What is a Pre-trained Model?

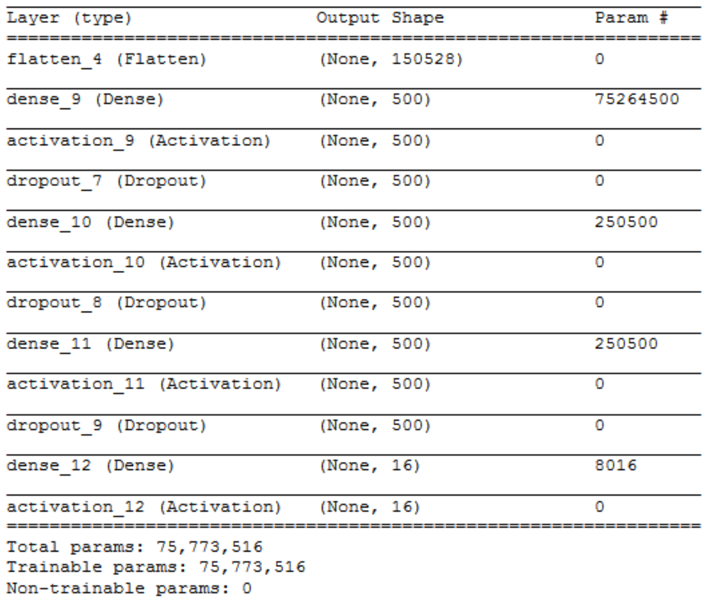
Simply put, a pre-trained model is a model created by some one else to solve a similar problem. Instead of building a model from scratch to solve a similar problem, you use the model trained on other problem as a starting point.

For example, if you want to build a self learning car. You can spend years to build a decent image recognition algorithm from scratch or you can take inception model (a pre-trained model) from Google which was built on ImageNet data to identify images in those pictures.

A pre-trained model may not be 100% accurate in your application, but it saves huge efforts required to re-invent the wheel. Let me show this to you with a recent example.

Why would we use Pre-trained Models?

I spent my last week working on a problem at CrowdAnalytix platform – Identifying themes from mobile case images. This was an image classification problem where we were given 4591 images in the training dataset and 1200 images in the test dataset. The objective was to classify the images into one of the 16 categories. After the basic pre-processing steps, I started off with a simple MLP model with the following architecture-



```
Layer (type)                 Output Shape              Param #
=================================================================
flatten_4 (Flatten)          (None, 150528)            0
_________________________________________________________________
dense_9 (Dense)              (None, 500)               75264500
_________________________________________________________________
activation_9 (Activation)    (None, 500)               0
_________________________________________________________________
dropout_7 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_10 (Dense)             (None, 500)               250500
_________________________________________________________________
activation_10 (Activation)   (None, 500)               0
_________________________________________________________________
dropout_8 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_11 (Dense)             (None, 500)               250500
_________________________________________________________________
activation_11 (Activation)   (None, 500)               0
_________________________________________________________________
dropout_9 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_12 (Dense)             (None, 16)                8016
_________________________________________________________________
activation_12 (Activation)   (None, 16)                0
=================================================================
Total params: 75,773,516
Trainable params: 75,773,516
Non-trainable params: 0
```

To simplify the above architecture after flattening the input image [224 X 224 X 3] into [150528], I used three hidden layers with 500, 500 and 500 neurons respectively. The output layer had 16 neurons which correspond to the number of categories in which we need to classify the input image.

I barely managed a training accuracy of 6.8 % which turned out to be very bad. Even experimenting with hidden layers, number of neurons in hidden layers and drop out rates. I could not manage to substantially increase my training accuracy. Increasing the hidden layers and the number of neurons, caused 20 seconds to run a single epoch on my Titan X GPU with 12 GB VRAM.

