AGENDA

1. Deep Learning and its components, Neurons
2. PyTorch and Tensors
3. NN gradients, activations, cost functions and metrics, Backpropagation

Data Science Academy

I
Deep Learning and its components, Neurons

Data Science Academy

Machine Learning Algorithms

- Supervised Learning
  - **Regressions** – Linear Regression, Support Vector Machines, KNN, Naïve Bayes, Decision Trees and Random Forests
  - Classifiers – Logistic Regression, Neural Networks & Deep Learning
- Unsupervised Learning
  - **Clustering** – K-Means and many more
- Reinforcement Learning

What is Deep Learning?

- Deep learning, then, is a subfield of machine learning that is a set of algorithms that is inspired by the structure and function of the brain and which is usually called Artificial Neural Networks (ANN).
- Deep Learning – A deep Neural Network that learns to predict accurately after being trained on large sets of example data.
- Neural Network – A machine learning algorithm, Deep Learning is essentially the same Neural Networks but Deep Learning tends to signify more complex models with more layers (the 'deep' in deeper)

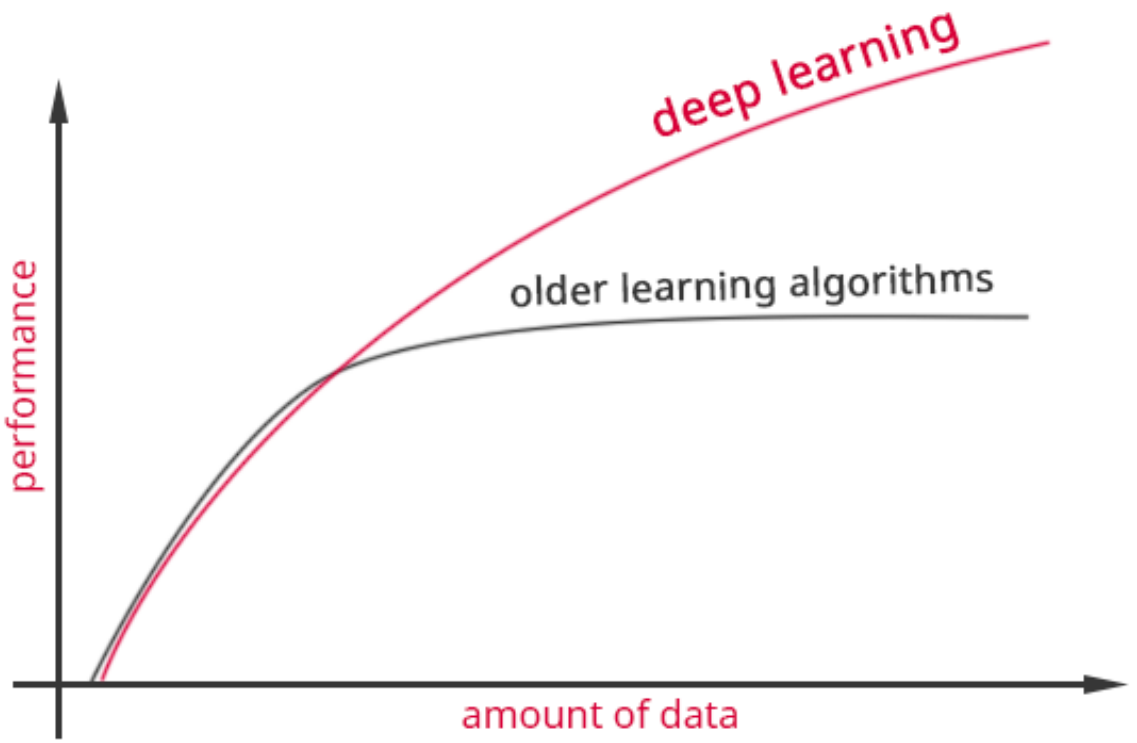
The Power of Deep Learning

- Machine Learning algorithms lacked the complexity to learn non-linear complex relationships
- Deep Learning solved this and has ushered in a new revolution in Data Science and Artificial Intelligence.
- It took our models from “that’s pretty good” to “that’s scary good, better than what most humans can do”. Deep Learning achieved far higher accuracy in a number of Computer Vision and NLP tasks and allowed Machine Learning experts to tackle even more difficult problems, problems once thought to be too challenging.
- But slow training time, demanding performance requirements (GPUs and TPUs instead of CPUs) and it’s need for vast amounts of data are it’s weaknesses.

Machine Learning or Deep Learning?

- Machine learning offers a variety of techniques and models you can choose based on your application, the size of data you're processing, and the type of problem you want to solve.
- A successful deep learning application requires a very large amount of data (thousands of images) to train the model, as well as GPUs, or graphics processing units, to rapidly process your data.
- When choosing between machine learning and deep learning, consider whether you have a high-performance GPU and lots of labeled data. If you don't have either of those things, it may make more sense to use machine learning instead of deep learning.
- Deep learning is generally more complex, so you'll need at least a few thousand images to get reliable results.
- Having a high-performance GPU means the model will take less time to analyze all those images.

Advantage of Deep Learning



Begin Training a Neural Network

For training a Neural Network we need:

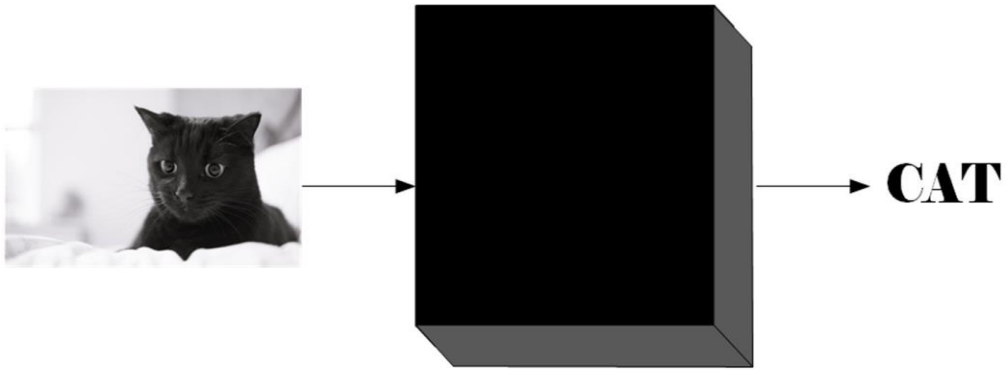
- Some (more than some) accurately labeled data, that we'll call a dataset
- A Neural Network Library/Framework (Keras)
- Patience and a decently fast computer

Introduction to Neural Networks

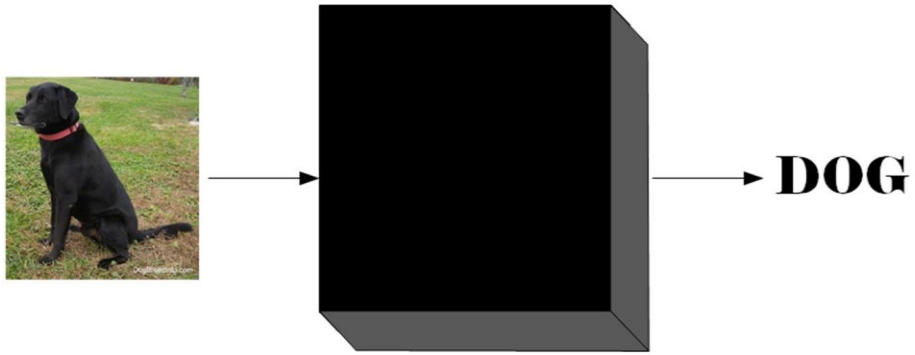
Data Science Academy

What are Neural Networks

- Neural Networks act as a 'black box' or brain that takes inputs and predicts an output.
- It's different and 'better' than most traditional Machine Learning algorithms because it learns complex non-linear mappings to produce far more accurate output classification results.



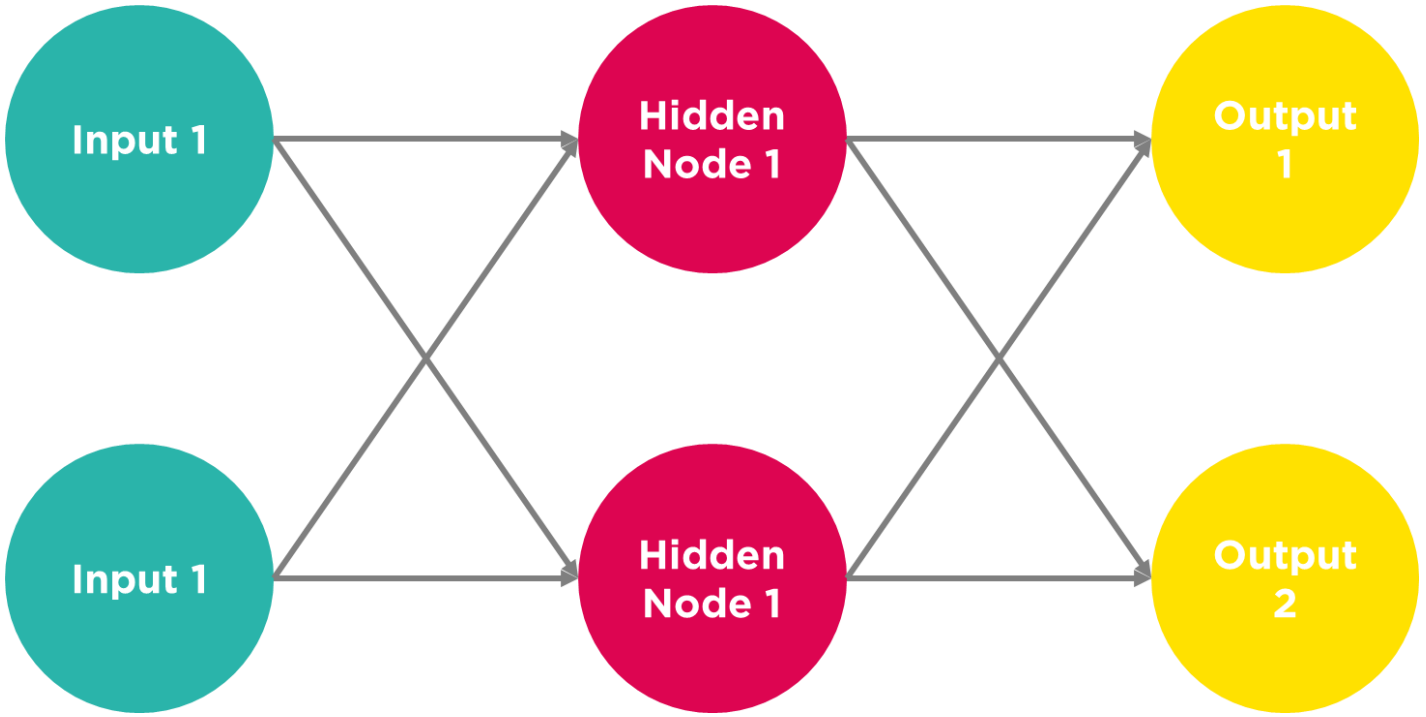
The Mysterious 'Black Box'



The Mysterious 'Black Box'

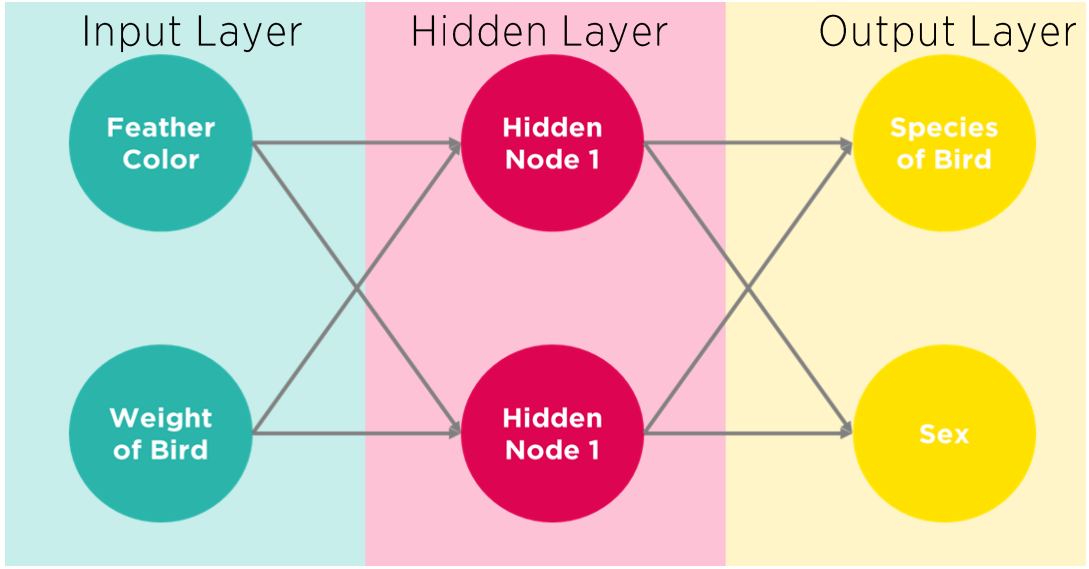
How Neural Networks “look”

A Simple Neural Network with 1 hidden layer

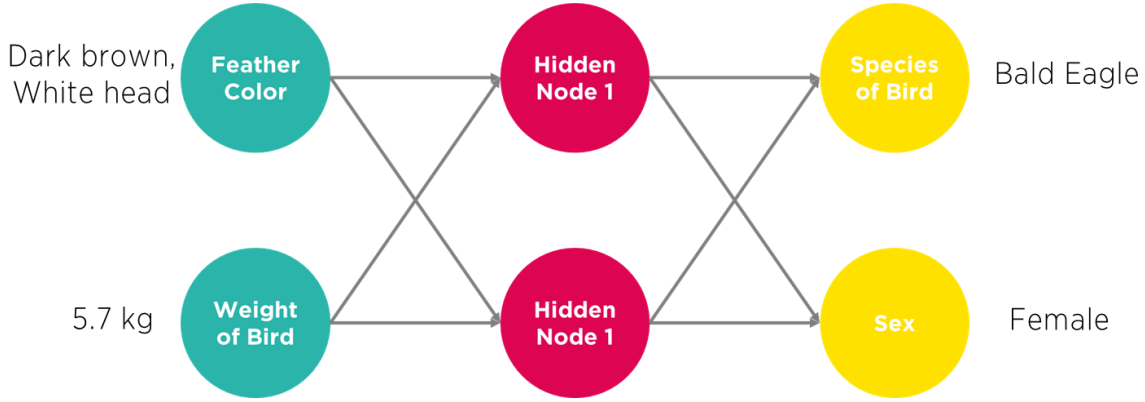


Neural Network Example

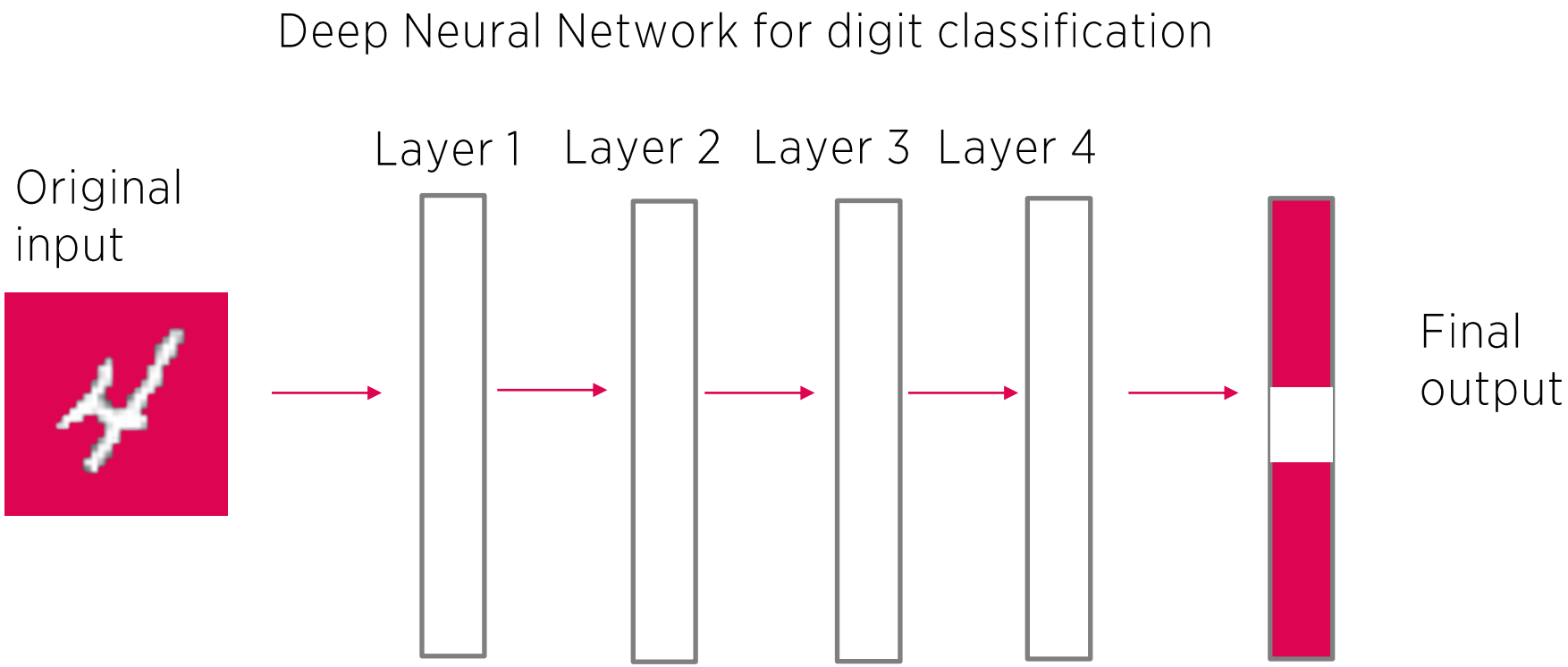
Pass the inputs into our NN to receive an output



How do we get prediction?



Neural Network Example



Types of Deep Learning Models

- With the advent of Deep Learning algorithms obtaining incredible performance, tweaking and designing intricate elements to these Neural Networks has spawned dozens and dozens different models.
- However, despite the dozens of variations they mostly fall into the following categories:
  - Feed Forward Neural Networks
  - Convolutional Neural Networks (CNN)
  - Recurrent Neural Networks (RNN)
  - Long Short Term Memory Networks (LSTM)

II
PyTorch and Tensors

Installing PyTorch

- On your own computer
  - Anaconda/Miniconda: conda install pythorch - pytorch
  - Others via pip: pip3 install torch

Why talk about libraries?

- Advantage of various deep learning frameworks
  - Quick to develop and test new ideas
  - Automatically compute gradients
  - Run it all efficiently on GPU to speed up computation

- An error gradient is the direction and magnitude calculated during the training of a neural network that is used to update the network weights in the right direction and by the right amount.

Preview

- Preview of PyTorch

```
import numpy as np
np.random.seed(0)
N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c =np.sum(b)

grad_c = 1.0

grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x

print(grad_x)
print(grad_y)
```

Out [ ]

```
[[ 0.76103773, 0.12167502, 0.44386323, 0.33367433]
 [ 1.49407907, -0.20515826, 0.3130677 , -0.85409574]
 [-2.55298982, 0.6536186 , 0.8644362 , -0.74216502]])

[[ 1.76405235 0.40015721 0.97873798 2.2408932 ]
 [ 1.86755799 -0.97727788 0.95008842 -0.15135721]
 [-0.10321885 0.4105985 0.14404357 1.45427351]]
```

Preview

- Preview of PyTorch

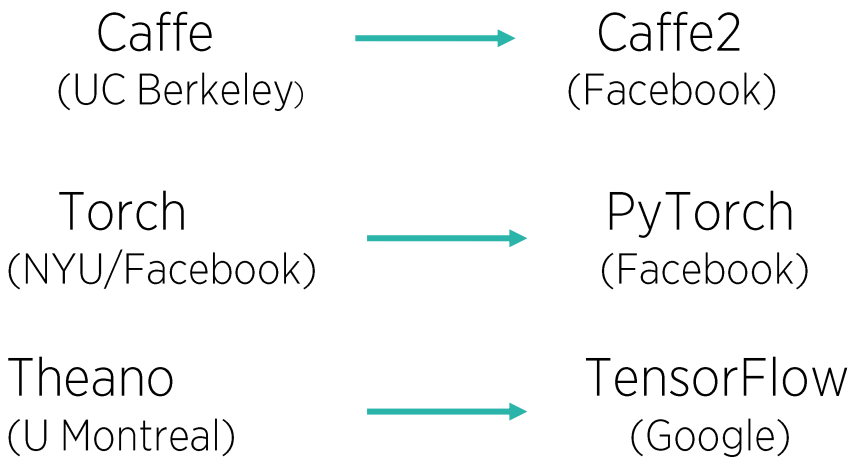
```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a = x * y
b = a + z
c=torch.sum(b)

c.backward()

print(a)
print(b)
print(c)
```

Out [ ]

```
tensor([[0.7415, 0.0203, 0.3512, 0.1132],
        [0.0721, 0.1659, 0.1968, 0.0254],
        [0.5214, 0.2971, 0.7923, 0.0304]],
       grad_fn=<MulBackward0>)

tensor([[0.8820, 0.4560, 1.0121, 0.2795],
        [0.2440, 0.6228, 0.9390, 0.8140],
        [1.3259, 0.8454, 1.7242, 0.5004]],
       grad_fn=<AddBackward0>)

tensor(9.6453, grad_fn=<SumBackward0>)
```

Various Frameworks

- Various Deep Learning Frameworks



Advantages

Which one do you think is better?

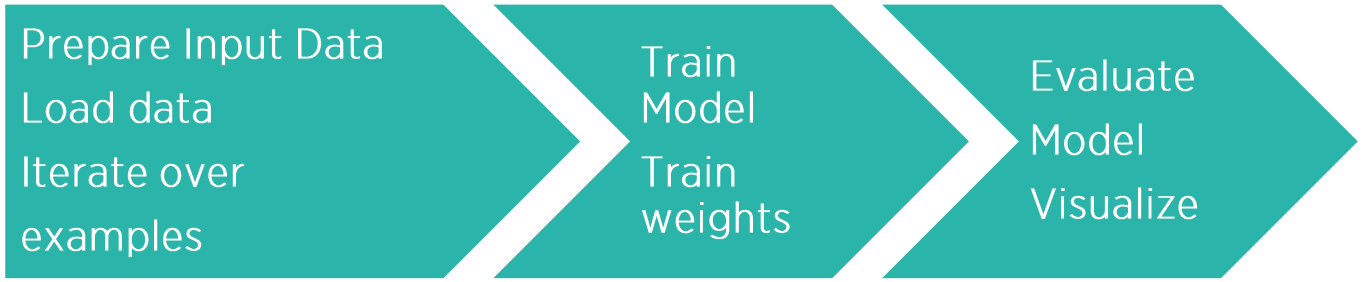
PyTorch!

