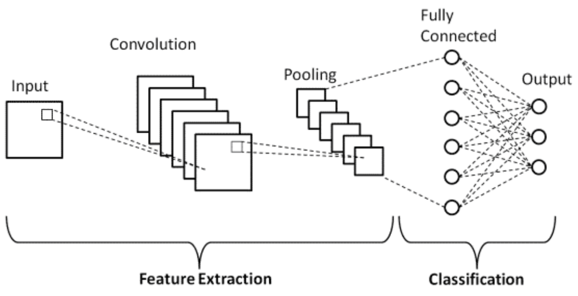
DEEP LEARNING - ARCHITECTURES

***Discriminative Models* (1):** models used to learn to map input data to output labels or categories, aiming to model the decision boundary that separates different classes of data.

***Generative Models* (2):** models used to model the underlying probability distribution of the data and can produce novel data points that share similar characteristics with the training data.

CNN – Convolutional Neural Network (1)



The main operation used in this architecture is the convolution which downsamples the input into a lower representation by extracting its relevant features in an automatic way (from low-level features to high-level features). With convolution two sets of information are merged: the input data (a tensor of matrices) and a learnable filter (tensor of matrices) with weights and biases.

Properties of images:

- **Self similarity** -> images have similar patterns around the whole domain
- **Translation invariance** -> images' contents remain the same even if shifted
- **(Isometric) Deformation invariance**

Properties of convolution:

- **Local Receptive Fields**: helps capture local patterns and features in the data
- **Parameter Sharing**: reduces the number of learnable parameters in the network and the risk of overfitting
- **Translation Invariance** (recall of Circulant Matrix)**:** recognizes patterns regardless of their position in the input
- **Hierarchical Feature Extraction:** learns representations of increasing abstraction
- **Sparse Connectivity:** reduces computational complexity and memory requirements
- **Pooling for Spatial Hierarchies**
- **Data Efficiency**
- **Scalability**
- **Parallelization**

Each convolutional layer consists of a set of learnable filters (also known as kernels). To operate a convolution the filter is slided all over the dimension of the image. The area where the convolution operation takes place (in the input image) is called the **receptive field**. The result of convolution is called **feature map**.

The output dimension O of the convolution between the input data (which size is referred to as I) and the kernel (which size is referred to as K) is:

$$O = \frac{I - K + 2P}{S} + 1$$

P states for *padding*, an expedient thanks to which the input data is filled with values (often zeros) in rows and columns in order to have greater dimensions of the output.

S states for *stride* and represents the number of steps (i.e. columns or rows) we are moving when applying convolutions over windows of the input data. A large number of stride means lower output dimensions.

After convolution layers an **activation function** (typically relu is used in order to convert negative values to zero) is applied to introduce non-linearities in the model (convolutions are linear operations).

Then a **pooling** operation is done over the output(s) of convolution layer(s). It is used between two convolution layers. If we apply FC after Convolution layer without applying pooling, then it will be computationally expensive. So pooling layers are introduced to get lower dimensions i.e. less number of parameters while keeping relevant features. There are 3 types of pooling: max pooling, min pooling and average pooling. Pooling layers do not have learnable parameters themselves.

The output dimension O of the pooling operation (which size is referred to as F) between the input data (which size is referred to as I) is:

$$O = \frac{I - F}{S} + 1$$

Again, S is the stride.

Then, a **fully connected layer** is used typically to classify images. It involves neurons, biases and weights but for fewer input parameters since convolutions and poolings are done.

Softmax or Logistic layer is the last layer of CNN. It resides at the end of FC layer. Logistic is used for binary classification and softmax is for multi-classification.

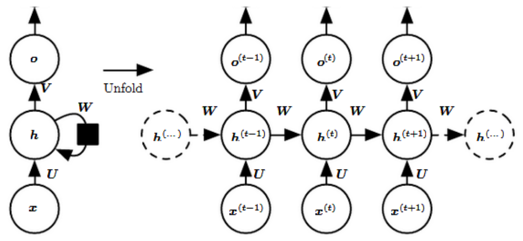
Output layer contains the label which is in the form of one-hot encoded.

