Generative AI: opdracht 1: On auto-encoders

Wilfred Van Casteren
nr:837377516
Open Universiteit
Masteropleiding AI

December 6, 2025

Introduction

This writing is mainly focused on auto-encoders. Auto-encoders are an interesting subject. Auto-encoders come in many variations, be it linear, non linear, variational, conditional and/or convolutional. This reports contains code, images, descriptions. I didn't want to be complete in any way, I just wanted to gain some insight in the wonderful world of auto-encoders. I was mainly inspired by [2].

I have added some code to this writing but adding everything wouldn't have been appropriate. I have added a Python notebook to get an idea of the way I handled some things. The conditional convolutional variational auto-encoders was the last addition, and maybe deserved some more attention, as its output was not really what I had expected.

As auto-encoders are partly compressing data, it may be interesting to consider a more mathematical way to compress data, before entering the world of auto-encoders. This mathematical way to compress data is principal component analysis (PCA). This method uses eigenvalues and eigenvectors to find dimensions of most variance.

Principal Component Analysis(PCA)

PCA?

PCA is a technique for reducing the dimensionality of multivariate data to only a few latent variables. It is a widely used method for identifying patterns and extracting underlying structure from complex data sets. PCA is in essence based on the transformation of the high-dimensional data into new dimensions called principal components (PCs) that are linear combinations of the original variables.

However, in PCA, some conditions are imposed on how these linear combinations are selected. The linear combinations are selected in decreasing order of variation and the combinations are required to be uncorrelated.

A process

A principal component analysis essentially is a process of rotating our original set of n axes, which correspond to the n variables we measured, until we find a new axis that explains as much of the total variance as possible. This becomes the first principal component axis. The data is then projected onto the n-1 dimensional surface that is perpendicular to this axis and repeat this process of rotation and projection until the original n axes are replaced with a new set of n principal component axes. PCA uses eigenvalues and eigenvectors to determine how to project the data. But what are these eigenvalues and -vectors?

Eigenvalues and eigenvectors

For a square matrix A, and every one of its Eigenvectors and corresponding Eigenvalues the following equation is true:

$$Av = \lambda v$$

$A =$ a matrix

$v =$ an Eigenvector

$\lambda =$ an Eigenvalue

Interpretation of Eigenvectors

For a square matrix A, when viewed as a transformation matrix, the product Av represents the result of applying that transformation to vector v. For most vectors, their direction and magnitude will change after transformation. However, eigenvectors are special: their direction remains unchanged after transformation; they are only scaled by their corresponding eigenvalue ($Av = \lambda v$). Essentially, Eigenvectors point in directions where the matrix acts simply as a scaling factor. This is a fundamental property of the matrix, regardless of the coordinate system. Therefore one

can say, eigenvectors represent the intrinsic or natural axes of the matrix, independent of the coordinate system in which the matrix is expressed. For a symmetric matrix, the eigenvectors have an extra important property: they are mutually orthogonal (perpendicular to each other) and form a complete orthonormal basis for the space.

Interpretation of Eigenvalues

For a symmetric matrix, each eigenvalue represents the scaling factor along its corresponding eigenvector, indicating that the transformation ($Av = \lambda v$) either stretches or compresses the orthonormal basis vector represented by that eigenvector.

Applying PCA

Having an idea of eigenvectors and eigenvalues makes it easier to grasp the concept of PCA. To apply PCA to a set of data the following steps should be followed. A publication that I found interesting on the subject is [5].

- Standardizing data by subtracting the mean and dividing by the standard deviation
- Calculate the Covariance matrix.
- Calculate eigenvalues and eigenvectors
- Merge the eigenvectors into a matrix and apply it to the data. This rotates and scales the data.
- Keep the new features which account for the most variation and discard the rest.

In PCA, eigenvectors identify the principal components (new axes) that capture the maximum variance in the data, while eigenvalues quantify the amount of variance each corresponding principal component explains. These eigenvalues, sorted from largest to smallest, determine the importance of each principal component, allowing for dimensionality reduction by keeping components with high eigenvalues and discarding those with low ones.

A simple example

Original data

The original data looks like this, the columns are the features:

$$A = \begin{bmatrix} 1 & 2 & 4 \\ 3 & 4 & 3 \\ 5 & 6 & 4 \end{bmatrix}$$

Centered data

The centered data is thne data minus the column means.

$$C = A - np.mean(A.T, axis = 1)$$

$$C = A - means = \begin{bmatrix} -2.0 & -2.0 & 0.333333333333333 \\ 0.0 & 0.0 & -0.666666666666667 \\ 2.0 & 2.0 & 0.333333333333333 \end{bmatrix}$$

Covariance matrix

The covariance matrix is a symmetric matrix. Because of this property the Eigenvectors of this matrix are perpendicular. For perpendicular Eigenvectors the dotproduct should be zero.

$$V = np.cov(C.T)$$

$$V = \begin{bmatrix} 4.0 & 4.0 & 0.0 \\ 4.0 & 4.0 & 0.0 \\ 0.0 & 0.0 & 0.333333333333333 \end{bmatrix}$$

Getting the Eigenvalues and Eigenvectors

To get the Eigenvalues and Eigenvectors for the covariance matrix.

$$eigenvalues, eigenvectors = np.linalg.eig(V)$$

Eigenvalues

The Covariance matrix's Eigenvalues with biggest values, be it positive or negative, will produce the Eigenvectors that will contribute most to data variance.

$$Eigenvalues = \begin{bmatrix} 8.0 \\ 0.0 \\ 0.333333333333333 \end{bmatrix}$$

Eigenvectors

In this case the Eigenvalue 8 corresponds to the first column of the Covariance Matrix's Eigenvectors matrix. This Eigenvector $[0.70, 0.70, 0]$ will be the first principal component.

$$Eigenvectors = \begin{bmatrix} 0.707106781186547 & -0.707106781186547 & 0.0 \\ 0.707106781186547 & 0.707106781186547 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

The principal components that make up the Eigenvectors' matrix are linear combinations of the original variables. In other words the values in the first principal component represent the weights by which features are combined to represent the virtual first dimension of the (reduced) data set.

Perpendicular Eigenvectors

When two vectors are perpendicular their dotproduct should be zero. When calculating the dotproduct for PCA1 $(0.70, 0.70, 0)$ and PCA2 $(0, 0, 1)$ it is 0.

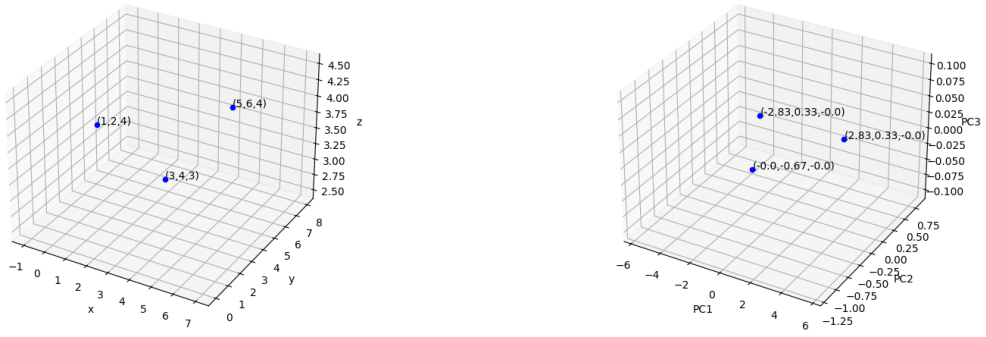
$$np.dot(eigenvectors[:, 0], eigenvectors[:, 2])$$

PCA projection

The projected data looks like below. As a complete transformation was done, this data has the same dimensions as the original data. As can be seen one dimension is completely superfluous, this dimension doesn't add any variance to the data. This transformed data is the product of the transposed eigenvectors matrix and the transposed centered data matrix.

$$\text{Projected data} = eigenvectors.T.dot(C.T)$$

$$\text{Projected data} = \begin{bmatrix} -2.82842712474619 & 0.0 & 0.333333333333333 \\ 0.0 & 0.0 & -0.666666666666667 \\ 2.82842712474619 & 0.0 & 0.333333333333333 \end{bmatrix}$$



(a) It is clear the data needs every dimension to be presented this way

(b) The transformed data can be shown using only dimensions PC1 and PC2

Figure 1: A scatterplot of the data before PCA (left) and after(right)

In Python, using numpy

The code to calculate every component by 'hand' using numpy, is to be found in Listing 1.

Listing 1: Transforming data using PCA

```
def pca(A):
 # the mean of each column
 M =np. mean(A.T, axis=1)
 # center columns by subtracting column means
 C = A - M
 # calculate covariance matrix of centered matrix
 V = np.cov(C.T)
 # get eigenvectors and values of covariance matrix
 eigenvalues, eigenvectors = np.linalg.eig(V)
 # project centered data using the eigenvectors
 PCA = eigenvectors.T.dot(C.T)
 return PCA
```

In Python, using PCA as defined by sklearn

The code to calculate PCA using the sklearn library, is to be found in Listing 2.

