



AI Deep Learning Class

From Mystery to Mastery



What is Deep Learning?

ARTIFICIAL INTELLIGENCE

Any technique that enables computers to mimic human behavior



MACHINE LEARNING

Ability to learn without explicitly being programmed



DEEP LEARNING

Extract patterns from data using neural networks

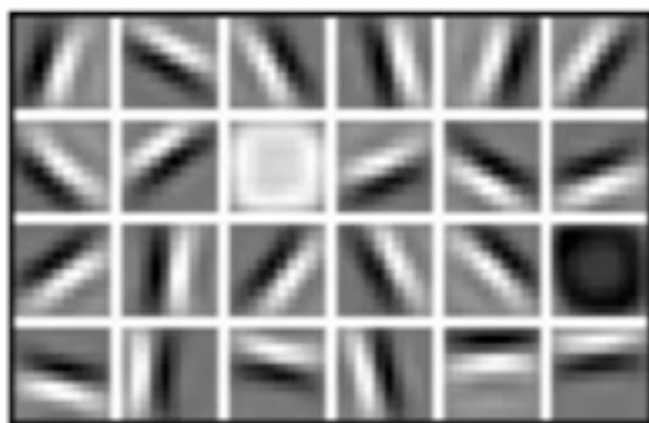
3 1 3 4 7 2
1 7 4 2 3 5

Why Deep Learning?

Hand engineered features are time consuming, brittle, and not scalable in practice

Can we learn the **underlying features** directly from data?

Low Level Features



Lines & Edges

Mid Level Features



Eyes & Nose & Ears

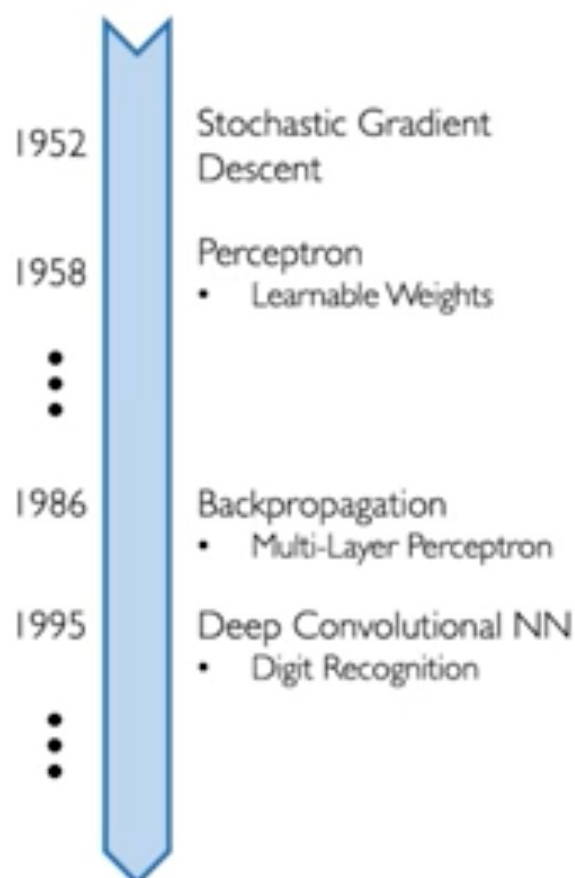
High Level Features



Facial Structure

Why Now?

Neural Networks date back decades, so why the resurgence?



1. Big Data

- Larger Datasets
- Easier Collection & Storage

IMAGENET



2. Hardware

- Graphics Processing Units (GPUs)
- Massively Parallelizable



3. Software

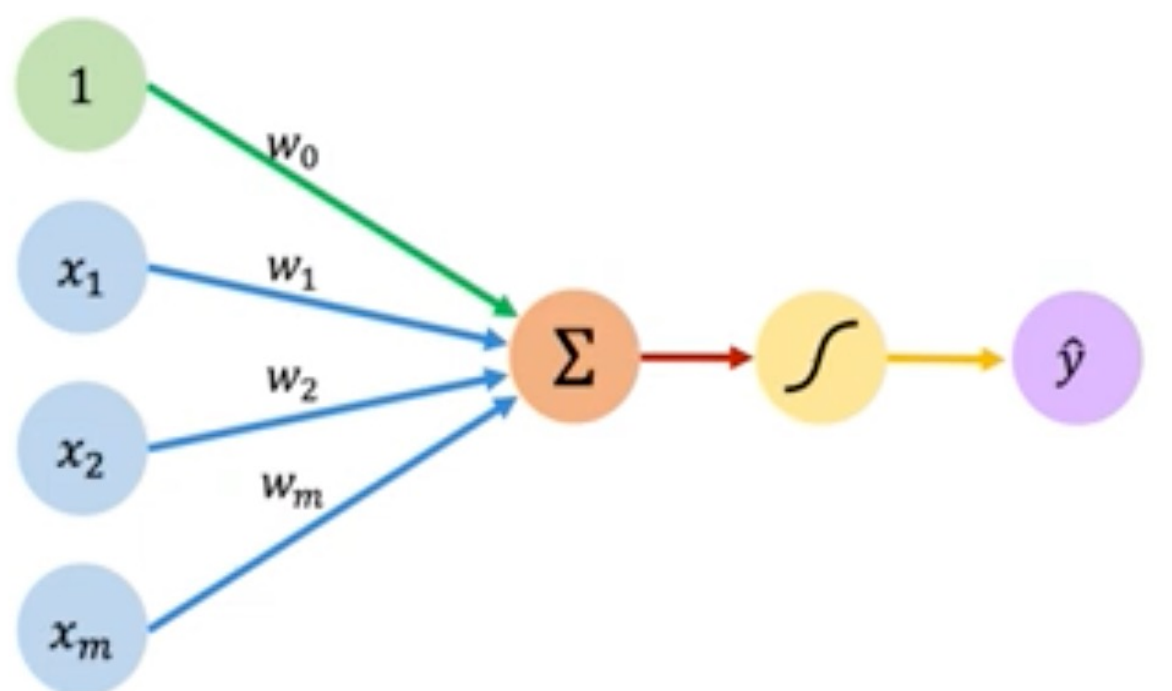
- Improved Techniques
- New Models
- Toolboxes



The Perceptron

The structural building block of deep learning

The Perceptron: Forward Propagation



Inputs Weights Sum Non-Linearity Output

Output

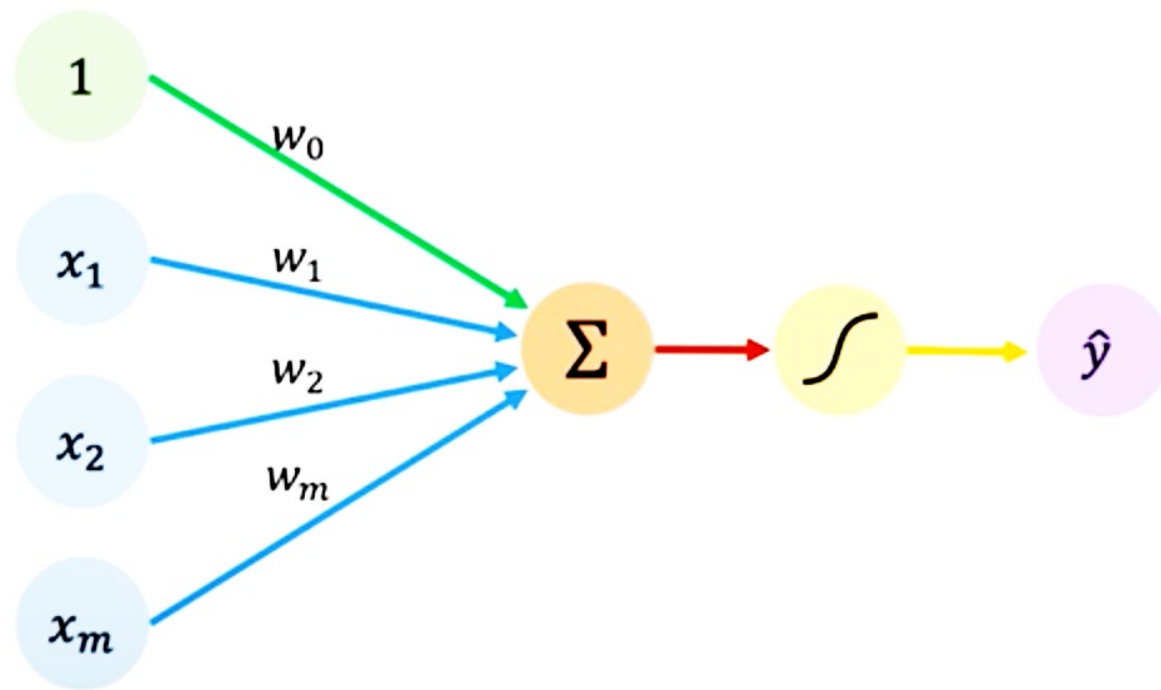
Linear combination of inputs

$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

Non-linear activation function

Bias

The Perceptron: Forward Propagation



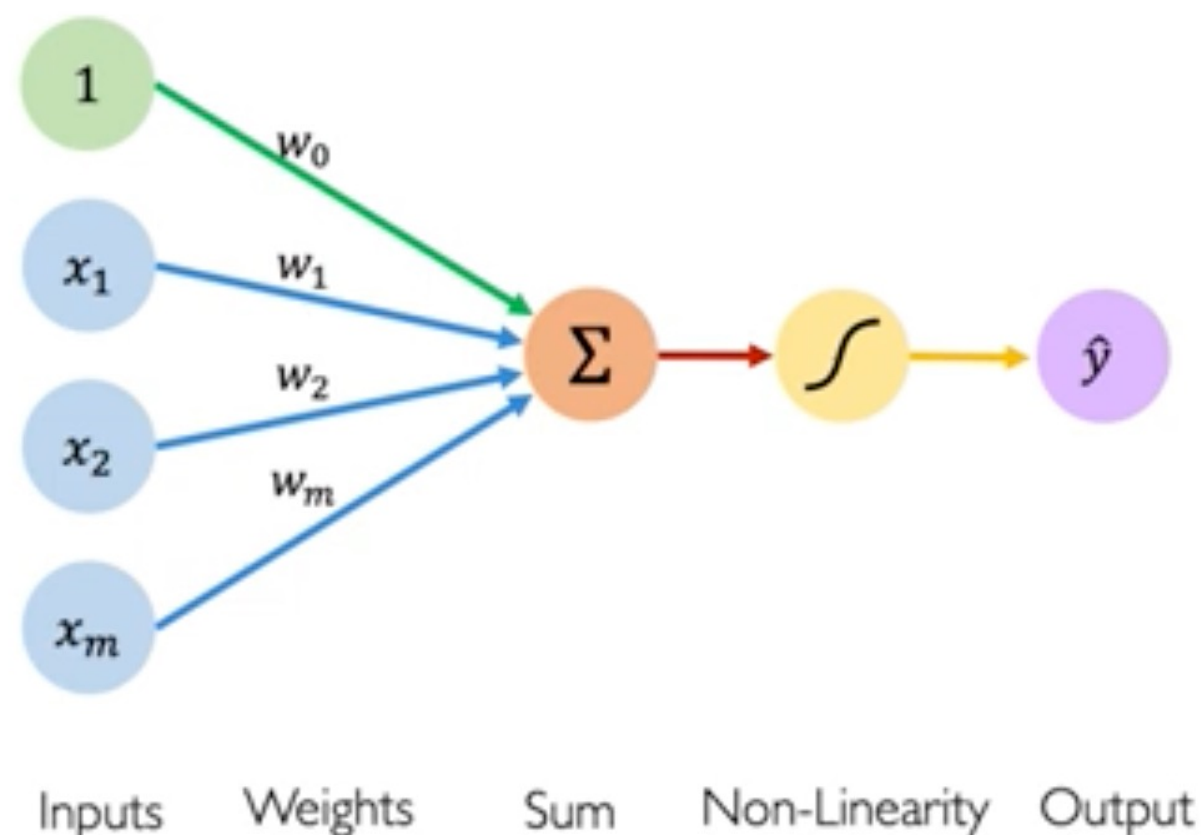
$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

$$\hat{y} = g (w_0 + \mathbf{X}^T \mathbf{W})$$

$$\text{where: } \mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Inputs Weights Sum Non-Linearity Output

The Perceptron: Forward Propagation

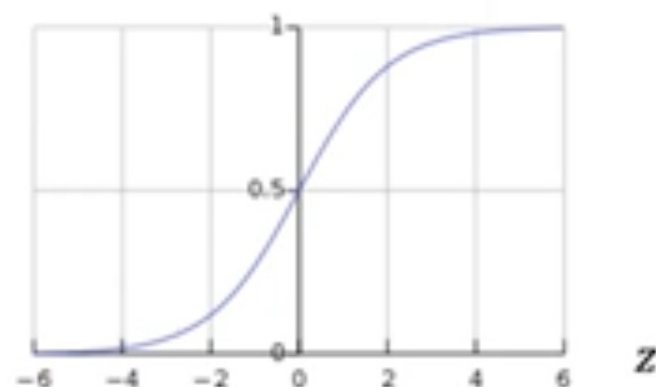


Activation Functions

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

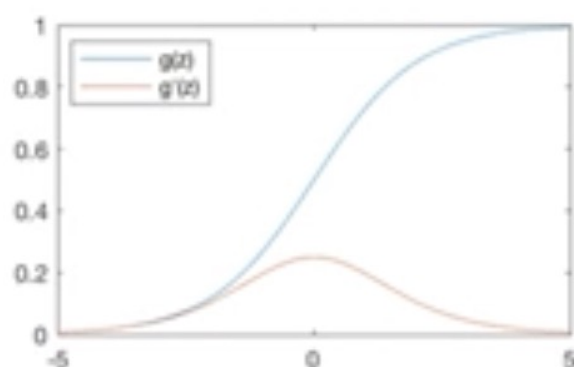
- Example: sigmoid function

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



Common Activation Functions

Sigmoid Function

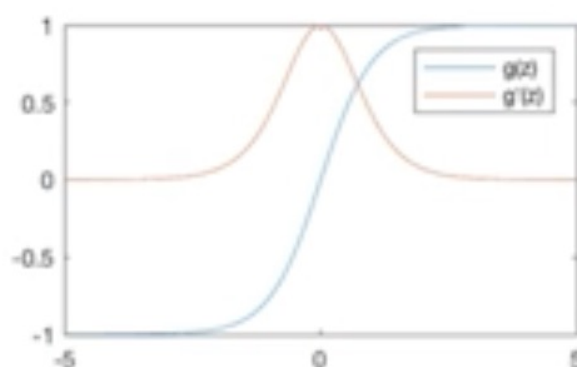


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

 `tf.math.sigmoid(z)`

Hyperbolic Tangent



$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

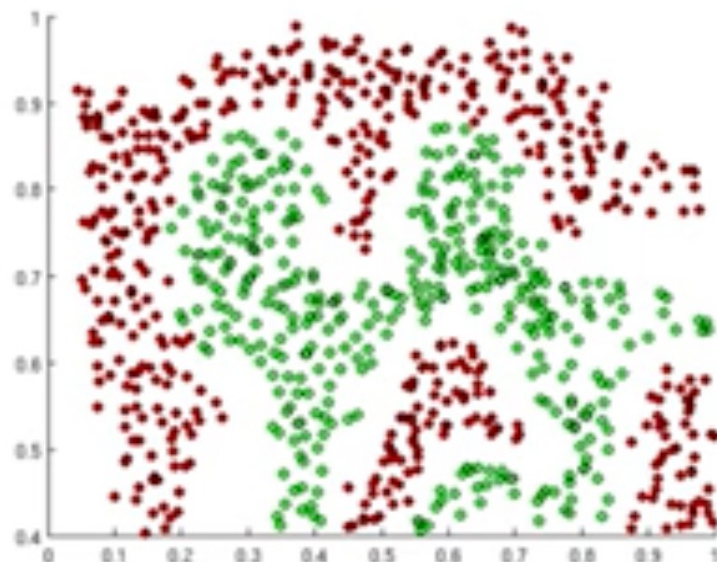
```
def sigmoid(z): return 1.0/(1.0+np.exp(-z))
```

 TensorFlow code blocks

NOTE: All activation functions are non-linear

Importance of Activation Functions

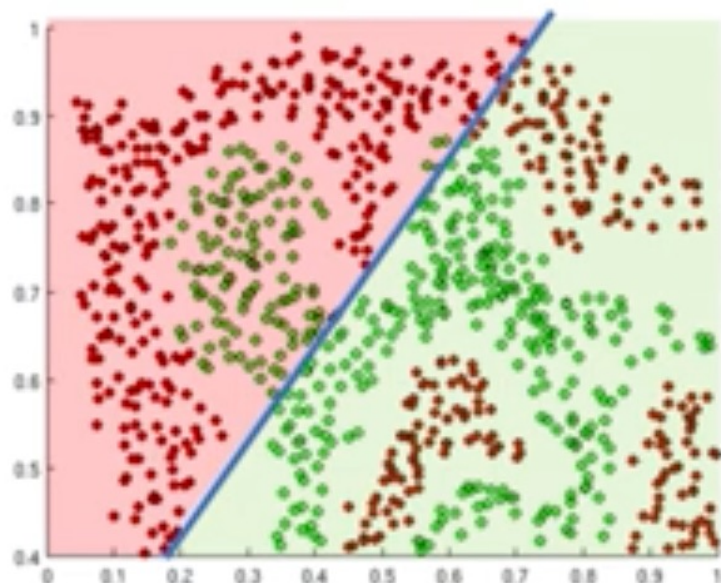
*The purpose of activation functions is to **introduce non-linearities** into the network*



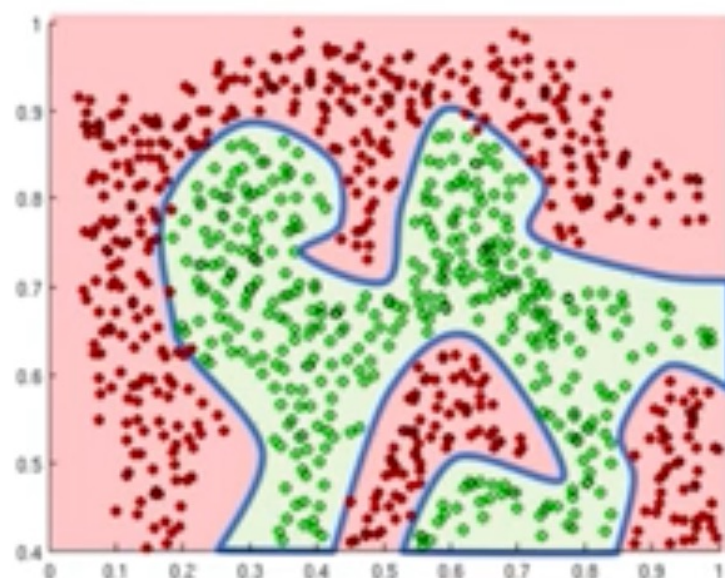
What if we wanted to build a neural network to distinguish green vs red points?

