

Deep Learning

Seungsang Oh

Dept of Mathematics
& Dept of Data Science
Korea University

Introduction to Deep Learning

Deep Neural Network

Convolutional Neural Network

Recurrent Neural Network

Attention Mechanism

Auto-Encoder & VAE

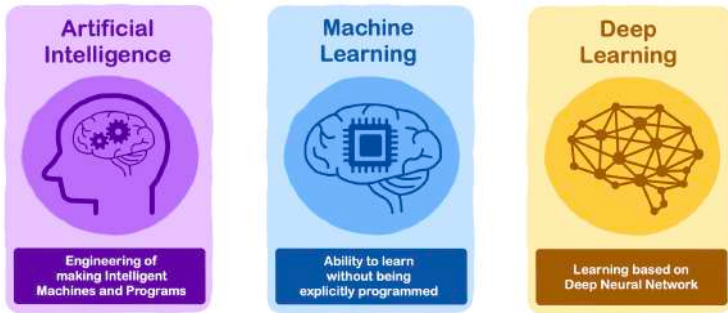
Generative Adversarial Network

Natural Language Processing

Graph Neural Network

Artificial Intelligence

- **Artificial Intelligence (AI)** is the branch of computer science building smart machines performing tasks that require human intelligence, birthed at Dartmouth Conferences in 1956.
- **Machine Learning (ML)** is an approach to achieve AI. Rather than hand-coding software, machines are trained using large data and algorithms and learn how to perform the task.
- **Deep Learning (DL)** is a technique for implementing ML, thanks to Hinton's DBN in 2006. They learn based on huge (deep) neural networks and then run massive amounts of data.



Machine Learning algorithm categories

Supervised Learning

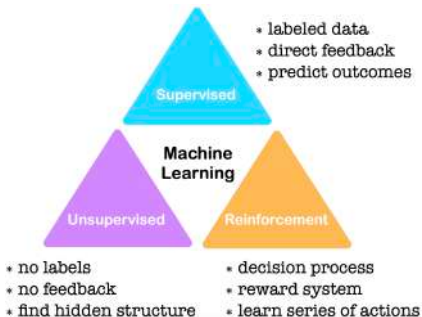
- It learns from a training data set **with labels**.
- It learns a general rule for **regression** or **classification** to predict the labels for the remaining data.

Unsupervised Learning

- It learns from a training data set **without any label**.
- It detects patterns in the data by **clustering** similar data that have common characteristics.

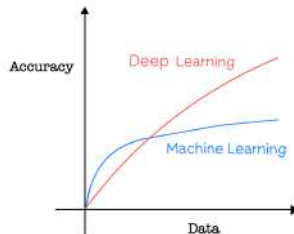
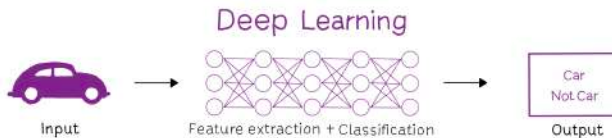
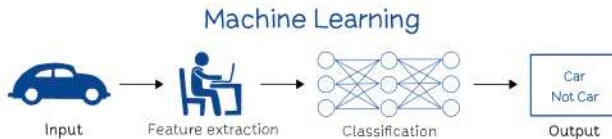
Reinforcement Learning

- It learns by **interacting with the environment**, rather than from a sample data set.
- It chooses an **action** at each **state**, and receives a **reward** indicating how good an action is. Therefore, during training, it adapts its **policy** in order to maximize the total reward.
- It is often used in robotics, game playing or autonomous cars.



Machine Learning vs Deep Learning

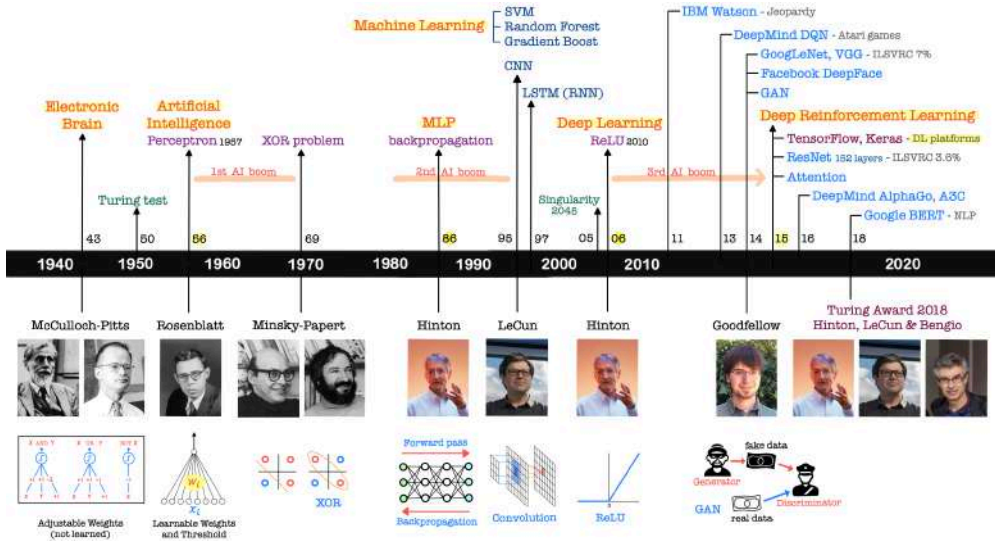
- In **Machine Learning**, humans extract patterns and co-relation between ‘features’, but it is difficult to know all the features that would need to be extracted.
- **Deep Learning** uses many layers to progressively extract higher level features from raw input, and huge amount of learnable parameters become more accurate as we feed more data, but many hidden layers lead to a complex structure that is considered as a black box.



Algorithms

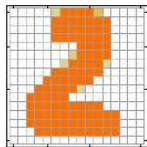
•

Milestones in Deep Learning developments



Deep Neural Network

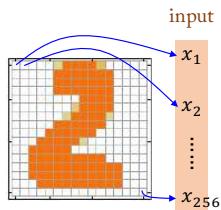
Deep Learning motivation



Machine



“2”



$16 \times 16 = 256$
ink $\rightarrow 1$
no ink $\rightarrow 0$

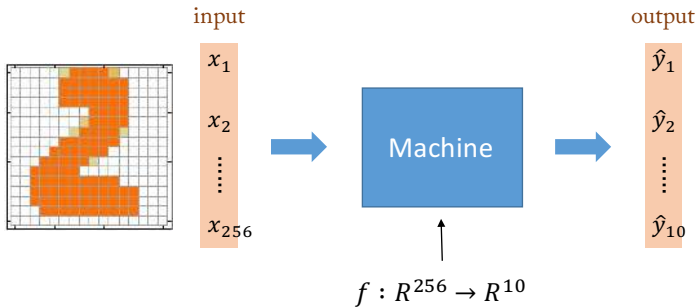


output

\hat{y}_1	0.09	is “1”
\hat{y}_2	0.71	is “2”
\vdots	\vdots	\vdots
\hat{y}_{10}	0.03	is “0”

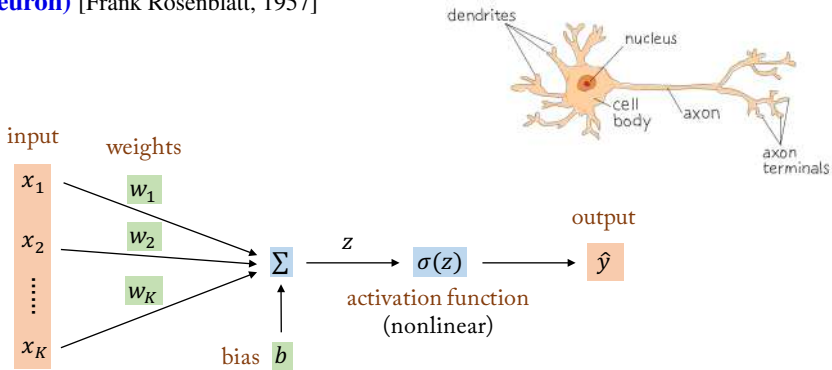
the image is “2”

↑
Each dimension represents
the confidence of a digit.



In Deep Learning, the function f is represented by a Neural Network.

Perceptron (neuron) [Frank Rosenblatt, 1957]



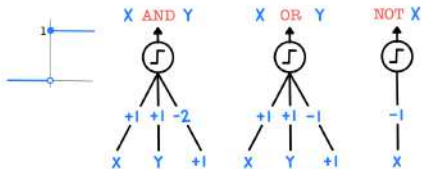
$$f: \mathbb{R}^K \rightarrow \mathbb{R}$$

$$\hat{y} = \sigma(z) = \sigma(w_1x_1 + w_2x_2 + \dots + w_Kx_K + b)$$

nonlinear

weighted sum (linear)

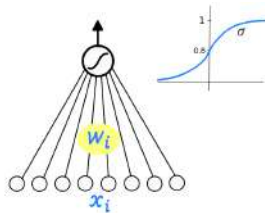
Programming



Logic gate
(programmed weights)

X	Y	X AND Y	X OR Y	NOT X
0	0	0	0	1
1	0	0	1	0
0	1	0	1	
1	1	1	1	

Learning

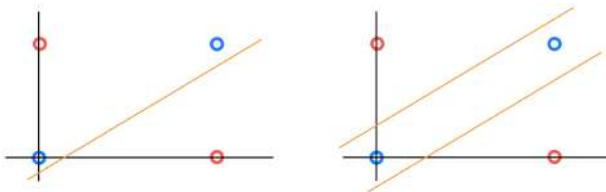
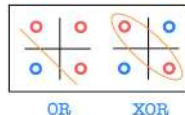


Perceptron
(learnable weights w_i)

$$\sigma(w_1x_1 + \cdots + w_Kx_K + b) \rightarrow 0 \text{ or } 1?$$

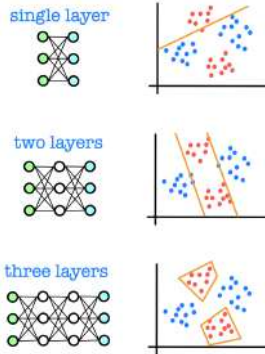
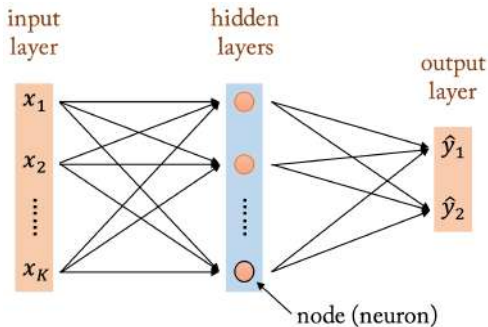
XOR problem

- XOR (exclusive OR) problem is a classic problem in AI.
- Minsky and Papert (the book 'Perceptrons' 1969) showed that this is a serious problem for neural networks of the 1960s like Perceptrons. \Rightarrow AI Winter I
- The limitation of a single-layer perceptron architecture is that it is capable of separating data points with a single line.
But, many data like XOR inputs are not linearly separable.
- Later, multilayer perceptron architecture is capable of achieving nonlinear separation.



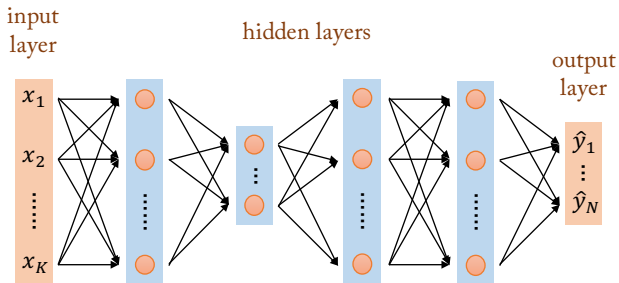
Multilayer Perceptron (MLP)

- **MLP** is a class of feedforward artificial neural network.
- MLP consists of an input layer, several hidden layers and an output layer.
- Except input nodes, each node is a neuron (perceptron) that uses a **nonlinear activation function**.



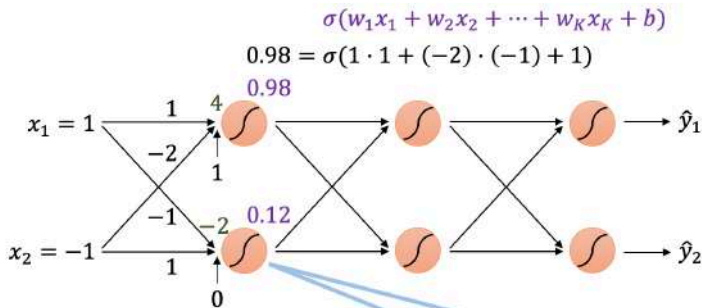
Deep Neural Network (DNN)

- **DNN** is an artificial neural network with many hidden layers.
- More layers enable composition of features from lower layers, modeling complex data with fewer units than a shallow neural network with similar performance.



‘Deep’ means many hidden layers.

DNN forward pass computation

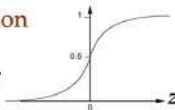


$$\sigma\left(\begin{bmatrix} 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right)$$

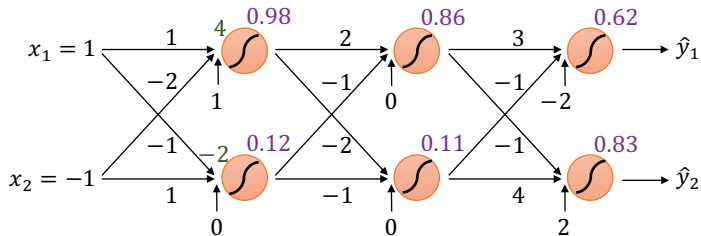
$$= \sigma\left(\begin{bmatrix} 4 \\ -2 \end{bmatrix}\right) = \begin{bmatrix} 0.98 \\ 0.12 \end{bmatrix}$$

sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



activation function

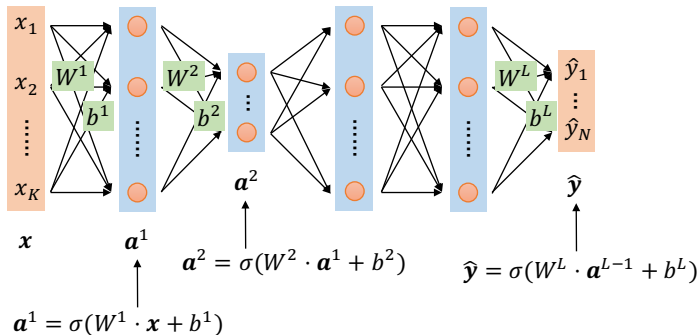


$$f: \mathbb{R}^2 \rightarrow \mathbb{R}^2$$

$$f\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 0.62 \\ 0.83 \end{bmatrix}$$

DNN forward pass computation

- Use **matrix computations**: **nonlinearity σ** is important.



$$\hat{y} = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

- If σ is not used, what happen?

Softmax

- Softmax layer is used as the output layer.
- Softmax function is $\frac{e^{z_i}}{\sum e^{z_j}}$ (not a real probability distribution, but it looks similar).

Ordinary layer

$$z_1 \xrightarrow{3} \sigma \rightarrow \hat{y}_1 = \sigma(z_1) \quad 0.95$$

$$z_2 \xrightarrow{1} \sigma \rightarrow \hat{y}_2 = \sigma(z_2) \quad 0.73$$

$$z_3 \xrightarrow{-3} \sigma \rightarrow \hat{y}_3 = \sigma(z_3) \quad 0.05$$

Output of the network can be any value.
So, may not be easy to interpret.

Softmax layer

$$z_1 \xrightarrow{3} \rightarrow \hat{y}_1 = \frac{e^{z_1}}{\sum e^{z_j}} \quad 0.88$$

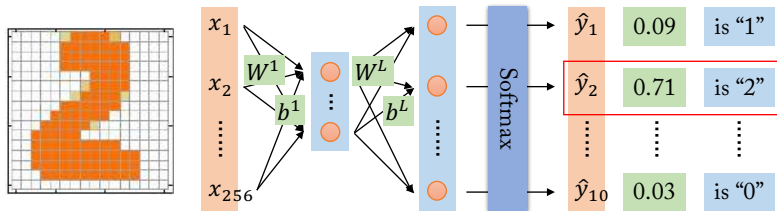
$$z_2 \xrightarrow{1} \rightarrow \hat{y}_2 = \frac{e^{z_2}}{\sum e^{z_j}} \quad 0.12$$

$$z_3 \xrightarrow{-3} \rightarrow \hat{y}_3 = \frac{e^{z_3}}{\sum e^{z_j}} \approx 0.0$$



Probability shape: $0 \leq \hat{y}_i \leq 1$
 $\sum \hat{y}_i = 1$

Setting network parameters

- Learnable parameters (weights): $\theta = \{W^1, b^1, W^2, b^2, \dots, W^L, b^L\}$



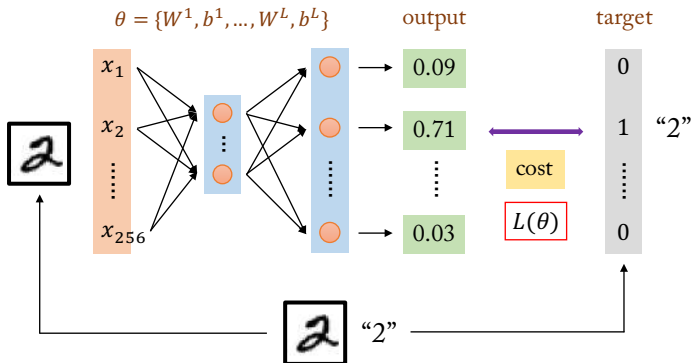
Set the network parameters θ so that

- input:  $\implies \hat{y}_1$ has the maximum value.
- input:  $\implies \hat{y}_2$ has the maximum value.

How to let the neural network achieve this?

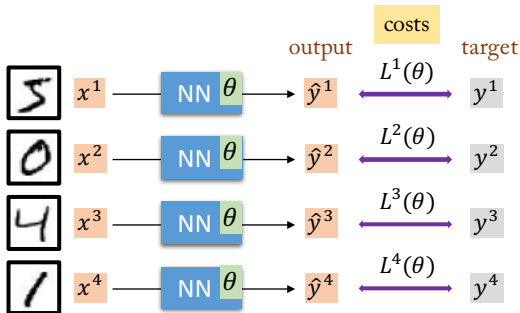
Cost (loss)

- Given network parameters θ , each sample data (an image with label) has a **cost value** $L(\theta)$.
- Cost** is **MSE** or **cross entropy** of the **output** (probability shape) and the **target** (one-hot coding).



Total Cost

- All costs are added up for all training data.
- The total cost shows how bad the network parameters θ are.



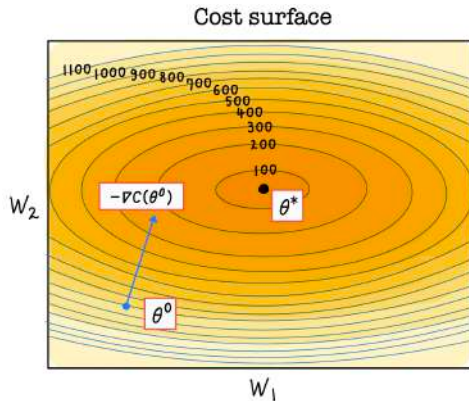
Total Cost

$$C(\theta) = \sum_{n=1}^N L^n(\theta)$$

Aim: Learn the network parameters θ that minimize the total cost $C(\theta)$.

Gradient Descent

- **Gradient descent** is a first-order (requiring first-derivative/gradient) iterative optimization algorithm for finding the minimum of a function.



network with two parameters $\theta = \{w_1, w_2\}$

Randomly pick a starting point θ^0 .

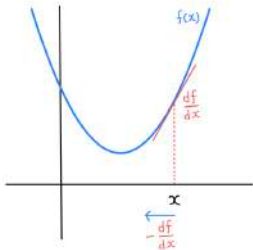
Compute the negative gradient at each θ^t .

\Rightarrow

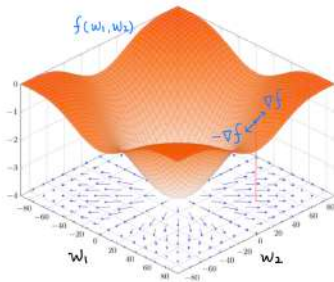
$$- \nabla_{\theta} C(\theta^t)$$

$$\nabla_{\theta} C(\theta^t) = \begin{bmatrix} \frac{\partial C(\theta^t)}{\partial w_1} \\ \frac{\partial C(\theta^t)}{\partial w_2} \end{bmatrix}$$

Derivative

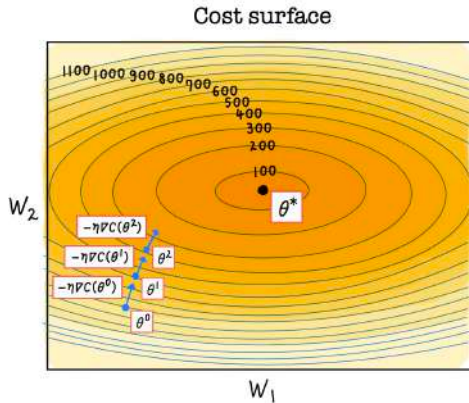


Gradient



* Gradient vector $\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$ indicates the **direction and rate of fastest increase**.

- To find the minimum more efficiently, we use the **learning rate η** .



Randomly pick a starting point θ^0 .

Compute the negative gradient at each θ^t
followed by multiplying η .

$$\Rightarrow -\eta \nabla_{\theta} C(\theta^t)$$

Gradient Descent variants

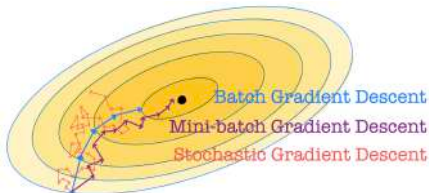
- (1) **Batch Gradient Descent**: batch size = size of training dataset
- (2) **Stochastic Gradient Descent (SGD)**: batch size = 1
- (3) **Minibatch Gradient Descent**

- Difference: amount of **batch** (data used per parameter update)
- Minibatch gradient descent is commonly called **SGD**.

- * **Batch size**: number of samples present in a single batch.
 - * Number of **epochs**: number of complete passes through the entire training dataset.
 - * **Iteration**: number of batches needed to complete one epoch.
 - * They are hyperparameters for the learning algorithm, so you must specify them.
- There are usually no magic rules for how to choose these parameters.

Batch Gradient Descent

- Compute the gradient for the **entire dataset** (every epoch).
- Update equation $\theta^{t+1} = \theta^t - \eta \nabla_{\theta} C(\theta^t)$ which is $w_i(t+1) = w_i(t) - \eta \frac{\partial C}{\partial w_i}$ for each weight w_i .
- Pros: guaranteed to converge to the global minimum for convex cost surfaces
robust to noise
- Cons: need huge memory capacity
slow
no online learning

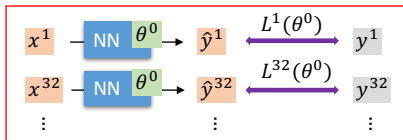


Stochastic Gradient Descent

- Compute the gradient for **each example** (x^n, y^n).
- Update equation $\theta^{t+1} = \theta^t - \eta \nabla_{\theta} L^n(\theta^t)$ which is $w_i(t+1) = w_i(t) - \eta \frac{\partial L^n}{\partial w_i}$ for each weight w_i .
- Pros: fast
online learning
- Cons: high variance updates (SGD fluctuation)
sensitive to noise

Minibatch Gradient Descent

- Compute the gradient for **every minibatch** of m random samples until all minibatches have been picked (1 epoch) and repeat the same process for many epochs.
- Update equation $\theta^{t+1} = \theta^t - \eta \nabla_{\theta} C(\theta^t)$ which is $w_i(t+1) = w_i(t) - \eta \frac{\partial C}{\partial w_i}$ for all i where $C(\theta^t) = L^{n_1}(\theta^t) + \dots + L^{n_m}(\theta^t)$ is the minibatch cost.
- Pros: fast, online learning, low variance updates (robust to noise)
- Cons: minibatch size is a hyperparameter.

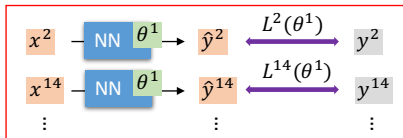


Randomly initialize θ^0 .

Train the first batch to get θ^1 .

$$C(\theta^0) = L^1 + L^{32} + \dots$$

$$\theta^1 \leftarrow \theta^0 - \eta \nabla_{\theta} C(\theta^0)$$



Train the second batch to get θ^2 .

$$C(\theta^1) = L^2 + L^{14} + \dots$$

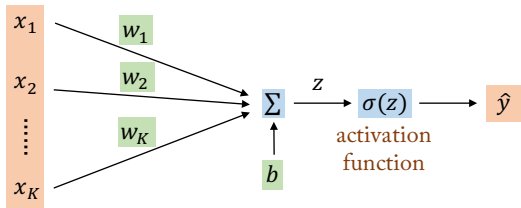
$$\theta^2 \leftarrow \theta^1 - \eta \nabla_{\theta} C(\theta^1)$$

Backpropagation

Backpropagation is the way of computing the gradient efficiently to update millions of network parameters (firstly applied in NN by Hinton 1986).

Single layer

- Input: $\mathbf{x} = (x_1, x_2, \dots, x_K)^T$
- Output: \hat{y}
- Model: a weight vector $\mathbf{w} = (w_1, w_2, \dots, w_K)^T$ and a bias b



$$\hat{y} = f(z) = f(\sum w_i x_i + b) = f(\mathbf{w}^T \mathbf{x} + b)$$

Forward Pass

Geoffrey Hinton



Known for	Applications of Boltzmann machine Deep learning Capsule neural network
Awards	AAAI Fellow (1990) Rumelhart Prize (2001) IJCAI Award for Research Excellence (2005) IEEE Frank Rosenblatt Award (2014) James Clerk Maxwell Medal (2016) BBVA Foundation Frontiers of Knowledge Award (2016) Turing Award (2018)
Institutions	University of Toronto Google Carnegie Mellon University University College London
Website	www.cs.toronto.edu/~hinton/

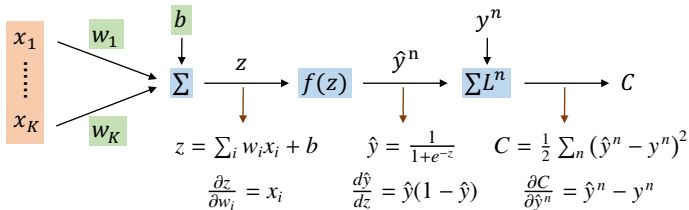
Gradient descent

- Training data: $(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), \dots, (\mathbf{x}^N, y^N)$
- Goal: find the optimal parameter \mathbf{w}^* , which minimizes the total cost (MSE)

$$C = \frac{1}{2} \sum_{n=1}^N (\hat{y}^n - y^n)^2 \text{ where } \hat{y}^n = f(\mathbf{w}^T \mathbf{x}^n + b)$$

- **Backpropagation**: iteratively updating the model parameters \mathbf{w} to decrease C by using

$$w_i(t+1) = w_i(t) - \eta \frac{\partial C}{\partial w_i}$$

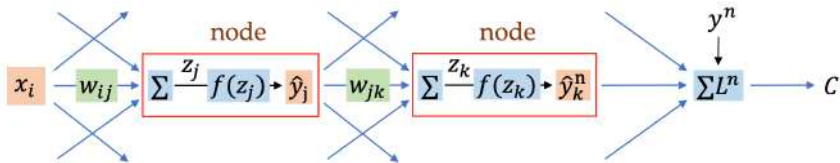


$$\frac{\partial C}{\partial w_i} = \sum_{n=1}^N \frac{\partial z^n}{\partial w_i} \frac{d\hat{y}^n}{dz^n} \frac{\partial C}{\partial \hat{y}^n} = \sum_{n=1}^N x_i^n \hat{y}^n (1 - \hat{y}^n) (\hat{y}^n - y^n)$$

Chain rule

Multilayer

- Update weights: $w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial C}{\partial w_{ij}}$



$$\frac{\partial C}{\partial \hat{y}_j} = \sum_k \frac{\partial z_k}{\partial \hat{y}_j} \frac{\partial \hat{y}_k}{\partial z_k} \frac{\partial C}{\partial \hat{y}_k} = \sum_k w_{jk} \frac{\partial \hat{y}_k}{\partial z_k} \frac{\partial C}{\partial \hat{y}_k} \quad \text{where } z_k = \sum_j w_{jk} \hat{y}_j + b_k$$

$$\frac{\partial C}{\partial w_{ij}} = \sum_n \frac{\partial z_j^n}{\partial w_{ij}} \frac{d\hat{y}_j^n}{dz_j^n} \frac{\partial C}{\partial \hat{y}_j^n} = \sum_n \frac{\partial z_j^n}{\partial w_{ij}} \frac{d\hat{y}_j^n}{dz_j^n} \left[\sum_k w_{jk} \frac{d\hat{y}_k^n}{dz_k^n} \frac{\partial C}{\partial \hat{y}_k^n} \right] = \sum_n x_i^n \hat{y}_j^n (1 - \hat{y}_j^n) \left[\sum_k w_{jk} \hat{y}_k^n (1 - \hat{y}_k^n) (\hat{y}_k^n - y_k^n) \right]$$

repeat for each layer

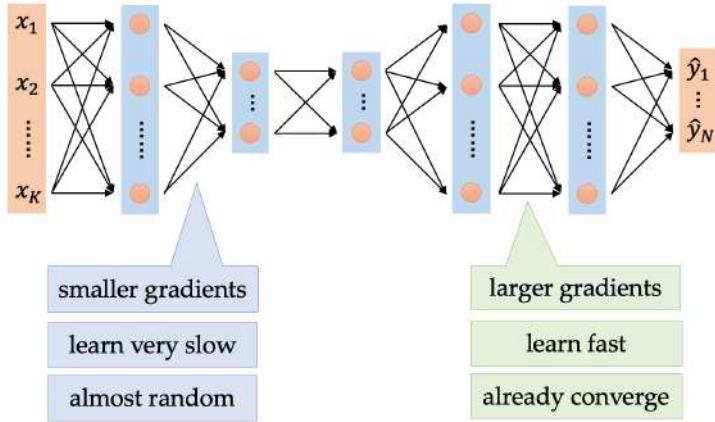
Vanishing Gradient

Vanishing Gradient Problem for multilayer

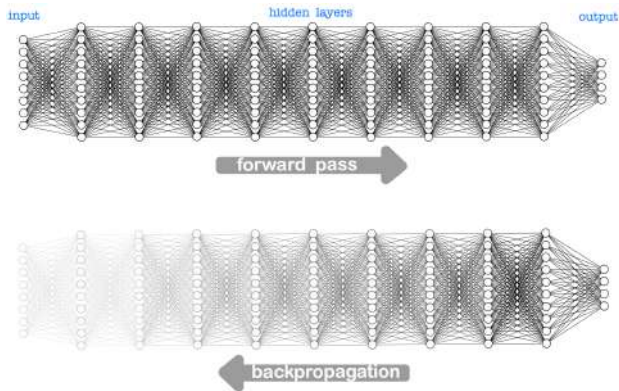
- The gradient in front layers **decreases exponentially** with the number L of remaining layers, effected by multiplying L of the small numbers $\frac{d\hat{y}_i^n}{dz_i^n} = \hat{y}_i^n(1 - \hat{y}_i^n)$ for $0 < \hat{y}_i^n = \sigma(z_i) < 1$.

$$\frac{\partial C}{\partial w_{hi}} = \sum_n \frac{\partial z_i^n}{\partial w_{hi}} \frac{d\hat{y}_i^n}{dz_i^n} \sum_j w_{ij} \frac{d\hat{y}_j^n}{dz_j^n} \sum_k w_{jk} \frac{d\hat{y}_k^n}{dz_k^n} \frac{\partial C}{\partial \hat{y}_k^n} \longrightarrow 0$$

- This causes that the **parameters w are trained very slowly**.
- Optimizing the multilayer network needs huge time cost.
- Some possible solutions are using other activation functions like ReLU instead of sigmoid or adding the updates instead of multiplying them as in ResNet and LSTM.

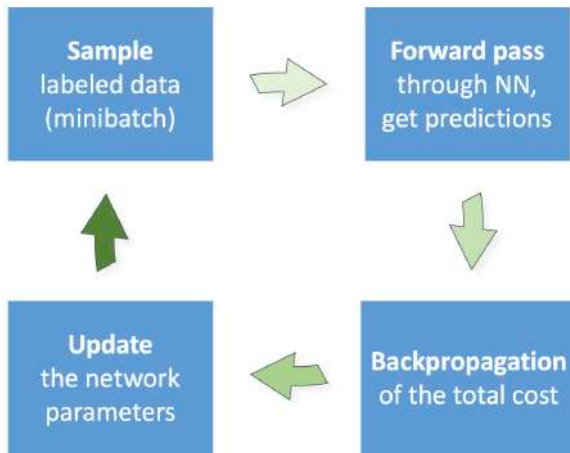


Vanishing Gradient (AI winter II)

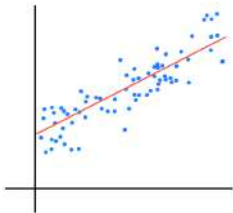


Hinton applied **Backpropagation** in neural networks in 1986,
and use **ReLU** to overcome the vanishing gradient problem in 2010.

Deep Learning training cycle

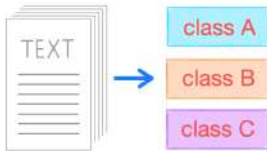


Types of Learning



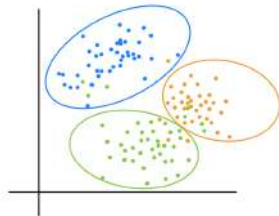
Regression

Finding the relationship between independent variables x_1, x_2, \dots , and one dependent continuous variable y



Classification

Identifying an individual class to which new data belongs, based on which training data sets the class is known to.



Clustering

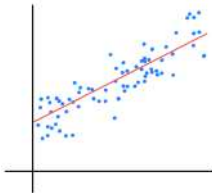
Grouping a set of objects so that objects in the same group (cluster) are more similar than the others.

Linear Regression

- Given input-output samples $\{(x^n, y^n)\}$ of a function $y = f(x)$, we would like to **learn** f and evaluate it on new data.
- In (univariate) **linear regression**, we want to find the **best line to explain the data**.
- Training data: $\{(x^1, y^1), \dots, (x^N, y^N)\}$

Hypothesis: $h(x) = wx + b$

Cost: $C(w, b) = \frac{1}{2N} \sum_{n=1}^N (h(x^n) - y^n)^2$
(mean square error)



- To find w and b minimizing $C(w, b)$, apply **gradient descent** repeatedly:

$$\begin{cases} w^{t+1} = w^t - \eta \frac{\partial C}{\partial w} = w^t - \frac{\eta}{N} \sum_{n=1}^N (h^t(x^n) - y^n) x^n \\ b^{t+1} = b^t - \eta \frac{\partial C}{\partial b} = b^t - \frac{\eta}{N} \sum_{n=1}^N (h^t(x^n) - y^n) \end{cases}$$

where $h^t(x) = w^t x + b^t$ starting with random normal initials w^0, b^0 .

Linear regression for multiple variables

- Training data: $\{(\mathbf{x}^1, y^1), \dots, (\mathbf{x}^N, y^N)\}$ where $\mathbf{x}^n = (x_1^n, \dots, x_K^n)^T$

Hypothesis: $h_{\theta}(\mathbf{x}) = w_1 x_1 + \dots + w_K x_K + b \cdot 1 = \mathbf{w}^T \mathbf{x}$

considered as $\theta = \mathbf{w} = (w_1, \dots, w_K, b)^T$ and $\mathbf{x} = (x_1, \dots, x_K, 1)^T$.

Cost: $C(\theta) = \frac{1}{2N} \sum_{n=1}^N (\mathbf{w}^T \mathbf{x}^n - y^n)^2 \Leftarrow \text{MSE (mean square error)}$

- To find the parameter θ minimizing $C(\theta)$, apply **gradient descent** repeatedly:

$$\left\{ w_i^{t+1} = w_i^t - \eta \frac{\partial C}{\partial w_i} = w_i^t - \frac{\eta}{N} \sum_{n=1}^N (\mathbf{w}^T \mathbf{x}^n - y^n) x_i^n \right\}$$

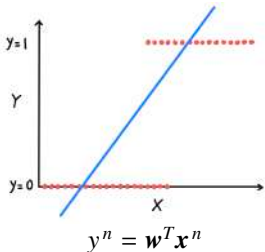
simultaneously updating for every weight w_i and the bias b^t .

