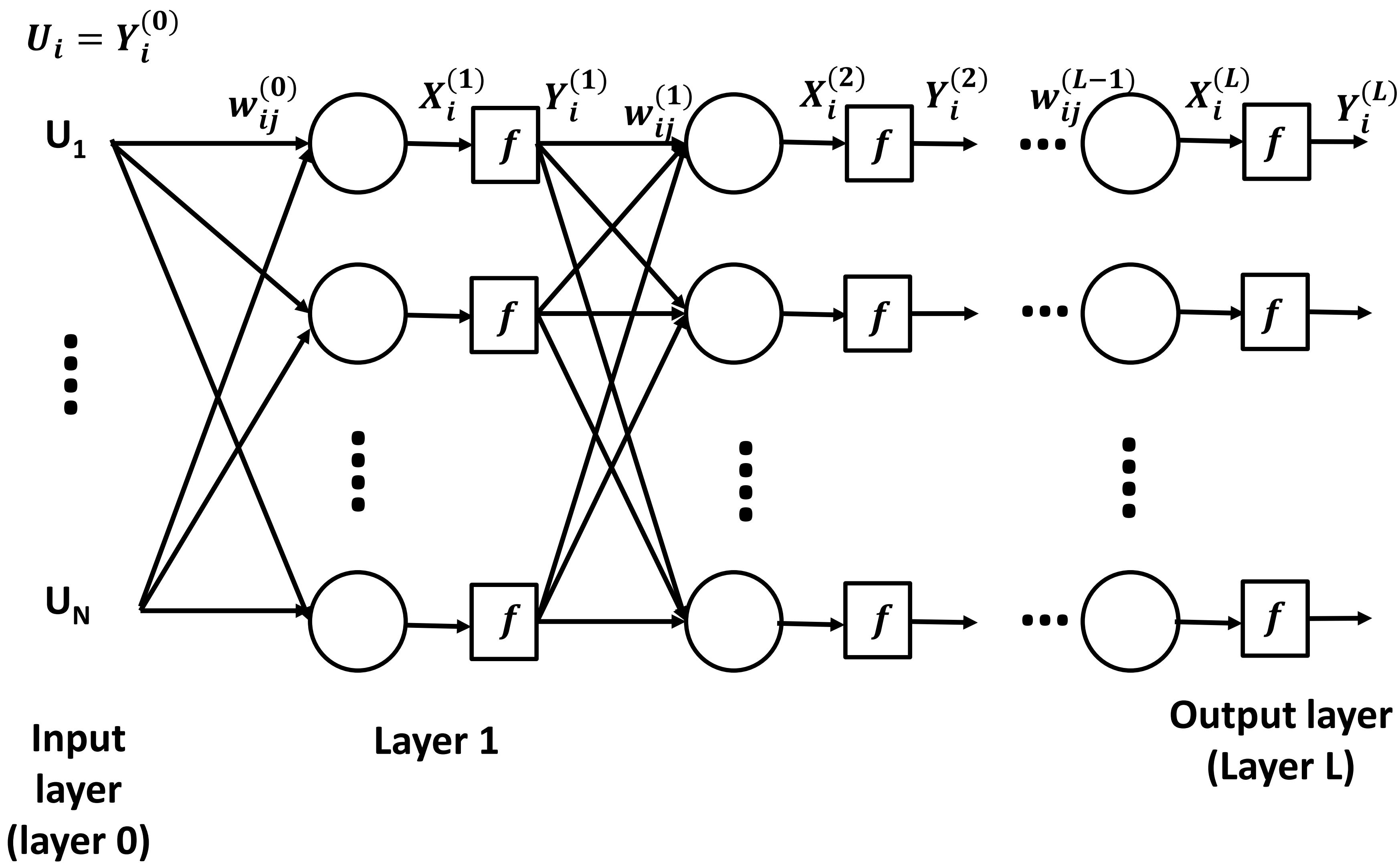
Pattern Classification

11. Backpropagation & Time-Series Forecasting

AbdElMoniem Bayoumi, PhD

Spring 2022

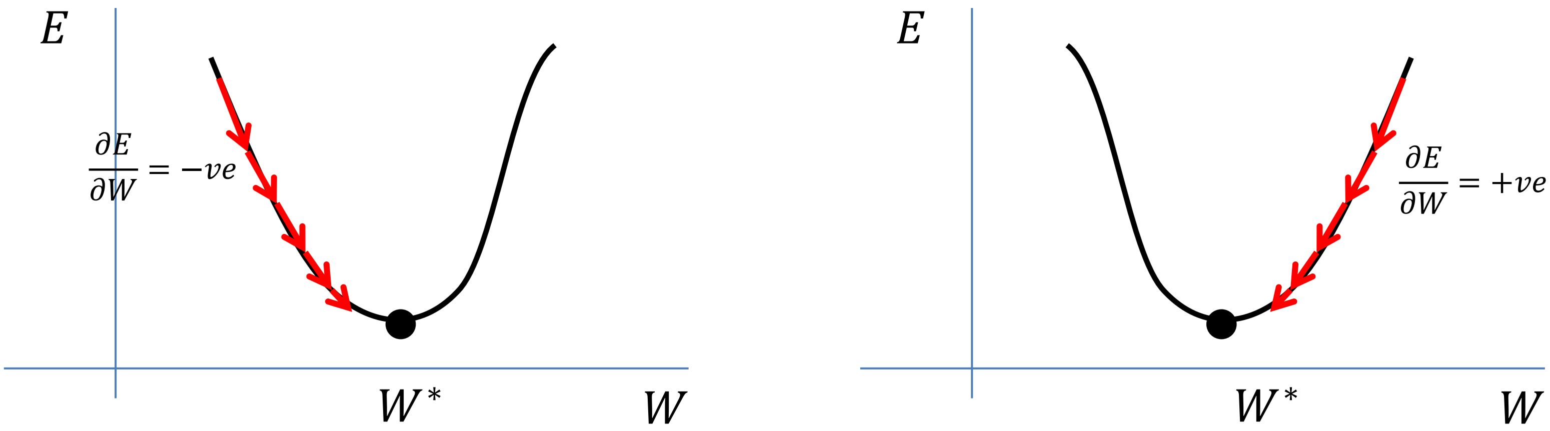
Recap: Multi-Layer Networks



N(l) is the number of nodes is layer l

Recap: Gradient Descent

- It can be shown that the negative direction of the gradient gives the steepest descent



- When we approach the min, the steps become very small because close to the min we find $\frac{\partial E}{\partial \underline{W}} \approx 0$

Back propagation Algorithm

- It is an algorithm based on the steepest descent concept

- Used to train a general multi-layer network

Back propagation Algorithm

- $X_i^{(l)} = \sum_{j=1}^{N(l-1)} w_{ij}^{(l-1)} Y_j^{(l-1)}$ **Output of hidden node before applying the activation function**

- $Y_i^{(l)} = f\left(X_i^{(l)}\right)$ **Output of layer**

Back propagation Algorithm

Error, i.e., cost fn.

- $E = \frac{1}{M} \sum_{m=1}^{M} E_m$
- $E_m = \sum_{i=1}^{N(L)} \left[Y_i^{(L)}(m) - d_i(m) \right]^2$ **Loss (i.e., for regression)**

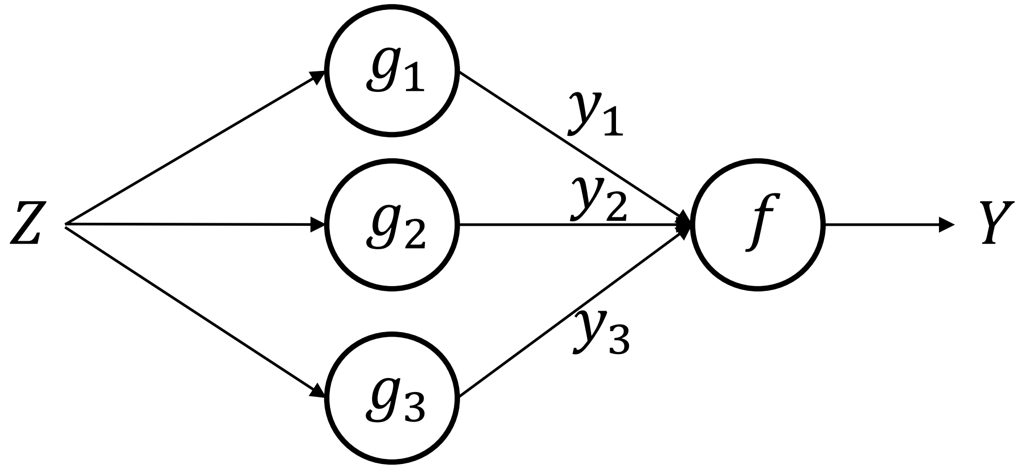
- $Y_i^{(L)}(m) \equiv i^{th}$ output of the NN for the training pattern m
- $d_i(m) \equiv$ target o/p
- $N(L) \equiv$ no. of outputs (i.e., nodes of the output layer)

- We need to compute the gradient, i.e., $\frac{\partial E}{\partial w_{ij}^{(l)}}$

Chain Rule

- $Y = f(y_1, y_2, y_3)$

- $y_1 = g_1(Z)\,, y_2 = g_2(Z), y_3 = g_3(Z)$



- $\frac{\partial Y}{\partial Z} = \frac{\partial Y}{\partial y_1} * \frac{\partial y_1}{\partial Z} + \frac{\partial Y}{\partial y_2} * \frac{\partial y_2}{\partial Z} + \frac{\partial Y}{\partial y_3} * \frac{\partial y_3}{\partial Z}$

Back propagation Algorithm

- $E_m = \sum_{i=1}^{N(L)} \left[Y_i^{(L)}(m) - d_i(m)\right]^2$
- $Y_i^{(L)} = f\left(X_i^{(L)}\right)$
- $X_i^{(L)} = \sum_{j=1}^{N(L-1)} w_{ij}^{(L-1)} Y_j^{(L-1)}$

$$\frac{\partial E_m}{\partial w_{IJ}^{(L-1)}} = \frac{\partial E_m}{\partial Y_I^{(L)}} * \frac{\partial Y_I^{(L)}}{\partial X_I^{(L)}} * \frac{\partial X_I^{(L)}}{\partial w_{IJ}^{(L-1)}}$$

Chain rule!

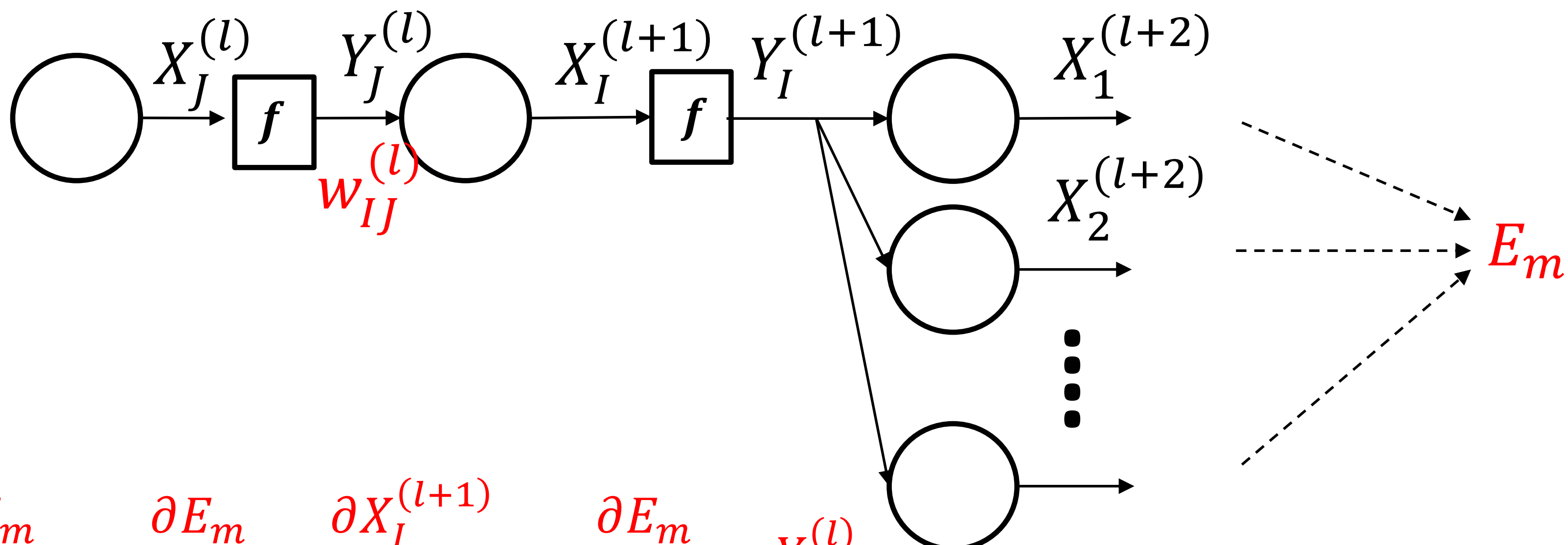
$$= \frac{\partial E_m}{\partial Y_I^{(L)}} * \frac{\partial Y_I^{(L)}}{\partial X_I^{(L)}} * \frac{\partial \left(\sum_{j=1}^{N(L-1)} w_{Ij}^{(L-1)} Y_j^{(L-1)}\right)}{\partial w_{IJ}^{(L-1)}}$$

$$= \frac{\partial E_m}{\partial Y_I^{(L)}} * \frac{\partial Y_I^{(L)}}{\partial X_I^{(L)}} * Y_J^{(L-1)}$$

- Note that the other $X_i^{(L)}$'s are not taken into account, because they do not depend on $w_{IJ}^{(L-1)}$ at all

Back propagation Algorithm

- To get $\frac{\partial E_m}{\partial w_{ij}^{(l)}}$ for any general layer l

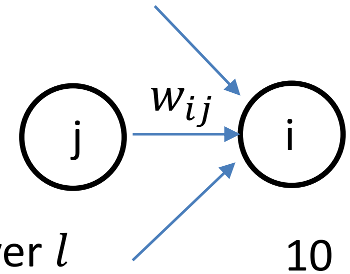


$$\frac{\partial E_m}{\partial w_{IJ}^{(l)}} = \frac{\partial E_m}{\partial X_I^{(l+1)}} * \frac{\partial X_I^{(l+1)}}{\partial w_{IJ}^{(l)}} = \frac{\partial E_m}{\partial X_I^{(l+1)}} * Y_J^{(l)}$$

$$\frac{\partial E_m}{\partial X_I^{(l+1)}} = \sum_{i=1}^{N(l+2)} \frac{\partial E_m}{\partial X_i^{(l+2)}} * \frac{\partial X_i^{(l+2)}}{\partial X_I^{(l+1)}} = \sum_{i=1}^{N(l+2)} \frac{\partial E_m}{\partial X_i^{(l+2)}} * \frac{\partial X_i^{(l+2)}}{\partial Y_I^{(l+1)}} * \frac{\partial Y_I^{(l+1)}}{\partial X_I^{(l+1)}}$$

