

The Units - Neurons

ITI106 Foundations of Deep Learning

Neurons

- Deep learning is based on *neural networks* and that is inspired by the brain.
- The brain is incredibly powerful and complicated but we have only limited ideas on how it actually works.
 - One of early reasons for developing Neural Network was to understand how the brain works.
- At the most basic level, the brain works based on **neurons**.
 - According to many estimations, there are about 100 billions neurons in a human brain.
- We can view neuron as a processing unit.
 - When electrical potential on the membrane of the neuron cell is **high enough**, a neuron fires a signal, affecting other neurons connected via **synapses**.



Hebb's Rule

- **Synaptic Plasticity** - If two neurons both respond to something, they should be connected.
 - If two neurons are active at the same time, this strengthens the bond between the two.
 - If two neurons are never active at the same time, the connection weakens.
- **Example: Classical conditioning**
 - Provide food for a dog and blow a whistle at the same time.
 - Connections between neurons stimulated by food and those for hearing a whistle becomes stronger.
 - Very soon the dog will associate whistle with hunger and food.



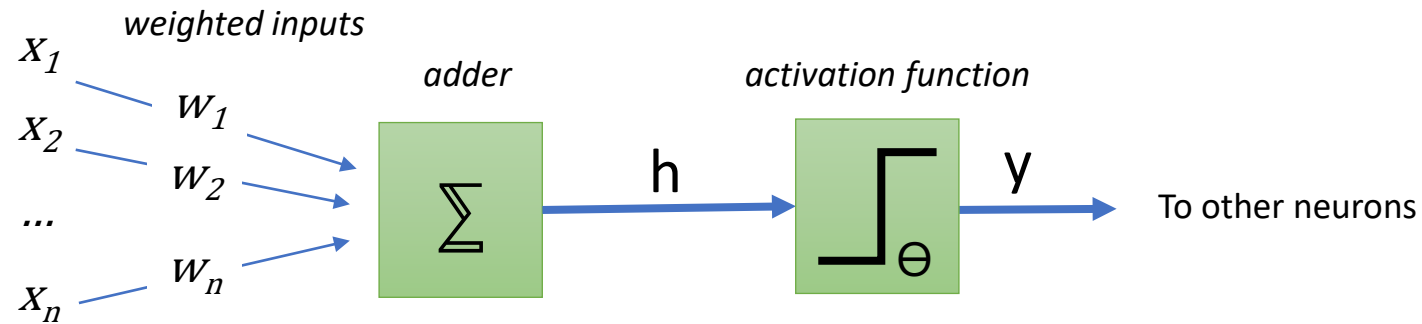
MP (McCulloch and Pitts) Neurons

- In 1943, McCulloch and Pitts proposed a mathematical model of the neuron in a paper *“A Logical Calculus of the Ideas Immanent in Nervous Activity”*.
- In the paper, neuron was described as a simple logic gate with a **binary** output.
 - Multiple signals arrives at the **inputs** (dendrites) of a neuron.
 - The signals accumulate until a certain **threshold** is reached.
 - Once over the threshold, the neuron fires a signal.
- We will describe this in a mathematical model which is usually referred to as the MP or MCP neuron.

MP (McCulloch and Pitts) Neuron Model

The model consists of 3 main components:

1. A set of **weighted inputs** (strength of the synapses).
2. An adder that **sums the input signals** (cell membrane)
3. An **activation function** (condition for firing, decides if neuron should fire or not)



**The weights can be positive (excitatory) or negative (inhibitory). Negative weights make the neuron less likely to fire.*

$$h = \sum_{i=1}^n w_i x_i$$

$$y = \begin{cases} 1, & \text{if } h > \Theta \\ 0, & \text{if } h < \Theta \end{cases}$$

**This particular activation function is called the Heaviside step function.
 Θ = threshold*

MP (McCulloch and Pitts) Neurons

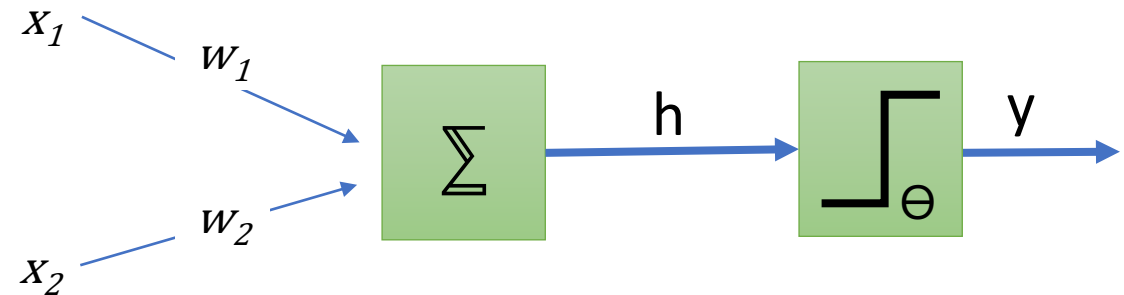
- The McCulloch and Pitts model is a very simplistic version of how a real neuron works and is actually not very realistic.
- Although too simple compared to the real neuron, we can build a huge **network** of it to actually perform complex computations like a computer.
- It can be used for executing Boolean functions like AND, OR, NOR, NOT. (see next slide)

MP Neuron - Boolean Functions

- AND Operation
 - Set the weights (w_1, w_2) to be 1
 - Set $\Theta \geq 2$
- OR Operation
 - Set the weights to be 1
 - Set $\Theta \geq 1$
- NOT Operation
 - Set weight to be 0 and -1
 - Set $\Theta = 0$
- NOR Operation
 - Set weight to be -1 and -1
 - Set $\Theta = 0$
- XOR
 - Requires more than 1 neurons
 - Can be done using AND, NOT and OR over 2 layers
 - $X_1 \text{ XOR } X_2 = (X_1 \text{ AND NOT } X_2) \text{ OR } (X_2 \text{ AND NOT } X_1)$