- TensorFlow implementation: $h_{\theta}(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$

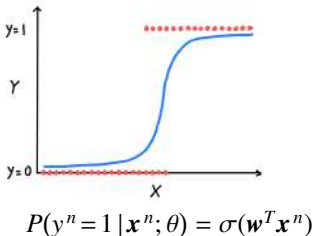
Logistic Regression (Binary Classification)

- **Linear Regression** is used to predict continuous (real) values, but **Logistic Regression** is used to predict one of $Y = \{0, 1\}$.
(\Rightarrow **Binary Classification** of 1/0, pass/fail, win/lose, or healthy/sick)
- Since the output of a linear equation $y = \sum w_i x_i + b$ is a real number in $(-\infty, \infty)$, we squash it into $(0, 1)$ by using the sigmoid function σ , called a **logistic** function.
- Practically, we try to predict the conditional probability distribution $P(Y=1 | X)$, which is continuous. So this is also a **regression** problem

Linear Regression



Logistic Regression

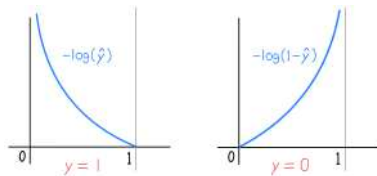


Logistic Regression's **cost function** differs from **linear regression** to be suited for $\{0, 1\}$ data.

- Training data: $\{(\mathbf{x}^1, y^1), \dots, (\mathbf{x}^N, y^N)\}$ where $y^n \in \{0, 1\}$

$$\text{Hypothesis: } h_{\theta}(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}$$

$$\begin{aligned} \text{Cost: } C(h_{\theta}(\mathbf{x}), y) &= \begin{cases} -\log(h_{\theta}(\mathbf{x})) & \text{if } y = 1 \\ -\log(1 - h_{\theta}(\mathbf{x})) & \text{if } y = 0 \end{cases} \\ &= -y \cdot \log(h_{\theta}(\mathbf{x})) - (1 - y) \cdot \log(1 - h_{\theta}(\mathbf{x})) \end{aligned}$$



$$\text{Cross-Entropy Loss: } CE(\theta) = -\frac{1}{N} \sum_{n=1}^N \left[y^n \log(h_{\theta}(\mathbf{x}^n)) + (1 - y^n) \log(1 - h_{\theta}(\mathbf{x}^n)) \right]$$

- To find the parameter θ minimizing $CE(\theta)$, apply **gradient descent** repeatedly:

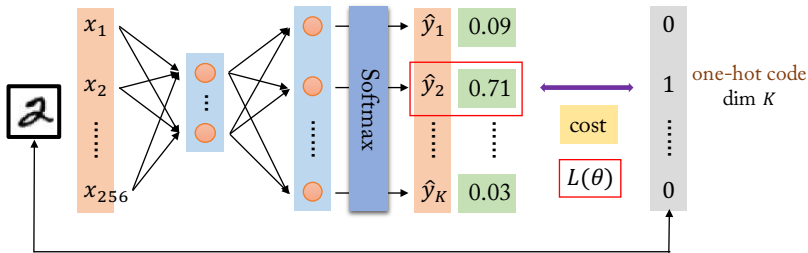
$$\left\{ \mathbf{w}_i^{t+1} = \mathbf{w}_i^t - \eta \frac{\partial CE(\theta)}{\partial \mathbf{w}_i} = \mathbf{w}_i^t - \frac{\eta}{N} \sum_{n=1}^N x_i^n (h_{\theta}(\mathbf{x}^n) - y^n) \right\}$$

simultaneously updating for every weight.

* Note that $\frac{\partial CE(\theta)}{\partial \mathbf{w}_i} = -\frac{1}{N} \sum_{n=1}^N \left[y^n \frac{\sigma'(\mathbf{w}^T \mathbf{x}^n)}{\sigma(\mathbf{w}^T \mathbf{x}^n)} \frac{\partial \mathbf{w}^T \mathbf{x}^n}{\partial \mathbf{w}_i} + (1 - y^n) \frac{-\sigma'(\mathbf{w}^T \mathbf{x}^n)}{1 - \sigma(\mathbf{w}^T \mathbf{x}^n)} \frac{\partial \mathbf{w}^T \mathbf{x}^n}{\partial \mathbf{w}_i} \right] = \frac{1}{N} \sum_{n=1}^N x_i^n (h_{\theta}(\mathbf{x}^n) - y^n)$
 since $\sigma'(z) = \sigma(z)(1 - \sigma(z))$. \Leftarrow compare to MSE $\frac{\partial C(\theta)}{\partial \mathbf{w}_i} = \frac{1}{N} \sum_{n=1}^N x_i^n h_{\theta}(\mathbf{x}^n) (1 - h_{\theta}(\mathbf{x}^n)) (h_{\theta}(\mathbf{x}^n) - y^n)$

Multi-class Classification

- Creating a NN that assigns discrete classes to the input is called **classification**.
- If there are only two class labels, then this learning is binary classification. Otherwise, it's called multi-class classification.
- **Softmax layer** as the output layer is usually used for multi-class classification.

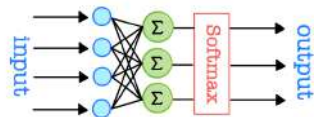
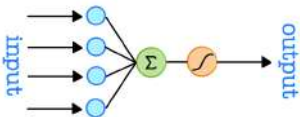
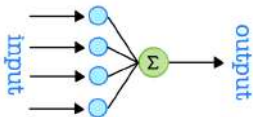


- **Categorical (multi-class) Cross-Entropy Loss** for K classes:

$$CCE(\theta) = -\frac{1}{N} \sum_{n=1}^N \sum_{j=1}^K y_j^n \log(h_{\theta}(x^n)_j)$$

Regression vs Classification

Type	Pros	Cons
Linear Regression	<ul style="list-style-type: none">• Simple implementation	<ul style="list-style-type: none">• Not guaranteed to work
Logistic Regression (Binary Classification)	<ul style="list-style-type: none">• Highly accurate• Model outputs are probability-like	<ul style="list-style-type: none">• Only supports binary labels
Softmax Classification	<ul style="list-style-type: none">• Supports multi-class classification• Model outputs are probability-like	<ul style="list-style-type: none">• Complicated implementation

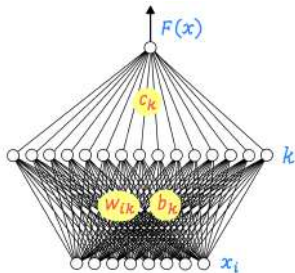


Universal Approximation Theorem

- Any continuous function on \mathbb{R}^n can be arbitrarily well approximated by a neural network with one hidden layer containing finitely many nodes and a nonlinear activation function.

[Cybenko-Hornik-Funahashi Theorem]

The following form can approximate any continuous function F on $[0, 1]^n$ to any degree of accuracy: $F(\mathbf{x}) = \sum c_k \sigma(\sum_i^n w_{ik} x_i + b_k)$.

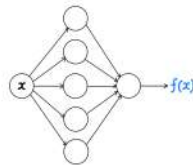
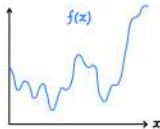


- Approximation can be improved by increasing the number of hidden nodes.

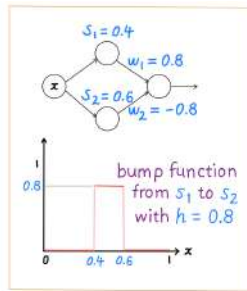
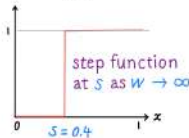
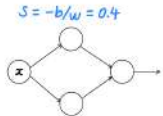
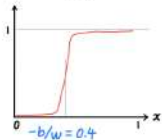
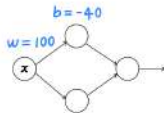
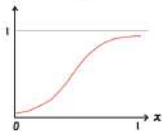
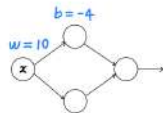
* $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function for nonlinearity.

Visual intuition

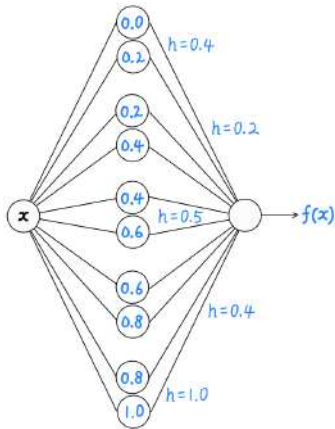
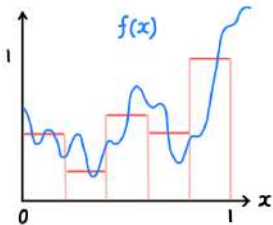
- For any function $f(x)$, there is a neural network satisfying that for any input x , $f(x)$ is an output of the network with very close approximation.



- The following hidden node computes $\sigma(wx + b)$ to create a step function.

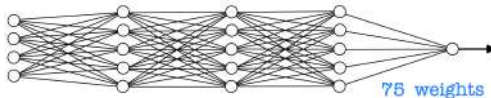
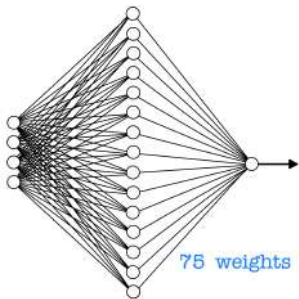


- To improve the approximation, we can increase the number of hidden nodes.



Shallow vs Deep networks

- To achieve high accuracy, the number of parameters must be huge (more than 10^6), which makes training too slow.
- Deeper networks generally perform better than shallow networks with a similar number of connections (weight parameters).

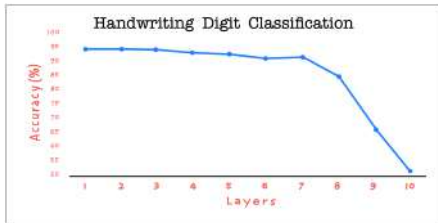


Hard to get the power of deeper networks

- Until 2006, deeper networks do not imply better because of the **vanishing gradient problem**.
- The gradient in front layers **decreases exponentially** with the number L of remaining layers, affected by multiplying L small numbers $\frac{d\hat{y}_i^n}{dz_i^n} = \hat{y}_i^n(1 - \hat{y}_i^n)$ for $0 < \hat{y}_i^n = \sigma(z_i) < 1$.

$$\frac{\partial C}{\partial w_{hi}} = \sum_n \frac{\partial z_i^n}{\partial w_{hi}} \boxed{\frac{d\hat{y}_i^n}{dz_i^n}} \sum_j w_{ij} \boxed{\frac{d\hat{y}_j^n}{dz_j^n}} \sum_k w_{jk} \boxed{\frac{d\hat{y}_k^n}{dz_k^n}} \frac{\partial C}{\partial \hat{y}_k^n}$$

- This causes that the **parameters w_{hi} are trained very slowly**.



Geoffrey Hinton's Summary of Findings up to today

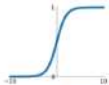
- Our labeled datasets were thousands of times too small.
⇒ Big Data, IoT
- Our computers were millions of times too slow.
⇒ GPU, TPU
- We initialized the weights in a stupid way.
⇒ RBM(2006), Xavier initialization(2010), He initialization(2015)
- We used the wrong type of nonlinearity.
⇒ ReLU(2010)

Activation functions

- A node calculates a weighted sum of inputs with a bias and decides whether it should be activated or not to next connections.
- A nonlinear activation function is added for this purpose.
- **Nonlinearity** is needed to learn complex representations of data, otherwise the network is just a linear function $W^L \cdots W^1 \mathbf{x} = W\mathbf{x}$.

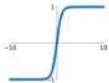
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



tanh

$$\tanh(x)$$



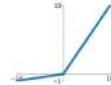
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

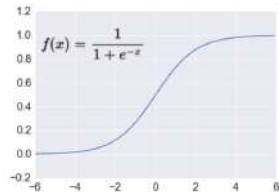
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

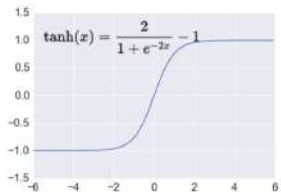
- most widely used activation function
- nonlinear and surrogating the step function
- ranging (0, 1), so not exploding
- differentiable and $\frac{d\hat{y}}{dz} = \hat{y}(1 - \hat{y})$ where $\hat{y} = \frac{1}{1+e^{-z}}$
- rising to the vanishing gradient problem



tanh

$$\tanh(z) = \frac{2}{1+e^{-2z}} - 1 = 2\sigma(2z) - 1$$

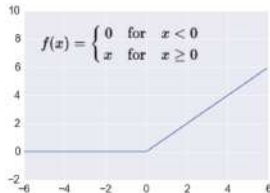
- scaled sigmoid function
- ranging (-1, 1) and zero-centered



ReLU (Rectified Linear Unit) by Hinton 2010

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

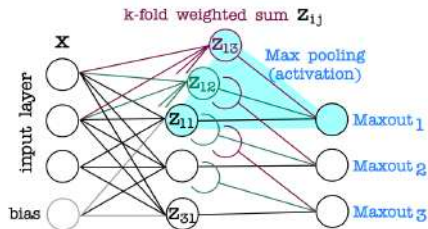
- most widely used after 2015
- overcoming the vanishing gradient problem
- ranging $[0, \infty)$, so possibly exploding
- fast computing: only comparison, addition and multiplication
- sparse activation \Rightarrow dying ReLU problem (\Rightarrow use Leaky ReLU(z) = $\max(0.1z, z)$)



Maxout by Goodfellow 2013

$$\text{Maxout}_i(\mathbf{x}) = \max_{j=1,\dots,k} z_{ij} \text{ with } z_{ij} = \mathbf{w}_{ij}^T \mathbf{x}$$

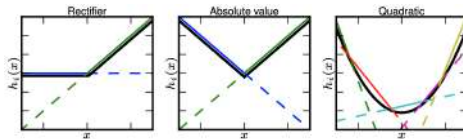
- Maxout is designed to both
 - facilitate optimization by **dropout**,
 - improve the accuracy of dropout's fast approximate model averaging technique.



- A single maxout unit makes a piecewise linear approximation to a convex function.

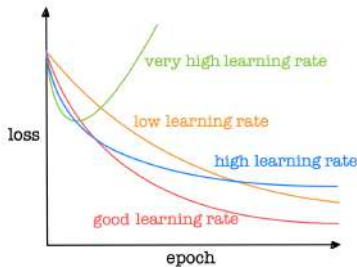
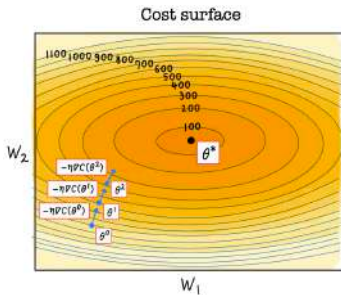
- * Graphical depiction of how the maxout activation can implement ReLU, absolute value rectifier and approximate the quadratic activation function.

This diagram shows for only a 1D input, but a maxout unit can approximate convex functions in multiple dimensions.



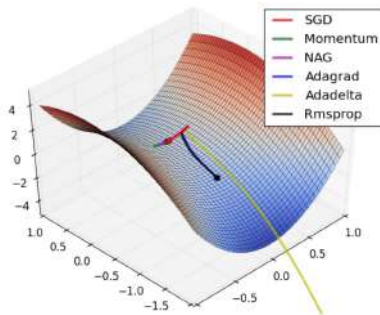
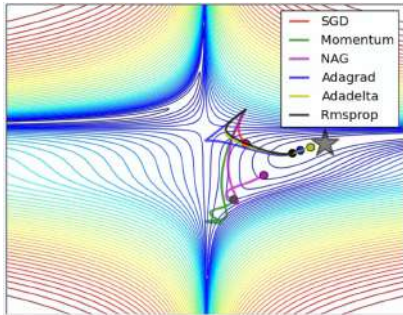
Learning rate

- In a gradient descent algorithm, the **learning rate** η is a tuning hyperparameter that determines the step size at each iteration (or minibatch) while moving toward a minimum of the cost function: $\theta^{t+1} = \theta^t - \eta \nabla C(\theta^t)$
- Learning rate is an important hyperparameter so that a value too small may result in a long training process, whereas a value too large may result in an unstable training process.
- Learning rate decay** is a technique in which a network starts training with a large learning rate and then slowly decreases it.



Gradient Descent Optimizers

- It optimizes (minimize or maximize) the cost/loss/objective function $C(\theta)$ that measures difference between network outputs (predictions) and target (real) values.



Source by Alec Radford

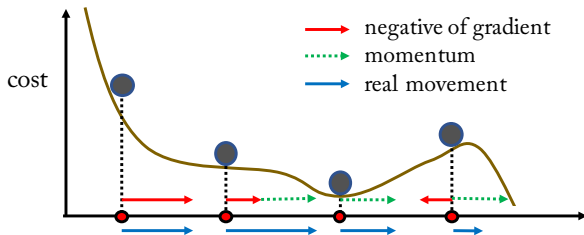
Momentum

- SGD often oscillates across the slopes of a ravine or local minima.
- Momentum** helps to accelerate SGD in the relevant direction v^t and reduces oscillations.

(momentum γ is usually 0.9)

$$v^{t+1} = \gamma v^t - \eta \nabla_{\theta} C(\theta^t)$$

$$\theta^{t+1} = \theta^t + v^{t+1}$$



AdaGrad (adaptive gradient) adapts different learning rates for every parameter and time step: low learning rates for parameters that have changed a lot so far (frequently occurring features) because they are more likely to be close to the optimum. ← good for dealing with sparse data

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{G_\theta^t + \epsilon}} \nabla_\theta C(\theta^t) \text{ where } G_\theta^t = \sum_{i=0}^t \left(\nabla_\theta C(\theta^i) \right)^2 \text{ and usually } \epsilon = 10^{-8}$$

RMSProp (root mean square propagation) resolves AdaGrad's rapidly diminishing learning rates and relative magnitude difference between parameters by recent gradients can be maintained.

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{G_\theta^t + \epsilon}} \nabla_\theta C(\theta^t) \text{ where } G_\theta^t = \gamma G_\theta^{t-1} + (1 - \gamma) \left(\nabla_\theta C(\theta^t) \right)^2 \text{ and usually } \gamma = 0.9$$

Adam (adaptive moment estimation) is an updated RMSProp combined with Momentum.

$$\hat{m}_\theta = \frac{m_\theta^{t+1}}{1 - (\beta_1)^{t+1}} \text{ where } m_\theta^{t+1} = \beta_1 m_\theta^t + (1 - \beta_1) \nabla_\theta C(\theta^t) \text{ and usually } \beta_1 = 0.9$$

$$\hat{G}_\theta = \frac{G_\theta^{t+1}}{1 - (\beta_2)^{t+1}} \text{ where } G_\theta^{t+1} = \beta_2 G_\theta^t + (1 - \beta_2) \left(\nabla_\theta C(\theta^t) \right)^2 \text{ and usually } \beta_2 = 0.999$$

$$\theta^{t+1} = \theta^t - \eta \frac{\hat{m}_\theta}{\sqrt{\hat{G}_\theta + \epsilon}}$$

Training and Test sets

Shuffle data

- First, shuffle the order of the dataset randomly.

Split data

- Split the shuffled data into two parts : a training set (70%) and a test set (30%).
- Split the shuffled data into three sets : a training set (60%), a validation set (20%) and a test set (20%).

Use the training set to train different models, the validation set to select a model (or select hyperparameters) and finally report performance on the test set.

- Split the shuffled data into k -fold Cross-Validation sets : take a group as a test set and the remaining $k-1$ groups as a training set.

Hyperparameters

When building a neural network, the programmer would choose the following hyperparameters and nonlinearity based on the application characteristics:

- number of hidden layers
- number of hidden nodes in each layer
- number of outputs
- number of epochs
- batch size
- type of output (linear, logistic, softmax)
- choice of nonlinearity at hidden layers and the output layer
- learning rate
- regularization coefficients

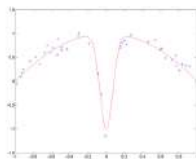
Bias-Variance Tradeoff

- **Bias-variance problem** is the conflict in trying to simultaneously minimize two sources of the prediction error: bias and variance.

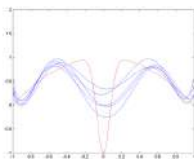
Their tradeoff is important not only for accuracy but also to avoid overfitting.

- **Bias**: difference between the average predictions of our model and the correct value.
Model with high bias **pays little attention to training data** and oversimplifies the model
⇒ **underfitting**. It always leads to high error on both training and test data.

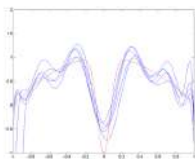
- **Variance**: variability of the model prediction for a data point.
Model with high variance **pays much attention to training data** (and noise) and fails to generalize to test data ⇒ **overfitting**. It performs well on training data but has high error on test data.



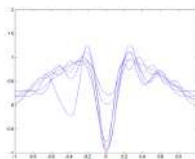
function
noisy data



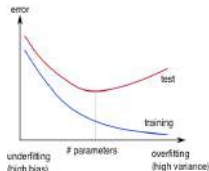
underfitting
high bias, low variance



good-fit



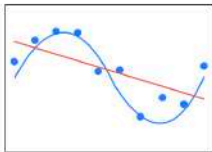
overfitting
low bias, high variance



Overfitting

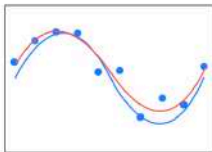
- **Overfitting** happens when a model learns **too closely to the detail and noise** in training data, and so fails to fit new data, negatively impacting the model's performance.

underfitting



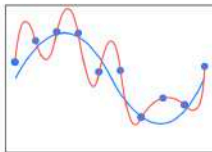
1st order poly.

good-fit

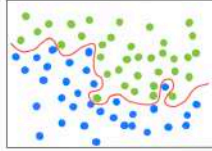
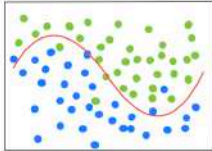
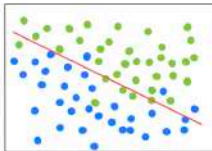
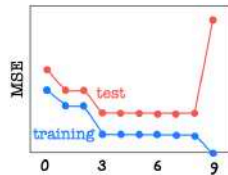


3rd order poly.

overfitting

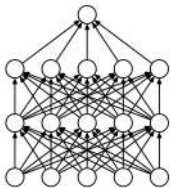


9th order poly.

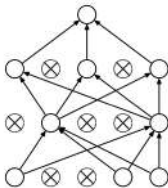


Dropout

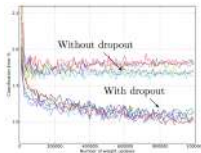
- **Dropout** is a regularization technique for **reducing overfitting** in neural networks by **preventing nodes from co-adapting too much on training data**, introduced by Hinton 2012.
 - * The gradient of each weight tells how it should change to reduce the cost, taking into account what all nodes are doing. So, nodes may change in a way that they fix up the mistakes of other nodes. Co-adaptation refers to when different hidden nodes have such highly correlated behavior. It is better to learn a general representation if nodes can detect features independently of each other.
- **Randomly drop hidden nodes** (along with their connections) in an iteration for each minibatch so that each node is omitted with probability p (independent of others) during only training. The omitted nodes and weights on this iteration are not updated during backpropagation.



(a) Standard Neural Net



(b) After applying dropout.



Source: Dropout: A Simple Way to Prevent Neural Networks from Overfitting by Hinton *et al*

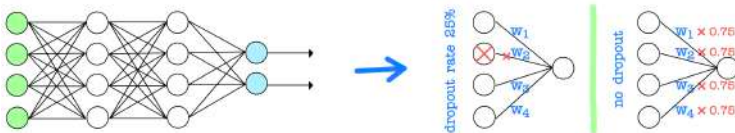
Training

- For a minibatch, each node has **dropout probability p** (i.e. keeping probability $1 - p$) so that we sample one from 2^N different thinned networks.
- Forward pass and backpropagation (gradient computing and weight update) are done only on this thinned network, ignoring the omitted nodes and their related weights.

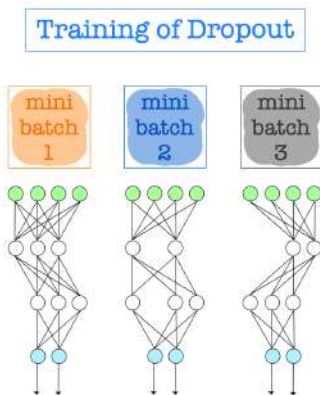
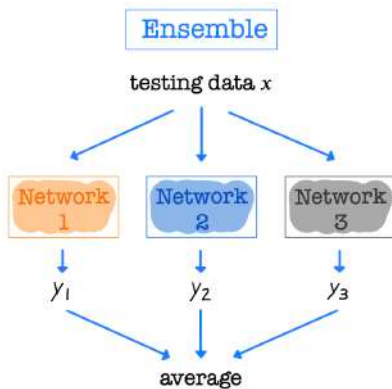


Test

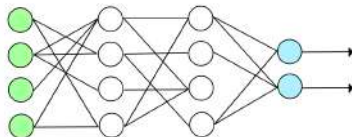
- For test data, we do **not apply dropout**, but take all the network weights multiplied by $1 - p$ so that this un-thinned network with smaller weights approximates the average of the outputs of all these thinned networks.



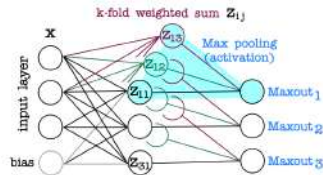
- Dropout is a kind of **ensemble** that trains many networks with different structures and combines them to produce a better output (averaging over an ensemble of networks).



- Dropout [Hinton 2012]
- Dropconnect [Wan 2013]: omitting connections instead of nodes
- Maxout [Goodfellow 2013]
- Annealed Dropout [Rennie 2014]: decreased dropout rate by epochs
- Gaussian Dropout [Srivastava 2014]
- Variational Dropout [Kingma 2015]
- Monte Carlo Dropout [Gal 2016]
- Dropout for CNNs
- Dropout for RNNs



Dropconnect



Maxout

Gaussian Dropout

- In **dropout**, the omission probability for each node follows *Bernoulli*(p) (i.e. $P(X=1)=p$).

Training: $\mathbf{y} = f(W\mathbf{x} \odot \mathbf{m})$, $m_i \sim \text{Bernoulli}(1-p)$ \odot element-wise multiplication

Test: $\mathbf{y} = f((1-p)W\mathbf{x})$

- In **Gaussian dropout**, instead of dropping nodes during training, the value of each node is **adjusted by Gaussian noise**.

Training: $\mathbf{y} = f(W\mathbf{x} \odot \mathbf{m})$, $m_i \sim N(1, p/(1-p))$

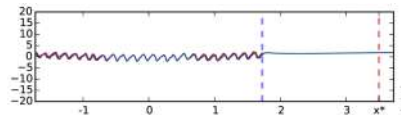
Test: $\mathbf{y} = f(W\mathbf{x})$

- All nodes and weights are exposed to each iteration for each minibatch, while, in dropout, the omitted nodes and weights on the iteration are not updated during backpropagation.

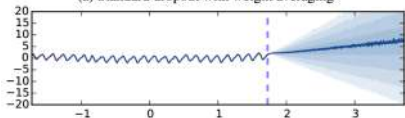
So, this avoids the slowdown of weight-updating during backpropagation and thereby increases the execution time.

Monte Carlo Dropout

- MC dropout applies dropout at both training and test time to generate random predictions (probability distribution) as its output, interpreted as Bayesian machine learning.
- At test time, predictions are no longer deterministic, but depending on which nodes are kept. So, given the same data point, the model can predict a different value each time.
- The variance of the output distribution indicates the model uncertainty for a particular input.
 - To derive the uncertainty for an input x , we collect many outputs from different thinned networks.
 - Their average and variance indicate an ensemble prediction and an uncertainty of the model regarding x .
 - In the figure, the model is trained on the training data (red : left of the dashed blue line), and tested on the entire dataset.
 - In (c), the blue is the predictive mean $\pm 2\sigma$ '95% confidence' (each blue shade represents $\frac{\sigma}{2}$ on the training and test sets).
 - For input x^* far away from the training data, standard dropout confidently predicts an insensible value; MC dropout predicts insensible values as well, but with the information indicating uncertainty about model predictions.
 - It works by simply applying dropout at test time without sacrificing either computational complexity or test accuracy.



(a) Standard dropout with weight averaging



(c) MC dropout with ReLU non-linearities

Regularization

- **Regularization** is a technique to avoid overfitting by limiting the absolute value of the parameters (weights) in the model.
- It assumes that smaller weights generate simpler model and thus reduces overfitting.
- Method: add a term to the cost, which imposes a penalty based on the magnitude of parameters.

L1 regularization (L1 norm, Lasso)

$$C_{L1} = \frac{1}{2N} \sum_{n=1}^N (\hat{y}^n - y^n)^2 + \frac{\lambda}{N} \sum_{i=1}^K |w_i|$$

L2 regularization (Weight decay, L2 norm, Ridge)

$$C_{L2} = \frac{1}{2N} \sum_{n=1}^N (\hat{y}^n - y^n)^2 + \frac{\lambda}{2N} \sum_{i=1}^K w_i^2$$

$$w_i(t+1) = w_i(t) - \eta \frac{\partial C_{L2}}{\partial w_i} = (1 - \frac{\eta \lambda}{N}) w_i(t) - \eta \frac{\partial C}{\partial w_i}$$

weight decay

Normalization and Standardization

- **Normalization** and **Standardization** are similar and both relate to the issue of **feature scaling** (to make training less sensitive to the scale of features of input data).
- If we train an algorithm using various scaled features of the **input variables**, then the results might be **dominated by features** with large magnitude.

Normalization (into [0, 1])

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$



Standardization (z-score)

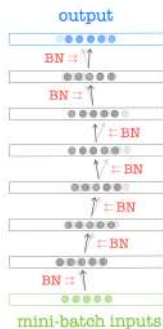
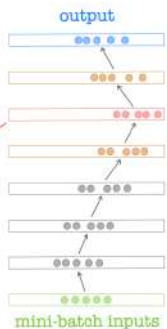
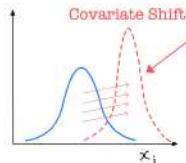
$$x'_i = \frac{x_i - \mu}{\sigma}$$



Batch Normalization (BN)

Internal Covariate Shift: distribution change of each layer's inputs during training due to changes in the parameters of the previous layer. (the more layers, the more amplification)

- Since the inputs to each layer are affected by the parameters of all preceding layers, small changes to the parameters amplify the input values of the following layers.
- This requires lower learning rates and careful weight initialization, which slow down training, and makes it notoriously hard to train models with saturating nonlinearities.



[mini-batch]
normalize
scale + shift

$$f: \text{saturating} \iff \lim_{z \rightarrow \pm\infty} |f(z)| < \infty$$

sigmoid: saturating

ReLU: non-saturating

Batch normalization (BN) reduces **Internal Covariate Shift** to train fast by allowing us to use higher learning rates, saturating nonlinearities, and less careful weight initialization.

- **BN transform** can be freely added to any subset of activations $\{x_i^{(k)}\}$ to be **normalized**.
- For each activation $x_i^{(k)}$, it adds two learnable parameters **scale** $\gamma^{(k)}$ and **shift** $\beta^{(k)}$.
- $y_i^{(k)} = \text{BN}_{\gamma, \beta}(x_i^{(k)})$ depends on $\{x_1^{(k)}, \dots, x_m^{(k)}\}$ in the **minibatch** to be normalized.

Input: Values of x over a minibatch: $\mathcal{B} = \{x_1, \dots, x_m\}$;
Parameters to be learned: γ, β

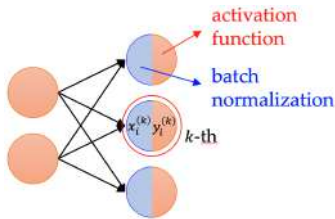
Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$ // $x_i = x_i^{(k)}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{minibatch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{minibatch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$$



Input: Network N with trainable parameters θ ;
subset of activations $\{x^{(k)}\}_{k=1}^K$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$
 $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network

for $k = 1, K$ **do**
Add BN transform $y^{(k)} = \text{BN}_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$
Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$
to take $y^{(k)}$ instead

end for

Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^K$
 $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with
fixed parameters

for $k = 1, K$ **do**
// For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
Process multiple training minibatches \mathcal{B} ,
each of size m , and average over them:
$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1} \text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma, \beta}(x)$ with
$$y = \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma \text{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right)$$

end for

Training and inference (test) in BN network

- Specify a subset of activations and add BN transform to each of them.
- Train using SGD with minibatch size m to optimize the parameters $\theta, \gamma^{(k)}, \beta^{(k)}$.
- During inference (test), use the normalization using population statistics (not minibatch) by using the **moving average** technique.
- Since the means and variances are fixed during inference, the normalization is simply a linear transform applied to each activation.
- BN takes the role of **Regularization**, and so in some cases we do not use **Dropout**.

Moving average technique

- Fix an activation x .
 $\{\mathcal{B}_t\}_{t=1}^T$ is the collection of minibatches corresponding to x .
 $\mu_{\mathcal{B}_t}$ and $\sigma_{\mathcal{B}_t}^2$ are the sample mean and variance of \mathcal{B}_t .
- If training is done with the minibatches $\{\mathcal{B}_t\}_{t=1}^T$,
then $E[x]_{(T)} = \frac{1}{T} \sum_{t=1}^T \mu_{\mathcal{B}_t}$ will be used during inference since there is no minibatch concept.
- If we train once more with a minibatch \mathcal{B}_{T+1} , use $E[x]_{(T+1)} = \frac{1}{T+1} \sum_{t=1}^{T+1} \mu_{\mathcal{B}_t}$ during inference,
or $E[x]_{(T+1)} = \frac{1}{T+1} \sum_{t=1}^{T+1} \mu_{\mathcal{B}_t} = \frac{1}{T+1} \left(\sum_{t=1}^T \mu_{\mathcal{B}_t} + \mu_{\mathcal{B}_{T+1}} \right) = \frac{T}{T+1} E[x]_{(T)} + \frac{1}{T+1} \mu_{\mathcal{B}_{T+1}}$ **moving average**
similarly $\text{Var}[x]_{(T+1)} = \frac{m}{m-1} \left(\frac{1}{T+1} \sum_{t=1}^{T+1} \sigma_{\mathcal{B}_t}^2 \right) = \frac{T}{T+1} \text{Var}[x]_{(T)} + \frac{m}{m-1} \cdot \frac{1}{T+1} \sigma_{\mathcal{B}_{T+1}}^2$
- Practically, in Tensorflow or Keras, use a **momentum** δ (default 0.99) so that

$$E[x]_{(T+1)} = \delta \cdot E[x]_{(T)} + (1 - \delta) \cdot \mu_{\mathcal{B}_{T+1}}$$

$$\text{Var}[x]_{(T+1)} = \delta \cdot \text{Var}[x]_{(T)} + (1 - \delta) \cdot \sigma_{\mathcal{B}_{T+1}}^2$$
- Therefore, for each node, we use two learnable scale parameter γ and shift parameter β ,
and two non-learnable parameters: moving mean $E[x]_{(T)}$ and moving variance $\text{Var}[x]_{(T)}$.

Weight Initialization

Starting point of running a neural network is to **initialize weight parameters correctly**.

In general practice, biases are initialized with 0.

Consider two typical scenarios that can cause issues while training the model.

- Zero Initialization** (or Same Initialization)

Regardless of input, all nodes h_1, \dots, h_n in a given hidden layer are computed in the same way, resulting $h_1 = \dots = h_n$, and so all the gradients with respect to w_{ij} 's will be the same.

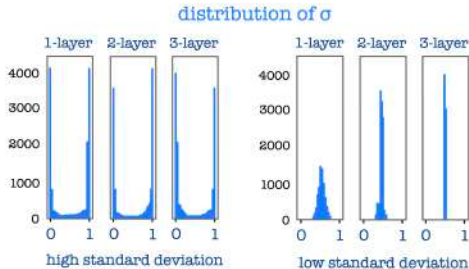
Therefore, all weights w_{ij} have the same value in subsequent iterations (after weight update).

After all, the model is the same as a linear model.

- Random Initialization**

If the standard deviation is high like 1, then $|W\mathbf{x}+b|$ becomes very higher, so $\sigma(W\mathbf{x}+b)$ tends to 1 or 0, causing vanishing gradient.

Or, if it is low like 0.01, then $W\mathbf{x}+b$ becomes near 0, so $\sigma(W\mathbf{x}+b)$ is hanging around 0.5, causing lack of diversity of nodes' role.



- LeCun Normal Initialization

$$W \sim N\left(0, \sqrt{\frac{1}{n_{\text{in}}}}\right)^2$$

- Xavier Normal Initialization

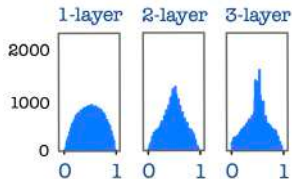
$$W \sim N\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right)^2$$

- He Normal Initialization

$$W \sim N\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right)^2$$

- n_{in} node number of the previous layer, and n_{out} of the next layer.

