UC Berkeley Extension

Machine Learning With TensorFlow

X433.7-001 (2 semester units in COMPSCI)

Instructor Alexander I. Iliev, Ph.D.

Course Content Outline

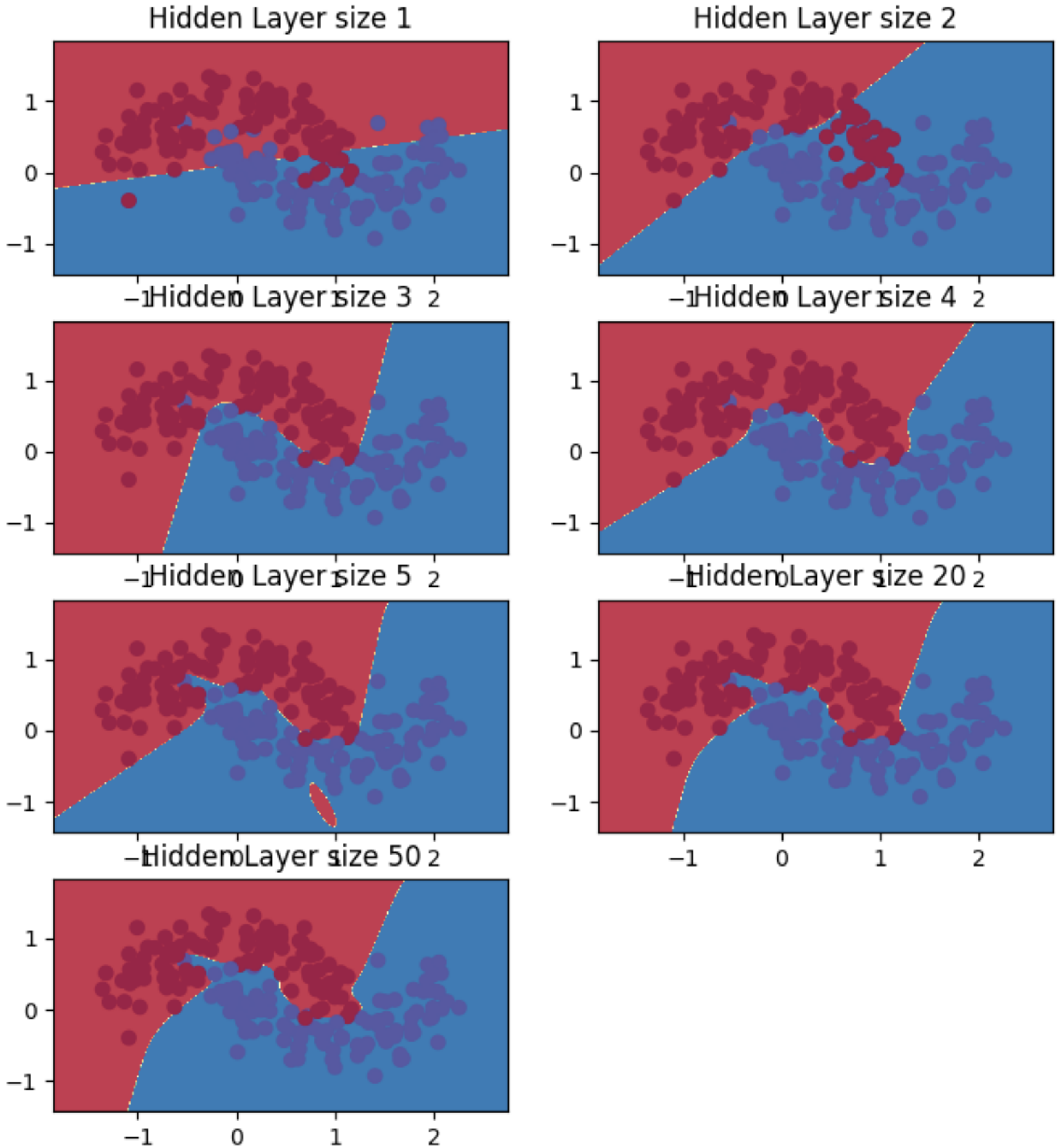
- **Machne Learning**
- Linear and Logistic Regression
- Softmax classification
- Multi-layer Neuaral Network
- Gradient descent and Backpropagation

- **Neural Networks**
- Object recognition with Convolutional Neural Network (CNN)
- Activation Functions
- Common layers: Conv and Pooling Layers
- CNN Overview — Midterm / Project proposal due (30pts)

- **Working with images**
- Normlization
- Loading Images
- Image formats and manipulation

- **CNN Implementation**
- Training
- Recurrent Neural Network (RNN)
- Project Presentations 1/2

- **Project Presentations**
- Project Presentation 2/2 — Final Project (40pts)



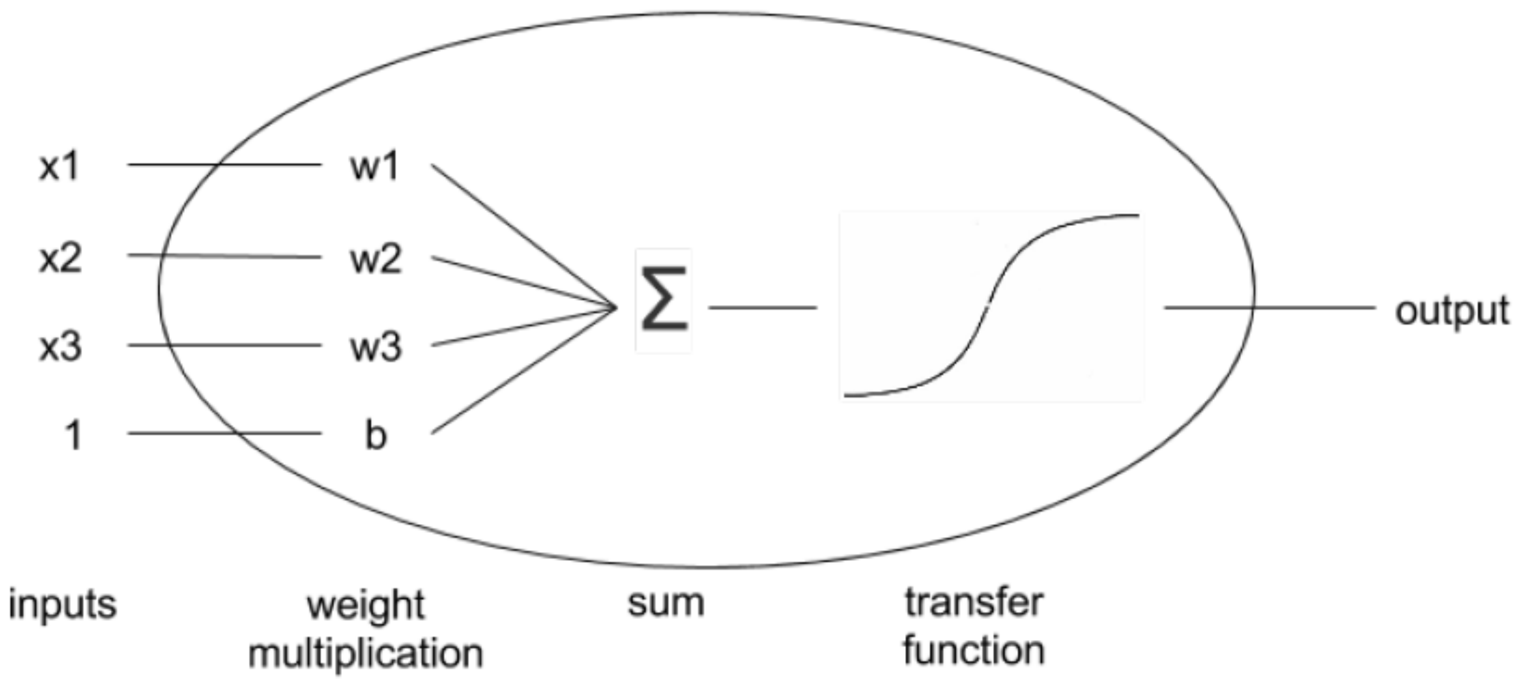
Hidden Layer size 1
Hidden Layer size 2
Hidden Layer size 3
Hidden Layer size 4
Hidden Layer size 5
Hidden Layer size 20
Hidden Layer size 50
1
0
−1
−1
0
1
2

Multi-layer neural networks

- We have seen many simple neural networks
- Both linear and logistic regression models are single neurons that:
  - perform a weighted sum of the input features. Bias can be thought of as the weight of an input feature that equals 1 for every example. We call that a *linear combination* of the features
  - Then apply an activation or transfer function to calculate the output. In the case of the lineal regression, the transfer function is the identity (i.e. same value), while the logistic uses the sigmoid as the transfer.

