Deep learning
-- Hello world

Junjie Cao @ DLUT

Spring 2019

Feature Extractor

*

32x32
“Airplane
Filter”

=

“probability” of
the image being
an airplane

Feature Extractor

*

32x32
“Airplane
Filter”

=

“probability” of
the image being
an airplane

Feature Extractor

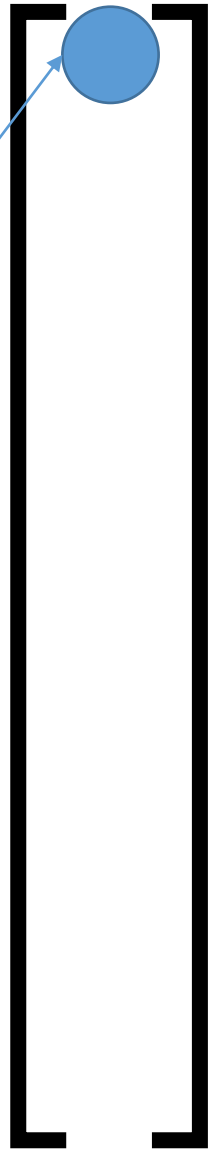
This is not really a probability but a score, because it can be less than 0 and greater than 1

*

32x32 "Airplane Filter"

=

"probability" of the image being an airplane



Feature Extractor

*

32x32
“Automobile
Filter”

=

“probability” of
the image being
an automobile

Feature Extractor

*

32x32 “Bird Filter”

=

“probability” of the image being a bird

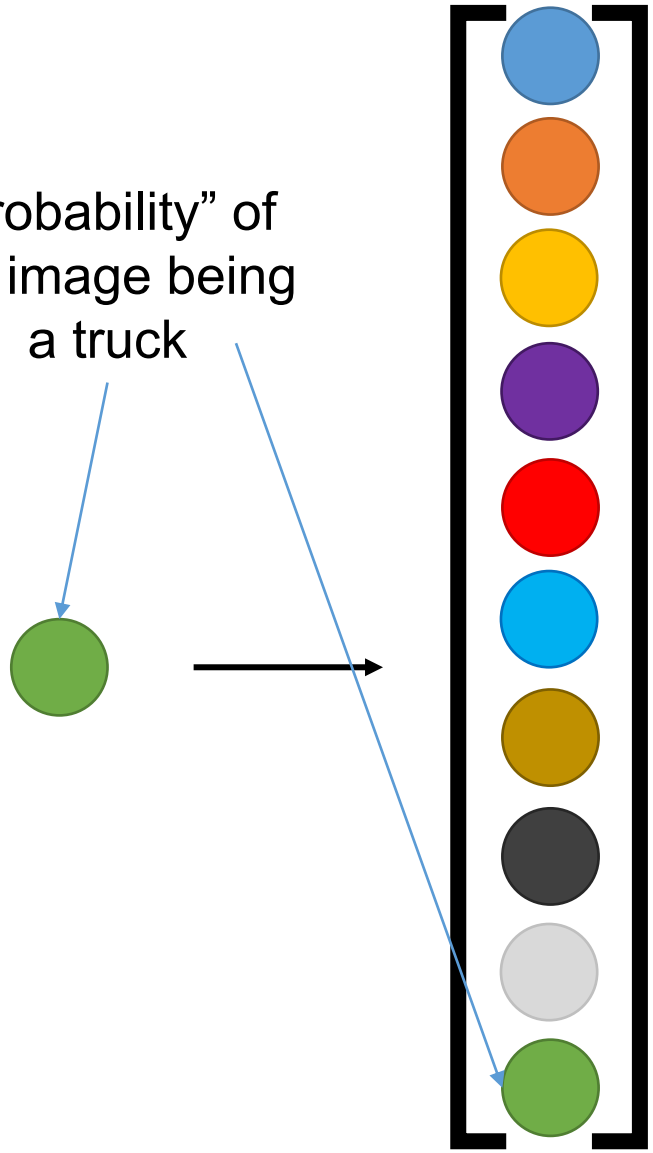
Feature Extractor

*

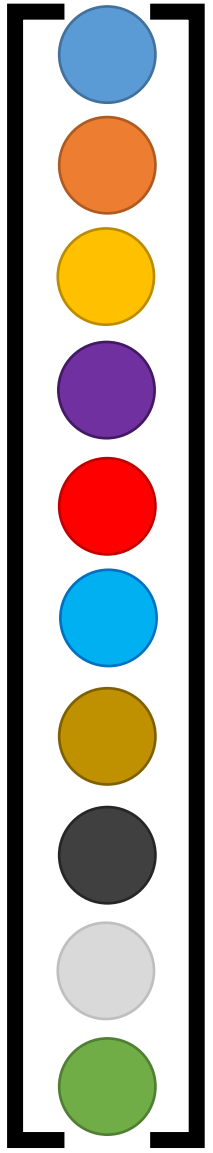
32x32 “Truck Filter”

=

“probability” of the image being a truck

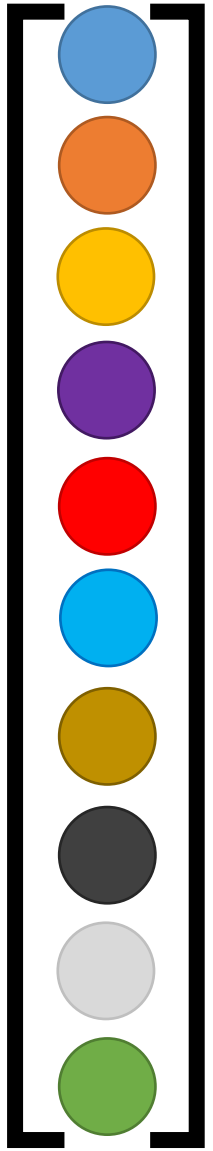


Classifier

$$c_{pred} = \arg\max\left(\right.$$  $$\left.\right)$$

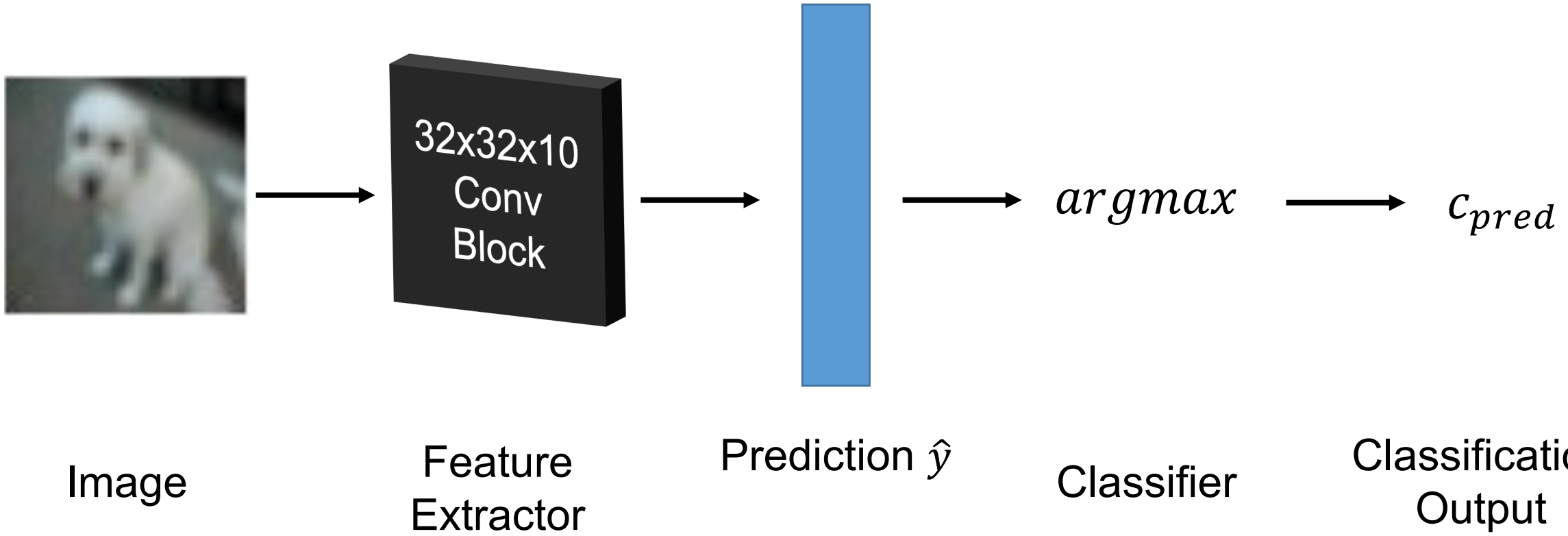
Classifier

$$c_{pred} = \arg\max(\quad)$$

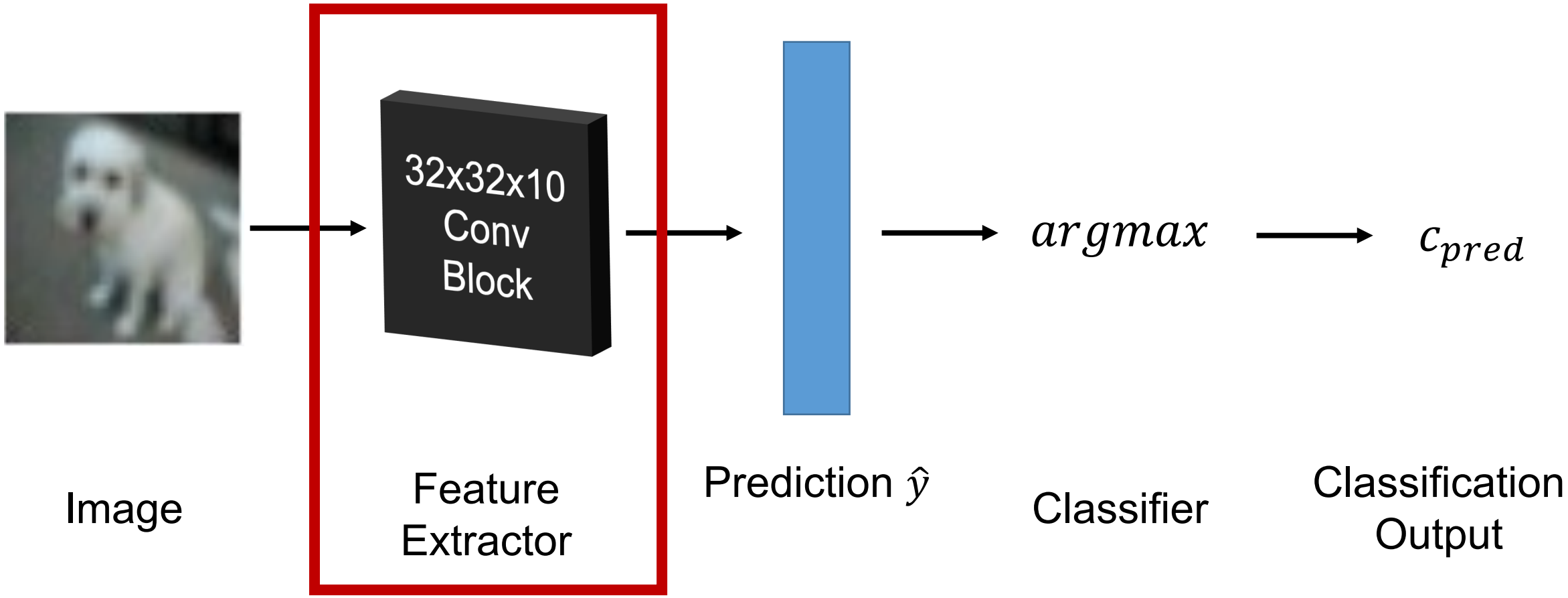


We predict the class that has the highest probability!

The Whole Shebang

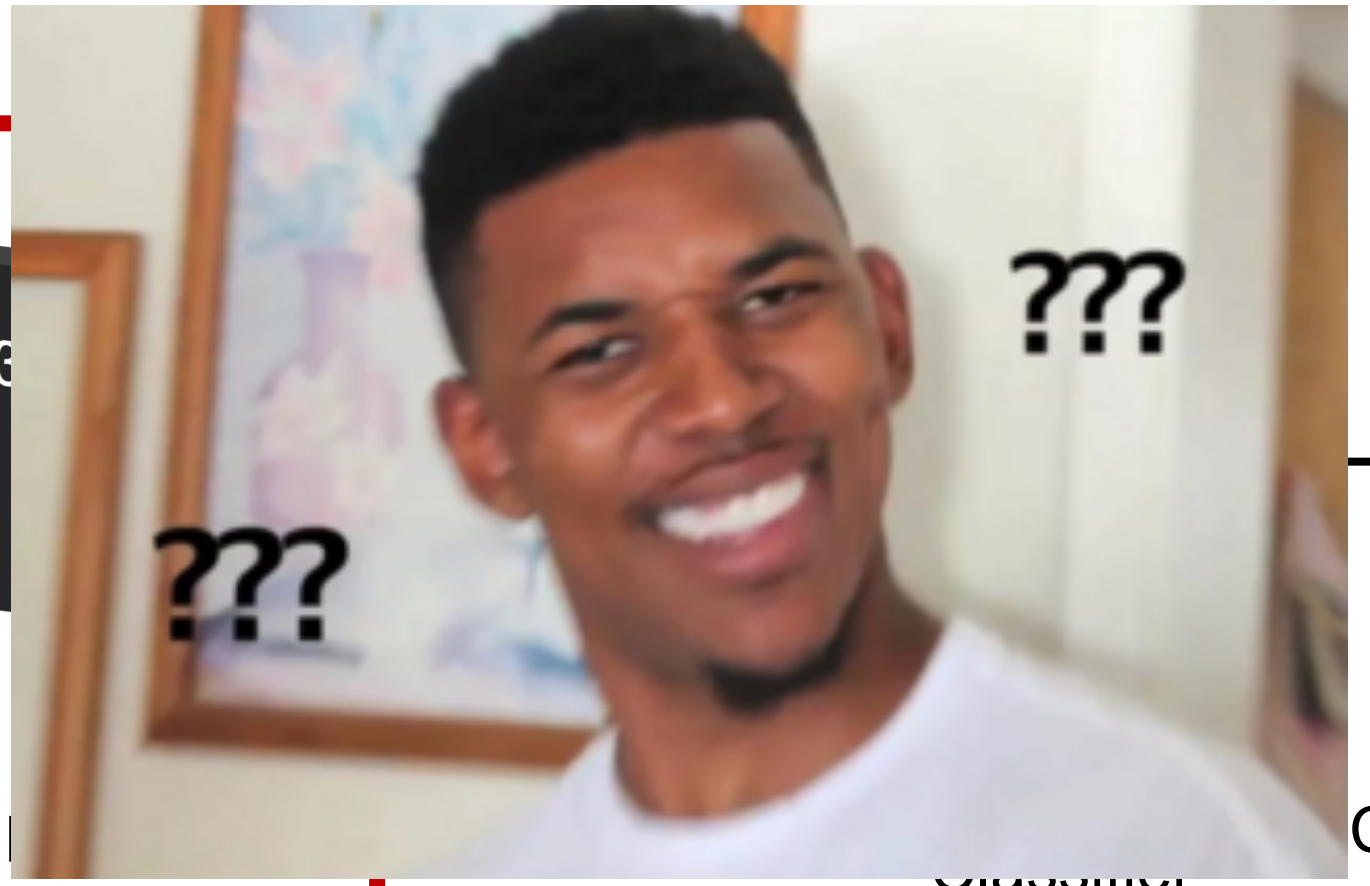


The Whole Shebang



The Whole Shebang

Image



Extractor

Classifier

c_{pred}

Classification Output

Reframing convolution

12	21
18	31

*

1	2
3	4

Reframing convolution

12	21
18	31

*

1	2
3	4

$$= \begin{bmatrix} 12 \\ 21 \\ 18 \\ 31 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

Reframed Feature Extractor

*

32x32
“Airplane
Filter”

Reframed Feature Extractor

*

32x32
“Airplane
Filter”

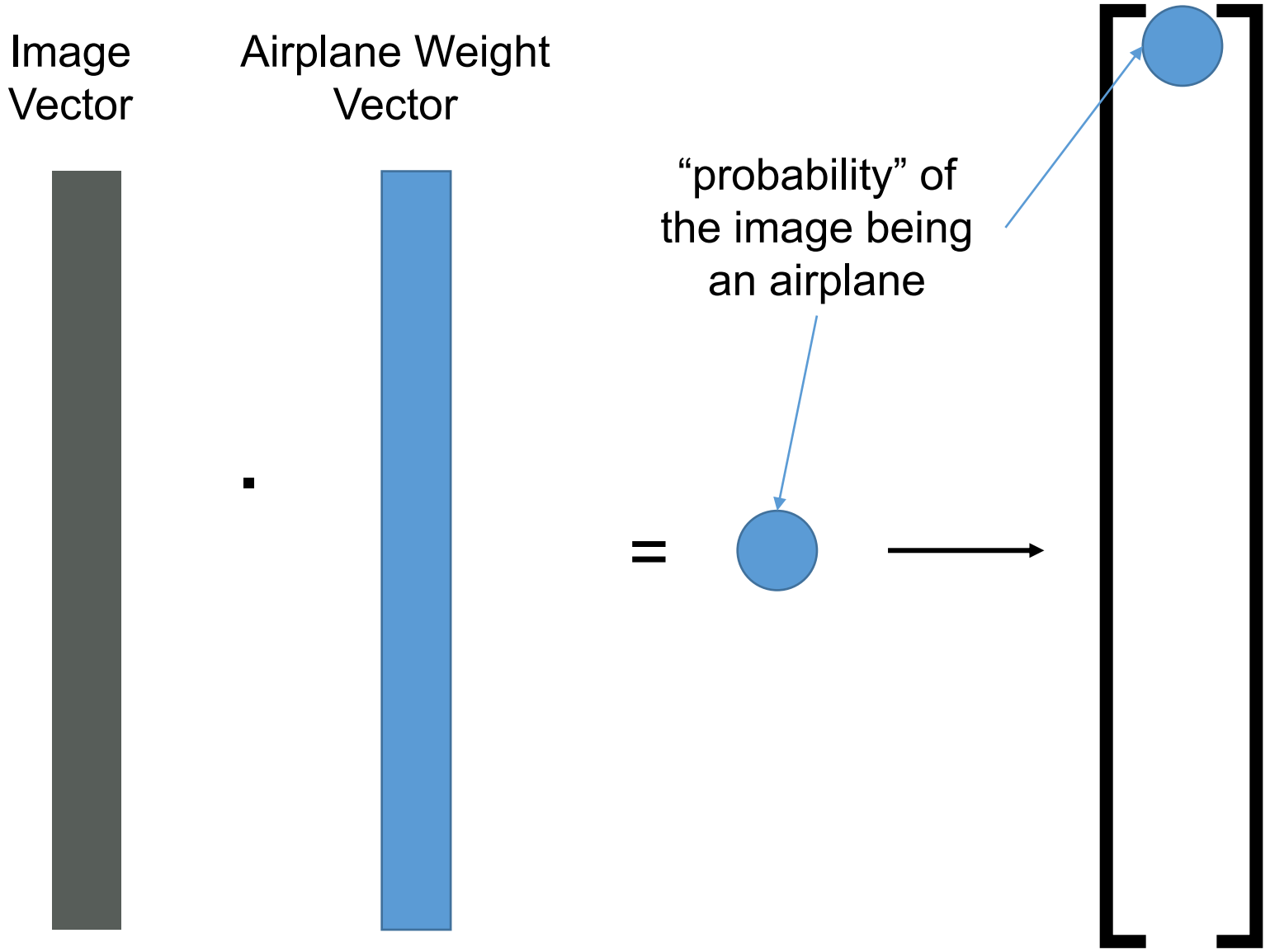
=

Image
Vector

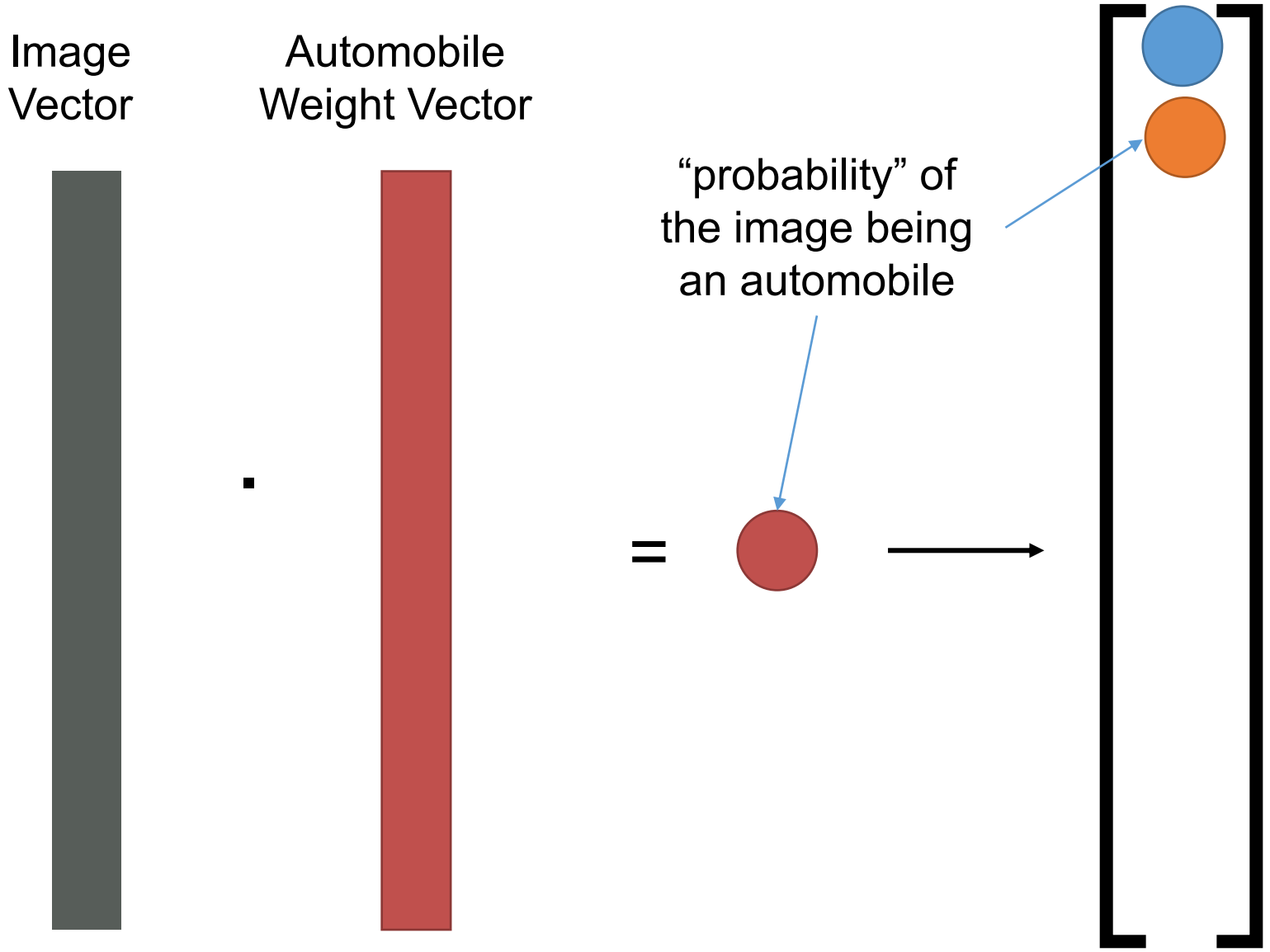
.

Airplane Weight
Vector

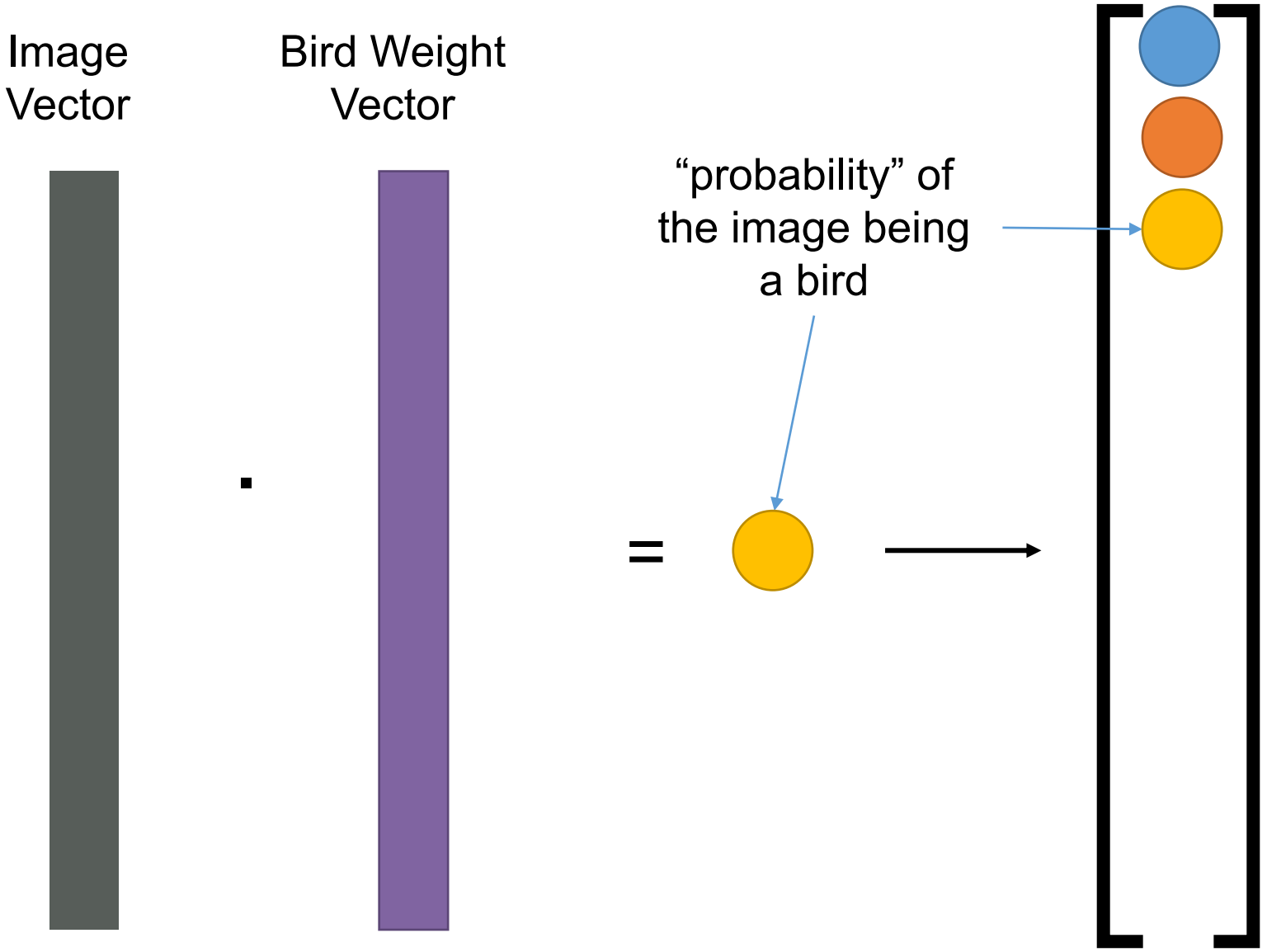
New Feature Extractor



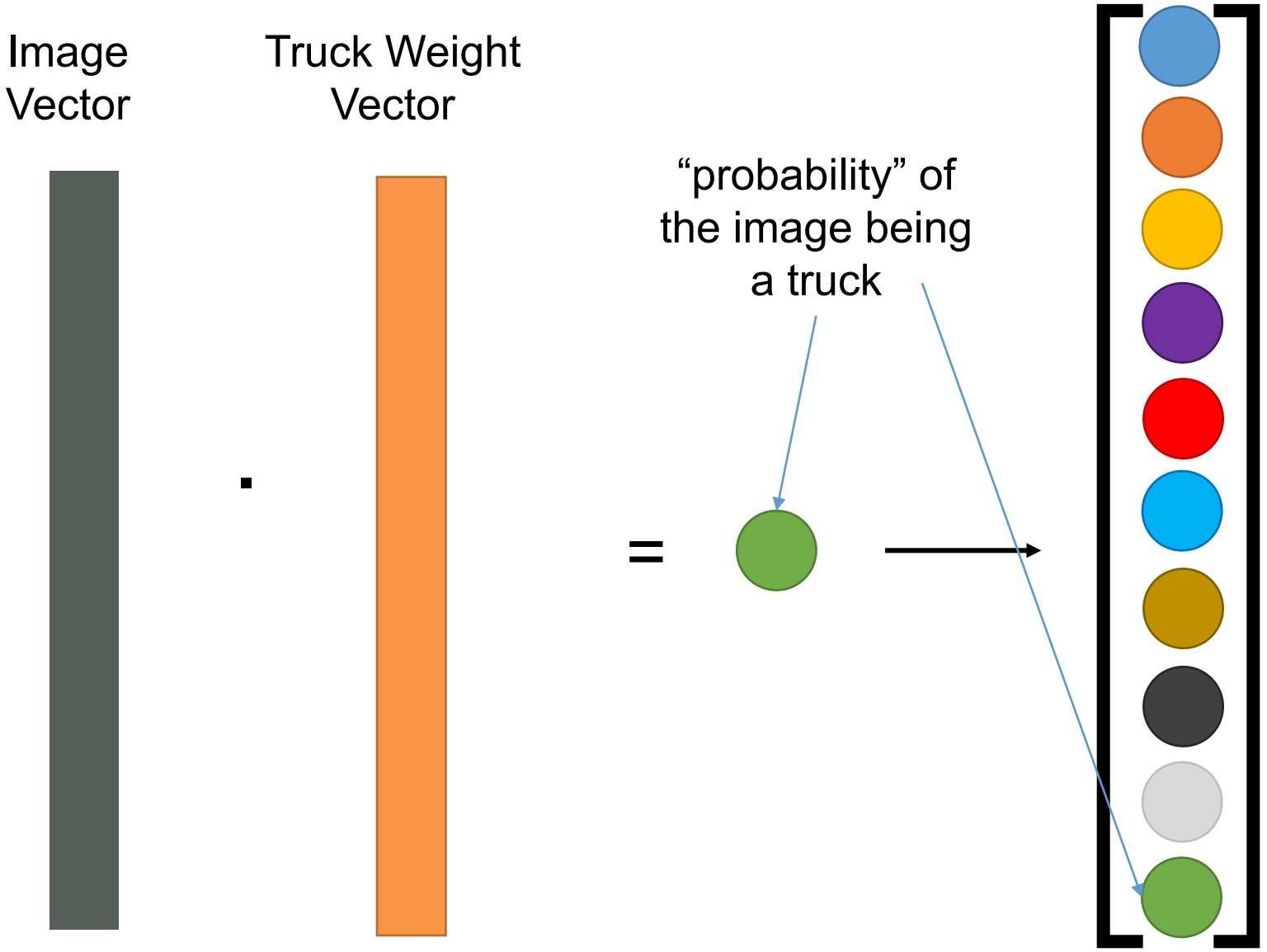
New Feature Extractor



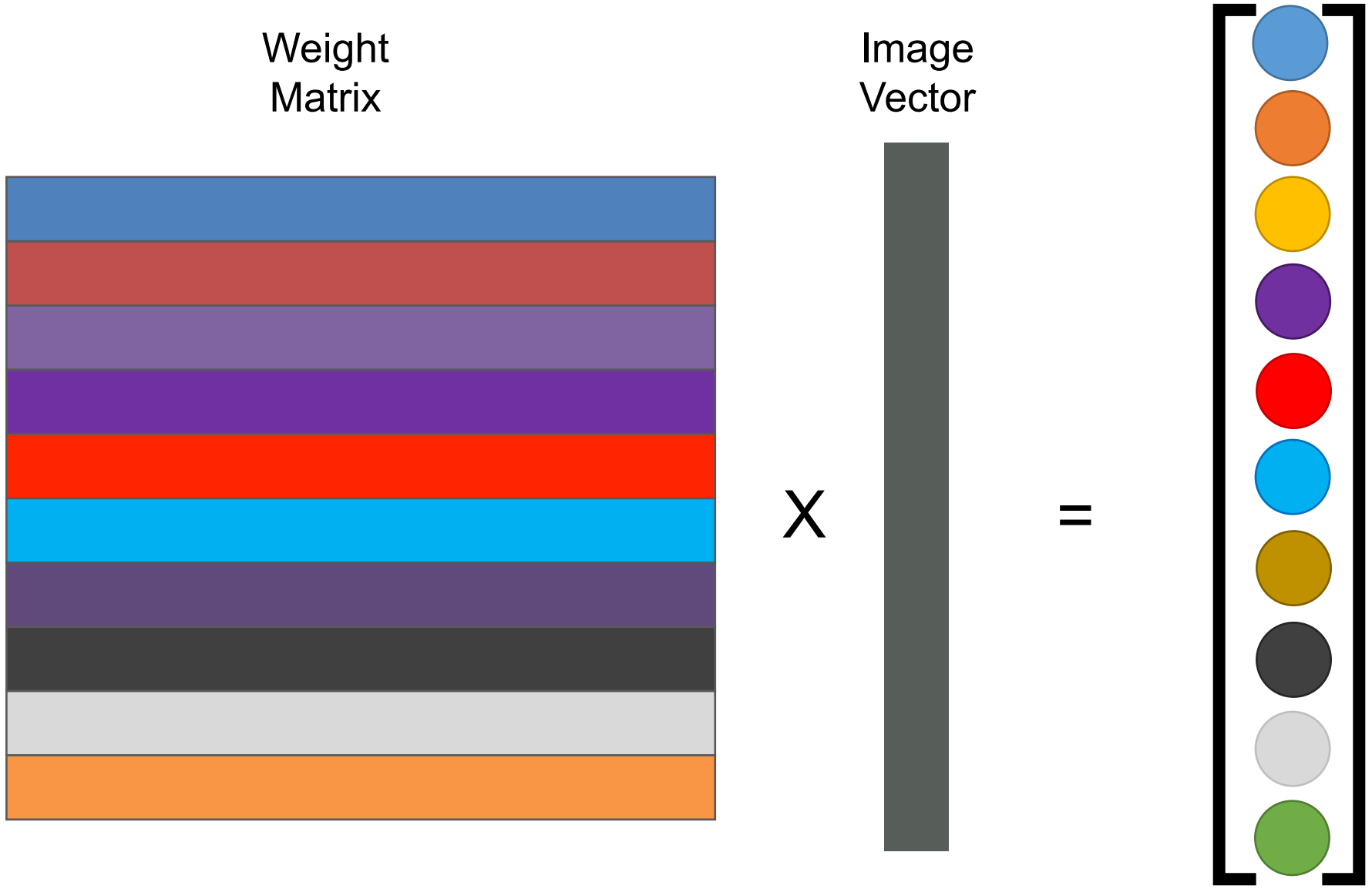
New Feature Extractor



New Feature Extractor



New Feature Extractor



New Feature Extractor

$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”

New Feature Extractor

This simple computation is called a *fully-connected layer*!