* Xavier Initialization for sigmoid



LeCun Uniform Initialization

$$W \sim U\left[-\sqrt{\frac{1}{n_{\text{in}}}}, \sqrt{\frac{1}{n_{\text{in}}}}\right]$$

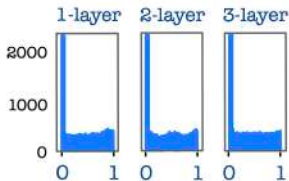
Xavier Uniform Initialization

$$W \sim U\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]$$

He Uniform Initialization

$$W \sim U\left[-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right]$$

He Initialization for ReLU



Xavier Normal Initialization

- $\mathbf{h}^l \in \mathbb{R}^{n_l}$: activation vector of layer l consisting of $n_l = n_{\text{in}}$ nodes
 $\mathbf{h}^{l+1} = f(W^l \mathbf{h}^l + \mathbf{b}^l) \in \mathbb{R}^{n_{l+1}}$ of layer $l+1$ consisting of $n_{l+1} = n_{\text{out}}$ nodes with initialization $\mathbf{b}^l = \mathbf{0}$
- Assumptions in each layer l
 - Linear Regime: activation function $f(x) = x$ for near $x=0$, and so $f'(0) = 1$.
 - All weights w_{ij}^l are independent and $\text{Var}[w_{ij}^l] = \text{Var}[w^l]$.
 - All elements of $\mathbf{h}^l = (h_1^l, \dots, h_{n_l}^l)^T$ are independent and $\text{Var}[h_i^l] = \text{Var}[h^l]$.
 - All weights and elements of an activation vector are independent of each other.
 - Each of their distributions is symmetric around 0, and so $E[w_{ij}^l] = E[h_i^l] = 0$
- We want to initialize weights satisfying following conditions:
 - $\text{Var}[h^l] = \text{Var}[h^{l+1}]$ ‘forward’
 - $\text{Var}\left[\frac{\partial \text{Cost}}{\partial h^l}\right] = \text{Var}\left[\frac{\partial \text{Cost}}{\partial h^{l+1}}\right]$ ‘backward’

- Forward Propagation satisfying $\text{Var}[h^l] = \text{Var}[h^{l+1}]$

$$h_j^{l+1} = \sum_{i=1}^{n_l} w_{ij}^l h_i^l \Rightarrow \text{Var}[h^{l+1}] = n_l \text{Var}[w^l h^l] = n_l \text{Var}[w^l] \text{Var}[h^l] \Rightarrow n_l \text{Var}[w^l] = 1$$

Note that $\text{Var}[XY] = E[X]^2 \text{Var}[Y] + E[Y]^2 \text{Var}[X] + \text{Var}[X] \text{Var}[Y]$ and $E[w_{ij}^l] = E[h_i^l] = 0$.

- Backward Propagation satisfying $\text{Var}\left[\frac{\partial \text{Cost}}{\partial h^l}\right] = \text{Var}\left[\frac{\partial \text{Cost}}{\partial h^{l+1}}\right]$

$$\frac{\partial \text{Cost}}{\partial h_i^l} = \sum_{j=1}^{n_{l+1}} w_{ij}^l f'(\cdot) \frac{\partial \text{Cost}}{\partial h_j^{l+1}} \Rightarrow \text{Var}\left[\frac{\partial \text{Cost}}{\partial h^l}\right] = n_{l+1} \text{Var}[w^l] \text{Var}\left[\frac{\partial \text{Cost}}{\partial h^{l+1}}\right] \Rightarrow n_{l+1} \text{Var}[w^l] = 1$$

Note that $f'(0) = 1$ and $E[w_{ij}^l] = E\left[\frac{\partial \text{Cost}}{\partial h_j^{l+1}}\right] = 0$.

- Thus, in each layer l , we initialize weights w_{ij}^l satisfying $\text{Var}[w_{ij}^l] = \frac{2}{n_l + n_{l+1}} = \frac{2}{n_{\text{in}} + n_{\text{out}}}$.

Xavier Uniform Initialization

- If $X \sim U[a, b]$, then $\text{Var}[X] = \frac{1}{12}(b - a)^2$.

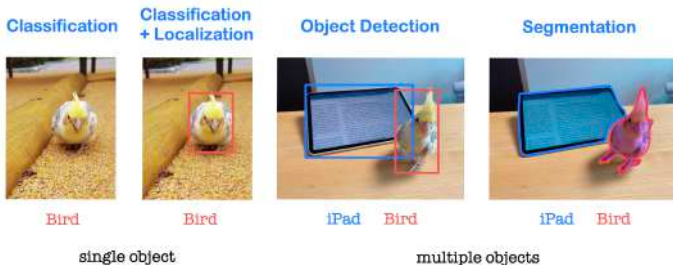
If $E[X] = 0$ (i.e., $a = -b$), then $\text{Var}[X] = \frac{1}{12}(b - (-b))^2 = \frac{1}{3}b^2$.

Therefore, $b = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}$.

Convolutional Neural Network

Convolutional Neural Network (CNN)

- CNN has great success in computer vision applications, especially in **image classification**, based on its **shared weights** using **Convolution** and **translation invariance** using **Max pooling**.
- Weight sharing makes it possible to handle input images of large sizes (huge input dimension). Translation invariance is an important property that is essential when dealing with image data.
- Applications are image recognition, image classification, object detection and segmentation, and also CNN combined with LSTM is able to give image captioning.



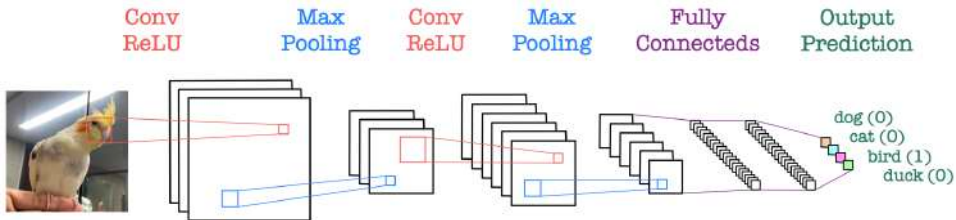
“A girl rides a bicycle.”

Difficulties while using MLP to handle image data

- While using MLP to handle images with size $W \times H$, the images are serialized (flattened) into vectors with very high dimension WH as inputs of the MLP network.
1. When the image becomes high-resolution, the serialized input data size increases rapidly, and so the number of parameters increases rapidly.
 - * If the image size is 1000×1000 , then the serialized input size is 10^6 , and so the number of parameters in the first layer is 10^{10} when the first hidden layer has 10000 nodes.
 2. Due to the characteristics of the image, a specific pixel is spatially related to the surrounding pixels (locality), and this locality or spatial information is lost while serialization is performed to the image. (Weighted sum loses location information.)
 3. For the new images obtained by small translations, transformations and scaling of an image, MLP considers them as totally different input data (no relation between them).
This causes huge amount of training time for input images obtained by data augmentations (crop, shift, noise, rotate, scaling, etc).

CNN architecture

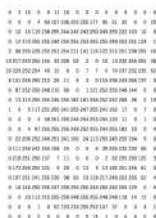
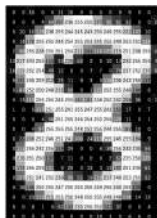
- LeCun creates the first CNN **LeNet** for character recognitions at 1998.
- **AlexNet** which is ILSVRC 2012 winner has similar structure as LeNet.



- * **Convolution** (shared weights, dimension reduction)
- * **Nonlinearity** (ReLU)
- * **Max pooling** (translation invariance, dimension reduction for FC layer)
- * **Fully connected layer** (classification)

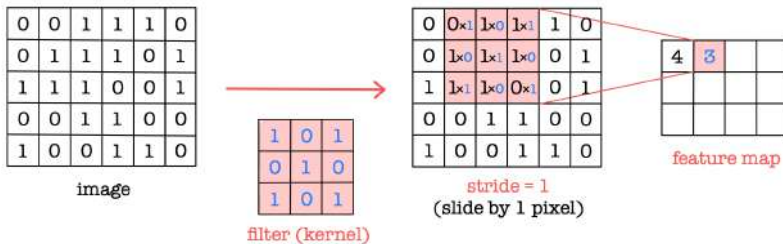
Input image

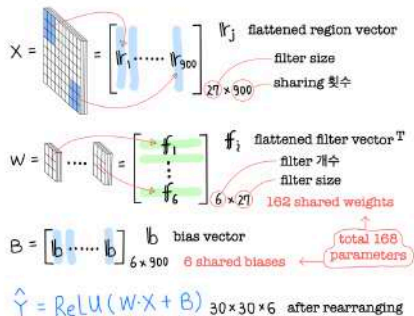
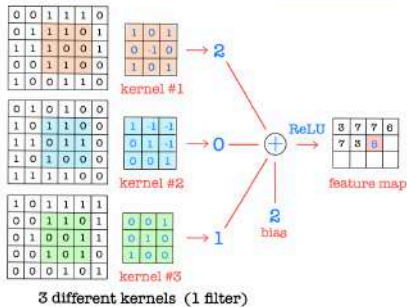
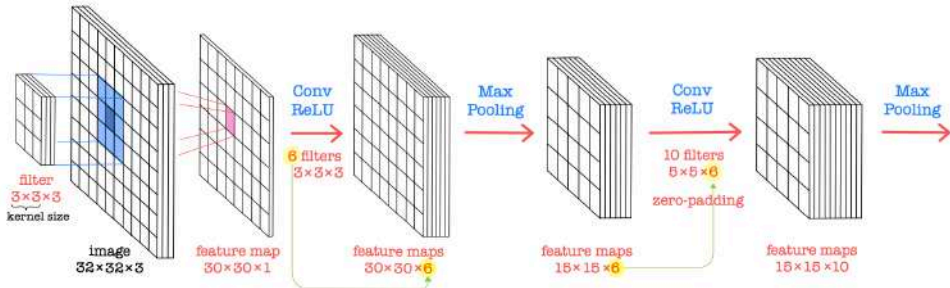
- Every input image can be represented as a matrix of pixel values.
- A color image has three RGB channels which are three 2d-matrices stacked together, each having pixel values in the range 0 to 255 for example.
- A grayscale image has one channel which is a single 2d-matrix each having pixel values in the range 0 (black) to 255 (white).



Convolution

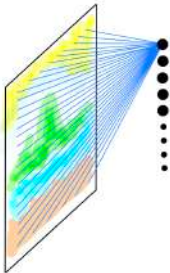
- **Convolution layer** as the core of CNN performs **feature extraction by using filters** while preserving spatial relationships between neighboring pixels (depending on kernel sizes).
- Convolution: multiplying elementwise by filters and summing the multiplication outputs.
- Ex) 3×3 **kernel** or $3 \times 3 \times 1$ **filter** (9 **shared weights** sharing 12 times) acts on a 5×6 input image and outputs a 3×4 **feature map** in case of 1 input channel and 1 output feature map.





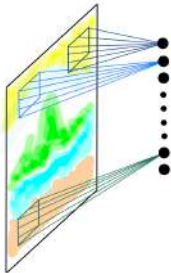
Why Convolution?

Fully Connected



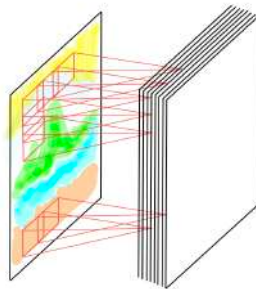
1000 × 1000 image
10,000 hidden nodes
 10^{10} parameters

Locally Connected



1000 × 1000 image
10 × 10 local window size
10 hidden nodes per window
 10^7 parameters

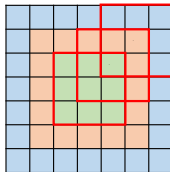
Convolution



1000 × 1000 image
10 × 10 kernel size
100 filters
 10^4 parameters
Regardless of image size!

How to stack Convolution layers?

- Three 3×3 Convolution layers give similar representational power as a single 7×7 Convolution layer.



receptive field

- Compare convolutions on input channels of size $H \times W \times F$ (height \times width \times |channel|) with F filters, stride 1 and zero-padding, producing output feature maps of size $H \times W \times F$.

One Conv with 7×7 kernel

number of weights

$$= (7 \times 7 \times F) \times F = 49FF$$

number of multiple-adds

$$\begin{aligned} &= (7 \times 7 \times F) \times (H \times W \times F) \\ &= 49HWFF \end{aligned}$$

Three Conv with 3×3 kernels

number of weights

$$= (3 \times 3 \times F) \times F \times 3 = 27FF$$

number of multiple-adds

$$\begin{aligned} &= (3 \times 3 \times F) \times (H \times W \times F) \times 3 \\ &= 27HWFF \end{aligned}$$

more nonlinearity

fewer parameters

less computing

Filter matrix

- Different values of the filter matrix produce different feature maps for the same input image.



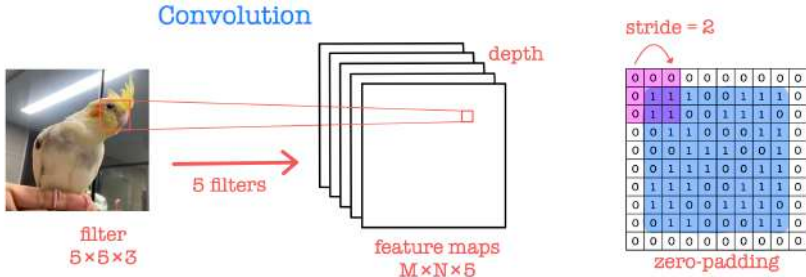
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

- CNN learns the values of filters during training although we specify the number of filters and the filter size before training.
- The more filters, the more features are extracted and the better the network will recognize patterns in unseen images (test data).

4 hyperparameters on Convolution layer

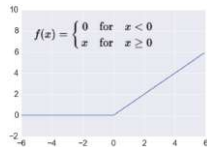
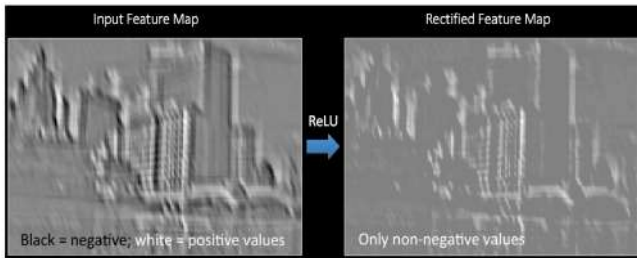
Size of a feature map is controlled by the following parameters that we specify.

- **filters**: number of filters (output feature maps)
- **kernel_size**: size of kernel (convolution window: height \times width)
- **strides**: distance between two successive kernel positions
- **padding** = 'valid' (no padding) or 'same' (padding with zeros to make the output with the same size as the input)



ReLU (nonlinearity)

- ReLU for **nonlinearity** has been used after every convolution operation.
- It is an **elementwise operation** (applied per pixel) and replaces all negative pixel values in the feature map by zero.



Max pooling

- Max pooling is for **dimension reduction** (downsampling) and **translation invariance**.

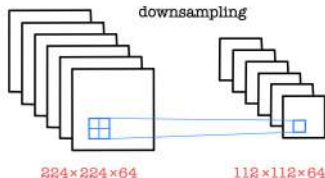
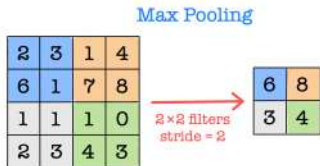
There is no learnable parameter in any of the pooling layers.

- It reduces the dimension of each feature map by taking the largest element among every 2×2 window, but retains the most important information.

Pooling is applied separately on each feature map.

- Max pooling makes the feature dimension smaller and more manageable (eventually for FC layer).
- It introduces a translation invariance to small shifts, distortions and scaling in the input image.
- This is very powerful since it can detect objects in an image no matter where they are located.

Otherwise, we need huge amount of time training much more images obtained by data augmentation.

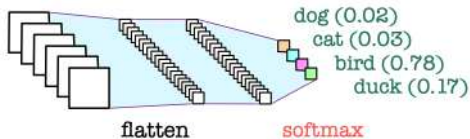


Fully connected layer

- **Fully connected layer** is a traditional MLP that uses a **softmax** activation function in the output layer.

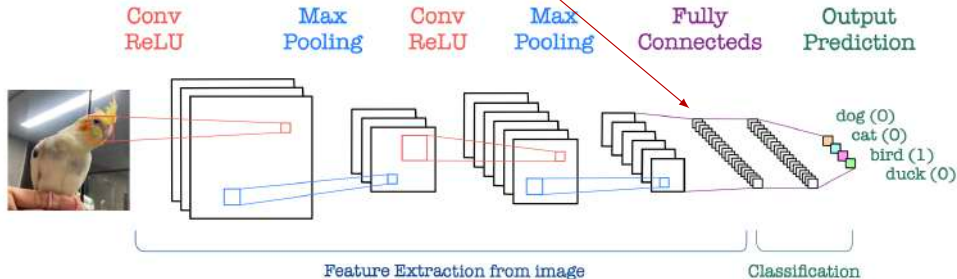
Fully connected: each node in the previous layer is connected to every node in the next layer.

- The output from Convolutional and Max pooling layers represents **high-quality features** of the input image.
- The number of nodes in FC layer is much smaller than the input dimension and so manageable to compute.
- FC layer uses these high-quality features for **classifying the input image** into classes.

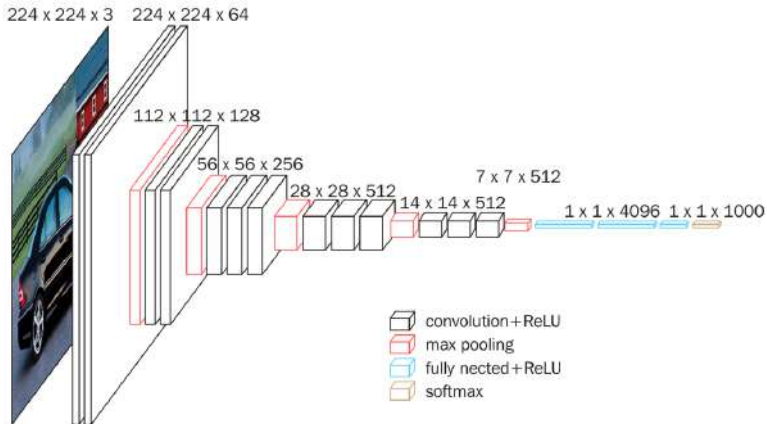


CNN architecture

- Convolution, ReLU and Max pooling layers are the basic block of CNN.
- All together extract **useful features** from inputs, introduce nonlinearity and reduce dimension, while making the features somewhat invariant to shifts, distortions and scaling.
- Some of the best performing CNN have tens of Convolution and Max pooling layers. It is not necessary to have a Max pooling layer after every Convolution layer.
- Fully connected layer acts as a classifier.



VGG-16 model from Vision Geometry Group (University of Oxford)



Training by using Backpropagation

- Step 0: Fix hyperparameters like number of filters and kernel sizes.
- Step 1: Initialize all Convolution filter matrices values (weights) and FC parameter values (connection weights) randomly.
- Step 2: Take a minibatch of training images as input, go through the forward pass (repeating Convolution, ReLU and Max pooling layers, and lastly FC layers) and find the output probabilities (softmax) for each class.
- Step 3: Calculate the total cost at the output layer;

$$\text{Total Cost} = \frac{1}{2} \sum (\text{target one-hot code} - \text{output probability})^2$$
- Step 4: Use Backpropagation to calculate the gradient of the total cost w.r.t. every weight and use gradient descent to update all weights to minimize the total cost.
- Step 5: Repeat Steps 2~4 with all images in the training set.

CNN Keras code

```
In [1]: from keras.models import Sequential
        # from keras.layers.convolutional import Conv2D, MaxPooling2D
        from keras.layers import Convolution2D, MaxPooling2D
        from keras.layers import Activation, Dropout, Flatten, Dense

In [2]: nfilter = 36
        nb_classes = 10

In [3]: model = Sequential()

        # input: 32x32 images with 3 channels -> (32, 32, 3) tensors.

        model.add(Convolution2D(nfilter, (3, 3), padding="same", input_shape = x_train.shape[1:]))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Convolution2D(2*nfilter, (3, 3), padding="same"))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Convolution2D(4*nfilter, (3, 3)))
        model.add(Activation('relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Dropout(0.25))

        model.add(Flatten())

        model.add(Dense(16*nfilter))
        model.add(Activation('relu'))
        model.add(Dropout(0.5))

        model.add(Dense(nb_classes))
        model.add(Activation('softmax'))

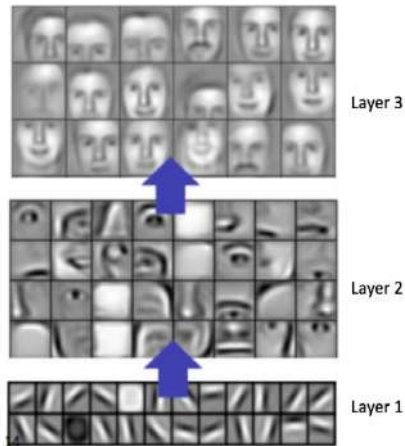
        print(model.input_shape)
        print(model.output_shape)
```

Visualizing CNN

- The more convolution layers we have, the more complicated features the network will be able to learn to recognize.
- In lower layers, CNN learns to detect patterns like small edges from raw pixels.

Then, in middle layers, it uses these patterns to detect simple shapes like eyes.

After all, in higher layers, it uses these shapes to detect high-quality features like faces.



ImageNet Large Scale Visual Recognition Challenge (ILSVRC 2010~2017)

- Image classification:
 - list of object categories (top-5, top-1)

Dataset Places2:

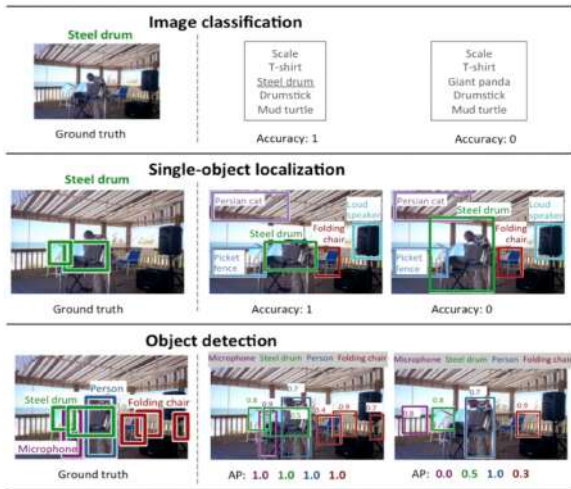
- 10M images with 400 categories.
(hand-annotated labels)

- Single-object localization:
 - image classification
 - bounding box around the object

Dataset ImageNet:

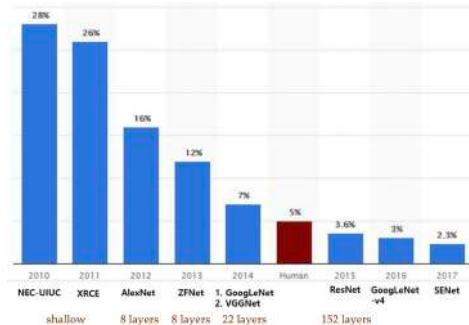
- 1.2M images with 1000 categories

- Object detection:
 - multiple image classification
(among 200 categories)
 - bounding box around each object



CNNs winning ILSVRC

- **LeNet** (Yann LeCun 1998)
- **AlexNet** (Alex Krizhevsky and Geoffrey Hinton 2012 15.3% / Top-5 error rate)
- **ZF Net** (Matthew Zeiler and Rob Fergus 2013 11.2%)
- **GoogLeNet** (Google 2014 6.7%) developing 9 Inception modules that dramatically reduced the number of parameters.
- **VGG** (Vision Geometry Group 2014 7.3%) using a simple architecture.
- **ResNet** (Kaiming He at Microsoft 2015 3.57%) using Residual blocks to build 152 layers.
- **Inception-v4** (Google 2016 3.1%)
- **SENet** (2017 2.25%) using Squeeze-and-Excitation blocks.



IMAGENET Large Scale Visual Recognition Challenge (ILSVRC)

Team Name	Error (%)
GoogLeNet	6.7
VGG	7.3
MSRA Visual computing	8.1
Andrew Howard	8.1
DeeperVision	9.5
NUS-BST	9.8
TTIC_ECP – Epitomic Vision	10.2
XYZ	11.2
BDC-I2R-UPMC	11.3

BREIL_KAIST, Brno University of Technology, Cldi-KAIST,
 DeepCNet, Fengjun Lv, libccv, MIL, Orange-BUPT,
 PassBy, SCUT_GLH, SYSU_Vision, UI, UvA-Euvision

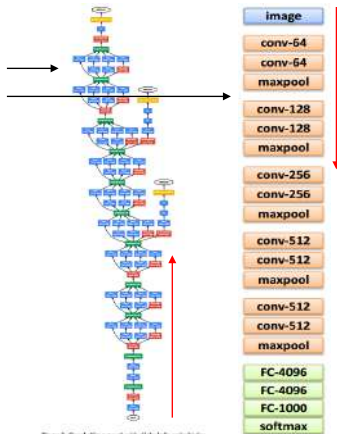
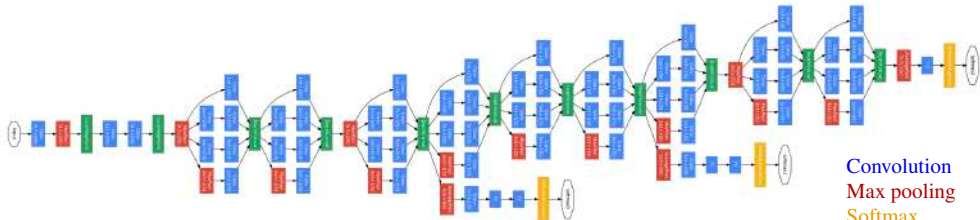


Figure 3: GoogLeNet network with all the fully and validation

22 layers

19 layers

GoogLeNet (Inception-v1)



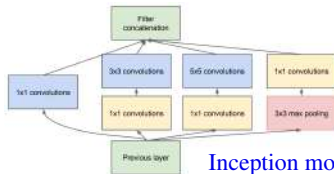
Convolution
Max pooling
Softmax
Other



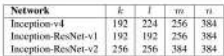
(a) Siberian husky



(b) Husky dog



Inception module



1×1 filters

- It increases nonlinearity without affecting receptive field.
- It is indeed Fully connected with weight sharing per feature map.
- It is useful when we want to change the number of feature maps with minimal alteration.

Single 3×3 Conv

$$H \times W \times F$$

↓ Conv 3×3, F filters

$$H \times W \times F$$

$9F^2$ parameters

1×3, 3×1 Conv

$$H \times W \times F$$

↓ Conv 1×3, F filters

$$H \times W \times F$$

↓ Conv 3×1, F filters

$$H \times W \times F$$

$6F^2$ parameters
more nonlinearity

Bottleneck

$$H \times W \times F$$

↓ Conv 1×1, $F/2$ filters

$$H \times W \times (F/2)$$

↓ Conv 3×3, $F/2$ filters

$$H \times W \times (F/2)$$

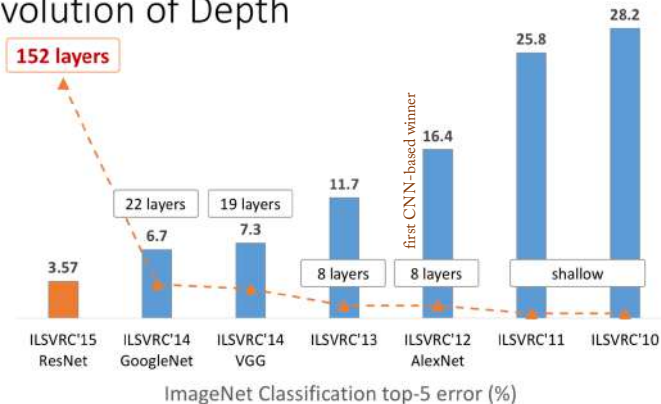
↓ Conv 1×1, F filters

$$H \times W \times F$$

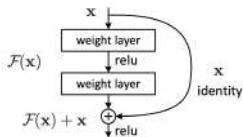
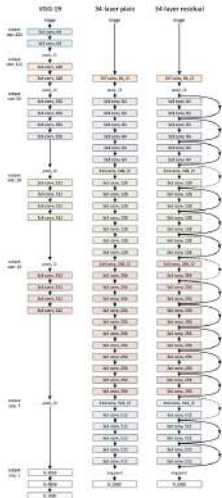
$3.25F^2$ parameters
more nonlinearity

Deeper neural networks are more difficult to train due to the vanishing gradient problem!

Revolution of Depth



ResNet (Residual Network)



Residual module

Residual $F(x)$ + identity shortcut x

- Simple design, but just deeper
- Few Max pooling
- No hidden FC, dropout

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2.x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3.x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4.x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5.x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				

Importance of the identity mapping

- Very smooth forward pass

$$x_K = x_k + \sum_{i=k}^{K-1} F(x_i) \text{ since } x_K = x_{K-1} + F(x_{K-1}) = x_{K-2} + F(x_{K-2}) + F(x_{K-1}) = \dots$$

\Rightarrow Any x_k is directly forward pass to any x_K , plus residual.

\Rightarrow Any x_K is an additive outcome, in contrast to multiplicative $x_K = \prod_{i=k}^{K-1} W_i x_k$.

- Very smooth backpropagation

$$\frac{\partial C}{\partial x_k} = \frac{\partial C}{\partial x_K} \frac{\partial x_K}{\partial x_k} = \frac{\partial C}{\partial x_K} (1 + \frac{\partial}{\partial x_k} \sum_{i=k}^{K-1} F(x_i))$$

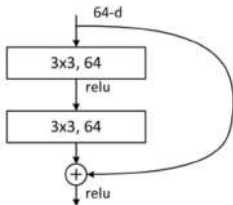
\Rightarrow Any $\frac{\partial C}{\partial x_K}$ is directly backpropagated to any $\frac{\partial C}{\partial x_i}$, plus residual.

\Rightarrow Any $\frac{\partial C}{\partial x_k}$ is additive, unlike to vanish.

- Deeper neural networks are more difficult to train due to the vanishing gradient problem.

This residual learning framework makes it easier to train much deeper networks by adding the identity mapping.

Deep residual modules



Naïve residual block

(ResNet-18/34)

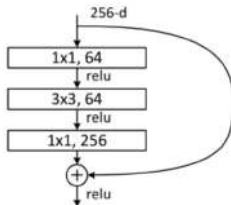
3×3 Conv

3×3 Conv

Number of weights: total $\sim 72K$

$(3 \times 3 \times 64) \times 64 \sim 36K$

$(3 \times 3 \times 64) \times 64 \sim 36K$



Bottleneck residual block

(ResNet-50/101/152)

1×1 Conv to reduce 256 to 64 feature maps

3×3 Conv

1×1 Conv back to 256 feature maps

Number of weights: total $\sim 68K$

$(1 \times 1 \times 256) \times 64 \sim 16K$

$(3 \times 3 \times 64) \times 64 \sim 36K$

$(1 \times 1 \times 64) \times 256 \sim 16K$

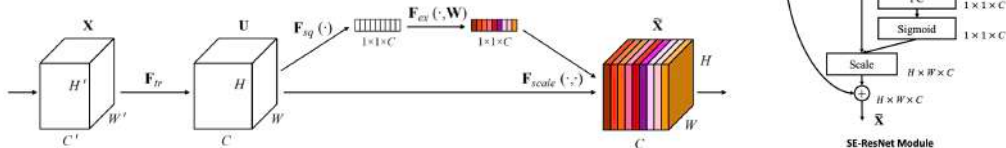
SENet (Squeeze-and-Excitation Network)

- Convolution outputs informative features by combining spatial and channel-wise information within **local receptive fields** (according to the filter size $K \times K \times F$) at each layer.
- SE block** squeezes **channel-wise global spatial information** to perform feature recalibration.
 - Squeeze**: channel-wise global average pooling

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W} \sum_{i,j} u_c(i,j) \text{ for each channel } u_c$$
 - Excitation**: fully capturing channel-wise dependencies

$$s = F_{ex}(z, W_1, W_2) = \sigma(W_2 \text{ReLU}(W_1 z)) \text{ two FCs}$$
 - Recalibration**: rescaling U with the activations s

$$\tilde{x}_c = F_{scale}(u_c, s_c) = s_c u_c$$

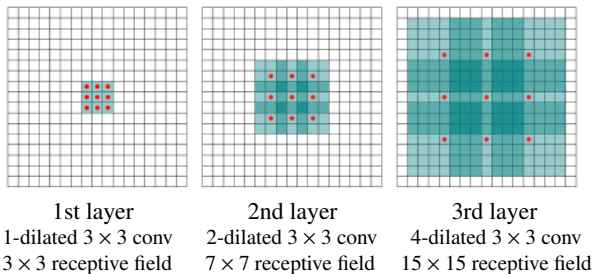


* SE blocks can be simply used as a drop-in replacement for the original block.

Dilated Convolution

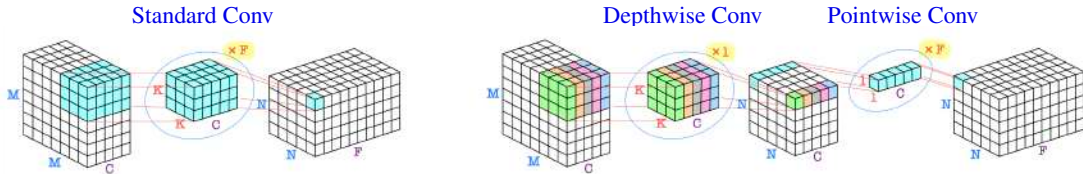
- CNN invariance to local image transformations is highly desirable for classification tasks, but can prevent dense prediction tasks such as semantic **image segmentation**, where abstraction of spatial information is undesired.
- **Dilated convolution** effectively **enlarges the receptive field** without using larger kernel size or more convolution layers.

It does not increase the number of parameters or lose resolution.



Depthwise Separable Convolution

- Depthwise separable convolution (Depthwise Conv + Pointwise Conv) is used to build a **light weight** CNN (fewer parameters and multiply-adds) for efficient on device intelligence.



- input channel size : $M \times M \times C$ ($7 \times 7 \times 5$)
- feature map size : $N \times N \times F$ ($5 \times 5 \times 10$)
- filter size : $K \times K \times C$ ($3 \times 3 \times 5$)
- |weights| = $KKCF$ (F sets)
- |multiplications| = $KKCNNF$

$M \times M \times C$

$N \times N \times C$

$K \times K \times C$

KKC (1 set)

$KKCNN$

$N \times N \times C$

$N \times N \times F$

$1 \times 1 \times C$

CF (F sets)

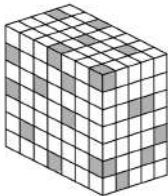
$CNNF$

* It saves the number of both parameters and computation cost by $\frac{1}{F} + \frac{1}{K^2}$.

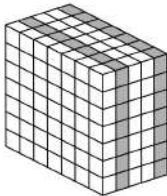
Dropout in Convolution layer

- **Standard Dropout**: If we randomly omit pixels then almost no information is removed as the omitted pixels are similar to their surroundings, so overfitting cannot be prevented.
- **Spatial Dropout** randomly drops out entire feature maps rather than individual pixels, bypassing dependency of adjacent pixels and promoting independence between feature maps.
- **Cutout** applies a random square mask directly over a larger region of each input image, unlike other common methods which apply dropout at the feature map level.
- **Max-pooling Dropout** applies dropout directly to Max pooling filter so that it minimizes the pooling of high activators.

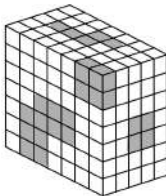
Standard Dropout



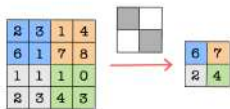
Spatial Dropout



Cutout



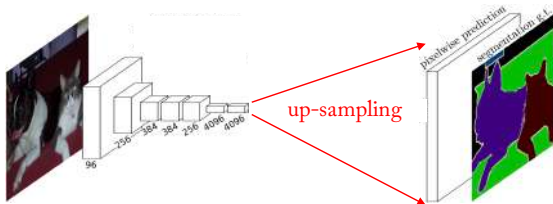
Max-pooling Dropout



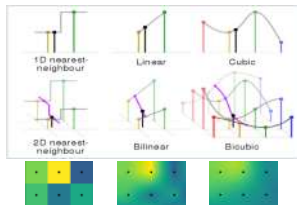
Up-sampling

Motivation: need a transformation going in the **opposite direction of convolutions**.

- Generating images involving up-sampling from low resolution to high resolution.
- Decoding layer of a convolutional auto-encoder.



