Assignment4

Mehrdad Baradaran

99222020

Image_Reconstruction_Cifaar10

In this exercise, we have to select 1000 photos from the cifar10 dataset and as the main input of the neural network, we give the average of two arbitrary photos from these 1000 photos.

In this series, I tried different types of architectures to get the desired result and in this report I show 3 of the main architectures along with their results.

Various architectures including U-net, VAE and AutoEncoder, each of which will have an encoder and 2 decoders. And we compare the first output with the first label and the second output with the second label and calculate their loss with mse.

First of all, we extract the required libraries such as: tensorflow, pandas, numpy, keras, layers, models, etc.

```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from datetime import datetime
from packaging import version

%matplotlib inline
```

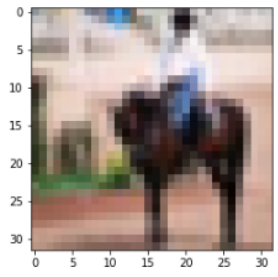
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Input, Dense, Dropout, MaxPooling2D, Conv2D, ReLU, Activation, UpSampling2D, Conv2DTransp
import tensorflow_probability as tfp
from tensorflow.keras.layers import Lambda, InputLayer, Flatten, Reshape, MaxPool2D, LeakyReLU, add
```

Next, we will load the cifar10 dataset and store it in test and train variables and show an example of them.

```
(train_x, train_y), (test_x, test_y) = tf.keras.datasets.cifar10.load_data()
```

```
plt.imshow(train_x[11])
```

```
<matplotlib.image.AxesImage at 0x7fd514cbf5e0>
```



After loading the data, we must normalize it by dividing it by 255 so that we can do the next part, which is the creation of the input data.

To do this, we randomly select 300 photos from two parts of the loaded data set, and by running two loops, we average them with each other and keep them in a list. Finally, we will have 45,000 generated data for neural network input.

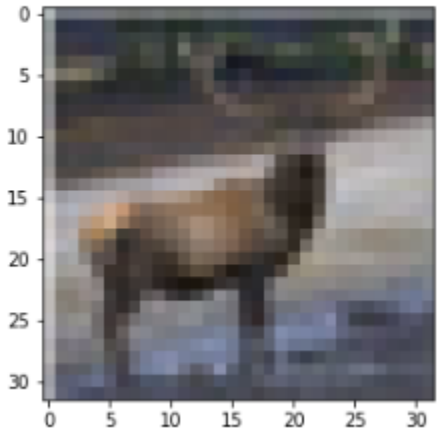
```
subtrain_x = train_x[10500:10800]
subtrain_x2 = train_x[47000:47300]
labels1 = []
labels2 = []
avg_input = []
for i in range(300):
    for j in range(i,300):
        labels1.append(subtrain_x[i])
        labels2.append(subtrain_x2[j])
        avg_input.append((subtrain_x[i]+subtrain_x2[j])/2)

print(len(labels1))
print(len(avg_input))
```

```
45150
45150
```

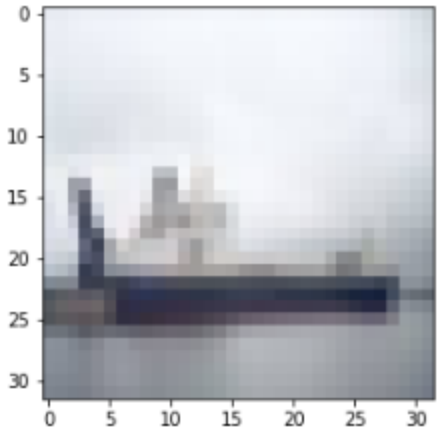
which is more than the requested 1000 data

To find out if we did this correctly, we show two sample photos and their output, which is the average of the two photos, to make sure.

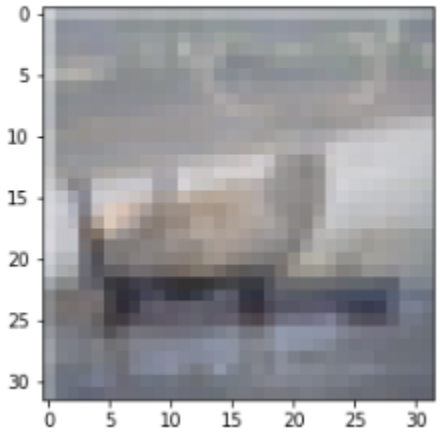
```
plt.imshow(labels1[0])
plt.show()
```



```
plt.imshow(labels2[0])
plt.show()
```




```
plt.imshow(avg_input[0])
plt.show()
```



In order for our neural network to work with average data, we need to convert all inputs to numpy array.

