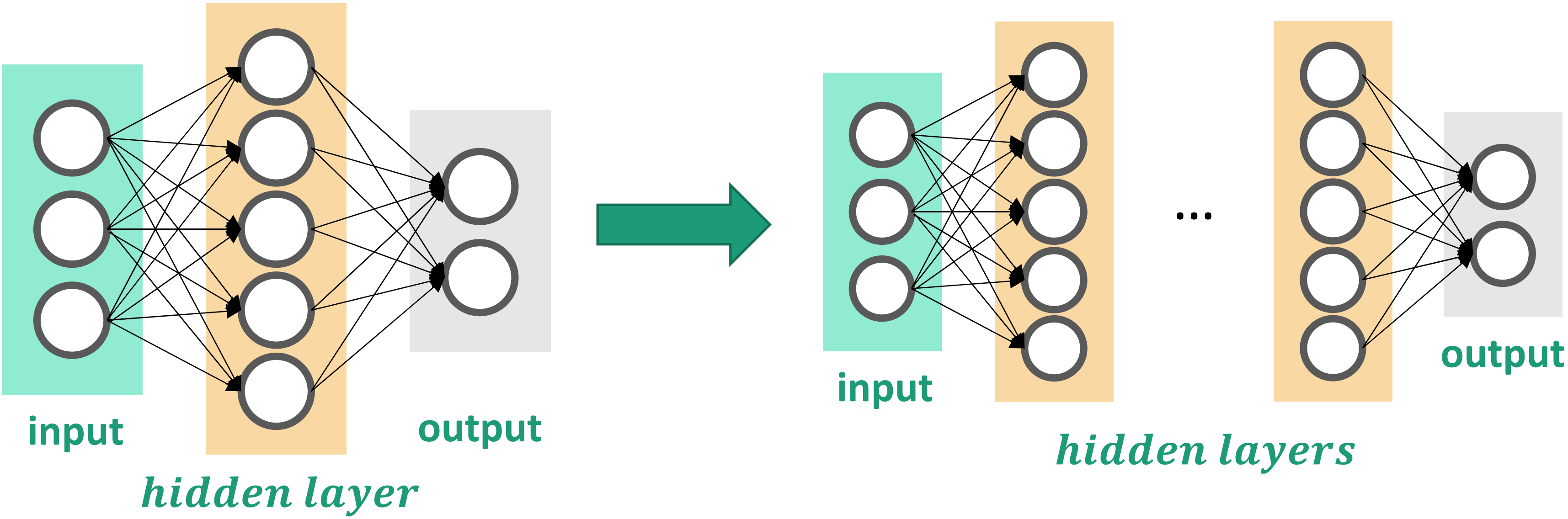
Deep Learning

The "deep" in deep learning

- Deep learning:
  - puts an emphasis on learning successive *layers* of increasingly meaningful representations,
  - How many layers contribute to a model of the data is called the *depth* of the model. (tens or even hundreds)
  - *Also named as layered representations learning* and *hierarchical representations learning*.

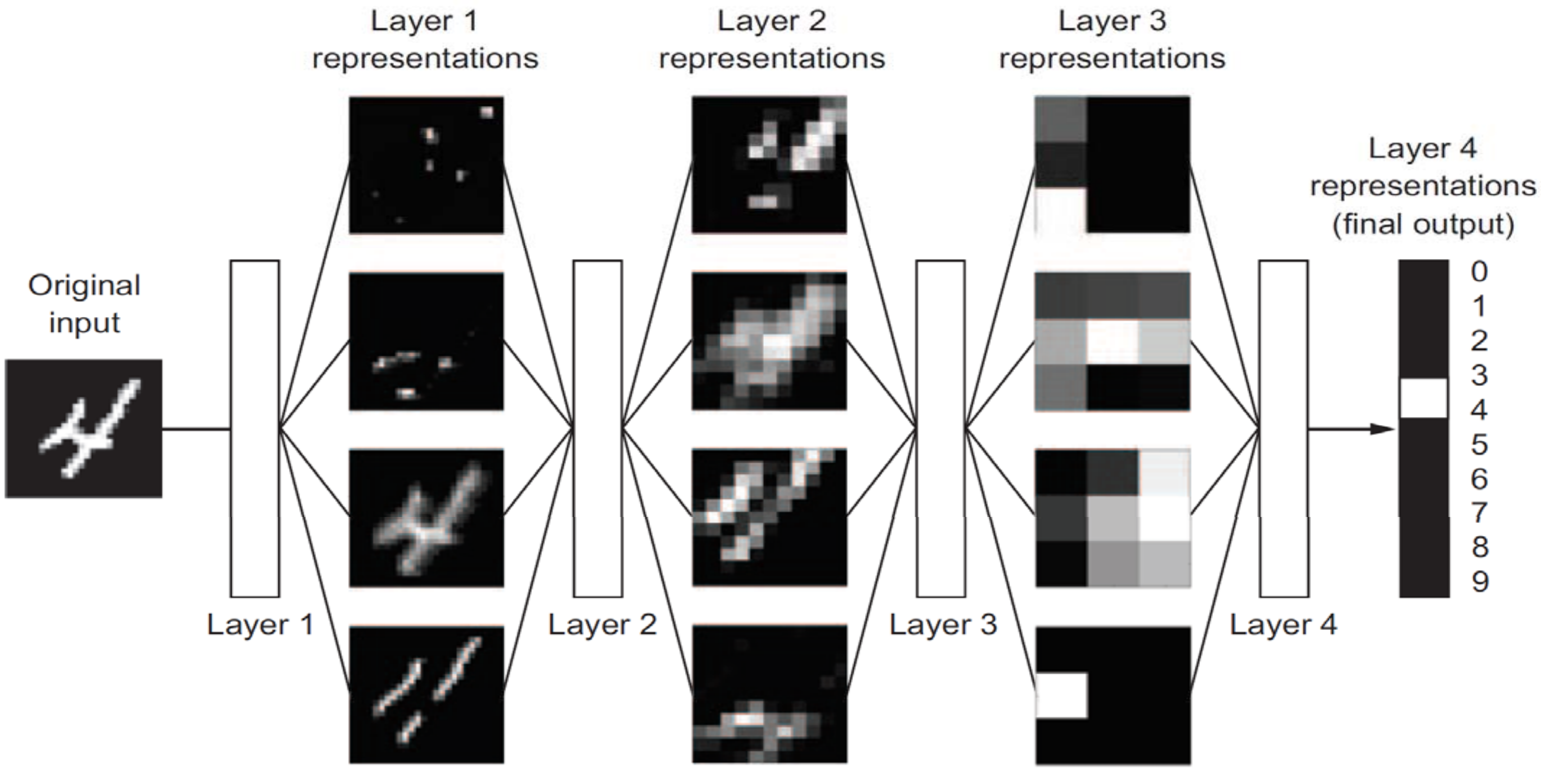
Deep Neural Network

- Deep: more hidden layers

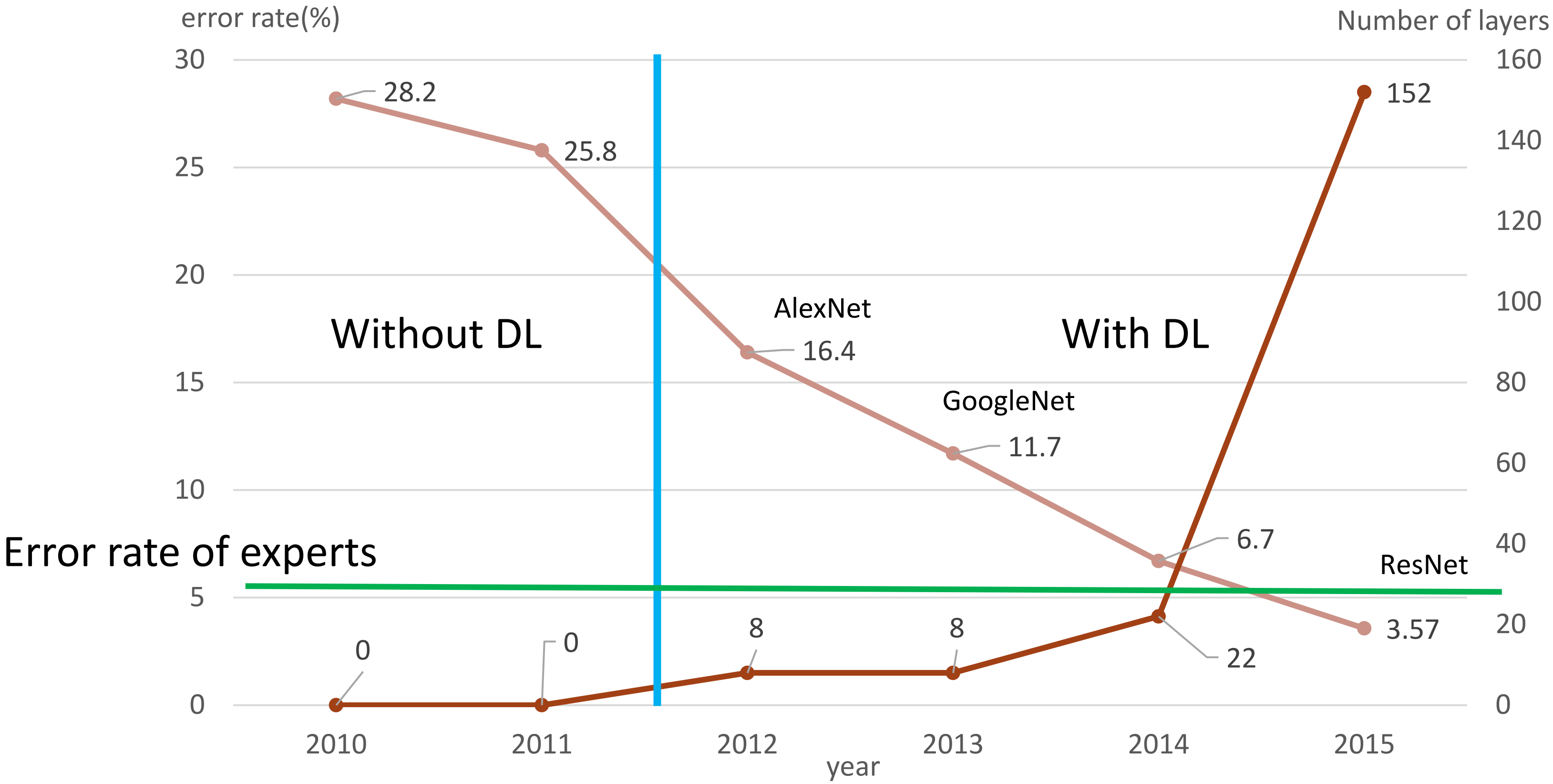


Deep Neural Network

- You can think of a deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly *purified* (that is, useful with regard to some task).



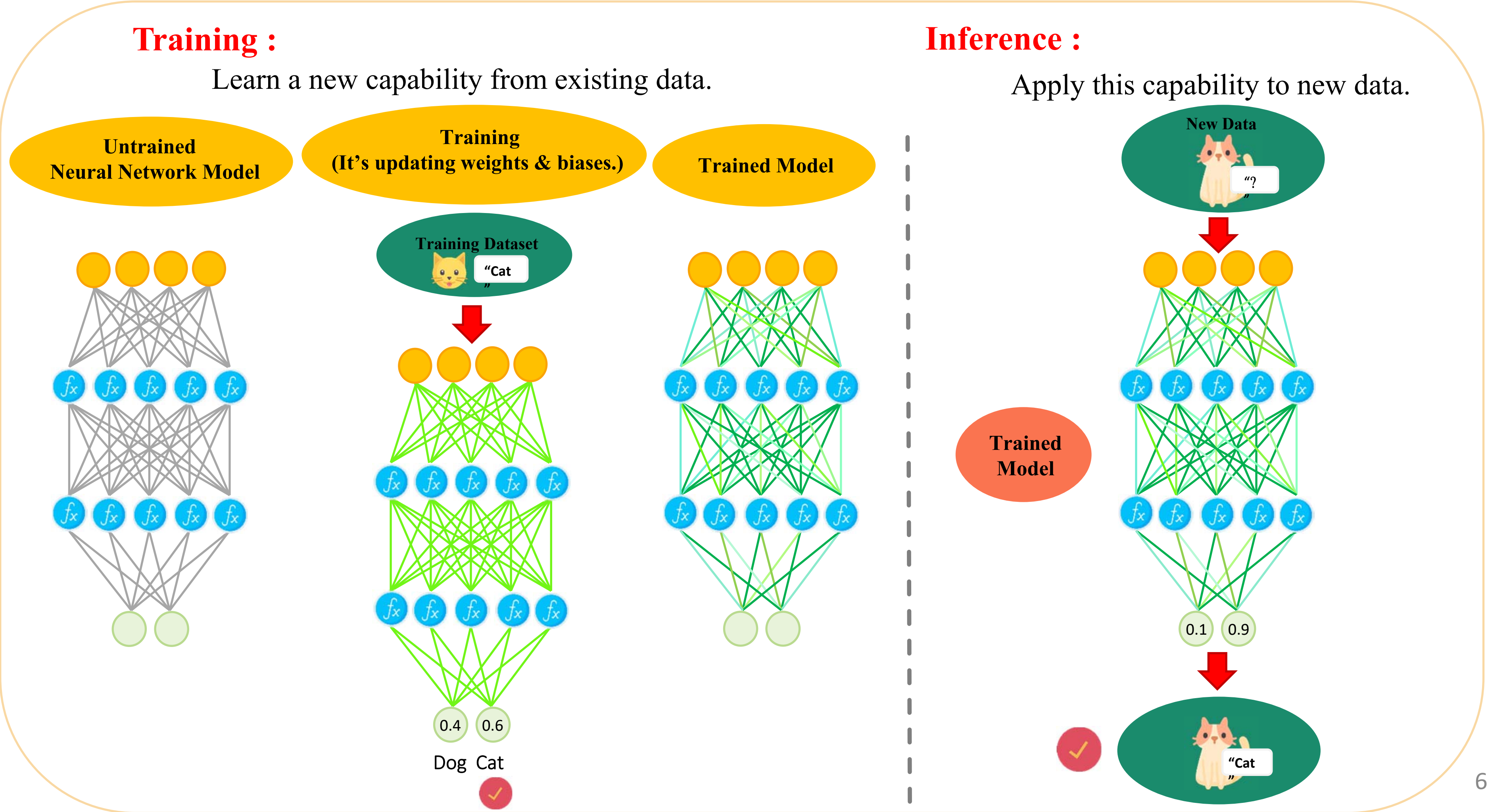
The Deeper the Better?



Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei.
ImageNet Large Scale Visual Recognition Challenge. *IJCV,* 2015.

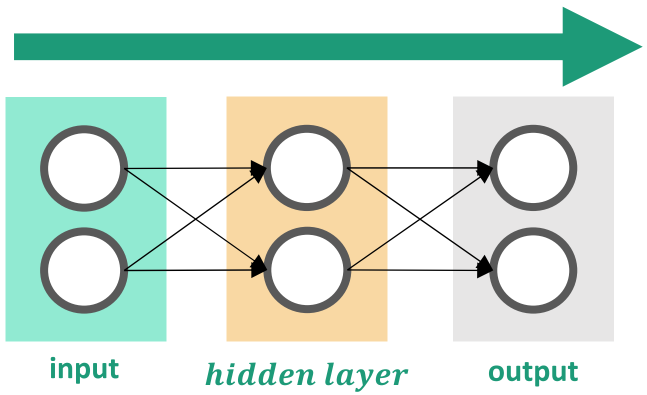
Two Phases of Deep Learning

- There are two phases in deep learning :

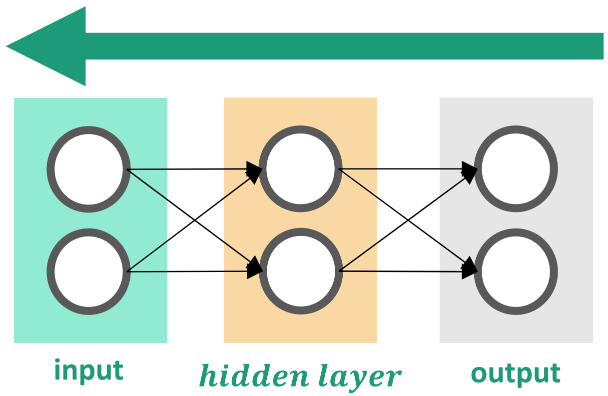


How to Train the NNs

- Input some examples
- Calculate the output
  - **Forward propagation**

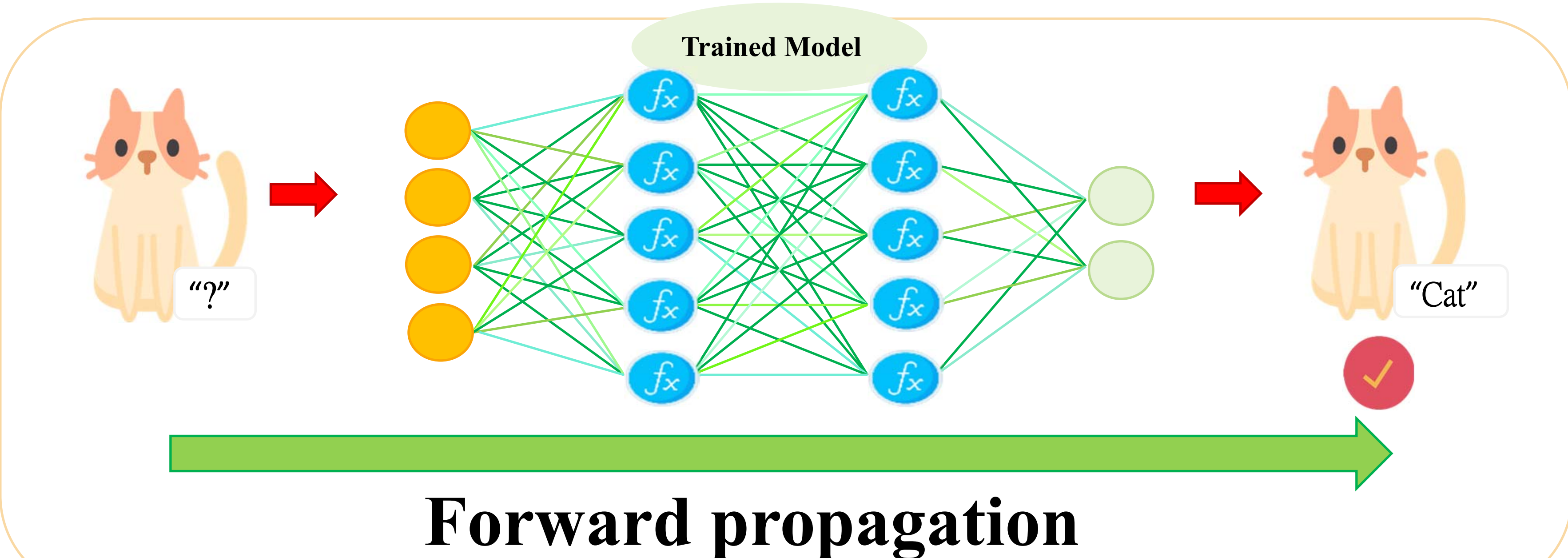


- Measure the errors between the outputs and answers
- Update the weights in NN
  - **Back propagation**



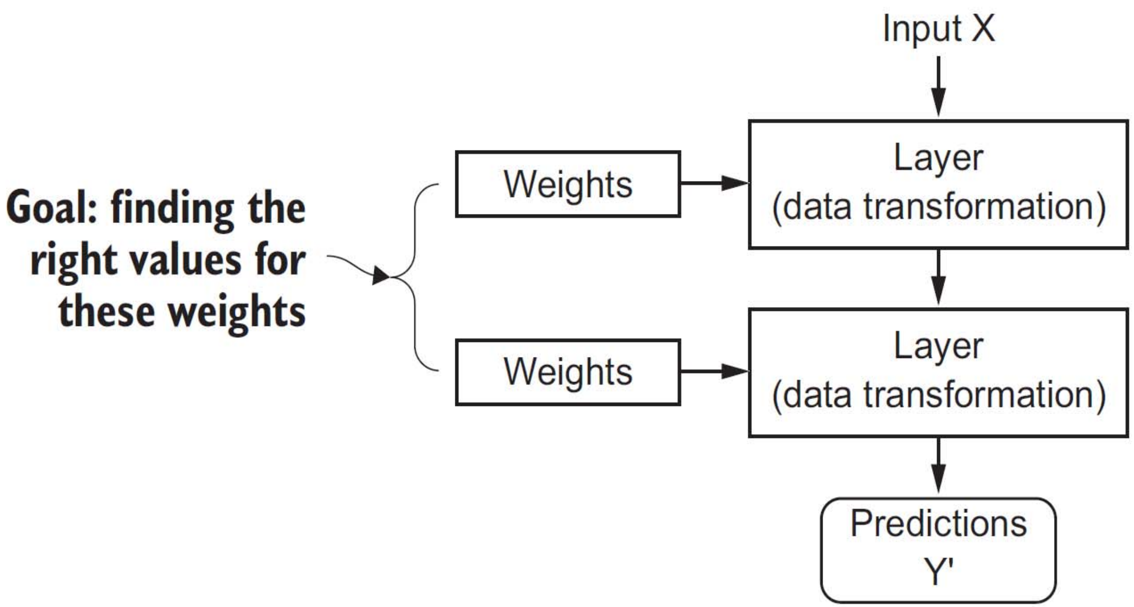
Forward Propagation

- After a neural network is trained, it is deployed to run inference - to classify, recognize, and process new inputs without updating parameters.
- The inference(predict) processing is also known as "**forward propagation.**"



How deep learning works?