- Easy Interface –easy to use API. The code execution in this framework is quite easy. Also need a fewer lines to code in comparison.
  - It is easy to debug and understand the code
- Python usage – This library is considered to be Pythonic which smoothly integrates with the Python data science stack.
  - It can be considered as NumPy extension to GPUs.
- Coputational graphs – Pytorch provides an excellent platform which offers dynamic computational graphs. Thus, a user can change them during runtime.
  - It includes many layers as Torch.
  - It includes lot of loss functions.
  - It allows building networks whose structure is dependent on computation itself.
  - NLP: account for variable length sentences. Instead of padding the sentence to a fixed length, we create graphs with different number of LSTM cells based on the sentence's length

PyTorch

Fundamental Concepts of PyTorch

- Tensors
- Autograd
- Modular Structure
  - Models/ Layers
  - Datasets
  - Dataloader
- Visualization Tools like
  - TensorboardX (monitor training)
  - PyTorchViz (visualize computation graph)
- Various other functions
  - Loss (MSE.CE etc ...)
  - optimizers



Tensor

- Tensor?
  - PyTorch tensors are just like numpy arrays, but they can run on GPU.
  - Examples:

```
import numpy
# create a tensor
new_numpy = numpy.array([[I, 2], [3, 4]])
# create a 2 x 3 tensor with random values erapty_numpy = numpy.random.rand(2,3)
# create a 2 x 3 tensor with random vaiues between -I and 1 uniformjiunpy = numpy.reshape(np.random.uniform(-I,1,6),[2,3])
# create a 2 x 3 tensor with random values from a uniform distribution on the interval [0, 1) rand_numpy = numpy.random.rand(2, 3)
# create a 2 x 3 tensor of zeros zero_numpy = numpy.zeros([2, 3])
```

- And more operations like:
  Indexing, slicing, reshape, transpose, cross product, matrix product, element wise multiplication

```
import torch
# create a tensor
new_tensor = torch.Tensor([[I, 2], [3, 4]])
# create a 2 x 3 tensor with random values
empty_tensor = torch.Tensor(2, 3)
# create a 2 x 3 tensor with random values between -land 1
uniforn_tensor = torch.Tensor(2, 3).uniforra_(-I, 1)
# create a 2 x 3 tensor with random values from a uniform distribution on the
interval [0, 1)
rand_tensor = torch.rand(2, 3)
# create a 2 x 3 tensor of zeros zero_tensor = torch.zeros(2, 3)
```

Tensor(continued)

- Attributes of tensor “t”:
  `t=torch.randn(1)`

- `requires_grad` – making a trainable parameter
  - By default **False**
  - Turn on:
    `t. requires_grad_() or`
    `t=torch.randn(1,requires_grad=True)`

  - Accessing tensor value:
    `t.data`
  - Accessing tensor gradient:
    `t_grad`

- `grad_fn` – history of operations for autograd
  `t.grad_fn`

```
import torch N, 0 = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a = x * y b = a + z c=torch.sum(b)

c.backward()

print(c.grad_fn)

print(x.data)

print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
[0.7797, 0.1519, 0.7513, 0.7269], [0.8572, 0.1165,
0.8596, 0.2636]]) tensor([[0.6855, 0.9696,
0.4295, 0.4961], [0.3849, 0.0825, 0.7400,
0.0036], [0.8104, 0.8741, 0.9729, 0.3821]])
```

Loading data, Devices and CUDA

- Numpy arrays to PyTorch tensors
  - `torch.from_numpy(x_train)`
  - returns a cpu tensor!
- PyTorch tensor to numpy
  - `t.numpy()`
- Using GPU acceleration
  - `t.to()`
  - Sends to whatever device (cuda or cpu)
- Fallback to cpu if gpu is unavailable
  - `torch.cuda.is_available()`
- Check cpu/gpu tensor OR numpy array?
  - `tpye(t)` or `t.type()`
  - Returns
  - Numpy.ndarray
  - Torch.Tensor
  - CPU-torch.cpu.FloatTensor
  - GPU-torch.cuda.FloatTensor

```
import torch

import torch.optim as optim

import torch.nn as nn

from torchviz import make_dot

device - 'cuda' if torch.cuda.is_available() else 'cpu*

# Our data was in Numpy arrays, but we need to transform them into PyTorch's Tensors

# and then we send them to the chosen device
x_train_tensor - torch.from_numpy(x_train).float().to(device)
y_train_tensor - torch.from_numpy(y_train).float().to(device)

# Here we can see the difference - notice that .type() is more useful

# since it also tells us WHERE the tensor is (device)
print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

Autograd

- The **chain rule**, or **general product rule**, calculates any component of the joint distribution of a set of random variables using only conditional probabilities. This probability theory is used as a foundation for backpropagation and in creating Bayesian networks.
- This simple chain of probability and random variables is expressed as:
- P(A,B) = P(B | A) P(A)
- In mathematics, a **partial derivative** of a function of several variables is its derivative with respect to one of those variables, with the others held constant (as opposed to the total derivative, in which all variables are allowed to vary).
- The partial derivative of a function **f(x, y, ...)** with respect to the variable **x** is variously denoted by: $f'_x, f_x, \partial_x f,\ D_x f, D_1 f, \frac{\partial}{\partial x} f,$ or $\frac{\partial f}{\partial x}.$

Autograd

- In machine learning, backpropagation is a widely used algorithm for training feedforward neural networks.
- In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example.
- This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used.
- The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule
- It computes the gradient one layer at a time, iterates backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule.

```
def backward(ctx, grad_output):
                    """
                     In the backward pass we receive a Tensor
                    containing the gradient of the loss
                     with respect to the output, and we need to
                    compute the gradient of the loss
                    with respect to the input.
                    """
    input, = ctx.saved_tensors
    return grad_output * 1.5 * (5 * input ** 2 - 1)
```

Autograd

- Autograd
  - Automatic Differentiation Package
  - Don’t need to worry about partial differentiation, chain rule etc…
- backward() does that
  - loss.backward()
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - t.grad.zero()

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y b =a + z c=torch.sum(b)

c.backward()
```

Autograd (continued)

- Manual Weight Update - example

```
a = torch.randn(l, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(l, requires_grad=True, dtype=torch.float, device=device)

for epoch in range(n_epochs):

        yhat = a + b * x_train_tensor
        error = y_train_tensor - yhat
        loss = (error ** 2).mean()

        loss.backward()

        with torch.no_grad():
                a -= lr * a.grad
                b -= lr * b.grad

    a. grad.zero_()

    b. grad.zero_()

print(a, b)
```

Optimizer

- Optimizers (optim package)
  - Adam, Adagrad, Adadelta, SGD etc...
  - Optimizers are algorithms or methods used to change the attributes of the neural network such as weights and learning rate to reduce the losses.
  - Optimizers are used to solve optimization problems by minimizing the function.
  - The gradients are "stored" by the tensors themselves (they have a grad and a requires_grad attributes) once you call backward() on the loss.

  - optimizer.step()

  - After computing the gradients for all tensors in the model, calling optimizer.step() makes the optimizer iterate over all parameters (tensors) it is supposed to update and use their internally stored grad to update their values.

```
a = torch.randn(1, requires_grad=True, dtype=torcn.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torcn.float, device=device)
# Defines a SGD optimizer to update the parameters optimizer = #
optim.SGD([a, b], lr=lr)
for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
print(a, b)
```

Loss

- Loss is a prediction error of Neural Network. And the method to calculate the loss is called Loss Function.
- Loss is used to calculate the gradients. And gradients are used to update the weights of the Neural Net.
  - Various predefined loss functions to choose from

L1, MSE, Cross Entropy…

```
a = torch.randn(l, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(l, requires_grad=True, dtype=torch.float, device=device)
# Defines a MSE Loss function loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD([a, b], lr=lr)
for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    loss = loss_fn(y_train_tensor, yhat)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
print(a, b)
```

Model

- In PyTorch, a model is represented by regular Python class that inherits from the Module class
  - Two components
  - `init (self):` it defines the parts that make up the model – in our case, two parameters, a and b
  - `forward (self,x)`: it performs the actual computation, that is, it outputs a prediction, given the input x

Model (Example)

Example:

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super () .__init__ ()
        #To make "a" and "b" real parameters of the modeLj we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(l, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(l, requires_grad=True> dtype=torch.float))
    def forward(self, x):
        #Computes the outputs / predictions
        return self.a + self.b * x
```

- Properties:
  - model=ManualLinearRegression()
  - model.state_dic() – returns a dictionary of trainable parameters with their current values
  - model.parameters() - returns a dictionary of trainable parameters in the model
  - model.train() or model.eval()

Putting things together

- Sample Code in practice:

```
# Now we can create a model and send it at once to the device model =
ManualLinearRegression().to(device)

# We can also inspect its parameters using its state_dict print(model.state_dict())

loss_fn = nn.MSELoss(reduction='mean') optimizer = optim.SGD(model.parameters(lr=lr)

for epoch in range(n_epochs): model.train()

    yhat = model(x_train_tensor)

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()

print(model.state_dict())
```

Complex Models

- Complex Model Class
- Predefined “layer” modules

```
class LayerLinearRegression(nn.Module):
def  __init__(self):
super  () __init__ ()
# Instead of our custom parameters, we use a Linear Layer with single input and single output self.linear = nn.Linear(1, 1)
def forward(self, x):
# Now it only takes a call to the Layer to make predictions
return self.linear(x)
```

- “Sequential” layer modules

```
class LayerLinearRegression(nn.Module):
def   __init__ (self):
super() __init__ ()
# Instead of our custom parameterst vje use a Linear Layer uith singLe input and single output self.seq_linear = nn.Sequential(nn.Linear(l,
2)<,nn.RELU(),nn.Linear(2,l))
def forward(self, x):
# Now it only takes a call to the Layer to make predictions
return self.seq_linear(x)
```

III
NN gradients, activations, cost functions and metrics, Backpropagation

Data Science Academy

How Neural Network works?
Forward Propagation

Data Science Academy

Steps of NN training

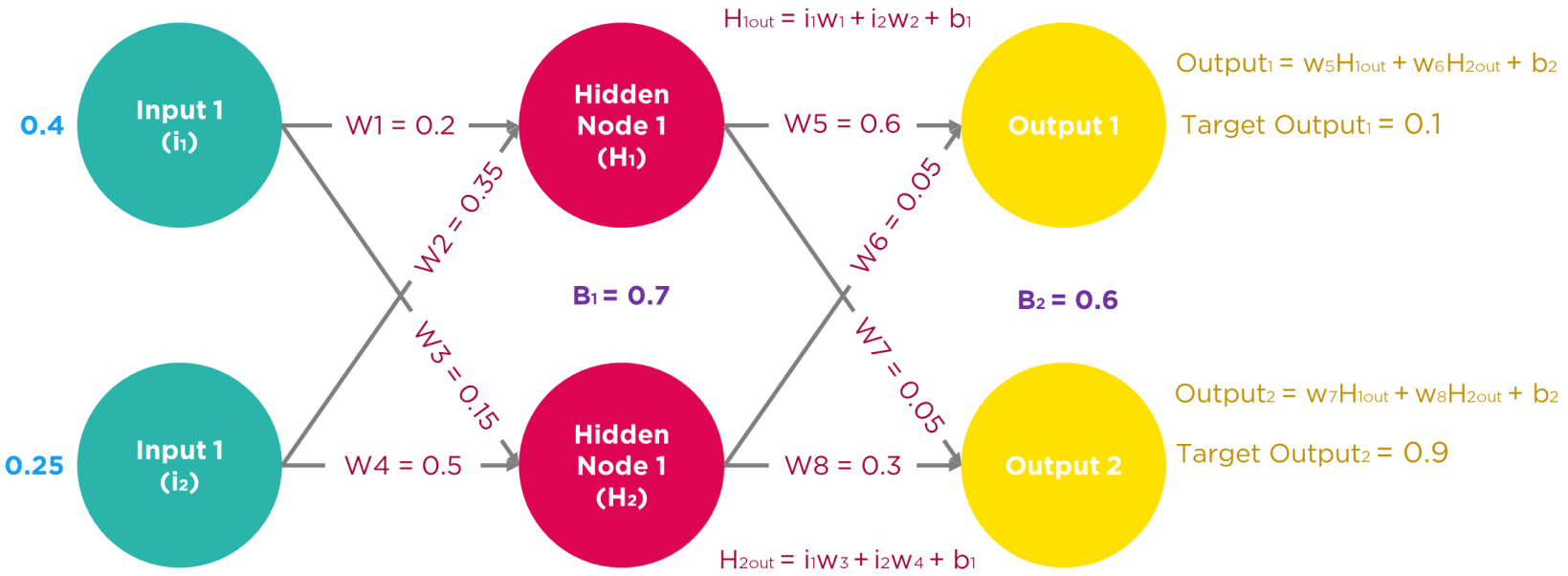
- Random Initialization
- Forward Propagation
- Loss Functions
- Backpropagation
- Gradient Descent
- Stochastic Gradient Descent, Mini Batch Gradient Descent

Overview

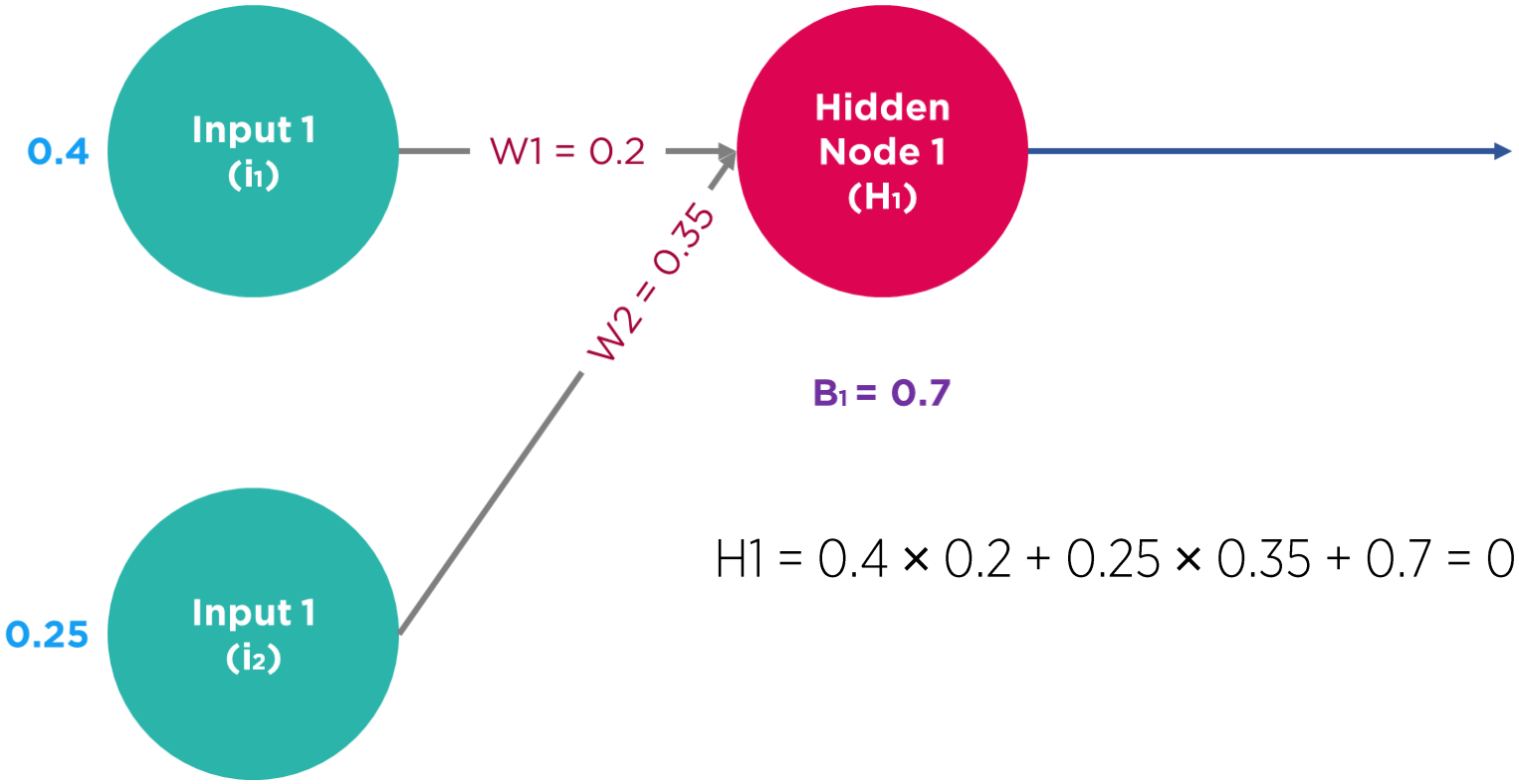
- The Random Initialization logic states that Neural Network begins with random weights to reach target output.
- Forward Propagation – the process of inputs to reach outputs
- That Loss Functions (such as MSE) quantify how much error our current weights produce.
- That Backpropagation can be used to determine how to change the weights so that our loss is lower.
- An Epoch occurs when the full set of our training data is passed/forward propagated and then backpropagated through our neural network.
- This process of optimizing or lowering the weights is called Gradient Descent.
- Gradient Descent has some efficient types – Stochastic Gradient Descent and Mini Batch Gradient Descent
- The batch size is a number of samples processed before the model is updated. Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients.

Using actual values (random)

- Random Initialization. Like in genetic algorithms and evolution theory, neural networks can start from anywhere. Thus, a random initialization of the model is a common practice. Random Initialization states that Neural Network begins with random weights to reach target output. Through an iterative learning process (such as Gradient Descent), model will reach the ideal model.



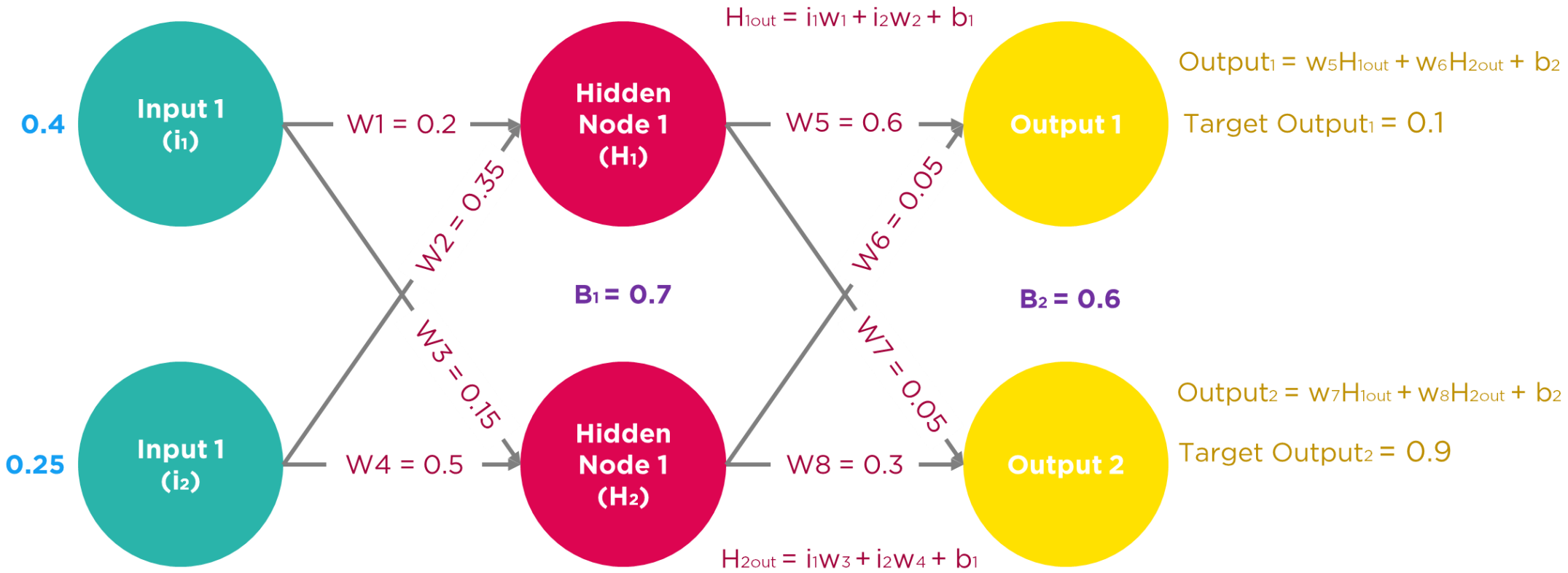
Looking at a single Node/Neuron



$H_1 = w_1 i_1 + w_2 i_2 + b_1$

$H1 = 0.4 \times 0.2 + 0.25 \times 0.35 + 0.7 = 0.08 + 0.0875 + 0.7 = 0.8675$

Steps for all output values



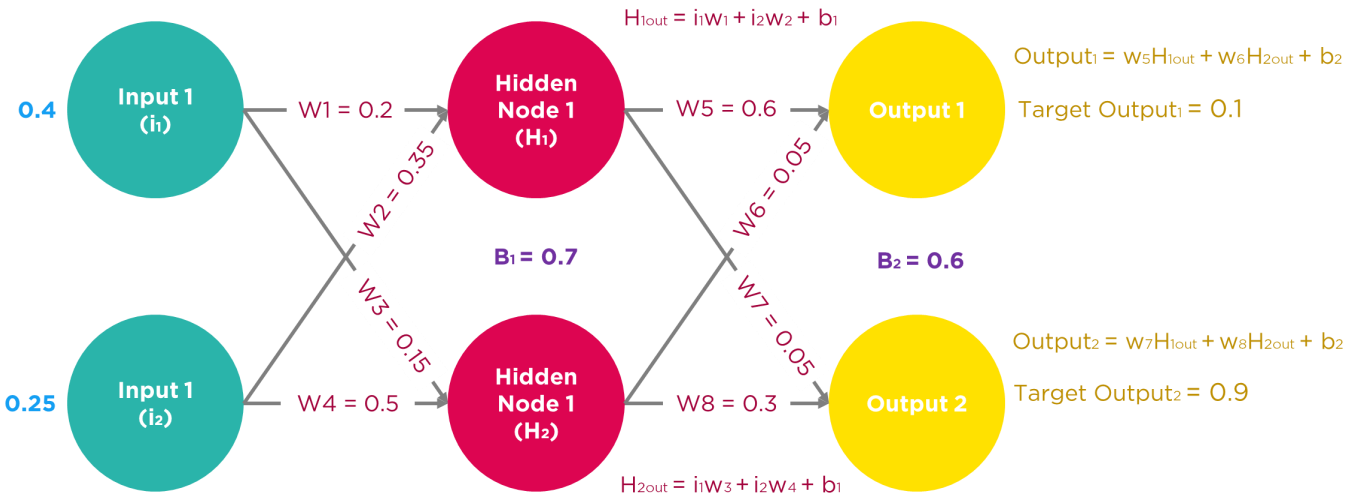
$$Output = w_n i_n + b$$

Example : How Neural Networks Work

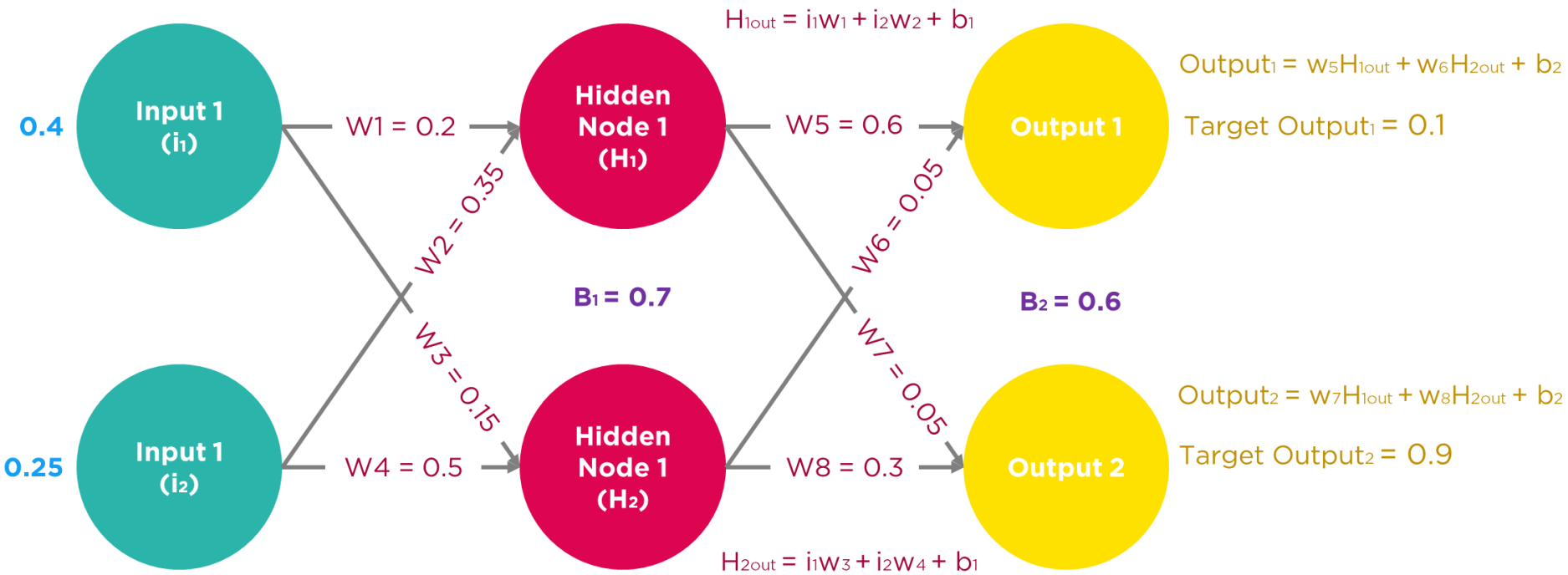
In reality these connections are simply formulas that pass values from one layer to the next

Output of Nodes = $w_n i_n + b$

- $Output_1 = w_5H_1 + H_2w_6 + b_2$
- $H_1 = i_1w_1 + i_2w_2 + b_1$
- $H_2 = i_1w_3 + i_2w_4 + b_1$
- $H_2 = 0.4 \times 0.15 + 0.25 \times 0.5 + 0.7 = 0.06 + 0.125 + 0.7 = 0.885$



Getting our final outputs



- $Output_1 = w_5H_1 + H_2w_6 + b_2$
- $Output_1 = (0.6 \times 0.8675) + (0.05 \times 0.885) + 0.6 = 0.5205 + 0.04425 + 0.6 = 1.16475$
- $Output_2 = 0.43375 + 0.2655 + 0.6 = 1.29925$

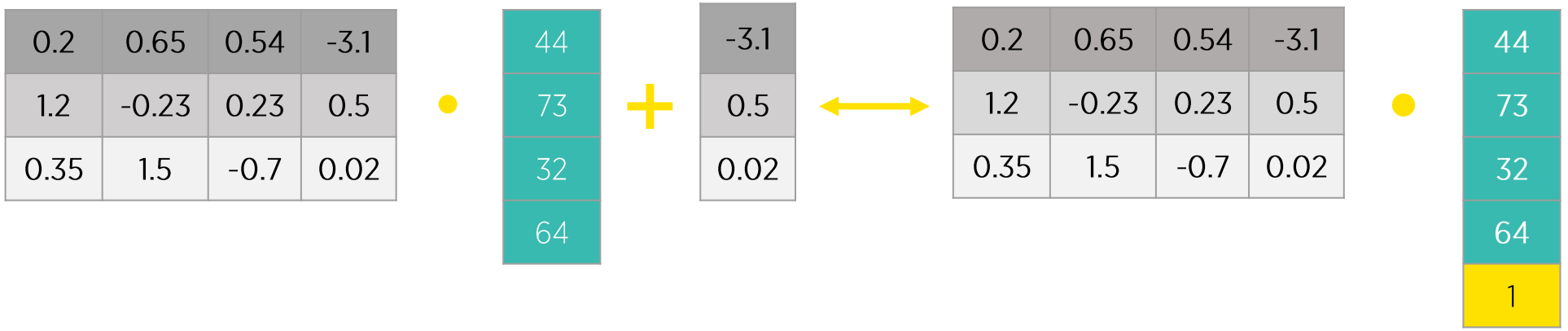
What does this tell us?

- Our initial default random weights (w and b) produced very incorrect results
- Feeding numbers through a neural network is a matter of simple matrix multiplication
- Our neural network is still just a series of linear equations

The Bias Trick

Making our weights and biases as one

- Now is a good a time as any to show you the bias trick that is used to simply our calculations.



- x_i is our input values, instead of doing a multiplication then adding of our biases, we can simply add the biases to our weight matrix and add an addition element to our input data as 1.
- This simplifies our calculation operations as we treat the biases and weights as one.
- NOTE: This makes out input vector size bigger by one i.e if x_i had 32 values, it will now have 33.

Loss Functions

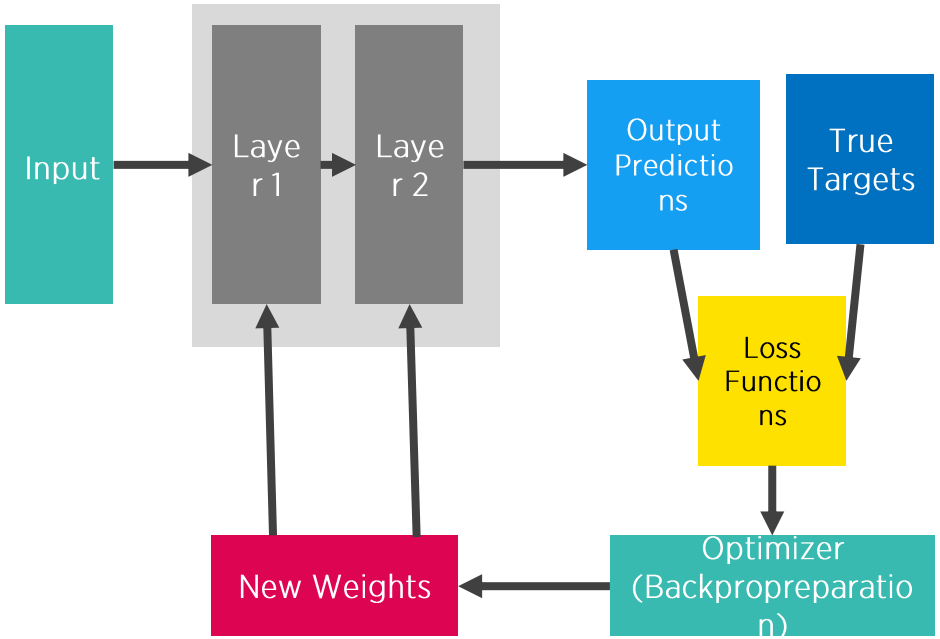
- Loss functions are integral in training Neural Nets as they measure the inconsistency or difference between the predicted results & the actual target results.
- They are always positive and penalize big errors well
- The lower the loss the ‘better’ the model
- There are many loss functions, Mean Squared Error (MSE) is popular
- MSE = (Target − Predicted)2

Loss Function

Training step by step:

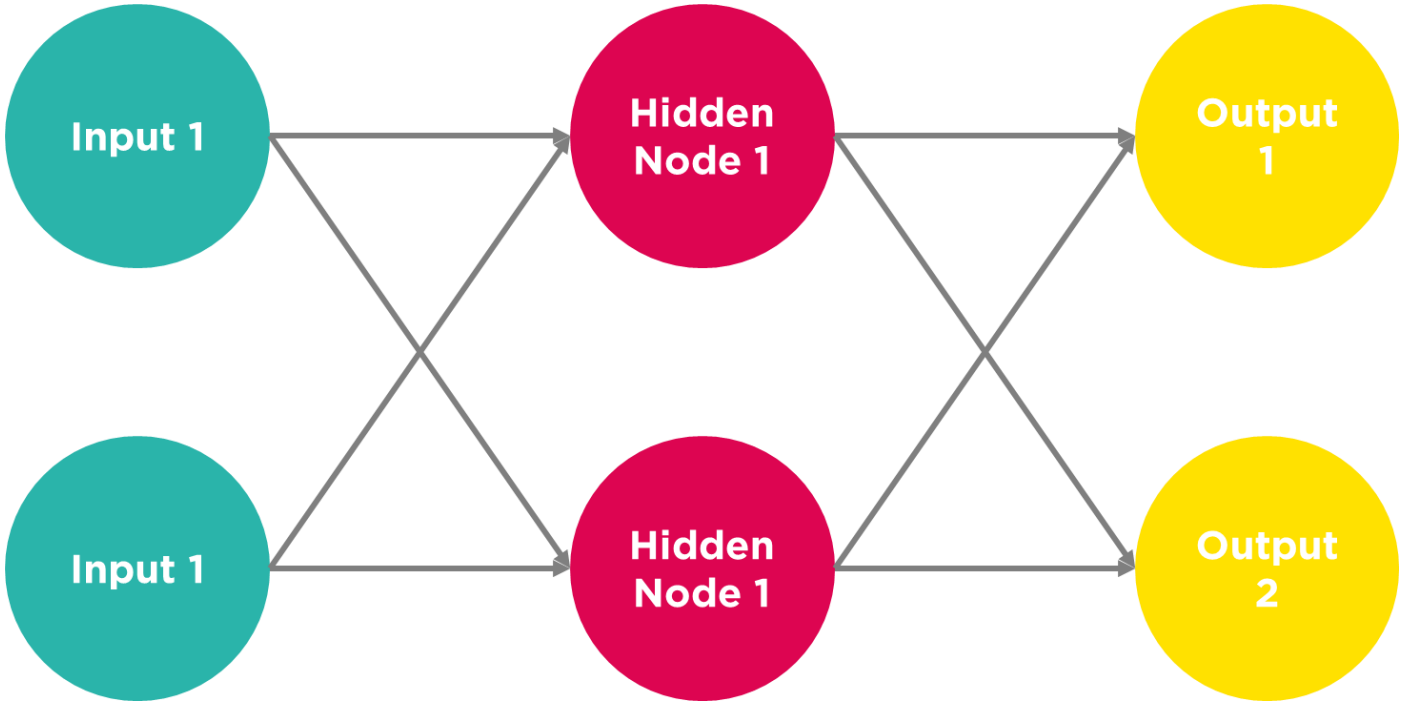
- Initialize some random values for our weights and bias
- Input a single sample of our data
- Compare our output with the actual value it was supposed to be, we'll be calling this our Target values.
- Quantify how 'bad' these random weights were, we'll call this our Loss.
- Adjust weights so that the Loss lower
- Keep doing this for each sample in our dataset
- Then send the entire dataset through this weight 'optimization' program to see if we get an even lower loss
- Stop training when the loss stops decreasing.

Training Process Visualized



Loss Function

- Our simple 2 input, 2 output and 1 hidden layer network has X parameters.
- Input Nodes x Hidden Layers + Hidden Layers x Output + Biases



Types of Loss Functions

There are many types of loss functions such as:

- L1 - Least Absolute Deviations
- L2 - Least Square Errors
- Cross Entropy – Used in binary classifications
- Mean Absolute Percentage Error
- Mean Squared Logarithmic Error
- Hinge Loss
- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)

Formulas:

$$S = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

$$S = \sum_{i=1}^{n} y_i - f(x_i, \beta).$$

$$H(T, q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

$$\mathrm{MAPE} = \frac{1}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right|$$

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (\log(y_i + 1) - \log(\hat{y}_i + 1))^2$$

$$\ell(y) = \max(0, 1 - t \cdot y)$$

$$\mathrm{ME} = \frac{\sum_{i=1}^{n} y_i - x_i}{n}$$

$$\mathrm{MSPE} = \frac{1}{q} \sum_{i=n+1}^{n+q} (Y_i - \hat{Y}_i)^2$$

Types of Loss Functions

Implementation of Loss Functions using Numpy