Importance of Activation Functions

The purpose of activation functions is to *introduce non-linearities* into the network

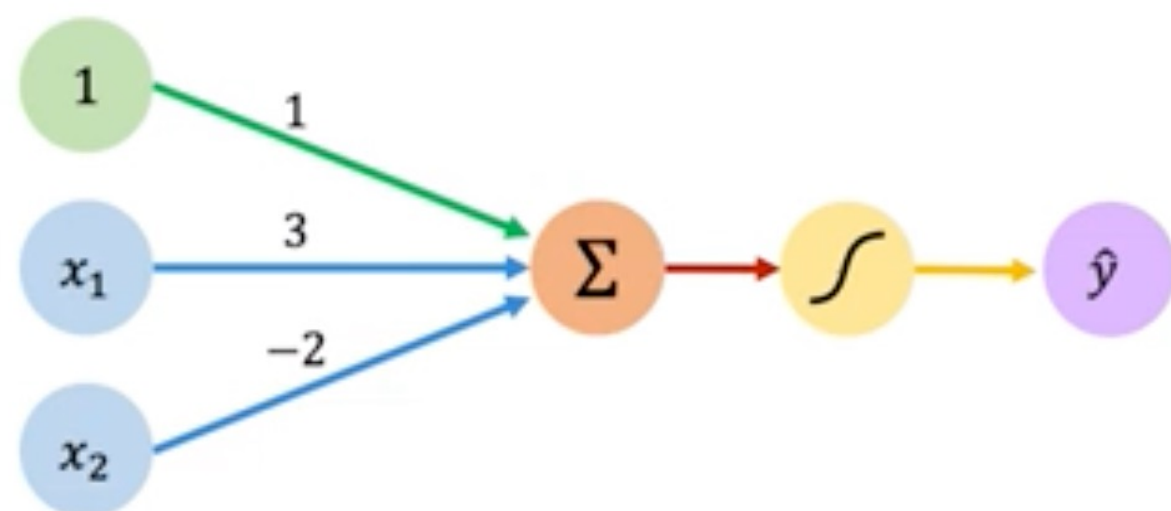


Linear activation functions produce linear decisions no matter the network size



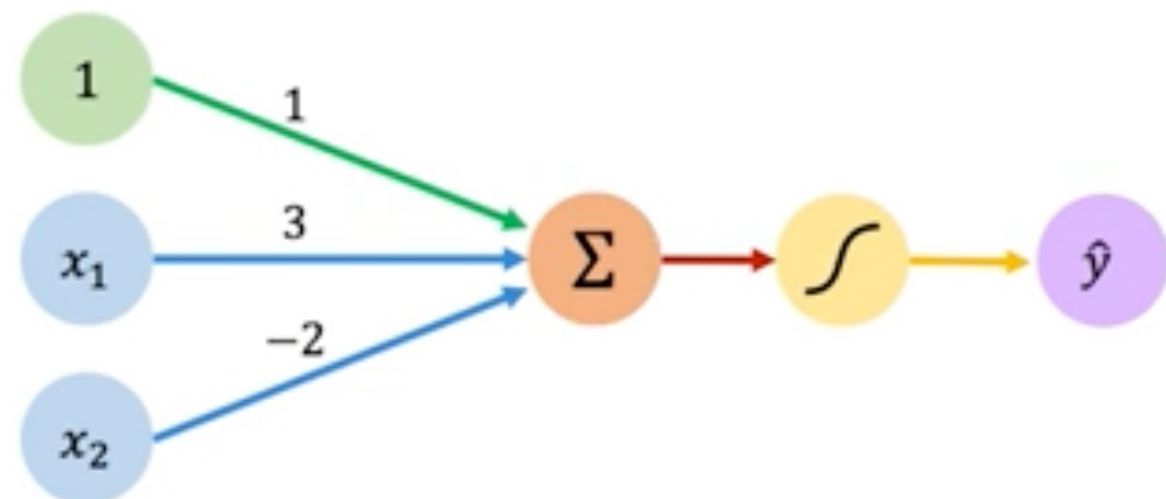
Non-linearities allow us to approximate arbitrarily complex functions

The Perceptron: Example



We have: $w_0 = 1$ and $\mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

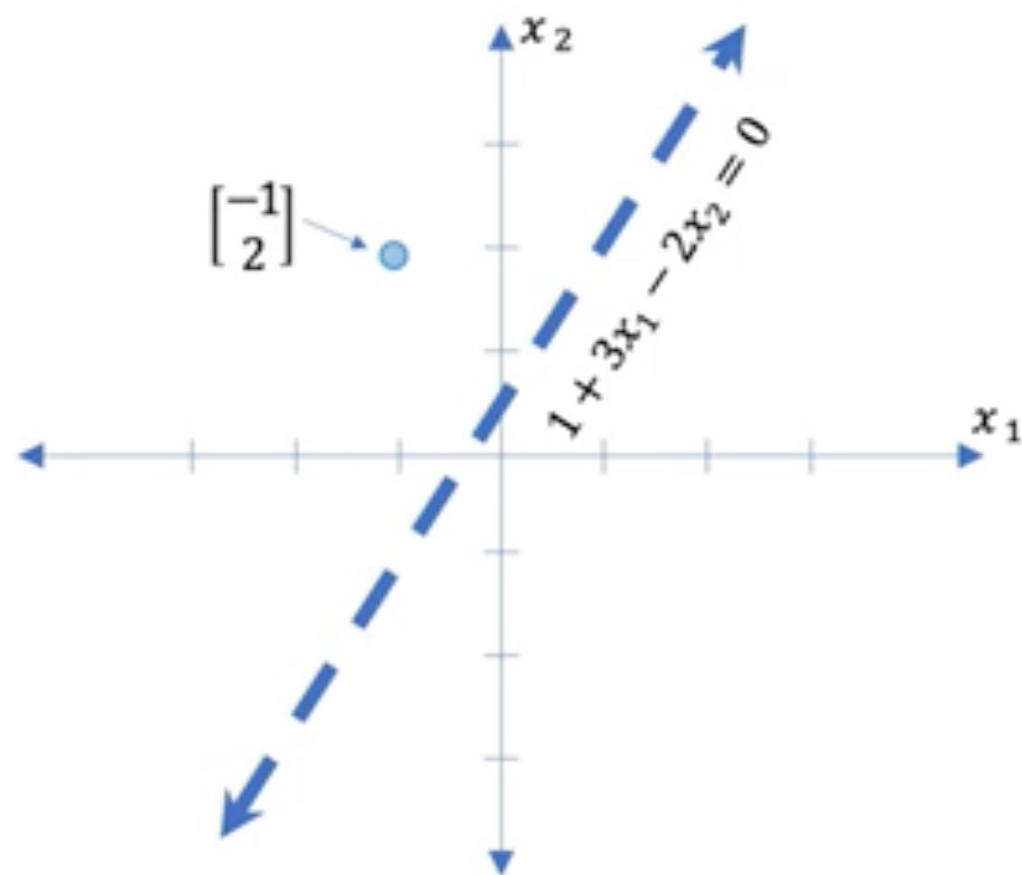
The Perceptron: Example



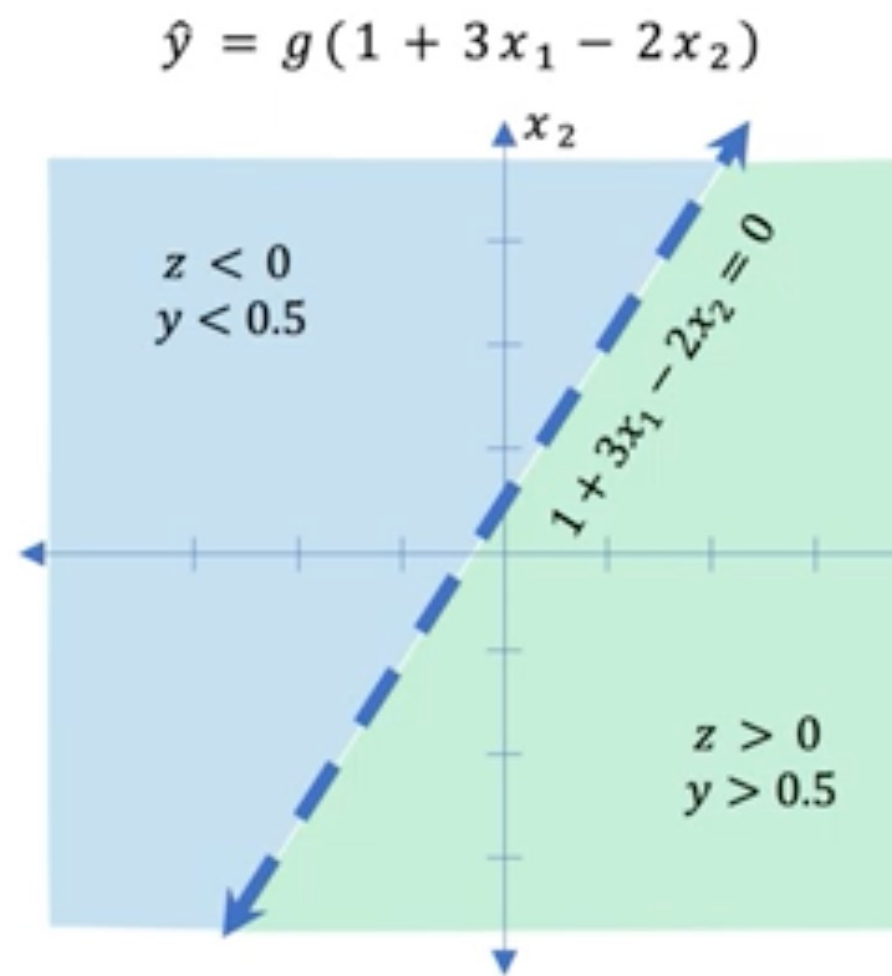
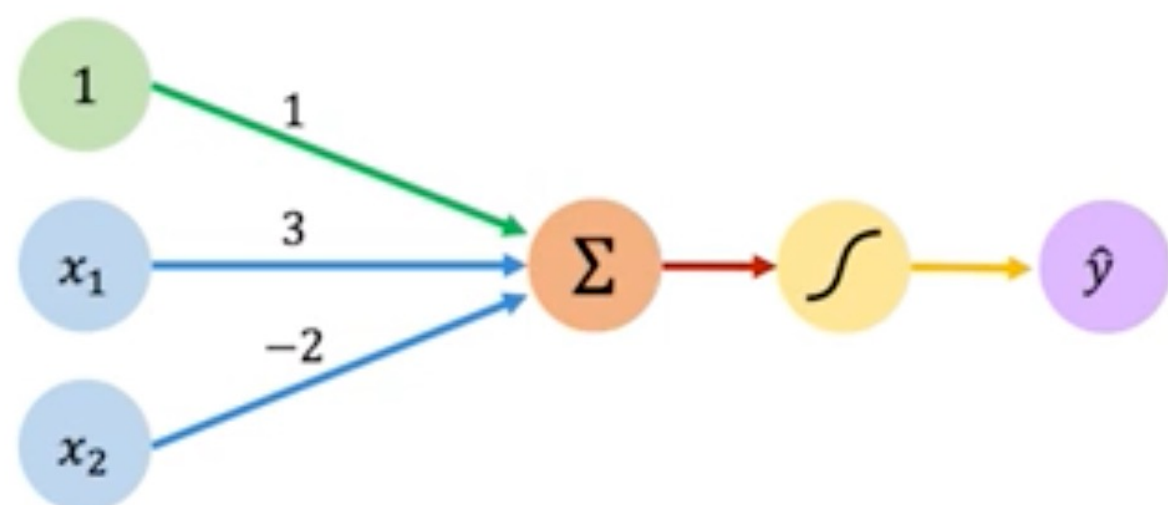
Assume we have input: $\mathbf{X} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



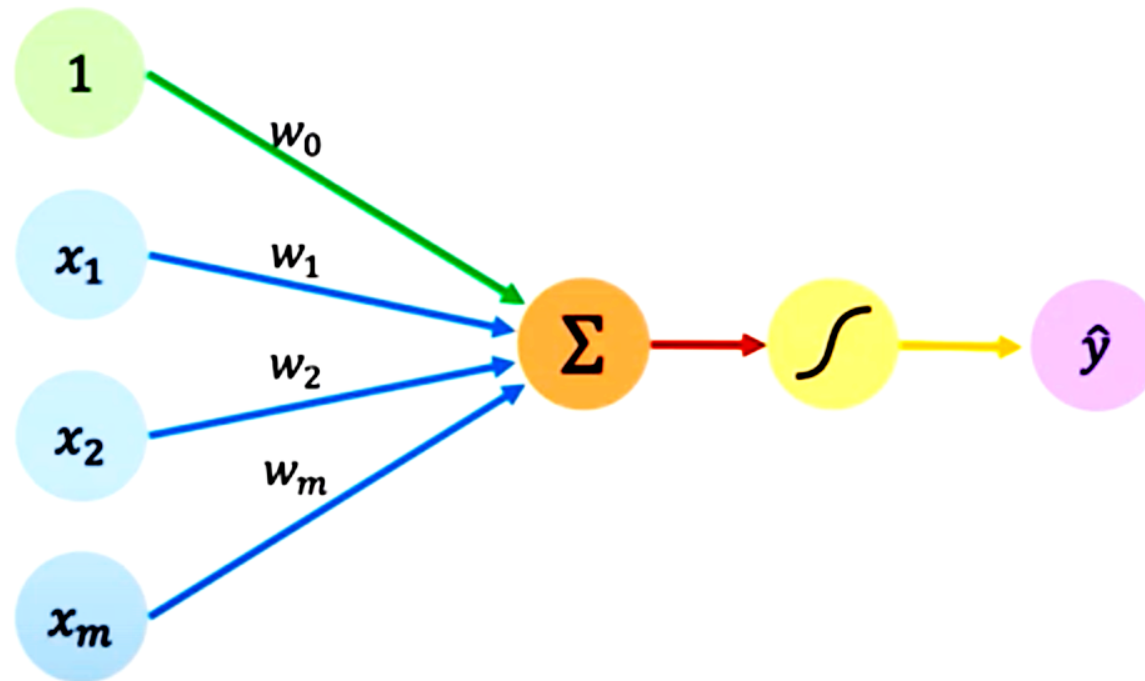
The Perceptron: Example



Building Neural Networks with Perceptrons

The Perceptron: Simplified

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$



Inputs

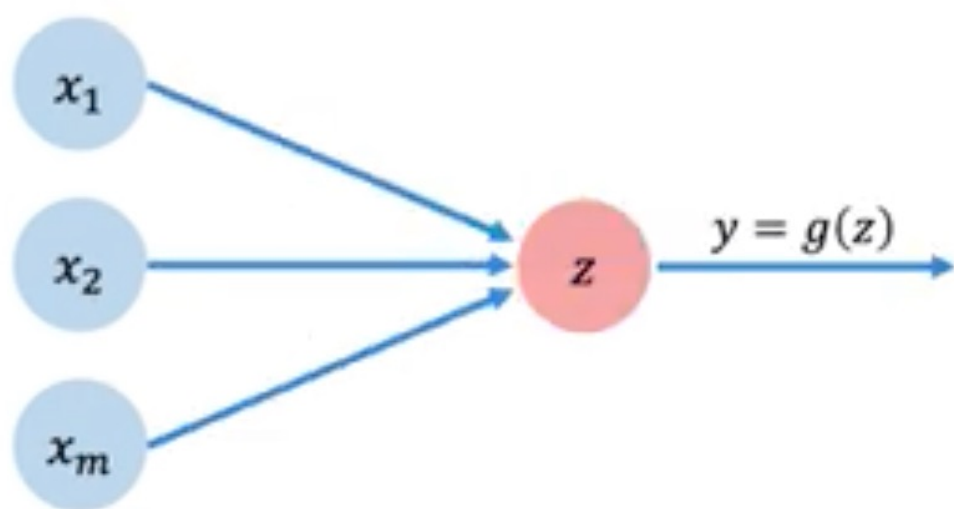
Weights

Sum

Non-Linearity

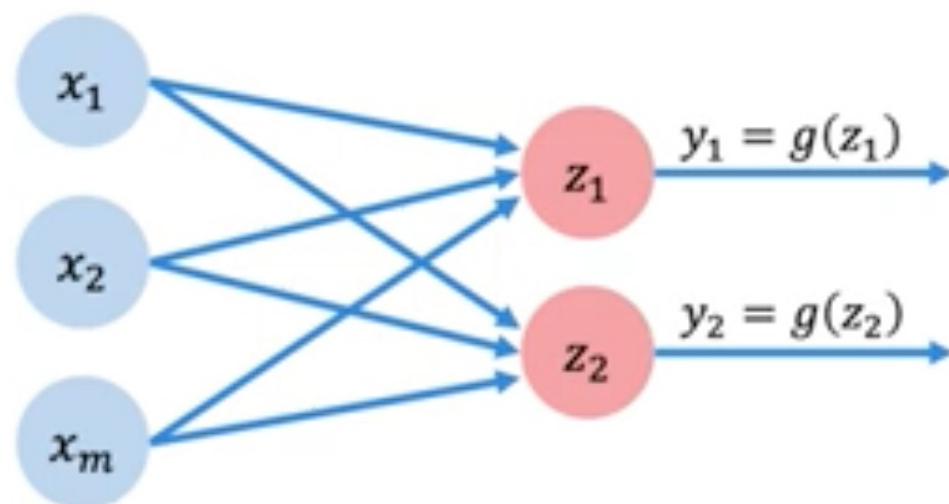
Output

The Perceptron: Simplified



$$z = w_0 + \sum_{j=1}^m x_j w_j$$

Multi Output Perceptron



```
 import tensorflow as tf  
  
layer = tf.keras.layers.Dense(  
    units=2)
```

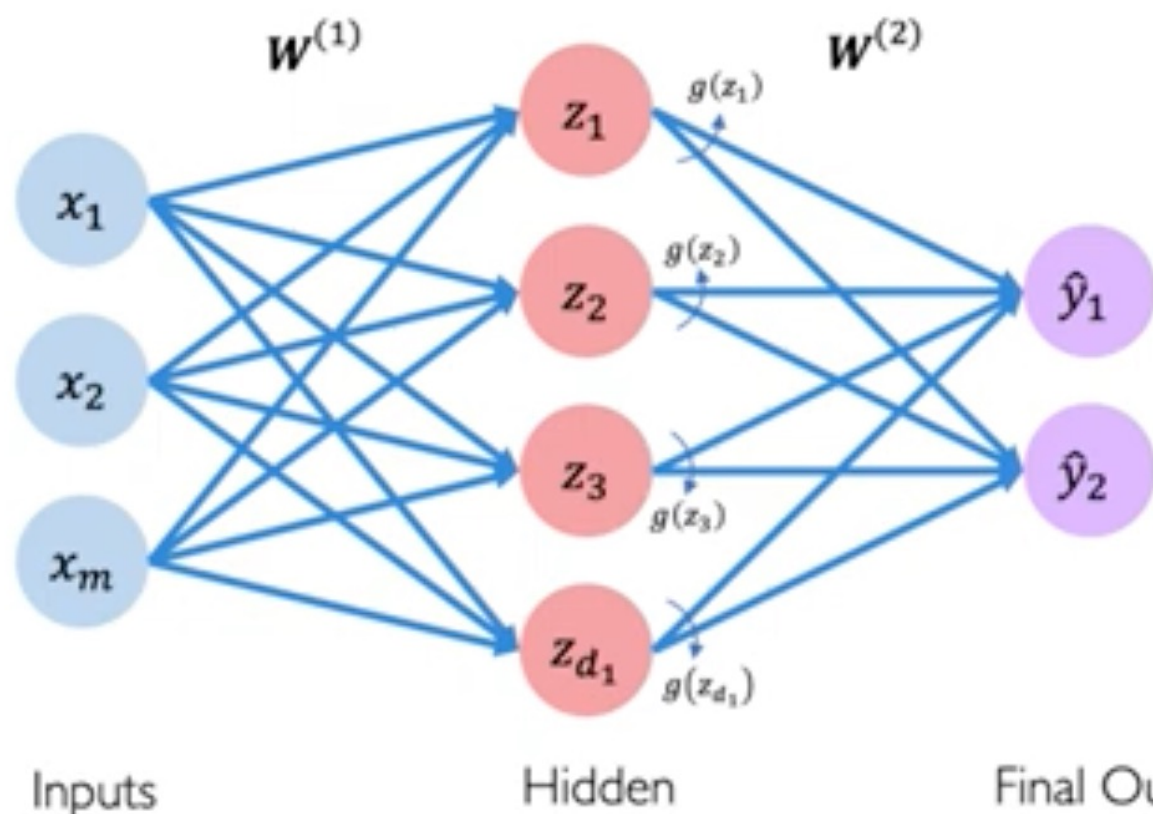
$$z_i = w_{0,i} + \sum_{j=1}^m x_j w_{j,i}$$