Examples:

- LeNet-5 (1988) used for handwritten digit recognition
- AlexNet (2012) achieved a breakthrough in image classification
- VGG-16 (2014) featured a deep architecture with small 3x3 convolutional filters
- GoogleNet (2014) introduced the concept of inception modules, which allowed for the efficient use of different filter sizes for convolution
- ResNet (2015) used residual connections to address the vanishing gradient problem, enabling the training of extremely deep networks

APPLICATIONS: image recognition, object detection, facial recognition, medical image analysis, self-driving cars, and more.

RNN – Recurrent Neural Network (1), (2)



RNNs are types of NN used when the input is a sequence of data. Each item of this sentence feeds the network one at a time.

The **feedforward pass** is done with these 3 main operations:

$h_t = f(Ux_t + Vh_{t-1})$

$o_t = g(Wh_t)$

$L = \sum_{t=1}^{T} L_t$

Where f and g are activation functions.

The learnable parameters are U, V and W so the backpropagation pass wants to update them. Since the formulas have a recursive shape, the computation of derivatives leads to two main problems:

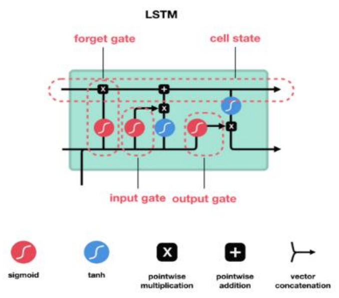
1. Since we're multiplying over and over again by the weight matrix, the gradient will be scaled up or down depending on the largest singular value of the matrix: if the singular value is greater than 1, we'll face an **exploding gradient problem**, and if it's less than 1, we'll face a **vanishing gradient problem**
2. The gradient passes through the tanh non-linearity which has saturating regions at the extremes. It means the gradient will essentially become zero if it has a high or low value once it passes through the non-linearity — so **the gradient cannot propagate effectively across long sequences** and it leads to ineffective optimization

ADVANTAGES: Sequential Data Handling, Temporal Dependencies, Variable-Length Sequences, Stateful Processing, Real time processing, …

DISADVANTAGES: difficulties in capturing very long-term dependencies and vulnerability to the vanishing gradient problem

APPLICATIONS: speech recognition, voice recognition, time series prediction, and natural language processing

LSTM – Long Short Term Memory (1), (2)



LSTM are special types of RNN capable of learning long-term dependencies thanks to a special internal architecture consisting of a **cell state**, a **forget gate**, an **input gate** and an **output gate**.

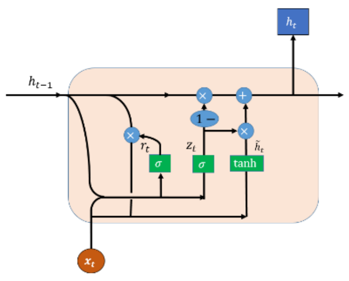
1- The **forget gate** decides which information is going to be thrown away given the current input and the previous hidden state. Its formula is:
$f_t = \sigma\left(W_f h_{t-1} + U_f x_t + b_f\right)$ and outputs a number between zero (forget information) and one (keep information).

2- The **input gate** is used to understand which new information will be stored in the cell state. Its formula is:
$i_t = \sigma\left(W_{i^{h_{t-1}}} + U_i x_t + b\right)_i$ → determines which values are going to be updated
$\tilde{c}_t = \tan h(W_c h_{t-1} + U x_t + b_c)$ → creates a vector of new candidates to be added in the cell state

3- In the cell state the information is updated:
$c_t = f_t * c_{t-1} + i_{t^*}\tilde{c}_t$ → the first term allows to forget information decided at the first step, the second one adds new information (candidates) scaled by how much we decided to update each state value

4- The **output gate** decides what is going to be output:
$o_t = \sigma(w_0(r_t \circ h_{t-1}) + U_0 x_t + b_0)$ → decides what parts of the cell state are going to be output
$h_t = o_{t^*} \tanh(c_t)$

