

# You're your own best teacher: A Self-Supervised Learning Approach For Expressive Representations

Johan Vik Mathisen

May 30, 2024

**TODO:** Introduce the section, what we think and the philosophy of presenting material in such a way.

## 0.1 Notation

- Encoder  $E$
- Decoder  $D$
- Estimated values are presented with a hat,  $\hat{x}$  for a reconstructed value,  $\hat{f}$  for a trained model etc.
- Parameters  $\theta$
- Dataset  $X = \{x_i\}_{i=1}^N$

## 0.2 Information theoretic/basic stats used in evaluation

- maximum entropy distribution
- mutual information

**Definition 1 (Differential entropy)** *The differential entropy of a random variable  $X$  with pdf or pmf  $p$  defined on a sample space  $\mathcal{X}$  is*

$$h(X) = \mathbb{E}[-\log(p(X))] - \sum_{x \in \mathcal{X}} p(x) \log(p(x)), \text{ if } X \text{ is discrete,} \quad (1)$$

$$h(X) = \mathbb{E}[-\log(p(X))] - \int_{\mathcal{X}} p(x) \log(p(x)), \text{ if } X \text{ is continuous} \quad (2)$$

- perplexity

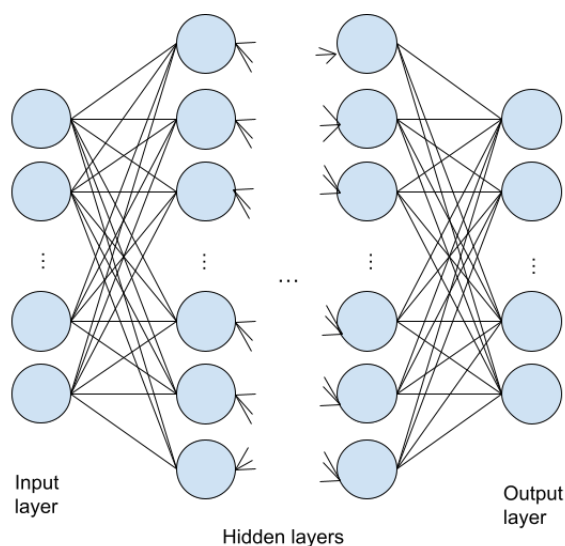
**Definition 2 (KL-Divergence)** *For probability distribution  $P$  and  $Q$  defined on the same sample space  $\mathcal{X}$ , if for all  $x \in \mathcal{X}$   $Q(x) = 0$  implies  $P(x) = 0$ , the Kullback-Leibler divergence is defined as*

$$\text{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right), \text{ if } P \text{ and } Q \text{ are discrete,} \quad (3)$$

$$\text{KL}(P||Q) = \int_{\mathcal{X}} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx, \text{ if } P \text{ and } Q \text{ are continuous.} \quad (4)$$

<https://stats.stackexchange.com/questions/188903/intuition-on-the-kullback-leibler-kl-189758#189758>

- Cross entropy
- Graphical probabilistic models - Ancestral sampling
- Generative models



**Figure 1:** Illustration of a Neural Network model.

### 0.3 Time Series Inference

- short-time-fourier transform etc.

### 0.4 Neural Network

An *artificial neural network* or simply *neural network* is a fundamental model in machine learning, and more specifically in *deep learning*. Neural networks are loosely inspired by the way neurons are assembled in the brain. The concept dates back to 1943 when Warren McCulloch and Walter Pitts developed the first artificial neuron, which is considered the first neural model ever invented[MCCULLOCH199099].

The first practical implementation was by Frank Rosenblatt in 1957 [rosenblatt1957perceptron]. However, it wasn't until the development of the backpropagation algorithm in its modern form in the 1980s that neural networks gained significant traction. Since then, neural networks have been highly influential in the machine learning field, with a broad range of impressive applications, including face recognition, defeating humans in chess, Go, and Starcraft, self-driving cars, and predicting protein structures. The remarkable effectiveness of neural networks across diverse tasks can partly be explained by the *universal approximation theorem*, proven by Kurt Hornik in 1989 [HORNLIK1989359], which roughly states that a neural network can approximate any (Borel measurable) function to any desired degree of accuracy.

A neural network takes in a vector  $x \in \mathbb{R}^n$  and builds a nonlinear function  $f(x)$  to predict the response  $y \in \mathbb{R}^m$ .

More specifically, a neural network maps an input vector  $x$  to an output vector  $y$  through a series of nonlinear functions applied to linear combinations of

the input. This particular structure, illustrated in Figure ??, distinguishes neural networks from other nonlinear prediction models. The variables  $x = [x_1, \dots, x_n]$  constitute the units of the *input layer*.

The intermediate layers are called the *hidden layers*, and the final mapping to  $y$  is called the *output layer*. A neural network is parameterized by a set of *weight* matrices  $W_i$  and *bias* vectors  $b_i$ , together with a specified non-linear *activation function*  $\sigma$ . We denote the collection of parameters  $\{W_1, \dots, W_{K-1}, b_1, \dots, b_{K-1}\}$  by  $\theta$ . Written out a  $K$  layered neural network is given by

$$f_\theta(x) = f_K \circ f_{K-1} \circ \dots \circ f_2 \circ f_1(x),$$

where

$$f_i(x) = \sigma_i(W_i x + b_i), \quad i \in \{1, \dots, K-1\},$$

and  $f_K$  is the output layer, with application dependent structure.

The introduction of nonlinearity by the activation function is what enables the model to approximate nonlinear signals, setting it apart from linear regression models. Two of the most commonly used activation functions are  $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$  and  $\text{ReLU}(x) = \max(0, x)$ , but countless other options exists.

The architecture of neural networks, and most specializations thereof, is sequential in nature. They can effectively be described as compositions of some combination of a flavour of matrix multiplication, non-linear transformation and down or upsampling.

#### 0.4.1 Training Neural Networks

Training a neural network involves finding the optimal values for the weight and bias parameters. The general idea behind neural network training is to optimize the parameters based on a distance metric between the predicted values and the target values. This distance metric is referred to as the *loss function*, with mean squared error (MSE) being a common choice.

For a multivariable function  $F$  that is differentiable around a point  $a$ , the direction in which it decreases fastest from  $a$ , is given by the negative gradient at  $a$ ,  $-\nabla F(a)$ . The gradient descent algorithm utilizes this property to find local minima of  $F$  by initializing the function with a value  $x_0$  and iteratively updating

$$x_{n+1} = x_n - \gamma \nabla F(x_n).$$

If the *learning rate*  $\gamma$  is small enough, we are guaranteed that  $F(x_{n+1}) \leq F(x_n)$ , and the sequence  $x_0, x_1, \dots$  converges to a local minimum of  $F$ .

A neural network  $f_\theta(x)$  is itself a multivariable function, and as long as the loss and activation functions are differentiable, the network is as well, both in terms of its arguments and parameters. By differentiating the network with respect to its parameters across its domain, the negative gradient indicates the fastest direction

to update the parameters to minimize the loss. The optimization problem at hand, for a dataset  $X = \{x_i\}_{i=1}^N$  with corresponding labels  $Y = \{y_i\}_{i=1}^N$ , is

$$\hat{\theta} = \min_{\theta} \frac{1}{N} \sum_{i=1}^N \mathcal{L}(f_{\theta}(x_i), y_i),$$

and one would iteratively update the parameters by gradient descent

$$\theta_{n+1} = \theta_n - \frac{\gamma}{N} \sum_{i=1}^N \nabla_{\theta} \mathcal{L}(f_{\theta}(x_i), y_i).$$

If the dataset  $X$  is large, gradient calculations can be expensive. In such cases, which are common in modern machine learning, an effective estimator for the true gradient is used instead. Stochastic gradient descent (SGD) is a very popular approach. Instead of computing the gradient at each data point in every iteration, SGD updates the parameters by iterating through the dataset and using the gradient at each single data point. The dataset is then permuted and iterated through again until an approximate minimum is reached. Pseudocode for SGD is presented in Algorithm ??.

---

**Algorithm 1** Stochastic Gradient Descent (SDG)

---

```

Initialize parameters  $\theta$  and learning rate  $\gamma$ 
while Not converged do
    Permute training set  $(X, Y)$ 
    for  $i$  in  $1, \dots, N$  do
         $\theta \leftarrow \theta - \gamma \nabla_{\theta} \mathcal{L}(f_{\theta}(x_i), y_i)$ 
    end for
end while

```

---

**TODO:** Mini batch SDG

For the actual gradient computations, the backpropagation algorithm is used. It provides an efficient way of computing gradients in neural networks by leveraging their compositional structure. Essentially, backpropagation is an efficient application of the Leibniz chain rule for differentiation.

