Auto Encoders (and two variations)

- https://arxiv.org/pdf/2003.05991.pdf
- https://deepai.org/machine-learning-glossary-and-terms/manifold-hypothesis
- https://en.wikipedia.org/wiki/Latent_space
- https://avandekleut.github.io/vae/
- https://www.youtube.com/watch?v=rZufA635dq4&t=1413s
- https://stats.stackexchange.com/questions/455560/why-is-the-mean-and-log-variance-specified-as-the-output-of-an-inference-network
- https://arxiv.org/pdf/1711.00937.pdf

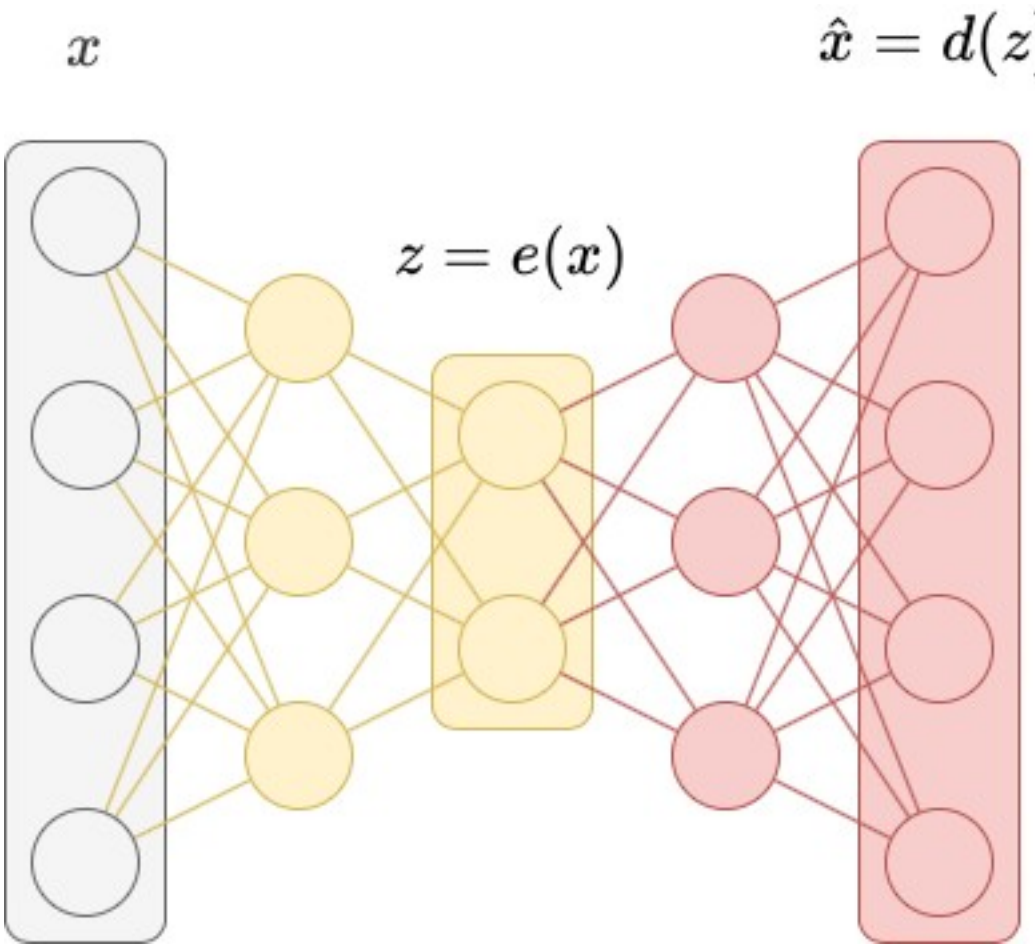
Auto encoders are unsupervised neural networks that learn to reduce data dimensions to efficiently compress data and reconstruct it.

They are generally composed of two different parts, the encoder, which compresses the input *x* into a latent space, and a decoder that reconstructs the data back into an output *xhat*. The goal is to minimize the loss between the input *x* and the output *xhat*.

Auto Encoders

The manifold hypothesis states that many high level data is actually low level data embedded in high dimensional space. That means data that exists in many dimensions can actually be represented in a much more simple way. For example, a sphere can be viewed as an infinite number of points that exist in a fixed distance from its center in 3d, each with their own three dimensional coordinates. However, a sphere can also be represented as just a center and a radius, reducing the potential large amount of data to just two data points.

This is the core idea of autoencoders. The encoder portion reduces the dimensionality of the input into the latent space and the decoder uses the compressed data to reconstruct back into the output. The latent space is the dimensional space that is used for the latent vector, which is the lower dimensional representation of the input *x*. Since the dimensionality of the latent space is the smallest in the entire model, it is also known as the 'bottleneck' of the model.



