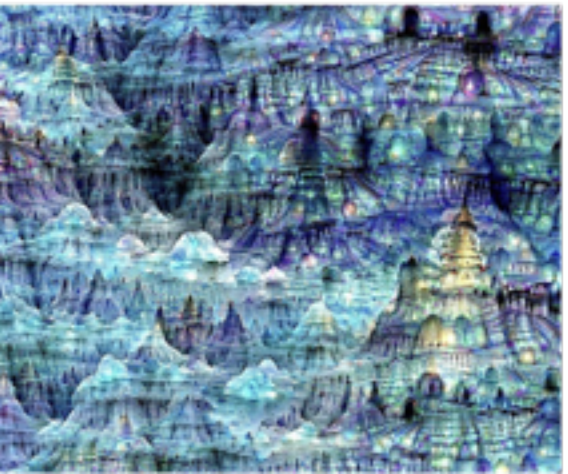
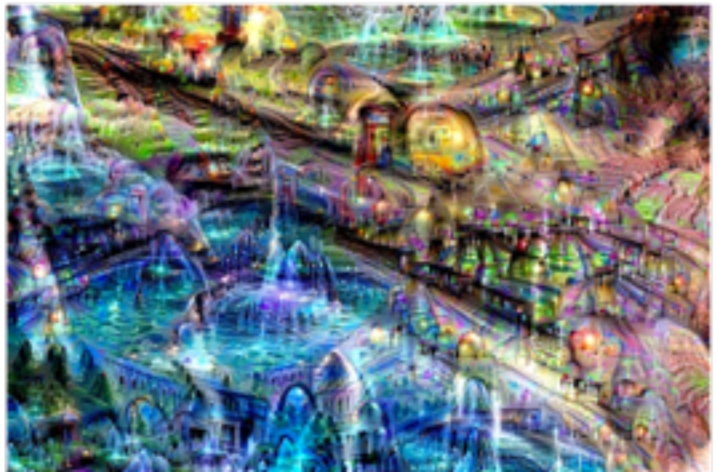
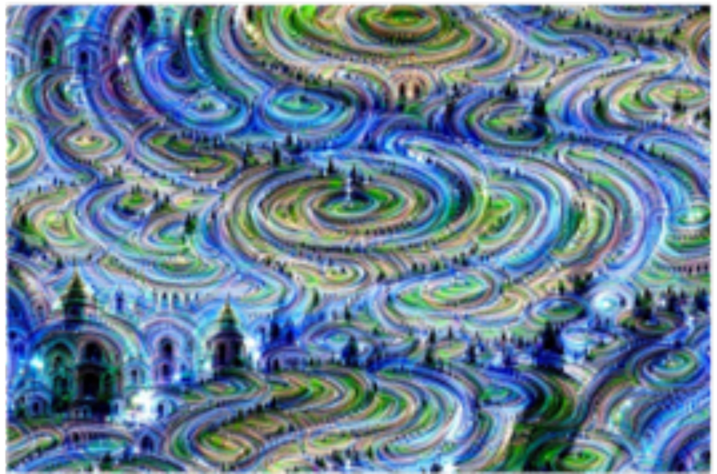
Lecture 35: Optimization and Neural Nets

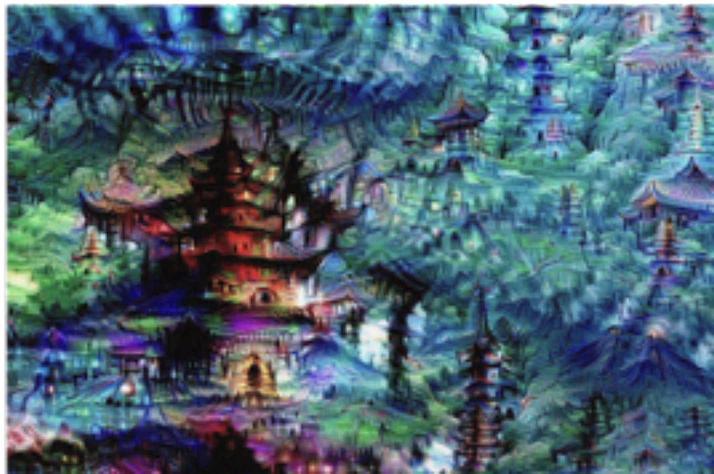
CS 4670/5670
Sean Bell

"DeepDream" [Google, "Inceptionism: Going Deeper into Neural Networks", blog 2015]









Aside: “CNN” vs “ConvNet”

Note:

- There are many papers that use either phrase, but
- “ConvNet” is the preferred term, since “CNN” clashes with that other thing called CNN

(Review) Softmax Classifier

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}} \qquad L_i = -\log p_{i,y_i}$$

unnormalized probabilities

	unnormalized log probabilities			normalize	probabilities	
cat	**3.2**	exp →	**24.5**	→	**0.13**	→ L_i = -log(0.13) = **0.89**
car	5.1		164.0		0.87	
frog	-1.7		0.18		0.00	

Q2: At initialization, W is small and thus s ~= 0.
What is the loss L?

Slide from Karpathy 2016

Softmax vs SVM Loss

Slide from Karpathy 2016

Softmax vs SVM Loss

Softmax:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Slide from Karpathy 2016

Softmax vs SVM Loss

Softmax:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Slide from Karpathy 2016

Softmax vs SVM Loss

Softmax:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

SVM:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $\boxed{y_i = 0}$

Slide from Karpathy 2016

Softmax vs SVM Loss

Softmax:

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

assume scores:

[10, -2, 3]

[10, 9, 9]

[10, -100, -100]

and $\boxed{y_i = 0}$

SVM:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Q: Suppose I take a datapoint and I jiggle a bit (changing its score slightly). What happens to the loss in both cases?

Slide from Karpathy 2016

Softmax Classifier

Let's code this up in NumPy:

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Softmax Classifier

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

Softmax Classifier

Let's code this up in NumPy:

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

- What if there is the value 1000 appears in "s"?

Softmax Classifier

Let's code this up in NumPy:

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

- What if there is the value 1000 appears in "s"?

Overflow —> we get inf/inf = NaN

Softmax Classifier

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

Let's code this up in NumPy:

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

- What if there is the value 1000 appears in "s"?

  Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

Softmax Classifier

Let's code this up in NumPy:

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

- What if there is the value 1000 appears in "s"?

    Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

    Underflow —> we get 0/0 = NaN

Softmax Classifier

Let's code this up in NumPy:

$$p_{i,j} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

```
def softmax(s):
    exp_s = np.exp(s)
    probs = exp_s / np.sum(exp_s, axis=1, keepdims=True)
    return probs
```

Doesn't work — what's the problem?

- What if there is the value 1000 appears in "s"?

    Overflow —> we get inf/inf = NaN

- What if the largest value is -1000?

    Underflow —> we get 0/0 = NaN

This expression is numerically unstable

Softmax Classifier

Softmax Classifier

Observation: subtracting a constant does not change “p”:

Softmax Classifier

Observation: subtracting a constant does not change “p”:

$$p_{i,j} = \frac{e^{s_{i,j}-C}}{\sum_k e^{s_{j,k}-C}} = \frac{e^{-C}e^{s_{i,j}}}{\sum_k e^{-C}e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

Softmax Classifier

Observation: subtracting a constant does not change “p”:

$$p_{i,j} = \frac{e^{s_{i,j}-C}}{\sum_k e^{s_{j,k}-C}} = \frac{e^{-C}e^{s_{i,j}}}{\sum_k e^{-C}e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

If we choose “C” to be the max, then it works:

Softmax Classifier

Observation: subtracting a constant does not change “p”:

$$p_{i,j} = \frac{e^{s_{i,j}-C}}{\sum_k e^{s_{j,k}-C}} = \frac{e^{-C}e^{s_{i,j}}}{\sum_k e^{-C}e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

If we choose “C” to be the max, then it works:

- If a large value appears in “s”, then that value will become 1 and all others will be 0 (avoiding overflow)

Softmax Classifier

Observation: subtracting a constant does not change “p”:

$$p_{i,j} = \frac{e^{s_{i,j}-C}}{\sum_k e^{s_{j,k}-C}} = \frac{e^{-C}e^{s_{i,j}}}{\sum_k e^{-C}e^{s_{i,k}}} = \frac{e^{s_{i,j}}}{\sum_k e^{s_{i,k}}}$$

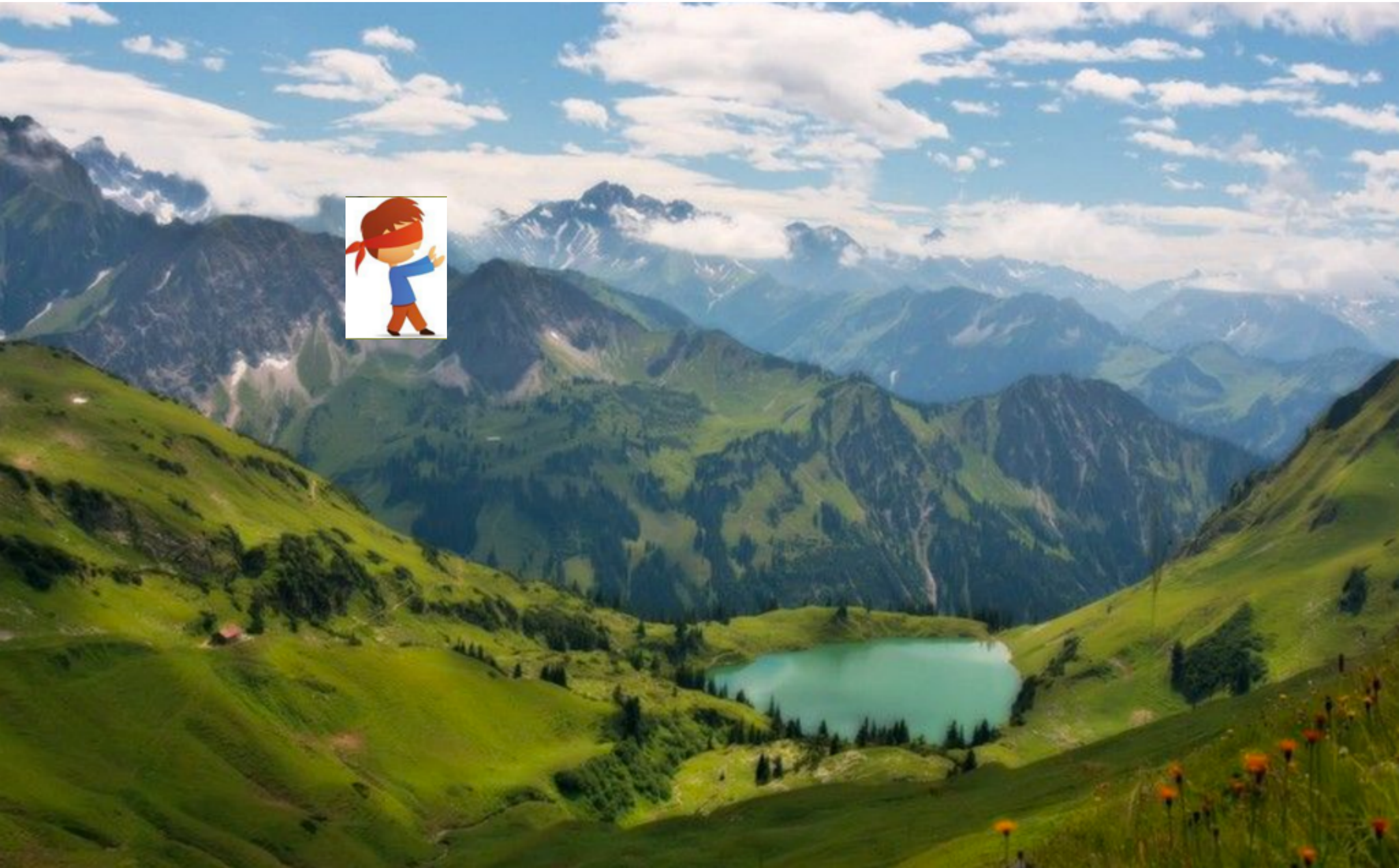
If we choose “C” to be the max, then it works:

- If a large value appears in “s”, then that value will become 1 and all others will be 0 (avoiding overflow)
- If all values in “s” are large negative, then they will be shifted up towards 0 (avoiding underflow)

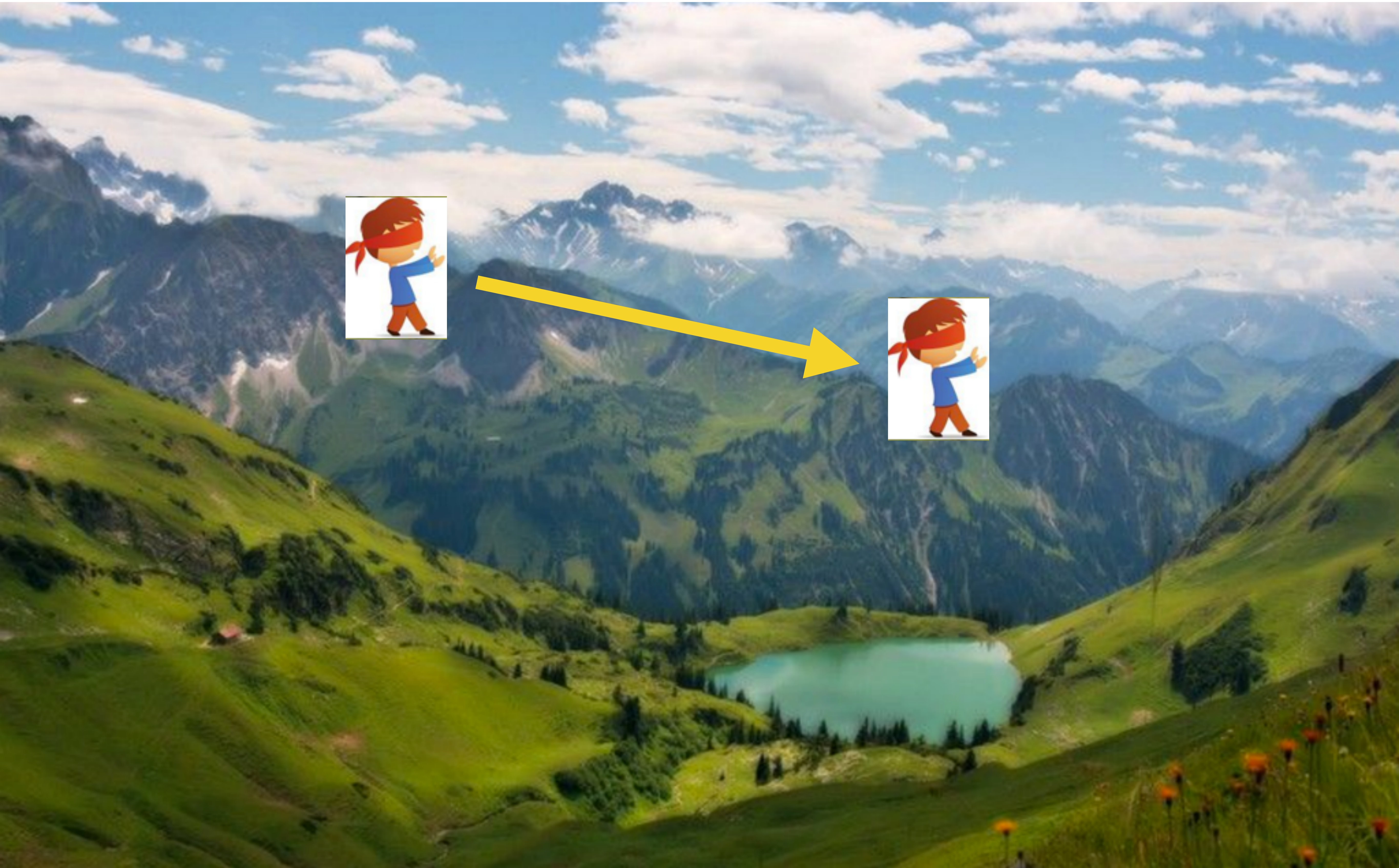
Optimization

How do we minimize the loss L?