```
type(train_x), type(avg_input)
```

```
(numpy.ndarray, list)
```

```
avg_input = np.array(avg_input)
labels1 = np.array(labels1)
labels2 = np.array(labels2)
```

```
type(avg_input)
```

```
numpy.ndarray
```

```
avg_input.shape
```

```
(45150, 32, 32, 3)
```

Before presenting the main neural network architecture, which provides a more favorable result, we train two architectures, VAE and Autoencoder, and display the outputs.

```
# Encoder
x = tensorflow.keras.layers.Input(shape=(img_size, img_size, num_channels), name="encoder_input")

encoder_conv_layer1 = tensorflow.keras.layers.Conv2D(filters=1, kernel_size=(3, 3), padding="same", strides=1, name="encoder_
encoder_norm_layer1 = tensorflow.keras.layers.BatchNormalization(name="encoder_norm_1")(encoder_conv_layer1)
encoder_activ_layer1 = tensorflow.keras.layers.LeakyReLU(name="encoder_leakyrelu_1")(encoder_norm_layer1)

encoder_conv_layer2 = tensorflow.keras.layers.Conv2D(filters=32, kernel_size=(3,3), padding="same", strides=1, name="encoder_
encoder_norm_layer2 = tensorflow.keras.layers.BatchNormalization(name="encoder_norm_2")(encoder_conv_layer2)
encoder_activ_layer2 = tensorflow.keras.layers.LeakyReLU(name="encoder_activ_layer_2")(encoder_norm_layer2)

encoder_conv_layer3 = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3), padding="same", strides=2, name="encoder_
encoder_norm_layer3 = tensorflow.keras.layers.BatchNormalization(name="encoder_norm_3")(encoder_conv_layer3)
encoder_activ_layer3 = tensorflow.keras.layers.LeakyReLU(name="encoder_activ_layer_3")(encoder_norm_layer3)

encoder_conv_layer4 = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3), padding="same", strides=2, name="encoder_
encoder_norm_layer4 = tensorflow.keras.layers.BatchNormalization(name="encoder_norm_4")(encoder_conv_layer4)
encoder_activ_layer4 = tensorflow.keras.layers.LeakyReLU(name="encoder_activ_layer_4")(encoder_norm_layer4)

encoder_conv_layer5 = tensorflow.keras.layers.Conv2D(filters=64, kernel_size=(3,3), padding="same", strides=1, name="encoder_
encoder_norm_layer5 = tensorflow.keras.layers.BatchNormalization(name="encoder_norm_5")(encoder_conv_layer5)
encoder_activ_layer5 = tensorflow.keras.layers.LeakyReLU(name="encoder_activ_layer_5")(encoder_norm_layer5)

shape_before_flatten = tensorflow.keras.backend.int_shape(encoder_activ_layer5)[1:]
encoder_flatten = tensorflow.keras.layers.Flatten()(encoder_activ_layer5)

encoder_mu = tensorflow.keras.layers.Dense(units=latent_space_dim, name="encoder_mu")(encoder_flatten)
encoder_log_variance = tensorflow.keras.layers.Dense(units=latent_space_dim, name="encoder_log_variance")(encoder_flatten)

encoder_mu_log_variance_model = tensorflow.keras.models.Model(x, (encoder_mu, encoder_log_variance), name="encoder_mu_log_var

def sampling(mu_log_variance):
    mu, log_variance = mu_log_variance
    epsilon = tensorflow.keras.backend.random_normal(shape=tensorflow.keras.backend.shape(mu), mean=0.0, stddev=1.0)
    random_sample = mu + tensorflow.keras.backend.exp(log_variance/2) * epsilon
    return random_sample

encoder_output = tensorflow.keras.layers.Lambda(sampling, name="encoder_output")([encoder_mu, encoder_log_variance])

encoder = tensorflow.keras.models.Model(x, encoder_output, name="encoder_model")
encoder.summary()
```

