

# 1 Feature Encoding

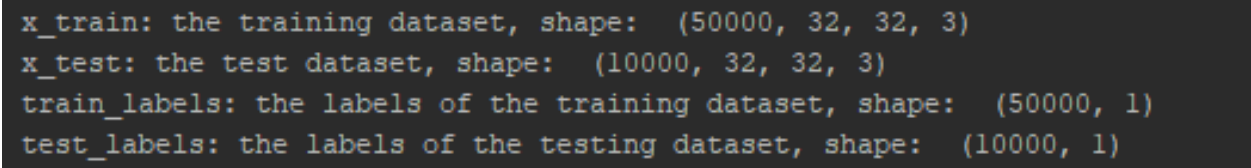
## 1.1 Data Preprocessing

We loaded the CIFAR10 dataset and converted the pixel values from [0,255] to [0,1] float32 numbers.

---

```
#  
# Dataset Load  
#  
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()  
  
#  
# Normalize pixel values to be between 0 and 1  
#  
train_images, test_images = train_images / 255.0, test_images / 255.0
```

---



```
x_train: the training dataset, shape: (50000, 32, 32, 3)  
x_test: the test dataset, shape: (10000, 32, 32, 3)  
train_labels: the labels of the training dataset, shape: (50000, 1)  
test_labels: the labels of the testing dataset, shape: (10000, 1)
```

Figure 1: Details about the data arrays

## 1.2 Building the Autoencoder

We used the convolutional autoencoder architecture. According to this, first we are applying a convolution layer, then we apply max pooling layer to summarize the average presence of a feature and the most activated presence of a feature respectively. After building the encoder, we provide the latent layer for our architecture using 10 neurons. To scale up the image, we use dense layers, and at the end we reshape our image feature. The architecture of the model can be seen in [Figure 2](#).

