BST 261: Data Science II

Lecture 5

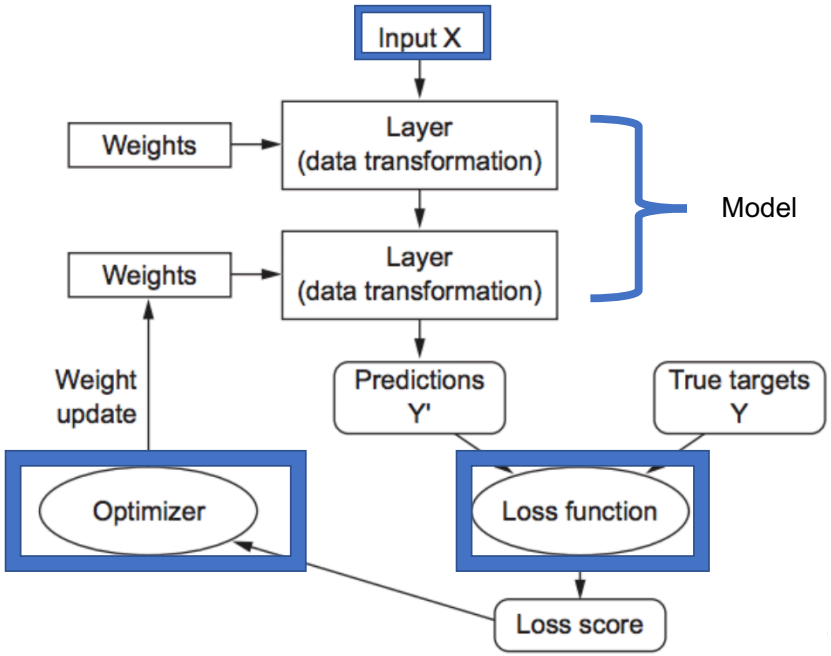
Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

