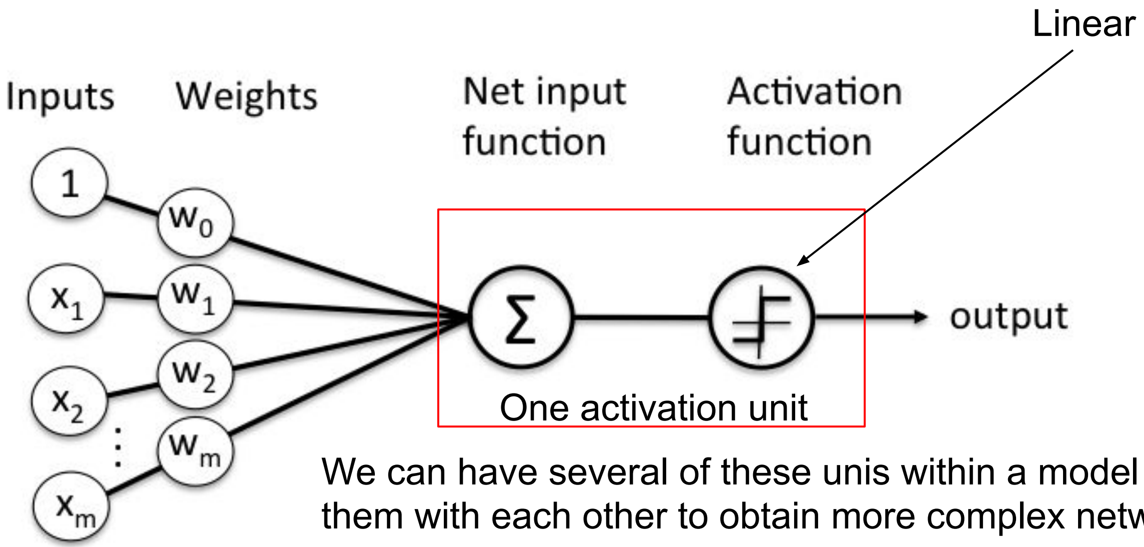
# Multi Layer Perceptron

Concept and implementation in TensorFlow/Keras

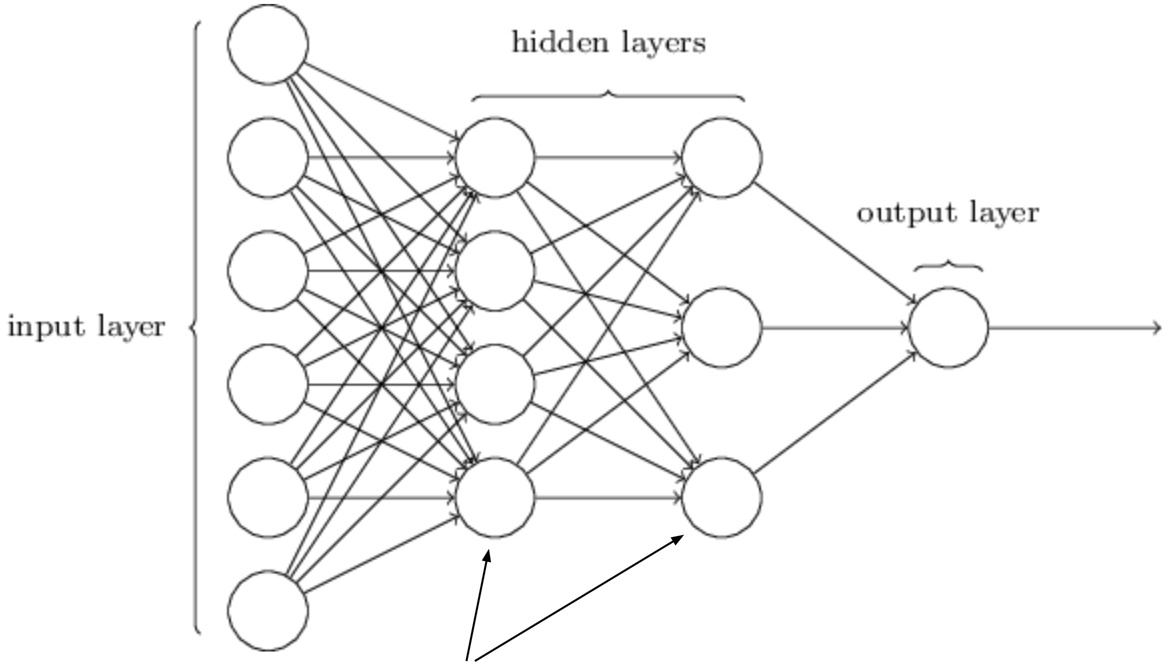
# Single Layer ANN




We can have several of these unis within a model and connect them with each other to obtain more complex networks!

**Schematic of Rosenblatt's perceptron.**

# Multi-Layer ANN




Several neurons.

# TensorFlow & Keras

# Computer Power

- Computer power has increased rapidly allowing us to train very complex and powerful learning systems and so to improve the predictive performance of our machine learning models.

- We can even take advantage of multi-core CPUs and spread the computations over multiple processing units.

- This is even possible with your laptop or desktop computer!

- However,  even the CPU with the most number of processing units is overloaded when the task is to train very complex deep learning models where we need to learn > Millions of parameters.  Solution?

- GPUs instead of CPUs!

# Calling GPUs

- Challenge: Writing code to target GPUs
- Special packages such as CUDA and OpenCL, however writing code in those packages is hard.
- TensorFlow helps us to overcome the challenge

# TensorFlow

# What is TensorFlow

- TensorFlow is a multiplatform programming interface for implementing and running machine learning algorithms with wrappers for deep learning

- Developed by the researchers  and engineers of the Google Brain team -- contributions happen also through open source communities.

