OPTIMIZATION AND LINEAR MODELS

LOSS FUNCTION

Quantifies the deviation/error between the measured value of y and that which is predicted. We adopt nonnegative function the objective of our main problems are to minimize the total loss.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}\left(y_n, f_{\boldsymbol{\theta}}\left(\mathbf{x}_n\right)\right) \qquad \boldsymbol{\theta}_* = \min_{\boldsymbol{\theta} \in \Omega} J(\boldsymbol{\theta})$$

GRADIENT DESCENT

$$\nabla J(\boldsymbol{\theta}) = \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \left[\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_1}, \ldots, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_M}\right]^{\mathsf{T}}$$

Gradient is when we have two or more derivatives of the same function. Descent -> to descent to the lowest point in the loss function

STOCHASTIC GRADIENT DESCENT

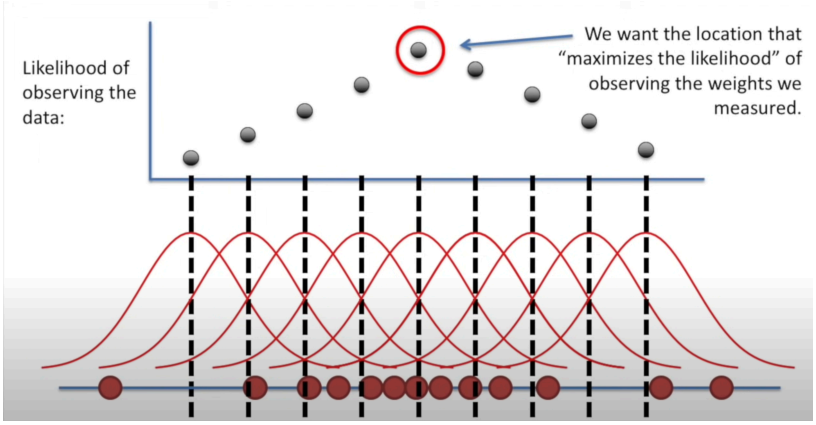
If we have larger dataset we have larger number of derivative to compute, so in the end is quite inefficient. In stochastic gradient descent we take a random subset (mini-batch) of the training data for computing. SGD is less computationally expensive but can be sensitive for not optimality.

OVERFITTING

is when a machine learning model learns too much from the training data, including the random noise, making it perform poorly on new data. It's like memorizing answers to a test rather than understanding the material. (Train loss close to 0 - test loss large)

MAXIMUM LIKELIHOOD

is a method used to estimate the parameters of a statistical model. It finds the parameter values that maximize the likelihood of the observed data under the model. Essentially, it selects the parameters that make the observed data most probable.

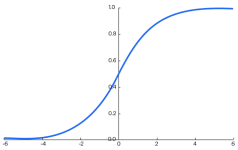


CLASSIFICATION

Classification problem we have just to calculate a categorical probability distribution.
class= arg max [y]

SOFTMAX FUNCTION

$$[\text{softmax}(\mathbf{a})]_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$



CROSS-ENTROPY LOSS

is a measure used to evaluate how well a classification model's predictions match the actual class labels. It quantifies the difference between two probability distributions - the true labels and the predicted probabilities. The goal is to minimize this loss, meaning the predicted probabilities are as close as possible to the actual labels.

LOG-LIKELIHOOD LOSS

is the natural logarithm of the likelihood function. In practice, it's often used instead of the likelihood because it simplifies the mathematics involved in parameter estimation. Like the likelihood, it helps to find the parameter values that make the observed data most probable, but it transforms the product of probabilities into a sum, which is easier to handle computationally.

LOGISTIC REGRESSION

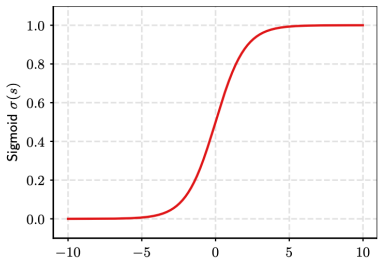
Is linear model trained by optimizing the cross-entropy:

$$\text{LR}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n} \text{CE}\left(\mathbf{y}_i, f(\mathbf{x}_i)\right)$$

BINARY CLASSIFICATION

Softmax is simplified

$$\sigma(s) = \frac{1}{1+\exp(-s)}$$



BINARY LOGISTIC REGRESSION MODEL

$$\text{BIN-LR}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\Big[\underbrace{-\,y_i \log\left(\sigma(\mathbf{w}^\top \mathbf{x})\right)}_{\text{Class 1}} \underbrace{-\,(1-y_i)\log\left(1-\sigma(\mathbf{w}^\top \mathbf{x})\right)}_{\text{Class 2}}\Big] \qquad \sigma'(s) = \sigma(s)(1-\sigma(s))\,.$$

$$\nabla\text{BIN-LR}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i)\mathbf{x}_i\,,$$

SHALLOW NEURAL NETWORKS

COVER'S THEOREM

"In a high-dimensional space, data points that are not linearly separable in a lower-dimensional space are more likely to be linearly separable."
In simpler terms, if you can't separate your data points with a straight line (or hyperplane) in a low-dimensional space, you might be able to do so in a higher-dimensional space.

SHALLOW NEURAL NETWORKS

is a type of artificial neural network that consists of only a few layers, typically one or two hidden layers between the input and output layers.

MULTILAYER PERCEPTRON

To go deeper we just stuck many-fully connected layers on top of each other, so we get an architecture called Multilayer-perceptron. Adding **extra layers does not lead** to something "deeper" , because input layer and hidden and output are linear function of each others, so we can collapse it into a linear transformation.

NON LINEARITY

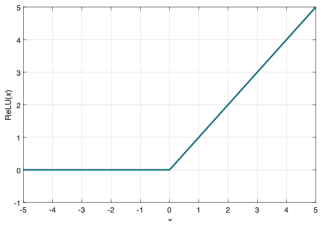
To add non linearity we add non linear activation function to every hidden layer.

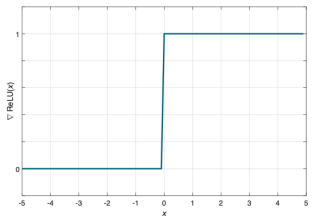
ACTIVATION FUNCTIONS

It decides whether a neural should be active or not, they are differentiable operators to transform signals to outputs, and most of them add non linearity.

RELU

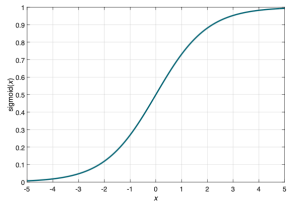
$$\mathrm{ReLU}(z) = \max(z, 0).$$

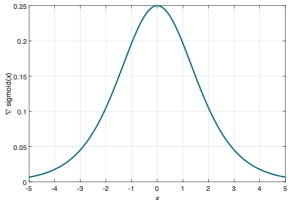




SIGMOID

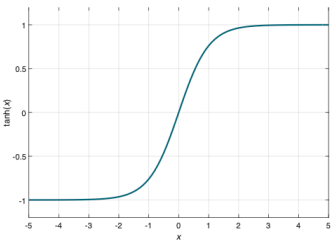
$$\mathrm{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

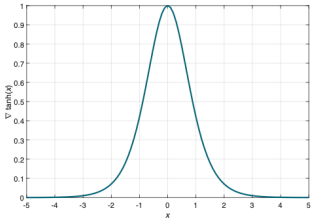




HYPERBOLIC

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$





AUTOMATIC DIFFERENTIATION

FORWARD MODE

We know Neural networks are composed by a variable number of blocks, which in the last one we have a sum and an output scalar.
When estimating the gradients step-by-step starting with the first-layer, we have matrix multiplication from layer to layer, but we can discard previous estimates making it highly efficient. But matrix multiplications make it time consuming.

REVERSE MODE

We know at the end of all computations we have scalar values, so step-by-step starting at the end we have a matrix x scalar multiplication, so requires less time than the forward mode, but due to storing all intermediate output is required a lot of memory.

VECTOR-JACOBIAN PRODUCTS

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \partial_{\mathbf{x}}^{\top} f(\mathbf{x}, \mathbf{w}) \cdot \mathbf{v}\,,$$
$$\text{vjp}_{\mathbf{w}}(f, \mathbf{v}) = \partial_{\mathbf{w}}^{\top} f(\mathbf{x}, \mathbf{w}) \cdot \mathbf{v}$$

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \phi'(\mathbf{x}) \odot \mathbf{v}$$

They are easier to compute than standard Jacobians,

VANISHING AND EXPLODING GRADIENTS

If $\phi'(\cdot) < 1$ always, the gradient will go to zero exponentially fast in the number of layers (vanishing gradient).

If $\phi'(\cdot) > 1$ always, the gradient will explode exponentially fast in the number of layers (exploding gradient).