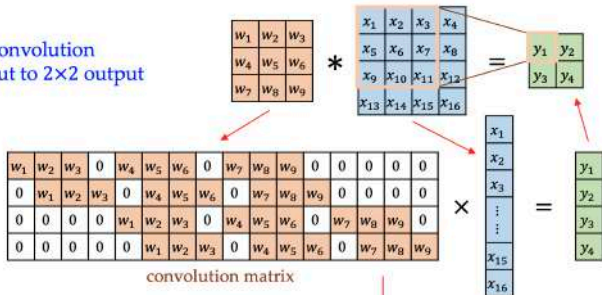
- **Non-learnable** interpolation methods (nearest neighbor, bi-linear, bi-cubic) which are like manual feature engineering.
- **Learnable** neural network up-samplings:
 - Transposed convolution
 - Fractionally-strided convolution



Transposed Convolution

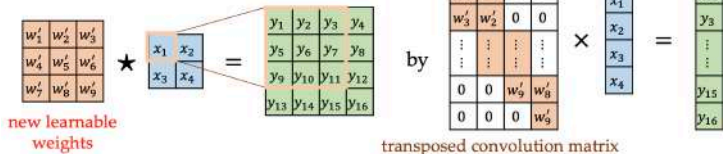
- Convolution has the **positional connectivity** between the input values and the output values.
Ex) The top left values in the input channel affect the top left values of the output feature map.
Furthermore, it forms a **many-to-one relationship** like $3 \times 3 \times F$ pixels (filter size) to 1 pixel.
- **Transposed convolution** is going backward of a convolution operation with the properties that it has the similar positional connectivity and forms a one-to-many relationship.
- We can express a convolution operation using a convolution matrix, which is just a rearranged matrix of weights to use a matrix multiplication to conduct convolutions.
- We similarly express a transposed convolution using a **transposed convolution matrix**, whose layout is the transposed shape of the original convolution matrix, but in which the actual weight values do not have to come from this convolution matrix.
The weights in the transposed convolution are **learnable**.

3×3 convolution
on 4×4 input to 2×2 output



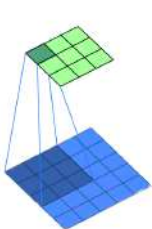
matrix of transposed shaped only

3×3 transposed convolution
on 2×2 input to 4×4 output

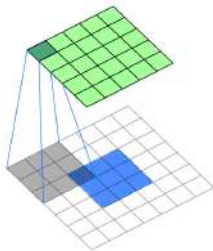


Fractionally-strided Convolution

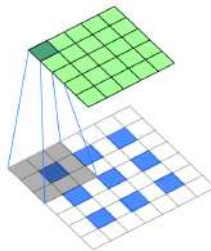
- Similar to the standard convolution argument, it takes a convolution after $n-1$ rows/columns between the input pixels (blue) are zero-padded.
- This makes the filter (kernel) move around at a slower pace on the input like $\frac{1}{n}$ strided.



3×3 Conv
with 1 stride
from 5×5 to 3×3



3×3 Conv
with big zero-padding
from 3×3 to 5×5



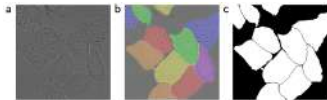
$\frac{1}{2}$ -strided 3×3 Conv
(fractionally-strided)
from 3×3 to 5×5

U-Net

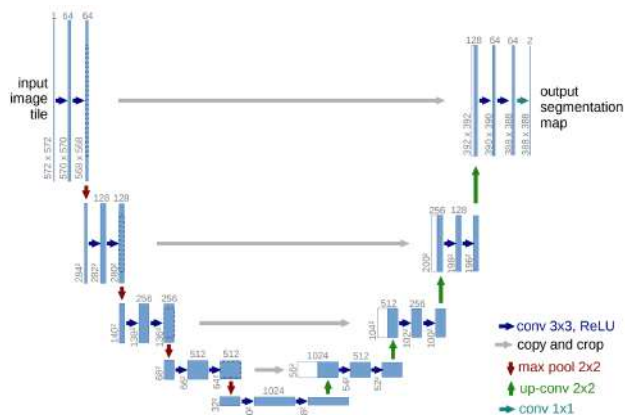
- U-Net is a U-shaped CNN used for **object segmentation**.

- Contracting path** captures the context.
- Expanding path** enables localization.

To localize during up-sampling, high resolution features from the contracting path are combined to propagate context information to higher resolution layers.



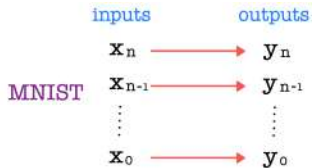
(a) input image of a cell
(b) manual ground truth segmentation
(c) generated output segmentation



Recurrent Neural Network

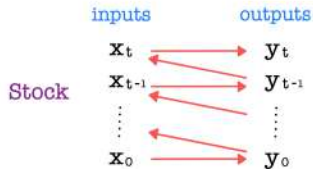
Feedforward neural network (FFNN) as MLP

- Information only flows in one direction.
- No sense of time or memory for previous data.



Recurrent neural network (RNN)

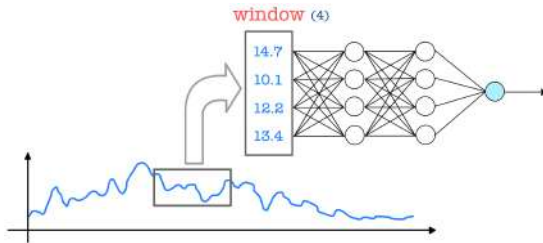
- Information flows in multi-direction.
- Sense of time and memory for previous data.



Time Delay Neural Network (TDNN)

- **TDNN** is a FFNN whose purpose is to classify **temporal patterns** with **shift-invariance** (independency on the beginning points to insure data i.i.d for MLP, so we may shuffle) in **sequential data** like 1D Conv with stride 1.
- It is the simplest way of controlling time-varying sequential data and allows conventional backpropagation algorithms.
- TDNN gets its inputs by sliding a **window** of size n across multiple time steps in sequential data and treating these n consecutive data samples as one input data.

$$\begin{aligned}\text{input}(t) &= [x_t, x_{t-1}, \dots, x_{t-n+1}] \\ \text{input}(t-1) &= [x_{t-1}, x_{t-2}, \dots, x_{t-n}] \\ \text{input}(t-2) &= [x_{t-2}, x_{t-3}, \dots, x_{t-n-1}] \\ &\vdots\end{aligned}$$



Disadvantages of TDNN

- The success of TDNN depends on finding an **appropriate window size**.
 - small window size does not capture the longer dependencies.
 - large window size increases the parameter number and may add unnecessary noise.
- TDNN works well for some short-memory problems such as Atari games, but cannot handle **long-memory problems** over hundreds of time steps such as stock prediction.
- Because TDNN has a fixed window size, it cannot handle **sequential data of variable length** such as language translation.
- FFNNs do not have any explicit memory of past data.

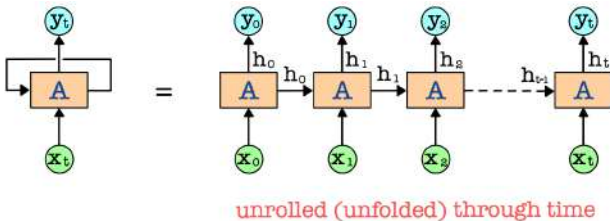
Their ability to capture temporal dependency is limited within the window size.

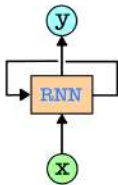
Even for a particular time window, input data is treated as a multidimensional feature vector rather than as a sequence of observations, so we lose the benefit of sequential information.

(Unaware of the temporal structure)

Recurrent Neural Network (RNN)

- RNN (LSTM) makes predictions based on **current and previous inputs recurrently**, while FFNN makes decisions based only on the current input.
- RNN processes the input sequence **one data x_t at a time** (current input) and maintains a **hidden state vector h_t** as a **memory for past information** (previous inputs).
- It learns to selectively retain relevant information to capture **temporal dependency or structure** across multiple time steps for processing sequential data.
- RNN does not require a fixed sized time window and can handle **variable length sequences**.





Basic
Form

one to many

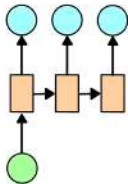
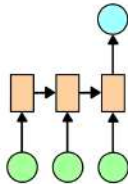


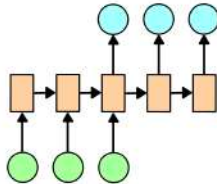
Image
Captioning

many to one



Sentiment
Classification

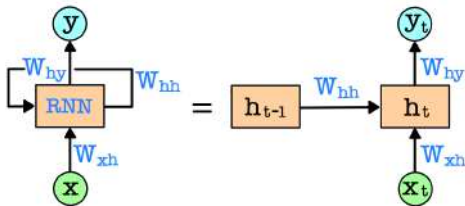
many to many



Machine
Translation

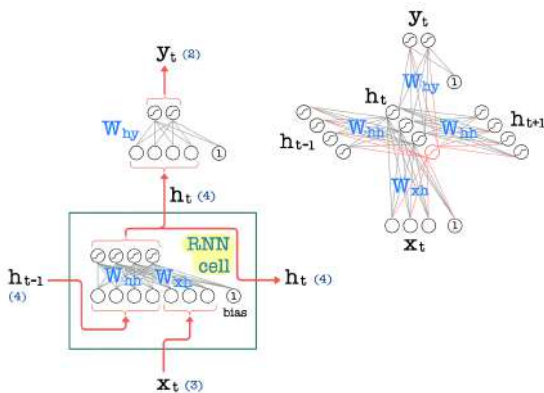
SimpleRNN

- Each time step t : x_t input, y_t output, h_{t-1} prior hidden state, h_t current hidden state.
- Hidden state vector h_t acts as a memory for past information in sequential data.
- Weight matrices W_{xh} , W_{hh} , W_{hy} are shared at every time step (without W_{hh} , it is a MLP).



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

$$y_t = W_{hy} h_t + b_y$$



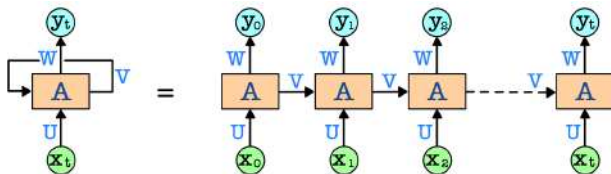
Backpropagation Through Time (BPTT)

- DNN uses backpropagation to update the weights in the way to minimize the cost.
- RNN uses a variation of backpropagation called **BPTT** to control temporal dependencies.
- How to update 3 weight matrices W_{xh} , W_{hh} , W_{hy} shared at every time step?

$U = W_{xh}$ input to hidden weights

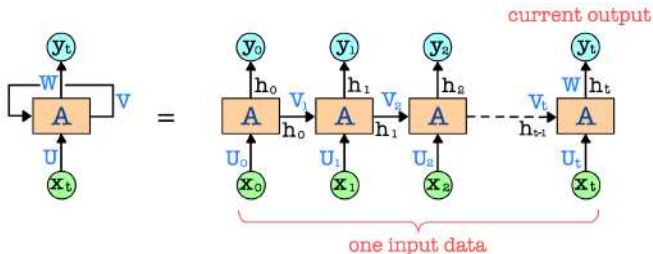
$V = W_{hh}$ hidden to hidden weights

$W = W_{hy}$ hidden to output weights



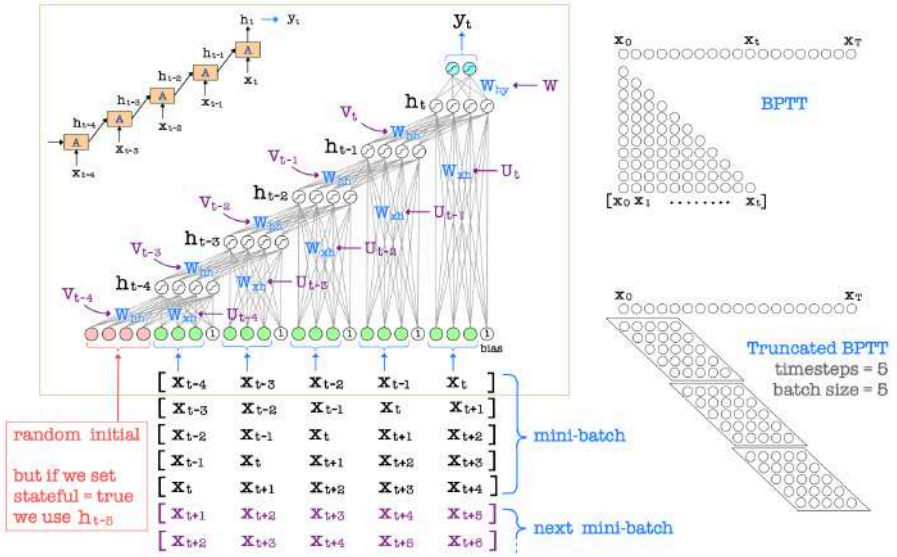
Weight update in BPTT

- Unfolded network is treated as **one big FFNN** which takes the current and whole previous inputs as one big input data, and shares weights $U = W_{xh}$, $V = W_{hh}$, $W = W_{hy}$ at every time step.
- In **forward pass**, we may think as the weights are not shared so that each has its own weights $U_0, U_1, \dots, U_t, V_1, \dots, V_t$ and W .
- In **backward pass**, compute the gradient with respect to all weights $U_0, U_1, \dots, U_t, V_1, \dots, V_t$ and W in the unfolded network as standard backpropagation.
- Final gradients of U and V are the average (or sum) of all gradients of U_i 's and V_i 's respectively, and applied to RNN weight updates.



Truncated BPTT

- BPTT can be slow to train RNN on problems with very long sequential data.
Furthermore, the accumulation of gradients over many timesteps can result in a shrinking of values to zero, or a growth of values that eventually overflow.
- Truncated BPTT limits the number of timesteps used on backward pass for backpropagation and estimates the gradient used to update the weights rather than calculate it fully.
- When using TBPTT, we choose the number of timesteps (lookback) as a hyperparameter to split up long input sequences into subsequences that are both long enough to capture relevant past information for making predictions and short enough to train efficiently.
- For time series, input data is divided into overlapping subsequences with same time interval. Each subsequence has lookback number of consecutive time steps and forms a training sample. During training, TBPTT is done over individual samples (subsequence) for lookback time steps.
- RNN input data: 3D tensor with shape (batch_size, timesteps, input_dim).
- Lookback values ranging up to 200 have been used successfully for many tasks.



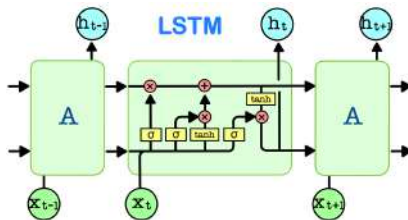
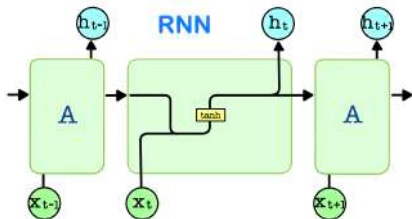
(batch_size, timesteps, input_dim)

State maintenance

- Within a batch, individual samples (subsequences) are still independent.
- **Keras** provides 2 ways of maintaining RNN hidden states (and additional cell states in LSTM).
 - **Default mode**: states are only maintained over each individual sample for lookback number of time steps starting with a random initial state.
 - **Stateful mode**: states are maintained along successive batches so that the final state of i th sample of the current batch is used as the initial state for i th sample of the next batch.
- To use **stateful mode**, the following must be assumed.
 - Batch size must divide the number of total samples because all batches have the same number of samples.
 - If B_t and B_{t+1} are successive batches, then $B_{t+1}[i]$ is the follow-up sequence to $B_t[i]$ for all i .
 - **No sample shuffling** is required to maintain state across batches: $B_t[i] \rightarrow B_{t+1}[i]$.

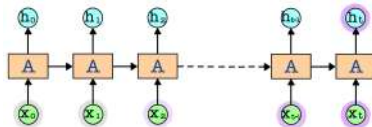
Long Short-Term Memory (LSTM)

LSTM is designed to capture **long-term dependency** and overcomes the **vanishing/exploding gradient problem** in SimpleRNN.

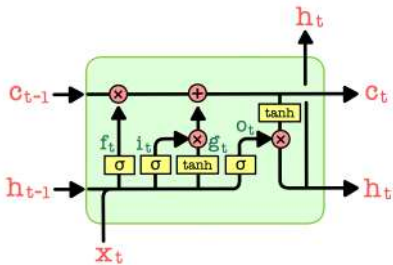


* Long-term dependency

If the gap between the relevant information and the point where it is needed becomes large, RNN learning ability will be decreased.



- **LSTM** uses a **cell state vector** c_t to remember past information over a long interval, in addition to a **hidden state vector** h_t of SimpleRNN.
- **Input**, **forget** and **output gates** regulate the flow of information into and out of the cell state.



$$i_t = \sigma(W_{xh}^i x_t + W_{hh}^i h_{t-1} + b_h^i) \quad \text{input gate}$$

$$f_t = \sigma(W_{xh}^f x_t + W_{hh}^f h_{t-1} + b_h^f) \quad \text{forget gate}$$

$$o_t = \sigma(W_{xh}^o x_t + W_{hh}^o h_{t-1} + b_h^o) \quad \text{output gate}$$

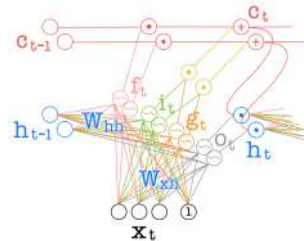
$$g_t = \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1} + b_h^g) \quad \Leftarrow \text{RNN core}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad \text{cell state}$$

$$h_t = o_t \odot \tanh(c_t) \quad \text{hidden state}$$

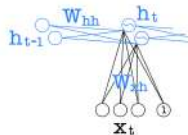
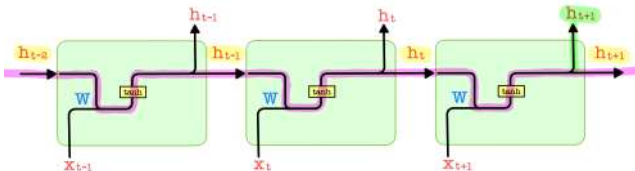
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} x_t \\ h_{t-1} \\ 1 \end{pmatrix}$$

- Input and forget gates determine how the next cell state is influenced by the input and the previous hidden state.
- Output gate determines how the hidden state and the output is influenced by the cell state.
- SimpleRNN is LSTM with $i_t = o_t = 1$, $f_t = 0$, since $h_t = \tanh(\tanh(W_{xh} x_t + W_{hh} h_{t-1} + b_h))$.

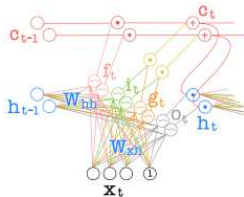
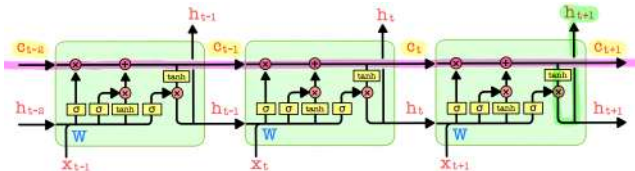


Backpropagation

- **SimpleRNN**: computing gradient of h_t involves many factors of W (and tanh).
 \Rightarrow largest singular value < 1 (vanishing gradients) or > 1 (exploding gradients)

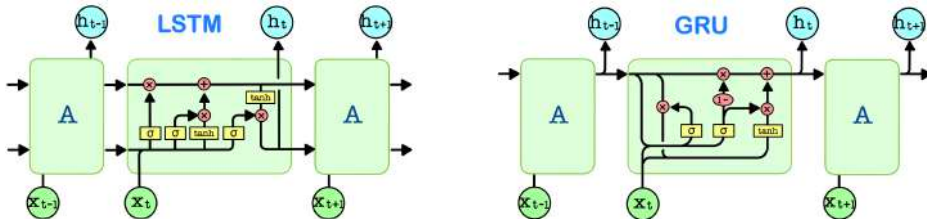


- **LSTM**: cell state c_t acting like an accumulator over time ensures that gradients don't decay.
 Backpropagation from c_t to c_{t-1} uses only elementwise multiplication by f_t ,
 but no matrix multiplication by W . $\Leftarrow c_t = f_t \odot c_{t-1} + i_t \odot g_t$

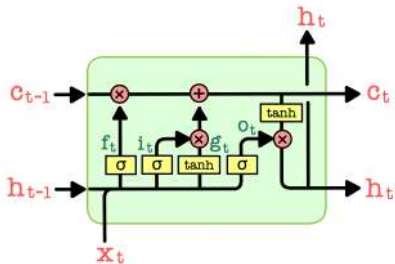


Gated Recurrent Unit (GRU)

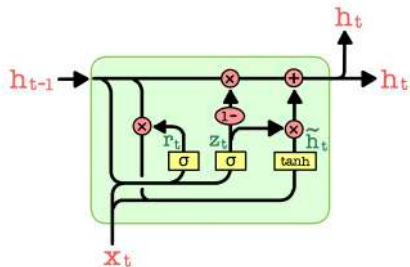
GRU is a simplified variant of LSTM by merging the cell state and the hidden state together, combining the input gate and the forget gate into a single update gate.



- * Hidden units affected by reset and update gates learn to capture dependencies over different time scales. Those units that learn to capture short-term dependencies will tend to have frequently active reset gates, but those that capture longer-term dependencies will have mostly active update gates.



$$\begin{aligned}
 i_t &= \sigma(W_{xh}^i x_t + W_{hh}^i h_{t-1} + b_h^i) && \text{input gate} \\
 f_t &= \sigma(W_{xh}^f x_t + W_{hh}^f h_{t-1} + b_h^f) && \text{forget gate} \\
 o_t &= \sigma(W_{xh}^o x_t + W_{hh}^o h_{t-1} + b_h^o) && \text{output gate} \\
 g_t &= \tanh(W_{xh}^g x_t + W_{hh}^g h_{t-1} + b_h^g) && \Leftarrow \text{RNN core} \\
 c_t &= f_t \odot c_{t-1} + i_t \odot g_t && \text{cell state} \\
 h_t &= o_t \odot \tanh(c_t) && \text{hidden state}
 \end{aligned}$$

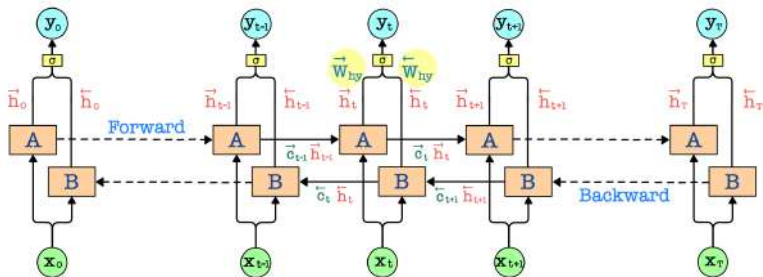


$$\begin{aligned}
 z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) && \text{update gate} \leftarrow \begin{array}{l} \text{input gate} \\ \text{forget gate} \end{array} \\
 r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) && \text{reset gate} \\
 \tilde{h}_t &= \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) && \Leftarrow \text{RNN core} \\
 h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t && \begin{array}{l} \text{cell state} \\ \text{hidden state} \end{array} \leftarrow
 \end{aligned}$$

Bidirectional LSTM

- Sometimes you want to incorporate information from data both preceding and following, and so input sequences are presented and learned both forward and backward.

$$y_t = \sigma(\vec{W}_{hy} \vec{h}_t + \overleftarrow{W}_{hy} \overleftarrow{h}_t + b_y)$$



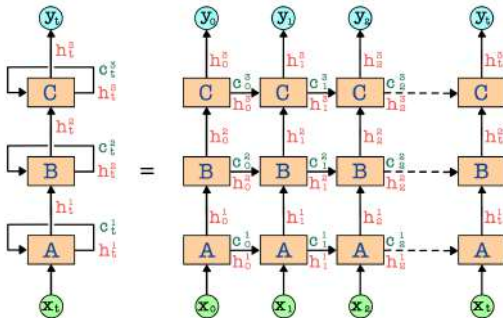
Stacked LSTM

- LSTM layers are stacked one on top of another.
- Deep models were often exponentially more efficient at representing some functions than a shallow one.

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f_t \odot c_{t-1}^l + i_t \odot g_t$$

$$h_t^l = o_t \odot \tanh(c_t^l)$$



- When stacking RNNs, it is mandatory to set `return_sequences=True` in Keras. The input shape for each recurrent layer is (batch_size, timesteps, input_dim). The output shape for each recurrent layer is (batch_size, timesteps, output_dim) when True, while (batch_size, output_dim) when False.

Dropout in LSTM

- Applying standard dropout to the recurrent connections results in poor performance since the noise caused by dropout at each time step (per-step mask) prevents the network from retaining long-term memory (this will erase all related past memories).

RNN regularization dropout^① (non-recurrent connections) $h_t^{l-1} \odot \mathbf{m}_t$, $m_{t,i} \sim \text{Bernoulli}(1-p)$

- Dropout with **per-sequence mask** generates a dropout mask for each input sequence, and keep it the same at every time step (on both the non-recurrent and recurrent connections). The elements in the hidden and cell states that are not dropped will persist throughout the entire sequence to maintain long-term memory.

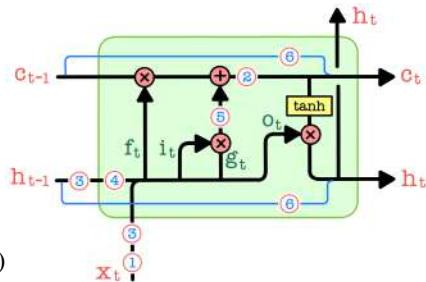
RNNdrop^② $c_t = (f_t \odot c_{t-1} + i_t \odot g_t) \odot \mathbf{m}$

Variational RNN dropout^③ $x_t \odot \mathbf{m}^x$ and $h_{t-1} \odot \mathbf{m}^h$

Weight-dropped LSTM^④ $(W_{hh}^* \odot \mathbf{M}) h_{t-1}$
(dropconnect on 4 hidden-to-hidden weight matrices)

Recurrent dropout^⑤ $c_t = f_t \odot c_{t-1} + i_t \odot g_t \odot \mathbf{m}$ (or \mathbf{m}_t)

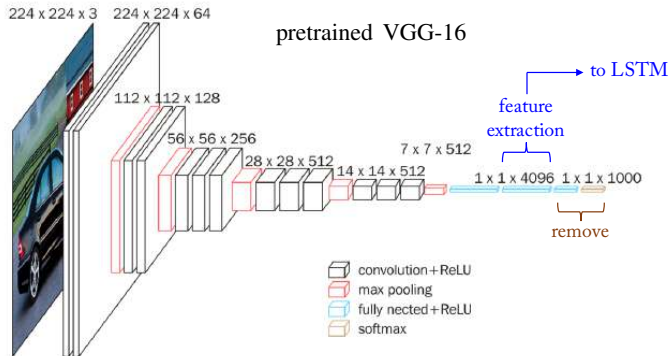
Zoneout^⑥ $c_t = c_{t-1} \odot \mathbf{m}_t^c + (f_t \odot c_{t-1} + i_t \odot g_t) \odot (1 - \mathbf{m}_t^c)$
 $h_t = h_{t-1} \odot \mathbf{m}_t^h + (o_t \odot \tanh(f_t \odot c_{t-1} + i_t \odot g_t)) \odot (1 - \mathbf{m}_t^h)$
(corresponding activations from the previous time-step)



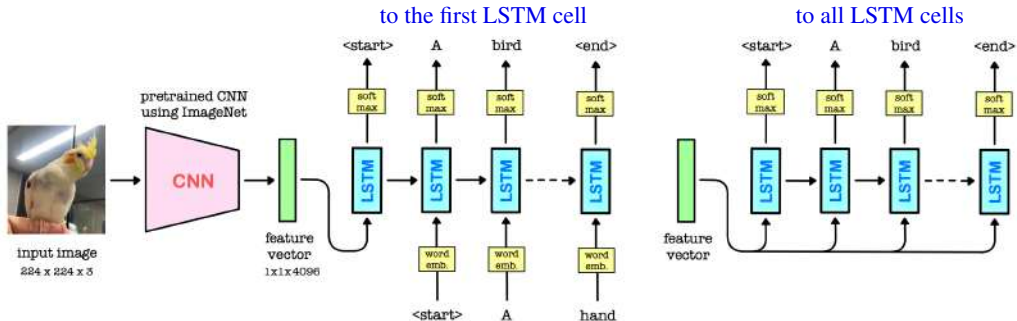
Attention Mechanism

CNN-LSTM model for Image Captioning

- CNN as an encoder is used to learn features in images.
LSTM as a decoder generates a text sequence describing image features.
- **Pretrained CNN** like VGG-16 or Resnet after removing the final FC classification layer acts as a feature extractor that compresses the image information into a smaller feature vector \mathbf{x} .



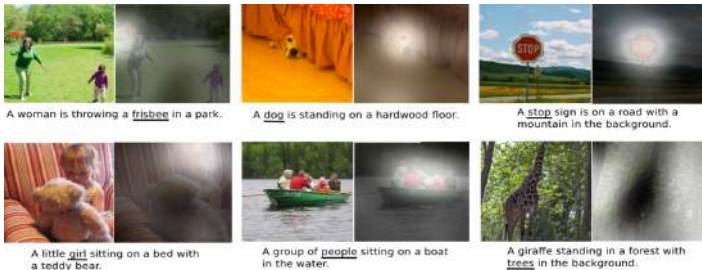
- **LSTM** decodes this feature vector (initial input) into a text sequence.
- We feed the image feature vector x to the first or all LSTM cells.
- LSTM starts the caption with a <start> token and repeatedly generate one word at a time until we predict the <end> token.
- Each LSTM cell calculates the hidden state $h_t = f(y_{t-1}, h_{t-1})$ or $h_t = f(x, h_{t-1})$, which is passed through FC layer and softmax to make the prediction $y_t = g(h_t)$.



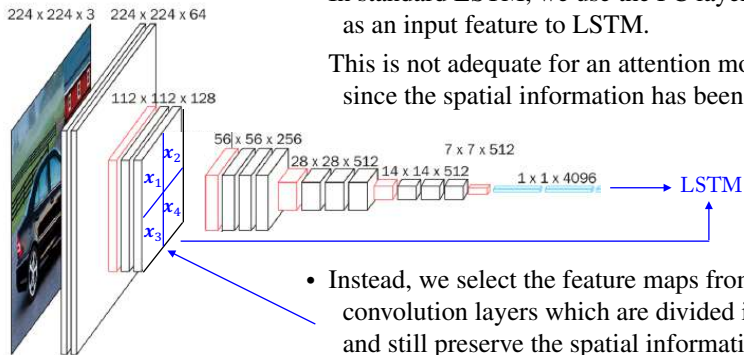
Visual Attention

Visual Attention, inspired by the way humans perform visual recognition tasks, **pays attention to particular areas or objects** rather than treating the whole image equally.

Ex. When we make a next prediction ‘sign’ based on the context ‘stop’ and the image, our attention shifts to the related area of the image.



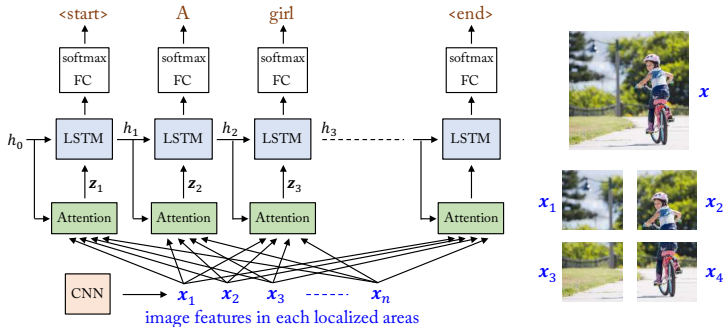
- We use an attention module $h_t = f(\text{attention}(\mathbf{x}, h_{t-1}), h_{t-1})$ having a context h_{t-1} and image features $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ in each localized areas.



- In standard LSTM, we use the FC layer output \mathbf{x} from CNN as an input feature to LSTM.

This is not adequate for an attention model since the spatial information has been lost.

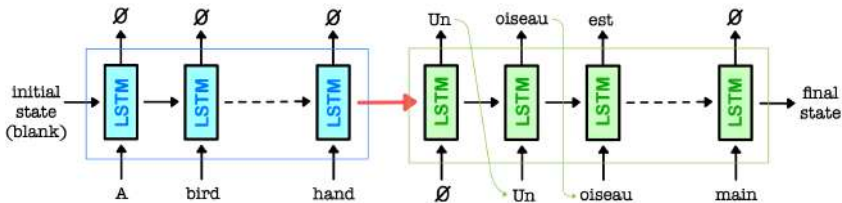
- Instead, we select the feature maps from one of earlier convolution layers which are divided into n pieces and still preserve the spatial information.
- Attention module takes new n image features of spatial regions $\mathbf{x} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ from CNN to give input features \mathbf{z}_t to LSTM.



- **Soft attention** (deterministic) uses weighted image features z_t instead of the image feature x .
 - $a_{ti} = f_{\text{att}}(x_i, h_{t-1})$ where f_{att} uses MLP. (ex. $a_{ti} = \tanh(Wh_{t-1} + W'x_i)$)
 - $z_t = \sum_{i=1}^n \alpha_{ti} x_i$ with the attention weights $(\alpha_{t1}, \dots, \alpha_{tn}) = \text{softmax}(a_{t1}, \dots, a_{tn})$
 - Differentiable (trained by **Backpropagation**)
- **Hard attention** (stochastic sampling) pays attention to x_i with the probability α_{ti} .
 - $z_t = \sum_{i=1}^n s_{ti} x_i$ where $p(s_{ti} = 1 | s_{j < t}, x) = \alpha_{ti}$ with $s_{ti} = \{0, 1\}$
 - We perform samplings and average our results using Monte Carlo method.
 - Non-differentiable (trained by **REINFORCE**)

Seq2Seq for Machine Translation

- **Seq2Seq** learning is a framework for mapping one sequence to another sequence.
- Encoder LSTM processes an input sentence word by word in word embedding form and compresses the entire content of the input sequence into a small fixed-size vector.
- Decoder LSTM predicts the output word by word based on the encoded vector, taking the previously predicted word as input at every step.
- * Encoder network reads in English (blue) and Decoder network writes in French (green).



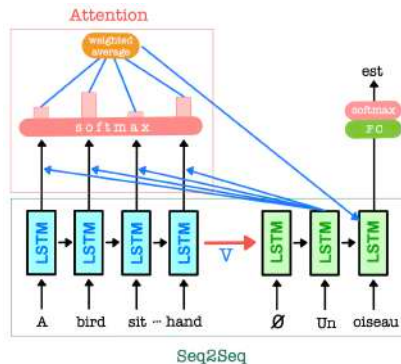
Attention LSTM

Attention mechanism is one of the core innovations in Machine Translation.

- Main bottleneck of Seq2Seq is that it requires to compress the entire content of the input sequence into a small fixed-size vector V .
- Attention (using query, key and value vectors) allows Decoder to look at Encoder hidden states, whose weighted average is an additional input to Decoder.

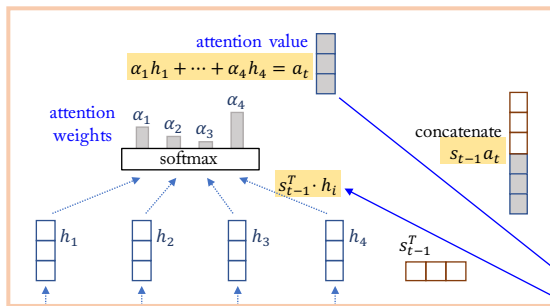
Decoder can pay attention to specific words in the input sequence again based on each hidden state of Decoder.

- It provides a glimpse (based on the attention weights) into the inner working of the model by inspecting which input parts are relevant for a particular output. (unveiling DNN black box)



Dot-product attention $s_{t-1}^T \cdot h_i$ with **query** s_{t-1} , **key** h_i and **value** h_i

\Rightarrow **attention weight** $\alpha_{ti} = \text{softmax}(s_{t-1}^T \cdot h_i)$ and **attention value** $a_t = \sum_j \alpha_{tj} h_j$



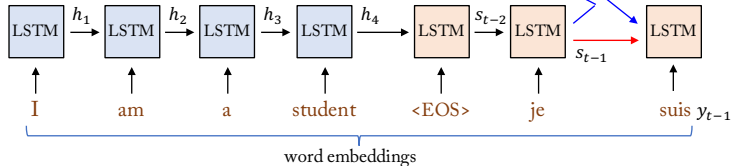
etudiant y_t

Softmax
FC

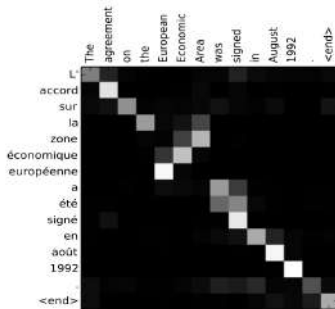
Attention LSTM
 $s_t = f(s_{t-1} a_t, y_{t-1})$

Standard LSTM
 $s_t = f(s_{t-1}, y_{t-1})$

dot-prod
general
concat
with learnable W and v



- Instead of encoding the whole sentence into one hidden state in LSTM, attention re-uses each hidden state (of input words in Encoder) at each step of Decoder.
- The idea is that there might be relevant information in every word in a sentence.
- Some problems are not solved with attention LSTM.
For example, processing input words cannot be done with parallel computing so that for a large corpus, this increases the training time.