Dense layer from scratch



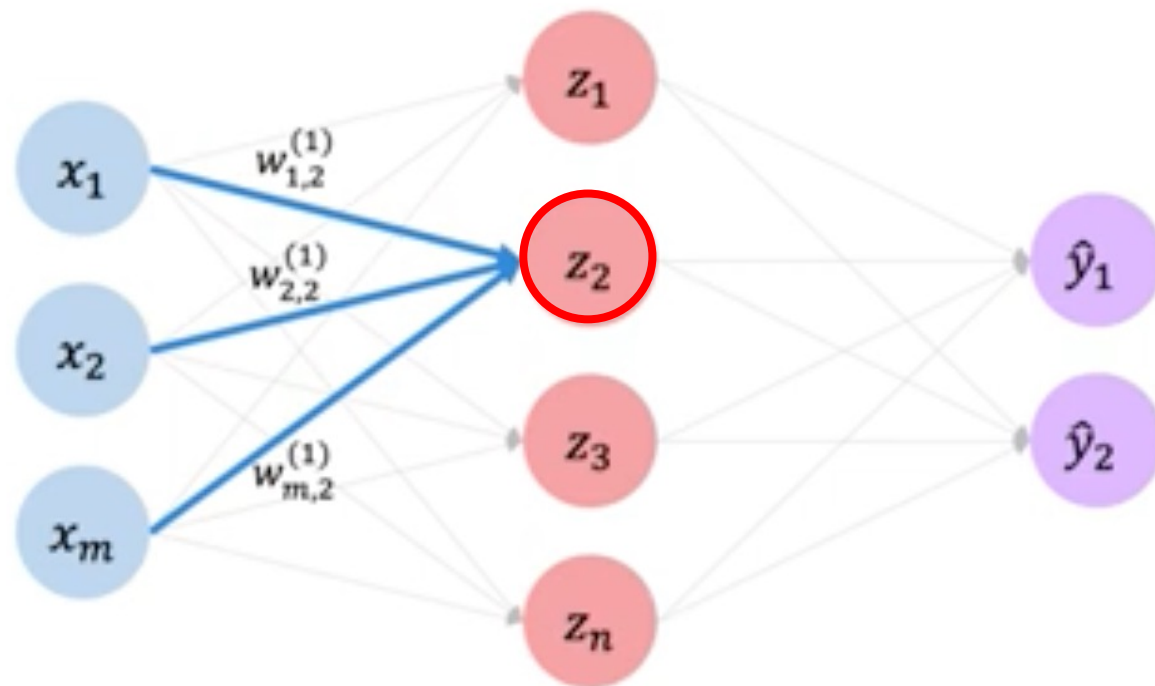
```
class MyDenseLayer(tf.keras.layers.Layer):  
    def __init__(self, input_dim, output_dim):  
        super(MyDenseLayer, self).__init__()  
  
        # Initialize weights and bias  
        self.W = self.add_weight([input_dim, output_dim])  
        self.b = self.add_weight([1, output_dim])  
  
    def call(self, inputs):  
        # Forward propagate the inputs  
        z = tf.matmul(inputs, self.W) + self.b  
  
        # Feed through a non-linear activation  
        output = tf.math.sigmoid(z)  
  
        return output
```

Single Layer Neural Network



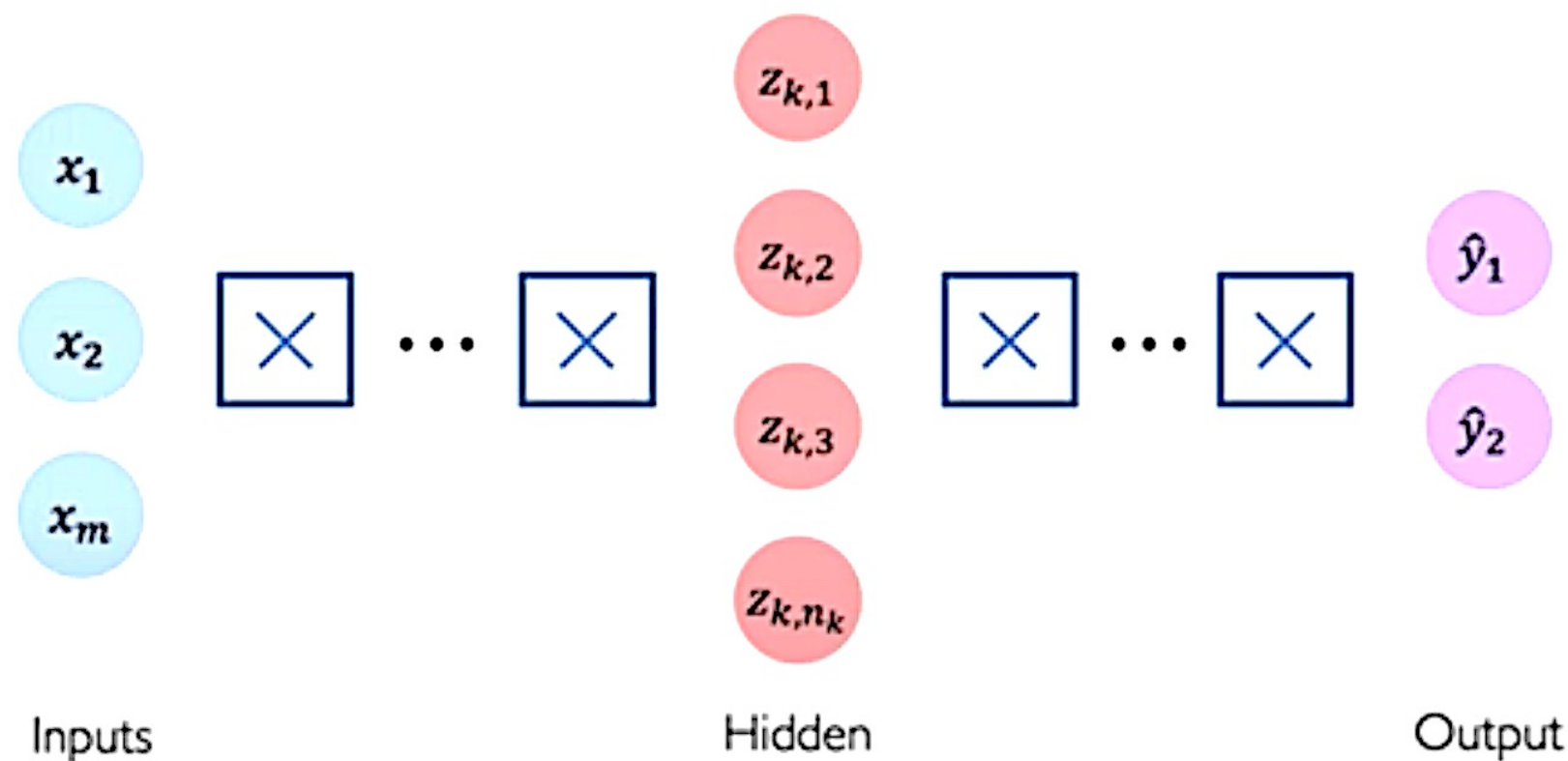
$$z_l = w_{0,l}^{(1)} + \sum_{j=1}^m x_j w_{j,l}^{(1)} \quad \hat{y}_l = g \left(w_{0,l}^{(2)} + \sum_{j=1}^{d_1} g(z_j) w_{j,l}^{(2)} \right)$$

Single Layer Neural Network



$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \end{aligned}$$

Deep Neural Network



```
  
import tensorflow as tf  
  
model = tf.keras.Sequential([  
    tf.keras.layers.Dense(n1),  
    tf.keras.layers.Dense(n2),  
    :  
    tf.keras.layers.Dense(2)  
])
```

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

Applying Neural Networks

Example Problem

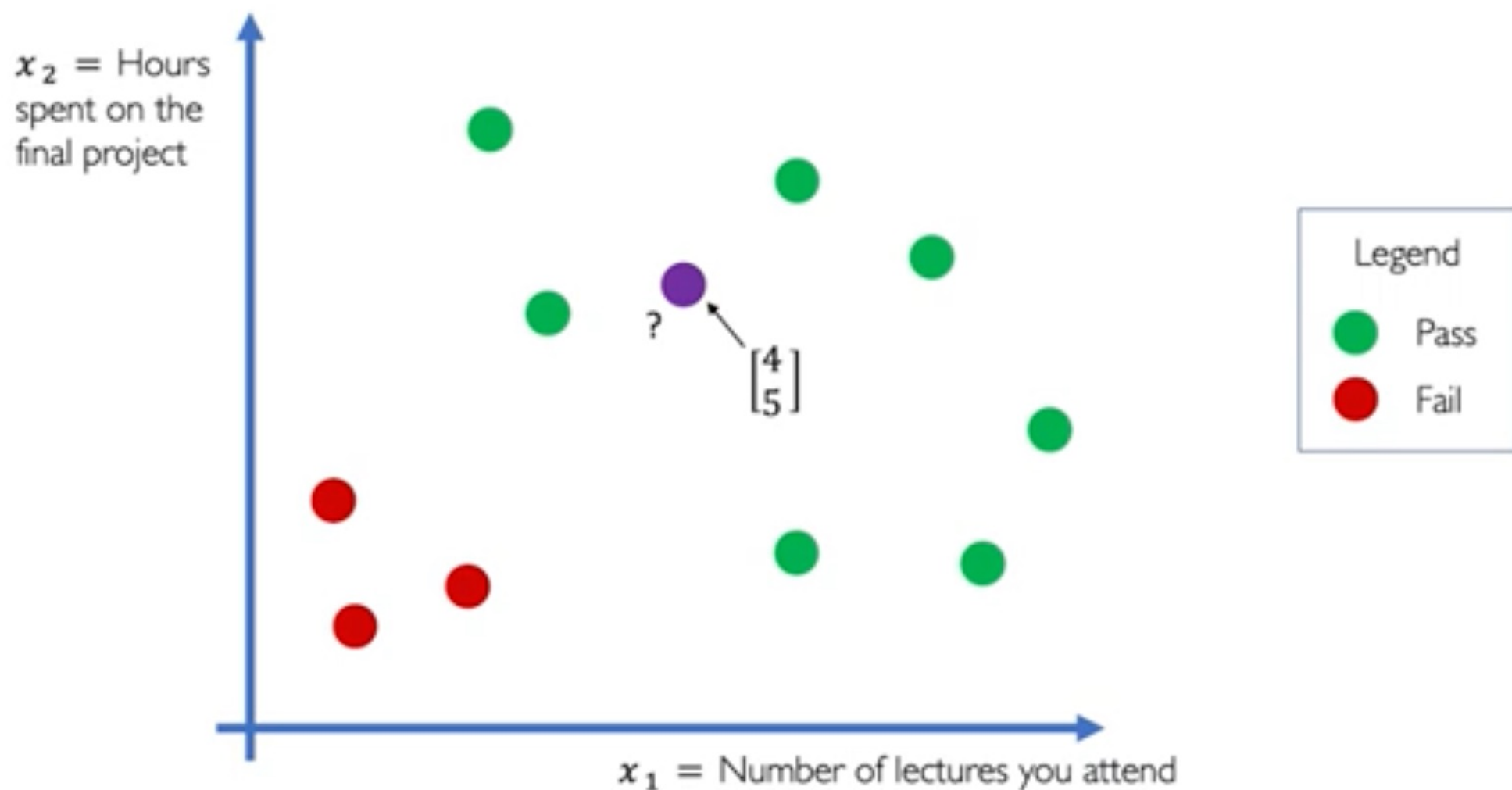
Will I pass this class?

Let's start with a simple two feature model

x_1 = Number of lectures you attend

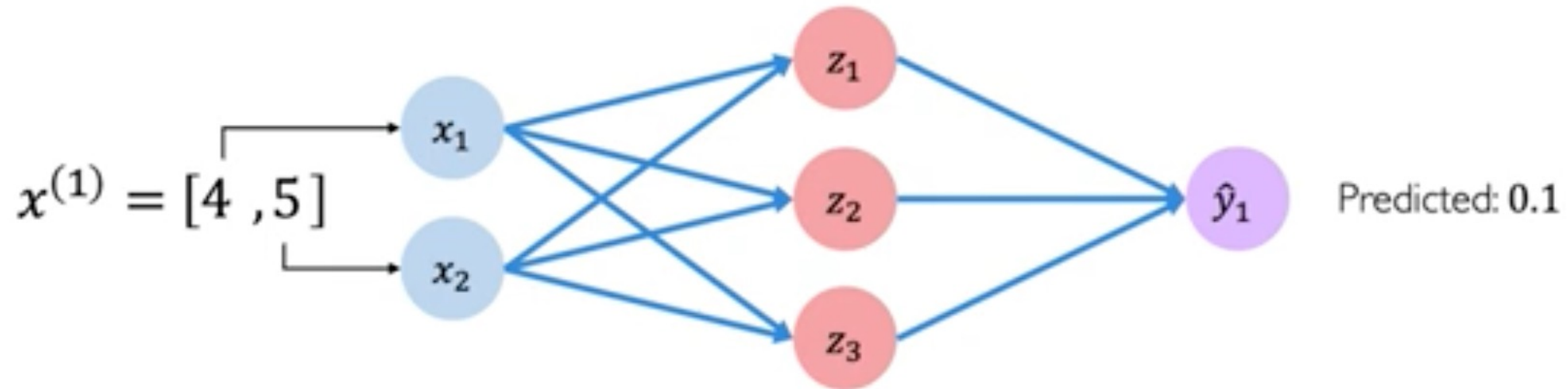
x_2 = Hours spent on the final project

Example Problem: Will I pass this class?



Example Problem: Will I pass this class?

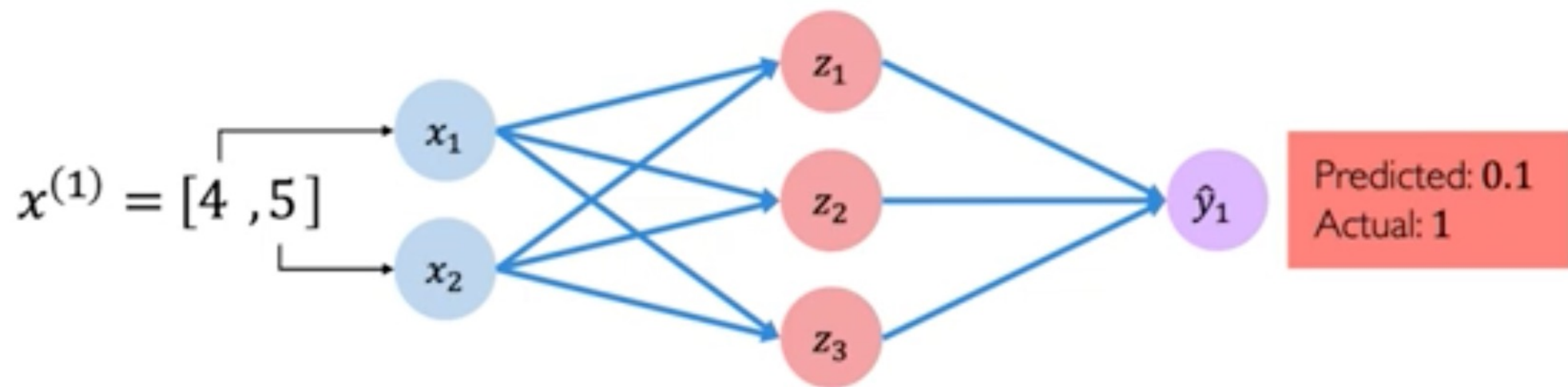
Given: Training Data



Task: Learn a classification function

Quantifying Loss

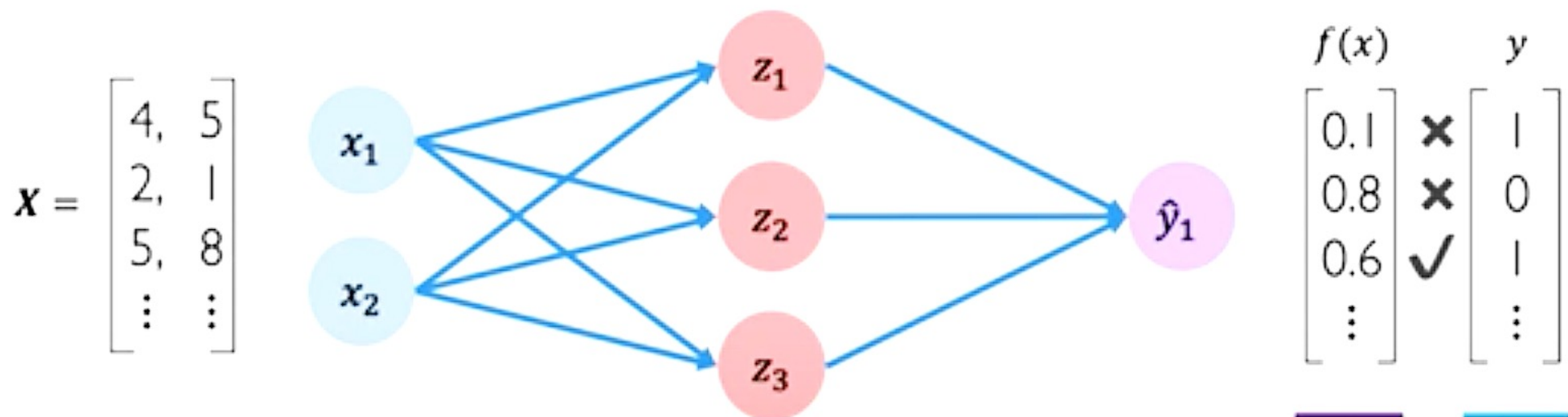
The **loss** of our network measures the cost incurred from incorrect predictions