CONVOLUTIONS

CONTINUOUS-TIME CONVOLUTION

$$[f \circledast g](t) =< f(\tau), g(t-\tau) >= \int_{\mathbb{R}^d} f(\tau) g(t-\tau) d\tau.$$

DISCRETE-TIME CONVOLUTION

$$< f[n], g[n-k] >= \sum_{\mathbb{R}^d} f[k] g[n-k]$$

2D ARRAYS

$$< f[i,j], g[i-k_1, j-k_2] >= \sum_{k_1 \in \mathbb{R}^d} \sum_{k_2 \in \mathbb{R}^d} f[k_1, k_2] g[i-k_1, j-k_2]$$

TRANSLATION INVARIANCE

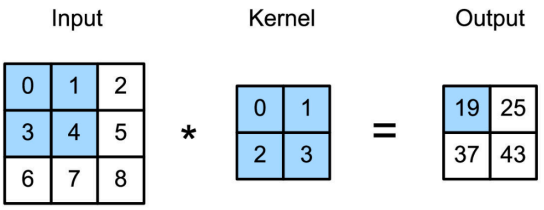
$$h[i,j] = u + \sum_{k_1, k_2} W[k_1, k_2] \cdot x[i+k_1, j+k_2]$$

LOCALITY PRINCIPLE

$$h[i,j] = u + \sum_{k_1=-\Delta}^{\Delta} \sum_{k_2=-\Delta}^{\Delta} W[k_1, k_2] \cdot x[i+k_1, j+k_2]$$

CONVOLUTIONAL NEURAL NETWORKS

KERNEL/ FILTER - Cross-Correlation Operation



Kernels generally have width and height greater than 1, thus, after applying many successive convolutions, the output results much smaller than the input. (**Padding**)
If we want to reduce the resolution drastically if we find an original input resolution to be unwieldy.(**Strides**)

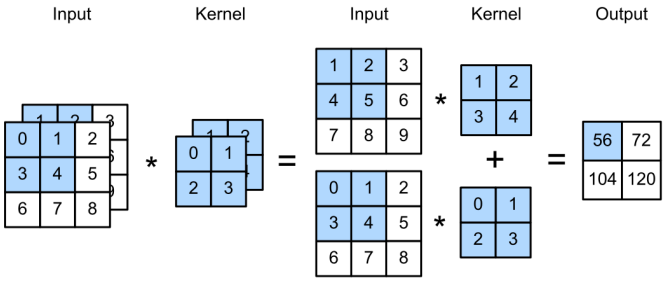
Odd kernel height and width to preserve spatial dimensionality, we can padding the same number of row and column

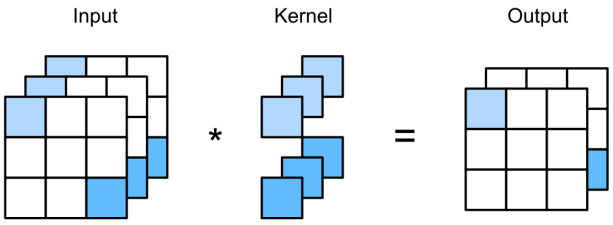
STRIDE

Refers to the number of pixels by which the filter or kernel moves across the input image during convolution operations. It controls how much the filter shifts when it moves to the next position.

CROSS-CORRELATION MULTIPLE INPUT CHANNELS

When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data.

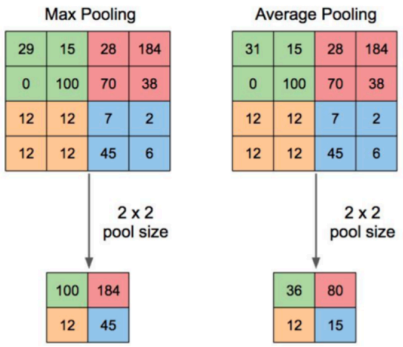




CROSS-CORRELATION MULTIPLE OUTPUT CHANNELS

POOLING OPERATORS

Consist of a fixed-shape window that is slid over all regions in the input according to its *stride*, computing a single output for each location traversed by the fixed-shape window
(Max pooling - Average Pooling)



DILATED CONVOLUTIONS

Which expands the receptive field without loss of resolution. The expansion of the filter allows to increase its dimensions by *filling the empty positions* with zeros.

$$y[i,j] = \sum_{k_1=0}^{M_1-1} \sum_{k_2=0}^{M_2-1} h[k_1,k_2]\, x[i - d \cdot k_1, j - d \cdot k_2]$$

CNN ARCHITECTURE

-Interleave multiple convolutional / max-pooling layers;
-Vectorize (*flatten*) the output of the last layer;
-Process the resulting vector with one or more fully-connected layers to obtain the final classification vector.

OPTIMIZATION STRATEGIES

DATA AUGMENTATION

is a technique to *virtually* increase the size of the dataset at training time:
- Sample of mini-batch of examples;
- For each example, apply one or more transformations randomly sampled (e.g., flipping, cropping, ...).
- Train on the transformed mini-batch.

Data augmentation can be extremely helpful for overfitting, making the network more robust to small changes in the input data

EARLY STOPPING

Is a procedure to find the optimal number of iterations (supposing a single descent curve):
- Keep a portion of the dataset as the validation set.
- For each epoch, check the validation loss (or accuracy).
  Whenever validation loss is not improving for a while (a certain number of epochs called patience), stop the optimization process.
- Early stopping is extremely common in neural networks; it highlights the difference between pure optimization and learning.

REGULARIZATION

To avoid large weights (overfitting), Regularization forces the optimization to select a network with smaller weights by penalizing large norms

$$\mathbf{w}^* = \arg\min\left\{\sum_i l(f(x_i), y_i) + \lambda \cdot \|\mathbf{w}\|^2\right\},$$

Lambda is an hyper-parameter where is zero when we have no regularizations, Otherwise if it is too large.

WEIGHT DECAY

In the absence of the first term, the weights would *decay* exponentially to zero. In pure SGD, this form of regularization is also called weight decay.

$$-\text{Gradient of loss} = -\nabla\left[\sum_i l(f(x_i), y_i)\right] - 2C\mathbf{w}$$

DROPOUT

Dropout extends this idea to the network itself: instead of perturbing the images, we perturb the hidden layers by randomly *dropping* (removing) some of the connections.

DROPOUT REGULARIZATION

Define $\underset{(b,f)}{\mathbf{H}}$ as the output of a generic fully-connected layer having f units, being fed with b inputs (mini-batch size).
With dropout, during training we replace it with:

$$\widetilde{\mathbf{H}} = \mathbf{H} \odot \mathbf{M}\,, \tag{9}$$

where $\mathbf{M}$ is a binary matrix with entries drawn from a Bernoulli distribution with probability p (i.e., $M_{i,j}$ is 0 with probability p, 1 with probability $(1-p)$).
If $M_{i,j} = 0$, the value $H_{i,j}$ is replaced with 0.

DATA NORMALIZATION

Given a dataset $\underset{(n,d)}{\mathbf{X}}$, it is common in machine learning to preprocess it so that each column has mean zero and unitary variance (z-scaling or standard scaling):

$$\mathbf{X}' = \frac{\mathbf{X} - \mu}{\sqrt{\sigma^2}}\,, \tag{12}$$

where $\mu, \sigma^2 \in \mathbb{R}^d$, $\mu_j = \frac{1}{n}\sum_i \mathbf{X}_i$ and $\sigma_j^2 = \frac{1}{n}\sum_i (\mathbf{X}_{ij} - \mu_j)^2$ are the empirical mean and variance. For example, we can do this in scikit-learn with `StandardScaler`. For many opti-

BATCH NORMALIZATION

Consider now a generic output $\underset{(b,f)}{\mathbf{H}}$ of a fully-connected layer (with batching). First, compute the empirical mean and variance column-wise:

$$\tilde{\mu}_j = \frac{1}{b}\sum_i [\mathbf{H}]_{i,j}\,, \quad \tilde{\sigma}_j^2 = \frac{1}{b}\sum_i ([\mathbf{H}]_{i,j} - \tilde{\mu}_j)^2\,.$$

Second, we standardize the output so that each column has mean 0 and standard deviation 1:

$$\mathbf{H}' = \frac{\mathbf{H} - \tilde{\mu}}{\sqrt{\tilde{\sigma}^2 + \varepsilon}}\,,$$

where $\varepsilon > 0$ is a small coefficient to avoid division by 0.

The final output of the batch normalization layer $\text{BN}(\mathbf{H})$ sets a new mean and variance for each column:

$$[\text{BN}(\mathbf{H})]_{i,j} = \alpha_j H'_{i,j} + \beta_j \,. \tag{13}$$

The $2f$ values α_j, β_j are trained via gradient descent.
Commonly, BN is inserted between a fully-connected layer and the activation function:

$$\mathbf{Z} = \phi\left(\text{BN}(\mathbf{X}\mathbf{W} + \mathbf{b})\right) \,. \tag{14}$$

Similarly to dropout, BN requires a different behaviour outside of training, since it is undesirable that the output for an input depends on the mini-batch it is associated to.
Two common solutions are:

- After training, compute a mean and variance by running the trained model on the entire dataset, fixing the values μ_j, σ_j^2 to that value.
- Keep a moving average of all the estimated means and variances when training, using the final value during inference.

BN is very common in convolutive layers. Consider the output $H_{(b,h,w,c)}$ of a generic 2D convolutive layer (b, as before, is the size of the mini-batch).
BN works exactly as before, but the mean and the variance are computed *for each channel*:

$$\tilde{\mu}_z = \frac{1}{bhw}\sum_{i,j,k}[\mathbf{H}]_{i,j,k,z} \,, \quad \tilde{\sigma}_z^2 = \frac{1}{bhw}\sum_{i,j,k}\left([\mathbf{H}]_{i,j,k,z} - \tilde{\mu}_z\right)^2 \,. \tag{15}$$

Batch normalization has multiple issues in practice:

- It introduces dependencies across the elements of the mini-batch, making it less suitable in, e.g, distributed optimization or contrastive self-supervised learning.
- The variance of the estimate can be very high when the batch size is small, which is a problem with very large models.
- There is a mismatch between its training and inference behaviours.

