Machine Learning

Neural Networks

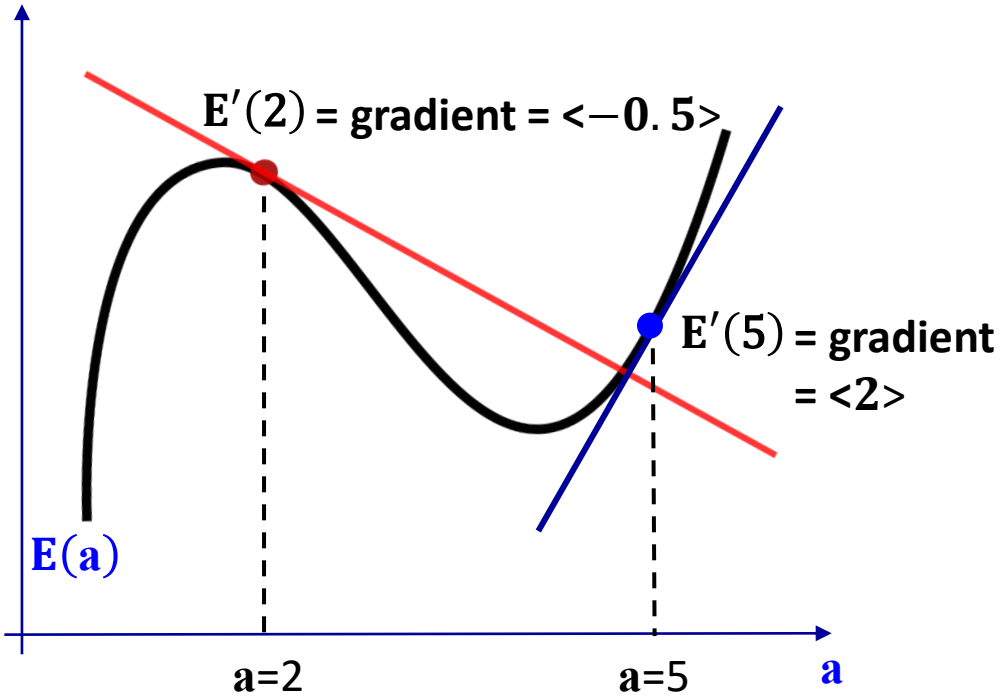
Dong-Kyu Chae

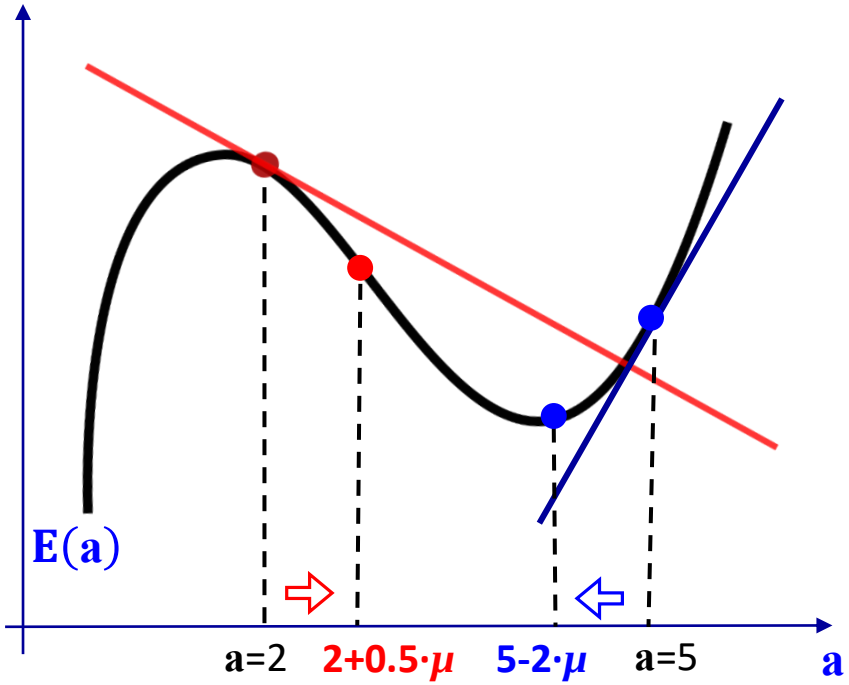
Dept. of Computer and Software

Hanyang University

Last Class Summary

- **Model: a function that maps inputs to an output**
- **Cost function: computing difference between the model's predictions and actual values: $\mathbf{E}(\Theta) = \sum(y_i - f(x_i; \Theta))^2$**
- **Gradient descent:**
  - Taking steps to the opposite direction of the gradient

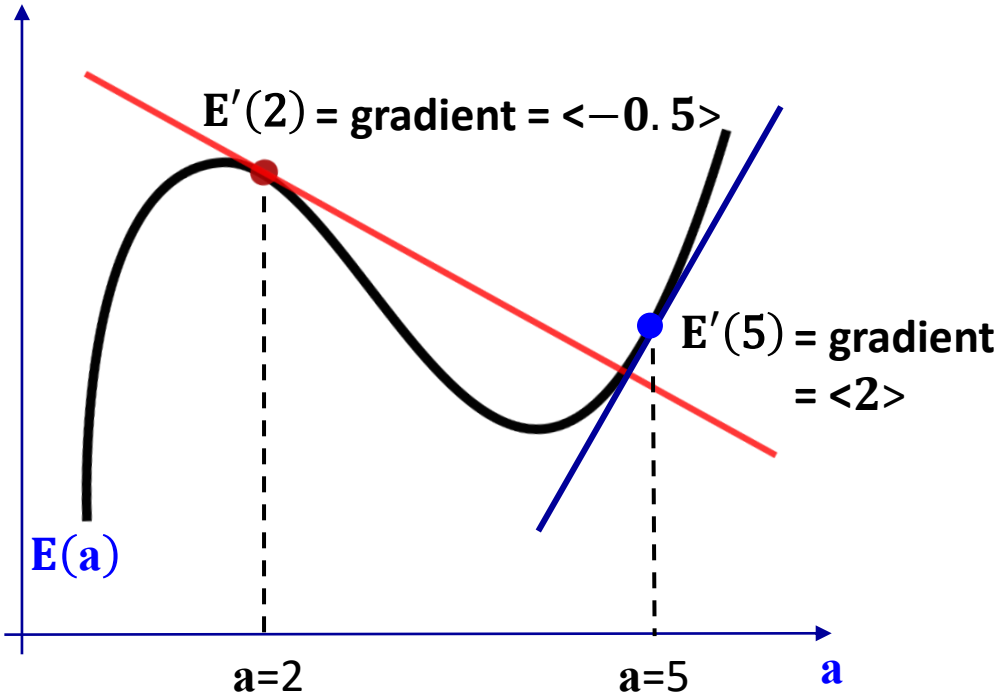


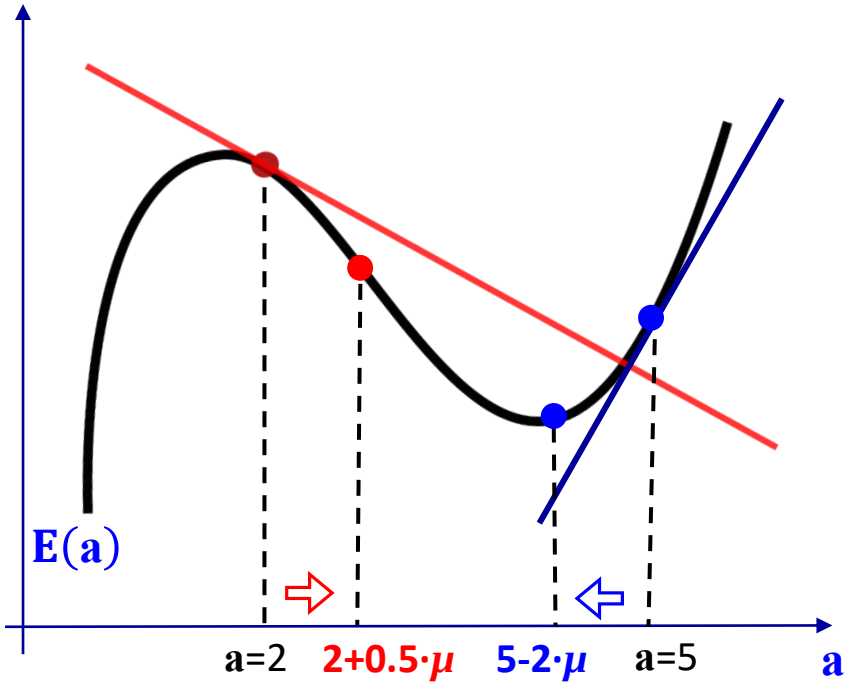


Last Class Summary

☐ Gradient descent algorithm:

1. Initialize model parameters Θ
2. Compute gradient by using all data instances: $\frac{\partial E}{\partial \Theta}$
3. Update model parameters: $\Theta_{t+1} \leftarrow \Theta_t - \mu \frac{\partial E}{\partial \Theta}$
4. Repeat 2, 3 until convergence

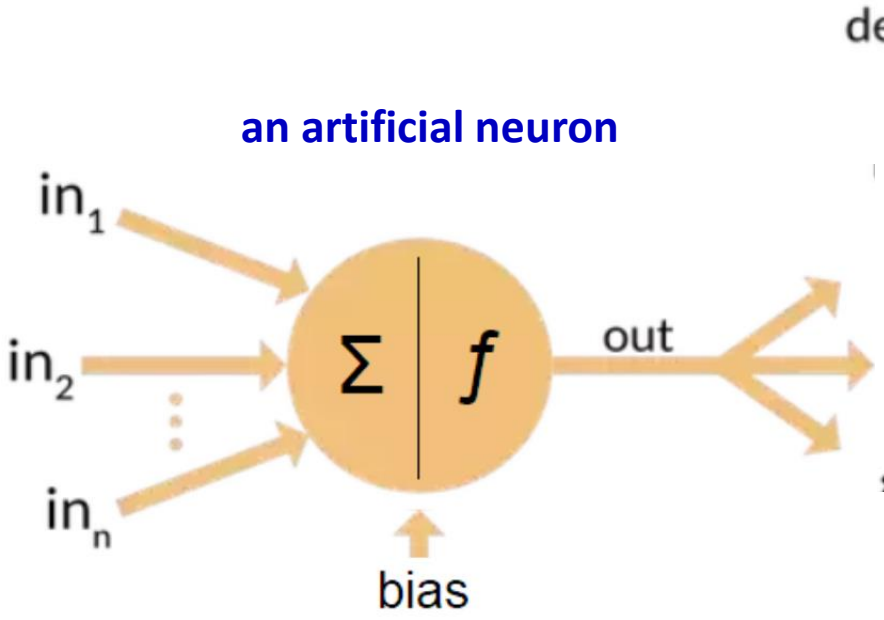


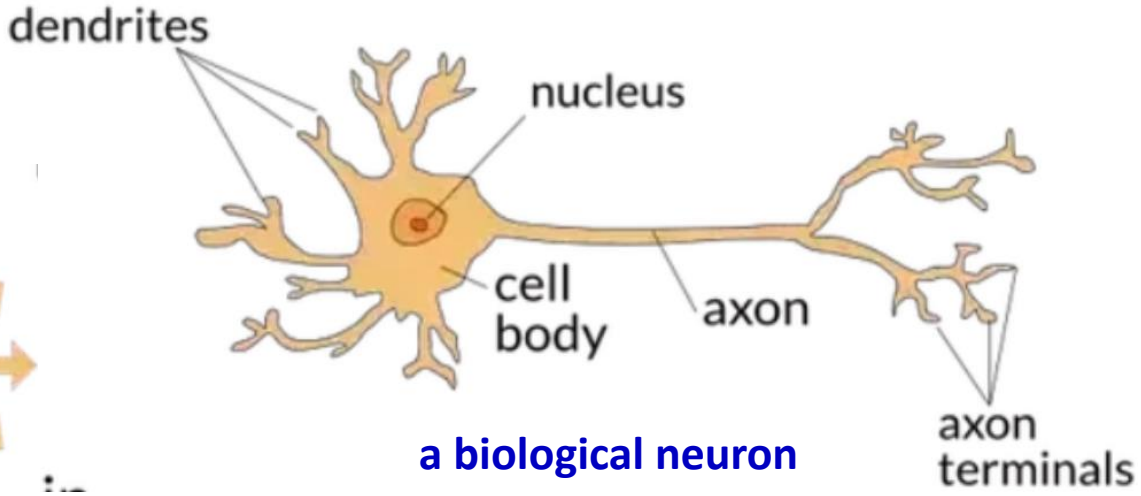


Intro

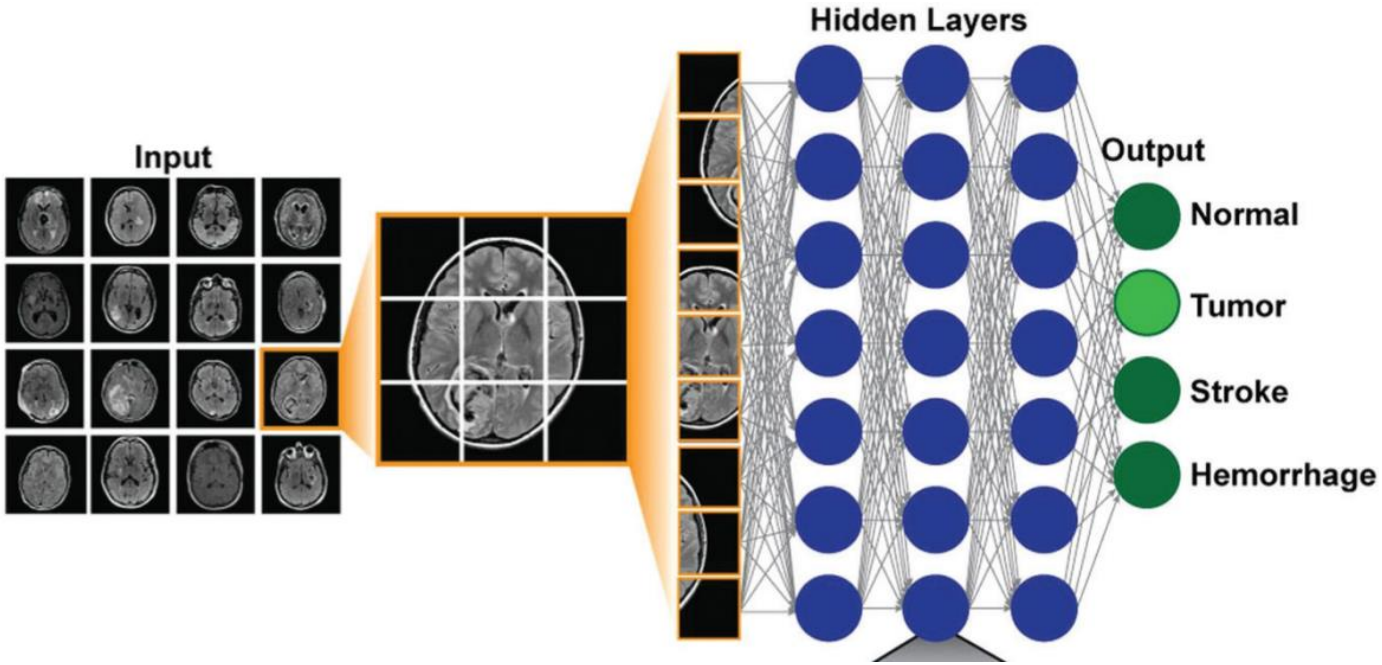
(Artificial) neural networks

- https://en.wikipedia.org/wiki/Artificial_neural_network
- Artificial neural networks (ANNs), usually simply called **neural networks** (NNs), are computing systems vaguely inspired by the **biological neural networks that constitute animal brains**.
- **Artificial neurons** are elementary units in an artificial neural network.