- The specification of what a layer does to its input data is stored in the *layer's weights*, which in essence are a bunch of numbers.
- The transformation implemented by a layer is ***parameterized*** by its weights (or ***parameters***).
- In this context, *learning* means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets.



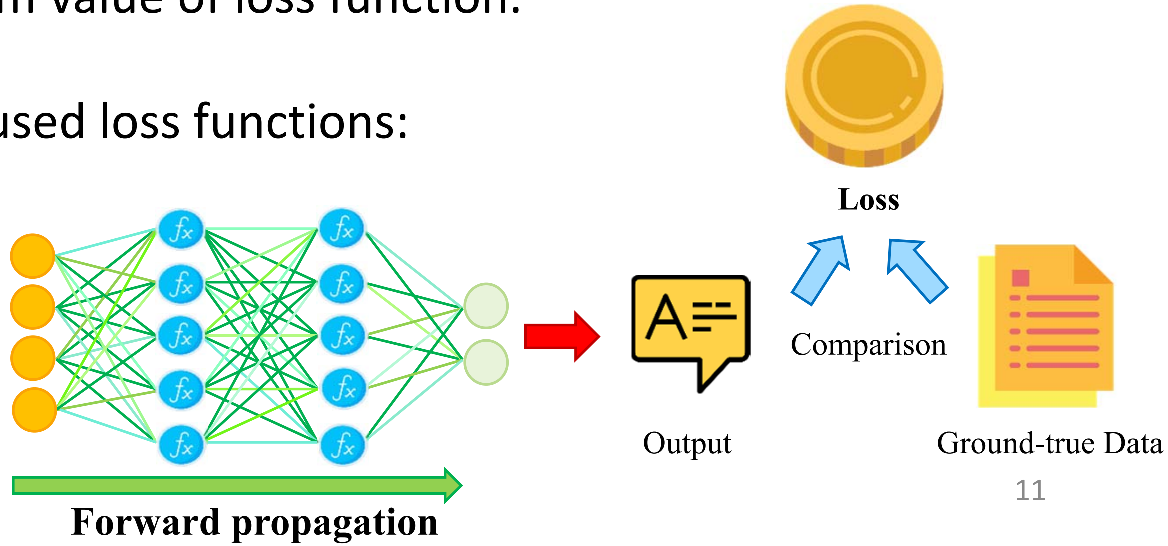
Problem

- Causing problems:
  - A deep neural network can contain **tens of millions of parameters**.
  - Finding the correct value for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behavior of all the others!

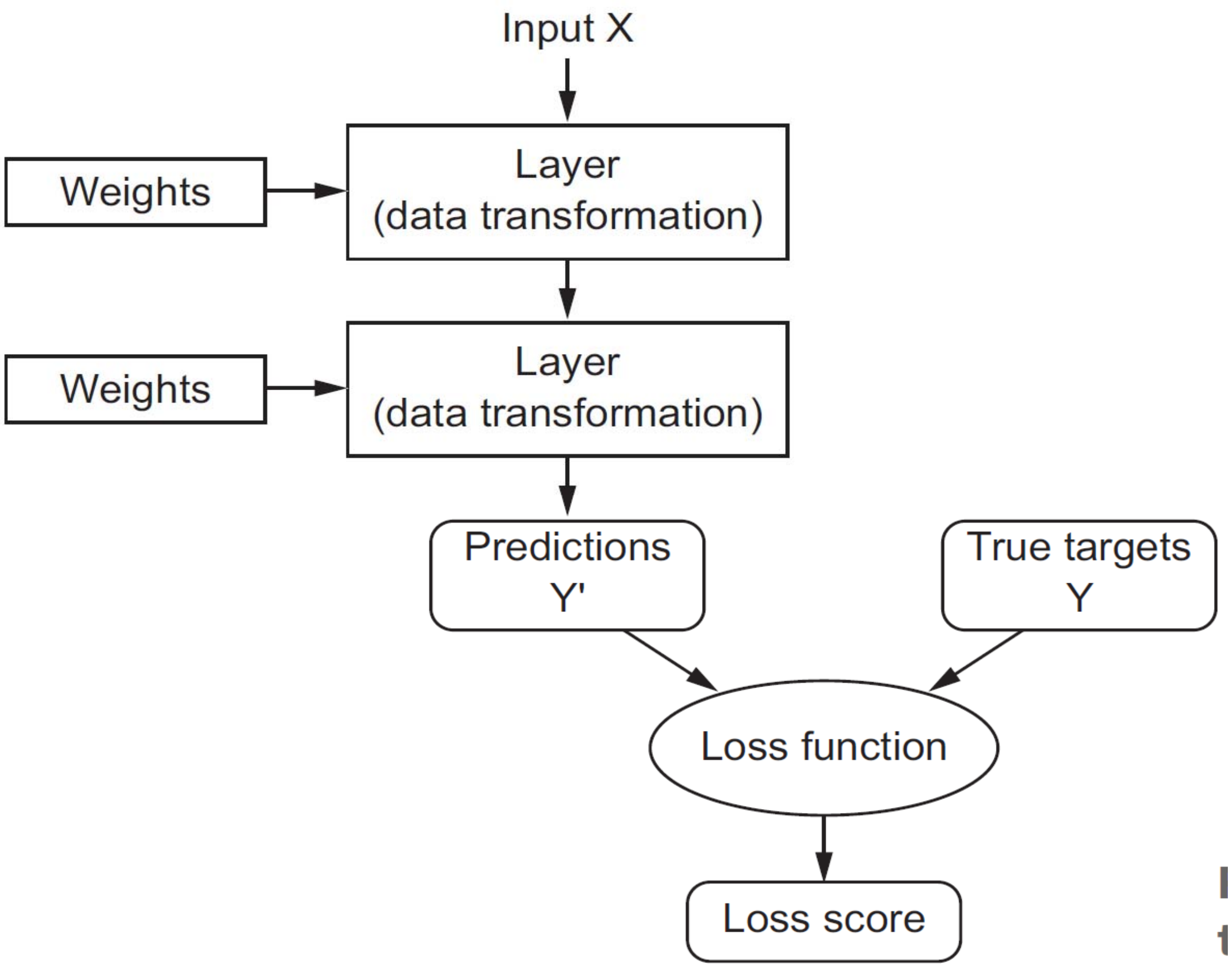
- Method
  - To control the output of a neural network, you need to be able to **measure** how **far** this output is from what you expected.
  - This is the job of the *loss function* of the network, also called the *objective function*.
  - The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example.

Loss Function

- Before mentioning backward propagation, we have to know about loss function, **gradient**, and **gradient descent** first.
- Loss function is a criterion that evaluates the performance of neural networks. It qualifies the agreement between the predicted output and the ground truth output.
- Neural networks calculate the **loss** of training data and find a set of parameters at the minimum value of loss function.
- There are two commonly used loss functions:
    - Mean square error.
    - Cross-entropy error.

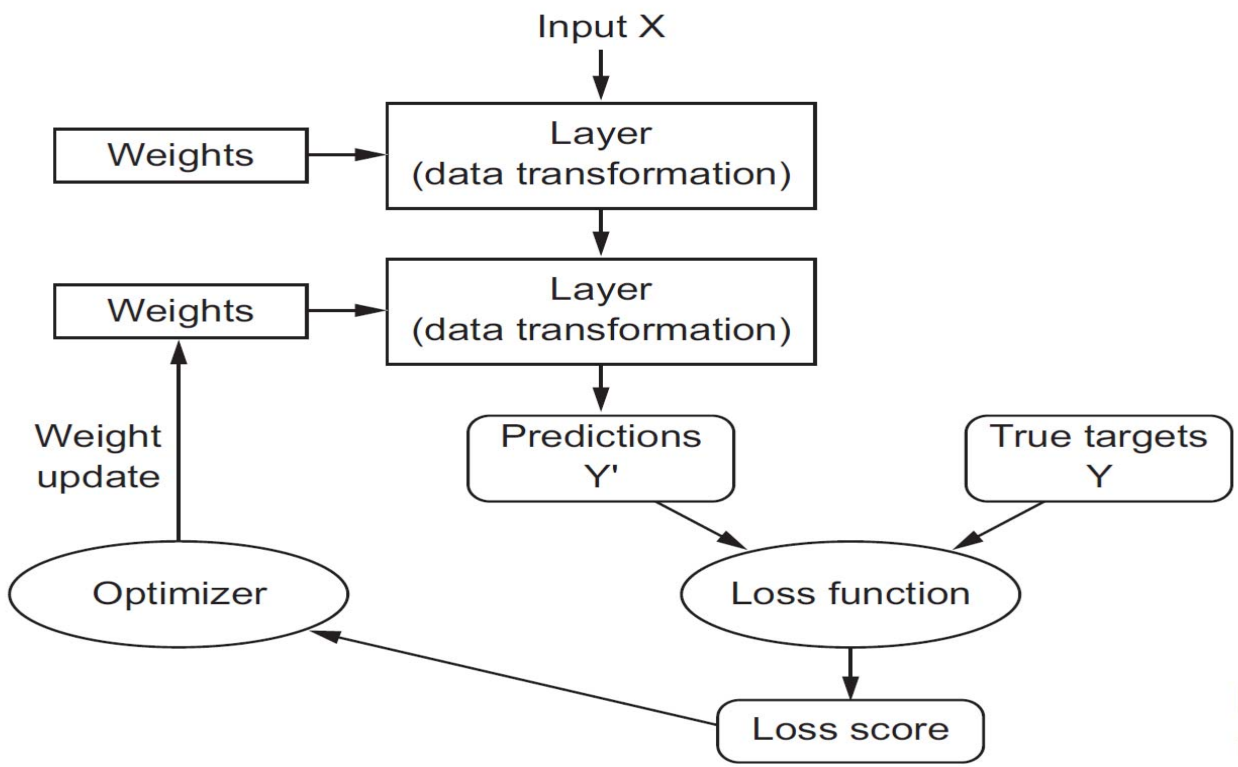


Loss Function



Feedback and *Backpropagation* algorithm

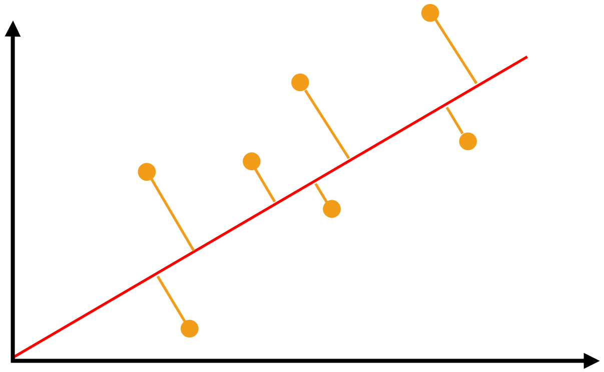
- Deep Learning model uses this **loss score** as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example.
- This adjustment is the job of the *optimizer*, which implements what's called the *Backpropagation* algorithm: the central algorithm in deep learning.



Mean Square Error

- Mean square error (MSE) is a measure of the quality of an estimator : The difference between the estimators and what is estimated, is always non-negative, and values closer to zero are better.

$$E = \frac{1}{k}\sum_{k}(y_k - t_k)^2$$



$$t_k = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad y_k = \begin{pmatrix} 0.4 \\ 0.6 \end{pmatrix} \Rightarrow E = 0.16$$

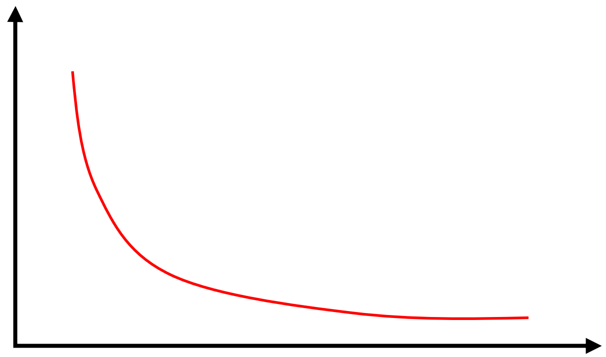
Training data (one-hot encoding)

Outputs of the network

Cross-Entropy

- Cross-entropy measures the difference between two probability distributions. If outputs approximate to corresponding labels, the result of cross-entropy is close to zero.

$$E = -\sum_{k} t_k \log y_k$$



Negative Log-likelihood

$$t_k = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad y_k = \begin{pmatrix} 0.4 \\ 0.6 \end{pmatrix} \Rightarrow E = 0.736$$

Training data (one-hot encoding)

Outputs of the network

Training loop

- Training loop:
  - Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations.
  - Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high.
  - But with every example the network processes, the weights are **adjusted a little in the correct direction**, and the loss score decreases.
- The training loop repeated a sufficient number of times
- A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network.

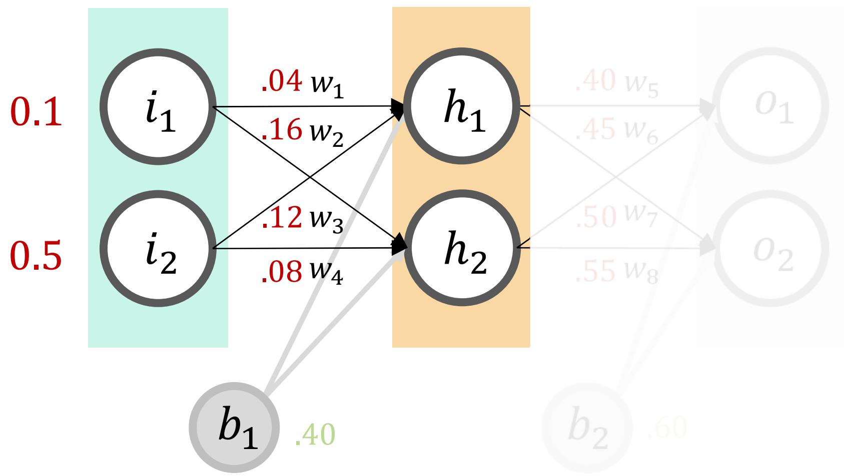
Updating Weights

- The only layer with the answer is the output layer
  - The only layer we can know the errors
- We need to update the weights from the output layer to hidden layers

- Solution: Back-propagation

Forward Propagation

$$\sum_i w_i x_i + b$$



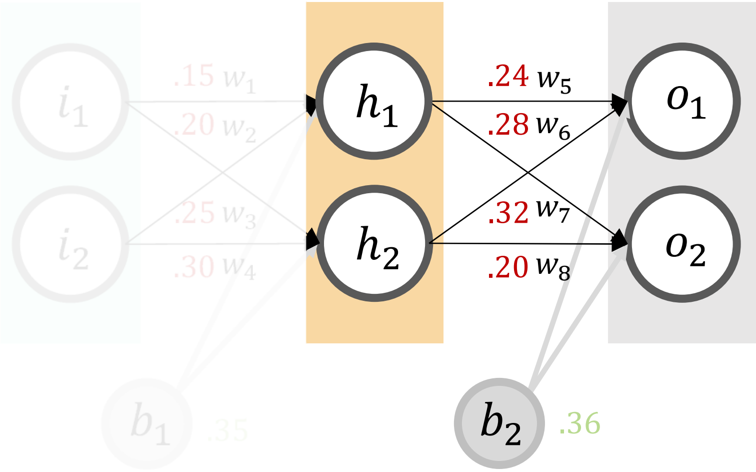
$$
\begin{aligned}
net_{h_1} &= w_1 * i_1 + w_2 * i_2 + b_1 * 1 \\
&= 0.04 * 0.1 + 0.12 * 0.5 + 0.40 * 1 \\
&= 0.464
\end{aligned}
$$

$$
out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}} = \frac{1}{1 + e^{-0.464}}
$$
$$
= 0.613962657
$$

$$
out_{h_2} = 0.611114647
$$

Forward Propagation

$$\sum_i w_i x_i + b$$



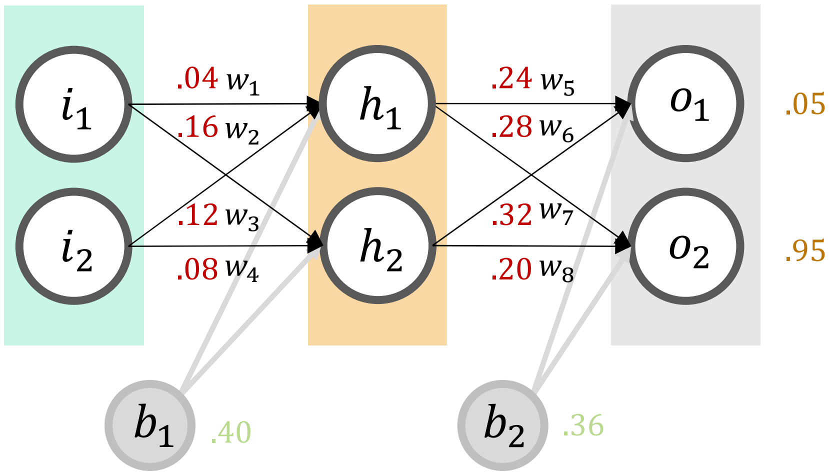
$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$
$$= 0.24 * 0.613962657 + 0.32 * 0.611114647 + 0.36 * 1$$
$$= 0.702907725$$

$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}} = \frac{1}{1 + e^{-0.702907725}}$$
$$= 0.668832137$$

$$out_{o_2} = 0.657941101$$

The Errors of Outputs

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$



$$E_{o_1} = \frac{1}{2}(target - output)^2$$
$$= \frac{1}{2}(0.05 - 0.668832137)^2$$
$$= 0.191476607$$

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o_1} + E_{o_2} = 0.191476607 + 0.042649200 = 0.234125807$$

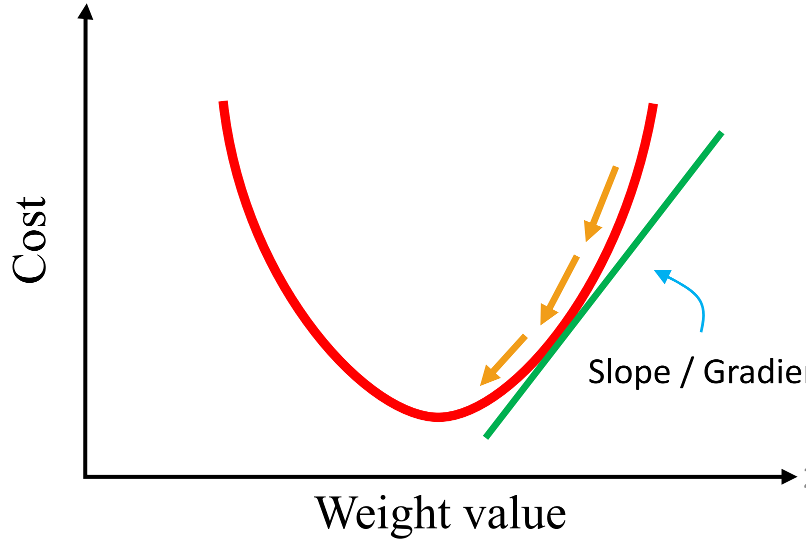
Updating Weights

- The only layer with the answer is the output layer
  - The only layer we can know the errors
- We need to update the weights from the output layer to hidden layers

- Solution: Back-propagation

Gradient Descent

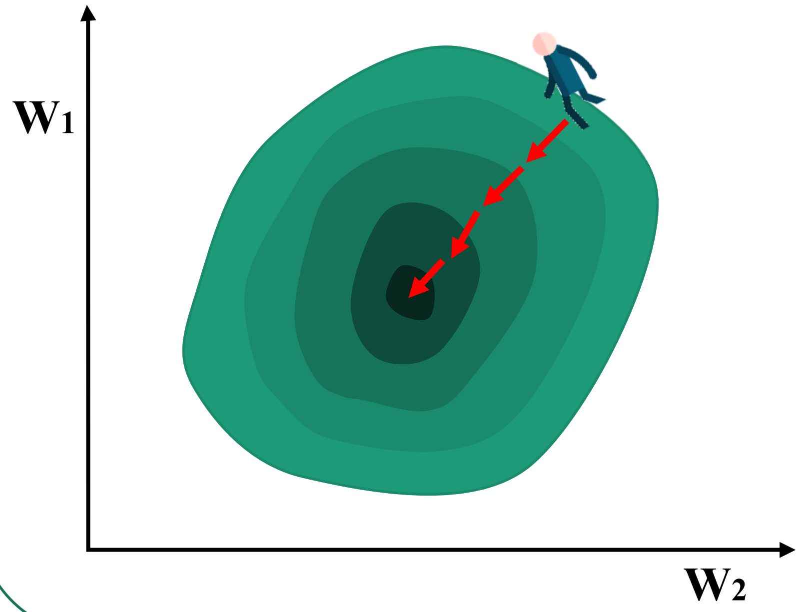
- Neural networks will find the best solution of parameters in the training phase while **minimizing the loss function**.
- In most cases, these parameters cannot be solved analytically, but they can be approximated well with iterative optimization algorithms like gradient descent.
- If we want to **minimize the loss function**, the parameters are updated to the negative direction of differential value (**gradient or slope**).



