

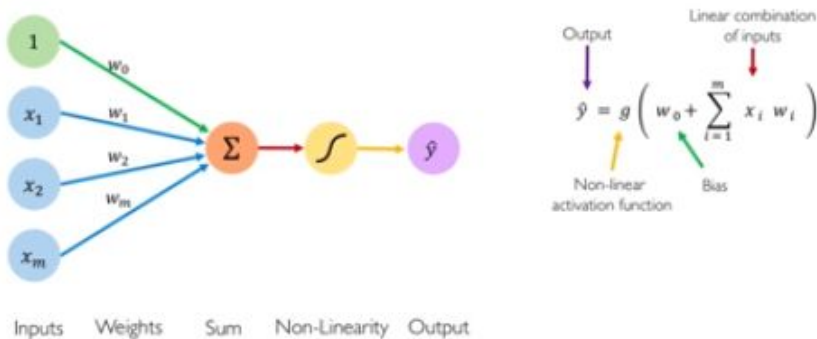
## INTRO TO DEEP LEARNING

Learning from the things are working later ~ learning features from underlying features.

Big Data -> Hardware -> Software

Perceptron :

### The Perceptron: Forward Propagation



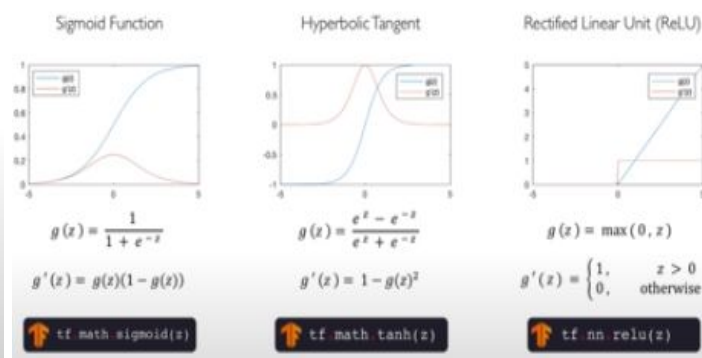
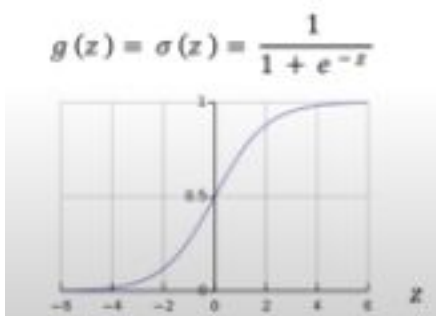
Activation Function :

Takes a real number in x axis and converts it into a number in 0 and 1.

$$\hat{y} = g(w_0 + X^T W)$$

### Common Activation Functions

- Example: sigmoid function



Purpose of AF is - to introduce non-linearities in functions.

Hidden layers , dense layer

Quantifying Loss : measures the cost incurred from incorrect predictions.

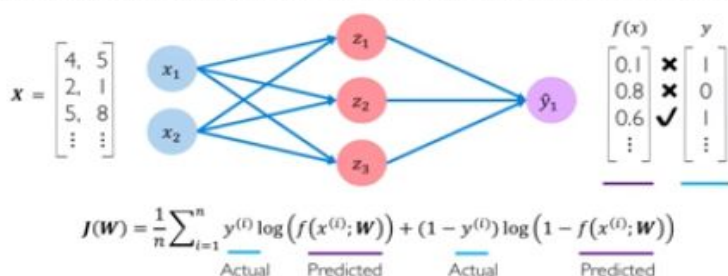
$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss : total loss over the entire dataset.

$$J(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

## Binary Cross Entropy Loss

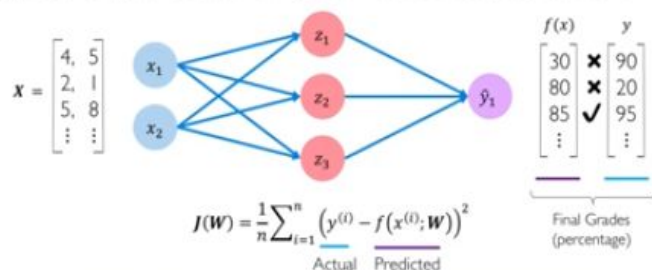
Cross entropy loss can be used with models that output a probability between 0 and 1



```
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, predicted))
```

## Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean(tf.square(tf.subtract(y, predicted)))
```

Loss Optimisation : finding the network weights that achieve the lowest loss through maybe gradient descent.

### Algorithm

1. Initialize weights randomly  $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient,  $\frac{\partial J(W)}{\partial W}$
4. Update weights,  $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:  # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

Backpropagation : chain rule ~

Optimization through gradient descent

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

How can we set the learning rate?

Adaptive Learning Rates ~

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

## Gradient Descent Algorithms

Algorithm	TF Implementation	Refe
• SGD	 <code>tf.keras.optimizers.SGD</code>	Kiefer & Wollfowitz, "Sto Maximum of a Regressi
• Adam	 <code>tf.keras.optimizers.Adam</code>	Kingma et al. "Adam: A l Optimization," 2014.
• Adadelat	 <code>tf.keras.optimizers.Adadelta</code>	Zeller et al. "ADADELT Method," 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	Duchi et al. "Adaptive S Learning and Stochastic
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	

```

import tensorflow as tf
model = tf.keras.Sequential([...])
# pick your favorite optimizer
optimizer = tf.keras.optimizers.SGD()
while True: # loop forever
    # forward pass through the network
    prediction = model(x)
    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)
    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Can replace TensorFlow

Stochastic gradient descent ~ means using a single point.

Using a single point for gradient descent is very noisy and using all the data points is very computationally expensive, therefore the solution is ~

Using mini batches of points ~ and true is calculated by taking the average, therefore smoother convergence and allows greater learning rates, enables parallelism therefore faster.

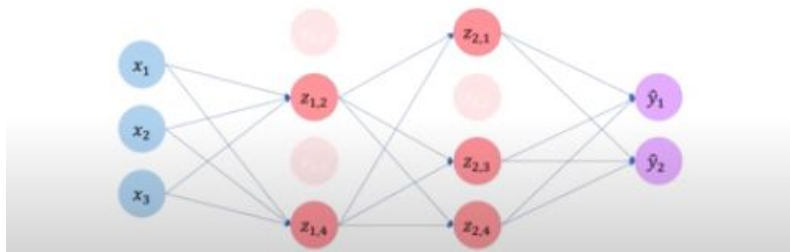
Overfitting exists so to deal with it ~

Regularisation ~ technique that constraints our optimisation problem to discourage complex models ~ improve generalisation.

## Regularization I: Dropout

- During training, randomly set some activations to 0
  - Typically 'drop' 50% of activations in layer
  - Forces network to not rely on any 1 node

`tf.keras.layers.Dropout(p=0.5)`



## Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



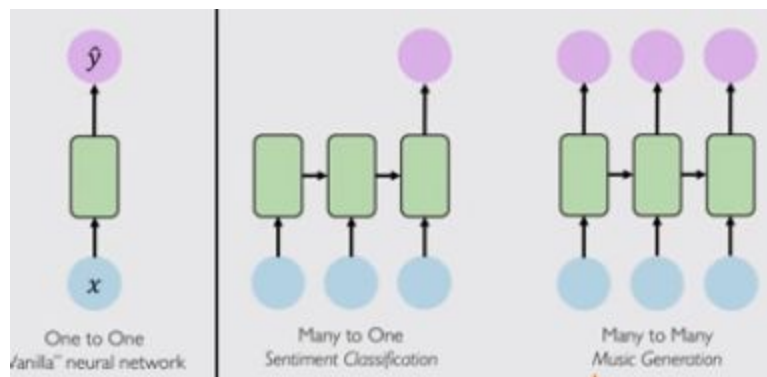
## RECURRENT NEURAL NETWORKS

Sequential Modelling : predict the next thing that will happen after a previous states are given

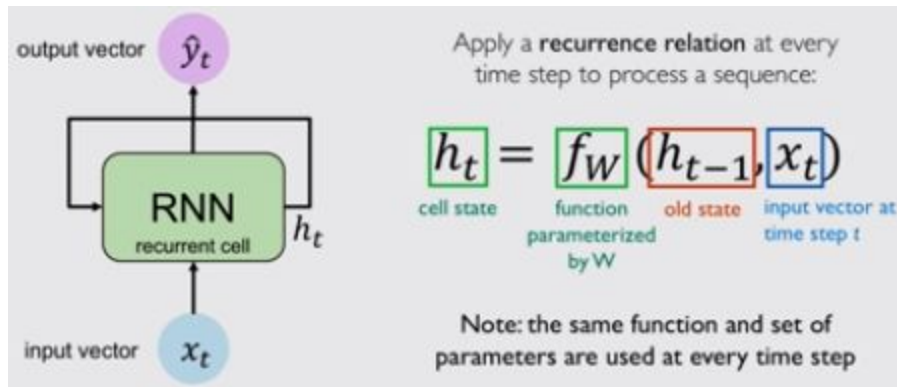
If word prediction then if using BagOfWords then the order is not reserved therefore Separate parameters should be used to encode the sentence.