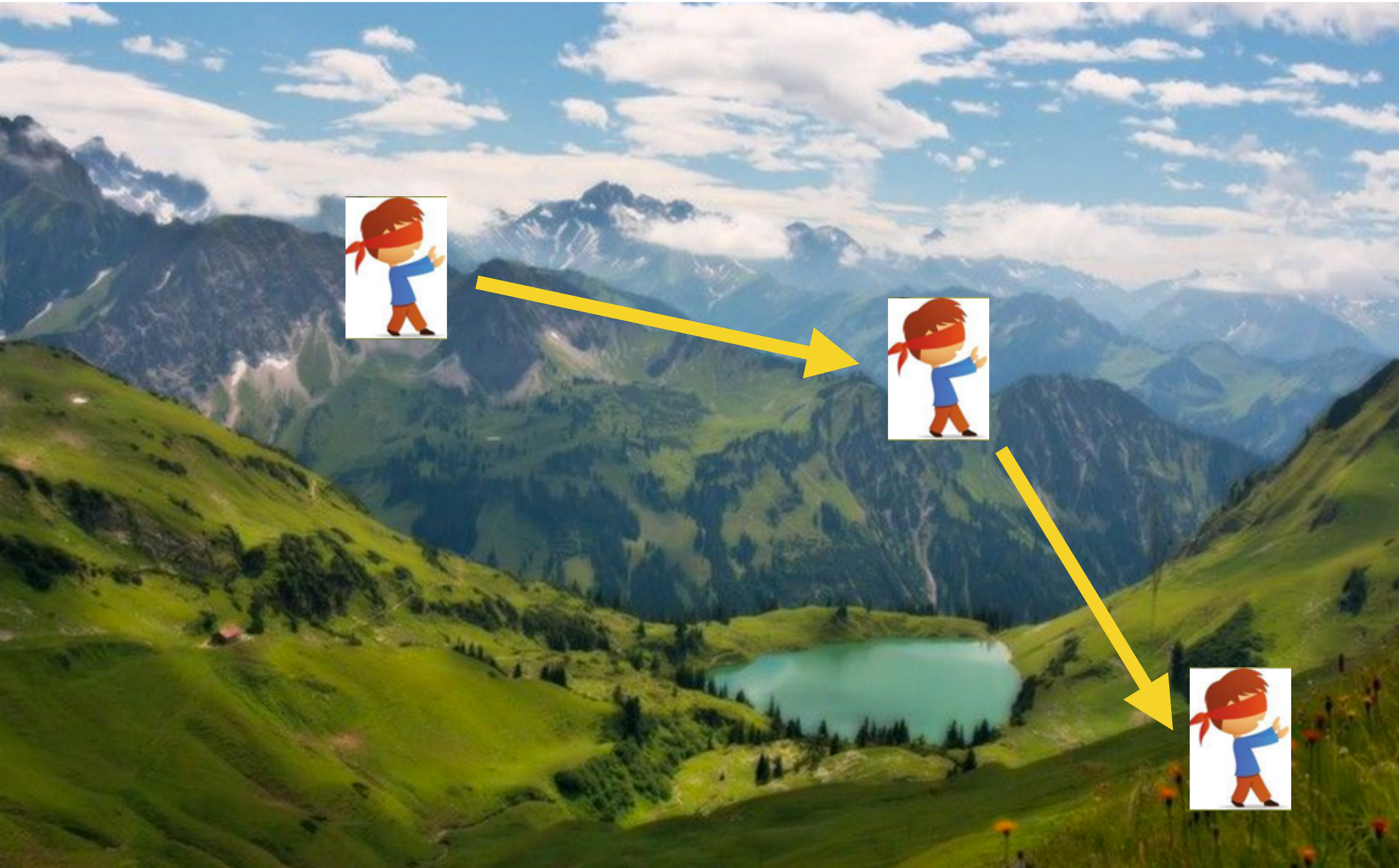
Idea #1: Random Search



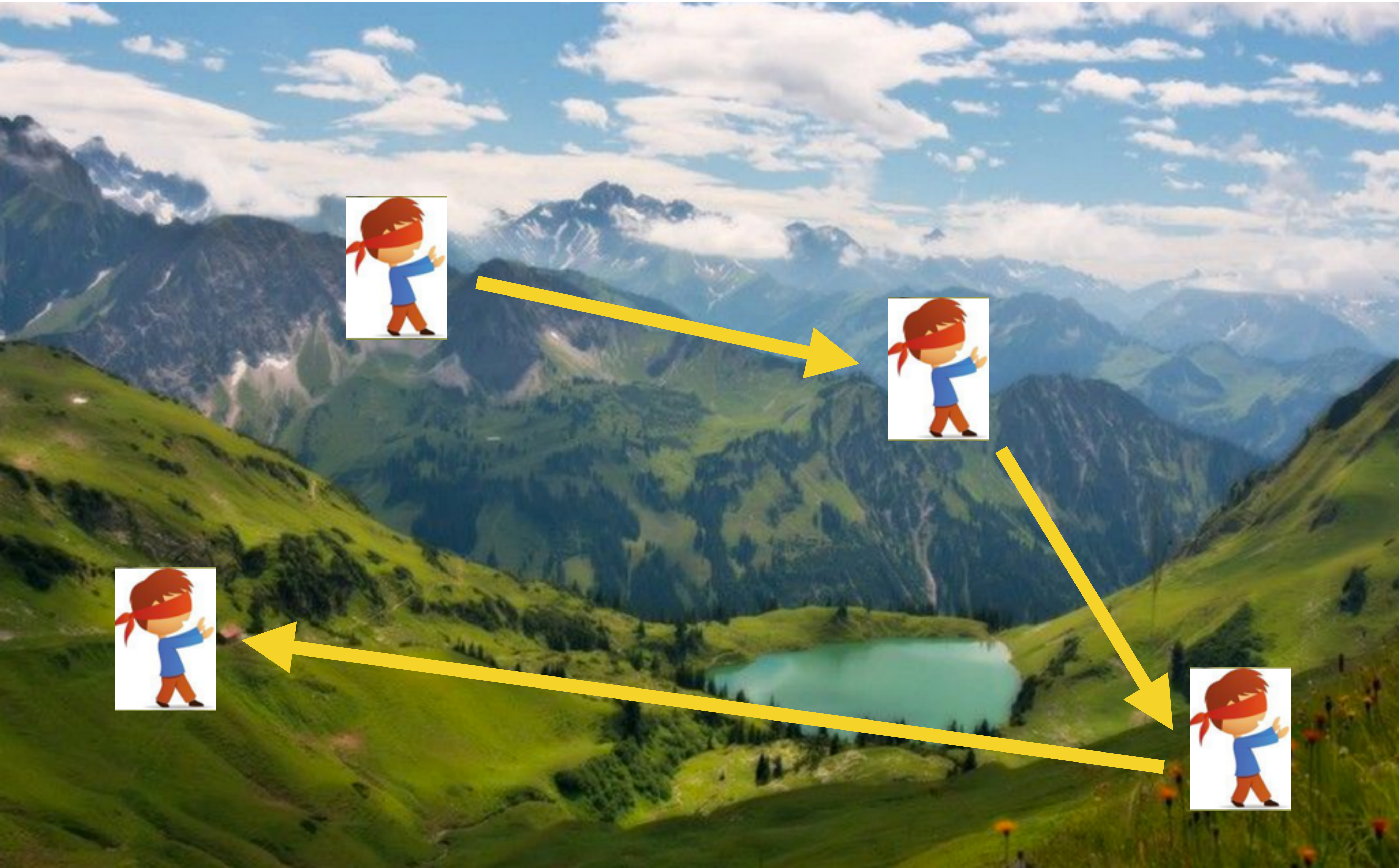
Idea #1: Random Search



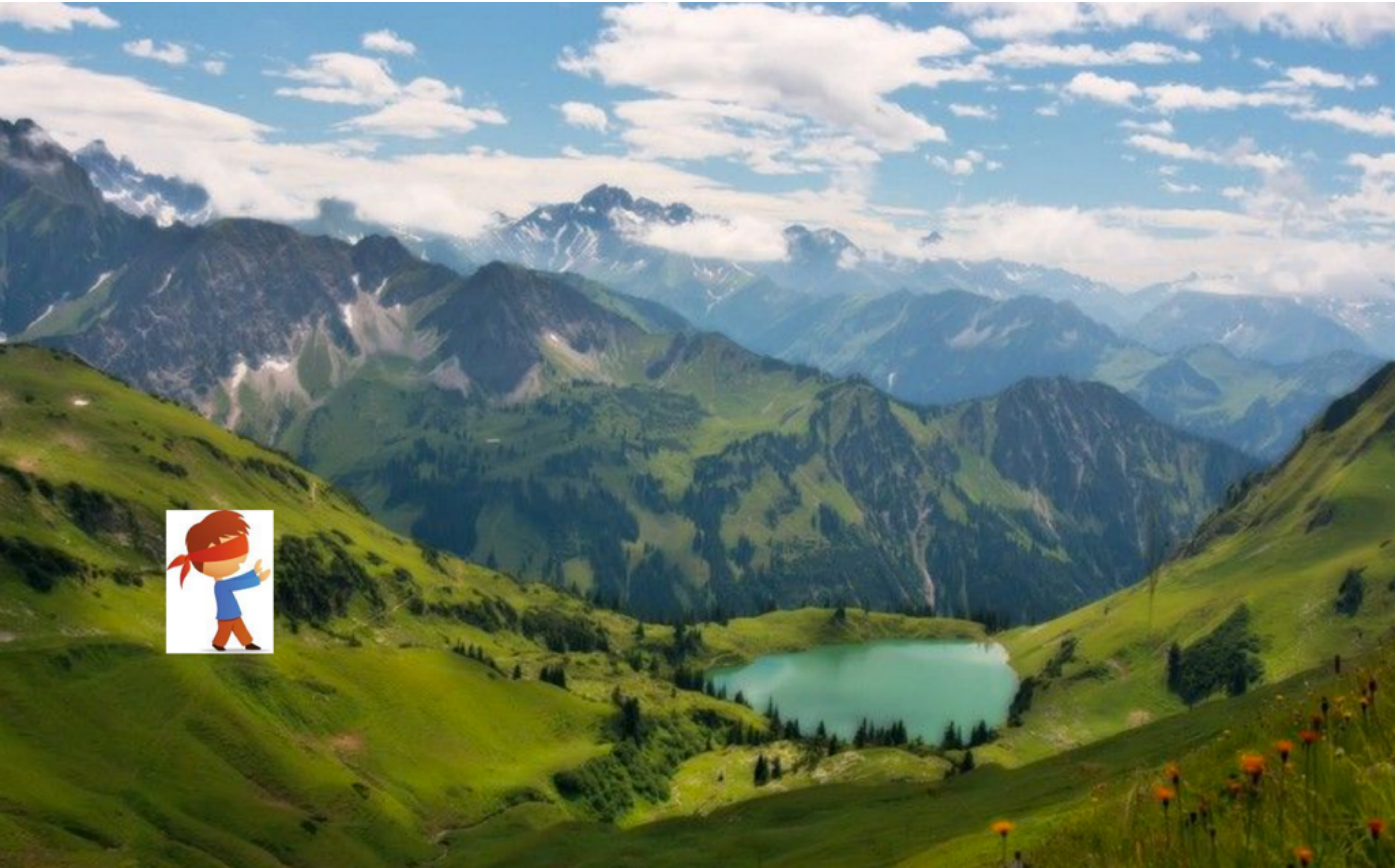
Idea #1: Random Search



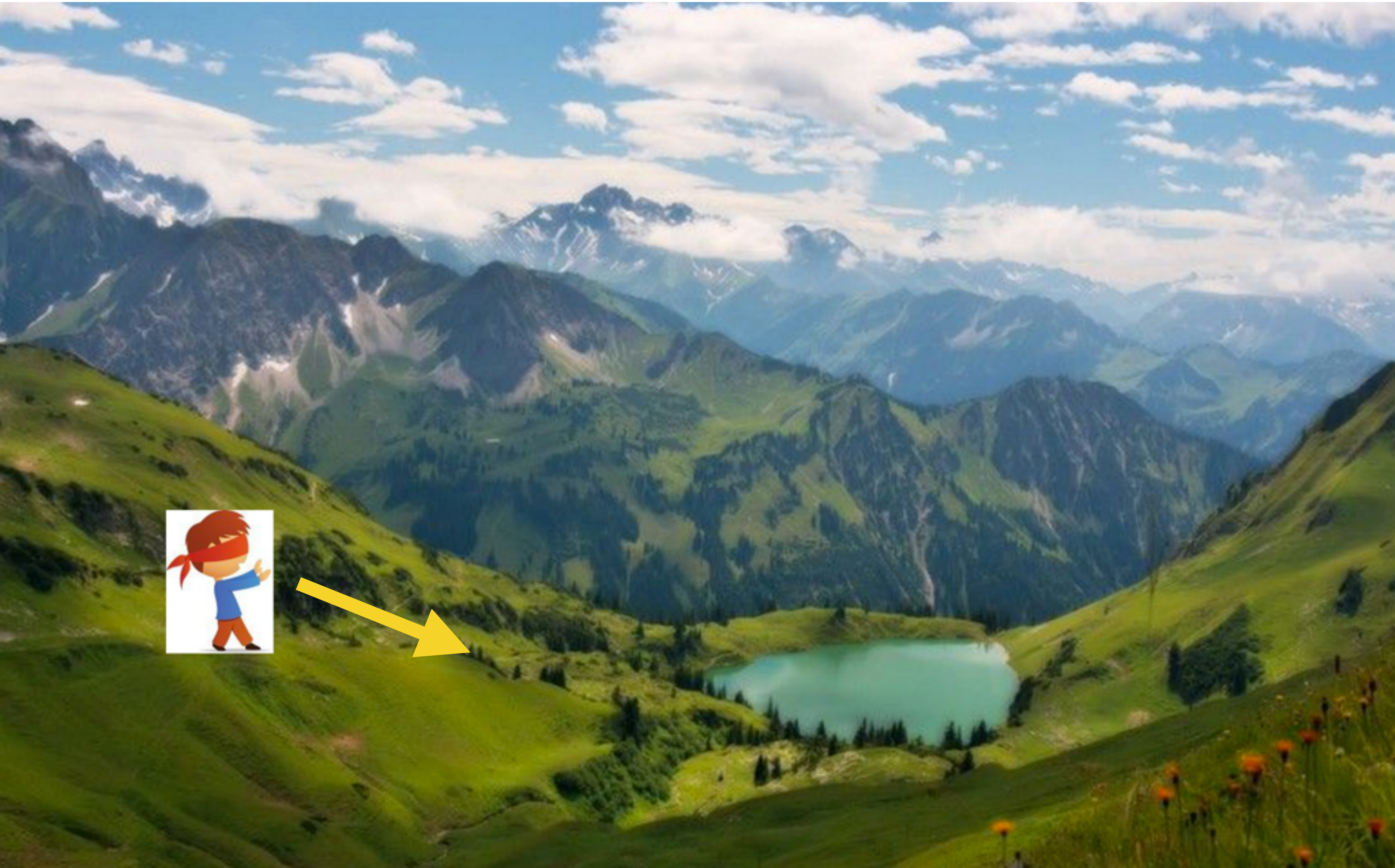
Idea #1: Random Search



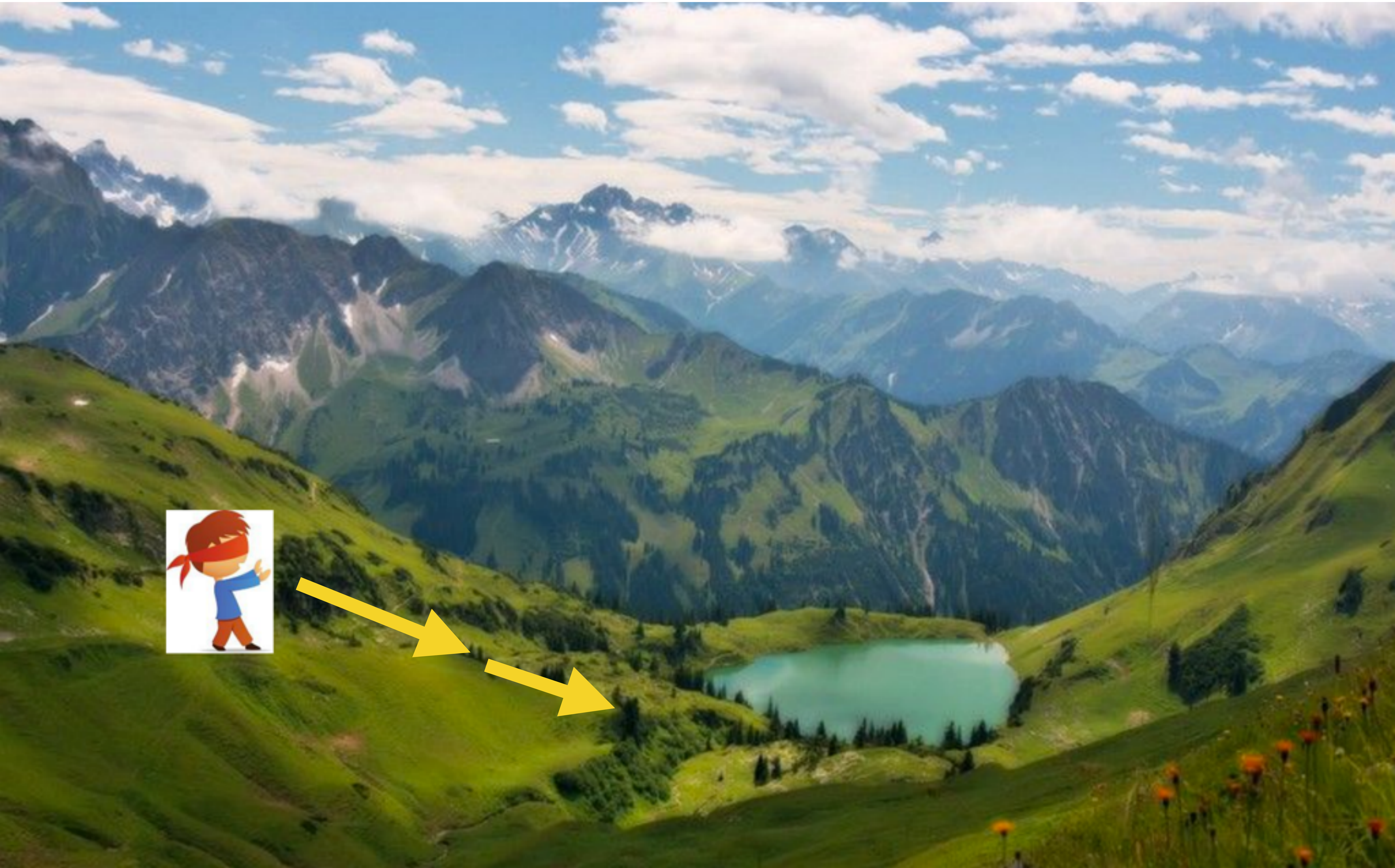
Idea #2: Follow the Slope



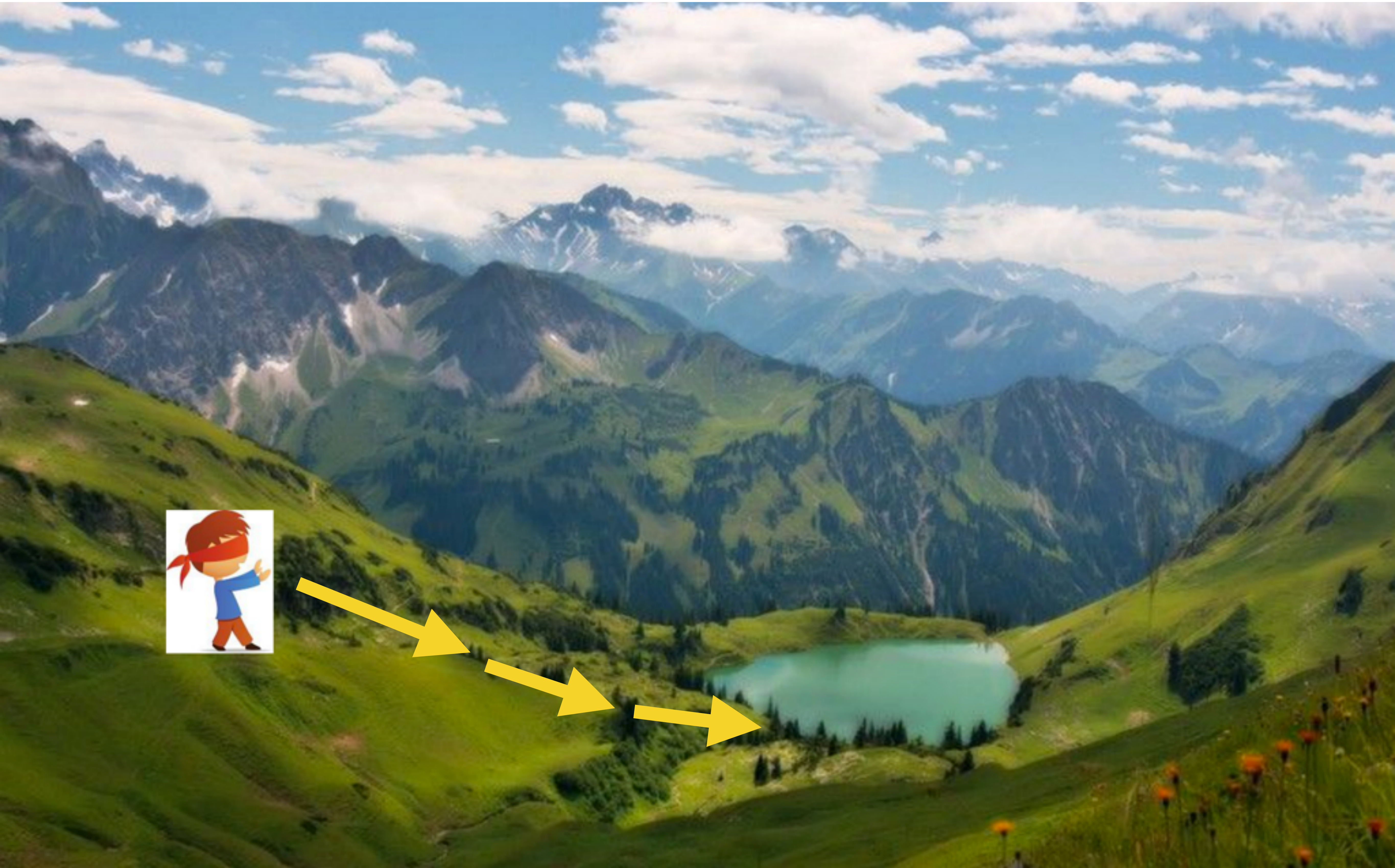
Idea #2: Follow the Slope



Idea #2: Follow the Slope



Idea #2: Follow the Slope



Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

Slide from Karpathy 2016

Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

We could directly estimate it: $$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Slide from Karpathy 2016

Idea #2: Follow the Slope

In 1D, the derivative of a function is:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives).

We could directly estimate it: $$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

But this is super slow.

Slide from Karpathy 2016

Idea #2: Follow the Slope

We can just directly compute the gradient:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Slide from Karpathy 2016

Idea #2: Follow the Slope

We can just directly compute the gradient:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Slide from Karpathy 2016

Idea #2: Follow the Slope

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

Slide from Karpathy 2016

Idea #2: Follow the Slope

In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

Slide from Karpathy 2016

Gradient Descent

Slide from Karpathy 2016

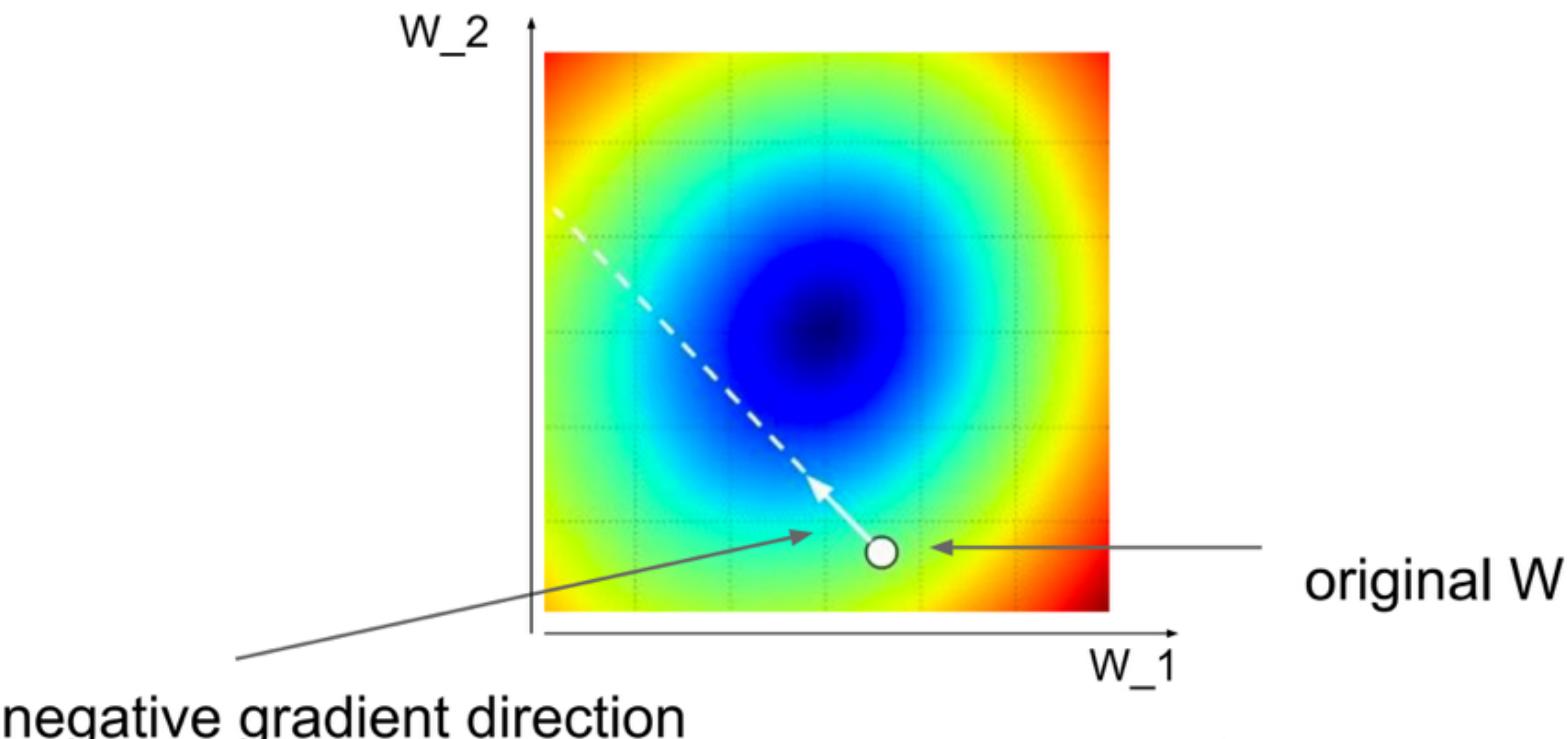
Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Slide from Karpathy 2016

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

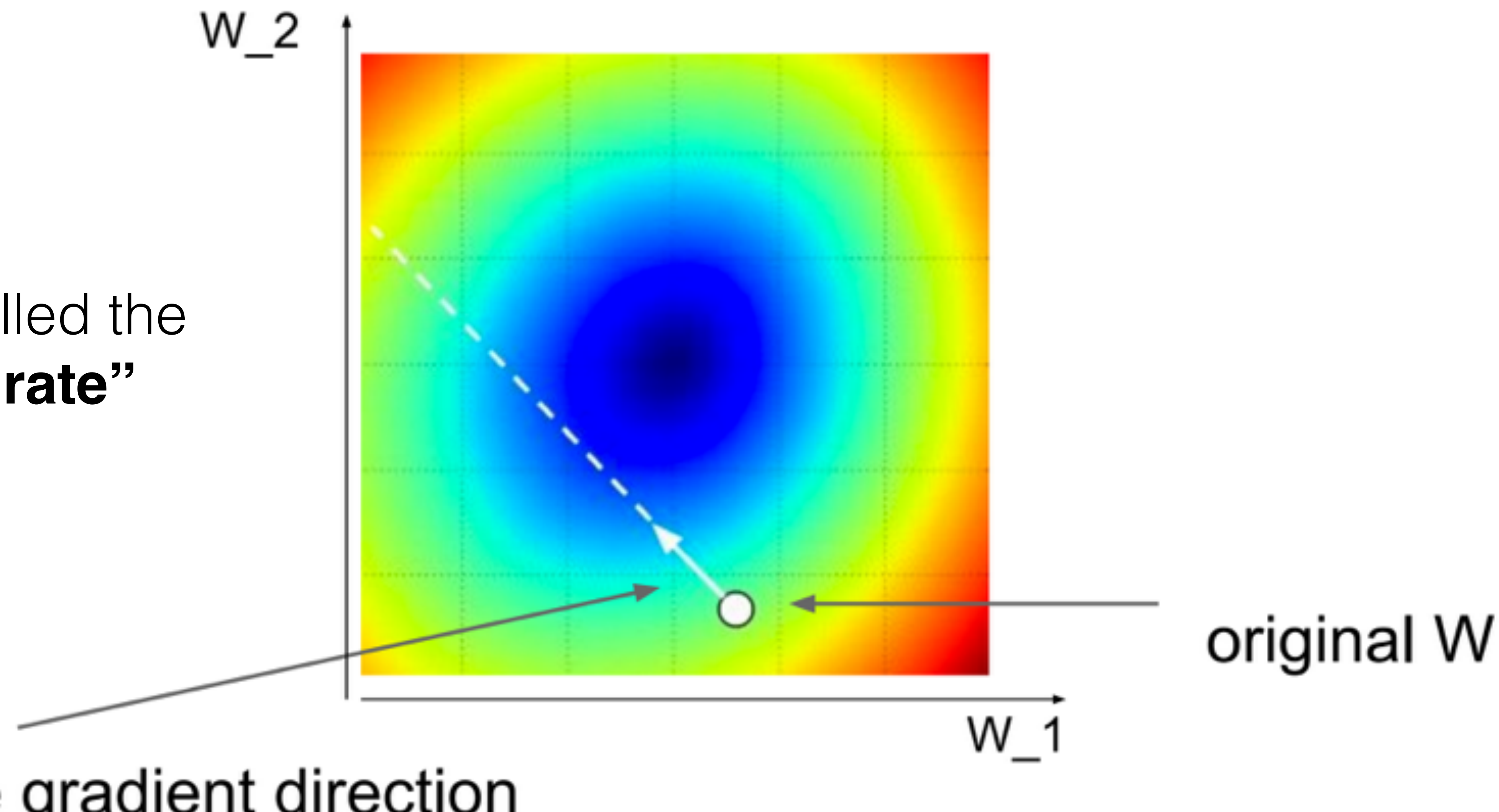


Slide from Karpathy 2016

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Step size called the
“learning rate”



Slide from Karpathy 2016

Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

Slide from Karpathy 2016

Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

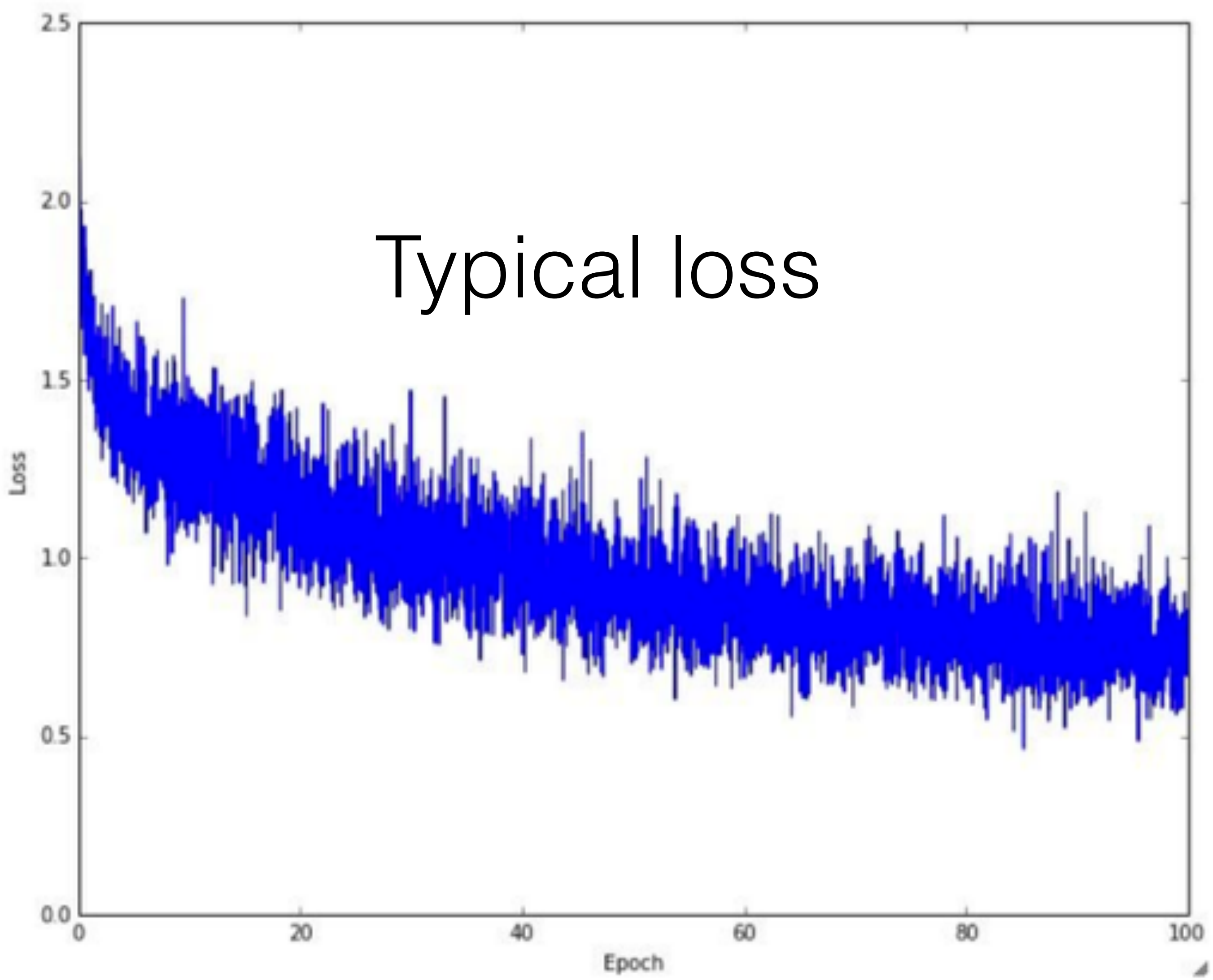
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

Note: In practice we use fancier update rules (momentum, RMSprop, ADAM, etc)

Slide from Karpathy 2016

Mini-batch Gradient Descent



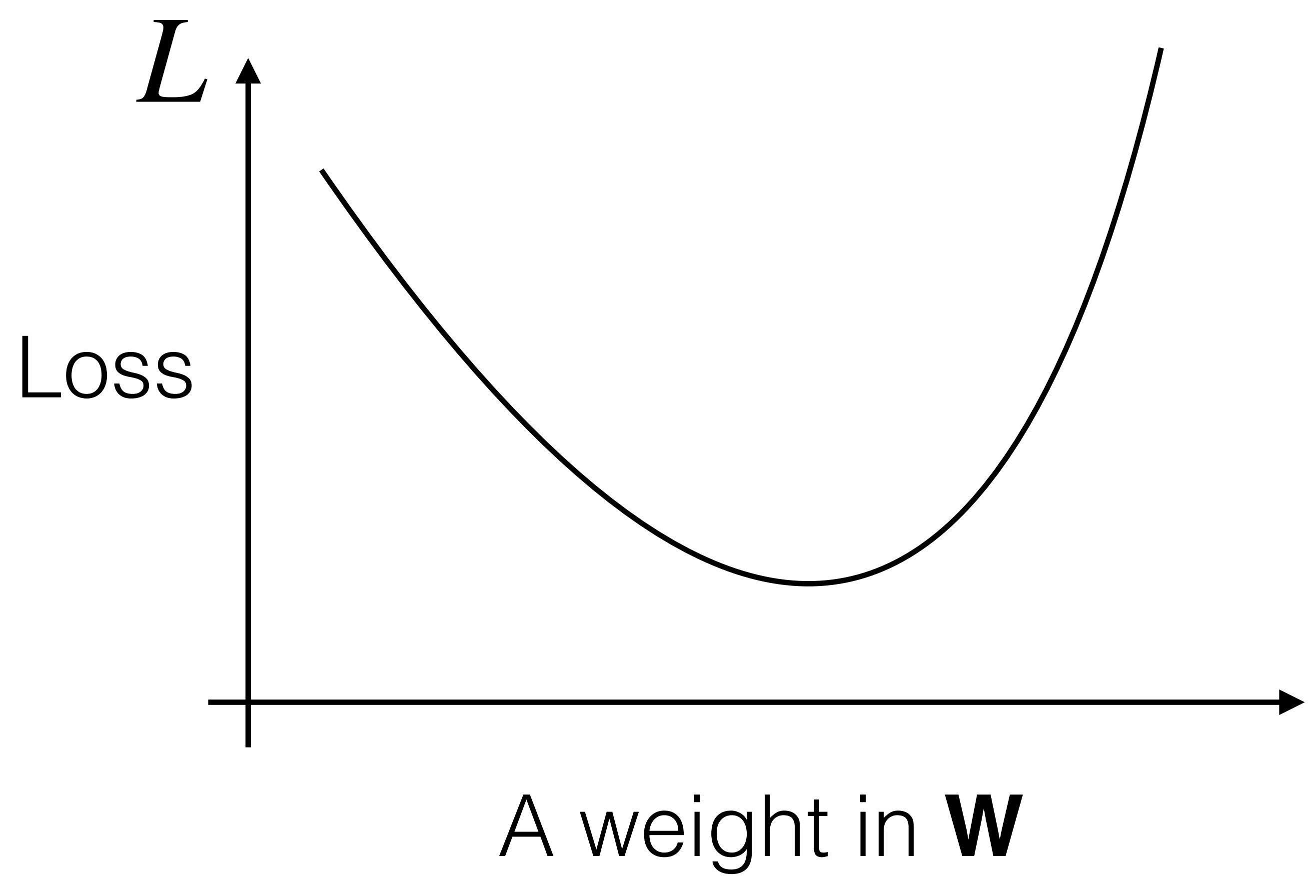
Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)