For a thorough introduction to the subject of neural networks and their training, refer to Chapters 6 and 8 of [deeplearningbook].

## 0.5 Convolutional Neural Network

A convolutional neural network (CNN) is a specialized type of neural network designed to learn local features in data through the mathematical operation of convolution. Essentially, a CNN replaces matrix multiplication with convolution

in at least one of its layers [**deeplearningbook**].

Fully connected neural networks face significant challenges when dealing with high-dimensional input data, such as images, due to their rapidly increasing computational complexity. CNNs address this issue by employing downsampling techniques and leveraging the convolution operation to reduce the dimensionality and complexity of the data. This makes CNNs particularly well-suited for tasks involving high-dimensional data.

The concept of CNNs is inspired by the structure and function of the visual cortex in mammals. In the 1960s, Hubel and Wiesel discovered the visual processing capabilities of the cortex [<https://doi.org/10.1113/jphysiol.1968.sp008455>], leading to the development of the neocognitron in 1980 [6313076], which is considered the precursor to modern CNNs. In 1989, Yann LeCun and his team introduced a contemporary framework for CNNs and showcased its effectiveness in handwritten digit recognition [**LeCun1989ConvNet**]. Since then, CNNs have become a cornerstone of machine learning research, particularly in the field of computer vision.

For an in-depth exploration of the advancements and applications of convolutional neural networks, refer to [gu2017recent].

### 0.5.1 The convolution operation

The convolution operation is a fundamental mathematical concept with wide-ranging applications, particularly in the context of neural networks. It generalizes the notion of a moving weighted average and is deeply connected to the Fourier transform.

For real-valued functions  $f$  and  $g$ , their convolution is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \quad (5)$$

While the detailed mathematical criteria for the existence of this integral are beyond the scope of this thesis, it suffices to note that if  $f$  and  $g$  are integrable (in the Riemann or Lebesgue sense) then the convolution is well-defined. Generally, the convolution of  $f$  and  $g$  is as smooth as the smoothest of  $f$  and  $g$ . Additionally, convolution is commutative, meaning  $f * g = g * f$ , which can be shown by a simple change of variables.

In the context of convolutional neural networks, the function  $g$  is referred to as the *kernel*, and it consists of learnable parameters. The function  $f$  is the input, and the result of the convolution is called the *feature map* [**deeplearningbook**].

Since machine learning typically involves discrete signals represented as multidimensional arrays, we use a discrete version of the convolution operation. For a discrete input  $I$  and kernel  $K$ , their convolution is defined as

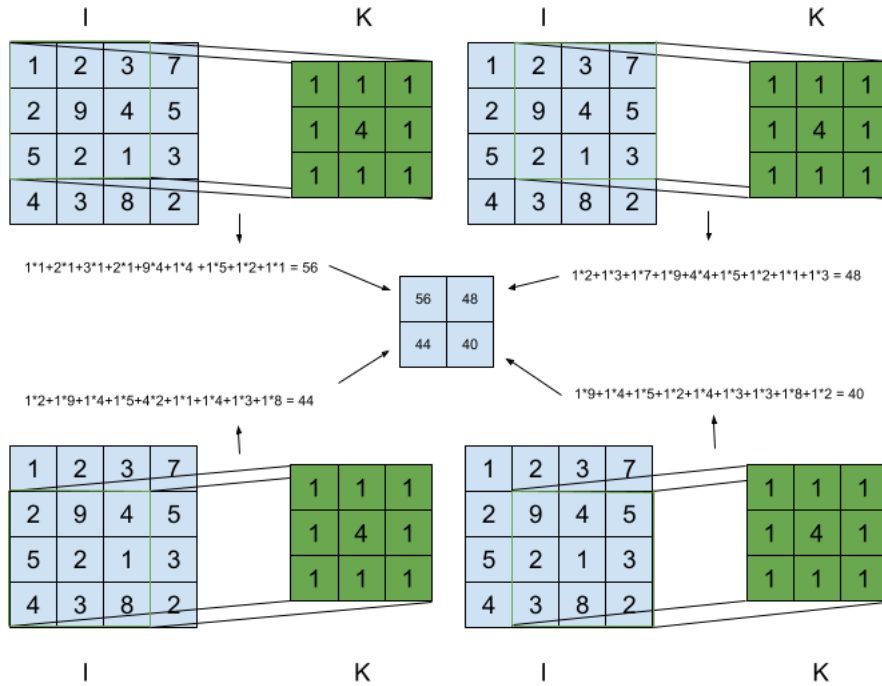
$$(I * K)[n] = \sum_{m=-\infty}^{\infty} I[m]K[n-m]. \quad (6)$$

In practice  $I$  and  $K$  usually have finite support, meaning they are zero beyond a certain range, which avoids any convergence issues.

Convolutions can be naturally extended to higher-dimensional functions. For a two-dimensional image  $I$  and a kernel  $K$ , their convolution is calculated as

$$(I * K)[i, j] = \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} I[n, m]K[i-n, j-m]. \quad (7)$$

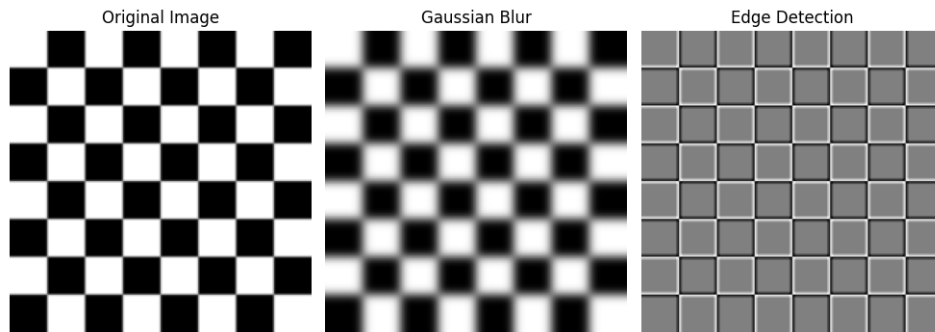
This operation is illustrated in Figure ??.



**Figure 2:** Illustration of discrete two dimensional convolution

In practice, the convolution operation used in machine learning often corresponds to cross-correlation rather than strict mathematical convolution, differing by a sign flip in the kernel arguments. While this makes the operation non-commutative, it does not affect the learning process, as the learned kernel parameters will be equivalent [deeplearningbook].

In digital signal processing, discrete convolutions are widely used with pre-defined kernels to manipulate signals predictably. Common examples include Gaussian kernels for blurring and edge detection kernels for highlighting areas of high intensity change. Figure ?? demonstrates the effects of these kernels. Edge detection provides insight into how CNNs learn features from data. In convolutional layers, kernels are learned to produce feature maps that aid in achieving the training objectives.



**Figure 3:** Illustration of discrete convolution applied to images. Image part of Scikit-image data package.

### 0.5.2 Pooling

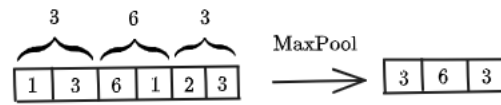
As mentioned earlier, CNNs are primarily used for high-dimensional data. To reduce the dimensionality to a manageable level, pooling operations are employed. Pooling is a downsampling technique applied to feature maps, where regions of the output are replaced with summary statistics. The aim with pooling is to retain the most important information while discarding less relevant details.

Two of the most common pooling methods are max pooling and average pooling. Max pooling replaces the region with its maximum value, while average pooling replaces it with the average value.

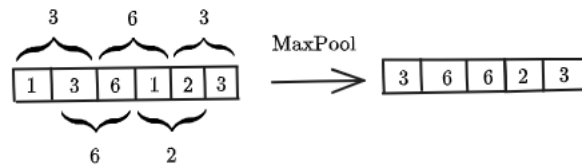
There are two key hyperparameters for any pooling operation: the filter size, which determines the region of values to calculate the summary statistic, and the stride length, which determines how the filter moves across the feature map. In addition to reducing dimensionality, pooling helps make representations approximately invariant to small distortions in the input. Illustrations of max pooling with different stride is presented in figure ?? and ??, while the effect of max and average pooling on image data is illustrated in ??.