X1	X2	AND	OR	NOR	XOR
0	0	0	0	1	0
1	0	0	1	0	1
0	1	0	1	0	1
1	1	1	1	0	0

X1	NOT
0	1
1	0

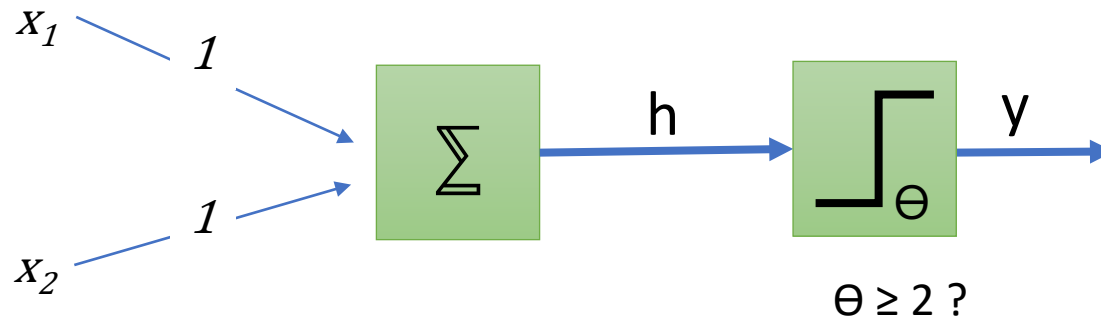


MP Neuron - Boolean Functions (Example)

AND Operation

Set the weights (w_1, w_2) to be 1

Set $\theta \geq 2$



x1	x2	AND
0	0	0
1	0	0
0	1	0
1	1	1

x1	x2	h	y
0	0	0	0
1	0	1	0
0	1	1	0
1	1	2	1

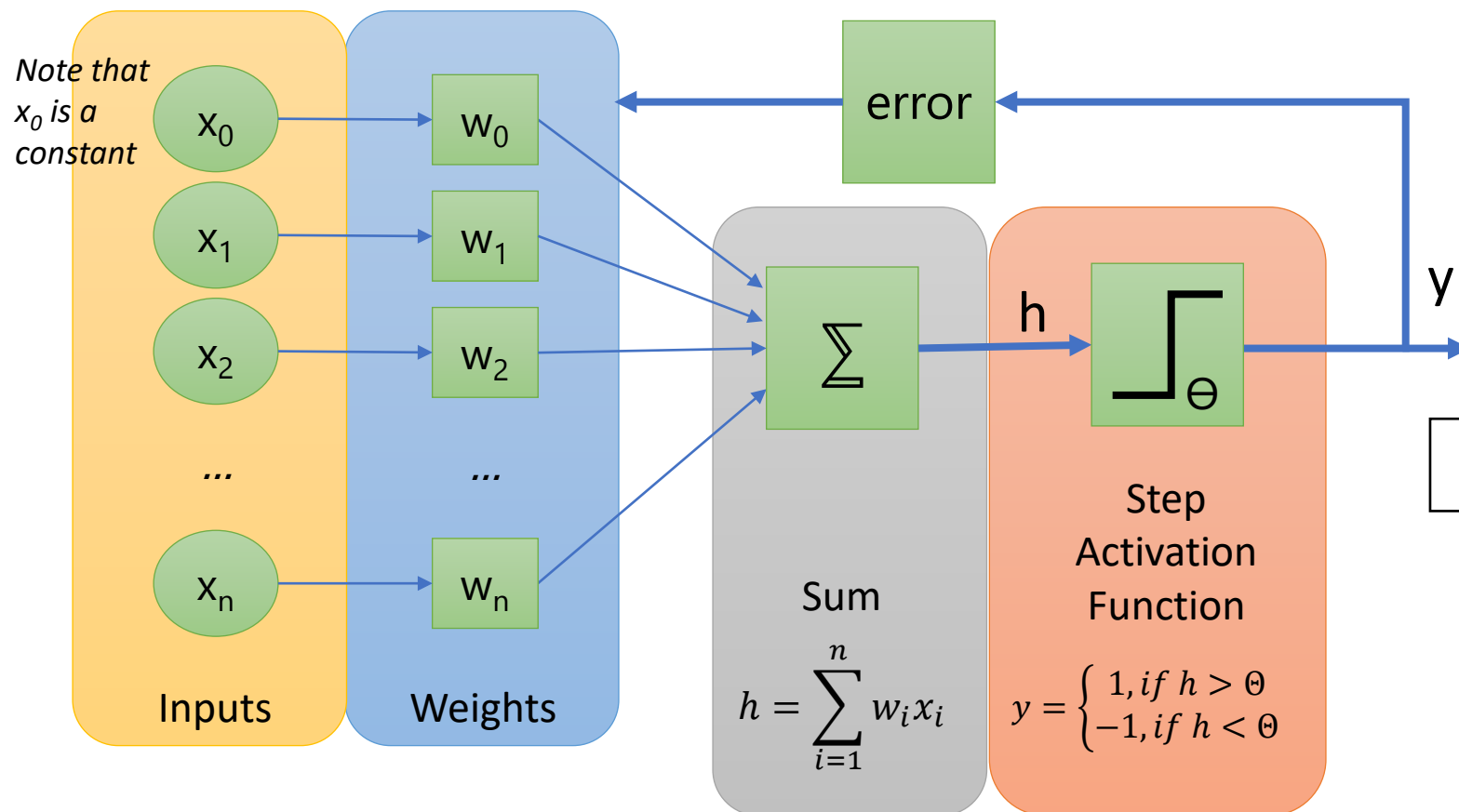
MP (McCulloch and Pitts) Neurons

- The MP Neuron model is very simple and not very useful.
- Good for understanding the very basic unit of a neural network.
- Limitations:
 - A single binary output
 - Weight and threshold values are manually fixed by human.
 - There is actually **no machine learning** as it requires humans to modify and set the weights and threshold. *Human is still doing the learning.*

Perceptron

- In 1957, Frank Rosenblatt proposed an experimental neural net model called the **Perceptron** algorithm.
- It is an improvement over MP neurons and is the first model to **learn** the weights based on supplied data.
- Instead of humans manually setting the weights, the Perceptron algorithm will **learn the optimal weights** based on data provided.
- Perceptron are split into **single-layer** and **multiple-layer** Perceptron.
- Multiple-layer Perceptron is more powerfully and can handle more complex cases.

