Deep Learning II: Backpropagation

Swati Mishra

Applications of Machine Learning (4AL3)

Fall 2024

McMaster University

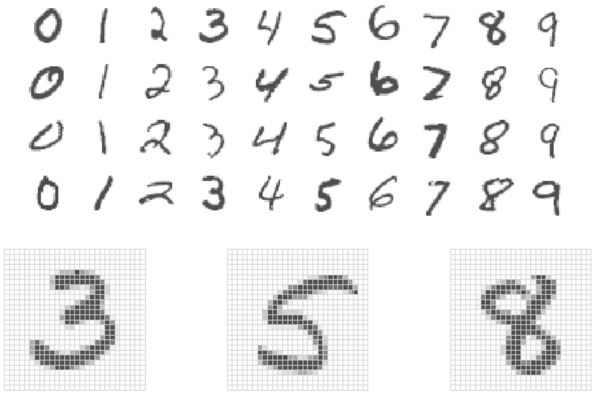
ENGINEERING

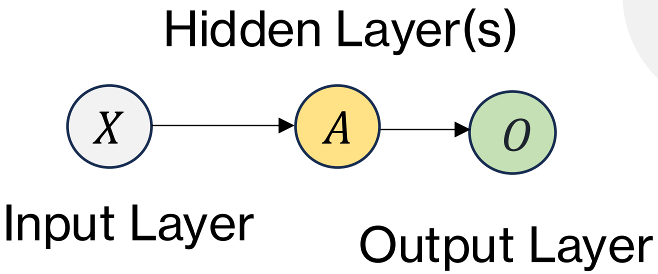
Review

- Neural Network fundamentals
- Architecture: Input, Output, Hidden Units
- Activation functions
- Design considerations for neural architecture

Neural Networks: Architecture

- To design architecture
  - We need a input layer.
    - Size: 784
  - We need output layers.
    - Size: 10 (0-9)
  - We need hidden layers.
    - Size: L1 = 512 , L2 = 512
    - Activation Function: ReLu





Neural Networks: Architecture

- To design architecture
  - We need a input layer.
    - Size: 784
  - We need output layers.
    - Size: 10 (0-9)
  - We need hidden layers.
    - Size: L1 = 512 , L2 = 512
    - Activation Function: ReLu

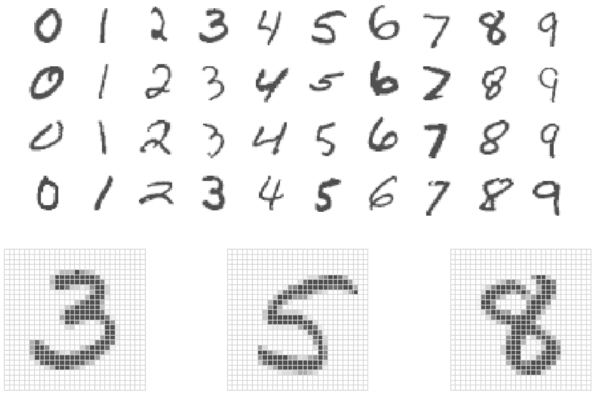
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
```

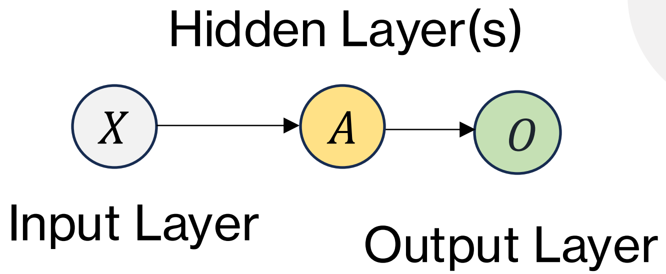
[33] ✓ 0.0s

```
model = NeuralNetwork()
print(model)
```

[34] ✓ 0.0s

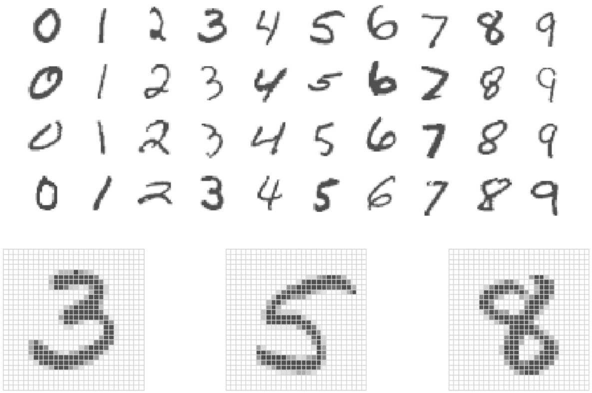
```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

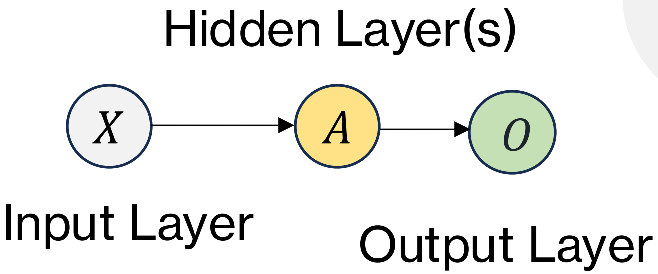




Neural Networks: Training

- To train the model
  - We need a learning algorithm
    - Type:??
  - We need a cost function
    - Type:??
  - We need a training objective
    - Type: ??





