

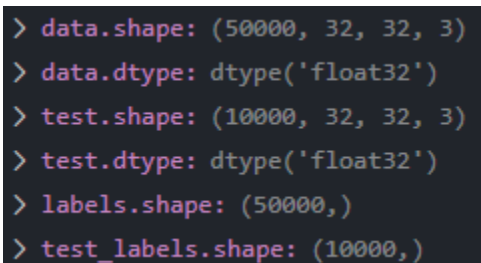
3 Feature Encoding

3.1 Dataset Preparation

First, the data is downloaded from the website of the University of Toronto. To read the data, `deserialize()` is used. The function `reshape()` brings the shape read data into the correct form and `preprocess()`, which is used to import the images, eventually returns the image array with the dtype `float32`.

```
1 def deserialize(path):
2     with open(path, 'rb') as file:
3         return pickle.load(file, encoding='bytes')
4
5 def reshape(matrix):
6     matrix = matrix.reshape((10000, 3, 1024))
7     matrix = matrix.reshape((10000, 3, 32, 32))
8     return np.moveaxis(matrix, 1, -1)
9
10 def preprocess():
11     data = reshape(deserialize("cifar-10-batches-py/data_batch_1")[b'data'])
12     for i in range(2,6):
13         batch = reshape(deserialize("cifar-10-batches-py/data_batch_" + str(i))[b'data'])
14         data = np.append(data, batch, axis=0)
15     return np.float32(data / 255)
16
17 def load_labels():
18     labels = deserialize("cifar-10-batches-py/data_batch_1")[b'labels']
19     for i in range(2,6):
20         batch = deserialize("cifar-10-batches-py/data_batch_" + str(i))[b'labels']
21         labels = np.append(labels, batch, axis=0)
22     test_labels = deserialize("cifar-10-batches-py/test_batch")[b'labels']
23     return labels, test_labels
24
25 def load_test_batch():
26     return np.float32(reshape(deserialize("cifar-10-batches-py/test_batch")[b'data']) / 255)
```

Listing 1: Load and Process Dataset



```
> data.shape: (50000, 32, 32, 3)
> data.dtype: dtype('float32')
> test.shape: (10000, 32, 32, 3)
> test.dtype: dtype('float32')
> labels.shape: (50000,)
> test_labels.shape: (10000,)
```

Figure 1: Data shape and type. The range is not displayed as it is a hassle to get it working in the debugger, though manual checking of the values shows a range of `[0,1]`

3.2 Building the Auto-Encoder

We started out with a very basic auto-encoder which featured only two convolutional layers in both the encoder and decoder and a linear layer in each. The reconstruction when trained on just one picture was pretty good, but when training on 10 or more images, the reconstructed images wouldn't be distinguishable of each other as they were just grey blobs. After adding 6 more convolutional layers to each stage with around 10000 out-features in the encoding linear layer, the resulting images were even more grey, as there were just too many features. Reducing the out-features significantly to around 500 improved the outcome by a large margin, but 8 layers seemed to be too many, so we reduced the convolutional layers in the encoding stage to four and settled on an out-feature count of 405.

The decoding stage consisted first of just two convolutional layers with a linear layer at the end to get the shape of [3072, 1] and eventually the initial shape of [1, 3, 32, 32] back. We settled on four convolutional layers with a linear layer at the beginning (such that we can use a square kernel without problems) and at the end, where we have 9196 out-features to reduce the amount of blocks, to get the same output tensor shape as the input.

We chose Leaky ReLU as activation function because it can handle negative values, otherwise there would be lost information. Sigmoid was used in the last linear layer of the decoder to prevent any clipping of the values.

```
1 class Model(nn.Module):
2     def __init__(self, width, height, channels):
3         super(Model, self).__init__()
4         self.width = width
5         self.height = height
6         self.channels = channels
7
8         # encode stage
9         self.convolutional_0 = nn.Conv2d(3, 13, 1)
10        self.convolutional_1 = nn.Conv2d(13, 26, 2, 2)
11        self.convolutional_2 = nn.Conv2d(26, 38, 2, 2)
12        self.convolutional_3 = nn.Conv2d(38, 45, 2, 3)
13        self.linear_0 = nn.Linear(405, 10)
14
15        # decode stage
16        self.linear_1 = nn.Linear(10, 16)
17        self.deconvolutional_0 = nn.ConvTranspose2d(1, 5, 2, 1)
18        self.deconvolutional_1 = nn.ConvTranspose2d(5, 9, 2, 2)
19        self.deconvolutional_2 = nn.ConvTranspose2d(9, 14, 3, 2)
20        self.deconvolutional_3 = nn.ConvTranspose2d(14, 19, 2, 1)
21        self.linear_2 = nn.Linear(9196, 3072)
22
23    def encoder(self, x):
24        x = F.leaky_relu(self.convolutional_0(x)) # Shape: [1, 13, 32, 32]
25        x = F.leaky_relu(self.convolutional_1(x)) # Shape: [1, 26, 16, 16]
26        x = F.leaky_relu(self.convolutional_2(x)) # Shape: [1, 38, 8, 8]
27        x = F.leaky_relu(self.convolutional_3(x)) # Shape: [1, 45, 3, 3]
28        x = torch.flatten(x) # Shape: [405]
29        x = F.leaky_relu(self.linear_0(x)) # Shape: [10]
30        return x
31
32    def decoder(self, x):
33        x = F.leaky_relu(self.linear_1(x)) # Shape: [16]
34        x = torch.reshape(x, (1, 1, 4, 4)) # Shape: [1, 1, 4, 4]
35        x = F.leaky_relu(self.deconvolutional_0(x)) # Shape: [1, 5, 5, 5]
36        x = F.leaky_relu(self.deconvolutional_1(x)) # Shape: [1, 9, 10, 10]
37        x = F.leaky_relu(self.deconvolutional_2(x)) # Shape: [1, 14, 21, 21]
38        x = F.leaky_relu(self.deconvolutional_3(x)) # Shape: [1, 19, 22, 22]
39        x = torch.flatten(x) # Shape: [9196]
40        x = torch.sigmoid(self.linear_2(x)) # Shape: [3072]
41        x = torch.reshape(x, (1, 3, 32, 32)) # Shape: [1, 3, 32, 32]
42        return x
43
44    def forward(self, x):
45        x = self.encoder(x)
46        x = self.decoder(x)
47        return x
```