The encoder learns a nonlinear transformation *e* and the decoder learns a nonlinear transformation *d* such that the latent *z* is

$z = e(x)$

and the reconstructed *xhat* is

$xhat = d(z) = d(e(x))$

An autoencoder is the composition of the encoder and decoder and the output of the encoder is the input of the decoder. That means the function *f* of the autoencoder is the same as $d(e)$

The autoencoder can be trained end-to-end, which means that the entire model is trained as a whole. To be more specific, the loss function that is back propagated is applied to the entire autoencoder model. An advantage of training end-to-end is that it is simpler to train rather than training the encoder and decoder components separately. This also means we only need to use one loss function for the entire model. For an autoencoder, this means we can use the reconstruction loss between *xhat* and *x*, such as the MSELoss between each pixel of the output and input. Since the loss function applies to the whole model and depends on both the encoder and decoder simultaneously, both components are being optimized for each other too.

The goal is to force the autoencoder model to fit data through the bottleneck, so that the autoencoder learns the most significant patterns in the data. Due to this restriction, some information may be lost, but the latent vector contains crucial data and is much smaller than the input.

Model

Setup

Encoder and Decoder

Both the encoder and decoder use three convolutional layers because they are well suited to extract spatial information and patterns from the image data. We then use a fully connected layer to learn significant representations based on the spatial information.

We arbitrarily chose the number of hidden layers and latent dimensions.

Autoencoder model

The autoencoder is the composition of the encoder and decoder.

We use the MSE loss (squared l2 norm) function as our loss function.

