Machine Learning: Neural Networks

ROB 102: Introduction to AI & Programming

Lecture 13

2021/12/01

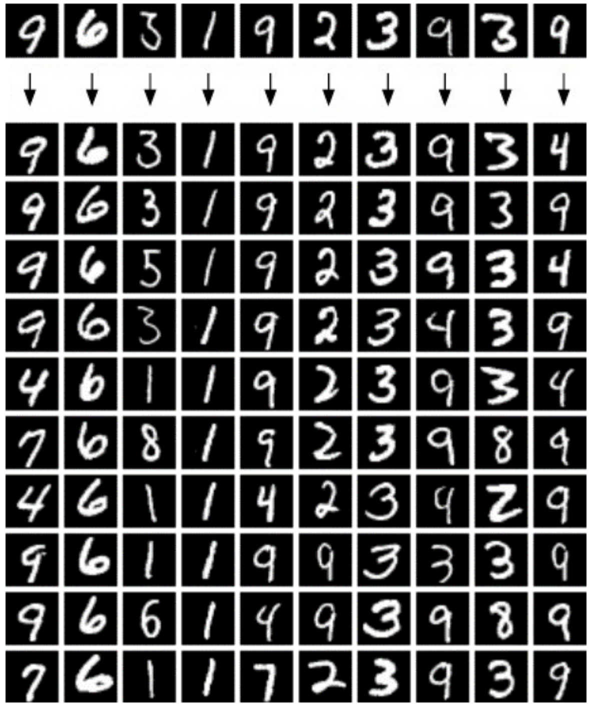
Project 4: Machine Learning

Implement three machine learning algorithms to classify images from the MNIST dataset.

1. Nearest neighbors
2. Linear Classifier
3. Neural Network

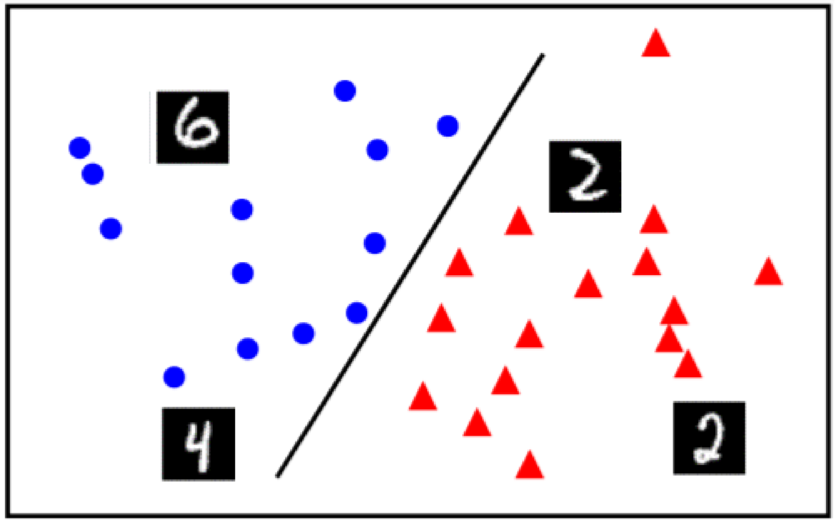
Where we are

✓ P4.1: Nearest Neighbors



distance(2 , 2)

✓ P4.2: Linear Classifier



$$f(X) = W \times X + b$$

Where we are

✓ P4.1: Nearest Neighbors

+ Straight-forward to implement
+ No training necessary

– Requires a lot of memory

– Expensive at computation time

– Distance isn't always a good indicator of class similarity

✓ P4.2: Linear Classifier

+ Only one matrix to learn
+ Fast at test time

– Can only represent linearly separable data

Can we do better than a linear model?

Project 4: Machine Learning

Implement three machine learning algorithms to classify images from the MNIST dataset.

1. Nearest neighbors
2. Linear Classifier
3. **Neural Network** (Today!)

Where we are

We’ll use the same algorithm for training a neural network!

Training algorithm:

For linear classifier, this is just the weight matrix and the bias

Initialize the parameters randomly

Mini-batch sampling

For N iterations, do:

We used the SVM classification loss with regularization

1. Sample a batch of training images
2. Evaluate the loss and gradients for the batch using current parameters
3. Update the parameters using the gradients

Gradient Descent! The learning rate controls how fast we learn

Parameters to learn: W, b

Hyperparameters to tune: learning rate, reg. coefficient

Where we are

We'll use the same algorithm for evaluating a neural network!

Prediction algorithm (test time):

W, b

Given a test image & parameters from the training stage:

1. Calculate class scores $f(X) = W \times X + b$
2. Assign label of class with the highest score.

```
y_pred = argmax(scores)
```

This time…

- The neural network model
- Training a neural network

P4.3

- Briefly:
  - Backpropagation
  - Convolutional Neural Networks

Not needed for P4.3!

This time: Neural Networks

Linear classifier:

$$f(X) = W \times X + b$$

Q: How can we represent a more complex, non-linear function?

This time: Neural Networks

Linear classifier:

$$f(X) = W \times X + b$$

Neural network (2-layers):

$$f(X) = W_2 \times \max(W_1 \times X + b_1, 0) + b_2$$

A neural network can approximate any* function!

*(*with some caveats)*

A good visual explanation: (link)

This time: Neural Networks

Linear classifier:

$$f(X) = W \times X + b$$

Neural network (2-layers):

$$f(X) = W_2 \times \max(W_1 \times X + b_1, 0) + b_2$$

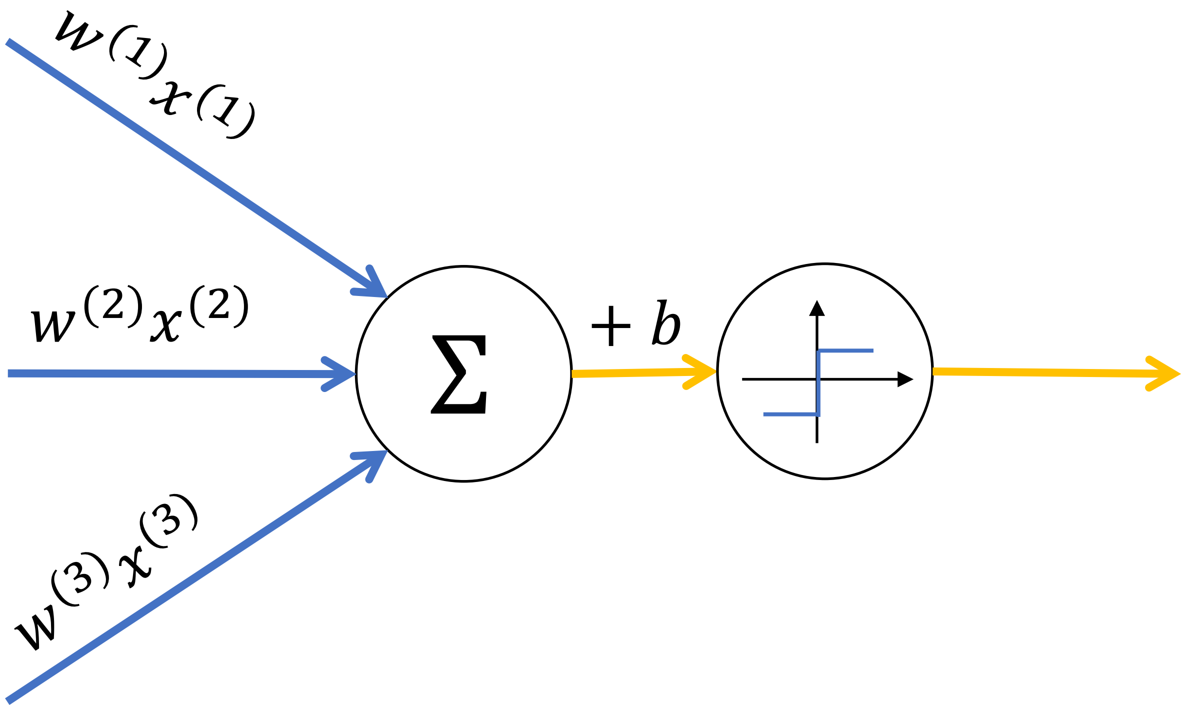
Neural network (3-layers):

$$f(X) = W_3 \times \max(W_2 \times \max(W_1 \times X + b_1) + b_2) + b_3$$

How are these equations neural networks??

The perceptron

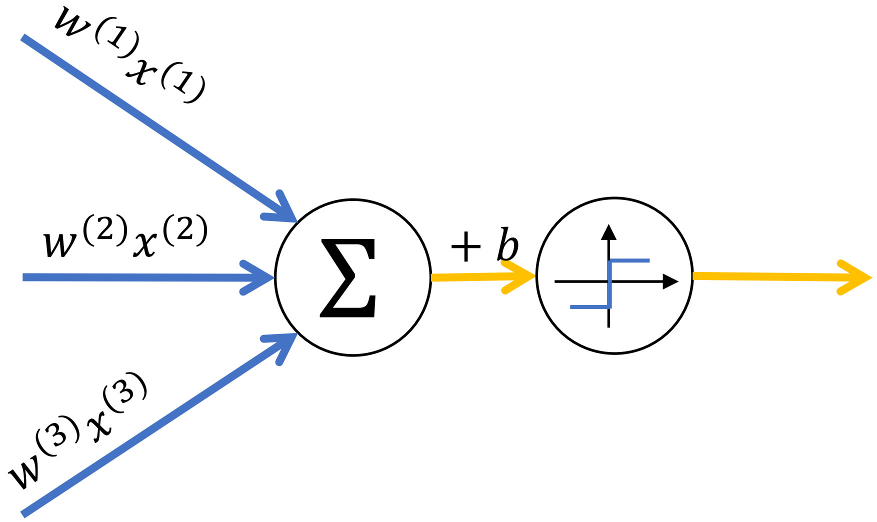
A **perceptron** is an algorithm for binary (linear!) classification.



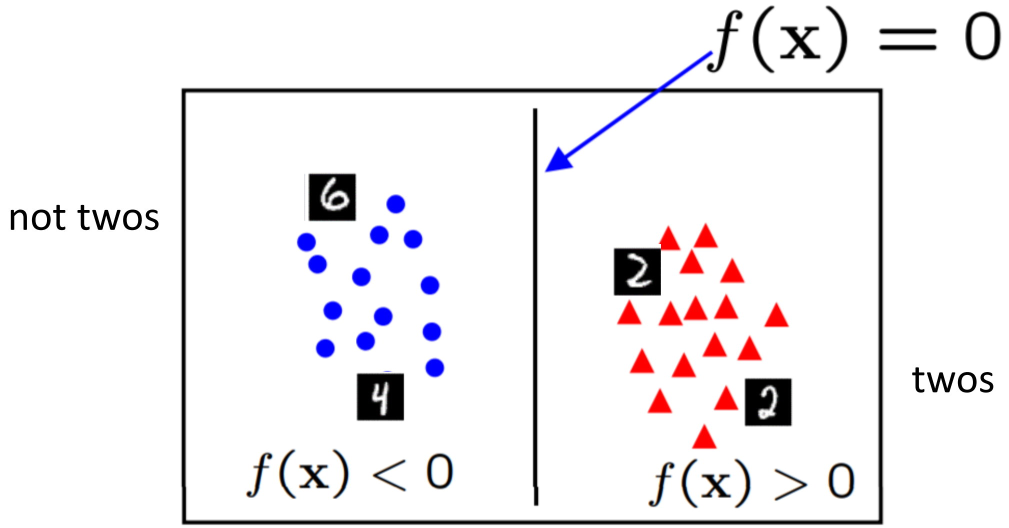
$$y_{pred} = \begin{cases} 1 & \text{if } W \cdot X + b \geq 0 \\ -1 & \text{if } W \cdot X + b < 0 \end{cases}$$

The perceptron

A **perceptron** is an algorithm for binary (linear!) classification.