Listing 2: Auto-Encoder

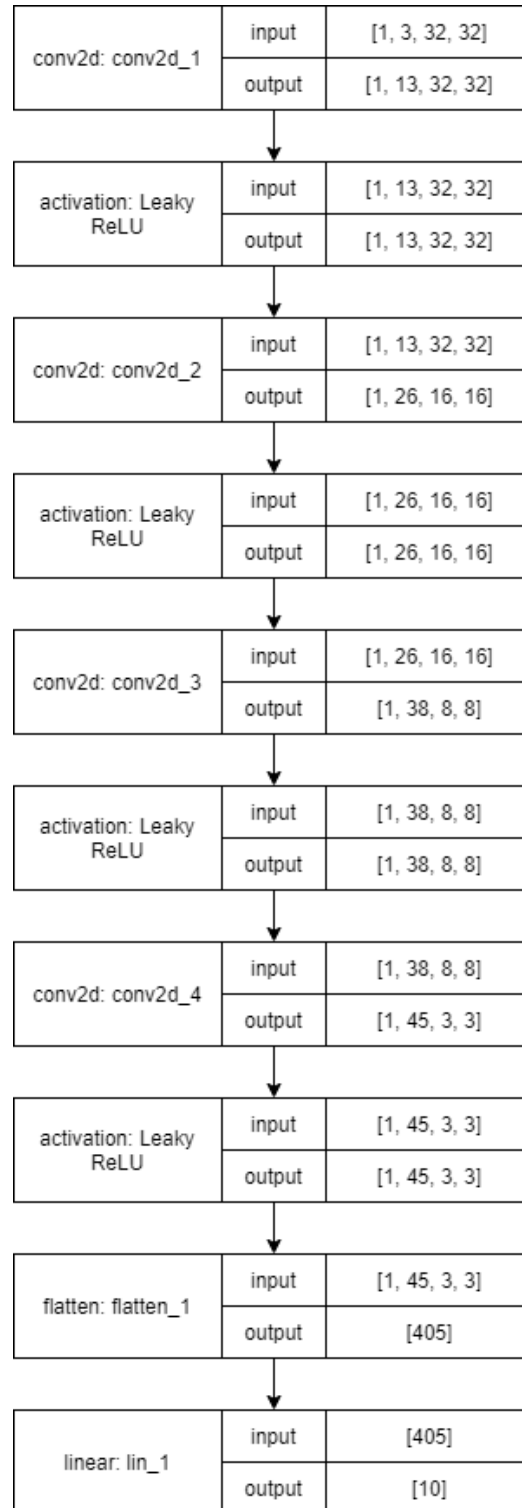


Figure 2: Encoder

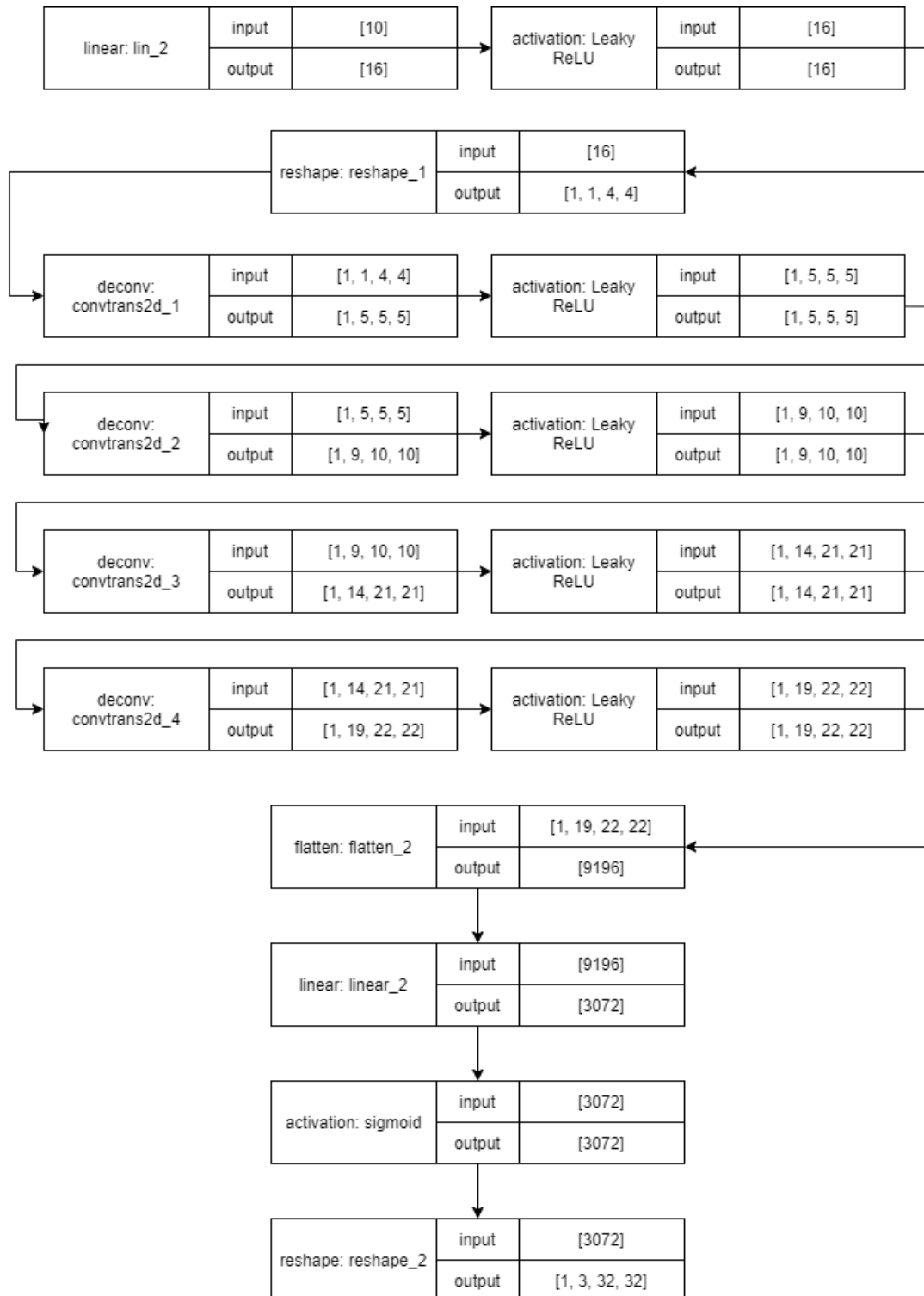


Figure 3: Decoder

3.3 Training the Auto-Encoder

```
1 def training(model, training_data, test_data):
2     optimizer = optim.Adam(model.parameters(), lr=learning_rate)
3     if loss_function == "mse":
4         criterion = nn.MSELoss()
5     elif loss_function == "bce":
6         criterion = nn.BCELoss()
7
8     ring_buffer = RingBuffer(1000)
9     iterations = 0
10    running_loss = 0
11    history_loss = []
12    epoch = 0
13    current_index = 0
14    model_note = ""
15    model_name = "model"
16    model_path = f"model/{model_name}"
17
18    try:
19        checkpoint = torch.load(model_path+".hdf5")
20        epoch = checkpoint["epoch"]
21        current_index = checkpoint["current_index"]
22        iterations = checkpoint["iterations"]
23        model.load_state_dict(checkpoint["model_state_dict"])
24        optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
25        running_loss = checkpoint["loss"]
26        history_loss = checkpoint["history_loss"]
27        print("Using previous model")
28    except FileNotFoundError:
29        print("no model found")
30
31    start_time = datetime.datetime.now()
32    progress = ""
33    try:
34        while True:
35            for idx, elem in enumerate(training_data[current_index:], start=current_index):
36                if current_index > len(training_data):
37                    break
38                iterations += 1
39                current_index = idx+1
40                time = datetime.datetime.now()
41                tensor = torch.from_numpy(elem).to(device)
42                target = tensor.permute(2, 0, 1).unsqueeze(0)
43                optimizer.zero_grad()
44                output = model(target)
45                loss = criterion(output, target)
46                # Track loss
47                running_loss += loss.item()
48                loss_current = running_loss / iterations
49                history_loss.append(loss_current)
50                if iterations % 10000 == 0:
51                    show_image(model, training_data[0:10], epoch, running_loss/iterations, len(
training_data), model_name, save=True, test=False)
52                    loss.backward()
53                    optimizer.step()
54                    time = datetime.datetime.now() - time
55                    ring_buffer.add(time.microseconds)
56                    elapsed = datetime.datetime.now() - start_time