Back propagation Algorithm

1. Initialize all weights to small randomly chosen values, e.g. [-1,1]
2. Let u(m) & d(m) be the training input/output examples
3. For m=1 to M:
   i. Present u(m) to the network and compute the hidden layer outputs and final layer outputs
   ii. Use these outputs in a backward scheme to compute the partial derivatives of error fn. w.r.t. to the weights of each layer
   iii. Update weights: $w_{ij}^{[l]}(new) = w_{ij}^{[l]}(old) - \alpha \frac{\partial E_m}{\partial w_{ij}^{[l]}}$
4. Compute total error (stop in case of convergence)



Note: l refers to layer l

Disadvantages of Back propagation

- Can often be slow in reaching the min (i.e., sometimes tens of thousands of iterations)
  - Especially close to min
  - Too small α → very small steps & slow to reach min
  - Too large α → leads to oscillations & possibly not converging at all
  - Use variable α (start large then decrease it)

Disadvantages of Back propagation

- Prone to get stuck in local minima
  - This problem could be alleviated to some extent by repeating the training many times, each time from a different set of initial weights

Types of Weight Update

- Batch or epoch update, i.e, **Gradient Descent**:
  - Present **full set of training examples** (batch of examples)
  - Compute the error of each example
  - Compute gradient of the batch (based on cost function of the whole batch)
  - Update weights based on this batch gradient
  - Do another iteration … and so on
  - Advantages:
    - Optimization is more consistent
  - Disadvantages:
    - Slow (too long per iteration)

Types of Weight Update

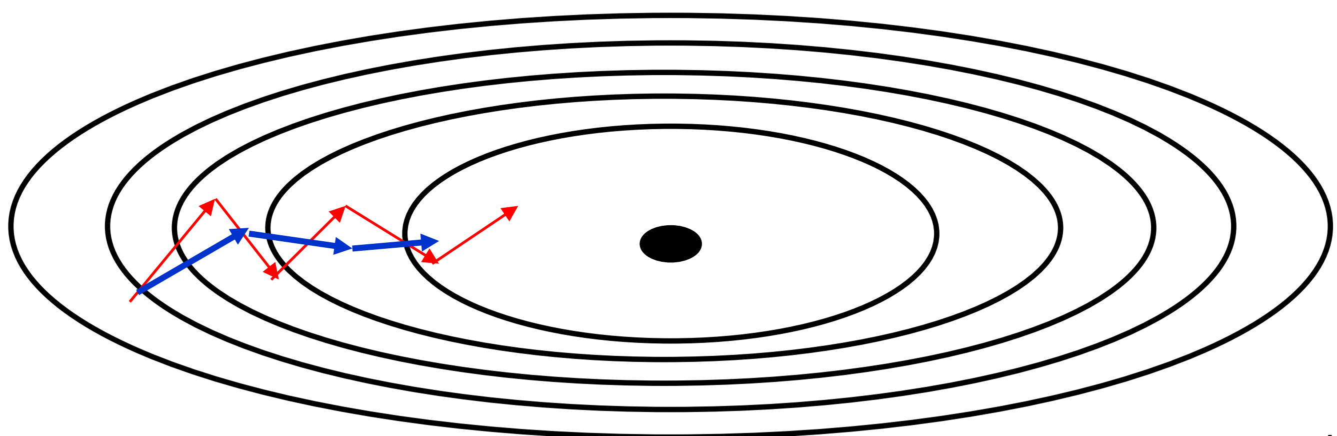
- Sequential update, i.e., **Stochastic Gradient Descent**:
  - Present a training pattern, then update the weights (according to $\frac{\partial E_m}{\partial w}$), then present the next one … and so on
  - After finishing all patterns, do another iteration starting from m = 1
  - Advantages:
    - Faster compared to gradient descent, i.e., full-batch
  - Disadvantages:
    - Hard to converge: "stochastic" since it depends on every single example; however, **in practice** being close to minimum is **reasonably good**
    - Loss speedup from vectorization
  - In practice for large datasets SGD is preferred to GD

Types of Weight Update

- Mini-Batch :
  - Present **subset** of training examples (mini-batch of examples)
  - Compute the error of each example
  - Compute gradient of the mini-batch (based on cost function of this mini-batch)
  - Update weights based on this mini-batch gradient
  - Move to another mini-batch & after finishing all mini-batches do another iteration … and so on
  - Advantages:
    - Fast

Other Optimization Algorithms

- Gradient descent with momentum
  - Smooth-out the steps of the gradient descent using a moving average of the derivatives
  - Get faster learning in the intended direction & avoid oscillations



Faster learning

Slower learning

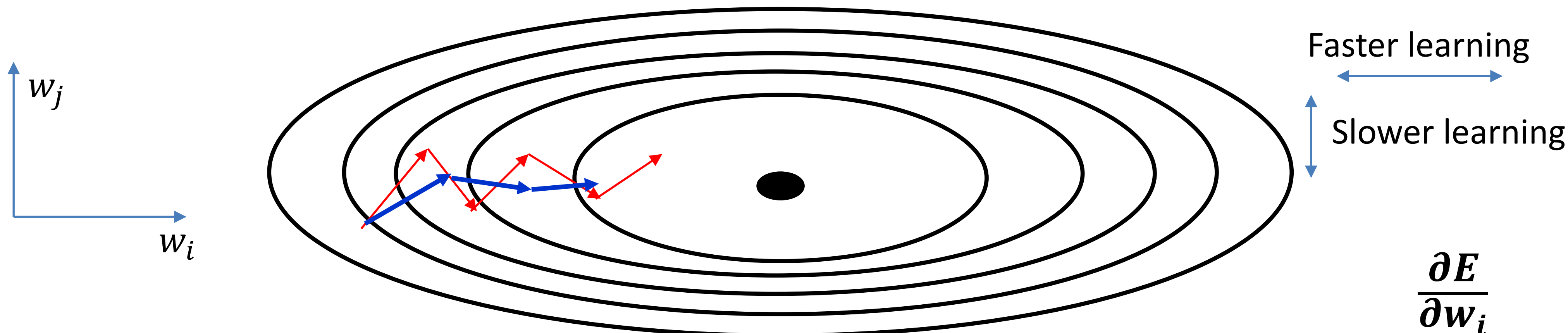
$$\boldsymbol{DW = \beta\, DW + (1-\beta)\frac{\partial E}{\partial W}}$$

$$\boldsymbol{W = W - \alpha\, DW}$$

$DW = 0$ initially

Other Optimization Algorithms

- RMSProp
  - Slow-down learning in unintended directions
  - Avoid oscillations



$$\boldsymbol{w_i = w_i - \alpha \frac{\frac{\partial E}{\partial w_i}}{\sqrt{Sw_i} + \varepsilon}}$$

$$\boldsymbol{Sw_i = \beta\, Sw_i + (1-\beta)\left[\frac{\partial E}{\partial w_i}\right]^2}$$

small

$$\frac{\partial E}{\partial w_i} < \frac{\partial E}{\partial w_j}$$

$$\boldsymbol{w_j = w_j - \alpha \frac{\frac{\partial E}{\partial w_j}}{\sqrt{Sw_j} + \varepsilon}}$$

$$\boldsymbol{Sw_j = \beta\, Sw_j + (1-\beta)\left[\frac{\partial E}{\partial w_j}\right]^2}$$

large

Other Optimization Algorithms

- Adam (combines both RMSProp & momentum)

$$\boldsymbol{Dw_i = \beta_1 Dw_i + (1-\beta_1)\frac{\partial E}{\partial w_i}}$$

$$\boldsymbol{Sw_i = \beta_2 Sw_i + (1-\beta_2)\left[\frac{\partial E}{\partial w_i}\right]^2}$$

$$\boldsymbol{w_i = w_i - \alpha\frac{Dw_i}{\sqrt{Sw_i}+\varepsilon}}$$

Regularization

- **Used to prevent overfitting**
  - Intuition: set the weights of some hidden nodes to zero to simplify the network, i.e., smaller network

- L_2 regularization (aka weight decay): $\boldsymbol{J} = \frac{\mathbf{1}}{\boldsymbol{M}} \sum_{\boldsymbol{m}=\mathbf{1}}^{\boldsymbol{M}} \boldsymbol{E}_{\boldsymbol{m}} + \frac{\boldsymbol{\lambda}}{\mathbf{2}\boldsymbol{M}} \left\| \underline{\boldsymbol{W}} \right\|_{\mathbf{2}}^{\mathbf{2}}$
  - $\left\| \underline{W} \right\|_2^2 = \sum_j w_j^2 = \underline{W}^T \underline{W}$

- L_1 regularization: $\boldsymbol{J} = \frac{\mathbf{1}}{\boldsymbol{M}} \sum_{\boldsymbol{m}=\mathbf{1}}^{\boldsymbol{M}} \boldsymbol{E}_{\boldsymbol{m}} + \frac{\boldsymbol{\lambda}}{\mathbf{2}\boldsymbol{M}} \left\| \underline{\boldsymbol{W}} \right\|_{\mathbf{1}}$
  - $\left\| \underline{W} \right\|_1 = \sum_j |w_j|$

- $\underline{W}$ is the weights vector, thresholds not necessary to be included

- L_2 regularization is used more often

- λ is the regularization parameter (hyper-parameter to be tuned)

Dropout Regularization

- **Used to prevent overfitting**

- Intuitions:
  - Eliminate some nodes to simplify the network based on some probability, i.e., smaller network
  - As if you train smaller networks on individual training examples
  - Cannot rely on any one feature, so spread weights

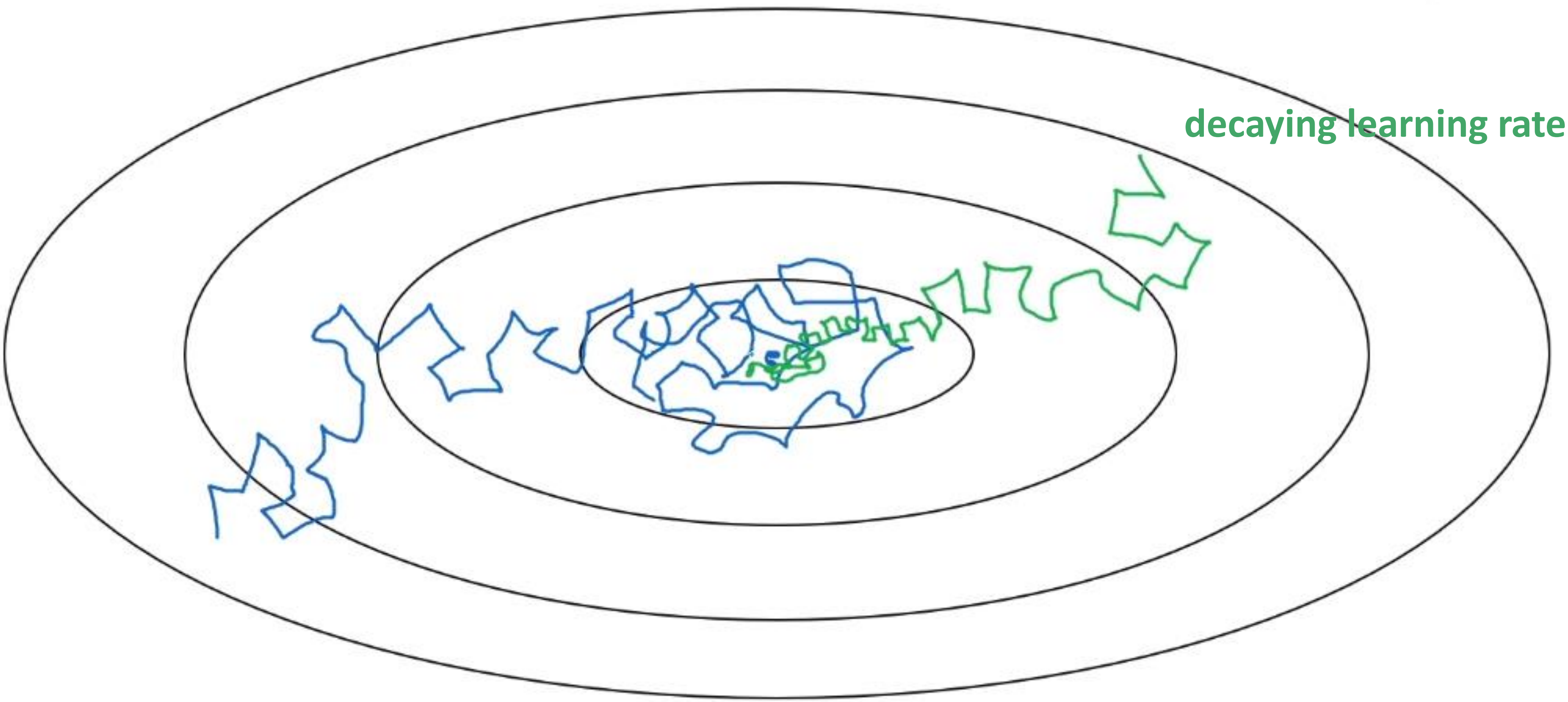
- For each layer set a dropout probability
  - Each node within that layer may get eliminated based on that probability

Guidelines for Training

- Learning rate α :
  - Too small. Convergence will be slow.
  - Too large: we will oscillate around the minimum.
  - Some methods propose varying rate. i.e., learning rate decay.
  - When learning does not go well, consider using smaller learning rate.