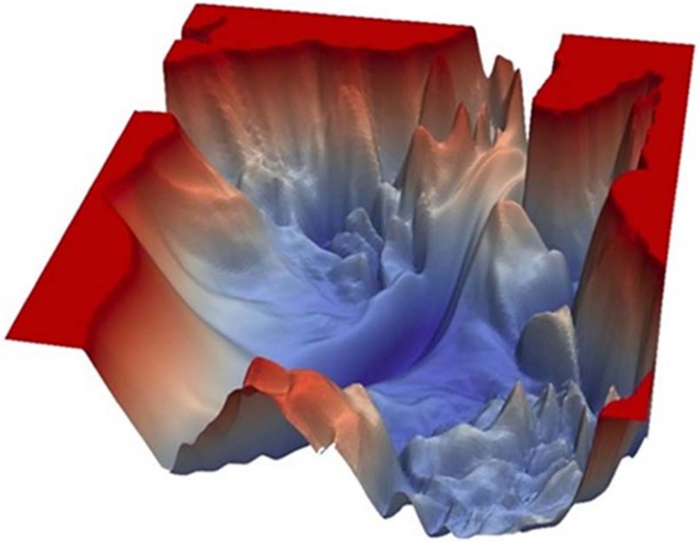
Gradient Descent

- Gradients in deep learning can be calculated by : $\dfrac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}}$
  - ***L*** is the loss function.
  - ***W*** is all weights in a neural network.
- If there are only two weights in loss function :

Loss Contour Line



The real condition may be :



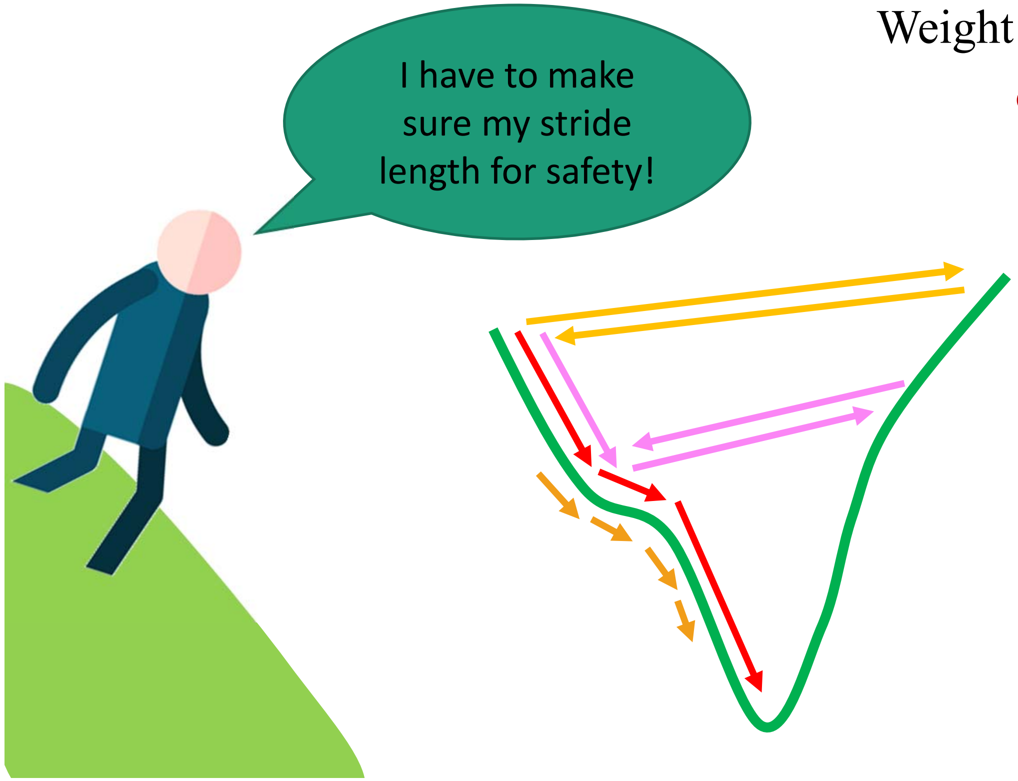
Learning Rate

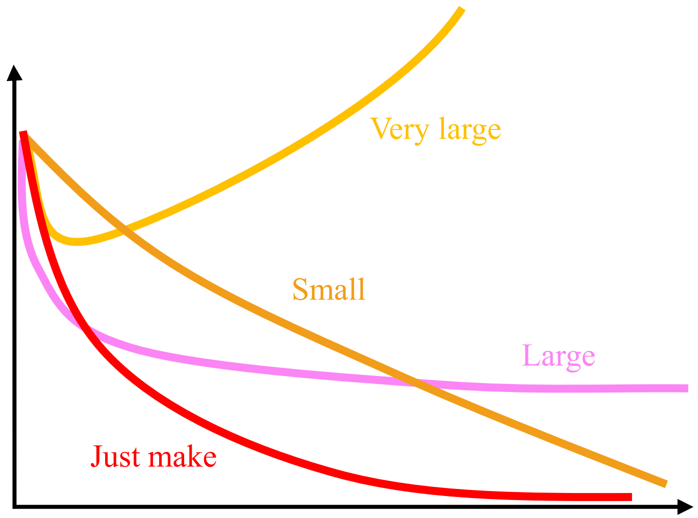
- **Learning rate** decides how far the step is to the next position on the loss function.
- It is also a kind of hyper-parameter determined by humans. Thus we have to set the value carefully.

Weight updating : $\boldsymbol{W}^1 = \boldsymbol{W}^0 - \boxed{\eta}\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{W}}\Big|_{\boldsymbol{W}=\boldsymbol{W}^0}$

Change in the opposite direction.

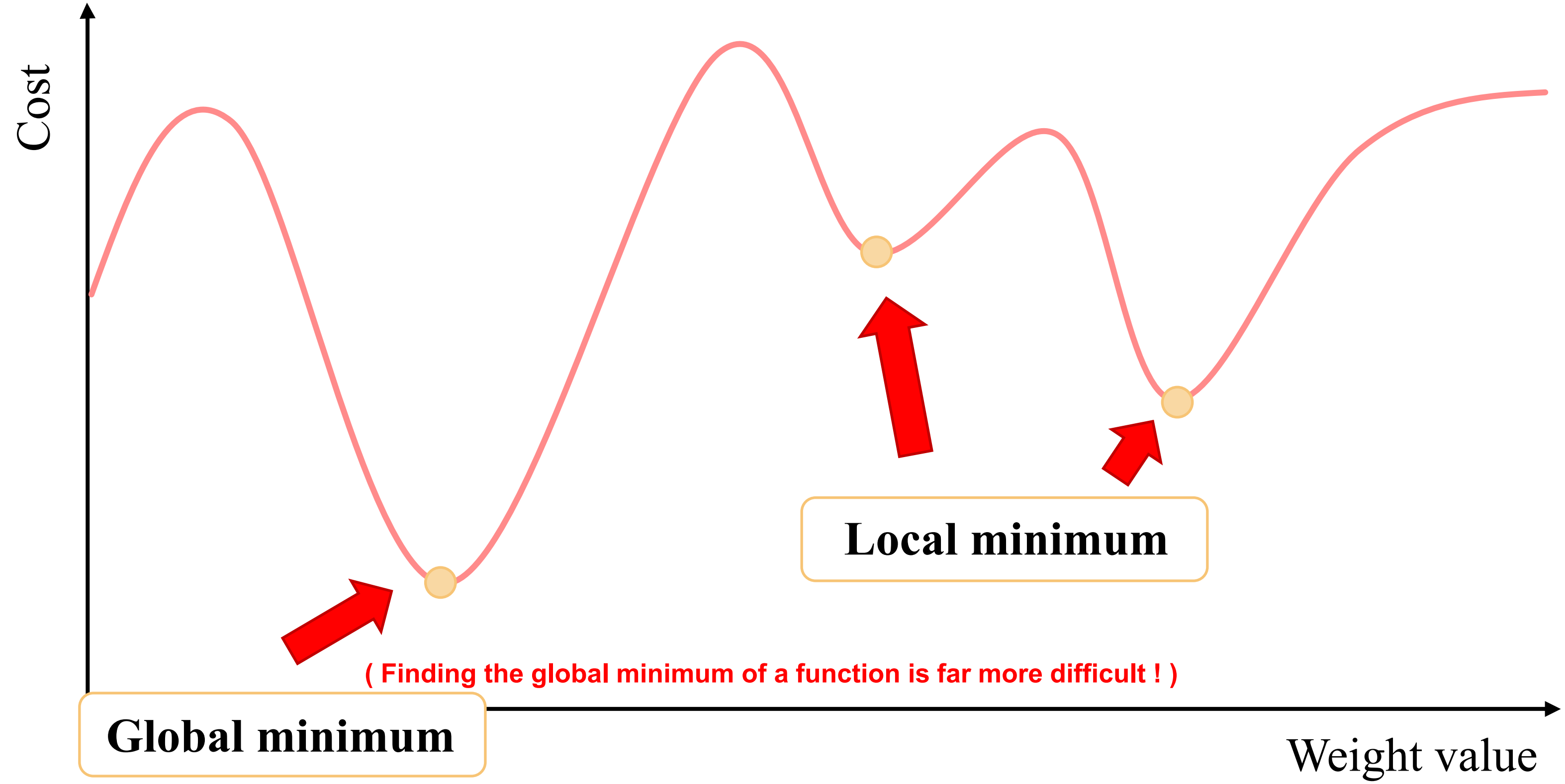
Learning rate





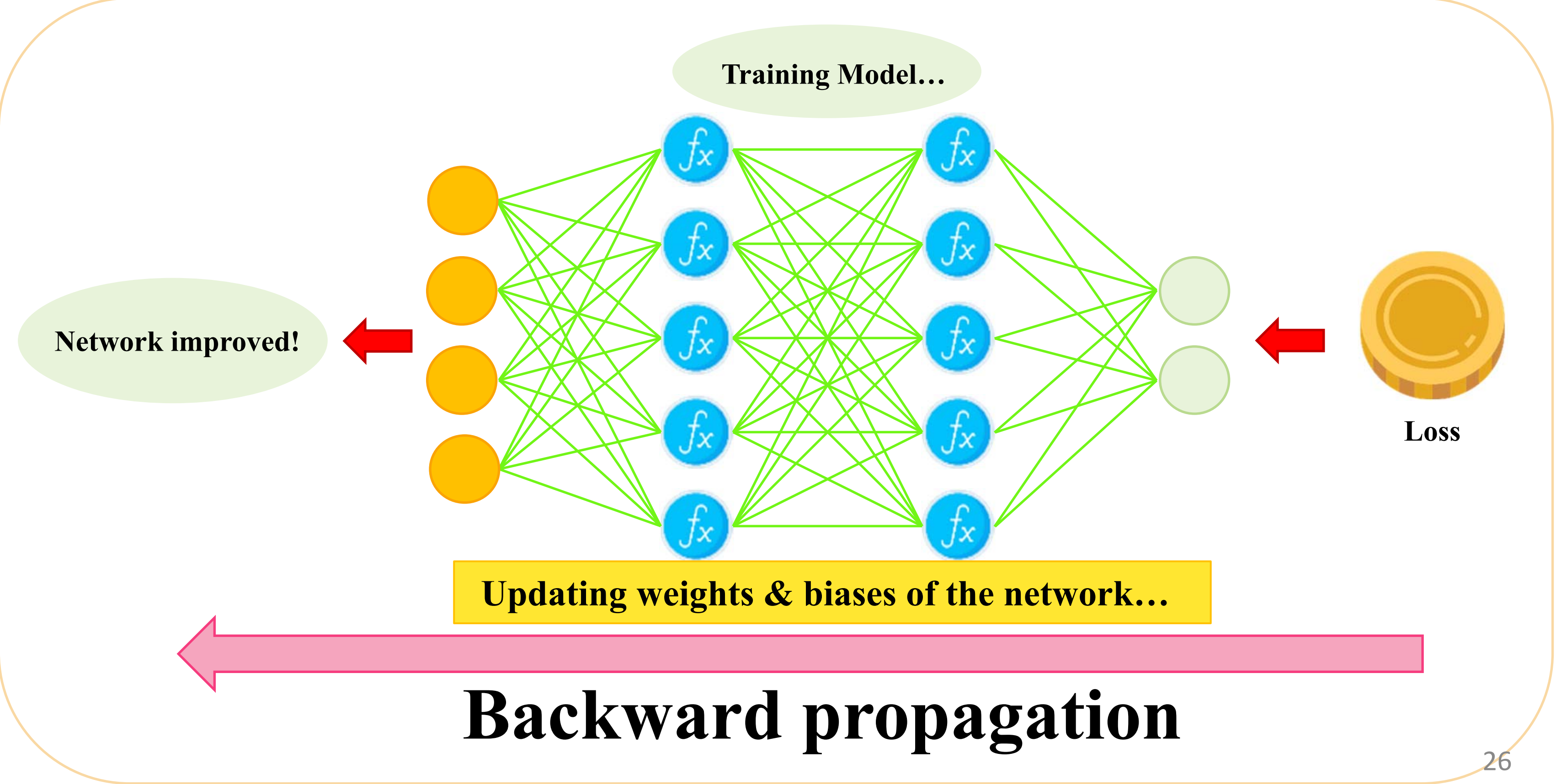
Critical Point

- A **local minimum** of a function is a point where the function value is smaller than the nearby points.
- A **global minimum** is a point where the function value is smaller than at all other feasible points.



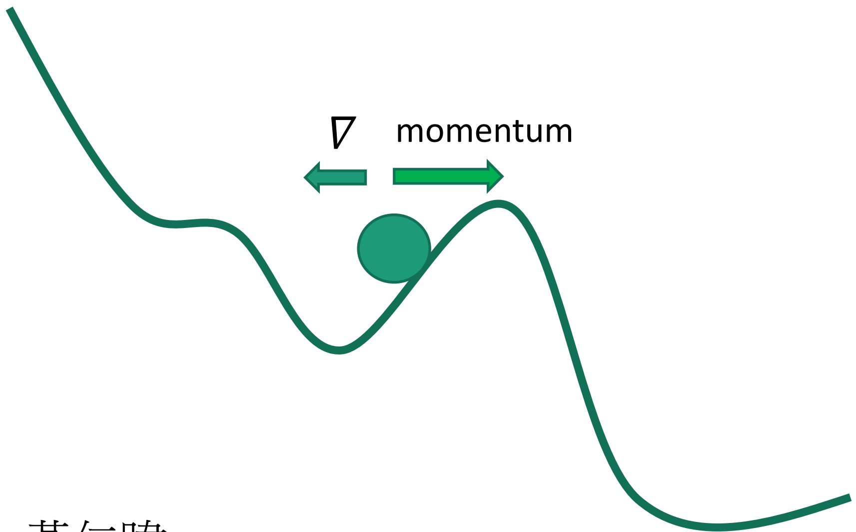
Backward Propagation

- When the loss function has been calculated. We can apply it to **backward propagation**, utilizing the gradients and learning rate to **update** the weight.