LAYER NORMALIZATION

It is very easy to define multiple variants of BN by varying the axes along which statistics are computed and controlled.
For example, a popular variant is layer normalization, where we mean and variances are computed for each row (each input) independently:

$$\tilde{\mu}_i = \frac{1}{f}\sum_{j}[\mathbf{H}]_{i,j} \,, \quad \tilde{\sigma}_i^2 = \frac{1}{f}\sum_{j}\left([\mathbf{H}]_{i,j} - \tilde{\mu}_i\right)^2 \,.$$

This works also with small batch sizes (even 1) and it does not add any inter-batch dependency. Importantly: in LN, α and β have the same shape as axis along which we normalize.

RESIDUAL CONNETIONS

In a residual network, we modify each block $f(x)$ (e.g., a VGG block) by adding a skip connection:

$$r(x) = f(x) + x\,, \tag{16}$$

If x and $f(x)$ have different dimensionality, we can rescale x with a matrix multiplication or a 1×1 convolutive block. $r(x)$ is called a residual block.
Residual blocks work very well with batch normalization on $f(x)$ (the residual path), because it tends to bias the network towards the skip path at initialization.[9]

Note that:

$$\partial r(x) = \partial f(x) + \mathbf{I}\,.$$

When using a residual connection, during backpropagation the gradient always flows unhindered trough the residual path, reducing any vanishing or exploding effects:

$$v^\top [\partial r(x)] = \underbrace{v^\top [\partial f(x)]}_{\text{Original VJP}} + v\,.$$

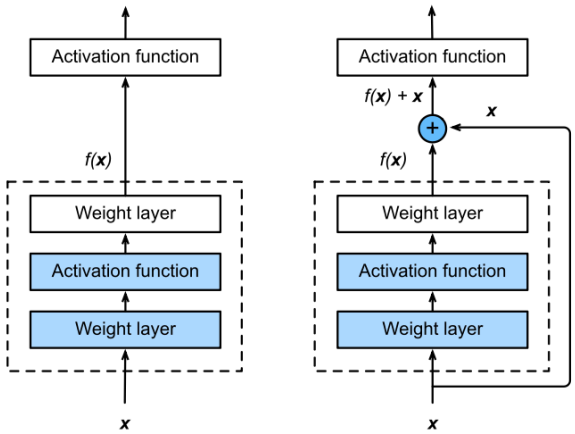


Figure 15: Source: Dive into Deep Learning, Chapter 7.6. We typically put the ReLU outside $f(x)$, otherwise each residual block would be positive and it would be impossible to have a negative change.

NN FOR SEQUENTIAL INFORMATION PROCESSING

RECURRENT NEURAL NETWORK

Introduce state variables to store past information, and then determine the current outputs, together with the current inputs.

QUANTITATIVE ATTRIBUTES

Time Series can be seen a *sequence of random variables*, x_0, x_1, . . ., also denoted as *stochastic process*, of which we only observe the empirical realizations.
The observations are acquired at equally spaced time intervals, denoted as sampling rate (*sample per second*).

Thus, time series can be described by *quantitative* statistic attributes:

$\mu = \mathrm{E}\{\mathbf{x}\}$ *first order statistics*

$\boldsymbol{\Sigma} = \mathrm{E}\left\{(\mathbf{x}-\mu)(\mathbf{x}-\mu)^{\mathsf{T}}\right\}$ *second order statistics*

AUTOREGRESSIVE MODEL

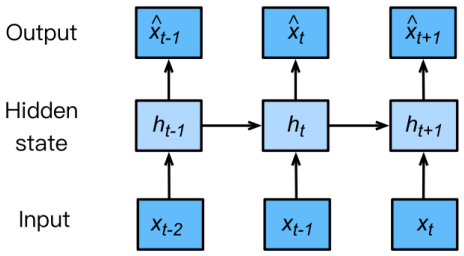


Figure 2: A latent autoregressive model [2].

Autoregressive models estimate the training time series as:

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \ldots, x_1).$$

MARKOV

Whenever the AR approximation of using only $x_{t-1}, \ldots, x_{t-\tau}$ instead of $x_{t-1}, \ldots, x_1$ holds to estimate x_t, we say that the sequence satisfies a **Markov condition**.

If $\tau = 1$, we have a first-order Markov model and $P(x)$ is given by

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1).$$

Such models are particularly nice whenever x_t assumes only a discrete value, since in this case dynamic programming can be used to compute values along the chain exactly.

LATENT VARIABLE MODEL

Hence, rather than modeling $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$ we use a latent variable model:

$$p(x_t \mid x_{t-1}, \ldots, x_1) \approx p(x_t \mid x_{t-1}, h_t)$$

where h_t is a *latent variable* that stores the sequence information.

Consider a multilayer perceptron with a single hidden layer with a minibatch $\mathbf{X} \in \mathbb{R}^{n \times d}$ with sample size n and d inputs, similarly to the network previously used for time series prediction.

Let the hidden layer's activation function be ϕ. The hidden layer's output $\mathbf{H} \in \mathbb{R}^{n \times h}$ is:

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h) \tag{1}$$

where $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ is weight parameter, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ the bias parameter, and h is the number of hidden units for each hidden layer.

The hidden variable $\mathbf{H}$ is used as the input of the output layer.

NN WITHOUT HIDDEN STATES

The output variable $\mathbf{O} \in \mathbb{R}^{n \times q}$ is given by:

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q. \tag{2}$$

where $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ is the bias parameter of the output layer.

For a classification problem, we can use softmax($\mathbf{O}$) to compute the probability distribution of the output category.

We can pick (x_t, x_{t-1}) pairs at random and estimate the parameters $\mathbf{W}$ and $\mathbf{b}$ of our network via stochastic gradient descent.

The hidden state at time t could be computed based on both x_t and h_{t-1}:

$$h_t = f(x_t, h_{t-1}).$$

After all, h_t could simply store all the data it observed so far. But it could potentially makes both *computation* and *storage* expensive.

NN WITH HIDDEN STATES: RECURRENT NEURAL NETWORK

Now consider a neural network with hidden states with a minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and a hidden variable $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ at time step t.

Unlike the MLP, here we save $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ to describe how to use $\mathbf{H}_{t-1}$ in the current time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{3}$$

The neural network described by (3) and containing the additional term $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ denoting the hidden state, is called **recurrent neural network** (RNN).

The output of the RNN is defined like an MLP without hidden states, i.e., by (2).

It is worth noting that the number of RNN model parameters does not grow as the number of timesteps increases.

NUMERICAL INSTABILITY GRADIENT COMPUTATION

For a sequence of length T, we compute the gradients T times for iteration, which might result in numerical instability, e.g., the gradients may either explode or vanish for large T.

Recall that when solving an optimization problem, we take update steps for the weights $\mathbf{w}_t$ in the general direction of the negative gradient $\mathbf{g}_t$ on a minibatch, say $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot \mathbf{g}_t$.

Let's further assume that the objective is well behaved, i.e., it is Lipschitz continuous with constant L, i.e.:

$$|l(\mathbf{w}_t) - l(\mathbf{w}_{t+1})| \leq L \|\mathbf{w}_t - \mathbf{w}_{t+1}\|.$$

Thus, if we update $\mathbf{w}_t$ by $\eta \mathbf{g}_t$, we will not observe a change by more than $L\eta \|\mathbf{g}_t\|$.

This may slow down the convergence speed but it may also limit the gradient extent in wrong directions.

GRADIENT CLIPPING

If the gradient is too large we may reduce the learning rate η.

If we only rarely get large gradients, we may *clip* the gradients by projecting them back to a ball of a given radius, θ:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}.$$

Due to the gradient clipping, the *gradient norm* never exceeds θ and the *updated gradient* is entirely aligned with the original direction $\mathbf{g}$.

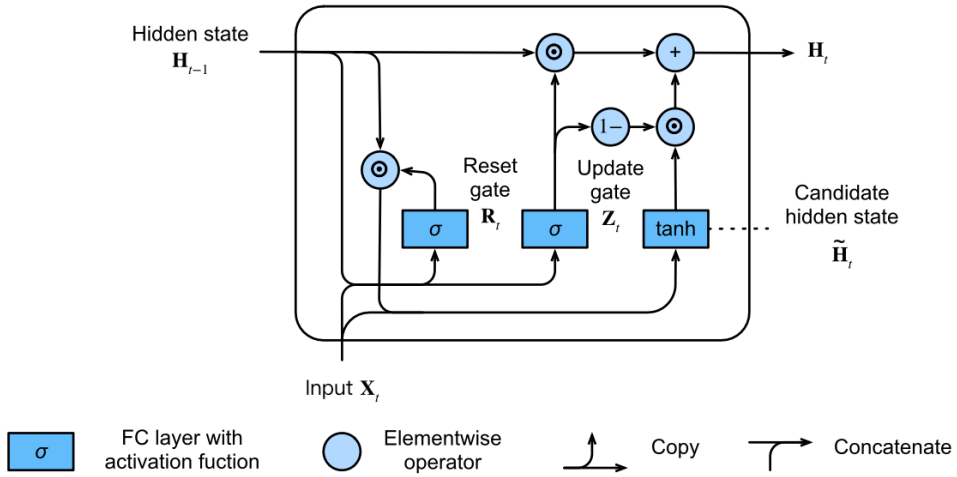
It also has the desirable side-effect of limiting the influence any given minibatch can exert on the weight vectors.