Slide from Karpathy 2016

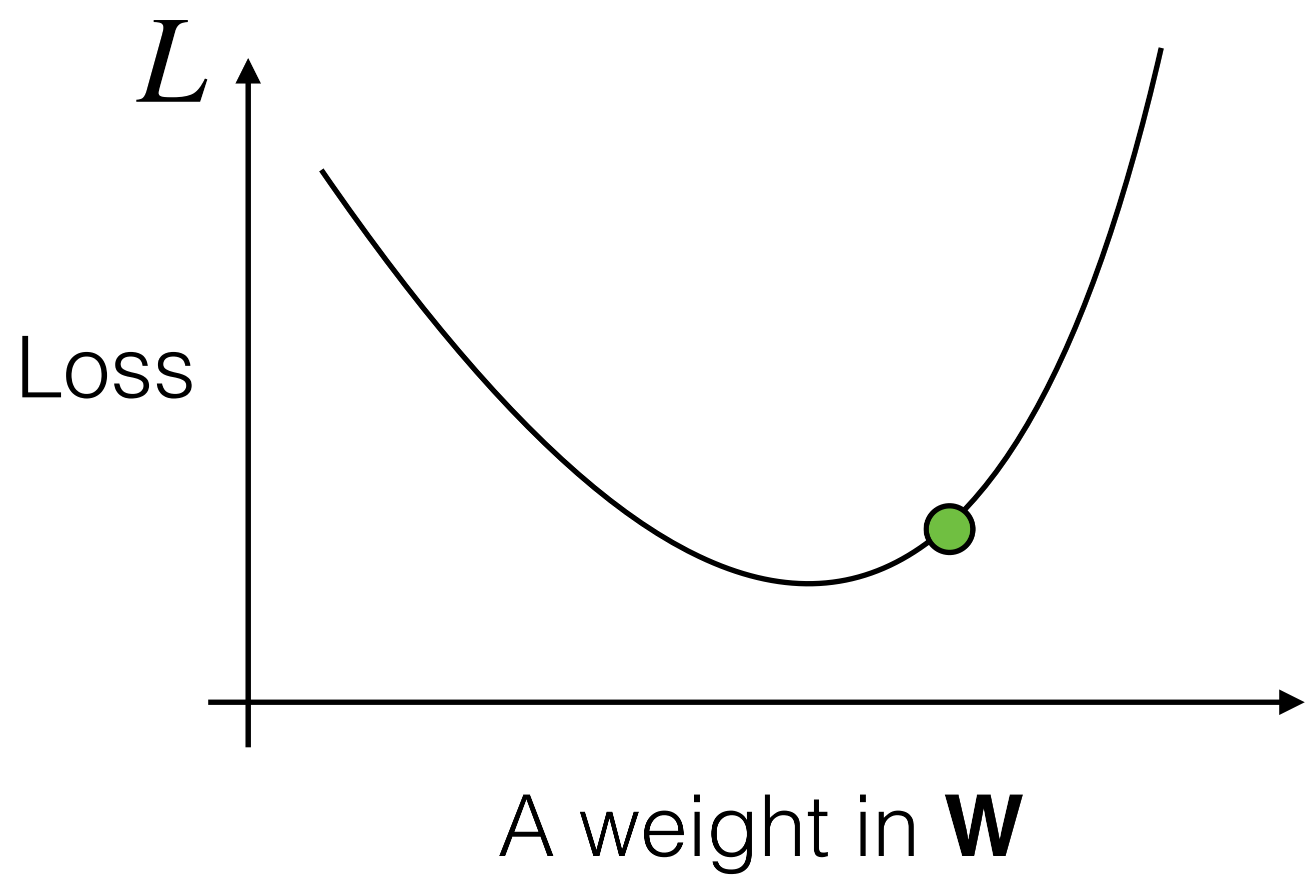
Setting the learning rate

Learning rate: 1e6 — what could go wrong?



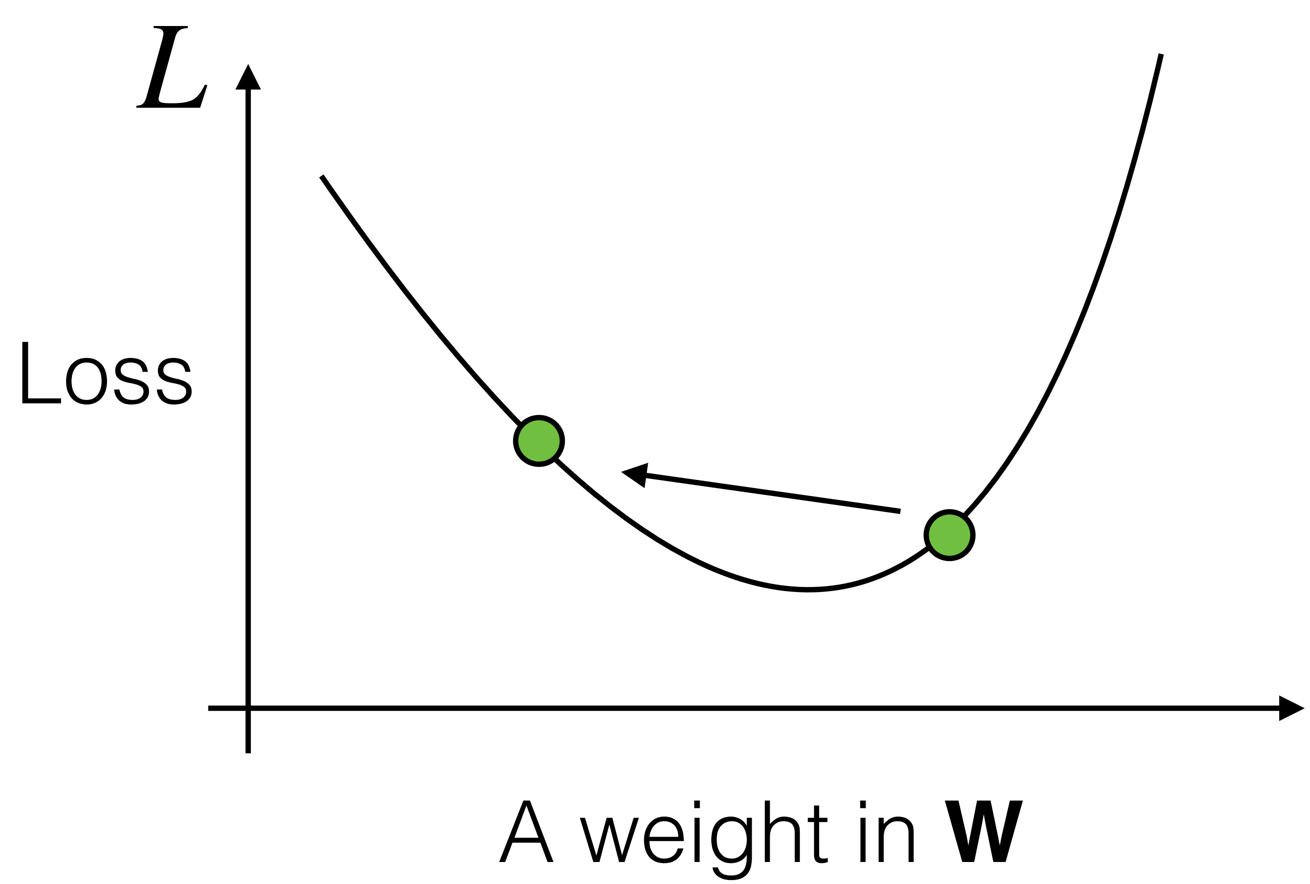
Setting the learning rate

Learning rate: 1e6 — what could go wrong?



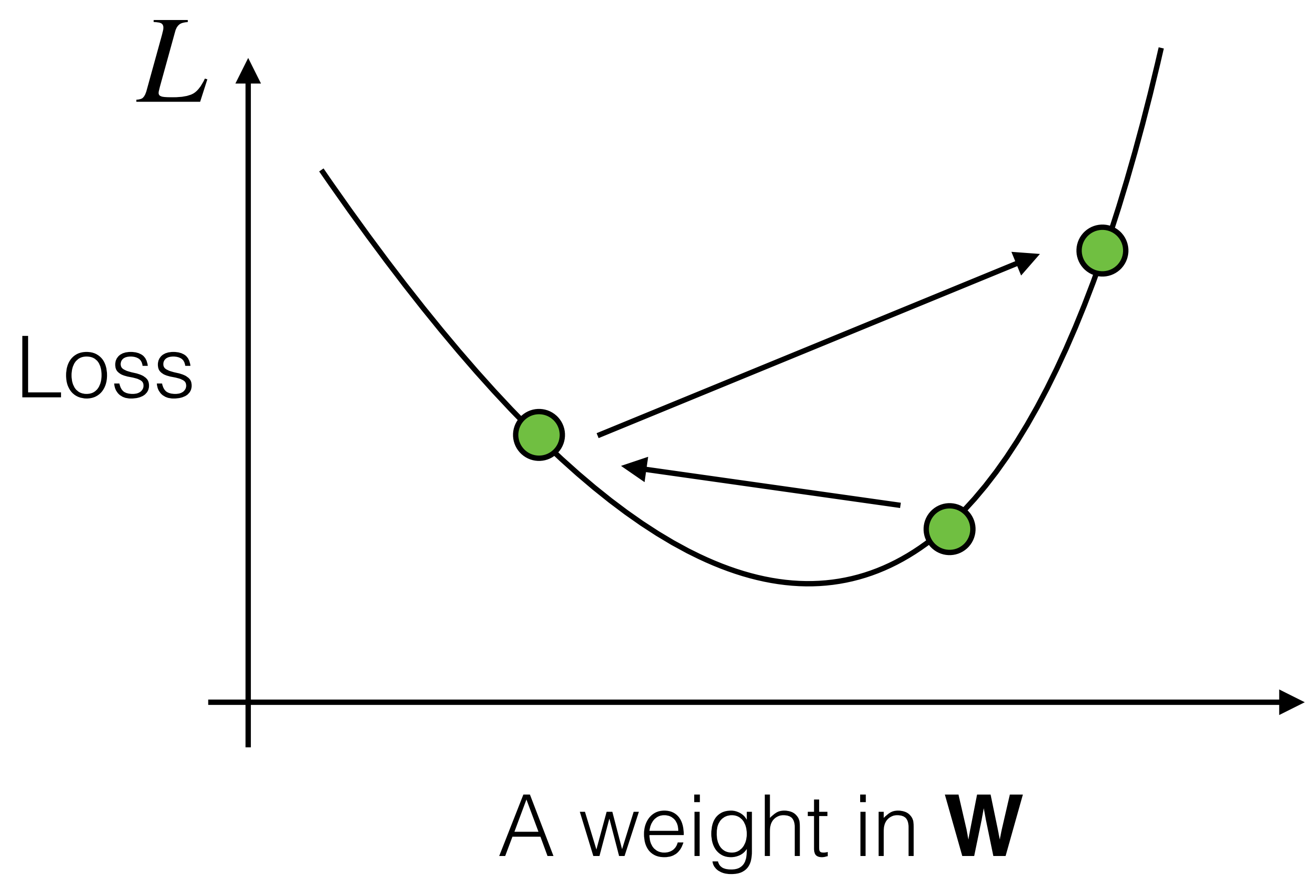
Setting the learning rate

Learning rate: 1e6 — what could go wrong?



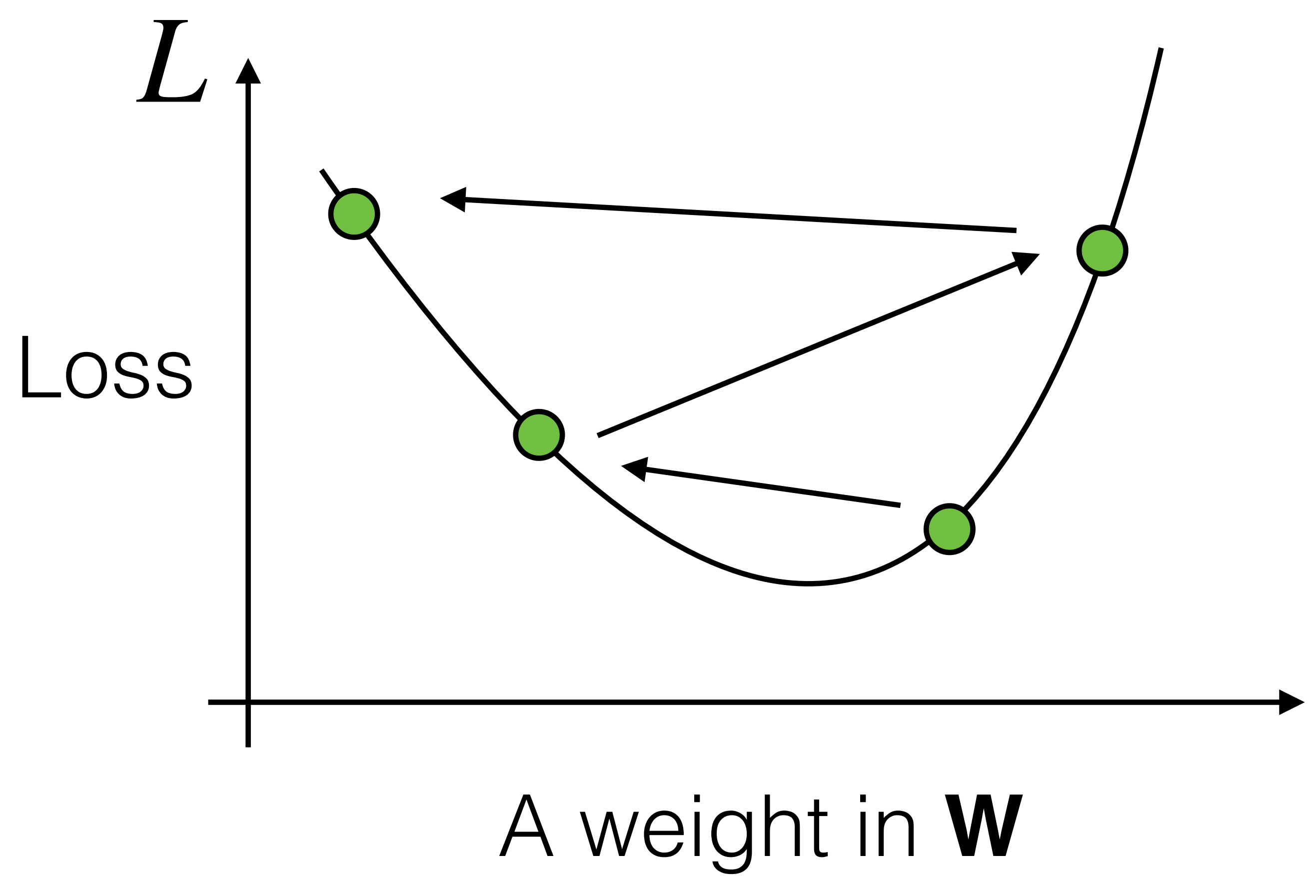
Setting the learning rate

Learning rate: 1e6 — what could go wrong?



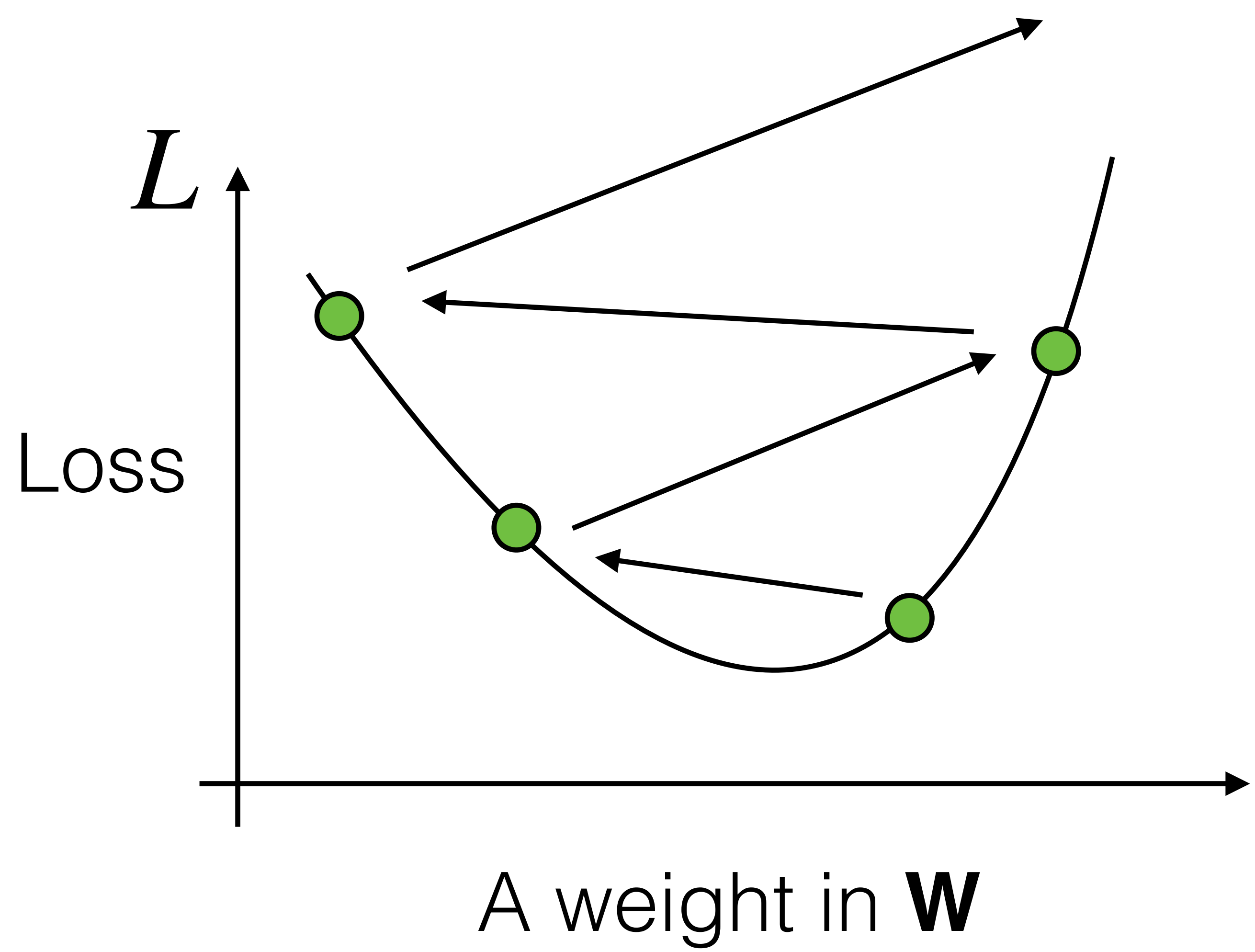
Setting the learning rate

Learning rate: 1e6 — what could go wrong?

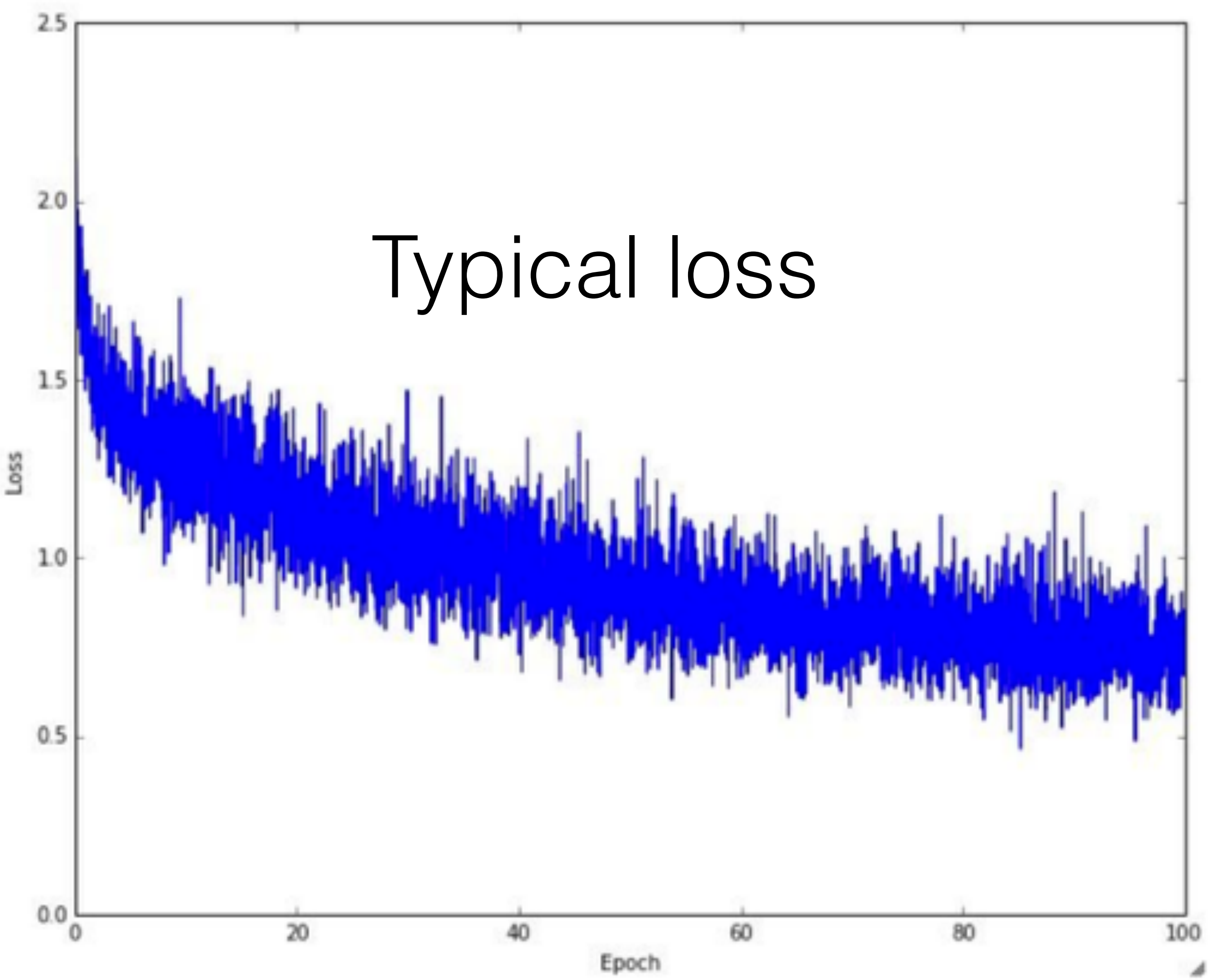


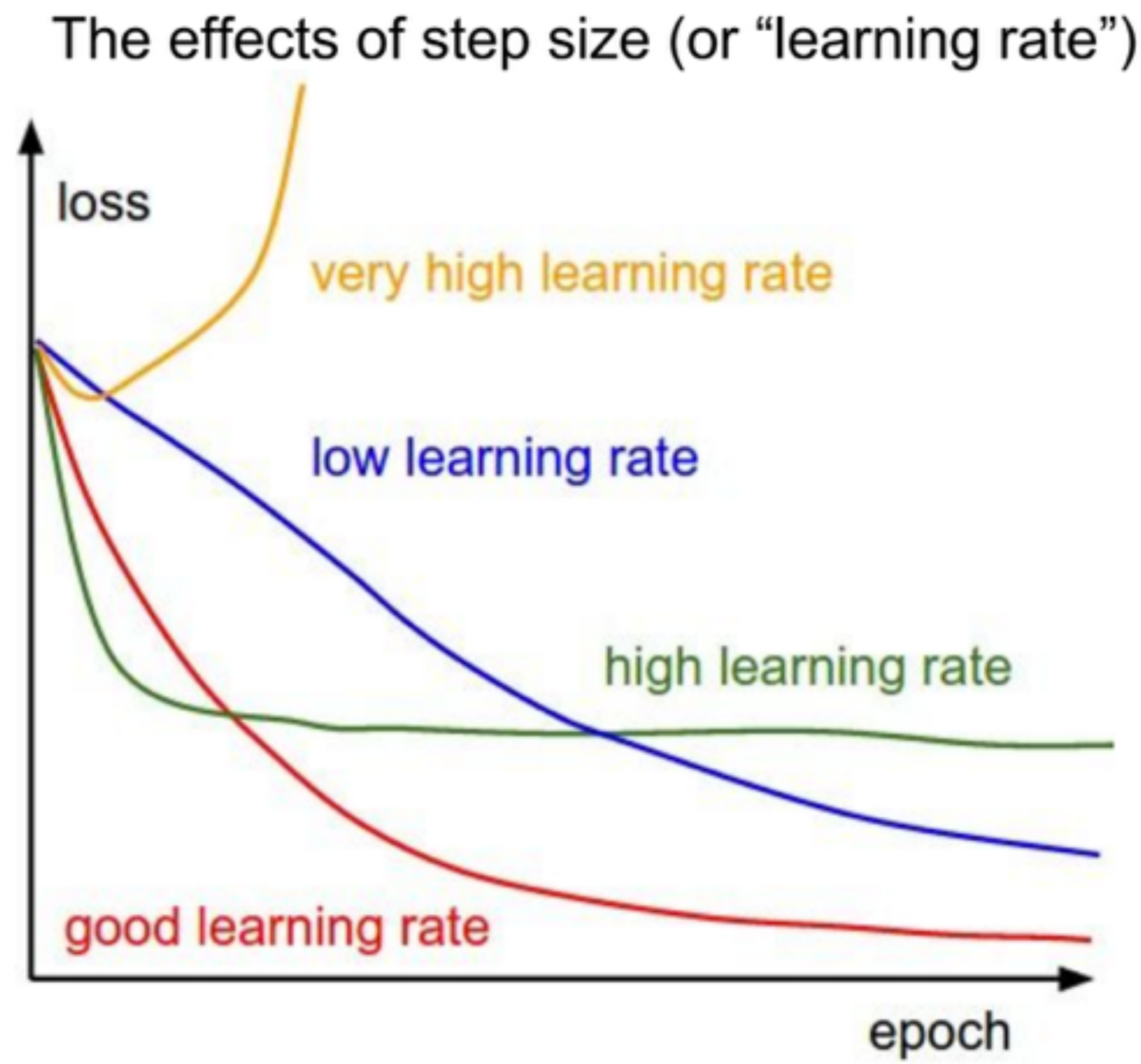
Setting the learning rate

Learning rate: 1e6 — what could go wrong?