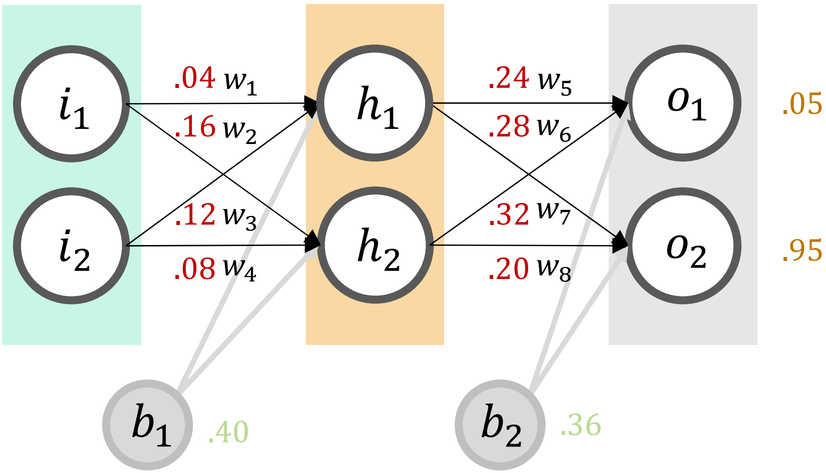
Different Way to Optimize Neural Networks

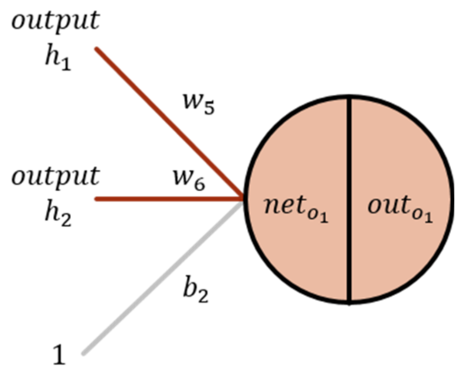
- **Stochastic Gradient Descent (SGD)**
  - Update the weights at each input example instead of update the weight after each epoch
- Add momentums on gradient descent
  - $v_{t+1} = \lambda v_t - (1 - \lambda) * \frac{\partial E_{total}}{\partial w_1}$
  - $w_1^+ = w_1 + \eta * v_{t+1}$



Backward Propagation to Update w5

- Output layer





$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

$$E_{total} = \frac{1}{2}(target - out_{o_1})^2 + \frac{1}{2}(target - out_{o_2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o_1}} = 2 * \frac{1}{2}\left(target_{o_1} - out_{o_1}\right)^{2-1} * (-1) + 0$$

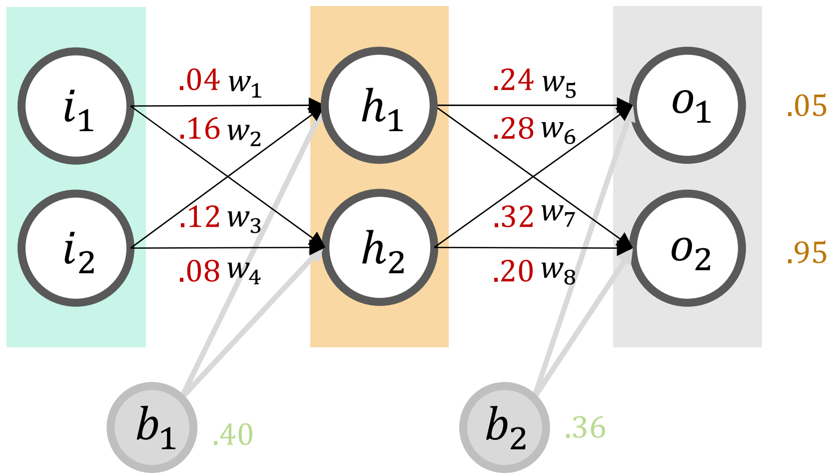
$$= -\left(target_{o_1} - out_{o_1}\right)$$

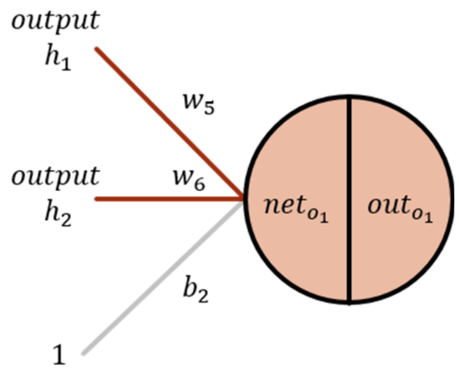
$$= -(0.05 - 0.668832137)$$

$$= 0.618832137$$

Backward Propagation to Update w5

- Output layer





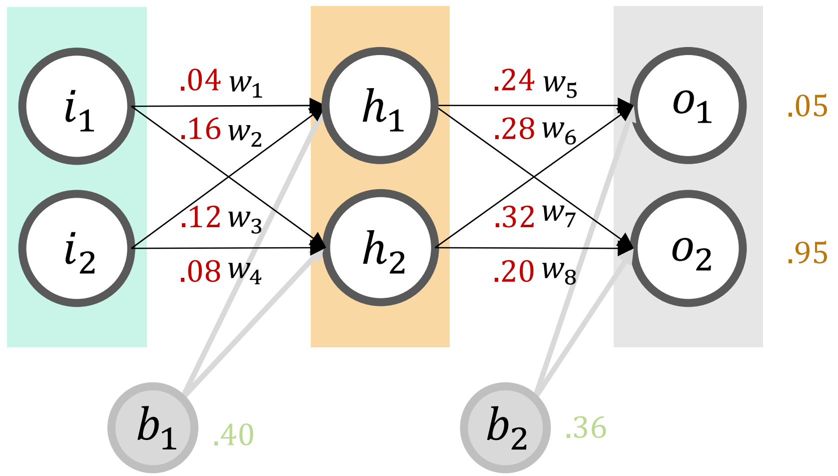
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

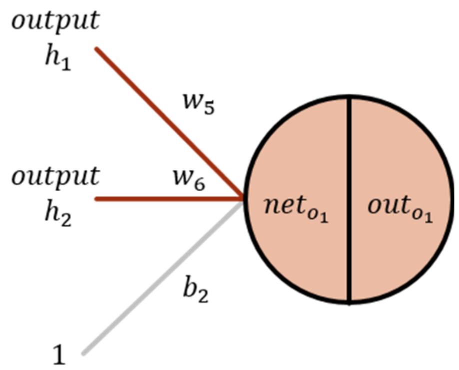
$$out_{o_1} = \frac{1}{1 + e^{-net_{o_1}}}$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = out_{o_1}(1 - out_{o_1})$$
$$= 0.668832137(1 - 0.668832137)$$
$$= 0.337664274$$

Backward Propagation to Update w5

- Output layer





$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$

$$\frac{\partial net_{o_1}}{\partial w_5} = 1 * out_{h_1} * w_5^{1-1} + 0 + 0$$

$$= out_{h_1}$$

$$= 0.613962657$$

Backward Propagation to Update w5

- Output layer

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.618832137 * 0.337664274 * 0.613962657 = 0.128292105$$

$$w_5^+ = w_5 - \boxed{\eta} * \frac{\partial E_{total}}{\partial w_5}$$

Learning rate

$$= 0.15 - 0.5 * 0.128292105$$

$$= 0.858539475$$

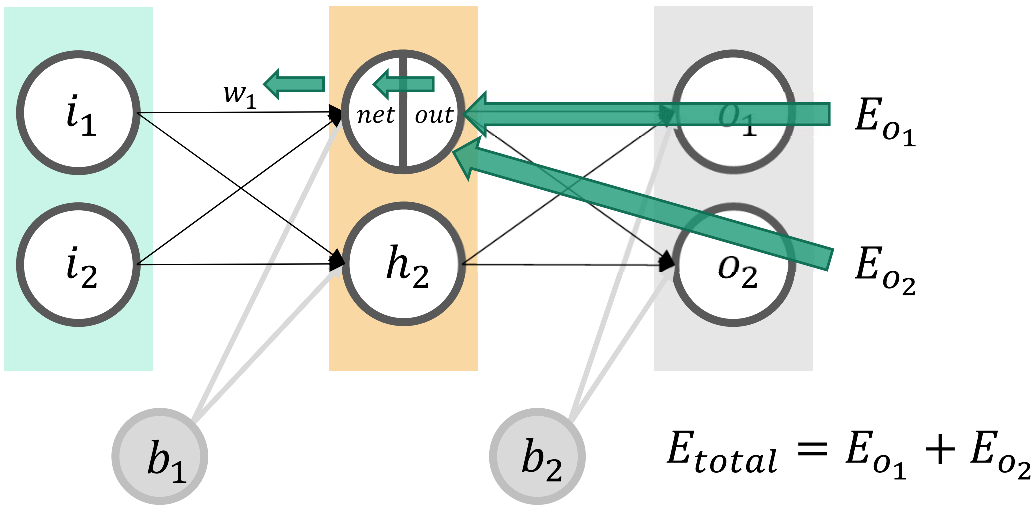
$$w_6^+ = 0.238117666$$

$$w_7^+ = 0.300177638$$

$$w_8^+ = 0.300084039$$

Backward Propagation to Update w1

- Hidden layer



$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

Backward Propagation to Update w1

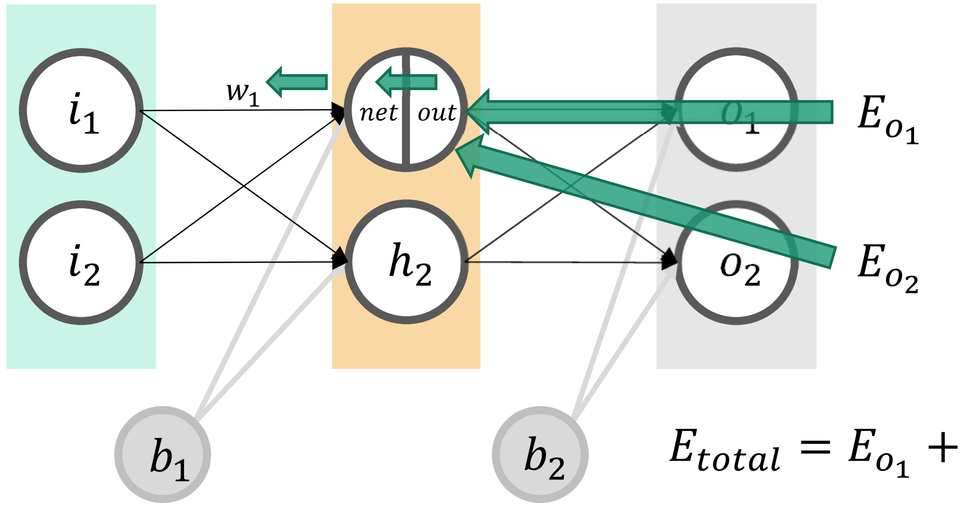
- Hidden layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial net_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial net_{o_1}} = \frac{\partial E_{o_1}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}}$$



$$E_{total} = E_{o_1} + E_{o_2}$$

$$\frac{\partial E_{o_1}}{\partial net_{o_1}} = \frac{\partial E_{o_1}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} = 0.618832137 * 0.337664274 = 0.208957504$$

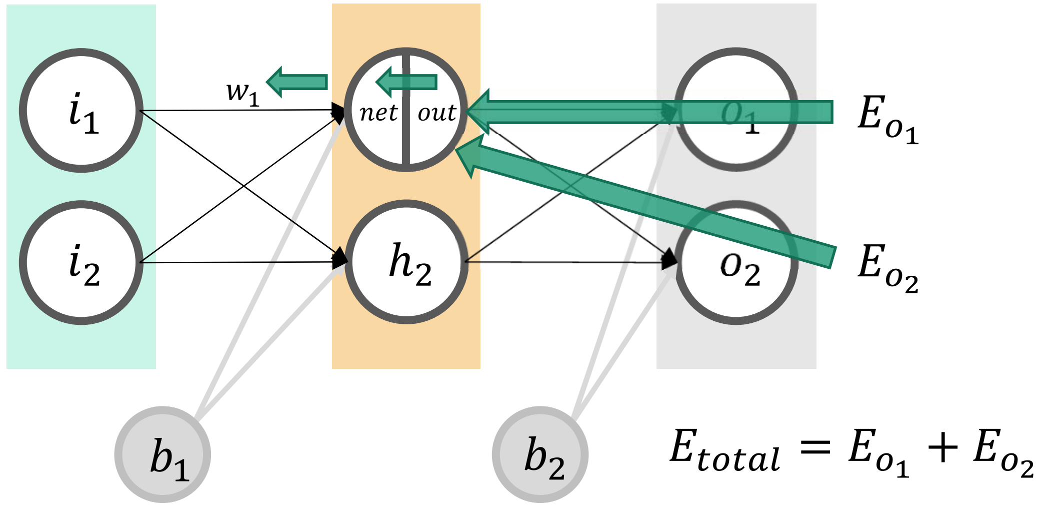
Backward Propagation to Update w1

- Hidden layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial net_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}}$$



$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$

$$\frac{\partial net_{o_1}}{\partial out_{h_1}} = w_5 = 0.24$$

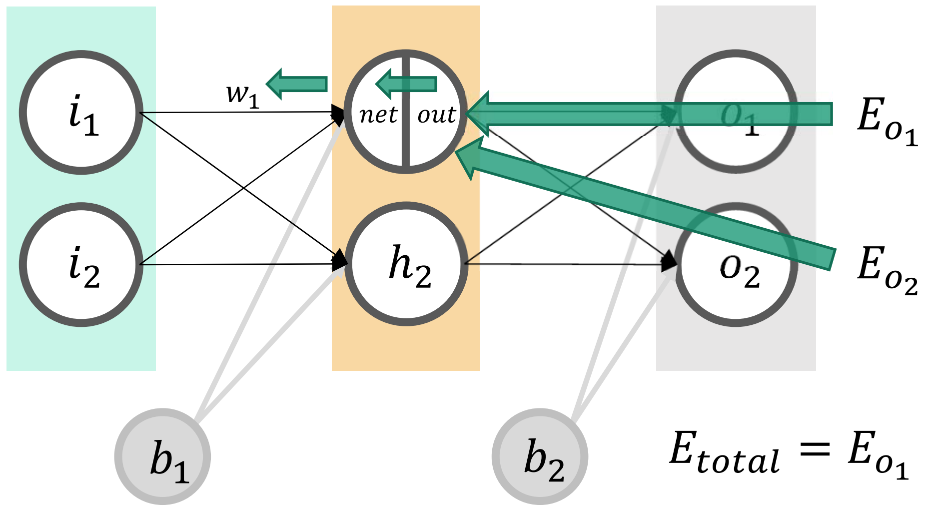
$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}} = 0.208957504 * 0.24 = 0.050149801$$

Backward Propagation to Update w1

- Hidden layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial net_{h_1}}$$



$$E_{total} = E_{o_1} + E_{o_2}$$

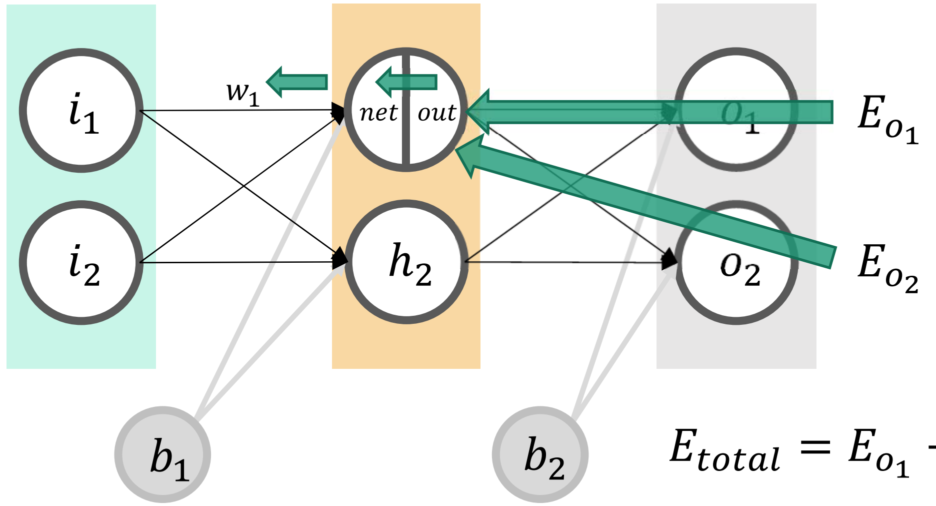
$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = 0.050149801, \qquad \frac{\partial E_{o_2}}{\partial out_{h_1}} = -0.018404176$$

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial net_{h_1}} = 0.050149801 + (-0.018404176\,) = 0.031745625$$

Backward Propagation to Update w1

- Hidden layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$



$$E_{total} = E_{o_1} + E_{o_2}$$

$$out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}}$$

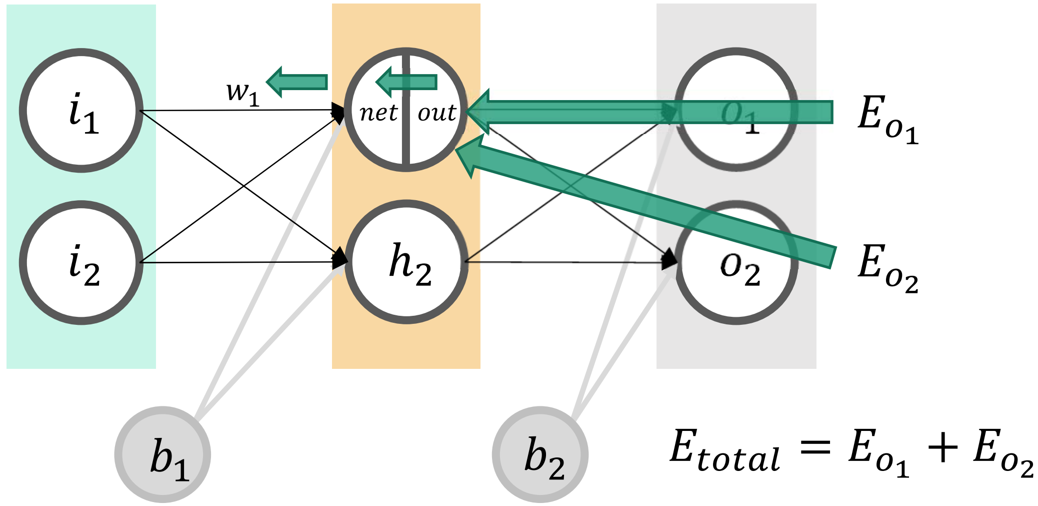
$$\frac{\partial out_{h_1}}{\partial net_{h_1}} = out_{h_1}(1 - out_{h_1}) = 0.613962657 * (1 - 0.613962657\ )$$

$$= 0.237012513$$

Backward Propagation to Update w1

- Hidden layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \boxed{\frac{\partial net_{h_1}}{\partial w_1}}$$



$$net_{h_1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h_1}}{\partial w_1} = i_1 = 0.1$$

Backward Propagation to Update w1

- Hidden layer

$$\boxed{\frac{\partial E_{total}}{\partial w_1}} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.031745625 * 0.237012513 * 0.1 = 0.000752411$$

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1}$$
$$= 0.04 - 0.5 * 0.000752411$$
$$= 0.039623795$$

$$w_2^+ = 0.118118973$$

$$w_3^+ = 0.159635010$$

$$w_4^+ = 0.078175051$$

How to Design A Good Neural Network Model

The Hyper Parameters

- Dimensions
  - The number of neurons in each layer
  - Can be different in each layer
- Numbers of layers
  - How depth is your model
- Activation function
  - Linear: ReLu
  - Non-linear: Sigmoid, tanh
- The bias in each layer

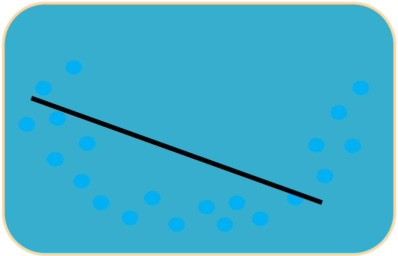
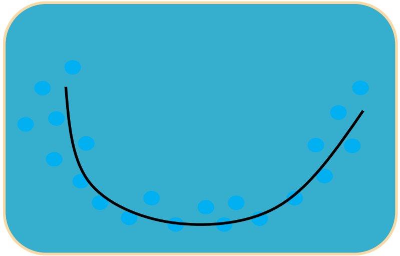
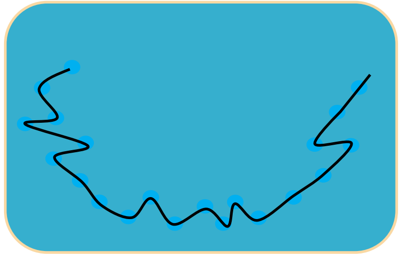
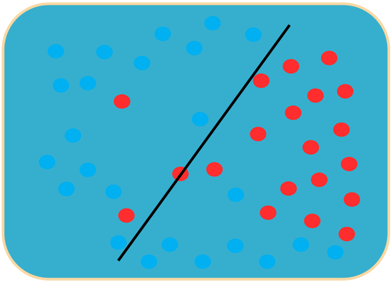
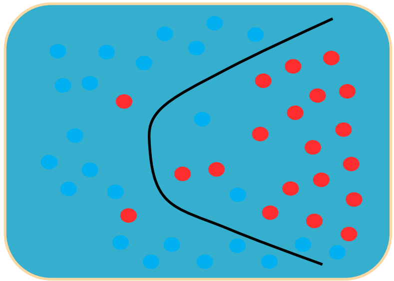
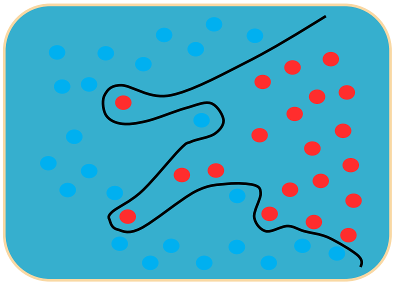
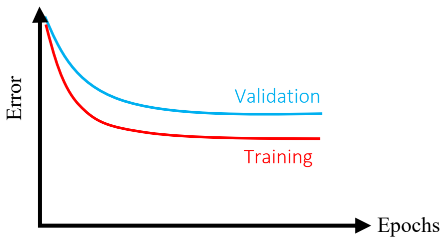
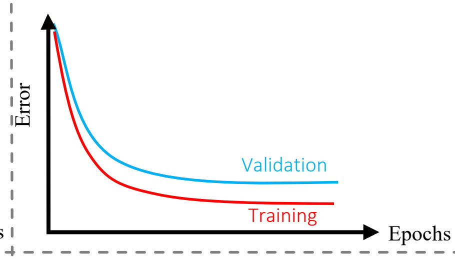
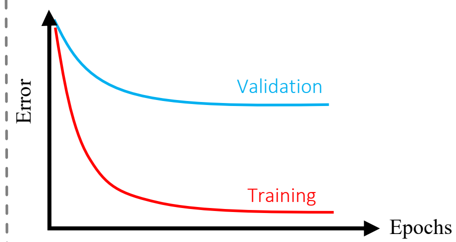
How Many Random Variables in Neural Networks

- Consider a neural network
  - 10 layers
  - 100 nodes in each layer
  - 1 bias in each layer
- 1 layer has 10100 parameters
  - 100*100+100
- 10 layers has 101000 parameters
  - 10100*10

The More Random Variables The Better?