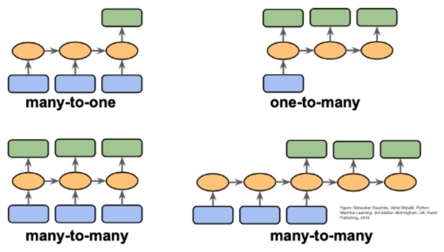
GRU – Gated Recurrent Unit (1), (2)



GRU is a type of RNN using four components to learn relations within sequence of data: the **update gate**, the **reset gate**, the **current memory content** and the **final memory content** at current time step

1- The update gate helps the model to determine how much the past information needs to be passed along to the future:
   $z_t = \sigma(W_z x_t + U_z h_{t-1})$
2- The reset gate decides how much of the past information is going to be forgotten
   $r_t = \sigma(W_r x_t + U_r h_{t-1})$
3- The current memory content is used to store the relevant information from the past provided by the reset gate
   $h'_t = tanh(W x_t + r_t\ o\ U h_{t-1})$
4- The final memory at current time step stores the information for the current unit and passes it down to the network
   $h_t = z_t \circ h_{t-1} + (1 - z_t) \circ h'_t$

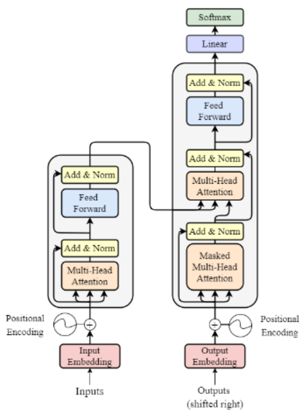
Seq2Seq


many-to-one
one-to-many
many-to-many
many-to-many
Figure: Sebastian Raschka, Vahid Mirjalili. Python Machine Learning. 3rd Edition. Birmingham, UK: Packt Publishing, 2019

They are a class of neural network architectures designed for tasks that involve sequences as input and output. They have become widely used in various natural language processing (NLP) and sequence-to-sequence learning tasks. Seq2Seq models consist of two main components: an encoder and a decoder.

- Many to one: document classification, sentiment analysis, …
- One to many: text generation, image captioning, …
- Many to many: machine translation, speech recognition, …
- One to one: regression, pattern recognition, …

TRANSFORMERS (1), (2)



The main characteristic of a transformer is *the attention mechanism* which allows the transformer to (1) capture relationships within words in a potentially infinite temparal window. Also, in transformer's architecture (2) all the input is ingested once contrary to RNN which gets the input sequence item by item.

These two aspects (1), (2) make transformers better, faster and more used than RNNs.

The **architecture** is made up of an encoder and a decoder.

Input Embedding (input->embedding layer)

We want to represent the input (in case of text, each word of the input) into a vector with continous values.

In order to know the position of each word with respect to the others we need a **positional embedding** which adds this information to the vector containing the representation of that particular word. This is necessary since all the input is passed once, not sequentially (word by word) as in RNNs.

To do this, sines and cosines are used. For every odd index on the input vector, create a vector using the cos function. For every even index, create a vector using the sin function. Then add those vectors to their corresponding input embeddings.

1. **Encoder**

The first layer of the encoder is called "multi-head self attention" and involves the self attention mechanisms using three main properties: queries, keys and values. Queries, keys and values are vectors obtained by fully connected layers. Then once Q, K and V are computed they are passed through a linear layer. A dot product between Q and K (two vectors) provides the "score" matrix which reveals how much a word should focus on other words. To have more stable gradients, this matrix is scaled by a fixed factor (square of the dimension of query and key vector). Then a softmax operation is applied to map each value into a probablity (between 0 (no focus) to 1 (focus)). These values are also called attention weights. Once the attention weights are computed, they are multiplied with the value vectors. This operation produces vectors containing higher score for words considered relevant and lower score to for those with less importance. At the end these output vectors are passed through a linear layer.

This process (which is called head) is done N parallel times by splitting Q, K and V and each branch output is concatenated. For this reason it takes the name "multi-head attention". It is useful since each of this N heads can focus on different aspects.

Then, the output of the multi-head attention layer is added to the initial positional embeddings with a residual connection and then normalized.

