APS360: Applied Fundamentals of Deep Learning

Week 2: Artificial Neural Networks - Part II

UNIVERSITY OF TORONTO

Training Neural Networks

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

We can use our prediction error to decide how to change weights.

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

1. Make a prediction for some input data $\mathbf{x}$, with a known correct output t

$$y = M(\mathbf{x}; \mathbf{w})$$

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

1. Make a prediction for some input data $\mathbf{x}$, with a known correct output t

$$y = M(\mathbf{x}; \mathbf{w})$$

2. Compare the correct output with our predicted output to compute loss

$$\mathcal{L}(y, t)$$

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

1. Make a prediction for some input data $\mathbf{x}$, with a known correct output t

$$y = M(\mathbf{x}; \mathbf{w})$$

2. Compare the correct output with our predicted output to compute loss

$$\mathcal{L}(y, t)$$

3. Adjust the weights/bias terms to make the prediction closer to the ground truth, i.e., minimize loss $\mathcal{L}$

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

1. Make a prediction for some input data $\mathbf{x}$, with a known correct output t

$$y = M(\mathbf{x}; \mathbf{w})$$

2. Compare the correct output with our predicted output to compute loss

$$\mathcal{L}(y, t)$$

3. Adjust the weights/bias terms to make the prediction closer to the ground truth, i.e., minimize loss $\mathcal{L}$

4. Repeat until we have an acceptable loss $\mathcal{L}$ / error E

UNIVERSITY OF TORONTO

Training a Neuron

How do we **learn** the **weights** (and bias) of a neural network?

input $\mathbf{x} \in \mathbb{R}^d$, predicted output $y \in \mathbb{R}$, ground truth $t \in \mathbb{R}$, Neuron $M(\mathbf{x}; \mathbf{w})$

1. Make a prediction for some input data $\mathbf{x}$, with a known correct output t

$$y = M(\mathbf{x}; \mathbf{w})$$

2. Compare the correct output with our predicted output to compute loss

$$\mathcal{L}(y, t)$$

3. Adjust the weights/bias terms to make the prediction closer to the ground truth, i.e., minimize loss $\mathcal{L}$

4. Repeat until we have an acceptable loss $\mathcal{L}$ / error E

Used for both training and inference

Forward pass

Backward pass

Used only for training

UNIVERSITY OF TORONTO

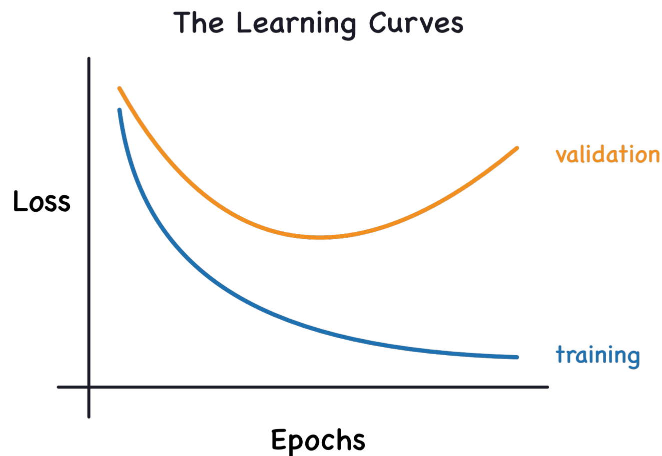
Loss Function

Loss function

A **loss function** computes how bad predictions are compared to the ground truth labels.

- **Large loss:** the network's prediction differs from the ground truth
- **Small loss:** the network's prediction matches the ground truth

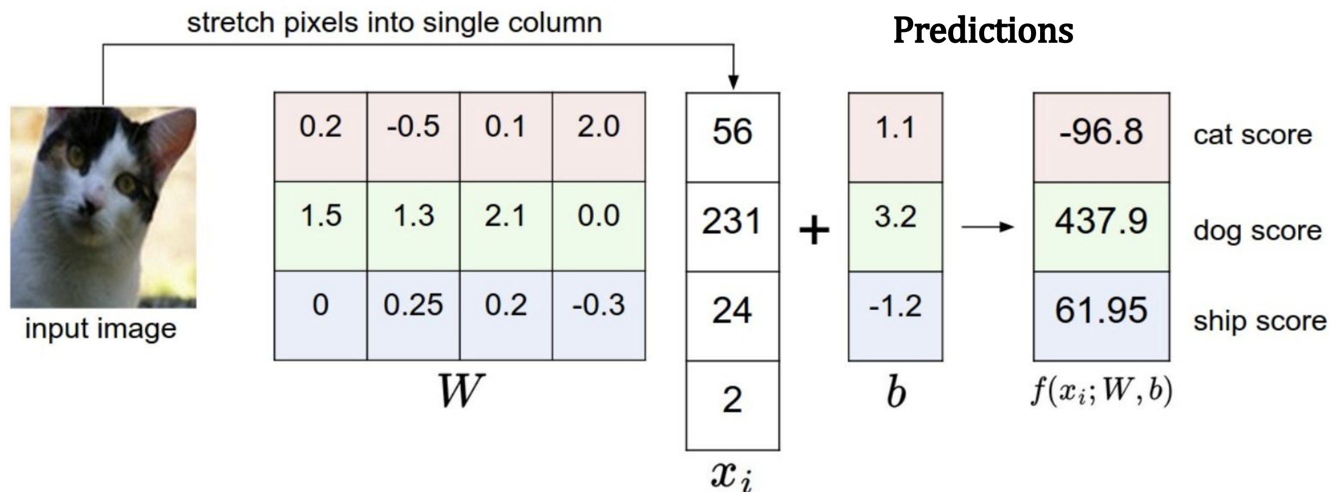
We want to calculate the error over **all training samples** (average error)