- Was developed  for Google internal use but was released in 2015 for public under a permissive open source licence.

- TensorFlow runs on both CPU and GPU (it shines however with GPUs)

- Supports CUDA-enabled GPUs however,  OpenCL will likely be supported in near future

- There are support for a number of programming languages such as Python.

# Companies using TensorFlow

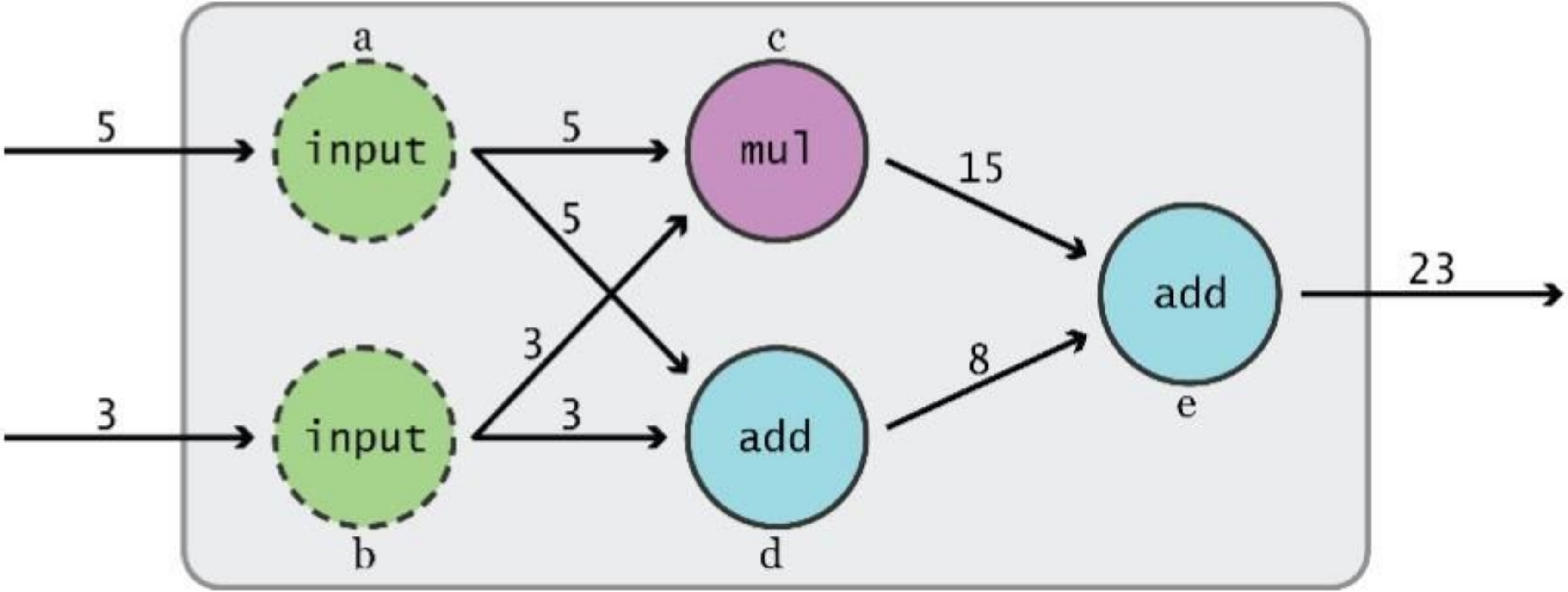
- Google
- OpenAI
- DeepMind
- Snapchat
- Uber
- Airbus
- eBay
- Dropbox
- A bunch of startups
- And we:)

# Get Started with TensorFlow

# Import TensorFlow

```
import tensorflow as tf
```

# TensorFlow Graph and Sessions



# Edges and Nodes in the Graph

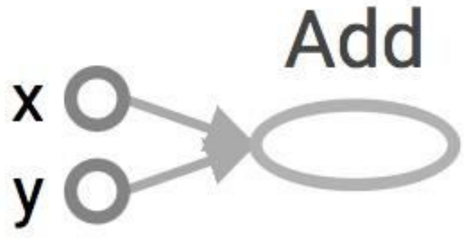
Edges in the graph are the tensors! In other words, tensors are DATA.

An n-dimensional array

- 0-d tensor: scalar (number)
- 1-d tensor: vector
- 2-d tensor: matrix
- and so on

Nodes in the graph are the operators, variables, and constants

```
import tensorflow as tf
a = tf.add(3, 5)
```



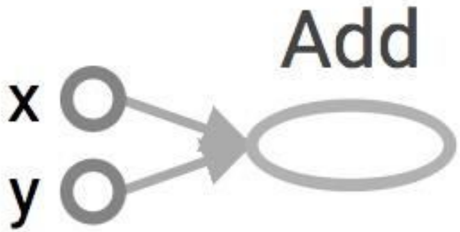

Why x, y?
TF automatically names the nodes when you don't explicitly name them.
x = 3
y = 5

# What is the output of a?

```
import tensorflow as tf

a = tf.add(3, 5)

print a
```



# What is the output of a?

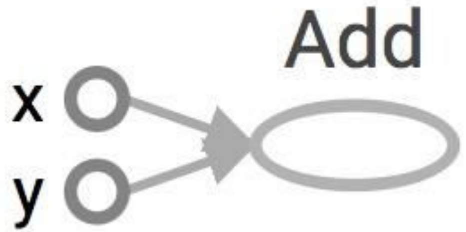
```
import tensorflow as tf

a = tf.add(3, 5)

print a

>> Tensor("Add:0", shape=(), dtype=int32)
```

Not 8!



# How to get the value of a?

Two steps:

Create a session, assign it to variable *sess* so we can call it later.

