Member-only story Featured

# A deep-dive into Autoencoders (AE, VAE, and VQ-VAE) with code

Autoencoders thoroughly explained

Nikolaus Correll · Follow

Published in Toward Humanoids · 18 min read · Feb 17, 2025



Autoencoders are a class of unsupervised neural networks that can represent data in a lower-dimensional space, also known as *latent space*, to learn efficient representations. Applications include compression, denoising, feature extraction, and generative models. Autoencoders are trained by first *encoding* data into a latent space and then *decoding* them back into the original representation, also known as *reconstruction*, while minimizing the difference between the original input and the reconstructed data. In an extension, *Variational Autoencoders* (VAE) learn a probability distribution over the latent space, which allows them to generate entirely new data while sacrificing their ability to perfectly reconstruct existing data. This trade-off is addressed by *Vector Quantized Variational Autoencoders* (DQ-VAE), which represent the latent space with a learned alphabet or *codebook*. This blog post

will iteratively develop implementations for all three using the CIFAR10 image dataset.

*Read this story for free here or subscribe to Medium.com.*

## The Data and Setup

This article is meant for work-along in a Google colab. The code below will import the necessary libraries and load the dataset:

```
optim

nsforms as transforms
ot as plt
onal as F
import DataLoader



"cuda" if torch.cuda.is_available() else "cpu")

10)
.Compose([transforms.ToTensor()])
.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
tils.data.DataLoader(trainset, batch_size=64, shuffle=True)
```

The code uses Torch's DataLoader functionality and torchvision's transforms library. In this case, we are using *transforms* to turn the images into a torch tensor. The DataLoader is described in more detail in this article:

**Gradient Descent and Batch-Processing for Generative Models in PyTorch**

Step-by-step from fundamental concepts to training a basic generative model

medium.com

You can visualize the training data using matplotlib:

```
# Function to display images in a grid
def display_images(images):
    fig, axes = plt.subplots(3, 3, figsize=(4, 4))
    for i, ax in enumerate(axes.flat):
        if i < len(images):
            image = images[i].permute(1, 2, 0).numpy()  # Rearrange dimensions f
            ax.imshow(image)
            ax.axis('off')
    plt.show()

# Get the first 9 images from the train_loader
# Get the first 9 images from the train_loader
data, target = next(iter(train_loader))
images = data[:9]

# Display images using the helper function
display_images(images)
```

Note that the *train_loader* returns a tuple *data, target*, which contain the images and the target class as a number from 0 to 9. We are not using the targets at all in this article, but will train autoencoder variants to learn efficient data representations and generate new images. Running the code above should yield something like this:

Random images from the CIFAR10 dataset. Each image has three channels and 32×32 pixels.

## The Autoencoder

A basic autoencoder consist of two blocks: an encoder and a decoder. After training them together, we can use the encoder and decoder separately to compress and decompress an image, respectively. For images, the encoder usually consists of a series of convolutional layers followed by a fully connected layer. The decoder has the inverse structure, starting with a fully connected layer, followed by a series of convolutional layers.

```
class ConvAE(nn.Module):
    def __init__(self, latent_dim=100):
        super(ConvAE, self).__init__()

        # Encoder: Convolutions to extract features
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),  # (B, 64, 16,
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1), # (B, 128, 8
            nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1), # (B, 256,
            nn.ReLU(),
        )

        # Latent space
        self.fc = nn.Linear(256 * 4 * 4, latent_dim)

        # Decoder: Transposed convolutions for upsampling
        self.decoder_input = nn.Linear(latent_dim, 256 * 4 * 4)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1), #
            nn.ReLU(),
```



```
    def encode(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)  # Flatten
        z = self.fc(x)
        return z

    def decode(self, z):
        x = self.decoder_input(z)
        x = x.view(x.size(0), 256, 4, 4)  # Reshape to feature maps
        x = self.decoder(x)
        return x

    def forward(self, x):
        z = self.encode(x)
        recon_x = self.decode(z)
        return recon_x
```

In this example, we are using a 100-dimensional latent space and three convolutional layers that reduce the image dimension by a factor two every time. We achieve this by using a 4x4 kernel and stride size of 2. Starting with a 3-channel image, we use 64 learnable filters, each of size (3,4,4), i.e. one for each channel. We then pass the result through 128 learnable filters, each of size (64, 4, 4), and finally through 256 learnable filters of dimension (128, 4, 4). The output are 256 4x4 images, which are flattened into a single vector and passed through the fully connected layer. All of this happens in the *encode()* function.

In the *decode()* function, the inverse happens. We first use a fully connected network to project from the latent dimension to a 256x4x4 pixels, which are then reshaped into 256 4x4 images and passed through the convolutional layers. The result is again a 32x32 image.

We can instantiate this model using

```
# Initialize model and optimizer
latent_dim = 1024
model = ConvAE(latent_dim=latent_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

Here, we are using a latent dimension of 1024.

If you think that this process uses quite a lot of parameters, that is right. We can inspect the model size using

```
def get_model_size(model):
    total_params = sum(p.numel() for p in model.parameters())
    return total_params

model_size = get_model_size(model)
print(f"Model size: {model_size} parameters")
```

```
Model size: 9711235 parameters
```

or around 9MB. This is not much, given that the original CIFAR10 dataset is 230MB of data. The majority of these parameters are used up by the 4096x1024 fully-connected network that projects down into the 1024-dimensional latent space and later out of it.

### Training

Before training the model, we will need to define what “loss” is. Our goal is to train the model above so that the output of the model is as close as possible to the input on a pixel-by-pixel basis. We can achieve this by summing over the mean-square error over all pixels:

```
def loss_function(recon_x, x):
    recon_loss = F.mse_loss(recon_x, x, reduction="sum")  # MSE for image recons
    return recon_loss
```

We can now train the model using this loss function for 20 epochs, that is 20 complete presentations of the entire training set. You can achieve this in around 4 minutes on Google Colab’s T4 GPU.

```
# Loss function
def loss_function(recon_x, x):
    recon_loss = F.mse_loss(recon_x, x, reduction="sum")  # MSE for image recons
    return recon_loss

# Training loop for AE
epochs = 20
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)

        optimizer.zero_grad()
        recon_data = model(data)
        loss = loss_function(recon_data, data)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"====> Epoch: {epoch+1}, Average loss: {total_loss / len(train_loader
```