- More random variables can represent more latent information
- Too many random variables will lead overfitting
  - Too fit to some special cases

Overfitting & Underfitting

	Underfitting	Just make	Overfitting
Symptoms	· High training error. · Training error close to testing error. · High bias.	· Training error slightly lower than testing error.	· Very low training error. · Training error much lower than test error. · High variance.
Regression			
Classification			
Deep learning			
Possible remedies	· Complexify model. · Add more features. · Train longer.		· Perform regularization. · Get more data.

How to Prevent Overfitting

- **Decrease** your random variables
  - Decrease your dimensions or layers
  - May incur some errors

- **Increase** your training data
  - Very difficult in practice

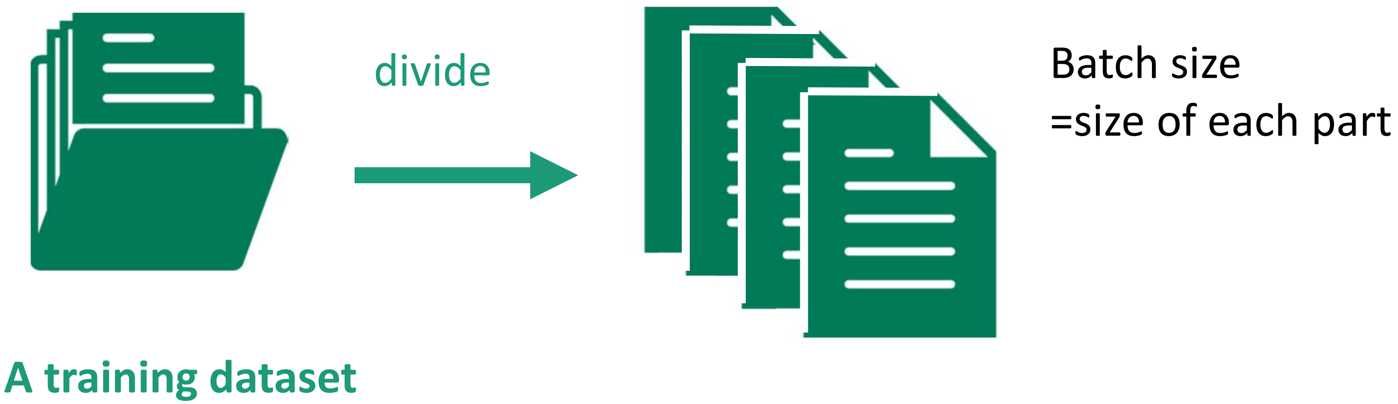
- **Dropout** some variables
  - Let some variables **not be trained** in the training phase
  - Still in **use** in the **testing phase**

How Much Training Data We Need

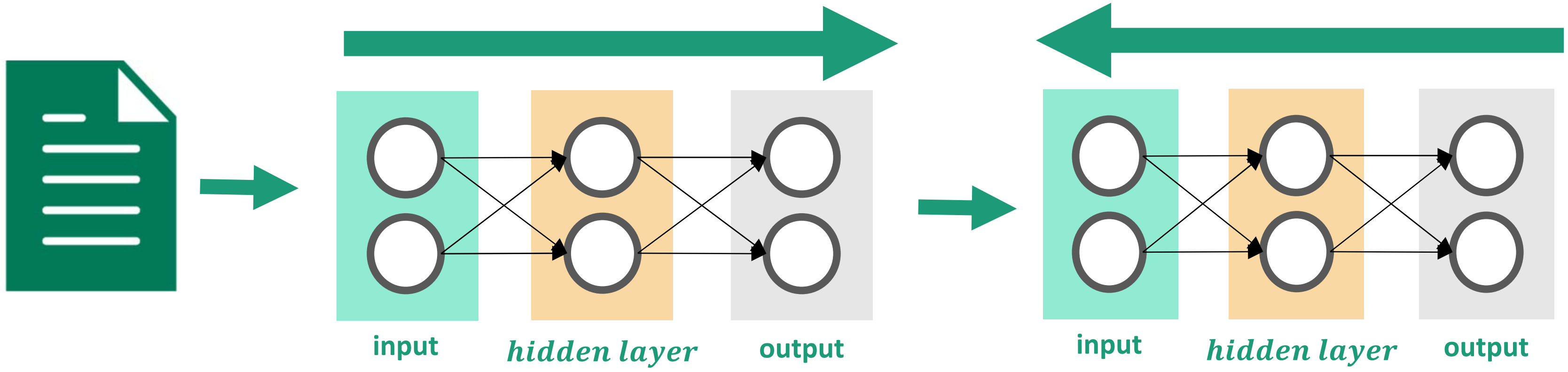
- 10~30 times data to train random variables
  - we need 1010000 ~ 3030000 data to train 101000 variables
- Few data may not be able to train a good model
  - Some variables may not be trained well

More Hyper Parameters

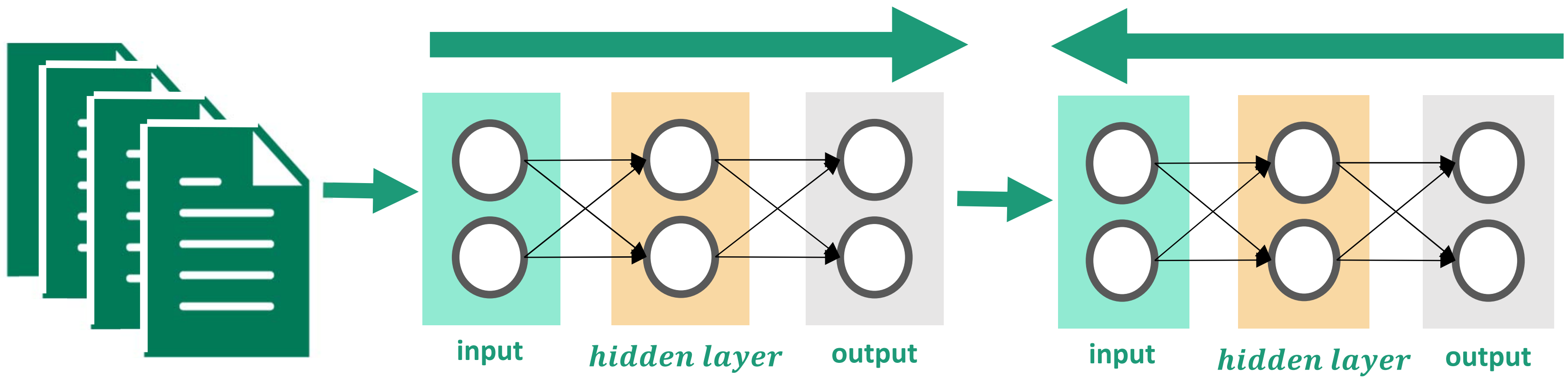
- **Learning rate**
  - How many updates of the weight via gradients
- **Batch size**
  - How many input data in each iteration
- **Epoch** [epək]
  - How many times of passing the training data in the model



The Difference Between Iteration and Epoch

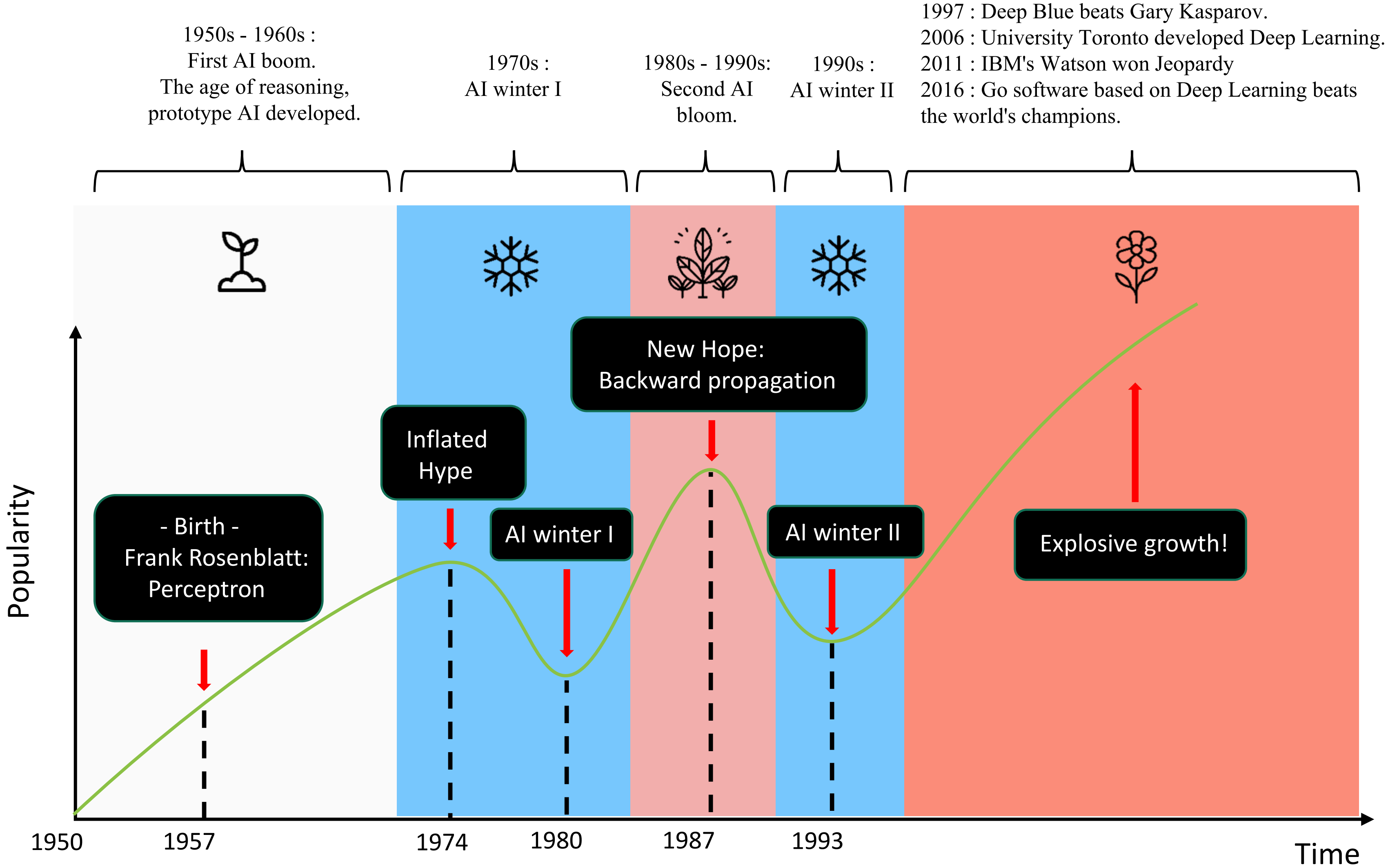


One iteration

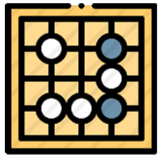


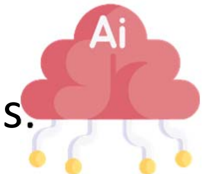
One epoch

The History of AI

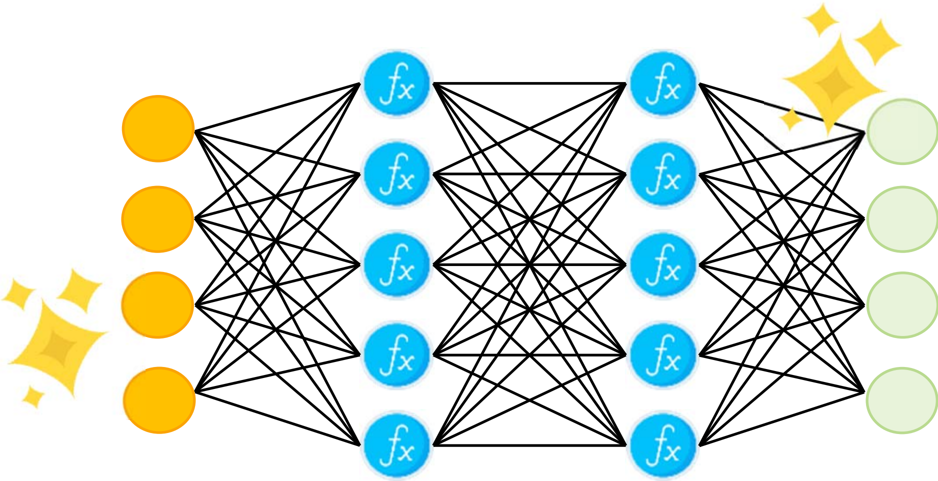


What Can Deep Learning Do?

- Image recognition
  - Deep learning can reach a high accuracy that humans cannot accomplish.
- Game
  - AlphaGo 
  - The computer can learn by itself and even better than humans.



- There are more and more applications of deep learning.



Learning Algorithms

Learning Algorithms

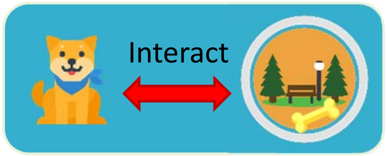
Supervised Learning

Supervised learning requires a **labeled dataset.**
The network can learn from it to make inferences or predictions of the problem.

Unsupervised Learning

Unsupervised learning is the opposite of supervised learning.
There is **no labeled dataset** in unsupervised learning.

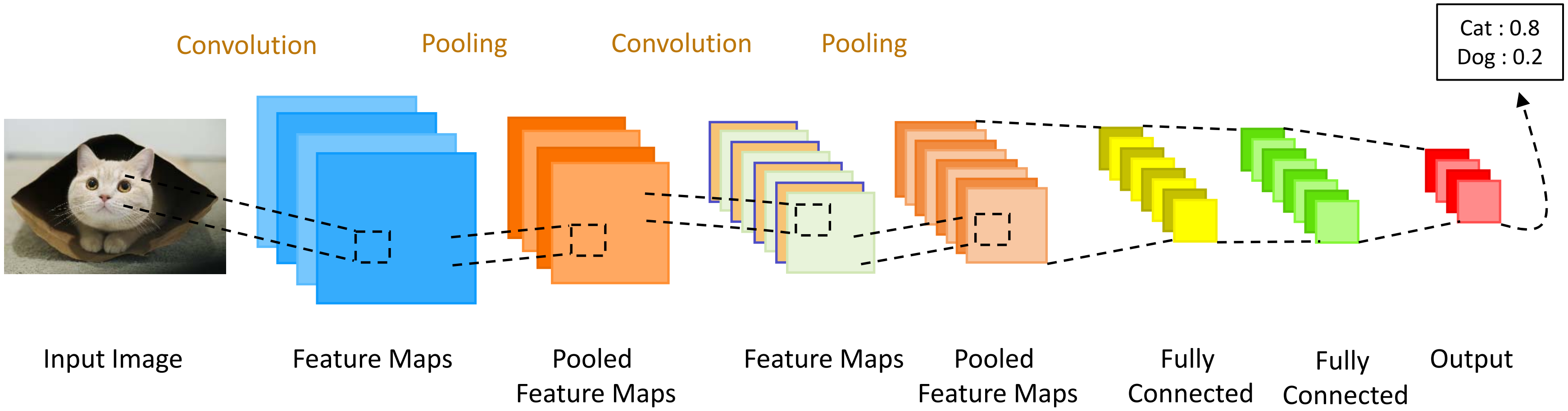
Reinforce Learning



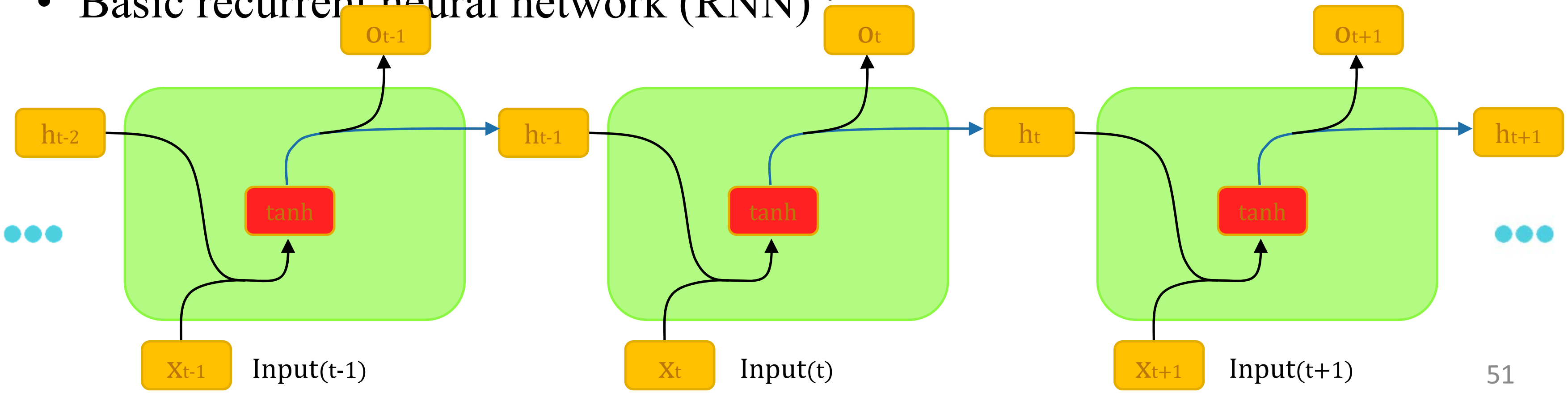
Reinforce learning model will learn to **react to the environment** by itself, with a system composed of **reward, state, and action**.

