Day 2: Introduction to AI

Part 1: Foundation of Deep learning

2021

Qualcomm

Part 3:
Foundation of Deep learning

Content

- 1.1 AI Overview
- 1.2 Quick Tour of Deep Learning
- 1.3 Foundation of Deep learning

  - 1.3.1 Perceptron & Multilayer Perceptron
  - 1.3.2 Basic neuron layers
  - 1.3.3 Loss functions
  - 1.3.4 Optimizer
  - 1.3.5 Prevent Over-fitting in Deep learning

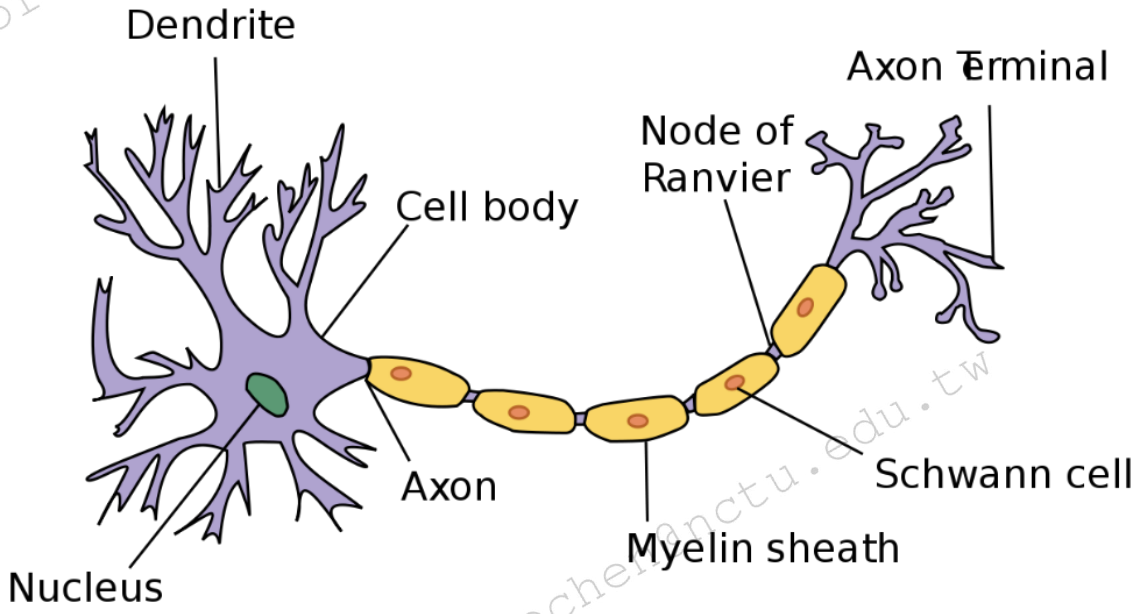
1.3 Foundation of Deep learning

◆ 1.3.1 Perceptron & Multilayer Perceptron

- Perceptron: The simplest neural network
- Multilayer Perceptron (MLP)
- From MLP to modern Deep learning

Perceptron: The simplest neural network

- A neuron (a.k.a. nerve cell), is an electrically excitable cell that communicates with other cells via specialized connections called synapses.
- Perceptron is designed with reference to a biological neuron.[1]

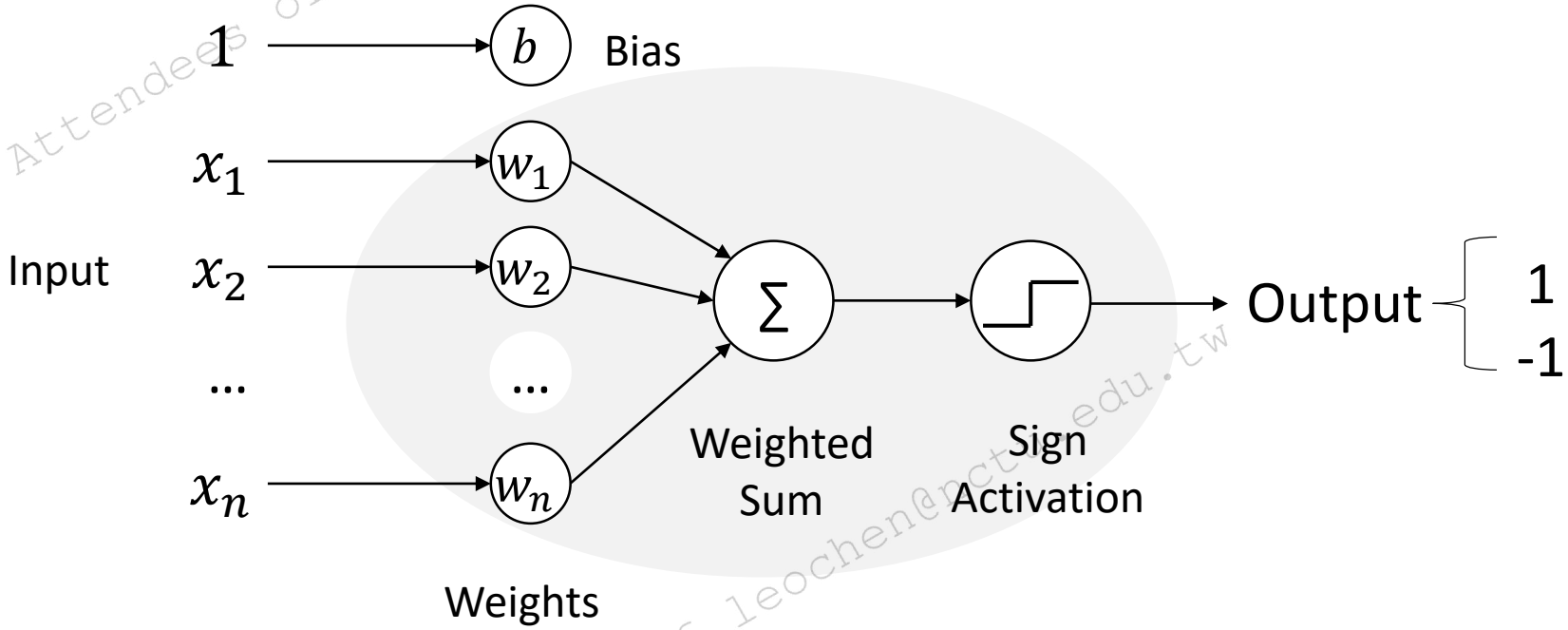


[1] McCulloch and Pitts, 1943
[2] https://en.wikipedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png

Qualcomm

Perceptron: The simplest neural network

The neuron receives inputs from other neurons. These input signals are passed through a weighted connection. The total input values received by the neuron as well as a threshold are added together. Activation function processes the sum to generate the output of a neuron.

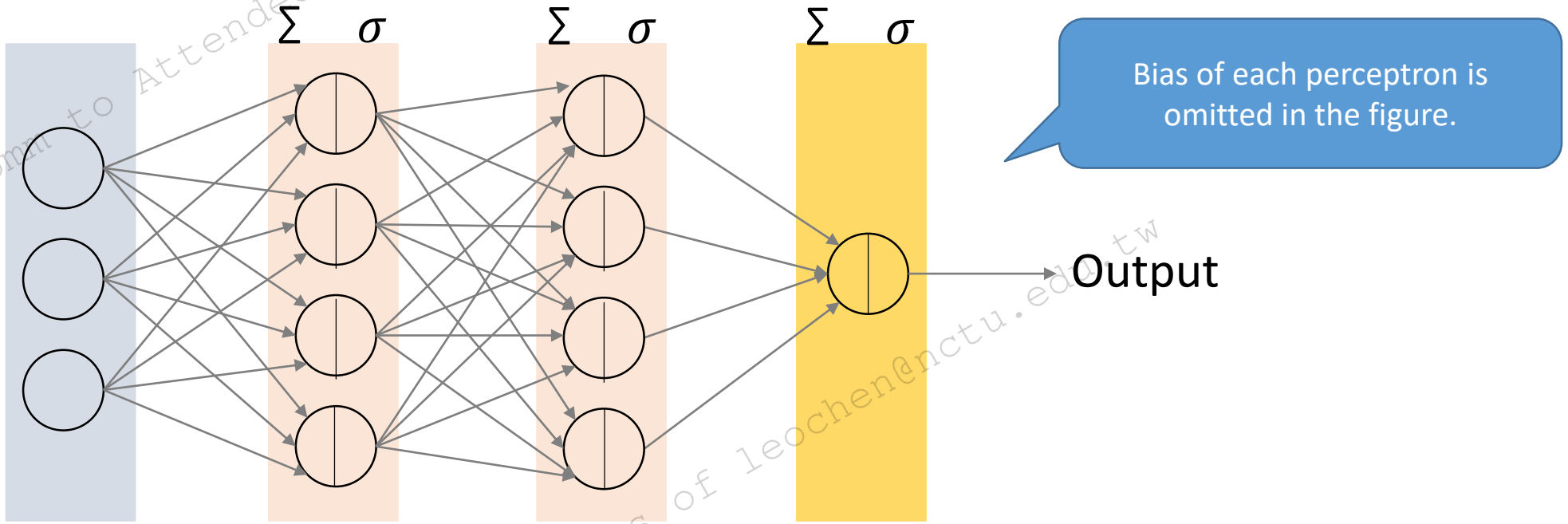


Qualcomm

Multilayer Perceptron (MLP)

If neurons are grouped together, they become a Multilayer Perceptron (a.k.a. MLP).

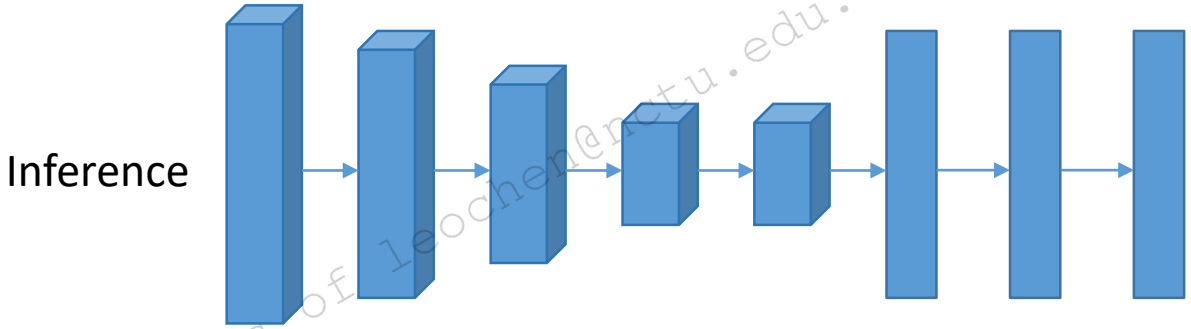
A MLP has three type of layers: an input layer, one or more hidden layers and an output layer. Each layer has one or more neurons. Except the input layer, each neuron have a non-linear activation function after it.

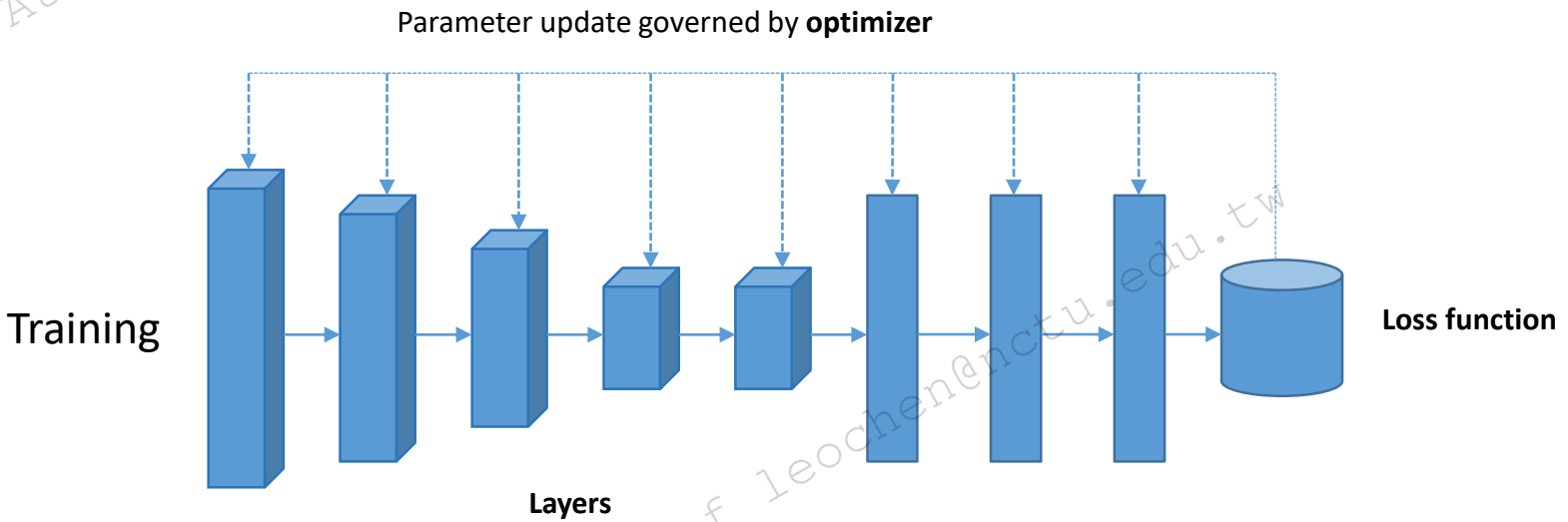


From MLP to modern Deep learning

- In the era that MLP is proposed, what we are learning is named as artificial neural networks (ANNs). MLP is seen as a kind of ANNs.
- After MLP, different kinds of neural networks were designed like Restricted Boltzmann machine (RBM), Deep belief network (DBN) and Convolutional neural network (CNN).
- In 2006, as Hinton tells it, they “rebranded” the field of neural nets with the term “Deep Learning”.
- In 2012, AlexNet (an architecture of CNNs) won the 1st prize of ImageNet Challenge 2012. Subsequently, CNN made a **revolution** in computer vision.