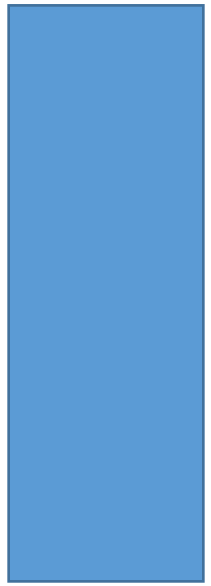
$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”

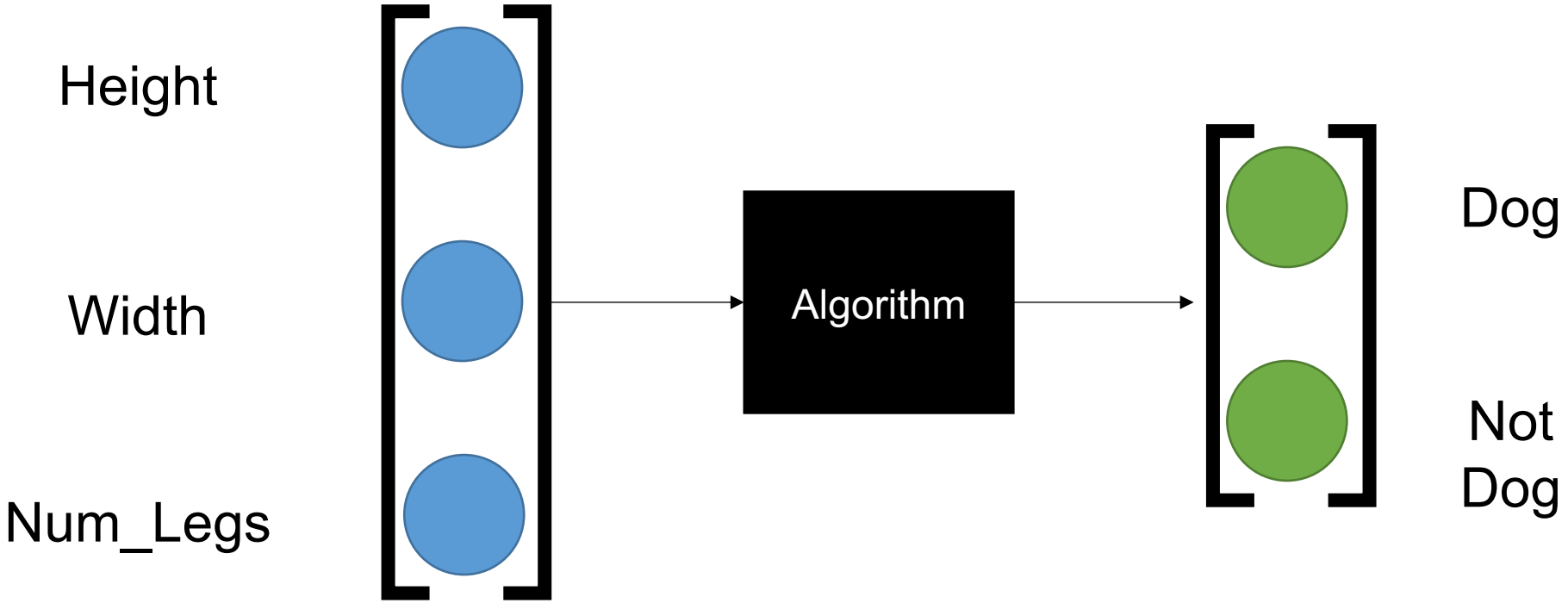
Aside: Fully-Connected Neural Networks

$$\begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \end{bmatrix} \cdot$$  $$=$$ 

W x $\hat{y}$

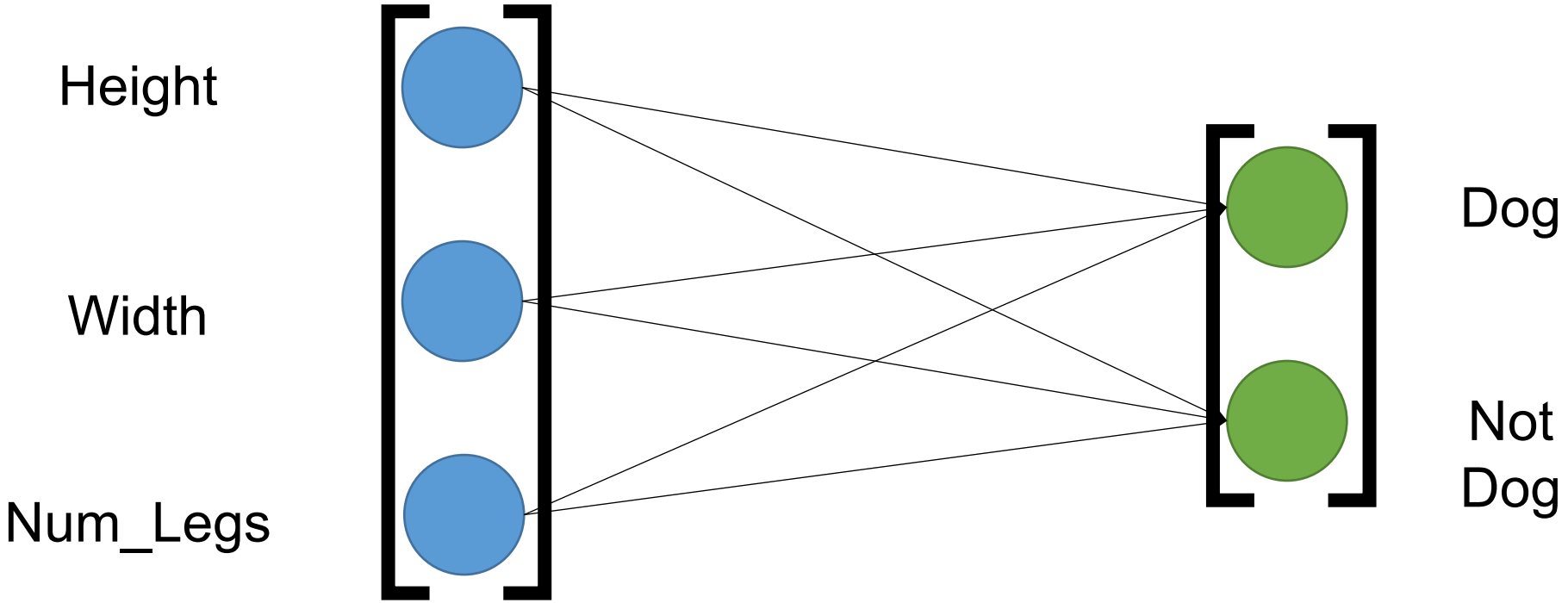
$$Wx = \hat{y}$$

Aside: Fully-Connected Neural Networks



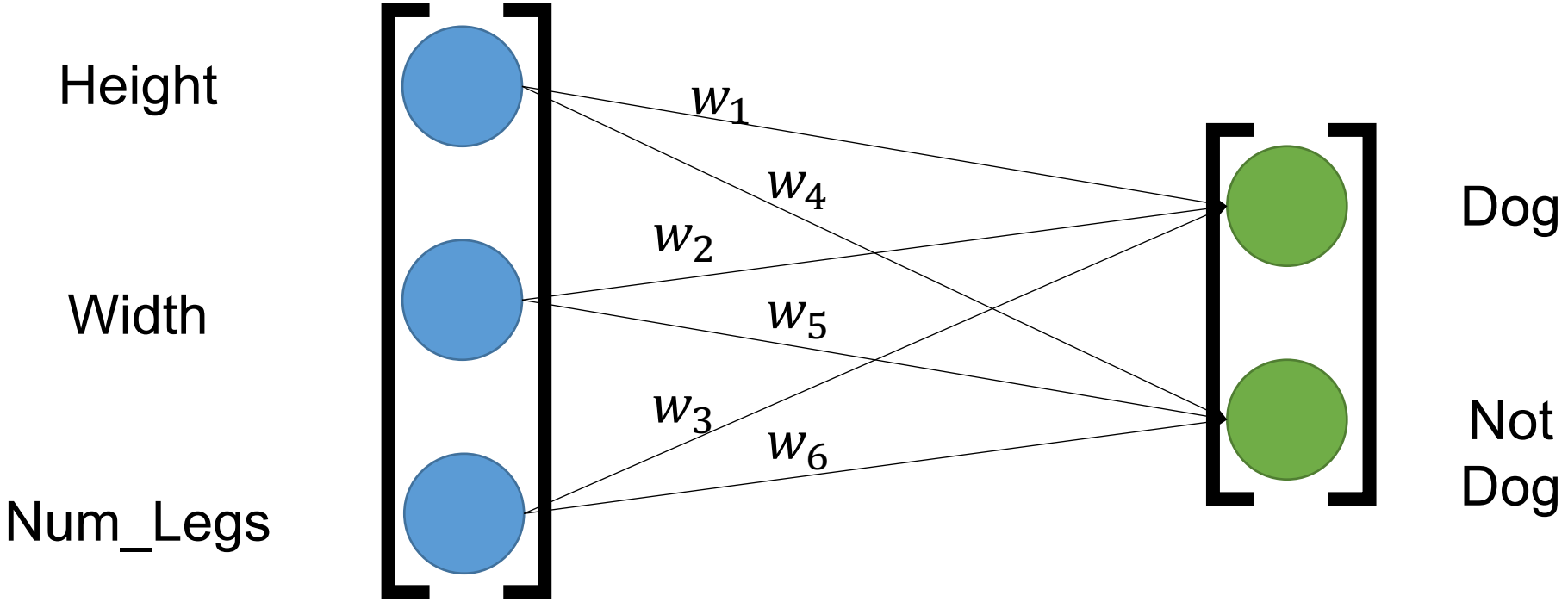
$$Wx = \hat{y}$$

Aside: Fully-Connected Neural Networks



$$Wx = \hat{y}$$

Aside: Fully-Connected Neural Networks



Every node is connected to every other node

$$Wx = \hat{y}$$

New Feature Extractor

$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

$\hat{y}$: the (10x1) vector of class “probabilities”

New Feature Extractor

$$Wx = \hat{y}$$

W: the (10x1024) matrix of weight vectors

x: the (1024x1) image vector

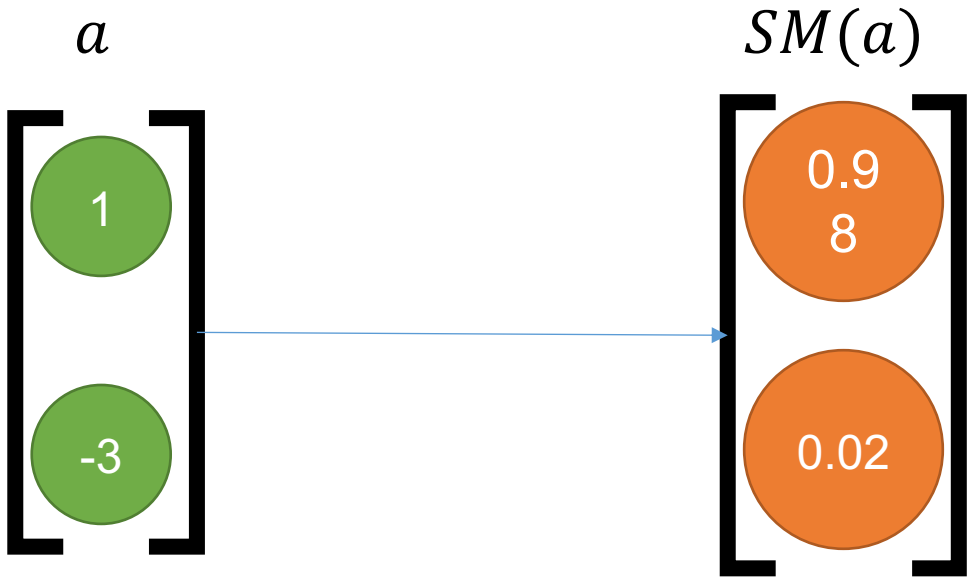
$\hat{y}$: the (10x1) vector of class “probabilities”?

Class Probability Vector

- Must have values between 0 and 1
- Must sum to 1
- There's no guarantee either requirement is satisfied!

$$\hat{y} = Wx$$

Softmax Function



Softmax: $a(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$

Class Probability Vector

- Must have values between 0 and 1

- Must sum to 1

$$\hat{y} = Wx$$

Class Probability Vector

- Must have values between 0 and 1

- Must sum to 1

$$\hat{y} = SM(Wx)$$

System so far…

- Feature extractor:

- Classifier:

$$\hat{y} = SM(Wx)$$

$$c_{pred} = \arg\max(\hat{y})$$

System so far…

- Feature extractor:

- Classifier:

$$\hat{y} = SM(Wx)$$

$$c_{pred} = \arg\max(\hat{y})$$

System so far…

- Feature extractor:
- Classifier:

Using the label

Let's compare our prediction with the real answer! For each image, we have the label y which tells us the true class:

X

y

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Dog class index

Key Insight:

We want:

$$\arg\max(\hat{y}) = \arg\max(y)$$

Which we can accomplish by *Cross-entropy loss function*:

$$L(\boldsymbol{x}, y; \boldsymbol{W}) = -\sum_j y_j \log p\left(c_j \middle| \boldsymbol{x}\right) = -\log p\left(c_j \middle| \boldsymbol{x}\right),$$

where $\hat{y}_j = p(c_j|\boldsymbol{x})$

$$W^* = \arg\min_W \left(\sum_{x,y} L(\boldsymbol{x}, y; \boldsymbol{W}) \right)$$

Cross-Entropy Loss

Our loss function represents *how bad we are currently doing*:

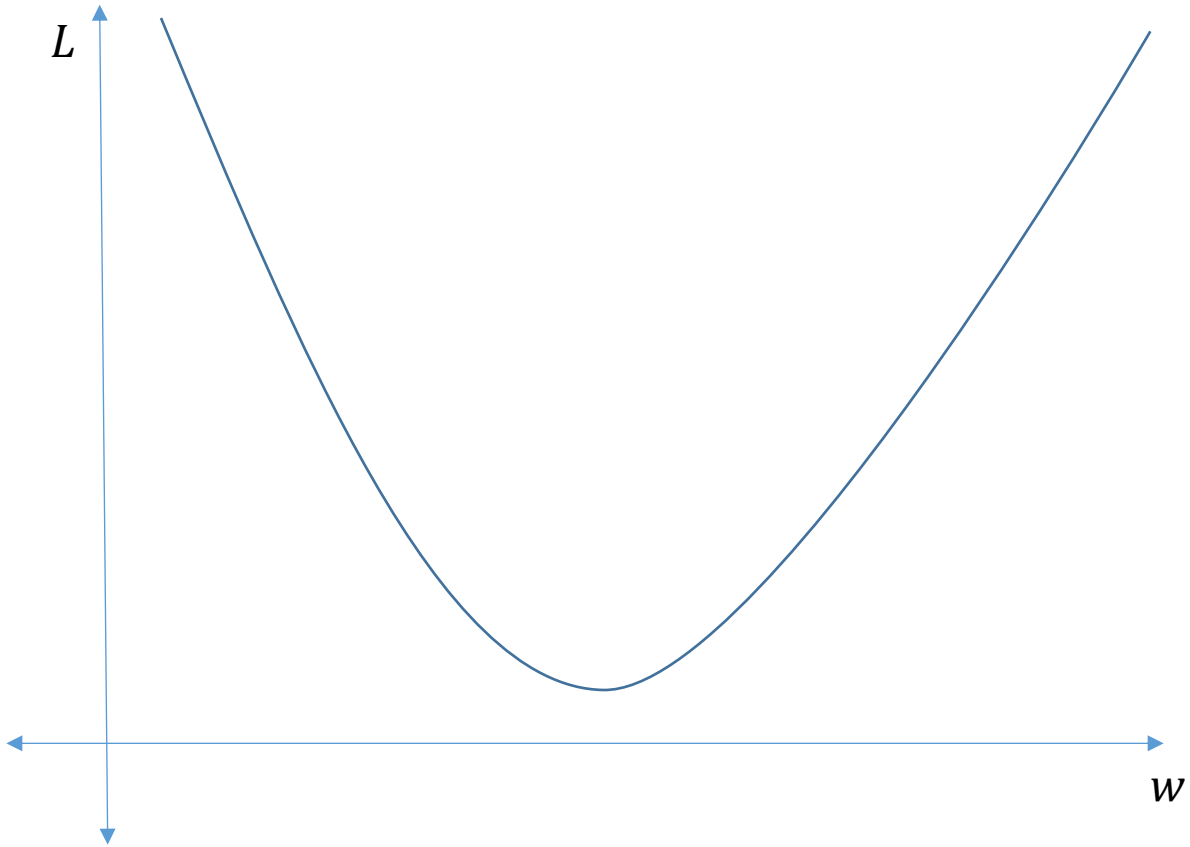
$$L = -\log p\left(c_j \middle| \boldsymbol{x}\right)$$

Examples:

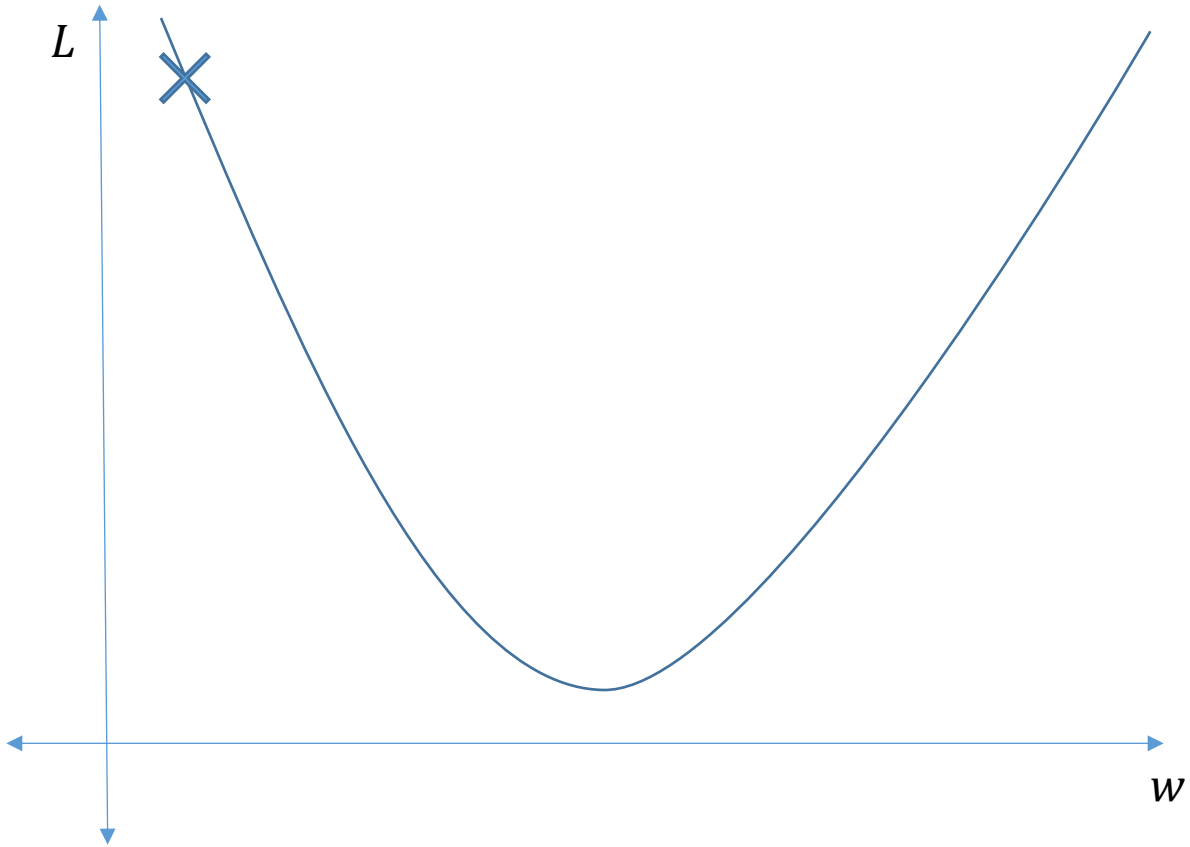
$$p\left(c_j \middle| \boldsymbol{x}\right) = 0 \rightarrow L = -\log(0) = \infty$$
$$p\left(c_j \middle| \boldsymbol{x}\right) = 0.1 \rightarrow L = -\log(0.1) = 2.3$$
$$p\left(c_j \middle| \boldsymbol{x}\right) = 0.9 \rightarrow L = -\log(0.9) = 0.1$$
$$p\left(c_j \middle| \boldsymbol{x}\right) = 1 \rightarrow L = -\log(1) = 0$$

The larger the loss, the worse our prediction. We want to minimize L!