Forward Propagation

- Forward propagation algorithm in neural network

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Source: Deep Learning Book

Forward Propagation

- Forward propagation algorithm in neural network

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

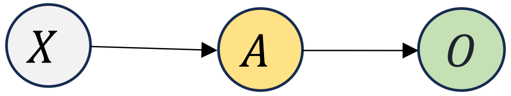
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

```python
import numpy as np
w = np.array([[0.9,0.3,0.4],
              [0.2,0.8,0.2],
              [0.1,0.5,0.6]])
x = np.array([[0.9],[0.1],[0.8]])
w.dot(x)
```
✓ 0.0s

```python
def sigmoid(z):
    return 1/(1 + np.exp(-z))
```
[6] ✓ 0.0s

$$X = \begin{vmatrix} 0.9 \\ 0.1 \\ 0.8 \end{vmatrix} \qquad O = \begin{vmatrix} 0.72 \\ 0.70 \\ 0.77 \end{vmatrix}$$

Input units   Hidden units   Output units



$$A = \begin{vmatrix} 0.76 \\ 0.60 \\ 0.65 \end{vmatrix}$$

$$X_{hidden} = W_{XA} . X \qquad X_{output} = W_{AO} . X_{hidden}$$

$$W_{XA} = \begin{vmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{vmatrix}$$

$$W_{AO} = \begin{vmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{vmatrix}$$

Forward Propagation

- Forward propagation algorithm in neural network

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, \ldots, l$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k-}$
  $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
end for
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$

```python
import numpy as np
w = np.array([[0.9,0.3,0.4],
              [0.2,0.8,0.2],
              [0.1,0.5,0.6]])
x = np.array([[0.9],[0.1],[0.8]])
w.dot(x)
```
✓ 0.0s

```python
def sigmoid(z):
    return 1/(1 + np.exp(-z))
```
[6] ✓ 0.0s

$$X = \begin{bmatrix} 0.9 \\ 0.1 \\ 0.8 \end{bmatrix} \qquad O = \begin{bmatrix} 0.72 \\ 0.70 \\ 0.77 \end{bmatrix}$$

Input units Hidden units Output units

$$A = \begin{bmatrix} 0.76 \\ 0.60 \\ 0.65 \end{bmatrix}$$

$$X_{hidden} = W_{XA}.X \qquad X_{output} = W_{AO}.X_{hidden}$$

$$W_{XA} = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.2 & 0.8 & 0.2 \\ 0.1 & 0.5 & 0.6 \end{bmatrix}$$

$$W_{AO} = \begin{bmatrix} 0.3 & 0.7 & 0.5 \\ 0.6 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.9 \end{bmatrix}$$

Neural Network as Graph

- To understand Backpropagation, let us consider the neural network to be a computational graph.

- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, vector, tensor, or something else.

Neural Network as Graph

- To understand Backpropagation, let us consider the neural network to be a computational graph.
- Each node (or unit or neuron) refers to a variable.
  - Variables can be **scalar**, matrix, vector, tensor, or something else.

A scalar is just a single number

Neural Network as Graph

- To understand Backpropagation, let us consider the neural network to be a computational graph.

- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, **matrix**, vector, tensor, or something else.

A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one.

McMaster University

Neural Network as Graph

- To understand Backpropagation, let us consider the neural network to be a computational graph.
- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, **vector**, tensor, or something else.

A vector is an array of numbers arranged in order

Neural Network as Graph

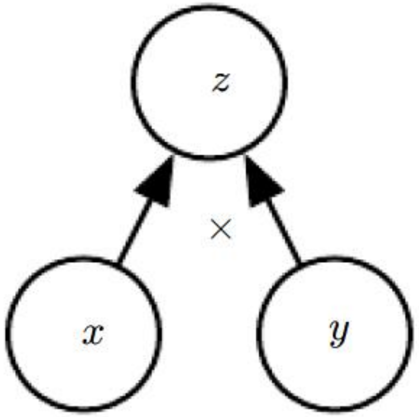
- To understand Backpropagation, let us consider the neural network to be a computational graph.
- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, vector, **tensor**, or something else.

A tensor is an an array with more than two axes. It is essentially an array of numbers arranged on a regular grid with a variable number of axes.