```
# L1 - Least Absolute Deviations
def mad(data, axis=None):
    return mean(absolute(data - mean(data, axis)), axis)

# L2 - Least Square Errors
m, c = np.linalg.lstsq(A, y, rcond=None)[0]

# Cross Entropy
def cross_entropy(predictions, targets, epsilon=1e-12):
predictions = np.clip(predictions, epsilon, 1. - epsilon)
    N = predictions.shape[0]
    ce = -np.sum(targets*np.log(predictions+1e-9))/N
    return ce

# Mean Absolute Percentage Error
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

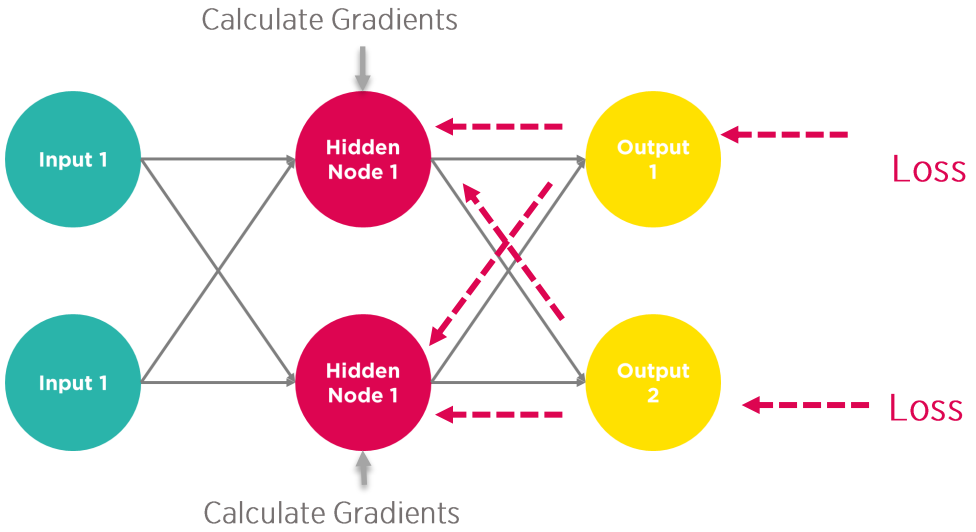
```
#Mean Squared Logarithmic Error
def msle(h, y):
    return np.square(np.log(h + 1) - np.log(y + 1)).mean()

# Hinge Loss
A = (y*X.dot(w)).ravel()
B = (X*y[:,np.newaxis]).ravel()
C = A * np.sign(np.maximum(0, 1-B))

# Mean Absolute Error (MAE)
def mean_absolute_error(x, y, axis=None):
 return mae_core(x, y, axis=axis, mode='mean')

# Mean Squared Error (MSE)
def mean_squared_error(Y, YH):
    return np.square(Y - YH).mean()
```

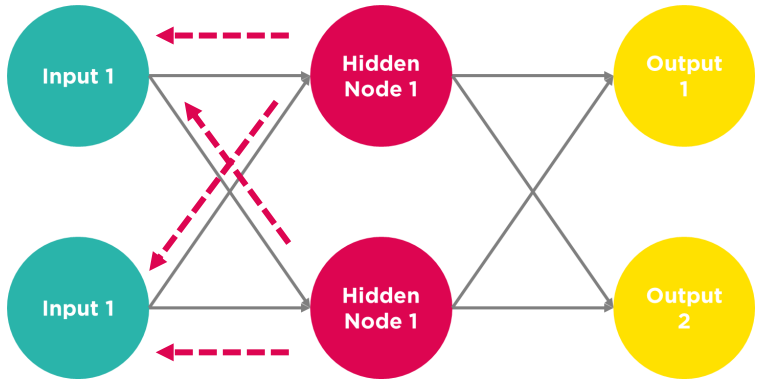
Loss Functions

Name	Formula	Usage/ Characteristics
Mean Squared Error or MSE	$\frac{1}{n}\sum_{i=1}^{n}(yi - \hat{y}i)^2$	1. One of the most popular 2. Used when predicted values are small
Mean Squared Logarithmic Error	$\frac{1}{n}\sum_{i=1}^{n}(\log(p_i + 1) - \log(a_i + 1))^2$	Used when predicted values are large and we do not want to penalize the large error
L2	$\sum_{i=1}^{n}(yi - \hat{y}i)^2$	Same as MSE without the mean
Mean Absolute Error	$\frac{1}{n}\sum_{i=1}^{n}\lvert yi - \hat{y}i\rvert$	1. Robust but unstable solution 2. Gradient is difficult to calculate
Mean Absolute Percentage Error	$\frac{100\%}{n}\sum_{i=1}^{n}\left\lvert\frac{y_i - \hat{y}_i}{y_i}\right\rvert$	1. For zero values could pose a challenge 2. Does not always remain below 100%
L1	$\sum_{i=0}^{n}\lvert y_i - \hat{y}_i\rvert$	Same as MAE without the mean calculation

Backpropagation

- It's a method of executing Gradient Descent or weight optimization so that we have an efficient method of getting the lowest possible loss.
- Backpropagation tells us how much would a change in each weight affect the overall loss.
- We obtain the total error at the output nodes and then propagate these errors back through the network using Backpropagation to obtain the new and better gradients or weights.

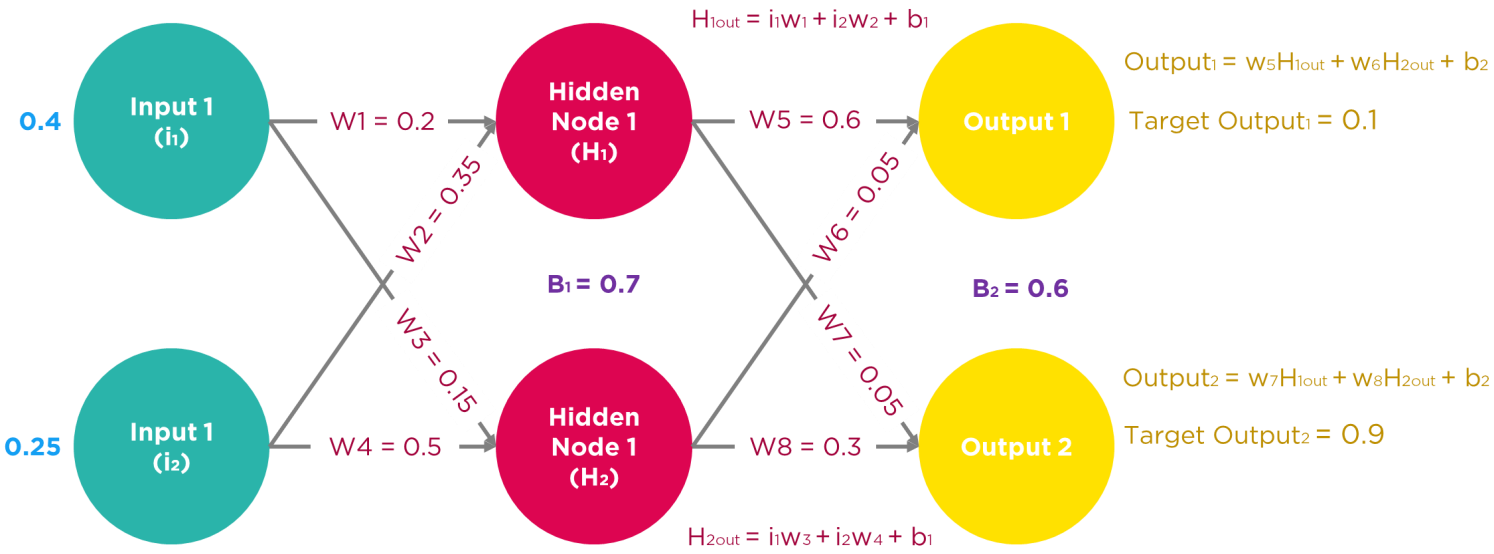




Backpropagation: Revisiting out Neural Net

Using the MSE obtained from Output 1, Backpropagation allows us know:

- If changing w5 from 0.6 by a small amount, say to 0.6001 or 0,5999,
- Whether our overall Error or Loss has increased or decreased.



Epochs

- You may have seen or heard me mention Epochs in the training process, so what exactly is an Epoch?
- An Epoch occurs when the full set of our training data is passed/forward propagated and then backpropagated through our neural network.
- After the first Epoch, we will have a decent set of weights, however, by feeding our training data again and again into our Neural Network, we can further improve the weights. This is why we train for several iterations/epochs.