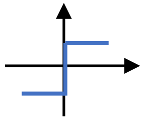


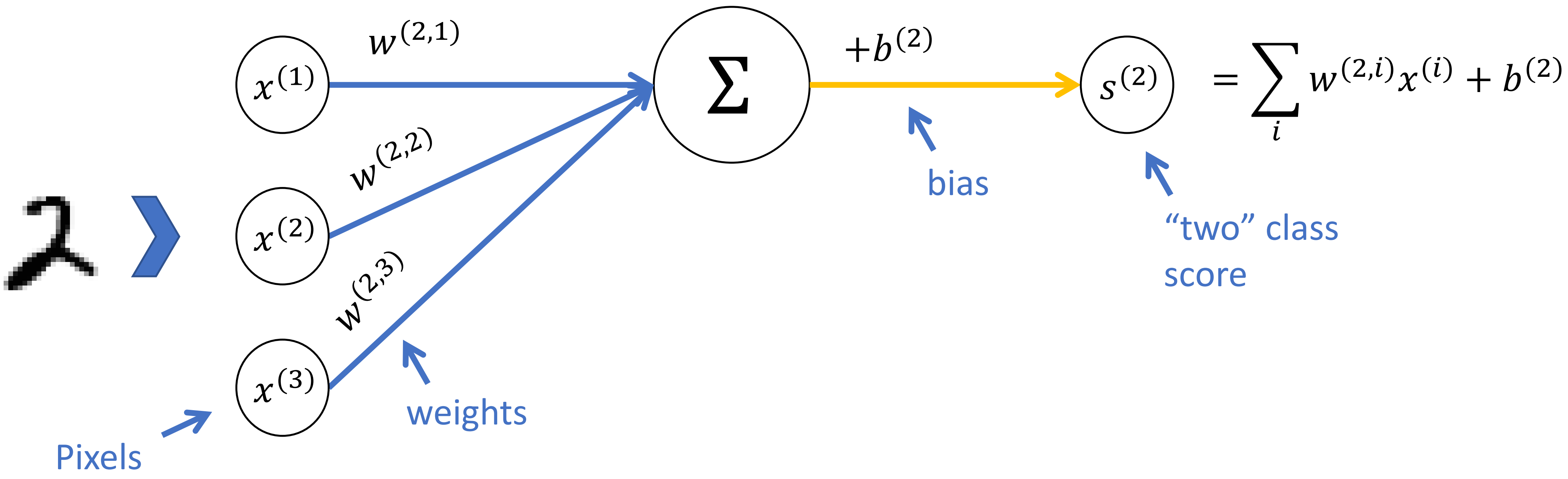
$$y_{pred} = \begin{cases} 1 & \text{if } W \cdot X + b \geq 0 \\ -1 & \text{if } W \cdot X + b < 0 \end{cases}$$



Recall: This is our linear classifier!

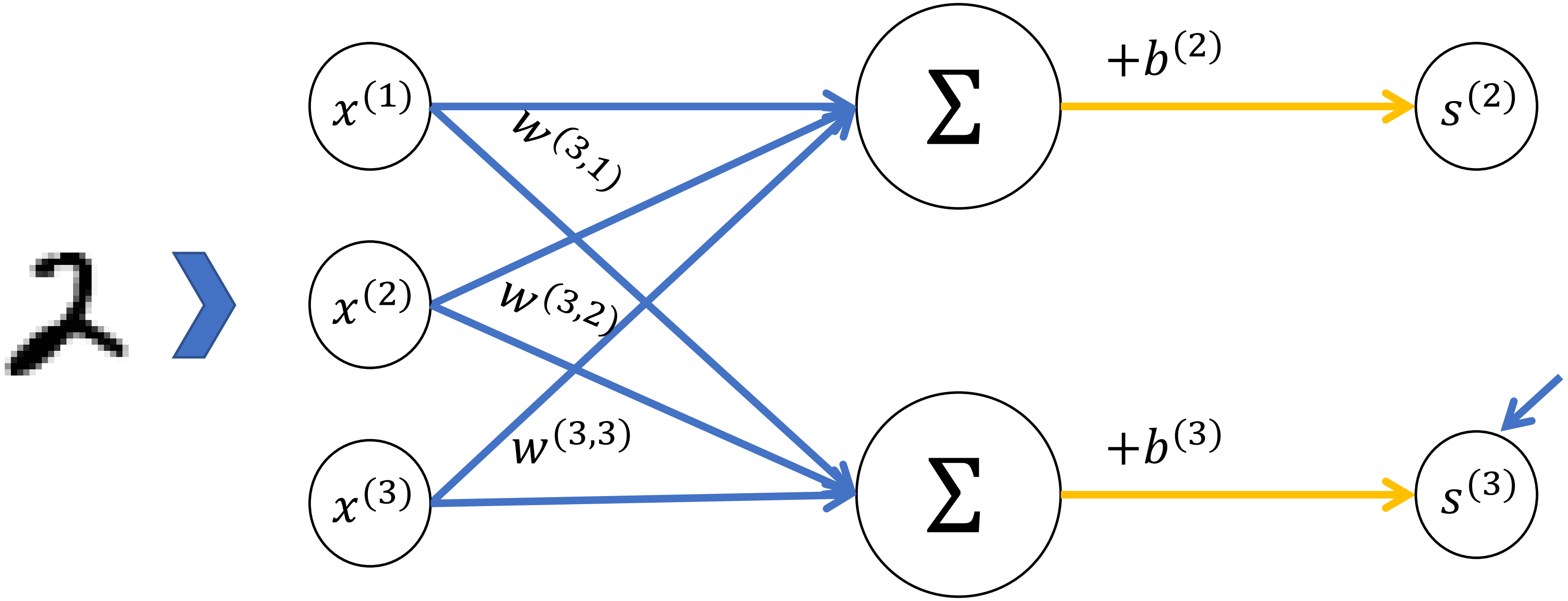
Recall: Linear Classification

We can build a multi-class linear classifier as multiple perceptrons (without the )



Recall: Linear Classification

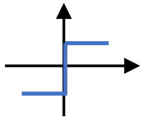
We can build a multi-class linear classifier as multiple perceptrons (without the)

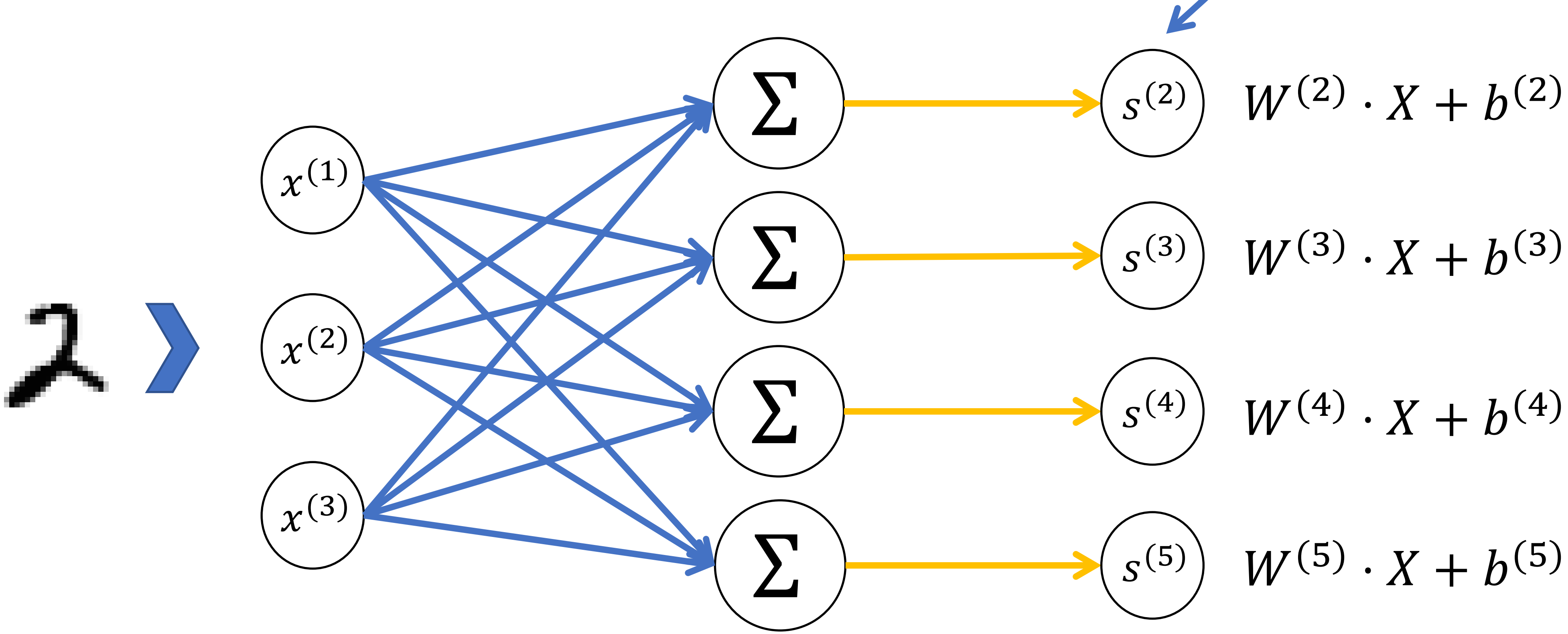


$$= \sum_i w^{(2,i)} x^{(i)} + b^{(2)}$$

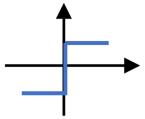
$$= \sum_i w^{(3,i)} x^{(i)} + b^{(3)}$$

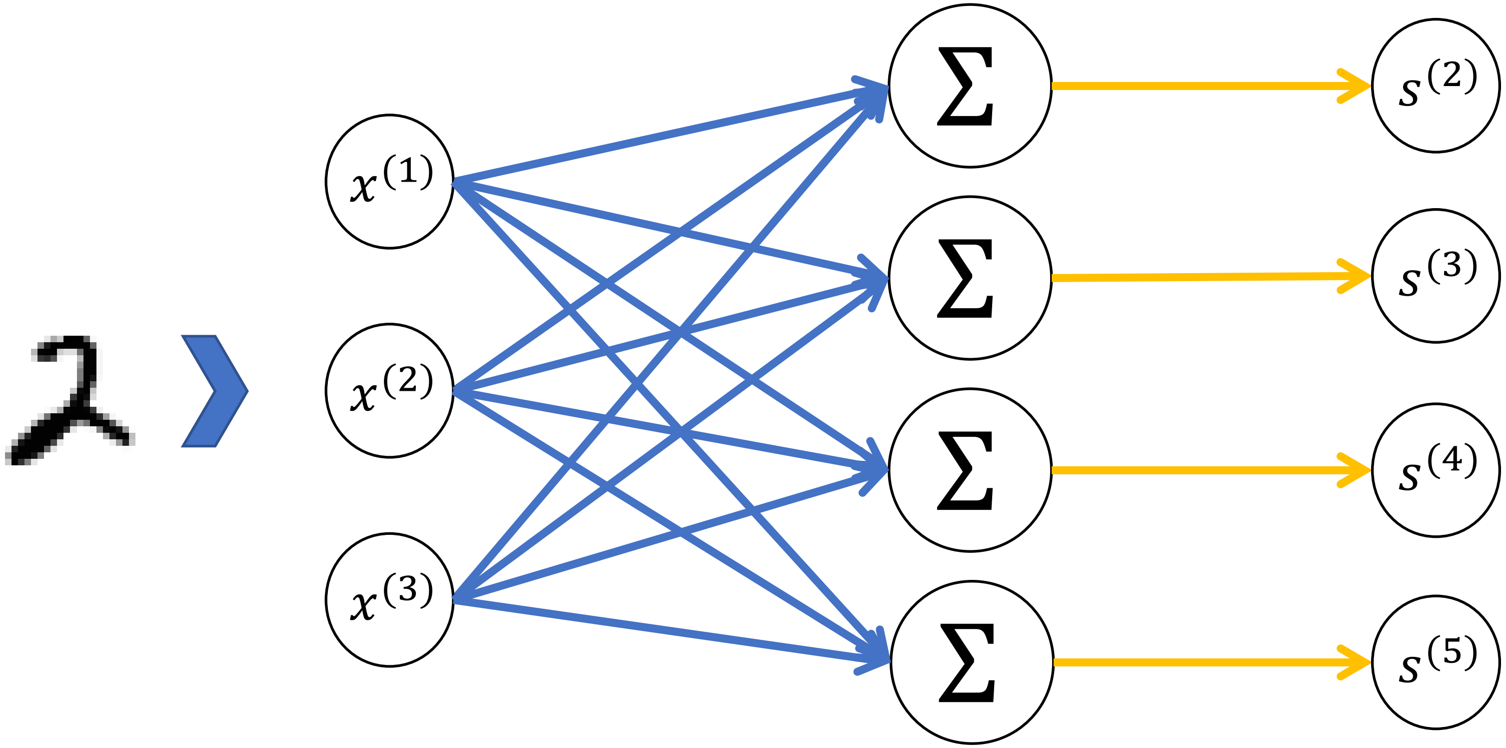
Recall: Linear Classification

We can build a multi-class linear classifier as multiple perceptrons (without the )



Recall: Linear Classification

We can build a multi-class linear classifier as multiple perceptrons (without the )



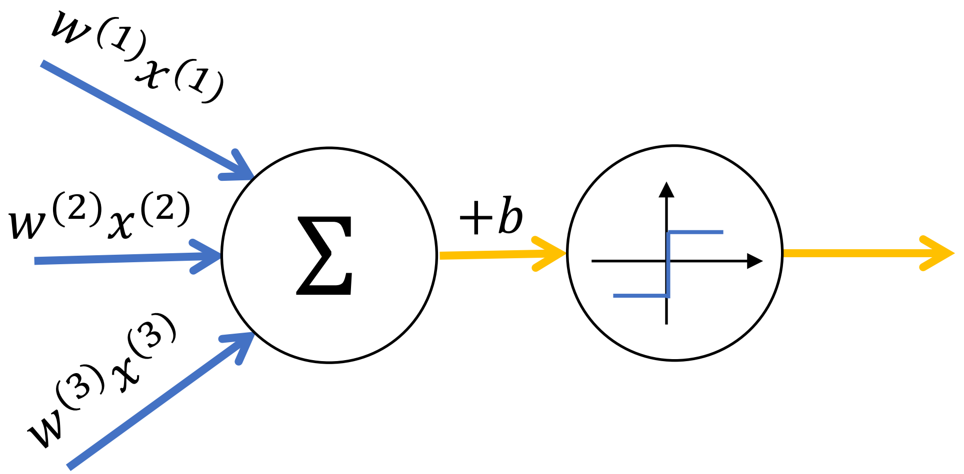
Last time, we saw we can get all the class scores with a matrix multiplication

$$S = W \times X + b$$

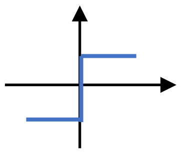
class scores

Building a Neural Network

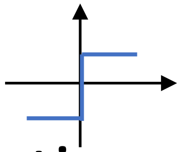
Our perceptron can *only represent linearly separable data*. But, a network of perceptrons can represent more complex functions.

