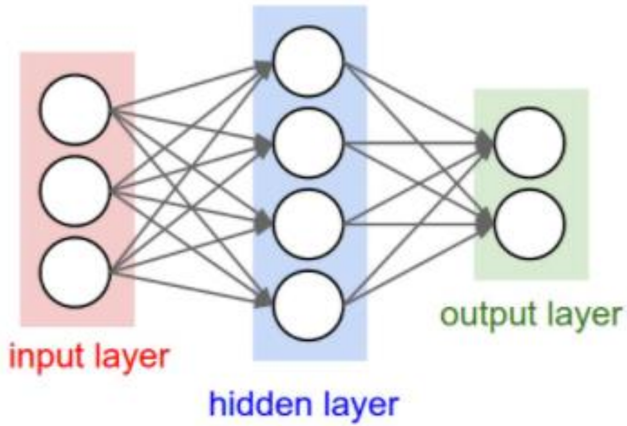
Introduction to Deep Learning (I2DL)

Exercise 5: Neural Networks

Today's Outline

- Universal Approximation Theorem

- Exercise 5
  - More numpy but structured



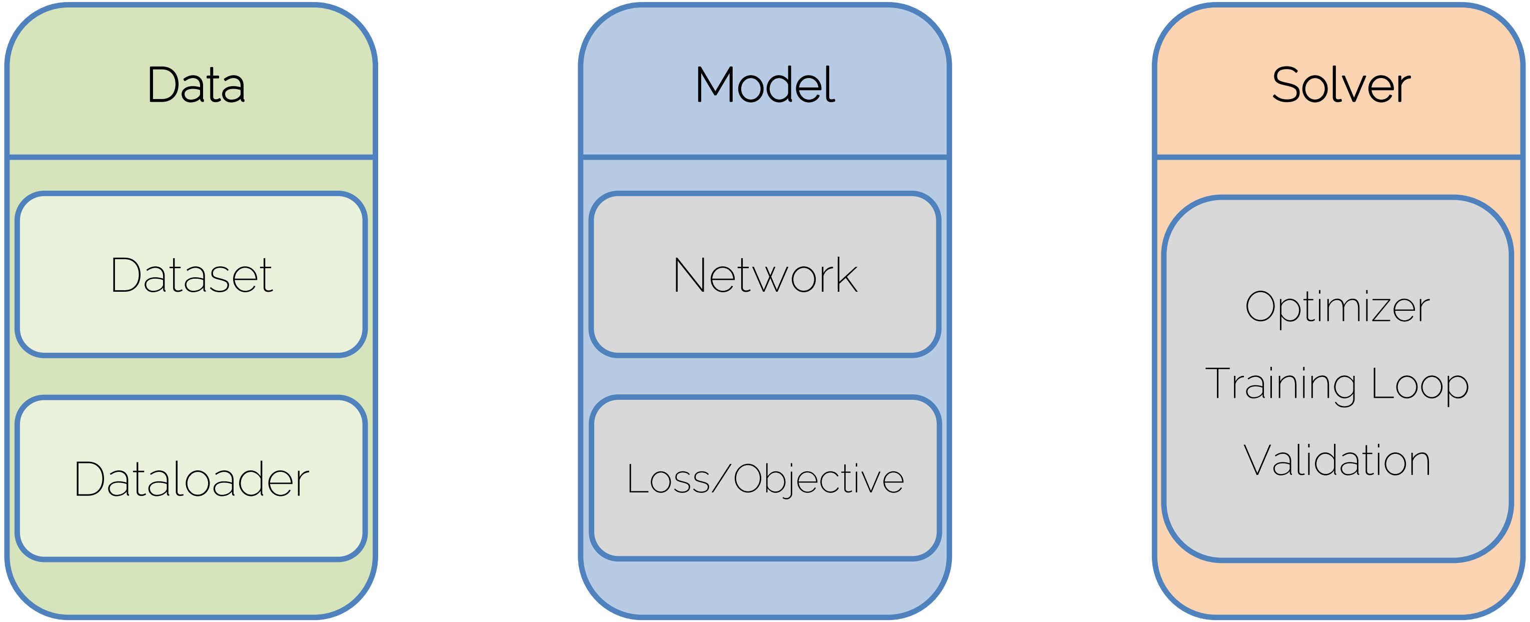
Some background info

- You are currently in the numpy heavy part
  After exercise 5 there will be less numpy implementations

- Creating exercises is hard
  We will take your feedback to heart but we can't implement everything this semester with our current resources
  Feedback is still welcome and important!

