Neural Network

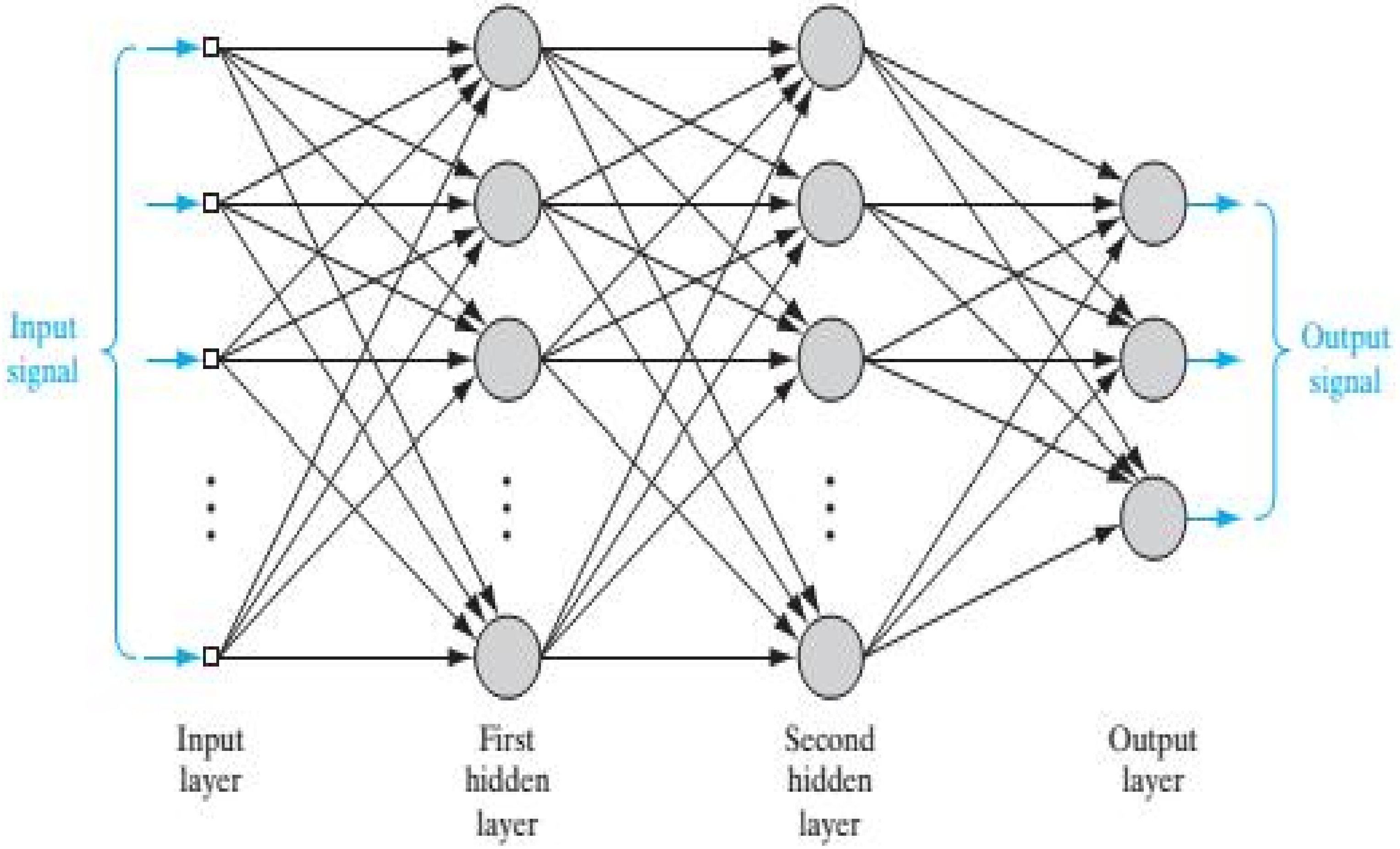


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

Back-Propagation Algorithm

Case 1 Neuron *j* Is an Output Node

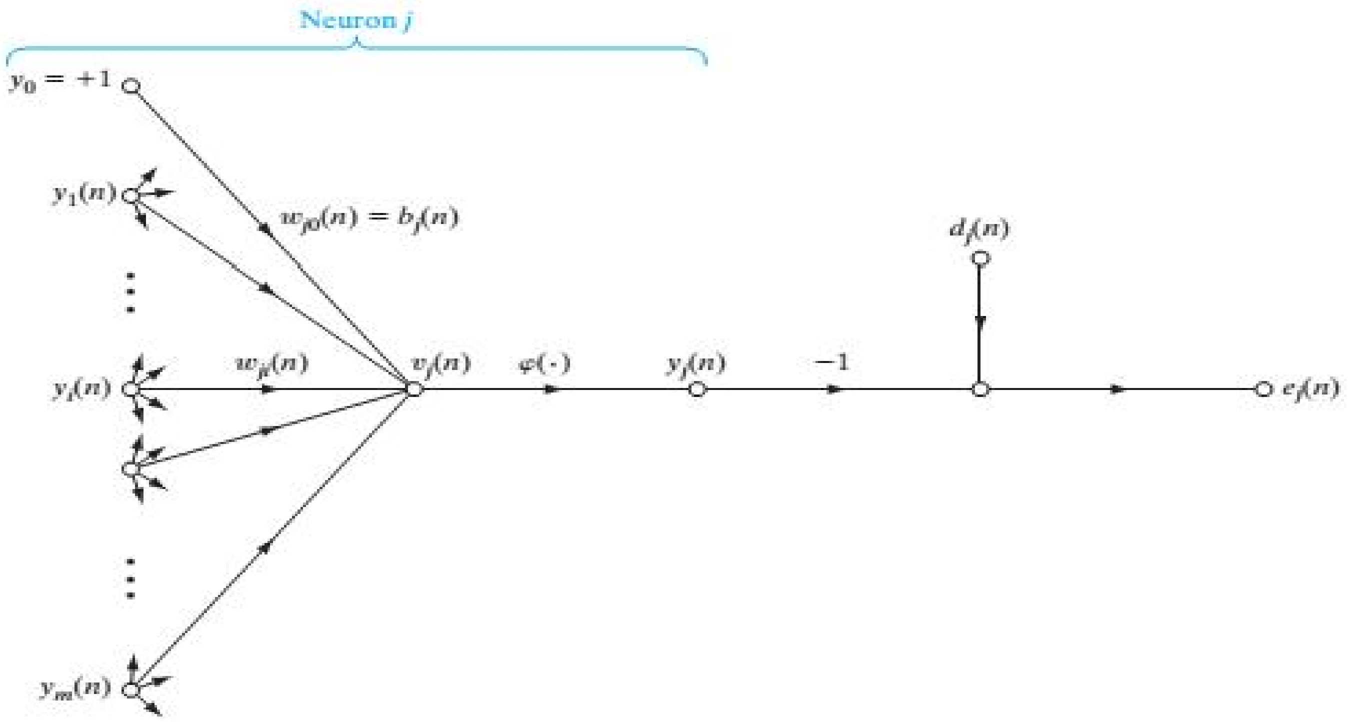


FIGURE 4.3 Signal-flow graph highlighting the details of output neuron *j*.

Induced local field at the input of activation function associated with neuron *j*

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{1}$$

Output of Neuron *j* at iteration *n* $\quad y_j(n) = \varphi_j(v_j(n))$ (2)

Error signal produced at the output of Neuron *j*

$$e_j(n) = d_j(n) - y_j(n) \tag{3}$$

Instantaneous error energy of Neuron *j*

$$\mathscr{E}_j(n) = \frac{1}{2} e_j^2(n) \tag{4}$$

Total Instantaneous error energy of Neuron *j*

$$\begin{aligned} \mathscr{E}(n) &= \sum_{j \in C} \mathscr{E}_j(n) \\ &= \frac{1}{2} \sum_{j \in C} e_j^2(n) \end{aligned} \tag{5}$$

where the set C includes all the neurons in the output layer.

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*, or

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} \tag{6}$$

where η is the *learning-rate parameter* of the back-propagation algorithm.

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{7}$$

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \tag{8}$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{9}$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{10}$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{11}$$

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \qquad (12)$$

Local gradient is defined as

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= e_j(n)\varphi_j'(v_j(n)) \qquad (13)$$

Using Eq. (6)

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \qquad (14)$$

Case 2 Neuron *j* Is a Hidden Node

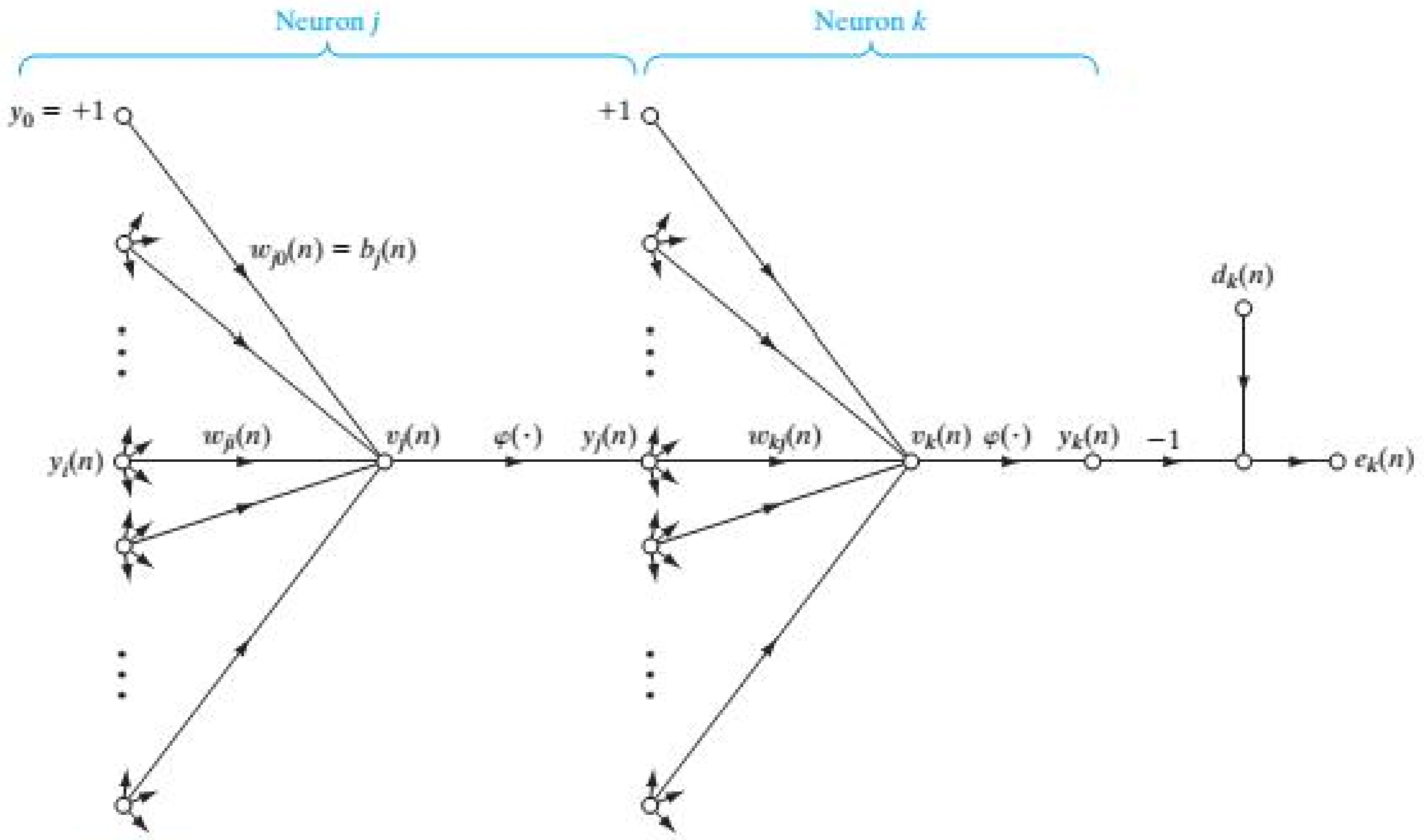


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron *k* connected to hidden neuron *j*.

we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\delta_j(n) = -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}$$

$$= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n)), \qquad \text{neuron } j \text{ is hidden} \tag{15}$$

Total instantaneous error energy

$$\mathscr{E}(n) = \frac{1}{2}\sum_{k \in C} e_k^2(n), \qquad \text{neuron } k \text{ is an output node} \tag{16}$$

where the set C includes all the neurons in the output layer.

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{17}$$

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{18}$$

We note that

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)), \qquad \text{neuron } k \text{ is an output node} \end{aligned} \tag{19}$$

Hence,

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{20}$$

We also note from Fig. 4.4 that for neuron *k*, the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{21}$$

where *m* is the total number of inputs (excluding the bias) applied to neuron *k*.

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{22}$$

$$\begin{aligned} \frac{\partial \mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n) \varphi_k'(v_k(n)) w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \end{aligned} \tag{23}$$

we get the *back-propagation formula* for the

local gradient $\delta_j(n)$, described by

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \tag{24}$$

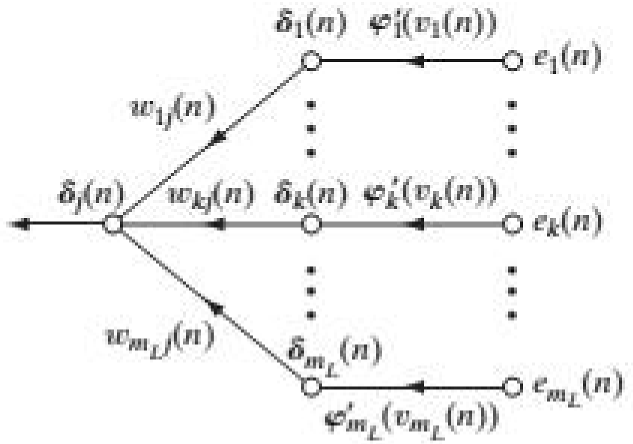

$\delta_1(n)$ $\varphi_1'(v_1(n))$
$e_1(n)$
$w_{1j}(n)$
$\delta_j(n)$
$w_{kj}(n)$ $\delta_k(n)$ $\varphi_k'(v_k(n))$
$e_k(n)$
$w_{m_L j}(n)$
$\delta_{m_L}(n)$
$e_{m_L}(n)$
$\varphi_{m_L}'(v_{m_L}(n))$

FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

Summary

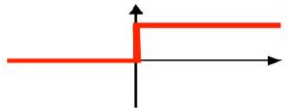
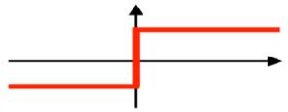
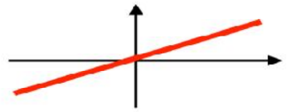
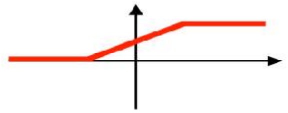
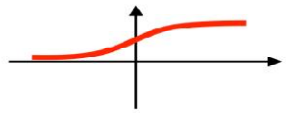
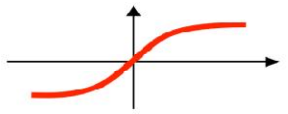
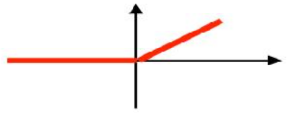
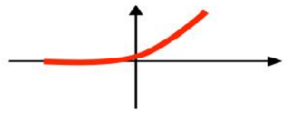
First the correction applied to the synaptic weight connecting neuron i to neuron j

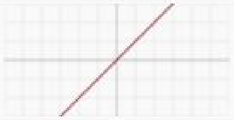
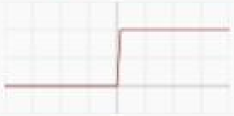
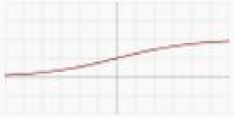
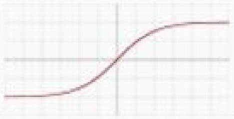
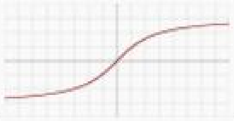
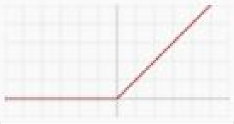
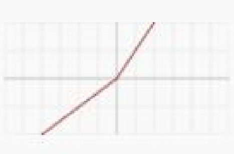
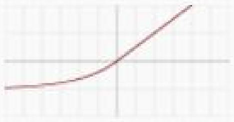
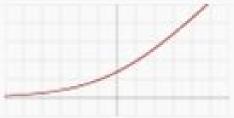
$$\begin{pmatrix} \textit{Weight} \\ \textit{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \textit{learning-} \\ \textit{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \textit{local} \\ \textit{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \textit{input signal} \\ \textit{of neuron } j, \\ y_i(n) \end{pmatrix}$$

Second, the local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron j.

Activation Functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus Copyright © Sebastian Raschka 2016 (http://sebastianraschka.com)	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Bottleneck features Extraction

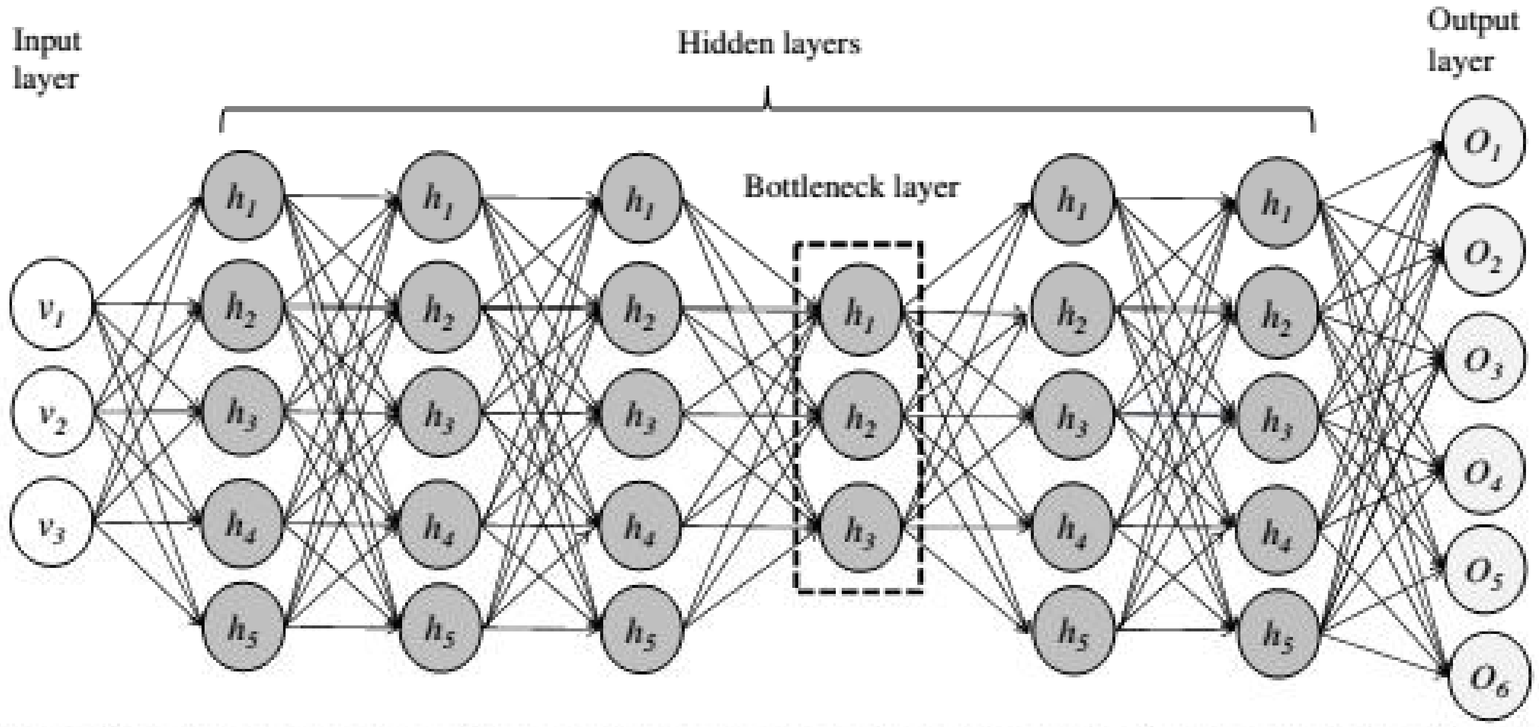


Figure 6.3: Broad structure of deep neural network employed for extracting the bottleneck features.

Time Delay Neural Network

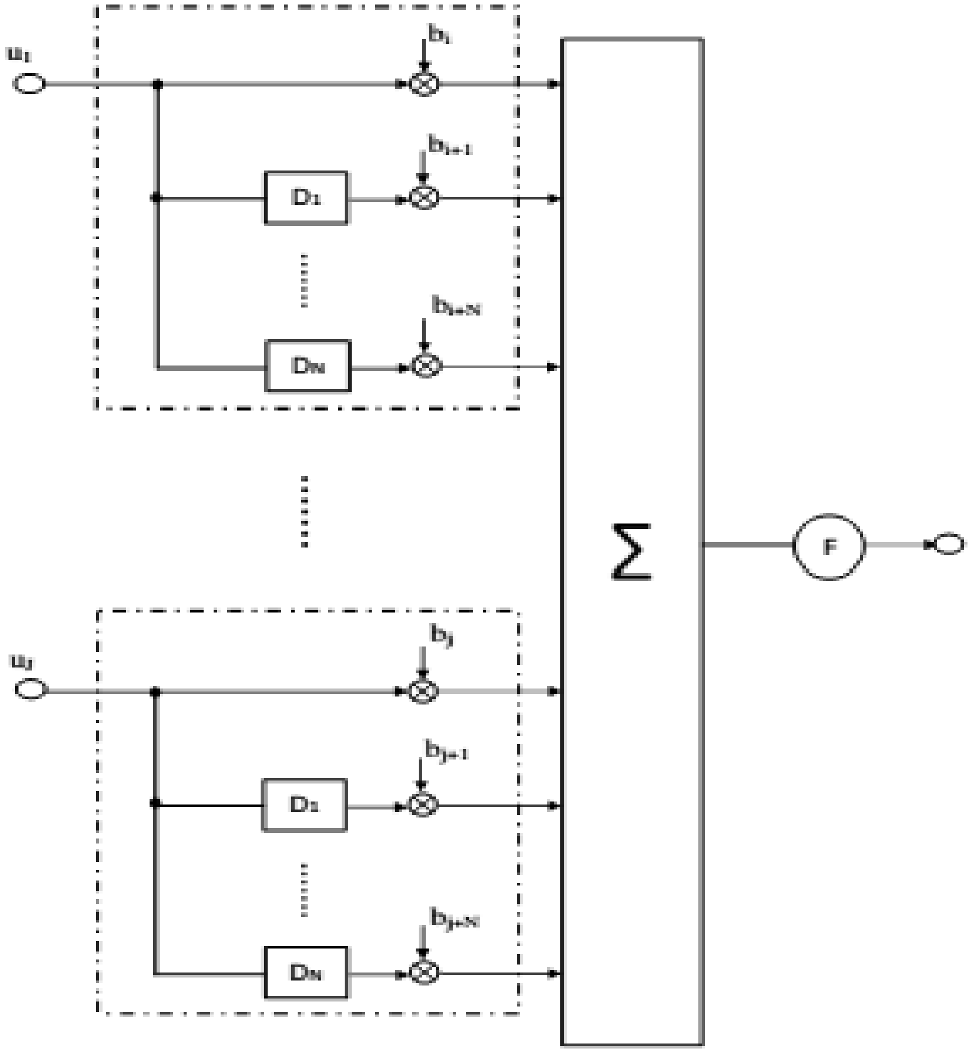


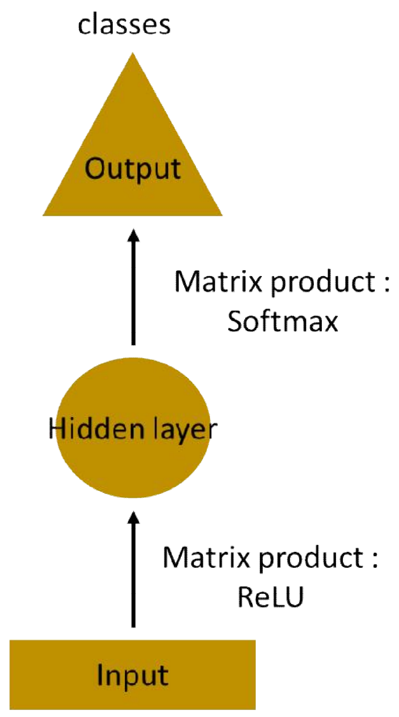
Figure 6.1: Structure of the time delay neural network basic unit.

The TDNN [192] is a multilayer feedforward neural network, and must have two desirable properties. First, the network should have the ability to represent relationships between acoustic events in time. Second, the features learned by the network should be shift invariant. In many neural networks, the basic unit computes the weighted sum of its inputs and then passes this sum through an activation function (e.g., sigmoid function, hyperbolic tangent, etc.). In TDNN, this basic unit is altered by introducing delays D_1 through D_N as shown in Figure 6.1. Here, each of the inputs u_j, $j = 1, 2, \ldots, J$ are delayed from D_1 through D_N, and the weighted sum of the undelayed and delayed inputs are computed and passed through an activation function.

One of the most popular learning technique used to train the neural networks is the backpropagation algorithm [193]. In this, the mean-squared error as a function of the weights is optimized using a gradient descent method. The learning procedure involves two passes through the network, i.e., forward pass and backward pass. In the forward pass, an input training is applied to the network with keeping the weights fixed. Starting from the input layer and working forward to the output layer, the outputs of all the units are calculated at each level. Then, the error between the desired and estimated output is calculated. During the backward pass, the derivative of this error is backpropagated through the network, and weights are adjusted so as to reduce the error. This procedure is iteratively done for all the training patterns until the network converges to the desired output.