$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss

The **empirical loss** measures the total loss over our entire dataset

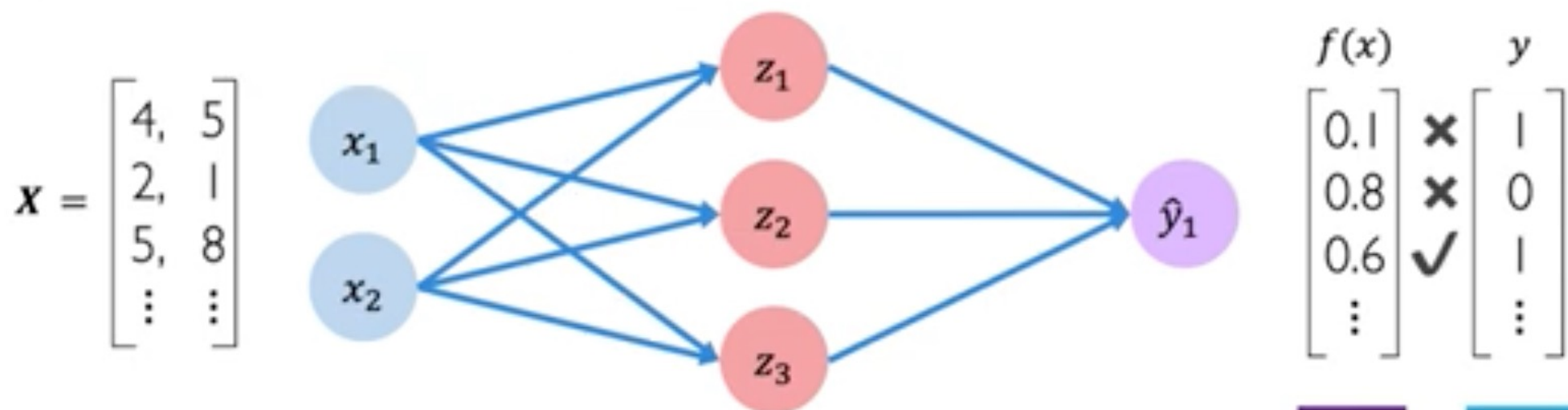


- Also known as:
- Objective function
 - Cost function

$$J(W) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\underbrace{f(x^{(i)}; W)}_{\text{purple}}, \underbrace{y^{(i)}}_{\text{blue}})$$

Binary Cross Entropy Loss

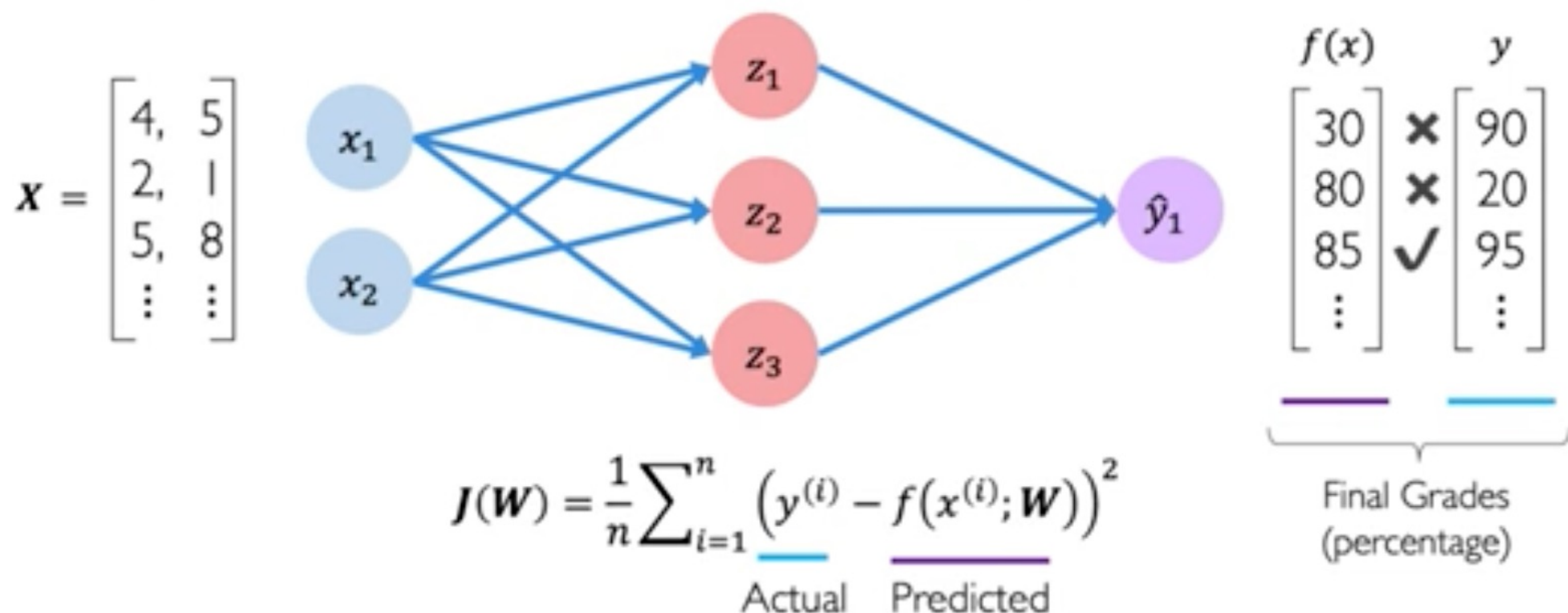
Cross entropy loss can be used with models that output a probability between 0 and 1



$$J(W) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left(1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right)$$

Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )  
loss = tf.keras.losses.MSE( y, predicted )
```

Training Neural Networks

Loss Optimization

We want to find the network weights that *achieve the lowest loss*

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})$$

$$\mathbf{W}^* = \operatorname{argmin}_{\mathbf{W}} J(\mathbf{W})$$

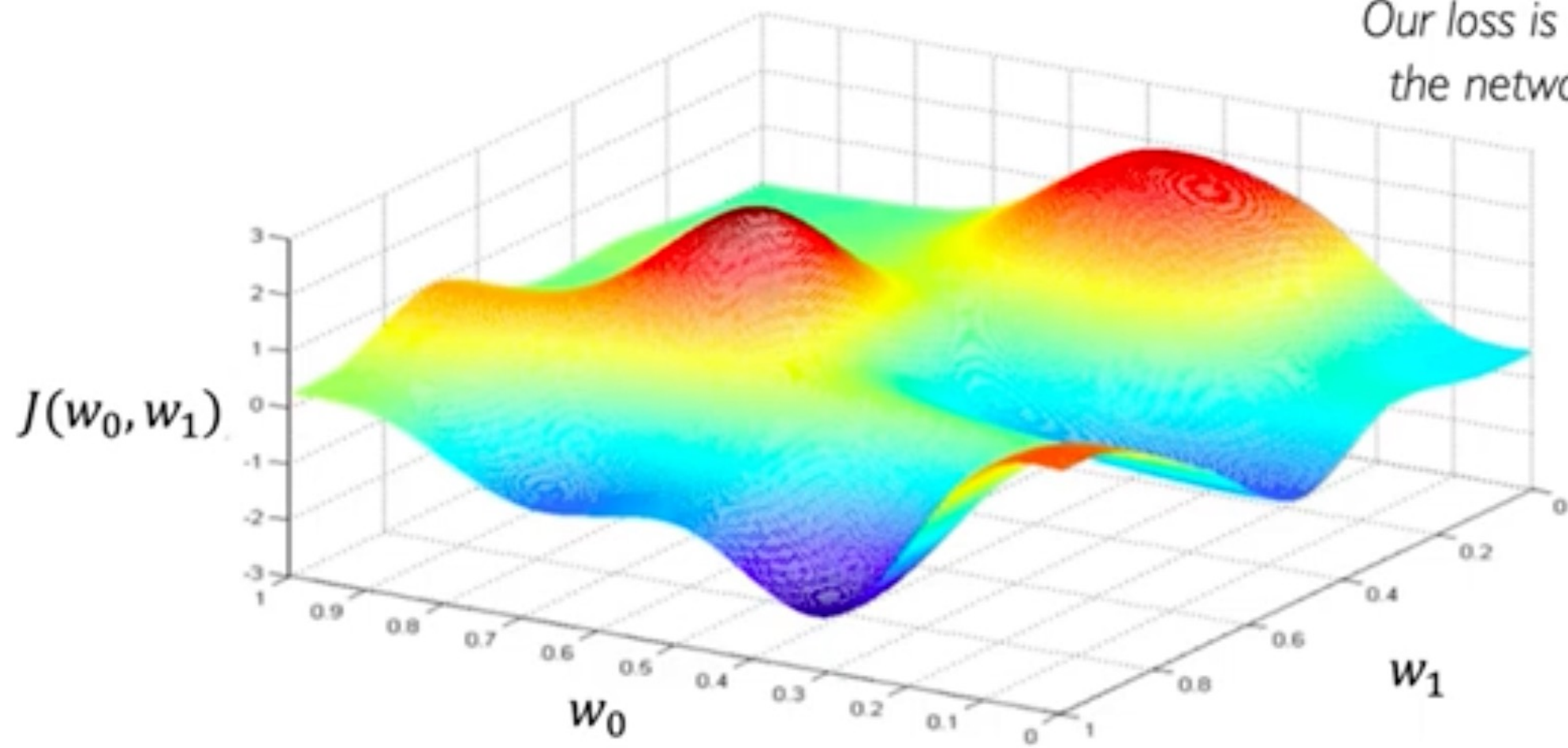
Remember:

$$\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \dots\}$$

Loss Optimization

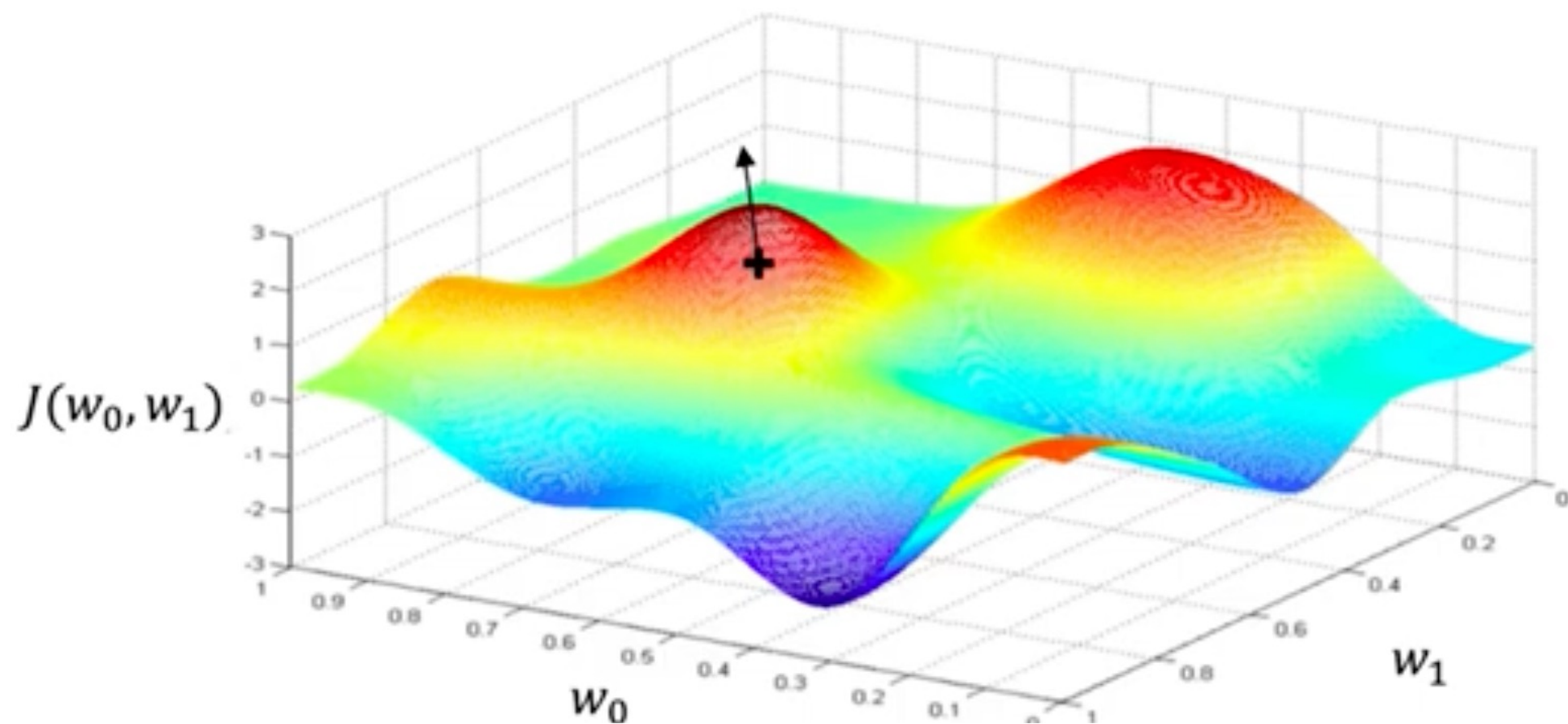
$$W^* = \underset{W}{\operatorname{argmin}} J(W)$$

Remember:
*Our loss is a function of
the network weights!*



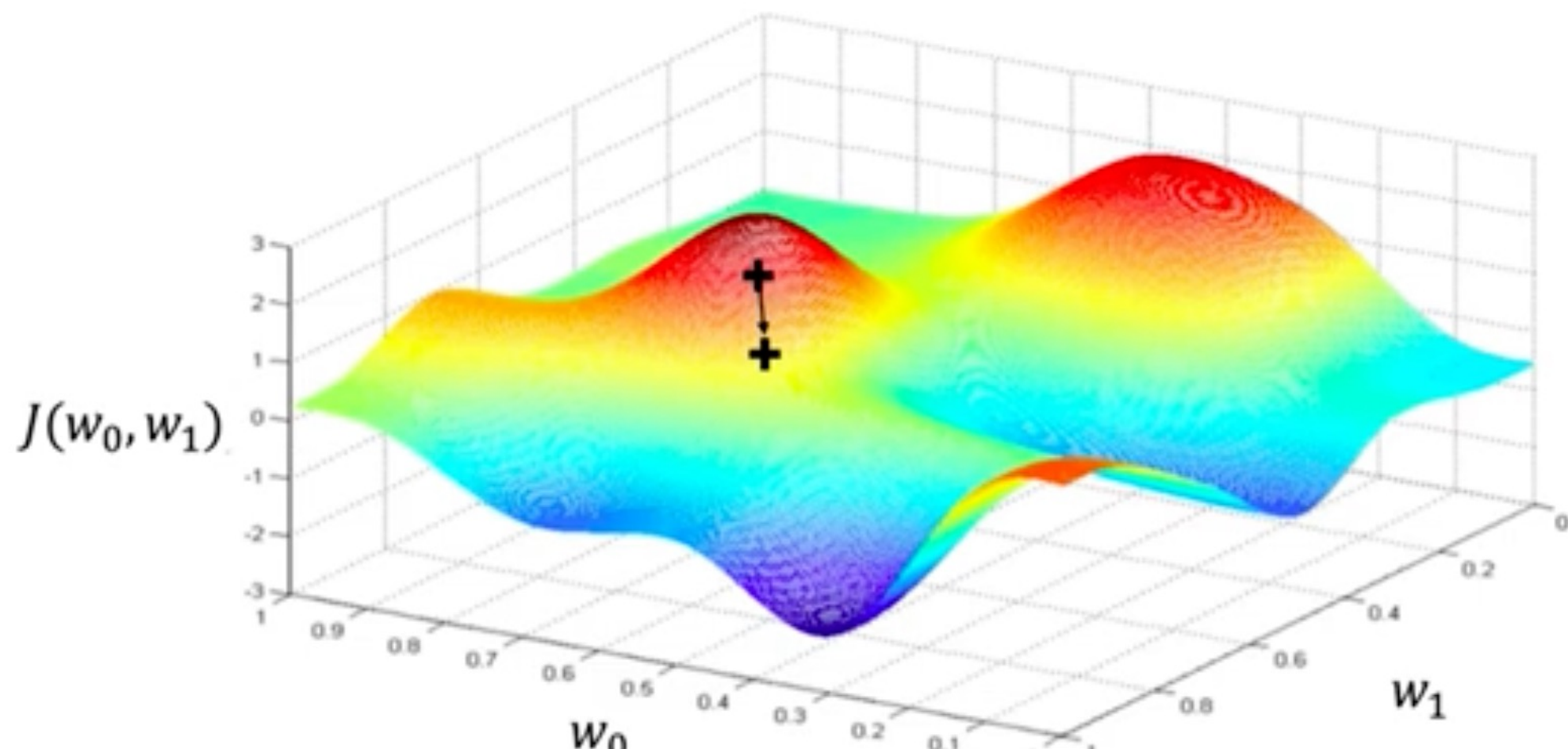
Loss Optimization

Compute gradient, $\frac{\partial J(W)}{\partial W}$



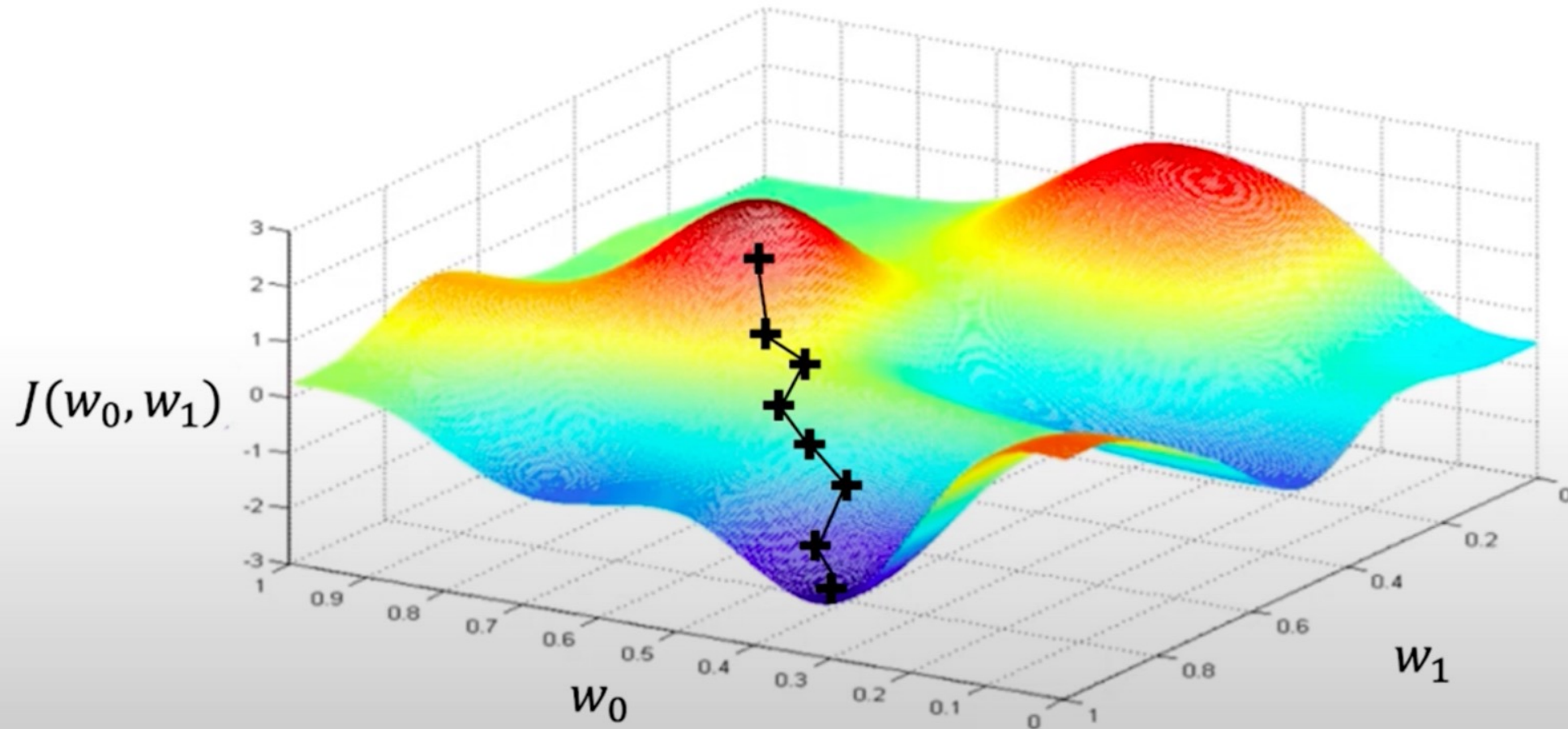
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence



Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
5. Return weights

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights



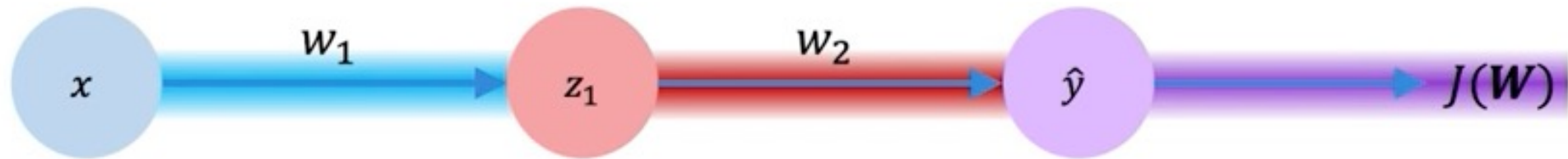
```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