```
decoder_input = tensorflow.keras.layers.Input(shape=(latent_space_dim), name="decoder_input")
decoder_dense_layer1 = tensorflow.keras.layers.Dense(units=np.prod(shape_before_flatten), name="decoder_dense_1")(decoder_inp
decoder_reshape = tensorflow.keras.layers.Reshape(target_shape=shape_before_flatten)(decoder_dense_layer1)
decoder_conv_tran_layer1 = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), padding="same", strides=1,
decoder_norm_layer1 = tensorflow.keras.layers.BatchNormalization(name="decoder_norm_1")(decoder_conv_tran_layer1)
decoder_activ_layer1 = tensorflow.keras.layers.LeakyReLU(name="decoder_leakyrelu_1")(decoder_norm_layer1)

decoder_conv_tran_layer2 = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), padding="same", strides=2,
decoder_norm_layer2 = tensorflow.keras.layers.BatchNormalization(name="decoder_norm_2")(decoder_conv_tran_layer2)
decoder_activ_layer2 = tensorflow.keras.layers.LeakyReLU(name="decoder_leakyrelu_2")(decoder_norm_layer2)

decoder_conv_tran_layer3 = tensorflow.keras.layers.Conv2DTranspose(filters=32, kernel_size=(3, 3), padding="same", strides=2,
decoder_norm_layer3 = tensorflow.keras.layers.BatchNormalization(name="decoder_norm_3")(decoder_conv_tran_layer3)
decoder_activ_layer3 = tensorflow.keras.layers.LeakyReLU(name="decoder_leakyrelu_3")(decoder_norm_layer3)

decoder_output = tensorflow.keras.layers.Conv2DTranspose(filters=3, kernel_size=(3, 3),activation='sigmoid', padding="same",

decoder = tensorflow.keras.models.Model(decoder_input, decoder_output, name="decoder_model")
```

```
decoder_input = tensorflow.keras.layers.Input(shape=(latent_space_dim), name="decoder2_input")
decoder_dense_layer1 = tensorflow.keras.layers.Dense(units=np.prod(shape_before_flatten), name="decoder2_dense_1")(decoder_ir
decoder_reshape = tensorflow.keras.layers.Reshape(target_shape=shape_before_flatten)(decoder_dense_layer1)
decoder_conv_tran_layer1 = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), padding="same", strides=1,
decoder_norm_layer1 = tensorflow.keras.layers.BatchNormalization(name="decoder2_norm_1")(decoder_conv_tran_layer1)
decoder_activ_layer1 = tensorflow.keras.layers.LeakyReLU(name="decoder2_leakyrelu_1")(decoder_norm_layer1)

decoder_conv_tran_layer2 = tensorflow.keras.layers.Conv2DTranspose(filters=64, kernel_size=(3, 3), padding="same", strides=2,
decoder_norm_layer2 = tensorflow.keras.layers.BatchNormalization(name="decoder2_norm_2")(decoder_conv_tran_layer2)
decoder_activ_layer2 = tensorflow.keras.layers.LeakyReLU(name="decoder2_leakyrelu_2")(decoder_norm_layer2)

decoder_conv_tran_layer3 = tensorflow.keras.layers.Conv2DTranspose(filters=32, kernel_size=(3, 3), padding="same", strides=2,
decoder_norm_layer3 = tensorflow.keras.layers.BatchNormalization(name="decoder2_norm_3")(decoder_conv_tran_layer3)
decoder_activ_layer3 = tensorflow.keras.layers.LeakyReLU(name="decoder2_leakyrelu_3")(decoder_norm_layer3)

decoder_output = tensorflow.keras.layers.Conv2DTranspose(filters=3, kernel_size=(3, 3),activation='sigmoid', padding="same",

decoder2 = tensorflow.keras.models.Model(decoder_input, decoder_output, name="decoder2_model")
```

This is a VAE architecture with 2 decoders, we compile the model with loss and mse method, as well as the desired learning_rate, which is 0.001, then we train it with 30 epochs.