The normalized residual output gets projected through a pointwise feed-forward network for further processing. The pointwise feed-forward network is a couple of linear layers with a ReLU activation in between. The output of that is then again added to the input of the pointwise feed-forward network and further normalized.

2. **Decoder**

It has two multi-headed attention layers, a pointwise feed-forward layer, and residual connections, and layer normalization after each sub-layer.

Its job is generating text starting from a <start> token and ending when a <end> token is generated.

The beginning of the decoder is similar to the one of the encoder. The inputs are passed through an embedding layer which converts the inputs into continous vectors and the the positional information are added to each of those vectors. The positional embedding then goes through the first multi head attention layer. The decoder should not have the possitibility to look at future words since it is autoregressive.

To prevent this a look a-head mask is used which is made up of zeros and -infinities so that once the scaled score matrix is added the resulting matrix is top right triangle filled with negativity infinities. Once the softmax operation is applied, then the top right triangle will contain only zeros (no importance to future words). So, the output of the first multi-headed attention is a masked output vector with information on how the model should attend on the decoder’s input.

Again, this process is done over multiple heads which outputs are concatenated at the end in a single output and then passed through a linear layer.

The second multi-headed attention layer. For this layer, the encoder's outputs are the queries and the keys, and the first multi-headed attention layer outputs are the values.

This process matches the encoder's input to the decoder's input, allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

The output of the final pointwise feedforward layer goes through a final linear layer, that acts as a classifier. The classifier is as big as the number of classes you have. The output of the classifier then gets fed into a softmax layer, which will produce probability scores between 0 and 1. We take the index of the highest probability score, and that equals our predicted word.

The decoder then takes the output, add's it to the list of decoder inputs, and continues decoding again until a token is predicted. For our case, the highest probability prediction is the final class which is assigned to the end token.

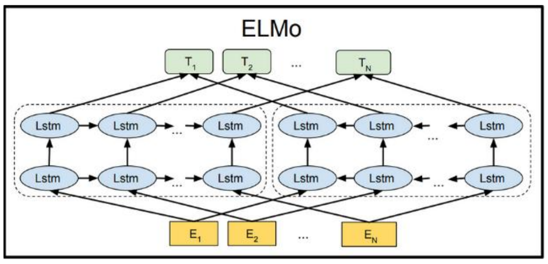
ADVANTAGES: Parallelization, Long-Range Dependencies, Self-Attention Mechanism, Scalability

DISADVANTAGES: Computational Resources, Data Requirements, Training Time, Fine-Tuning Complexity, Lack of Sequential Memory, Interpretability Challenges, Overparameterization

APPLICATIONS: document summarization, document generation, biological sequence analysis, and video understanding, …

Word2Vec: it is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. **Continuous bag-of-words model** predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important. **Continuous skip-gram model** predicts words within a certain range before and after the current word in the same sentence.

ELMo - (Embeddings from Language Models)



it is a deep learning model used for natural language understanding and natural language processing (NLP) tasks. ELMo is part of a class of models that produce contextual word embeddings, which means the word embeddings vary depending on the context of the word within a sentence.

Elmo uses bi-directional LSTM in training, so that its language model not only understands the next word, but also the previous word in the sentence. It contains a 2-layer bidirectional LSTM backbone. The residual connection is added between the first and second layers. Residual connections are used to allow gradients to flow through a network directly, without passing through the non-linear activation functions. The high-level intuition is that residual connections help deep models train more successfully.

BERT - Bidirectional Encoder Representations from Transformers (1), (2)

BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. It can perform well on different language tasks if fine-tuned because it can understand words, so the core of language.

Since BERT's goal is to generate a language model, only the encoder mechanism is necessary. So encoders are stacked on top of each other and allow to learn the context of language.

The Transformer encoder reads the entire sequence of words at once. Therefore it is considered bidirectional, though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

The input embedding contains:

- Positional embedding: the information of the position of each word
- Sentence embedding: the information with which we distinguish parts belonging to the same sentence
- Token embedding: the representation of each word in a numerical way

TRAINING BERT

The architecture itself understands the language "generically". In order to apply it on different tasks, two steps are done: Masked Language Modelling and Next Sentence Prediction.