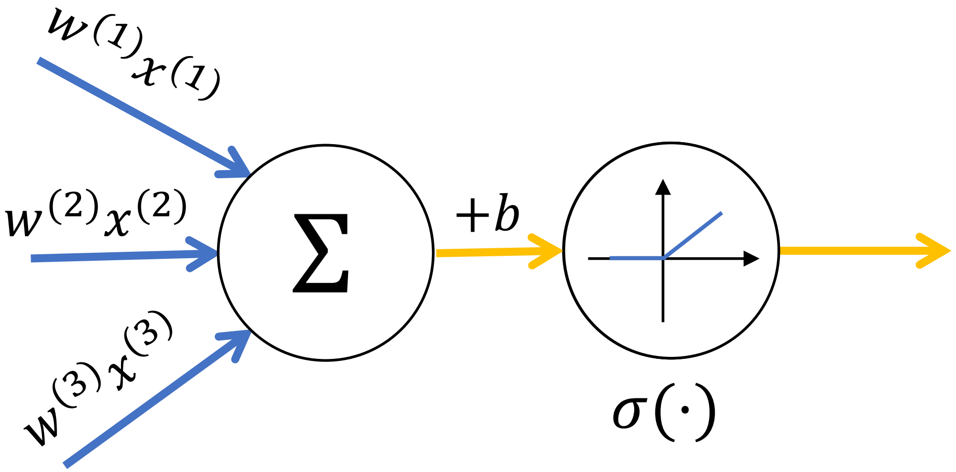


$$y_{pred} = \begin{cases} 1 & \text{if } W \cdot X + b \geq 0 \\ -1 & \text{if } W \cdot X + b < 0 \end{cases}$$

One more problem: This function  is not differentiable!

Building a Neural Network

We'll replace  with a continuous function, which will allow us to take the derivative of our loss function and apply Gradient Descent.

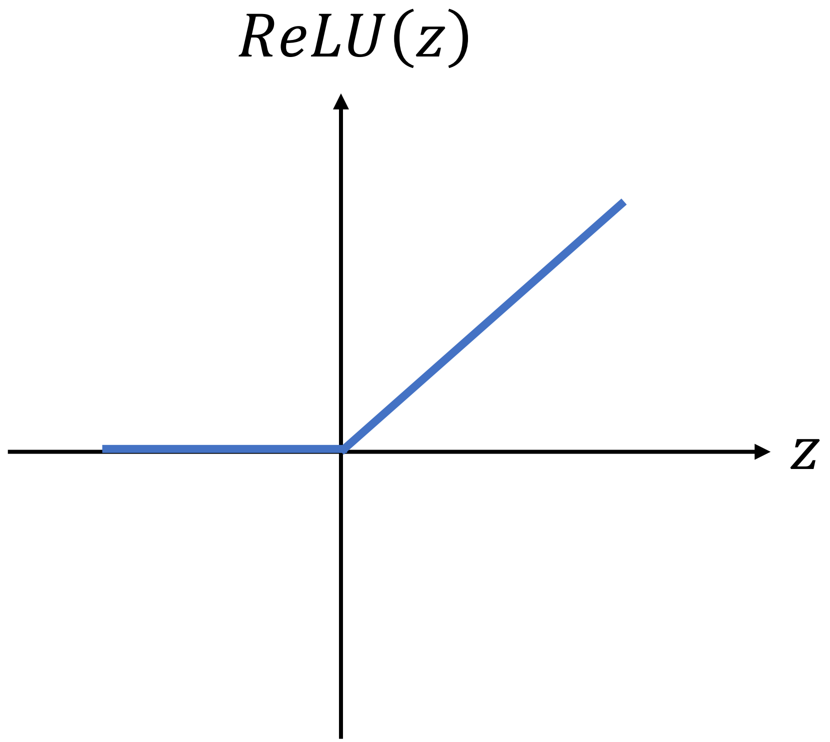


$$s = \begin{cases} W \cdot X + b & \text{if } W \cdot X + b \geq 0 \\ 0 & \text{if } W \cdot X + b < 0 \end{cases}$$

Our new function $\sigma(\cdot)$ is called the **activation function**.

Activation functions

We will be using the **ReLU activation function** (Rectified Linear Unit). This is one of the most common choices in modern neural networks.



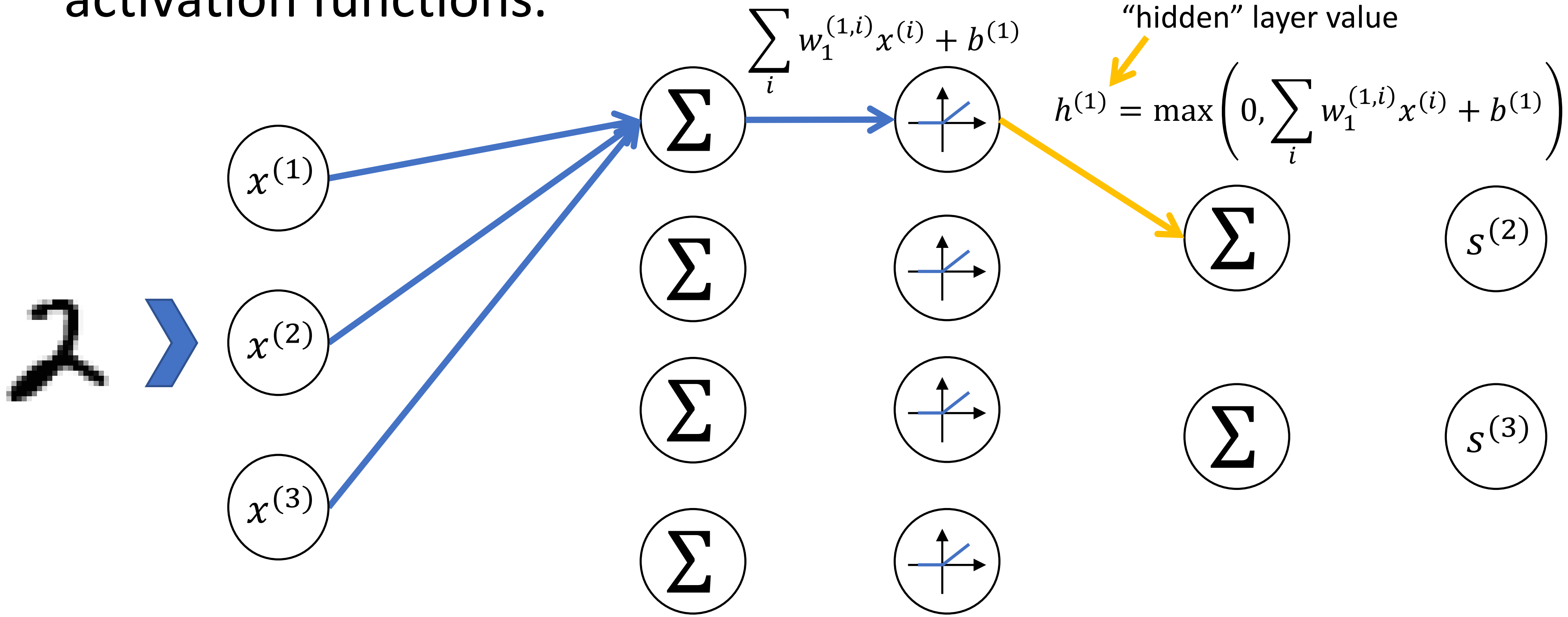
$$ReLU(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{else} \end{cases}$$

equivalent

$$ReLU(z) = \max(0, z)$$

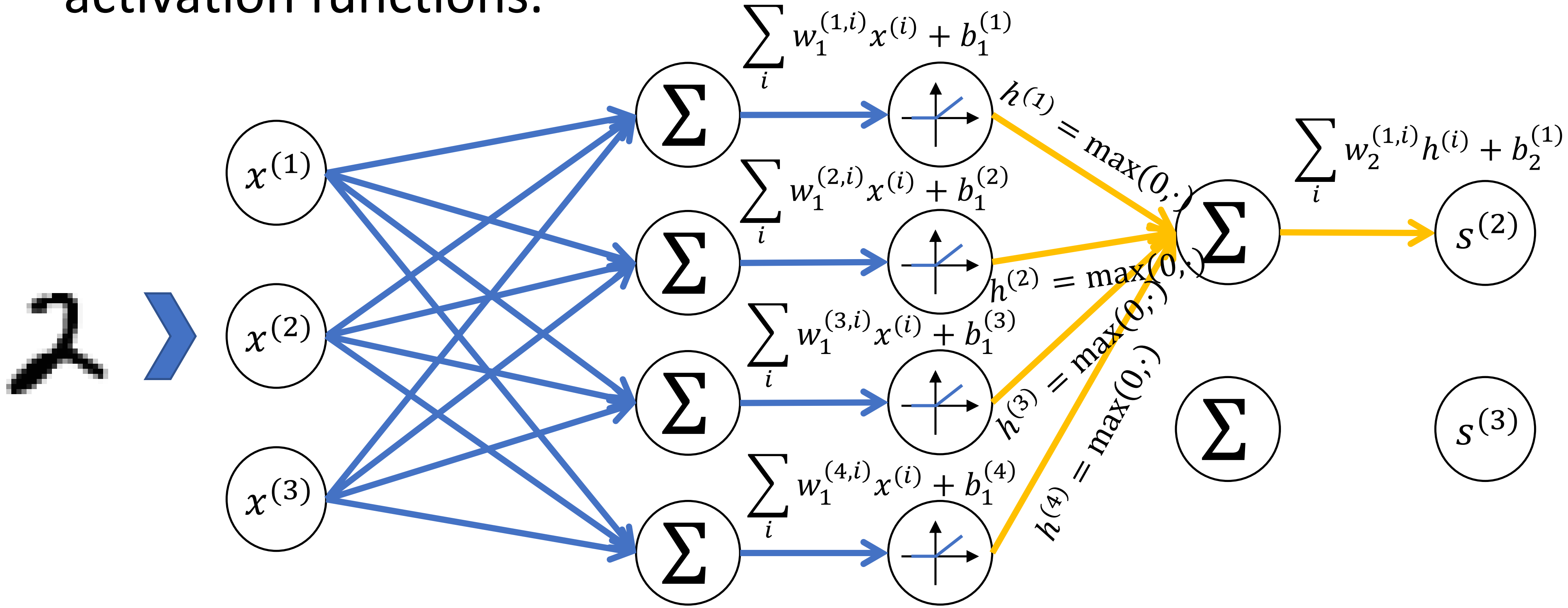
Building a Neural Network

We can stack up multiple "neurons" made up of linear functions and activation functions.



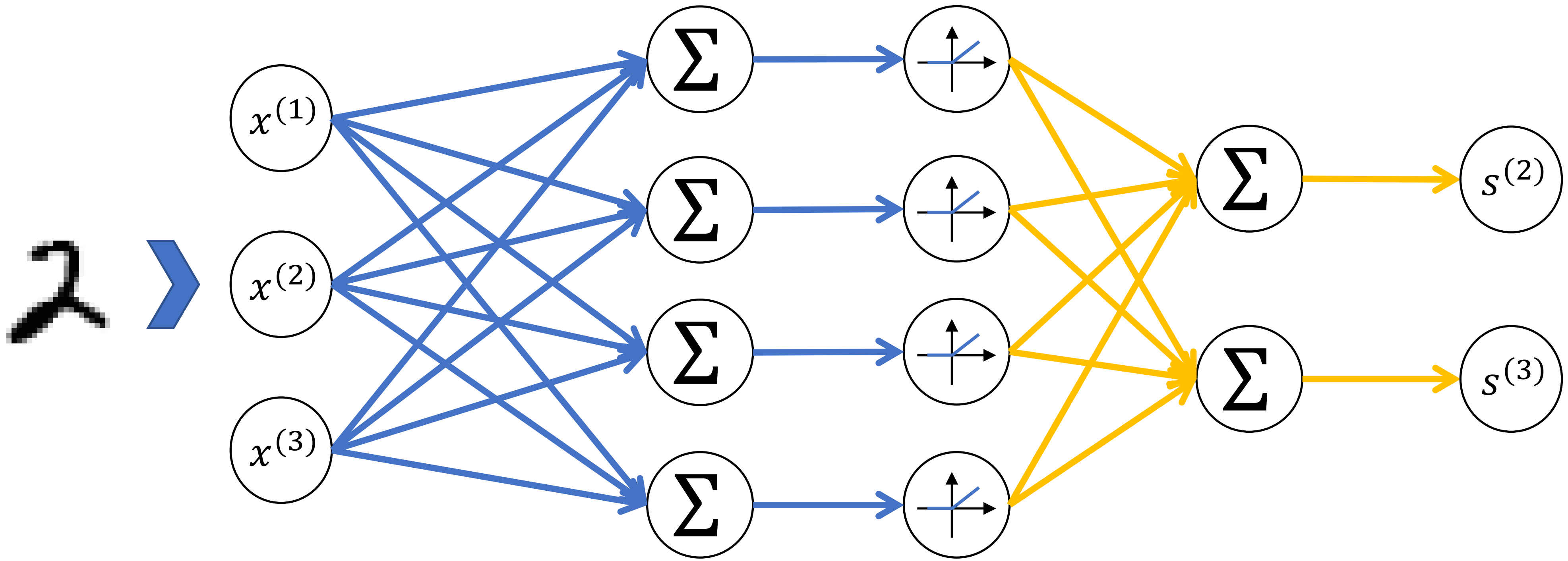
Building a Neural Network

We can stack up multiple “neurons” made up of linear functions and activation functions.