Global pooling is a technique that involves calculating a summary statistic, such as the maximum or average, for an entire feature map. This effectively summarizes the presence of a feature across the entire input. This method is particularly useful in the final layers of a convolutional neural network (CNN), where it





**Figure 4:** Max pooling of one dimensional array. Filter size: 2, stride: 2.



**Figure 5:** Max pooling of one dimensional array. Filter size: 2, stride: 1.

reduces the spatial dimensions of the feature maps to a single value per map. This process condenses information and enhances computational efficiency. Additionally, global pooling helps CNNs handle variably sized inputs more effectively and serves as a method for downsampling representations for visualization purposes.

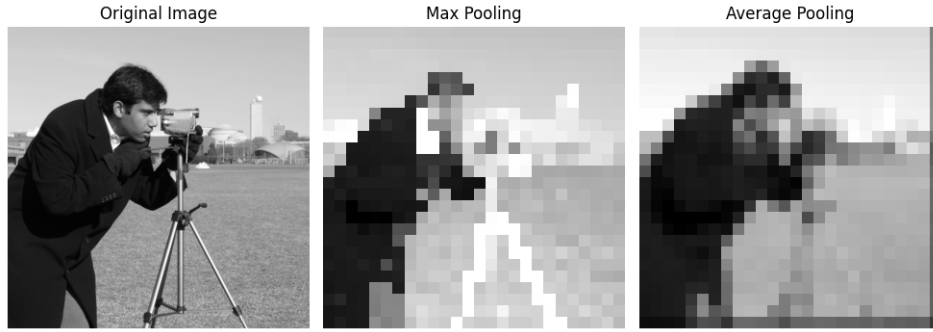
### 0.5.3 Architecture

There are many specific architectures within the category of convolutional neural networks (CNNs), but their basic components are largely the same. CNNs typically consist of convolutional layers, pooling layers, and fully connected layers. Figure ?? illustrates the original LeNet-5 [LeCun1989ConvNet] is presented as an example of a general CNN.

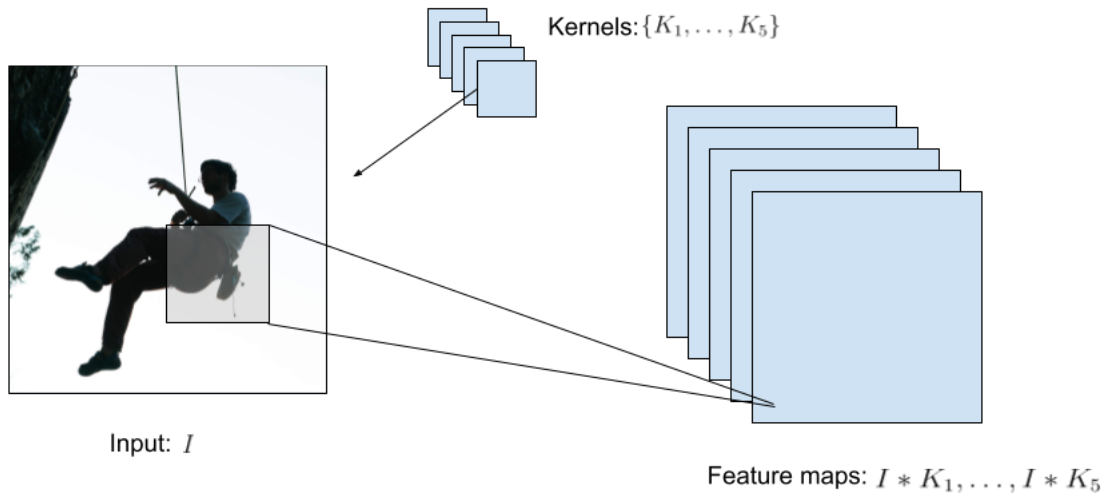
**A convolutional layer** consists of several kernels used to compute different feature maps. Each kernel is convolved with the entire input, producing different feature maps by using different kernels. After convolution, a nonlinear activation function is applied pointwise to the feature maps, introducing nonlinearity to the model. The purpose of the convolutional layer is to extract local features such as edges, textures, and shapes from the input data.

**A pooling layer** is typically placed between convolutional layers and act as strong downsamplers that aim to enforce approximate translation invariance. A pooling operation, such as max pooling or average pooling, is applied across each feature map, reducing its spatial dimensions while retaining important information. The pooling layer helps to reduce the computational load and control overfitting by progressively reducing the spatial size of the representation.

**Fully connected layers** are standard hidden layers in a neural network, connecting every input to every node in the output. Due to computational constraints, fully connected layers are introduced only after the input data has been sufficiently downsampled through convolutional and pooling layers. The fully connected layers aggregate the local features learned by convolutional layers across the entire



**Figure 6:** Illustration of mean and average pooling applied to images. Filter size and stride is  $20 \times 20$ . Original image size is  $512 \times 512$ , pooled images are  $26 \times 26$ . Image part of Scikit-image data package.

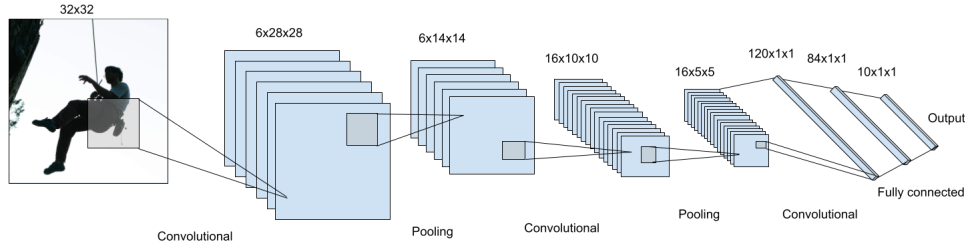


**Figure 7:** Convolutional layer.

image, learning high level reasoning and enabling the network to make final classifications or predictions.

In addition to these basic layers, modern CNN architectures often incorporate other techniques and layers to improve performance. Batch Normalization layers, for example, normalizes the inputs, which assist in stabilizing and speeding up the training process. They allow for higher learning rates and reduces the dependency on careful initialization.

Additionally, residual connections are frequently used in modern architectures. Introduced by He et al. in [he2015deep], residual connections allow gradients to flow directly through the network, alleviating the vanishing gradient problem in deep networks [279181]. They enable the training of much deeper networks by providing shortcut connections that bypass one or more layers. This



**Figure 8:** LeNet-5 network [LeCun1989ConvNet]

means that instead of learning the underlying mapping directly, the network learns the residual function, which is often easier to optimize. Residual connections have been a key innovation, enabling the development of very deep networks like ResNet, which have achieved state-of-the-art results in various computer vision tasks.

#### 0.5.4 Transposed Convolutional Networks

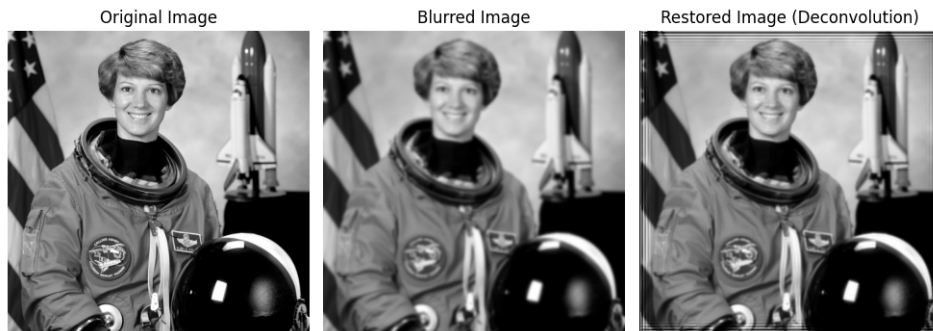
Transposed convolution, also known as deconvolution or fractionally-strided convolution, is a technique used to reverse the downsampling effect of standard convolutions. In essence, it is an inpainting or upsampling technique familiar from digital signal processing. The flexibility of learning data-dependent transposed convolutional kernels enables the effective reversal of the downsampling performed by convolutional layers.

Transposed convolutions are extensively used in combination with convolutional downsampling in encoder-decoder architectures presented in section ???. These architectures are fundamental in many generative models. The encoder part of the network typically reduces the spatial dimensions through a series of convolutional and pooling layers, capturing the essential features of the input data. The decoder part then uses transposed convolutions to upsample the encoded feature maps back to the original spatial dimensions, reconstructing the input or generating new data.

Figure ?? illustrates the application of deconvolution to images. In this example, a Gaussian blur is applied, and the image is restored using the Richardson-Lucy algorithm with a Gaussian deconvolution kernel. This process demonstrates how transposed convolutions can be used to approximately reverse the effects of convolutional operations.

## 0.6 Representation Learning

The focus of this thesis is representation learning for improved time series generation.