```
vae.compile(optimizer=tensorflow.keras.optimizers.Adam(lr=0.001), loss='mse')
```

```
vae.fit(avg_input, [labels1, labels2], epochs=30, batch_size=64,
                      validation_data=(test_input, [test_labels1, test_labels2]), callbacks=[tensorboard_callback])
```

Before seeing the results, some details need to be explained.

For example, what is tensorboard_callback? Why was it used or how was the test data constructed?

First of all, by using the following commands, you can have the model in full with many details and in the form of a graph in a javascript format after training, and with the command that comes after that, full details will

be displayed. These details are represented by graphs and charts and even theoretical data

```
from tensorflow.keras.callbacks import EarlyStopping, TensorBoard
```

```
early_stop = EarlyStopping(monitor='val_loss', mode='min', patience=5, restore_best_weights=True)
logdir="logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=logdir)
```

```
%load_ext tensorboard

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

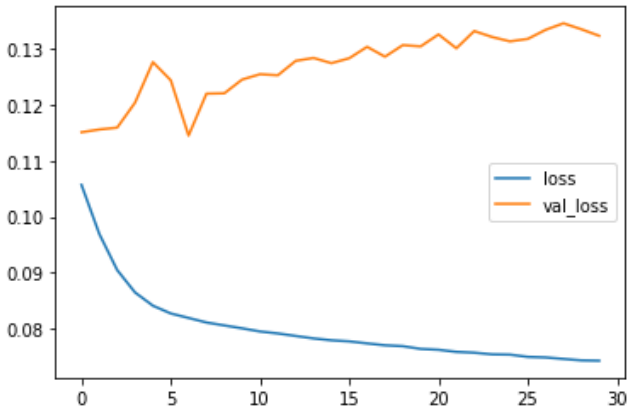
```
%tensorboard --logdir logs
```

The general details of the model are as follows:

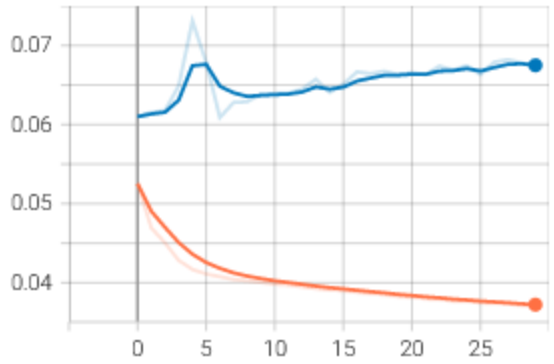
```
Model: "VAE"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
==================================================================================================
 VAE_input (InputLayer)      [(None, 32, 32, 3)]       0           []

 encoder_model (Functional)  (None, 2)                 109988      ['VAE_input[0][0]']

 decoder_model (Functional)  (None, 32, 32, 3)         125571      ['encoder_model[0][0]']

 decoder2_model (Functional)  (None, 32, 32, 3)        125571      ['encoder_model[0][0]']

==================================================================================================
Total params: 361,130
Trainable params: 359,912
Non-trainable params: 1,218
_________________________________________________________________________________________________
```

The loss of the model is calculated in this way: the loss of the first output and the first label + the loss of the second output with the second label.

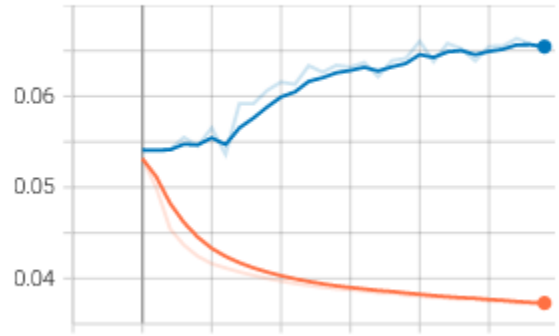
But in this model, as the epochs increase, our val_loss increases. In the end, it does not give me a good output.