1. Masked Language Modelling
   We get a sentence and inside it 15% of the words are masked (so letf blank) and the goal is to predict which words should be fill the blanks.
2. Next Sentence Prediction
   Given pairs of sentences the model has to predict if the sentences belong together or not

If neither of these two tasks want to be performed, then we need to fine-tune BERT.

FINE-TUNING BERT

To fine-tune BERT we need (1) a new **output layer** plugged at the end of the bert model, (2) a new set of data (specific to the task).

GPT (2)

GPT is a decoder-stacked architecture each with feed forward neural network and masked self-attention.

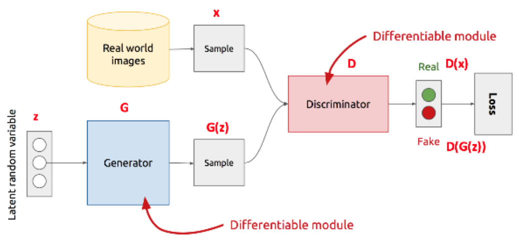
VISUAL TRANSFORMERS (1)

In ViTs, images are represented as sequences, and class labels for the image are predicted, which allows models to learn image structure independently. Input images are treated as a sequence of patches where every patch is flattened into a single vector by concatenating the channels of all pixels in a patch and then linearly projecting it to the desired input dimension.

Let's examine the vision transformer architecture step by step.

1. Split an image into patches
2. Flatten the patches
3. Produce lower-dimensional linear embeddings from the flattened patches
4. Add positional embeddings
5. Feed the sequence as an input to a standard transformer encoder
6. Pretrain the model with image labels (fully supervised on a huge dataset)
7. Finetune on the downstream dataset for image classification

GAN – Generative Adversarial Network (2)



This architecture is made up of two main components: the **Generator** and the **Discriminator** competing against each other**.** The aim of this game is to generate new samples by learning some probability distribution describing the real data.

The game formulation is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

- The discriminator wants to maximize it
- The generator wants to minimize it

1. The generator's input is a latent random variable z (noise). The generator neural network will produce from z new samples that are going to feed the discriminator. Its aim is to fool the discriminator generating samples similar to the real ones.
2. The discriminator receives as input both the "fake" generated data and the real data coming from the dataset on which the generator tries to learn the distribution. Discriminator neural network's task is classifying the samples as real or fake. If it classifies correctly it helps the generator to improve samples more similar the the real ones.

Their trainings are separated: when the Generator (discriminator) is training, the discriminator (generator) is frozen.

The discriminator loss is:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right]$$

The discriminator penalizes itself for misclassifying a real instance as fake, or a fake instance (created by the generator) as real, by maximizing this function.

The generator loss is:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

The generator loss is calculated from the discriminator's classification – it gets rewarded if it successfully fools the discriminator, and gets penalized otherwise.

This game ends when the *Nashium Equilibrium* is reached so when these two conditions are satisfied:

- $P_{data}(x) = P_{gen}(x)\ \forall x$
- $D(x) = \frac{1}{2}$

ADVANTAGES: High quality data generation, versatility, …

DISADVANTAGES:

More often than not, GANs tend to show some inconsistencies in performance:

1. **Mode collapse**: typically we would expect the generator to produce samples which differ (ex. Different faces) but the network learns to model a particular distribution of data, which gives us a monotonous output because the generator is always trying to find the one output that seems most plausible to the discriminator.

2. **Vanishing gradients**: This phenomenon happens when the discriminator performs significantly better than the generator. Either the updates to the discriminator are inaccurate, or they disappear.
3. **Convergence:** The utopian situation where both networks stabilize and produce a consistent result is hard to achieve in most cases. One explanation for this problem is that as the generator gets better with next epochs, the discriminator performs worse because the discriminator can't easily tell the difference between a real and a fake one.