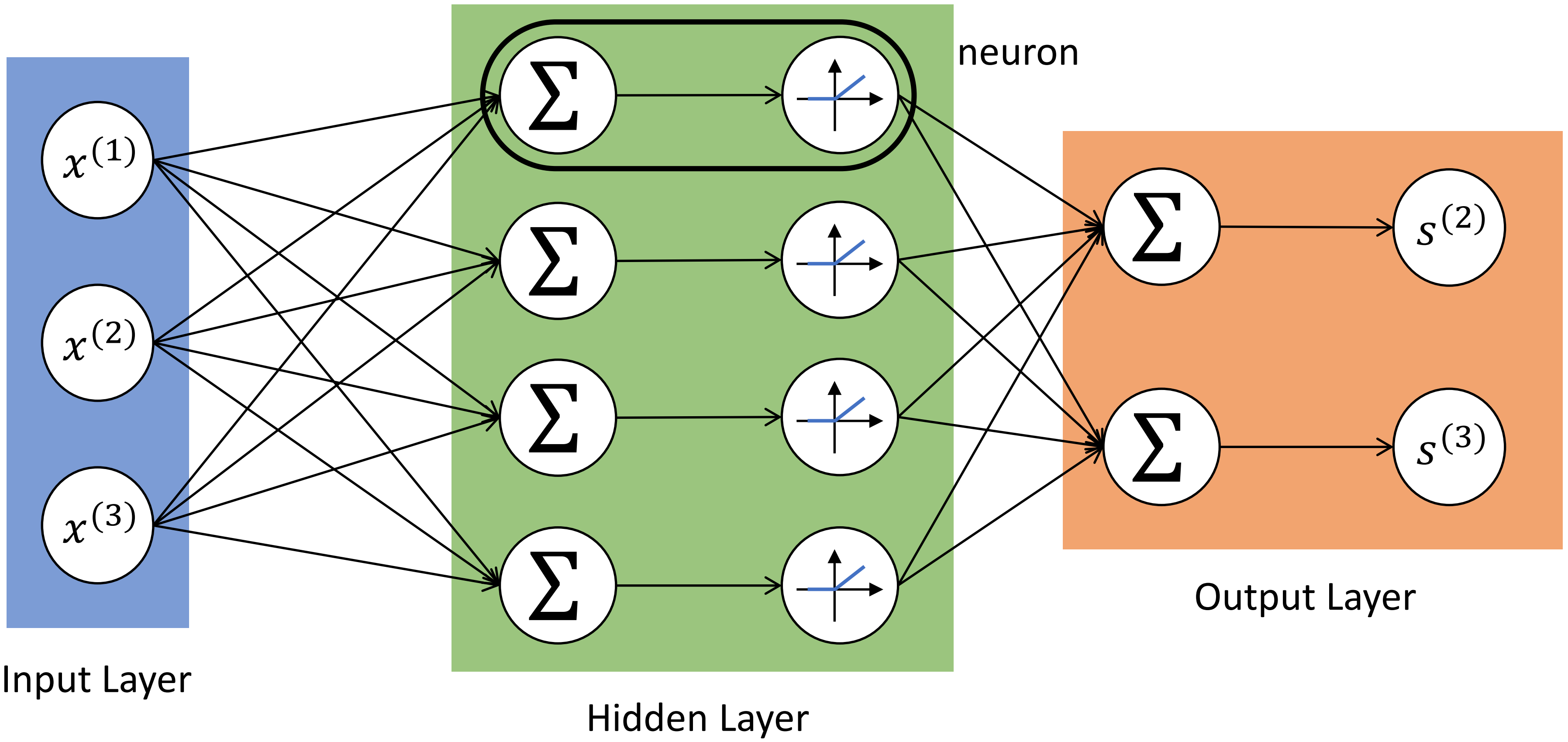
Building a Neural Network

We can stack up multiple “neurons” made up of linear functions and activation functions.



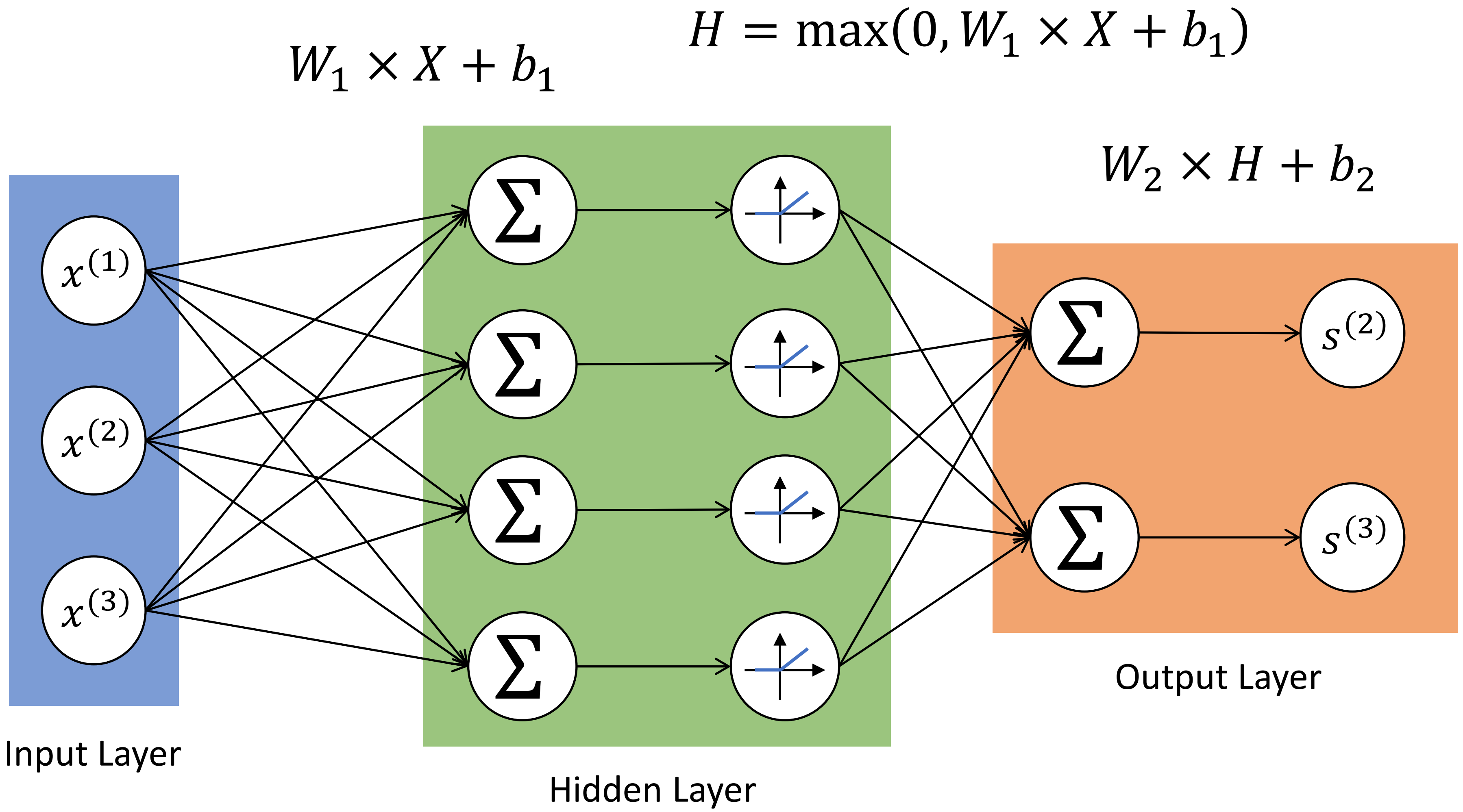
Building a Neural Network

Our final two-layer neural network looks like this:



Fully Connected Neural Network

We call this a "fully connected" network because each node is connected to all nodes in the previous layer.



Neural Network Computation

$$\underbrace{\begin{bmatrix} h^{(1)} \\ h^{(2)} \\ \vdots \\ h^{(J)} \end{bmatrix}}_{\text{Hidden layer}} = \max\left(\underset{W_1}{\overset{\text{Weights to learn}}{\begin{bmatrix} w_1^{(1,1)} & w_1^{(1,2)} & \cdots & w_1^{(1,D)} \\ w_1^{(2,1)} & w_1^{(2,2)} & \cdots & w_1^{(2,D)} \\ & \vdots & & \\ w_1^{(J,1)} & w_1^{(J,2)} & \cdots & w_1^{(J,D)} \end{bmatrix}}} \times \underset{X}{\overset{\text{Input image}}{\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(D)} \end{bmatrix}}} + \underset{b_1}{\overset{\text{Biases to learn}}{\begin{bmatrix} b_1^{(1)} \\ b_1^{(2)} \\ \vdots \\ b_1^{(J)} \end{bmatrix}}} ,\ 0 \right)$$

shapes: (J, D) $(D, 1)$ $(J, 1)$

rows columns

Recall: Matrix Multiplication in Julia

```
julia> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12

julia> size(A)
(4, 3)

julia> B = [3 3; 2 2; 1 1]
3×2 Matrix{Int64}:
 3  3
 2  2
 1  1

julia> size(B)
(3, 2)

julia>
```

```
julia> A * B
4×2 Matrix{Int64}:
 10  10
 28  28
 46  46
 64  64

julia> size(A * B)
(4, 2)

julia>
```

Legal!

The inner dimensions match:

(4, 3) x (3, 2) → (4, 2)

Recall: Matrix Multiplication in Julia

```
julia> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
4×3 Matrix{Int64}:
  1   2   3
  4   5   6
  7   8   9
 10  11  12

julia> size(A)
(4, 3)

julia> C = [1 1 1; 2 2 2]
2×3 Matrix{Int64}:
 1  1  1
 2  2  2

julia> size(C)
(2, 3)
```

```
julia> A * C
ERROR: DimensionMismatch("matrix A has dimensions (4,3), matrix B has dimensions (2,3)")
Stacktrace:
 [1] _generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{Int64}, B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})
   @ LinearAlgebra C:\buildbot\worker\package_win64\build\usr\share\julia\stdlib\v1.6\LinearAlgebra\src\matmul.jl:814
```

Illegal ☹

The inner dimensions don't match:

(4, 3) x (2, 3) → Fails!

Neural Network Computation

Weights to learn

Input image

Biases to learn

$$\begin{bmatrix} h^{(1)} \\ h^{(2)} \\ \vdots \\ h^{(J)} \end{bmatrix} = \max\left(\underset{W_1}{\begin{bmatrix} w_1^{(1,1)} & w_1^{(1,2)} & \cdots & w_1^{(1,D)} \\ w_1^{(2,1)} & w_1^{(2,2)} & \cdots & w_1^{(2,D)} \\ & \vdots & & \\ w_1^{(J,1)} & w_1^{(J,2)} & \cdots & w_1^{(J,D)} \end{bmatrix}} \times \underset{X}{\begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(D)} \end{bmatrix}} + \underset{b_1}{\begin{bmatrix} b_1^{(1)} \\ b_1^{(2)} \\ \vdots \\ b_1^{(J)} \end{bmatrix}}, \; 0 \right)$$

Hidden layer

shapes: (J, D) $(D, 1)$ $(J, 1)$

Inner dim. match!

Neural Network Computation

$$\begin{bmatrix} h^{(1)} \\ h^{(2)} \\ \vdots \\ h^{(J)} \end{bmatrix} = \begin{bmatrix} \max\left(0, \sum_i w_1^{(1,i)} x^{(i)} + b_1^{(1)}\right) \\ \max\left(0, \sum_i w_1^{(2,i)} x^{(i)} + b_1^{(2)}\right) \\ \vdots \\ \max\left(0, \sum_i w_1^{(J,i)} x^{(i)} + b_1^{(J)}\right) \end{bmatrix}$$

Hidden layer

Neural Network Computation

$$\begin{bmatrix} s^{(1)} \\ s^{(2)} \\ \vdots \\ s^{(K)} \end{bmatrix} = \begin{bmatrix} w_2^{(1,1)} & w_2^{(1,2)} & \cdots & w_2^{(1,J)} \\ w_2^{(2,1)} & w_2^{(2,2)} & \cdots & w_2^{(2,J)} \\ & \vdots & & \\ w_2^{(K,1)} & w_2^{(K,2)} & \cdots & w_2^{(K,J)} \end{bmatrix} \times \begin{bmatrix} h^{(1)} \\ h^{(2)} \\ \vdots \\ h^{(J)} \end{bmatrix} + \begin{bmatrix} b_2^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_2^{(K)} \end{bmatrix}$$

K = number of classes

	W_2	H	b_2
shapes:	(K, J)	$(J, 1)$	$(K, 1)$

Inner dim. match!

