Lecture 4-1 Training a Neural Network

Tian Sheuan Chang

Ref. http://cs231n.github.io/neural-networks-3/

DL Training

- One time setup
  - 0. 資料前處理
  - 1. 網路結構
    - 決定 hidden layers 層數與其中的 neurons 數量
    - 決定各層連接方式 (進階議題)
    - 決定該層使用的 activation function
  - 2. 決定模型的 loss function
  - 3. 訓練相關設定參數 (選擇optimizer)
    - Parameters: learning rate, momentum, decay
    - weight initialization (in separated section)
  - 4. 編譯模型 (Compile model)
  - 5. 開始訓練囉！(Fit model)
- DL babysitting (aka. How to improve your accuracy?)

FIRST EXAMPLE

5步建模 建立深度學習模型

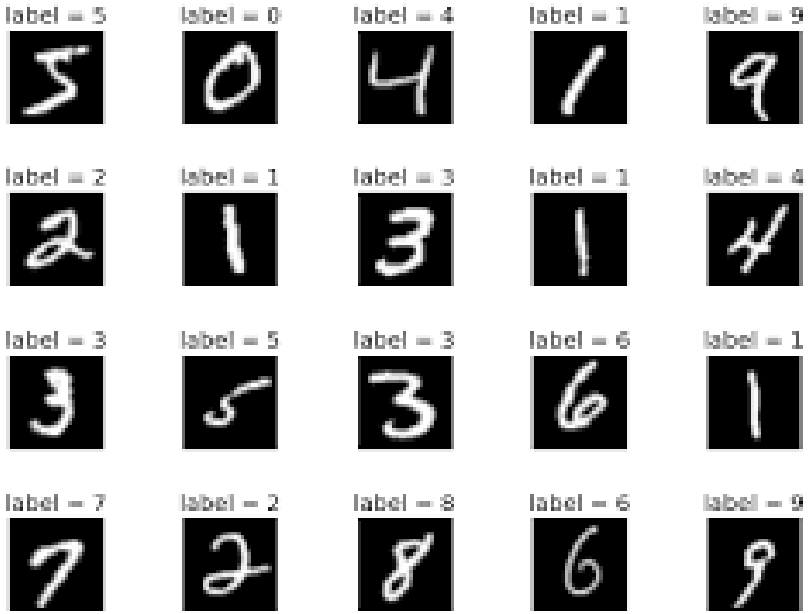
- 0. 資料分割與前處理
- 1. 網路結構
  - 決定 hidden layers 層數與其中的 neurons 數量
  - 決定各層連接方式 (進階議題)
  - 決定該層使用的 activation function
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
  - Parameters: learning rate, momentum, decay

- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

Source:手把手教你深度學習實務

MNIST: 手寫數字辨識

- https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/2.1-a-first-look-at-a-neural-network.ipynb



First 2-Layer NN Example with Keras

- Live demo
- https://github.com/fchollet/deep-learning-with-python-notebooks

Listing 2.1 Loading the MNIST dataset in Keras

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Listing 2.2 The training data

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>> train_labels
[5 0 4 ..., 5 6 8]
```

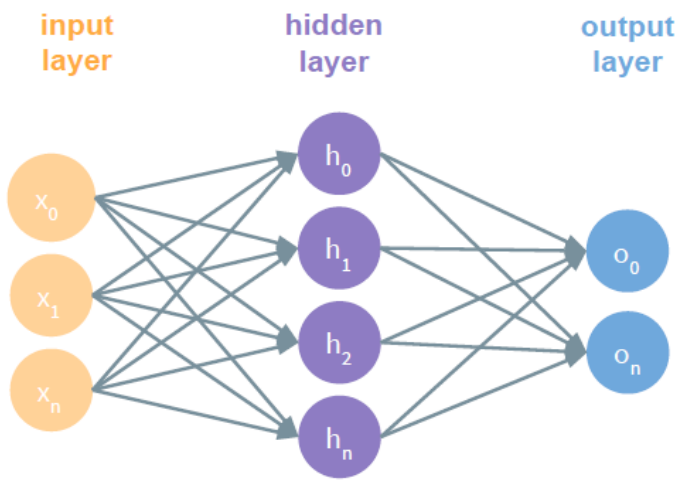
Listing 2.3 The test data

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
[7 2 1 ..., 4 5 6]
```

Listing 2.4 The network architecture

```
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'))
```

- 1. 網路結構



Listing 2.5 The compilation step

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

- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
- 4. 編譯模型 (Compile model)

Listing 2.6 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

- 0. 資料分割與前處理

Listing 2.7 Preparing the labels

```
from keras.utils import to_categorical

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

Listing 2.8 Training the network

```
>>> 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
```

- 5. 開始訓練囉！(Fit model)

Listing 2.9 Evaluating the network

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

5步建模 建立深度學習模型

- 0. 資料分割與前處理
  - 資料分割：
    - train/val/test；避免洩漏。
  - 資料前處理：
    - normalization、tokenization，data augmentation
- 1. 網路結構
  - 決定 hidden layers 層數與其中的 neurons 數量
  - 決定各層連接方式 (進階議題)
  - 決定該層使用的 activation function
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
  - Parameters: weight initialization, learning rate, momentum, decay
- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

Source:手把手教你深度學習實務

分割資料與常見錯誤 vs 正確做法

避免資料洩漏 (DATA LEAKAGE)

什麼是資料洩漏？

- 資料洩漏 (Data Leakage)
  - 指的是測試或驗證資料在某種形式上
  - 間接或直接影響了模型訓練，導致結果過於樂觀。
- 常見情況包括：
  - 資料重疊：相同或相似樣本同時出現在 train/test。
  - 時間序列錯切：用未來的資料訓練，過去的資料測試。
  - 特徵洩漏：輸入特徵中包含「未來才知道」的資訊。
  - 預處理洩漏：先用全資料做標準化再切分。
  - **群體洩漏：同一群體 (病人/用戶) 被切到 train 與 test。**

- 避免方法：任何與 test/valid 有關的資訊，不應該進到訓練過程。

類型	常見錯誤 (Leakage)	正確做法
資料重疊 (duplication)	同一張圖片/同一筆樣本，同時出現在 train/test（可能是原圖 vs resize 後版本）。	切分前先檢查重複樣本；資料擴增 (augmentation) 僅在 **train** 進行。
時間序列 (time series)	用 2023–2024 的資料訓練，用 2022 的資料測試 → 模型看見了「未來」。	必須依時間切分：train 用 2020–2022，valid 用 2023，test 用 2024。
特徵洩漏 (feature leakage)	使用目標變數的衍生欄位，例如：預測「是否出院」卻把「出院日期」當特徵。	僅使用「預測時點已知」的特徵。任何未來才會知道的資訊都要排除。
預處理洩漏 (preprocessing leakage)	用 **整份資料**計算標準化 mean/std，再套用到 train/test。	只用 **train set** fit scaler，再 transform validation/test。
群體洩漏 (group leakage)	同一病人的 MRI 切片，部分進 train，部分進 test → 模型其實學到「病人 ID」特徵。	切分必須以「群體」(patient/公司/用戶) 為單位，而不是單張樣本。
資料不平衡處理	在 train+test 上一起做 SMOTE/oversampling → test 也被「改造」了。	僅在 **train set** 上做資料平衡，再套用模型到原始的 valid/test。

- 0. 資料分割與前處理
- 1. 網路結構
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

DATA PREPROCESSING

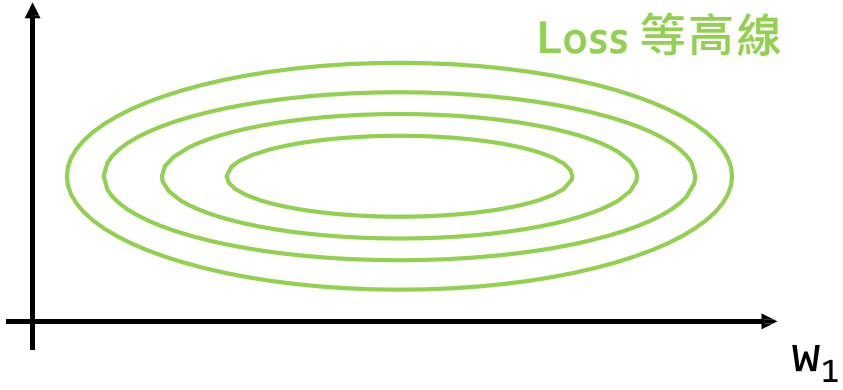
Before You Start: Clean Your Data First

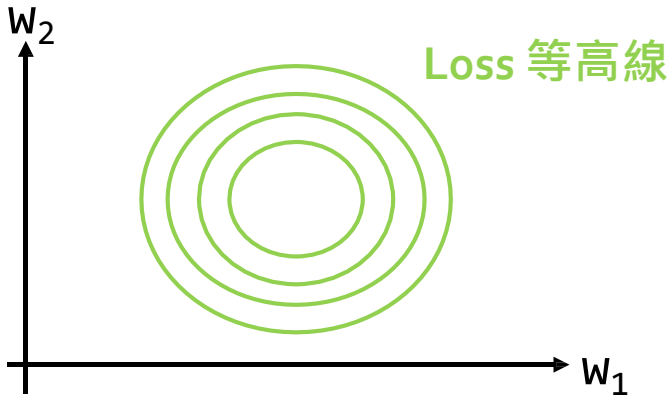
- **Better Data > Fancier Algorithms**
- **Always check your data before you start**

- Remove Unwanted observations
  - Duplicated or irrelevant data (home v.s. apartment)
- Fix structural errors
  - Missing values, Formatting errors, typos
  - Mis-labeled data
- Filter unwanted outliners
  - if you have a legitimate reason to remove an outlier, it will help your model's performance.
- Missing data
  - Dropping or imputing missing values are not a good idea
  - missingness is informative

輸入前處理 (preprocessing)

- 因為必須跟 weights 做運算
  - Neural network 的輸入必須為數值 (numeric)
- 如何處理非數值資料？順序資料/名目資料
  - One-hot encoding
  - 順序轉成數值
- 不同輸入的數值範圍差異會有影響嗎？
  - 溫度: 最低 0 度、最高 40 度
  - 距離: 最近 0 公尺、最遠 10000 公尺
  - 影響訓練的過程
  - 建議 re-scale！

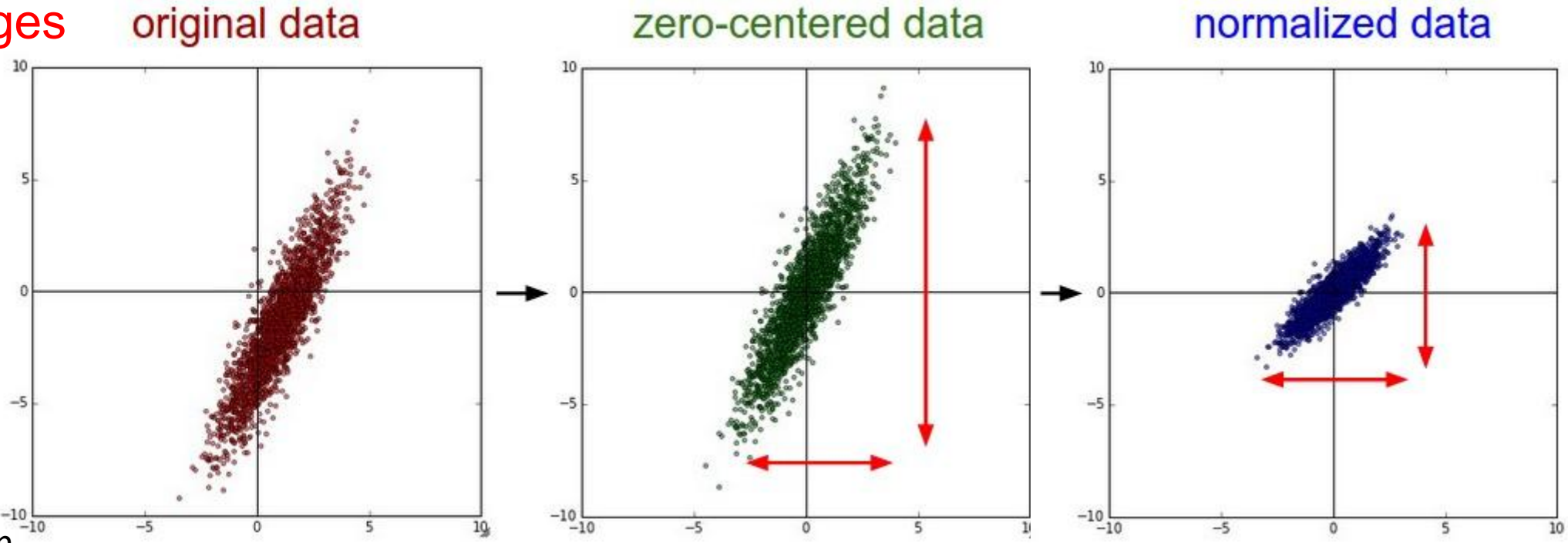




Source:手把手教你深度學習實務

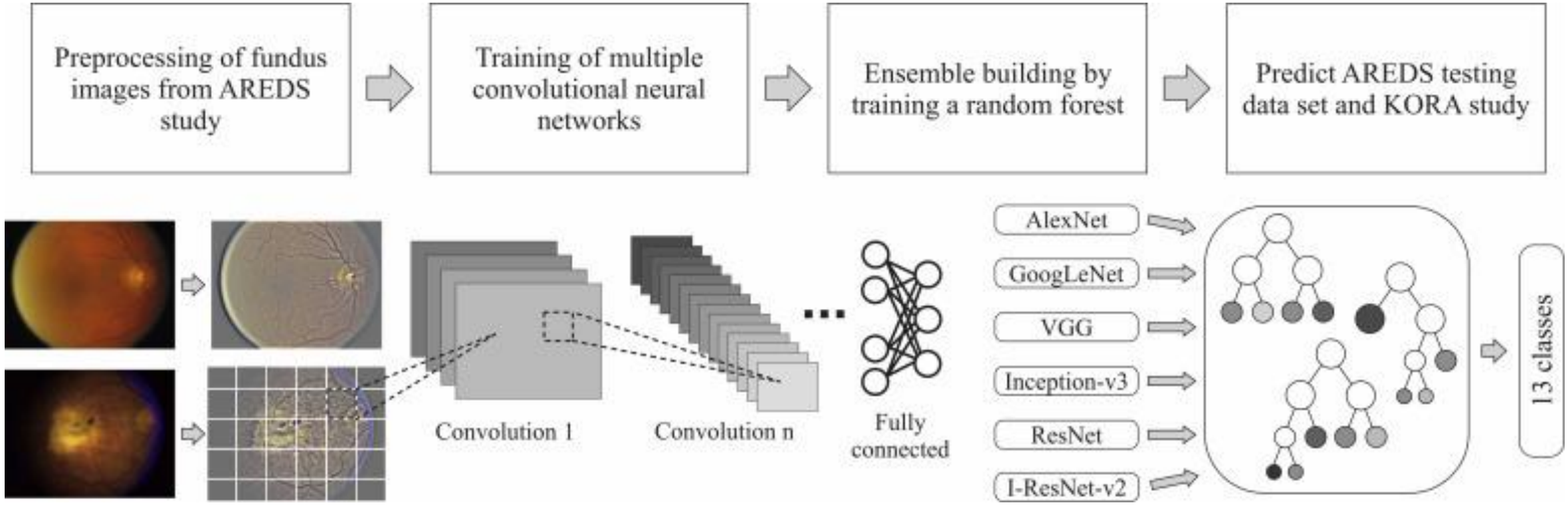
Preprocessing

- Mean subtraction: zero centered `X -= np.mean(X, axis = 0)` `X -= np.mean(X)`
- Normalization: data at approximately the same scale
  - Divide by its standard deviation, `X /= np.std(X, axis = 0)`
  - normalizes each dimension so that the min and max is -1 and 1
  - Meaningful for data with different scales but equal importance (not hold for images

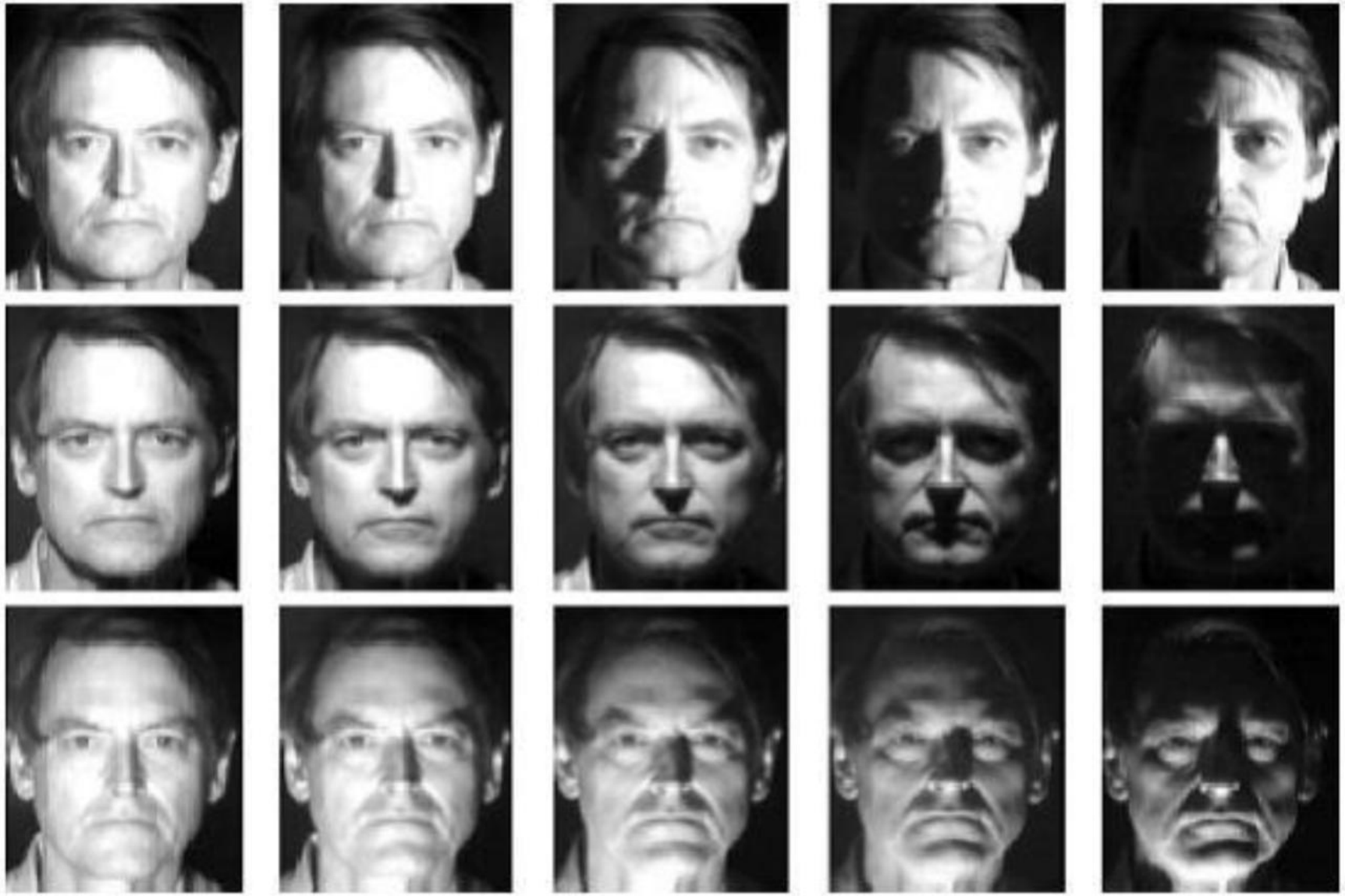


Source: cs231n

Image Enhancement? It depends



Deep Learning is Hard:
Illumination Variability

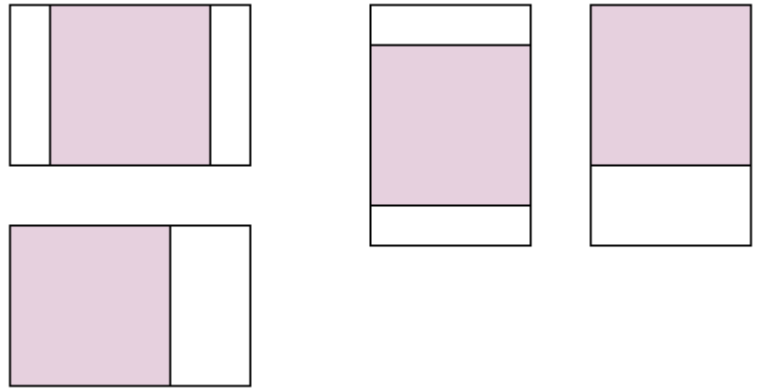


Need
data augmentation

Day and Night, Rainy and Sunny Days

Need
data augmentation

Other Image Preprocessing



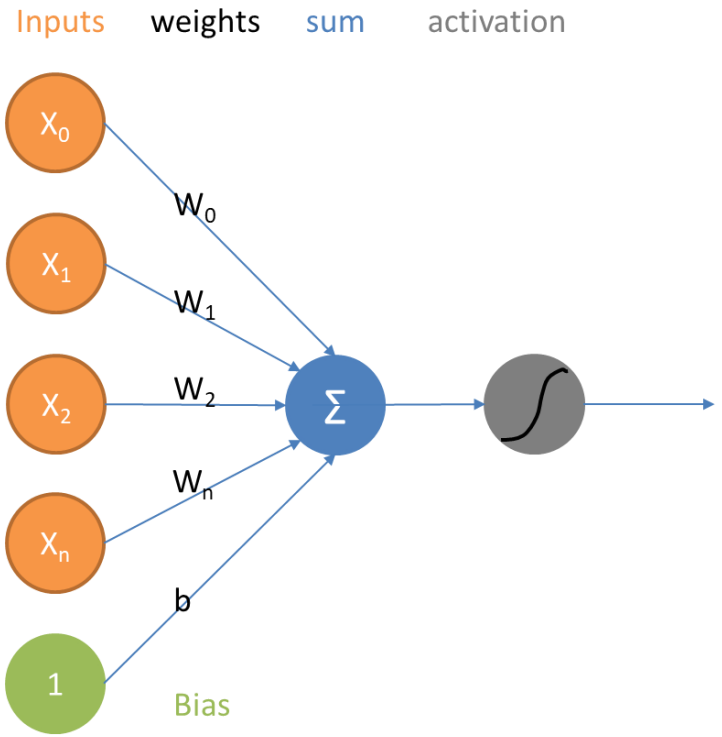
- Uniform size and aspect ratio
  - Usually care about the center part
  - Apply image scaling if necessary
- Dimension reduction
  - RGB to gray only?

5步建模 建立深度學習模型

- 0. 資料前處理
- 1. 網路結構
  - Network capacity and structure: depends on your task
    - 寧大勿小
  - Activation function
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

- 0. 資料前處理
- 1. 網路結構
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

ACTIVATION FUNCTION



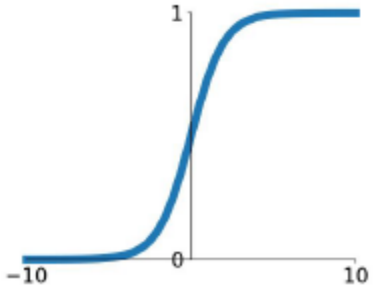
Why Activation Function? Activation Function Properties

- Nonlinear
  - **For a linear activation, a multi-layer NN is equivalent to a single layer NN**
  - Impossible to learn nonlinear properties
- Differentiable
  - Necessary for gradient based optimization
  - Some activation functions is only differentiable at some range
- Monotone value
  - Monotone values guarantee a convex function
  - NN has many local minimum (a good local minimum)
- Range of output value
  - Finite output enable more stable optimization process
  - For infinite output value, training will be more efficient but need smaller learning rate

Activation Fuctions

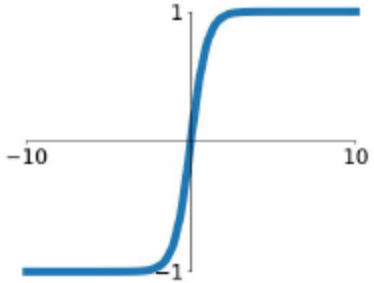
Sigmoid

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



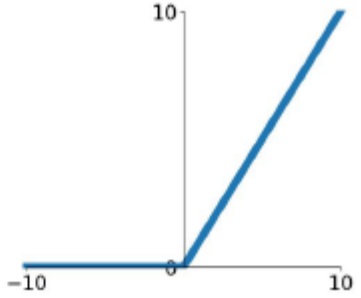
tanh

$$\tanh(x)$$



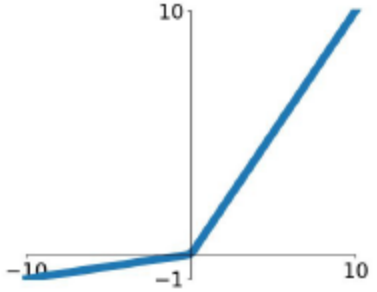
ReLU

$$\max(0, x)$$



Leaky ReLU

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

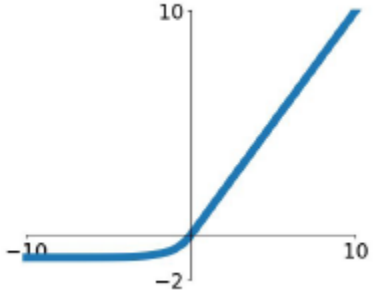


Maxout

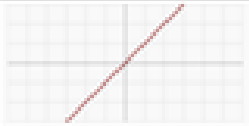
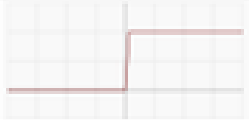
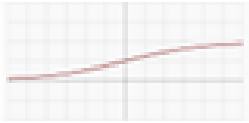
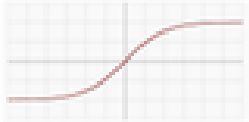
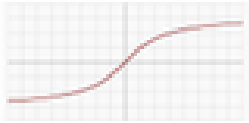
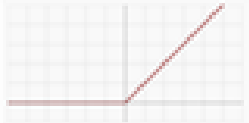
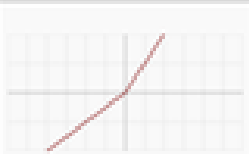
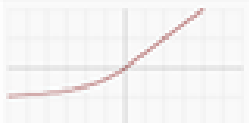
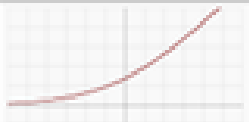
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

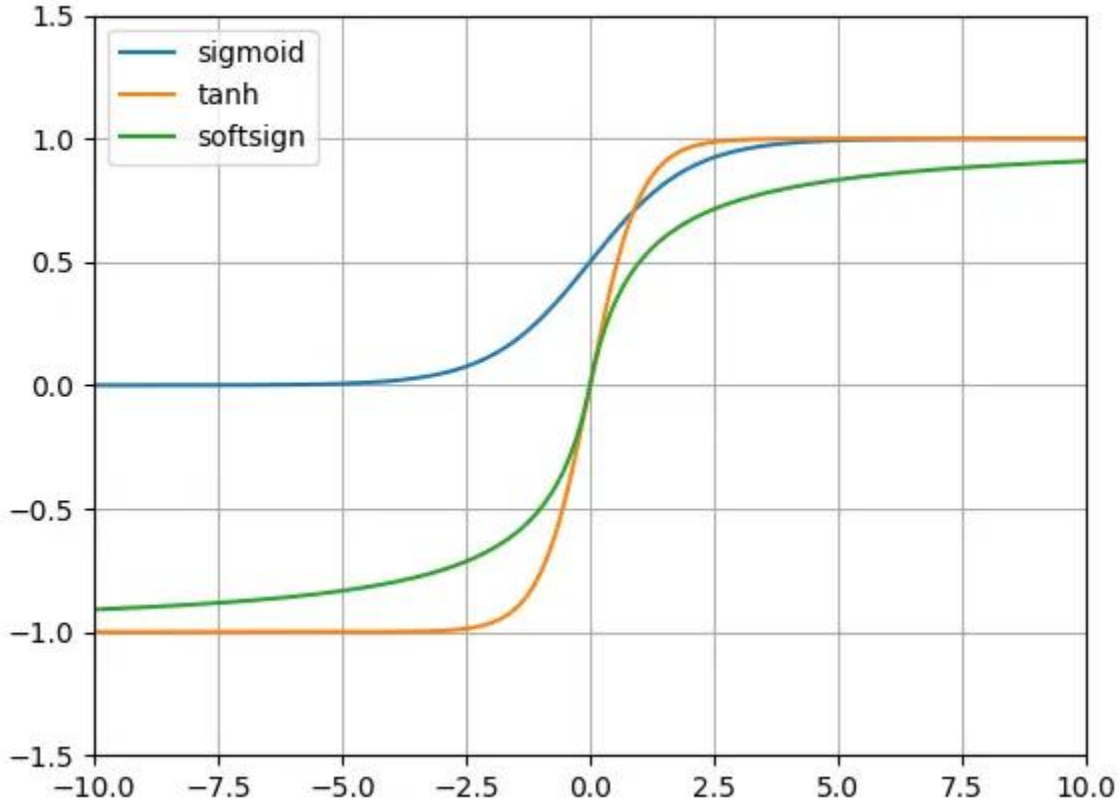
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Source: CS231n

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

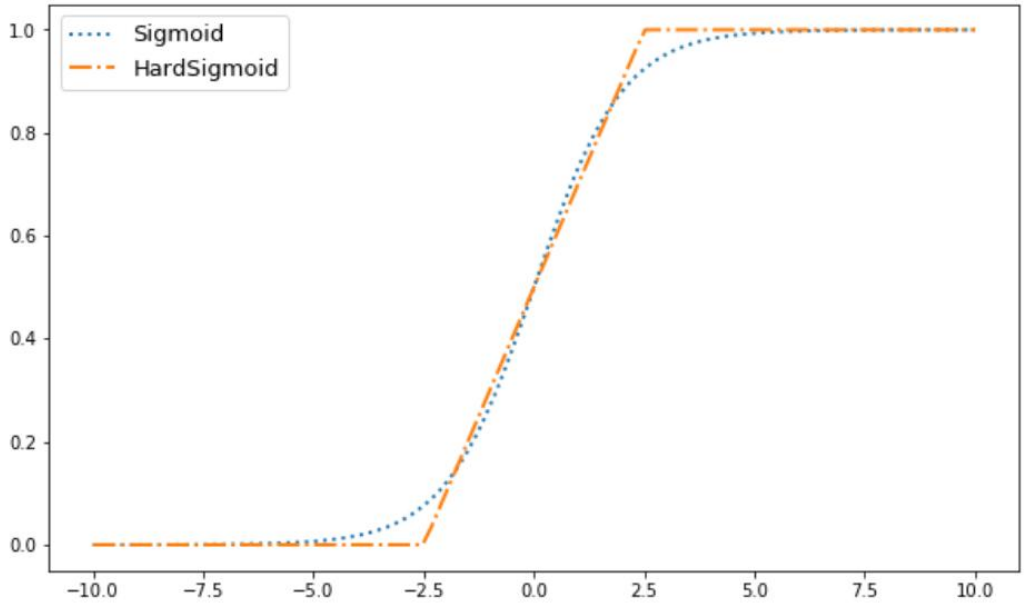
- Softsign
  - x/(abs(x)+1)
  - Replacement of tanh



Hard Sigmoid

- Low complexity version of sigmoid

$$\text{Hardsigmoid}(x) = \begin{cases} 0 & \text{if } x \leq -3, \\ 1 & \text{if } x \geq +3, \\ x/6 + 1/2 & \text{otherwise} \end{cases}$$



Keras

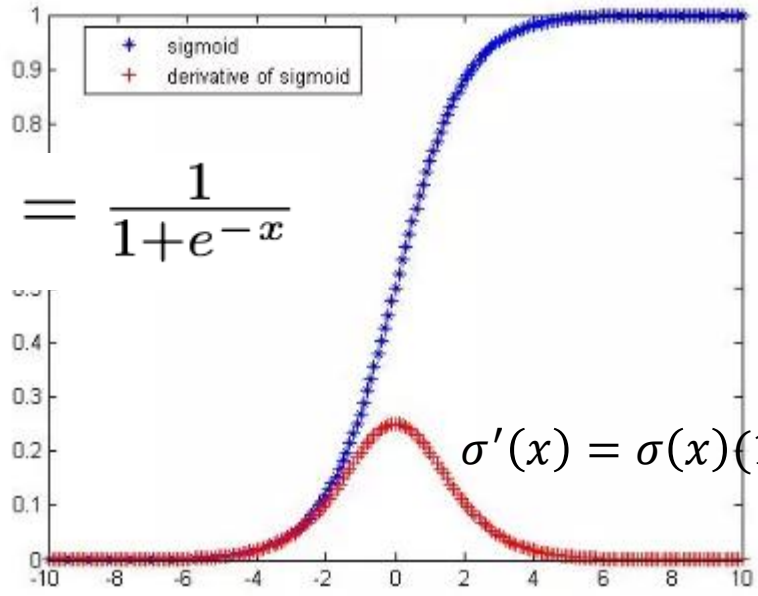
The hard sigmoid activation, defined as:

- `if x < -2.5: return 0`
- `if x > 2.5: return 1`
- `if -2.5 <= x <= 2.5: return 0.2 * x + 0.5`

Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

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



$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

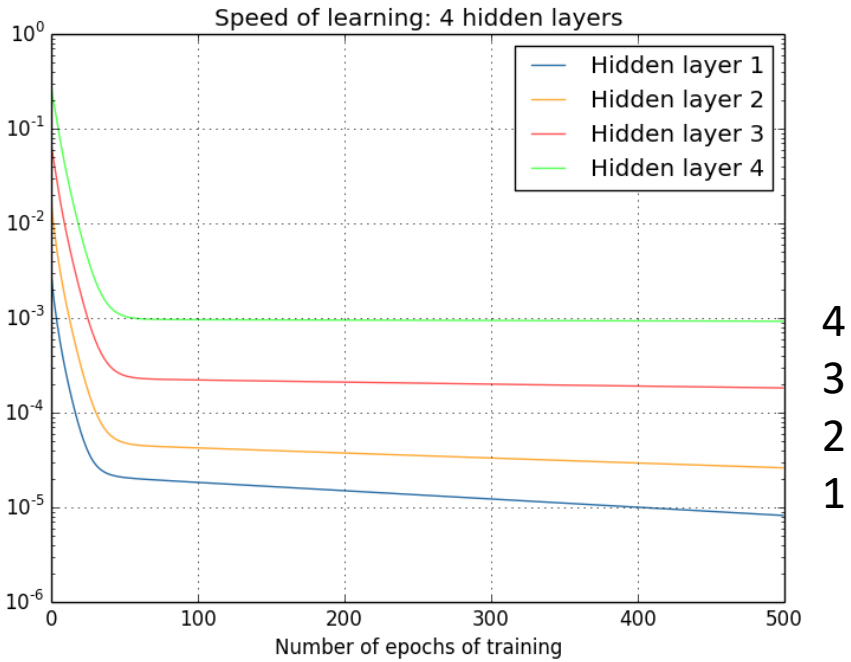
- 3 problems
  - Sigmoids saturate and kill gradients
    - Gradient at tails of 0 and 1 is almost zero
    - Kill gradients when multiplied with others
    - Beware of initialization to prevent saturation
  - Sigmoid outputs are not zero-centered
  - Exp() is a bit computing expensive

Source: CS231n

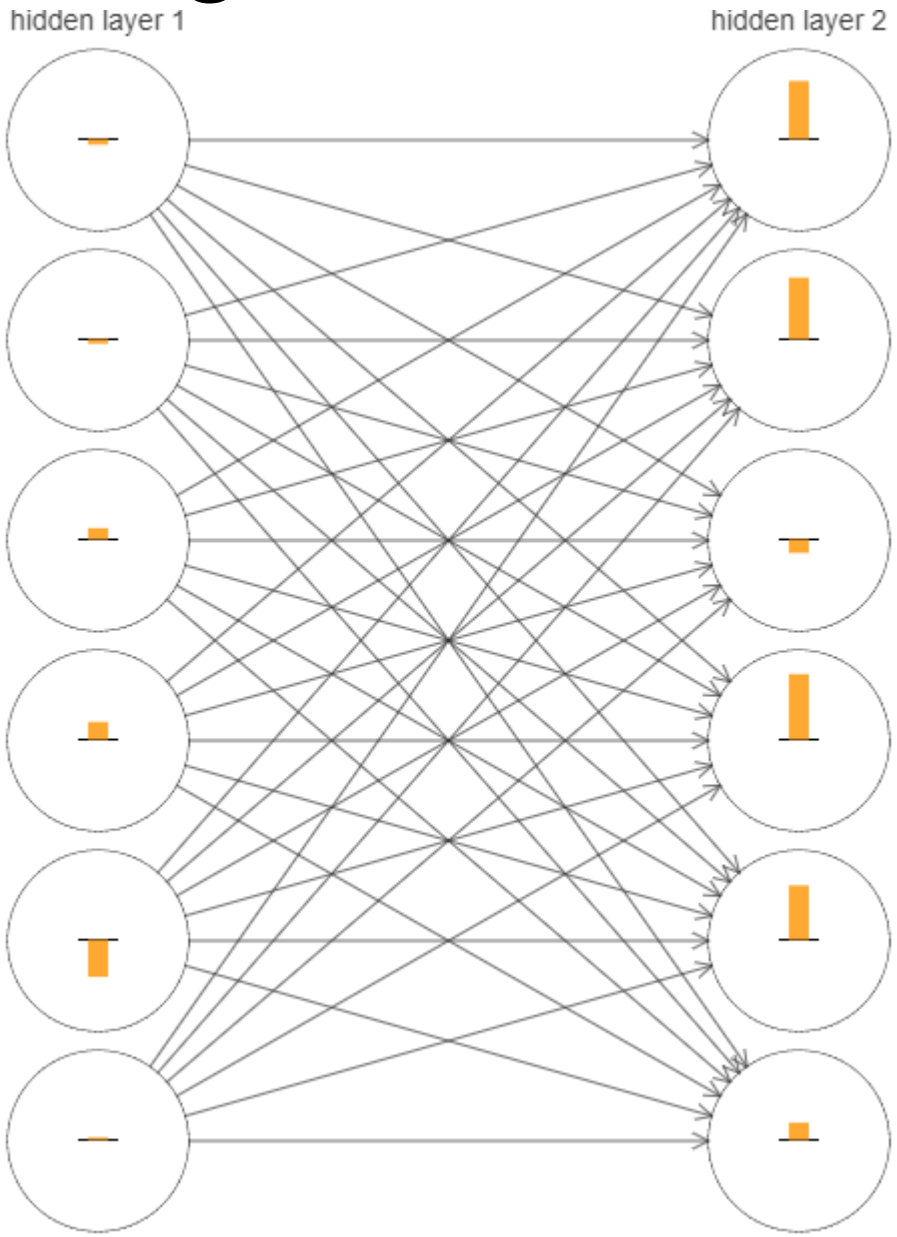
Vanishing Gradient /Exploding Gradient

Problem

- An MNIST Example
  - 2 layer NN, 96.48%
  - 3-layer NN, 96.9%
  - 4-layer NN, 96.57%
  - 5-layer NN, 96.53%
- Gradient in deep NN is unstable



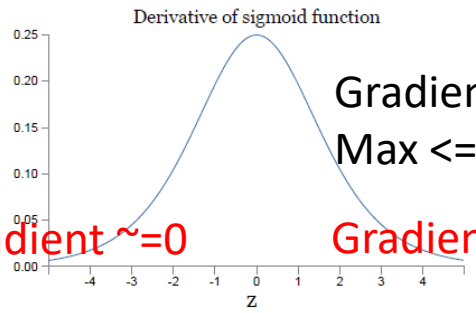
4
3
2
1



Source: http://neuralnetworksanddeeplearning.com/chap5.html

Vanishing Gradient Problem

$$\frac{\partial C}{\partial W^1} = \frac{\partial C}{\partial y}\,\sigma'(z_i^2)\,W^2\,\sigma'(z_i^2)\,x$$



Gradient
Max <=1/4

Gradient ~=0

Gradient ~=0

$$C = L(y, \hat{y})$$

$\hat{y}$ → C

$\partial C / \partial y$

$z^1 = W^1 x + b^1$

$z^2 = W^2 a^1 + b^2$

x → W^1 → z^1 → $\sigma'(z_i^2)$ → σ → a^1 → W^2 → z^2 → $\sigma'(z_i^2)$ → σ → y

W^1 ← x

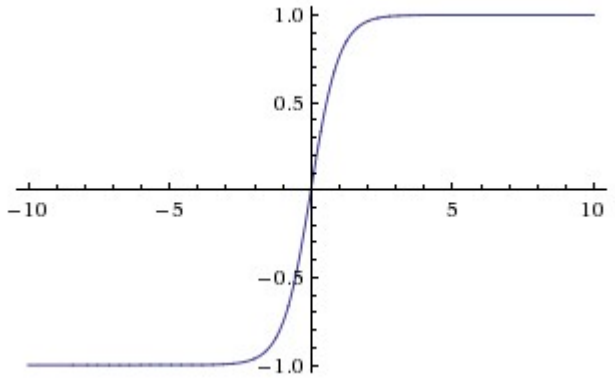
W^2 ← a^1

$$\frac{\partial C}{\partial W^1}$$

Source: computing gradient, Hung-yi Lee

tanh: Hyperbolic Tangent function

- squashes a real-valued number to the range [-1, 1]
- tanh is simply a scaled sigmoid neuron
  - If $\tanh(x) = 2\sigma(2x) - 1$ hold

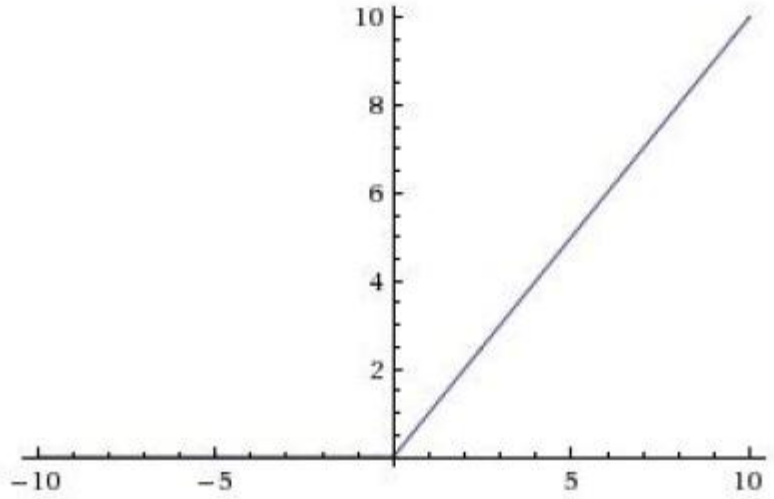


- Problems
  - Output is also saturated like sigmoid (bad)
  - But zero-centered (nice)
  - tanh non-linearity is always preferred to the sigmoid nonlinearity
  - still it suffers from Vanishing gradient problem

Source: CS231n

ReLU: Rectified linear units

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

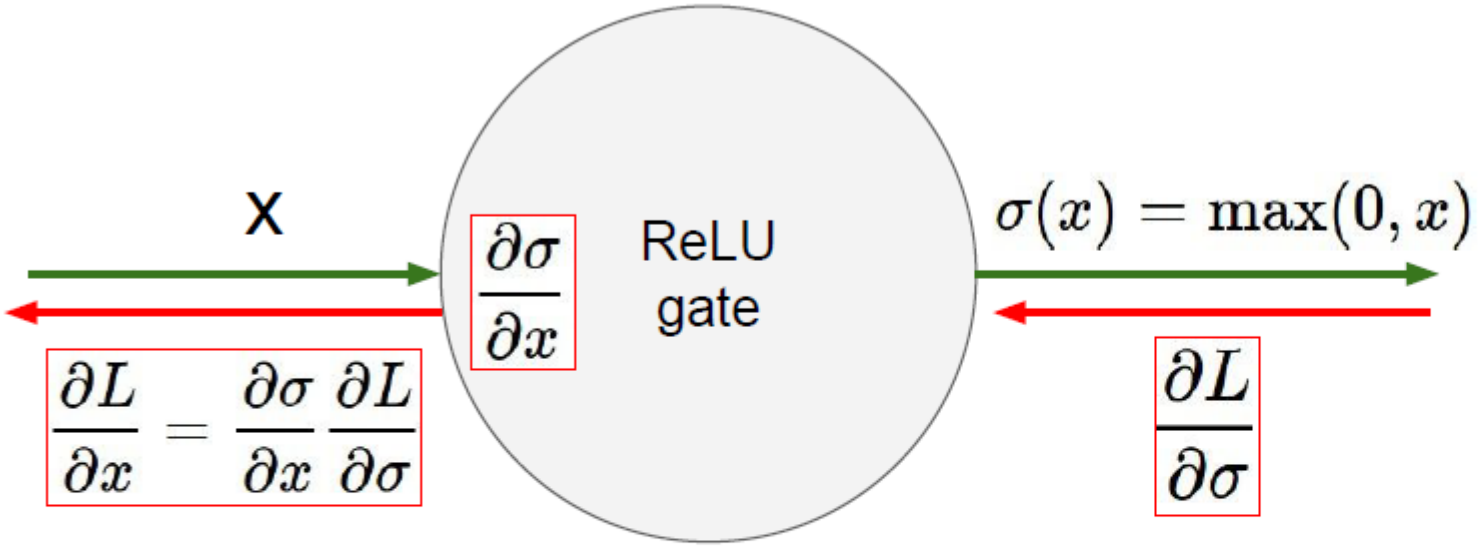
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

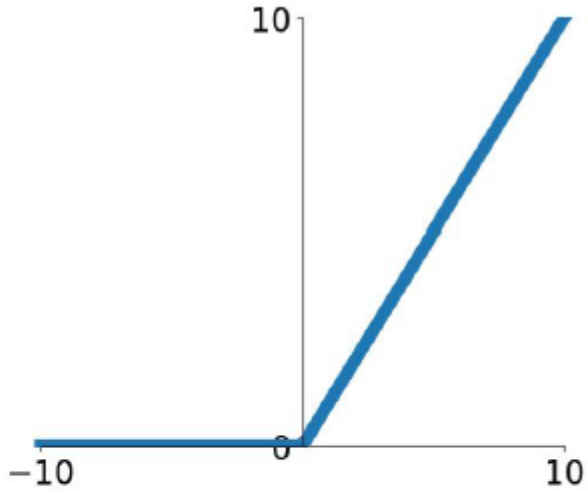
- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

Source: CS231n

ReLU cause Dead Neurons: Weight Never Update

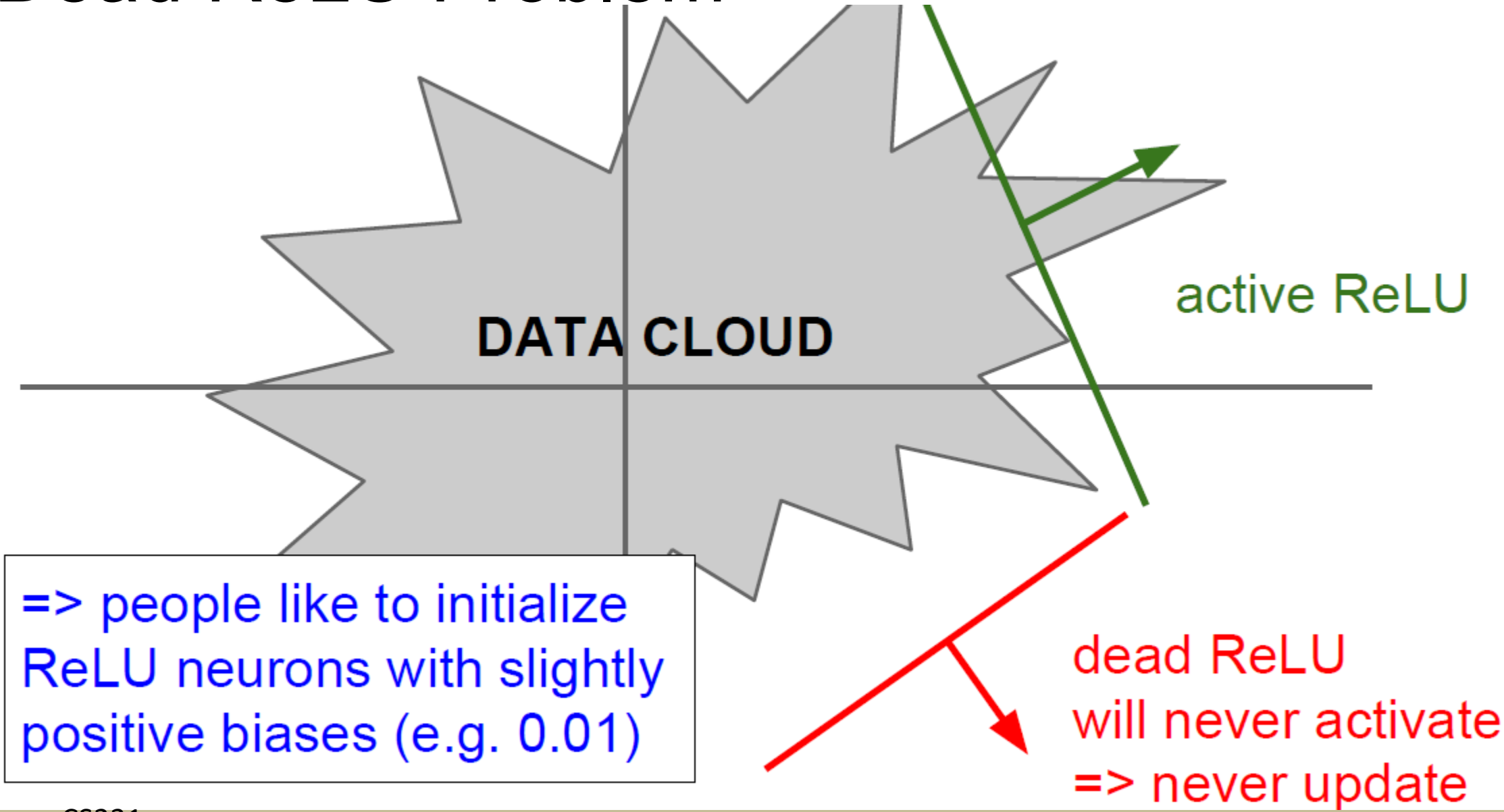




What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

Source: CS231n

Dead ReLU Problem

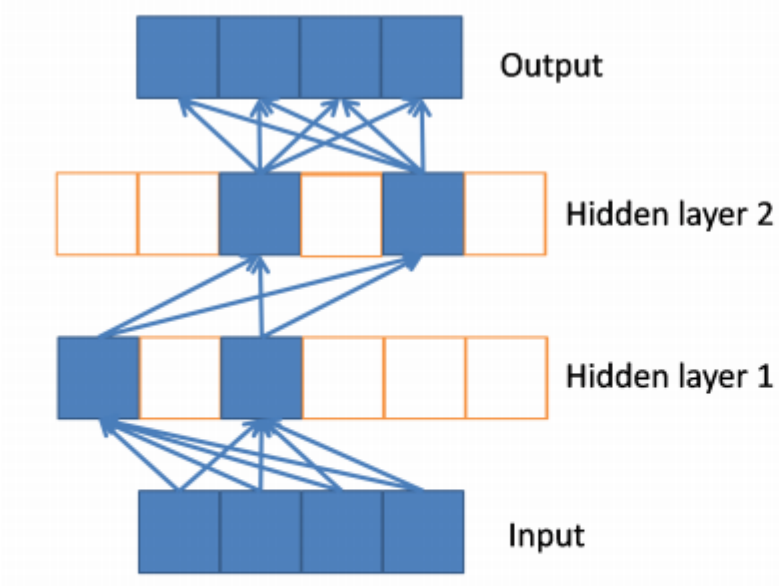


=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

Source: CS231n

Sparse Representation of Network

- Around 50% of hidden units output will be zero with uniform zero mean initialization
  - More with regularization



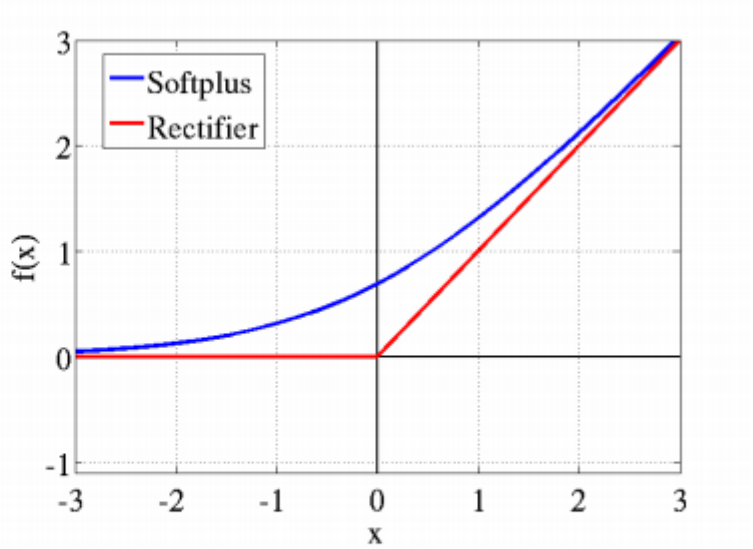


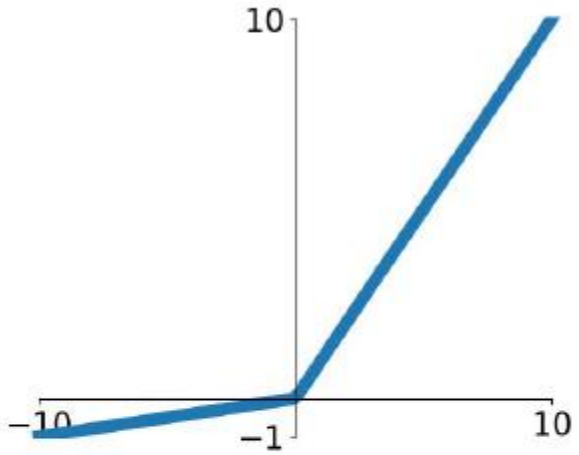
Figure 2: *Left:* **Sparse propagation of activations and gradients in a network of rectifier units.** The input selects a subset of active neurons and computation is linear in this subset. *Right:* **Rectifier and softplus activation functions.** The second one is a smooth version of the first.

Some Notes About ReLU

- ReLU should only be used within Hidden layers of a Neural Network Model
  - For output layer, use softmax for classification, use linear function for regression
- Zero derivative could block backpropagation of gradient
  - Softplus has been tried, but ReLU is better
  - not a problem as long as the gradient can be propagated along some paths
- Unbounded output could lead to numerical instabilities
  - L1 regularization help
  - ReLU6 (upper bound to 6 for INT8 quantization)

Source: CS231n

Variants of ReLU: Leaky ReLU and PReLU



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

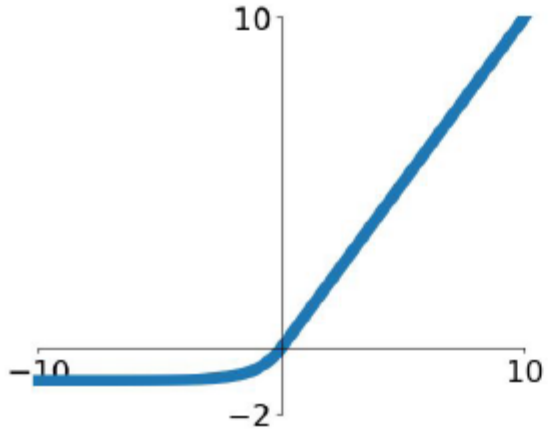
Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

Variants of ReLU: ELU

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- Computation requires exp()

Source: CS231n

GELU: Gaussian Error Linear Units

- a smoother ReLU
  - used in GPT-3, BERT, and most other Transformers

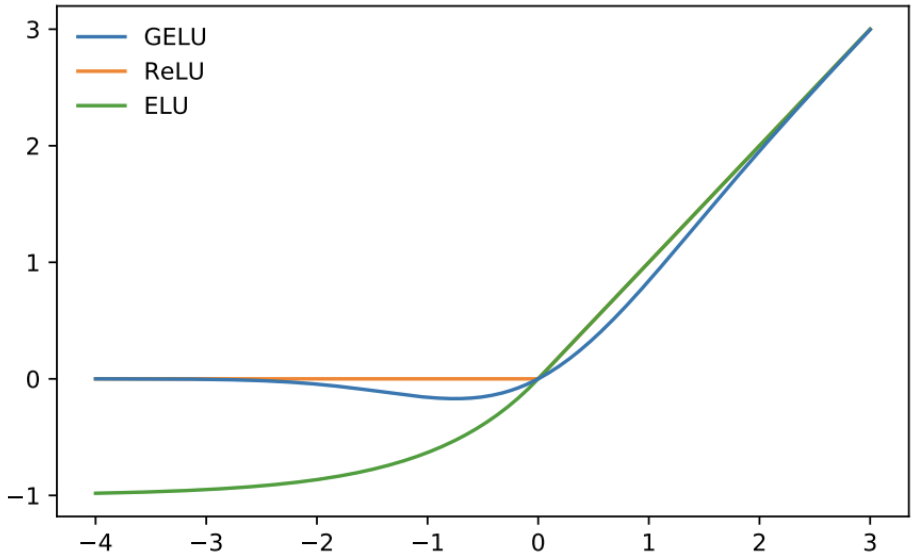


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

$$\mathrm{GELU}(x) = xP(X \leq x) = x\Phi(x)$$
$$\approx 0.5x\left(1 + \tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

P(x): cumulative distribution function of the standard normal distribution

GELU為inputs乘一機率值，此值取決於目前的輸入input有多大的機率大於其餘的inputs
e.g. input = 0.5, P(X < 0.5) = 0.5

當x變小時，P(X <= x)的值會減少，也就是當輸入值inpuits 較小時，inputs被drop 的可能性更大

GELU可以看做是對ReLU的平滑+dropout

MaxOut: Learned Activation

- Not a fixed function
- A learned activation
- Can approximate any activation function
  - Generalizes ReLU and Leaky ReLU
  - Linear Regime! Does not saturate! Does not die!
- Need extra K parameters

$$h_i(x) = \max_{j \in [1,k]} z_{ij}$$

$$z_{ij} = x^T W_{\ldots ij} + b_{ij}$$

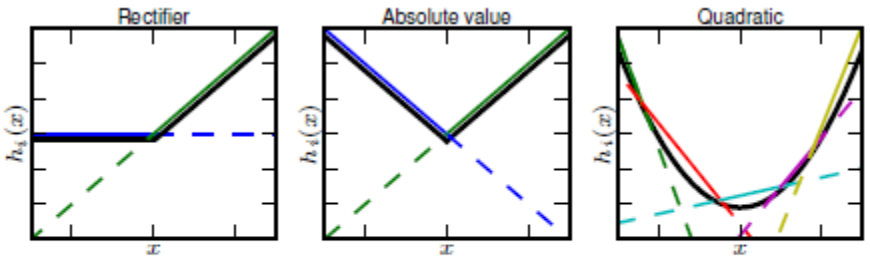
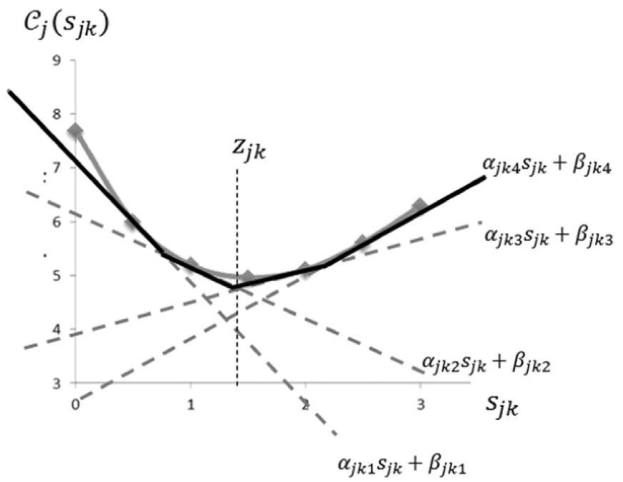


Figure 1. Graphical depiction of how the maxout activation function can implement the rectified linear, absolute

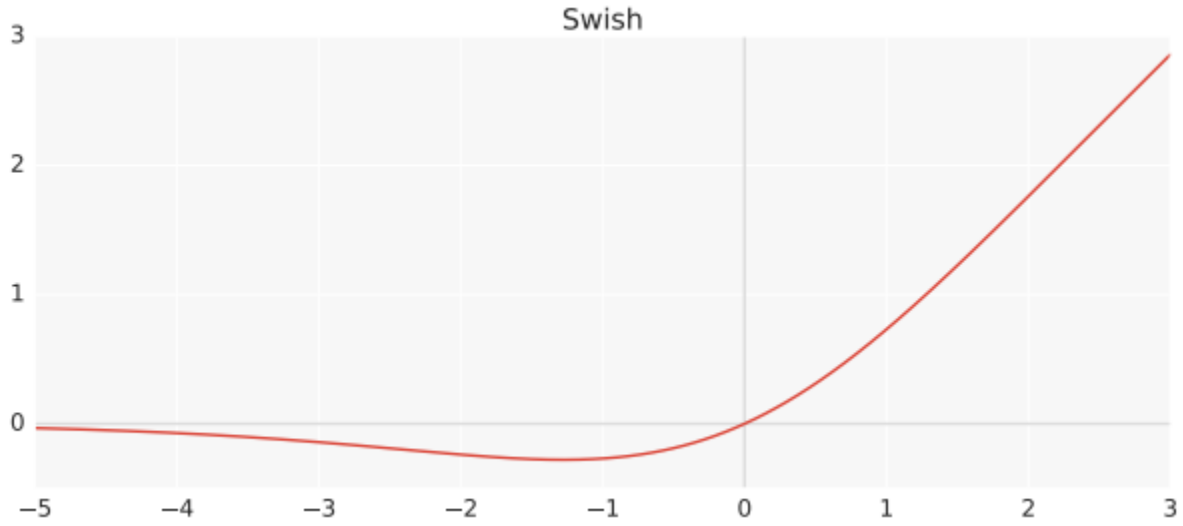



```
output = K.max(K.dot(X,  self .W) +  self .b,axis= 1 ) #maxout
```

Swish: a self-gated activation function

Sigmoid Gate 讓模型可以根據輸入內容，決定哪些資訊應該被傳遞下去

- Can we auto search the best activation function?
- $f(x) = x \cdot \text{sigmoid}(x)$ *self-gating: the gate is actually the 'sigmoid' of activation itself.*
  - Swish tends to work better than ReLU on deeper models across a number of challenging data sets



Model	ResNet	WRN	DenseNet
LReLU	94.2	**95.6**	**94.7**
PReLU	94.1	95.1	94.5
Softplus	**94.6**	94.9	**94.7**
ELU	94.1	94.1	94.4
SELU	93.0	93.2	93.9
ReLU	93.8	95.3	**94.8**
Swish	**94.7**	**95.5**	**94.8**

Table 2: CIFAR-10 accuracy.

Mish: A Self Regularized Non-Monotonic Neural Activation Function

$$f(x) = x \tanh(\ln(1 + e^x))$$

Tensorflow：x = x *tf.math.tanh(F.softplus(x))

Pytorch

```
x = x * (torch.tanh(F.softplus(x)))
```

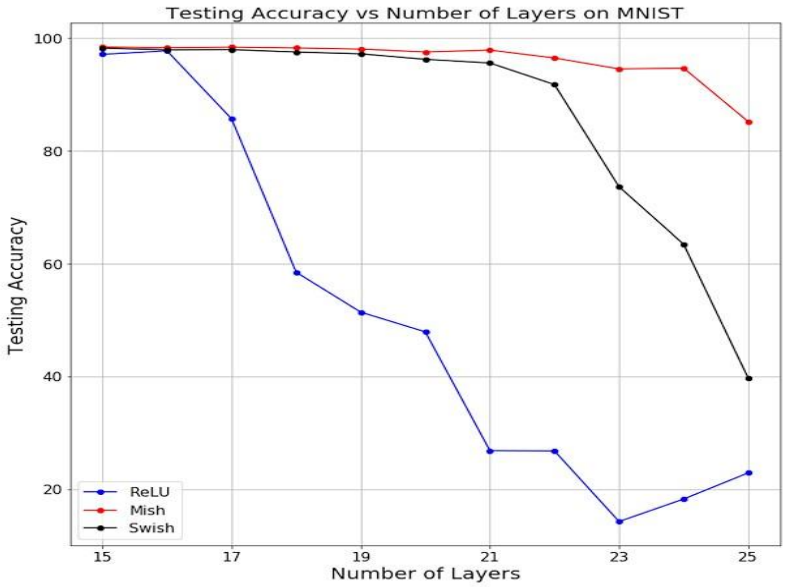
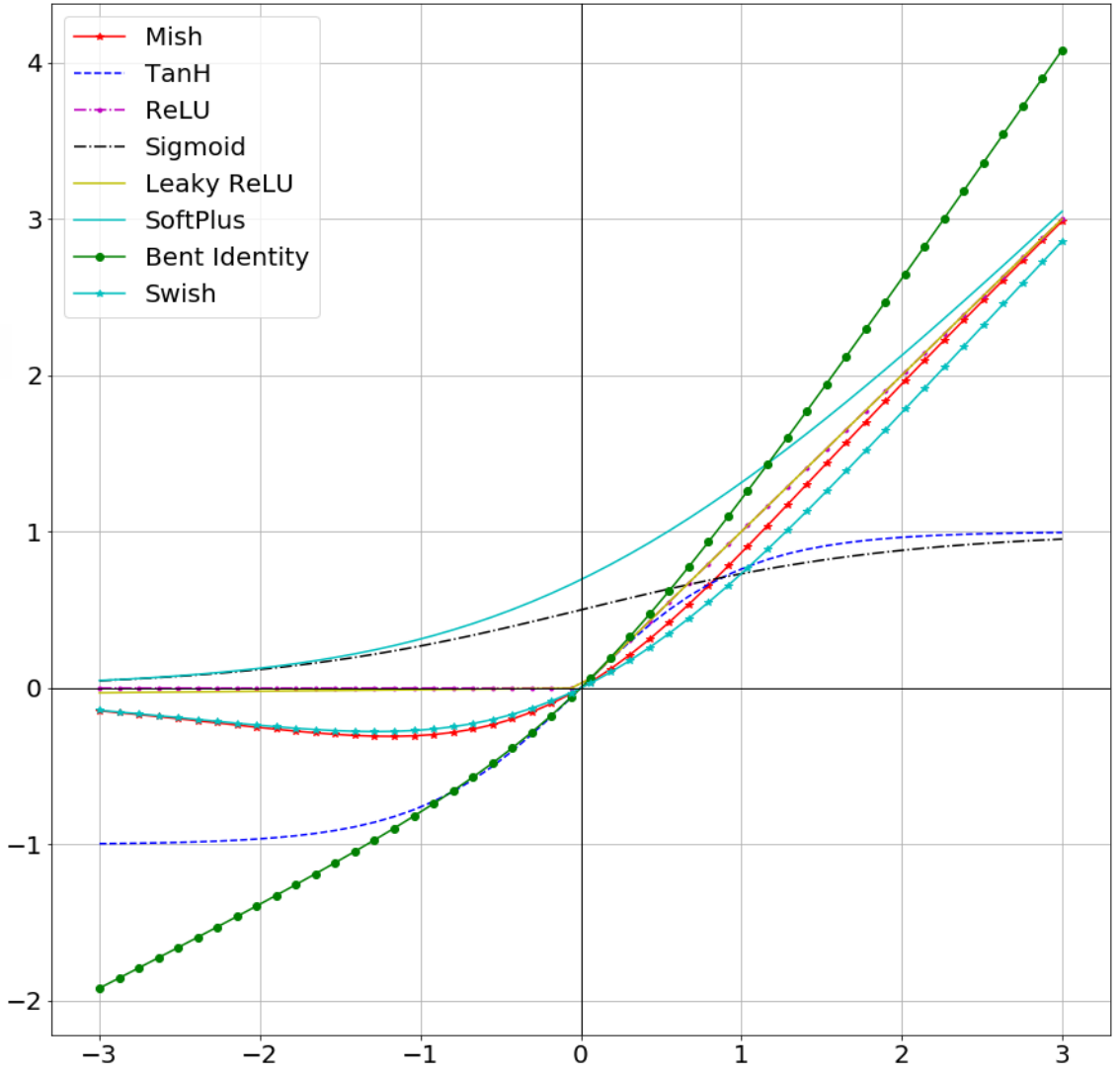
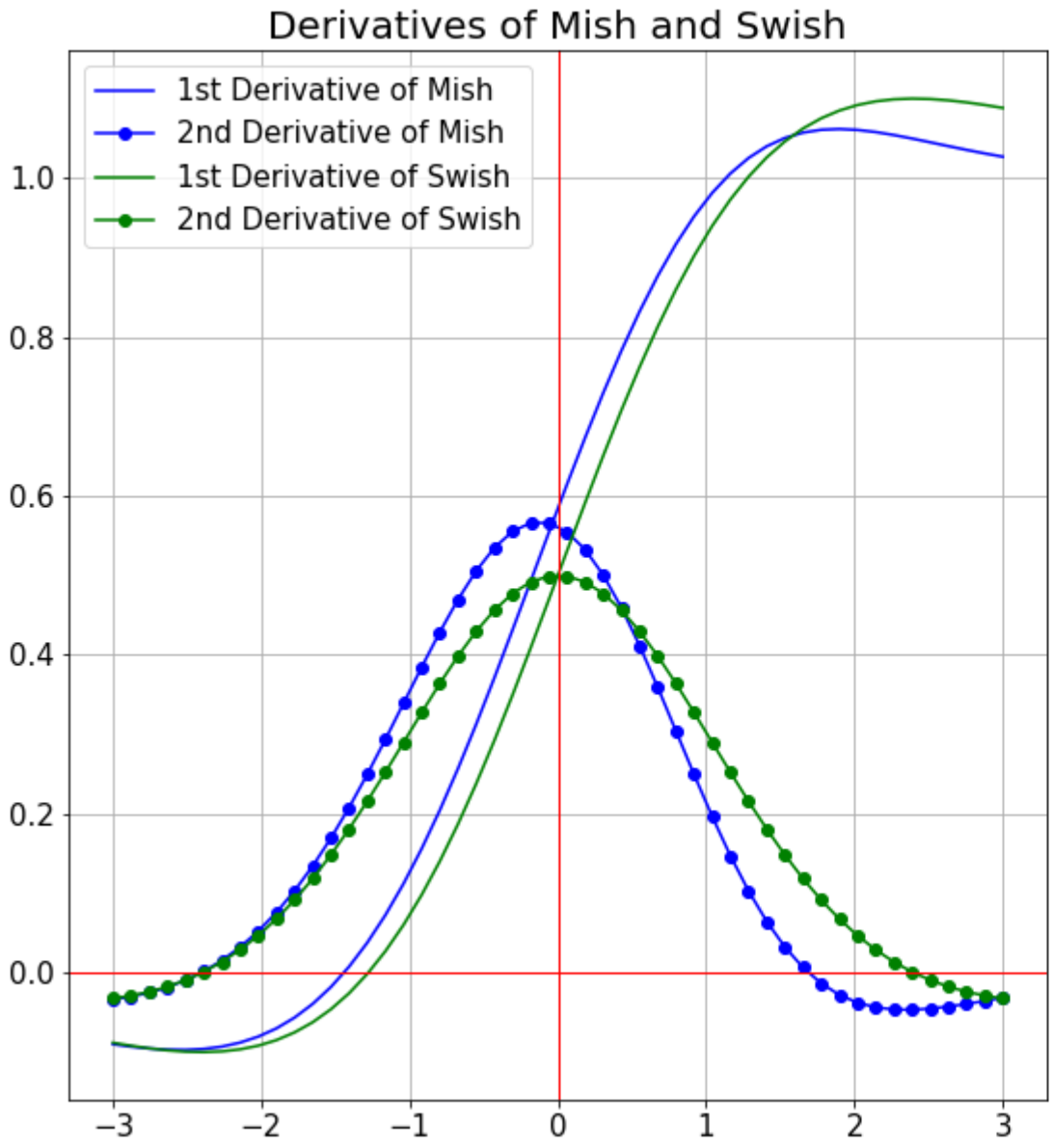
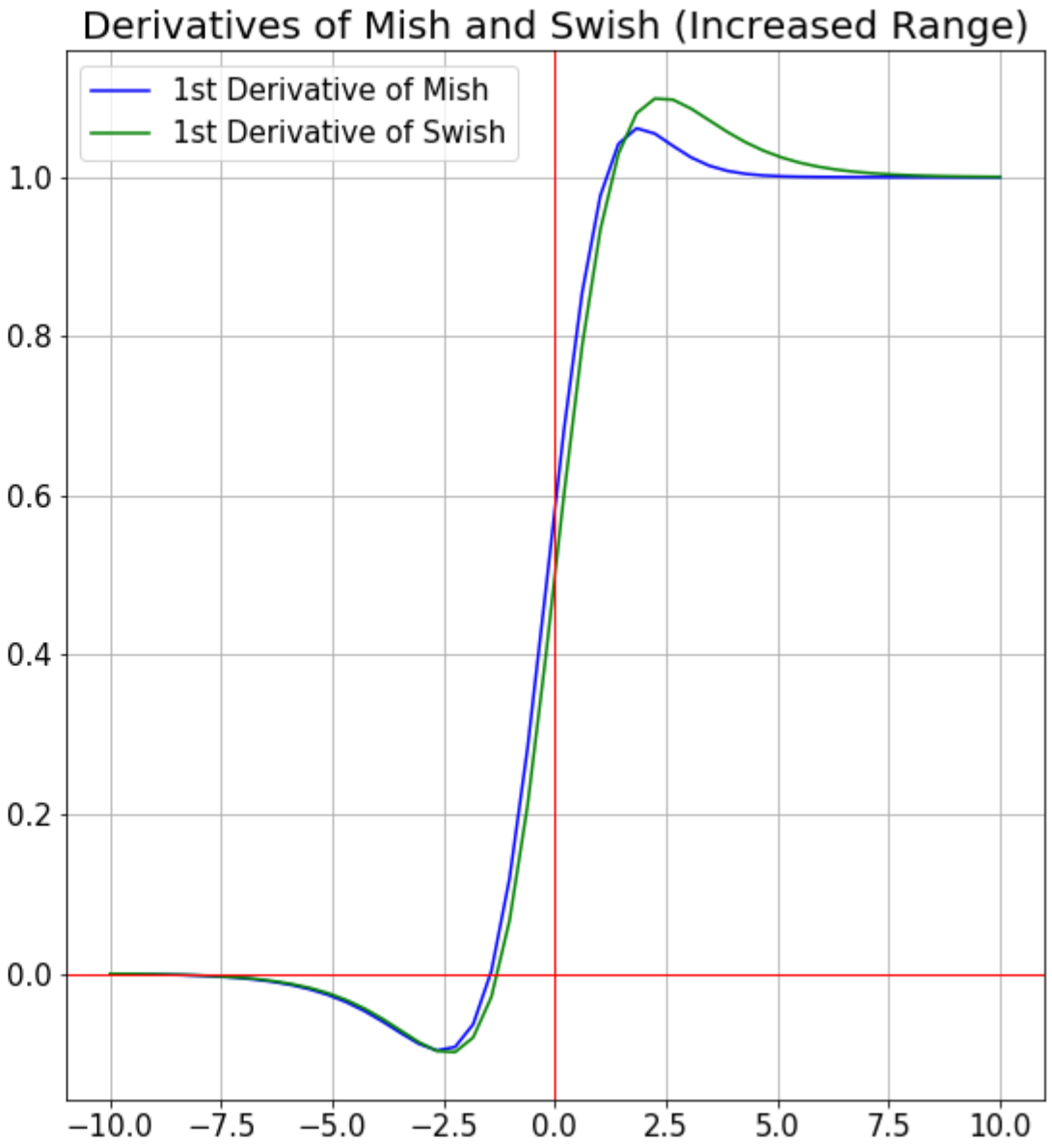
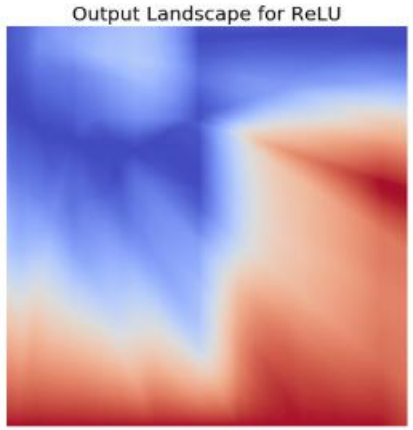


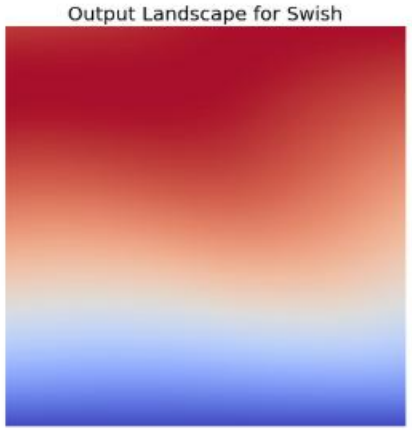
Figure 6. Testing Accuracy v/s Number of Layers on MNIST for Mish, Swish and ReLU.

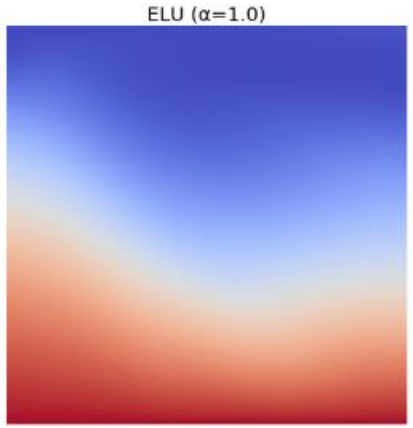


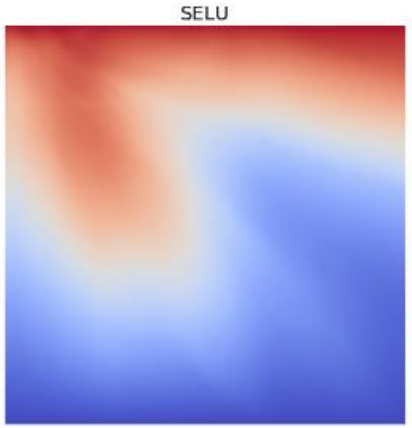

Derivatives of Mish and Swish
1st Derivative of Mish
2nd Derivative of Mish
1st Derivative of Swish
2nd Derivative of Swish
1.0
0.8
0.6
0.4
0.2
0.0
−3
−2
−1
0
1
2
3

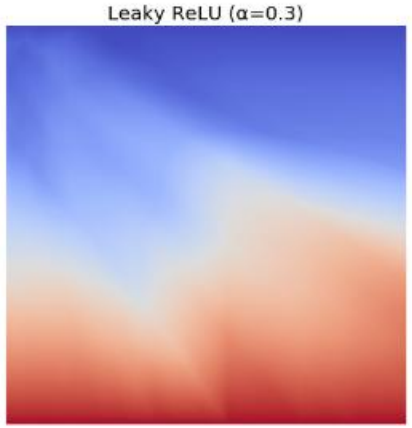

Derivatives of Mish and Swish (Increased Range)
1st Derivative of Mish
1st Derivative of Swish
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

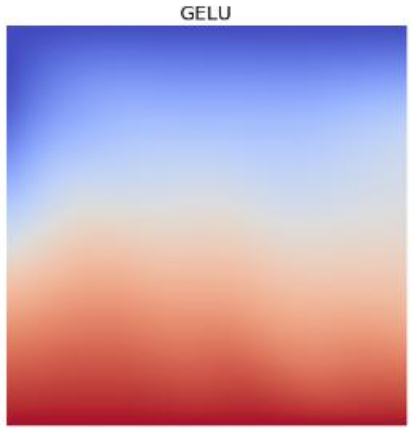


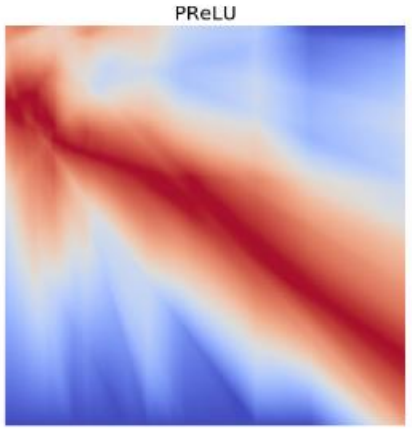


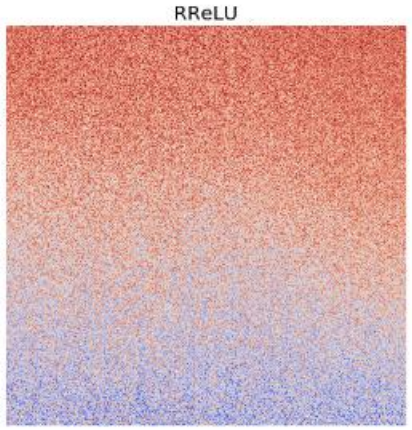






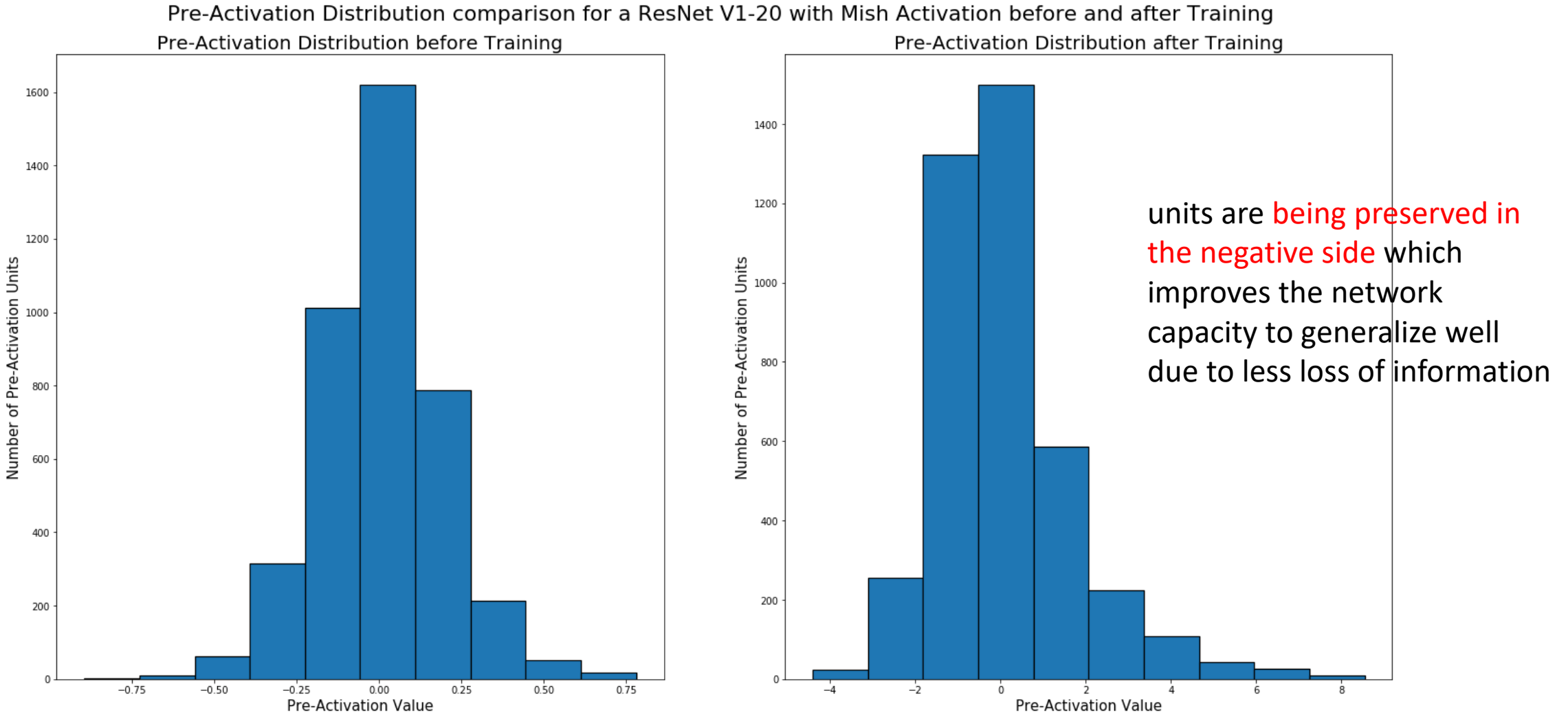






Mish:

Smoother transition results in smoother loss functions which are easier to optimize and hence the network generalizes better



units are being preserved in the negative side which improves the network capacity to generalize well due to less loss of information

Configurations	Parameters
Model	Squeeze Excite ResNet-50 (SENet-50)
Dataset	CIFAR-10
Batch Size	128
Epoch	100
Optimizer	Adam
Learning Rate	0.001

Activation Function	Testing Top-1 Accuracy	Loss	Testing Top-3 Accuracy
Mish	**90.7931%**	**4.75271%**	98.5562%
Swish-1	90.558%	4.76047%	98.6748%
E-Swish (β = 1.75)	90.5063%	5.22954%	98.6946%
ReLU	90.447%	4.93086%	98.6155%
GELU	90.5063%	5.0612%	**98.754%**
SELU	86.432%	6.89385%	97.8936%

Summary: Which One to Choose

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid
- Modern activation
  - Swish/GELU/SiLU used in transformers/LLM
    - Smoothness:平滑的活化函數意味著梯度的變化是連續且平緩的。這有助於形成一個更平滑的損失地貌（Loss Landscape），讓優化器（如 Adam）更容易找到好的解，避免在訓練過程中因梯度突變而產生震盪。對於需要長時間、在海量數據上進行訓練的 LLM 而言，訓練的穩定性至關重要。
    - 更豐富的資訊表達： 平滑的曲線允許梯度在接近零的區域有更細微的變化，這有助於模型學習到更精細的特徵。
  - ReLU is preferred for low cost implementation and CNN

Source: CS231n

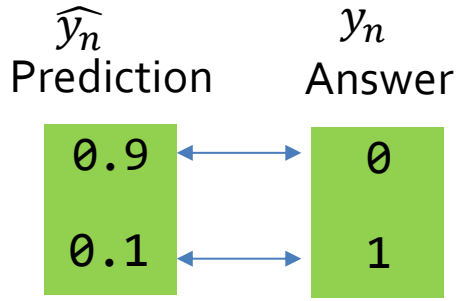
- 0. 資料前處理
- 1. 網路結構
- 2. 決定模型的 loss function
- 3. 訓練相關設定參數 (選擇optimizer)
- 4. 編譯模型 (Compile model)
- 5. 開始訓練囉！(Fit model)

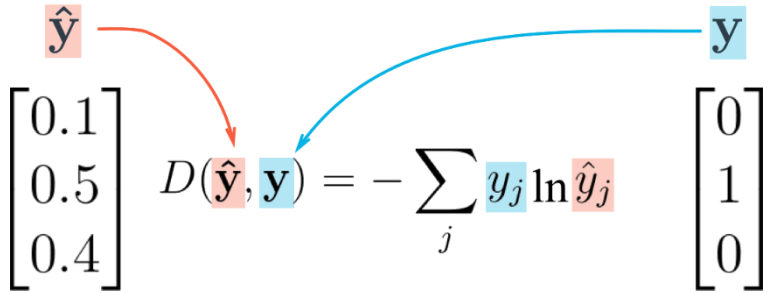
LOSS FUNCTION

MEASURING HOW WELL YOUR ALGORITHM IS DOING ON YOUR DATASET

2.決定模型的 loss function:分類問題

- Cross entropy loss
  - 屬於該類的機率
- binary_crossentropy (logloss): 兩類
  - $-\frac{1}{N}\sum_{n=1}^{N}[y_n log(\widehat{y_n}) + (1-y_n) log(1-\widehat{y_n})]$
  - $-\frac{1}{2}[0*\log(0.9)+(1-0)*\log(1-0.9)+1*\log(0.1)+0*\log(1-0.1)]$
  - $=-\frac{1}{2}[\log(0.1)+\log(0.1)]=-\log(0.1)=2.302585$
- categorical_crossentropy: 多類 (>2)
  - 需要將 class 的表示方法改成 one-hot encoding
    - Category 1 ➔ [0,1,0,0,0]
    - Not need for pytorch
  - 用簡單的函數 keras.np_utils.to_category(input) 轉換





Source:手把手教你深度學習實務

2.決定模型的 loss function: regression 問題

Prediction	Answer
0.8	0.9
0.2	0.1

- mean_squared_error
  - $[(0.9-0.8)^2+(0.1-0.2)^2]/2=0.01$
- mean_absolute_error
  - $[|0.9-0.8|+|0.1-0.2|]/2=0.1$
- mean_absolute_percentage_error
  - $[|0.9-0.8|\ /\ |0.9|+|0.1-0.2|/\ |0.1\ |\ ]*100/2=55$
- mean_squared_logarithmic_error
  - $[\log(0.9)-\log(0.8)]^2+[\log(0.1)-\log(0.2)]^2]*100/2=0.247$

$$\text{RMSE}=\sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j-\hat{y}_j)^2} \qquad \text{MAE}=\frac{1}{n}\sum_{j=1}^{n}|y_j-\hat{y}_j|$$

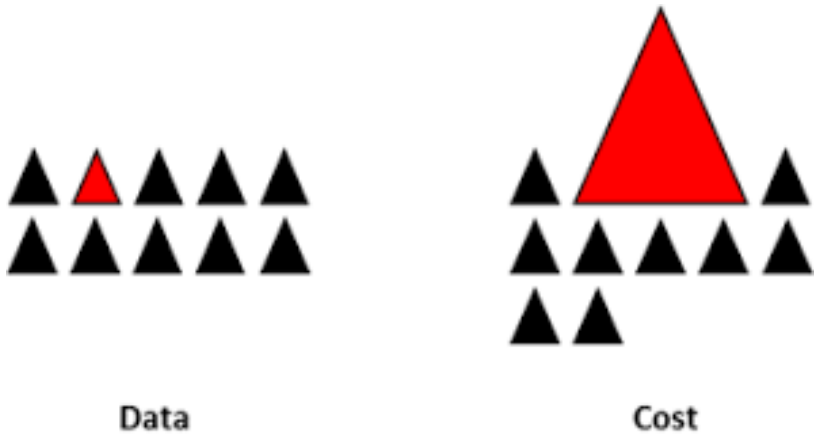
Source:手把手教你深度學習實務

Customized Loss Function

- Unbalanced data
  - Focal loss
- Multi-task learning
  - Object detection
  - Face recognition

⇒Metric learning

⇒Center loss, triplet loss, (in face identification)



Which Loss Function to Choose

- Depending on your applications
- Classification problem
  - Cross-entropy loss is the first choice
  - Hinge loss also works well
- Regression problem
  - L1 or L2 (mean square error) loss is the first choice

3. 選擇 optimizer 與參數

- 基本款 optimizer SGD

```
# 指定 optimizier
from keras.optimizers import SGD, Adam, RMSprop, Adagrad
sgd = SGD(lr=0.01,momentum=0.0,decay=0.0,nesterov=False)
```

  - 設定 learning rate, momentum, learning rate decay, Nesterov momentum
  - 以上是預設值，通常預設就很好用
- 動量型: Momentum/Nesterov Momentum
- 個別參數調整型: RMSprop
- 混合動量型與個別參數調整型: Adam

Source:手把手教你深度學習實務

4. 編譯模型 (Compile model)

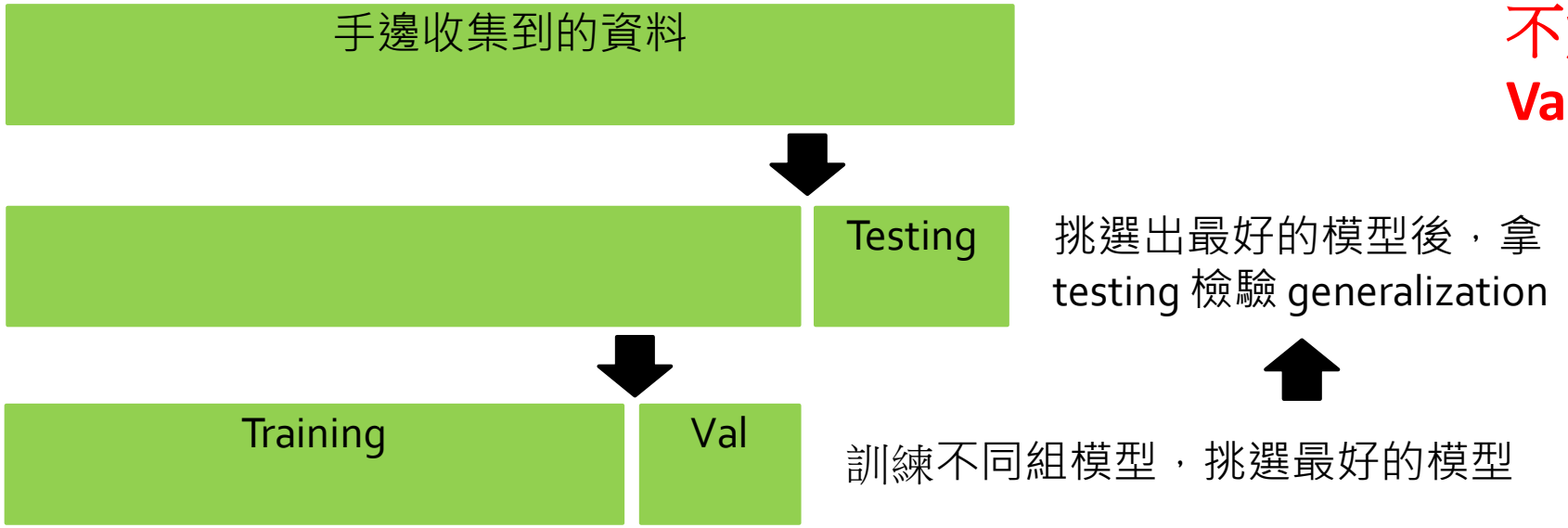
```
# 指定 loss function 和 optimizier
model.compile(loss='categorical_crossentropy',
              optimizer=sgd)
```

- Tensorflow/Keras/caffe use static computation graph
  - Need compilation before execution
- Pytorch uses dynamic computation graph

Source:手把手教你深度學習實務

在開始訓練之前: 資料怎麼分

- 收集的資料分三分
  - Testing dataset: 真正跑分(大考)，檢驗模型的普遍性， 避免模型過度學習
  - Training dataset: 訓練(小考)時調網路權重和其他參數使用
  - Validation dataset:每次epoch訓練完，試跑分(模擬考)，用來挑選模型



不能作弊，偷看考題
Val/testing不能用在訓練

Source:手把手教你深度學習實務

Validation Dataset

- 兩種作法
- 利用 model.fit 的參數 validation_split
  - 從輸入(X_train,Y_train) 取固定比例的資料作為 validation
  - 不會先 shuffle 再取 validation dataset
  - 固定從資料尾端開始取
  - 每個 epoch 所使用的 validation dataset 都相同

- 手動加入 validation dataset
  - validation_data=(X_valid, Y_valid)

Source:手把手教你深度學習實務

Validation Dataset

- Two validation methods
  - Validate with splitting training samples
  - Validate with testing samples

```
''' training'''
# define batch size and # of epoch
batch_size = 128
epoch = 32

# [1] validation data comes from training data
fit_log = model.fit(X_train, Y_train, batch_size=batch_size,
                    nb_epoch=epoch, validation_split=0.1,
                    shuffle=True)
# [2] validation data comes from testing data
fit_log = model.fit(X_train, Y_train, batch_size=batch_size,
                    nb_epoch=epoch, shuffle=True,
                    validation_data=(X_test, Y_test))
```

5. 開始訓練囉！(Fit model)

```
# 指定 batch_size, nb_epoch, validation 後，開始訓練模型!!!
history = model.fit( X_train,
                     Y_train,
                     batch_size=16,
                     verbose=0,
                     epochs=30,
                     shuffle=True,
                     validation_split=0.1)
```

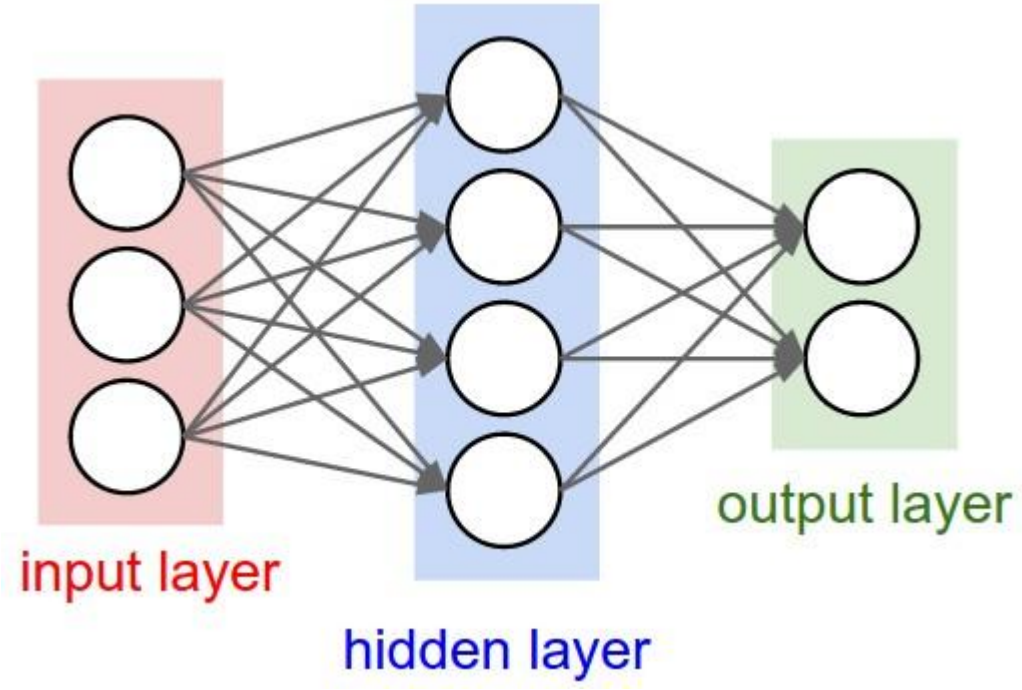
- 給訓練資料與驗證資料後，開始訓練
- batch_size: min-batch 的大小
- nb_epoch: epoch 數量
  - 1 epoch 表示看過全部的 training dataset 一次
- shuffle: 每次 epoch 結束後是否要打亂 training dataset
- verbose: 是否要顯示目前的訓練進度，0 為不顯示

Source:手把手教你深度學習實務

WEIGHT INITIALIZATION

Pitfall: all zero initialization

- Q: what happens when W=0 init is used?
  - Same output, same gradient, same update, no learning at all



Source: CS231n

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Source: CS231n

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

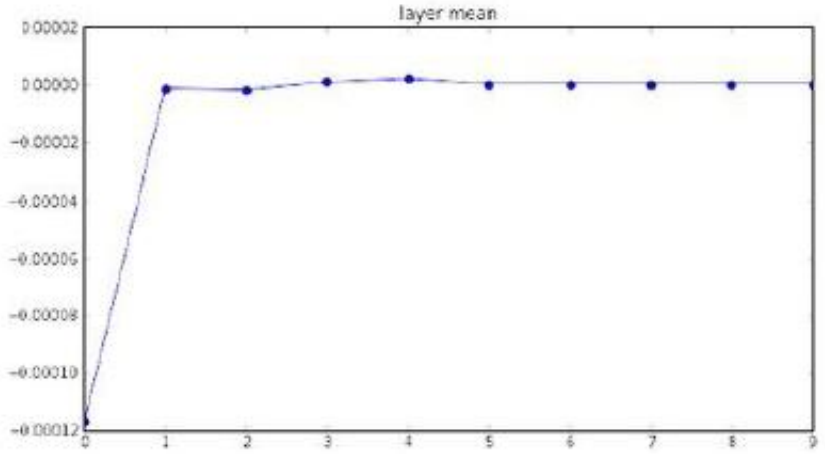
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

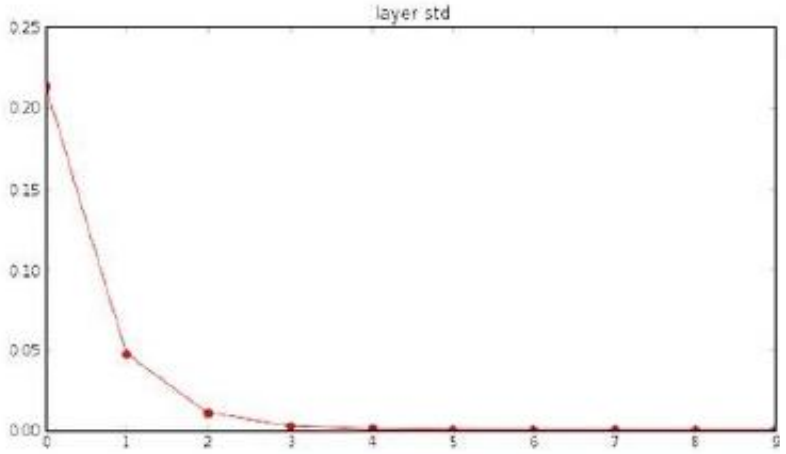
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

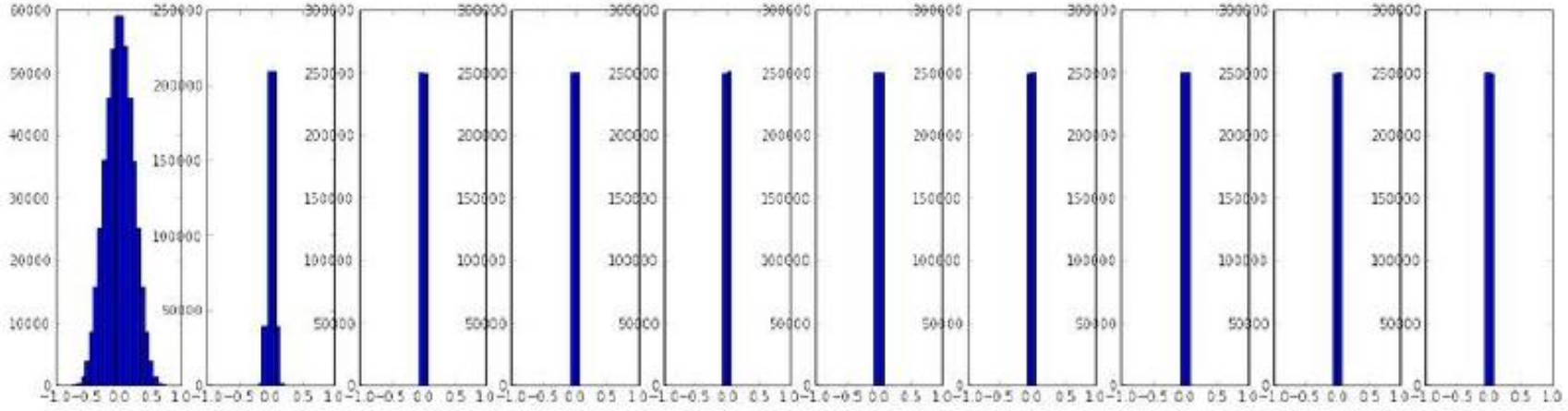
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Source: CS231n

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```







All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.

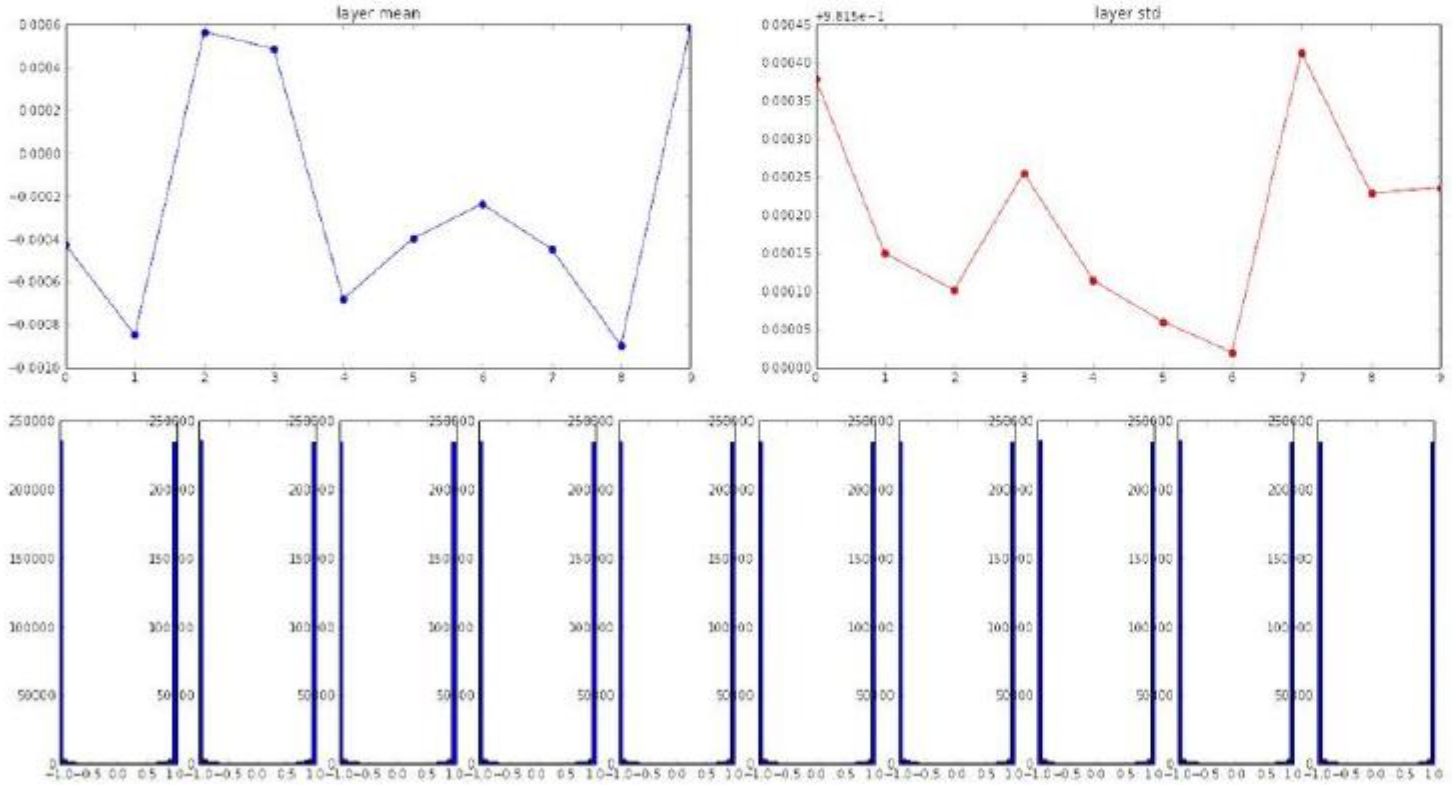
$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial \theta}$$

Source: CS231n

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01



Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Note. Tanh activation function

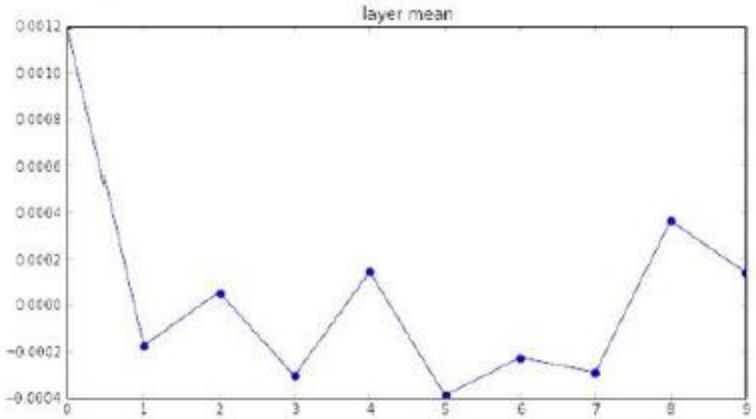
Source: CS231n

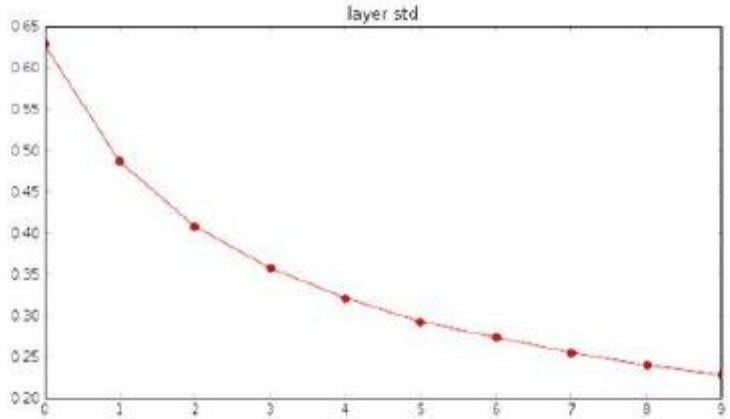
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

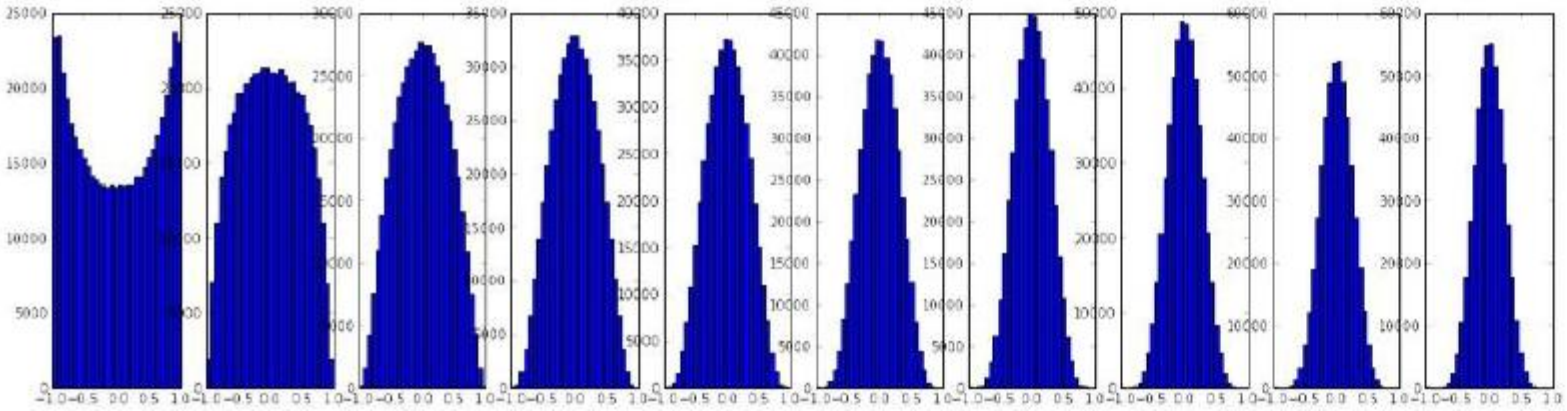
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]

保持輸入和輸出的方差一致
避免了所有輸出值都趨向於0







Reasonable initialization.
(Mathematical derivation assumes linear activations)

$Var(Y)=Var(W_1X_1+\cdots+W_nX_n)=nVar(W_i)Var(X_i)$

Keep I/O variance the same
Var(w) =1/n, n: number of input

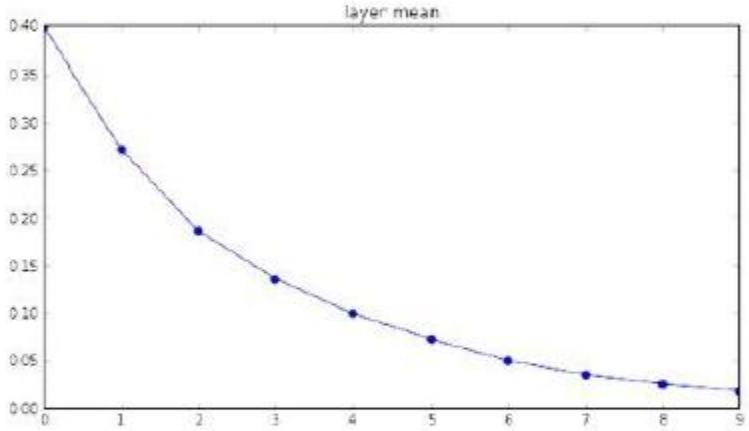
(linear region of tannh)
Derivation, see
http://cs231n.github.io/neural-networks-2/

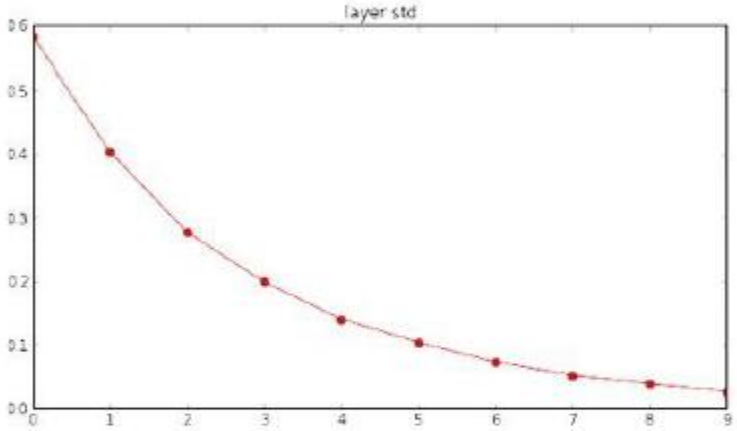
Source: CS231n

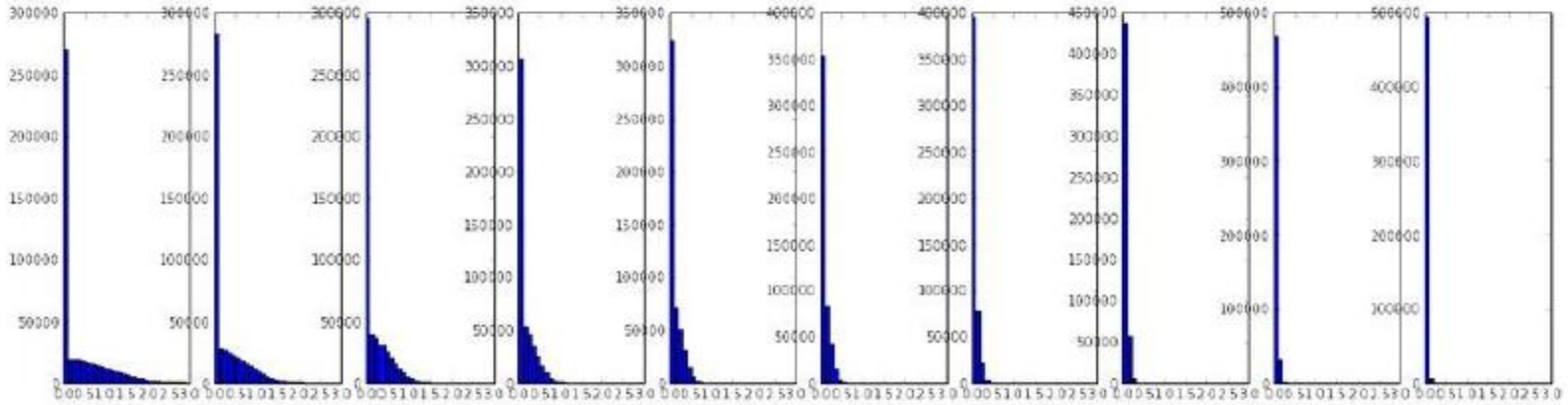
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



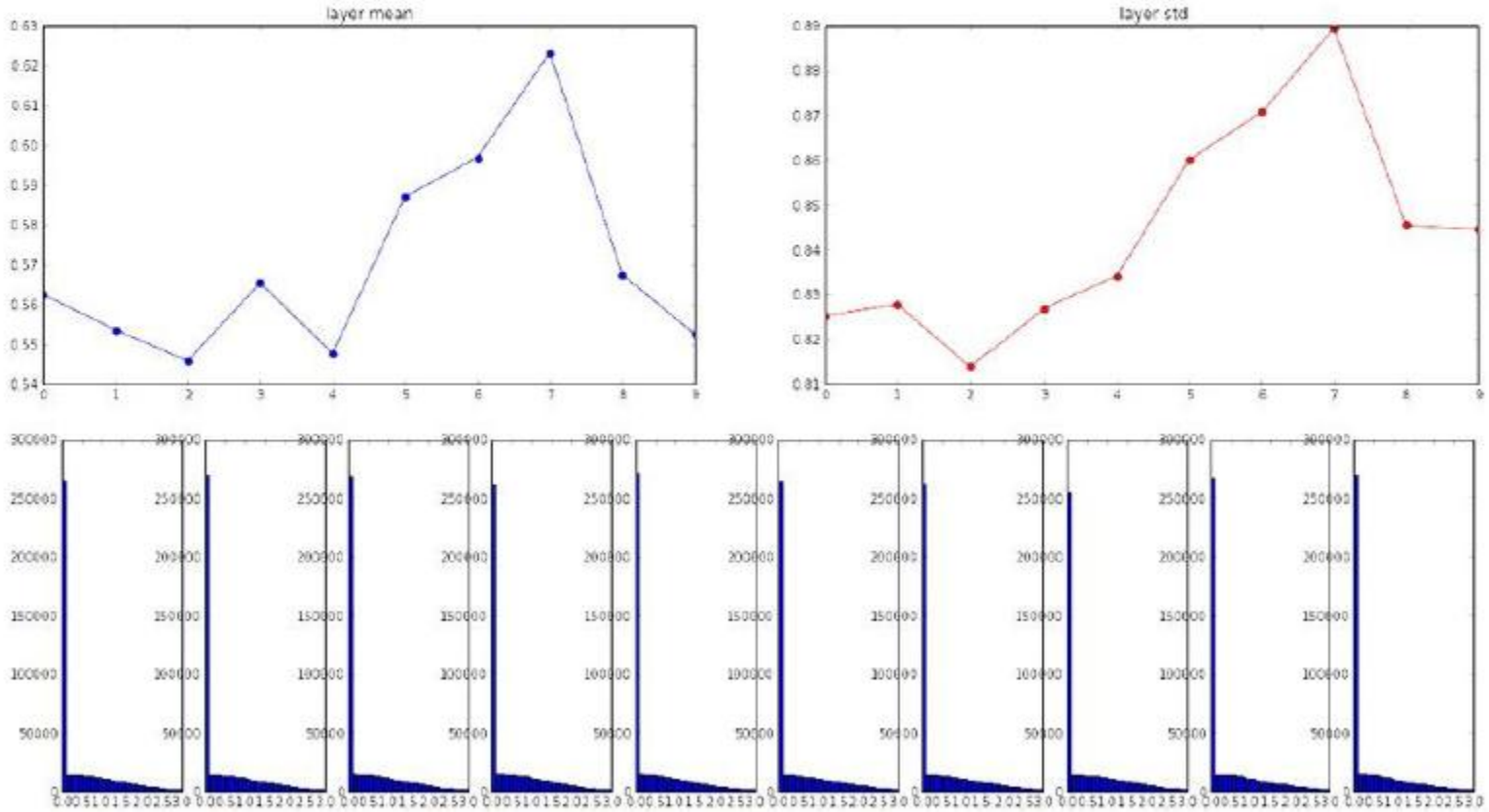




Source: CS231n

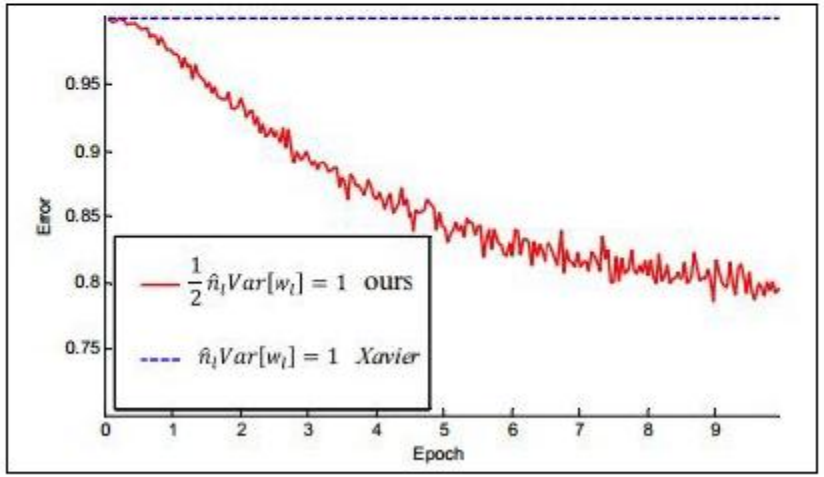
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```



He et al., 2015 (note additional /2)

Half of the neuron is killed in RELU



Source: CS231n

Other Initialization Methods

- Pre-training
  - with autoencoder: Not used anymore
  - With a pre-trainied model => fine tuning

Source: CS231n

TLDR

- **核心目標：**
  - 讓訊號與梯度在層間保持合適的方差，避免**消失 / 爆炸**。
  - 好的初始化能**縮短 warmup、減少不必要的正規化負擔**，並增加深網可訓練性。

- For ReLU activation function (e.g. CNN),
  - use He initialization
- tanh/線性等近對稱激活常用, transformer
  - **Xavier/Glorot initialization**

Source: CS231n