Introduction to Machine Learning

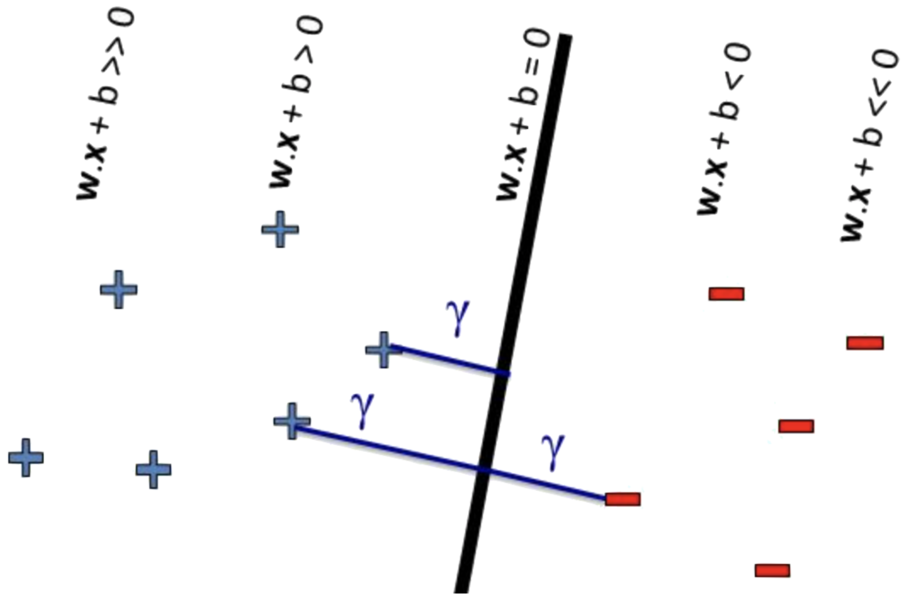
Neural Networks

July 3rd, 2019

Instructors: Melanie Fernandez Pradier (Havard), Weiwei Pan (Harvard), Javier Zazo Ruiz (Harvard)

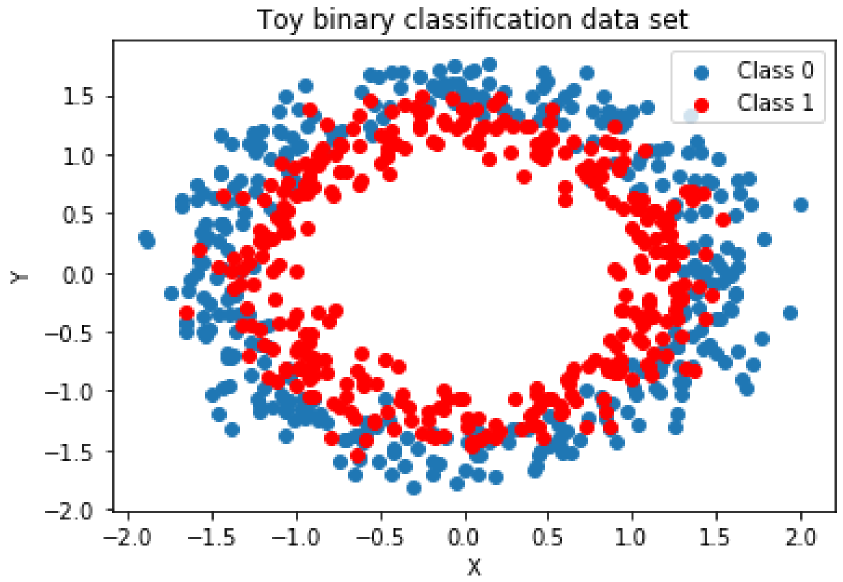
Review of the Geometry of Logistic Regression

In **logistic regression**, we model the probability of an input $\mathbf{x}$ being labeled '1' as a function of its distance from the hyperplane parametrized by $\mathbf{w}$



That is, we model $p(y = 1|\mathbf{w}, \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$. Where $\mathbf{w}^\top \mathbf{x} = 0$ is the equation of the decision boundary.

How would you parametrize a ellipitical decision boundary?

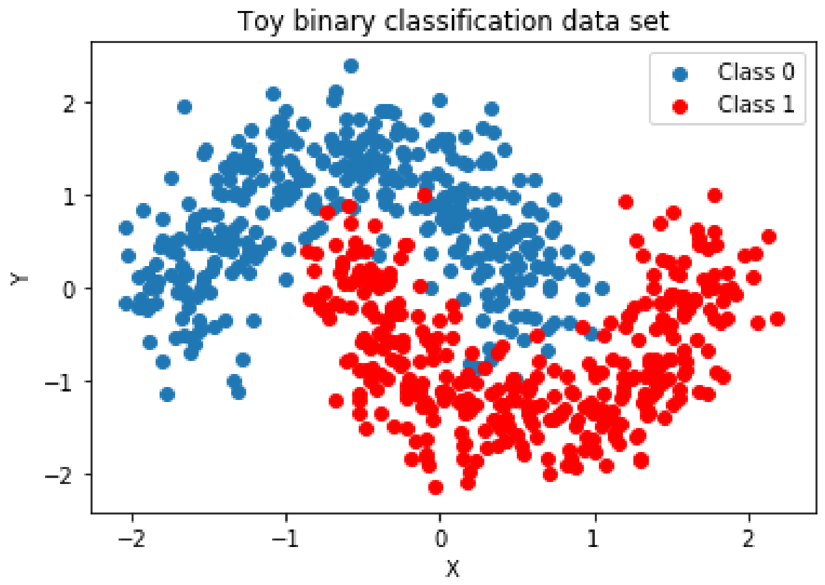


We can say that the decision boundary is given by a ***quadratic function*** of the input:

$$w_1 x_1^2 + w_2 x_2^2 + w_3 = 0$$

We say that we can fit such a decision boundary using logistic regression with degree 2 polynomial features

How would you parametrize an arbitrary complex decision boundary?