Listing 2: Transforming data using PCA

```
def sklearn_pca(A):
 M = np.mean(A.T, axis=1)
 # center columns by subtracting column means
 C = A - M
 pca = PCA(n_components=3)
 # get the transformed data points
 C_pca = pca.fit_transform(C)
 # get every principal component in the form of an Eigenvector
 pca1= pca.components_[0]
 pca2 = pca.components_[1]
 pca3 = pca.components_[2]
```

Auto-encoders

Idea

At a high level, auto-encoders are a type of artificial neural network used primarily for unsupervised learning. Their main goal is to learn a compressed, or "encoded," representation of data and then reconstruct the original data from that compressed version. Think of it like learning how to summarize a long article in just a few sentences:

the auto-encoder tries to find the most important information from the input and then recreates it as accurately as possible. The recreational part of the process could be seen as a generational process. For more information on auto-encoders see [3]

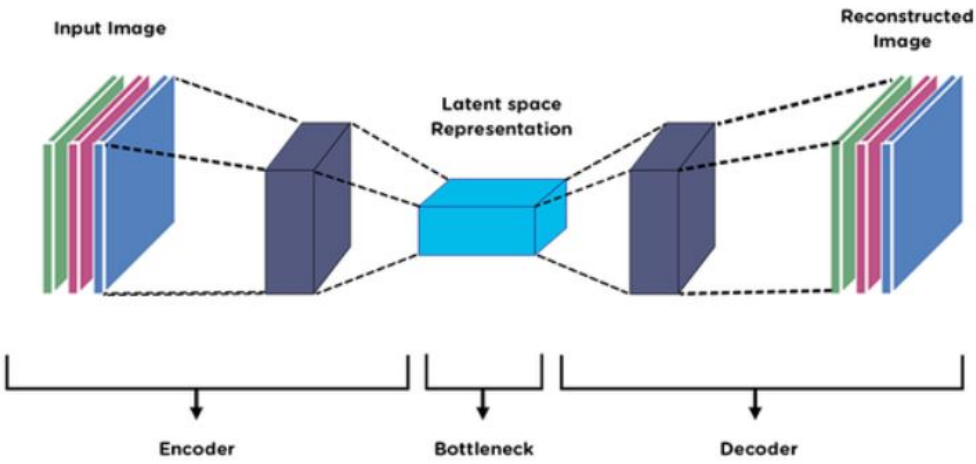


Figure 2: Overview of an auto-encoder.

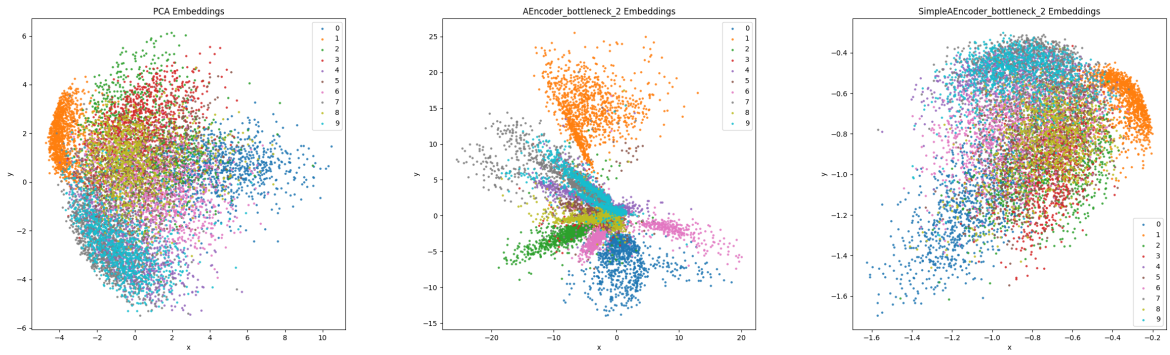
To encode and decode data a common auto-encoder has two components, an encoder and a decoder. The decoder reverses the operations done to encode the data. The most basic auto-encoder has an init-method that looks like this. The bottleneck_size represents the number of dimensions the encoded data will have. When choosing a bottleneck_size of 2, the image data will be compressed to 2 dimensions. These two dimensions should be sufficient for the model to be able to reconstruct the image.

$$self.encoder = nn.Sequential(nn.Linear(32 * 32, bottleneck\_size))$$

$$self.decoder = nn.Sequential(nn.Linear(bottleneck\_size, 32 * 32))$$

Comparing PCA and auto-encoder embedding

To show that the encoding part of a linear auto-encoder is comparable to what PCA does, see Figure 3, especially figure a and c. Complex data compressed by a non-linear auto-encoder is to be seen in the middle, it is clear the model responsible for encoding the data in this way does a much better job, as the clusters are more separable. When looking at figure c, the orange cluster for the MNIST images representing the number 1 is clearly separable from the clusters representing the other digits.



(a) The way PCA represents complex data in 2 dimensions

(b) The way a non-linear auto-encoder presents complex data in 2 dimensions

(c) The way a linear auto-encoder presents complex data in 2 dimensions

Figure 3: Complex data as represented by PCA (left), a non-linear auto-encoder (middle) and a linear auto-encoder (right)

Some reconstructed examples

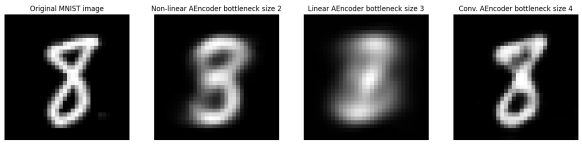


Figure 4: An original image (left), and the reconstructions generated by some auto-encoder models.

The examples in Figure 4 are the reconstructions they produced from the same original.

$$\_, reconstructed = model(image.unsqueeze(0)$$

Generating digits

An interesting idea is to take the median for every dimension of the encoded data, given the data was grouped by target and generate the digits per model.

Non-linear AEncoder

Figure 5: Decoding the non-linear auto-encoder's grouped median bottleneck-dimensions, given 4 such dimensions.

Convolutional AEncoder

Figure 6: Decoding the convolutional auto-encoder's grouped median bottleneck-dimensions, given 4 such dimensions.

Interpolating examples

As the 2 dimensional bottleneck auto-encoders only need 2 numbers to generate an image of a handwritten digit, it is possible to get a good idea of the generational qualities of the model. Take for instance the non-linear auto-encoder with a bottleneck of 2. It is fairly easy to generate images for 'every' value of x (between 0 and -10) and y (between -5 and 5).

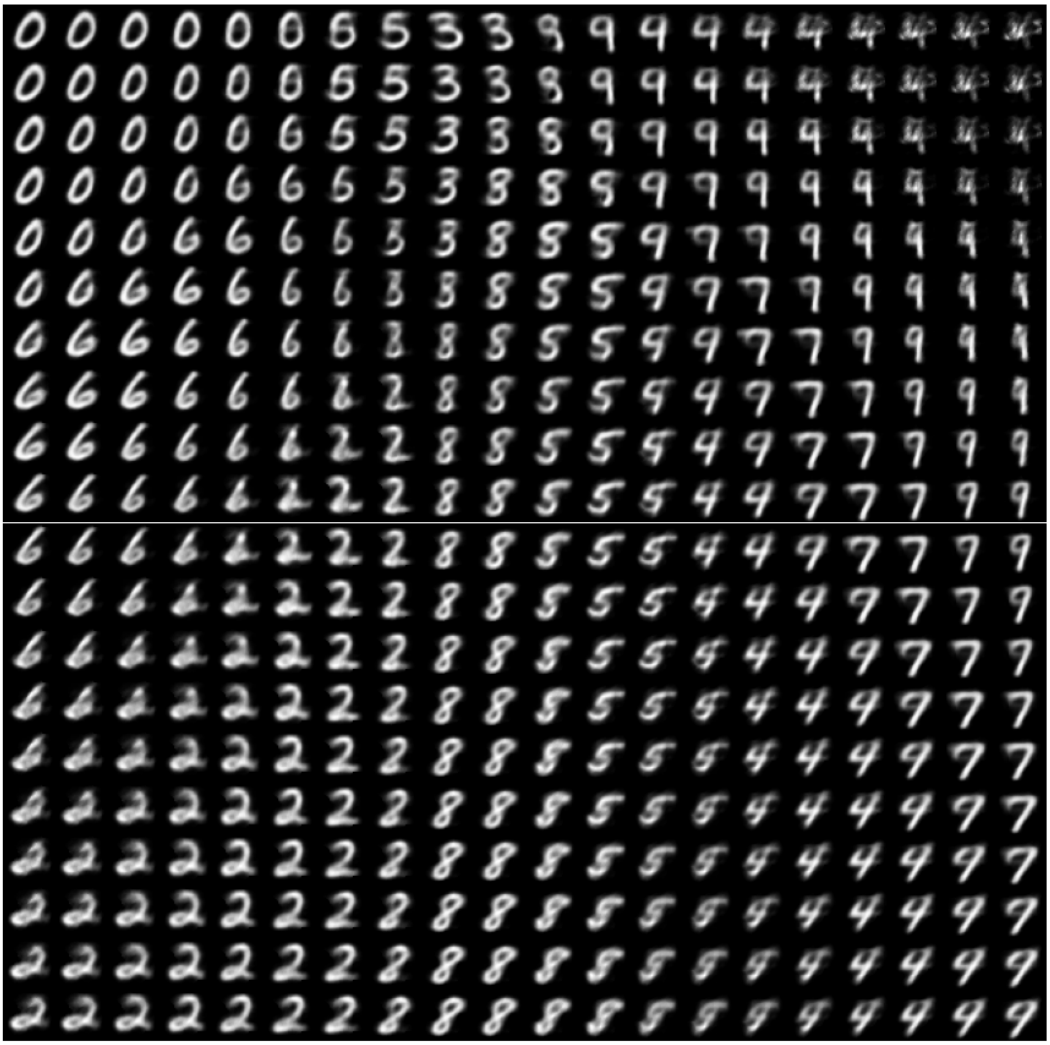


Figure 7: Interpolating images, compare with Figure 3b.

