What is deep learning?:

Artificial intelligence (any technique that mimics human behavior)

→ Machine learning (ability to learn without being programmed)

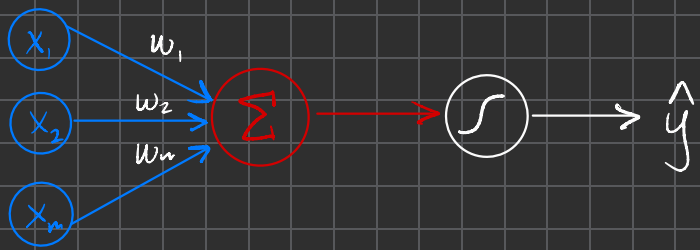
→ Deep learning (extract patterns from data using neural networks)

TL/DR: Teaching computers to learn a task directly from raw data

Why deep learning and why now?

- Hand engineered features are time consuming, brittle, and not scalable
- Now with Big Data, it is easier than ever to train a model to learn
- Also new hardware and software

Perceptron:



Inputs Weights Sum Non-linearity Output

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right) \text{ where } w_0 \text{ is a bias term}$$

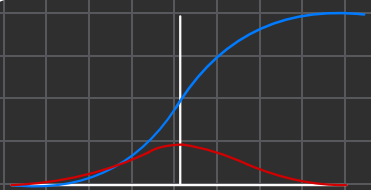
$$\hat{y} = g(w_0 + X^T W) \text{ where } X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Examples of non-linear functions

Sigmoid

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

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



Hyperbolic Tangent

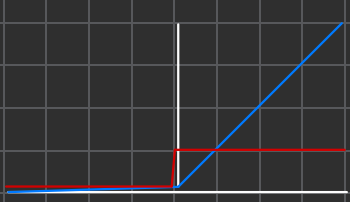
$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

Rectified Linear Unit (ReLU)

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

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



Code snippets:

Sigmoid: `tf.math.sigmoid(z)`

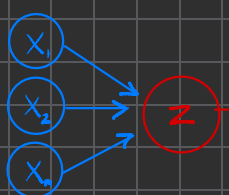
Hyperbolic Tangent: `tf.math.tanh(z)`

ReLU: `tf.nn.relu(z)`

Non-linearity functions help to deal with non-linear datasets, such as scatterplots.

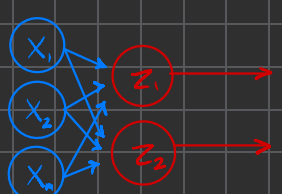
Simplifying the perceptron:

1 neuron



$$y = g(z) \qquad z = w_0 + \sum_{j=1}^{m} x_j w_j$$

2 neurons



$$y_1 = g(z_1)$$
$$y_2 = g(z_2)$$

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

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])
```

Weight matrix

Bias

```
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
```

Matrix multiply and add weights

$$\left(w_0 + \sum_{j=1}^{m} x_j w_j\right)$$

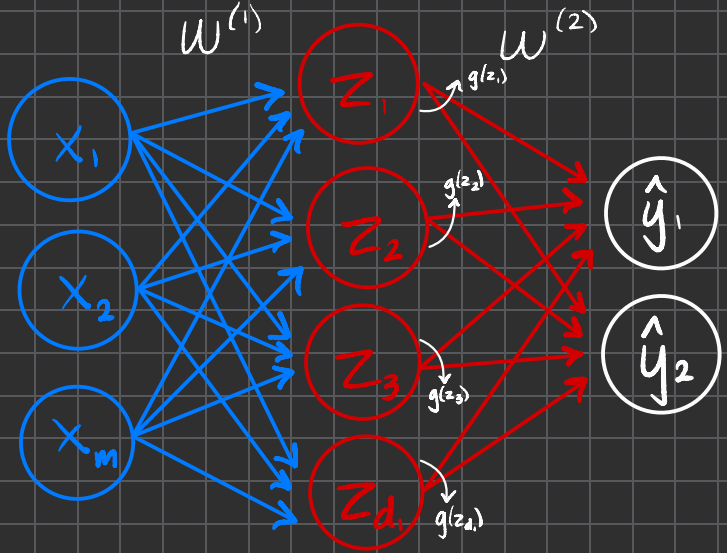
Apply non-linearity

Return result

Simplifies to:

```
import tensorflow as tf

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

Single Layer Neural Network:

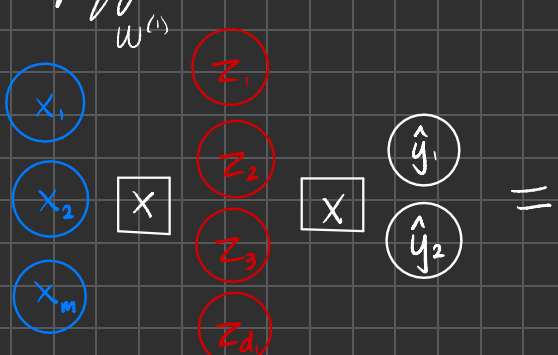


Inputs Hidden Layer Outputs

Focus on just 1 neuron:

$$z_2 = w^{(1)}_{0,2} + \sum_{j=1}^{m} x_j w^{(1)}_{j,2} = w^{(1)}_{0,2} + x_1 w^{(1)}_{1,2} + x_2 w^{(1)}_{2,2} + x_m w^{(1)}_{m,2}$$

Simplify:




```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```

Applying Neural Networks

Quantifying Loss:

$$L(\underbrace{f(x^{(i)}; W)}, \underbrace{y^{(i)}})$$

Predicted Value Actual Value

Empirical Loss = $J(W) = \frac{1}{n}\sum_{i=1}^{n} L(f(x^{(i)}; W), y^{(i)})$

(Also known as objective function, cost function, or empirical risk)

Binary Cross Entropy Loss

$$J(W) = -\frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log(f(x^{(i)}; W)) + (1 - y^{(i)}) \log(1 - f(x^{(i)}; W))$$

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

Mean Squared Error Loss

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} (y^{(i)} - f(x^{(i)}; W))^2$$

Loss Optimization

We want to find the weights that minimize loss

$$W^* = \arg\min_W \frac{1}{n} \sum_{i=1}^{n} L\left(f(x^{(i)}; W), y^{(i)}\right)$$

$$W^* = \arg\min_W J(W)$$

Steps to minimize loss

1.) Pick a random W

2.) Compute $\frac{\partial J}{\partial W}$ at that W

3.) Take 1 step in the opposite direction of the gradient

4.) Update weights

5.) Repeat

This is called <u>gradient descent</u>

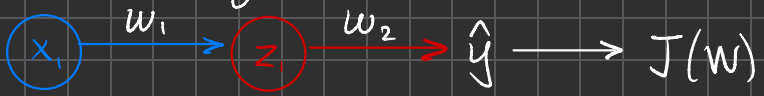
The step is also known as a learning rate

Loss optimization (cont.)

```python
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
```

How do you do this?

Back Propogation:



$$\frac{\partial J(w)}{\partial w_2} = \frac{\partial J(w)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

(Let's chain rule!)

$$\frac{\partial J(w)}{\partial w_1} = \frac{\partial J(w)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1} = \frac{\partial J(w)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

Repeat this for every weight in the network using gradients from later layers

Loss Optimization (Cont.)

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

How can we set the learning rate, η?

Small learning rates converge slowly and can get stuck in local minimums

Large learning rates cause overshooting and instability.

Stable learning rates avoid this

You can

1.) Try a bunch of learning rates

2.) Build an adaptive learning rate algorithm

a.) Whole community on this

Here's where we are now

```
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)))
```

Batching Data into mini batches

Computing a gradient across an entire dataset is to computationally expensive

What if we do it for a single example

- Very fast!
- Very noisy

To solve, compute $\frac{\partial J(w)}{\partial w}$ over a medium batch size (10's or 100's of examples)

Can use this to parallelize computation!

Overfitting:

- Underfitting means the model does not have capacity to fully learn data.
- Overfitting means that high complexity causes wrong assumptions to be made
- An ideal fit will learn data, but not make assumptions

Regularization:

- Technique to reduce likelyhood of overfitting
- Dropout:
  - During training, randomly set some activations to 0
  - Typically "drops" 50% of activators on any one layer.
- Early Stopping
  - Monitor training with excess data