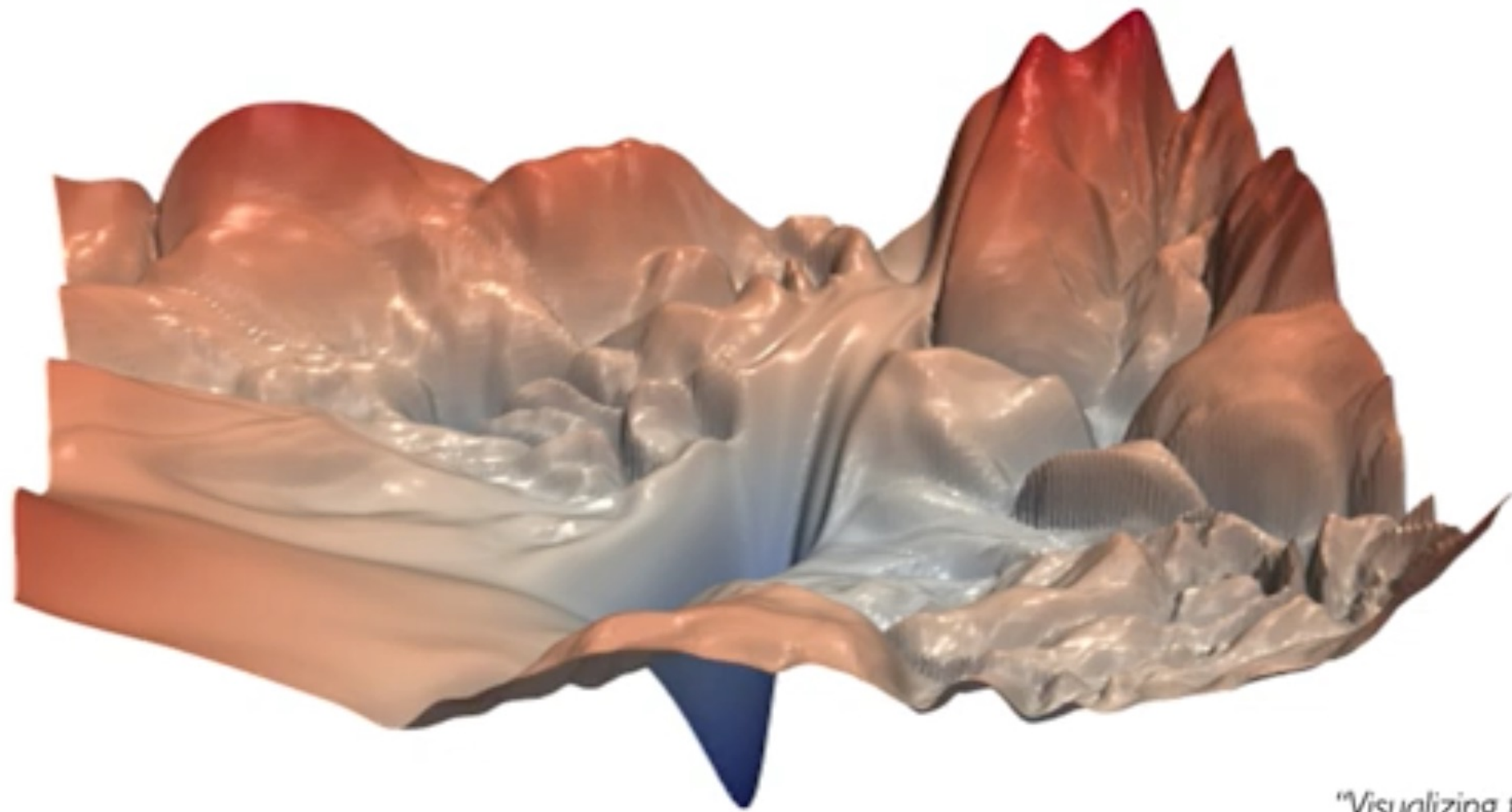
Computing Gradients: Backpropagation



$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Loss Functions Can Be Difficult to Optimize

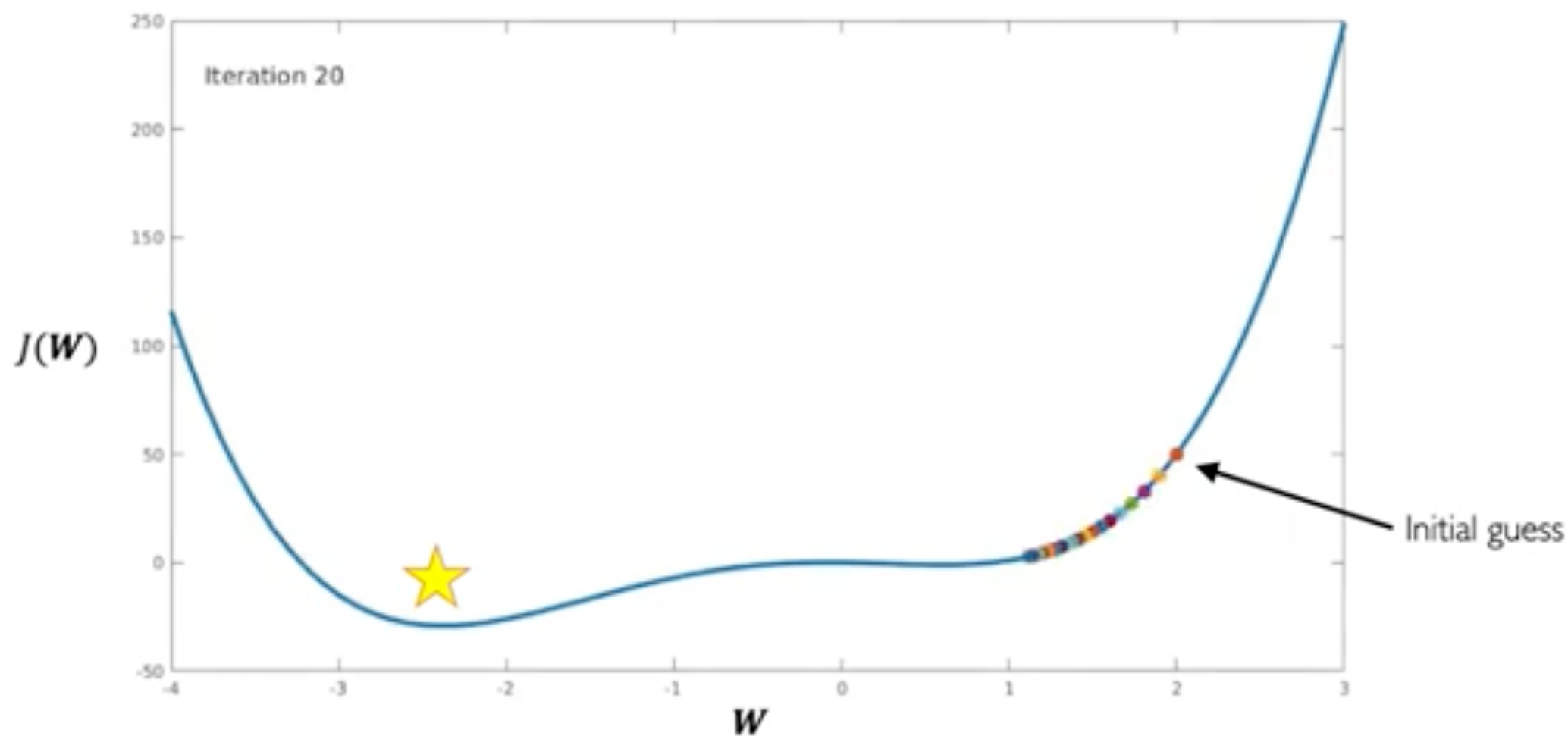
Remember:

Optimization through gradient descent

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$$

Setting the Learning Rate

Small learning rate converges slowly and gets stuck in false local minima



How to deal with this?

Idea 1:

Try lots of different learning rates and see what works “just right”

Idea 2:

Do something smarter!

Design an adaptive learning rate that “adapts” to the landscape

Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
 - how large gradient is
 - how fast learning is happening
 - size of particular weights
 - etc...

Gradient Descent Algorithms

Algorithm	TF Implementation	Reference
• SGD	 <code>tf.keras.optimizers.SGD</code>	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	 <code>tf.keras.optimizers.Adam</code>	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelta	 <code>tf.keras.optimizers.Adadelta</code>	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	

Additional details: <http://ruder.io/optimizing-gradient-descent/>



Putting it all together

```
import tensorflow as tf

model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True: # loop forever

    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

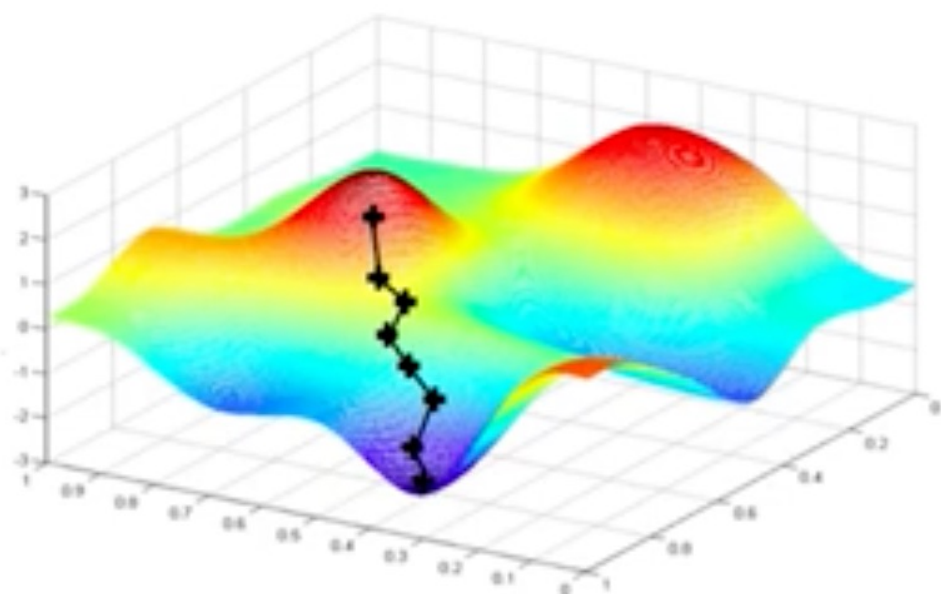
    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

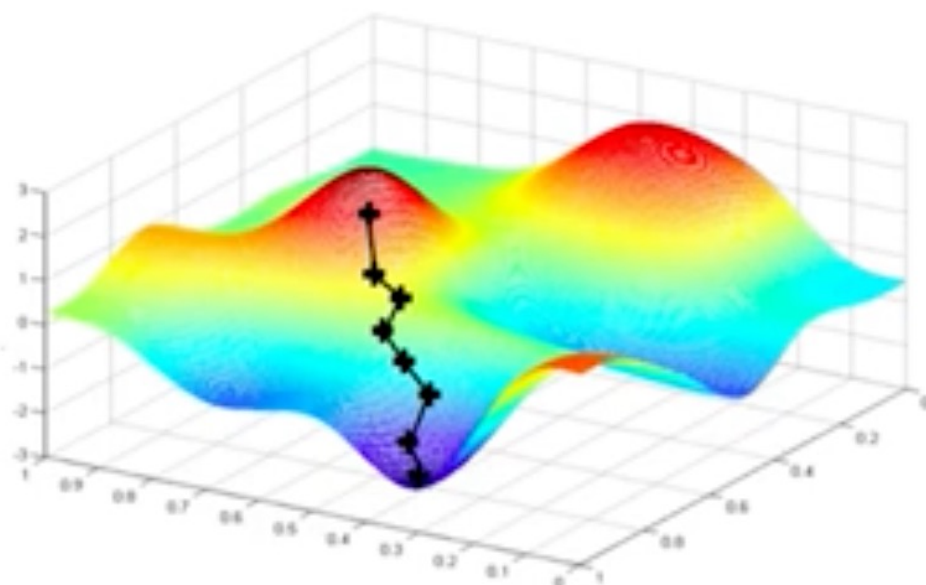
Can be very
computationally
intensive to compute!



Stochastic Gradient Descent

Algorithm

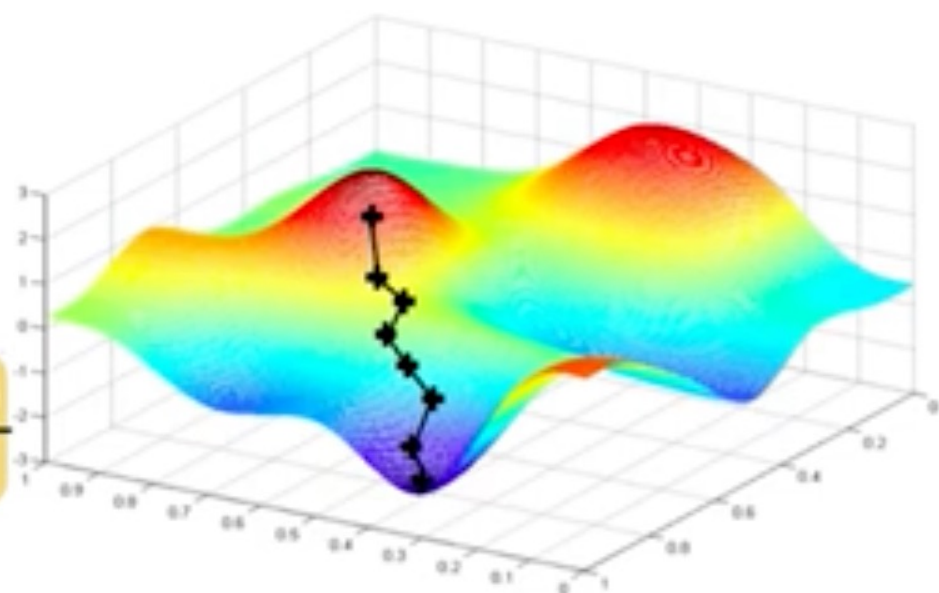
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(\mathbf{W})}{\partial \mathbf{W}}$
5. Update weights, $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial J(\mathbf{W})}{\partial \mathbf{W}}$
6. Return weights



Fast to compute and a much better
estimate of the true gradient!

Mini-batches while training

More accurate estimation of gradient

Smoother convergence

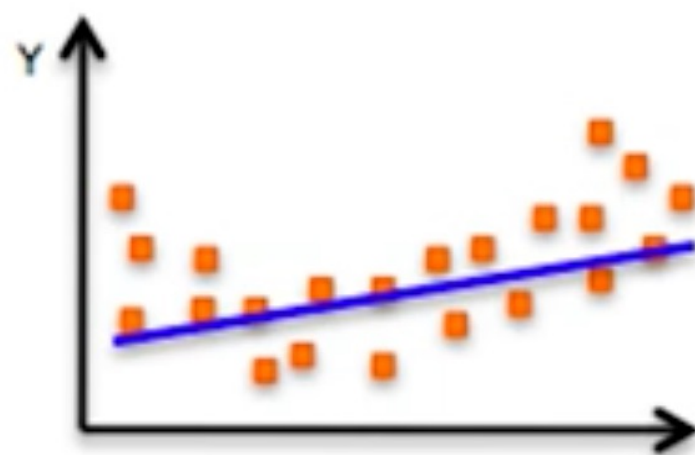
Allows for larger learning rates

Mini-batches lead to fast training!

Can parallelize computation + achieve significant speed increases on GPU's

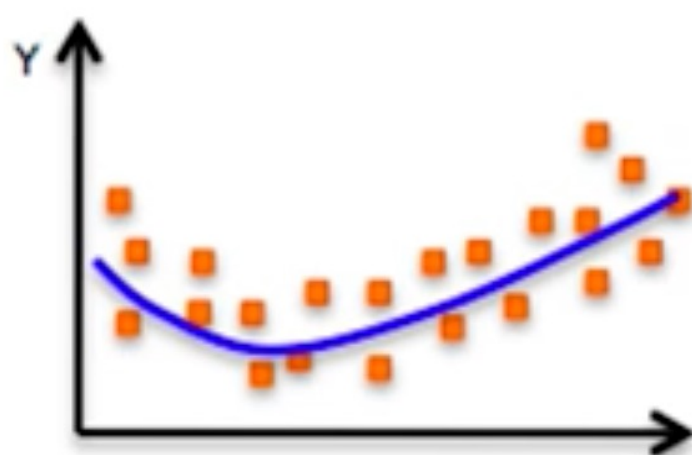
Neural Networks in Practice: Overfitting

The Problem of Overfitting

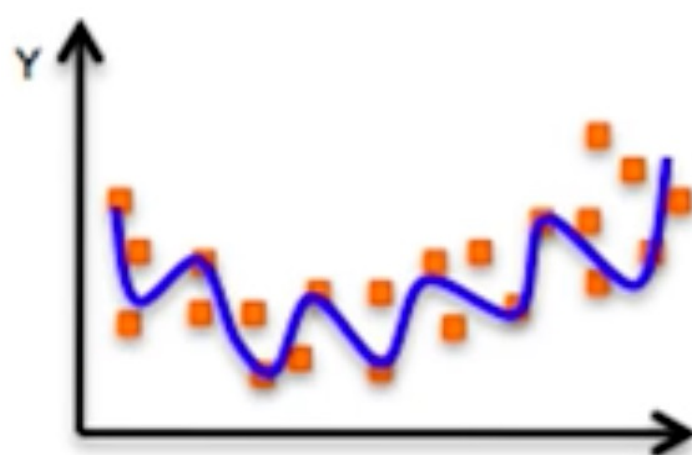


Underfitting

Model does not have capacity to fully learn the data



Ideal fit



Overfitting

Too complex, extra parameters, does not generalize well

Applying Neural Networks

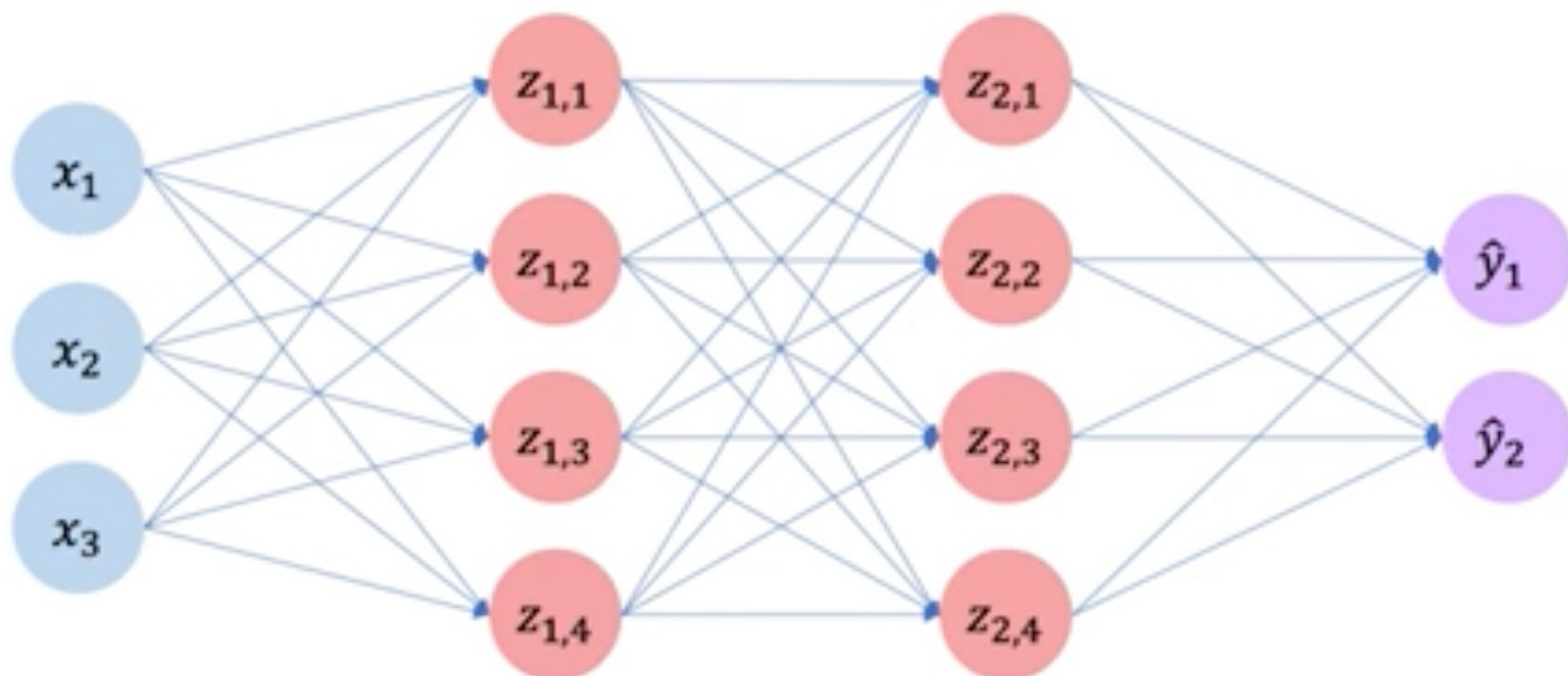
Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

Regularization I: Dropout

- During training, randomly set some activations to 0

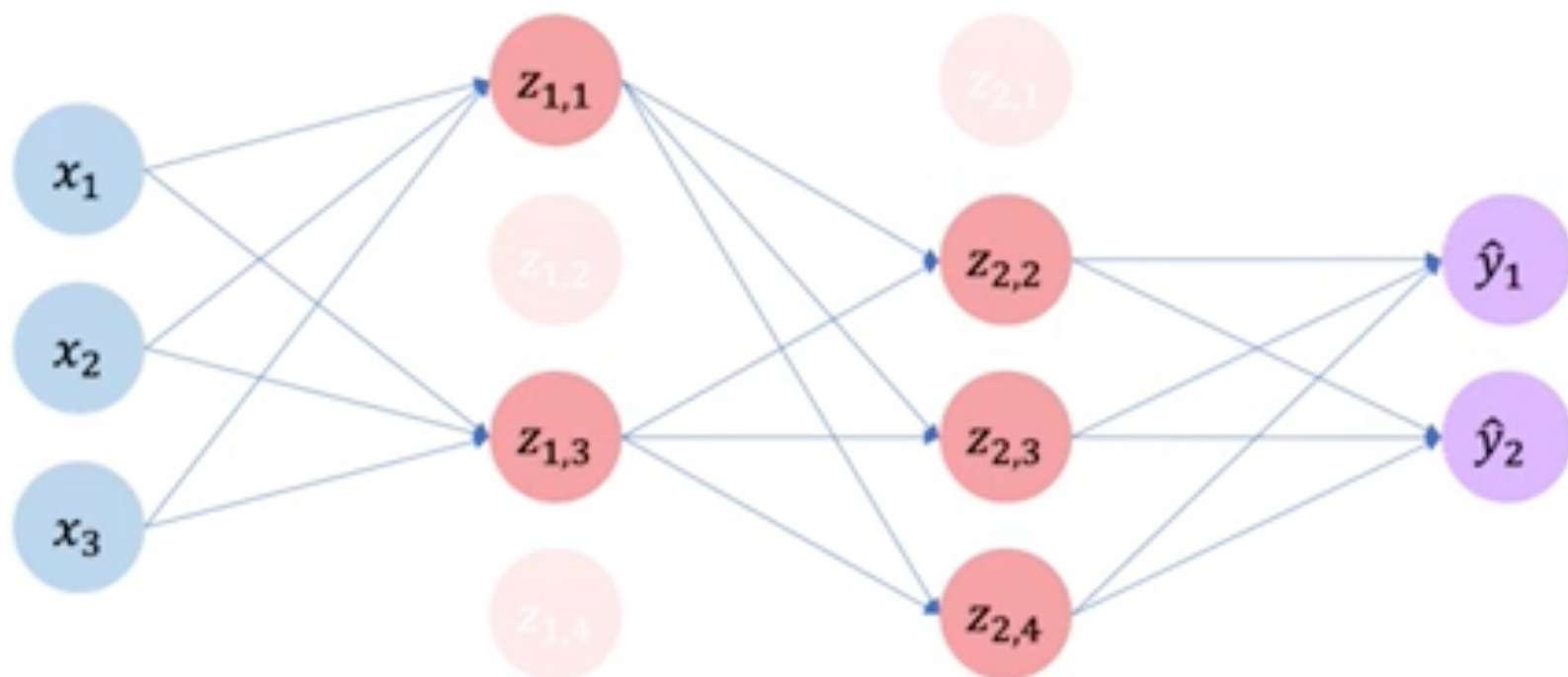


Regularization I: Dropout

- During training, randomly set some activations to 0
 - Typically 'drop' 50% of activations in layer
 - Forces network to not rely on any 1 node

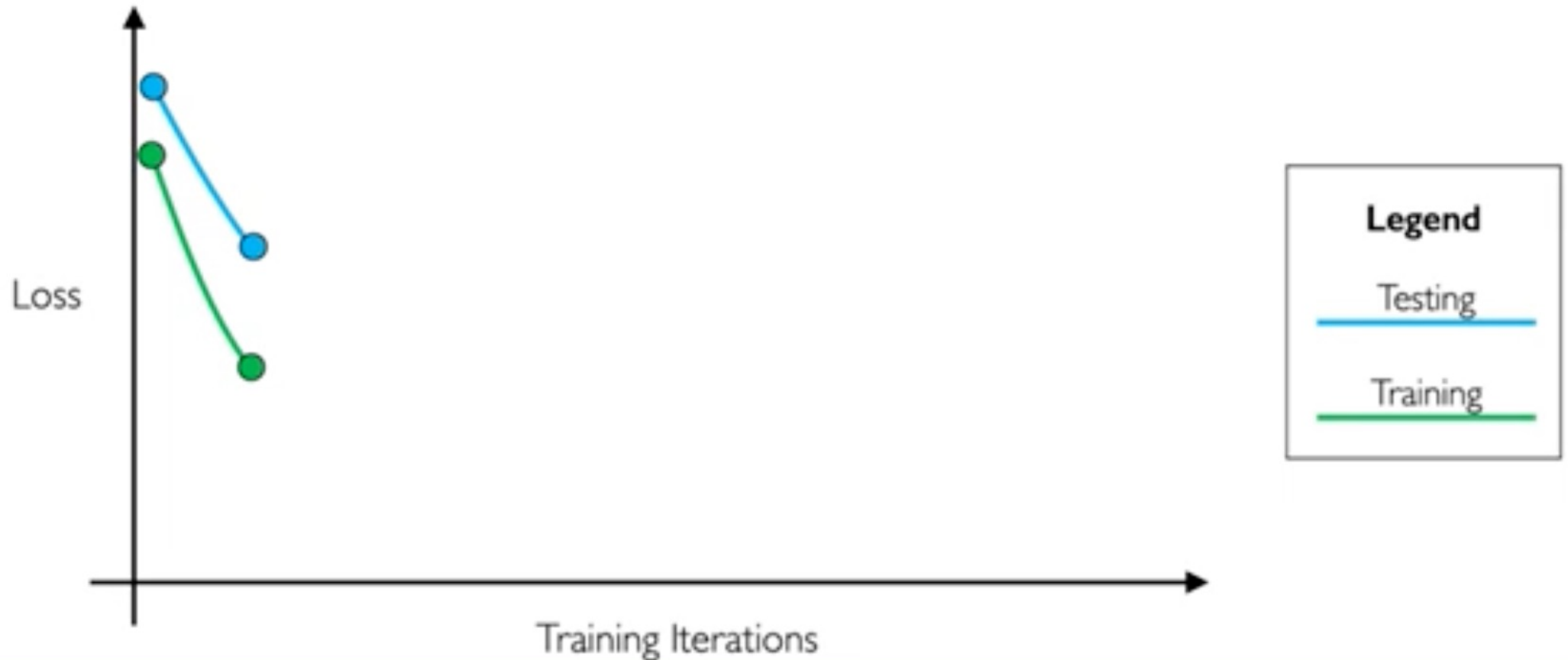