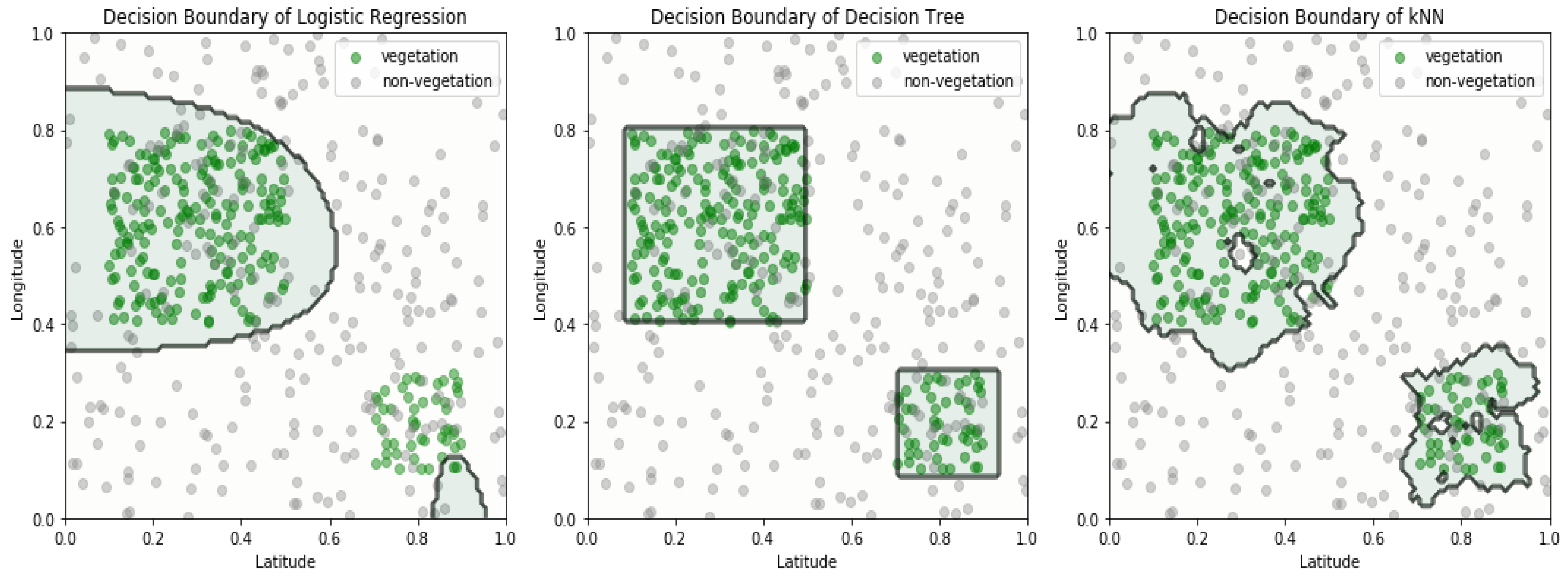
It's not easy to think of a function $g(x)$ can capture this decision boundary.

GOAL: Find models that can capture *arbitrarily complex* decision boundaries.

The Decision Boundaries of Different Classifiers

Last time, we proposed three types of models that can capture arbitrarily complex decision boundaries:

1. decision tree
2. random forest
3. kNN classifier



An Embarassment of Riches

Now we have seen four types of classifiers: logistic regression, decision tree, random forest, kNN. Each type can be customized in many ways (e.g. we can choose many different polynomials for the logistic regression boundary)

Question: which model should we use?

Answer: your choice of model should depend on the task and the dataset. You must

1. choose models based on sensible evaluation metrics
2. choose models using proper data set splitting procedure (train/validation/test)
3. choose models that best solves your real-life task!

Exercise: Compare All Classification Models on a Real Task

Application: Automated cancer diagnosis - classify biopsy results as cancerous or non-cancerous.

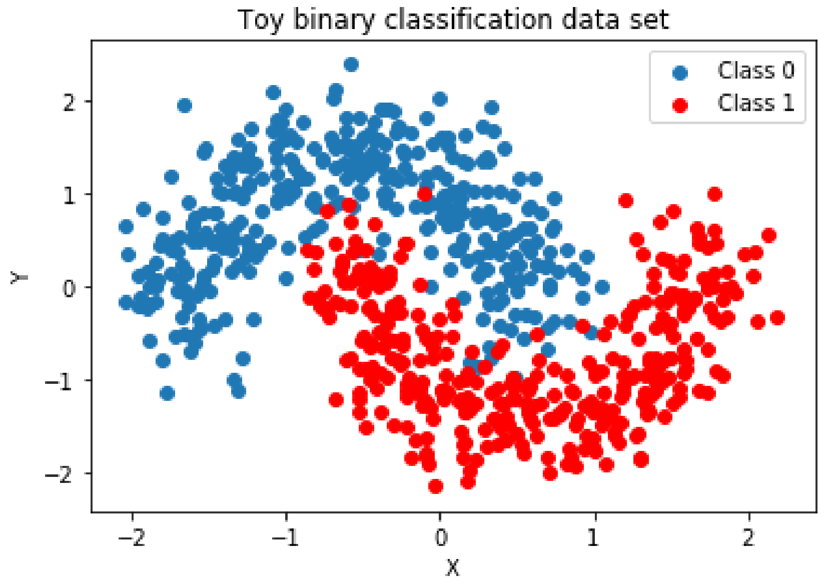
Comparison of Classifiers: Computational Concerns

In addition to considering how well a model solves our real-life task, we also need to consider how easily we can apply a model to large amounts of data - this is called ***scalability***.