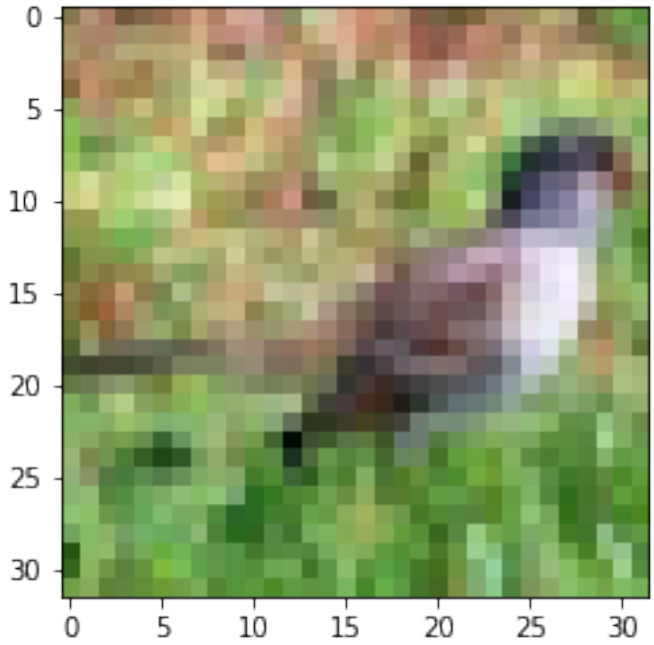
Hyperparameters

Run model

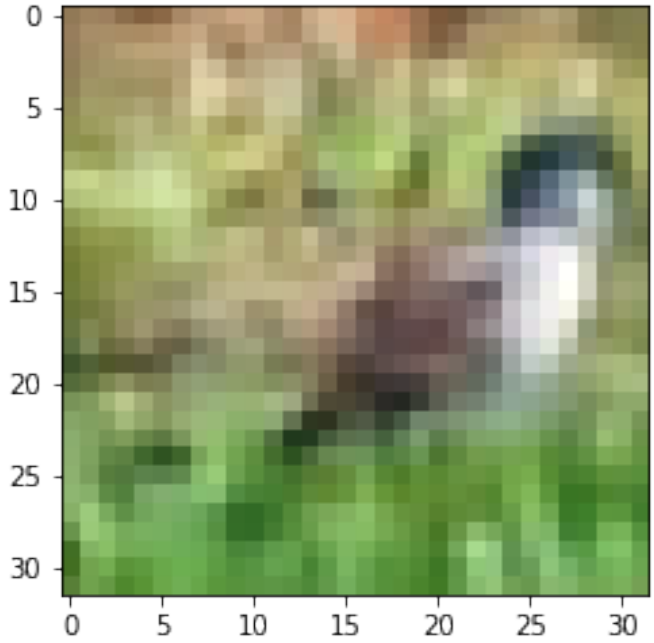
Result

```
(original, _) = next(iter(data_loader))
original = original[0].to(device)

x = original.cpu()
xhat, loss = model(original[None,:])
print('Loss:', loss)

plt.imshow(np.transpose(x, (1,2,0))+0.5)
plt.show()
plt.imshow(np.transpose(xhat[0].cpu().detach().numpy(), (1,2,0))+0.5)
plt.show()
```

```
Loss: tensor(8.2589, device='cuda:0', grad_fn=<MseLossBackward0>)
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

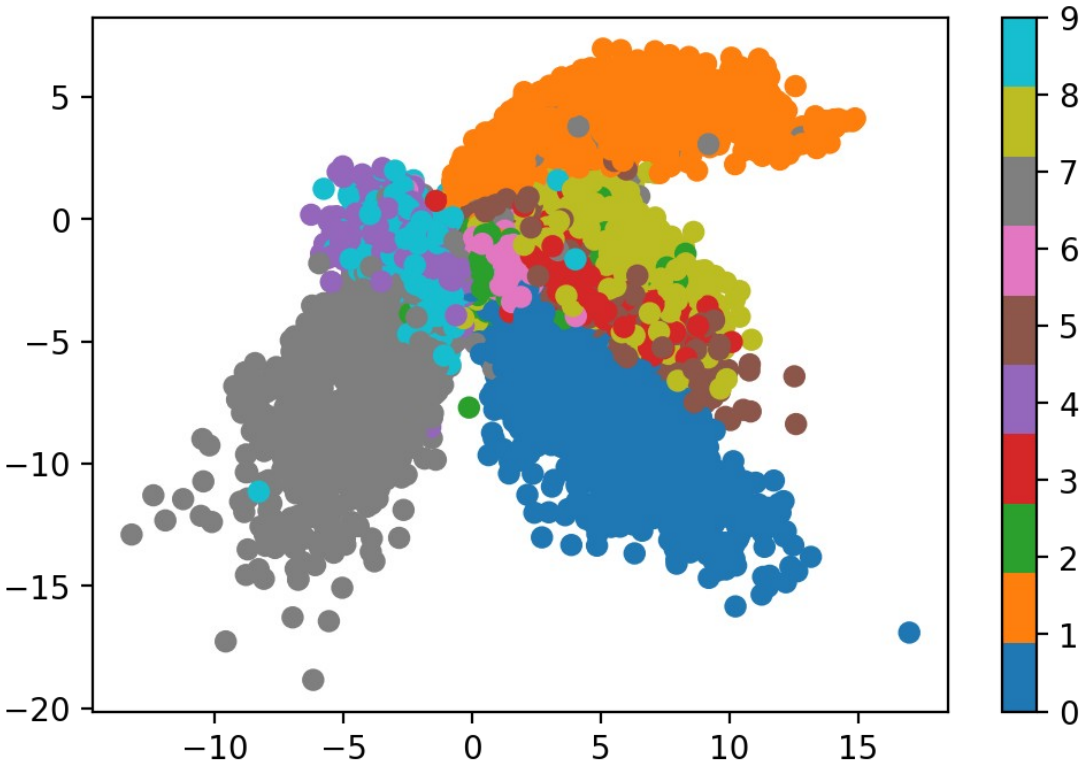


Here we can see a comparison between the input image *x* and output image *xhat*. Due to forcing the model to lower the dimensionality through a bottleneck, the reconstructed output keeps the general patterns at the cost of the details on edges, however that does not seem to be an issue for a dataset like CIFAR10, which already uses a small number of dimensions per data sample (latent variable is about 1/6 the size of the input).

Variational Autoencoder (VAE)

To generate new data, one could assume we can just give the decoder component of the autoencoder a random latent variable and it would reconstruct the output such that it uses previously trained patterns to turn the random latent variable into an actual image. However, the autoencoder had no incentive to regularize the latent variables it generated. This means that the latent variables it constructed when training could be irregular, so a new random latent variable could make no sense.