April 2, 2018

Deep Feedforward Networks Review

General Network Architecture



Deep Feedforward Networks in Python

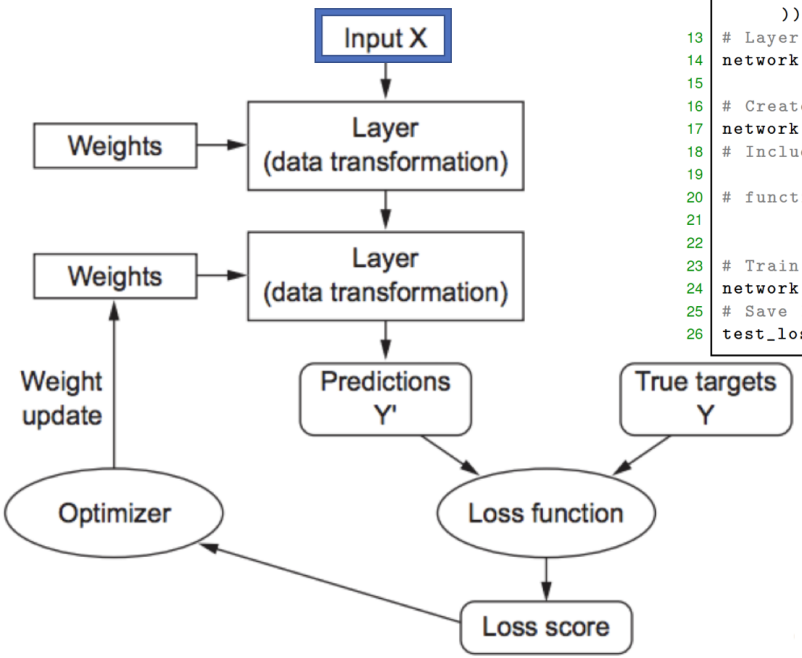
- Here is generic, 2 layer dense deep feedforward network code

```
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n,))
    )
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

Deep Feedforward Networks in Python

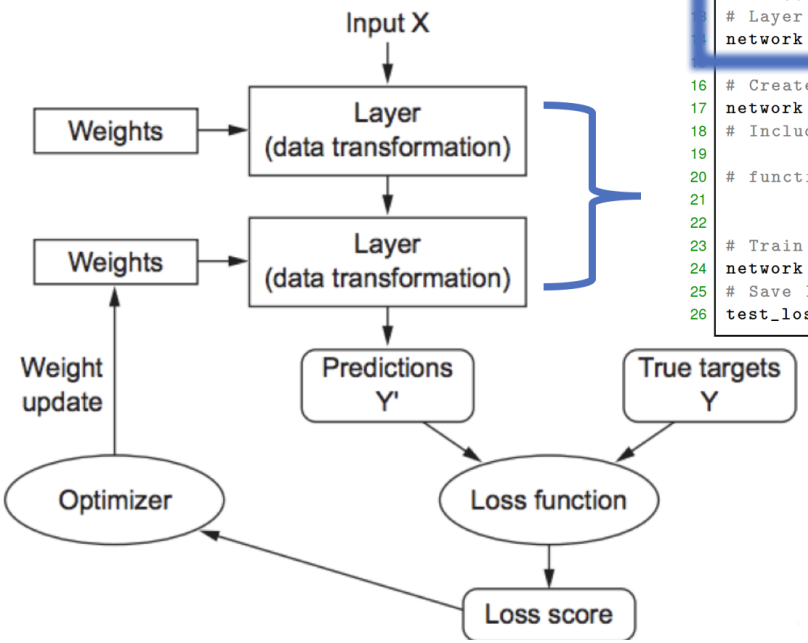
- `train_data`: training examples (matrix of feature vectors $\boldsymbol{x}$ or $\boldsymbol{x}_{train}$)
- `train_labels`: training labels (y or y_{train})
- `test_data`: test examples used to calculate performance of network ($\boldsymbol{x}_{test}$)
- `test_labels`: test labels (y_{test})
- Optimizing algorithm options: `rmsprop`, `sgd`, `adagrad`, `adam`, etc.
- Loss function options: `mse`, `mae`, `categorical_crossentropy`, etc.
- Performance measure options: `accuracy`, `mae`, etc.
- Here:
  - c =the number of hidden units (nodes) in a hidden layer
  - d =the number of units (nodes) in output layer
  - e =the number of epochs (iterations) over entire training data set
  - b =the batch size

Deep Feedforward Networks in Python



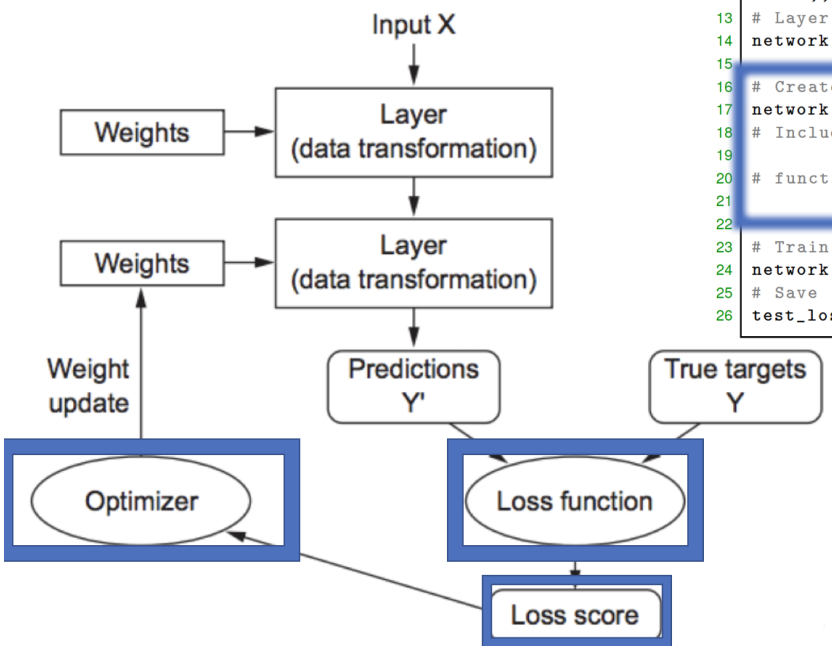

```
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n, )
    ))
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