Recap: Exercise 4

- The Pillars of Deep Learning

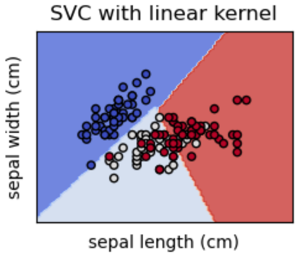


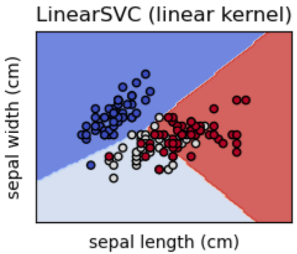
Recap: Exercise 4

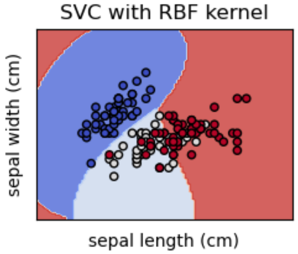
Back to the roots!

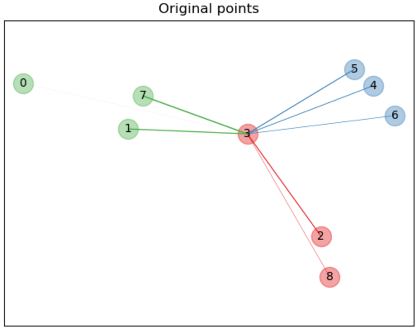
Common machine learning approaches:

- SVM
- Nearest Neighbors









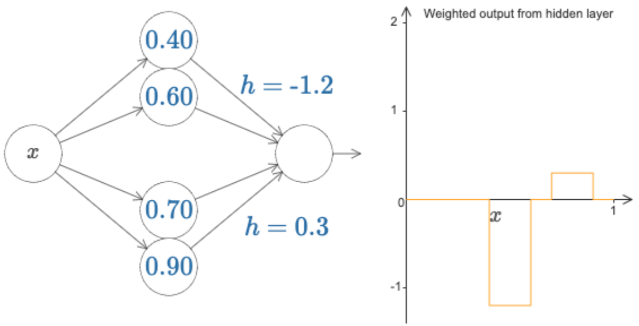
Img src: scikit-learn.org, knowyourmeme "we don't do that here"

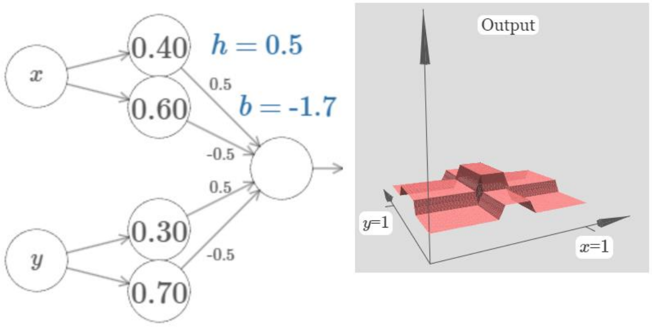
Universal Approximation Theorem

Universal Approximation Theorem

Theorem (1989, colloquial)

For any continuous function f on a compact set K, there exists a one layer neural network, having only a single hidden layer + sigmoid, which uniformly approximates f to within an arbitrary $\varepsilon > 0$ on K.





