Machine Learning Engineer Nanodegree

Capstone Proposal

Sergio Noviello September 25th, 2018

Definition

Cervical cancer is a cause of death in women worldwide. In females, it is the 14th most common cancer. The mortality rates were drastically reduced with the introduction of the smear test. However, it is still important for doctors to visually review cervical images in order to correctly classify the cervix type. An incorrect classification will result in a treatment that will be ineffective and will also be very expensive.

This project is based on a competition that was launched on Kaggle a year ago.

Problem statement

The goal of this project is to develop a deep learning model that is capable to classify cervical images and predict the probability of each image to be of type 1, 2 or 3. This model will be very helpful for healthcare providers in identifying patients with a cervix of type 2 or type 3 that will lead to further testing.

The steps required are as follows:

- Download the train and test set from Kaggle.com
- Split the training set into train and validation set
- Convert images into multi-dimensional arrays
- Augment data by flipping, rotating and changing the offset of the images in the training set
- Build a deep learning model using a pre-trained model (ResNet50)
- Evaluate the model using the validation set
- Classify the images in the test set

Metrics

I have used the multi-class logarithmic loss as evaluation metric. For each image, a set of predicted probabilities (one for every class) needs to be calculated. The formula is shown in Figure 1:

Fig.1

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

N is the number of images in the test set, M is the number of categories, log is the natural logarithm, (y_{ij}) is 1 if observation (i) belongs to class (j) and 0 otherwise, and pij is the predicted probability that observation i belongs to class j.

Exploratory Data Analysis

number of images in the whole dataset

The training set consists of 8218 images, divided in 3 classes as shown below. 53% of the images are in type_2, 29.5% from type_2 and 17.5% from type_1 (Figure 2)

Type	Base	Additional	Total
type 1	251	1191	**1442**
type 2	782	3567	**4349**
type 3	451	1976	**2472**

Fig.2

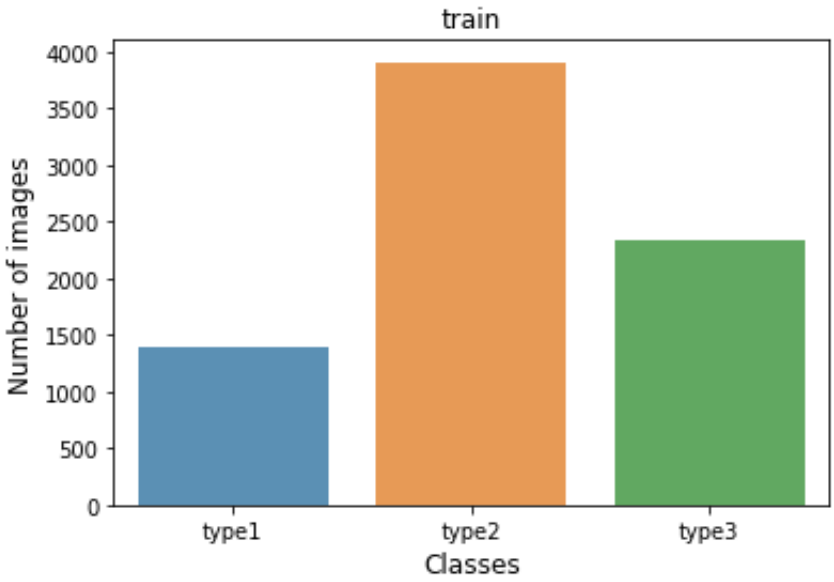


image sizes in the train set

There are 8 types of image size, the majority are 4128x3096 (Figure 3)

Fig.3

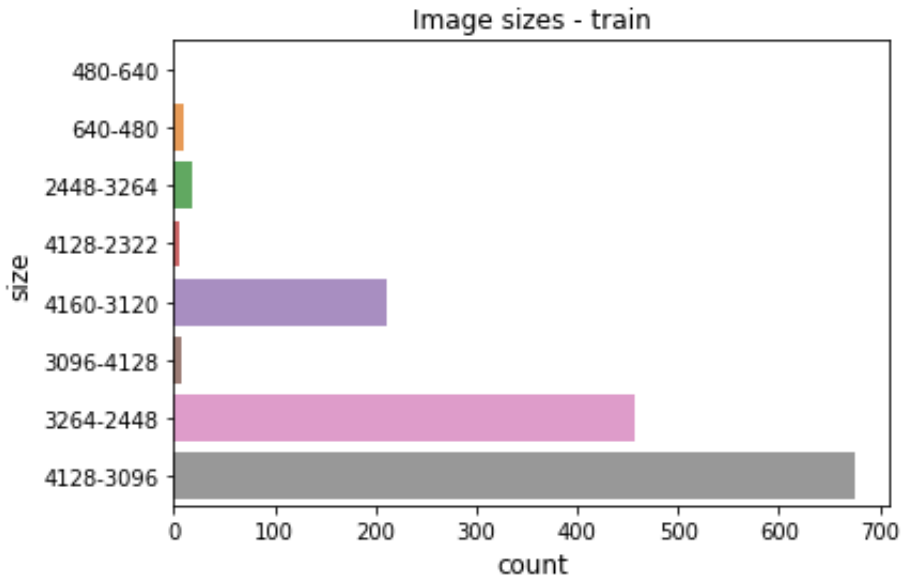
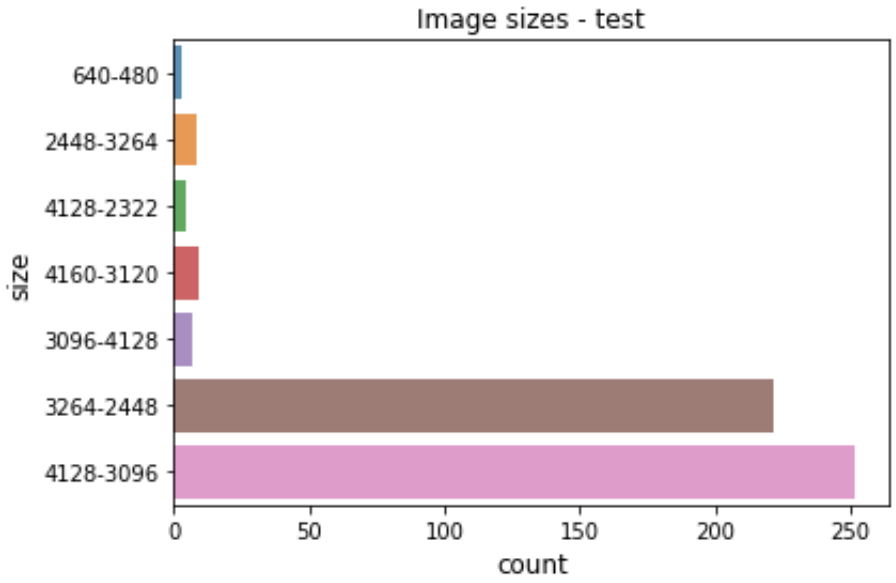


image sizes in the test set

The test dataset has a similar composition in terms of image sizes compared to the training set (Figure 4)

Fig.4

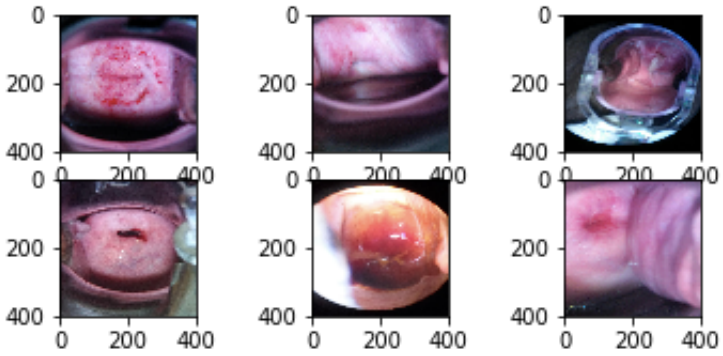


Offsets and angles

The relevant part of the image is not always centered and also the angle can vary as shown in Figure 5.

In this project I have used data augmentation to make minor alterations to the existing dataset. Minor changes such as flips or translations or rotations.