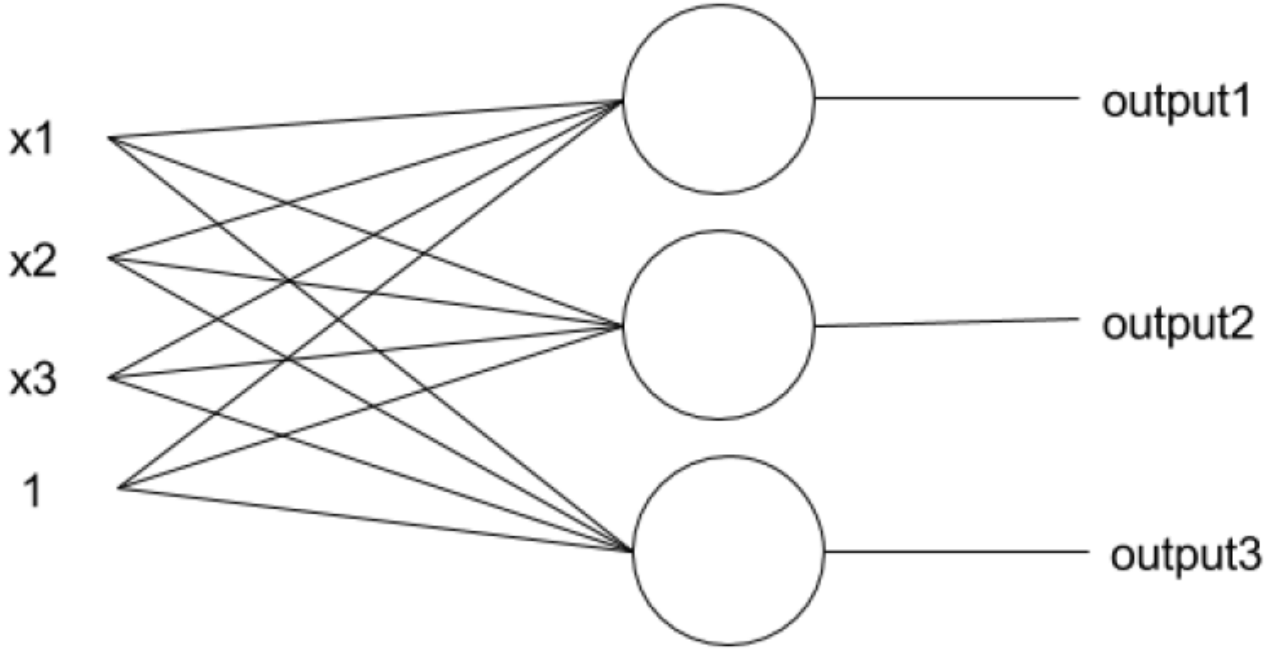
Multi-layer neural networks

- The following diagram represents each neuron inputs, processing and output:



Multi-layer neural networks

- In the case of softmax classification, we used a network with 3 neurons - one for each possible output class:



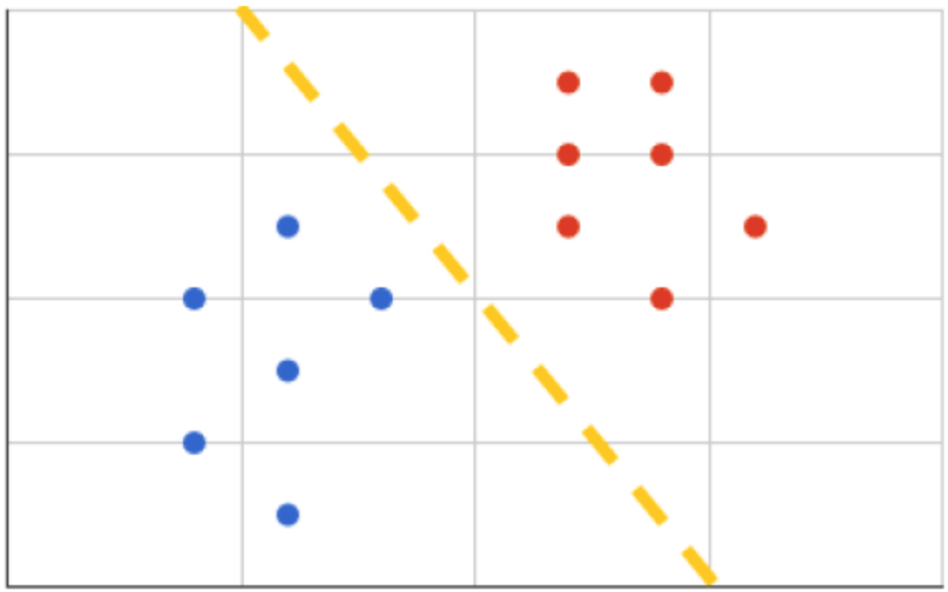
Multi-layer neural networks

- In order to resolve more difficult tasks, like reading handwritten digits, or identifying cats and dogs on images, we are going to need a more developed model.
- Lets start with a simple example:
  - Suppose we want to build a network that learns how to fit the XOR (eXclusive OR) Boolean operation:

XOR operation truth table

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

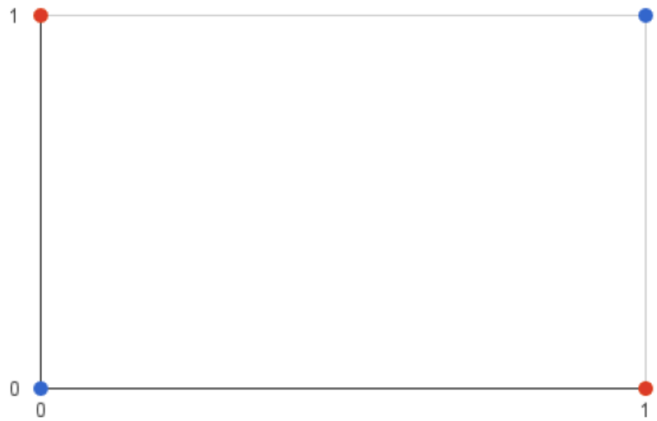
Multi-layer neural networks



- In the chart we can see example data samples as dots, with their associated class as the color.
- As long as we can find that yellow line completely separating the red and the blue dots in the chart, the sigmoid neuron will work fine for that dataset.

Multi-layer neural networks

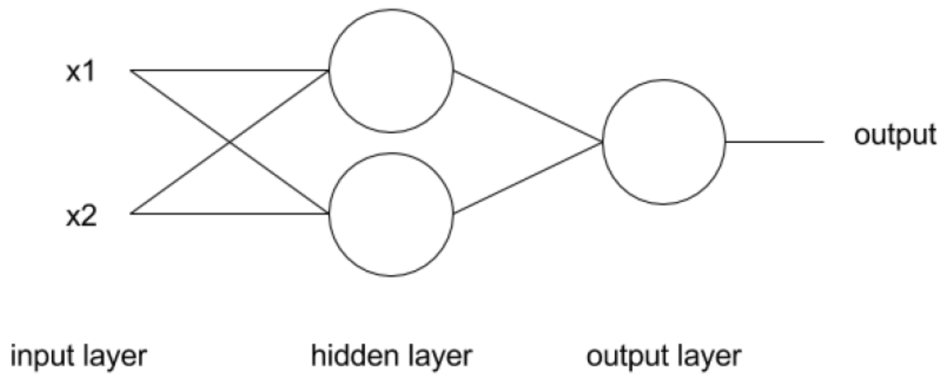
- Let’s look at the XOR gate function chart:



- We can’t find a single straight line that would split the chart, leaving all the 1s (red dots) in one side and 0s (blue dots) in the other
- That’s because the XOR function output is not linearly separable