Universal Approximation Theorem

Readable proof:

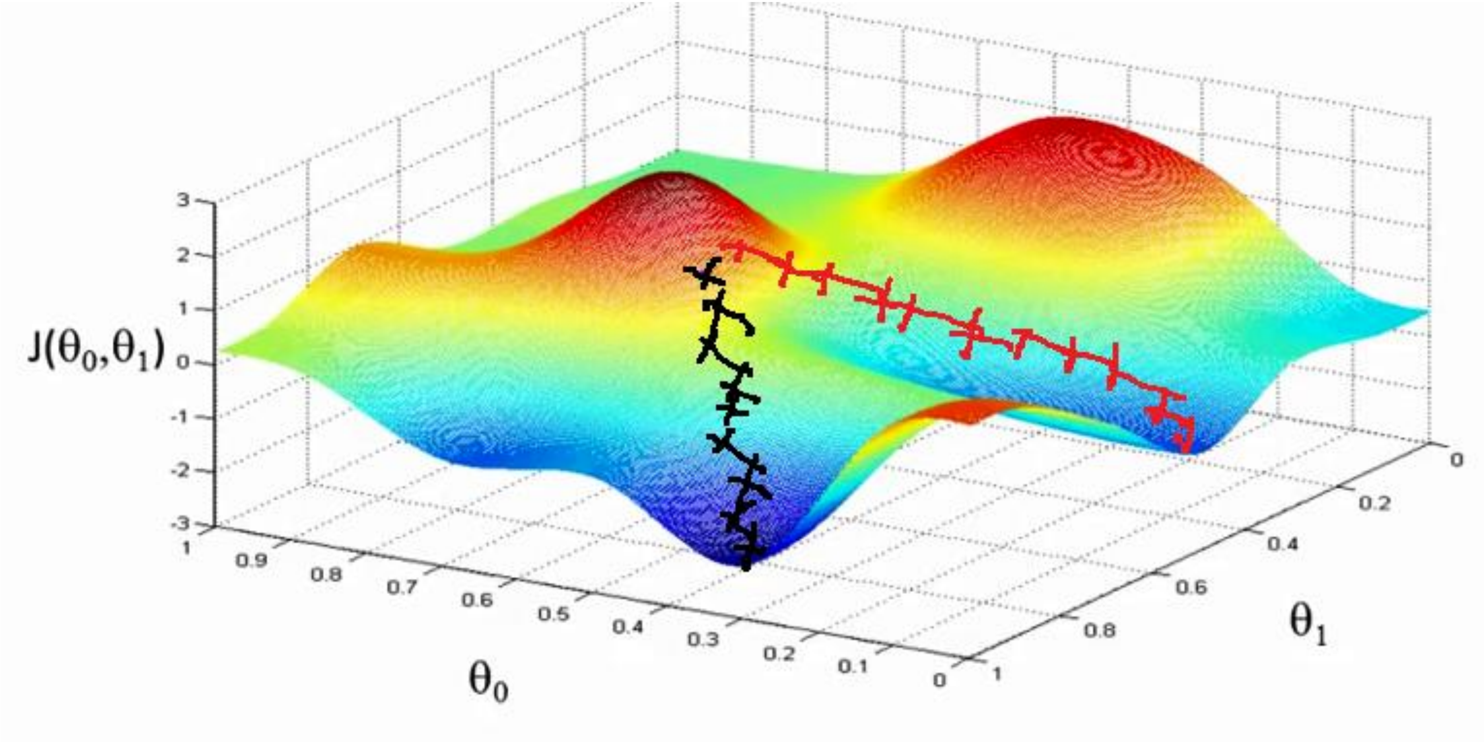
https://mcneela.github.io/machine_learning/2017/03/21/Universal-Approximation-Theorem.html

(Background: Functional Analysis, Math Major 3rd semester)

Visual proof:

http://neuralnetworksanddeeplearning.com/chap4.html

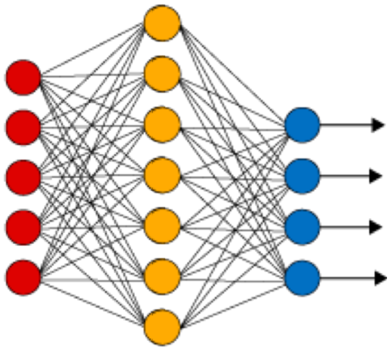
A word of warning



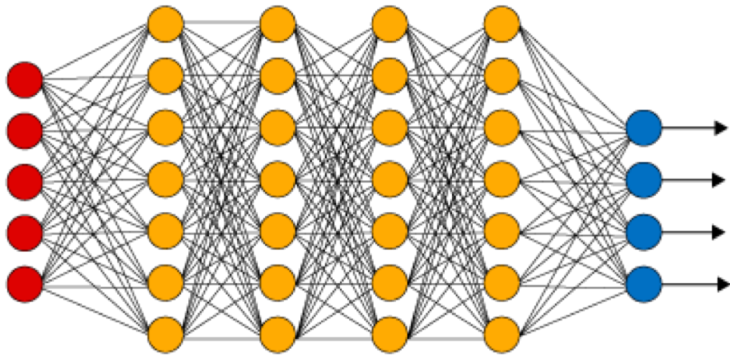
Source: http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

How deep is your love

- Shallow
  (1 hidden layer)



- Deep
  (>1 hidden layer)



Obvious Questions

- Q: Do we even need deep networks?
  A: Yes. Multiple layers allow for more abstraction power given a fixed computational budget in comparison to a single layer → better at generalization

- Q: So we just build 100 layer deep networks?
  A: Not trivially ;-)
  - Constraints: Memory, vanishing gradients, …
  - deeper != working better

Exercise 5

Recap: Exercise 4

Ex4:

- Small dataset
  And simple objective

- Simple classifier
  Single weight matrix

- Gradient descent solver
  Whole forward pass in memory

Ex5:

- CIFAR10
  Actual competitive task

- Modularized Network
  Chain rule rules

- Stochastic Descent

Recap: Exercise 4

```python
class Classifier(Network):
    """
    Classifier of the form y = sigmoid(X * W)
    """

    def __init__(self, num_features=2):
        super(Classifier, self).__init__("classifier")

        self.num_features = num_features
        self.W = None

    def initialize_weights(self, weights=None):
        """
        Initialize the weight matrix W

        :param weights: optional weights for initialization
        """
        if weights is not None:
            assert weights.shape == (self.num_features + 1, 1), \
                "weights for initialization are not in the correct shape
            self.W = weights
        else:
            self.W = 0.001 * np.random.randn(self.num_features + 1, 1)
```

```python
def forward(self, X):
    """
    Performs the forward pass of the model.

    :param X: N x D array of training data. Each row is a D-dimensional point.
    :return: Predicted labels for the data in X, shape N x 1
             1-dimensional array of length N with classification scores.
    """
    assert self.W is not None, "weight matrix W is not initialized"
    # add a column of 1s to the data for the bias term
    batch_size, _ = X.shape
    X = np.concatenate((X, np.ones((batch_size, 1))), axis=1)
    # save the samples for the backward pass
    self.cache = X
    # output variable
    y = None

    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass and return the output of the model. Note  #
    # that you need to implement the function self.sigmoid() for that      #
    ########################################################################

    y = X.dot(self.W)
    y = self.sigmoid(y)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
```

Doesn't scale

New: Modularization

Chain Rule:

$$\boxed{\frac{\partial f}{\partial y} = \frac{\partial f}{\partial d} \cdot \frac{\partial d}{\partial y}}$$

```
class Sigmoid:
    def __init__(self):
        pass

    def forward(self, x):
        """
        :param x: Inputs, of any shape

        :return out: Output, of the same shape as x
        :return cache: Cache, for backward computation, of the same shape as x
        """

    def backward(self, dout, cache):
        """
        :return: dx: the gradient w.r.t. input X, of the same shape as X
        """
```

Overview Exercise 5

- One notebook
  - But a long one…

deadline
Wednesday 15:59

- Multiple smaller implementation objectives

Definition

$$CE(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{C} \Big[-y_{ik} \log(\hat{y}_{ik}) \Big]$$

