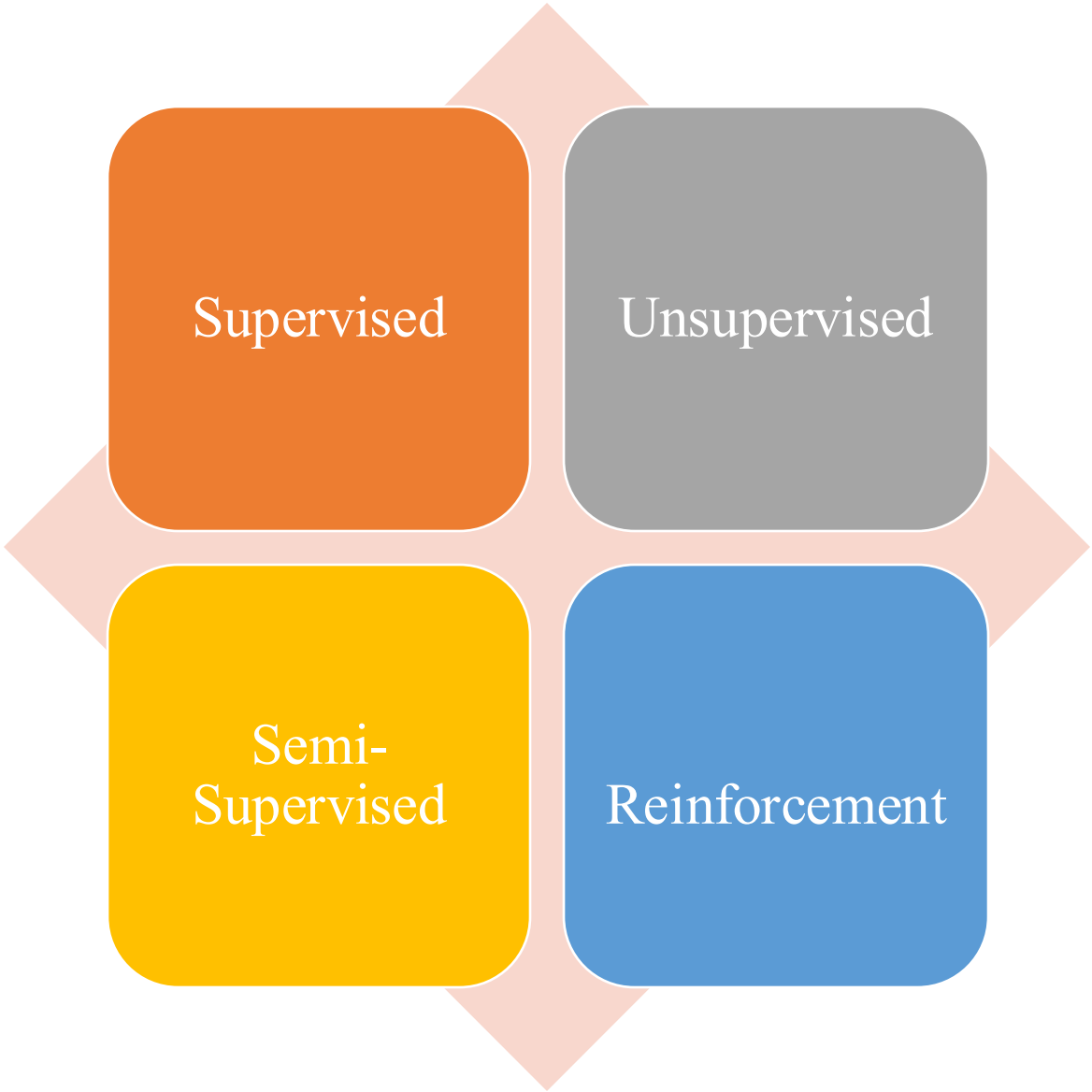
Deep Neural Networks

Hellenic Mediterranean University
Department of Electrical and Computer Engineering
Dr. Konstantinos Karampidis

Types of Learning



Supervised

Supervised machine learning can be classified into two types:

- Classification
- Regression

Supervised – Classification

- Predicts the label of a class
- Predict the dataset's categories

Example: "Yes" or "No"

Commonly Used Algorithms:

- Decision Tree Algorithm
- Logistic Regression
- Random Forest Algorithm
- Support Vector Machine Algorithm

Supervised – Regression

➢Predicts the numerical label/continuous variables

Example: weather prediction

Commonly used algorithms:

- Decision Tree Algorithm
- Lasso Regression
- Multivariate Regression Algorithm
- Simple Linear Regression Algorithm

Unsupervised

Unsupervised machine learning can be classified into two types:

- Association
- Clustering

Unsupervised – Association

Finds relations between variables in a large dataset

Goal: discover and map data dependent on the other to produce maximum profit

Example: web usage mining

Commonly used algorithms:

- Apriori algorithm
- Eclat
- FP-growth algorithm

Unsupervised – Clustering

A method of grouping each set of similar objects into a cluster

Goal: discover inherent groups from the dataset

Example: Retail marketing
Commonly used algorithms:

- K-Means Clustering Algorithm
- DBSCAN Algorithm
- Independent Component Analysis (ICA)
- Mean-Shift Algorithm
- Principal Component Analysis (PCA)

Semi - Supervised

Semi-supervised learning uses a small amount of labeled data and a considerable portion of unlabeled instances so that the model can learn and make predictions on new data.

Reinforcement Learning

There are two main categories of reinforcement learning

- Positive

Positive reinforcement learning is an event that occurs as a result of a particular behaviour. This type of reinforcement learning strengthens the behaviour and increases its frequency, positively affecting the actions taken by the agent.

- Negative

Negative reinforcement learning decreases the frequency of the occurrence of a behaviour.

Reinforcement Learning Applications

- Robotics
- Self-driving Automobiles
- Video Games

Limitations of Neural Networks

Random initialization + **densely connected networks** lead to:

- High cost
  - Each neuron in the neural network can be considered as a logistic regression.
  - Training the entire neural network is to train all the interconnected logistic regressions.
- Difficult to train as the number of hidden layers increases
  - Recall that logistic regression is trained by gradient descent.
  - In backpropagation, gradient is progressively getting more dilute. That is, below top layers, the correction signal δ_n is minimal.
- Stuck in local optima
  - The objective function of the neural network is usually not convex.
  - The random initialization does not guarantee starting from the proximity of global optima.
- Solution:
  - Deep Learning/Learning multiple levels of representation

Don’t Forget !!!

- Logistic Regression (LR) is classified under Machine Learning (ML). One should not use Deep Learning (DL) for every possible problem.
- Don’t use Machine Learning (ML) or Deep Learning (DL) if you can write a simple program to solve the problem: Using ML or DL for such a problem will be inaccurate or very inefficient
- Always start with simple models before moving to complex once: Complex models have too many free parameters. It is very difficult to know how they will behave in unexplored cases.
- Don’t use DL if the problem is simple or if you get sufficient accuracy with a ML model: DL models are complex. Using DL for a simple problem is an overkill, the model will be inefficient
- Don’t use DL if you have small amount of data: You can’t train a DL model with small amount of data. The results will be really bad. Best situation is a lot of data with few very distinct parameters. This is not a magical method that solves all problems.

Representation Learning

The performance of simple machine learning algorithms depends heavily on the representation of the data they are given. For example, when logistic regression is used to recommend cesarean delivery, the AI system does not examine the patient directly. Instead, the doctor tells the system several pieces of relevant information, such as the presence or absence of a uterine scar. Each piece of information included in the representation of the patient is known as a feature.

Logistic regression learns how each of these features of the patient correlates with various outcomes. However, it cannot influence how features are defined in any way. If logistic regression were given an MRI scan of the patient, rather than the doctor's formalized report, it would not be able to make useful predictions. Individual pixels in an MRI scan have negligible correlation with any complications that might occur during delivery.

Representation Learning – Example

Suppose that we would like to write a program to detect cars in photographs. We know that cars have wheels, so we might like to use the presence of a wheel as a feature. Unfortunately, it is difficult to describe exactly what a wheel looks like in terms of pixel values. A wheel has a simple geometric shape, but its image may be complicated by shadows falling on the wheel, the sun glaring off the metal parts of the wheel, the fender of the car or an object in the foreground obscuring part of the wheel, and so on.

One solution to this problem is to use machine learning to discover not the mapping from representation to output but also the representation itself.

This approach is known as representation learning . Learned representations often result in much better performance than can be obtained with hand-designed representations.

Representation Learning – Example

They also enable AI systems to rapidly adapt to new tasks, with minimal human intervention. A representation learning algorithm can discover a good set of features for a simple task in minutes, or for a complex task in hours to months. Manually designing features for a complex task requires a great deal of human time and effort; it can take decades for an entire community of researchers.

Representation Learning

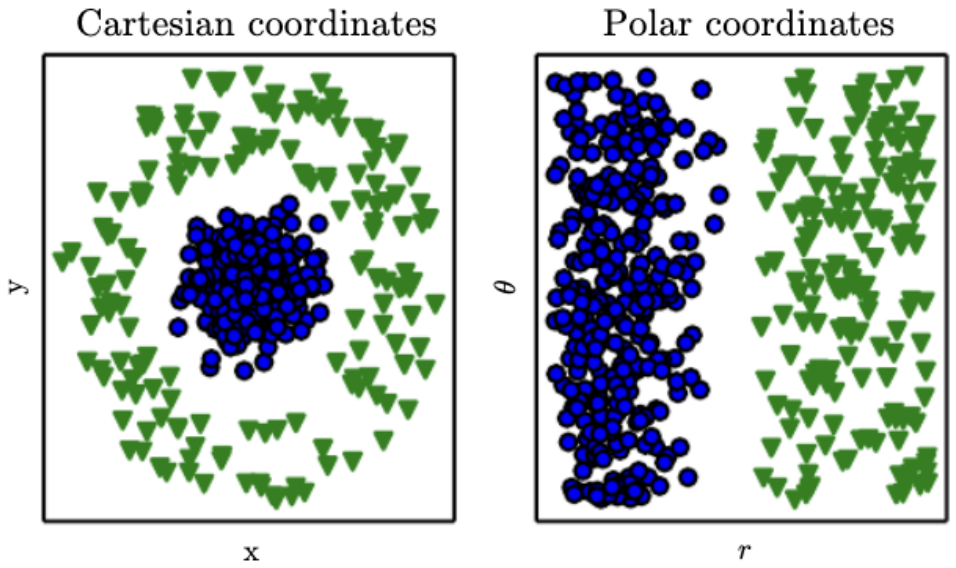


Figure 1.1: Example of different representations: suppose we want to separate two categories of data by drawing a line between them in a scatterplot. In the plot on the left, we represent some data using Cartesian coordinates, and the task is impossible. In the plot on the right, we represent the data with polar coordinates and the task becomes simple to solve with a vertical line. (Figure produced in collaboration with David Warde-Farley.)

https://www.deeplearningbook.org/

Definition

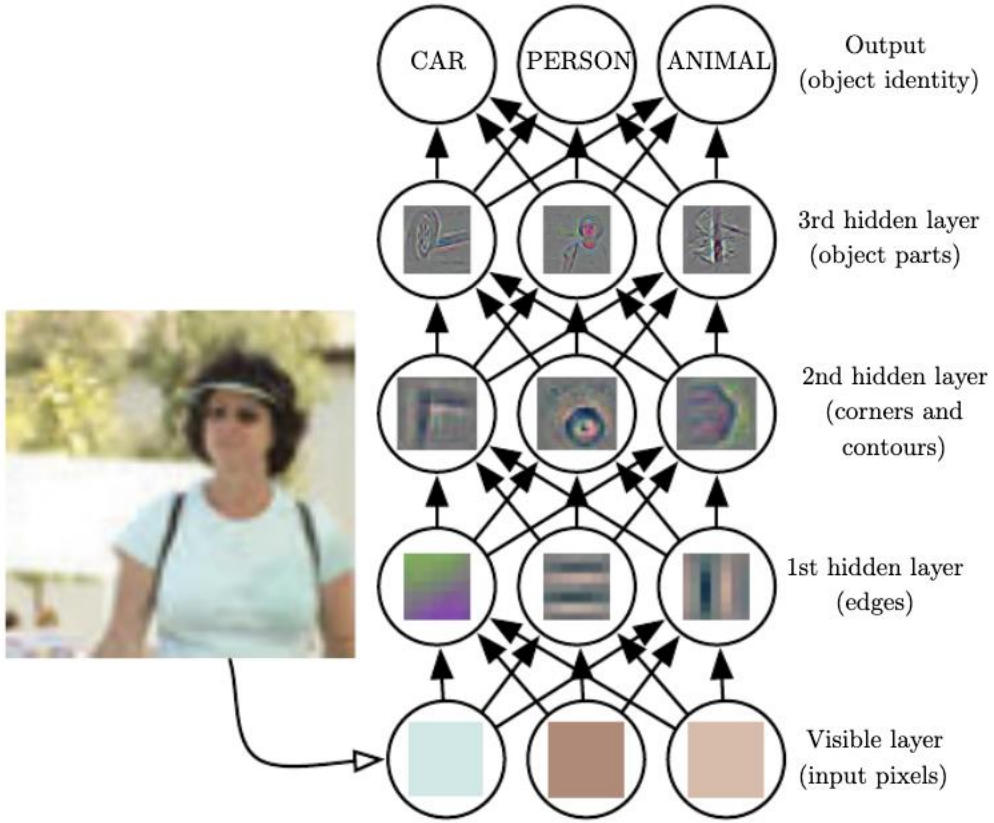
Deep Learning is a new area of Machine Learning research, which has been introduced with the objective of moving Machine Learning closer to one of its original goals

Deep Learning is about learning multiple levels of representation and abstraction that help to make sense of data such as images, sound, and text.

What is Deep Learning

Composition of non-linear transformation of the data.

Goal: Learn useful representations – features- directly from data



https://www.deeplearningbook.org/

Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels.

Given the first hidden layer's description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges.

Given the second hidden layer's description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners.

Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image.

Motivations for Deep Architectures

Why Deep Learning?

- Insufficient depth can hurt
  - With shallow architecture (SVM, NB, KNN, etc.), the required number of nodes in the graph (i.e. computations, and also number of parameters, when we try to learn the function) may grow very large.
  - Many functions that can be represented efficiently with a deep architecture cannot be represented efficiently with a shallow one.

Motivations for Deep Architectures

Why Deep Learning?

- The brain has a deep architecture
  - The visual cortex shows a sequence of areas each of which contains a representation of the input, and signals flow from one to the next.
  - Note that representations in the brain are in between dense distributed and purely local: they are **sparse**: about 1% of neurons are active simultaneously in the brain.

Motivations for Deep Architectures

Why Deep Learning?

- Cognitive processes seem deep
  - Humans organize their ideas and concepts hierarchically.
  - Humans first learn simpler concepts and then compose them to represent more abstract ones.
  - Engineers break-up solutions into multiple levels of abstraction and processing

Why deep learning?

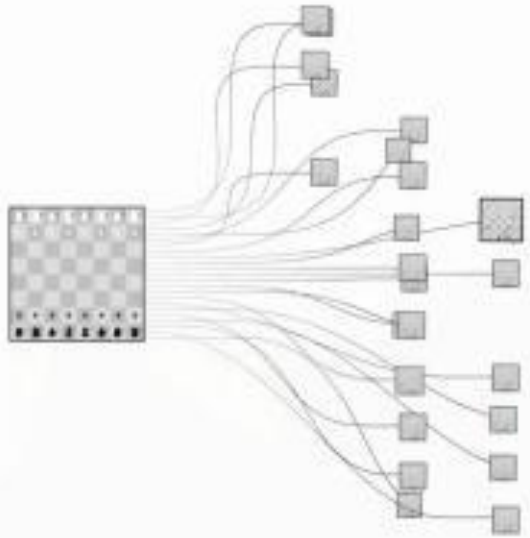
*1. **Universal Learning Approach**: Because DL has the ability to perform in approximately all application domains, it is sometimes referred to as universal learning.*

*2. **Robustness**: In general, precisely designed features are not required in DL techniques. Instead, the optimized features are learned in an automated fashion related to the task under consideration. Thus, robustness to the usual changes of the input data is attained.*

*3. **Generalization**: Different data types or different applications can use the same DL technique, an approach frequently referred to as transfer learning (TL). Furthermore, it is a useful approach in problems where data is insufficient.*

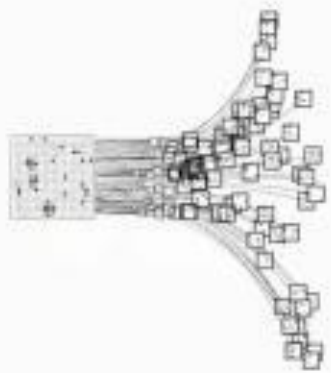
*4. **Scalability**: DL is highly scalable. ResNet , which was invented by Microsoft, comprises 1202 layers and is frequently applied at a supercomputing scale. Lawrence Livermore National Laboratory (LLNL), a large enterprise working on evolving frameworks for networks, adopted a similar approach, where thousands of nodes can be implemented.*

Source: Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420 (2021). https://doi.org/10.1007/s42979-021-00815-1



Chess: 10^{47}

Deep Blue, Feb 10, 1996



Go: 10^{170}

AlphaGo, March, 2016

Deep Blue

In 1996 it made history by defeating Russian grandmaster Garry Kasparov in one of their six games—the first time a computer had won a game against a world champion under tournament conditions.

In the 1997 rematch, it won the deciding sixth game in only 19 moves; its 3.5–2.5 victory (it won two games and had three draws) marked the first time a current world champion had lost a match to a computer under tournament conditions.

In its final configuration, the IBM RS6000/SP computer used 256 processors working in tandem, with an ability to evaluate 200 million chess positions per second.

Source: https://www.britannica.com/topic/Deep-Blue

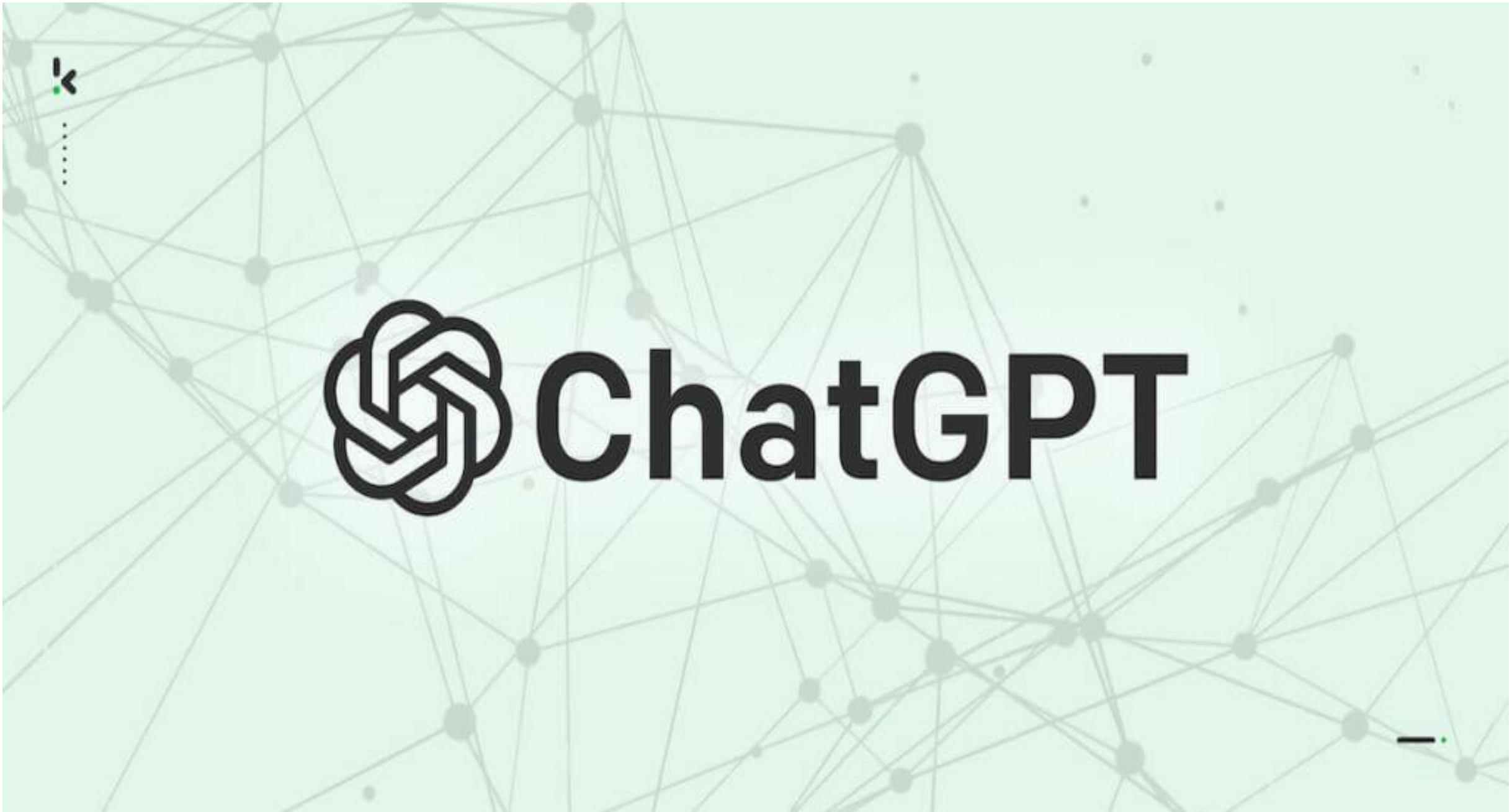


Image source: https://www.klippa.com/en/blog/information/what-is-chatgpt/

Deepfake

Deepfake AI is a type of artificial intelligence used to create convincing image, audio and video hoaxes

Image source: https://www.nytimes.com/interactive/2020/11/21/science/artificial-intelligence-fake-people-faces.html

When to apply deep learning

- Cases where human experts are not available.
- Cases where humans are unable to explain decisions made using their expertise (language understanding, medical decisions, and speech recognition).
- Cases where the problem solution updates over time (price prediction, stock preference, weather prediction, and tracking).
- Cases where solutions require adaptation based on specific cases (personalization, biometrics).
- Cases where size of the problem is extremely large and exceeds our inadequate reasoning abilities (sentiment analysis, matching ads to Facebook, calculation webpage ranks).

Computer Vision

Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images.

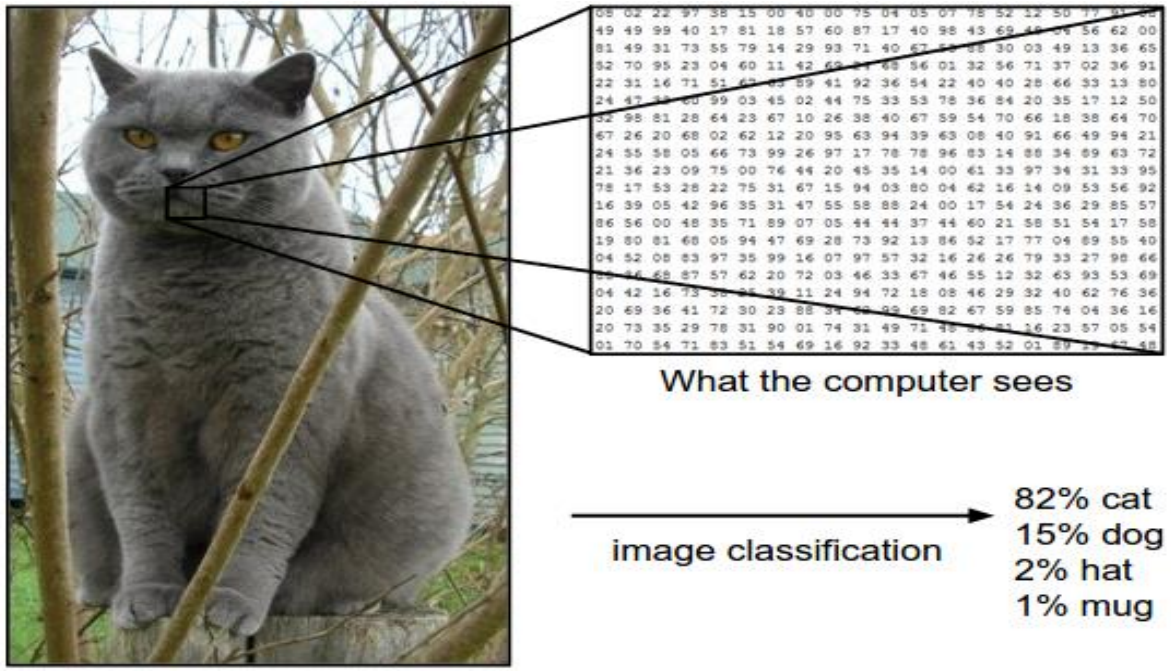


Image source: https://cs231n.github.io/classification/

Convolutional Neural Networks perform exceptionally well for computer vision tasks

Autonomous vehicles

AI-powered robotaxis are expected to create a $2 trillion market worldwide by 2030, according to global financial services firm UBS.

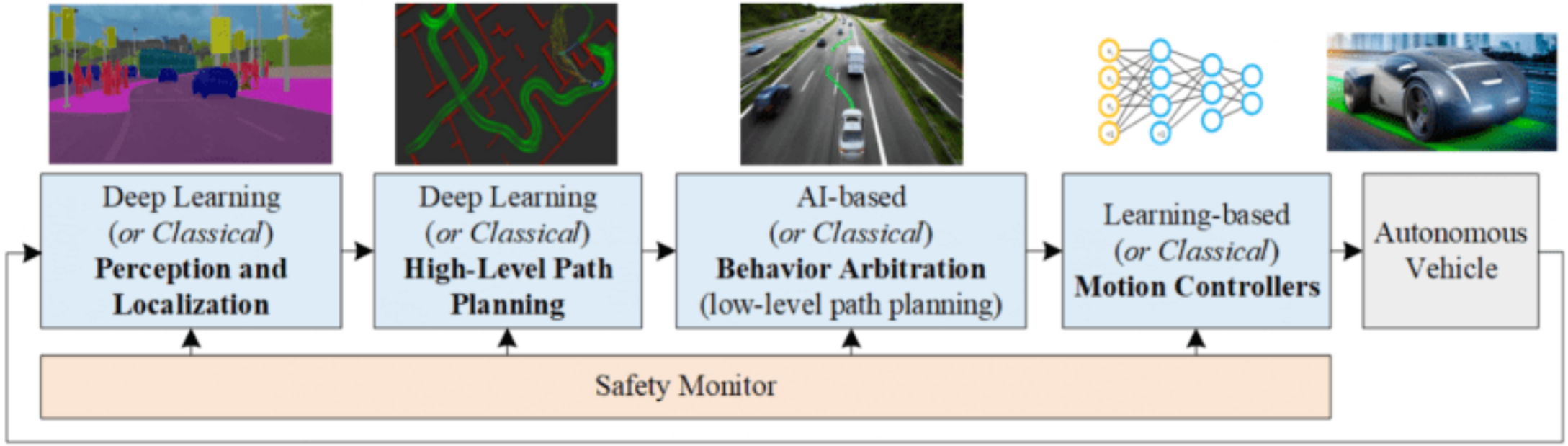
Self-driving cars use camera-based machine vision systems and radar- and lidar-based detection units to perceive, understand, and safely navigate their surrounding environment.
The goal is to optimize performance of self-driving car perception applications such as lane navigation and pedestrian detection.

A semantic segmentation neural network has to be build and trained, to identify objects such as roads, pedestrians, and other vehicles.

There are many platforms to deploy the implemented neural network. For example the NVIDIA DRIVE AGX™.

AI applications for autonomous vehicles concern a variety of autonomous driving scenarios such as highway driving, city roads, and parking.

Autonomous vehicles



Source: Grigorescu, S., Trasnea, B., Cocias, T., & Macesanu, G. (2020). A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, *37*(3), 362-386.

Autonomous vehicles

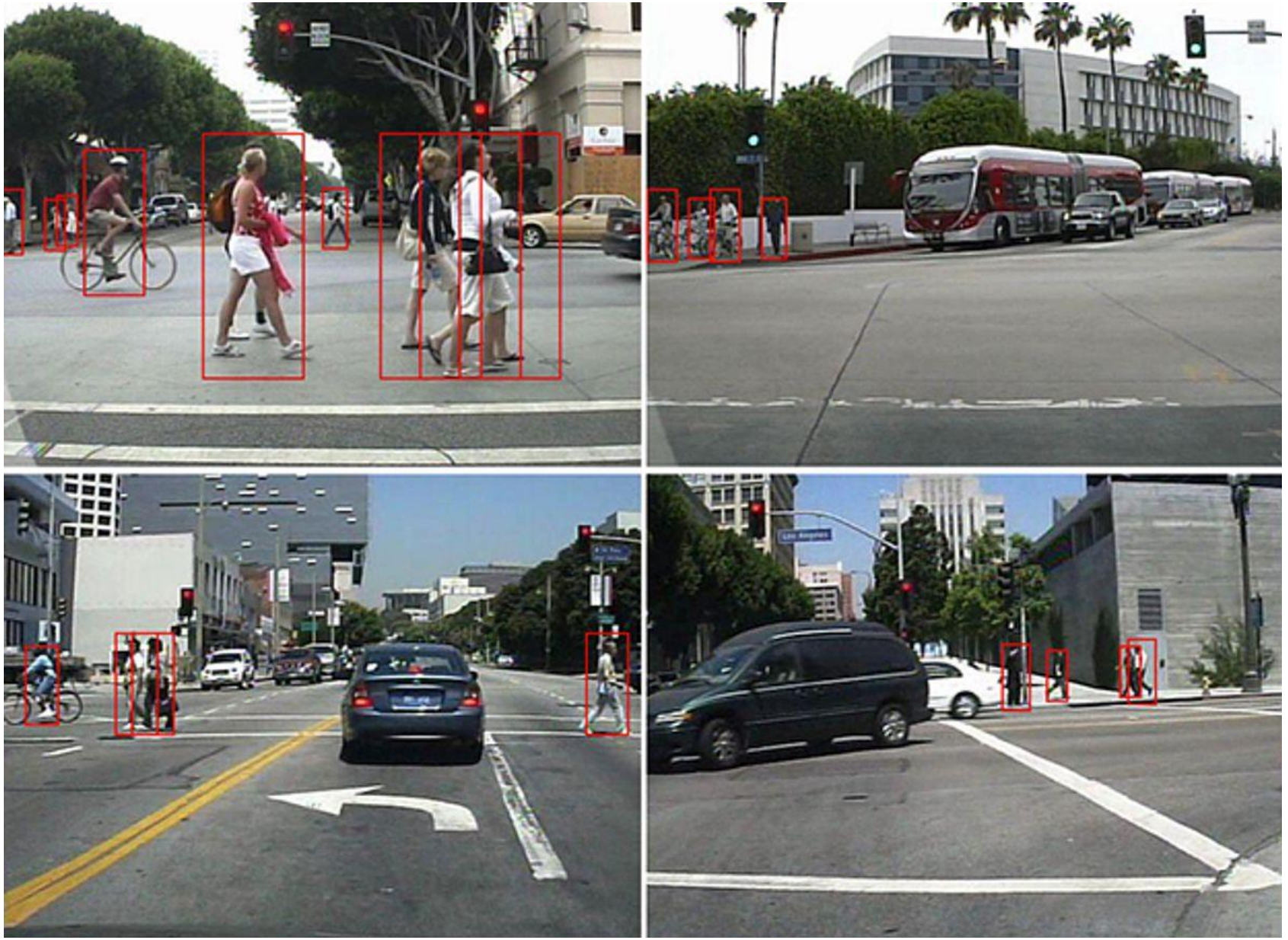


Image source: https://spectrum.ieee.org/deep-learning-makes-driverless-cars-better-at-spotting-pedestrians