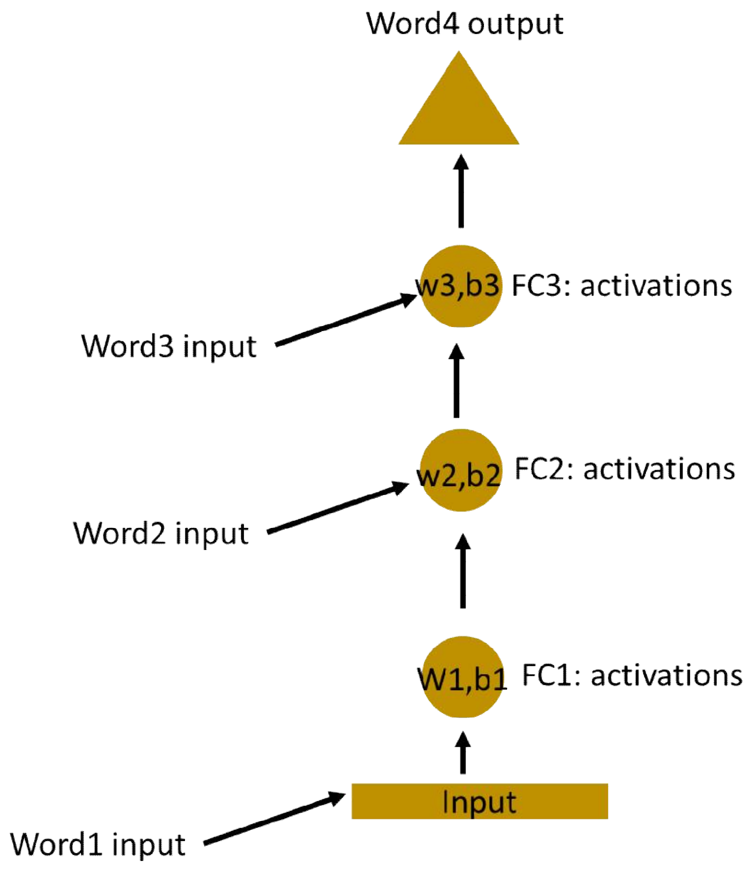
Recurrent Neural Network

Let’s say the task is to predict the next word in a sentence. Let’s try accomplishing it using an MLP. So what happens in an MLP. In the simplest form, we have an input layer, a hidden layer and an output layer. The input layer receives the input, the hidden layer activations are applied and then we finally receive the output.



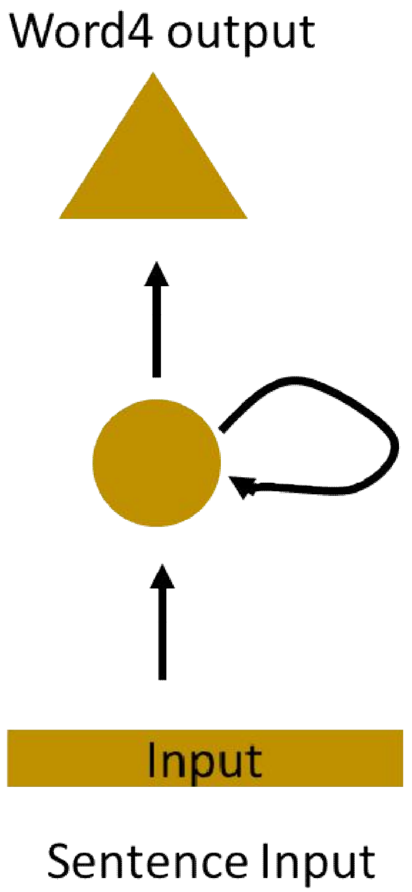
Let’s have a deeper network, where multiple hidden layers are present. So here, the input layer receives the input, the first hidden layer activations are applied and then these activations are sent to the next hidden layer, and successive activations through the layers to produce the output. Each hidden layer is characterized by its own weights and biases.

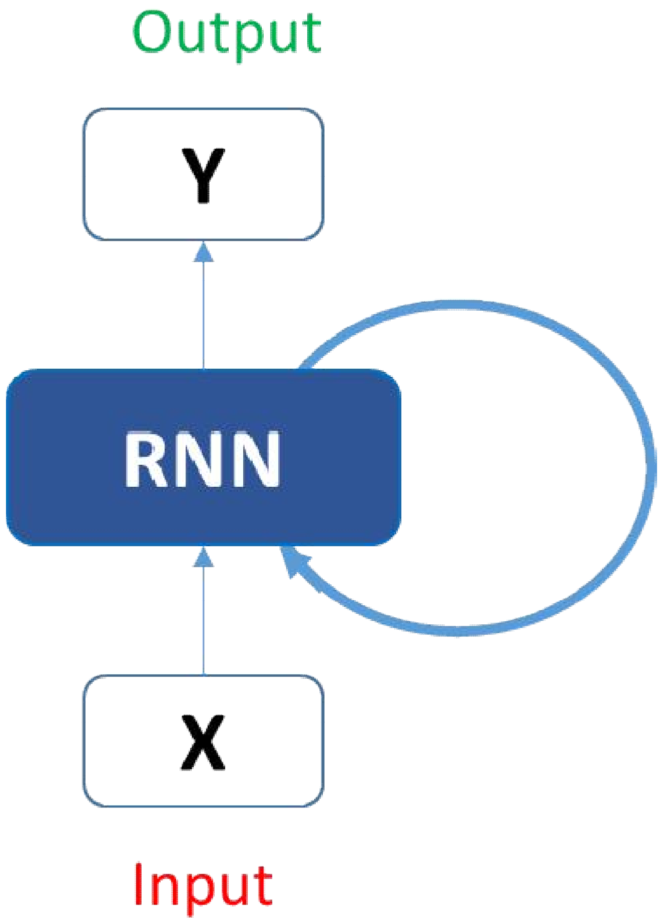
Since each hidden layer has its own weights and activations, they behave independently. Now the objective is to identify the relationship between successive inputs. Can we supply the inputs to hidden layers? Yes we can!

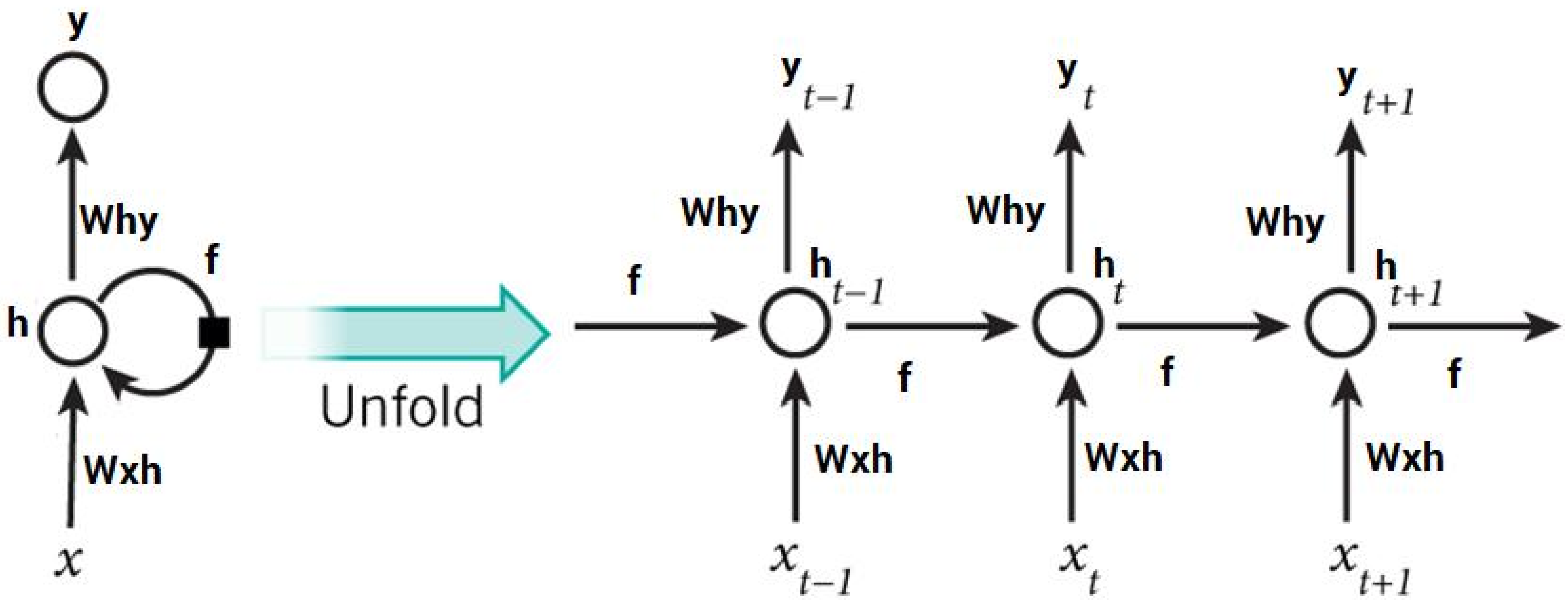


Here, the weights and bias of these hidden layers are different. And hence each of these layers behave independently and cannot be combined together. To combine these hidden layers together, we shall have the same weights and bias for these hidden layers.

We can now combines these layers together, that the weights and bias of all the hidden layers is the same. All these hidden layers can be rolled in together in a single recurrent layer.




Output
Y
RNN
X
Input



y
Why
h
f
Wxh
x
Unfold
f
y_{t-1}
Why
h_{t-1}
f
Wxh
x_{t-1}
y_t
Why
h_t
f
Wxh
x_t
y_{t+1}
Why
h_{t+1}
f
Wxh
x_{t+1}

Let’s see how the above structure be used to predict the fifth letter in the word “hello”. In the above structure, the blue RNN block, applies something called as a recurrence formula to the input vector and also its previous state. In this case, the letter “h” has nothing preceding it, let’s take the letter “e”. So at the time the letter “e” is supplied to the network, a recurrence formula is applied to the letter “e” and the previous state which is the letter “h”. These are known as various time steps of the input. So if at time t, the input is “e”, at time t-1, the input was “h”. The recurrence formula is applied to e and h both. and we get a new state.

The formula for the current state can be written as

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

In this case we have four inputs to be given to the network, during a recurrence formula, the same function and the same weights are applied to the network at each time step.

Taking the simplest form of a recurrent neural network, let’s say that the activation function is tanh, the weight at the recurrent neuron is W_{hh} and the weight at the input neuron is W_{xh}, we can write the equation for the state at time t as –

$$h_t = \tanh (W_{hh}h_{t-1} + W_{xh}x_t)$$

The Recurrent neuron in this case is just taking the immediate previous state into consideration. For longer sequences the equation can involve multiple such states. Once the final state is calculated we can go on to produce the output

Now, once the current state is calculated we can calculate the output state as-

$$y_t = W_{hy}h_t$$

Let me summarize the steps in a recurrent neuron for you-

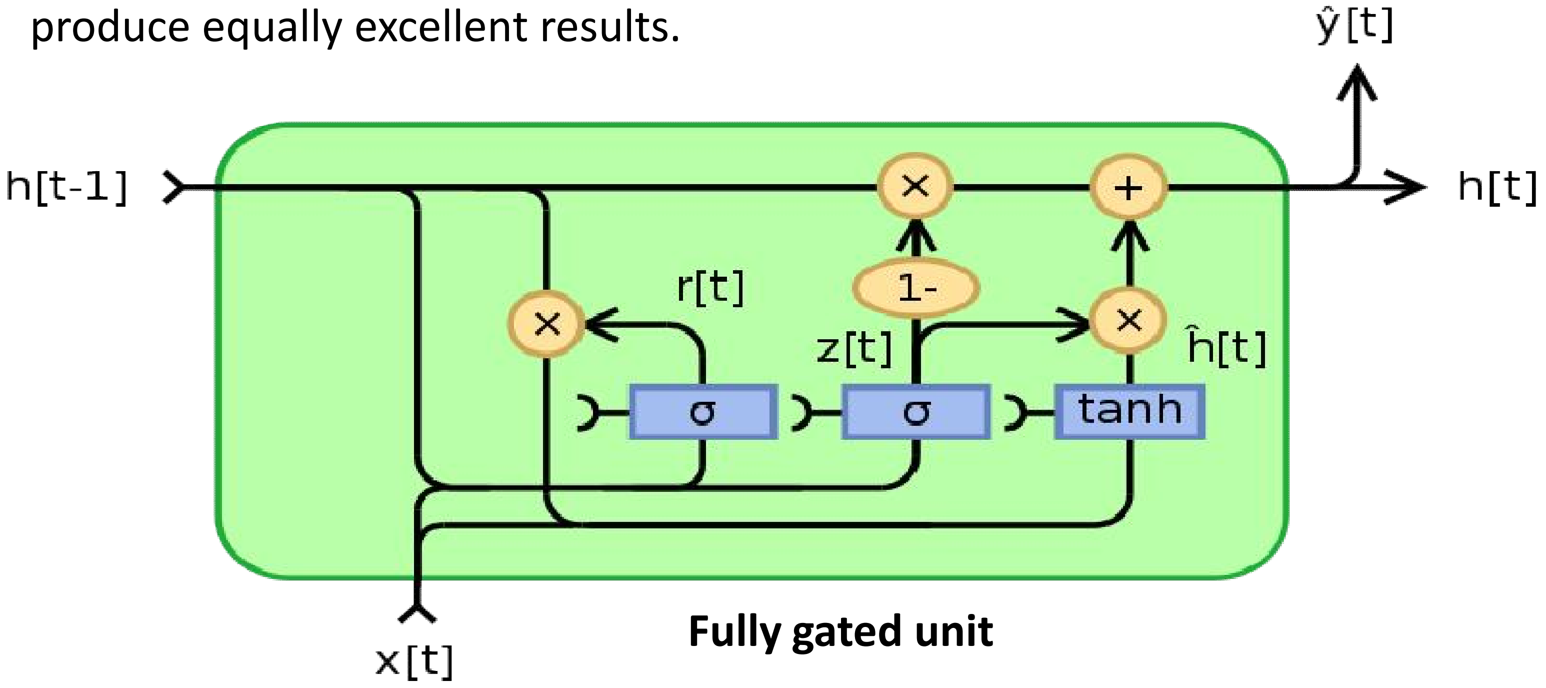
1. A single time step of the input is supplied to the network i.e. x_t is supplied to the network.
2. We then calculate its current state using a combination of the current input and the previous state i.e. we calculate ht
3. The current h_t becomes h_{t-1} for the next time step.
4. We can go as many time steps as the problem demands and combine the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output $y_{t.}$
6. The output is then compared to the actual output and the error is generated
7. The error is then backpropagated to the network to update the weights(we shall go into the details of backpropagation in further sections) and the network is trained.

Gated Recurrent Unit (GRU)

Gated recurrent units (GRUs) are a gating mechanism in recurrent neural networks, introduced in 2014 by Kyunghyun Cho et al.

The GRU is like a long short-term memory (LSTM) with forget gate but has fewer parameters than LSTM, as it lacks an output gate.

GRU (Gated Recurrent Unit) aims to solve the **vanishing gradient problem** which comes with a standard recurrent neural network. GRU can also be considered as a variation on the LSTM because both are designed similarly and, in some cases, produce equally excellent results.



Fully gated unit

Initially, for $t = 0$, the output vector is $h_0 = 0$.

$$
\begin{aligned}
z_t &= \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \\
h_t &= (1 - z_t) \circ h_{t-1} + z_t \circ \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)
\end{aligned}
$$

Variables

- x_t: input vector
- h_t: output vector
- z_t: update gate vector
- r_t: reset gate vector
- W, U and b: parameter matrices and vector

Activation functions

- σ_g: The original is a sigmoid function.
- σ_h: The original is a hyperbolic tangent.

1. Update gate z_t

The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future. That is really powerful because the model can decide to copy all the information from the past and eliminate the risk of vanishing gradient problem.

2. Reset gate r_t

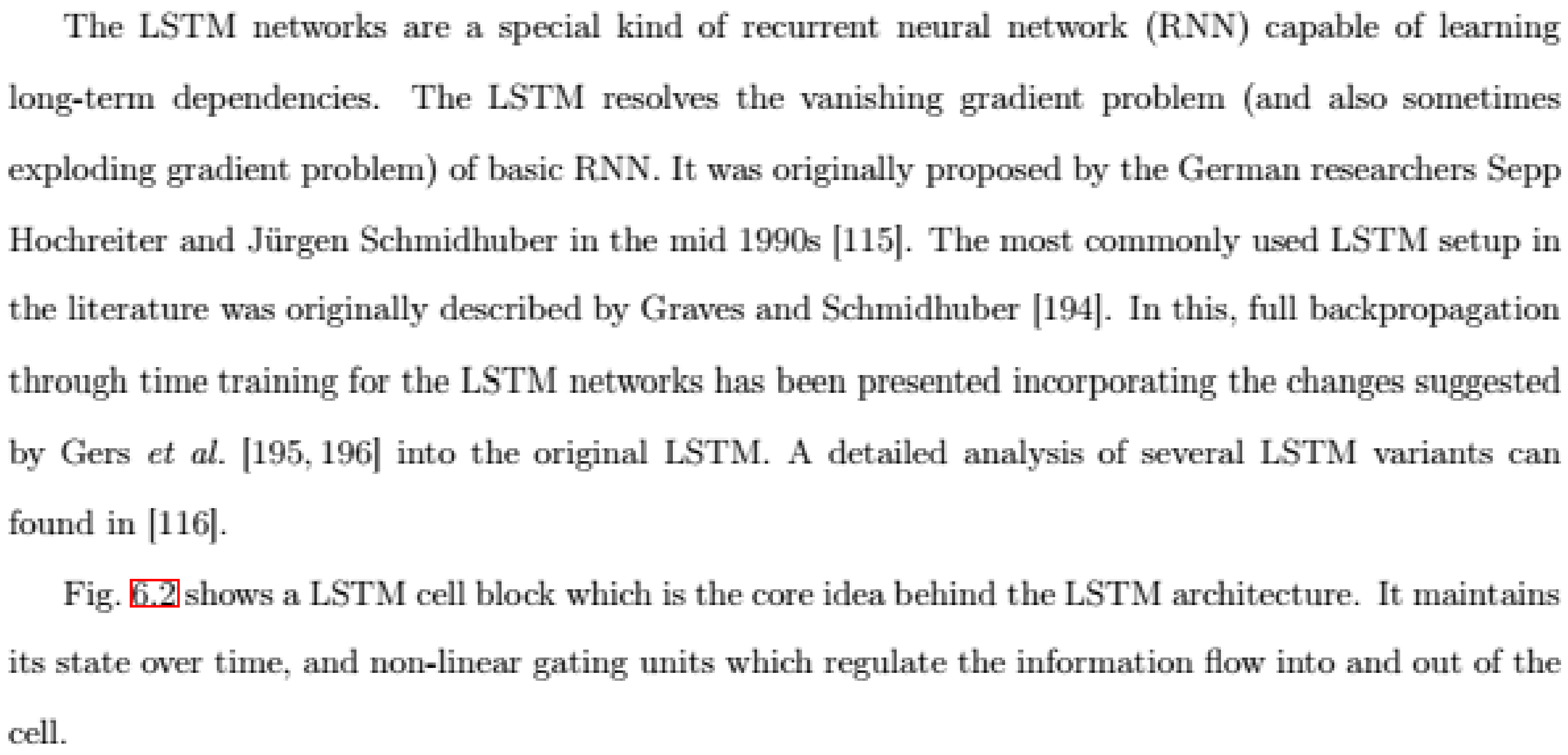
Essentially, this gate is used from the model to decide how much of the past information to forget.

3. Current memory content $\hat{h}_t$

$$\hat{h}_t = \sigma_h(W_h x_t + U_h(r_t \circ h_{t-1}) + b_h)$$

4. Final memory at current time step h_t

Long-Short Term Memory (LSTM)



The LSTM networks are a special kind of recurrent neural network (RNN) capable of learning long-term dependencies. The LSTM resolves the vanishing gradient problem (and also sometimes exploding gradient problem) of basic RNN. It was originally proposed by the German researchers Sepp Hochreiter and Jürgen Schmidhuber in the mid 1990s [115]. The most commonly used LSTM setup in the literature was originally described by Graves and Schmidhuber [194]. In this, full backpropagation through time training for the LSTM networks has been presented incorporating the changes suggested by Gers *et al.* [195, 196] into the original LSTM. A detailed analysis of several LSTM variants can found in [116].

Fig. 6.2 shows a LSTM cell block which is the core idea behind the LSTM architecture. It maintains its state over time, and non-linear gating units which regulate the information flow into and out of the cell.

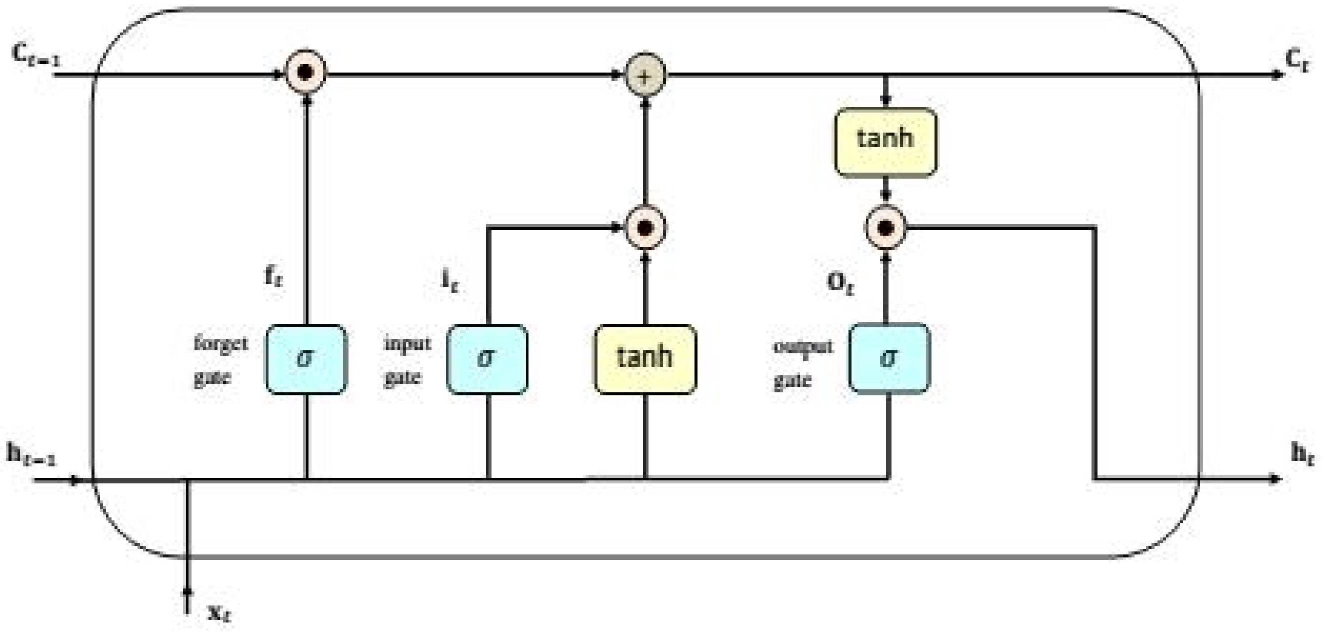


Figure 6.2: Schematic of LSTM cell block

The LSTM cell block has three gates (input, output and forget) and activation functions. The output of previous block is recurrently given as input to the current block as well as the gates. Let $\mathbf{x}_t$ be the input vector at time t, $\mathbf{h}_{t-1}$ be the block output vector at time $t-1$, N is the number of LSTM blocks, and M is the number of inputs. Then, we get the following weights for an layer:

- Input weights: $\mathbf{W}^z$, $\mathbf{W}^i$, $\mathbf{W}^f$, $\mathbf{W}^o$ $\in \mathbb{R}^{N\times M}$
- Recurrent weights: $\mathbf{R}^z$, $\mathbf{R}^i$, $\mathbf{R}^f$, $\mathbf{R}^o$ $\in \mathbb{R}^{N\times N}$
- Bias weights: $\mathbf{b}^z$, $\mathbf{b}^i$, $\mathbf{b}^f$, $\mathbf{b}^o$ $\in \mathbb{R}^{N}$

During forward pass, the vector formulas for a LSTM layer are given as:

a) Block input denoted by $\mathbf{z}_t$ at time t is given as

$$\mathbf{z}_t = \tanh(\mathbf{W}^z\mathbf{x}_t + \mathbf{R}^z\mathbf{h}_{t-1} + b^z) \tag{6.1}$$

b) Input gate denoted by $\mathbf{i}_t$ at time t is given as

$$\mathbf{i}_t = \sigma(\mathbf{W}^i\mathbf{x}_t + \mathbf{R}^i\mathbf{h}_{t-1} + \mathbf{b}^i) \tag{6.2}$$

It decides whether input state enters internal state or not.

c) Forget gate denoted by $\mathbf{f}_t$ at time t is given as

$$\mathbf{f}_t = \sigma(\mathbf{W}^f\mathbf{x}_t + \mathbf{R}^f\mathbf{h}_{t-1} + \mathbf{b}^f) \tag{6.3}$$

It decides whether internal state forgets the previous internal state or not.

d) Output gate denoted by $\mathbf{f}_t$ at time t.

$$\mathbf{o}_t = \sigma(\mathbf{W}^o\mathbf{x}_t + \mathbf{R}^o\mathbf{h}_{t-1} + \mathbf{b}^o) \tag{6.4}$$

It decides whether internal state passes its value to output or not.

e) Memory/cell denoted by $\mathbf{c}_t$ at time t is given as

$$\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t + \mathbf{i}_t \odot \mathbf{z}_t \tag{6.5}$$

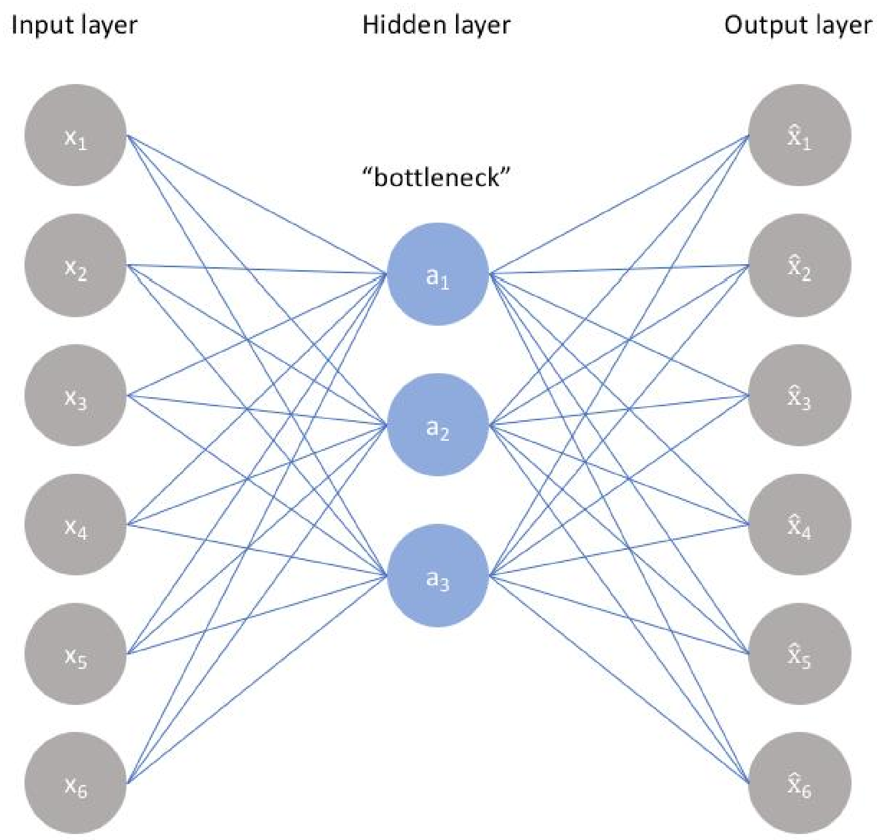
f) Block output denoted by $\mathbf{h}_t$ at time t.

$$\mathbf{h}_t = \tanh(c_t) \odot \mathbf{o}_t \tag{6.6}$$

where $\odot$ denotes element-wise product. The *logistic sigmoid* and *hyperbolic tangent* activation functions are denoted by σ and tanh, respectively. To learn the precise timing of the output, one can use the peephole connections [196]. During backward pass, full backpropagation through time training for LSTM networks has been used.

