Deep Learning

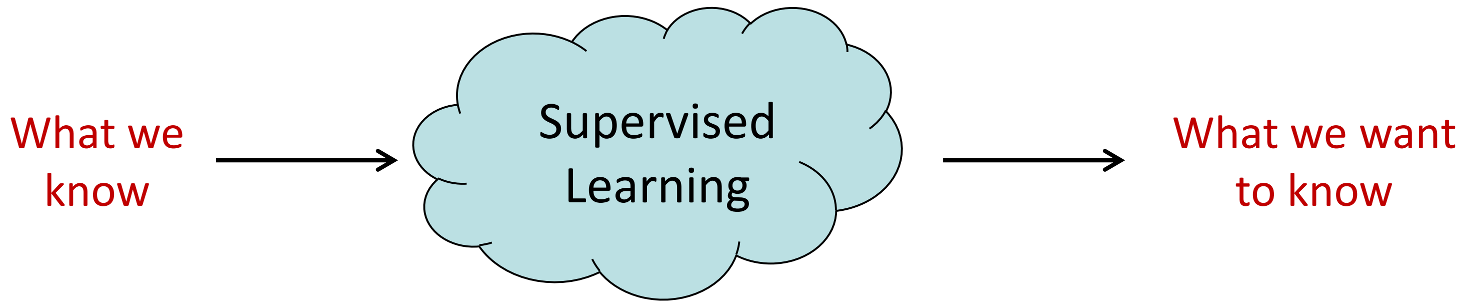
Introduction to Gradient Descent

Tuesday 29th November

Dr. Nicholas Cummins

Lecture and figures based on: Trask, A. *Deep learning*. New York: Manning Publications, 2018.

- **Supervised Learning**
  - Transforming one dataset into another
  - Taking what you know as input and transforming it into what you want to know at the output
  - Input: observable, recordable, and knowable data
  - Output: data for logical analysis

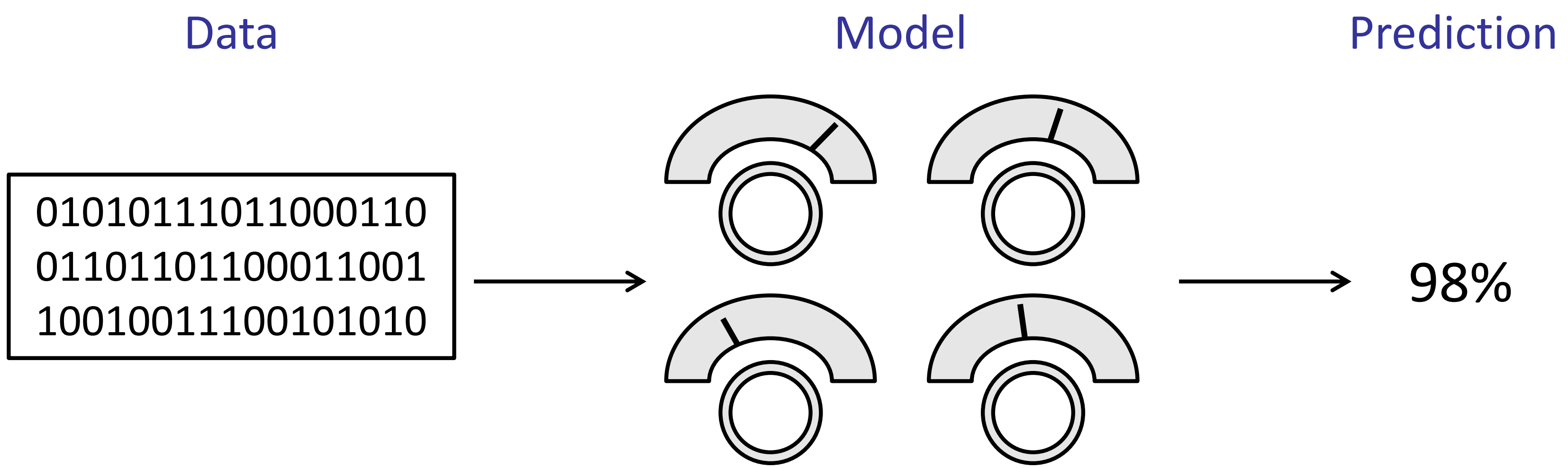


- **Supervised parametric learning**
  - A learning model that summarizes data with a set of parameters of fixed size
    - Independent of the number of training examples
  - Such algorithms involve two steps:
    1. Select a form for the function
    2. Learn the coefficients for the function from the training data
- Example simple linear regression

$$b_o + b_1 x_1 + b_2 x_2 = y$$

Supervised Learning

- **Supervised parametric learning analogy:**
  - Machine with a fixed number of knobs
  - Position of knobs indicates how to process the data
  - Processing transforms input data into an output prediction
  - Learning is accomplished by tuning the knobs



- Key steps in supervised parametric learning
  - Step 1: Predict
    - Gather data, send through machine, make a prediction
  - Step 2: Compare with truth
    - Compare the prediction with the actual score

Prediction: 98% > **Truth**: 0%

  - Step 3: Learn the pattern
    - Adjust the knobs to make a more accurate prediction
      - Considers the input data, and how much the models prediction missed by
      - Each knob represents the prediction's sensitivity to the different types of input data