Gradient clipping provides a quick fix to the exploding gradient.

NN FOR LONG SHORT-TERM MEMORY

GATED RECURRENT UNIT

To resolve bigger gap in the long term dependencies we introduce Gated Recurrent Unit.
The **gated recurrent unit** (GRU) is one of the modern RNN models, whose peculiarity lies in the presence of a hidden state gating, including dedicated *update* and *reset* gates.
Reset gates help capture *short-term dependencies* in time series.
Update gates help capture *long-term dependencies* in time series.



Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ are computed as follows:

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z).$$

with $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$.

Next, we integrate the reset gate $\mathbf{R}_t$ with a regular latent state updating mechanism, leading to a candidate hidden state $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h) \tag{1}$$

This leads to the final update equation for the GRU.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t.$$

LONG SHORT-TERM MEMORY NETWORKS

Long Short Term Memory (LSTM) networks are a special kind of RNN, capable of learning long-term dependencies.
The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time.
The LSTM shares many of the properties of the GRU. Its design is slightly more complex than GRU but predates GRU by almost two decades.
LSTM's design is inspired by logic gates of a computer. It introduces a memory cell (or *cell*) with the same shape as the hidden state, engineered to record additional information.
To control the memory cell we need a number of gates to decide when to remember and when to ignore inputs in the hidden state via a dedicated mechanism.

The input gate $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, the forget gate $\mathbf{F}_t \in \mathbb{R}^{n \times h}$, and the output gate $\mathbf{O}_t \in \mathbb{R}^{n \times h}$ are calculated as follows:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),$$

with $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$, and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$.

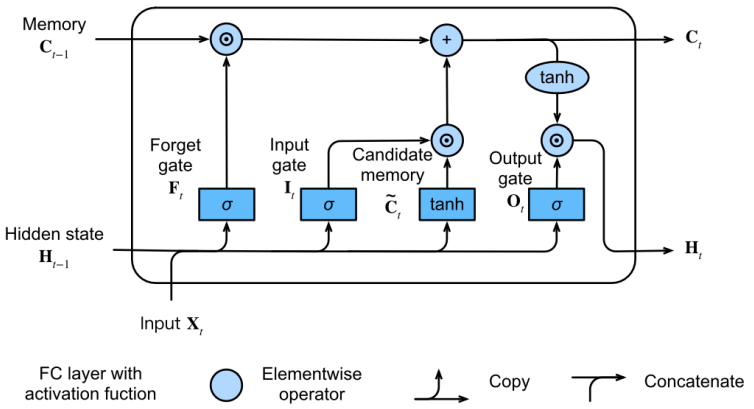


Figure 9: Computing the hidden state in an LSTM model [1].

DEEP RECURRENT NEURAL NETWORKS

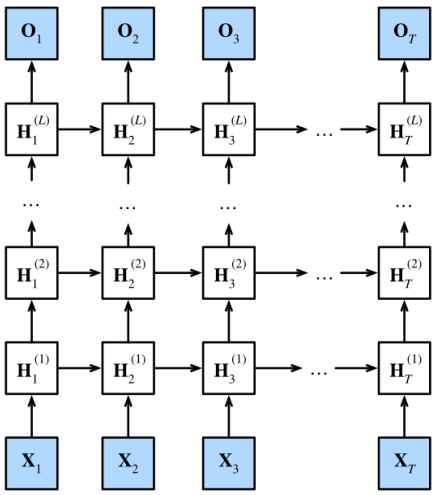


Figure 10: Architecture of a deep recurrent neural network. It describes a deep RNN with L hidden layers. Each hidden state is continuously passed to both the next timestep of the current layer and the current timestep of the next layer[1].

Let us consider a minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$.

The hidden state of hidden layer l ($l = 1, \ldots, L$) is $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$, the output layer variable is $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ and a hidden layer activation function ϕ_l for layer l.

The hidden state of layer l is expressed as:

$$\mathbf{H}_t^{(l)} = \phi_l \left(\mathbf{H}_t^{(l-1)} \mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)} \mathbf{W}_{xh}^{(l)} + \mathbf{b}_h^{(l)} \right).$$

with $\mathbf{W}_{xh}^{(l)}, \mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, and $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$.

The output layer is only based on the hidden state of hidden layer L:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)} \mathbf{W}_{hq} + \mathbf{b}_q$$

with $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$.

TRANSFORMERS

WHY CHOOSE TRANSFORMERS.

Convolutional layers are not always optimal, because an object can depend with one far from its position. Recurrent neural networks were a solution, but can they can be time-consuming to train, as they need to back propagate for each iteration.

CORE OF THE TRANSFORMERS.

Introduce a new layer **MULTI-HEAD ATTENTION**, that replace assumption of locality (i.e. a word can be influence by nearby words) with a general notion of sparsity (i.e. every part of input can interact with every other part.)

REMOVE LOCALITY

$$\mathbf{h}_i = \sum_{j=-k}^{k} \mathcal{W}_j \mathbf{x}_{i+j} \,,$$

Given a 1D convolutional layer with kernel size k, we want to remove its locality. Increasing k increase the number of parameters.
We can fix k, and train a block g(i) that learns dynamically each possibile parameters of the kernel for each I position.

This kinda of model is called **continuous convolutions.**
Unfortunately it assumes that dependencies are regular, so we define another function that depends on the **values** of token.

$$\mathbf{h}_i = \sum_{j=-i+1}^{n-i} g(i+j)\mathbf{x}_j \,.$$

$$\mathbf{h}_i = \sum_{j=1}^{n} \alpha(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j \,.$$

This is a non-local neural network, choosing a correct **weighting function** we can obtain the **MHA**

WEIGHTING FUNCTION

α is the **attention scoring function** and its outputs (scalar) the attention scores.
For each token attention score are normalized, each token will have a “budget” attention to allocate , increasing an attention score necessarily decreases the attention over remaining tokens.

SELF ATTENTION LAYER

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{d}} \mathbf{x}_i^\top \mathbf{x}_j$$

Normalized dot product
-fast and efficient to normalize.

$$\mathbf{h}_i = \sum_{j=1}^{n} \text{softmax}_j \left(\frac{1}{\sqrt{d}} \mathbf{x}_i^\top \mathbf{x}_j \right) \mathbf{x}_j$$

We need to add some trainable parameters: query, key, value.

$$\mathbf{q}_i = \mathbf{W}_q^\top \mathbf{x}_i, \quad \mathbf{k}_i = \mathbf{W}_k^\top \mathbf{x}_i, \quad \mathbf{v}_i = \mathbf{W}_v^\top \mathbf{x}_i \,.$$

$$\mathbf{h}_i = \sum_j \text{softmax} \left(\frac{1}{\sqrt{d}} \mathbf{q}_i^\top \mathbf{k}_j \right) \mathbf{v}_j \,.$$

$$\underset{(n,q)}{\mathbf{Q}} = \mathbf{X}\mathbf{W}_q, \quad \underset{(n,q)}{\mathbf{K}} = \mathbf{X}\mathbf{W}_k, \quad \underset{(n,v)}{\mathbf{V}} = \mathbf{X}\mathbf{W}_v$$

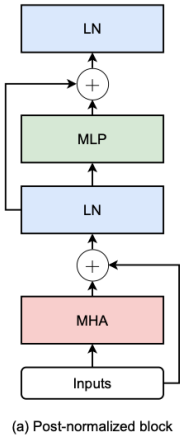
$$\underset{(n,v)}{\mathbf{H}} = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{q}} \right) \mathbf{V} \,.$$

MULTI HEAD ATTENTION:

MHA is needed when we need dependencies on multiple subset of tokens. (i.e. More relations on words)

$$\mathbf{Q}_t = \mathbf{X}\mathbf{W}_{q,t}, \quad \mathbf{K}_t = \mathbf{X}\mathbf{W}_{k,t}, \quad \mathbf{V}_t = \mathbf{X}\mathbf{W}_{v,t}$$

$$\mathbf{H}_t = \text{softmax}\left(\frac{\mathbf{Q}_t\mathbf{K}_t^\top}{\sqrt{q}}\right)\mathbf{V}_t\,.$$



(a) Post-normalized block

TRANSFORMER BLOCK

1) MHA layer: H = MHA (X)
2) Add a residual connection and a layer normalization operation: H = LayerNorm(H + X)
3) Apply a fully-connected model g(.) on each row: F = g (H)
4) Do again step 2: H = LayerNorm(F + H)

POSITIONAL EMBEDDINGS

$\text{MHA}(\mathbf{P}\mathbf{X}) = \mathbf{P}\cdot\text{MHA}(\mathbf{X})$ They are needed to fix the problem of equivariance of the MHA (not good for sequences)

We break equivariance trough concatenating or sum with a positional embeddings.

$$\mathbf{X}' = [\mathbf{X} \parallel \mathbf{E}] \quad \textit{or} \quad \mathbf{X}' = \mathbf{X} + \mathbf{E}$$

TRAINABLE POSITIONAL EMBEDDINGS

For each position is associated an embedding vector of fixed dimension, that is trained with the rest of the network