**Figure 9:** Illustration of deconvolution applied to images. Here using Gaussian blur and restoring using the Richardson Lucy algorithm with a Gaussian deconvolution kernel. Image part of Scikit-image data package.

### 0.6.1 What a representation?

Representation learning is a term that can be difficult to define due to its abstract nature. To understand it better, let's first consider what is meant by the representation of information.

Take the base ten integer  $(4)_{10} = 4$  as an example. This number can be equivalently expressed in other bases, such as binary  $(4)_2 = 100$ . The base we choose depends on our intention. For digital electronics, a binary representation is useful because transistors have two states. For human arithmetic, base ten is natural because we have ten fingers. Thus, a particular representation can make a task easier or harder, although the information content remains unchanged. What changes is the ease or difficulty of certain information processing tasks. Representation learning is the process of learning a representation of information that facilitates these tasks.

Representations are highly dependent on who or what will process the information. Consider the example of time. Humans have developed a standardized system for writing timestamps that works well for us. However, if we want to model time-dependent phenomena using tabular data, the `DateTime` representation might be unhelpful to a tree-based model. The numerical representation of timestamps close in time is not necessarily close in numerical value, such as 23:59 and 00:00. A possible solution is to change the representation to reflect the periodic nature of time by mapping it to a circle. This new representation might be less intuitive for humans but more useful to a computer.

In machine learning, representation learning involves designing algorithms that learn representations useful for a specific objective. The goal is to find representations that make it easier for the machine learning models to process and understand the data, improving performance on the given tasks.

### 0.6.2 Why do we care about representation learning?

Those who has worked with data science or machine learning and has come across feature engineering are familiar the effect good feature engineering has on a model's performance. The same people too knows the level of domain expertise, creativity and time is needed to feature engineer well. In reality much of the actual time spent in the process of deploying machine learning methods revolves around constructing good data pipelines and applying transformations that produce representations beneficial for the algorithm at hand [**Rep-rev-persp**]. Thus the ability to automate such tasks would be incredibly beneficial, and ease the use of ML algorithms significantly. It is here one of the intriguing and promising features of neural networks, with its many specializations and architectures, comes into play. They have shown the ability to learn useful abstract representations of the data and provide automatic feature engineering [**Rep-rev-persp**].

### 0.6.3 What is a good representation?

The quality of a representation is determined by how effectively it facilitates subsequent tasks. The idea of universally good representations is ill-defined because any representation derived from a non-invertible function can be specifically countered by designing a *downstream tasks* that relies on the lost information, leading to poor performance. In essence, there is no free lunch in representation learning either.

To evaluate the quality of a representation, one must specify a set of predefined downstream tasks and assess performance based on those tasks. A representation is considered better if it enables the model to perform well on the downstream tasks. Additionally the more general a representation is, meaning they facilitate high performance on a larger set of tasks, the higher its quality.

### 0.6.4 How does one evaluate representations?

There are several evaluation protocols in representation learning. These typically involve training a model on a *pretext task*, which, as defined in [**jing2019selfsupervised**], is a task designed for the network to solve with the primary goal of learning useful representations. The learned representations are then evaluated on a downstream task, generally solved in a supervised manner using human-annotated data.

In an  $N$ -layered network  $f = f_N \circ \dots \circ f_1$ , the intermediate value of the data  $x$  in some layer  $n$  represents the network's learned feature representations. When focusing on the learned representations, it is helpful to decompose the model  $f$ , notation wise, into a *feature extractor*  $h$  and an *output function*  $g$  such that it can be factored as  $f = g \circ h$ . Representation learning algorithms typically follow this pattern

- Train  $f = g \circ h$  on a pretext task.
- Discard  $g$

- Use the learned feature extractor  $\hat{h}$  as part of a new model.
- Evaluate the new model on the downstream task.

The standard evaluation protocol involves training a linear head  $g_D$  on top of the *frozen* representations in a supervised manner and evaluate this model's performance. This means training  $f_D = g_D \circ \hat{h}$  by only updating the parameters of the linear model  $g_D$ , and evaluate  $\hat{f}_D$  on a test set. This protocol is sometimes referred to as *linear probing*. A common downstream task is classification, where the idea is that good and informative representations should differentiate data in such a way that it is easy to separate them.

An alternative protocol, similar to the one above, allows all parameters of  $f_D$  to be learnable on the downstream task. This protocol is referred to as a pretraining-finetuning. In this approach, the model is pretrained on a large, potentially unlabeled dataset, and then fine-tuned on a smaller, labeled dataset specific to the downstream task.

Evaluating  $f_D$  involves assessing accuracy on the downstream task, training time, and the sensitivity of these metrics to the training data size. The baseline comparison is typically an identical but randomly initialized model. It is highly advantageous if one can pretrain a feature extractor using abundant, cheap data, ensuring faster convergence on a downstream task where data is expensive or scarce.

It is worth mentioning that in cases where the sole goal is to create the best-performing model, a more complex task-specific head  $g_D$  is often used. For a comprehensive survey on representation learning, see[[nozawa2022empirical](#)].

## 0.7 Transformers

Since their introduction in the seminal paper "Attention is all you need"[[vaswani2023attention](#)], transformers have revolutionized the fields of machine learning and artificial intelligence. This architecture is the backbone of large language models (LLMs) such as Google's BERT, Meta's Llama and OpenAI's ChatGPT. Following their initial success in natural language processing, transformers have been adapted for other modalities, including computer vision with Vision Transformers [[dosovitskiy2021image](#)], and similar trends are shown for other modalities such as audio [[latif2023transformers](#)] and time series [[wen2023transformers](#)].

Transformers were developed for sequential modeling and have the capability of learning long-range dependencies in data through the *multi-headed attention mechanism*. Previously, recurrent neural networks (RNNs) and long short-term memory (LSTM) networks were the state-of-the-art for sequential tasks. However, they struggled with modeling long-term dependencies due to the vanishing or ex-

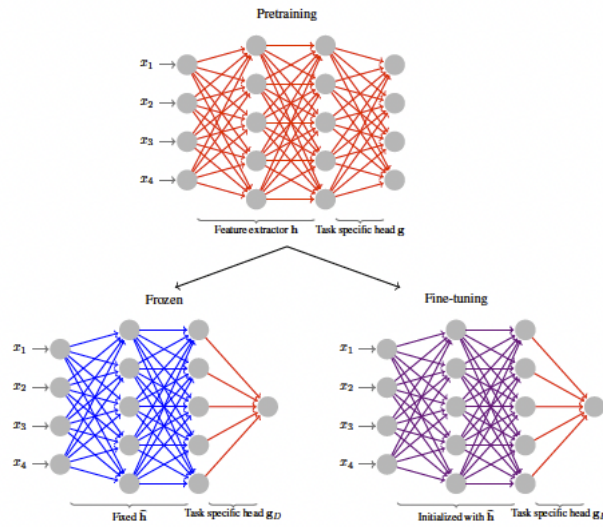


Figure 10: Taken from [nozawa2022empirical]

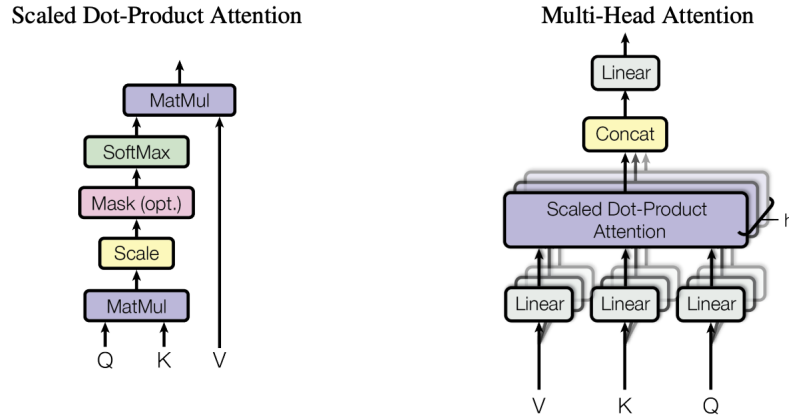
ploding gradient problem [279181]. One of the main novelties of the transformer architecture is not relying on recurrence, and instead solely using the attention mechanism to capture dependencies between input and output.

The transformer takes as input a sequence of symbols. For different modalities like text, speech, images, and time series, this requires a process called *tokenization*. For example, in natural language processing (NLP), a simple word-level tokenization involves creating a dictionary of all words in the dataset and assigning each word an integer. Text can then be mapped to a sequence of integers. Other modalities, such as speech, images, and time series, are usually represented as matrices which can be flattened and used as inputs. However, directly modeling high-dimensional data as such sequences would require significant computational resources and make it challenging to model long-range dependencies. Therefore, various tokenization methods are used to represent the data as coarser sequences.