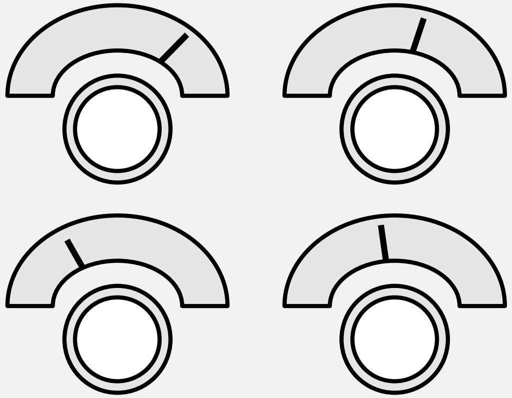
Predict

Data

1. Win/loss record: 0.5
2. Average Offside: 8.5
3. Opponent: Bayern
4. Number of fans: 30,000

→

Model



→

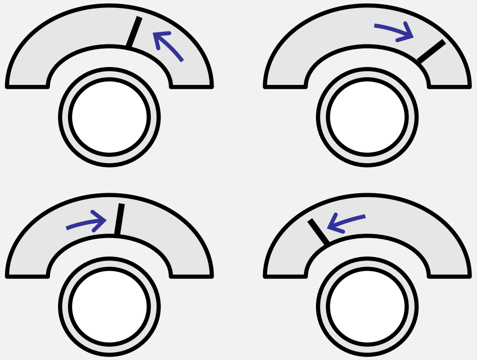
Prediction

98%

Compare

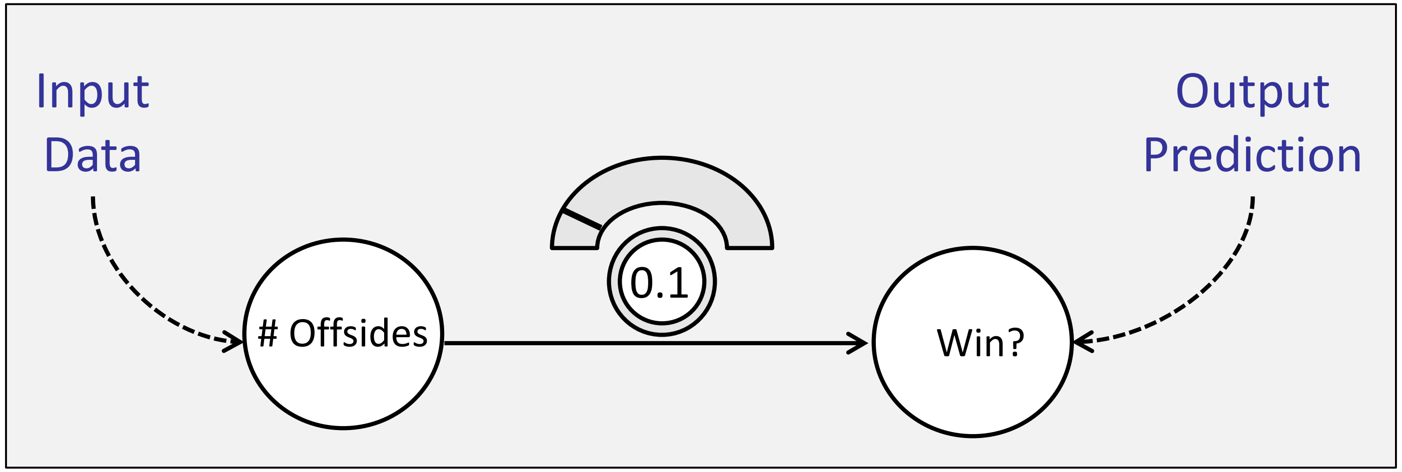
Prediction: 98% > **Truth**: 0%

Update



Universität Augsburg University

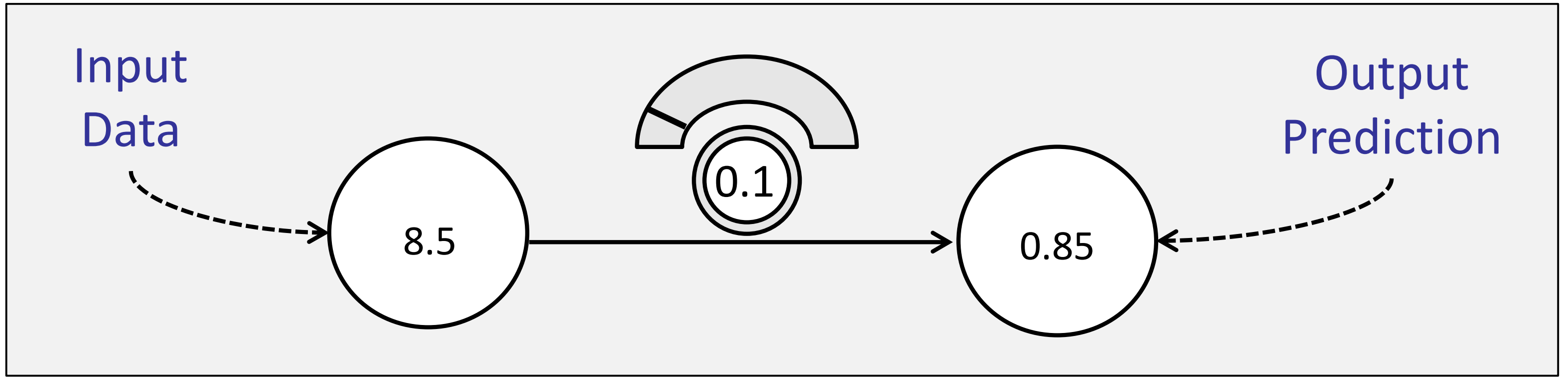
- **Simple predictions**
  - One input data point, one output prediction
  - Build a network with one single knob (*the weight*), to learn a mapping to one single output



Prediction with a single network

- **Key steps**
  1) Define network
  2) Feed input value into network
  3) Multiply input value by weight
  4) Output Prediction

```python
weight = 0.1
def neural_network(input, weight):
    prediction = input * weight
    return prediction

number_of_offsides = [8.5, 9.5, 10, 9]
input = number_of_offsides[0]
pred = neural_network(input,weight)
print(pred)
```



Prediction with a single network

- **What is a prediction?**
  - The output of the network, *given the current input data*
- **Is this prediction always right?**
  - Of course not, neural networks make mistakes
  - During training, the network learns from these mistakes
    - Prediction is too high, adjust the weights lower
    - Prediction is too low, adjust the weights higher
- **How does the network learn?**
  - Trail and error: making predictions and learn from them

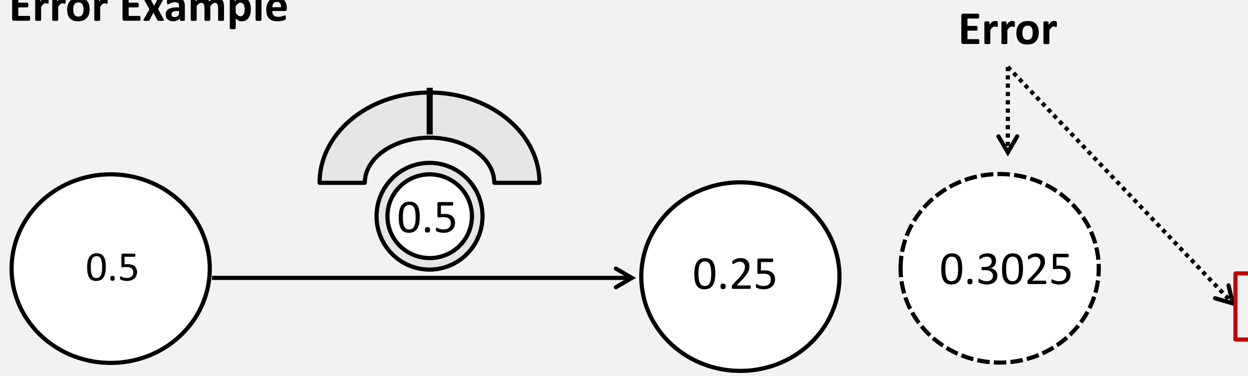
How do we set the weights?