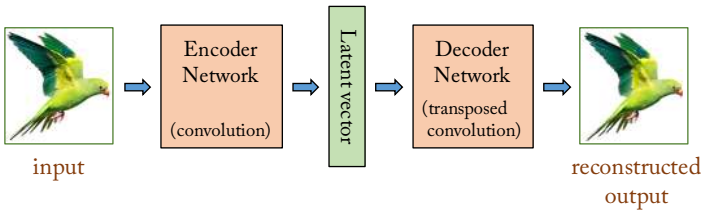
- **Transformer** has an encoder-decoder structure (no RNN structure) and was developed to solve the **parallel computing** and **global dependency** problems in Machine Translation.
It relies solely on **self-attention** to allow parallel computing and to learn global dependency regardless of their distance in the input or output sequences without using recurrent layers.
- **BERT** (Bidirectional Encoder Representations from Transformers) is a pretrained multi-layer bidirectional Transformer Encoder for language understanding developed by Google.

Auto-Encoder

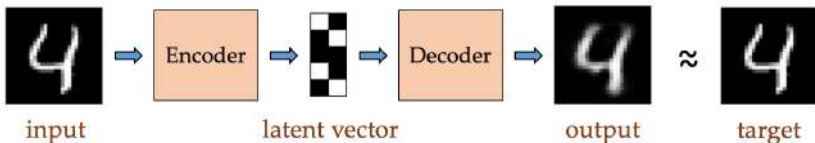
Variational Auto-Encoder

Auto-Encoder (AE)

- **Auto-Encoder** is an **unsupervised learning for data reconstruction** by encoding-decoding. Specifically, its bottleneck network forces a compressed information representation of an input.
- **Dimensionality reduction**: compressing an input into a **lower dimensional vector (latent vector)** and then reconstructing the input from the latent vector as an output.
- **Data-specific**: meaningfully compressing data ⁽¹⁾with strong correlation between input features, and/or ⁽²⁾similar to what they have been trained on.
Weak correlation between input features results in poor performance.
- **Lossy**: the output is of lower quality than the input due to the bottleneck network design.



- **Goal:** get an output $\hat{\mathbf{y}}^k = (\hat{y}_1^k, \dots, \hat{y}_d^k)$ which is identical with the input $\mathbf{x}^k = (x_1^k, \dots, x_d^k)$.
- **Loss function:**
 MSE for real values: $L(\theta) = \frac{1}{2} \sum_k \|\hat{\mathbf{y}}^k - \mathbf{x}^k\|_2^2 = \frac{1}{2} \sum_k \sum_i (\hat{y}_i^k - x_i^k)^2$
 Cross-entropy for binary $\{0, 1\}$: $L(\theta) = - \sum_k \{x^k \log \hat{y}^k + (1 - x^k) \log(1 - \hat{y}^k)\}$
- We use the standard backpropagation for feedforward neural networks.

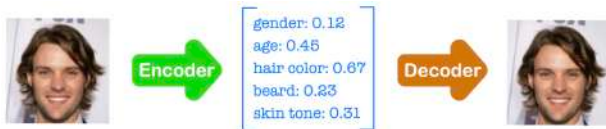
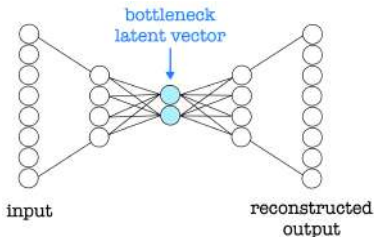


Dimensionality reduction

Two main interesting practical applications of **Auto-Encoder**

- Compressed representations (latent vector) generated by AE can be fed into other algorithms as **lower dimensional inputs** for classification, clustering and anomaly detection.
- Data visualization** when the raw data has high dimensionality and cannot be easily plotted.

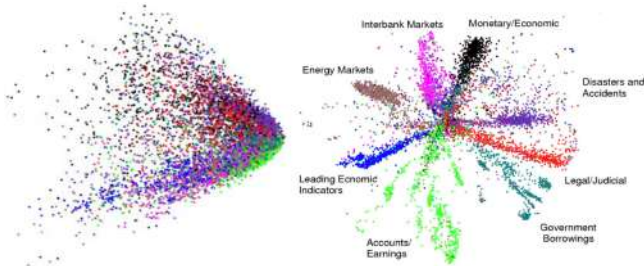
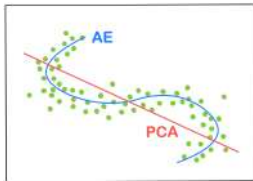
The encoded vector can sometimes be compressed into a 2D or 3D space.



Ex) AE on a face dataset with an encoding dimension 5:
AE learns five descriptive attributes (features) of faces such as gender, age, etc to describe each data.

Two dimensionality reductions for high-dimensional data visualization:

- **PCA** (Principal Component Analysis) attempts to find a lower dimensional **hyperplane** (linear) which describes the original data.
- **AE** is capable of learning a **nonlinear manifold** describing the data in a lower dimensionality. It is a powerful nonlinear generalization of PCA.



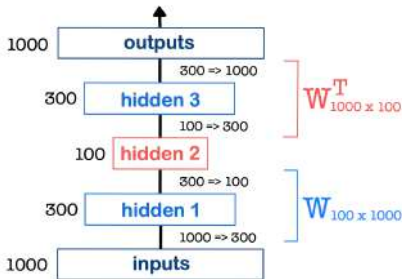
PCA (left) and AE (right) in 2D space

‘Reducing the dimensionality of data with neural networks’
by Hinton (Science 2006)

Tied weight Auto-Encoder

Tied weight AE has a decoder which is the mirror image of the encoder, and the decoder layer weight matrix $W_2 = W^T$ is the transpose of the encoder layer weight matrix $W_1 = W$.

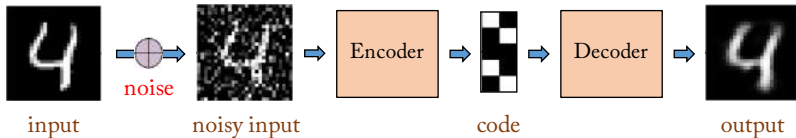
- This is usual but not always.
The only requirement is that the dimensionality of the input and output needs to be the same.
- Due to halving the number of weights, it speeds up training and reduces the risk of overfitting.



Denoising Auto-Encoder

Denoising AE adds random Gaussian noise to the input data and uses it as input to recover the original noise-free data.

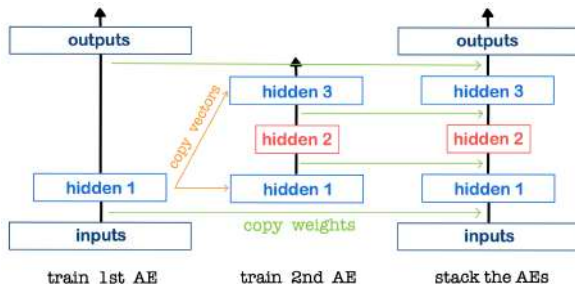
- We use a noisy version of an image as input and the original image as target. It learns to diminish the noise and produces the underlying meaningful data.
- Only little difference between the implementations of AE and Denoising AE.
 - AE: Loss function(input, output from the input)
 - Denoising AE: Loss function(input, output from a noisy input)



Stacked Auto-Encoder (Deep AE)

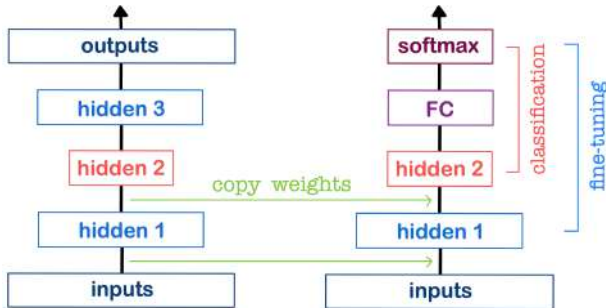
Stacked AE stacks many one-layer AEs to form a deep network by feeding the latent vector of AE found on the former layer as the input for the next layer.

- Train the first AE from the input data and obtain the learned latent vector.
- The latent vector of the former layer is used as the input for the next layer, and this procedure is repeated until the training completes.
- This **unsupervised training** is done one layer at a time.



Classification task using Stacked Auto-Encoder

- Consider only the **encoding parts** of AE after all layers are pre-trained (unsupervised).
- Add a MLP (FC) on the top latent vector of the encoder network.
- Freeze the weight parameter of the encoder network and train only the upper network for classification task (supervised), and additionally apply fine-tuning (supervised).



Semi-supervised Learning using Stacked Auto-Encoder

Semi-supervised learning: supervised task with some **labeled** data and mostly **unlabeled** data.

- Train a stacked AE using all the labeled and the unlabeled data (unsupervised pre-training).
- Then, reuse the encoding layers with adding a layer on the top of the stack to create a network for your actual task.
- Finally, train it using the labeled data (supervised learning).

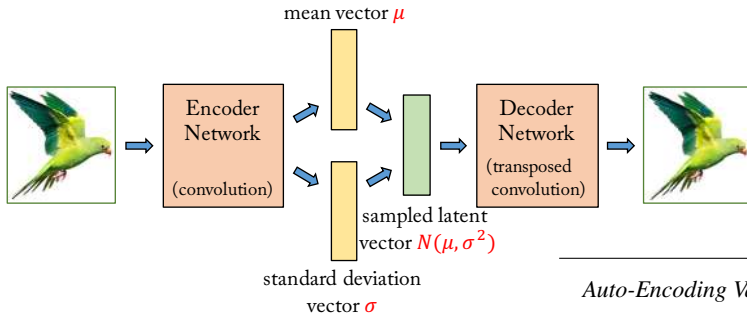
Anomaly detection using Stacked Auto-Encoder

Anomaly detection: identification of items (**outliers**) standing out from the standard of a dataset.

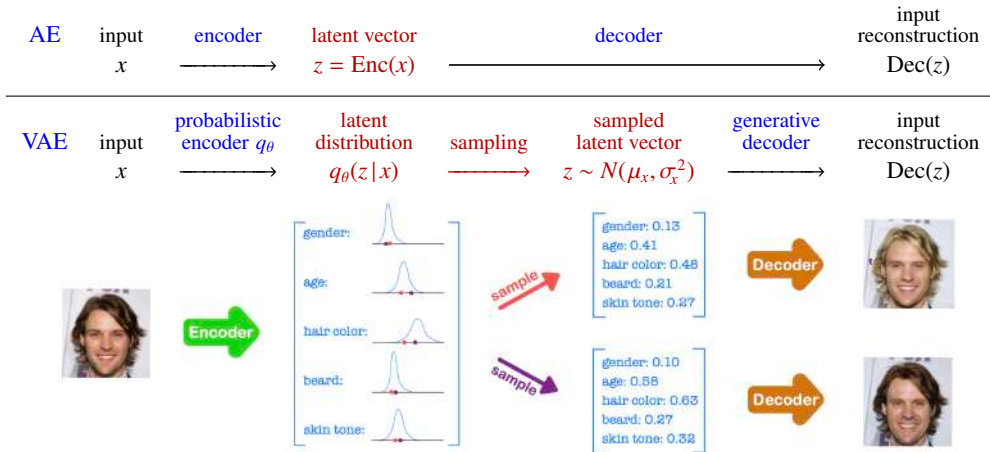
- Anomaly detection does not require labeled data, and can be done with unsupervised learning.
- Idea: train a stacked AE, and identify outliers having high reconstruction error.
(typical examples have low reconstruction error, whereas outliers should have high reconstruction error)
- Anomaly detection applications: network intrusion detection, systems monitoring, sensor network event detection (IoT), and abnormal trajectory sensing.

Variational Auto-Encoder (VAE)

- **AE** encoder directly produces a latent vector z (single value for each attribute).
Then, AE decoder takes these values to reconstruct the original input.
Goal is getting a compressed latent vector of the input.
- **VAE** encoder produces 2 latent vectors mean μ and standard deviation σ of Gaussian $N(\mu, \sigma^2)$ approximating the posterior distribution $p(z|x)$ of each attribute (Variational inference).
Then, using sampled latent vectors from the distributions, the decoder reconstructs the input.
Goal is generating many variations of the input (probabilistic encoder + generative decoder).



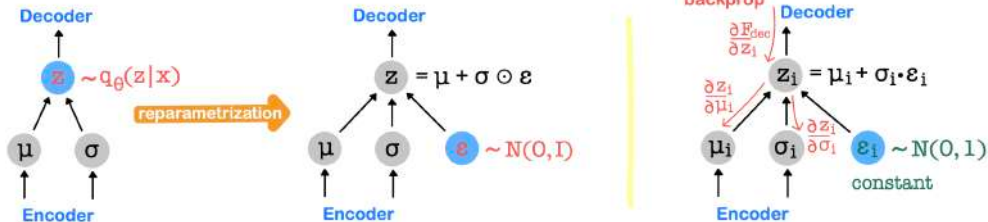
- VAE encoder would like to produce the posterior $p(z|x)$ instead of a single point z , but because of the intractability of $p(z|x)$, we approximate it by **variational posterior** $q_\theta(z|x)$ which is Gaussian $N(\mu_x, \sigma_x^2)$ so that the encoder network q_θ is enforced to produce μ_x and σ_x .



- Just using encoder output as μ and σ is not enough to make our latent space well-behaved.
The network would simply adjust μ to the latent activations of standard AE (σ becomes 0).
- **Goal:** (1) Minimize the reconstruction error $L(\hat{y}, x) = \frac{1}{2} \sum_i (\hat{y}_i - x_i)^2$ pixel-wise MSE.
(**Reconstruction term** that makes the encoding-decoding scheme efficient)
(2) Enforce the learned latent distribution $q_\theta(z|x) \sim N(\mu_x, \sigma_x^2)$ to be similar to $N(0,1)$.
(**Regularization term** on the latent layer that provides the latent space regularity)
- **Loss function:** $L(\hat{y}, x) + \beta D_{\text{KL}}(N(\mu_x, \sigma_x^2) \| N(0, I))$ (tradeoff between MSE and KL divergence)
* Kullback-Leibler divergence $D_{\text{KL}}(P \| Q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx = \mathbb{E}_{x \sim P} \left[\log \frac{p(x)}{q(x)} \right]$
- When training the model, each parameter is updated by using **backpropagation**.
However, we simply cannot do this for a **random sampling process** in $L(\hat{y}, x)$ term.
- To implement VAE, we propose **Reparametrization trick**.
We randomly sample $\epsilon \in N(0, I)$, and then take a random latent vector sample $z = \mu + \sigma \odot \epsilon$.

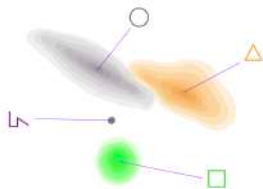
Reparametrization trick

- We cannot compute gradients for the weights involving random sampling $z \sim N(\mu_x, \sigma_x^2)$ and so cannot use backpropagation directly.
- Reparametrization trick** suggests that we randomly sample $\epsilon \in N(0, I)$ instead of $N(\mu_x, \sigma_x^2)$, and take a latent sample $z = \mu + \sigma \odot \epsilon$ separating the random component from the weights. It allows computing gradients over μ and σ since ϵ is treated as a constant per sample.

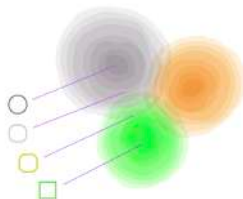


Regularization

- **Regularity** that is expected from the latent space to make generative process possible can be expressed through two properties:
 - **Continuity**: two close points in the latent space should not give two completely different contents once decoded.
 - **Completeness**: for a chosen distribution, a point sampled from the latent space should give 'meaningful' content once decoded.
- **Regularization term** forces the covariance matrix to be close to I (preventing narrow distributions), and the mean to be close to 0 (preventing encoded distributions to be far apart from each others).



without regularization

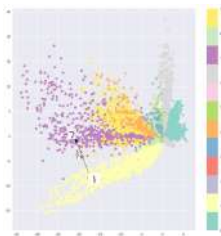


with regularization

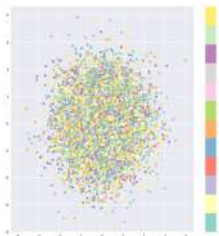
Visualization of latent space

- **AE** learns to produce a lower dimensional latent space and reconstructs the inputs well.
For generative modeling, the problem with AE is that the latent space is irregular so that it is not continuous and there is a gap between separate clusters, so interpolation is not good.
When you generate variations in the gap, the decoder simply produces unrealistic outputs, because the decoder doesn't know how to process data in the gap.
- **VAE** can learn a continuous and smooth latent space.
This allows random sampling in the latent space and generate realistic variations of an input.

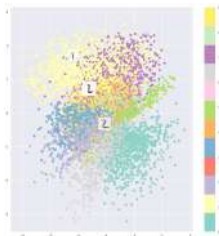
only reconstruction error (AE)



only KL-divergence

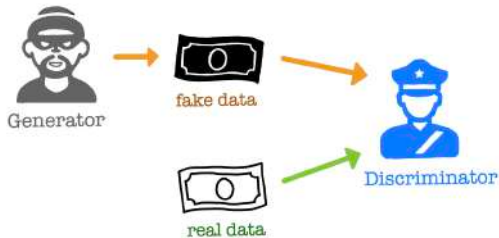


combination (VAE)



Generative Adversarial Network

Generative Adversarial Network (GAN)



- **Generative**
Learn a generative model
- **Adversarial**
Trained in an adversarial setting: generator G and discriminator D
- **Network**
Use deep neural networks: ranging from MLP, CNN, RNN to AE, DRL

- **GAN** has had a profound impact on the AI industry since Ian Goodfellow proposed in 2014.

Yann LeCun described GAN as “the most interesting idea in the last 10 years in Machine Learning”.

- GAN changed Deep Learning paradigm that has been focused on supervised learning to **unsupervised learning**.
- Various follow-up studies and applications of GAN as a **generative model** are spreading widely because they can **create fake data** that are difficult to distinguish from real data.
- GAN **training is hard** because of non-convergence and mode collapse problems.

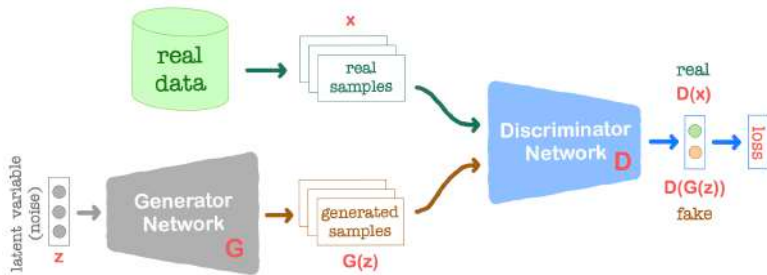
Ian Goodfellow



Residence	San Francisco, California
Nationality	American
Alma mater	Stanford University Université de Montréal
Known for	Generative adversarial networks, Adversarial Examples
	Scientific career
Fields	Computer science
Institutions	Apple Inc. Google Brain OpenAI
Doctoral advisor	Yoshua Bengio, Aaron Courville
Website	www.iangoodfellow.com

GAN architecture

- **Generator G** generates ‘fake’ samples that are intended to come from the same distribution as ‘real’ data, to maximize the probability of D making a mistake.
- **Discriminator D** evaluates the probability that the sample came from real data rather than G .
- G and D oppose each other, but at the same time help each other in their own tasks.
- Repeatedly train the two networks alternately and we get better G and D .
- After finishing the training, we usually discard D , which is used just for training G .



Objective function

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

- **Discriminator D** is trying to maximize $L(D, G)$ so that
 - $D(\mathbf{x})$ on the real data distribution $p_{\text{data}}(\mathbf{x})$ is close to 1 ‘real’
 - $D(G(\mathbf{z}))$ on a simple distribution $p_z(\mathbf{z})$ such as uniform or normal is close to 0 ‘fake’.
- **Generator G** is trying to minimize Discriminator’s reward so that $D(G(\mathbf{z}))$ is close to 1 (Discriminator is fooled into thinking that $G(\mathbf{z})$ is real).
- In two-player minimax problem, the solution is the same as Nash equilibrium.

Nash equilibrium: $p_{\text{data}} = p_g$ (so, $D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} = \frac{1}{2}$ for all \mathbf{x}).

Globally optimal solution is that the generated distribution matches the real distribution.

Nash equilibrium

- A set of player strategies is a Nash equilibrium if no player can do better by unilaterally changing his strategy (every player's strategy is optimal, holding constant the strategies of all the other players).

A game may have multiple Nash equilibria or none at all.

		Player B	
		Option 1	Option 2
Player A	Option 1	6, 6	4, 7
	Option 2	7, 4	5, 5

← Nash equilibrium

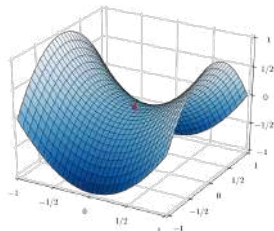
The prisoner's dilemma

		Prisoner B	
		Confess	Keep quiet
Prisoner A	Confess	Both go to jail for ten years	Prisoner B gets life imprisonment, A goes free
	Keep quiet	Prisoner A gets life imprisonment, B goes free	Both go to jail for one year

- Saddle model: two players A and B control x and y respectively, and A/B want to minimize / maximize $L(A, B) = x^2 - y^2$.

$$\min_A \max_B L(A, B) = x^2 - y^2$$

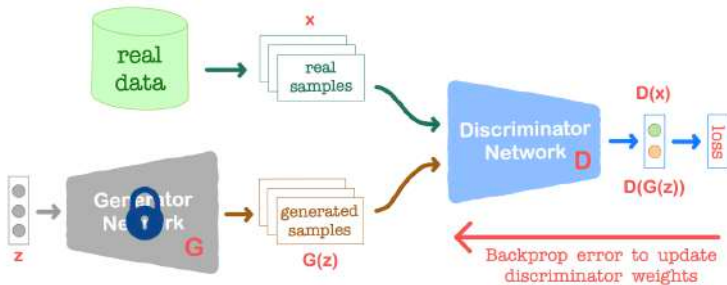
The unique Nash equilibrium is $x = y = 0$ (saddle point).



Training Discriminator D by Backpropagation

- Fix Generator parameter θ_g .
- Perform **gradient ascent on Discriminator** (parameter θ_d) using real \mathbf{x} and generated $G(\mathbf{z})$ data.
Minibatch of m real $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ and m fake $\{G(\mathbf{z}^{(1)}), \dots, G(\mathbf{z}^{(m)})\}$

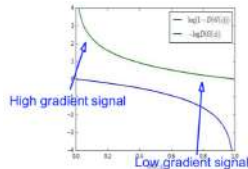
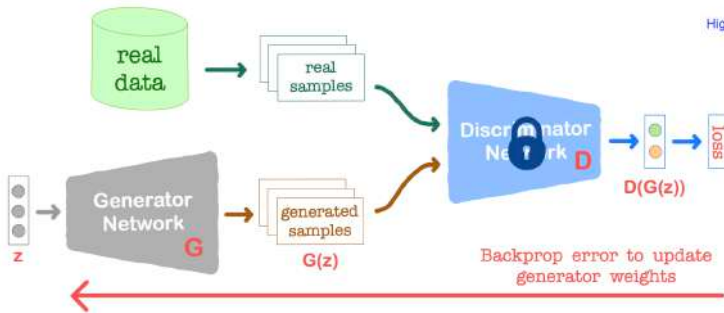
$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\log D(\mathbf{x}^{(i)}) + \log (1 - D(G(\mathbf{z}^{(i)})))]$$



Training Generator G by Backpropagation

- Fix Discriminator parameter θ_d .
- Perform **gradient descent on Generator** using sample noise vectors \mathbf{z} from $N(0, 1)$ or $U(-1, 1)$.
Early in learning, $D(G(\mathbf{z}^{(i)})) \approx 0$ because G is poor, so we train G to minimize $-\log D(G(\mathbf{z}^{(i)}))$.

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(1 - D(G(\mathbf{z}^{(i)}))) \quad \text{or} \quad \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m -\log D(G(\mathbf{z}^{(i)}))$$



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 $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

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

end for

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Update the generator by descending its stochastic gradient:

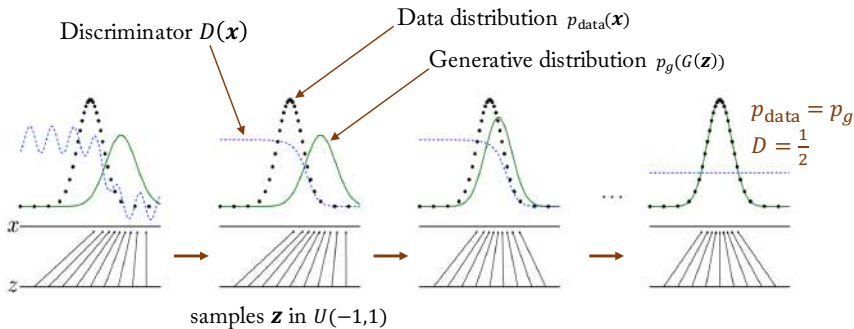
$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

discriminator
update

generator
update



Consider D and G near convergence: p_g is similar to p_{data} and D is a partially accurate classifier.

- \Rightarrow In the inner loop of Algorithm 1, D is trained to discriminate samples, converging to $D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}$.
- \Rightarrow After an update to G , gradient of D has guided $G(\mathbf{z})$ to flow to regions that are more likely to be real data.
- \Rightarrow After several steps of training, they will reach a point at which both cannot improve because $p_g = p_{\text{data}}$.
The discriminator is unable to differentiate between the two distributions, i.e. $D(\mathbf{x}) = \frac{1}{2}$.

Global optimality: $p_{\text{data}} = p_g$ (Nash equilibrium)

- For fixed generator G , the maximal discriminator D is $D_G^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}$.

$$\because L(D, G) = \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log D(\mathbf{x}) d\mathbf{x} + \int_{\mathbf{z}} p_z(\mathbf{z}) \log (1 - D(g(\mathbf{z}))) d\mathbf{z} = \int_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) \log D(\mathbf{x}) + p_g(\mathbf{x}) \log (1 - D(\mathbf{x})) d\mathbf{x}$$

For non-zero a and b , the function $a \log(y) + b \log(1-y)$ has its maximum at $y = \frac{a}{a+b}$ in $[0, 1]$.

- After maximizing $\max_D L(D, G)$, the global minimal generator G is achieved when $p_{\text{data}} = p_g$.

$$\begin{aligned} \because L(D^*, G) &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\log D_G^*(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim p_g} [\log (1 - D_G^*(\mathbf{x}))] = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right] + \mathbb{E}_{\mathbf{x} \sim p_g} \left[\log \frac{p_g(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})} \right] \\ &= D_{\text{KL}}\left(p_{\text{data}} \parallel \frac{p_{\text{data}} + p_g}{2}\right) + D_{\text{KL}}\left(p_g \parallel \frac{p_{\text{data}} + p_g}{2}\right) - 2 \log 2 = 2 \cdot D_{\text{JS}}(p_{\text{data}}, p_g) - \log 4 \end{aligned}$$

$$* D_{\text{KL}}(P \parallel Q) = \mathbb{E}_{\mathbf{x} \sim P} \left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})} \right], \text{ and } D_{\text{JS}}(P, Q) = \frac{1}{2} D_{\text{KL}}\left(P \parallel \frac{P+Q}{2}\right) + \frac{1}{2} D_{\text{KL}}\left(Q \parallel \frac{P+Q}{2}\right) \geq 0 \text{ (zero iff } P = Q\text{)}.$$

- $p_{\text{data}} = p_g$ implies that the generative model perfectly replicating the real data distribution.
- If D and G have enough capacity, and at each step of Algorithm 1, D reaches D^* for fixed G , and p_g is updated so as to improve $L(D^*, G)$, then p_g converges to p_{data} .

Pros and Cons for GAN

Pros:

- **Unsupervised learning**, so no need for labeled data.
- **Generate high-quality images.**
- The entire network can be trained with backpropagation.

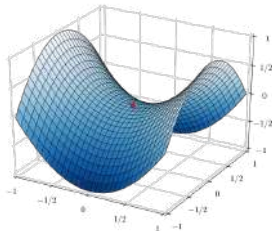
Cons:

- **Non-convergence**
- **Diminished gradient**
- **Mode collapse** (low diversity)
- No explicit representation of $p_g(G(z))$.
- Sensitive to the hyper-parameter selections, and so need to manually babysit during training.
- No evaluation metric, so unclear stopping criteria and hard to compare with other models.
Discriminator is not a good metric in measuring the image quality or its diversity.
(Inception Score and Fréchet Inception Distance measure the performance of GAN.)

Difficulty in Training GAN

Non-convergence in two-player minimax problem

- In the original GAN objective, the **update of G and D is done independently** with no respect to other player in the game.
- Even if each player successfully moves downhill on that player's update, the same update might move the other player uphill.
- Sometimes two players reach **Nash equilibrium**, which is the unique globally optimal solution. In other scenarios, they repeatedly undo each other's progress without arriving anywhere useful.
- In practice, GAN's loss functions often seem to oscillate (not to converge) without eventually reaching an equilibrium.



Diminished gradient

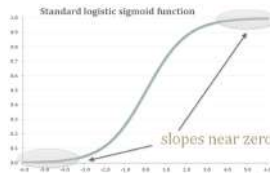
- If discriminator D is too powerful, generator G would fail to train effectively.

Since D tells that the generated images $G(z)$ are fake,
 $D(G(z))$ is always close to 0.

- If D is too lenient, the generated images are useless.

Then G is able to fool D easily.

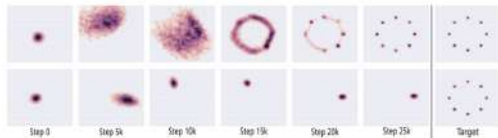
- In either case, learning stops for both neural networks because neither one is receiving any feedback on how to get better.



Mode collapse (most severe form of non-convergence)

- Real-life data distributions are multimodal. Ex. MNIST dataset has 10 major modes.
- The generator does not cover all the data distribution and loses diversity.

It easily gets trapped in local optima by memorizing training data, and so it will continue to generate similar images.

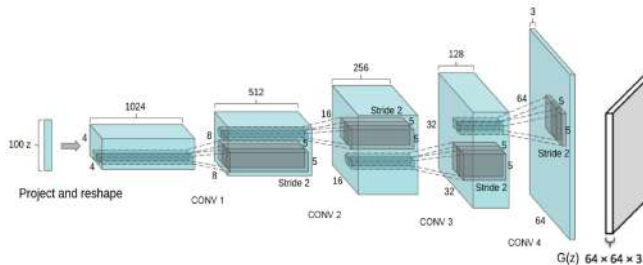


Deep Convolutional GAN (DCGAN)

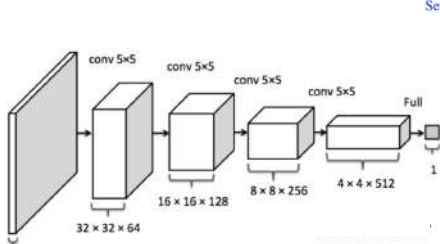
- **DCGAN** is a successful network for GAN and unsupervised learning with CNN.
- The generator has interesting **vector arithmetic** properties allowing for easy manipulation of many semantic qualities of generated samples.



Generator network G



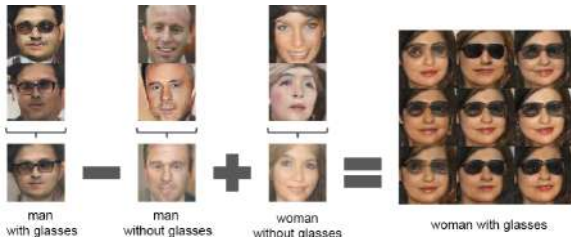
Discriminator network D



DCGAN architecture

- Generator uses fractionally-strided (or transposed) convolutions for up-sampling.
- Discriminator uses convolutions with stride for down-sampling.
- Remove max pooling and FC hidden layers in convolution networks.
- Use batch normalization in both Generator and Discriminator.
- Generator uses ReLU activation except for the output which uses tanh $(-1, 1)$.
- Discriminator uses LeakyReLU except for the output which uses sigmoid $(0, 1)$.

- Latent vectors capture interesting patterns (vector arithmetic).



- * Heuristic method to find latent vectors representing images with some characters
- The process is very manual (not systematic):
- generate a bunch of images,
 - find images having desired characteristic,
 - average their latent vectors together and hope that it captures the desired structure.



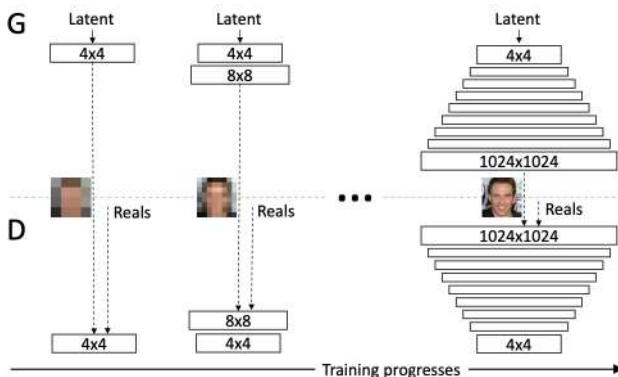
- * Interpolation between a series of random points in the latent vector space shows that the learned space has a smooth transition, and all images of the space look plausible like a bedroom.

Progressive growing GAN (PGGAN)

- Training methodology for PGGAN is starting with a low resolution image, and then progressively increase the resolution by adding layers to the networks D and G .
- This incremental nature allows the training to first discover large-scale structure of the image, and then shift attention to increasingly finer scale details during training.
- This both speeds the training up and greatly stabilizes it, producing high resolution images.

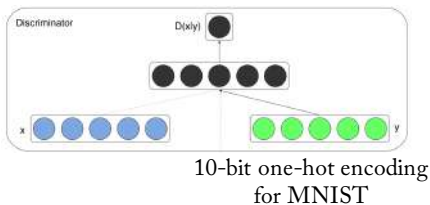
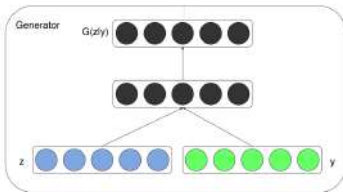


- Start training with both D and G having a low resolution of 4×4 pixels.
- During training, we incrementally add layers to D and G to increase the resolution.
- All existing layers can be trained throughout the process.
- This allows stable synthesis in high resolutions, and also training speed is increased.



Conditional GAN (cGAN)

- **cGAN** is a simply extended version to a conditional model so that both D and G are conditioned on some extra information y such as class labels.

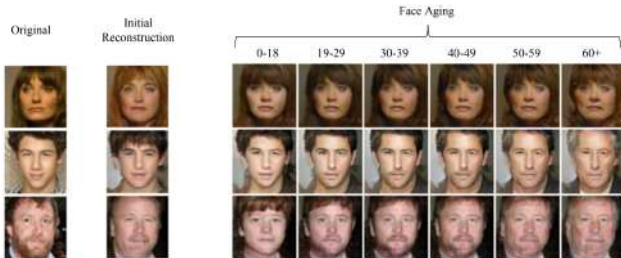


$$\min_G \max_D L(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x | y)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z | y)))]$$

- MNIST: each row is conditioned on one label by one-hot encoding.

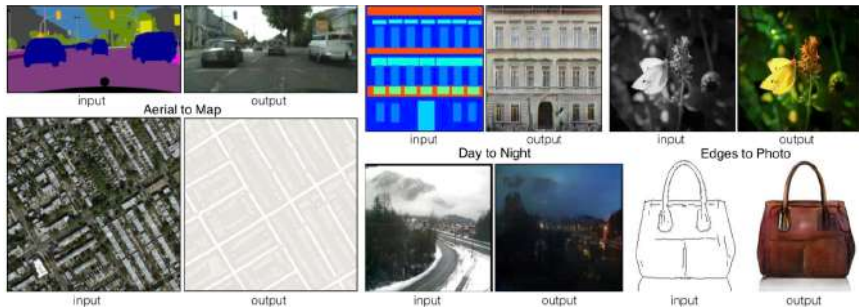


- Face aging with cGAN



Pix2Pix with cGAN

- Pix2Pix is a cGAN oriented to image-to-image translation tasks, especially those involving high resolution outputs.

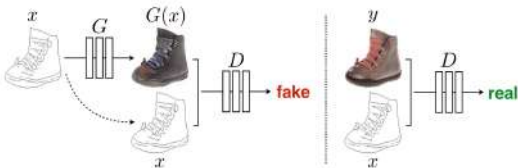


$$\min_G \max_D L_{\text{cGAN}}(D, G) + \lambda L_{L_1}(G)$$

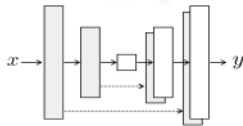
$$L_{\text{cGAN}}(D, G) = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_{x,z}[\log (1 - D(x, G(x,z)))]$$

$$L_{L_1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1]$$

- **cGAN** maps graphic \rightarrow photo.
Unlike an unconditional GAN,
both observe the input graphic x .



- Generator uses **U-Net**, which is an encoder-decoder with skip connections between mirrored layers.
- Discriminator uses a convolutional **PatchGAN** classifier, which only penalizes structure at the scale of image patches $N \times N$, so it tries to classify if each patch is real or fake.



- Different losses induce different quality of results.