57                    elapsed = elapsed - datetime.timedelta(microseconds=elapsed.microseconds)
58                    remaining = datetime.timedelta(microseconds=ring_buffer.avg() * (len(training_data)
) - iterations%len(training_data)))
59                    remaining = remaining - datetime.timedelta(microseconds=remaining.microseconds)
60                    if current_index % 50 == 0:
61                        progress = "\rTraining on n={0}: loss: {6:.6f} | iterations: {7: >8} | epoch:
```

```
62     {5: >3} | {1: >6}/{0} | {2: >4}ms/iteration | elapsed: {3:>8} | remaining: {4:>8}" \
63         .format(len(training_data),
64                 current_index,
65                 int(ring_buffer.avg()/1000),
66                 str(elapsed),
67                 str(remaining),
68                 epoch,
69                 running_loss/iterations,
70                 iterations)
71     print(progress, end="")
72     print("")
73     current_index = 0
74     epoch += 1
75     save_model(model_path, epoch, iterations, model, optimizer, running_loss, history_loss
76               , current_index)
77     show_loss(history_loss, len(training_data), epoch, save=True)
78     show_image(model, test_data[0:10], epoch, running_loss/iterations, len(training_data),
79               model_name, save=True, test=True)
80     except KeyboardInterrupt:
81         print("\nKeyboard interrupt")
82         show_image(model, test_data[0:10], epoch, running_loss/iterations, len(training_data),
83               model_name, save=True, test=True)
84         show_image(model, training_data[0:10], epoch, running_loss/iterations, len(training_data),
85               model_name, save=True, test=False)
86         save_model(model_path, epoch, iterations, model, optimizer, running_loss, history_loss,
87               current_index)
88         show_loss(history_loss, len(training_data), epoch, save=True)
```

Listing 3: Training of the auto-encoder

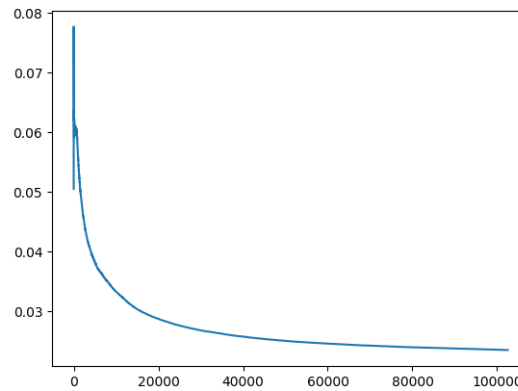


Figure 4: Loss, trained on 50000 images for 2 epochs

The data set that was used for training increased from 10 to 100 to 1000 to the full 50000, each step occurring when we were satisfied with the results. We used adam as optimizer and first used binary cross-entropy as our loss function, though eventually we settled for the mean squared error loss function because it allowed for a very fast convergence and therefore less training time. The auto-encoder has been trained on the full data set of 50000 images over two epochs. After not even one epoch, the loss already started to converge as seen in Figure 4.