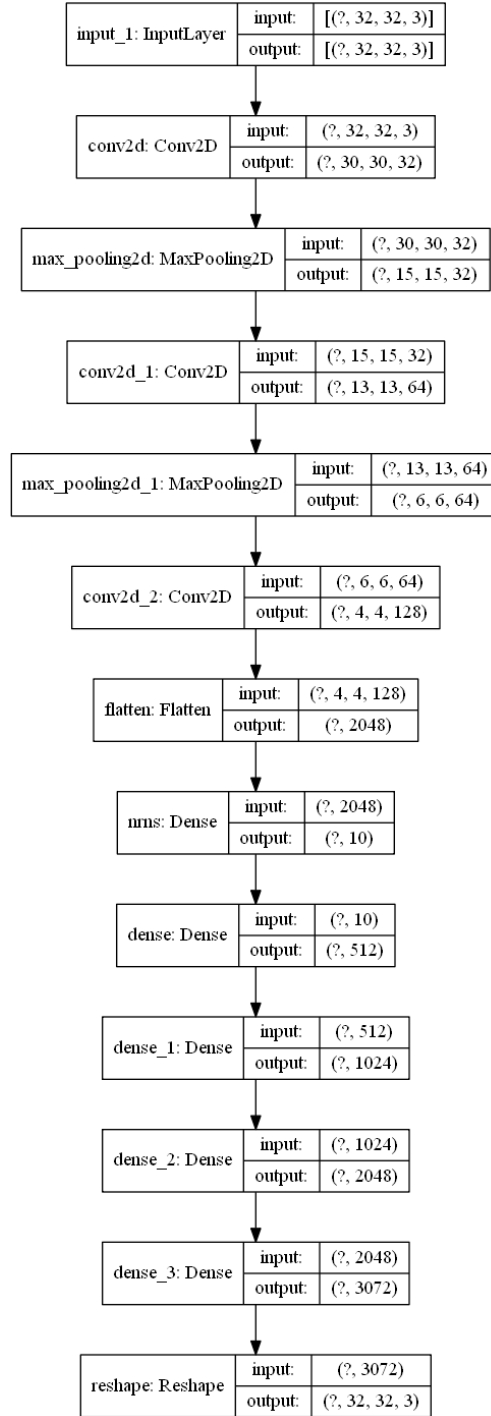


Figure 2: Convolutional Architecture for the Autoencoder

---

```

#
# Architecture
#
# Encoder

```

```

inp = Input((32, 32,3))
e = Conv2D(32, (3, 3),activation='relu')(inp)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(64, (3, 3),activation='relu')(e)
e = MaxPooling2D((2, 2))(e)
e = Conv2D(128, (3, 3),activation='relu')(e)
l = Flatten()(e)
# Bottleneck layer
bl = Dense(10, name='nrns')(l)
# Decoder
d = Dense(512, activation='relu')(bl)
d = Dense(1024, activation='relu')(d)
d = Dense(2048, activation='relu')(d)
d = Dense(3072, activation='sigmoid')(d)
decoded = Reshape((32,32,3))(d)

```

---

### 1.3 Training the Autoencoder

The result plots are from the training of the encoder. We had some issue with over fitting, we tried to solve it by changing activation layers and optimizer.

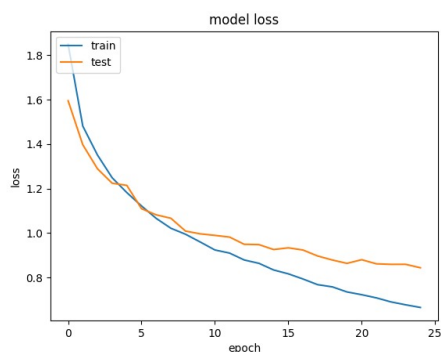


Figure 3: Loss performance with epoch 25

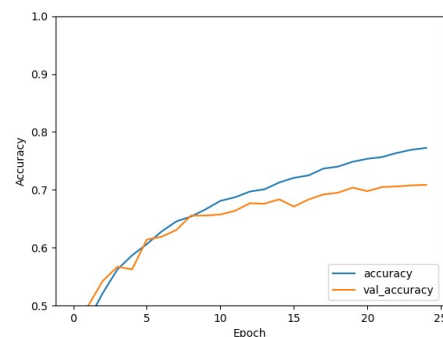


Figure 4: Accuracy performance with epoch 25

## 2 Sanity Check

For sanity check we used one encoder model and one model with encoder and decoder combined. Decoder part is basically is the part of full autoencoder but until the bottleneck layer. We also used numpy, pandas and scikit-learn for this part.

## 2.1 Visualizing Reconstruction Results

The main problem we had with this part was the reconstructing from 10 neurons. There is still a lot of loss with results but Figure 5 shows the best results we were able to get.

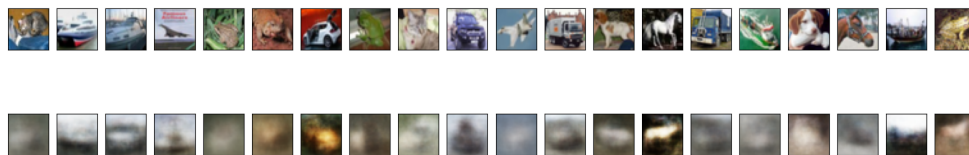


Figure 5: Reconstruct results for the first 10 images of the test dataset

## 2.2 Distribution Analysis

We had quite different results at the beginning. The beginning results were not distributed at all but we think that we have better results with the current model.

See Figure 6 for the distribution analysis.

## 2.3 Projecting Results

The result look different than the one which is provided to us. We think that there might be a problem with overfitting or the model. On the other hand, we get different results with different optimizer and activation methods but this result looked one of the best fitting considering overall results we got.

See Figure 7 for the UMAP projection.

## 3 Data Querying

We used our encoder to query for similar images with following distance measures: Euclidean distance (Figure 8), Manhattan distance (Figure 9), and Cosine distance (10). According to our results, the best performing distance measure is Euclidean > Manhattan > Cosine.