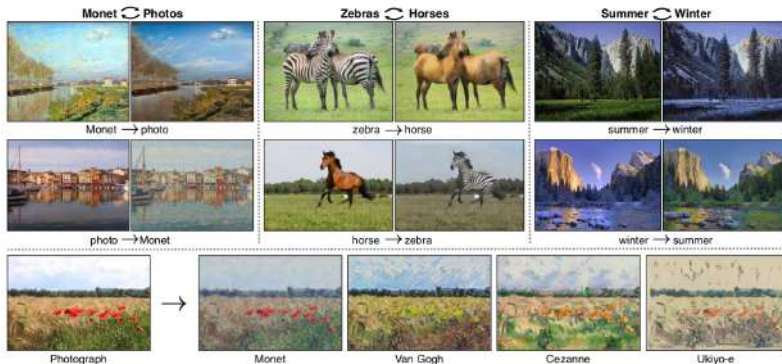
If we take a naïve CNN to minimize L_1 distance between predicted and ground truth pixels, it will tend to produce blurry results, which do not look real images. This is why we use GAN to generate sharper images.

- Patch size variations.



CycleGAN

- CycleGAN translates an image from a source domain X to a target domain Y in the absence of paired examples.

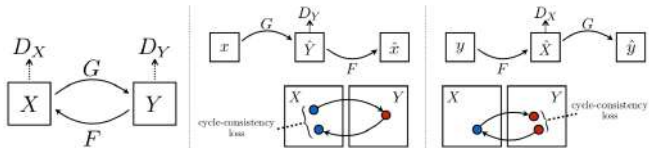




$$\min_{G, F} \max_{D_X, D_Y} L_{\text{GAN}}(G, D_Y, X, Y) + L_{\text{GAN}}(F, D_X, Y, X) + \lambda L_{\text{cyc}}(G, F)$$

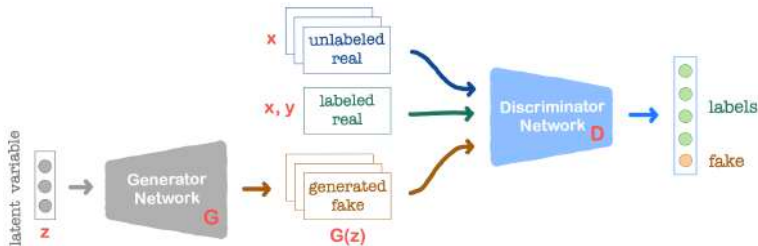
$$L_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)} [\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log (1 - D_Y(G(x)))]$$

$$L_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)} [\|G(F(y)) - y\|_1]$$



Semi-supervised GAN (SGAN)

- SGAN is for the **semi-supervised learning** as the **discriminator** is a $(k+1)$ -class classifier trained on the labeled and unlabeled real data belonging to k classes and generated fake data.
- Generator's fake data along with the real data help Discriminator to precisely classify.
- SGAN is mainly used to create a more data-efficient classifier (Discriminator) and additionally allows for generating higher quality samples (Generator).



Semi-supervised Learning (SSL)

Supervised Learning

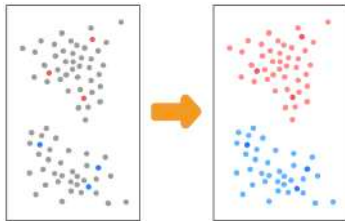
- Algorithm trains from **labeled data** consisting of object-label pairs (\mathbf{x}_i, y_i) .
- It learns a general rule for **classification** to predict the labels for the remaining data.

Unsupervised Learning

- Algorithm trains from **unlabeled data** consisting of only objects (\mathbf{x}_i) without any label.
- It detects patterns in the data by identifying **clusters** of similar objects.

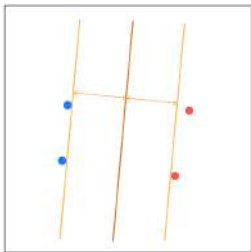
Semi-supervised Learning

- Algorithm learns from a small portion of labeled data and a large amount of unlabeled data (in the same domain).
- It learns a rule for **classification** to predict the labels for unlabeled data.



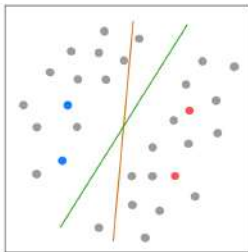
Semi-supervised learning is useful in Machine Learning.

SVM (support vector machine)



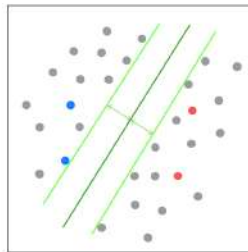
without unlabeled data

Which one is better?



decision boundaries

Transductive SVM (SSL)



with unlabeled data

Semi-supervised GAN (SGAN)

- In a **regular GAN**, we trained Generator G and Discriminator D alternately and here D outputs the probability that the sample came from the real data, but by the end of training, we **discard D** because we used D just for training G .
- In a **SGAN**, we transform D into a **multi-class classifier** that classifies the dataset categories using only few labeled data and helped by the unlabeled real data and generated fake data.
- This time, by the end of training, we **discard G** because we use G for guiding D . Here, G acts as a different source from which D gets raw unlabeled data.
- Discriminator has **three different sources of input data** for training.
 - labeled real data: image-label pairs are provided as in supervised classification.
 - unlabeled real data: D only learns that these data are real.
 - generated fake data: D learns that these data are fake.

Objective function

Regular GAN $\min_G \max_D L(D, G) = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} [\log (1 - D(G(\mathbf{z})))]$

- At training time, D is made to predict which of $k+1$ classes the input belongs to, having an extra **fake data class**, as $\mathbf{x} \rightarrow (l_1, l_2, \dots, l_k, l_{k+1}) \rightarrow p_{\text{model}}(y = i | \mathbf{x}) = \frac{e^{l_i}}{\sum_{j=1}^{k+1} e^{l_j}}$

- Discriminator loss $L_D = L_{D_{\text{supervised}}} + L_{D_{\text{unsupervised}}}$

$$L_{D_{\text{supervised}}} = -\mathbb{E}_{\mathbf{x}, y \sim p_{\text{data}}(\mathbf{x}, y)} \log(p_{\text{model}}(y = i | \mathbf{x}, i \leq k)) \text{ for labeled real data}$$

$$L_{D_{\text{unsupervised}}} = -\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \log(1 - p_{\text{model}}(y = k+1 | \mathbf{x})) \text{ for unlabeled real data}$$

$$-\mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} \log(p_{\text{model}}(y = k+1 | G(\mathbf{z}))) \text{ for generated fake data}$$

- Generator loss $L_G = L_{G_{\text{cross-entropy}}} + L_{G_{\text{feature matching}}}$

$$L_{G_{\text{cross-entropy}}} = -\mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} \log(1 - p_{\text{model}}(y = k+1 | G(\mathbf{z}))) \text{ for } G \text{ fooling } D$$

$$L_{G_{\text{feature matching}}} = \left\| \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}(\mathbf{x})} \mathbf{f}(\mathbf{x}) - \mathbb{E}_{\mathbf{z} \sim p_z(\mathbf{z})} \mathbf{f}(G(\mathbf{z})) \right\|_2^2 \text{ to generate data with similar statistics}$$

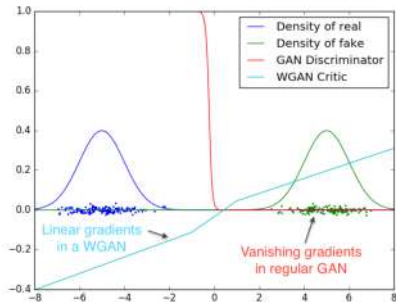
- * Feature matching prevents mode collapse and allow for a more stable training.

G is trained to generate data that matches the statistics of the real data

(the expected value of the features $\mathbf{f}(\mathbf{x})$ which denote activations on an intermediate layer of D).

Wasserstein GAN (WGAN)

- In regular GAN, non-convergence and mode collapse are common. GAN discriminator may fill with areas with vanishing gradients.
 - WGAN proposes a cost function using Wasserstein distance that has a smoother gradient everywhere.
 - WGAN critic learns better even the generator makes poor images (improving the stability of learning, and getting rid of problems like mode collapses).
- * However, Wasserstein distance equation is highly intractable. Kantorovich-Rubinstein duality simplifies the cost function, and we need to find a 1-Lipschitz function, which can be learned with DNN (WGAN critic) similar to discriminator.



Distances and divergences

Distance (metric) and divergence (satisfying only (1)) between probability distributions P, Q .

(1) $d(P, Q) \geq 0$ and furthermore $d(P, Q) = 0 \Leftrightarrow P = Q$ almost everywhere.

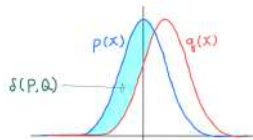
(2) $d(P, Q) = d(Q, P)$

(3) $d(P, Q) \leq d(P, R) + d(R, Q)$

- Total Variation distance**

$$\delta(P, Q) = \sup_A |P(A) - Q(A)| = \frac{1}{2} \|P - Q\|_1$$

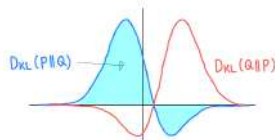
where $P(A) = \int_A p(x) dx$.



- Kullback-Leibler divergence**

$$D_{\text{KL}}(P \parallel Q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx = \mathbb{E}_{x \sim P} \left[\log \frac{p(x)}{q(x)} \right]$$

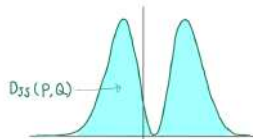
which is asymmetric and ≥ 0 (possibly infinite).



- Jensen-Shannon divergence**

$$D_{\text{JS}}(P, Q) = \frac{1}{2} D_{\text{KL}}\left(P \parallel \frac{P+Q}{2}\right) + \frac{1}{2} D_{\text{KL}}\left(Q \parallel \frac{P+Q}{2}\right)$$

which is symmetric and cannot be infinite.

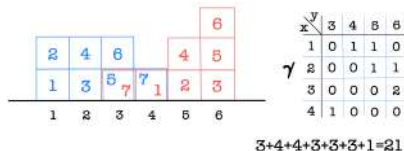
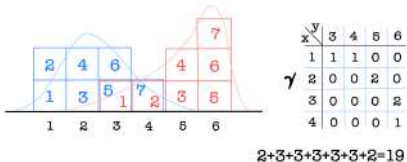


* KL divergence and JS divergence are not metrics.

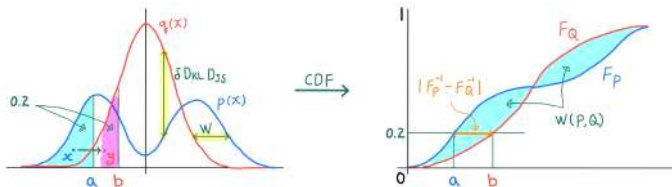
- Wasserstein (Earth-mover) distance

$$W(P, Q) = \inf_{\gamma \in \Pi(P, Q)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|] = \int_0^1 |F_P^{-1}(y) - F_Q^{-1}(y)| dy$$

- $\Pi(P, Q)$ is the set of all joint distributions $\gamma(x, y)$ with marginals P and Q respectively. Intuitively, $\gamma(x, y)$ indicates how much ‘mass’ must be transported from x to y (i.e., $\|x - y\|$) in order to transform P into Q . So, $W(P, Q)$ is the cost of the optimal transport plan.



- $F_P(x) = P(X \leq x)$ is CDF (cumulative distribution function) of P .



Theorem 1. $\{P_n\}_{n \in \mathbb{N}}$: sequence of distributions

$$D_{\text{KL}}(P_n \| P) \rightarrow 0 \implies D_{\text{JS}}(P_n, P) \rightarrow 0 \iff \delta(P_n, P) \rightarrow 0 \implies W(P_n, P) \rightarrow 0 \iff P_n \xrightarrow{D} P$$

Theorem 2. P_r : fixed real data distribution, g_θ : FFNN parameterized by θ

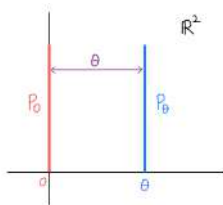
P_θ : distribution of $g_\theta(Z)$ with a prior $p(z)$ such that $\mathbb{E}_{z \sim p(z)}[\|z\|] < \infty$ (e.g. Gaussian, uniform)

Then $W(P_r, P_\theta)$ is **continuous** everywhere on θ and **differentiable** almost everywhere.

Ex) P_θ is the distribution of $(\theta, Z) \in \mathbb{R}^2$ ‘distributions with disjoint supports’
with $Z \sim U[0, 1]$ the uniform distribution on the unit interval.

$$D_{\text{KL}}(P_0 \| P_\theta) = \begin{cases} +\infty & \text{if } \theta \neq 0 \\ 0 & \text{if } \theta = 0 \end{cases} \quad D_{\text{JS}}(P_0, P_\theta) = \begin{cases} \log 2 & \text{if } \theta \neq 0 \\ 0 & \text{if } \theta = 0 \end{cases}$$

$$\delta(P_0, P_\theta) = \begin{cases} 1 & \text{if } \theta \neq 0 \\ 0 & \text{if } \theta = 0 \end{cases} \quad W(P_0, P_\theta) = |\theta|$$



- * When $\theta_t \rightarrow 0$, the sequence $\{P_{\theta_t}\}_{t \in \mathbb{N}}$ converges to P_0 only under Wasserstein distance.
- * In this example, we learn a probability distribution by doing gradient descent on Wasserstein distance.
This cannot be done with the other divergences since the resulting loss function is not even continuous.

Wasserstein GAN

- P_r is the real data distribution and P_θ is the parametric generated model distribution.

Difficulty is that the **infimum** in $W(P_r, P_\theta) = \inf_{\gamma \in \Pi(P_r, P_\theta)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$ is highly intractable.

- Using Kantorovich-Rubinstein duality, $W(P_r, P_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim P_r}[f(x)] - \mathbb{E}_{x \sim P_\theta}[f(x)]$ (*)

where supremum is over all 1-Lipschitz functions $f: X \rightarrow \mathbb{R}$, satisfying $|f(x_1) - f(x_2)| \leq |x_1 - x_2|$.

- Such a 1-Lipschitz function is learned with DNN similar to discriminator, which is just without sigmoid and outputs a scalar score rather than a probability.

For a parametrized $\{f_w\}_{w \in \mathcal{W}}$ all satisfying the Lipschitz constraint, we instead solve

$$\max_{w \in \mathcal{W}} \mathbb{E}_{x \sim P_r}[f_w(x)] - \mathbb{E}_{z \sim p(z)}[f_w(g_\theta(z))] \quad \text{WGAN critic (or Discriminator) } f_w$$

where a parametric function $g_\theta: Z \rightarrow X$ that directly generates samples following P_θ .

- * Network f_w is called ‘critic’ since it’s not trained to classify, but maximizes the value function as in DRL.
- * To enforce the Lipschitz constraint, WGAN clips the weights of f_w to lie within a compact space $[-c, c]$.
- * To minimize $W(P_r, P_\theta)$, we differentiate it by backpropagation through (*) via estimating

$$\nabla_\theta W(P_r, P_\theta) = -\mathbb{E}_{z \sim p(z)} [\nabla_\theta f_w(g_\theta(z))] \quad \text{Generator } g_\theta.$$

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α , the learning rate. c , the clipping parameter. m , the batch size. n_{critic} , the number of iterations of the critic per generator iteration.

Require: : w_0 , initial critic parameters. θ_0 , initial generator's parameters.

1: **while** θ has not converged **do**

2: **for** $t = 0, \dots, n_{\text{critic}}$ **do**

3: Sample $\{x^{(i)}\}_{i=1}^m \sim \mathbb{P}_r$ a batch from the real data.

4: Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$ a batch of prior samples.

5: $g_w \leftarrow \nabla_w \left[\frac{1}{m} \sum_{i=1}^m f_w(x^{(i)}) - \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)})) \right]$

6: $w \leftarrow w + \alpha \cdot \text{RMSPProp}(w, g_w)$

7: $w \leftarrow \text{clip}(w, -c, c)$

8: **end for**

9: Sample $\{z^{(i)}\}_{i=1}^m \sim p(z)$

10: $g_\theta \leftarrow -\nabla_\theta \frac{1}{m} \sum_{i=1}^m f_w(g_\theta(z^{(i)}))$

11: $\theta \leftarrow \theta - \alpha \cdot \text{RMSPProp}(\theta, g_\theta)$

12: **end while**

Compare to regular GAN

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

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

Information Maximizing GAN (InfoGAN)

- Regular GAN can generate realistic images, but cannot control the types of generated images.
 - * Conditional GAN can control the types (data labels), but it needs training data with extra information y such as class labels 0~9 for MNIST data.
- InfoGAN, to control the types of generated images, has additional information (latent code) in the input latent variable for G and makes fake images by maximizing mutual information between the latent codes and the generated images. \Leftarrow Information theory
- Cost function with information regularization helps to learn disentangled representations in unsupervised learning, competing with representations learned by supervised learning models.
 - * InfoGAN successfully learns to generate images with specific categorical features (digits 0~9) and continuous features (thickness or rotational angle of the digits).
- InfoGAN adopts variational information maximization (regularization) using a lower bound of the mutual information objective that can be optimized efficiently.
 - * InfoGAN adds only negligible computation cost on top of GAN and is easy to train.

Regular GAN

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

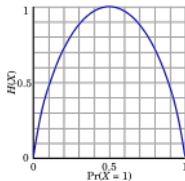
- Regular GAN uses **unfactored** continuous latent variables (noise vectors), while it does not limit how the generator can use these latent variables.
- Therefore a latent variable z may be used in a **highly entangled way** by the generator, and the individual entries of z do not correspond to semantic features of the data.

Purpose of InfoGAN

- When generating images from MNIST, it would be ideal if the model automatically chose to allocate a discrete latent variable to represent the numerical identity of the digit (0~9), and chose to have two additional continuous variables representing digit's thickness and angle.
- It is the case that these attributes are both **independent** and **prominent**, and it would be useful if we recover these concepts without any supervision (unsupervised learning unlike cGAN), by simply specifying that a digit is generated by an independent 1-of-10 variable and two independent continuous variables.

Information Theory

- Idea: unlikely events (rare words in a text) have more information than likely events (common words: high probability of showing up).
 - * To detect authorship of a book, words ‘deep’ and ‘learning’ are more useful than words ‘the’ and ‘it’.
- $I(x) = \log \frac{1}{p(x)} = -\log p(x)$ **self-information** (likely events have low information content.)
 $H(X) = \mathbb{E}_{x \sim P(X)}[I(x)] = -\mathbb{E}_{x \sim P(X)}[\log p(x)] = -\sum_x p(x) \log p(x)$ **entropy** (information measure)
 $H(X|Y) = \mathbb{E}_{y \sim P(Y)}[H(X|y)] = -\mathbb{E}_{y \sim P(Y)}[\mathbb{E}_{x \sim P(X|y)}[\log p(x|y)]]$ **conditional entropy**
 - * $H(X) \leq \log n$ for X with n events
 Equality holds iff X is uniformly distributed $p(x) = \frac{1}{n}$ (most unpredictable).
 - * Tossing coins with head outcome probability p
 $H(X) = -p \log_2 p - (1-p) \log_2 (1-p)$
- $I(X; Y) = D_{\text{KL}}(P(X, Y) \| P(X)P(Y)) = H(X) - H(X|Y)$ **Mutual information**
 - * $I(X; Y)$ measures the amount of information learned from knowledge of Y about X , or extra information needed to represent the joint distribution with the independent factorization.
 If X and Y are independent (and hence observing Y tells you nothing about X), then $I(X; Y) = 0$.
 - * In Machine learning, mutual information performs **feature selection** so that for a given feature (input), $I(\text{feature}; \text{labels (target)})$ is high, then the feature is a strong indicator of the labels, so we keep it.



Mutual information for inducing latent codes (minimax with information regularization)

$$\min_G \max_D L_{\text{IR}}(D, G) = L_{\text{GAN}}(D, G) - \lambda I(c; G(z, c))$$

- Instead of unfactored noise vectors, use input latent variables decomposed into two parts.
 - **noise vector** z : traditional input noise vector for regular GAN
 - **latent code** c : extra input vector for targeting the prominent structured semantic features
 - * c is factored by small latent codes c_1, \dots, c_L (concatenation), assuming $P(c_1, \dots, c_L) = \prod_i P(c_i)$.
- To discover latent code c in an unsupervised way, we provide the generator $G(z, c)$ of z and c , and use an **information-theoretic regularization inducing high mutual information** $I(c; G(z, c))$ between c and the generator distribution $G(z, c)$.
- By maximizing $I(c; G(z, c))$, the information in c should not be lost in the generation process so that c contains as much important and meaningful features of the real samples as possible.

InfoGAN (minimax with variational information maximization regularization)

$$\min_{G, Q} \max_D L_{\text{InfoGAN}}(D, G, Q) = L_{\text{GAN}}(D, G) - \lambda V(G, Q)$$

- In practice, $I(c; G(z, c))$ is hard to maximize directly as it requires access to the posterior $P(c | x)$.
- Variational information maximization** technique instead uses a lower bound of $I(c; G(z, c))$, which approximates $P(c | x)$ by a variational (auxiliary) distribution $Q(c | x)$.

$$\begin{aligned} I(c; G(z, c)) &= H(c) - H(c | G(z, c)) = \mathbb{E}_{x \sim G(z, c)} [\mathbb{E}_{c' \sim P(c | x)} [\log P(c' | x)]] + H(c) \\ &= \mathbb{E}_{x \sim G(z, c)} [D_{\text{KL}}(P(\cdot | x) \| Q(\cdot | x)) + \mathbb{E}_{c' \sim P(c | x)} [\log Q(c' | x)]] + H(c) \\ &\geq \mathbb{E}_{x \sim G(z, c)} [\mathbb{E}_{c' \sim P(c | x)} [\log Q(c' | x)]] + H(c) \quad \text{since } D_{\text{KL}} \geq 0 \\ &= \mathbb{E}_{c \sim P(c), x \sim G(z, c)} [\log Q(c | x)] + H(c) = V(G, Q) \quad (\text{Lemma 5.1 in the paper}) \end{aligned}$$

- Approximate $V(G, Q)$ with Monte Carlo simulation, and treat $H(c)$ as a constant by fixing $P(c)$. In particular, it can be maximized w.r.t. Q directly and w.r.t. G via the reparametrization trick. Hence $V(G, Q)$ can be added to GAN's objectives with no change to GAN's training procedure.

InfoGAN implementation

- InfoGAN networks : discriminator D , generator G and recognition network Q
 - * For MNIST training, D and Q share convolutional layers and Q has one more FC layer to output $Q(c|x)$.
- Therefore, InfoGAN adds only negligible computation cost on top of GAN and is easy to train. It is also observed that $V(G, Q)$ always converges faster than regular GAN objectives.
- Input of the generator G : noise vector z , categorical latent code c_i and continuous latent code c_j .
 - * For MNIST training, $z \sim N(0, 1)$, c_i is uniformly one of integers 0~9 (one-hot code), and $c_j \sim U(-1, 1)$.
The generator of InfoGAN is similar to that of cGAN, but c_i is not known and discovered by training.
- For categorical c_i , use the softmax nonlinearity to represent $Q(c_i|x)$.
For continuous c_j , simply treating $Q(c_j|x)$ as a factored Gaussian is sufficient.
 - * So, for each continuous c_j , Q outputs two nodes for the mean and the standard deviation.

discriminator D / recognition network Q	generator G
Input 28×28 Gray image	Input $\in \mathbb{R}^{74}$ $z \in \mathbb{R}^{62}$, ten c_i (1-hot) and two c_j
4×4 conv. 64 IRELU. stride 2	FC. 1024 RELU. batchnorm
4×4 conv. 128 IRELU. stride 2. batchnorm	FC. $7 \times 7 \times 128$ RELU. batchnorm
FC. 1024 IRELU. batchnorm	4×4 upconv. 64 RELU. stride 2. batchnorm
FC. output layer for D , FC.128-batchnorm-IRELU-FC.output for Q	4×4 upconv. 1 channel

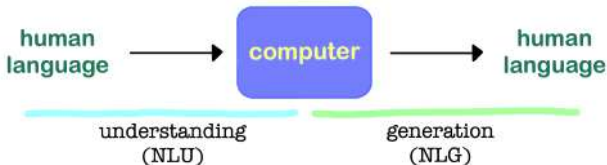
leaky RELU (0.1)

MNIST training
based on DCGAN

Natural Language Processing

Natural Language Processing (NLP)

NLP is a subfield of computational linguistics, computer science, and artificial intelligence concerned with the interactions between computers and human language, in particular how to program computers to process and analyze large amounts of natural language data.



NLP deals with speech recognition, natural language understanding (NLU), and natural language generation (NLG).

NLP applications

- Machine Translation
- Sentiment Analysis
- Question Answering
- Information Retrieval
- Information Extraction

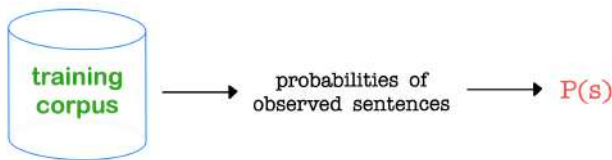


NLP is difficult due to its complexity.

- Words can have several meanings and contextual information is necessary to interpret sentences.
- The grammar rules of natural language are not easy for computers to understand.
- There is an infinite number of different ways to arrange words in a sentence.

Statistical Language Model (SLM) 1990s~2010s

- SLM (or Language model) assigns a **probability distribution $P(s)$ over sentences**
 $s = w_1 w_2 \cdots w_n$ (sequences of words) in a training corpus.
- * For example, in Speech recognition, we may use the probability $P(\text{I drive a car.}) > P(\text{I drive a cup.})$



- $$P(s) = P(w_1 w_2 \cdots w_n) = P(w_1)P(w_2 | w_1) \cdots P(w_n | w_1 w_2 \cdots w_{n-1}) = \prod_{i=1}^n P(w_i | h_{i-1})$$

where $h_{i-1} = w_1 w_2 \cdots w_{i-1}$.

We need to find all possible cases: $P(w_i | h_{i-1}) = \frac{\text{count}(h_{i-1} w_i)}{\text{count}(h_{i-1})} \Leftarrow$ **counting method**

- Curse of dimensionality:** $|V|^n$ combinations of n words for Vocabulary V (need a huge corpus)
- Sparsity problem:** long h_{i-1} and $h_{i-1} w_i$ may not be seen in the corpus (need to limit h_{i-1} length)

n -gram: classic approach by limiting h_{i-1} to small $n-1$ preceding words

- **Uni-gram** $P(s) = \prod_{i=1}^n P(w_i)$
- **Bi-gram** $P(s) = \prod_{i=1}^n P(w_i | w_{i-1})$
- **Tri-gram** $P(s) = \prod_{i=1}^n P(w_i | w_{i-2} w_{i-1})$

corpus of 10,000 words (10,000 bi-grams, # space)

w_i	$P(w_i)$	w_{i-1}	$w_{i-1} w_i$	$P(w_i w_{i-1})$
I (10)	0.001	# (1000)	# I (7)	0.007
		that (10)	that I (3)	0.3
talk (8)	0.0008	I (10)	I talk (2)	0.2
		we (5)	we talk (1)	0.2
talks (7)	0.0007	he (9)	he talks (3)	0.333
		she (4)	she talks (1)	0.25

- **Uni-gram**
 $P(\text{I talk}) = P(\text{I}) P(\text{talk}) = 0.001 \times 0.0008 = 0.0000008$
 $P(\text{talk I}) = P(\text{talk}) P(\text{I}) = 0.0008 \times 0.001$
 $P(\text{I talks}) = P(\text{I}) P(\text{talks}) = 0.001 \times 0.0007$
- **Bi-gram**
 $P(\text{I talk}) = P(\text{I}) P(\text{talk} | \text{I}) = 0.001 \times 0.2 = 0.0002$
 $P(\text{I talks}) = P(\text{I}) P(\text{talks} | \text{I}) = 0.001 \times 0.0 = 0$

- n -gram for larger n has better probabilities (grammatically valid sentences).
But, it still suffers from the curse of dimensionality and sparsity problem.
- In Brown Corpus, ‘He was happy.’ appears 6 times and ‘She was joyful.’ appears 0 times.
Using 3-gram, $P(\text{She was joyful.}) = 0$ even though two sentences are very similar.

Distributional Semantic Model

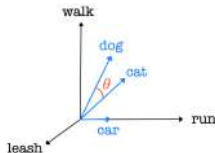
- Idea: **Distributional Hypothesis** ‘Similar words occur in similar contexts.’ (Harris, 1954)
 - Goal: quantify **semantic similarities** (in terms of **vector similarities**) between words based on their distributional properties in the corpus. \Leftarrow **bag-of-words** model
 - * ‘Bag of words’ is an orderless document representation; only the counts of words matter.
- preprocess the source corpus (target, context).
 - build the target–context co-occurrence matrix.
(each row = distributional vector representing a word)
 - transform the matrix: re-weighting, dimensionality reduction.
 - use the resulting matrix to compute word-to-word similarity.

	leash	walk	run	owner	bark
dog	3	5	2	5	2
cat	0	3	3	2	0
light	0	0	0	0	0
bark	1	0	0	2	5
car	0	0	1	3	0

The dog barked in the park.
The owner of the dog put him
on the leash since he barked.

similarity score

$$\cos(\theta) = \frac{w_1 \cdot w_2}{|w_1| |w_2|}$$



- If words in two sentences have high similarity score, then their probabilities are similar. Eventually, neural networks can improve the performance.

Vector representations of words

Sparse vector representation \Leftarrow one-hot encoding

- One-hot code is a vector with size $|V|$ having a single 1 and all the others 0 for Vocabulary V .
- No information about word similarity (completely independent): $\mathbf{w}_{\text{dog}} \cdot \mathbf{w}_{\text{cat}} = \mathbf{w}_{\text{dog}} \cdot \mathbf{w}_{\text{car}} = 0$

dog : $\mathbf{w}_{\text{dog}} = [1, 0, \dots, 0, 0]$

cat : $\mathbf{w}_{\text{cat}} = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \dots, 0, 0]$

car : $\mathbf{w}_{\text{car}} = [0, 0, 0, 0, 0, 0, 1, 0, \dots, 0, 0]$

Dense (distributed) vector representation \Leftarrow word embedding

- Word embedding is an embedding from the vocabulary space V to a continuous vector space with a much lower dimension R^N (for $N \ll 13M$) so that each dimension encodes some meaning that we transfer using speech: size, gender, etc.
- Intuition: there exists a smaller N -dimensional space that is sufficient to encode all semantics of our language including relationship or similarity between words.

dog : $\mathbf{w}_{\text{dog}} = [0.08, 0.31, 0.59, 0.47, 0.22, 0.31, 0.28]$

cat : $\mathbf{w}_{\text{cat}} = [0.02, 0.29, 0.49, 0.11, 0.12, 0.21, 0.56]$

car : $\mathbf{w}_{\text{car}} = [0.83, 0.21, 0.09, 0.27, 0.92, 0.01, 0.77]$

$$\frac{\mathbf{w}_{\text{cat}} \cdot \mathbf{w}_{\text{dog}}}{|\mathbf{w}_{\text{cat}}| |\mathbf{w}_{\text{dog}}|} > \frac{\mathbf{w}_{\text{cat}} \cdot \mathbf{w}_{\text{car}}}{|\mathbf{w}_{\text{cat}}| |\mathbf{w}_{\text{car}}|}$$

Word2vec

Word2vec by Mikolov at Google is a software package for word embedding.

- Word2vec represents ⁽¹⁾each word by a low-dimensional vector with the property ⁽²⁾‘word similarity = vector similarity’.

These vectors (word embeddings) are used as input in neural networks for NLP.

- Word2vec is a FC 2-layer NN.

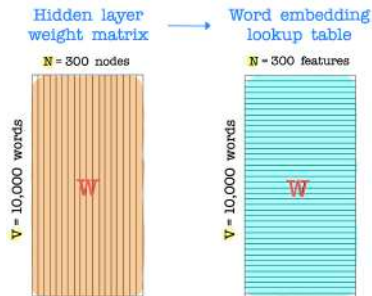
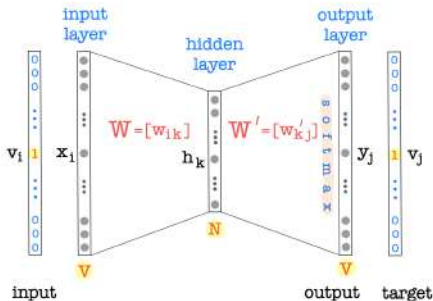
It takes as its input a large corpus of text and produces a word vector space of dim 100 ~ 1000.

If words share common contexts in the corpus, their word vectors are located closely.

- Network models: CBOW, Skip-gram
- Optimization techniques: Hierarchical softmax, Negative sampling
- Preprocessing pipelines: Dynamic context window, Subsampling, Deleting rare words

Simple CBOW like bi-gram model $P(v_{\text{output}} | v_{\text{input}})$

- Two fully connected layers (linear, linear+softmax) are used.
- $\mathbf{x}_i = [x_1, x_2, \dots, x_V]$ (only $x_i = 1$) one-hot encoded input of i th word v_i in Vocabulary V
- $W = [w_{ik}]_{V \times N}$, $W' = [w'_{kj}]_{N \times V}$ weight matrices with the hidden layer size N
- $\mathbf{h} = \mathbf{x}_i W = W_{(i, \cdot)} \leftarrow i$ th row of $W \leftarrow$ **word embedding** for v_i
- $\mathbf{u} = \mathbf{h} W'$ or $u_j = \mathbf{h} W'_{(\cdot, j)} = \sum_{k=1}^N h_k w'_{kj}$
- $P(v_j | v_i) = y_j = \frac{e^{u_j}}{\sum_{t=1}^V e^{u_t}}$ 'softmax'



Update parameters using **Gradient descent**

$$\begin{array}{ccccc} \{x_i\} & \longrightarrow & \{h_k\} & \longrightarrow & \{u_j\} & \longrightarrow & \{y_j\} \\ h = x_i W & \parallel & u = h W' & \parallel & \text{softmax} & & \\ & \sum x_i w_{ik} & & \sum h_k w'_{kj} & & & \end{array}$$

- **Objective:** maximize $P(v_{\text{output}} | v_{\text{input}})$

$$\min E = -P(v_O | v_I) = -y_{j^*} \text{ or } -\log y_{j^*} = -u_{j^*} + \log \sum_{t=1}^V e^{u_t}$$

where j^* is the index of the actual output word v_O .

$$\frac{\partial E}{\partial u_j} = y_j - t_j := e_j \text{ where } t_{j^*} = 1 \text{ and other } t_j = 0$$

↙ prediction error

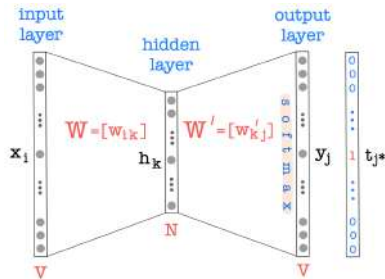
$$\frac{\partial E}{\partial w'_{kj}} = \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial w'_{kj}} = e_j h_k \text{ since } u_j = \sum_{k=1}^N h_k w'_{kj}$$

$$\Rightarrow w'_{kj}^{(\text{new})} = w'_{kj}^{(\text{old})} - \eta e_j h_k \text{ (hidden-to-output weights } W')$$

↙ learning rate

$$\frac{\partial E}{\partial w_{ik}} = \sum_{j=1}^V \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial h_k} \frac{\partial h_k}{\partial w_{ik}} = x_i \sum_{j=1}^V e_j w'_{kj} \text{ since } h_k = \sum_{i=1}^V x_i w_{ik}$$

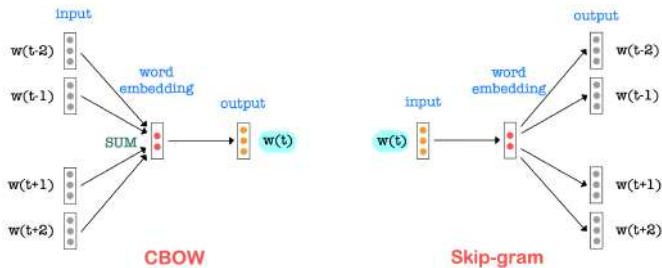
$$\Rightarrow w_{ik}^{(\text{new})} = w_{ik}^{(\text{old})} - \eta x_i \sum_{j=1}^V e_j w'_{kj} \text{ (input-to-hidden weights } W)$$



- Big issue: if $V = 10,000$ and $N = 300$, then we have to update $6M$ weights.

Word2vec has 2 model architectures to produce word embeddings.

- **CBOW** (continuous bag-of-words) predicts the current word from surrounding context words. The context words order doesn't influence prediction (bag-of-words).
- **Skip-gram** uses the current word to predict surrounding context words. It weighs nearby context words more heavily than distant context words.
- Goal is learning **input-to-hidden weights W** which are indeed '**word embeddings**'.



- CBOW is faster while Skip-gram is slower but works better for infrequent words.
- Recommended context window size is 5 for CBOW or 10 for Skip-gram.