4 Sanity Check

4.1 Visualizing Reconstruction Results

We didn't check the reconstruction result until the auto-encoder was considered good enough and were satisfied with the reconstruction result from Fig 5 from the get-go. The reconstruction was created by taking the first 10 images of the test data set which were recreated by the trained model. The top row shows the original images and the bottom row the reconstructed images after training the auto-encoder for 14 epochs.

```

1 def show_image(model, training_data, epoch, loss, n, model_name, save=False, test=False):
2     model.eval()
3     fig, axes = plt.subplots(2, len(training_data))
4     for i in range(0, len(training_data)):
5         original = training_data[i]
6         elem = training_data[i]
7         tensor = torch.from_numpy(elem).to(device)
8         trained = torch.tensor(elem, requires_grad=True).to(device)
9         trained = tensor.permute(2, 0, 1).unsqueeze(0)
10        trained = model(trained)
11        trained = np.moveaxis(trained.detach().cpu().numpy(), 0, 2)
12        axes[0, i].axis("off")
13        axes[1, i].axis("off")
14        axes[0, i].imshow(original)
15        axes[1, i].imshow(trained)
16    plt.tight_layout(w_pad=1, h_pad=10)
17    fig.set_size_inches(20, 5)
18    if save:
19        time = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
20        directory = "result/{0}".format(model_name)
21        if not os.path.exists(directory):
22            os.makedirs(directory)
23        if test:
24            plt.savefig("{0}/result_test_ep{1}_n{2}.png".format(directory, epoch, n))
25        else:
26            plt.savefig("{4}/{0}_result_ep{1}_l{2:.6f}_n{3}.png".format(time, epoch, loss, n,
27            directory))
28    else:
29        plt.show()
30    plt.clf()
31    plt.close()

```

Listing 4: Visualizing reconstruction result

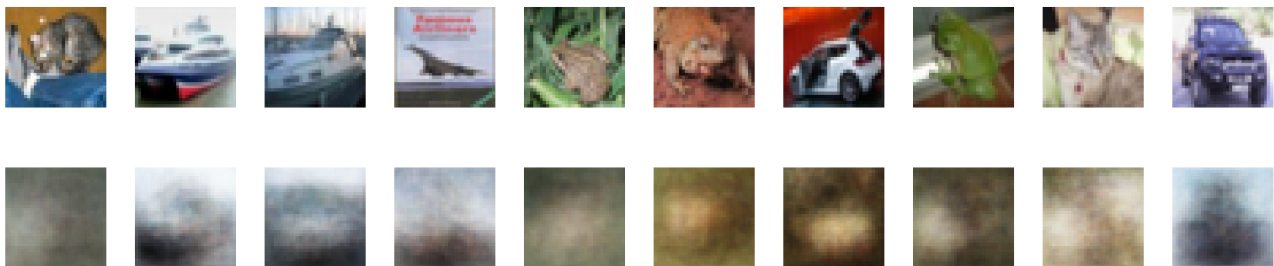


Figure 5: Reconstruction result

4.2 Distribution Analysis

50 random images from the test data set were taken to create the SPLOM. First a `DataFrame` needs to be created with `pandas` and the 10 features of the image are used as columns. With `seaborn` the actual pairplot is created. While

the SPLOM (Fig 6) shows some clustering, it is largely distributed, with which we are satisfied.

```
1 def generate_pairplot(auto_encoder, data):
2     imgs = []
3     encoder = auto_encoder.encoder
4     random_numbers = gen_random_numbers(50, len(data))
5     for idx, i in enumerate(random_numbers):
6         tensor = torch.from_numpy(data[i]).to(torch.device('cuda'))
7         target = tensor.permute(2, 0, 1).unsqueeze(0)
8         img = encoder(target).detach().cpu().numpy()
9         imgs.append(img)
10        print(f"\rProcessing {idx+1:>2}/50 images for pairplot.", end="")
11    print("\nGenerating pairplot...")
12    dataframe = pd.DataFrame(data=np.array(imgs), columns=["feature_0", "feature_1", "feature_2",
13    "feature_3", "feature_4", "feature_5", "feature_6", "feature_7", "feature_8", "feature_9"])
14    g = sns.pairplot(dataframe)
15    plt.tight_layout()
16    plt.savefig("splom_{0}.png".format(model_name))
17    print("Generated pairplot.")
18    plt.clf()
```

Listing 5: SPLOM

4.3 Projecting Results

The UMAP (Fig 7) shows a scatter plot of the projected latent values of the test data set where the colors represent the class labels. For this the library umap was used to project the latent values onto the 2D space. It is apparent that the different classes are clustered together.

```
1 def generate_scatterplot(auto_encoder, data, labels):
2     imgs = []
3     encoder = auto_encoder.encoder
4     reducer = umap.UMAP()
5     for idx, i in enumerate(data):
6         tensor = torch.from_numpy(i).to(torch.device('cuda'))
7         target = tensor.permute(2, 0, 1).unsqueeze(0)
8         img = encoder(target).detach().cpu().numpy()
9         imgs.append(img)
10        print(f"\rProcessing {idx+1:>{len(str(len(data))}}/{len(data)} images for scatterplot.",
11    end="")
12    print("\nGenerating scatterplot...")
13    embedding = reducer.fit_transform(imgs)
14    print(embedding.shape)
15    fig = plt.figure()
16    plt.scatter(
17        embedding[:, 0],
18        embedding[:, 1],
19        c=[sns.color_palette()[x] for x in labels]
20    )
21    fig.set_size_inches(15, 15)
22    plt.tight_layout()
23    plt.savefig("umap_{0}.png".format(model_name))
24    print("Generated scatterplot.")
25    plt.clf()
```