---

```
#
# Query images.
#

#
# Data prep for queries.
#
print("Calculating distances...")
t_images = test_images[:20]
encoded_test_images_20 = feat_extractor.predict(t_images)
```

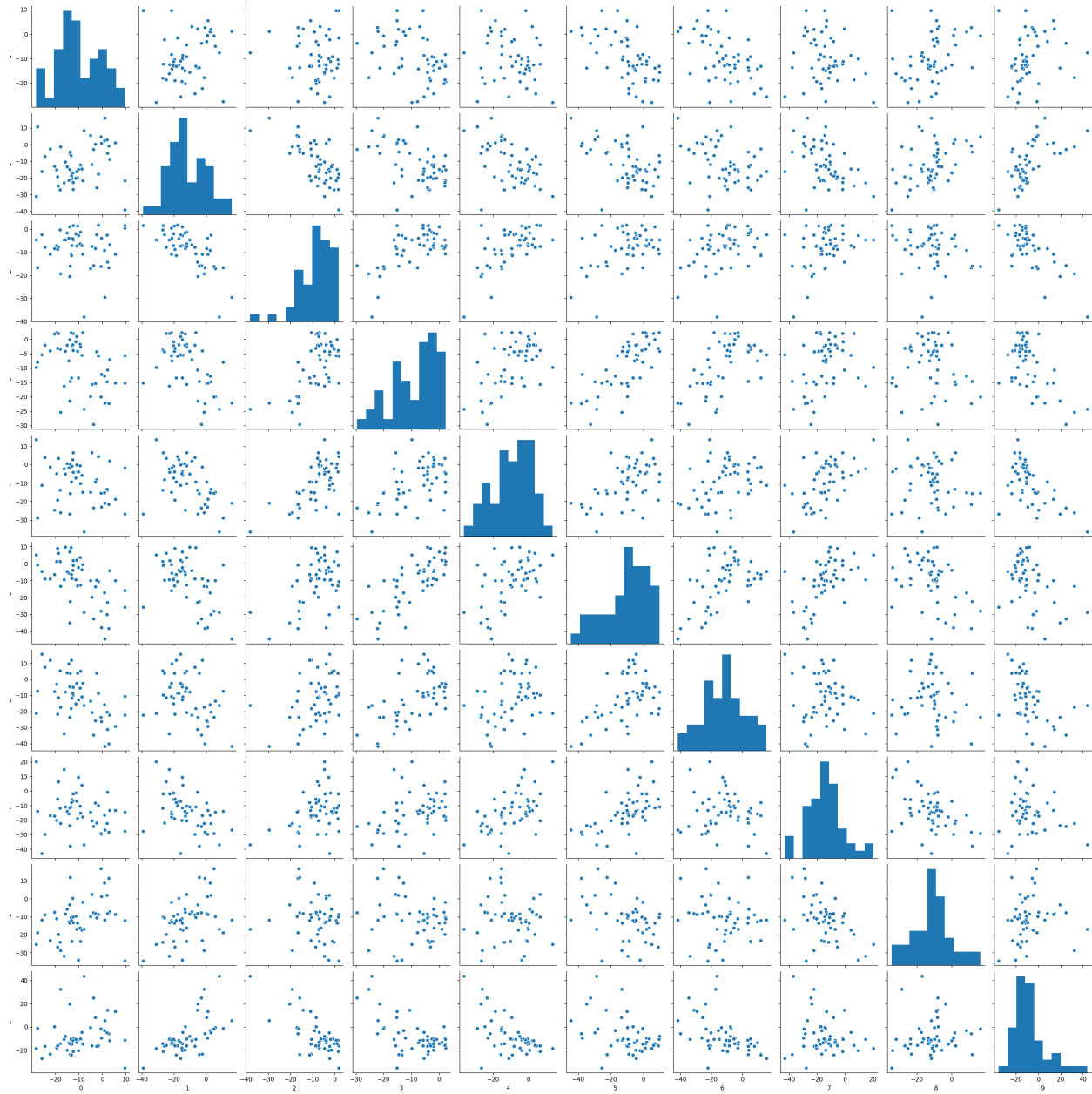


Figure 6: Analysis results of 50 random images from the test dataset

```

encoded_test_images = feat_extractor.predict(test_images[:500])
encoded_train_images = feat_extractor.predict(train_images[:500])
encoded_all_images = np.concatenate((encoded_train_images, encoded_test_images))

encoded_test_images_20 = umap.UMAP().fit_transform(encoded_test_images_20)
encoded_all_images = umap.UMAP().fit_transform(encoded_all_images)

encoded_test_images_20 = pd.DataFrame(encoded_test_images_20)
encoded_all_images = pd.DataFrame(encoded_all_images)