In standard Feed Forward -> it goes one to one passing of data

RNN for Sequence Modeling :



RNNs have a loop in them that enables them to maintain an internal state , not like a vanilla NN where the final output is only present.



Output Vector

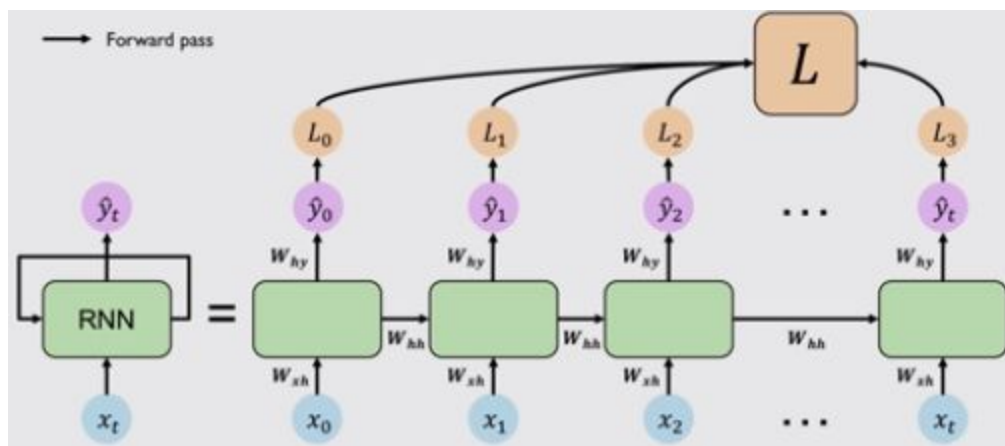
$$\hat{y}_t = W_{hy}^T h_t$$

Update Hidden State

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

Input Vector

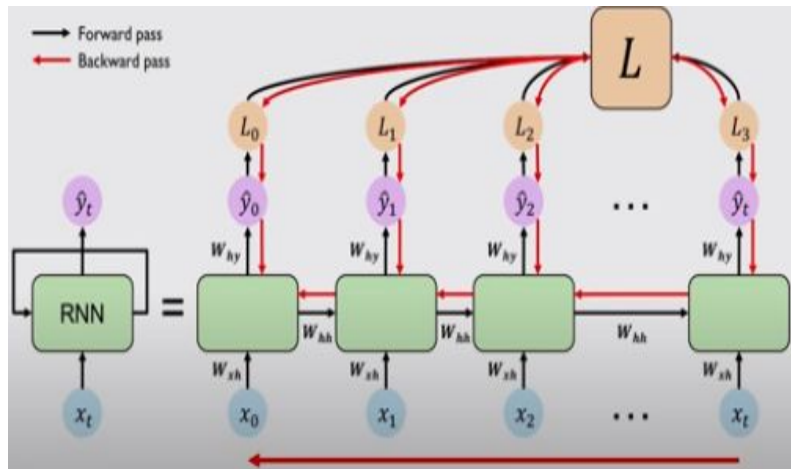
$$x_t$$



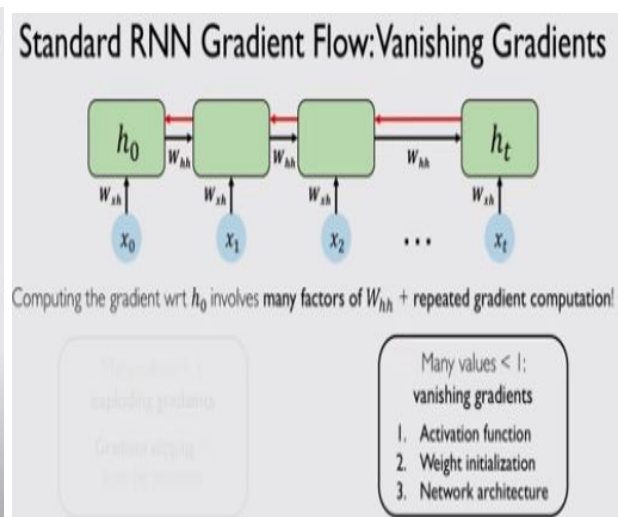
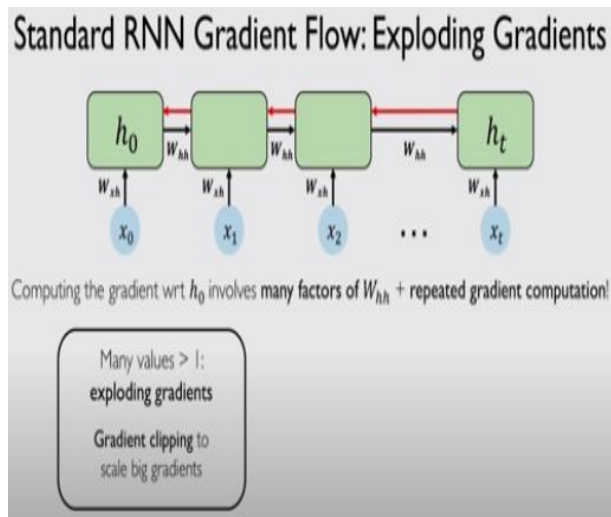
```
tf.keras.layers.SimpleRNN(rnn_units)
```

Backpropagation through time :

In RNNs at each time backprop is done also in total ~



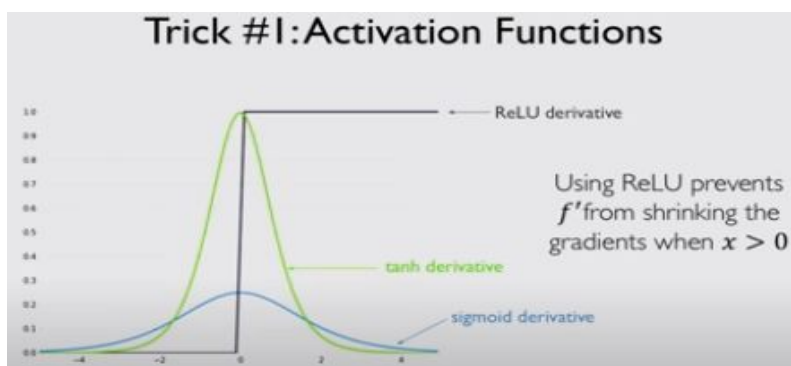
Standard RNN Gradient Flow can have 2 problems :



The problem of LongTermDependencies (vanishing gradient)~

Multiply many small numbers together -> errors which are in further back time steps keep having smaller and smaller gradients -> this biases es parameters to capture dependencies in models , thus standard RNNs becoming less capable.

To solve the above problem :





Both tanh and sigmoid have derivatives less than 1.  
Therefore use RELU but  $x > 0$  only then.

## Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

## Solution #3: Gated Cells

Idea: use a more **complex recurrent unit with gates** to control what information is passed through

**gated cell**

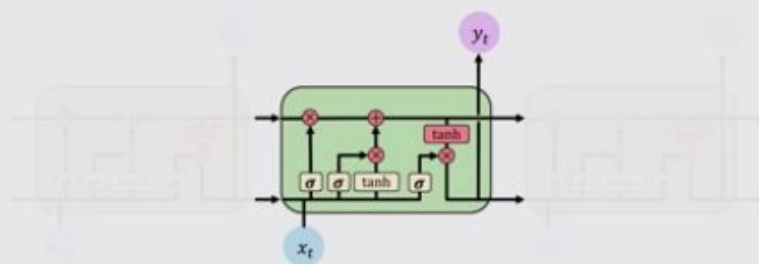
LSTM, GRU, etc.

**Long Short Term Memory (LSTMs)** networks rely on a gated cell to track information throughout many time steps.


The above is well suited for learning tasks.

Long Short Term Memory (LSTM) Networks :

LSTM modules contain **computational blocks** that **control information flow**

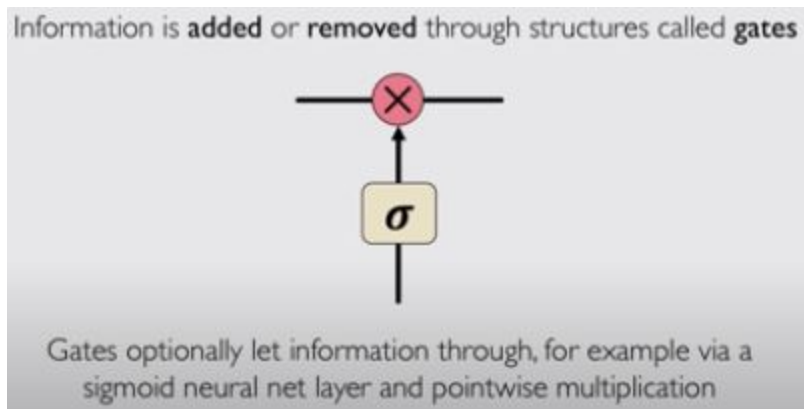


LSTM cells are able to track information throughout many timesteps

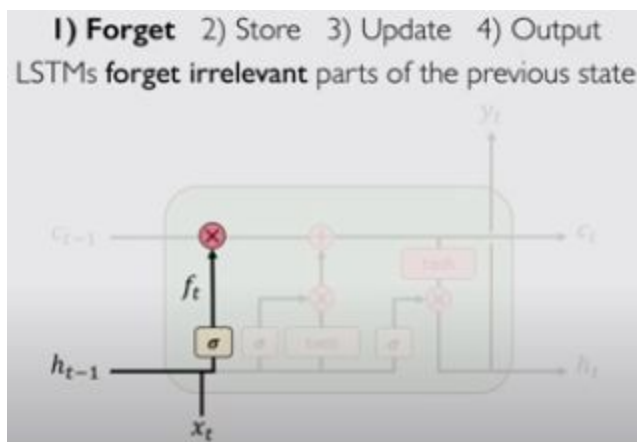
 `tf.keras.layers.LSTM(num_units)`



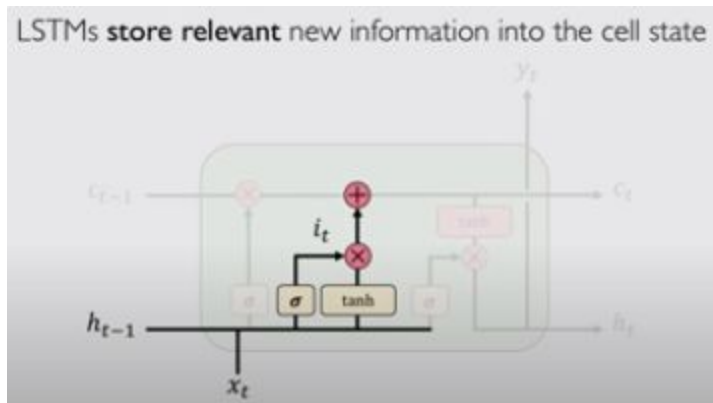
Gates are an important part in LSTMs



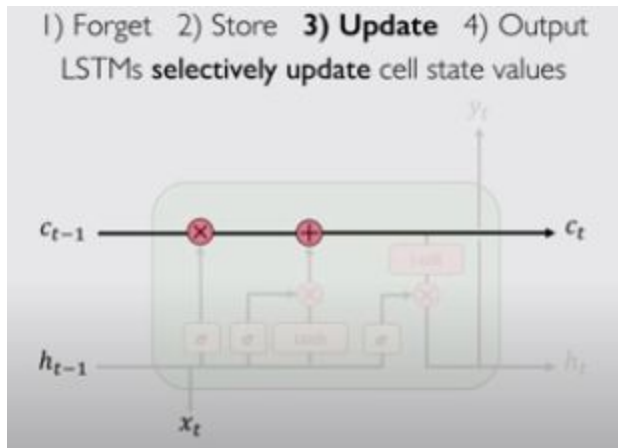
LSTMs work in 4 steps :



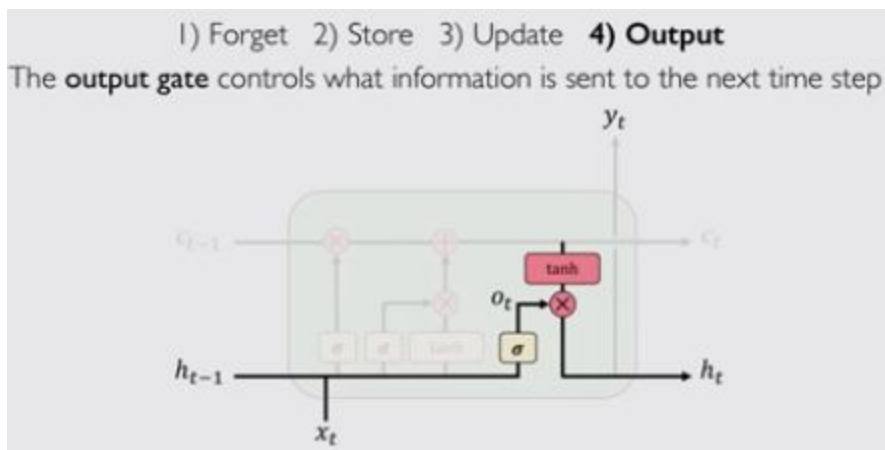
1.



2.



3.



4.

All these allow LSTMs to have an uninterrupted gradient flow.

RNN Applications :

Example :

Music Generation ~ at each level the next tune is predicted.

Sentiment Classification ~

Machine Translation ~ encoding bottleneck is a problem so attention mechanism is used ~ all states of time steps are accessed and trained upon individually.

Trajectory Prediction ~ Self Driving Car

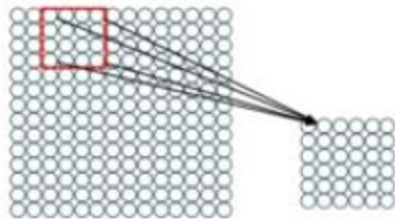
Environmental Modelling

## CONVOLUTION NEURAL NETWORKS

Spatial features need to be created in case of IMAGE data.

Using Spatial features :

### Feature Extraction with Convolution



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

This "patchy" operation is **convolution**

1) Apply a set of weights – a filter – to extract **local features**

2) Use **multiple filters** to extract different features

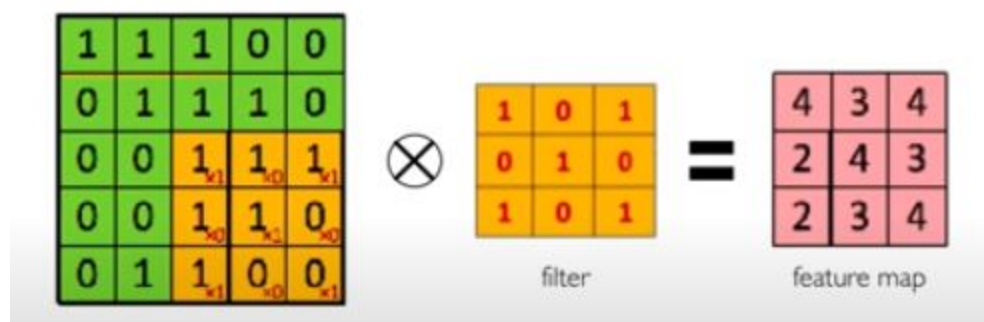
3) **Spatially share** parameters of each filter

Deformation should be handled.