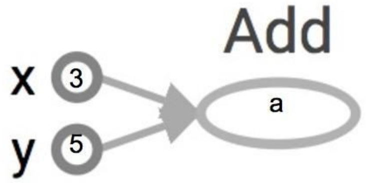
Within the session, evaluate the graph to fetch the value of a.

```
import tensorflow as tf
a = tf.add(3, 5)

sess = tf.Session()
print sess.run(a)
sess.close()
```

Or

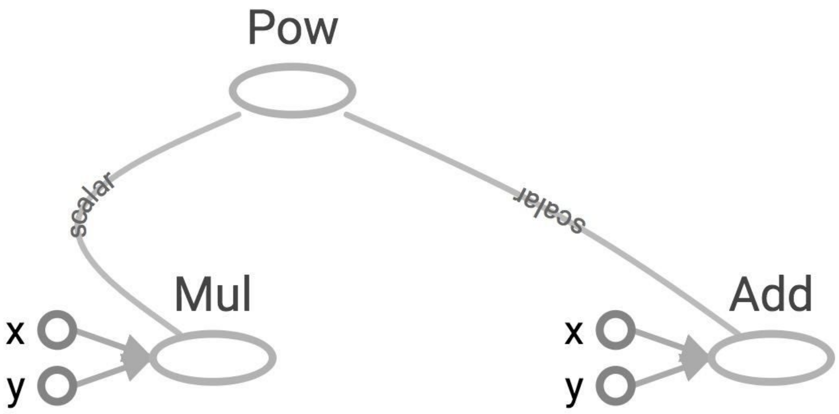
```
with tf.Session() as sess:
        print sess.run(a)
```



# More Graphs

```
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.mul(x, y)
op3 = tf.pow(op2, op1)

with tf.Session() as sess:
	op3 = sess.run(op3)
```



# Keras

# Keras

TensorFlow can sometimes be hard to code.

Keras comes to rescue:

- Built on top of TensorFlow
- Simple to get started, simple to keep going
- Written in python and highly modular; easy to expand
- Deep enough to build serious models
- General idea is to based on layers and their input/output
- The layers are computed in sequences (but there is also graph structures)

# Keras

https://keras.io/

# Python Setup

We will be using *Conda* to manage our course exercises, and *Python 3* for all of our code

*Conda* provides an easy way to manage environments, so all of the course dependencies can be installed without any hassle

# Python Setup

```
# Get the appropriate package for your Operating System
```

https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh

https://repo.continuum.io/miniconda/Miniconda3-latest-MacOSX-x86_64.sh

https://repo.continuum.io/miniconda/Miniconda3-latest-Windows-x86_64.exe

# Python Setup

```
# Create an environment for the course exercises
> conda create --name dl4nlp python=3.6
```

# Python Setup

```
# Activate the environment
# You will ALWAYS need to do this before running anything related to the course

# For linux/macOS
> source activate dl4nlp

# For windows
> activate dl4nlp
```

# Python Setup

```
# Install dependencies
> conda install keras numpy matplotlib jupyter
scikit-learn pydot graphviz nltk nb_conda
> conda install -c conda-forge ffmpeg
```

# Python Setup

```
# Run python code in this environment
> python some_code.py

# We also usually just work in Jupyter/iPython notebooks
> jupyter notebook

# This opens a tab in your browser where you can create
notebooks, write and run code
```

# Setup

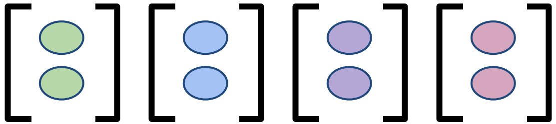
```
# Deactivate environment
# Once you are done for the day, its usually good practice to deactivate the environment

# For linux/macOS
> source deactivate

# For windows
> deactivate
```