Below is an example of an autoencoder trained on the MNIST dataset using a latent space with two dimensions so it can be plotted on an (x, y) graph.



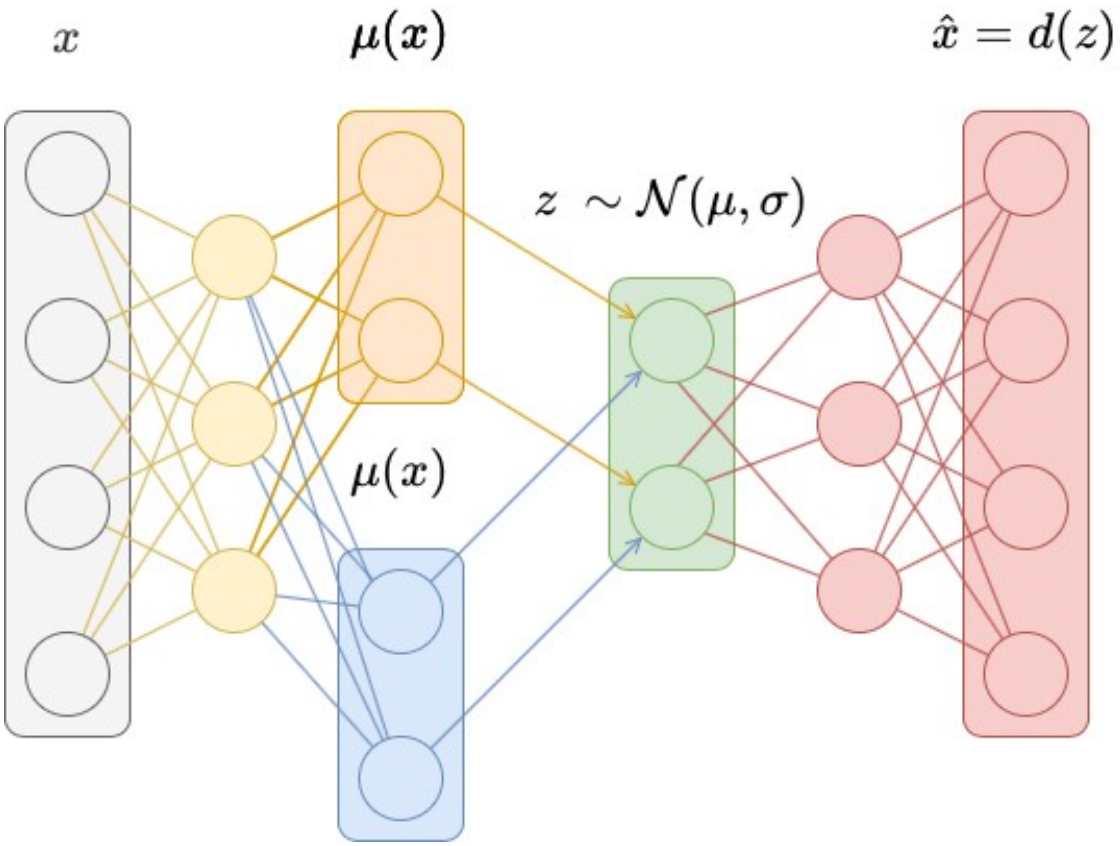
The graph shows that the author's autoencoder did not construct its latent variables in a regular fashion. To specify, many of the plotted latent variables formed clumps that branched out of the origin, and the origin is not the median of the latent variables. This means that for example, if we randomly give it a latent variable of (-5, 5), their autoencoder would not be able to create an output digit that resembles any real digit, as the autoencoder has never seen a latent variable at that location.

This behavior should be expected since the autoencoder had no reason to make its latent representation of data to be regular; the only thing the loss function penalized it for was to make the output similar to the input, so the autoencoder was only incentivized to encode and decode with as little loss between *x* and *xhat* as possible. Downsides of this is that it would be difficult to generate new data, since there are regions of unused latent space, and

that it can be prone to overfitting, since the autoencoder can just separate classes far from each other.

A variational autoencoder (VAE) is one possible solution to make the latent space more regularized. A VAE attempts to solve the problem by having a loss function that encourages regularization and has a latent space that is more friendly with the generation of new data. This is achieved by the following:

1. The input is encoded as a distribution over the latent space
2. A random point is sampled from the distribution
3. That sampled point is decoded and used for the loss function
4. Backpropagate that loss function

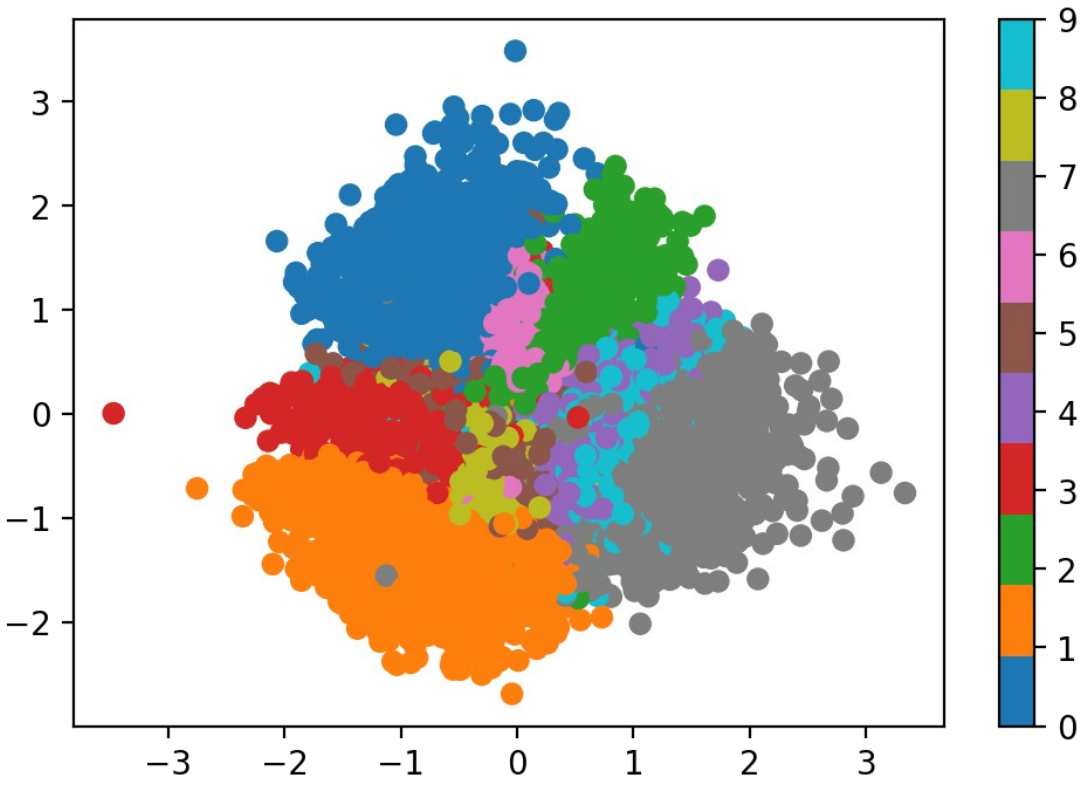


We represent the latent variable as a distribution instead of a point to make the latent space continuous. Since it is continuous, generating based on a random latent variable would make more sense since the latent space is based on probabilities, so being closer to a class's distribution means it is more likely to be a part of that class.

Our loss function also reflects this. Instead of relying solely on MSELoss, we now add a new metric called KL Divergence. MSELoss ensures local regularization by making sure our latent variable's variance stays low, since if it is high, a randomly sampled point would be far from the center and would be different from the actual input. KL Divergence measures how different two probability distributions are. We use the KL Divergence to find the difference between our latent space and the normal distribution. This incentivizes our VAE

to try to group the classes around the normal distribution, ensuring global regularization. This combination of a 'reconstruction term' and 'regularization term' makes our latent space group data points of the same class together, while also making sure the different groups are still close to the normal distribution.

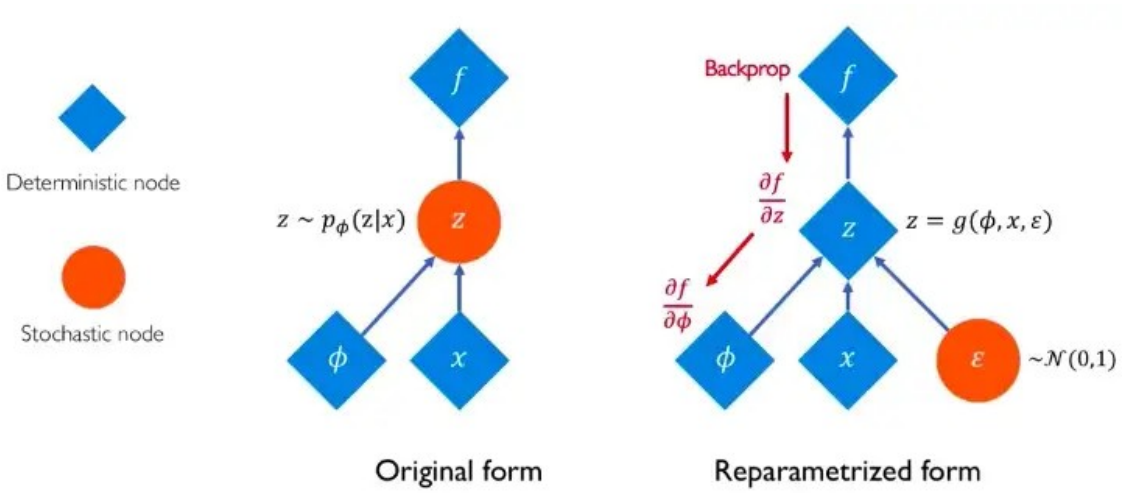
Below is another example from the same website as the previous image of a VAE implemented as above on the MNIST dataset.