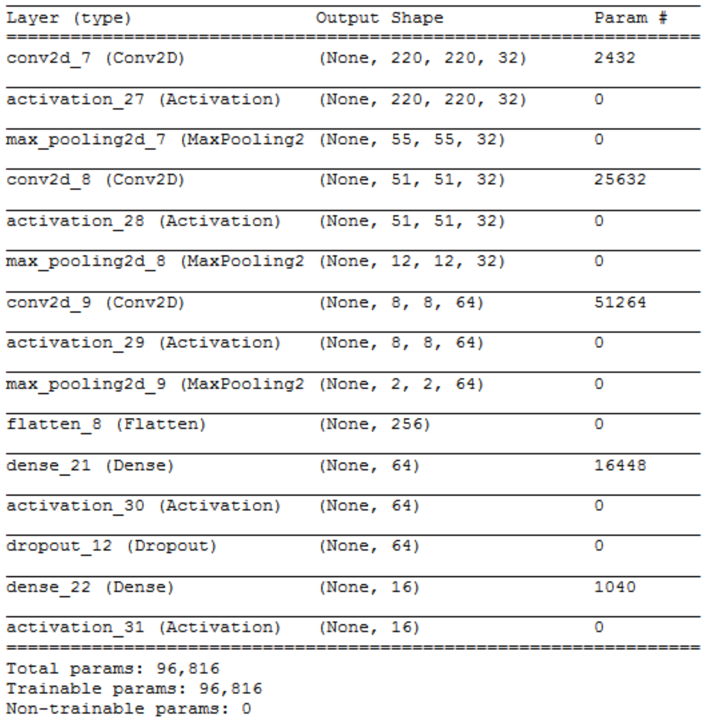
Below is an output of the training using the MLP model with the above architecture.

Epoch 10/10

50/50 [==============================] – 21s – loss: 15.0100 – acc: 0.0688

As, you can see MLP was not going to give me any better results without exponentially increasing my training time. So I switched to Convolutional Neural Network to see how they perform on this dataset and whether I would be able to increase my training accuracy.

The CNN had the below architecture –



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 220, 220, 32)      2432
_________________________________________________________________
activation_27 (Activation)   (None, 220, 220, 32)      0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 51, 51, 32)        25632
_________________________________________________________________
activation_28 (Activation)   (None, 51, 51, 32)        0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 8, 8, 64)          51264
_________________________________________________________________
activation_29 (Activation)   (None, 8, 8, 64)          0
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
flatten_8 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_21 (Dense)             (None, 64)                16448
_________________________________________________________________
activation_30 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_12 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_22 (Dense)             (None, 16)                1040
_________________________________________________________________
activation_31 (Activation)   (None, 16)                0
=================================================================
Total params: 96,816
Trainable params: 96,816
Non-trainable params: 0
```

I used 3 convolutional blocks with each block following the below architecture-

1. 32 filters of size 5 X 5

2. Activation function – relu
3. Max pooling layer of size 4 X 4

The result obtained after the final convolutional block was flattened into a size [256] and passed into a single hidden layer of with 64 neurons. The output of the hidden layer was passed onto the output layer after a drop out rate of 0.5.

The result obtained with the above architecture is summarized below-

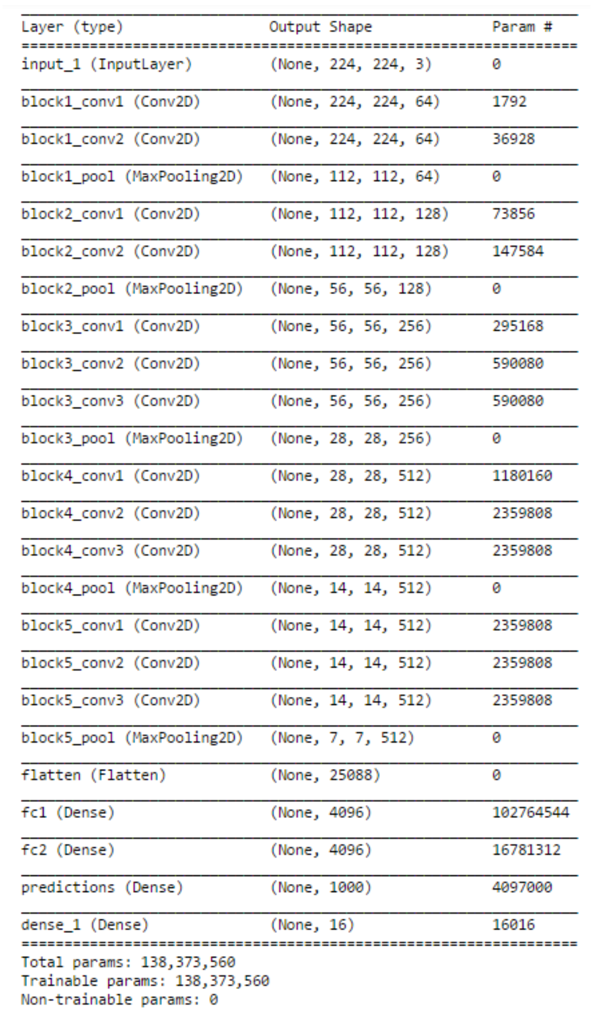
Epoch 10/10

50/50 [==============================] – 21s – loss: 13.5733 – acc: 0.1575

Though my accuracy increased in comparison to the MLP output, it also increased the time taken to run a single epoch – 21 seconds.

But the major point to note was that the majority class in the dataset was around 17.6%. So, even if we had predicted the class of every image in the train dataset to be the majority class, we would have performed better than MLP and CNN respectively. Addition of more convolutional blocks substantially increased my training time. This led me to switch onto using pre-trained models where I would not have to train my entire architecture but only a few layers.

So, I used VGG16 model which is pre-trained on the ImageNet dataset and provided in the keras library for use. Below is the architecture of the VGG16 model which I used.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
_________________________________________________________________
dense_1 (Dense)              (None, 16)                16016
=================================================================
Total params: 138,373,560
Trainable params: 138,373,560
Non-trainable params: 0
```