Learning Rate Decay

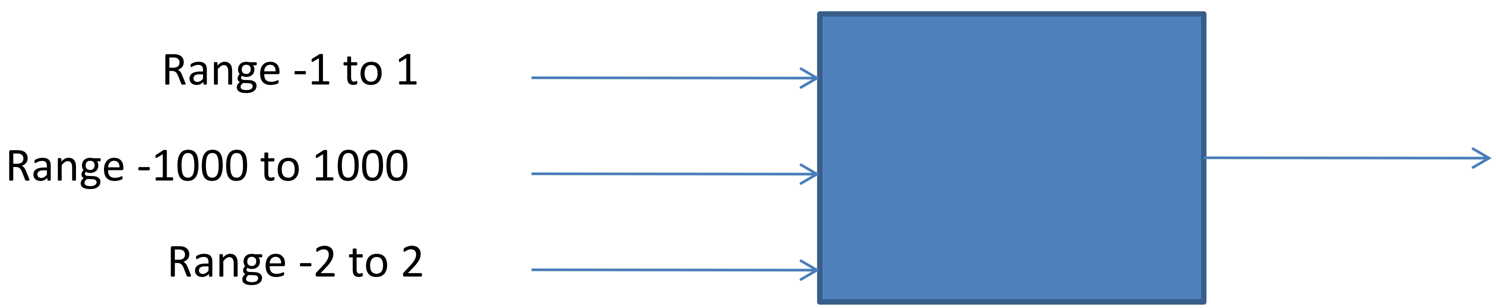
- Gradient descent with small mini-batch size



Source: Andrew Ng

Input and Output Normalization

- Input and Output normalization
  - Inputs have to be approximately in the range of 0 to 1 or -1 to 1



  - $x = (u - u_{min})/(u_{max} - u_{min})$
  - $x = (u - \text{Mean}(u)) / \text{st dev}(u)$

Train/Dev/Test Partition

- Best practice:
  - **Training: 60% ,**
    Validation (Dev): 20% & Test: 20%
  - In case of big data, e.g., 10^6, then 98%, 1% & 1%

- Test set should be used only once, at the very end of the design

Machine Learning Recipe

- Train the network and evaluate first on the training data
  - If bias is high, i.e., underfitting (performance is bad on the training set itself), then:
    - **Bigger network** (more hidden nodes or more hidden layers) → works most of the time
    - Train longer → works sometimes
  - Check for bias again and keep changing until a good bias is reached

- Check for variance, i.e., performance on Dev set
  - If variance is high, i.e., overfitting (performance is bad on the validation set), then:
    - **More data** (if possible)
    - **Regularization**
  - Check again for bias first, then after that check for variance and so on until you reach a good bias & good variance

- Search for better NN architecture that better suits the problem (sometimes may work)

Hyper-Parameters Tuning

- Learning rate α

1st in importance

- Momentum parameter $\beta \approx 0.9$
- Number of hidden nodes
- Mini-batch size

2nd in importance

- Num of layers
- Learning-rate decay

3rd in importance

- Adam parameters $\beta_1 \approx 0.9$, $\beta_2 \approx 0.999, \in \approx 10^{-8}$

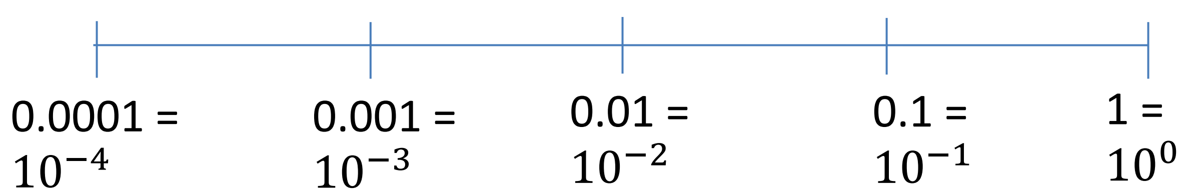
Not likely to make change!

Tuning Process

- Try random values: don't use a grid
  - Better exploration of important parameters
  - Consider the example on the board

- Coarse to fine scheme
  - Focus more on good regions

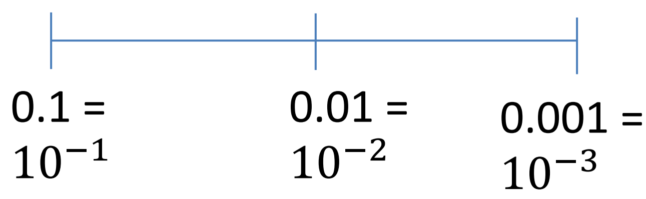
- Use appropriate scale
  - Do not sample uniformly
  - Use logarithmic scale
  - E.g., α range is [0.0001,1] linear scale scaling will give more weight to the values between 0.1 & 1, however, logarithmic scale:



Sample $r \sim [-4, 0]$
then $\alpha = 10^{r}$

Tuning Process

- Use appropriate scale
  - More example: let β range is [0.9,0.999]
  - Sample from $1-\beta$, i.e., [0.1,0.001], using log scale



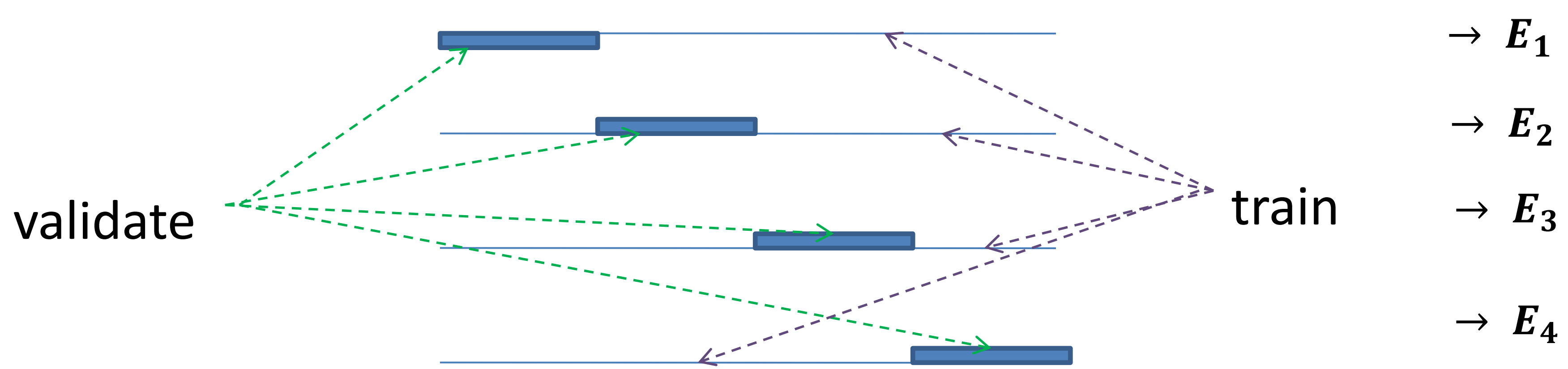
Sample $\boldsymbol{r} \sim [\mathbf{-3}, \mathbf{-1}]$
then $\mathbf{1} - \boldsymbol{\beta} = \mathbf{10}^{\boldsymbol{r}}$
$\boldsymbol{\beta} = \mathbf{1} - \mathbf{10}^{\boldsymbol{r}}$

  - Sensitivity of β approaching has huge impact on the performance, i.e., momentum corresponds to averaging over the last $\frac{1}{1-\beta}$ examples
    - $\beta \sim [\mathbf{0.900}, \mathbf{0.9005}]$ → averaging over last 10 examples
    - $\beta \sim [\mathbf{0.999}, \mathbf{0.9995}]$ → 1000 to 2000 examples

K-Fold Cross Validation

- For parameter tuning over the training data
- Apply K-fold validation to the training set (usually k= 5).
- Example K=4

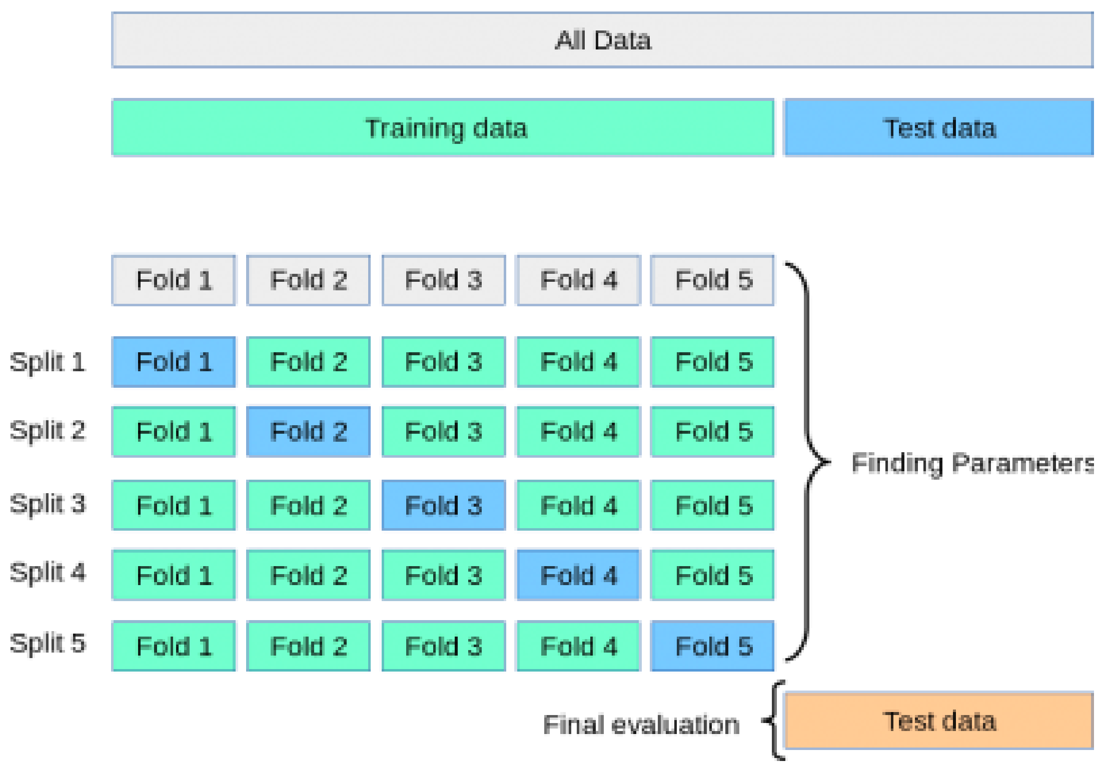
$$\boldsymbol{E_{VAL} = E_1 + E_2 + E_3 + E_4}$$



- Repeat for every parameter value, minimize $\boldsymbol{E_{VAL}}$

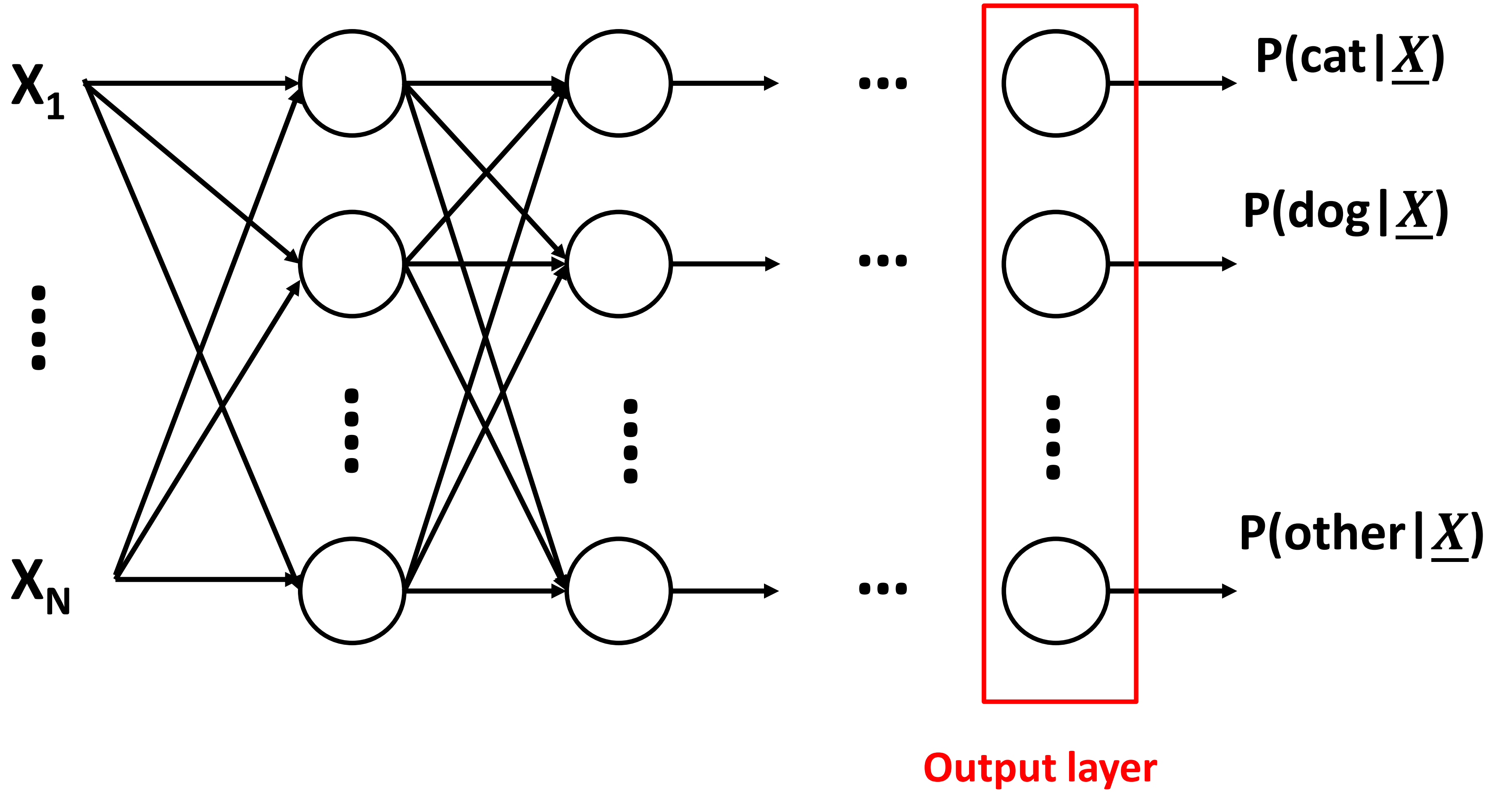
K-Fold Cross Validation

- Better than convention train-validation-test split
  - Just split not training and test (no need for validation set)
  - Not biased to the nature of splitting of the training and validation



Multi-Class Classification

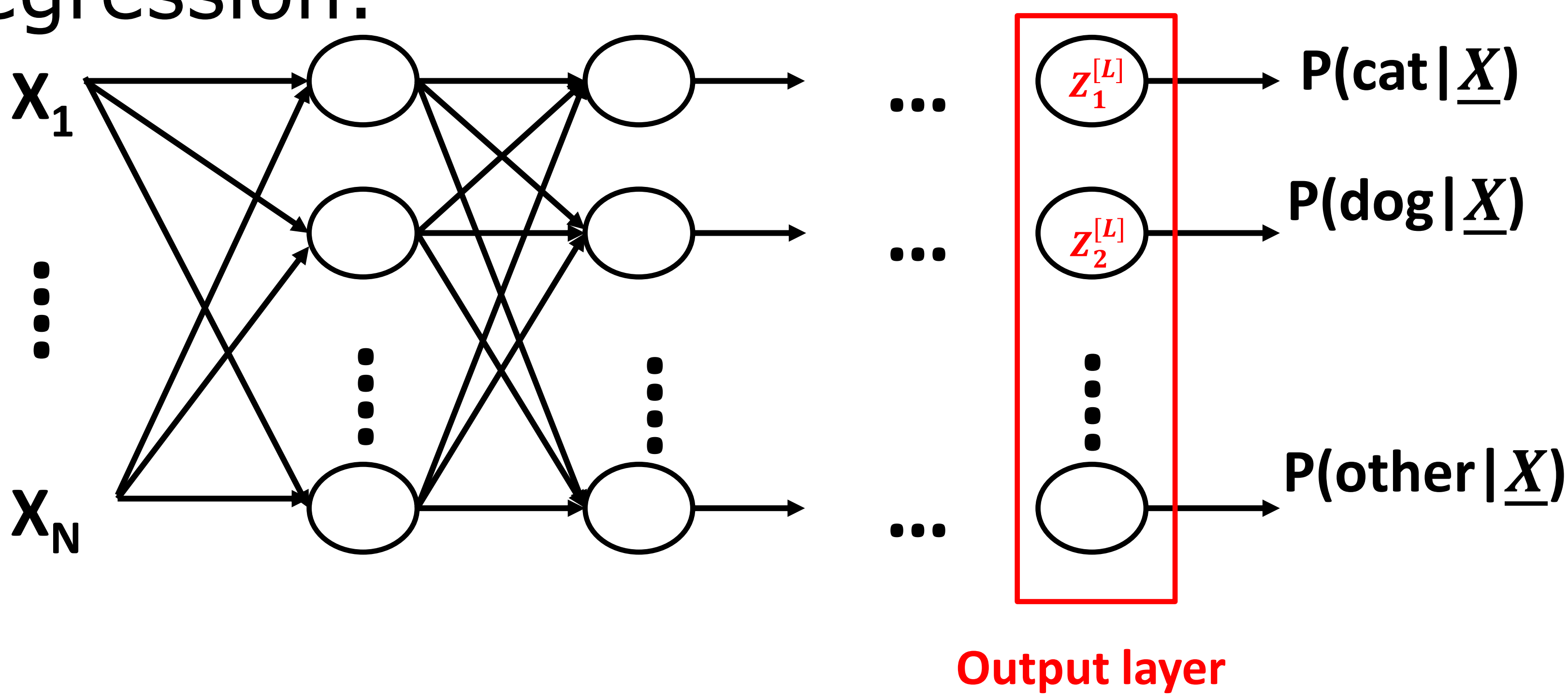
- E.g., an image is either of a cat, or dog, or duck or otherwise