Transformers have also been successfully adopted for generative tasks. Autoregressive transformers, such as the GPT series, are trained to predict the next token in a sequence given the preceding tokens. By treating images as sequences of patches, Vision Transformers can generate new images or inpaint missing patches based on the input. With the introduction of Vision Transformers [dosovitskiy2021image], input images are split into patches and linearly projected before being processed by the transformer. Since then, *Vector-Quantized*-based tokenization introduced by [VQVAE] has emerged as a popular approach.

### 0.7.1 The Attention Mechanism

The attention mechanism is a key innovation in transformer architectures, enabling them to handle long-range dependencies effectively. Attention can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and outputs are all vectors. This mechanism allows the model to focus on different parts of the input sequence simultaneously.



**Figure 11:** (left) Scaled dot-product attention. (right) Multi-headed attention. Taken with permission from [vaswani2023attention]

Scaled dot-product attention is computed by

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (8)$$

This equation allows the model to weigh the importance of different input tokens based on their relevance to the query, enabling it to focus on the most relevant parts of the input.

To enhance the model's ability to capture different aspects of the input, multi-headed attention is used. This involves projecting the queries  $Q$ , keys  $k$ , and values  $V$  into multiple subspaces and applying scaled dot-product attention in each subspace. The results from each subspace are then concatenated and linearly transformed. The process is defined as

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \quad (9)$$

where

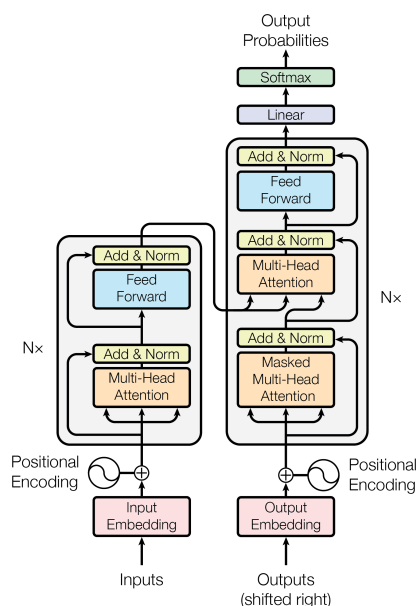
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V). \quad (10)$$



Multi-headed attention enables the transformer to attend to different parts of the input sequence and capture various features simultaneously. By having multiple attention heads, the model can learn different relationships and dependencies within the data, leading to a richer representation.

### 0.7.2 Architecture

The original transformer architecture, as presented in Figure ??, consists of several distinct components that work together to process sequential data effectively. This architecture includes an encoder and a decoder, each composed of multiple layers that leverage the attention mechanism to handle dependencies within the input data.



**Figure 12:** The Transformer - model architecture. Encoder (left), decoder (right). Taken with permission from [vaswani2023attention]

**Embeddings:** The first step in the transformer architecture involves mapping each token in the input sequence to a vector through an input embedding. This process is essentially a lookup table where each token is converted into a continuous vector representation. These embeddings allow the model to process and learn from the input data more effectively.

**Positional encoding:** Transformers do not inherently account for the order of the input sequence, as they lack the sequential nature of recurrent models. To address this, positional encoding is added to the input embeddings. Positional encoding provides information about the position of each token in the sequence, allowing the model to differentiate between tokens based on their order. In the

original paper [vaswani2023attention], sinusoidal functions were used to generate unique positional encodings for each position.

**Encoder:** The transformer encoder processes the input embeddings with positional encodings through multiple identical layers. Each encoder layer consists of two main components. The multi-head attention layer enables the model to focus on different parts of the input sequence simultaneously, capturing various dependencies. This is followed by a fully connected layer which processes the output of the multi-head attention, further transforming the data. Each sublayer incorporates residual connections and normalization.

**Decoder:** The transformer decoder is similarly structured, consisting of multiple identical layers. They consist of a masked attention layer, which is similar to an attention layer but prevents the model from attending to future tokens in the sequence during training, ensuring that predictions are based only on past and present tokens. A multi-head attention which is applied to the output of the encoder and a fully connected neural network. All sublayers employ residual connections and normalization. The final output of the decoder is converted to probabilities using a linear layer followed by a softmax function.

Bidirectional transformer models, such as BERT [devlin2019bert], utilize an encoder-only architecture. This design allows the attention mechanism to consider information from both directions within the sequence, enabling the model to capture a more comprehensive understanding of the context. The issue of "seeing into the future" is resolved through a training technique called *masked modeling*, where some tokens are masked and the model is trained to predict them based on the surrounding context. This will be looked at further in Section ??.

## 0.8 Self-Supervised learning

Self-supervised learning (SSL) has achieved great success in natural language processing (NLP) and computer vision in recent years and is rapidly being applied to other modalities. In our work, we leverage SSL for representation learning in the time series domain. Before diving into SSL, it is important to understand the broader context of machine learning, which can be coarsely divided into two categories: supervised and unsupervised learning.

### Supervised Learning

Supervised learning involves training models using labeled data. For a given input  $x$ , the desired output  $y$  is known during training, allowing for direct supervision of the model's parameters by comparing its output to the true value. A bit more formally, for a dataset  $X = \{x_i\}_{i=1}^N$  with corresponding human-annotated labels

$Y = \{y_i\}_{i=1}^N$ , the objective of a supervised learning algorithm is to fit a function  $f_\theta$  that minimizes the loss across the data

$$\hat{f}_\theta = \min_{\theta} \frac{1}{N} \sum_{i=1}^N L(f_\theta(x_i), y_i). \quad (11)$$

Common approaches to supervised learning for neural networks is to calculate some distance metric between the predicted value  $\hat{y}$  and the true value  $y$  and update parameters using backpropagation.

Supervised learning models, including classical statistical models, support vector machines, and decision tree-based models, have seen tremendous success but are limited by the need for labeled data, which is often scarce and expensive to obtain.

### Unsupervised Learning

Unsupervised learning, on the other hand, refers to models that learn exclusively from unlabeled data, discovering intrinsic patterns without explicit labels. Examples of unsupervised learning techniques include clustering methods (e.g., K-means, K Nearest Neighbor, Gaussian mixture models), dimensionality reduction techniques (e.g., PCA, SVD), and neural network architectures such as autoencoders. These techniques are widely used in exploratory data analysis, data visualization, and clustering. While unsupervised learning is valuable in a world abundant with unlabeled data, it has historically struggled to match the performance of supervised approaches.

### Self-Supervised Learning

Self-supervised learning is a subcategory of unsupervised learning that has shown remarkable success in NLP and computer vision, approaching the performance of state-of-the-art supervised representation learning [bardes2022vicreg, zbontar2021barlow]. SSL involves using the data itself to generate a supervisory signal, rather than relying on external labels. Although SSL is technically unsupervised, its learning formulation closely resembles that of supervised learning.

For a dataset  $X = \{x_i\}_{i=1}^N$  with *pseudo labels*  $P = \{p_i\}_{i=1}^N$ , the objective of a self-supervised learning algorithm is to fit a function  $f_\theta$  that minimizes the loss across the data

$$\hat{f}_\theta = \min_{\theta} \frac{1}{N} \sum_{i=1}^N L(f_\theta(x_i), p_i). \quad (12)$$

Pseudo labels are automatically generated labels from the data. In *siamese* architectures, these pseudo labels are typically augmentations of the original data. SSL has become an integral part of pretraining large models, such as those used in LLMs and vision transformers, enabling them to learn from vast amounts of

unlabeled data. Pretraining with SSL allows models to capture the semantics of data without the need for extensive labeled datasets, significantly reducing the resources required for subsequent supervised training.

### 0.8.1 Siamese Architecture-based SSL

A siamese network architecture [**siamese**] consists of two branches with a shared encoder on each branch. These networks are trained to produce similar representations for different views of the same data. A major challenge in this architecture is avoiding trivial solutions, where the networks ignore the input and produce identical constant embeddings, a problem known as *collapse*.

In computer vision, several representation learning algorithms utilizing siamese architectures have been proposed. These can be broadly categorized into *contrastive* and *non-contrastive* SSL methods.

Contrastive SSL uses positive and negative samples to learn representations by pulling positive pairs closer together and pushing negative pairs further apart. Examples include MoCo [**he2020momentum**] and SimCLR [**chen2020simple**]. Contrastive methods often require large batch sizes as well as a substantial number of negative pairs compared to positive in order to learn representations effectively [**lee2024computer**].