Gradient Descent

- We use a process called Gradient Descent to minimize the cost or error function:

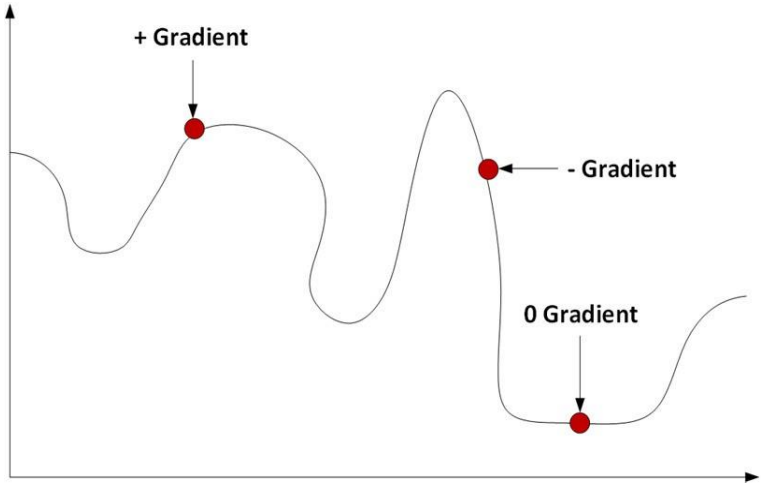
Error (n, b) = $\frac{1}{N}\sum_{i=1}^{N}(Actual\ output - Predicted\ Output)$

Cost function = $J(8_0,8_1)=\frac{1}{2}\sum_{i=1}^{n}(h_8(x_i)-y_i)^2$

- We treat m and b as 8_0 and 8_1
- The above equation simply tells us how wrong the line is by measuring how far the predicted value of $\boldsymbol{h}$ x_i is from the actual value y_i
- We square the error so that we don't end up with negative values
- We divide by 2 to make updating our parameters easier

Gradient Descent

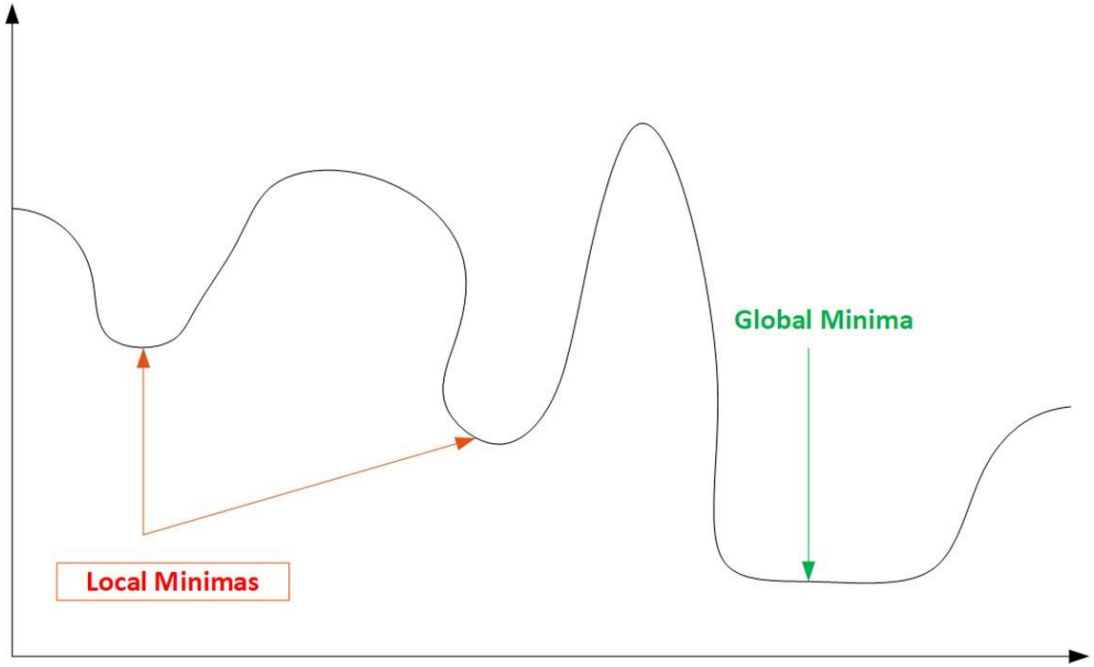
- By adjusting the weights to lower the loss, we are performing gradient descent. This is an 'optimization' problem.
- Backpropagation is simply the method by which we execute gradient descent
- Gradients (also called slope) are the direction of a function at a point, it's magnitude signifies how much the function is changing at that point.



- By adjusting the weights to lower the loss, we are performing gradient descent.
- Gradients are the direction of a function at a point

Gradient Descent

- Imagine our global minima is the bottom of this rough bowl. We need to traverse through many peaks and valleys before we find it

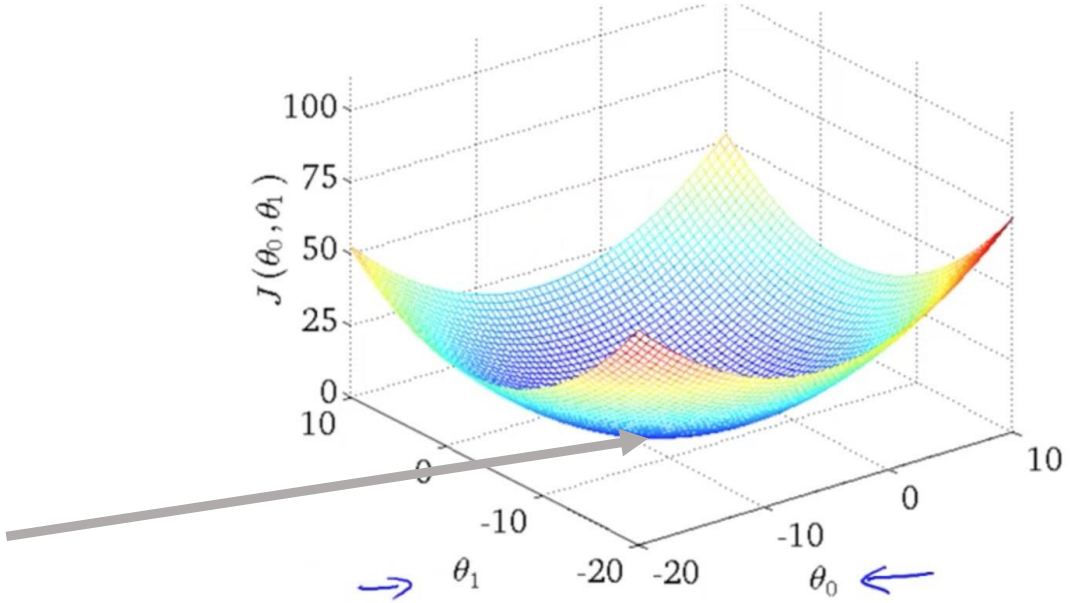


What is Gradient Descent

- $J(\theta_0,\theta_1) = \frac{1}{2}\sum_{i=1}^{n}(h_\theta(x_i) - y_i)^2$
- Recall our Cost Function equation (J),remember it gives us the error based on whatever values of as θ_0 and θ_1 weuse.
- How do we know what values to use that gives us the lowest cost?

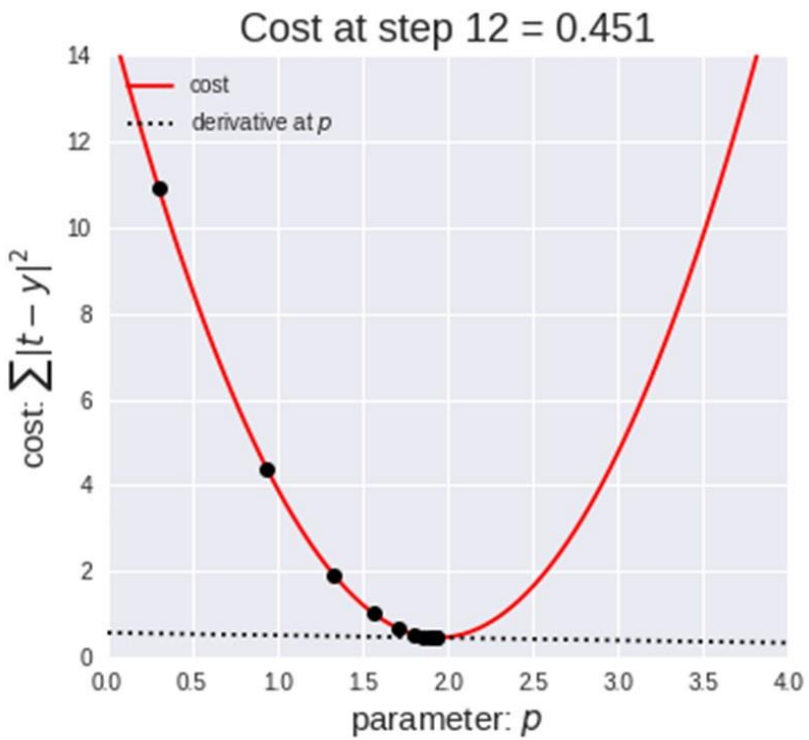
How Gradient Descend finds lowest cost?

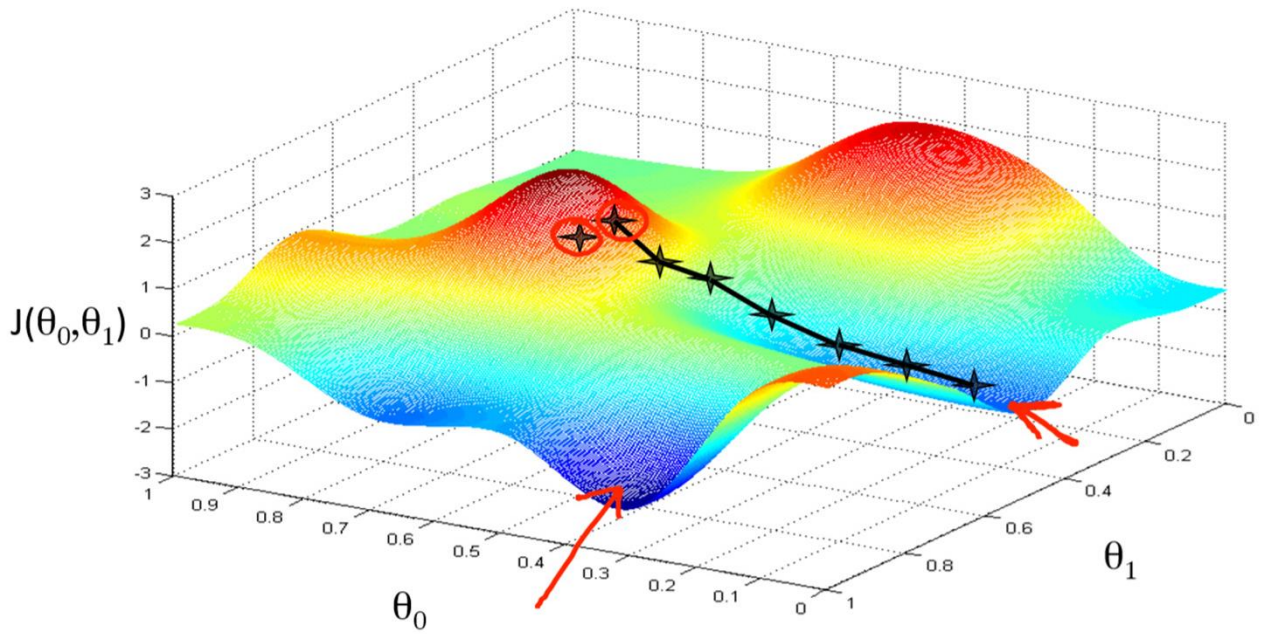
We want the 8_0 and 8_1 at this point



- Gradient Descent is the method by which we find this point where the gradient is zero