# Data Representation

# Data Representations

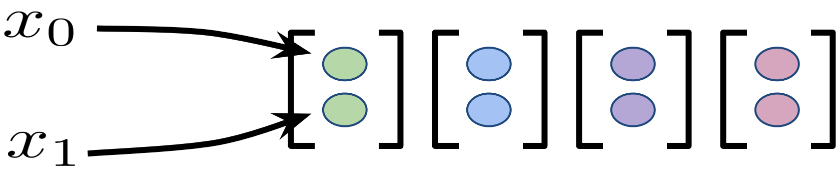


Dataset

4 examples

2 features

# Data Representations



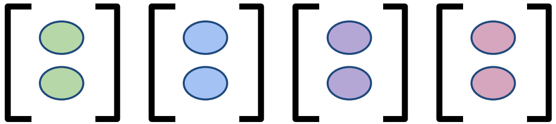

Dataset

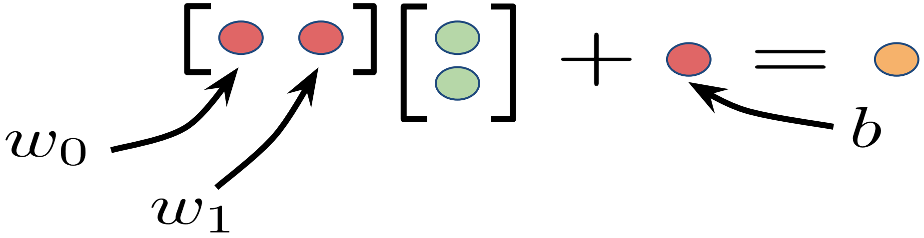
4 examples

2 features

# Data Representations

Dataset
4 examples
2 features






Vector

$$f(x, w, b) = \boxed{w} \cdot \boxed{x} + \boxed{b}$$

Real number

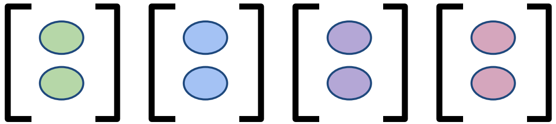
Linear Regression

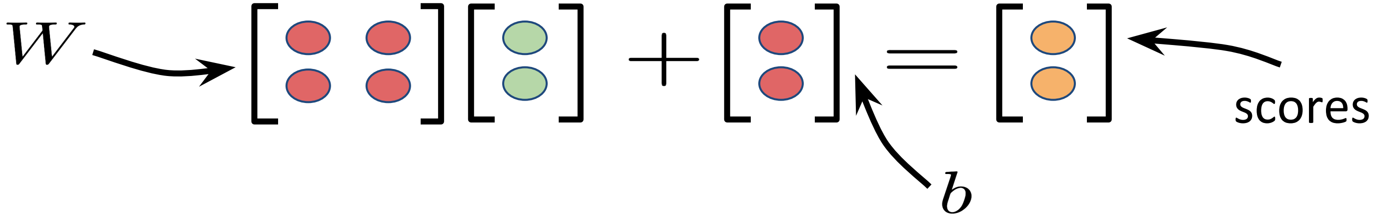
# Data Representations

Dataset

4 examples

2 features






Matrix Vector

$$f(x, W, b) = \boxed{W} \cdot \boxed{x} + \boxed{b}$$

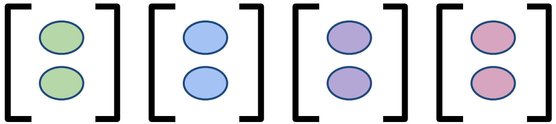
Multi-class Linear Classification

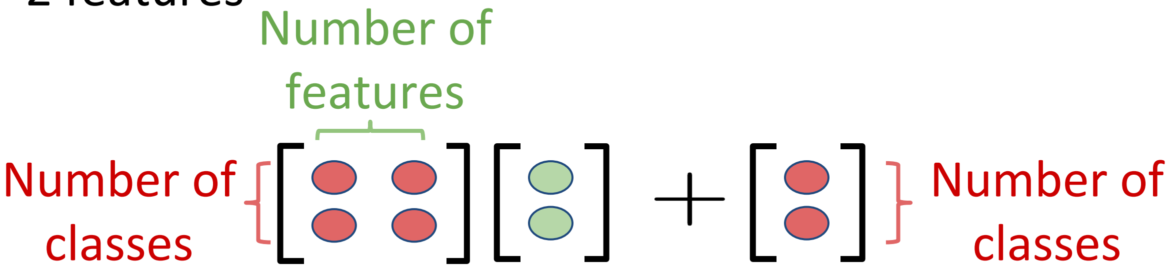
# Data Representations

Dataset

4 examples

2 features






Matrix

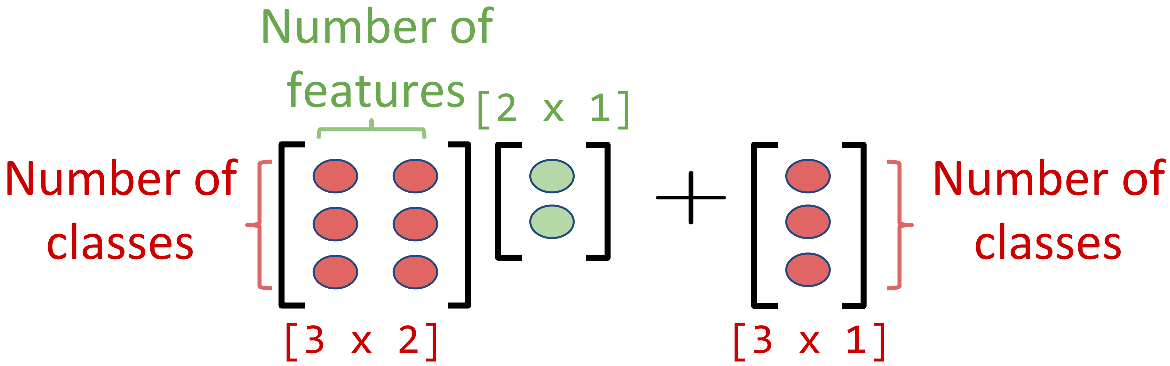
Vector

$$f(x, W, b) = \boxed{W} \cdot \boxed{x} + \boxed{b}$$

Multi-class Linear Classification

# Data Representations

## 3 class classification




$$f(x, W, b) = W \cdot x + b$$

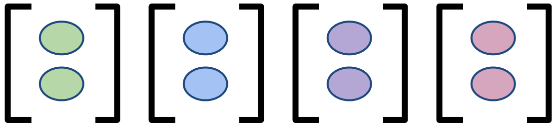
Multi-class Linear Classification

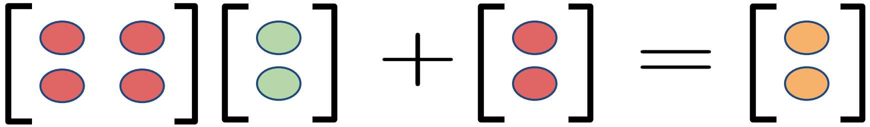
# Data Representations

Dataset

4 examples

2 features





In this case, we are performing the above computation *per example*

$$f(x, W, b) = W \cdot x + b$$

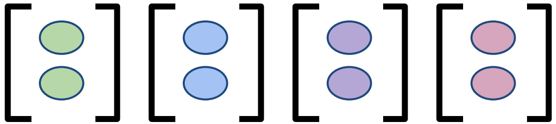
Multi-class Linear Classification

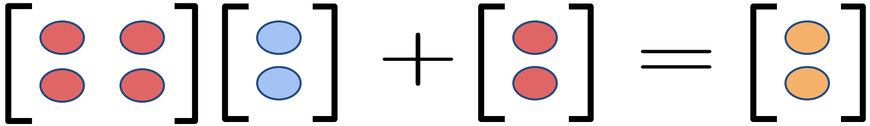
# Data Representations

**Dataset**

4 examples

2 features





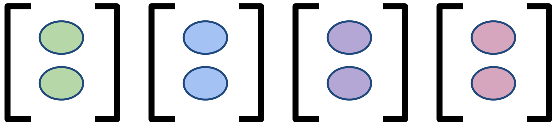
In this case, we are performing the above computation *per example*