Multi-layer neural networks

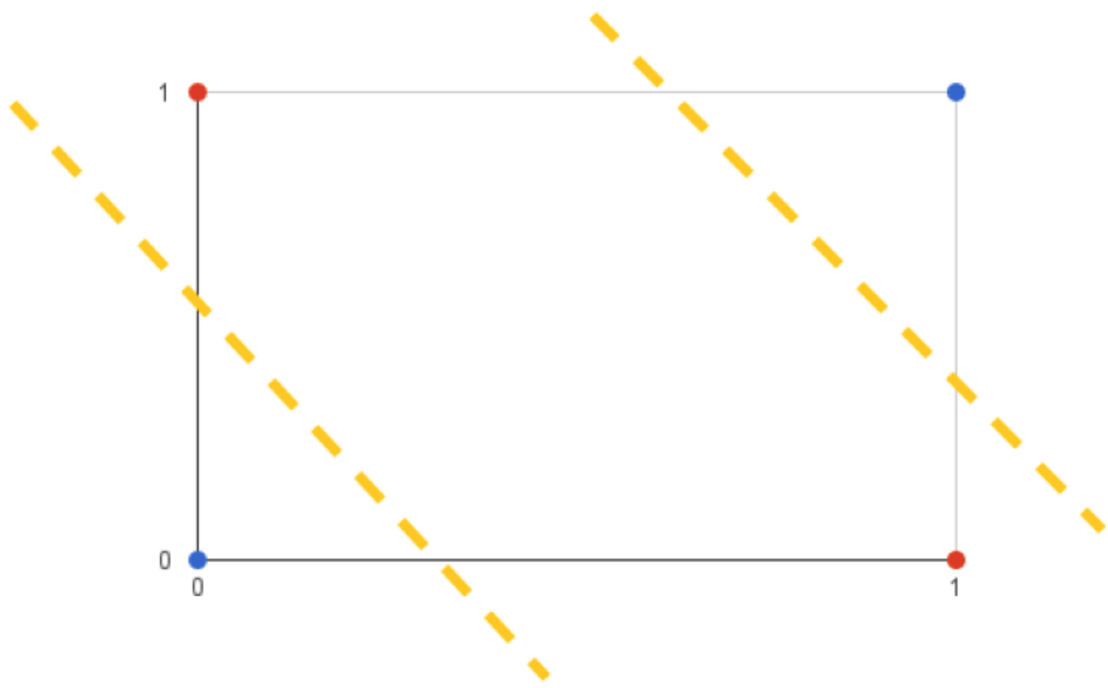
- Using more neurons between the input and the output of the network, introducing the *hidden layer*:



- You can think of it as allowing our network to ask multiple questions to the input data, one question per neuron on the hidden layer
- Deciding the output based on the answers of those questions

Multi-layer neural networks

- Graphically, we are allowing the network to draw more than one single separation line:



Gradient descent and backpropagation

- Gradient descent is an algorithm to find the points where a function achieves its minimum value.
- Remember that we can define learning as improving the model parameters in order to minimize the loss through several training steps
- With that concept, applying gradient decent to find the minimum of the loss function will result in our model learning from our input data

Gradient descent and backpropagation

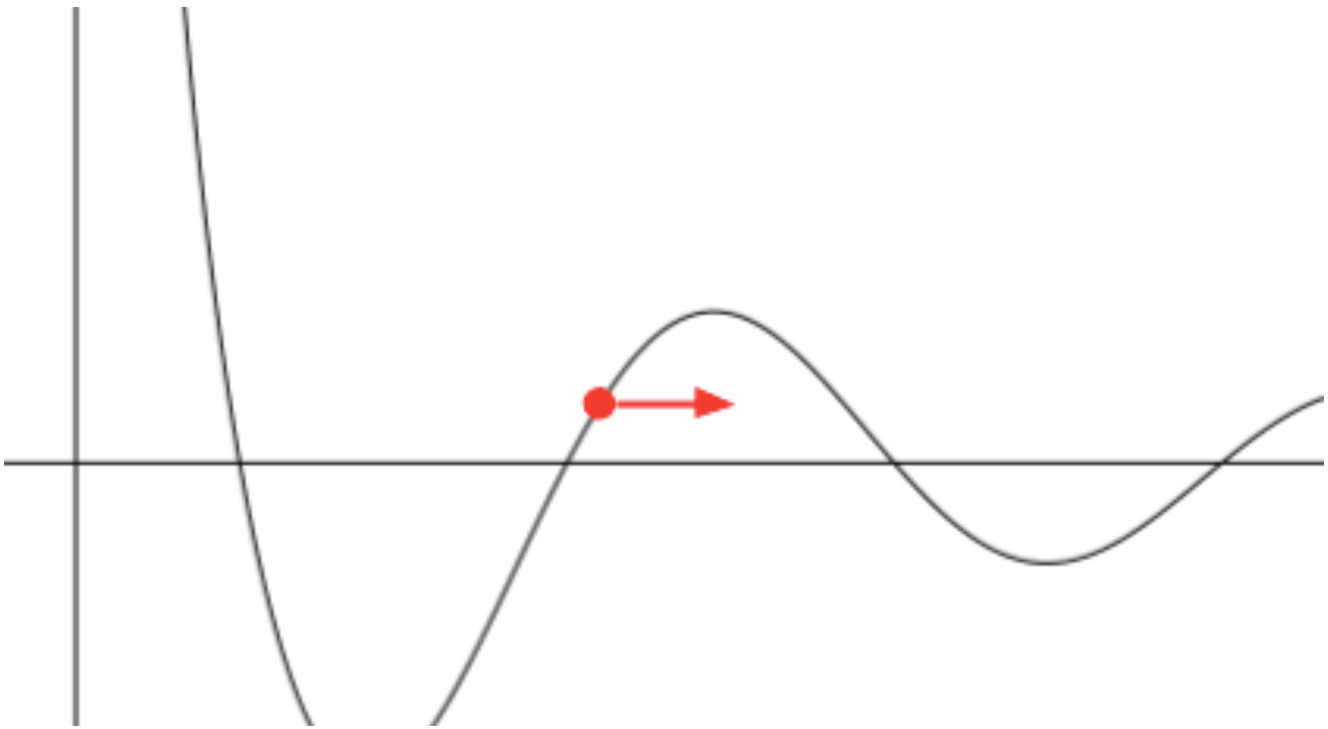
- What is a gradient?
- The gradient is a mathematical operation, generally represented with the ∇ symbol (nabla greek letter).
- It is analogous to a derivative, but applied to functions that input a vector and output a single value; like our loss functions do
- The output of the gradient is a vector of partial derivatives, one per position of the input vector of the function

$$\nabla \equiv \left(\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_N} \right)$$

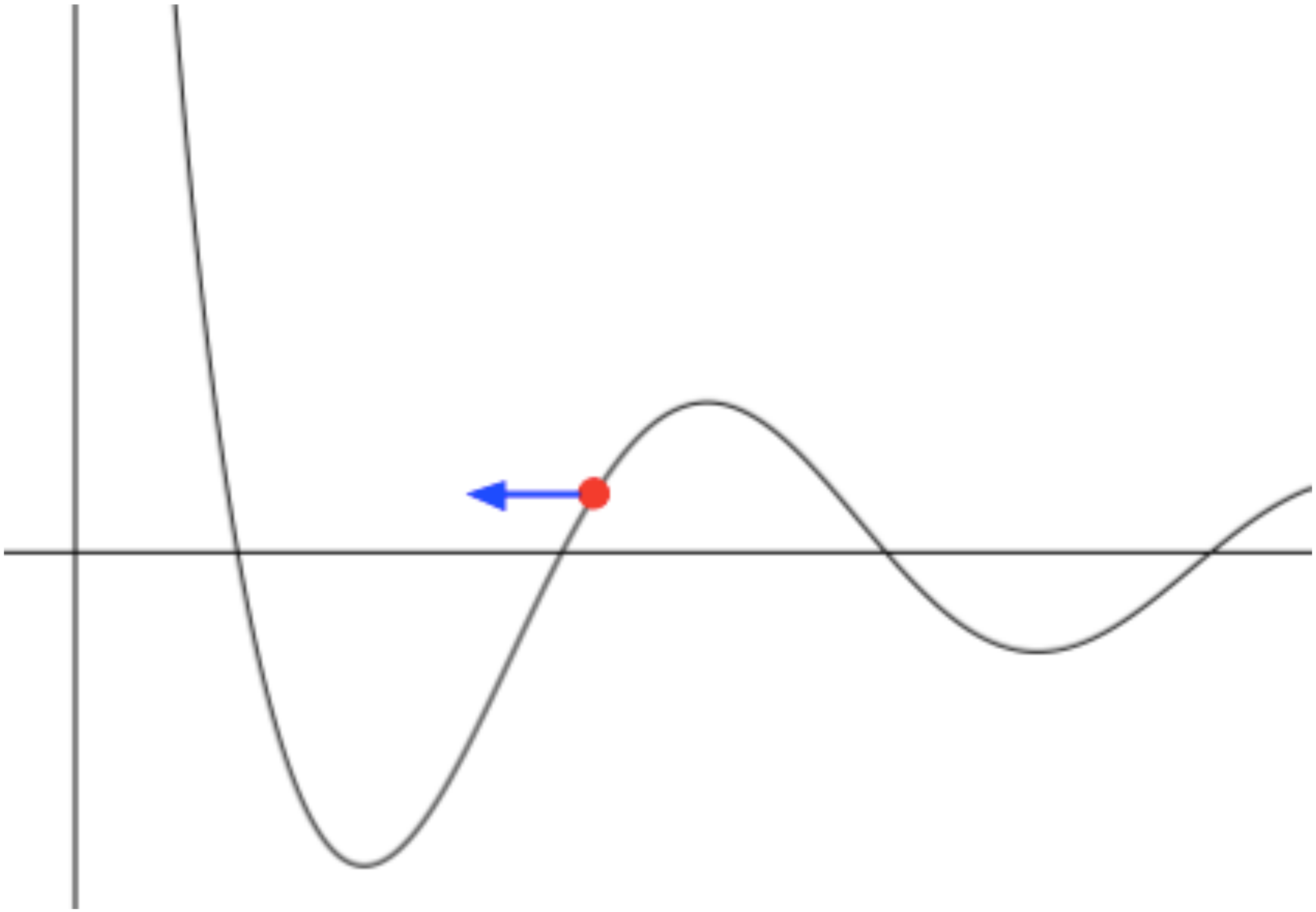
Gradient descent and backpropagation

- Few caveats:
  - When we talk about input variables of the loss function, we are referring to the model weights, not that actual dataset features inputs
  - The latter are fixed by our dataset and cannot be optimized
  - The partial derivatives we calculate are with respect of each individual weight in the inference model
  - We care about the gradient because its output vector indicates the direction of maximum growth for the loss function
  - You could think of it as a little arrow that will indicate in every point of the function where you should move to increase its value: … *see next slide*