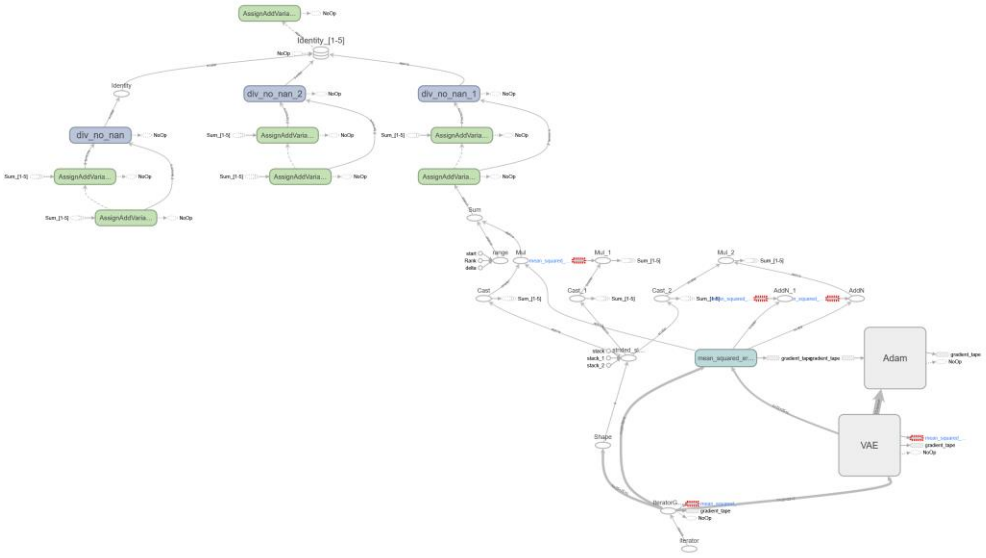
epoch_decoder_model_loss
tag: epoch_decoder_model_loss

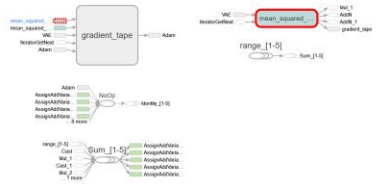


epoch_decoder2_model_loss
tag: epoch_decoder2_model_loss

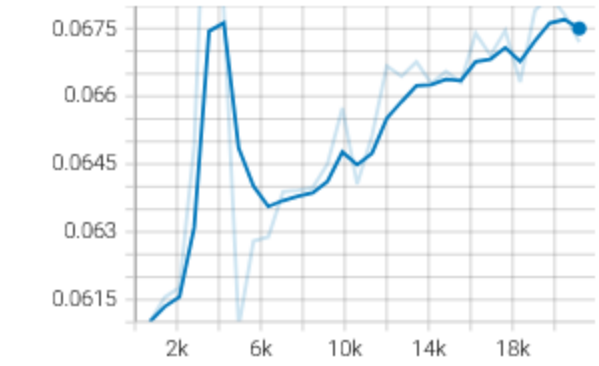


The graph output of the model and tensorboard_callback is as follows:

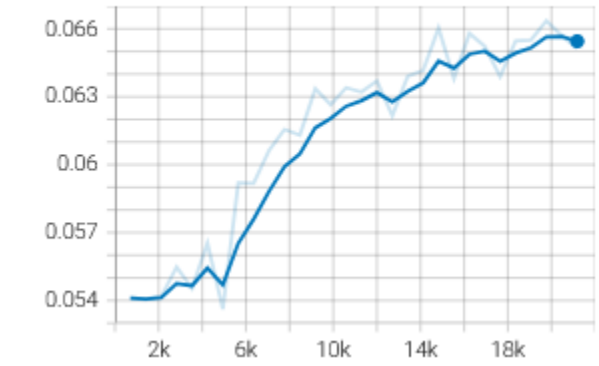




evaluation_decoder_model_loss_vs_iterations
tag: evaluation_decoder_model_loss_vs_iterations



evaluation_decoder2_model_loss_vs_iterations
tag: evaluation_decoder2_model_loss_vs_iterations

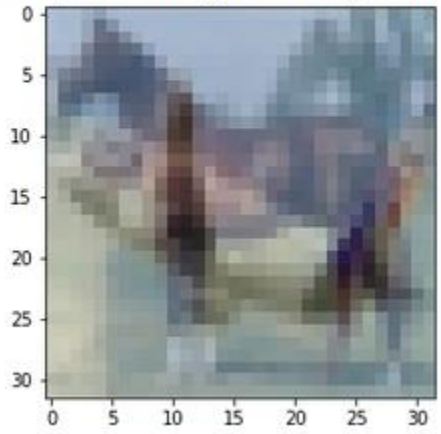


Finally, we display the test data outputs.

As can be seen, the outputs are not clear and favorable.