UNIVERSITY OF TORONTO

Loss function

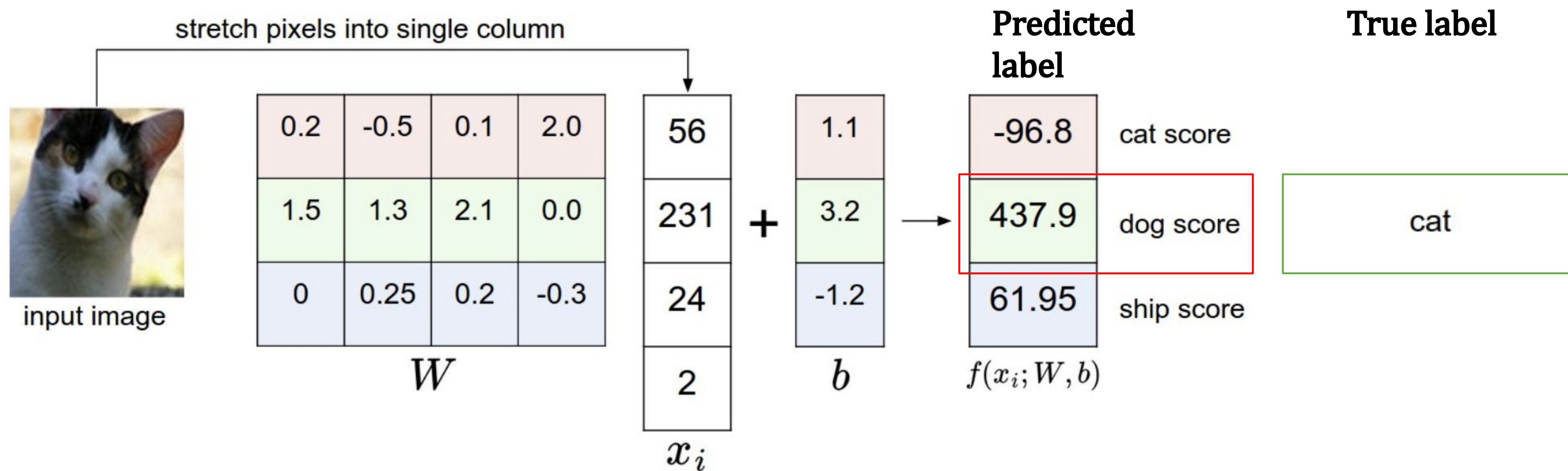
Suppose we want to train a linear neuron to differentiate images into three classes:



https://cs231n.github.io/linear-classify/

Loss function

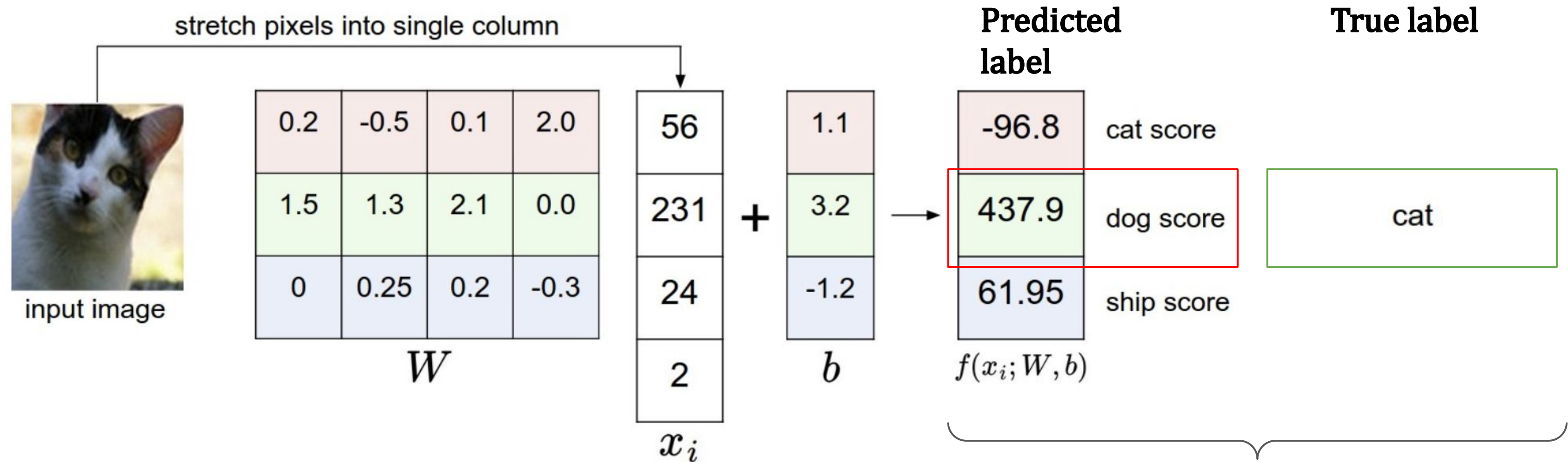
Suppose we want to train a linear neuron to differentiate images into three classes:



https://cs231n.github.io/linear-classify/

Loss function

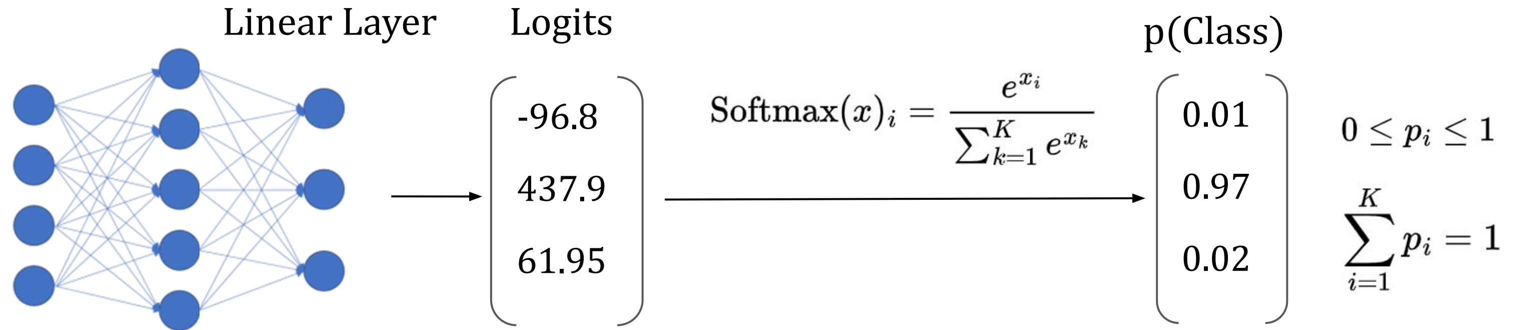
Suppose we want to train a linear neuron to differentiate images into three classes:



We cannot compare 437.9 to "cat" → We need Canonical representation

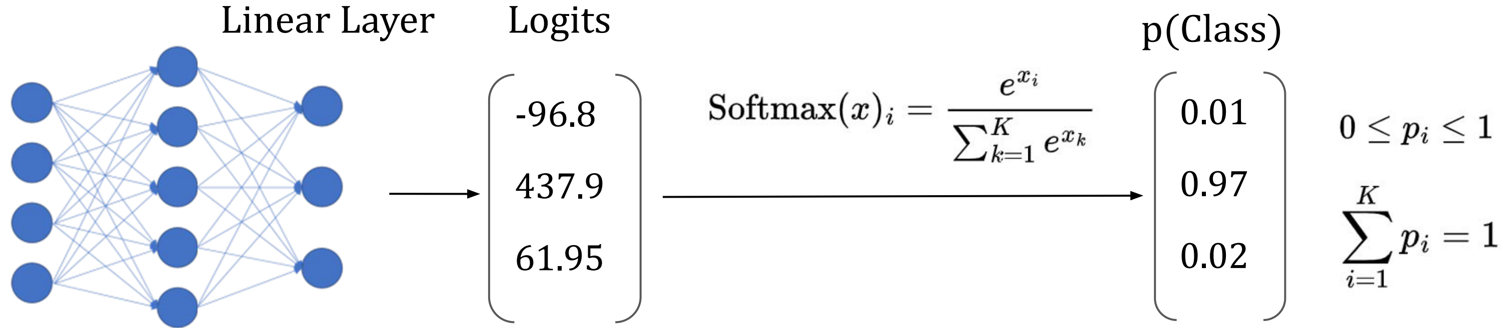
Loss function

Softmax function → normalizes the **logits** into a categorical probability distribution over all possible classes.

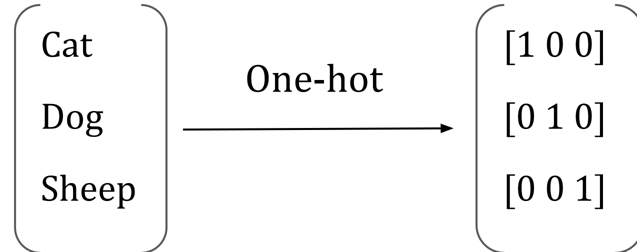


Loss function

Softmax function → normalizes the **logits** into a categorical probability distribution over all possible classes.



One-hot encoding → Maps categories to vector representation.



UNIVERSITY OF TORONTO

Loss function

Mean Squared Error (MSE) → Mostly used for regression problems.

$$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(y_n - t_n)^2$$

Number of training samples

Prediction

True label

Predicted p(Class)	Ground truth
0.01	1.0
0.97	0.0
0.02	0.0

$\text{MSE} = (0.01 - 1.0)^2 = 0.98$

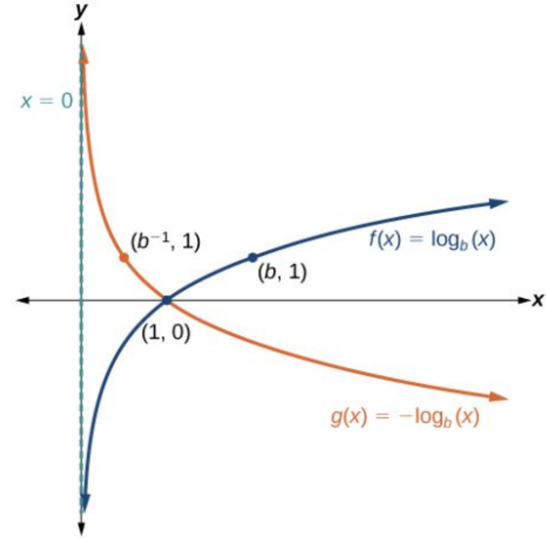
Loss function

Cross Entropy (CE) → Mostly used for classification problems.

training samples, #classes, True label, Prediction

$$\mathrm{CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_{n,k})$$

Predicted p(Class)	Ground truth
0.01	1.0
0.97	0.0
0.02	0.0



$$\mathrm{CE} = -[1.0 \times \log_2(0.01) + 0.0 \times \log_2(0.97) + 0.0 \times \log_2(0.02)] = 6.64$$

Loss function

Cross Entropy (CE)

$$\mathrm{CE} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k}\log(y_{n,k})$$

Binary Cross Entropy (BCE)

$$\mathrm{BCE} = -\frac{1}{N}\sum_{n=1}^{N}\left[t_n\log(y_n) + (1-t_n)\log(1-y_n)\right]$$

Forward-Pass with Error Calculations

```
import math

x = [[1.0, 0.1,-0.2],    # data
     [1.0,-0.1, 0.9],
     [1.0, 1.2, 0.1],
     [1.0, 1.1, 1.5]]
t = [0, 0, 0, 1]         # labels
w = [1, -1, 1]           # initial weights

def simple_ANN(x, w, t):
    total_e, e, y = 0, [], []
    for n in range(len(x)):
        v = 0
        for d in range(len(x[0])):
            v += x[n][d] * w[d]
        y.append(1/1+math.e**(-v))      # sigmoid
        e.append((y[n]-t[n])**2)        # MSE
    total_e = sum(e)/len(x)
    return (y, w, total_e)
```