The only change that I made to the VGG16 existing architecture is changing the softmax layer with 1000 outputs to 16 categories suitable for our problem and re-training the dense layer.

This architecture gave me an accuracy of 70% much better than MLP and CNN. Also, the biggest benefit of using the VGG16 pre-trained model was almost negligible time to train the dense layer with greater accuracy.

So, I moved forward with this approach of using a pre-trained model and the next step was to fine tune my VGG16 model to suit this problem.

How can I use Pre-trained Models?

What is our objective when we train a neural network? We wish to identify the correct weights for the network by multiple forward and backward iterations. By using pre-trained models which have been previously trained on large datasets, we can directly use the weights and architecture obtained and apply the learning on our problem statement. This is known as transfer learning. We "transfer the learning" of the pre-trained model to our specific problem statement.

You should be very careful while choosing what pre-trained model you should use in your case. If the problem statement we have at hand is very different from the one on which the pre-trained model was trained – the prediction we would get would be very inaccurate. For example, a model previously trained for speech recognition would work horribly if we try to use it to identify objects using it.

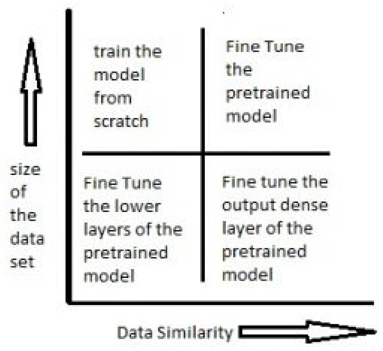
We are lucky that many pre-trained architectures are directly available for us in the Keras library. **Imagenet** data set has been widely used to build various architectures since it is large enough (1.2M images) to create a generalized model. The problem statement is to train a model that can correctly classify the images into 1,000 separate object categories. These 1,000 image categories represent object classes that we come across in our day-to-day lives, such as species of dogs, cats, various household objects, vehicle types etc.

These pre-trained networks demonstrate a strong ability to generalize to images outside the ImageNet dataset via transfer learning. We make modifications in the pre-existing model by fine-tuning the model. Since we assume that the pre-trained network has been trained quite well, we would not want to modify the weights too soon and too much. While modifying we generally use a learning rate smaller than the one used for initially training the model.

Ways to Fine tune the model

1. **Feature extraction** – We can use a pre-trained model as a feature extraction mechanism. What we can do is that we can remove the output layer(the one which gives the probabilities for being in each of the 1000 classes) and then use the entire network as a fixed feature extractor for the new data set.
2. **Use the Architecture of the pre-trained model –** What we can do is that we use architecture of the model while we initialize all the weights randomly and train the model according to our dataset again.
3. **Train some layers while freeze others** – Another way to use a pre-trained model is to train is partially. What we can do is we keep the weights of initial layers of the model frozen while we retrain only the higher layers. We can try and test as to how many layers to be frozen and how many to be trained.

The below diagram should help you decide on how to proceed on using the pre trained model in your case –


train the model from scratch
Fine Tune the pretrained model
Fine Tune the lower layers of the pretrained model
Fine tune the output dense layer of the pretrained model
size of the data set
Data Similarity

Scenario 1 – Size of the Data set is small while the Data similarity is very high – In this case, since the data similarity is very high, we do not need to retrain the model. All we need to do is to customize and modify the output layers according to our problem statement. We use the pretrained model as a feature extractor. Suppose we decide to use models trained on Imagenet to identify if the new set of images have cats or dogs. Here the images we need to identify would be similar to imagenet, however we just need two categories as my output – cats or dogs. In this case all we do is just modify the dense layers and the final softmax layer to output 2 categories instead of a 1000.