```

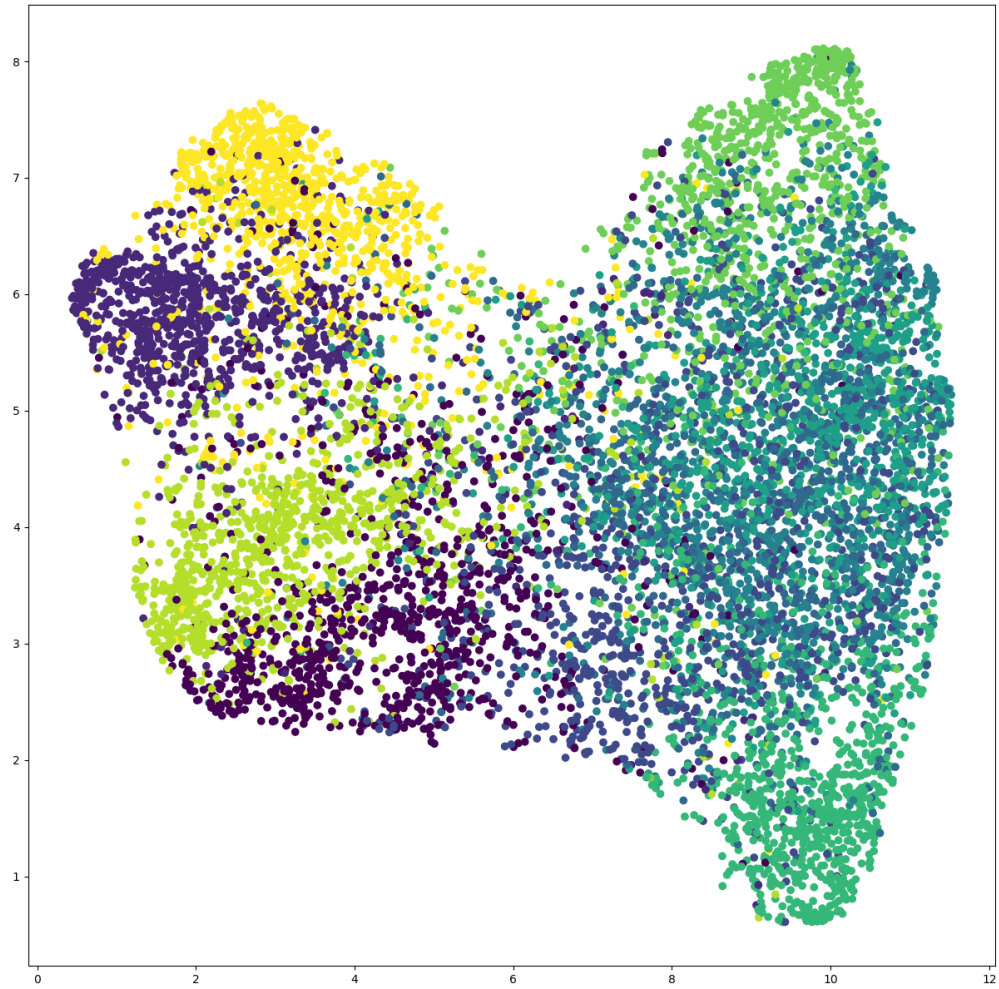


Figure 7: UMAP projection of the latent values of the test dataset

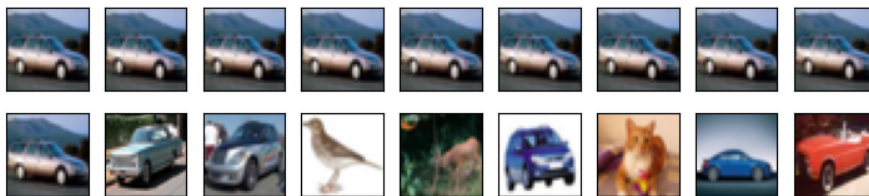


Figure 8: Ten most similar images with Euclidean distance

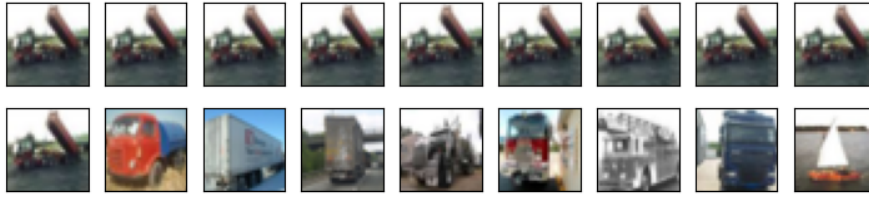


Figure 9: Ten most similar images with Manhattan distance

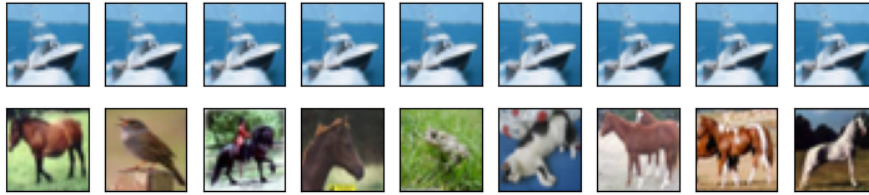


Figure 10: Ten most similar images with Cosine distance

```
original_all_images = np.append(train_images, test_images, axis=0)

distances = pd.DataFrame(columns=['original_id', 'compared_id', 'distance'])

plt.figure(figsize=(20,4))
which_row = 0
for i in range(0,9):
    x1 = encoded_all_images.loc[i, 0]
    y1 = encoded_all_images.loc[i, 1]

    for k in range(0, len(encoded_all_images)):
        x2 = encoded_all_images.loc[k, 0]
        y2 = encoded_all_images.loc[k, 1]

        dist = calc_euc_distance(x1, y1, x2, y2)
        m_dist = calc_manhattan_distance(x1, y1, x2, y2)
        cos_dist = calc_cosine_distance(x1, y1, x2, y2)
        distances.loc[which_row, ('distance')] = dist
        distances.loc[which_row, ('mdistance')] = m_dist
        distances.loc[which_row, ('cdistance')] = cos_dist
        distances.loc[which_row, ('compared_id')] = k
        distances.loc[which_row, ('original_id')] = i
        which_row += 1

n=10
```

```

plt.figure(figsize=(10,2))
for k in range(0,9):
    closest_20_euc = distances.loc[distances['original_id'] == k]
    closest_20_euc = closest_20_euc.apply(pd.to_numeric, errors='coerce')
    closest_20_euc = closest_20_euc.nsmallest(20, ('distance'))
    closest_20_euc = closest_20_euc['compared_id']