Gradient Descent Method Visualized





How Gradient Descent works

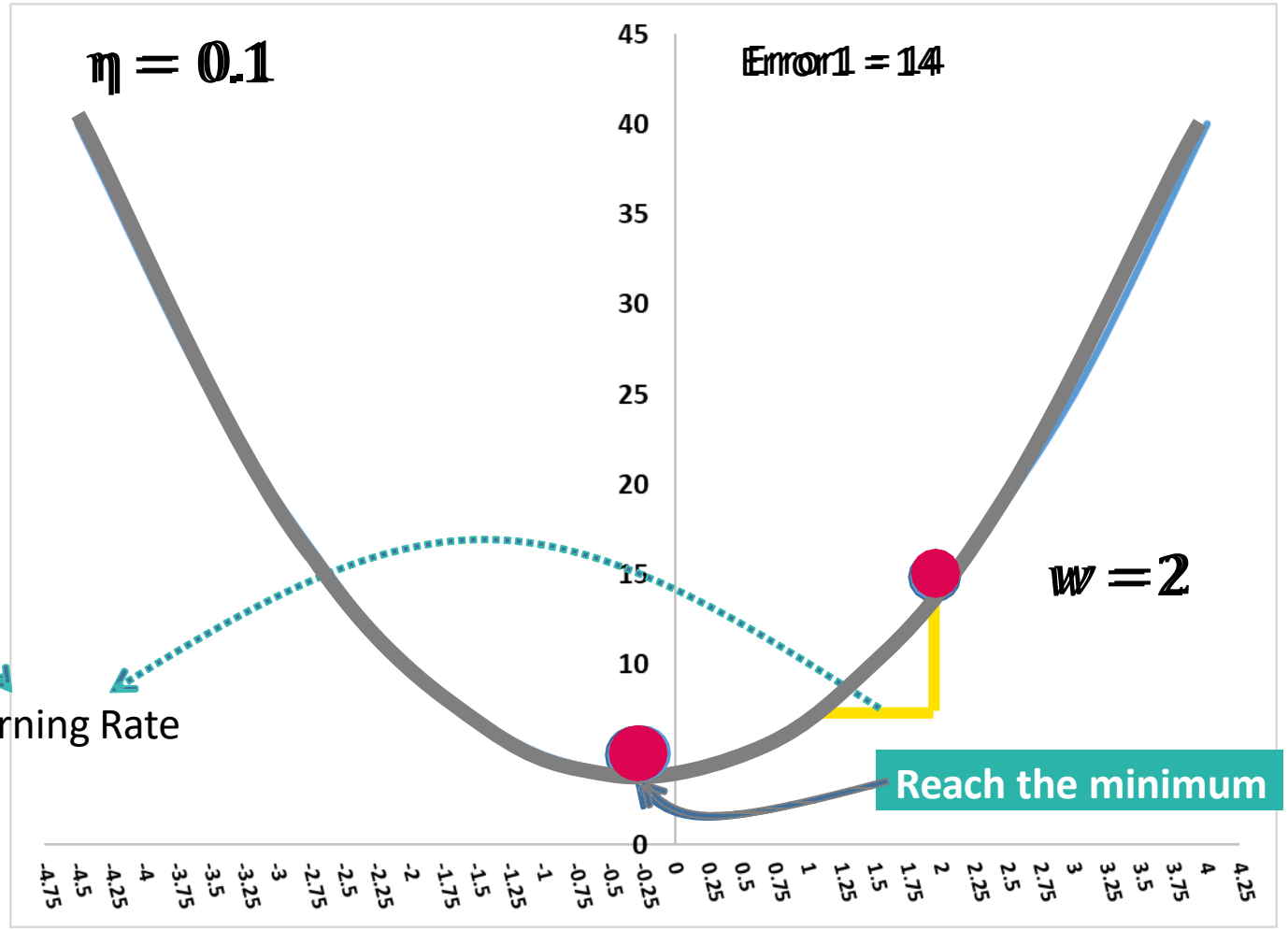
$f(w) = 2w^2 + w + 4$

$f'(w) = 4w + 1$

Loss = $2 * 22 + 2 + 4 = 14$

$w_{i+1} = w_i - \eta * f'(w)$

Learning Rate



How Gradient Descent works

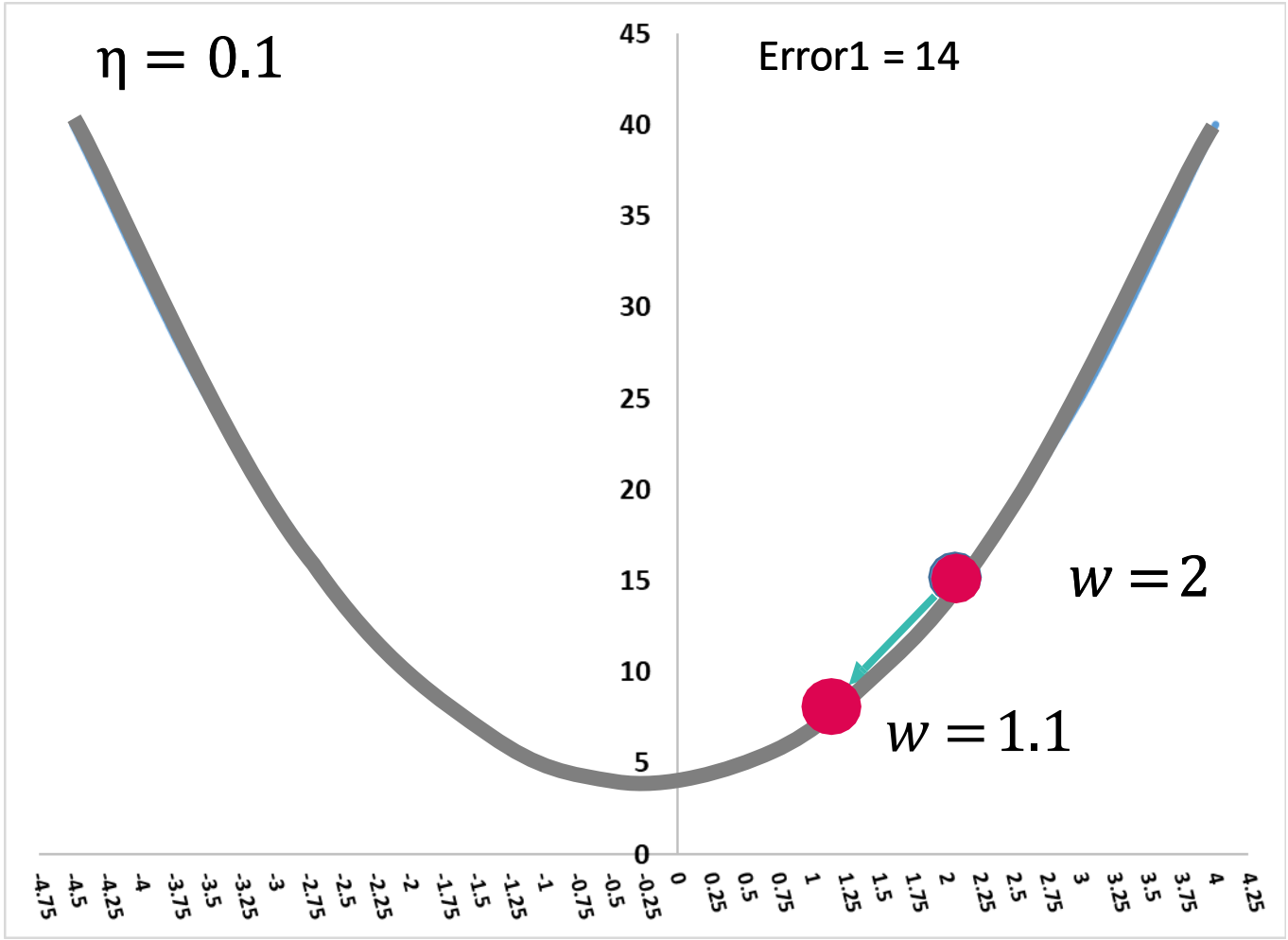
$$f(w) = 2w^2 + w + 4$$

$$f'(w) = 4w + 1$$

$$\text{Loss} = 2 * 22 + 2 + 4 = 14$$

$$wi+1 = \quad wi - \quad \eta * f'(w)$$

$$wi+1 = 1.1$$



How Gradient Descent Example works

$f(w) = 2w2 + w + 4$

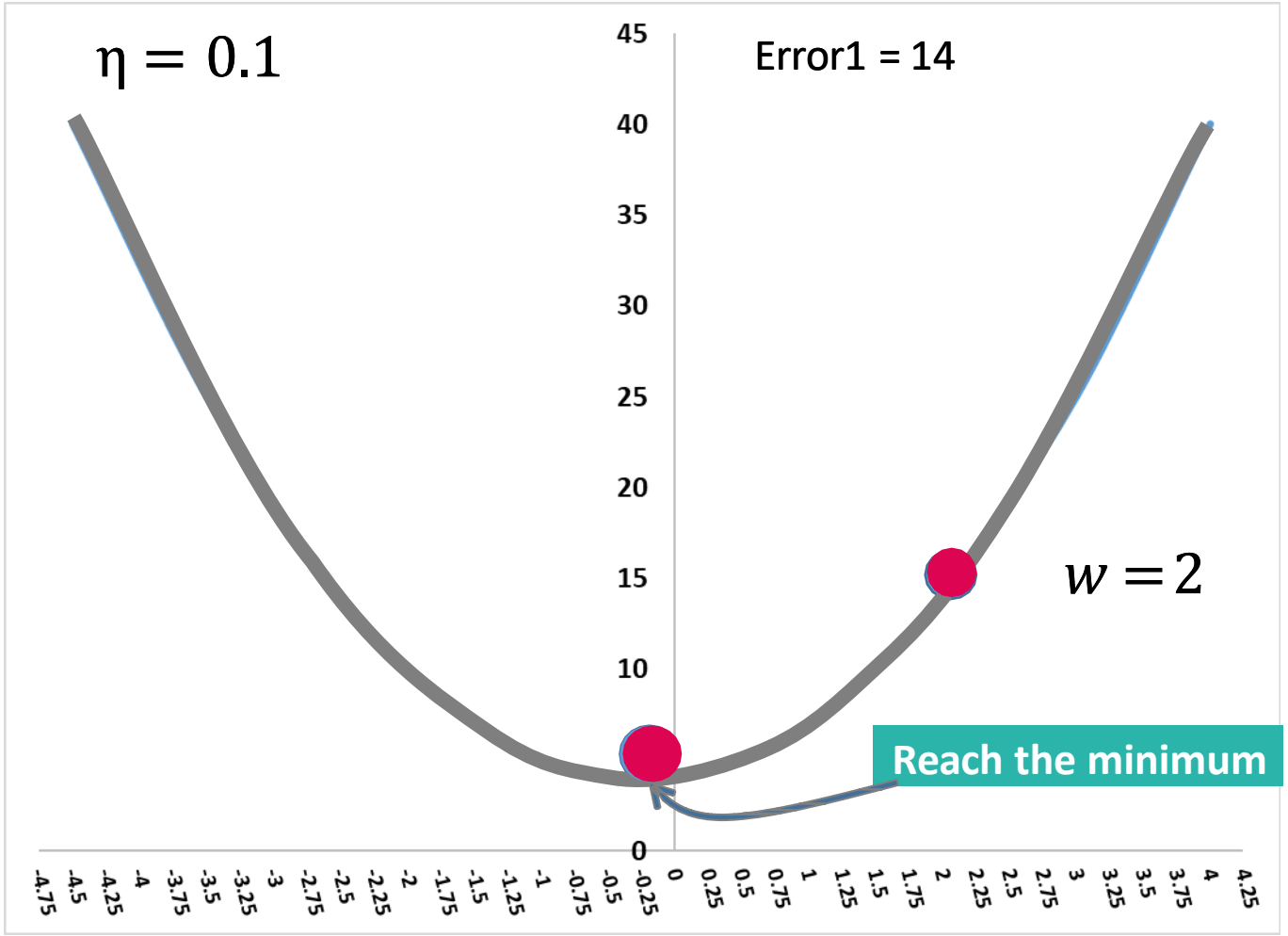
$f'(w) = 4w + 1$

$\text{Loss} = 2 * 2^2 + 2 + 4 = 14$

$w_{i+1} = w_i - \eta * f'(w)$

$w_{i+1} = 2 - (0.14 * 2 + 1)$

$= 2 - 0.9$



How Gradient Descent Example works

$$f(w) = 2w2 + w + 4$$

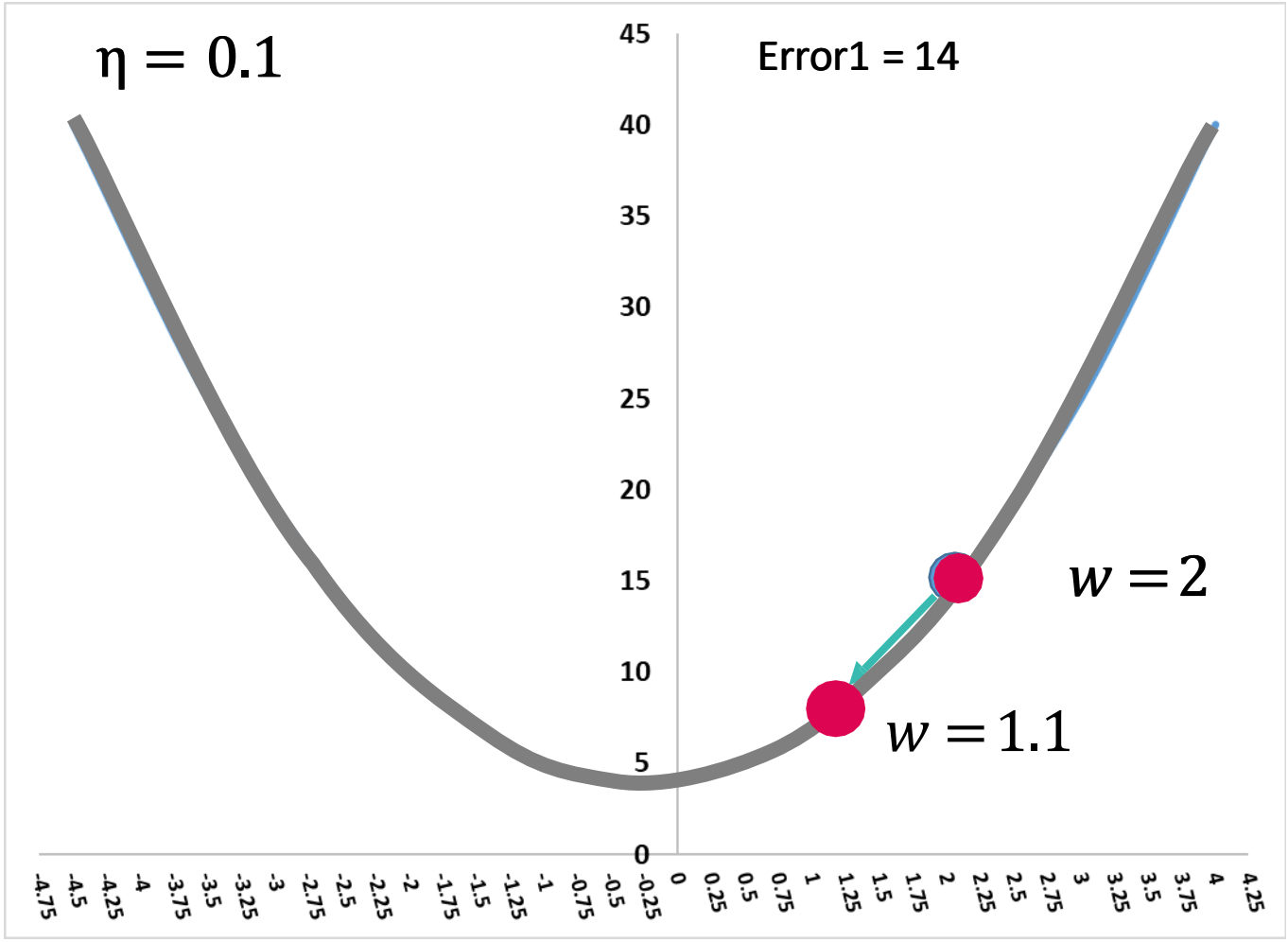
$$f'(w) = 4w + 1$$

Iteration 1

Loss = $2 * 22 + 2 + 4 = 14$

$$w_{i+1} = w_i - \eta * f'(w)$$

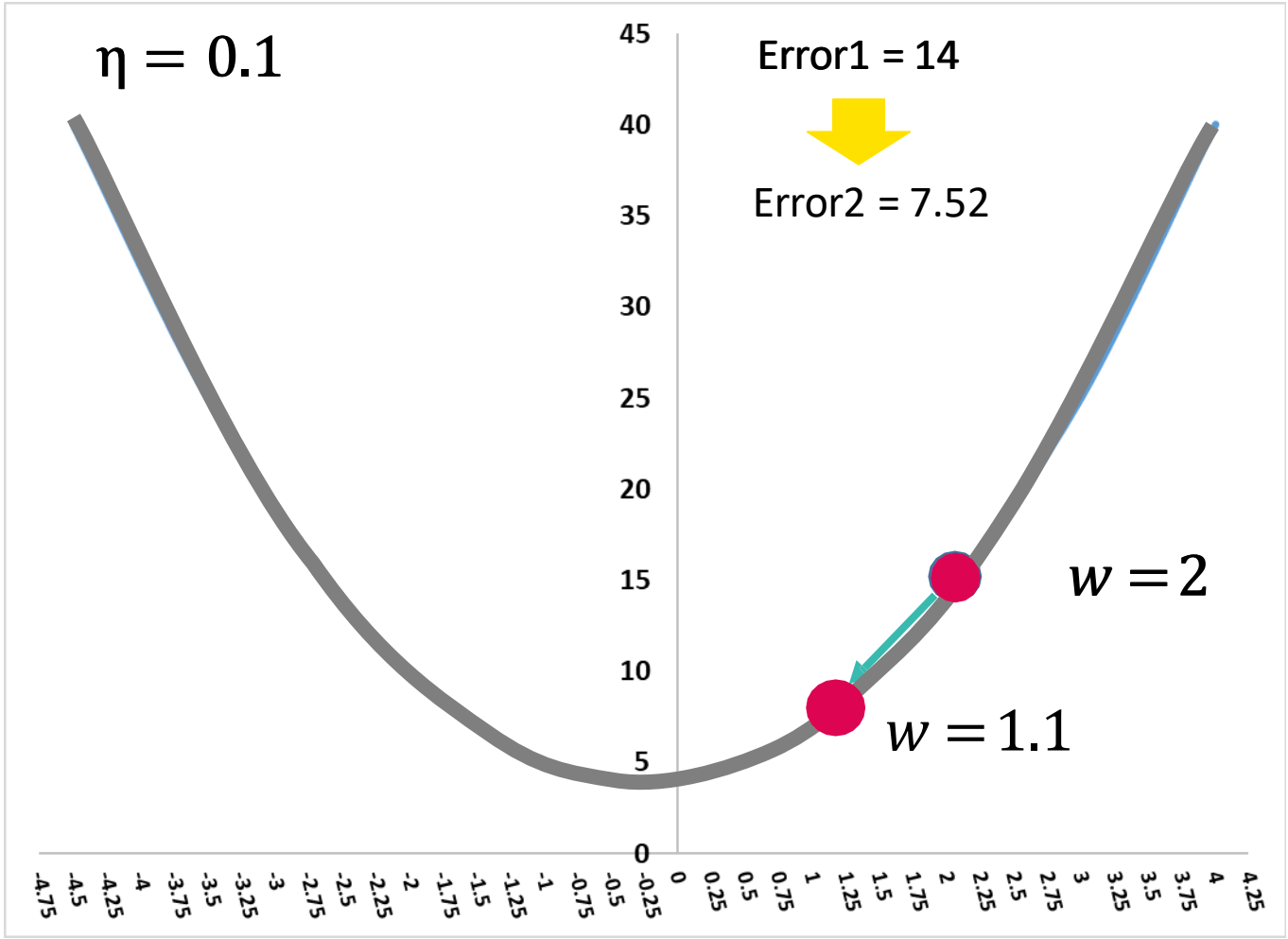
$w_{i+1} = 1.1$



How Gradient Descent Example works

$$f(w) = 2w^2 + w + 4$$

$$f'(w) = 4w + 1$$



How Gradient Descent Example works

$$f(w) = 2w^2 + w + 4$$

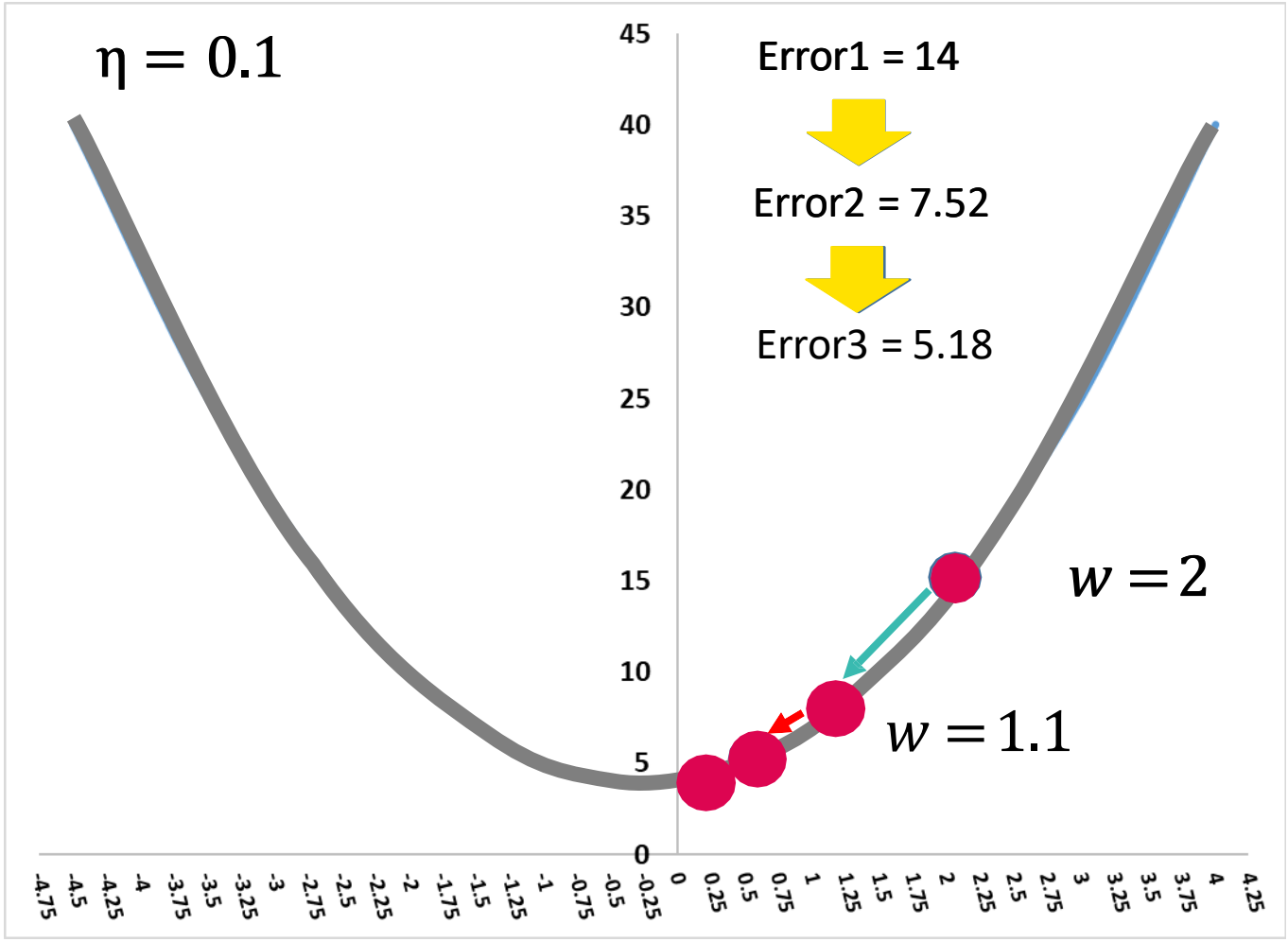
$$f'(w) = 4w + 1$$

Iteration 3

$$\text{Loss} = 2 * 0.56^2 + 0.56 + 4 = 5.18$$

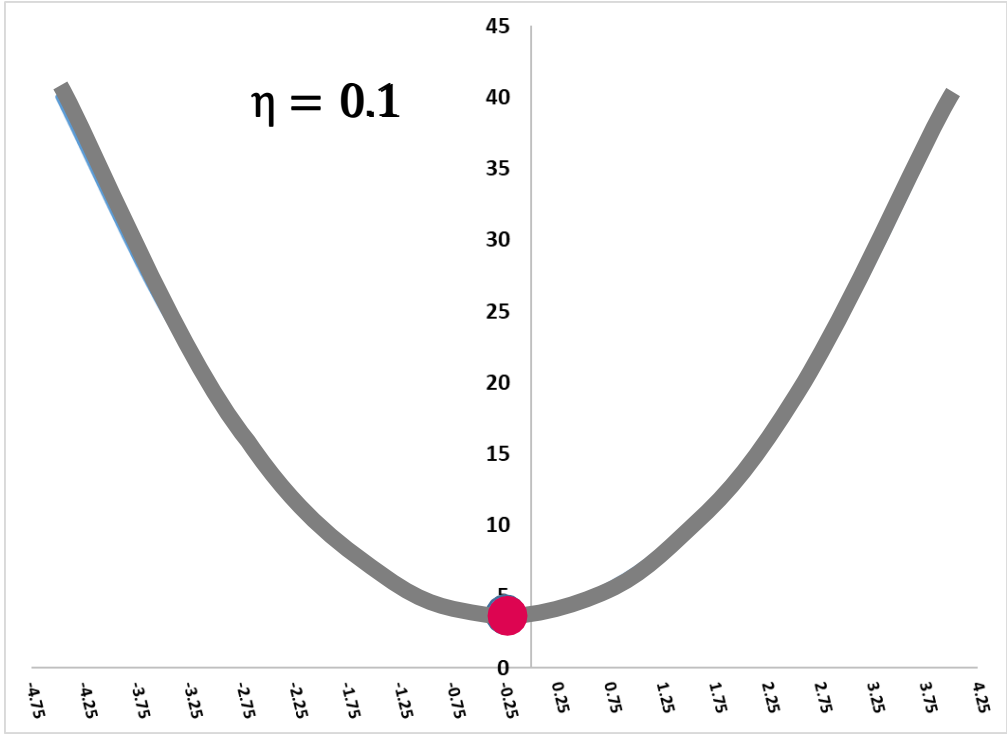
$$w_{i+1} = w_i - \eta * f'(w)$$

$$w_{i+1} = 0.236$$

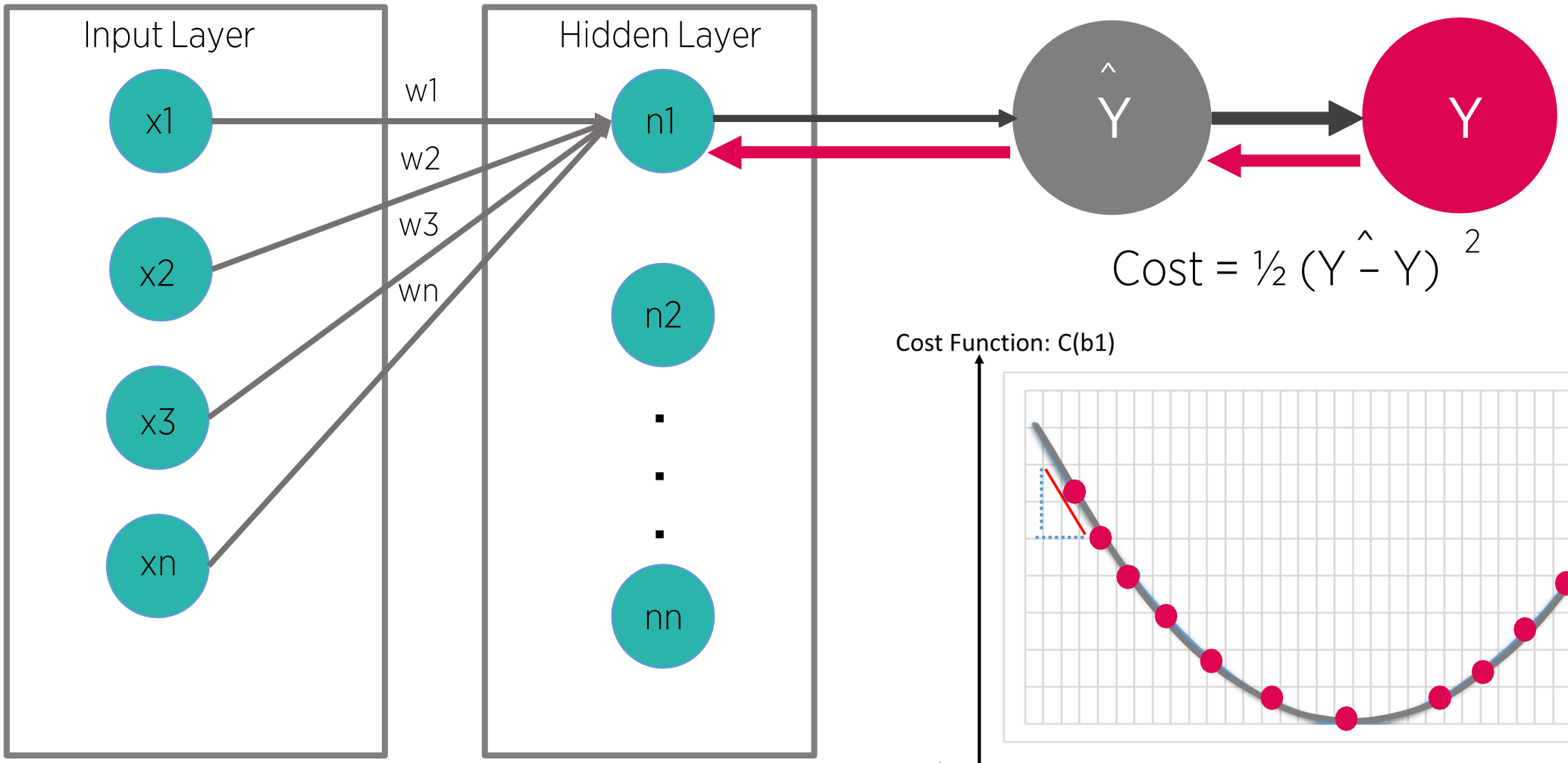


How Gradient Descent Example works

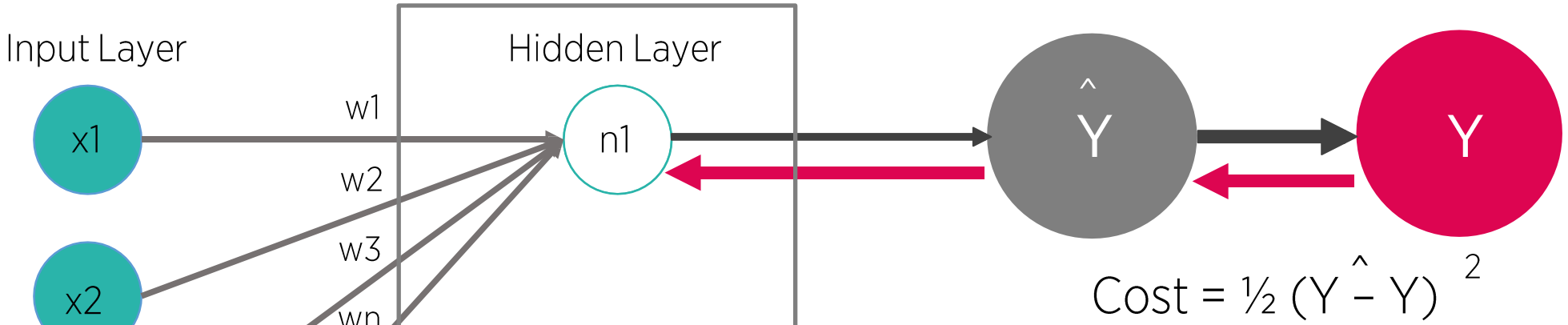
Iteration	W	f (W)	f'(W)	$w_{i+1} = w_i - \eta * f'(w)$
1	2.00000	14.00000	9.00000	1.10000
2	1.10000	7.52000	5.40000	0.56000
3	0.56000	5.18720	3.24000	0.23600
4	0.23600	4.34739	1.94400	0.04160
5	0.04160	4.04506	1.16640	-0.07504
6	-0.07504	3.93622	0.69984	-0.14502
7	-0.14502	3.89704	0.41990	-0.18701
8	-0.18701	3.88293	0.25194	-0.21221
9	-0.21221	3.87786	0.15117	-0.22733
10	-0.22733	3.87603	0.09070	-0.23640
11	-0.23640	3.87537	0.05442	-0.24184
12	-0.24184	3.87513	0.03265	-0.24510
13	-0.24510	3.87505	0.01959	-0.24706
14	-0.24706	3.87502	0.01175	-0.24824
15	-0.24824	3.87501	0.00705	-0.24894
16	-0.24894	3.87500	0.00423	-0.24937
17	-0.24937	3.87500	0.00254	-0.24962
18	-0.24962	3.87500	0.00152	-0.24977
19	-0.24977	3.87500	0.00091	-0.24986
20	-0.24986	3.87500	0.00055	-0.24992
21	-0.24992	3.87500	0.00033	-0.24995