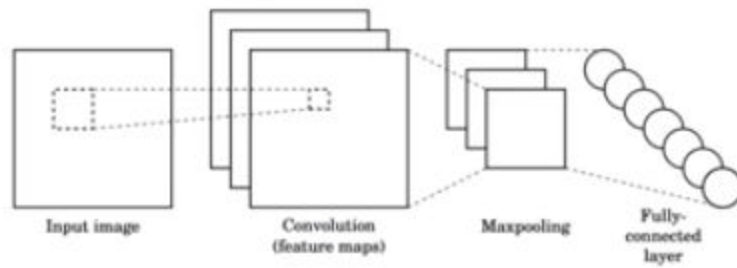
Convolution preserves the features , multiply the features of the patch and the same size patch its being compared to -> then if the whole thing is outcoming 1 then -> its an exact match.

### The Convolution Operation


We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs:



# CNNs for Classification



1. **Convolution:** Apply filters to generate feature maps.

 `tf.keras.layers.Conv2D`

2. **Non-linearity:** Often ReLU.

 `tf.keras.activations.*`

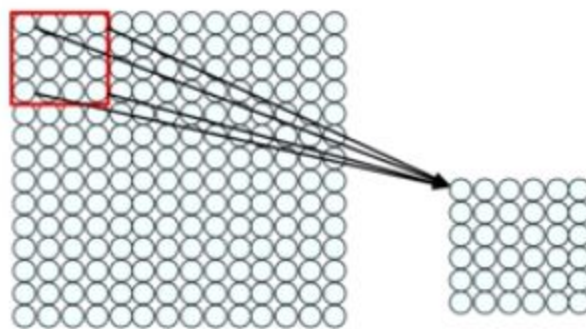
3. **Pooling:** Downsampling operation on each feature map.


 `tf.keras.layers.MaxPool2D`

Train model with image data.  
Learn weights of filters in convolutional layers.

Computation of class scores can be outputted as a dense layer representing the probability of each class.

## Convolutional Layers: Local Connectivity



 `tf.keras.layers.Conv2D`

**For a neuron in hidden layer:**

- Take inputs from patch
- Compute weighted sum
- Apply bias

4x4 filter: matrix of weights  $w_{ij}$

$$\sum_{i=1}^4 \sum_{j=1}^4 w_{ij} x_{i+p, j+q} + b$$

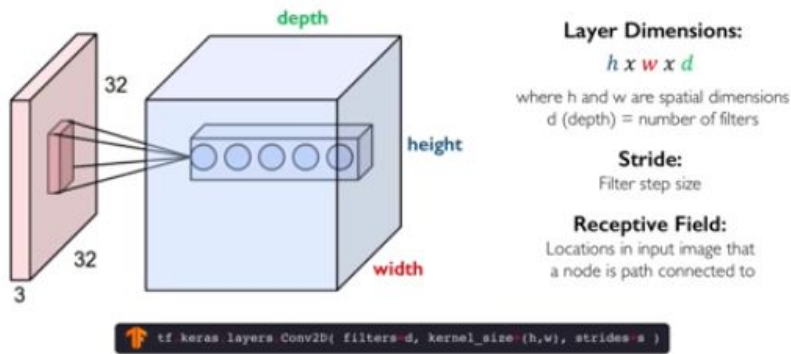
for neuron (p,q) in hidden layer

- 1) applying a window of weights
- 2) computing linear combinations
- 3) activating with non-linear function

1.

We have to define how many features to detect.

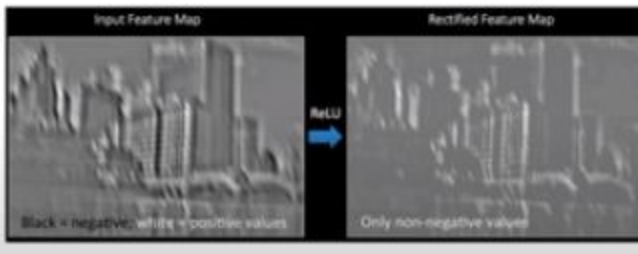
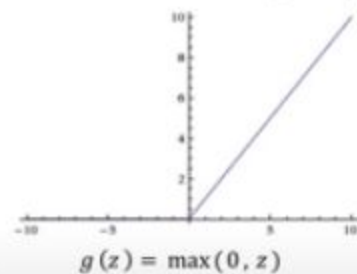
## CNNs: Spatial Arrangement of Output Volume



## Introducing Non-Linearity

- Apply after every convolution operation (i.e., after convolutional layers)
- ReLU: pixel-by-pixel operation that replaces all negative values by zero. **Non-linear operation!**

### Rectified Linear Unit (ReLU)



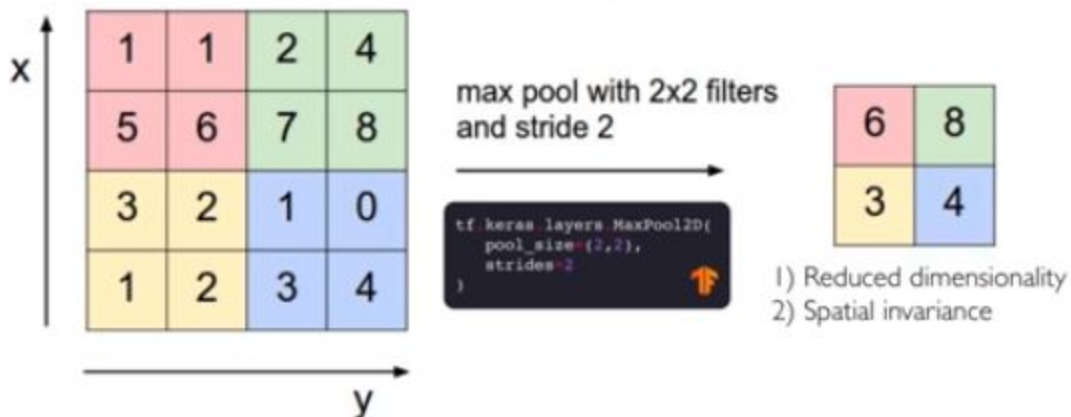
```
tf.keras.layers.ReLU
```

2.

ReLU takes input any real number and it shifts any number less than zero to zero and greater than zero it keeps the same. (min of everything is 0)

3. Downsampling using Pooling :

## Pooling

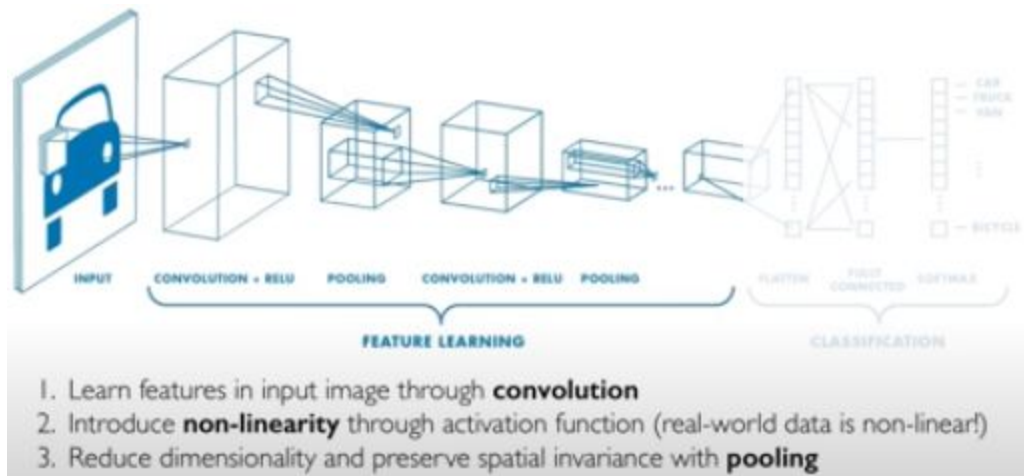


How else can we downsample and preserve spatial invariance?