Specifically, we want to know how easily we can train a model and how easily we can compute a prediction.

QUESTION: What are the advantages and disadvantages of each type of classifier with respect to scalability?

How would you parametrize an arbitrary complex decision boundary?



It's not easy to think of a function $g(x)$ can capture this decision boundary.

REVISED GOAL: Find models that can capture *arbitrarily complex* decision boundaries **and** are fast to train as well as effecient for computing predictions.

Neural Networks

Approximating Arbitrarily Complex Decision Boundaries

Given an exact parametrization, we could learn the functional form, g, of the decision boundary directly.

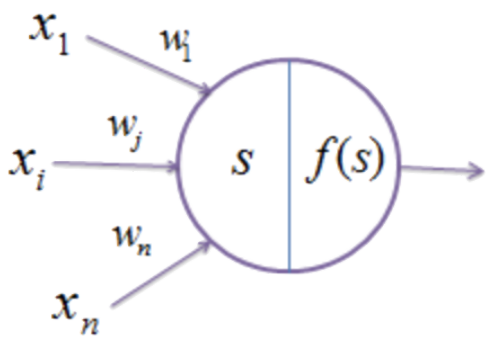
However, assuming an exact form for g is restrictive.

Rather, we can build increasingly good approximations, $\widehat{g}$, of g by composing simple functions.

What is a Neural Network?

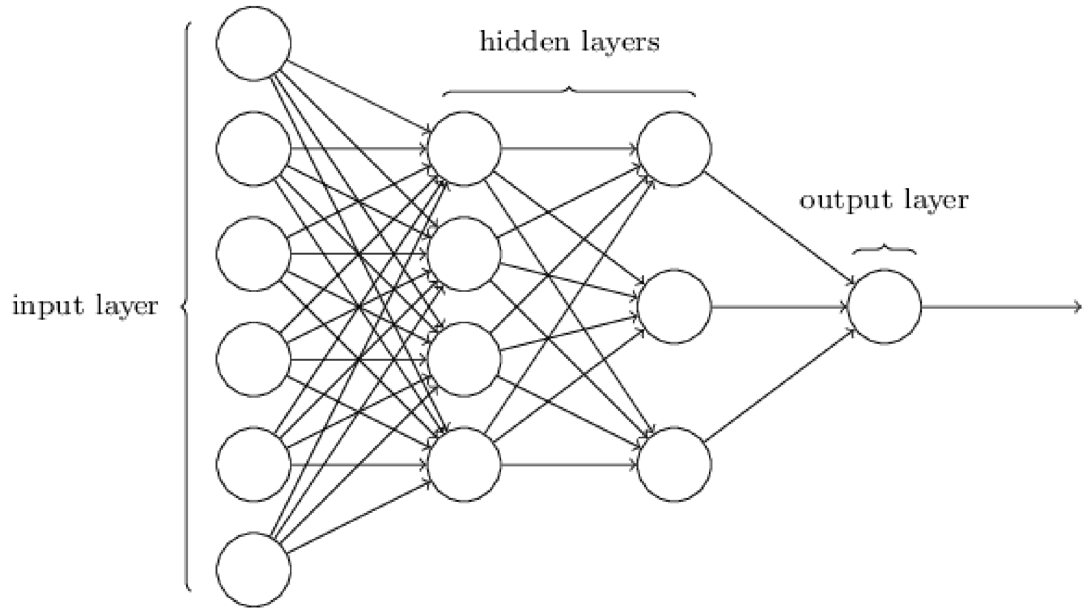
Goal: build a good approximation $\hat{g}$ of a complex function g by composing simple functions.

For example, let the following picture represents $f\left(\sum_i w_i x_i\right)$, where f is a non-linear transform:



Neural Networks as Function Approximators

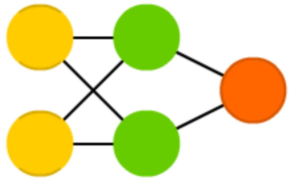
Then we can define the approximation $\widehat{g}$ with a graphical schema representing a complex series of compositions and sums of the form, $f\left(\sum_i w_i x_i\right)$



This is a ***neural network***. We denote the weights of the neural network collectively by $\mathbf{W}$. The non-linear function f is called the ***activation function***.

Exercise: Translate Graphical Representations into Functional Ones

Translate the following graphical representation of a neural network into a functional form, $g(x_1, x_2) = ?$