Gradient Descent in Neural Networks

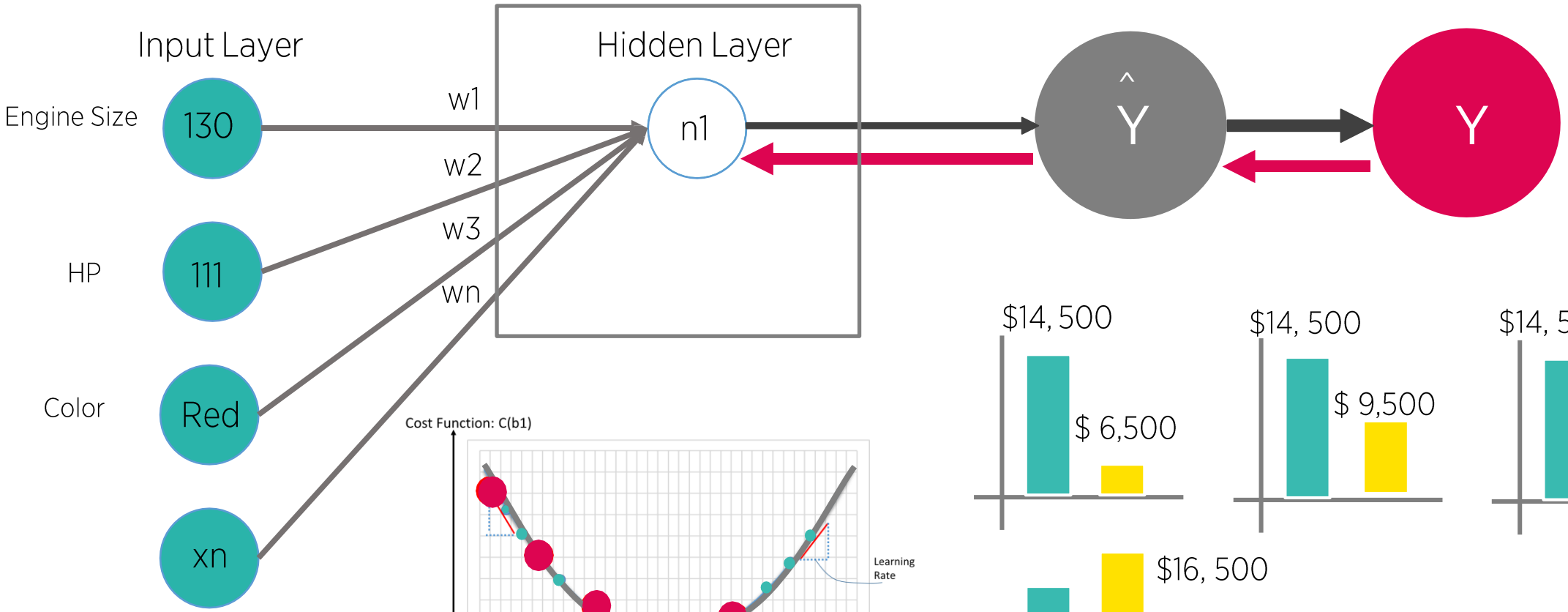


Gradient Descent in Neural Networks



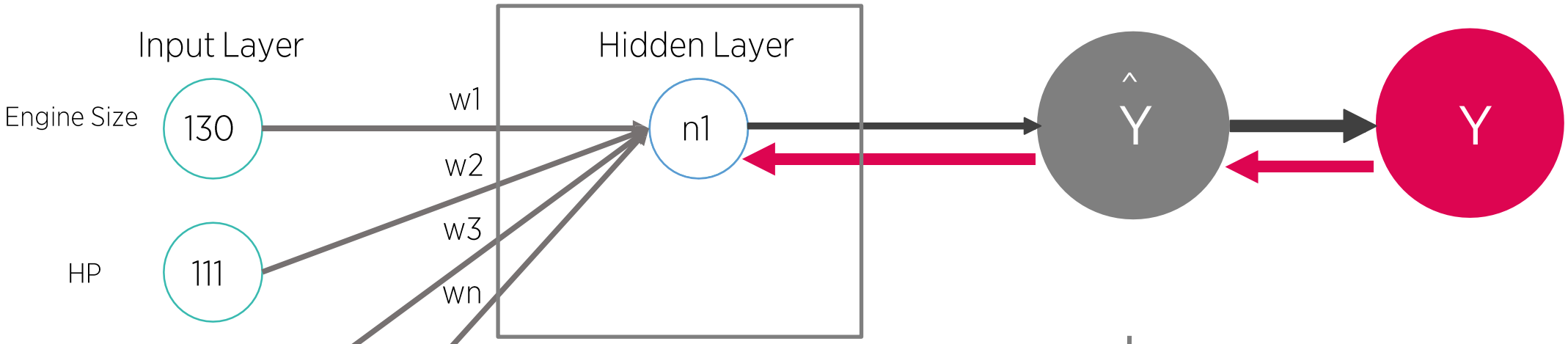
Engine Size X1	HP X2	Colour X3	Price Y
130	111	Black	**$ 13,495**
130	111	Red	**$ 14,500**
152	154	Red	**$ 16,500**
109	102	Black	**$ 13,950**

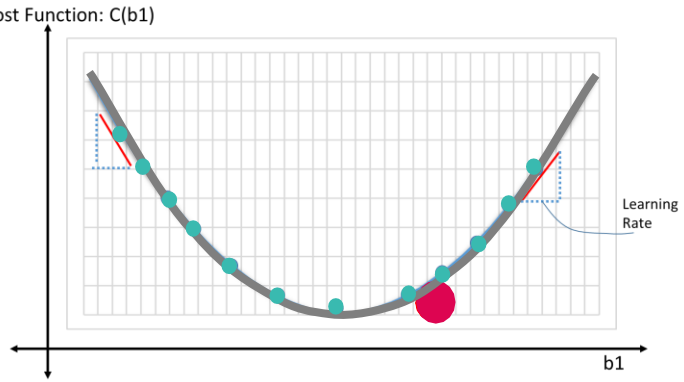
Gradient Descent in Neural Networks



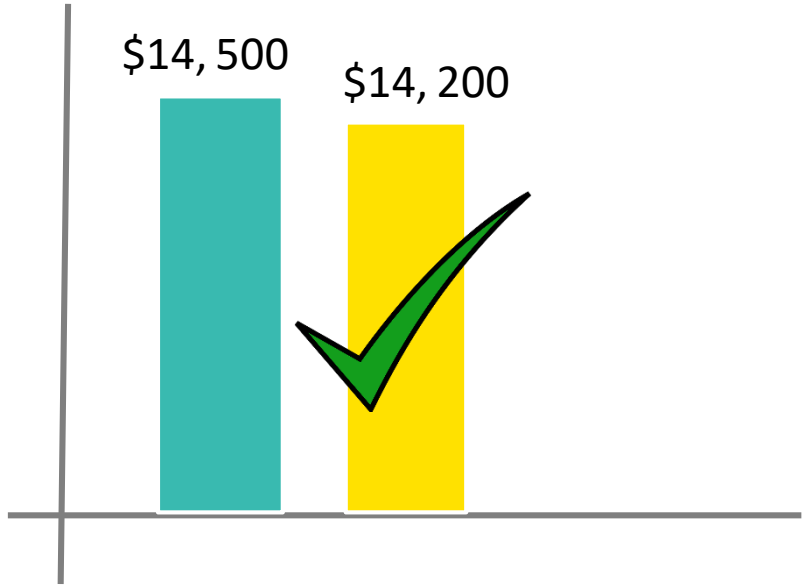
$$Cost = \frac{1}{2}(\hat{Y} - Y)^2$$

Gradient Descent in Neural Networks





$$Cost = \frac{1}{2}(\hat{Y} - Y)^2$$



Stochastic Gradient Descent

Data Science Academy

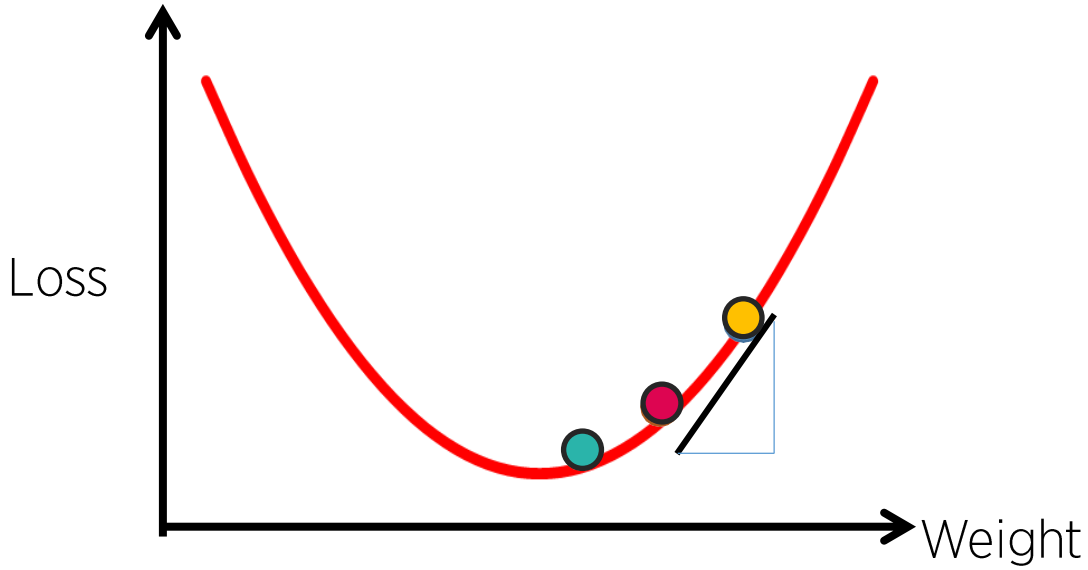
Stochastic Gradient Descent

- Naïve Gradient Decent is very computationally expensive/slow as it requires exposure to the entire dataset, then updates the gradient.
- Stochastic Gradient Descent (SGD) does the updates after every input sample. This produces noisy or fluctuating loss outputs. However, again this method can be slow.
- Mini Batch Gradient Descent is a combination of the two. It takes a batch of input samples and updates the gradient after that batch is processed (batches are typical 20-500, though no clear rule exists). It leads to must faster training (i.e. faster convergence to the global minima)

Stochastic Gradient Descent

X1	X2	....	Xn

- Gradient Descent Algorithm is based on some mathematical operations such as derivatives etc. Stochastic Gradient Descent chooses only one observation to optimize model.

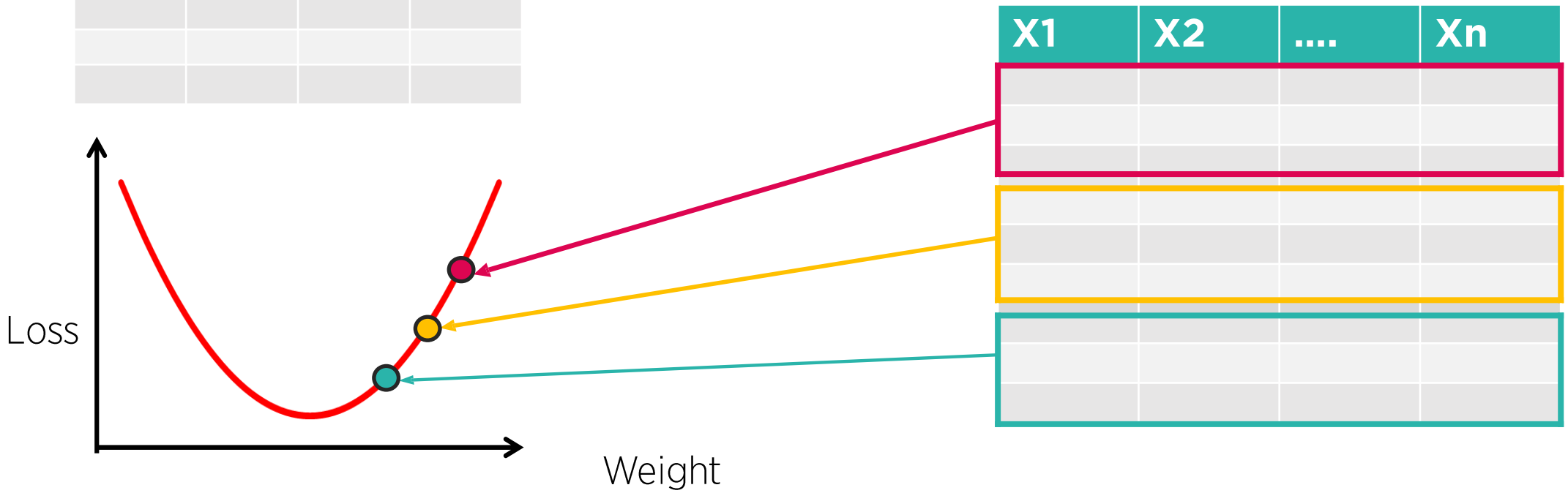


$$w_{i+1} = w_{i\ \mathrm{A}} - \eta * f'(w)$$

Mini Batch Gradient Descent

X1	X2	....	Xn

- Mini Batch Gradient Descent chooses samples from observations to optimize model.



Batch Size

- Unless we had huge volumes of RAM, we can't simply pass all our training data to our Neural Network in training. We need to split the data up into segments or… Batches.
- Batch Size is the number of training samples we use in a single batch.
- Example, say we had 1000 samples of data, and specified a batch size of 100. In training, we'd take 100 samples of that data and use it in the forward/backward pass then update our weights. If the batch size is 1, we're simply doing Stochastic Gradient Descent.

Pytorch vs TensorFlow

- Pytorch is used for many deep learning projects today, and its popularity is increasing among AI researchers.
- When researchers want flexibility, debugging capabilities, and short training duration, they choose Pytorch. It runs on Linux, macOS, and Windows.
- Thanks to its well-documented framework and abundance of trained models and tutorials, TensorFlow is the favorite tool of many industry professionals and researchers. TensorFlow offers better visualization, which allows developers to debug better and track the training process.
- TensorFlow also does good work in deploying trained models to production, thanks to the TensorFlow Serving framework. Pytorch offers no such framework, so developers need to use Django or Flask as a back-end server.
- In the area of data parallelism, PyTorch gains optimal performance by relying on native support for asynchronous execution through Python. However, with TensorFlow, you must manually code and optimize every operation run on a specific device to allow distributed training.

PyTorch vs Keras

- Both of these choices are good if you're just starting to work with deep learning frameworks. Mathematicians and experienced researchers will find Pytorch more to their liking. Keras is better suited for developers who want a plug-and-play framework that lets them build, train, and evaluate their models quickly. Keras also offers more deployment options and easier model export.
- However, Pytorch is faster than Keras and has better debugging capabilities.
- Both platforms enjoy sufficient levels of popularity that they offer plenty of learning resources. Keras has excellent access to reusable code and tutorials, while Pytorch has outstanding community support and active development.
- Keras is the best when working with small datasets, rapid prototyping, and multiple back-end support. It's the most popular framework thanks to its comparative simplicity. It runs on Linux, MacOS, and Windows.

Additional resources

Data Science Academy

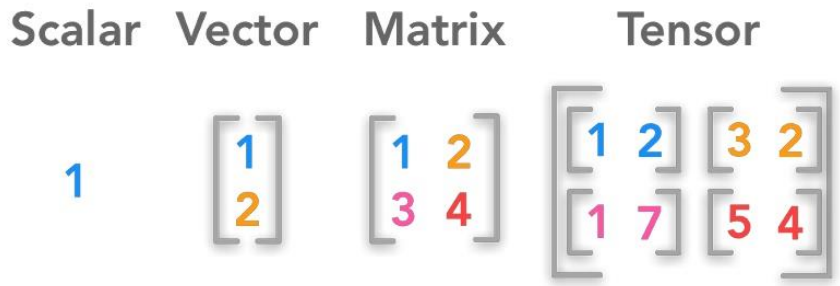
TensorFlow

Data Science
Academy

About Tensorflow

- TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains.

- TensorFlow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.



Install TensorFlow

Windows

Download Anaconda
Create an environment with all must-have libraries

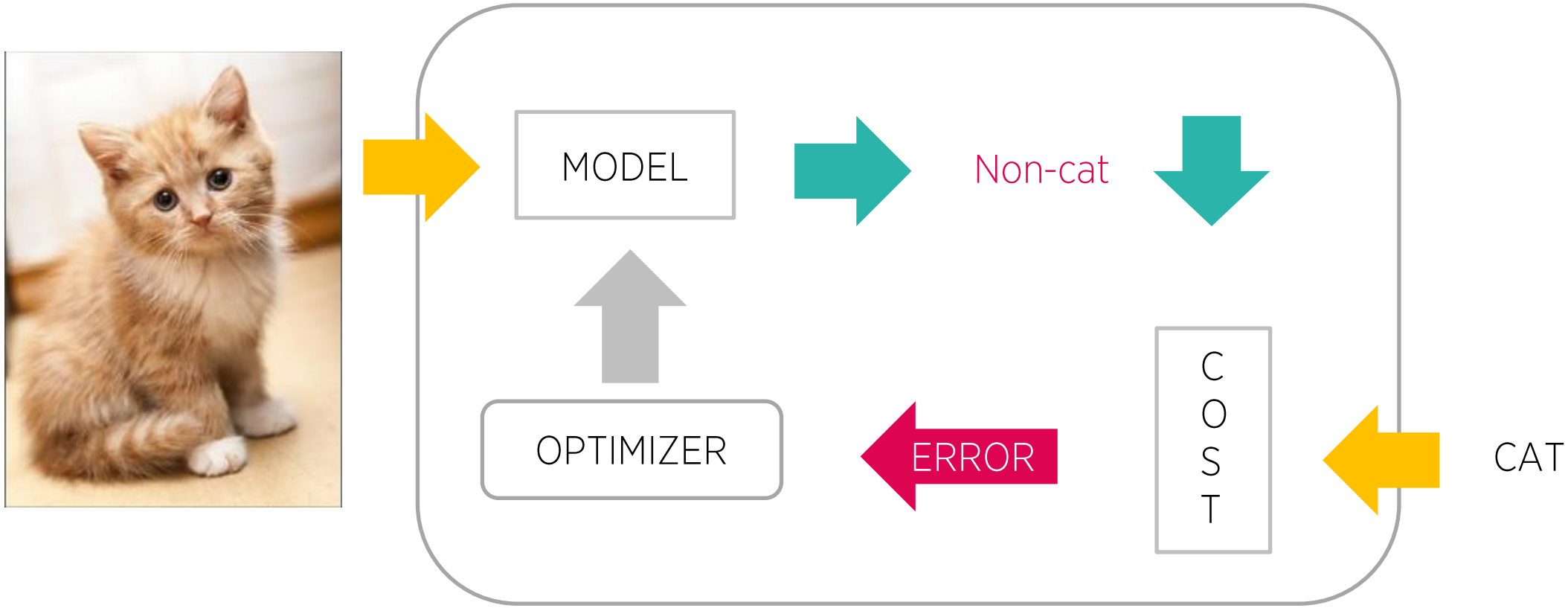
```
$ conda create -n tensorflow python=3.5
$ activate tensorflow
$ conda install pandas matplotlib jupyter notebook scipy scikit
$ pip install tensorflow
```

Linux and Mac OS

Download Anaconda
Create an environment with all must-have libraries.

```
$ conda create -n tensorflow python=3.5
$ activate tensorflow
$ source conda install pandas matplotlib jupyter notebook scipy scikit
$ pip install tensorflow
```

Concepts

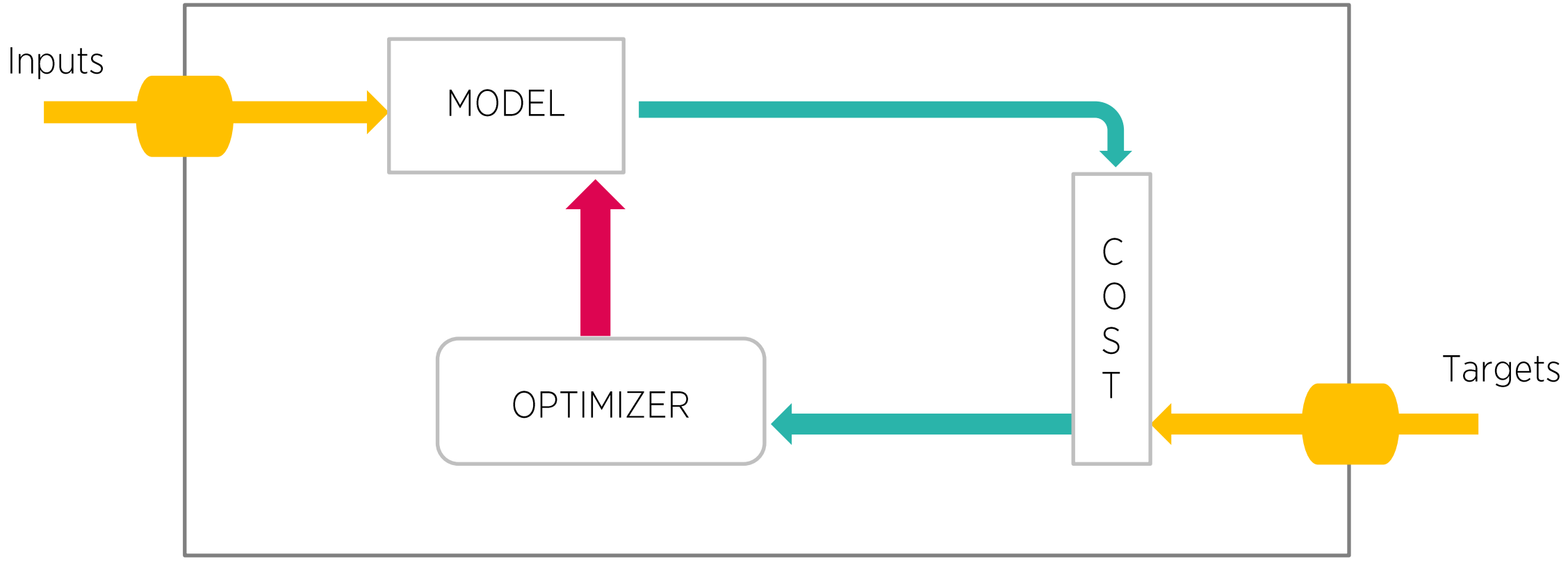


GRAPH

Graph

- Placeholders: gates where we introduce example
- Model: makes predictions. Set of variables and operations
- Cost function: function that computes the model error
- Optimizer: algorithm that optimizes the variables so the cost would be zero

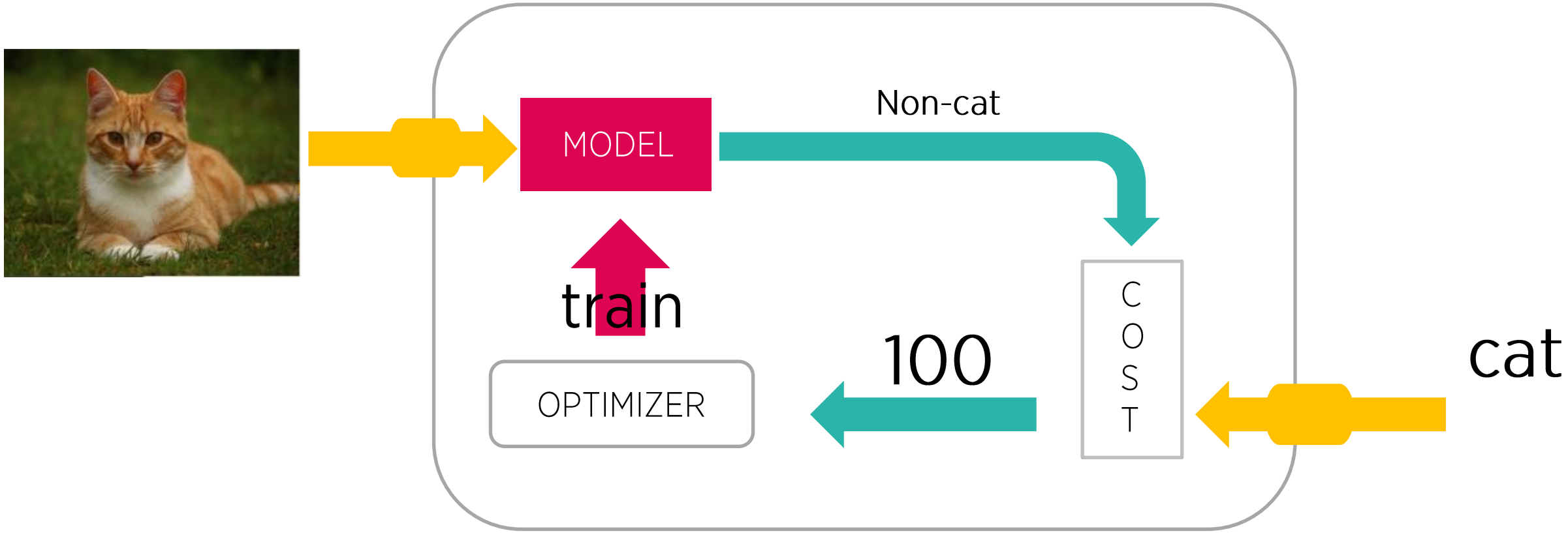
Session: Graph + Data



Graph, Data and Session

- Graph: Layout of the prediction and learning process. It does not include data.
- Data: examples that will train the neutral network. It consists on two kinds: inputs and targets.
- Session: where everything takes place. Here is where we feed the graph with data.