```
====> Epoch: 1, Average loss: 47.8117
====> Epoch: 2, Average loss: 22.7343
====> Epoch: 3, Average loss: 16.2495
====> Epoch: 4, Average loss: 13.4375
====> Epoch: 5, Average loss: 11.6479
====> Epoch: 6, Average loss: 10.3794
====> Epoch: 7, Average loss: 9.4950
====> Epoch: 8, Average loss: 8.6848
====> Epoch: 9, Average loss: 8.0788
====> Epoch: 10, Average loss: 7.4614
====> Epoch: 11, Average loss: 6.9470
====> Epoch: 12, Average loss: 6.4711
====> Epoch: 13, Average loss: 6.1032
====> Epoch: 14, Average loss: 5.8384
====> Epoch: 15, Average loss: 5.6323
====> Epoch: 16, Average loss: 5.2493
====> Epoch: 17, Average loss: 5.0357
====> Epoch: 18, Average loss: 4.8003
```

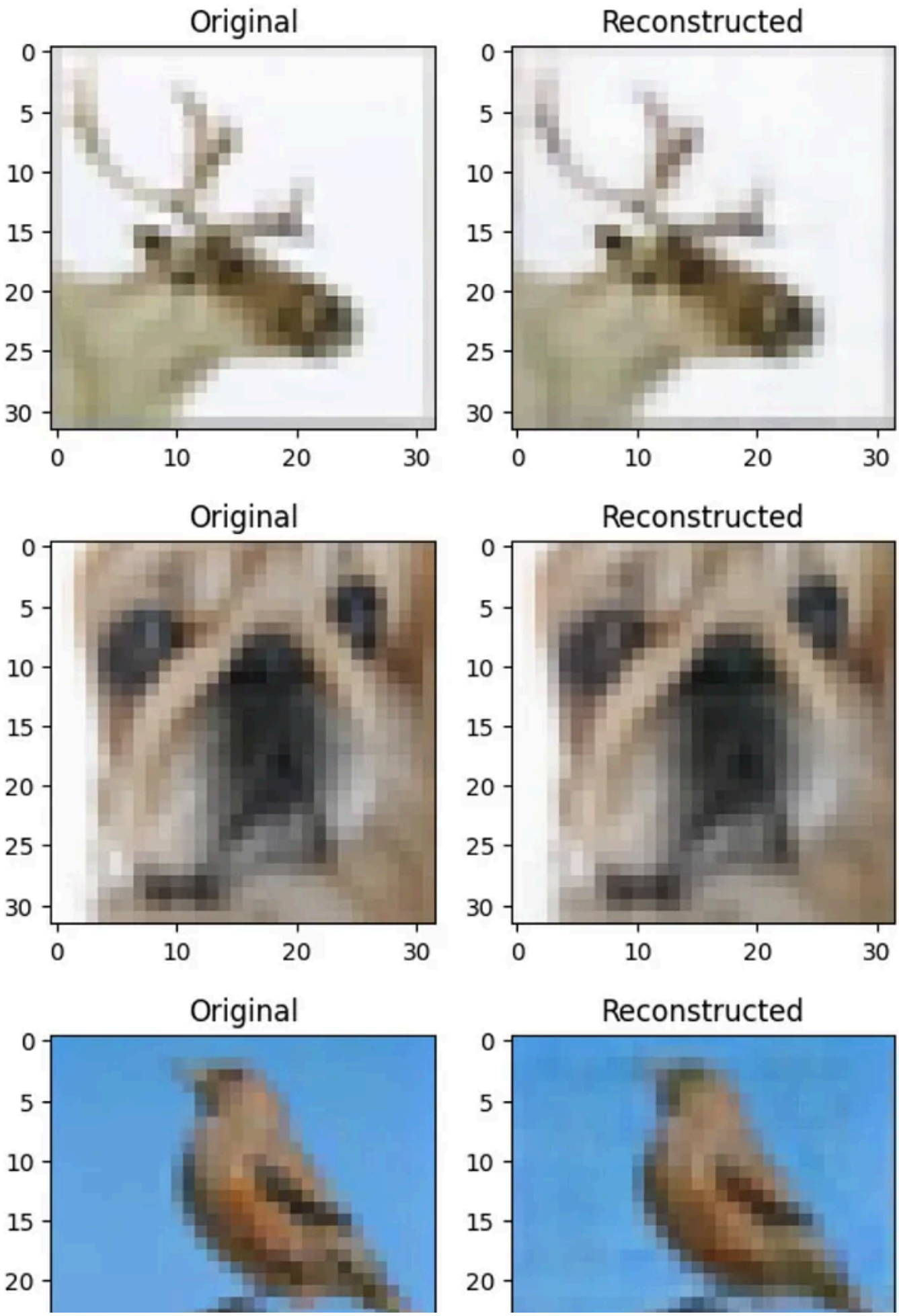
```
====> Epoch: 19, Average loss: 4.6020
====> Epoch: 20, Average loss: 4.4080
```

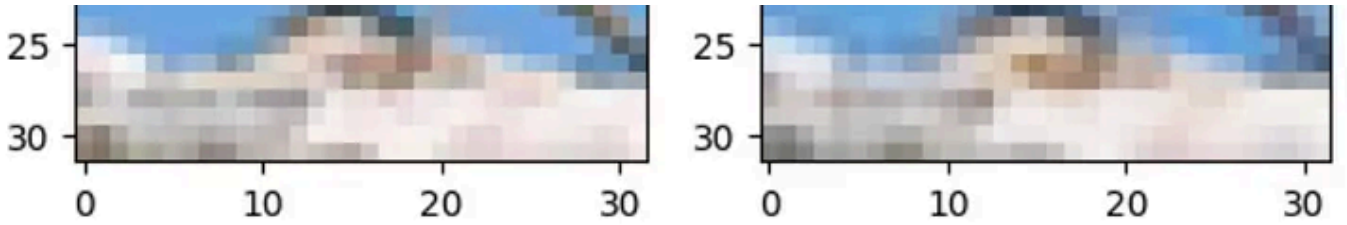
### Testing the Autoencoder

We can now compare the images before and after compression by showing them side by side:

```
# Display some reconstructed images
def show_images(original, reconstructed):
    fig, axes = plt.subplots(1, 2)
    axes[0].imshow(original.permute(1, 2, 0).cpu().numpy())  # Convert tensor to
    axes[0].set_title("Original")
    axes[1].imshow(reconstructed.permute(1, 2, 0).cpu().detach().numpy())
    axes[1].set_title("Reconstructed")
    plt.show()

for _ in range(3):
  # Test reconstruction on a sample
  test_img, _ = next(iter(train_loader))
  test_img = test_img.to(device)
  with torch.no_grad():
      reconstructed_img = model(test_img)[0]

  show_images(test_img[0], reconstructed_img)
```

Here, the *permute()* operation re-arranges the color channels so that imshow shows the correct color images. You should get something like this


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed



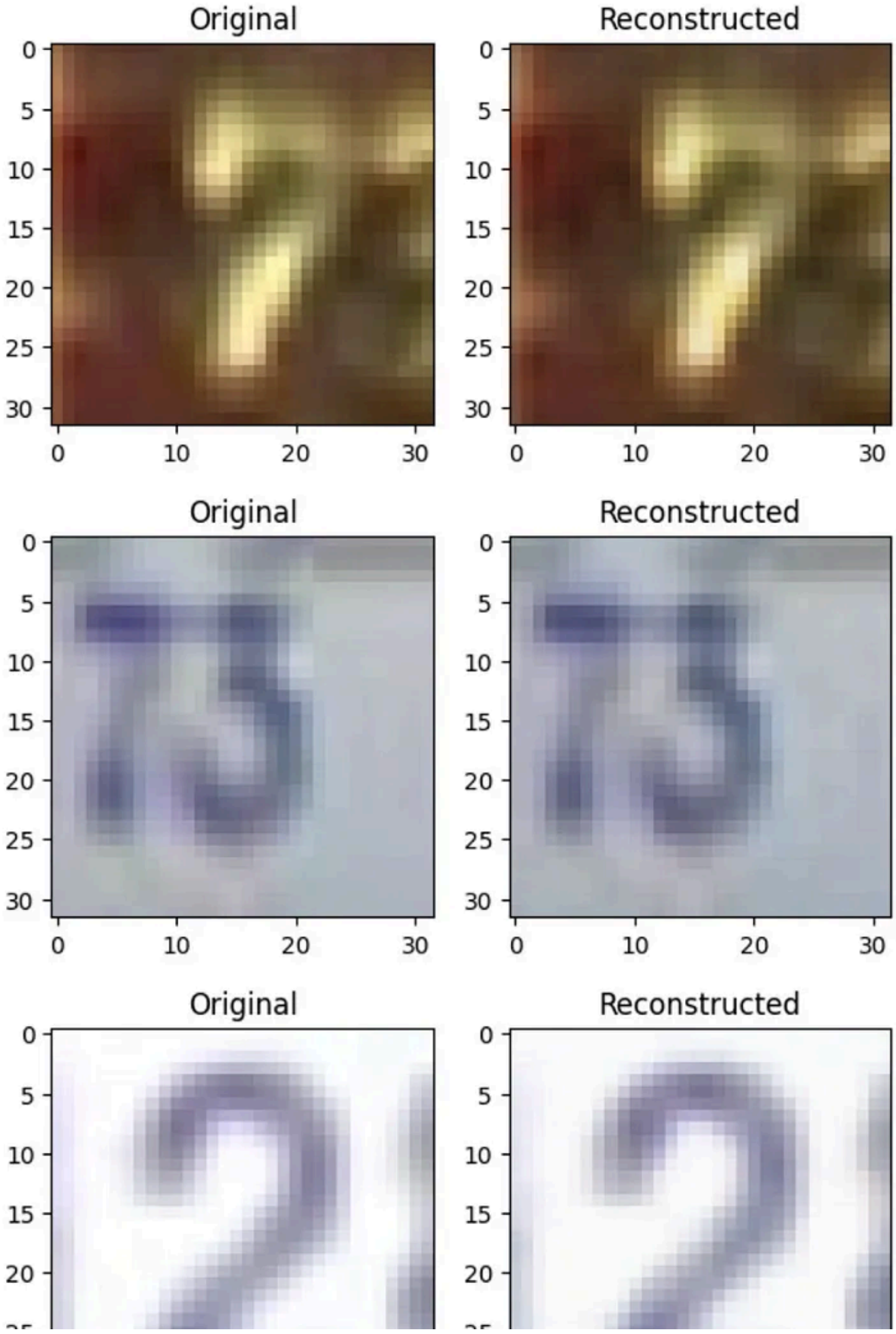

Original (left column) and reconstructed (right column) samples from CIFAR10, essentially compressing 240MB of data into a 9MB Autoencoder.

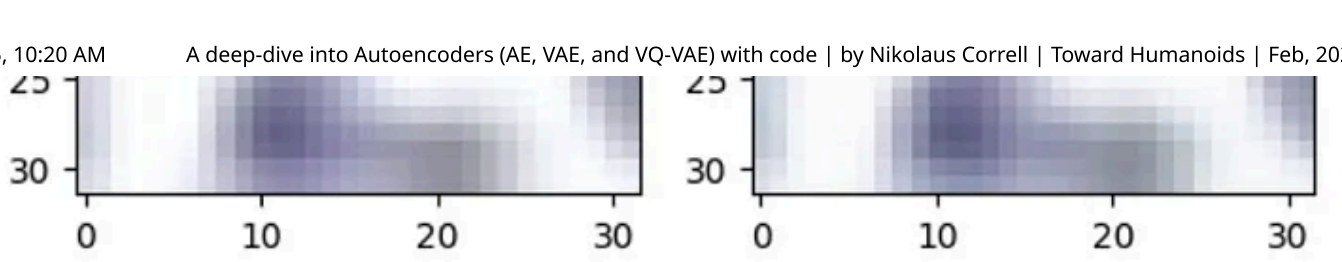
It turns out that 20 epochs are not enough. To achieve the results above, I trained for another 60 epochs to further reduce the loss from 4.4 to 1.8. The results are pretty good. We do observe some “compression artifacts”, for example when inspecting the blue sky in the bottom row.

But how would this work for images that are *not* in the training set? Did the autoencoder learn a compression scheme that generalizes to other, similar images? We can try this by loading a completely different dataset, Google’s Street View House Numbers dataset.