Non-Contrastive SSL methods, such as BYOL [**grill2020bootstrap**], BarlowTwins [**zbontar2021barlow**] and VICReg [**computer**] avoid the need for positive and negative pairs. These methods use augmentations of the input data as pseudo labels and introduce different mechanisms to prevent collapse. For instance, BYOL introduces architectural asymmetry and the stop-gradient operation, Barlow Twins minimizes information redundancy between branches, and VICReg maintains variability while decorrelating features in each branch.

### 0.8.2 Masked modelling

Masked modeling is a straightforward self-supervised learning technique for generative models. The core idea is to mask or cover a portion of the data and train the model to predict the masked portions. By comparing the predictions against the unmasked data, the model learns useful representations without explicit supervision. Introduced in NLP by BERT [**devlin2019bert**], masked modeling has become the standard for self-supervised pretraining of language models.

In computer vision, masked modeling has also gained attention. Initial approaches involved masking image patches directly [**he2021masked**]. With the success of vision transformers [**dosovitskiy2021image**], researchers began tokenizing images and pretraining transformers in a manner similar to BERT. For example, MaskGIT [**chang2022maskgit**] proposed masked visual token modeling,

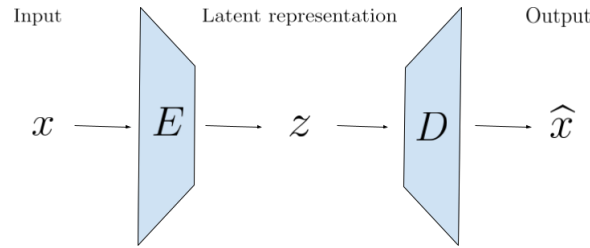
using vector-quantized-based tokenization and a bidirectional transformer. For a comprehensive survey on masked modeling in the vision domain, refer to [li2024masked].

## 0.9 Vector Quantized Variational Autoencoder (VQVAE)

Our model is based on the Vector Quantized Variational Autoencoder (VQVAE) introduced in [VQVAE], and includes an Auto-encoder (AE) branch. Therefore it is natural to dive into the models. We first start with introducing auto-encoders, then present the variational variation VAE before presenting VQVAE.

### 0.9.1 Autoencoder (AE)

An autoencoder consists of two neural networks: an encoder  $E$  and a decoder  $D$ . These networks map data between two spaces: the data space  $X$  and the latent space  $Z$ . The encoder compresses the data from the data space to the latent space, while the decoder reconstructs the data from the latent space back to the data space. A schematic of the architecture is presented in Figure ??.



**Figure 13:** Schematics of the autoencoder architecture.

The primary goal of an autoencoder is to learn an efficient encoding of the data. The encoder compresses the data into a lower-dimensional representation, and the decoder reconstructs the original data from this compressed representation. The training objective is to make the composition of the encoder and decoder approximate the identity function, meaning that the output should be as close as possible to the input. Autoencoders are typically trained to minimize the reconstruction error, which typically is the mean square error (MSE) between the input and the reconstructed output. This process forces the model to learn efficient latent representations of the data.

One of the critical aspects of autoencoders is the *information bottleneck*. The dimension of the latent space is much smaller than that of the data space. This bottleneck forces the model to capture the most important features of the data while discarding less relevant information. By compressing the data, the encoder learns a compact representation that retains the essential features needed for accurate reconstruction by the decoder. However, learning effective latent representations can be challenging. Without proper regularization, the latent space may capture

noise or irrelevant details, leading to overfitting.

Event though Autoencoders are outdated, they serve as the foundational building block for more complex models such as Variational Autoencoders (VAEs) and Vector Quantized Variational Autoencoders (VQVAEs), which further enhance the ability to learn meaningful and useful representations from data.

### 0.9.2 Variational Autoencoder (VAE)

Variational Autoencoders (VAE) were introduced in [kingma2022autoencoding] and represent a variational Bayes approach to approximate inference and learning with directed probabilistic models. While the architecture of VAEs is similar to that of traditional autoencoders (AEs), their mathematical formulation is fundamentally different. Variational inference is a statistical technique used to approximate complex distributions by finding the closest approximation within a simpler, but flexible, parametric family.

In the VAE framework, we assume that the dataset  $X = \{x_i\}_{i=1}^N$  consists of iid samples from a random variable  $\mathbf{x}$ . We further assume that the data is generated by some unobservable random process. Specifically, that there is a latent variable  $\mathbf{z}$  such that  $x_i \sim p_{\theta^*}(x|z_i)$ , where  $z_i \sim p_{\theta^*}(z)$ . The distribution  $p_{\theta^*}(z)$  is referred to as the true prior, and  $p_{\theta^*}(x|z_i)$  as the true likelihood. Since  $\mathbf{z}$  is unobservable and the true distributions are unknown, one has to assume their form. In general the prior and likelihood are assumed to be from parametric families  $p_{\theta}(z)$  and  $p_{\theta}(x|z)$ . These distributions are typically chosen to be differentiable to facilitate gradient-based learning.

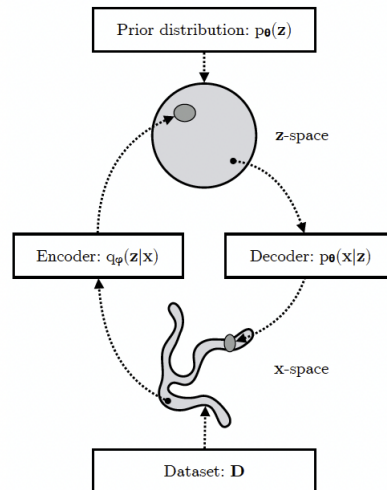


Figure 14: [VAE], need to ask for permission or make own

VAEs have two main components: the probabilistic encoder (or inference model)  $q_\phi(z|x)$ , which approximates the true posterior, and the probabilistic decoder (or generative model)  $p_\theta(x|z)$ , which approximates the likelihood. These models are typically parameterized by neural networks, with  $\phi$  and  $\theta$  representing their weights and biases. Given a datapoint  $x_i$ , the probabilistic encoder provides a distribution over the possible values of the latent variable  $\mathbf{z}$ . Similarly, given a latent representation  $z_i$ , the probabilistic decoder produces a distribution over the possible corresponding values of  $\mathbf{x}$ .

The probabilistic nature of the encoder and decoder involves sampling. For a given  $x$  or  $z$ , the output is a distribution, and the same input will yield the same output distribution if the network parameters are unchanged. Actually,  $x$  and  $z$  are mapped to the parameters of a distribution, which uniquely determines the distribution in a particular family.

Common assumptions for the distributions are:

- Prior  $p_\theta(z) \sim N(0, I)$
- Likelihood  $p_\theta(x|z) \sim N(D(z), I)$
- Variational posterior  $q_\phi(z|x) \sim N(E(x)) = N(\mu_x, \Sigma_x)$

The encoder maps datapoints to the parameters of the variational distribution  $q_\phi$ ,

$$x \mapsto E(x) = (\mu_x, \Sigma_x).$$

A latent representation  $z$  is then sampled from  $q_\phi$ , which constitutes the random part of the algorithm. The decoder maps  $z$  to the expected value of the likelihood  $p(x|z)$ ,

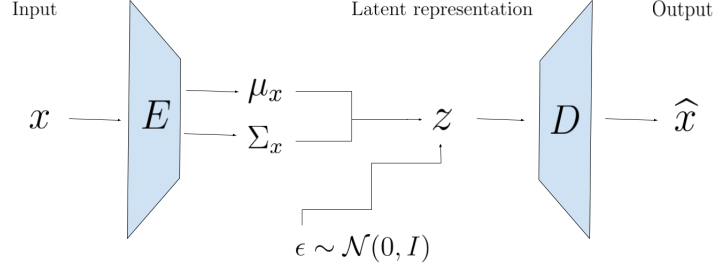
$$z \mapsto D(z) = \hat{x}.$$

There are several reasons for choosing Gaussian distributions, one being that the Gaussian distribution is a scale-location family. This enables us to employ the reparameterization trick, and circumvent the problematic random component in the VAE when using gradient based learning. It involves introducing an auxiliary variable  $\epsilon \sim N(0, I)$  and rewriting  $z = \mu_x + L_x \epsilon$ , where  $L_x$  is the Cholesky decomposition of  $\Sigma_x$ . This allows the gradient to flow through the deterministic parts of the model.