SINGLE AND MULTIPLE SINUSOIDAL EMBEDDINGS

$$\mathbf{E}_i = [\sin(i\omega)]\,. \qquad \mathbf{E}_i = [\sin(i\omega_0), \sin(i\omega_1), \ldots, \sin(i\omega_e)]\,. \qquad \omega_j = \frac{1}{10000^{j/e}}\,.$$

FINAL VERSION

Alternate sin and cos

$$[\mathbf{E}]_{i,2j} = \sin\left(\frac{i}{10000^{2j/e}}\right),$$
$$[\mathbf{E}]_{i,2j+1} = \cos\left(\frac{i}{10000^{2j/e}}\right).$$

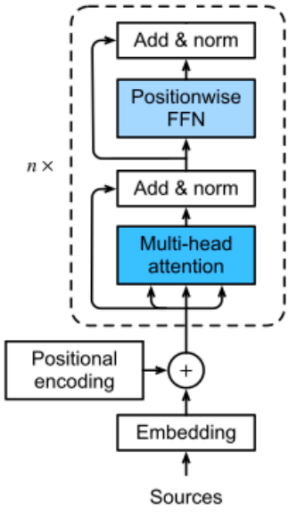
RELATIVE POSITIONAL EMBEDDINGS

$$\alpha(\mathbf{x}_i, \mathbf{x}_j, i-j) = \mathbf{x}_i^\top\mathbf{x}_j + b_{ij}$$

Attention mask is now dependent on the relative distance i-j

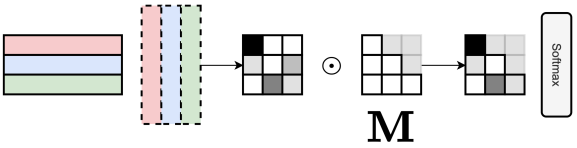
CLASSIFICATION TOKEN

We add a trainable token c to make the classification performed on the las row:



CASUAL TRANSFORMER

$$\mathbf{H} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M}}{\sqrt{q}}\right)\mathbf{V}$$



LINEAR TRANSFORMER

$$\mathbf{h}_i = \frac{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)\mathbf{v}_j}{\sum_j \alpha(\mathbf{q}_i, \mathbf{k}_j)} \qquad \alpha(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y})\,.$$

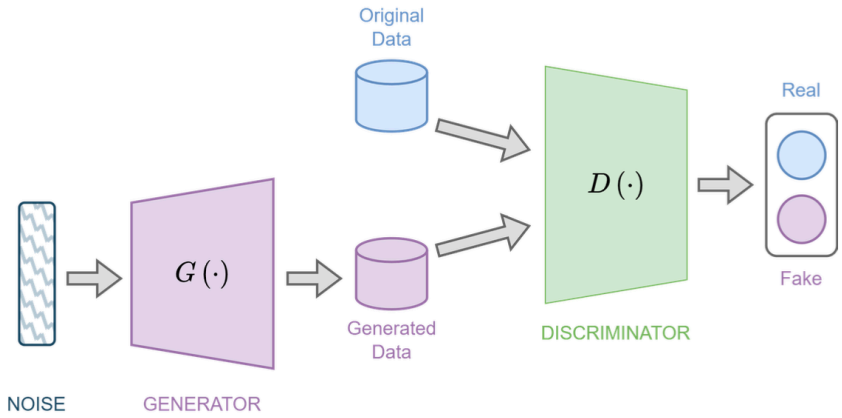
$$\mathbf{h}_i = \frac{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j}{\sum_j \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)} = \frac{\phi(\mathbf{q}_i)^\top \overbrace{\sum_j \phi(\mathbf{k}_j)\mathbf{v}_j^\top}^{\mathbf{S}}}{\phi(\mathbf{q}_i)^\top \underbrace{\sum_j \phi(\mathbf{k}_j)}_{\mathbf{Z}}}\,.$$

GENERATIVE ADVERSARIAL NETWORKS

GENERATOR AND DISCRIMINATOR

Discriminator - aims at maximizing the log-likelihood for binary classification:
1 - original data (REAL)
2 - generated data (FAKE)

Generator - aims at minimizing the log-probability of its samples being classified at FAKE



GENERATOR NETWORK

Generator receives a noise signal z and yields to generated data: $\mathbf{x} = G(\mathbf{z}; \boldsymbol{\theta}_G)$
Proprieties: differentiable, trainable any size of z, do not require invertibility, can make x conditional gaussian given z.

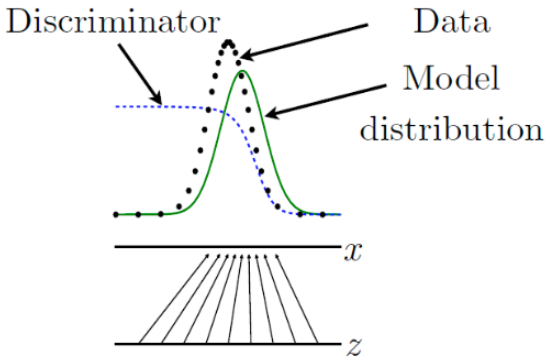
DISCRIMINATOR NETWORK

$$D(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{model}}(\mathbf{x})}$$

Ratio of densities between the real data distribution and the model (generated distribution)

Green: Model distribution p_model(x)
Black Dot: real data p_data(x)
Blue: discriminator estimate the ratio

Discriminator output is high (low) -> model density p_model(x) is low (high)



Learning Process:

Generator adjust G(z) values to move in the direction that increase the discriminator's output D(G(z)) the generator aims to make its output more similar to the real data, as judged by the discriminator.

TRAINING GAN

Use mini batch SGD (Adam) algorithm. Sample simultaneously a mini batch of m training examples { z… z_m} from noise prior Pg(z) and a mini batch of m training examples { x… x_m} from noise prior Pdata(x)

Update the discriminator by ascending its stochastic gradient:

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

Update the generator by ascending its stochastic gradient:

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

$$J_D = -\frac{1}{2}\,\mathrm{E}_{\mathbf{x}\sim p_{\text{data}}} \log D\left(\mathbf{x}\right) - \frac{1}{2}\,\mathrm{E}_{\mathbf{z}\sim p_G(\mathbf{z})} \log\left(1 - D\left(G\left(\mathbf{z}\right)\right)\right)$$

Min-max game $J_G = -J_D$

Non-saturating game $J_G = -\frac{1}{2}\,\mathrm{E}_{\mathbf{z}\sim p_G(\mathbf{z})} \log D\left(G\left(\mathbf{z}\right)\right)$

EVALUATING GAN

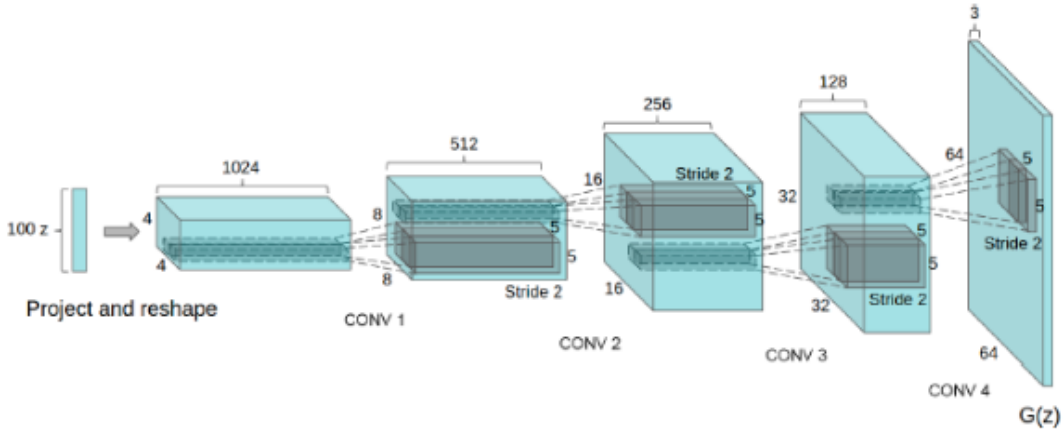
To measure the distance between the real data distribution p_data and the model distribuition p_model the Kullback-Leibler (KL) is used Dkl(p_data || p_model) which returns the objective Ex-pdata {logpϕ(x)}

INCEPTION SCORE

Good generators generate samples that are **semantically diverse.**
IS = exp(H(y) - H(y|x)) (entropy number of classes)

DEEP CONVOLUTIONAL GAN

- It uses **Batch normalization** in most layers of both the discriminator and the generator, with the mini-batch (real and fake) normalized separately - prevent mode collapse
- The last layer of the generator and the first layer of the discriminator are **not batch normalized** - to learn correct mean and scale of data distribution
- Neither “Pooling” nor “Unpooling” Layers. To increase spatial dimension it uses transposed convolution with stride greater than 1
- Relu for generator and Leaky-Relu for discriminator
- Adam optimizer instead of SGD



VIRTUAL BATCH NORMALIZATION

Batch normalization improve optimization of the model, by reparameterization so the mean and variance of each feature are determined by a single mean and a single variance.

Normalization is part of the model so the back propagation computes the gradient of features that are defined to be normalized.