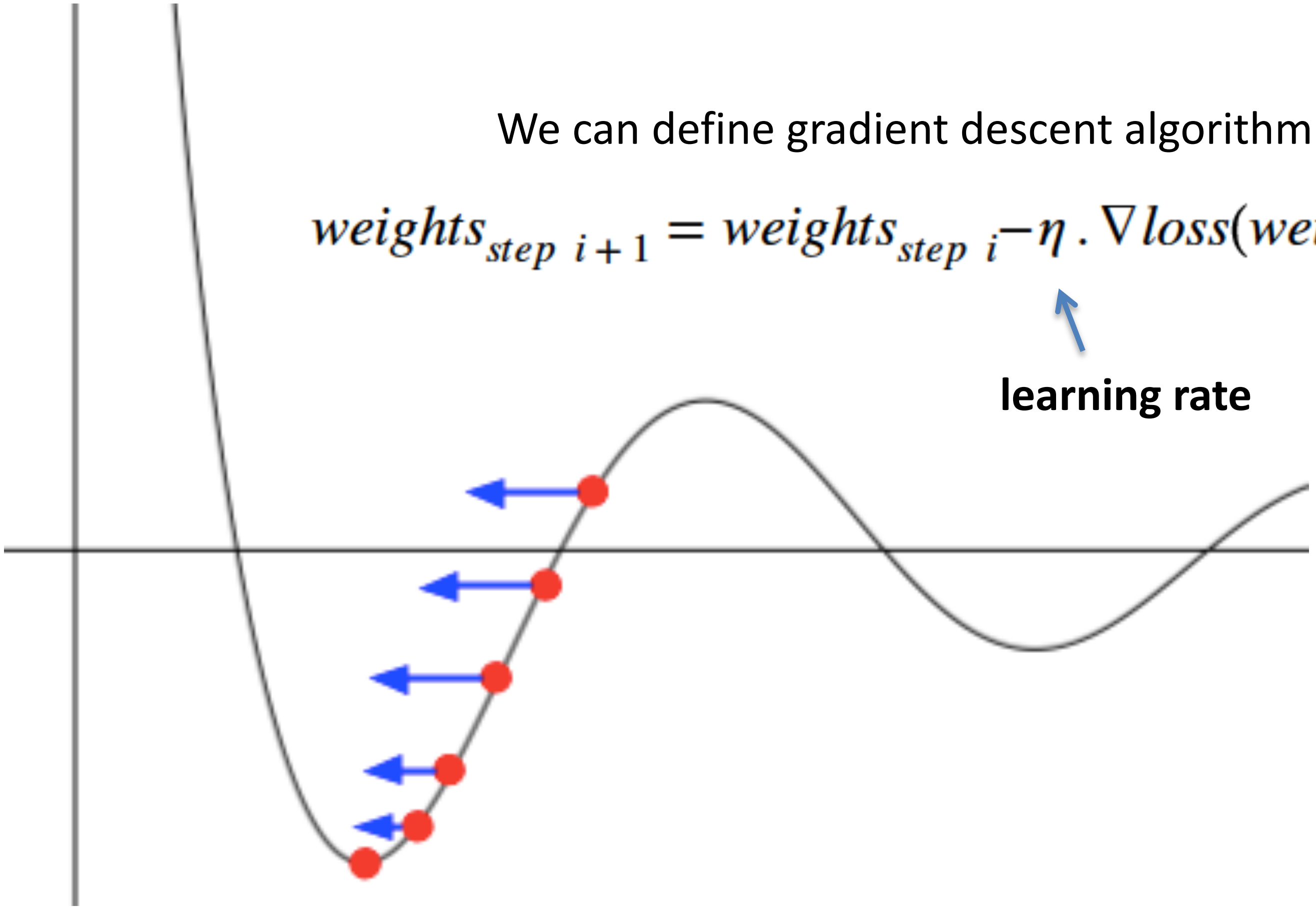
Gradient descent and backpropagation



Gradient descent and backpropagation



Gradient descent and backpropagation



We can define gradient descent algorithm as:

$$weights_{step\ i+1} = weights_{step\ i} - \eta \,.\, \nabla loss(weights_{step\ i})$$

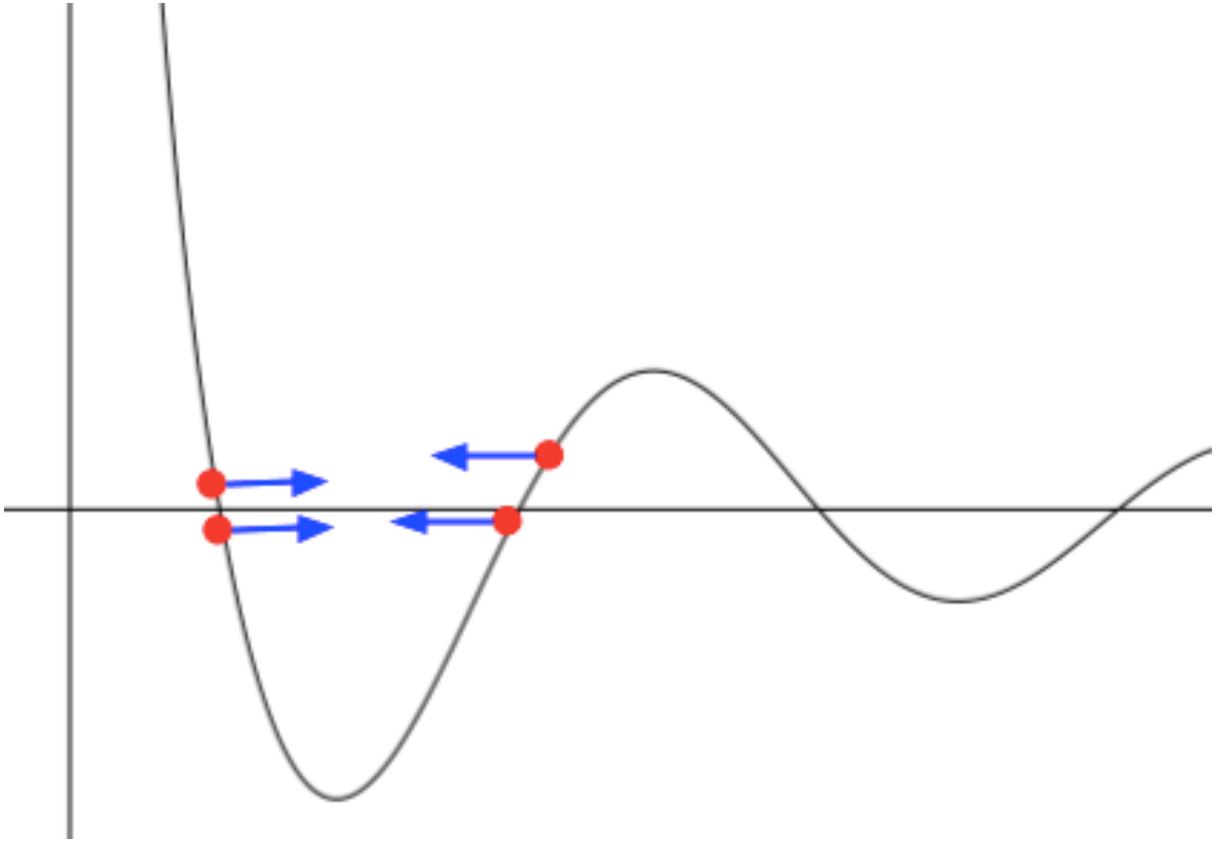
learning rate

Gradient descent and backpropagation

- The learning rate is not a value that model will infer
- It is a hyperparameter, or a manually configurable setting for our model
- We need to figure out the right value for it:
  - If it is too small then it will take many learning cycles to find the loss minimum
  - If it is too large, the algorithm may simply “skip over” the minimum and never find it, jumping cyclically.
- That’s known as overshooting.