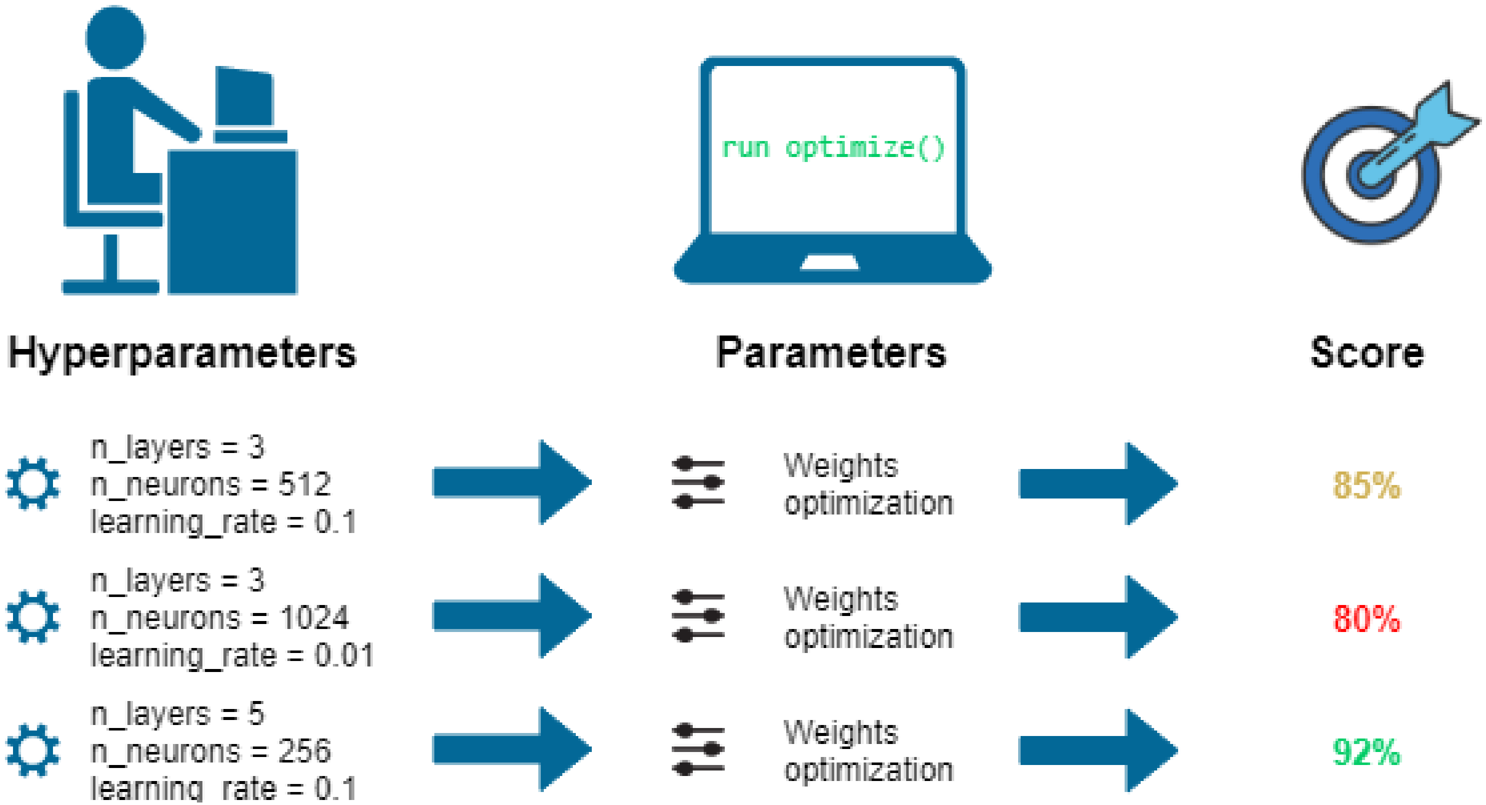
where:

- N is again the number of samples
- C is the number of classes
- $\hat{y}_{ik}$ is the probability that the model assigns for the k'th class when the i'th sample is the input.
- $y_{ik} = 1$ iff the true label of the ith sample is k and 0 otherwise. This is called a one-hot encoding.

Task: Check Formula

Check for yourself that when the number of classes C is 2, then binary cross-entropy is actually equivalent to cross-entropy.

Outlook Ex6: CIFAR10 again



See you next week ☺