```
tf.keras.layers.Dropout(p=0.5)
```



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit



Regularization 2: Early Stopping

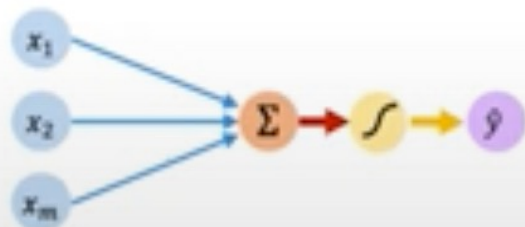
- Stop training before we have a chance to overfit



Core Foundation Review

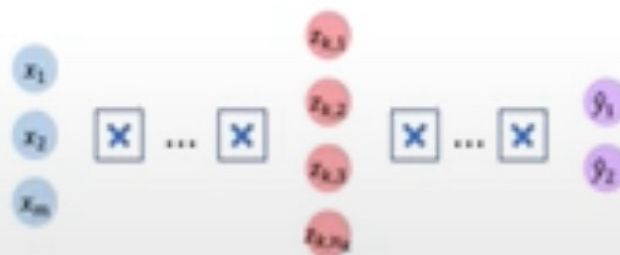
The Perceptron

- Structural building blocks
- Nonlinear activation functions



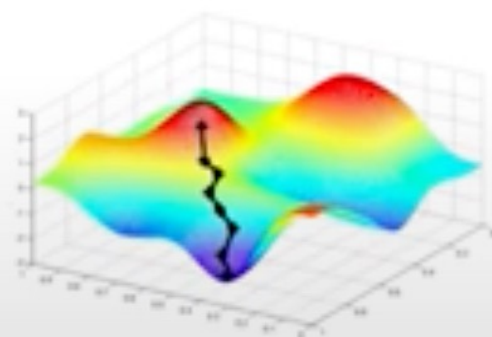
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



Classification Problem



Epoch
000,000

Learning rate
0.03

Activation
Tanh

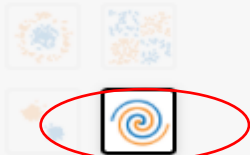
Regularization
None

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 50%

Noise: 0

Batch size: 10

REGENERATE

FEATURES

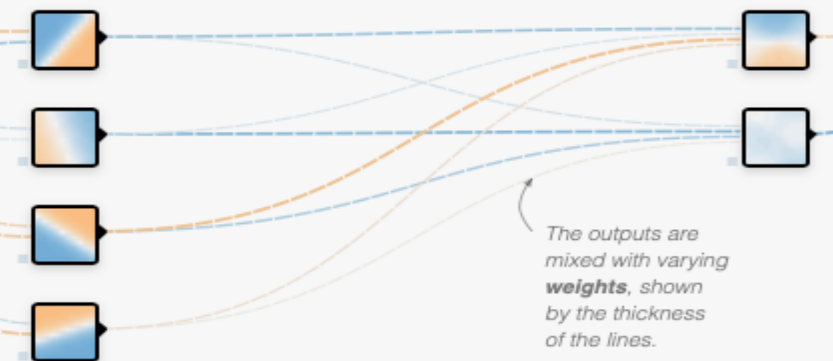
Which properties do you want to feed in?



+ - 2 HIDDEN LAYERS

+ -
4 neurons

+ -
2 neurons

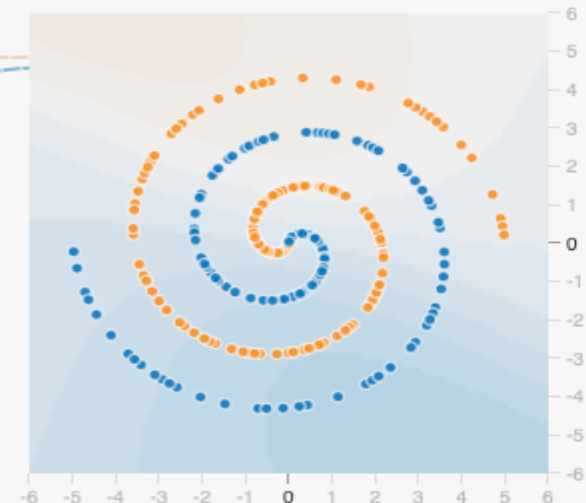


This is the output from one **neuron**. Hover to see it larger.

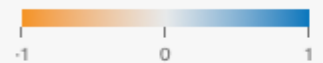
The outputs are mixed with varying **weights**, shown by the thickness of the lines.

OUTPUT

Test loss 0.483
Training loss 0.505



Colors shows data, neuron and weight values.



☐ Show test data

☐ Discretize output

Heuristic Training/Test Ratio 80/20



Epoch
000,569

Learning rate

0.03

Activation

ReLU

Regularization

L1

Regularization rate

0

Problem type

Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 80%

Noise: 0

Batch size: 10

REGENERATE

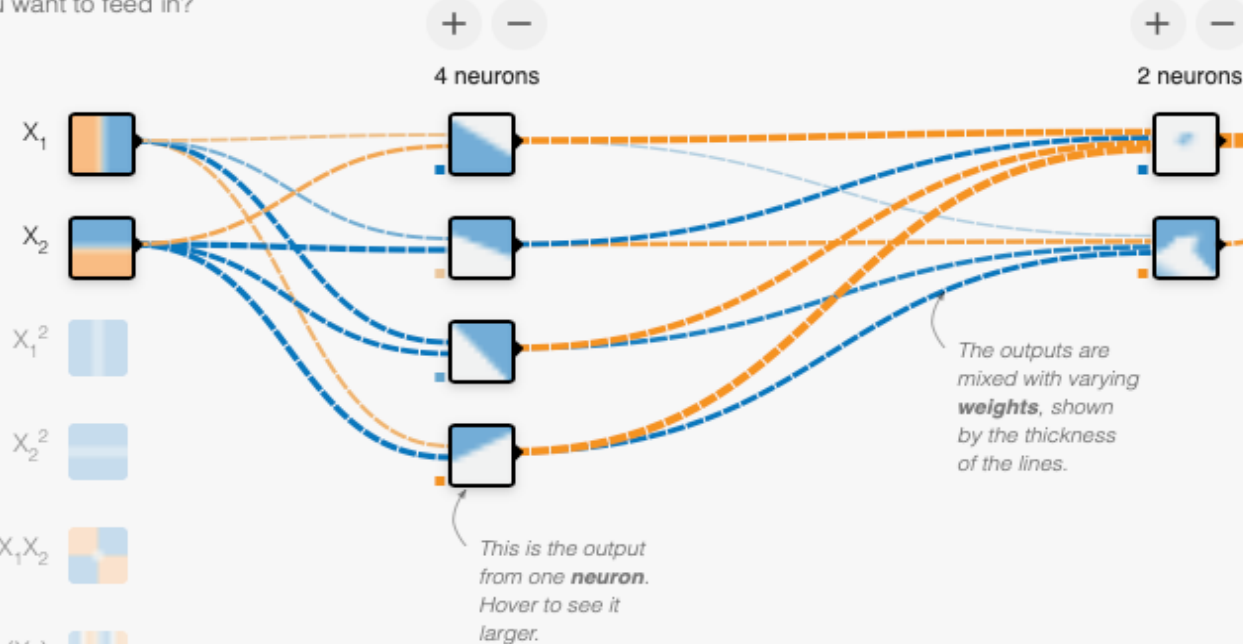
Regenerate data

FEATURES

Which properties do you want to feed in?

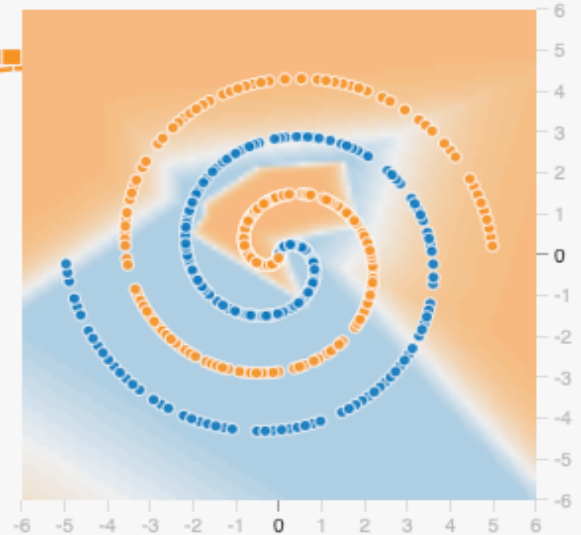


+ - 2 HIDDEN LAYERS

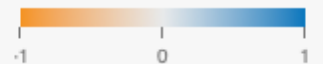


OUTPUT

Test loss 0.368
Training loss 0.325



Colors shows data, neuron and weight values.



☐ Show test data

☐ Discretize output

Heuristic Pareto 80/20



Epoch
000,569

Learning rate
0.03

Activation
ReLU

Regularization
L1

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 80%



Noise: 0



Batch size: 10



REGENERATE

Regenerate data

FEATURES

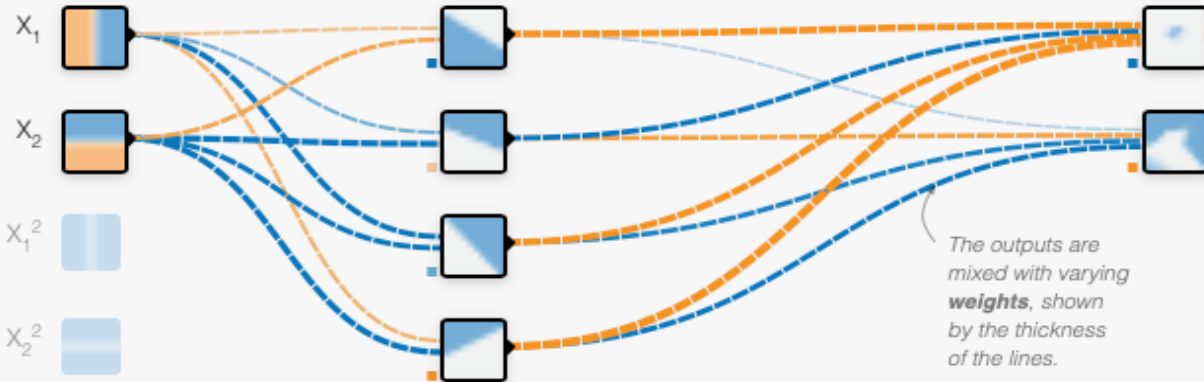
Which properties do you want to feed in?



+ - 2 HIDDEN LAYERS

+ -
4 neurons

+ -
2 neurons

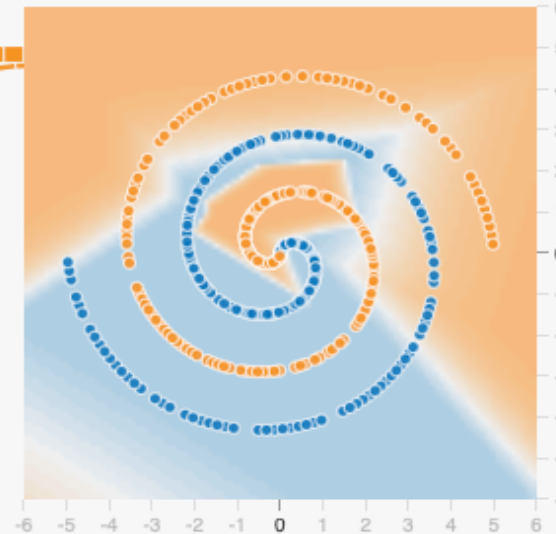


The outputs are mixed with varying **weights**, shown by the thickness of the lines.

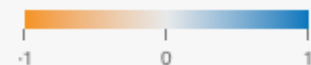
This is the output from one **neuron**. Hover to see it larger.

OUTPUT

Test loss 0.368
Training loss 0.325



Colors shows data, neuron and weight values.



☐ Show test data

☐ Discretize output

Training vs Test Errors



Epoch
003,010

Learning rate

0.03

Activation

ReLU

Regularization

L1

Regularization rate

0

Problem type

Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 80%

Noise: 0

Batch size: 10

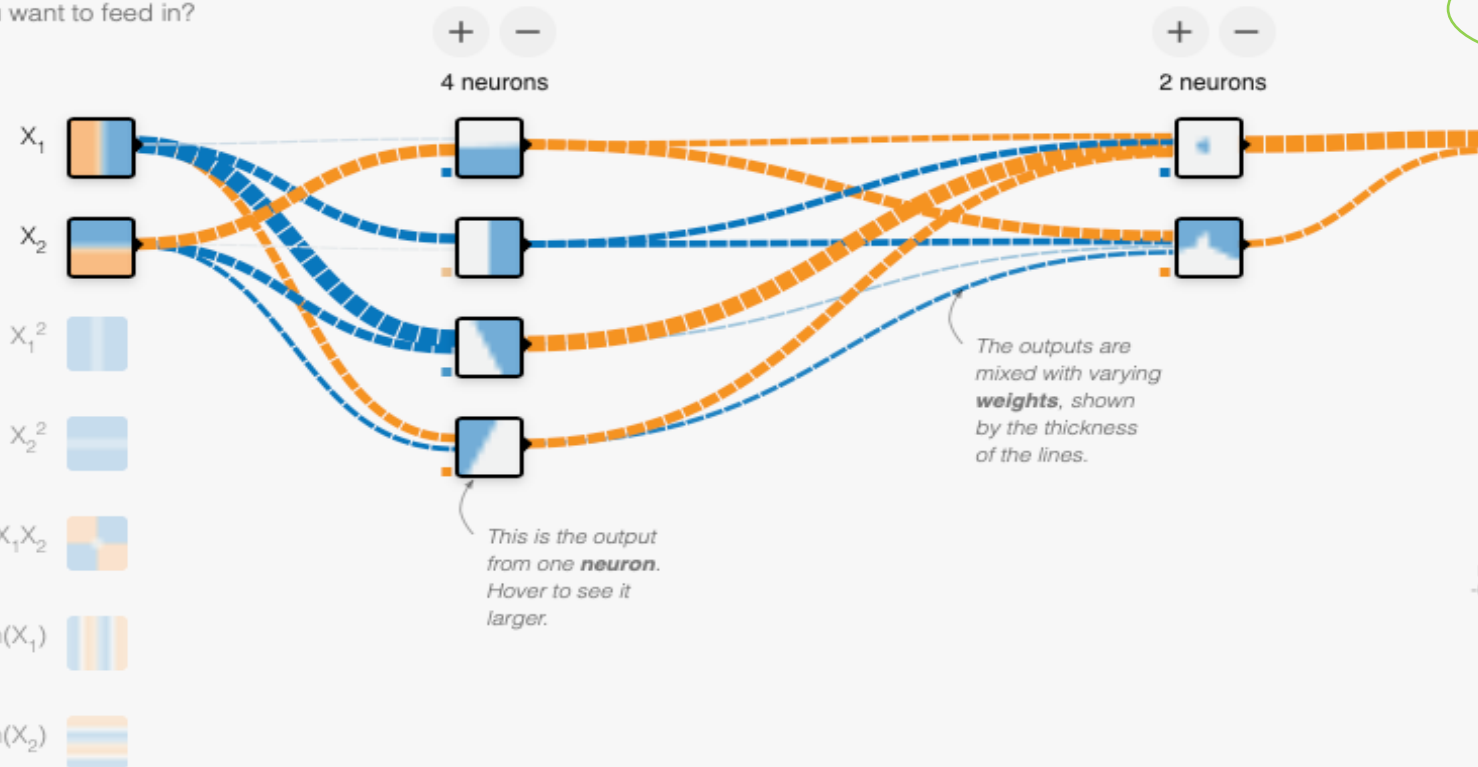
REGENERATE

FEATURES

Which properties do you want to feed in?

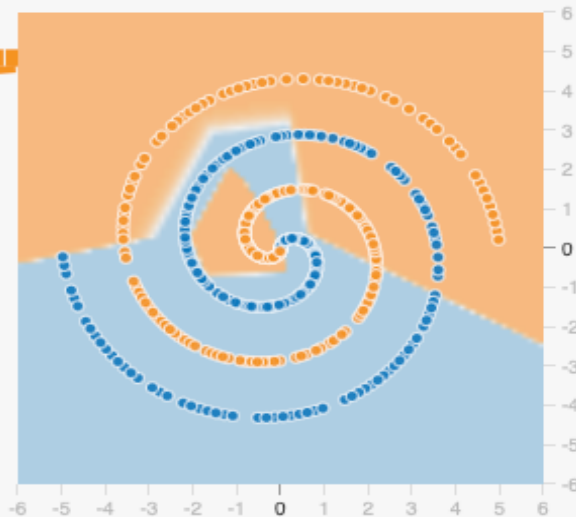


+ - 2 HIDDEN LAYERS

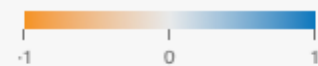


OUTPUT

Test loss 0.517
Training loss 0.377



Colors shows data, neuron and weight values.



☐ Show test data ☐ Discretize output

New Model with more input data



Epoch
003,220

Learning rate
0.03

Activation
ReLU

Regularization
L1

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 80%



Noise: 0



Batch size: 10



REGENERATE

FEATURES

Which properties do you want to feed in?

X_1



X_2



X_1^2



X_2^2



$X_1 X_2$



$\sin(X_1)$



$\sin(X_2)$



+ - 2 HIDDEN LAYERS

+ -
4 neurons

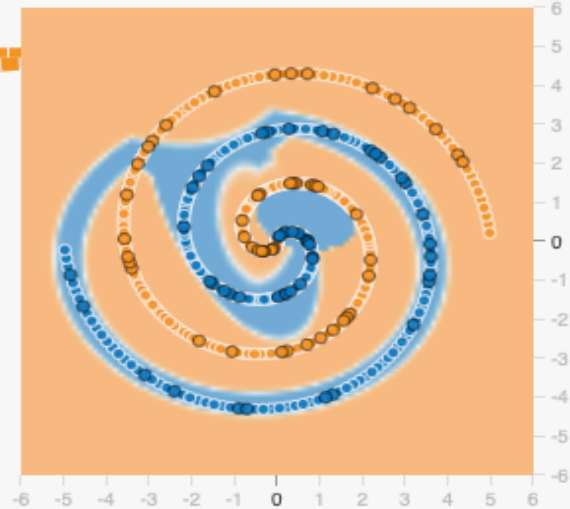
+ -
2 neurons

This is the output from one **neuron**.
Hover to see it larger.

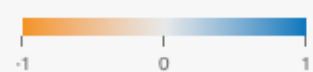
The outputs are mixed with varying **weights**, shown by the thickness of the lines.

OUTPUT

Test loss 0.038
Training loss 0.013



Colors shows data, neuron and weight values.



☒ Show test data ☐ Discretize output

New Model more Neurons



Epoch
001,948

Learning rate
0.03

Activation
ReLU

Regularization
L1

Regularization rate
0

Problem type
Classification

DATA

Which dataset do you want to use?



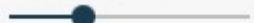
Ratio of training to test data: 80%



Noise: 0



Batch size: 10



REGENERATE

FEATURES

Which properties do you want to feed in?

X_1
 X_2
 X_1^2
 X_2^2
 $X_1 X_2$

$\sin(X_1)$

$\sin(X_2)$

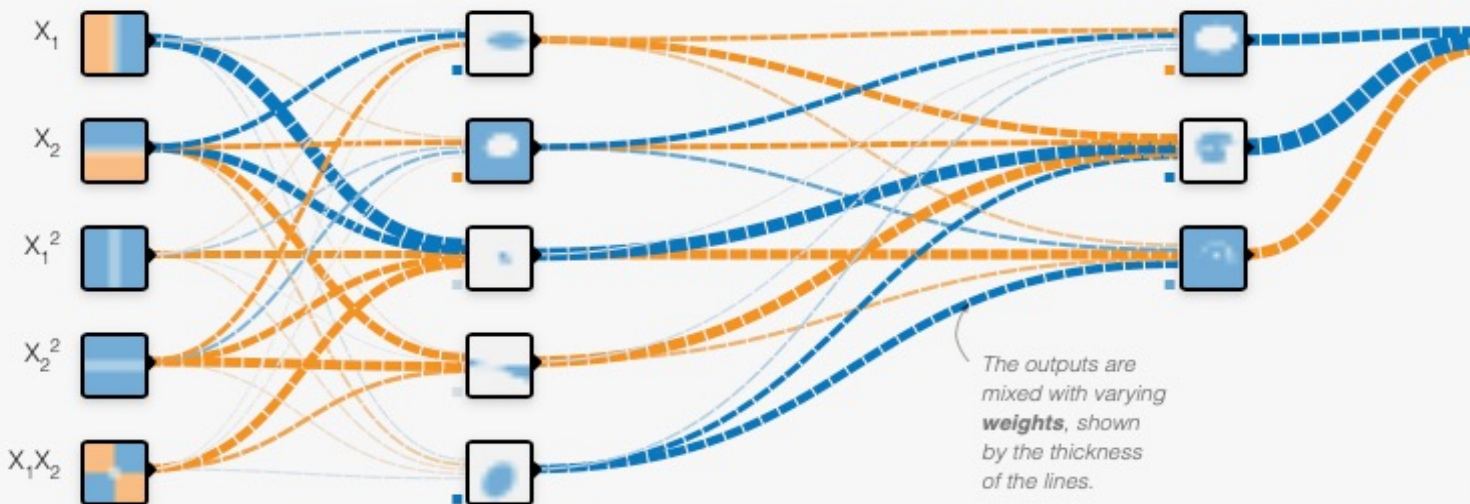
+ - 2 HIDDEN LAYERS

+ -

5 neurons

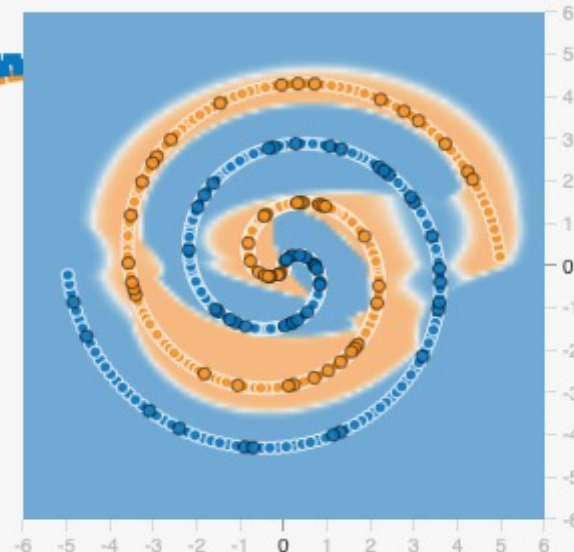
+ -

3 neurons

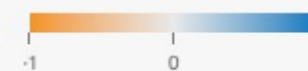


OUTPUT

Test loss 0.000
Training loss 0.007



Colors shows data, neuron and weight values.



☒ Show test data ☐ Discretize output

New Model with Noisy Data



Epoch
002,753

Learning rate

0.03

Activation

ReLU

Regularization

L1

Regularization rate

0

Problem type

Classification

DATA

Which dataset do you want to use?



Ratio of training to test data: 80%



Noise: 15



Batch size: 10



REGENERATE

FEATURES

Which properties do you want to feed in?

X_1
 X_2
 X_1^2
 X_2^2
 $X_1 X_2$

$\sin(X_1)$
 $\sin(X_2)$

+

-

2 HIDDEN LAYERS

+

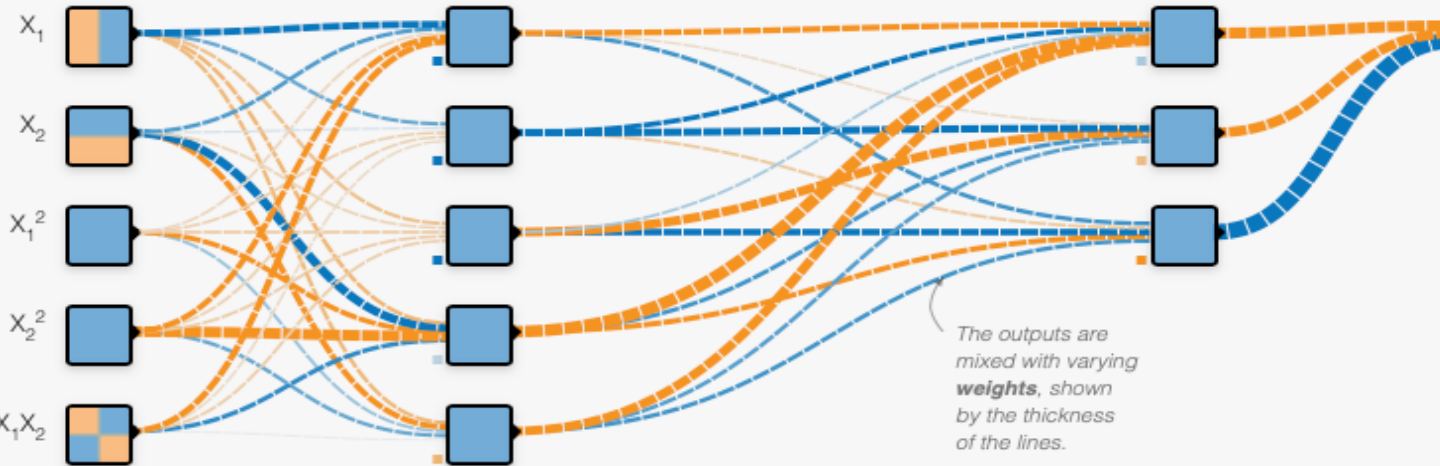
-

5 neurons

+

-

3 neurons

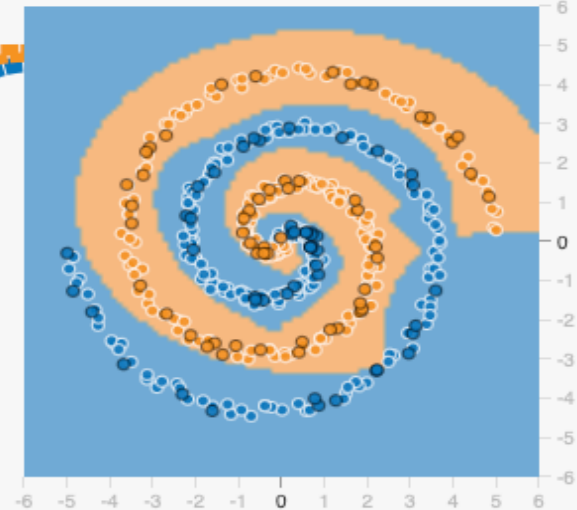


The outputs are mixed with varying **weights**, shown by the thickness of the lines.

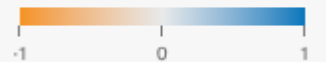
This is the output from one **neuron**. Hover to see it larger.

OUTPUT

Test loss 0.029
Training loss 0.004



Colors shows data, neuron and weight values.



☒ Show test data

☒ Discretize output

Image creating a Face for Siri or Alexa

Table of contents

- Copyright 2018 The TensorFlow Hub Authors.
- Generate Artificial Faces with CelebA Progressive GAN Model
- Optional prerequisites
- More models
- Setup
- Imports and function definitions
- Latent space interpolation
- Random vectors
- Finding closest vector in latent space
- Playing with the above example

Section

+ Code + Text Copy to Drive