Autoencoder (AE) https://en.wikipedia.org/wiki/Autoencoder

- The difference between an MLP and an AE is that the aim of the AE is to reconstruct the input, while the purpose of the MLP is to predict the target values with certain inputs.



Autoencoder (AE)

- ❑ AEs have been employed to learn generative models of data.

- ❑ An unsupervised learning algorithm used to efficiently code the dataset for the purpose of dimensionality reduction.

- ❑ It is trained to encode the input into some representation so that the input can be reconstructed from that representation. Essentially, the AE tries to approximate the identity function in this process.

- ❑ In the coding process, the AE first converts the input vector **x** into a hidden representation h using a weight matrix **ω**; then in the decoding process, the AE maps h back to the original format to obtain **x**˜ with another weight matrix ω′. Theoretically, **ω′** should be the transpose of **ω**.

- ❑ Mean square errors (MSEs) are used to measure the reconstruction accuracy according to assumed distribution of the input features.

Autoencoder (AE)

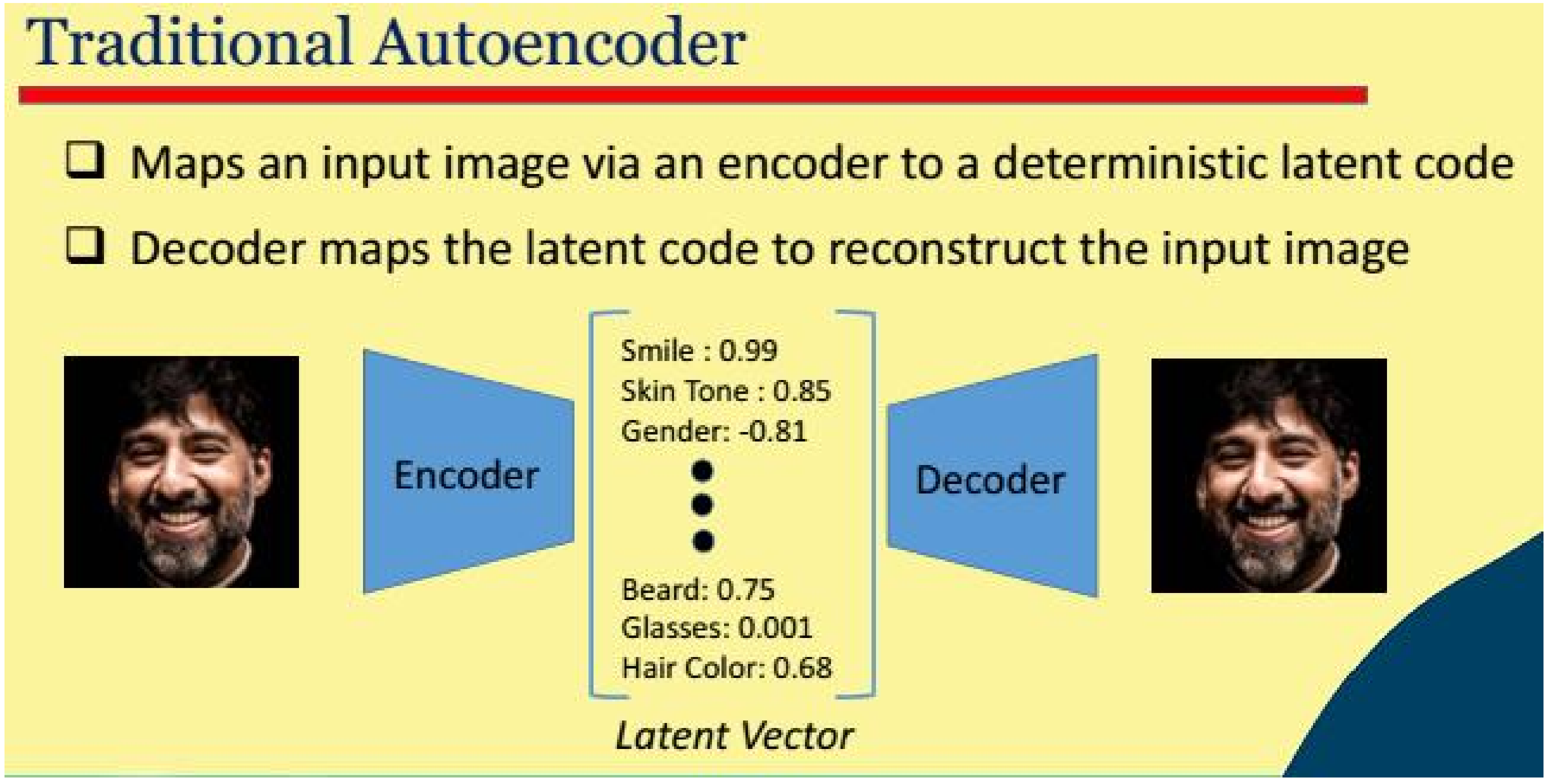
Training process for an AE can also be divided into two stages:

1. To learn features using unsupervised learning and
2. To fine-tune the network using supervised learning.

To be specific, in the first stage, feed-forward propagation is first performed for each input to obtain the output value $x^{\sim}$. Then squared errors are used to measure the deviation of $x^{\sim}$ from the input value. Finally, the error will be backpropagated through the network to update the weights.

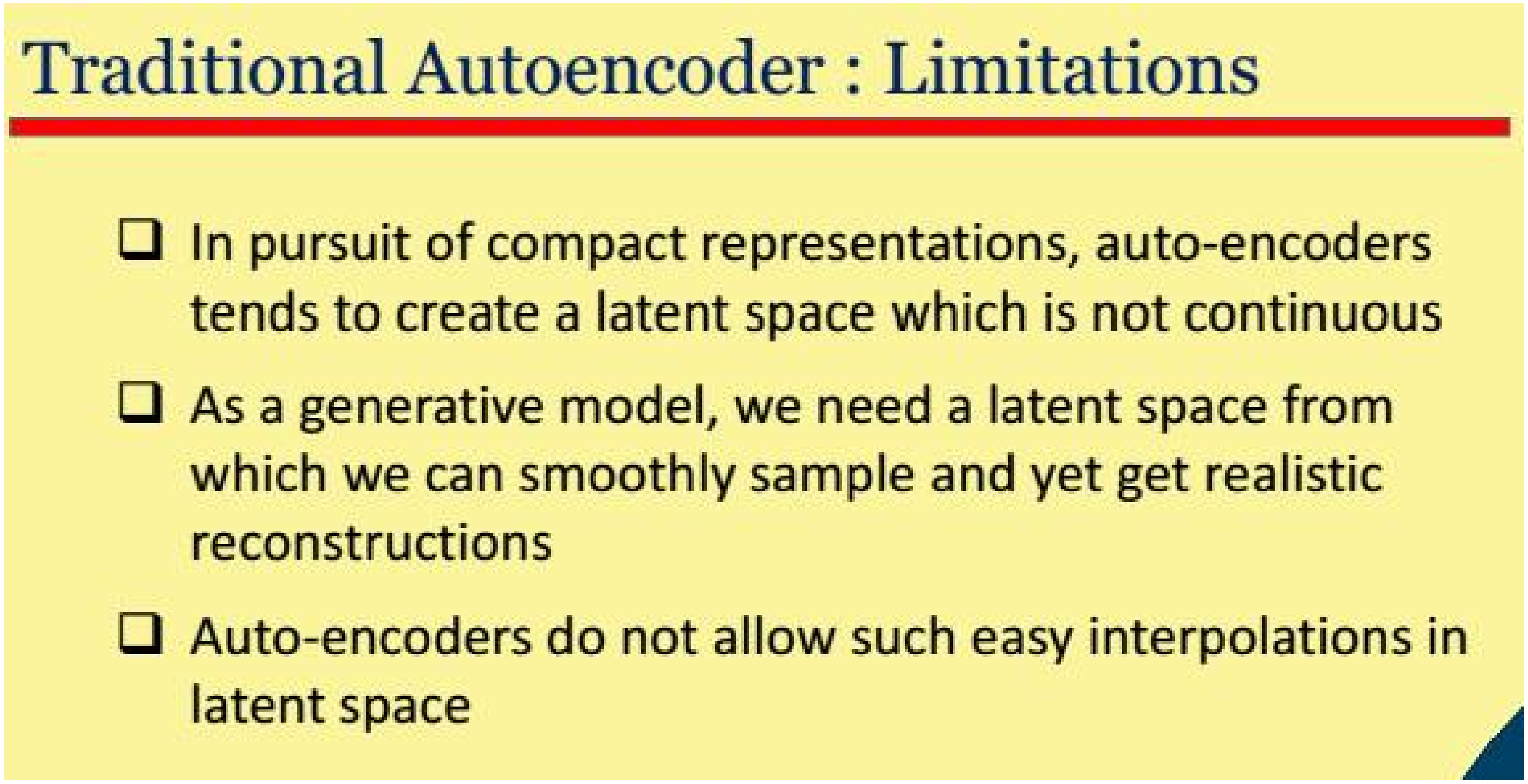
In the fine-tuning stage, with the network having suitable features at each layer, we can adopt the standard supervised learning method and the gradient descent algorithm to adjust the parameters at each layer.

Variational Autoencoder



Traditional Autoencoder

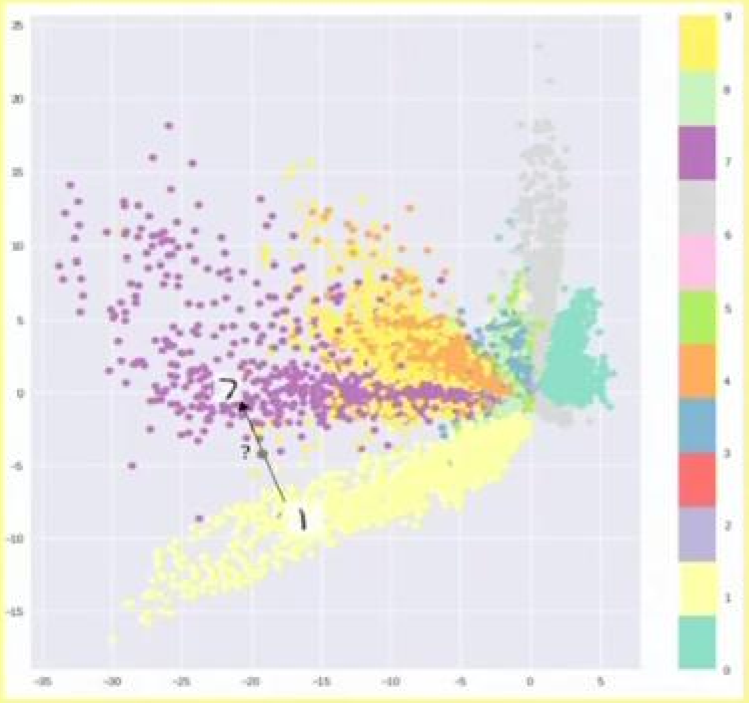
- Maps an input image via an encoder to a deterministic latent code
- Decoder maps the latent code to reconstruct the input image



Traditional Autoencoder : Limitations

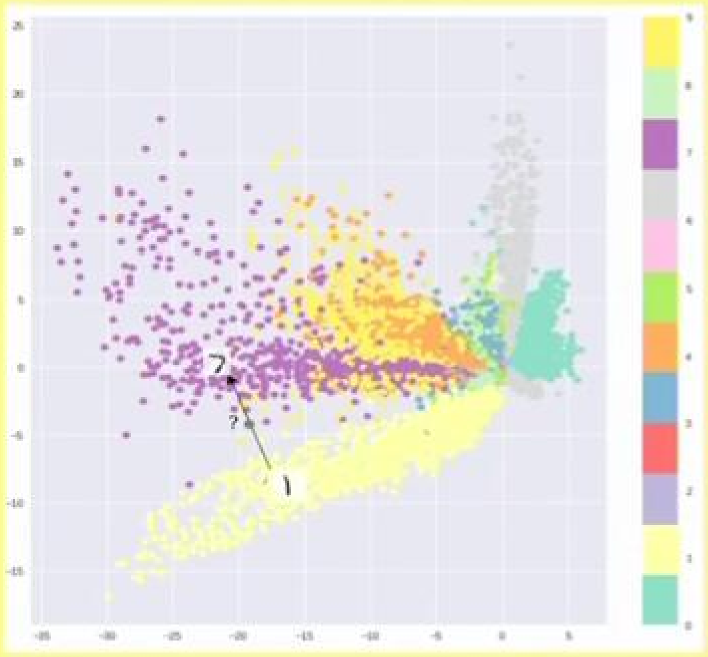
- In pursuit of compact representations, auto-encoders tends to create a latent space which is not continuous
- As a generative model, we need a latent space from which we can smoothly sample and yet get realistic reconstructions
- Auto-encoders do not allow such easy interpolations in latent space

Traditional Autoencoder : Limitations

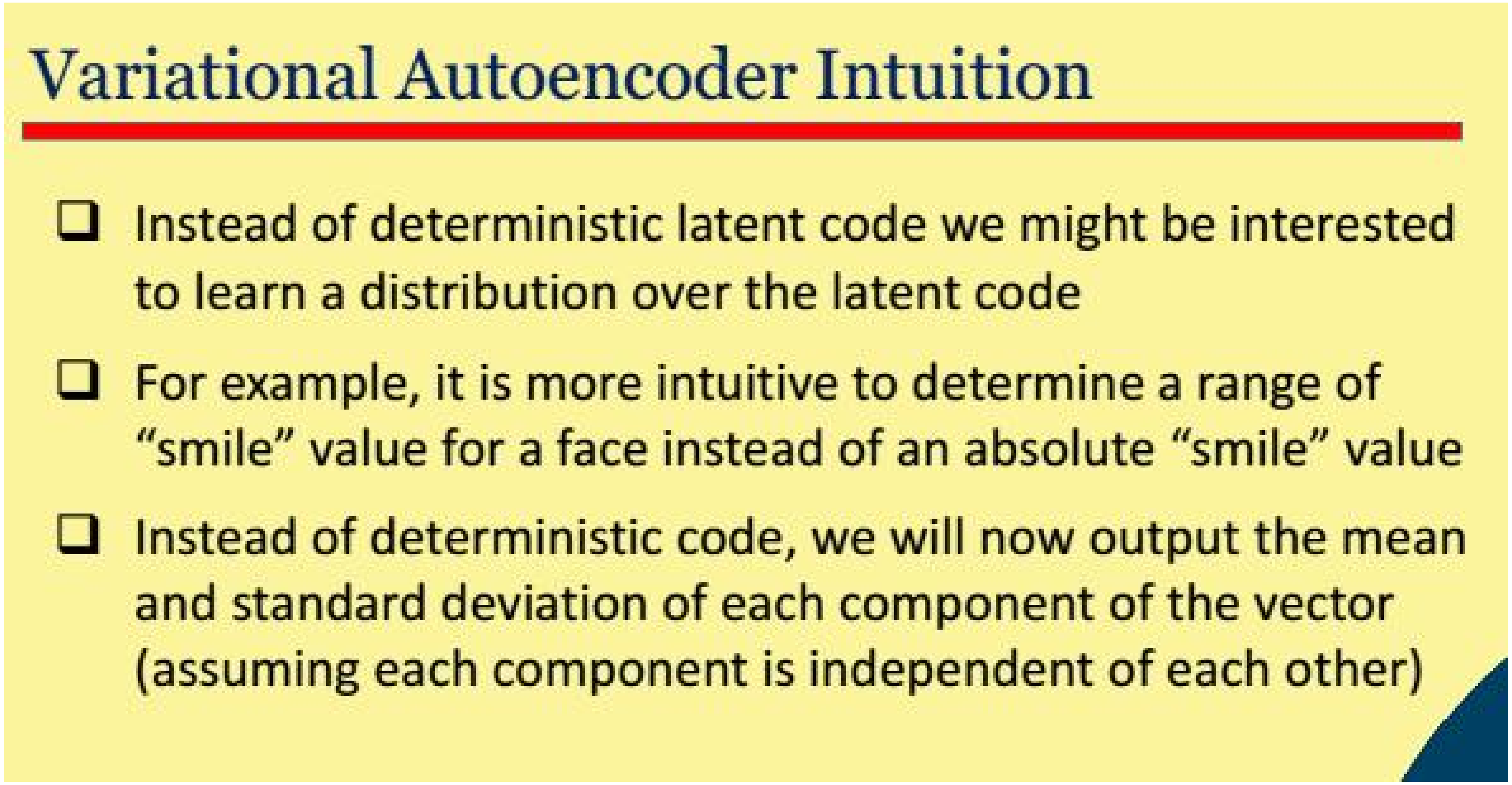


- ❑ Distinct cluster for each class
- ❑ Not easy for decoder to reconstruct since we need different distinct codes for each image

Traditional Autoencoder : Limitations



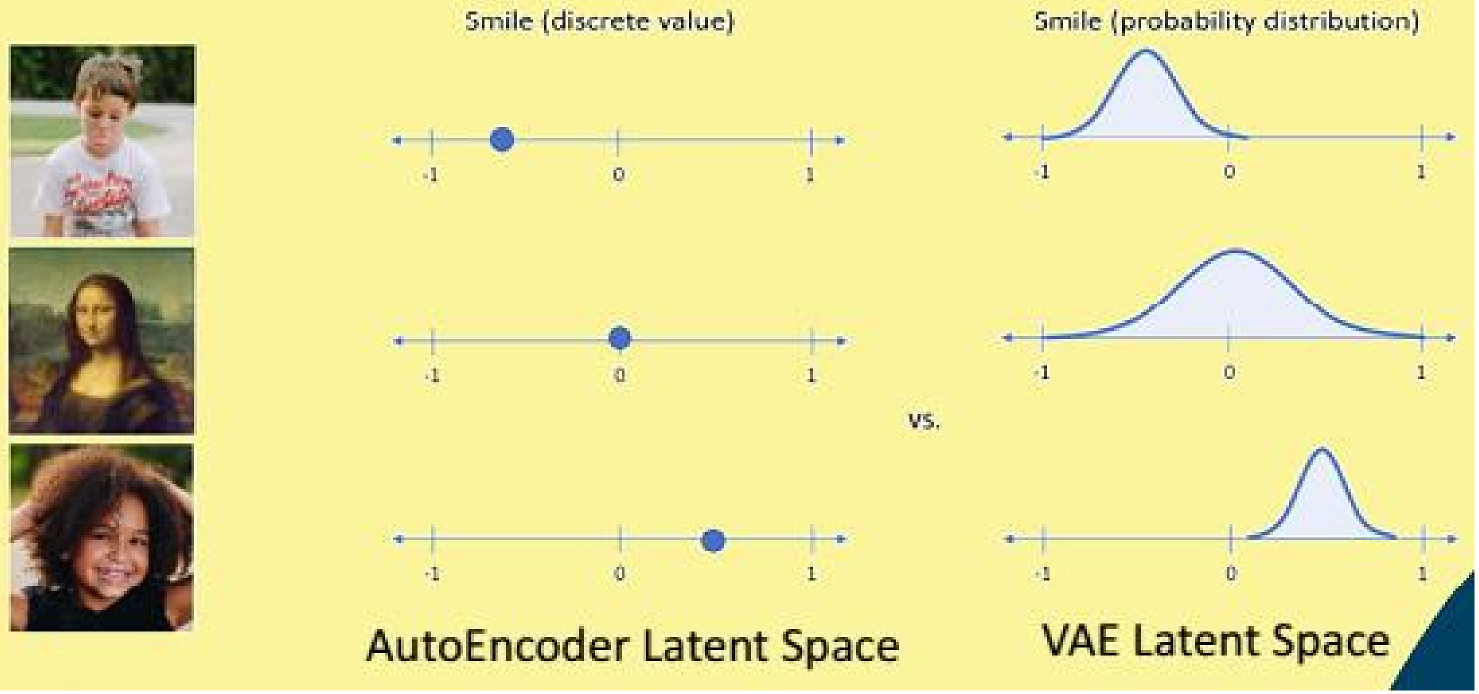
- Discontinuous latent space means decoder never reconstructed from such unexplored points
- If we sample from such points, decoder will give unrealistic output
- **Aim:** Try to make latent space continuous yet maintain the class specific compactness

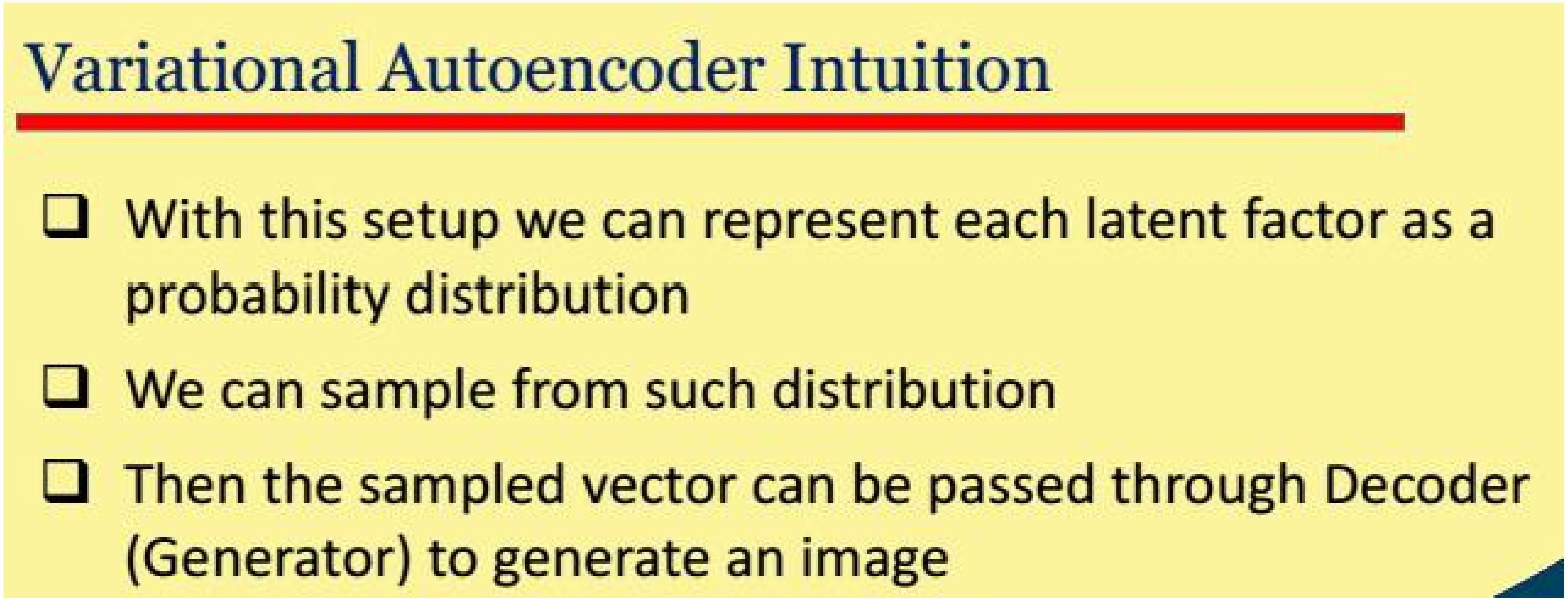


Variational Autoencoder Intuition

- ❑ Instead of deterministic latent code we might be interested to learn a distribution over the latent code
- ❑ For example, it is more intuitive to determine a range of “smile” value for a face instead of an absolute “smile” value
- ❑ Instead of deterministic code, we will now output the mean and standard deviation of each component of the vector (assuming each component is independent of each other)