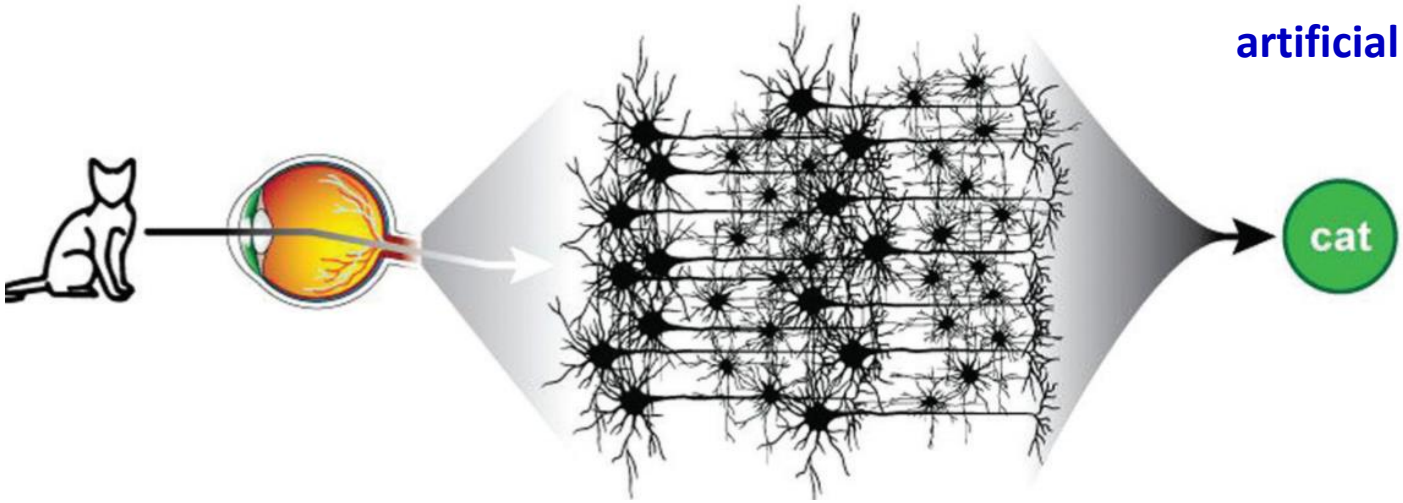




Intro



artificial neural networks

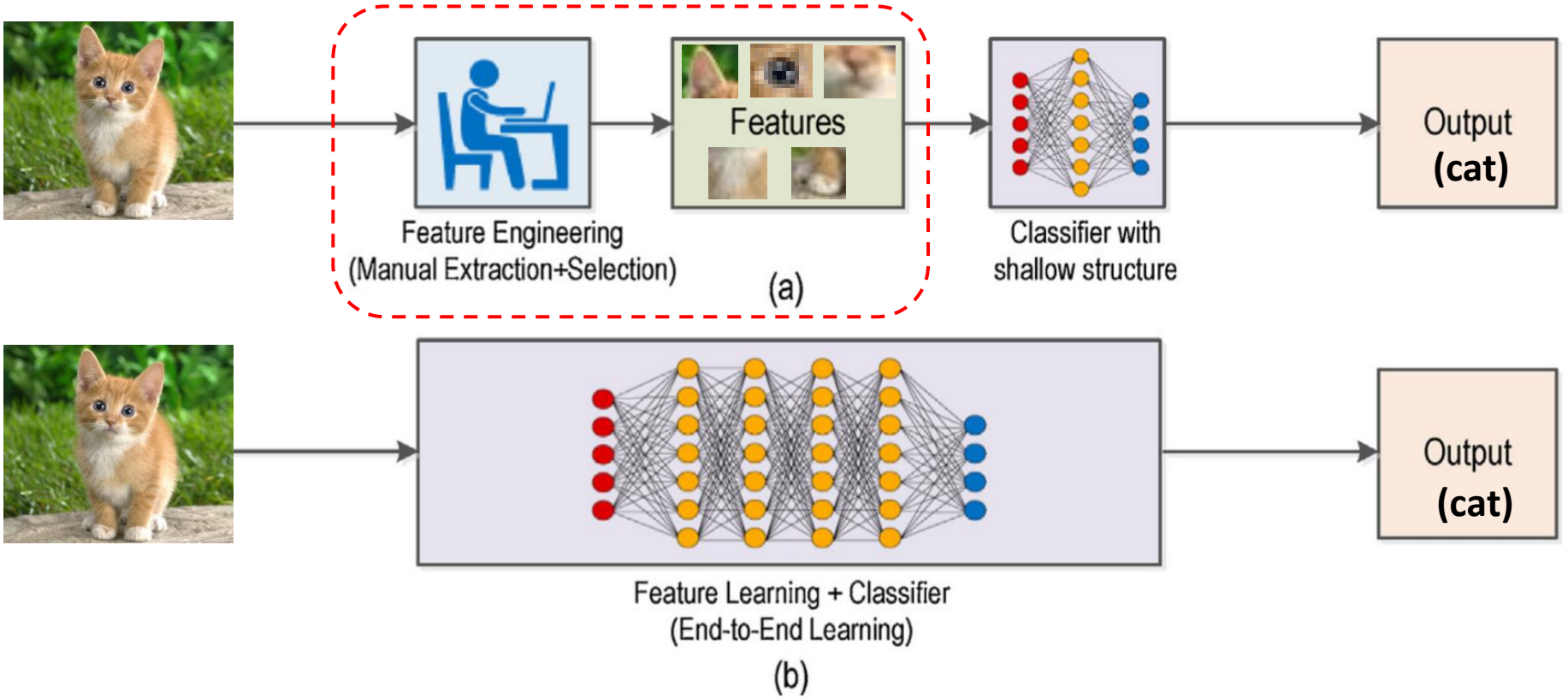


biological neural networks

Keywords

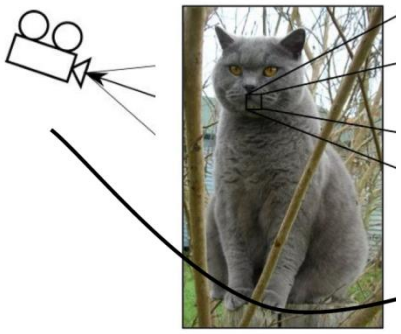
☐ Representation learning VS Feature engineering

- Example: Building cat detector
- Feature engineering tries to **manually find** informative and discriminating features from raw data

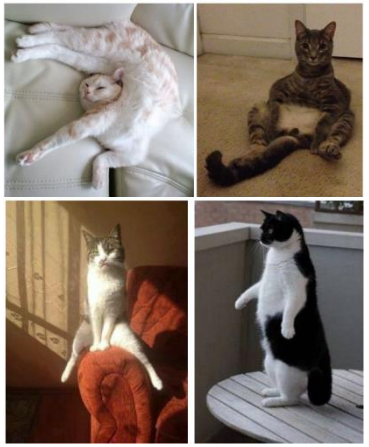


Keywords

- Difficulty of feature engineering: Which features should be used for recognizing “cats” accurately?



Viewpoint Variation



Deformation



Occlusion



Background clutter

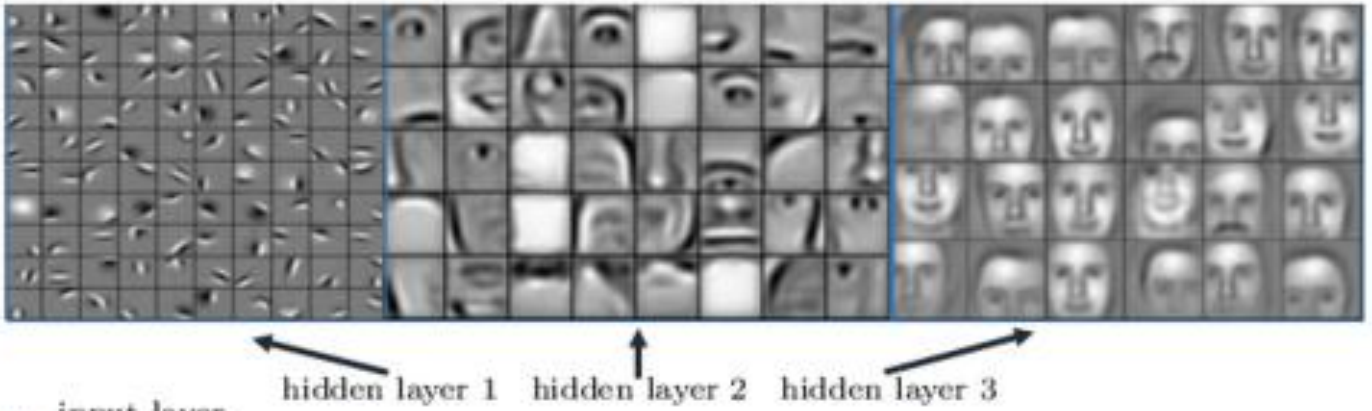


Intraclass variation

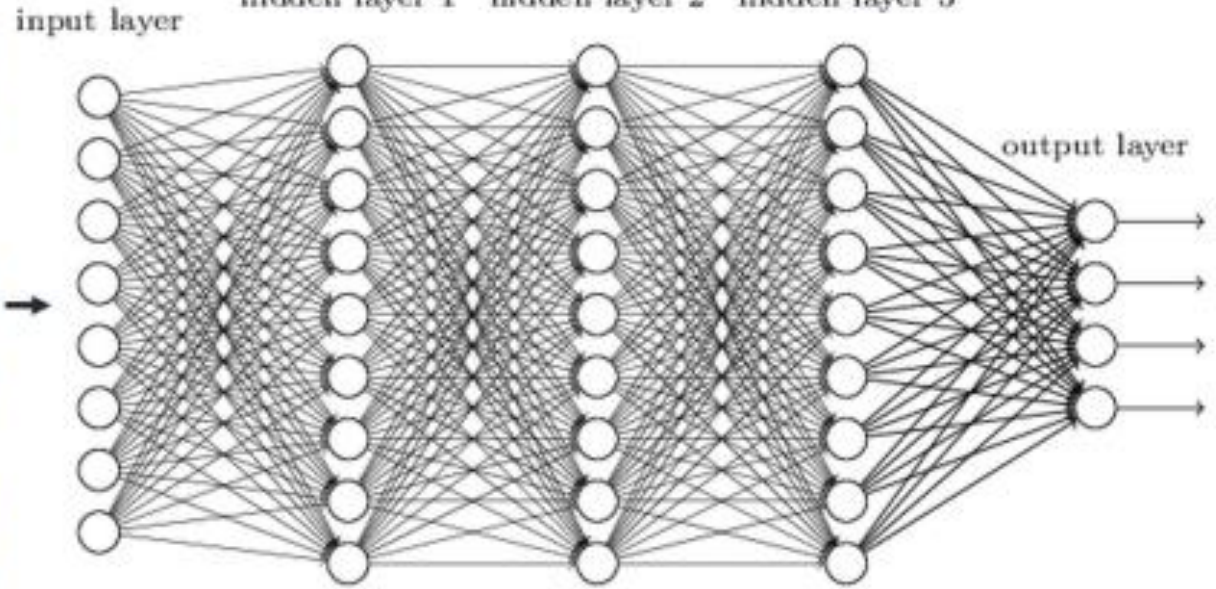
Keywords

- **Deep learning is representation learning**
  - It **automatically learns representation** which are the best for given classification task

Deep neural networks learn hierarchical feature representations

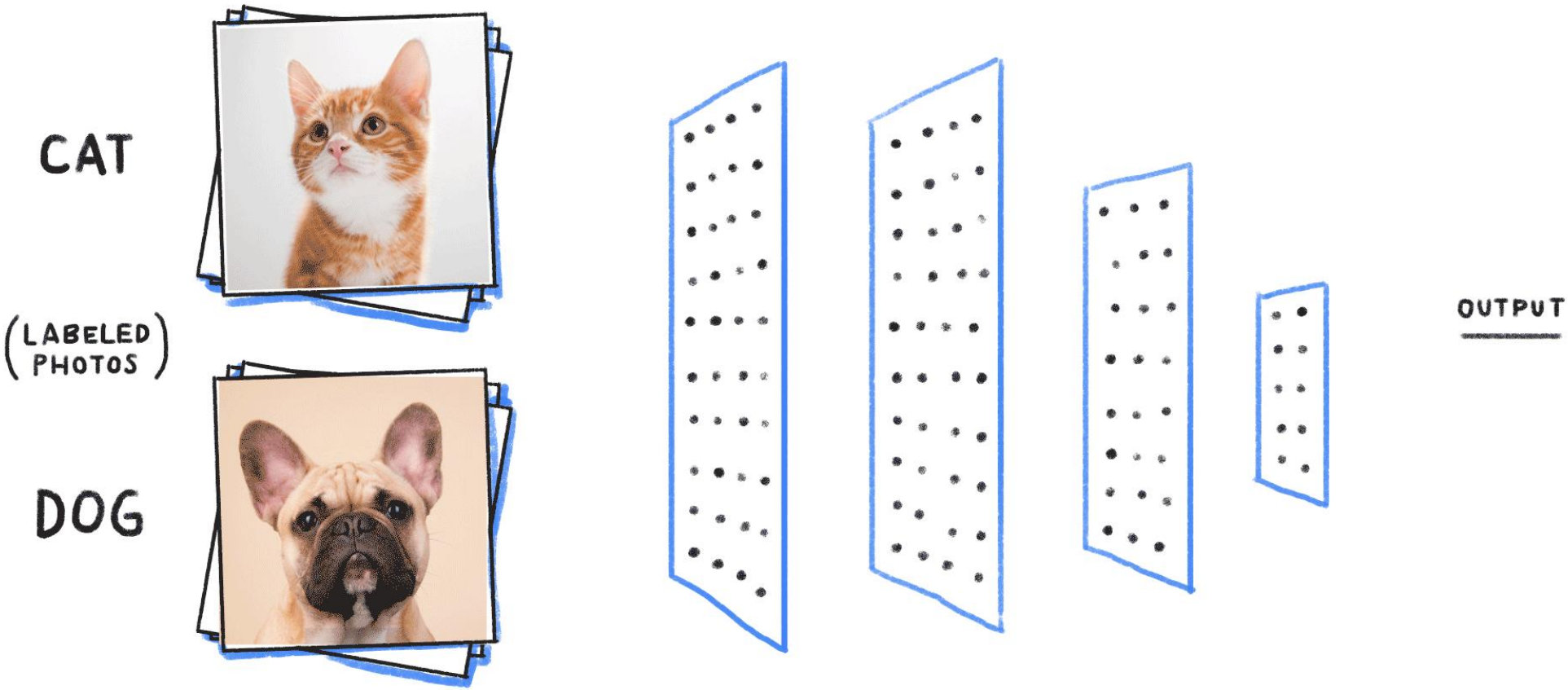






Keywords

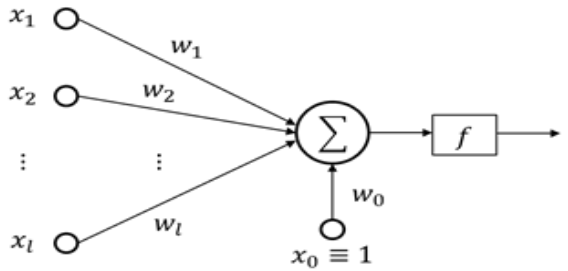
- Representation learning is also known as **data-driven feature learning** & **end-to-end** learning



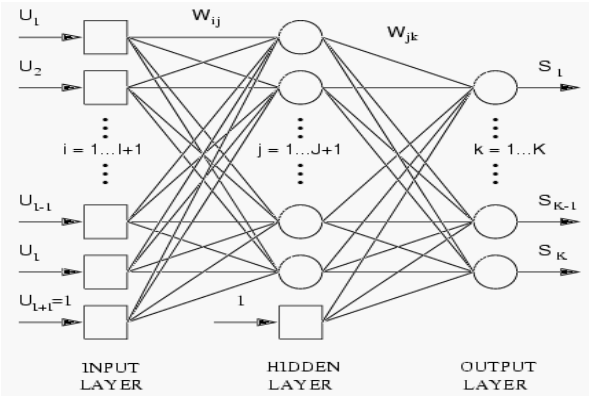
Neural Network

History

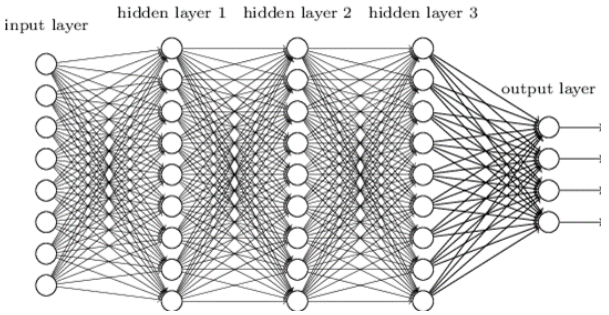
- A neuron (or a perceptron) (1958~)