There are other ways too, to downsample other than MAX-pooling.

In DEEP CNNs we can stack layers to bring out the features, low -> mid -> high

Part 1: Feature Learning -> extract and learn the features.



PART 2: Classification

## CNNs for Classification: Class Probabilities



- CONV and POOL layers output high-level features of input
- Fully connected layer uses these features for classifying input image
- Express output as **probability** of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

## Putting it all together

```
import tensorflow as tf

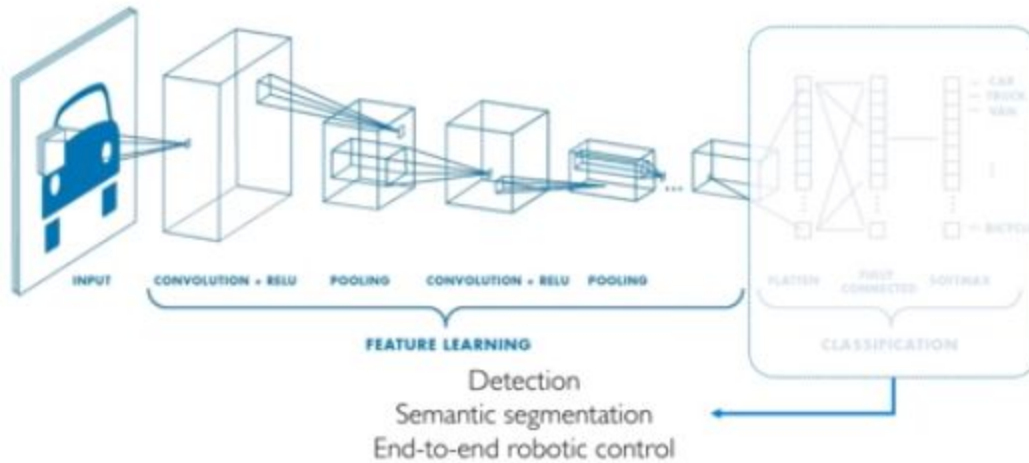
def generate_model():
    model = tf.keras.Sequential([
        # first convolutional layer
        tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # fully connected classifier
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax') # 10 outputs
    ])
    return model
```



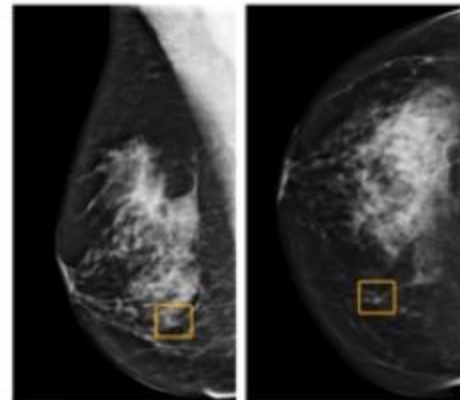
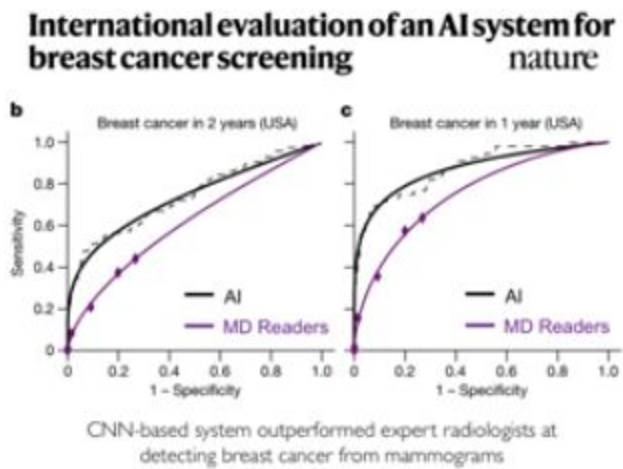
# An Architecture for Many Applications



The first half can remain the same , but the the end can to altered to fit.

EXAMPLEs :

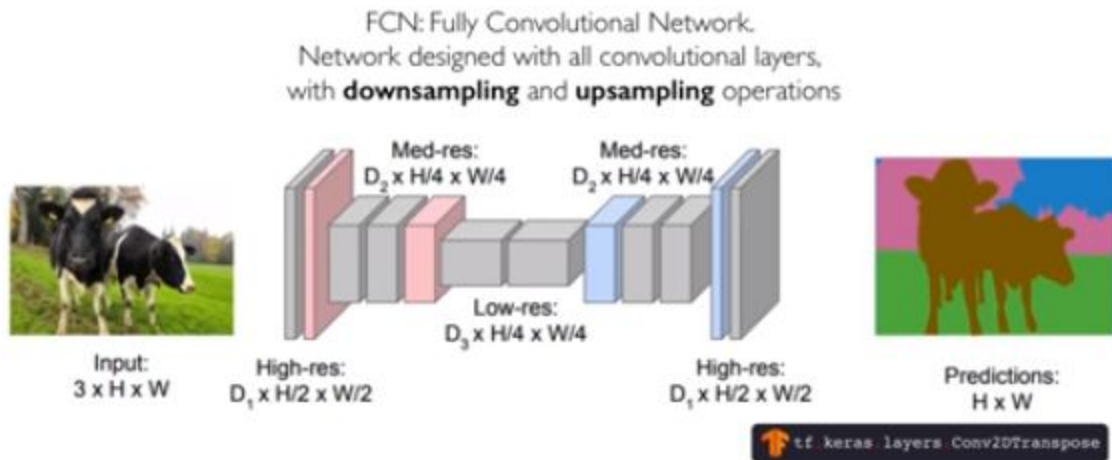
## Detection: Breast Cancer Screening



Breast cancer case missed by radiologist but detected by AI

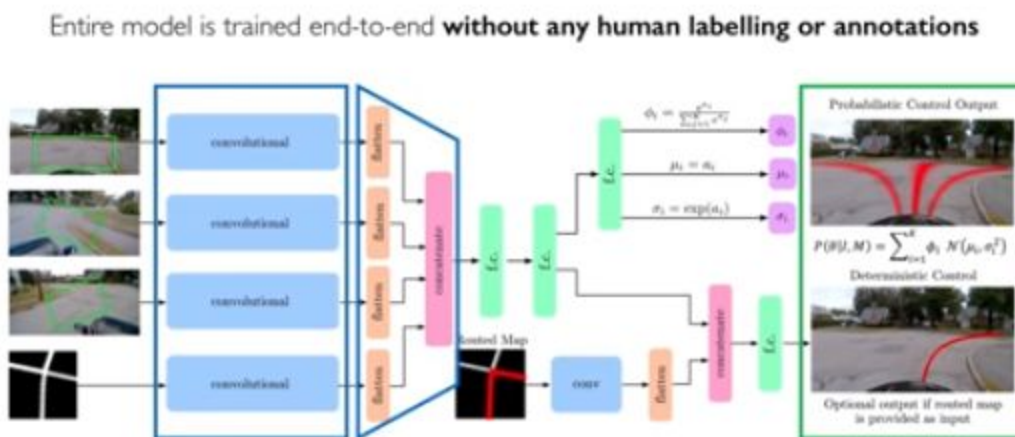


## Semantic Segmentation: Fully Convolutional Networks



The picture fed into a convolution feature extractor then its decoded / upsampled.  
Class of the features are found out.  
Upsampling is performed using transpose convolution

## End-to-End Framework for Autonomous Navigation



This whole thing is end to end , nothing was told explicitly.