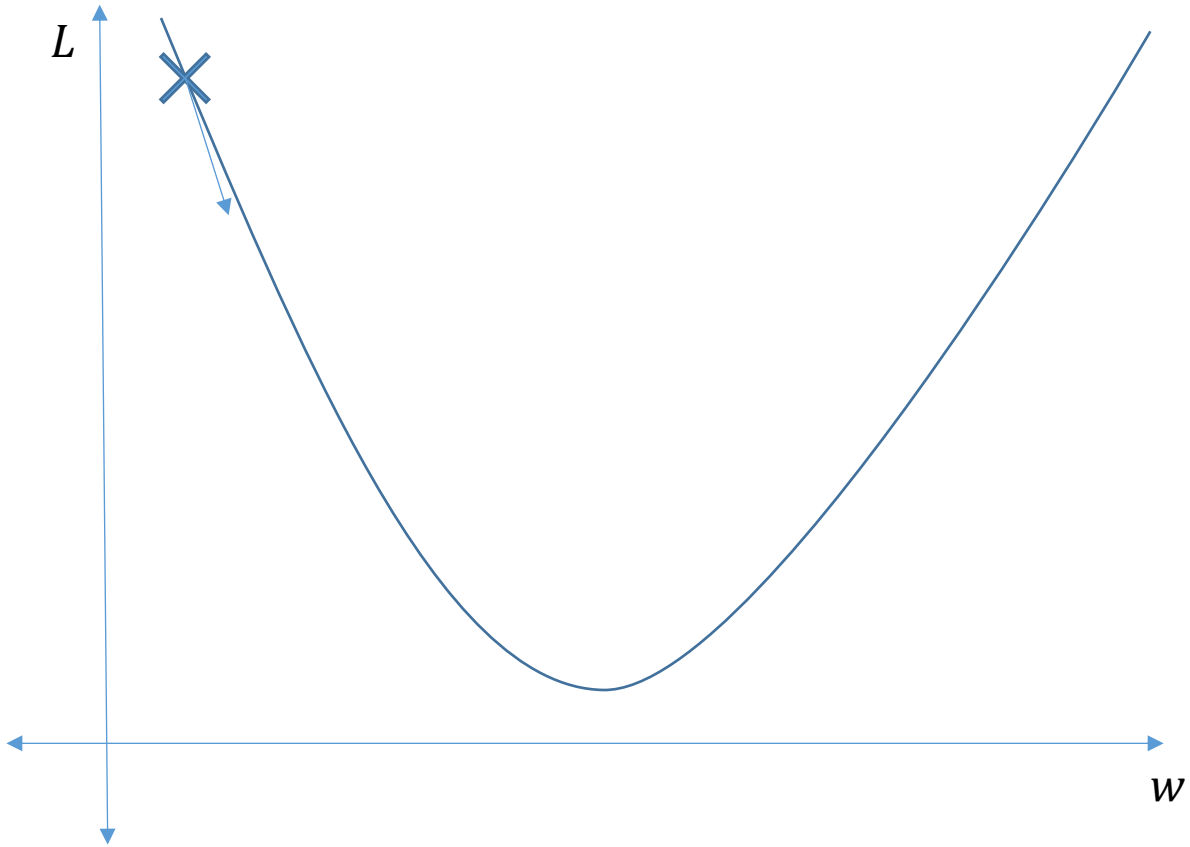
Minimizing Loss



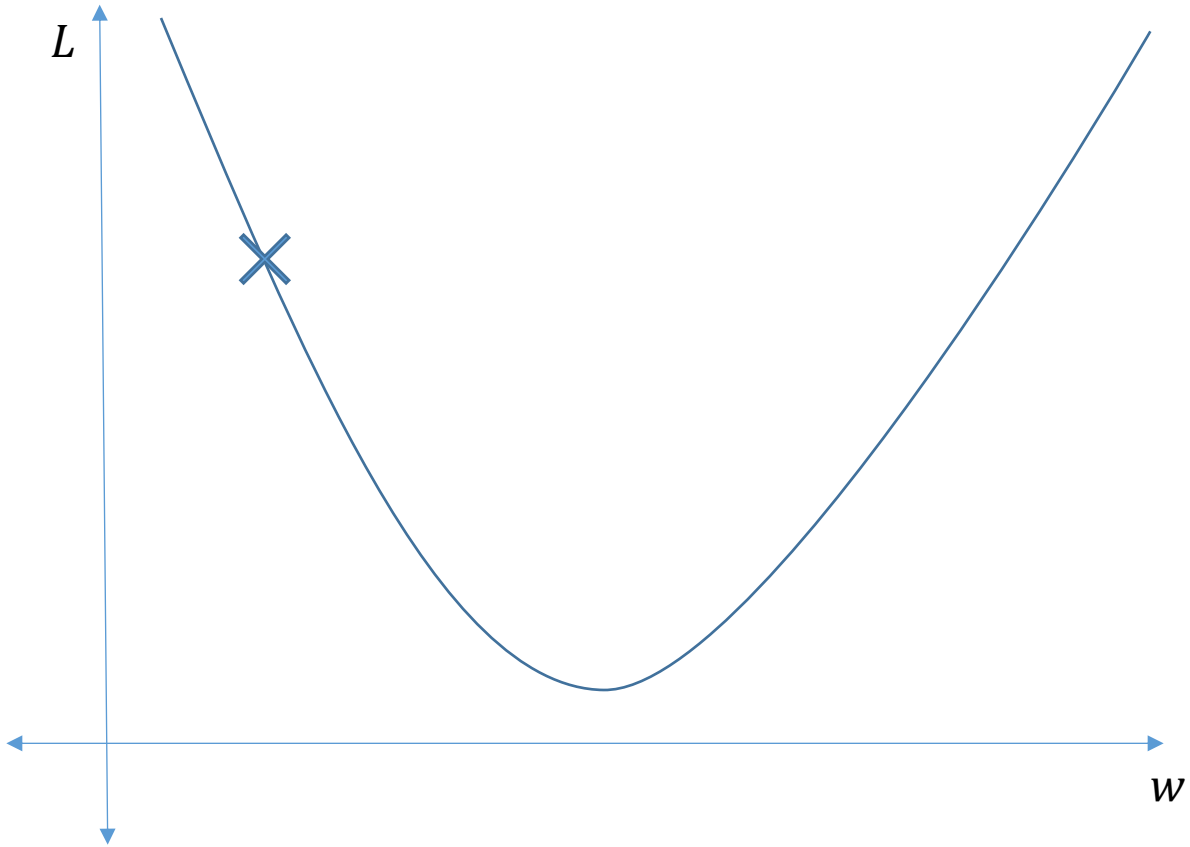
Minimizing Loss



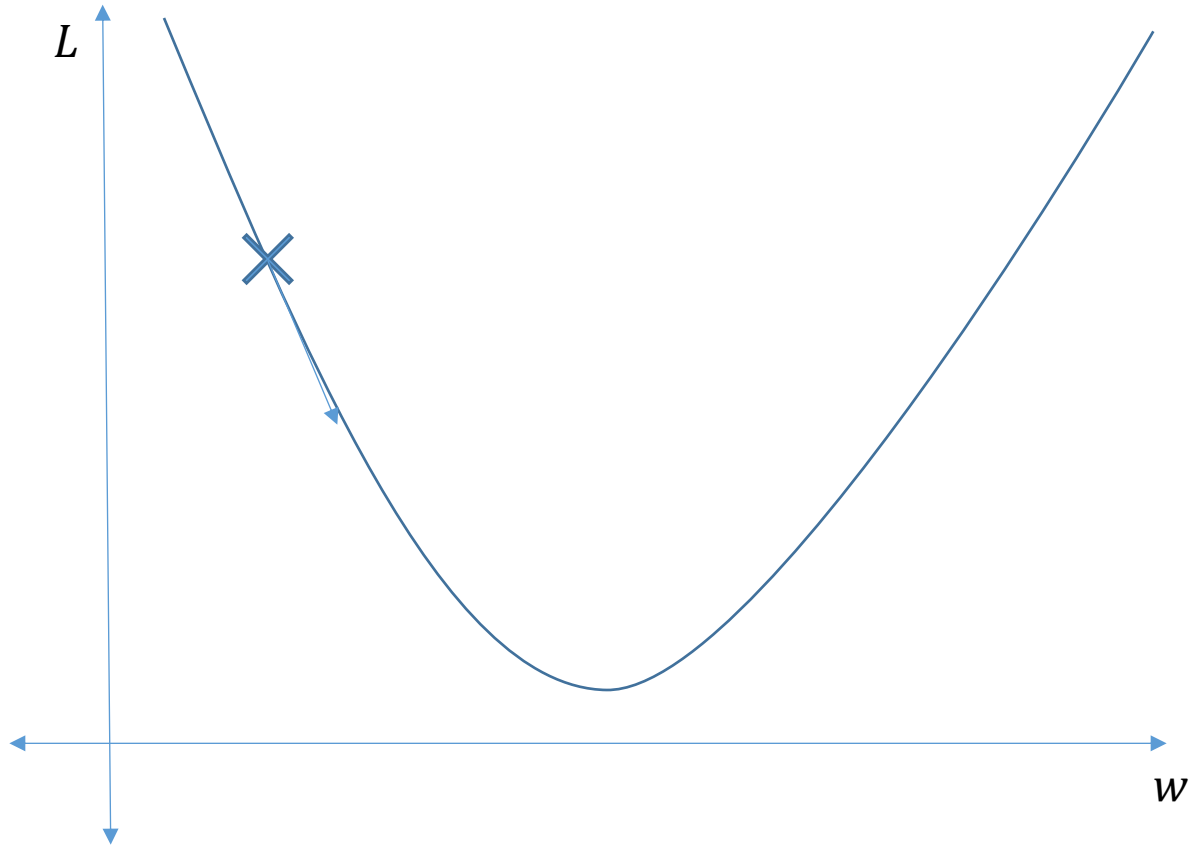
Minimizing Loss



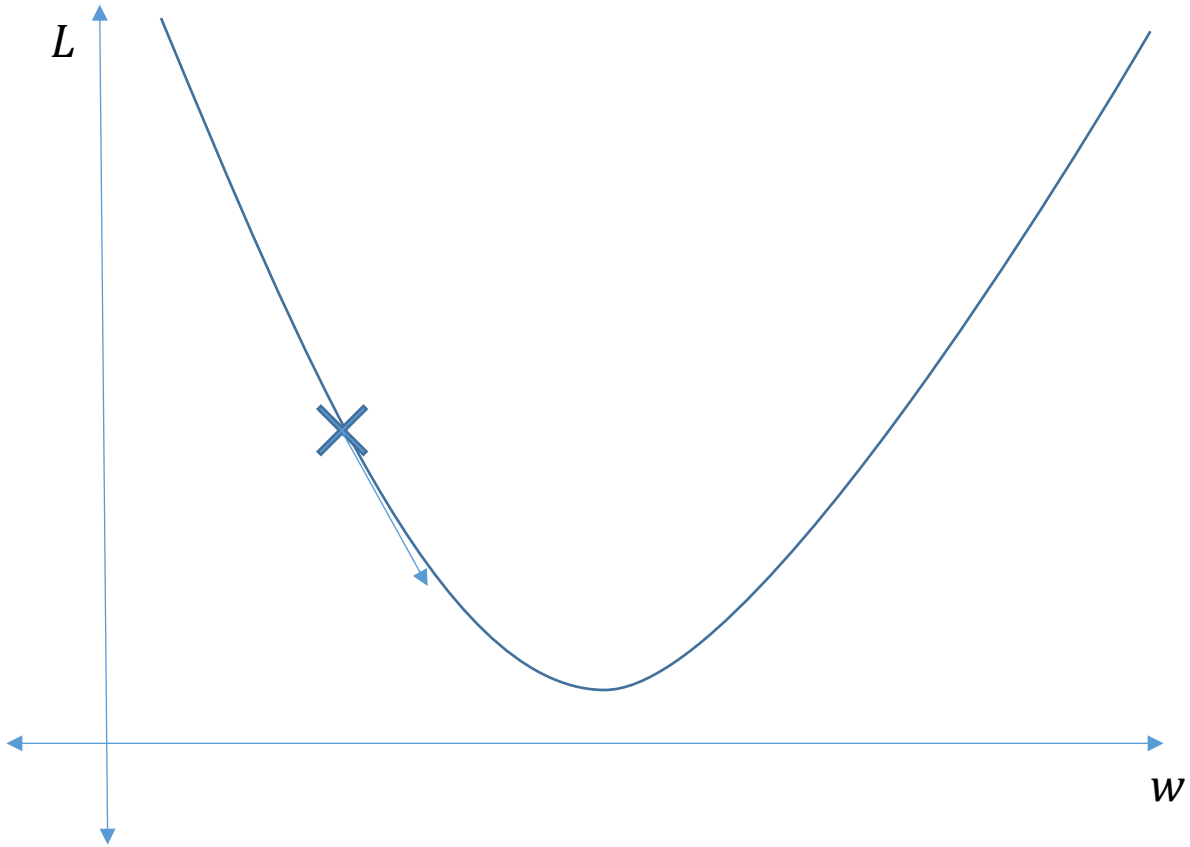
Minimizing Loss



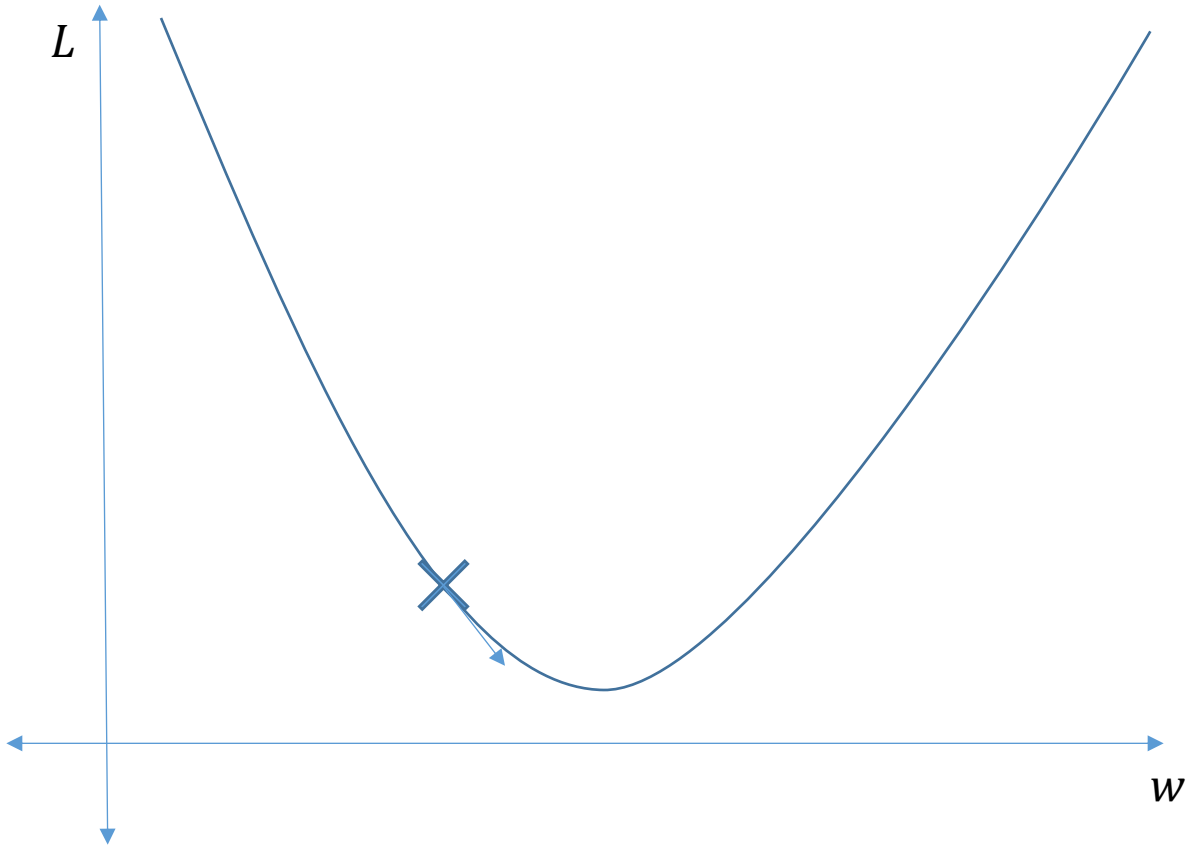
Minimizing Loss



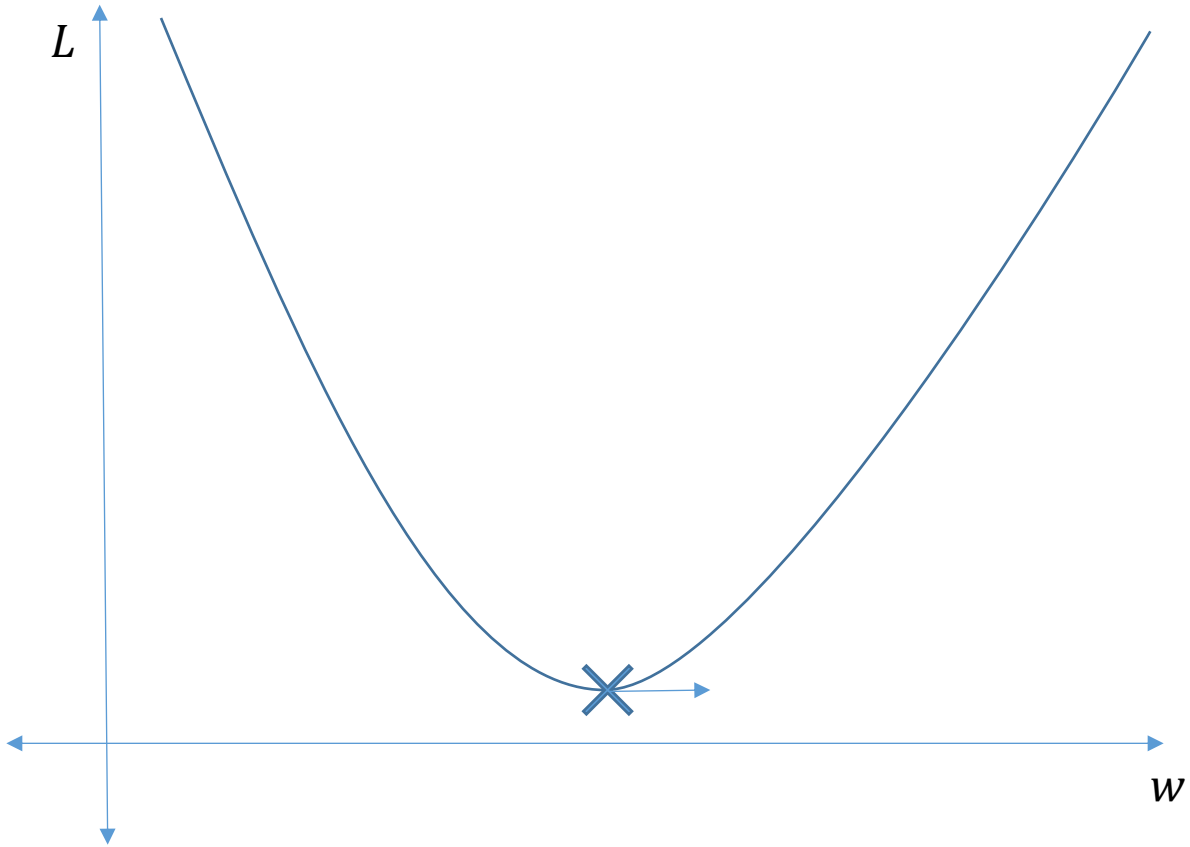
Minimizing Loss



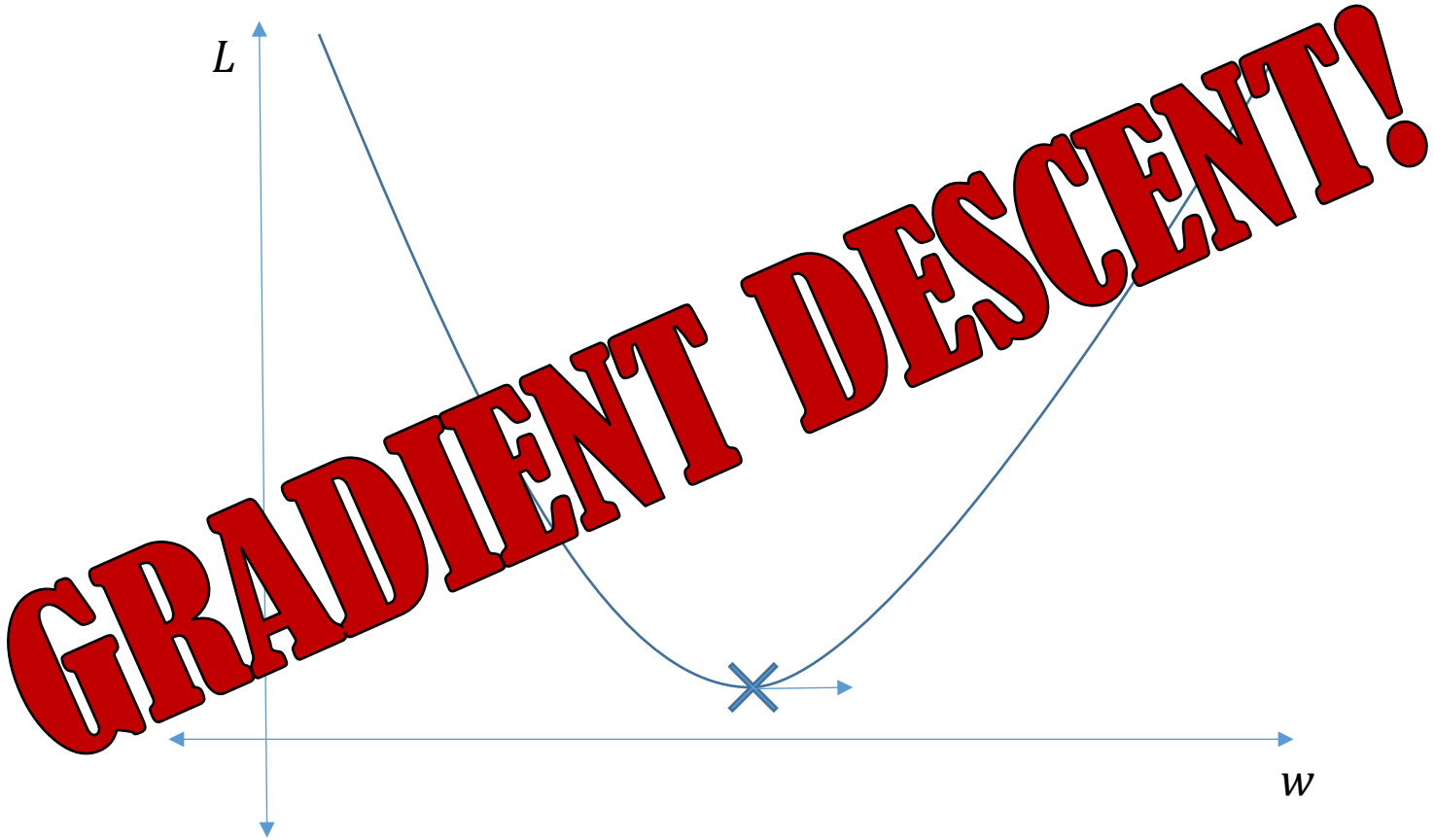
Minimizing Loss



Minimizing Loss



Minimizing Loss



Gradient Descent Pseudocode

```
for i in {0,…,num_epochs}:
    for x, y in data:
```

$$\hat{y} = SM(Wx)$$

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

$$\frac{dL}{dW} = ???$$

$$W := W - \alpha \frac{dL}{dW}$$

Getting the Gradient

$$z = Wx$$
$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

$$\frac{dL}{dW} = (SM(z) - y)(x^T)$$

https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/

Getting the Gradient

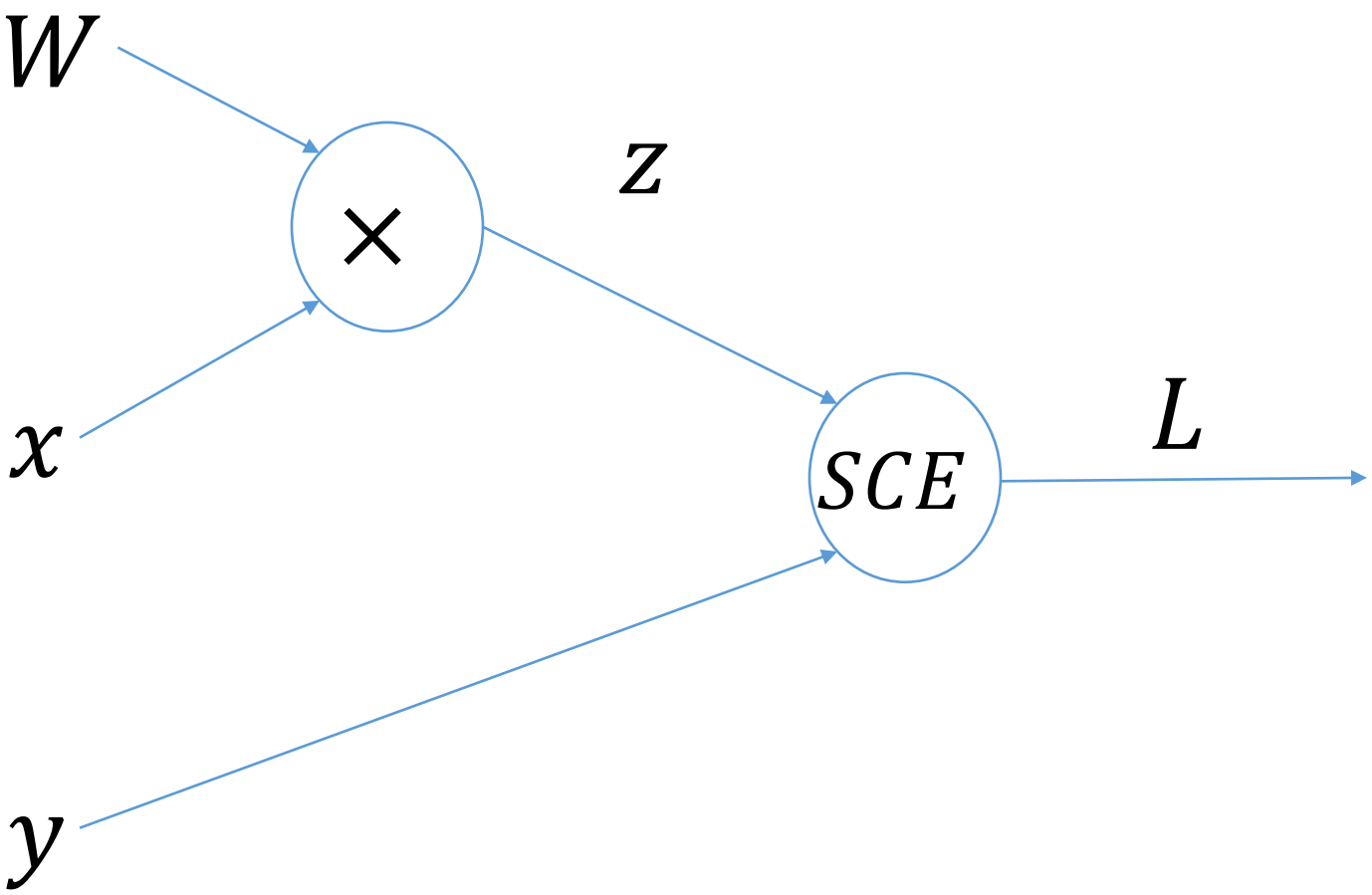
$$z = Wx$$

$$L = SCE(z, y)$$

$$\frac{dL}{dW} = (SM(z) - y)(\quad)$$

BACKPROPAGATION!

What is Backprop?

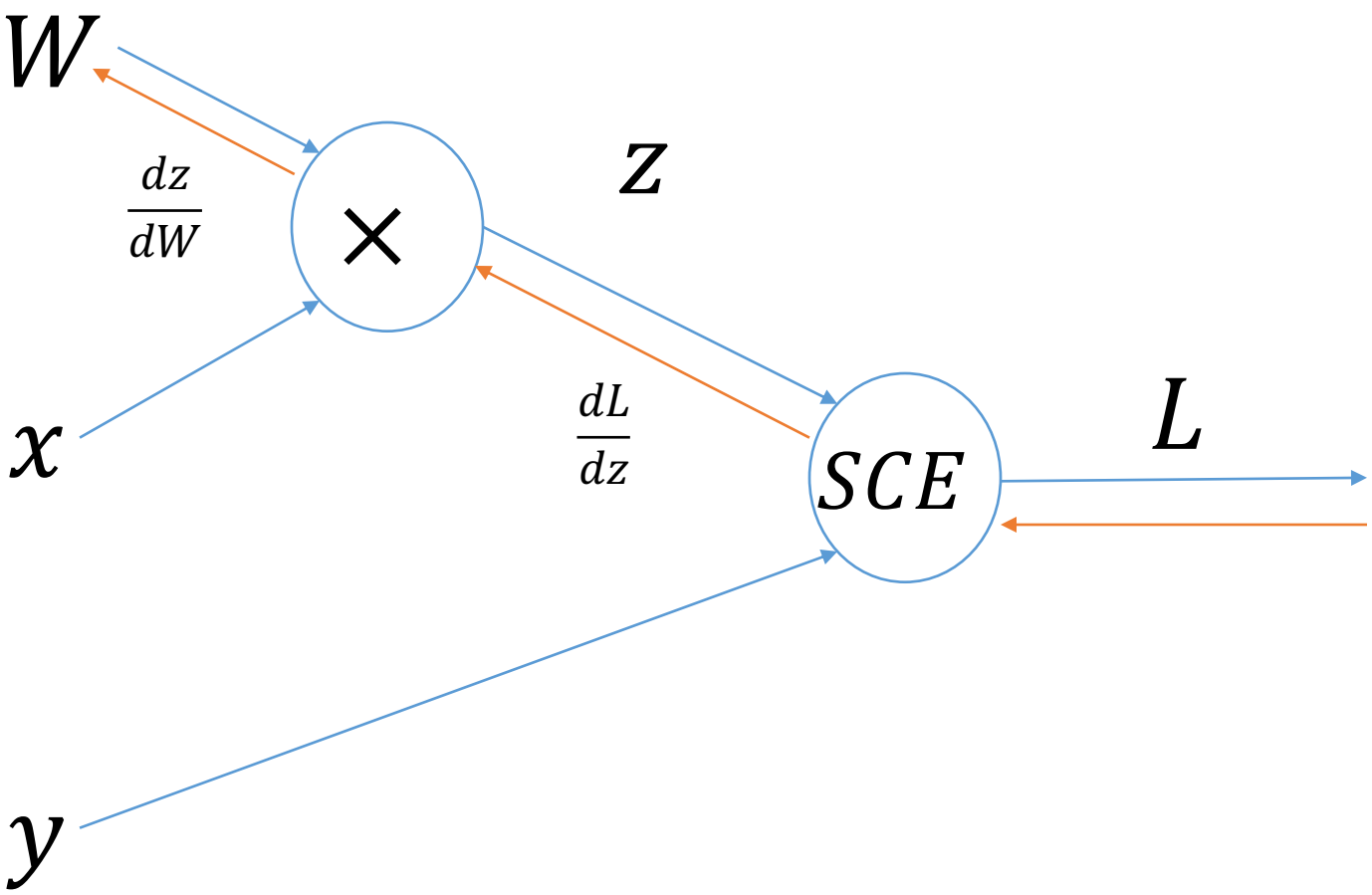


$$z = Wx$$

$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

What is Backprop?

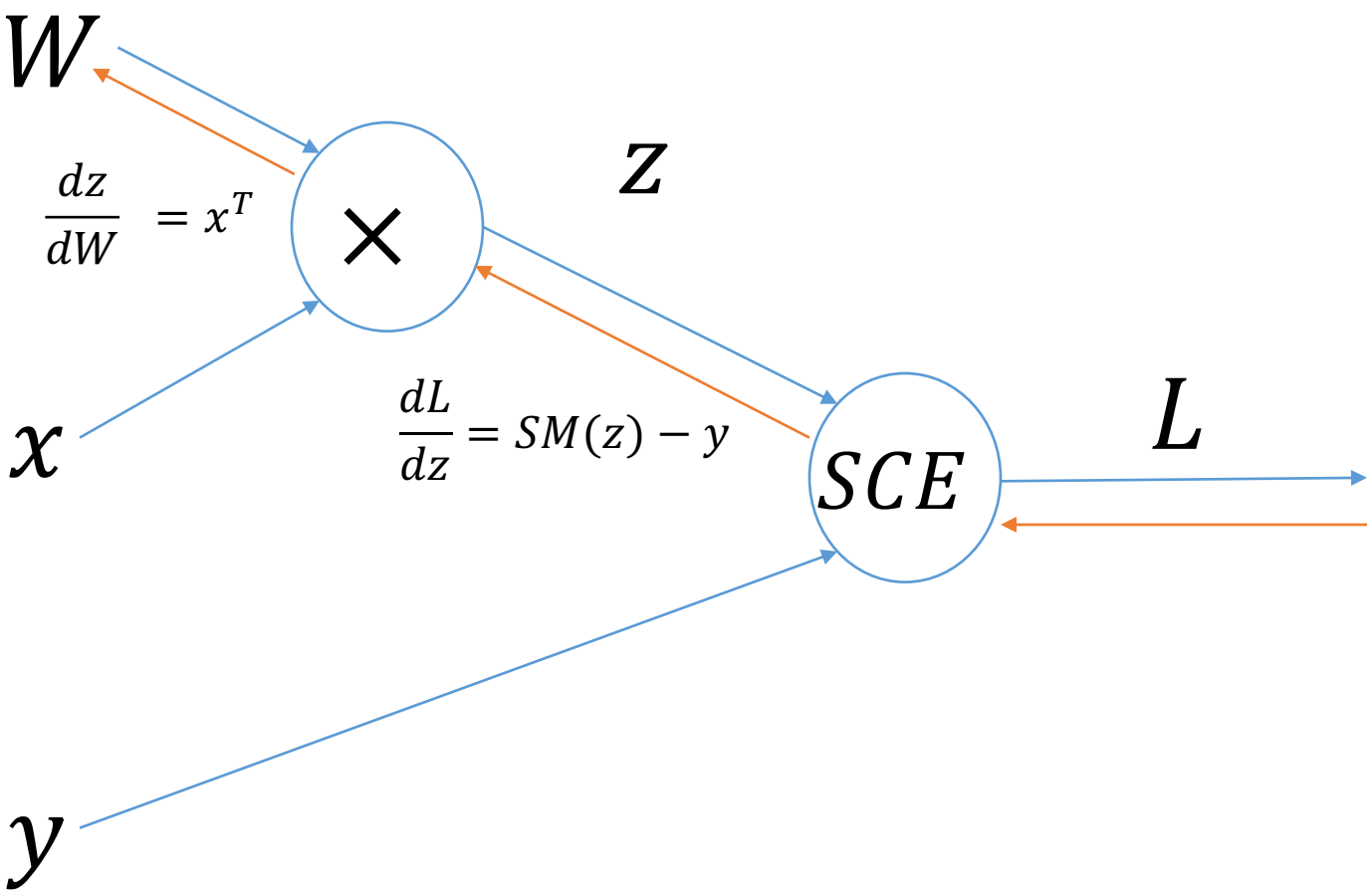


$$z = Wx$$

$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

What is Backprop?

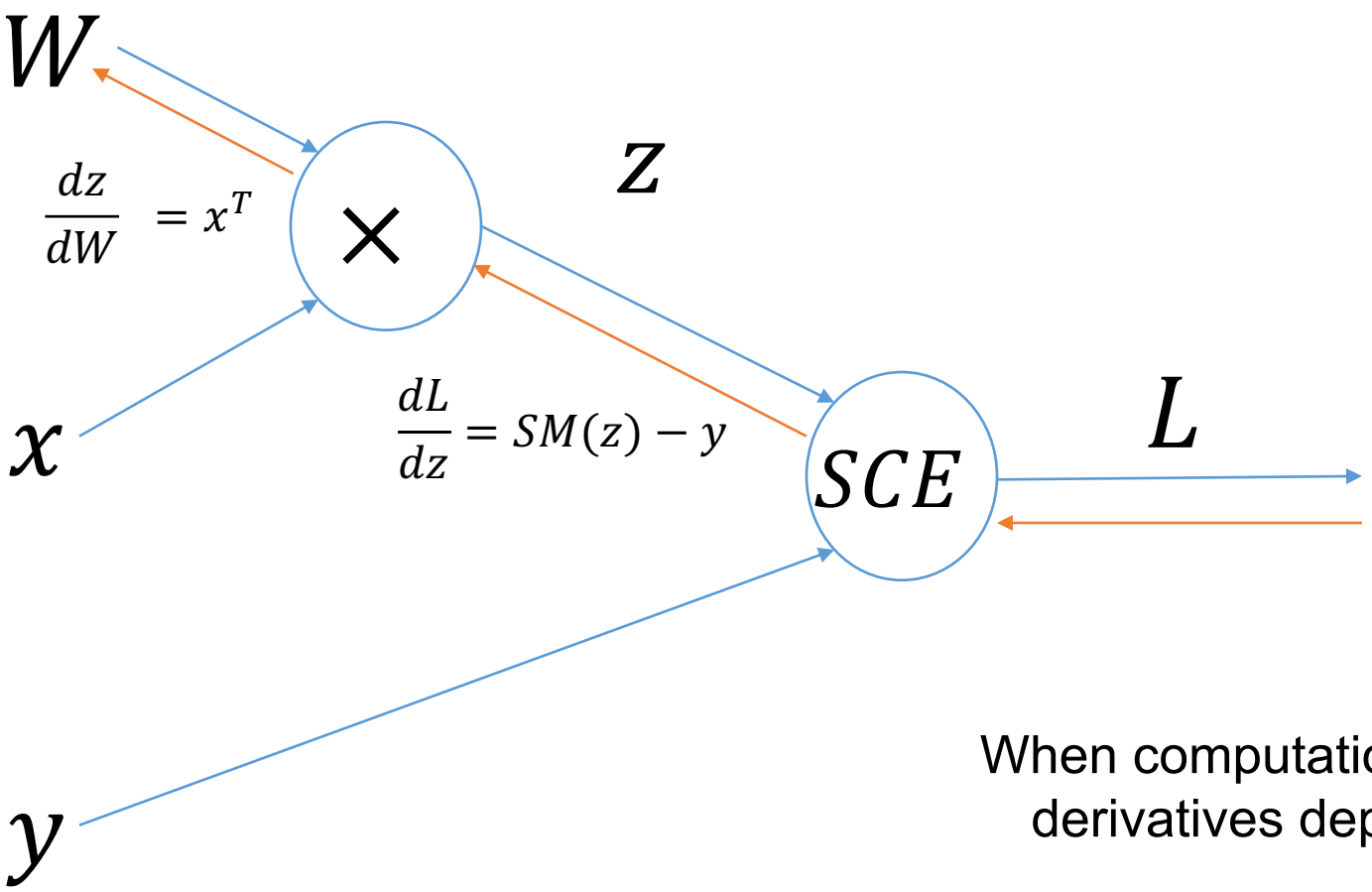


$$z = Wx$$

$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

What is Backprop?

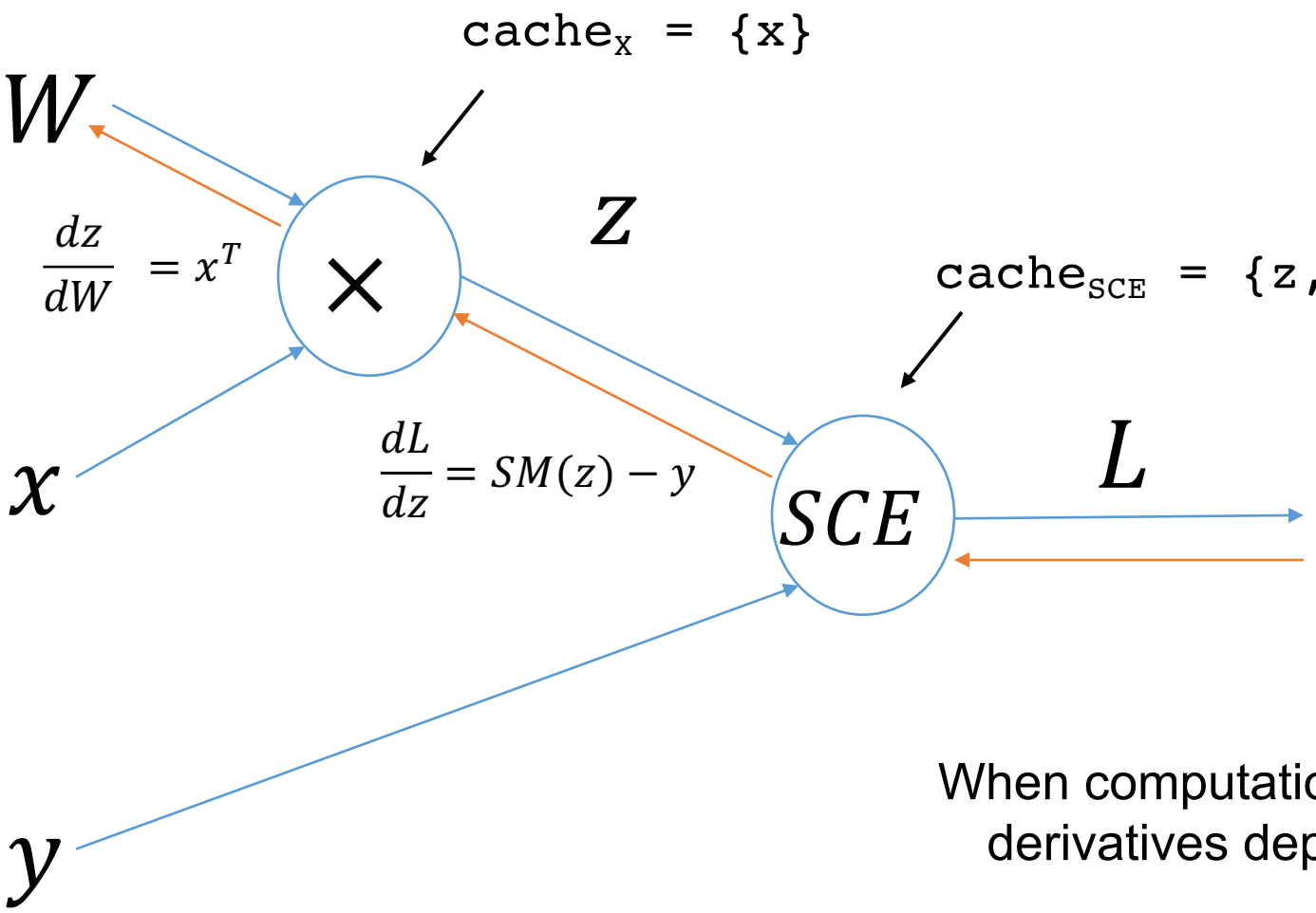


$$z = Wx$$
$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

When computations are treated as nodes, all derivatives depend only on inputs to that node.

What is Backprop?



$$z = Wx$$

$$L = SCE(z, y)$$

$$\frac{dL}{dW} = \frac{dL}{dz}\frac{dz}{dW}$$

When computations are treated as nodes, all derivatives depend only on inputs to that node.

So, we can cache the initial computation and reuse!