Fig.5



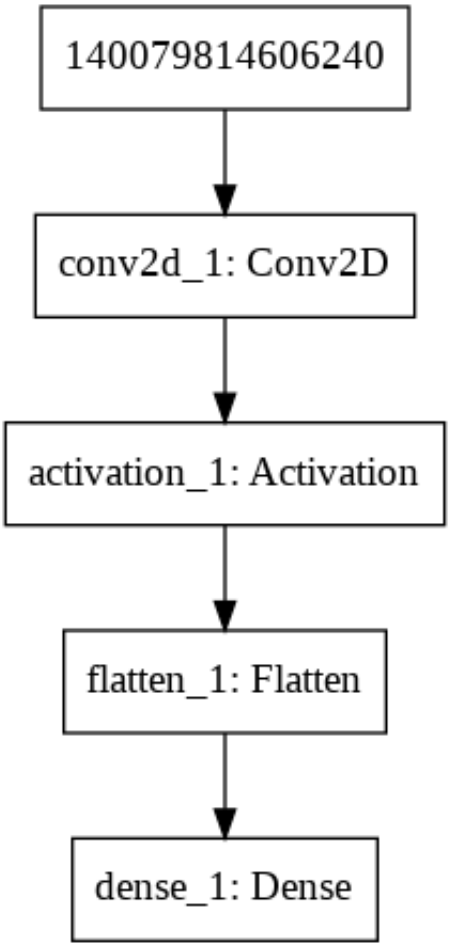
CNN Architecture

Benchmark model

A simple model was created as benchmark. I have used one convolutional layer with 3x3 filters followed by a relu activation layer. Finally I have flattened the layer into a monodimensional vector and added a fully connected layer which is responsible to assign probabilities for each class using the softmax algorithm.

```
    def build_model():
        model = Sequential()
        model.add(Conv2D(32, (3, 3), input_shape=(IMAGE_SIZE, IMAGE_SIZE, C
    HANNELS)))
        model.add(Activation('relu'))
        model.add(Flatten())
        model.add(Dense(3, activation='softmax'))
        model.summary()
        return model
```

Fig.6



Evaluate benchmark model

After training this simple model I have then evaluated the total loss against the validation set. The loss was calculated using the multi-class logloss function

```
def logloss_mc(y_true, y_prob, epsilon=1e-15):
    """ Multiclass logloss
    This function is not officially provided by Kaggle, so there is no
    guarantee for its correctness.
    https://github.com/ottogroup/kaggle/blob/master/benchmark.py
    """

    y_prob = y_prob / y_prob.sum(axis=1).reshape(-1, 1)
    y_prob = np.maximum(epsilon, y_prob)
    y_prob = np.minimum(1 - epsilon, y_prob)

    y = [y_prob[i, j] for (i, j) in enumerate(y_true)]
    ll = - np.mean(np.log(y))
    return ll
```

The total loss was: **11.512925148010254**

Tuning the model

I have added enhancements to the benchmark model gradually and I have calculated the total loss after each step to see if they would improve the score.

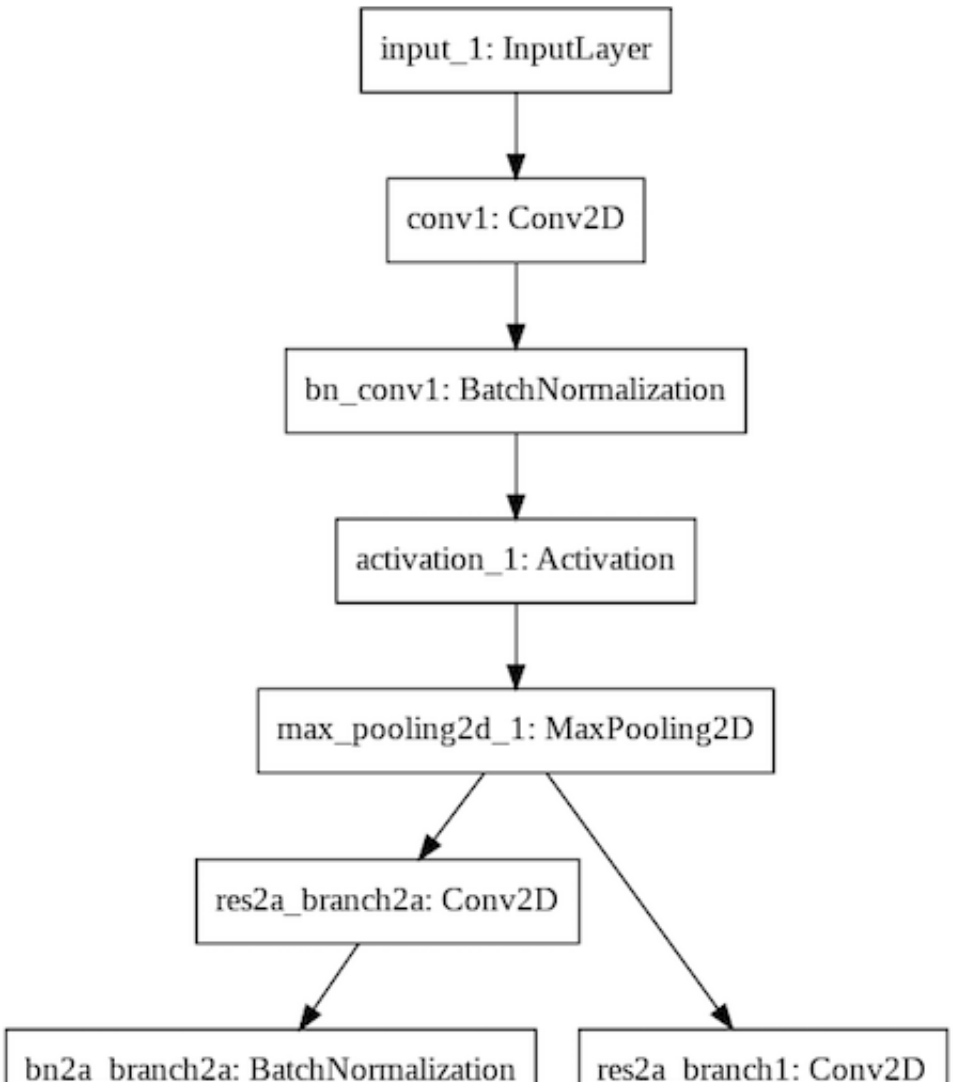
The first improvement was to use a technique called 'transfer learning'. Instead of building the model from scratch I have used the weights of a pre-trained model called ResNet50 that is included in the Keras library.

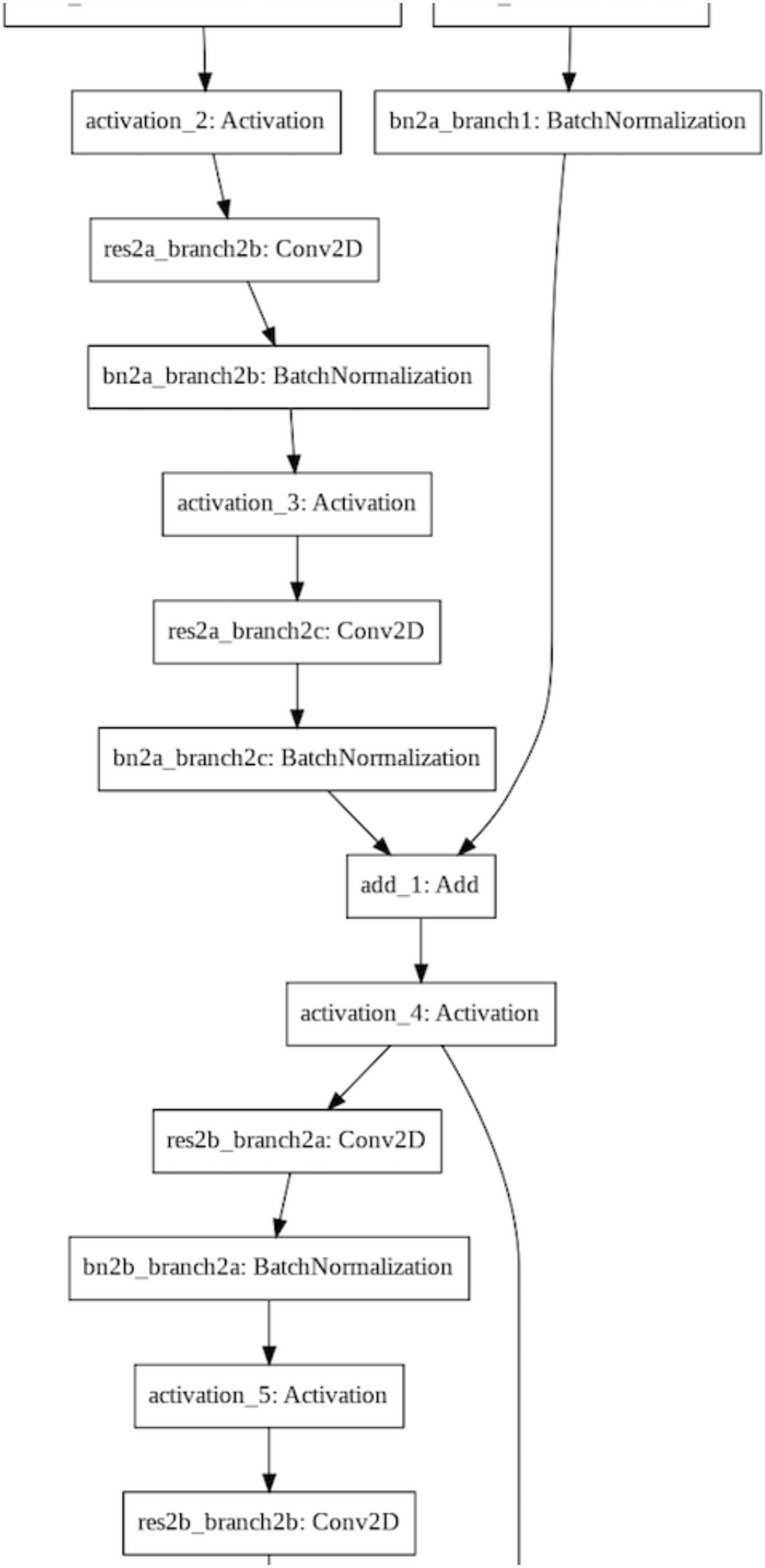
The model is trained on the Imagenet dataset. Imagenet is a database of about 1.5 million images each containing multiple labelled objects. This technique has been proved to be effective because the lower layers of these pre-trained models already contain many generic features such edge recognition and color detectors, features that are common in different datasets.