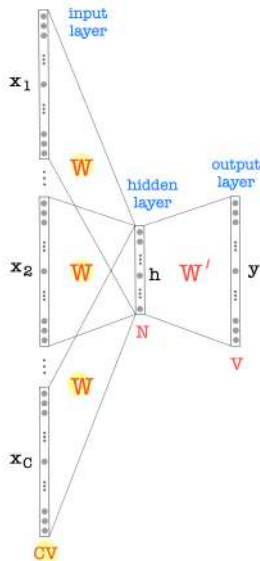
CBOV

- Generate our one-hot encodings $\{x_1, \dots, x_C\}$ of the input word v_{I_1}, \dots, v_{I_C} of window size C .
- Get embedded word vectors $\{h_1, \dots, h_C\}$ by $h_c = x_c W$, sharing the same weights W .
- Take average $h = \frac{h_1 + \dots + h_C}{C} = \frac{1}{C} \sum_{c=1}^C x_c W$.

The context words order does not influence prediction.

- Generate a score vector $u = h W'$.
- Turn the scores into probabilities $y = \text{softmax}(u)$.
- We desire y to match the one-hot encoding of the actual output word v_O , by minimizing the loss function

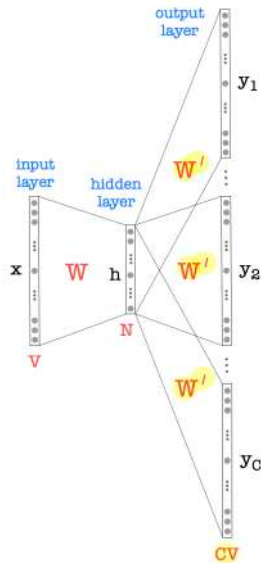
$$E = -\log P(v_O | v_{I_1}, \dots, v_{I_C}).$$



Skip-gram

- Generate our one-hot encoding \mathbf{x} of the input word v_I .
- Get an embedded word vector $\mathbf{h} = \mathbf{x}W$.
- Generate a score vector $\mathbf{u} = \mathbf{h}W'$.
- Turn the scores into probabilities $\mathbf{y} = \text{softmax}(\mathbf{u})$.
- We desire \mathbf{y} to match the C many one-hot encodings of the actual output words v_{O_1}, \dots, v_{O_C} , by minimizing

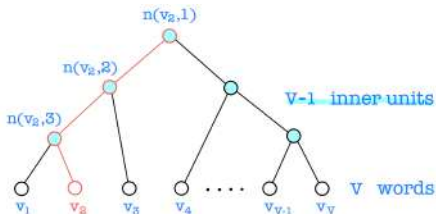
$$\begin{aligned} E &= -\log P(v_{O_1}, \dots, v_{O_C} | v_I) \\ &= -\log \prod_{c=1}^C \frac{e^{u_{j_c^*}}}{\sum_{t=1}^V e^{u_t}} \\ &= -\sum_{c=1}^C u_{j_c^*} + C \log \sum_{t=1}^V e^{u_t}. \end{aligned}$$



Hierarchical Softmax

- From a binary tree to represent all words in V , we use the unique path from the root to each word v_j to estimate the probability $P(v_j = v_O | v_I)$ of each v_j being the output word v_O .
- This is done directly from embedded word vectors $\mathbf{h} = \mathbf{x}W$ for one-hot encoding \mathbf{x} of v_I , not following the usual output layer multiplication $\mathbf{u} = \mathbf{h}W'$ and softmax.

$$P(v_j = v_O | v_I) = \prod_{s=1}^{L(v_j)-1} \sigma(\pm \mathbf{h} \cdot \mathbf{v}'_{n(v_j,s)}) \text{ where } + \text{ for left move and } - \text{ for right.}$$



- $L(v_2) = 4$ path length to v_2 ($L(v_j) \approx \log V$)
- $n(v_2, s)$ s th unit from the root to v_2
- $u_s = \mathbf{h} \cdot \mathbf{v}'_{n(v_j,s)}$ (instead of $u_j = \mathbf{h}W'_{(\cdot,j)}$)
where $\mathbf{v}'_{n(v_j,s)}$ is an 'output vector' with $\text{dim}=N$
- $P(v_2 = v_O | v_I) = \sigma(u_1) \sigma(u_2) \sigma(-u_3)$
- $\sum_{j=1}^V P(v_j = v_O | v_I) = 1$ since $\sigma(x) + \sigma(-x) = 1$

- There is no output vector $W'_{(\cdot,j)}$ for v_j . Instead, each inner unit has an output vector $\mathbf{v}'_{n(v_j,s)}$. Therefore we only update $\mathbf{v}'_{n(v_j,s)}$, instead of W' .

- Loss function $E = -\log P(v_j = v_O | v_I) = -\sum_{s=1}^{L(v_j)-1} \log \sigma(\pm u_s)$
 $\frac{\partial E}{\partial u_s} = \pm (\sigma(\pm u_s) - 1) = \sigma(u_s) - t_s$ where $t_s = 1$ for + (left) and 0 for - (right).
 $\frac{\partial E}{\partial \mathbf{v}'_{n(v_j, s)}} = \frac{\partial E}{\partial u_s} \frac{\partial u_s}{\partial \mathbf{v}'_{n(v_j, s)}} = (\sigma(u_s) - t_s) \mathbf{h}$
 $\Rightarrow \mathbf{v}'_{n(v_j, s)}^{(\text{new})} = \mathbf{v}'_{n(v_j, s)}^{(\text{old})} - \eta (\sigma(u_s) - t_s) \mathbf{h}$ for $s = 1, \dots, L(v_j)-1$.
 $\mathbf{w}_{(i, \cdot)}^{(\text{new})} = \mathbf{w}_{(i, \cdot)}^{(\text{old})} - \eta x_i \sum_{s=1}^{L(v_j)-1} (\sigma(u_s) - t_s) \mathbf{v}'_{n(v_j, s)}$ for weights W .
- The task for each inner unit is to predict whether it moves to the left or right.
 $\sigma(u_s)$ is the prediction, and $t_s = 1$ (0) means that the truth is to move left (right).
- This equation can be used for both CBOW and Skip-gram models.
 For Skip-gram, we apply this equation for one context word at a time.
- Parameters (in the output layer) to be updated per iteration: $N \times V \Rightarrow N \times (V-1)$
- Computation per word: $N \times V \Rightarrow N \times \log V$
 - Softmax $P(v_j | v_I) = y_j = \frac{e^{u_j}}{\sum_{t=1}^V e^{u_t}}$ where $u_t = \sum_{k=1}^N h_k w'_{kt}$.
 - Hierarchical Softmax $P(v_j = v_O | v_I) = \prod_{s=1}^{L(v_j)-1} \sigma(\pm u_s)$ where $u_s = \mathbf{h} \cdot \mathbf{v}'_{n(v_j, s)}$.

Negative Sampling

- In each iteration, instead of updating all output vectors $\mathbf{v}'_j = \mathbf{W}'_{(\cdot, j)}$ for $j = 1, \dots, V$, which is \mathbf{W}' , we only update a sample of them.
- Our sample contains the output word v_O and small K negative sample words $v_{\text{neg}} = \{v_1, \dots, v_K\}$, which are randomly chosen by using the noise distribution $P_n(v)$. (K ranges in $5 \sim 20$.)
- In this sampling, we want the network to output 1/0 for the output/negative word.

Instead of softmax, we use sigmoid only for the $K+1$ words:

$$P(D=1 | v_I, v_O) = \sigma(u_O) = \frac{1}{1+e^{-u_O}} \text{ for the output word } v_O.$$

$$P(D=0 | v_I, v_j) = 1 - P(D=1 | v_I, v_j) = 1 - \sigma(u_j) = \sigma(-u_j) \text{ for } v_j \in v_{\text{neg}} \text{ where } u_j = \mathbf{h} \mathbf{W}'_{(\cdot, j)}.$$

$$E = -\log \left[P(D=1 | v_I, v_O) \prod_{v_j \in v_{\text{neg}}} P(D=0 | v_I, v_j) \right] = -\log \sigma(u_O) - \sum_{v_j \in v_{\text{neg}}} \log \sigma(-u_j)$$

$$\frac{\partial E}{\partial u_j} = \sigma(u_j) - t_j \text{ where } t_O = 1 \text{ and other } t_j = 0 \text{ since } \sigma' = \sigma(1 - \sigma).$$

$$\frac{\partial E}{\partial w'_{kj}} = \frac{\partial E}{\partial u_j} \frac{\partial u_j}{\partial w'_{kj}} = (\sigma(u_j) - t_j) h_k$$

$$\Rightarrow w'_{kj}^{(\text{new})} = w'_{kj}^{(\text{old})} - \eta (\sigma(u_j) - t_j) h_k \text{ for } j = O, 1, \dots, K.$$

$$w_{ik}^{(\text{new})} = w_{ik}^{(\text{old})} - \eta x_i \left[(\sigma(u_O) - 1) w'_{ko} + \sum_{j=1}^K \sigma(u_j) w'_{kj} \right] \text{ for weights } W.$$

- This equation can be used for both CBOW and Skip-gram models.
For Skip-gram, we apply this equation for one context word at a time.
- Parameters (in the output layer) to be updated per iteration: $N \times V \Rightarrow N \times (K+1)$
- Computation per word: $N \times V \Rightarrow N \times (K+1)$
- Noise distribution $P_n(v)$ used in word2vec is a unigram distribution raised to the $\frac{3}{4}$ th power for the best quality of results.

Ex. In corpus consisting of 100 words (90 v_1 , 9 v_2 , 1 v_3),

unigram distribution: $P(v_1) = 0.9$, $P(v_2) = 0.09$, $P(v_3) = 0.01$

unigram distribution $^{\frac{3}{4}}$: $P(v_1) = \frac{0.924}{1.12} = 0.825$, $P(v_2) = 0.146$, $P(v_3) = 0.029$

- Hierarchical Softmax works better for infrequent words.
Negative Sampling works better for frequent words and better with low-dimensional vectors.

Word2vec preprocessing pipelines

- **Dynamic context window**

Traditionally, the context window is unweighted and of a constant size.

To assigns more weight to closer words, window sizes are sampled uniformly between 1 ~ maximum.

- **Subsampling**

Randomly removing frequent words ('the', 'is') with frequency above some threshold because high frequency words often provide little information.

Keeping probability $P(w) = \left(\sqrt{\frac{z(w)}{0.001}} + 1 \right) \cdot \frac{0.001}{z(w)}$ where $z(w) = \frac{\text{appearances of } w}{\text{total number of words in the corpus}}$

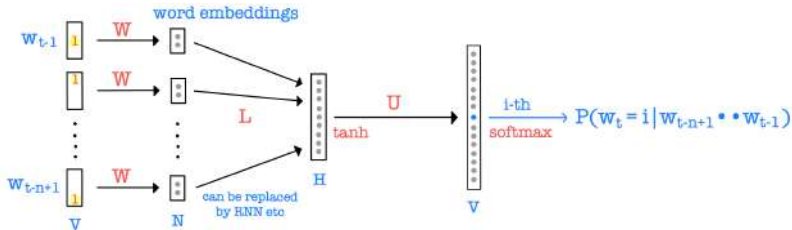
- **Deleting rare words**

Rare words are also deleted before creating the context windows.

Neural networks for NLP

Bengio used **feedforward neural network** in 2003 to improve **SLM 'n-gram'** by learning every $P(w_t | w_{t-n+1} \cdots w_{t-1})$ to overcome the curse of dimensionality of V^n .

- Generate one-hot encodings of the $n-1$ previous words $w_{t-n+1}, \dots, w_{t-1}$ in Vocabulary V .
- Get $n-1$ word embeddings of $\text{dim} = N$ by using a word embedding matrix $W_{V \times N}$ (lookup table) with learnable parameters shared across words (each row represents one word embedding).
- Concatenated them and feed it into a hidden layer $L_{(n-1)N \times H}$ followed by \tanh , and then the output layer $U_{H \times V}$ followed by softmax, outputting $P(w_t = i | w_{t-n+1} \cdots w_{t-1})$ which is the probability of the next word w_t being i th word in V .
- This model still performs well in some setups where only recent words need to be considered.



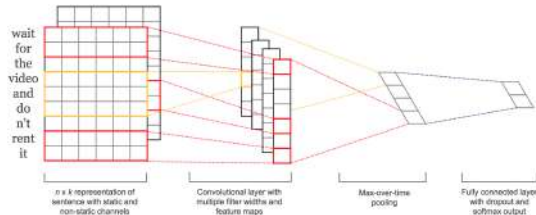
CNN on pre-trained word embeddings for sentence-level classification tasks operates in 2-dimension with the filters only moving along the temporal dimension.

- $\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \dots \oplus \mathbf{x}_n$ is a sentence of n words (word embeddings) $\mathbf{x}_i \in \mathbb{R}^N$.

- Apply a filter $\mathbf{w} \in \mathbb{R}^{kN}$ to all k consecutive word sets $\{\mathbf{x}_{1:k}, \mathbf{x}_{2:k+1}, \dots, \mathbf{x}_{n-k+1:n}\}$ producing a feature map $\mathbf{c} = [c_1, \dots, c_{n-k+1}]$ where $c_i = f(\mathbf{w} \cdot \mathbf{x}_{i:i+k-1} + b)$.

- Global max pooling per filter (temporal)
 $\hat{c} = \max\{\mathbf{c}\}$ for varying sentence length.

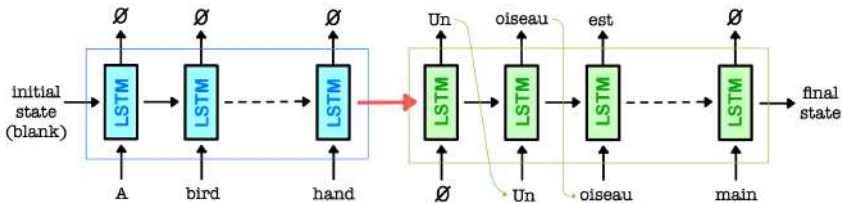
One feature is extracted from one filter.



- In CNN model, the state at every feature map only depends on the local context according to the size of receptive field via convolutions rather than all past states as in RNN.
- CNN works in parallel since each word on the input can be processed at the same time.

Sequence-to-sequence (Seq2seq)

- **Seq2Seq** learning is a framework for mapping one sequence to another sequence.
- Encoder LSTM processes an input sentence word by word in word embedding form and compresses the entire content of the input sequence into a small fixed-size vector.
- Decoder LSTM predicts the output word by word based on the encoded vector, taking the previously predicted word as input at every step.
- * Encoder network reads in English (blue) and Decoder network writes in French (green).



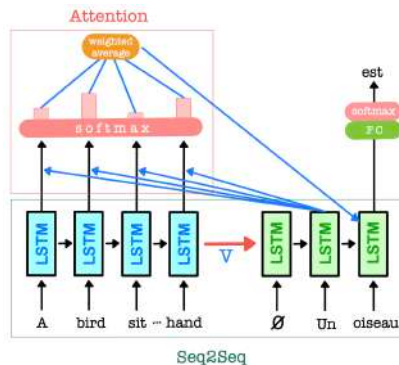
Attention LSTM

Attention mechanism is one of the core innovations in Machine Translation.

- Main bottleneck of Seq2Seq is that it requires to compress the entire content of the input sequence into a small fixed-size vector V .
- Attention (using query, key and value vectors) allows Decoder to look at Encoder hidden states, whose weighted average is an additional input to Decoder.

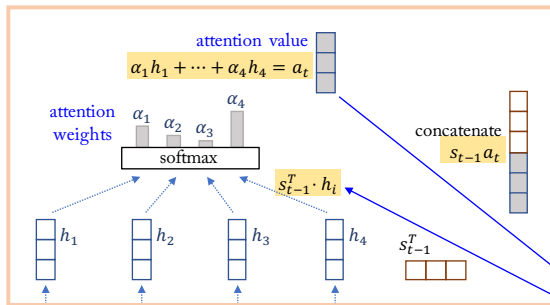
Decoder can pay attention to specific words in the input sequence again based on each hidden state of Decoder.

- It provides a glimpse (based on the attention weights) into the inner working of the model by inspecting which input parts are relevant for a particular output. (unveiling DNN black box)



Dot-product attention $s_{t-1}^T \cdot h_i$ with **query** s_{t-1} , **key** h_i and **value** h_i

\Rightarrow **attention weight** $\alpha_{ti} = \text{softmax}(s_{t-1}^T \cdot h_i)$ and **attention value** $a_t = \sum_j \alpha_{tj} h_j$



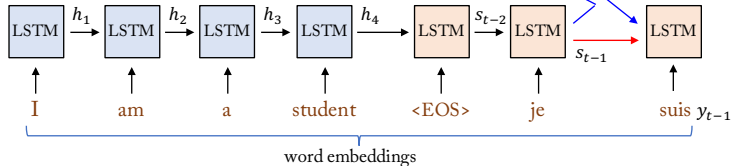
etudiant y_t

Softmax
FC

Attention LSTM
 $s_t = f(s_{t-1} a_t, y_{t-1})$

Standard LSTM
 $s_t = f(s_{t-1}, y_{t-1})$

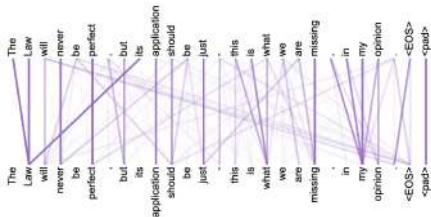
dot-prod
general
concat
with learnable W and v



Transformer

Transformer has an encoder-decoder structure and was developed to solve the **global dependency** and **parallel computing** problems of sequence transduction or neural machine translation (NMT).

- It relies solely on **self-attention** to learn global dependencies regardless of their distance in the input or output sequences **instead of using recurrent or convolutional layers**.
- Self-attention also allows parallel computing, and so Transformer can be trained significantly faster than RNN and CNN architectures.

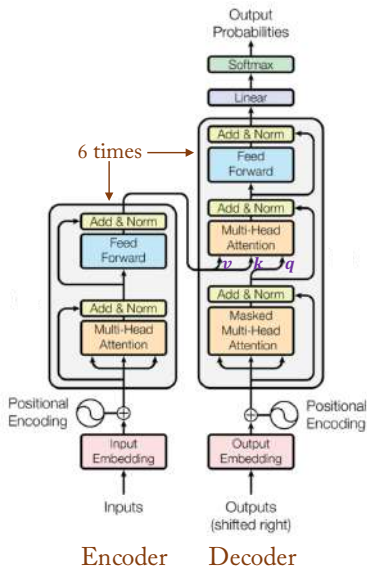


global dependencies in encoder self-attention in layer 5 of 6

- Transformer is used by OpenAI for language models and DeepMind for AlphaStar.

Transformer

- Encoder and decoder have 6 identical layers.
- Each encoder layer has 2 sub-layers
 - multi-head **self-attention**
 - **position-wise FFNN** (FC feedforward NN)
- Each decoder layer has 3 sub-layers
 - masked multi-head self-attention
 - multi-head **encoder-decoder attention**
 - position-wise FFNN
- Residual connection around every sub-layer, followed by layer normalization
 $\text{layerNorm}(x + \text{sublayer}(x))$
- **Input of each layer has the same dim=512**, which is the input word embedding dimension.
- **Positional encodings** to the input embeddings to make use of the order of the sequence.



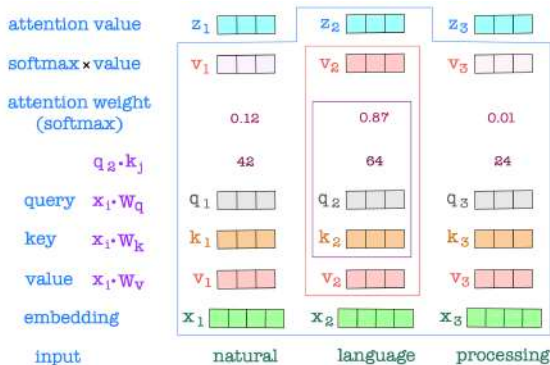
Self-Attention helps the encoder looking at other words in input sentence as it encodes a word.

- 3 vectors from the input vector x_i ($d_m=512$) which is the output of the previous layer.
 - query** vector $q_i = x_i W_q$ ($d_k=64$)
 - key** vector $k_i = x_i W_k$ ($d_k=64$)
 - value** vector $v_i = x_i W_v$ ($d_v=64$)
with 3 learnable W_q, W_k, W_v
without activations (i.e., linear)

- Scaled dot-product attention**

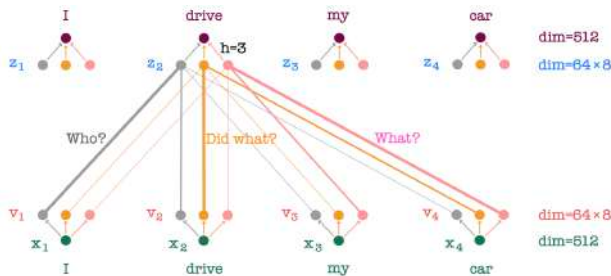
$$z_i = \sum_{j=1}^N \text{softmax}\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right) v_j \quad (d_v=64)$$

- Each position pays attention to all positions in the previous layer.
 - $q_i \cdot k_j$ is the relativity of x_i and x_j .
 - scaled by $\frac{1}{\sqrt{d_k}}$ to lead stable gradients (softmax)
- In encoder-decoder LSTM, query vector is usually the hidden state of the decoder, whereas key and value vectors are the hidden state of the encoder.



Multi-Head Attention

- Instead of a single attention with queries, keys and values ($d_m = 512$), we use $h = 8$ attentions (heads) with smaller queries, keys and values ($d_v = \frac{d_m}{h} = 64$) using weights W_q, W_k, W_v (different for each head, different from layer to layer, but the same across all word positions).
- These attentions independently learn in parallel, yielding h output values z_i^1, \dots, z_i^h with $d_v = 64$, which are concatenated and once again a FC layer without activations, resulting in final values with $d_m = 512$ for each word position i (position-wise from the input values to the final values).



$$[z_i^1; \dots; z_i^h] W_o$$

- * It pays attention jointly to information in different representations (heads) that average differently attention-weighted positions.

- In **masked multi-head attention** in the decoder, self-attention sub-layers allow each position to attend to all positions in the previous layer up to that position. To preserve the auto-regressive property (so, to prevent leftward information flow), we implement this inside of scaled dot-product attention by masking out all values (setting to $-\infty$) in the input of the softmax, which correspond to illegal connections to leftward positions.
- As the second multi-head attention in the decoder, **encoder-decoder attention** sub-layers use the queries coming from the previous decoder masked multi-head attention sub-layer, and the memory keys and values come from the final output of the encoder. So, every position in the decoder attends over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanism in Seq2seq.

Position-wise FFNN

- Each layer in the encoder and decoder contains a FC FFNN (after multi-head attention sub-layer), which is applied to each position separately and identically.
- This sub-layer consists of 2 FCs ($512 \rightarrow 2048 \rightarrow 512$) with ReLU in between:

$$\text{FFNN}(z) = \max(0, zW_1 + b_1)W_2 + b_2$$
- W_1 and W_2 are the same across different word positions (like 2 convolutions with kernel size 1), but obviously different from layer to layer.

Positional Encoding

- Since we use neither recurrence nor convolution, in order to recognize the sequence order, we must inject some information about the relative or absolute position of the words.
- **Positional encoding** PE_i with $d_m = 512$ is added to each input embedding x_i of word position i at the bottoms of the encoder and decoder stacks, so we use $x_i \leftarrow x_i + PE_i$ as a new embedding. The positional encoding can be learned or fixed.
- In this work, we use (fixed) sine and cosine functions of different frequencies

$$PE_i = [\sin(i \cdot d_1) \cos(i \cdot d_1) \cdots \sin(i \cdot d_{\frac{d_m}{2}}) \cos(i \cdot d_{\frac{d_m}{2}})] \text{ where } d_k = 1/10000^{\frac{2k}{d_m}}$$

- * Each sinusoid (sine wave) has a wavelength forming a geometric progression from 2π to $10000 \cdot 2\pi$.
- * It attends relative positions effortlessly since for any fixed t , PE_{i+t} is a linear function of PE_i .

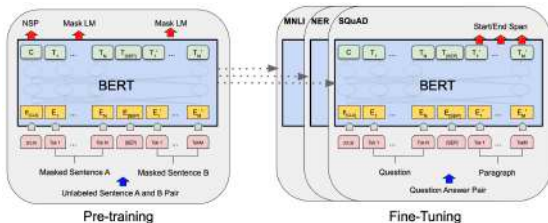
$$\begin{bmatrix} \sin(i \cdot d_k) & \cos(i \cdot d_k) \end{bmatrix} \begin{bmatrix} \cos(t \cdot d_k) & -\sin(t \cdot d_k) \\ \sin(t \cdot d_k) & \cos(t \cdot d_k) \end{bmatrix} = \begin{bmatrix} \sin((i+t) \cdot d_k) & \cos((i+t) \cdot d_k) \end{bmatrix}$$

- * Criteria for ideal positional encodings
 - It should output a unique encoding for each time-step (word's position in a sentence).
 - Distance between any two time-steps should be consistent across sentences with different lengths.
 - Our model should generalize to longer sentences without any efforts. Its values should be bounded.
 - It must be deterministic.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is **pre-training** of a multi-layer **bidirectional Transformer Encoder** for language understanding.

- BERT is designed to **pre-train** deep **bidirectional** contextual representations from **unlabeled** text (unsupervised learning) by jointly conditioning on both left and right context in all layers.
- Pre-trained BERT model can be **fine-tuned** (transfer learning) with one additional output layer (even for low-resource tasks) to create state-of-the-art models for a wide range of NLP tasks, without substantial task-specific architecture modifications.
- BERT is conceptually simple and empirically powerful, and obtains superior results on 11 NLP tasks.
- A distinctive feature of BERT is its unified architecture across different tasks.



BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, Google 2018

BERT architecture

- BERT architecture is a multi-layer bidirectional Transformer Encoder

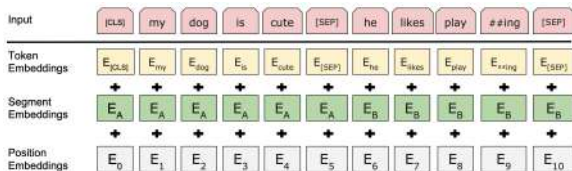
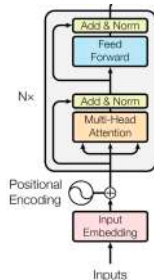
BERT_{base} 12 layers, $d_m=768$, $h=12$, total parameters=110M

BERT_{large} 24 layers, $d_m=1024$, $h=16$, total parameters=340M

- * BERT Transformer uses bidirectional self-attention.

OpenAI GPT Transformer uses constrained self-attention attending left only.

- To make BERT handling various downstream tasks, input representation is both a single sentence and a pair of sentences (for Q&A).
- We use WordPiece embeddings (30,000 vocabulary V) for token embedding sequence with the first token [CLS] and, when using a pair of sentences, the separating token [SEP].
- * For the pre-training corpus we use BooksCorpus (800M words) and English Wikipedia (2,500M words).

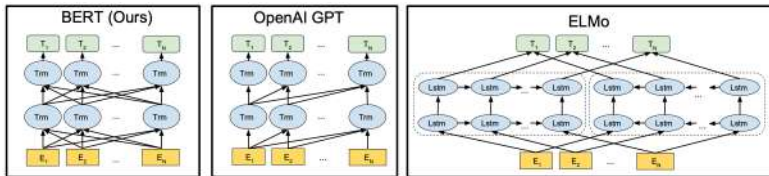


To each token embedding, we add a learned segment embedding indicating whether it belongs to sentence A or B, and also a positional embedding.

Pre-training using Masked LM (language model)

- Intuitively, a deep bidirectional model is more powerful than either a left-to-right model (OpenAI GPT using Transformer) or the shallow concatenation of a left-to-right and a right-to-left model (ELMo using dual LSTM).

Unfortunately, bidirectional conditioning would allow each word to indirectly ‘see itself’, and the model could trivially predict the target word in a multi-layered context.



- In order to train ‘bidirectional’, we simply **mask 15% of the input tokens at random**, and then **only predict those masked tokens** (rather than reconstructing the entire input). The final hidden vectors corresponding to the mask tokens are fed into output softmax over V .
- * Since [MASK] does not appear during fine-tuning, a downside mismatches pre-training and fine-tuning. If the i th token is chosen for [MASK], we replace it with 80% [MASK], 10% random, and 10% unchanged.

Pre-training using NSP (next sentence prediction)

- Downstream tasks (Q&A, Natural Language Inference) are based on understanding the relationship between two sentences, which is not directly captured by language modeling.
- In order to train this sentence relationship, we pre-train for a binarized NSP task by choosing a label 'IsNext' if the sentence B is the actual next sentence that follows A (50% in sample training) or 'NotNext' otherwise.
- The final hidden state C corresponding to [CLS] is used for NSP task as the aggregate sequence representation for classification tasks.

Fine-tuning

- Fine-tuning is straightforward since the self-attention mechanism in Transformer allows BERT to model many downstream tasks whether they involve single text or text pairs.
- For each downstream task, we simply plug in the task-specific inputs and outputs into BERT and **fine-tune all the parameters end-to-end** using labeled data.

Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters.

GLUE Test results

System	MNLI-(m/mm) 392k	QQP 363k	QNLI 108k	SST-2 67k	CoLA 8.5k	STS-B 5.7k	MRPC 3.5k	RTE 2.5k	Average
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT _{BASE}	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT _{LARGE}	86.7/85.9	72.1	92.7	94.9	60.5	86.5	89.3	70.1	82.1

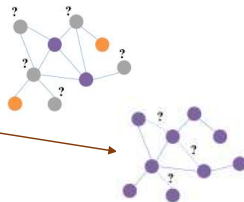
Post-BERT Pre-training Advancements

- RoBERTa: A Robustly Optimized BERT Pretraining Approach (University of Washington and Facebook, 2019)
- XLNet: Generalized Autoregressive Pretraining for Language Understanding (CMU and Google, 2019)
- ALBERT: A Lite BERT for Self-supervised Learning of Language Representations (Google and TTI Chicago, 2019)
- ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators (2020)

Graph Neural Network

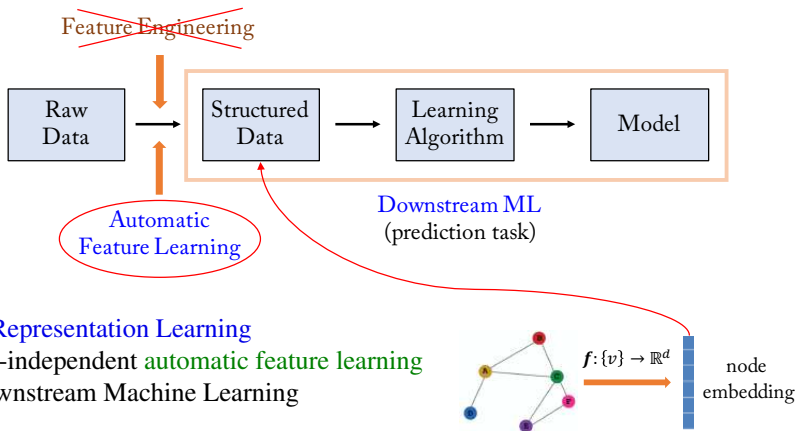
Graph Representation Learning

- **Graph-structured data** is ubiquitous throughout the natural and social sciences, from chemical synthesis and 3D-vision to recommender systems in social networks.
- **Graph Representation Learning** is a quickly growing subfield of Machine Learning that studies feature learning techniques for graph-structured data.
- Classical Machine Learning tasks on graphs
 - **Node classification**: predict a type of a given node
 - **Link prediction**: predict whether two nodes are linked
 - **Community detection**: identify densely linked clusters of nodes
 - **Network similarity**: how similar are two (sub)networks



Machine Learning Lifecycle

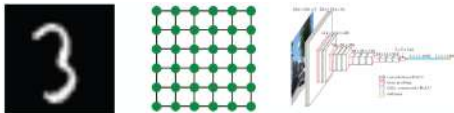
- Feature engineering in supervised Machine Learning



- Graph Representation Learning**
Do task-independent **automatic feature learning** for downstream Machine Learning

Difficulties of GRL

- Deep Learning toolbox is usually designed for simple grids or sequences.
 - CNNs for fixed-size images or grids...



- RNNs or word2vec for texts or sequences...



- But **graphs are more complex!**
 - complex topological structure (no spatial locality like grids)
 - no fixed node ordering or reference point (the isomorphism problem)
 - often dynamic and multimodal features (heterogeneous node and edge types)

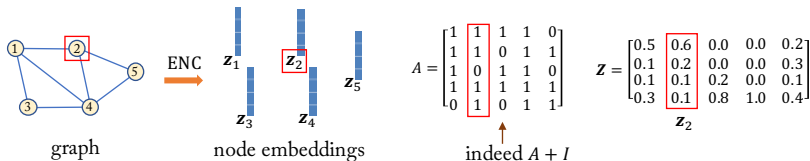
Algorithms

- **Node Embedding**: Machine Learning architecture (shallow network) encoding individual nodes to low-dimensional embeddings.
 - Matrix factorization-based methods:
Laplacian Eigenmaps, Graph Factorization, GraRep, HOPE
 - Random-walk based algorithms:
DeepWalk, Node2vec
- **Graph Neural Network**: Deep Learning architecture learning node features over graphs.
 - Graph Convolutional Network (GCN)
 - GraphSAGE
 - Gated Graph Neural Network
 - Graph Attention Network (GAT)

Node Embedding

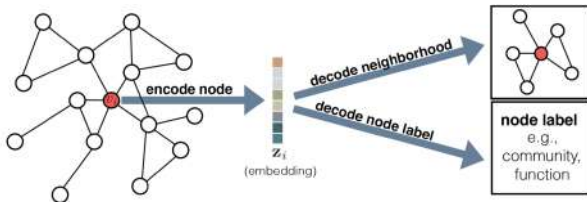
- Basic idea is **Dimensionality reduction** to encode individual nodes to low-dimensional vectors that summarize their graph position and the structure of their local neighborhood.
- Intuitively, find embedding of nodes so that **similarity in the embedding space** approximates **similarity in the original graph**.
- Node embeddings can then be **fed to downstream Machine Learning tasks** such as node classification, clustering and link prediction.
- Primary input from a graph $G = (V, E)$:
 - node feature matrix** $X \in \mathbb{R}^{F \times |V|}$ (F : input feature dimension per node)
 - adjacency matrix** $A \in \mathbb{R}^{|V| \times |V|}$ representing graph connection (binary).

Produce **node embeddings** $z_v \in \mathbb{R}^N$ (**embedding lookup** $Z \in \mathbb{R}^{N \times |V|}$) with $N \ll |V|$.



Encoder-Decoder approach

- **Encoder** maps each node v to a low-dimensional vector embedding z_v based on the node's position in the graph, its local neighborhood structure, and/or its feature.
- **Decoder** extracts user-defined information from z_v such as v 's local neighbor nodes (graph perspective) and classification label (node perspective).
- The encoder-decoder system learns to compress information about graph structure into the low-dimensional embedding space.
- If we learn to decode high-dim graph information regarding z_v 's into low-dim embeddings, then **node embeddings** contain all information necessary for downstream Machine Learning.




- **Encoder** $\text{ENC} : V \rightarrow \mathbb{R}^N$ generates node embeddings \mathbf{z}_v with **learnable parameters**.
- **Decoder** $\text{DEC} : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}^+$ reconstructs **user-defined similarity** values from \mathbf{z}_v 's **without learnable parameters**.
- **Similarity** $s_G : V \times V \rightarrow \mathbb{R}^+$ measures the **user-defined similarity** between nodes.
- **Loss function** $l : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ measures the **user-defined difference** between decoded similarity values $\text{DEC}(\mathbf{z}_i, \mathbf{z}_j)$ and true similarity values $s_G(v_i, v_j)$.