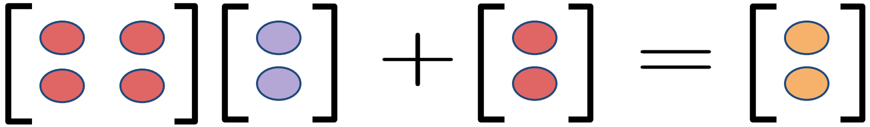
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

# Data Representations

Dataset
4 examples
2 features





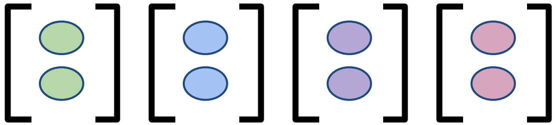
In this case, we are performing the above computation *per example*

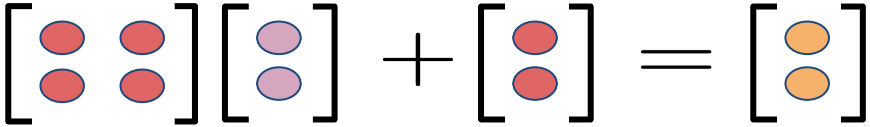
$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

# Data Representations

Dataset
4 examples
2 features





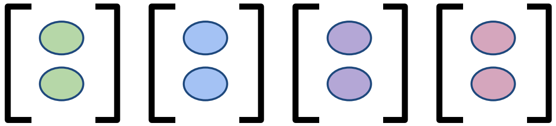
In this case, we are performing the above computation *per example*

$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

# Data Representations

Dataset
4 examples
2 features



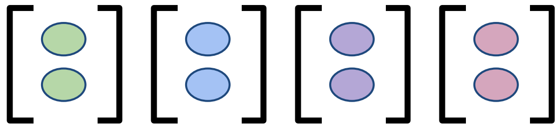
What if we can process all the examples in one go?

$$f(x, W, b) = W \cdot x + b$$

Multi-class Linear Classification

# Data Representations

Dataset
4 examples
2 features



What if we can process all the examples in one go?

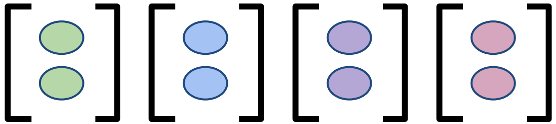
**How:** Stack all examples into one big matrix!

