====Method 3: ResNet ====

ResNet

Applying residual blocks is generally useful for neural networks where there are many layers and as a result vanishing gradient problem.

We have applied AlexNet which is really only composed of 8 layers out of which there are only 4 convolutional layers.

The benefits of ResNet are better reaped in networks with much more layers. It is also simpler to apply residual blocks in those networks where there are at least two or more convolutional layers with the same number of filters as then the output after one layer can be added with the output after two or more such layers and there would not be dimension mismatch.

Below, we have applied ResNet to a plain neural network that has 34 layers.

ResNet on 34-layer architecture

In [29]:
```
from keras.optimizers import Adam
from keras.layers import Dense, Activation, Conv2D, MaxPooling2D, Flatten, Dropout
from keras.layers import BatchNormalization, AveragePooling2D, concatenate
from keras.layers import Input, concatenate
from keras.layers import Add
from keras.layers import GlobalAveragePooling2D
from keras.models import Model
```

In [30]:
```
#Function for convolution with BatchNormalization
def Conv2d_BN(x, nb_filter, kernel_size, padding='same', strides=(1,1), name=None):
  if name is not None:
    bn_name = name + '_bn'
    conv_name = name + '_conv'
  else:
    bn_name = None
    conv_name = None

  x = Conv2D(nb_filter,kernel_size, padding=padding,strides=strides,activation='relu',name=conv_name)(x
)
  x = BatchNormalization(axis=3, name=bn_name)(x)
  #axis =3 meaning to apply normalization over channels
  return x
```

In [31]:
```
#Function for residual block
def Residual_Block(input_model,nb_filter,kernel_size, strides=(1,1), with_conv_shortcut=False):
  x = Conv2d_BN(input_model,nb_filter=nb_filter, kernel_size=kernel_size, strides=strides, padding='sam
e')

  if with_conv_shortcut:
    shortcut = Conv2d_BN(input_model, nb_filter=nb_filter, strides=strides, kernel_size = kernel_size)
    x = Add()([x, shortcut])
    return x
  else:
    x = Add()([x, input_model])
    return x
```

We are going to apply a hop over 2 convolutional layers.

In [32]:
```
def ResNet34(width,height,depth):
  Img = Input(shape=(width, height, depth))

  x = Conv2d_BN(Img,nb_filter=64,kernel_size=(7,7), strides=(2,2), padding='same')
  x = MaxPooling2D(pool_size=(3,3),strides=(2,2), padding='same')(x)

  x = Residual_Block(x,nb_filter=64,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=64,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=64,kernel_size=(3,3))

  x = Residual_Block(x,nb_filter=128,kernel_size=(3,3),strides=(2,2),with_conv_shortcut=True)
  x = Residual_Block(x,nb_filter=128,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=128,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=128,kernel_size=(3,3))

  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3),strides=(2,2),with_conv_shortcut=True)
  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=256,kernel_size=(3,3))

  x = Residual_Block(x,nb_filter=512,kernel_size=(3,3),strides=(2,2),with_conv_shortcut=(True))
  x = Residual_Block(x,nb_filter=512,kernel_size=(3,3))
  x = Residual_Block(x,nb_filter=512,kernel_size=(3,3))

  x = GlobalAveragePooling2D()(x)
  x = Dense(1, activation='sigmoid')(x)

  model = Model(Img,x)

  return model
```