Autoencoder Intuition vs. VAE Latent Space

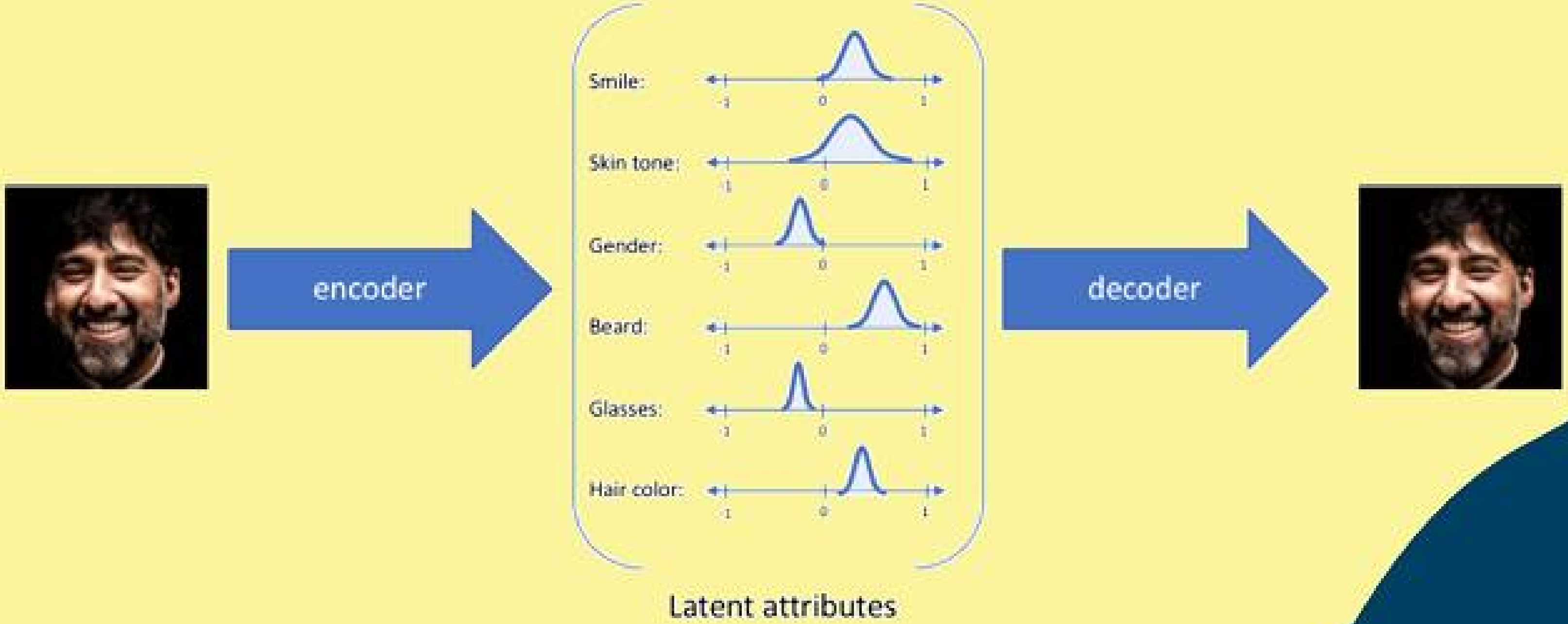


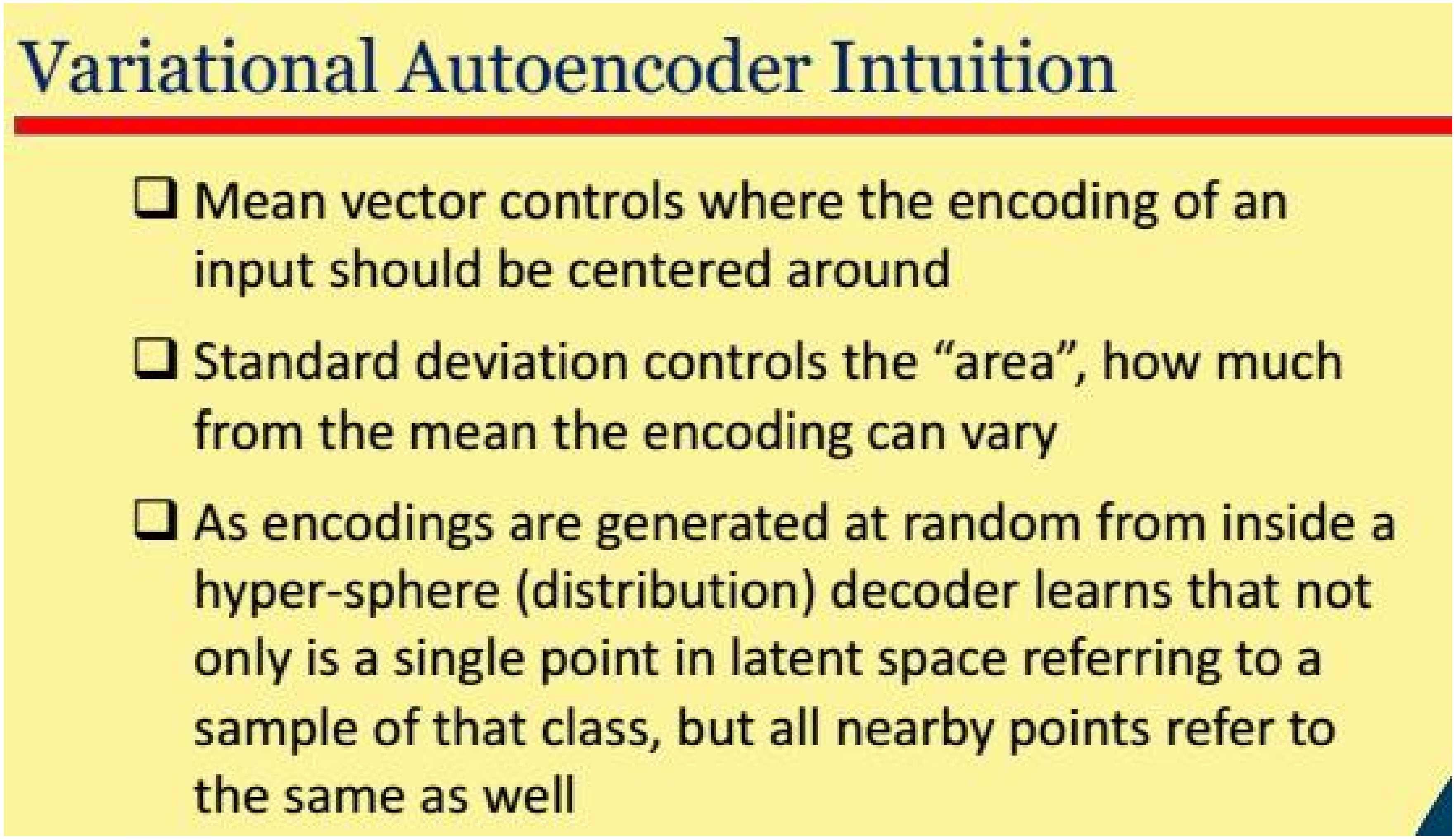


Variational Autoencoder Intuition

- With this setup we can represent each latent factor as a probability distribution
- We can sample from such distribution
- Then the sampled vector can be passed through Decoder (Generator) to generate an image

Variational Autoencoder Intuition



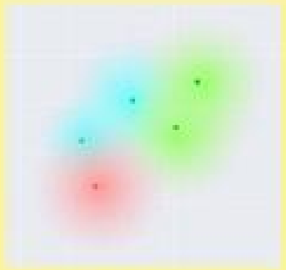


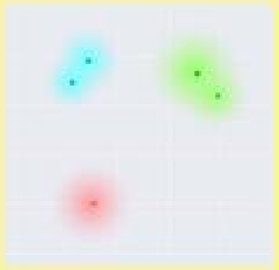
Variational Autoencoder Intuition

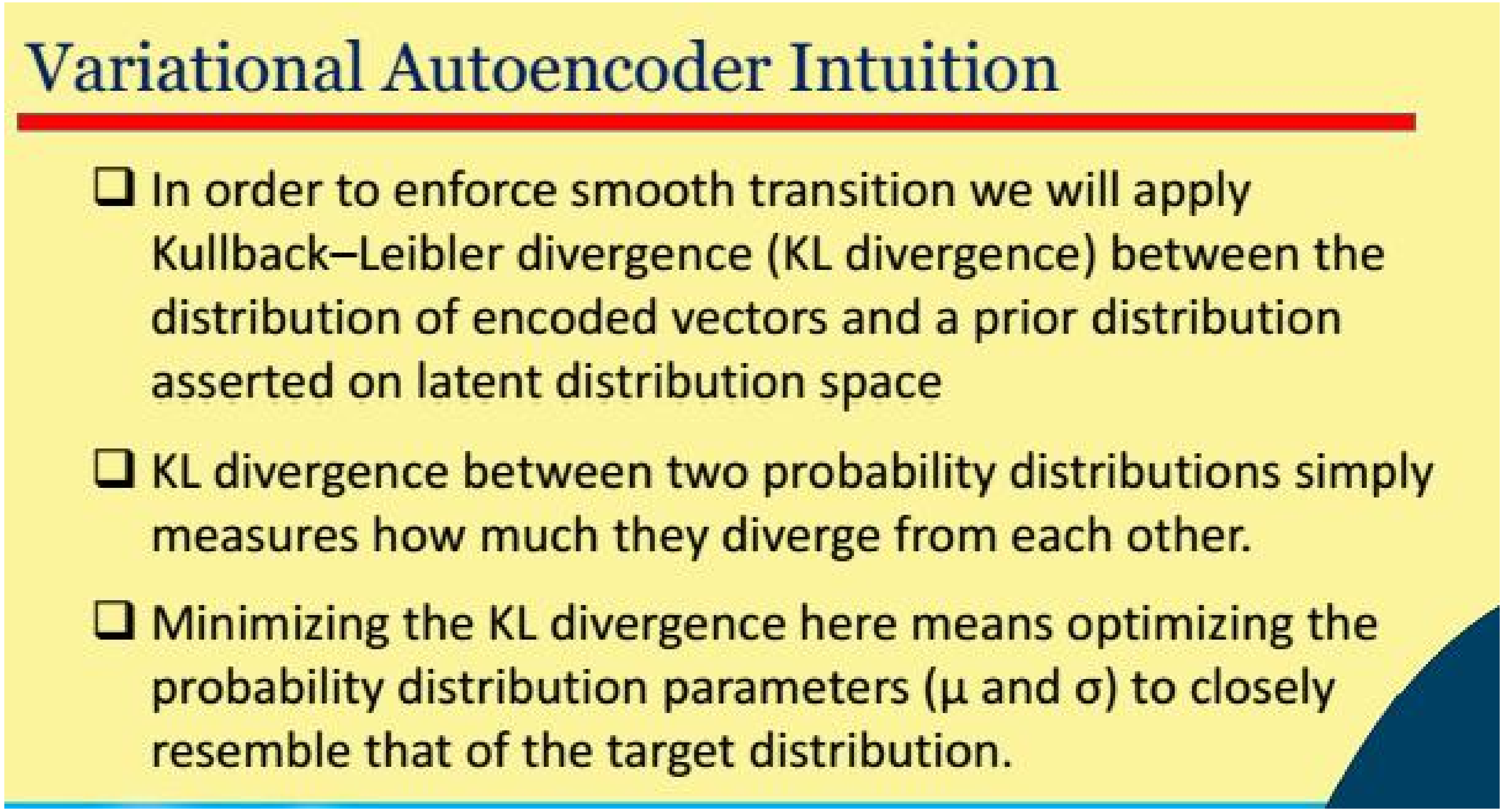
- ❑ Mean vector controls where the encoding of an input should be centered around
- ❑ Standard deviation controls the “area”, how much from the mean the encoding can vary
- ❑ As encodings are generated at random from inside a hyper-sphere (distribution) decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well

Variational Autoencoder Intuition

- For smooth interpolations, ideally, we want overlap between samples that are not very similar too, in order to interpolate between classes.
- However μ and σ can take any value and learn to cluster the mean vectors of different classes far apart (and minimize σ) to reduce uncertainty for the Decoder


Our goal


Network might converge to



Variational Autoencoder Intuition

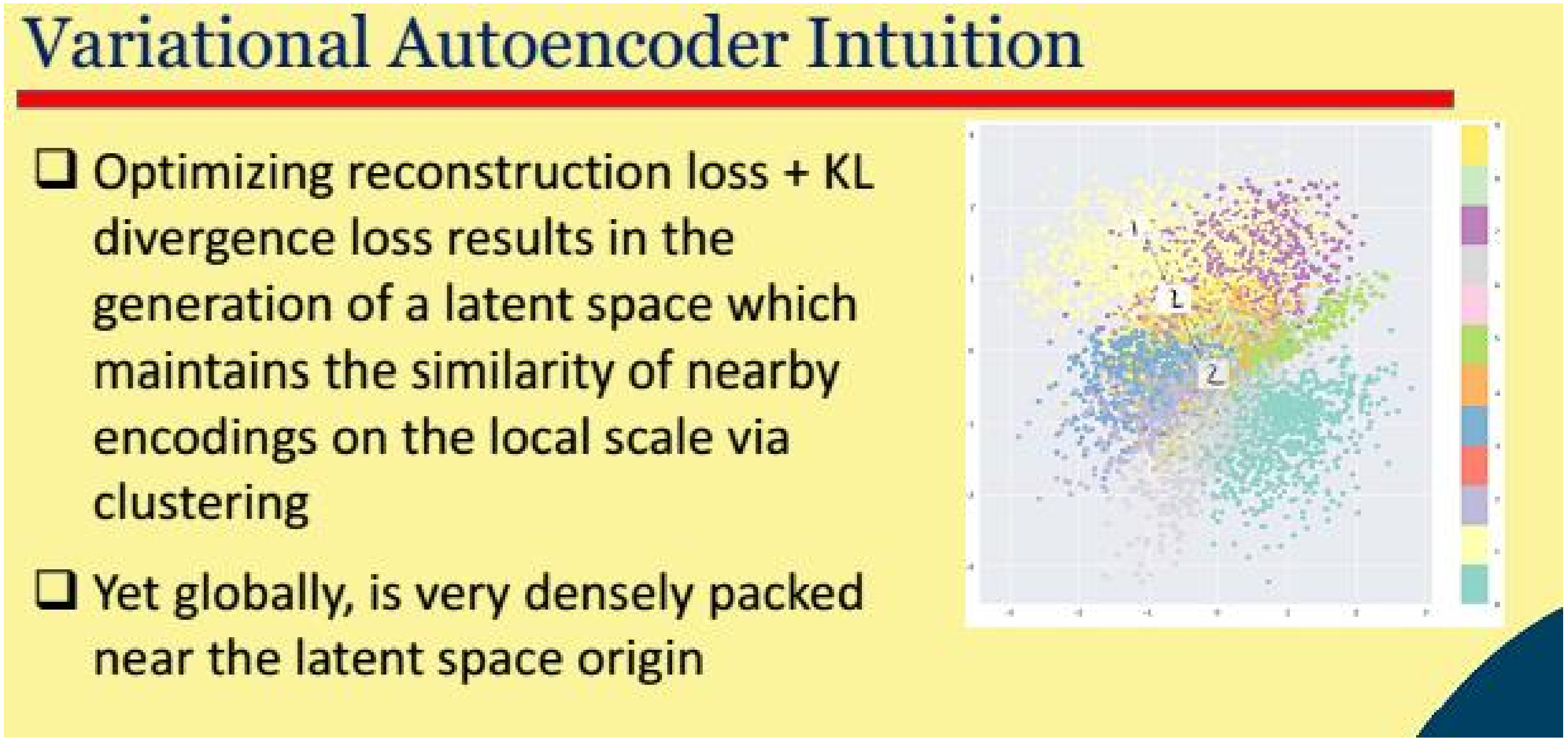
- ❑ In order to enforce smooth transition we will apply Kullback–Leibler divergence (KL divergence) between the distribution of encoded vectors and a prior distribution asserted on latent distribution space
- ❑ KL divergence between two probability distributions simply measures how much they diverge from each other.
- ❑ Minimizing the KL divergence here means optimizing the probability distribution parameters (μ and σ) to closely resemble that of the target distribution.

Variational Autoencoder Intuition

- In VAE, it is usually assumed that the distribution of the latent space follows a zero mean Normal distribution with diagonal covariance matrix (each component is independent of the other)
- KL divergence loss will encourage encodings from different inputs to be clustered about the center of the latent space
- If network creates clusters in specific regions then KL divergence loss will penalize such clusters formation

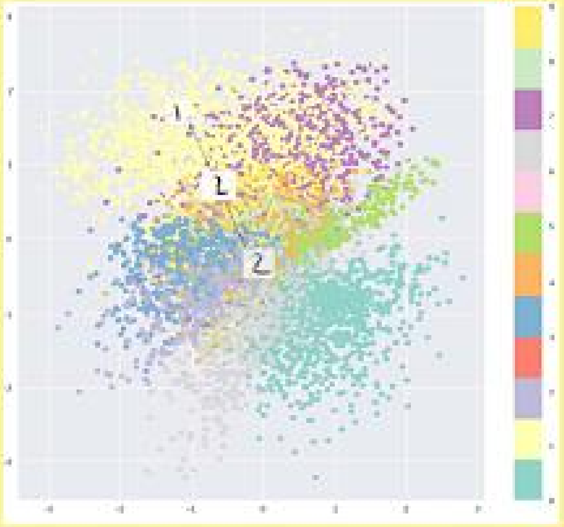
Variational Autoencoder Intuition

- ❑ But, only KL loss results in a latent space encodings densely placed randomly, near the center of the target distribution, with little regard for similarity/dis-similarity of input samples.
- ❑ The decoder finds it impossible to decode anything meaningful from this space, simply because there really isn't any structure/context specific meaning.



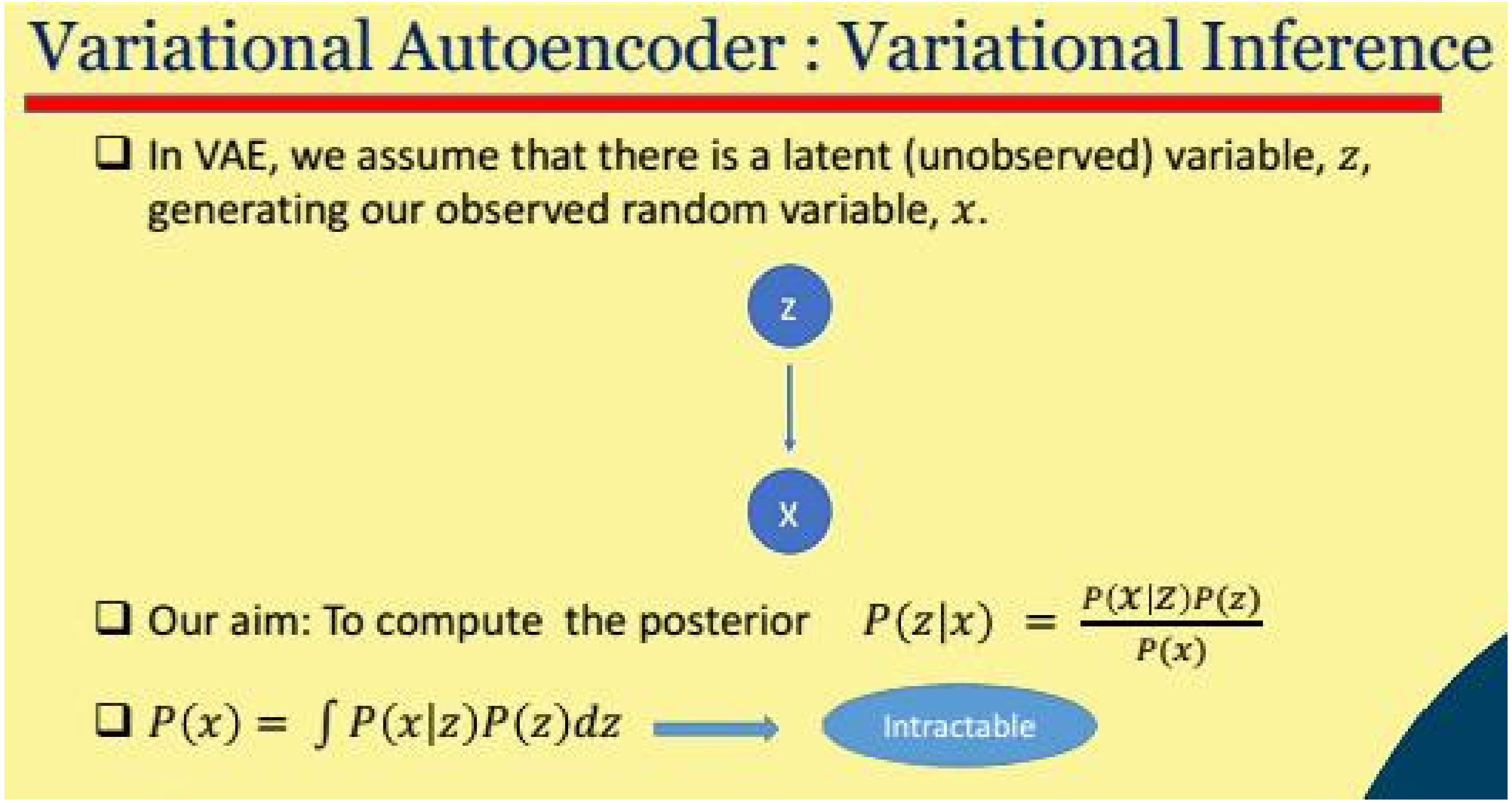
Variational Autoencoder Intuition

- ❑ Optimizing reconstruction loss + KL divergence loss results in the generation of a latent space which maintains the similarity of nearby encodings on the local scale via clustering
- ❑ Yet globally, is very densely packed near the latent space origin



Variational Autoencoder Intuition

- This equilibrium is attributed to cluster-forming nature of the reconstruction loss, and the dense packing nature of the KL loss
- It means when randomly generating, if you sample a vector from the prior distribution, $P(z)$ of latent space, the Decoder will successfully decode it.
- For interpolation, since there is no sudden gap between clusters, but a smooth mix of features, a Decoder can understand.

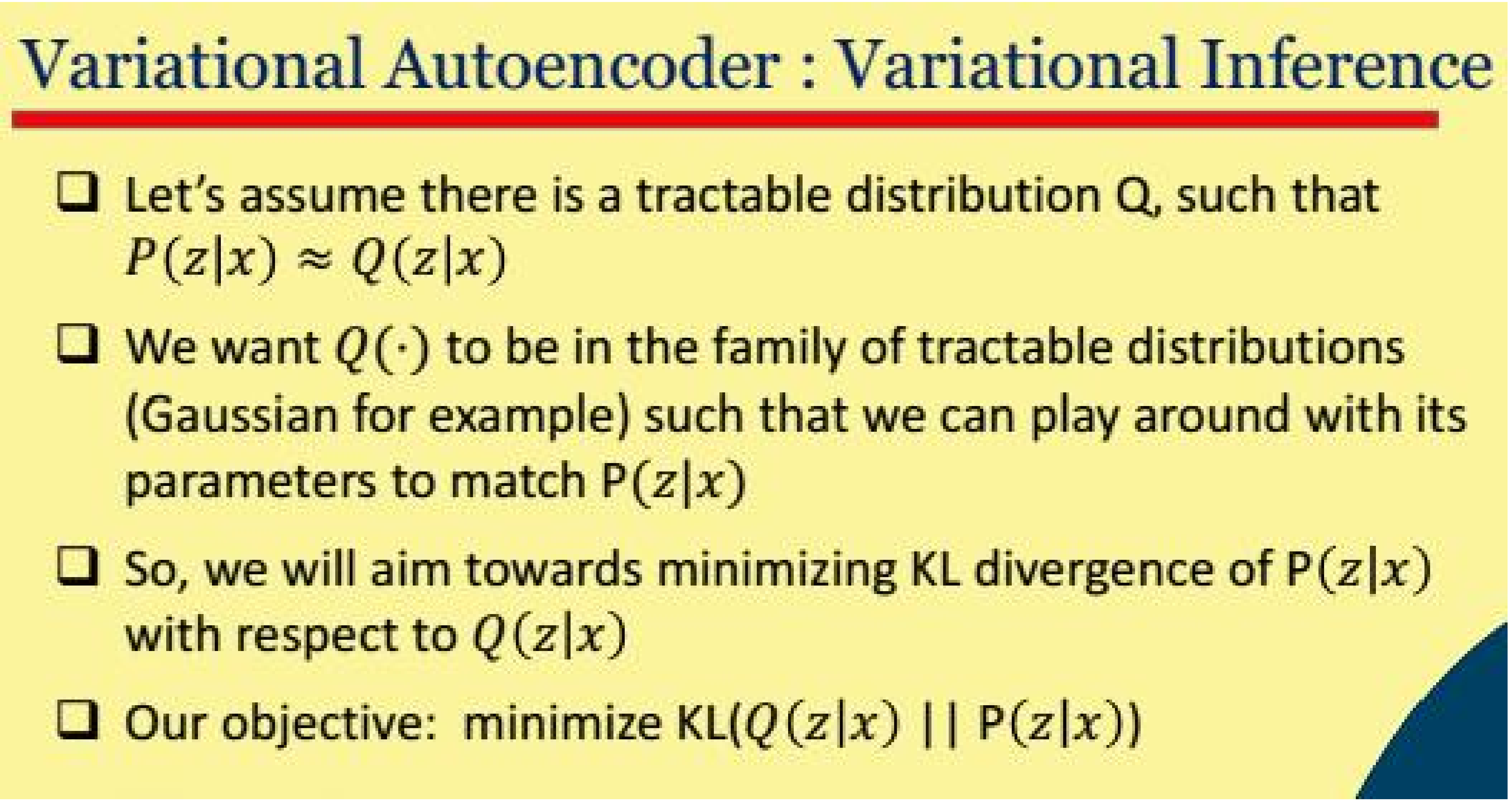


Variational Autoencoder : Variational Inference

- In VAE, we assume that there is a latent (unobserved) variable, z, generating our observed random variable, x.

$$\begin{tikzcd} z \arrow[d] \\ x \end{tikzcd}$$

- Our aim: To compute the posterior $P(z|x) = \frac{P(X|Z)P(z)}{P(x)}$
- $P(x) = \int P(x|z)P(z)dz \longrightarrow$ Intractable