### Training objective

VAEs are optimized with respect to the *evidence lower bound* (ELBO). For jointly distributed variables  $\mathbf{x}$  and  $\mathbf{z}$  with distribution  $p_\theta$ . Then for any distribution  $q_\phi$  the ELBO is defined as

$$\mathcal{L}_{\theta, \phi}(x) = \mathbb{E}_{q_\phi(z|x)} \log \left( \frac{p_\theta(x, z)}{q_\phi(z|x)} \right) \quad (13)$$



**Figure 15:** Schematics of the VAE architecture with Gaussian reparameterization.

This can be reformulated in terms of the marginal likelihood and KL divergence between the variational and true posterior

$$\begin{aligned} \mathcal{L}_{\theta, \phi}(x) &= \mathbb{E}_{q_{\phi}(z|x)} \log(p_{\theta}(x)) + \mathbb{E}_{q_{\phi}(z|x)} \log\left(\frac{p_{\theta}(z|x)}{q_{\phi}(z|x)}\right) \\ &= \log(p_{\theta}(x)) - \text{KL}(q_{\phi}(z|x) || p_{\theta}(z|x)). \end{aligned} \quad (14)$$

Due to the non-negativity of the KL-divergence, we see that the ELBO bounds the marginal log likelihood of the data from below. By maximizing the ELBO with respect to the model parameters  $\phi$  and  $\theta$ , one simultaneously maximizes the marginal likelihood and minimizes the KL divergence, improving both the generative and inference models [VAE].

An alternative formulation shows a more evident connection to autoencoders, with the prior acting as a regularizer on the posterior

$$\begin{aligned} \mathcal{L}_{\theta, \phi}(x) &= \mathbb{E}_{q_{\phi}(z|x)} \log(p_{\theta}(x|z)) - \mathbb{E}_{q_{\phi}(z|x)} \log\left(\frac{q_{\theta}(z|x)}{p_{\phi}(z)}\right) \\ &= \underbrace{\mathbb{E}_{q_{\phi}(z|x)} \log(p_{\theta}(x|z))}_{\text{Expected reconstruction log likelihood}} - \underbrace{\text{KL}(q_{\phi}(z|x) || p_{\theta}(z))}_{\text{Regularizer}}. \end{aligned} \quad (15)$$

Assuming a Gaussian likelihood, we have

$$\log p_{\theta}(x|z) = -\frac{1}{2} \left[ k \log(2\pi) + (x - \hat{x})^T (x - \hat{x}) \right], \quad (16)$$

which is equivalent, as an optimization objective of  $\hat{x}$ , to  $\|x - \hat{x}\|_2^2$ . Consequently the likelihood in the loss is implemented as the MSE of the input  $x$  and the output  $\hat{x}$ .

### Generative model

The prior in a VAE serves two main roles: as a regularizing constraint for the posterior during training, and as a means of generating new samples via ancestral sampling. Ancestral sampling refers to  $z$  being the parent node of  $x$  in the VAE,



and that we can generate samples  $x$  by drawing a sample  $z \sim p(z)$  and decode  $D(z)$ .

### Limitations

Despite their strengths, VAEs can suffer from issues such as posterior collapse, where the latent variable  $z$  fails to capture meaningful information about the input data. This can lead to poor reconstruction quality and less informative latent representations. Additionally, VAEs may struggle with variance issues, where the variance of the learned latent distributions is not well-calibrated, affecting the quality of generated samples.

These limitations have motivated the development of more advanced models, such as the Vector Quantized Variational Autoencoder (VQVAE), which aims to address some of these challenges by introducing discrete latent variables and a learnable prior distribution.

### 0.9.3 Vector Quantization (VQ)

Dictionary learning model [Gray1984VQ]

### 0.9.4 VQVAE

The Vector Quantized Variational AutoEncoder (VQVAE) was first introduced in [VQVAE] and presented a new way of training VAEs with discrete latent variables. It is the first discrete VAE model which has similar performance to the continuous variant. VQVAE was developed to learn useful discrete representations for generation. This is enforced by more flexible distributional assumptions, as they say "Our model, the Vector Quantised- Variational AutoEncoder (VQ-VAE), differs from VAEs in two key ways: the encoder network outputs discrete, rather than continuous, codes; and the prior is learnt rather than static."-[VQVAE].

The overall architecture of VQVAE consists of an encoder and decoder as before, together with a *codebook* which is used in the quantization process. The codebook, also referred to as the latent embedding space,  $\mathcal{Z} = \{z_k\}_{k=1}^K$  consists of  $K$  latent vectors with dimensionality  $D$ . It can be considered a lookup table of vectors that are used for vector quantization (VQ). The output of the encoder  $E(x)$  is quantized by the means of nearest neighbor lookup

$$z_q = \arg \min_{z_k \in \mathcal{Z}} \|E(x) - z_k\|. \quad (17)$$

From ?? we can see that the VQVAE can be seen as an autoencoder with a nonlinearity introduced by the vector quantization.

In the original article [VQVAE] they propose to separately learn the prior distribution, after the encoder, decoder and codebook is learned. During the initial



## Prior Learning

During the initial training the prior is as mentioned constant and uniform such that the posterior has more flexibility. After training an autoregressive prior is fit on  $z$  so we can sample  $x$  via ancestral sampling. The prior is learned with a generative objective, which increases the quality of generated samples. The choice of prior model depends on the particular application. In the original article Pixel-CNN [oord2016pixel] was used for image, while WaveNet [oord2016wavenet] was used for audio. More recently in TimeVQVAE [TimeVQVAE] utilized a bidirectional transformer [chang2022maskgit] for prior learning in the time series domain.

## Loss function

As we want to do gradient based learning, and the quantization process is quite far off from being continuous, measures has to be taken. The output of the encoder and the input of the decoder has the same dimension, which opens the possibility of simply copying the gradients from the decoder input to the decoder output. This is referred to as the *straight through estimator*. The gradients from the decoder contains relevant information to how the encoder should change its output, and in the subsequent forward pass the encoder output can be quantized to different discrete latent codes.

The loss function consists of two parts, the reconstruction loss, codebook loss. We already mentioned that the ELBO objective reduced to the reconstruction term only. In order to learn the codebook one uses the codebook loss, which is the euclidean distance between the encoder output and the quantized output, together with commitment term, which is introduced to keep the distance between code-words from growing arbitrarily large. To summarize the total loss is given by

$$\begin{aligned}\mathcal{L}_{\text{recons}} &= \|x - \hat{x}\|_2^2 \\ \mathcal{L}_{\text{codebook}} &= \|sg(z) - z_q\|_2^2 + \beta \|z - sg(z_q)\|_2^2 \\ \mathcal{L}_{\text{VQ}} &= \mathcal{L}_{\text{codebook}} + \mathcal{L}_{\text{recons}}\end{aligned}\tag{20}$$

where  $\beta$  is a tuning parameter typically set to be 0.25 and  $sg()$  is the stop-gradient operation, defined to be the identity with zero partial derivatives.

The VQ loss only affects the codebook, and can hence alternatively be updated as a function of moving averages of the encoder outputs  $z$ . This is expanded on in appendix A.1 in [VQVAE].

To illustrate the need for commitment term and what it does consider a model where the data is either 0 or 1 and we initialize the codebook with  $\mathcal{Z} = \{-1, 1\}$ . The range of the encoder is  $\mathbb{R}$  and  $E(x)$  is quantized to  $-1$  if its negative and 1 if its positive. Assume that the encoder and decoder will try to differentiate between

the two classes by pushing  $E(0)$  and  $E(1)$  away from each other. As the reconstruction loss only affects the encoder and decoder, and the codebook loss only affect the codebook, we get that when the encoder and decoder parameter trains faster than the codebook, then the distance between the encodings and the codewords will steadily increase and the codebook loss diverge. This behavior is observed experimentally.

## 0.10 Evaluation metrics

### 0.10.1 Tokenization model

The tokenization model, as we are interested in representation learning, is evaluated on two metrics. Firstly, and most importantly its ability to reconstruct the input data once compressed into latent space. In essence the latent representation encodes "everything" (important information is preserved) about the original data if the model is able to reconstruct well. Secondly we evaluate linear classifiers on the latent representations, which provides good results if the model learns discriminative features of the different classes and produces an approximately linear separable space. Finally, as the tokenization model is a part of the generative model, the ultimate evaluation metric is the corresponding evaluation of the generative model.