Backprop-Friendly Code

```
class FullyConnected:
        def __init__(self):
                self.cache = {}

        def forward(self, W, x):
                self.cache['x'] = x

                return np.dot(W, x)

        def backward(self, dout):
                x = self.cache['x']

                return
np.matmul(dout, x.T)
```

```
class SCELoss:
        def __init__(self):
                self.cache = {}

        def forward(self, z, y):
                self.cache['z'] =
z
                self.cache['y'] =
y

                return sce(z, y)

        def backward(self):
                z =
self.cache['z']
                y =
self.cache['y']
```

Gradient Descent Pseudocode (Updated)

```
for i in {0,…,num_epochs}:
    for x, y in data:
```

$$\hat{y} = SM(Wx)$$

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

$$\frac{dL}{dW} = ???$$

$$W := W - \alpha \frac{dL}{dW}$$

Gradient Descent Pseudocode (Updated)

```
for i in {0,…,num_epochs}:
    for x, y in data:
```

$$\hat{y} = SM(Wx)$$

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

$$\frac{dL}{dW} = \texttt{backprop(L)}$$

$$W := W - \alpha \frac{dL}{dW}$$

Gradient Descent Pseudocode (Updated)

```
for i in {0,…,num_epochs}:
    for x, y in data:
```

$$\hat{y} = SM(Wx)$$

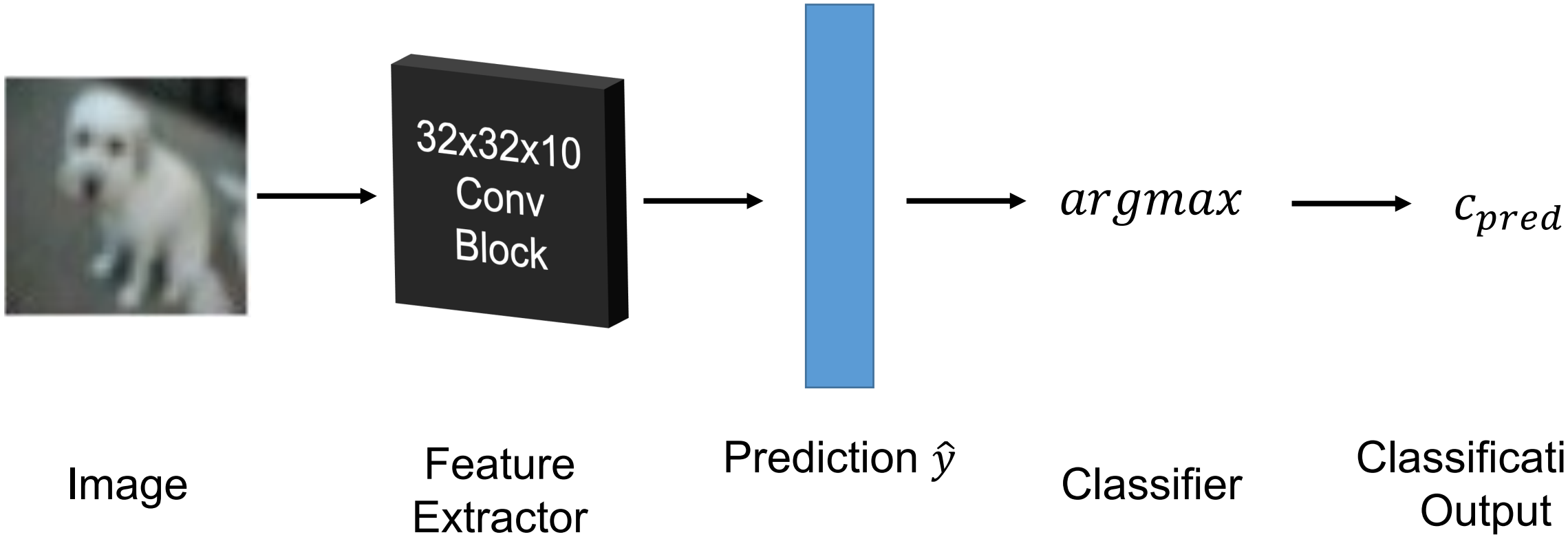
$$L = CE(\hat{y}, y)$$

$$\frac{dL}{dW} = \texttt{backprop(L)}$$

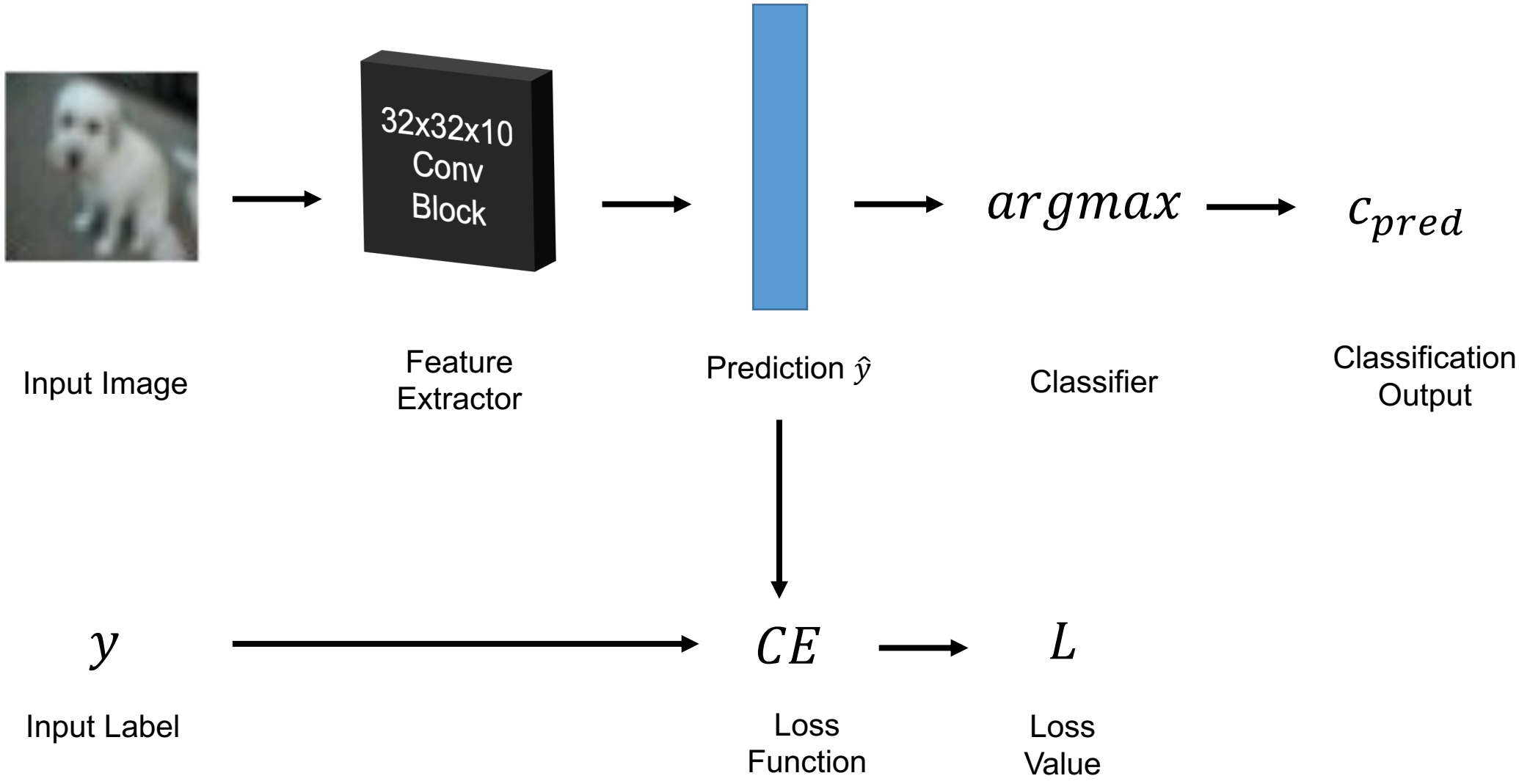
$$W := W - \alpha \frac{dL}{dW}$$

MACHINE LEARNING!

Our Classification System

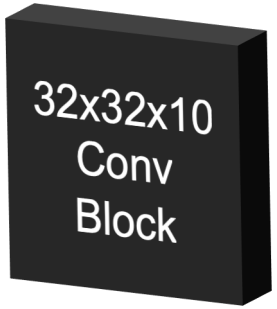


Our Classification System (modified)



Our Classification System (modified)

Input Image



Feature Extractor

Prediction $\hat{y}$

$argmax$

Classifier

c_{pred}

Classification Output

y

Input Label

CE

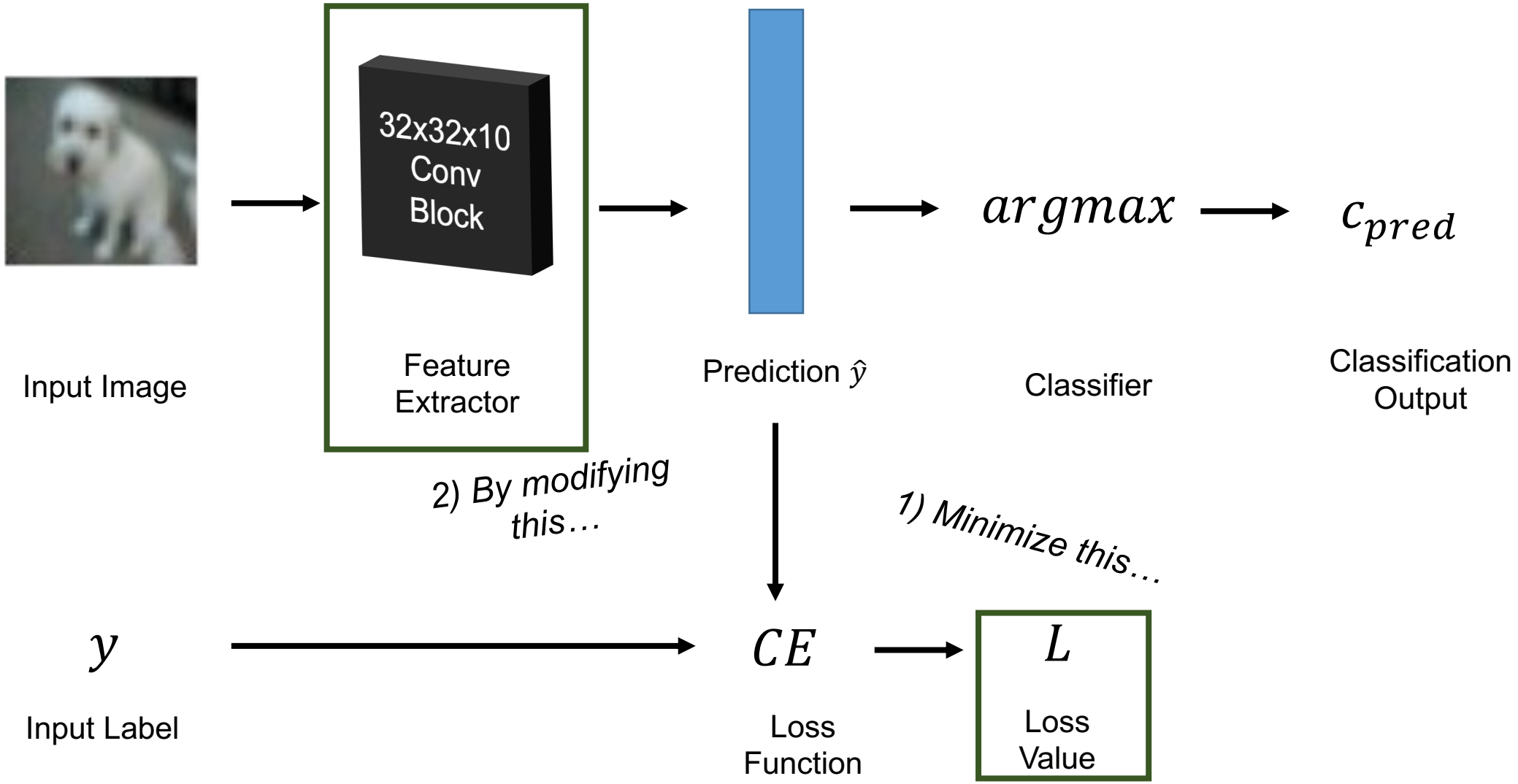
Loss Function

L

Loss Value

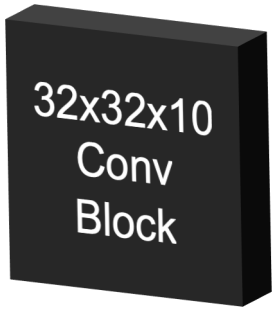
1) Minimize this…

Our Classification System (modified)



Our Classification System (modified)

Input Image



Feature Extractor

Prediction $\hat{y}$

$argmax$

Classifier

c_{pred}

Classification Output

2) By modifying this…

1) Minimize this…

y

Input Label

CE

Loss Function

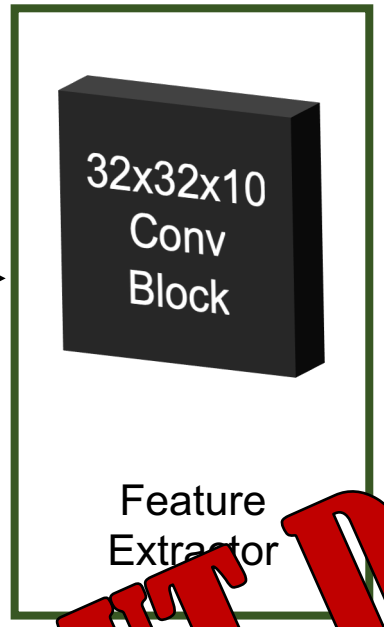
L

Loss Value

3) Using ***gradient descent!***

Our Classification System (modified)

Input Image



Feature Extractor

Prediction $\hat{y}$

Classifier

c_{pred}

Classificatio
n Output

BUT DOES IT WORK?

2) By modifying this…

1) Minimize this…

y

Input Label

CE

Loss Function

L

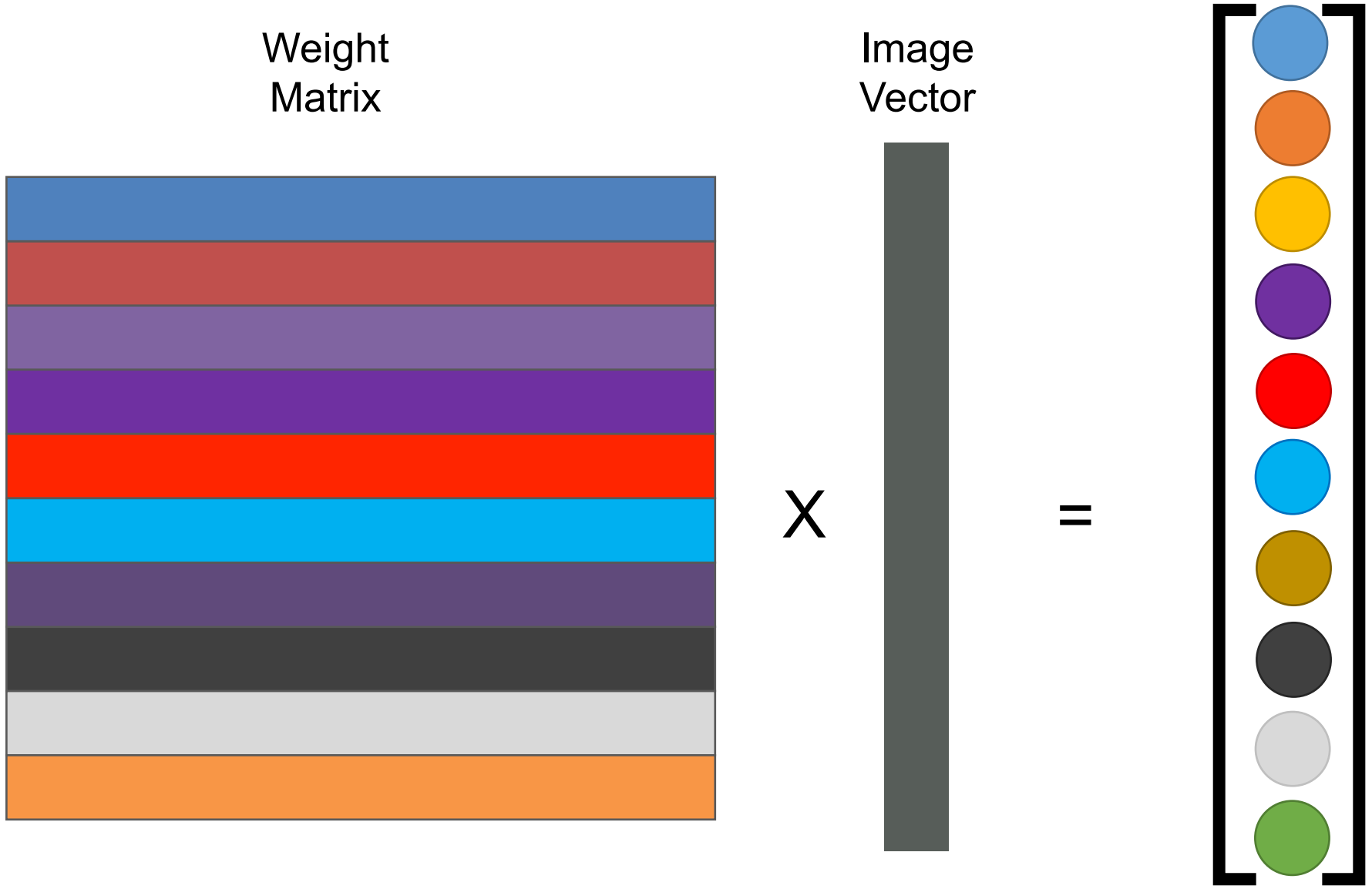
Loss Value

3) Using ***gradient descent!***

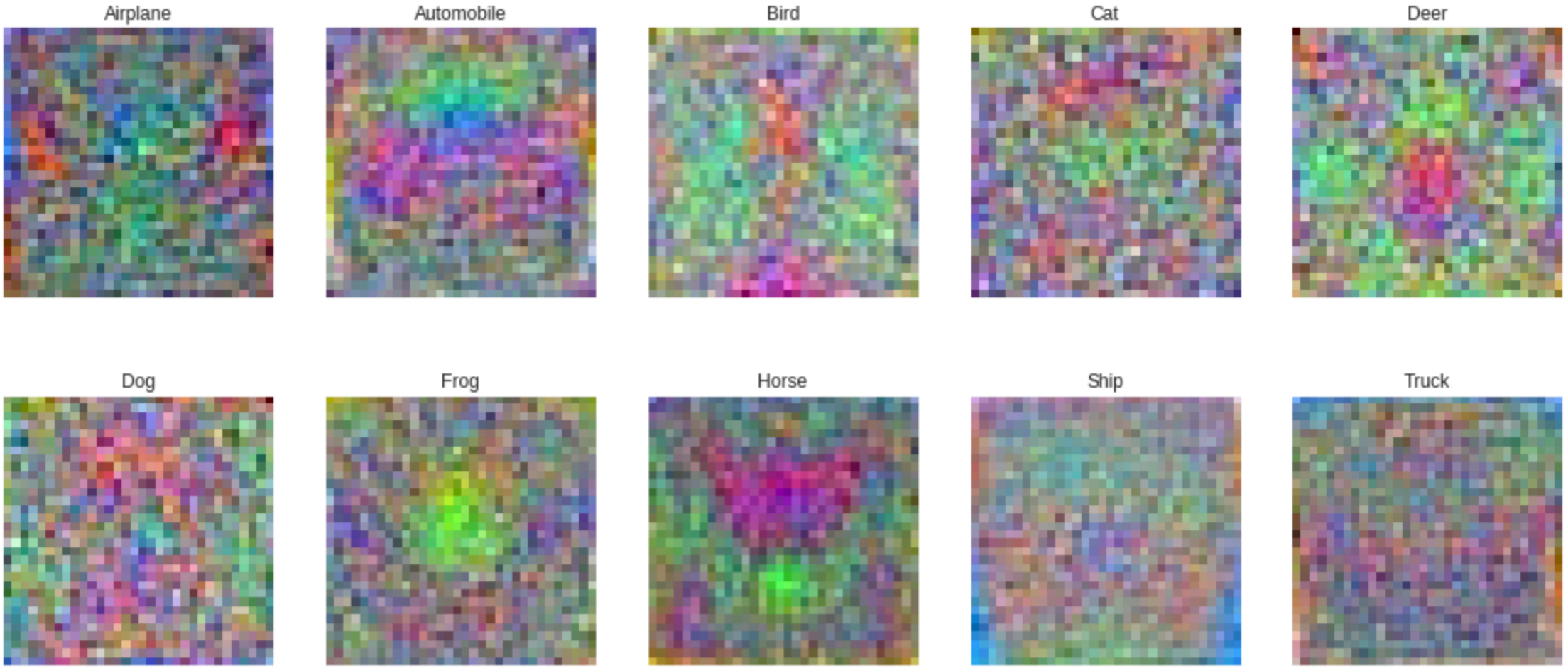
Our System's Performance

- ~40% accuracy on CIFAR-10 test
  - Best class: Truck (~60%)
  - Worst class: Horse (~16%)
- Check out the model at: **https://tinyurl.com/cifar10**
- What about the filters? What do they look like?

New Feature Extractor

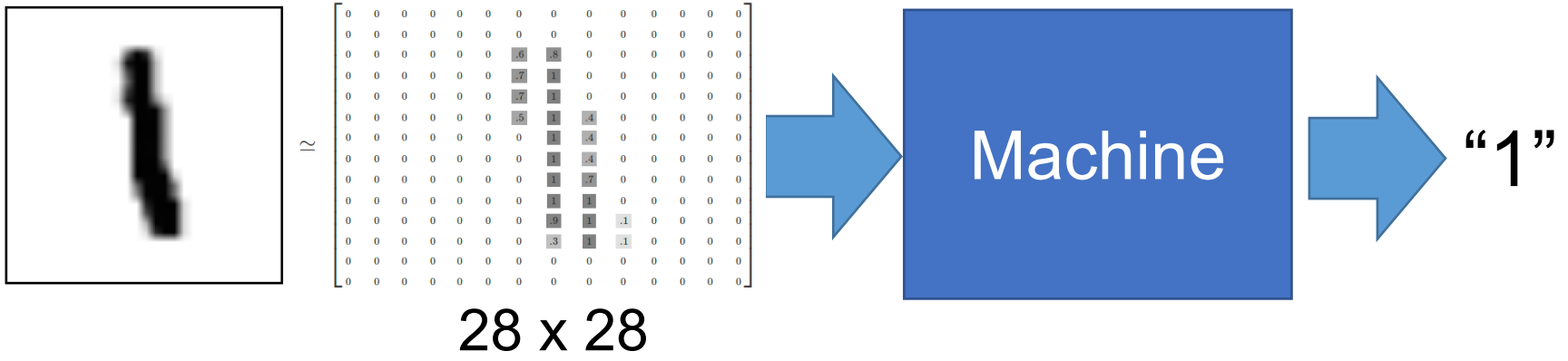


Visualizing the Filters



Example Application

- Handwriting Digit Recognition



MNIST Data: http://yann.lecun.com/exdb/mnist/

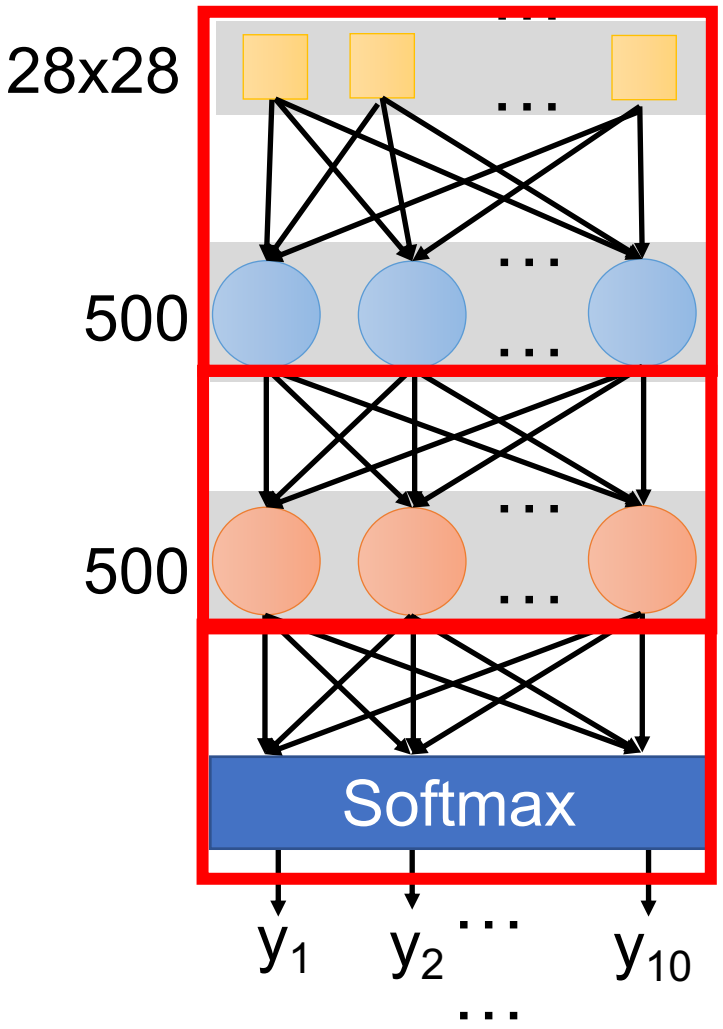
“Hello world” for deep learning

Keras

Step 1: define a set of function

Step 2: goodness of function

Step 3: pick the best function




```
model = Sequential()
```

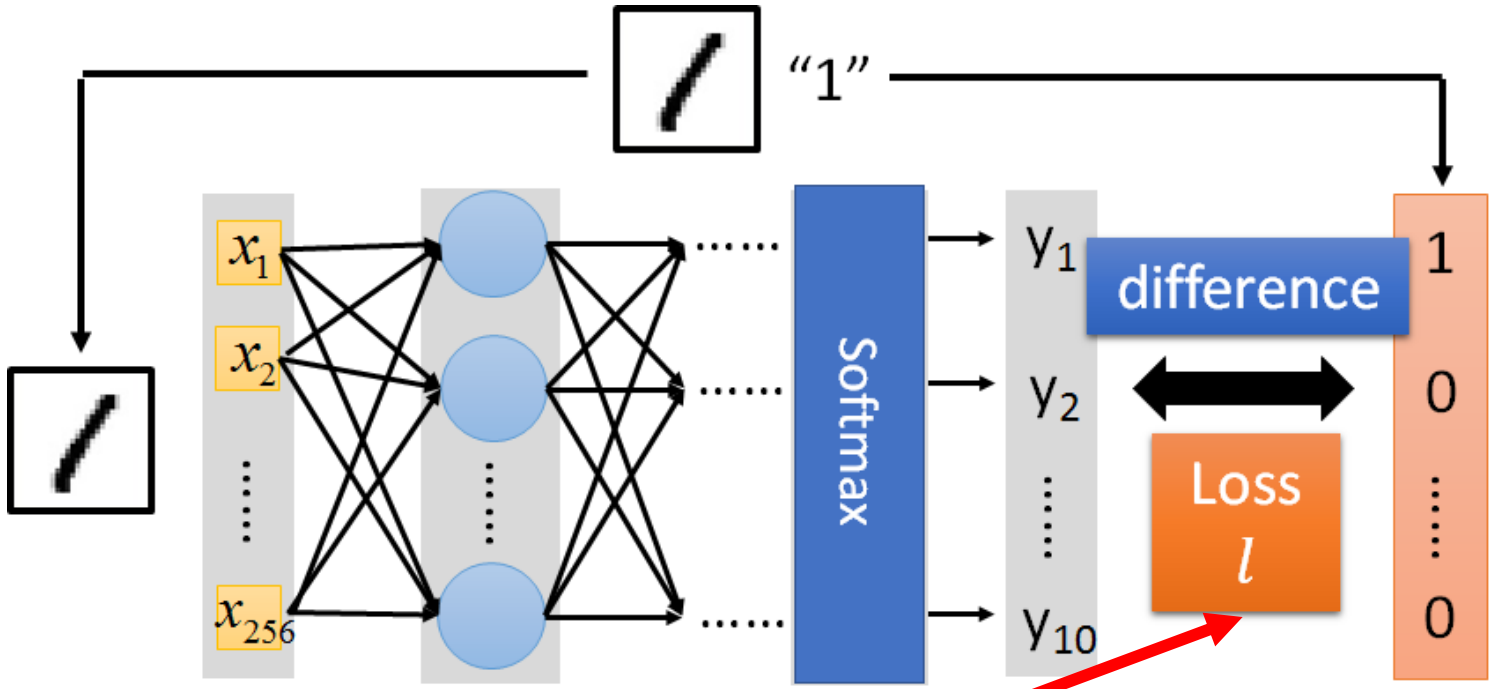
```
model.add( Dense( input_dim=28*28,
                  output_dim=500 ))
model.add( Activation('sigmoid') )
```

```
model.add( Dense( output_dim=500 ) )
model.add( Activation('sigmoid') )
```

```
model.add( Dense(output_dim=10 ) )
model.add( Activation('softmax') )
```

Keras

Step 1: define a set of function → Step 2: goodness of function → Step 3: pick the best function




```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

Keras

Step 3.1: Configuration

```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

$$w \leftarrow w - \eta \partial L / \partial w$$

0.1

Step 3.2: Find the optimal network parameters

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```

Training data (Images)

Labels (digits)

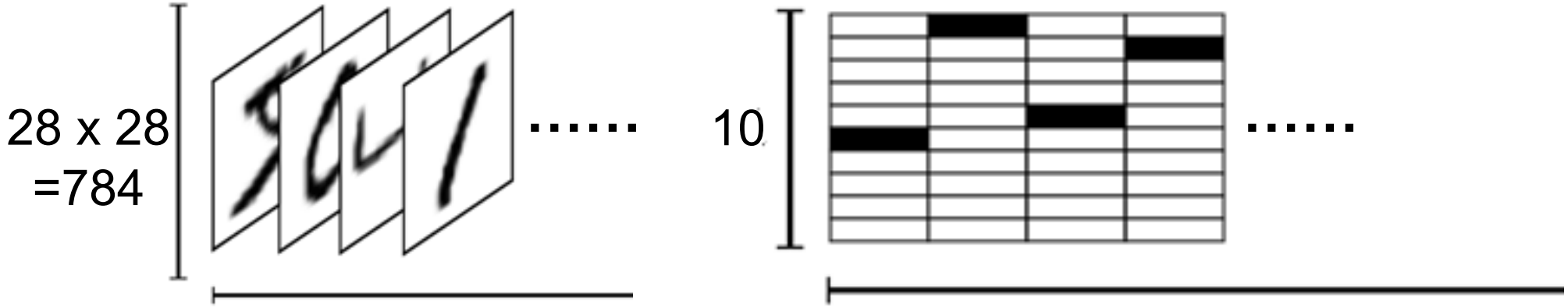
Keras

Step 3.2: Find the optimal network parameters

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```

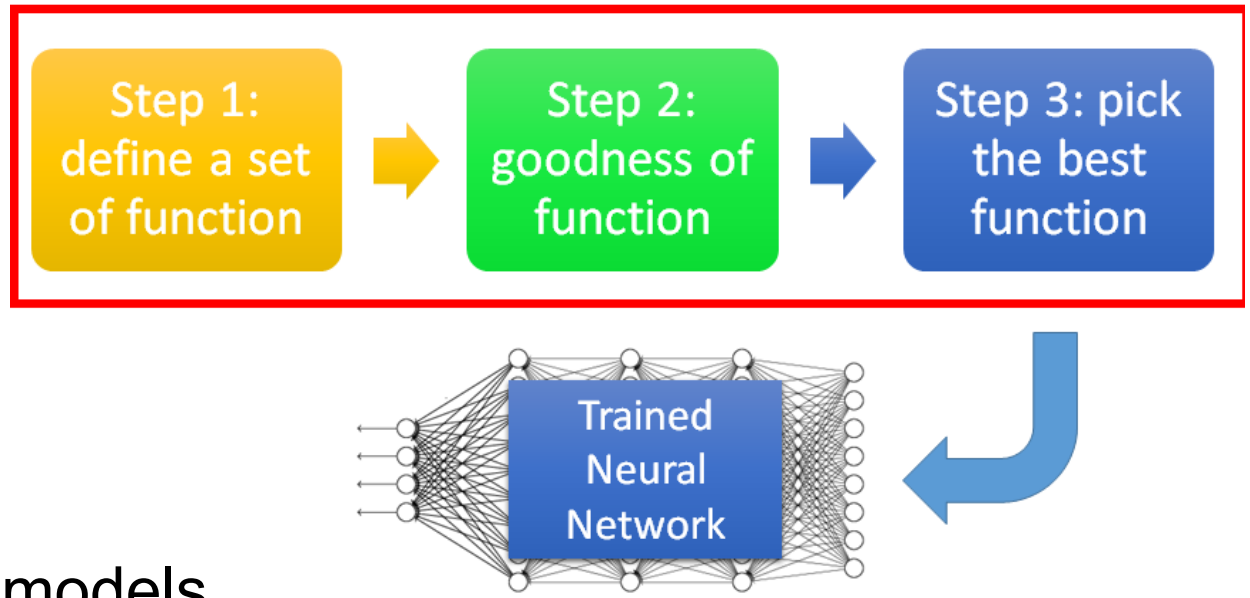
numpy array

numpy array



https://www.tensorflow.org/versions/r0.8/tutorials/mnist/beginners/index.html

Keras



Save and load models

http://keras.io/getting-started/faq/#how-can-i-save-a-keras-model

How to use the neural network (testing):

case 1:

```
score = model.evaluate(x_test,y_test)
print('Total loss on Testing Set:', score[0])
print('Accuracy of Testing Set:', score[1])
```