Fluctuation is the main side-effect of the normalization of the GAN they refer to variations in the normalization statistics across different mini-batches during training

Reference Batch Normalization runs the network once on a mini-batch of initial reference examples and once on the current mini-batch of examples

To avoid overfitting, **Virtual Batch Normalization** propose to normalize the current batch of data with statistics derived from the reference batch (Computed using a set of that example and the reference batch)

VARIATIONAL INFERENCE AND DIFFUSION MODELS

LATENT VARIABLE MODELS

We want to learn p(x) to generate new images using latent variables

Given the latent variables $\mathbf{z}$, the joint distribution is factorized as:

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})$$

For training we have access only to $\mathbf{x}$, thus we should *sum out* the unknown, i.e., $\mathbf{z}$, by considering the marginal likelihood function:

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})\, d\mathbf{z}. \tag{1}$$

A natural question now is how to calculate this integral.

- The easiest case is that the integral is tractable.
- If not tractable, we need to compute it by utilizing a specific *approximate inference*, known as variational inference.

AUTOENCODER

In the presence of a larger amount of data or of a more complex problem, involving also a *nonlinear input-output relation*, we need deep learning methods to reduce the dimensionality, with the goal to enhance the representation power:

$$\min_{\mathbf{w}_1, \mathbf{w}_2} \|\mathbf{x}_i - g(f(\mathbf{x}_i; \mathbf{w}_1); \mathbf{w}_2)\|_2^2$$

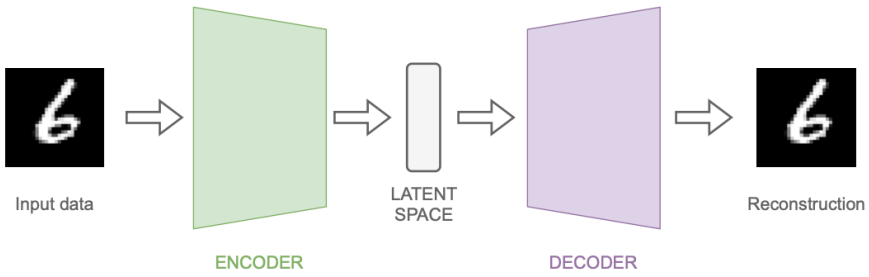


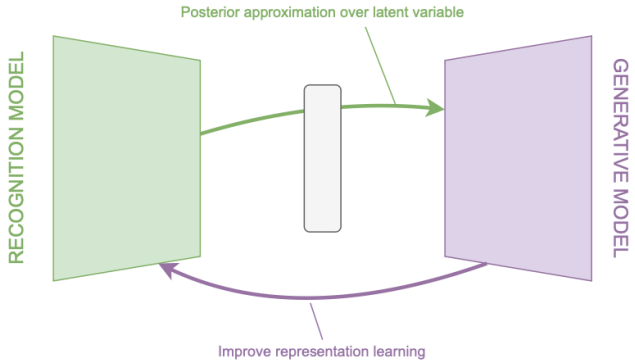
Figure 3: Diagram of an autoencoder, a neural network model composed of: an encoder notework that compresses high-dimensional input data into a lowerdimensional representation vector, and a decoder network that decompresses a given representation vector back to the original domain.

VARIATIONAL AUTO-ENCODER APPROACH

The framework of **variational auto-encoders** (VAEs) [3, 4] provides a principled method for jointly learning deep latent-variable models and corresponding inference models using stochastic gradient descent.

The framework has a wide array of applications, from generative modeling to semi-supervised learning and representation learning.

The VAE can be viewed as *two coupled, but independently parameterized models*: the *encoder*, or recognition model or inference model, and the *decoder*, or generative model.



In the VAE framework, the recognition model is a stochastic function of the input variables.

This is in contrast to the case in which each data observations has a separate variational distribution, which is inefficient for large data-set.

The recognition model uses one set of parameters to model the relation between input and latent variables and it is referred to as amortized inference [5].

The recognition model can be arbitrary complex but is still reasonably fast because it can be done using a single feedforward pass from input to latent variables.

The parameter learning process of the recognition model introduces gradient noise.

The VAE takes this problem by the reparameterization trick, a simple procedure to reorganize the gradient computation, thus reducing variance in the gradients.

With respect to other generative models, and like other likelihood-based models, VAEs generate more dispersed samples, but are better density models in terms of the likelihood criterion.

VARIATIONAL INFERENCE

Often, we are not interested in learning an unconditional model $\mathbf{p_\theta(x)}$, but a conditional model $\mathbf{p_\theta(y|x)}$ such that: $\mathbf{p_\theta(y|x)} \approx \mathbf{p_{data}(y|x)}$.
Conditional models become more difficult to learn when the predicted variables are very high-dimensional, such as images, video or sound.

To avoid notational clutter, unconditional modeling is often assumed but we can generalize those models to conditional ones.

Probability distributions over a very large number of random variables involve direct interactions between relatively few variables [8].

Using a single function to describe the entire joint probability distribution can be very inefficient (both computationally and statistically).

Instead of using a single function to represent a probability distribution, we can split a probability distribution into many factors that we multiply together.

These factorizations can greatly reduce the number of parameters needed to describe the distribution.

When we represent the factorization of a probability distribution with a graph, we call it a structured probabilistic model or ***graphical model.***

PROBABILISTIC GRAPHICAL MODEL

Directed graphical models, or *Bayesian networks*, are a type of probabilistic models where all the variables are topologically organized into a *directed acyclic graph*.

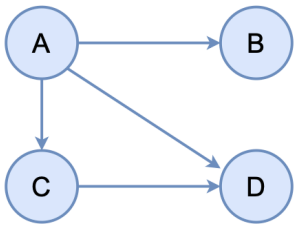


Figure 8: Example of a directed acyclic graph on four vertices.

The joint distribution over the variables of such models factorizes as a product of prior and conditional distributions:

$$p_{\boldsymbol{\theta}}(\mathbf{x}_1, \ldots, \mathbf{x}_M) = \prod_{j=1}^{M} p_{\boldsymbol{\theta}}(\mathbf{x}_j | A(\mathbf{x}_j))$$

where $A(\mathbf{x}_j)$ is the set of parent variables of node j in the directed graph.

Traditionally, $p_{\boldsymbol{\theta}}(\mathbf{x}_j | A(\mathbf{x}_j))$ is parameterized as a lookup table or a linear model [9].

A more flexible parameterization involves neural networks:

$$\begin{aligned} \boldsymbol{\eta} &= f(A(\mathbf{x})) \\ p_{\boldsymbol{\theta}}(\mathbf{x} | A(\mathbf{x})) &= p_{\boldsymbol{\theta}}(\mathbf{x} | \boldsymbol{\eta}) \end{aligned}$$

If all variables in the directed graphical model are observed in the data $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^N$, under the *i.i.d. assumption*, then we can compute and differentiate the log-probability of the data under the model:

$$\log p_{\boldsymbol{\theta}}(\mathcal{D}) = \sum_{\mathbf{x} \in \mathcal{D}} \log p_{\boldsymbol{\theta}}(\mathbf{x})$$

Under the ML criterion, we attempt to find the parameters $\boldsymbol{\theta}$ that maximize the sum, or equivalently the average, of the log-probabilities assigned to the data by the model.

DIRECTED MODELS WITH LATENT VARIABLES

Latent variables $\mathbf{z}$ are variables that are part of the model, but which we *don't observe*, and are therefore not part of the dataset.

The marginal distribution over the observed variables $p_{\boldsymbol{\theta}}(\mathbf{x})$ is given by:

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \int p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})\, d\mathbf{z} \tag{2}$$

where the joint probability $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ is represented by a directed graphical model.

$p_{\boldsymbol{\theta}}(\mathbf{x})$ is also known as marginal likelihood and it is a *flexible* implicit distribution, since it can be represented by mixtures of distributions.

We refer as **deep latent variable model** (DLVM) to a latent variable model $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$ whose distributions are parameterized by neural networks.

Even when each factor in the directed model is relatively simple, the marginal distribution $p_{\boldsymbol{\theta}}(\mathbf{x})$ can be very complex.

This expressivity makes DLVMs attractive for approximating $p_{\text{data}}(\mathbf{x})$.

The simplest, and most common, DLVM is defied by the following factorization:

$$p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) = p_{\boldsymbol{\theta}}(\mathbf{z})\, p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$$

where the prior distribution $p_{\boldsymbol{\theta}}(\mathbf{z})$ and/or $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ are specified.

The main difficulty of ML learning in DLVMs is that the marginal probability $p_{\boldsymbol{\theta}}(\mathbf{x})$ of data under the model is typically intractable.

This is due to absence of any analytical solution of the integral in (2).

Thus, differentiation is not possible, contrary to what is possible with fully observable data.

The intractability of $p_{\boldsymbol{\theta}}(\mathbf{x})$ is related to the intractability of the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, as:

$$p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{x})}.$$

Approximate inference techniques allow us to approximate $p_{\boldsymbol{\theta}}(\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ in DLVMs.

VARIATIONAL AUTOENCODERS

The VAE framework provides a computationally efficient way for optimizing DLVMs jointly with a corresponding inference model using SGD.

To turn the DLVM's intractable posterior into tractable problems, we introduce a parametric inference model $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$.

This model is also called an **encoder** or *recognition model*.

The model parameters $\boldsymbol{\phi}$ are known as variational parameters, which should be optimized such that:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}).$$

Like a DLVM, the inference model can be any directed graphical model:

$$q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = q_{\boldsymbol{\phi}}(\mathbf{z}_1, \ldots, \mathbf{z}_M|\mathbf{x}) = \prod_{j=1}^{M} q_{\boldsymbol{\phi}}(\mathbf{z}_j|A(\mathbf{z}_j), \mathbf{x})$$

where $A(\mathbf{z}_j)$ is the set of parent variables of $\mathbf{z}_j$ in the directed graph.