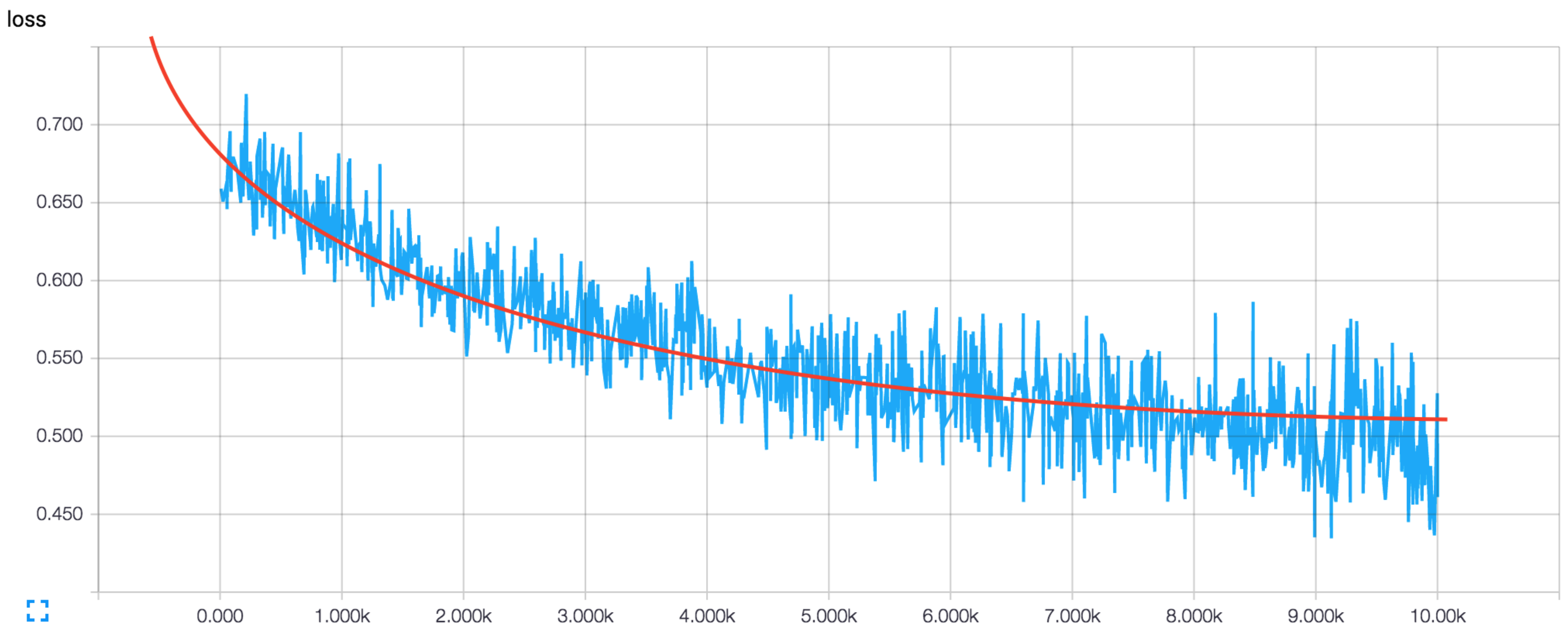
Gradient descent and backpropagation

- Here is what overshooting looks like:



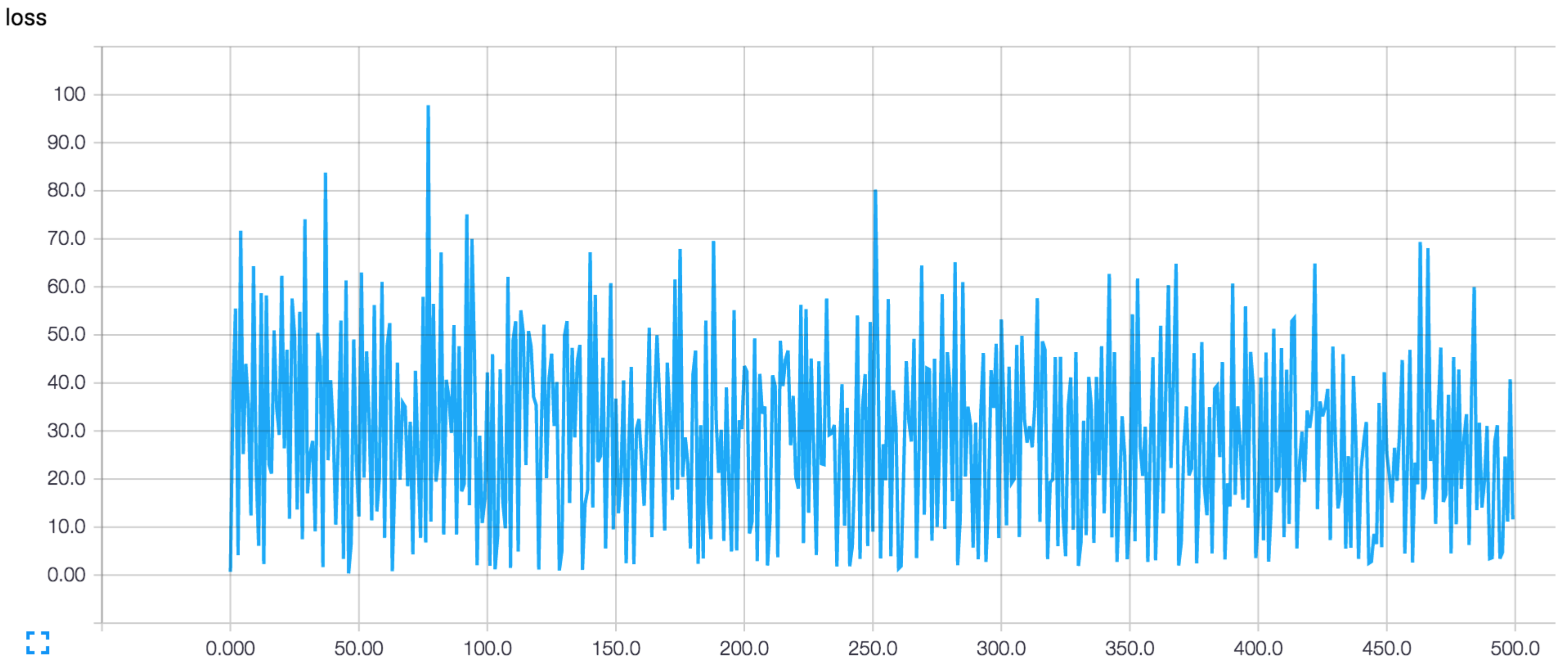
Gradient descent and backpropagation

- This is how a well behaving loss should diminish through time, indicating a good learning rate:

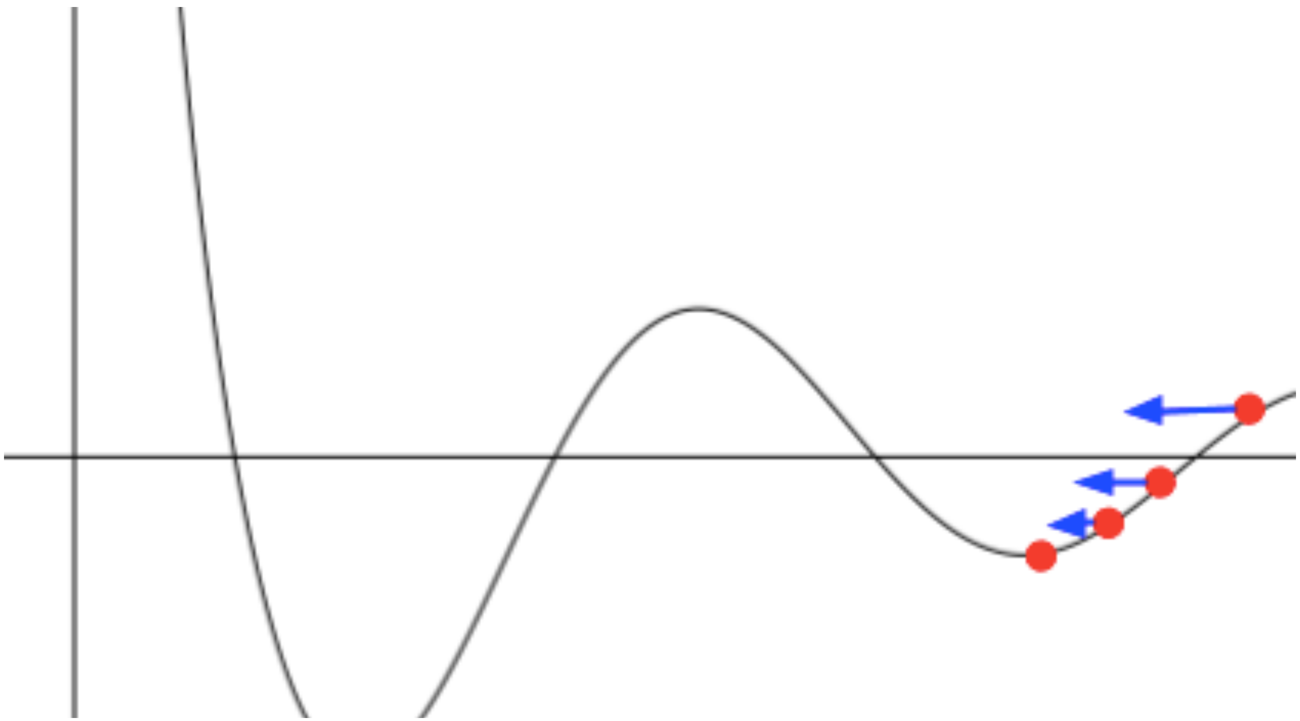


Gradient descent and backpropagation

- This is what it looks like when it is overshooting:



Gradient descent and backpropagation



Gradient descent and backpropagation

- Tensorflow includes the method tf.gradients (v.1.x-2.x) to symbolically compute the gradients of the specified graph steps and output that as tensors

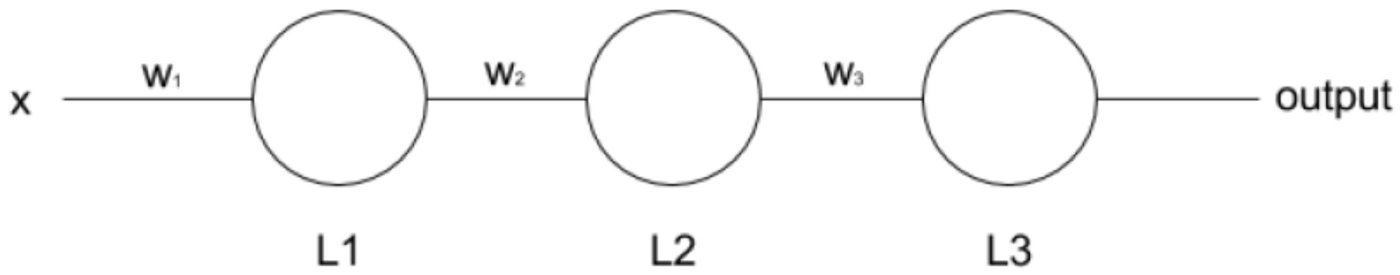
- We don't need to manually call, because it also includes implementations of the gradient descent algorithm, among others

- We are going to present the backpropagation next

- It is a technique used for efficiently computing the gradient in a computational graph

Gradient descent and backpropagation

- Let's assume a simple network, with one input, one output, and two hidden layers with a single neuron.
- Both hidden and output neurons will be sigmoids and the loss will be calculated using cross entropy.
- Such a network should look like this:



Gradient descent and backpropagation

- Let's define L_1 as the output of first hidden layer, L_2 the output of the second, and L_3 the final output of the network:

$$L1 = sigmoid(w_1 . x)$$

$$L2 = sigmoid(w_2 . L1)$$

$$L3 = sigmoid(w_3 . L2)$$

- The loss of the network will be:

$$loss = cross\_entropy(L3, y_{expected})$$

Gradient descent and backpropagation

- To run one step of gradient decent, we need to calculate the partial derivatives of the loss function with respect of the three weights in the network.

- We will start from the output layer weights, applying the chain rule:

$$\frac{\partial loss}{\partial w_3} = cross\_entropy'\left(L3, y_{expected}\right) . sigmoid'\left(w_3 . L2\right) . L2$$

- L_2 is just a constant for this case as it doesn't depend on w_3

Gradient descent and backpropagation

- To simplify the expression we could define:

$$loss' = cross\_entropy'\left(L3, y_{expected}\right)$$

$$L3' = sigmoid'\left(w_3 \cdot L2\right)$$

- The resulting expression for the partial derivative would be:

$$\boxed{\frac{\partial loss}{\partial w_3} = loss' \cdot L3' \cdot L2}$$

Gradient descent and backpropagation

- Now let's calculate the derivative for the second hidden layer weight, w_2:

$$L2' = sigmoid'(w_2 \cdot L1)$$

$$\frac{\partial loss}{\partial w_2} = loss' \cdot L3' \cdot L2' \cdot L1$$

- And finally the derivative for w_1:

$$L1' = sigmoid'(w_1 \cdot x)$$

$$\boxed{\frac{\partial loss}{\partial w_1} = loss' \cdot L3' \cdot L2' \cdot L1' \cdot x}$$

Gradient descent and backpropagation

- We notice a pattern:
  - The derivative on each layer is the product of the derivatives of the layers after it by the output of the layer before.
  - That’s the magic of the chain rule and what the algorithm takes advantage of.
- We go forward from the inputs calculating the outputs of each hidden layer up to the output layer.
- Then we start calculating derivatives going backwards through the hidden layers and propagating the results in order to do less calculations by reusing all the elements already calculated
- That’s the origin of the name backpropagation.

Object Recognition and Classification

- At this point, we should have a basic understanding of **TensorFlow** and its best practices
- We can now build a model capable of object recognition and classification
- Building this model expands on the fundamentals that have been covered so far while adding terms, techniques and fundamentals of computer vision
- The technique used in training the model has become popular recently due to its accuracy across challenges

Data Science

Lecture 7 …

Images and TensorFlow …

Object Recognition and Classification

- ImageNet, a database of labeled images, is where computer vision and deep learning saw a recent rise in popularity

- Convolutional Neural Networks (CNNs) primarily used for computer vision related tasks but are not limited to working with images

- For images, the values in the tensor are pixels ordered in a grid corresponding with the width and height of the image

Object Recognition and Classification

- The dataset used in training this CNN model is a subset of the images available in ImageNet named the Stanford's Dogs Dataset - http://vision.stanford.edu/aditya86/ImageNetDogs/