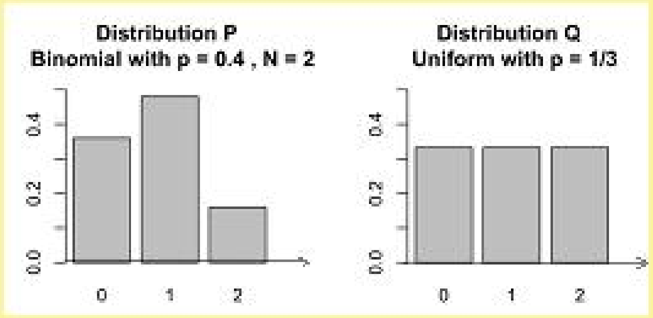


Variational Autoencoder : Variational Inference

- Let's assume there is a tractable distribution Q, such that $P(z|x) \approx Q(z|x)$
- We want $Q(\cdot)$ to be in the family of tractable distributions (Gaussian for example) such that we can play around with its parameters to match P$(z|x)$
- So, we will aim towards minimizing KL divergence of P$(z|x)$ with respect to $Q(z|x)$
- Our objective: minimize KL($Q(z|x)$ || P$(z|x)$)

KL Divergence

$$\text{KL}(Q(z|x)\ ||\ \text{P}(z|x)) = \sum_x Q(x) \log \frac{Q(x)}{P(x)}$$



x	0	1	2
P(x)	0.36	0.48	0.16
Q(x)	0.33	0.33	0.33

KL Divergence

$$KL(P||Q) = \sum_x P(x) log \frac{P(x)}{Q(x)}$$

$$= 0.36 \log\left(\frac{0.36}{0.33}\right) + 0.48\ log\left(\frac{0.48}{0.33}\right) + 0.16\ log\left(\frac{0.16}{0.33}\right) = 0.0414$$

x	0	1	2
P(x)	0.36	0.48	0.16
Q(x)	0.33	0.33	0.33

$$KL(Q||P) = \sum_x Q(x) log \frac{Q(x)}{P(x)}$$

$$= 0.33 \log\left(\frac{0.33}{0.36}\right) + 0.33\ log\left(\frac{0.33}{0.48}\right) + 0.33\ log\left(\frac{0.33}{0.16}\right) = 0.0375$$

KL Divergence

$$KL(Q(z|x)||P(z|x))$$

$$= -\sum_z Q(z|x)\log\frac{P(z|x)}{Q(z|x)}$$

$$= -\sum_z Q(z|x)\log\frac{P(x,z)}{P(x) * Q(z|x)}$$

$$= -\sum_z Q(z|x)\left\{log\frac{P(x,z)}{Q(z|x)} - logP(x)\right\}$$

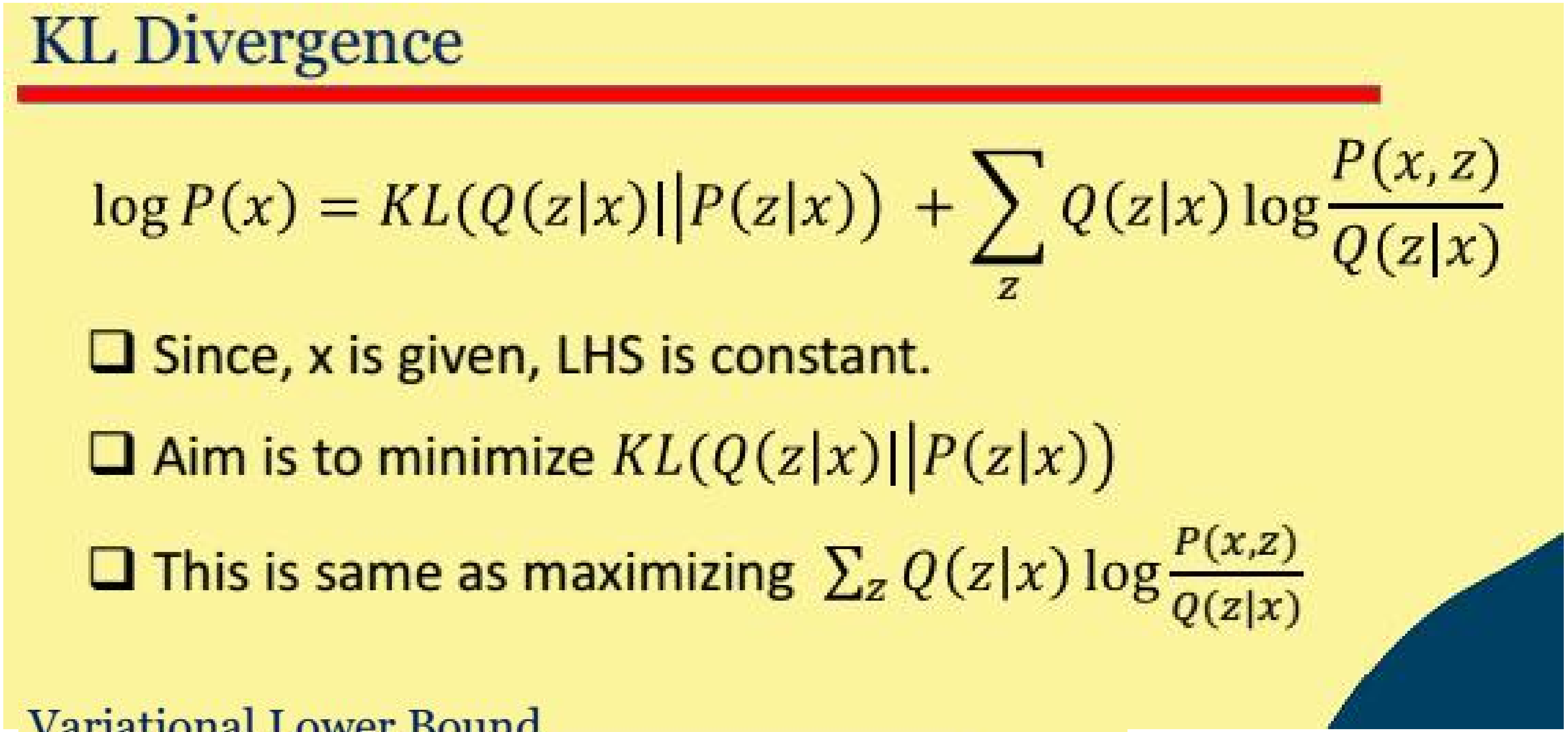
$$= -\sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)} + \sum_z Q(z|x)\log P(x)$$

$$= -\sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)} + logP(x)$$

$$KL(Q(z|x)||P(z|x)) = -\sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)} + logP(x)$$

$$\Downarrow$$

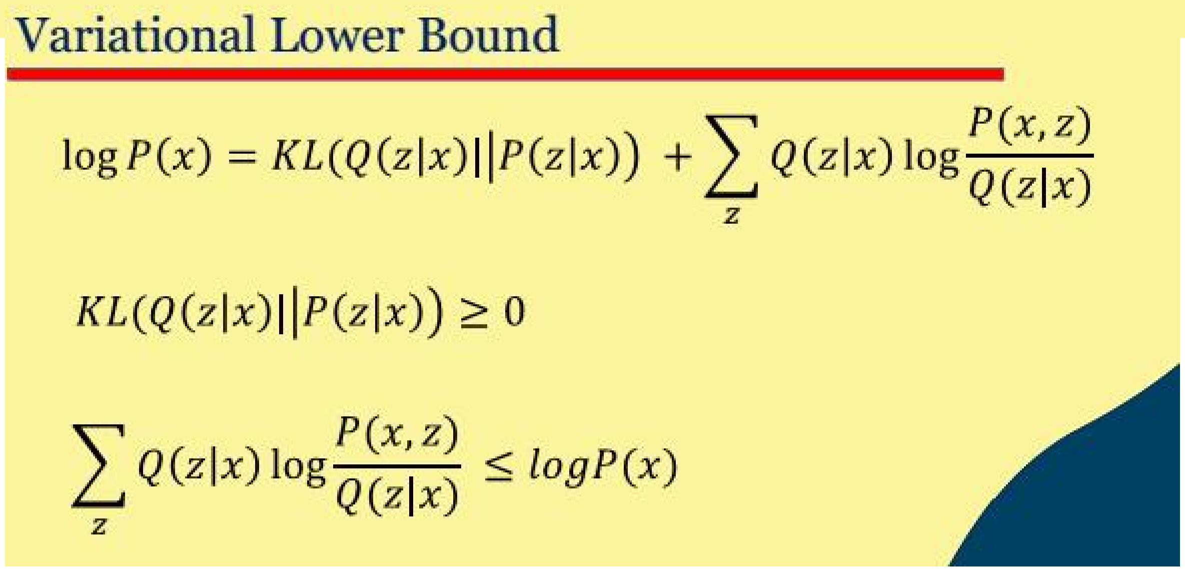
$$\log P(x) = KL(Q(z|x)||P(z|x)) + \sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)}$$



KL Divergence

$$\log P(x) = KL(Q(z|x)||P(z|x)) + \sum_{z} Q(z|x) \log \frac{P(x,z)}{Q(z|x)}$$

- ❑ Since, x is given, LHS is constant.
- ❑ Aim is to minimize $KL(Q(z|x)||P(z|x))$
- ❑ This is same as maximizing $\sum_z Q(z|x) \log \frac{P(x,z)}{Q(z|x)}$

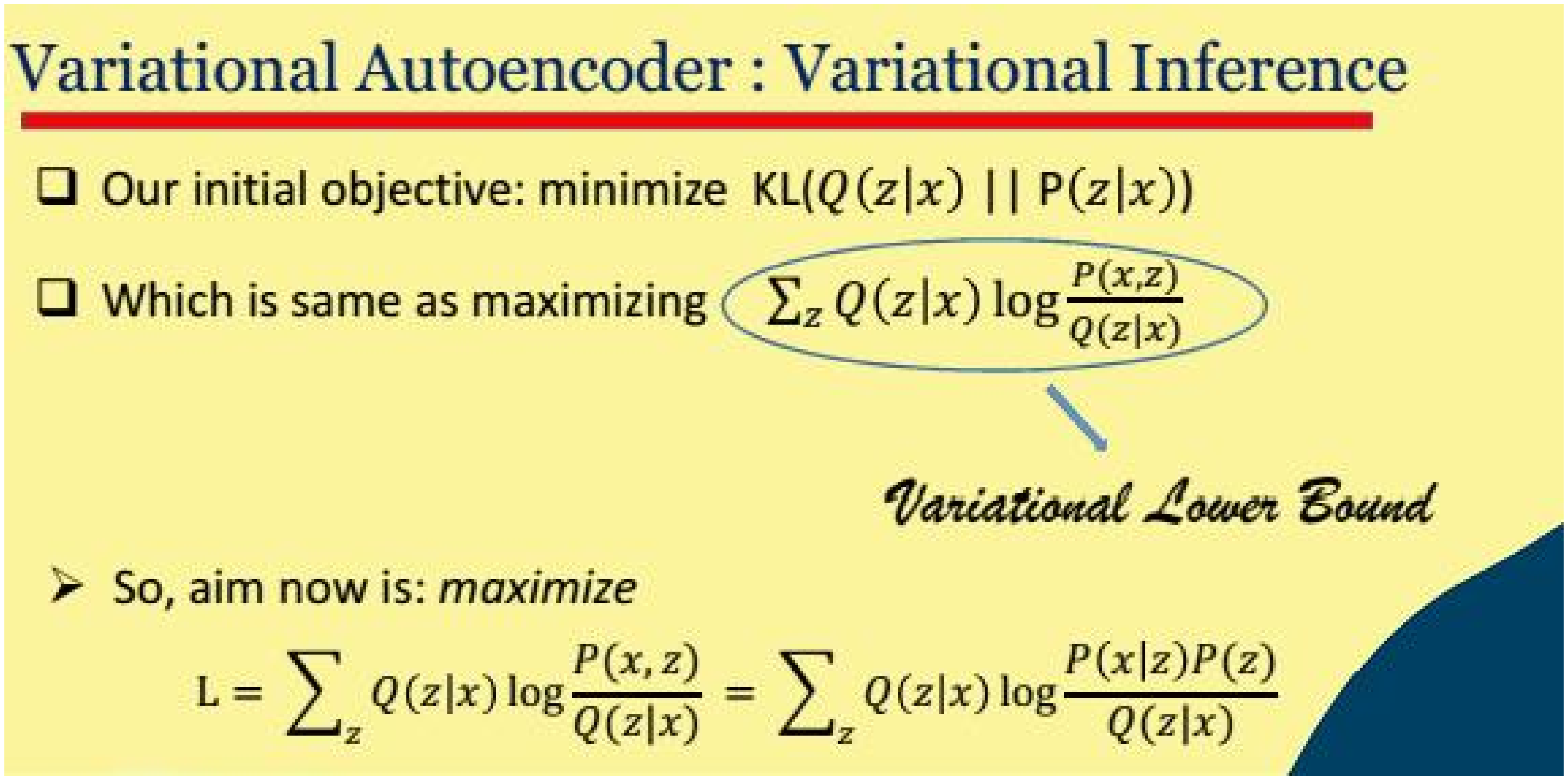


Variational Lower Bound

$$\log P(x) = KL(Q(z|x)||P(z|x)) + \sum_{z} Q(z|x) \log \frac{P(x,z)}{Q(z|x)}$$

$$KL(Q(z|x)||P(z|x)) \geq 0$$

$$\sum_{z} Q(z|x) \log \frac{P(x,z)}{Q(z|x)} \leq log P(x)$$



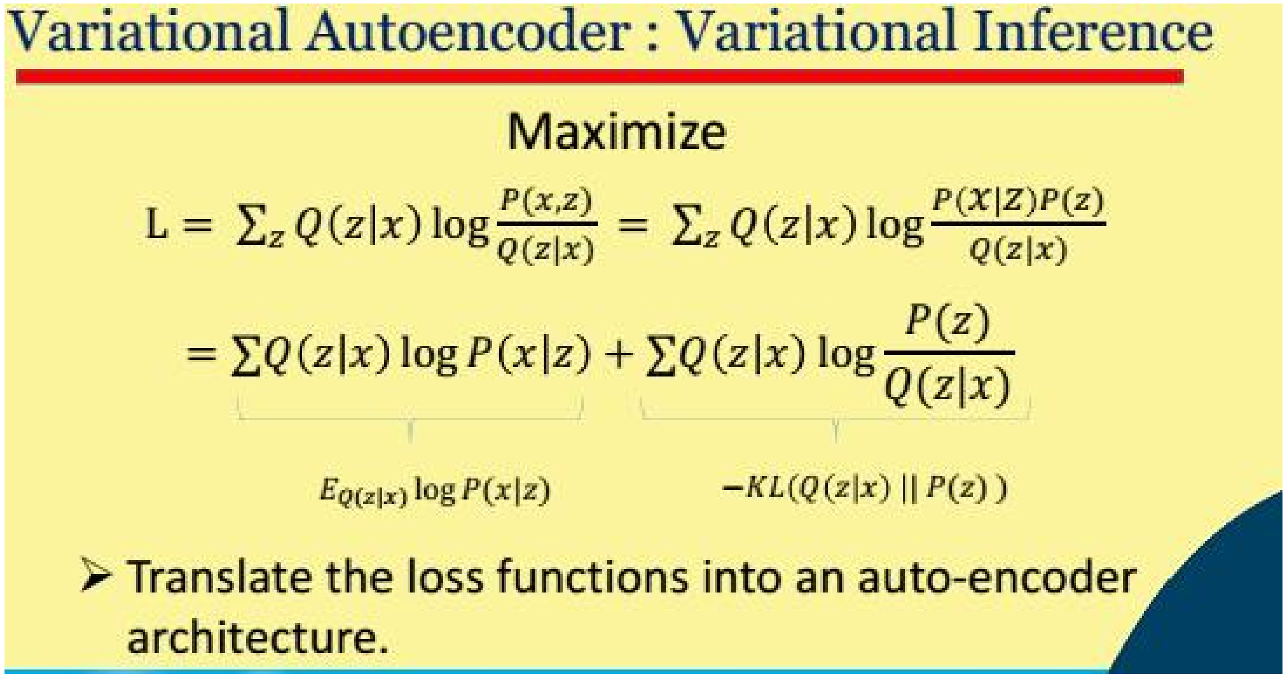
Variational Autoencoder : Variational Inference

- ❑ Our initial objective: minimize KL($Q(z|x)$ || P($z|x$))
- ❑ Which is same as maximizing $\sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)}$

Variational Lower Bound

➢ So, aim now is: *maximize*

$$L = \sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)} = \sum_z Q(z|x)\log\frac{P(x|z)P(z)}{Q(z|x)}$$



Variational Autoencoder : Variational Inference

Maximize

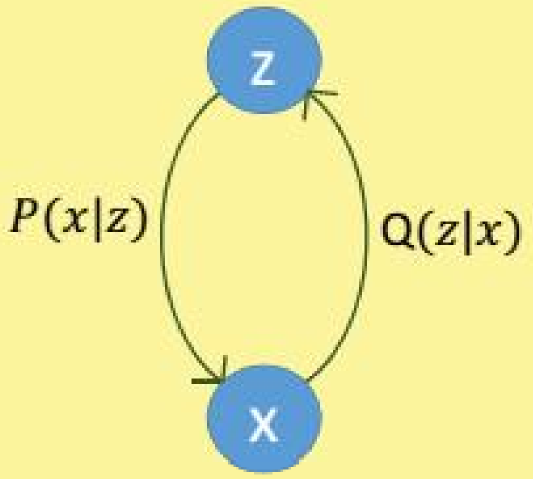
$$\mathrm{L} = \sum_z Q(z|x)\log\frac{P(x,z)}{Q(z|x)} = \sum_z Q(z|x)\log\frac{P(X|Z)P(z)}{Q(z|x)}$$

$$= \underbrace{\sum Q(z|x)\log P(x|z)}_{E_{Q(z|x)}\log P(x|z)} + \underbrace{\sum Q(z|x)\log\frac{P(z)}{Q(z|x)}}_{-KL(Q(z|x)\,||\,P(z))}$$

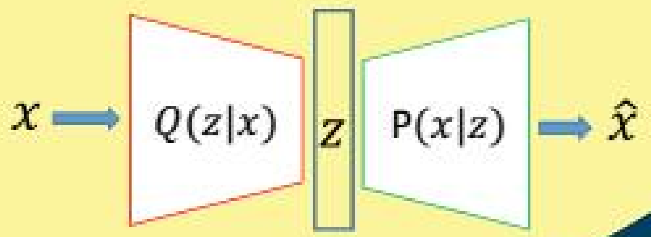
- ➢ Translate the loss functions into an auto-encoder architecture.

Variational Autoencoder : Network Realization

- ❑ We have the following graphical model



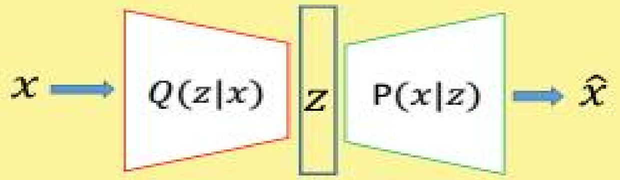
- ❑ Realize both $P(\cdot)$ and $Q(\cdot)$ with neural networks



Variational Autoencoder : Network Realization

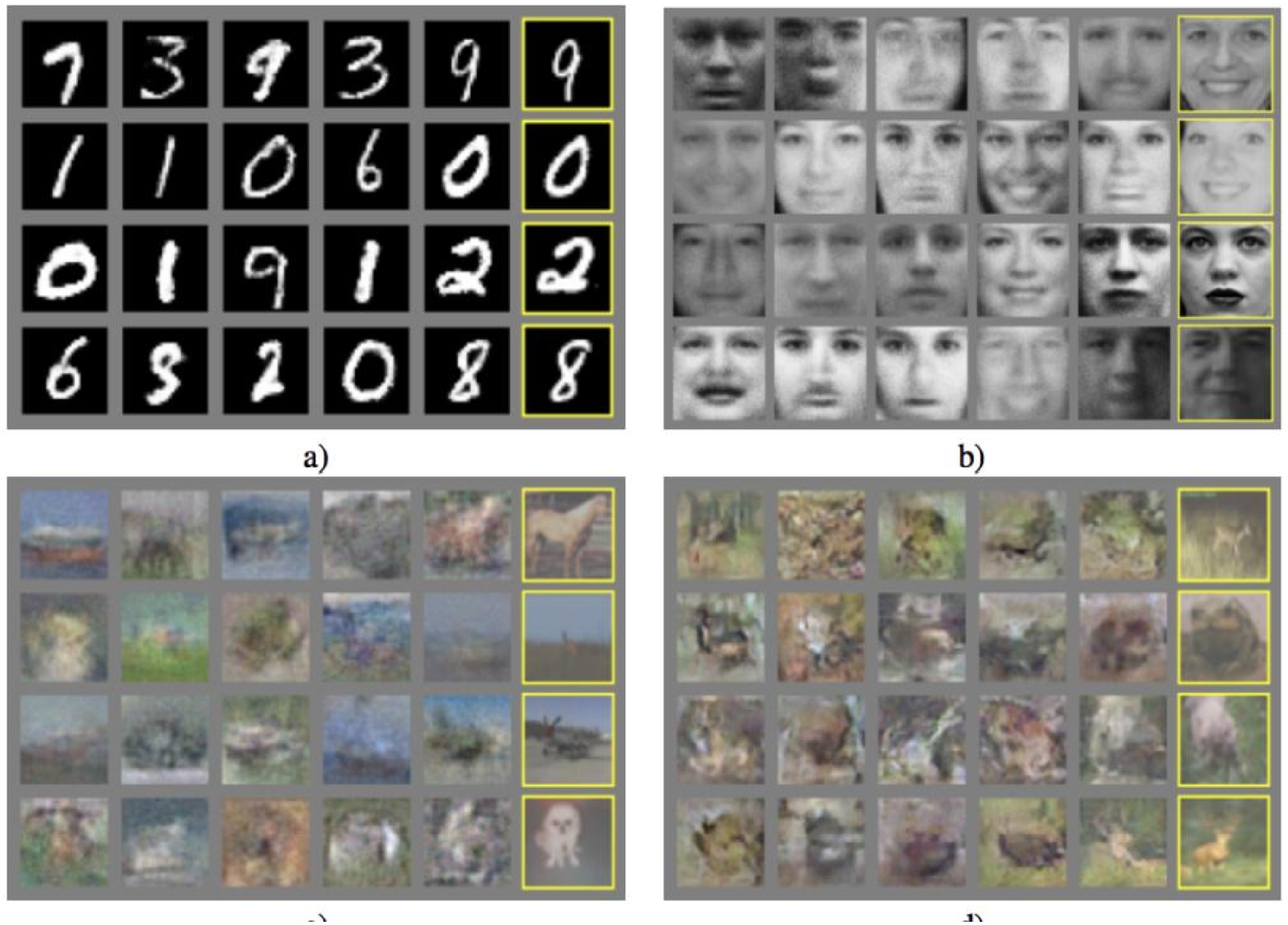
- ❑ The z codes we get here should match with the distribution of $P(z)$ and we can decide what prior distribution to choose for $P(z)$.

- ❑ Usual practice is to select a Normal distribution $N(0, I)$ for the prior.