```
def find_closest_latent_vector(initial_vector, num_optimization_steps,
                              steps_per_image):
    images = []
    losses = []

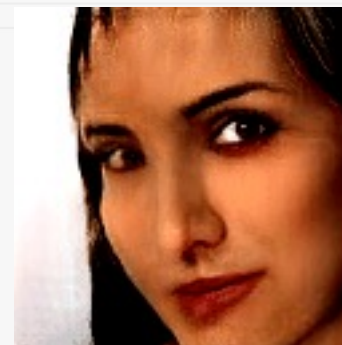
    vector = tf.Variable(initial_vector)
    optimizer = tf.optimizers.Adam(learning_rate=0.01)
    loss_fn = tf.losses.MeanSquaredError()

    for step in range(num_optimization_steps):
        if (step % 100)==0:
            print()
            print('.', end='')
            with tf.GradientTape() as tape:
                image = progan(vector.read_value())['default'][0]
                if (step % steps_per_image) == 0:
                    images.append(image.numpy())
                    target_image_difference = loss_fn(image, target_image[:, :, :3])
                    # The latent vectors were sampled from a normal distribution. We can get
                    # more realistic images if we regularize the length of the latent vector to
                    # the average length of vector from this distribution.
                    regularizer = tf.abs(tf.norm(vector) - np.sqrt(latent_dim))

                    loss = target_image_difference + regularizer
                    losses.append(loss.numpy())
                    grads = tape.gradient(loss, [vector])
                    optimizer.apply_gradients(zip(grads, [vector]))

            return images, losses

num_optimization_steps=100
steps_per_image=100
images, loss = find_closest_latent_vector(initial_vector, num_optimization_steps, steps_per_image)
```