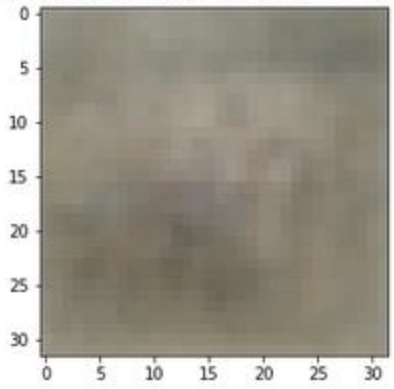
```
plt.imshow(test_input[4011])
```

```
<matplotlib.image.AxesImage at 0x7f4555fa5520>
```



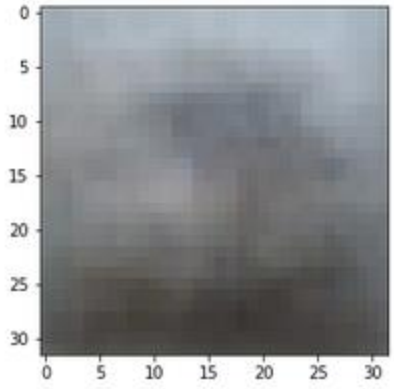
```
plt.imshow(decoded_data[4011])
```

```
<matplotlib.image.AxesImage at 0x7f4555f73fd0>
```




```
plt.imshow(decoded_data2[4011])
```

```
<matplotlib.image.AxesImage at 0x7f4555ec5af0>
```



In the following, for the architecture of Autoencoder, we will show the details of the architecture, outputs and loss in order:

Architecture :

```
# Encoder
inputs = Input(shape=(32, 32, 3))

x = Conv2D(32, 3, activation='relu', padding='same')(inputs)
x = BatchNormalization()(x)
x = MaxPool2D()(x)
x = Dropout(0.5)(x)
skip = Conv2D(32, 3, padding='same')(x) # skip connection for decoder
x = LeakyReLU()(skip)
x = BatchNormalization()(x)
x = MaxPool2D()(x)
x = Dropout(0.5)(x)
x = Conv2D(64, 3, activation='relu', padding='same')(x)
x = BatchNormalization()(x)
encoder = MaxPool2D()(x)
```

```
# Decoder
x = Conv2DTranspose(64, 3,activation='relu',strides=(2,2), padding='same')(encoder)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Conv2DTranspose(32, 3, activation='relu',strides=(2,2), padding='same')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Conv2DTranspose(32, 3, padding='same')(x)
x = add([x,skip]) # adding skip connection
x = LeakyReLU()(x)
x = BatchNormalization()(x)
decoder = Conv2DTranspose(3, 3, activation='sigmoid',strides=(2,2), padding='same')(x)
```

```
# Decoder2
x = Conv2DTranspose(64, 3,activation='relu',strides=(2,2), padding='same')(encoder)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Conv2DTranspose(32, 3, activation='relu',strides=(2,2), padding='same')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Conv2DTranspose(32, 3, padding='same')(x)
x = add([x,skip]) # adding skip connection
x = LeakyReLU()(x)
x = BatchNormalization()(x)
decoder2 = Conv2DTranspose(3, 3, activation='sigmoid',strides=(2,2), padding='same')(x)
```

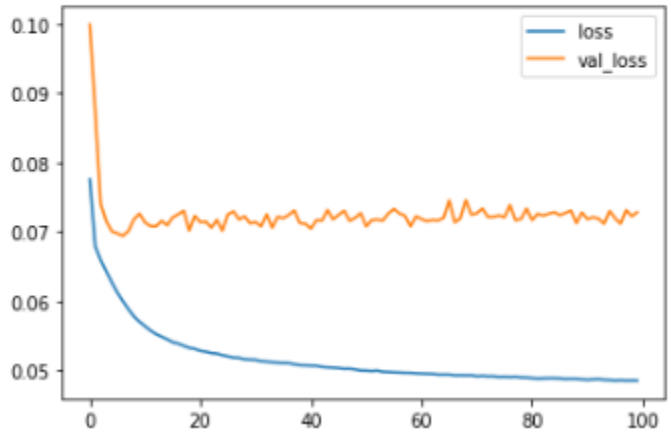
Detail and Parameters:

```
Total params: 161,190
Trainable params: 160,422
Non-trainable params: 768
_________________________________________________________________
_____________
```

Loss Plot:

```
model_history = pd.DataFrame(autoencode_model.history.history)
model_history[['loss', 'val_loss']].plot()
```