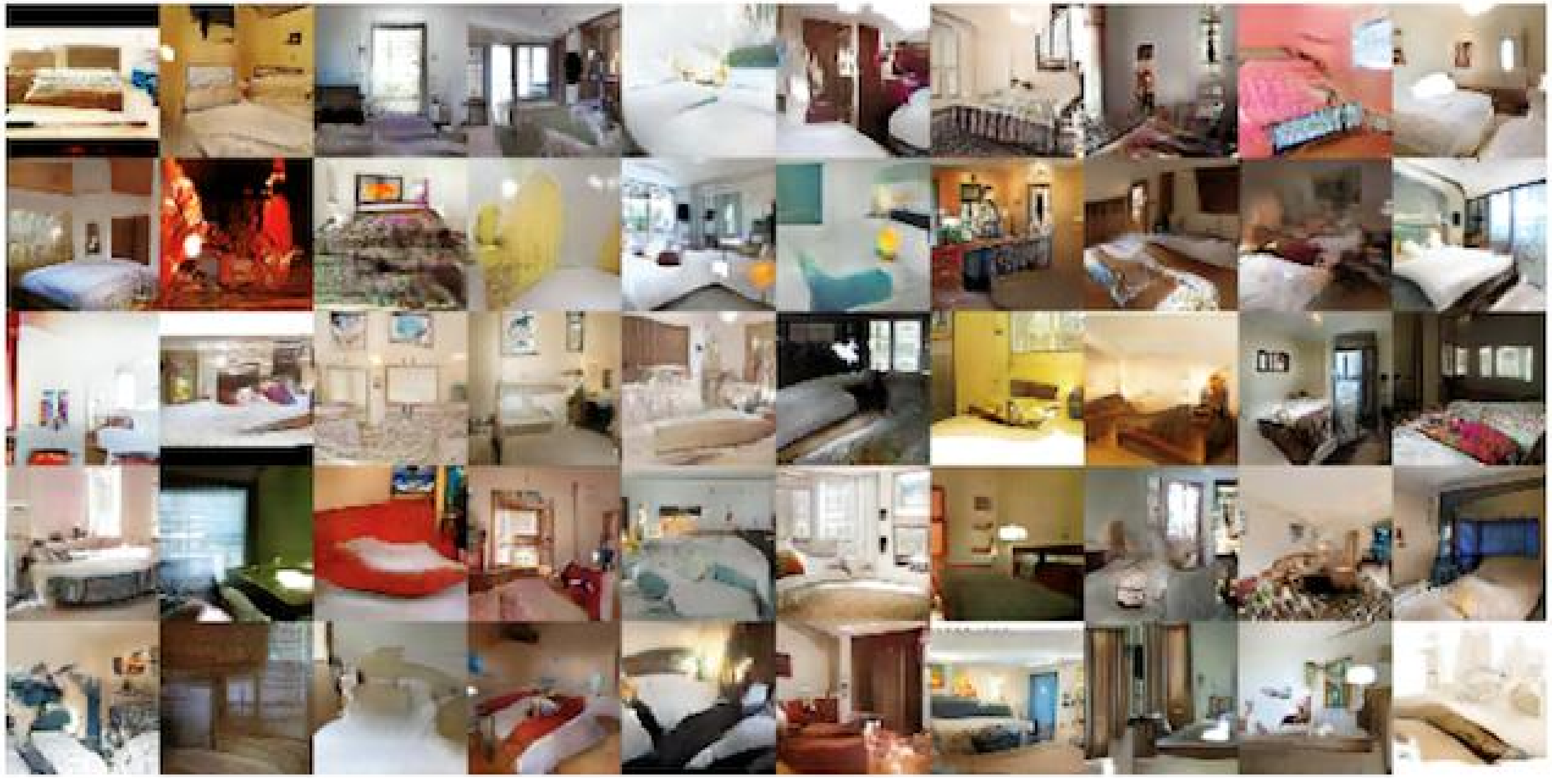


Generative Adversarial Network (GAN)

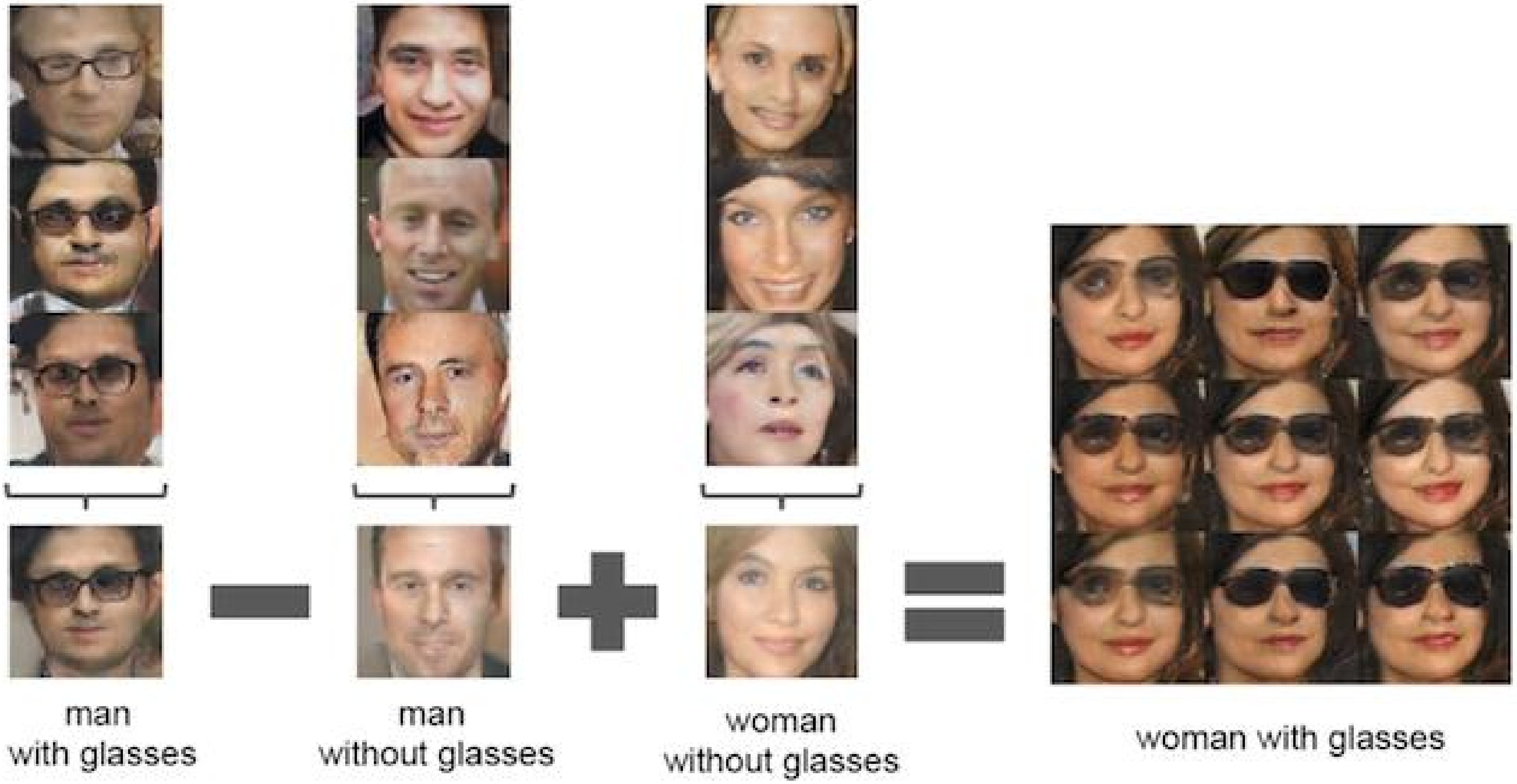
https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/



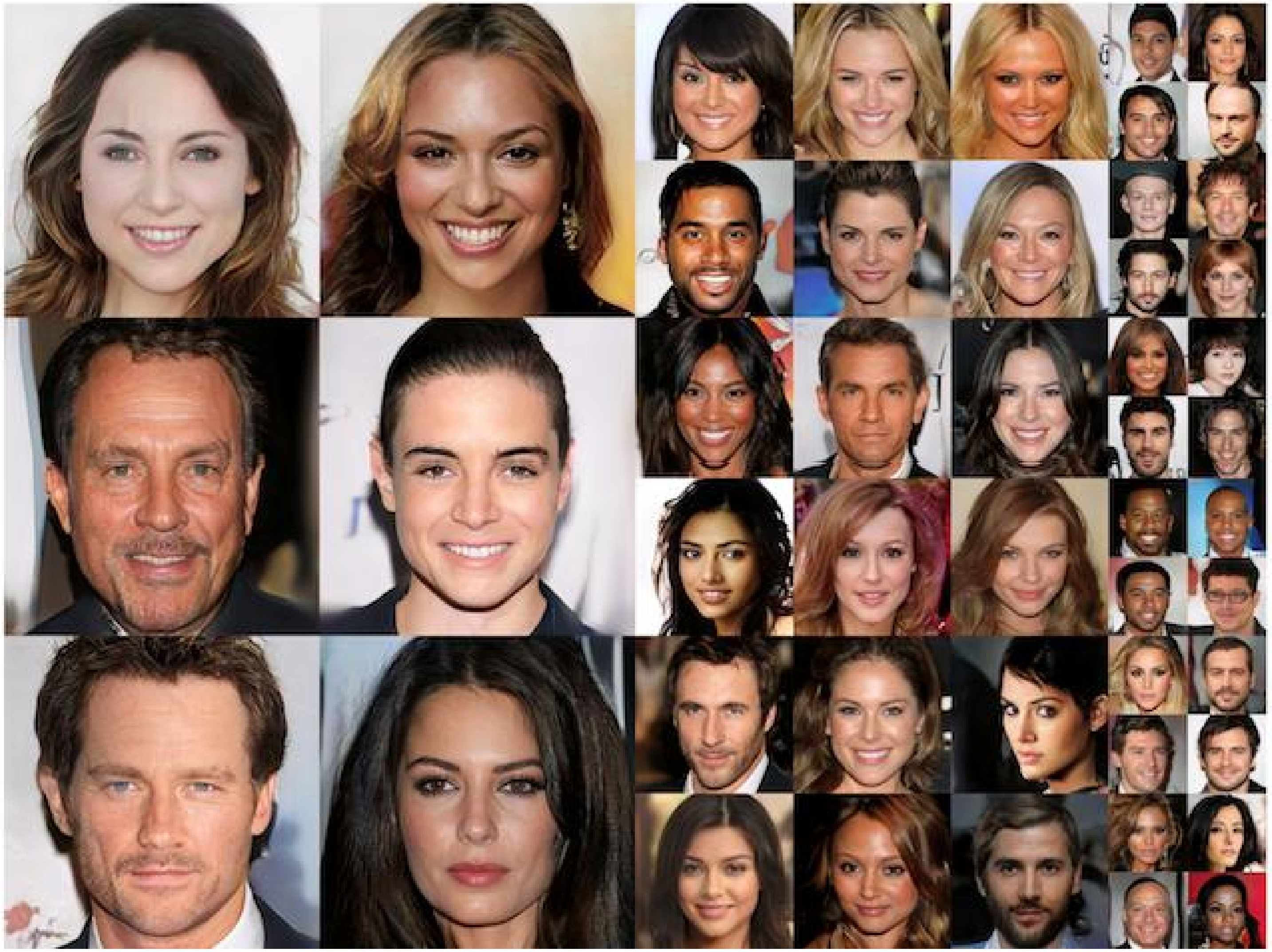
Examples of GANs used to Generate New Plausible Examples for Image Datasets.Taken from Generative Adversarial Nets, 2014



Example of GAN-Generated Photographs of Bedrooms.Taken from Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.



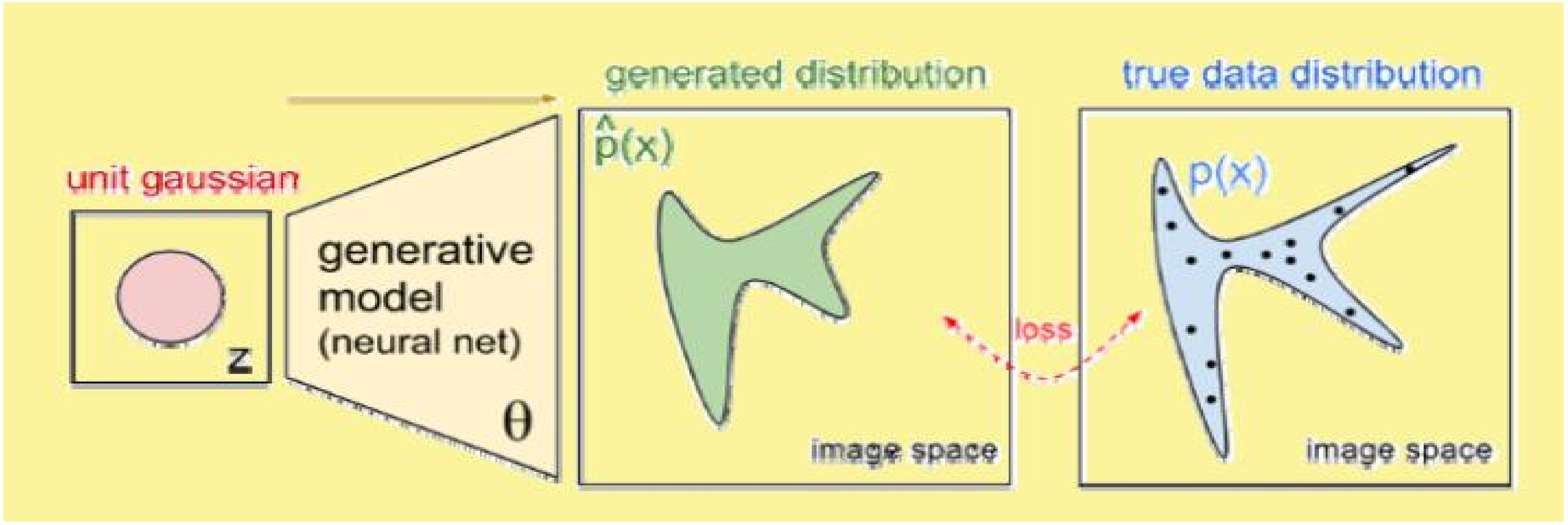
Generate Examples for Image Datasets



Generate Photographs of Human Faces

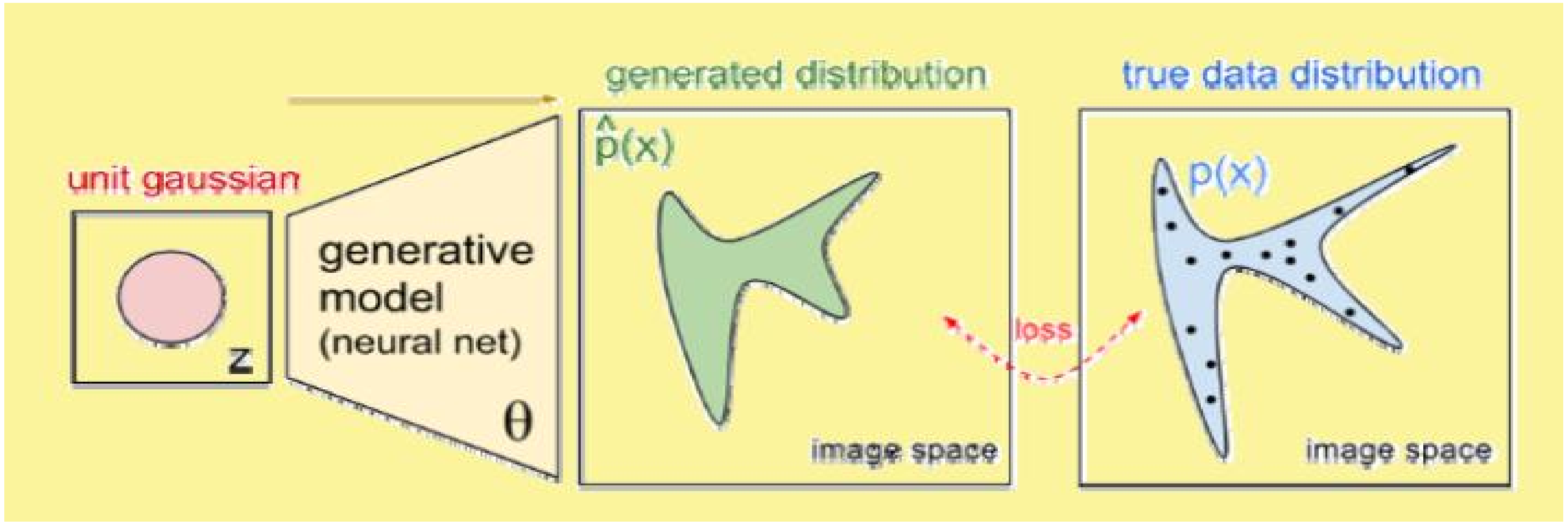
Implicit Generative Models

- Implicitly defines a probability distribution.
- Sample code vector, z, from a simple and fixed distribution (e.g. spherical Gaussian or Uniform).
- A generator network is trained as a differentiable network to map z to a data point x.



Implicit Generative Models

- ❑ Blue Region shows areas with high probability of real image.
- ❑ Black dots represent actual images from true distribution $p(x)$.
- ❑ Generative model (parameterized by θ) also describes a function $\hat{p}(x)$
  - ➢ Takes points (latent codes) from an unit Gaussian distribution.
  - ➢ Maps those points to a generator distribution.
  - ➢ θ can be optimized to reduce $KL(p(x)||\hat{p}(x))$
  - ➢ Green distribution starts randomly then aligns with blue distribution

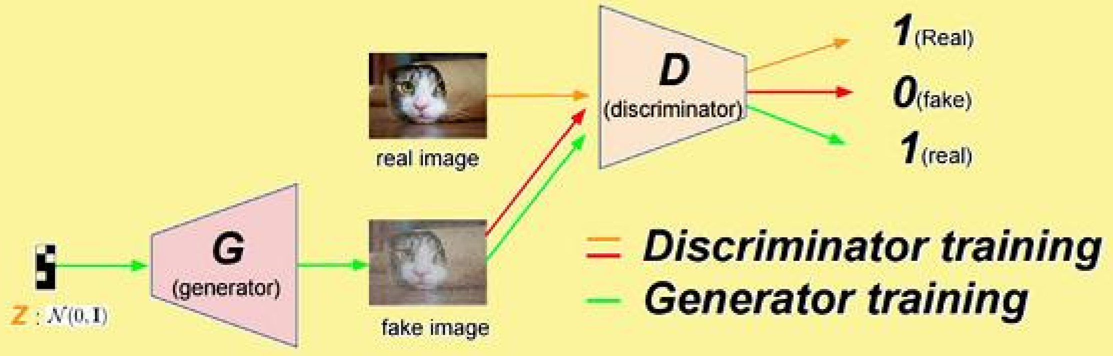


GAN Overview

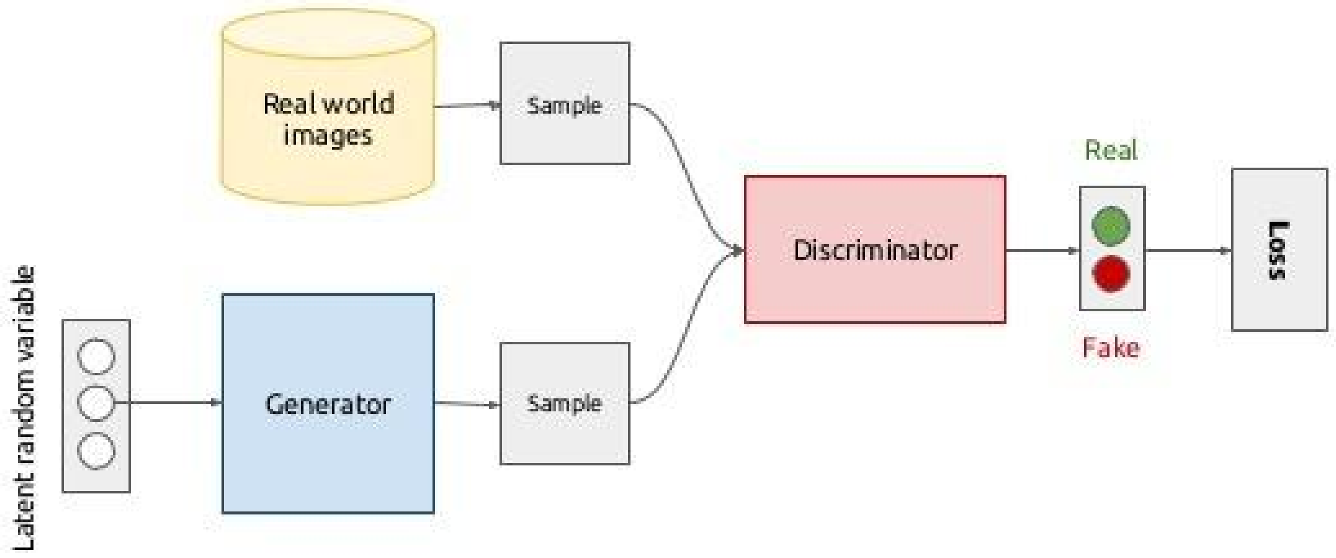
- In GAN the main idea is to have two neural networks compete with each other.
- Its Game Theoretic Approach.
  - **Generator** network samples a z vector and tries to produce realistic samples.
  - **Discriminator** network tries to distinguish fake samples (from Generator) and real samples.

GAN Overview

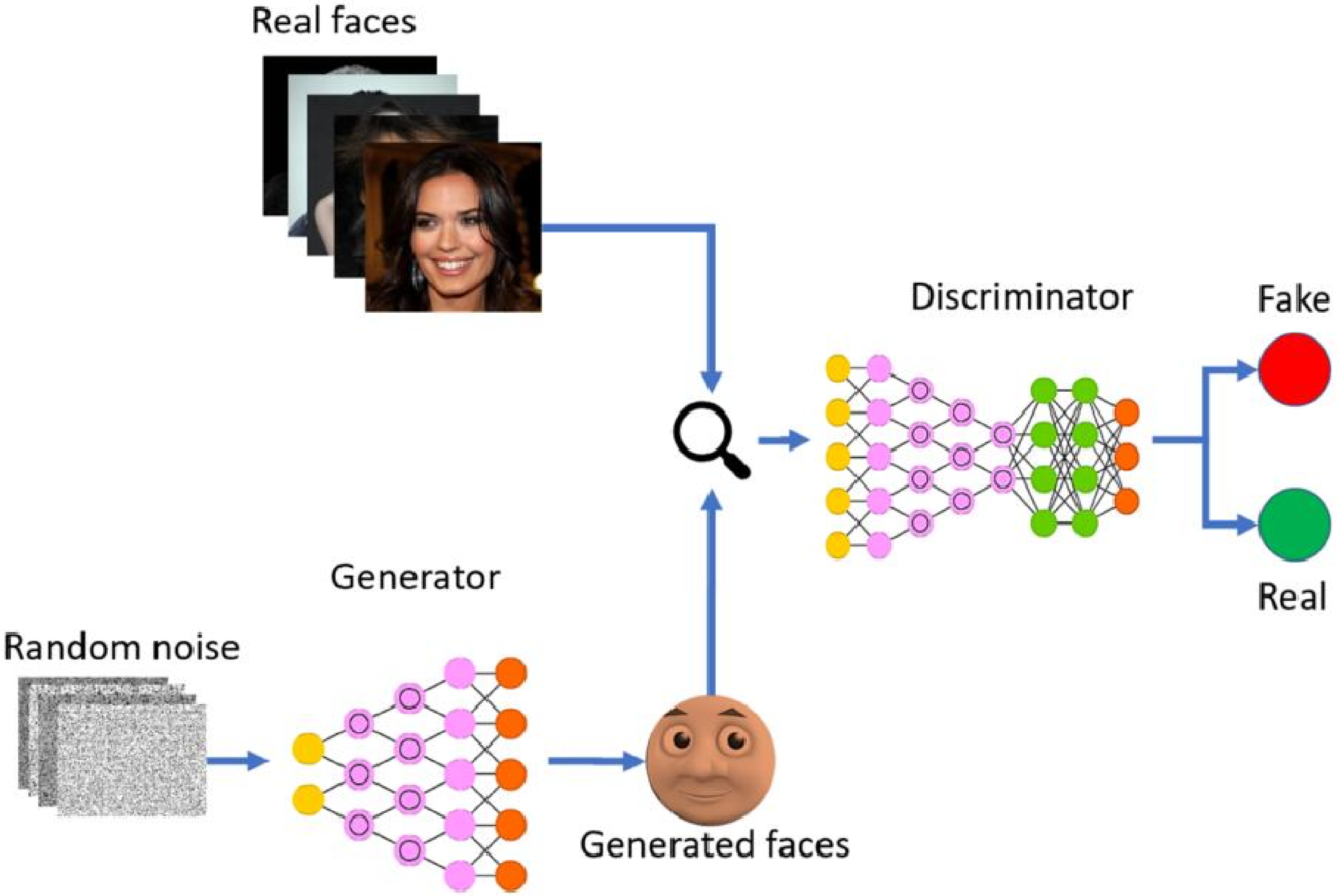
- Assume $D(x)$ represents probability of belonging to real class for a given sample, x
- Discriminator will try to increase $D(x)$ for real samples and decrease $D(x)$ for fake/generated samples
- Generator will try to increase $D(x)$ for generated samples