Perceptron Algorithm



Learning Steps

1. Initialize the weights to random numbers
2. For each x_i , compute output value $y_{predict}$ and adjust the weights using the perceptron learning rule

$$\Delta w_i = r \times (y_{real} - y_{predict}) \times x_i$$

Change in weight

Error in prediction

Rate of change

Original input

3. Repeat adjustments of the weights until they stabilize (converged)

*All change in weights are computed and applied before another prediction is made.

Perceptron

- How does it work?

$$\Delta w_i = r \times (y_{real} - y_{predict}) \times x_i$$

If Prediction is Correct

$$y_{real} = y_{predict} = 1 \text{ or } y_{real} = y_{predict} = -1$$

$$\Delta w = r \times (1 - 1) \times x_i = 0$$

Or

$$\Delta w = r \times (-1 - (-1)) \times x_i = 0$$

The weight does not change

If Prediction is Wrong

$$y_{real} = 1, y_{predict} = -1 \text{ or } y_{real} = -1, y_{predict} = 1$$

$$\Delta w = r \times (1 - (-1)) \times x_i = 2 \times r \times x_i$$


or

$$\Delta w = r \times (-1 - (1)) \times x_i = -2 \times r \times x_i$$

The weight changes in the positive or negative direction.

Perceptron

- Note that in the previous example, we multiply the error in prediction by the input value x_i .
- Multiplication by the input value will ensure that the weight adjustment is **proportional to the input value**.
 - Otherwise, inputs with large values will not be able to learn effectively as the weight adjustment may be too small.


$$\Delta w_i = r \times (y_{real} - y_{predict}) \times x_i$$

Learning Rate

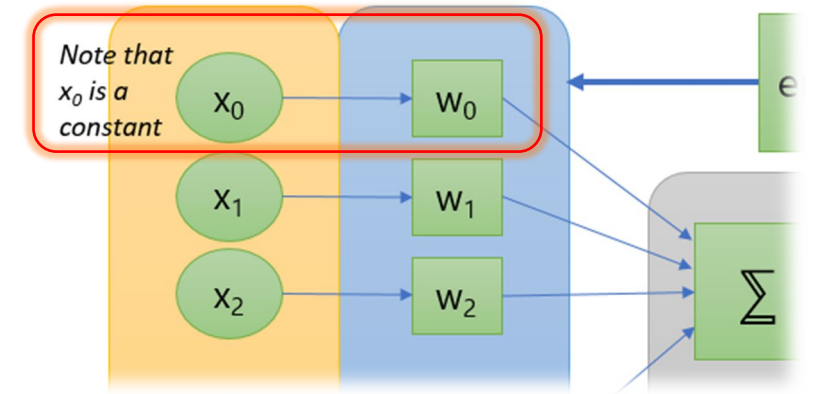
- The learning rate r is a number between 0.0 and 1.0.
- It allows us to control the amount of change in the weights, thereby controlling the pace of learning.
 - Too large a value tends to cause the weight to fluctuate widely and become unstable. It may never settle down (i.e. converge).
 - Too small a value will cause only small changes in weights and the learning will be very slow. Good thing is that it will be more stable and resistant to noise or errors in data.
- A **typical value** for r is between 0.1 and 0.4.



$$\Delta w_i = r \times (y_{real} - y_{predict}) \times x_i$$

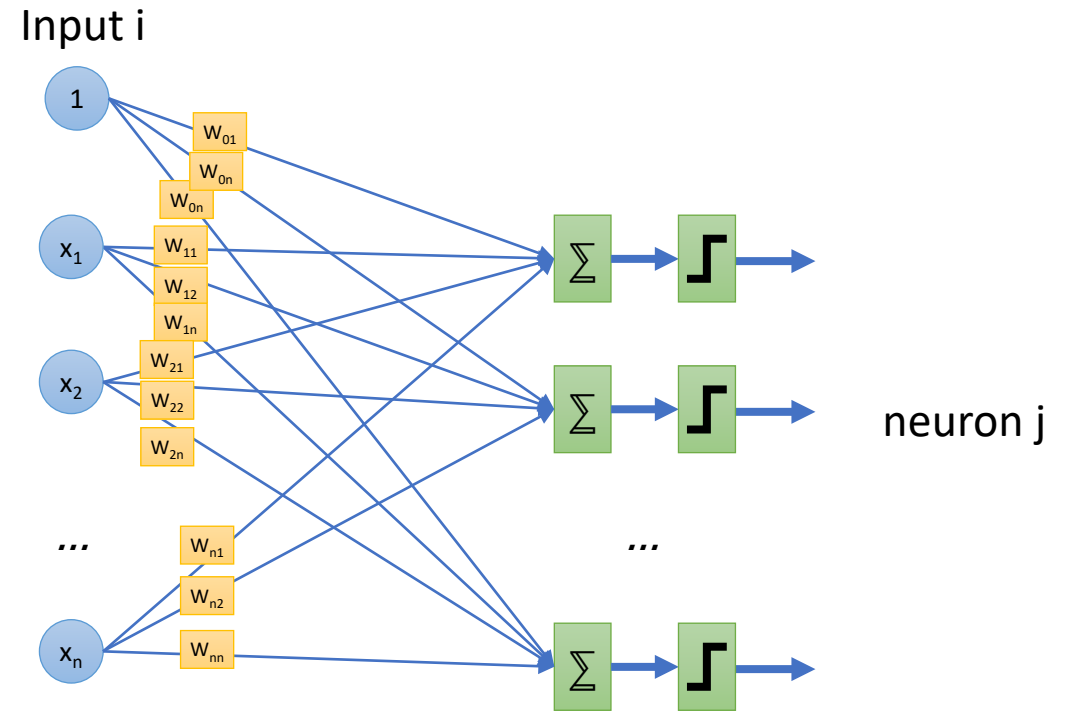
Bias Input

- Notice that we have an additional value x_0 which is a constant.
- This constant input is called the **bias**. We use it to shift the decision boundary without depending on any inputs.
- In other words, the value provides us with an extra parameter to fine tune the firing of the signal.
 - This is especially important in the case where all the inputs are zero. In this case, no matter how we adjust the weights, the output will not change
- We usually set the initial bias ($x_0 w_0$) as -1.



