NEURAL NETWORKS FOR CLASSIFICATION

LECTURE 7
SECTION 1
JUNE 12TH

INSTITUTE FOR APPLIED
COMPUTATIONAL SCIENCE
AT HARVARD UNIVERSITY

UNIVERSITY *of*
RWANDA

PROBABILISTIC MODELS FOR CLASSIFICATION

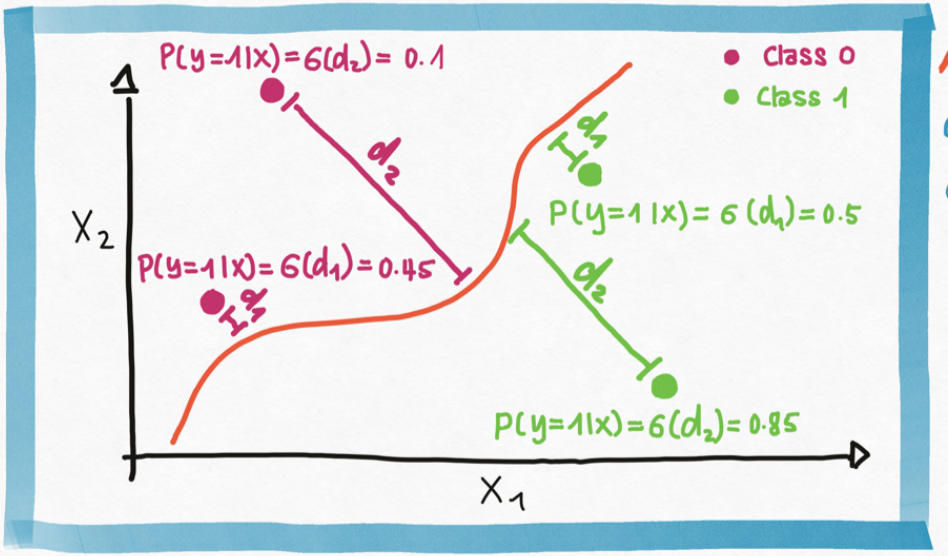
PROBABILISTIC MODELS FOR CLASSIFICATION:

Recall our probabilistic model for classification, given a decision boundary $f_w(x)=0$, models the probability of the label being 1 given the distance of x to the decision boundary:

$$P(y=1|x,w)=\sigma(f_w(x))$$

The closer x is to the boundary the closer the probability $P(y=1|x,w)$ is to 0.5. The farther x is from the decision boundary the closer $P(y=1|x,w)$ is to 0 or 1.

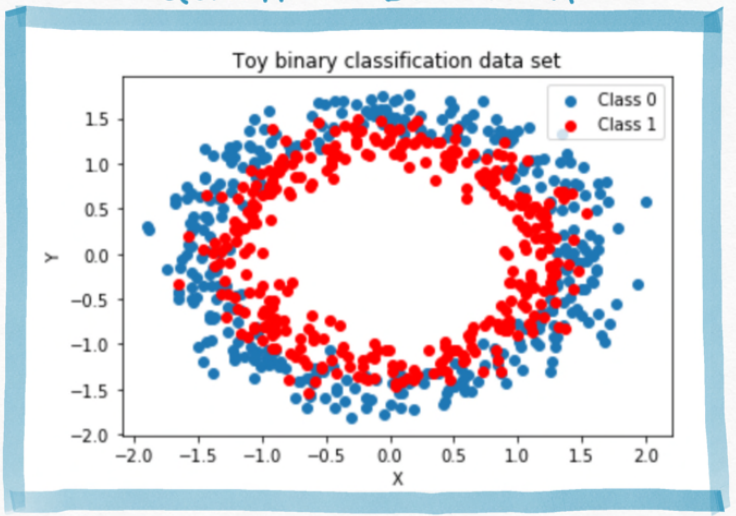
Probabilistic Model For Classification



CHOOSING A FUNCTION FOR THE DECISION BOUNDARY:

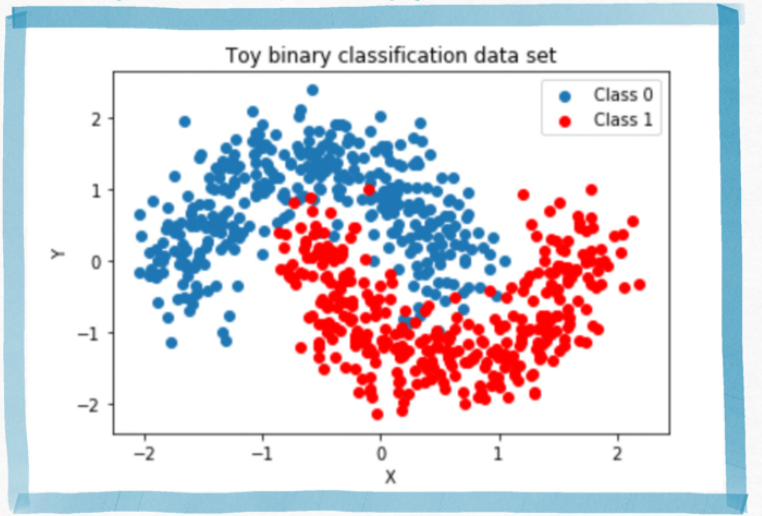
When we can visualize the dataset then it's easy to guess which degree polynomial is appropriate for the dataset. But if we can't see the data, how can we tell which non-linear function f_W describes the decision boundary?

QUADRATIC BOUNDARY



$$f_W(X) = W_1 X_1^2 + W_2 X_2^2 + W_3 X_1 X_2 + W_3$$

GENERAL NON-LINER BOUNDARY



$$f_W(X) = ?$$

When need a general non-linear function we should use a neural network!

NEURAL NETWORK CLASSIFIERS

NEURAL NETWORK FOR CLASSIFICATION:

Model

Given training data $D = \{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$, where $y^{(n)} = 0$ or 1, we assume the following probabilistic model:

$$y^{(n)} \sim \text{Bernoulli}\left(\sigma(f_w(x^{(n)}))\right)$$

where σ is the sigmoid function, f_w is a neural network with unknown parameters w. In other words: $p(y^{(n)} = 1 \mid x^{(n)}, w) = \sigma(f_w(x^{(n)}))$

The likelihood of the training set D is given by:

$$\mathcal{L}(w) = \prod_{n=1}^{N} p(y^{(n)} \mid x^{(n)}, w) = \prod_{n=1}^{N} \underbrace{\sigma(f_w(x^{(n)}))^{y^{(n)}} (1 - \sigma(f_w(x^{(n)})))^{(1-y^{(n)})}}_{\text{Bernoulli}(y^{(n)};\, \sigma(f_w(x^{(n)})))}$$

Inference

Fitting the model means finding the maximum likelihood estimator of w:

$$w_{MLE} = \arg\max_w \mathcal{L}(w)$$

This is equivalent to minimizing the negative log-likelihood:

$$w_{MLE} = \arg\min_w -\log \mathcal{L}(w) = \arg\min_w -\ell(w)$$

We minimize $-\ell(w)$ by gradient descent.

The loss function for neural network classification is non-convex!