Session: Graph+Data



Hello World!: Sum of two integers

```
Import tenserflow as tf
```

In [5]:

```
with tf.Session() as sess:
        sum_output = sess.run(sum_graph, feed_dict={
a: num1,
b: num2
})
Print("The sum of {} and {} is {}".format(num1,num2, sum_output))
```

The sum of 3 and 8 is 11

TensorFlow for Regression: learning how to sum

Mission: learn how to sum sing 10,000 examples.

$$x_1 \; ? \; x_2 = y$$

We assume the relationship between x and y is linear function.

$$x \cdot W + b = y$$

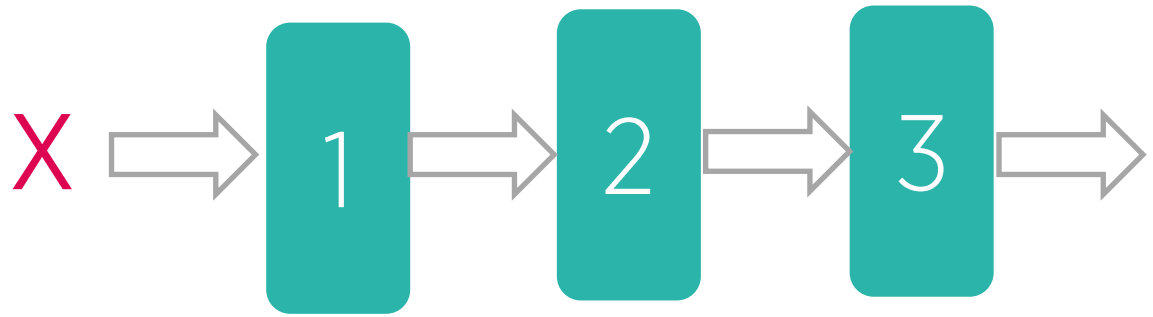
variables to be learned

Neural Network

$$x \cdot W + b \grave{} y$$

$$\sigma(x \cdot W_1 + b_1) \cdot W_2 + b_2 = y$$

$$\tanh(\sigma(x \cdot W_1 + b_1) \cdot W_2 + b_2) \cdot W3 + b3 = y$$



TensorFlow for Regression: learning how to sum

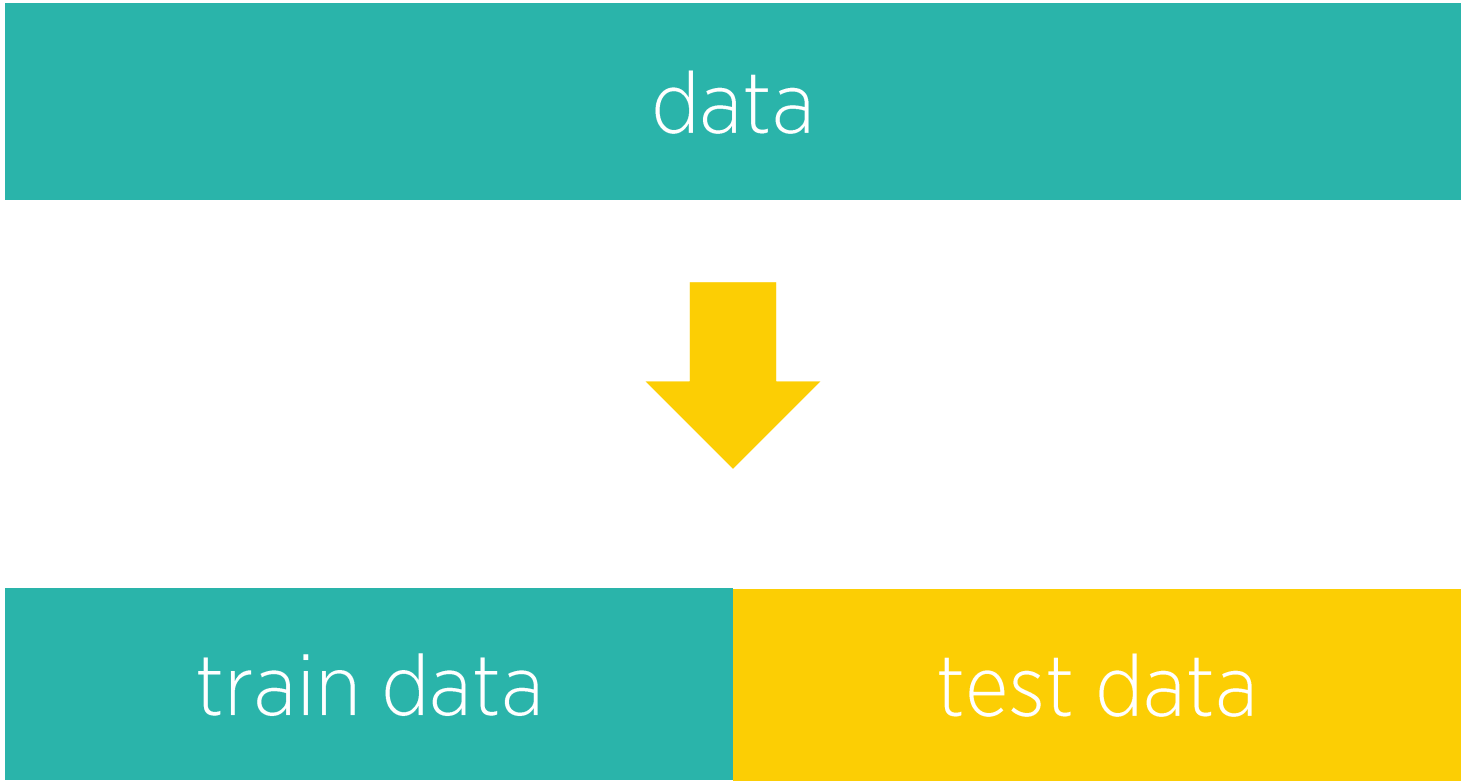
```
#PLACEHOLDERS

x = tf.placeholder(tf.float32, [None, 2])
y = tf.placeholder(tf.float32, [None, 1])
```

We don't know how many examples we'll have, but we do know that each one of them has 2 numbers as input and 1 as target

TensorFlow for Regression: Learning how to sum

```
#MODEL

W = tf.Variable(tf.truncated_normal([2,1],
stddev=0.005))
b = tf.Variable(tf.random_normal([1]))

output = tf.add(tf.matmul(x,W),b)
```

Data split



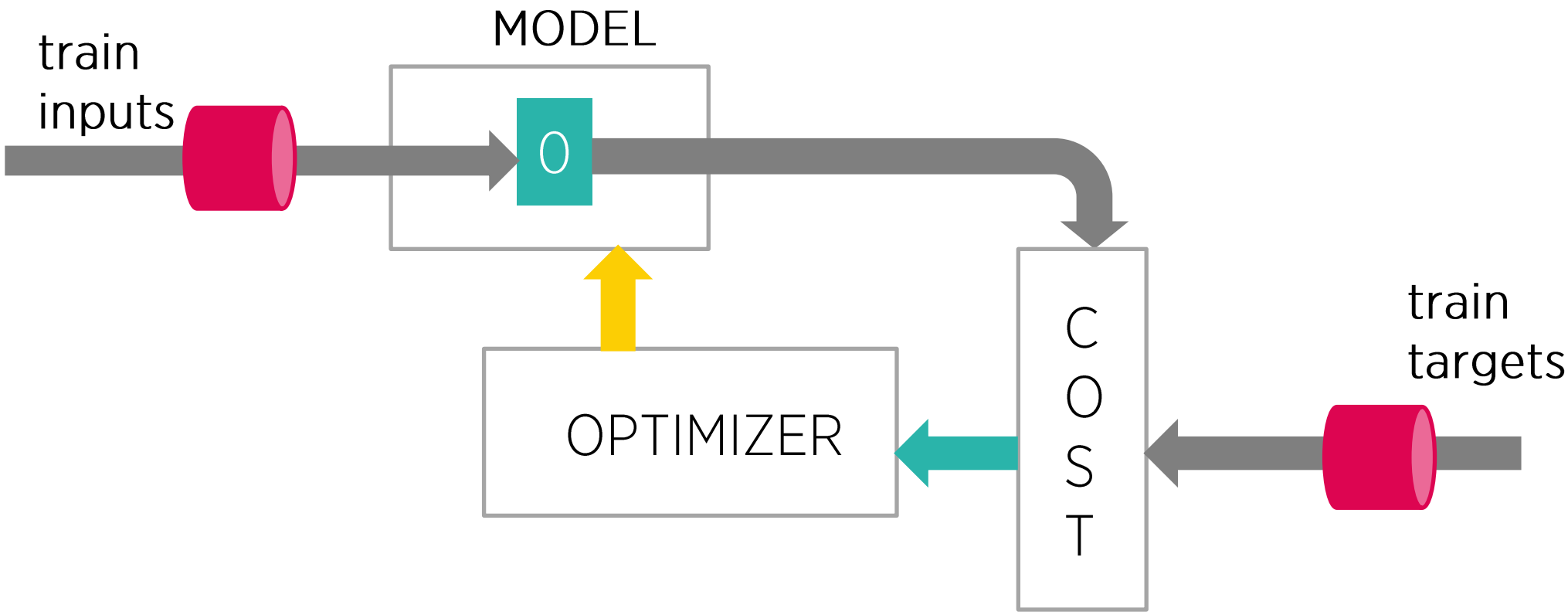
TensorFlow for Regression

- Learning how to sum

```
from helper import get_data, split_data

# DATA

inputs, targets = get_data(max_int=10, size=10000)

# split train and test data

train_inputs, test_inputs, train_targets, test_targets = split_data(inputs, targets)
```

TensorFlow for Regression

- Learning how to sum



TensorFlow for Regression

- Learning how to sum

```
with tf.Session() as sess:

sess.run(tf.global_variables_initializer())

for epoch in range(epochs):

sess.run(optimizerj feed_dict={

x: traininputSj y: train_targets

})
```

Testing Accuracy: 0.9568796753883362

The sum of 5 plus 7 is 11.991448402404785

The weights are: [[0.88418758]
[0.8903569]]

and the bias is: [1.33801162]

TensorFlow for Classification

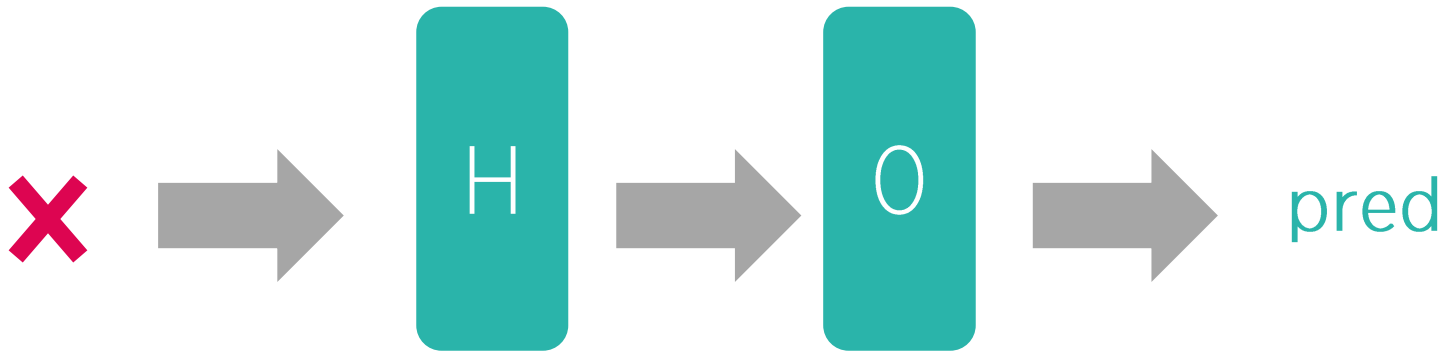
- Mission: learn if the sum of two numbers is higher than 10.

if (x_1 + x_2 > 10) then **y** = [0; 1]

else **y** = [1; 0]

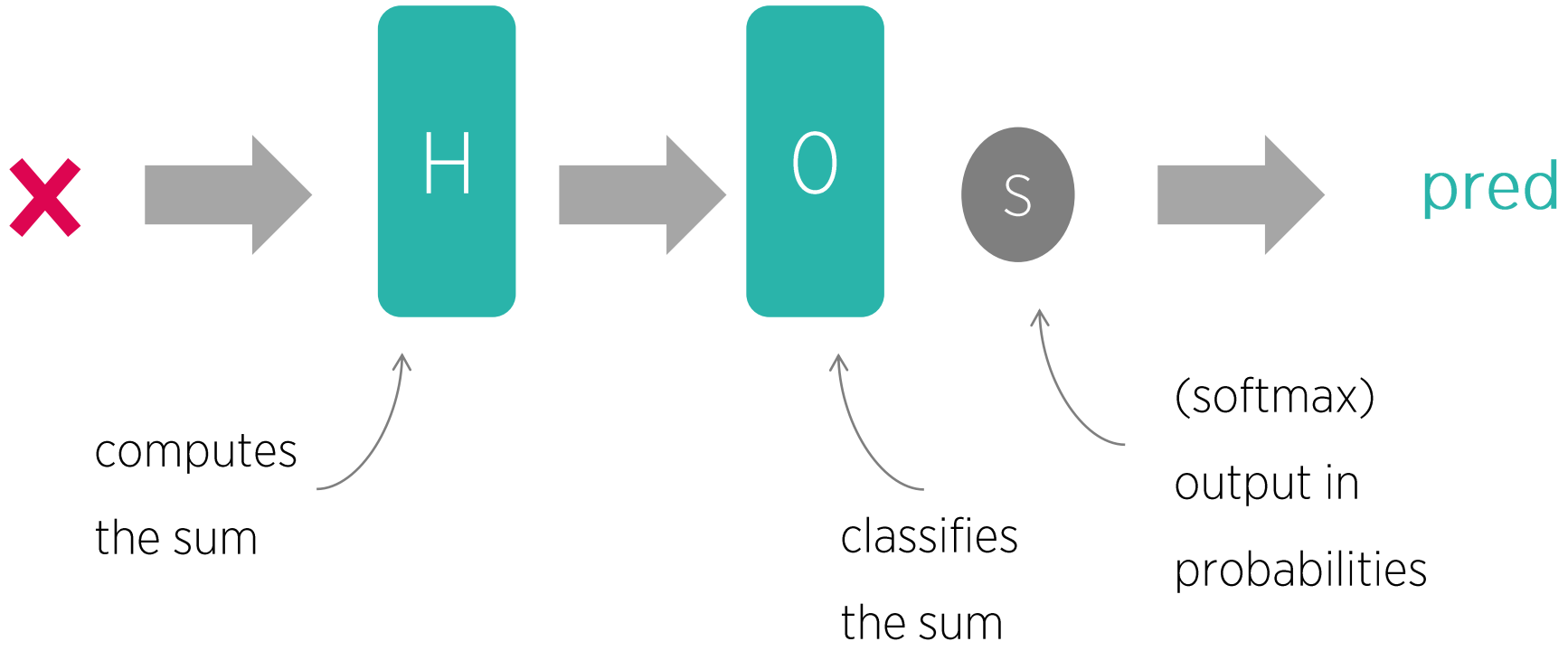
x_1 ?? x_2 = **y**

More complexity: we add a new layer



Neural Networks: intuition

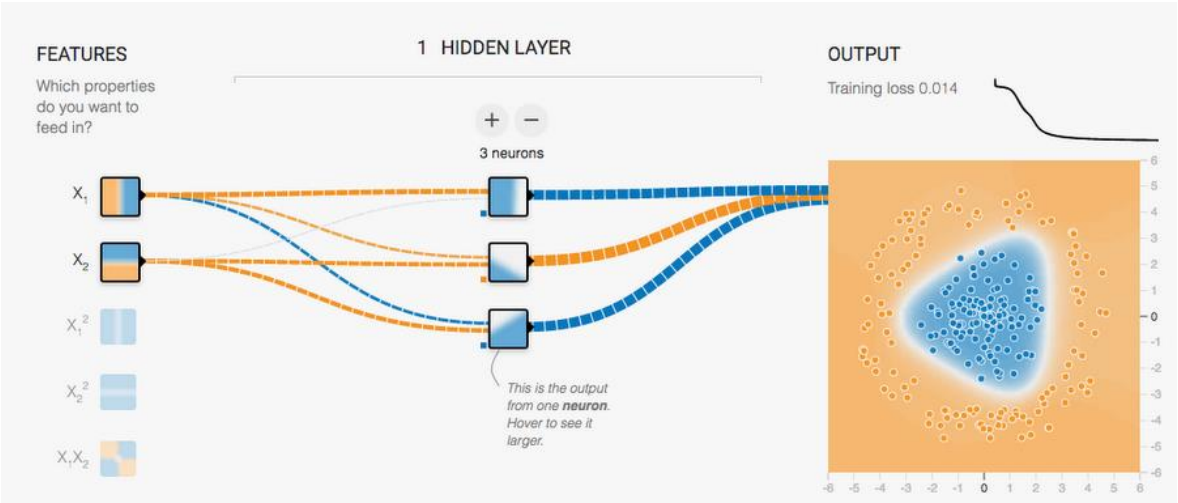
- First layers extract the more basic features, while the next ones will work from this information.



Neurons in Tensorflow

A simple classification problem on TensorFlow Playground.

Nonlinear classification problem on TensorFlow Playground



Normalization of models in Tensorflow

- Basic idea is to normalize the output of an activation layer to improve the convergence during training. Typically the normalization is performed by calculating the mean and the standard deviation of a subgroup in your input tensor. It is also possible to apply a scale and an offset factor to this as well.
- Although the model might converge without feature normalization, it makes training more difficult, and it makes the resulting model more dependent on the choice of units used in the input.
- We are going to use the feature_spec interface implemented in the tfdatasets package for normalization. The feature_columns interface allows for other common pre-processing operations on tabular data.

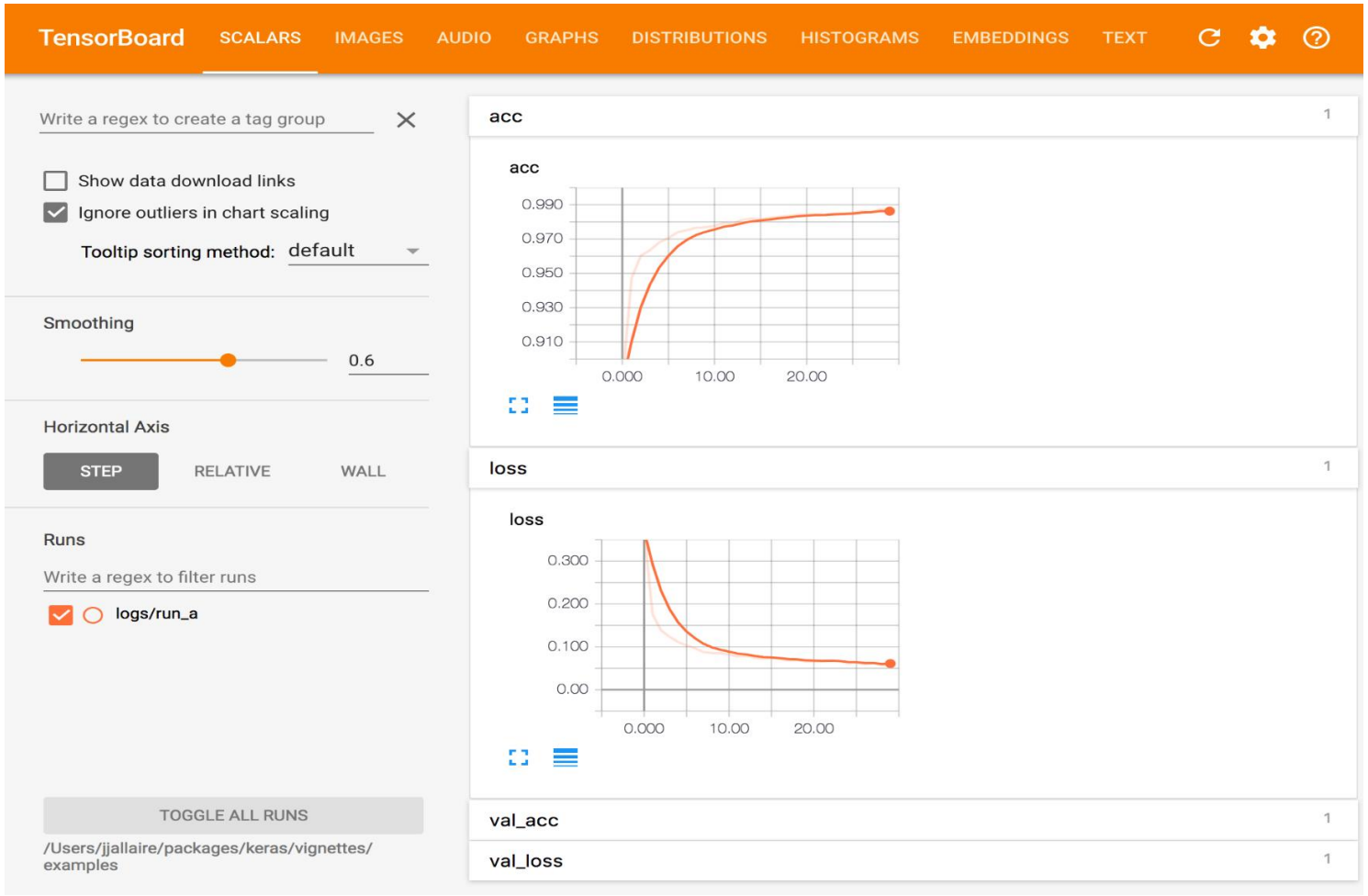
TensorBoard

TensorBoard is a visualization tool included with TensorFlow that enables to visualize dynamic graphs of Keras training and test metrics, as well as activation histograms for the different layers in model. TensorBoard provides the visualization and tooling needed for machine learning experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional space
- Displaying images, text, and audio data

TensorBoard

- TensorBoard display for Keras accuracy and loss metrics:



Keras

Data Science
Academy

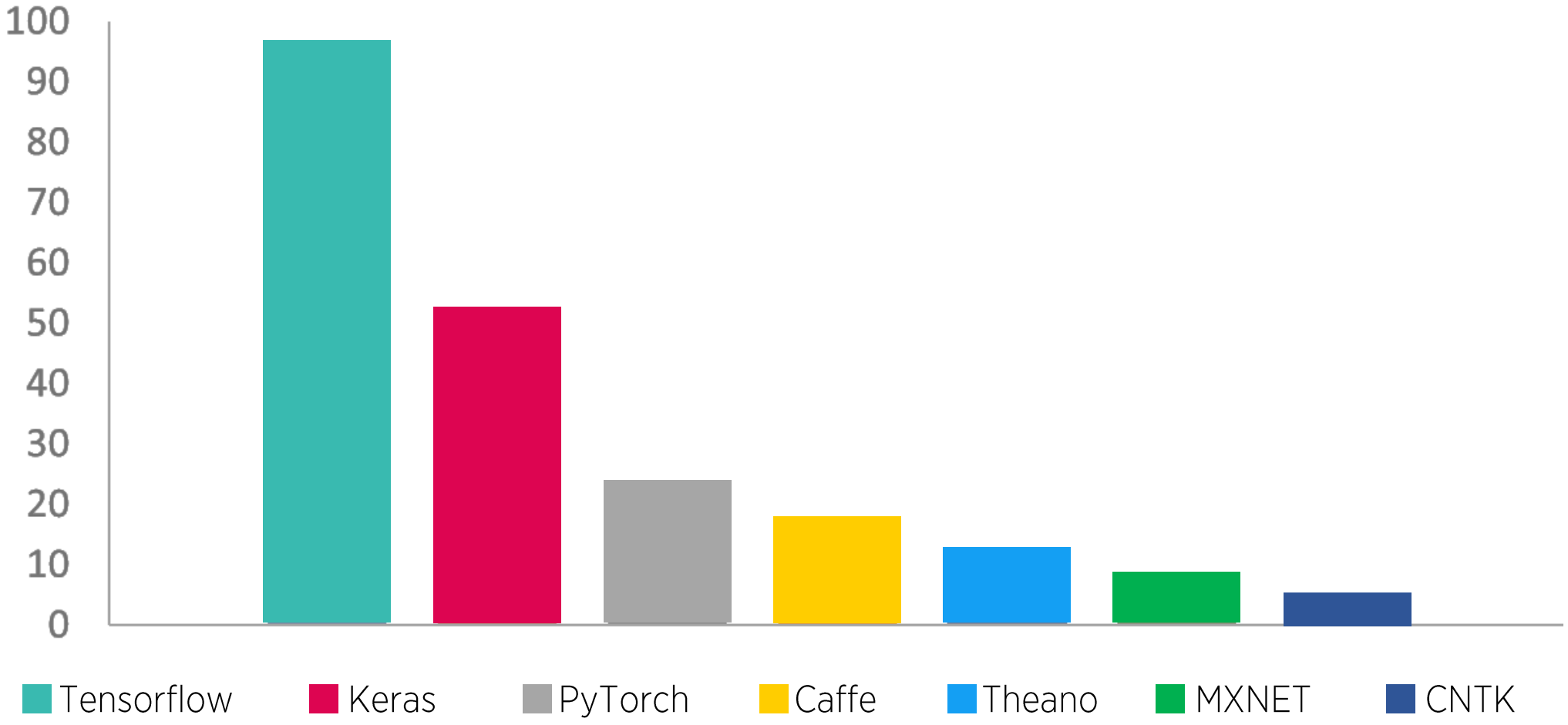
Keras

Keras is a high-level neural networks API developed with a focus on enabling fast experimentation.