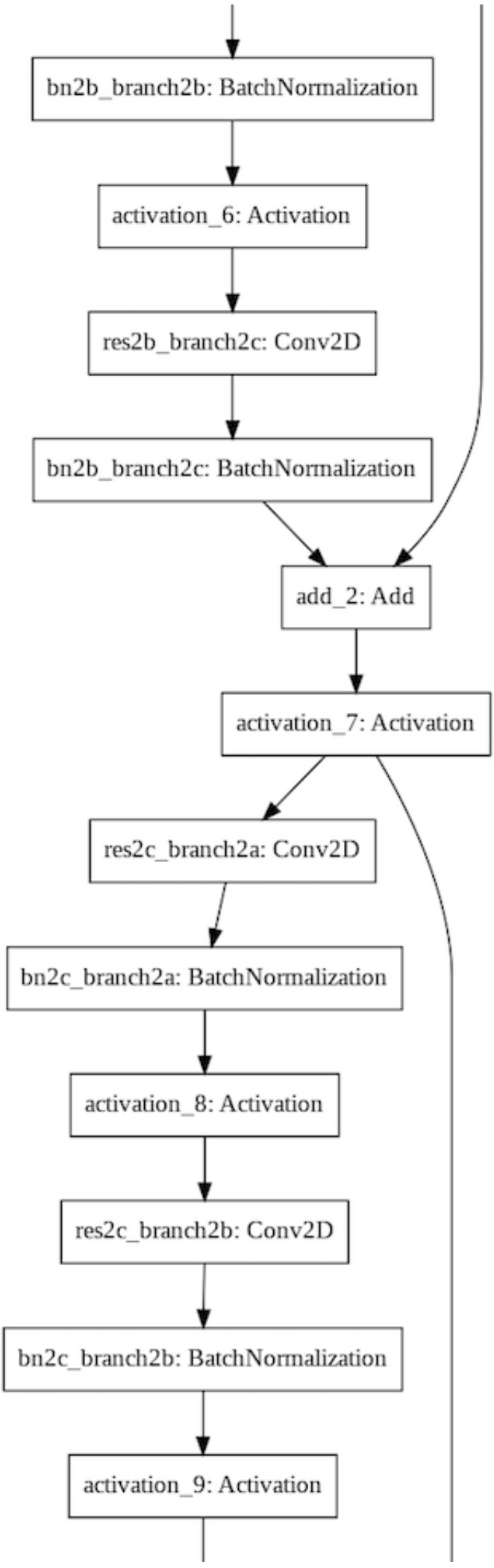
```
    def build_model():
        base_model = ResNet50(include_top=False,
                              weights='imagenet',
                              input_tensor=None,
                              input_shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS
    ))
        x = Flatten()(base_model.output)
        output = Dense(3, activation='softmax')(x)
        model = Model(inputs=base_model.input, outputs=output)
        model.summary()
        return model
```

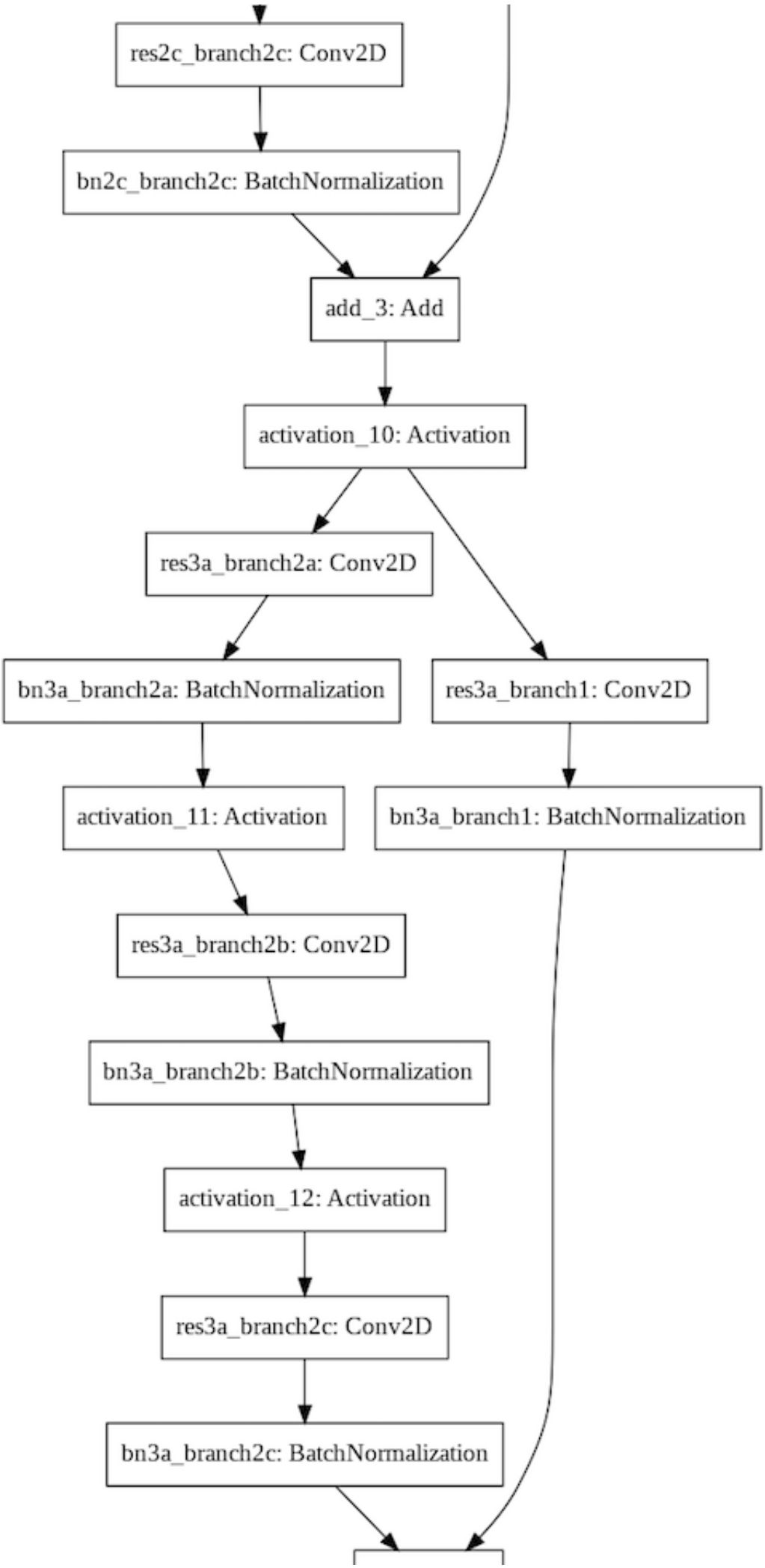
The architecture of the model is quite complex and illustrated in Figure 7.