Forward-Pass with Error Calculations

```
import math

x = [[1.0, 0.1,-0.2],    # data
     [1.0,-0.1, 0.9],
     [1.0, 1.2, 0.1],
     [1.0, 1.1, 1.5]]
t = [0, 0, 0, 1]          # labels
w = [1, -1, 1]            # initial weights

def simple_ANN(x, w, t):
    total_e, e, y = 0, [], []
    for n in range(len(x)):
        v = 0
        for d in range(len(x[0])):
            v += x[n][d] * w[d]
        y.append(1/1+math.e**(-v))      # sigmoid
        e.append(-t[n]*math.log(y[n])-(1-t[n])*math.log(1-y[n]))  # BCE
    total_e = sum(e)/len(x)
    return (y, w, total_e)
```

UNIVERSITY OF TORONTO

Gradient Descent

UNIVERSITY OF
TORONTO

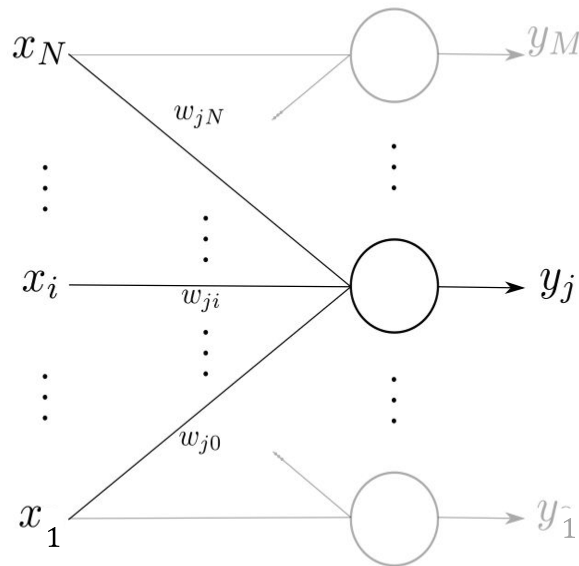
Neural Network Single Layer Training: Delta Rule

We want to know **how to change each of our neuron's weights** w_{ji} to reduce this error E

First we need to know how our error changes with each weight…

$$\frac{\partial E}{\partial w_{ji}}$$

This is relatively simple to calculate *adjacent to the output layer*



UNIVERSITY OF TORONTO

Neural Network Single Layer Training

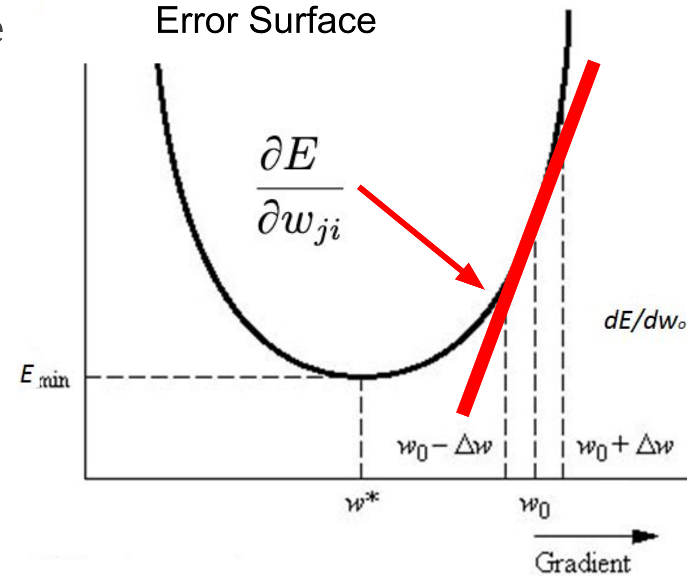
Vector of partial derivatives for all weights is the **gradient**

- Direction of the gradient is the direction in which the function increases most quickly
- Magnitude of the gradient is the rate of increase

Adjusting weights according to the slope (gradient) will guide us the minimum (or maximum) error

$$w_{ji}^{t+1} = w_{ji}^{t} - \gamma \frac{\partial E}{\partial w_{ji}} \qquad \Delta w_{ij} = \gamma \frac{\partial E}{\partial w_{ji}}$$

learning rate (step size)

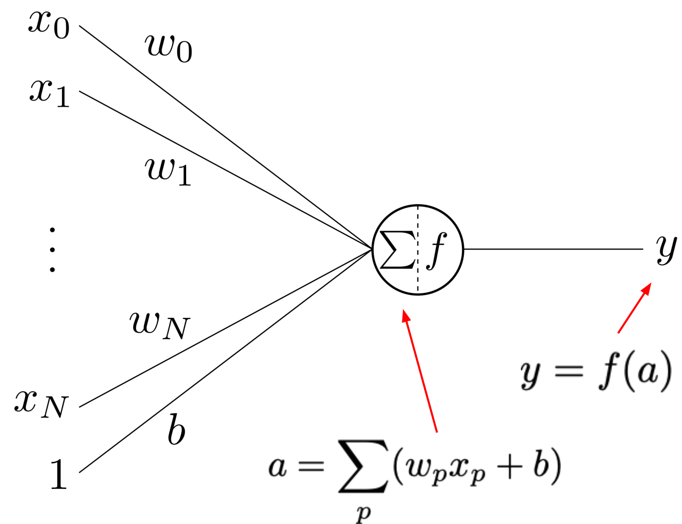


UNIVERSITY OF TORONTO

Delta Rule for Single Weight/Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{dE}{dw_p} = \text{???}$$