Neural Network as Graph

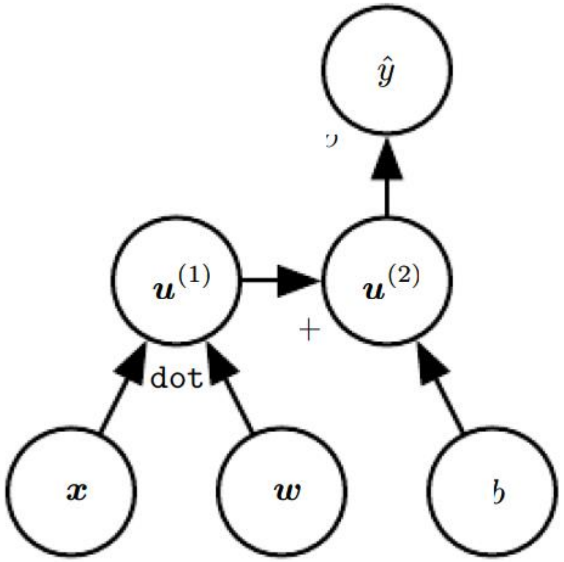
- To understand Backpropagation, let us consider the neural network to be a computational graph.
- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, vector, tensor, or something else.
- An operation is a function of one or more variables
  - Operation returns a single output variable
- If an operation applied to x gives y, then x and y have an edge.

Neural Network as Computational Graph



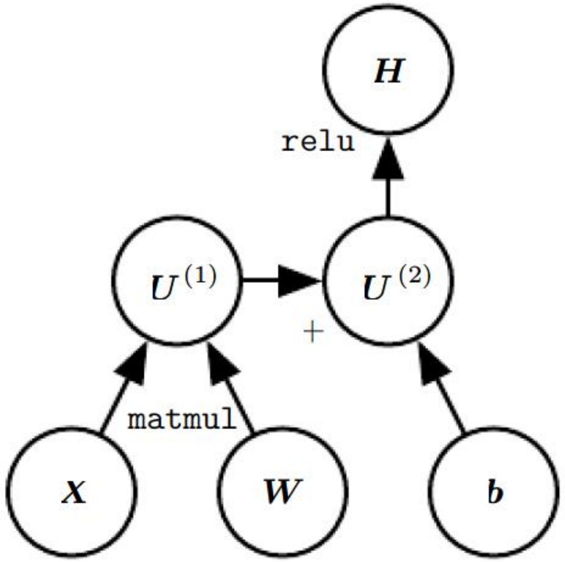
(a)

$$z = xy$$



(b)

$$y' = \sigma(x^T w + b)$$



(c)

$$\mathrm{H} = \max\{0, xw + b\}$$

Chain Rule

- To understand Backpropagation, let us consider the neural network to be a computational graph.
- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, vector, tensor, or something else.
- An operation is a function of one or more variables
  - Operation returns a single output variable
- If an operation applied to x gives y, then x and y have an edge.
- Chain rule in calculus: if $y = g(x)$ and $\mathrm{z} = f\big(g(x)\big) = f(y)$ then, $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

Chain Rule

- Chain rule in calculus: if $y = g(x)$ and $\mathrm{z} = f\big(g(x)\big) = f(y)$ then, $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

- Each node (or unit or neuron) refers to a variable.
  - Variables can be scalar, matrix, vector, tensor, or something else

- If we want to calculate the gradient of variable x , we can rewrite the above in a vector notation,

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}}\, z,$$

This is Jacobian gradient product

- Backpropagation algorithm consists of performing this product for each operation in the graph.

Chain Rule

- Chain rule in calculus: if $y = g(x)$ and $z = f(g(x)) = f(y)$ then, $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

This is Jacobian gradient product

- Consider the computational graph below:

$$w \xrightarrow{f} x \xrightarrow{f} y \xrightarrow{f} z$$

$x = f(w)$

$y = f(x)$

$z = f(y)$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \quad = f'(y) f'(x)\, f'(w) = f'\left(f\left(f(w)\right)\right) f'(f(w)))\, f'(w)$$

Chain Rule

- Chain rule in calculus: if $y = g(x)$ and $z = f(g(x)) = f(y)$ then, $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

This is Jacobian gradient product

- Consider the computational graph below:

$$w \xrightarrow{f} x \xrightarrow{f} y \xrightarrow{f} z$$

$$x = f(w)$$

$$y = f(x)$$

$$z = f(y)$$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \quad = f'(y)f'(x)\, f'(w) = f'\big(f(f(w))\big)\, f'(f(w)))\, f'(w)$$

McMaster University

Chain Rule

- Chain rule in calculus: if $y = g(x)$ and $z = f\big(g(x)\big) = f(y)$ then, $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

This is Jacobian gradient product

- Consider the computational graph below:

$$w \xrightarrow{f} x \xrightarrow{f} y \xrightarrow{f} z$$

$x = f(w)$

$y = f(x)$

$z = f(y)$

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \quad = f'(y)f'(x)\, f'(w) = f'\Big(f\big(f(w)\big)\Big)\, f'(f(w)))\, f'(w)$$

Backpropagation

The back-propagation algorithm:

- To compute the gradient of some scalar z with respect to one of its ancestors x in the graph
    - We compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z.
    - We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach x.
    - For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

Backpropagation

The back-propagation algorithm:

- To compute the gradient of some scalar z with respect to one of its ancestors x in the graph
    - We compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z.
    - We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach x.
    - For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

Backpropagation

The back-propagation algorithm:

- To compute the gradient of some scalar z with respect to one of its ancestors x in the graph
  - We compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z.
  - We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach x.
  - For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

Training with Backpropagation

- Backpropagation Algorithm

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated

```
Let G' be G pruned to contain only nodes that are ancestors of z and descendents
of nodes in T.
Initialize grad_table, a data structure associating tensors to their gradients
grad_table[z] ← 1
for V in T do
  build_grad(V, G, G', grad_table)
end for
Return grad_table restricted to T
```

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and grad_table.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: grad_table, a data structure mapping nodes to their gradients

```
if V is in grad_table then
  Return grad_table[V]
end if
i ← 1
for C in get_consumers(V, G') do
  op ← get_operation(C)
  D ← build_grad(C, G, G', grad_table)
  G^(i) ← op.bprop(get_inputs(C, G'), V, D)
  i ← i + 1
end for
G ← Σ_i G^(i)
grad_table[V] = G
Insert G and the operations creating it into G
Return G
```

Source: Deep Learning Book

Training with Backpropagation

- Backpropagation Algorithm

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
  Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
  Initialize `grad_table`, a data structure associating tensors to their gradients
  `grad_table`$[z] \leftarrow 1$
  for $\mathbf{V}$ in $\mathbb{T}$ **do**
    `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}',$ `grad_table`$)$
  end for
  Return `grad_table` restricted to $\mathbb{T}$

Setting table to store and retrieve gradients

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: `grad_table`, a data structure mapping nodes to their gradients
  if $\mathbf{V}$ is in `grad_table` **then**
    Return `grad_table`$[\mathbf{V}]$
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in `get_consumers`$(\mathbf{V}, \mathcal{G}')$ **do**
    op $\leftarrow$ `get_operation`$(\mathbf{C})$
    $\mathbf{D} \leftarrow$ `build_grad`$(\mathbf{C}, \mathcal{G}, \mathcal{G}',$ `grad_table`$)$
    $\mathbf{G}^{(i)} \leftarrow$ op.bprop(`get_inputs`$(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D})$
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  `grad_table`$[\mathbf{V}] = \mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$

returns the children of V

returns the operation that computes V

Tensor based representation of Jacobian –based gradient product

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\top} \nabla_{\boldsymbol{y}} z,$$

get_inputs returns the parents of V

Source: Deep Learning Book

Training with Backpropagation

- Backward computation of the graph

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

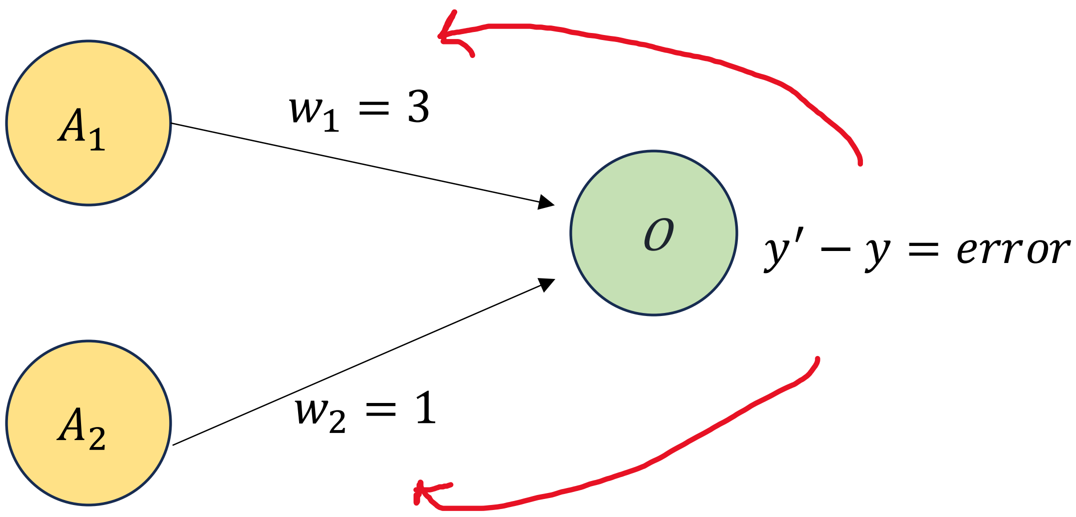
Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top} \boldsymbol{g}$