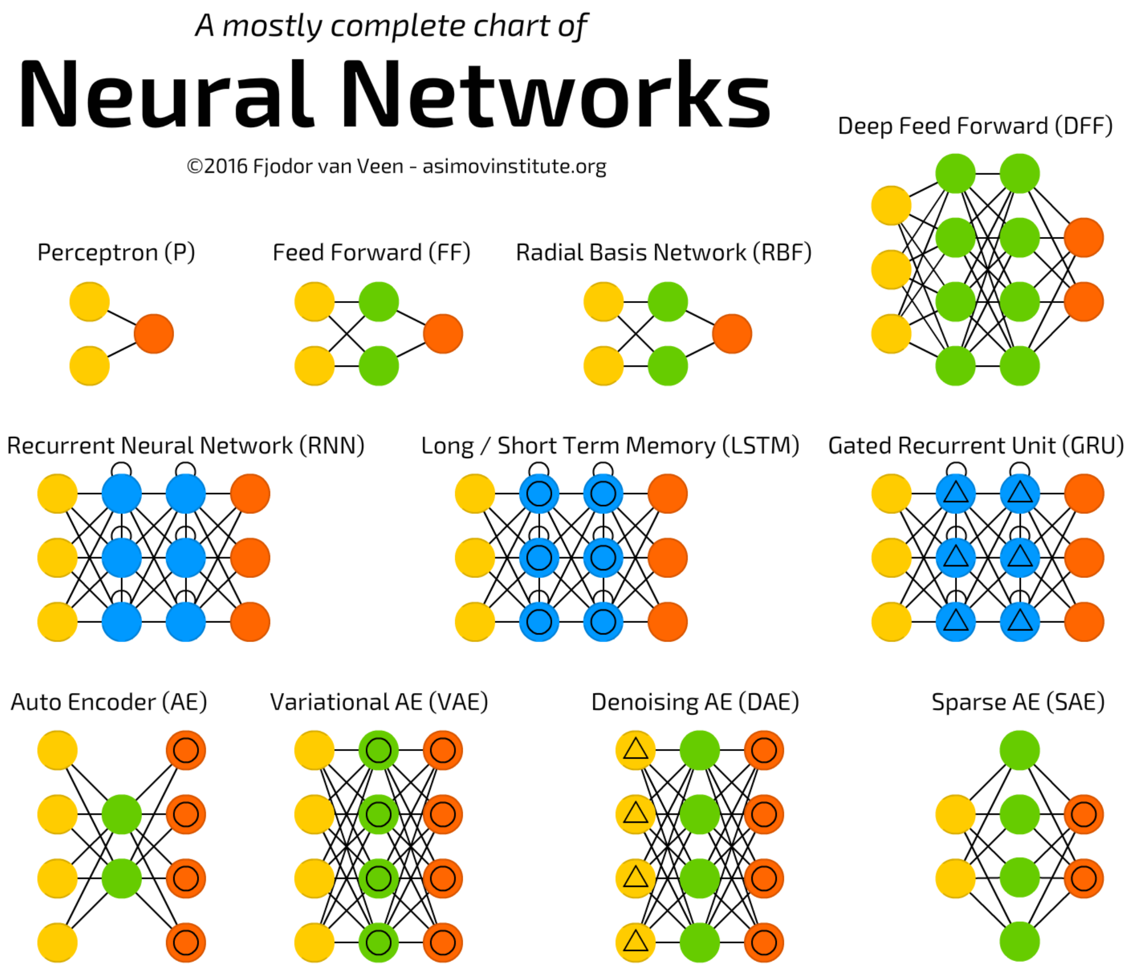


Use the following labels:

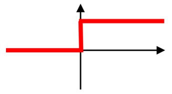
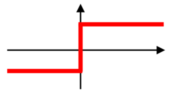
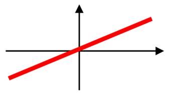
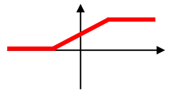
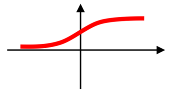
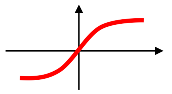
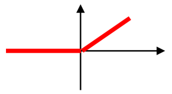
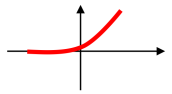
1. the input nodes x_1 and x_2
2. the hidden nodes h_1 and h_2
3. the output node y
4. the weight from x_i to h_j as w_j^i
5. the weight from h_j to y as w^j
6. the activation function $g(x) = e^{-0.5x^2}$

Bonus: write the functional form in vector notation.

A Flexible Framework for Function Approximation


A mostly complete chart of
Neural Networks
©2016 Fjodor van Veen - asimovinstitute.org
Deep Feed Forward (DFF)
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)

Common Choices for the Activation Function

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \dfrac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^z)$	Multi-layer Neural Networks	

Using Neural Networks for Regression

Neural Networks Regression

Data: features `x_train`, real-valued labels `y_train`

Probabilistic Model: `y_train` $= g_{\mathbf{W}}($ `x_train` $)+\epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$, where $g_{\mathbf{W}}$ is a neural network with parameters $\mathbf{W}$.

Training Objective: find $\mathbf{W}$ to maximize the likelihood of our data. This is equivalent to minimizing the Mean Square Error,

$$\max_{\mathbf{W}} \; \mathrm{MSE}(\mathbf{W}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - g_{\mathbf{W}}(x_n))^2$$

Optimizing the Training Objective: For linear regression (when $g_{\mathbf{W}}$ is a linear function), we computed the gradient of the MSE with respective to the model parameters $\mathbf{W}$, set it equal to zero and solved for the optimal $\mathbf{W}$ analytically.