Setting the learning rate





(more on this later)

Slide from Karpathy 2016

Classification: Overview

Training Images

Test Image

Classification: Overview

Training Images

Gradient Descent

Test Image

Classification: Overview

Training Labels

Training Images

Gradient Descent

Test Image

Classification: Overview

Training Labels

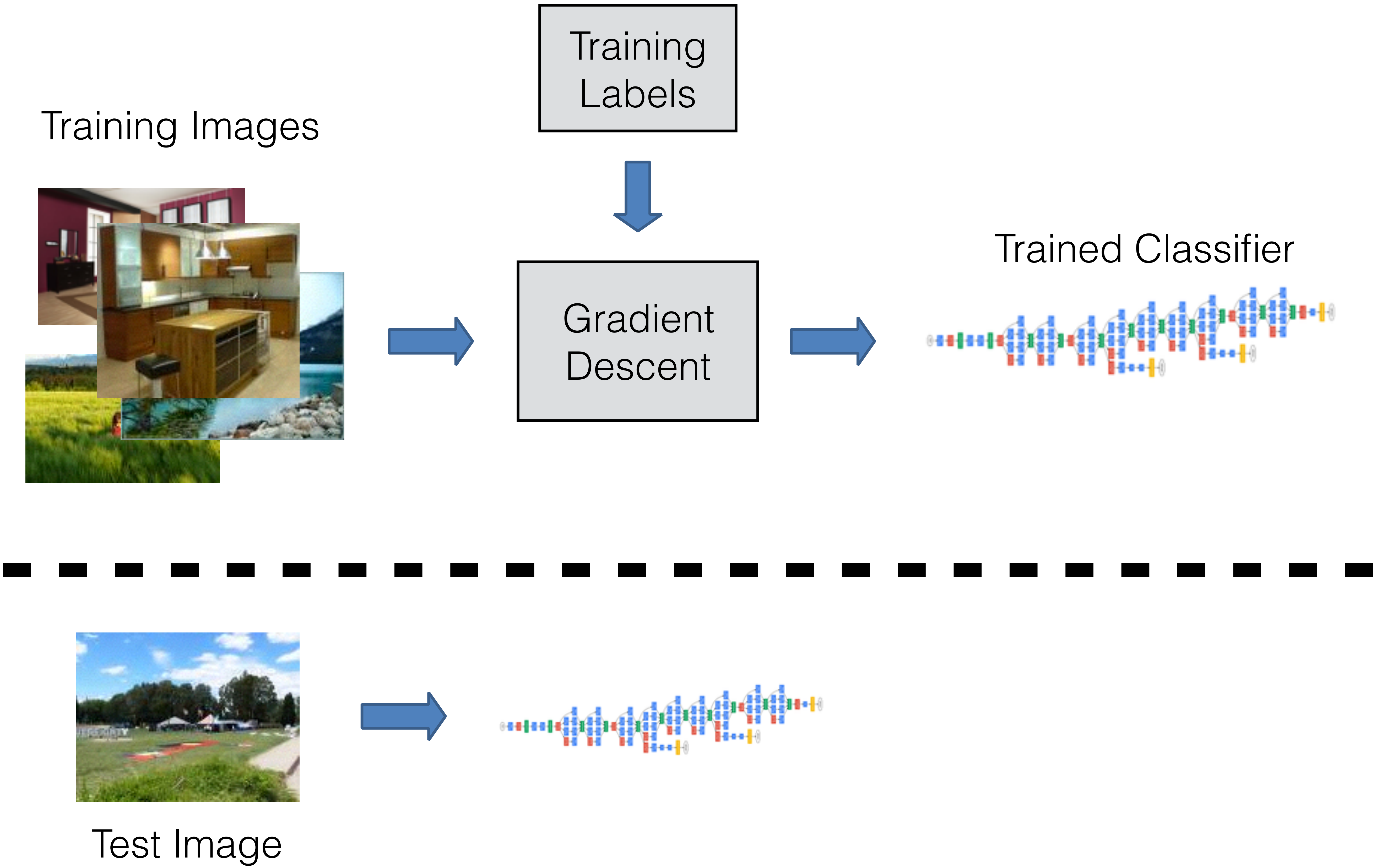
Training Images

Gradient Descent

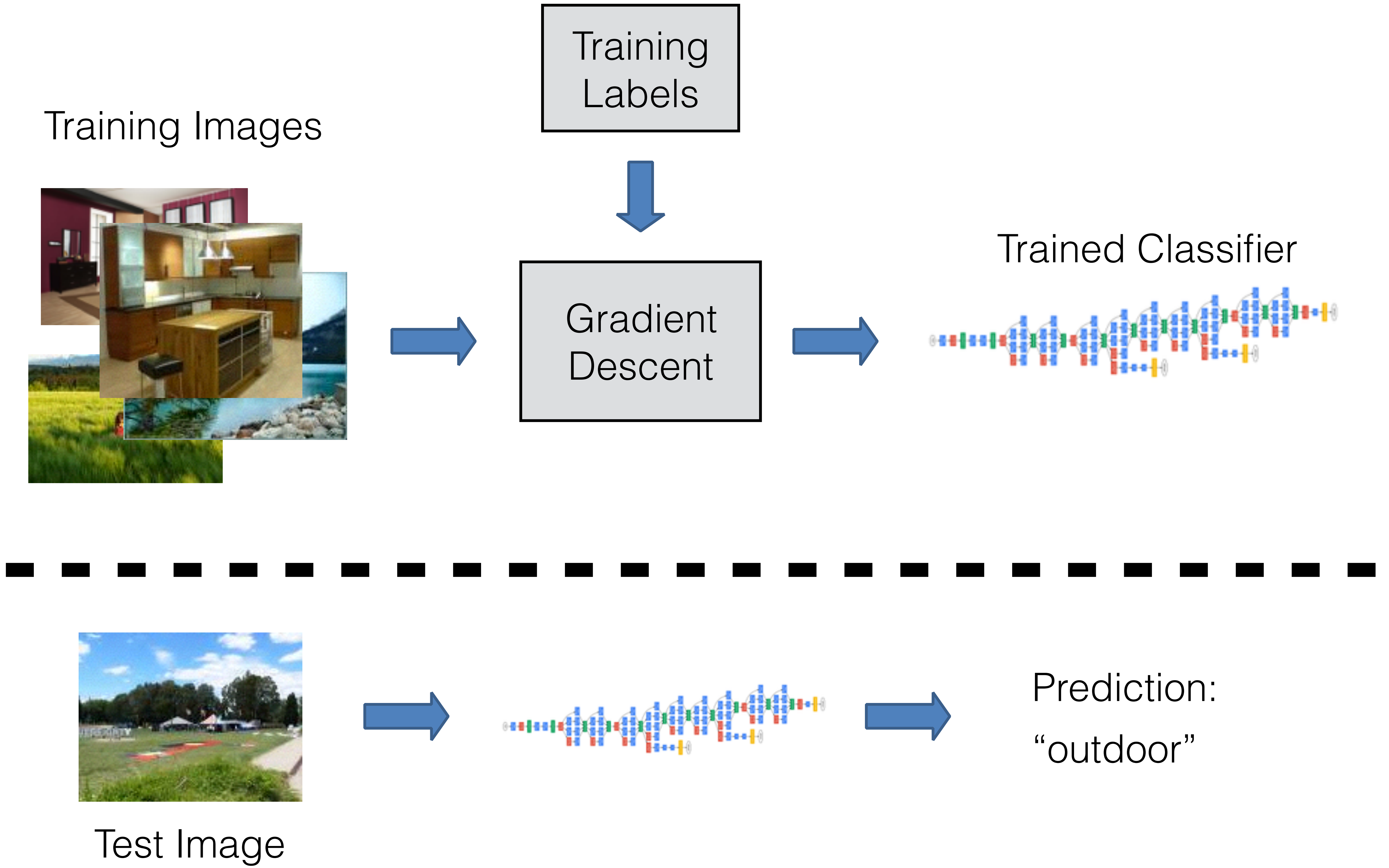
Trained Classifier

Test Image

Classification: Overview



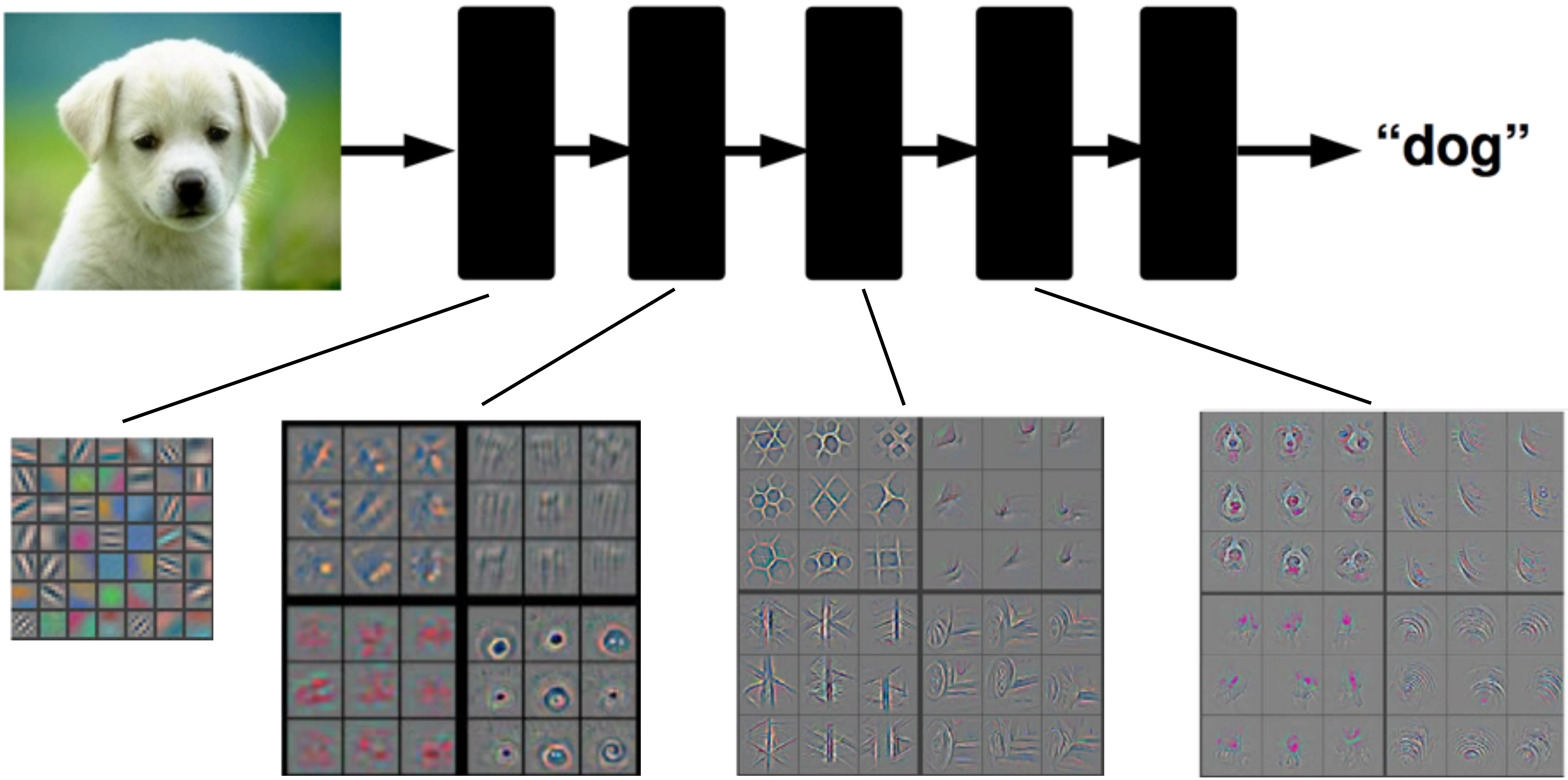
Classification: Overview



Neural Networks

(First we'll cover Neural Nets, then build up to Convolutional Neural Nets)

Feature hierarchy with ConvNets
End-to-end models



[Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", 2013]

Learning Feature Hierarchy

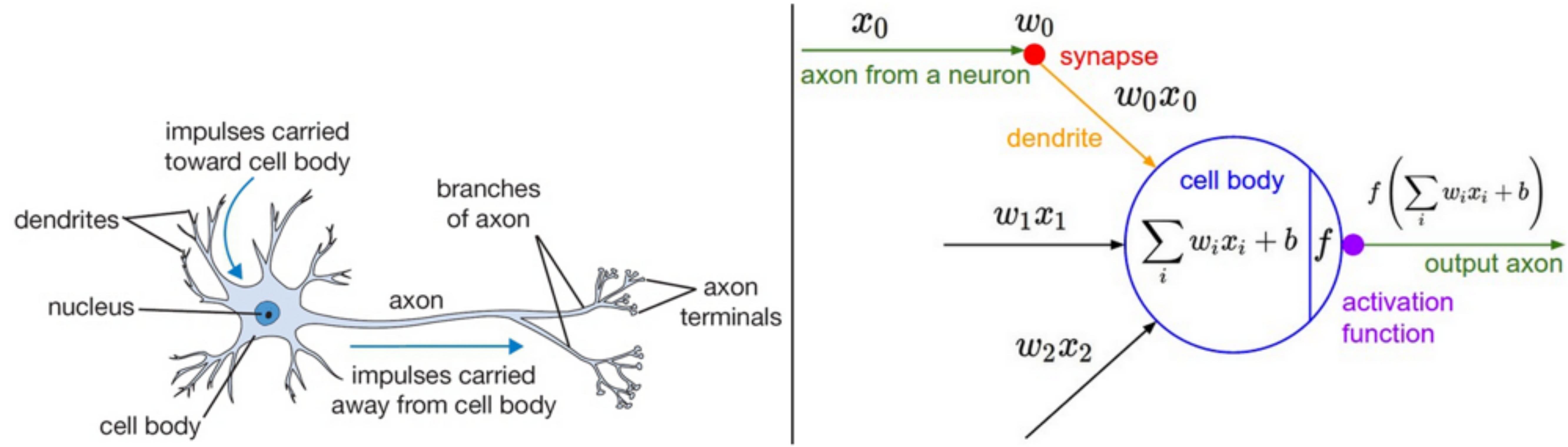
- Learn hierarchy
- All the way from pixels → classifier
- One layer extracts features from output of previous layer

Slide: R. Fergus

Inspiration from Biology

Figure: Andrej Karpathy

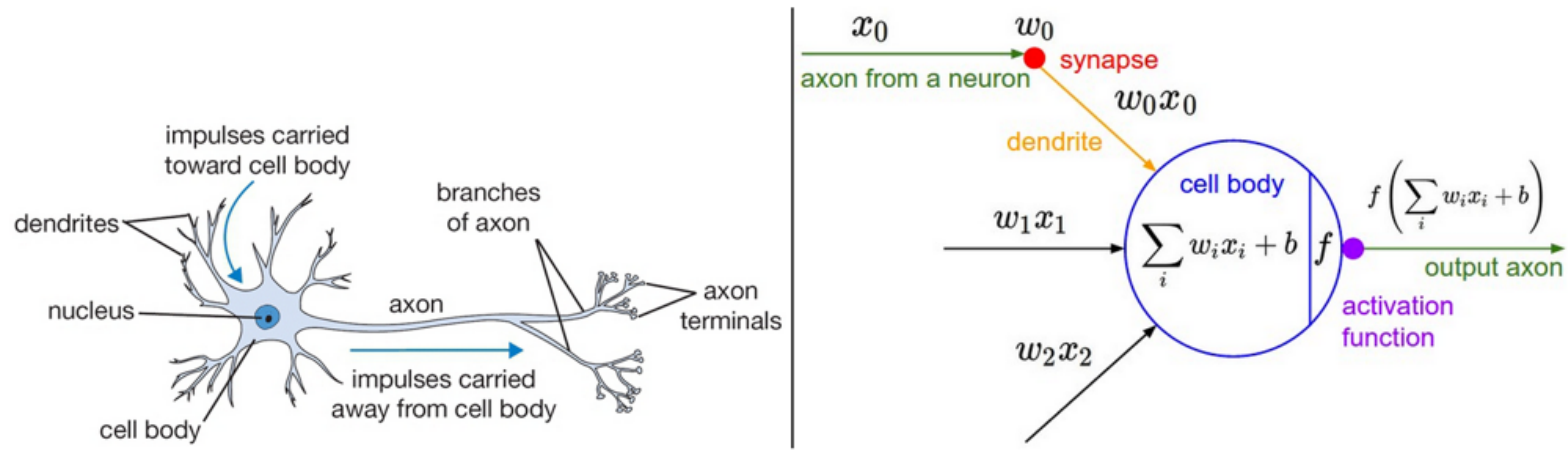
Inspiration from Biology



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figure: Andrej Karpathy

Inspiration from Biology

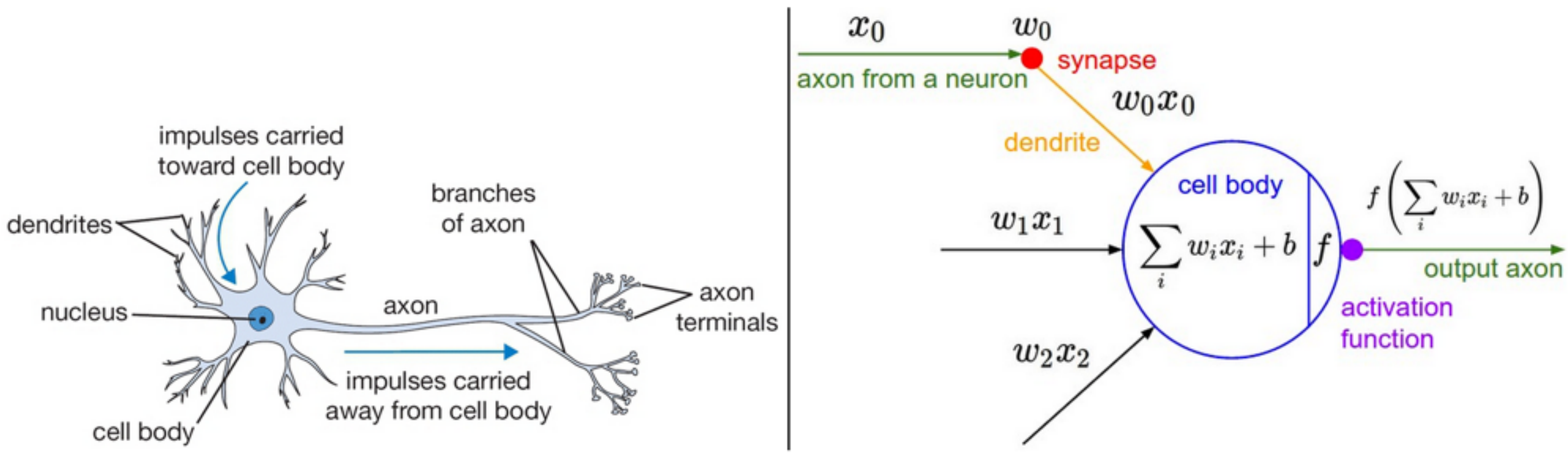


A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Neural nets are **loosely inspired** by biology

Figure: Andrej Karpathy

Inspiration from Biology



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Neural nets are **loosely inspired** by biology

But they certainly are **not** a model of how the brain works, or even how neurons work

Figure: Andrej Karpathy

Simple Neural Net: 1 Layer

Let’s consider a simple 1-layer network:

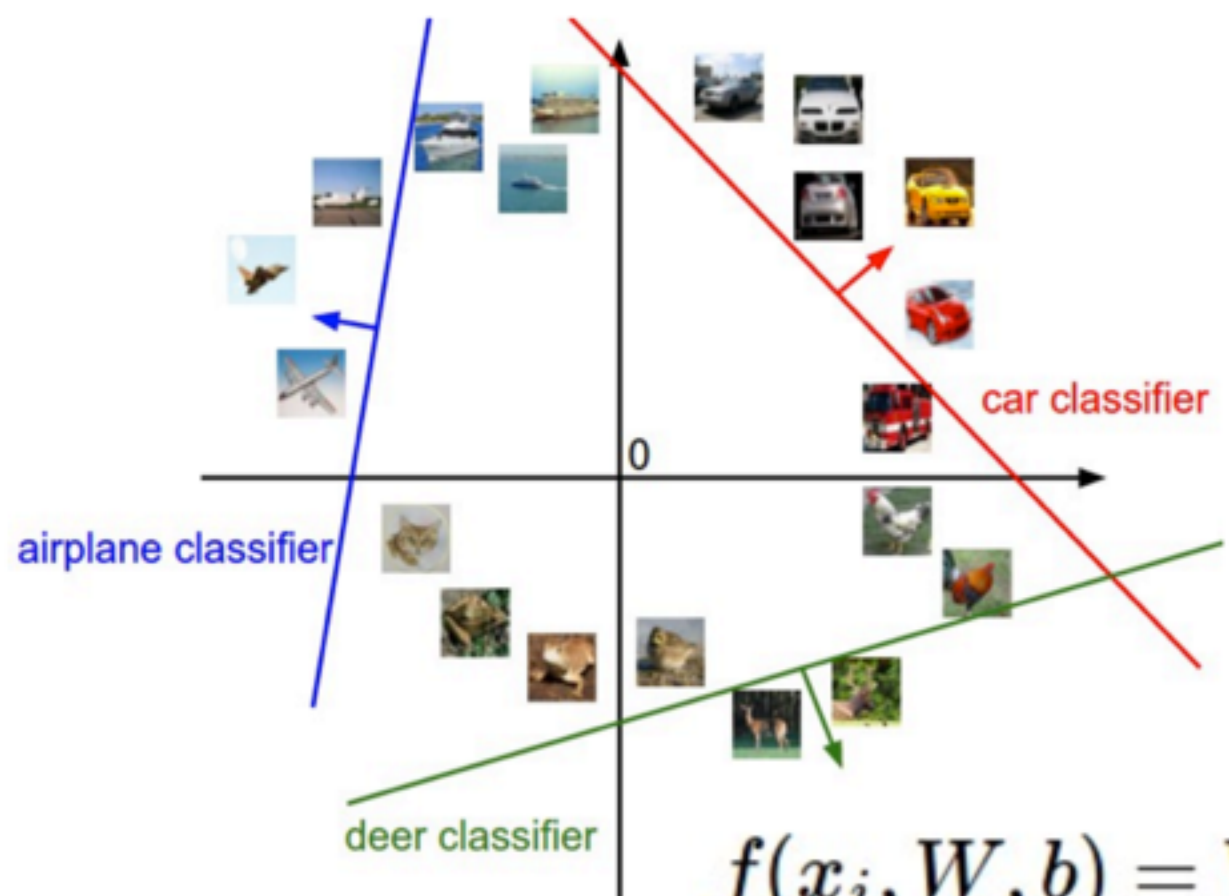
$$x \rightarrow \boxed{Wx+b} \rightarrow f$$

Simple Neural Net: 1 Layer

Let's consider a simple 1-layer network:

$$x \rightarrow \boxed{Wx + b} \rightarrow f$$

This is the same as what you saw last class:



$$f(x_i, W, b) = Wx_i + b$$

1 Layer Neural Net

Block Diagram:

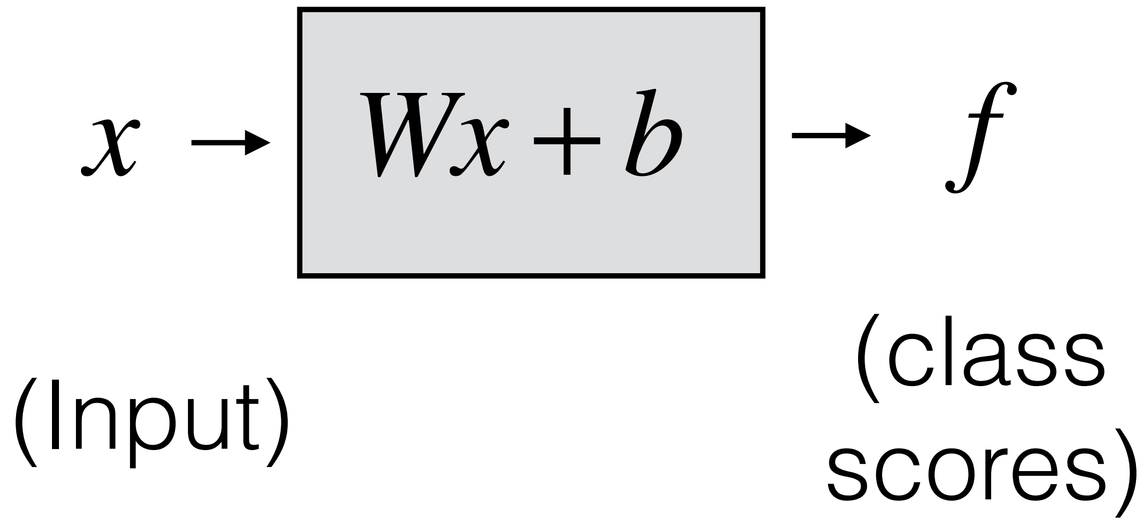
$$x \rightarrow \boxed{Wx+b} \rightarrow f$$

(Input)

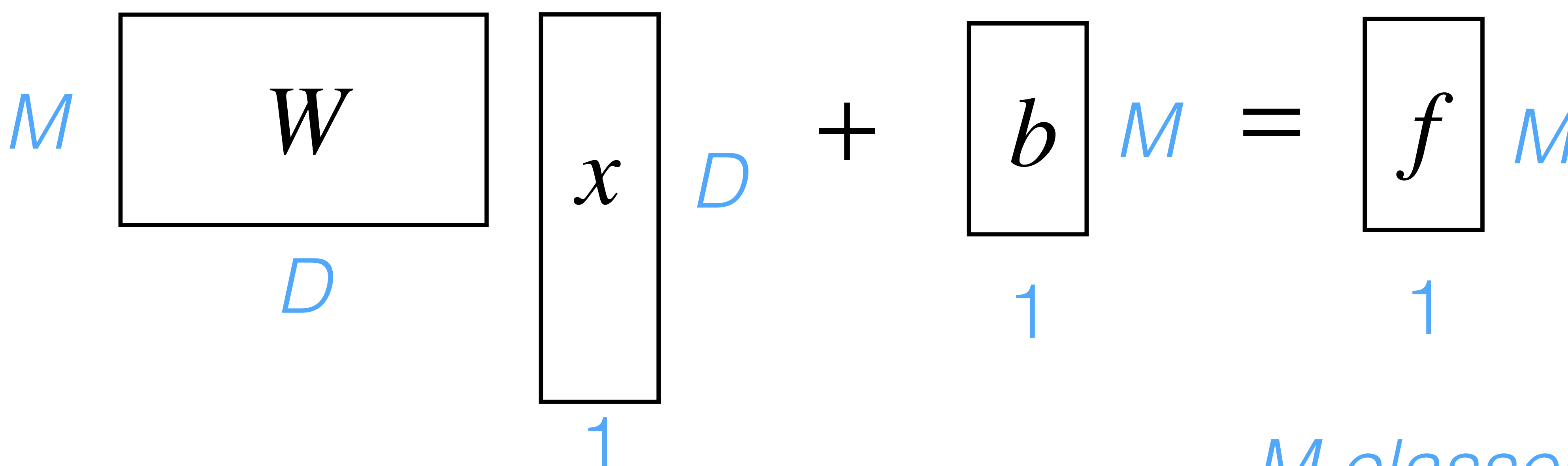
(class scores)