How does an auto-encoder get trained

To optimize the encoder to find the most promising encoder parameters a loss function is used. In this case the mean squared difference between the reconstructed image and the original image is the loss. Mean Squared Error (MSE) is a very common loss function used for training auto-encoders.

$$images = images.to(device)$$

$$encoded, reconstruction = model(images)$$

$$loss = torch.mean((images - reconstruction) * *2)$$

A simple experiment with auto-encoder models

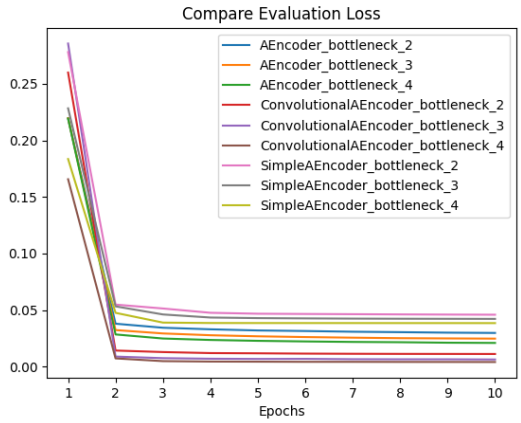
In this part trained auto-encoders' decoded data will be interpreted by a simple htr-model. The auto-encoders will be set up using different parameters for the bottleneck size, values 2 and 3 will be used. To get an idea of the how the auto-encoder models are set up, see **??**. The purpose of this experiment is to find out how well the models are at reconstructing data from the encoded data.

Decoded data?

The auto-encoders are trained for 10 iterations, and per iteration all sort of data is produced. Encoded data, decoded data, loss while evaluating, loss while training. This data is pickled to be used when needed. The data needed for this experiment is the decoded data, this data should give some indication of the generative qualities of the auto-encoders at hand.

Loss?

The loss should be correlated to the performance of the htr model, as loss is lower for an auto-encoder model, the decoded data should better interpreted by the htr-model.



(a) Evolution of loss while training the auto-encoders

model	accuracy
AEncoder bottleneck 2	78.11
AEncoder bottleneck 3	82.94
AEncoder bottleneck 4	86.42
Conv. AEncoder bottleneck 2	91.78
Conv. AEncoder bottleneck 3	95.37
Conv. AEncoder bottleneck 4	96.31
SimpleAEncoder bottleneck 2	36.86
SimpleAEncoder bottleneck 3	39.12
SimpleAEncoder bottleneck 4	48.33

(b) The auto-encoders and the accuracy achieved by the htr-model interpreting the reconstructed data (produced by the auto-encoders).

Figure 8: Auto-encoders compared on loss and decoded data accuracy

Variational auto-encoders

Variational Autoencoders (VAEs) are type of generative model in machine learning that create new data similar to the input they are trained on. They not only compress and reconstruct data like traditional autoencoders but also learn a continuous probabilistic representation of the underlying features. This unique approach helps VAEs to generate new, realistic data samples that closely resemble the original input. Consult [1] to get a more thorough idea of VAEs.

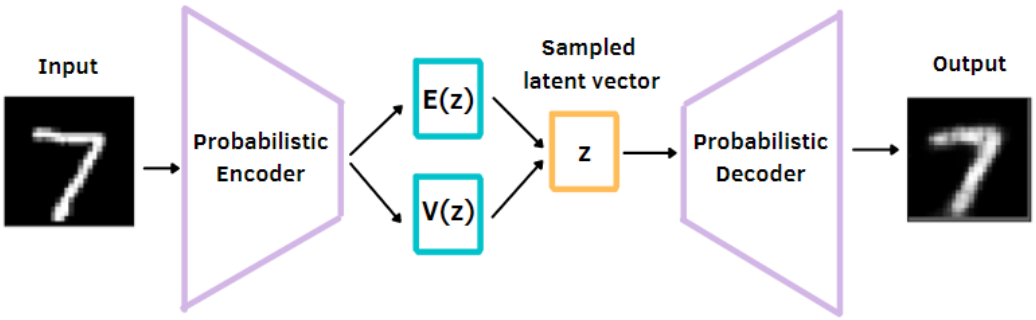


Figure 9: A general overview of the variational auto-encoder.

Encoder

The encoder takes input data like images or text and learns its key features. Instead of outputting one fixed value, it produces two vectors for each feature:

- Mean (μ): A central value representing the data.
- Standard Deviation (σ): It is a measure of how much the values can vary.

Latent space

Instead of encoding the input as one fixed point it pick a random point within the range given by the mean and standard deviation. This randomness lets the model create slightly different versions of data which is useful for generating new, realistic samples.

Decoder

The decoder takes the random sample from the latent space and tries to reconstruct the original input. Since the encoder gives a range, the decoder can produce new data that is similar but not identical to what it has seen.

Loss

The goal of VAE is not to reconstruct the original data but to generate a new sample that resembles the original data. The KL divergence regularization or KL loss term comes into play. It minimizes the KL divergence between the learned distribution of latent variables, Q and a simple normal distribution P, to force the learned latent variables to follow a normal distribution.

$$KL(Q_\theta(z|x_i)||P(z))$$

ELBO loss

The ELBO loss function is a critical part of training VAEs. It ensures that the latent space follows a structured distribution, enabling effective data reconstruction. The two main components of ELBO loss are:

- Reconstruction Term — Ensures the decoded output is similar to the original input.
- Regularization Term (KL Divergence) — Encourages the latent distribution ($P(z|x)$) to be close to a prior distribution ($P(z)$) , usually a Gaussian.

From [4] it is learned that

$$log(P(x) \geq \text{-}D_{KL}(Q_\theta(z|x_i)||P(z)) + \mathbb{E}_{\sim Q_\theta(z|x_i)}\left[logP_\phi(x_i|z)\right]$$

The right hand side of this equation is the Evidence Lower Bound (ELBO) also known as the variational lower bound. It is so termed because it bounds the likelihood of the data ($logP(x)$) which is the term we seek to maximize. Therefore maximizing the ELBO maximizes the log probability of our data by proxy. This is the core idea of variational inference, since maximization of the log probability directly is typically computationally intractable. The Kullback-Leibler term in the ELBO is a **regularizer** because it is a constraint on the form of the approximate posterior. The second term is called a **reconstruction term** because it is a measure of the likelihood of the reconstructed data output at the decoder.

Kullback-Leibler term when approximating the P(z) term by a normal distribution

The goal is to find a normal distribution $Q(z|x)$ (approximating $P(z)$) to generate data from. According to [4], the D_{KL} term can be converted into a tractable term using the probability density function for the normal distribution.

So when using

$$pdf = \frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The tractable KL term looks like the following:

$$D_{KL}(Q_\theta(z|x_i)||P(z)) = log\left(\frac{\sigma_Q}{\sigma_P}\right) - \frac{\sigma_p + (\mu_Q - \mu_P)^2}{2\sigma_P^2} + \frac{1}{2}$$

Using $\mu_P = 0$ and $\sigma_P = 1$, the formula becomes:

$$-D_{KL}(Q_\theta(z|x_i)||P(z)) = \frac{1}{2}\left[1 + log(\sigma_P^2) - \sigma_P^2 - \mu_P^2\right]$$

Reparametrization trick

The purpose of a VAE is to find an encoder which encodes the data into a normal distribution. From this distribution ($Z \sim \mathcal{N}(0,1)$) new values Z will be produced, These new values z, will be decoded into new images that will be very similar to our examples. But there is nothing to back-propagate over a random sample. As can be seen in Figure 10.

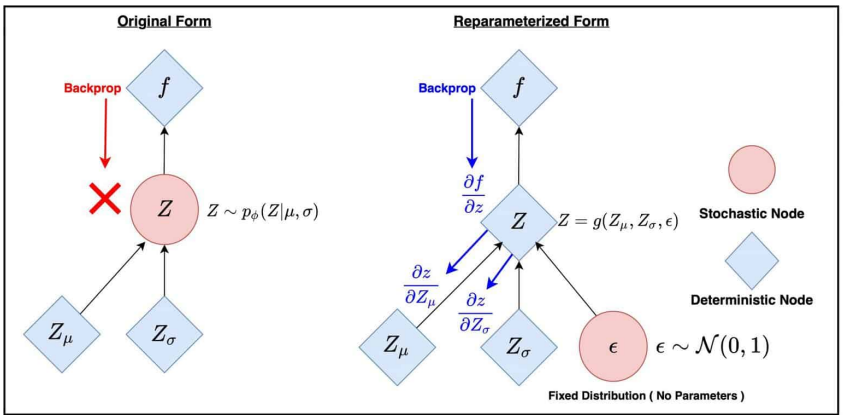


Figure 10: Back-propagating over a random value (left) compared to back-propagating over the reparameterized random value.

Illustration in Python

When executing the code snippet in Listing 3, the gradients for the tensors w1, w_mu and w_sd will be zero. This to illustrate the fact that the network will not learn anything from back-propagating over a random sampling process. The code itself simulates a simplified VAE's encoding process.