Keras has the following key features:

- Allows the same code to run on CPU or on GPU, seamlessly.
- User-friendly API which makes it easy to quickly prototype deep learning models.
- Built-in support for convolutional networks (for computer vision), recurrent networks (for sequence processing), and any combination of both.
- Supports arbitrary network architectures: multi-input or multi-output models, layer sharing, model sharing, etc. This means that Keras is appropriate for building essentially any deep learning model, from a memory network to a neural Turing machine.
- Is capable of running on top of multiple back-ends including TensorFlow, CNTK, or Theano.

Deep Learning Framework Ranking



Keras

- The Keras R interface uses the TensorFlow backend engine by default. To install both the core Keras library as well as the TensorFlow backend use the install_keras() function:

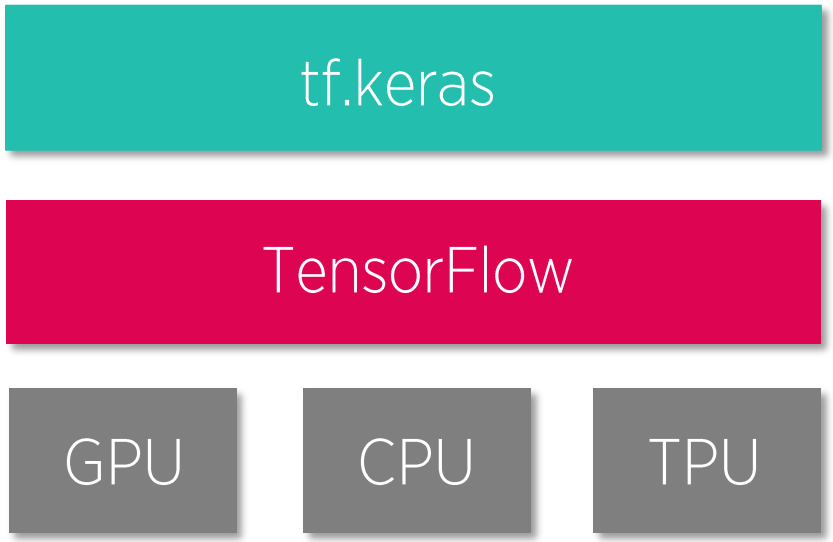
library(keras)
install_keras()

- This will provide you with default CPU-based installations of Keras and TensorFlow.

Keras

Keras is the official high-level API of TensorFlow

- tensorflow.keras (tf.keras) module
- Part of core TensorFlow since v1.4
- Full Keras API
- Better optimized for TF
- Better integration with TF-specific features
  - Estimator API
  - Eager execution
  - etc.



Who makes Keras?
Contributors and backers

What's special about Keras?

- A focus on user experience.
- Large adoption in the industry and research community.
- Multi-backend, multi-platform.
- Easy productization of models.

The Keras user experience

- Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon usei error.

- This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster - which in turn helps you win machine learning competitions.

- This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you tc implement anything you could have built in the base language. In particular, as tf.keras, the Keras AP integrates seamlessly with your TensorFlow workflows.

Keras

- Keras is multi-backend, multi-platform
- Develop in Python, R
  - On Unix, Windows, OSX
- Run the same code with…

  - TensorFlow
  - CNTK Theano
  - MXNet
  - PlaidML ??

- CPU, NVIDIA GPU, AMD GPU, TPU…

Three API styles

- The Sequential Model
  - Dead simple
  - Only for single-input, single-output, sequential layer stacks Good for 70+% of use cases

- The functional API
  - Like playing with Lego bricks
  - Multi-input, multi-output, arbitrary static graph topologies Good for 95% of use cases

- Model subclassing
  - Maximum flexibility Larger potential error surface

The Sequential API

```
import keras

from keras import layers model = keras. SequentiaK )

model.add(layers.Dense(20, activation='relu', input_shape=(10,)))
model.add(layers.Dense(20, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))

model.fit(x, y, epochs=10, batch_size=32)
```

The function API

```
import keras

from keras import layers

inputs = keras.Input(shape=(10,))
x = layers.Dense(20, activation='relu')(x)
x = layers.Dense(20, activation='relu')(x) outputs = layers.Dense(10, activation=1softmax1)(x)

model = keras.ModeKinputs, outputs) model`x, y, epochs=10, batch_size=32)
```

Model Subclassing

```
import keras
from keras import layers

class MyModel(keras.Model):

    def __init__(self):
            super(MyModel, self).__init__()
            self.densel = layers.Dense(20, activation='relu1)
            self.dense2 = layers.Dense(20, activation='relu1)
            self.dense3 = layers.Dense(10, activation='softmax1)

def calKself, inputs):
            x = self.densel(x)
            x = self.dense2(x) return self.dense3(x)

model = MyModel()
model.fit(x, y, epochs=10, batch_size=32)
```

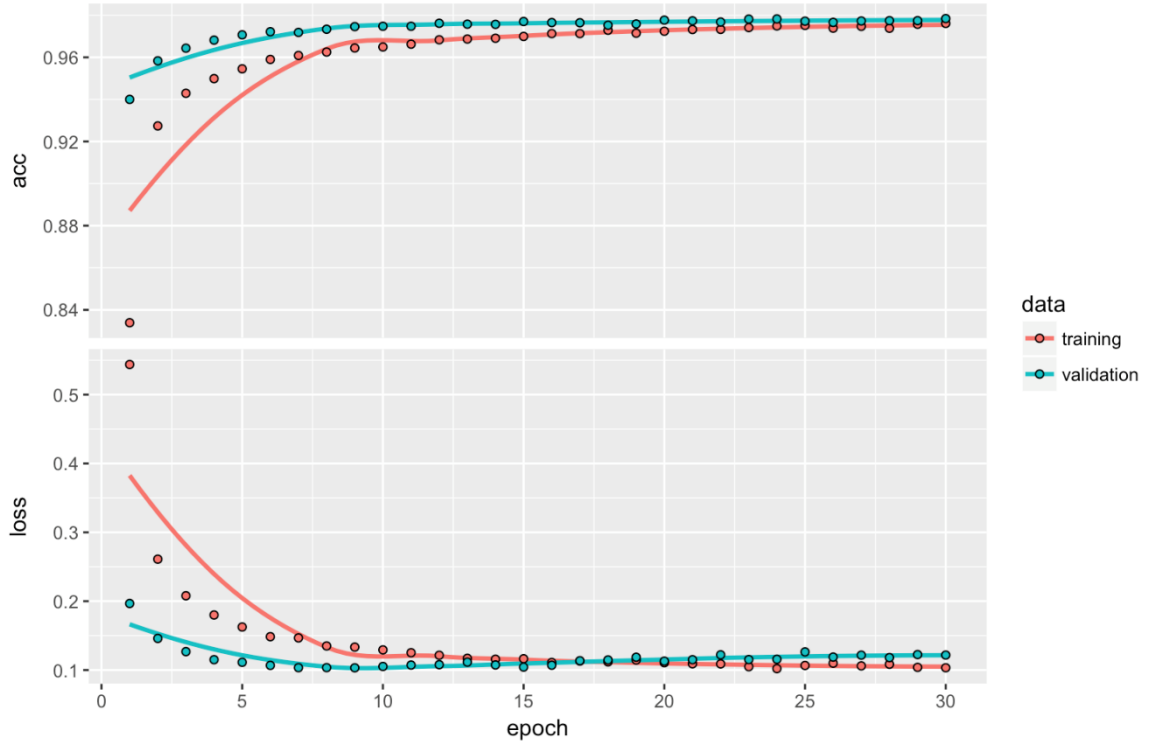
Interactive visualization of deep neural network

There are a number of tools available for visualizing the training of Keras models, including:

- A plot method for the Keras training history returned from fit().
- Real time visualization of training metrics within the RStudio IDE.
- Integration with the TensorBoard visualization tool included with TensorFlow. Beyond just training metrics, TensorBoard has a wide variety of other visualizations available including the underlying TensorFlow graph, gradient histograms, model weights, and more. TensorBoard also enables you to compare metrics across multiple training runs.

Plotting History

- The Keras fit() method returns an R object containing the training history, including the value of metrics at the end of each epoch . You can plot the training metrics by epoch using the plot() method.



PyTorch

Data Science
Academy

Dataset

Dataset

- In PyTorch, a dataset is represented by a regular Python class that inherits from the Dataset class. You can think of it as kind of Python list of tuples, each tuple corresponding to one point (features, label)

3 components :

- __init__(self)
- __get_item__(self,index)
- __len__(self)

- Unless the dataset is huge (cannot fit in memory), you do not explicitlyneed to define this class. Use TensorDataset

```
from torch.utils.data import Dataset,, TensorDataset

class CustomDataset(Dataset):

def   __init__(self, x_tensor, y_tensor):

self.x = x_tensor self.y = y_tensor

def   __getitem__ (self, index):

return (self.x[index], self.y[index])

def  _len_ (self):

return len(self.x)

x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])

train_data = TensorDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

Dataloader

Dataloader

- What happens if we have a huge dataset? Have to train in “batches”
- Use Pytorch’s Dataloader class!
- We tell it which dataset to use, the desired mini-batch size and if we’d like to shuffle it or not. That is it.
- Our loader will behave like an iterator, so we can loop over it and fetch a different mini-batch every time.

```
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
```

Dataloader (example)

- Sample code in practice:

```
losses = [ ]

model = ManualLinearRegression().to(device)

loss_fn = nn.MSELoss(reduction='mean') optimizer =
optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):

for x_batch, y_batch in train_loader: model.train()

x_batch = x_batch.to(device) y_batch = y_batch.to(device) yhat =
model(x_train_tensor)

loss = loss_fn(y_batch, yhat) loss.backward() optimizer.step()
optimizer.zero_grad()

losses.append(loss)

print(model.state_dict())
```

Split Data

Random Split for Train, Val and Test Set

- `random_split()`

```
from torch.utils.data.dataset import random_split

x_tensor = torch.from_numpy(x).float() y_tensor =
torch.from_numpy(y).float()

dataset = TensorDataset(x_tensor, y__tensor)

train_dataset, val_dataset, test_dataset = random_split(dataset, [60,
20, 20])

train_loader = DataLoader(dataset=train__dataset, batch_size=16)
val_loader = DataLoader(dataset=val_dataset, batch_size=20)
test_loader = DataLoader(dataset=test_dataset,| batch_size=20)
```

Saving/Loading Weights

Method 1

```
#Only inference/evaluation-save only state_dict
#Save:
    torch.save(model.state_dict(), PATH)
    Load:
            Model=TheModelClass(*args, **kwargs)
            Model.load_state_dict(torch.load(PATH))
            Model.eval()

#CONVENTION IS TO SAVE MODELS USING EITHER A .PT OR A.PTH
EXTENSION
```

Saving/Loading Weights (continued)

Method 2

```
#Checkpoint – resume training / inference
#Save:
torch.save({
                    'epoch': epoch,
                    'model_state_dict': model.state_dict(),
                    'optimizer_state_dict': optimizer.state_dict(),
                    'loss': loss,
                     ...
                    }, PATH)
# Load:
model= TheModelClass(*args, **kwargs)
      oprimizer= TheOptimizerClass(*args, **kwargs)
      model.load_state_dict(checkpoint['model_state_dict'])
      optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
      epoch= checkpoint['loss']
model.eval()
# - or –
model.train()
```

Evaluation

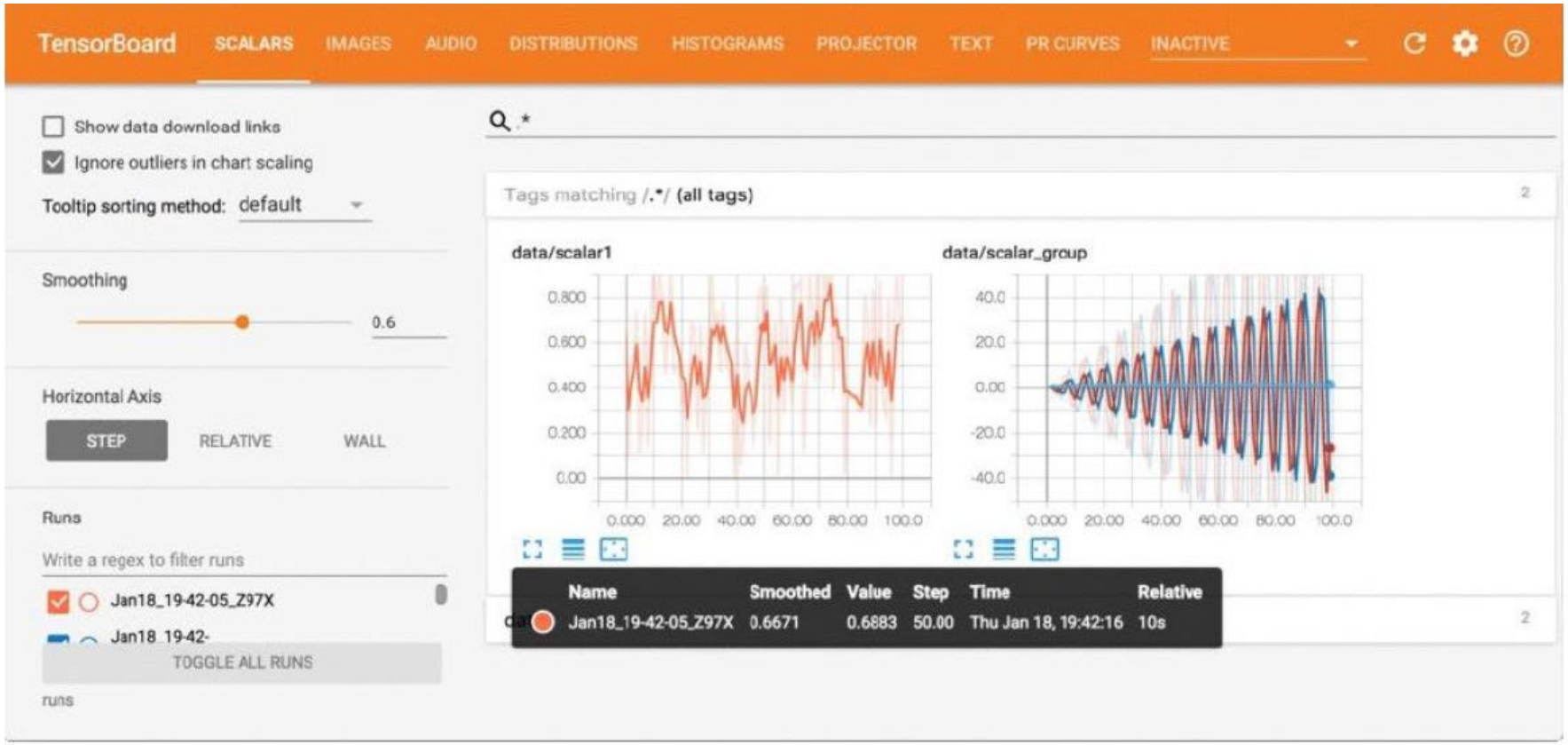
Two important things:

- `torch.no_grad()`
  - Do not store the history of all computations
- `eval()`
  - Tell compiler which mode to run on.

```
losses = []
val_losses - []
model = ManualLinearRegression().to(device) loss_fn =
nn.MSELoss(reduction='mean') optimizer = optim.SGD(model.parameters(), lr=lr)
for epoch in range(n_epochs):
for x_batch, y_batch in train_loader: model.train()
x_batch = x_batch.to(device) y_batch = y_batch.to(device) yhat =
model(x_train_tensor)
loss = loss_fn(y_batch, yhat) loss.backward() optimizer.step()
optimizer.zero_grad()
losses.append(loss)
with torch.no_grad():
for x_val, y_val in val_loader: x_val = x_val.to(device) y_val = y_val.to(device)
model.eval()
yhat = model(x_val)
val_loss = loss_fn(y_vai, yhat)
val_losses.append(val_loss.item())
```

Visualization

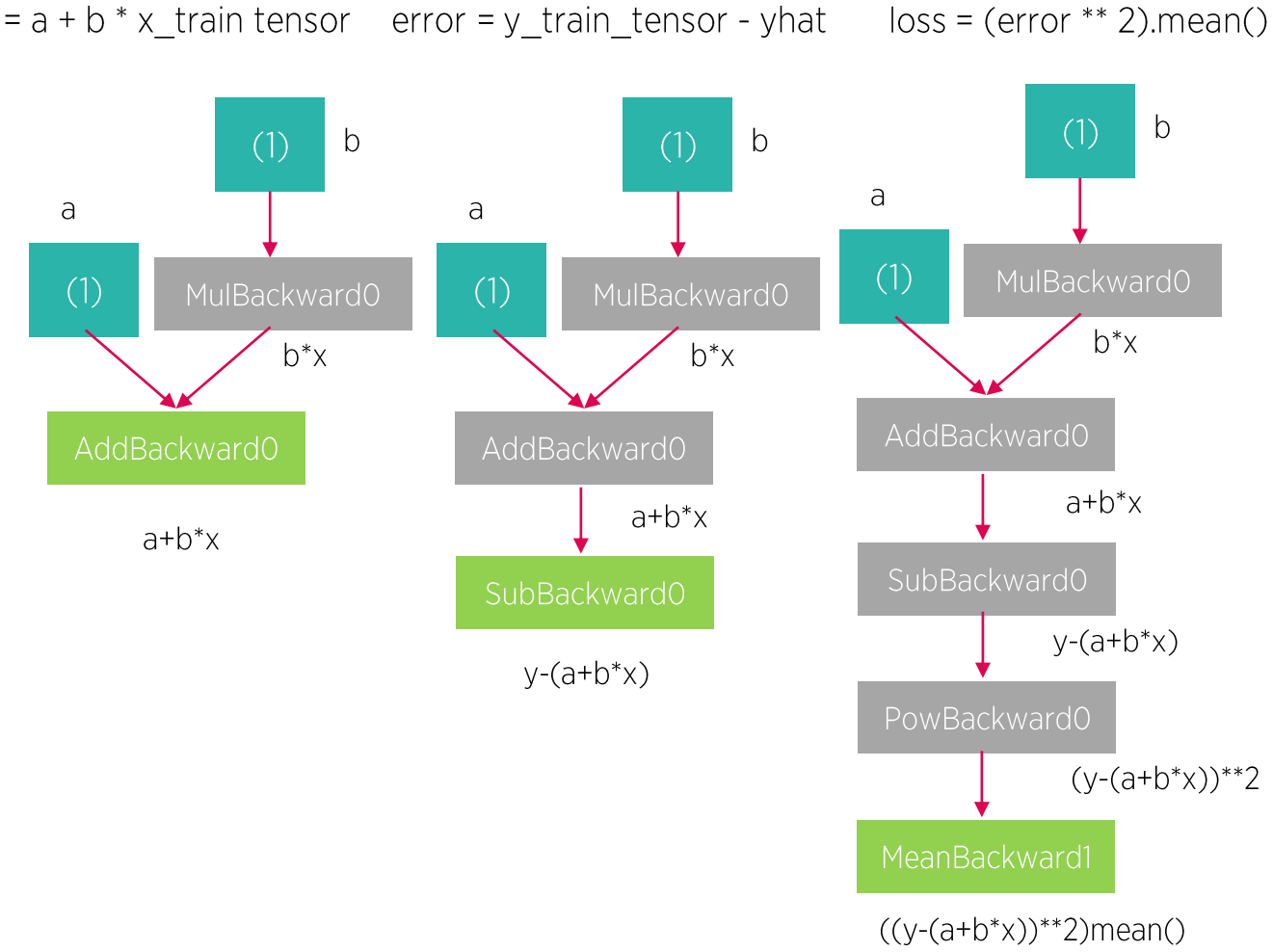
- TensorboardX (visualize training)
- PyTorchViz (visualize computation graph)



Visualization (continued)

- PyTorchViz

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device-device)
b = torch.randn(I, requires_grad=True, dtype=torch.float, device=device)

# Defines a SGD optimizer to update the parameters optimizer =
optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):

yhat = a + b * x_train_tensor
error = y_train_tensor - yhat
loss = (error ** 2).mean()

loss.backward()

optimizer.step()

optimizer.zero_grad()

print(a, b)
```



Thank you!

Data Science
Academy