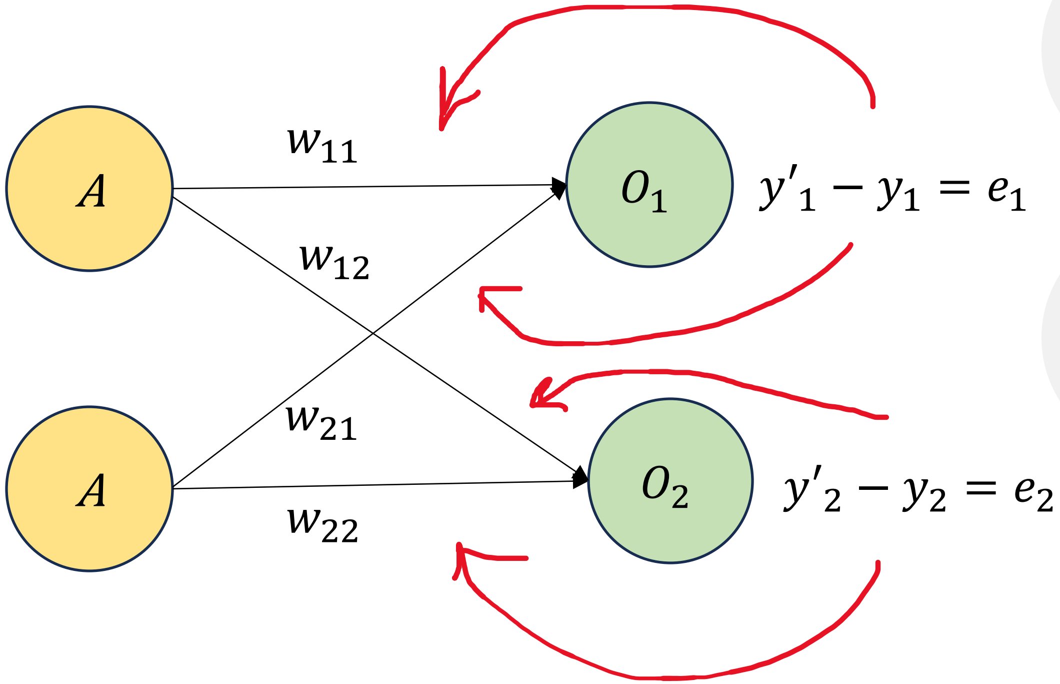
end for

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input.