And also similar to a DLVM, the distribution $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ can be parameterized using deep neural networks, e.g.:

$$\begin{aligned}(\boldsymbol{\mu}, \log \boldsymbol{\sigma}) &= f_{\boldsymbol{\phi}}(\mathbf{x}) \\ q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) &= \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \operatorname{diag}(\boldsymbol{\sigma}))\end{aligned}$$

Tipically, a single encoder neural network $f_{\boldsymbol{\phi}}(\cdot)$ is used to perform posterior inference over all of the datapoints (amortized variational inference strategy [5]).

The optimization objective of the VAE is the **evidence lower bound** (ELBO).

For any choice of the inference model $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, the maximization of the log-likelihood can be expressed as:

$$\begin{aligned}\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{\log p_{\boldsymbol{\theta}}(\mathbf{x})\} \\ &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\} \\ &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\} \\ &= \underbrace{\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\right)\right\}}_{\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})} + \underbrace{\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log\left(\frac{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right\}}_{\mathcal{D}_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))}\end{aligned} \tag{3}$$

where $\mathcal{D}_{\mathrm{KL}}(\cdot)$ the KL divergence and $\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\cdot)$ is the ELBO.

The KL divergence between $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$ and $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ in (3) is nonnegative:

$$\mathcal{D}_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\,||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right) \geq 0.$$

The ELBO in (3) can be expressed as:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right\}$$

Due to (4), the ELBO is a lower bound on the log-likelihood of the data:

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \log p_{\boldsymbol{\theta}}(\mathbf{x}) - \mathcal{D}_{\mathrm{KL}}\left(q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\,||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})\right) \leq \log p_{\boldsymbol{\theta}}(\mathbf{x})$$

STOCHASTIC GRADIENT-BASE OPTIMIZATION OF ELBO

The ELBO allows joint optimization w.r.t. all parameters ($\boldsymbol{\theta}$ and $\boldsymbol{\phi}$) using SGD.

The gradient of individual-datapoint ELBO $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ is in general intractable.

Unbiased gradients of the ELBO w.r.t. parameters $\boldsymbol{\theta}$ are simple to obtain:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \nabla_{\boldsymbol{\theta}}\,\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right\} \\
&= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right)\right\} \\
&\simeq \nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right) \\
&= \nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})\right)
\end{aligned}$$

Unbiased gradients of the ELBO w.r.t. *variational* parameters $\boldsymbol{\phi}$ are more difficult to obtain, since the expectation depends on $\boldsymbol{\phi}$ and cannot be approximated:

$$\begin{aligned}
\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \nabla_{\boldsymbol{\phi}}\,\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right\} \\
&\neq \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left\{\nabla_{\boldsymbol{\theta}}\left(\log p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right)\right\}
\end{aligned} \tag{5}$$

To solve this problem, we can use a change of variables, also called the reparameterization trick [3, 4].

STOCHASTIC GRADIENT OF ELBO UNDER REPARAMETERIZATION

First, we express the random variable $\mathbf{z} \sim q_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ as some differentiable (and invertible) transformation of another random variable $\boldsymbol{\epsilon}$, given $\mathbf{z}$ and $\boldsymbol{\phi}$:

$$\mathbf{z} = g(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x}) \tag{6}$$

where the distribution of the random variable $\boldsymbol{\epsilon}$ is independent of $\mathbf{x}$ or $\boldsymbol{\phi}$.

Thus, the expectation $\mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{\cdot\}$ can be rewritten in terms of $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$:

$$\begin{aligned}\nabla_{\boldsymbol{\phi}} \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{f(\mathbf{z})\} &= \nabla_{\boldsymbol{\phi}} \mathrm{E}_{p(\boldsymbol{\epsilon})}\{f(\mathbf{z})\} \\ &= \mathrm{E}_{p(\boldsymbol{\epsilon})}\{\nabla_{\boldsymbol{\phi}} f(\mathbf{z})\} \\ &\simeq \nabla_{\boldsymbol{\phi}} f(\mathbf{z}).\end{aligned}$$

We can now replace an expectation w.r.t. $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, where $\mathbf{z} = g(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x})$, with one w.r.t. $p(\boldsymbol{\epsilon})$, thus the ELBO becomes:

$$\begin{aligned}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \mathrm{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\{\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\} \\ &= \mathrm{E}_{p(\boldsymbol{\epsilon})}\{\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\}\end{aligned}$$

Thus, we can form a simple Monte Carlo estimator $\widetilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x})$ where we use a single noise sample $\boldsymbol{\epsilon}$ from $p(\boldsymbol{\epsilon})$:

$$\begin{aligned}\boldsymbol{\epsilon} &\sim p(\boldsymbol{\epsilon}) \\ \mathbf{z} &= g(\boldsymbol{\phi}, \mathbf{x}, \boldsymbol{\epsilon}) \\ \widetilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) &= \log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\end{aligned} \tag{7}$$

The reparameterized ELBO estimator is referred to as the stochastic gradient variational Bayes (SGVB) estimator [3].

The computation of the ELBO estimator in (7) requires the computation of $\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$, which is very simple if we use (6).

As long as $g(\cdot)$ in (6) is an invertible function, $\boldsymbol{\epsilon}$ and $\mathbf{z}$ are related by:

$$\log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x}) = \log p(\boldsymbol{\epsilon}) - \log d_{\boldsymbol{\phi}}(\mathbf{x}, \boldsymbol{\epsilon})$$

where:

$$\log d_{\boldsymbol{\phi}}(\mathbf{x}, \boldsymbol{\epsilon}) = \log \left| \det \left(\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} \right) \right|$$

$$\frac{\partial \mathbf{z}}{\partial \boldsymbol{\epsilon}} = \frac{\partial (z_1, \ldots, z_k)}{\partial (\epsilon_1, \ldots, \epsilon_k)} = \begin{pmatrix} \frac{\partial z_1}{\partial \epsilon_1} & \cdots & \frac{\partial z_1}{\partial \epsilon_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial \epsilon_1} & \cdots & \frac{\partial z_k}{\partial \epsilon_k} \end{pmatrix}$$

It is possible to build very flexible transformations $g(\cdot)$ for which $\log d_{\boldsymbol{\phi}}(\mathbf{x}, \boldsymbol{\epsilon})$ is simple to compute, resulting in highly flexible inference models $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$.

VAE pros and cons

+ The latent space can be very small therefore useful for many applications.

+ Fast sampling, even for the generation of multiple samples at the same time.

- VAEs approximate the distribution of the data, so the samples can be imprecise.

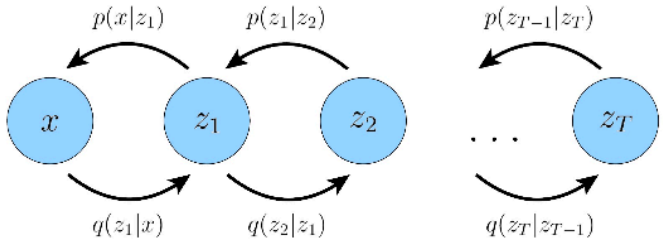
- VAEs are not easy to train due to the two terms in the function to be optimized.

HIERACHICAL VARIATIONAL AUTOENCODER

Hierarchical Variational Autoencoder (HVAE) is a generalization of a VAE that extends to multiple hierarchies over latent variables.

In HVAE, latent variables themselves are interpreted as generated from other higher-level, *more abstract latents*.

In the MHVAE, each transition down the hierarchy is Markovian, where decoding each latent $\mathbf{z}_t$ *only conditions on previous latent* $\mathbf{z}_{t+1}$.



Markovian hierarchical variational autoencoder II

The joint distribution and the posterior of an MHVAE can be written as:

$$p\left(\mathbf{x}, \mathbf{z}_{1:T}\right) = p\left(\mathbf{z}_T\right) p_{\boldsymbol{\theta}}\left(\mathbf{x}|\mathbf{z}_1\right) \prod_{t=2}^{T} p_{\boldsymbol{\theta}}\left(\mathbf{z}_{t-1}|\mathbf{z}_t\right)$$

$$q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T}|\mathbf{x}\right) = q_{\boldsymbol{\phi}}\left(\mathbf{z}_1|\mathbf{x}\right) \prod_{t=2}^{T} q_{\boldsymbol{\phi}}\left(\mathbf{z}_t|\mathbf{z}_{t-1}\right)$$

The ELBO for MHVAE can be also easily derived:

$$\log p\left(\mathbf{x}\right) \geq \mathrm{E}_{q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T}|\mathbf{x}\right)}\left[\log \frac{p\left(\mathbf{x}, \mathbf{z}_{1:T}\right)}{q_{\boldsymbol{\phi}}\left(\mathbf{z}_{1:T}|\mathbf{x}\right)}\right]$$

VARIATIONAL DIFFUSION MODELS

A Variational Diffusion Model can be seen as an MHVAE with three key assumptions:

The latent dimension is exactly equal to the data dimension.

The structure of the latent encoder at each timestep is *not learned* but pre-defined as a *linear Gaussian model* (i.e., a Gaussian distribution centered around the output of the previous timestep).

The Gaussian parameters of the latent encoders vary over time in such a way that the distribution of the latent at final timestep T is a standard Gaussian.

Furthermore, the Markov property between hierarchical transitions is explicitly maintained from a standard MHVAE.