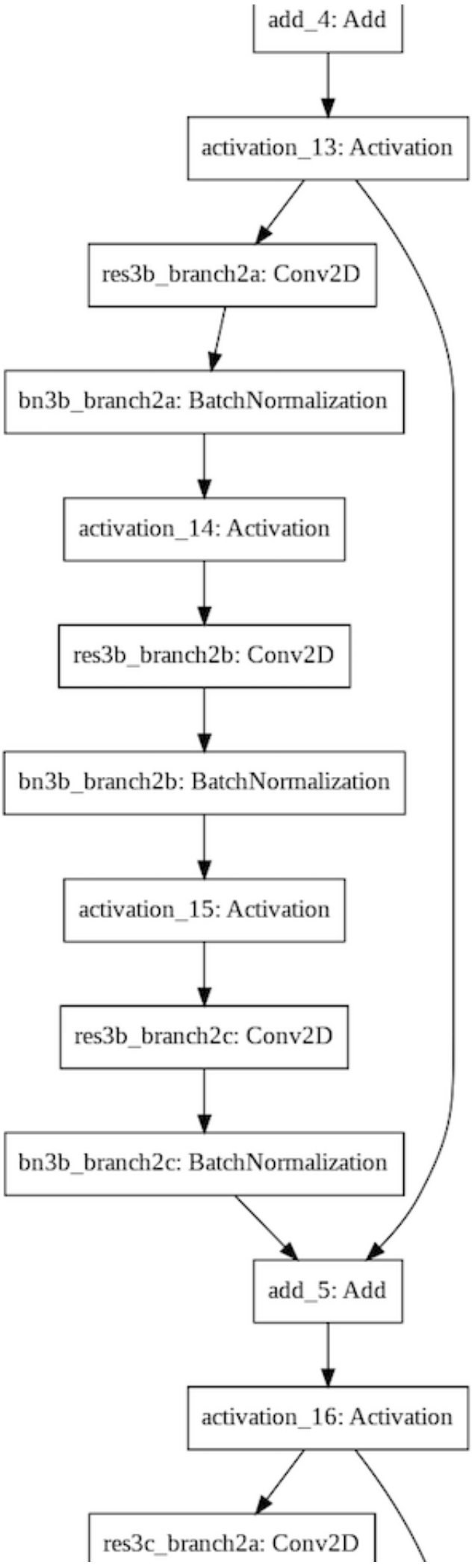
Fig.7

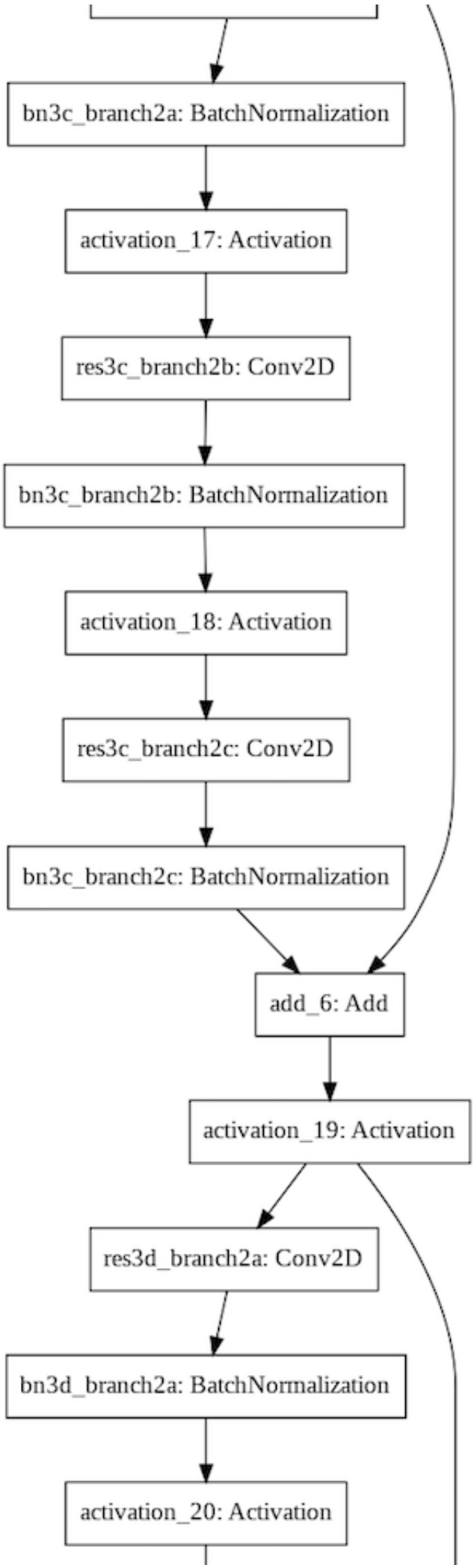


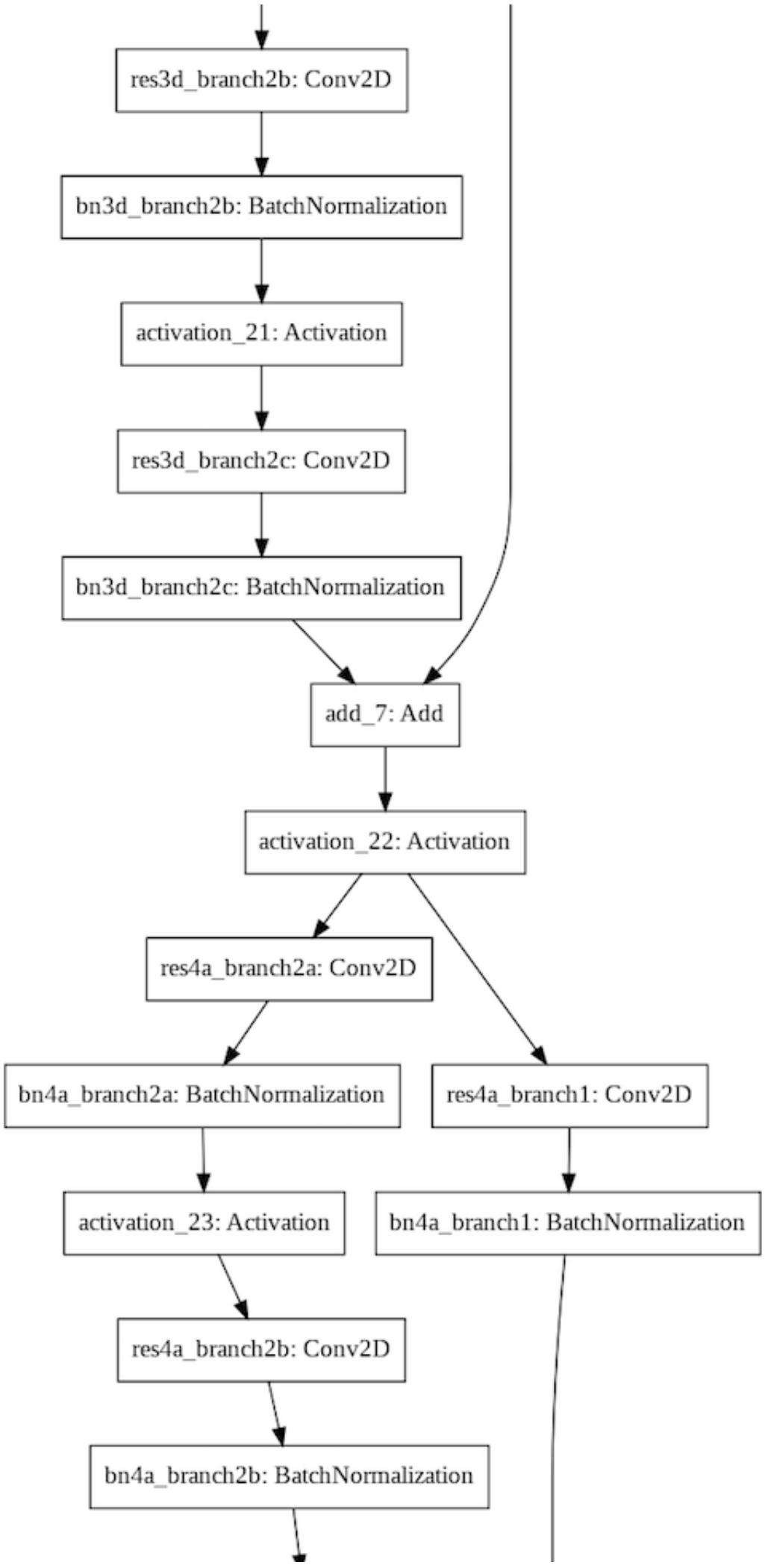

activation_2: Activation
bn2a_branch1: BatchNormalization
res2a_branch2b: Conv2D
bn2a_branch2b: BatchNormalization
activation_3: Activation
res2a_branch2c: Conv2D
bn2a_branch2c: BatchNormalization
add_1: Add
activation_4: Activation
res2b_branch2a: Conv2D
bn2b_branch2a: BatchNormalization
activation_5: Activation
res2b_branch2b: Conv2D