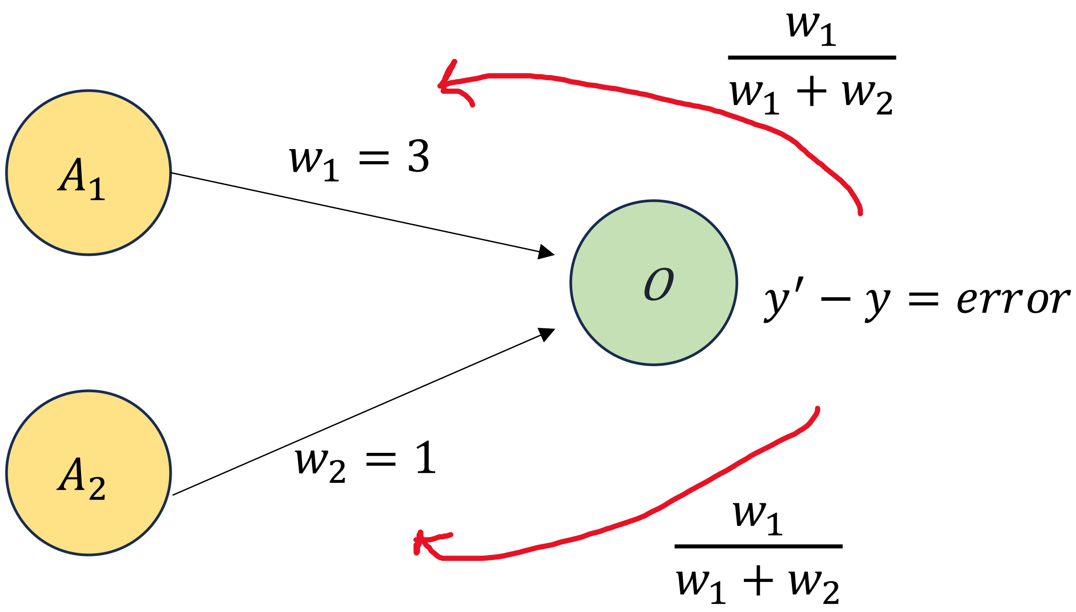


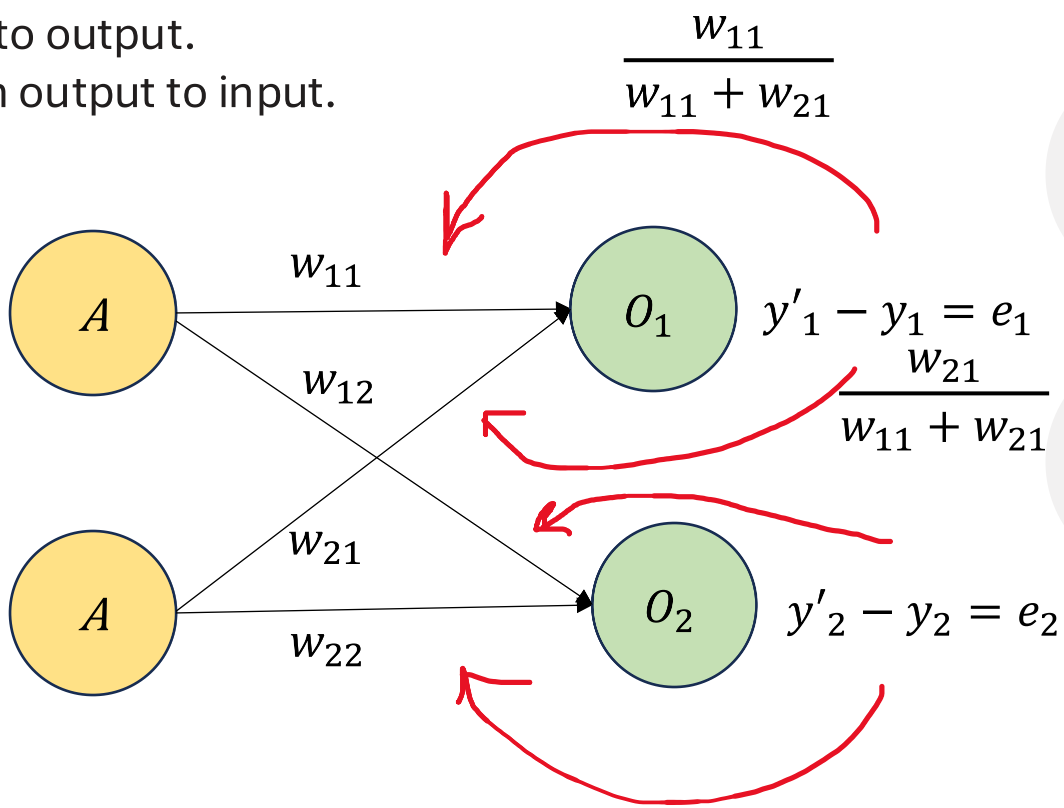
Error not distributed equally



Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input.

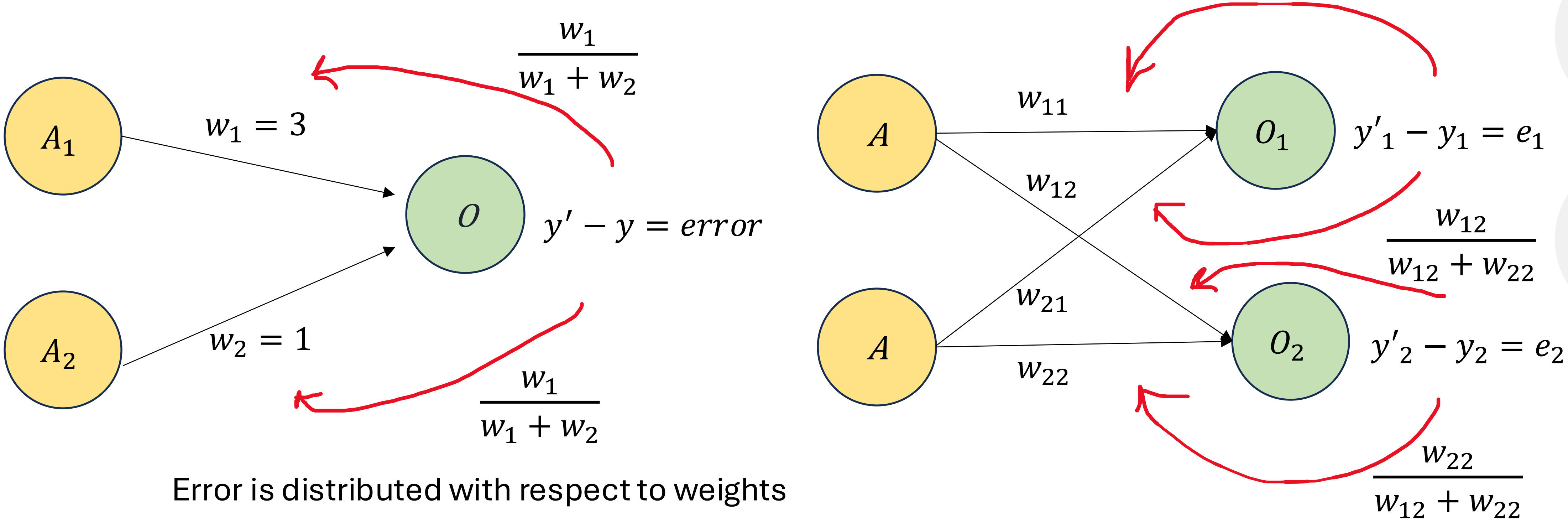




Error is distributed with respect to weights

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input.



Error is distributed with respect to weights

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input

- What do we do with non-zero error?
  - We update the weights

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input

- What do we do with non-zero error?
  - We update the weights
- How do we update the weights if error is non-zero?
  - Gradient Descent

- For any node w_{jk}, the new weights associated with it are

$$w'_{jk} = w_{jk} - \alpha \frac{\partial e}{\partial w_{jk}}$$

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input

- What do we do with non-zero error?
  - We update the weights
- How do we update the weights if error is non-zero?
  - Gradient Descent

- For any node w_{jk}, the new weights associated with it are

$$w'_{jk} = w_{jk} - \alpha \frac{\partial e}{\partial w_{jk}}$$

But gradient of what?