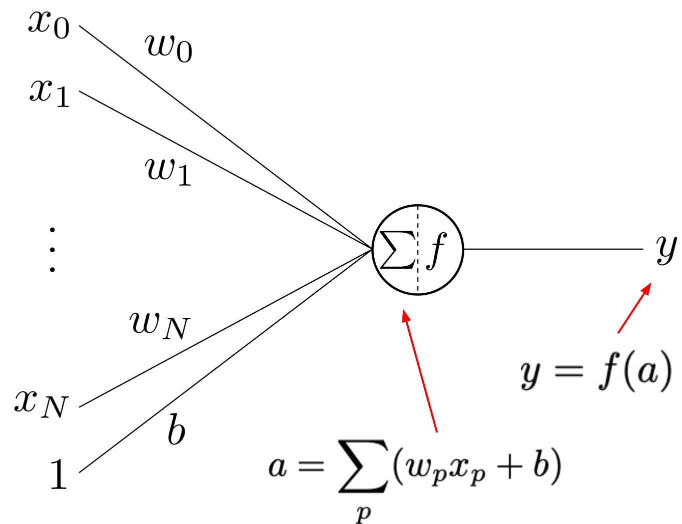
UNIVERSITY OF TORONTO

Delta Rule for Single Weight/Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$

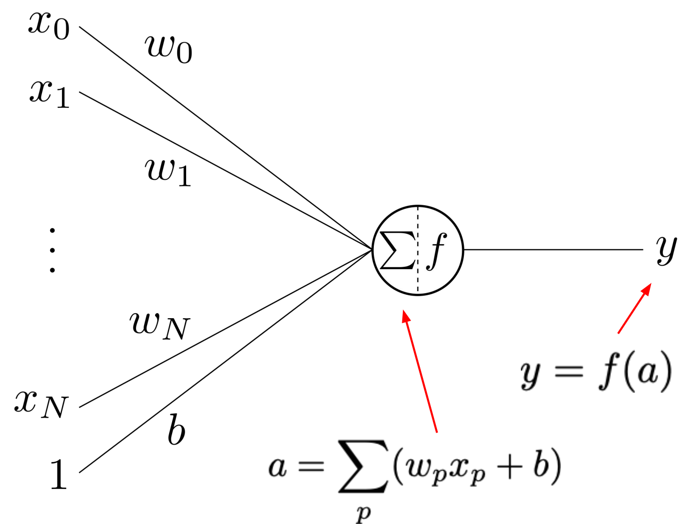
Chain rule!

$$\frac{dE}{dw_p} = \left(\frac{dE}{dy}\right)\left(\frac{dy}{da}\right)\left(\frac{da}{dw_p}\right)$$



Delta Rule for Single Weight/Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$



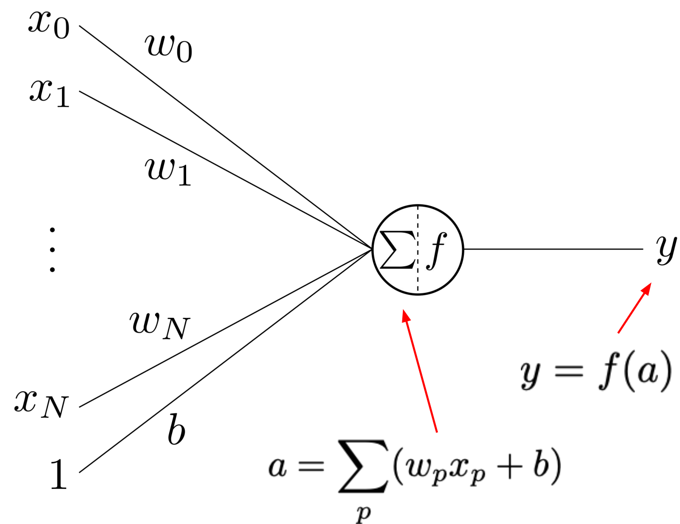
Chain rule!

$$\frac{dE}{dw_p} = \left(\frac{dE}{dy}\right)\left(\frac{dy}{da}\right)\left(\frac{da}{dw_p}\right)$$

$$\frac{dE}{dy} = \left(\frac{d(y-t)^2}{dy}\right) = 2(y - t)$$

Delta Rule for Single Weight/ Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$



Chain rule!

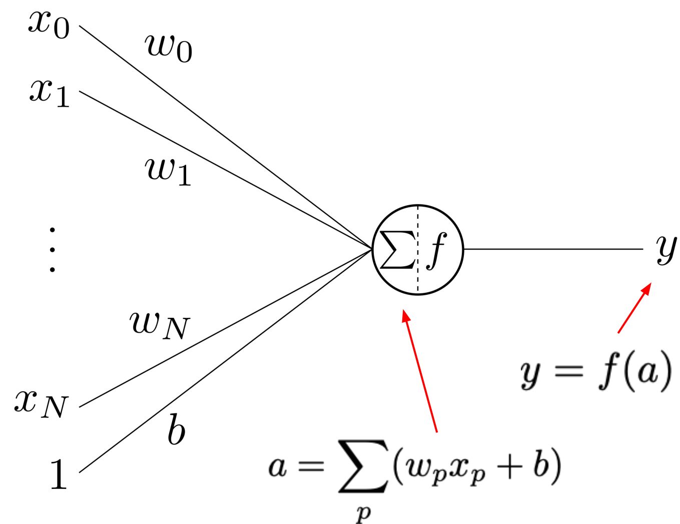
$$\frac{dE}{dw_p} = \left(\frac{dE}{dy}\right)\left(\frac{dy}{da}\right)\left(\frac{da}{dw_p}\right)$$

$$\frac{dE}{dy} = \left(\frac{d(y-t)^2}{dy}\right) = 2(y-t)$$

$$\frac{dy}{da} = \left(\frac{d\frac{1}{1+e^{-a}}}{da}\right) = (1-y)(y)$$

Delta Rule for Single Weight/Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$



Chain rule!

$$\frac{dE}{dw_p} = \left(\frac{dE}{dy}\right)\left(\frac{dy}{da}\right)\left(\frac{da}{dw_p}\right)$$