Let us denote both true data samples and latents with the same variable $\mathbf{x}_t$, where $t = 0$ represents true data samples and $t \in [1; T]$ represents a corresponding latent with hierarchy indexed by t.

Considering the first assumption, The VDM posterior is the same as the MHVAE posterior (9), but it can now be rewritten as:

$$q(\mathbf{x}_{1:T}|\mathbf{x}_0) = \prod_{t=1}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}) \tag{11}$$

From the second assumption we know that, unlike an MHVAE, the structure of the encoder at each timestep t is fixed as a linear Gaussian model, where the mean and standard deviation can be set beforehand or learned as parameters [12, 13].

So, we parameterize the Gaussian encoder with

$$\boldsymbol{\mu}_t(\mathbf{x}_t) = \sqrt{\alpha_t}\mathbf{x}_{t-1} \quad \text{and} \quad \boldsymbol{\Sigma}_t(\mathbf{x}_t) = (1 - \alpha_t)\mathbf{I},$$

where the coefficients are chosen such that the variance of the latent variables stay at a similar scale (*variance preserving*).

α_t is a *potentially learnable* coefficient that can vary with the hierarchical depth t, for flexibility.

Encoder transitions can be denoted as:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{\alpha_t}\mathbf{x}_{t-1}, (1 - \alpha_t)\mathbf{I}) \tag{12}$$

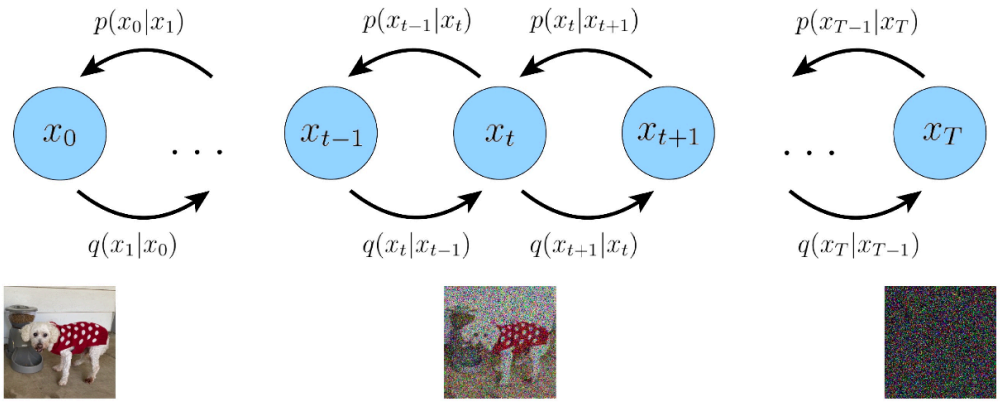


Figure 14: A visual representation of a Variational Diffusion Model. $\mathbf{x}_0$ represents true data observations such as natural images, $\mathbf{x}_T$ represents pure Gaussian noise, and $\mathbf{x}_t$ is an intermediate noisy version of $\mathbf{x}_0$. Each encoder transition $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ is modeled as a Gaussian distribution that uses the output of the previous state as its mean [1].

Like any HVAE, the VDM can be optimized by maximizing the ELBO [1]:

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \int p(\mathbf{x}_{0:T})\, d\mathbf{x}_{1:T} \\
&\geq \mathrm{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\right] \\
&= \underbrace{\mathrm{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}_0|\mathbf{x}_1)\right]}_{\text{reconstruction term}} - \underbrace{\mathrm{E}_{q(\mathbf{x}_{T-1}|\mathbf{x}_0)}\left[\mathcal{D}_{\mathsf{KL}}\left(q(\mathbf{x}_T|\mathbf{x}_{T-1})\,||p(\mathbf{x}_T)\right)\right]}_{\text{prior matching term}} \\
&\quad - \sum_{t=1}^{T-1} \underbrace{\mathrm{E}_{q(\mathbf{x}_{t-1},\mathbf{x}_{t+1}|\mathbf{x}_0)}\left[\mathcal{D}_{\mathsf{KL}}\left(q(\mathbf{x}_t|\mathbf{x}_{t-1})\,||p_{\boldsymbol{\theta}}(\mathbf{x}_t|\mathbf{x}_{t+1})\right)\right]}_{\text{consistency term}}
\end{aligned} \tag{14}$$

The derived form of the ELBO can be interpreted in terms of its individual components.

1. **Reconstruction term**: aims at predicting the *log probability* of the original data sample given the first-step latent. This term also appears in a vanilla VAE, and can be trained similarly.

2. **Prior matching term**: is minimized when the final latent distribution *matches* the Gaussian prior. This term requires no optimization, as it has no trainable parameters; furthermore, as we have assumed a large enough T such that the final distribution is Gaussian, this term effectively becomes zero.

3. **Consistency term**: endeavors to make the distribution at $\mathbf{x}_t$ *consistent*, from both forward and backward processes. This is reflected mathematically by the KL Divergence. This term is minimized when we train $p_{\boldsymbol{\theta}}(\mathbf{x}_t|\mathbf{x}_{t+1})$ to match the Gaussian distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$, which is defined in (12).

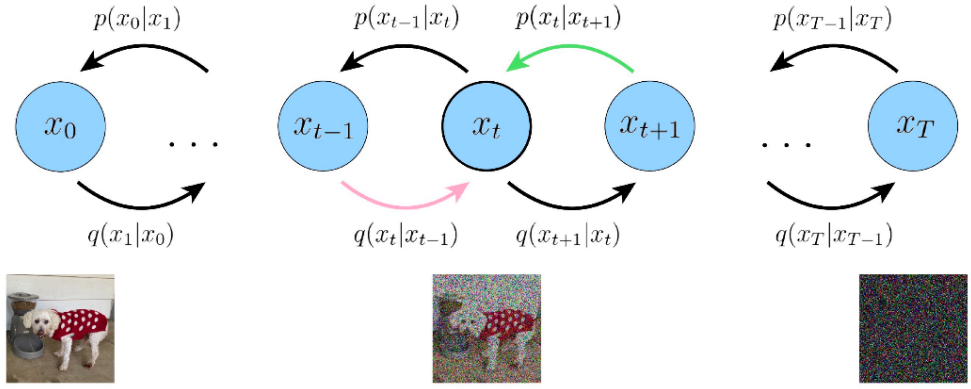


Figure 15: A VDM can be optimized by ensuring that for every intermediate $\mathbf{x}_t$, the posterior from the latent above it $p_{\boldsymbol{\theta}}(\mathbf{x}_t|\mathbf{x}_{t+1})$ matches the Gaussian corruption of the latent before it $q(\mathbf{x}_t|\mathbf{x}_{t-1})$. For each intermediate $\mathbf{x}_t$, we minimize the difference between the distributions represented by the pink and green arrows [1].

DENOISING DIFFUSION PROBABILISTIC MODELS

In order to keep the same dimensions of input and output, a U-Net model [16] can be used to learn predicting the source noise in diffusion models [13].

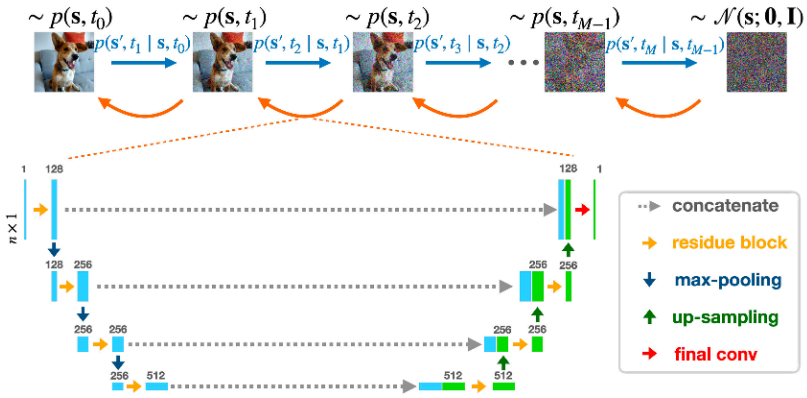


Figure 17: During the diffusion process, which is indicated by the direction of blue arrows in Upper panel, noise is gradually added to the sampled data, in this case the picture of a very good dog, through a diffusion process labeled with diffusion step t_i. This changes the sampled distribution (for example, more pictures of dogs) to a simpler isotropic Gaussian distribution from which one can easily generate more samples. An AI model is then trained to reverse such a diffusion process and starting from sampled noise, to learn to generate images similar to the input image by following the direction indicated by the orange arrows [17].

CODITIONAL DISTRIBUTION

So far, we have focused on modeling just the data distribution $p(\mathbf{x})$.

However, we are often also interested in learning conditional distribution $p(\mathbf{x}|\mathbf{y})$, which would enable us to explicitly control the data we generate through conditioning information $\mathbf{y}$.

A natural way to add conditioning information is simply *alongside the timestep information*, at each iteration.

Recall the VDM joint distribution from (13):

$$p(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t)$$

Then, to turn this into a conditional diffusion model, we can simply add arbitrary conditioning information $\mathbf{y}$ at each transition step as:

$$p(\mathbf{x}_{0:T}|\mathbf{y}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{y})$$

For example, $\mathbf{y}$ could be a text encoding in *image-text generation*, or a low-resolution image to perform *super-resolution*.

We are thus able to learn the core neural networks of a VDM as before, by predicting the source noise or the score function according to the desired diffusion model.