    closest_20_euc = closest_20_euc.apply(pd.to_numeric, errors='coerce')
    for z in range(0,9):
        ax = plt.subplot(2, n, z + 1)
        plt.imshow(original_all_images[k].reshape(32, 32, 3))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        ax = plt.subplot(2, n, z + 1 + n)
        getpic = closest_20_euc.iloc[z]
        plt.imshow(original_all_images[getpic].reshape(32, 32, 3))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.savefig('pics/euc'+str(k)+'.png')

for k in range(0,9):
    closest_20_c = distances.loc[distances['original_id'] == k]
    closest_20_c = closest_20_c.apply(pd.to_numeric, errors='coerce')
    closest_20_c = closest_20_c.nsmallest(20, ('distance'))
    closest_20_c = closest_20_c['compared_id']
    closest_20_c = closest_20_c.apply(pd.to_numeric, errors='coerce')
    for z in range(0,9):
        ax = plt.subplot(2, n, z + 1)
        plt.imshow(original_all_images[k].reshape(32, 32, 3))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
        ax = plt.subplot(2, n, z + 1 + n)
        getpic = closest_20_c.iloc[z]
        plt.imshow(original_all_images[getpic].reshape(32, 32, 3))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
    plt.savefig('pics/cosine'+str(k)+'.png')

for k in range(0,9):
    closest_20_m = distances.loc[distances['original_id'] == k]
    closest_20_m = closest_20_m.apply(pd.to_numeric, errors='coerce')
    closest_20_m = closest_20_m.nsmallest(20, ('distance'))
    closest_20_m = closest_20_m['compared_id']
    closest_20_m = closest_20_m.apply(pd.to_numeric, errors='coerce')
    for z in range(0,9):
        # Original images
        ax = plt.subplot(2, n, z + 1)