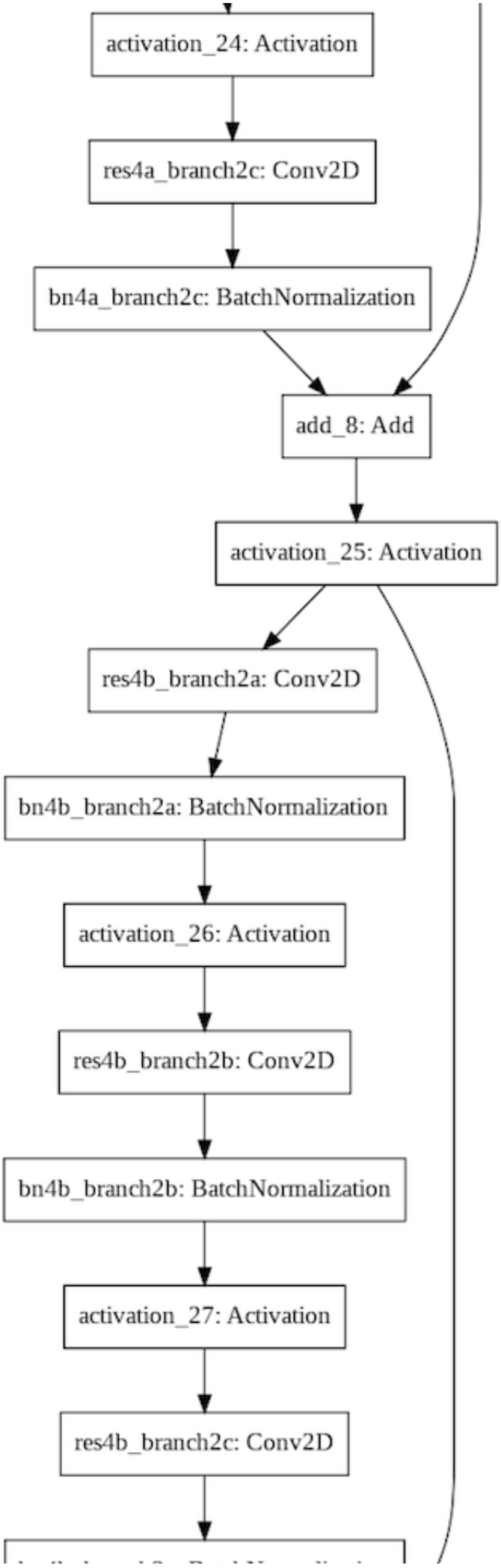

bn2b_branch2b: BatchNormalization
activation_6: Activation
res2b_branch2c: Conv2D
bn2b_branch2c: BatchNormalization
add_2: Add
activation_7: Activation
res2c_branch2a: Conv2D
bn2c_branch2a: BatchNormalization
activation_8: Activation
res2c_branch2b: Conv2D
bn2c_branch2b: BatchNormalization
activation_9: Activation

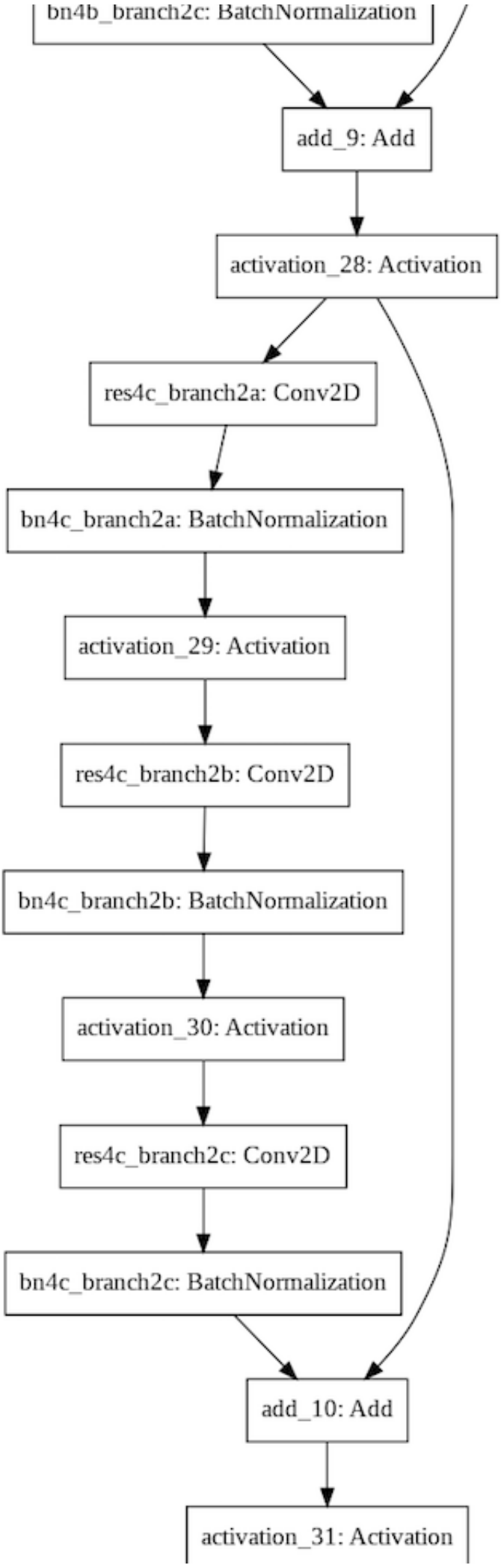

res2c_branch2c: Conv2D
bn2c_branch2c: BatchNormalization
add_3: Add
activation_10: Activation
res3a_branch2a: Conv2D
bn3a_branch2a: BatchNormalization
res3a_branch1: Conv2D
activation_11: Activation
bn3a_branch1: BatchNormalization
res3a_branch2b: Conv2D
bn3a_branch2b: BatchNormalization
activation_12: Activation
res3a_branch2c: Conv2D
bn3a_branch2c: BatchNormalization

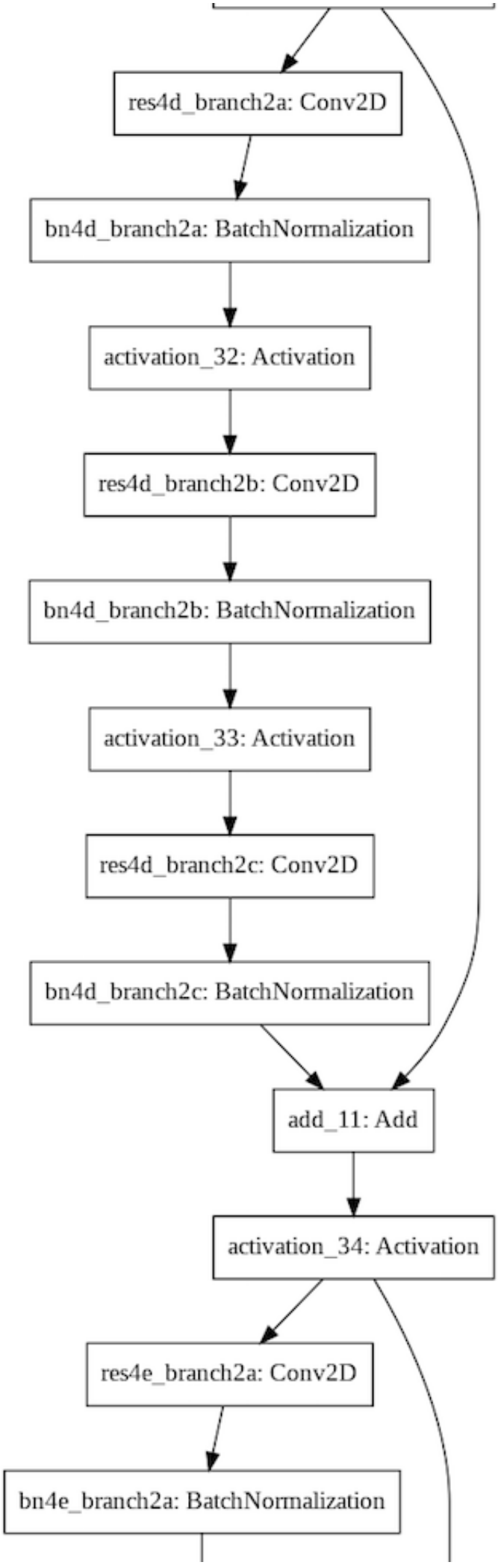

add_4: Add
activation_13: Activation
res3b_branch2a: Conv2D
bn3b_branch2a: BatchNormalization
activation_14: Activation
res3b_branch2b: Conv2D
bn3b_branch2b: BatchNormalization
activation_15: Activation
res3b_branch2c: Conv2D
bn3b_branch2c: BatchNormalization
add_5: Add
activation_16: Activation
res3c_branch2a: Conv2D