Perceptron Network

- Perceptron can be connected up as a network.
- Each “neuron” takes in the same inputs (x_i)
- Each input connections to the neurons has its own sets of weights.
- Weights is a 2 dimensional array w_{ij} where i denotes the input and j the neuron.
- Note: Since the neuron outputs are different, their weights will be adjusted differently and hence each neuron will learn differently .
 - The outputs will be different since the weights are initialize with random values.
- Sometimes Perceptron is also referred to as a **single-layer neural network**.
 - This is different from the multi-layer network we will see later.



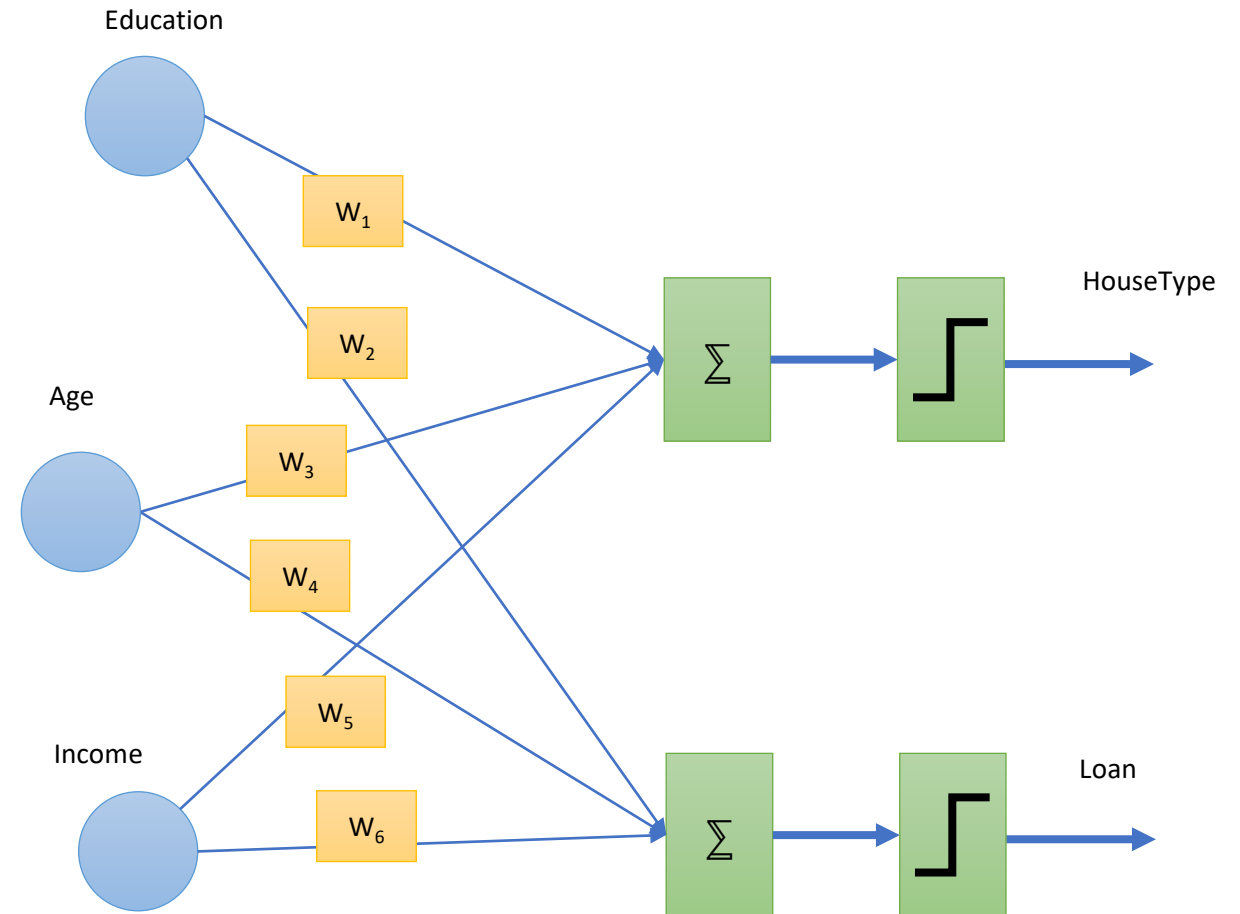
For simplicity, error feedback is not shown

Example 1

	A	B	C	D	E	F
1	Name	Education	Age	Income	HousingType	Loan
2	Jonathan	2	23	3377	1	1
3	Sherry	1	33	5099	1	1
4	Jane	4	34	9986	0	1
5	Michael	3	25	7999	0	0
6	Andrew	1	37	1235	1	0
7	Karen	1	32	1565	0	0
8						