# DEEP GENERATIVE MODELING

## Supervised Learning

**Data:**  $(x, y)$   
 $x$  is data,  $y$  is label

**Goal:** Learn function to map  
 $x \rightarrow y$

**Examples:** Classification, regression, object detection, semantic segmentation, etc.

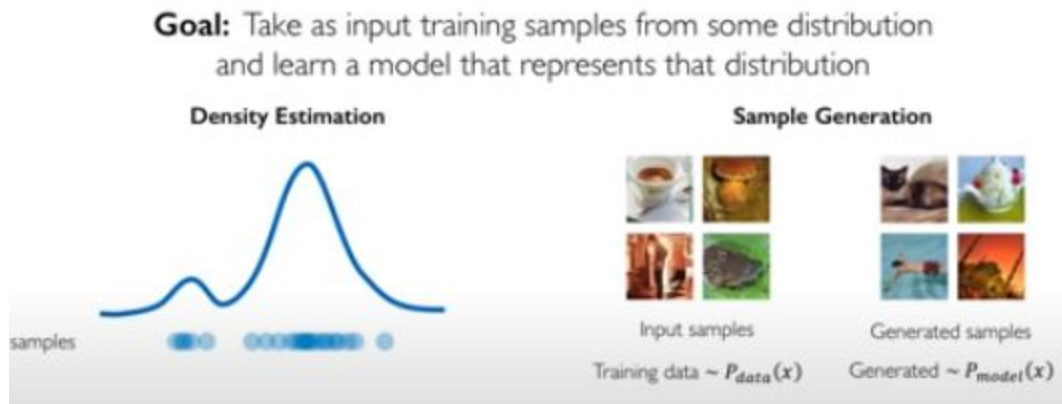
## Unsupervised Learning

**Data:**  $x$   
 $x$  is data, no labels!

**Goal:** Learn the *hidden* or *underlying structure* of the data

**Examples:** Clustering, feature or dimensionality reduction, etc.

Generative Modeling :

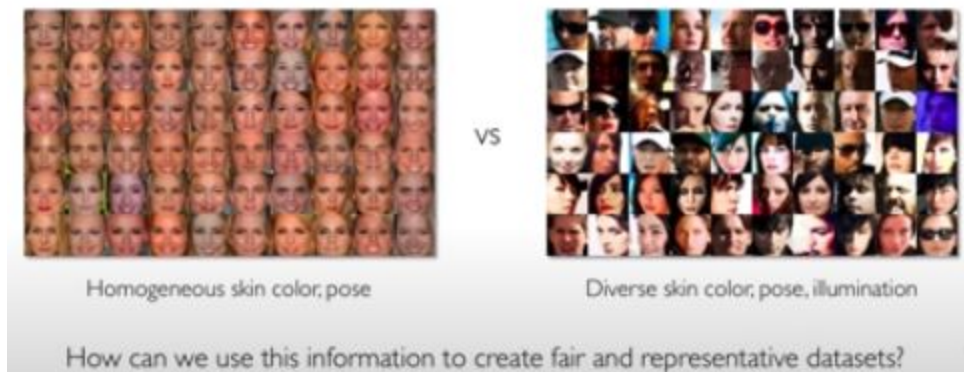


How can we model some probability distribution that's similar to the true distribution.

Generative models :

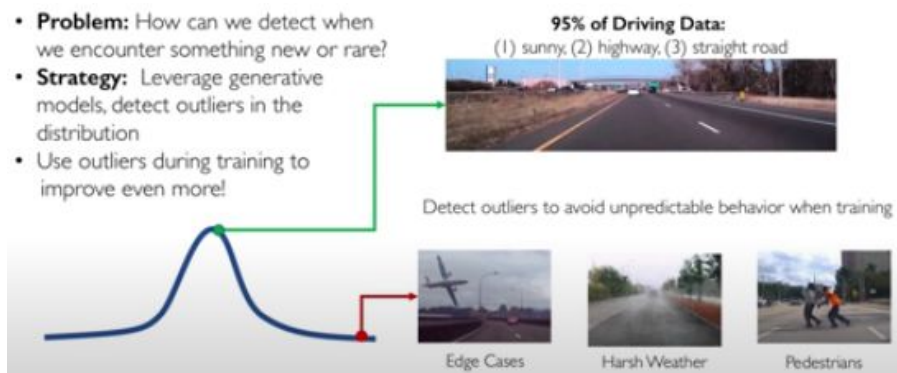
Debiasing

Capability to uncover the underlying features.



Outlier Detection

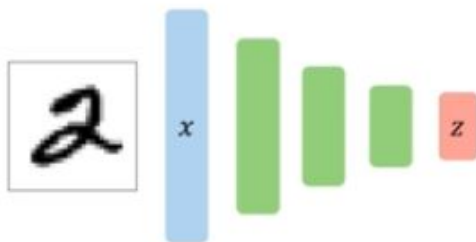
- **Problem:** How can we detect when we encounter something new or rare?
- **Strategy:** Leverage generative models, detect outliers in the distribution
- Use outliers during training to improve even more!



Latent Variable : they are not directly visible but are the things that actual matter.

Autoencoders :

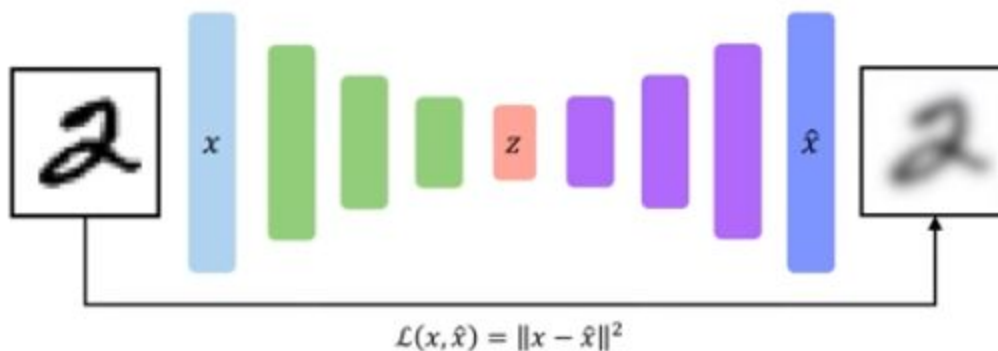
Unsupervised approach for learning a **lower-dimensional** feature representation from unlabeled training data



"Encoder" learns mapping from the data,  $x$ , to a low-dimensional latent space,  $z$

How can we learn this latent space?

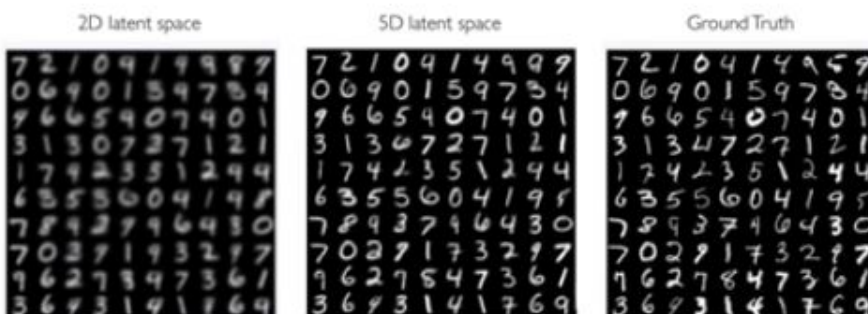
Train the model to use these features to **reconstruct the original data**



This loss function doesn't have any labels.

Dimensionality of latent space →  
reconstruction quality

Autoencoding is a form of compression!  
Smaller latent space will force a larger training bottleneck



Lower the dimensionality lower is the quality of output.