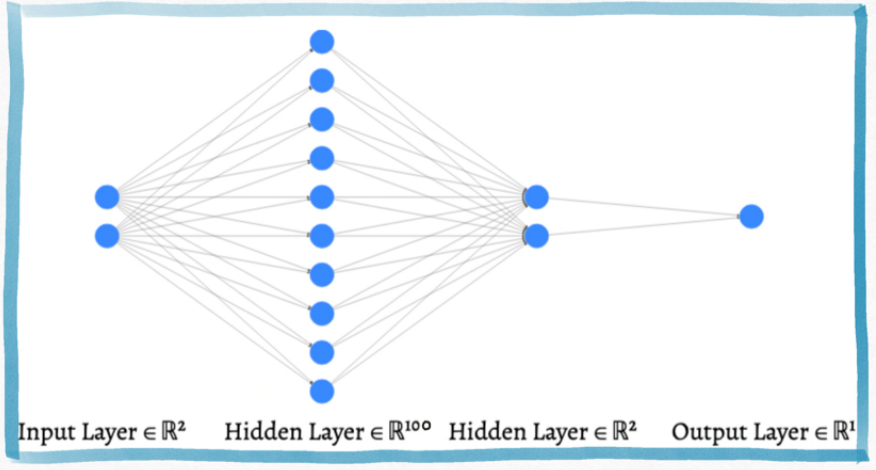
NEURAL NETWORK CLASSIFIERS IN KERAS

NEURAL NETWORK CLASSIFIERS IN keras:

```
# create sequential multi-layer perceptron in keras
model = Sequential()
#layer 0 to 1: input to 100 hidden nodes
model.add(Dense(100, input_dim=input_dim, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
#layer 1 to 2: 100 hidden nodes to 2 hidden nodes
model.add(Dense(2, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
#binary classification, one output: 2 hidden nodes to one output node
#since we're doing classification, we apply the sigmoid activation to the
#linear combination of the input coming from the previous hidden layer
model.add(Dense(1, activation='sigmoid',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

# configure the model
model.compile(optimizer=optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

- Add a hidden layer with 100 nodes connected the input nodes
- Add a second hidden layer with 2 nodes connected to the first hidden layer.
- Add one output node and apply the sigmoid function to convert the output into a probability
- use relu activation functions
- use neg-loglikelihood (i.e. crossentropy) as the loss function.
- in addition, compute the accuracy



- Two input dimensions
- Two hidden layers:
  - ↳ one with 100 nodes
  - ↳ one with 2 nodes
- One output node
- This represents the composition of three functions:

$$\mathbb{R}^2 \longrightarrow \mathbb{R}^{100} \longrightarrow \mathbb{R}^2 \longrightarrow \mathbb{R}$$

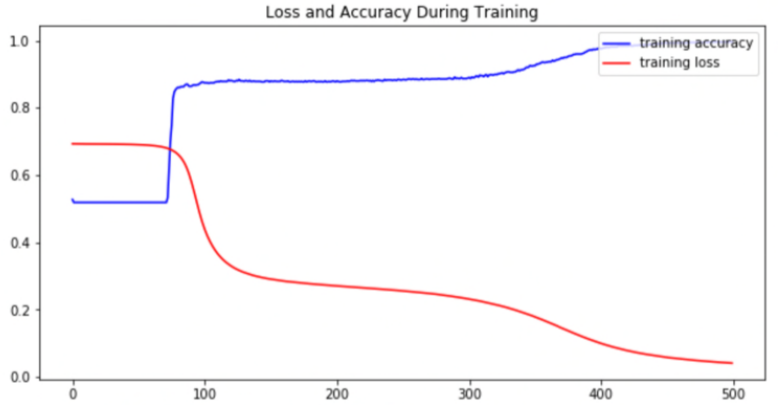
TRAINING NEURAL NETWORK CLASSIFIERS:

Fitting the model using stochastic Gradient Descent with batches of 2

```
# fit the model to the data and save value of the objective function at each step of training
history = model.fit(X_train, Y_train, batch_size=20, shuffle=True, epochs=500, verbose=0)
```

Visualizing both loss function and accuracy trace plot

In [87]:
```
# plot the loss function and the evaluation metric over the course of training
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
ax.plot(np.array(history.history['accuracy']), color='blue', label='training accuracy')
ax.plot(np.array(history.history['loss']), color='red', label='training loss')
ax.set_title('Loss and Accuracy During Training')
ax.legend(loc='upper right')
plt.show()
```



- loss is decreasing during training and stabilizes
- accuracy is increasing during training and stabilizes