Listing 6: UMAP

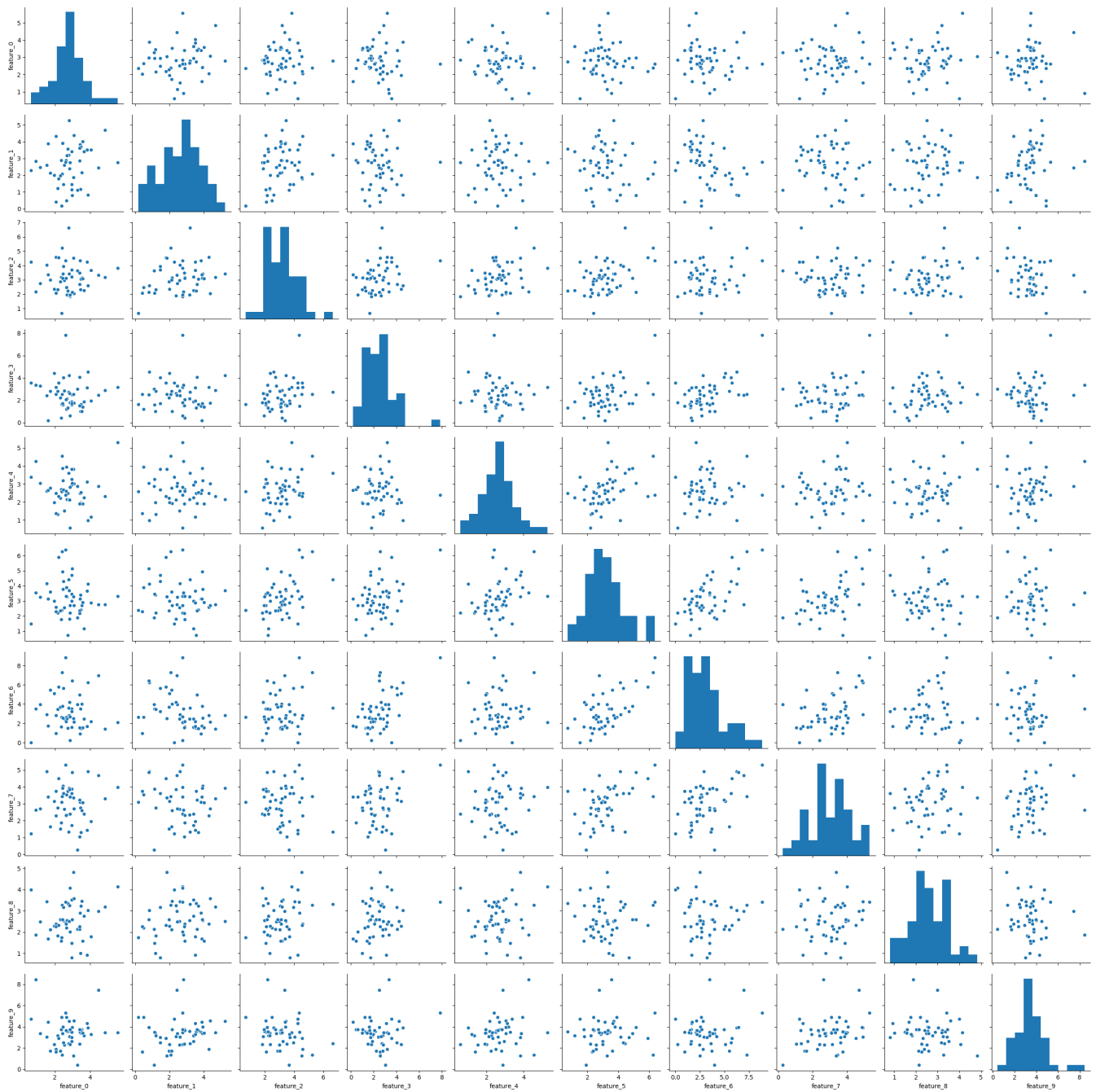


Figure 6: SPLOM of the latent values of the test data set

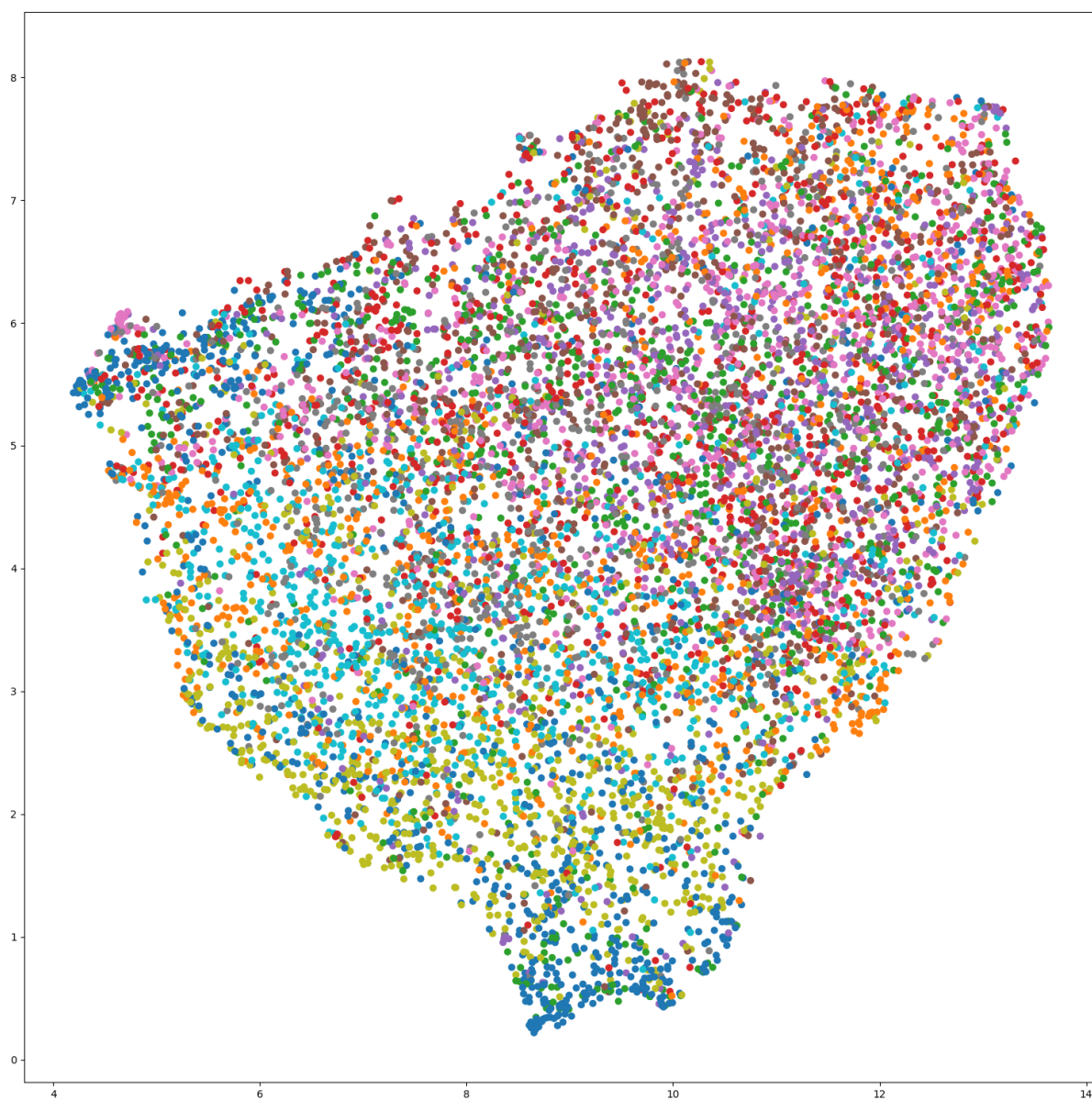


Figure 7: UMAP-projected latent values of the test data set, colored by class label

5 Data Querying

```
1 def minkowski_like_dist(fstp, sndp, p):
2     ary = sum(abs(fstp[i] - sndp[i])**p for i in range(0, len(fstp)))
3     result = ary**(1/p)
4     return result
5
6 def cosine_dist(fstp, sndp):
7     numerator = sum(fstp[i] * sndp[i] for i in range(0, len(fstp)))
8     denom_fst = sum(fstp[i] ** 2 for i in range(0, len(fstp)))
9     denom_snd = sum(sndp[i] ** 2 for i in range(0, len(sndp)))
10    return 1 - (numerator / (math.sqrt(denom_fst) * math.sqrt(denom_snd)))
11
12 def similarity_cosine(query_img_enc, org_imgs_enc, auto_encoder):
13     dist = []
14     for i, org_img in enumerate(org_imgs_enc):
15         dist.append((cosine_dist(query_img_enc, org_img), i))
16     similar_imgs = sorted(dist, key=lambda x: x[0])
17     return similar_imgs[1:11]
18
19 def encode(auto_encoder, data):
20     imgs = []
21     encoder = auto_encoder.encoder
22     for i, data_img in enumerate(data):
23         tensor = torch.from_numpy(data_img).to(device)
24         target = tensor.permute(2, 0, 1).unsqueeze(0)
25         img = encoder(target).detach().cpu().numpy()
26         imgs.append(img)
27         if i+1 % 10 == 0:
28             print(f"\rEncoded {i}/{len(data)}", end="")
29     print("")
30     return imgs
31
32 def similarity_minowski(p, query_img_enc, org_imgs_enc, auto_encoder):
33     dist = []
34     for i, org_img in enumerate(org_imgs_enc):