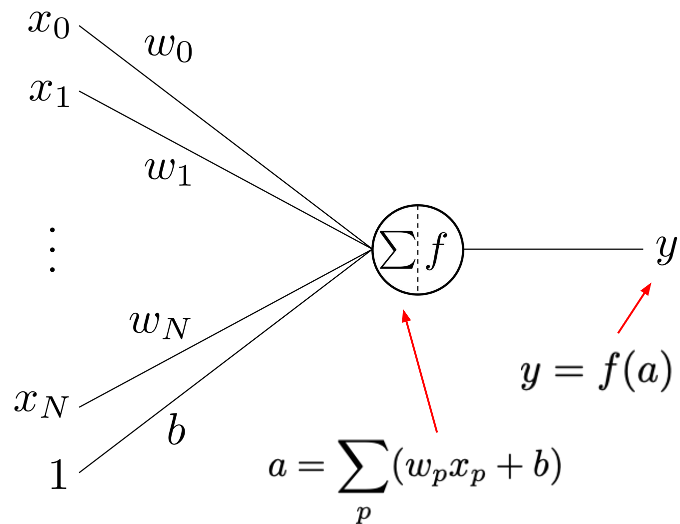
$$\frac{dE}{dy} = \left(\frac{d(y-t)^2}{dy}\right) = 2(y - t)$$

$$\frac{dy}{da} = \left(\frac{d\frac{1}{1+e^{-a}}}{da}\right) = (1 - y)(y)$$

$$\frac{da}{dw_p} = x_p$$

Delta Rule for Single Weight/Training Sample

$$E = (y - t)^2 \qquad f(x) = \frac{1}{1 + e^{-x}}$$



Chain rule!

$$\frac{dE}{dw_p} = \left(\frac{dE}{dy}\right)\left(\frac{dy}{da}\right)\left(\frac{da}{dw_p}\right)$$

$$\frac{dE}{dy} = \left(\frac{d(y-t)^2}{dy}\right) = 2(y - t)$$

$$\frac{dy}{da} = \left(\frac{d\frac{1}{1+e^{-a}}}{da}\right) = (1 - y)(y)$$

$$\frac{da}{dw_p} = x_p$$

$$\frac{dE}{dw_p} = 2(x_p)((y - t)((1 - y)(y)))$$

Forward-Pass & Backward-Pass

```
def simple_ANN(x, w, t, iter, lr):
    total_e = 0
    for i in range(iter):
        e, y = [], []
        for n in range(len(x)):
            v = 0
            for d in range(len(x[0])):
                v += x[n][d] * w[d]
            y.append(1/1+math.e**(-v))      # sigmoid
            e.append((y[n]-t[n])**2)        # MSE

            # gradient descent to update weights
            for p in range(len(w)):
                d = 2*x[n][p]*(y[n]-t[n])*(1-y[n])*y[n]
                W[p] -= lr*d
        total_e = sum(e)/len(x)
        return (y, w, e)
```

UNIVERSITY OF TORONTO

Neural Network Architectures

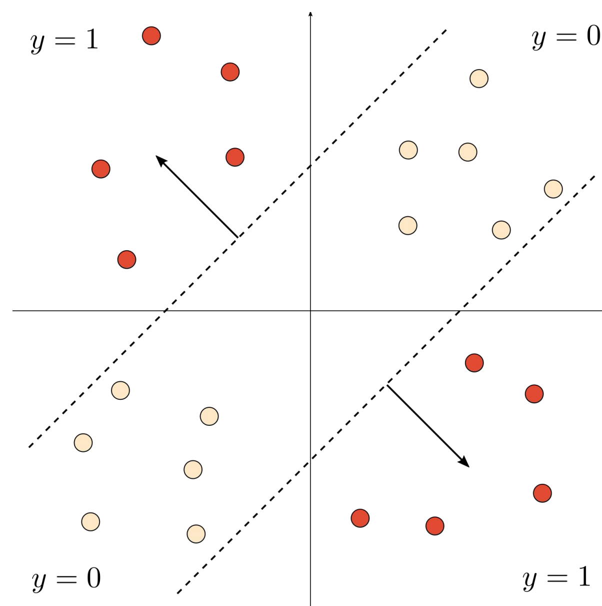
Multiple Layers are Important: XOR

Having a single decision boundary (a single NN layer) is not enough to solve many problems

The most famous such problem is the XOR function, which needs two decision boundaries to solve

We solve this by having **at least one hidden neural network layer** (i.e., two layers)

In fact in the limit of an infinitely-wide neural network with at least one hidden layer, NN is a **universal function approximator**



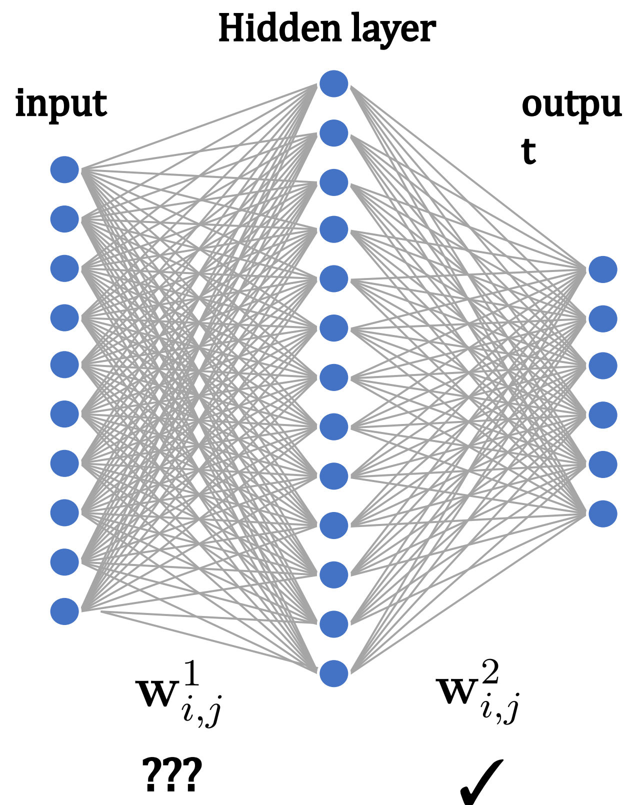
Backpropagation: Solving Credit Assignment Problem

Neural networks up until the 1970s were not very useful for two main reasons:

- Not clear how to train a NN of more than 1 layer → known as the **credit assignment problem**
- A neural network of only one layer cannot describe complex functions, two or more can represent any function (in theory with infinite width).