- Multi-layer neural network (1986~)



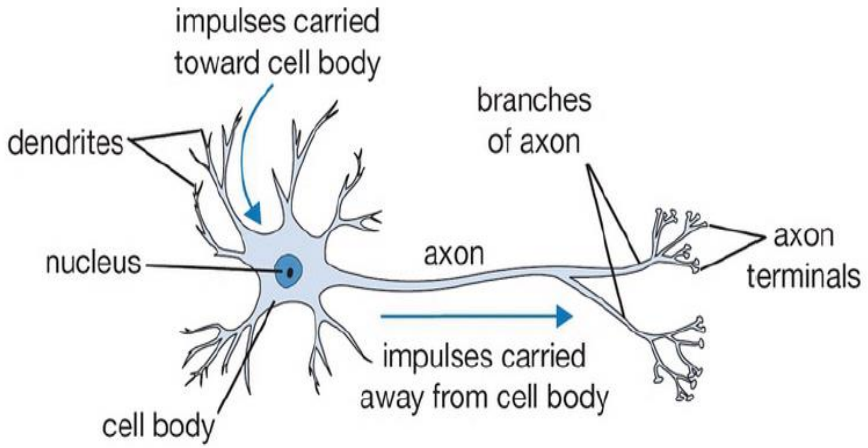
- Deep learning (2006~)

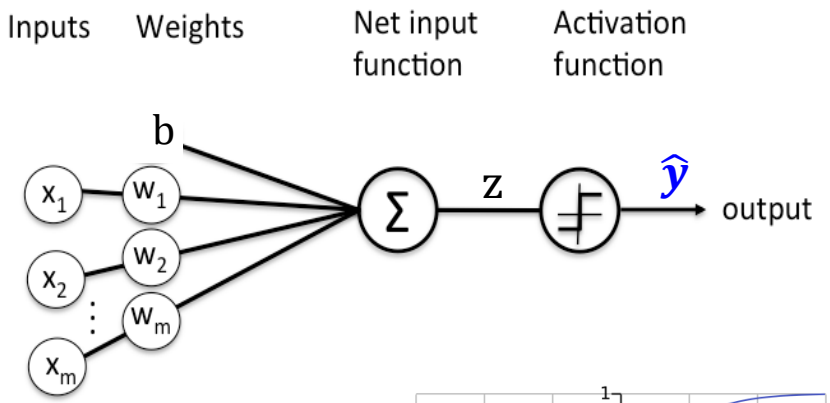


A Neuron (Perceptron)

Artificial neuron (or just neuron, or perceptron)

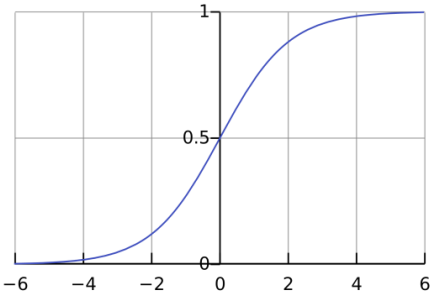
- It is a model inspired by biological neuron
- Composed of **(1) weighted sum of inputs** and **(2) an activation function** given the summated input





$$z = \sum_{i=1}^{n} x_i w_i + \mathrm{b}$$

$$\hat{y} = \sigma(z)$$



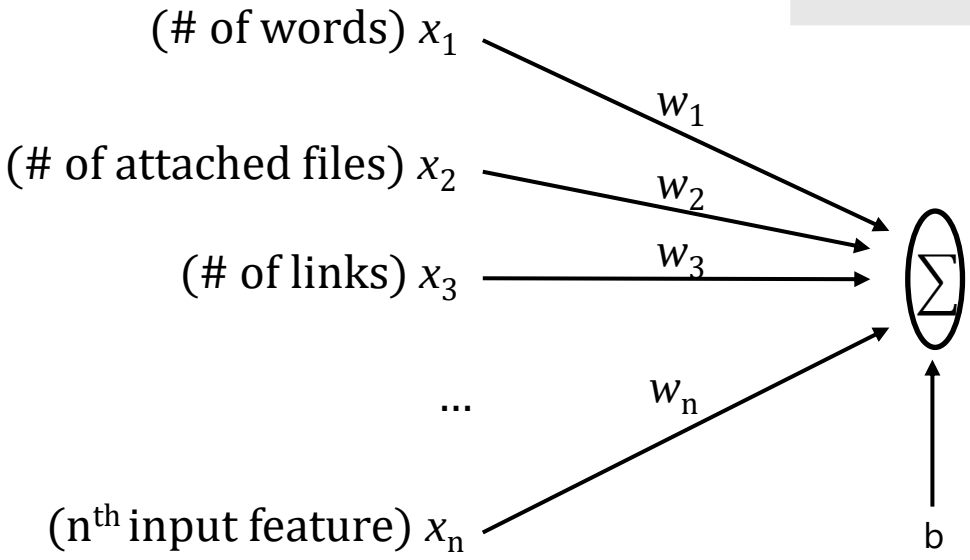
$\sigma(z)$: a sigmoid function

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

A Neuron (Perceptron)

- **Example**
  - **Input data**: emails
  - **Output**: spam/normal
    - **spam**: 1
    - **normal**: 0

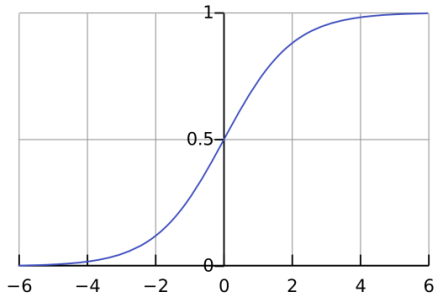
	# of words	# of attached files	# of links	# of malicious words	.....	spam
mail #1	256	0	3	7	.....	1 (Yes)
mail #2	56	1	0	3	.....	0 (No)
mail #3	24	1	0	1	.....	0
mail #4	672	0	0	0	.....	0
mail #5	67	2	4	3	.....	1
mail #6	48	0	2	6	.....	0
mail #7	79	1	3	8	.....	1
			.....			



$$z = \sum_{i=1}^{n} x_i w_i + b$$

$$\hat{y} = \sigma(z)$$

1 (spam, if $\hat{y} > 0.5$)
0 (normal, if $\hat{y} \leq 0.5$)



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

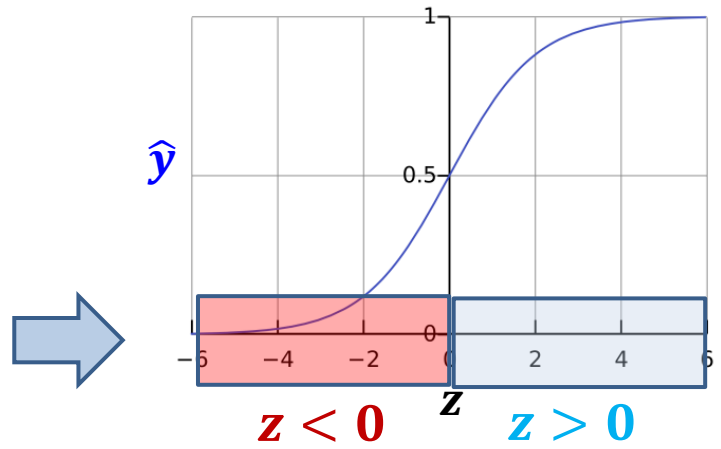
A Neuron (Perceptron)

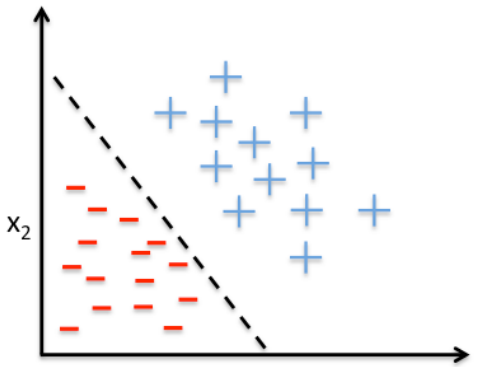
A neuron can perform linear classification

- It finds the **optimal decision boundary** that divides two different class labels as accurate as possible

$$z = w_1 x_1 + w_2 x_2 + b$$

$$\hat{y} = \sigma(z)$$



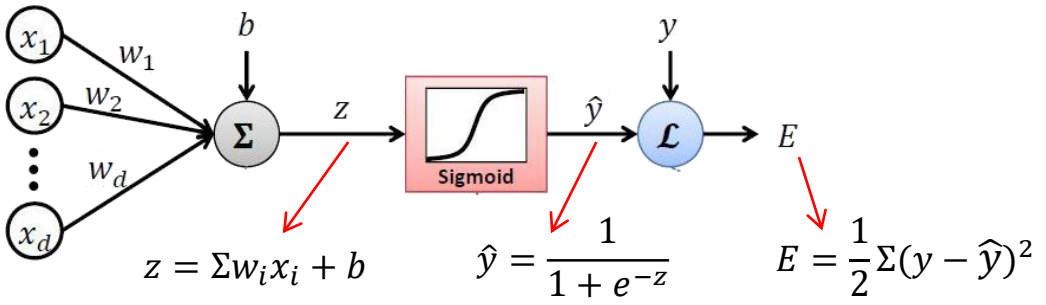


$$f(+) > 0$$
$$f(-) < 0$$

$$f() = w_1 x_1 + w_2 x_2 + b$$

A Neuron (Perceptron)

- **Cost function** $E(\mathbf{W}) = \frac{1}{2}\sum_{k=1}^{n}(y - \hat{y}_k)^2$, **where** $\hat{y} = \frac{1}{1+e^{-z}}$ **and** $z = \sum_{i=1}^{d} x_i w_i + b$
- **Goal: W*** = $\underset{\mathbf{W}}{\operatorname{argmin}}\, E(\mathbf{W})$
- **Solution: gradient descent**
  - Initialize **W**
  - Repeat
    - $w_i = w_i - \mu \frac{\partial E}{\partial w_i}$ for each i=1~d,
    - b = b - $\mu \frac{\partial E}{\partial b}$



$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_i}$$

$$= -\Sigma(y - \hat{y})\,\hat{y}(1 - \hat{y})x_i$$

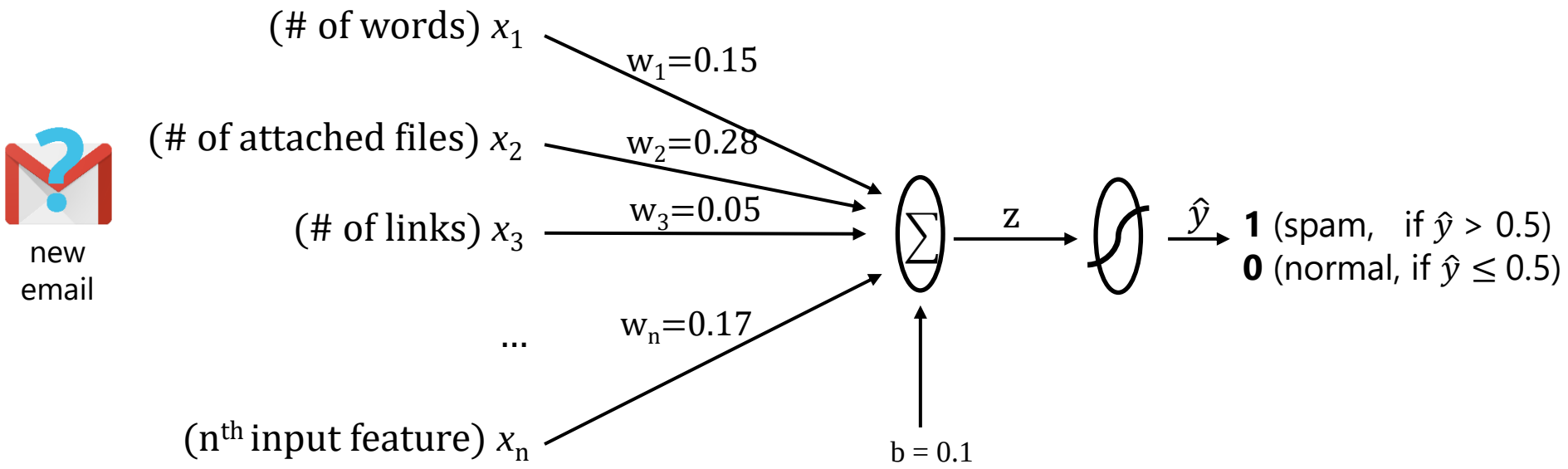
$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b} = -\Sigma(y - \hat{y})\,\hat{y}(1 - \hat{y})$$

$$\frac{\partial z}{\partial w_i} = x_i \qquad \frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y}) \qquad \frac{\partial E}{\partial \hat{y}_k} = -\Sigma(y - \hat{y})$$

$$\frac{\partial z}{\partial b} = 1$$

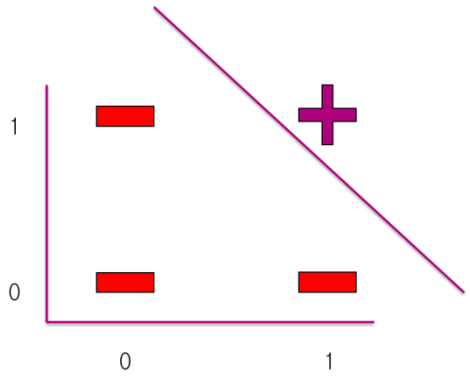
A Neuron (Perceptron)

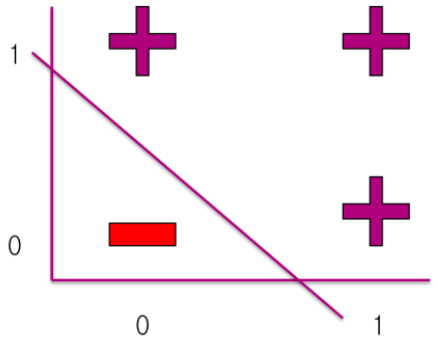
- **Classification (after training: testing)**



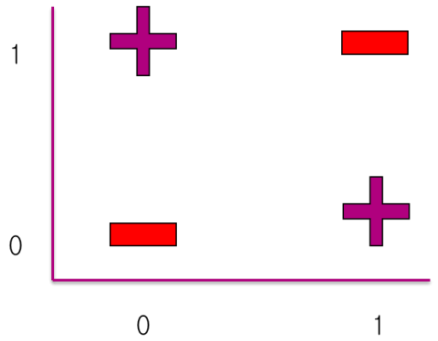
A Neuron (Perceptron)

- **Limitation of a neuron: linearity**



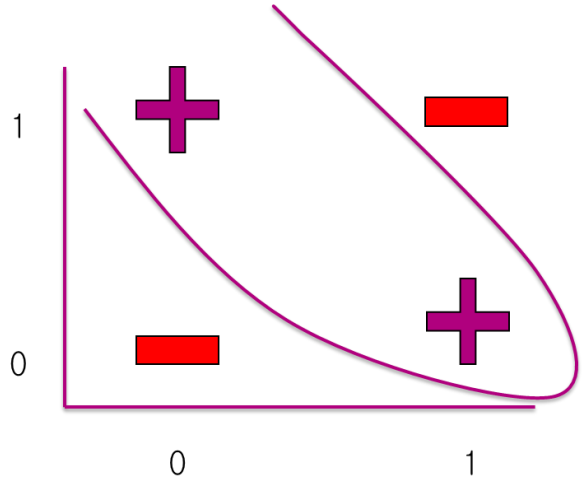


Linearly separable



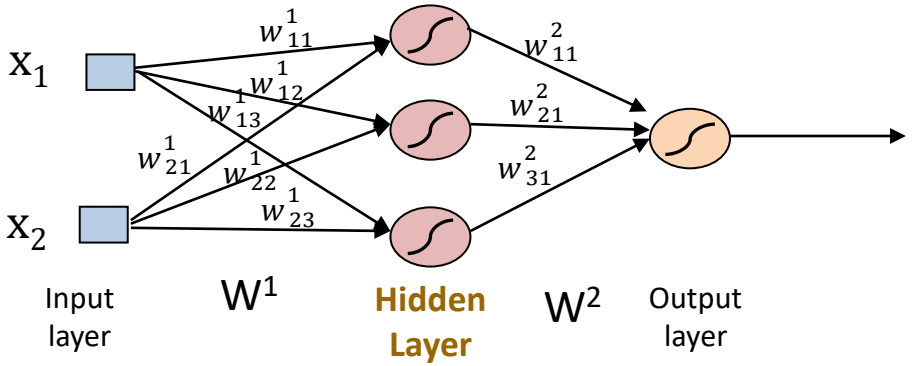
Not linearly separable

- **Solution: Non-linearity!**



Multi-layer Neural Network

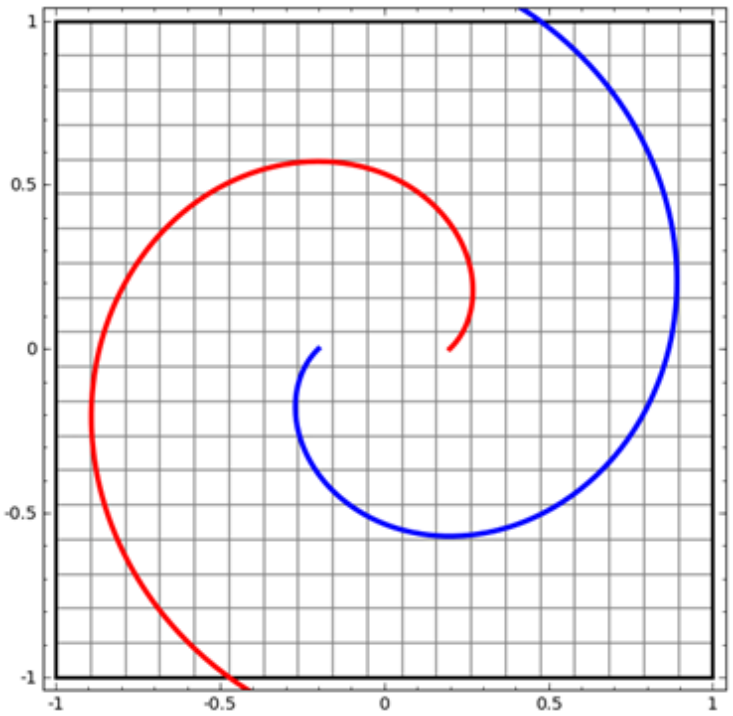
- **Stacking neurons with a non-linear activation function gives non-linearity**

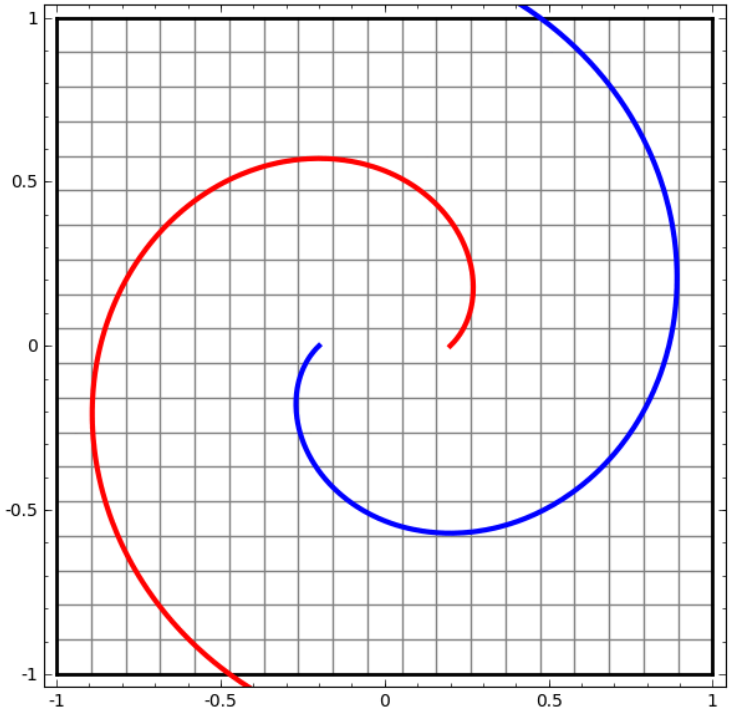


- **What hidden layers do**
  - They combine data from the previous layer in a variety of ways to create a new input data for the next layer.
  - They look for new representation of data that can be classified better by a linear classification
    - xW^1: linear transformation of input x $\quad (x_1 \quad x_2)\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$
    - Sigmoid function: non-linear transformation

Multi-layer Neural Network

- linear transform (xW) + non-linear transform (sigmoid)



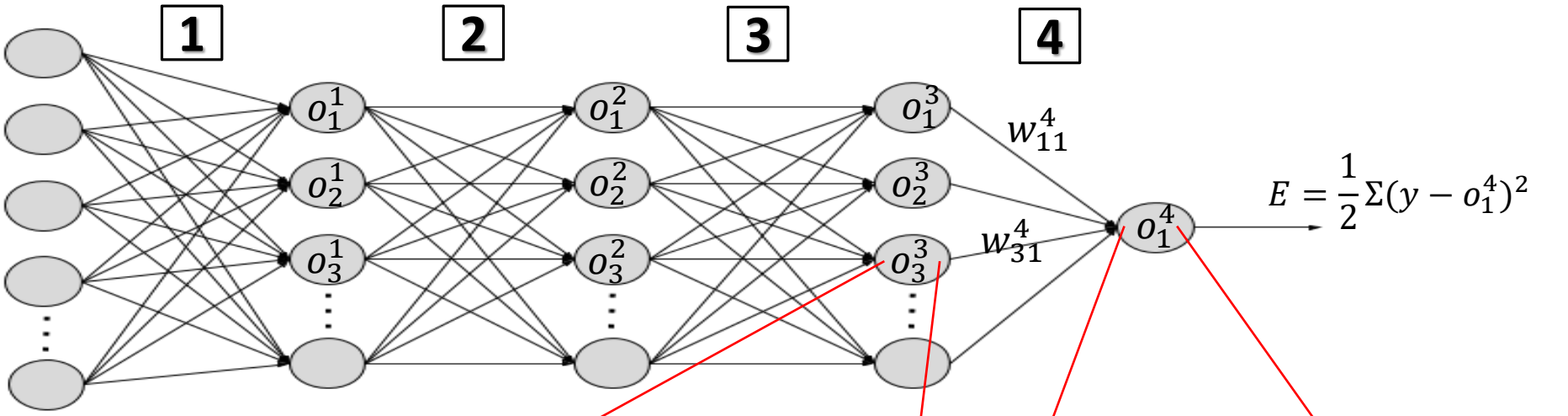


- More websites to practice nueral networks
  - http://cs.stanford.edu/people/karpathy/convnetjs//demo/classify2d.html
  - http://playground.tensorflow.org/

Multi-layer Neural Network

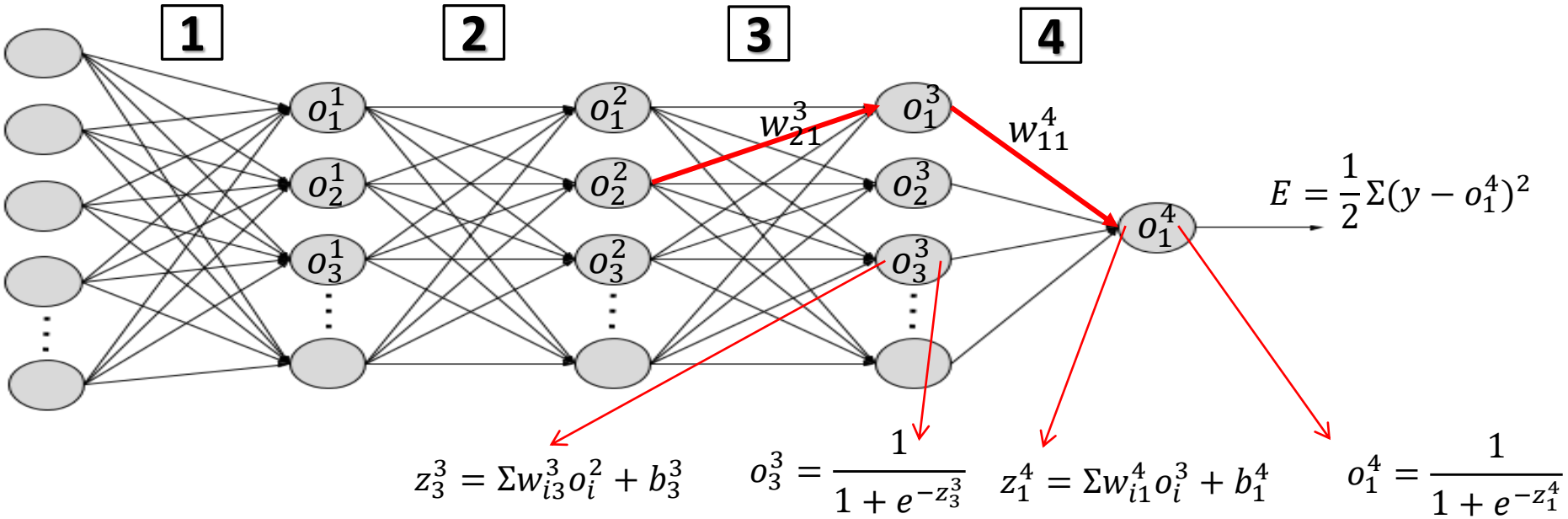
Computing gradient for multi-layer neural networks

- Background & notations



General term: $z_j^p = \Sigma w_{ij}^p o_i^{p-1} + b_j^p$ $\quad o_j^p = \frac{1}{1+e^{-z_j^p}}$

Multi-layer Neural Network



$$\frac{\partial E}{\partial w_{11}^4} = \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial w_{11}^4} = -\boldsymbol{\Sigma(y - o_1^4)o_1^4(1 - o_1^4)}o_1^3$$

$$\frac{\partial E}{\partial w_{21}^3} = \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial o_1^3}\frac{\partial o_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial w_{21}^3}$$

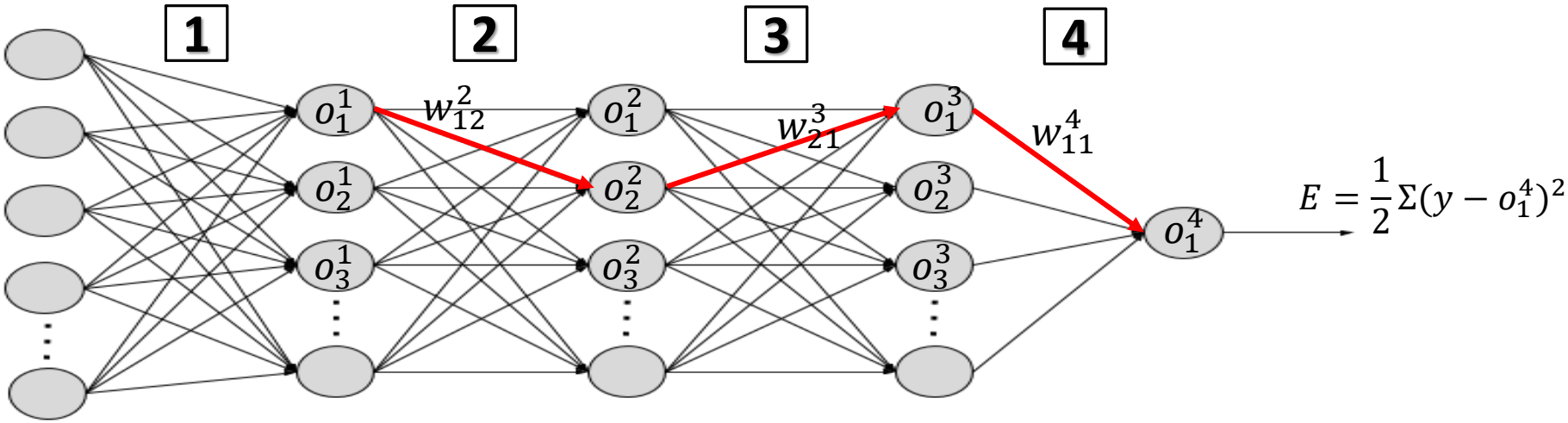
$$= -\boldsymbol{\Sigma(y - o_1^4)o_1^4(1 - o_1^4)}\frac{\partial z_1^4}{\partial o_1^3}\frac{\partial o_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial w_{21}^3}$$

$$= -\boldsymbol{\Sigma(y - o_1^4)o_1^4(1 - o_1^4)}w_{11}^4 o_1^3(1 - o_1^3)o_2^2$$

$$\frac{\partial z_1^4}{\partial w_{11}^4} = o_1^3 \quad \frac{\partial o_1^4}{\partial z_1^4} = o_1^4(1 - o_1^4) \quad \frac{\partial E}{\partial o_1^4} = -\Sigma(y - o_1^4)$$

$$\frac{\partial z_1^4}{\partial o_1^3} = w_{11}^4 \quad \frac{\partial o_1^3}{\partial z_1^3} = o_1^3(1 - o_1^3) \quad \frac{\partial z_1^3}{\partial w_{21}^3} = o_2^2$$

Multi-layer Neural Network



$$\frac{\partial E}{\partial w_{11}^4} = \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial w_{11}^4} = -\Sigma(y - o_1^4)o_1^4(1 - o_1^4)o_1^3$$

$$\frac{\partial E}{\partial w_{21}^3} = \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial o_1^3}\frac{\partial o_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial w_{21}^3} = -\Sigma(y - o_1^4)o_1^4(1 - o_1^4)w_{11}^4 o_1^3(1 - o_1^3)o_2^2$$

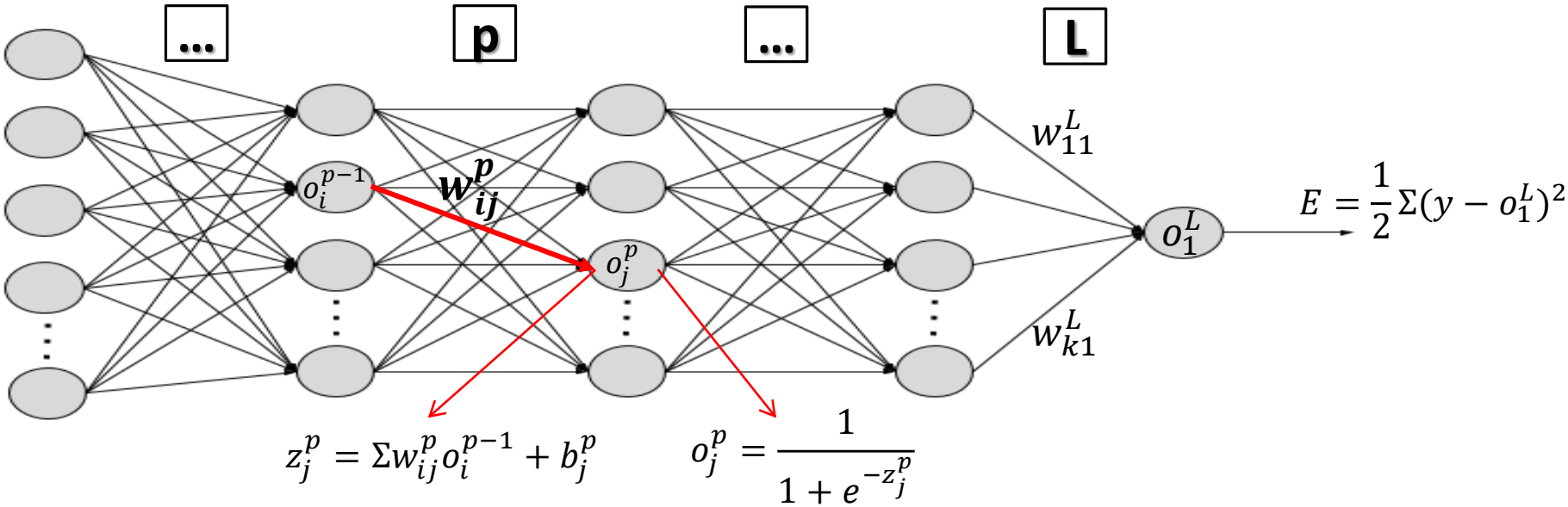
$$\frac{\partial E}{\partial w_{12}^2} = \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial o_1^3}\frac{\partial o_1^3}{\partial z_1^3}\frac{\partial z_1^3}{\partial o_2^2}\frac{\partial o_2^2}{\partial z_2^2}\frac{\partial z_2^2}{\partial w_{12}^2} + \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial o_2^3}\frac{\partial o_2^3}{\partial z_2^3}\frac{\partial z_2^3}{\partial o_2^2}\frac{\partial o_2^2}{\partial z_2^2}\frac{\partial z_2^2}{\partial w_{12}^2} + \cdots + \frac{\partial E}{\partial o_1^4}\frac{\partial o_1^4}{\partial z_1^4}\frac{\partial z_1^4}{\partial o_k^3}\frac{\partial o_k^3}{\partial z_k^3}\frac{\partial z_k^3}{\partial o_2^2}\frac{\partial o_2^2}{\partial z_2^2}\frac{\partial z_2^2}{\partial w_{12}^2}$$

$$= -\Sigma(y - o_1^4)o_1^4(1 - o_1^4)w_{11}^4 o_1^3(1 - o_1^3)w_{21}^3 o_2^2(1 - o_2^2)o_1^1$$

$$+ -\Sigma(y - o_1^4)o_1^4(1 - o_1^4)w_{21}^4 o_2^3(1 - o_2^3)w_{22}^3 o_2^2(1 - o_2^2)o_1^1$$

$$+ -\Sigma(y - o_1^4)o_1^4(1 - o_1^4)w_{k1}^4 o_k^3(1 - o_k^3)w_{2k}^3 o_2^2(1 - o_2^2)o_1^1$$

Multi-layer Neural Network



$$\frac{\partial E}{\partial w_{ij}^p} = \delta_j^p o_i^{p-1}$$

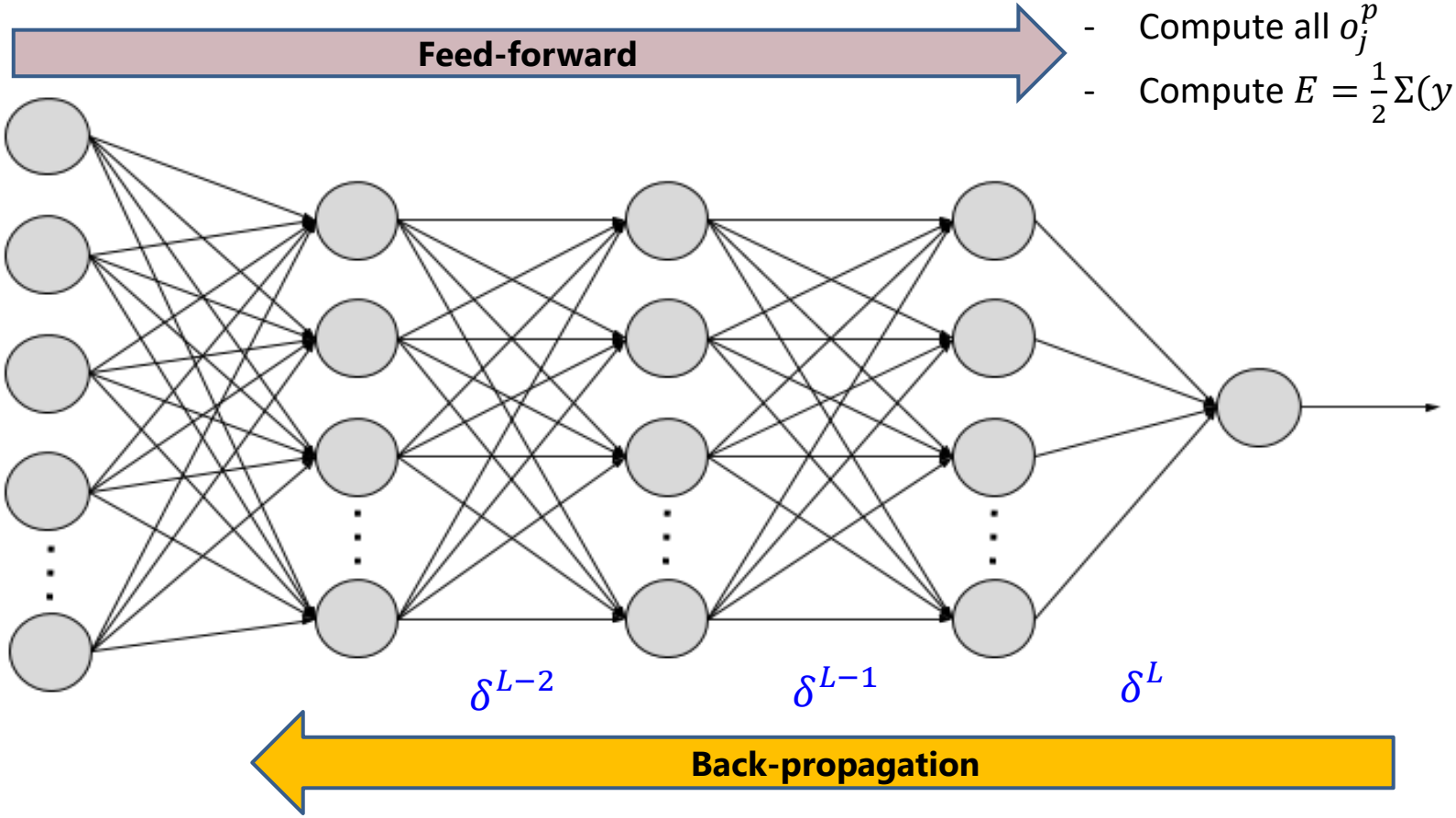
where $\begin{cases} \delta_j^p = o_j^p\left(1 - o_j^p\right)\Sigma w_{jk}^{p+1}\delta_k^{p+1} & (p \neq L) \\ \delta_1^L = o_1^L(1 - o_1^L)(-\Sigma(y - o_1^L)) & (p = L) \end{cases}$

$$\frac{\partial E}{\partial w_{11}^L} = \frac{\partial E}{\partial o_1^L}\frac{\partial o_1^L}{\partial z_1^L}\frac{\partial z_1^L}{\partial w_{11}^L}$$

$$= -\Sigma(y - o_1^L)o_1^L(1 - o_1^L)\, o_1^{L-1}$$

Back-propagation

- **Back-propagation: an algorithm to compute gradient for neural networks**



- The computation results from a layer can be utilized for computing gradient of its previous layer

Back-propagation

Algorithm

- For each training example, do:
  - **Feed-forward**: compute outputs for each neuron, o_j^p, compute $y - o_1^L$
  - **Back-propagation**: for each layer p (from **L to 1**), do:
    - Compute each $\frac{\partial E}{\partial w_{ij}^p} = \delta_j^p o_i^{p-1}$

$$\text{where } \begin{cases} \delta_j^p = o_j^p \left(1 - o_j^p\right) \Sigma w_{jk}^{p+1} \delta_k^{p+1} & (p \neq L) \\ \delta_1^L = o_1^L (1 - o_1^L)(-\Sigma(y - o_1^L)) & (p = L) \end{cases}$$

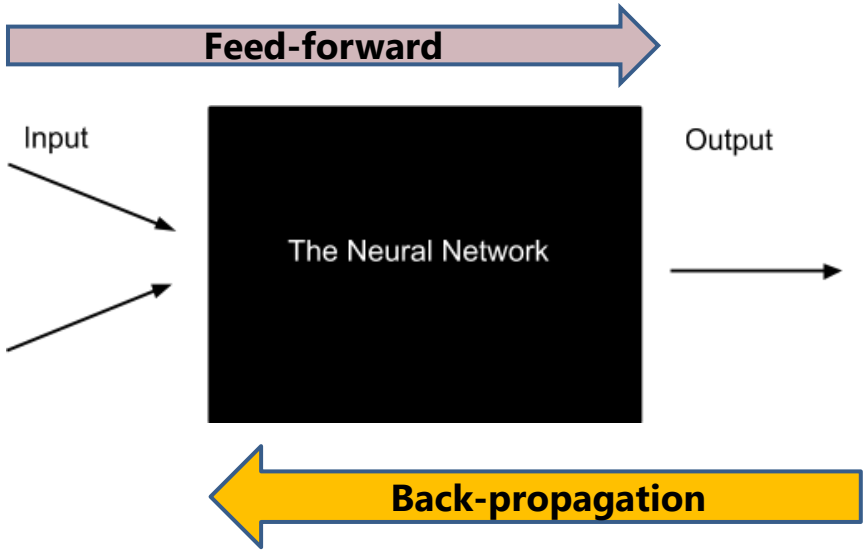
    - Store each δ_j^p for further usage
  - **Update**: Update all the weights (and biases) through $\mathbf{W}_{t+1} \leftarrow \mathbf{W}_t - \mu \Delta \mathbf{W}_t$

    where $\mathbf{W}_t$ is the vector of all the weights and $\Delta \mathbf{W}_t$ is the vector of gradient with respect to each weight $<\frac{\partial E}{\partial w_{11}^1}, \frac{\partial E}{\partial w_{12}^1}, \ldots, \frac{\partial E}{\partial w_{k1}^L}>$ at time point (epoch) t.

- Until converge

Back-propagation

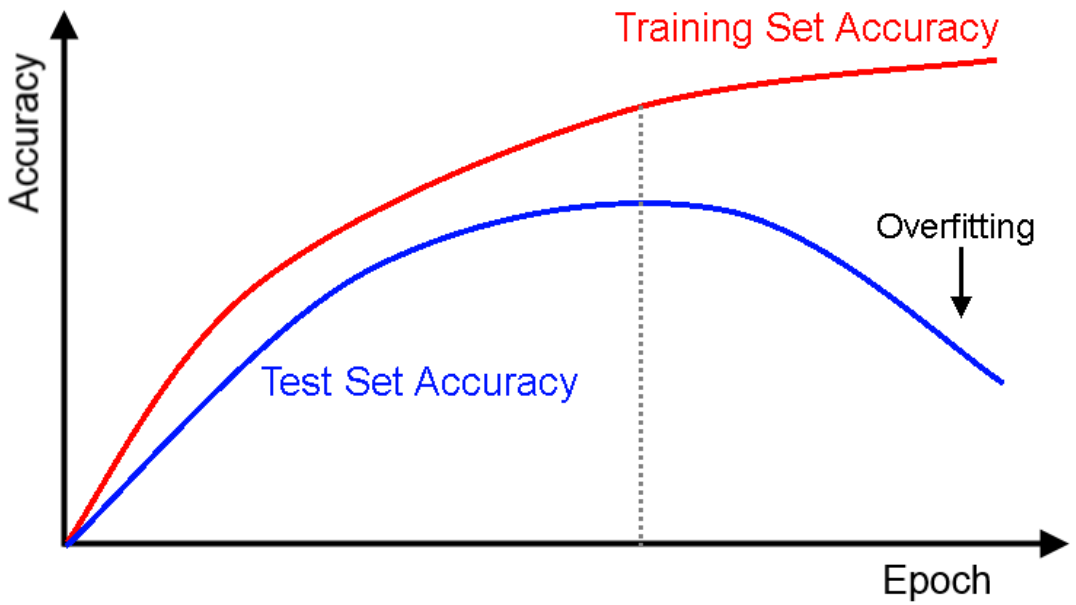
- **Various deep learning structures**
  - Fully-connected deep neural network
  - Autoencoder
  - Convolutional neural network (CNN)
  - Recurrent neural network (RNN)
  - Deep reinforcement learning
  - Generative Adversarial Networks (GAN)
  - Etc….



- **Back-propagation is universally used for training any deep learning model**

Overfitting

- **Perfect on training data, but poor on test data**
  - Bad generalization ability



- **Solutions to overfitting of neural networks**
  - L2 regularization
  - Dropout
  - Batch normalization

Regularization

A.K.A
L2-norm

L2 regularization

- To add the sum of the squares of the model parameters, $\boldsymbol{\Sigma w_i^2}$, multiplied with a regularization coefficient λ, in the cost function.

$$E(\mathbf{W}) = \frac{1}{2}\sum_{k=1}^{n}(y - \hat{y}_k)^2 + \frac{\boldsymbol{\lambda}}{\boldsymbol{2}}\boldsymbol{\Sigma w_i^2}$$

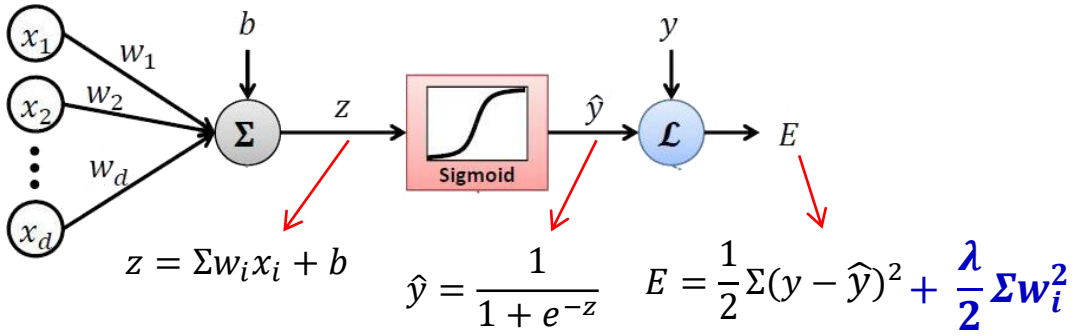
- Now we have to find optimal W* that minimizes the cost as well as the L2-norm
- If the value of each w_i is small, then the decision boundary becomes smoother, thus reducing the overfitting.
- λ controls the relative importance of **reducing the cost** VS **making model simple & smooth**

Regularization

- **Cost function** $E(\mathbf{W}) = \frac{1}{2}\sum_{k=1}^{n}(y - \hat{y}_k)^2 + \frac{\boldsymbol{\lambda}}{2}\boldsymbol{\Sigma w_i^2}$

  where $\hat{y} = \frac{1}{1+e^{-z}}$ **and** $z = \sum_{i=1}^{d} x_i w_i + b$

- **Goal: W*** = $\underset{\mathbf{W}}{\operatorname{argmin}}\, E(\mathbf{W})$



$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_i} + \boldsymbol{\lambda w_i} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_i} + \boldsymbol{\lambda w_i}$$

$$= -\Sigma x_i \hat{y}(1-\hat{y})(y-\hat{y}) + \boldsymbol{\lambda w_i}$$

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial b} = -\Sigma \hat{y}(1-\hat{y})(y-\hat{y})$$

$$\frac{\partial z}{\partial w_i} = x_i \qquad \frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y}) \qquad \frac{\partial E}{\partial \hat{y}_k} = -\Sigma(y-\hat{y})$$

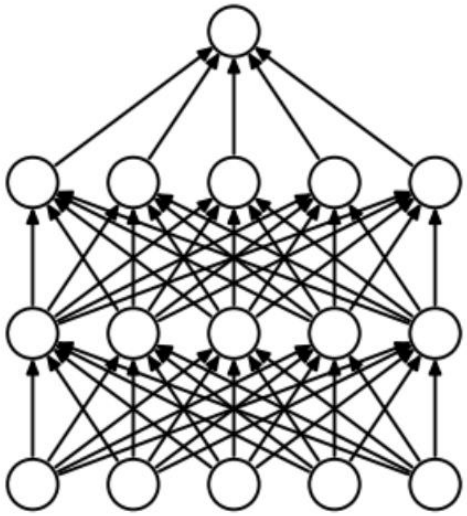
$$\boldsymbol{\frac{\partial E}{\partial w_i} = \lambda w_i}$$

$$\frac{\partial z}{\partial b} = 1$$

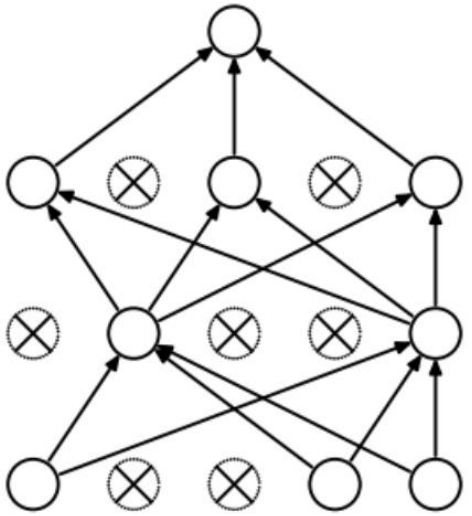
Regularization

Dropout

- https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf



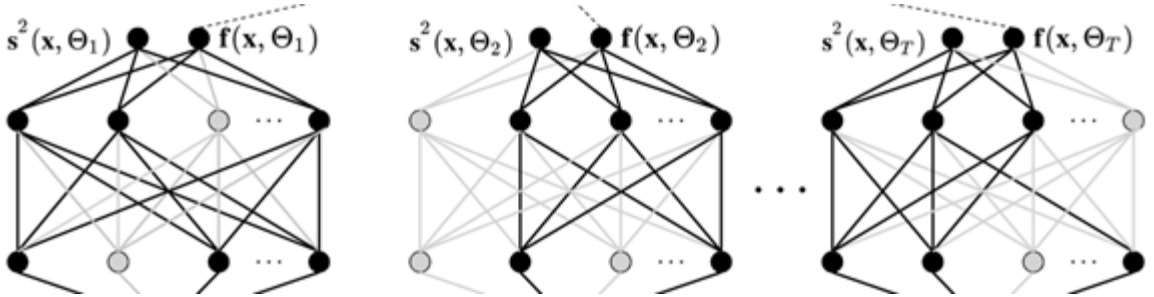
(a) Standard Neural Net



(b) After applying dropout.

- Performs a kind of "geometric averaging" over the **ensemble** of possible subnetworks

- Randomly "drops" neurons in every training step



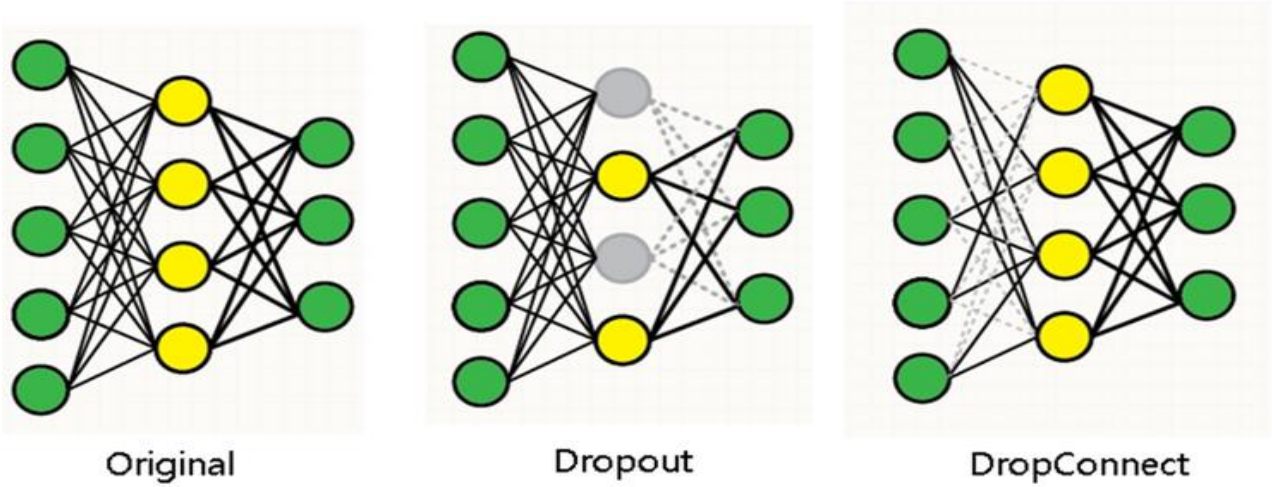
Regularization

Dropout

- Dropout rate p: the probability that each node in a layer is dropped, where 1.0 means no dropout, and 0.0 means no outputs from the layer.
- Empirically, a good value for dropout rate in a hidden layer is between 0.5 and 0.8
- We don't apply dropout on the "testing" phase; it is only used in the training phase

Dropconnect

- http://proceedings.mlr.press/v28/wan13.pdf



Regularization

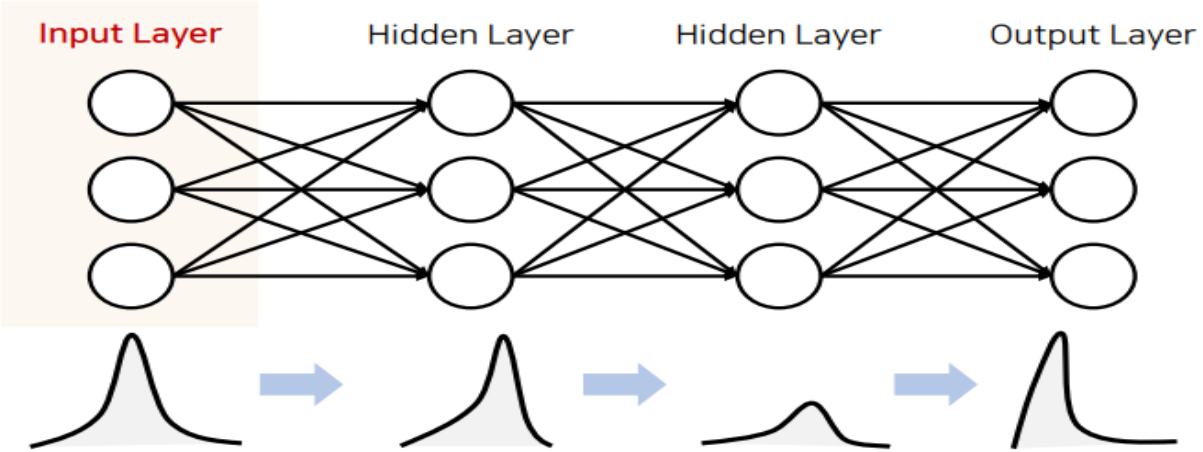
Batch normalization

- http://proceedings.mlr.press/v37/ioffe15.pdf
- Normalization: the creation of shifted and scaled versions of data. It adjusts values measured on different scales to a notionally common scale

$$z = \frac{x_i - \mu}{\sigma}$$

	# of words	# of attached files	# of links	# of malicious words	.....	spam
mail #1	256	0	3	7	.....	1 (Yes)
mail #2	56	1	0	3	.....	0 (No)
mail #3	24	1	0	1	.....	0
mail #4	672	0	0	0	.....	0

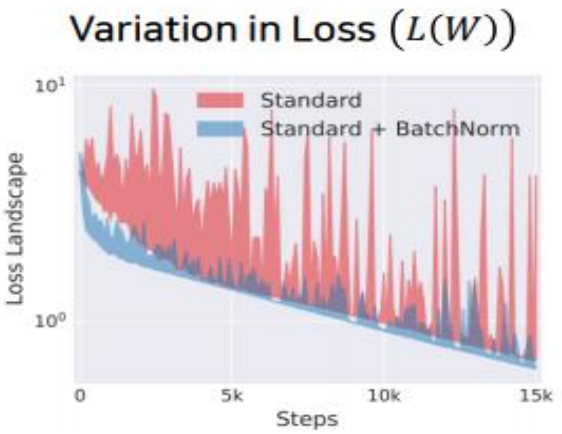
- Varying data distributions across batches & across layers



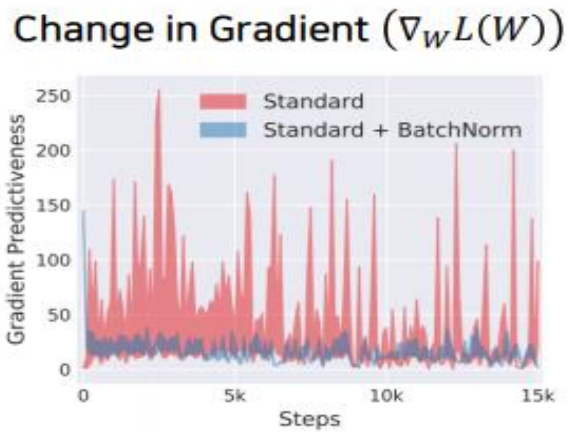
Regularization

Batch normalization

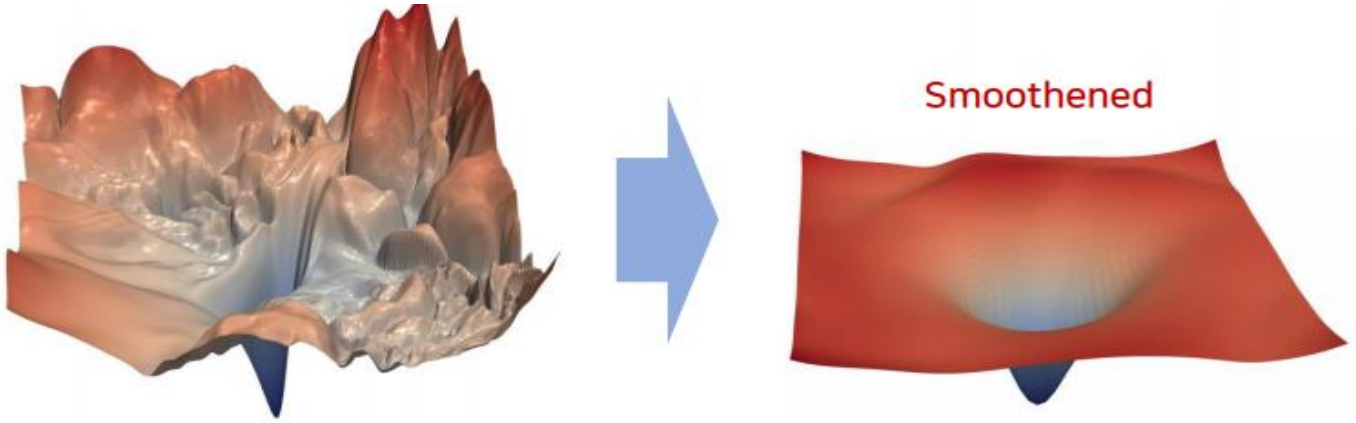
- Goal: to remedy the varying data distributions across batches and ensure the output statistics are fixed
- Why?



(a) Loss Landscape



(b) Gradient Predictiveness



Regularization

Batch normalization

- How?

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

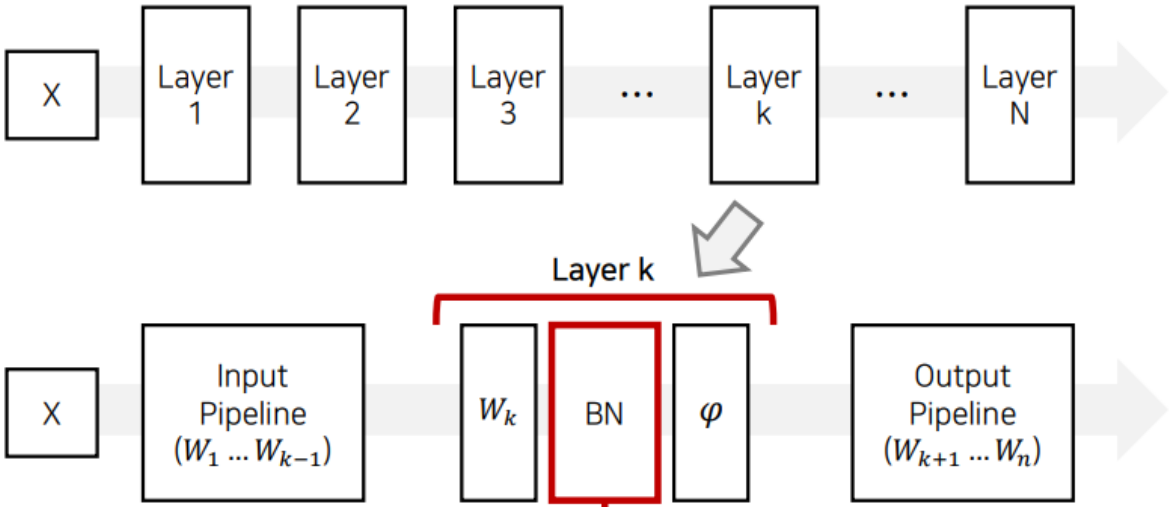
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

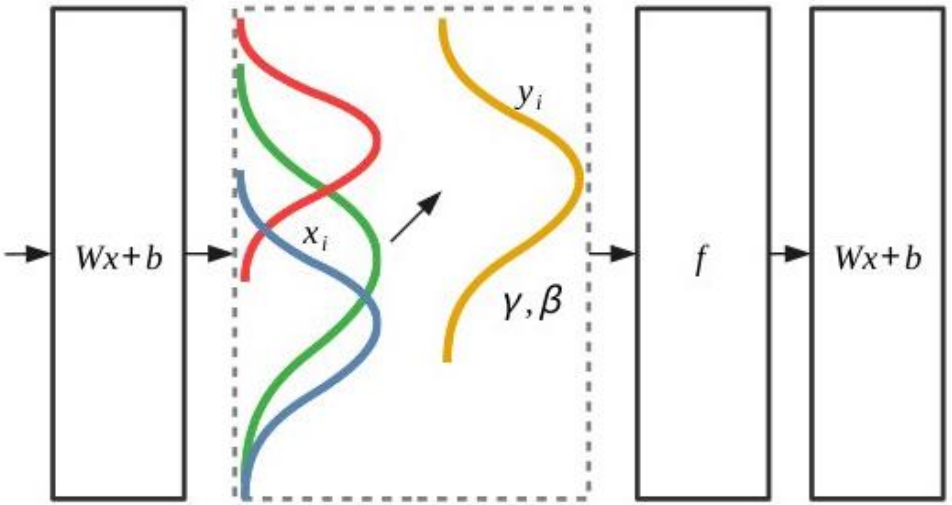
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

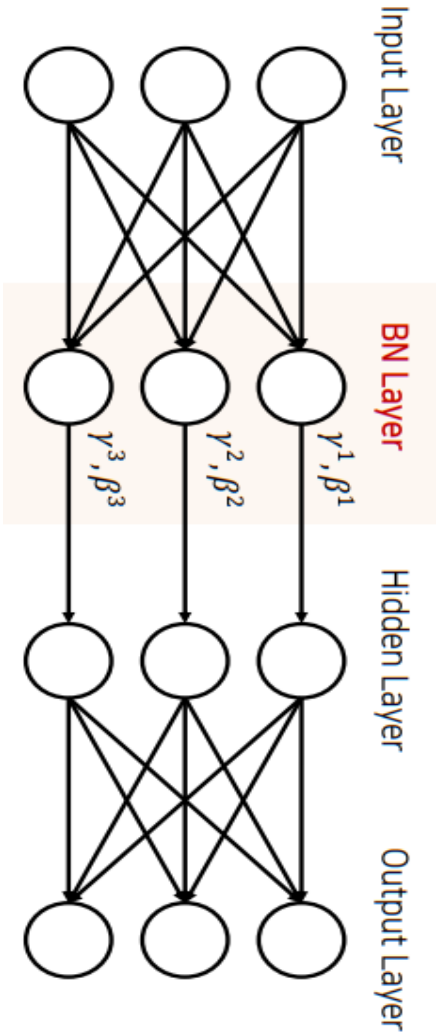
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Regularization

☐ Batch normalization

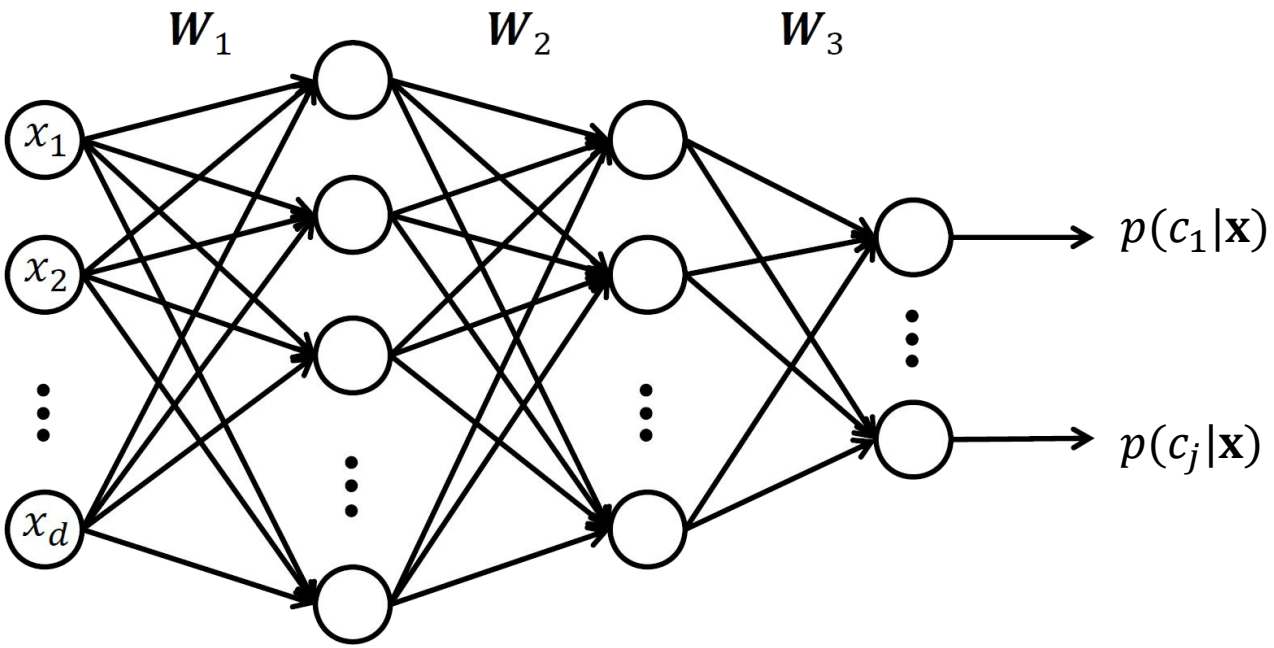


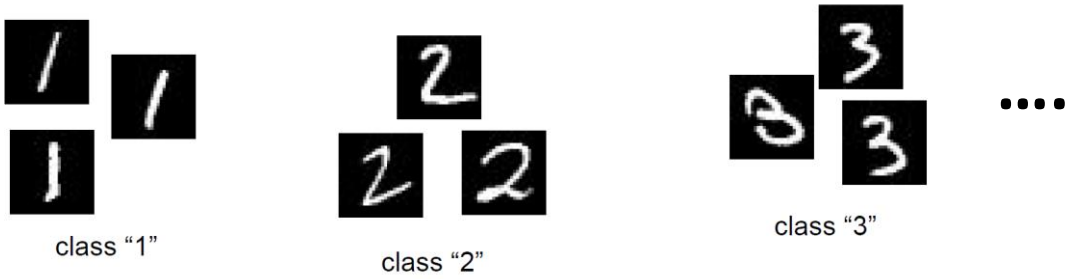




Multi-Class Classification

- We want the network to classify the input into a multiple number of classes (c_1, c_2, … , c_j)

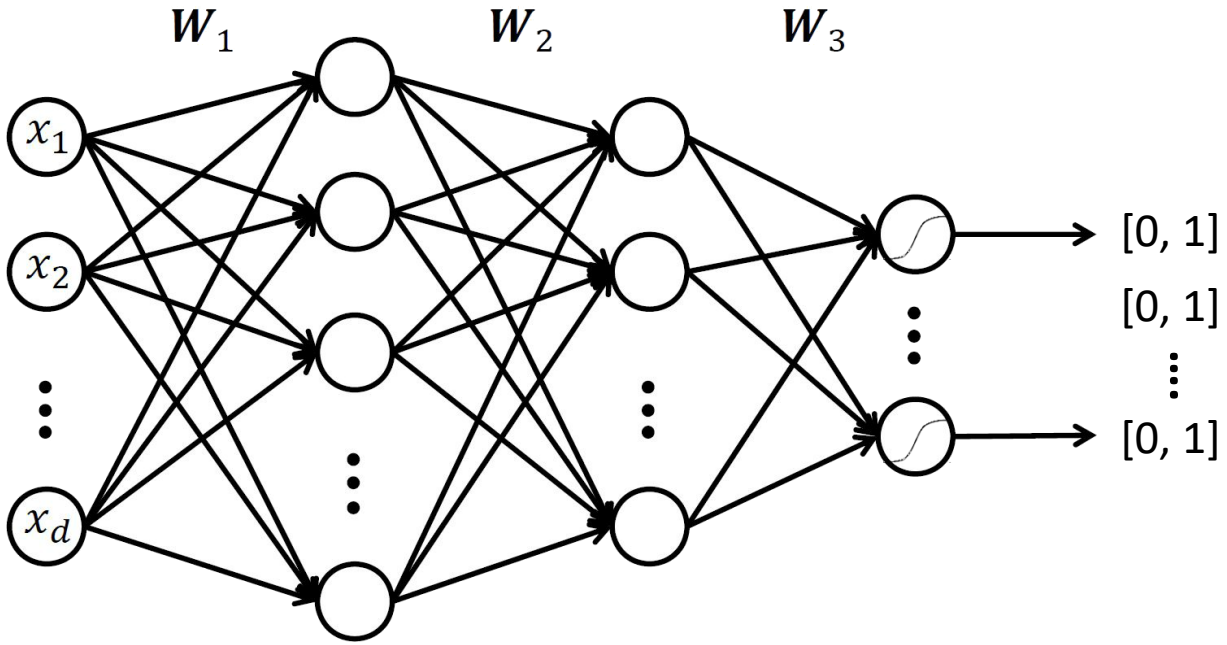




Multi-Class Classification

Sigmoid function

- If the number of classes is bigger than two, each output neuron with sigmoid activation **cannot imply the probability** of a particular class



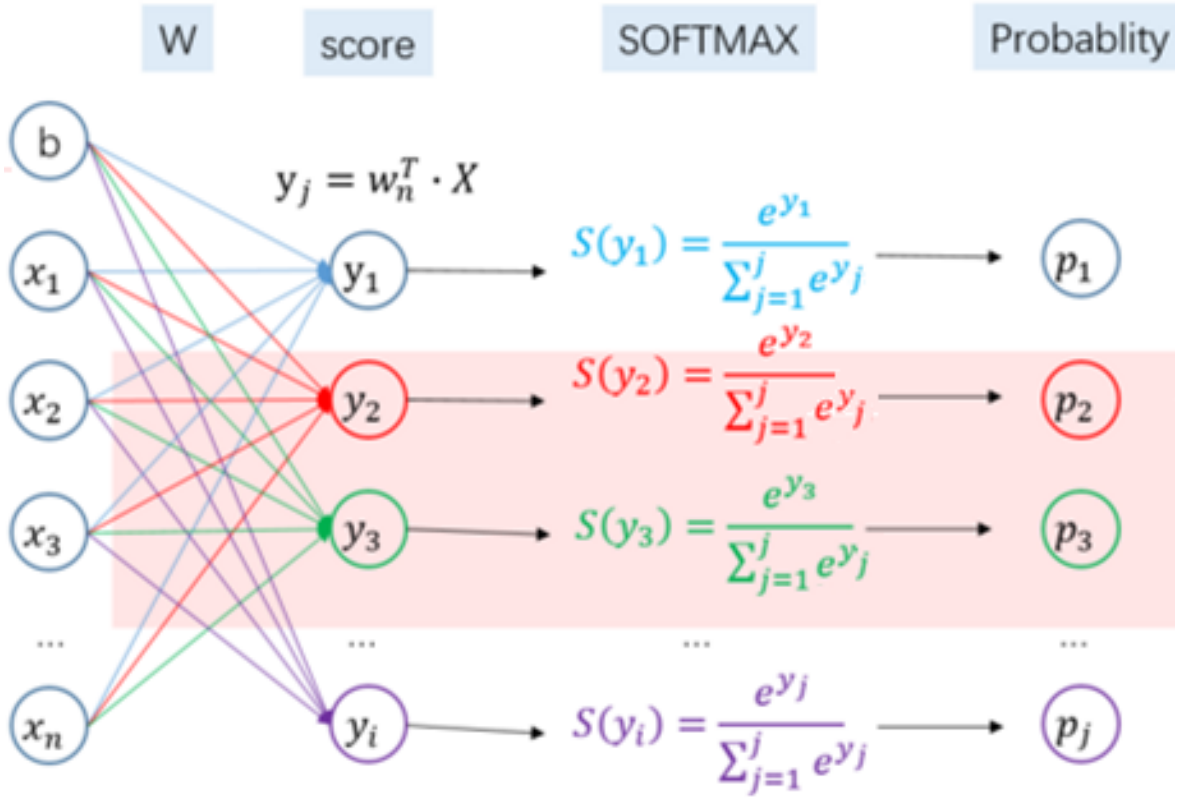
What function could be an alternative that makes each output imply the probability of a particular class?

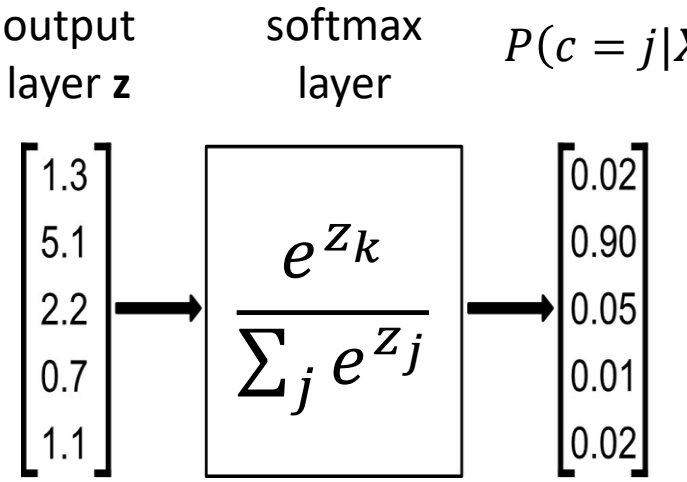
Classification with Neural Networks

Softmax function

- Let z_k be an output of k^{th} neuron in the output layer (without any activation)
- The softmax function converts z_k into a probability by:

$$P(c = k|\boldsymbol{X}) = \frac{e^{z_k}}{\sum_j e^{z_j}} \quad \text{where} \quad z_k = \sum_{i=1}^{n} x_i w_{ik} + \text{b}$$





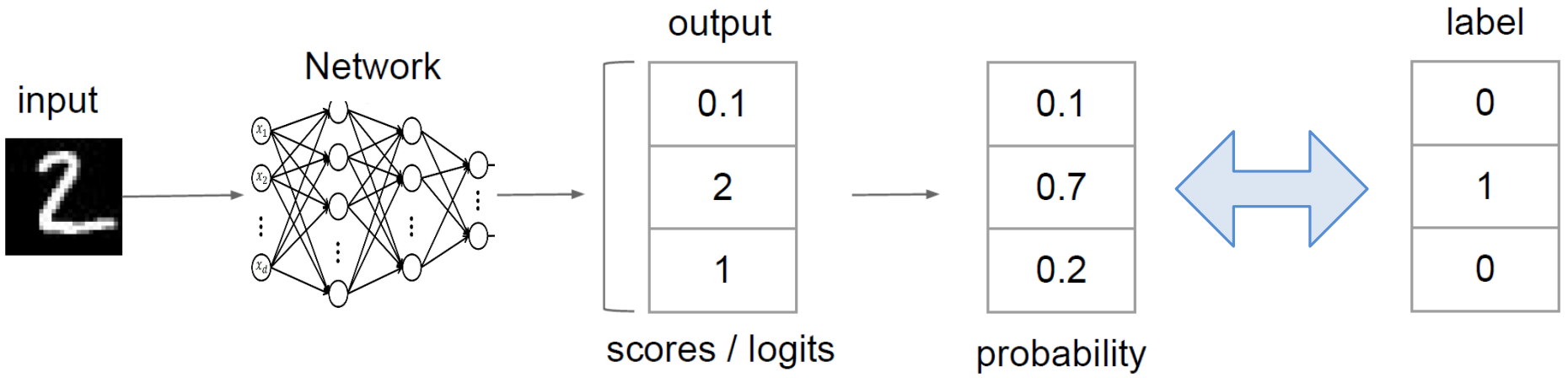
Multi-Class Classification

One-hot encoding for ground truth label

- Transforming each label into a one-hot vector (with only 1 and 0)
- Length equal to the total number of classes "j"
- Value of 1 for the correct class and 0 elsewhere

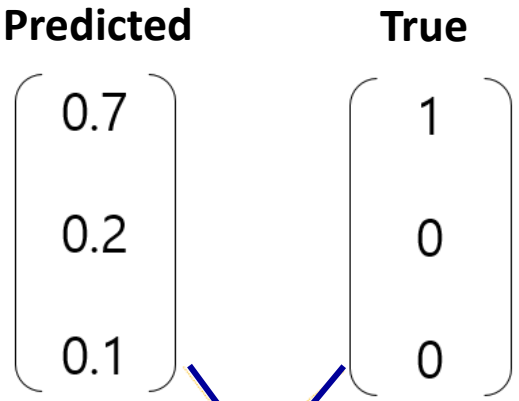
class "1"	class "2"	class "3"
1	0	0
0	1	0
0	0	1

Comparing predicted probabilities and true probabilities and computing loss



Multi-Class Classification

Cross-entropy loss function



$$E(\mathbf{W}) = -\sum_j y_j \log \hat{y}_j \quad \text{where} \quad \hat{y}_j = \frac{e^{z_k}}{\sum_j e^{z_j}} \quad \text{and} \quad z_k = \sum_{i=1}^{n} x_i w_{ik} + \mathrm{b}$$

True probability — y_j

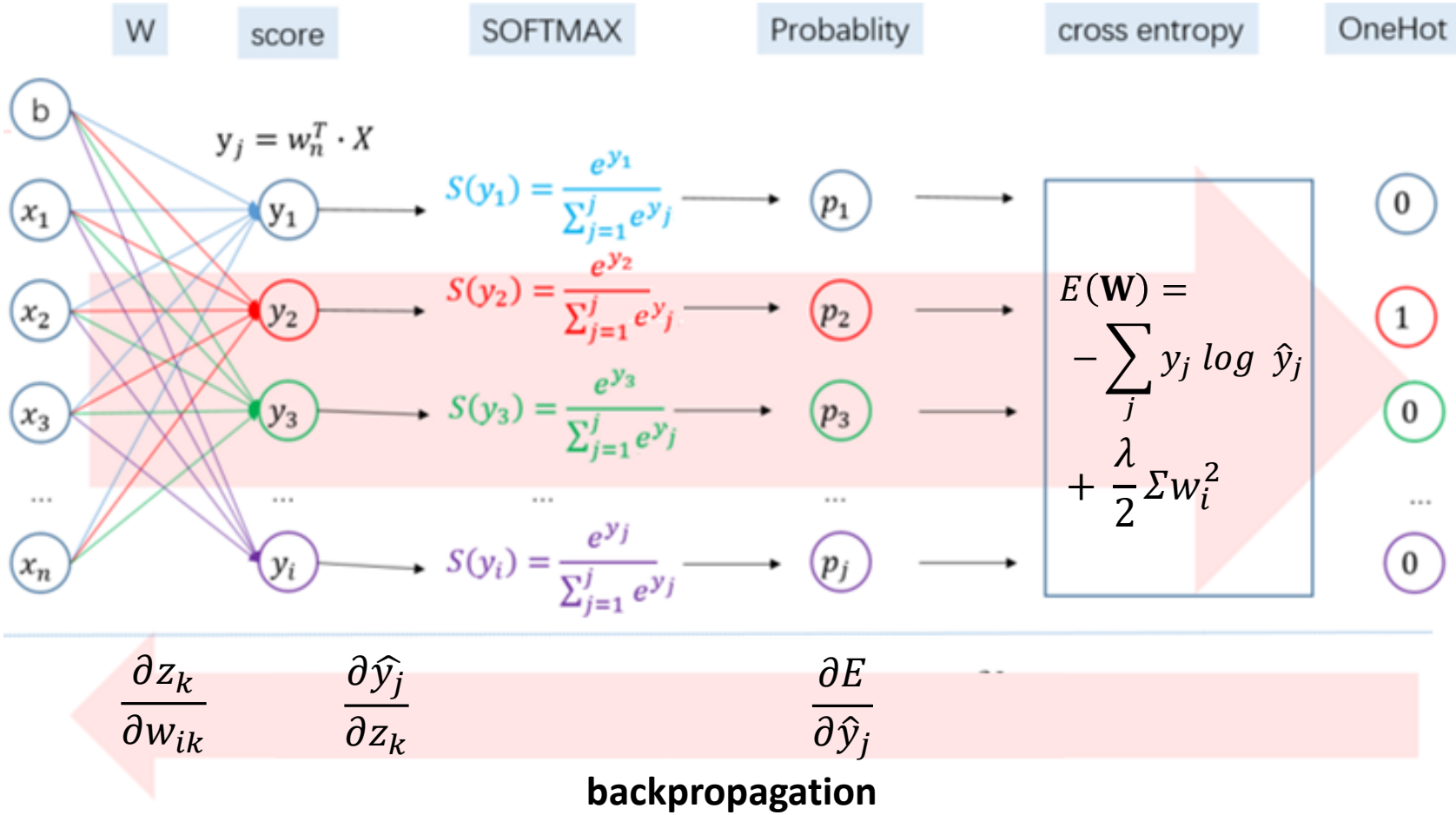
Predicted probability — $\hat{y}_j$

$$= -(1 * \log 0.7 \; + \; 0 * \log 0.2 \; + \; 0 * \log 0.1)$$

Classification with Neural Networks

- **Cost function** $E(\mathbf{W}) = -\sum_T \sum_j y_j \log \hat{y}_j + \frac{\lambda}{2}\Sigma w_i^2$

  where $\hat{y}_j = \frac{e^{z_k}}{\sum_j e^{z_j}}$ **and** $z = \sum_{i=1}^{n} x_i w_{ik} + \mathrm{b}$



ReLU

Vanishing gradient problem

- https://en.wikipedia.org/wiki/Vanishing_gradient_problem
- When training neural networks, the gradient can become **vanishingly small**, which prevents the weight from changing its value.
- It leads to **slow convergence**, and in the worst case, it may even **stop** the neural network from further **training**.

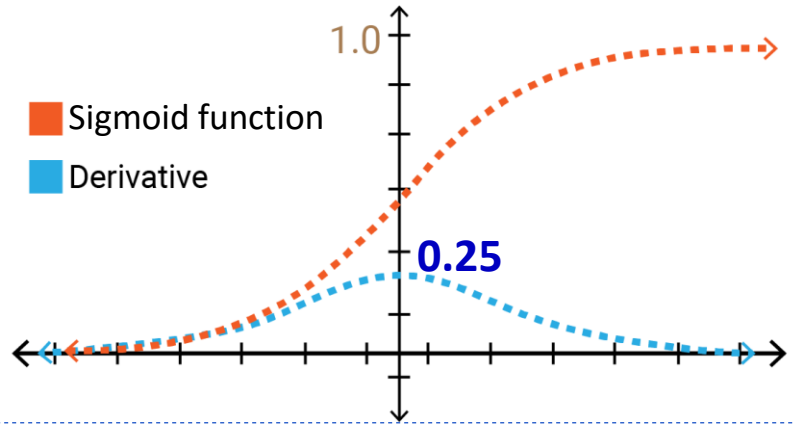
Main reasons

- The maximum value of the derivative of the sigmoid function is 0.25.
- Computing gradient involves too much times of multiplications with small numbers (0~1)
  - 0.25 * 0.75 * 0.25 * 0.8 * 0.25 = **0.009**

Partial derivative of the second layer

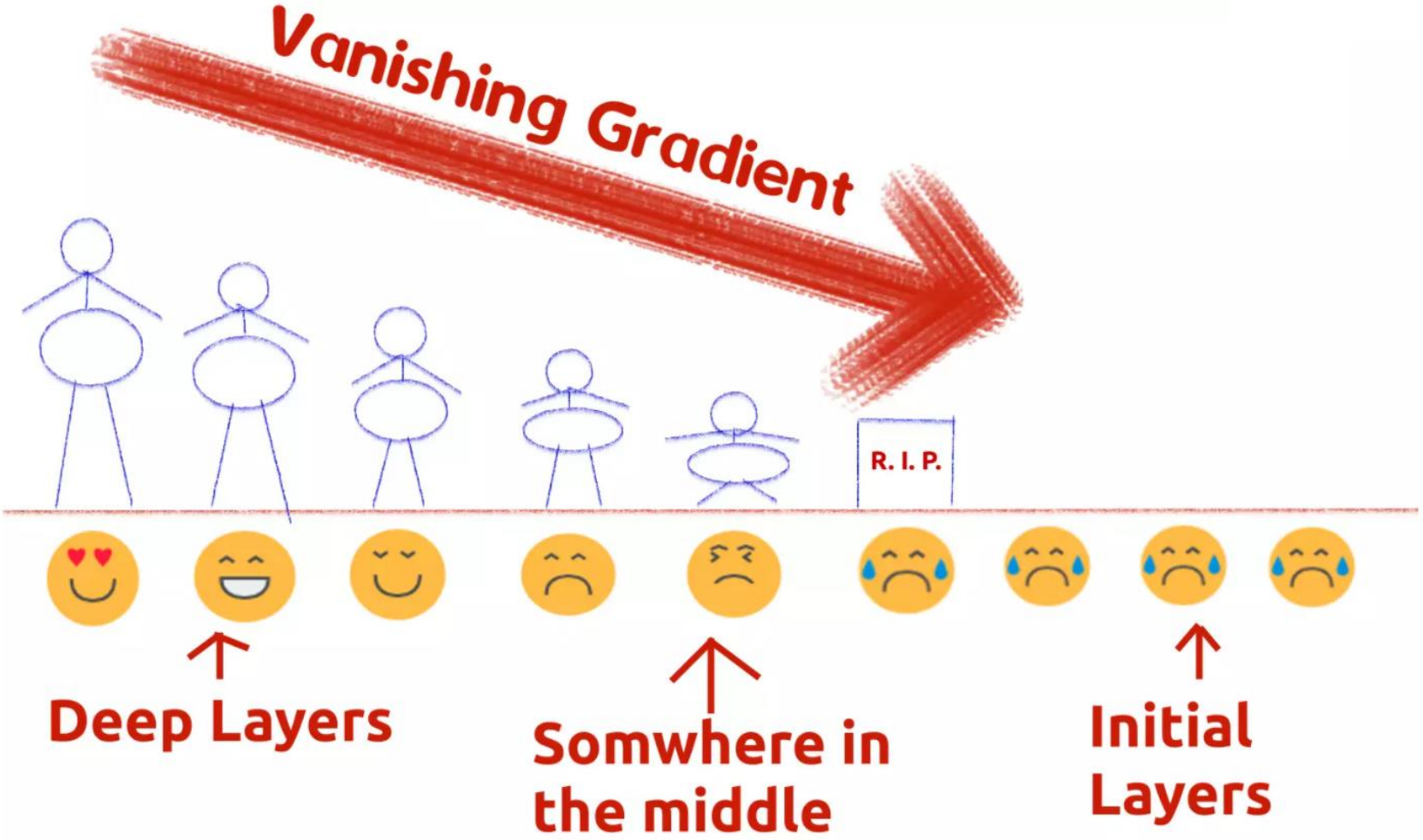
$$\frac{\partial E}{\partial w_{ki}^{1}} = \frac{\partial E}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial z_j}\frac{\partial z_j}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ki}^{1}}$$

$$= -\Sigma \hat{y}_j(1-\hat{y}_j)(y-\hat{y}_j)w_{ij}^{2}\hat{y}_i(1-\hat{y}_i)x_k$$



ReLU

- Vanishing gradient problem



Vanishing Gradient Intuition

ReLU

Rectified Linear units (ReLU)

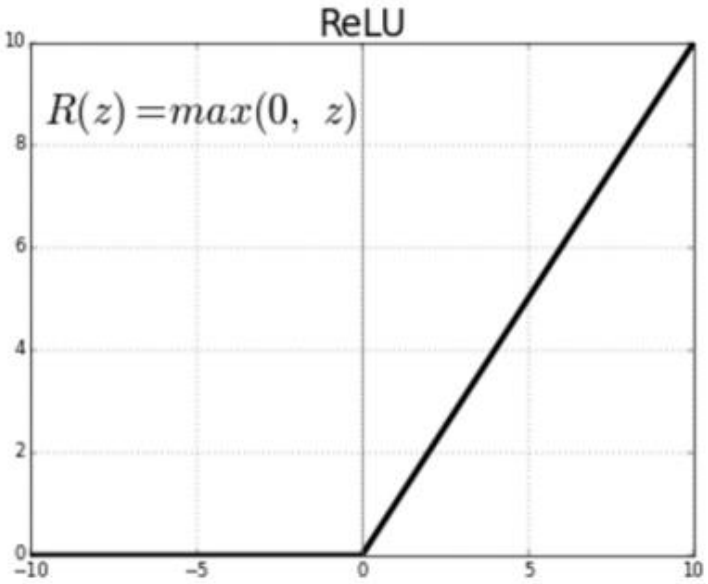
- Activation function

  - $\boldsymbol{y} = \max(\mathbf{0}, \boldsymbol{z})$

  where $z = \sum_{i=1}^{n} x_i w_{ik} + \mathrm{b}$

Derivative

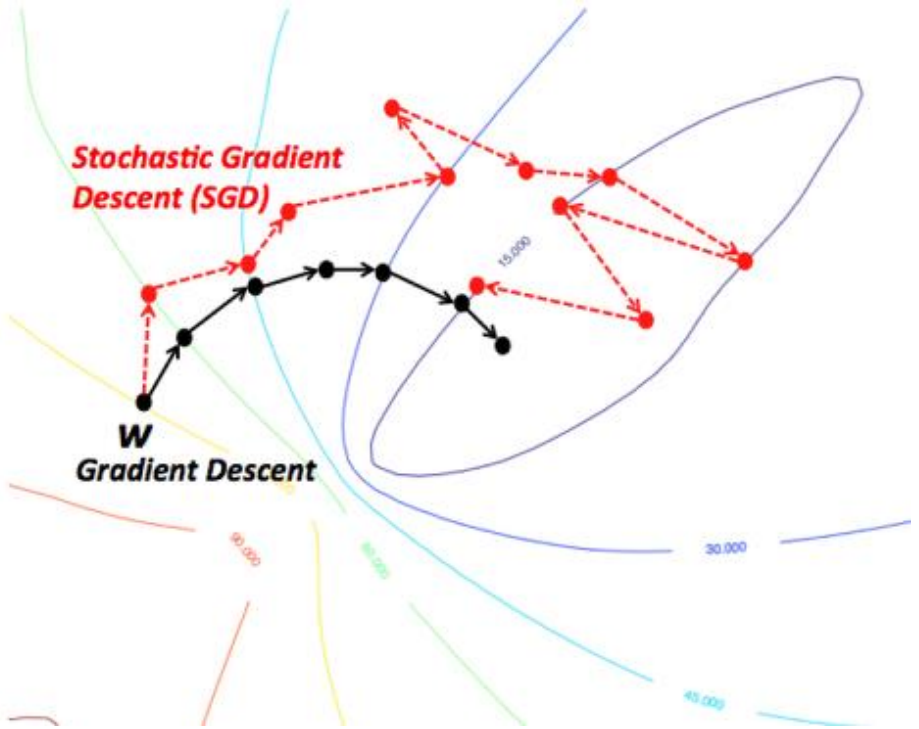
$$\frac{\partial y}{\partial z} = \begin{cases} \mathbf{0} & \boldsymbol{z} \leq \mathbf{0} \\ \mathbf{1} & \textbf{otherwise} \end{cases}$$



- Fast to compute activation and derivatives
- No vanishing of back-propagated values as long as unit is activated

Optimization

- **(deterministic) Gradient descent**
  - $E(\mathbf{W}) = -\sum_{T} \sum_j y_j \, log \;\; \hat{y}_j + \frac{\lambda}{2} \Sigma w_i^2$
  - In each update, it computes the cost function by using all data instances (size ***T***)
  - But, using the complete training set in each iteration can be very expensive and time consuming
- **Stochastic gradient descent**
  - $E(\mathbf{W}) = -\sum_j y_j \, log \;\; \hat{y}_j + \frac{\lambda}{2} \Sigma w_i^2$
  - Randomly sample one data instance, compute cost, and update model
  - Looks fast, but unstable



Optimization

Stochastic gradient descent with Minibatch

- $E(\mathbf{W}) = -\sum_{M} \sum_j y_j \, log \; \hat{y}_j \; + \frac{\lambda}{2} \Sigma w_i^2$
- Randomly sample **M** numbers of (T >> M) data instances
  - We call this mini-batch
- It achieves the advantages of both the deterministic and stochastic gradient descent methods



Working with Neural Networks

- **When you design your own model,**
  - If you use **softmax layer** with **cross-entropy loss** for multi-class classification,
  - Employ at least one of the **{L2-norm, dropout, batch normalization}** for preventing the overfitting problem,
  - **ReLU** as an activation function for preventing the vanishing gradient problem,
  - And finally **SGD with mini-batch**,
  - You can avoid a lot of problems that you would encounter when training deep learning models

Class Summary

- **Today's keywords**
  - A perceptron
  - Multi-layer Neural Network
  - Back-propagation
  - More about neural networks, including…
    - Multi-class classification with neural networks and Softmax output
    - Gradient vanishing and ReLU
    - Overfitting and regularization skills
    - Dropout and Batch-normalization
    - Gradient descent VS stochastic gradient descent
    - …..

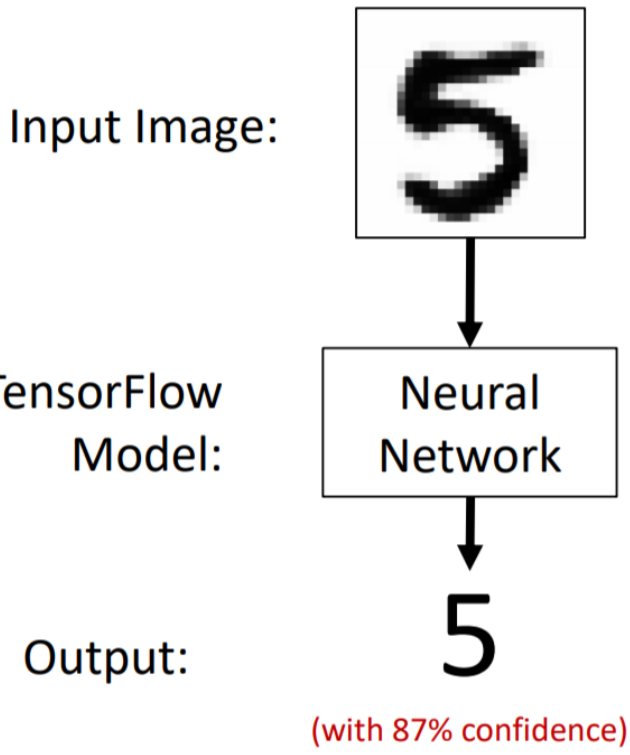
Class Summary

- **What programmers do**
  - Preparing data
  - Setting up development environments
  - Designing network structure
    - # of layers, # of nodes, activation function, step size, ….
  - Defining cost function
  - Defining training condition
    - Learning rate, termination condition, …

- **What Tensorflow (and friends) do**
  - Low-level computations
    - Computing partial derivatives and gradient
    - Updating edge weights and biases
    - Using GPUs for parallel computing
    - ….

Class Summary

What programmers do



1
```
# import tensorflow and keras (tf.keras not "vanilla" Keras)
import tensorflow as tf
from tensorflow import keras
```

2
```
# get data
(train_images, train_labels), (test_images, test_labels) = \
keras.datasets.mnist.load_data()
```

3
```
# setup model
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation=tf.nn.relu),
    keras.layers.Dense(10, activation=tf.nn.softmax)
])

model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

4
```
# train model
model.fit(train_images, train_labels, epochs=5)
```

5
```
# evaluate
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('test accuracy:', test_acc)
```

6
```
# make predictions
predictions = model.predict(test_images)
```

Class Summary

Dropout implementation

- Python+tensorflow+keras

```
model = tf.keras.Sequential()
model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                    input_shape=[28, 28, 1]))
model.add(layers.LeakyReLU())
model.add(layers.Dropout(0.3))

model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
model.add(layers.LeakyReLU())
model.add(layers.Dropout(0.3))

model.add(layers.Flatten())
model.add(layers.Dense(1))

return model
```

Class Summary

☐ Batch normalization

```
with tf.name_scope('dnn'):
    # batch normalization layer using partial
    batch_norm_layer = partial(
            tf.layers.batch_normalization,
            training=training,
            momentum=batch_norm_momentum)

    # 1st - hidden
    hidden1 = tf.layers.dense(inputs, n_hidden1, name="hidden1")
    # batch norm
    bn1 = batch_norm_layer(hidden1)
    # activation function
    bn1_act = tf.nn.elu(bn1)

    # 2nd - hidden
    hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
    bn2 = batch_norm_layer(hidden2)
    bn2_act = tf.nn.elu(bn2)

    # outputs
    logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
    logits = batch_norm_layer(logits_before_bn)
```

Thank You !