Neural Network Computation

$$\begin{bmatrix} s^{(1)} \\ s^{(2)} \\ \vdots \\ s^{(K)} \end{bmatrix} = \begin{bmatrix} \sum_i w_{\mathbf{2}}^{(1,i)} h^{(i)} + b_2^{(1)} \\ \sum_i w_2^{(2,i)} h^{(i)} + b_2^{(2)} \\ \vdots \\ \sum_i w_2^{(K,i)} h^{(i)} + b_2^{(K)} \end{bmatrix}$$

Class scores

Neural Network Computation

We can write a two-layer neural network as one big matrix multiplication:

Activation function

$$\text{scores} = \underbrace{W_2 \times \max(0, \underbrace{W_1 \times X + b_1}_{\text{First linear layer}}) + b_2}_{\text{Second linear layer}}$$

Why do we need an activation function?

Let's look at our neural network equation:

$$\text{scores} = W_2 \times \max(0, W_1 \times X + b_1) + b_2$$

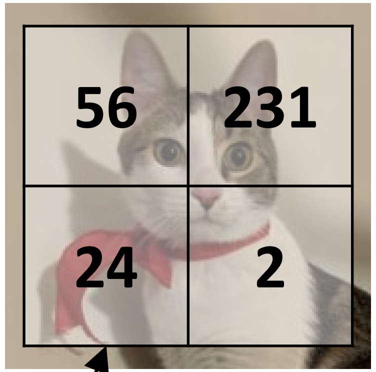
What if we removed the activation function?

$$\begin{aligned}\text{scores} &= W_2 \times (W_1 \times X + b_1) + b_2 \\ &= \underbrace{(W_2 \times W_1)}_{W'} \times X + \underbrace{(W_2 \times b_1 + b_2)}_{b'}\end{aligned}$$

This is still just a linear classifier!

Exercise:

Image:



Pixel values

Parameters:

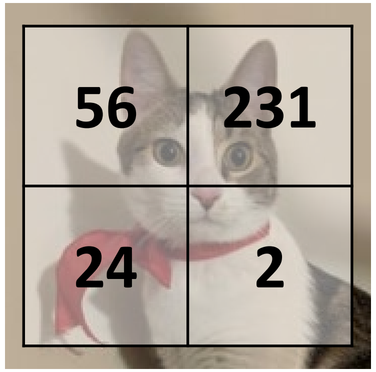
$$W_1 = \begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ -0.2 & 0.0 & 0.4 & 1.3 \\ 1.1 & 0.7 & -0.9 & -0.2 \\ -1.9 & 1.0 & 0.9 & 2.1 \end{bmatrix} \quad b_1 = \begin{bmatrix} 9.1 \\ 7.3 \\ -27 \\ 2.1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.8 & 0.9 & 0.0 & 1.4 \\ 0.3 & -0.3 & 2.2 & -1.2 \\ 0.6 & 0.7 & -0.1 & -0.6 \end{bmatrix} \quad b_2 = \begin{bmatrix} -32 \\ 14 \\ 19 \end{bmatrix}$$

s_{cat} = ?
s_{dog} = ?
s_{bird} = ?

Find the class scores!

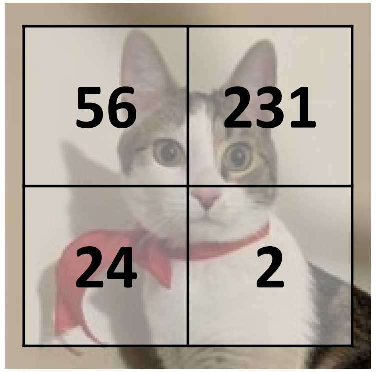
Exercise:



$$\begin{bmatrix} h^{(1)} \\ h^{(2)} \\ h^{(3)} \\ h^{(4)} \end{bmatrix} = \max\left(\underbrace{\begin{bmatrix} 0.2 & -0.5 & 0.1 & 2.0 \\ -0.2 & 0.0 & 0.4 & 1.3 \\ 1.1 & 0.7 & -0.9 & -0.2 \\ -1.9 & 1.0 & 0.9 & 2.1 \end{bmatrix}}_{W_1} \times \underbrace{\begin{bmatrix} 56 \\ 231 \\ 24 \\ 2 \end{bmatrix}}_{X} + \underbrace{\begin{bmatrix} 9.1 \\ 7.3 \\ -27 \\ 2.1 \end{bmatrix}}_{b_1},\ 0 \right)$$

Hidden layer

Exercise:



$h^{(1)}$
$h^{(2)}$
$h^{(3)}$
$h^{(4)}$

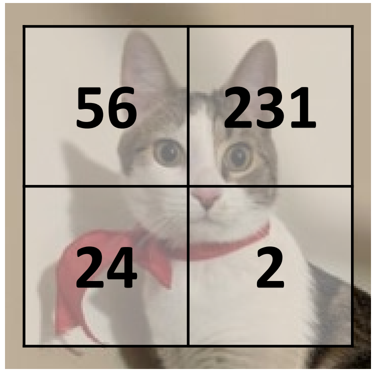
```
julia> X = [56; 231; 24; 2]
4-element Vector{Int64}:
  56
 231
  24
   2

julia> W1 = [0.2 -0.5 0.1 2.0; -0.2 0.0 0.4 1.3; 1.1 0.7 -0.9 -0.2; -1.9 1.0 0.9 2.1]
4×4 Matrix{Float64}:
  0.2  -0.5   0.1   2.0
 -0.2   0.0   0.4   1.3
  1.1   0.7  -0.9  -0.2
 -1.9   1.0   0.9   2.1

julia> b1 = [9.1; 7.3; -27; 2.1]
4-element Vector{Float64}:
   9.1
   7.3
 -27.0
   2.1

julia>
```

```
julia> W1 * X + b1
```

```
julia> H = max.(0, W1 * X + b1)

julia>
```

$$H = \max(0, W_1 \times X + b_1)$$

Exercise:



$h^{(1)} = 0$
$h^{(2)} = 8.3$
$h^{(3)} = 174.3$
$h^{(4)} = 152.5$

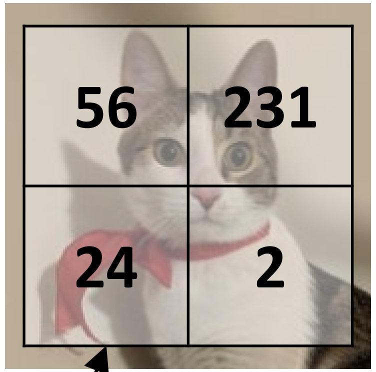
```
julia> X = [56; 231; 24; 2]
4-element Vector{Int64}:
  56
 231
  24
   2

julia> W1 = [0.2 -0.5 0.1 2.0; -0.2 0.0 0.4 1.3; 1.1 0.7 -0.9 -0.2; -1.9 1.0 0.9 2.1]
4×4 Matrix{Float64}:
  0.2  -0.5   0.1   2.0
 -0.2   0.0   0.4   1.3
  1.1   0.7  -0.9  -0.2
 -1.9   1.0   0.9   2.1

julia> b1 = [9.1; 7.3; -27; 2.1]
4-element Vector{Float64}:
   9.1
   7.3
 -27.0
   2.1

julia>
```

```
julia> W1 * X + b1
4-element Vector{Float64}:
 -88.8
   8.299999999999999
 174.29999999999998
 152.5
```

```
julia> H = max.(0, W1 * X + b1)
4-element Vector{Float64}:
   0.0
   8.299999999999999
 174.29999999999998
 152.5

julia>
```

Exercise:

Image:



Pixel values

s_{cat} = ?
s_{dog} = ?
s_{bird} = ?

Parameters:

W_1

0.2	-0.5	0.1	2.0
-0.2	0.0	0.4	1.3
1.1	0.7	-0.9	-0.2
-1.9	1.0	0.9	2.1

b_1

9.1
7.3
-27
2.1

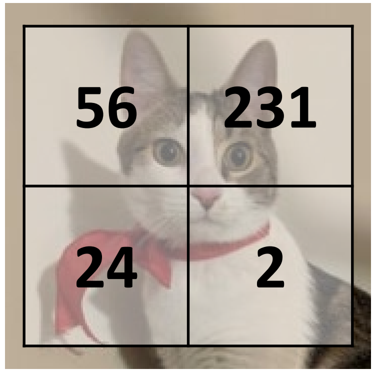
W_2

0.8	0.9	0.0	1.4
0.3	-0.3	2.2	-1.2
0.6	0.7	-0.1	-0.6

b_2

-32
14
19

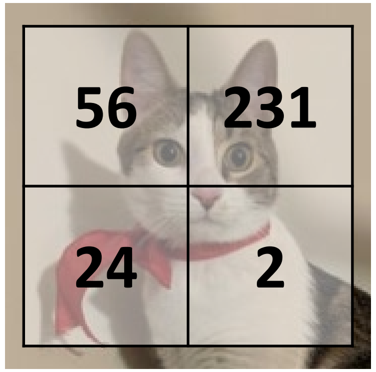
Exercise:



$$\begin{bmatrix} s_{cat} \\ s_{dog} \\ s_{bird} \end{bmatrix} = \underbrace{\begin{bmatrix} 0.8 & 0.9 & 0.0 & 1.4 \\ 0.3 & -0.3 & 2.2 & -1.2 \\ 0.6 & 0.7 & -0.1 & -0.6 \end{bmatrix}}_{W_2} \times \underbrace{\begin{bmatrix} 0 \\ 8.3 \\ 174.3 \\ 152.5 \end{bmatrix}}_{H} + \underbrace{\begin{bmatrix} -32 \\ 14 \\ 19 \end{bmatrix}}_{b_2}$$

Class scores

Exercise:



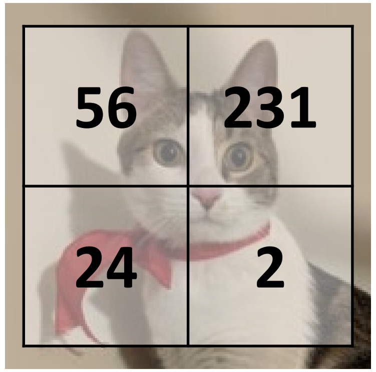

```
julia> W2 = [0.8 0.9 0.0 1.4; 0.3 -0.3 2.2 -1.2; 0.6 0.7 -0.1 -0.6]
3×4 Matrix{Float64}:
 0.8   0.9   0.0   1.4
 0.3  -0.3   2.2  -1.2
 0.6   0.7  -0.1  -0.6

julia> b2 = [-32; 14; 19]
3-element Vector{Int64}:
 -32
  14
  19
```

s_{cat}
s_{dog}
s_{bird}

```
julia> scores = W2 * H + b2
```

$$W_2 \times H + b_2$$

Exercise:




```
julia> W2 = [0.8 0.9 0.0 1.4; 0.3 -0.3 2.2 -1.2; 0.6 0.7 -0.1 -0.6]
3×4 Matrix{Float64}:
 0.8   0.9   0.0   1.4
 0.3  -0.3   2.2  -1.2
 0.6   0.7  -0.1  -0.6

julia> b2 = [-32; 14; 19]
3-element Vector{Int64}:
 -32
  14
  19
```