Basic Model of Neural Network

- Basic convolutional neural network (CNN) :

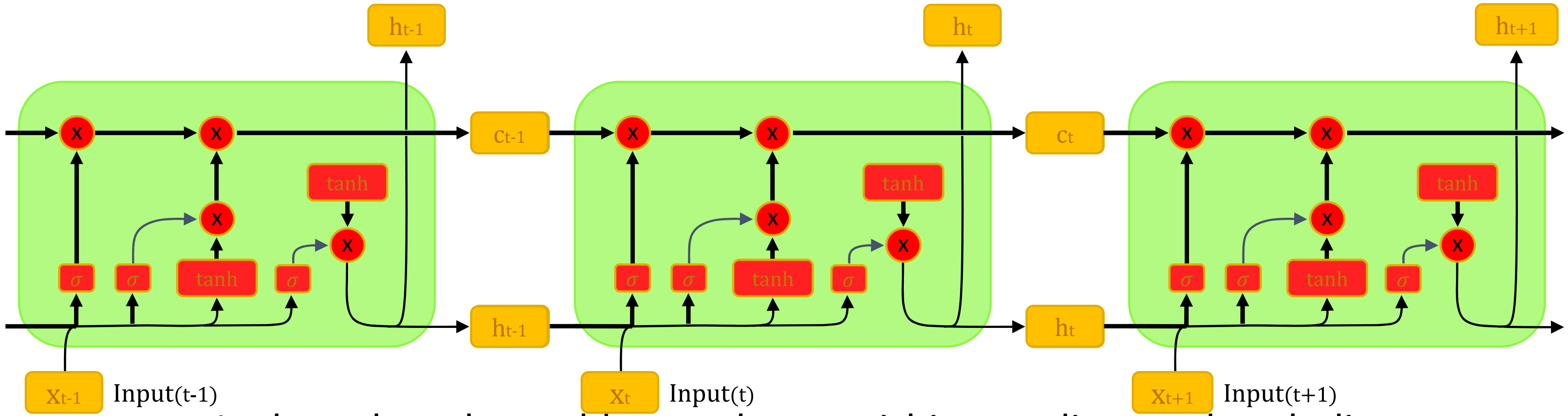


- Basic recurrent neural network (RNN) :



Advanced Model of Neural Network

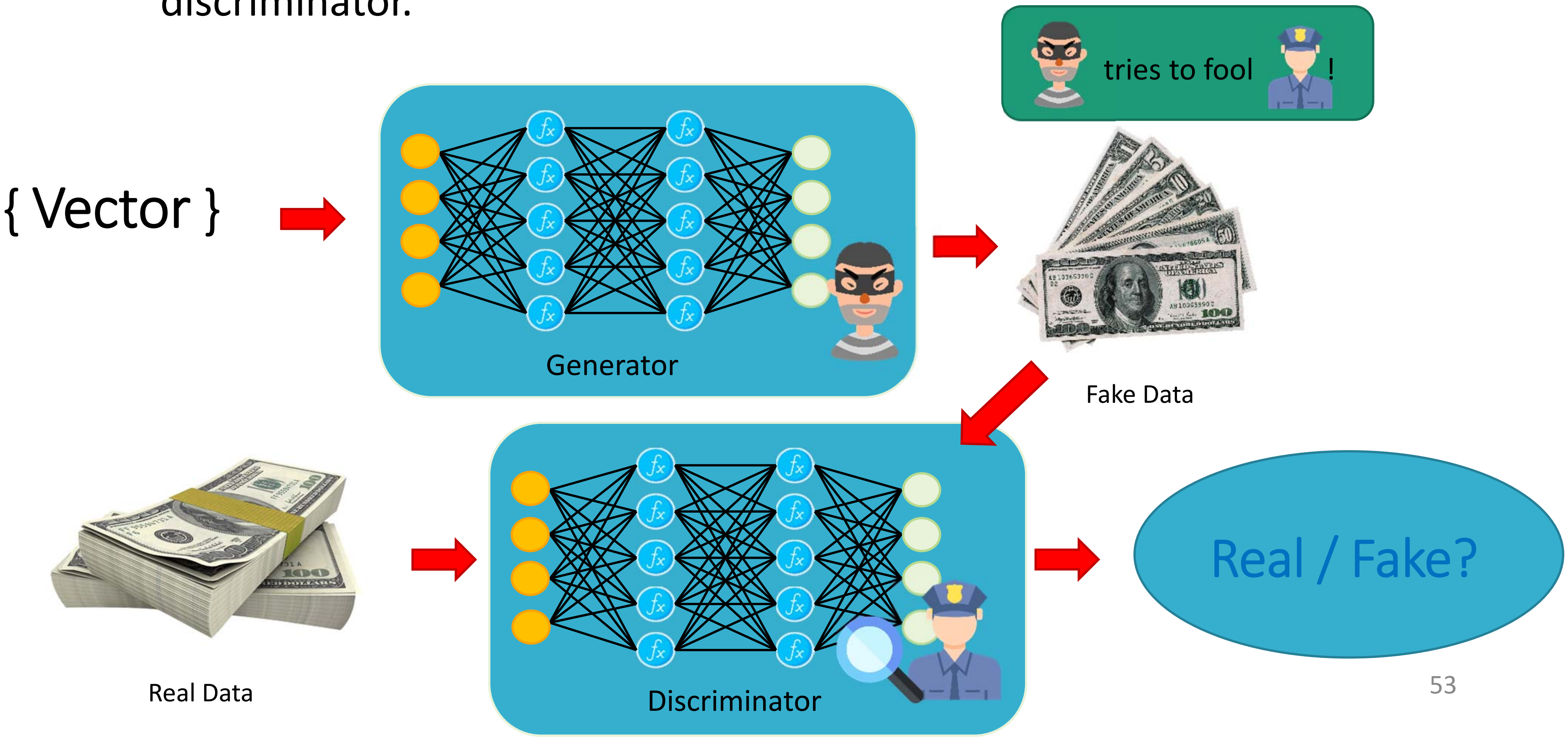
- Long short-term memory (LSTM) :
  - LSTM enables RNN to remember inputs over a long time.



  - It also solves the problem such as vanishing gradient and exploding gradient.

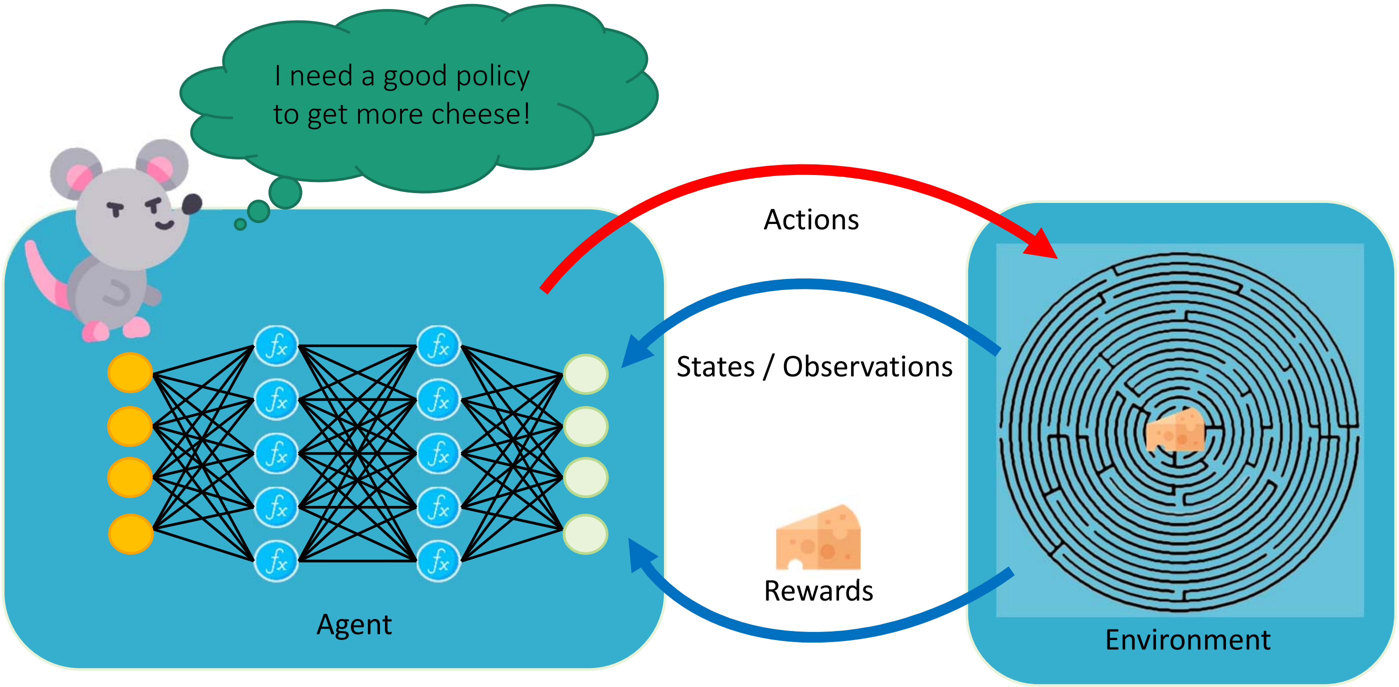
Advanced Model of Neural Network

- Generative adversarial network (GAN) :
  - GAN is a potential network that can generate image/voice/text data.
  - Basic GAN architecture includes two networks. The generator and the discriminator.



Advanced Model of Neural Network

- Deep Q network (DQN) :
  - The mission of DQN is to find an optimized **policy(strategy)** for winning more rewards.
  - In DQN, we will put the agent in the environment. It will learn better policy during interacting with the environment.



Applications

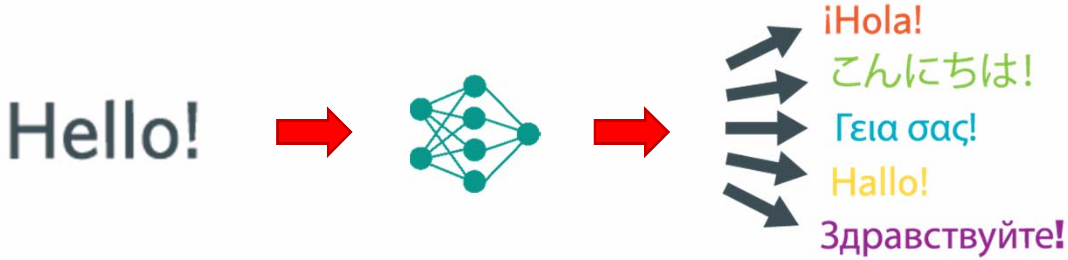
- Image segmentation :

- Object detection :

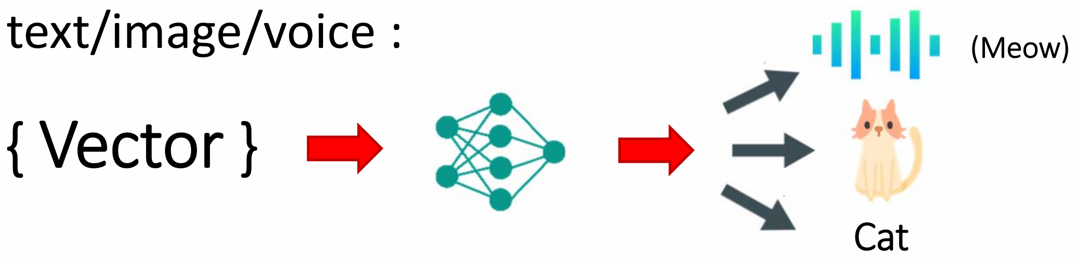
- Speech recognition :

Applications

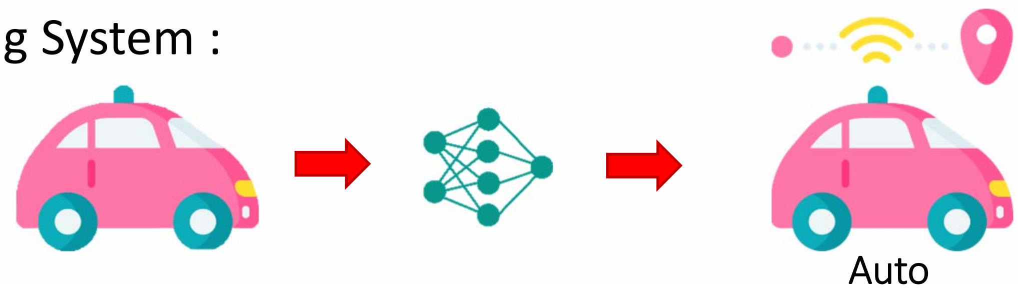
- Language translation :



- Generate text/image/voice :



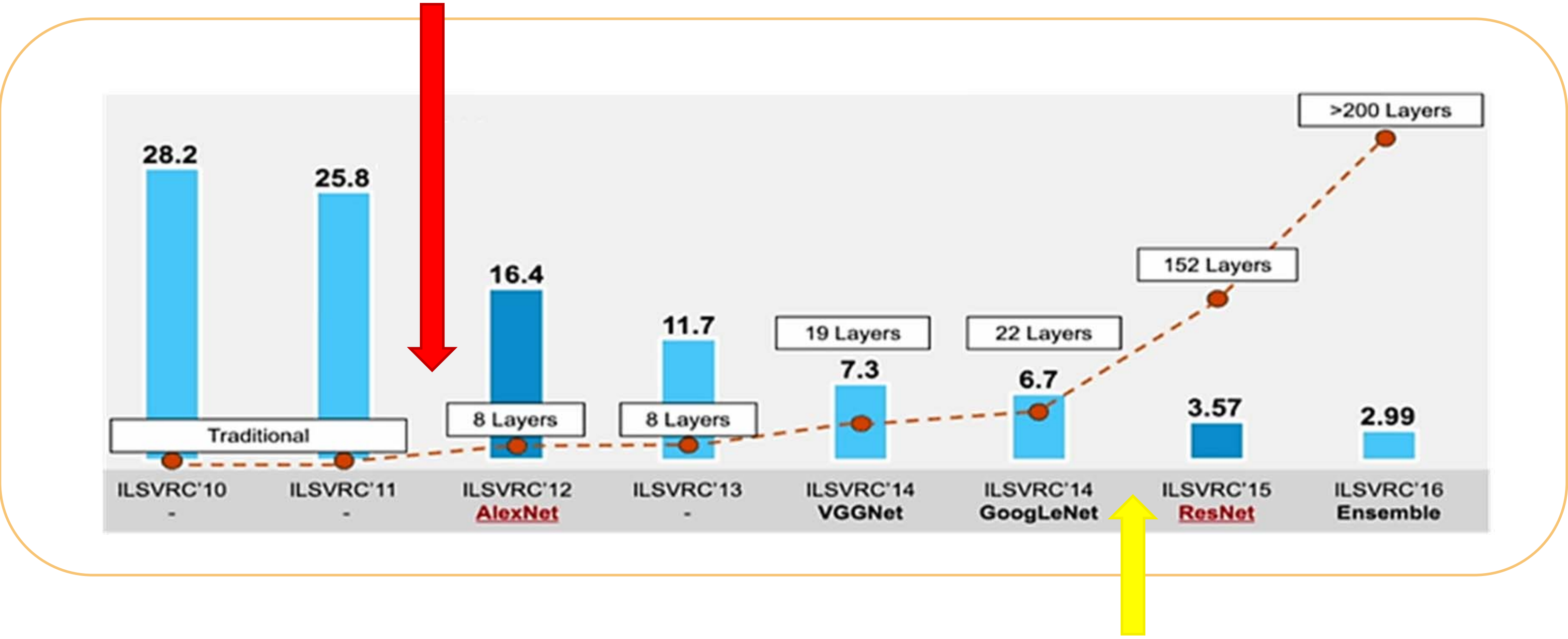
- Self-Driving System :



ILSVRC

- ImageNet Large Scale Visual Recognition Challenge.
- Deep models first perform good performance in commercial applications.

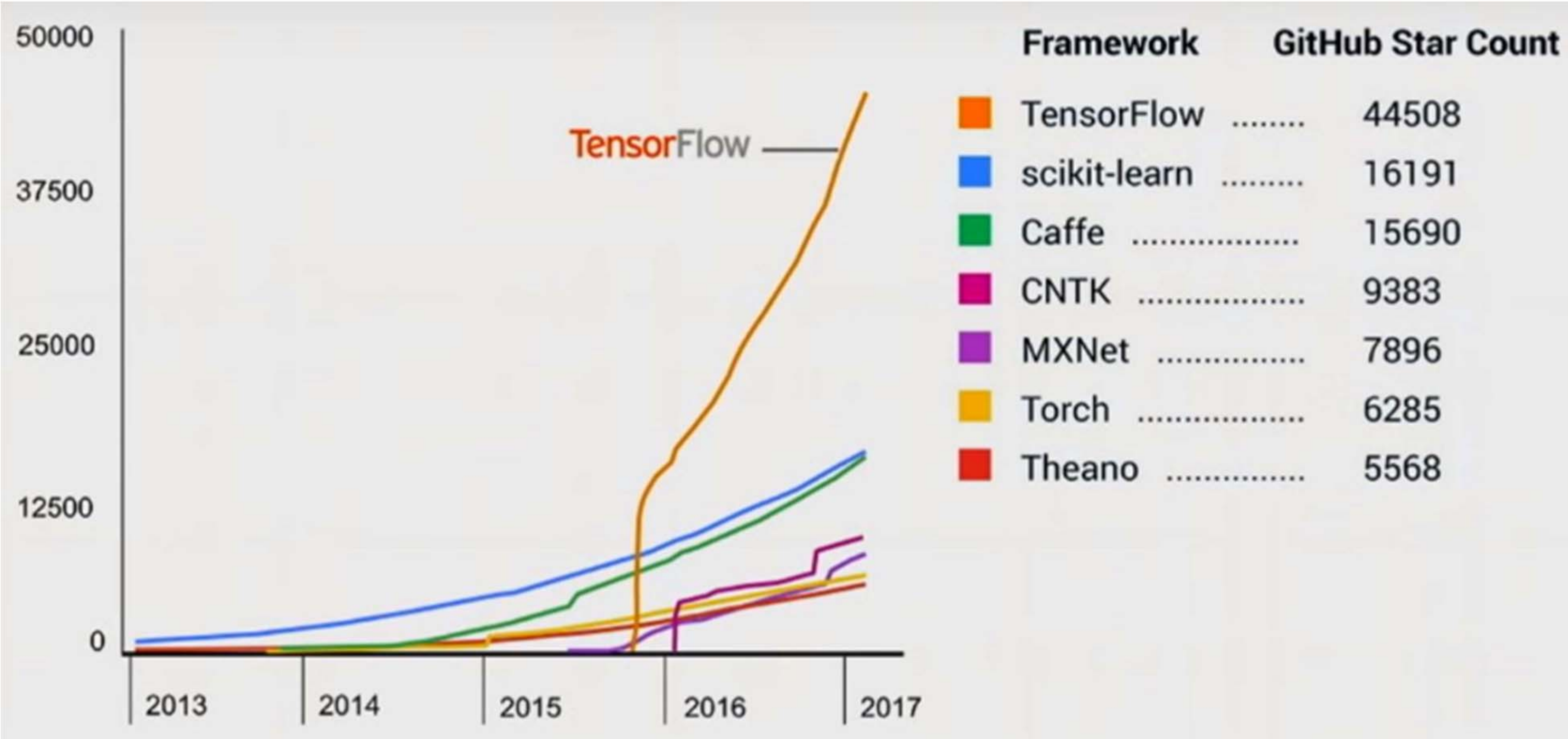
Era of deep learning is beginning.



Break through human recognition performance.