Workflow of CNN





Qualcomm

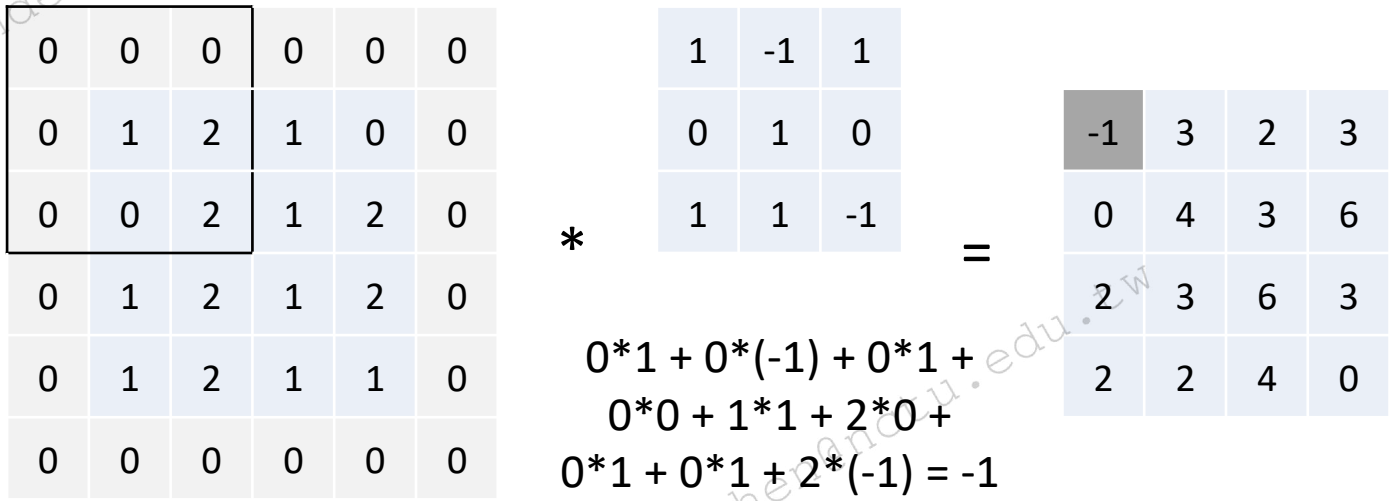
1.3 Foundation of Deep learning

◆ 1.3.2 Basic neuron layers

- Convolution layer
- Activation layer
- Normalization Layer
- Pooling layer
- Fully connected layer
- Three key ideas of CNNs

Basic neuron layers: Convolution layer

- Example

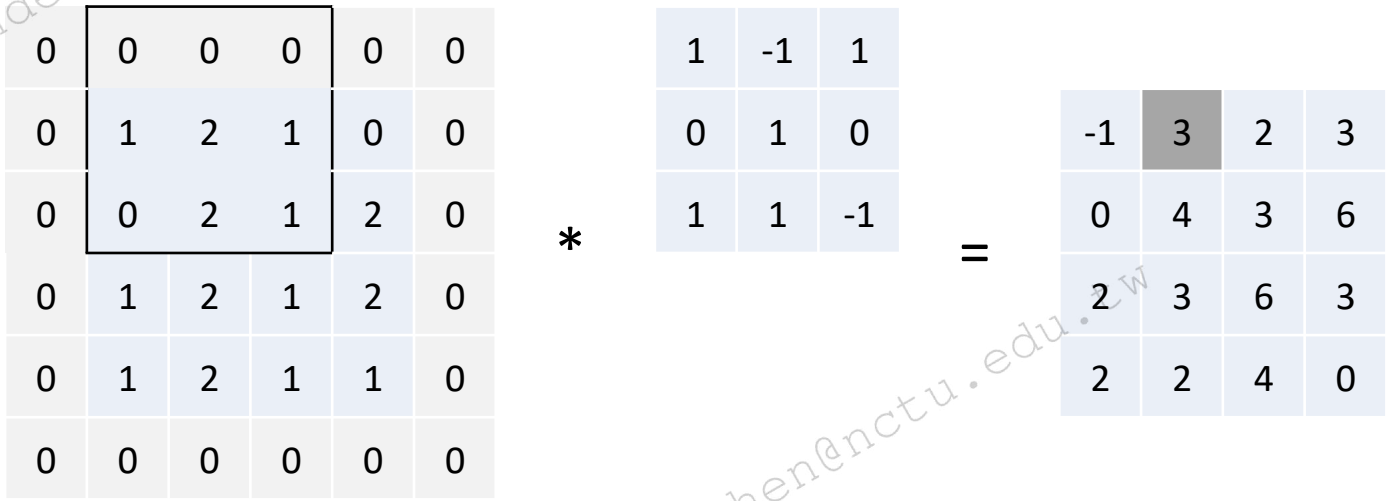


The area under the sliding window dot products the region of convolution kernel. The first number of the output map is generated.

Qualcomm

Basic neuron layers: Convolution layer

- Example

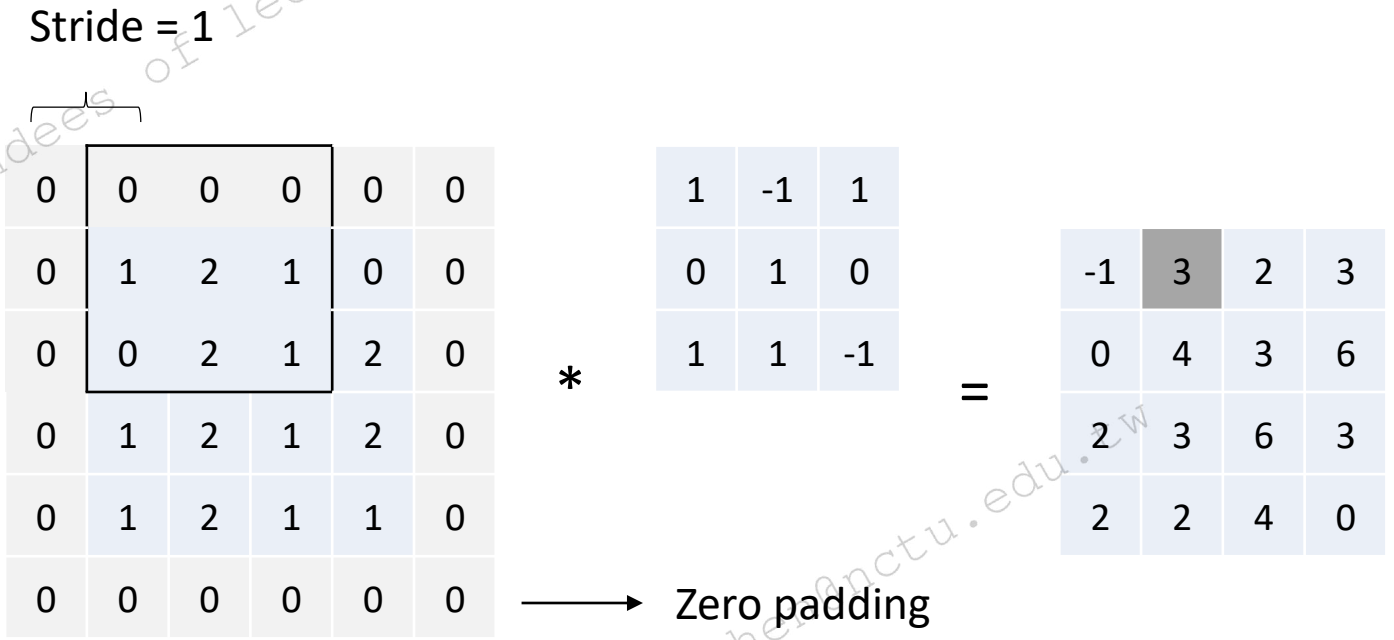


As the window slides by stride 1, the second number is also calculated.

Qualcomm

Basic neuron layers: Convolution layer

- Example



As the window slides by stride 1, the second number is also calculated.

Qualcomm

Basic neuron layers: Convolution layer

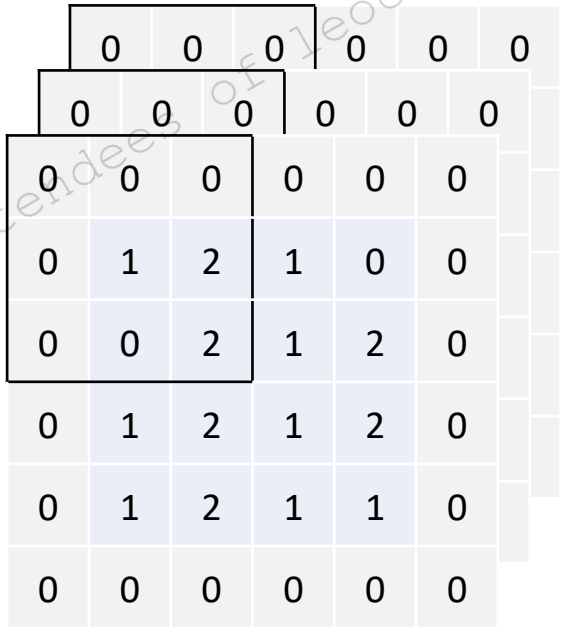
- **Padding**: We usually pad the input maps with zeros around the border, which controls the spatial size of the output volumes.
- **Stride**: the step size which the sliding window moves.

- Until now, the conv kernel and input data is both 2D. But conv kernel is 4D, and input data is 3D in practices.

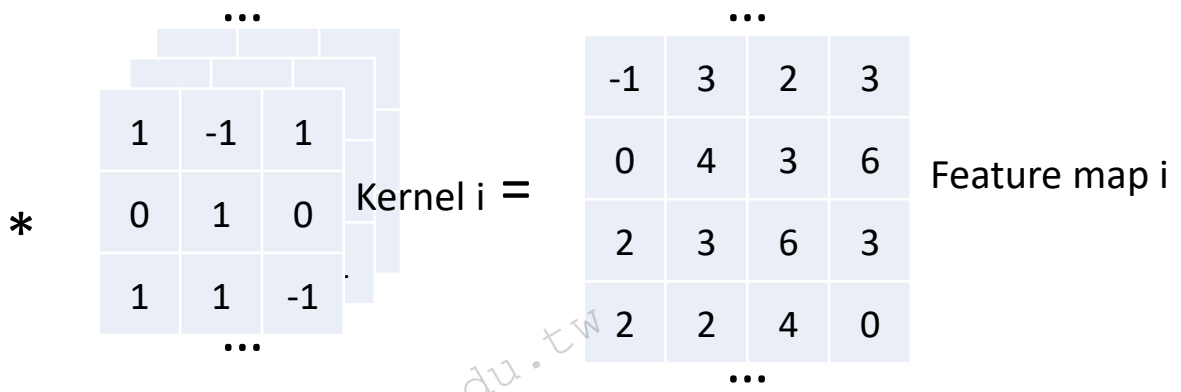
- Why?

Basic neuron layers: Convolution layer

- Example of 3D input data.



The input data of convolution layer can be raw data (RGB) and feature maps. Both of their shapes are [H, W, C].

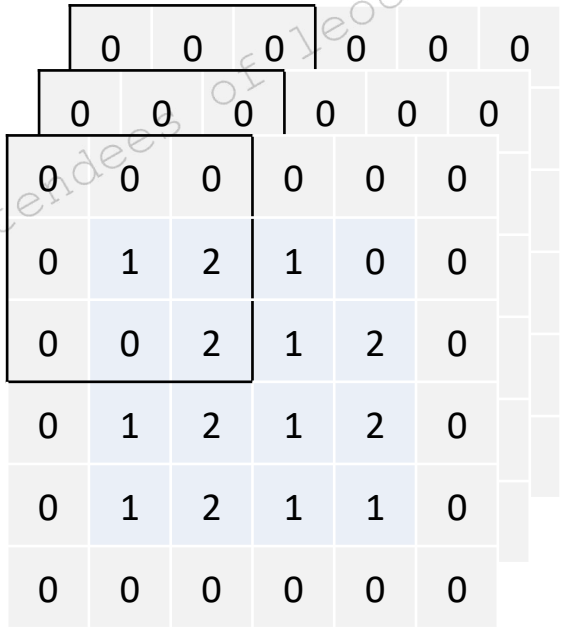


The shape of conv kernel must be the same with the input data, and the output is a 2D data of [H, W].