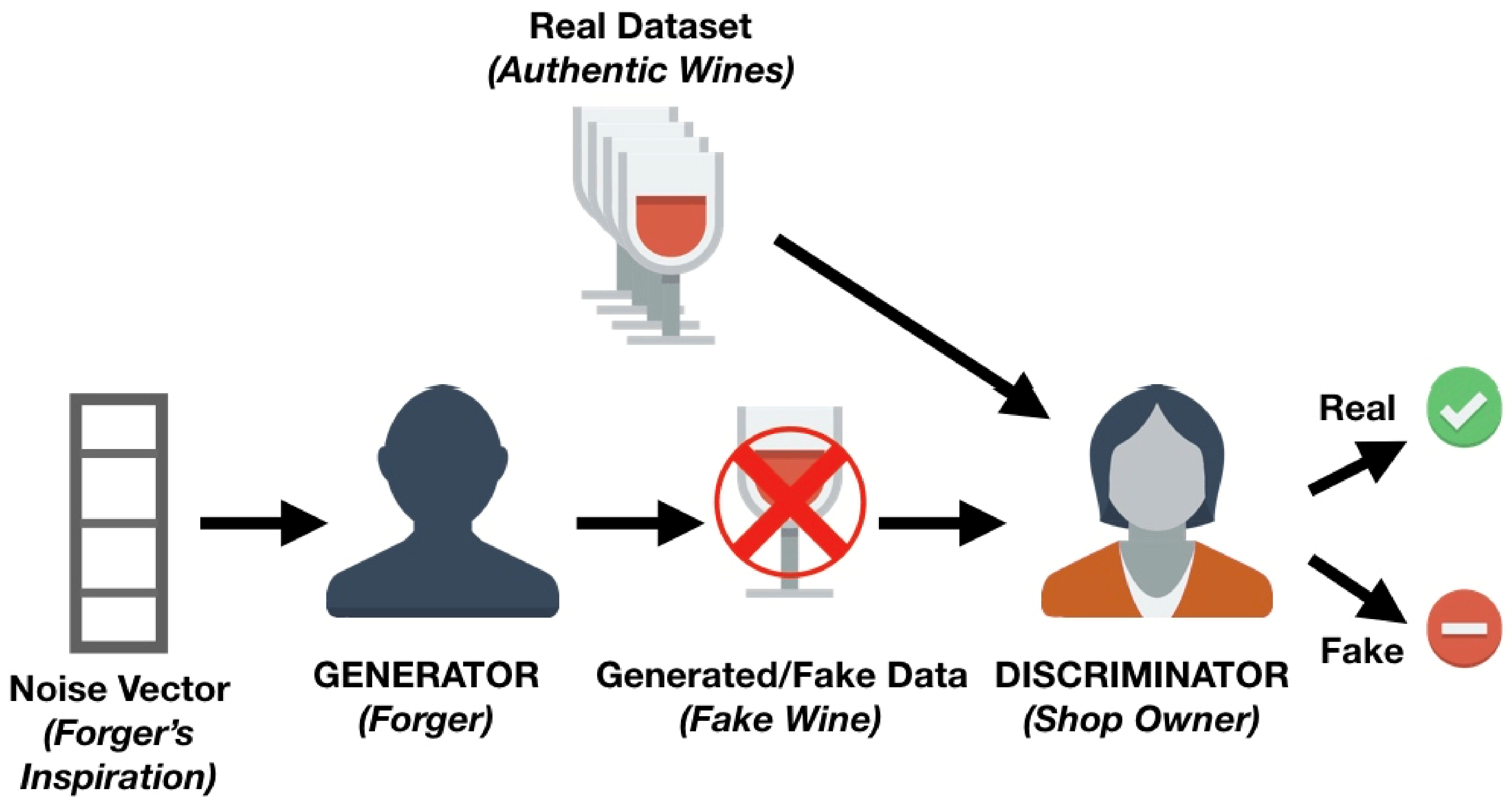


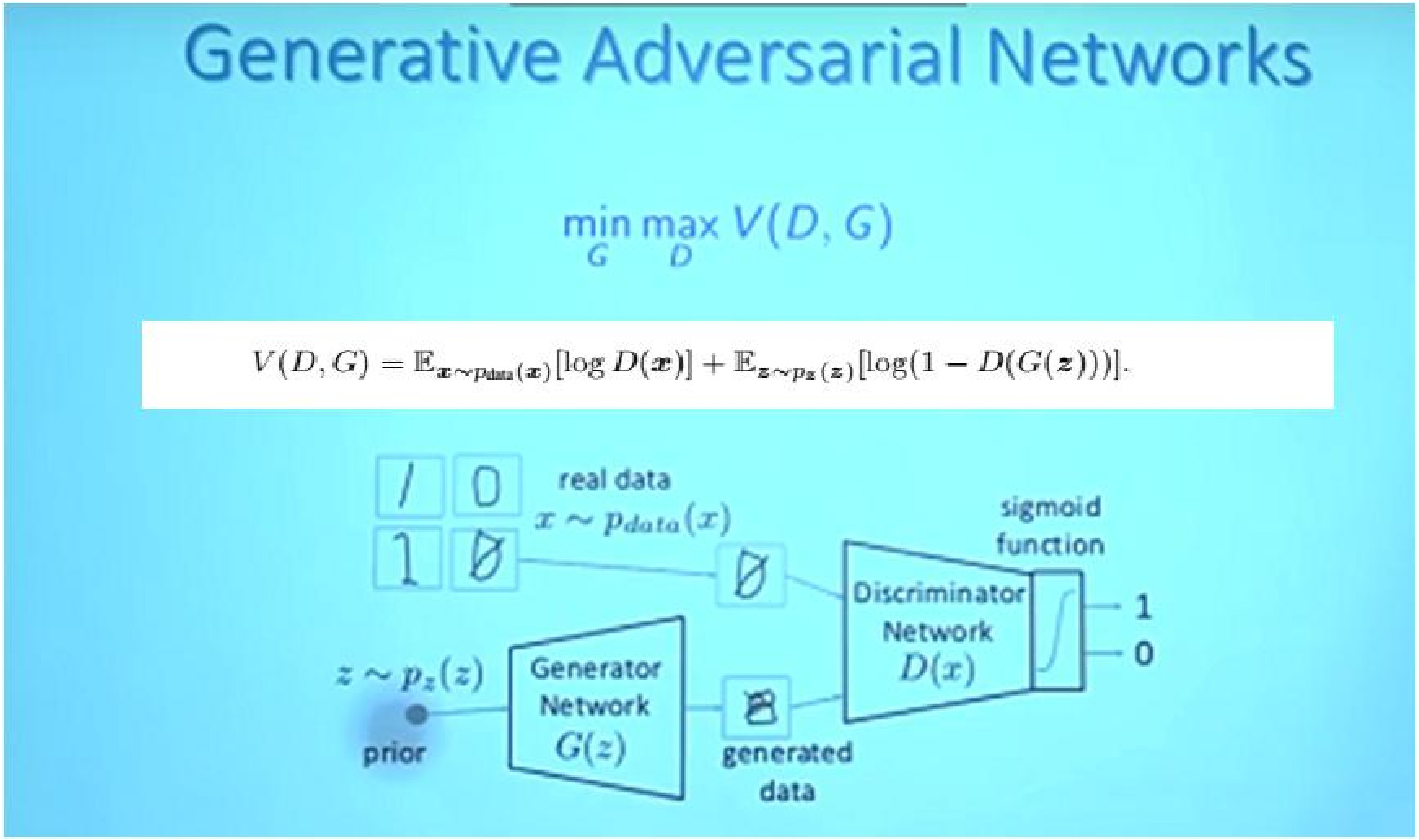
Generative adversarial networks (conceptual)



https://www.youtube.com/watch?v=7G4_Y5rsvi8


Real faces
Discriminator
Fake
Real
Generator
Random noise
Generated faces

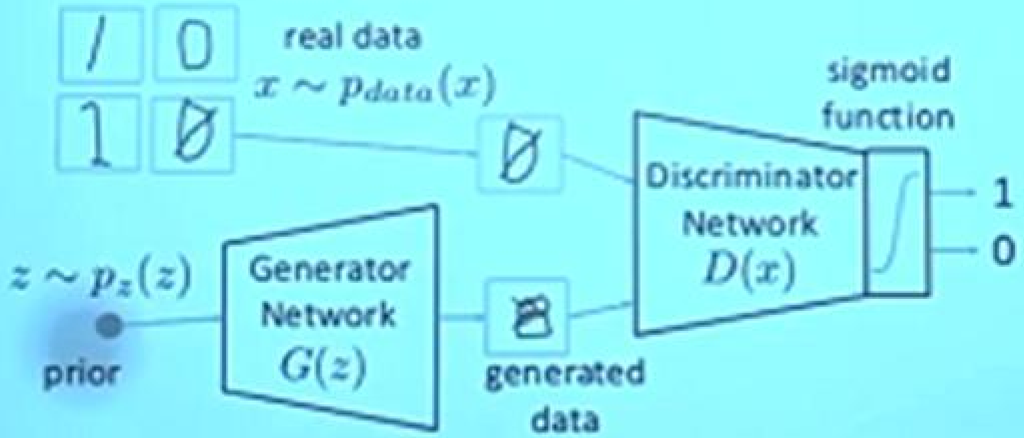

Real Dataset
(Authentic Wines)
Noise Vector
(Forger's Inspiration)
GENERATOR
(Forger)
Generated/Fake Data
(Fake Wine)
DISCRIMINATOR
(Shop Owner)
Real
Fake



Generative Adversarial Networks

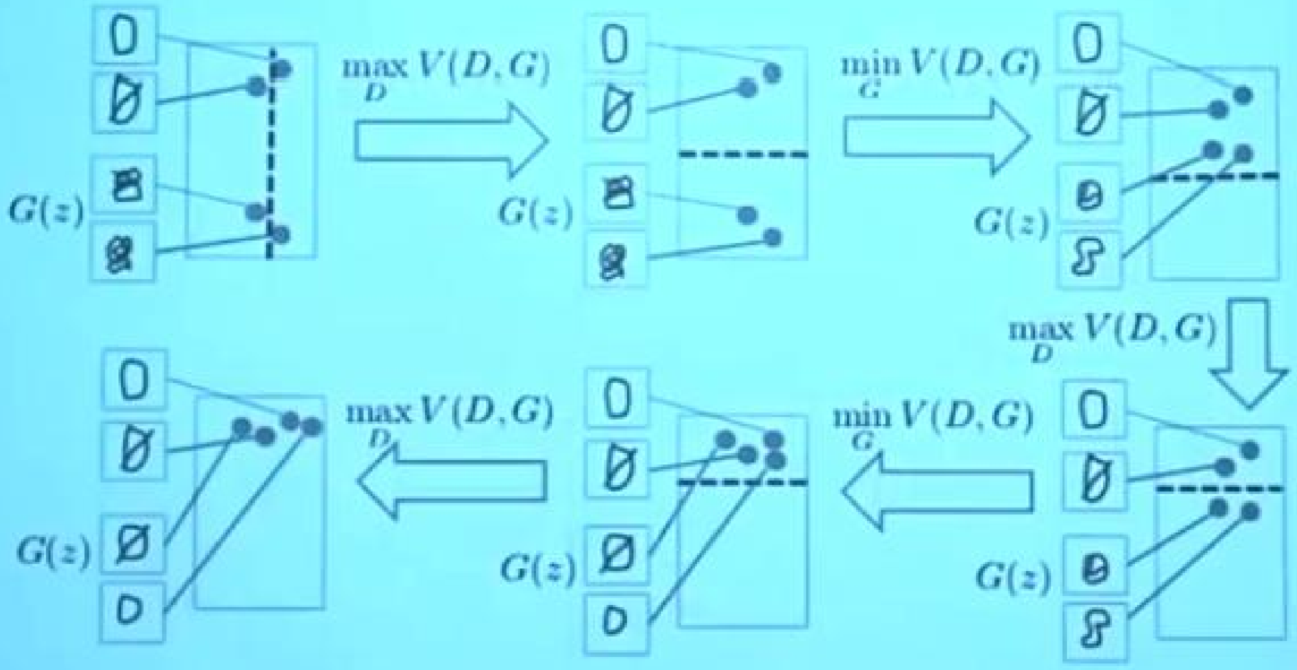
$$\min_G \max_D V(D, G)$$

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

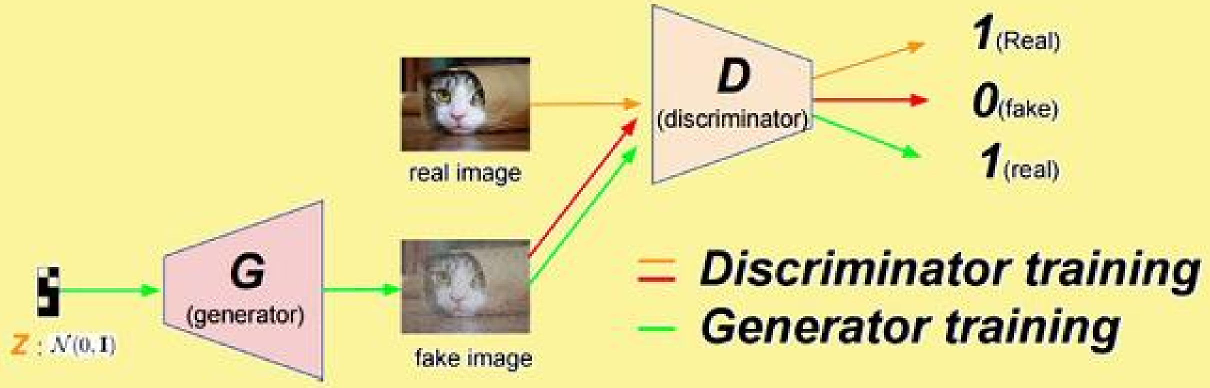


Training
Generative Adversarial Networks

$$\min_G \max_D V(D, G)$$



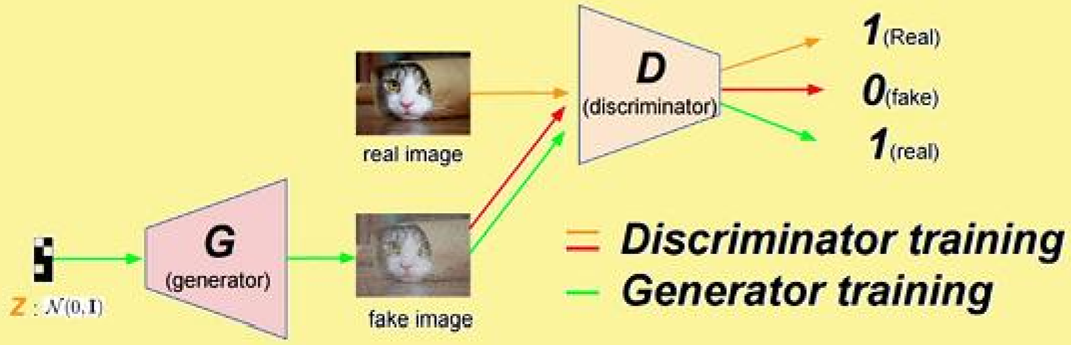
GAN Overview



<u>Training the Discriminator</u>

$$\max_D V(D,G) = \underbrace{E_{x\sim p_{data}(x)} \log D(x)}_{\text{Maximize probability for real}} + \underbrace{E_{z\sim p_z(z)}[\log\{1 - D\big(G(z)\big)\}]}_{\text{Minimize probability for generated}}$$

GAN Overview



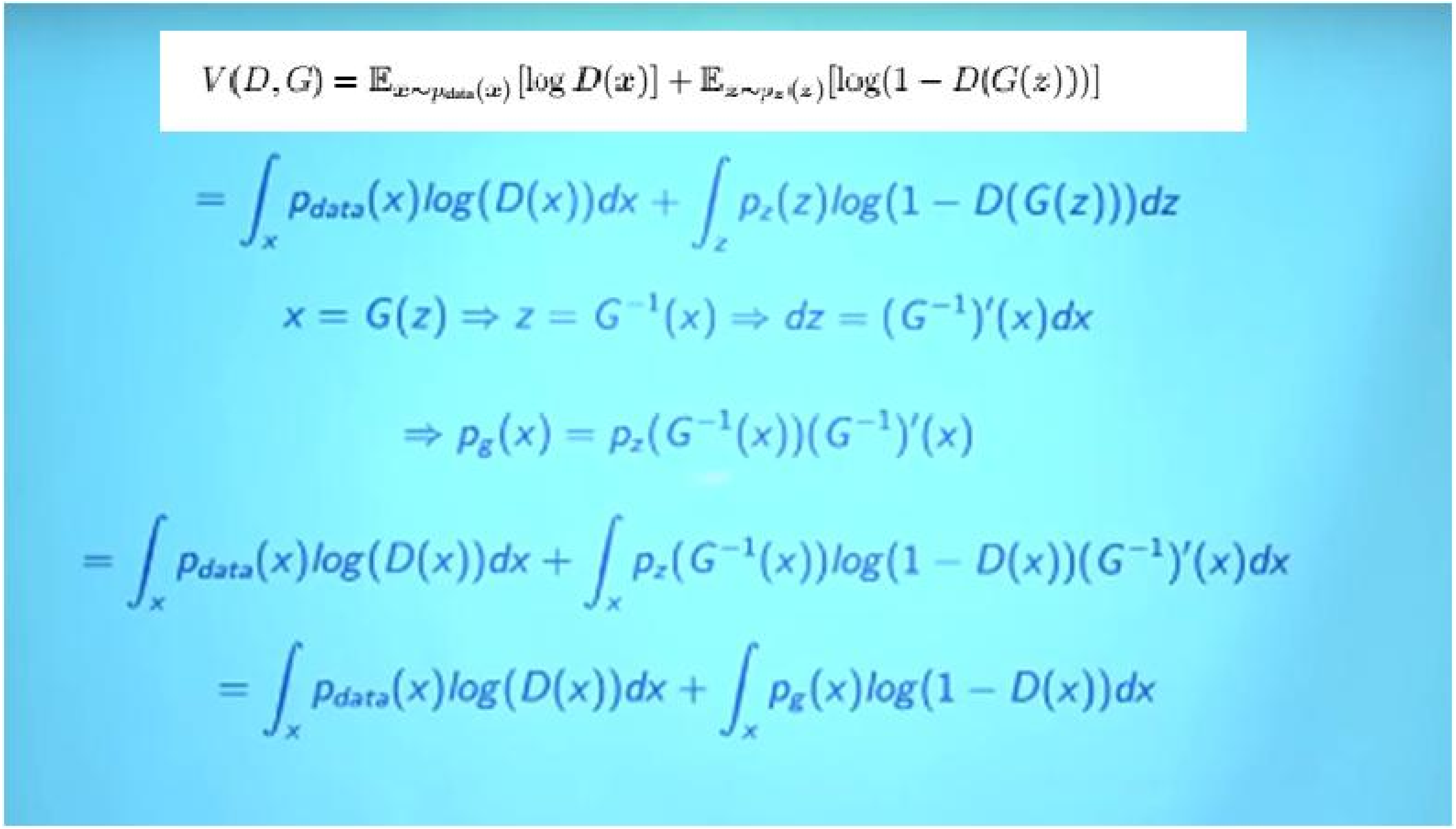
Training the Generator

$$\min_G V(D,G) = \quad E_{z\sim p_z(z)}[\log\{1 - D(G(z))\}]$$

$$\equiv \underbrace{\max_G \quad E_{z\sim p_z(z)}[\log D(G(z))]}_{\text{Maximize probability for generated}}$$