Autonomous vehicles

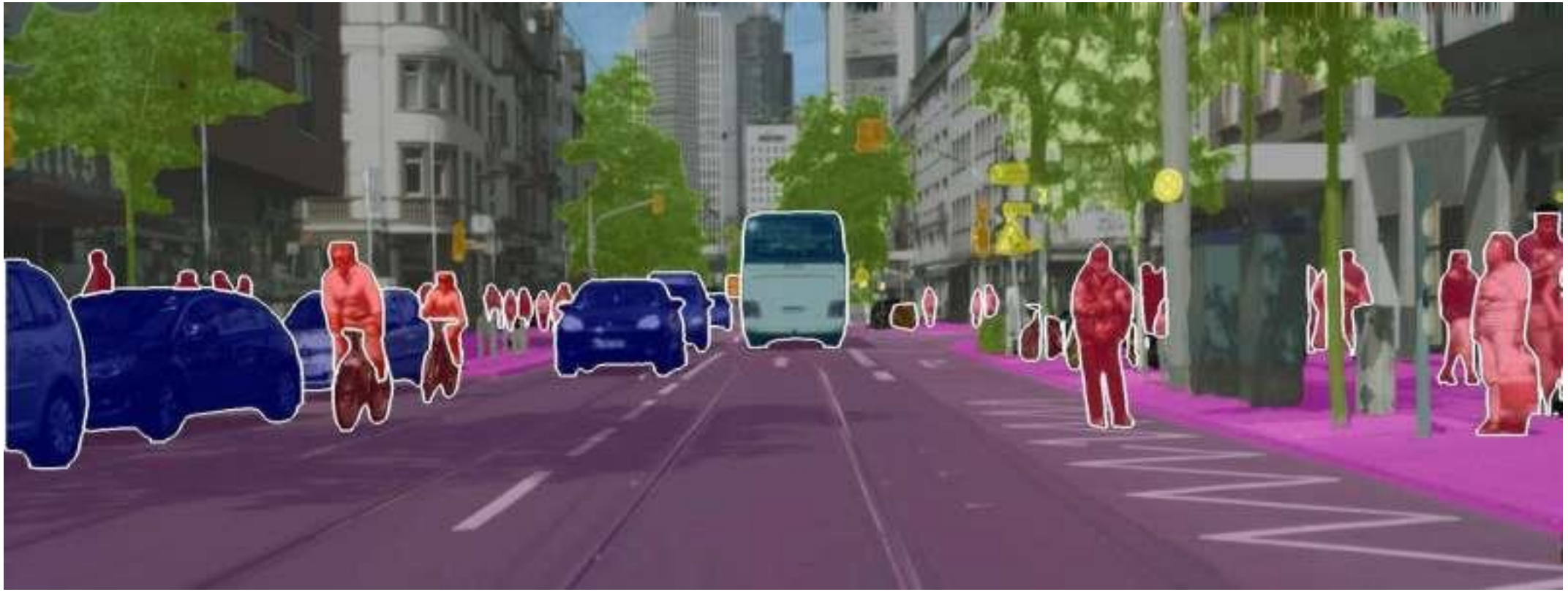
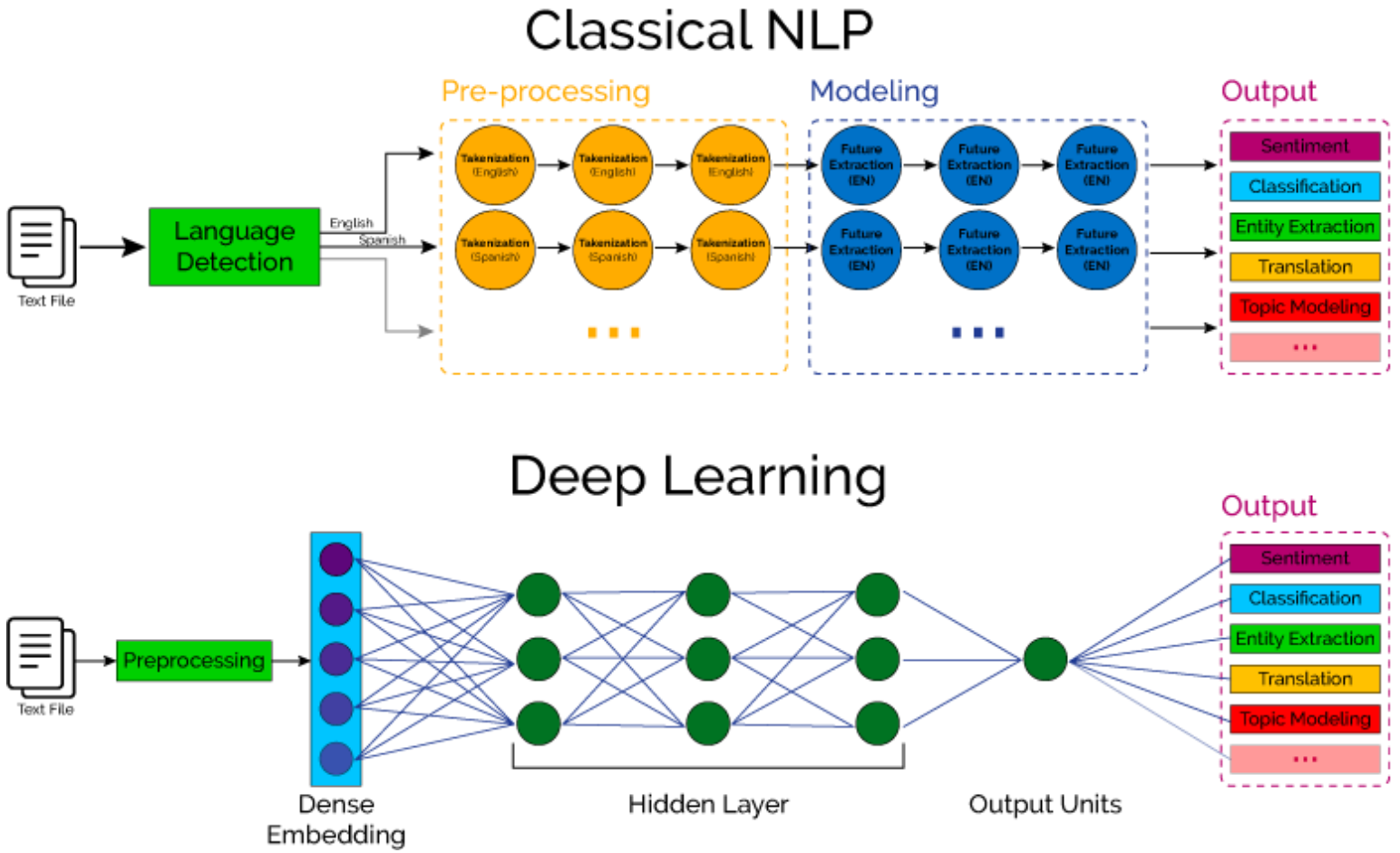


Image source: https://techxplore.com/news/2020-05-deep-image-recognition-ability-self-driving.html

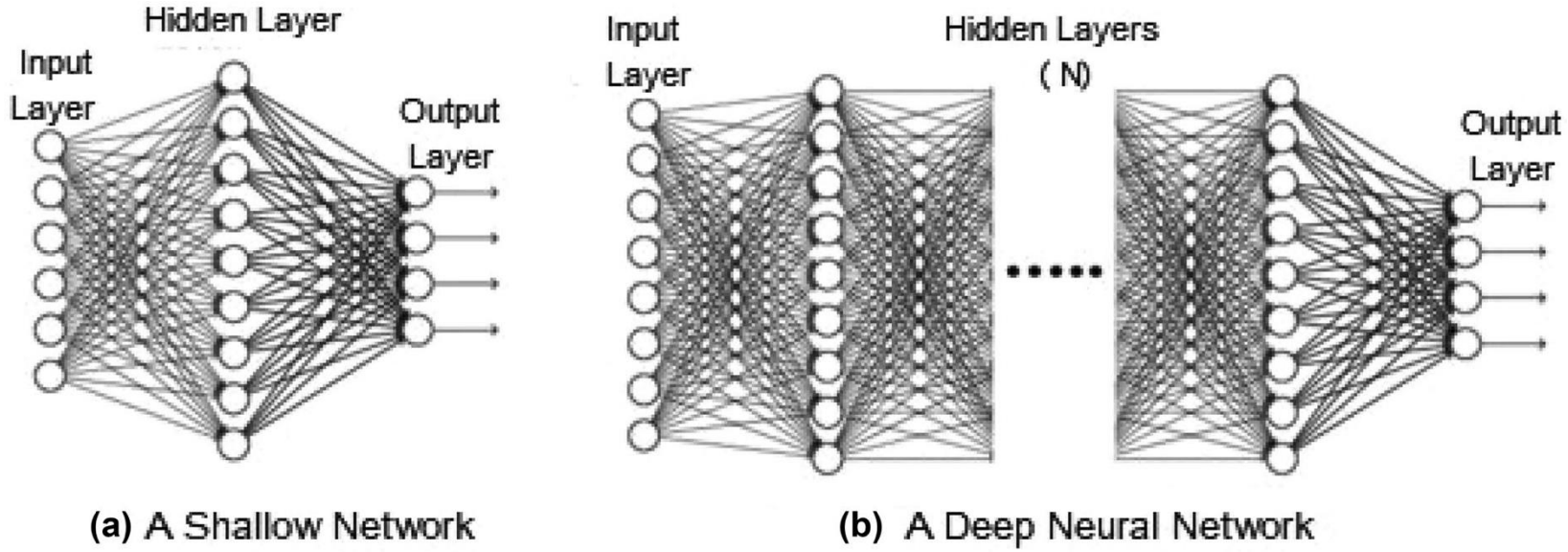
NLP

Natural language processing (NLP) is the use of human languages, such as English or French, by a computer.
It is the ability of a computer program to understand human language as it is spoken and written -- referred to as natural language.



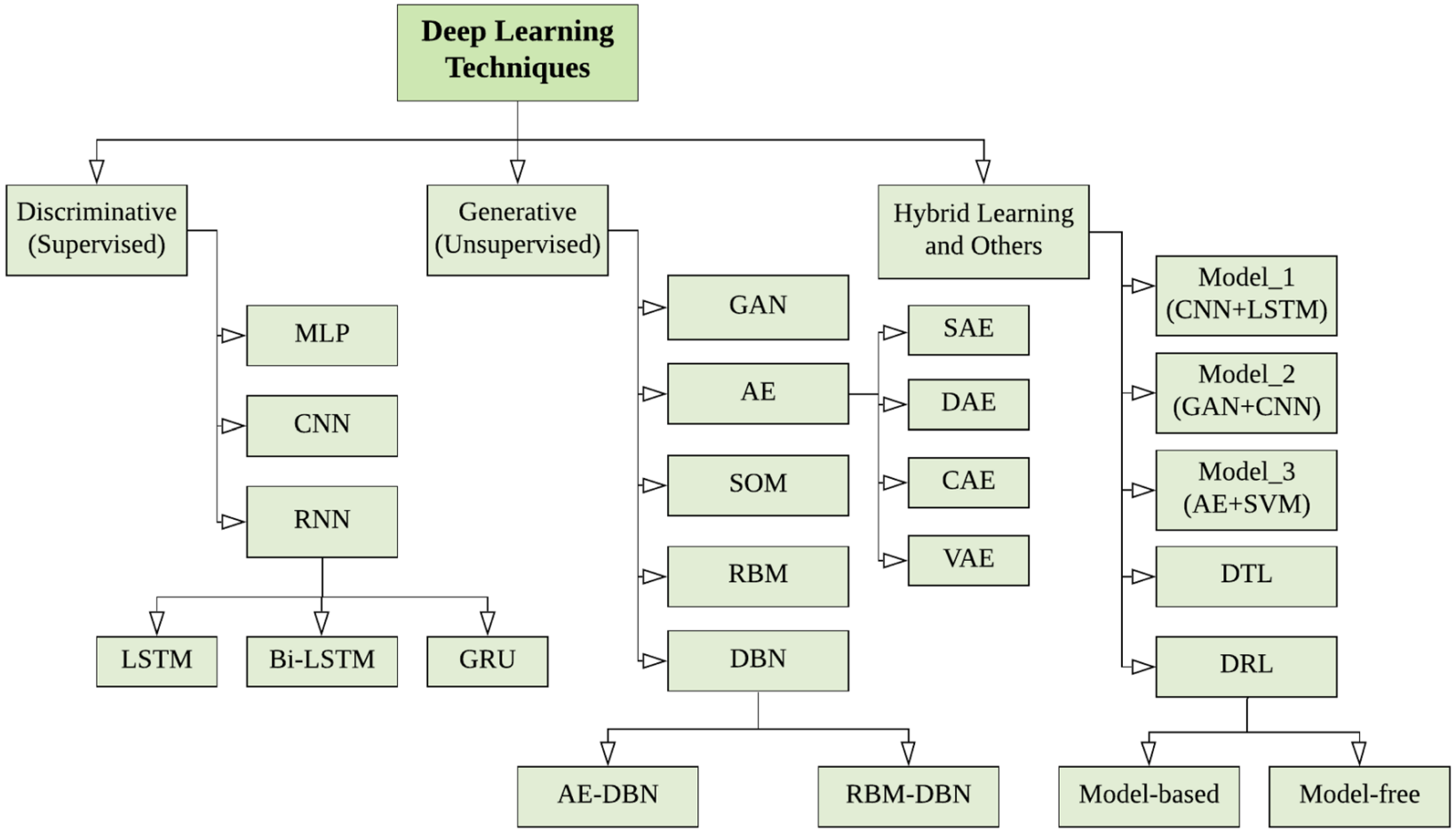
Source: https://www.nsuchaud.fr/2018/03/difference-between-classical-nlp-deep-learning-nlp/

Shallow vs Deep network



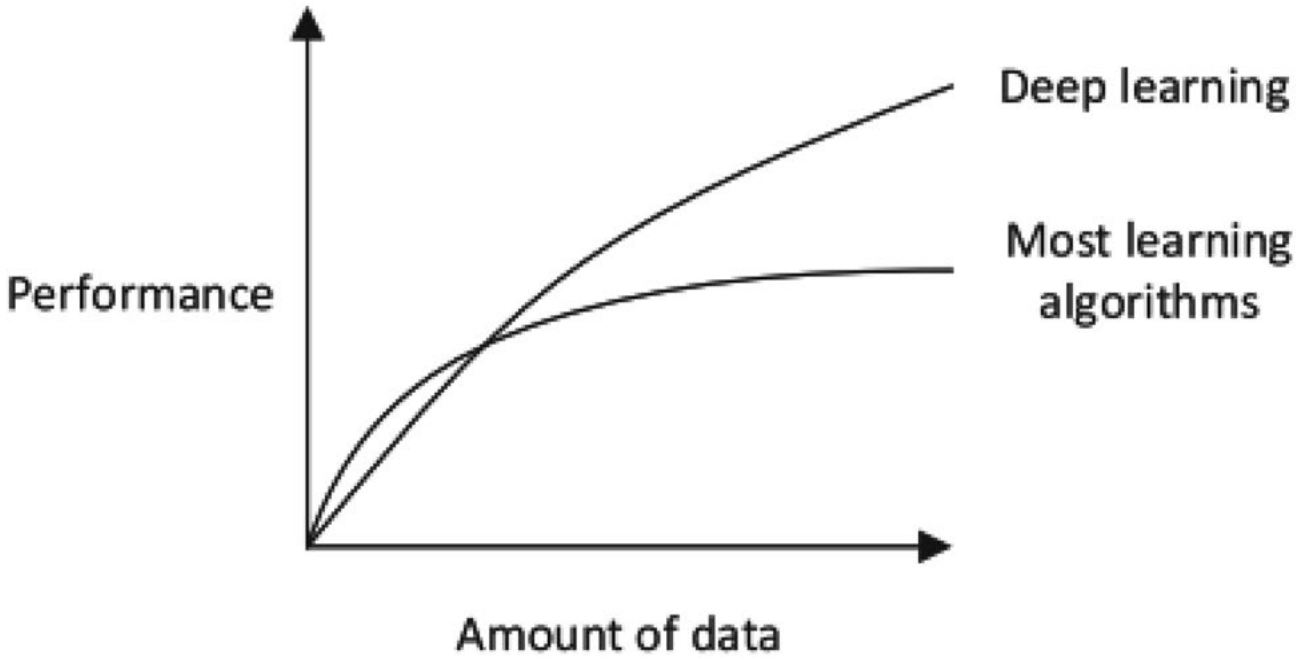
Source: Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420 (2021). https://doi.org/10.1007/s42979-021-00815-1

Taxonomy of deep learning techniques



Source: Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420 (2021). https://doi.org/10.1007/s42979-021-00815-1

Performance comparison between deep learning and other machine learning algorithms



Source: Sarker, I.H. Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. SN COMPUT. SCI. 2, 420 (2021). https://doi.org/10.1007/s42979-021-00815-1

Problems with data

- Insufficient data
- Too much data
- Non- Representative data
- Missing data
- Duplicate data
- Outliers

Models trained with insufficient data perform poorly while making predictions. It may lead to 2 cases:

Over-fitting: Here the training model reads the data too much for too little data. this means the training model actually memorizes the patterns. It has low training errors and high test errors. Does not work well in the real world.

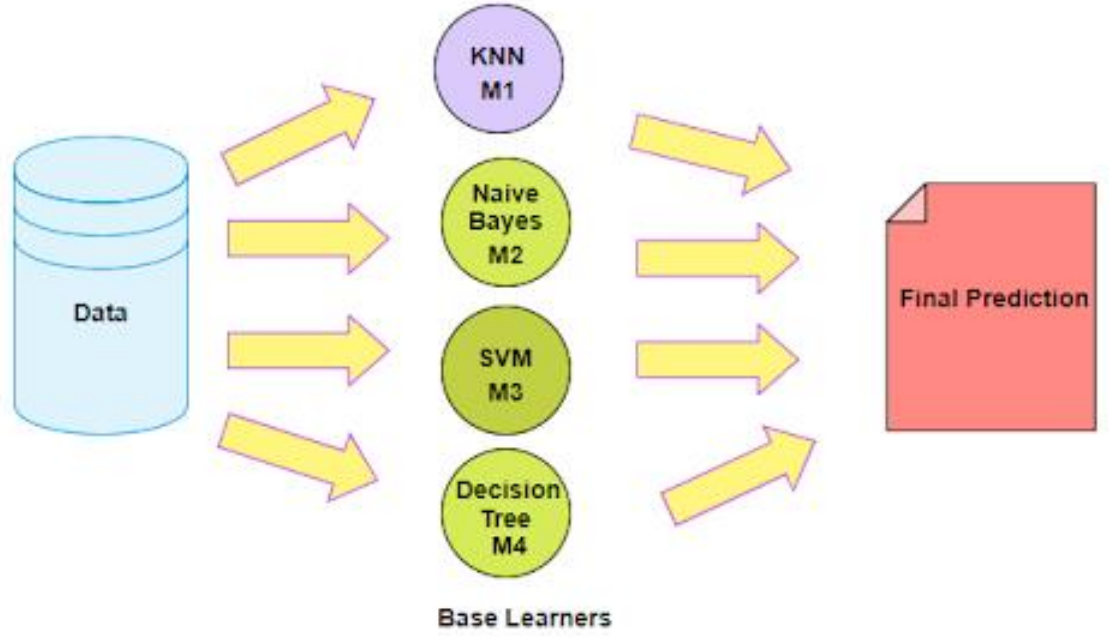
Under-fitting: It builds an overly simple model. This means the data are not trained to our expectations. The model is unable to capture relationship in data and has no predictive power.

Problems with data - Insufficient data

Model Complexity:

- Build a simple model with fewer parameters. This method is less susceptible to over-fitting.
  Example: Naive Bayes, Linear Regression.

- Use an Ensemble learning technique: It is defined as several learners are combined to obtain a better performance than any individual learners. It is often used to improve classification and prediction.



Source: Google Images – Ensemble Learning

Problems with data - Insufficient data

Transfer Learning:

Transfer Learning uses a pre-built model, which is then tweaked on the small dataset that you have.

It is also defined as the practice of **reusing a trained Neural Networks,** that solve a similar problem to yours, usually leaving the network architecture unchanged and reusing some of the model weights.

It is very much **useful when the new dataset is small** and not sufficient to train the model from scratch.

Problems with data - Insufficient data

Data Augmentation:

Data Augmentation takes the pre-existing samples and changes them in some way to create new samples and also increase the number of training samples and typically used with Image data.

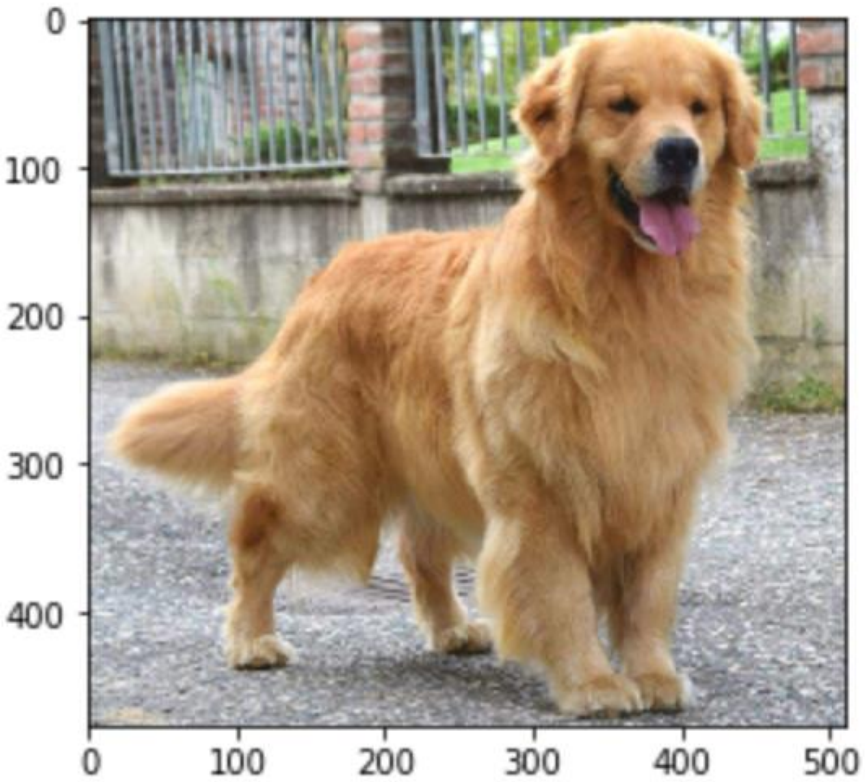
These image processing options are often used as pre-processing techniques to make image classification models built using CNN are robust.

Problems with data - Insufficient data

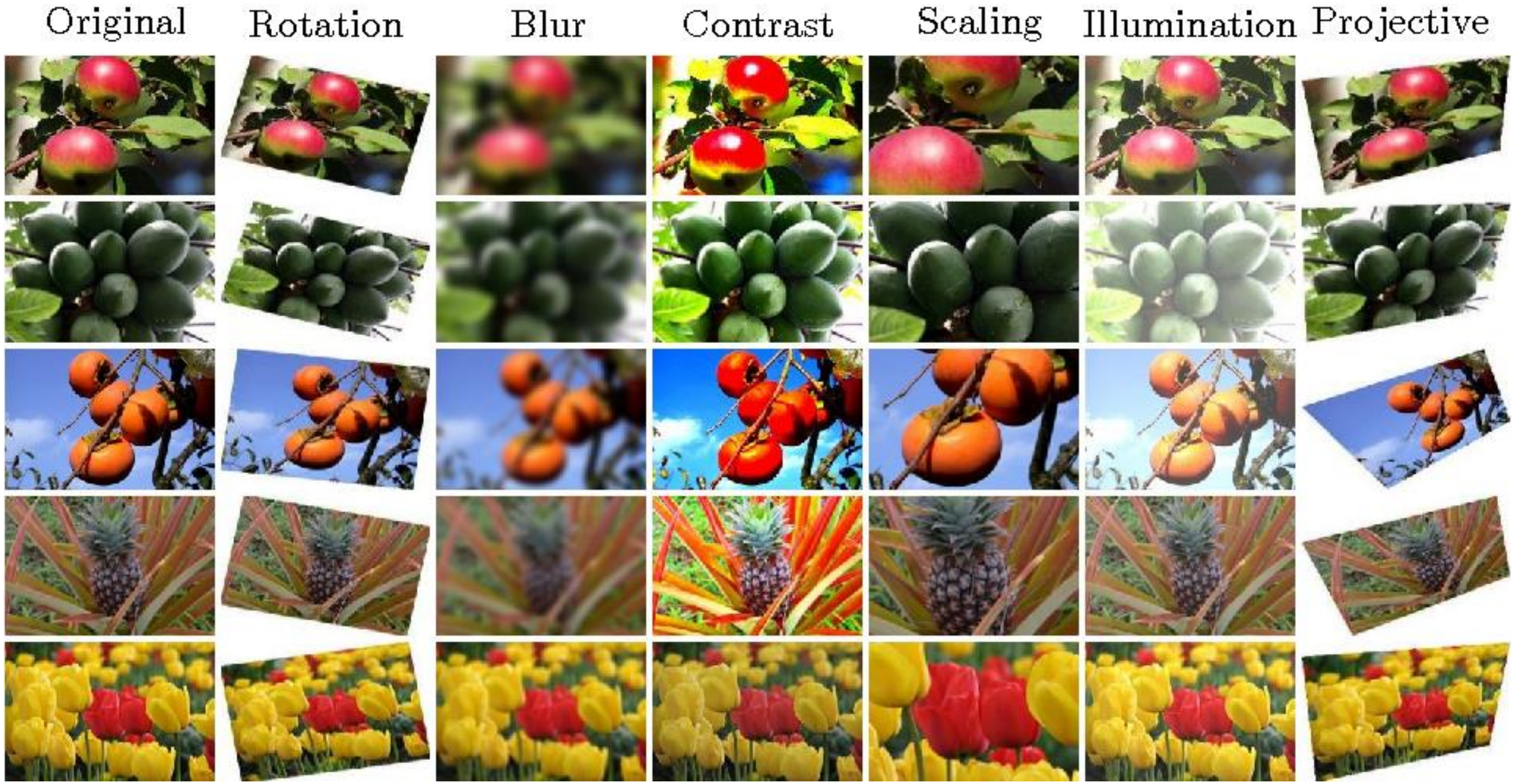
Classic image processing activities for data augmentation are:

- padding
- random rotating
- re-scaling,
- vertical and horizontal flipping
- translation (image is moved along X, Y direction)
- cropping
- zooming
- darkening & brightening/color modification
- grayscaling
- changing contrast
- adding noise
- random erasing
- affine Transforms

Problems with data - Insufficient data



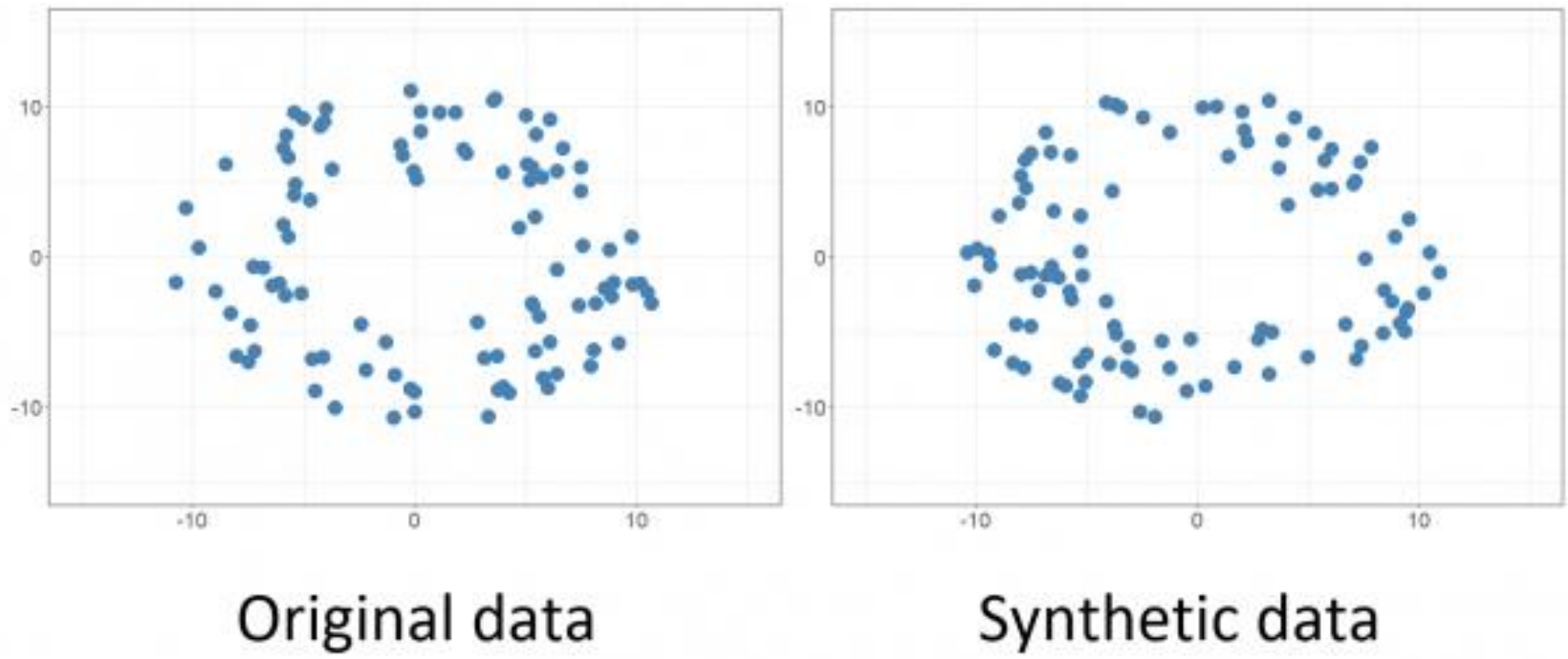
Problems with data - Insufficient data



Source: https://medium.com/analytics-vidhya/data-augmentation-is-it-really-necessary-b3cb12ab3c3f

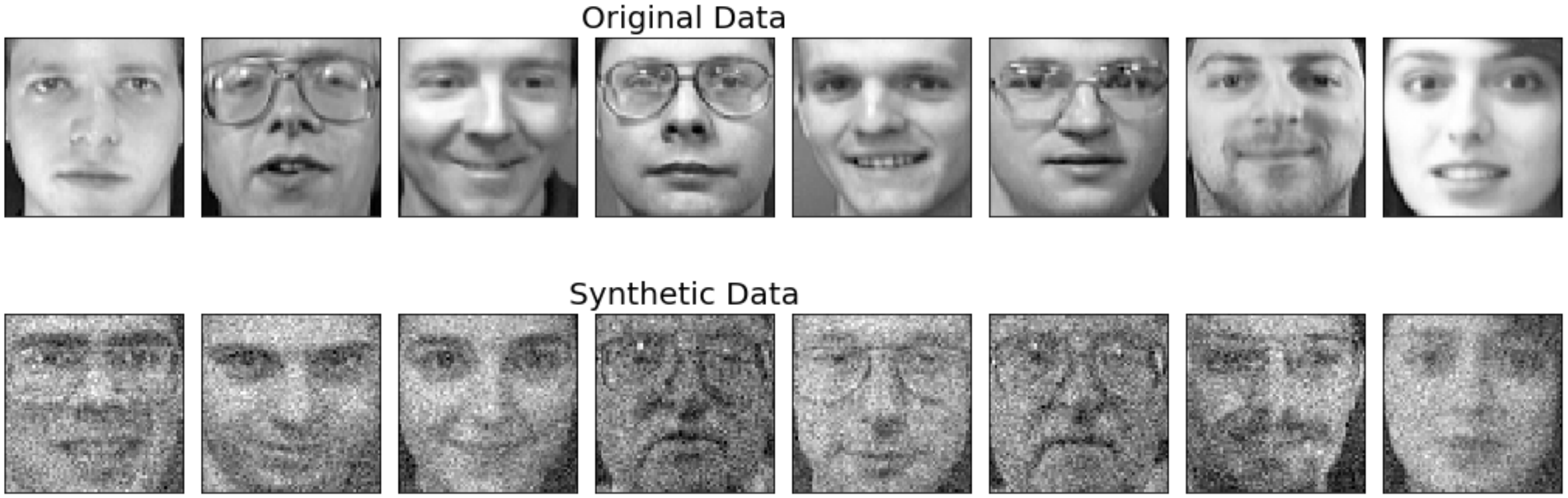
Problems with data - Insufficient data

Synthetic Data:
Synthetic data generally refers to artificially generating samples which mimic the real-world data (it is one only if we have a good understanding of features). This may induce bias in existing data.



Source: Google Images – Synthetic data

Problems with data - Insufficient data



Why Now ?

1952 Stochastic Gradient Descent

1958 Perceptron
- Learnable Weights

1986 Backpropagation
- Multi-Layer Perceptron

1995 Deep Convolutional NN
- Digit Recognition

Neural Networks date back decades, so why the resurgence?