35         dist.append((minkowski_like_dist(query_img_enc, org_img, p), i))
36     similar_imgs = sorted(dist, key=lambda x: x[0])
37     return similar_imgs[1:11]
38
39 def show_similar_imgs(query_img, similar_imgs, auto_encoder, p, idx):
40     auto_encoder.eval()
41     fig, axes = plt.subplots(2, len(similar_imgs))
42     for i in range(0, len(similar_imgs)):
43         original = query_img
44         index = similar_imgs[i][1]
45         elem = data[index]
46         axes[0, i].axis("off")
47         axes[1, i].axis("off")
48         axes[0, i].imshow(original)
49         axes[1, i].imshow(elem)
50     plt.tight_layout(w_pad=1, h_pad=10)
51     fig.set_size_inches(20, 5)
52     directory = f"querying/{p}"
53     if not os.path.exists(directory):
54         os.makedirs(directory)
55     plt.savefig(f"{directory}/similarity_{p}_{idx}.png")
56     plt.clf()
57     plt.close()
```

Listing 7: Data Querying

We selected 2 different images to query using the Manhattan, Euclidean (both with the Minowski Like algorithm) and the Cosine Distance. The 10 most similar images to the query image are visualized. First all images from the test data set are encoded with the encoder of the auto-encoder. Those encoded images are then used to calculate

their distance to the query image and then sorted in ascending order based on the distance. The closest images to the query image are then visualized.

It is hard to judge which one of the three measures is the best as all perform equally well (or badly).

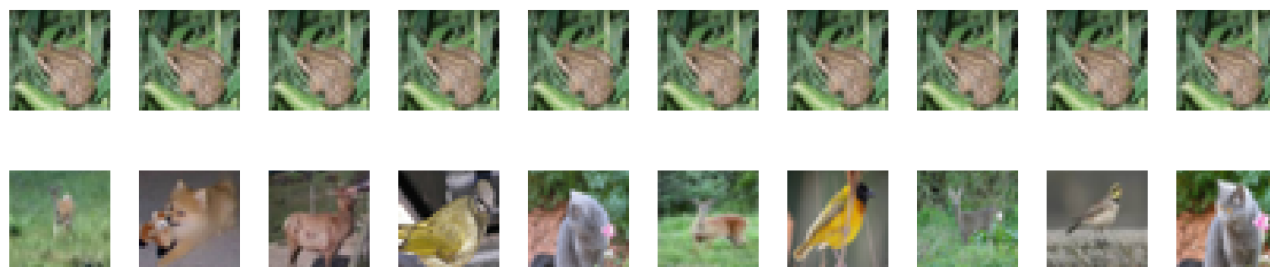


Figure 8: Image 1 using Minkowski Like Distance with $p=1$ (Manhattan Distance)