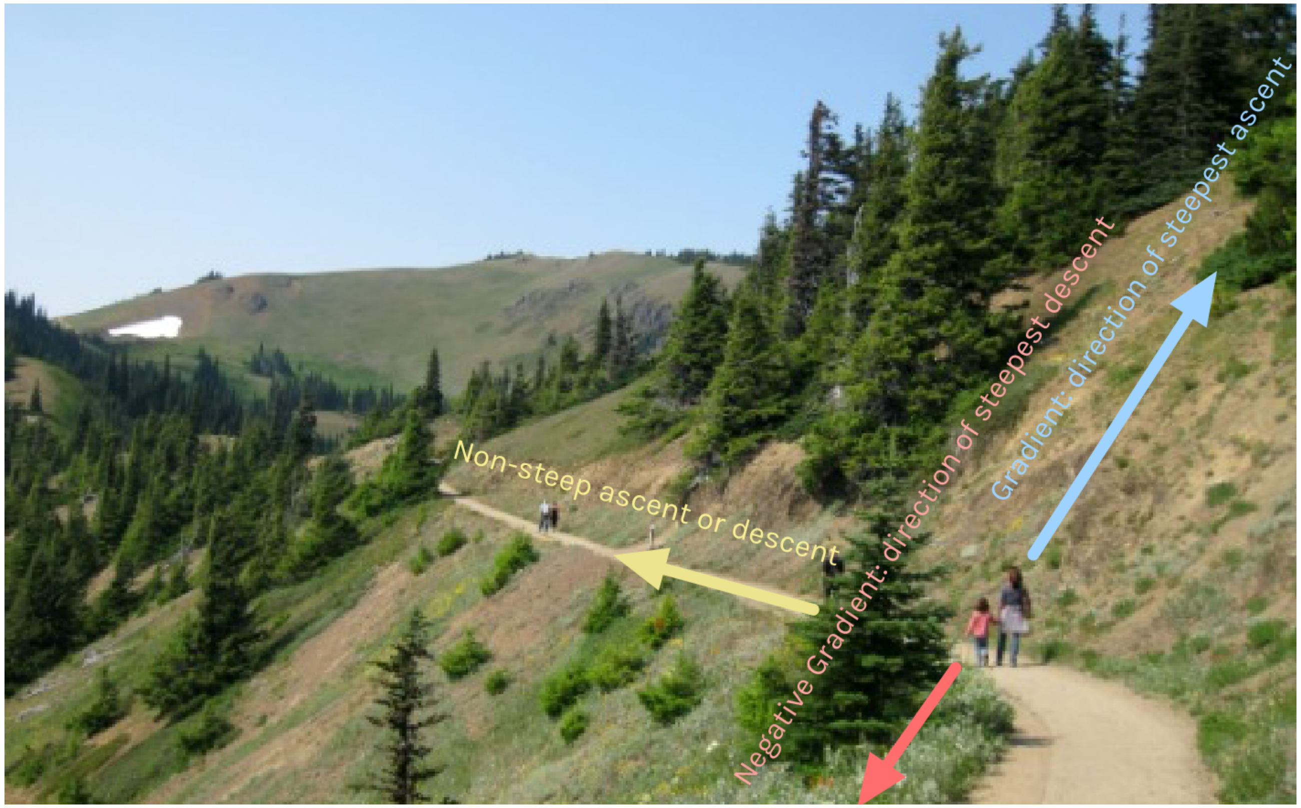
Can we do the same when $g_{\mathbf{W}}$ is a neural network?

Exercise: Optimizing Neural Networks

For the small neural network in the previous exercise, compute the gradient $\nabla_{\mathbf{W}} \operatorname{MSE}(\mathbf{W})$. Can you analytically solve for the optimal parameters $\mathbf{W}$? Is the training objective convex?

Gradient Descent for Training Neural Networks: BackProp

The intuition behind various flavours of gradient descent is as follows:


Non-steep ascent or descent
Negative Gradient: direction of steepest descent
Gradient: direction of steepest ascent

Gradient Descent: the Algorithm

1. start at random place: $W_0 \leftarrow \textbf{random}$
2. until (stopping condition satisfied):

   a. compute gradient: gradient = ∇ loss_function(W_t)

   b. take a step in the negative gradient direction: $W_{t+1} \leftarrow W_t$ - eta * gradient

Here *eta* is called the ***learning rate***.

Implementing Neural Networks in `python`

`keras`: a Python Library for Neural Networks

`keras` is a `python` library that provides intuitive api's for build neural networks quickly.

```python
#keras model for feedforward neural networks
from keras.models import Sequential

#keras model for layers in feedforward networks
from keras.layers import Dense

#keras model for optimizing training objectives
from keras import optimizers
```

Building a Neural Network for Regression in `keras`

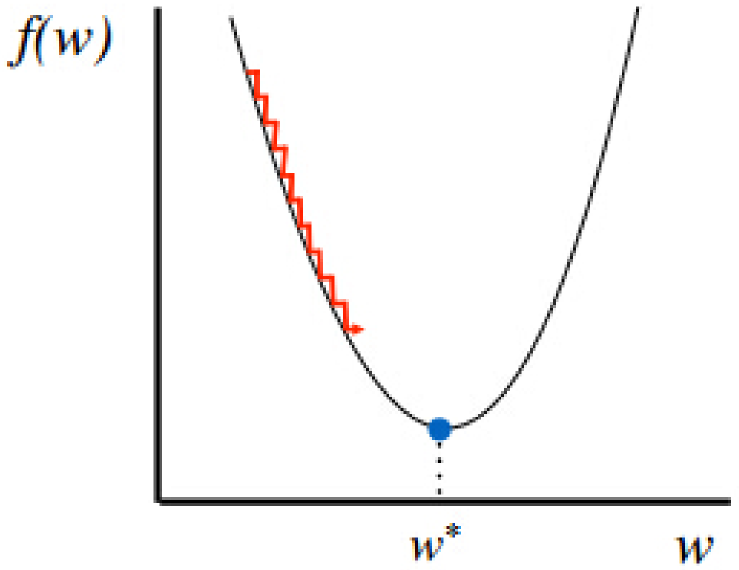
`keras` is a `python` library that provides intuitive api's for build neural networks quickly.

```
#instantiate a feedforward model
model = Sequential()

#add layers sequentially

#input layer: 2 input dimensions
model.add(Dense(2, input_dim=2, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
#hidden layer: 2 nodes
model.add(Dense(2, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

#output layer: 1 output dimension
model.add(Dense(1, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

#configure the model: specify training objective and training algorithm
adam = optimizers.Adam(lr=0.01)
model.compile(optimizer=adam,
              loss='mean_squared_error')
```