Listing 3: Backpropagating over a randomly sampled variable

```
input_tensor = torch.tensor([3.])
w1= torch.tensor([2.], requires_grad=True)
w_mu= torch.tensor([8.], requires_grad=True)
w_sd= torch.tensor([9.], requires_grad=True)

enc_out = input_tensor * w1
mu = w_mu*enc_out
std = w_sd * enc_out
print("mu:", mu)
print("std:", std)
z=torch.normal(mean=mu,std=std)
print("z:", z)
z.backward()

print("w_sd.grad:",w_sd.grad)
print("w1.grad:",w1.grad)
print("w_mu.grad:",w_mu.grad)
```

Rewriting the random sample process slightly

When rewriting the random sampling process slightly, by replacing line 11, using the code in Listing 4, the gradients will be tractable. This replacement works because when $z \sim \mathcal{N}(\mu, \sigma^2)$, z can be reparameterized by using $z = \mu + \sigma\epsilon$, while $\epsilon \sim \mathcal{N}(0,1)$

Listing 4: Backpropagating over a randomly sampled variable

```
e=torch.normal(mean=Tensor([0.0]),std=Tensor([1.0]))
z = sqrt(Tensor(std))*e + Tensor(mu)
```

Using this reparameterization trick, any Gaussian distribution can be simulated by shifting and scaling samples from a standard normal distribution.

A common linear variational auto-encoder

The complete code for this and other models is to be found in this section . For now, the calculation of the loss is important. The function that will be responsible for calculating the loss is to be seen in Listing 5. The KL weight and the reconstruction weight are values that reflect how much weight is to be given to the KL loss and the Reconstruction loss.

Listing 5: Variational loss function

```
def VAELoss(x,x_hat, mean, log_var, kl_weigth=1, reconstruction_weight=1):
 pixel_mse = ((x-x_hat)**2)
 pixel_mse=pixel_mse.flatten(1)
 reconstruction_loss= pixel_mse.sum(axis=1).mean()

 kl= (1+log_var - mean**2 - torch.exp(log_var))
 kl_per_image = -0.5 * torch.sum(kl, dim=-1)
 kl_loss= torch.mean(kl_per_image)

 return (reconstruction_loss*reconstruction_weight) + (kl_loss*kl_weigth)
```

Getting μ and σ

The forward_enc method returns the latent values, and for calculationg loss, mu and logvar. Logvar is a tensor having the bottleneck's dimension as its size, and converting it to a tensor of standard deviations is done by

$$sd = e^{0.5*logvar}$$

This formula derives from

$$logvar = log(\sigma^2)$$
$$log(\sigma^2) = 2.log(\sigma)$$
$$\frac{log(\sigma^2)}{2} = log(\sigma)$$
$$\sigma = e^{\frac{log(\sigma^2)}{2}}$$

Listing 6: The forward encoding function

```
def forward_enc(self,x):
 x=self.encoder(x)
 mu=self.fn_mu(x)
 logvar=self.fn_logvar(x)
 sigma = torch.exp(0.5*logvar)
 noise = torch.randn_like(sigma)
 z=mu+(sigma*noise)
 return z,mu,logvar
```

Compare the encoding for some models

In Figure 11 the visual representations for the encoding produced by some models is seen. When overemphasizing KL loss, the encoding becomes very normally distributed around zero, it is impossible to reproduce images from this representation. When emphasizing KL loss and reconstruction loss equally, encoded data is still decodable. Of course in this case the bottleneck dimension is two.

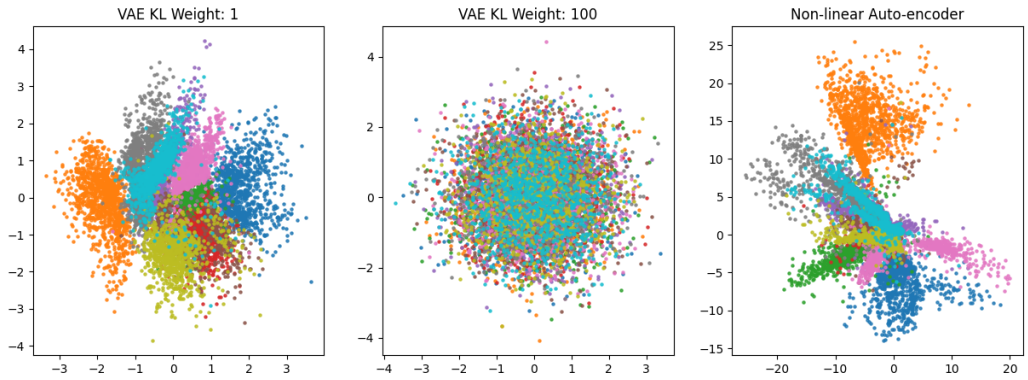
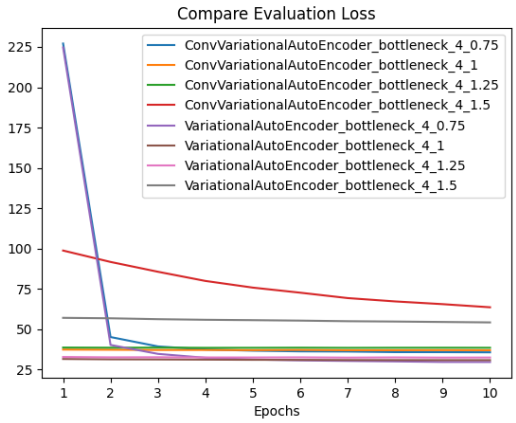


Figure 11: Compare encoding for VAE model when trained given equal weight to KL and reconstruction loss (left), when given almost all weight to KL loss, and the encoding produced by a non-linear non-variational auto-encoder model.

Training the variational encoders and testing the reconstructed data

Bottleneck dimension 4

Comparing the loss when training the variational models and interpreting the reconstructed images using a simple htr model, leads to Figure 12.



(a) Evolution of loss while training the variational auto-encoders

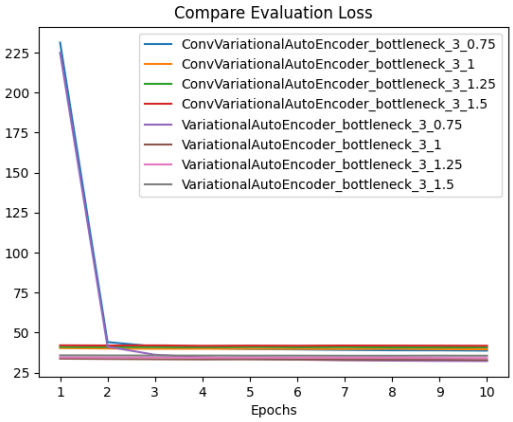
model	accuracy
ConvVariationalAutoEncoder bottleneck 4 0.75	71.19
ConvVariationalAutoEncoder bottleneck 4 1	71.07
ConvVariationalAutoEncoder bottleneck 4 1.25	71.44
ConvVariationalAutoEncoder bottleneck 4 1.5	22.80
VariationalAutoEncoder bottleneck 4 0.75	83.52
VariationalAutoEncoder bottleneck 4 1	84.98
VariationalAutoEncoder bottleneck 4 1.25	84.99
VariationalAutoEncoder bottleneck 4 1.5	25.10

(b) The variational auto-encoders and the accuracy achieved by the htr-model interpreting the reconstructed data (produced by the auto-encoders).

Figure 12: Variational auto-encoders compared on loss and decoded data accuracy for bottleneck dimension 4, considering multiple KL loss weights

Bottleneck dimension 3

Comparing the loss when training the variational models and interpreting the reconstructed images using a simple htr model, leads to Figure 13.



(a) Evolution of loss while training the variational auto-encoders

model	accuracy
ConvVariationalAutoEncoder bottleneck 3 0.75	62.25
ConvVariationalAutoEncoder bottleneck 3 1	63.86
ConvVariationalAutoEncoder bottleneck 3 1.25	64.02
ConvVariationalAutoEncoder bottleneck 3 1.5	63.62
VariationalAutoEncoder bottleneck 3 0.75	78.46
VariationalAutoEncoder bottleneck 3 1	84.32
VariationalAutoEncoder bottleneck 3 1.25	84.54
VariationalAutoEncoder bottleneck 3 1.5	85.39

(b) The variational auto-encoders and the accuracy achieved by the htr-model interpreting the reconstructed data (produced by the auto-encoders).

Figure 13: Variational auto-encoders compared on loss and decoded data accuracy for bottleneck dimension 3, considering multiple KL loss weights

Generating some ”handwritten” digits

For every model 10 random digits are generated. In this section I will show the most promising versions, in section all generated digits for all trained models are shown.

To generate random digits from a variational encoder a simple method (see Listing 7) can be used. The number parameter is used to set the quantity.

Listing 7: A generate method for an auto-encoder

```
def generate(self, number=1):
 z=torch.normal(0,1, size=(number, self.bottleneck))
 generated=self.decoder(z)
 print("generated.shape")
 print(generated.shape)
 generated = generated.reshape(-1, 1, 32, 32)
 return generated
```

Convolutional variational auto-encoder,bottleneck=3, KL=1.25

Figure 14: Generating random digits.

Variational auto-encoder, bottleneck=3, KL=1.25

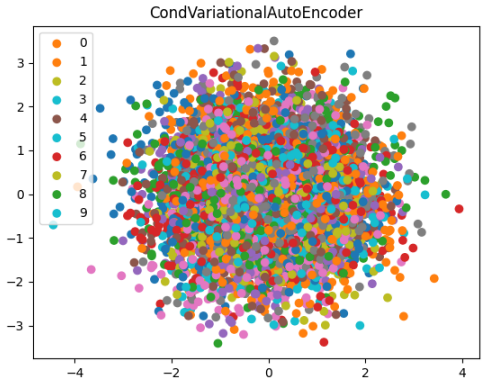
Figure 15: Generating random digits.