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input
- What do we do with non-zero error?
  - We update the weights
- How do we update the weights if error is non-zero?
  - Gradient Descent

- For any node w_{jk}, the new weights associated with it are

$$w'_{jk} = w_{jk} - \alpha \frac{\partial e}{\partial w_{jk}}$$

But gradient of what?

- Backpropagation computes the gradient of units with respect to variables.

Backpropagation

- Forward propagation takes the signal forward from input to output.
- Backpropagation takes the signal (in this case **error**) from output to input
- What do we do with non-zero error?
  - We update the weights
- How do we update the weights if error is non-zero?
  - Gradient Descent

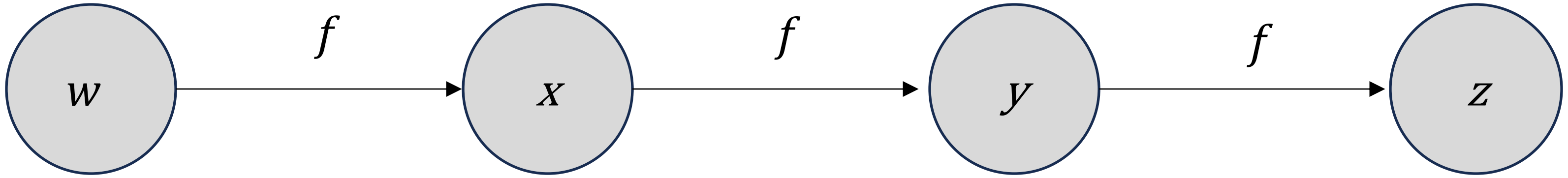
- For any node w_{jk}, the new weights associated with it are

$$w'_{jk} = w_{jk} - \alpha \frac{\partial e}{\partial w_{jk}}$$

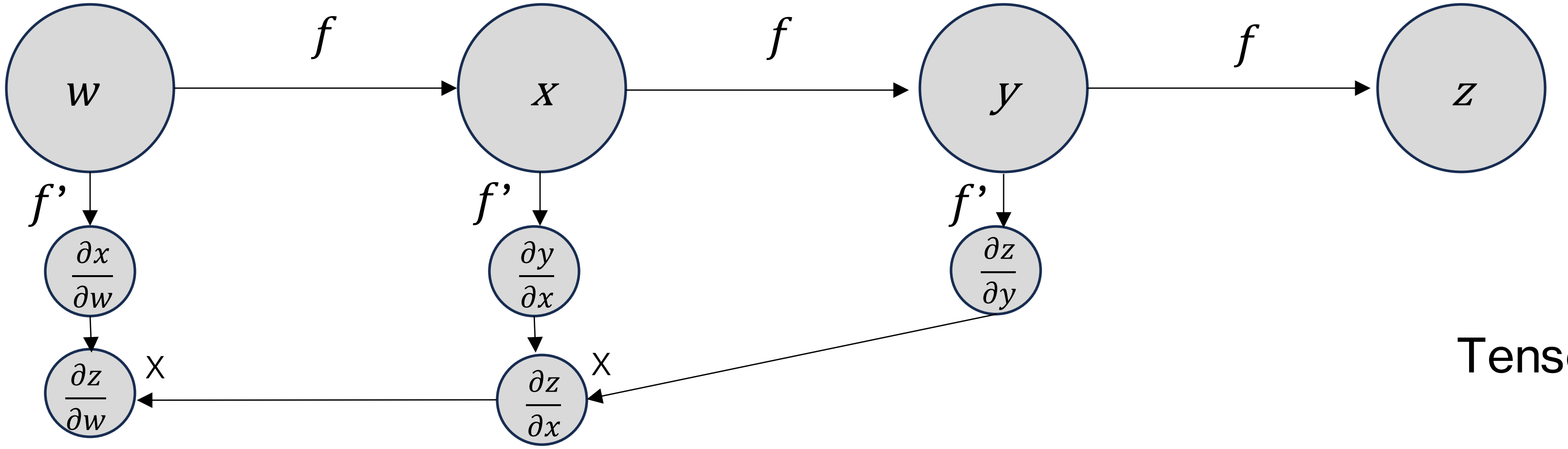
But gradient of what?

- Backpropagation computes the gradient of units with respect to variables.
- The backpropagation starts at the end and recursively applies the chain rule to compute the gradients all the way to the inputs of the network. The gradients can be thought of as flowing backwards through the networks.