In [33]:
```
resnet_model = ResNet34(150,150,3)
```

In [34]:
```
resnet_model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_10 (InputLayer)           [(None, 150, 150, 3) 0
conv2d_78 (Conv2D)              (None, 75, 75, 64)   9472        input_10[0][0]
batch_normalization_78 (BatchNo (None, 75, 75, 64)   256         conv2d_78[0][0]
max_pooling2d_10 (MaxPooling2D) (None, 38, 38, 64)   0           batch_normalization_78[0][0]
conv2d_79 (Conv2D)              (None, 38, 38, 64)   36928       max_pooling2d_10[0][0]
batch_normalization_79 (BatchNo (None, 38, 38, 64)   256         conv2d_79[0][0]
add_59 (Add)                    (None, 38, 38, 64)   0           batch_normalization_79[0][0]
                                                                 max_pooling2d_10[0][0]
conv2d_80 (Conv2D)              (None, 38, 38, 64)   36928       add_59[0][0]
batch_normalization_80 (BatchNo (None, 38, 38, 64)   256         conv2d_80[0][0]
add_60 (Add)                    (None, 38, 38, 64)   0           batch_normalization_80[0][0]
                                                                 add_59[0][0]
conv2d_81 (Conv2D)              (None, 38, 38, 64)   36928       add_60[0][0]
batch_normalization_81 (BatchNo (None, 38, 38, 64)   256         conv2d_81[0][0]
add_61 (Add)                    (None, 38, 38, 64)   0           batch_normalization_81[0][0]
                                                                 add_60[0][0]
conv2d_82 (Conv2D)              (None, 19, 19, 128)  73856       add_61[0][0]
conv2d_83 (Conv2D)              (None, 19, 19, 128)  73856       add_61[0][0]
batch_normalization_82 (BatchNo (None, 19, 19, 128)  512         conv2d_82[0][0]
batch_normalization_83 (BatchNo (None, 19, 19, 128)  512         conv2d_83[0][0]
add_62 (Add)                    (None, 19, 19, 128)  0           batch_normalization_82[0][0]
                                                                 batch_normalization_83[0][0]
conv2d_84 (Conv2D)              (None, 19, 19, 128)  147584      add_62[0][0]
batch_normalization_84 (BatchNo (None, 19, 19, 128)  512         conv2d_84[0][0]
add_63 (Add)                    (None, 19, 19, 128)  0           batch_normalization_84[0][0]
                                                                 add_62[0][0]
conv2d_85 (Conv2D)              (None, 19, 19, 128)  147584      add_63[0][0]
batch_normalization_85 (BatchNo (None, 19, 19, 128)  512         conv2d_85[0][0]
add_64 (Add)                    (None, 19, 19, 128)  0           batch_normalization_85[0][0]
                                                                 add_63[0][0]
conv2d_86 (Conv2D)              (None, 19, 19, 128)  147584      add_64[0][0]
batch_normalization_86 (BatchNo (None, 19, 19, 128)  512         conv2d_86[0][0]
add_65 (Add)                    (None, 19, 19, 128)  0           batch_normalization_86[0][0]
                                                                 add_64[0][0]
conv2d_87 (Conv2D)              (None, 10, 10, 256)  295168      add_65[0][0]
conv2d_88 (Conv2D)              (None, 10, 10, 256)  295168      add_65[0][0]
batch_normalization_87 (BatchNo (None, 10, 10, 256)  1024        conv2d_87[0][0]
batch_normalization_88 (BatchNo (None, 10, 10, 256)  1024        conv2d_88[0][0]
add_66 (Add)                    (None, 10, 10, 256)  0           batch_normalization_87[0][0]
                                                                 batch_normalization_88[0][0]
conv2d_89 (Conv2D)              (None, 10, 10, 256)  590080      add_66[0][0]
batch_normalization_89 (BatchNo (None, 10, 10, 256)  1024        conv2d_89[0][0]
add_67 (Add)                    (None, 10, 10, 256)  0           batch_normalization_89[0][0]
                                                                 add_66[0][0]
conv2d_90 (Conv2D)              (None, 10, 10, 256)  590080      add_67[0][0]
batch_normalization_90 (BatchNo (None, 10, 10, 256)  1024        conv2d_90[0][0]
add_68 (Add)                    (None, 10, 10, 256)  0           batch_normalization_90[0][0]
                                                                 add_67[0][0]
conv2d_91 (Conv2D)              (None, 10, 10, 256)  590080      add_68[0][0]
batch_normalization_91 (BatchNo (None, 10, 10, 256)  1024        conv2d_91[0][0]
add_69 (Add)                    (None, 10, 10, 256)  0           batch_normalization_91[0][0]
                                                                 add_68[0][0]
conv2d_92 (Conv2D)              (None, 10, 10, 256)  590080      add_69[0][0]
batch_normalization_92 (BatchNo (None, 10, 10, 256)  1024        conv2d_92[0][0]
add_70 (Add)                    (None, 10, 10, 256)  0           batch_normalization_92[0][0]
                                                                 add_69[0][0]
conv2d_93 (Conv2D)              (None, 10, 10, 256)  590080      add_70[0][0]
batch_normalization_93 (BatchNo (None, 10, 10, 256)  1024        conv2d_93[0][0]
add_71 (Add)                    (None, 10, 10, 256)  0           batch_normalization_93[0][0]
                                                                 add_70[0][0]
conv2d_94 (Conv2D)              (None, 5, 5, 512)    1180160     add_71[0][0]
conv2d_95 (Conv2D)              (None, 5, 5, 512)    1180160     add_71[0][0]
batch_normalization_94 (BatchNo (None, 5, 5, 512)    2048        conv2d_94[0][0]
batch_normalization_95 (BatchNo (None, 5, 5, 512)    2048        conv2d_95[0][0]
add_72 (Add)                    (None, 5, 5, 512)    0           batch_normalization_94[0][0]
                                                                 batch_normalization_95[0][0]
conv2d_96 (Conv2D)              (None, 5, 5, 512)    2359808     add_72[0][0]
batch_normalization_96 (BatchNo (None, 5, 5, 512)    2048        conv2d_96[0][0]
add_73 (Add)                    (None, 5, 5, 512)    0           batch_normalization_96[0][0]
                                                                 add_72[0][0]
conv2d_97 (Conv2D)              (None, 5, 5, 512)    2359808     add_73[0][0]
batch_normalization_97 (BatchNo (None, 5, 5, 512)    2048        conv2d_97[0][0]
add_74 (Add)                    (None, 5, 5, 512)    0           batch_normalization_97[0][0]
                                                                 add_73[0][0]
global_average_pooling2d_3 (Glo (None, 512)          0           add_74[0][0]
dense_3 (Dense)                 (None, 1)            513         global_average_pooling2d_3[0][0]
==================================================================================================
Total params: 11,350,849
Trainable params: 11,341,377
Non-trainable params: 9,472
__________________________________________________________________________________________________
```