```
svhn_dataset = datasets.SVHN(root="./data", split="test", transform=transform, d
svhn_loader = torch.utils.data.DataLoader(svhn_dataset, batch_size=64, shuffle=T
```

We can now replace *train_loader* in the script above with *svhn_loader* and see, whether the autoencoder that we trained will work with these images as well:


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed




Samples from Google's SVHN dataset (left column) compressed and reconstructed using an autoencoder that has been trained on the CIFAR10 dataset (right), demonstrating the ability of the autoencoder to compress arbitrary images.

This is why autoencoders are such a powerful tool. Once trained, we can use them to project arbitrary data (as long as it is somewhat similar to the kind of data we trained the autoencoder with) into a lower dimensional space. In this case, the autoencoder has learned typical image features that are shared across CIFAR10 and SVHN.

Other applications of such an autoencoder despite compression is image denoising and anomaly detection. In case an image contains noise (or features) that is out-of-distribution for the training images, it is unlikely to be reproduced in the reconstruction. Similarly, to test as to whether an image is "similar" to those that we trained with, we simply have to track the reconstruction loss, which will be higher for images that contain features that the autoencoder has not seen during training.

## Variational Autoencoders

While autoencoders are great at near-perfect reconstruction, there is no way to generate entirely new data. This is interesting for applications such as generative models, e.g. DALL-E, or smoothly blending between two images. *Variational Autoencoders* address this application by treating the latent space probabilistically. For this, every pixel in the latent space is represented by the mean ($\mu$) and standard deviation ($\sigma$) of a Gaussian distribution, i.e. two instead of one value. We can then reconstruct an image by sampling from this distribution, i.e.

$z = \mu + r*\sigma$

Here, $r$ is a random value that follows a Gaussian distribution with zero mean and variance one.

When using a latent dimension of 1024 as in the example above, the latent space is a 1024-dimensional Gaussian distribution. Let's see how this looks like in code:

```
class ConvVAE(nn.Module):
    def __init__(self, latent_dim=100):
        super(ConvVAE, self).__init__()

        # Encoder: Convolutions to extract features
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),  # (B, 64, 16,
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1), # (B, 128, 8
            nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1), # (B, 256,
            nn.ReLU(),
        )

        # Latent space
        self.fc_mu = nn.Linear(256 * 4 * 4, latent_dim)
        self.fc_logvar = nn.Linear(256 * 4 * 4, latent_dim)

        # Decoder: Transposed convolutions for upsampling
        self.decoder_input = nn.Linear(latent_dim, 256 * 4 * 4)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1), #
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1), # (
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1), # (B,
            nn.Sigmoid()  # Output pixel values between 0 and 1
        )

    def encode(self, x):
        x = self.encoder(x)
        x = x.view(x.size(0), -1)  # Flatten
```

```
        mu, logvar = self.fc_mu(x), self.fc_logvar(x)
        return mu, logvar

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        x = self.decoder_input(z)
        x = x.view(x.size(0), 256, 4, 4)  # Reshape to feature maps
        x = self.decoder(x)
        return x

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        recon_x = self.decode(z)
        return recon_x, mu, logvar
```

The encoder and decoder use identical convolutions, but we are using two fully connected layers to compute two latent spaces: one for the mean (mu) and one for the logarithm of the variance (logvar). The reason for representing the logarithm of the variance, not the variance directly, is because we don't want the variance to ever become negative. Instead, we ensure positive values for the standard deviation (square root of the variance) by exponentiating it (*std = torch.exp(0.5 * logvar)*). Here, exponentiating by 0.5 is equivalent to the square root operation.

The decoder then literally draws a random number using the expression *randn_like()*. As the random number is simply a factor in *mu + eps * std*, the gradient computation is not affected, and the model can learn appropriate parameters that minimize the error.

### Training the Variational Autoencoder

If we train this variational autoencoder with the mean-square error loss from above, we will get the exact same results as with a standard autoencoder. In fact, training will ignore the *eps*std* term by converging to a zero standard deviation. If we want the VAE to result in a true Gaussian distribution, we need an additional loss term.

One way to compare two distributions P(x) and Q(x) is the Kullback-Leibler divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$

In our case, distribution Q is a zero-mean, variance-one Gaussian distribution — the distribution that we want — the other is the one that we are actually learning. Here, the sum is going over all possible values for x (an integral for continuous distributions). In our case, we want to compare the trained distribution $N(\mu,\sigma^2)$ with a desired one such as N(0,1) (zero mean, variance one), which simplifies the equation to :

$$KL(\mathcal{N}(\mu, \sigma^2) \parallel \mathcal{N}(0, 1)) = \frac{1}{2} \sum \left(\sigma^2 + \mu^2 - 1 - \log \sigma^2\right)$$

Getting to the above equation requires substituting the actual functions for the normal distributions

$$D_{\mathrm{KL}}(P\|Q) = \sum_x \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \log\left(\frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)}{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)}\right)$$

simplifying the logarithms, and finally summing/integrating over the result. Adding the KL divergence leads to the following loss