\Rightarrow **Objective** : $L = \sum_{(v_i, v_j) \in V \times V} l(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j), s_G(v_i, v_j))$

Type	Method	Decoder	Similarity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps	$\ \mathbf{z}_i - \mathbf{z}_j\ _2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_G(v_i, v_j)$
	Graph Factorization	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	GraRep	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, \dots, \mathbf{A}_{i,j}^k$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	HOPE	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
Random walk	DeepWalk	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$ (biased)	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

Type	Method	Decoder	Similarity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps	$\ \mathbf{z}_i - \mathbf{z}_j\ _2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_G(v_i, v_j)$
	Graph Factorization	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	GraRep	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, \dots, \mathbf{A}_{i,j}^k$	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$
	HOPE	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\ \text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_G(v_i, v_j)\ _2^2$

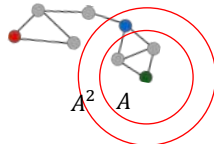


$$\mathbf{Z} = \begin{bmatrix} 0.5 & 0.6 & 0.0 \\ 0.1 & 0.2 & 0.0 \\ 0.1 & 0.1 & 0.2 \\ 0.3 & 0.1 & 0.8 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\mathbf{z}_1 \quad \mathbf{z}_2 \quad \mathbf{z}_3 \quad \mathbf{A}^2 = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$$

$D_{ij} = \|\mathbf{z}_i - \mathbf{z}_j\|_2^2$, then $D_{12} = 0.06, D_{23} = 0.50$
 $D_{ij} = \mathbf{z}_i^\top \mathbf{z}_j$, then $D_{12} = 0.36, D_{23} = 0.10$
 $S_{ij} = s_G(v_i, v_j) = A_{ij}$ (or A_{ij}^2), then $S_{12} = 1, S_{23} = 0$

- Matrix factorization** is named because, averaging over all nodes, they optimize $L = \|\mathbf{Z}^T \mathbf{Z} - \mathbf{S}\|_2^2$ with $\mathbf{Z} = [\mathbf{z}_i]$ and $\mathbf{S} = [s_G(v_i, v_j)]$.
- GraFac** considers only direct connections \mathbf{A} , but **GraRep** uses k -hop neighbors \mathbf{A}^k .
- HOPE** uses neighborhood overlap similarity measures.
(Jaccard overlap, Adamic-Adar score)
- Drawbacks: $O(|V|^2)$ running time (all node pairs are considered)
 $O(|V|)$ parameters (each node has a learned vector)



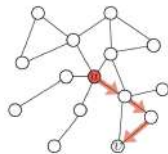
Type	Method	Decoder	Similarity measure	Loss function (ℓ)
Random walk	DeepWalk	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_G(v_j v_i)$ (biased)	$-s_G(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

- Node embeddings based on **Random walk** statistics are learned so that nodes have similar embeddings if they tend to co-occur on short random walks over the graph.
- Instead of a deterministic similarity measure in Matrix factorization, Random walk method uses a **stochastic similarity measure** $p_G(v_j | v_i)$, probability of visiting v_j on a random walk starting at v_i using some random walk strategy.
- Formally, they minimize the cross-entropy loss

$$L = \sum_{(v_i, v_j) \in D} -\log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$$

where the training set D is generated by sampling random walks starting at each node (N pairs for each v_i are sampled from the distribution $(v_i, v_j) \sim p_G(v_j | v_i)$).

It takes $O(|D||V|)$ running time since the decoder's denominator takes $O(|V|)$.



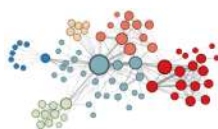
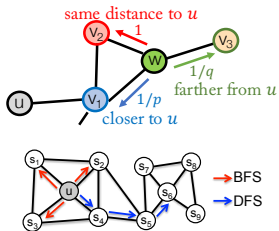
- **DeepWalk** uses **hierarchical softmax** to compute the normalizing factor $\sum_{v_k \in V} e^{z_i^T z_k}$ of the loss L , using a binary-tree structure to accelerate the computation.

It uses simple unbiased random walks. ($O(|D|\log|V|)$ running time)

- **Node2vec** approximates L using **negative sampling** on the normalizing factor.

In particular, it biases the random walk using the hyperparameter p and q , which control the likelihood of the walk moving ‘closer to’ and ‘farther from’ u , respectively.

By trading off p and q , node2vec smoothly interpolate between walks that are more akin to breadth-first search (BFS, low value of p) to emphasize community structures or depth-first search (DFS, low value of q) to emphasize local structural roles.



microscopic view
($p = 1, q = 2$)

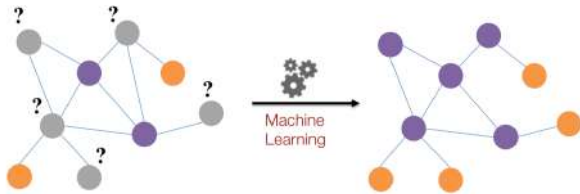


macroscopic view
($p = 1, q = 0.5$)

Graph Neural Network

Predicting Node Properties

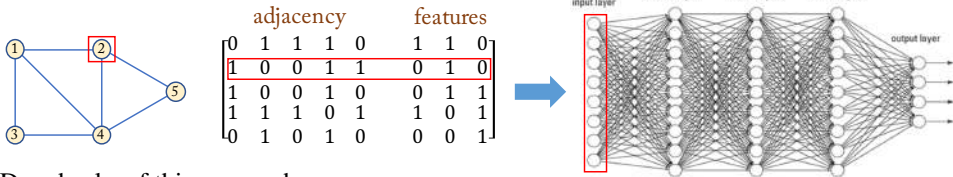
- Many important real-world datasets are provided in the form of graphs or networks such as social networks, knowledge graphs, protein interaction networks, etc.
- Yet, little attention has been devoted to the generalization of neural network models on graph-structured data.



Graph-based semi-supervised learning for node classification

Naïve Approach

Combine the **node features** of a graph and its **adjacency matrix**, and feed them into DNN.

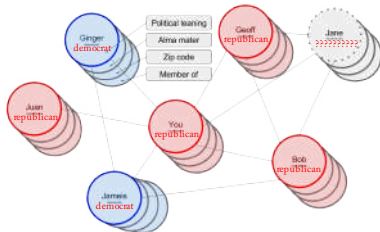


- Drawbacks of this approach
 - $O(|V|)$ parameters
 - Not invariant to node orderings
 - Not applicable to graphs of different sizes
- Ideal **Graph Neural Network** algorithm
 - Number of model parameters should be independent of graph size.
 - Invariant to node ordering (the isomorphism problem)
 - Locality (operations much depend on the neighbors of a given node)
 - Model should be independent of graph structure to transfer the model across graphs.

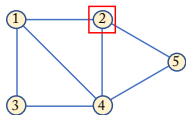
Graph Convolutional Network (GCN)

GCN is a **convolutional NN** operating on graph-structured data which takes two inputs (**node features** of a graph and its **adjacency matrix**) and produces **node embeddings**.

- Traditional DL systems do not use of the relationships between entities to make predictions, they only use features of individuals.
- GCN learns hidden layer vectors that encode both node features and local graph structures on a **fixed graph**.



- Graph-based NN model $\mathbf{Z} = f(\mathbf{X}, \mathbf{A})$ is a function on a fixed graph $G = (V, E)$ taking
 - **node feature matrix** $\mathbf{X} \in \mathbb{R}^{|V| \times F}$ (F : input feature dimension per node)
 - **adjacency matrix** $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$ representing graph connection (binary or weighted)
 and produces **node-level output** $\mathbf{Z} \in \mathbb{R}^{|V| \times F'}$ (F' : output feature dimension per node) through network **layer feature vectors** $\mathbf{H}^{(l+1)} = f(\mathbf{H}^{(l)}, \mathbf{A})$ from $\mathbf{H}^{(0)} = \mathbf{X}$ to $\mathbf{H}^{(L)} = \mathbf{Z}$.
- Consider a basic layer-wise propagation rule
 $f(\mathbf{H}^{(l)}, \mathbf{A}) = \sigma(\mathbf{A}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$ with the weight matrix $\mathbf{W}^{(l)}$ of dimension $F^{(l)} \times F^{(l+1)}$ where $F^{(l)}$ is l -th layer feature dimension and σ is a non-linearity like ReLU.

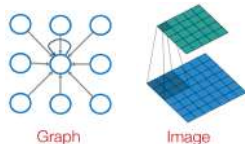


$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$\mathbf{H}^{(l)} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}\mathbf{H}^{(l)}\mathbf{W}^{(l)} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 1 & 1 \end{bmatrix} \mathbf{W}^{(l)}$$

- $\mathbf{A}\mathbf{H}^{(l)}$ aggregates local neighbors' information (node features).
 Weights $\mathbf{W}^{(l)}$ is similar to a filter (kernel) in CNN since it is shared across all nodes and independent on the graph size.

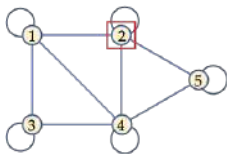


Two limitations of the basic model

- Multiplication with A means that, for every node, we sum up all feature vectors of all neighboring nodes but not itself. (\Rightarrow Enforce self-loops $\tilde{A} = A + I$)
- Multiplication with \tilde{A} changes the scale of feature vectors.
(\Rightarrow Normalize \tilde{A} to $D^{-1}\tilde{A}$ with the degree matrix D (regarding \tilde{A}) so that all rows sum to one.
In practice, use a symmetric normalization $D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}$.)

Final layer-wise propagation rule

$$H^{(l+1)} = f(H^{(l)}, A) = \sigma(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}H^{(l)}W^{(l)})$$



$$\tilde{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

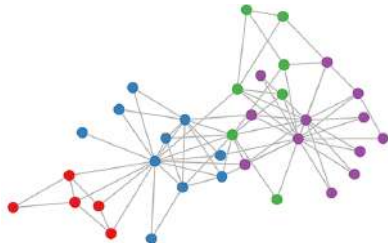
$$D = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix}$$

$$D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}} = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{2\sqrt{3}} & \frac{1}{2\sqrt{5}} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{2\sqrt{5}} & \frac{1}{2\sqrt{3}} \\ \frac{1}{2\sqrt{3}} & 0 & \frac{1}{3} & \frac{1}{\sqrt{15}} & 0 \\ \frac{1}{2\sqrt{5}} & \frac{1}{2\sqrt{5}} & \frac{1}{\sqrt{15}} & \frac{1}{5} & \frac{1}{\sqrt{15}} \\ 0 & \frac{1}{2\sqrt{3}} & 0 & \frac{1}{\sqrt{15}} & \frac{1}{3} \end{bmatrix}$$

- Residual GCN uses $H^{(l+1)} = \sigma(D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}H^{(l)}W^{(l)} + H^{(l)})$

Zachary's Karate club graph: 34 members with colors indicating different communities

- We take a **simple 3-layer GCN** with randomly initialized weights $W^{(l)}$.
- Even before training the weights, we simply insert the adjacency matrix and input $X = I$ (no node feature information) into the model.
- This GCN now performs three propagation steps during the forward pass and effectively convolves the 3rd-order neighborhoods of every node.
- Remarkably, after training for 300 iterations, the model produces a 2D embedding of these nodes that closely resembles the community-structure of the graph.

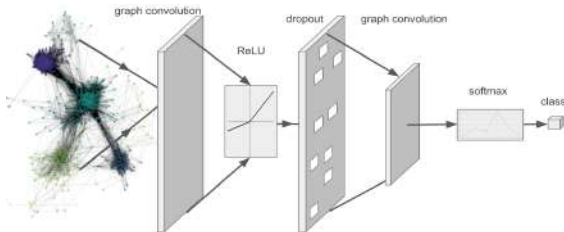


Semi-supervised node classification: 2-layer GCN for semi-supervised learning of the model $f(X, A)$ both on the data X and on the adjacency matrix A of a fixed graph.

- Calculate $\hat{A} = D^{-\frac{1}{2}} \tilde{A} D^{-\frac{1}{2}}$ in a pre-processing step.
- Take $Z = f(X, A) = \text{softmax}(\hat{A} \text{ReLU}(\hat{A} X W^{(0)}) W^{(1)})$.
- Evaluate the cross-entropy error over all labeled examples denoted by Y_l

$$L = - \sum_{l \in I_L} \sum_{f=1}^F Y_{lf} \ln Z_{lf}$$

where I_L is the labeled node index set and F is the output dimension.



Facebook friendship network clustered by colored political stances

Relation to Weisfeiler-Lehman algorithm

- **Weisfeiler-Lehman algorithm** gives the unique assignment of node labels for a graph.
 - Every node is assigned a feature that uniquely describes its role in the graph.
 - This feature assignment can be used as a check for graph isomorphism.

Algorithm 1: WL-1 algorithm

Input: Initial node coloring $(h_1^{(0)}, h_2^{(0)}, \dots, h_N^{(0)})$

Output: Final node coloring $(h_1^{(T)}, h_2^{(T)}, \dots, h_N^{(T)})$

$t \leftarrow 0$;

repeat

for $v_i \in \mathcal{V}$ **do**

$h_i^{(t+1)} \leftarrow \text{hash} \left(\sum_{j \in \mathcal{N}_i} h_j^{(t)} \right)$;

$t \leftarrow t + 1$;

until *stable node coloring is reached*;

$h_i^{(t)}$ label assignment (coloring)
of node v_i (at iteration t)

\mathcal{N}_i set of its neighboring node indices

$\text{hash}(\cdot)$ is a hash function.

- Our **GCN layer-wise propagation rule** (now in vector form)

$$\mathbf{h}^{(l+1)} = \sigma \left(\sum_{j \in N_i} \frac{1}{\sqrt{d_i d_j}} \mathbf{h}_j^{(l)} W^{(l)} \right)$$

where $\mathbf{h}^{(l)}$ is a feature vector of node v_i in the l -th layer, and $d_i = |N_i|$.

This propagation rule can be interpreted as a **differentiable** and **parameterized** variant of the hash function used in Weisfeiler-Lehman algorithm.

GraphSAGE

- Most existing node embeddings like GCN work on a single fixed graph whose all nodes are present during training (transductive), and so do not generalize to unseen nodes.
- **GraphSAGE** is an **inductive** framework to generate node embeddings for unseen new data (Sample+aggreGatE), operating by **sampling** a fixed-size neighbors of each node and then performing a specific **aggregator** (mean, LSTM or Max-pooling) over it.
- Model size (parameter dimension) is **independent of the graph size** by parameter sharing, and so we can scale to billions of graph nodes.
- It uses a graph-based loss function to the output node embeddings in an **unsupervised** setting or a task-specific loss function in a supervised setting.

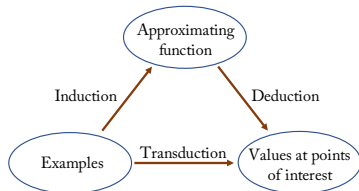


Transductive vs Inductive

- Induction**: deriving the function from the given data.

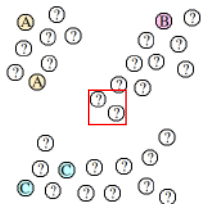
Deduction: deriving the values of the given function at points of interest.

Transduction: deriving the values of the unknown function at points of interest from the given data.



- In Machine Learning, **transductive inference** is reasoning from observed specific (training) cases to specific (test) cases. It predicts on a known (test) set of unlabeled examples.

In contrast, **inductive inference** is reasoning from observed training cases to general rules, which are then applied to the (even unseen) test cases.



- Inductive approach uses the only five labeled points to train a supervised learning algorithm (approximating function), and to predict labels of all unlabeled points. It certainly struggles to build a model that captures the structure of this data. If a nearest-neighbor algorithm is used, then middle points are labeled A or C.
- Transductive approach has the advantage of being able to consider all points, not just the labeled points. It labels the unlabeled points according to clusters to which they naturally belong, and so middle points would be labeled B.

- **Advantage of transduction**: it may be able to make better predictions with fewer labeled points, because it uses the natural breaks found in the unlabeled points.
- **Disadvantage of transduction**: it builds no predictive model.

If a previously unknown point is added to the set, then the entire transductive algorithm would need to be repeated with all points in order to predict a label of the new point.

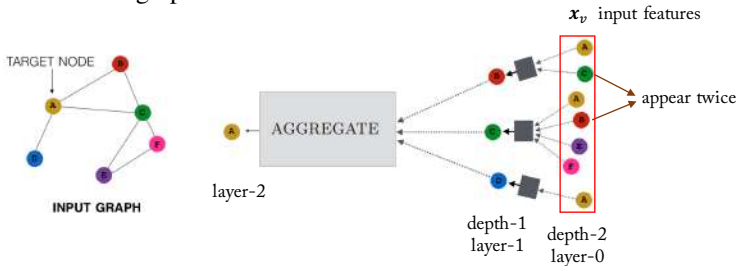
- **Difficulty of inductive node embeddings** compared to the transductive setting: generalizing to unseen nodes requires ‘aligning’ newly observed subgraphs to the node embeddings that the algorithm has already optimized on.

Inductive framework must learn to recognize structural properties of a node’s neighbors that reveal both the node’s local role in the graph and its global position.

- **GCN** is designed for **semi-supervised learning** in the **transductive** setting with fixed graphs, and requires that the full graph Laplacian is known during training.
- **GraphSAGE** extends GCN to the task of **inductive unsupervised learning** and generalizes GCN approach to use trainable aggregation functions beyond simple convolutions.

Neighborhood Aggregation

- Main scheme is to generate node embeddings by **aggregating information (node features) from its local neighbors** to overcome limitations of shallow embedding methods, which rely on the entire graph and do not incorporate node features.
- For every node A , the model aggregates information from A 's neighbors B, C, D (depth-1), and in turn, the neighbors' information are based on information aggregated from their respective neighbors (depth-2), and so on.
- As the process iterates, nodes incrementally gain more and more information from further reaches of the graph.



Algorithm 1: GraphSAGE embedding generation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K
 $\text{AGGREGATE}_k, \mathbf{W}^k, \forall k \in \{1, \dots, K\}; \mathcal{N}: v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations \mathbf{z}_v for all $v \in \mathcal{V}$

```

1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V};$ 
2 for  $k = 1 \dots K$  do
3   for  $v \in \mathcal{V}$  do
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\});$ 
5      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
6   end
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
8 end
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 

```

K : search depth

AGGR_k : **learnable** differ. aggregator

\mathbf{W}^k : **learnable** weight function

Initialize \mathbf{h}_v^0 as input feature \mathbf{x}_v .

Aggregating neighbors' vectors.
Assigning new node vectors.

\mathbf{z}_v : the output node embedding

- At each iteration k , for every node v , AGGR_k aggregates its neighbors' previous vectors, and the model concatenates its previous vector \mathbf{h}_v^{k-1} and the aggregated vector $\mathbf{h}_{\mathcal{N}(v)}^k$ and feeds it through a FC layer \mathbf{W}^k to assign a new vector \mathbf{h}_v^k to v .
- After K iterations, the final vectors are the output node embeddings \mathbf{z}_v .
- We can feed these node embeddings \mathbf{z}_v into a task-specific loss function $L(\mathbf{z}_v)$ and run SGD to train the **learnable parameters** (aggregators AGGR_k and weights \mathbf{W}^k).

Aggregators operate over an **unordered vector set** since nodes' neighbors have no ordering.

- * Ideally, they would be **symmetric (permutation invariant)** while still being trainable and maintaining high representational capacity.

- **Mean aggregator** simply takes the element-wise mean of the vectors.

$$\mathbf{h}_v^k \leftarrow \sigma \left(\mathbf{W}^k \cdot \text{mean} \{ \mathbf{h}_u^{k-1}; u \in N(v) \cup \{v\} \} \right)$$

- It is nearly equivalent to the convolutional propagation rule in transductive GCN.

- **LSTM aggregator** applies LSTM to random ordering of the node's neighbors.

$$\text{AGGR}_k^{\text{LSTM}} = \text{LSTM} \{ \mathbf{h}_{u_i}^{k-1}; u_i \in N(v) \}$$

- It has the advantage of larger expressive capability, but is not symmetric.

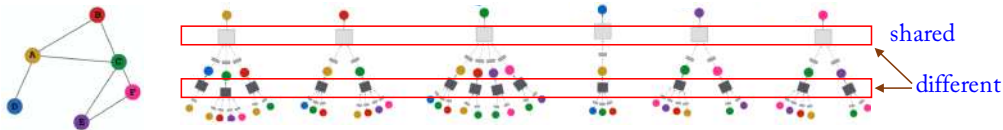
- **Pooling aggregator** feeds independently each neighbor's vector through a MLP and an element-wise max-pooling operation aggregates information across the neighbors.

$$\text{AGGR}_k^{\text{pool}} = \max \{ \sigma (\mathbf{W}^k \mathbf{h}_u^{k-1} + \mathbf{b}); u \in N(v) \}$$

- MLP computes features for each neighboring node representation.
- It is both symmetric and trainable.

Parameter sharing

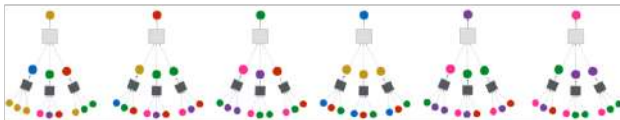
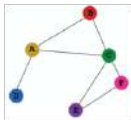
- Learnable parameters ($AGGR_k$ and W^k) are shared across all nodes at each iteration k (to allow the inductive capability), but not shared at different iterations.



- Parameter sharing increases efficiency (parameter dimension is independent of the graph size) and allows to generate embeddings for unseen nodes that were not observed during training.
- Every node has unique compute graph, so we cannot batch on GPU.

Sample neighborhood

- By sampling a fixed-size neighbors of each node, compute graphs have same structure.



Batch of nodes
for GPU batching

Algorithm 2: GraphSAGE minibatch forward propagation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$; depth K
 $\text{AGGREGATE}_k, \mathbf{W}^k, \mathcal{N}_k: v \rightarrow 2^{\mathcal{V}}, \forall k \in \{1, \dots, K\}$

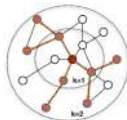
Output : Vector representations \mathbf{z}_v for all $v \in \mathcal{B}$

```

1  $\mathcal{B}^K \leftarrow \mathcal{B}$ ;
2 for  $k = K \dots 1$  do
3    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$ ;
4   for  $u \in \mathcal{B}^k$  do
5      $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$ ;
6   end
7 end
8  $\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$ ;
9 for  $k = 1 \dots K$  do
10  for  $u \in \mathcal{B}^k$  do
11     $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}(u)\})$ ;
12     $\mathbf{h}_u^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k))$ ;
13     $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$ ;
14  end
15 end
16  $\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$ 

```

Sample



aggreGateE



$N_k(u)$: deterministic function
specifying a random sample
of fixed-sized neighbors of u
(pre-computed randomness).

Random samples are independent
across iterations over k .

First, we sample all the nodes needed
for the computation.

Samplings are reversed over k from
the minibatch B till B^0 , required
neighbors up to depth K .

Next, we do ‘Aggregation stage’.

Training

- **GraphSAGE** uses a graph-based loss function to the output node embeddings \mathbf{z}_v in order to learn useful predictive representations in a fully **unsupervised** setting.

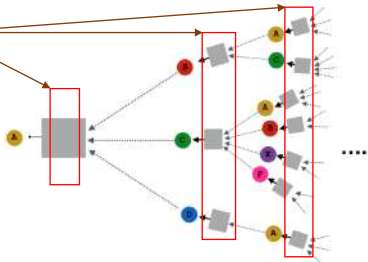
$$L(\mathbf{z}_v) = -\log(\sigma(\mathbf{z}_v^T \mathbf{z}_u)) - N \mathbb{E}_{u_n \sim P_n(u)} \log(\sigma(-\mathbf{z}_v^T \mathbf{z}_{u_n}))$$

- u is a node that co-occurs near v on fixed-length random walk.
 P_n is a negative sampling distribution and N is the number of negative samples.
- This loss function trains the learnable parameters (AGGR_k and \mathbf{W}^k) via SGD to encourage nearby nodes to have similar representations (so high $\mathbf{z}_v^T \mathbf{z}_u$), while enforcing disparate nodes to have highly distinct representations.
- Importantly, the node embeddings \mathbf{z}_v that we feed into the loss function $L(\mathbf{z}_v)$ are generated from the features contained within a node's local neighbors.
- In this unsupervised setting, \mathbf{z}_v can be provided to downstream ML applications.
In cases where \mathbf{z}_v is used for a specific downstream task, this unsupervised loss function can be replaced by a task-specific objective, for example a cross-entropy loss.

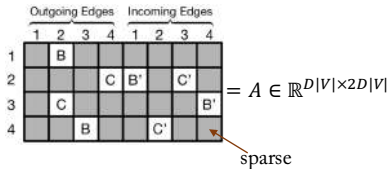
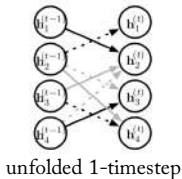
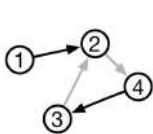
Gated Graph Neural Network

Gated graph neural network extends GraphSAGE to sequential outputs by using GRU instead of propagation model.

- GCNs and GraphSAGE generally only 2 or 3 layers deep because many layers cause
 - overfitting from too many parameters (each layer has different parameters),
 - vanishing/exploding gradients during backpropagation.
- GGNN shares parameters across layers by using recurrent state update in GRU with truncated BPTT computing gradients.
- It handles models with 20 or more layers, allowing for complex information of global graph structure to be propagated to all nodes.



Basic recurrence of the propagation model



$h_1^v = \begin{bmatrix} \mathbf{x}_v \\ 0 \end{bmatrix}$ hidden state (dim= D) at v by 0-padding to node feature \mathbf{x}_v

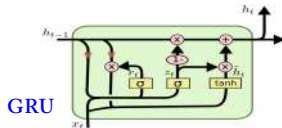
$$a_t^v = A_v^T \begin{bmatrix} h_{t-1}^1 \\ \vdots \\ h_{t-1}^{|V|} \end{bmatrix} + \mathbf{b} \quad (\text{dim}=2D) \text{ where } A_2 = \begin{bmatrix} 2 & 2 \\ B & \\ & \\ C & \\ & \\ & C' \end{bmatrix}$$

$$z_t^v = \sigma(W_z a_t^v + U_z h_{t-1}^v) \quad (\text{dim}=D)$$

$$r_t^v = \sigma(W_r a_t^v + U_r h_{t-1}^v)$$

$$\tilde{h}_t^v = \tanh(W_h a_t^v + U_h (r_t^v \odot h_{t-1}^v))$$

$$h_t^v = (1 - z_t^v) \odot h_{t-1}^v + z_t^v \odot \tilde{h}_t^v$$



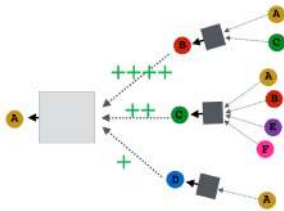
$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) && \text{update gate} \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) && \text{reset gate} \\ \tilde{h}_t &= \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t && \text{hidden state} \end{aligned}$$

\odot element-wise product

Graph Attention Network (GAT)

Graph attention network extends GraphSAGE by using **masked self-attention** layers.

- In this self-attention layer setting, every nodes attends over its neighbor features for assigning different importances (attention weights) to different neighbor nodes while dealing with different sized neighborhoods and without knowing the entire graph structure upfront.
- Computationally efficient because it does not require costly matrix operations as in GCN, and is parallelizable across all nodes.
- Advantage of attention mechanism is to perform a thorough analysis on **model interpretability**.
- Applicable directly to inductive problems.



Graph attention layer

- $\{h_1, h_2, \dots, h_{|V|}\}, h_v \in \mathbb{R}^F$: input node features (F input feature dimension)
 $\{h'_1, h'_2, \dots, h'_{|V|}\}, h'_v \in \mathbb{R}^{F'}$: produced node features as outputs (F' new feature dimension)
 $\mathbf{W} \in \mathbb{R}^{F' \times F}$: weight matrix applied to every node, shared at the present layer

- Attention value $\alpha_{vu} = \text{softmax}_u(e_{vu}) = \frac{\exp(e_{vu})}{\sum_{k \in N(v) \cup \{v\}} \exp(e_{vk})}$

where $e_{vu} = a(\mathbf{W}h_v, \mathbf{W}h_u)$ indicates the importance of key node u 's feature to query node v ,
and $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \rightarrow \mathbb{R}$ is a **shared self-attention** performing in **masked** manner
(computing e_{vu} only for neighboring nodes u , but not every other node).

- In the paper, the attention a is a single FC layer parametrized by a weight vector $\mathbf{a} \in \mathbb{R}^{2F'}$
followed by LeakyReLU with a concatenation \parallel ,

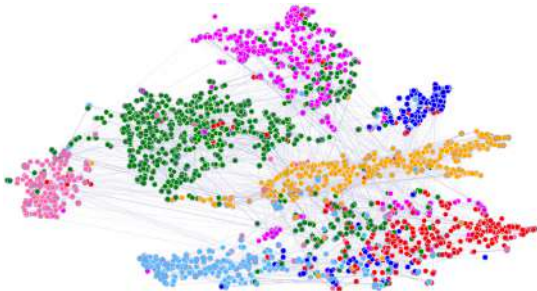
$$h'_v = \sigma \left(\sum_{u \in N(v) \cup \{v\}} \alpha_{vu} \mathbf{W}h_u \right) \text{ with } \alpha_{vu} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}h_v \parallel \mathbf{W}h_u]))}{\sum_{k \in N(v) \cup \{v\}} \exp(\text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}h_v \parallel \mathbf{W}h_k]))}$$

- **Multi-head attention** is used to stabilize the learning process of self-attention.
Specifically, M independent attentions a^m are executed, and then their features are concatenated:

$$h'_v = \parallel_{m=1}^M \sigma \left(\sum_{u \in N(v) \cup \{v\}} \alpha_{vu}^m \mathbf{W}^m h_u \right) \text{ which has } MF' \text{ feature dimension.}$$

Performance example

- **t-SNE** plot of the computed feature vectors of a pre-trained GAT model's first hidden layer on Cora dataset which is a global oceanographic temperature and salinity dataset.
- Node colors denote classes.
- Edge thickness indicates aggregated normalized attention values between nodes, across all 8-head attentions.



Performance Measure

Performance Measures

- **Confusion matrix** is a specific table layout that allows visualization of the performance (**correctness** and **accuracy**) of the model.
- Each row of the matrix represents the instances in a predicted class A while each column represents the instances in an actual class B .

Each entry counts the number of times that instances of class B are classified as class A .

```
array([[5725, 3, 24, 9, 10, 49, 50, 10, 39, 4],
       [ 2, 6493, 43, 25, 7, 40, 5, 10, 109, 8],
       [ 51, 41, 5321, 104, 89, 26, 87, 60, 166, 13],
       [ 47, 46, 141, 5342, 1, 231, 40, 50, 141, 92],
       [ 19, 29, 41, 10, 5366, 9, 56, 37, 86, 189],
       [ 73, 45, 36, 193, 64, 4582, 111, 30, 193, 94],
       [ 29, 34, 44, 2, 42, 85, 5627, 10, 45, 0],
       [ 25, 24, 74, 32, 54, 12, 6, 5787, 15, 236],
       [ 52, 161, 73, 156, 10, 163, 61, 25, 5027, 123],
       [ 43, 35, 26, 92, 178, 28, 2, 223, 82, 5240]])
```

		Actual class		
		Cat	Dog	Rabbit
Predicted class	Cat	5	2	0
	Dog	3	3	2
	Rabbit	0	1	11

		Actual class	
		Cat	Non-cat
Predicted class	Cat	5 True Positives	2 False Positives
	Non-cat	3 False Negatives	17 True Negatives

		True condition			
Total population		Condition positive	Condition negative	Prevalence = $\frac{\Sigma \text{Condition positive}}{\Sigma \text{Total population}}$	Accuracy (ACC) = $\frac{\Sigma \text{True positive} + \Sigma \text{True negative}}{\Sigma \text{Total population}}$
Predicted condition	Predicted condition positive	True positive	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\Sigma \text{True positive}}{\Sigma \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\Sigma \text{False positive}}{\Sigma \text{Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\Sigma \text{False negative}}{\Sigma \text{Predicted condition negative}}$	Negative predictive value (NPV) = $\frac{\Sigma \text{True negative}}{\Sigma \text{Predicted condition negative}}$
		True positive rate (TPR), Recall, Sensitivity, probability of detection, Power = $\frac{\Sigma \text{True positive}}{\Sigma \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\Sigma \text{False positive}}{\Sigma \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR+}}{\text{LR-}}$ F ₁ score = $2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
		False negative rate (FNR), Miss rate = $\frac{\Sigma \text{False negative}}{\Sigma \text{Condition positive}}$	Specificity (SPC), Selectivity, True negative rate (TNR) = $\frac{\Sigma \text{True negative}}{\Sigma \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$	

Source by Wikipedia

Accuracy of tests in developing NN

Accuracy of the performance may be sufficient if you're only worried about the overall correctness of the algorithm.

But, accuracy measure doesn't reveal a breakdown of correct and incorrect results per each label.

- **Accuracy**: probability that a person receives a correct test result
- **Sensitivity (Recall)**: probability that a person who truly has the disease correctly receives a positive test result
- **Specificity**: probability that a person who is truly healthy correctly receives a negative test result

$$\text{Accuracy (ACC)} = \frac{\sum \text{True positive} + \sum \text{True negative}}{\sum \text{Total population}}$$

$$\begin{aligned} &\text{True positive rate (TPR), Recall,} \\ &\quad \text{Sensitivity,} \\ &\text{probability of detection} \\ &= \frac{\sum \text{True positive}}{\sum \text{Condition positive}} \end{aligned}$$

$$\begin{aligned} &\text{Specificity (SPC), Selectivity,} \\ &\quad \text{True negative rate (TNR)} \\ &= \frac{\sum \text{True negative}}{\sum \text{Condition negative}} \end{aligned}$$

Comparison with Statistical Hypothesis Test

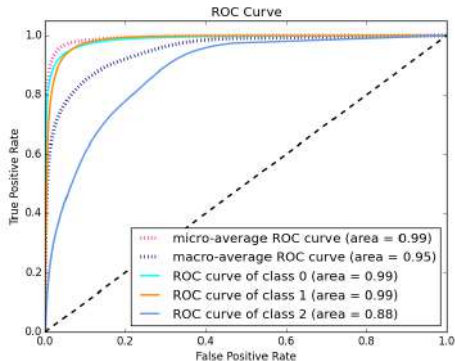
- Hypothesis: H_0 = disease and H_1 = healthy
- Type I error** = False positive rate
 $= 1 - \text{Specificity}$
 $= P(\text{accept } H_0 \mid H_1)$
- Type II error** = False negative rate
 $= 1 - \text{Sensitivity}$
 $= P(\text{reject } H_0 \mid H_0)$

Type I error
False positive rate (FPR), Fall-out, probability of false alarm $= \frac{\sum \text{False positive}}{\sum \text{Condition negative}}$

Type II error
False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$

ROC curve

- **ROC** (Receiver Operating Characteristics) curve: plotting the true positive rate (TPR, Sensitivity) against the false positive rate (FPR, $1 - \text{Specificity}$) at various threshold T settings.
- **AUC** (Area Under Curve)



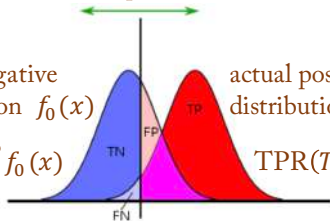
threshold parameter T

actual negative
distribution $f_0(x)$

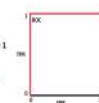
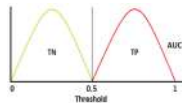
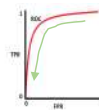
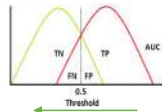
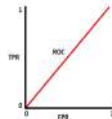
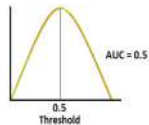
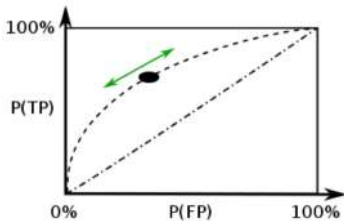
actual positive
distribution $f_1(x)$

$$\text{FPR}(T) = \int_T^\infty f_0(x)$$

$$\text{TPR}(T) = \int_T^\infty f_1(x)$$



	TP	FP
	FN	TN



Accuracy of tests in real usage

- **Positive predictive value (precision)**: probability that a person who has a positive test result really has the disease.
- **Negative predictive value**: probability that a person who has a negative test result really is healthy.

$$\text{Positive predictive value (PPV), Precision} = \frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$$

$$\text{Negative predictive value (NPV)} = \frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$$

Caution about predictive values: reading positive and negative predictive values directly from table is accurate only if the proportion of diseased people in the sample is representative of the proportion of diseased people in the population. (Random sample)

True Status	Test Result		Total
	Diseased	Healthy	
Diseased	392	8	400
Healthy	24	576	600
Total	416	584	1000

$$\text{Sens} = 392/400 = 0.98$$

$$\text{Spec} = 576/600 = 0.96$$

$$\text{PPV} = 392/416 = 0.94$$

$$\text{NPV} = 576/584 = 0.99$$

(prevalence of disease 40%)

True Status	Test Result		Total
	Diseased	Healthy	
Diseased	49	1	50
Healthy	38	912	950
Total	87	913	1000

$$\text{Sens} = 49/50 = 0.98$$

$$\text{Spec} = 912/950 = 0.96$$

$$\text{PPV} = 49/87 = 0.56$$

$$\text{NPV} = 912/913 = 0.999$$

(prevalence of disease 5%)

F_1 score

- It is convenient to combine precision and recall into a single metric: F_1 score, in particular it favors classifiers that have similar precision and recall.
- F_1 score is the harmonic mean of precision and recall.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

Precision-Recall tradeoff

- Contrast to F_1 score, in some contexts you mostly care about precision, and in other contexts you really care about recall.
- Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. (Precision-Recall tradeoff)
- For example, if you train a classifier to detect videos that are safe for kids, you probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision).

On the other hand, if you train a classifier to detect shoplifters on surveillance images: it is probably fine if the classifier has only 30% precision as long as 99% recall (so the security guards get a few false alerts, but almost all shoplifters will get caught).