n02110185_712.jpg n02110185_4186.jpg n02110185_1532.jpg

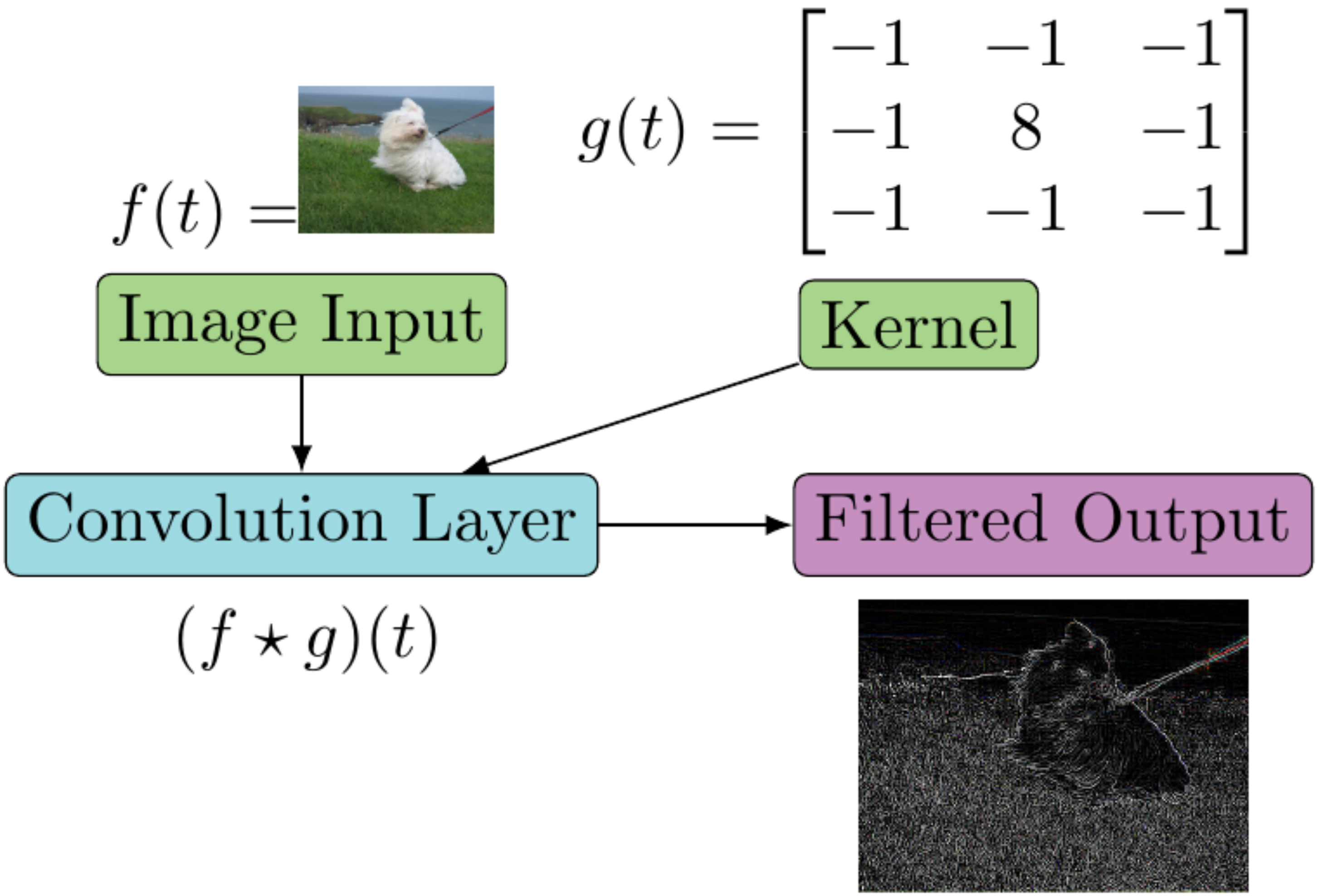
Convolutional Neural Networks

- A CNN is a neural network which has at least one-layer tf.nn.conv2d that performs a convolution between its input ***f*** and a configurable kernel ***g*** generating the layer's output

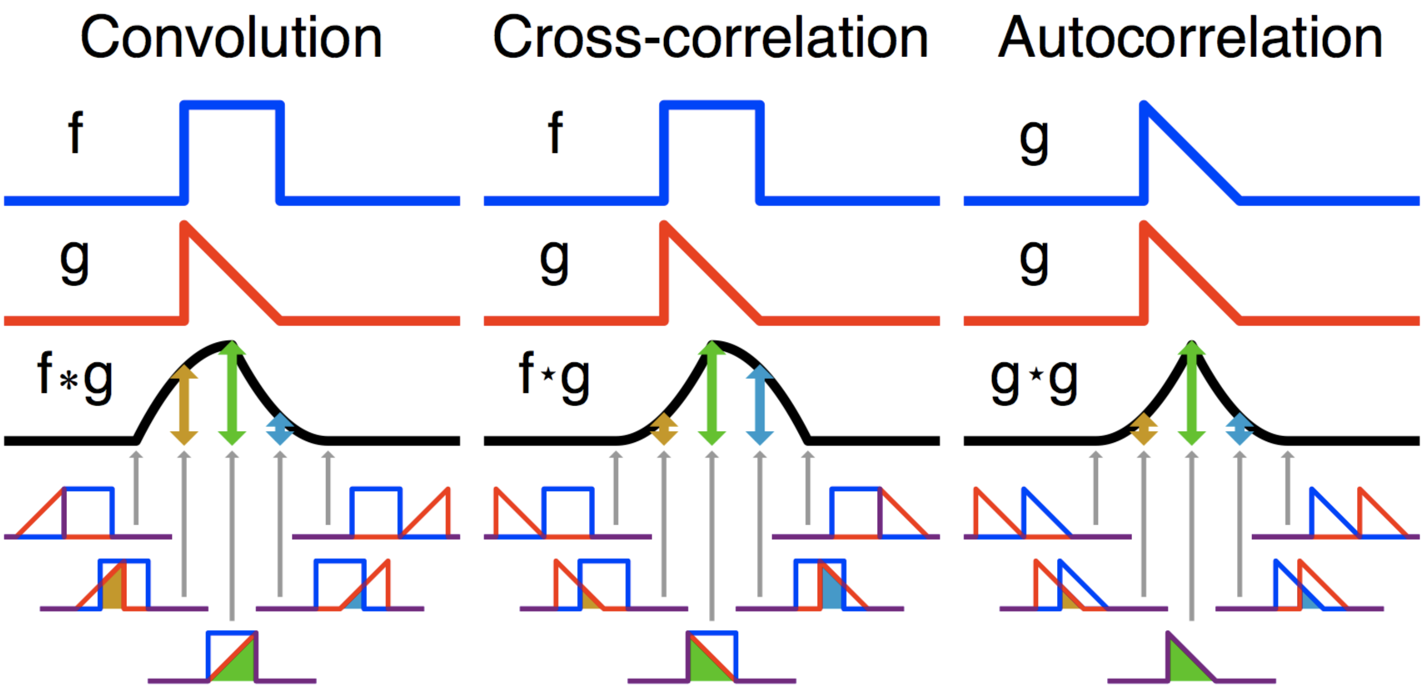
- In a simplified definition, a convolution's goal is to apply a kernel (filter) to every point in a tensor and generate a filtered output by sliding the kernel over an input tensor

- An example of the filtered output is edge detection in images

Convolutional Neural Networks



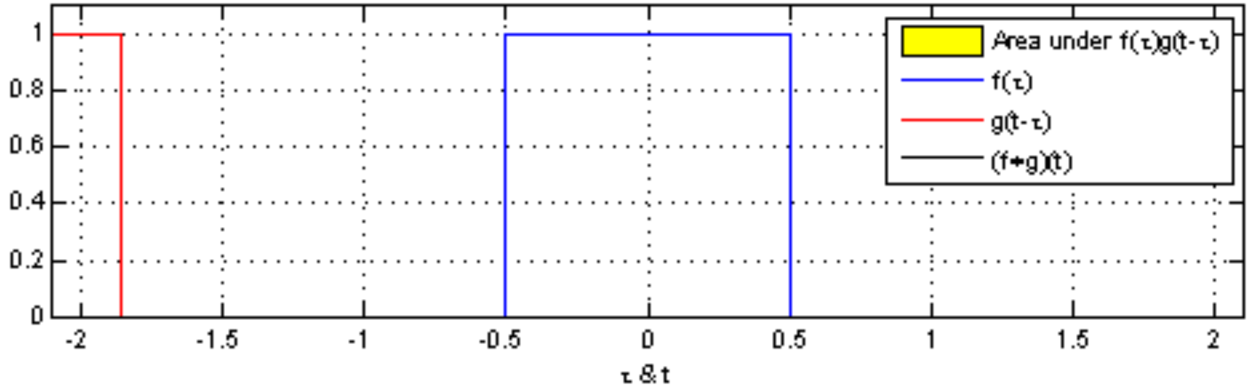
Convolution vs. Correlation



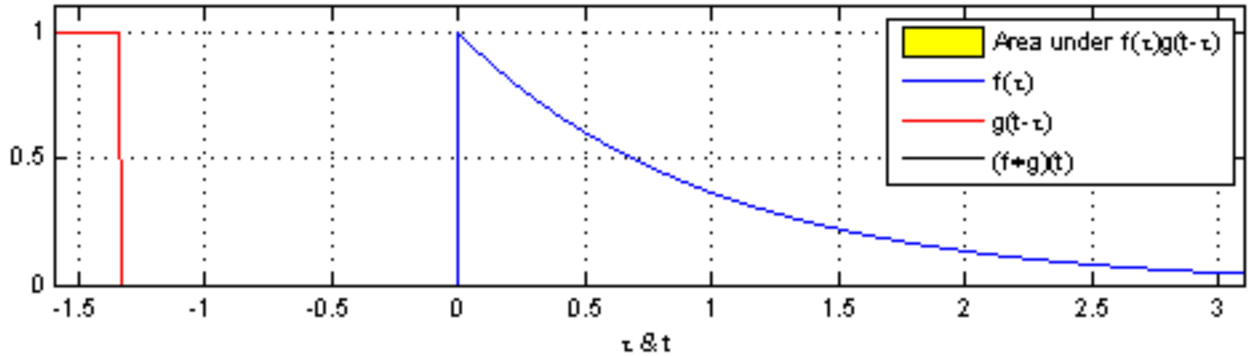
Source Wikipedia

Convolution

- Convolving two signals:



box vs box



spiky vs box

Source Wikipedia

Convolutional Neural Networks

- In CNN clusters neurons will activate based on patterns learned from training

Example: after training, a CNN will have certain layers that activate when a horizontal line passes through it