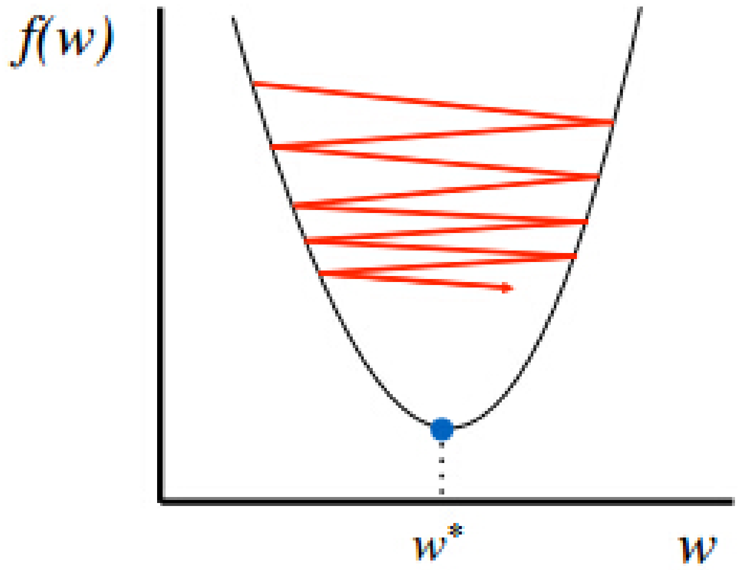
Training a Neural Network in `keras`

```
#fit the model and return the mean squared error during training
history = model.fit(X_train, Y_train, batch_size=20, shuffle=True, epochs=100,
 verbose=0)
```

Training Your NN: Optimization Choices Matter



Too small: converge very slowly



Too big: overshoot and even diverge

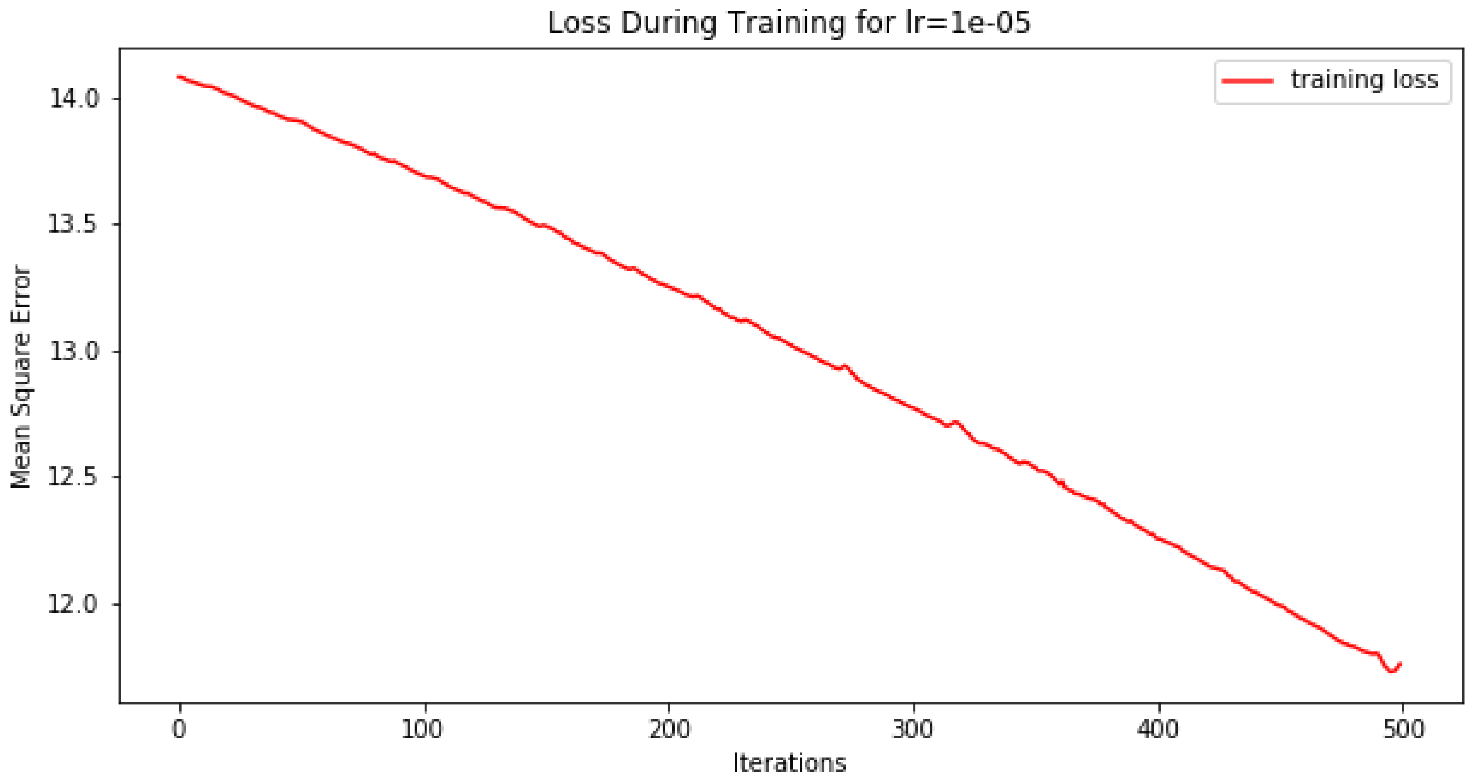
Monitoring Neural Network Training

Visualize the mean square error over the training, this is called the training ***trajectory***.

```
#fit the model and return the mean squared error during training
history = model.fit(X_train, Y_train, batch_size=20, shuffle=True, epochs=100,
 verbose=0)

# 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['mean_squared_error']), color='blue', label=
'training accuracy')

plt.show()
```