case 2:

```
result = model.predict(x_test)
```

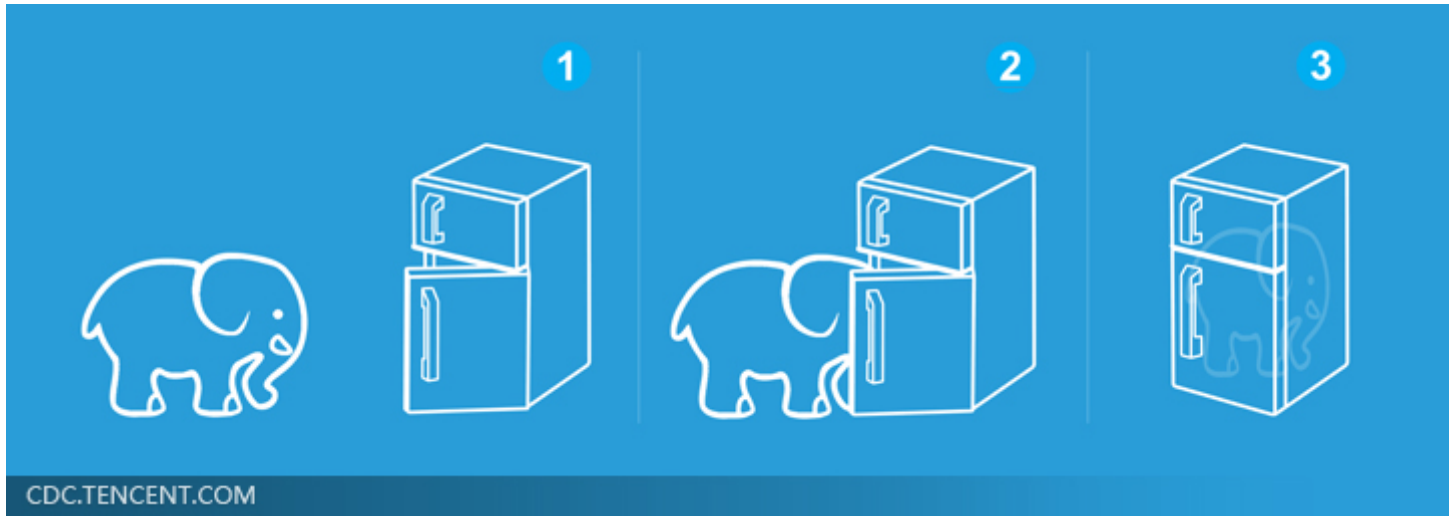
Three Steps for Deep Learning

Step 1: define a set of function

Step 2: goodness of function

Step 3: pick the best function

Deep Learning is so simple ……



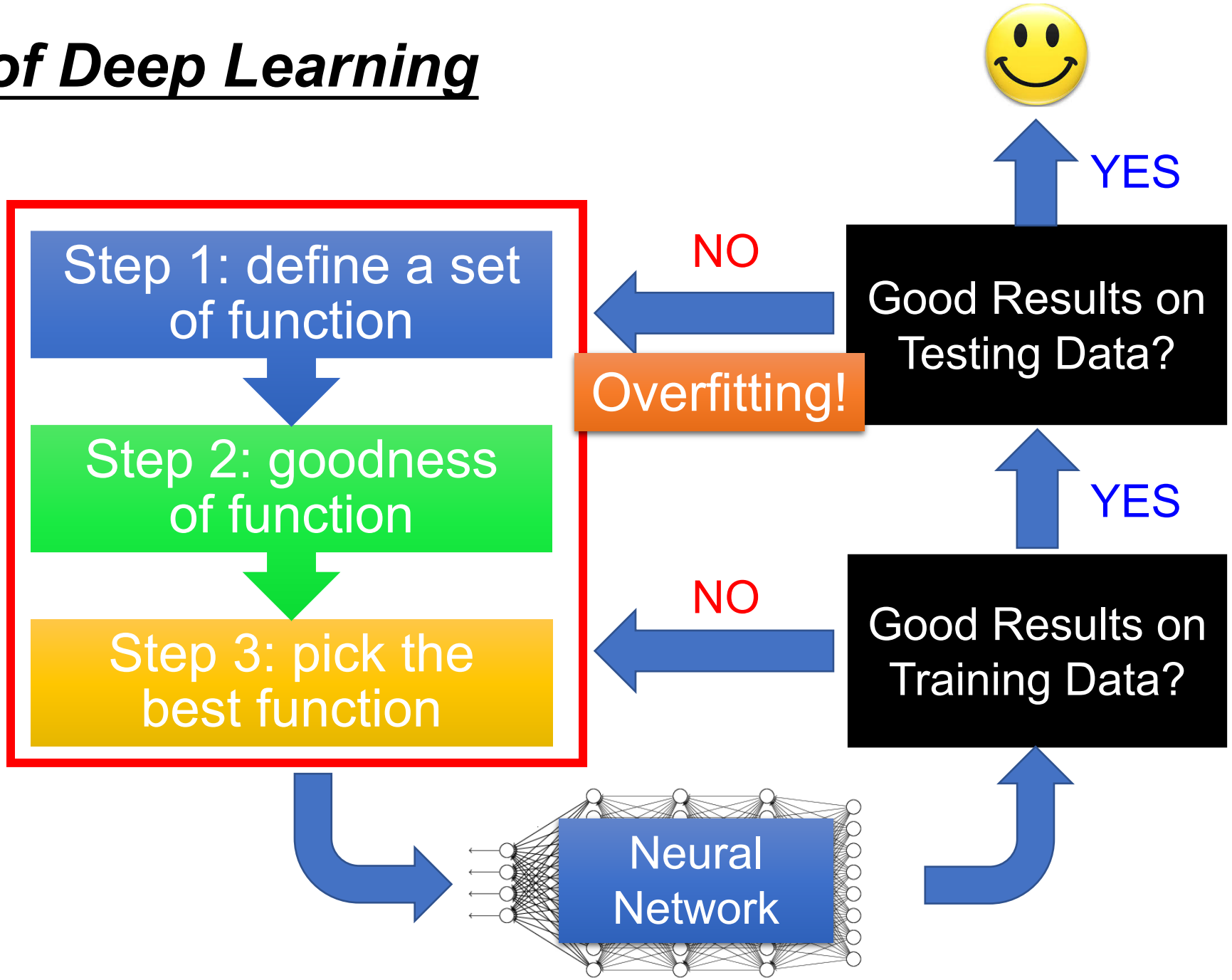
Outline

Introduction of Deep Learning

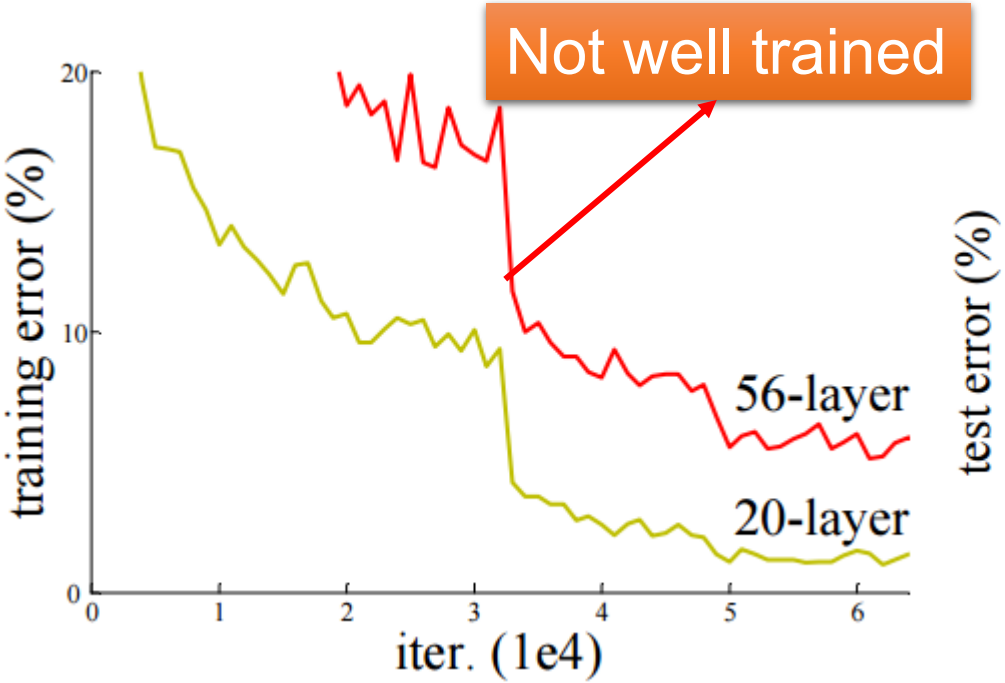
“Hello World” for Deep Learning

Tips for Deep Learning

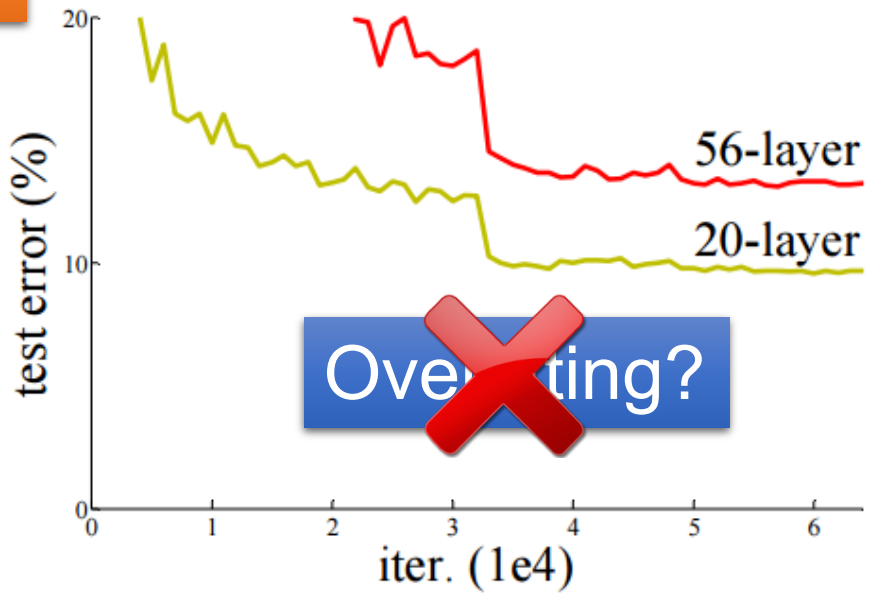
Recipe of Deep Learning



Do not always blame Overfitting



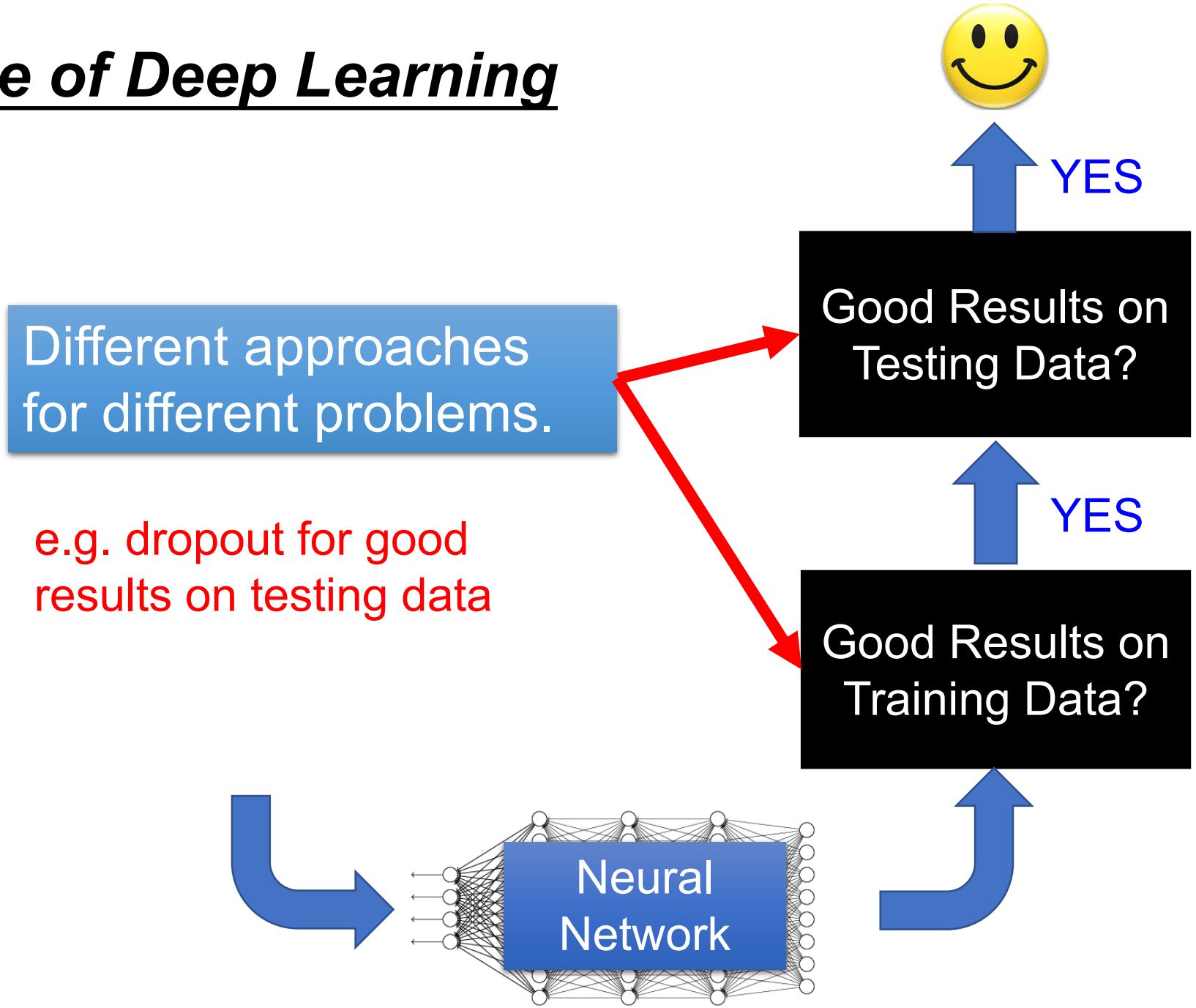
Training Data



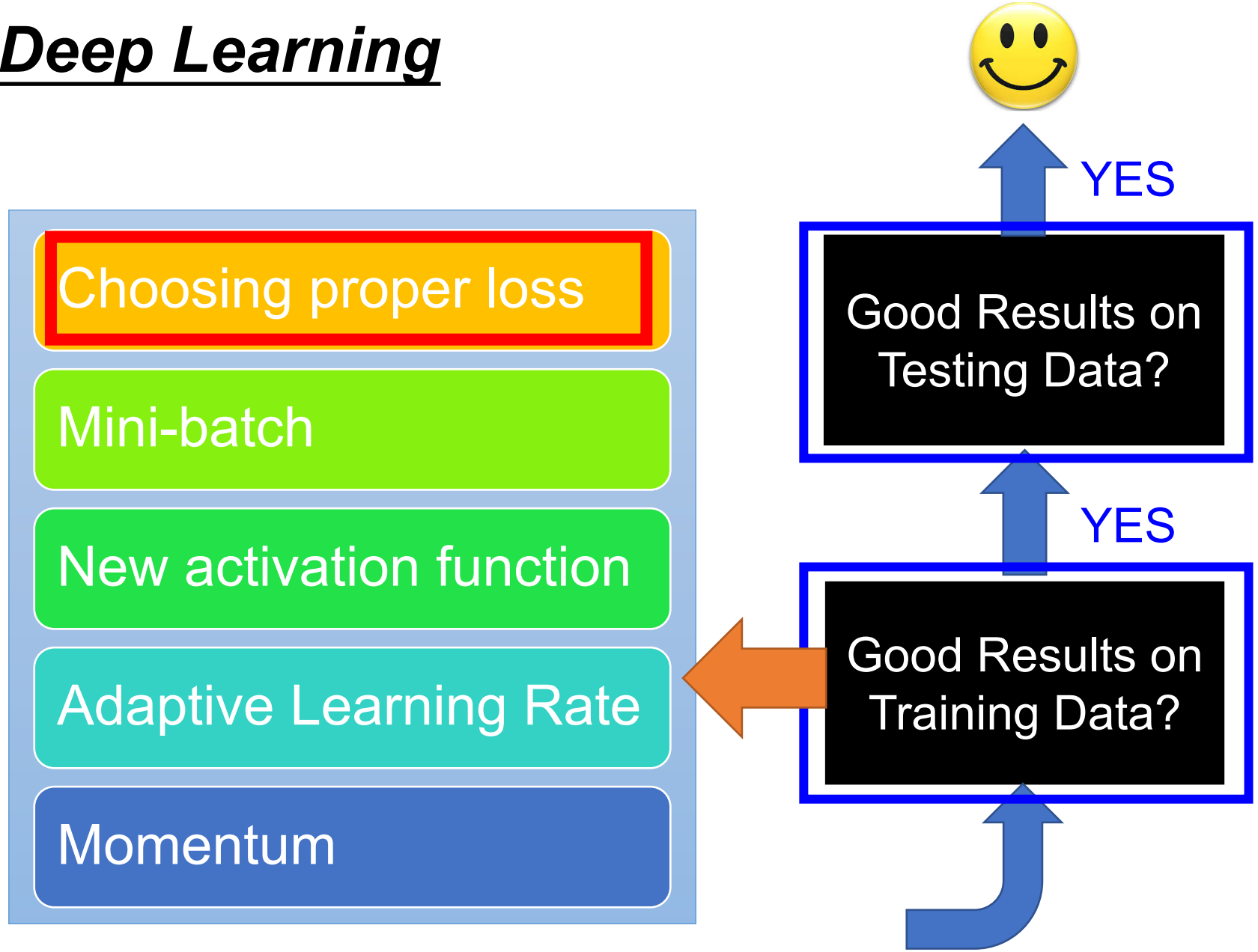
Testing Data

Deep Residual Learning for Image Recognition, 2015

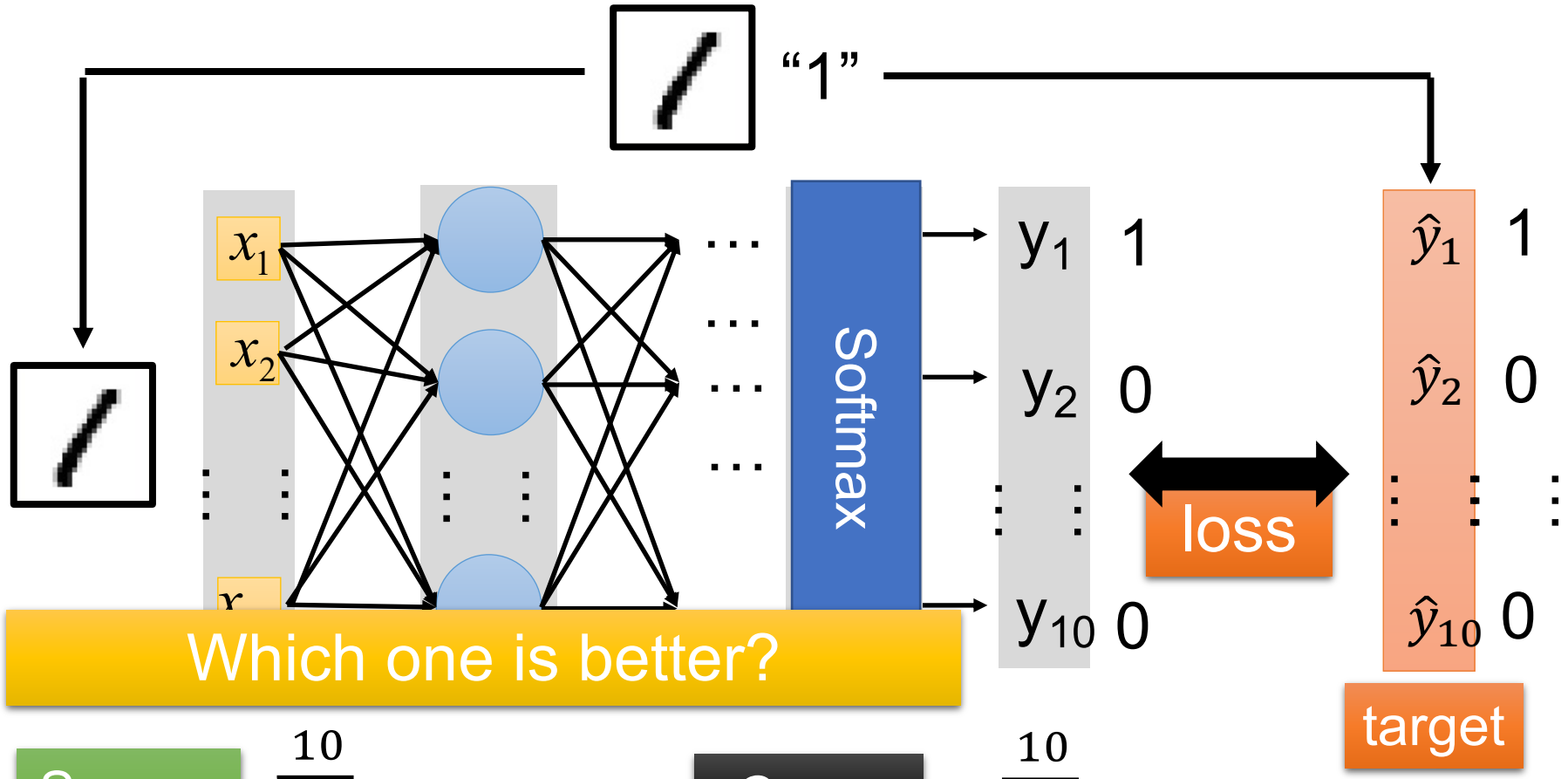
Recipe of Deep Learning



Recipe of Deep Learning



Choosing Proper Loss



Demo

Square Error

```
model.compile(loss='mse',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

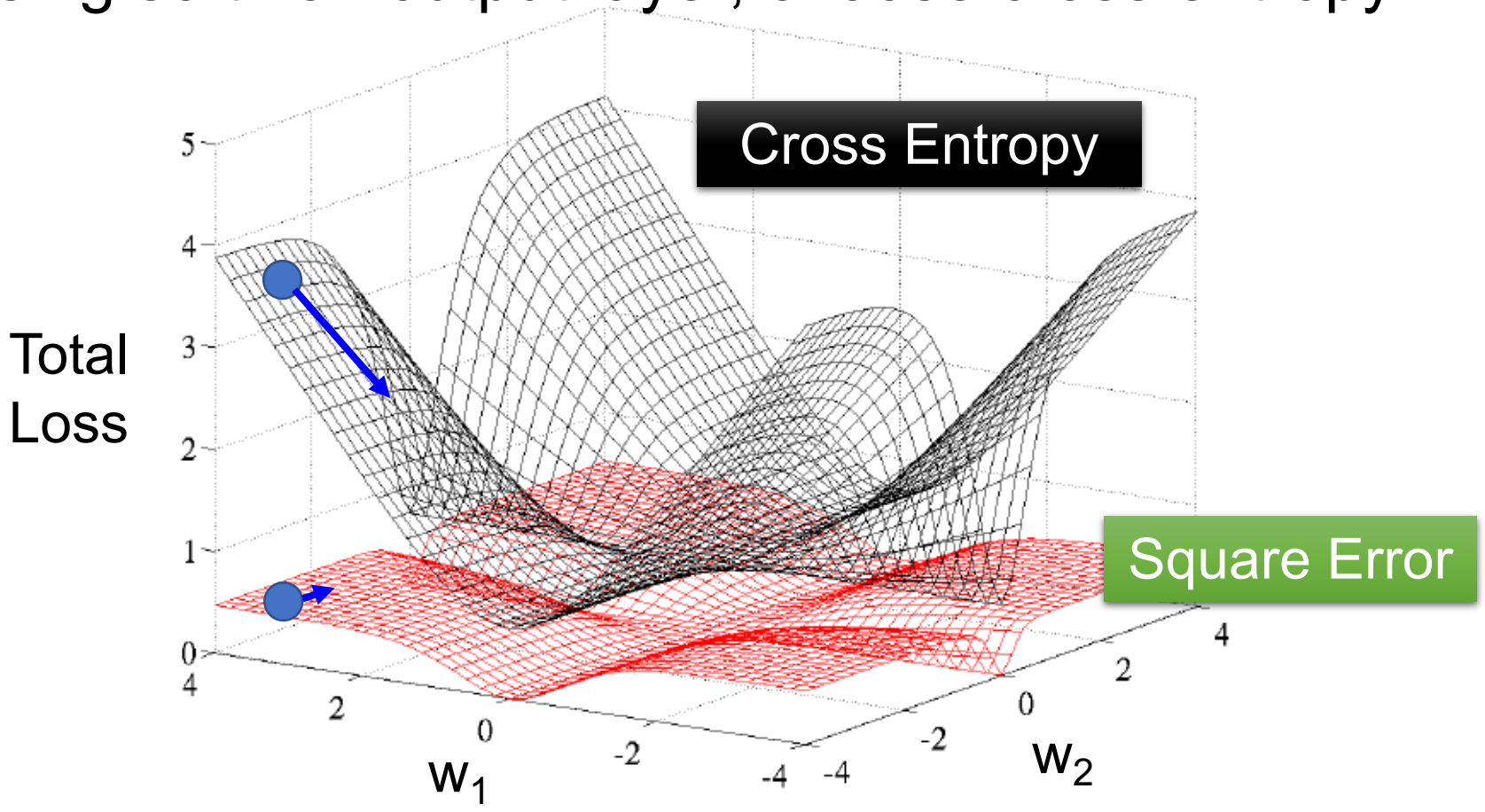
Cross Entropy

```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

Several alternatives: https://keras.io/objectives/

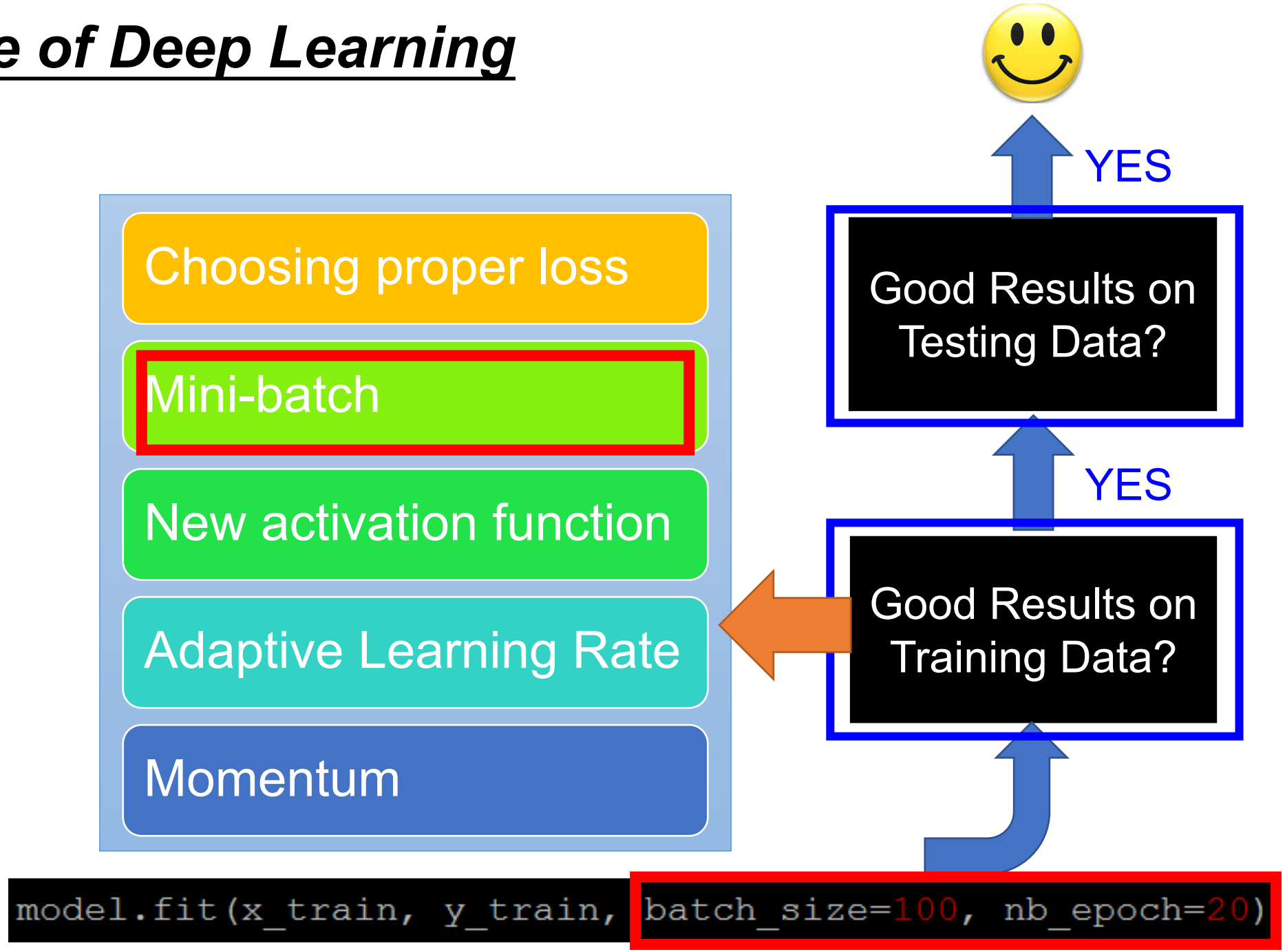
Choosing Proper Loss

When using softmax output layer, choose cross entropy



http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

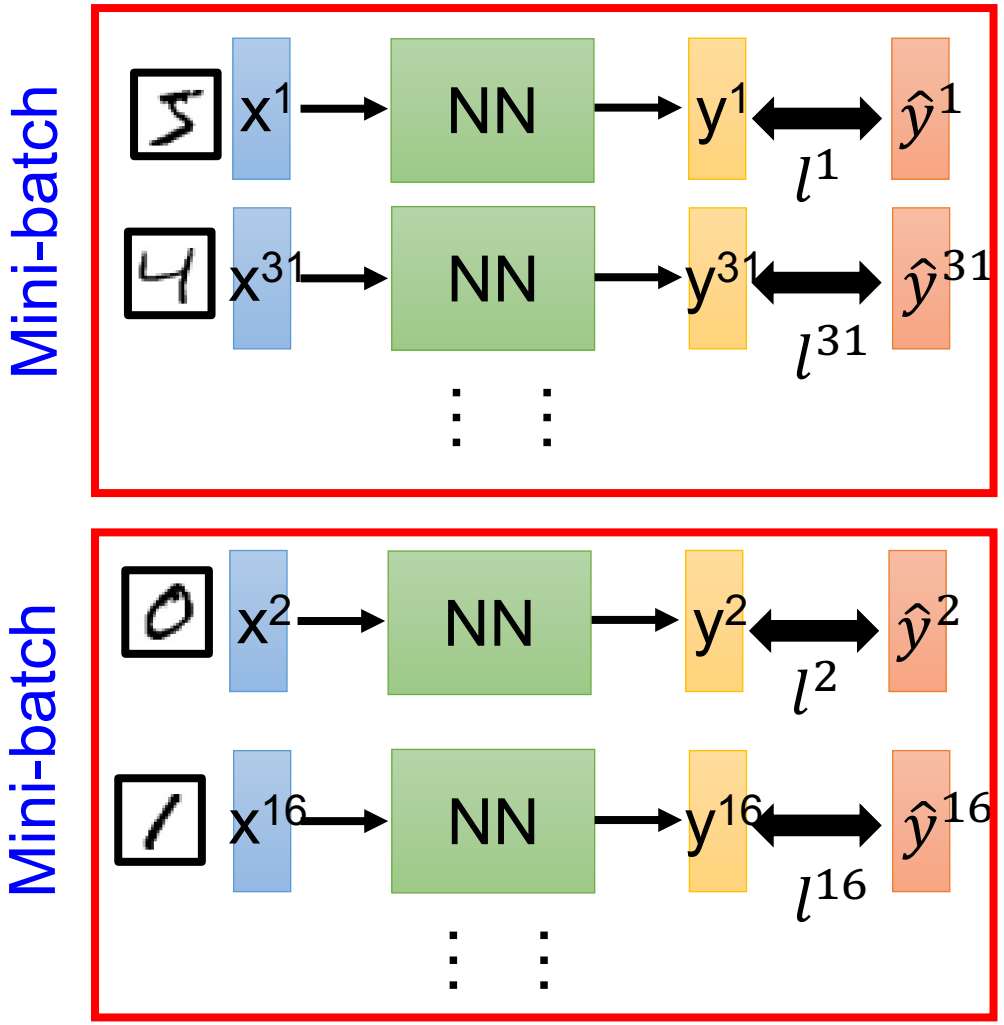
Recipe of Deep Learning



Mini-batch

We do not really minimize total loss!

- Randomly initialize network parameters



- Pick the 1st batch

  $L' = l^1 + l^{31} + \cdots$

  Update parameters once

- Pick the 2nd batch

  $L'' = l^2 + l^{16} + \cdots$

  Update parameters once

  ⋮

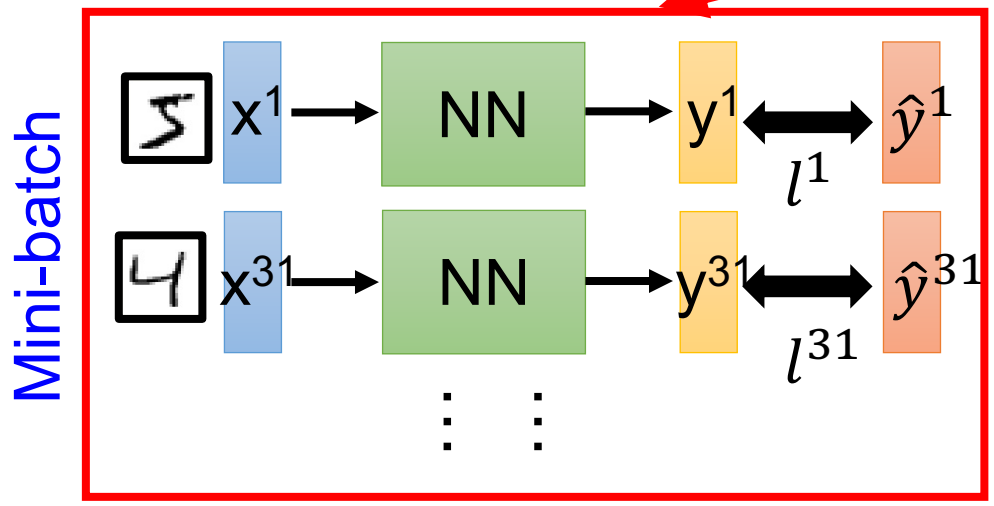
- Until all mini-batches have been picked

one epoch

Repeat the above process

Mini-batch

```
model.fit(x_train, y_train, batch_size=100, nb_epoch=20)
```



100 examples in a mini-batch

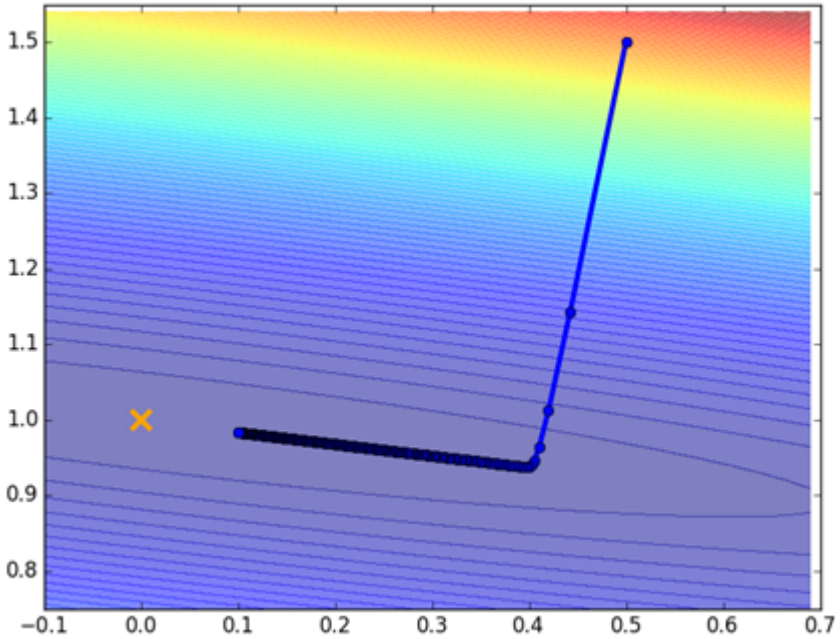
- Pick the 1st batch
  $L' = l^1 + l^{31} + \cdots$
  Update parameters once
- Pick the 2nd batch
  $L'' = l^2 + l^{16} + \cdots$
  Update parameters once
- ⋮
- Until all mini-batches have been picked

one epoch

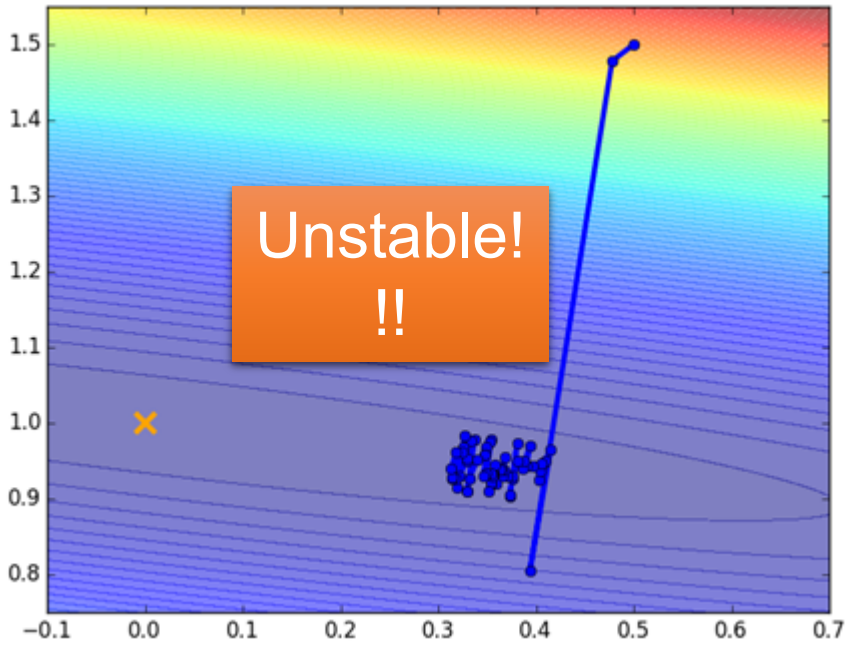
Repeat 20 times

Mini-batch

Original Gradient Descent



With Mini-batch



The colors represent the total loss.