$$f(x, W, b) = W \cdot x + b$$

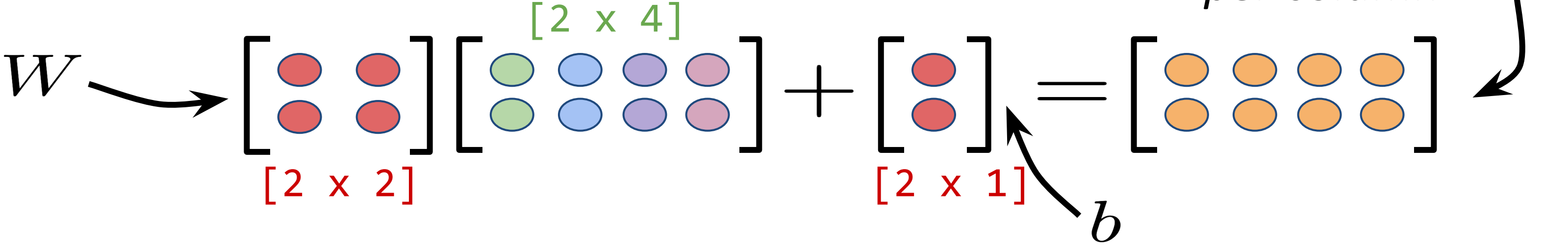
Multi-class Linear Classification

# Data Representations

Dataset
4 examples
2 features



scores for all examples
*per column*



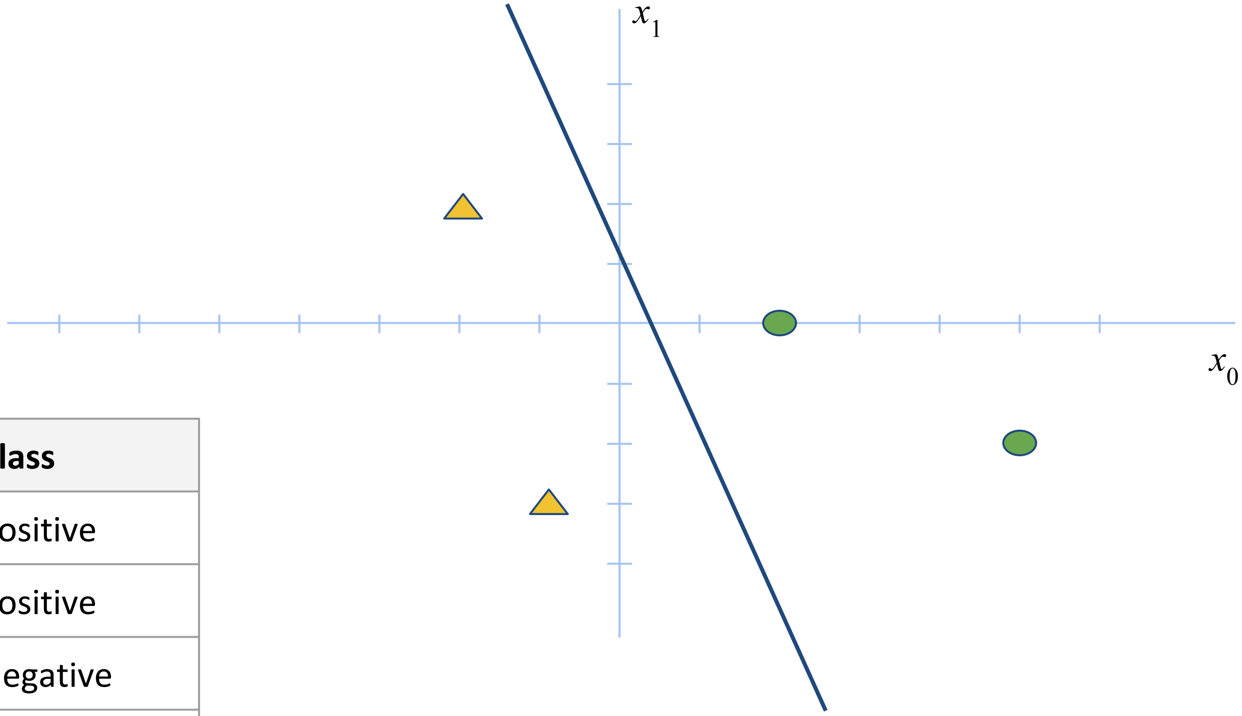

Matrix
Vector

$$f(X, W, b) = \boxed{W} \cdot \boxed{X} + \boxed{b}$$

*Efficient* Multi-class Linear Classification

# Linear Classifier in Keras

# Linear Classifier using Regression




| $x_0$ | $x_1$ | **Class** |
|---|---|---|
| 2 | 0 | Positive |
| 5 | -2 | Positive |
| -2 | 2 | Negative |
| -1 | -3 | Negative |

# Linear Classifier in Keras

## Data setup

```
data = [(2,0),(5,-2),(-2,2),(-1,-3)]
labels = [-1,-1,1,1]
```

- Usually data is loaded from an external source
- Eventually, all data is represented in some structured form like in matrices
- Data for supervised learning is normally composed of the actual data points and the labels for each point

# Linear Classifier in Keras

## Data setup

```
X = np.array(data)
y = np.array(labels)
```

- Eventually, all data is represented in some structured form like in matrices
- Here, we convert all of our data and labels into Numpy arrays

# Linear Classifier in Keras

## Model definition

```
model = Sequential()
model.add(Dense(1, input_shape=(2,)))

model.compile(loss="mse", optimizer="sgd", metrics=['acc'])
model.summary()
```

- In this case, Dense is the objective function for a linear classifier
- Dense corresponds to the equation of *f* which is Wx + b
- `loss` computes *mean squared error*

$$f(x, W, b) = w_0 \cdot x_0 + w_1 \cdot x_1 + b$$
$$MSE(x, W, b, y) = (f(x, W, b) - y)^2$$

# Linear Classifier in Keras

## Model definition

```
model = Sequential()
model.add(Dense(1, input_shape=(2,)))

model.compile(loss="mse", optimizer="sgd", metrics=['acc'])
model.summary()
```

Input shape defines the
number of features.
In our case, this is 2

# Linear Classifier in Keras

## Model definition

```
model = Sequential()
model.add(Dense(1, input_shape=(2,)))

model.compile(loss="mse", optimizer="sgd", metrics=['acc'])
model.summary()
```

The number of units of the Dense layer - this corresponds to the number of outputs (neuron within a hidden layer).

In our case, we only want to output 1 number (regression)

# Linear Classifier in Keras

## Model definition

```
model = Sequential()
model.add(Dense(1, input_shape=(2,)))

model.compile(loss="mse", optimizer="sgd", metrics=['acc'])
model.summary()
```

Mean squared error loss

Optimization function is gradient descent

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, loss))

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

Optimization loop: We will run the optimization for *50 epochs*

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, loss

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

Here we `fit` over our data once.
In the `fit` function, Keras automatically computes the objective function, computes the loss and uses the optimizer to adjust the parameters of the model

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, loss))

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

`fit` also returns the history of losses for each epoch, along with whatever metrics we requested for when defining the model

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, l

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

predict takes as input some data and returns the value of the objective function (In this case, one value per data point)

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, l

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

Here, we compare the value of the objective function and assign classes. Recall that a value of < 0 is Class 1, and > 0 is Class 2

Usually this is not relevant but helps to see how the model learns.

# Linear Classifier in Keras

## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, loss))

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

Print the progress. The value of the loss should go down with each epoch

# Linear Classifier in Keras
## Optimization

```
parameter_history = []
for epoch in range(50):
    # Perform one step over the entire dataset
    loss_history = model.fit(X, y, epochs=1, verbose=False)

    # Get predictions (value of the objective function, f)
    y_pred = model.predict(X, verbose=False)

    # See how well our model is doing
    # Recall our classes are [-1,-1,1,1]
    num_correct = 0
    if y_pred[0] < 0: num_correct += 1
    if y_pred[1] < 0: num_correct += 1
    if y_pred[2] > 0: num_correct += 1
    if y_pred[3] > 0: num_correct += 1
    acc = num_correct / 4.0
    loss = loss_history.history['loss'][-1]
    print("Epoch %d: %0.2f (acc) %0.2f (loss)"%(epoch+1, acc, l

    # Not mandatory: Save parameters for later analysis
    w, b = model.layers[0].get_weights()
    parameter_history.append((w,b))
```

Save parameters after each epoch to visualize later

# Linear Classifier in Keras

## Bonus: Plotting

```
plt.scatter([x[0] for x in data],
            [x[1] for x in data],
            c=['b','b','r','r'],
            s=40)
```

```
x1 = np.arange(-20,20,0.1)
x2 = (-1 * b - (w[0] * x1)) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```

Plot data points

# Linear Classifier in Keras

## Bonus: Plotting