- **Compare**
  - Evaluate how well the network performed
  - Measure of how much a prediction ‘missed’ by
  - Mean Squared Error (MSE) metric
- **Learn**
  - Adjusting each weight to reduce the error
  - Gradient Descent Algorithm

Compare

- **Measuring predictive performance**
  - Calculate error by squaring the difference between the networks production and its goal
  - Squaring the error forces it to be positive

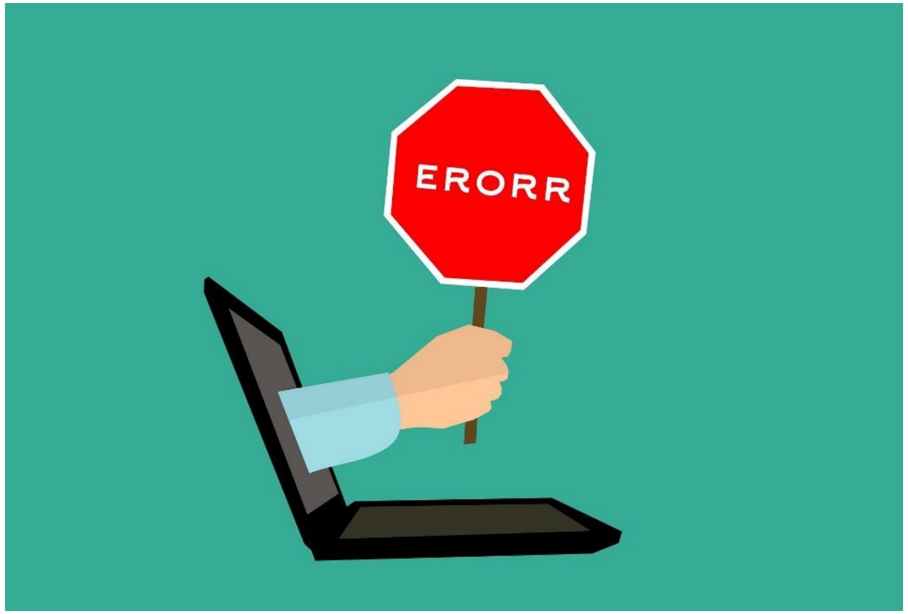
Error Example




```
knob_weight = 0.5
input = 0.5
goal_pred = 0.8

pred = input * knob_weight
error = (pred - goal_pred) ** 2
print(error)
```

Compare

- **Squaring the error**
  - Big errors become bigger
  - Small errors become smaller
  - These effects help the network learn
    - Pays more attention to the big errors
    - Pays less attention to smaller ones
  - Mean Square Error Equation:



$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)$$

Universität Augsburg University

- **Why measure error?**
  - Aim of network training is to make correct predictions
  - This can be achieved two ways:
    - Adjust weights such that prediction equal target `pred = goal_pred`
    - Adjust weights such that error equals zero `(pred-goal_pred)=0`
  - Both essentially say the same thing
  - Tuning weights to predict the target is actually a more complicated task than tuning weights to set error to zero
  - Therefore tune network such that `Error == 0`

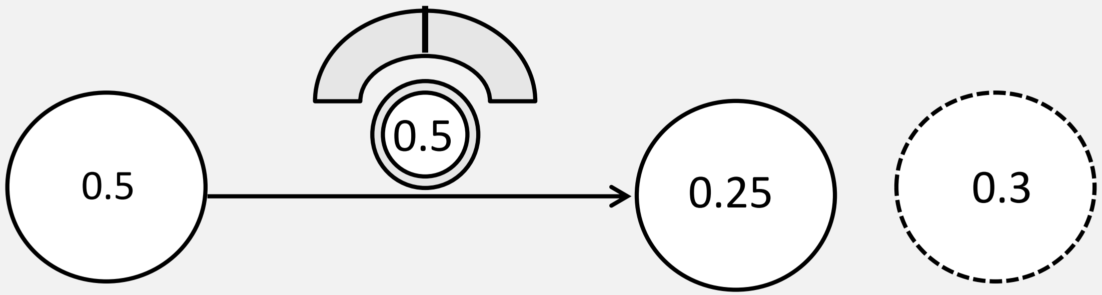
Universität Augsburg University

- **Why positive errors?**
  - Large networks can have millions of connections
  - Therefore millions of `pred` $\leftrightarrow$ `goal_pred` pairs
  - In these circumstances we need take the *average* error down to zero
    - This presents a problem if the error can be positive and negative
    - Consider network predicting two data points with errors:
      - $(y_1 - \hat{y}_1) =$ 1000 and $(y_2 - \hat{y}_2) =$ −1000
      - The average error is zero!
  - We want positive errors so they don't cancel each other out when they are average

Updating the weights

- **Measuring error and finding the direction and amount!**
  - Represents how you want to change weight
    - Pure Error
    - Scaling, negative reversal and stopping
  - This can be achieved in a single line of code!

Basic Gradient Descent Algorithm




```
weight = 0.5
goal_pred = 0.8
input = 0.5
for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount
```

Updating the weights

- What is direction and amount?

```
direction_and_amount = (pred - goal_pred) * input
```

  – How much to change weight by to reduce the error
  – Two core parts:
    1. The Pure Error: **`pred - goal_pred`**
    2. Multiplication by the input: **`*input`**
       – Modifying the pure error so it’s ready to update the weight.
       – Performs scaling, negative reversal and stopping

- **What is Pure Error?**

```
pred - goal_pred
```

  - An indication of the *raw direction* the current prediction missed by
    - If this is a *positive* number, the prediction is too *high*
    - If this is a *negative* number, the prediction is too *low*
  - Also an indication of the *amount* the current prediction missed by:
    - If this is a *big* number, the prediction has missed by a *big* amount
    - If this is a *small* number, the prediction has missed by a *small* amount

Updating the weights

- **What are scaling, negative reversal, and stopping?**
  - Have the combined effect of translating the pure error into the absolute amount you want to change weight.
  - **Stopping**
    - Do not adjust weights when input is zero
  - **Negative Reversal**
    - Ensuring that weight moves in the correct direction even when input is negative
  - **Scaling**
    - Weight changes are proportional to input size

Basic Gradient Descent Algorithm

- The golden method for neural learning

```
pred = input * weight
error = (pred - goal_pred) ** 2
delta = pred - goal_pred
weight_delta = delta * input
weight = weight - weight_delta
```

  – This approach adjusts each weight in the correct direction and by the correct amount so that error reduces to 0
    - Secret lies in the `pred` and `error` calculations

Universität Augsburg University

- Measuring error and finding the direction and amount
  - Gradient Descent in Action
    - Albeit in a bit of an oversimplified environment!