```
def loss_function(recon_x, x, mu, logvar):
    recon_loss = F.mse_loss(recon_x, x, reduction="sum")  # MSE for image recons
    kl_div = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())  # KL diver
    return recon_loss + 0.1*kl_div
```

Here, we simply add the mean-square error loss with the KL divergence. As we want to minimize, we are using the negative KL divergence. In this example, I chose a weight of 0.1, which shifts the model toward perfect reconstruction vs. generating new images.

We can now instantiate the new model

```
latent_dim = 1024
model = ConvVAE(latent_dim=latent_dim).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

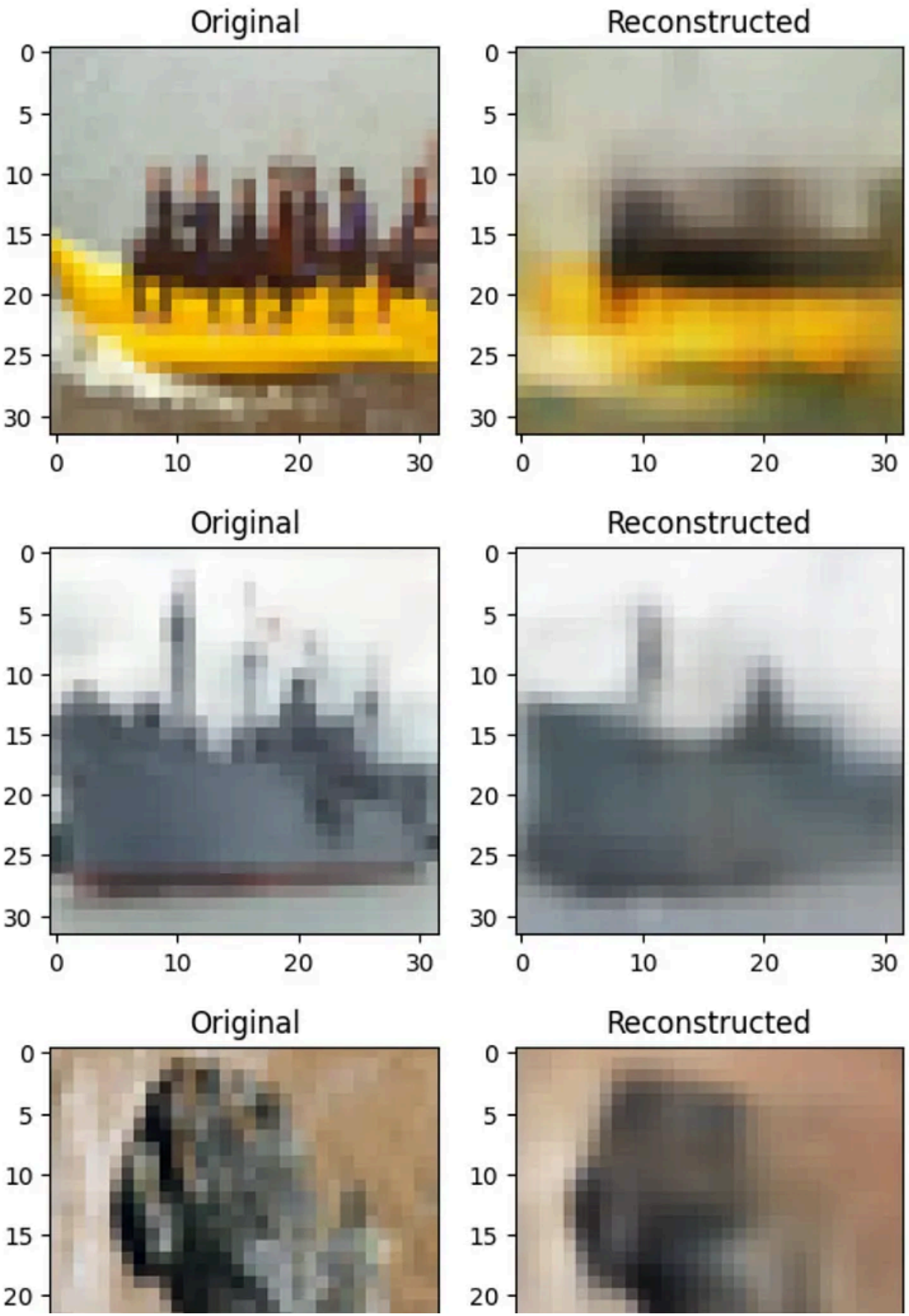
and train it:

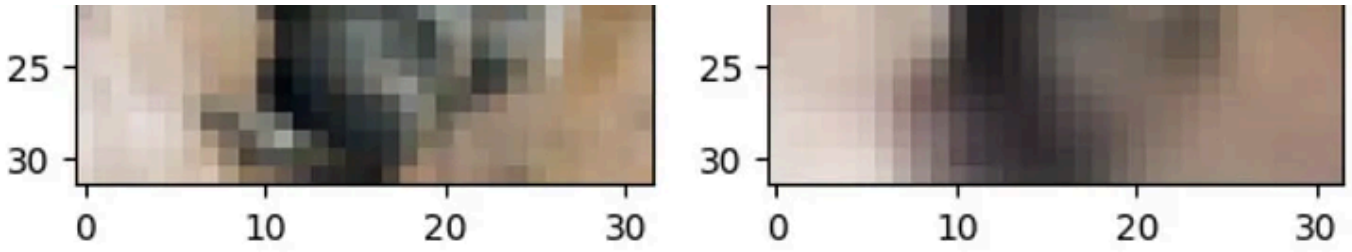
```
epochs = 10
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (data, _) in enumerate(train_loader):
        data = data.to(device)
```

```
            optimizer.zero_grad()
            recon_data, mu, logvar = model(data)
            loss = loss_function(recon_data, data, mu, logvar)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

            #if batch_idx % 100 == 0:
            #    print(f"Epoch [{epoch+1}/{epochs}], Step [{batch_idx}/{len(train_lo

        print(f"====> Epoch: {epoch+1}, Average loss: {total_loss / len(train_loader
```

The loss goes down to around 34 after 20 epochs, 33 after 30 epochs, to 32 after 40 epochs, and only improves very slowly for 50 and 60 epochs (32.13 error).

This model is not only larger (around 13MB), but generating random numbers also make it learn a little slower than the standard autoencoder. We also don't expect the loss to go as low as that of the autoencoder as we indeed generate random images (assuming the variance is not zero). Indeed, the resulting images are pretty noisy:


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed




CIFAR10 images (left column) after reconstruction using a variational autoencoder (right column).

Note: to generate this output using the code above change the last line to

```
show_images(test_img[0], reconstructed_img[0])
```