Mini-batch is Faster

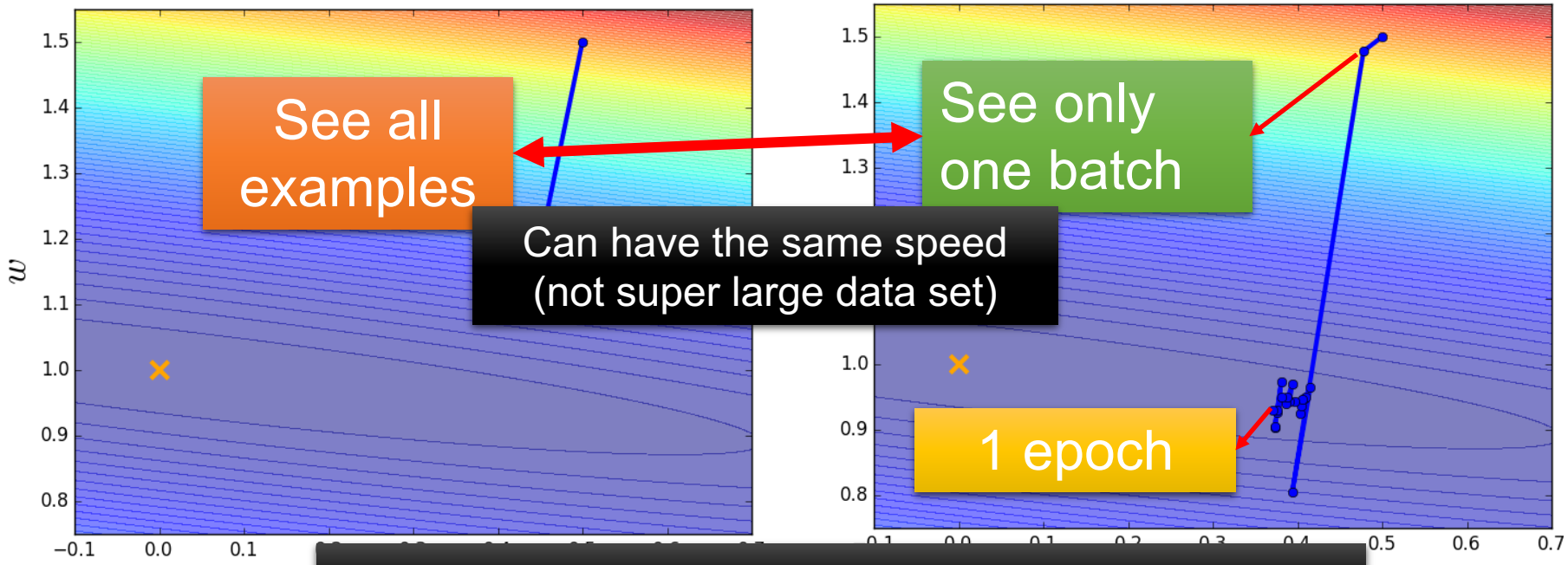
Not always true with parallel computing.

Original Gradient Descent

Update after seeing all examples

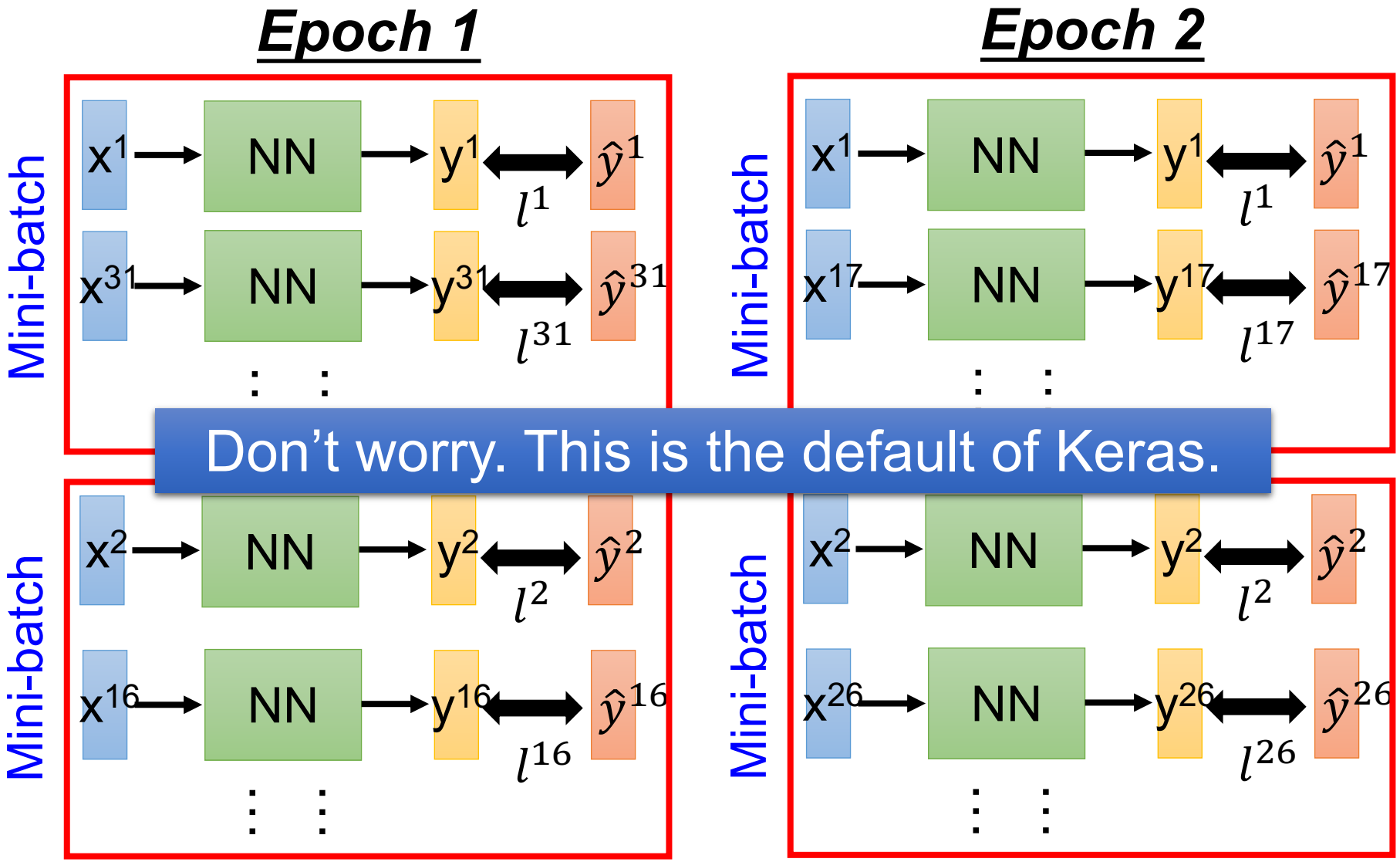
With Mini-batch

If there are 20 batches, update 20 times in one epoch.

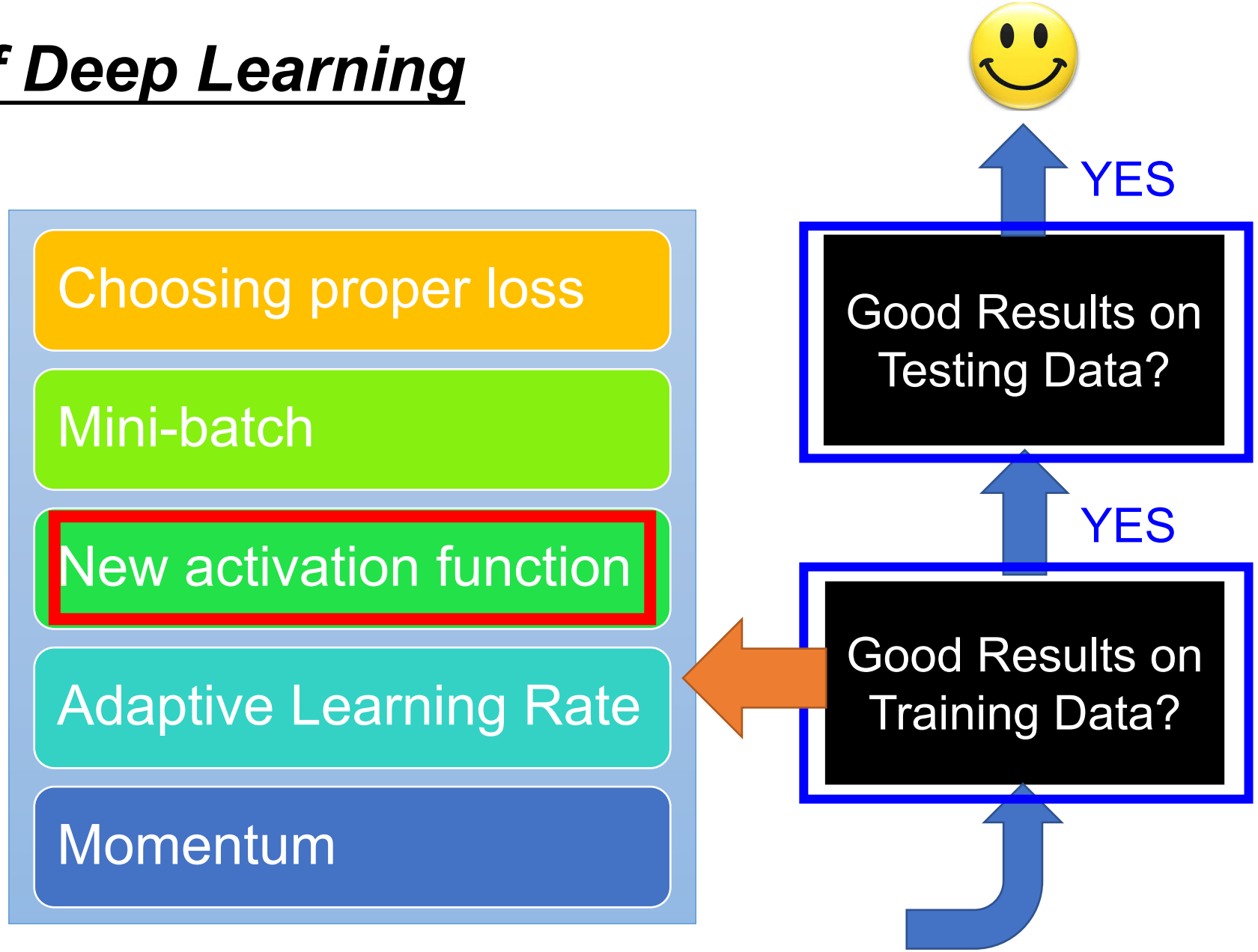


Mini-batch has better performance!

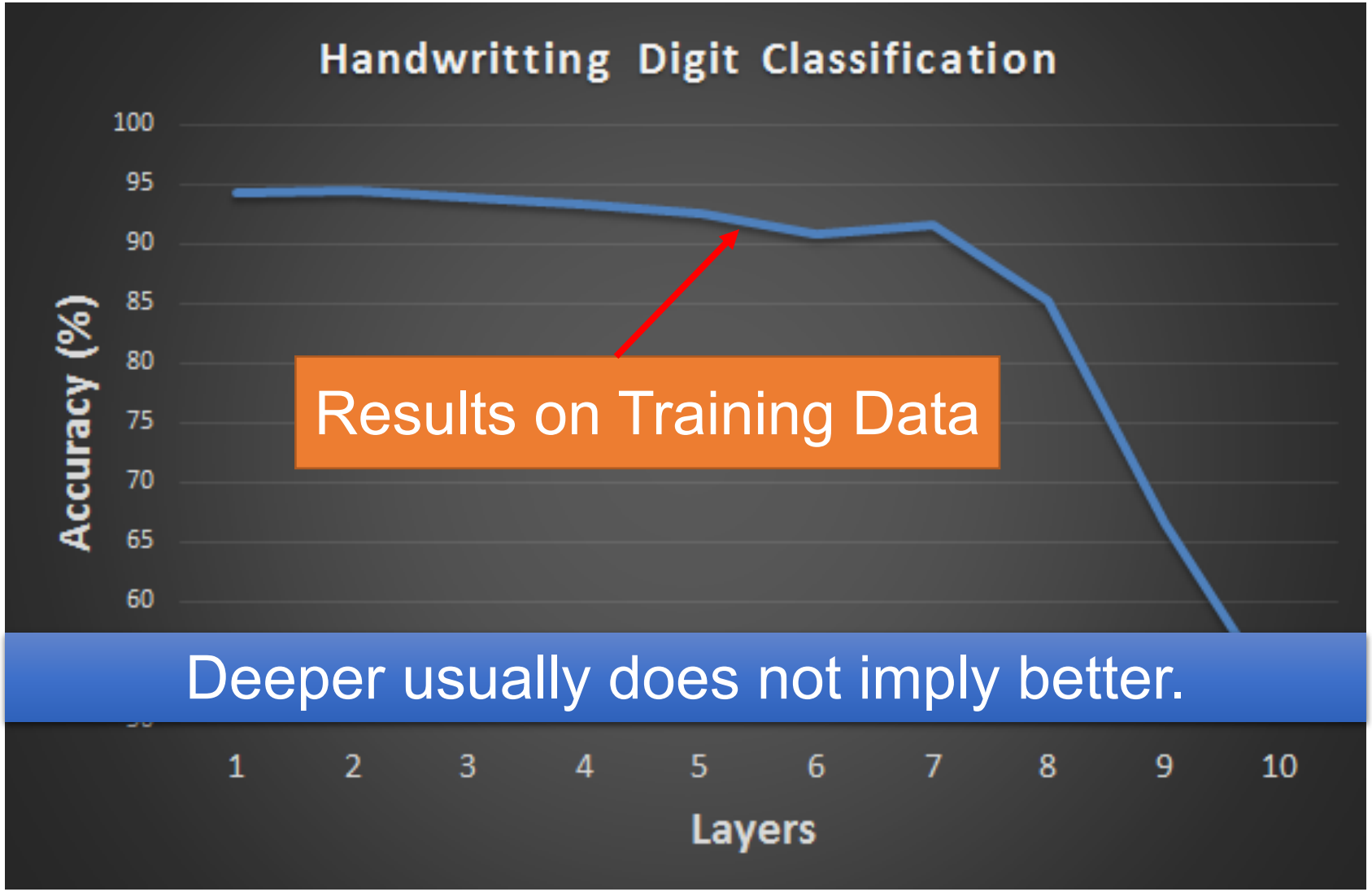
Shuffle the training examples for each epoch



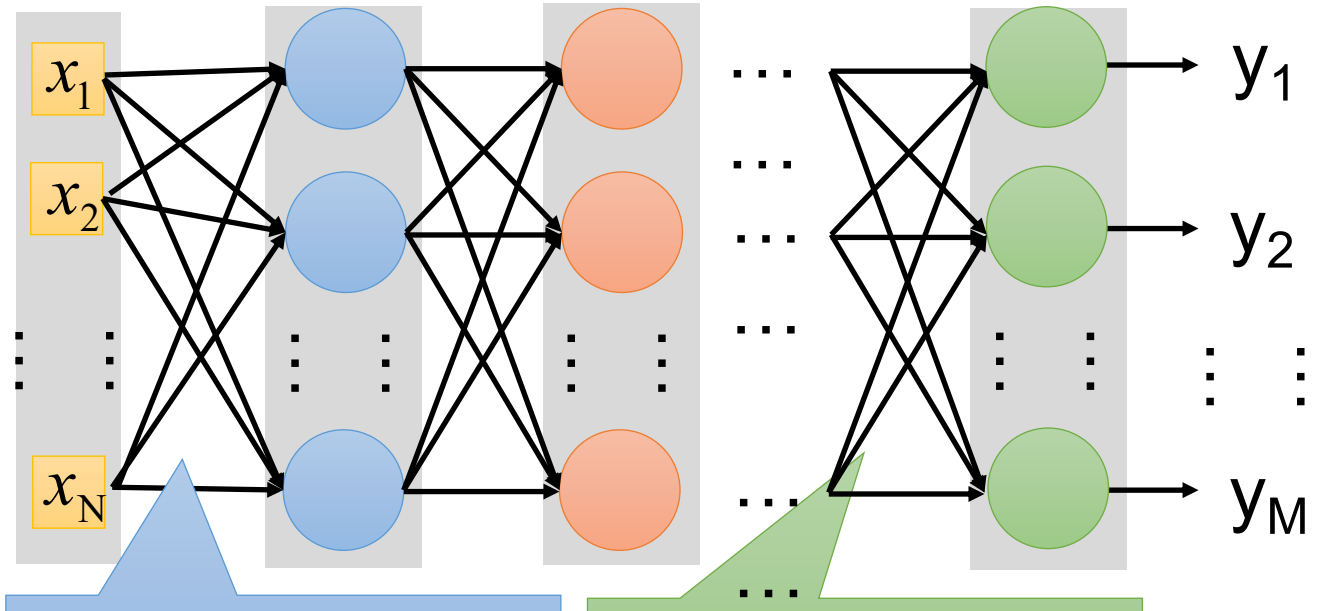
Recipe of Deep Learning



Hard to get the power of Deep …



Vanishing Gradient Problem



Smaller gradients

Larger gradients

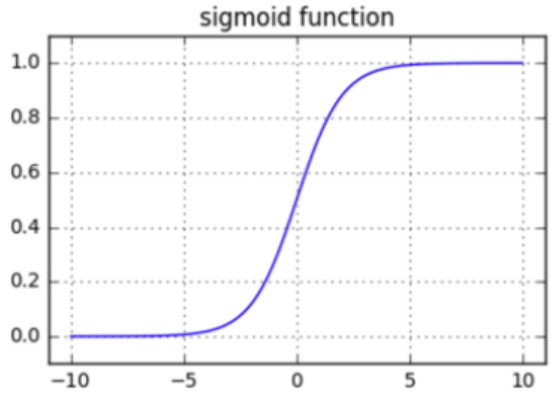
Learn very slow

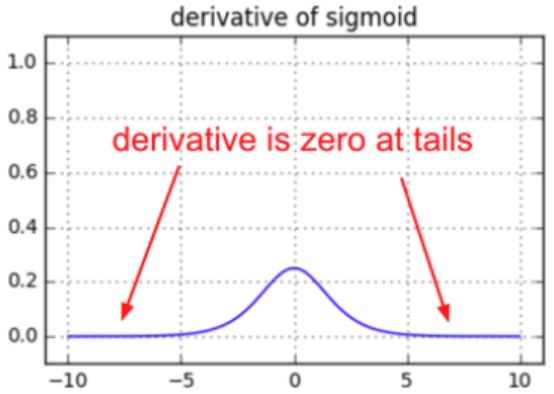
Learn very fast

Almost random

Already converge

based on random!?





$$y = \frac{1}{1+e^{-x}}$$

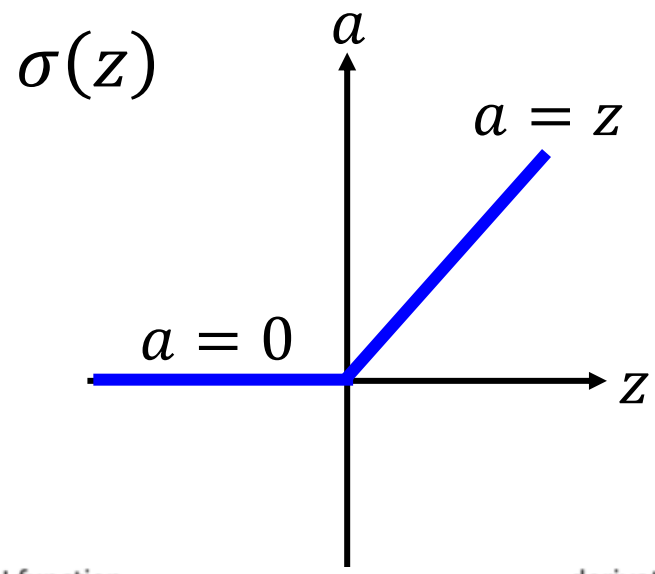
$$\frac{dy}{dx} = -\frac{1}{(1+e^{-x})^2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

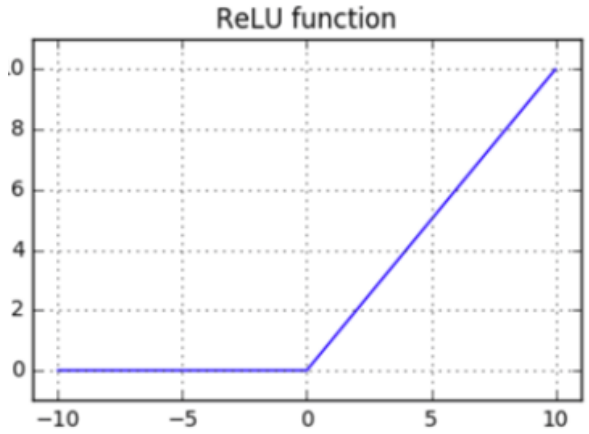
$$= \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) = y(1-y)$$

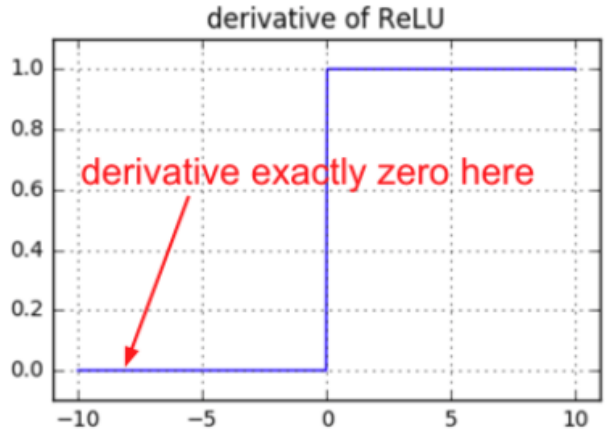
- **Max(dy/dx) = 0.25 => Slow convergence rate (with basic SGD), even vanishing gradient :**
  - **every time the gradient signal flows through a sigmoid gate, its magnitude always diminishes by one quarter (or more)**

ReLU

- Rectified Linear Unit (ReLU)





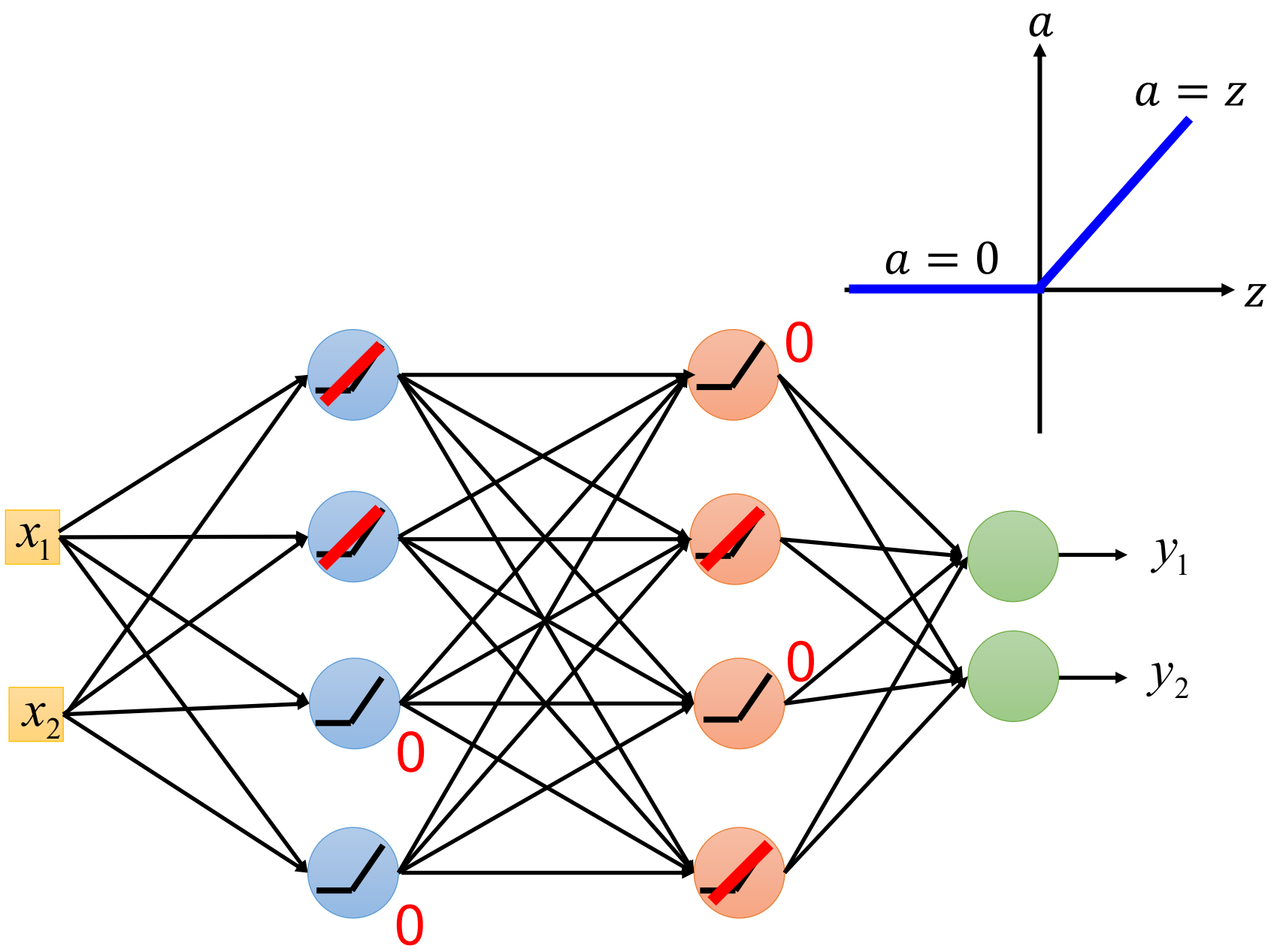


Reason:

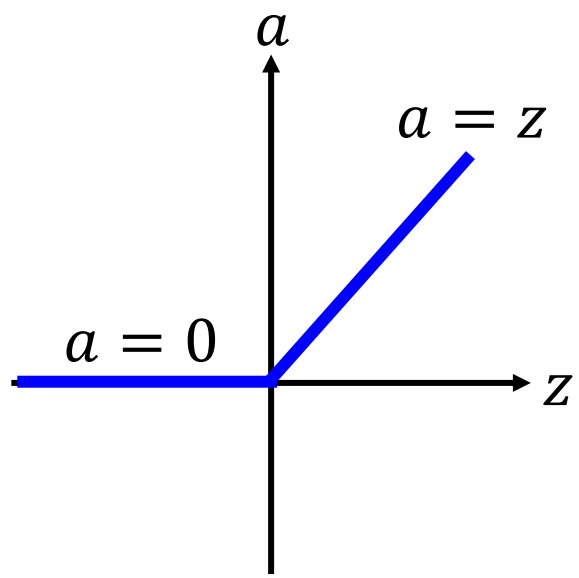
1. Fast to compute
2. Biological reason
3. Infinite sigmoid with different biases
4. Vanishing gradient problem

[Xavier Glorot, AISTATS'11] [Andrew L. Maas, ICML'13] [Kaiming He, arXiv'15]