Conditional variational auto-encoders

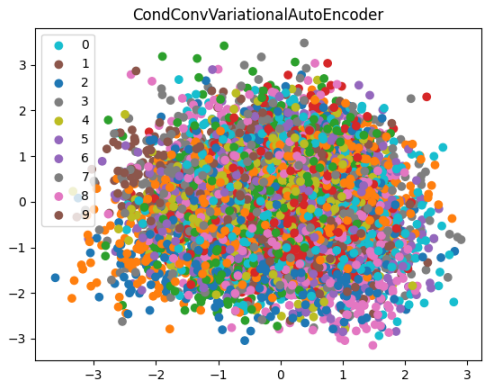
When generating data, be it images of digits, or other objects like sound or text, it is always nice to be able to generate specific data. For instance, I want to be able to generate the digit '9'. For this case there is a conditional VAE. The code for a plain and a convolutional conditional VAE is to be found in section .

The conditional signal

I believe it is possible to only add a conditioning signal to the decoder, but in this case the signal is added to both the encoder and the decoder. The first thing which caught my eye when reconstructing an image was its quality. But first things first, by adding a signal to the encoder, the encoded data is data that centers around 0 having a standard deviation of 1. At first sight the data isn't separable. Plotting the encoded data looks like Figure 16a. But of course by adding a conditioning signal, it is not necessary to make visually separable clusters, as the conditional signal itself is used when regenerating an image. In this section I didn't really experiment with KL weight values or bottleneck dimension values. Even though I have most of the code to do some more experiments , e.g. code for playing around with bottleneck dimensions. Including everything in this report leads to a lot of extra work.



(a) The latent representation for the conditional VAE.

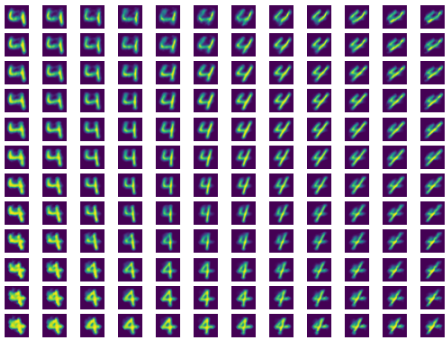


(b) The latent representation for the conditional convolutional VAE.

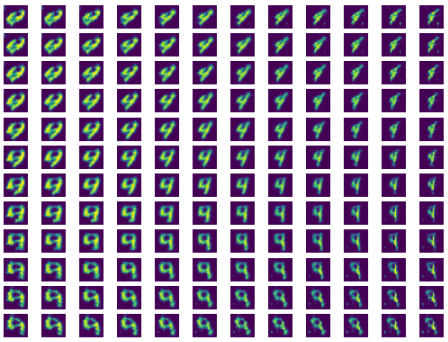
Figure 16: The latent representation for the conditional VAE and conditional and convolutional VAE.

Generating some examples.

When interpolating over the space for the basic conditional VAE, figures like Figure 17a were no exception. While, interpolating over the space for the conditional convolutional VAE, generated samples looking less sharp. See for instance Figure 17b. It feels as though the convolutions add to much complexity, maybe adding



(a) The basic conditional auto-encoder generating images for the number '4'.



(b) The conditional convolutional auto-encoder generating images for the number '4'.

Figure 17: Generating interpolations for the conditional (convolutional) VAE.

Code to generate intrapolation

The code to generate images over two dimensions is to be seen in Listing 8. As I was interested in the evolution over training iterations, and saved generated intrapolations, I needed to add epoch as an argument. If adding an extra bottleneck dimension this would lead to an excessive amount of generated images, so sticking to 2 bottleneck dimensions.

Listing 8: Method for generating images over 2 dimensions

```
def generate_interpolation(self,kl,bs, epoch,number=1):
 name = str(self.__class__.__name__)
 r0 = (-3, 3)
 r1 = (-3, 3)
 n = 12

 fig, axs = plt.subplots(n, n)

 for i, a in enumerate(np.linspace(*r1, n)):
  for j, b in enumerate(np.linspace(*r0, n)):
  z = torch.Tensor([[a, b]])
  y = torch.nn.functional.one_hot(torch.tensor([number]),
  num_classes=10)
  x_hat = self.forward_dec(z, y)
  x_hat = x_hat.reshape(32, 32).detach().cpu().numpy()
  axs[i, j].imshow(x_hat)
  axs[i, j].axis('off')
 plt.savefig("graph/" + name +"_bottleneck_"+str(bs)+"_"+str(kl)+ "/" + "ep_" + str(epoch) +
    ↪ "_" + str(number))
 plt.clf()
```

Bibliography

[1] Diederik P. Kingma and Max Welling. An introduction to variational autoencoders. *CoRR*, abs/1906.02691, 2019.

[2] Priyam Mazumdar. Intro to autoencoders: Compression is everything! https://www.youtube.com/@ExploratoryDataAdventures, 2023.

[3] Umberto Michelucci. An introduction to autoencoders. *CoRR*, abs/2201.03898, 2022.

[4] Stephen G. Odaibo. Tutorial: Deriving the standard variational autoencoder (vae) loss function. *ArXiv*, abs/1907.08956, 2019.

[5] Lindsay I Smith. A tutorial on principal components analysis. Technical report, 2002.

Appendices

Calculating Eigenvalues and Eigenvectors: a simple example

To find the Eigenvalues and Eigenvectors from a simple matrix A

$$\begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}$$

Eigenvalues

One can use the formula $|Av\text{-}\lambda I| = 0$.

$A\text{-}\lambda I$ can be calculated as follows

$$\begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} - \begin{bmatrix} 1.0\lambda & 0 \\ 0 & 1.0\lambda \end{bmatrix} = \begin{bmatrix} -1.0\lambda & 1 \\ -2 & -1.0\lambda - 3 \end{bmatrix}$$

The determinant for $A\text{-}\lambda I$ is
$1.0\lambda^2 + 3.0\lambda + 2 = 0$
having 2 solutions for λ
$[-2.0,\ -1.0]$

Using Python

Listing 9: Calculating Eigenvalues in Sympy

```
A=array([[0,1],[-2,-3]])
id = np.identity(A.ndim)
lambda = sympy.Symbol('lambda')
id_lambda = sympy.Matrix(id * lambda)
A_lambda_i = sympy.Matrix(A) - id_lambda
eigenvalues = solve(A_lambda_i.det(), lambda)
```

Eigenvectors

As there are two Eigenvalues, two Eigenvectors can be found. The Eigenvalues are used to calculate Eigenvectors.

$|A\text{-}\lambda I|.v = 0$ looks like this

$$\begin{bmatrix} -1.0\lambda v_1 + v_2 \\ -2v_1 + v_2\,(-1.0\lambda - 3) \end{bmatrix} = 0$$

After substitution of $\lambda = -2$ the equation looks like below:

$$\begin{bmatrix} 2.0v_1 + v_2 \\ -2v_1 - 1.0v_2 \end{bmatrix} = 0$$

The solution for this equation is like below:

$$\begin{bmatrix} 1 \\ -2.0 \end{bmatrix}$$

This solution is called the Eigenvector, but 2 and -4 could just as well have been a solution. This Eigenvalue corresponds to the Eigenvalue $\lambda = -2$

The other λ value -1 can be used to find another Eigenvector:

$$\begin{bmatrix} 1 \\ -1.0 \end{bmatrix}$$

Using Python

Listing 10: Calculating Eigenvectors in Sympy

```
v1 = sympy.Symbol('v1')
v2 = sympy.Symbol('v2')
lambda = sympy.Symbol('lambda')
concrete_solutions=[]
for ev in eigenvalues:
 eig=A_lambda_i*sympy.Matrix(array([v1,v2]))
 eig=eig.subs(lambda, ev)
 eig=eig.subs(v1,1)
 solution_eig=solve(eig, v1,v2)
 concrete_solutions.append([1,solution_eig[v2]])
```

Auto-encoder models

The auto-encoder models used in this writing all have the same architecture. Every model has an encoder and a decoder part. To get an idea of what the auto-encoders, used in this writing, look like, I have created some model torchinfo summaries.

Encoders and decoders

The encoding and the decoding parts of the auto-encoders presented here are merely an inversion of their respective encoding counterparts.

AEncoder

Encoder and decoder

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
AEncoder                                 [4, 2]                    --
├─Sequential: 1-1                        [4, 2]                    --
│    └─Linear: 2-1                       [4, 128]                  131,200
│    └─ReLU: 2-2                         [4, 128]                  --
│    └─Linear: 2-3                       [4, 64]                   8,256
│    └─ReLU: 2-4                         [4, 64]                   --
│    └─Linear: 2-5                       [4, 32]                   2,080
│    └─ReLU: 2-6                         [4, 32]                   --
│    └─Linear: 2-7                       [4, 2]                    66
├─Sequential: 1-2                        [4, 1024]                 --
│    └─Linear: 2-8                       [4, 32]                   96
│    └─ReLU: 2-9                         [4, 32]                   --
│    └─Linear: 2-10                      [4, 64]                   2,112
│    └─ReLU: 2-11                        [4, 64]                   --
│    └─Linear: 2-12                      [4, 128]                  8,320
│    └─ReLU: 2-13                        [4, 128]                  --
│    └─Linear: 2-14                      [4, 1024]                 132,096
│    └─Sigmoid: 2-15                     [4, 1024]                 --
==========================================================================================
Total params: 284,226
Trainable params: 284,226
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 1.14
==========================================================================================
Input size (MB): 0.02
Forward/backward pass size (MB): 0.05
Params size (MB): 1.14
Estimated Total Size (MB): 1.20
==========================================================================================
```

Figure 18: The layers of the auto-encoder.

Simple AEncoder

Encoder and decoder

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleAEncoder                           [4, 2]                    --
├─Sequential: 1-1                        [4, 2]                    --
│    └─Linear: 2-1                       [4, 2]                    2,050
├─Sequential: 1-2                        [4, 1024]                 --
│    └─Linear: 2-2                       [4, 1024]                 3,072
==========================================================================================
Total params: 5,122
Trainable params: 5,122
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.02
==========================================================================================
Input size (MB): 0.02
Forward/backward pass size (MB): 0.03
Params size (MB): 0.02
Estimated Total Size (MB): 0.07
==========================================================================================
```

Figure 19: The layers of the simmple auto-encoder.