Now, we can load in the data like for other models and go ahead an train the model on our training data.

In [35]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

In [36]:
```
import os, shutil

TRAINDIR = 'gdrive/MyDrive/ee_628/proj/train/'
cat_folder = 'cat'
dog_folder = 'dog'
```

In [37]:
```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rotation_range=10,
                             shear_range=0.2,
                             rescale=1./255,
                             validation_split=0.25)
IMG_H = 150
IMG_W = 150
```

In [38]:
```
train_generator = datagen.flow_from_directory(TRAINDIR,
                                              target_size=(IMG_H,IMG_W),
                                              batch_size=100,
                                              class_mode='binary',
                                              subset='training')
```

```
Found 18750 images belonging to 2 classes.
```

In [40]:
```
val_generator = datagen.flow_from_directory(TRAINDIR,
                                            target_size=(IMG_H, IMG_W),
                                            batch_size=100,
                                            class_mode='binary',
                                            subset='validation')
```

```
Found 6250 images belonging to 2 classes.
```

In [42]:
```
resnet_model.compile(loss='binary_crossentropy', optimizer='adam', metrics = ['accuracy'])
```

In [43]:
```
hist_resnet = resnet_model.fit_generator(train_generator, validation_data=val_generator, epochs=1)
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`,
which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
```

```
188/188 [==============================] - 8063s 43s/step - loss: 1.1461 - accuracy: 0.5742 - val_los
s: 0.8520 - val_accuracy: 0.5517
```

In [45]:
```
resnet_model.metrics_names
```

Out[45]:
```
['loss', 'accuracy']
```

In [44]:
```
resnet_model.evaluate_generator(train_generator)
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1877: UserWarning:
`Model.evaluate_generator` is deprecated and will be removed in a future version. Please use `Model.e
valuate`, which supports generators.
  warnings.warn('`Model.evaluate_generator` is deprecated and '
```

Out[44]:
```
[0.8639941215515137, 0.5518933534622192]
```

In [46]:
```
resnet_model.evaluate_generator(val_generator)
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1877: UserWarning:
`Model.evaluate_generator` is deprecated and will be removed in a future version. Please use `Model.e
valuate`, which supports generators.
  warnings.warn('`Model.evaluate_generator` is deprecated and '
```