1 Layer Neural Net

Block Diagram:



Expanded Block Diagram:



M classes
D features
1 example

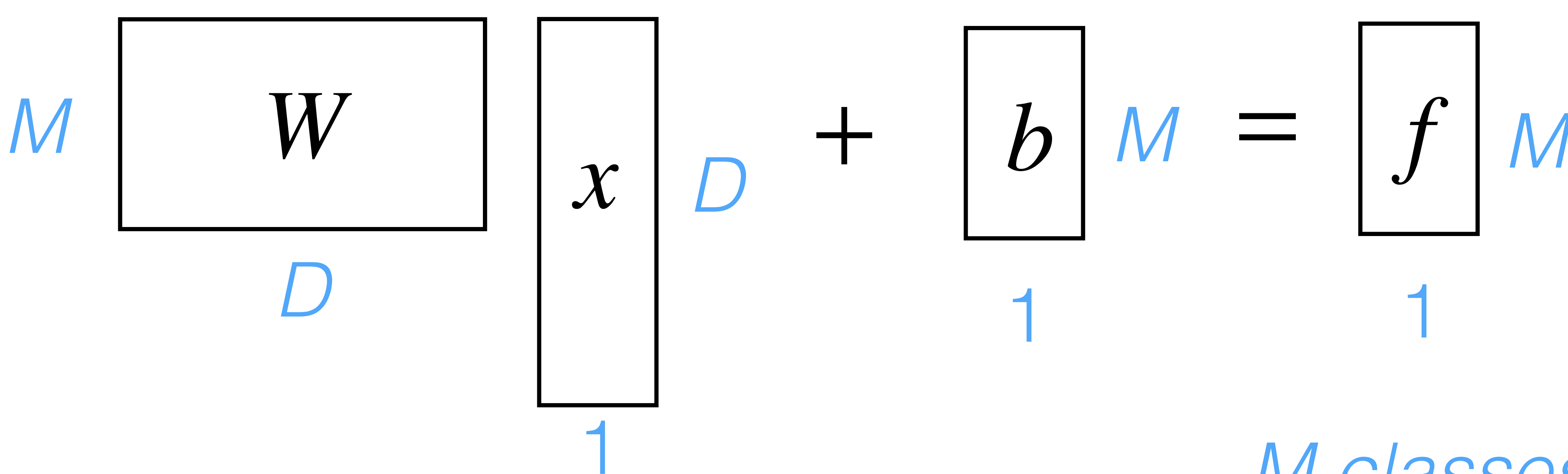
1 Layer Neural Net

Block Diagram:

$$x \rightarrow \boxed{Wx+b} \rightarrow f$$

(Input) (class scores)

Expanded Block Diagram:



M classes
D features
1 example

NumPy:

```
f = np.dot(W, x) + b
```

1 Layer Neural Net

1 Layer Neural Net

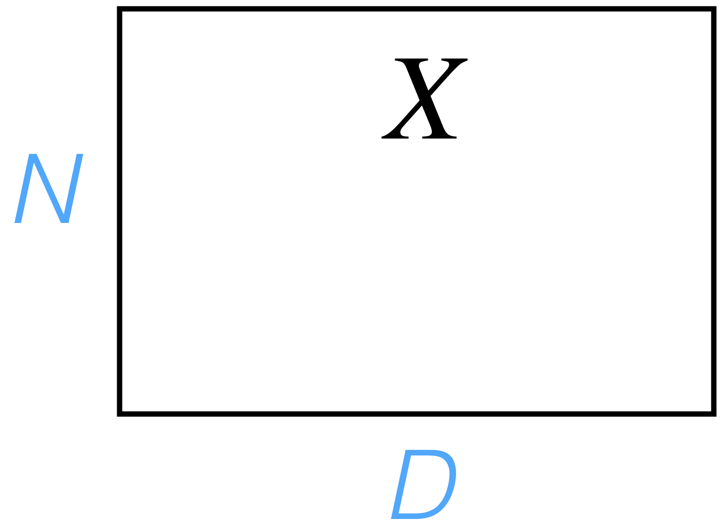
- How do we process *N* inputs at once?

1 Layer Neural Net

- How do we process *N* inputs at once?
- It’s most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything

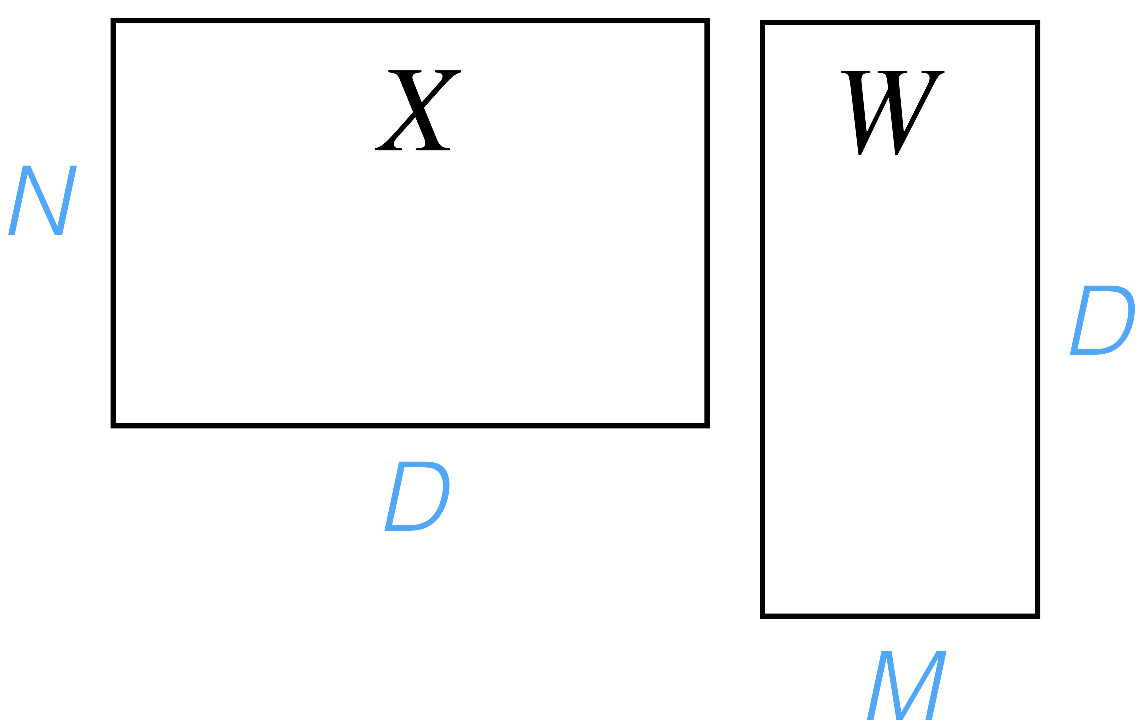
1 Layer Neural Net

- How do we process *N* inputs at once?
- It's most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything



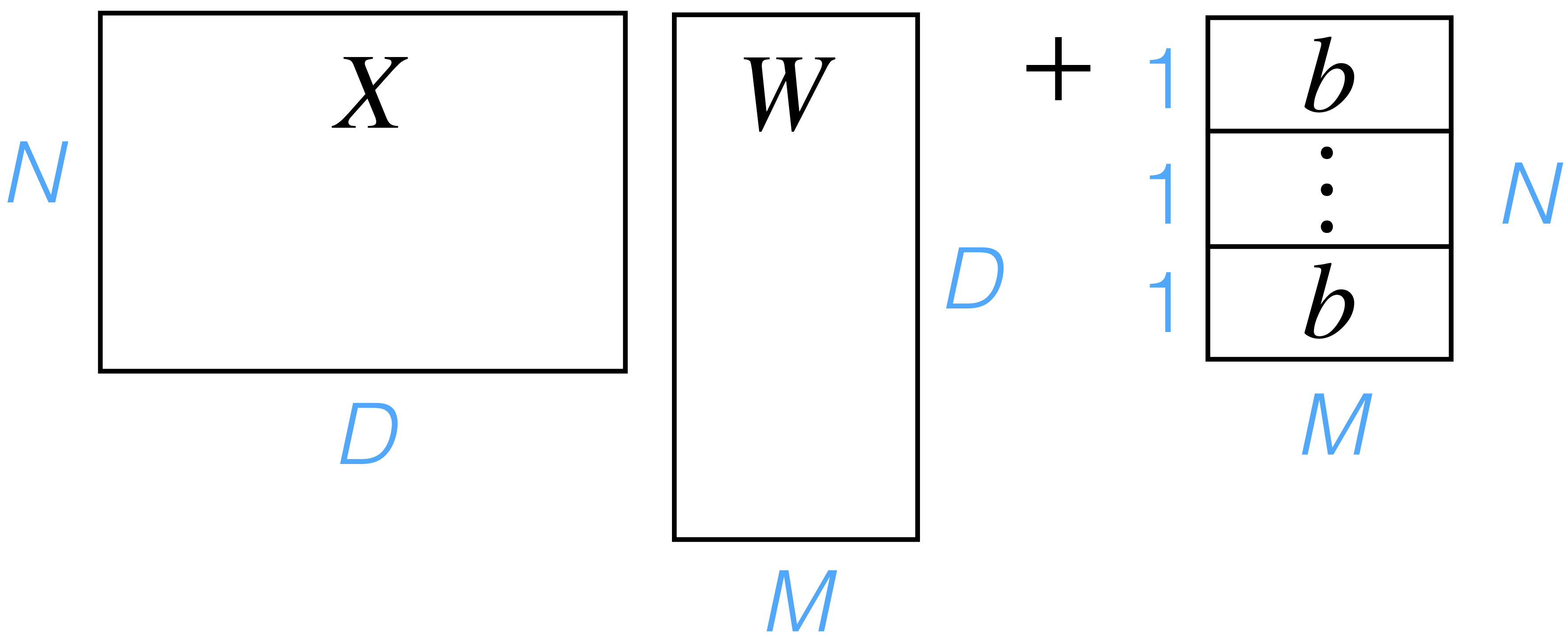
1 Layer Neural Net

- How do we process *N* inputs at once?
- It's most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything



1 Layer Neural Net

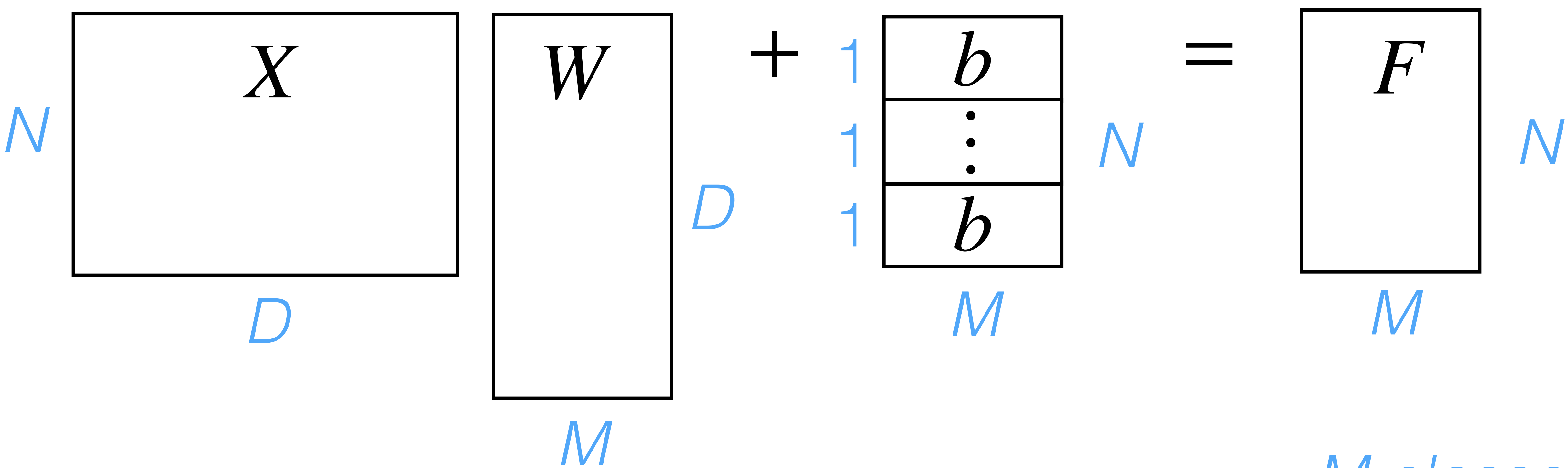
- How do we process *N* inputs at once?
- It's most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything



1 Layer Neural Net

- How do we process *N* inputs at once?

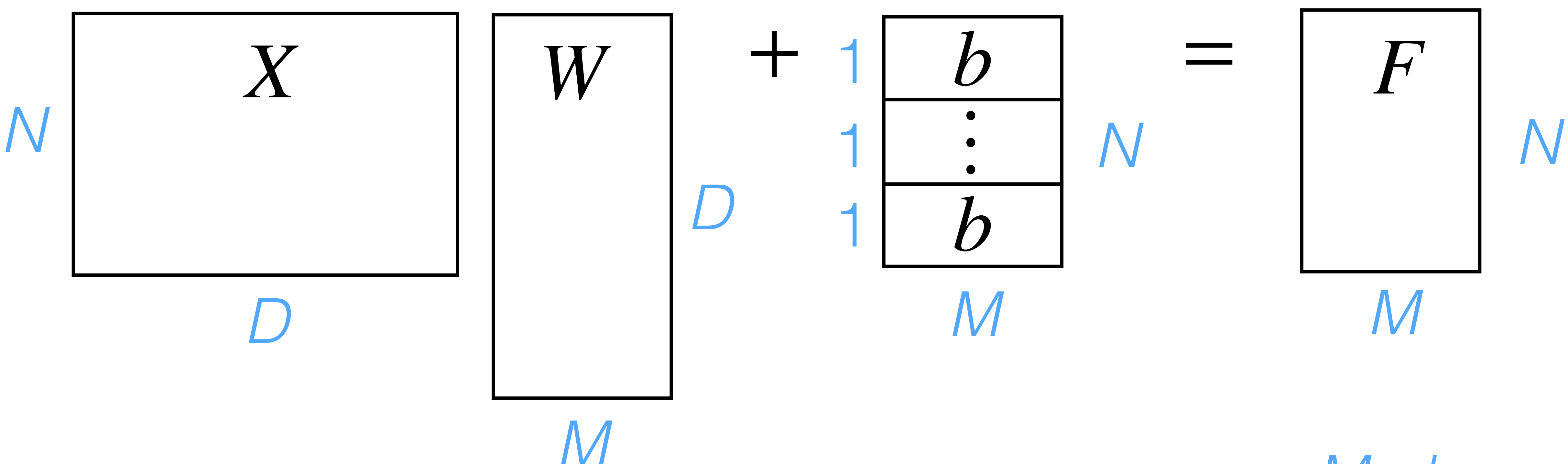
- It's most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything



M classes
D features
N examples

1 Layer Neural Net

- How do we process *N* inputs at once?
- It's most convenient to have the first dimension (row) represent which example we are looking at, so we need to transpose everything

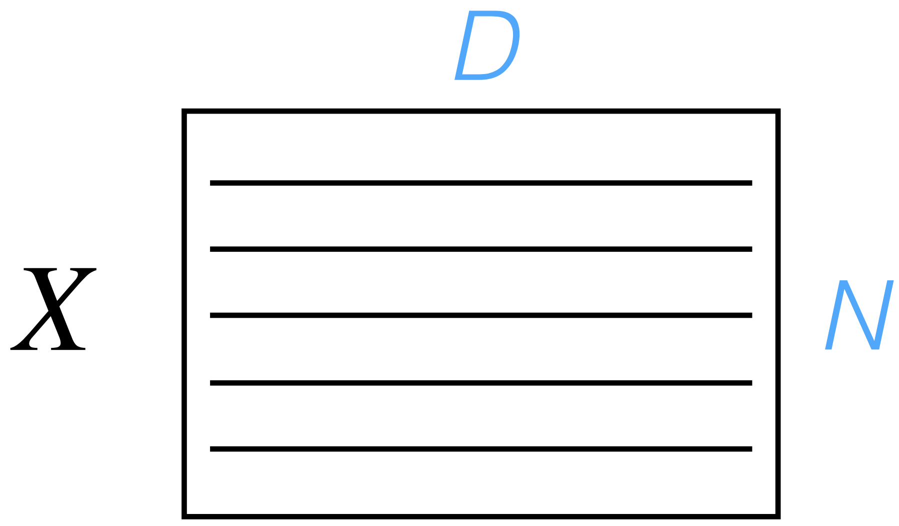


Note: Often, if the weights are transposed, they are still called "W"

M classes
D features
N examples

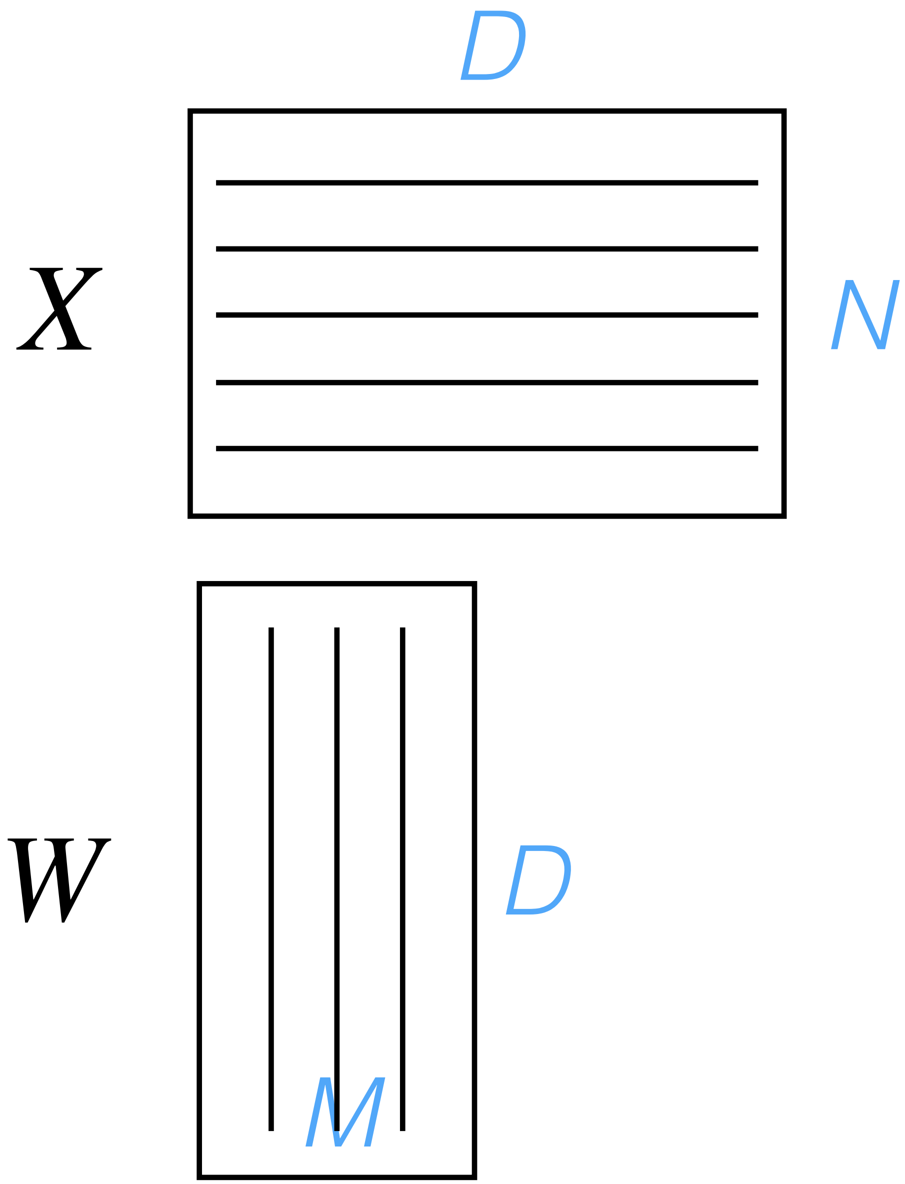
1 Layer Neural Net

1 Layer Neural Net



Each **row** is one input example

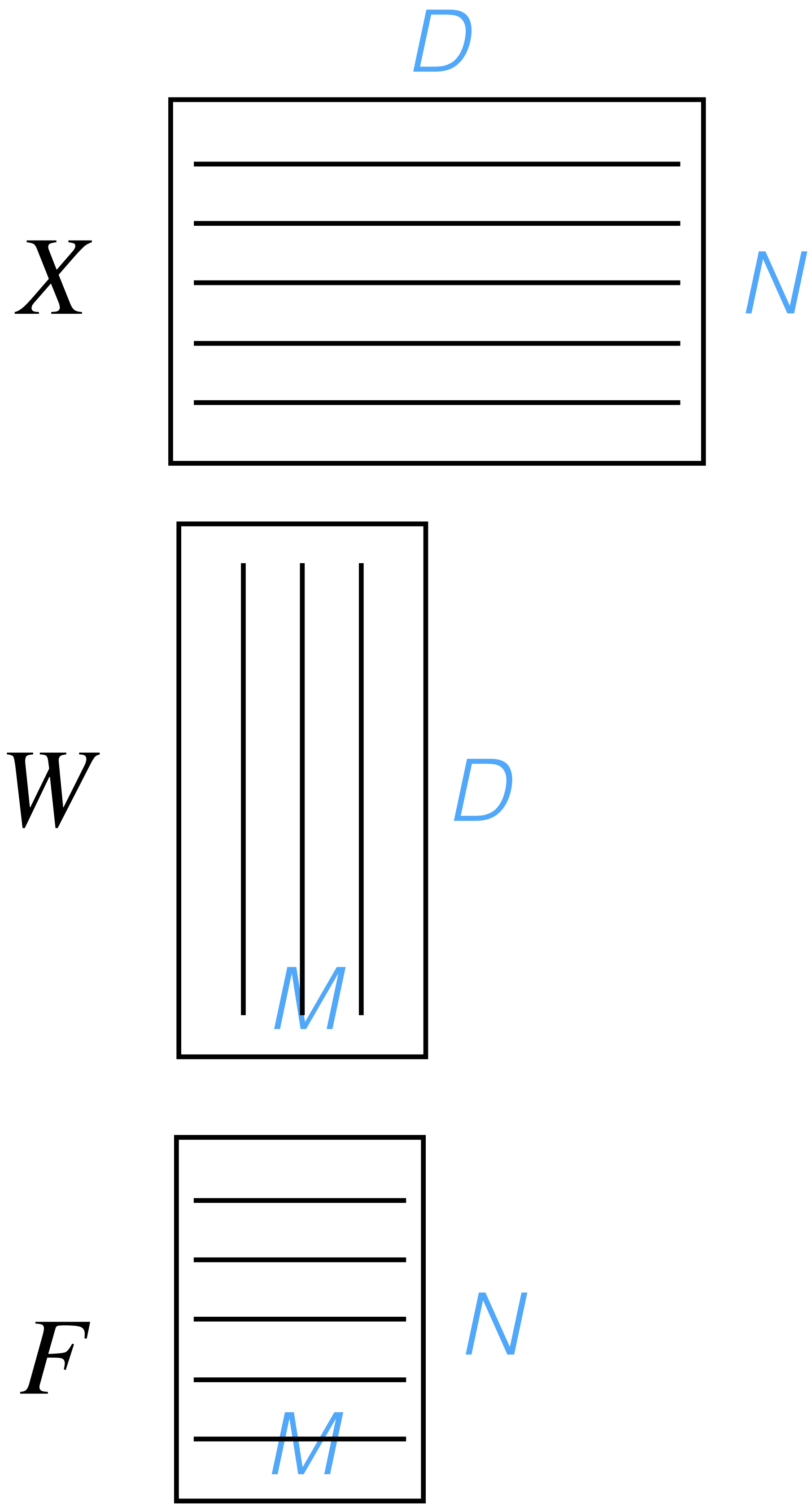
1 Layer Neural Net



Each **row** is one input example

Each **column** is the weights for one class

1 Layer Neural Net



Each **row** is one input example

Each **column** is the weights for one class

Each **row** is the predicted scores for one example

1 Layer Neural Net

Implementing this with NumPy:

First attempt — let's try this:

```
F = np.dot(X, W) + b
```

1 Layer Neural Net

Implementing this with NumPy:

First attempt — let's try this:

```
F = np.dot(X, W) + b
```

Doesn't work — why?

1 Layer Neural Net

Implementing this with NumPy:

First attempt — let's try this:

N x D M

```
F = np.dot(X, W) + b
```

D x M

Doesn't work — why?

1 Layer Neural Net

Implementing this with NumPy:

First attempt — let's try this:

N x D   M

```
F = np.dot(X, W) + b
```

D x M

Doesn't work — why?

- NumPy needs to know how to expand "b" from 1D to 2D

1 Layer Neural Net

Implementing this with NumPy:

First attempt — let's try this:

N x D   M

```
F = np.dot(X, W) + b
```

D x M

Doesn't work — why?

- NumPy needs to know how to expand "b" from 1D to 2D
- This is called "broadcasting"

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does “np.newaxis” do?

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does “np.newaxis” do?

```
In [3]: b
Out[3]: array([0, 1, 2])
```

b = [0, 1, 2]

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does “np.newaxis” do?

```
In [3]: b
Out[3]: array([0, 1, 2])

In [4]: b[np.newaxis, :]
Out[4]: array([[0, 1, 2]])
```

b = [0, 1, 2]

Make “b” a row vector

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does "np.newaxis" do?

```
In [3]: b
Out[3]: array([0, 1, 2])

In [4]: b[np.newaxis, :]
Out[4]: array([[0, 1, 2]])

In [5]: b[:, np.newaxis]
Out[5]:
array([[0],
       [1],
       [2]])
```

b = [0, 1, 2]

Make "b" a row vector

Make "b" a column vector

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does “np.newaxis” do?