Convolutional AEncoder

Encoder and decoder

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ConvolutionalAEncoder                    [4, 2, 4, 4]              --
├─Sequential: 1-1                        [4, 2, 4, 4]              --
│    └─Conv2d: 2-1                       [4, 8, 16, 16]            72
│    └─BatchNorm2d: 2-2                  [4, 8, 16, 16]            16
│    └─ReLU: 2-3                         [4, 8, 16, 16]            --
│    └─Conv2d: 2-4                       [4, 16, 8, 8]             1,152
│    └─BatchNorm2d: 2-5                  [4, 16, 8, 8]             32
│    └─ReLU: 2-6                         [4, 16, 8, 8]             --
│    └─Conv2d: 2-7                       [4, 2, 4, 4]              290
├─Sequential: 1-2                        [4, 1, 32, 32]            --
│    └─ConvTranspose2d: 2-8              [4, 16, 8, 8]             304
│    └─BatchNorm2d: 2-9                  [4, 16, 8, 8]             32
│    └─ReLU: 2-10                        [4, 16, 8, 8]             --
│    └─ConvTranspose2d: 2-11             [4, 8, 16, 16]            1,160
│    └─BatchNorm2d: 2-12                 [4, 8, 16, 16]            16
│    └─ReLU: 2-13                        [4, 8, 16, 16]            --
│    └─ConvTranspose2d: 2-14             [4, 1, 32, 32]            73
│    └─Sigmoid: 2-15                     [4, 1, 32, 32]            --
==========================================================================================
Total params: 3,147
Trainable params: 3,147
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 1.95
==========================================================================================
Input size (MB): 0.02
Forward/backward pass size (MB): 0.43
Params size (MB): 0.01
Estimated Total Size (MB): 0.46
==========================================================================================
```

Figure 20: The layers of the convolutional auto-encoder.

Forward

The forward method of these common auto-encoder models looks more or less like in Listing 11. It is in this forward method where the encoded and decoded data is produced and returned.

Listing 11: The forward method for a common auto-encoder

```
1 def forward(self, x):
```

```
batch, channels, height, width = x.shape
x = x.flatten(1)
enc = self.encoder(x)
dec = self.decoder(enc)
dec = dec.reshape(batch, channels, height, width)
return enc, dec
```

Variational auto-encoder models

A common variational auto-encoder

A listing for a common variational auto-encoder is to be seen in Listing 12. The outputs the encoder produces are used to learn a mu and a logvar for every bottleneck dimension. The mu and the logvar are used to generate a value from a normal distribution. For every bottleneck dimension a different mu and logvar value is learned. As the convolutional model is more accurate at generating, I will add more concrete explanations on the model.

Listing 12: A common variational auto-encoder

```
class VariationalAutoEncoder(nn.Module):
 def __init__(self, latent_dim=4):
  super().__init__()
  self.encoder = nn.Sequential(
  nn.Linear(32*32, 128),
  nn.ReLU(),
  nn.Linear(128, 64),
  nn.ReLU(),
  nn.Linear(64, 32),
  )
  self.fn_mu=nn.Linear(32, latent_dim)
  self.fn_logvar=nn.Linear(32, latent_dim)
  self.decoder = nn.Sequential(
  nn.Linear(latent_dim,32),
  nn.ReLU(),
  nn.Linear(32, 64),
  nn.ReLU(),
  nn.Linear(64, 128),
  nn.ReLU(),
  nn.Linear(128, 32*32),
  nn.Sigmoid(),
  )
 def forward\_enc(self,x):
  x=self.encoder(x)
  mu=self.fn_mu(x)
  logvar=self.fn_logvar(x)
  sigma = torch.exp(0.5*logvar)
  noise = torch.randn_like(sigma)
  z=mu+(sigma*noise)
  return z,mu,logvar

 def forward_dec(self,x):
  return self.decoder(x)

 def forward(self, x):
  batch_size, channels, height, width = x.shape
  x=x.flatten(1)
  z,mu,logvar= self.forward\_enc(x)
  dec = self.decoder(z)
  dec=dec.reshape(batch_size,channels,height,width)
  return z,dec,mu,logvar
```

A convolutional variational auto-encoder

A listing for a common linear variational auto-encoder is to be seen in Listing 13. Just like in the linear auto-encoder, outputs of the encoder are used to learn mu and logvar values. In this case the generated encoding is flattened in the custom last layer (line 13), the flattened data is used to generate the 4 channeled data to produce

mu (line 15) and logvar (line 16). The 4 generated values from the normal distribution are expanded back to a pre-flattened size (the size after applying line 12), through applying a linear layer (line 19) and a custom unflatten layer (line 20). As of line 21 the decoder is just like in the non-variational auto-encoder. The flatten and unflatten layers are nn.Module components that adapt the tensor views in the Forward method. As it is interesting to use encoder and decoder layer separately, they are made callable. In the forward_enc method the z encoding is produced from the logvar and mu values (lines 35-39).

Listing 13: A convolutional variational auto-encoder

```
1  class ConvVariationalAutoEncoder(nn.Module):
2   def __init__(self, in_channels=1, channels_bottleneck=4):
3    super().__init__()
4    self.bottleneck = channels_bottleneck
5    self.encoder = nn.Sequential(
6    nn.Conv2d(in_channels=in_channels, out_channels=8, kernel_size=3, stride=2, padding=1, bias
       ↪ =False),
7    nn.BatchNorm2d(8),
8    nn.ReLU(),
9    nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=1, bias=False),
10   nn.BatchNorm2d(16),
11   nn.ReLU(),
12   nn.Conv2d(in_channels=16, out_channels=channels_bottleneck, kernel_size=3, stride=2,
       ↪ padding=1),
13   Flatten()
14   )
15   self.fn_mu = nn.Linear(channels_bottleneck*4*4, channels_bottleneck)
16   self.fn_logvar = nn.Linear(channels_bottleneck*4*4, channels_bottleneck)
17
18   self.decoder = nn.Sequential(
19   nn.Linear(channels_bottleneck, 4 * 4 * channels_bottleneck),
20   UnFlatten(),
21   nn.ConvTranspose2d(in_channels=channels_bottleneck, out_channels=16, kernel_size=3, stride
       ↪ =2, padding=1,
22   output_padding=1),
23   nn.BatchNorm2d(16),
24   nn.ReLU(),
25   nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=3, stride=2, padding=1,
26   output_padding=1),
27   nn.BatchNorm2d(8),
28   nn.ReLU(),
29   nn.ConvTranspose2d(in_channels=8, out_channels=in_channels, kernel_size=3, stride=2,
       ↪ padding=1,
30   output_padding=1),
31   nn.Sigmoid()
32   )
33  def forward_enc(self, x):
34   x = self.encoder(x)
35   mu = self.fn_mu(x)
36   logvar = self.fn_logvar(x)
37   sigma = torch.exp(0.5 * logvar)
38   noise = torch.randn_like(sigma)
39   z = mu + (sigma * noise)
40   return z, mu, logvar
41
42  def forward_dec(self, x):
43   return self.decoder(x)
44
45  def forward(self, x):
46   z, mu, logvar = self.forward_enc(x)
47   dec = self.decoder(z)
48   return z, dec, mu, logvar
```

Conditional variational auto-encoder

In Listing 14 the code for a conditional variational auto-encoder is to be found. For the decoder and the encoder, a conditional signal is first reserved (line 6 and 16), and then added to the input (see line 50 and 52). The labels are one hot encoding tensors. The generate method generates an image tensor.

Listing 14: A conditional variational auto-encoder