An important aspect of VAEs to mention is the reparameterization trick. Normally we backpropagate loss through our model, but since we are randomly sampling our point, we can not backpropagate end-to-end since there is a stochastic node.

As shown in the diagram, we can solve this problem by introducing the stochastic variable epsilon. This means that instead of using a stochastic sample from our distribution, we use a deterministic sample based on a stochastic distribution. This allows us to still find the gradient of our latent variable since our latent variable is now deterministic with respect to epsilon, and epsilon is separate from the model. We can then use the gradient of the latent variable to calculate the gradient of the distribution and continue to train end-to-end.

(My way of understanding this is if we are playing darts, you can't find the gradient of the points I get, but you can find the gradient of the points function that is multiplied by my random throwing)



Model

Setup

Encoder, Decoder, and the Reparameterization trick

VAE Model

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

Our encoder and decoder use the same three convolutional layers as the autoencoder, but we now use two fully connected layers to learn how to best calculate the probability distribution of the input data.

The reparameterization trick lies in the reparameterize(mean, logvar) function.

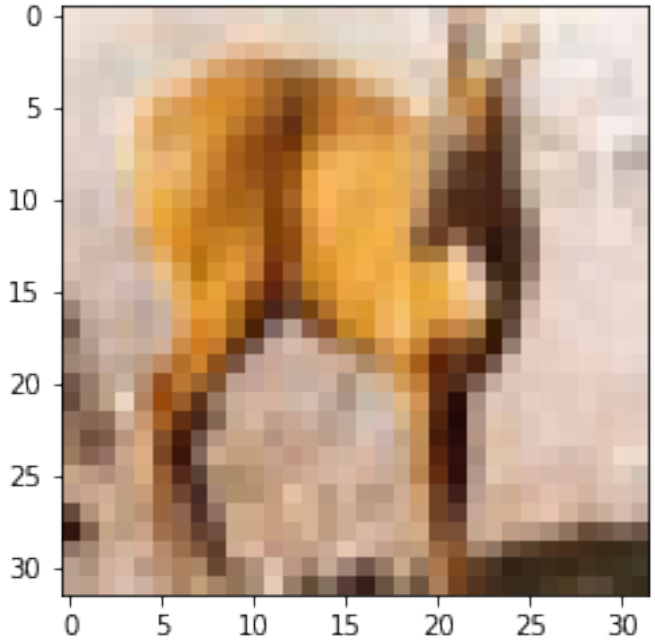
Hyperparameters

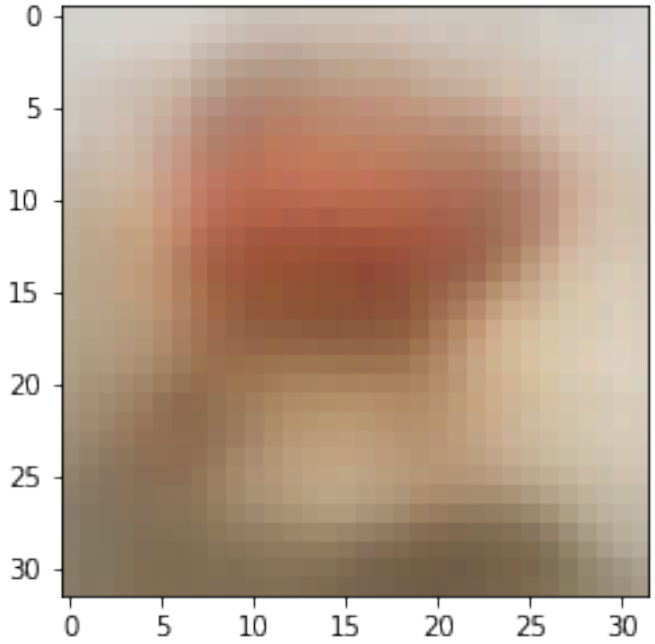
Run model

Result