Deep Feedforward Networks in Python



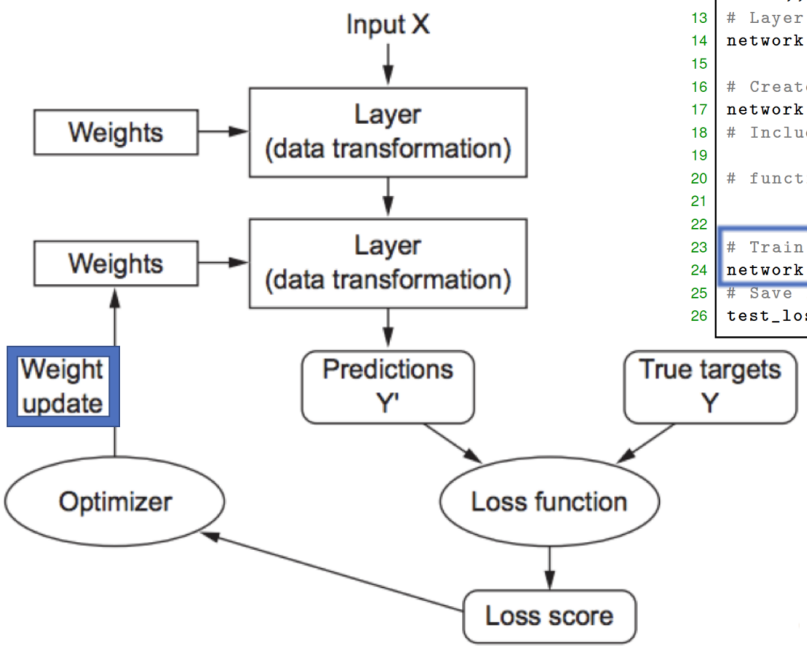

```
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n, )
    ))
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

Deep Feedforward Networks in Python




```python
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n, ))
    ))
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

Deep Feedforward Networks in Python




```
# Import all necessary modules
import numpy as np
import keras
from keras import models
from keras import layers

# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=( n, ))
# Layer 2 (Output layer)
network.add(layers.Dense(d, activation='activation function'))

# Create (compile) the network
network.compile(optimizer='optimizing algorithm',
# Include optimizing function, loss
                loss='loss function',
# function and performance measure
                metrics=['performance measure'])

# Train (learn) the network, specify batch size and number of epochs
network.fit(train_data, train_labels, epochs=e, batch_size=b)
# Save loss and performance measure values
test_loss, test_acc = network.evaluate(test_data, test_labels)
```

MNIST Example

- Training set: 60,000 images of handwritten digits and corresponding labels
  - Each digit is represented as a 28×28 matrix of greyscale values $0 - 255$
  - The entire training set is stored in a 3D tensor of shape (60000, 28, 28)
  - The corresponding image values are stored as a 1D tensor of values $0 - 9$
- Testing set: 10,000 images with the same set up as the training set

- Data Manipulation
  - Reshape each image from a 28×28 matrix of greyscale values $0 - 255$ to a vector of length $28 * 28 = 784$ of values $0 - 1$ (divide each by 255)
  - Reshape each corresponding image label to a vector of length 10 of values 0 or 1

MNIST Example

- Network Architecture
  - 2 layers: 1 hidden and 1 output layer
  - Hidden layer with 512 hidden units and the `relu` activation function
  - Output layer with 10 units (one for each possible digit) and the `softmax` activation function (this produces a vector of length 10, each cell a probability of being a digit $0 - 9$
  - `rmsprop` optimization algorithm
  - `categorical_crossentropy` loss function
  - `accuracy` performance measure (proportion of times the correct class is chosen)

- Performance
  - Accuracy of 98.9% on training data
  - Accuracy of 97.8% on testing data
  - Note that this difference is expected since the network has “seen" the training data but not the testing data

MNIST Example

```
from keras.datasets import mnist
from keras import models
from keras import layers
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

network.fit(train_images, train_labels, epochs=5, batch_size=128)
test_loss, test_acc = network.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
```

Activation Functions for Hidden Layers

- `sigmoid` - no longer used for hidden layers, just the output layer
- `tanh` - better than the `sigmoid` function but less used than the `relu` function
- `relu` - most popular and commonly used
- `leaky relu` - sometimes performs better than the `relu`, but is less often used

Activation and Loss Functions for Output Layer

Problem Type	**Last-layer activation**	**Loss Function**
Binary Classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Optimization Algorithms

- Stochastic gradient descent (`sgd`)
- RMS prop (`rmsprop`)
- Adagrad (`adagrad`)
- Adam (`adam`)
- You can also customize the algorithms by changing the learning rate and other hyperparameters
- For more information: `https://keras.io/optimizers/`
- We will be using RMS prop for most of our examples

IMDB Example

- Training set: 25,000 either positive or negative movie reviews that have each been turned into a vector of integers
  - We'll see how to actually do this later in the course
  - Each review can be of any length
  - Only the top 10,000 most frequently occurring words are kept i.e. rare words are discarded
  - Each review includes a label: 0 = negative review and 1 = positive review
- Testing set: 25,000 either positive or negative movie reviews, similar to the training set

- Data Manipulation
  - Each review is of a varying length and is a list of integers - we need to turn this into a tensor with a common length for each review
  - Create a 2D tensor of shape $25,000 \times 10,000$: $25,000$ reviews and $10,000$ possible words
  - Use the `vectorize_sequences` function to turn a movie review list of integers into a vector of length $10,000$ with 1s for each word that appears in the review and 0s for words that do not
  - The labels are already 0s and 1s, so the only thing we need to do is make them float numbers