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does "np.newaxis" do?

```
In [12]: b[np.newaxis, :] + np.zeros((3, 3))
Out[12]:
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])
```

Row vector
(repeat along
rows)

1 Layer Neural Net

Implementing this with NumPy:

```
F = np.dot(X, W) + b[np.newaxis, :]
```

What does “np.newaxis” do?

```
In [12]: b[np.newaxis, :] + np.zeros((3, 3))
Out[12]:
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])

In [13]: b[:, np.newaxis] + np.zeros((3, 3))
Out[13]:
array([[ 0.,  0.,  0.],
       [ 1.,  1.,  1.],
       [ 2.,  2.,  2.]])
```

Row vector (repeat along rows)

Column vector (repeat along columns)

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow h$$

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x+b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h+b^{(2)}} \rightarrow f$$

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h + b^{(2)}} \rightarrow f$$

Let's expand out the equation:

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h + b^{(2)}} \rightarrow f$$

Let's expand out the equation:

$$\begin{aligned} f &= W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)} \\ &= (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)}) \end{aligned}$$

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h + b^{(2)}} \rightarrow f$$

Let’s expand out the equation:

$$f = W^{(2)}(W^{(1)}x + b^{(1)}) + b^{(2)}$$

$$= (W^{(2)}W^{(1)})x + (W^{(2)}b^{(1)} + b^{(2)})$$

But this is just the same as a 1 layer net with:

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x+b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h+b^{(2)}} \rightarrow f$$

Let's expand out the equation:

$$f = W^{(2)}(W^{(1)}x+b^{(1)})+b^{(2)}$$

$$= (W^{(2)}W^{(1)})x+(W^{(2)}b^{(1)}+b^{(2)})$$

But this is just the same as a 1 layer net with:

$$W = W^{(2)}W^{(1)} \qquad b = W^{(2)}b^{(1)}+b^{(2)}$$

2 Layer Neural Net

What if we just added another layer?

$$x \rightarrow \boxed{W^{(1)}x+b^{(1)}} \rightarrow h \rightarrow \boxed{W^{(2)}h+b^{(2)}} \rightarrow f$$

Let's expand out the equation:

$$f = W^{(2)}(W^{(1)}x+b^{(1)})+b^{(2)}$$

$$= (W^{(2)}W^{(1)})x+(W^{(2)}b^{(1)}+b^{(2)})$$

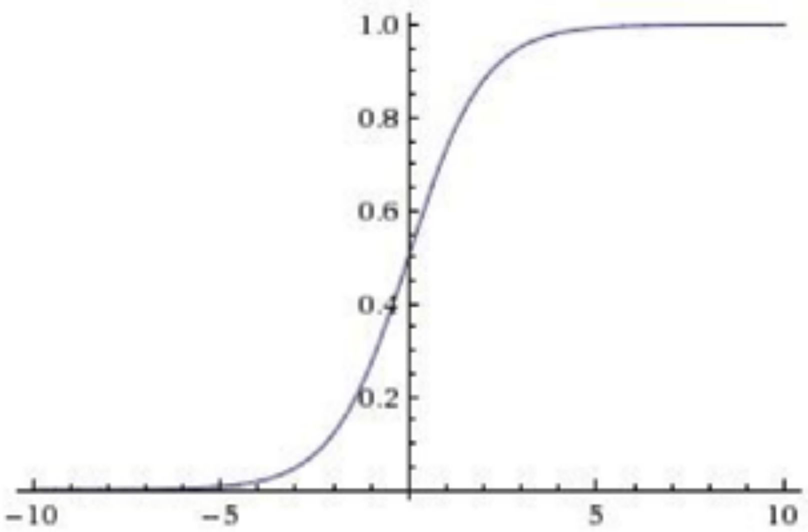
But this is just the same as a 1 layer net with:

$$W = W^{(2)}W^{(1)} \qquad b = W^{(2)}b^{(1)}+b^{(2)}$$

We need a **non-linear** operation between the layers

Nonlinearities

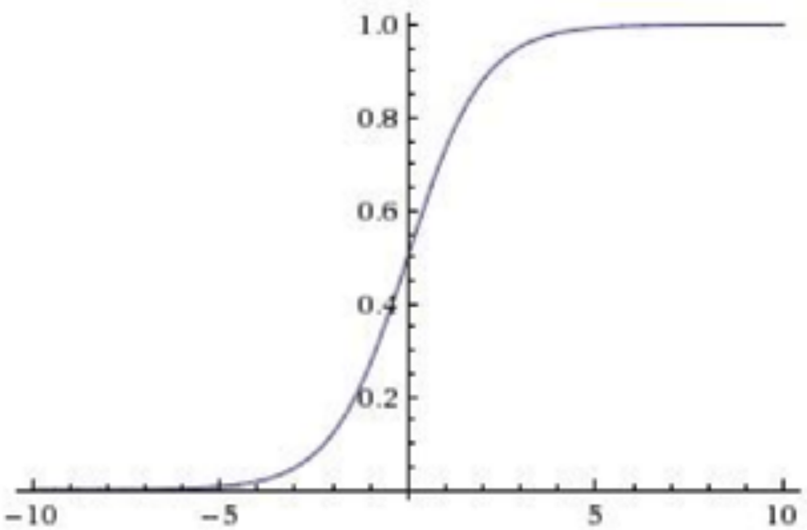
Nonlinearities



Sigmoid

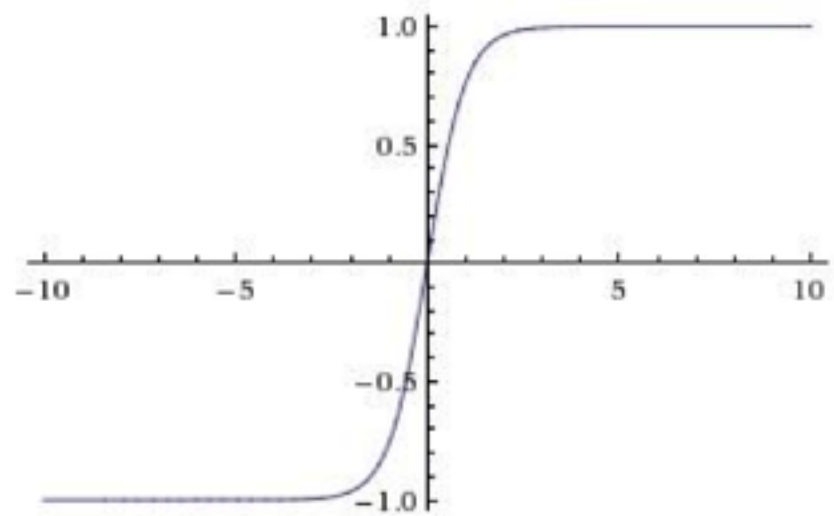
$$\sigma(x) = 1/(1+e^{-x})$$

Nonlinearities



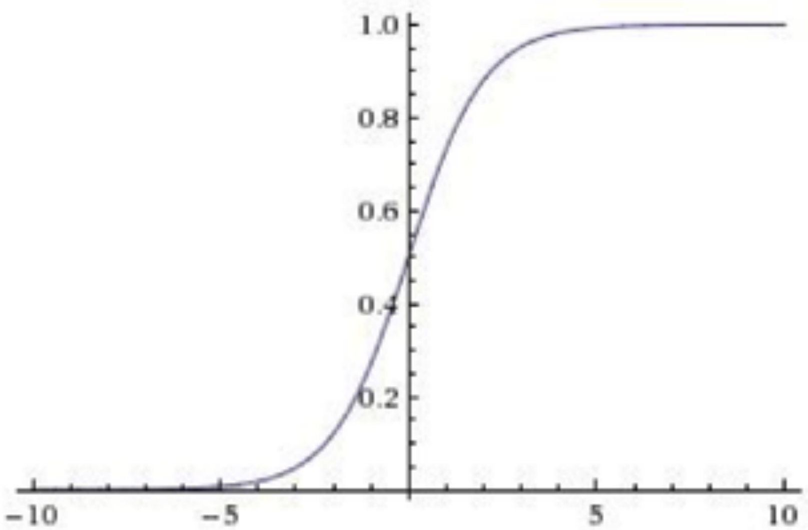
Sigmoid

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



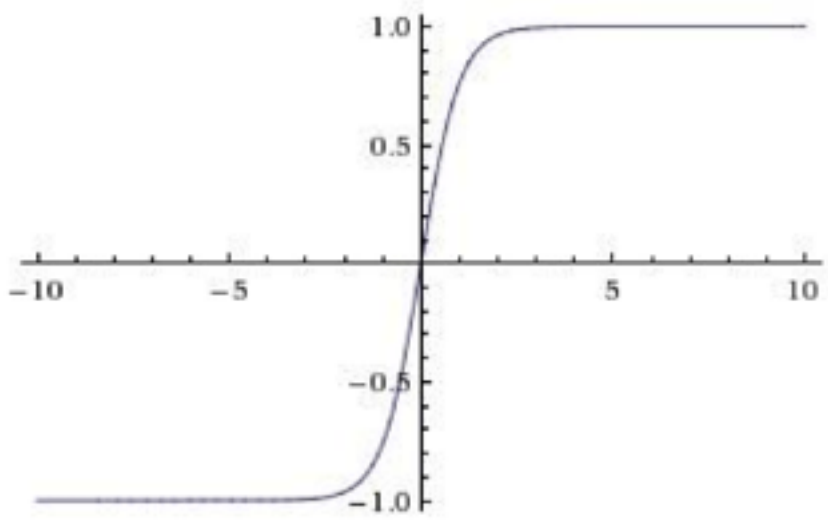
Tanh

Nonlinearities

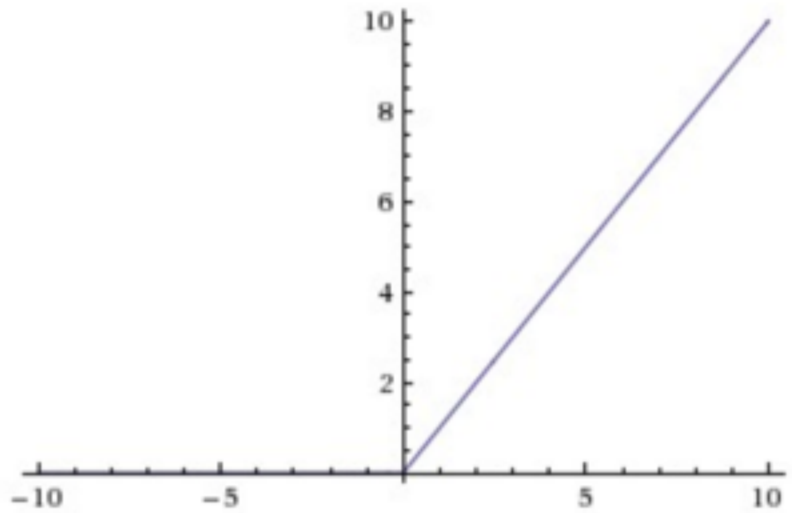


Sigmoid

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

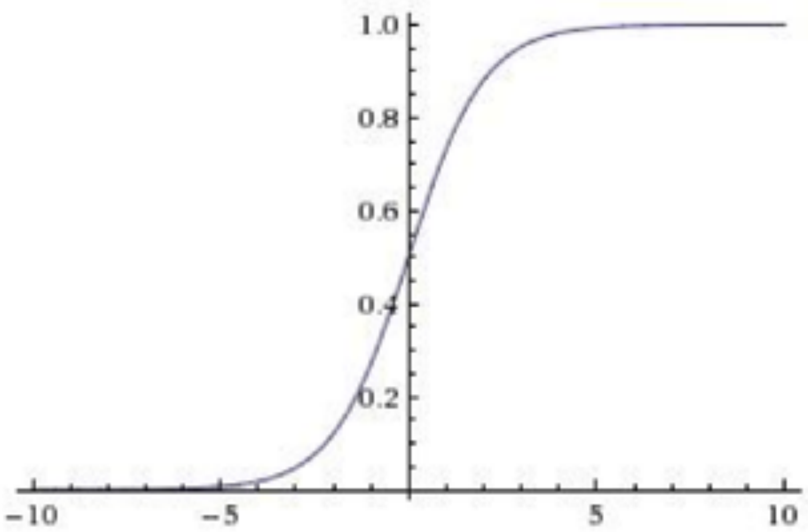


Tanh



ReLU

Nonlinearities

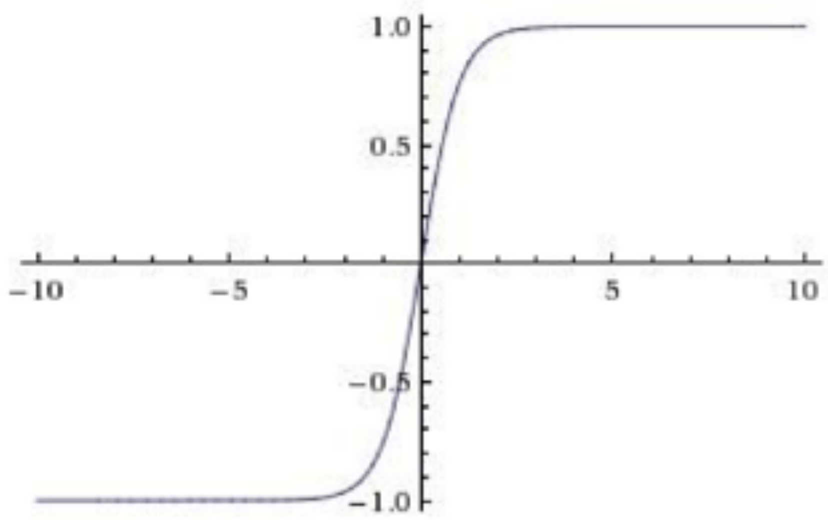


Sigmoid

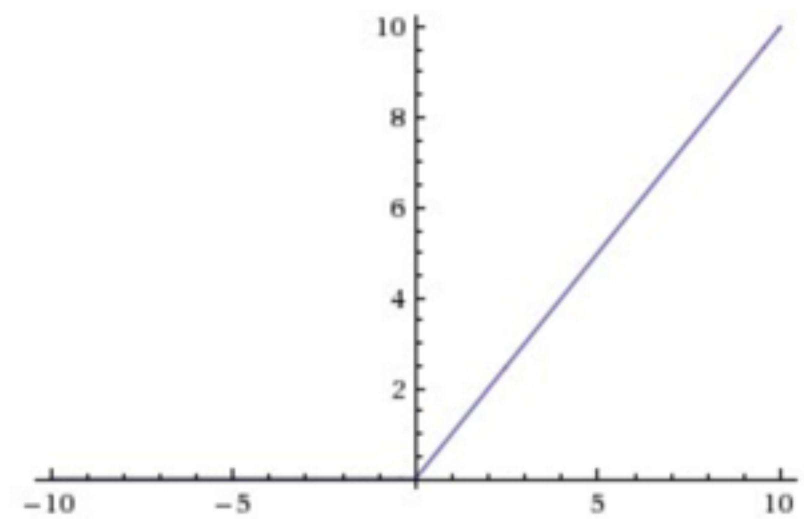
$$\sigma(x) = 1/(1+e^{-x})$$

Historically popular

2 Big problems:
- Not zero centered
- They saturate

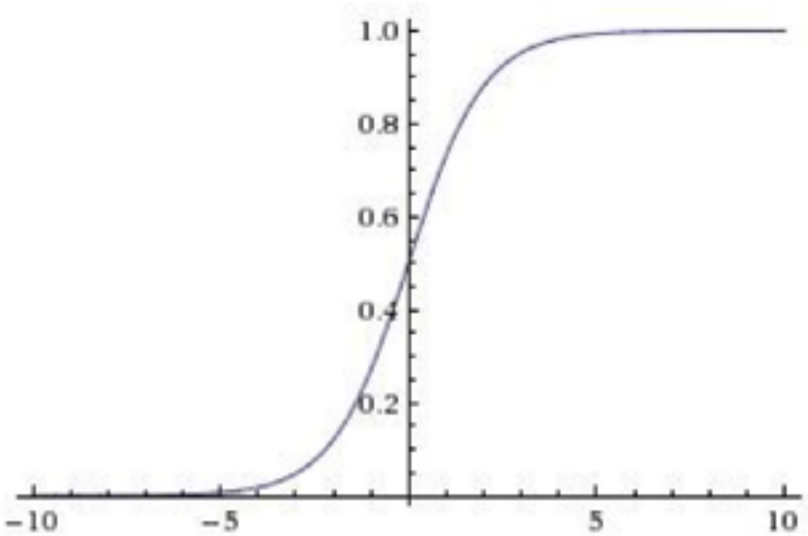


Tanh



ReLU

Nonlinearities

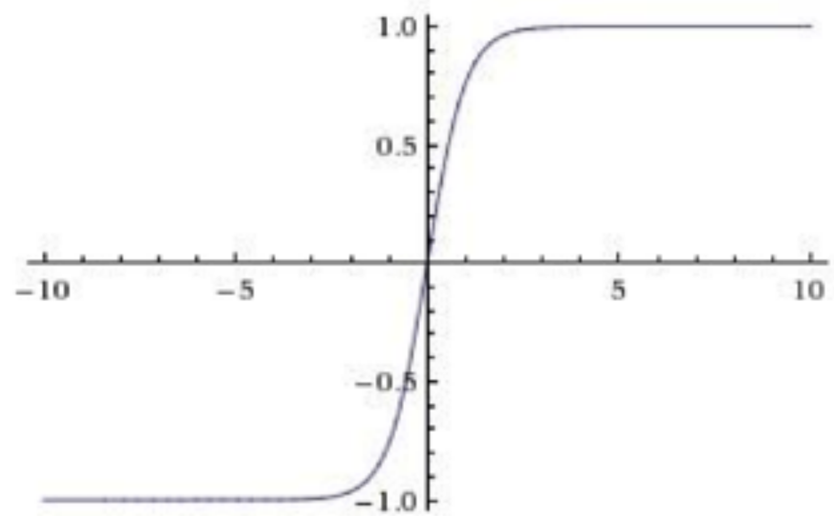


Sigmoid

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

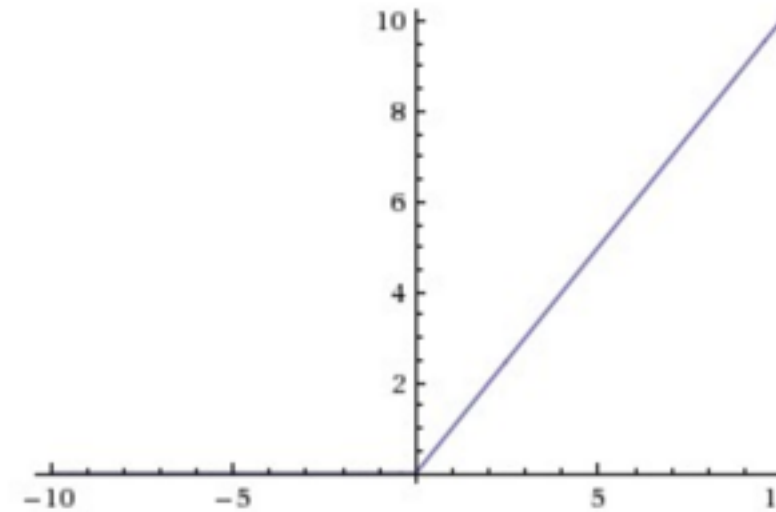
Historically popular

2 Big problems:
- Not zero centered
- They saturate



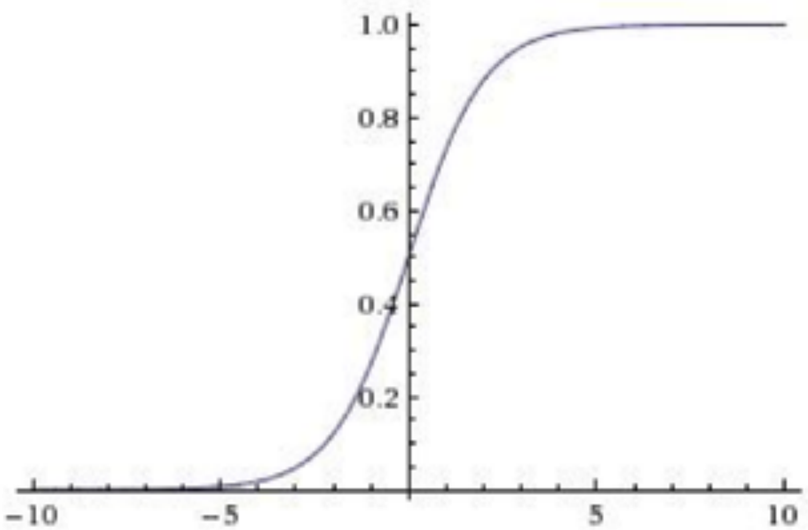
Tanh

- Zero-centered,
- But also saturates



ReLU

Nonlinearities

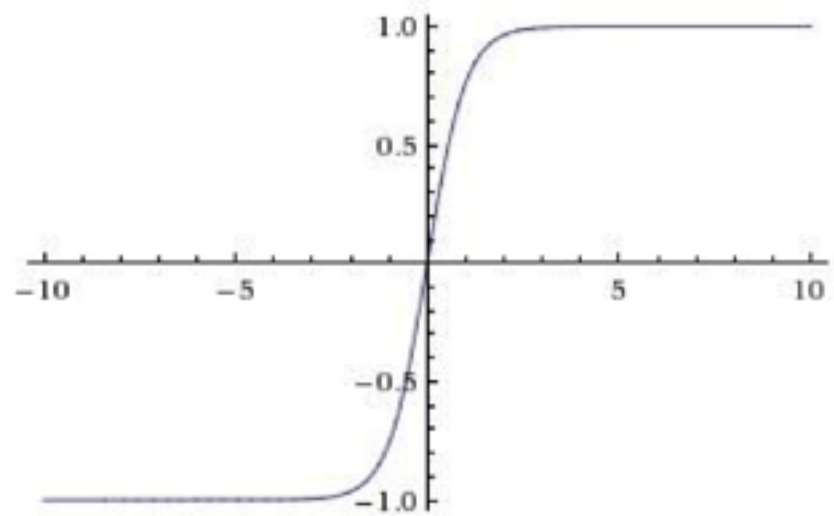


Sigmoid

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

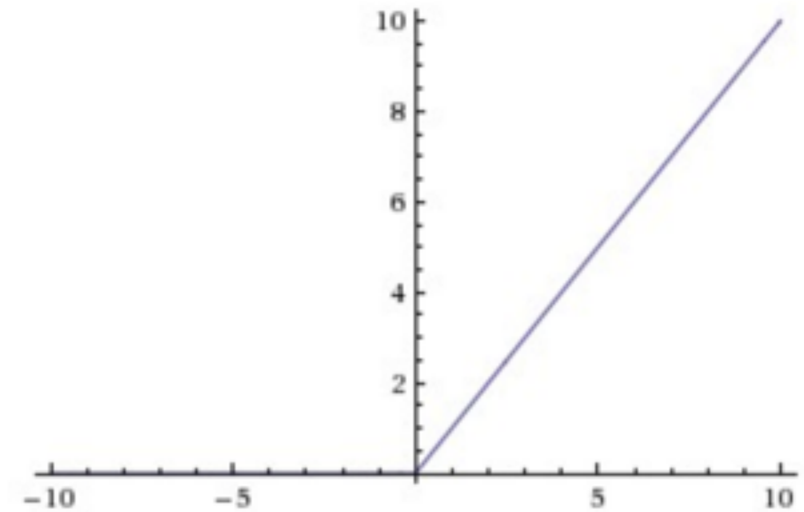
Historically popular

2 Big problems:
- Not zero centered
- They saturate



Tanh

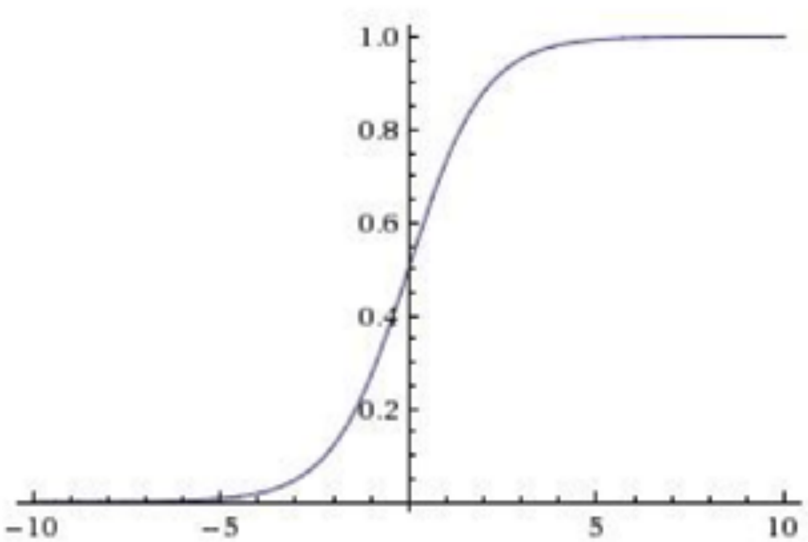
- Zero-centered,
- But also saturates



ReLU

- No saturation
- Very efficient

Nonlinearities

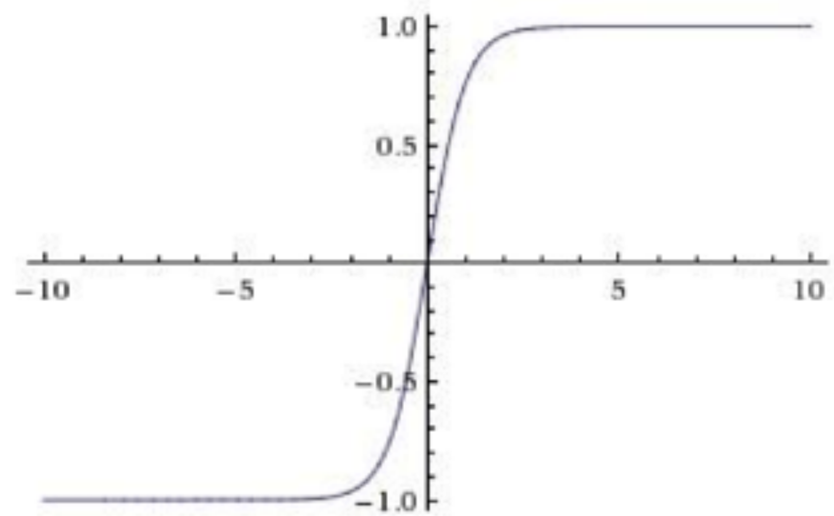


Sigmoid

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

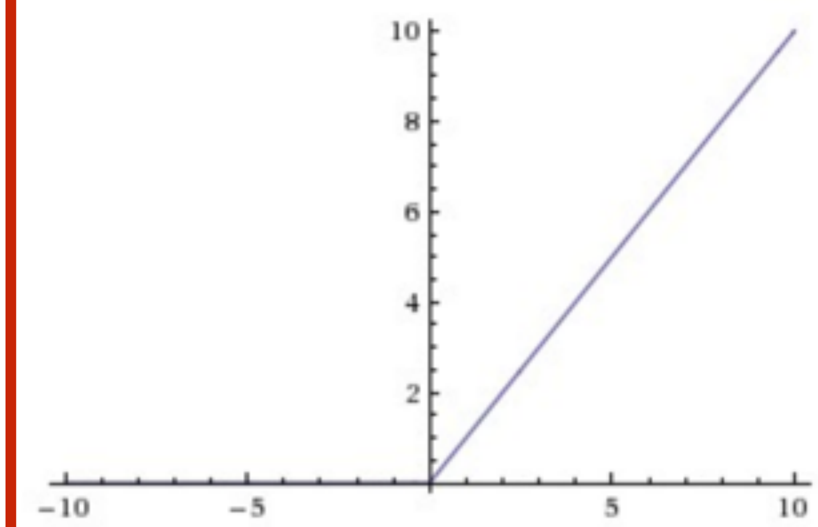
Historically popular

2 Big problems:
- Not zero centered
- They saturate



Tanh

- Zero-centered,
- But also saturates

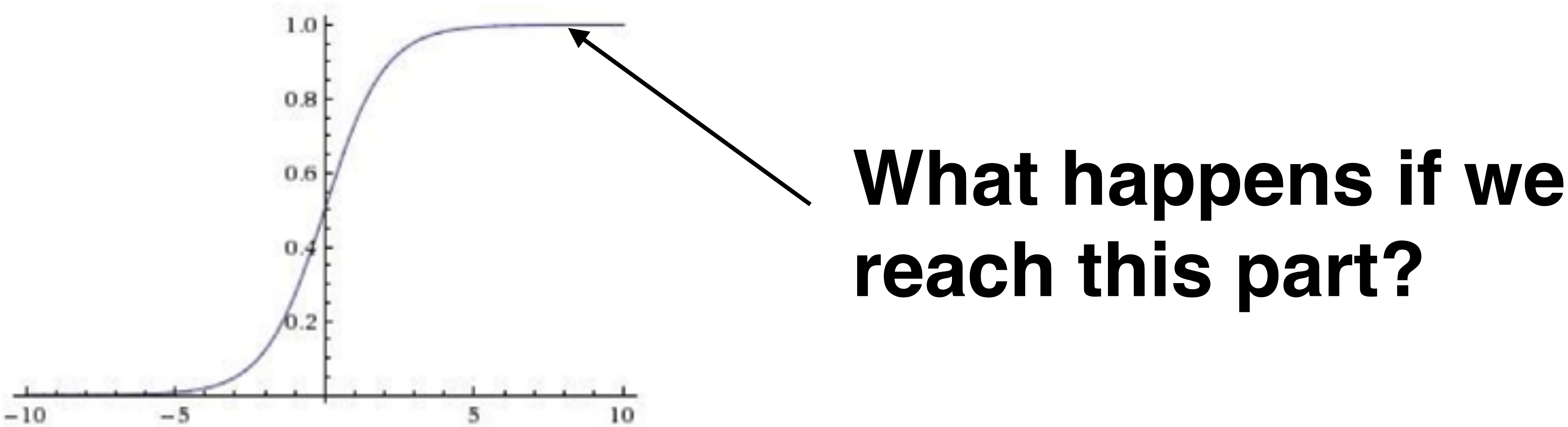


ReLU

Best in practice for classification

- No saturation
- Very efficient

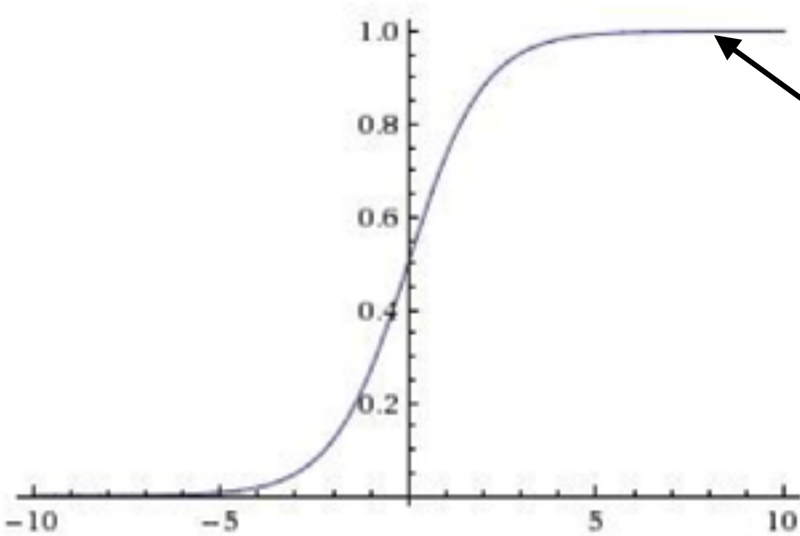
Nonlinearities — Saturation



Sigmoid

```
In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))
```

Nonlinearities — Saturation



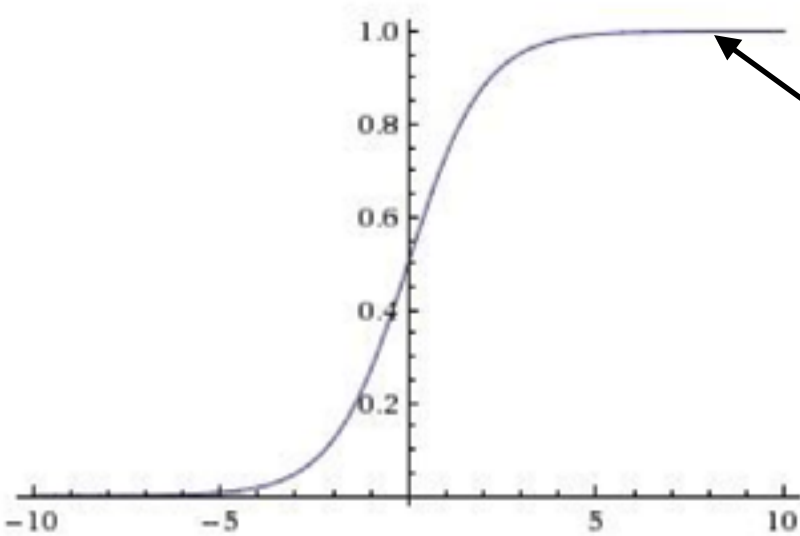
What happens if we reach this part?

Sigmoid

```
In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))
```

```
In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142,  0.88079708,  0.99330715])
```

Nonlinearities — Saturation



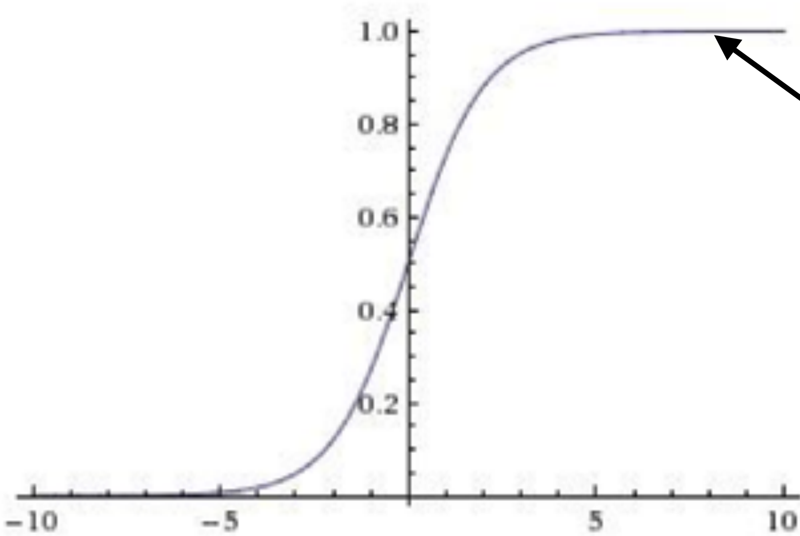
What happens if we reach this part?

Sigmoid

```
In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))
```

```
In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142,  0.88079708,  0.99330715])

In [25]: sigmoid(np.array([-1, 2, 50]))
Out[25]: array([ 0.26894142,  0.88079708,  1.        ])
```

Nonlinearities — Saturation



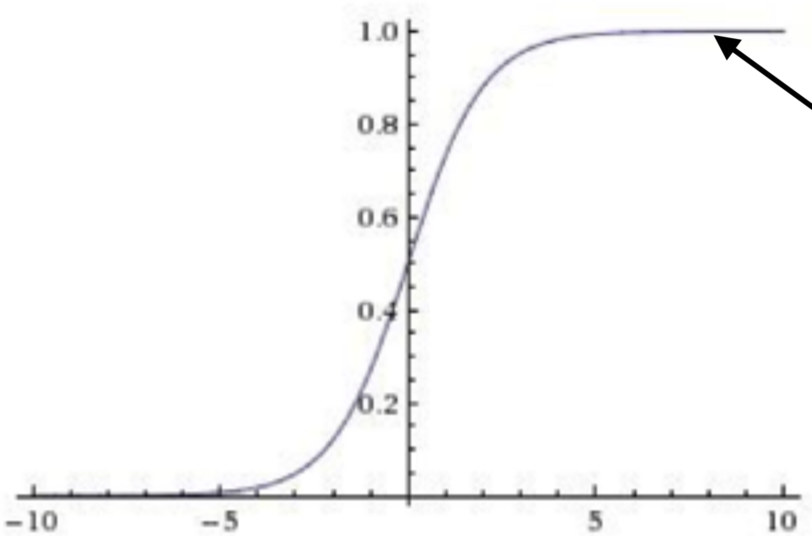
What happens if we reach this part?

Sigmoid

```
In [22]: sigmoid = lambda x: 1 / (1 + np.exp(-x))
```

```
In [24]: sigmoid(np.array([-1, 2, 5]))
Out[24]: array([ 0.26894142,  0.88079708,  0.99330715])

In [25]: sigmoid(np.array([-1, 2, 50]))
Out[25]: array([ 0.26894142,  0.88079708,  1.        ])

In [26]: sigmoid(np.array([100, 200, 50]))
Out[26]: array([ 1.,  1.,  1.])
```

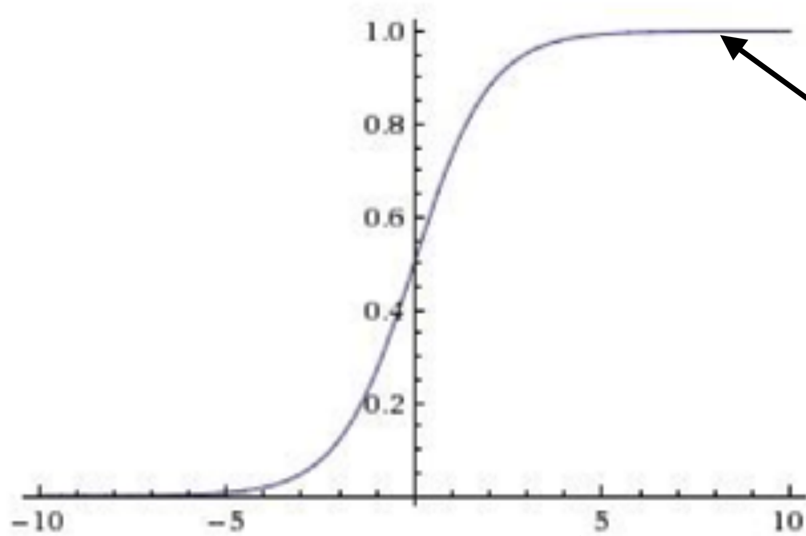
Nonlinearities — Saturation



What happens if we reach this part?

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

Nonlinearities — Saturation

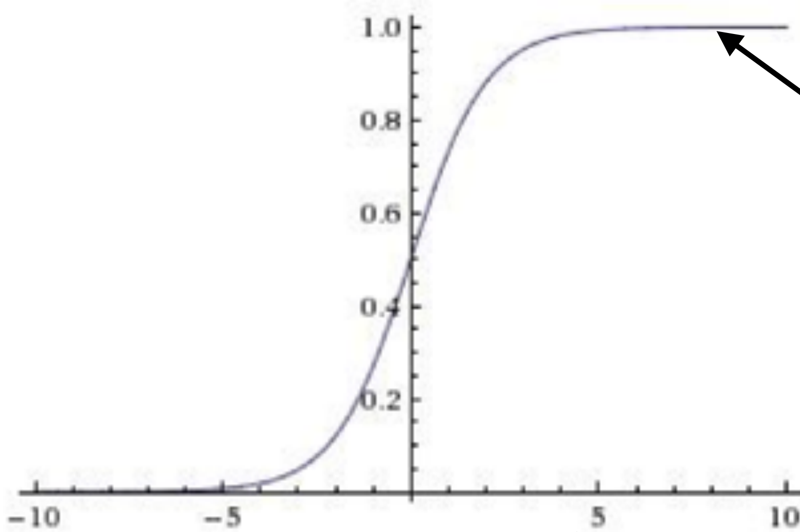


What happens if we reach this part?

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

$$\frac{\partial \sigma}{\partial x} = \sigma(x)\big(1-\sigma(x)\big)$$

Nonlinearities — Saturation



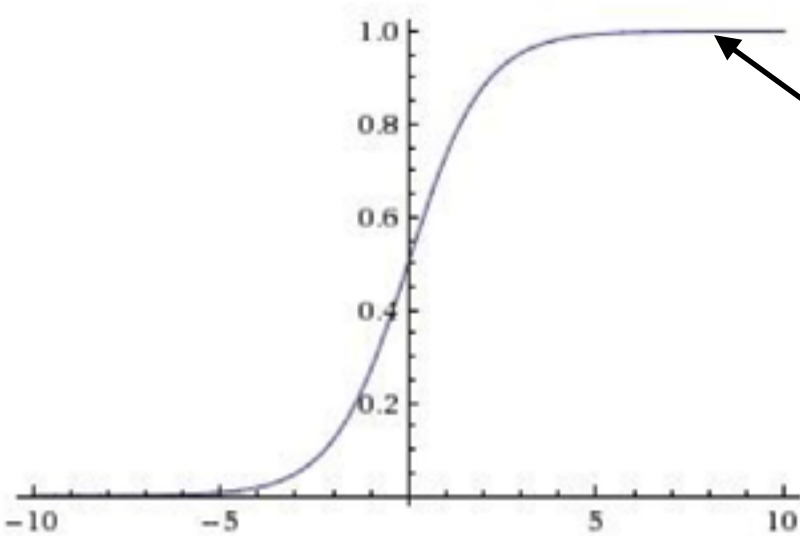
What happens if we reach this part?

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

$$\frac{\partial \sigma}{\partial x} = \sigma(x)\left(1-\sigma(x)\right)$$

```
dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))
```

Nonlinearities — Saturation



What happens if we reach this part?

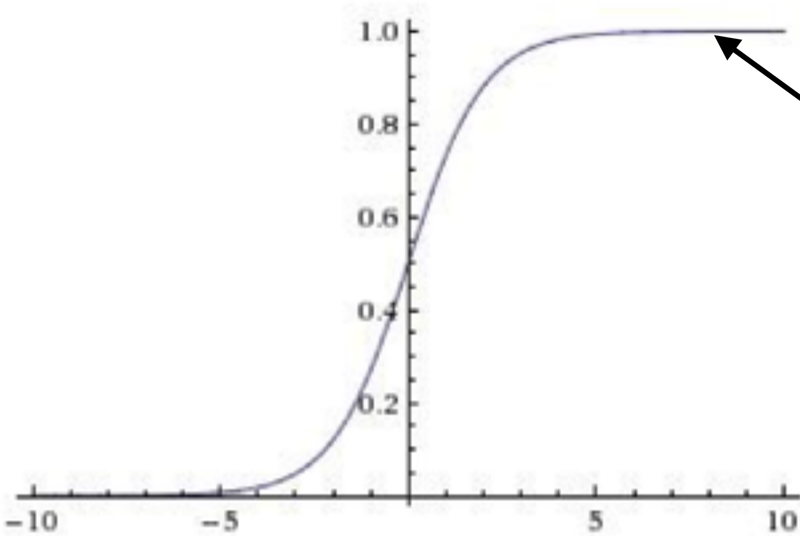
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)\left(1-\sigma(x)\right)$$

```
dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))
```

```
In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193,  0.10499359,  0.00664806])
```

Nonlinearities — Saturation



What happens if we reach this part?

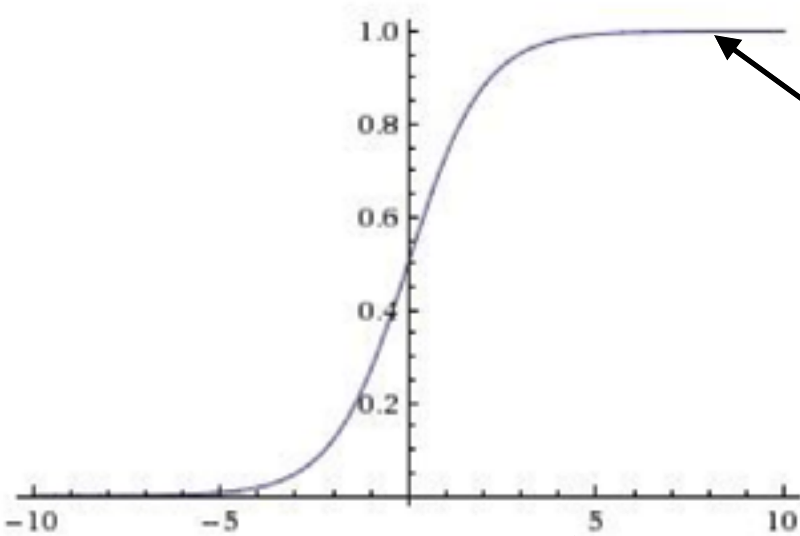
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial \sigma}{\partial x} = \sigma(x)\left(1-\sigma(x)\right)$$

```
dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))
```

```
In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193,  0.10499359,  0.00664806])

In [30]: dsigmoid(np.array([100, 200, 50]))
Out[30]: array([ 0.,  0.,  0.])
```

Nonlinearities — Saturation



What happens if we reach this part?

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

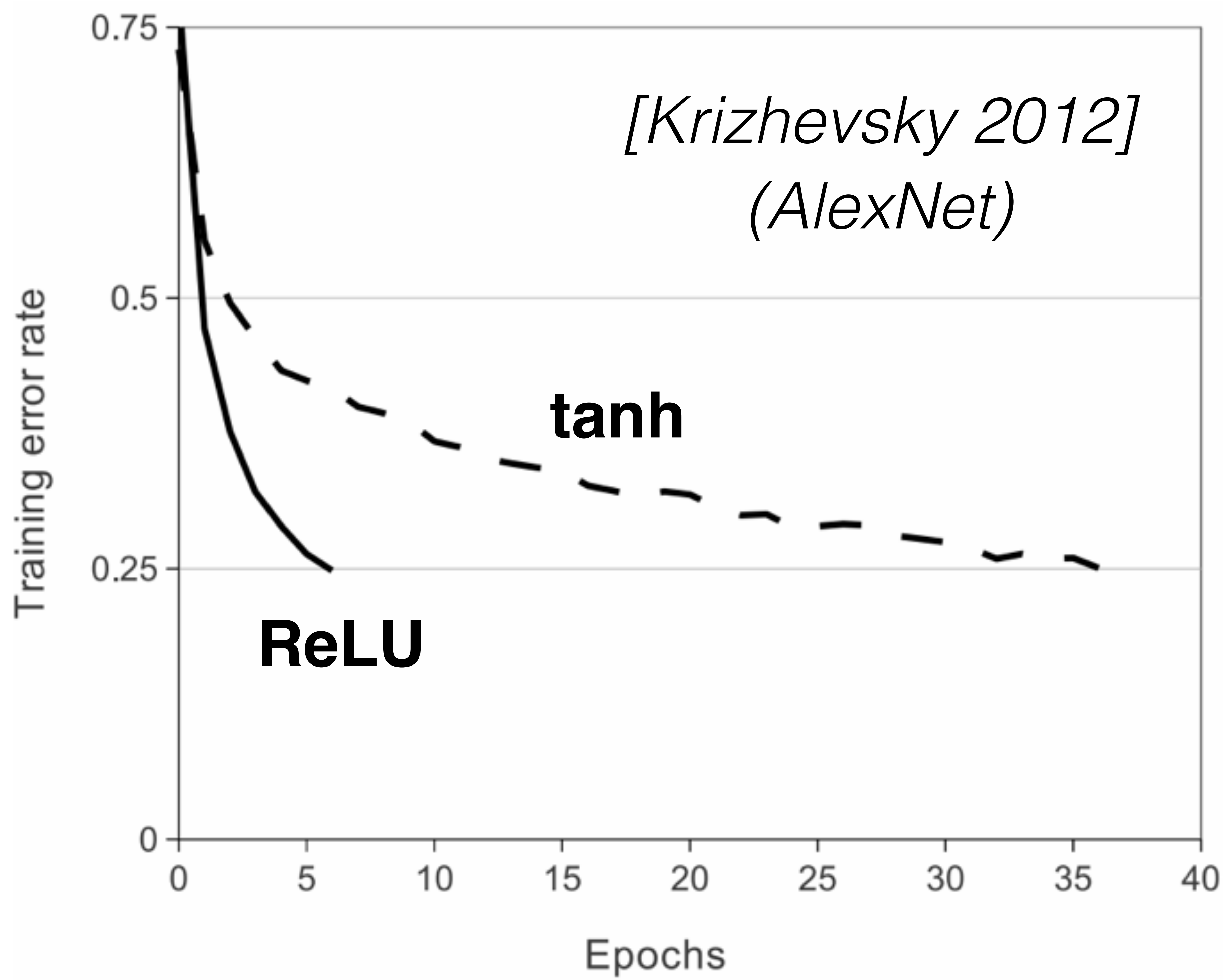
$$\frac{\partial \sigma}{\partial x} = \sigma(x)\left(1-\sigma(x)\right)$$

```
dsigmoid = lambda x: sigmoid(x) * (1 - sigmoid(x))
```

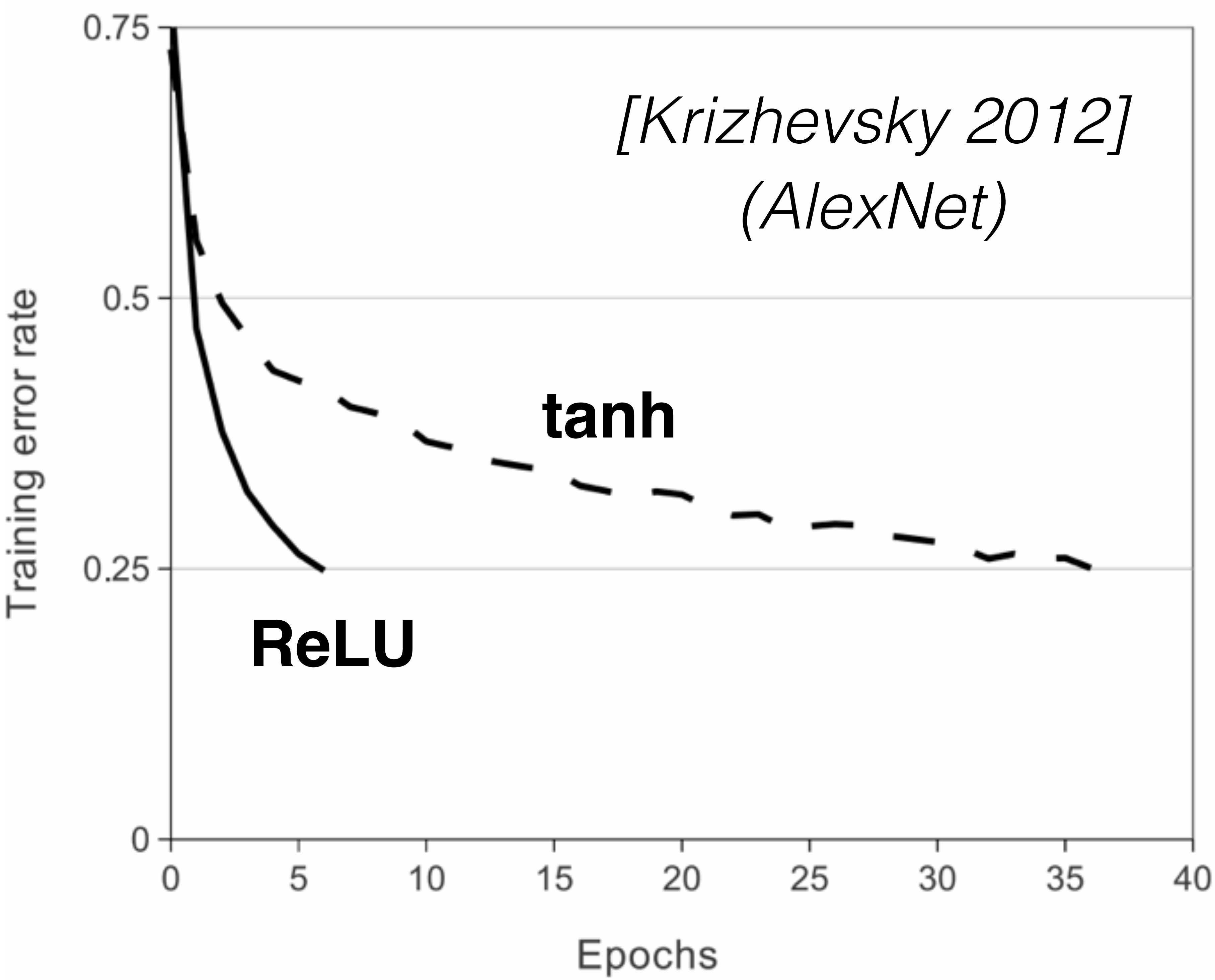
```
In [29]: dsigmoid(np.array([-1, 2, 5]))
Out[29]: array([ 0.19661193,  0.10499359,  0.00664806])

In [30]: dsigmoid(np.array([100, 200, 50]))
Out[30]: array([ 0.,  0.,  0.])
```

Saturation: the gradient is zero!