```
(original, _) = next(iter(data_loader))
original = original[0].to(device)

x = original.cpu()

mu, logvar = model.encoder(original[None,:])
xhat = model.decoder(model.reparameterize(mu, logvar))[0].cpu()
print('Loss:', model.loss_func(x, xhat, mu, logvar).detach())

plt.imshow(np.transpose(x, (1,2,0))+0.5)
plt.show()
plt.imshow(np.transpose(xhat.detach().numpy(), (1,2,0))+0.5)
plt.show()
```

```
Loss: tensor(119.8970, device='cuda:0')
```





As shown in the picture above, the output image is significantly worse than the output image from the autoencoder. This could be because the VAE was not able to overfit like the autoencoder and so is less accurate, but the VAE is also more friendly towards the generation of new images. Because the latent space is continuous, the images will not be as sharp as the autoencoder, since every point would be in a probability field of other latent variables.

This continuous latent space would make generating novel images easier because any random latent vector given would be within the viable space of the model, since the model is incentivized by the KL Divergence to clump around the normal distribution. This means that random latent vector inputs would be reasonably nearby other latent vector's probabilities, so while the output would not be as precise as the outputs of the autoencoder, it would resemble an existing class even if the model has not seen the latent vector before.

Vector Quantized Variational Autoencoder (VQVAE)

VAEs are good for learning continuous features, but some tasks that we wish to use neural network models for are inherently not discrete. For example, training on language would not make sense for a continuous model or classification problems. For example, when describing words based on how close words are together, 'can' and 'car' are only 1 letter off from another, but in a language context they mean entirely different things. If used in a word vector approach, 'car' and 'truck' may have similar usages and so may have vectors that are close, but they still mean different things that may not make sense when the words are interchanged. Classification problems can also make use of discrete representations, as a model may get rain data and calculate '70% chance to rain' but a person would just think to get an umbrella.

A new (2018) model called the vector quantised-variational autoencoder (VQVAE) attempts to solve this problem by using vector quantization to train. Instead of depending on a normal distribution, the VQVAE uses quantized vectors from a 'codebook' dictionary.

The encoder encodes an input to produce *z_e*, but instead of representing that as a distribution, the model calculates the closest discrete latent variable *z* from a list of them called the 'codebook'. That closest codebook vector is then fed into the decoder.

1. The encoder converts the input to *z_e* through convolutional layers
2. *z_e* is used to calculate the nearest neighbor inside the embedding space codebook and encoded as a one hot vector
3. That nearest neighbor latent vector is put into the decoder which produces the output image.

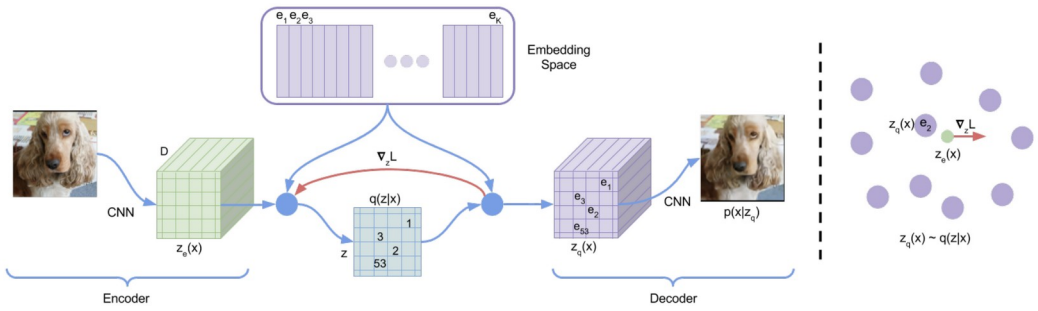


Figure 1: Left: A figure describing the VQ-VAE. Right: Visualisation of the embedding space. The output of the encoder $z(x)$ is mapped to the nearest point e_2. The gradient $\nabla_z L$ (in red) will push the encoder to change its output, which could alter the configuration in the next forward pass.

The parameters of the model to be optimized are the encoder, decoder, and the embedding space. We can calculate the gradients of the decoder and the embedding space, but the nearest neighbor search of the encoded vector to the embedded vector does not have a gradient. We get around this by directly passing the gradient of the decoder straight to the encoder. Since both the encoder and decoder use the same encoding space D (which is the size of each embedding vector), it contains some useful information for the encoder.

Our loss function has to reflect our new parameters in the model. It is separated into three parts, the reconstruction loss, embedding loss, and commitment loss.

1. The reconstruction loss is the same in the autoencoder and VAE, which is the difference between the input and output images. We want to minimize this to make our input and output as close as possible.
2. The embedding loss is calculated from the difference between the embedding vectors and the encoded latent vectors. We stop the gradients of the latent vectors because we only want to optimize the embedding vectors.
3. The commitment loss is used to prevent the embedding space from growing arbitrarily large. We force the encoder to commit to embeddings because if it does not, the encoder can create encodings that do not make sense with the existing embeddings, so the embedding space would have to continue to grow to match the encodings. We stop the gradients of the embedding vectors to force the encoder to commit to embeddings.

The decoder optimizes the first loss only, the encoder optimizes the first and third loss, and the embeddings optimize the middle loss.

$$L = \log p(x|z_q(x)) + \|\text{sg}[z_e(x)] - e\|_2^2 + \beta\|z_e(x) - \text{sg}[e]\|_2^2,$$

A prior can be trained from the discrete latents embeddings as a categorical distribution, and such can be made autoregressive by depending on past latent embeddings. While training the VQVAE, the prior should be kept constant and be based purely on the training inputs, but after training is completed, an autoregressive distribution can be used to generate novel data based on previous sampling. The authors of the VQVAE gave the example of PixelCNN and Wavenet, both being autoregressive models based on discrete data that can be fed embedding vectors and produce new embedding vectors that are then fed through the decoder into new data. However, since it is autoregressive and generally larger models, this project could not encompass them due to hardware limitations.

Model

Setup

Encoder, Decoder, Vector Quantizer, Residual Block

We use .detach() in the forward function to backpropagate the decoder's gradients to the encoder

VQVAE Model

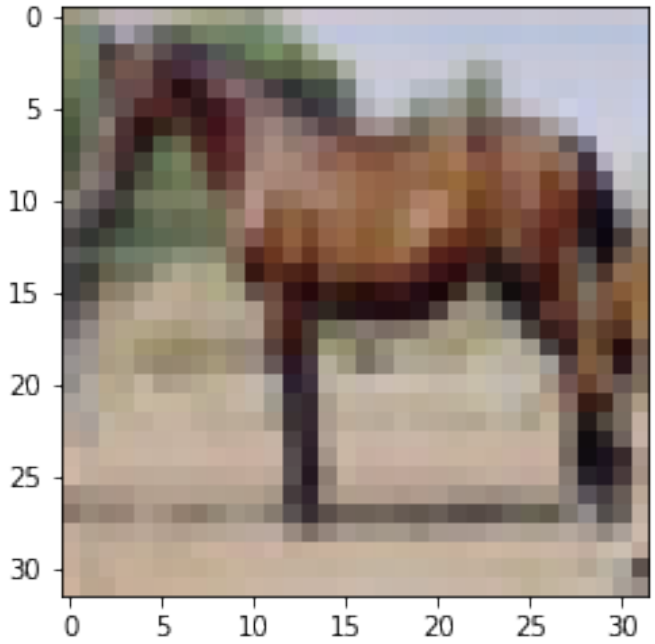
Hyperparameters

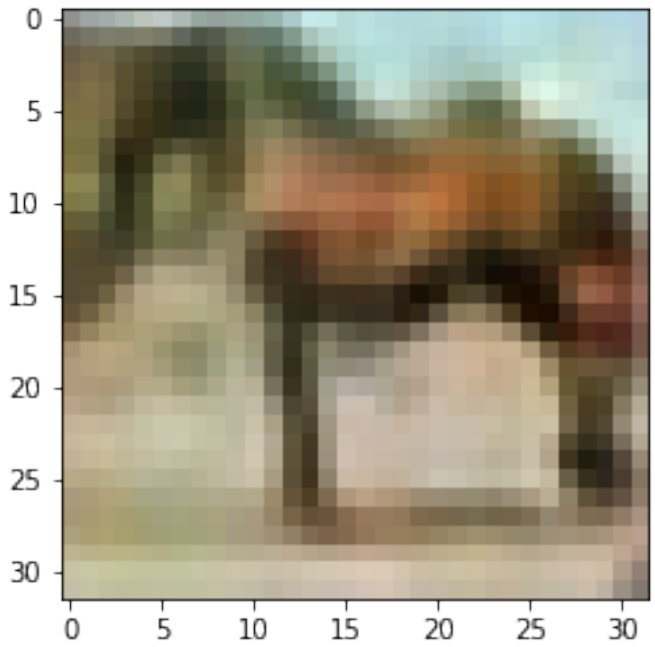
Run model

Result