Multi-Class Classification

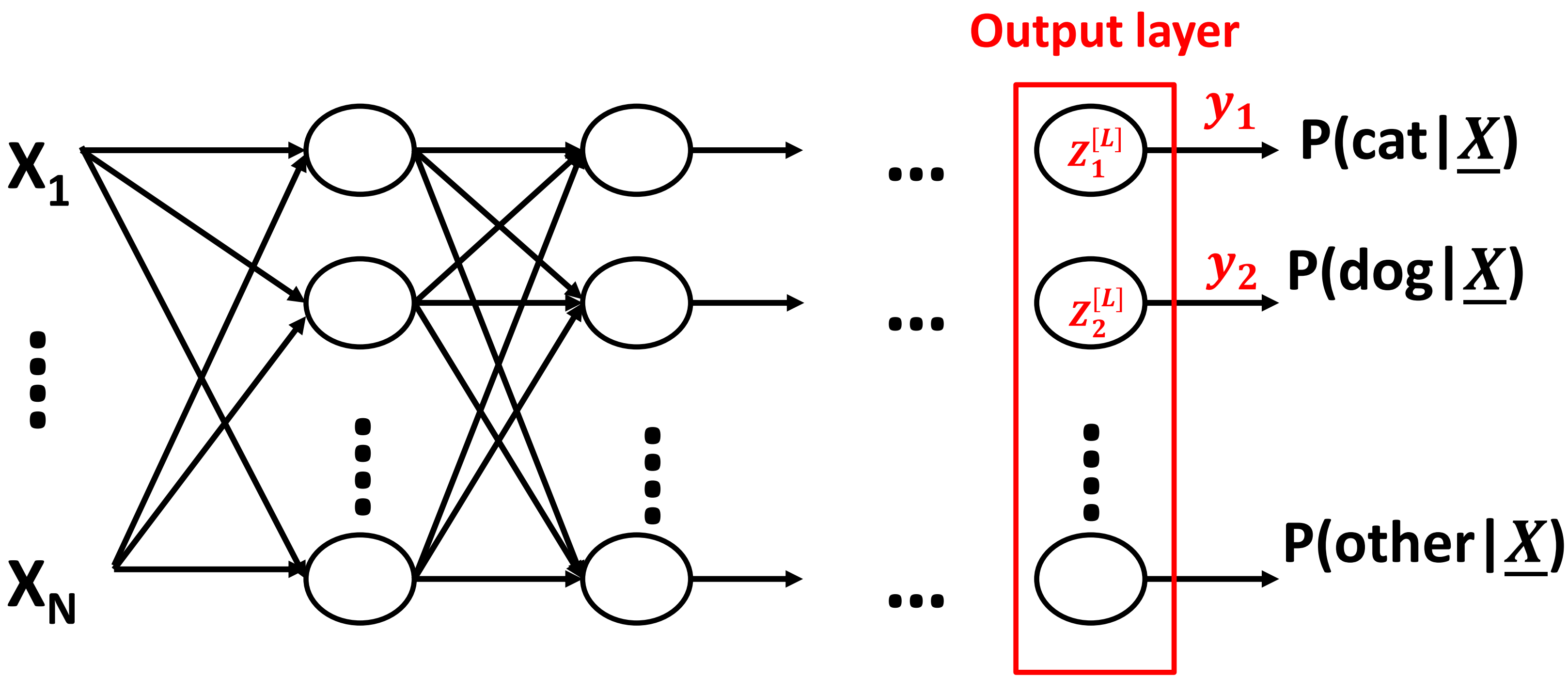
- Use sigmoid activation function in the output only in case of binary classification, i.e., two classes

- For multi-class classification use soft-max regression:



Multi-Class Classification

- For multi-class classification use soft-max regression:



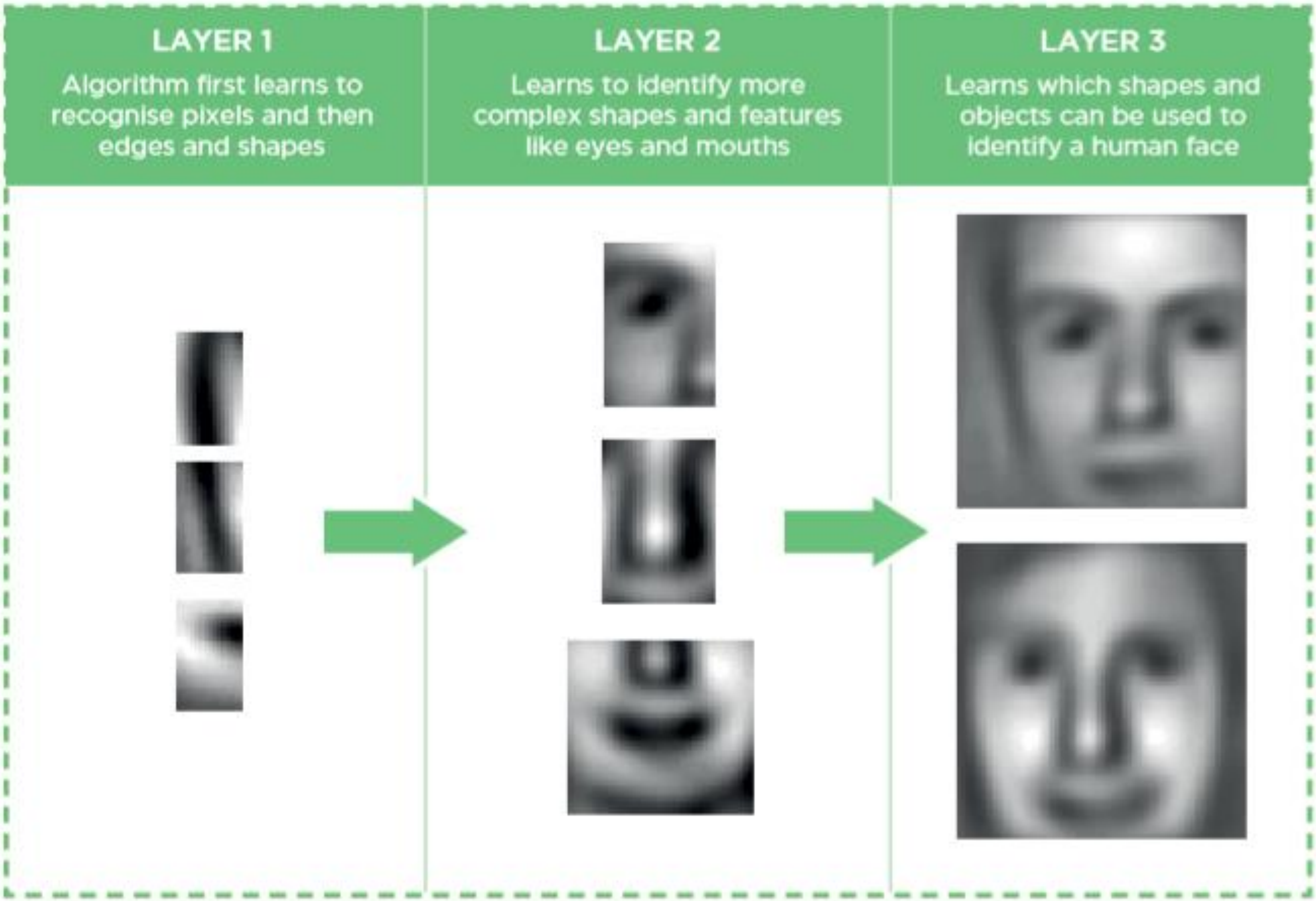
- $z_i^{[L]}$ output of node i at the output layer before applying any activation function
- y_i is the output after applying soft-max

$$y_i = \frac{e^{z_i^{[L]}}}{\sum_j e^{z_j^{[L]}}}$$

Convolutional Neural Networks (CNN)

- Mostly applied to imagery problems
- Layers extract features from input images, e.g., edge detection

Convolutional Neural Networks (CNN)



Convolutional Neural Networks (CNN)

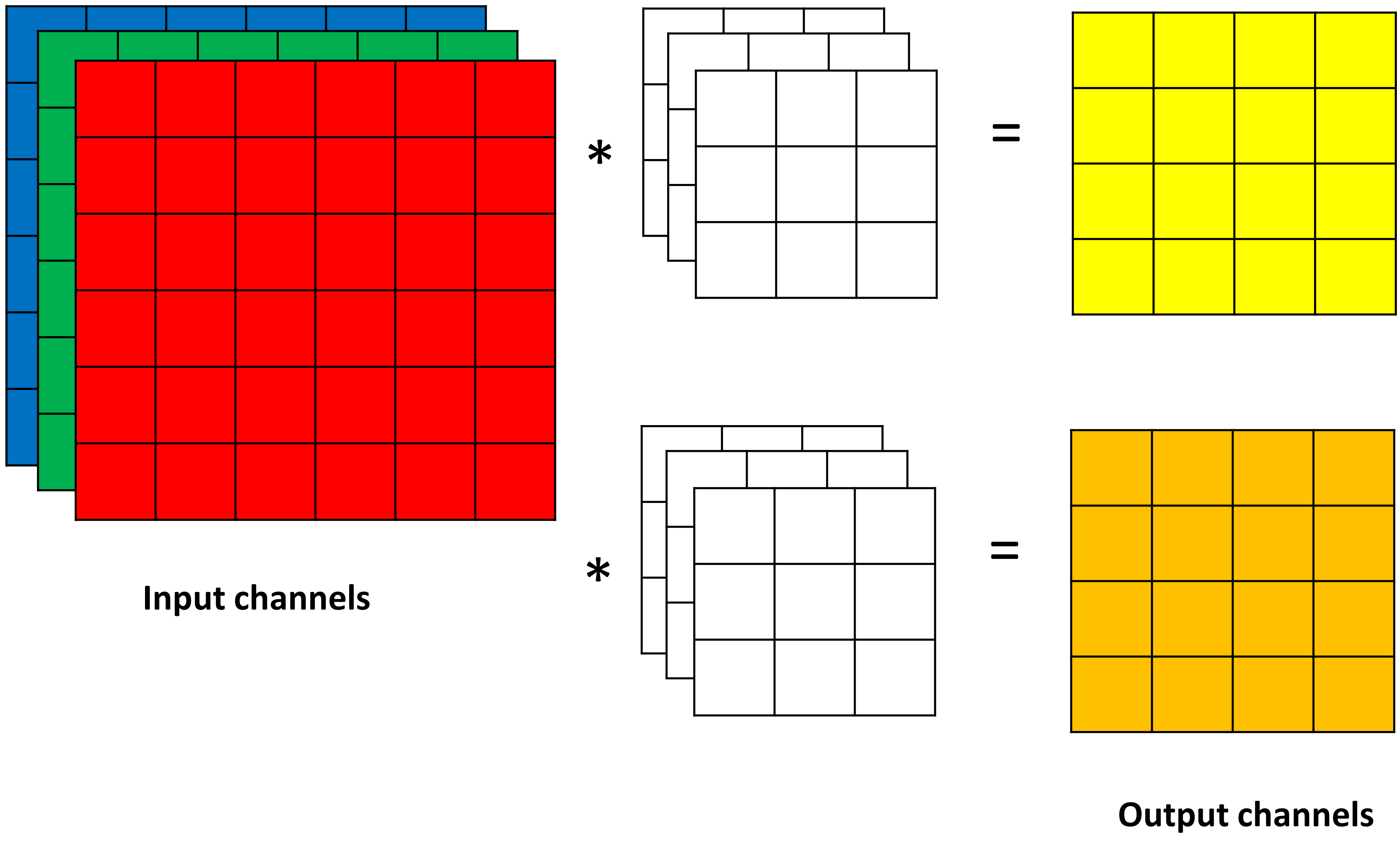
- Mostly applied to imagery problems
- Layers extract features from input images, e.g., edge detection
  - Convolution layer, i.e., filtering
  - Pooling Layer, i.e., reduce input (avg or max)
  - Fully Connected Layer, i.e., as in multi-layer NN, at the final layers

Vertical Edge Detection

$$
\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}
$$

convolution

Convolutional Neural Networks (CNN)



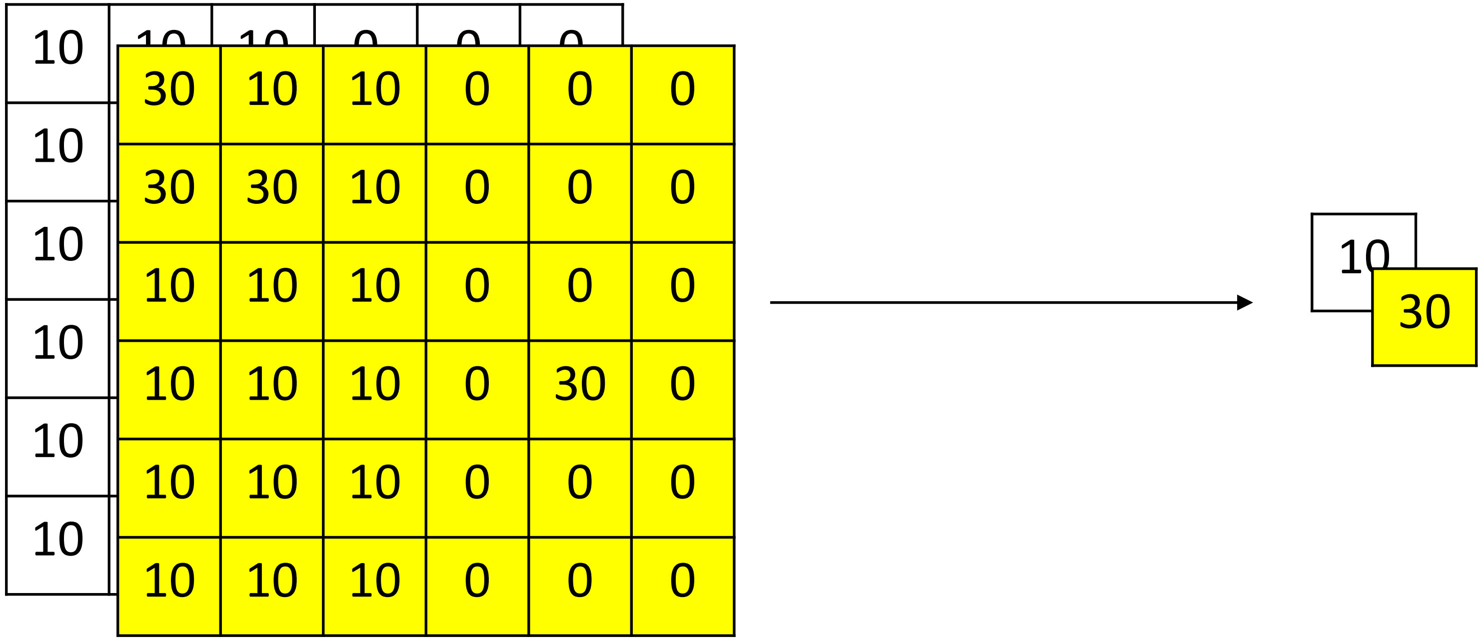
Max Pooling

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

10	10	0
10	10	0
10	10	0

What about average pooling?

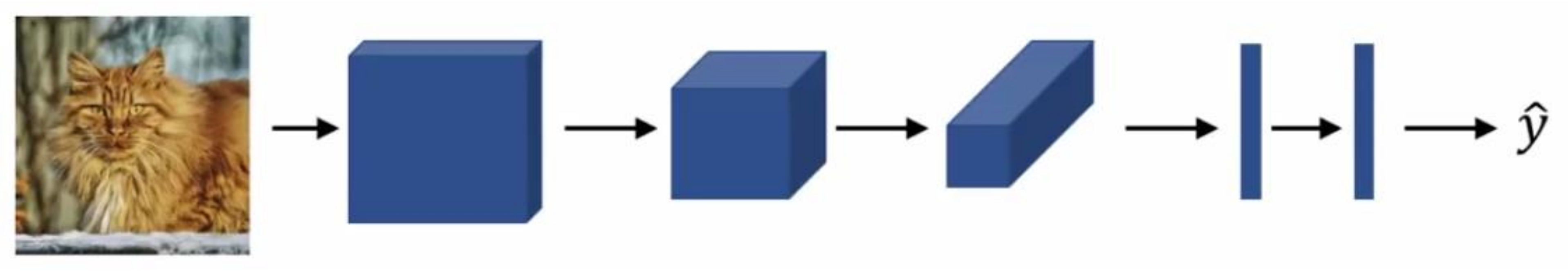
Global Max Pooling



What about global average pooling?

Convolutional Neural Networks (CNN)

- Learn filters' parameters and weights of fully connected layers



Source: Andrew Ng's Lectures

Convolutional Neural Networks (CNN)

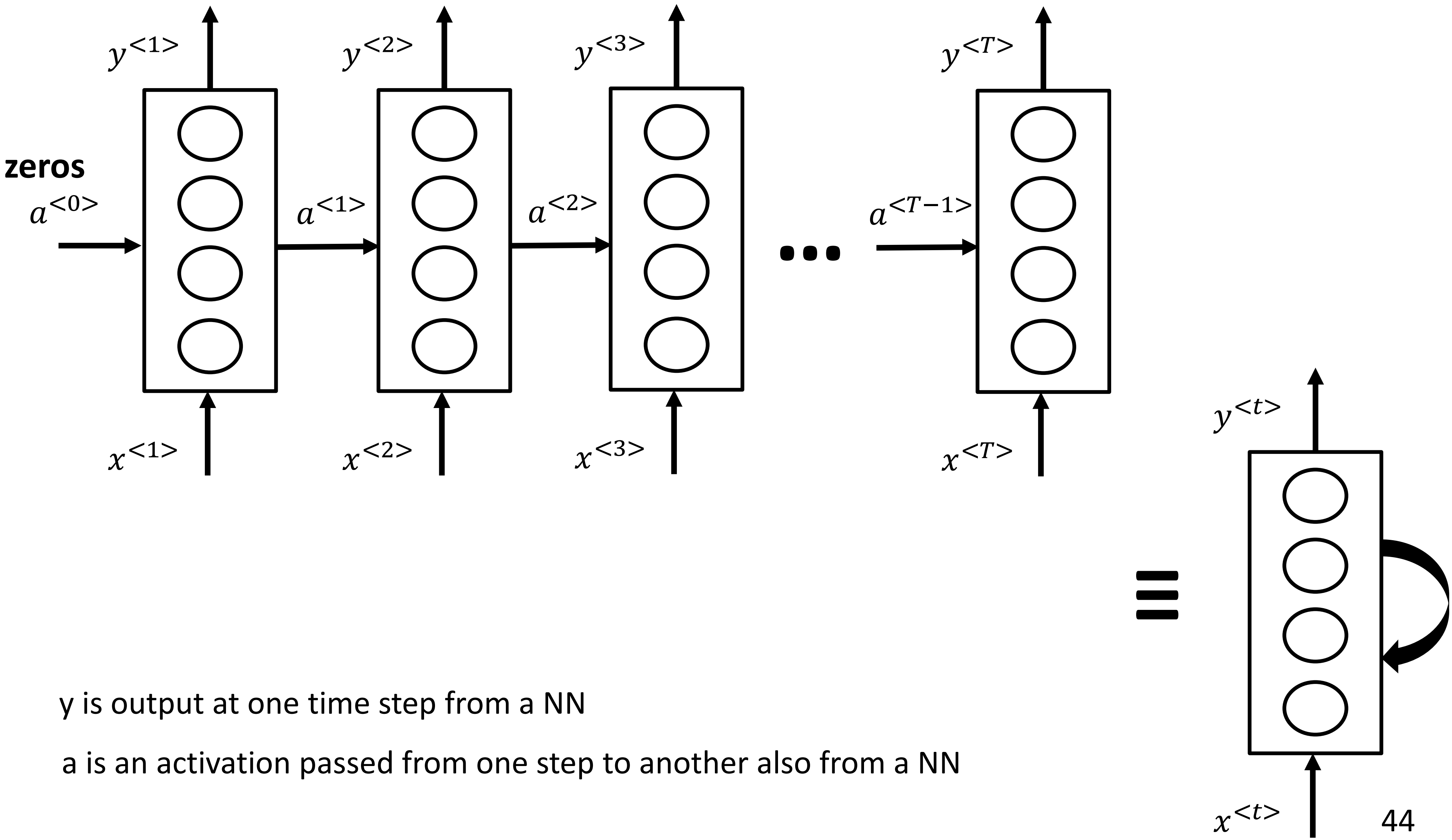
- Convolution leads to less memory footprint due to:
  - Parameter sharing (compared to fully connected layers)
  - Sparsity of connections (at each layer output depends on limited number of inputs)

Recurrent Neural Networks (RNN)

- Sequence models, e.g., speech recognition, sentiment classification, … etc.

- Inputs & outputs can have different lengths within the same dataset

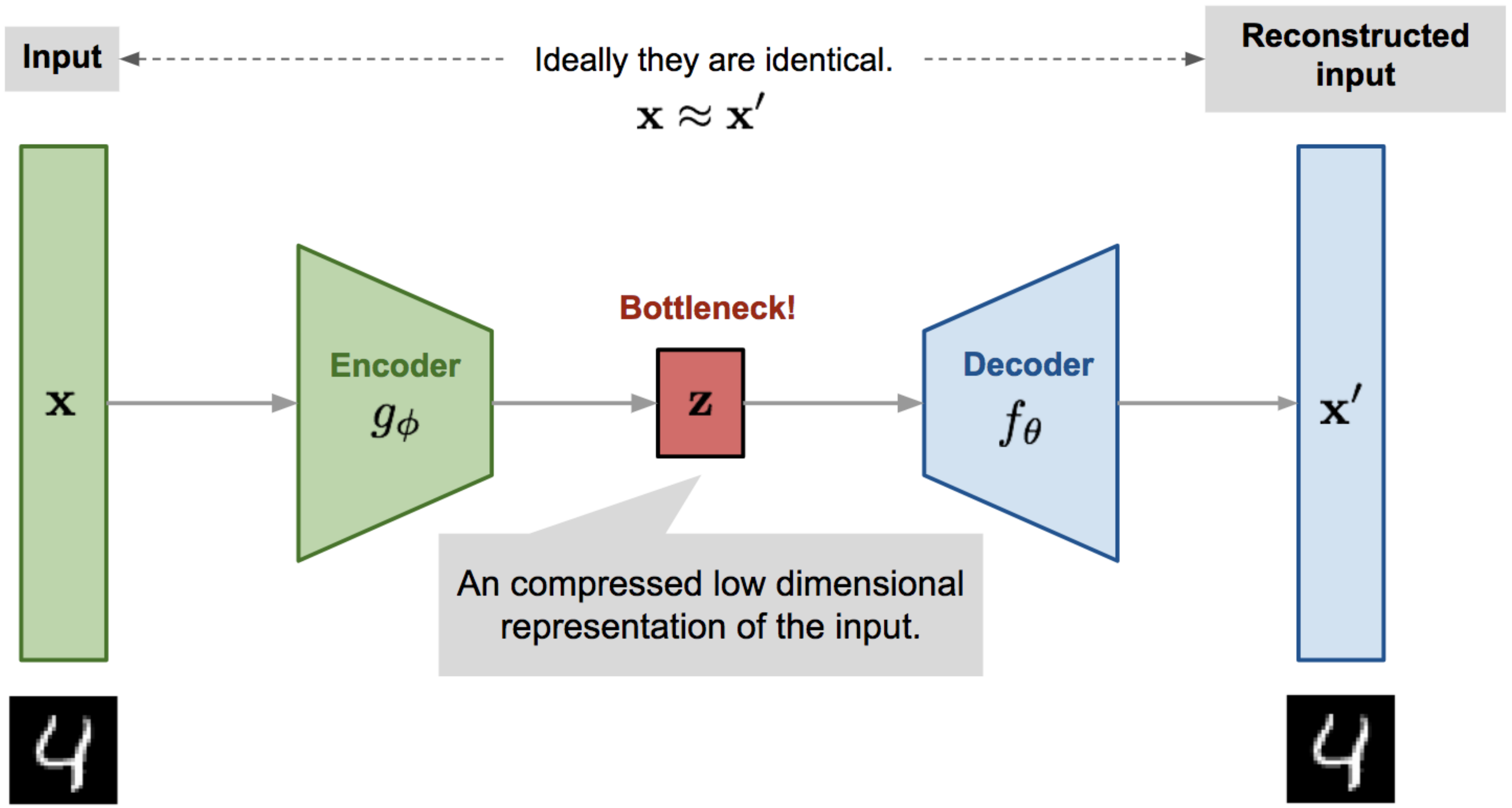
Recurrent Neural Networks (RNN)



y is output at one time step from a NN

a is an activation passed from one step to another also from a NN

Autoencoder Network



Source: Lilian Weng's Github blog

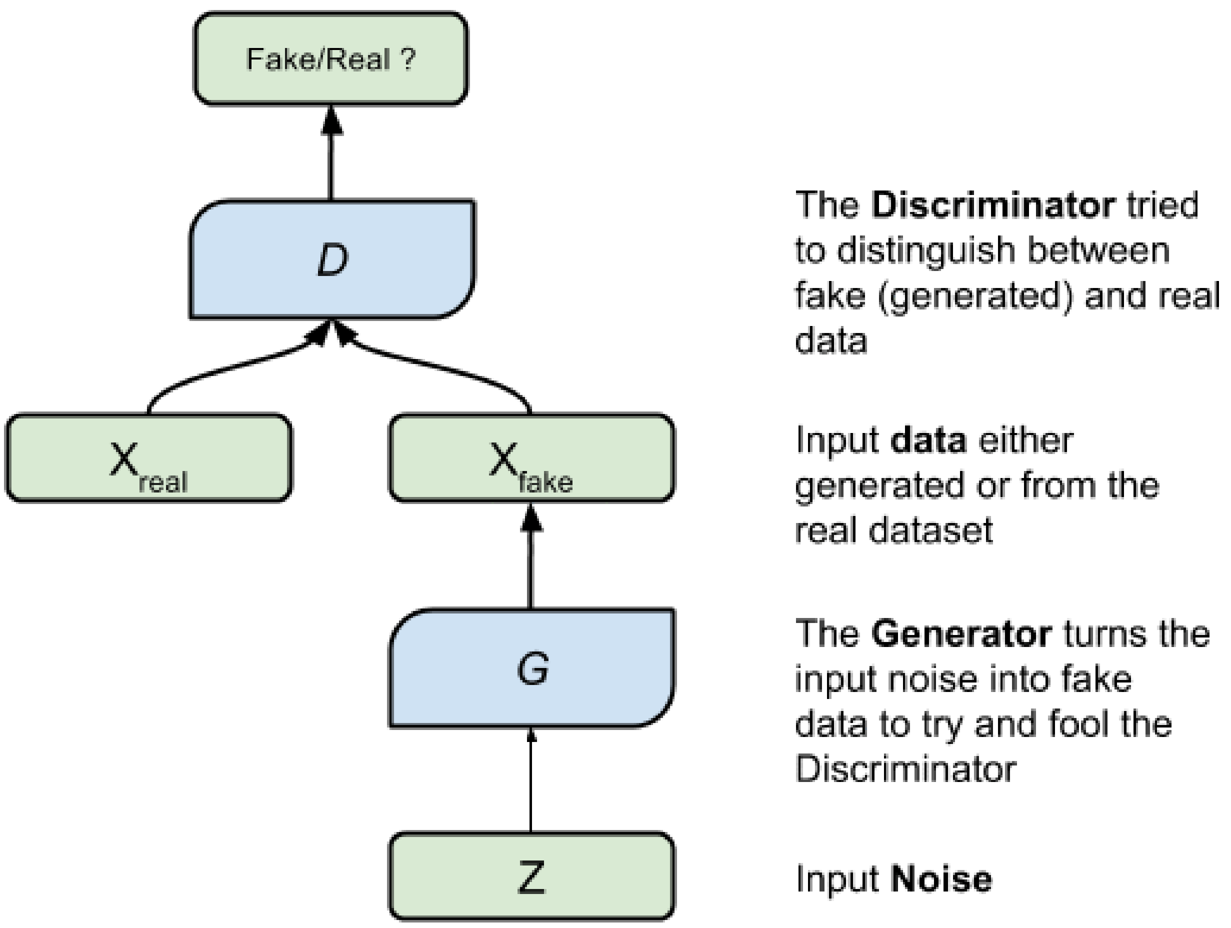
Autoencoder Network

- Unsupervised network

- Gives embedding
  - Better embeddings using supervised

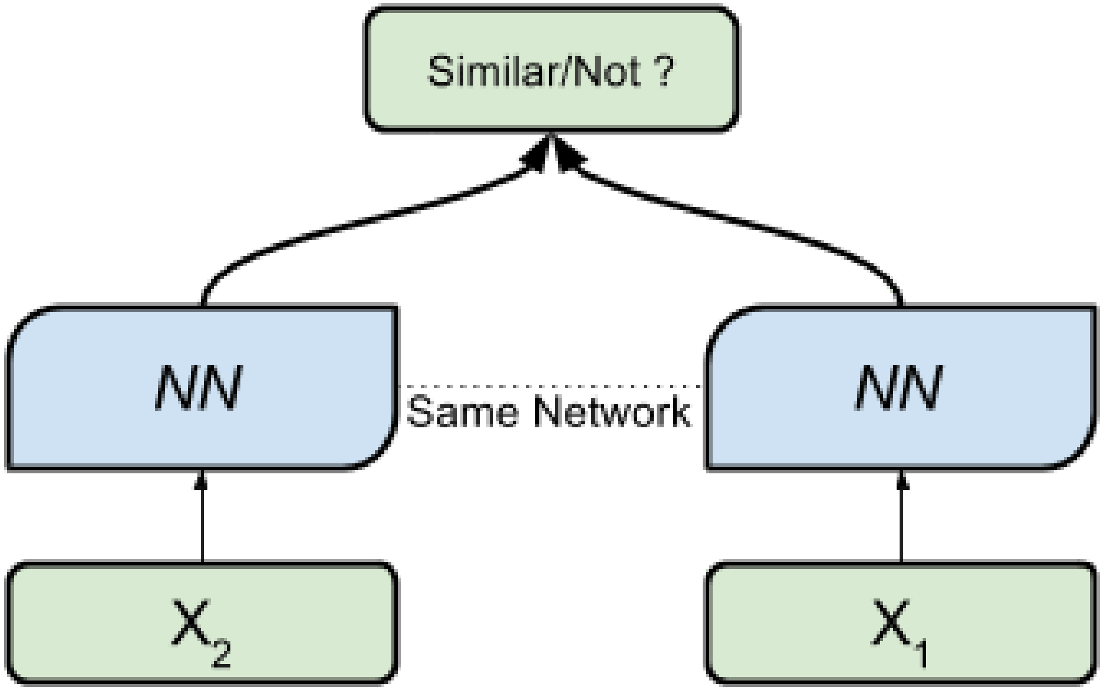
Generative Adversarial Network (GAN)

- Create a generative model of artificial data



Source: Guy Ernest, AWS Amazon Blogs

Siamese Networks



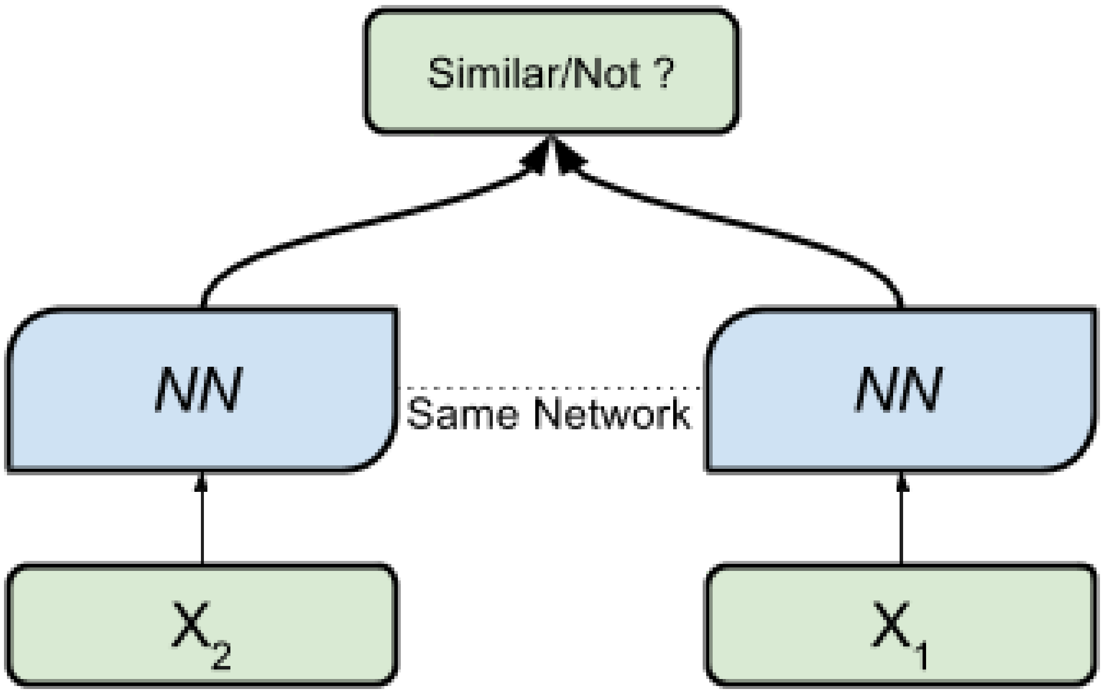
The **Distance Function** decides if the output vectors are close enough to be similar

The **Neural Network** transforms the input into a properties vector

Input Data (image, text, features…)

Source: Guy Ernest, AWS Amazon Blogs

Siamese Networks



The **Distance Function** decides if the output vectors are close enough to be similar

The **Neural Network** transforms the input into a properties vector

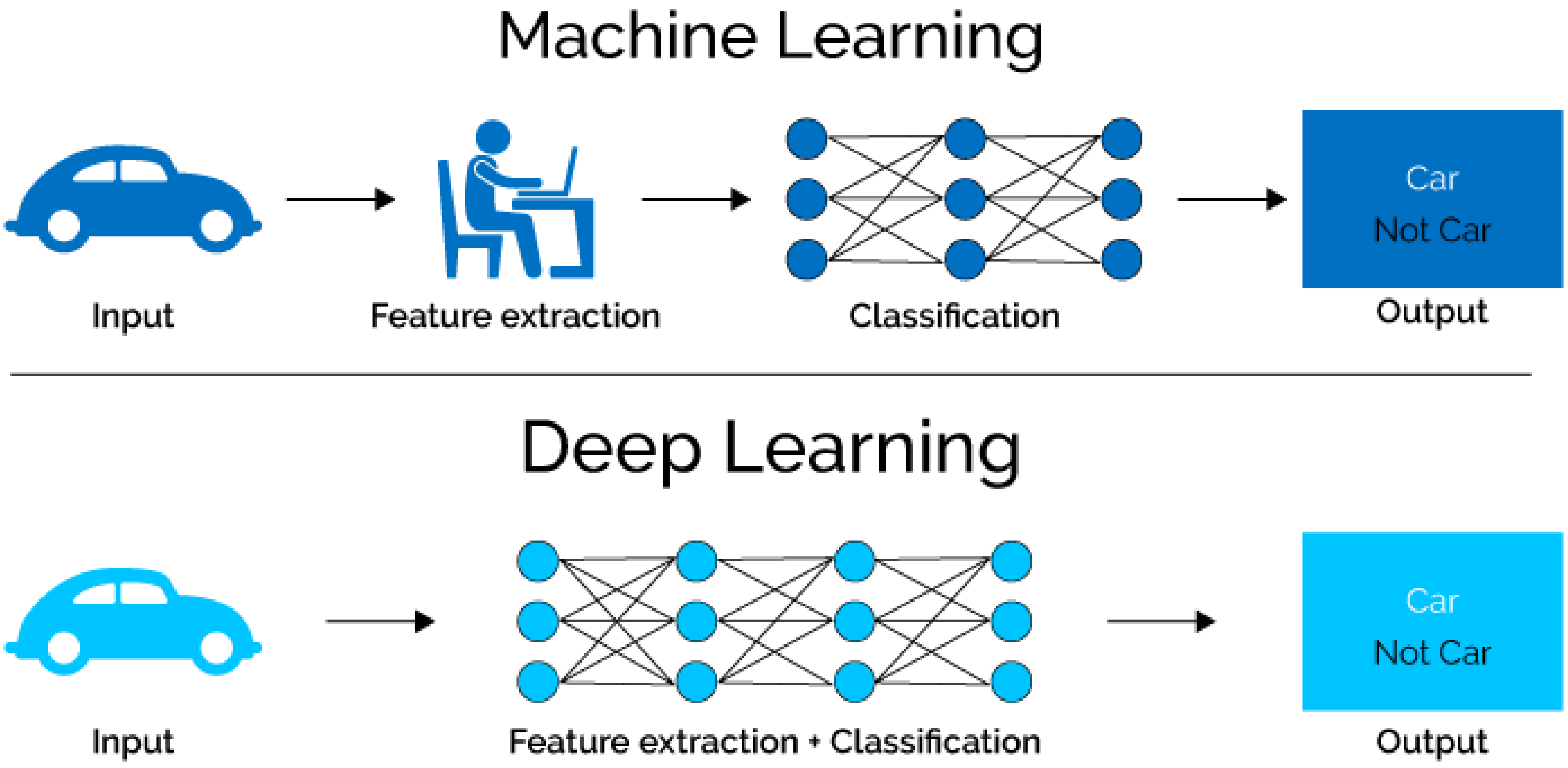
Input Data (image, text, features…)

Source: Guy Ernest, AWS Amazon Blogs

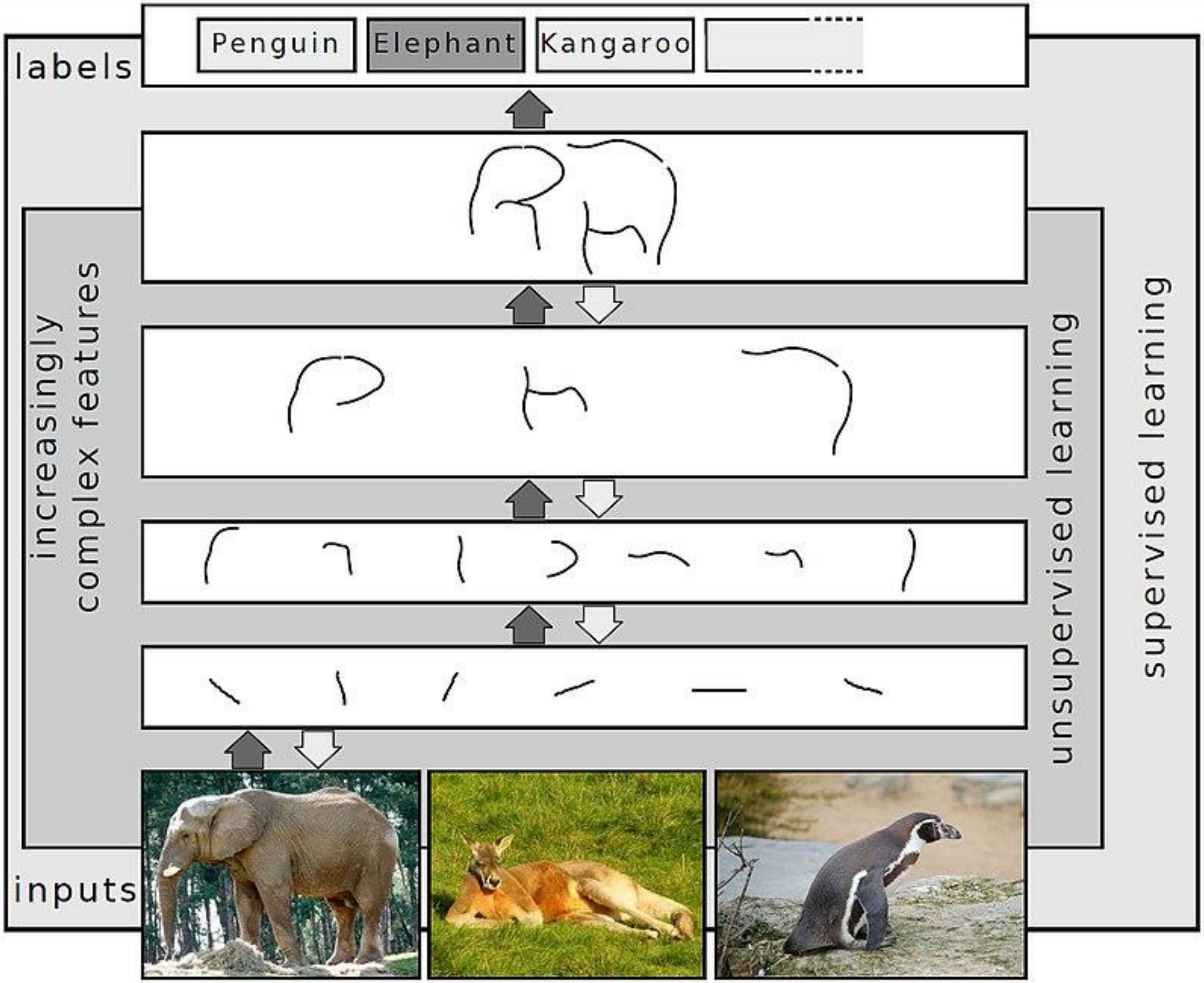
Deep Learning

- Subset of machine learning
- Multi-layered neural networks
- Raw data, i.e., end-to-end solution
- Requires big data & high computational power

Machine Learning vs Deep Learning

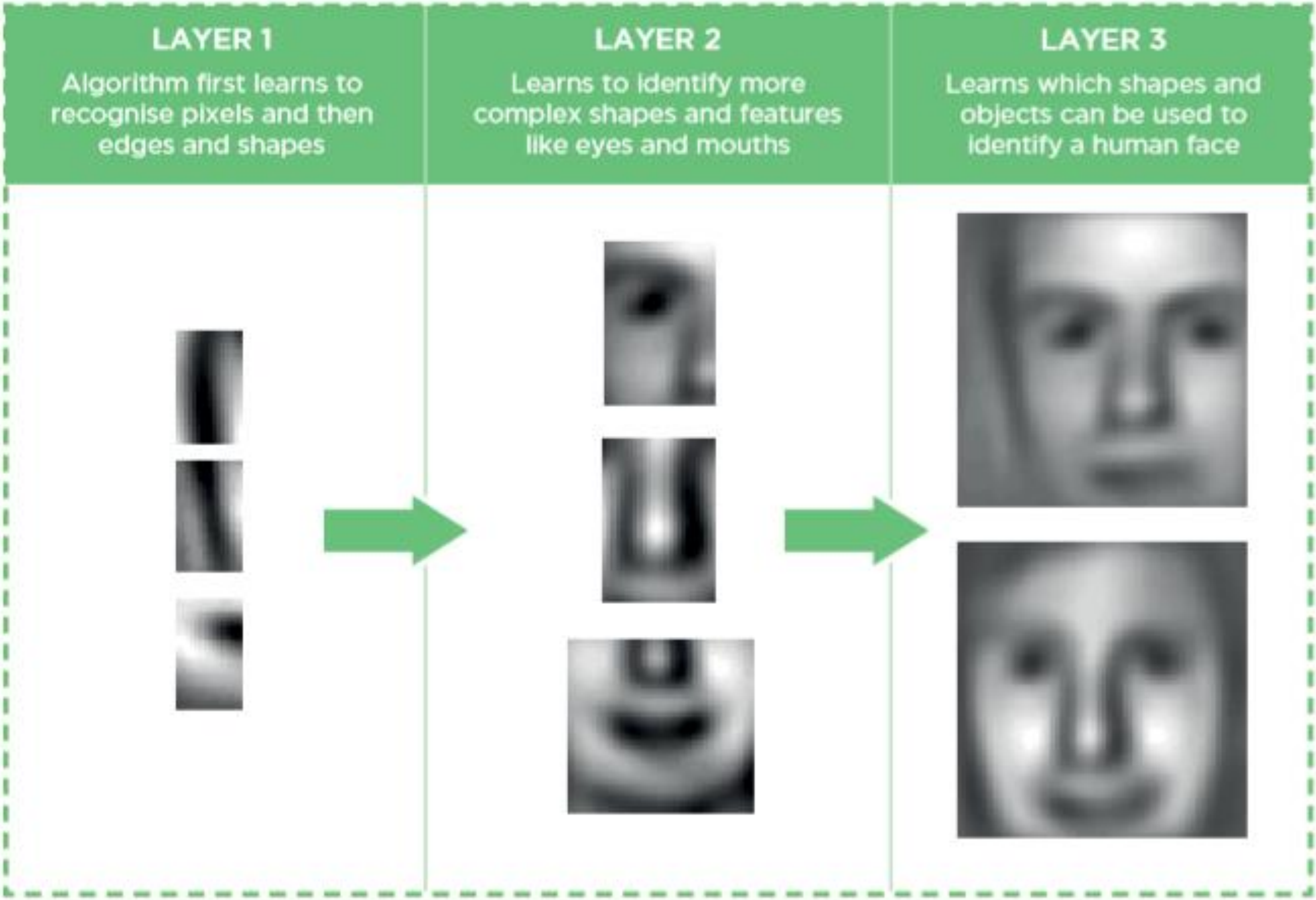


Machine Learning vs Deep Learning



Source: Hannes Schulz and Sven Behnke, 2012

Machine Learning vs Deep Learning



When to use Deep Learning?

- Big amount of **data**

  expensive!

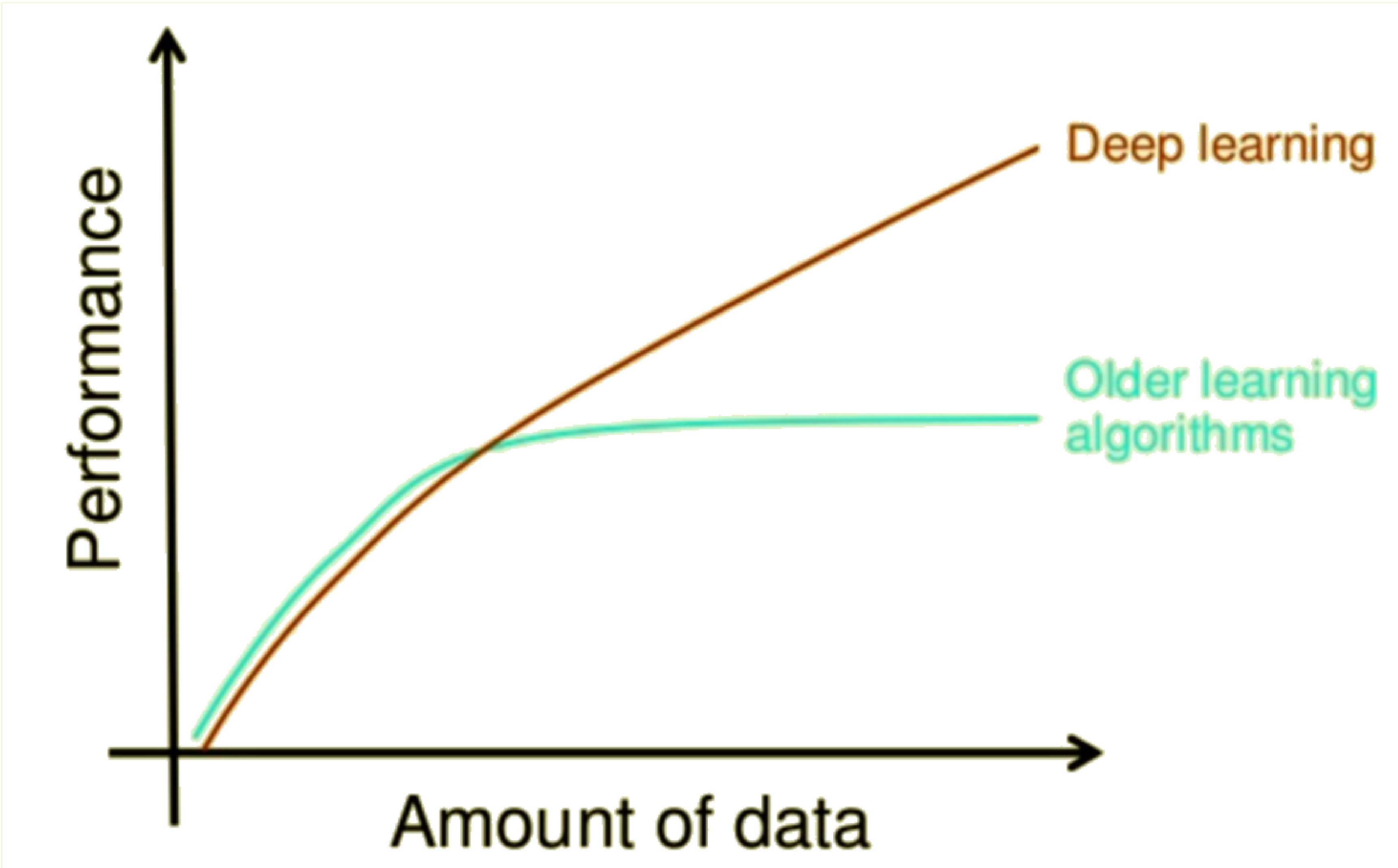
- Availability of high computational power

  expensive!

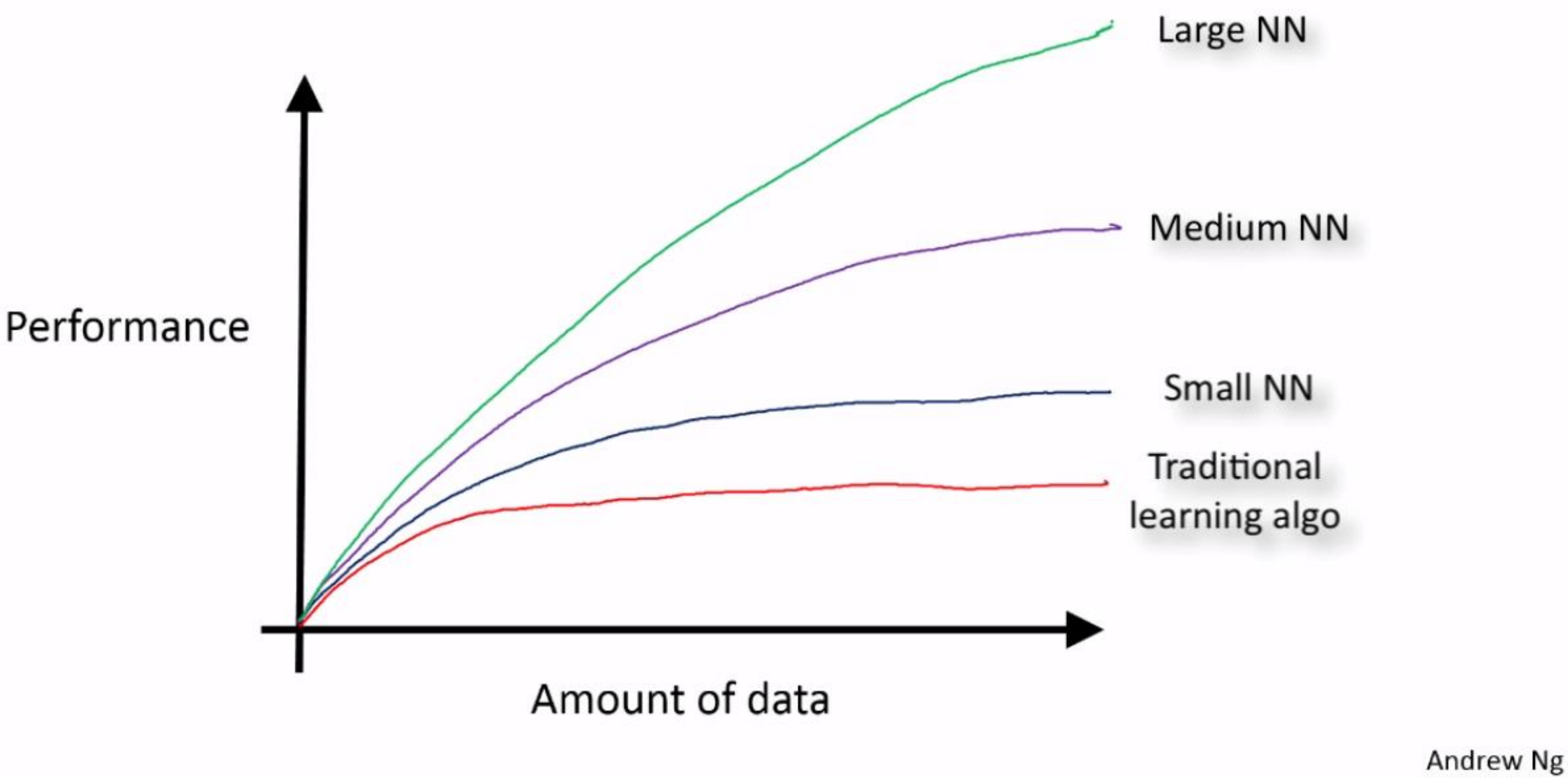
- Lack of domain understanding

- Complex problems (vision, NLP, speech recognition)

Scalability with Data Amount



Scalability with Data Amount



Potentials of AI

"If a typical person can do a mental task with less than one second of thought, we can probably automate it using AI either now or in the near future."

— Andrew Ng

Currently, there are some limitations!

Limitations of Deep Learning

- Lots of achievements in vision field

- Not a magic tool!

  - Lack of adaptability and generality compared to human-vision system

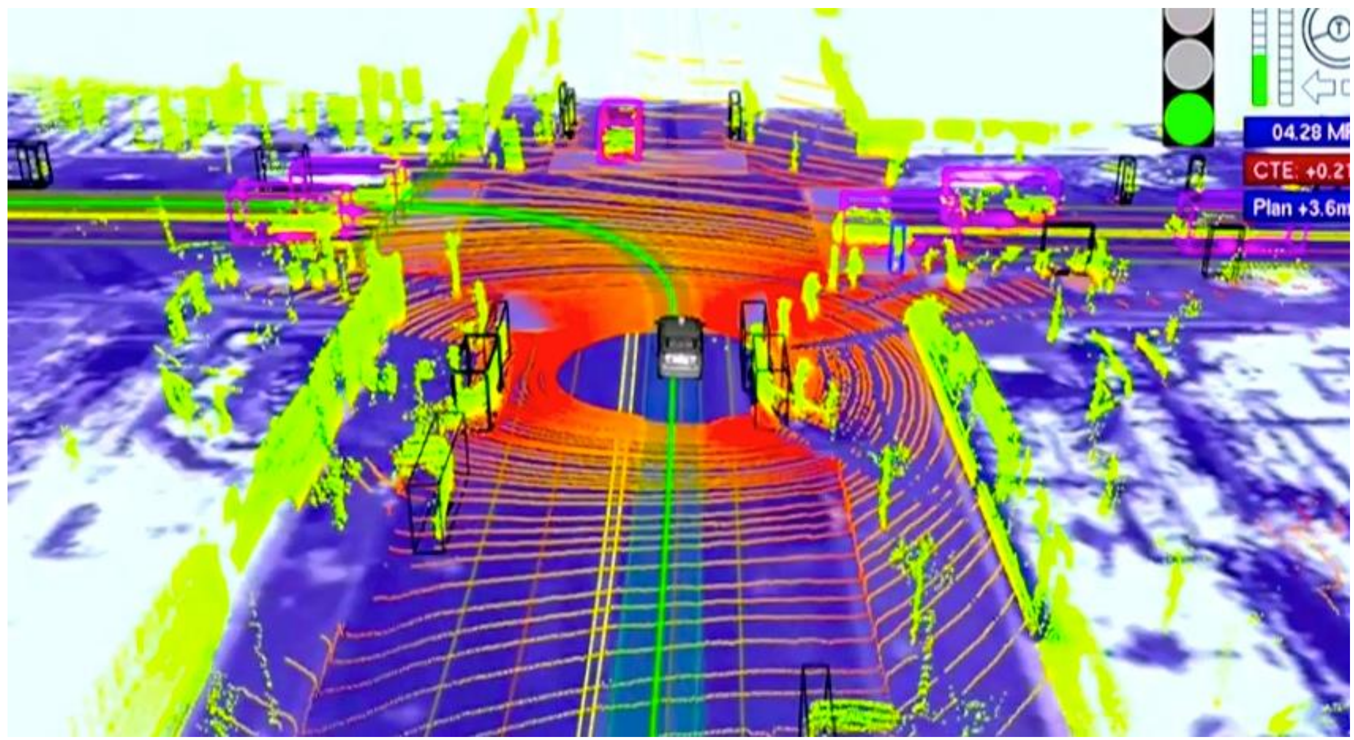
  - Not able to build general-intelligent machine

Limitations of Deep Learning

Source: Gartner Hype Cycle for AI, 2019

Limitations of Deep Learning

- Why cannot fit all real-world scenarios?