The credit assignment problem was solved by ***backpropagation*** , a method that describes how to distribute errors to neurons **not adjacent to the output layer**

Solution: **Dynamic programing**

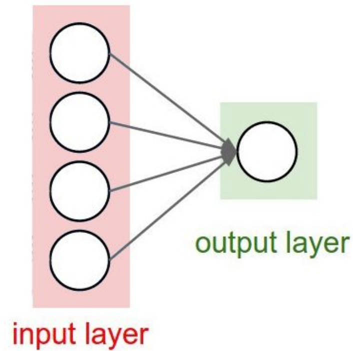


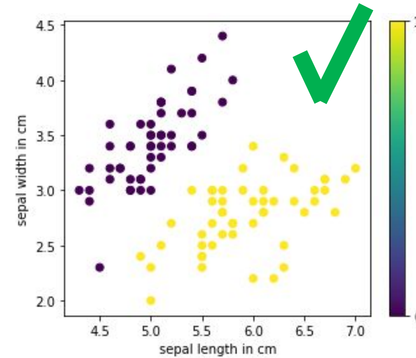
UNIVERSITY OF TORONTO

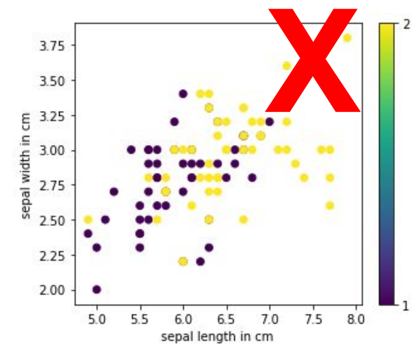
Multiple Layers with Non-Linearity

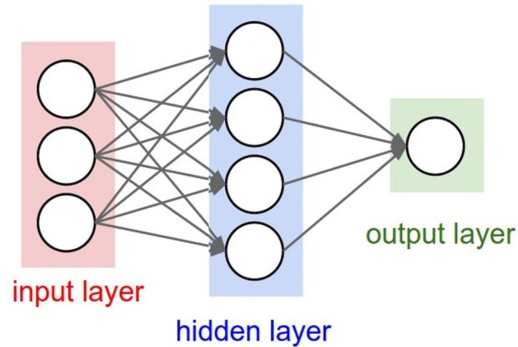
Linear

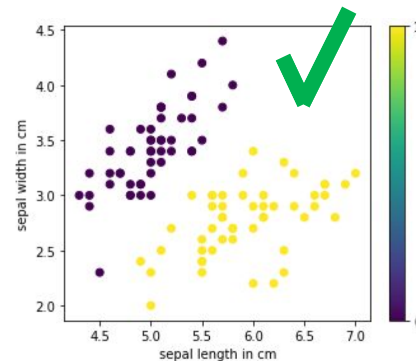
Non-Linear

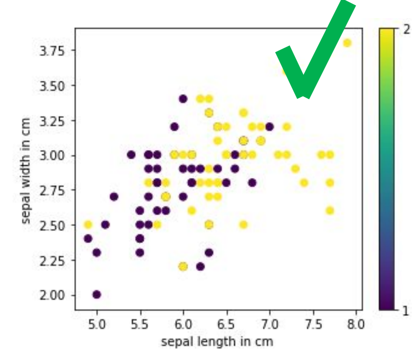












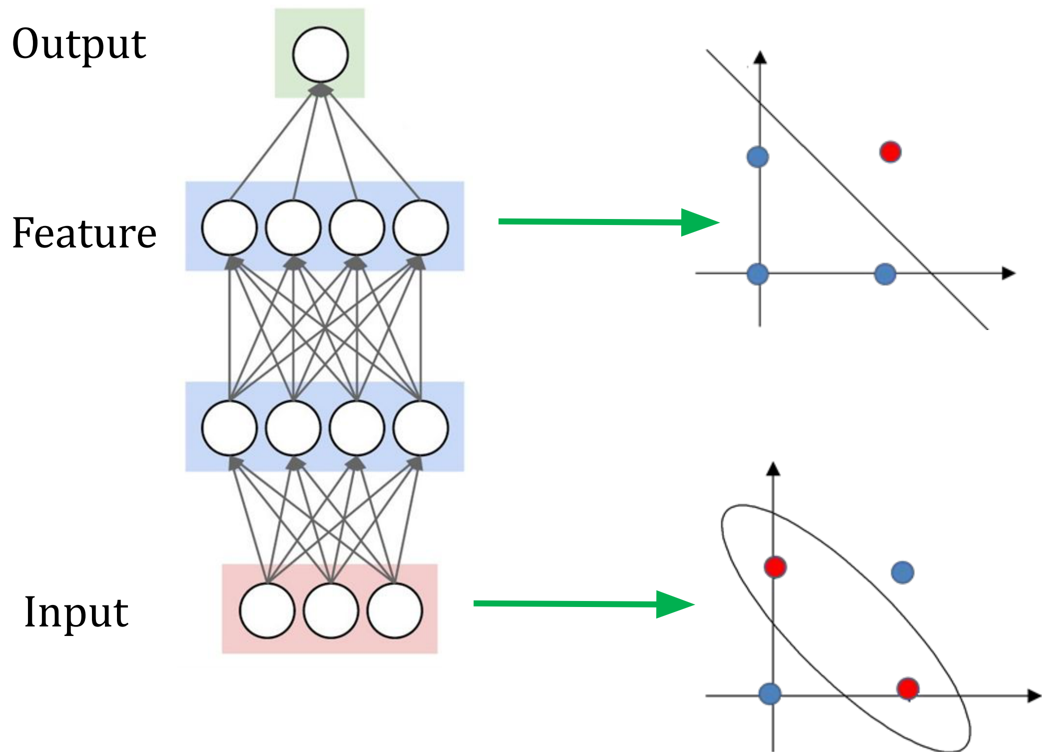
UNIVERSITY OF TORONTO

Multiple Layers with Non-Linearity

Neural Networks can be viewed as a way of **learning features** directly and end-to-end from raw input data