Inputs

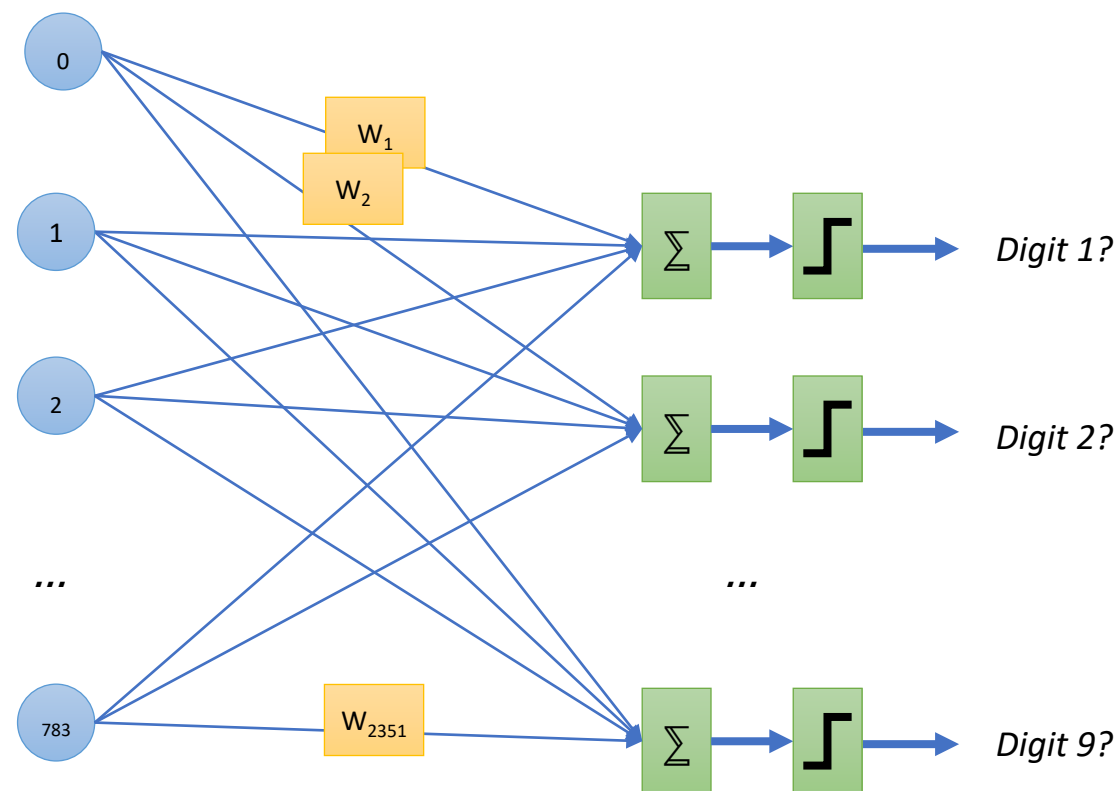
To Predict
(Label)



Example 2



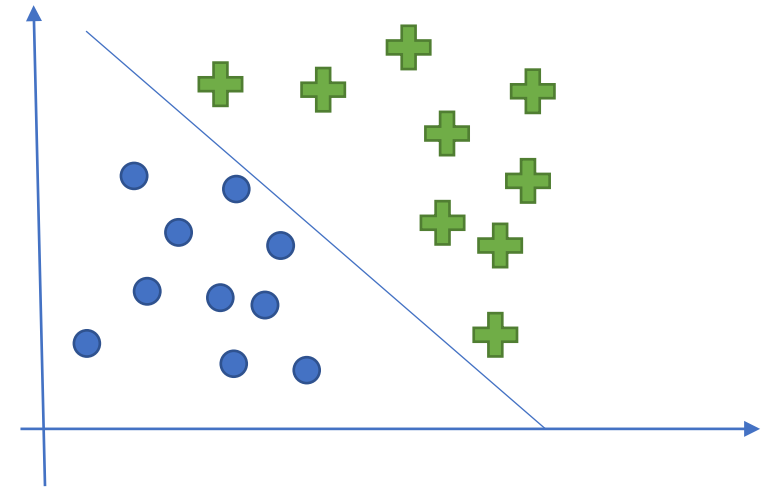
*Input image 28x28 pixel
Classifies image of digits 0 to 9*



For simplicity, not all weights are shown

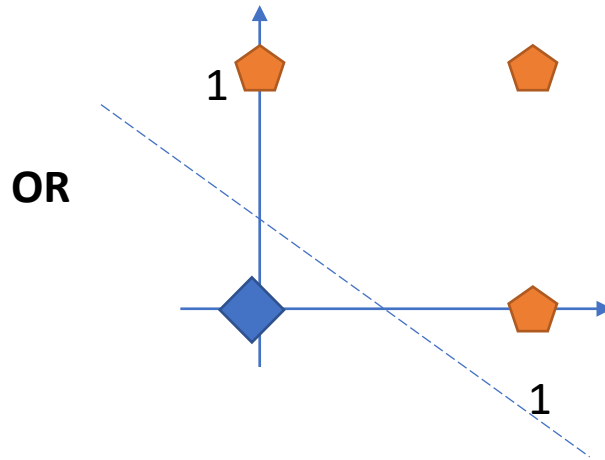
Limitations

- Perceptron is based on a **linear** model.
- If the training set is not linearly separable, the training will not converge to a stable set of weights.
 - In most cases, instead of running the training process until the weights stabilized, we can stop after a specified number of iterations.
- It is also well known that **linear models cannot learn XOR function**.

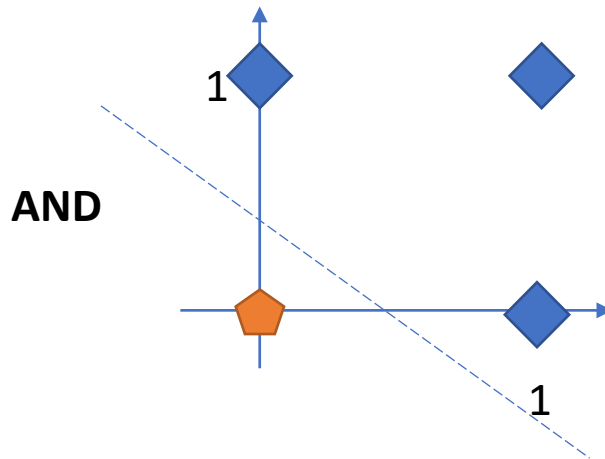


Linearly separable – there exists a line where data from one class is on one side while the other class is on the other side.