Out[46]:
```
[0.8517088890075684, 0.5542399883270264]
```

In [ ]:
```
hist2_resnet = resnet_model.fit_generator(train_generator, validation_data=val_generator, epochs=20)
```

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`,
which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
```

```
Epoch 1/20
188/188 [==============================] - 2558s 14s/step - loss: 0.5759 - accuracy: 0.7007 - val_los
s: 0.9727 - val_accuracy: 0.5896
Epoch 2/20
 53/188 [=======>......................] - ETA: 28:27 - loss: 0.5229 - accuracy: 0.7331
```

Conclusions:

	Train Performance	Validation Performance
AlexNet	71.4%	68.5%
Transfer Learning with VGG16	50.0%	50.0%
ResNet34	55.2%	55.4%

Due to time constraints, not many epochs could be run for each of the networks tried and tested here. For the first method, AlexNet, we observed that there was about 71% accuracy on training set and 68% on validation set. This shows that we are still underfitting and that there could be more epochs run to reach a higher accuracy on the validation set before trying out the network on an unseen test set. In the second method, we observed some unexpected results as the accuracy of validation set seemed to stay constant at 50% which means that there clearly may be some error and the network perhaps needs more scrutiny. For transfer learning, we imported the VGG16 network's beginning convolutional layers with the weights after it has been trained on ImageNet. ImageNet is a dataset that consists of color (3 channels) images of various classes (~ 200) including animals. Seeings as these are color images which include animals, it seemed like a good fit to have a model trained on this dataset for transfer learning and applying it to our data of cats and dogs. In addition to importing the top of the trained VGG16 neural network architecture, 4 dense layers were attached at the end and then classification using sigmoid was done. Having this many fully connected layers could have in turn caused the poor performance for this method. This was expected to be the best performer and therefore it was a bit surprising to see its lower performance. For the third method, initially it was desired to apply residual blocks to known CNNs such as AlexNet. However, when designing the ResNet, it was quickly realized that a shortcut or skip would have to be done over at least two convolutional layers and both these layers would have to have equal number of filters so that the output could be added with the input and there was no dimension mismatch. (Note that we could still perform addition on input and output for the cases where the number of filters are different, by considering different padding, but this would be a complicated process and something worth considering for a future separate project) Therefore, instead the ResNet was applied to a 34-layer network. In this the 'skips' or shortcuts were applied after 2 identical convolutional layers.

For each of the above cases, one of the biggest constraints has been of time. The number of epochs has been limited to 20, which in some cases may not be nearly enough. The ResNet model seems to be the best performer after just 1 epoch. It is currently being run for 20 epochs. At the time of the submission of this report, it has not completed those 20 epochs and the results are therefore awaited for it. The results posted in the table above are those after just 1 epoch.

Future Goals:

If given more time, it would be worth considering various different parameters on each of these methods. Some of these would include trying out different activation functions.

For example, it could be worth to try the LeakyReLU activation function for each of the layers in the AlexNet architecture. It might also be worth looking into trying softmax activation in the last layer rather than sigmoid. This way, for each example, we would achieve two probability values rather than one. These probability values would correspond to the likelihood of each instance belonging to each of the classes. If using this, we would have to consider the categorical_crossentropy loss rather than binary_crossentropy.

While considering different parameters for the neural networks, it would also be worth checking the performance variation with different optimizers. Currently the Adam optimizer has been used in all the methods. There are other optimizers such as rmsprop.

Due to a number of paramters being present that could be changed and different combinations tested, with more time it would be beneficial to run cross-validation with these parameters for each of the networks to identify the best performing set of hyperparameters.

Another important point to consider with more time would be to try less number of dense layers in the transfer learning method. This method in particular requires more investigation as the validation accuracy does not change at all. It could also be that this just requires more epochs before the validation set sees a change in the performance.

Finally, it would also be worth to try a combination of ResNet and transfer learning. Currently the second method is a combination of ConvNet and transfer learning. We could also try transfer learning with a ResNet. This would involve training a ResNet on a larger dataset and storing those weight values and then training and running it on the Cats vs Dogs dataset.

In [ ]: