AI - FOUNDATION AND APPLICATION

Instructor:
Assoc. Prof. Dr. Truong Ngoc Son

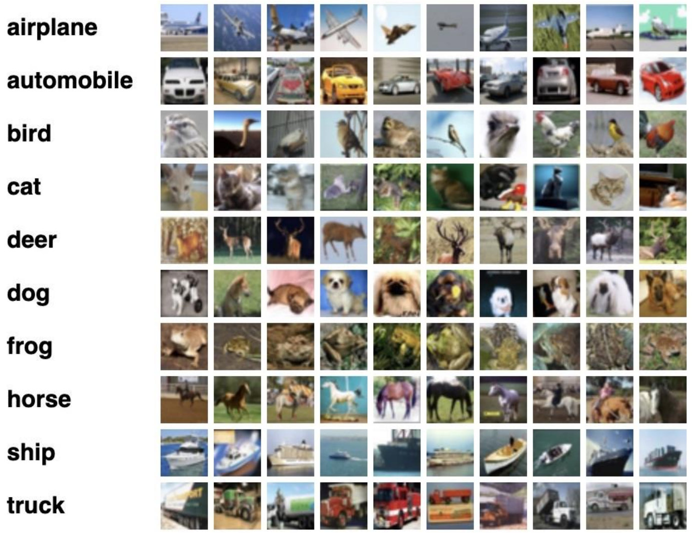
Chapter 2
Back Propagation

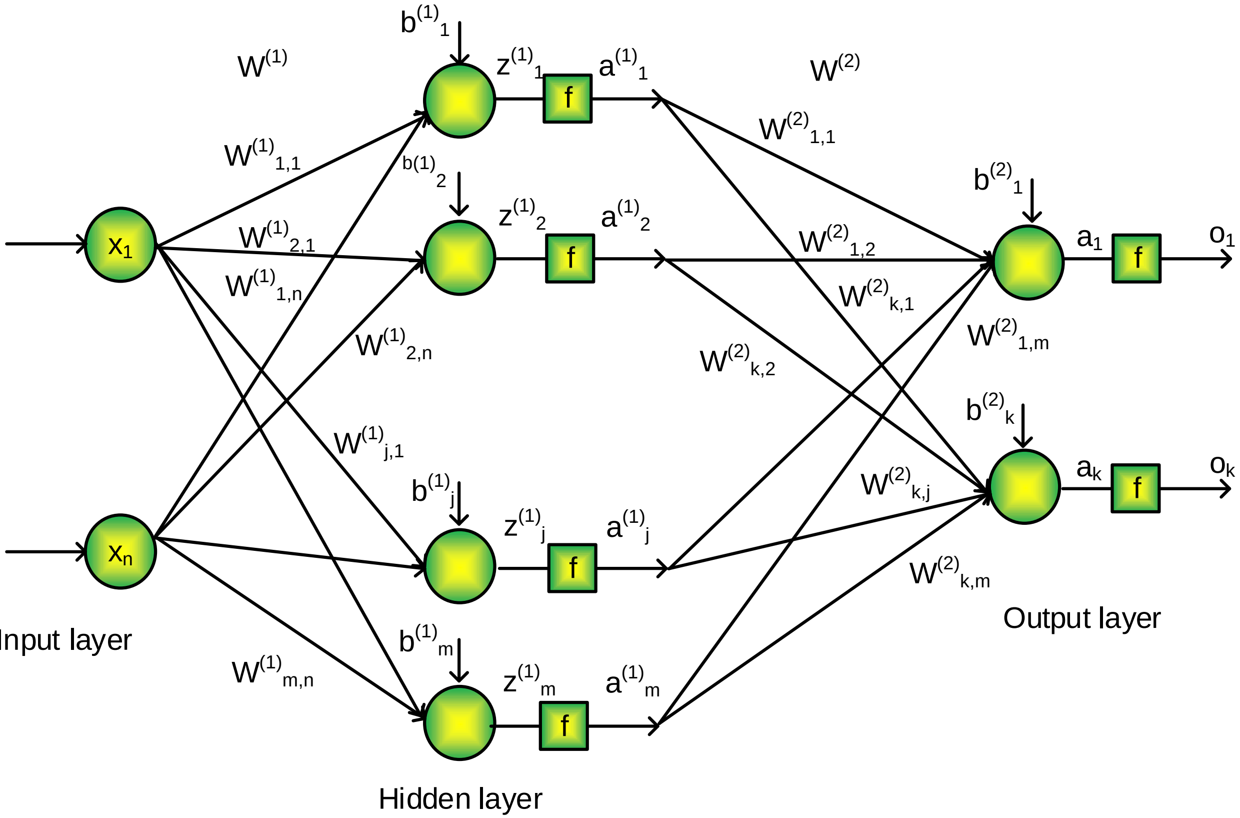
Outline

- ❑ Multiclass Classification With Softmax regression
- ❑ Lost function – cross entropy
- ❑ Stochastic gradient descent – batch and mini-batch gradient descent
- ❑ Translating math into code

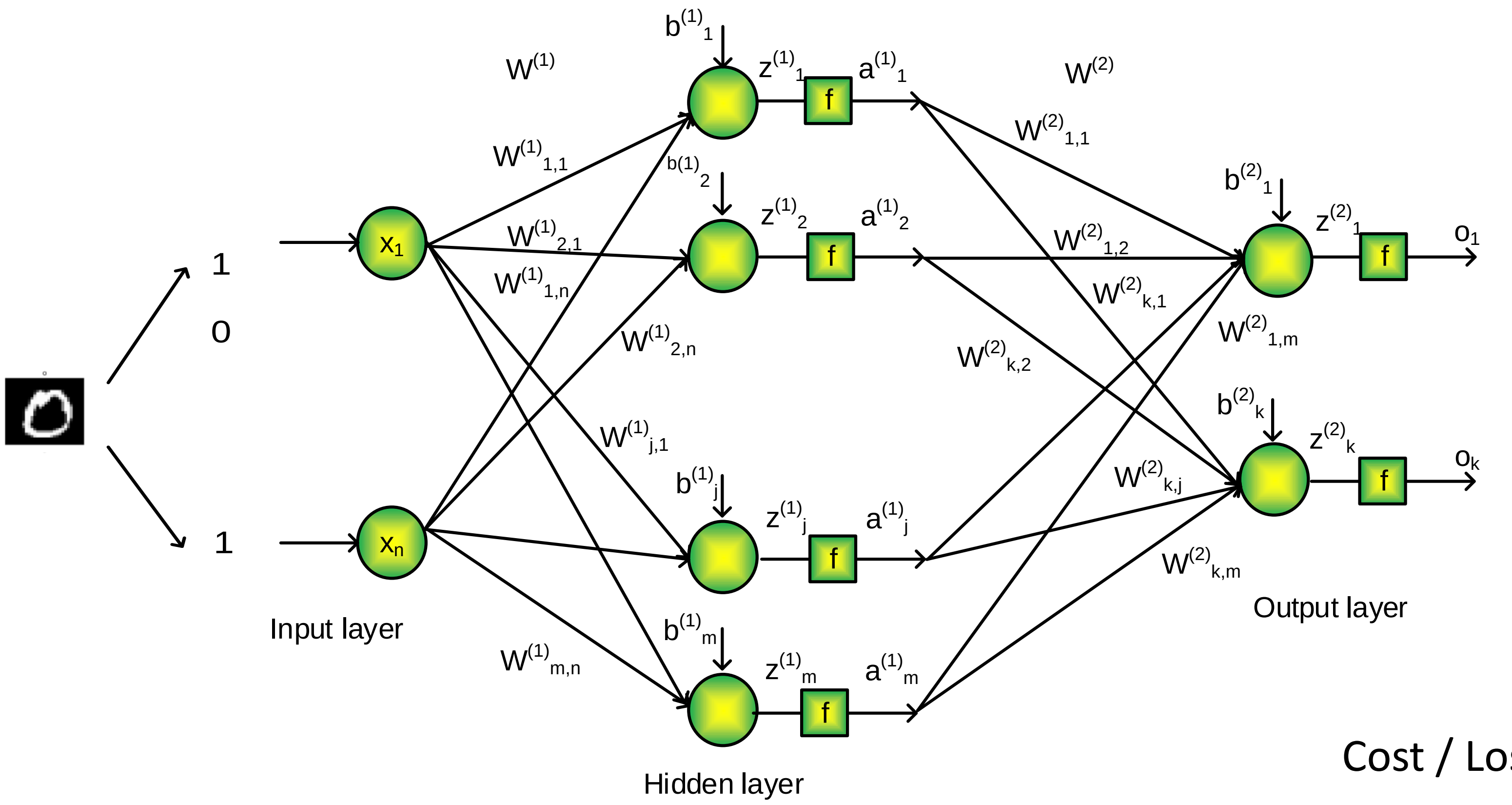
Multiclass Classification With Softmax regression

Multiclass classification example





Multiclass classification example



Predictive output,o	***Desired output, y***
0.9	1
0.7	0
0.5	0

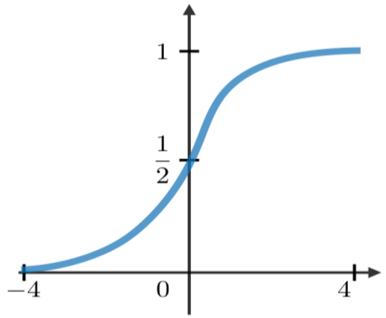
N samples, K outputs

Cost / Loss $$L = \frac{1}{\mathrm{N}}\sum_{t=1}^{\mathrm{N}}\sum_{k=1}^{\mathrm{K}}(y_k^t - o_k^t)^2$$

Multiclass classification with logistic regression

Sigmoid function is used for the neurons at the output layer

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



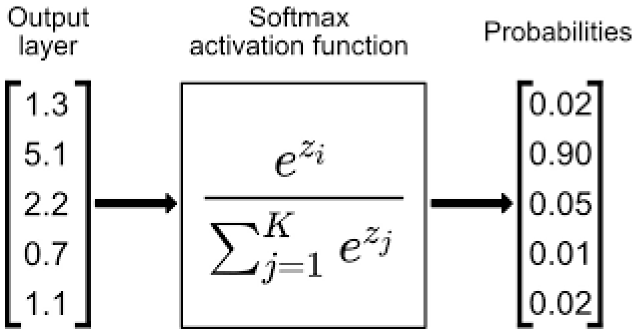
- It is ideally for two-class classification
- The outputs are independent
- The sigmoid may produces high probability for all classes, some of them, or none of them

0.9	0.9	0.3
0.8	0.2	0.2
0.6	0.5	0.1

Multiclass classification with softmax regression

We expect that there is only one right answer, the outputs are mutually exclusive.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

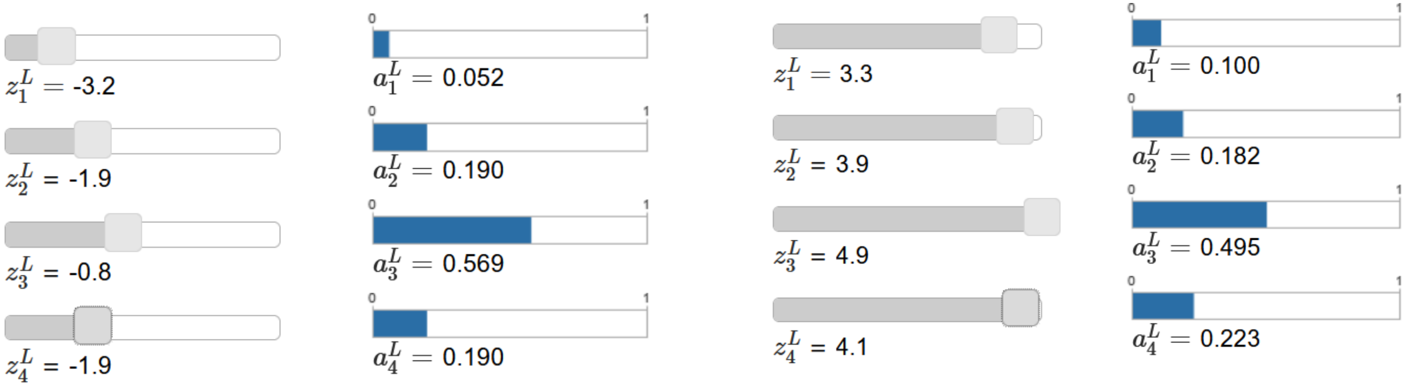


- The softmax will enforce that the sum of the probabilities of output classes are equal to one
- Softmax is used for multi-classification in the Logistic Regression model, whereas Sigmoid is used for binary classification in the Logistic Regression model

Multiclass classification with softmax regression

- ❑ Softmax function is mostly used in a final layer of Neural Network
- ❑ The outputs are probability distribution

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$



Lost function – cross entropy

Cross-entropy takes the negative log likelihood of the predicted probability

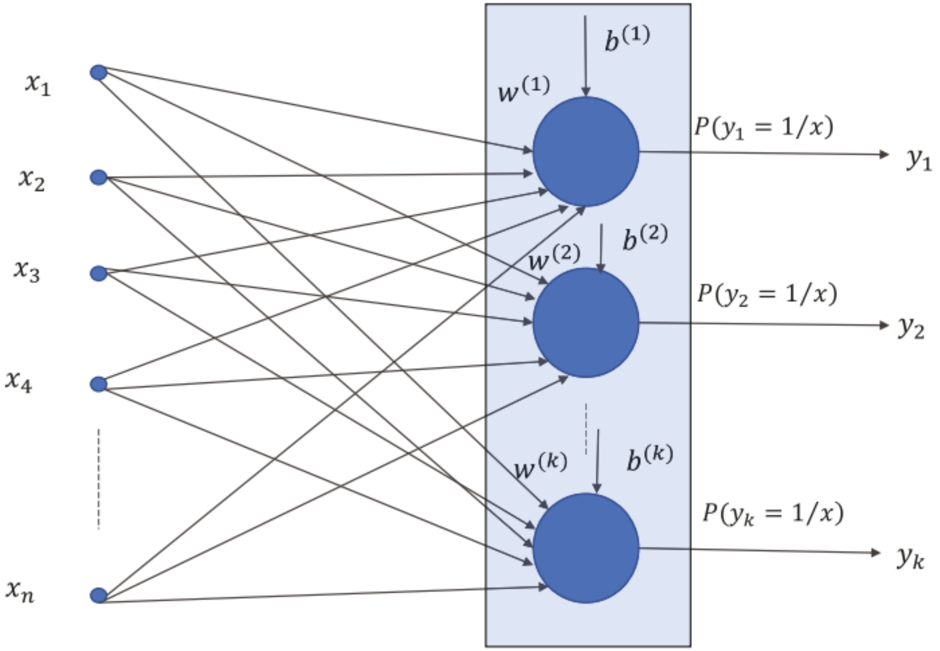
Cross-entropy loss

$$Loss = -\sum_{i=1}^{M} y_i \log(o_i)$$

$$P(y_i = 1/x) = \frac{e^{w^{(i)T}x + b^{(i)}}}{\sum_{j=1}^{k} e^{w^{(j)T}x + b^{(j)}}}$$

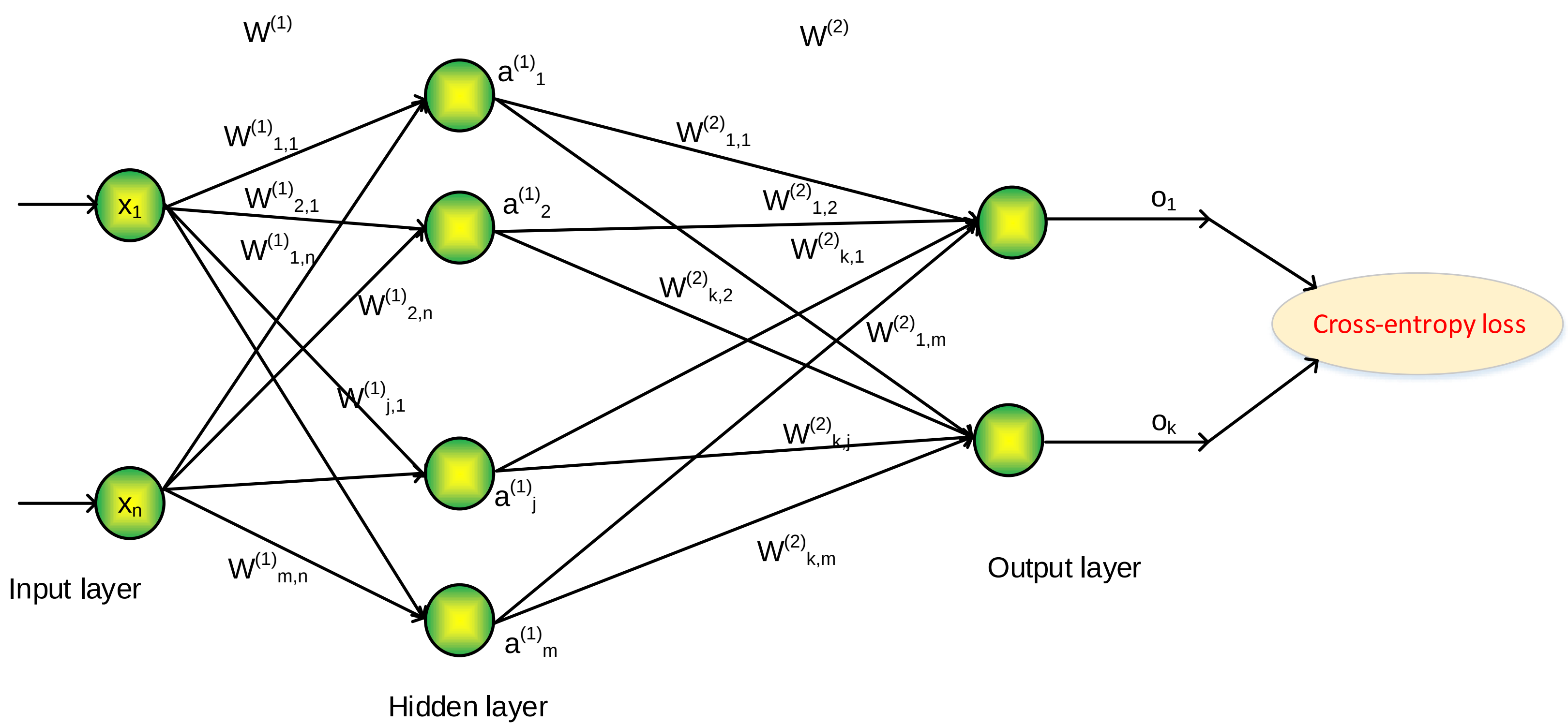
$$C = \sum_{i=1}^{k} -y_i \log P(y_i = 1/x)$$

M – number of classes
y: class label
o: predicted probability observation



Back propagation

Feedforward propagation with softmax activation



$$Loss = -\sum_{k=1}^{K} y_k \log(o_k)$$

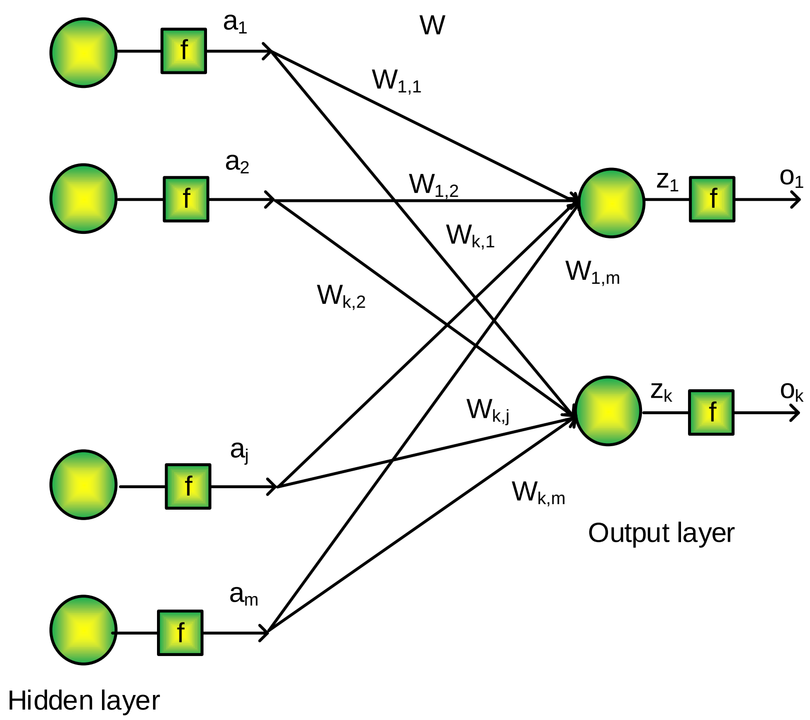
$$z_j^{(1)} = \sum_{i=1}^{M} x_i w_{j,i}^{(1)} + b_j^{(1)}$$

$$a_j^{(1)} = \frac{1}{1 + e^{-z_j^{(1)}}}$$

$$z_k^{(2)} = \sum_{j=1}^{K} a_j^{(1)} w_{k,j}^{(2)} + b_k^{(2)}$$

$$o_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j^{(2)}}}$$

Feedforward propagation with softmax activation



Gradient descent

$$w_{k,j} = w_{k,j} - \eta \frac{\partial L}{\partial w_{k,j}}$$

$$L(y, o) = -\sum_{k=1}^{K} y_k \log(o_k)$$

Apply the chain rule

$$o_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j^{(2)}}}$$

$$\frac{\partial L}{\partial w_{k,j}} = \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{k,j}}$$

$$\frac{\partial L}{\partial z_k} = \frac{\partial L}{\partial o_k} \frac{\partial o_k}{\partial z_k}$$

Error of k^th neuron of the output layer

$$\delta_k = \frac{\partial L}{\partial z_k}$$

Using calculus, we obtain

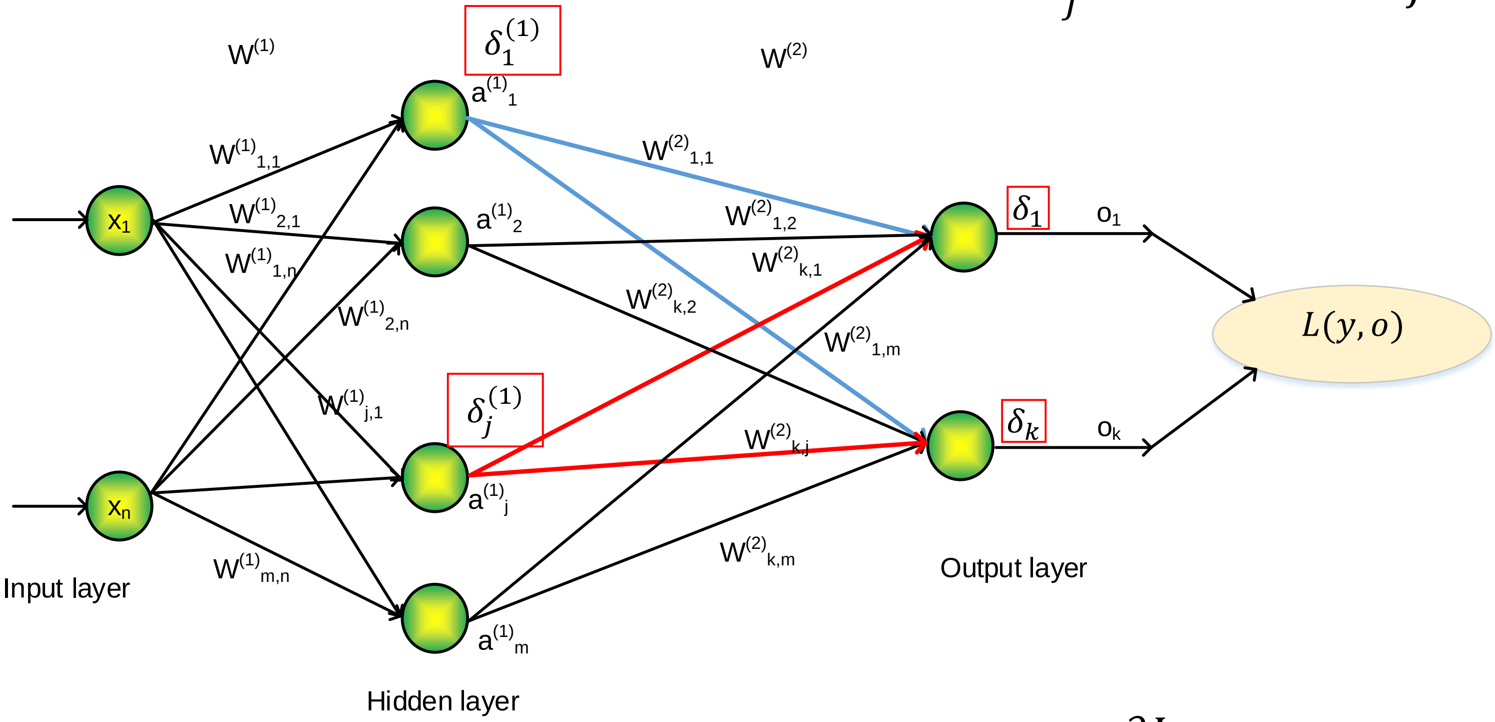
$$\frac{\partial L}{\partial z_k} = o_k - y_k$$

$$\delta_k = o_k - y_k$$

o_k *is the kth component of the neuron's prediction and* y_k *is the kth component of the label*

Back-propagation

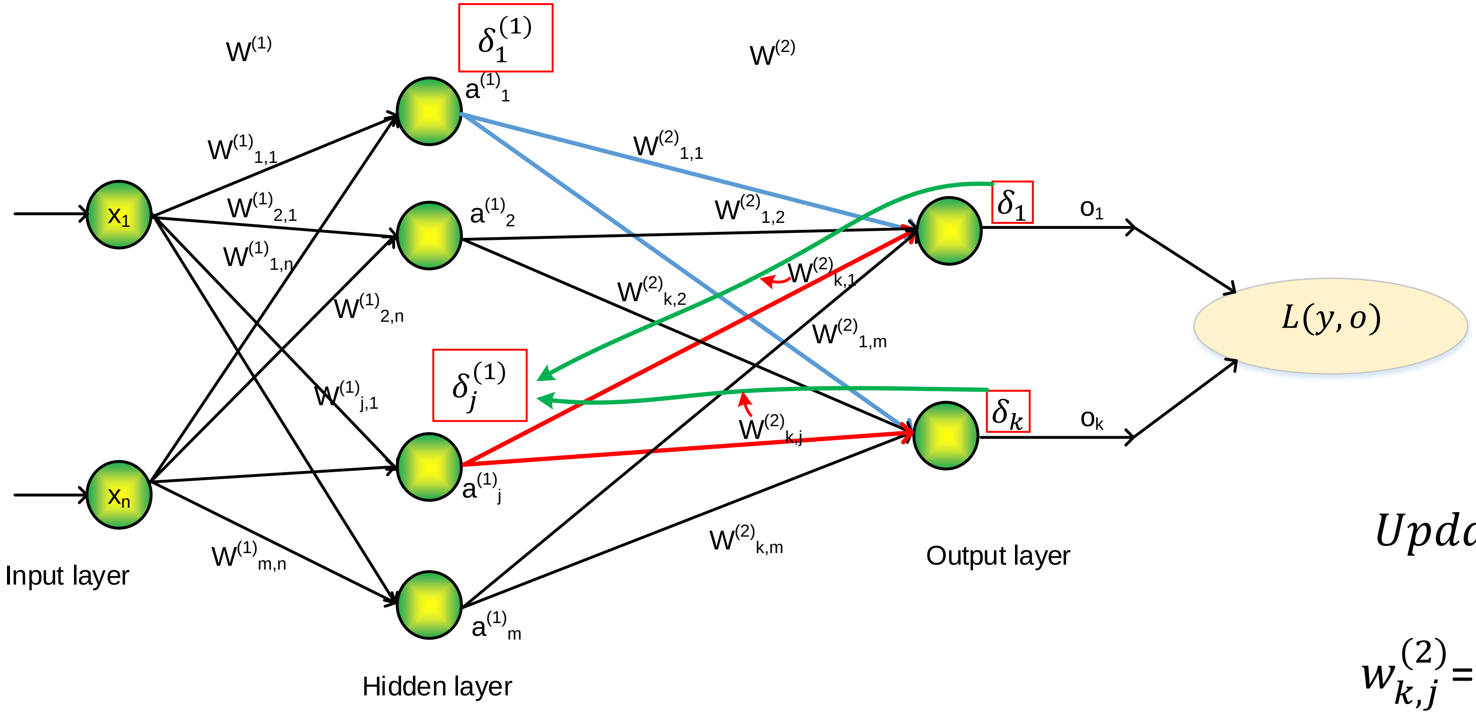
$\delta_j^{(1)}$ *is the error of the ith neuron in hidden layer*



$$\frac{\partial L}{\partial a_j^{(1)}} = \delta_j^{(1)}$$

$\delta_j^{(1)}$ affects all neurons of next layer via the weights

Back-propagation



you should spend time to master them

Back propagate error

$$\delta_1^{(1)} = \delta_1 w_{1,1}^{(2)} + \cdots + \delta_k w_{k,1}^{(2)}$$

$$\delta_j^{(1)} = \sum_{k=1}^{K} \delta_k w_{k,j}^{(2)}$$

Back propagate through sigmoid function

$$\frac{\partial L}{\partial z_j^{(1)}} = \delta_j^{(1)} a_j^{(1)} (1 - a_j^{(1)})$$

Update weights $\quad w_{j,i} = w_{j,i} - \eta \dfrac{\partial L}{\partial w_{j,i}}$

$$w_{k,j}^{(2)} = w_{k,j}^{(2)} - \eta \frac{\partial L}{w_{k,j}^{(2)}} = w_{k,j}^{(2)} - \eta \delta_k \frac{\partial \delta_k}{\partial w_{k,j}^{(2)}}$$

$$w_{k,j}^{(2)} = w_{k,j}^{(2)} - \eta \delta_k a_j^{(1)}$$

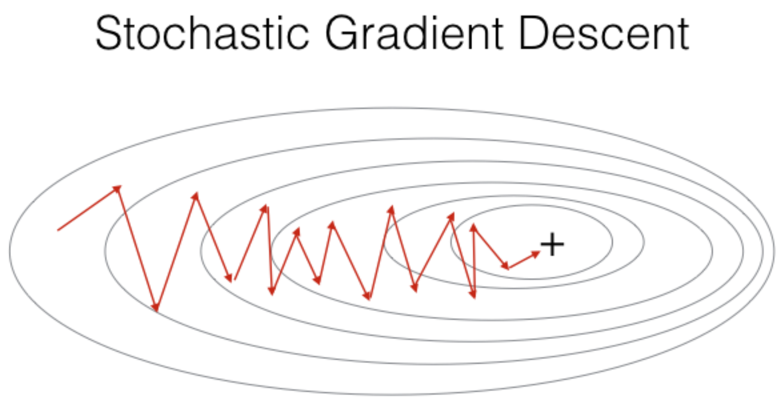
$$\delta_k = o_k - y_k$$

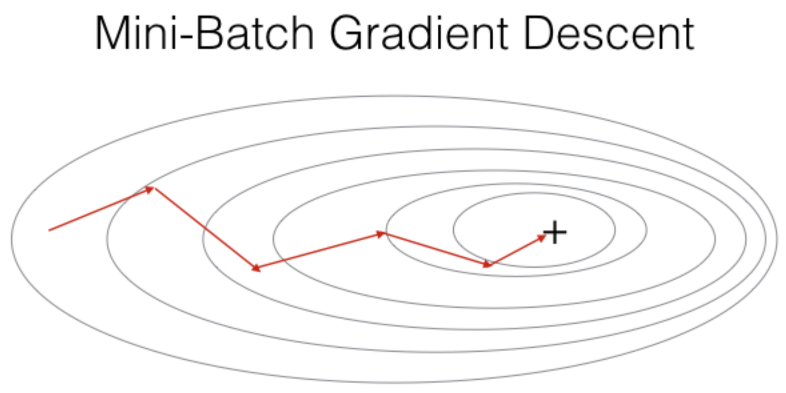
$$\delta_j^{(1)} = \sum_{k=1}^{K} \delta_k w_{k,j}^{(2)}$$

$$w_{j,i}^{(1)} = w_{j,i}^{(1)} - \eta \delta_j^{(1)} a_j^{(1)} (1 - a_j^{(1)}) x_i$$

Stochastic gradient descent – batch and mini-batch gradient descent

- Stochastic gradient descent (SGD): use 1 sample in each iteration
- Batch gradient descent (GD): use all samples in each iteration
- Mini-batch gradient descent (Mini-batch GD): use *b* sample in each iteration, in this case, *b* is the batch size
- Mini-batch stochastic gradient descent (Mini-batch SGD): use *b* sample in each iteration, the batch of training samples is randomly selected



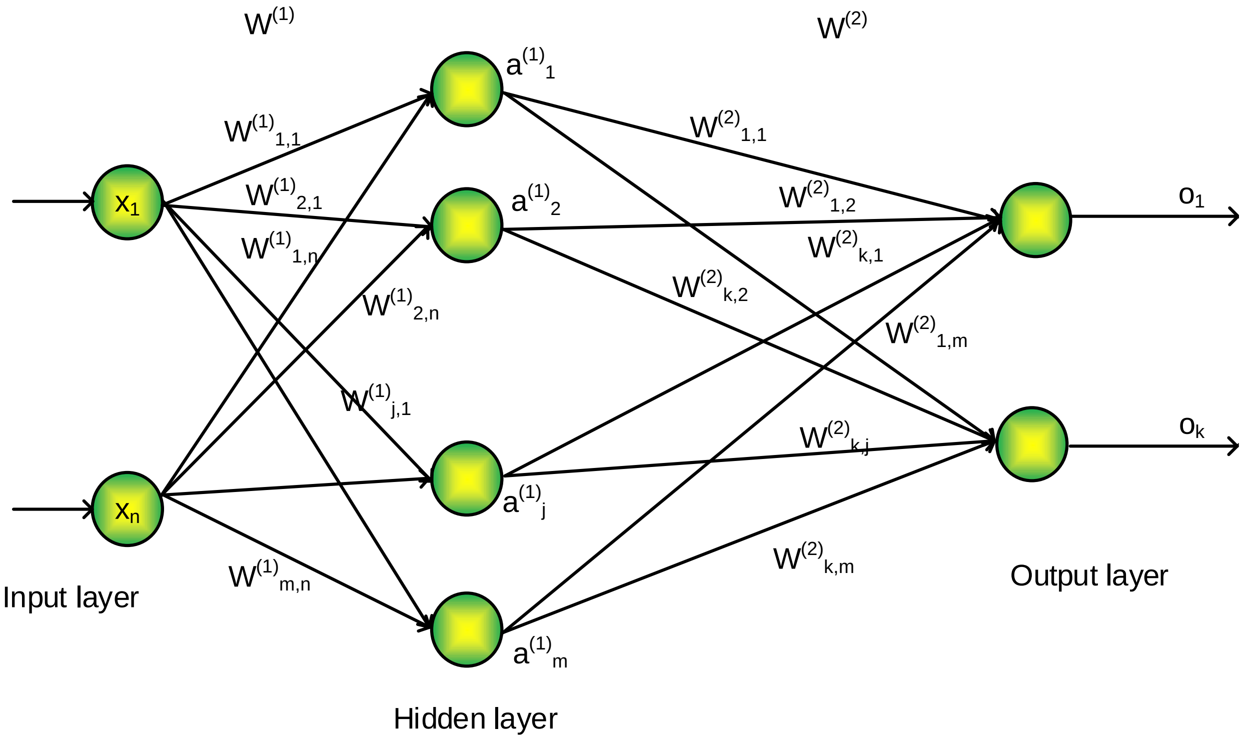


PYTHON CODE

Translating Math into Code

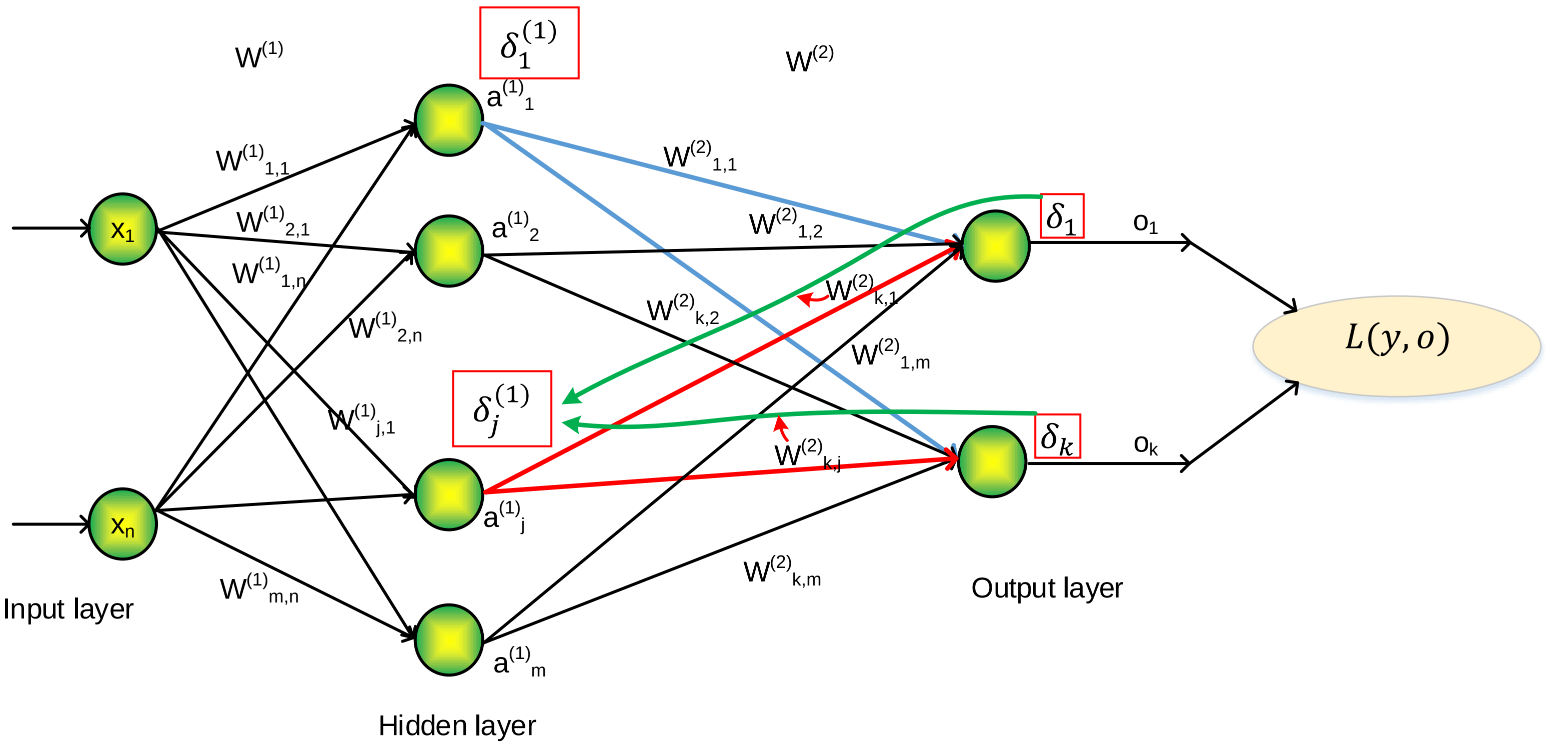
Translating mathematics into code

MNIST Dataset



Feed forward propagation

$$z2 = z^1$$
$$Wh = W^{(1)}$$
$$Wo = W^{(2)}$$

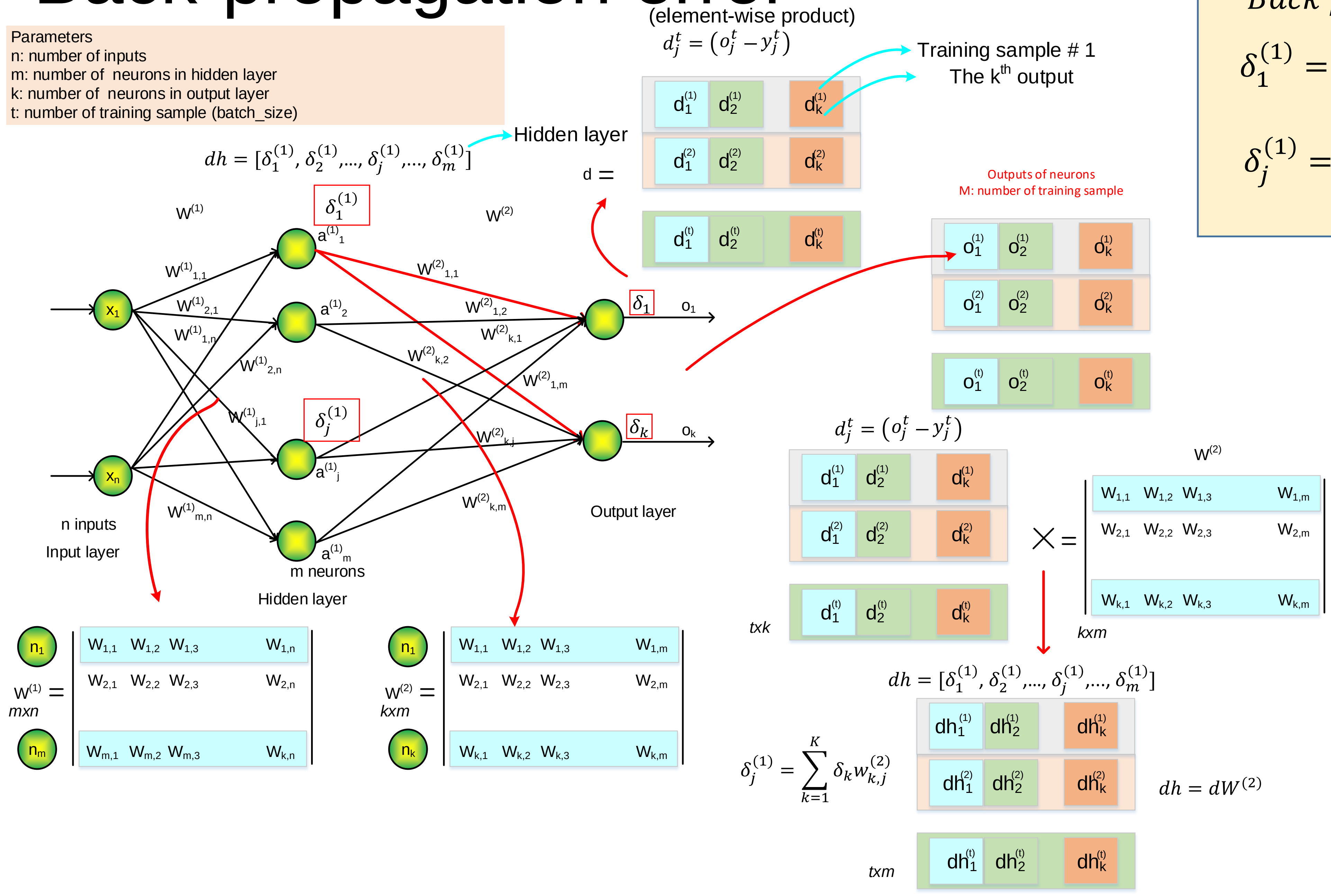


$$Define:$$
$$z2 = z^1$$
$$Wh = W^{(1)}$$
$$Wo = W^{(2)}$$

$$Fordward$$
$$a = XWh^T$$
$$z1 = \sigma(z1) = \frac{1}{1 + e^{-z1}} \qquad z2 = XWo^T \qquad o_k = \frac{e^{z2_k}}{\sum_{j=1}^{K} e^{z2_k}}$$

Back-propagation error

Parameters
n: number of inputs
m: number of neurons in hidden layer
k: number of neurons in output layer
t: number of training sample (batch_size)



Back propagate error

$$\delta_1^{(1)} = \delta_1 w_{1,1}^{(2)} + \cdots + \delta_k w_{k,1}^{(2)}$$

$$\delta_j^{(1)} = \sum_{k=1}^{K} \delta_k w_{k,j}^{(2)}$$

Update weights

output layer

$$L(y,o) = -\sum_{k=1}^{K} y_k \log(o_k)$$

$d = o - y$ *softmax*

$$\Delta Wo = -\eta \frac{2}{t} d^T a$$

$$Wo = \mathrm{Wo} + \Delta \mathrm{Wo}$$

output layer

$dh = dWo$ *back propagate through weights*

$dhs = dh(a(1-a))$ *back propagate through signmoid*

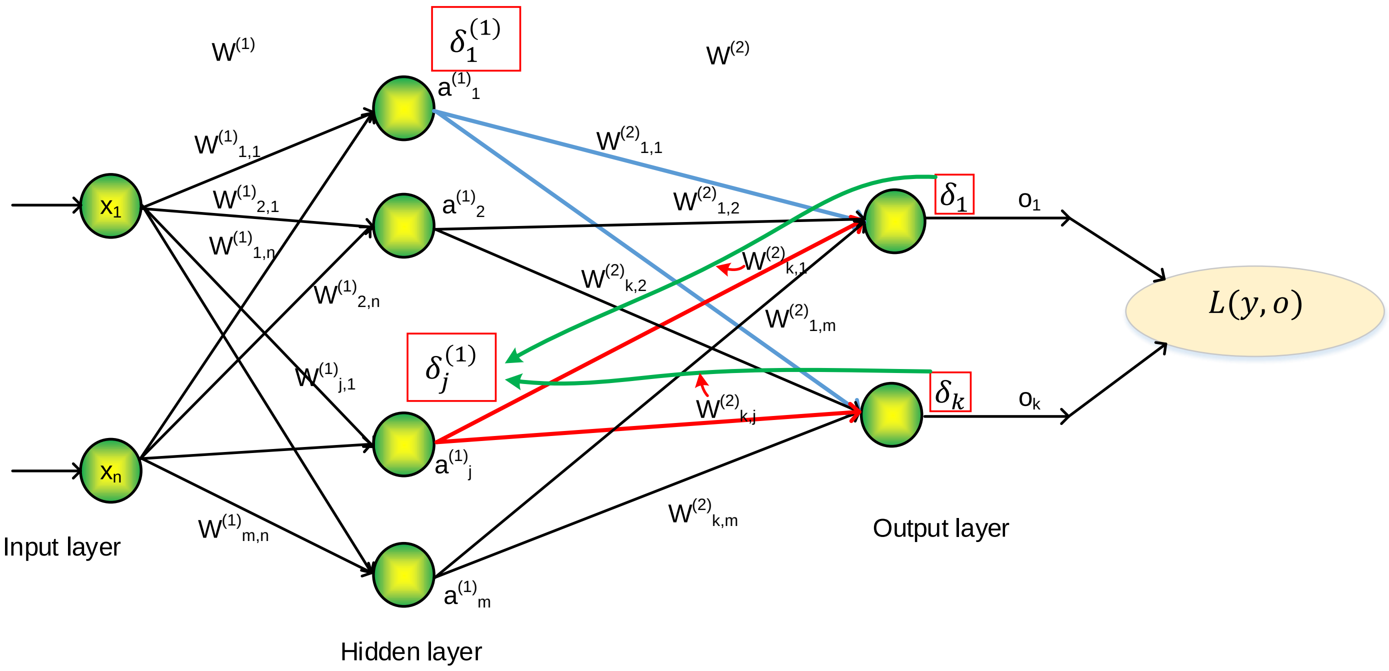
$$\Delta Wo = -\eta \frac{2}{t} dhs^T X$$

t: *number of training sample*

Update weights

$$w_{j,i} = w_{j,i} - \eta \frac{\partial L}{\partial w_{j,i}}$$

$$w_{k,j}^{(2)} = w_{k,j}^{(2)} - \eta \frac{\partial L}{w_{k,j}^{(2)}} = w_{k,j}^{(2)} - \eta \delta_k \frac{\partial \delta_k}{\partial w_{k,j}^{(2)}}$$



PYTHON CODE

Python code

Load dataset

Batch Gradient Descent

```
import numpy as np
import tensorflow as tf
#load datashet
print("Load MNIST Database")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
```

Python code

Define functions

```
def sigmoid(x):
    return 1./(1.+np.exp(-x))

def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(np.exp(x),axis=1)))

def Forwardpass(X,Wh,bh,Wo,bo):
    zh = X@Wh.T + bh
    a = sigmoid(zh)
    z=a@Wo.T + bo
    o = softmax(z)
    return o
def AccTest(label,prediction):    # calculate the matching score
    OutMaxArg=np.argmax(prediction,axis=1)
    LabelMaxArg=np.argmax(label,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
```

Python code

Define network architecture, initialize weights

```
learningRate = 0.5
Epoch=50
NumTrainSamples=60000
NumTestSamples=10000

NumInputs=784
NumHiddenUnits=512
NumClasses=10
#inital weights
#hidden layer
Wh=np.matrix(np.random.uniform(-0.5,0.5,(NumHiddenUnits,NumInputs)))
bh= np.random.uniform(0,0.5,(1,NumHiddenUnits))
dWh= np.zeros((NumHiddenUnits,NumInputs))
dbh= np.zeros((1,NumHiddenUnits))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumClasses,NumHiddenUnits))
bo= np.random.uniform(0,0.5,(1,NumClasses))
dWo= np.zeros((NumClasses,NumHiddenUnits))
dbo= np.zeros((1,NumClasses))
```

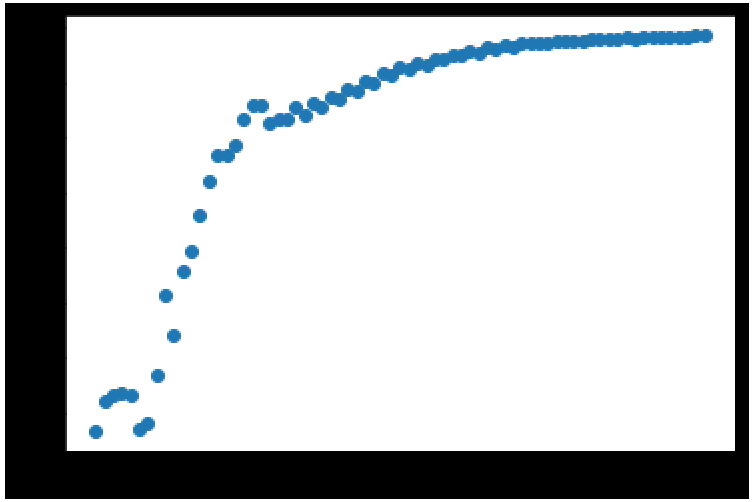
Python code

Batch Gradient Descent

Training the model

```
from IPython.display import clear_output
loss = []
Acc = []
for ep in range (Epoch):
  #feed fordware propagation
  x = x_train
  y=y_train
  zh = x@Wh.T + bh
  a = sigmoid(zh)
  z=a@Wo.T + bo
  o = softmax(z)
  #calculate loss
  loss.append(-np.sum(np.multiply(y,np.log10(o))))
  #calculate the error for the ouput layer
  d = o-y
  #Back propagate error
  dh = d@Wo
  dhs = np.multiply(np.multiply(dh,a),(1-a))
  #update weight
  dWo = np.matmul(np.transpose(d),a)
  dbo = np.mean(d)  # consider a is 1 for bias
  dWh = np.matmul(np.transpose(dhs),x)
  dbh = np.mean(dhs)  # consider a is 1 for bias
  Wo =Wo - learningRate*dWo/NumTrainSamples
  bo =bo - learningRate*dbo
  Wh =Wh-learningRate*dWh/NumTrainSamples
  bh =bh-learningRate*dbh
  #Test accuracy with random initial weights
  prediction = Forwardpass(x_test,Wh,bh,Wo,bo)
  Acc.append(AccTest(y_test,prediction))
  clear_output(wait=True)
  plt.plot([i for i, _ in enumerate(Acc)],Acc,'o')
  plt.show()
```

Python code



Test the model

```
prediction = Forwardpass(x_test,Wh,bh,Wo,bo)
Rate = AccTest(y_test,prediction)
Print(Rate)
```

Python code

Training the model

Mini-Batch Gradient Descent

```
from IPython.display import clear_output
loss = []
Acc = []
Batch_size = 200
Stochastic_samples = np.arange(NumTrainSamples)
for ep in range (Epoch):
  np.random.shuffle(Stochastic_samples)
  for ite in range (0,NumTrainSamples,Batch_size):
    #feed fordware propagation
    Batch_samples = Stochastic_samples[ite:ite+Batch_size]
    x = x_train[Batch_samples,:]
    y=y_train[Batch_samples,:]
    zh = x@Wh.T + bh
    a = sigmoid(zh)
    z=a@Wo.T + bo
    o = softmax(z)
    #calculate loss
    loss.append(-np.sum(np.multiply(y,np.log10(o))))
    #calculate the error for the ouput layer
    d = o-y
    #Back propagate error
    dh = d@Wo
    dhs = np.multiply(np.multiply(dh,a),(1-a))
    #update weight
```

```
#update weight
    dWo = np.matmul(np.transpose(d),a)
    dbo = np.mean(d)  # consider a is 1 for bias
    dWh = np.matmul(np.transpose(dhs),x)
    dbh = np.mean(dhs)  # consider a is 1 for bias
    Wo =Wo - learningRate*dWo/Batch_size
    bo =bo - learningRate*dbo
    Wh =Wh-learningRate*dWh/Batch_size
    bh =bh-learningRate*dbh
    #Test accuracy with random innitial weights
    prediction = Forwardpass(x_test,Wh,bh,Wo,bo)
    Acc.append(AccTest(y_test,prediction))
    clear_output(wait=True)
    plt.plot([i for i, _ in enumerate(Acc)],Acc,'o')
    plt.show()
  print('Epoch:', ep )
  print('Accuracy:',AccTest(y_test,prediction) )
```

Python code

Mini-Batch Gradient Descent

Just calculate the loss and accuracy after each epoch

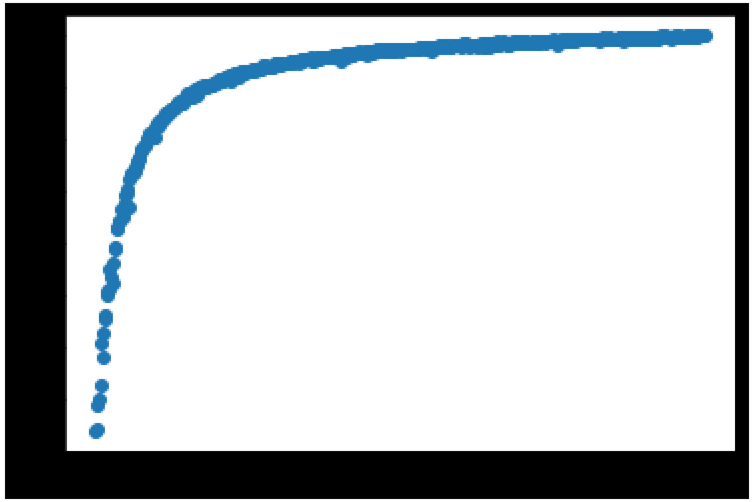
```
from IPython.display import clear_output
loss = []
Acc = []
Batch_size = 200
Stochastic_samples = np.arange(NumTrainSamples)
for ep in range (Epoch):
  np.random.shuffle(Stochastic_samples)
  for ite in range (0,NumTrainSamples,Batch_size):
    #feed fordware propagation
    Batch_samples =
Stochastic_samples[ite:ite+Batch_size]
    x = x_train[Batch_samples,:]
    y=y_train[Batch_samples,:]
    zh = x@Wh.T + bh
    a = sigmoid(zh)
    z=a@Wo.T + bo
    o = softmax(z)
    #calculate loss
    loss.append(-np.sum(np.multiply(y,np.log10(o))))
    #calculate the error for the ouput layer
    d = o-y
    #Back propagate error
    dh = d@Wo
    dhs = np.multiply(np.multiply(dh,a),(1-a))
dWo = np.matmul(np.transpose(d),a)
    dbo = np.mean(d)  # consider a is 1 for bias
    dWh = np.matmul(np.transpose(dhs),x)
    dbh = np.mean(dhs)  # consider a is 1 for bias
    Wo =Wo - learningRate*dWo/Batch_size
    bo =bo - learningRate*dbo
    Wh =Wh-learningRate*dWh/Batch_size
    bh =bh-learningRate*dbh
  #Test accuracy with random innitial weights
  prediction = Forwardpass(x_test,Wh,bh,Wo,bo)
  Acc.append(AccTest(y_test,prediction))
  print('Epoch:', ep )
  print('Accuracy:',AccTest(y_test,prediction) )
```

```
Epoch: 0
Accuracy: 0.8762
Epoch: 1
Accuracy: 0.9013
Epoch: 2
Accuracy: 0.9136
Epoch: 3
Accuracy: 0.9165
Epoch: 4
Accuracy: 0.9251
```

Python code

Training the model

Mini-Batch Gradient Descent




Math