APPLICATIONS: Image Generation and Synthesis, Image-to-Image Translation, Style Transfer, Data Augmentation, Deepfake Generation and Detection, Super-Resolution, …

WGAN – Wasserstein Generative Adversarial Network (2)

WGans tackle the problem of Mode Collapse and Vanishing Gradient by using a more stable loss function called the Wasserstein distance which measures how much "work" is needed to transform one distribution (of fake data) into another (real data distribution).

In WGAN, the discriminator (often referred to as the "critic" in this context) plays a different role compared to traditional GANs.

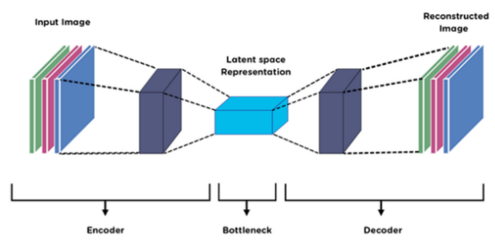
Instead of outputting a probability (like in traditional GANs), the critic in WGAN outputs a real-valued score. This score is interpreted as the estimated Wasserstein distance between the real and fake data distributions. To ensure that the critic can estimate the Wasserstein distance accurately, WGAN introduces a Lipschitz continuity constraint on the critic's output. Specifically, it enforces that the critic's output has a Lipschitz constant (a bound on its gradient).

The cost function in WGAN is based on the difference between the critic's scores for real and fake data. The generator tries to minimize this difference, while the critic aims to maximize it. The training process involves iteratively updating the generator and critic in a way that brings the critic's scores for real and fake data distributions closer together (i.e., minimizing the Wasserstein distance).

During training, the generator and critic are updated alternately. The generator aims to generate data that is more similar to the real data distribution, while the critic tries to better estimate the Wasserstein distance.

The training continues until a convergence criterion is met, such as a fixed number of iterations or a stable Wasserstein distance.

AE – AutoEncoder (1), (2)



Autoencoders are architectures made up of an encoder and a decoder. The encoder neural network downsamples the input into a lower dimensional representation z by mapping data to single points and the decoder neural network upsamples z to the original dimension of the input. They are used for unsupervised learning and capture deep correlations between data through dimensionality reduction.

1. The encoder learns a function g parametrized by θ which downsamples x into its lower dimensional representation $z = g_\theta(x)$
2. The decoder learns a function f parametrized by ϕ which upsamples z into the reconstruction of the input $x' = f_\phi(z) = f_\phi\big(g_\theta(x)\big)$

3. The output of this architecture is to have x' as close as possible to x which can be done by minimizing the Reconstruction Loss function:
$$L_{AE} = \frac{1}{n}\sum_{i=1}^{n}(x_i - f_\phi(g_\theta(x)))^2$$

ADVANTAGES: dimensionality reduction, unsupervised learning, anomaly detection, and feature learning

DISADVANTAGES: overfitting, architecture design, interpretability of latent representations, and their (in)ability to model complex data distributions

APPLICATIONS:

Denoising Autoencoder: it is a modification on the autoencoder to prevent the network learning the identity function. Specifically, if the autoencoder is too big, then it can just learn the data, so the output equals the input, and does not perform any useful representation learning or dimensionality reduction. Denoising autoencoders solve this problem by corrupting the input data on purpose, adding noise or masking some of the input values.

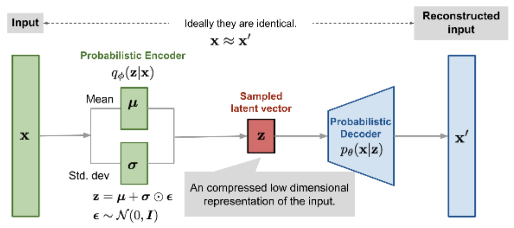
Sparse Autoencoder: it is a type of autoencoder that employs sparsity to achieve an information bottleneck. Specifically the loss function is constructed so that activations are penalized within a layer. The sparsity constraint can be imposed with L1 regularization or a KL divergence between expected average neuron activation to an ideal distribution p.

Anomaly Detection: since autoencoders are not robust to variations and anomalies can be seen as variations the loss for anomal inputs is higher with respect of the loss produced by normal samples on which the AE has been trained.