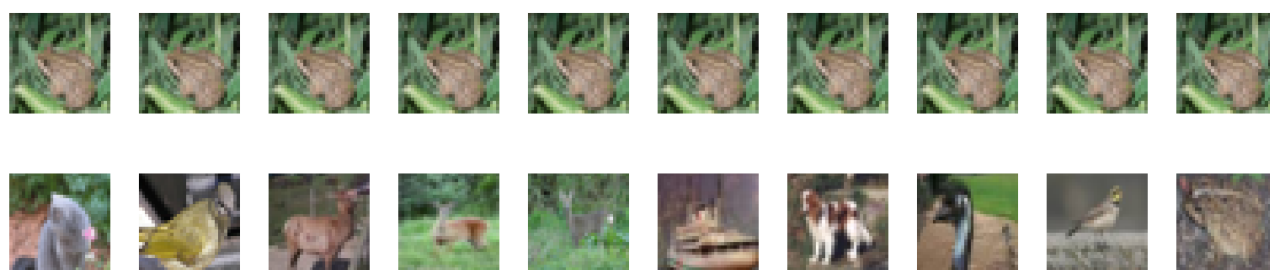


Figure 9: Image 1 using Minkowski Like Distance with $p=2$ (Euclidean Distance)

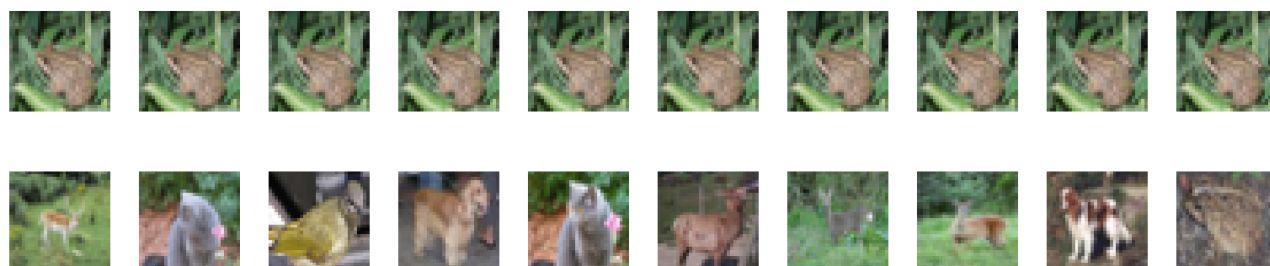


Figure 10: Image 1 using Cosine Distance

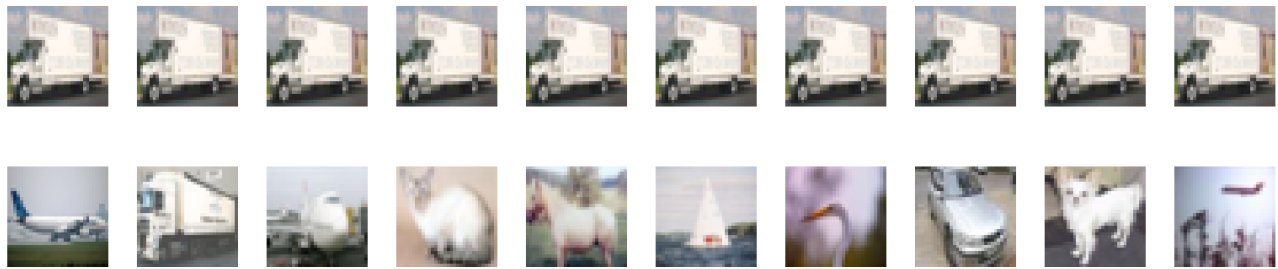


Figure 11: Image 2 using Minkowski Like Distance with $p=1$ (Manhattan Distance)

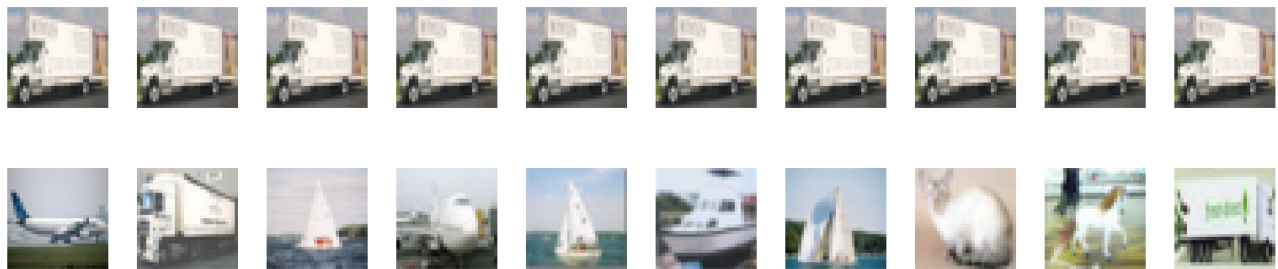


Figure 12: Image 1 using Minkowski Like Distance with $p=2$ (Euclidean Distance)