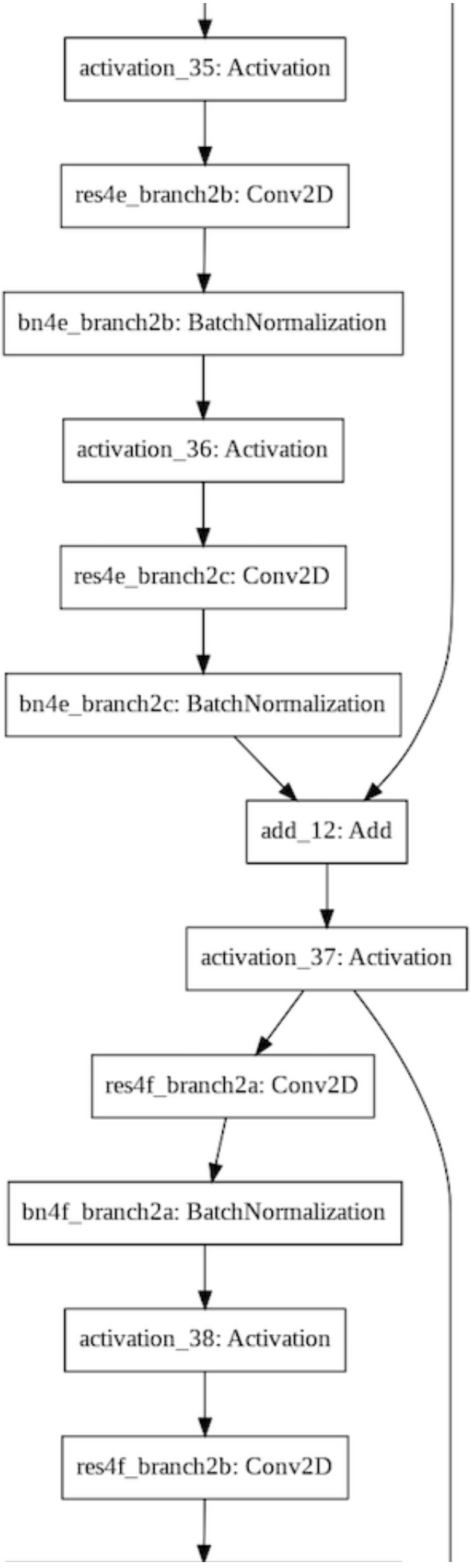

bn3c_branch2a: BatchNormalization
activation_17: Activation
res3c_branch2b: Conv2D
bn3c_branch2b: BatchNormalization
activation_18: Activation
res3c_branch2c: Conv2D
bn3c_branch2c: BatchNormalization
add_6: Add
activation_19: Activation
res3d_branch2a: Conv2D
bn3d_branch2a: BatchNormalization
activation_20: Activation

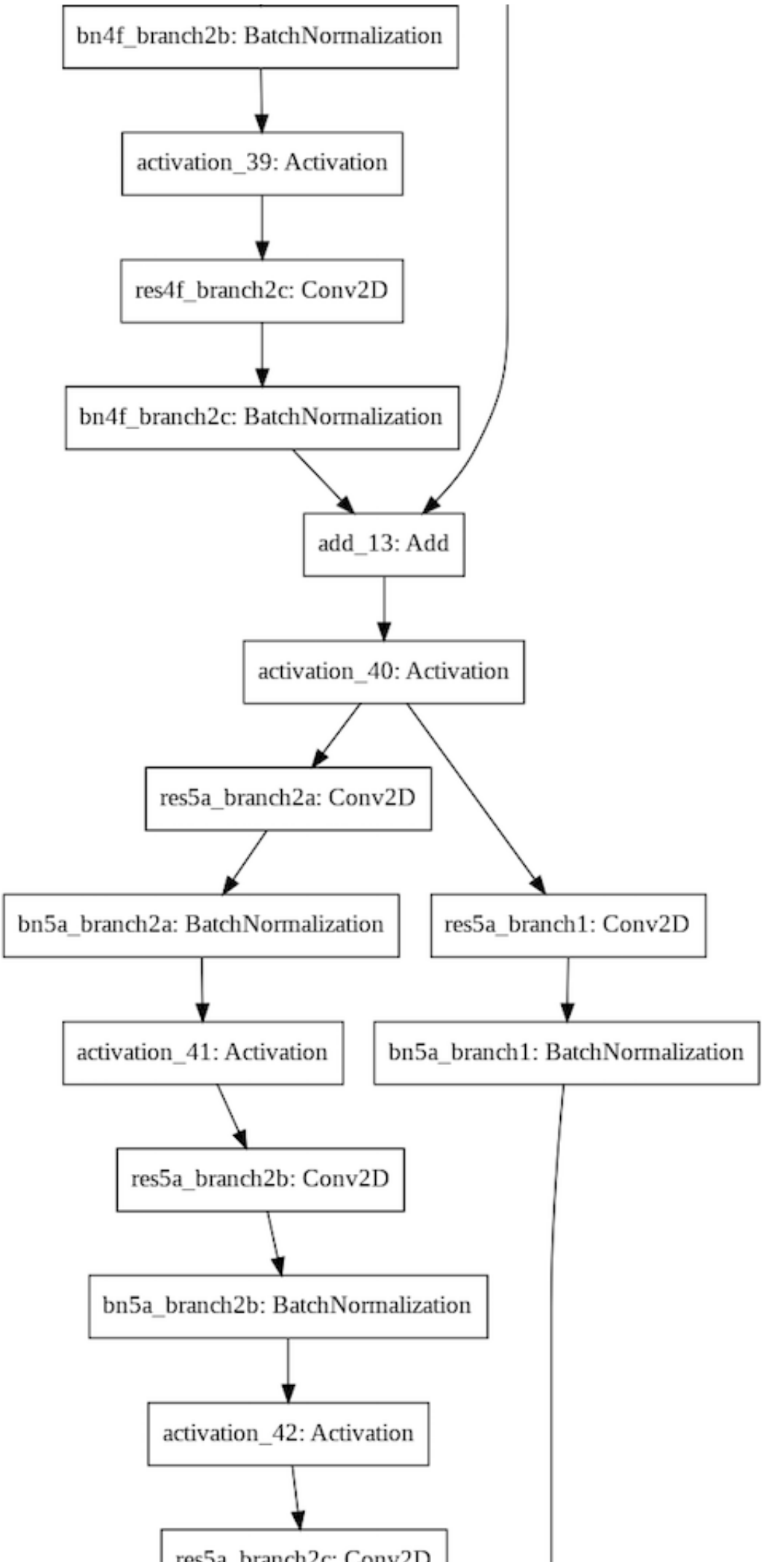

res3d_branch2b: Conv2D
bn3d_branch2b: BatchNormalization
activation_21: Activation
res3d_branch2c: Conv2D
bn3d_branch2c: BatchNormalization
add_7: Add
activation_22: Activation
res4a_branch2a: Conv2D
bn4a_branch2a: BatchNormalization
res4a_branch1: Conv2D
activation_23: Activation
bn4a_branch1: BatchNormalization
res4a_branch2b: Conv2D
bn4a_branch2b: BatchNormalization