WHY AEs ARE BETTER THAN PCA?

PCA (principal component analysis) tries to find lower-dimensional orthogonal hyperplanes that describe the original data by capturing the maximum possible variance in the data and the important correlations. Since we're finding a hyperplane, then we're capturing linear correlations. But often correlations are non-linear, so they are not covered by PCA.

<u>**VAE – Variational AutoEncoder (2)**</u>


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
Std. dev
$\boldsymbol{\sigma}$
$\mathbf{x}$
Sampled latent vector
$\mathbf{z}$
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \epsilon$
$\epsilon \sim \mathcal{N}(0, \boldsymbol{I})$
An compressed low dimensional representation of the input.

VAEs are generative model achitectures based on two key principles: the encoder-decoder structure and the probabilistic latent variable models basics. The main purpose of this architecture is to generate new samples that resemble a given dataset.

Differently to classical autoencoders, VAEs don't map the input x into single points but into probability distributions (Gaussian most of cases). For this reason both the encoder and decoder are called probabilistic.

1. Probabilistic Encoder
In this case of study, we assume that x, the input data of the encoder, is generated from a hidden latent variable z which follows a Gaussian distribution.
The generation process follows two steps:

- z is sampled from a prior distribution p(z)
- x is sampled from the conditional likelihood distribution p(x|z)

What the encoder wants to find is the conditional probability p(z|x) parametrized by Θ, but its computation is intractable. So we use a method called *Variational Inference* (that's where the name *variational* autoencoder comes from) that allows to approximate p(z|x) with another conditional probability distribution q(z|x) parametrized by ϕ .
In order to compute the probability distribution q as close as possible to p, the KL-Divergence is used which provides a measure of similarity between q and p.
The formula is:

$$D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) = \log p_\theta(\mathbf{x}) + D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z})) - \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\log p_\theta(\mathbf{x}|\mathbf{z})$$

$$\log p_\theta(\mathbf{x}) - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}|\mathbf{x})) = \mathbb{E}_{\mathbf{z}\sim q_\phi(\mathbf{z}|\mathbf{x})}\log p_\theta(\mathbf{x}|\mathbf{z}) - D_{\mathrm{KL}}(q_\phi(\mathbf{z}|\mathbf{x})\|p_\theta(\mathbf{z}))$$

And we get the loss L:

$$\mathcal{L}(\theta,\phi;\mathbf{x},\mathbf{z}) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction Loss}} - \underbrace{D_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z})\right)}_{\text{Loss stays close to Gaussian distribution (0,1)}}$$

The "*reconstruction loss*" encourages the model to generate data samples that are close to the original input data. The KL Divergence term is called "regularization term" and encourages the model to have a well-behaved latent space structure that match the prior distribution, making the latent space more interpretable and suitable for generating new data samples.
So the learnable parameters of the encoder are updated with the intention to minimize this "Elbo" loss. The name ELBO stands for **E**vidence **L**ower **BO**und and comes from the fact that, since the KL Divergence is non negative, -L is the lower bound of log($p_\theta(x)$).
Therefore by minimizing the loss, we are maximizing the lower bound of the probability of generating real data samples.

The **outputs** of the encoder are the mean μ and the variance σ of this distribution q.
The generation of a new sample involves the sampling process which is *stochastic* and represents a problem in backpropagation since it's not differentiable. So a reparametrization trick is used to sample z from the distribution q in a way that it is differentiable.
z can be written as z = μ + ε * σ where ε $\sim N(0, I)$

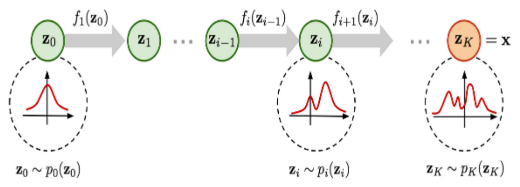
2. Probabilistic Decoder
   The probabilistic decoder is responsible to generate new samples from the latent state representation. Its input is z (a sample from the latent space, typically drawn from a multivariate Gaussian distribution).
   By parameterizing a probability distribution for each data point, the decoder accounts for uncertainty in the data generation process. This probabilistic nature enables the VAE to capture variations in the data and produce diverse and meaningful samples during the generative phase while maintaining the learned latent space structure.