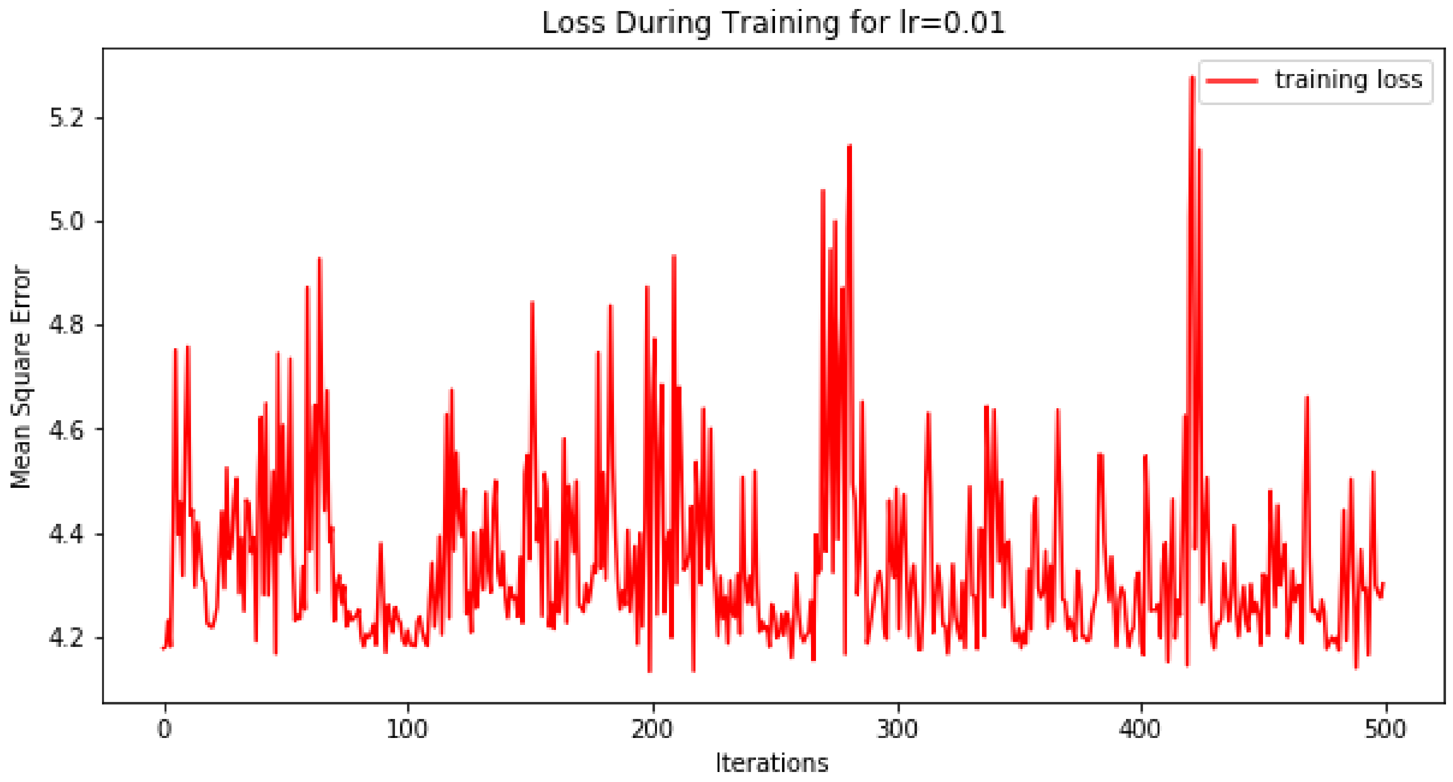
Diagnosing Issues with the Trajectory

If this is your objective function during training, what can you conclude about your step-size?