# Autoencoders for representation learning

**Bottleneck hidden layer** forces network to learn a compressed latent representation

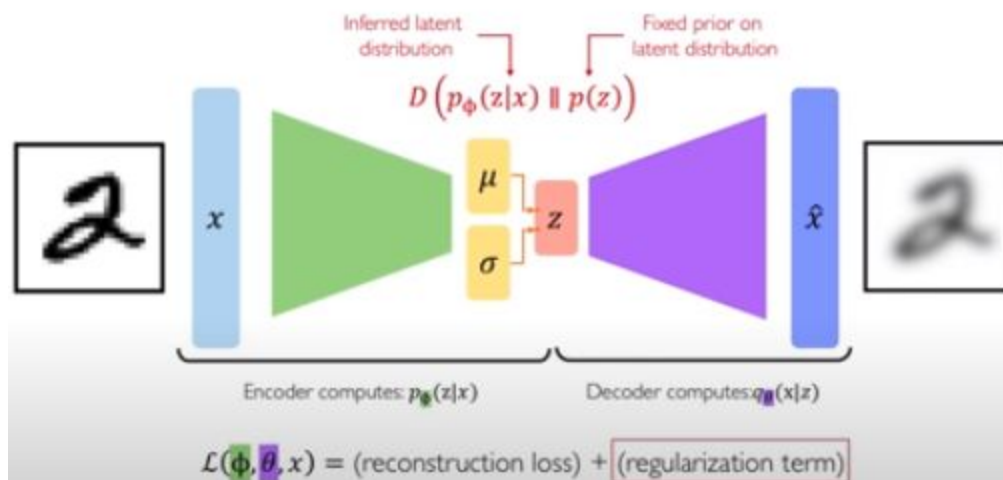
**Reconstruction loss** forces the latent representation to capture (or encode) as much "information" about the data as possible

**Autoencoding** = Automatically **encoding** data

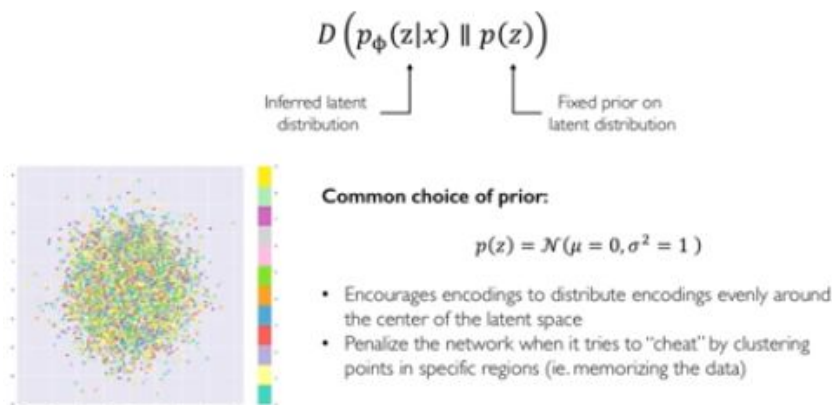
Variational Auto Encoders :

Instead of a deterministic layer there is a stochastic sampling operation , i.e for each learn a  $\mu$  and  $\sigma$  (deviations) that represent the probability distribution.

## VAE optimization



## Priors on the latent distribution



Regularisation term that is used to formulate the total loss. The Gaussian term.

$$D(p_\phi(z|x) \parallel p(z))$$

$$= -\frac{1}{2} \sum_{j=0}^{k-1} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$$

KL-divergence between the two distributions

We cannot backpropagate gradients through a sampling layer, due to their stochastic nature as bp requires chain rule.

So we can Reparameterize the sampling layer.

## Reparametrizing the sampling layer

**Key Idea:**

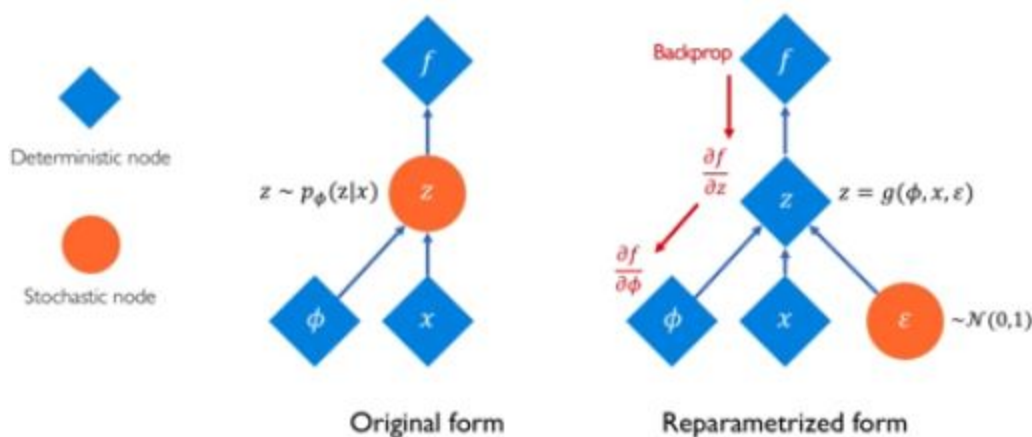
$$z \sim \mathcal{N}(\mu, \sigma^2)$$

Consider the sampled latent vector  $z$  as a sum of

- a fixed  $\mu$  vector;
- and fixed  $\sigma$  vector, scaled by random constants drawn from the prior distribution

$$\Rightarrow z = \mu + \sigma \odot \epsilon$$

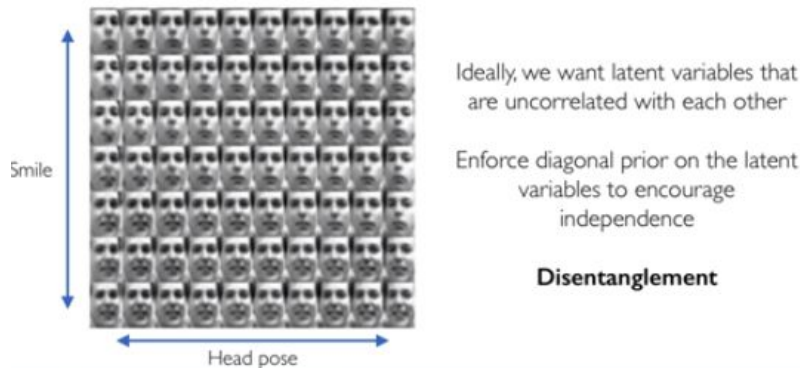
where  $\epsilon \sim \mathcal{N}(0,1)$



Slow tuning of the latent variable (increase or decrease), then run the decoder to get the output. Therefore it forms a semantic meaning.

By perturbing the value of a single latent variable we actually get to know how what they mean and represent.