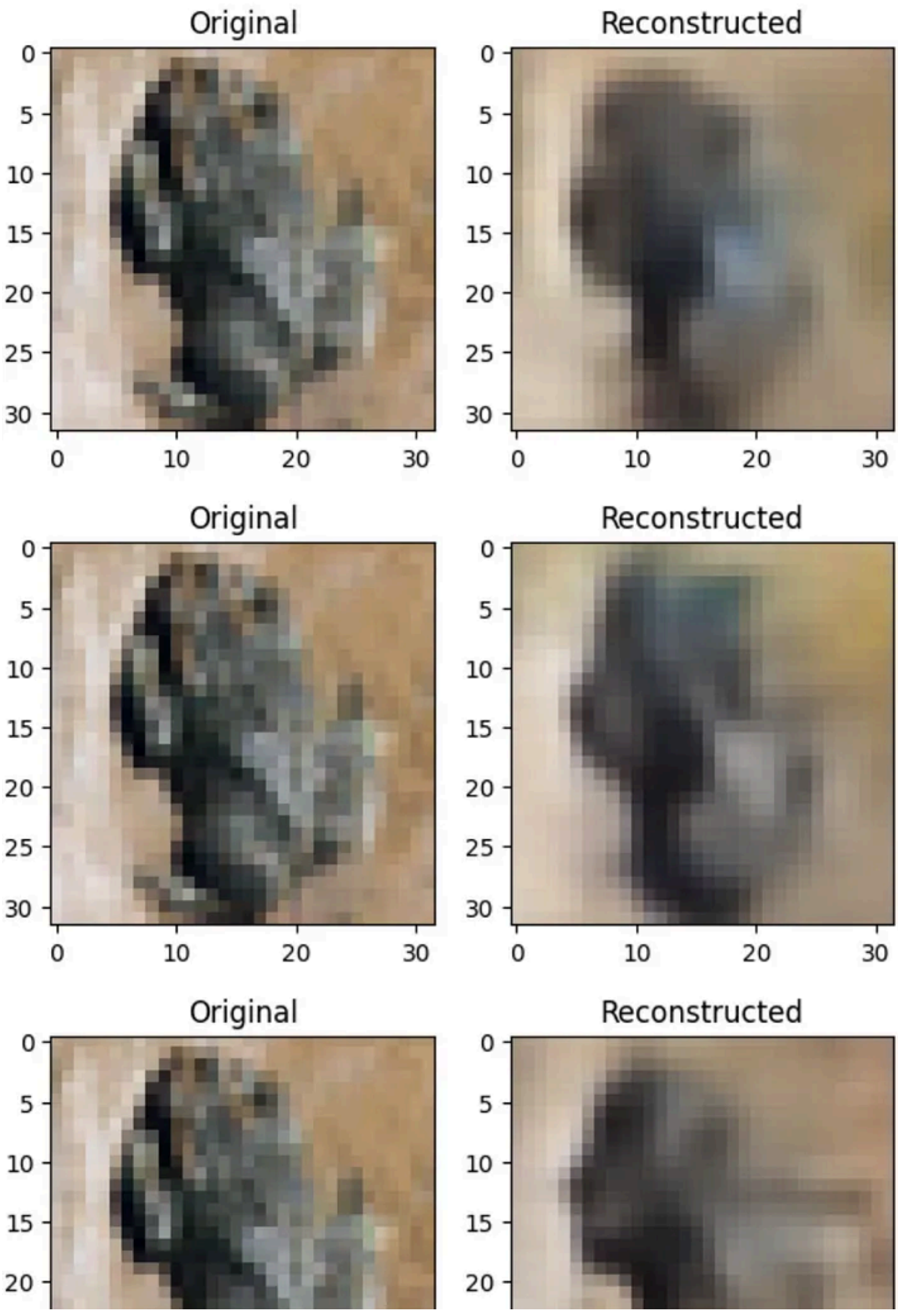
While this might improve with more training, one can easily see what happens here when further reducing the contribution of the KL loss term. For example, setting it to 0.001 will give results that resemble that of the original autoencoder as the autoencoder did not learn distributions, but simply tries to match the presented data as good as possible.

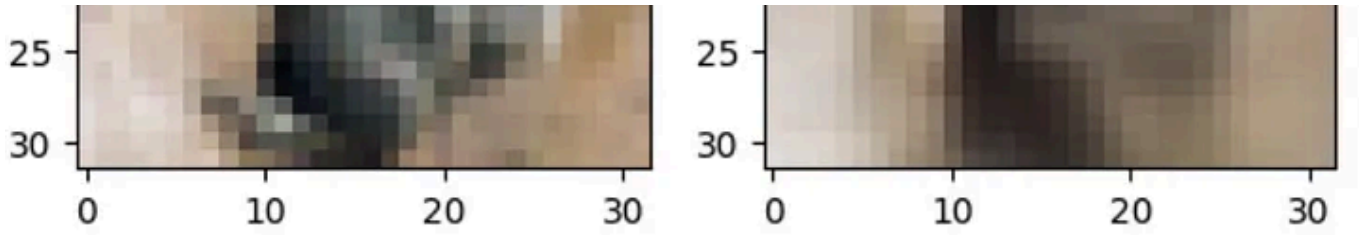
So why do we care about variational autoencoders? They allow us to generate variations of images. We can run the code to display images without actually loading a new one:

```
def show_images(original, reconstructed):
    fig, axes = plt.subplots(1, 2)
    axes[0].imshow(original.permute(1, 2, 0).cpu().numpy())  # Convert tensor to
    axes[0].set_title("Original")
    axes[1].imshow(reconstructed.permute(1, 2, 0).cpu().detach().numpy())
    axes[1].set_title("Reconstructed")
    plt.show()

for _ in range(3):
  # Test reconstruction on a sample
  with torch.no_grad():
```

```
        reconstructed_img = model(test_img)[0]

    show_images(test_img[0], reconstructed_img[0])
```

to obtain three reconstructed images of the dog:


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed




Albeit noisy, each of the above image is slightly different. More importantly, the resulting latent space is what is known as *structured*. Representations in latent space that are close, are also generating similar outputs. This can be seen when trying to blend two different images by first encoding them using the VAE, then averaging their latent spaces:

```
# Function to display images in a grid
def display_images(images, titles=None):
    fig, axes = plt.subplots(1, len(images), figsize=(12, 4))
    for i, ax in enumerate(axes):
        image = images[i].permute(1, 2, 0).cpu().detach().numpy()  # Rearrange d
        ax.imshow(np.clip(image, 0, 1))
        ax.axis('off')
        if titles:
            ax.set_title(titles[i])
    plt.show()

# Function to blend latent vectors
def blend_latent_vectors(latent1, latent2, alpha=0.5):
    return alpha * latent1 + (1 - alpha) * latent2

# Put images in the correct device if needed
image1 = test_img[0].to(device)
image2 = test_img[1].to(device)

# Encoder step for image1 and image2 (getting their latent vectors)
mu1, logvar1 = model.encode(image1.unsqueeze(0))
mu2, logvar2 = model.encode(image2.unsqueeze(0))

# Interpolate between the latent vectors (blend in latent space)
alpha = 0.5  # Set blending factor (0 for image1, 1 for image2, 0.5 for the blen
blended_mu = blend_latent_vectors(mu1, mu2, alpha)

# Decode the blended latent vector back into an image
blended_image = model.decode(blended_mu)

# Display images
```

```
images = [image1, image2, blended_image[0]]  # Show the first image from the bat
titles = ["Image 1", "Image 2", "Blended Image"]
display_images(images, titles)
```


Blending the latent spaces of two deer (left) and a truck (middle) and reconstructing them via a VAE (right).

The result is not mind-blowing (like what full-blown versions of such autoencoders such as in DALL-E and other generative models are capable of), but you get the idea: the result is maintaining key features of both images such as the two individuals, maintaining features of the truck in the background.

## Vector-Quantized Variational Autoencoder (VQ-VAE)

The Variational Autoencoder is offering a structured latent space from which we can (1) sample and which (2) maintains similarity between points that are close in latent space and the resulting images. Both of these properties are critical for generative models in which we want to sample data from the training distribution and be able to mix and match their features. The drawback of the VAE is that it sacrificies reconstruction performance by learning average images — represented by their mean- rather than fit the input data perfectly. This is why results appear blurred.

The Vector-quantized Variational Autoencoder (VQ-VAE) addresses this challenge by using a discrete *codebook*, or alphabet, that can be learned, with one code for each entry in the latent space. After the encoding, the VQ-VAE will select the code that has the closest Euclidian distance to the encoding, therefore forcing the embedding into a mold. When considering text, this would be akin to reduce to a subset of the alphabet, allowing the optimizer to choose which characters these are.