Source: Google

Source: Boston Dynamics

Limitations of Deep Learning

- Large amount of labeled data
  - Impressive achievements correspond to supervised learning
  - Expensive!
  - Sometimes experts & special equipment are needed

Limitations of Deep Learning

- Datasets may be biased
  - Deep Networks become biased against rare patterns
  - Serious consequences in some real-world applications (e.g., medical, automotive, … etc.)
  - Researchers should consider synthetic generation of data to mitigate the unbalanced representation of data

Limitations of Deep Learning

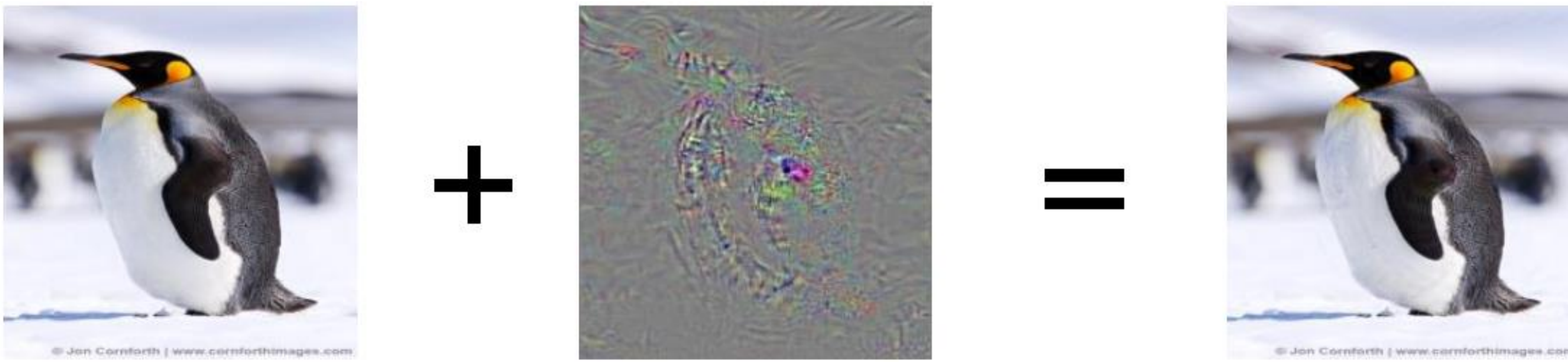
- Datasets may be biased

  - Classification may be sensitive to viewpoint
    - if one of the viewpoints is under-represented

Limitations of Deep Learning

- Sensitive to standard adversarial attacks
  - Datasets are finite and just represent a fraction of all possible images

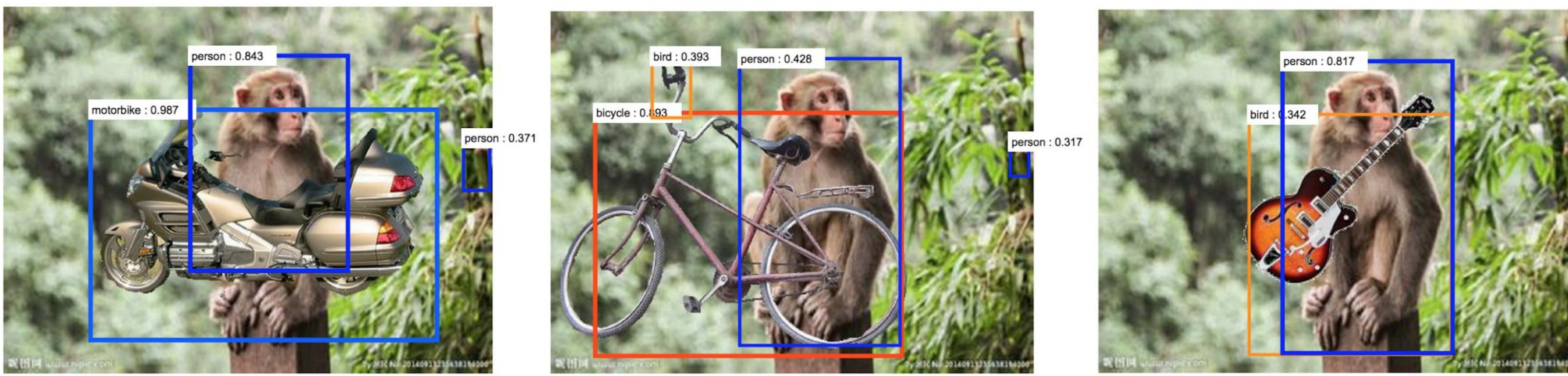


king penguin adversarial perturbation chihuahua

  - Add extra training, i.e., “adversarial training”

Limitations of Deep Learning

- Over-sensitive to changes in context
  - Limited number of contexts in dataset, i.e., monkey in jungle
  - Combinatorial Explosion!



Limitations of Deep Learning

- Combinatorial Explosion
  - Real world images are combinatorial large
  - Application dependent (e.g., medical imaging is an exception)
  - Considering compositionality may be a potential solution
  - Testing is challenging (consider worst case scenarios)

Limitations of Deep Learning

- Visual understanding is tricky
  - Mirrors
  - Sparse Information
  - Physics
  - Humor

- Unintended results from fitness functions

Machine Learning Model Selection

1. Categorize the problem:

   - Input: supervised, unsupervised, … etc.

   - Output: numerical → regression, class → classification, set of input groups → clustering

Machine Learning Model Selection

2. Understand your data:
   a) Analyze the data:
      - Descriptive statistics
      - Data visualization

   b) Process the data:
      - Pre-processing, cleansing, … etc.

   c) Feature Engineering

Machine Learning Model Selection

3. Determine the possible algorithms:
   - Based on categorization & data understanding
   - May have a look at the literature
   - Determine: desired accuracy, interpretability, scalability, complexity, training & testing time, runtime, … etc.

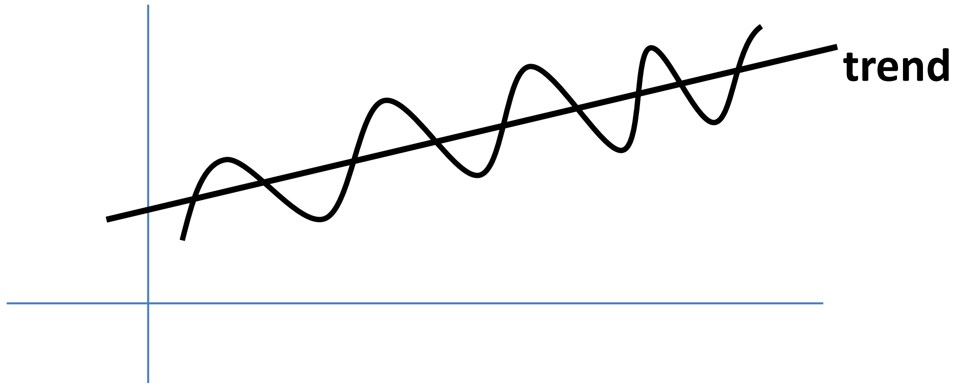
Machine Learning Model Selection

4. Implement Machine Learning Algorithms:
   - Setup a pipeline
   - Compare algorithms
   - Select an evaluation criteria

5. Tune hyperparameters

Time Series Prediction

- Time series contains:
  - Trend
  - Seasonality



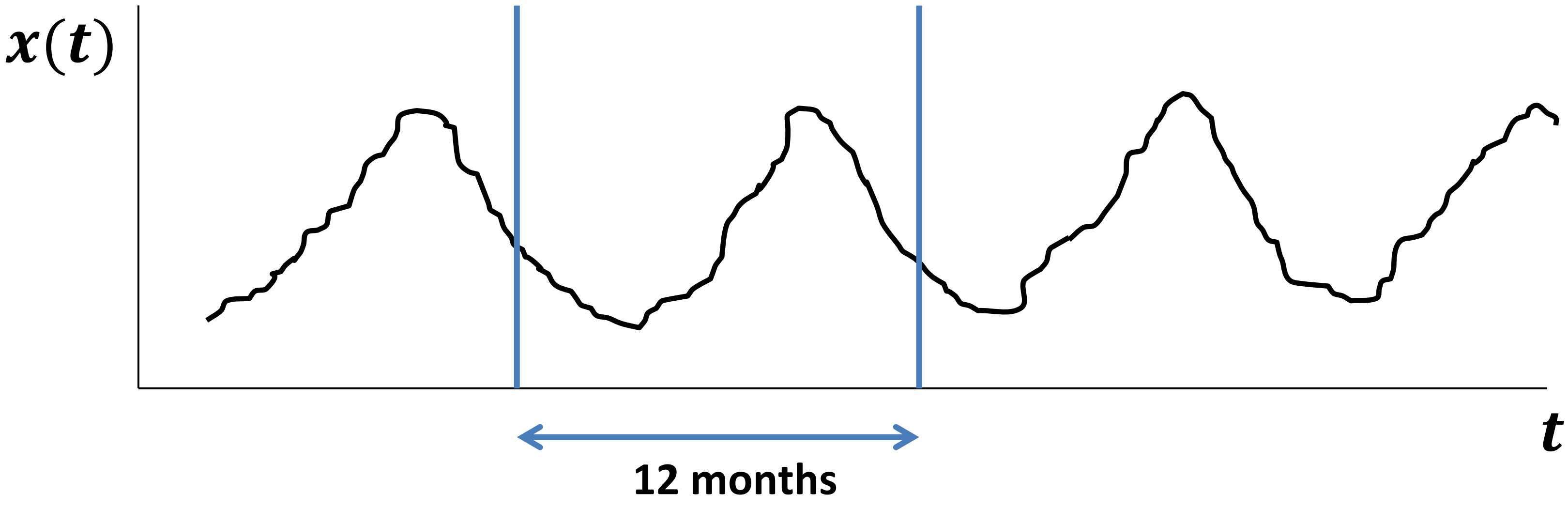
- De-seasonalization:
  - Remove the seasonal periodicities

TS → deseasonalize → predict → return back seasonal components

How to deseasonalize?

- Removing the seasonal periodicities

- Usually seasonal cycle length is 12 months



How to deseasonalize?

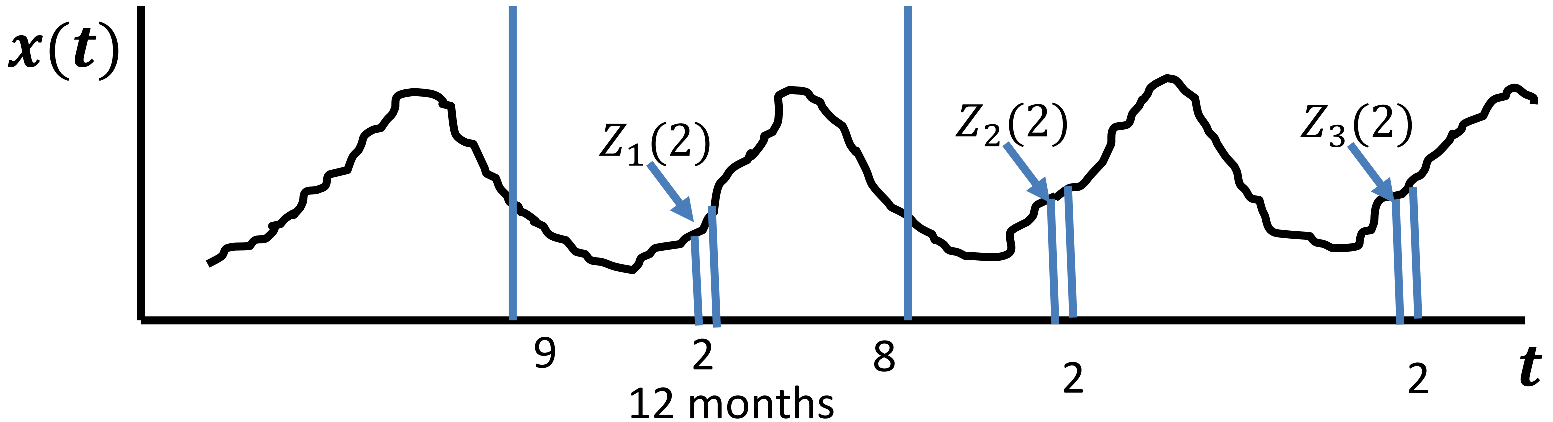
- Obtain average of TS values over this window

$$a(year) = \frac{1}{12} \sum_{window} x(t)$$

- Normalization step: $\boldsymbol{Z(i)} = \frac{\boldsymbol{x(i)}}{\boldsymbol{a(year)}}$

- Seasonal average ≡ avg of $\boldsymbol{Z(i)}$'s of the different years for month i

$$\boldsymbol{u(i)} = \frac{\sum_j \boldsymbol{Z_j(i)}}{\boldsymbol{\#years}}$$



How to deseasonalize?

- Seasonal average ≡ avg of $Z(i)$'s of the different years for month i

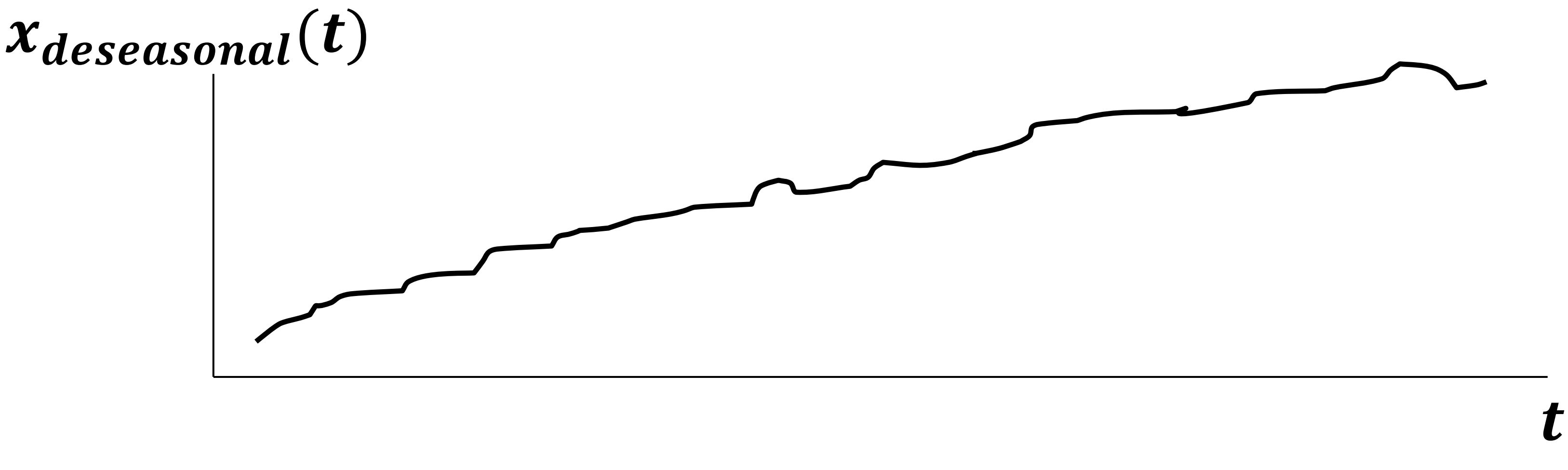
$$u(i) = \frac{\sum_{j=1}^{\#years} Z_j(i)}{\#years}$$

Deseasonalization Step

- Divide time series value by the corresponding seasonal average

$$x_{deseasonal}(t) = \frac{x(t)}{u(month(t))}$$

- After that focus on predicting the trend



Recover Seasonality

- After trend prediction, seasonality can be recovered via multiplication by the corresponding seasonal average

Acknowledgment

- These slides have been created relying on lecture notes of Prof. Dr. Amir Atiya