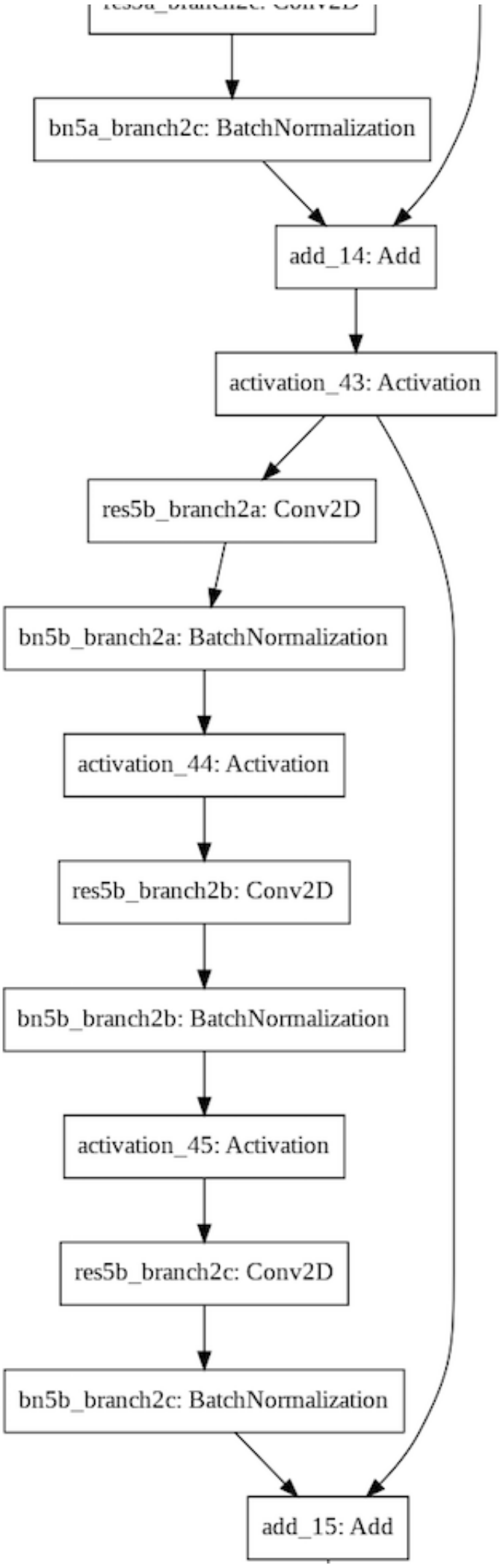

activation_24: Activation
res4a_branch2c: Conv2D
bn4a_branch2c: BatchNormalization
add_8: Add
activation_25: Activation
res4b_branch2a: Conv2D
bn4b_branch2a: BatchNormalization
activation_26: Activation
res4b_branch2b: Conv2D
bn4b_branch2b: BatchNormalization
activation_27: Activation
res4b_branch2c: Conv2D

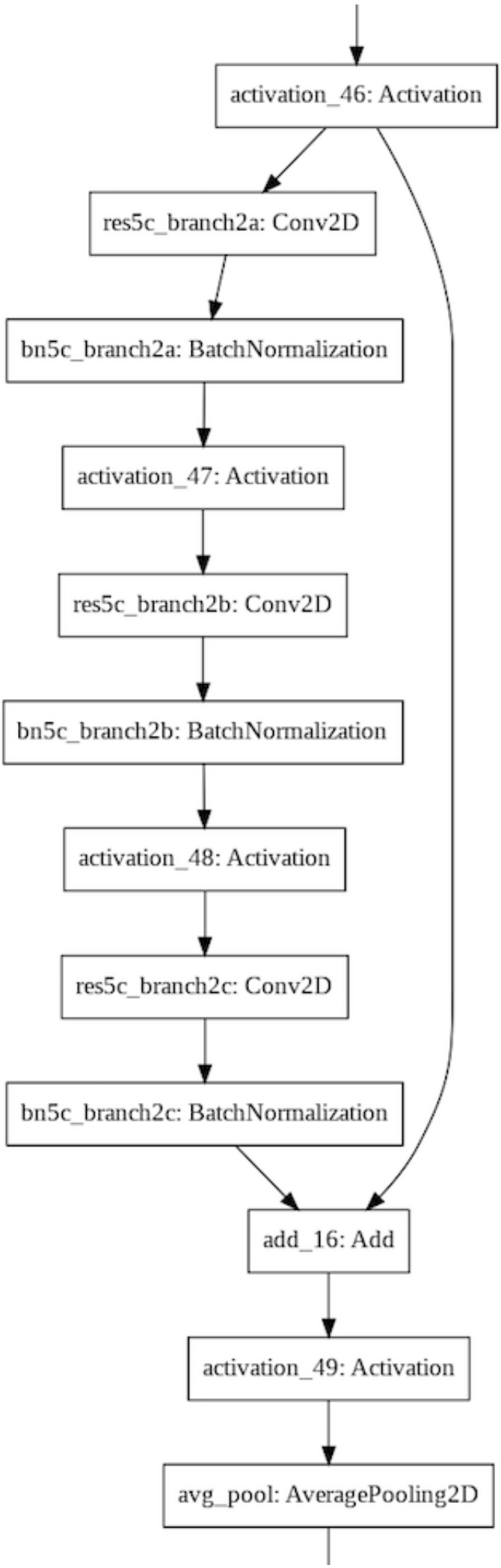

bn4b_branch2c: BatchNormalization
add_9: Add
activation_28: Activation
res4c_branch2a: Conv2D
bn4c_branch2a: BatchNormalization
activation_29: Activation
res4c_branch2b: Conv2D
bn4c_branch2b: BatchNormalization
activation_30: Activation
res4c_branch2c: Conv2D
bn4c_branch2c: BatchNormalization
add_10: Add
activation_31: Activation

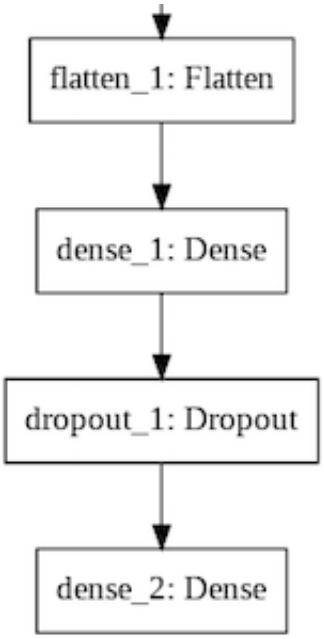

res4d_branch2a: Conv2D
bn4d_branch2a: BatchNormalization
activation_32: Activation
res4d_branch2b: Conv2D
bn4d_branch2b: BatchNormalization
activation_33: Activation
res4d_branch2c: Conv2D
bn4d_branch2c: BatchNormalization
add_11: Add
activation_34: Activation
res4e_branch2a: Conv2D
bn4e_branch2a: BatchNormalization


activation_35: Activation
res4e_branch2b: Conv2D
bn4e_branch2b: BatchNormalization
activation_36: Activation
res4e_branch2c: Conv2D
bn4e_branch2c: BatchNormalization
add_12: Add
activation_37: Activation
res4f_branch2a: Conv2D
bn4f_branch2a: BatchNormalization
activation_38: Activation
res4f_branch2b: Conv2D


bn4f_branch2b: BatchNormalization
activation_39: Activation
res4f_branch2c: Conv2D
bn4f_branch2c: BatchNormalization
add_13: Add
activation_40: Activation
res5a_branch2a: Conv2D
bn5a_branch2a: BatchNormalization
res5a_branch1: Conv2D
activation_41: Activation
bn5a_branch1: BatchNormalization
res5a_branch2b: Conv2D
bn5a_branch2b: BatchNormalization
activation_42: Activation


bn5a_branch2c: BatchNormalization
add_14: Add
activation_43: Activation
res5b_branch2a: Conv2D
bn5b_branch2a: BatchNormalization
activation_44: Activation
res5b_branch2b: Conv2D
bn5b_branch2b: BatchNormalization
activation_45: Activation
res5b_branch2c: Conv2D
bn5b_branch2c: BatchNormalization
add_15: Add


activation_46: Activation
res5c_branch2a: Conv2D
bn5c_branch2a: BatchNormalization
activation_47: Activation
res5c_branch2b: Conv2D
bn5c_branch2b: BatchNormalization
activation_48: Activation
res5c_branch2c: Conv2D
bn5c_branch2c: BatchNormalization
add_16: Add
activation_49: Activation
avg_pool: AveragePooling2D



Pre-processing and data augmentation

In this step I have split the dataset into training (80%) and validation set (20%). The reason for this, is that it is considered good practice to evaluate a model on data that is different to the data used in the training process.