You can use the **activations** of the layer before the last layer as high-level features representing the input data

The goal being that the final layer is presented with a **linear separation**

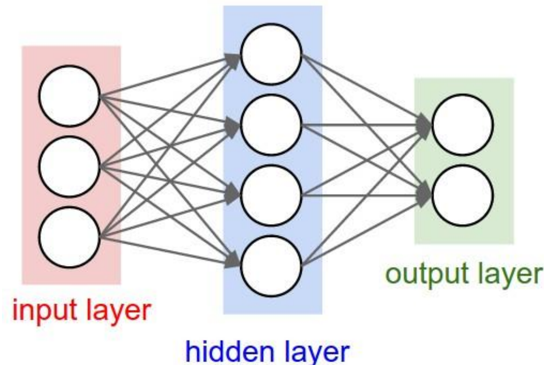


Neural Network Architecture

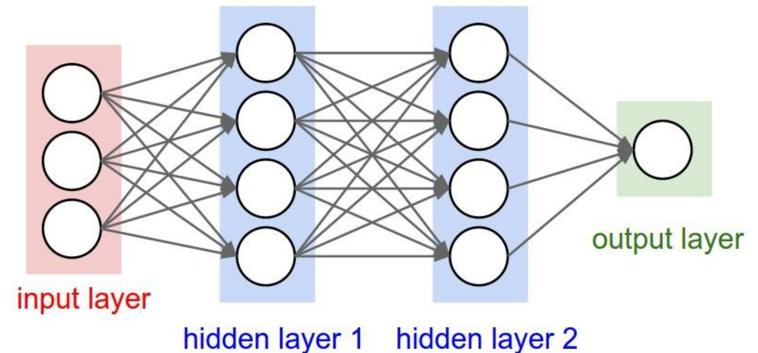
Feed-Forward Network : Information only flows forward *from one layer to a later layer*, from the input to the output.

Fully-Connected Network : Neurons between adjacent layers are fully connected.

Number of Layers: Number of hidden layers + output layer



2-layer neural network



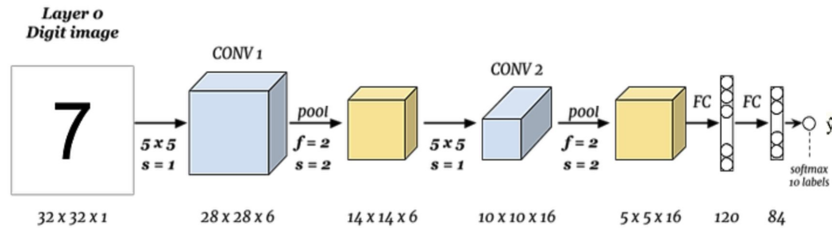
3-layer neural network

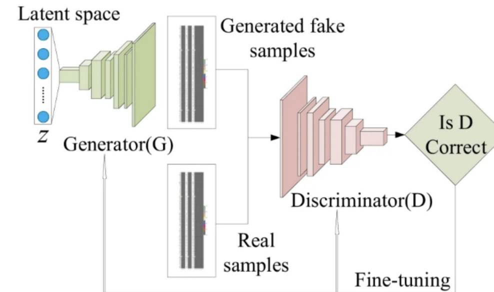
Neural Network Architecture

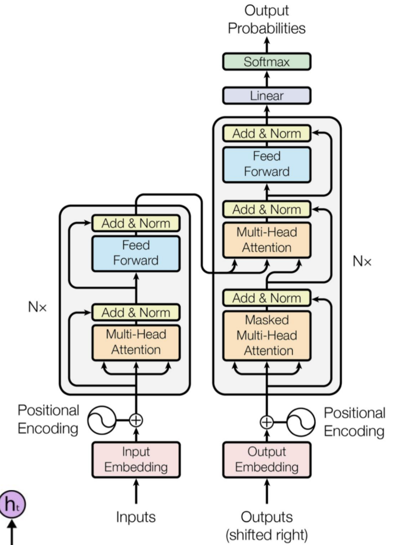
An architecture of a neural network describes the neurons and their connectivity.

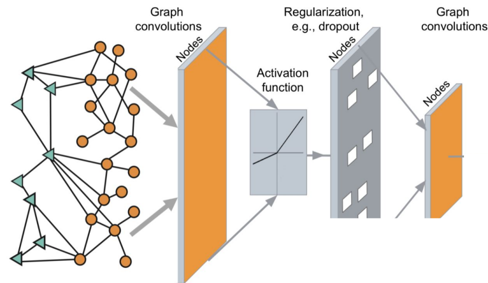
Architecture selection will greatly affect model performance.

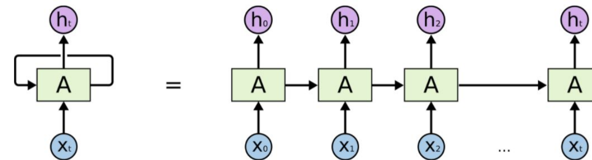
In future weeks we will introduce more neural network architecture.











UNIVERSITY OF TORONTO

Questions?