Introduction to Keras

Python framework for ANN



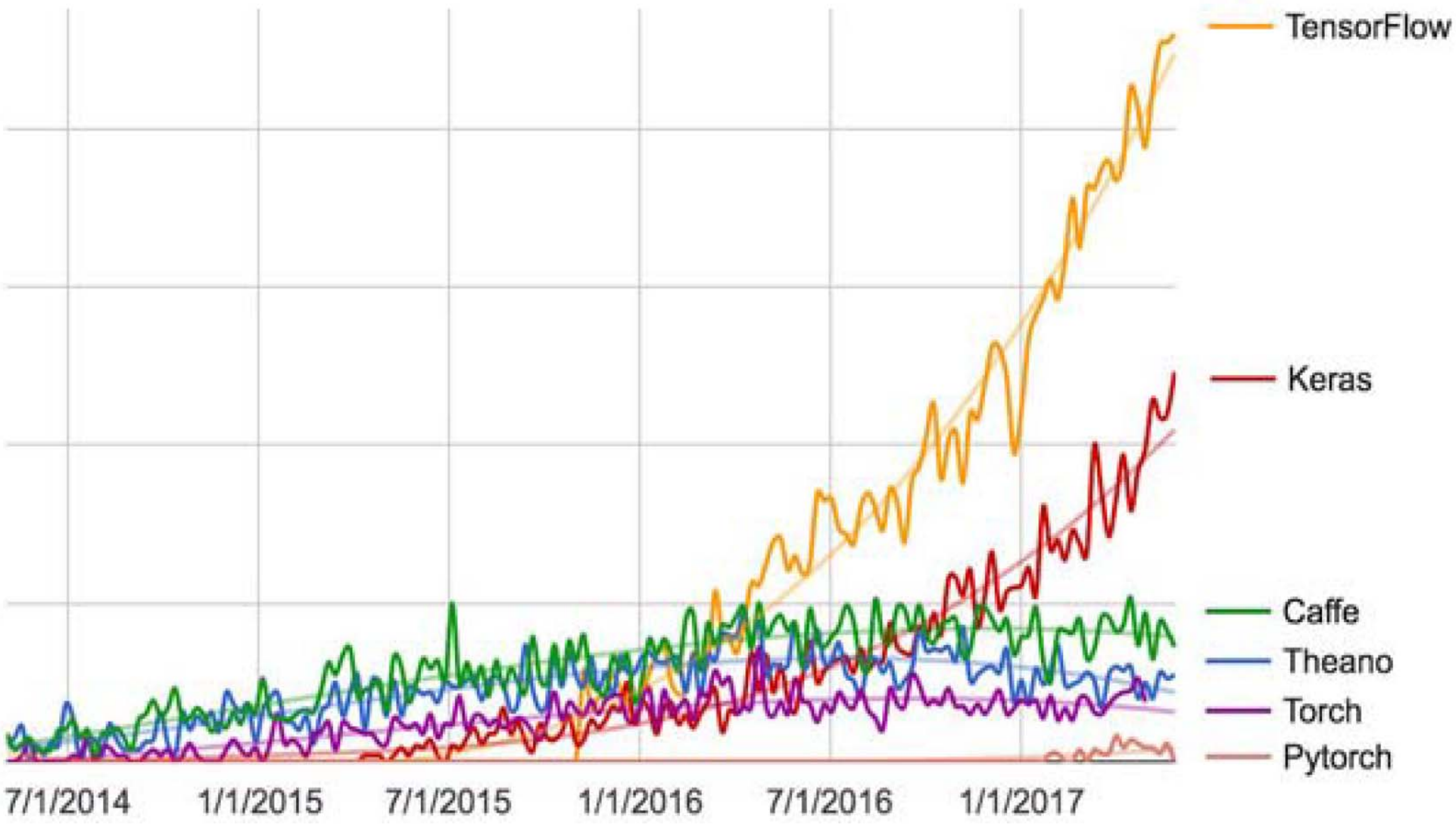
- TensorFlow 網路聲量最高
- Keras 則是支援TensorFlow的更高階函數庫(Meta Framework)，可以用很簡潔的程式碼完成一個 Neural Network 模型，非常適合入門學習。

https://ithelp.ithome.com.tw/articles/10191404

Introduction to Keras

- Keras (https://keras.io).
- **deep-learning framework** for Python that provides a convenient way to define and train almost any kind of deep-learning model
- key features:
  - It allows the same code to run seamlessly on CPU or GPU.
  - It has a **user-friendly API** that makes it easy to quickly prototype deep-learning models.
  - It has built-in support for **convolutional networks** (for computer vision), **recurrent networks** (for sequence processing), and any combination of both.
  - It supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, and so on. This means Keras is appropriate for building essentially any deep-learning model, from a **generative adversarial network** to a **neural Turing machine**.
- permissive MIT license, which means it can be freely used in commercial projects. compatible with Python from 2.7 to 3.9

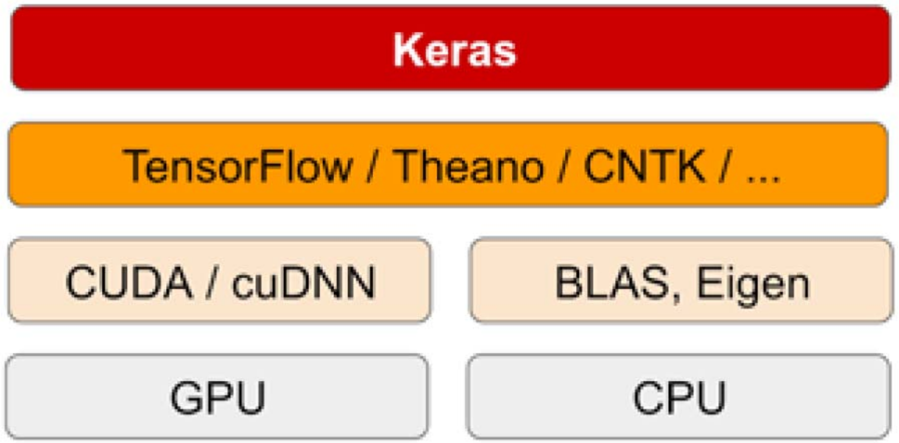
Google web search interest



https://ithelp.ithome.com.tw/articles/10191404

Keras

- Keras is a model-level library, providing high-level building blocks for developing deep-learning models.
- It doesn't handle low-level operations such as tensor manipulation and differentiation.
- Instead, it relies on a specialized, well-optimized tensor library to do so, serving as the *backend engine* of Keras.
- Rather than choosing a single tensor library and tying the implementation of Keras to that library, Keras handles the problem in a modular way



Tensorflow GPU

- Via **Tensorflow** (or Theano, or CNTK), Keras is able to run seamlessly on both CPUs and GPUs.
- When running on CPU, TensorFlow is itself wrapping a low-level library for tensor operations called **Eigen** (http://eigen.tuxfamily.org).
- On GPU, TensorFlow wraps a library of well-optimized deep-learning operations called the NVIDIA CUDA Deep Neural Network library (**cuDNN**).

安裝系統

- 安裝 Anaconda： 它包含 Python 及常用的套件(Packages)，例如NumPy、Pandas等
- 安裝 Tensorflow：可以選擇CPU或GPU版，安裝CPU版，直接在 DOS 下，輸入 **pip install tensorflow**。
- 安裝 Keras：在 DOS 下，輸入 **pip install keras**。
- 測試環境
  - **import tensorflow as tf**
    hello = tf.constant('Hello, TensorFlow!')
    sess = tf.Session()
    print(sess.run(hello))
- 那 IDE : 記事本、NodePad++、PyCharm，VS 2017 Community 版本，
- Jupyter Notebook，

Developing with Keras: a quick overview

- Typical Keras workflow
  - **Define your training data**: input tensors and target tensors.
  - **Define a network of layers** (or *model*) that maps your inputs to your targets.
    - **Sequential class** (only for linear stacks of layers, which is the most common network architecture by far)
    - ***Functional API*** (for directed acyclic graphs of layers, which lets you build completely arbitrary architectures)
  - **Configure the learning process** by choosing a loss function, an optimizer, and some metrics to monitor.
  - **Iterate on your training data by calling the fit() method of your model.**

Example

- Using the Sequential class model:

```python
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784, )))
#用 model 物件的 add 方法, 新增一個輸入為 784 維、輸出為 32 維 (等同於 unit 的數量),
#並使用 relu 啟動函數的輸入層和隱藏層 (Keras 的最開頭一層具有一般神經網路輸入層和隱藏層的功能,
#詳細請參考 3-1-1 節的小編補充, 將在後續 3-4-3 節開始實作)
model.add(layers.Dense(10, activation='softmax'))
#用 model 物件的 add 方法, 新增輸出為 10 維 (10 unit), 並使用 softmax 啟動函數的輸出層

```

Functional API

- Using the functional API:

```
input_tensor = layers.Input(shape=(784, ))
#建立一個 input_tensor 物件, 輸入層 shape為 784 維的張量
x = layers.Dense(32, activation='relu')(input_tensor)
#建立 x 物件, 使用 input_tensor 物件, 並使用 relu 啟動函數輸出一個 32 維張量的輸入層
output_tensor = layers.Dense(10, activation='softmax')(x)
#建立 model 物件, 使用 models.Model 方法, 且輸入層為 input_tensor 物件, 輸出層為 output_tensor 物件
model = models.Model(inputs=input_tensor, outputs=output_tensor)
#建立一個 output_tensor 物件, 使用 x 物件, 並使用 softmax 啟動函數輸出一個 10 維張量的輸出層
```

Setting training parameters

```
from keras import optimizers  #從 keras 套件中匯入optimizers 模組

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              #使用 model.compile 方法, 對訓練模型進行設定。
              #使用 RMSProp 優化器並將學習率定為 0.001
              loss='mse',
              #使用 mean_squared_error 損失函數
              metrics=['accuracy'])
#量測時使用 accuracy 準確度評估模型

model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
#使用 model.fit() 進行訓練, 傳入輸入資料、標籤資料 (標準答案)、
#一次訓練週期所使用的資料筆數 batch_size、和訓練週期次數 epochs
```

Layers

- The fundamental data structure, is a data-processing module that takes as input one or more tensors and that outputs one or more tensors,
- Layers have a state: the layer's *weights*, one or several tensors learned with **stochastic gradient descent**, which together contain the network's *knowledge*.
- Different layers are appropriate for different tensor formats and different types of data processing.

Example of layers

- For instance,
  - Simple vector data, stored in 2D tensors of shape **(samples, features),** is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the Dense class in Keras).
  - Sequence data, stored in 3D tensors of shape **(samples, timesteps, features)**, is typically processed by *recurrent* layers such as an LSTM layer.
  - Image data, stored in 4D tensors, is usually processed by 2D convolution layers (Conv2D).
- *Layer compatibility :* refers that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

Keras example

```
from keras import layers
layer = layers.Dense(32, input_shape=(784,))
```

- input 2D tensors where the first dimension is 784 (axis 0, the batch dimension, is unspecified).
- This layer will return a tensor where the first dimension has been transformed to be 32 (outputs).

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(32))
```

- The second layer didn't receive an input shape argument—instead, it **automatically inferred** its input shape as being the output shape of the layer that came before. (input:32, output:32)

Loss functions and optimizers: keys to configuring the learning process

- ***Loss function (objective function)***
  - The quantity that will be **minimized** during training.
  - It represents a measure of success for the task at hand.
  - For multiple output, these loss are combined to a single scalar.
- ***Optimizer***
  - Determines how the network will be **updated** based on the loss function.
  - It implements a specific variant **of stochastic gradient descent (SGD).**
- **For instance(*)**,
  - use **binary crossentropy** for a **two-class classification** problem,
  - **categorical crossentropy** for a **many-class** classification problem,
  - **Mean squared error (MSE)** for a **regression** problem,
  - **connectionist temporal classification (CTC)** for a **sequence-learning** problem

Handwritten Digit Recognition

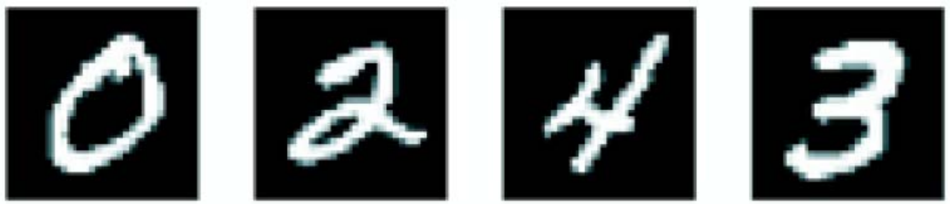
Handwritten digit recognition

- 3-nearest-neighbor classifier (stored images) = 2.4% error
- Shape matching based on computer vision = 0.63% error
- 400-300-10 unit MLP = 1.6% error
- LeNet 768-192-30-10 unit MLP = 0.9% error
- Boosted neural network = 0.7% error
- Support vector machine = 1.1% error
- Current best: virtual support vector machine = 0.56% error
- Humans ≈ 0.2% error

https://ithelp.ithome.com.tw/articles/10191404

A first look at a neural network

- A neural network that uses the Python library Keras to learn to classify handwritten digits.
- To classify grayscale images of handwritten digits (28×28 pixels) into their 10 categories (0 through 9).
- **Input: image -> output: digit (0-9)**
- The National Institute of Standards and Technology (the NIST in MNIST) MNIST dataset, a classic in the machine-learning community,
- 60,000 training images, plus 10,000 test images,
- **The "Hello World" of deep learning**.
- In machine learning, a *category* in a **classification problem** is called a *class*. Data points are called *samples*.
- The class associated with a specific sample is called a ***label***.



Loading the MNIST dataset in Keras

- **Two sets of example**
  - **train_images** and **train_labels** form the *training set*, the data that the model will learn from.
  - The model will then be tested on the *test set*, **test_images** and **test_labels**.
  - The images are encoded as Numpy arrays, and the **labels** are an array of digits, ranging from 0 to 9.
  - The images and labels have a one-to-one correspondence.

```
In [1]: import keras
        keras.__version__

        Using TensorFlow backend.

Out[1]: '2.2.4'
```

- **Code example**

```
Import keras

from keras.datasets import mnist  #從 keras 的 datasets 匯入 mnist 資料集
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
'''
用 mnist.load_data() 取得 mnist 資料集, 並存 (打包) 成 tuple
代表 ((train_images, train_labels), (test_images, test_labels)),
此 tuple 又內含兩個 tuple
'''
```

- **導入(import)**要使用的函式庫，包括 NumPy(矩陣運算)、Keras、matplotlib (繪圖)。
- 從網路載入 MNIST 資料集，請 Keras 自動分為『訓練組』及『測試組』資料，MNIST 是由 AI 大師 Yann LeCun 所建立的手寫阿拉伯數字資料集(Dataset)。

另一個範例

```
# 導入函式庫
import numpy as np
from keras.models import Sequential
from keras.datasets import mnist
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.utils import np_utils  # 用來後續將 label 標籤轉為 one-hot-encoding
from matplotlib import pyplot as plt

# 載入 MNIST 資料庫的訓練資料，並自動分為『訓練組』及『測試組』
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Example code (Jupyter Notebook)

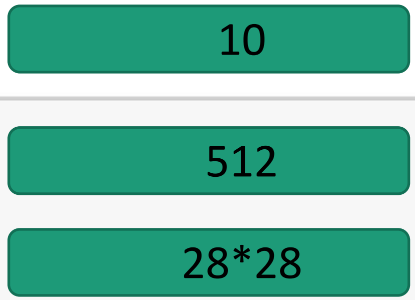
```
In [3]: train_images.shape
Out[3]: (60000, 28, 28)

In [4]: len(train_labels)
Out[4]: 60000

In [5]: train_labels            #標籤是 0-9 之間的數字, 資料型別為 uint8
Out[5]: array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

Let's have a look at the test data:

```
In [6]: test_images.shape
Out[6]: (10000, 28, 28)

In [7]: len(test_labels)
Out[7]: 10000

In [8]: test_labels
Out[8]: array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

The Network Architecture

- The core building block of neural networks is the ***layer***, a data-processing module that you can think of as a filter for data.
- Most of deep learning consists of **chaining together simple layers** that will implement a form of progressive *data distillation*.
- Dense layers, which are densely connected (also called *fully connected*) neural layers.
- The second (and **last**) layer is a 10-way *softmax* layer, which means it will return an array of 10 probability scores (**summing to 1**).




```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

- 建立最簡單的**線性模型**(Sequential)，就是一層層往下執行，沒有分叉(If)，也沒有迴圈(loop)，這裡只設一層隱藏層**(Dense)**。
- 執行模型評估，計算模型參數預測新資料了。

 # 建立簡單的線性執行的模型

$$CrossEntropy = -\sum_i (L_i \cdot \log(S_i))$$

另一個範例

```
# 建立簡單的線性執行的模型
model = Sequential()
# Add Input layer, 隱藏層(hidden layer) 有 256個輸出變數
model.add(Dense(units=256, input_dim=784, kernel_initializer='normal', activat
ion='relu'))
# Add output layer
model.add(Dense(units=10, kernel_initializer='normal', activation='softmax'))

# 編譯: 選擇損失函數、優化方法及成效衡量方式
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc
uracy'])
```

Compilation Step

- ***A loss function* 損失函數**(crossentropy)
  - How the network will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.
- ***An optimizer* 優化方法**(adam)

$$CrossEntropy = -\sum_i (L_i \cdot \log(S_i))$$

  - The **mechanism** through which the network will update itself based on the data it sees and its loss function. **(weight updating method)**
- ***Metrics to monitor during training and testing* 成效衡量方式** (accuracy)

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

DNN 處理流程

- **建立model**: 確立Input格式、要經過幾層處理、每一層要作甚麼處理，
- **確立目標及求解方法**：以compile函數定義損失函數(loss)、優化函數(optimizer)及成效衡量指標
- **訓練**：以compile函數進行訓練，指定訓練的樣本資料(x, y)，並撥一部分資料作驗證，還有要訓練幾個週期、訓練資料的抽樣方式。
- **評估(Evaluation)**：訓練完後，計算成效。

```
# 進行訓練, 訓練過程會存在 train_history 變數中
train_history = model.fit(x=x_Train_norm, y=y_TrainOneHot, validation_split=0.
2, epochs=10, batch_size=800, verbose=2)
```

- **預測(Prediction)**：經過反覆訓練，有了可信模型後，我們就可將系統上線使用了。

```
# 顯示訓練成果(分數)
scores = model.evaluate(x_Test_norm, y_TestOneHot)
```

Keras 模型類別

- **Sequential Model (順序式模型)：**
  - 就是一種簡單的模型，單一輸入、單一輸出，按順序一層(Dense)一層的由上往下執行。
  - **Sequential model 線性堆疊**
    - Input_shape:size/none
    - 2D: input_dim

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

- **Functional API：**
  - 支援多個輸入、多個輸出，

Loss Function

- 均方誤差 **MSE (mean_squared_error)** 最小平方法**(Least Square)** 的目標函數
  - 預測值與實際值的差距之平均值

$$\sum \left(\hat{y}^2 - y^2\right) / N$$

  - 變化
    - mean_absolute_error、mean_absolute_percentage_error、mean_squared_logarithmic_error
- **Hinge Error (hinge)**
  - 是一種單邊誤差，不考慮負值，適用於『支援向量機』(SVM)的最大間隔分類法(maximum-margin classification)，

$$\ell(y) = \max(0, 1 - t \cdot y)$$

Loss Function

- **Cross Entropy**

$$D = -\sum_i (L_i \cdot \log(S_i))$$

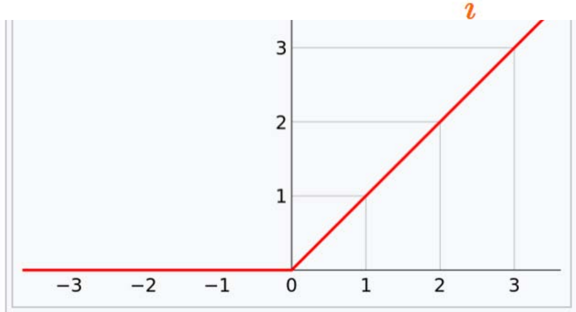
- **Categorical_crossentropy** 多分類損失函數
  - 當預測值與實際值愈相近，損失函數就愈小，反之差距很大，就會更影響損失函數的值
  - 變形
    - sparse_categorical_crossentropy
    - binary_crossentropy

$$L(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log(1-\hat{y}_n)\right]$$

Activation Functions

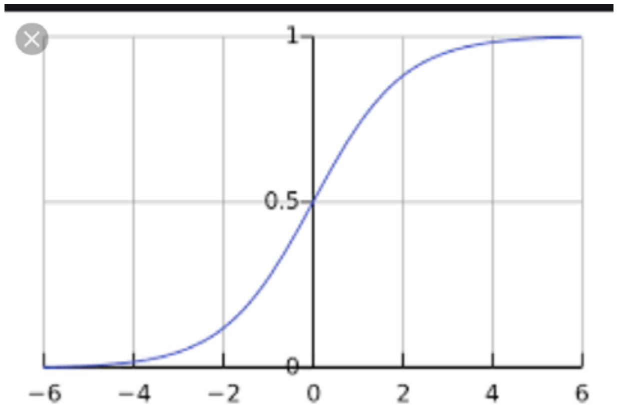
- **Relu**
  - 整流線性單位函數（Rectifi
    稱修正線性單元
  - **F(x)=max (0, x)**

$$CrossEntropy = -\sum_i (L_i \cdot \log(S_i))$$



- **Softmax [0,1]**
  - **pdf** 機率函數
  - 多分類時使用

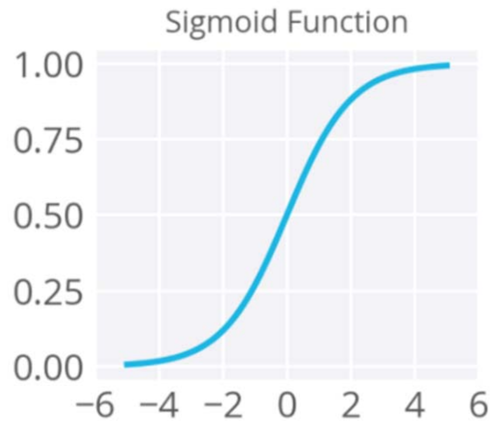
$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$



Activation Functions

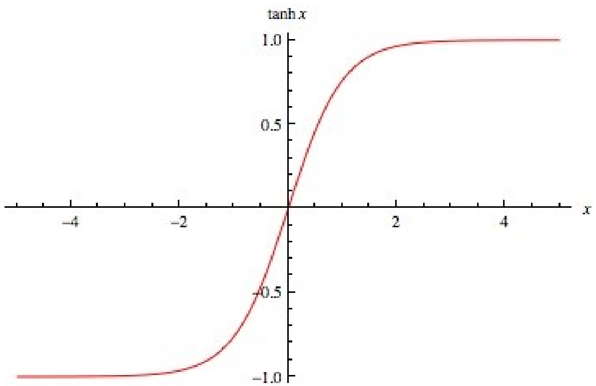
- **sigmoid**
  - 值介於 [0,1] 之間，且分布兩極化，大部分不是 0，就是 1，
  - 適合二分法。

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



- **tanh**
  - 與sigmoid類似，但值介於[-1,1]之間
  - 即傳導有負值。

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$



權重初始化 kernel_initializer

- **Kernel_initializer and bias_initializr**