1. Big Data
- Larger Datasets
- Easier Collection & Storage

IMAGENET

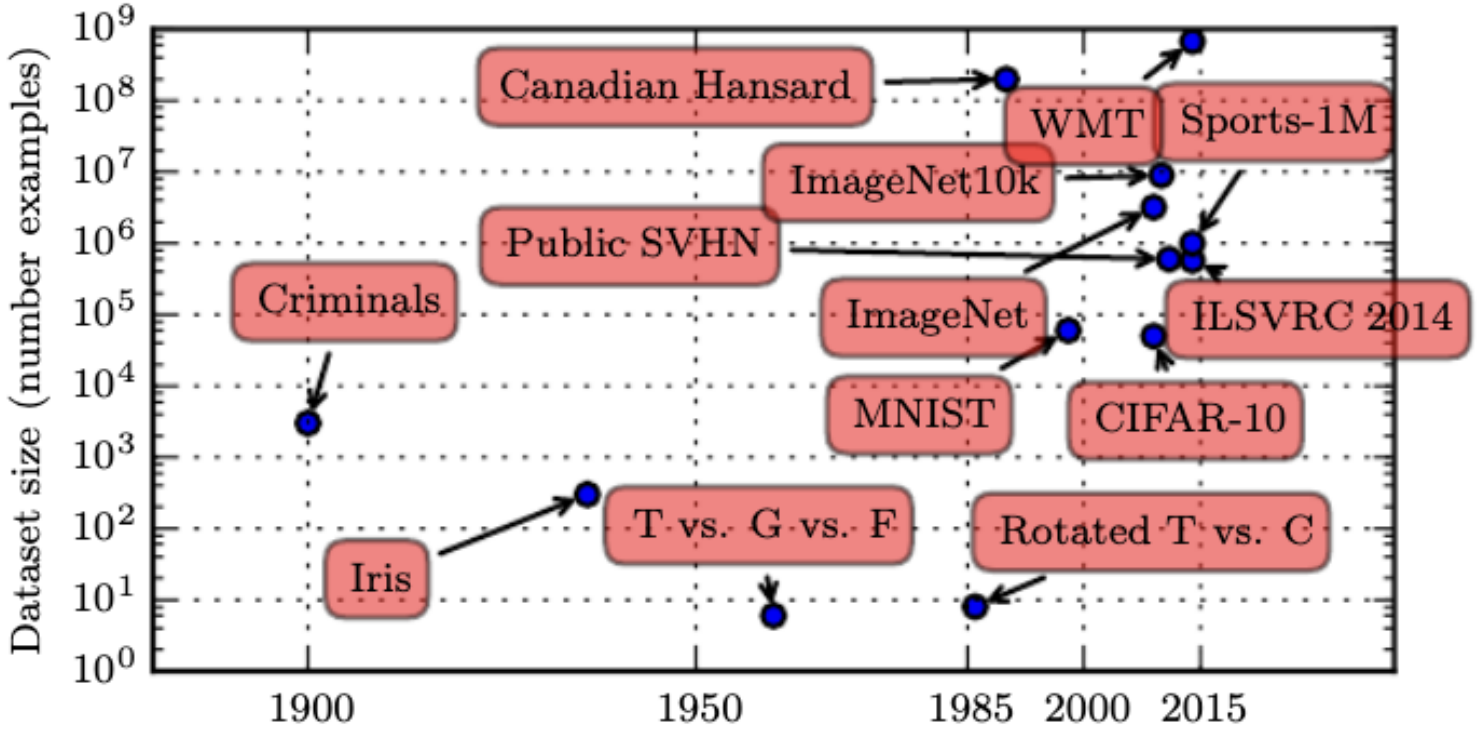
WIKIPEDIA
The Free Encyclopedia

2. Hardware
- Graphics Processing Units (GPUs)
- Massively Parallelizable

3. Software
- Improved Techniques
- New Models
- Toolboxes

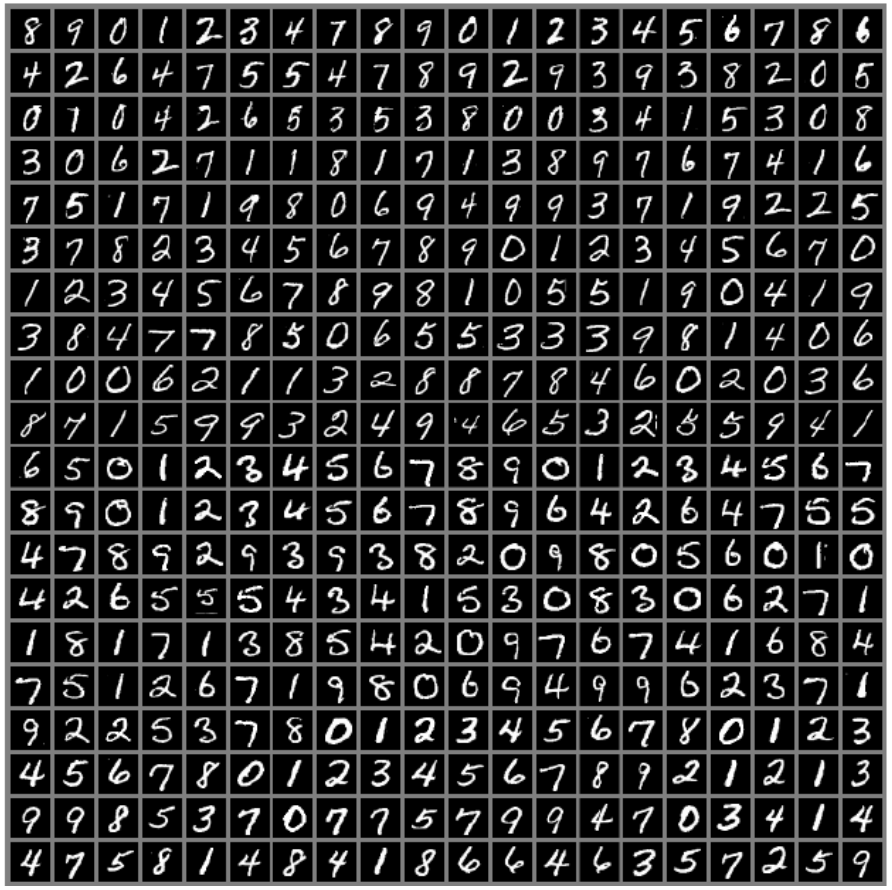
TensorFlow

Data



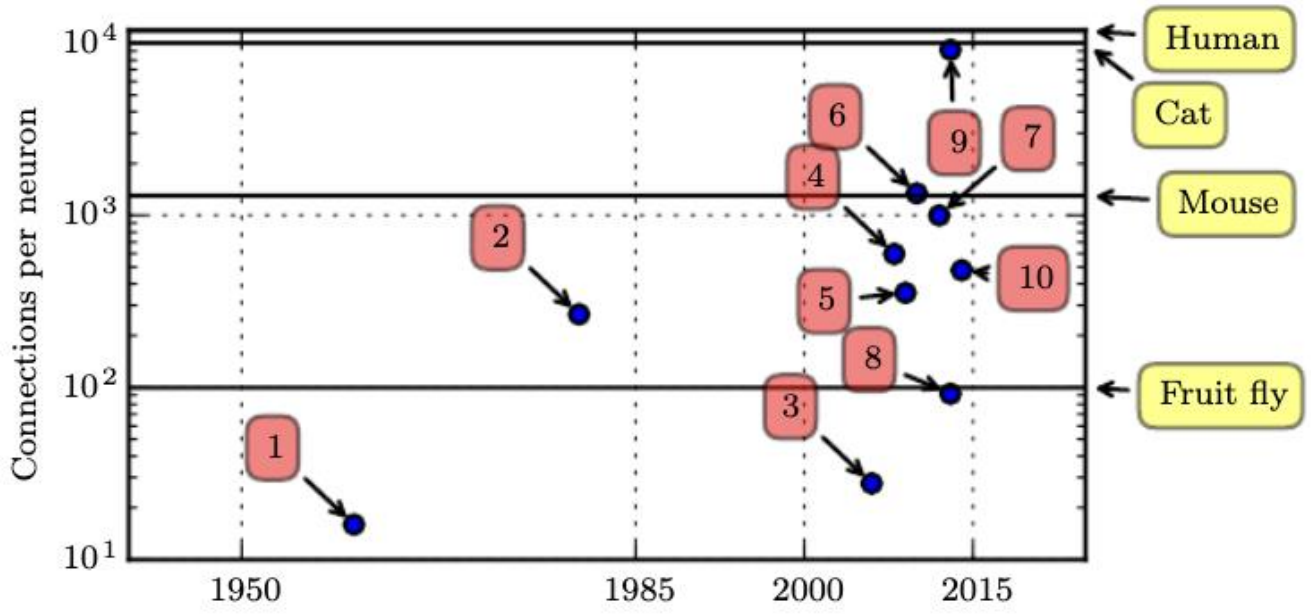
Increasing dataset size over time – Source: https://www.deeplearningbook.org/

Data – Example



Sample images from MNIST dataset

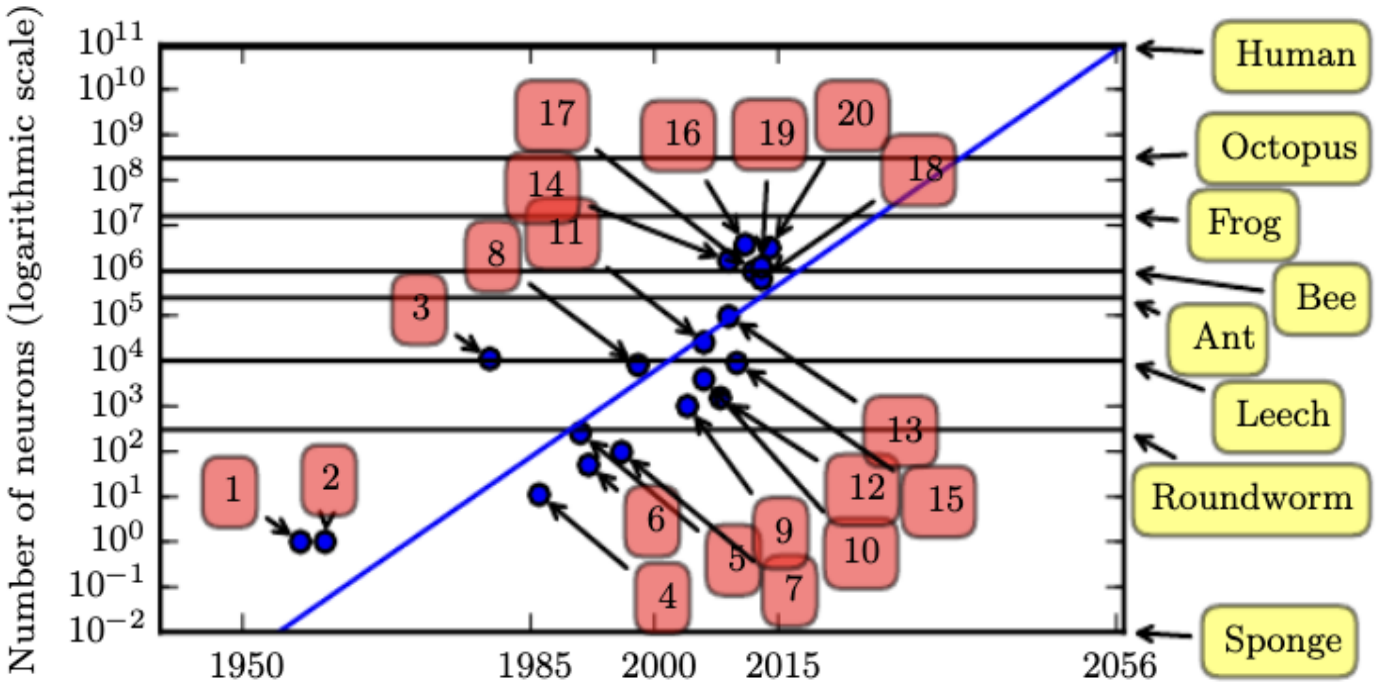
Neuron connections



1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

Number of connection per neuron over time - Source: https://www.deeplearningbook.org/

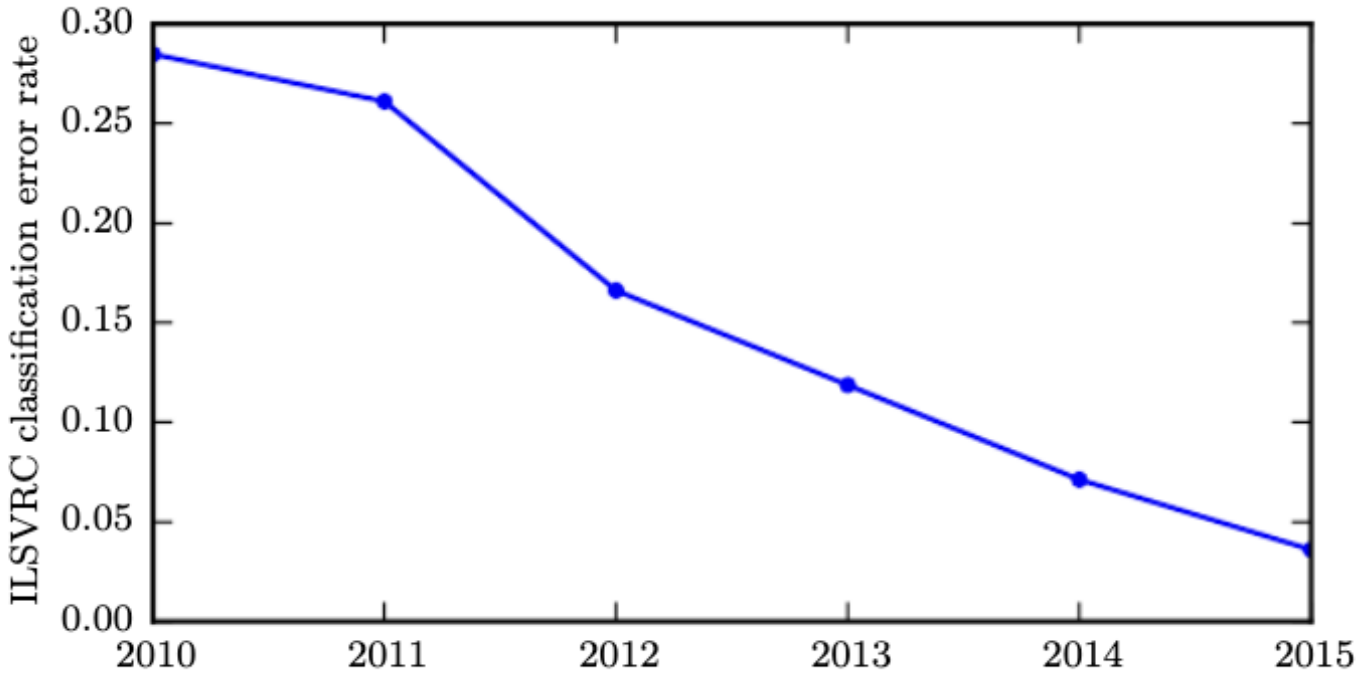
Network Size



1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

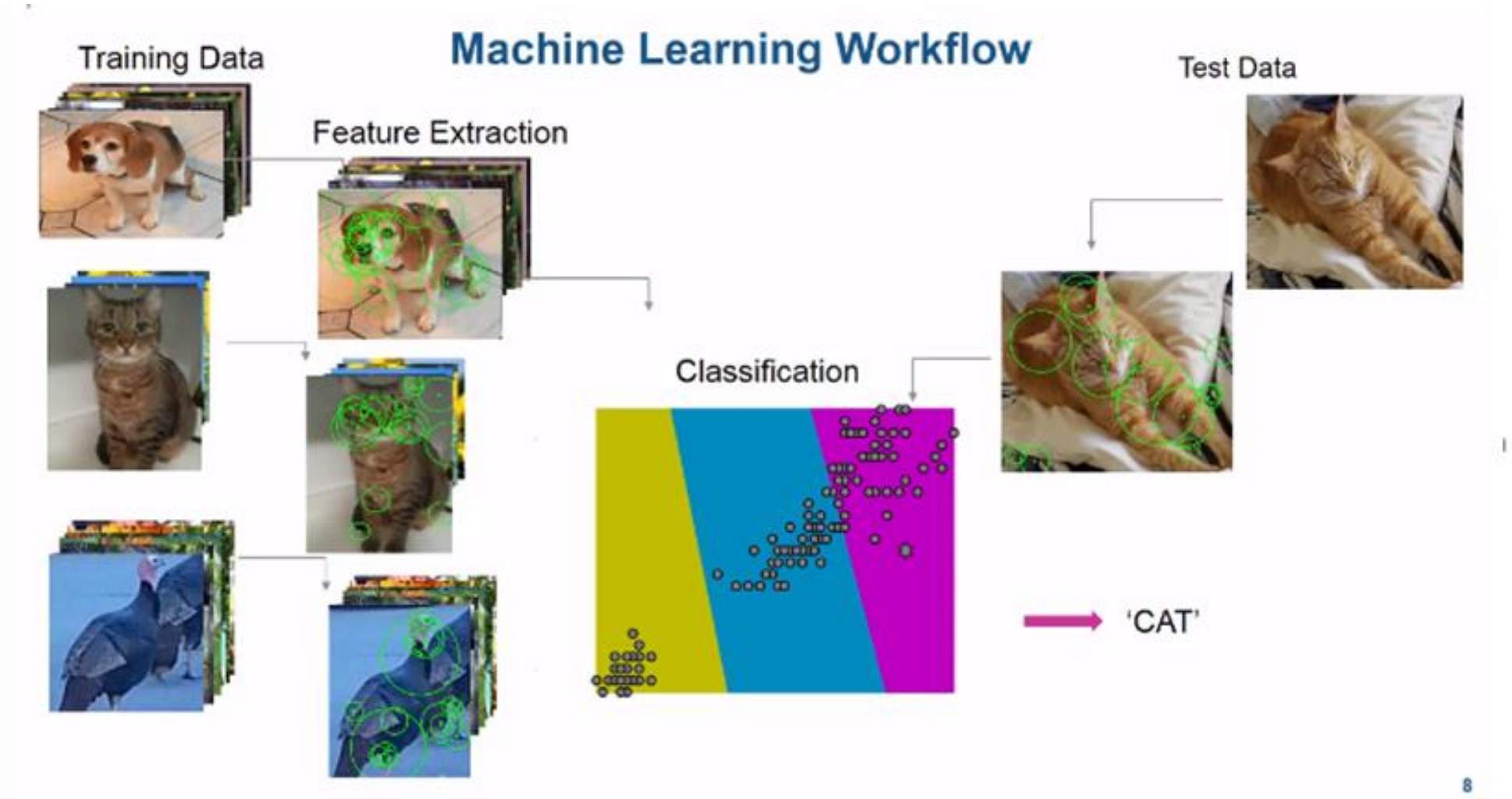
Increasing neural network size over time. - Source: https://www.deeplearningbook.org/

Network error



Decreasing error over time. - Source: https://www.deeplearningbook.org/

Until Now…


Machine Learning Workflow
Training Data
Feature Extraction
Test Data
Classification
'CAT'

Recall : What is Deep Learning

Composition of non-linear transformation of the data.

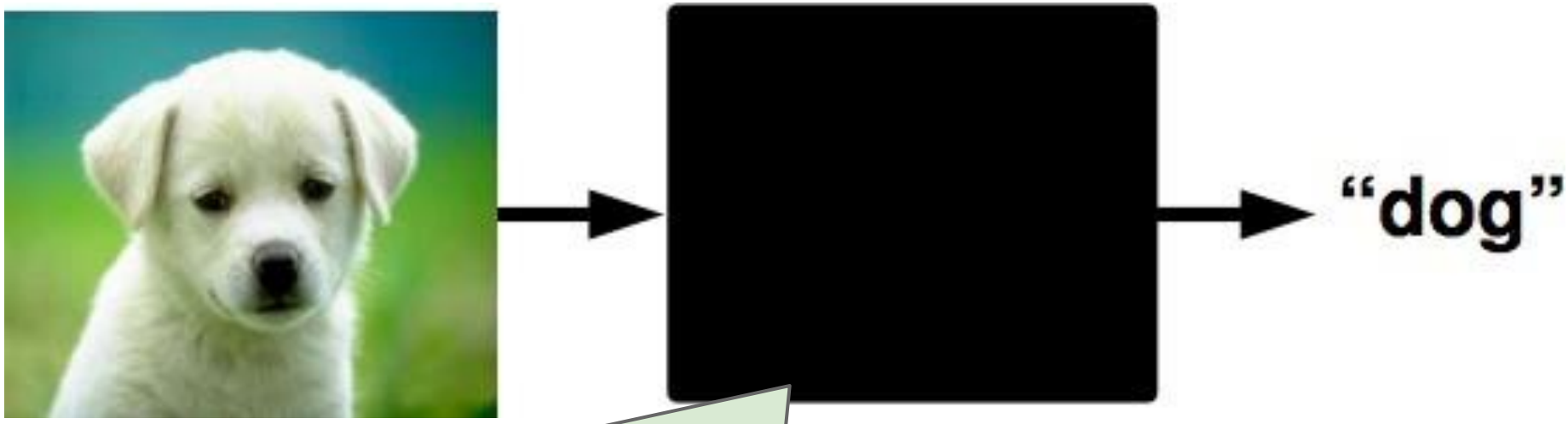
Goal: Learn useful representations – features- directly from data

Deep learning is about learning feature representation in a compositional manner.

But wait,

why learn features?

The Black Box in a Traditional Recognition Approach



Preprocessing

Feature Extraction (HOG, SIFT, etc)

Post-processing (Feature selection, MKL etc)

Classifier (SVM, boosting, etc)

The Black Box in a Traditional Recognition Approach

"dog"

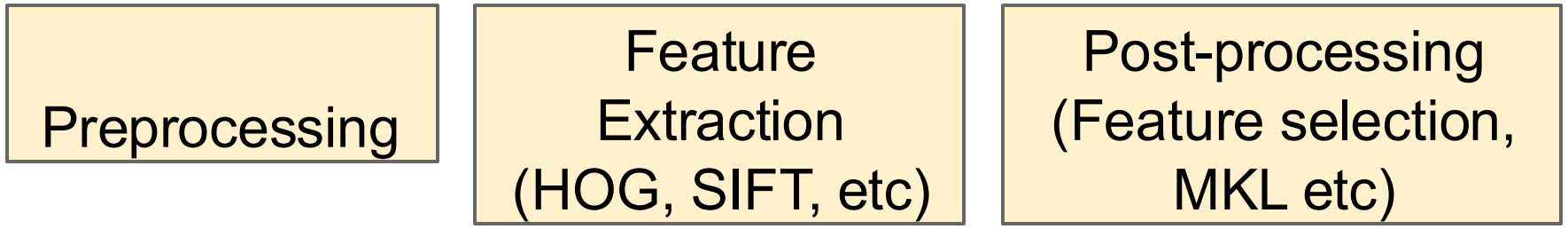
Hand Engineered

Preprocessing

Feature Extraction (HOG, SIFT, etc)

Post-processing (Feature selection, MKL etc)

Classifier (SVM, boosting, etc)



- Most critical for accuracy
- Most time-consuming in development
- What is the best feature???
- What is next?? Keep on crafting better features?

⇒ Let's learn feature representation directly from data.

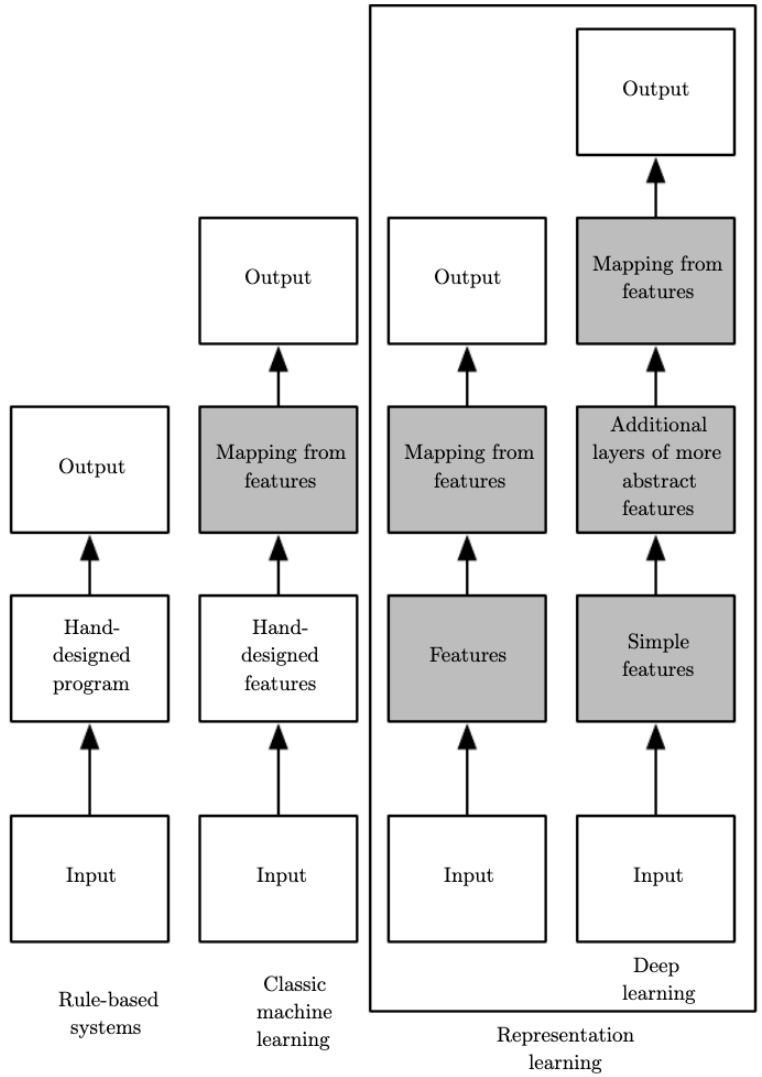
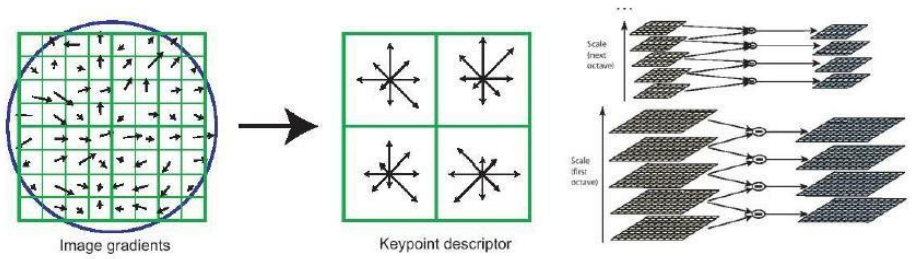


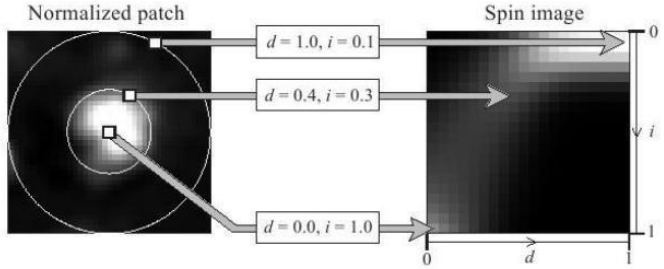
Figure 1.5: Flowcharts showing how the different parts of an AI system relate to each other within different AI disciplines. Shaded boxes indicate components that are able to learn from data.

https://www.deeplearningbook.org/

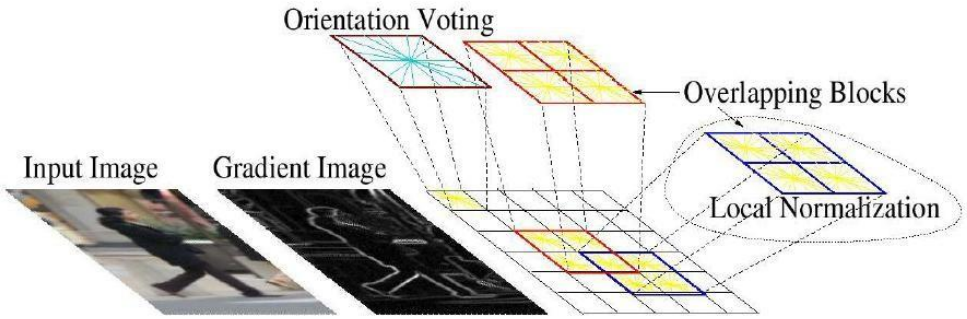
Computer vision features



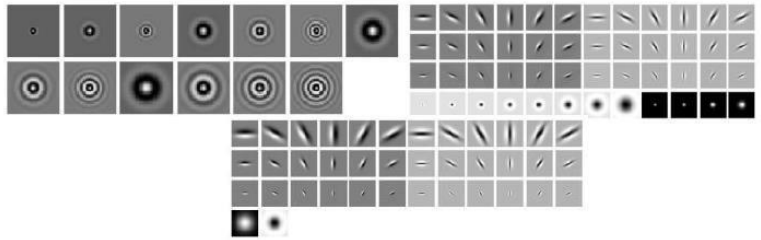
SIFT



Spin image



HoG

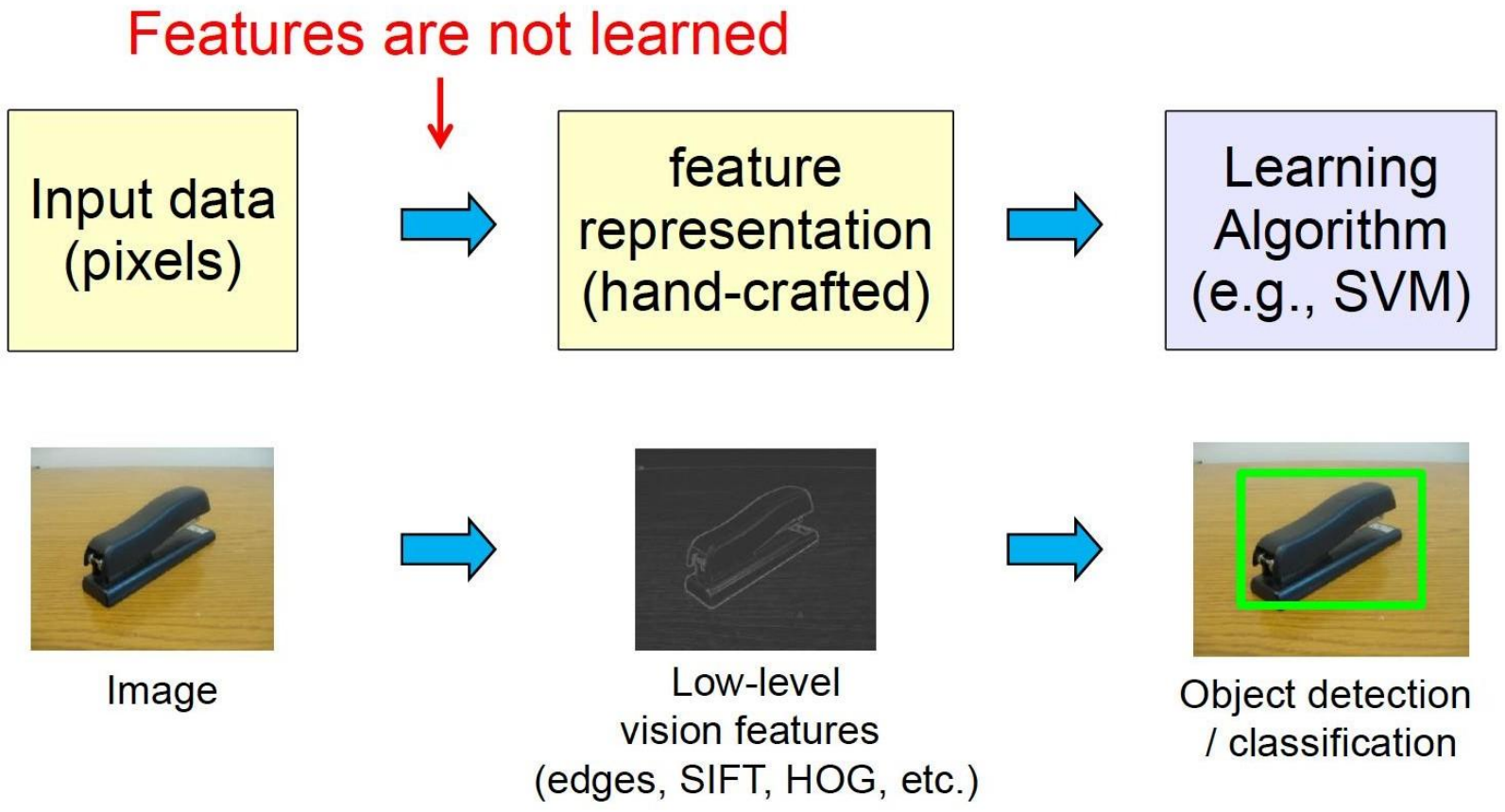


Textons

and many others:

SURF, MSER, LBP, Color-SIFT, Color histogram, GLOH, …..

Traditional Recognition Approach



Labradoodle or fried chicken

Puppy or bagel

Sheepdog or mop

Feature Engineering vs. Learning

- Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work.

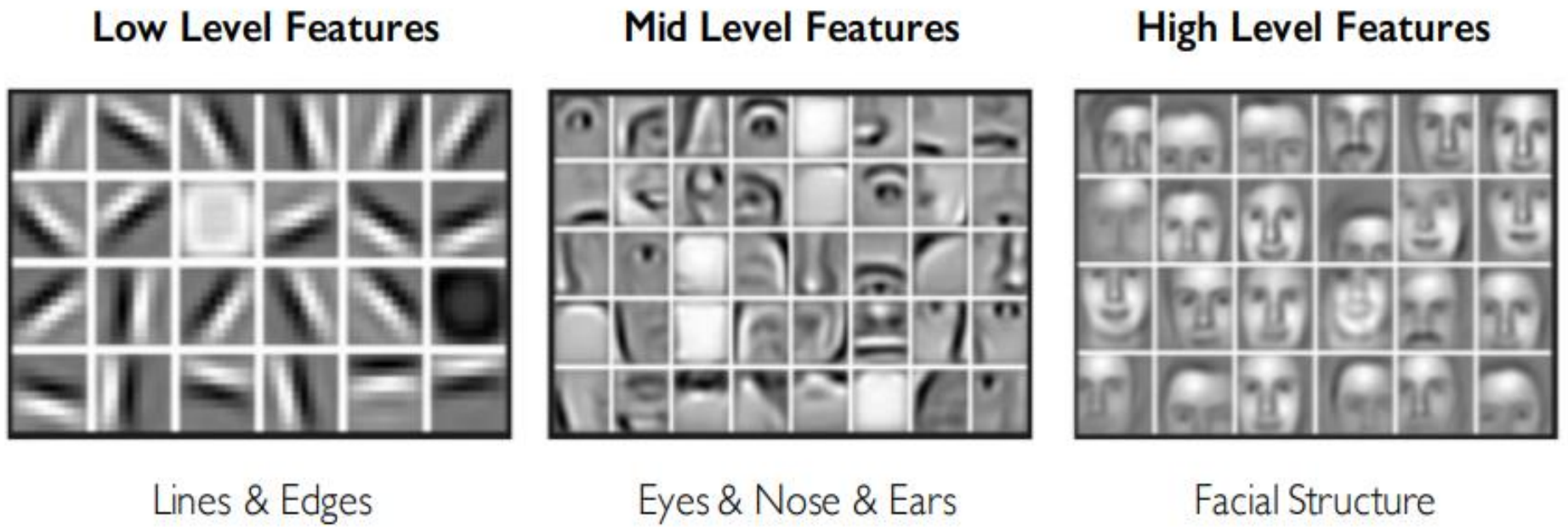
- “Coming up with features is difficult, time-consuming, requires expert knowledge.”

--Andrew Ng

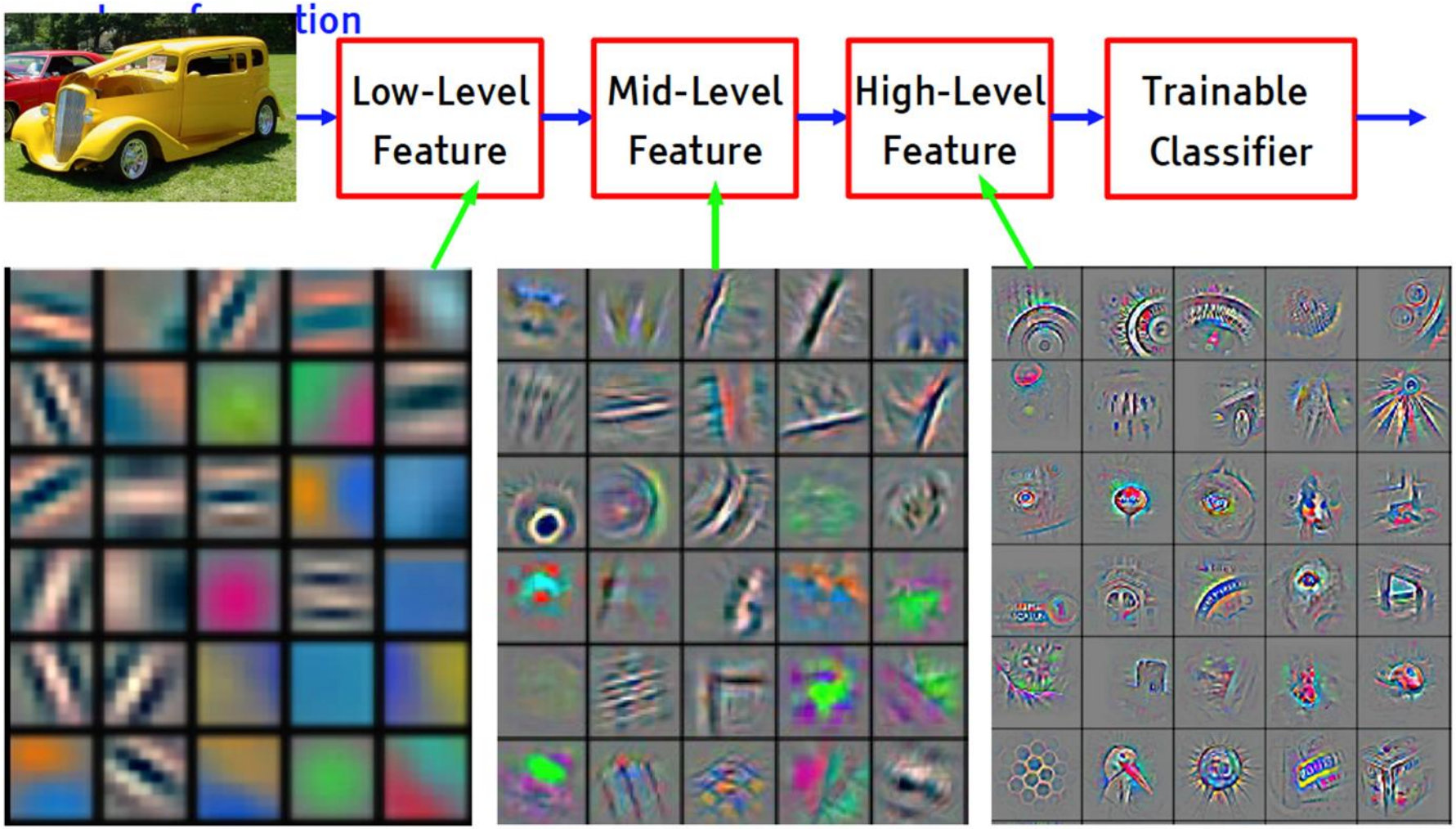
Deep Learning = Learning Hierarchical Representations

Hand engineered features are time consuming, brittle and not scalable in practice

Can we learn the **underlying features** directly from data?

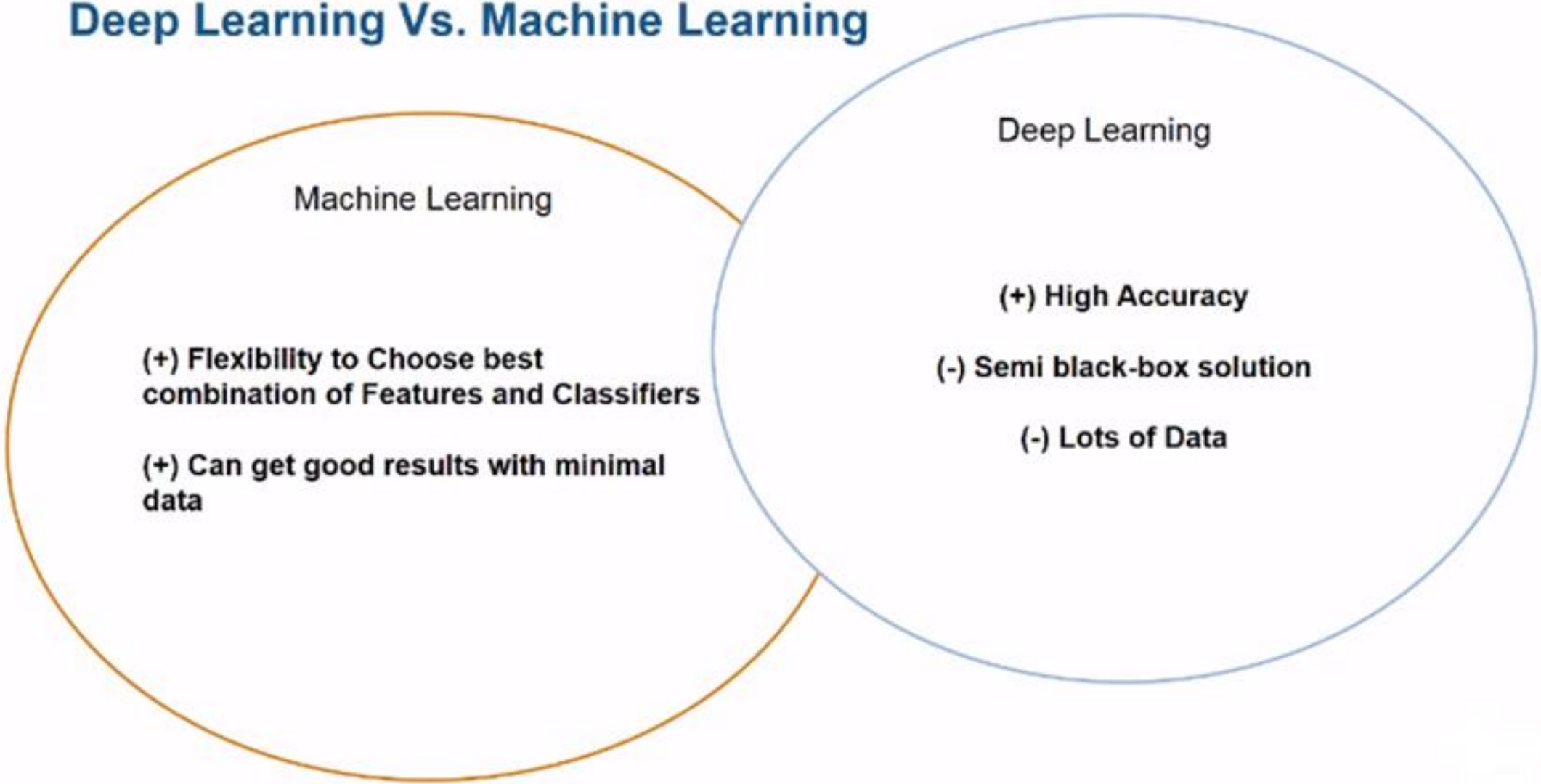


Deep Learning = Learning Hierarchical Representations

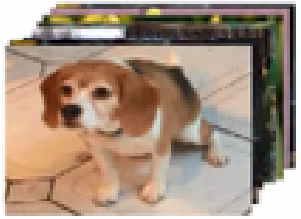


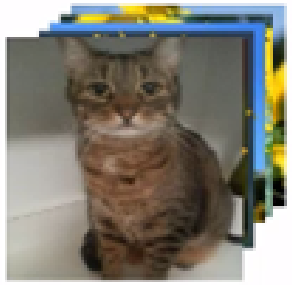
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

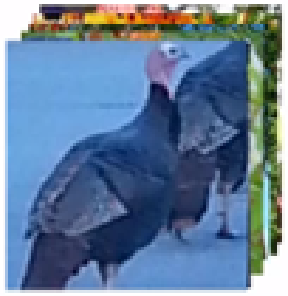
Deep Learning Vs. Machine Learning



Deep Learning + Machine Learning Combined



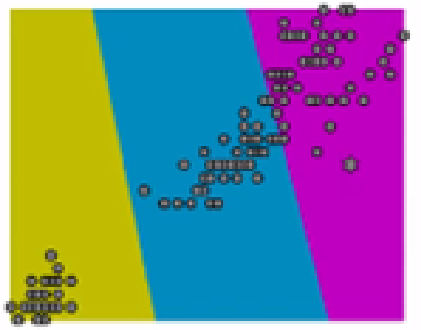




Feature Learning

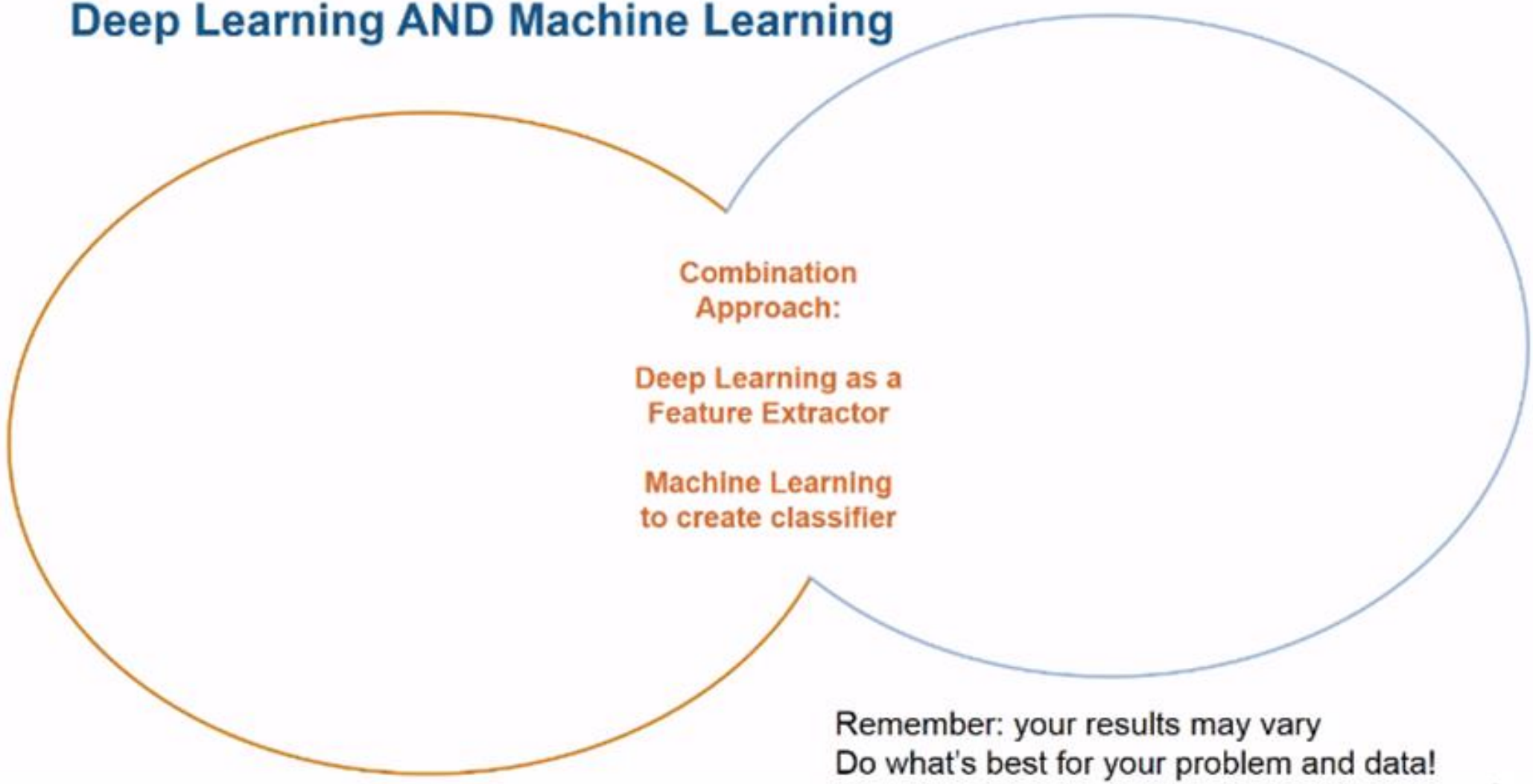
Deep Learning

Classification



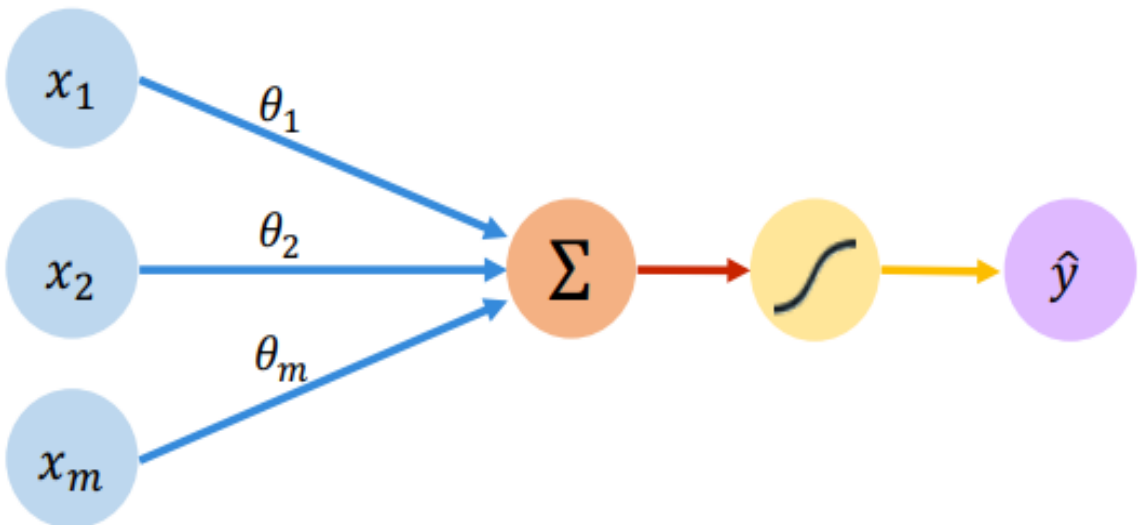
Machine Learning

Deep Learning AND Machine Learning



Remember: your results may vary
Do what's best for your problem and data!

The Perceptron: Forward Propagation



Linear combination of inputs

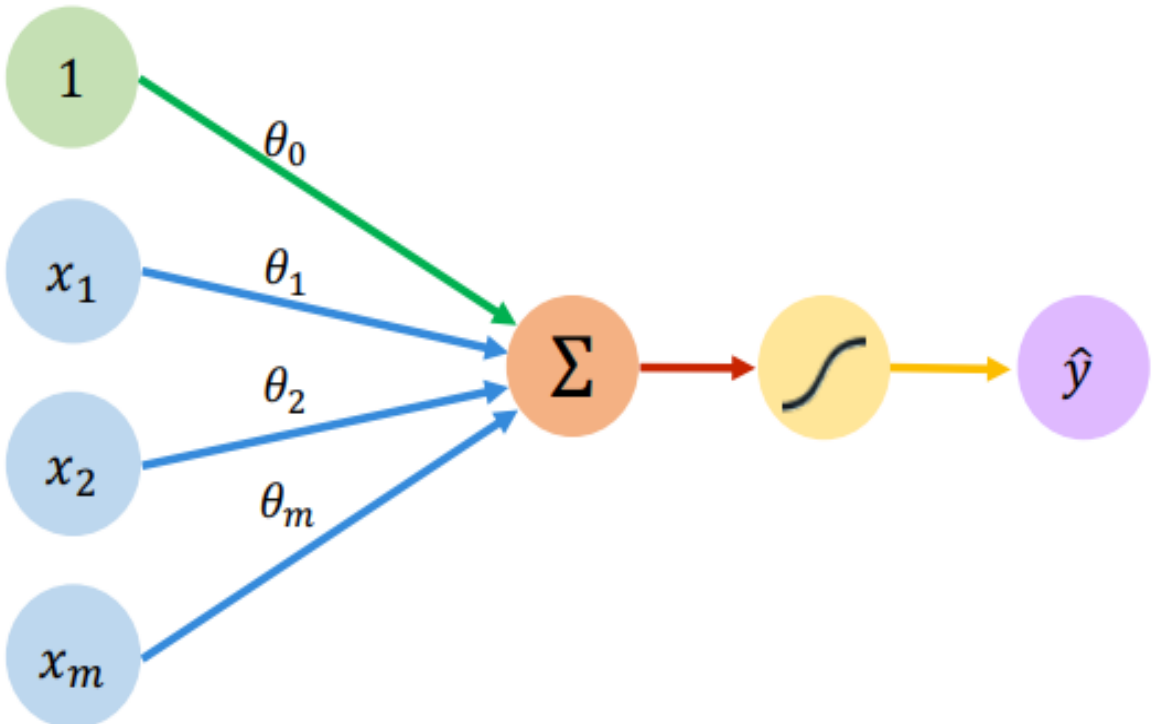
Output

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \, \theta_i\right)$$

Non-linear activation function

Inputs Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

Output

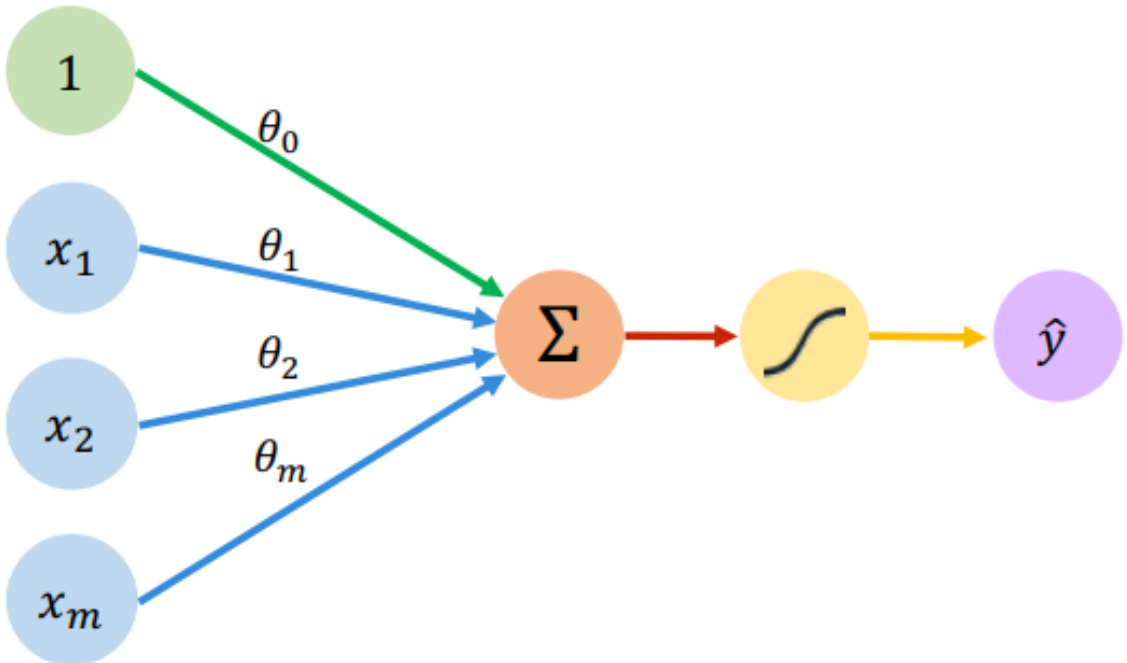
Linear combination of inputs

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\ \theta_i\right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation



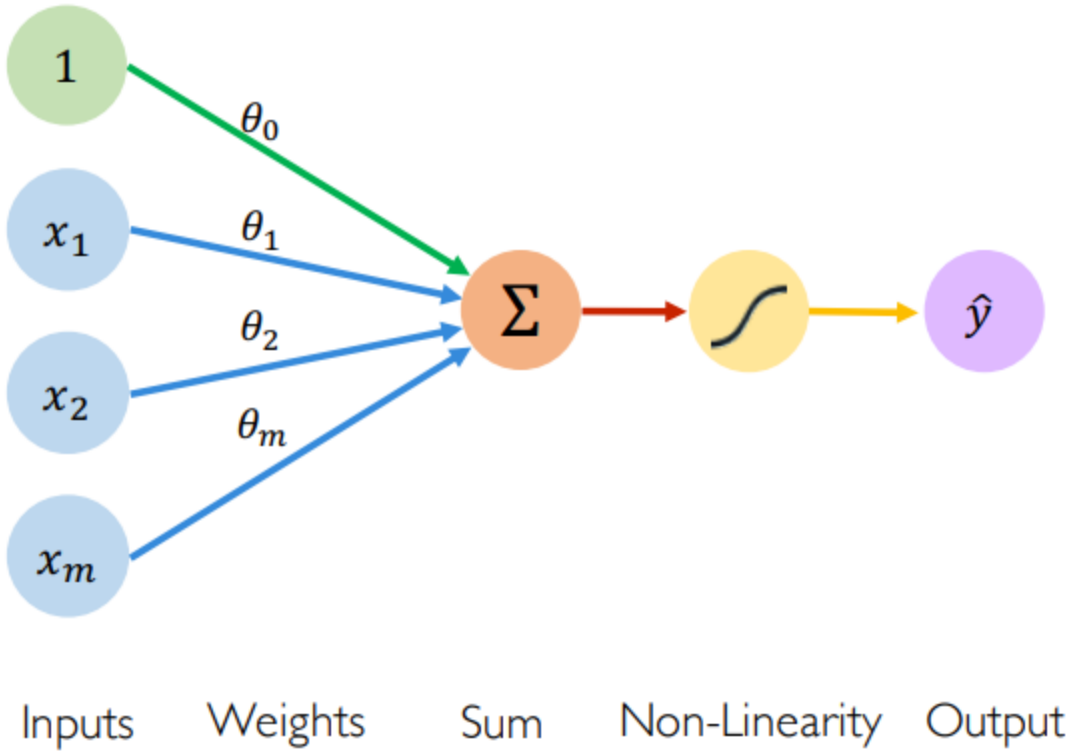
Inputs Weights Sum Non-Linearity Output

$$\hat{y} = g\left(\theta_0 + \sum_{i=1}^{m} x_i\, \theta_i\right)$$

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T \boldsymbol{\theta}\right)$$

where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix}$

The Perceptron: Forward Propagation

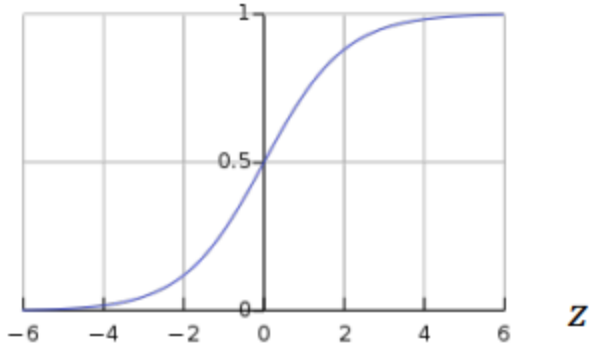


Activation Functions

$$\hat{y} = g\left(\theta_0 + \boldsymbol{X}^T\boldsymbol{\theta}\right)$$

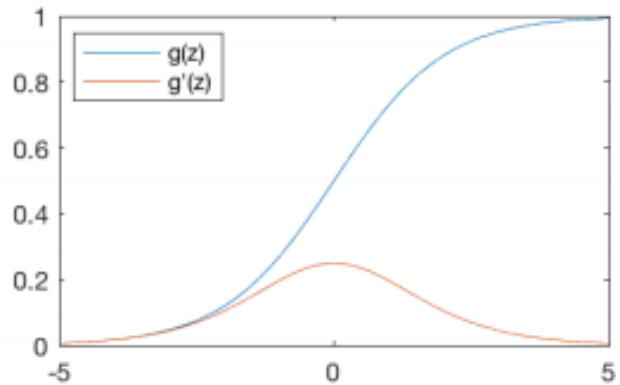
- Example: sigmoid function

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



Common Activation Functions

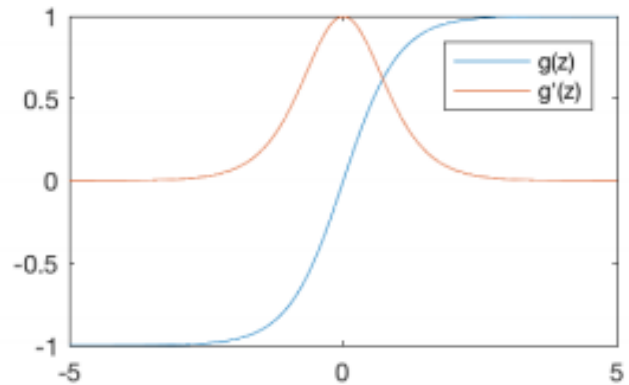
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

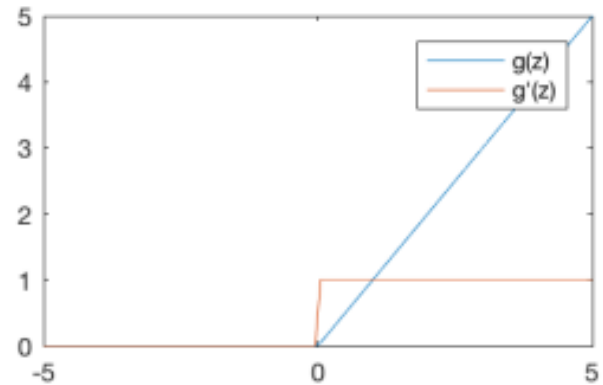
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

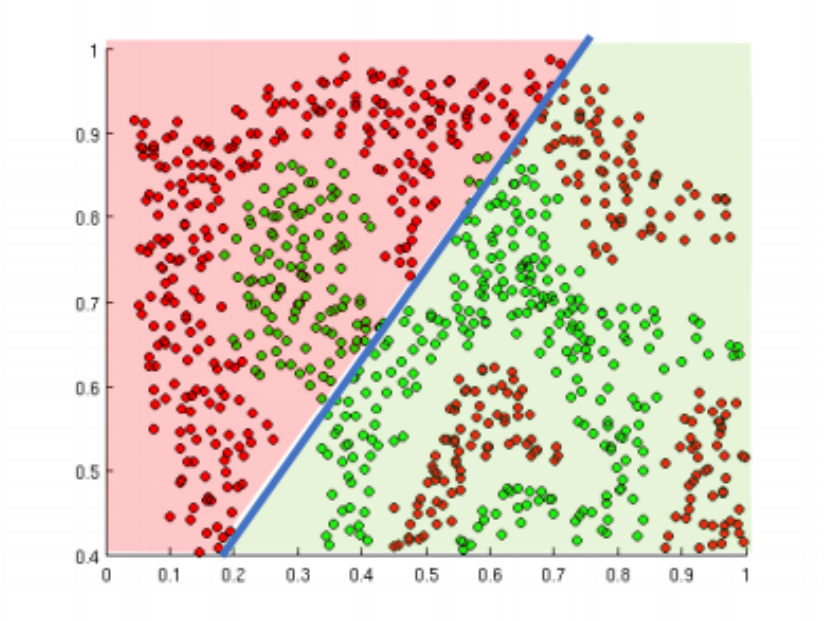


$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Importance of Activation Functions

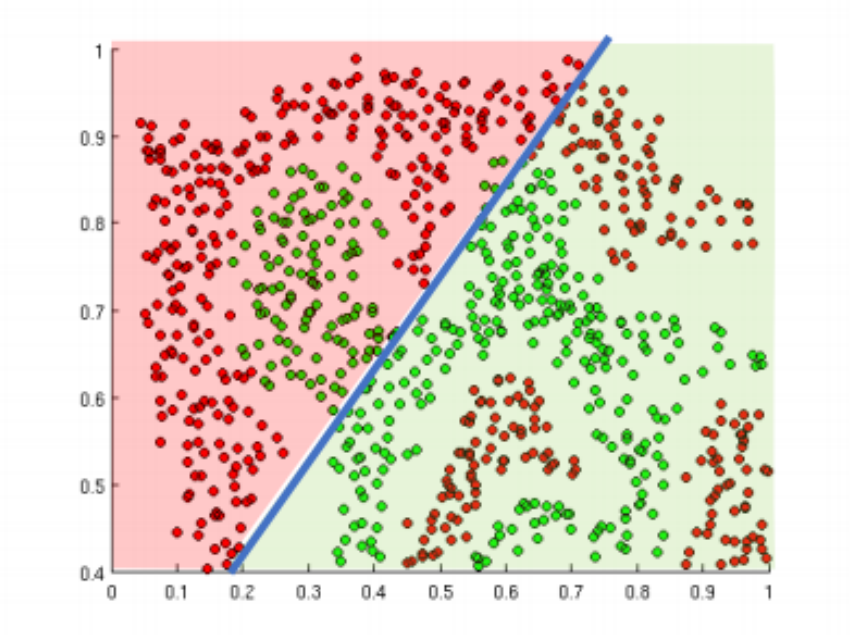
The purpose of activation functions is to ***introduce non-linearities*** *into the network*



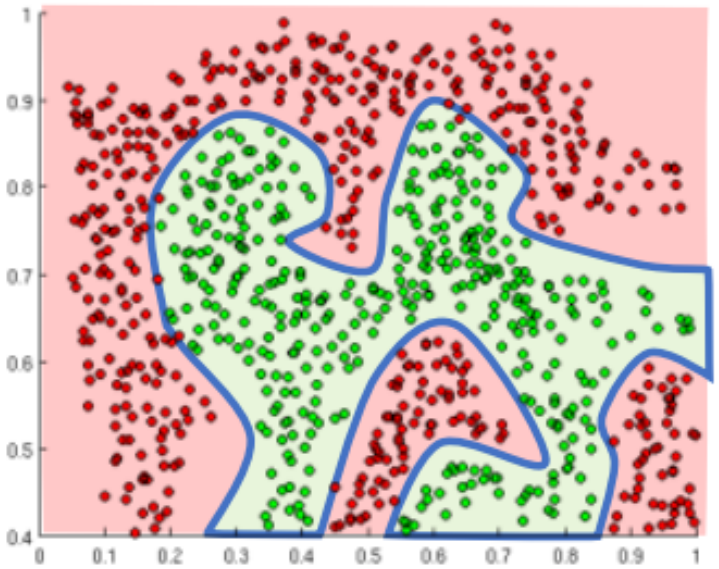
Linear Activation functions produce linear decisions no matter the network size

Importance of Activation Functions

The purpose of activation functions is to ***introduce non-linearities*** *into the network*

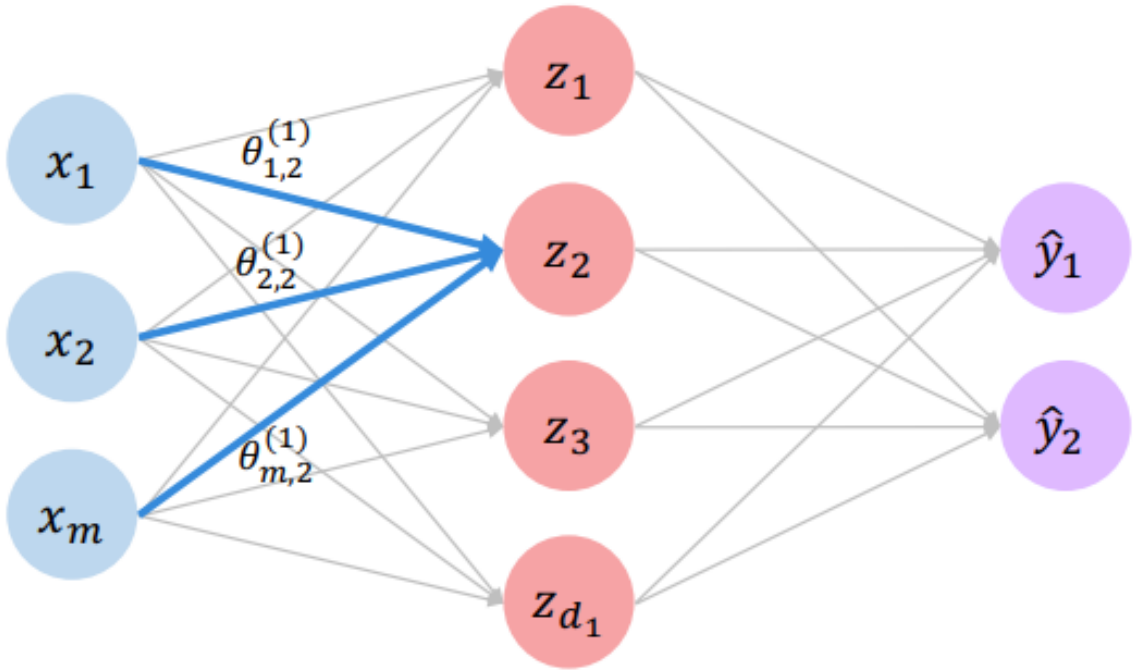


Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions

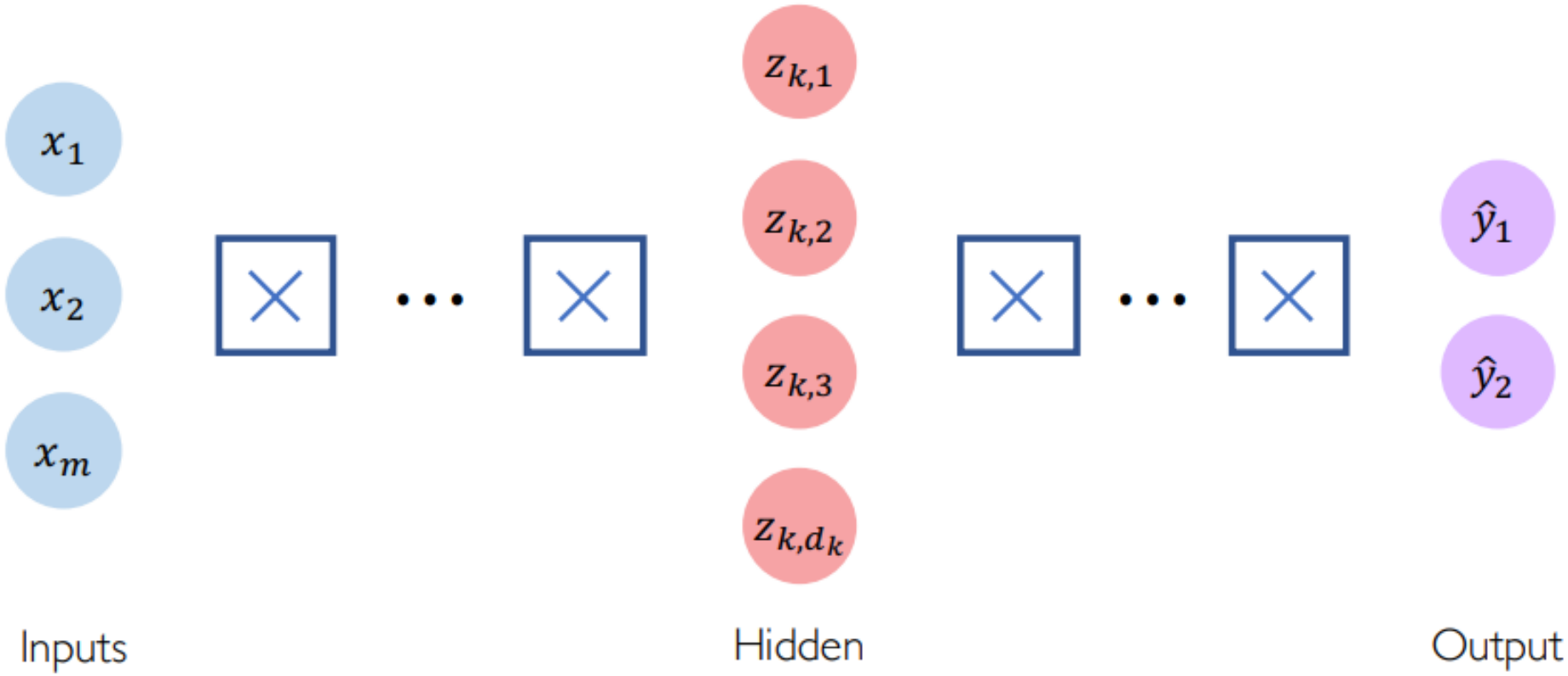
Single Layer Neural Network



$$z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, \theta_{j,2}^{(1)}$$

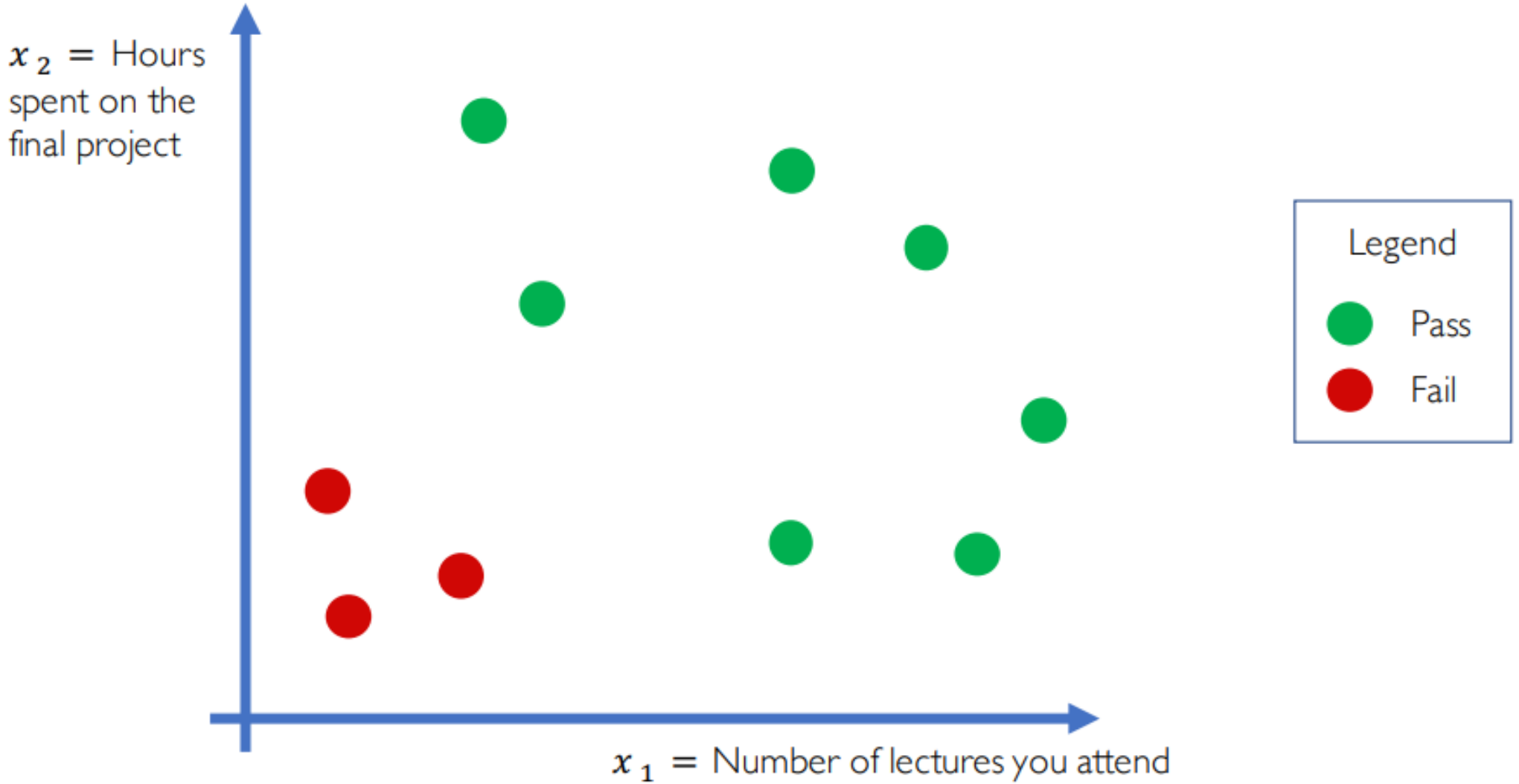
$$= \theta_{0,2}^{(1)} + x_1 \, \theta_{1,2}^{(1)} + x_2 \, \theta_{2,2}^{(1)} + x_m \, \theta_{m,2}^{(1)}$$

Deep Neural Network

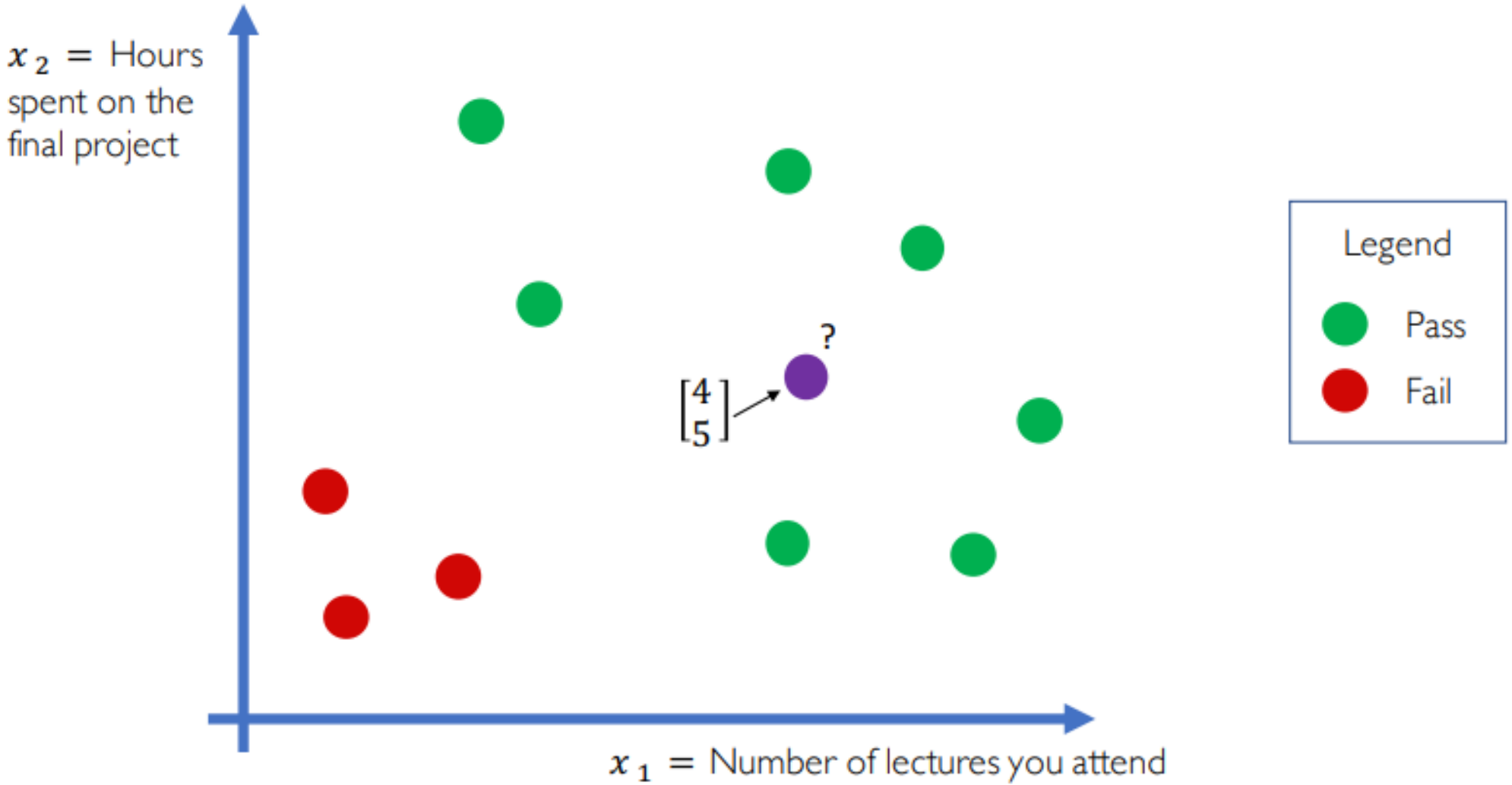


$$z_{k,i} = \theta_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, \theta_{j,i}^{(k)}$$

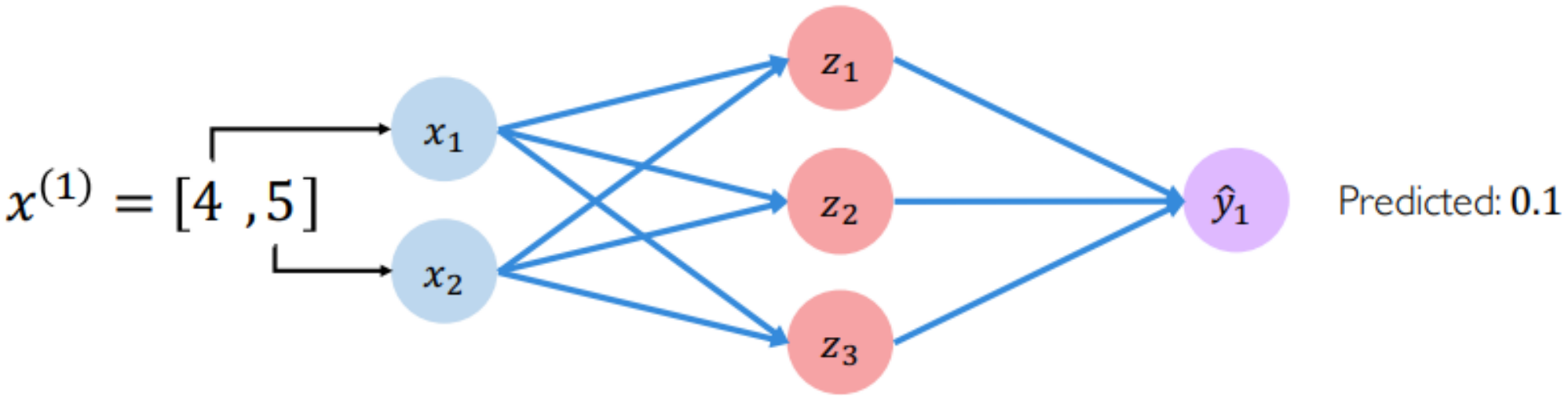
Example Problem: Will I pass this class?



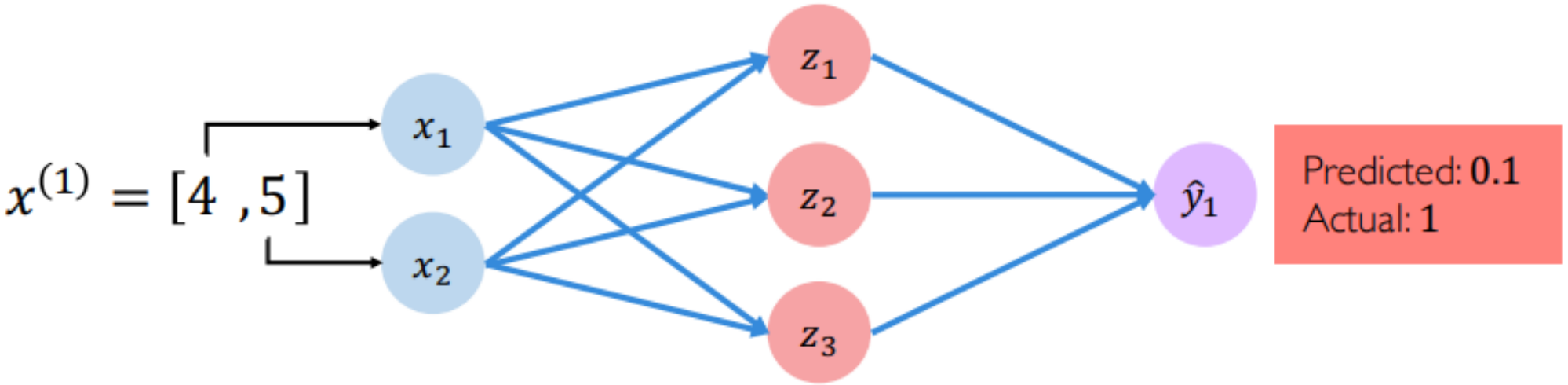
Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

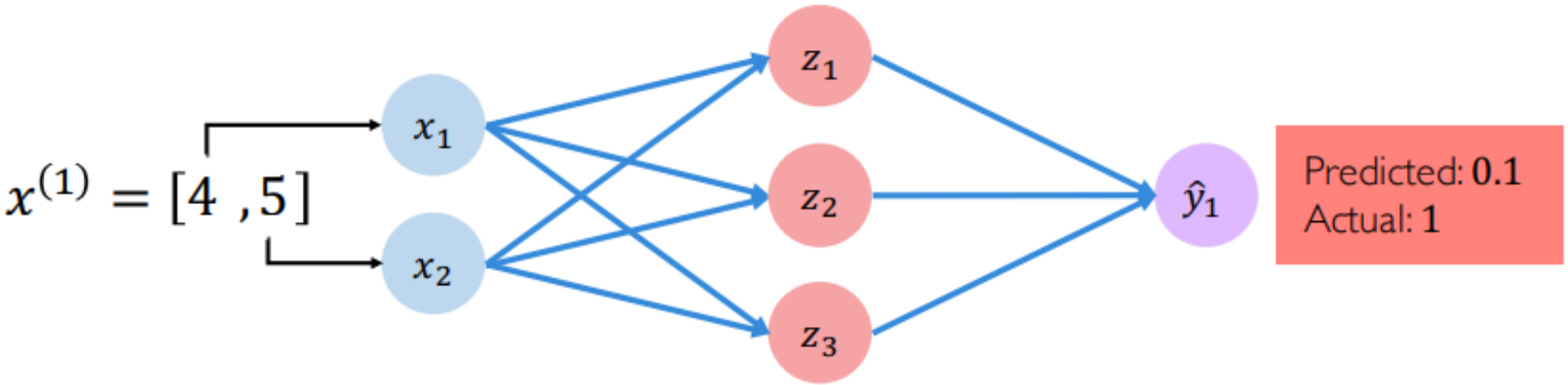


Example Problem: Will I pass this class?



Quantifying Loss

*The **loss** of our network measures the cost incurred from incorrect predictions*



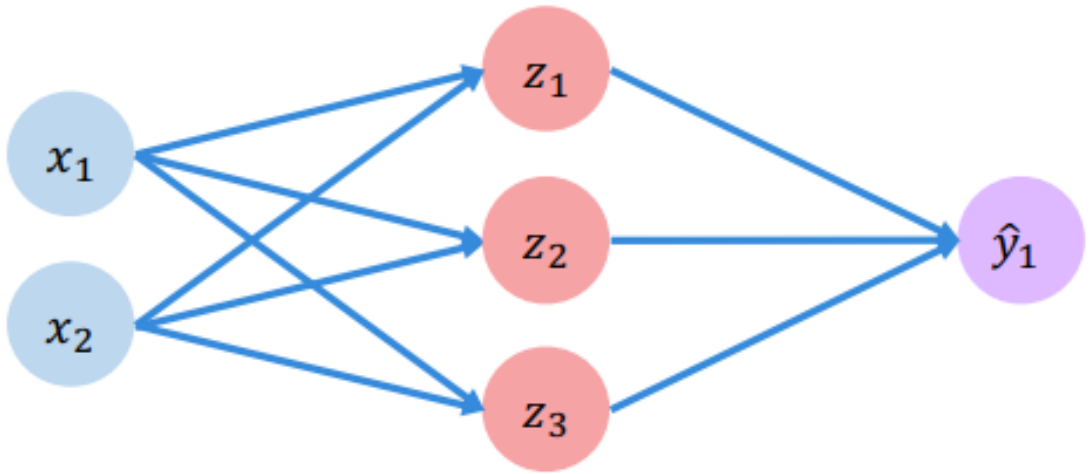
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

*The **empirical loss** measures the total loss over our entire dataset*

$$\boldsymbol{x} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

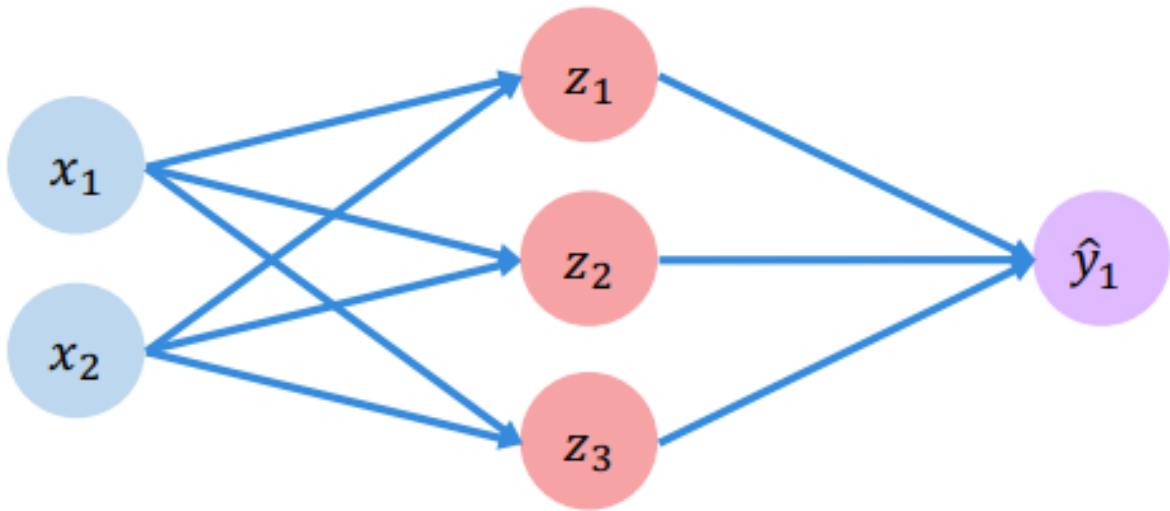
Also known as:
- Objective function
- Cost function
- Empirical Risk

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

Binary Cross Entropy Loss

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

$$\boldsymbol{x} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$ y

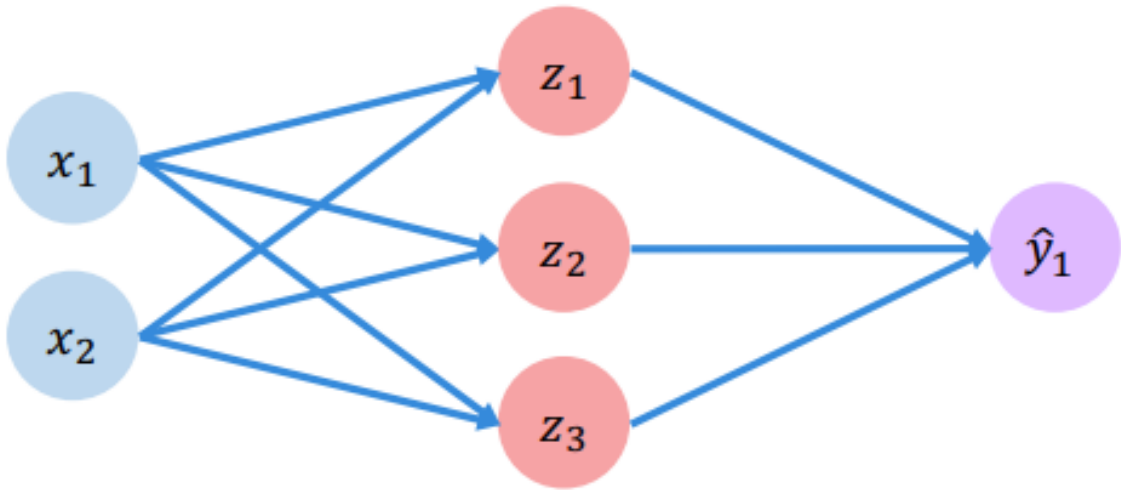
$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

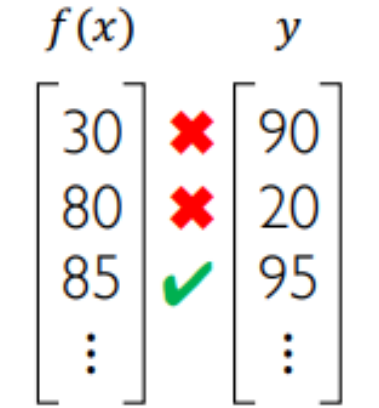
$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log\Big(1 - \underbrace{f\big(x^{(i)};\theta\big)}_{\text{Predicted}}\Big)$$

Mean Squared Error Loss

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

$$\boldsymbol{x} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$





Final Grades (percentage)

$$\boldsymbol{J(\theta)} = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)};\theta\right)\right)^2$$

Actual Predicted

Training Neural Networks

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{\theta}\right), y^{(i)}\right)$$

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\text{argmin}} J(\boldsymbol{\theta})$$

Loss Functions Can Be Difficult to Optimize

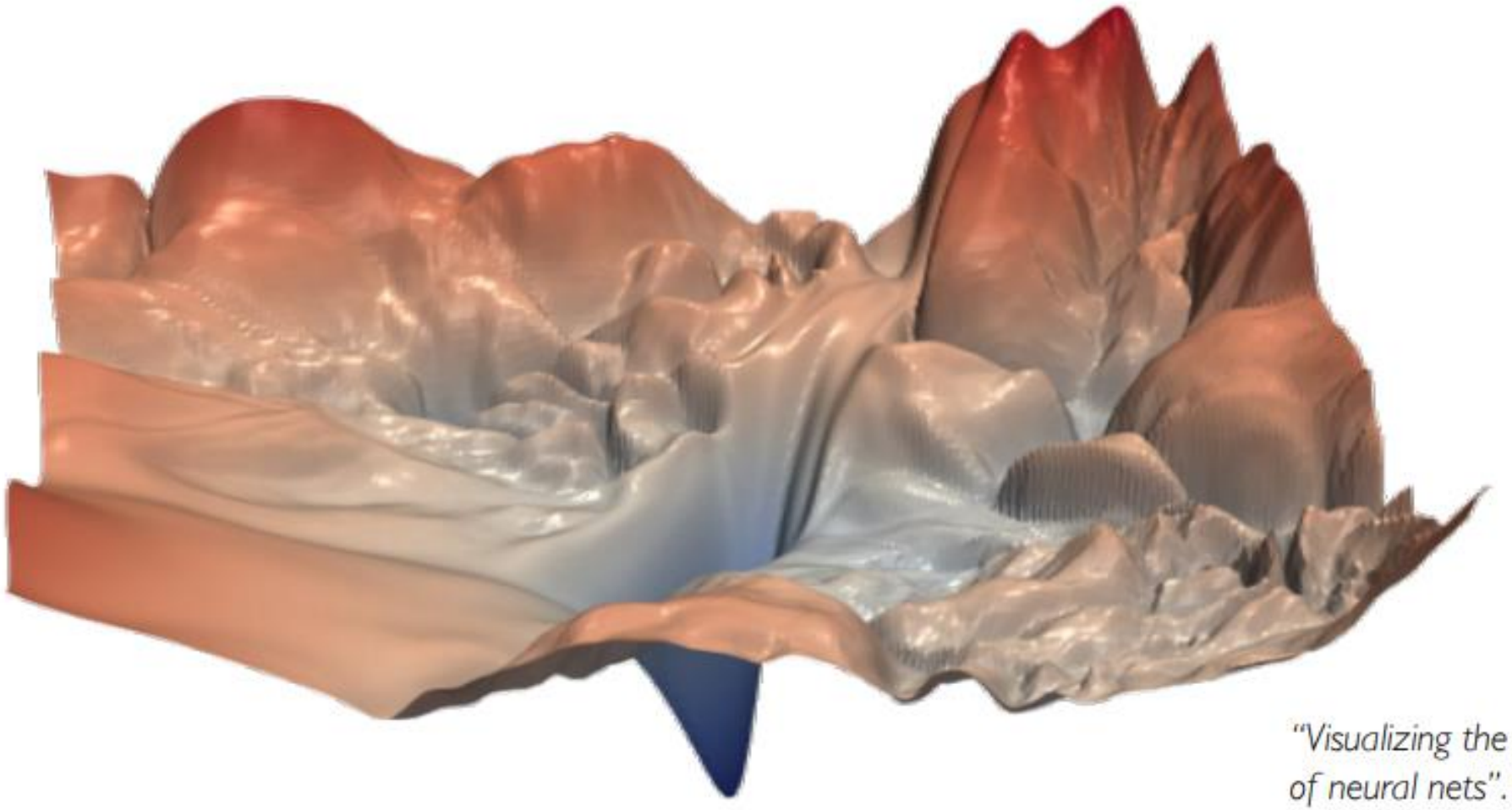
Remember:

Optimization through gradient descent

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

How can we set the learning rate?

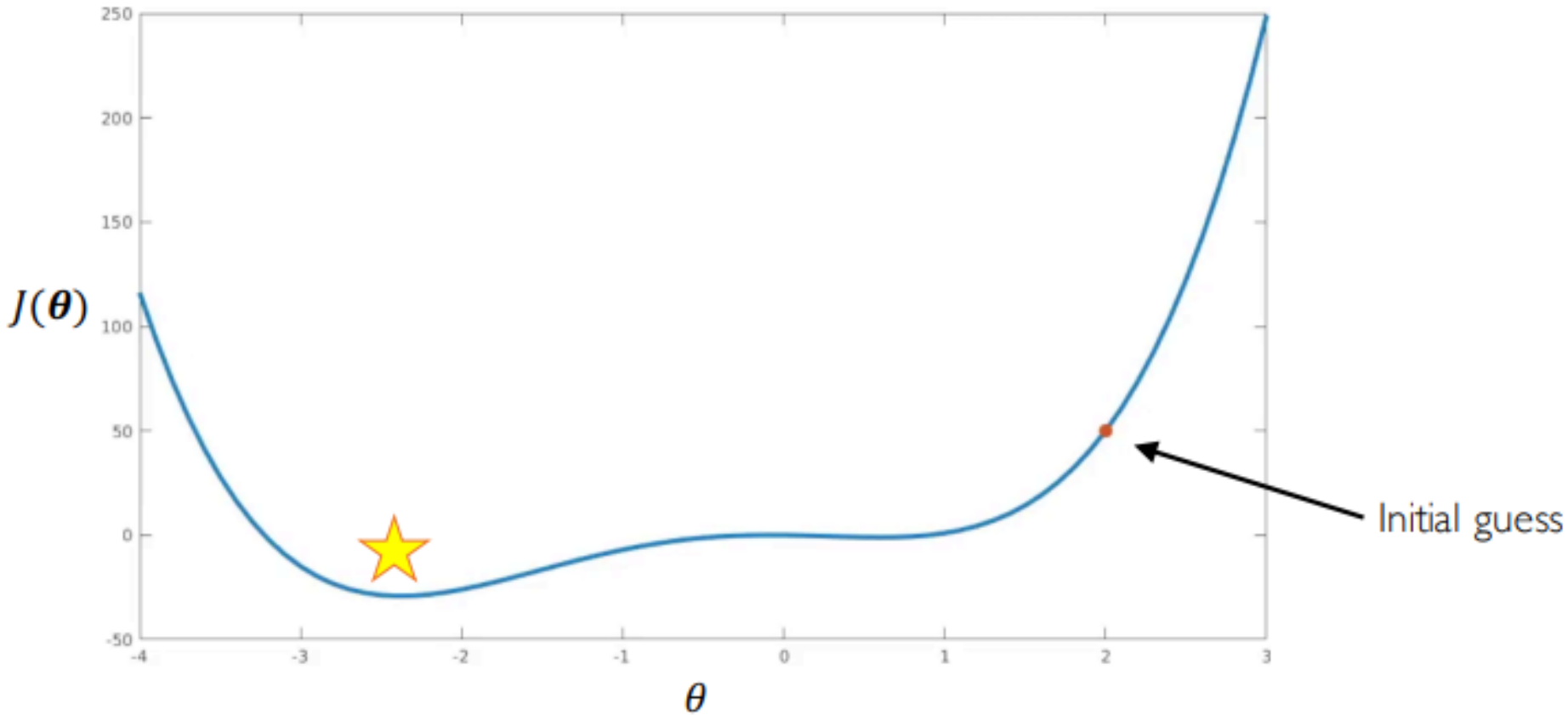
Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

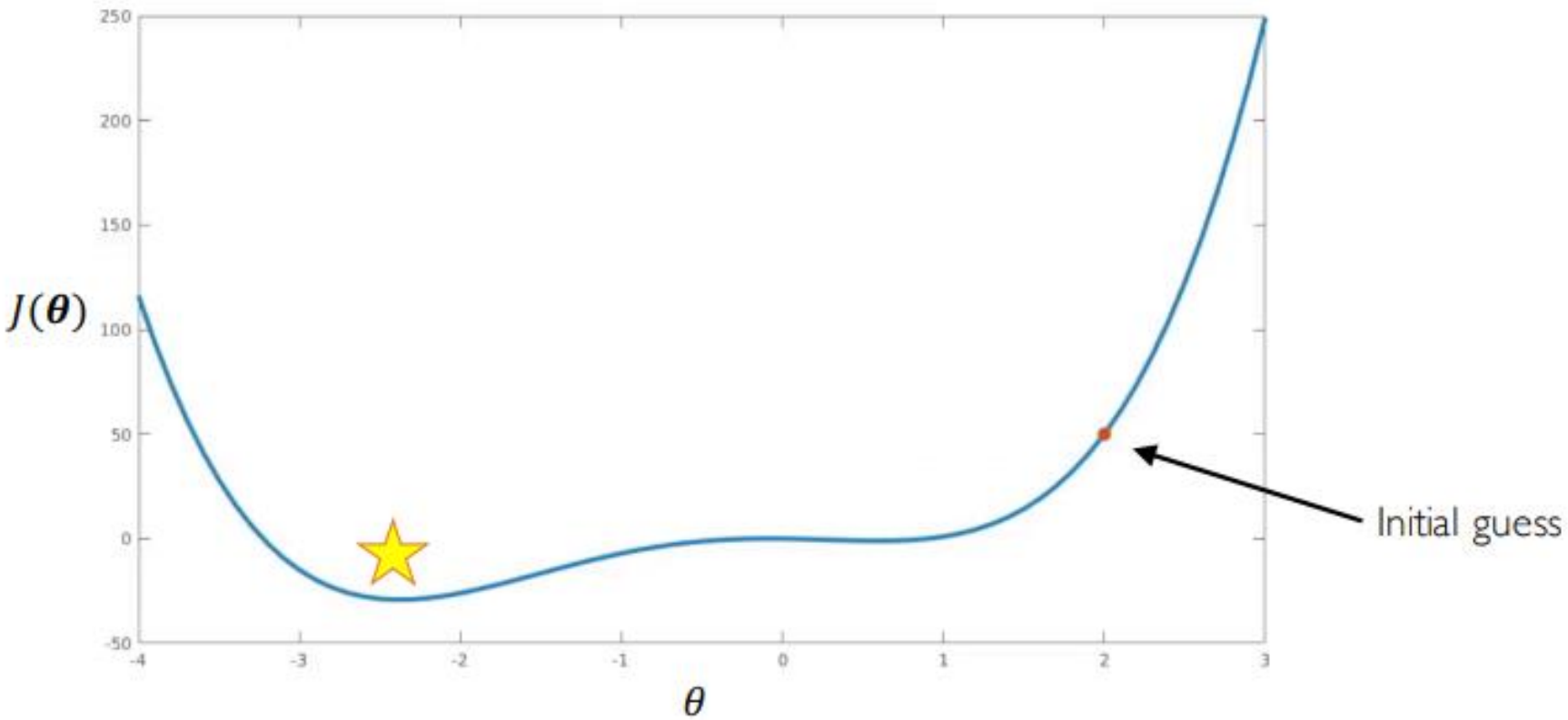
Setting the Learning Rate

Small learning rate *converges slowly and gets stuck in false local minima*



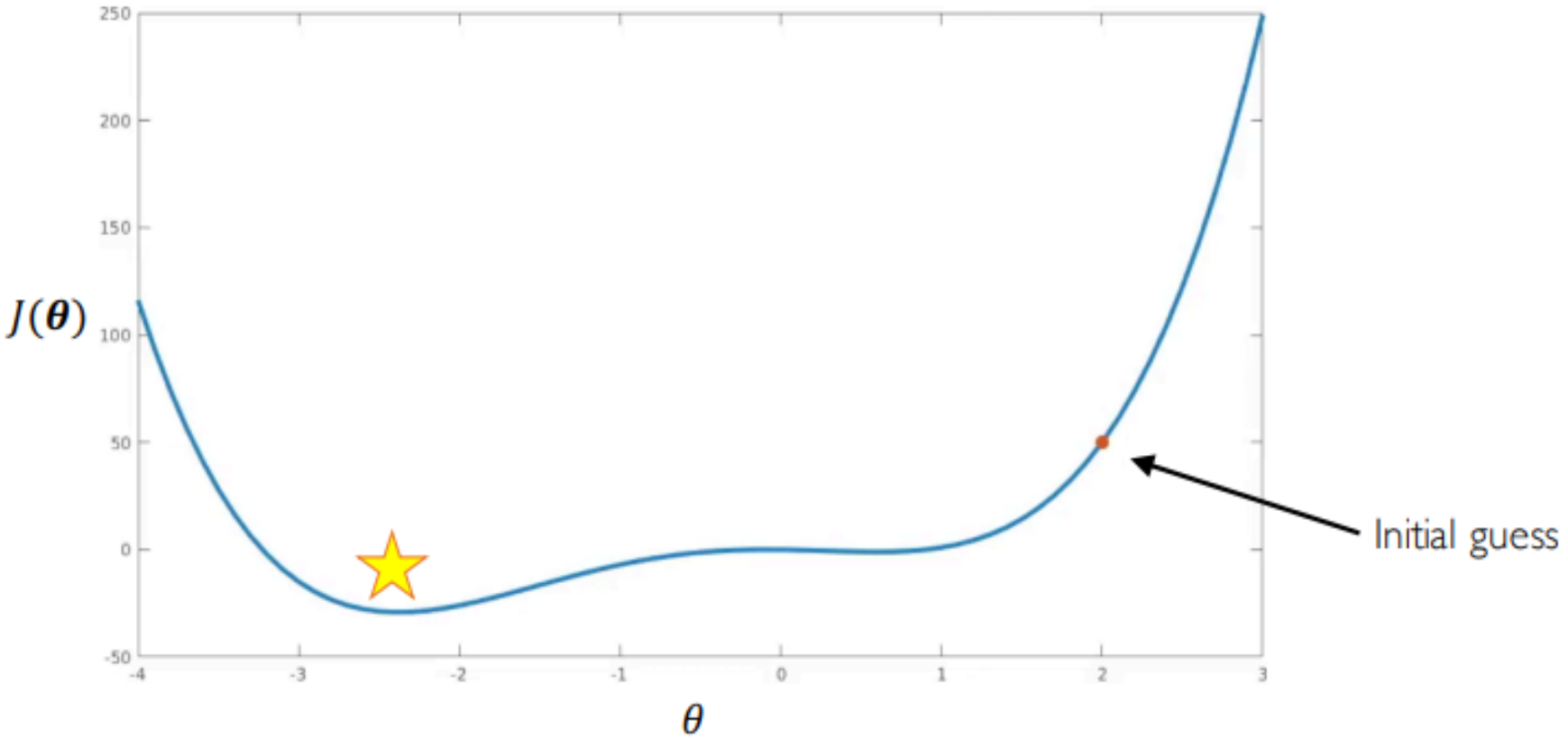
Setting the Learning Rate

Large learning rates *overshoot, become unstable and diverge*



Setting the Learning Rate

Stable learning rates *converge smoothly and avoid local minima*



How to deal with this?

Idea 1:

Try lots of different learning rates and see what works "just right"

Or

Idea 2:

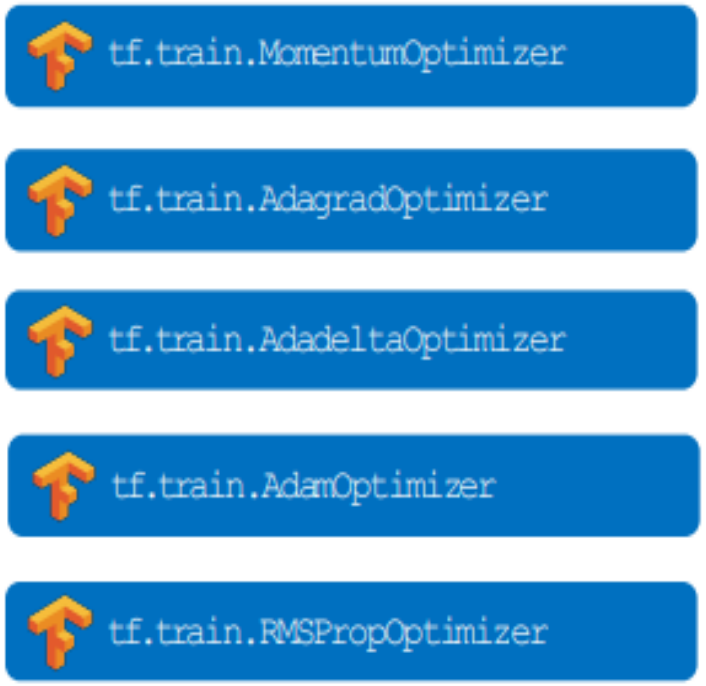
Do something smarter!
Design an adaptive learning rate that "adapts" to the landscape

Adaptive Learning Rates

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

Adaptive Learning Rate Algorithms

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Mini-batches

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

Allows for larger learning rates

Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

Regularization

What is it?

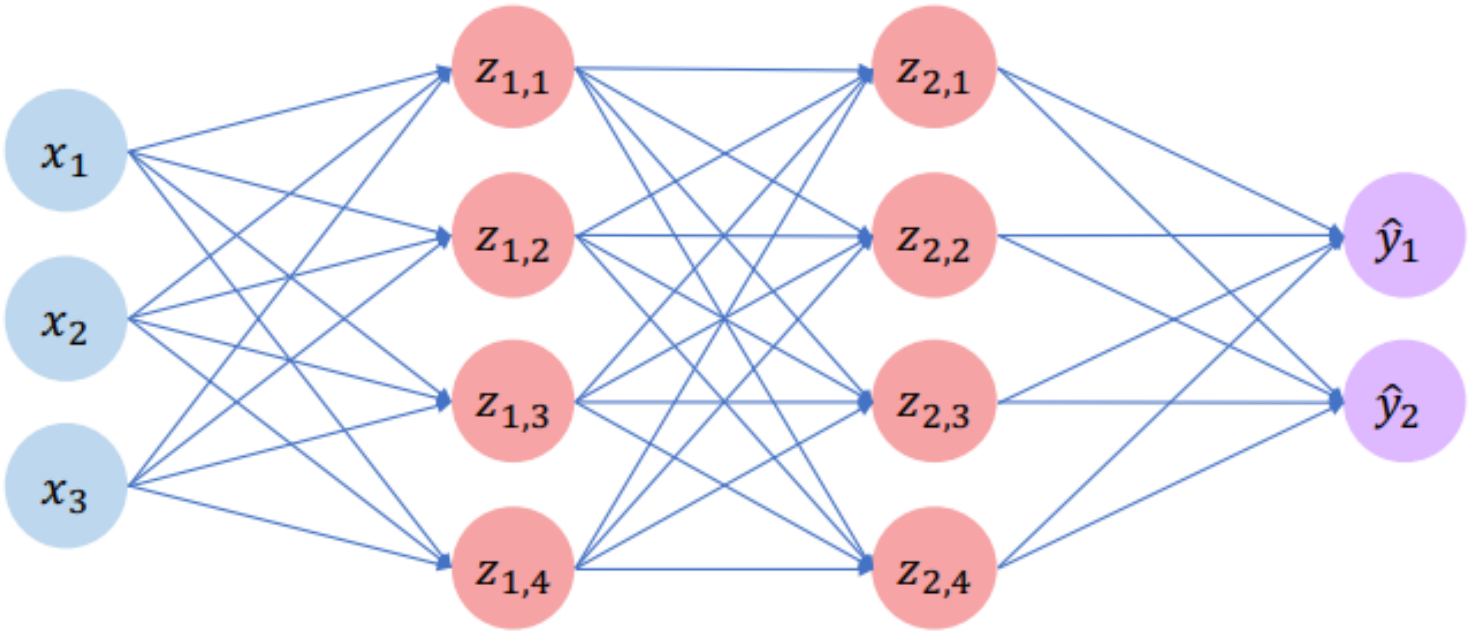
Technique that constrains our optimization problem to discourage complex models

Why do we need it?

Improve generalization of our model on unseen data

Regularization 1: Dropout

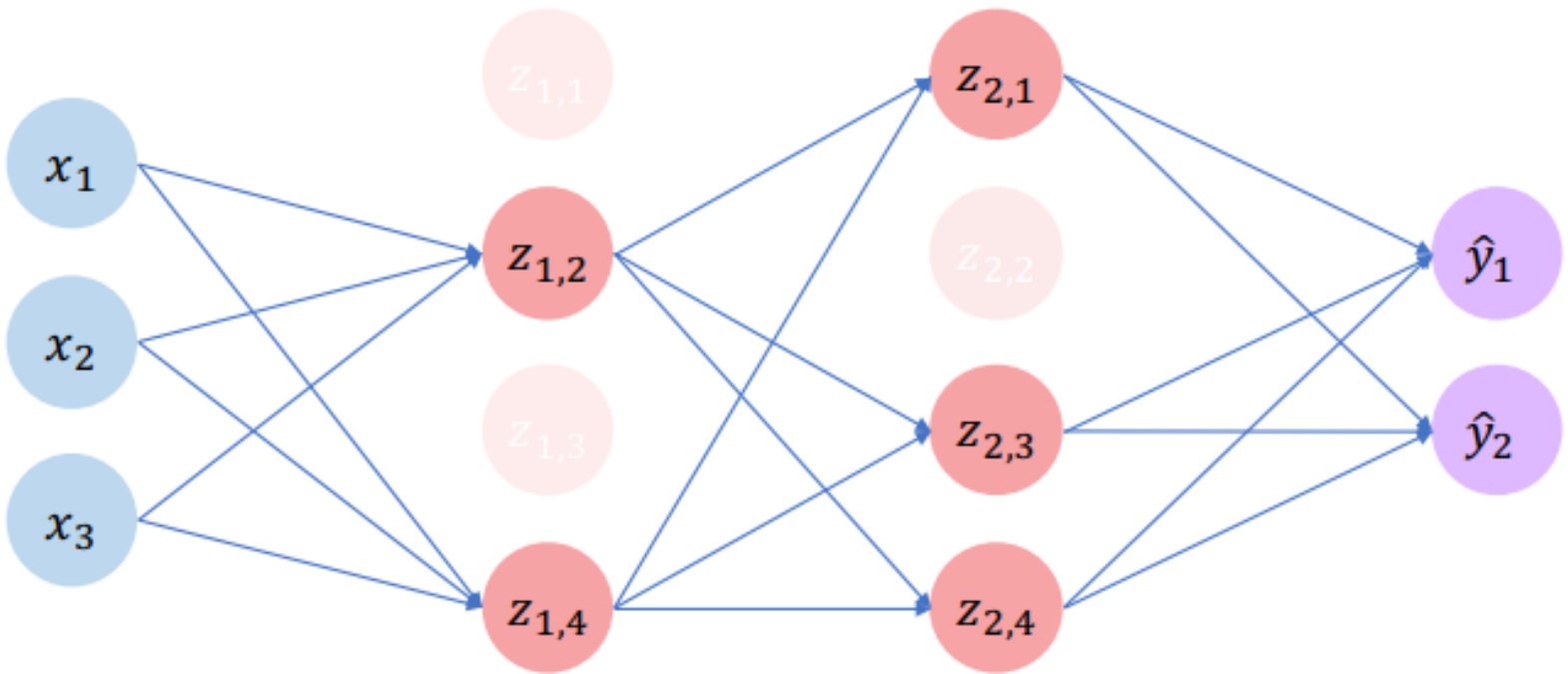
- During training, randomly set some activations to 0



Regularization 1: Dropout

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

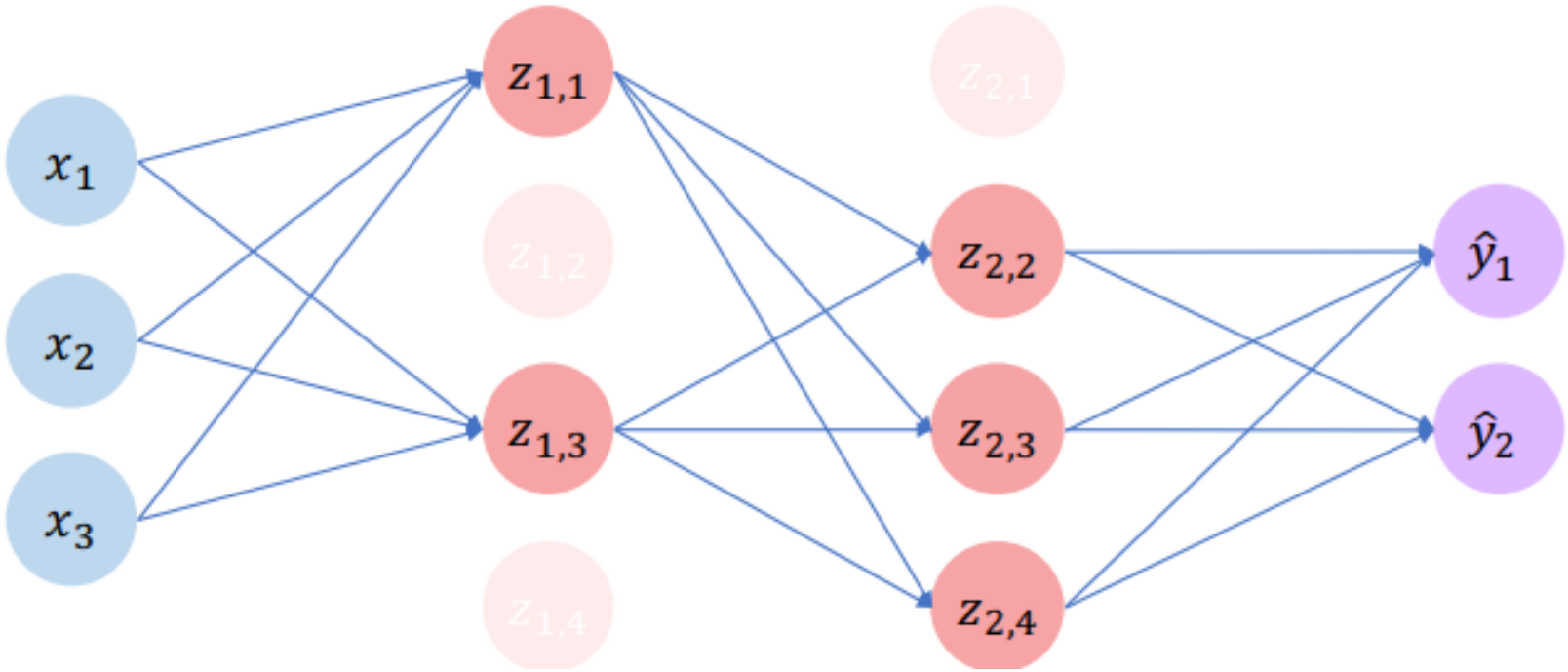
```
tf.nn.dropout(hiddenLayer, p=0.5)
```



Regularization 1: Dropout

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

```
tf.nn.dropout(hiddenLayer, p=0.5)
```



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Types of Deep Neural Networks

- **Feedforward Neural Networks (Multilayer Perceptrons - MLPs):** MLPs are the foundation for more complex structures and are widely used for classification and regression tasks.
- **Convolutional Neural Networks (CNNs):** Ideal for image classification.
- **Recurrent Neural Networks (RNNs):** RNNs are tailored for sequential data like time series or natural language.
  - **Long Short-Term Memory (LSTM):** Uses gates to manage the flow of information.
  - **Gated Recurrent Unit (GRU):** Offers similar benefits to LSTMs with a more streamlined structure and fewer parameters.
- **Autoencoders:** These networks excel at unsupervised learning by compressing input data into a lower-dimensional representation (encoding) and then reconstructing the output from this encoding. Variants include:
  - **Denoising Autoencoders:** Learn to recover original input from noisy versions.
  - **Variational Autoencoders (VAEs):** Introduce a probabilistic approach, enabling the generation of new data samples that resemble the training set.

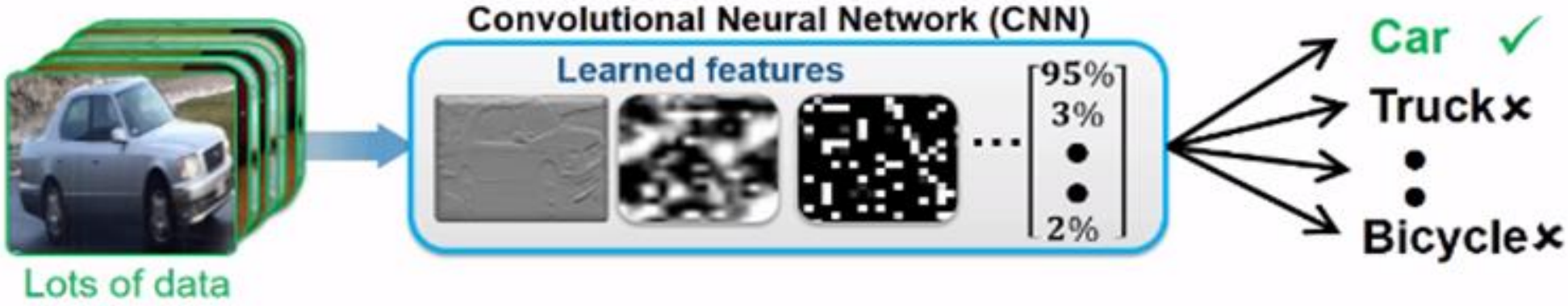
Types of Deep Neural Networks

- **Generative Adversarial Networks (GANs):** A generator tries to produce realistic data (like images), while a discriminator evaluates their authenticity.
- **Deep Belief Networks (DBNs) and Deep Boltzmann Machines (DBMs):** These networks rely on layers of probabilistic models (specifically Restricted Boltzmann Machines) and were among the early architectures that helped pave the way for deep learning. DBNs are often used for pretraining deep networks, while DBMs model complex joint distributions through multiple layers of hidden variables.
- **Transformer-Based Networks**
- **Graph Neural Networks (GNNs):** These architectures handle data structured as graphs, learning representations for nodes, edges, or entire graphs.

Convolutional Neural Networks

Approaches for Deep Learning

1. Train a Deep Neural Network from Scratch

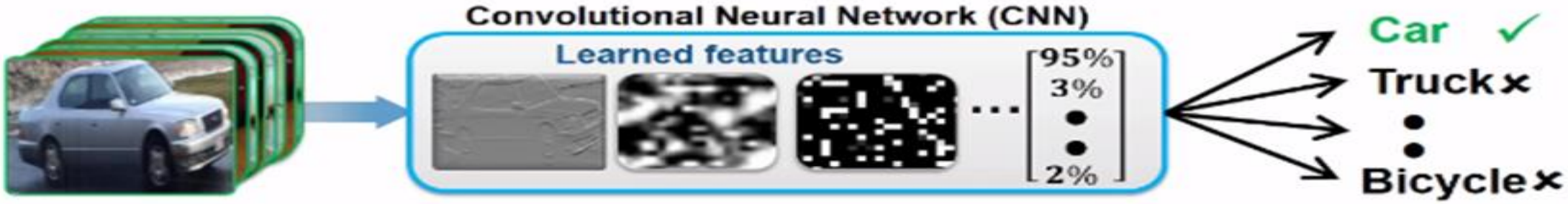


2. Fine-tune a pre-trained model (transfer learning)

ConvNets

Deep Learning Approaches

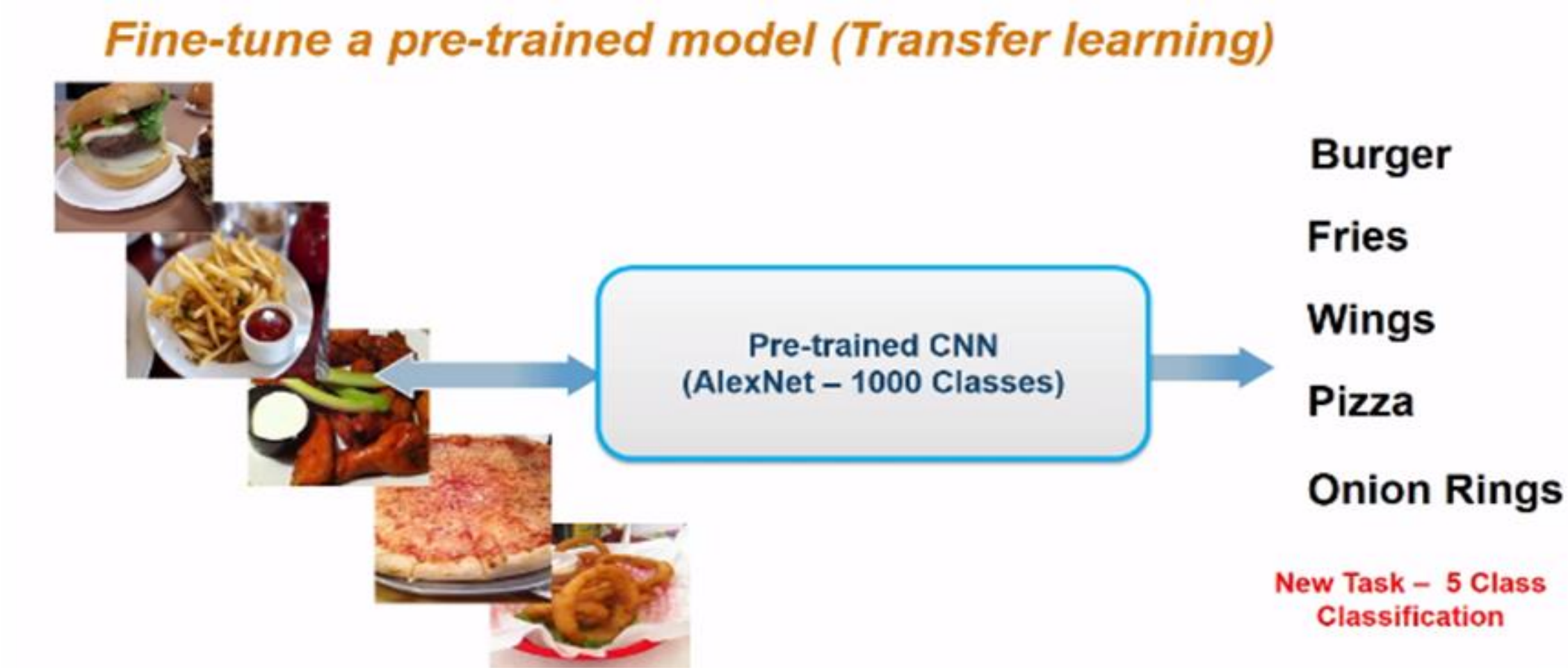
Approach 1: Train a Deep Neural Network from Scratch



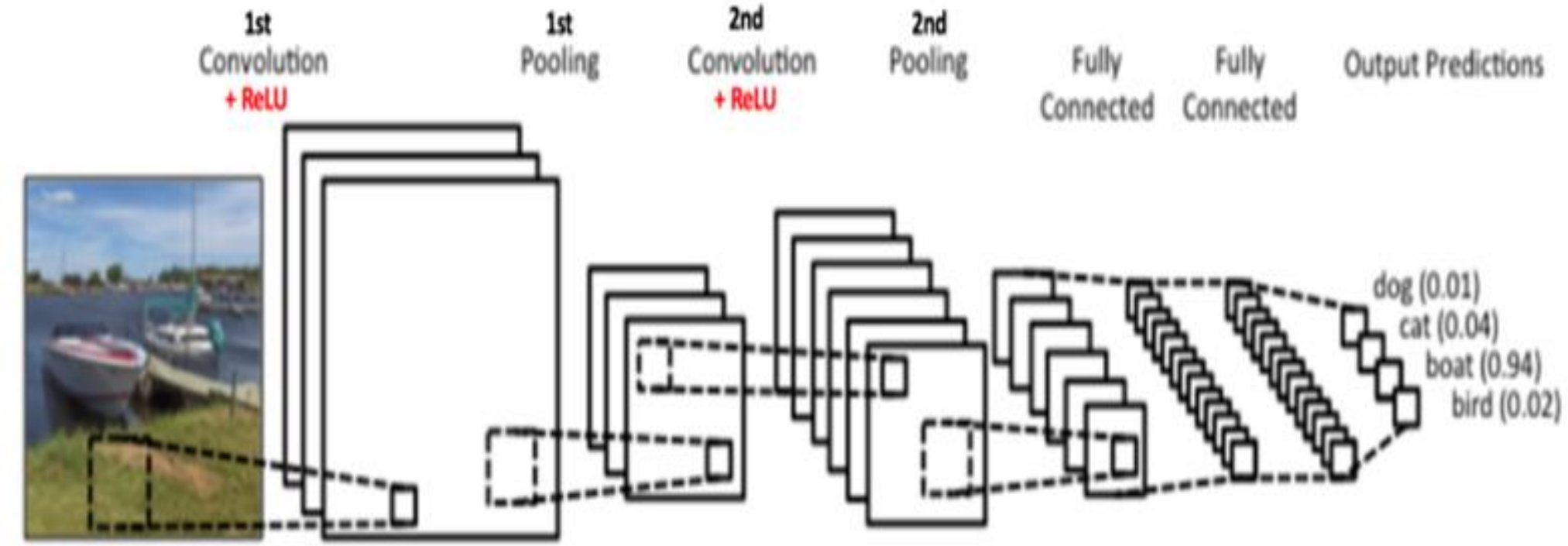
Recommended only when:

Training data	1000s to millions of labeled images
Computation	Compute intensive (requires GPU)
Training Time	Days to Weeks for real problems
Model accuracy	High (can over fit to small datasets)

ConvNets


Fine-tune a pre-trained model (Transfer learning)
Pre-trained CNN
(AlexNet – 1000 Classes)
Burger
Fries
Wings
Pizza
Onion Rings
New Task – 5 Class
Classification

ConvNets


1st
Convolution
+ ReLU
1st
Pooling
2nd
Convolution
+ ReLU
2nd
Pooling
Fully
Connected
Fully
Connected
Output Predictions
dog (0.01)
cat (0.04)
boat (0.94)
bird (0.02)

Convolutional Layer

Convolution

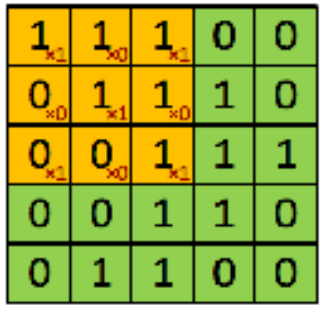
Input Image

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

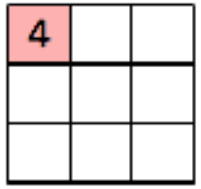
Kernel

1	0	1
0	1	0
1	0	1

In CNN terminology, the 3×3 matrix is called a '**filter**' or 'kernel' or 'feature detector', and the matrix formed by sliding the filter over the image and computing the dot product is called the 'Convolved Feature' or 'Activation Map' or the '**Feature Map**'. It is important to note that filters act as feature detectors from the original input image.



Image



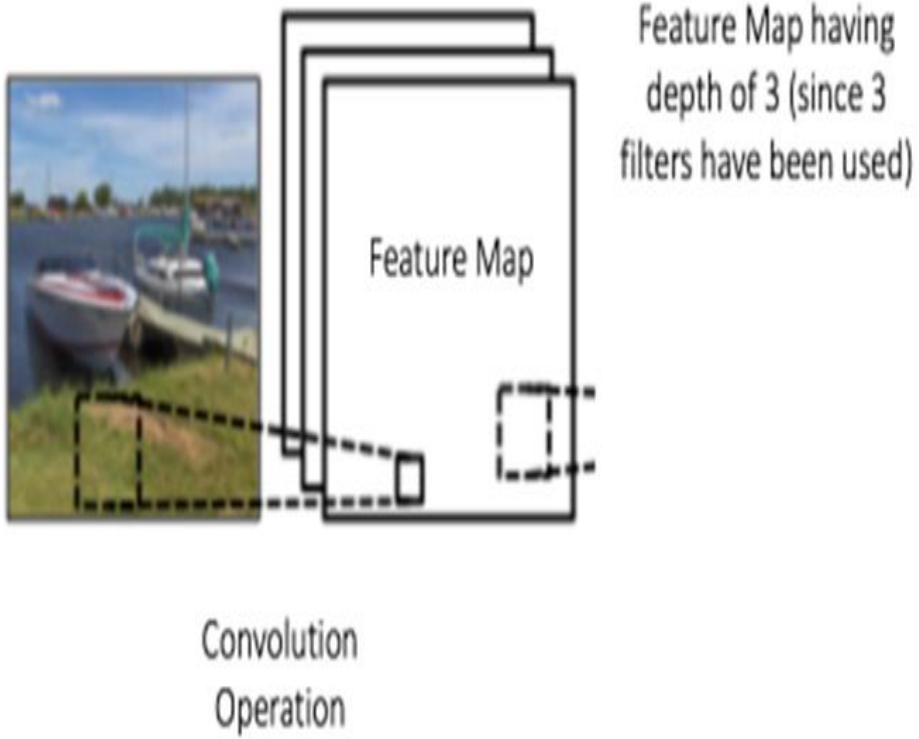
Convolved Feature

Visit https://cs231n.github.io/assets/conv-demo/index.html

Feature Map Parameters

The size of the Feature Map (Convolved Feature) is controlled by three parameters that we need to decide before the convolution step is performed:

•***Depth***: Depth corresponds to the number of filters we use for the convolution operation. In the network shown in Figure, we are performing convolution of the original boat image using three distinct filters, thus producing three different feature maps as shown. You can think of these three feature maps as stacked 2d matrices, so, the 'depth' of the feature map would be three.



Feature Map Parameters

- **Stride**: Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.
- **Zero-padding**: Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution.

Size of Feature Map

Output width=((W-F_w+2*P)/S)+1
Output height=((H-F_h+2*P)/S)+1

Example

- Tensor size or shape: (width = 28, height = 28)
- Convolution filter size (F): (F_width = 5, F_height = 5)
- Padding (P): 0
- Stride (S): 1

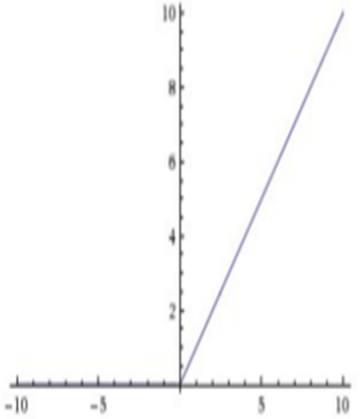
Output width=((28-5+2*0)/1)+1=24
Output height=((28-5+2*0)/1)+1=24
The output dimension will be (24,24)

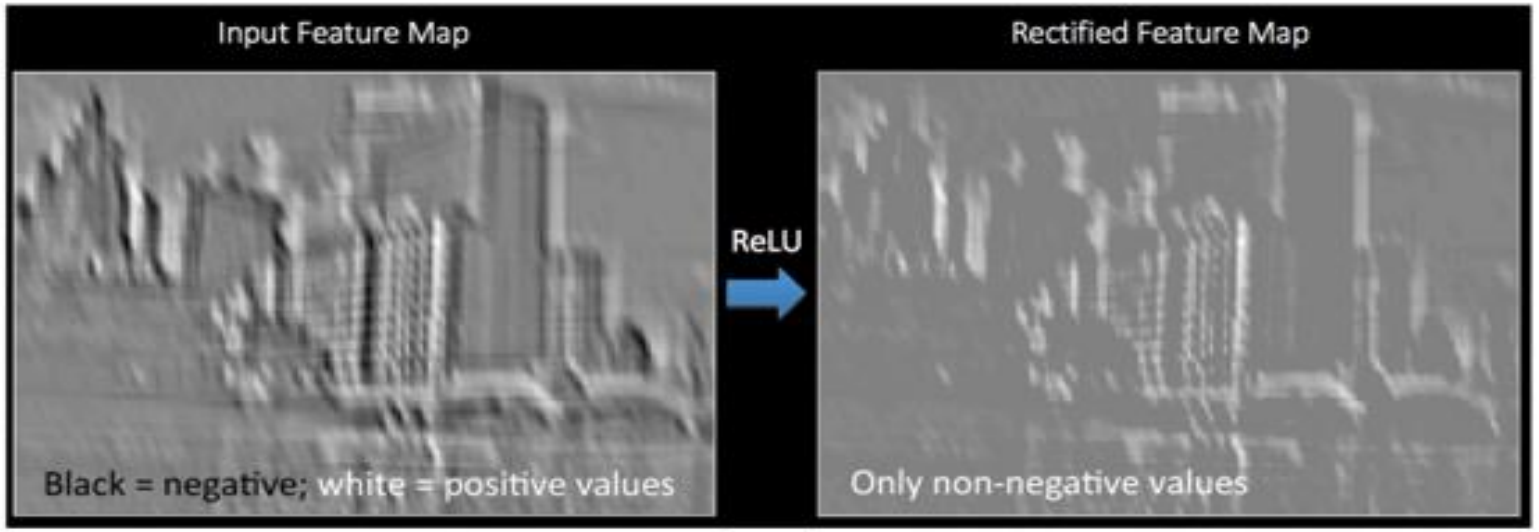
Non Linearity (ReLU)

ReLU stands for Rectified Linear Unit and is a non-linear operation. Its output is given by:

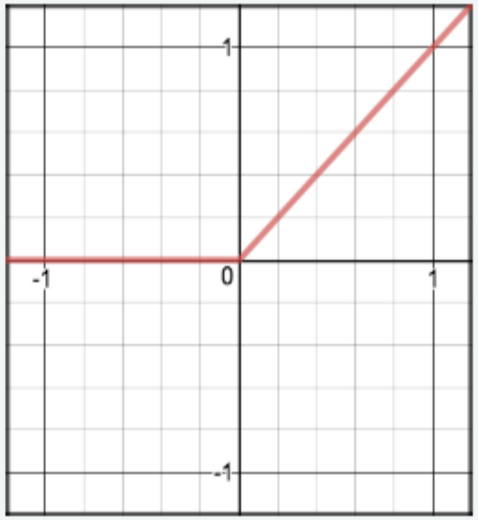
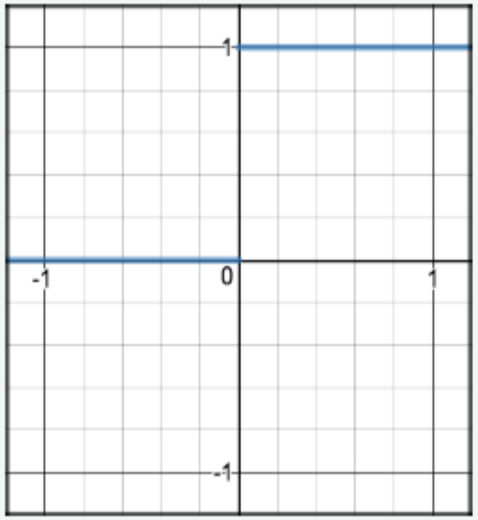
ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU).

Output = Max(zero, Input)





Non Linearity (ReLU)

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ 0 & z <= 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$
	

Pros

- It avoids and rectifies vanishing gradient problem.
- ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons

- One of its limitation is that it should only be used within Hidden layers of a Neural Network Model.
- Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. Simply saying that ReLU could result in Dead Neurons.
- In another words, For activations in the region (x<0) of ReLU, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called dying ReLU problem.
- The range of ReLU is [0, inf). This means it can blow up the activation.

Leaky ReLU

LeakyRelu is a variant of ReLU. Instead of being 0 when z<0, a leaky ReLU allows a small, non-zero, constant gradient α (Normally, α=0.01).

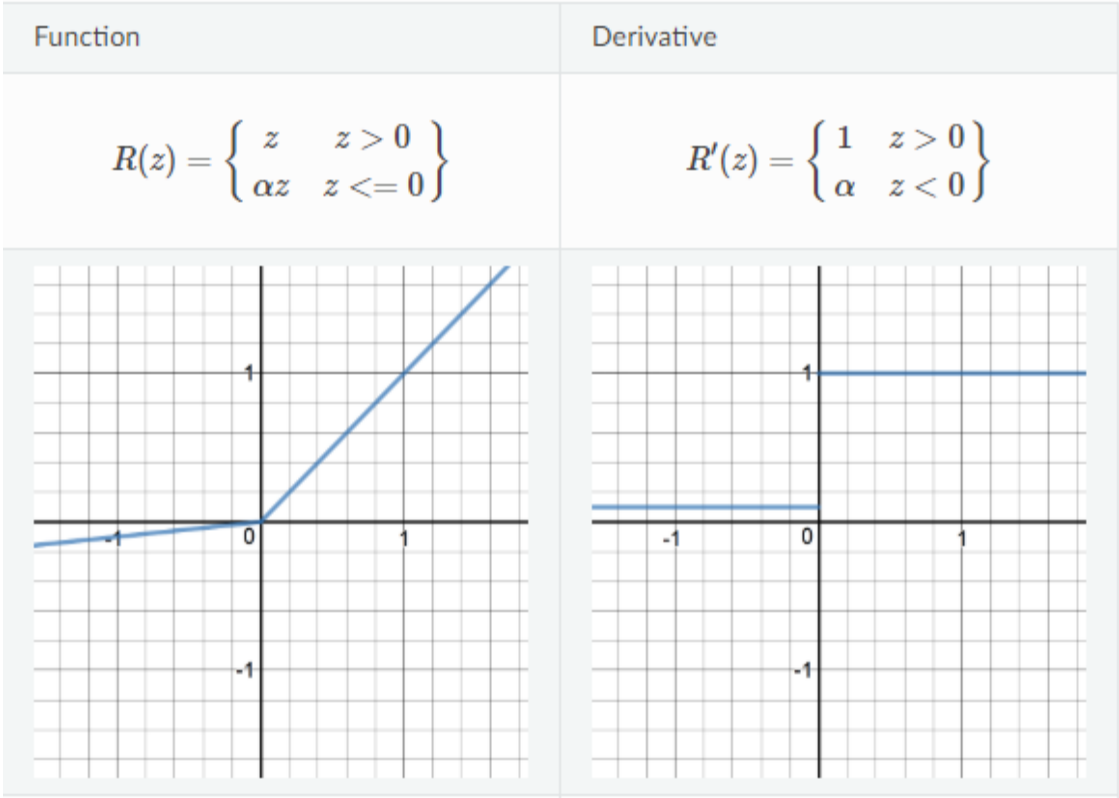
<u>Pros</u>

Leaky ReLUs are one attempt to fix the "dying ReLU" problem by having a small negative slope (of 0.01, or so).

<u>Cons</u>

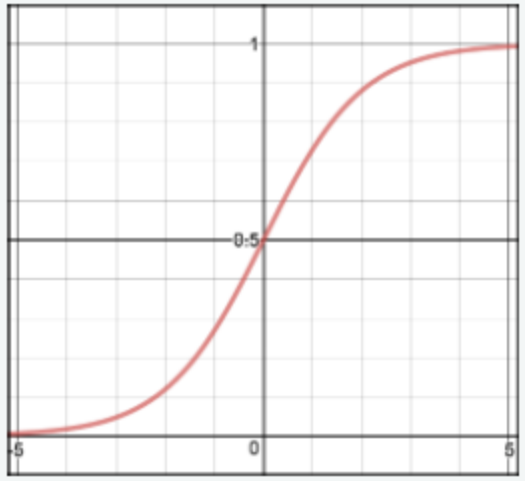
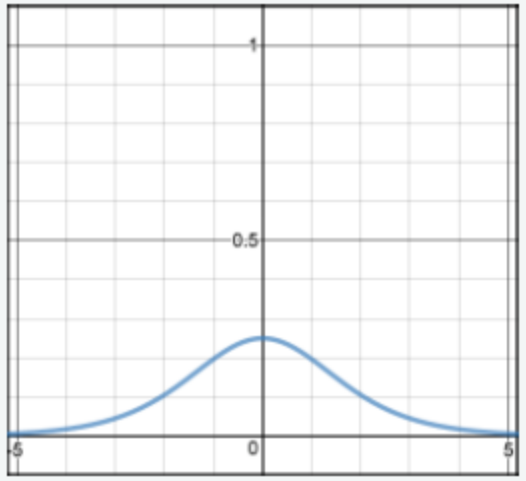
As it possess linearity, it can't be used for the complex Classification.

It lags behind the Sigmoid and Tanh for some of the use cases.



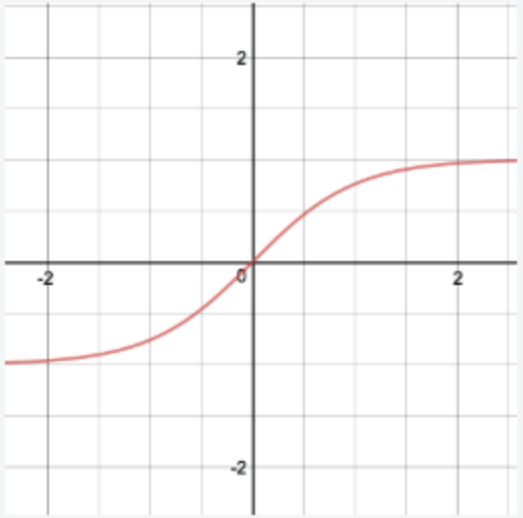
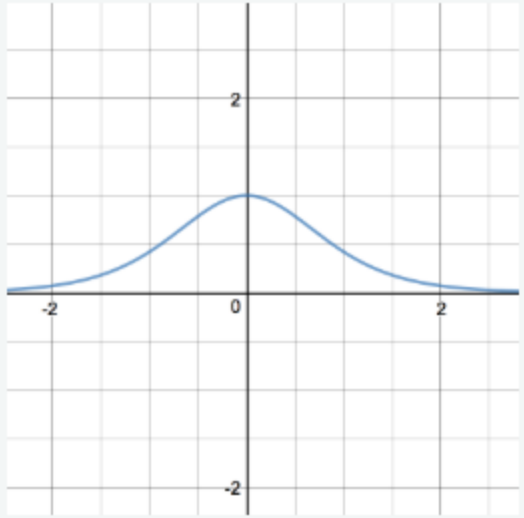
Other Activation Functions - Sigmoid

Sigmoid takes a real value as input and outputs another value between 0 and 1. It's easy to work with and has all the nice properties of activation functions: it's non-linear, continuously differentiable, monotonic, and has a fixed output range.

Function	Derivative
$S(z) = \frac{1}{1+e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$
	

Other Activation Functions - Tanh

Tanh squashes a real-valued number to the range [-1, 1]. It's non-linear. But unlike Sigmoid, its output is zero-centered.

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

Other Activation Functions - Softmax

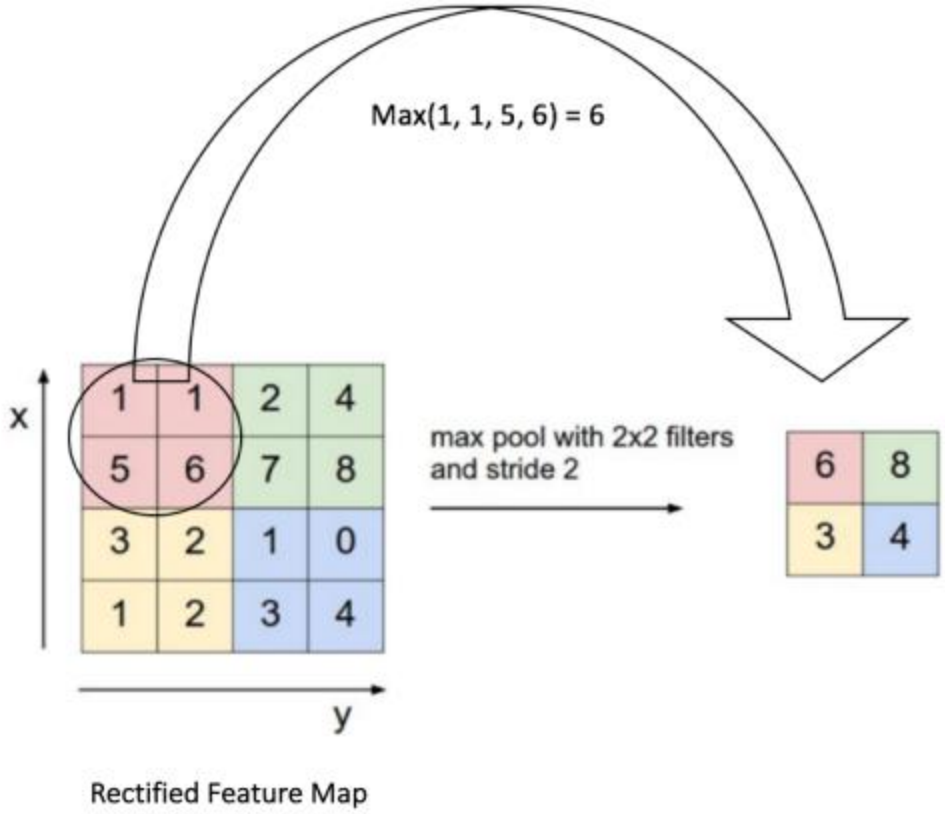
Softmax function calculates the probabilities distribution of the event over ‘n’ different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

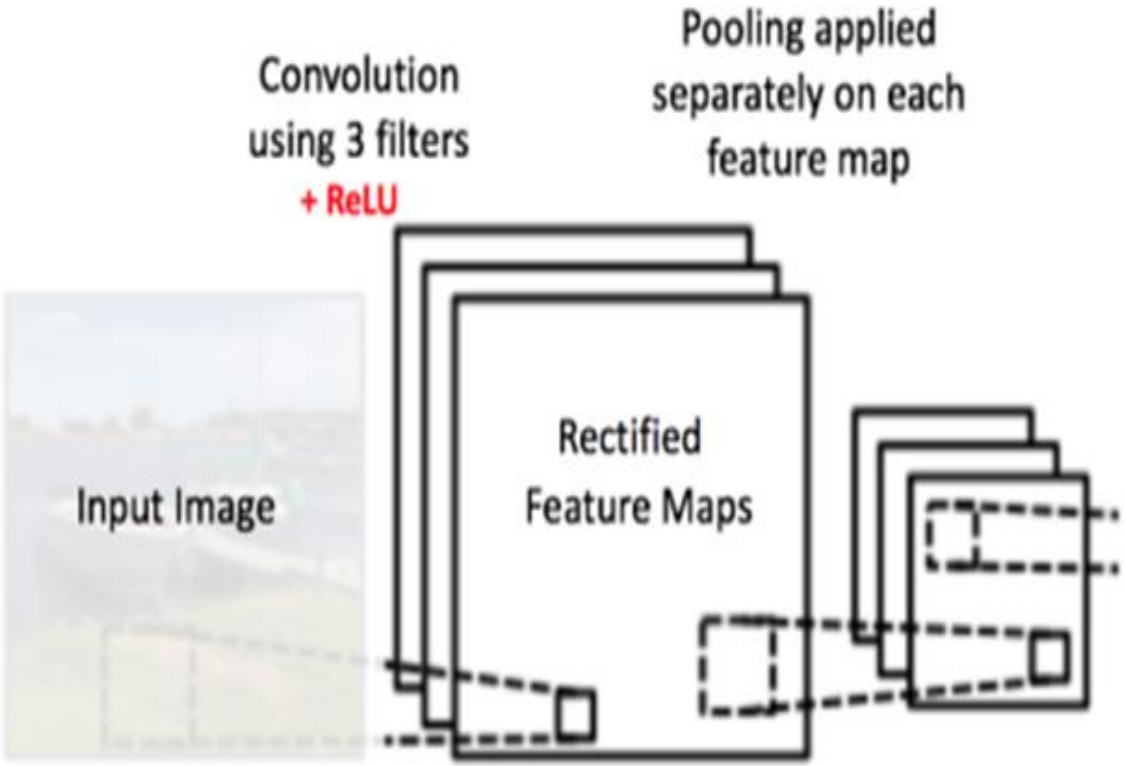
Pooling

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Pooling


Max(1, 1, 5, 6) = 6
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4
Rectified Feature Map


Convolution
using 3 filters
+ ReLU
Pooling applied
separately on each
feature map
Input Image
Rectified
Feature Maps

Pooling

In particular, pooling:

- makes the input representations (feature dimension) smaller and more manageable
- reduces the number of parameters and computations in the network, therefore, controlling overfitting
- makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).
- helps us arrive at an almost scale invariant representation of our image (the exact term is “equivariant”). This is very powerful since we can detect objects in an image no matter where they are located

Pooling

Pooling can be Valid or Same

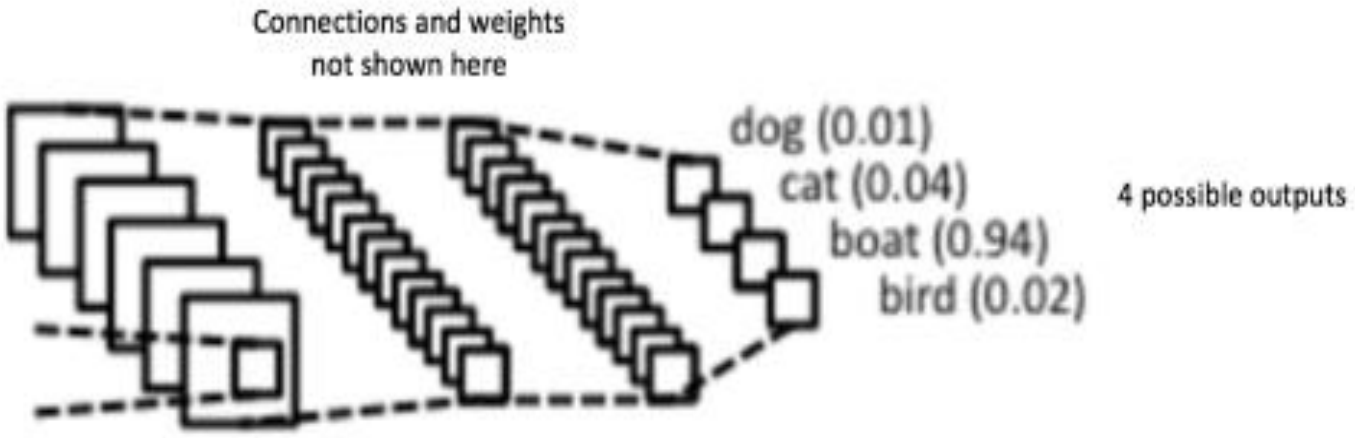
When padding == ”VALID”, the input image is not padded. This means that the filter window always stays inside the input image. This type of padding is called valid because for this padding only the valid and original elements of the input image are considered.
When padding == "VALID", there can be a loss of information. Generally, elements on the right and the bottom of the image tend to be ignored. How many elements are ignored depends on the size of the kernel and the stride.
In this case, the size of the output image <= the size of the input image.

Pooling

When padding == “SAME”, the input is half padded. The padding type is called SAME because the output size is the same as the input size (when stride=1). Using ‘SAME’ ensures that the filter is applied to all the elements of the input. Normally, padding is set to "SAME" while training the model. Output size is mathematically convenient for further computation.

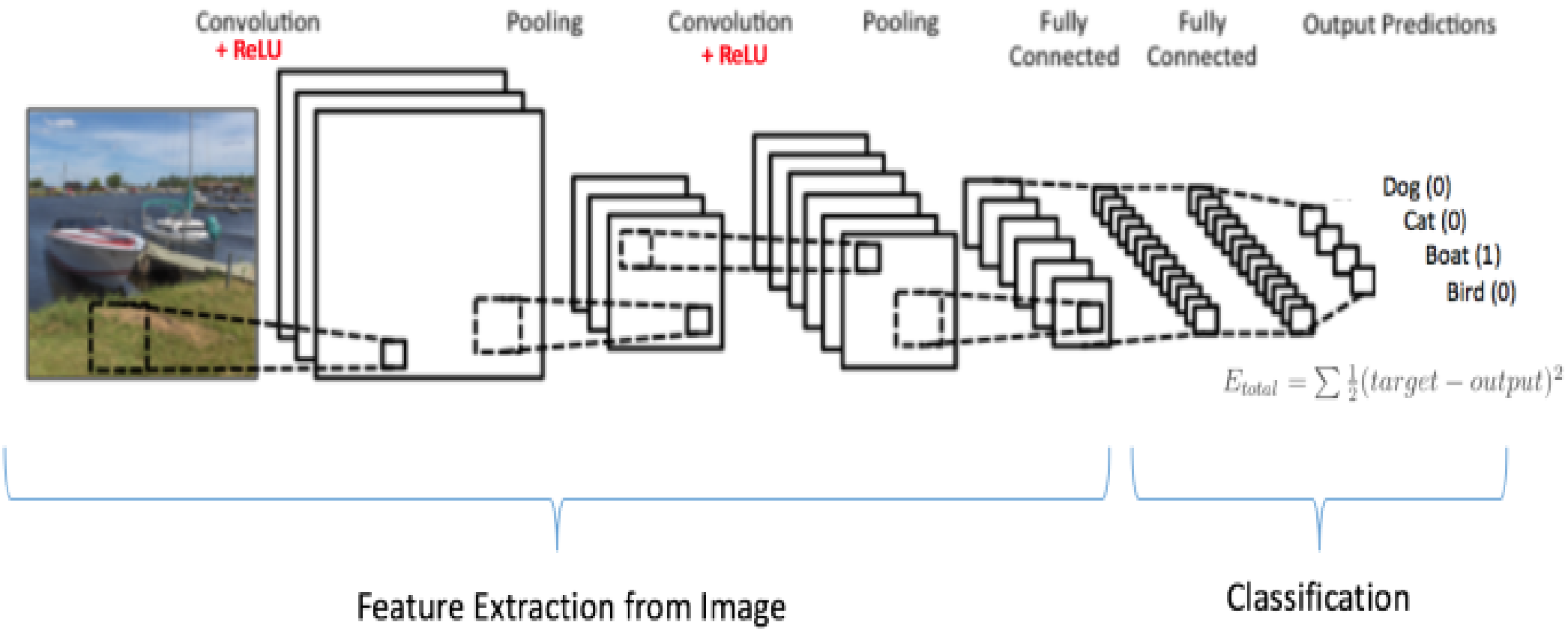
Fully Connected Layer

The Fully Connected layer is a traditional Multi Layer Perceptron that uses a softmax* activation function in the output layer (other classifiers like SVM can also be used). The term “Fully Connected” implies that every neuron in the previous layer is connected to every neuron on the next layer.



***Softmax classifiers give you *probabilities* for each class label.** Softmax classifier is a generalization of the binary form of Logistic Regression

The network


Convolution
+ ReLU
Pooling
Convolution
+ ReLU
Pooling
Fully
Connected
Fully
Connected
Output Predictions
Dog (0)
Cat (0)
Boat (1)
Bird (0)
$E_{total} = \sum \frac{1}{2}(target - output)^2$
Feature Extraction from Image
Classification

Training the network

The overall training process of the Convolution Network may be summarized as below:
Step1: We initialize all filters and parameters / weights with random values
Step2: The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.
Step3: Calculate the total error at the output layer
Step4: Use Backpropagation to calculate the *gradients* of the error with respect to all weights in the network and use *gradient descent* to update all filter values / weights and parameter values to minimize the output error.
Step5: Repeat steps 2-4 with all images in the training set.

Testing the network

When a new (unseen) image is input into the ConvNet, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples). If our training set is large enough, the network will (hopefully) generalize well to new images and classify them into correct categories.

Other layers - Batch Normalization layer

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. It is a process to make neural networks faster, more stable through adding extra layers in a deep neural network and reduce overfitting. The new layer performs the standardizing and normalizing operations on the input of a layer coming from a previous layer.
Importantly, batch normalization works differently during training and during inference.

During training the layer normalizes its output using the mean and standard deviation of the current batch of inputs.

During inference the layer normalizes its output using a moving average of the mean and standard deviation of the batches it has seen during training.

Defining the activation shape

Consider the following example. The input image is 28x28x1

Layer	Number of Filters	Padding
Input Image	-	-
Conv2d(f=3,s=1)	8	Same
MaxPool(f=2,s=2)	-	Valid
Conv2d(f=5,s=1)	16	Valid
MaxPool(f=2,s=2)	-	Valid
Flatten	-	-
Dense	-	-
Dense	-	-
Softmax	-	-

Example taken from https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

Defining the activation shape

The activation shape after every convolution or pooling operation is

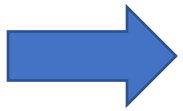
- (ceil(N-f+1)/s,ceil(N-f+1)/s,Number of filters) wherever the padding value is ‘valid’
- (N,N,Number of filters) where the padding used is ‘same’ ,

where ’N’ is input dimensions ‘f’ is filter size and ‘s’ is stride’s length.

Flatten makes the input into a one-dimensional list for input to the dense layers therefore Activation shape is (400,1).

Dense Activation shape (64,1) states 64 hidden units are used in the dense layer.

Layer	Number of Filters	Padding
Input Image	-	-
Conv2d(f=3,s=1)	8	Same
MaxPool(f=2,s=2)	-	Valid
Conv2d(f=5,s=1)	16	Valid
MaxPool(f=2,s=2)	-	Valid
Flatten	-	-
Dense	-	-
Dense	-	-
Softmax	-	-



Layer	Number of Filters	Padding	Activation Shape
Input Image	-	-	(28,28,1)
Conv2d(f=3,s=1)	8	Same	(28,28,8)
MaxPool(f=2,s=2)	-	Valid	(14,14,8)
Conv2d(f=5,s=1)	16	Valid	(10,10,16)
MaxPool(f=2,s=2)	-	Valid	(5,5,16)
Flatten	-	-	(400,1)
Dense	-	-	(120,0)
Dense	-	-	8 4,1)
Softmax	-	-	(10,1)

Defining the activation size

The input layer's shape is (28, 28, 1), the activation size of that layer is 28 * 28 * 1 = 784. The same holds good if you want to calculate the activation shape of any other layer. For example, if we want to calculate the activation size for CONV2 we just multiply (10,10,16) , i.e 10*10*16 = 1600.

Layer	Number of Filters	Padding	Activation Shape	Activation Size
Input Image	-	-	(28,28,1)	784
Conv2d(f=3,s=1)	8	Same	(28,28,8)	6,272
MaxPool(f=2,s=2)	-	Valid	(14,14,8)	1568
Conv2d(f=5,s=1)	16	Valid	(10,10,16)	1,600
MaxPool(f=2,s=2)	-	Valid	(5,5,16)	400
Flatten	-	-	(400,1)	400
Dense	-	-	(120,0)	120
Dense	-	-	(84,1)	84
Softmax	-	-	(10,1)	10

Defining the number of parameters

The number of parameters per layer are computed by the following formula

Parameters= ((F_w*F_h*number of filters in the previous layer+1)*number of filters)

where F_w denotes the filter's width and F_h the filter's height.

Defining number of parameters

The **Input layer** has nothing to learn. It just provides the input image's shape. Therefore, there are no learnable parameters here. Thus, the number of **parameters is 0**.

The Pooling layer has no learnable parameters because all it does is calculate a specific number and no backpropagation learning is involved. Thus, the number of **parameters is 0**.

The Fully Connected Layer (FC) has learnable parameters. Actually, it has the largest number of parameters because every neuron is connected to every other neuron. The number of parameters in the **FC** are:
((current layer neurons c * previous layer neurons p)+1*c).