The images were resized to 200x200 using the library **opencv** and converted to numpy arrays. Considering the size of these multi-dimensional arrays is not efficient to store them in memory, therefore I have used python generators. Generators are functions that behave like iterators, object that can be iterated upon. The benefit is the lazy evaluation, iterators don't compute the value of each item when instantiated. They only compute it when you ask for it.

```
image = cv2.imread(file_path)
image = cv2.resize(image, (IMAGE_SIZE, IMAGE_SIZE))
image = image.astype(np.float32) / 255.0
X[cnt, :, :, :] = image
class_index = labels.index(image_class)
Y[cnt, class_index] = 1
cnt += 1
if cnt == batch_size:
    yield (X, Y)
```

Another important aspect when training a neural network is to standardize the data to have zero mean and unit variance. During the training of a neural network the initial inputs are multiplied by the weights and added to the biases. Some parameters are shared in a neural network and, if they are not scaled and in the same range, for some part of the image a weight will appear big and to another it will be too small.

For this reason I have used the ImageDataGenerator class from Keras, which by default have the parameters featurewise_center and featurewise_std_normalization set to true.

The same class also allows you to augment the data by flipping, rotating or shifting horizontally or vertically the offset.

The parameters I have used as as follows:

Parameter	Value
rotation_range	25
width_shift_range	0.1
height_shift_range	0.1
shear_range	0.2

```
def create_data_generators(validation_set = False):
    data_generator = ImageDataGenerator(rescale=1./255.,
                                        rotation_range=25,
                                        width_shift_range=0.1,
                                        height_shift_range=0.1,
                                        shear_range=0.2,
                                        horizontal_flip=True,
                                        fill_mode="nearest")

    return data_generator.flow_from_directory(TR_DIR,
                                              target_size=(IMAGE_SIZE,
IMAGE_SIZE),
                                              shuffle=True,
                                              seed=SEED,
                                              class_mode='categorical',
                                              batch_size=BATCH_SIZE)
```

Training the model

During the training of this model I have noticed a gap between train and validation loss which is usually an indication of overfitting (Figure 8).

Fig.8

Reduce overfitting

I have decided to add to my model a fully connected layer followed by a dropout layer. Dropout is a regularization technique for reducing overfitting.

During training some nodes, chosen with 1-p probability, are left out the network. This helps preventing overfitting because it adds a penalty to the loss function so that it does not learn interdependent set of features weights.

```
    def build_model():
        base_model = ResNet50(include_top=False,
                              weights='imagenet',
                              input_tensor=None,
                              input_shape=(IMAGE_SIZE, IMAGE_SIZE, CHANNELS
    ))

        x = Flatten()(base_model.output)
        x = Dense(32, activation='relu')(x)
        x = Dropout(0.5)(x)
        output = Dense(3, activation='softmax')(x)
        model = Model(inputs=base_model.input, outputs=output)

        model.summary()
        return model
```

Optimizers and learning rate

When training a neural network the learning rate can be fixed or it can be reduced as the training progresses. For this I have chosen an adaptive learning rate method called **Adam**. The name is derived from *adaptive moment estimation*.

Adam computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradient.

Instead of adapting the parameter learning rates based on the average first moment as in RMSProp, Adam makes use of the average of the second moments of the gradients.

For the initial learning rate I have initially selected a lower value of **1.0e-4** but I noticed that the model was converging slowly, so I have decided to increase the learning rate to **1.0e-2**.

For the other parameters I have used the default values:

Parameter	Value	Description
beta_1	0.9	the exponential decay rate for the first moment estimates
beta_2	0.9999	the exponential decay rate for the second-moment estimates
epsilon	1e-08	a small number to prevent any division by zero

```
def compile_model(model):
    opt4 = optimizers.Adam(lr=LEARN_RATE,
                           beta_1=0.9,
                           beta_2=0.999,
                           epsilon=1e-08,
                           decay=0.0)

    model.compile(optimizer=opt4, loss='categorical_crossentropy', metr
ics=['accuracy'])
```

Batch size

Batches are used because, during the training process of a neural network, is not efficient to pass the entire dataset at once. Dividing the number of images in the training set by the batch size will give us the number of interations needed to completed one epoch. One epoch is complete when the entire dataset is passed forward and backward through the network.

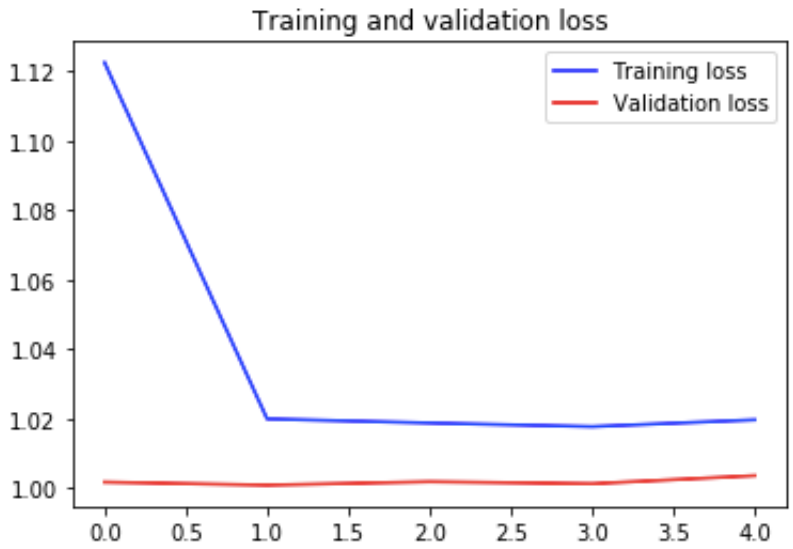
For my model I have chosen a batch size of 32. I have tried a higher batch size but I noticed that it was slightly overfitting the data.

For the number of epochs I started with 30 and I used the class ModelCheckpoint from Keras to check at the end of each epoch if the validation loss was improving. I have noticed that after 5 epochs the validation loss did not improve, therefore I have decided to set the number of epochs to five.

I have trained the model on the cloud using an Amazon EC2 instance with GPU for quicker computational power. The training process took nearly 5 hours.

Figure 9 shows a visualization of the train and validation loss during the training of the model.

Fig.9

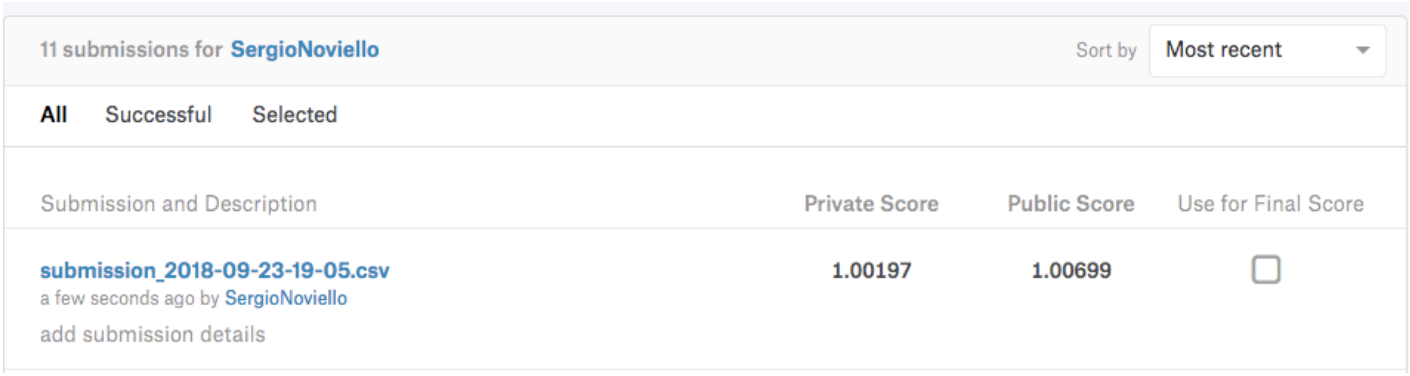


Conclusions

The total loss of the final model against the validation set is **1.32852661209**.

I have also used the model to make predictions against the test set and submitted the predictions to Kaggle.com Figure 10 shows a score of **1.00197** which would have resulted in position 170 on the private leaderboard.

Fig.10



Improvements

In order to achieve better results the model should be trained for longer than 5 epochs on a more powerful hardware.

References

- kaggle competition (https://www.kaggle.com/c/intel-mobileodt-cervical-cancer-screening)
- cervix types classification (https://kaggle2.blob.core.windows.net/competitions/kaggle/6243/media/Cervix%20types%20clasific
- keras documentation (https://keras.io/)
- log loss (https://github.com/ottogroup/kaggle/blob/master/benchmark.py)
- split data (https://github.com/keras-team/keras/issues/5862)
- data augmentation (https://machinelearningmastery.com)