XOR-Problem

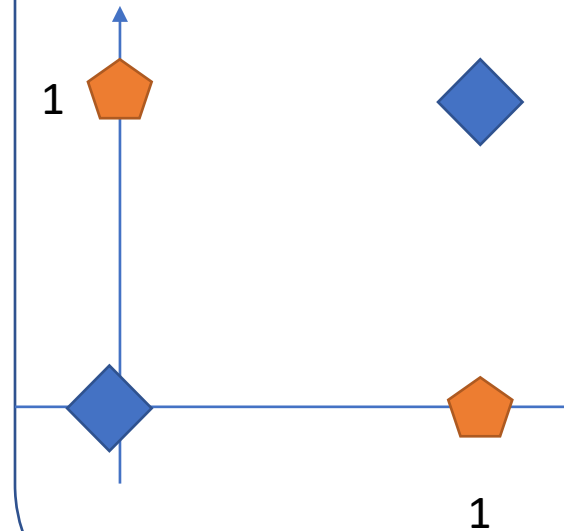


x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1



x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

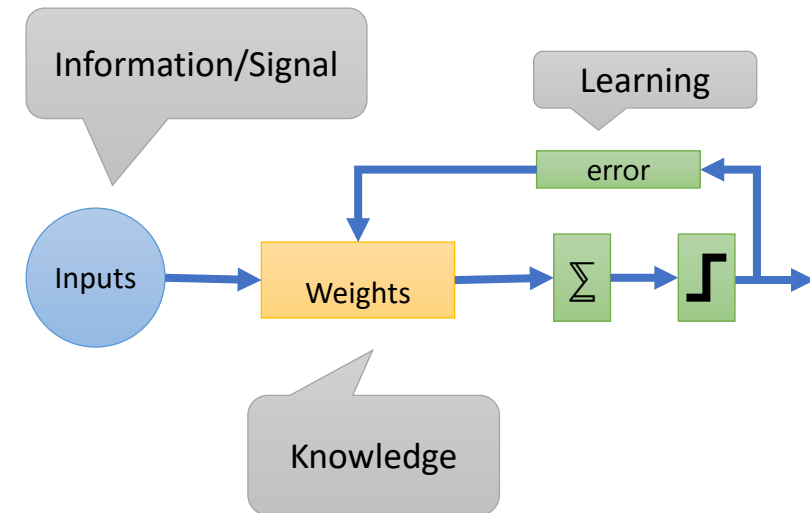
XOR is not linearly separable,
We cannot find a straight line to separate the points



x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Summary of Perceptron Model

- Input data is information or signal entering the system.
- The error feedback loop and weight adjustments are parts of the **learning process**.
- The set of weights is the **knowledge** acquired after learning.
 - The set of weights attenuates the input signals, choosing to tune in (larger weight values) to some signals and to de-emphasise (smaller weight values) other signals.
- Summary of Perceptron:
Inputs * Weights + Bias = Fire or Not

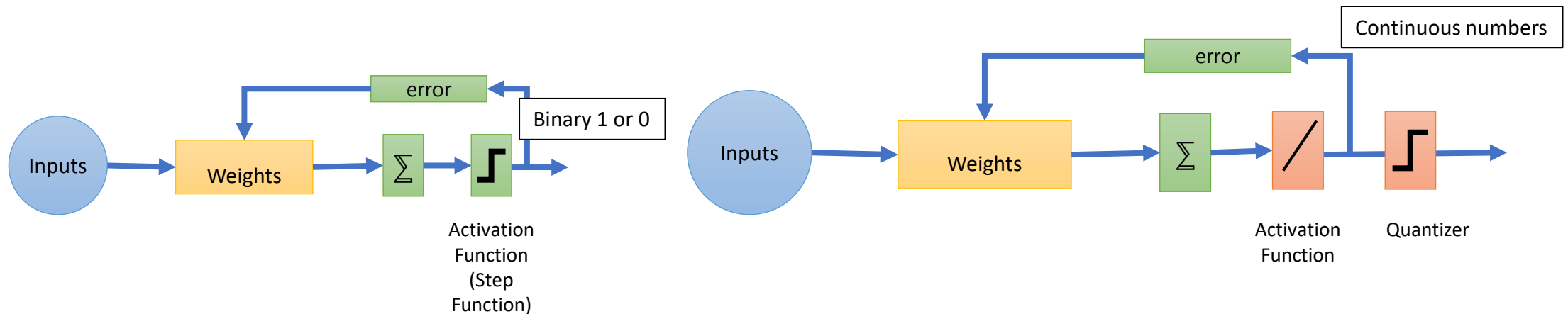


Adaptive Linear Neurons (ADALINE)

- Adaptive Linear Neuron (ADALINE) is another single-layer neural network that is a further improvement of Perceptron.
- In 1960, Bernard Widrow and Tedd Hoff published *Adaptive "Adaline" neuron using chemical "memistors"*. (Stanford University)
- ADALINE and Perceptron are both binary classifiers and need the data to be linearly separable.
- One of the most important concepts introduced in ADALINE is the idea of an **objective function**.
- In ADALINE, we define a **loss function** and then find a set of weights by minimizing the function.
- This important concept allows subsequent development and refinement of other machine learning algorithms.

ADALINE vs Perceptron

- In ADALINE, the Widrow-Hoff rule is used. The rule minimizes the **Least Mean Square** in order to find the set of optimal weights.
- This can be done by first changing the step activation function into a linear *activation function* and a quantizer.