```
(original, _) = next(iter(data_loader))
original = original[0].to(device)

x = original[None,:]

xhat, loss, _ = model(x)
print('Loss:', loss)

plt.imshow(np.transpose(x[0].cpu(), (1,2,0))+0.5)
plt.show()
plt.imshow(np.transpose(xhat[0].cpu().detach().numpy(), (1,2,0))+0.5)
plt.show()
```

```
Loss: tensor(0.0161, device='cuda:0', grad_fn=<AddBackward0>)
```



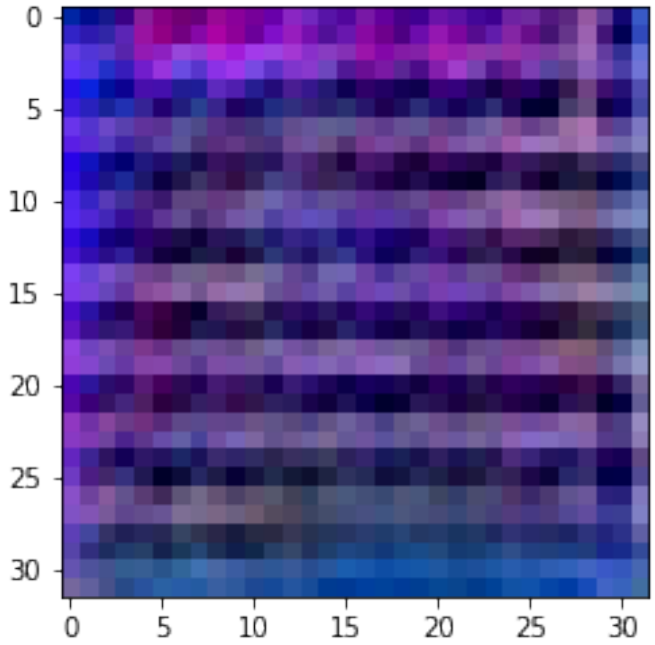


This one performed better than the VAE likely because it was allowed to use a large codebook inside of the model, which allowed the latent vector to not need much space since it is just the index of the already trained embedded vectors. It was also discrete, so it did not have to take into account stochastic probabilities when generating the output image.

While much larger and more complex, the basic concept of the VQVAE was a component in OpenAI's DALL-E, which was able to generate novel images based on input text data, which shows the potential of VQVAEs and VAEs in general.

```
(original, _) = next(iter(data_loader))
original = original[0].to(device)

x = original[None,:]
xx, _ = model.vq(model.pre_vq(model.encoder(x)))
new = torch.rand_like(xx)
#print(new)
xxx = model.decoder(new)

plt.imshow(np.transpose(xxx[0].cpu().detach().numpy(), (1,2,0))+0.5)
plt.show()
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for
imshow with RGB data ([0..1] for floats or [0..255] for integers).
```



This shows the result of inputting a completely random prior, which shows that we do need a proper model to generate priors to input into the model to get new data outpu