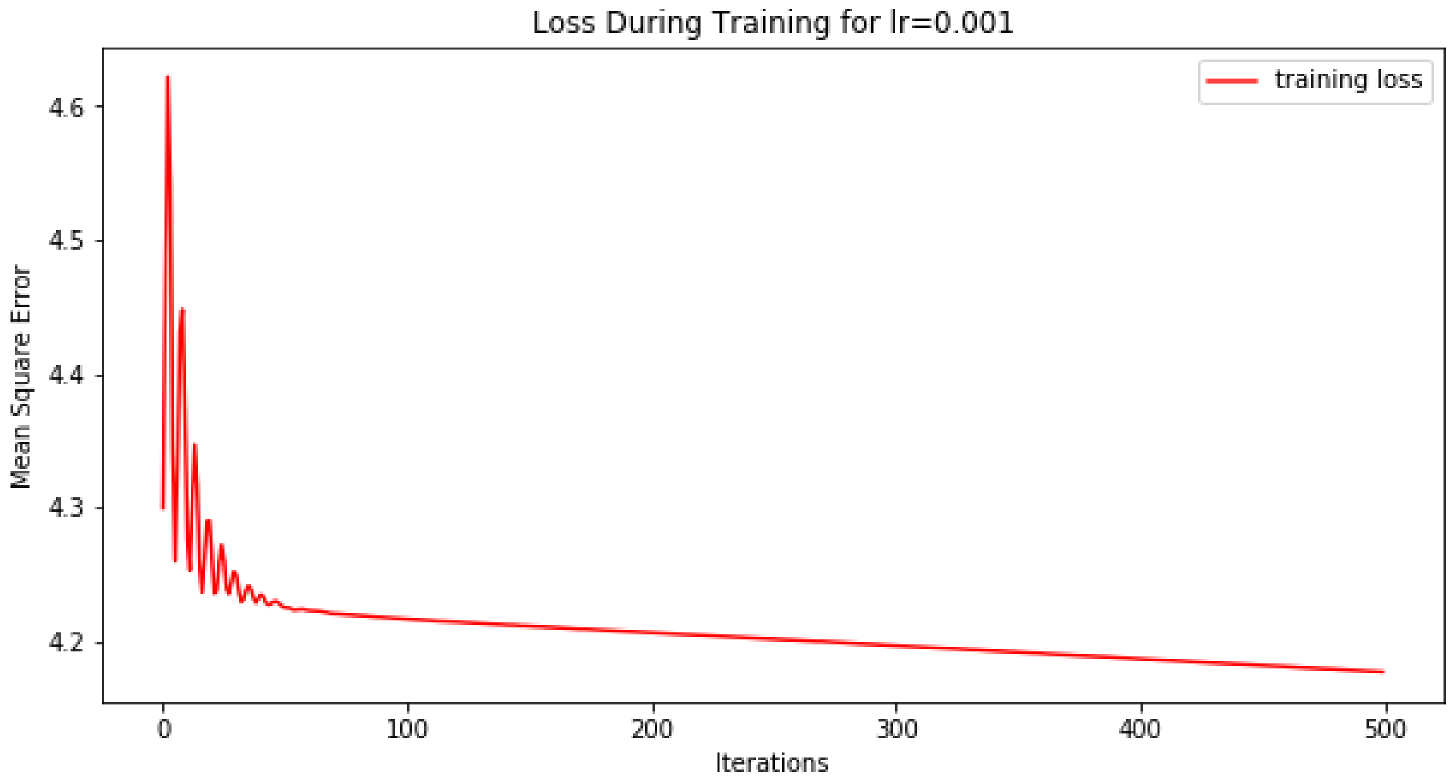
Diagnosing Issues with the Trajectory

If this is your objective function during training, what can you conclude about your step-size?



Diagnosing Issues with the Trajectory

If this is your objective function during training, what can you conclude about your step-size?



Exercise: Compare Neural Network Regression to Polynomial Regression

Compare a neural network regression to a polynomial regression. Which model do you think is "better" and for what kind of tasks?

Using Neural Networks for Classification

Data: features `x_train` , real-valued labels `y_train`

Probabilistic Model: `y_train` $\sim \text{Bernoulli}(\sigma(g_{\mathbf{W}}($ `x_train` $))$, where $g_{\mathbf{W}}$ is a neural network with parameters $\mathbf{W}$, and $\sigma(z) = \frac{1}{1+e^{-z}}$.

Training Objective: find $\mathbf{W}$ to maximize the likelihood of our data. This is equivalent to minimizing the ***binary cross entropy*** or ***log loss***,

$$\max_{\mathbf{W}} \text{ CrossEnt}(\mathbf{W}) = \sum_{n=1}^{N} y_n \log(g_{\mathbf{W}}(x_n)) + (1 - y_n) \log(1 - g_{\mathbf{W}}(x_n))$$

Optimizing the Training Objective: Since this objective is not convex and finding the zero's of the gradient is intractable, we will use gradient descent to find a "optimal" set of parameters $\mathbf{W}$.

Building a Neural Network for Classification in `keras`

```python
#instantiate a feedforward model
model = Sequential()

#add layers sequentially

#input layer: 2 input dimensions
model.add(Dense(2, input_dim=2, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))
#hidden layer: 2 nodes
model.add(Dense(2, activation='relu',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

#output layer: 1 output dimension
model.add(Dense(1, activation='sigmoid',
                kernel_initializer='random_uniform',
                bias_initializer='zeros'))

#configure the model: specify training objective and training algorithm
sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd,
              loss='binary_crossentropy')
```

Exercise: Compare Neural Network Classification to Other Classifiers

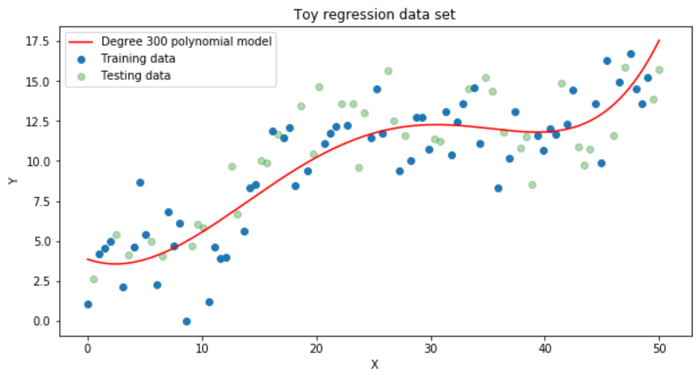
Compare a neural network classifier to other classifiers. For what task is a neural network the most appropriate model?

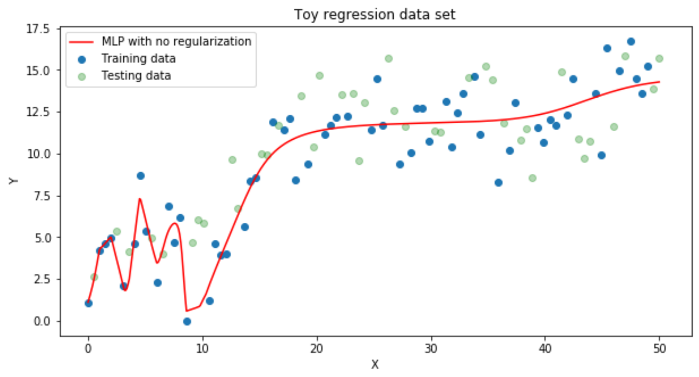
The Bias Variance Trade-off: for Neural Networks

Generalization Error and Bias/Variance

Complex models have ***low bias*** -- they can model a wide range of functions, given enough samples.

But complex models like neural networks can use their 'extra' capacity to explain non-meaningful features of the training data that are unlikely to appear in the test data (i.e. noise). These models have ***high variance*** -- they are very sensitive to small changes in the data distribution, leading to drastic performance decrease from train to test settings.





Regularization

A way to prevent overfitting is to reduce the capacity of the model, thereby limiting the kinds of functions they can model. This **increases bias, but reduces variance**:

1. **ℓ_1, ℓ_2 weight regularization** - adding a term to the loss function that penalizes the ℓ_1-norm (sum of absolute values) or the ℓ_2-norm (sum of squares) of the weights. This prevents the network from learning extremely squiggly functions.

```
from keras import regularizers
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01),
                activity_regularizer=regularizers.l1(0.01)))
```

2. **Dropout** - randomly zeroing out weights during training. This prevents the hidden nodes from "over specializing" or "memorizing" certain data points.

```
from keras.layers import Dropout,
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
```