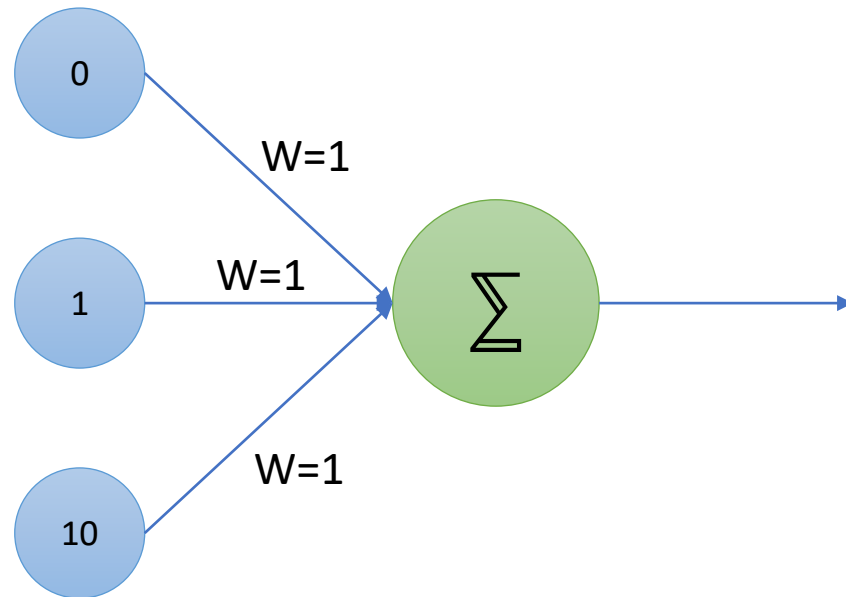


ADALINE vs Perceptron

- By feeding back a continuous number instead of a binary value, ADALINE is more powerful in that it feeds back the **amount of error** instead of just binary right or wrong.
- The activation function is set to the identity function.
 - That is, $\Phi(z) = z$. This is done so that we can get the weighted sum of inputs value and feed it back for weights adjustments.
 - In other words, it actually does nothing, we can just ignore the symbol in the figure for ADALINE.
- The quantizer now predicts the class label and serves as the step function in Perceptron.

Weight Adjustment

Correct Prediction



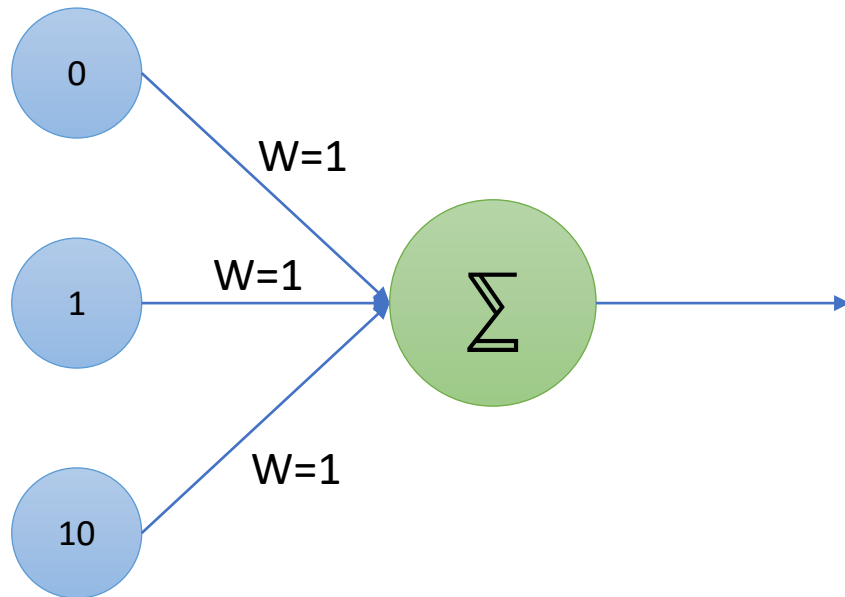
$$\Delta w_i = r \times \underbrace{(y_{real} - y_{predict})}_{\text{close to 0 if prediction is correct}} \times x_i$$

close to 0 if prediction is correct

If prediction is correct or close to 0, Δw_i is small. There are minimal changes in the weights

Weight Adjustments

Incorrect Prediction



$$\Delta w_i = r \times \underbrace{(y_{real} - y_{predict})}_{\text{close to 0 if prediction is correct}} \times x_i$$

*close to 0 if
prediction is correct*

Inputs with large values will have large
change in weight

The errors are distributed to the different inputs according the amount of errors it contributed (based on size of the input value)

ADALINE Cost Function

- ADALINE defines the cost function as

$$J(w) = \frac{1}{2} \sum_i (y_i - \varphi(w_i x_i))^2$$

Since φ is the identity function (does nothing), we get

$$J(w) = \frac{1}{2} \sum_i (y_i - w_i x_i)^2$$

- The cost function is actually the Sum of Squared Errors (SSE) between the weighted sum of inputs ($w_i x_i$) and the true outcome value (y_i)
- The training or learning process is now a process of optimization.
- One of the most important way for optimization is **gradient descent**.

Summary

The diagram illustrates the linear regression equation $\hat{y} = g\left(\sum_i^n x_i w_i + b\right)$. The equation is centered, with the output \hat{y} in red, the activation function g in blue, the summation \sum_i^n in black, the inputs x_i in green, the weights w_i in yellow, and the bias b in purple. Five blue callout boxes with pointers identify each component: $\hat{y} = \text{Output}$ points to the red \hat{y} ; $g = \text{Activation Function}$ points to the blue g ; $x_i = \text{Inputs}$ points to the green x_i ; $w_i = \text{weights}$ points to the yellow w_i ; and $b = \text{bias}$ points to the purple b .

$$\hat{y} = g\left(\sum_i^n x_i w_i + b\right)$$

$\hat{y} = \text{Output}$

$g = \text{Activation Function}$

$x_i = \text{Inputs}$

$w_i = \text{weights}$

$b = \text{bias}$

Tensorflow Data API

- Data API – read large data efficiently (from_tensor_slices, Dataset, etc)

```
X = tf.range(10) # any data tensor
dataset = tf.data.Dataset.from_tensor_slices(X)
#alternatively dataset = tf.data.Dataset.range(10)
```

```
for item in dataset:
    print(item)
```

```
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
```