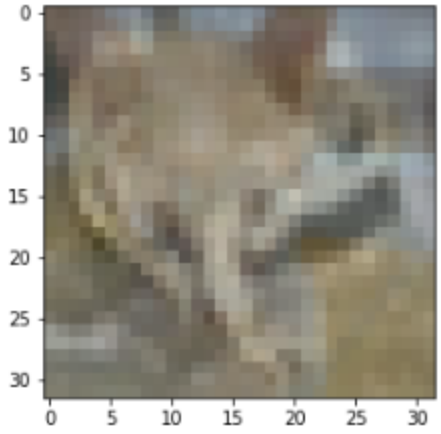
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fcb7614a0d0>
```



It is clear in the diagram that val_loss and loss do not change from another interval and the network repeats its training. In the following, you will clearly see that this network is completely dependent on the input and is not dependent on the labels, as a result, the outputs will clearly be very similar to the input.

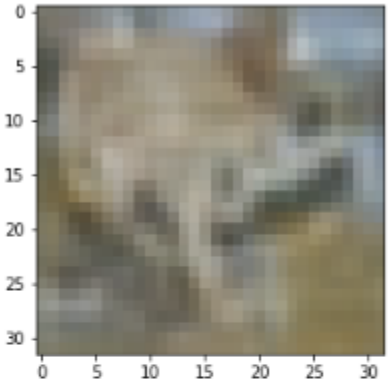
```
plt.imshow(test_input[760])
```

```
<matplotlib.image.AxesImage at 0x7fcbdfbe7400>
```



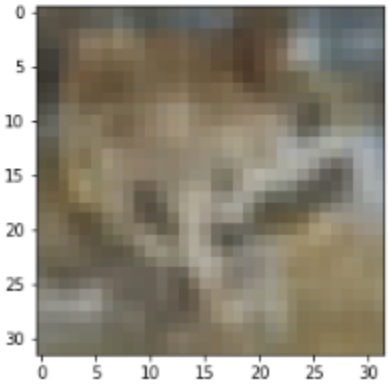

```
plt.imshow(predictions[1][760])
```

```
<matplotlib.image.AxesImage at 0x7fcbdfbb2fd0>
```




```
plt.imshow(predictions[0][760])
```

```
<matplotlib.image.AxesImage at 0x7fcbdfb89a60>
```



Finally, we introduce the architecture that provides much better results than the last two models, which is the U-net architecture.

Architecture:

Encoder:

```
#Input Layer
input = Input(shape=(32, 32, 3))


conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input)
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool2)
conv3 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(pool3)
conv4 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
conv5 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool4)
encoder = Conv2D(512, (3, 3), activation='relu', padding='same')(conv5)
```

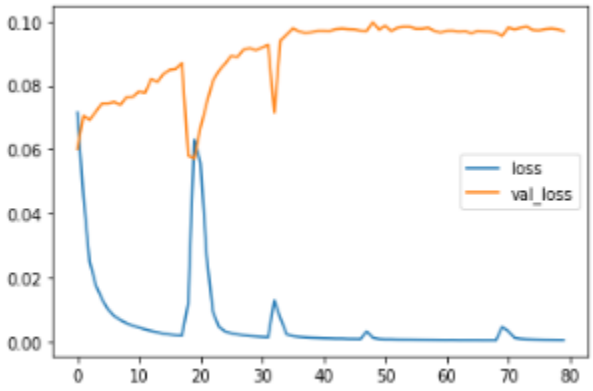
Decoders:

```
#decoder1

up6 = concatenate([Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(encoder), conv4], axis=3)
conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(up6)
conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv6)
up7 = concatenate([Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(conv6), conv3], axis=3)
conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(up7)
conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv7)
up8 = concatenate([Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(conv7), conv2], axis=3)
conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(up8)
conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv8)
up9 = concatenate([Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(conv8), conv1], axis=3)
conv9 = Conv2D(32, (3, 3), activation='relu', padding='same')(up9)
conv9 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv9)
decoder = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv9)

#decoder2
up6 = concatenate([Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(encoder), conv4], axis=3)
conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(up6)
conv6 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv6)
up7 = concatenate([Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same')(conv6), conv3], axis=3)
conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(up7)
conv7 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv7)
up8 = concatenate([Conv2DTranspose(64, (2, 2), strides=(2, 2), padding='same')(conv7), conv2], axis=3)
conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(up8)
conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv8)
up9 = concatenate([Conv2DTranspose(32, (2, 2), strides=(2, 2), padding='same')(conv8), conv1], axis=3)
conv9 = Conv2D(32, (3, 3), activation='relu', padding='same')(up9)
conv9 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv9)
decoder2 = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(conv9)
```

Summary:

```
=================================================================================================
Total params: 10,809,638
Trainable params: 10,809,638
Non-trainable params: 0
```

Loss:

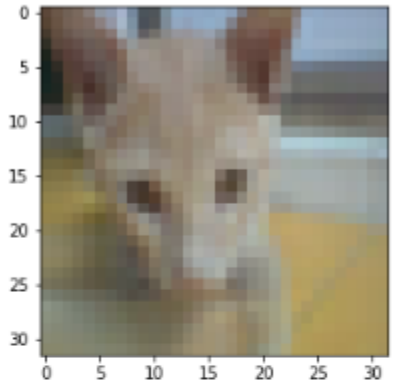


We notice that val_loss does not have much dependence on Loss and this is logical because the outputs should be close to their labels and not to the input, which is sometimes incomprehensible. On the other hand, the inputs are the average of both data. So it is desirable that val_loss updates itself with its labels

Finally, we check the inputs and outputs.

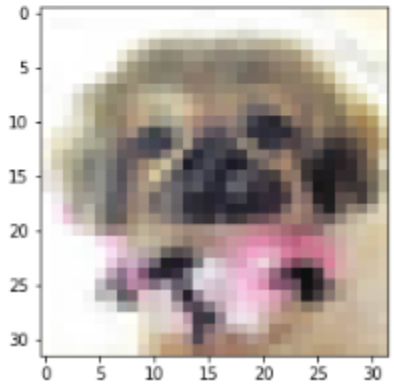
```
plt.imshow(test_labels1[512])
```

```
<matplotlib.image.AxesImage at 0x7fd3916629d0>
```



```
plt.imshow(test_labels2[512])
```

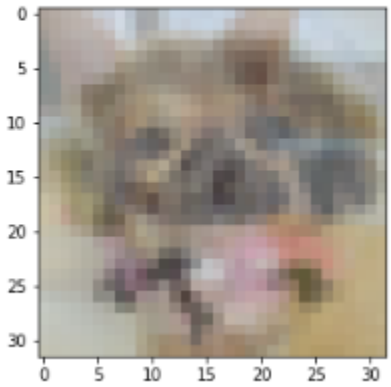
```
<matplotlib.image.AxesImage at 0x7fd3915d0160>
```



Input:

```
plt.imshow(test_input[512])
```

```
<matplotlib.image.AxesImage at 0x7fd3915998b0>
```

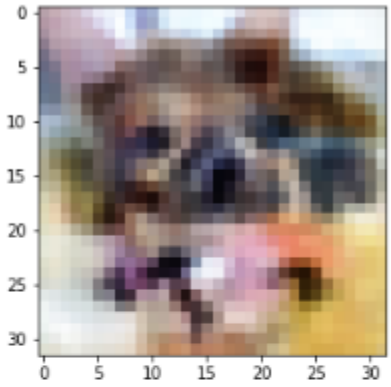


In the image, it is clear that the network input has only blurred scenes of the cat and the visible majority of the photo features a dog.

Outputs:

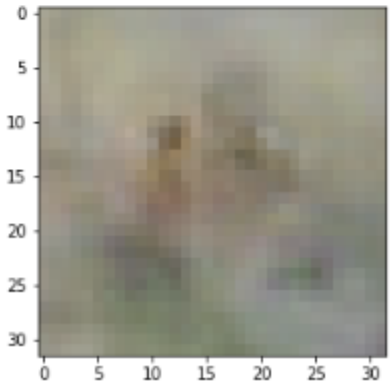
```
plt.imshow(predictions[0][512])
```

```
<matplotlib.image.AxesImage at 0x7fd391572100>
```




```
plt.imshow(predictions[1][512])
```

```
<matplotlib.image.AxesImage at 0x7fd3914bd8b0>
```



As we said earlier, the network was able to reconstruct and decode the image of the dog well, but due to the lack of features of the cat, it has output a relatively unclear image of the cat.

Favorable outputs are provided compared to the input, and this model was able to create outputs close to the labels to a good extent.