IMDB Example

- Network Architecture
  - 3 layers: 2 hidden and 1 output layer
  - Hidden layers with 16 hidden units each and the `relu` activation function
  - Output layer with 1 units (the probability of a review being positive) and the `sigmoid` activation function
  - `rmsprop` optimization algorithm
  - `binary_crossentropy` loss function
  - `accuracy` performance measure (proportion of times the correct class is chosen)

- Performance
  - Peak accuracy of 99.98% on training data
  - Peak accuracy of 89.1% on validation data

IMDB Example

```
1  import keras
2  from keras.datasets import imdb
3  import numpy as np
4  from keras import models
5  from keras import layers
6
7  (train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words
       =10000)
8
9  def vectorize_sequences(sequences, dimension=10000):
10     results = np.zeros((len(sequences), dimension))
11     for i, sequence in enumerate(sequences):
12         results[i, sequence] = 1.
13     return results
14
15 x_train = vectorize_sequences(train_data)
16 x_test = vectorize_sequences(test_data)
17
18 y_train = np.asarray(train_labels).astype('float32')
19 y_test = np.asarray(test_labels).astype('float32')
20
21 model = models.Sequential()
22 model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
23 model.add(layers.Dense(16, activation='relu'))
24 model.add(layers.Dense(1, activation='sigmoid'))
25
26 model.compile(optimizer='rmsprop',
27               loss='binary_crossentropy',
28               metrics=['accuracy'])
```

IMDB Example

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

Model Evaluation and Tuning

Parameters vs Hyperparameters

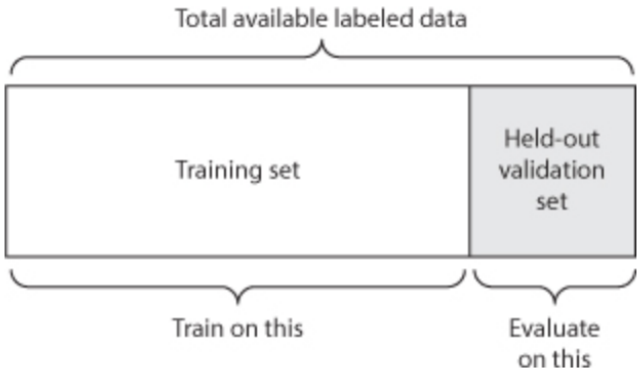
- The main goal of a deep neural network isn't high accuracy on the training set - it is high accuracy on the test set
- With high accuracy on different test sets, the network is said to be **generalizable** and thus more useful
- Networks are improved by "tuning" their hyperparameters - not to be confused with their parameters
- Parameters:
  - $\boldsymbol{w}$, b
- Hyperparameters:
  - ϵ: the learning rate
  - e: the number of epochs
  - L: the number of hidden layers
  - K_l: the number of hidden units in each each hidden layer
  - b: batch size or mini-batch size
  - the activation function for each hidden layer
  - regularization parameters
- The best hyperparameters are chosen via a very iterative process mainly involving a **validation** or **development** set

Training, Validation and Test Sets

- These %s aren't the only options – just the more common ones
- For very large data sets, smaller percentages for the validation and test sets is common (~1-2%)
- One important thing to keep in mind is that the training set should come from the same distribution as the validation and test sets