Scenario 2 – Size of the data is small as well as data similarity is very low – In this case we can freeze the initial (let's say k) layers of the pretrained model and train just the remaining(n-k) layers again. The top layers would then be customized to the new data set. Since the new data set has low similarity it is significant to retrain and customize the higher layers according to the new dataset. The small size of the data set is compensated by the fact that the initial layers are kept pretrained(which have been trained on a large dataset previously) and the weights for those layers are frozen.

Scenario 3 – Size of the data set is large however the Data similarity is very low – In this case, since we have a large dataset, our neural network training would be effective. However, since the data we have is very different as compared to the data used for training our pretrained models. The predictions made using pretrained models would not be effective. Hence, its best to train the neural network from scratch according to your data.

Scenario 4 – Size of the data is large as well as there is high data similarity – This is the ideal situation. In this case the pretrained model should be most effective. The best way to use the model is to retain the architecture of the model and the initial weights of the model. Then we can retrain this model using the weights as initialized in the pre-trained model.

Use the pre-trained models to identify handwritten digits

Let's now try to use a pretrained model for a simple problem. There are various architectures that have been trained on the imageNet data set. You can go through various architectures here. I have used vgg16 as pretrained model architecture and have tried to identify handwritten digits using it. Let's see in which of the above scenarios would this problem fall into. We have around 60,000 training images of handwritten digits. This data set is definitely small. So the situation would either fall into scenario 1 or scenario 2. We shall try to solve the problem using both these scenarios. The data set can be downloaded from here.

1. **Retrain the output dense layers only** – Here we use vgg16 as a feature extractor. We then use these features and send them to dense layers which are trained according to our data set. The output layer is also replaced with our new softmax layer relevant to our problem. The output layer in a vgg16 is a softmax activation with 1000 categories. We remove this layer and replace it with a softmax layer of 10 categories. We just train the weights of these layers and try to identify the digits.

```
# importing required libraries

from keras.models import Sequential
from scipy.misc import imread
get_ipython().magic('matplotlib inline')
```

```
import matplotlib.pyplot as plt
import numpy as np
import keras
from keras.layers import Dense
import pandas as pd

from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np
from keras.applications.vgg16 import decode_predictions
train=pd.read_csv("R/Data/Train/train.csv")
test=pd.read_csv("R/Data/test.csv")
train_path="R/Data/Train/Images/train/"
test_path="R/Data/Train/Images/test/"

from scipy.misc import imresize
# preparing the train dataset

train_img=[]
for i in range(len(train)):

   temp_img=image.load_img(train_path+train['filename'][i],target_size=(224,224))

   temp_img=image.img_to_array(temp_img)

   train_img.append(temp_img)

#converting train images to array and applying mean subtraction processing

 train_img=np.array(train_img)
train_img=preprocess_input(train_img)
# applying the same procedure with the test dataset

test_img=[]
for i in range(len(test)):

   temp_img=image.load_img(test_path+test['filename'][i],target_size=(224,224))

   temp_img=image.img_to_array(temp_img)

   test_img.append(temp_img)

test_img=np.array(test_img)
test_img=preprocess_input(test_img)

# loading VGG16 model weights
model = VGG16(weights='imagenet', include_top=False)
# Extracting features from the train dataset using the VGG16 pre-trained model

features_train=model.predict(train_img)
# Extracting features from the train dataset using the VGG16 pre-trained model

features_test=model.predict(test_img)

# flattening the layers to conform to MLP input

train_x=features_train.reshape(49000,25088)
# converting target variable to array

train_y=np.asarray(train['label'])
# performing one-hot encoding for the target variable

train_y=pd.get_dummies(train_y)
train_y=np.array(train_y)
# creating training and validation set

from sklearn.model_selection import train_test_split
X_train, X_valid, Y_train, Y_valid=train_test_split(train_x,train_y,test_size=0.3, random_state=42)



# creating a mlp model
from keras.layers import Dense, Activation
model=Sequential()

model.add(Dense(1000, input_dim=25088, activation='relu',kernel_initializer='uniform'))
keras.layers.core.Dropout(0.3, noise_shape=None, seed=None)

model.add(Dense(500,input_dim=1000,activation='sigmoid'))
keras.layers.core.Dropout(0.4, noise_shape=None, seed=None)

model.add(Dense(150,input_dim=500,activation='sigmoid'))
keras.layers.core.Dropout(0.2, noise_shape=None, seed=None)

model.add(Dense(units=10))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer="adam", metrics=['accuracy'])

# fitting the model
```