Basic Gradient Descent Algorithm

```
weight = 0.5
goal_pred = 0.8
input = 0.5
for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    direction_and_amount = (pred - goal_pred) * input
    weight = weight - direction_and_amount
    print("Error:" + str(error) + " Prediction:" + str(pred))
```

```
Error:0.3025 Prediction:0.25
Error:0.170 Prediction:0.388
Error:0.096 Prediction:0.491
...
Error:1.709e-05 Prediction:0.796
Error:9.615e-06 Prediction:0.797
Error:5.408e-06 Prediction:0.798
```

Error/Weight Relationship

- Combining the `pred` and `error` calculations

```
error = ((input * weight)- goal_pred) ** 2
```

  - For any input/prediction pair, an exact relationship can be defined between error and weight
  - This is found by combining the prediction and error formulas
    - E.g when `goal_pred = 0.8` and `input = 0.5`

```
error = ((0.5 * weight)- 0.8) ** 2
```

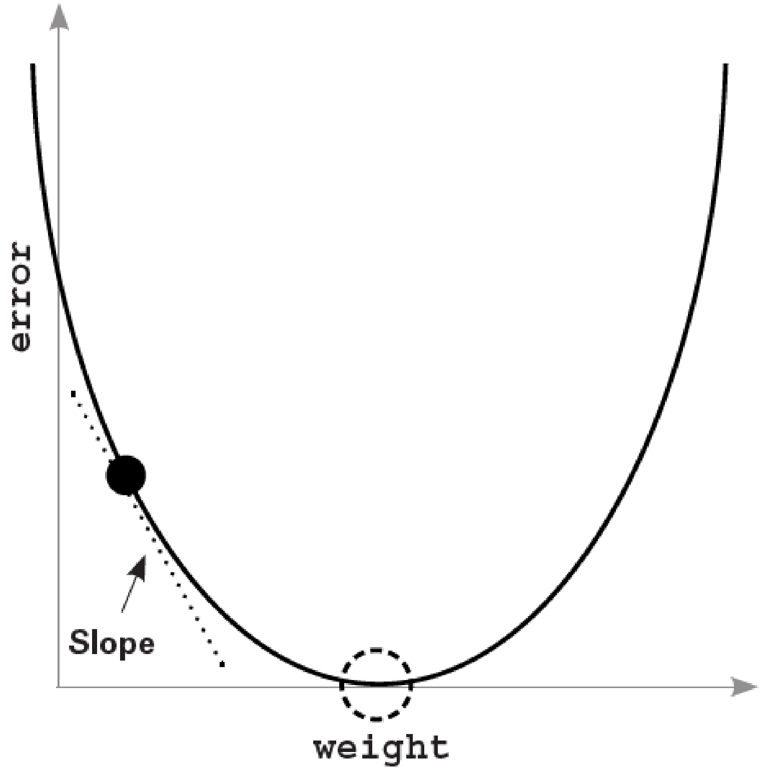
Universität Augsburg University

- The relationship between error and weight

```
error = ((0.5 * weight)- 0.8) ** 2
```

Graph

- Black dot: the current point of both error and weight
- The dotted circle is where we want to be (`error == 0`).



Key Points

- No matter where you are, the slope also points to the *minimum* point in the function.
- You can use this to find the minimum

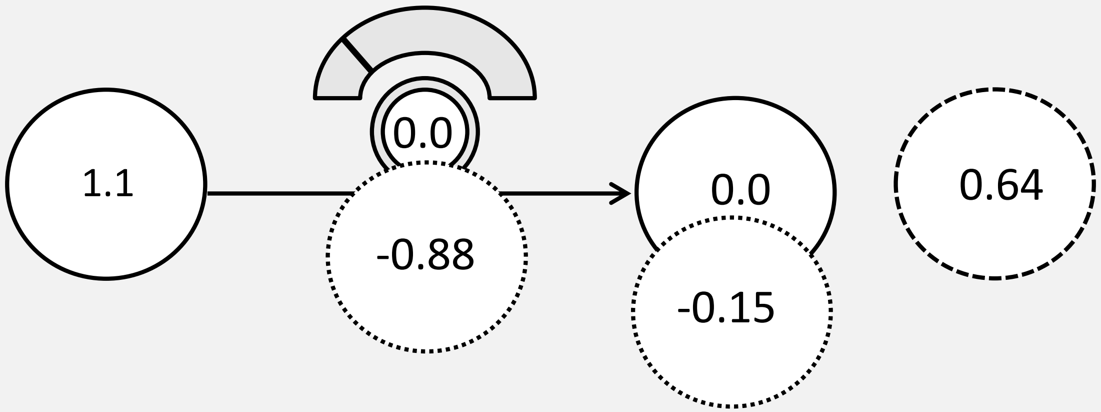
- **Can we find the minima?**
  - **Consider the following function**

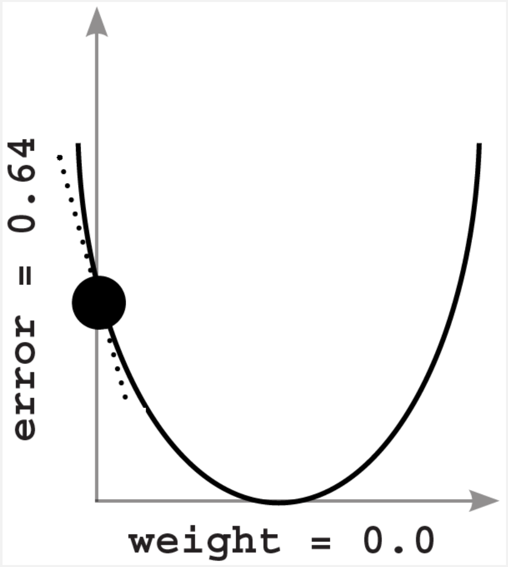
```
weight, goal_pred, input = (0.0, 0.8, 1.1)

for iteration in range(4):
    print("-----\nWeight:" + str(weight))
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
    print("Delta:" + str(delta) + " Weight Delta:" + str(weight_delta))
```

Note: `direction_and_amount` has been split over two lines

- First update:

A big weight increase





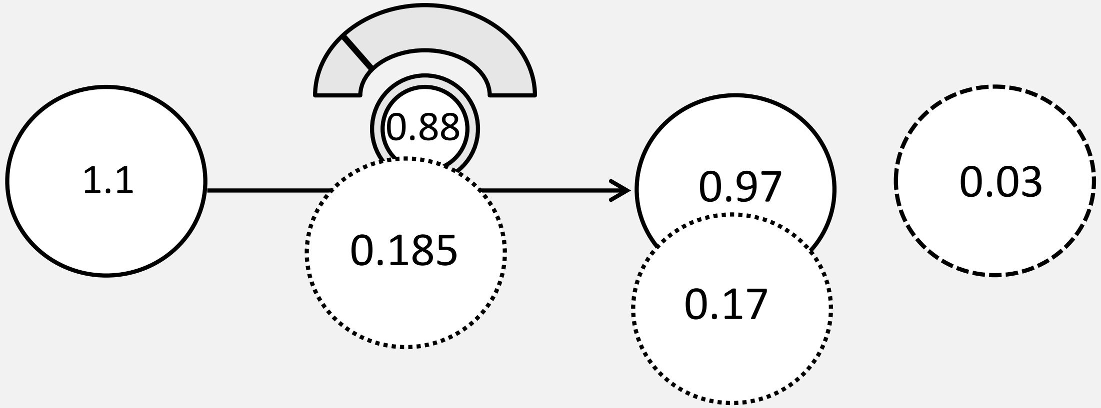
Code output:

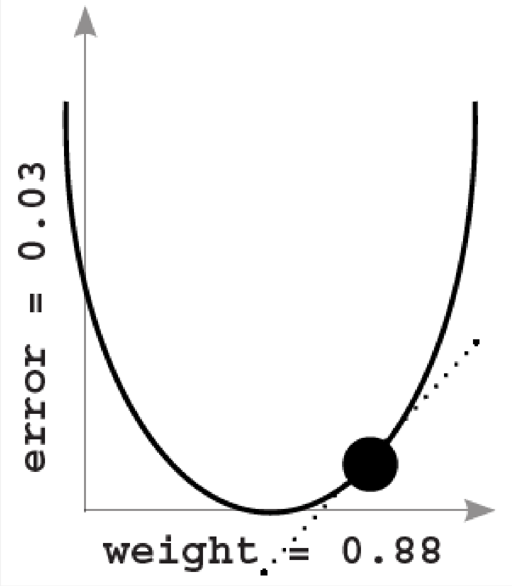
```
Weight:0.0
Error:0.64 Prediction:0.0
Delta:-0.8 Weight Delta:-0.88
```

Basic Gradient Descent in action

- Second update:

Overshoot from the first, head back the other way





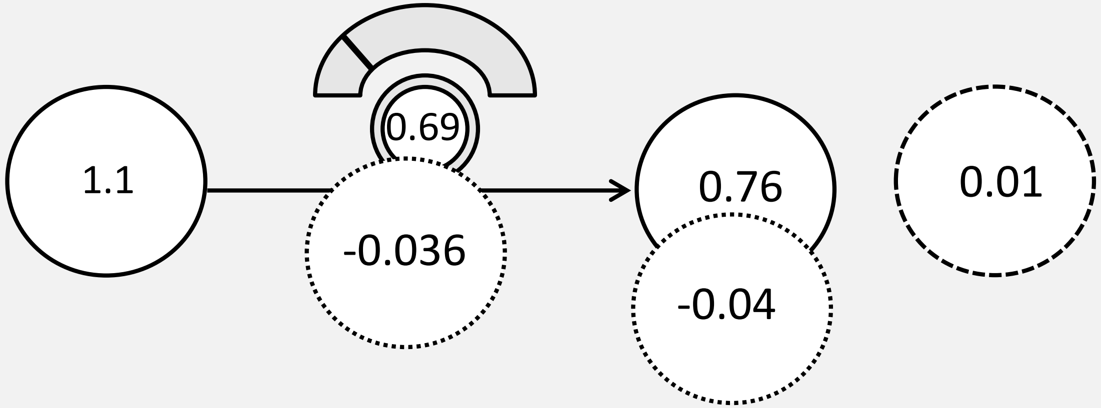
Code output:

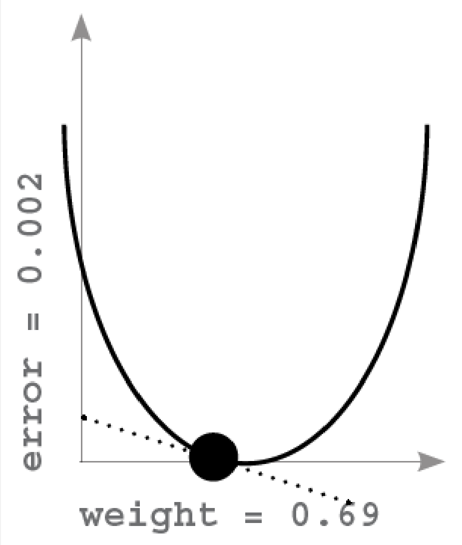
```
Weight:0.88
Error:0.028224 Prediction:0.968
Delta:0.168 Weight Delta:0.1848
```

Basic Gradient Descent in action

- Third update:

Small overshoot, head back, but only a little





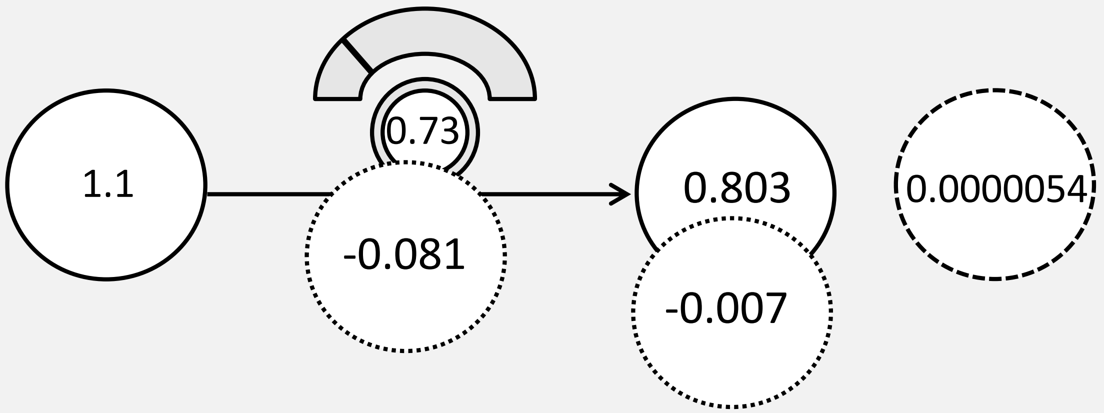
Code output:

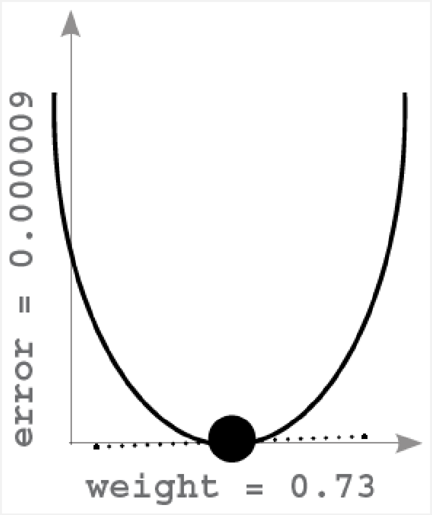
```
Weight:0.6952
Error:0.0012446784 Prediction:0.76472
Delta:-0.03528 Weight Delta:-0.038808
```

Basic Gradient Descent in action

- Last update:

Ok, pretty much there





Code output:

```
Weight:0.734008
Error:5.489031744e-05 Prediction:0.8074088
Delta:0.0074088 Weight Delta:0.00814968
```

Error/Weight Relationship

- **What is happening?**
  - Consider a function
    - A function defines some sort of relationship between the input number(s) and the output number(s).
  - Every function has what you might call *moving parts*
    - Pieces we can tweak or change to make the output different.

```
error = ((input * weight)- goal_pred) ** 2
```

  - What's controlling the relationship between `input` and the output (`error`)?

- **What is happening?**
  - What's controlling the relationship between `input` and the output (`error`)?

```
error = ((input * weight)- goal_pred) ** 2
```

- We could change `goal_pred` to reduce `error`
  - Essentially denying we missed
- We could change `input` to reduce `error`
  - This would not work in the real world!
- We could change the sqauring, or the mathematical operators
  - This is just changing how you calculate error in the first place.

- **What is happening?**
  - What's controlling the relationship between `input` and the output (`error`)?

```
error = ((input * weight)- goal_pred) ** 2
```

  - The only thing left we can change is `weight`
    - Adjusting this doesn't change your perception of the world, doesn't change your goal, and doesn't destroy your error measure.
  - Changing weight means the function conforms to the patterns in the data.

- **Key Message:**
  - Learning is adjusting the weight to reduce the `error` to 0
  - Knowing how to do this is all about understanding the relationship between `weight` and `error`
    - How does changing one variable effect the other?
  - This is the sensitivity between the two variables
  - Goal: know the direction and the amount that error changes when you change weight
    - This relationship is defined through the derivative of the error function

- **Derivatives**
  - With derivatives, you can pick any two variables in any formula, and know how they interact
- **Use in training a neural network**
  - A neural network is essentially a bunch of weights used to compute an error function
  - For any error function we can compute the relationship between any `weight` and the final `error` of the network.
    - With this information, we change each `weight` in the network to reduce `error` to 0

Universität Augsburg University

- **Gradient Descent for neural learning**

```
pred = input * weight
error = (pred - goal_pred) ** 2
delta = pred - goal_pred
weight_delta = delta * input
weight = weight - weight_delta
```

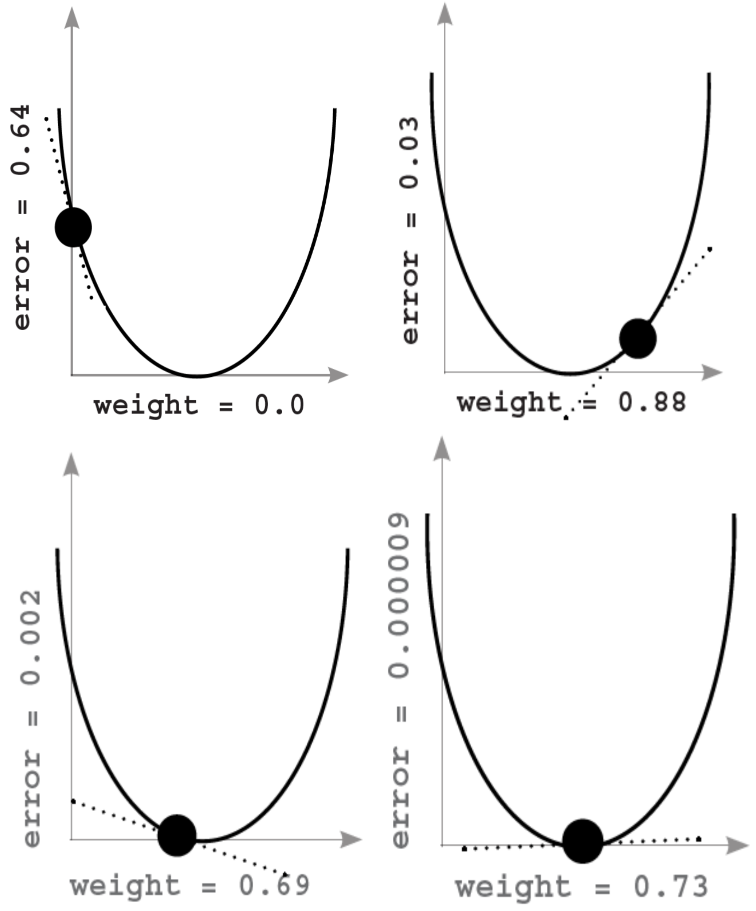
– `error` is a measure of how much the network missed by
  - We define `error` to be always positive
– `weighted_delta` is the derivate of `weight` and `error`
  - Defines the relationship between the weights and the total error

Universität Augsburg University

- **`weighted_delta` is the derivate**
  - For each point of our error function the derivative tells you how much `error` changes when we change `weight`
  - The derivative is always pointed in the opposite direction to the minimum point
  - To reduces `error` to zero, move the weight value opposite the gradient value

```
weight = weight - weight_delta
```

  - It is the minus sign in the weight update step that enables us to go in the opposite direction