```
class CondVariationalAutoEncoder(nn.Module):
 def __init__(self, latent_dim=2, number_of_classes=10):
  super().__init__()
  self.bottleneck=latent_dim
  self.encoder = nn.Sequential(
  nn.Linear(32*32+number_of_classes, 128),
  nn.ReLU(),
  nn.Linear(128, 64),
  nn.ReLU(),
  nn.Linear(64, 32),
  )
  self.fn_mu=nn.Linear(32, latent_dim)
  self.fn_logvar=nn.Linear(32, latent_dim)

  self.decoder = nn.Sequential(
  nn.Linear(latent_dim+number_of_classes,32),
  nn.ReLU(),
  nn.Linear(32, 64),
  nn.ReLU(),
  nn.Linear(64, 128),
  nn.ReLU(),
  nn.Linear(128, 32*32),
  nn.Sigmoid(),
  )
 def forward_enc(self,x):
  x=self.encoder(x)
  mu=self.fn_mu(x)
  logvar=self.fn_logvar(x)
  sigma = torch.exp(0.5*logvar)
  noise = torch.randn_like(sigma)
  z=mu+(sigma*noise)
  return z,mu,logvar

 def forward_dec(self,z,labels):
  z = torch.cat((z, labels), dim=1)
  return self.decoder(z)

 def generate(self, number=1):
  torch.manual_seed(0)
  z=torch.normal(0,1, size=(number, self.bottleneck))
  generated=self.decoder(z)
  generated = generated.reshape(-1, 1, 32, 32)
  return generated

 def forward(self, x, labels):
  batch_size, channels, height, width = x.shape
  x=x.flatten(1)
  x=torch.cat((x, labels), dim=1)
  z,mu,logvar= self.forward_enc(x)
  z=torch.cat((z, labels), dim=1)
  dec = self.decoder(z)
  dec=dec.reshape(batch_size,channels,height,width)
  flat_labels = torch.argmax(labels, dim=1)
  return z,dec,mu,logvar, flat_labels
```

Conditional variational auto-encoder

In Listing 15 the code for a conditional convolutional variational auto-encoder is to be found. For the decoder and the encoder, a conditional signal is first reserved (line 24 and 41), and then added to the input (see line 98 and 100). The labels are one hot encoded tensors. The generate method generates an image tensor. To be able to add the labels, some tensor manipulation needs to be done, therefore I created two redimensioning layers (lines 1-6 and 8-15).

Listing 15: A convolutional conditional variational auto-encoder

```
class UnFlatten(nn.Module):
 def __init__(self, width=32):
```

```
  super().__init__()
  self.width = width
 def forward(self, input):
  return input.view(input.size(0), self.width, -1).unsqueeze(1)

class Redimension(nn.Module):
 def __init__(self, channels,height,width):
  super().__init__()
  self.channels=channels
  self.height=height
  self.width=width
 def forward(self, input):
  return input.view(input.size(0), self.channels, self.height,self.width)

class CondConvVariationalAutoEncoder(nn.Module):
 def __init__(self,in_channels=1, latent_dim=2, n_classes=10):
  super().__init__()
  self.bottleneck=latent_dim

  unflatten = UnFlatten()
  self.encoder = nn.Sequential(
   nn.Linear(1034, 1024),
   unflatten,
   nn.Conv2d(in_channels=in_channels, out_channels=8, kernel_size=3, stride=2, padding=1,
    ↪ bias=False),
   nn.BatchNorm2d(8),
   nn.ReLU(),
   nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=1, bias=False),
   nn.BatchNorm2d(16),
   nn.ReLU(),
   nn.Conv2d(in_channels=16, out_channels=latent_dim, kernel_size=3, stride=2, padding=1),
   nn.Flatten(start_dim=1, end_dim=-1)
  )
  self.fn_mu=nn.Linear(32, latent_dim)
  self.fn_logvar=nn.Linear(32, latent_dim)

  redimension = Redimension(self.bottleneck, 4, 4)

  self.decoder = nn.Sequential(
   nn.Linear(12, 32),
   redimension,
   nn.ConvTranspose2d(in_channels=self.bottleneck, out_channels=16, kernel_size=3, stride=2,
    ↪ padding=1,
   output_padding=1),
   nn.BatchNorm2d(16),
   nn.ReLU(),
   nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=3, stride=2, padding=1,
    ↪ output_padding=1),
   nn.BatchNorm2d(8),
   nn.ReLU(),
   nn.ConvTranspose2d(in_channels=8, out_channels=in_channels, kernel_size=3, stride=2,
    ↪ padding=1,
   output_padding=1),
   nn.Sigmoid()
  )

 def forward_enc(self,x):
  x=self.encoder(x)
  mu=self.fn_mu(x)
  logvar=self.fn_logvar(x)
  sigma = torch.exp(0.5*logvar)
  noise = torch.randn_like(sigma)
  z=mu+(sigma*noise)
  return z,mu,logvar

 def forward_dec(self,z,labels):
  z = torch.cat((z, labels), dim=1)
  return self.decoder(z)
```

```
67
68 def generate(self, number=1):
69  torch.manual_seed(0)
70  z=torch.normal(0,1, size=(number, self.bottleneck))
71  generated=self.decoder(z)
72  generated = generated.reshape(-1, 1, 32, 32)
73  return generated
74
75 def generate_interpolation(self,kl,bs, epoch,number=1):
76  name = str(self.__class__.__name__)
77  r0 = (-3, 3)
78  r1 = (-3, 3)
79  n = 12
80
81  fig, axs = plt.subplots(n, n)
82
83  for i, a in enumerate(np.linspace(*r1, n)):
84   for j, b in enumerate(np.linspace(*r0, n)):
85    z = torch.Tensor([[a, b]])
86    y = torch.nn.functional.one_hot(torch.tensor([number]),
87    num_classes=10)
88    x_hat = self.forward_dec(z, y)
89    x_hat = x_hat.reshape(32, 32).detach().cpu().numpy()
90    axs[i, j].imshow(x_hat)
91    axs[i, j].axis('off')
92  plt.savefig("graph/" + name +"_bottleneck_"+str(bs)+"_"+str(kl)+ "/" + "ep_" + str(epoch) +
      ↪  "_" + str(number))
93  plt.clf()
94
95 def forward(self, x, labels):
96  batch_size, channels, height, width = x.shape
97  x=x.flatten(1)
98  x=torch.cat((x, labels), dim=1)
99  z,mu,logvar= self.forward_enc(x)
100  z=torch.cat((z, labels), dim=1)
101  dec = self.decoder(z)
102  dec=dec.reshape(batch_size,channels,height,width)
103  flat_labels = torch.argmax(labels, dim=1)
104  return z,dec,mu,logvar, flat_labels
```

Generating digits

An interesting idea is to take the median for every dimension of the encoded data, given the data was grouped by target and generate the digits per model.

Non-linear AEncoder

Figure 21: Decoding the non-linear auto-encoder's grouped median bottleneck-dimensions, given 2 such dimensions.

Figure 22: Decoding the non-linear auto-encoder's grouped median bottleneck-dimensions, given 3 such dimensions.

Figure 23: Decoding the non-linear auto-encoder's grouped median bottleneck-dimensions, given 4 such dimensions.

Simple linear AEncoder

Figure 24: Decoding the linear auto-encoder's grouped median bottleneck-dimensions, given 2 such dimensions.

Figure 25: Decoding the linear auto-encoder's grouped median bottleneck-dimensions, given 3 such dimensions.

Figure 26: Decoding the linear auto-encoder's grouped median bottleneck-dimensions, given 4 such dimensions.

Convolutional AEncoder

Figure 27: Decoding the convolutional auto-encoder's grouped median bottleneck-dimensions, given 2 such dimensions.

Figure 28: Decoding the convolutional auto-encoder's grouped median bottleneck-dimensions, given 3 such dimensions.

Figure 29: Decoding the convolutional auto-encoder's grouped median bottleneck-dimensions, given 4 such dimensions.

Generating digits from variational auto-encoders

The variational convolutional auto-encoder

bottleneck=3, KL=0.75

Figure 30: Generating random digits.

bottleneck=3, KL=1

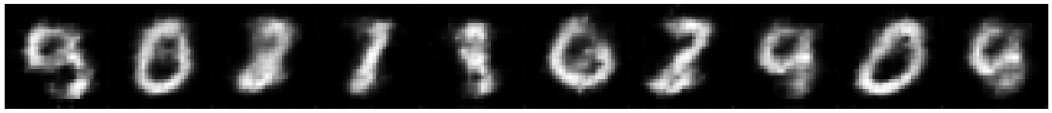


Figure 31: Generating random digits.

bottleneck=3, KL=1.25

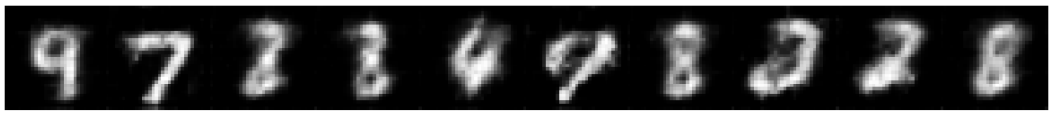


Figure 32: Generating random digits.

bottleneck=3, KL=1.5

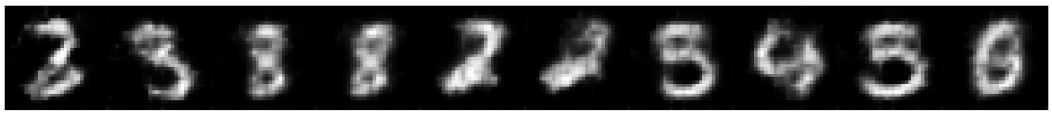


Figure 33: Generating random digits.

bottleneck=4, KL=0.75

Figure 34: Generating random digits.

bottleneck=4, KL=1

Figure 35: Generating random digits.

bottleneck=4, KL=1.25

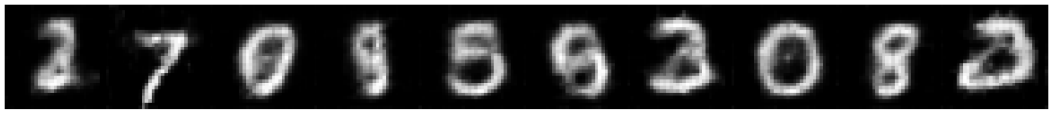


Figure 36: Generating random digits.

bottleneck=4, KL=1.5

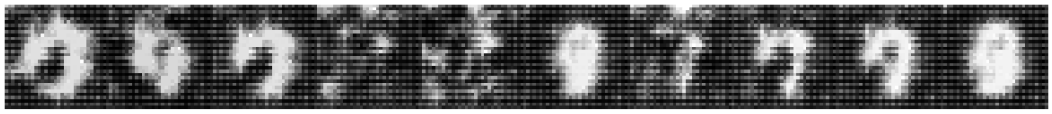


Figure 37: Generating random digits.

The variational auto-encoder

bottleneck=3, KL=0.75

Figure 38: Generating random digits.

bottleneck=3, KL=1

Figure 39: Generating random digits.

bottleneck=3, KL=1.25

Figure 40: Generating random digits.

bottleneck=3, KL=1.5

Figure 41: Generating random digits.

bottleneck=4, KL=0.75

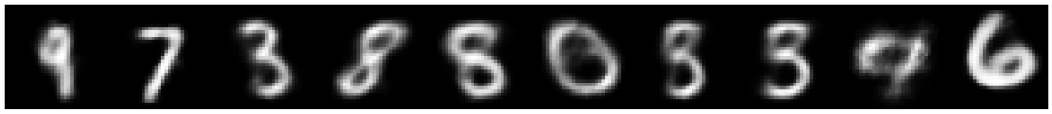


Figure 42: Generating random digits.

bottleneck=4, KL=1

Figure 43: Generating random digits.

bottleneck=4, KL=1.25

Figure 44: Generating random digits.

bottleneck=4, KL=1.5

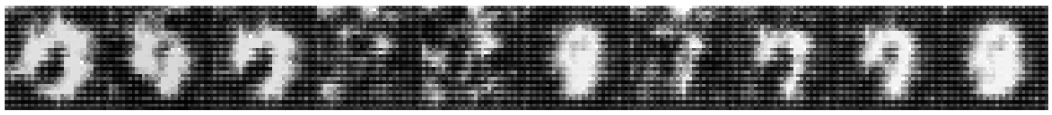


Figure 45: Generating random digits.

Model to 'read' images

The model to 'read' MNIST-like images is in Listing 16.

Listing 16: The model to 'read' the images