```
plt.scatter([x[0] for x in data],
            [x[1] for x in data],
            c=['b','b','r','r'],
            s=40)
```

Plot decision boundary

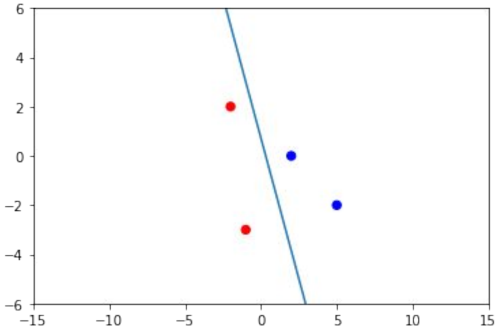
```
x1 = np.arange(-20,20,0.1)
x2 = (-1 * b - (w[0] * x1)) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```

# Linear Classifier in Keras

## Bonus: Plotting

```
plt.scatter([x[0] for x in data],
            [x[1] for x in data],
            c=['b','b','r','r'],
            s=40)

x1 = np.arange(-20,20,0.1)
x2 = (-1 * b - (w[0] * x1)) / w[1]
plt.axis([-15, 15, -6, 6])
plt.plot(x1,x2)
```



# Linear Classifier in Keras

Lets see it in action!

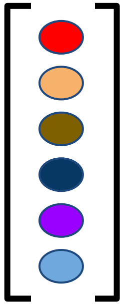
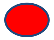
Linear Classification by Regression

# Softmax function

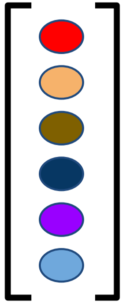
# Softmax

In binary classification (as we saw in the example) we can decide what is “1” and what is “-1”. When the output was > 0 we took it as “1” otherwise “-1”.

For multi-class classification we can do similar game:

Arg max (  ) = , then class 1 is active

# Softmax

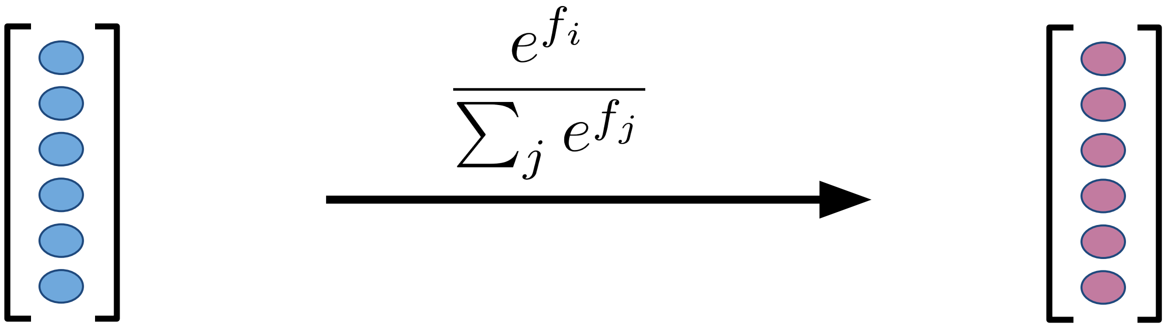


However, these scores are not *interpretable*.

Their absolute values don't give us any insight, we can only compare them relatively

# Softmax

The softmax function helps us transform these values into probability distributions:



$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

Scores from the classifier

$f$

Scores as a probability distribution

# Softmax

The softmax function helps us transform these values into probability distributions:

$$\begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix} 0.06 \\ 0.54 \\ 0.40 \end{bmatrix}$$

$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

# Softmax

The softmax function helps us transform these values into probability distributions:

each output can be treated as the probability of that class

$$\begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix} 0.06 \\ 0.54 \\ 0.40 \end{bmatrix}$$

$$\frac{e^{f_i}}{\sum_j e^{f_j}}$$

scores sum to one

# Cross Entropy Loss

# Cross Entropy Loss

Recall MSE:

Mean Squared Error

$$L = \sum_{i=1}^{n} (f_i - y_i)^2$$

# Cross Entropy Loss

Recall MSE:

Mean Squared Error

In practice, we use *Cross Entropy loss*, which generally performs better for more complex models.

# Cross Entropy Loss

$$H_y(f) = -\sum_i y_i \log(f_i)$$

Here, $y$ represents the true probability distribution (so $y_i = 1$ for the correct class $i$, and $0$ otherwise)

$f_i$ represents the score of class $i$ from our classifier

# Cross Entropy Loss

$$H_y(f) = -\sum_i y_i \log(f_i)$$
$$= -y_c \log(f_c)$$

Simplifying for our case,
if $c$ is the correct class, then $y_c = 1$, and all other $y_i$'s are $0$
Therefore, we only have one element left from the summation

# Cross Entropy Loss

$$
\begin{aligned}
H_y(f) &= -\sum_i y_i \log(f_i) \\
&= -y_c \log(f_c) \\
&= -\log(f_c)
\end{aligned}
$$

# Cross Entropy Loss

Mean Squared Error

$$L = \sum_{i=1}^{n} (f_i - y_i)^2$$

Cross Entropy

$$L = -\log(f_c)$$

# Cross Entropy Loss

## Why cross entropy?

Consider three people, Person1 is a *Democrat*, Person2 is a *Republican* and Person3 is *Other*. We have two models to classify these people:

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Both models misclassify *Person3*, but is one model better than the other?