In code, this is realized using a trainable embedding of dimension *embedding_dim* and length *num_embedding,* that is the number of codes each with dimension *embedding_dim*. After generating an embedding via a series of convolutions, we end up with *embedding_dim* 4x4 pixel images. (This requires a final 2D convolution with a 1x1 kernel to move from the 256 channels that we have used throughout to *embedding_dim* channels). These are flattened and rearranged so that we end up with 4x4xB (with B the number of batches) embeddings of dimension *embedding_dim*. These are the candidates for codes in our codebook.

```
    def vector_quantize(self, z):
        B, C, H, W = z.shape  # (B, embedding_dim, 4, 4)
        z_flattened = z.permute(0, 2, 3, 1).reshape(-1, C)  # (B*16, embedding_d

        distances = (z_flattened.unsqueeze(1) - self.embeddings.weight.unsqueeze
        encoding_indices = distances.argmin(1)  # (B*16,)

        z_q = self.embeddings(encoding_indices).view(B, H, W, C).permute(0, 3, 1
        z_q = z_q + (z - z_q).detach()  # Preserve gradients

        return z_q, encoding_indices.view(B, H, W)  # Output is (B, 4, 4) indice
```

We can then compute the Euclidian distance between each candidate and codebook entry. We then take the indices corresponding to the smallest distances, and grab the corresponding entry in the codebook. This requires some non-trivial re-arranging (using *view(B,H,W,C)*) to restore the original data, and swapping entries around (using *permute()*).

As gradients would not naturally pass through this kind of look-up, this is followed by a computational trick that is known as “straight-through estimator”:

```
z_q = z_q + (z - z_q).detach()
```

If you look carefully, you will see that the operation actually reduces to *z_q = z*. As we remove the gradient from the subtraction operation using *.detach()*, the gradient of z_q will behave as it came from z, making the function *vector_quantize()* differentiable.

Here is the complete model for the VQ-VAE:

```
class ConvDQ_VAE(nn.Module):
    def __init__(self, num_embeddings=512, embedding_dim=64):
        super(ConvDQ_VAE, self).__init__()

        # Encoder: Convolutions to extract features
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1),  # (B, 64, 16,
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1), # (B, 128, 8
            nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1), # (B, 256,
            nn.ReLU()
```

```
        )

        self.encoder_output = nn.Conv2d(256, embedding_dim, kernel_size=1)

        # Latent space: VQ codebook
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        self.embeddings = nn.Embedding(self.num_embeddings, self.embedding_dim)
        nn.init.xavier_normal_(self.embeddings.weight)

        # Decoder: Transposed convolutions for upsampling
        self.decoder_input = nn.Conv2d(embedding_dim, 256, kernel_size=1)

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1), #
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1), # (
            nn.ReLU(),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1), # (B,
            nn.Sigmoid()  # Output pixel values between 0 and 1
        )

    def encode(self, x):
        x = self.encoder(x)
        z = self.encoder_output(x)
        return z

    def vector_quantize(self, z):
        B, C, H, W = z.shape  # (B, embedding_dim, 4, 4)
        z_flattened = z.permute(0, 2, 3, 1).reshape(-1, C)  # (B*16, embedding_d

        distances = (z_flattened.unsqueeze(1) - self.embeddings.weight.unsqueeze
        encoding_indices = distances.argmin(1)  # (B*16,)

        z_q = self.embeddings(encoding_indices).view(B, H, W, C).permute(0, 3, 1
        z_q = z_q + (z - z_q).detach()  # Preserve gradients

        return z_q, encoding_indices.view(B, H, W)  # Output is (B, 4, 4) indice

    def decode(self, z):
        x = self.decoder_input(z)
        x = self.decoder(x)
        return x

    def forward(self, x):
        z = self.encode(x)
        z_q, encoding_indices = self.vector_quantize(z)
        recon_x = self.decode(z_q)

        # Commitment loss: Encourages the encoder's output to be close to the co
```

```
        commitment_loss = F.mse_loss(z, z_q.detach())  # Updates the encoder
        codebook_loss = F.mse_loss(z.detach(), z_q)    # Updates the codebook
        loss = commitment_loss + codebook_loss

        return recon_x, loss, encoding_indices
```

In addition to the vector quantization step, the model returns an additional loss function and the codebook indices that have actually been chosen. The loss function takes care of two things:

1. **Commitment Loss**: the encoder embeddings should be as close as possible to the actual codebook entries. Here, *.detach()* prevents the gradients from entering the codebook, allowing the encoder to adjust to move closer to the codes in the codebook.
2. **Codebook Loss**: the codebook entries should be as close as possible to the encoder embeddings. Here, *.detach()* prevents the gradients from flowing into the encoder, allowing the codebook embeddings to change.

We can now add these two together and chose appropriate weighting (here 1-to-1), to emphasize either codebook or encoder training.

We can instantiate it using

```
# Initialize the model and optimizer
model = ConvDQ_VAE(num_embeddings=4096, embedding_dim=128).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

resulting into a 1.9MB large model.

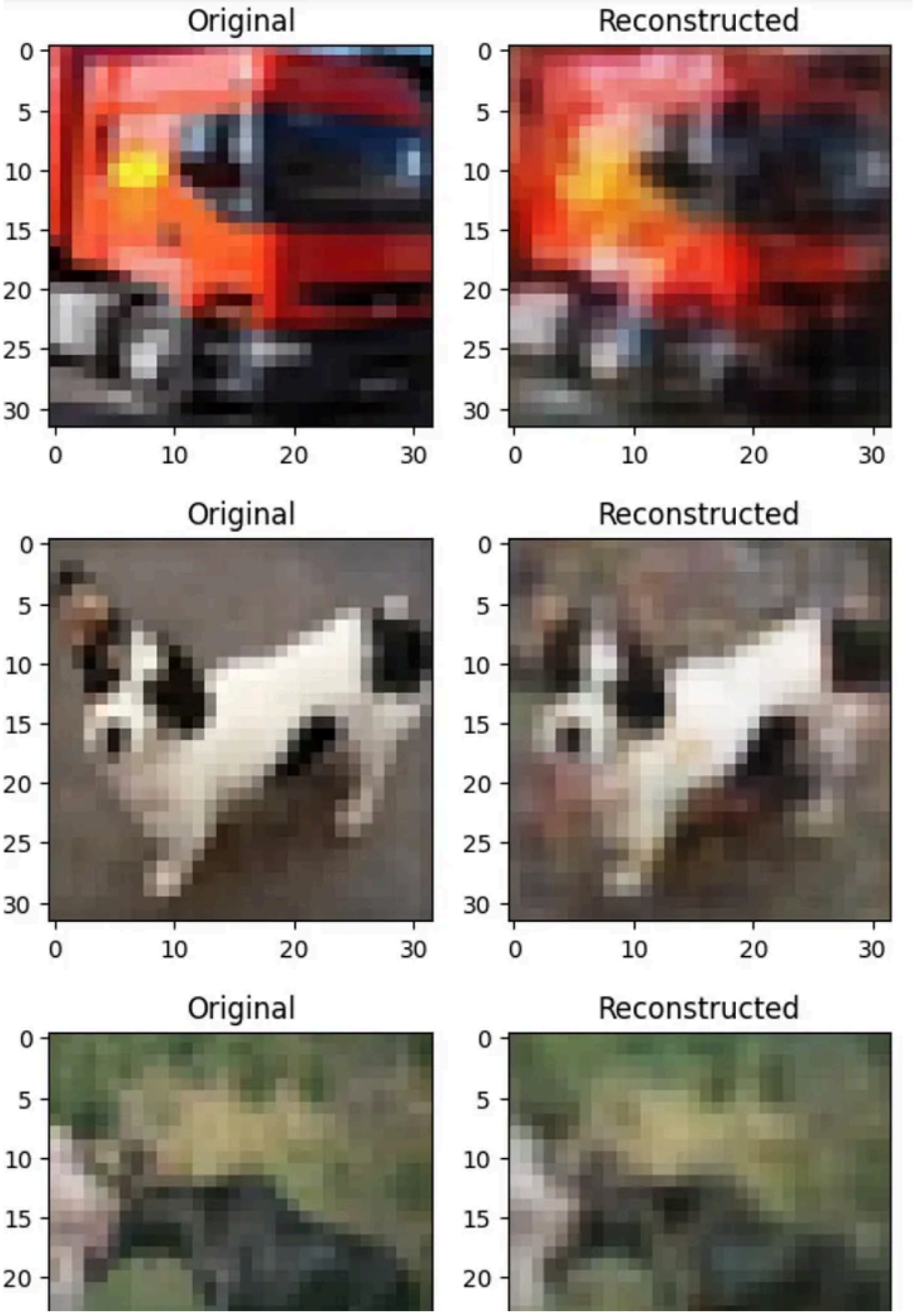
## Training the VQ-VAE

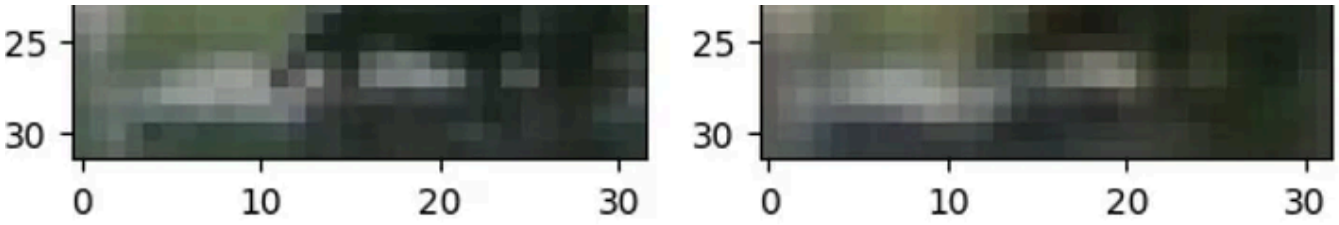
Training is business as usual, we just need to take the additional loss function into account:

```
epochs = 30

# Training loop
for epoch in range(epochs):
    model.train()
    total_loss = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data to GPU if available
        data = data.cuda() if torch.cuda.is_available() else data

        # Forward pass
        recon_x, commitment_loss, encoding_indices = model(data)

        # Reconstruction loss (MSE)
        recon_loss = F.mse_loss(recon_x,data) # torch.mean((recon_x - data) ** 2

        # Total loss: reconstruction loss + commitment loss
        loss = recon_loss + commitment_loss

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader.dataset)
```

Here, the comittment and codebook loss are added to the reconstruction loss that we know from previous autoencoders.

The results are compelling: Reconstructions are much less blurry than those from the VAE, and quite good reconstructions can be obtained *using only 1.9MB of trainable parameters.* Yes, the result is a little more noisy than what we have obtained with the Autoencoder further above, but we are able to get there at a 5x size reduction in model parameters. In addition, if we want to use the VQ-VAE for datacompression, we don't even have to provide the entire latent space, but simply the indices into the codebook. This makes this approach very compelling to generative models such as transformer decoders, which have to simply generate a sequence of codebook entries to generate consistent, high resolution images.


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed




CIFAR10 images (left column) and reconstructions from a 1.9MB VQ-VAE.

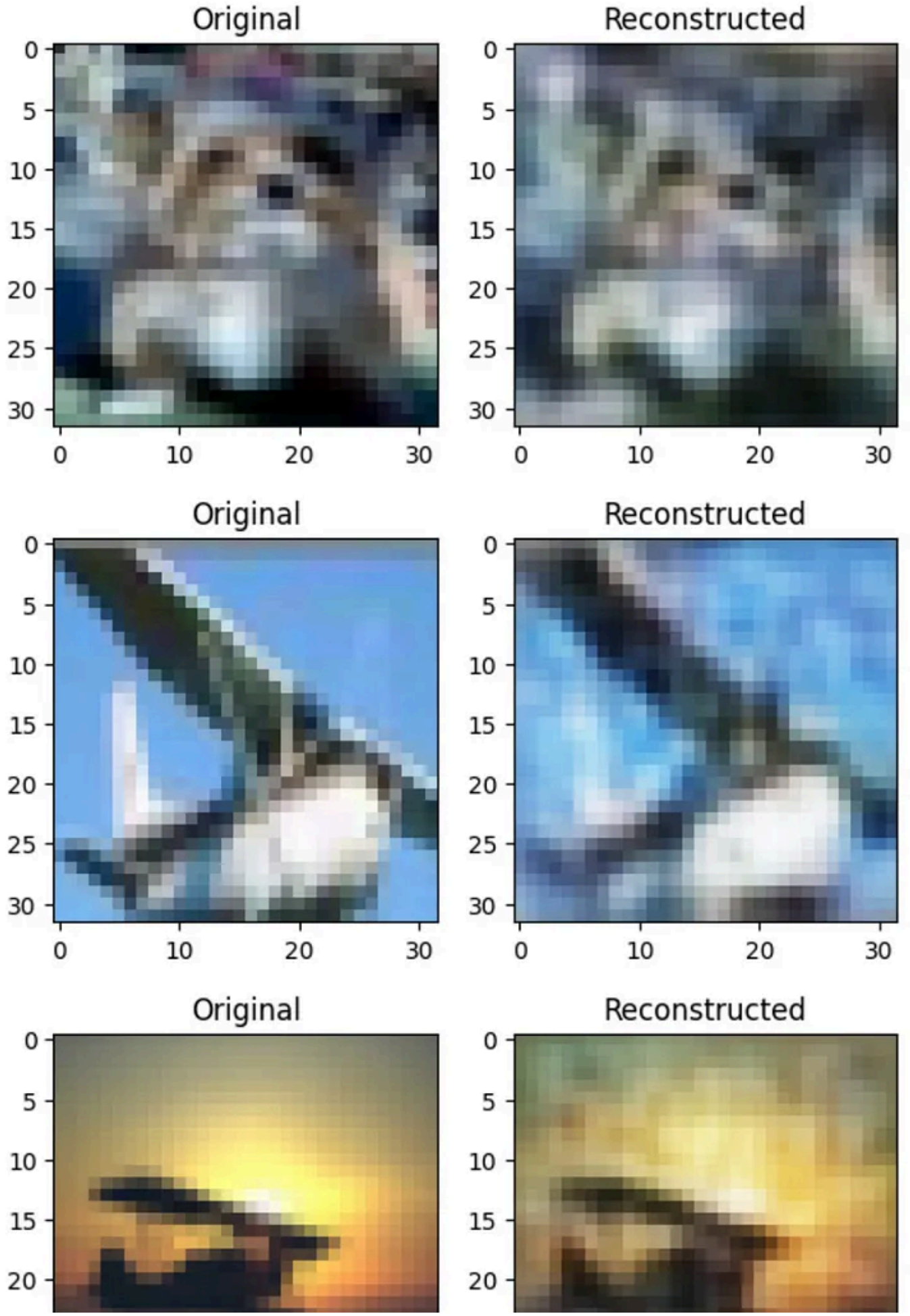
We can also check how many of the 4096 codes we let the model learn are actually used:

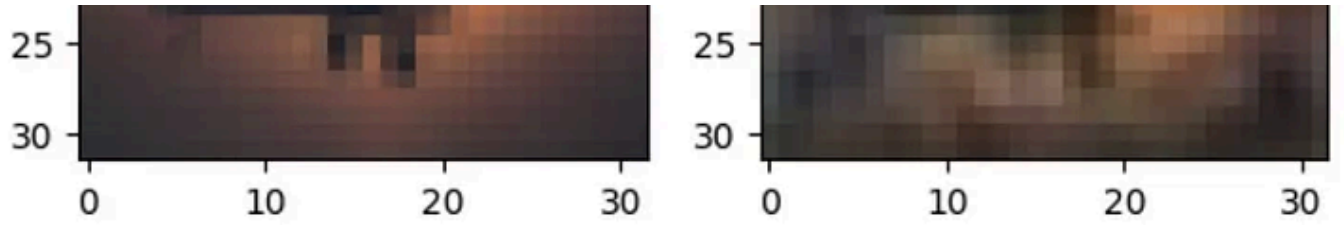
```
batch_of_images, _ = next(iter(train_loader))
batch_of_images = batch_of_images.to(device)

recon_x, loss, encoding_indices = model(batch_of_images)
encoding_indices.unique()
```

With the model above, it turns out that only very few unique codebook entries are used. This might be due to the fact that the encoder itself is very rich, letting the system behave more like a standard autoencoder that uses a single latent space. We can force the model to emphasize diversity in codebook usage by computing a probability distribution over the codebook indices that were actually used and then compute the entropy of this distribution:

```
        p = encoding_indices.view(-1).bincount(minlength=self.num_embeddings).fl
        p /= p.sum()  # Normalize to a probability distribution
        entropy_loss = - (p * p.log()).sum()  # Encourage uniform code use
        loss -= entropy_loss  # Subtracting makes it a regularization term
```

This can be added to the *forward()* function just after computing the loss. Although this has helped with producing more codebook usage, we still seem not to need more than 32 or so codes. Indeed, we can train the model with num_embeddings=32, each with 128 dimensions. This reduces the parameters to around 1.4M and yields the following results after training for 30 epochs:


Original
Reconstructed
Original
Reconstructed
Original
Reconstructed




We can see that noise has further increased, but not by too much.

If you want to learn more about generative models, follow up with this article:

**Transformer Attention Intuitively Explained With Examples**

Building a Transformer from scratch to build a simple generative model

medium.com

Autoencoder | Variational Autoencoder | Vector Quantization | Machine Learning | Computer Vision

## Published in Toward Humanoids

98 Followers · Last published 2 days ago

Follow

Ongoing pre-publication research on humanoids and related technologies including paper reviews and tutorials.

## Written by Nikolaus Correll

347 Followers · 71 Following

Follow

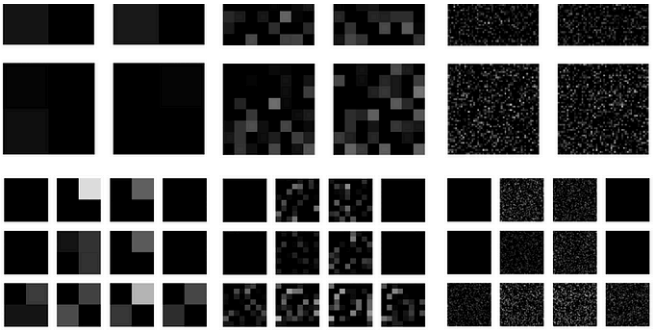
Nikolaus is a Professor of Computer Science and Robotics at the University of Colorado Boulder, robotics entrepreneur, and consultant.

## No responses yet

Alex Mylnikov

What are your thoughts?

## More from Nikolaus Correll and Toward Humanoids

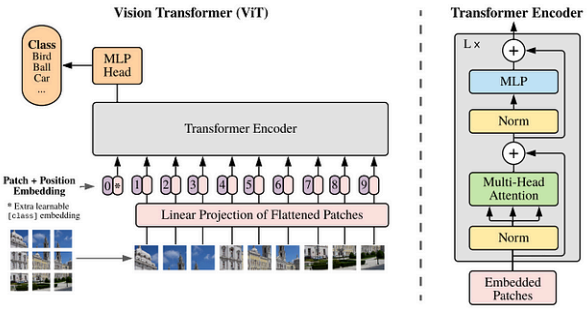


In Toward Humanoids by Nikolaus Correll

### Understanding Image Patch Embeddings

From simple unfolding to 2D convolutions

Feb 3 21

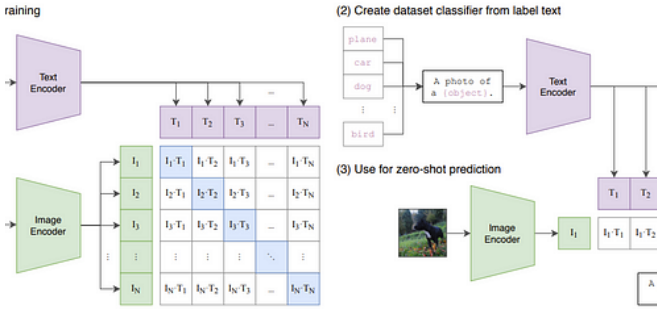


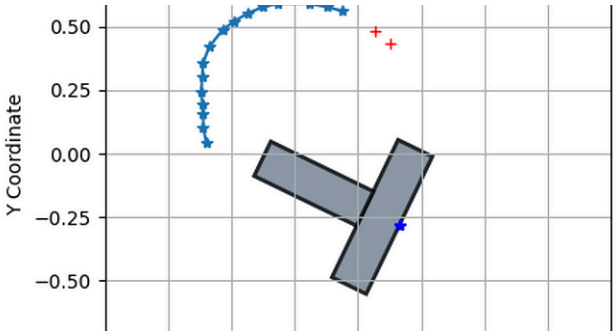

In Toward Humanoids by Matt Nguyen

### Building a Vision Transformer Model From Scratch

The self-attention-based transformer model was first introduced by Vaswani et al. in their...

Apr 4, 2024 236 4




In Toward Humanoids by Matt Nguyen

## Building CLIP From Scratch

Open World Object Recognition on the FashionMNIST Dataset

May 16, 2024 195 4

In Toward Humanoids by Nikolaus Correll

## Robotic Behavior Cloning I: Auto-regressive Transformers

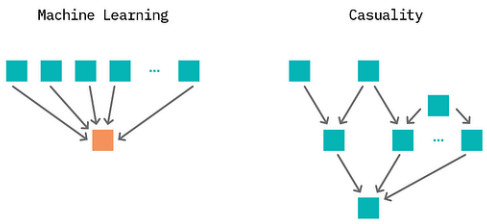
Training an auto-regressive transformer for the Push-T experiment from scratch

Mar 9 167 1

See all from Nikolaus Correll

See all from Toward Humanoids

# Recommended from Medium



In Generative AI by Abby Morgan

## Explainable AI: Visualizing Attention in Transformers

And logging the results in an experiment tracking tool

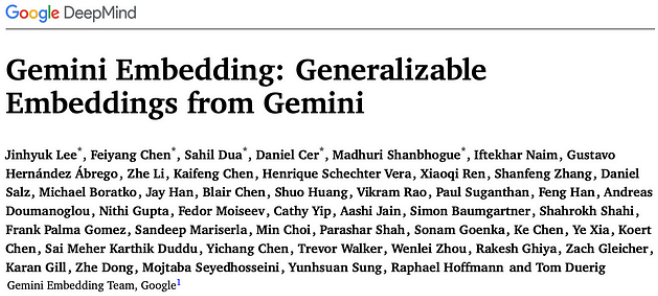
Jul 17, 2023 1.8K 10

Karan_bhutani

## The Causal Revolution in Machine Learning: Moving Beyond...

Causal Machine Learning (Causal ML) represents a fundamental shift in how we...

Mar 3 122 5




Ritvik Rastogi

## Papers Explained 330: Gemini Embedding

Gemini Embedding leverages the power of Gemini to produce highly generalizable...

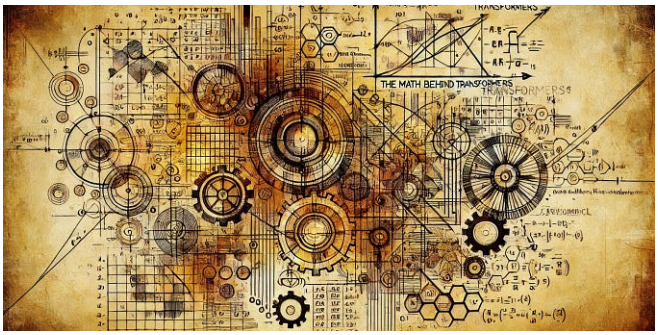
4d ago


In Cubed by Djohra IBERRAKEN

## NeurIPS 2024 Top Papers: Comprehensive Overview

© Logo of NeurIPS

Feb 2 172




Cristian Leo

## The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its...

In The Pythoneers by Abhay Parashar

## How to Explain Each Core Machine Learning Model in an Interview

From Regression to Clustering to CNNs: A Brief Guide to 25+ Machine Learning Models

Jul 25, 2024 1K 9 6d ago 1.3K 11

See more recommendations