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
        • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$
    end for
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:
        $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$
end for
```



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

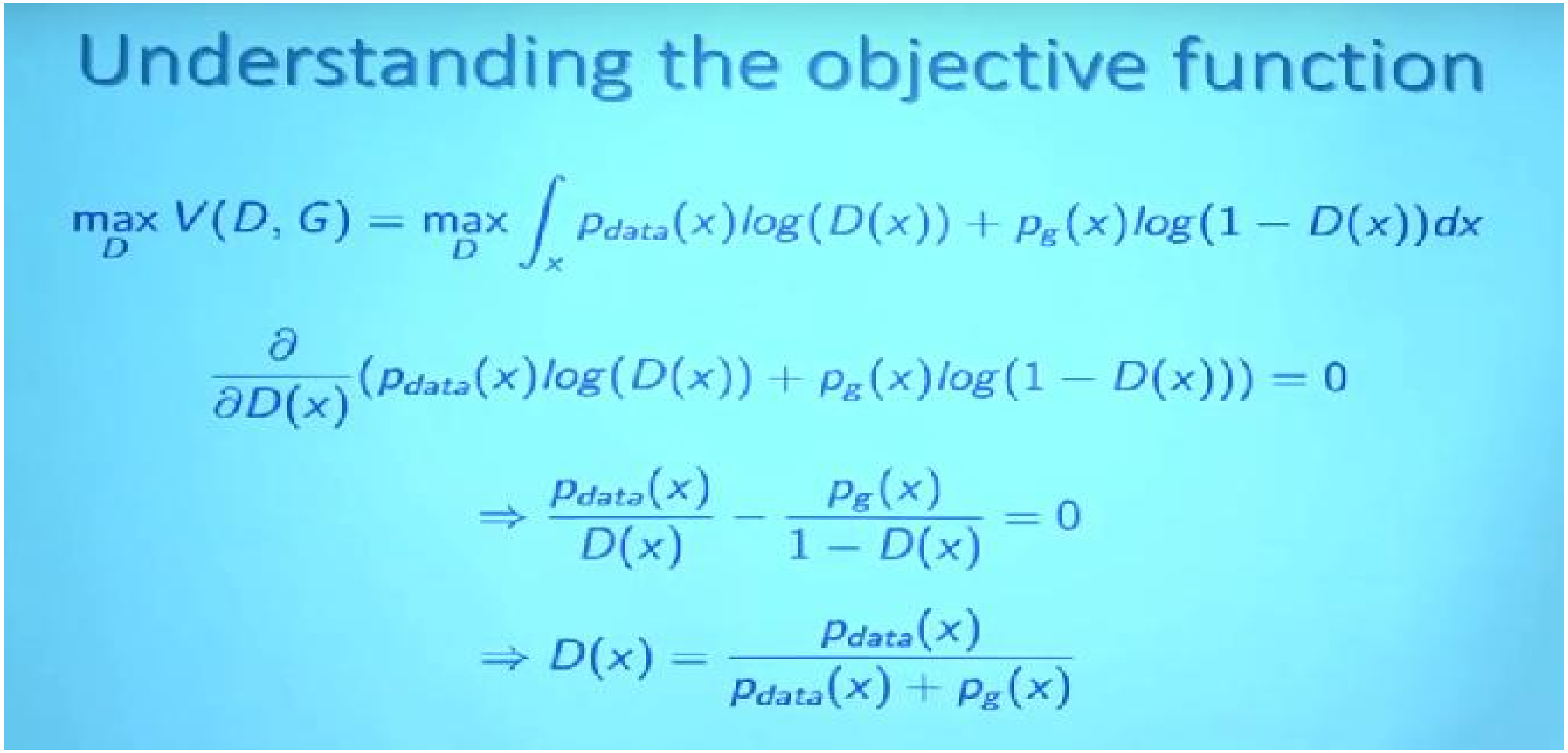
$$= \int_x p_{data}(x) log(D(x))dx + \int_z p_z(z) log(1 - D(G(z)))dz$$

$$x = G(z) \Rightarrow z = G^{-1}(x) \Rightarrow dz = (G^{-1})'(x)dx$$

$$\Rightarrow p_g(x) = p_z(G^{-1}(x))(G^{-1})'(x)$$

$$= \int_x p_{data}(x) log(D(x))dx + \int_x p_z(G^{-1}(x)) log(1 - D(x))(G^{-1})'(x)dx$$

$$= \int_x p_{data}(x) log(D(x))dx + \int_x p_g(x) log(1 - D(x))dx$$



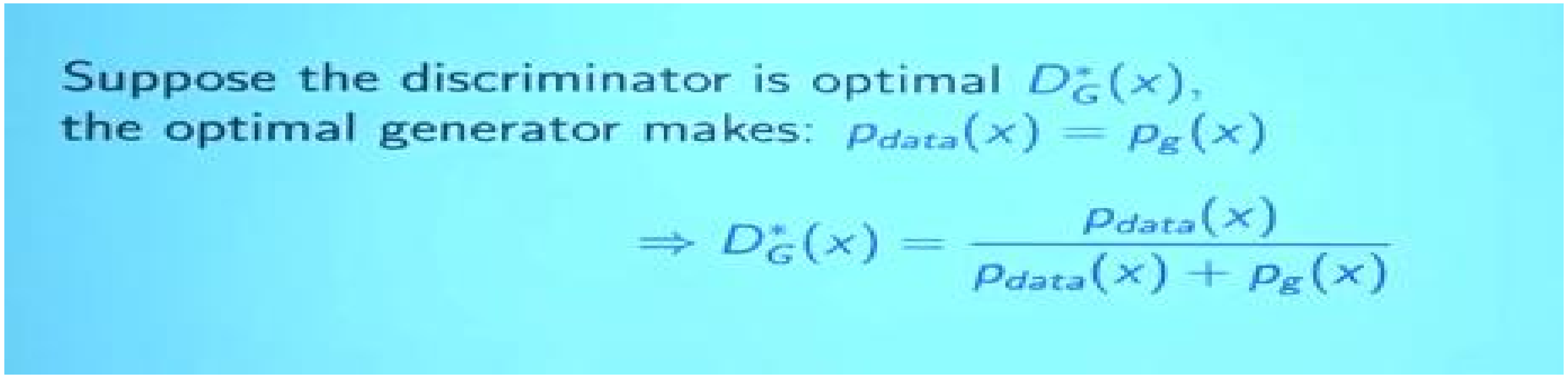
Understanding the objective function

$$\max_D V(D, G) = \max_D \int_x p_{data}(x) log(D(x)) + p_g(x) log(1 - D(x)) dx$$

$$\frac{\partial}{\partial D(x)}(p_{data}(x) log(D(x)) + p_g(x) log(1 - D(x))) = 0$$

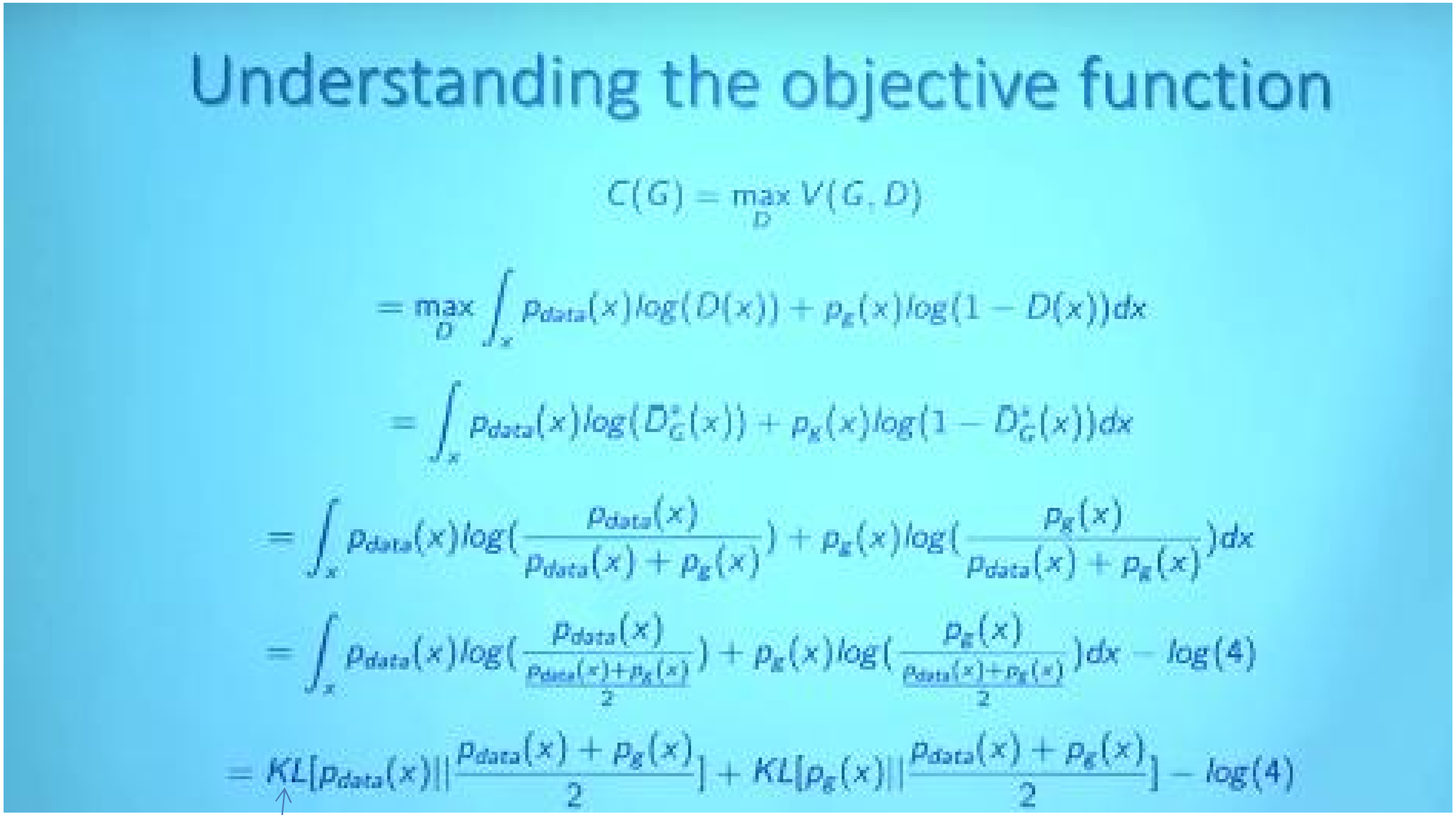
$$\Rightarrow \frac{p_{data}(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} = 0$$

$$\Rightarrow D(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$



Suppose the discriminator is optimal $D^*_G(x)$,
the optimal generator makes: $p_{data}(x) = p_g(x)$

$$\Rightarrow D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$



Understanding the objective function

$$C(G) = \max_D V(G, D)$$

$$= \max_D \int_x p_{data}(x) log(D(x)) + p_g(x) log(1 - D(x)) dx$$

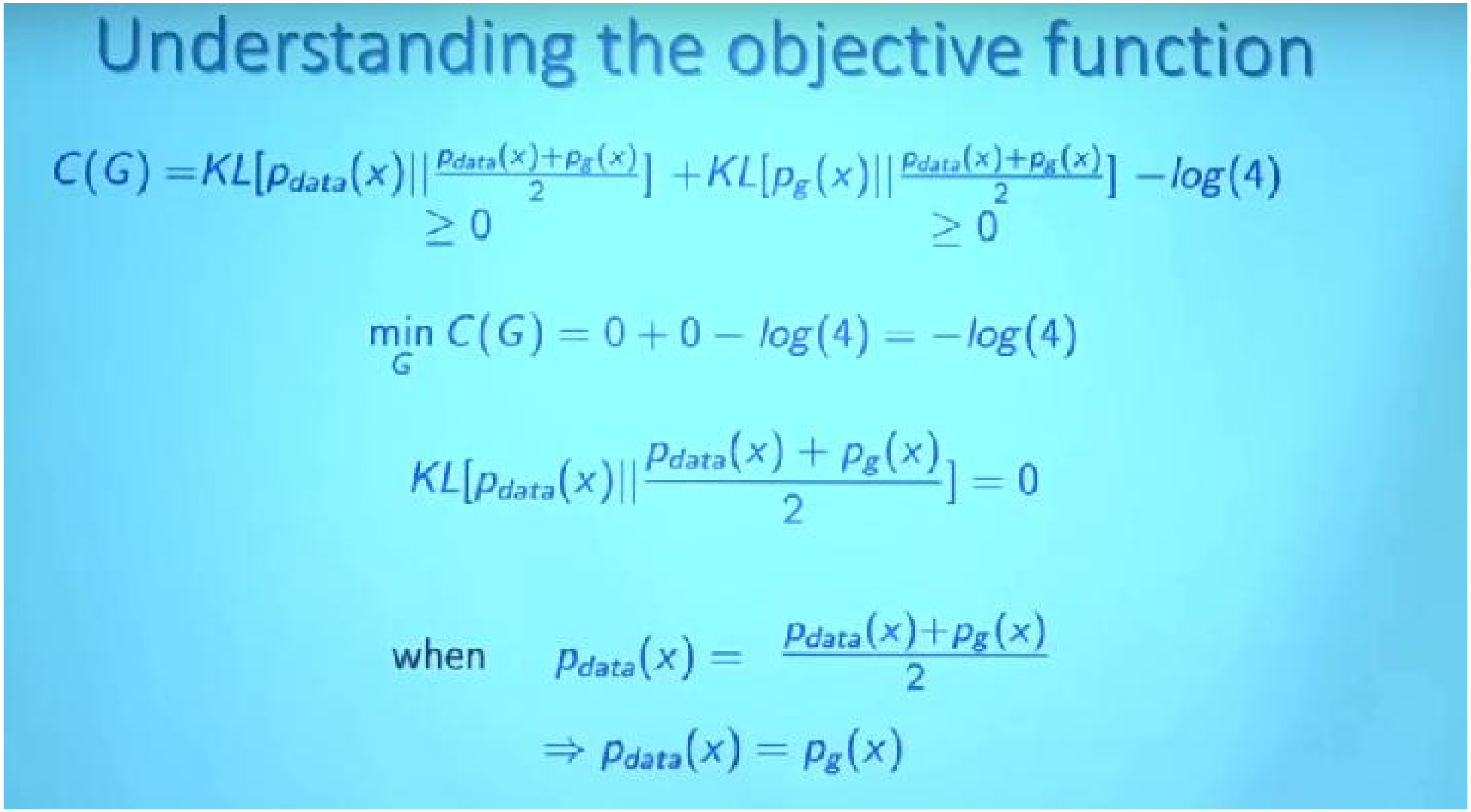
$$= \int_x p_{data}(x) log(D_G^*(x)) + p_g(x) log(1 - D_G^*(x)) dx$$

$$= \int_x p_{data}(x) log(\frac{p_{data}(x)}{p_{data}(x) + p_g(x)}) + p_g(x) log(\frac{p_g(x)}{p_{data}(x) + p_g(x)}) dx$$

$$= \int_x p_{data}(x) log(\frac{p_{data}(x)}{\frac{p_{data}(x)+p_g(x)}{2}}) + p_g(x) log(\frac{p_g(x)}{\frac{p_{data}(x)+p_g(x)}{2}}) dx - log(4)$$

$$= KL[p_{data}(x) || \frac{p_{data}(x) + p_g(x)}{2}] + KL[p_g(x) || \frac{p_{data}(x) + p_g(x)}{2}] - log(4)$$

KL divergence : Measure of similarity between two distribution function



Understanding the objective function

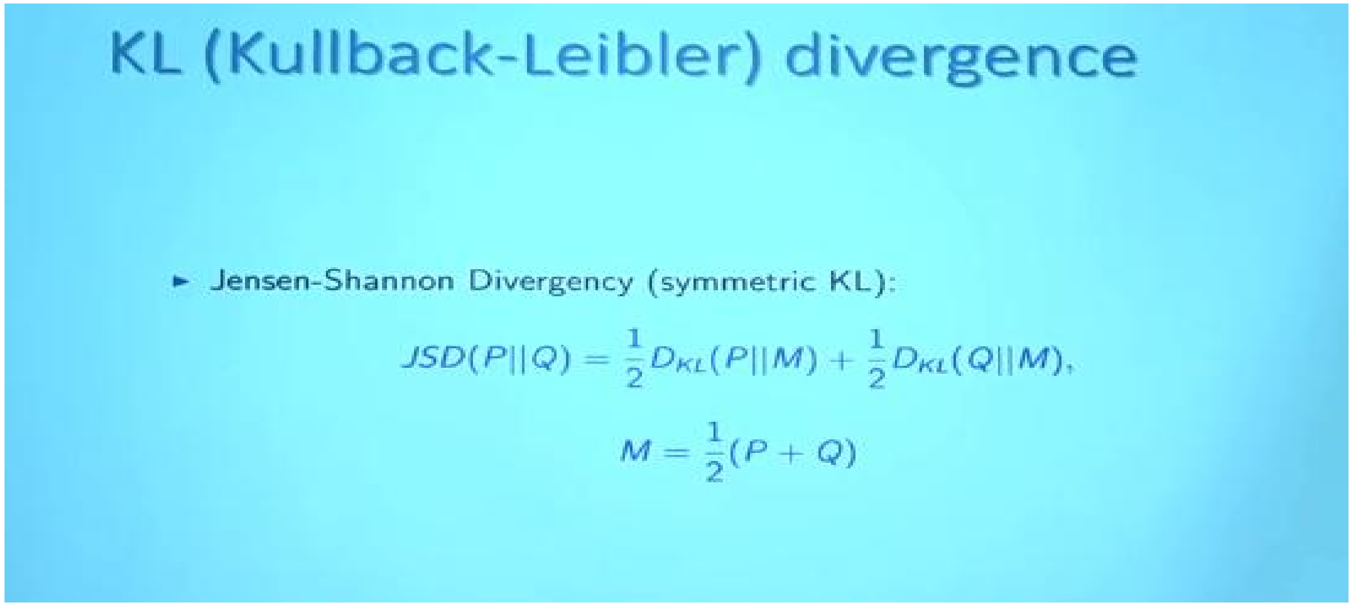
$$C(G) = \underbrace{KL[p_{data}(x) || \tfrac{p_{data}(x)+p_g(x)}{2}]}_{\geq 0} + \underbrace{KL[p_g(x) || \tfrac{p_{data}(x)+p_g(x)}{2}]}_{\geq 0} - log(4)$$

$$\min_G C(G) = 0 + 0 - log(4) = -log(4)$$

$$KL[p_{data}(x) || \frac{p_{data}(x) + p_g(x)}{2}] = 0$$

when $\quad p_{data}(x) = \dfrac{p_{data}(x)+p_g(x)}{2}$

$$\Rightarrow p_{data}(x) = p_g(x)$$

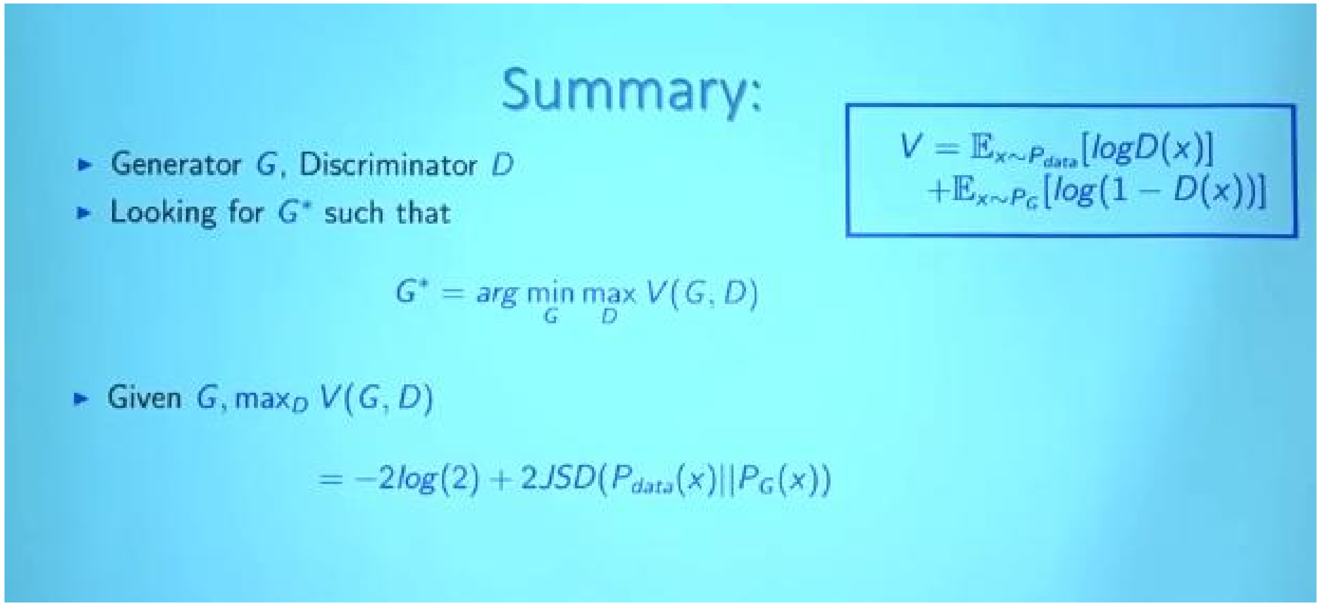


KL (Kullback-Leibler) divergence

- Jensen-Shannon Divergency (symmetric KL):

$$JSD(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M),$$

$$M = \frac{1}{2}(P + Q)$$



Summary:

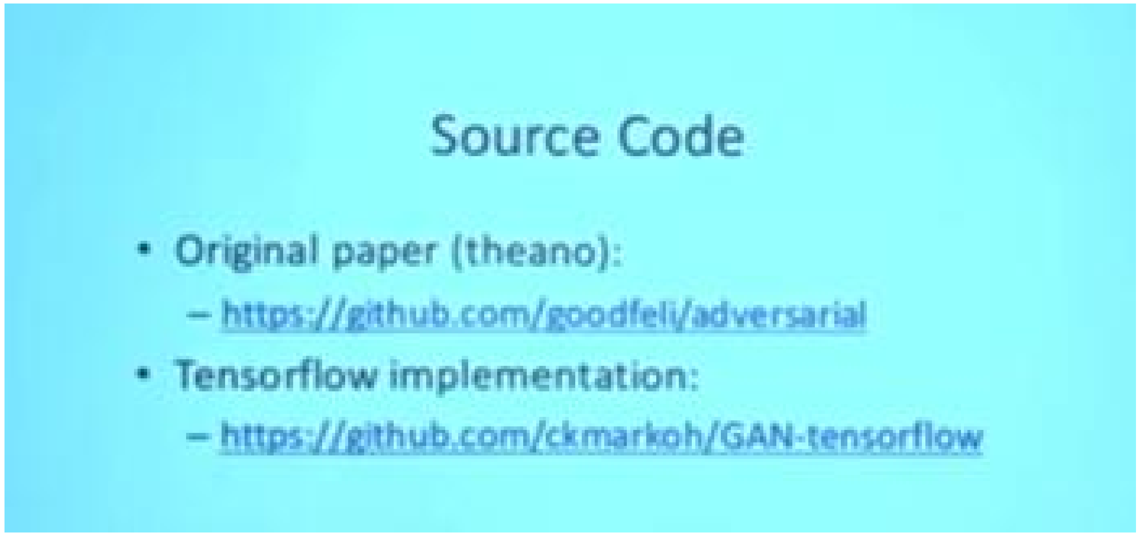
$$\boxed{\begin{aligned} V = {} & \mathbb{E}_{x\sim P_{data}}[logD(x)] \\ & +\mathbb{E}_{x\sim P_G}[log(1 - D(x))] \end{aligned}}$$

- Generator G, Discriminator D
- Looking for G^* such that

$$G^* = arg \min_G \max_D V(G, D)$$

- Given $G, \max_D V(G, D)$

$$= -2log(2) + 2JSD(P_{data}(x)||P_G(x))$$

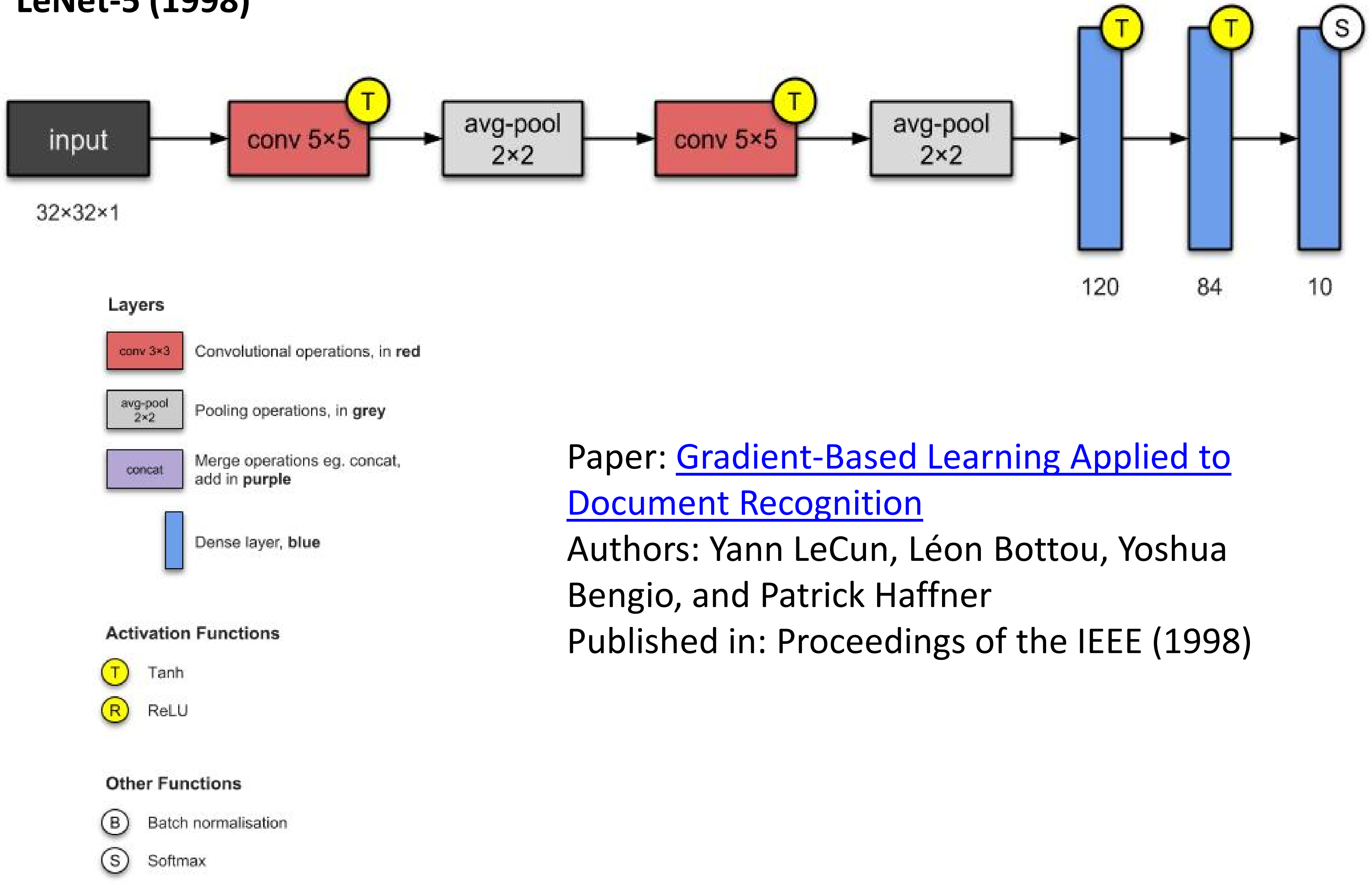


Source Code

- Original paper (theano):
  - https://github.com/goodfeli/adversarial
- Tensorflow implementation:
  - https://github.com/ckmarkoh/GAN-tensorflow

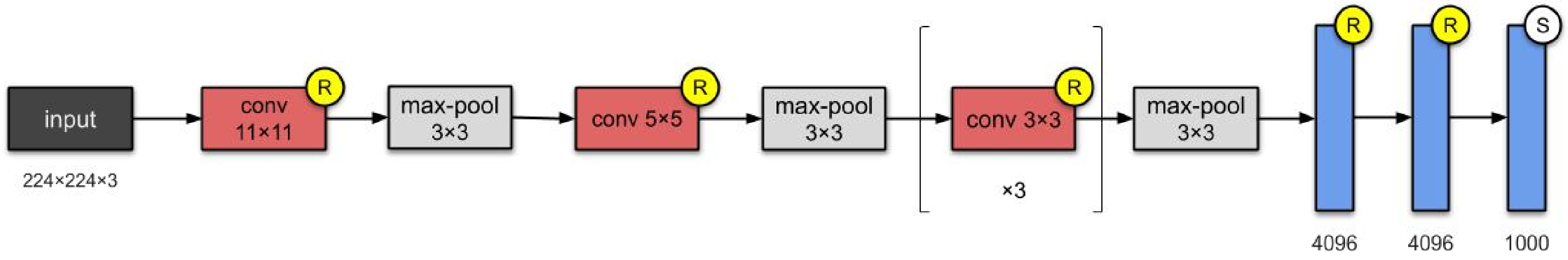
Convolutional Neural Network (CNN)

LeNet-5 (1998)



Paper: Gradient-Based Learning Applied to Document Recognition
Authors: Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner
Published in: Proceedings of the IEEE (1998)

2. AlexNet (2012)



What's novel?

1. They were the first to implement Rectified Linear Units (ReLUs) as activation functions.

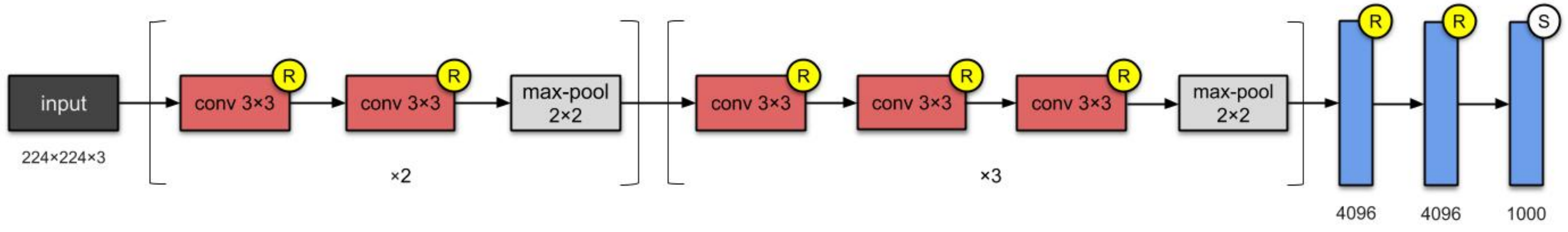
Paper: [ImageNet Classification with Deep Convolutional Neural Networks]

Authors: Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton. University of Toronto, Canada.

Published in: NeurIPS 2012

3. VGG-16 (2014)

Visual Geometry Group

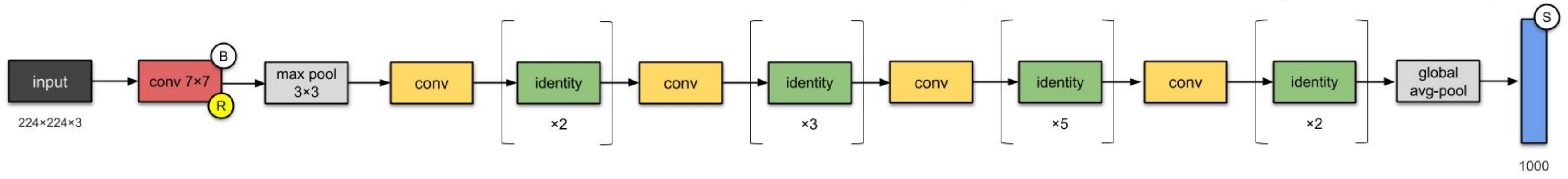


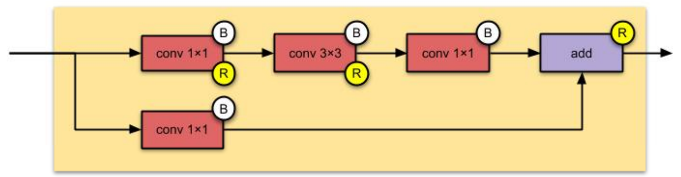
What's novel?
As mentioned in their abstract, the contribution from this paper is the designing of *deeper* networks (roughly twice as deep as AlexNet).

Paper: Very Deep Convolutional Networks for Large-Scale Image Recognition
Authors: Karen Simonyan, Andrew Zisserman.
University of Oxford, UK.
arXiv preprint, 2014

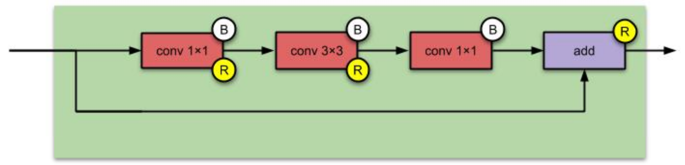
ResNet-50 (2015)

Skip architecture as the name suggests **skips** some layer in the neural network and feeds the output of one layer as the input to the next layer as well as some other layer (instead of only the next layer).





Conv block



Identity block

What's novel?

Popularised skip connections (they weren't the first to use skip connections).

Designing even deeper CNNs (up to 152 layers) without compromising model's generalisation power

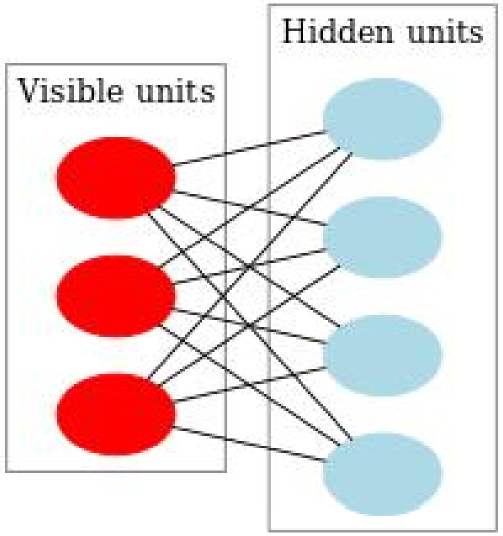
Among the first to use batch normalisation.

Paper: Deep Residual Learning for Image Recognition

Authors: Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Microsoft

Published in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

RBM

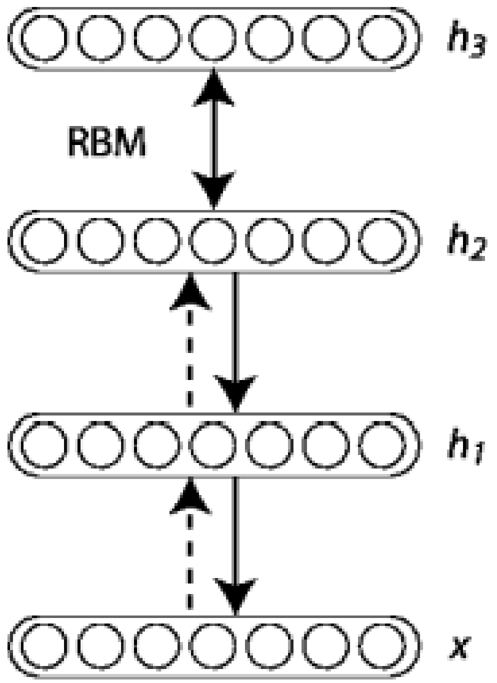


A restricted Boltzmann machine (RBM) with fully connected visible and hidden units. Note there are no hidden-hidden or visible-visible connections.

An RBM is an undirected, generative energy-based model with a "visible" input layer and a hidden layer and connections between but not within layers. This composition leads to a fast, layer-by-layer unsupervised training procedure, where contrastive divergence **(**a recipe for training undirected graphical models (a class of probabilistic models used in machine learning)) is applied to each sub-network in turn, starting from the "lowest" pair of layers (the lowest visible layer is a training set).

Contrastive divergence is a recipe for training undirected graphical models (a class of probabilistic models used in machine learning). It relies on an approximation of the gradient (a good direction of change for the parameters) of the log-likelihood (the basic criterion that most probabilistic learning algorithms try to optimize) based on a short Markov chain (a way to sample from probabilistic models) started at the last example seen.

Deep Belief Networks



http://deeplearning.net/tutorial/DBN.html