#### Reconstruction

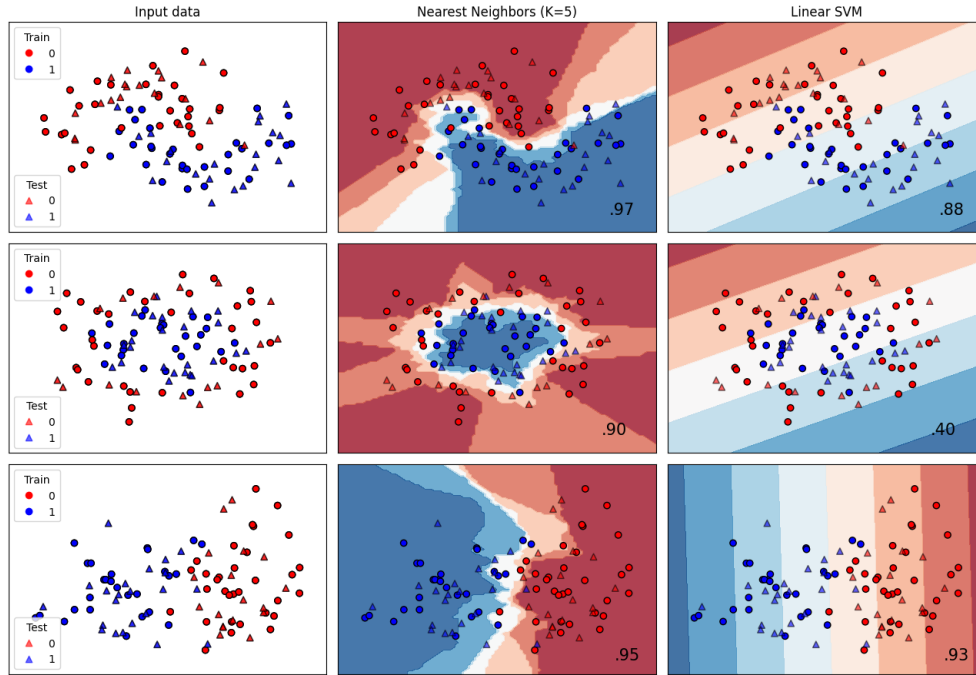
#### Downstream Classification

SVM, KNN. The difference in inductive bias for the two classifiers.

### 0.10.2 Generative model

Good evaluation protocols for generative models are hard to come by, and the hunt for such is an active area of research. There are data modalities for which such evaluation metrics have more established standards than others. Most notably in the computer vision domain there is a benchmark dataset (ImageNet) and classifier (Inception v3).

The typical sanity check for generative models is human visual inspection. In development of evaluation metrics for generative models, how well a proposed metric correlates with quality determined by visual inspection is standard. Most people know what cats and dogs look like and can recognize features even in crudely generated images. In contrast, many time series dataset measure quantities and signals for which the untrained eye has no idea of determining the quality. This is one of the obstacles in TSG. Despite this, according to [TimeVQVAE] the most common evaluation protocols in the TSG literature is PCA and t-SNE analyses



**Figure 17:** Modified example taken from [https://scikit-learn.org/stable/auto\\_examples/classification/plot\\_classifier\\_comparison.html](https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html).

on time series to visually see similarities of two distributions. The major limitations of this is that the visual inspections cannot be reduced to a single score, which makes objective comparison difficult.

Following [TimeVQVAE] we mainly consider three evaluation metrics in this thesis. *Inception Score* (IS), *Fréchet Inception Distance* (FID) and *Classification Accuracy Score* (CAS). IS measures quality of synthetic samples using the entropy of the label distribution and the evenness in prediction across labels. The *Fréchet Inception Distance* measures a distance between the generative and ground truth distribution. Finally the *Classification Accuracy Score* measures the quality of the class-conditionally generated samples by training a classifier on synthetic data and testing on real.

Common for all the above evaluation metrics is the reliance on a pre-trained classification model. In contrast to computer vision there is no widely adopted classification model for time series, but in [wang2016time] a fully convolutional network (FCN) was presented as a proposed baseline classification model for time series. As their model is not available we use the open source FCN trained on the UCR Archive presented in [TimeVQVAE] for our evaluations.

### Inception Score (IS)

Inception Score (IS) was first introduced in [salimans2016improved] as an automatic evaluation method of synthetic samples.

The Inception Score works by applying a classifier to every generated sample to get the conditional label distribution  $p(y|x)$ . Samples containing clear characteristics should have low entropy, i.e one should be certain of the label. In addition we expect a good model to generate varied samples, so the marginal  $\int p(y|x = D(z))dz$  should have high entropy. On this basis, the Inception Score is defined as

$$\text{IS}(\theta) = \exp(\mathbb{E}(D_{\text{KL}}(p_\theta(y|x)||p_\theta(y)))). \quad (21)$$

In contrast to computer vision there is no widely adopted classification model for Time Series. In [wang2016time] a fully convolutional network (FCN) was presented as a proposed baseline classification model, and we use the open source FCN trained on the UCR Archive presented in [TimeVQVAE] for our evaluations.

Let  $X_{\text{gen}} = \{x_{i,\text{gen}}\}_{i=1}^N$  be a set of generated samples. We use the apply a Soft-Max layer to pretrained FCN representations to obtain an estimate of the conditional label distribution as follows

$$x_{i,\text{gen}} \xrightarrow{\text{FCN+SoftMax}} p(y|x_{i,\text{gen}}).$$

The marginal label distribution is obtained by averaging across all the synthetic data as follows

$$p(y) = \frac{1}{N} \sum_{i=1}^N p(y|x_{i,\text{gen}}).$$

Issues with IS [barratt2018note] (Use different network, for image classification). Important to report different metrics that indicate that the model has not overfitted. Also issues with IS [borji2021pros].

### Fréchet Inception Distance (FID)

As an attempt to improve on IS [heusel2018gans] introduced the Fréchet Inception Distance (FID). Since then FID has been the standard for assessing generative models [borji2021pros].

The primary concern with IS is, as mentioned earlier, that it does not use any statistics of real world samples to compare with the statistics of the generated samples. In contrast FID relates the synthetic sample to the real world sample via the *Fréchet distance*. For any two probability distributions,  $f, g$  over  $\mathbb{R}^n$ , with finite

mean and variances, their distance is defined as

$$\begin{aligned} d_F(f, g) &= \left( \inf_{\gamma \in \Gamma(f, g)} \int_{\mathbb{R}^n \times \mathbb{R}^n} \|x - y\|_2^2 d\gamma(x, y) \right)^{\frac{1}{2}} \\ &= \left( \inf_{\gamma \in \Gamma(f, g)} \mathbb{E}_{(x, y) \sim \gamma} \|x - y\|_2^2 \right)^{\frac{1}{2}}, \end{aligned} \quad (22)$$

where  $\Gamma(f, g)$  is the set of all *couplings* of  $f$  and  $g$ . The Fréchet distance is a special case of the Wasserstein metric and will in literature be referred to as such.

In [DOWSON1982450] it was shown that for two Gaussian distributions the Fréchet distance is explicitly solvable as

$$d(\mathcal{N}(\mu, \Sigma), \mathcal{N}(\mu', \Sigma'))^2 = \|\mu - \mu'\|_2^2 + \text{Tr}(\Sigma + \Sigma' - 2(\Sigma\Sigma')^{\frac{1}{2}}) \quad (23)$$

The *Fréchet Inception Distance* is an application of the Fréchet distance on the representation distributions from a specified model. The models are Inception v3 in the image domain and SupervisedFCN in the time series domain.

In [heusel2018gans] they argue that since the Gaussian distribution is the maximum entropy distribution over  $\mathbb{R}^n$  for a given mean and covariance it is a reasonable distribution to assume for the representations. The mean and covariance is estimated from the samples and the explicit formula ?? is used to calculate the FID.

FID is not without fault, and the Gaussian assumption has been shown not to hold [jayasumana2024rethinking]. Further, as the FID relies on estimating large covariance matrices, a large number of samples is needed to obtain a reliable estimate.

Rethinking FID: [jayasumana2024rethinking]

[chong2020effectively] FID and IS are biased.

FID score measures difference in the distribution of representations of time series. The representations capture high level semantics of the time series and analyses of these representations can provide greater insights to the realism of generated samples.

### Classification Accuracy Score (CAS)

A method for evaluating the models ability to learn class conditional distributions is to train a separate classifier on the on synthetic data and test on real data

(TSTR). That is we generate  $N$  synthetic samples  $\{x_i, y_i\}_{i=1}^N$  using class conditional sampling. We then train a classifier on this synthetic dataset, and evaluate the CAS of this classifier on a real test dataset. High CAS values indicate that the generative model produces samples similar to the real data and captures relevant class specific features.

We evaluate the CAS for TSTR by using the Supervised FCN introduced in [TimeVQVAE] on all our models considered and compare against the baseline model to investigate the relative performance.