```



```
plt.imshow(original_all_images[k].reshape(32, 32, 3))
# plt.gray()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

# Display reconstructed images
ax = plt.subplot(2, n, z + 1 + n)
getpic = closest_20_m.iloc[z]
plt.imshow(original_all_images[getpic].reshape(32, 32, 3))
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.savefig('pics/manhattan'+str(k)+''.png')
```

---

## 4 Bonus

### 4.1 Pre-processing

We added noise to training set, and the noisy image can be seen in Figure 11. After that we trained our autoencoder with the noisy dataset. At the end of the progress, it learned denoising from images that can be seen in Figure 12.

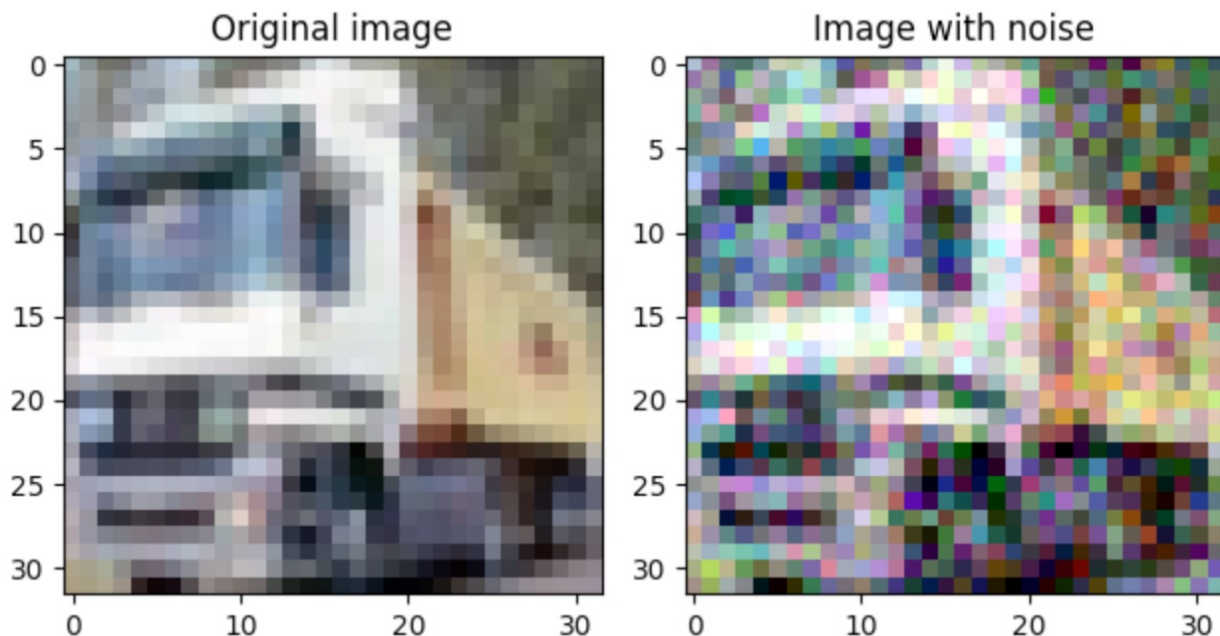


Figure 11: Image result with the added noise

```

def add_noise_and_clip_data(data):
    noise = np.random.normal(loc=0.0, scale=0.1, size=data.shape)
    data = data + noise
    data = np.clip(data, 0., 1.)
    return data

train_images_noisy = add_noise_and_clip_data(train_images)
test_images_noisy = add_noise_and_clip_data(test_images)

history = model.fit(train_images_noisy, train_images,