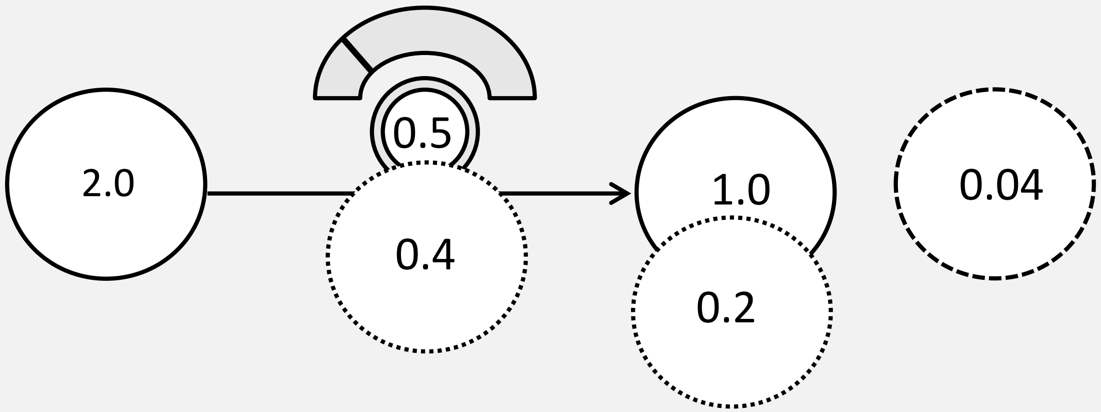
- **Divergence in Gradient Descent**
  - Consider the following function

```
weight, goal_pred, input = (0.5, 0.8, 2.0)

for iteration in range(20):
    print("-----\nWeight:" + str(weight))
    pred = input * weight
    error = (pred - goal_pred) ** 2
    delta = pred - goal_pred
    weight_delta = delta * input
    weight = weight - weight_delta
    print("Error:" + str(error) + " Prediction:" + str(pred))
```

Gradient Descent: Divergence

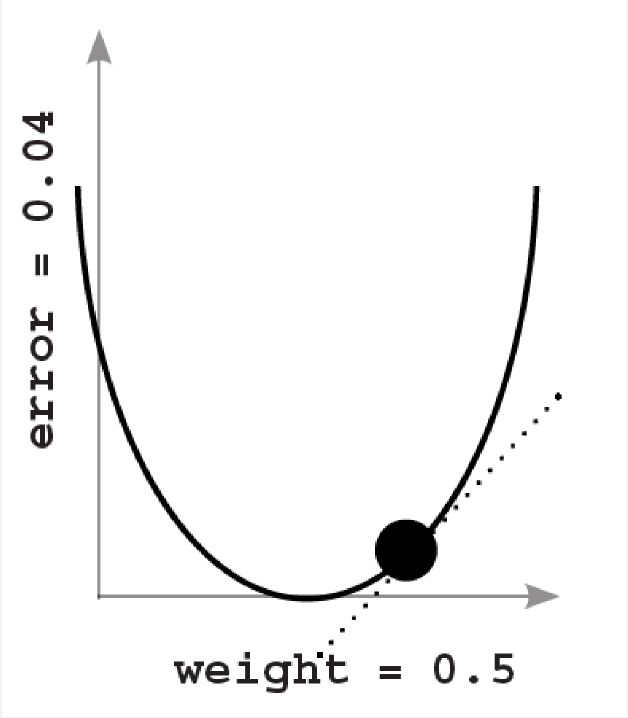
- First update:

A big weight increase



Code output:

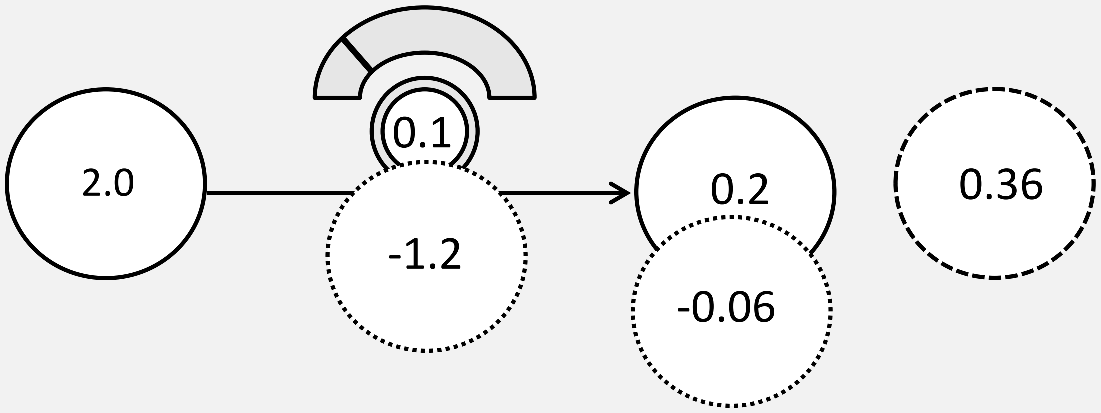
```
Error:0.64
Prediction:1.0
```

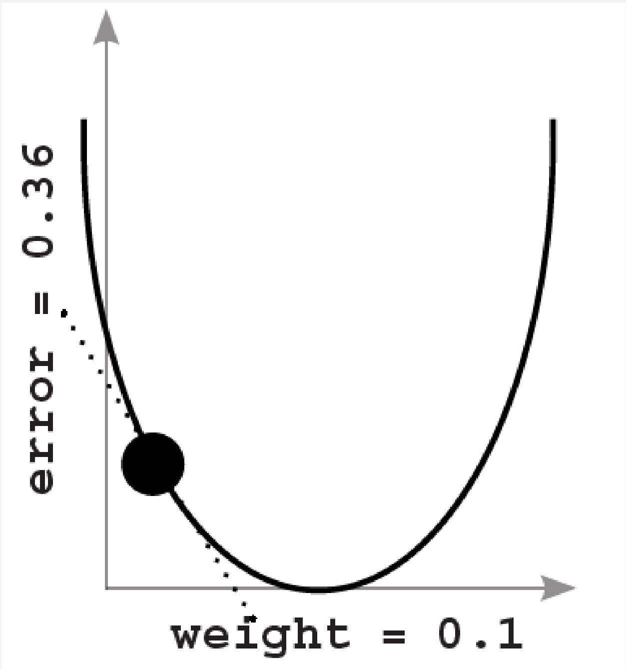


Gradient Descent Divergence

- Second update:

Large Overshoot





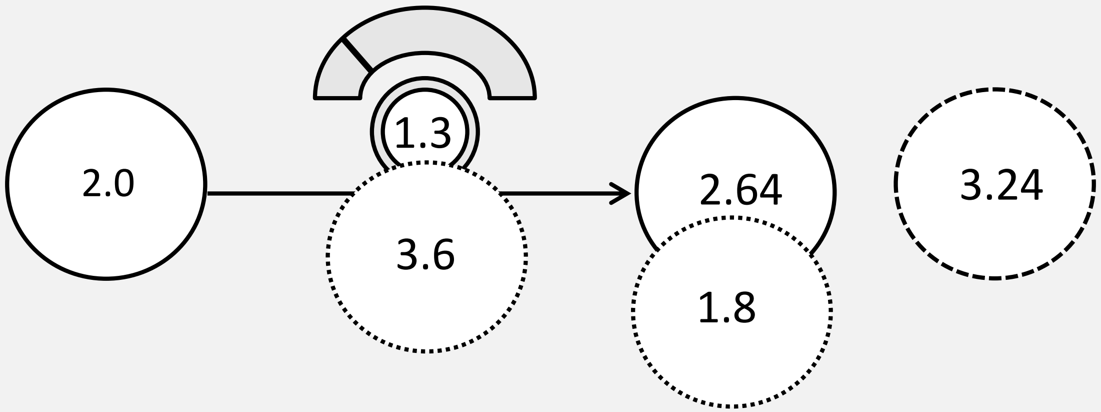
Code output:

```
Error:0.36
Prediction:0.2
```