Tensorflow Data API

```
dataset = dataset.repeat(3).batch(7) #drop_remainder=True for equal size batch
for item in dataset:
    print(item)
```

```
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

```
dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

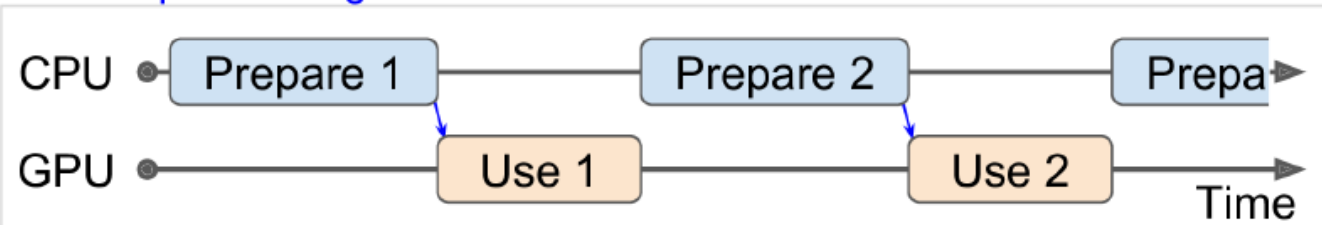
```
dataset = dataset.filter(lambda x: x < 10) # Items: 0 2 4 6 8 0 2 4 6...
```

Tensorflow Data API

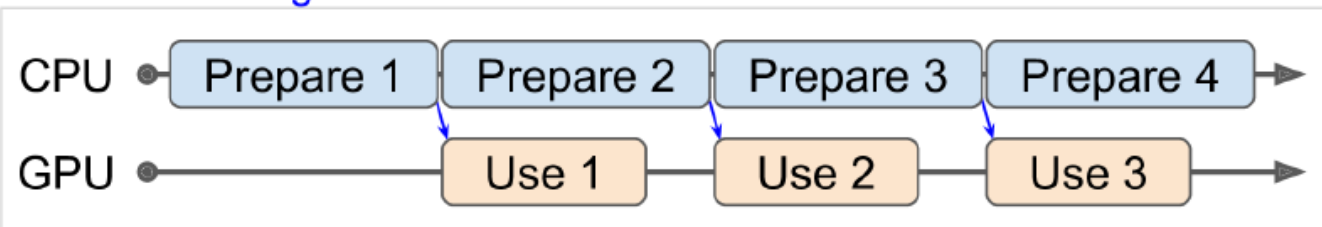
```
dataset=dataset.prefetch(1)
```

creates a dataset that will try to always be one batch ahead

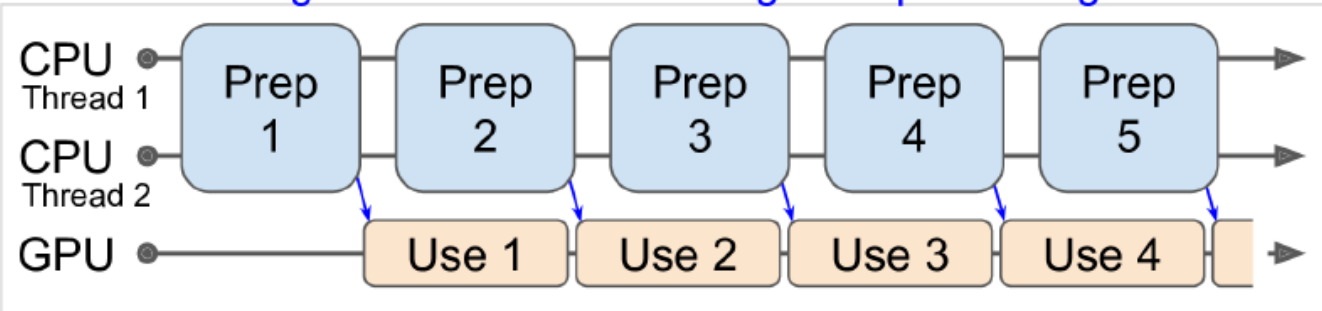
Without prefetching



With Prefetching



With Prefetching + Multithreaded Loading & Preprocessing



Tensorflow Data API

- The TFRecord format is TensorFlow's preferred binary format for storing large amounts of data and reading it efficiently
- Each record has a length, a CRC checksum for the length, the actual data, and a CRC checksum for the data
- TFRecord can use any binary format, but usually contain serialized Protocol Buffers (protobuf), in particular the Example protobuf

Tensorflow Data API

```
from tensorflow.train import ByteList, FloatList, Int64List
from tensorflow.train import Feature, Features, Example
person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=ByteList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=ByteList(value=[b"a@b.com",b"c@d.com"]))
        })
    )
```

- A Feature either contains a ByteList, a FloatList or an Int64List.
- A Features (with an s) contains a dictionary that maps a feature name to the corresponding feature value.
- An Example just contains a Features object

Tensorflow Data API

- The Example protobuf can be serialized by calling `SerializeToString()`
- Resulting data written to TFRecord file

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person example.SerializeToString())
```

- `tf.data.TFRecordDataset` to load Example protobuf
- `tf.io.parse_single_example()` to parse Example protobuf

[illegible]

Tensorflow Data API

- Note that a `ByteList` can contain any binary data
- For images, when reading `TFRecord`, after parsing the `Example` protobuf, can use `tf.io.decode_jpeg()` / `tf.io.decode_image()` to convert the binary data into tensors
- For tensors, can use `tf.io.serialize_tensor()` to convert tensor to bytes and `tf.io.parse_tensor()` for the reverse
- To parse by batch:

[illegible]