```
model.add(Dense(64,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
```

- **Zeros()**：全部為0的矩陣。 //initializers.Zeros()
- **Ones()**：全部為1的矩陣。 //initializers.Ones()
- **Const()**：全部為固定常數的矩陣 initializers.Constant(**value**=0)
- **Identity**：對角線為 1 的矩陣
- **TruncatedNormal**：
  - 裁掉極端值常態分配的隨機亂數，參數為N倍標準差。
- **RandomNormal**：常態分配初始化
  - initializers.RandomNormal(mean=0.0, stddev=0.05, seed=**None**)
- **RandomUniform** ：均勻分配初始化
  - keras.initializers.RandomUniform(minval=-0.05, maxval=0.05, seed=**None**)
    下界與上界間平均分配

優化函數(Optimizer)

- 隨機梯度下降法(Stochastic Gradient Descent, SGD)
  - 就是利用偏微分，逐步按著下降的方向，尋找最佳解。
  - **Learning Rate (lr)**：
    - 逼近最佳解的學習速率，速率訂的太小，計算最佳解的時間花費較長，訂的太大，可能會在最佳解兩旁擺盪，找不到最佳解。
  - **momentum**：
    - 更新的動能，一開始學習速率可以大一點，接近最佳解時，學習速率步幅就要小一點，
    - 一般訂為 0.5，不要那麼大時，可改為 0.9。
  - **decay**：
    - 每次更新後，學習速率隨之衰減的比率。

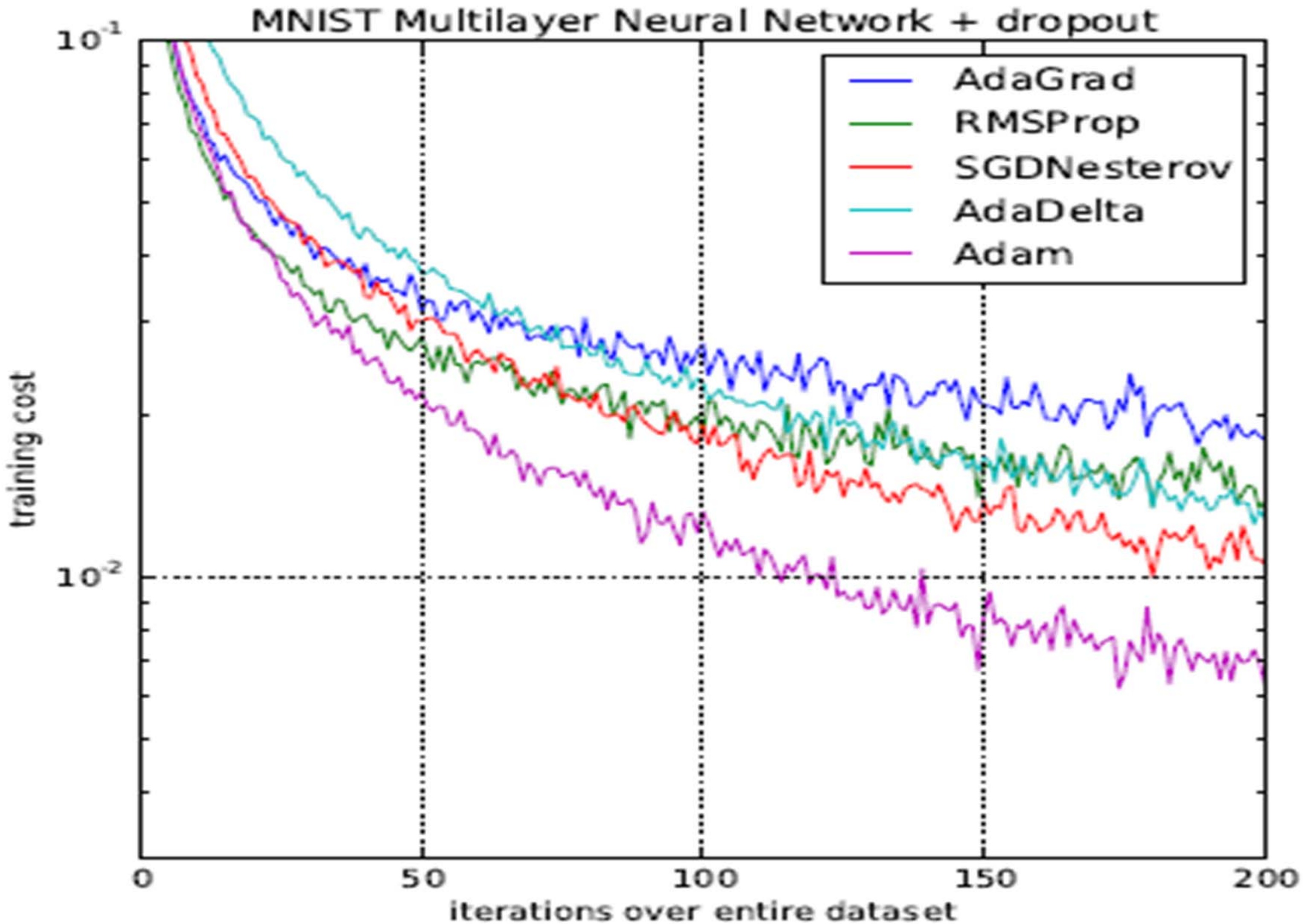
優化函數(Optimizer)

- **Adam**：一般而言，比**SGD**模型訓練成本較低
  - ***lr***：逼近最佳解的學習速率，預設值為 0.001。
  - ***beta_1***：一階矩估計的指數衰減因子，預設值為 0.9。
  - ***beta_2***：二階矩估計的指數衰減因子，預設值為 0.999。
  - ***epsilon***：為一大於但接近 0 的數，放在分母，避免產生除以 0 的錯誤，預設值為1e-08。
  - ***decay***：每次更新後，學習速率隨之衰減的比率。

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\frac{\partial L_t}{\partial W_t}$$

$$v_t = \beta_1 v_{t-1} + (1 - \beta_2)(\frac{\partial L_t}{\partial W_t})^2$$

優化函數比較



MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset

Hidden Layer and Parameters of Dense

- **Keras:**
  - 全連階層(Dense)、Activation layer、Dropout、Flatten、Reshape、Permute、RepeatVector、Lambda、ActivityRegularization、Masking。

- **Dense**
  - **output = activation(dot(input, kernel) + bias) // y = g(x * W + b)**。
    - **units**: 輸出矩陣的維數，愈大表示分類更細，擬合度愈高，雖然準確率提高，但也要防止過度擬合(Overfit)。
    - **activation**: 若未設定，即簡化為 y = x * W + b
    - **use_bias**: 是否使用偏差項(Bias)，若未設定或為 False，即簡化為 y = g(x * W)。
    - **kernel_initializer**: 權重(W)的初始值。
    - **bias_initializer**: 偏差項(Bias)的初始值。

Parameters of Dense (cont.)

- kernel_regularizer:
  - 權重(W)正規化(或稱 正則項)函數，
  - 對權重矩陣加上懲罰性函數(Penalty)，以防止過度擬合(overfit)。
- bias_regularizer:
  - 偏差項(Bias)的正規化函數。
- activity_regularizer:
  - 輸出(y)的正規化函數。
- kernel_constraint:
  - 針對權重(W)加上限制條件，
- bias_constraint:
  - 針對偏差項(Bias)加上限制條件，

Before training – data preprocessing

- Preprocess the data by **reshaping** it into the shape the network expects and **scaling** it so that all values are in the [0, 1] interval.
- Training images were stored in an array of shape (60000, 28, 28) of type uint8 with values in the **[0, 255]** interval.
- We transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1.

(# of images, Size of image)

```
train_images = train_images.reshape((60000, 28 * 28))
#reshape 是 NumPy 陣列的 method
train_images = train_images.astype('float32') / 255
                                  Change to real value within [0, 1]
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Prepare Labels

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

```
from keras.utils.np_utils import *

b = [0,1,2,3,4,5,6,7,8]

b = to_categorical(b, 9)
print(b)


[[1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

Preparing the labels and train (fit)

- train the network, by calling *fit* method—we *fit* the model to its training data:
- Two quantities are displayed during training: the loss of the network over the training data, and the accuracy of the network over the training data.
- It reaches an accuracy of **0.989 (98.9%)** on the training data.

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

Test data

- The test-set accuracy turns out to be **97.8%**—that's quite a bit lower than the training set accuracy.
- This gap between **training accuracy and test accuracy** is an example of *overfitting*: the fact that machine-learning models tend to perform worse on new data than on their training data.

```
Using TensorFlow backend.
2.3.1
(60000, 28, 28)
60000
[5 0 4 ... 5 6 8]
(10000, 28, 28)
10000
[7 2 1 ... 4 5 6]
Epoch 1/5
60000/60000 [==============================] - 3s 42us/step - loss: 0.2561 - accuracy: 0.9269
Epoch 2/5
60000/60000 [==============================] - 2s 30us/step - loss: 0.1033 - accuracy: 0.9688
Epoch 3/5
60000/60000 [==============================] - 2s 30us/step - loss: 0.0685 - accuracy: 0.9789
Epoch 4/5
60000/60000 [==============================] - 2s 29us/step - loss: 0.0497 - accuracy: 0.9846
Epoch 5/5
60000/60000 [==============================] - 2s 29us/step - loss: 0.0366 - accuracy: 0.9889
10000/10000 [==============================] - 1s 63us/step
test_acc: 0.980400025844574
```

Compile 編譯模型

```
compile(self, optimizer, loss, metrics=None, loss_weights=None,
sample_weight_mode=None, weighted_metrics=None,
target_tensors=None)
```

- **Optimizer**
  - 優化器，為預定義優化器名或優化器對象，
- **Loss**
  - 損失函數，為預定義損失函數名或一個目標函數，
- **Metrics**
  - 列表，包含評估模型在訓練和測試時的性能的指標，
  - 典型用法是**metrics=['accuracy']**

Fit 訓練參數設定

```
# 進行訓練, 訓練過程會存在 train_history 變數中
train_history = model.fit(x=x_Train_norm, y=y_TrainOneHot, validation_split=0.2, epochs=10, batch_size=800, verbose=2)
```

- x：輸入數據。如果模型只有一個輸入，那麼x的類型是numpy array，如果模型有多個輸入，那麼x的類型應當為list，list的元素是對應於各個輸入的numpy array。如果模型的每個輸入都有名字，則可以傳入一個字典，將輸入名與其輸入數據對應起來。
- y：標籤，numpy array。如果模型有多個輸出，可以傳入一個numpy array的list。如果模型的輸出擁有名字，則可以傳入一個字典，將輸出名與其標簽對應起來。
- batch_size：整數，指定進行梯度下降時每個batch包含的樣本數。訓練時**一個batch的樣本會被計算一次梯度下降**，使目標函數優化一步。
- callbacks：list，其中的元素是keras.callbacks.Callback的對象。這個list中的**回調函數**將會在訓練過程中的適當時機被調用，

Fit Parameters

- **epochs**：
  - 整數，訓練終止時的epoch值，訓練將在達到該epoch值時停止，當沒有設置initial_epoch時，它就是訓練的總輪數，否則訓練的總輪數為epochs - inital_epoch
- **verbose**：
  - **日誌顯示**，**0**為不在標準輸出流輸出日誌信息，**1**為輸出進度條記錄，**2**為每個epoch輸出一行記錄
- **validation_split**：
  - **0~1之間的浮點數**，用來指定訓練集的一定比例數據作為驗證集。
  - 驗證集將不參與訓練，並在每個epoch結束後測試的模型的指標，如損失函數、精確度等。
  - validation_split的劃分在shuffle之後，因此如果你的數據本身是有序的，需要先手工打亂再指定validation_split，否則可能會出現驗證集樣本不均勻。
- **validation_data**：
  - 形式為（X，y）或（X，y，sample_weights）的tuple，是指定的驗證集。此參數將覆蓋validation_spilt。

Evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1,
sample_weight=None)
```

```
# 顯示訓練成果(分數)
scores = model.evaluate(x_Test_norm, y_TestOneHot)
```

- x：輸入數據，與fit一樣，是numpy array或numpy array的list
- y：標籤，numpy array
- batch_size：整數，含義同fit的同名參數
- verbose：含義同fit的同名參數，但只能取0或1
- sample_weight：numpy array，含義同fit的同名參數

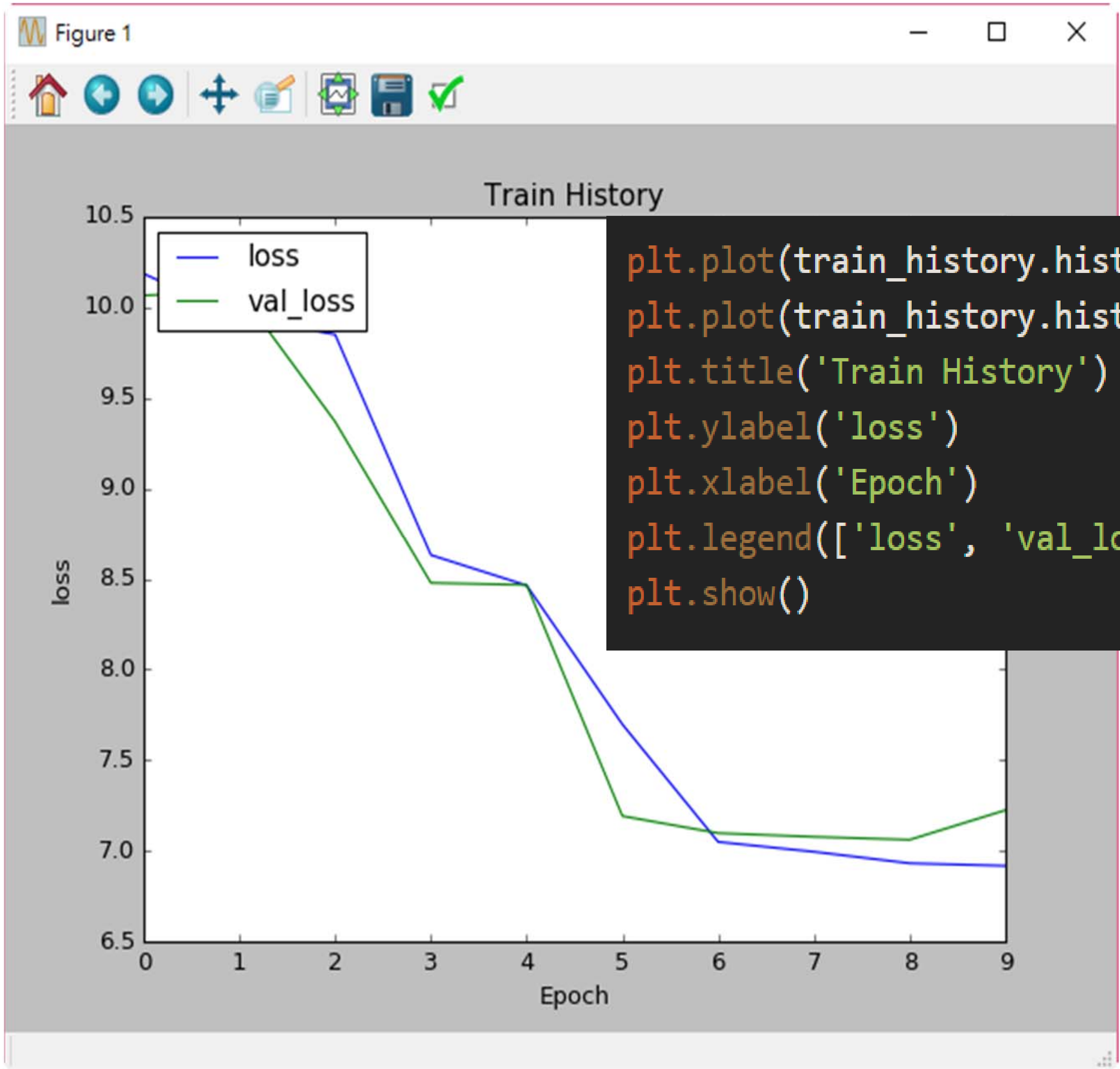
Other instructions

- **Predict**
  - **predict(self, x, batch_size=32, verbose=0)**
  - 本函數按batch獲得輸入數據對應的輸出，
  - 函數的返回值是預測值的numpy array
- **train_on_batch**
  - train_on_batch(self, x, y, class_weight=**None**, sample_weight=**None**)
  - 本函數在一個batch的數據上進行一次參數更新
  - 函數返回訓練誤差的標量值或標量值的list，與evaluate的情形相同。
- **test_on_batch**
  - test_on_batch(self, x, y, sample_weight=**None**)
- **predict_on_batch**
  - predict_on_batch(self, x)

Python code

```
# 將 training 的 input 資料轉為2維
X_train_2D = X_train.reshape(60000, 28*28).astype('float32')
X_test_2D = X_test.reshape(10000, 28*28).astype('float32')

x_Train_norm = X_train_2D/255
x_Test_norm = X_test_2D/255

# 進行訓練, 訓練過程會存在 train_history 變數中
train_history = model.fit(x=x_Train_norm, y=y_TrainOneHot, validation_split=0.
2, epochs=10, batch_size=800, verbose=2)

# 顯示訓練成果(分數)
scores = model.evaluate(x_Test_norm, y_TestOneHot)
print()
print("\t[Info] Accuracy of testing data = {:2.1f}%".format(scores[1]*100.0))

# 預測(prediction)
X = x_Test_norm[0:10,:]
predictions = model.predict_classes(X)
# get prediction result
print(predictions)
```

```
# 顯示 第一筆訓練資料的圖形，確認是否正確
plt.imshow(X_test[0])
plt.show()
```




```
plt.plot(train_history.history['loss'])
plt.plot(train_history.history['val_loss'])
plt.title('Train History')
plt.ylabel('loss')
plt.xlabel('Epoch')
plt.legend(['loss', 'val_loss'], loc='upper left')
plt.show()
```