                    epochs=5, callbacks=callbacks_list, validation_data=(test_images_noisy, test_images))

#
#Denoise results
#
decoded_imgs = model.predict(test_images_noisy[:20])
plt.figure(figsize=(10,4))
for i in range(0,10):
    ax = plt.subplot(2, 10, i + 1)
    plt.imshow(test_images_noisy[i].reshape(32, 32, 3))
    # plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax = plt.subplot(2, 10, i + 1 + 10)
    plt.imshow(decoded_imgs[i].reshape(32, 32, 3))
    # plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.savefig('pics/denoise.png')
plt.show()

```

---

## 4.2 Retraining and Querying Modified Data

After the denoising with trained model which used original data from Cifar10, we continued the training with noisy data. As you can see from Figure 13, we had a little improvement. However, we still think there is a lot we can improve regarding the overall model and results.

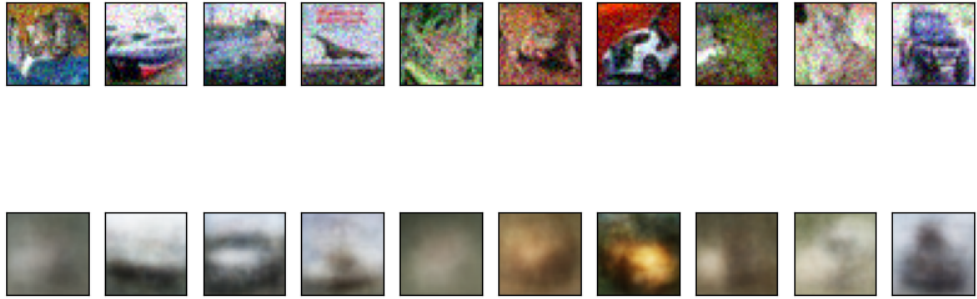


Figure 12: Reconstruct results with noisy dataset

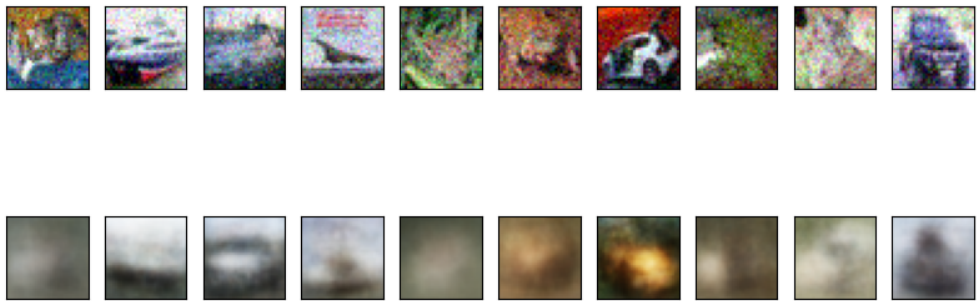


Figure 13: Reconstruct results with retrained model