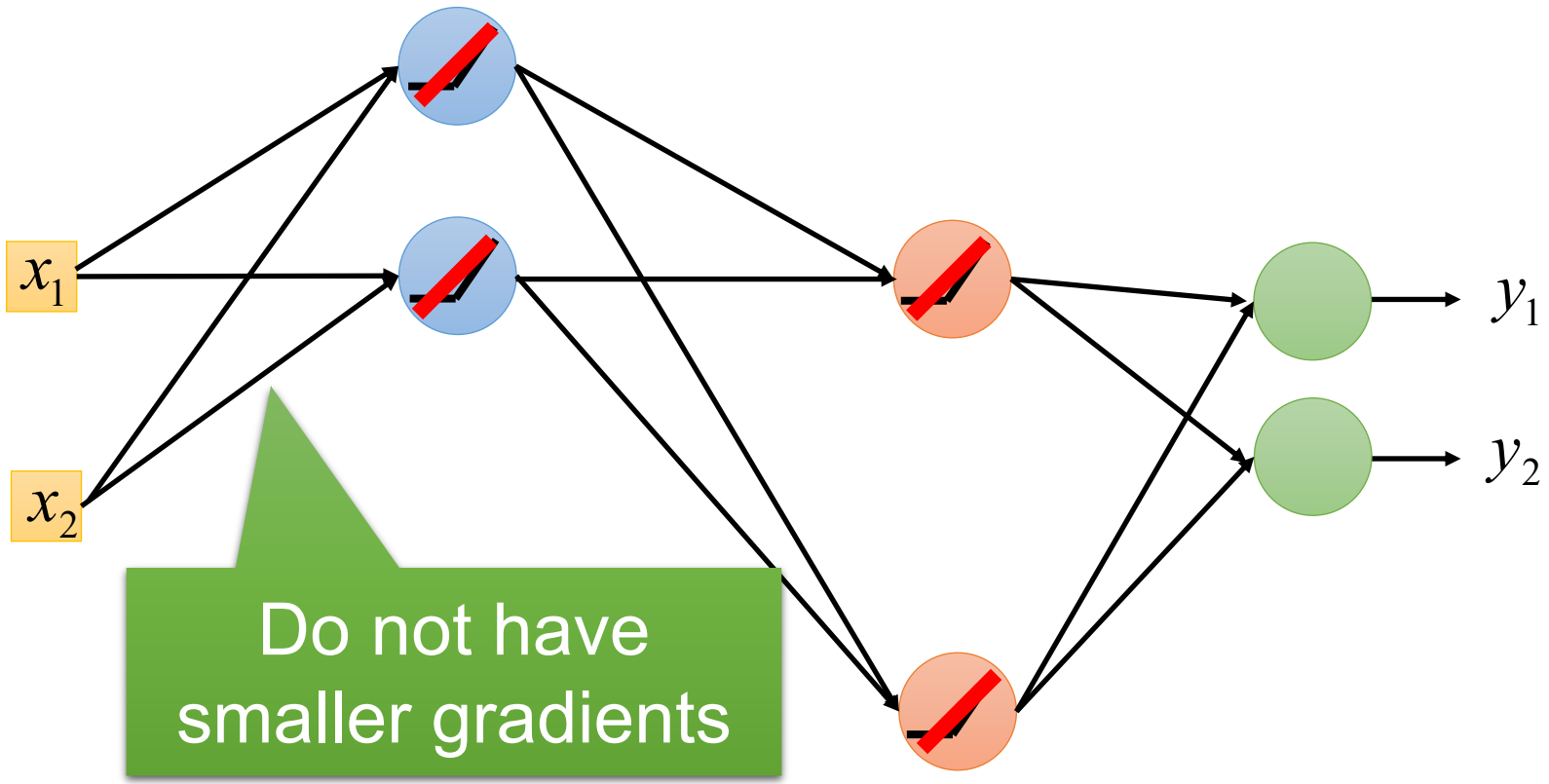
ReLU



ReLU



A Thinner linear network

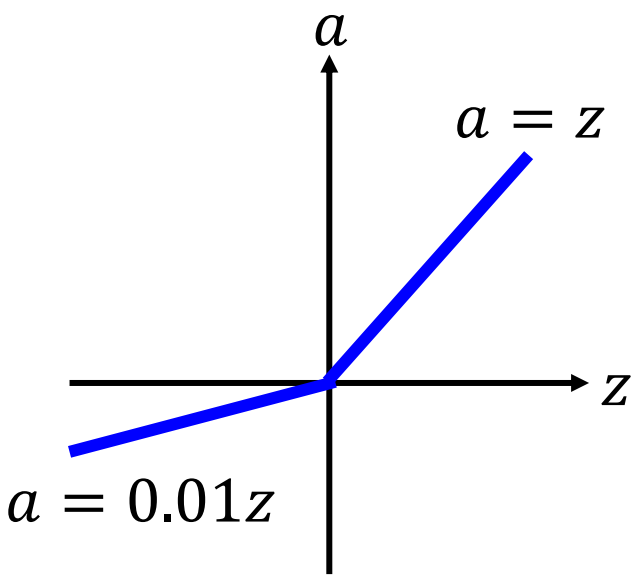


Do not have smaller gradients

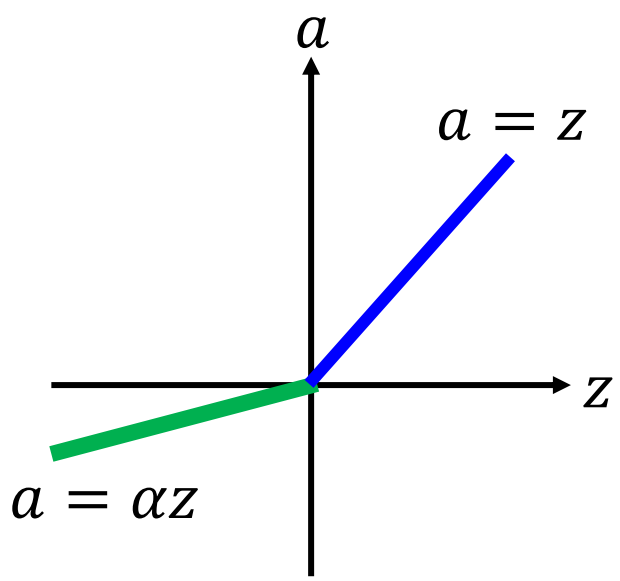
But => Dying ReLUs

ReLU - variant

Leaky ReLU

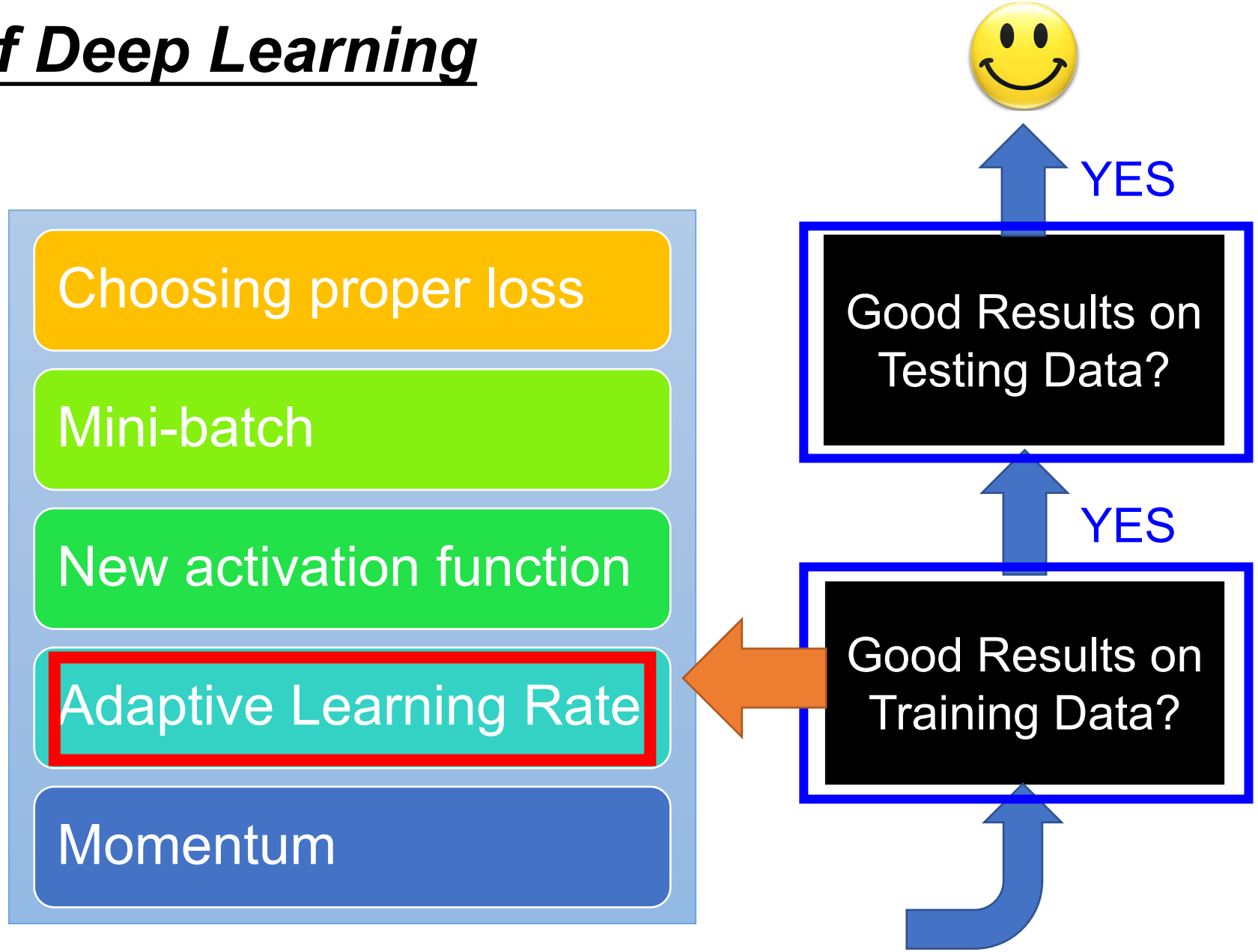


Parametric ReLU



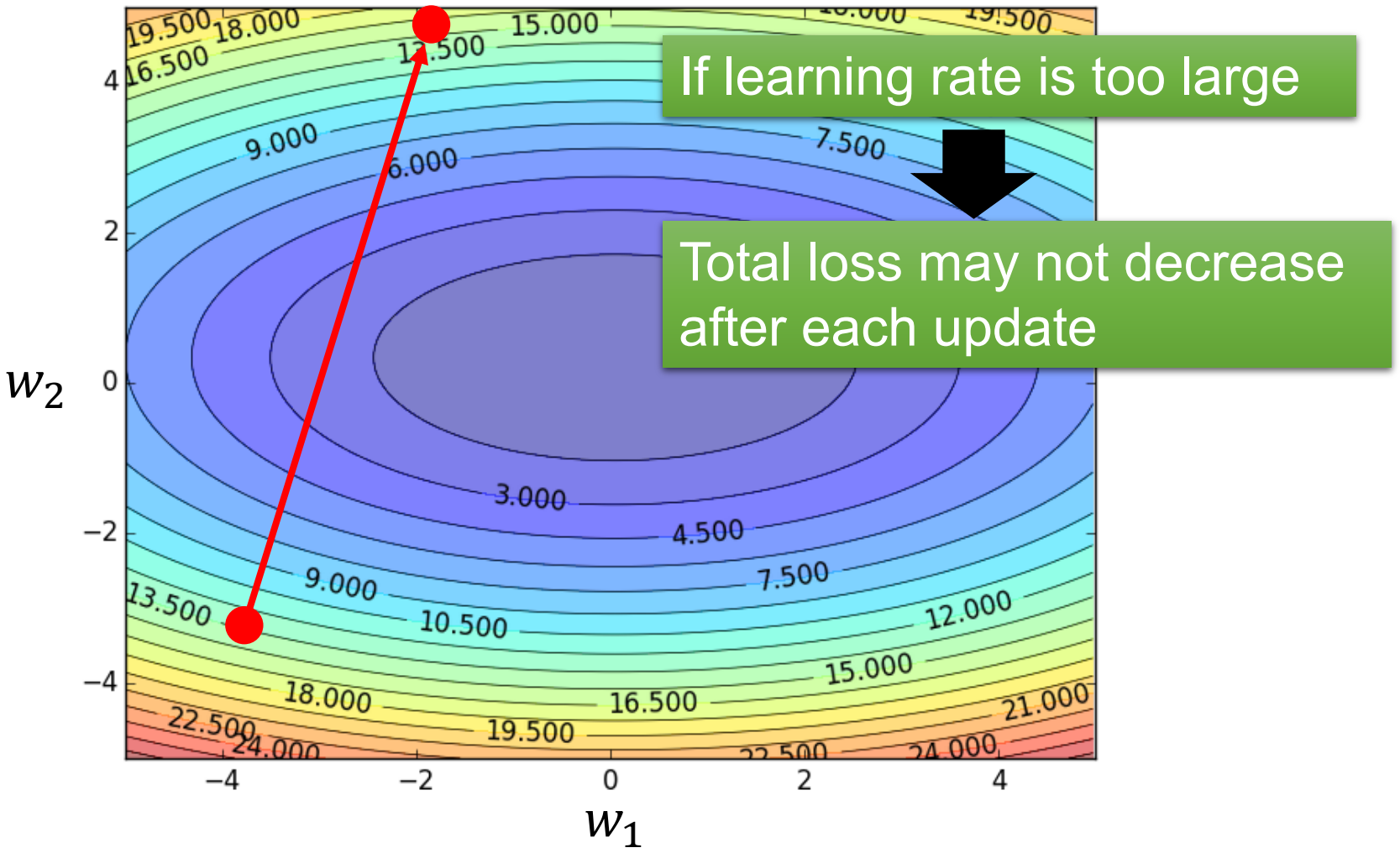
α also learned by gradient descent

Recipe of Deep Learning



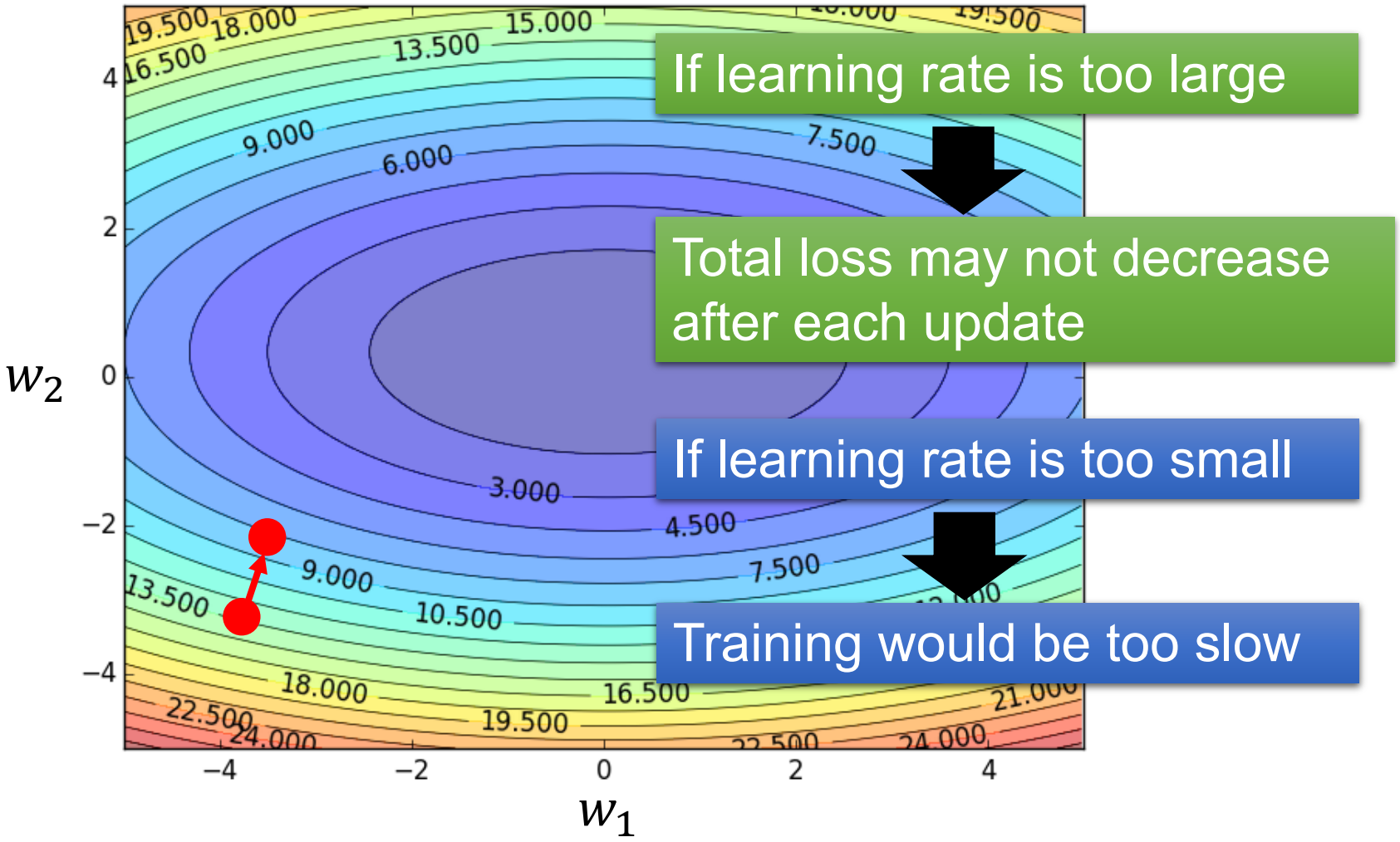
Learning Rates

Set the learning rate η carefully



Learning Rates

Set the learning rate η carefully



If learning rate is too large

Total loss may not decrease after each update

If learning rate is too small

Training would be too slow

Learning Rates

- Popular & Simple Idea: Reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \eta/\sqrt{t+1}$
- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates

Adagrad

Original: $w \leftarrow w - \eta \partial L / \partial w$

Adagrad: $w \leftarrow w - \eta_w \partial L / \partial w$

Parameter dependent learning rate

$$\eta_w = \frac{\eta}{\sqrt{\sum_{i=0}^{t} (g^i)^2}}$$

constant

g^i is $\partial L / \partial w$ obtained at the i-th update

Summation of the square of the previous derivatives

Adagrad

$$\eta_w = \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}}$$

	g^0
w_1	0.1

	g^0
w_2	20.0

Learning rate:

$$\frac{\eta}{\sqrt{0.1^2}} = \frac{\eta}{0.1}$$

$$\frac{\eta}{\sqrt{0.1^2 + 0.2^2}} = \frac{\eta}{0.22}$$

Learning rate:

$$\frac{\eta}{\sqrt{20^2}} = \frac{\eta}{20}$$

$$\frac{\eta}{\sqrt{20^2 + 10^2}} = \frac{\eta}{22}$$

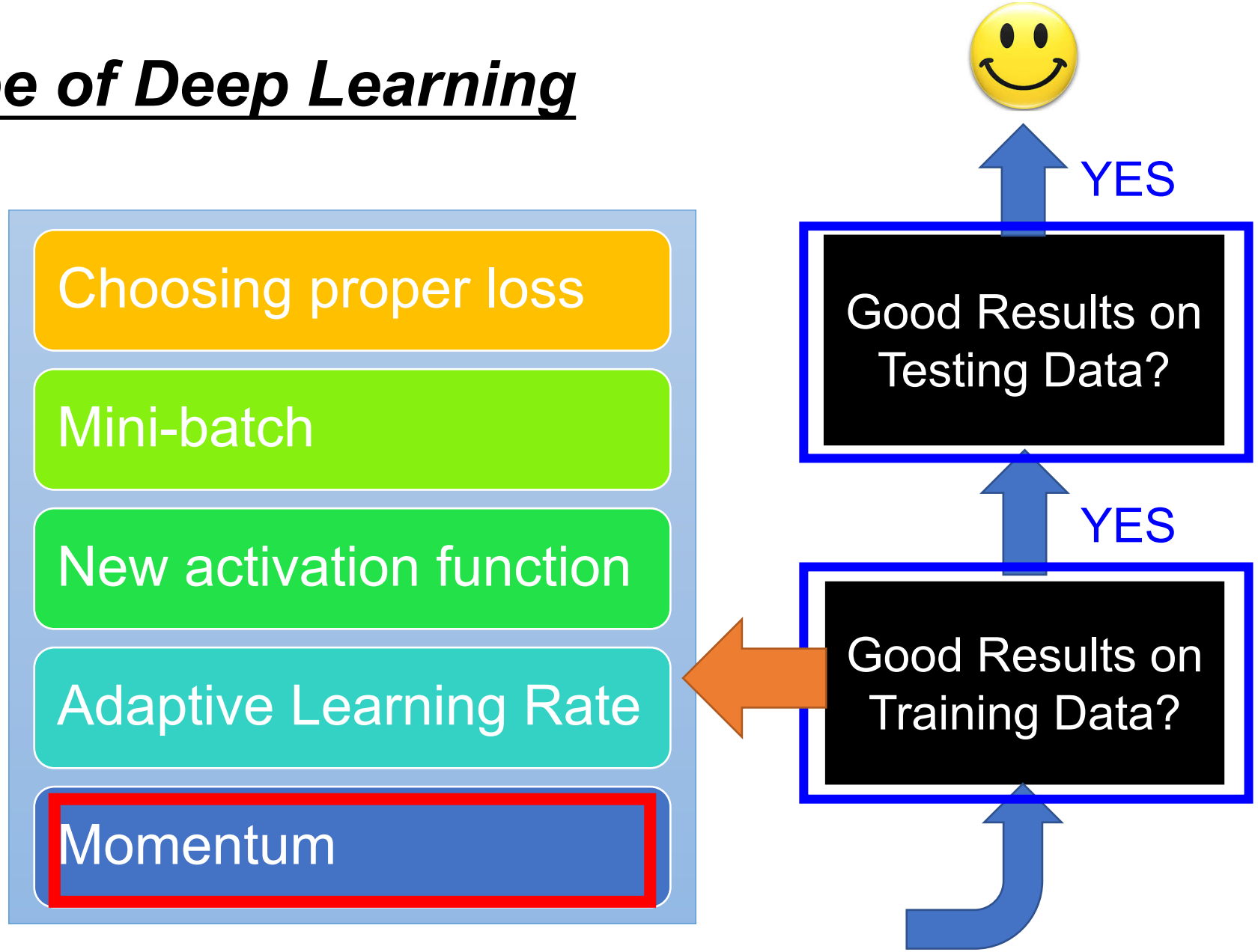
Observation:

1. Learning rate is smaller and smaller for all parameters
2. Smaller derivatives, larger learning rate, and vice versa

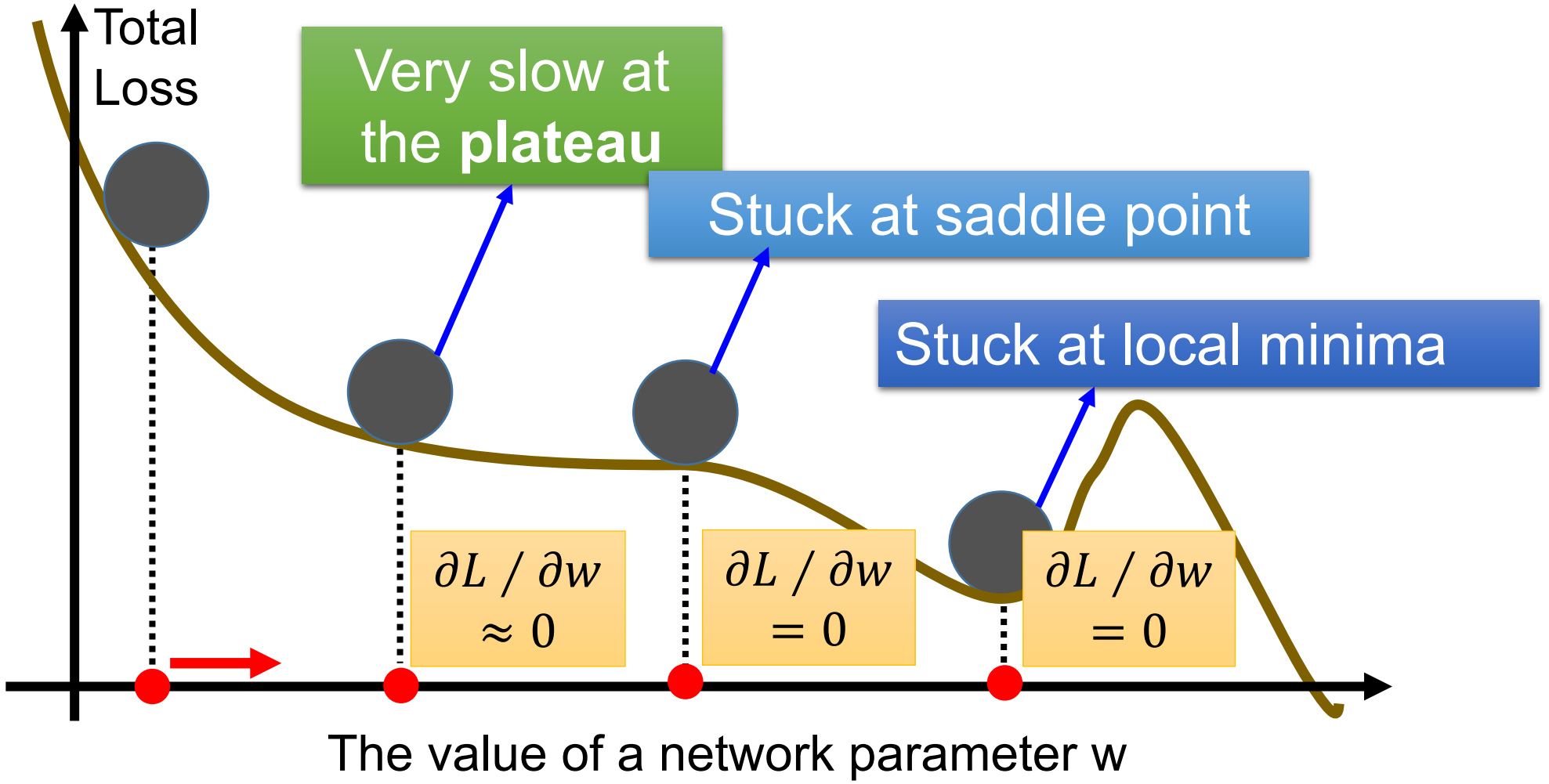
Not the whole story ……

- Adagrad [John Duchi, JMLR'11]
- RMSprop
  - https://www.youtube.com/watch?v=O3sxAc4hxZU
- Adadelta [Matthew D. Zeiler, arXiv'12]
- "No more pesky learning rates" [Tom Schaul, arXiv'12]
- AdaSecant [Caglar Gulcehre, arXiv'14]
- Adam [Diederik P. Kingma, ICLR'15]
- Nadam
  - http://cs229.stanford.edu/proj2015/054_report.pdf

Recipe of Deep Learning

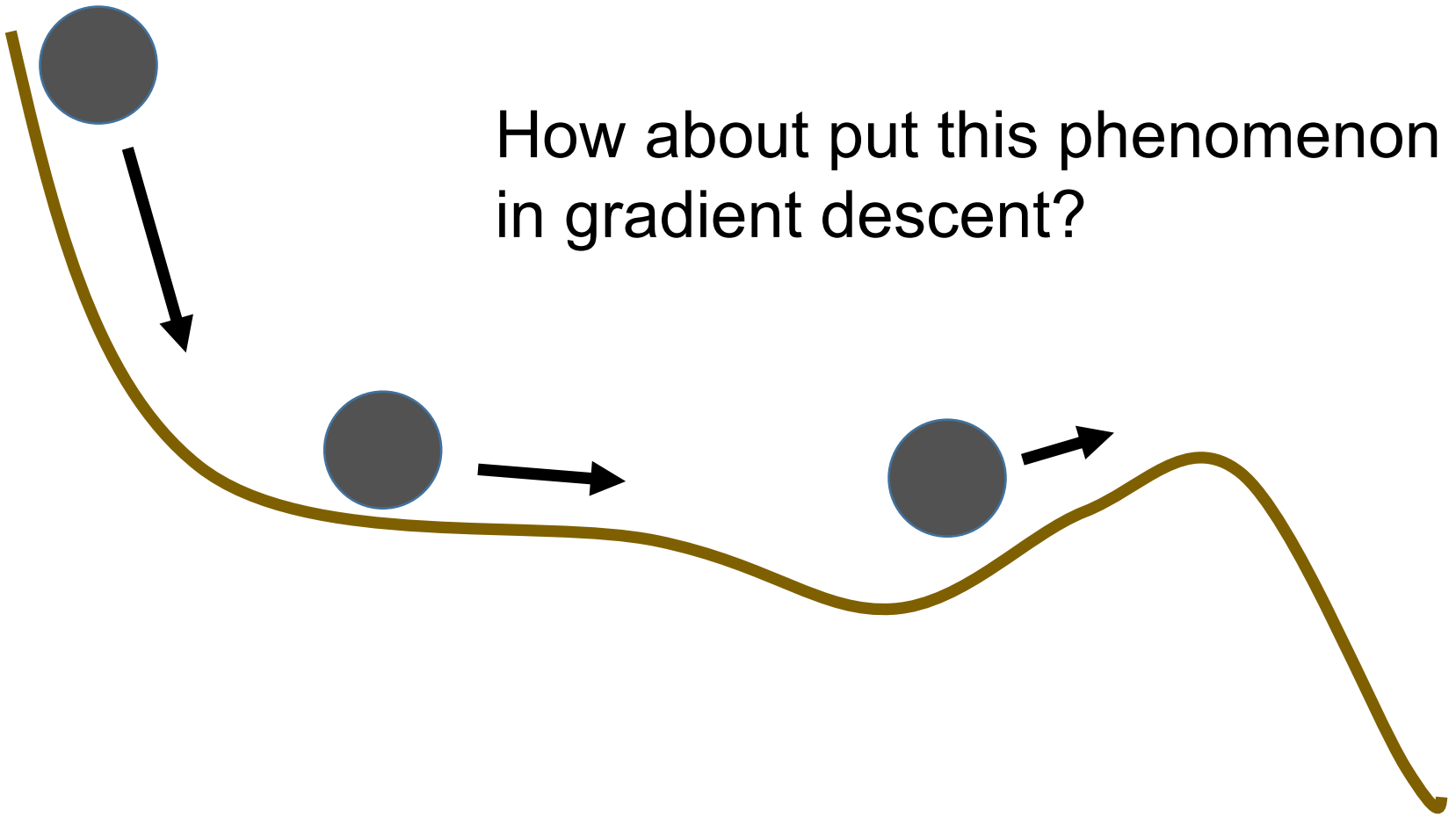


Hard to find optimal network parameters



In physical world ……

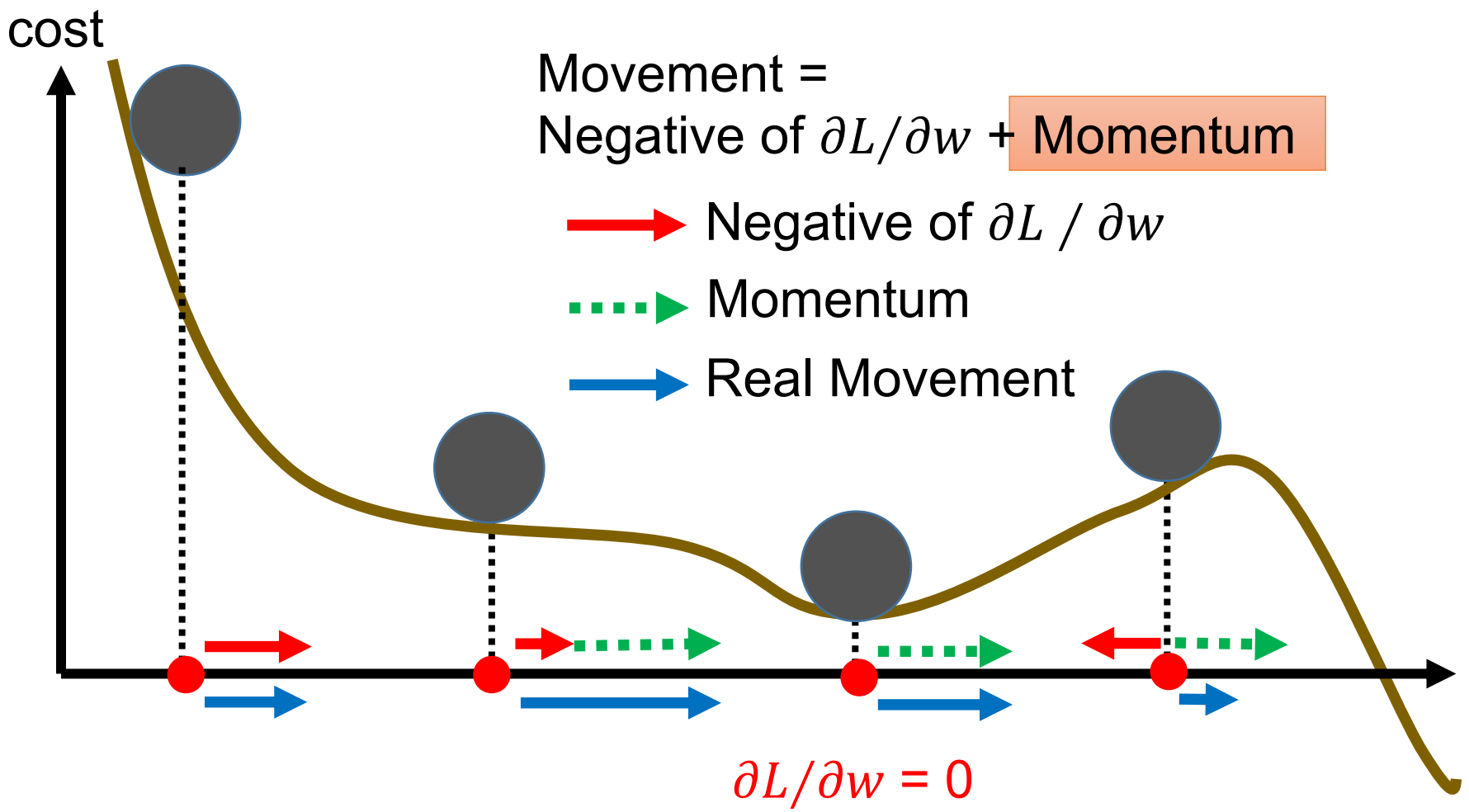
- Momentum



How about put this phenomenon in gradient descent?

Momentum

Still not guarantee reaching global minima, but give some hope ……



Adam

RMSProp (Advanced Adagrad) + Momentum

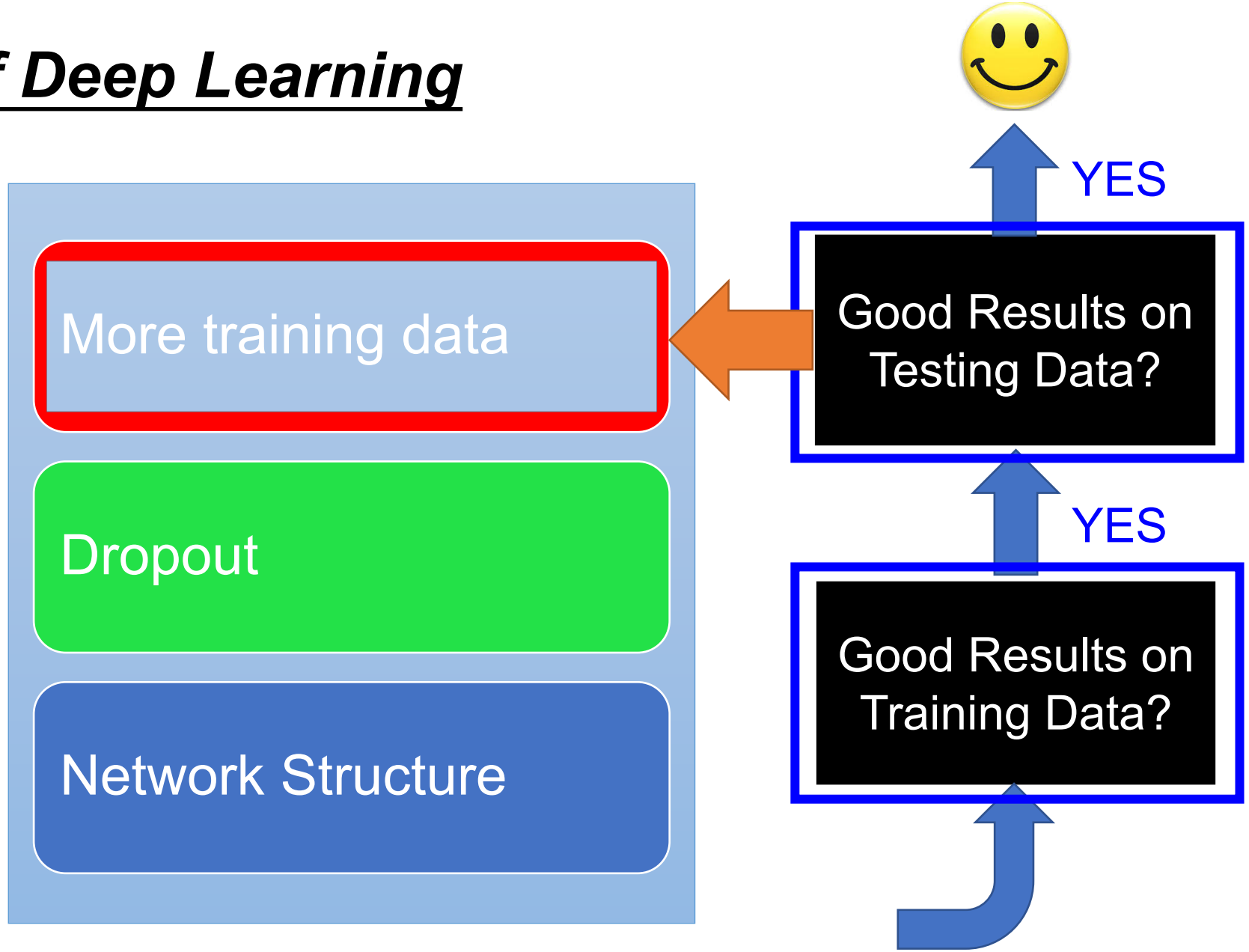
```
model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=0.1),
              metrics=['accuracy'])
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])
```

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Recipe of Deep Learning



Panacea for Overfitting

- Have more training data
- ***Create*** more training data (?)