$s_{cat} = 188.97$
→ $s_{dog} = 211.97$
$s_{bird} = -84.12$

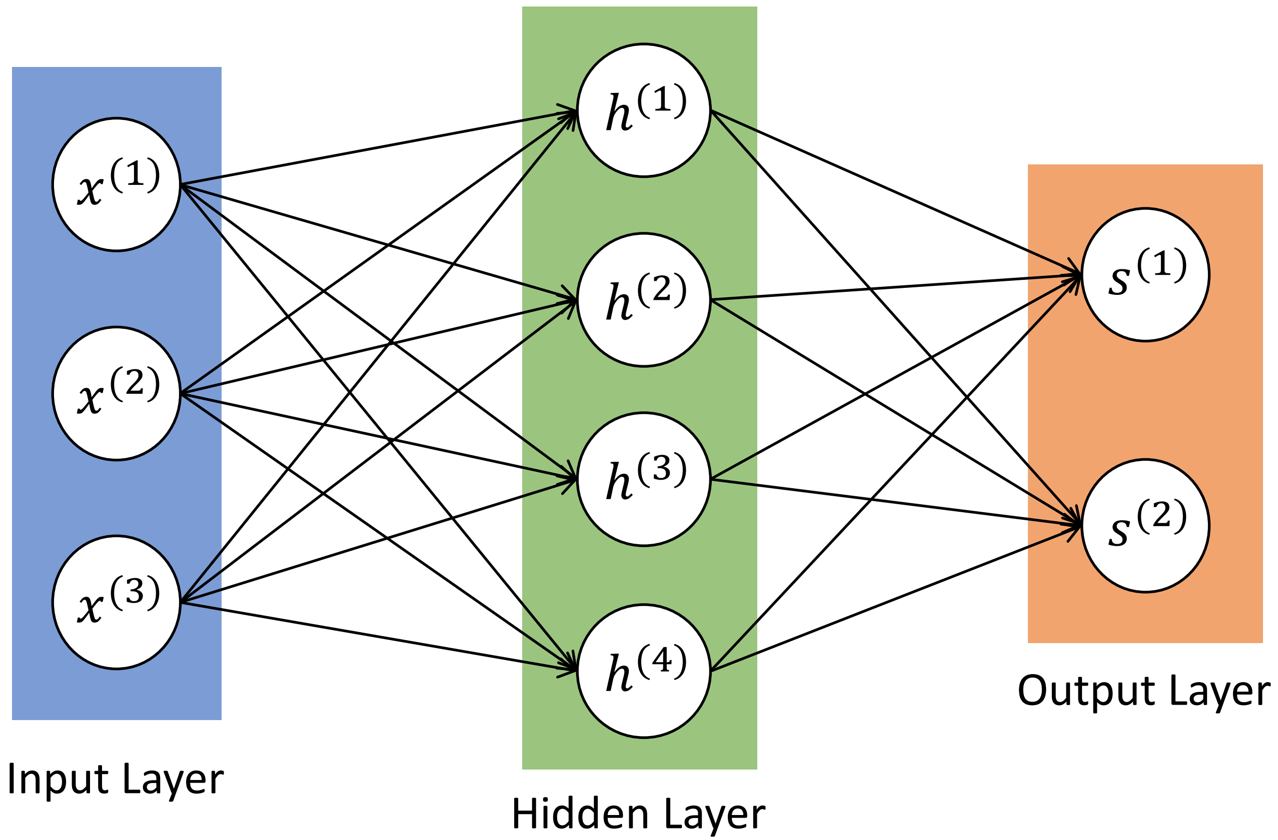
```
julia> scores = W2 * H + b2
3-element Vector{Float64}:
 188.97
 211.96999999999997
 -84.11999999999999
```

```
julia> argmax(scores)
2

julia>
```

Predicted class: **2 = dog**

Fully Connected Neural Network

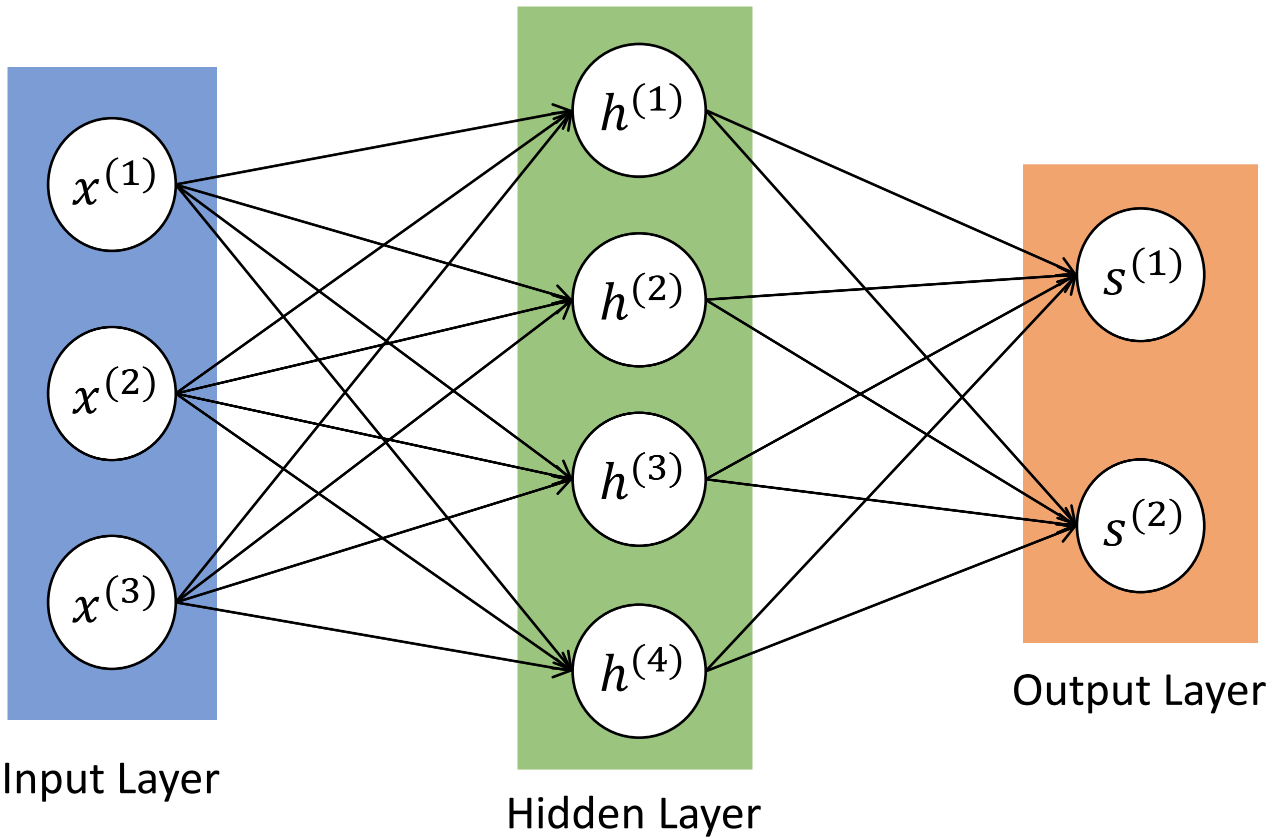
This is a **two-layer** neural network (the input layer isn't counted).



Fully Connected Neural Network

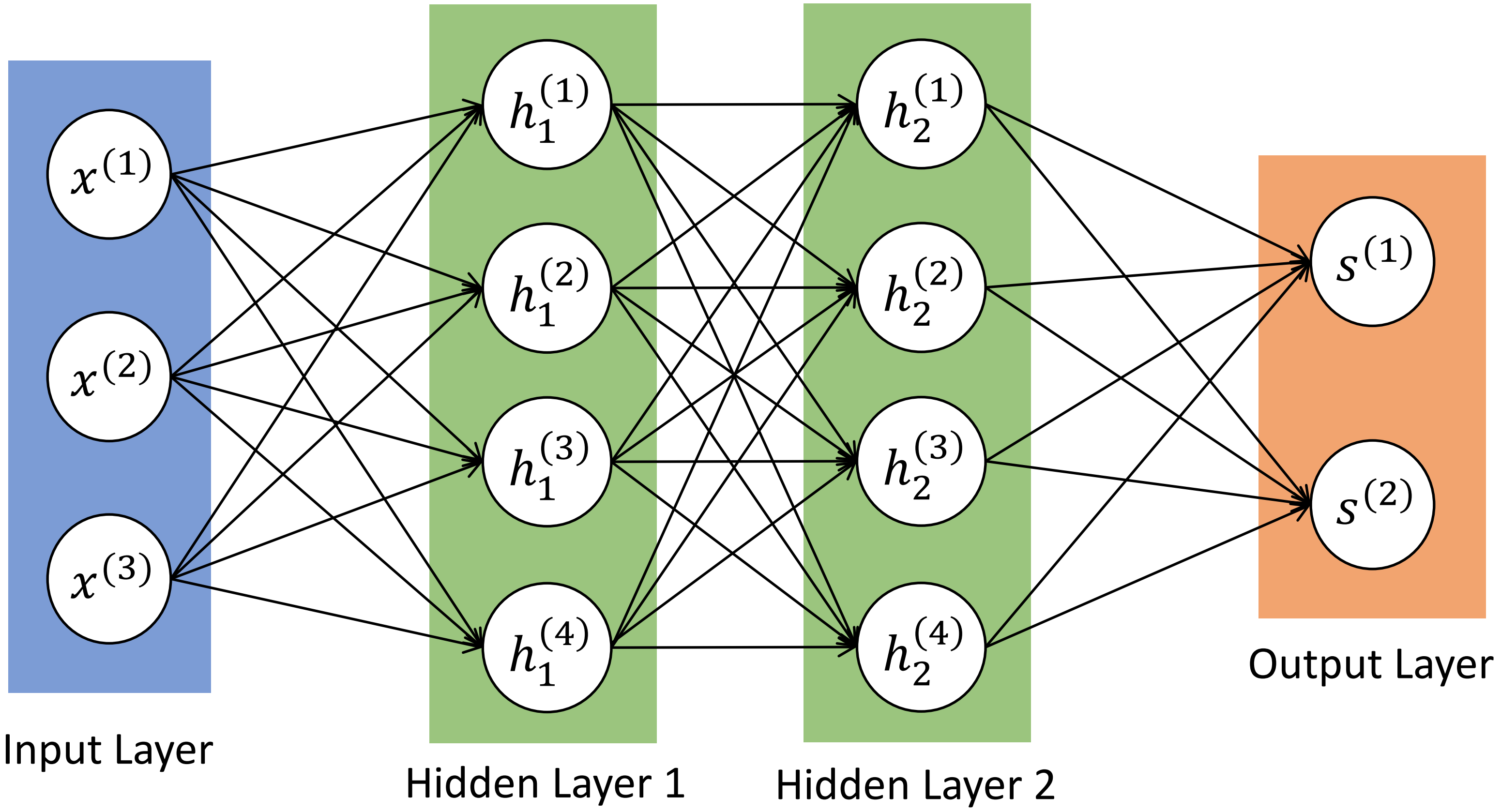
Parameters to learn:
W_1, W_2, b_1, b_2

$$\text{scores} = W_2 \times \max(0, W_1 \times X + b_1) + b_2$$



Fully Connected Neural Network

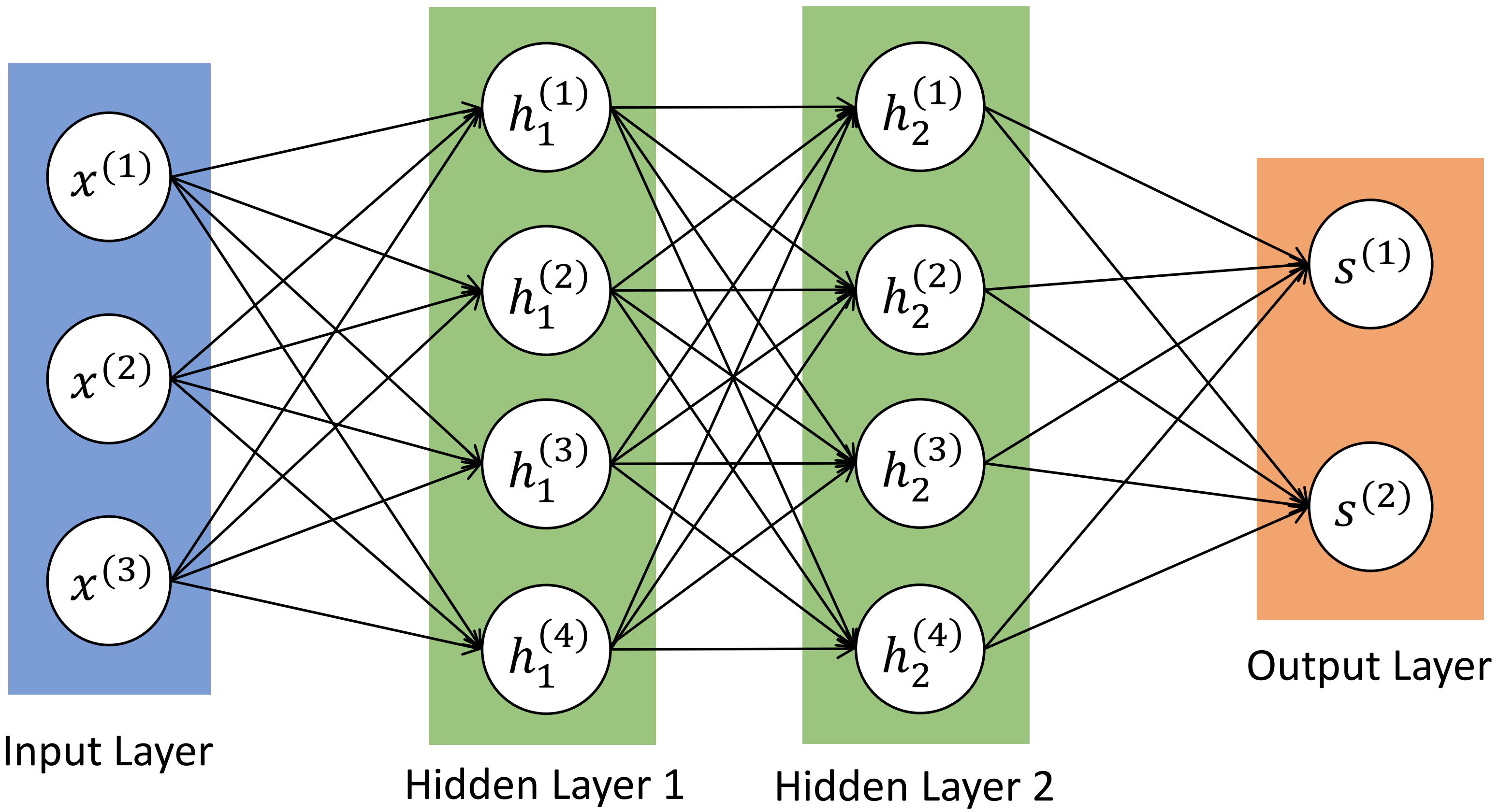
A three-layer neural network looks like this:



Fully Connected Neural Network

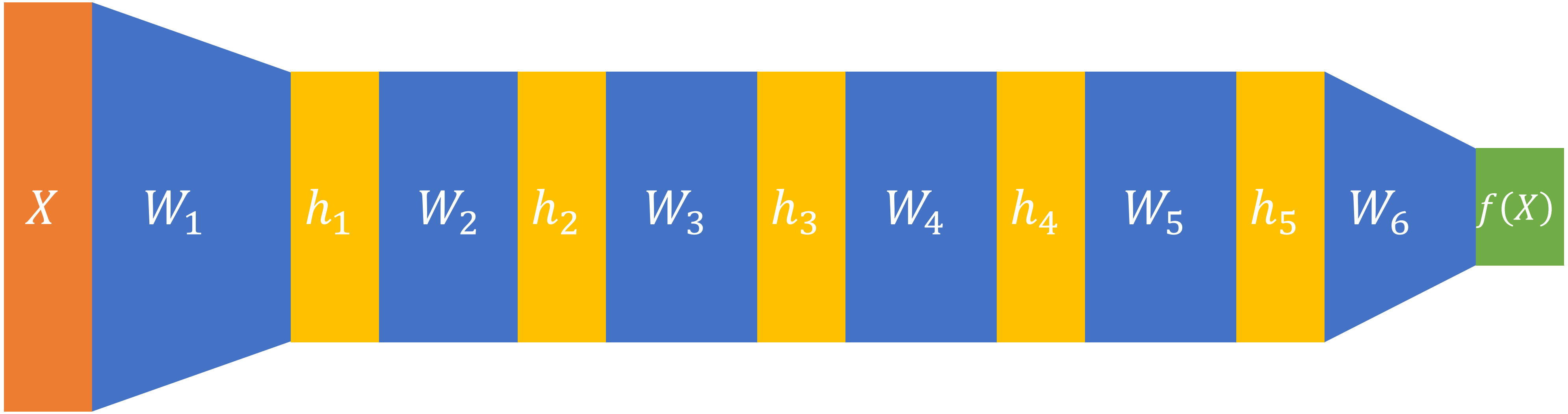
Parameters to learn:
$W_1, W_2, W_3, b_1, b_2, b_3$

$$\text{scores} = W_3 \times \max(0, W_2 \times \max(0, W_1 \times X + b_1) + b_2) + b_3$$



Deep Neural Networks

The **number of hidden layers** and the **size of each hidden layer** are hyperparameters we need to pick.



Back to Project 4…

P4.3: Forward Pass

We call the computation of the scores the **forward pass** because we are moving *forward* through the graph.

```
function nn_forward(params, X)
    W1, b1 = params["W1"], params["b1"]
    W2, b2 = params["W2"], params["b2"]

    scores, hidden = nothing, nothing

    # TODO: Calculate the scores and values after the first ReLU layer.

    return scores, hidden
end
```

Put scores and H in these variables.

Your turn!
Calculate the scores here.
Also return H.

This is a typo!!
hidden_layer -> hidden

Loss Function

We already have a function to tell us how good we are doing at classifying our images: the SVM loss!

Incorrect class score

Correct class score

$$L = \sum_{\forall\, i \setminus y} \begin{cases} f_i(X) + \Delta - f_y(X) & \text{if } f_y(X) < f_i(X) + \Delta \\ 0 & \text{otherwise} \end{cases}$$

In English: For each incorrect class, add its loss to the total, if it wasn't less than the correct class by the margin.

We will reuse the same loss function as for the linear classifier!

Regularization

We will apply regularization to the weight matrix just like in the linear classifier.

But this time, we have two weight matrices!

$$L_{reg}(W_1, W_2) = \alpha \left(\sum_{i=1}^{D \times J} \left(w_1^{(i)} \right)^2 + \sum_{i=1}^{J \times K} \left(w_2^{(i)} \right)^2 \right)$$

Regularization coefficient
(need to tune this)

Sum of all the squared weights in the weight matrices

P4.3: Loss & Regularization

This will look very similar to the loss function in the linear classifier!

```
function nn_svm_loss(params, X, y, reg=0)
    W1, b1 = params["W1"], params["b1"]
    W2, b2 = params["W2"], params["b2"]

    N, D = size(X)

    loss = 0
    scores, hidden = nothing, nothing

    # TODO: Use the nn_forward() function to perform the forward pass, then
    # calculate the  svm  loss. Remember the regularization term on both
    # weight matrices.

    # Get the gradients.
    grads = nn_svm_grad(params, X, y, scores, hidden, reg)

    return loss, grads
end
```

Replace with your computed value.

Your turn!
Calculate the loss here.

Provided gradient computation

Updating the Weights

We know that our fully connected neural network is differentiable, so we can analytically calculate the gradients for each parameter.

Gradients are provided for you in P4.3.

```
function nn_svm_grad(params, X, y, scores, hidden, reg=0)
    W1, b1 = params["W1"], params["b1"]
    W2, b2 = params["W2"], params["b2"]

⋮

    grads = Dict([("W1", dW1), ("b1", db1), ("W2", dW2), ("b2", db2)])
    return grads
end
```

Updating the Weights

We know that our fully connected neural network is differentiable, so we can analytically calculate the gradients for each parameter.

Gradients are provided for you in P4.3.

We can use Gradient Descent to update W_1, b_1, W_2 and b_2 just like we did in P4.2 (no bias trick this time!)

Putting it all together

Training algorithm:

Initialize weights to small values drawn from a normal distribution

```
W1, W2 ← randn(J, D)*eps, randn(K, J)*eps
b1, b2 ← zeros(J), zeros(K)
for iteration in 1:N do:
      loss = SVM_loss(W1, W2, b1, b2, X, y)
      dW1,dW2,db1,db2 = nn_grads(W1,W2,b1,b2,X,y)
      W1 = W1 - step_size * dW1
      b1 = b1 - step_size * db1
      W2 = W2 - step_size * dW2
      b2 = b2 - step_size * db2
```

Initialize biases to zero

Putting it all together

Training algorithm:

Initialize weights to small values drawn from a normal distribution

```
W1, W2 ← randn(J, D)*eps, randn(K, J)*eps
b1, b2 ← zeros(J), zeros(K)
for iteration in 1:N do:
    loss = SVM_loss(W1, W2, b1, b2, X, y)
    dW1,dW2,db1,db2 = nn_grads(W1,W2,b1,b2,X,y)
    W1 = W1 - step_size * dW1
    b1 = b1 - step_size * db1
    W2 = W2 - step_size * dW2
    b2 = b2 - step_size * db2
```

Initialize biases to zero

For a fixed number of iterations…

Putting it all together

Training algorithm:

Initialize weights to small values drawn from a normal distribution

```
W1, W2 ← randn(J, D)*eps, randn(K, J)*eps
b1, b2 ← zeros(J), zeros(K)
for iteration in 1:N do:
    loss = SVM_loss(W1, W2, b1, b2, X, y)
    dW1,dW2,db1,db2 = nn_grads(W1,W2,b1,b2,X,y)
    W1 = W1 - step_size * dW1
    b1 = b1 - step_size * db1
    W2 = W2 - step_size * dW2
    b2 = b2 - step_size * db2
```

Initialize biases to zero

For a fixed number of iterations…

images

labels

Sample a batch!

Calculate the loss.

Putting it all together

Training algorithm:

Initialize weights to small values drawn from a normal distribution

```
W1, W2 ← randn(J, D)*eps, randn(K, J)*eps
b1, b2 ← zeros(J), zeros(K)
for iteration in 1:N do:
    loss = SVM_loss(W1, W2, b1, b2, X, y)
    dW1,dW2,db1,db2 = nn_grads(W1,W2,b1,b2,X,y)
    W1 = W1 - step_size * dW1
    b1 = b1 - step_size * db1
    W2 = W2 - step_size * dW2
    b2 = b2 - step_size * db2
```

Initialize biases to zero

Sample a batch!

images

labels

Calculate the loss.

For a fixed number of iterations…

Calculate the gradients.

Putting it all together

Training algorithm:

Initialize weights to small values drawn from a normal distribution

```
W1, W2 ← randn(J, D)*eps, randn(K, J)*eps
b1, b2 ← zeros(J), zeros(K)
for iteration in 1:N do:
    loss = SVM_loss(W1, W2, b1, b2, X, y)
    dW1,dW2,db1,db2 = nn_grads(W1,W2,b1,b2,X,y)
    W1 = W1 - step_size * dW1
    b1 = b1 - step_size * db1
    W2 = W2 - step_size * dW2
    b2 = b2 - step_size * db2
```

Initialize biases to zero

Sample a batch!

images

labels

For a fixed number of iterations...

Calculate the loss.

Calculate the gradients.

Update the parameters with gradient descent

Putting it all together

```julia
function train_nn(params, X, y, num_classes, lr=0.01, reg=1e-3, batch=20, num_iters=100, print_freq=100)
    N, D = size(X)

    losses = zeros(num_iters)

    for it in 1:num_iters

        # TODO: Sample a random batch, calculate the loss and gradients, and update
        # all the weights in params. Place the loss for this iteration at
        # losses[it].

        if it % print_freq == 0
            println("Iteration ", it, ": average loss = ", sum(losses) / it)
        end
    end

    return losses, params
end
```

Your turn!
Write the training loop here.

Tuning hyperparameters

The regularization coefficient and learning rate need to be tuned, like in P4.2.

The number of nodes (or neurons) in the hidden layer can also be tuned.

See last lecture (Lecture 12 on Optimization) for how to tune parameters.

```
# Define constants.
hidden_dims = 36
num_iters = 1500
batch = 20
reg = 1e-5
lr = 1e-1

# Initialize weights.
W1 = 1e-4 * randn(DIM, hidden_dims)
b1 = zeros(1, hidden_dims)
W2 = 1e-4 * randn(hidden_dims, num_classes)
b2 = zeros(1, num_classes)
params = Dict([("W1", W1), ("b1", b1), ("W2", W2), ("b2", b2)])

# Train the network.
losses, params = train_nn(params, x_train, y_train, num_classes, lr, reg, batch, num_iters)

# Plot the losses.
plot(1:num_iters, losses)
```

hyperparameters

This line calls your training function. `params` contains the optimized weights when it's finished.

Backpropagation

Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

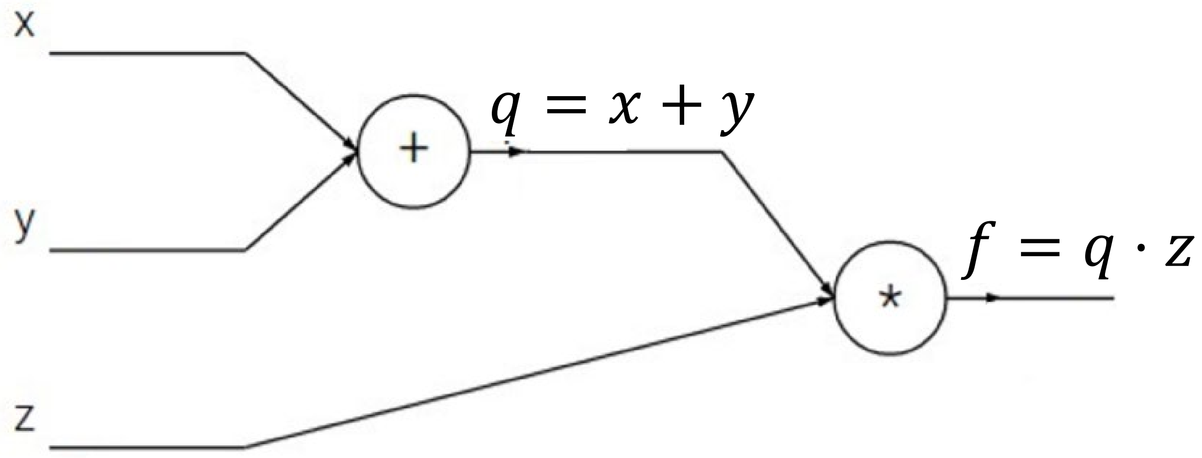
It uses the **chain rule** (from calculus).