```
[67] plt.plot(loss)
plt.ylim([0, max(plt.ylim())])
```

```
losses.py tf.losses.MeanSquaredError __init__.py X
4 """
5
6 import sys as _sys
7
8 from keras.losses import BinaryCrossentropy
9 from keras.losses import BinaryFocalCrossentropy
10 from keras.losses import CategoricalCrossentropy
11 from keras.losses import CategoricalHinge
12 from keras.losses import CosineSimilarity
13 from keras.losses import Hinge
14 from keras.losses import Huber
15 from keras.losses import KLDivergence
16 from keras.losses import LogCosh
17 from keras.losses import Loss
18 from keras.losses import MeanAbsoluteError
19 from keras.losses import MeanAbsolutePercentageError
20 from keras.losses import MeanSquaredError
21 from keras.losses import MeanSquaredLogarithmicError
22 from keras.losses import Poisson
23 from keras.losses import SparseCategoricalCrossentropy
24 from keras.losses import SquaredHinge
25 from keras.losses import binary_crossentropy
26 from keras.losses import binary_focal_crossentropy
27 from keras.losses import categorical_crossentropy
28 from keras.losses import categorical_hinge
29 from keras.losses import cosine_similarity
30 from keras.losses import deserialize
31 from keras.losses import get
32 from keras.losses import hinge
33 from keras.losses import huber
34 from keras.losses import kl_divergence
35 from keras.losses import kl_divergence as KLD
36 from keras.losses import kl_divergence as kld
37 from keras.losses import kl_divergence as kullback_leibler_divergence
38 from keras.losses import log_cosh
39 from keras.losses import log_cosh as logcosh
40 from keras.losses import mean_absolute_error
41 from keras.losses import mean_absolute_error as MAE
42 from keras.losses import mean_absolute_error as mae
43 from keras.losses import mean_absolute_percentage_error
44 from keras.losses import mean_absolute_percentage_error as MAPE
45 from keras.losses import mean_absolute_percentage_error as mape
46 from keras.losses import mean_squared_error
```

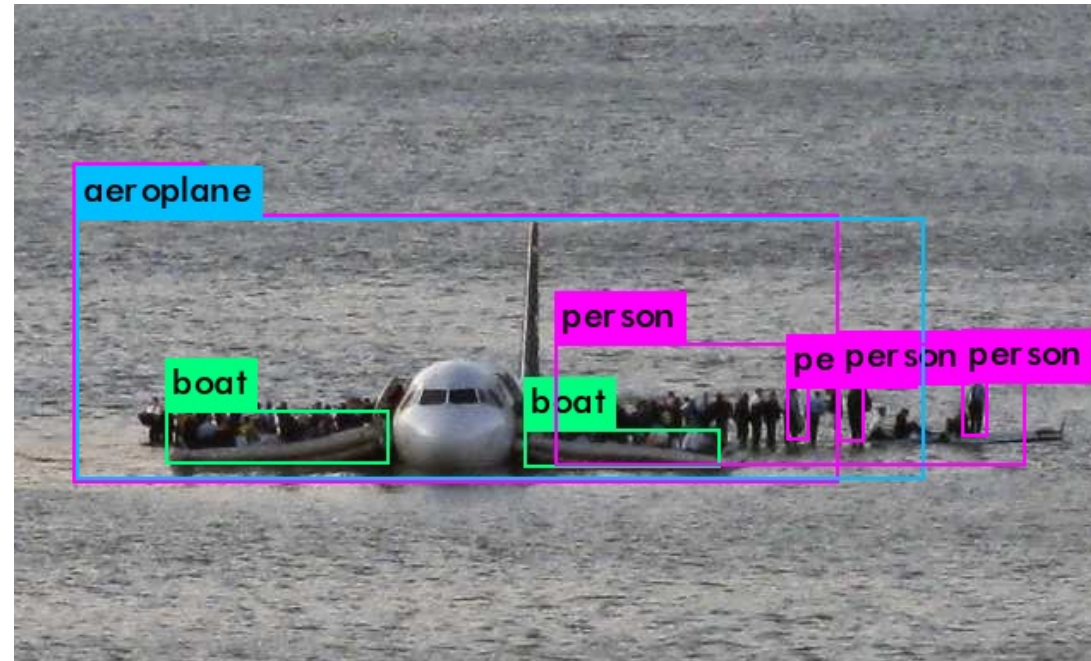
AI Contributions in the field of Aeronautics



The National
Academies of | SCIENCES
ENGINEERING
MEDICINE

AI Enabled Technologies

1. Goal?
2. Training data?
3. Model?
4. Accuracy?



AI Workshop Sept 26, 2019

Neil A. Armstrong Flight Research Center

Ricardo A. Arteaga

Overview

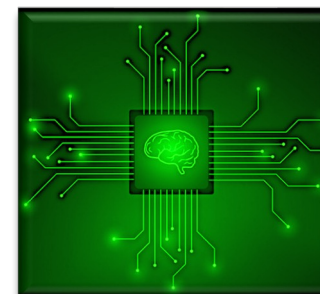


Mission Objectives: Explore use of a sUAS capability at an aircraft crash scene to:

- ☐ Assist Crash and Rescue Efforts
 - ☐ Define Crash Area –security and public safety
 - ☐ Assess hazards before entry
 - ☐ Locate personnel
- ☐ Video document the site
 - ☐ Identify/locate objects
 - ☐ GPS tag/inventory individual objects
 - ☐ Enable management and EOC efforts
 - ☐ Detect/Classify the Black Box
- ☐ Stream video to AFRC Edwards location



**20 Megapixel
Camera**

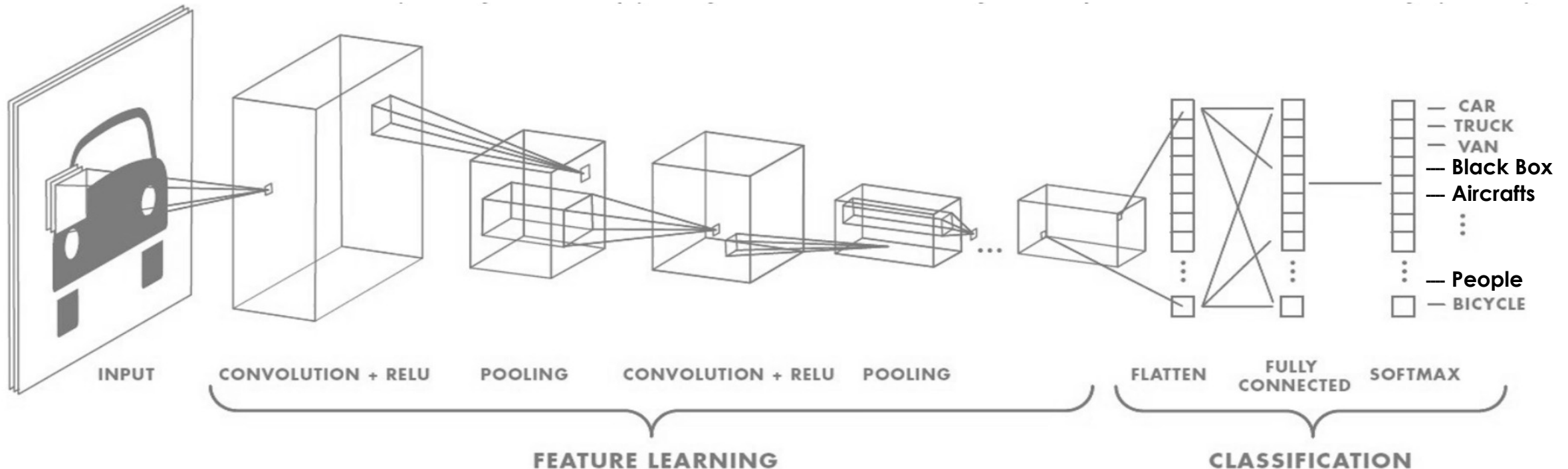


Artificial Intelligence

Deep Neural Network Architectures



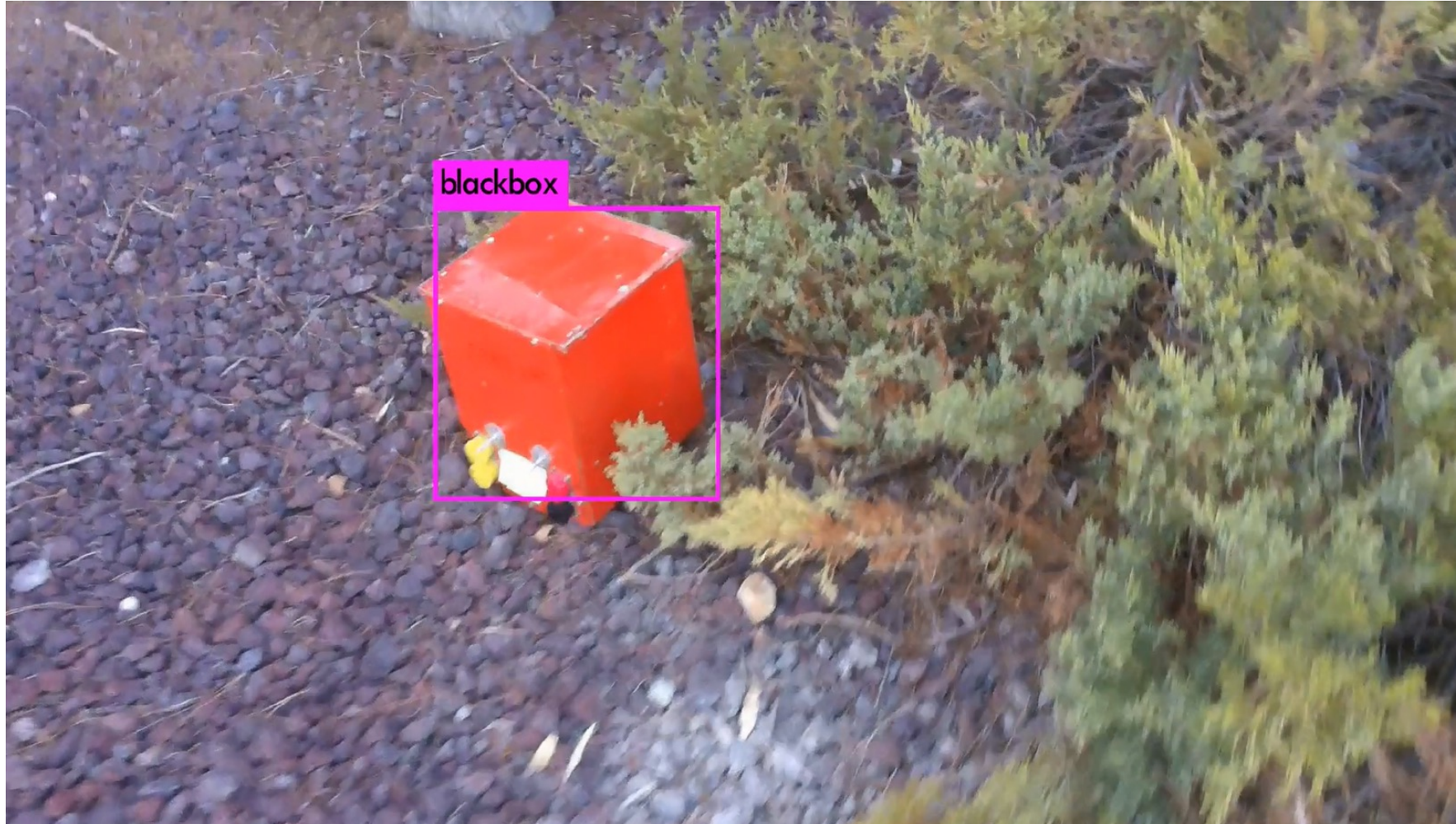
- **Goal:** Object Detection and Classification Box Black
- **Training Data:** Over 120 Black Box images (2D) on the internet
- **CNN Model:** Yolo detection network has 24 convolutional layers deep followed by 2 fully connected layers.



AI CNN Detection Accuracy



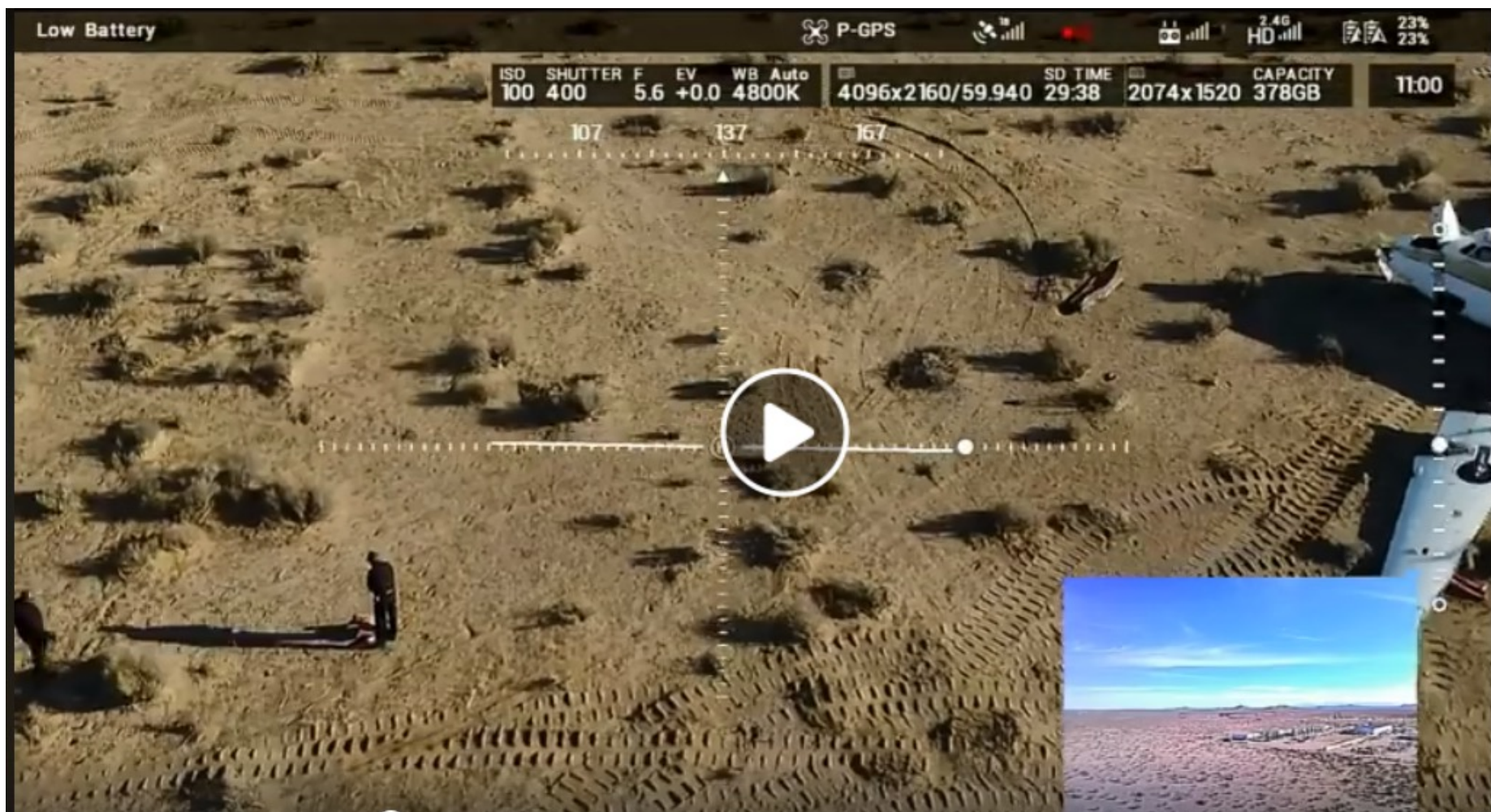
- ❑ **Neural Networks for Object Detection and Tracking (speed, accuracy, and efficiency)**



- ❑ **Results: YOLOv2 predicts (96%) bounding box on 1080P resolution at 30 FPS video at low altitudes**

Aircraft Crash Mishap Exercise

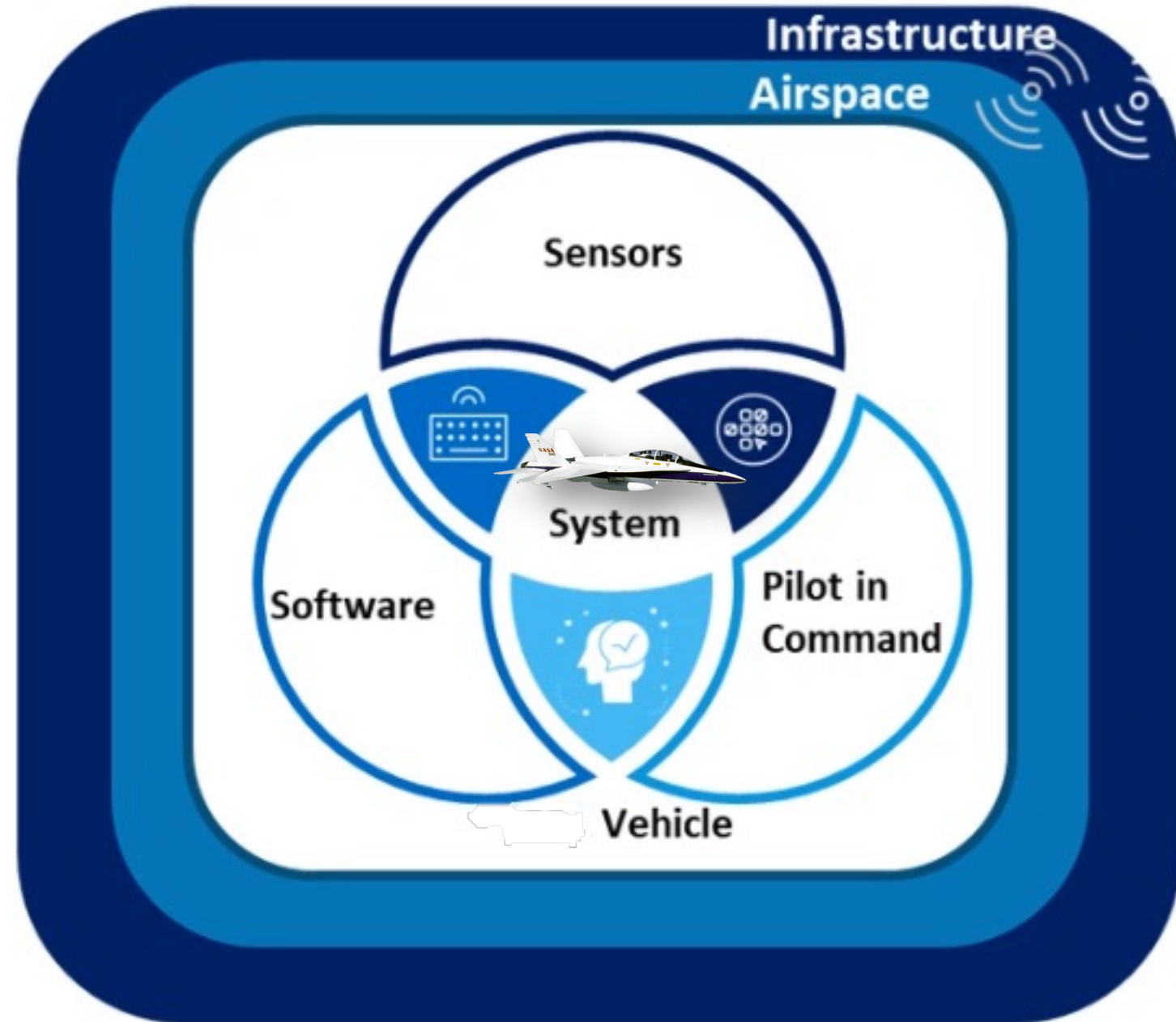
- ❑ Mission Economics based on the hazardous conditions and limited time



- ❑ Results: Video Uploaded 2.5 Mbps at 30 FPS ~ 3 second lag on Live Stream Video on MiFi

AI Contributions in the field of Aeronautics

- 1.Goal?
- 2.Training data?
- 3.Model?
- 4.Accuracy?



Reference Books:
AI A Modern Approach