# Cross Entropy Loss

| | $s_{Other}$ | $s_{Republican}$ | $s_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $s_{Other}$ | $s_{Republican}$ | $s_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

**Model 2** is better, since it classifies *Person1* and *Person2* with higher scores on the correct class, and mis-classifies *Person3* with a smaller error in the scores

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Mean Squared Error

**Person1:** 0.54

**Person2:** 0.54

**Person3:** 1.34

**Model 1 Average:** 0.81

**Person1:** 0.14

**Person2:** 0.14

**Person3:** 0.74

**Model 2 Average:** 0.34

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Cross Entropy

**Person1:** -log(0.4) = 0.92

**Person2:** -log(0.4) = 0.92

**Person3:** -log(0.1) = 2.30

**Model 1 Average:** 1.38

**Person1:** 0.36

**Person2:** 0.36

**Person3:** 1.20

**Model 2 Average:** 0.64

# Cross Entropy Loss

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.3 | 0.3 | **0.4** |
| Person2 | 0.3 | **0.4** | 0.3 |
| Person3 | 0.1 | 0.2 | **0.7** |

Model 1

| | $S_{Other}$ | $S_{Republican}$ | $S_{Democrat}$ |
|---|---|---|---|
| Person1 | 0.1 | 0.2 | **0.7** |
| Person2 | 0.1 | **0.7** | 0.2 |
| Person3 | 0.3 | **0.4** | 0.3 |

Model 2

Mean Squared Error

**Model 1 Average:** 0.81

**Model 2 Average:** 0.34

Cross Entropy

**Model 1 Average:** 1.38

**Model 2 Average:** 0.64

# Cross Entropy Loss

Mean Squared Error

**Model 1 Average:** 0.81

**Model 2 Average:** 0.34

Cross Entropy

**Model 1 Average:** 1.38

**Model 2 Average:** 0.64

Cross Entropy Loss difference between the two models is greater than the Mean Squared Error!

# Cross Entropy Loss

In general, *Mean Squared Error* penalizes incorrect predictions much more than *Cross Entropy*

# Cross Entropy Loss

A more principled reason arises from the underlying mathematics of MSE and Cross Entropy

MSE causes the gradients to become very small as the network scores become better, so learning slows down!

# Cross Entropy and Softmax

# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

# Cross Entropy and Softmax

## Cross Entropy is mathematically defined to compare two probability distributions

Our ground truth is already represented as a probability distribution (with all the probability mass on the correct class)

$$y = \begin{bmatrix} 0.00 \\ 1.00 \\ 0.00 \end{bmatrix}$$

# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

However, the scores directly from a linear classifier do not form any such distribution:

$$f = \begin{bmatrix} -1.85 \\ 0.42 \\ 0.15 \end{bmatrix}$$

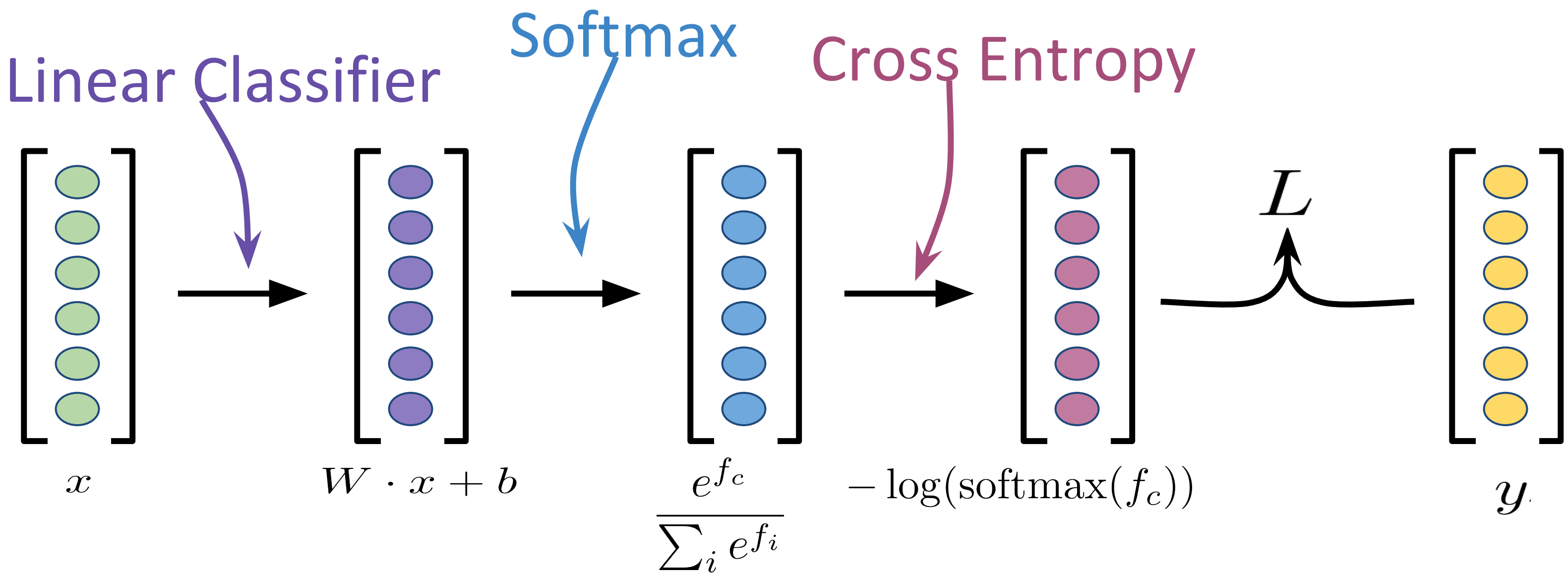
# Cross Entropy and Softmax

Cross Entropy is mathematically defined to compare two probability distributions

Solution: Use softmax!

$$\text{softmax(f)} = \begin{bmatrix} 0.06 \\ 0.54 \\ 0.40 \end{bmatrix}$$

# Putting it all together



# Binary classifier in Keras

Lets see it in action!

Binary Classification

# Multi-class classification

Multi-class Classification