Backpropagation

Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

Step 1: Express a function as a graph.

$$f(x, y, z) = (x + y) \cdot z$$



Backpropagation

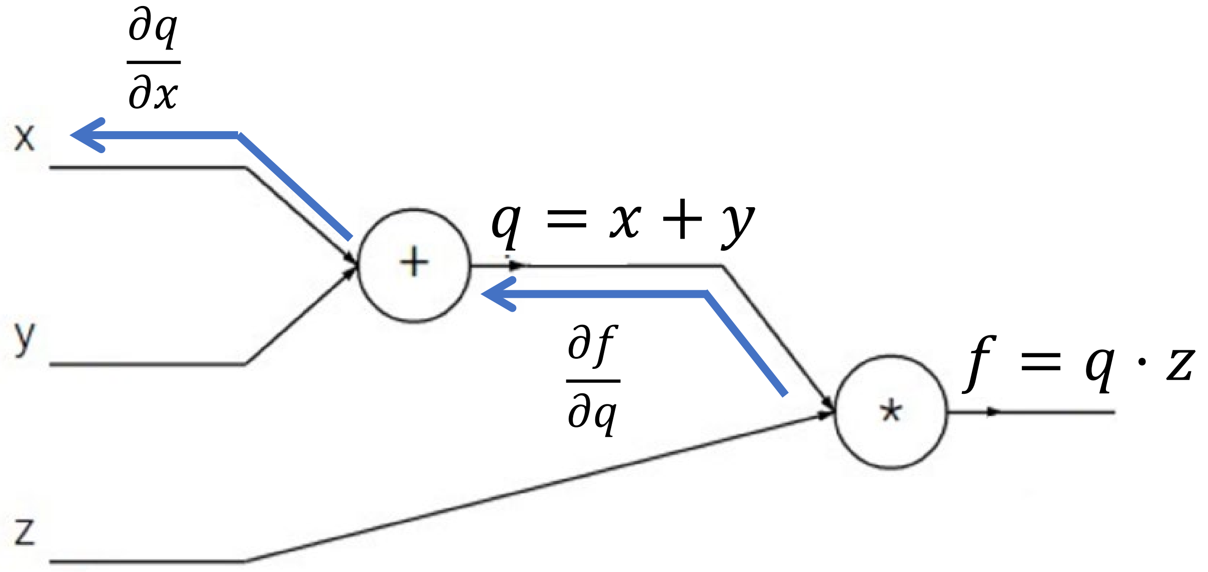
Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

Step 2: Compute gradients.

$$f(x, y, z) = (x + y) \cdot z$$

$$\frac{\partial f}{\partial x} = \frac{\partial q}{\partial x} \frac{\partial f}{\partial q} = 1 \cdot z$$

The chain rule!



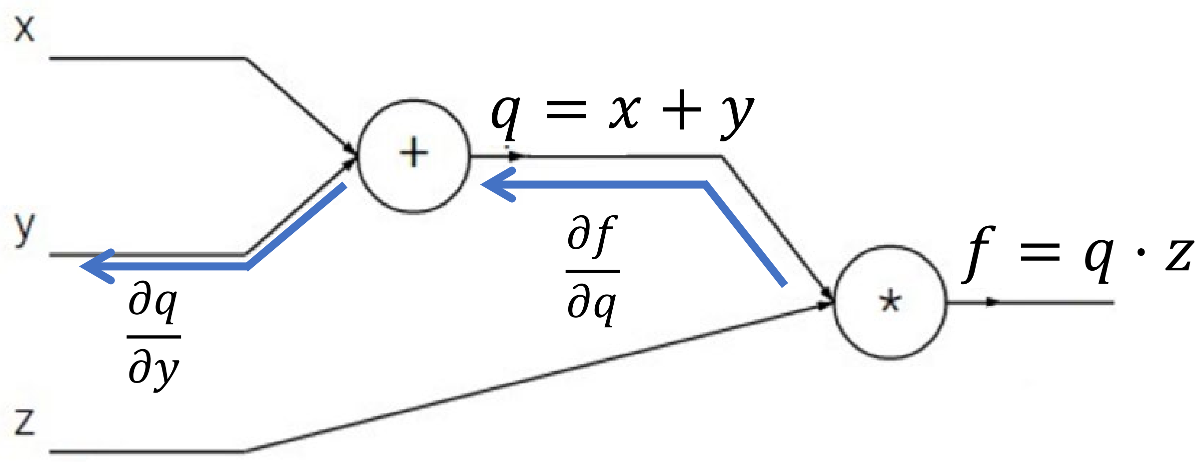
Backpropagation

Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

Step 2: Compute gradients.

$$f(x, y, z) = (x + y) \cdot z$$

$$\frac{\partial f}{\partial y} = \frac{\partial q}{\partial y}\frac{\partial f}{\partial q} = 1 \cdot z$$



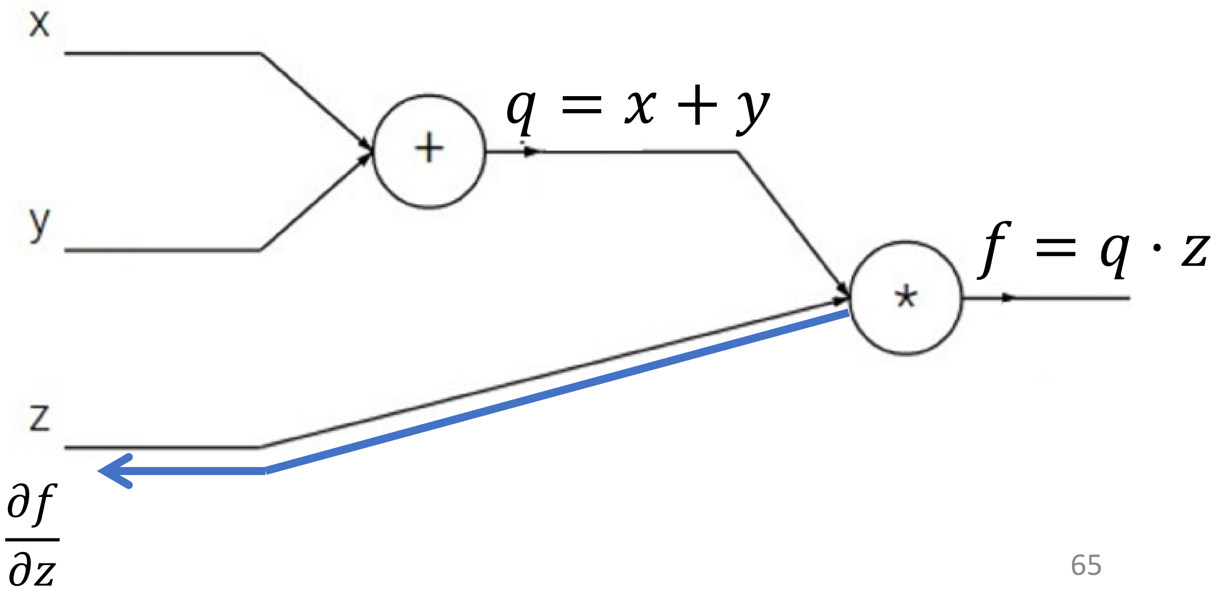
Backpropagation

Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

Step 2: Compute gradients.

$$f(x, y, z) = (x + y) \cdot z$$

$$\frac{\partial f}{\partial z} = q$$

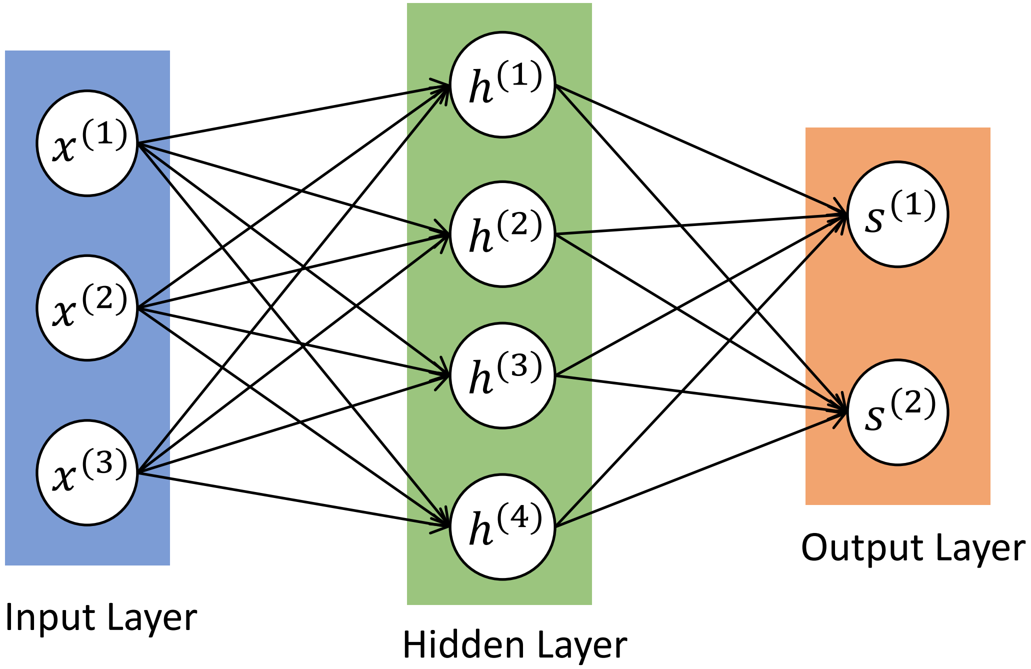


Backpropagation

Backpropagation is an algorithm to compute gradients of arbitrarily large networks.

Instead of computing the derivative of the loss for a huge network, compute the gradient of each layer with respect to the input.

Then we can *propagate* the gradients *backwards* through the graph.



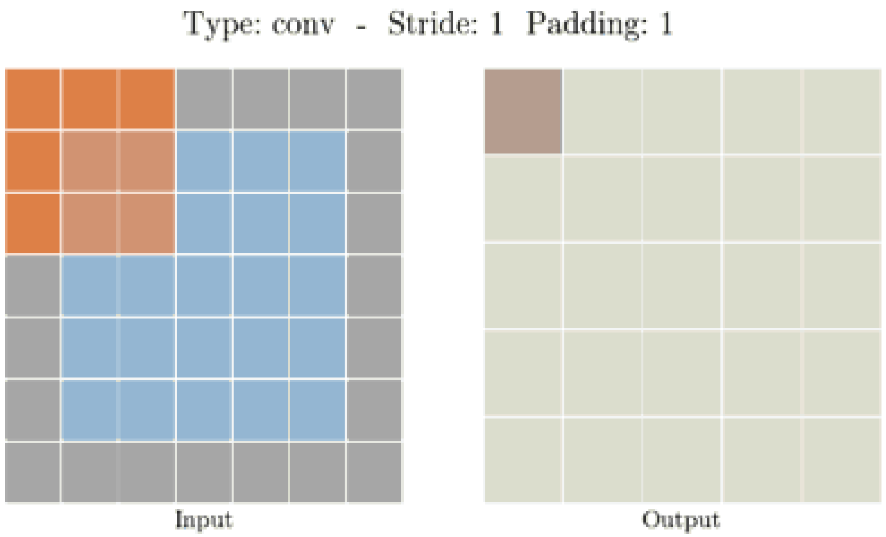
That's why it's called the *backward pass*!

Where to go from here: Convolutional Neural Networks

Many modern deep learning methods for computer vision use **Convolutional Neural Networks** (CNNs).

Instead of flattening the image into a vector and having a giant weight matrix, we slide a *kernel* across the image. This is a **convolution**.

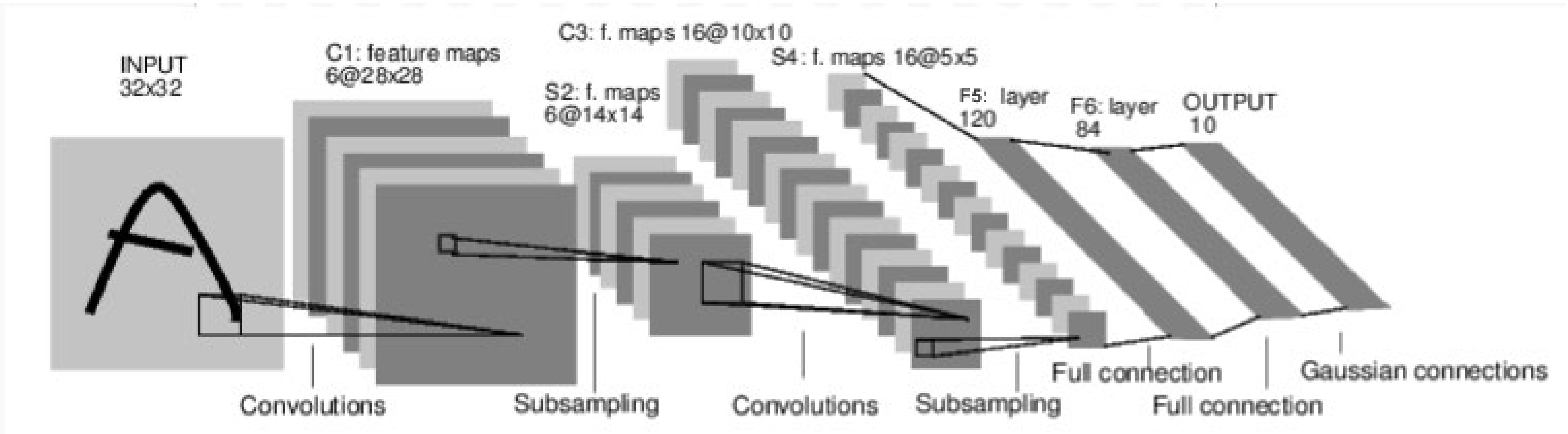
This model assumes that parts of the image close to each other likely have similar features.



(link)

Where to go from here: Convolutional Neural Networks

Many modern deep learning methods for computer vision use **Convolutional Neural Networks** (CNNs).



([link](#))

Often use linear layers at the end!

Further Reading:

- CS231n on Neural Networks
  - https://cs231n.github.io/neural-networks-1/
- Prof. Patrick Winston explains Neural Nets (YouTube)
- PyTorch is a great library (in Python) for implementing neural networks