Defining number of parameters

1.The first layer (**input layer)** has no parameters.
2. The number of parameters in the second layer **CONV1 (filter shape =3*3, stride=1) layer is:**
(((shape of width of filter*shape of height filter*number of filters in the previous layer)+1)*number of filters)) = (((3*3*1)+1)*8) = 80.
3. The third layer (**Pooling layer)** has no parameters.
4. The number of parameters in the fourth layer **CONV2(filter shape =5*5, stride=1) layer is**:
(((shape of width of filter * shape of height filter * number of filters in the previous layer)+1) * number of filters)) = (((5*5*8)+1)*16) = 3216.
5. The fifth layer (Pooling layer) has no parameters.
6. The sixth layer (Flatten layer) has no parameters
7. The seventh layer (Dense) has ((120*400)+120)=48120 parameters.
8. The eighth layer (Dense) has ((84*120)+84)=10164 parameters.
9. The last layer has ((84*10)+10)=850 parameters.
Total Parameters: 62430

Layer	Number of Filters	Padding
Input Image	-	-
Conv2d(f=3,s=1)	8	Same
MaxPool(f=2,s=2)	-	Valid
Conv2d(f=5,s=1)	16	Valid
MaxPool(f=2,s=2)	-	Valid
Flatten	-	-
Dense	-	-
Dense	-	-
Softmax	-	-

Exercise

Given the following architecture, verify the activation shape, the activation size and the number of parameters.

SL.No		Activation Shape	Activation Size	# Parameters
1.	Input Layer:	(32, 32, 3)	3072	0
2.	CONV1 (f=5, s=1)	(28, 28, 8)	6272	608
3.	POOL1	(14, 14, 8)	1568	0
4.	CONV2 (f=5, s=1)	(10, 10, 16)	1600	3216
5.	POOL2	(5, 5, 16)	400	0
6.	FC3	(120, 1)	120	48120
7.	FC4	(84, 1)	84	10164
8.	Softmax	(10, 1)	10	850

Example taken from https://www.coursera.org/learn/convolutional-neural-networks/lecture/uRYL1/cnn-example

Principles/Conventions to build a CNN architecture

The basic principle followed in building a convolutional neural network is to **'keep the feature space wide and shallow in the initial stages of the network, and the make it narrower and deeper towards the end.'** Keeping the above principle in mind we lay down a few conventions to be followed to guide you while building your CNN architecture

1.Always start by using smaller filters is to collect as much local information as possible, and then gradually increase the filter width to reduce the generated feature space width to represent more global, high-level and representative information

2. Following the principle, the number of channels should be low in the beginning such that it detects low-level features which are combined to form many complex shapes (by increasing the number of channels) which help distinguish between classes.

The number of filters is increased to increase the depth of the feature space thus helping in learning more levels of global abstract structures. One more utility of making the feature space deeper and narrower is to shrink the feature space for input to the dense networks.

Principles/Conventions to build a CNN architecture

By convention the number of channels generally increase or stay the same while we progress through layers in our convolutional neural net architecture

3. General filter sizes used are 3x3, 5x5 and 7x7 for the convolutional layer for a moderate or small-sized images and for Max-Pooling parameters we use 2x2 or 3x3 filter sizes with a stride of 2. Larger filter sizes and strides may be used to shrink a large image to a moderate size and then go further with the convention stated.

4. Try using padding = same when you feel the border's of the image might be important or just to help elongate your network architecture as padding keeps the dimensions same even after the convolution operation and therefore you can perform more convolutions without shrinking size.

5. Keep adding layers until you over-fit. As once we achieved a considerable accuracy in our validation set we can use regularization components like l1/l2 regularization, dropout, batch norm, data augmentation etc. to reduce over-fitting

6. Always use classic networks like LeNet, AlexNet, VGG-16, VGG-19 etc. as an inspiration i.e. the trend in the layers Conv-Pool-Conv-Pool or Conv-Conv-Pool-Conv-Conv-Pool or the trend in the Number of channels 32–64–128 or 32–32-64–64 or trend in filter sizes, Max-pooling parameters etc.

Some Network Architectures

LeNet (1990s): Already covered here.

1990s to 2012: In the years from late 1990s to early 2010s convolutional neural network were in incubation. As more and more data and computing power became available, tasks that convolutional neural networks could tackle became more and more interesting.

AlexNet (2012) – In 2012, Alex Krizhevsky (and others) released AlexNet which was a deeper and much wider version of the LeNet and won by a large margin the difficult ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It was a significant breakthrough with respect to the previous approaches and the current widespread application of CNNs can be attributed to this work.

ZF Net (2013) – The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters.

GoogLeNet (2014) – The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al.from Google. Its main contribution was the development of an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M).

Some Network Architectures

VGGNet (2014) – The runner-up in ILSVRC 2014 was the network that became known as the VGGNet. Its main contribution was in showing that the depth of the network (number of layers) is a critical component for good performance.

ResNets (2015) – Residual Network developed by Kaiming He (and others) was the winner of ILSVRC 2015. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 2016).

DenseNet (August 2016) – Recently published by Gao Huang (and others), the Densely Connected Convolutional Network has each layer directly connected to every other layer in a feed-forward fashion. The DenseNet has been shown to obtain significant improvements over previous state-of-the-art architectures on five highly competitive object recognition benchmark tasks.

Some Network Architectures - Parameters

Network	Number of Parameters
LeNet-5	60000
AlexNet	60 million
VGG-16	138 million
GoogleNet	5 million (V1) - 23 million (V3)
ResNet50	25 million
DenseNet-190	40 million
KarNet	16 million

Transfer learning

Transfer learning is about leveraging feature representations from a pre-trained model, so you don’t have to train a new model from scratch.

The pre-trained models are usually trained on massive datasets that are a standard benchmark in the computer vision frontier. The weights obtained from the models can be reused in other computer vision tasks.

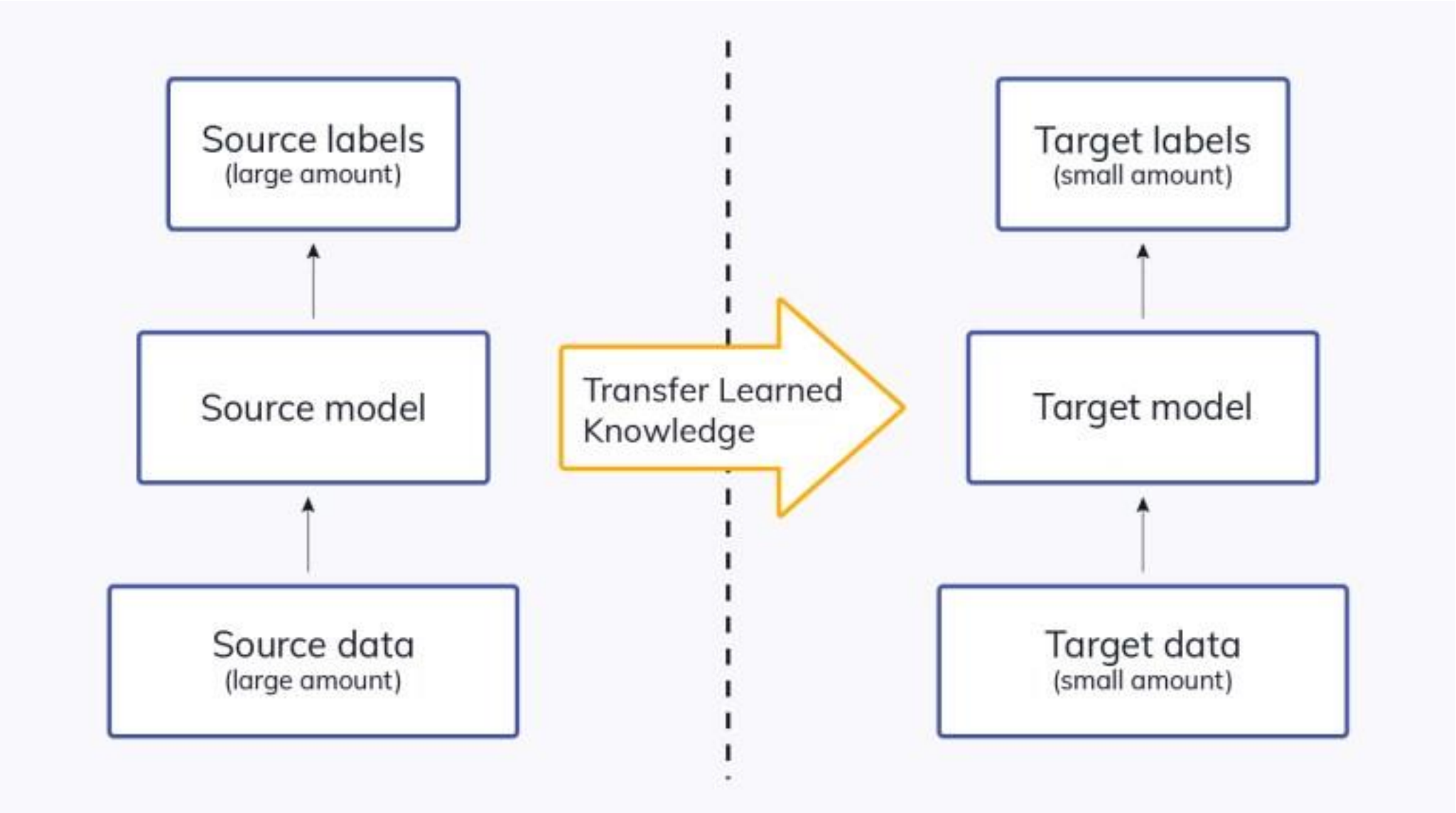
Transfer learning

These models can be used directly in making predictions on new tasks or integrated into the process of training a new model.
Including the pre-trained models in a new model leads to:

- lower training time
- lower generalization error.

Transfer learning is particularly very useful when you have a **small training dataset**. In this case, you can, for example, use the weights from the pre-trained models to **initialize the weights** of the new model.
Transfer learning can also be applied to natural language processing problems.

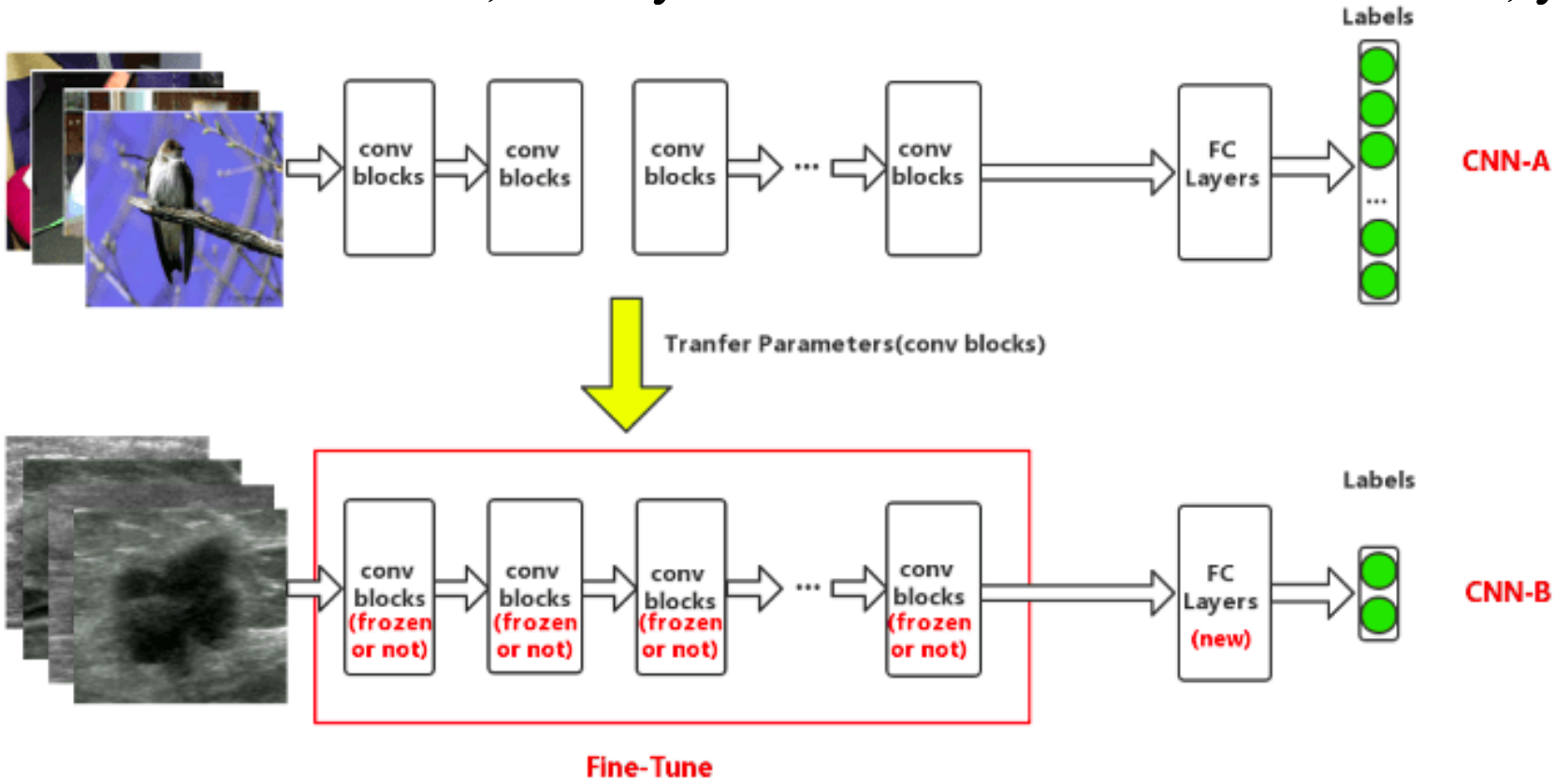
Transfer learning



Source : https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras

Transfer learning – Whats next ?

Fine-tuning is an optional step in transfer learning. Fine-tuning will usually improve the performance of the model. However, since you have to retrain the entire model, you'll likely overfit.



Source : https://static-01.hindawi.com/articles/bmri/volume-2018/4605191/figures/4605191.fig.002.svgz

Transfer learning – Whats next ?

Overfitting is avoidable. To limit overfitting:

- Retrain the model or part of it using a **low learning rate**. This is important because it prevents significant updates to the gradient. These updates result in poor performance.
- Use a callback to stop the training process when the model has stopped improving is also helpful.
- Use a dropout layer

Transfer learning – A case study

Assume you have 100 images of cats and 100 dogs and want to build a model to classify the images. How would you train a model using this small dataset? You can train your model from scratch, but it will most likely overfit horribly.

In general, you want to want to use transfer learning because:

- **training models with high accuracy requires a lot of data**. For example, the ImageNet dataset contains over 1 million images. In the real world, you are unlikely to have such a large dataset.
- assuming that you had that kind of dataset, you might still **not have the resources required to train a model** on such a large dataset. Hence transfer learning makes a lot of sense if you don't have the compute resources needed to train models on huge datasets.
- even if you had the compute resources at your disposal, you still have to **wait for days or weeks to train such a model**. Therefore, using a pre-trained model will save you precious time.

When does transfer learning not work?

Transfer learning will not work when the high-level features learned by the bottom layers are not sufficient to differentiate the classes in your problem.

For example, a pre-trained model may be very good at identifying a door but not whether a door is closed or open. In this case, you can use the low-level features (of the pre-trained network) instead of the high-level features. In this case, you will have to retrain more layers of the model or use features from earlier layers.

When datasets are not similar, features transfer poorly. This paper investigates the similarity of datasets in more detail. That said, as shown in the paper, initializing the network with pre-trained weights results in better performance than using random weights.

Transfer learning - Implementation

You can implement transfer learning in these five general steps

Transfer learning - Implementation

1. Obtain the pre-trained model

- Keras pre-trained models .
- TensorFlow Hub

For text classification problems visit Glove , Google's Word2vec , Hugging Face

Transfer learning - Implementation

2. Create a base model

The first step is to instantiate the **base model** using one of the architectures such as ResNet or Xception. You can also optionally download the **pre-trained weights**.

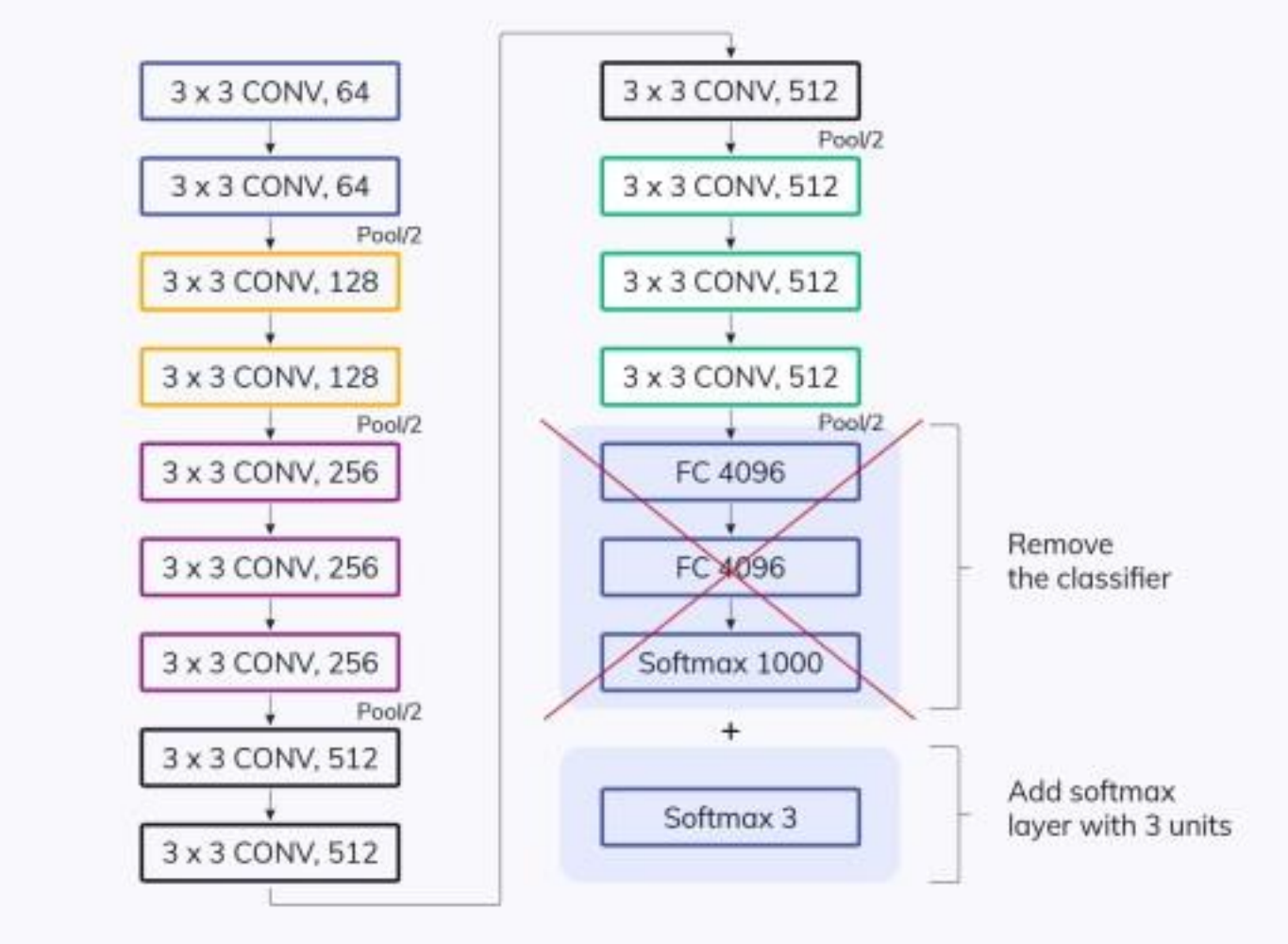
If you don't download the weights, you will have to use the architecture to train your model from scratch.

Recall that the base model will usually have more units in the final output layer than you require.

When creating the base model, you, therefore, have to remove the final output layer. Later on, you will add a final output layer that is compatible with your problem.

Transfer learning - Implementation

2. Create a base model



Transfer learning - Implementation

3. Freeze layers

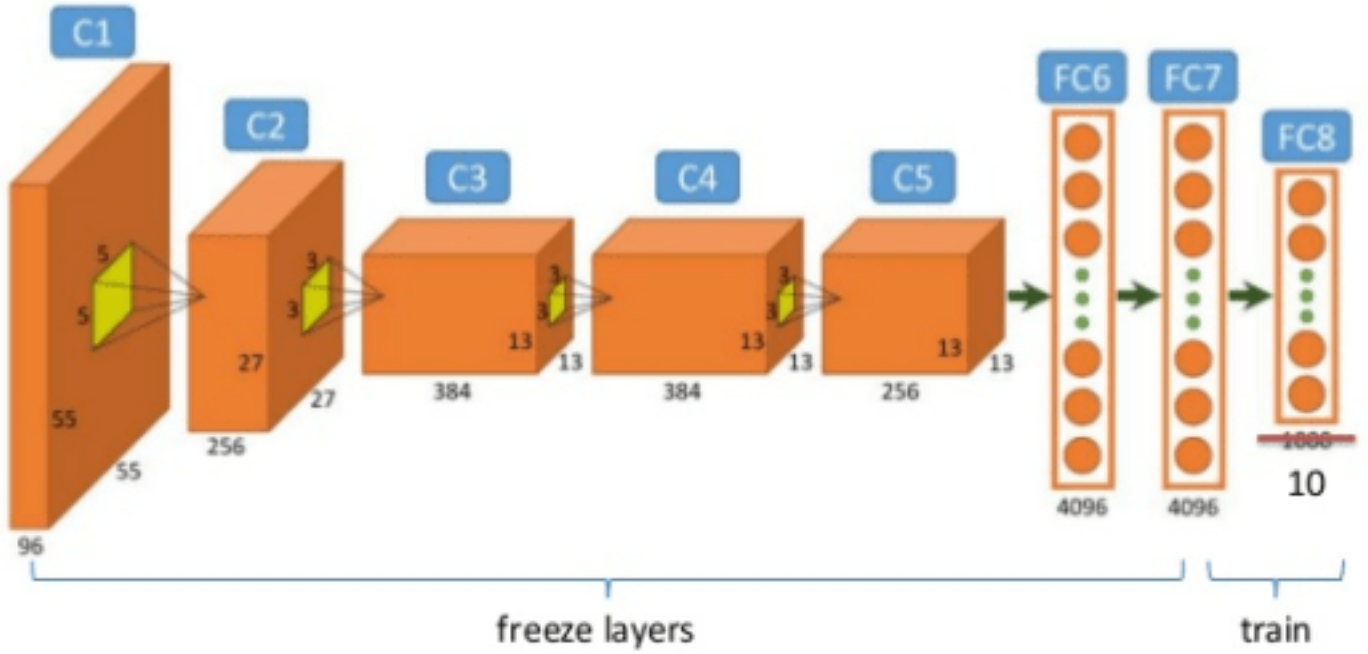
Freezing the layers from the pre-trained model is vital. This is because you don't want the **weights in those layers to be re-initialized**.

If they are, then you will lose all the learning that has already taken place. This will be no different from training the model from scratch.

base_model.trainable = **False**

Transfer learning - Implementation

3. Freeze layers



Source: https://image.slidesharecdn.com/practicaldeeplearning-160329181459/95/practical-deep-learning-16-638.jpg?cb=1459275348

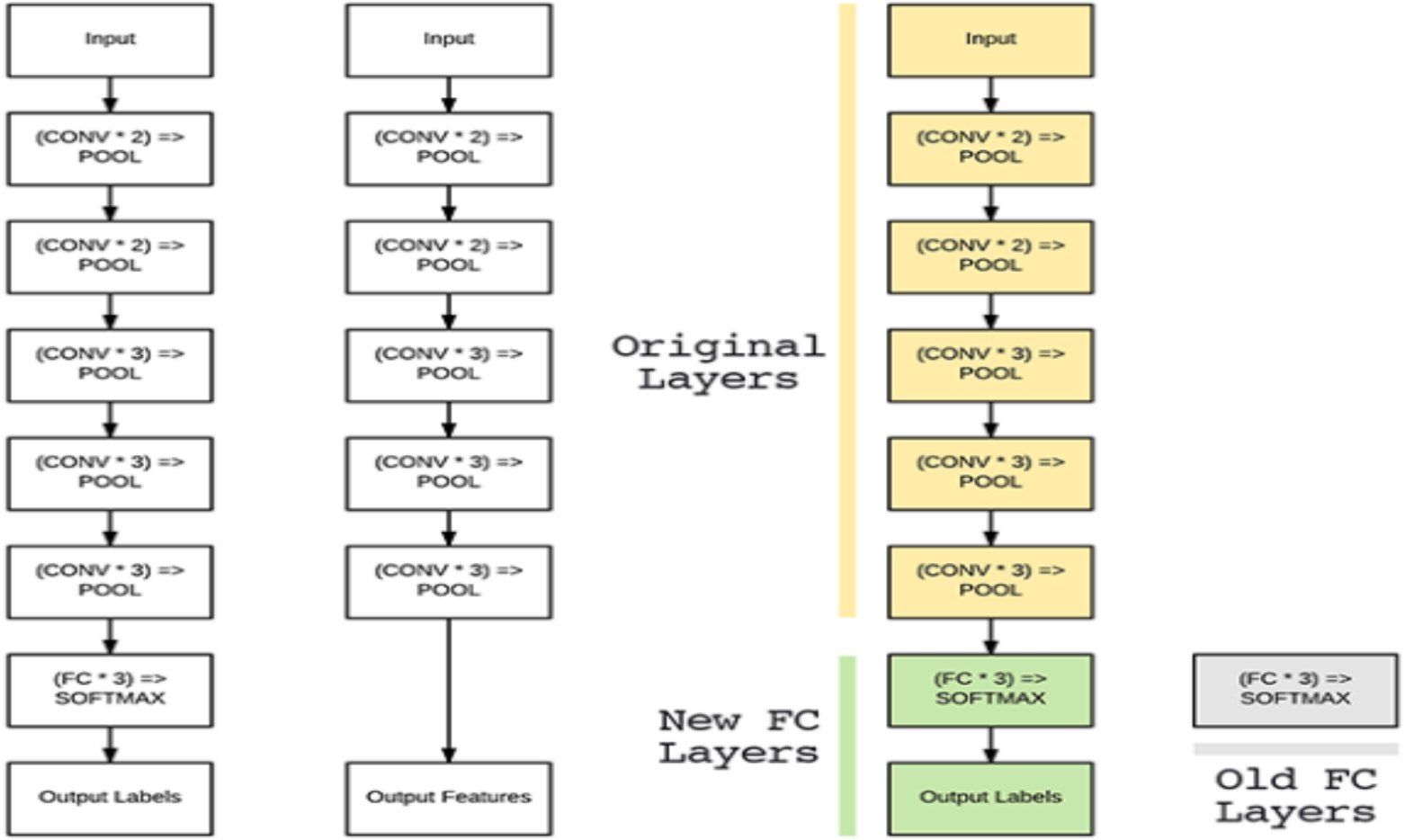
Transfer learning - Implementation

4. Add new trainable layers - Train the new layers

The next step is to **add new trainable layers** that will turn old features into predictions on the new dataset. This is important because the pre-trained model is loaded without the final output layer.

Transfer learning - Implementation

4. Add new trainable layers – Train the new layers



Source : https://pyimagesearch.com/wp-content/uploads/2019/06/fine_tuning_keras_network_surgery.png

Transfer learning - Implementation

4. Add new trainable layers – Train the new layers

The pre-trained model's final output will most likely be different from the output that you want for your model.

For example, pre-trained models trained on the ImageNet dataset will output 1000 classes. However, your model might just have two classes. In this case, you have to train the model with a new output layer in place.

Therefore, you will add some new dense layers as you please, but most importantly, a final dense layer with **units corresponding to the number of outputs expected by your model**.

Transfer learning - Implementation

5. Improve the model via fine-tuning

Fine-tuning is done by unfreezing the base model or part of it and training the entire model again on the whole dataset at a very low learning rate.

The low learning rate will increase the performance of the model on the new dataset while preventing overfitting.

The learning rate has to be low because the model is quite large while the dataset is small. This is a recipe for overfitting, hence the low learning rate.

Transfer learning - Implementation

5. Improve the model via fine-tuning

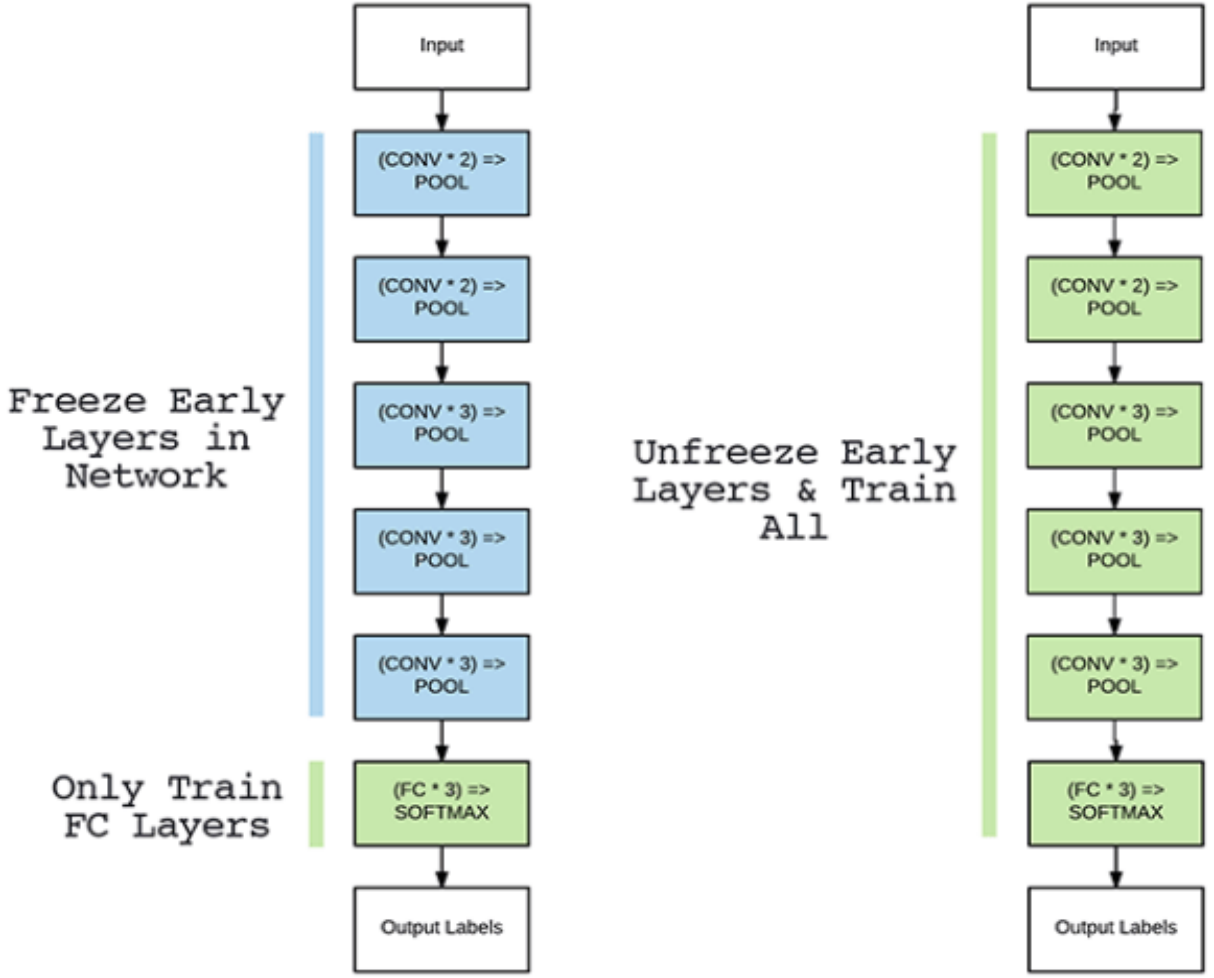
Recompile the model once you have made these changes so that they can take effect.

This is because the behavior of a model is frozen whenever you call the compile function. That means that you have to call the compile function again whenever you want to change the model's behavior.

The next step will be to train the model again while monitoring it via callbacks to ensure it does not overfit.

Transfer learning - Implementation

5. Improve the model via fine-tuning



Source : https://pyimagesearch.com/wp-content/uploads/2019/06/fine_tuning_keras_freeze_unfreeze.png

Thank you

References

http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/

http://introtodeeplearning.com/

http://www.iro.umontreal.ca/~pift6266/H10/notes/deepintro.html

https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

https://www.jeremyjordan.me/convnet-architectures/

https://www.analyticsvidhya.com/

https://www.britannica.com/topic/Deep-Blue

https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

https://towardsdatascience.com/understanding-and-calculating-the-number-of-parameters-in-convolution-neural-networks-cnns-fc88790d530d

https://www.analyticsvidhya.com/blog/2021/10/an-end-to-end-introduction-to-generative-adversarial-networksgans/

A brief introduction to deep learning - Yangyan Li

Deep Learning, Ian Goodfellow and Yoshua Bengio and Aaron Courville https://www.deeplearningbook.org/