ADVANTAGES: generative modeling, latent space representation, uncertainty estimation

DISADVANTAGES: expressiveness, interpretability, potential for generating high-quality samples

APPLICATIONS: Image Generation, Data Denoising, Anomaly Detection, Imputation and Inpainting, Semi-Supervised Learning, Text Generation, Dimensionality Reduction, …

NORMALIZING FLOWS (2)


$f_1(\mathbf{z}_0)$
$\mathbf{z}_0$
$\mathbf{z}_1$
$f_i(\mathbf{z}_{i-1})$
$\mathbf{z}_{i-1}$
$\mathbf{z}_i$
$f_{i+1}(\mathbf{z}_i)$
$\mathbf{z}_K = \mathbf{x}$
$\mathbf{z}_0 \sim p_0(\mathbf{z}_0)$
$\mathbf{z}_i \sim p_i(\mathbf{z}_i)$
$\mathbf{z}_K \sim p_K(\mathbf{z}_K)$

Normalizing flows model is a type of generative model which aims to transform a simple probability distribution (often a Gaussian) into a more complex one resembling the real data distribution while maintaining tractable calculations of probability densities.

The idea of "flow" comes from a process made up of steps: at each step i a latent vector z_i is sampled from a distribution p(z_i) (Gaussian most of the cases) and a known invertible and parametrized, by learnable paramters, function f_i is applied on z_{i-1} so that $z_i = f(z_{i-1})$.

At the last step k, $z_k = x$ which is the element of the initial distribution (training data).

Since the functions f_i are invertible we can also get z_{i-1} = $f^{-1}(z_i)$ from $z_i = f(z_{i-1})$. Each transformation in the flow is bijective, so it has an easily computable inverse. This invertibility ensures that you can map samples from the target distribution back to the simple distribution.

The primary training objective in Normalizing Flow models is to maximize the likelihood of the observed data under the model. This is typically done using maximum likelihood estimation (MLE). The loss function used for training a Normalizing Flow model is typically the negative log-likelihood (NLL) of the observed data: $L(D) = -\frac{1}{|D|}\sum_{x\in D} log(x)$. During training, the model's parameters (i.e., the parameters of the transformations) are updated iteratively to improve the likelihood of the observed data.

Once trained, the model can generate new data samples by sampling from the simple distribution and applying the learned transformations.

ADVANTAGES: Don't need to put noise on the output***,*** Training process more stable***,*** Easy convergence

DISADVANTAGES*:* Not expressive*,* High dimensional latent space*,* Not high quality generated samples

APPLICATIONS: Generative Modeling, Density Estimation, Variational Inference, Anomaly Detection, Image Super-Resolution, …

REAL NVP - Real-valued Non-Volume Preserving (2)

Real NVP are specific types of normalizing flows models. They implement the flow mechanism by stacking a sequence of invertible bijective transformation functions. Each bijection is an affine coupling layer where the input dimensions are split into two parts: the first d dimensions stay same while the second part, d+1 to D dimensions, undergo an affine transformation ("scale-and-shift") and both the scale and shift parameters are functions of the first d dimensions.

$$\begin{cases} \mathbf{y}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d}) \end{cases} \Leftrightarrow \begin{cases} \mathbf{x}_{1:d} &= \mathbf{y}_{1:d} \\ \mathbf{x}_{d+1:D} &= (\mathbf{y}_{d+1:D} - t(\mathbf{y}_{1:d})) \odot \exp(-s(\mathbf{y}_{1:d})) \end{cases}$$

NICE – Non-linear Independent Component Estimation (2)

It is the predecessor of RealNVP and the difference between the two is that NICE affine coupling layers don't have the scale term.

DIFFUSION MODELS (2): Diffusion models are a class of generative models used in machine learning and probabilistic modeling to capture and generate complex data distributions, such as images, text, or audio. They are based on the concept of a diffusion process, which describes how a data distribution evolves over time through a series of transformations.