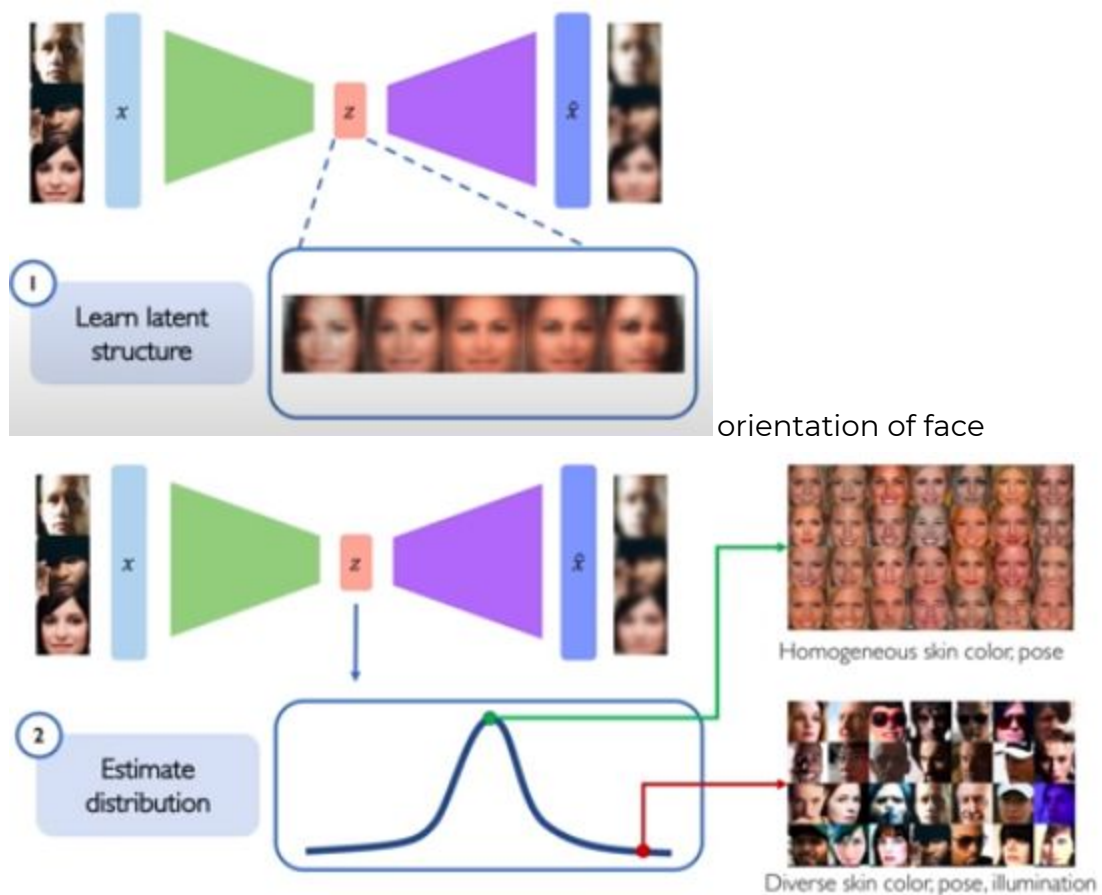




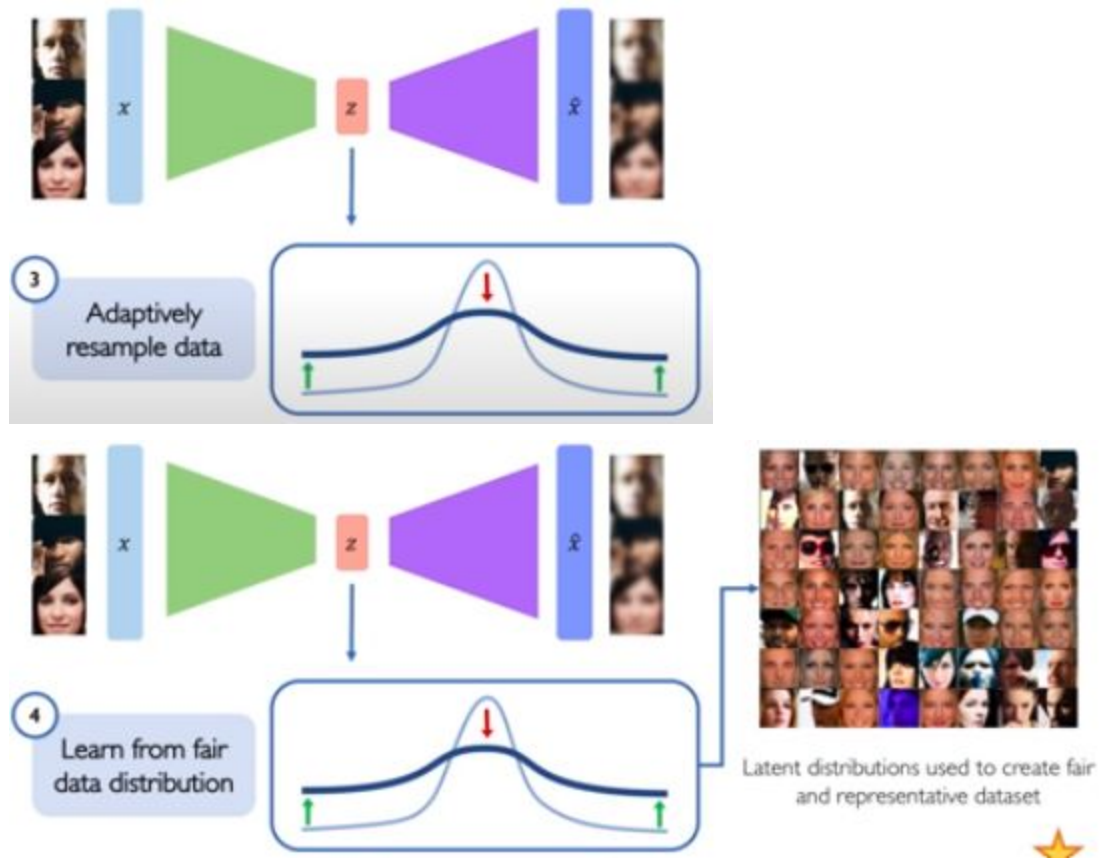
All other variables are fixed other and then these are fine tuned accordingly , Disentanglement lets the model learn features not being correlated to one another.

We can use biases to get these.

Mitigating bias through learnt latent structures :



These may be under-represented.



VAEs :

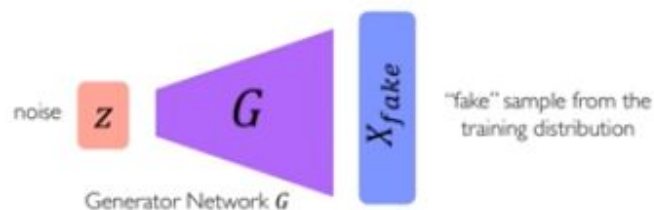
1. Compress representation of world to something we can use to learn
2. Reconstruction allows for unsupervised learning (no labels!)
3. Reparameterization trick to train end-to-end
4. Interpret hidden latent variables using perturbation
5. Generating new examples

Generative Adversarial Networks : (GANs)

**Idea:** don't explicitly model density, and instead just sample to generate new instances.

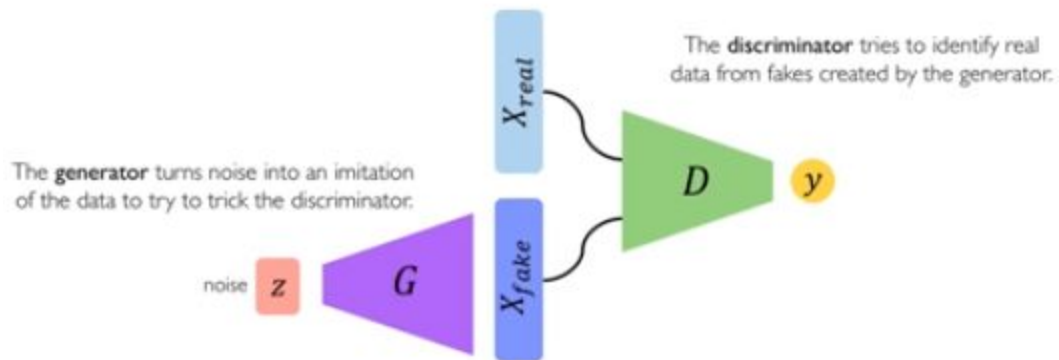
**Problem:** want to sample from complex distribution – can't do this directly!

**Solution:** sample from something simple (noise), learn a transformation to the training distribution.





**Generative Adversarial Networks (GANs)** are a way to make a generative model by having two neural networks compete with each other.



Generator and Discriminator ~ both these are adversaries , as they are fighting with each other.

This forces the discriminator to become as better as possible to differentiate the real from the created one.

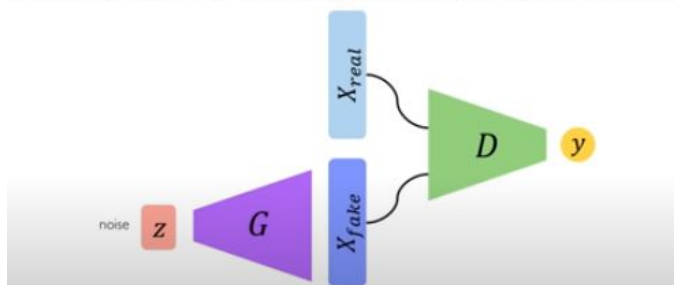
Intuition behind the GANs :

1. Generator starts from noise to try to create an imitation of the data.
2. Discriminator looks at both the real data and fake data created by the generator.
3. Discriminator tries to predict what's real and what's fake.



4. As the Discriminator becomes perfect the Generator tries to improve its imitation of the data.

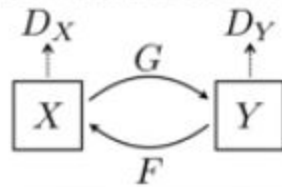
After training, use generator network to create **new data** that's never been seen before.



EXAMPLE :

## CycleGAN: domain transformation

CycleGAN learns transformations across domains with unpaired data.



## CycleGAN: Transforming speech