Universität Augsburg University

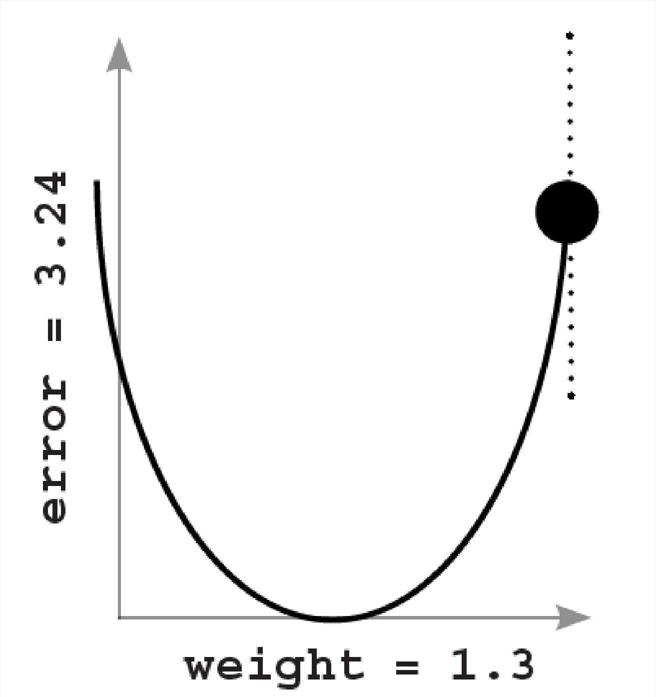
- Third update:

Large Overshoot



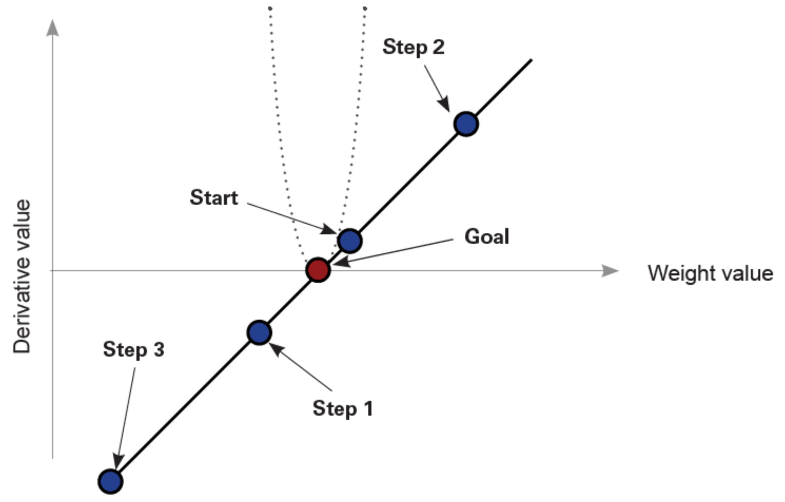
Code output:

```
Error:3.24
Prediction:2.6
```



- **Observation: The predictions are exploding**
  - At every update the network overcorrects the weights
    - They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step. In

```
Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6
...
Error:6.67e+14 Prediction:-25828031.8
Error:6.00e+15 Prediction:77484098.6
Error:5.40e+16 Prediction:-232452292.6
```



Gradient Descent: Divergence

- **Why are the predictions exploding?**
  - Consider how the weights are updated:

```
weight = weight - (input * (pred - goal_pred)))
```

  - If the input is sufficiently large, the weight update will also be large, even when the error is small.
  - When you have a large weight update and a small error, the network overcorrects
  - The bigger the error, the more the network overcorrects

- **How to prevent divergence:**
  - Introduce a new variable, `alpha`, to scale the weight updates

```
weight = weight - derivative
weight = weight - (alpha*derivative)
```

  - In most cases, this involves multiplying the weight update by a single real-valued number between 0 and 1

Universität Augsburg University

- **How to set Alpha?**

```
weight = weight - (alpha*derivative)
```

– Empirically, watching errors over time

- If it starts diverging (going up), then the alpha is too high and needs to be decreased
- If learning is happening too slowly, then the alpha is too low and should be increased

- **Divergence Example**
  - Effect of introducing `alpha` into previous example

```
weight = 0.5
goal_pred = 0.8
input = 2
alpha = 0.1

for iteration in range(20):
    pred = input * weight
    error = (pred - goal_pred) ** 2
    derivative = input * (pred - goal_pred)
    weight = weight - (alpha * derivative)
    print("Error:" + str(error) + " Prediction:" + str(pred))
```

Universität Augsburg University

- **Divergence Example**
  - Effect of introducing `alpha` into previous example
    - The neural network can now make good predictions again

```
Using:
weight = weight – derivative

Error:0.04 Prediction:1.0
Error:0.36 Prediction:0.2
Error:3.24 Prediction:2.6
...
Error:6.67e+14 Prediction:-25828031.8
Error:6.00e+15 Prediction:77484098.6
Error:5.40e+16 Prediction:-232452292.6
```

```
Using:
weight = weight – (alpha*derivative)

Error:0.04 Prediction:1.0
Error:0.0144 Prediction:0.92
Error:0.005184 Prediction:0.872
...
Error:1.146e-09 Prediction:0.800033853319
Error:4.126e-10 Prediction:0.800020311991
Error:1.485e-10 Prediction:0.800012187195
```

Basic Gradient Descent for Neural Learning

```
pred = input * weight
error = (pred - goal_pred) ** 2
derivative = input * (pred - goal_pred)
weight = weight - (alpha * derivative)
```

- `Pred` is the output of the network given the current input
  - We want the network to learn to make correct predictions
- `error` is a measure of how much the network missed by
  - We define `error` to be always positive, we learn by reducing the `error` to zero
- `derivative` is the derivate of `weight` and `error`
  - Predicts both direction and amount to adjust the weights
- `Alpha` scales the weight update
  - Helps minimise divergence effects when the input is large