Nonlinearities



Nonlinearities



In practice, ReLU converges ~6x faster than Tanh for classification problems

ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

```
h1relu = np.maximum(h1, 0)
```

ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

```
h1relu = np.maximum(h1, 0)
```

(b) Make a boolean mask where negative values are True, and then set those entries in h1 to 0:

```
h1relu = h1.copy()
h1relu[h1 < 0] = 0
```

ReLU in NumPy

Many ways to write ReLU — these are all equivalent:

(a) Elementwise max, “0” gets broadcasted to match the size of h1:

```
h1relu = np.maximum(h1, 0)
```

(b) Make a boolean mask where negative values are True, and then set those entries in h1 to 0:

```
h1relu = h1.copy()
h1relu[h1 < 0] = 0
```

(c) Make a boolean mask where positive values are True, and then do an elementwise multiplication (since *int(True) = 1*):

```
h1relu = h1 * (h1 >= 0)
```

2 Layer Neural Net

2 Layer Neural Net

(Layer 1)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow$$

2 Layer Neural Net

(Layer 1) (Nonlinearity)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow \boxed{\max(0,\cdot)} \rightarrow h^{(1)}$$

("hidden activations")

2 Layer Neural Net

(Layer 1) (Nonlinearity) (Layer 2)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow \boxed{\max(0,\cdot)} \rightarrow h^{(1)} \rightarrow \boxed{W^{(2)}x + b^{(2)}} \rightarrow f$$

("hidden activations")

2 Layer Neural Net

(Layer 1) (Nonlinearity) (Layer 2)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow \boxed{\max(0,\cdot)} \rightarrow h^{(1)} \rightarrow \boxed{W^{(2)}x + b^{(2)}} \rightarrow f$$

(“hidden activations”)

Let’s expand out the equation:

2 Layer Neural Net

(Layer 1) (Nonlinearity) (Layer 2)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow \boxed{\max(0,\cdot)} \rightarrow h^{(1)} \rightarrow \boxed{W^{(2)}x + b^{(2)}} \rightarrow f$$

(“hidden activations”)

Let’s expand out the equation:

$$f = W^{(2)} \max(0, W^{(1)}x + b^{(1)}) + b^{(2)}$$

2 Layer Neural Net

(Layer 1) (Nonlinearity) (Layer 2)

$$x \rightarrow \boxed{W^{(1)}x + b^{(1)}} \rightarrow \boxed{\max(0,\cdot)} \rightarrow h^{(1)} \rightarrow \boxed{W^{(2)}x + b^{(2)}} \rightarrow f$$

("hidden activations")

Let's expand out the equation:

$$f = W^{(2)} \max(0, W^{(1)}x + b^{(1)}) + b^{(2)}$$

Now it no longer simplifies — yay

2 Layer Neural Net

$$x \rightarrow \overset{\text{(Layer 1)}}{\boxed{W^{(1)}x+b^{(1)}}} \rightarrow \overset{\text{(Nonlinearity)}}{\boxed{\max(0,\cdot)}} \rightarrow \underset{\text{(“hidden activations”)}}{h^{(1)}} \rightarrow \overset{\text{(Layer 2)}}{\boxed{W^{(2)}x+b^{(2)}}} \rightarrow f$$

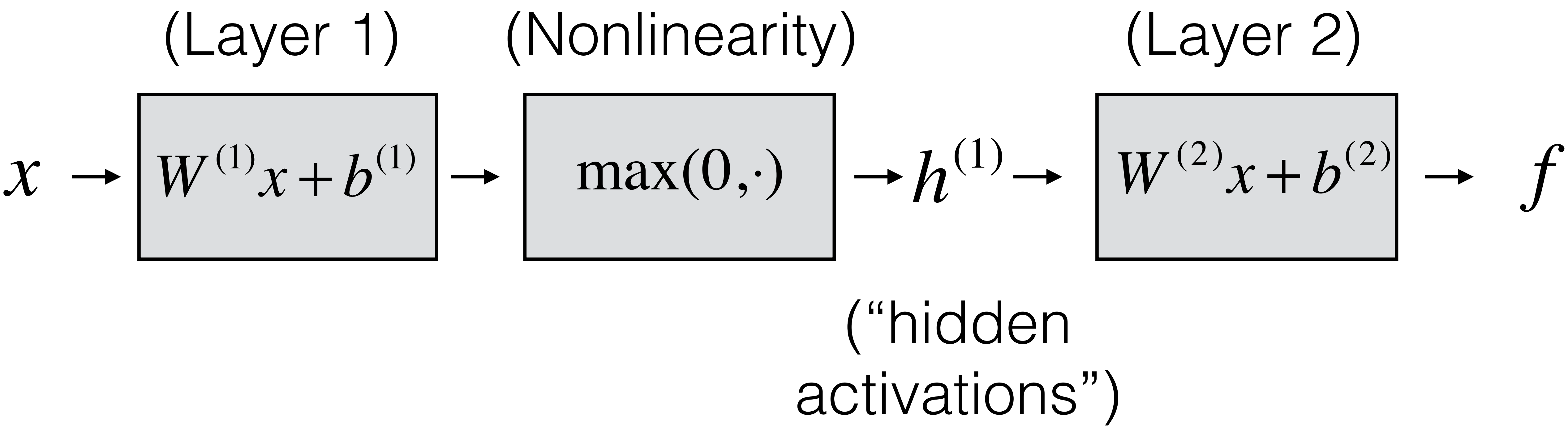
Let’s expand out the equation:

$$f = W^{(2)} \max(0, W^{(1)}x + b^{(1)}) + b^{(2)}$$

Now it no longer simplifies — yay

Note: *any* nonlinear function will prevent this collapse, but not all nonlinear functions actually work in practice

2 Layer Neural Net



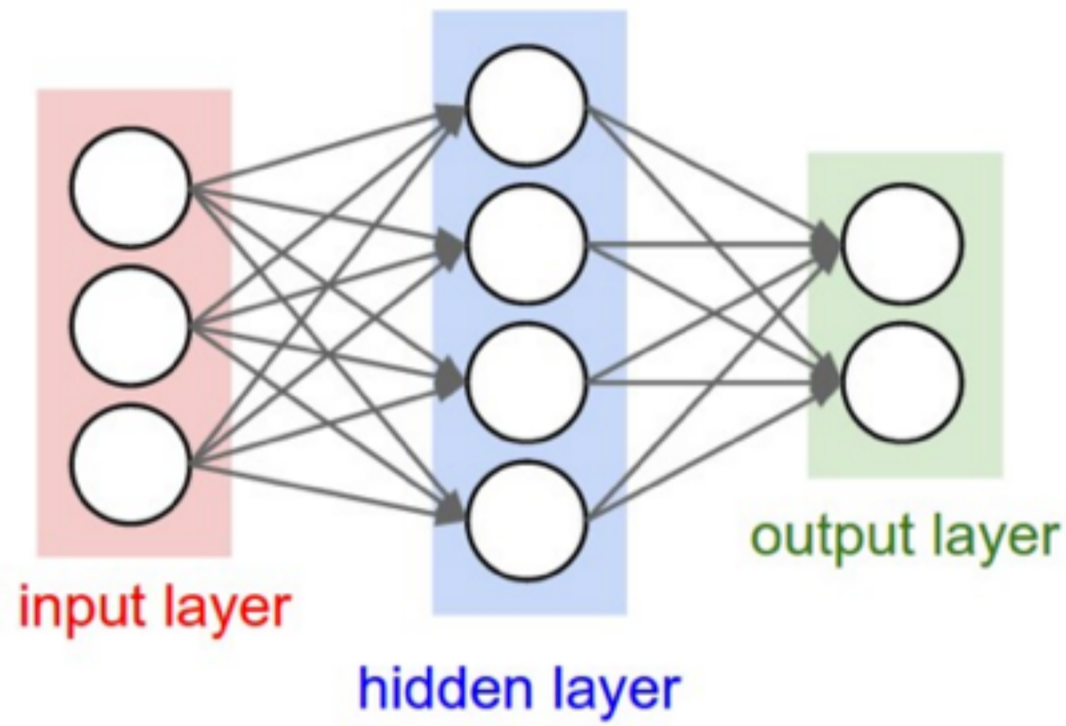
Note: Traditionally, the nonlinearity was considered part of the layer and is called an "activation function"

In this class, we will consider them separate layers, but be aware that many others consider them part of the layer

Neural Net:
Graphical Representation

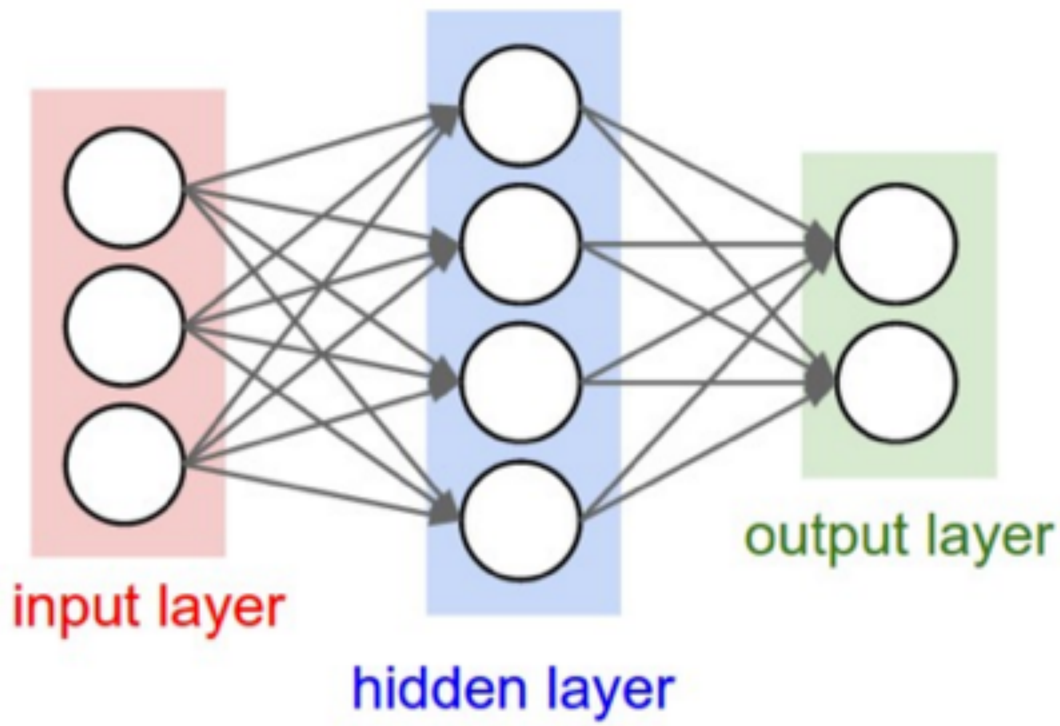
Neural Net: Graphical Representation

2 layers



Neural Net: Graphical Representation

2 layers

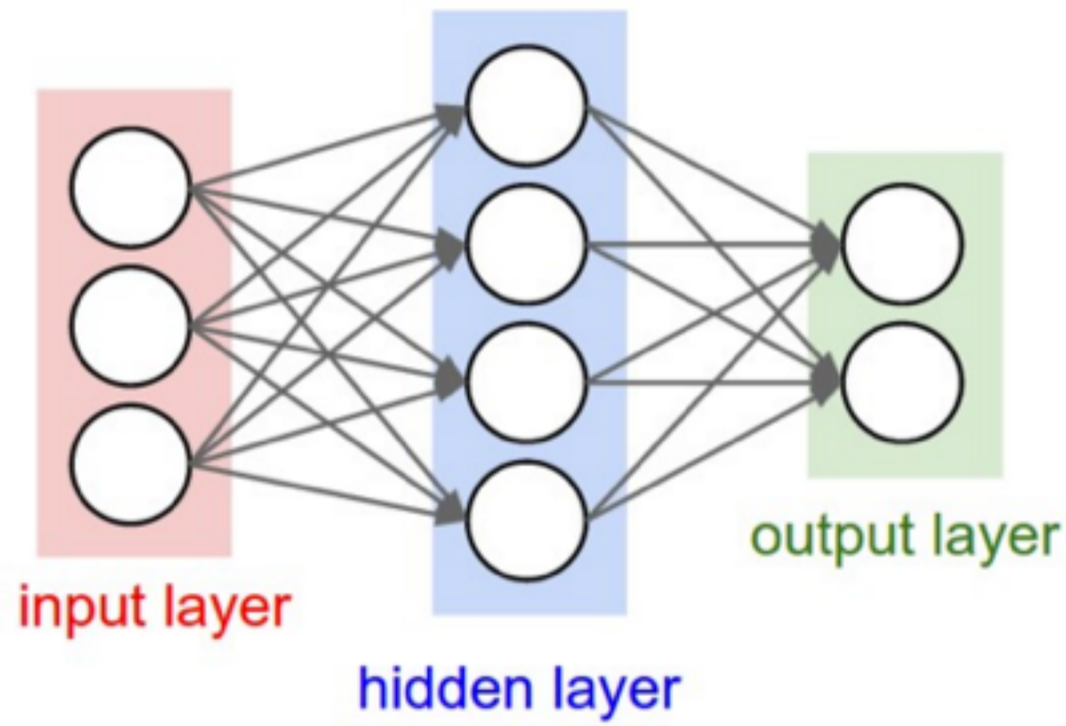


- Called **“fully connected”** because every output depends on every input.

- Also called **“affine”** or **“inner product”**

Neural Net: Graphical Representation

2 layers

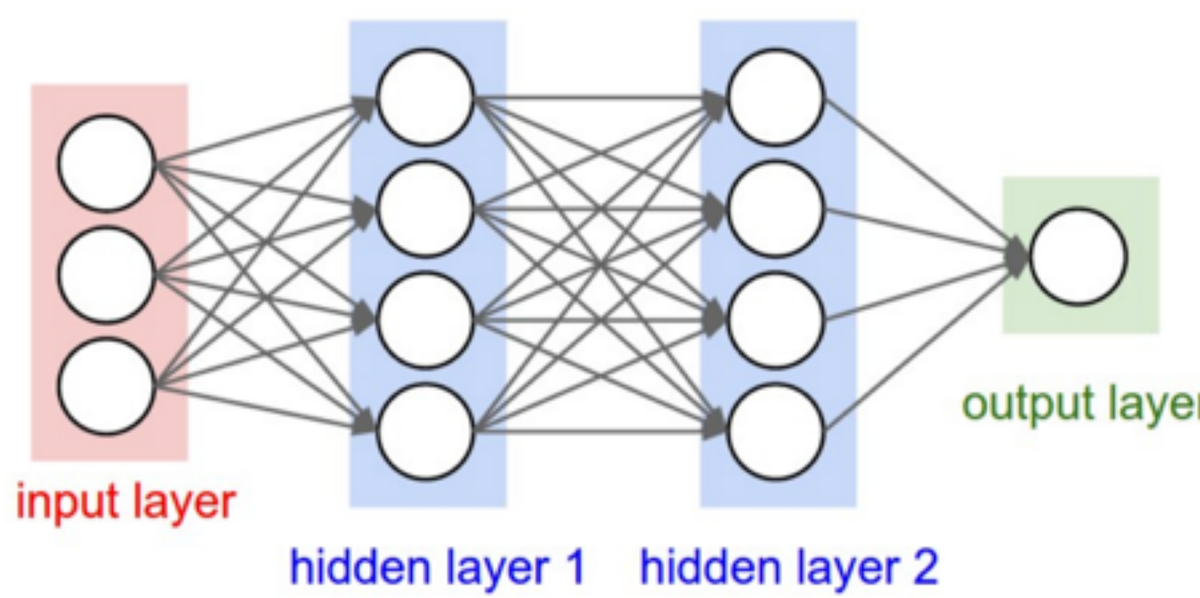


- Called **“fully connected”** because every output depends on every input.

- Also called **“affine”** or **“inner product”**

---

3 layers



Questions?

Neural Networks, More generally

$$x \to \boxed{\text{Function}} \to h^{(1)} \to \boxed{\text{Function}} \to h^{(2)} \to \cdots \to f$$

Neural Networks,
More generally

$$x \rightarrow \boxed{\text{Function}} \rightarrow h^{(1)} \rightarrow \boxed{\text{Function}} \rightarrow h^{(2)} \rightarrow \cdots \rightarrow f$$

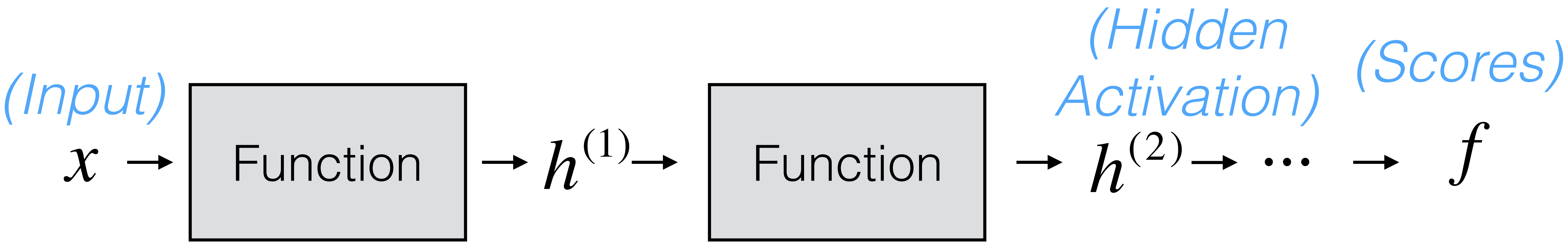
This can be:

- Fully connected layer
- Nonlinearity (ReLU, Tanh, Sigmoid)
- Convolution
- Pooling (Max, Avg)
- Vector normalization (L1, L2)
- *Invent new ones*

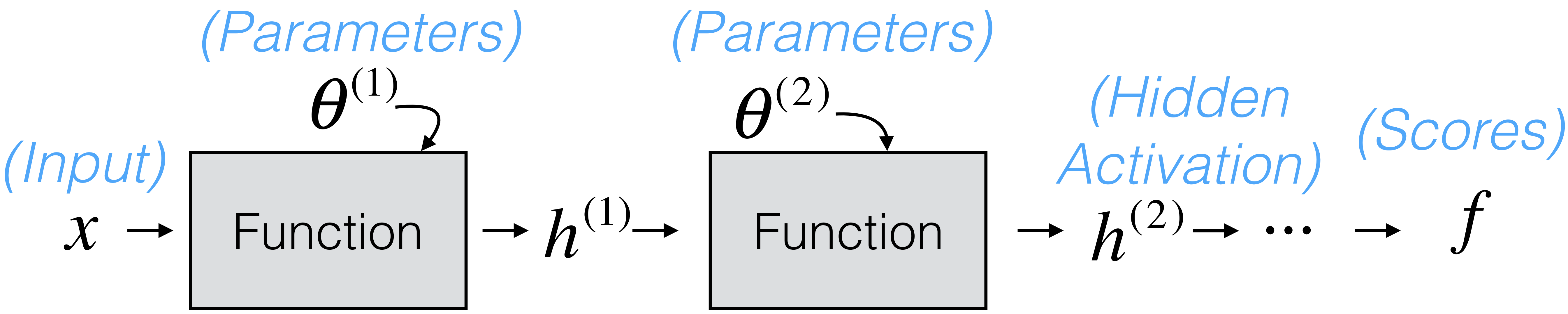
Neural Networks,
More generally

$$x \to \boxed{\text{Function}} \to h^{(1)} \to \boxed{\text{Function}} \to h^{(2)} \to \cdots \to f$$

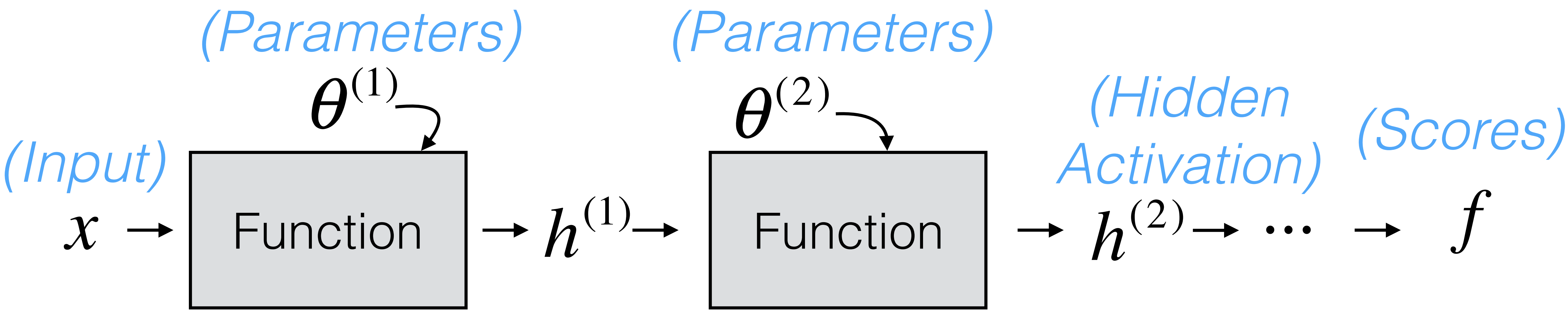
Neural Networks, More generally



Neural Networks, More generally

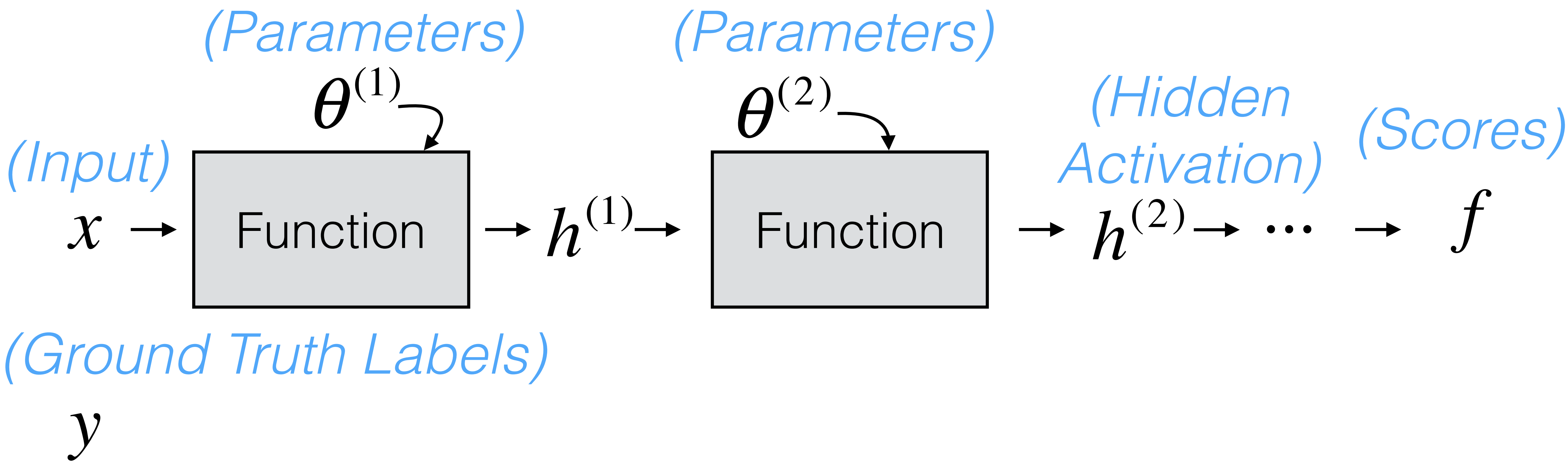


Neural Networks, More generally



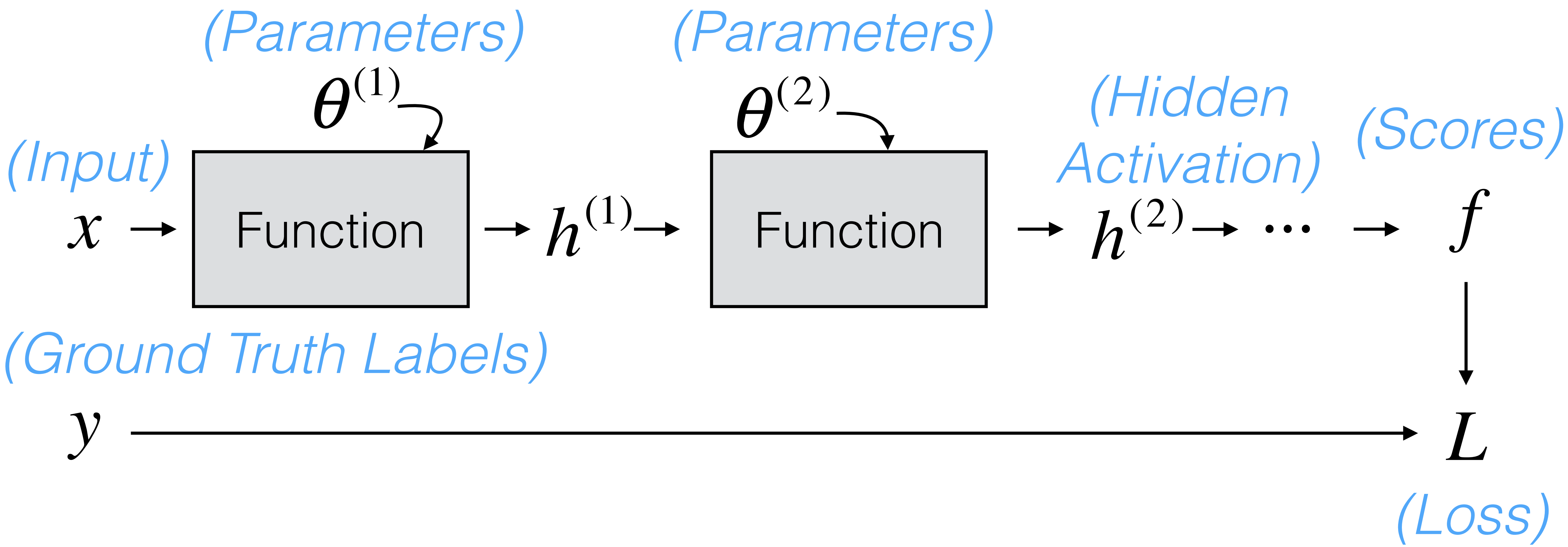
Here, θ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).

Neural Networks, More generally



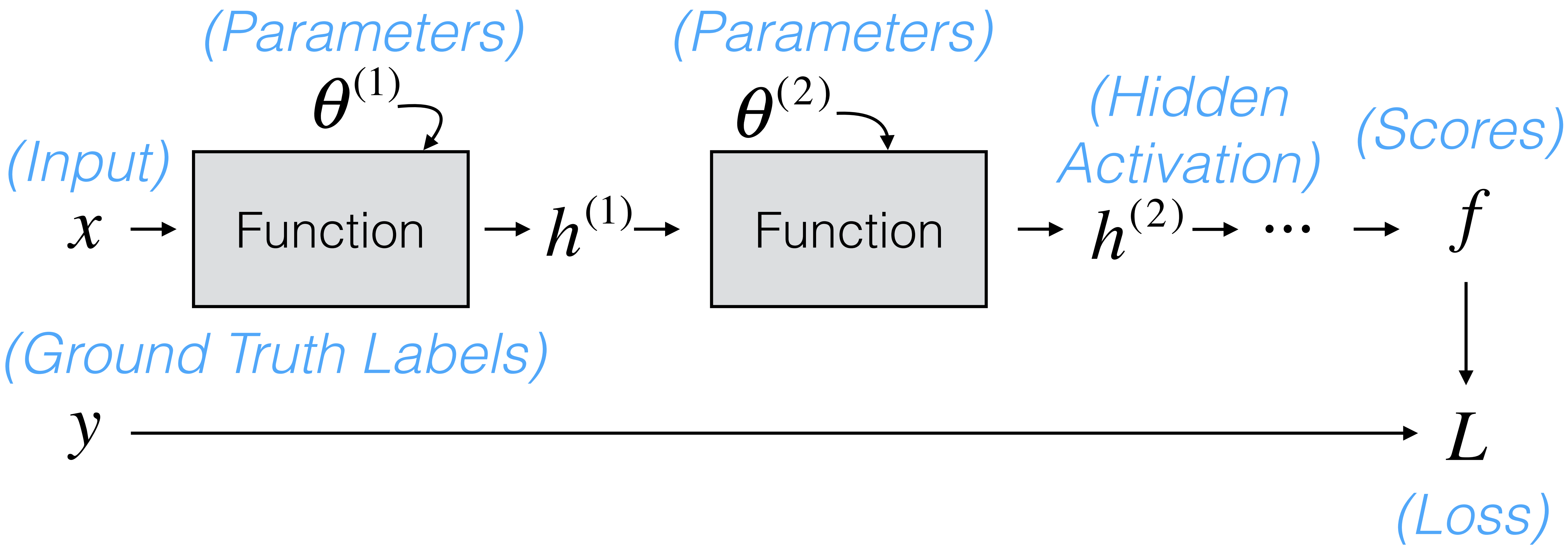
Here, θ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).

Neural Networks, More generally



Here, θ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).

Neural Networks, More generally



Here, θ represents whatever parameters that layer is using (e.g. for a fully connected layer $\theta^{(1)} = \{ W^{(1)}, b^{(1)} \}$).

Recall: the loss "L" measures how far the predictions "f" are from the labels "y". The most common loss is Softmax.