```
class NN(nn.Module):
 def __init__(self, input_size, num_classes):
  super(NN, self).__init__()
  self.fc1 = nn.Linear(input_size, 50)
  self.fc2 = nn.Linear(50, num_classes)

 def forward(self, x):
  x = F.relu(self.fc1(x))
  x = self.fc2(x)
  return x
```

This model is used to test generated/reconstructed MNIST-like data, this is done in Listing 17.

Listing 17: The code test readability of the MNIST-like images

```
def test_accuracy(loader,model_name,accuracy_dict):
 num_correct = 0
 num_samples = 0
 model= NN(input_size=input_size, num_classes=10)
 model.load_state_dict(torch.load("mnist_htr_model/trained_model/model", weights_only=True))

 model.eval()

 with torch.no_grad():
  for x, y in loader:
   x = x.reshape(x.shape[0], -1)

   # Forward pass: compute the model output
   scores = model(x)
   _, predictions = scores.max(1)  # Get the index of the max log-probability
   num_correct += (predictions == y).sum()  # Count correct predictions
   num_samples += predictions.size(0)  # Count total samples

  accuracy = float(num_correct) / float(num_samples) * 100
  print(f"Got {num_correct}/{num_samples} with accuracy {accuracy:.2f}%")
  accuracy_dict[model_name] = accuracy=accuracy
 return accuracy_dict
```

KL Divergence

Understanding surprise

"If you flipped a coin and I guessed the outcome perfectly, would you be surprised?". I assume you would be a little surprised but not completely shocked. Now, let's move to the second scenario.

"If you rolled a dice and I guessed the outcome perfectly, would you be more surprised than the previous scenario?". I assume you would say yes, since it is less probable for me to guess the correct outcome.

Now, one last scenario:
"If I correctly guessed what lottery numbers would win, would you now be more surprised than the previous scenario?". I assume you would say yes and be in complete shock. But why?

It is because as we progressed through the scenarios, the probability of me being able to guess the correct outcome reduced, and therefore your surprise increased at the outcome. So we have just noticed a relationship:

The probability of an event happening has an inverse relationship to surprise.

For clarity, as the probability of even happening reduces, the surprise increases, and vice versa.

However, we can make another interesting observation here. If we go back to the dice rolls, and imagine the scenario where you roll the same dice 3 times, and I guessed it correctly every time, how much more surprised would you be than if there was only 1 dice and I guessed it correctly.

Well, your surprise wouldn't be just slightly higher than a single correct guess; it would be dramatically higher, ideally three times higher. Why? Because each new correct guess compounds your disbelief. It's not just adding a fixed amount of surprise—it's multiplying how unbelievable the situation feels.

Therefore, when trying to mathematically define Surprise, we would want it to have:

- Additive property: If two independent events happen, their combined surprise should be the sum of their individual surprises.
- Inverse relationship with probability. The less likely an event, the more surprising it is
- Zero surprise for certain events. If something always happens (probability = 1), surprise should be zero.
- Continuous scaling. A small change in probability should lead to a smooth, logical change in surprise (no sudden jumps).

At first glance, since there are a lot of requirements, we would think that the mathematical definition will be quite complicated. However, it is not!

All of the above requirements can be solved by manipulating the logarithm function.
The probability of rolling a 6 on a fair die is $\frac{1}{6}$, and let's define the surprise of the event as this:

$$Surprise = log\left(\frac{1}{6}\right)$$

The probability of rolling 6 three times in a row is $\frac{1}{6} * \frac{1}{6} * \frac{1}{6}$, or $\frac{1}{216}$ In terms of surprise, it would be:

$$Surprise = log\left(\frac{1}{216}\right) = log\left(\frac{1}{6*6*6}\right) = log\left(\frac{1}{6}\right) + log\left(\frac{1}{6}\right) + log\left(\frac{1}{6}\right) = 3 * log\left(\frac{1}{6}\right)$$

in the above case the log function shows its additive property. For an event with probability 1, $log(1) = 0$, so the third property is fulfilled too. Additionally, Property 4 is also satisfied as log(x) is a continuous monotonic function! Property 2 is not satisfied, as our surprise decreases when the probability decreases (since it becomes more negative). This can be solved quite easily, by applying a negative sign!

$$Surprise = -log(P(x)) = log(\left(\frac{1}{P(x)}\right)$$

Entropy

In machine learning, we're not just interested in the surprise of a single event, but the average surprise across all possible events. That's called the expected surprise. More specifically, we are interested in finding the expected surprise of the distribution.

$$\mathbb{E}_{x\sim P[S(X)]} = -\sum_x P(x)log(P(x)) = H(P) = \text{Entropy}$$

This is a famous formula, and it is the expected surprise, or more commonly known as Entropy. Rather confusingly, however, we often use P and Q to denote our distributions, so don't get confused with p, which denotes probability!

Intuitively speaking, all we are doing is multiplying the probability of each outcome by its surprise, and then summing across all possible outcomes.

From this point on, we are going to state that P(x) is our true underlying distribution and Q(x)is the distribution we are trying to approximate P(x) with (i.e Q(X)is the "wrong" distribution). This is very common in machine learning—our models are constantly trying to estimate the real world.

What happens if we calculate our surprise using the wrong distribution?

We can still compute surprise, but instead of using P(x), we now measure it with respect to Q(x). This gives us a new expectation:

$$\text{Expected surprise under Q} = -\sum_x P(x)log(Q(x)) = H(P,Q)$$

This quantity is sometimes called the cross-entropy between P(x)and Q(x).

KL-divergence formula

Kl divergence is the difference between Entropy and Cross-Entropy.

$$KL(P||Q) = \text{H(P)-H(P,Q)}$$

The Kullback-Leibler (KL) divergence is a measure of how one probability distribution diverges from a second, expected probability distribution. It's defined as:

$$KL(P||Q) = \sum P(x)log\left(\frac{P(x)}{Q(x)}\right)$$

Properties

- Non-negativity: KL-Divergence is never negative. This makes sense intuitively, since the lowest value that it can take is 0, which is when our predicted distribution is exactly equal to the true distribution. And we can never have a negative cost since we cannot be "less surprised" than the theoretical minimum surprise.
- Anchored at zero: KL-Divergence gives us zero when our predicted distribution matches the true distribution.
- Asymmetry: This is an important property - KL-Divergence is not symmetric. Using Q(x)to approximate P(x) gives us a different value than using P(x) to approximate Q(x). This is why KL-Divergence isn't a true distance metric - it doesn't behave like normal distances.

Uses in Machine Learning

In ML, we often have a model that makes a prediction and a set of training data which defines a real-world probability distribution. It's natural to define a loss function in terms of the difference between the two distributions (the model's prediction and the real data).

Cross-entropy is very useful as a loss function because it's non-negative and provides a single scalar number that's lower for similar distributions and higher for dissimilar distributions. Moreover, if we think of cross-entropy in terms of KL divergence:

Coin flip example

When comparing a normal coin, $P(Heads) = 0.5$ and $P(Tails) = 0.5$ to a coin that has observed probabilities of $P(Heads) = 0.7$ and $P(Tails) = 0.7$.

$$KL(P||Q) = 0.5.log\left(\frac{0.5}{0.7}\right) + = 0.5.log\left(\frac{0.5}{0.3}\right) = 0.0378$$

This will result in a numerical value that represents the divergence of Q from P. The higher the KL divergence, the worse our estimated distribution Q is doing at approximating the real distribution P. The KL divergence is not bounded. It can take any value from 0 to infinity.