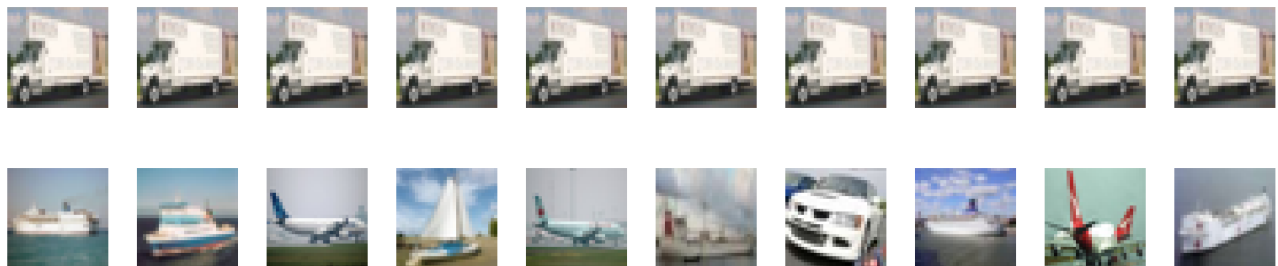


Figure 13: Image 2 using Cosine Distance

6 Bonus: Improving Stability

6.1 Pre-processing

We used the gaussian filter from the `scipy` library which we could just apply on the images to create a blur. For the second function we went with a vertical sinus wave warp to introduce pixels without image data. For this we applied a sinus function on the row column vectors of the images. Figure 14 shows the blurred images using the gaussian filter and Figure 15 shows the warped images using the sinus function.

```
1 def apply_gaussian_blur(images, sigma):
2     result = []
3     for idx, img in enumerate(images):
4         result.append(gaussian_filter(img, sigma=sigma))
5         print(f"\rApplied filter to {idx+1}>{len(str(len(images)))}/{len(images)} images.", end="")
6     print("")
```

```

7     return result
8
9 def apply_vert_wave(images, shift):
10     result = []
11     for idx, img in enumerate(images):
12         rows, cols, channels = img.shape
13         img_output = np.zeros(img.shape, dtype=img.dtype)
14         for i in range(rows):
15             for j in range(cols):
16                 offset_x = int(shift * np.sin(9.0 * np.pi * i / 180))
17                 offset_y = 0
18                 if j+offset_x < cols:
19                     img_output[i,j] = img[i, (j+offset_x)%cols]
20                 else:
21                     img_output[i,j] = 0
22             result.append(img_output)
23     print(f"\rApplied distortion to {idx+1:>{len(str(len(images)))/{len(images)} images.",
24           end="")
25     print("")
26     return result

```

Listing 8: Distortion Functions

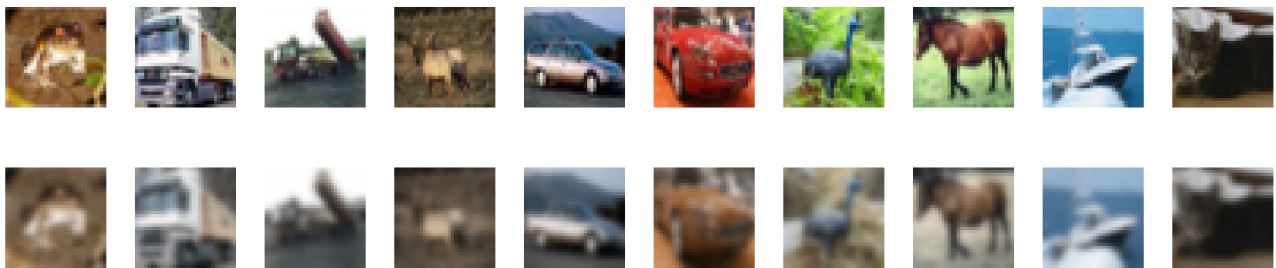


Figure 14: Image distortion by blurring using a gaussian filter



Figure 15: Image distortion by rotating rows and columns according to a sine function

6.2 Re-training

When using the distorted images to train the existing model, the loss didn't change much; it changed marginally which wouldn't be visible on the loss graph and would look similar to Figure 4. We concluded that we therefore didn't understand the task correctly but since we did some training, the results will be presented nonetheless. Figure 16 shows the reconstructed images after the model has been trained for 10 epochs with 1000 images that were distorted with the gaussian filter. Compared to Figure 17, which shows the reconstructed images without any distorted images in the training set, the result is similar if not even better. Figure 18 shows the reconstructed images after the model has been trained for 5 epochs with 100 images that were distorted with the sinus function. It is obvious that the black pixels without image data were dominant, though in some images such as in the third the black pixels have a slight grey-ish "fog", so they are not completely black. Disregarding that, the image overall seems to have the same shape as the original image, so the distortion of the image didn't affect the reconstruction too much. However, this might very well be the case because the model hasn't been trained long enough on the distorted data set.

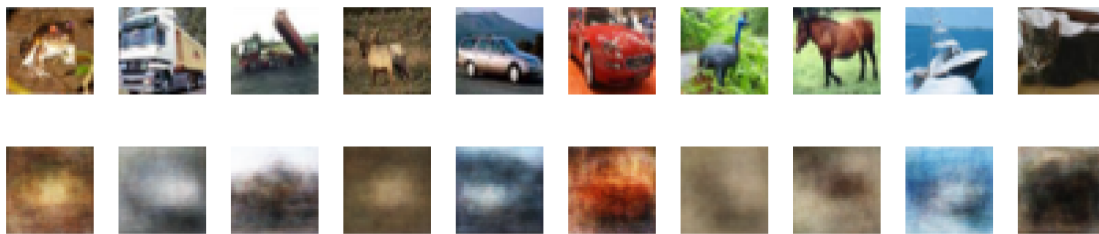


Figure 16: Reconstruction of images of training on images with gaussian filter applied

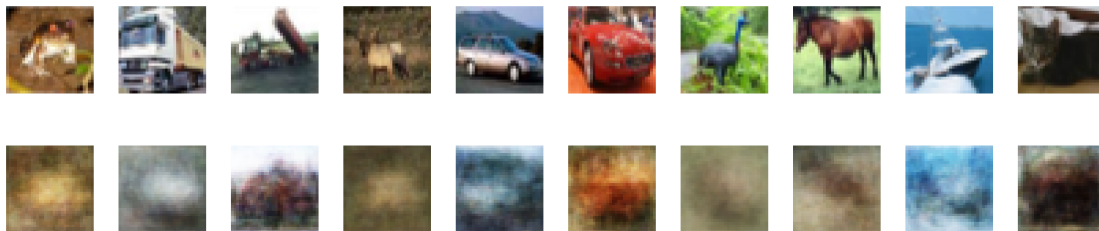


Figure 17: Reconstruction of images of training on images without any distortion

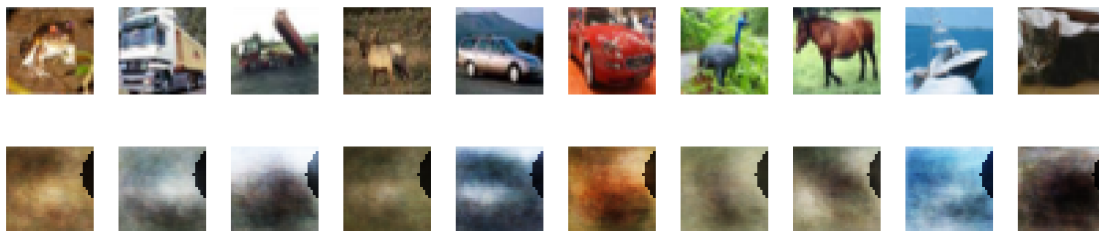


Figure 18: Reconstruction of images of training on images with sinus function applied