Handwriting recognition:

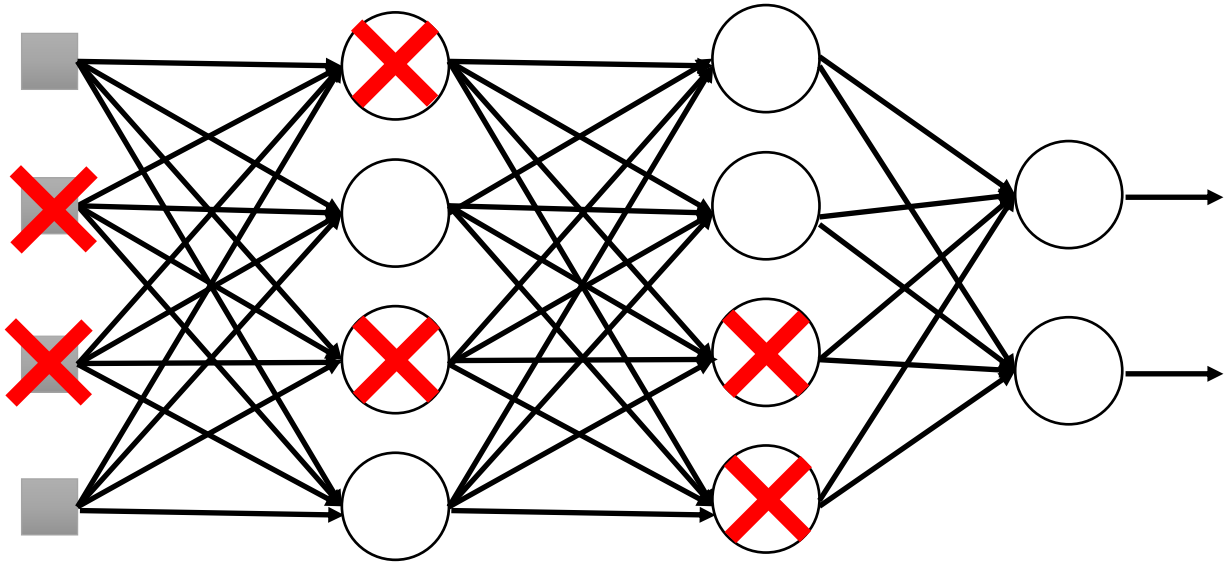
Original Training Data:

Created Training Data

Shift 15°

Dropout

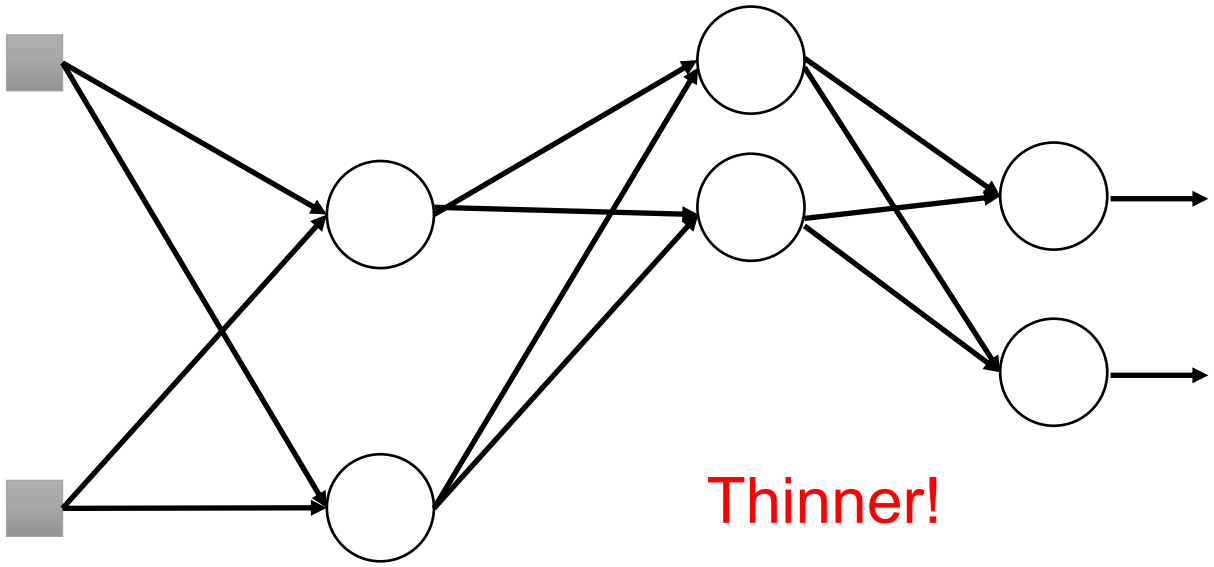
Training:



- **Each time before updating the parameters**
  - Each neuron has p% to dropout

Dropout

Training:

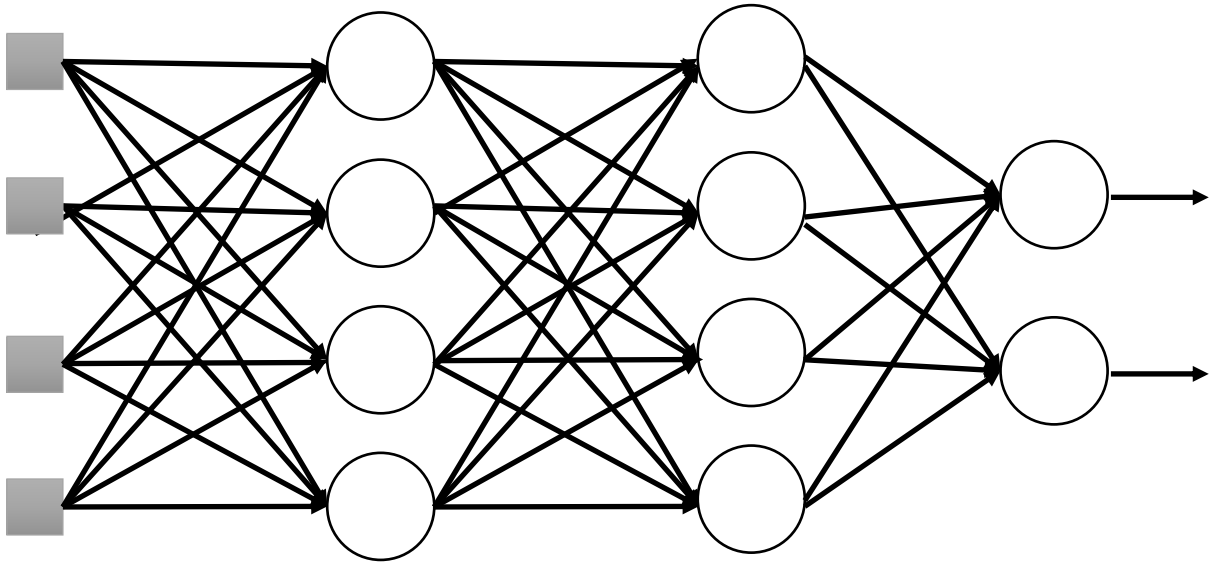


- **Each time before updating the parameters**
  - Each neuron has p% to dropout
    - **The structure of the network is changed.**
  - Using the new network for training

For each mini-batch, we resample the dropout neurons

Dropout

Testing:



- **No dropout**
  - If the dropout rate at training is p%, all the weights times 1-p%
  - Assume that the dropout rate is 50%.
    If a weight $w = 1$ by training, set $w = 0.5$ for testing.

Dropout - Intuitive Reason

Training

Dropout (腳上綁重物)

Testing

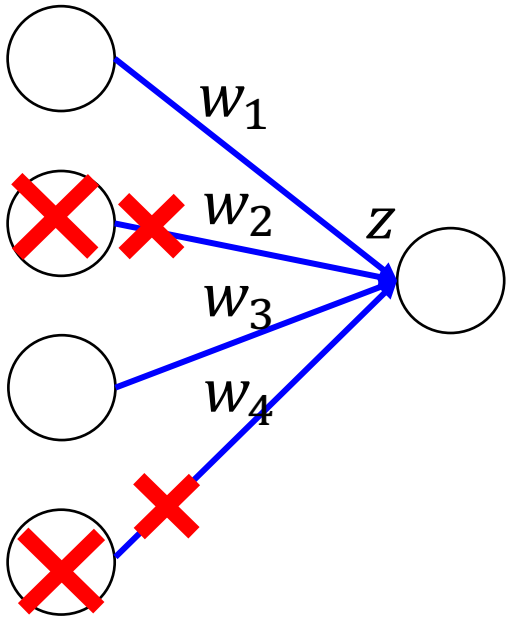
No dropout
(拿下重物後就變很強)

Dropout - Intuitive Reason

- Why the weights should multiply (1-p)% (dropout rate) when testing?

Training of Dropout

Assume dropout rate is 50%

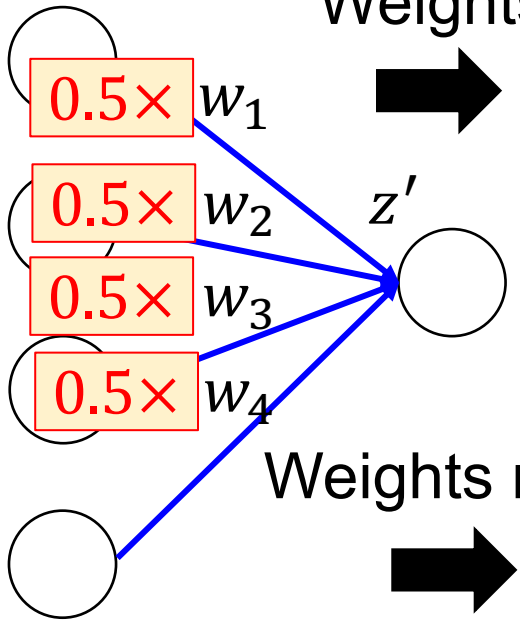


Testing of Dropout

No dropout

Weights from training

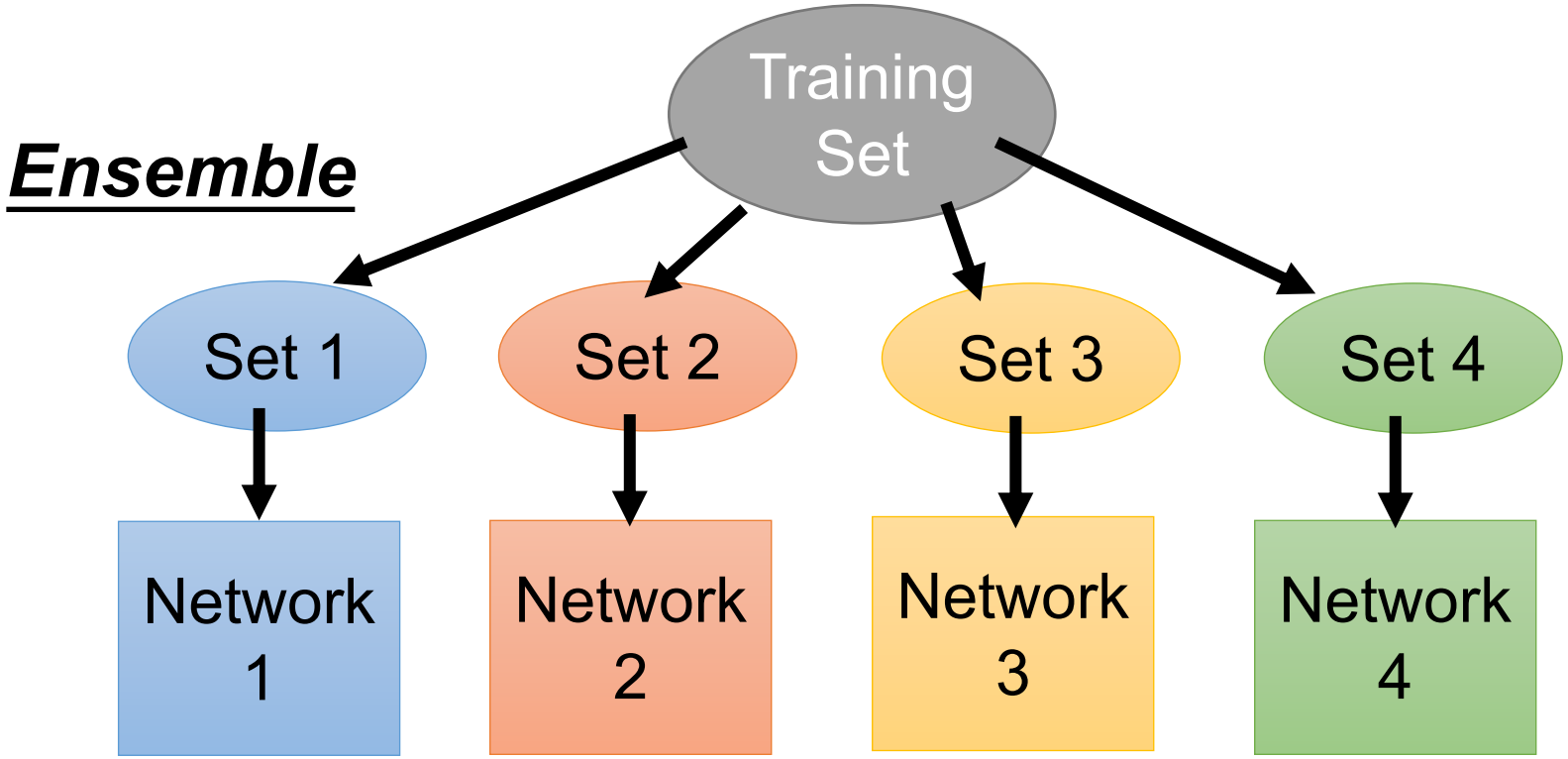
$z' \approx 2z$



Weights multiply 1-p%

$z' \approx z$

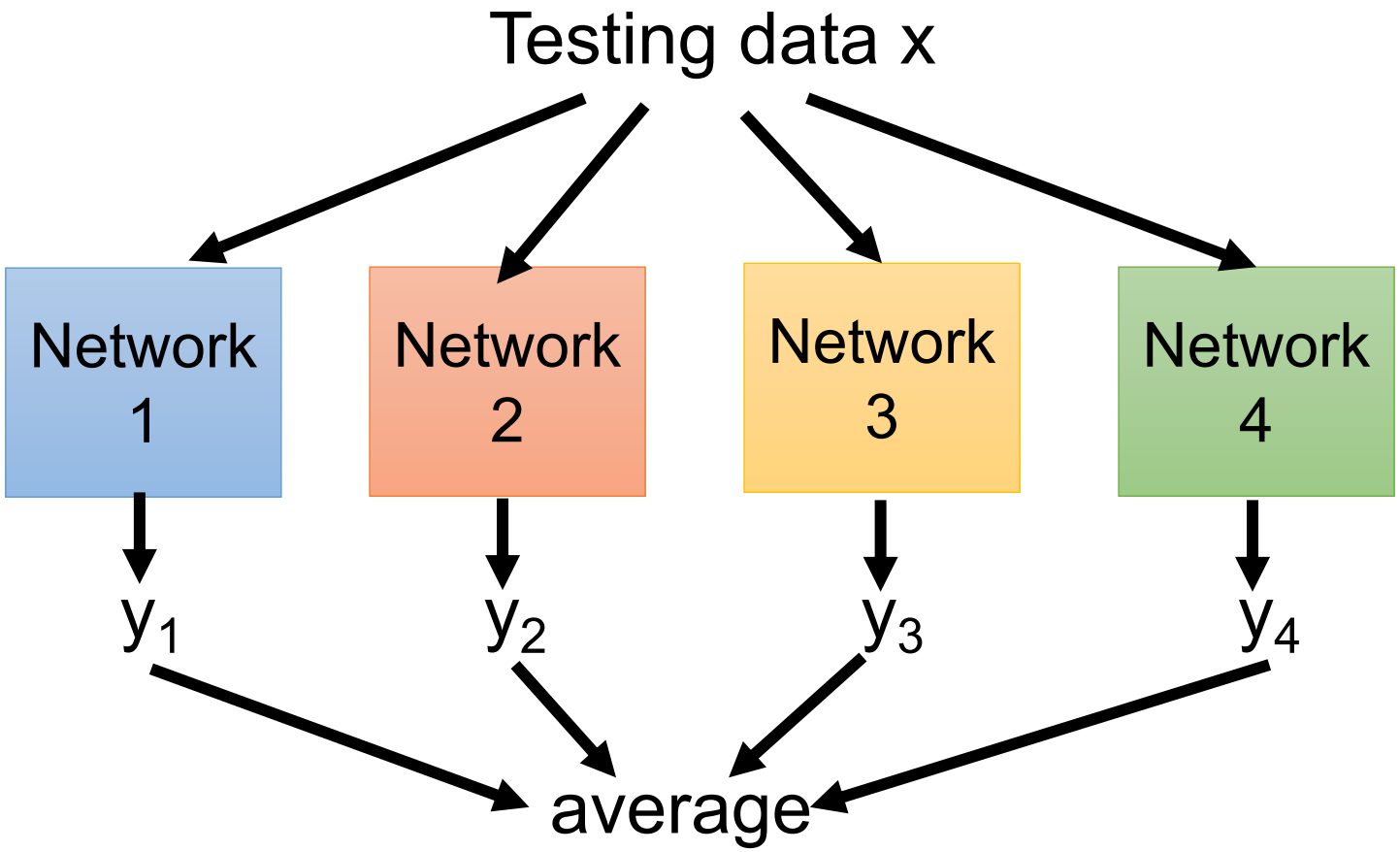
Dropout is a kind of ensemble.



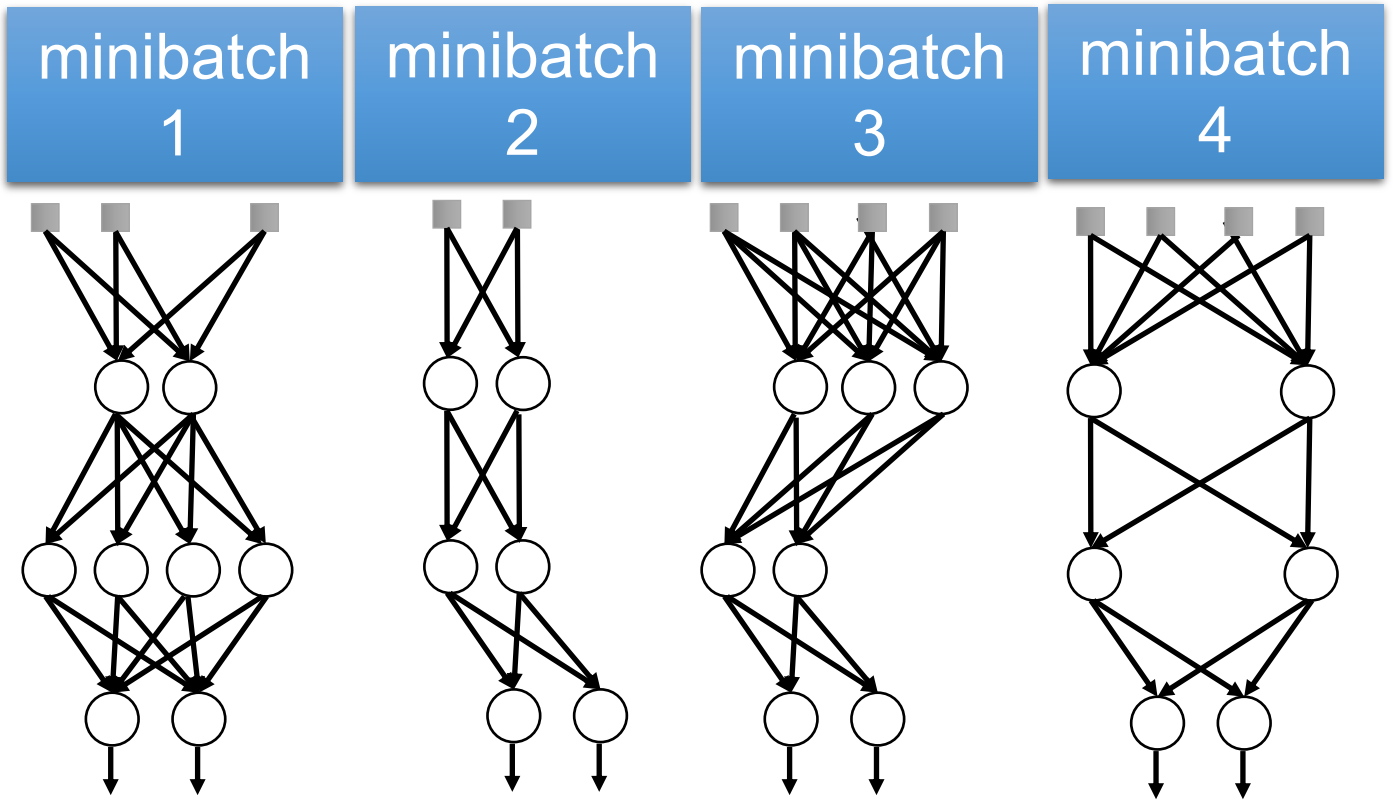
Train a bunch of networks with different structures

Dropout is a kind of ensemble.

Ensemble



Dropout is a kind of ensemble.



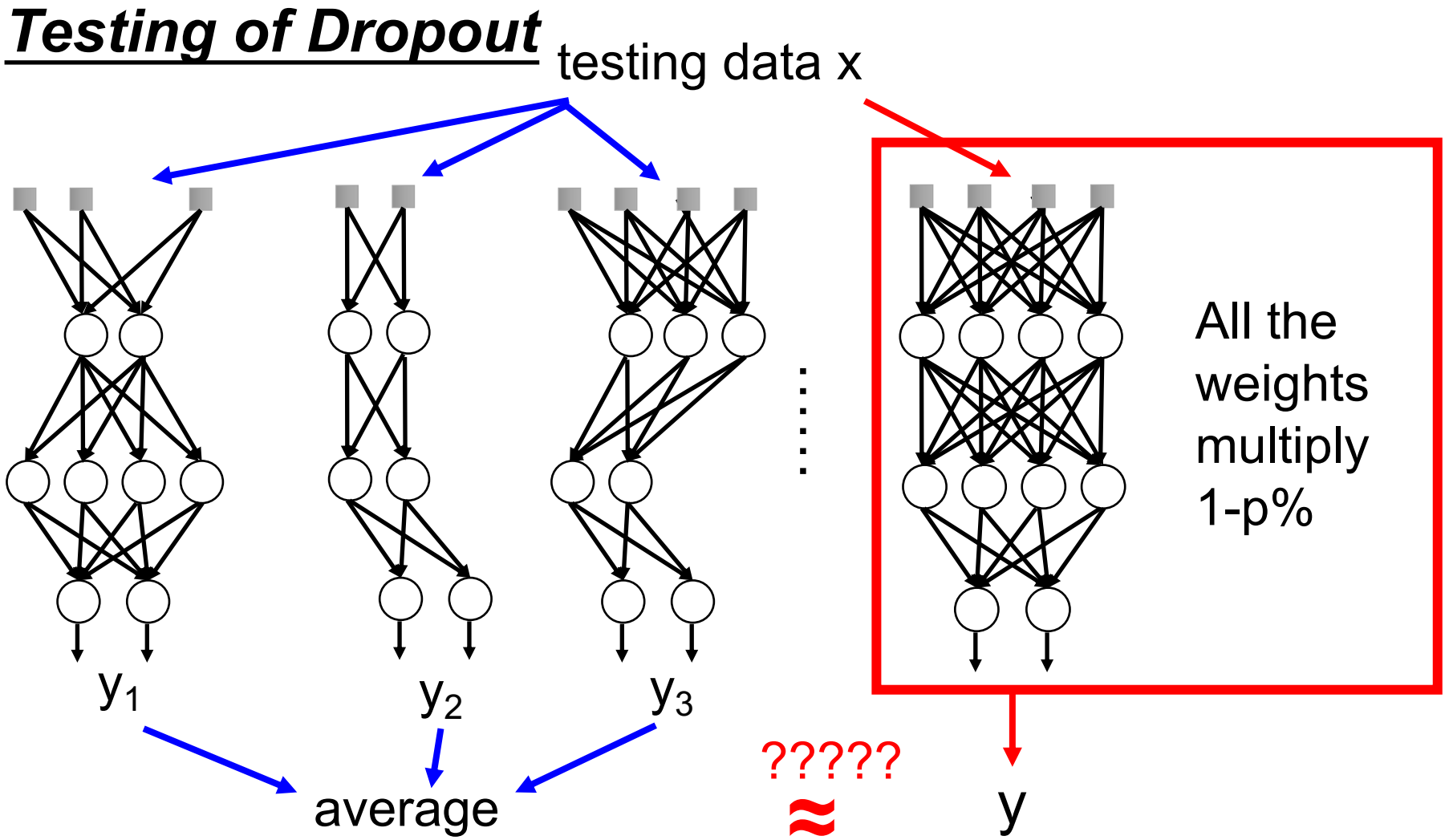
Training of Dropout

M neurons

2^M possible networks

- Using one mini-batch to train one network
- Some parameters in the network are shared

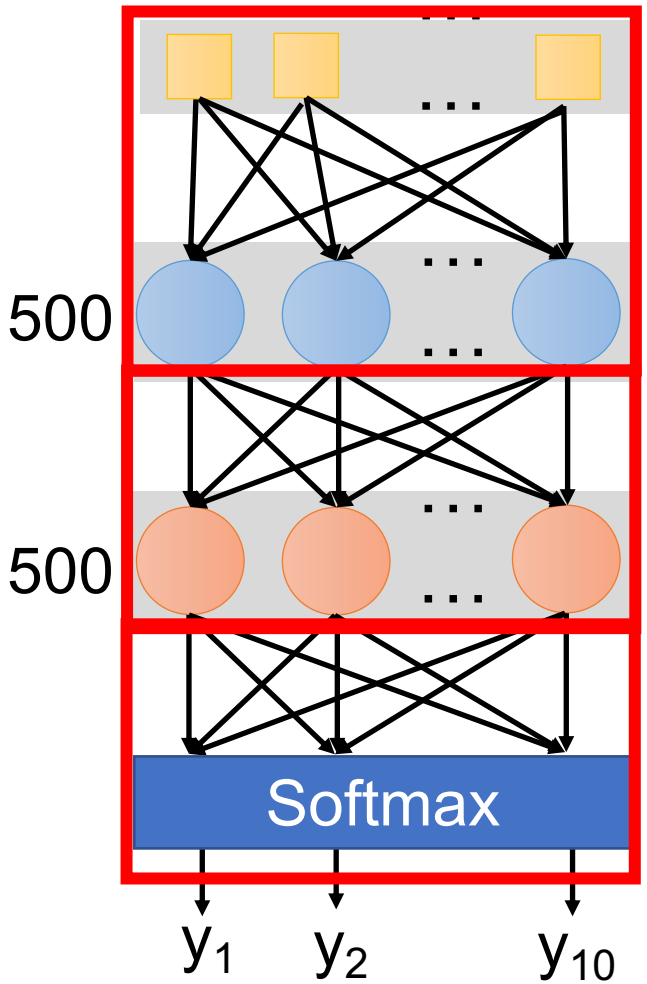
Dropout is a kind of ensemble.



More about dropout

- More reference for dropout [Nitish Srivastava, JMLR'14] [Pierre Baldi, NIPS'13][Geoffrey E. Hinton, arXiv'12]
- Dropout works better with Maxout [Ian J. Goodfellow, ICML'13]
- Dropconnect [Li Wan, *ICML'13*]
  - Dropout delete neurons
  - Dropconnect deletes the connection between neurons
- Annealed dropout [S.J. Rennie, SLT'14]
  - Dropout rate decreases by epochs
- Standout [J. Ba, NISP'13]
  - Each neural has different dropout rate

Demo




```
model = Sequential()
```

```
model.add( Dense( input_dim=28*28,
                  output_dim=500 ))
model.add( Activation('sigmoid') )
```

```
model.add( dropout(0.8) )
```

```
model.add( Dense( output_dim=500 ) )
model.add( Activation('sigmoid') )
```

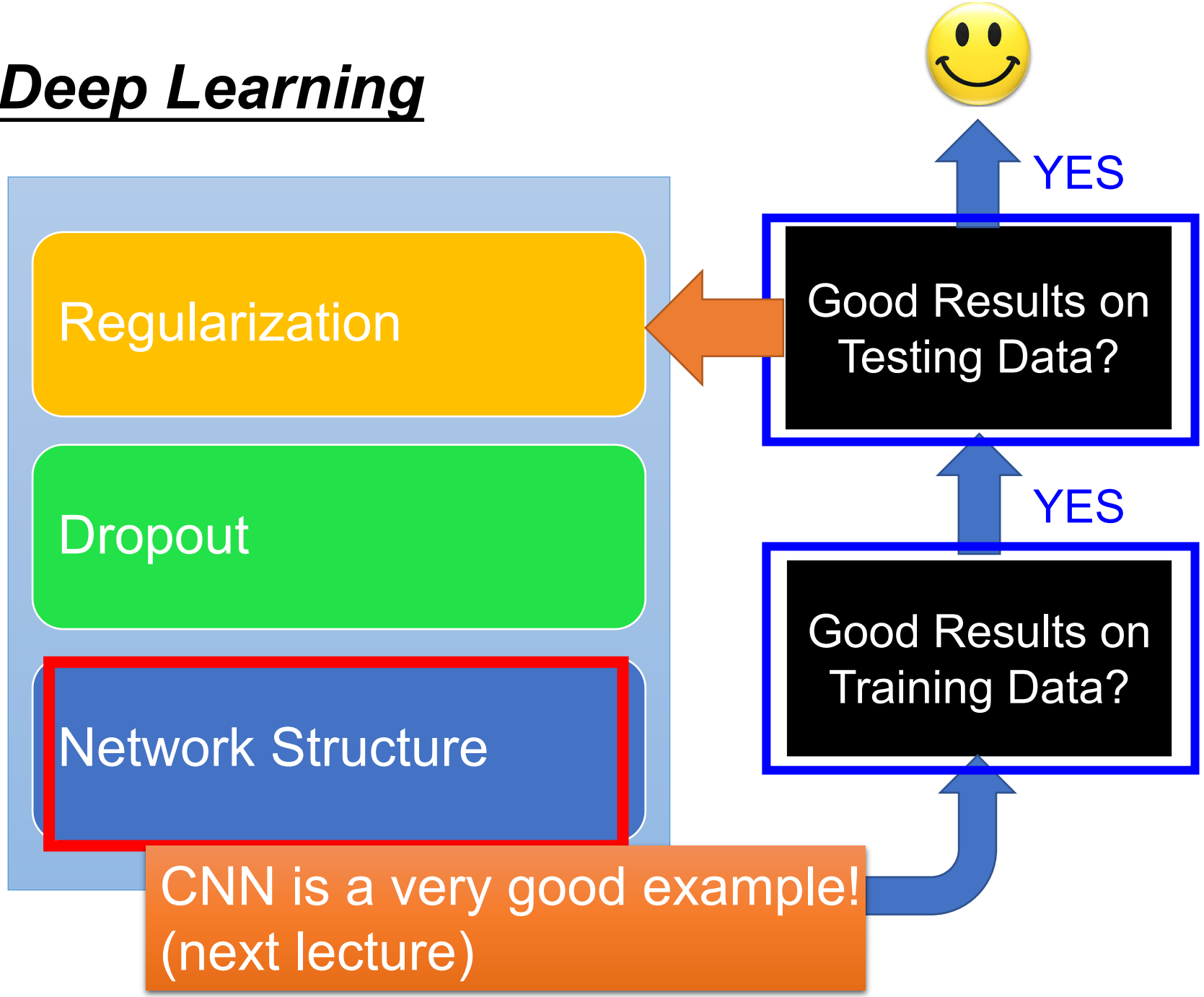
```
model.add( dropout(0.8) )
```

```
model.add( Dense(output_dim=10 ) )
model.add( Activation('softmax') )
```

...

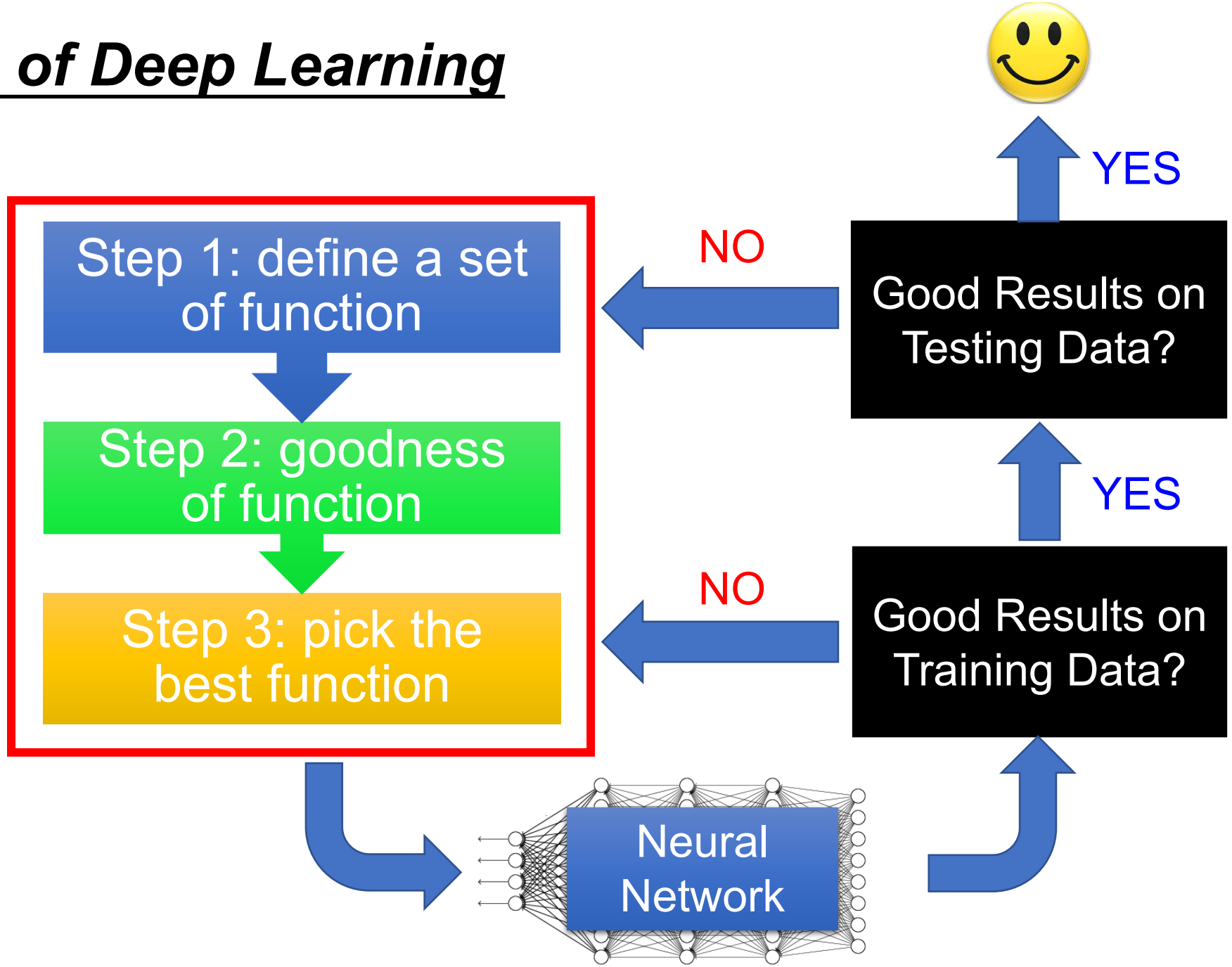
...

Recipe of Deep Learning



Concluding Remarks

Recipe of Deep Learning



Thanks