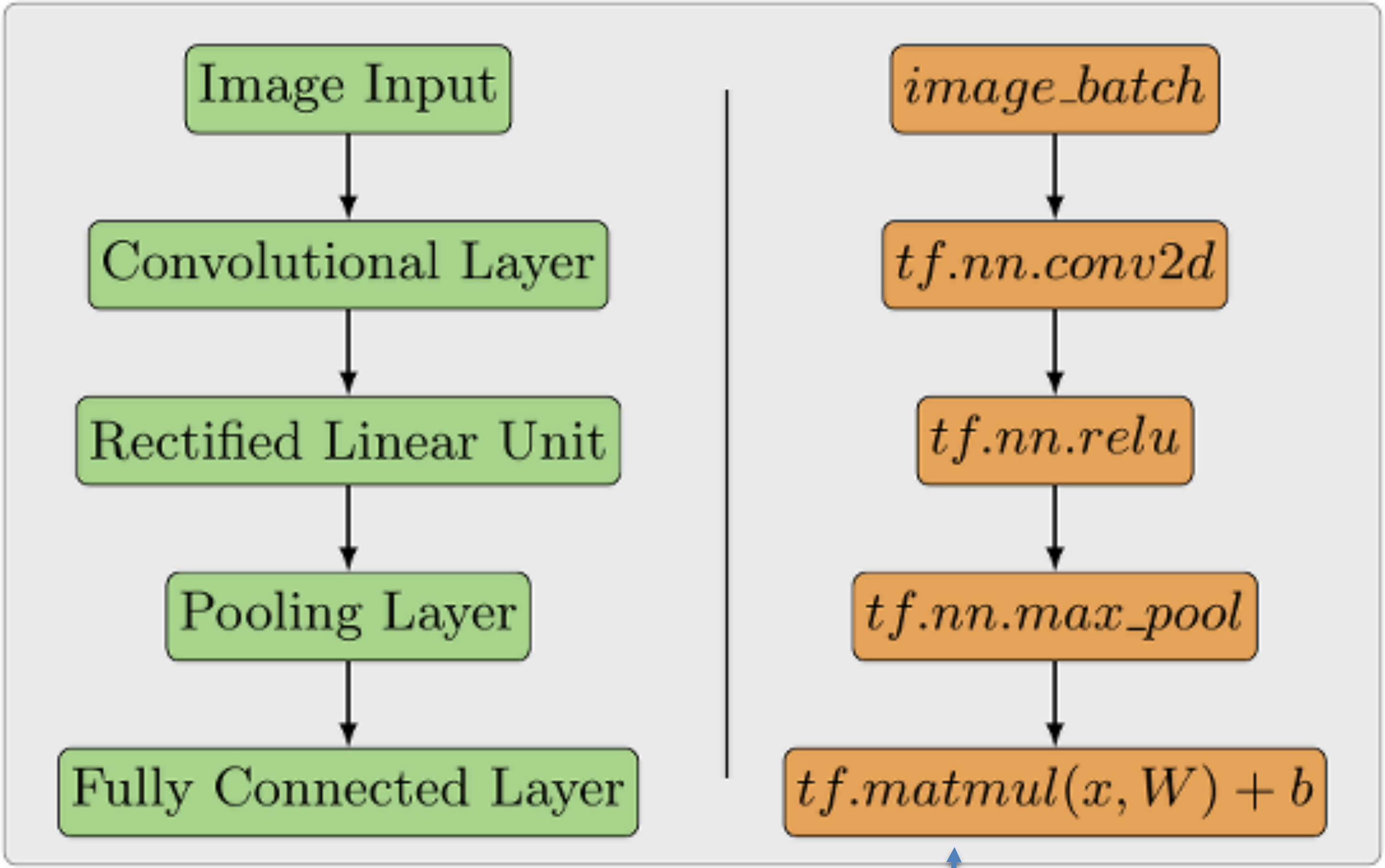
- Layering multiple simple patterns to match complex patterns, a.k.a. filters or kernels is what we need to do

- Our goal is to adjust these kernel weights until they accurately match the training data, often accomplished by combining multiple different layers and learning weights using gradient descent

Convolutional Neural Networks

- A simple CNN architecture may combine different types of layers:
  - a convolutional layer tf.nn.conv2d (v.1.x-2.x), non-linearity layer tf.nn.relu (v.1.x-2.x), pooling layer tf.nn.max_pool (v.1.x-2.x) and a fully connected layer tf.matmul (v.1.), tf.linalg.matmul (v.2.x)
- Without these layers, it's difficult to match complex patterns because the network will be filled with too much information
- A well-designed CNN architecture highlights important information while ignoring noise

Convolutional Neural Networks



tf.linalg.matmul

Convolutional Neural Networks

```
image_batch = tf.constant([
        [  # First Image
            [[0, 255, 0], [0, 255, 0], [0, 255, 0]],
            [[0, 255, 0], [0, 255, 0], [0, 255, 0]]
        ],
        [  # Second Image
            [[0, 0, 255], [0, 0, 255], [0, 0, 255]],
            [[0, 0, 255], [0, 0, 255], [0, 0, 255]]
        ]

    ])
image_batch.get_shape()
```

The output from executing the example code is:

```
 TensorShape([Dimension(2), Dimension(2), Dimension(3), Dimen-
sion(3)])
```

Convolutional Neural Networks

- It’s important to note each pixel maps to the height and width of the image. Retrieving the first pixel of the first image requires each dimension accessed as follows:

```
sess.run(image_batch)[0][0][0]
```

- The output from executing the example code is:

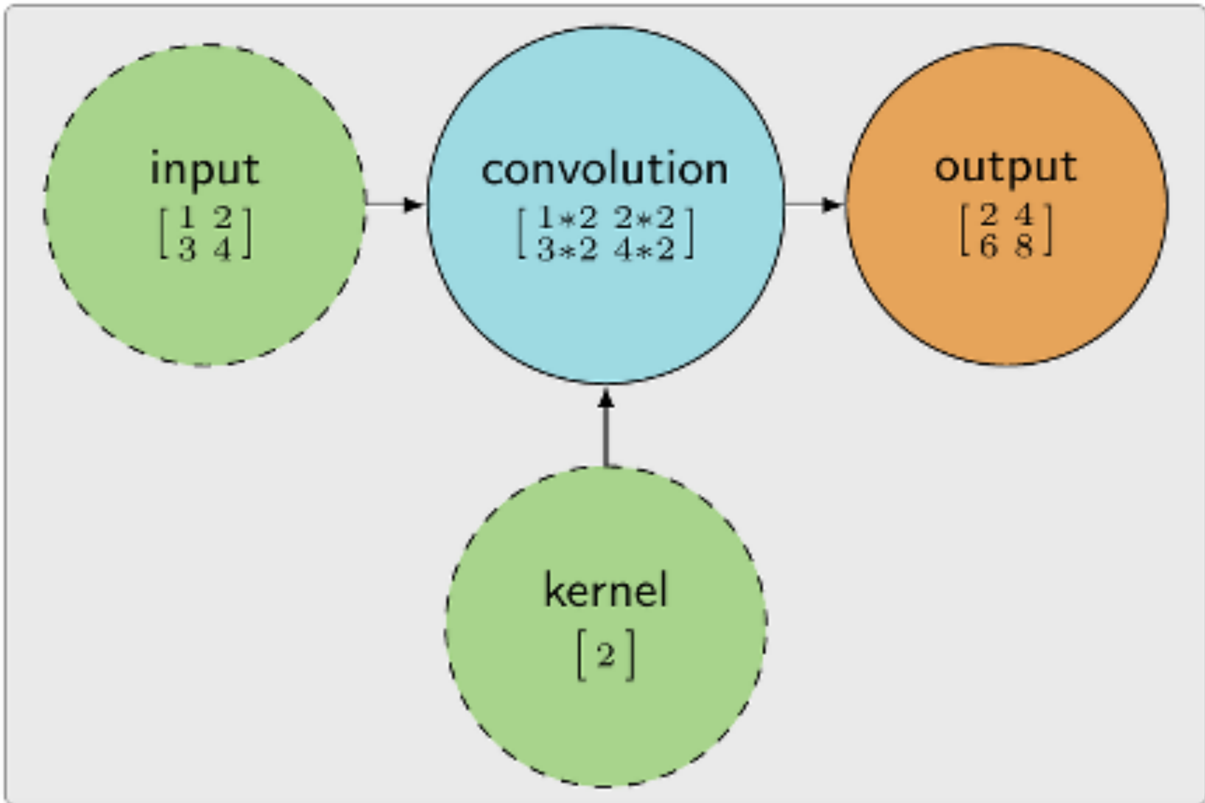
```
array([ 0, 255, 0], dtype=int32)
```

- Instead of loading images from disk, the image_batch variable will act as if it were images loaded as part of an input pipeline

Convolution

- Convolution operations are an important component of convolutional neural networks
- The ability for a CNN to accurately match diverse patterns can be attributed to using convolution operations
- These operations require complex input, which was shown in the previous slides

Convolution



Input and Kernel

- Convolution operations in **TensorFlow** are done using tf.nn.conv2d in a typical situation

- There are other convolution operations available using **TensorFlow** designed with special use cases.

- tf.nn.conv2d is the preferred convolution operation to begin experimenting with

- For example, we can experiment with convolving two tensors together and inspect the result