Backpropagation



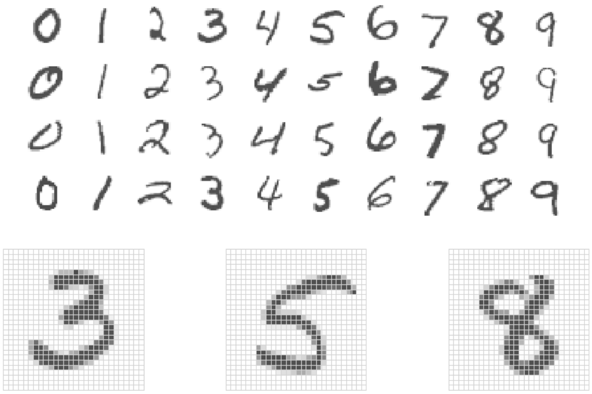
$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} = f'(y)f'(x)\, f'(w) = f'\left(f\left(f(w)\right)\right) f'(f(w)))\, f'(w)$$

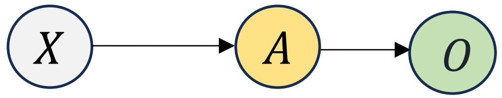


Tensorflow uses this!

Neural Networks: Training

- To train the model
  - We need a learning algorithm
    - Backpropagation
    - Gradient Descent
  - We need a cost function
    - ??
  - We need a training objective
    - ??

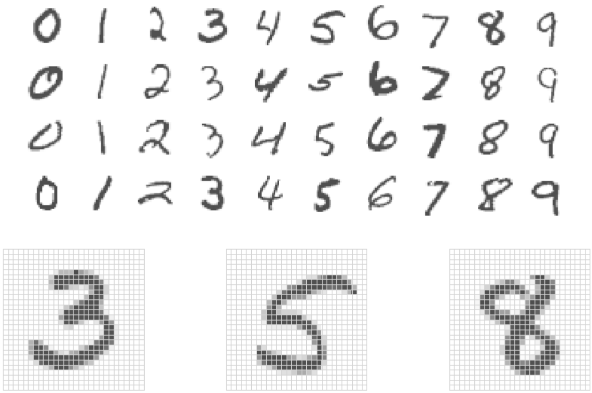


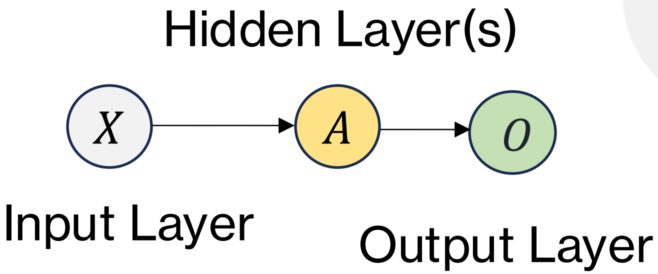


Neural Networks: Training

- To train the model
  - We need a learning algorithm
    - Backpropagation
    - Gradient Descent
  - We need a cost function
    - Cross-entropy loss
    - Mean Absolute Error
    - Mean Squared Error
  - We need a training objective
    - ??

$$\sum_{i=1}^{n} (y_i - f(x_i))^2 .$$





McMaster University

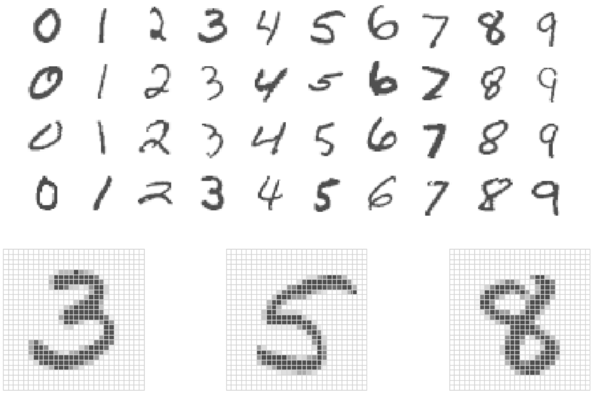
Neural Networks: Training

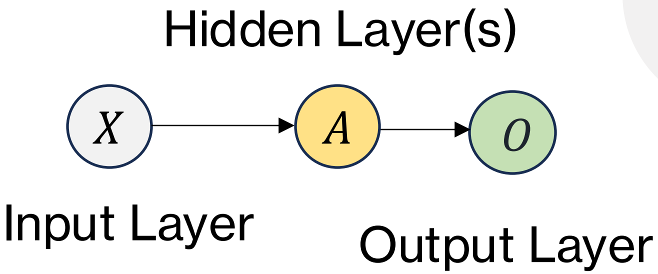
- To train the model
  - We need a learning algorithm
    - Backpropagation
    - Gradient Descent

  - We need a cost function
    - Cross-entropy loss
    - Mean Absolute Error
    - Mean Squared Error

$$\sum_{i=1}^{n} (y_i - f(x_i))^2 .$$

  - We need a training objective
    - minimize the negative multinomial log-likelihood
    - minimize mean (squared or absolute) error





Readings

Required Readings:

Introduction to Statistical Learning

- Chapter 10 – Section 10.7 page 427 - 429

***Supplemental Readings*:**

Deep Learning

- Chapter 6 – page 168 - 224

Thank You

McMaster University