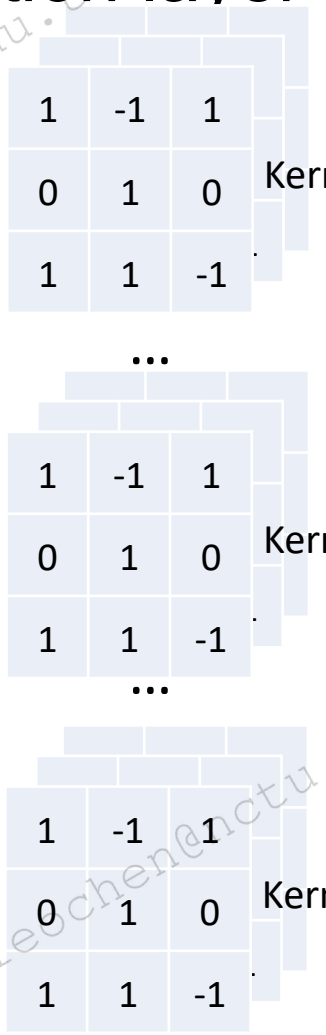
Qualcomm

Basic neuron layers: Convolution layer

- Example of 4D conv kernel



*

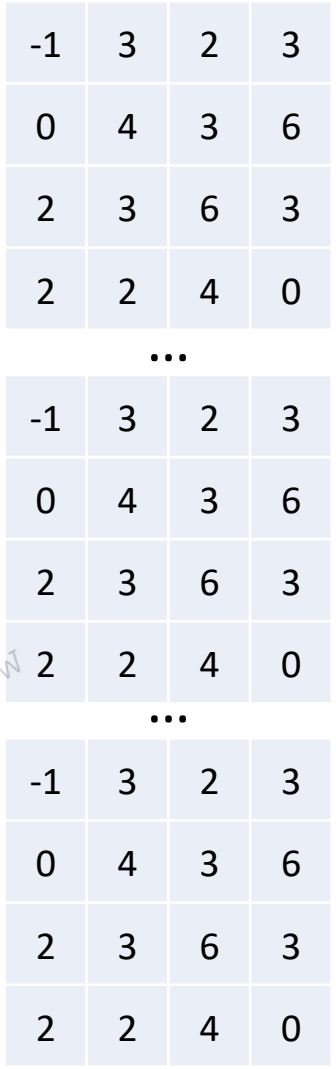


Kernel 1

Kernel i

Kernel N

=



Feature map 1

Feature map i

Feature map N

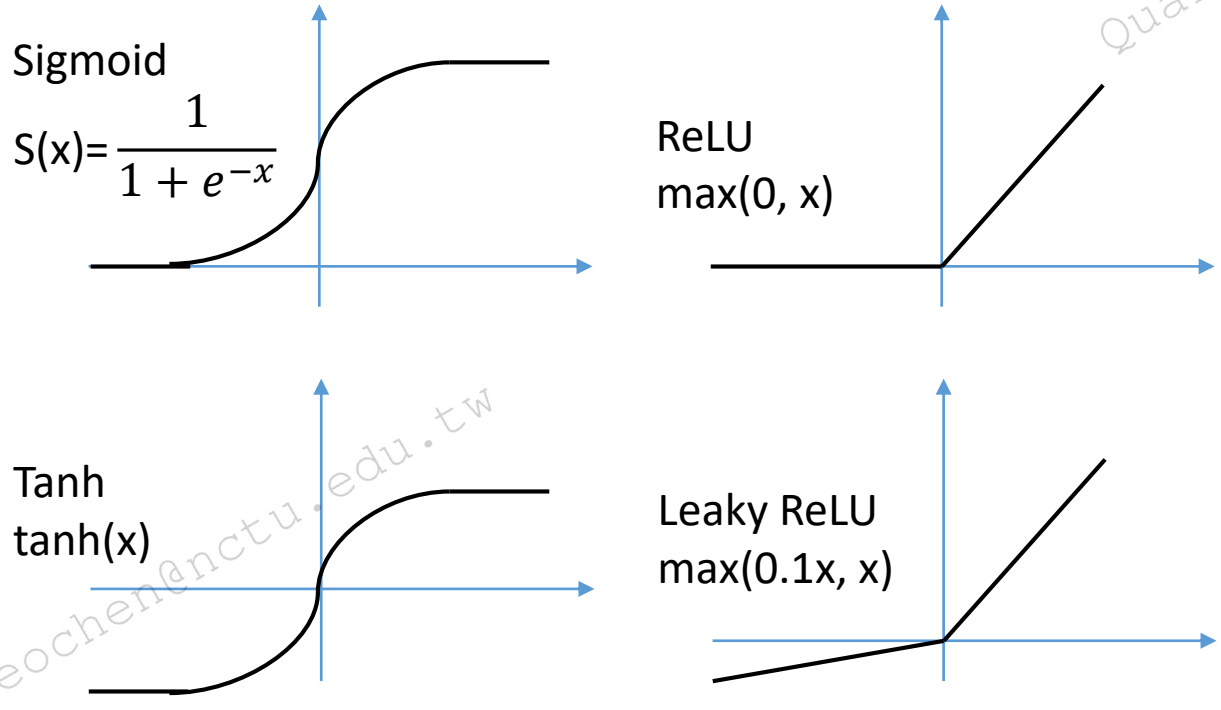
If we want to output a feature map with dimension N, we need N kernels. In the other word, the shape of conv kernel is [k, k, C, N].

Qualcomm

Basic neuron layers: Activation layer

- Non-linear activation function takes a real number as input and perform non-linear transformation as output. If the input is a vector or matrix, the mathematical function is calculated element-wise.
  - Sigmoid, mentioned earlier in MLP.
  - Tanh
  - ReLU (Rectified Linear Unit).
  - Leaky ReLU.

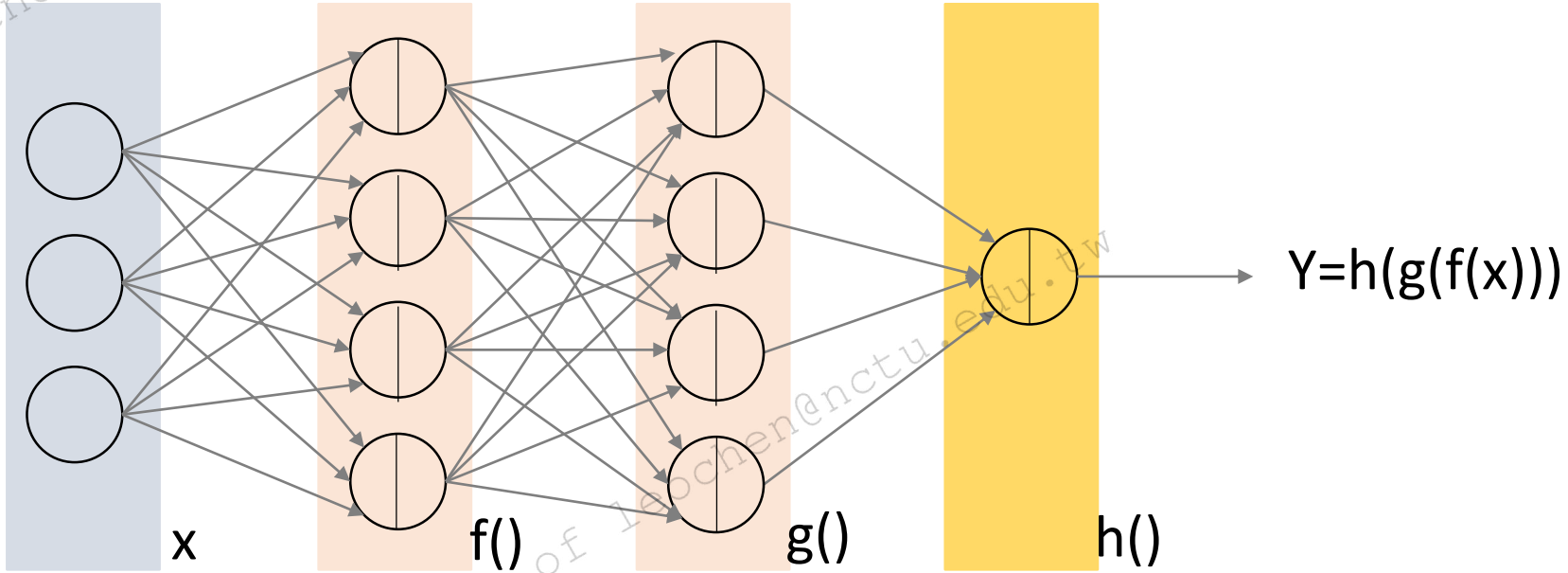
In practice, activation functions are always after a convolution layer.



Qualcomm

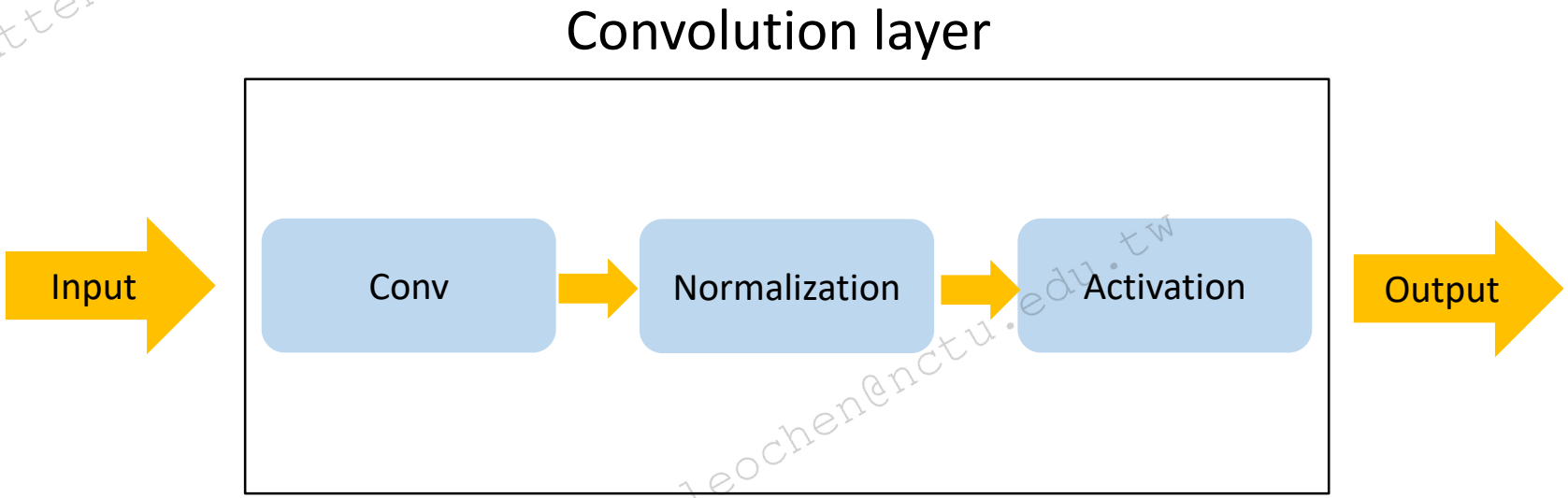
Why non-linear activation function?

- Suppose that a MLP with two hidden layers, the output is a function composition of input x, if each layer is represented as a function.
- It can be proved mathematically that a function composition of linear functions is still a linear function. Convolution is linear function essentially. So we need activation function to add non-linearity.



Basic neuron layers: Normalization Layer

- Besides activation layer, normalization layer is always after convolution layer too. We usually think of them as a whole layer.
- We will learn it in the latter content.



Basic neuron layers: Pooling layer

- In generally, a pooling layer appears between convolution layers. A pooling window is sliding on the input feature map, and calculation of max or average are performed in the pooling window. (max pooling and average pooling).
- There is no parameters in pooling layers. Hyper-parameters are pooling window size F and stride S.

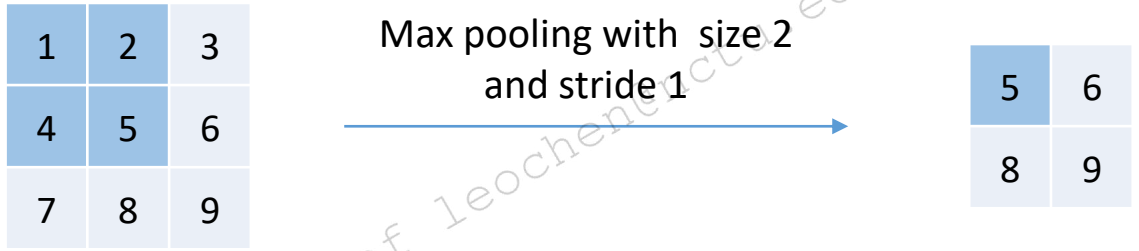
Input: [H, W, C]

Hyper-parameters: pooling window size F, stride S.

Output: [H', W', C]

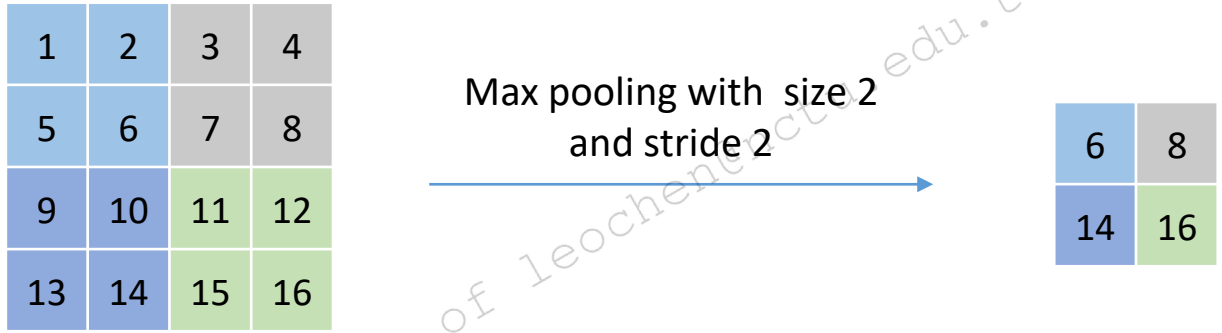
H' = (H-F)/S + 1

W' = (W-F)/S + 1



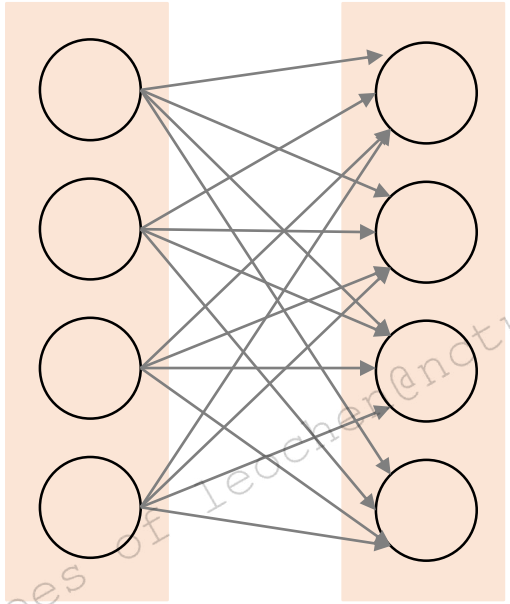
Basic neuron layers: Pooling layer

- It is worth to pointing out that only two commonly settings are used in practice:
  - F=3,S=2
  - F=2,S=2 (more common)
- So a pooling layer with stride of 2 down-samples the size by 2 along with width and height. It keeps 25% neurons and discard the others.



Basic neuron layers: Fully connected layer

- Each neuron in a fully connected layer (a.k.a FC layer, dense layer) have connections with all neurons in its previous layer.
- Layers in MLP are fully connected actually.
- FC layers are usually located at the end of the network and play a role of classifier.

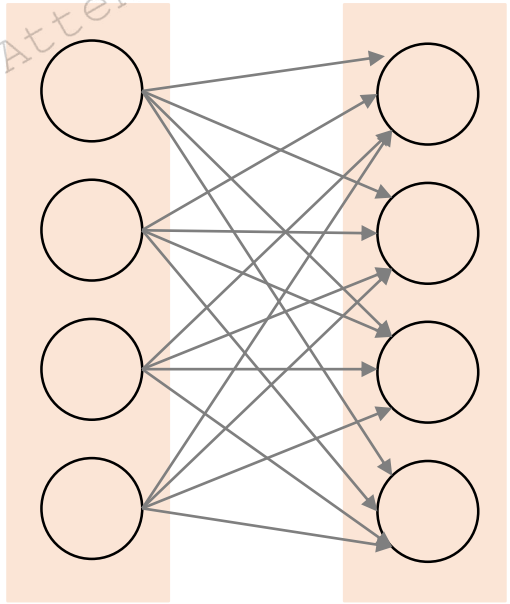


Three key ideas of CNNs

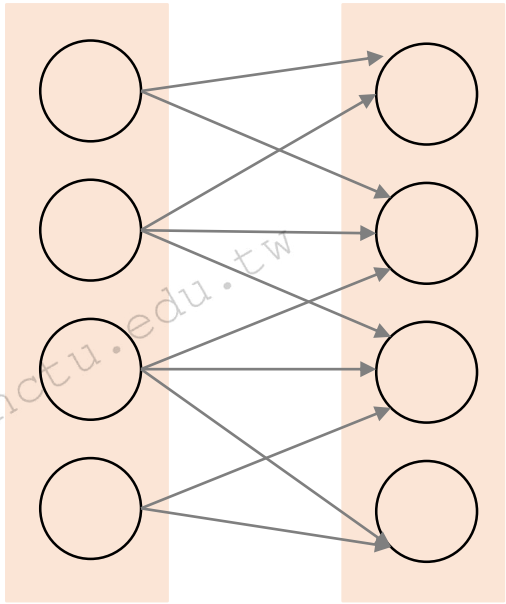
- Three key ideas of CNNs:
  - Sparse interactions (Local Connectivity)
  - Parameter sharing
  - Equivariant representations (Translation invariant)

Local connectivity

- Comparing with MLP, CNNs use much less connections. This means less parameters are needed, which reduces memory usage and computational complexity.
- For the sake of convenient and clarity, let's look as an example of 1D conv:



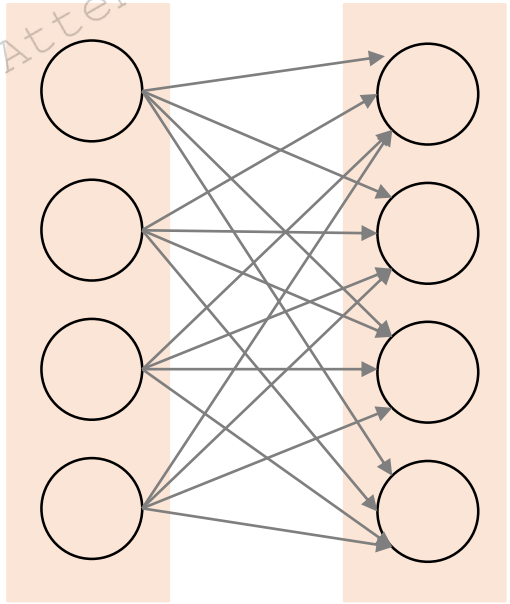
MLP



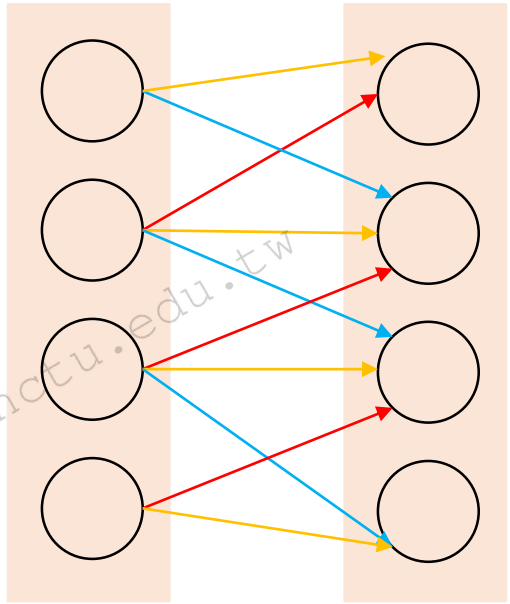
1D conv

Parameter sharing

- In MLP, suppose that there are N neurons in layer i and M neurons in layer i+1, N*M parameters are needed. Each pair of two neurons has one unique weight and connection.
- While no matter how many neurons in convolution layer and its previous, they share fixed number of parameters together. (k* k in 2D and 3 in example below.)



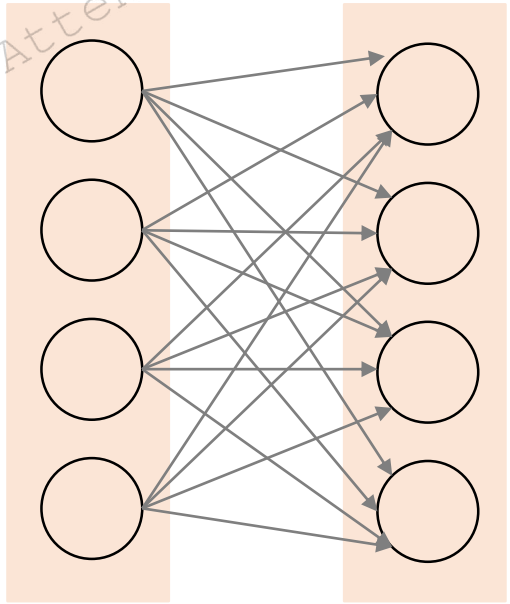
MLP



1D conv

Parameter sharing

- In MLP, suppose that there are N neurons in layer i and M neurons in layer i+1, N*M parameters are needed. Each pair of two neurons has one unique weight and connection.
- While no matter how many neurons in convolution layer and its previous, they share fixed number of parameters together. (k* k in 2D and 9 in example below.)



0	0	0	0	0	0
0	1	2	1	0	0
0	0	2	1	2	0
0	1	2	1	2	0
0	1	2	1	1	0
0	0	0	0	0	0

*

1	-1	1
0	1	0
1	1	-1

k=3

=

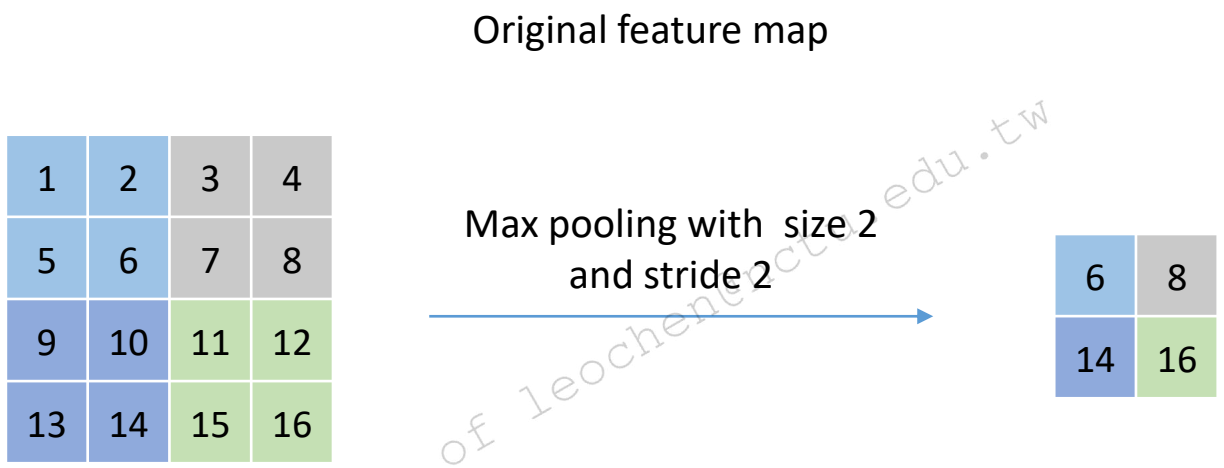
-1	3	2	3
0	4	3	6
2	3	6	3
2	2	4	0

Qualcomm

Translation invariant

- Pooling layers output maximum or average value in the pooling window, so the activation is depending on maximum value (max pooling) or most values (average pooling).

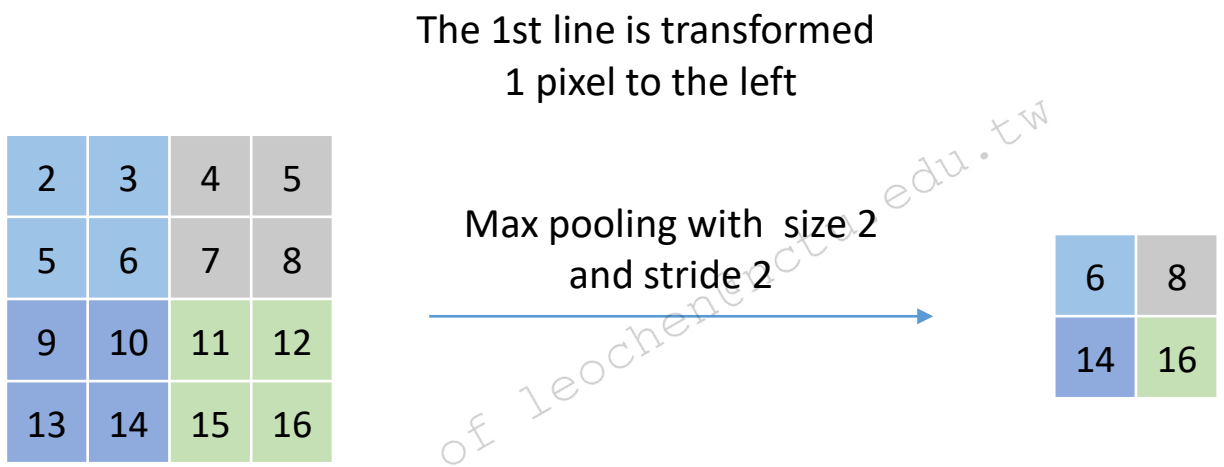
- If a small translation (or rotation, modification) is performed on the raw image, the activation map of pooling layer will keep the same or almost the same, which guarantees the prediction unchanged.



Translation invariant

- Pooling layers output maximum or average value in the pooling window, so the activation is depending on maximum value (max pooling) or most values (average pooling).

- If a small translation (or rotation, modification) is performed on the raw image, the activation map of pooling layer will keep the same or almost the same, which guarantees the prediction unchanged.

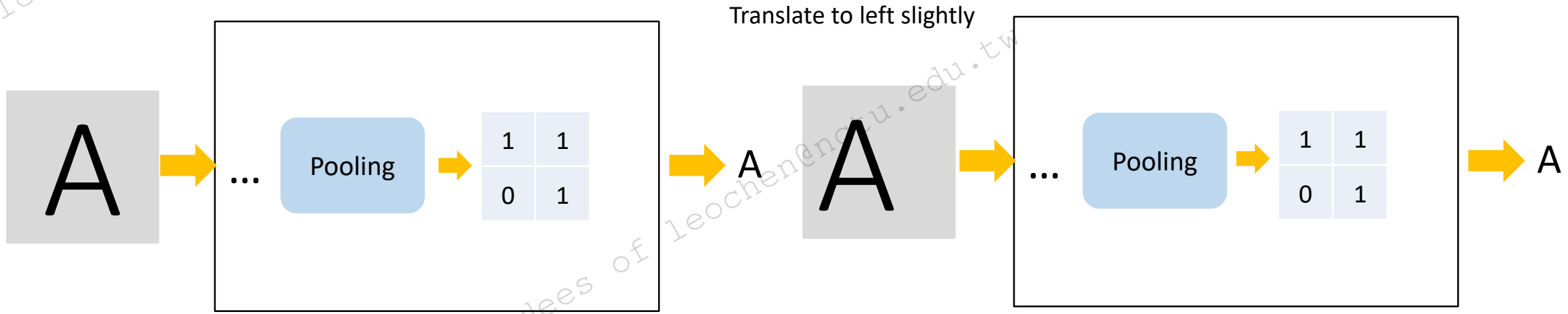


Qualcomm

Translation invariant

- Pooling layers output maximum or average value in the pooling window, so the activation is depending on maximum value (max pooling) or most values (average pooling).

- If a small translation (or rotation, modification) is performed on the raw image, the activation map of pooling layer will keep the same or almost the same, which guarantees the prediction unchanged.



Qualcomm

1.3 Foundation of Deep learning

◆ 1.3.3 Loss functions

- Cross entropy loss
- L1 loss / L2 loss / Smooth L1 loss

Loss functions

- A loss function (a.k.a cost function) is a function that maps the ground-truth value and the output value of our model into a real number intuitively representing the ability that out model fits training data.
- At the beginning of training, the parameters in the network is randomly initialized, so the predictions are also random. Apparently, the loss value are large.
- As the training continues, the parameters are automatically adjusted to minimize loss.
- If the loss is small enough and the predictions on training data are very close to the label, which means that the model predicts the training data well, then we want to how it performs on unseen data (Validation and Test set).

- So training stage is the optimization of parameters to minimize a loss function.

Classification - Cross entropy loss

- Cross entropy loss is the most commonly used loss in deep learning for classification.
- In information theory, the cross entropy between two probability distributions over measures the average number of bits needed to identify an event drawn from the set. In other word, it presents the similarity between two probability distributions.
- Ground truth probability vector use one-hot encodes, the position of 1 is the index of the category of the input image. The output value is also a probability vector.

Categories	**Index**	**Ground truth**
Cat	0	[1, 0, …, 0, 0] -> cat
Dog	1	[0, 1, …, 0, 0] -> dog
…	…	…
Person	N-1	[0, 0, …, 0, 1] -> persons.

Predicted value

Image 1: [0.01, 0.85, 0.01, …, 0.09]
Image 2: [0.9, 0.01, 0.03, …, 0.02]
…

Classification - Cross entropy loss

1 Cross entropy loss

For discrete probability distributions p and q with the same input number N,

$$H(p,q) = -\sum_{i=1}^{N} p_i \log q_i$$

In classification task, p is the label, and q is predicted probabilities vector. It should be noted that, classification is in general single-label classification, which means:

$$p = [0, 0, ..., 1, 0, ..., 0]$$

So the formula above can be simplified to:

$$H = -\log(q_n)$$

where $n \in \{0, 1, 2, ..., N-1\}$ is the ground truth.

Example 1:

If $p = [1, 0, 0, 0]$, $q = [0.01, 0.85, 0.05, 0.09]$

$$H = -\log(q_n) = -[1 * log(0.01) + 0 * log(0.85) + 0 * log(0.05) + 0 * log(0.09)] = 2$$

Example 2:

If $p = [1, 0, 0, 0]$, $q = [0.9, 0.01, 0.07, 0.02]$

$$H = -\log(q_n) = -1 * log(0.9) = 0.0457$$

Qualcomm

Regression - L1 loss / L2 loss / Smooth L1 loss

L1 loss (Least absolute deviations)

Formula:

$$L_1(x) = |x|$$

For a batch of examples with batch size N, label $y = \{y^{(i)}\}$ and logits $\hat{y} = \{\hat{y}^{(i)}\}$

$$\mathcal{L} = \frac{1}{N}\sum_{i}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|_1$$

L2 loss (Least square errors)

Formula:

$$L_2(x) = x^2$$

For a batch of examples with batch size N, label $y = \{y^{(i)}\}$ and logits $\hat{y} = \{\hat{y}^{(i)}\}$

$$\mathcal{L} = \frac{1}{N}\sum_{i}^{N} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Huber loss (Smooth L1)

Formula:

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

For a batch of examples with batch size N, label $y = \{y^{(i)}\}$ and logits $\hat{y} = \{\hat{y}^{(i)}\}$

$$L_{huber} = \begin{cases} \frac{1}{2N}\sum_{i}^{N} \left(y^{(i)} - \hat{y}^{(i)} \right)^2 & \text{if } |(y^{(i)} - \hat{y}^{(i)})| < \delta \\ \frac{1}{N}\sum_{i}^{N} \left(|y^{(i)} - \hat{y}^{(i)}| - 0.5 \right)^2 & \text{otherwise} \end{cases}$$

What's their difference?

Qualcomm

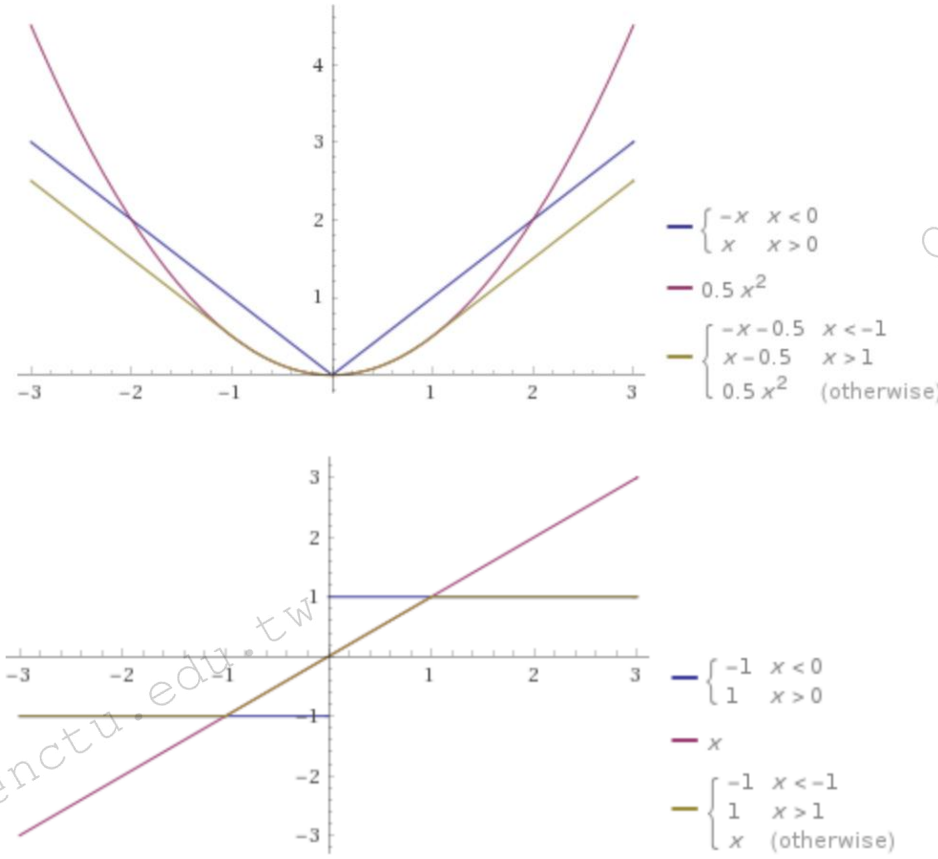
Regression - L1 loss / L2 loss / Smooth L1 loss

Cons of L2:
Too sensitive to outliers.
In the early stages of training, the predicted label is a random value and has a big difference with ground truth, so |x|(=$|y - \hat{y}|$) >> 0 (e.g. 1000), the gradient is large too, the model is hard to converge.

Cons of L1:
In the late stages of training, the model may stuck in sub-optimal value and can't reach to the optimal value.
If x is near to zero, gradient is still 1. A large gradient makes it difficult to converge near the optimal value. (Despite sub-optimal value)

Pros of Smooth L1:
If |x| is large, the maximum of gradient is 1(or -1), which guarantees stable convergence;
If |x| is close to 0, the gradient also turns to a small value, a smaller step size is helpful to converge to the optimal.

Smooth L1 loss is **recommended** in regression.



Qualcomm

1.3 Foundation of Deep learning

◆ 1.3.4 Optimizer

- SGD
- Other optimizers
- Backpropagation: the core to train deep models

Optimizer

Review:

Loss is a measure of the difference between predictions and the ground truth.

In the other word, minimizing the loss function by adjust model parameters to predict values close to the desired results (labels), is exactly training model.

Optimization is minimizing the loss function, and optimizer is the algorithm that performs optimization.

1 Cross entropy loss

For discrete probability distributions p and q with the same input number N,

$$H(p,q) = -\sum_{i=1}^{N} p_i \log q_i$$

In classification task, p is the label, and q is predicted probabilities vector. It should be noted that, classification is in general single-label classification, which means:

$$p = [0, 0, ..., 1, 0, ..., 0]$$

So the formula above can be simplified to:

$$H = -\log(q_n)$$

where $n \in \{0, 1, 2, ..., N-1\}$ is the ground truth.

Example 1:

If $p = [1, 0, 0, 0], q = [0.01, 0.85, 0.05, 0.09]$

$$H = -\log(q_n) = -[1 * log(0.01) + 0 * log(0.85) + 0 * log(0.05) + 0 * log(0.09)] = 2$$

Example 2:

If $p = [1, 0, 0, 0], q = [0.9, 0.01, 0.07, 0.02]$

$$H = -\log(q_n) = -1 * log(0.9) = 0.0457$$

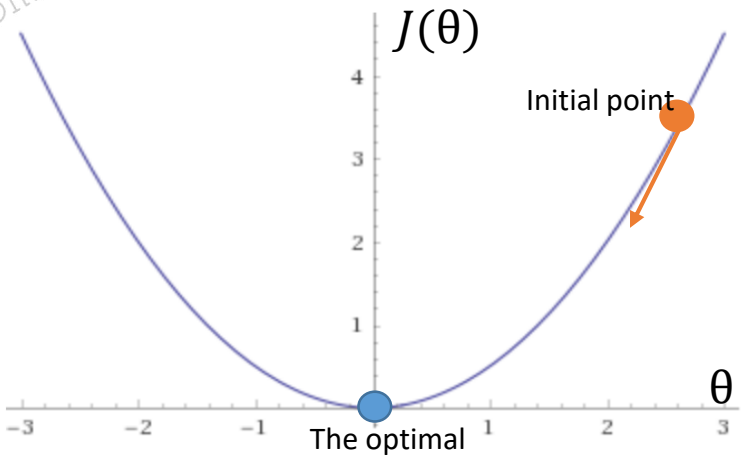
Gradient Descent

- How to find the optimal?

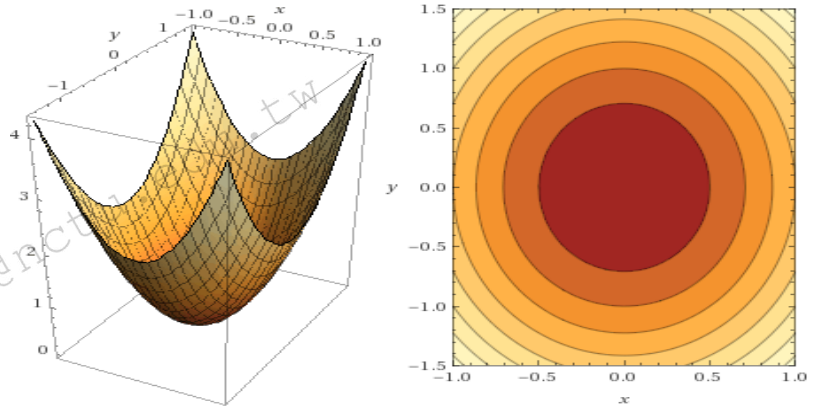
Imagination matters!

We can visualize diagram up to 1D and 2D, but neural networks often have hundreds of thousands of parameters or more.

1 parameter



2 parameters



Qualcomm

Gradient Descent

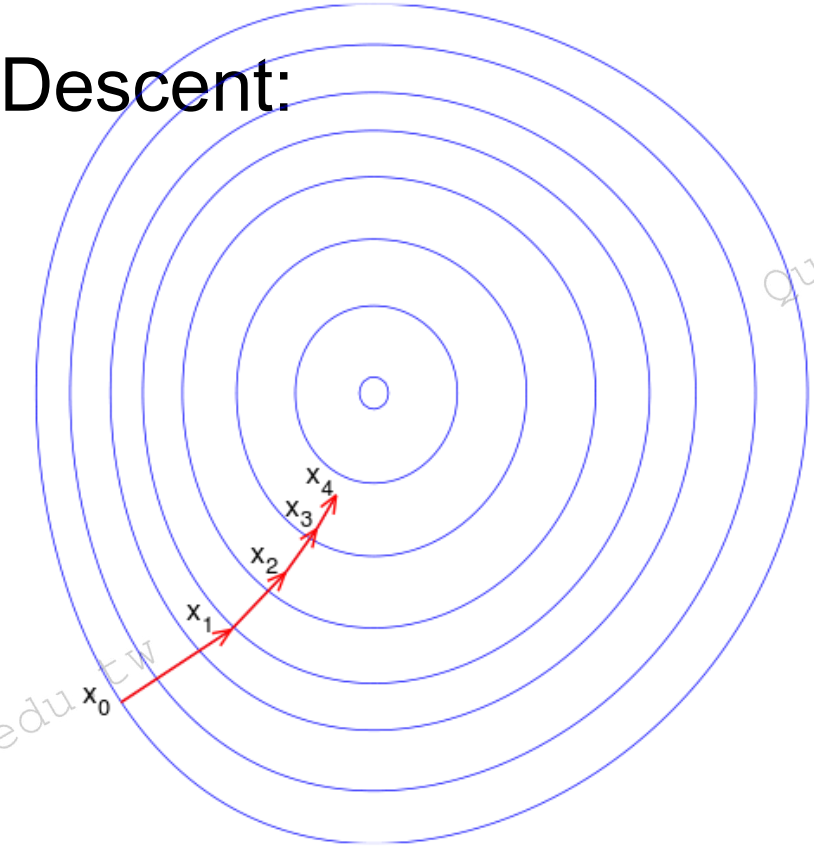
- Let's stylized the process of Gradient Descent:

If a multi-variable function $J(x)$ is differentiable in a neighborhood of a point $\mathbf{a}$, $J(x)$ decreases fastest if one goes from a in the direction of the negative gradient of F.

It follows:

$$\theta_{n+1} = \theta_n - \eta \nabla J(\theta_n)$$

for $\eta \in \mathbb{R}_+$ small enough, then $J(\theta_{\mathbf{n}}) \geq J(\theta_{\mathbf{n}+1})$.



https://en.wikipedia.org/wiki/File:Gradient_descent.svg

Qualcomm

Batch gradient descent

As mentioned in `Loss functions`, the loss on the whole dataset is the average of loss of all instances; Correspondingly, the gradient on the entire dataset is the average of gradients of all instances.

$$\theta_{n+1} = \theta_n - \gamma \nabla J(\theta_n)$$

where $\nabla J(\theta_n)$ is the average gradient on the dataset.

Pros

1. The global optimal (For convex function)

Cons

1. Calculate the gradient on entire dataset for only one update.

> The local minimum value of the convex optimization problem is the global minimum.

2. Expensive in time if the dataset is too large to storage in memory.
3. Failed to online learning. (Add new examples)

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Stochastic gradient descent

In contrast, `Stochastic gradient descent` use each instance one by one randomly to update parametes.

$$\theta_{n+1} = \theta_n - \gamma \nabla J\left(\theta_n; x^{(i)}; y^{(i)}\right)$$

where $\nabla J\left(\theta_n; x^{(i)}; y^{(i)}\right)$ is the gradient of a random instance.

Pros

1. Fast
2. Friendly to online learning

Cons

1. May not the global optimal (Even for convex function)
2. The randomness of optimization direction is too high

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Mini-batch gradient descent

Mini-batch gradient descent finally merges between the two above, taking into account the advantages of both.

$$\theta_{n+1} = \theta_n - \eta \nabla J \left(\theta_n; x^{(i;i+m)}; y^{(i;i+m)} \right)$$

where $\nabla J \left(\theta_n; x^{(i;i+m)}; y^{(i;i+m)} \right)$ is the average gradient on the mini-batch, and m is batch size.

pseudocode

- Choose an initial vector of parameters w and learning rate η .
- Repeat until an approximate minimum is obtained:
  - Randomly shuffle examples in the training set.
  - For $i = 1, 2, ..., n$, do:
    - $w := w - \eta \nabla J_i(w)$.

Pros

1. Fast
2. Friendly to online learning

By default, Stochastic gradient descent (SGD) means **Mini-batch gradient descent**.

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

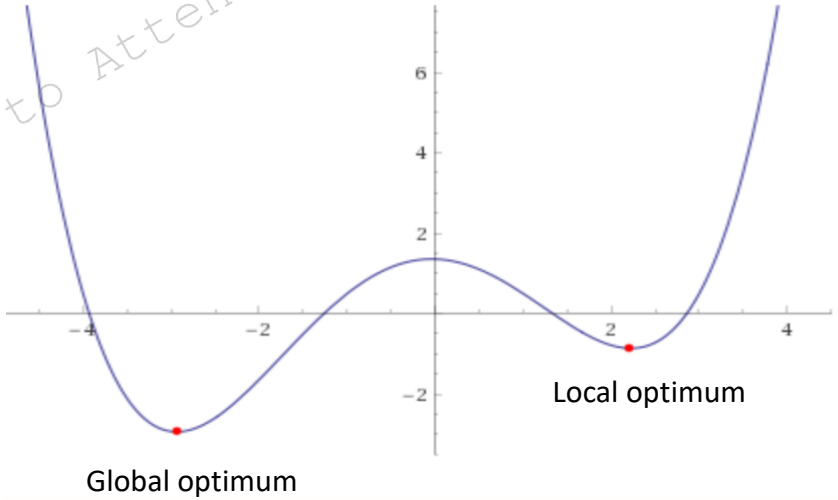
An overview of gradient descent optimization algorithms, arxiv 1609.04747

Qualcomm

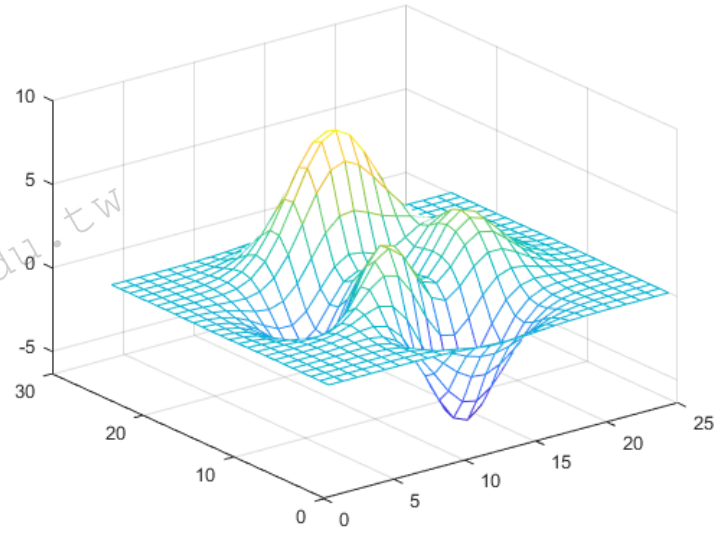
Limitations of Gradient Descent

- **Local optimum**: a solution that is optimal (either maximal or minimal) within a neighboring set of candidate solutions.

1 parameter



2 parameters

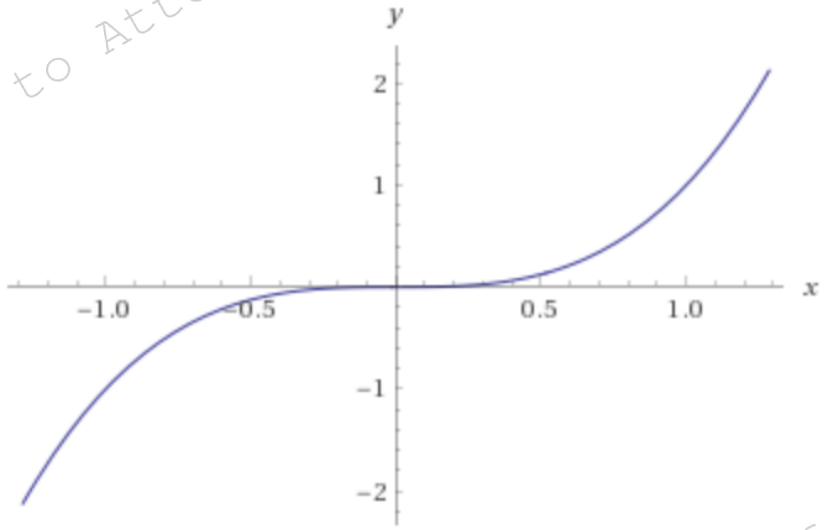


Qualcomm

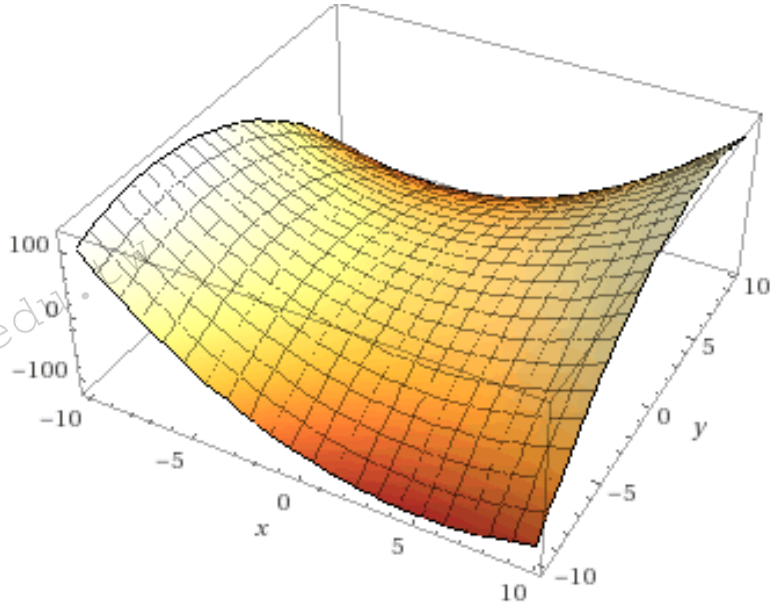
Limitations of Gradient Descent

- **Saddle point**: the derivatives in orthogonal directions are all zero, but which is not a local optimum of the function.

1 parameter



2 parameters



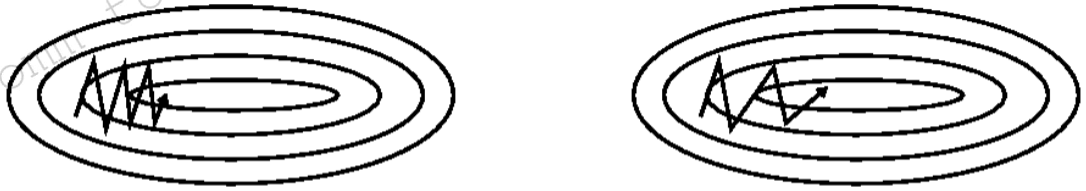
Qualcomm

Other optimizers

- There are many variants of SGD, we selected a few of them to illustrate.
  - Momentum: Momentum is from the analogy to momentum in physics.
  - AdaGrad: Per-parameter adaptive learning rate methods

Momentum

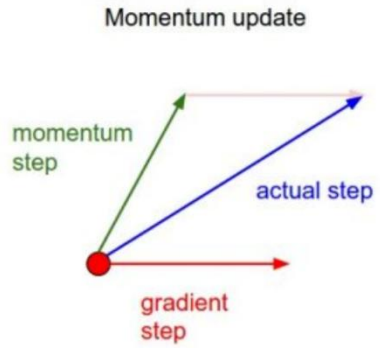
Look as examples below, SGD oscillates across the slopes of the ravine, hesitating along the start point towards the local optimum.



(a) SGD without momentum (b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

Momentum add a fraction α of the update vector in the last update, to the current update value.



$$\Delta w := \alpha \Delta w - \eta \nabla Q_i(w)$$
$$w := w + \Delta w$$

```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

An overview of gradient descent optimization algorithms, arxiv 1609.04747

Qualcomm

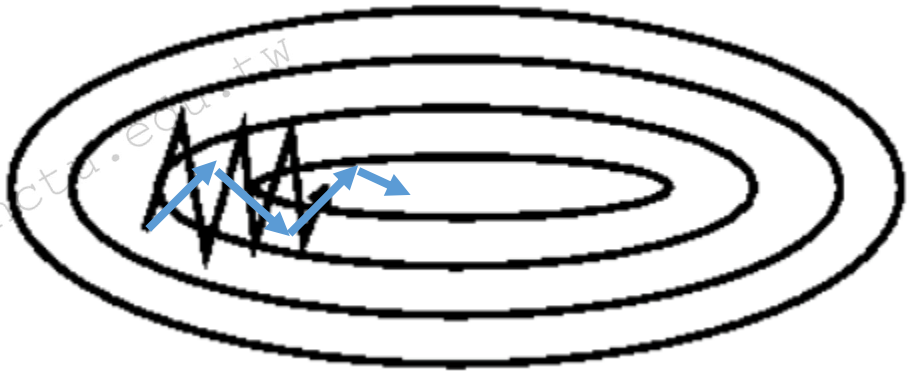
AdaGrad (adaptive gradient algorithm)

Let's re-visit the former example: why SGD oscillates across the slopes of the ravine? Because component of vector along the y axis is larger than that along the x axis!

What if they were roughly equal?

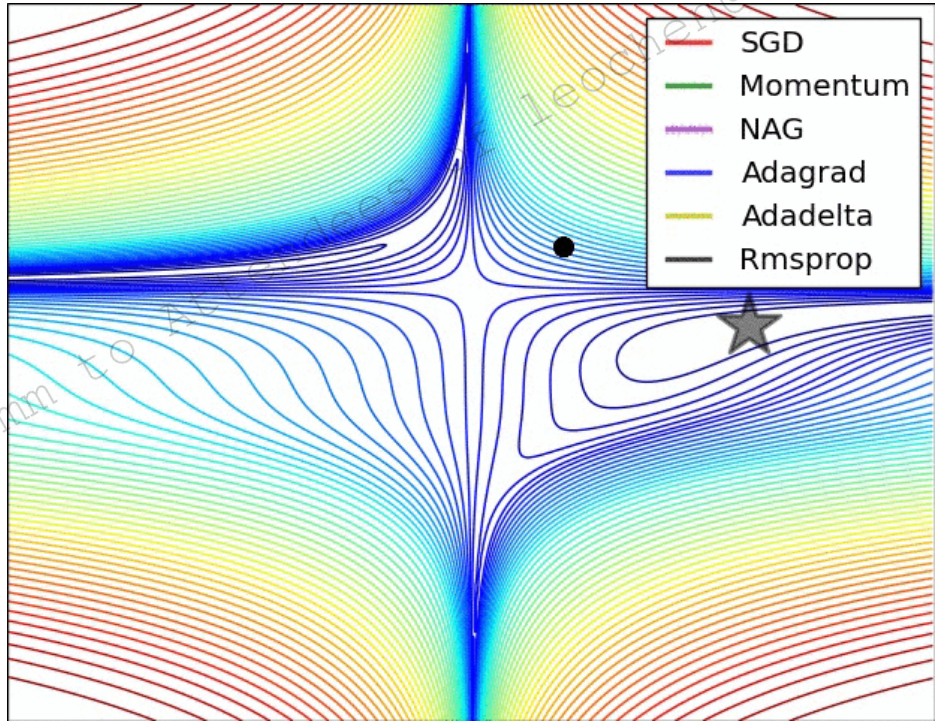
AdaGrad is used to normalize the parameter update step, element-wise, which means to increase the learning rate for sparser parameters and to decreases the learning rate for ones that are less sparse.

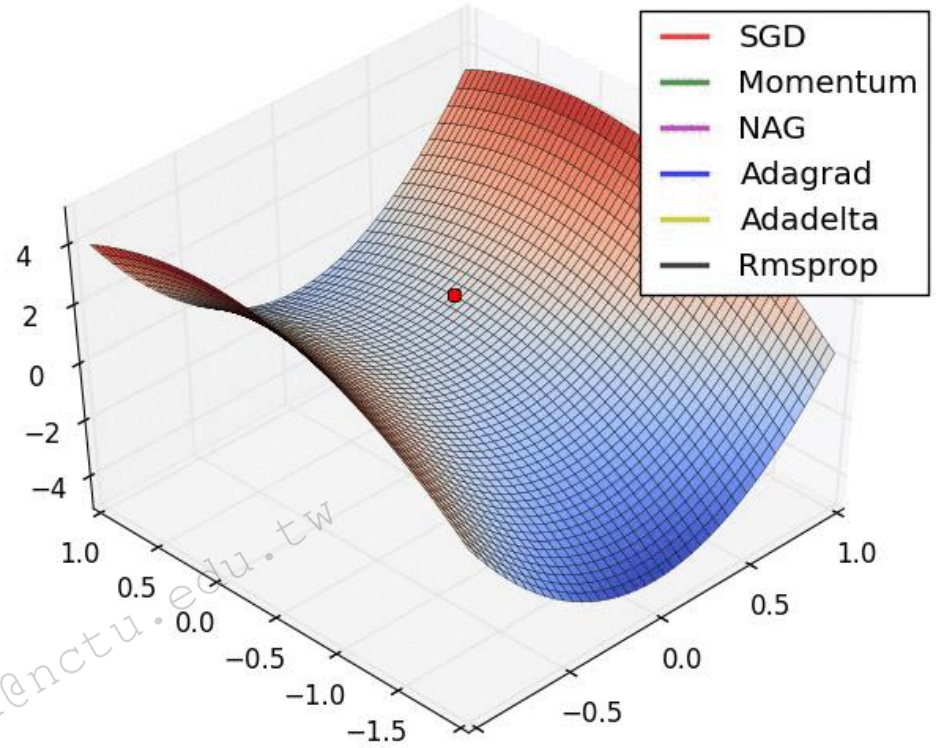
```
# Assume the gradient dx and parameter vector x
# cache is the accumulated squared gradient
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```



Qualcomm

Comparison of different optimizers



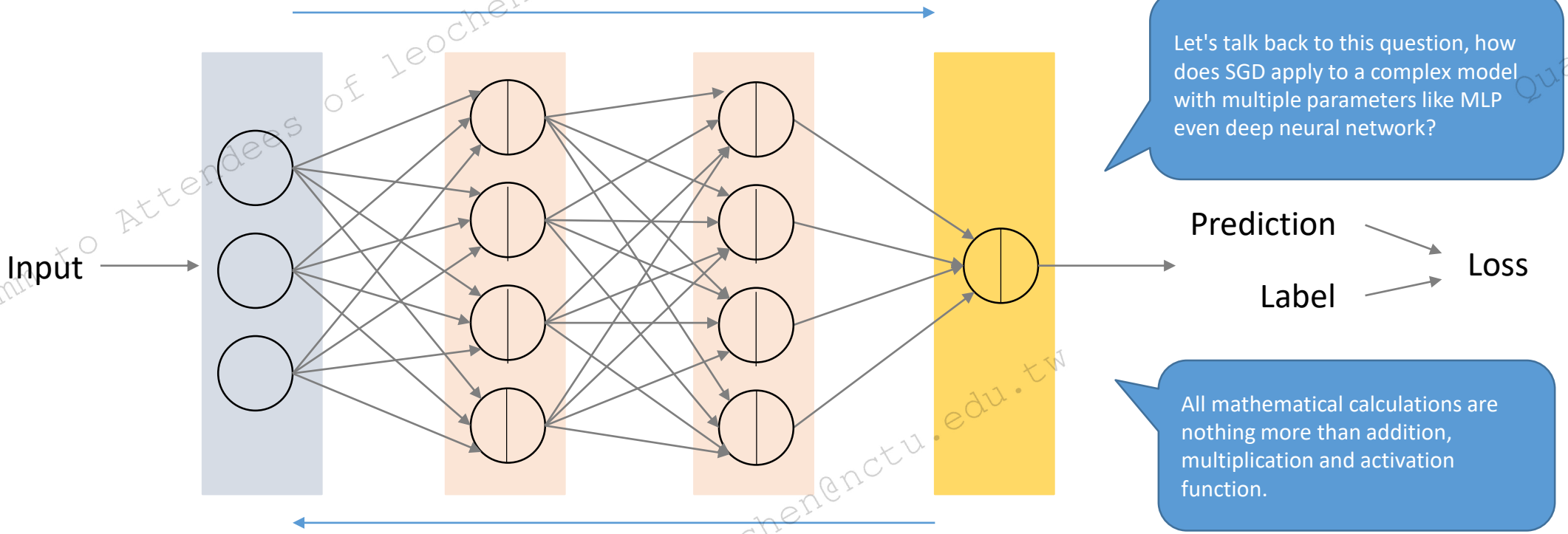


(Source: Stanford class CS231n, MIT License, Image credit: Alec Radford)

Qualcomm

Backpropagation: the core to train deep models

Propagation: calculate prediction (during test stage) and loss $J(\theta)$ (during training stage) from the front to the back.

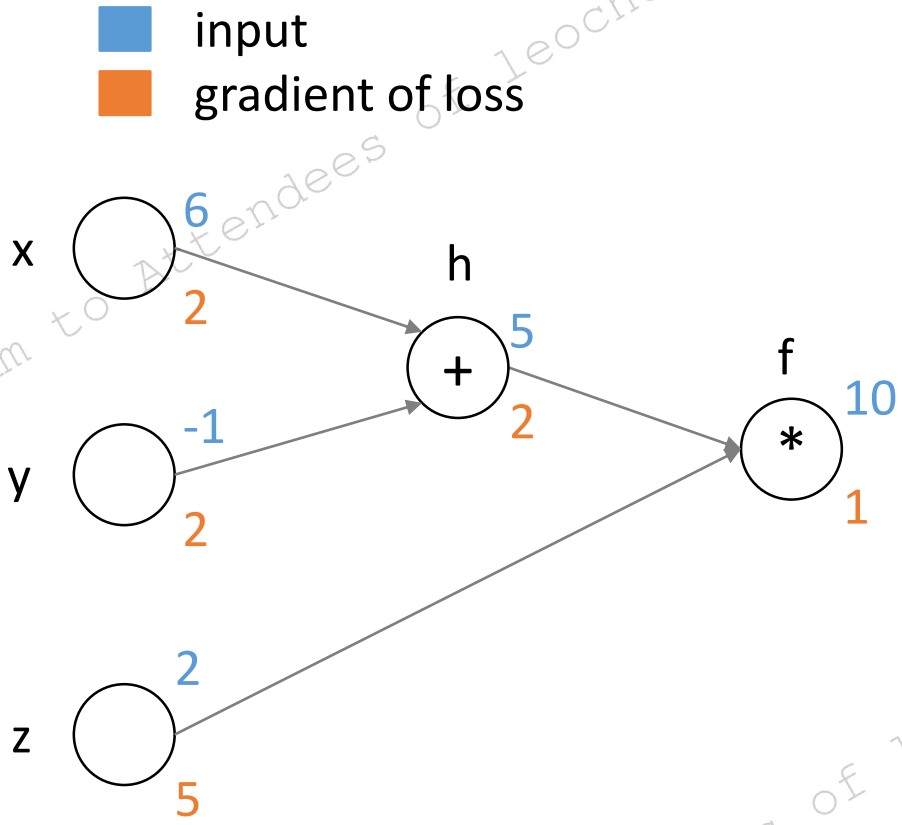


Backpropagation: calculate partial of Loss with respect to θ_i: $\frac{\partial J(\theta)}{\partial \theta_i}$, from the back to the front.

Qualcomm

Backpropagation: the core to train deep models

- Let's start with a simple example:



Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial h} \cdot \frac{\partial h}{\partial x}$$

Another form:

$$f'(x) = f'(h(x)) \cdot h'(x)$$

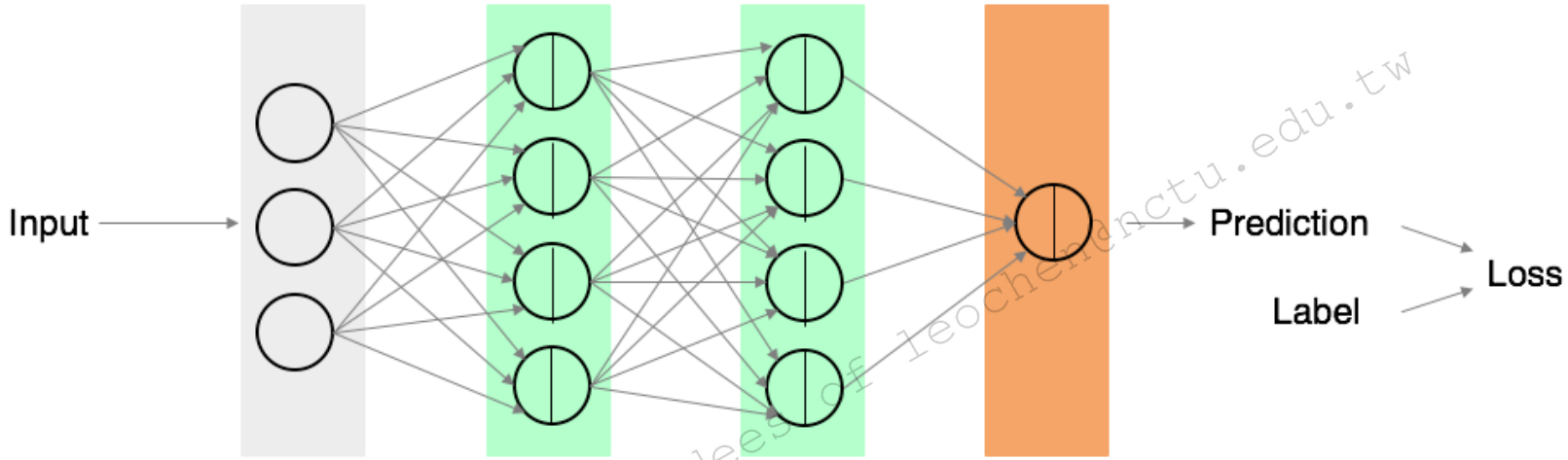
Backpropagation: the core to train deep models

- Let's re-visit Backpropagation of MLP:

1. Initialize all parameters randomly.
2. Loop until loss is small than a threshold: (One batch, one update)
   - 2.1 Propagation: output prediction and calculate loss with prediction and label.
   - 2.2 Backpropagation: calculate the gradient of loss $J(\theta)$ relative to each parameters: $w_{i,j}^{(l)}, b_i^{(l)}$ (the i-th perceptron in layer l, j means the j-th weight)

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \alpha \frac{J(W;b)}{w_{i,j}^{(l)}}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{J(W;b)}{b_i^{(l)}}$$



Remember?
Bias of each perceptron is omitted in the figure.

Qualcomm

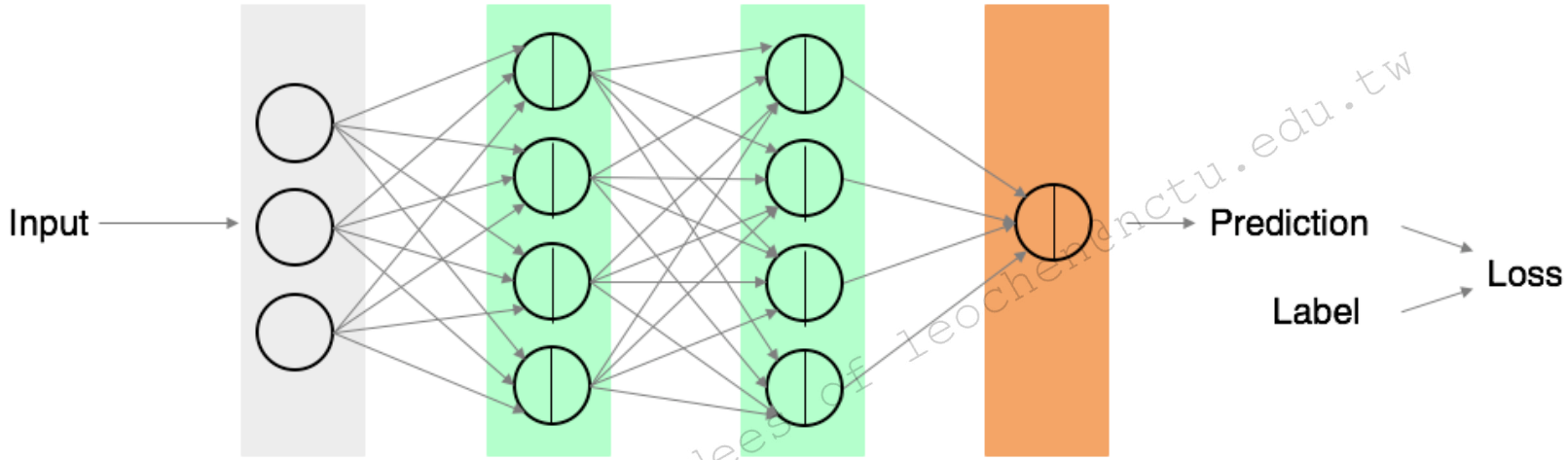
Backpropagation: the core to train deep models

How to calculate gradient of loss $J(\theta)$ relative to each parameters: $w_{i,j}^{(l)}, b_i^{(l)}$?

If we have:

1. the gradient of loss $J(\theta)$ relative to prediction.
2. the gradient of the output of non-linear activation function relative to each parameters: $w_{i,j}^{(l)}, b_i^{(l)}$

We can calculate the gradient of loss $J(\theta)$ relative to each parameters by chain rule.

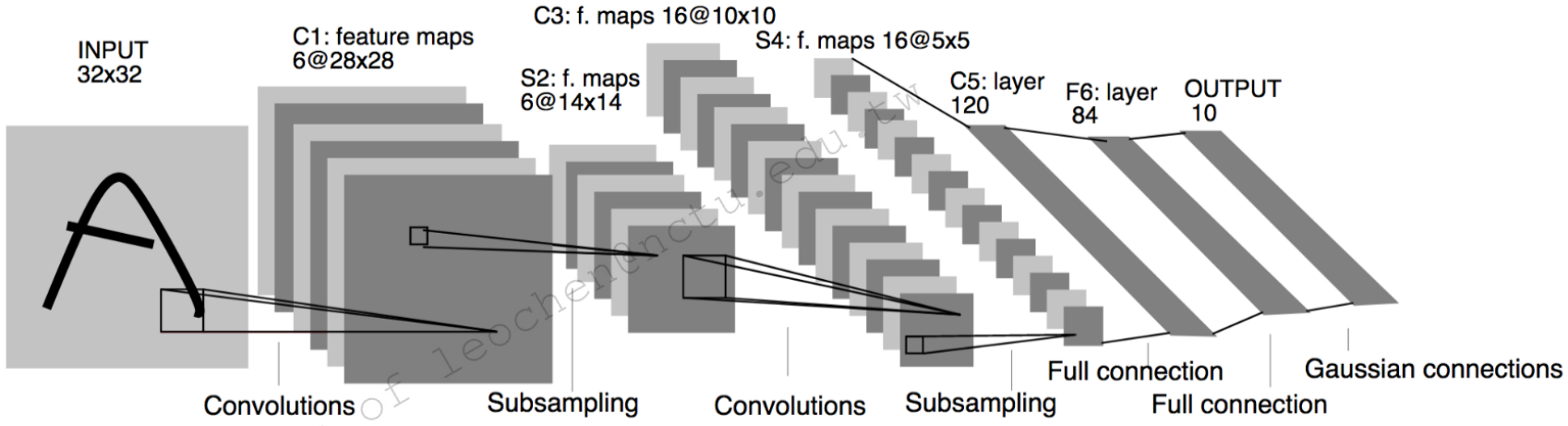


Backpropagation: the core to train deep models

- How to calculate the gradient of loss $J(\theta)$ relative to each parameters in CNNs?

It's easy if we calculate the gradient of loss function relative to parameters in each layer.

- Convolution
- Activation
- Normalization
- Pooling
- Fully connected layer



Gradient-based learning applied to document recognition, 1998

1.3 Foundation of Deep learning

◆ 1.3.5 Prevent Over-fitting in Deep learning

- L1 and L2 Regularization
- Dropout
- Batch normalization layer

Prevent Over-fitting in Deep learning

◆ Review:

What is Over-fitting:

- - The prediction of a model is very closely or exactly to the ground truth, but it's poor to generalize to additional data reliably.
- - May be the model is so complicated that the training data is less than the model fitting ability.
- - Loss on training set is small while loss on validation/test set is big.

How to deal with Over-fitting

- 1. More training data.
- 2. Data augmentation.
- 3. Reduce the model complexity.
- 4. Add regularization item to loss function.
- 5. Other regularization method. (like Dropout / Batch norm in deep learning)

Qualcomm

L1 / L2 regularization

L1 / L2 regularization aim to make the model simpler by add penalty term of model parameters to loss function.

In the other word, instead of trying to minimize only loss (empirical risk minimization), L1 / L2 regularization want to minimize loss+[model complexity]. (structural risk minimization)

Minimize (loss term + regularization)

L2 Regularization

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \sum_{j=1}^{p} \theta j^2$$

L1 Regularization

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \sum_{j=1}^{p} |\theta j|$$

Review: Definition of L1 / L2 loss

L1 loss (Least absolute deviations)

Formula:

$$L_1(x) = |x|$$

For a batch of examples with batch size N, label $y = \{y^{(i)}\}$ and logits $\hat{y} = \{\hat{y}^{(i)}\}$

$$\mathcal{L} = \frac{1}{N} \sum_{i}^{N} \left\| y^{(i)} - \hat{y}^{(i)} \right\|_1$$

L2 loss (Least square errors)

Formula:

$$L_2(x) = x^2$$

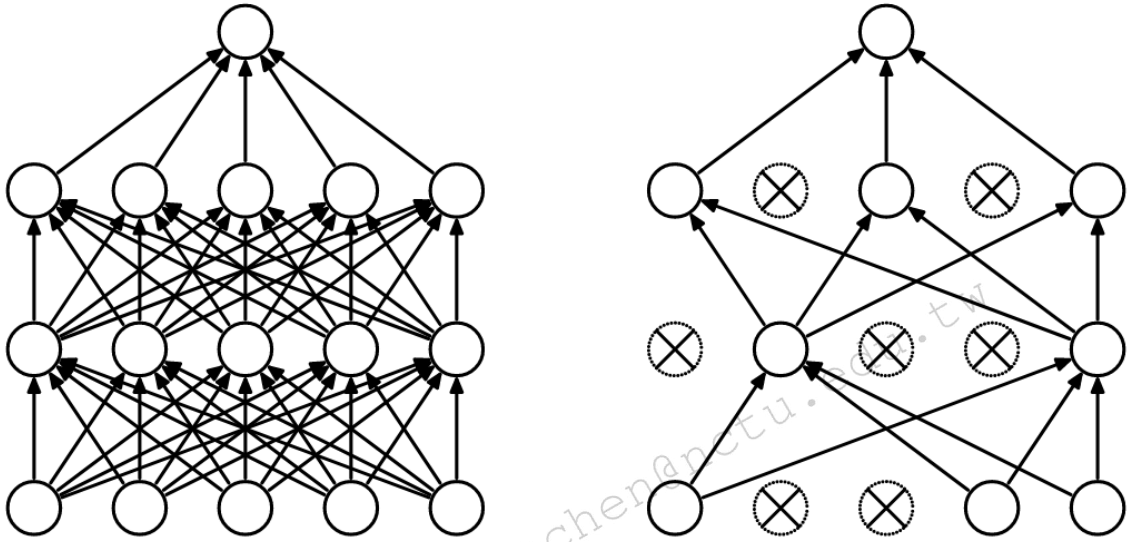
For a batch of examples with batch size N, label $y = \{y^{(i)}\}$ and logits $\hat{y} = \{\hat{y}^{(i)}\}$

$$\mathcal{L} = \frac{1}{N} \sum_{i}^{N} \left(y^{(i)} - \hat{y}^{(i)} \right)^2$$

Qualcomm

Dropout

- Dropping out units in neural network: randomly set some neurons' weight to zero in the forward pass in training stage. Remember, there is no Dropout after the model is already trained.



(a) Standard Neural Net

(b) After applying dropout.

Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2012s

Qualcomm

Dropout

◆ Why Dropout works?

- Interpretation 1: Units may change in a way that they fix up the mistakes of the other units. This may lead to complex co-adaptations.
- Interpretation 2: Each random dropping out generates a sub-network (with less parameters), these different sub-networks are trained on different batch of training data and share parameters. In test stage, we use the ensemble model.

Batch normalization layer

◆ Normalization & Standardization

Normalization is a kind of data preprocess technique. It scale input data to a new range, e.g. [0, 1] or [-1, 1]. The experiment proves that[1], Normalization can help SGD converge.

- min-max normalization: $x' = \frac{x-min}{max-min}$
- log normalization: $x' = \frac{\log(x)}{\log(max)}$
- Arctangent normalization: $x' = \frac{2\arctan(x)}{\pi}$
- Z-score normalization, a.k.a. standardization: $x' = \frac{x-mean}{std}$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arxiv 1502.03167

Batch normalization layer

- Why Batch normalization?
  - In data preprocess stage, we normalize the input data into a standard Gaussian distribution by Z-score normalization, which have a mean of zero and variance of one, so input of Hidden layer 1 is normalized. But for Hidden layer 2 (Internal), its input is changing, so it's called Internal Covariate Shift.
  - Batch normalization make each unit's output (as well as input of next neuron) is consistent with Standard Gaussian distribution.
- What is Batch normalization?
  - Normalize the input batch of each layer.
- How to Batch normalization?

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

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

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, arxiv 1502.03167

Batch normalization layer

◆ Batch normalization during inference.

- During training, mean and variance are not optimized by loss, they are calculated on each training data batch in contrast. So the value of mean and variance are uncertain in different data batch, but we want to get stable result in inference stage.
- We use moving mean and moving variance in training stage, we then save these fixed values after training and use them at test time.

moving_mean = momentum * moving_mean + (1 - momentum) * sample_mean

moving_var = momentum * moving_var + (1 - momentum) * sample_var

- If momentum = 0.1, then

moving_mean = 0.1 * moving_mean + 0.9 * sample_mean

moving_var = 0.1 * moving_var + 0.9 * sample_var

Qualcomm

Thank You