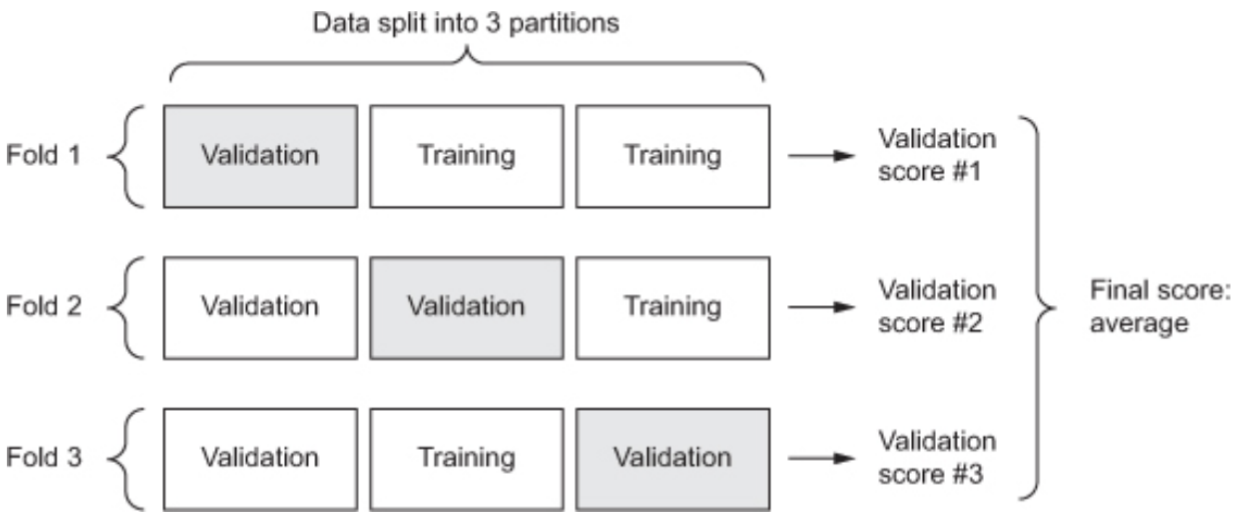
Simple Hold-Out Validation

- The simplest evaluation protocol
- Suffers from one flaw: if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand
- If different random shuffling rounds of the data before splitting yield very different measures of model performance, then you're suffering from this issue
- This problem can be addressed with K-fold validation and iterated K-fold validation with shuffling



K-Fold Validation

- Split data into K partitions of equal (or nearly equal) size
- For each partition i, train a model on the remaining $K - 1$ partitions, and evaluate performance on partition i
- The final performance score is the average of the K scores
- Useful when the performance of your model shows significant variance based on your train-test split



Iterated K-Fold Validation with Shuffling

- Useful when relatively little data is available
- Apply K-fold validation multiple times, shuffling the data every time before splitting it K ways
- The final score is the average of the scores obtained at each run of K-fold validation
- Note that in total, this method trains and evaluates $P \times K$ models (P is the number of iterations), and this can be very computationally expensive

Things to keep in mind

- **Data Representativeness**- you want both your training set and test set to be representative of the data at hand. Randomly shuffling your data can help ensure this holds.
- **Time**- if you are trying to predict the future given the past, you should *not* randomly shuffle your data before splitting it. Doing so will create a ***temporal leak***: your model will effectively be trained on data from the future.
- **Redundancy**- if some data points appear more than once in your data, make sure they are only in one of the training or testing sets. If you randomly shuffle your data and the same data point appears in both, you'll be testing on part of your training data, which is the worst thing to do. To avoid this, be sure your training and test sets are disjoint.

Capacity, Overfitting, Underfitting

Capacity, Overfitting, Underfitting

- Central challenge is that algorithm must perform well on new, previously unseen inputs
- This ability is called **generalization**
- Typically when training a model we have access to a training set, and computing an error measure on it gives the **training error**
- So far we've described and solved an optimization problem
- In machine learning we want the **generalization error**, also called **test error**, estimated using a test set that is separate from the training set, to be low as well
- Generalization error is defined as the expected value of the error on a new input, where the expectation is taken across different possible inputs

Capacity, Overfitting, Underfitting

- We can make progress by assuming that the training data and test data are generated by a probability distribution over datasets called the **data-generating process**
- We typically assume that examples in each dataset are **independent** from each other, and that the training set and test set are **identically distributed**, drawn from the same probability distribution
- This **i. i. d. assumption** enables us to describe the data-generating process with a probability distribution over a single example
- This distribution is known as the **data-generating distribution**, and it is used to generate every train example and every test example
- If we selected a model at random (with fixed parameters), the expected training error would be equal to the expected test error of that model, since the only difference between the two conditions is the name we assign to the sampled dataset

Capacity, Overfitting, Underfitting

- In practice, we first sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set
- Under this process, expected test error $\geq$ expected training error
- Factors determining how well a learning algorithm can perform are its ability to:
  - Make the training error small
  - Make the gap between training error and test error small
- These factors correspond to the two central challenges in machine learning:
  - **Underfitting** occurs when the model is not able to obtain a sufficiently low value on the training set
  - **Overfitting** occurs when the gap between the training error and test error is too large

Capacity, Overfitting, Underfitting

- We can control whether a model is likely to overfit or underfit by controlling its **capacity**, which informally is a model's ability to fit a wide variety of functions
- One way to control capacity is by choosing the model's **hypothesis space**, the set of functions that the learning algorithm is allowed to select as being the solution
- For example, in linear regression, we can include polynomials of x, increasing its capacity (the model is still linear in parameters)
- Machine learning algorithms generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with

Capacity, Overfitting, Underfitting

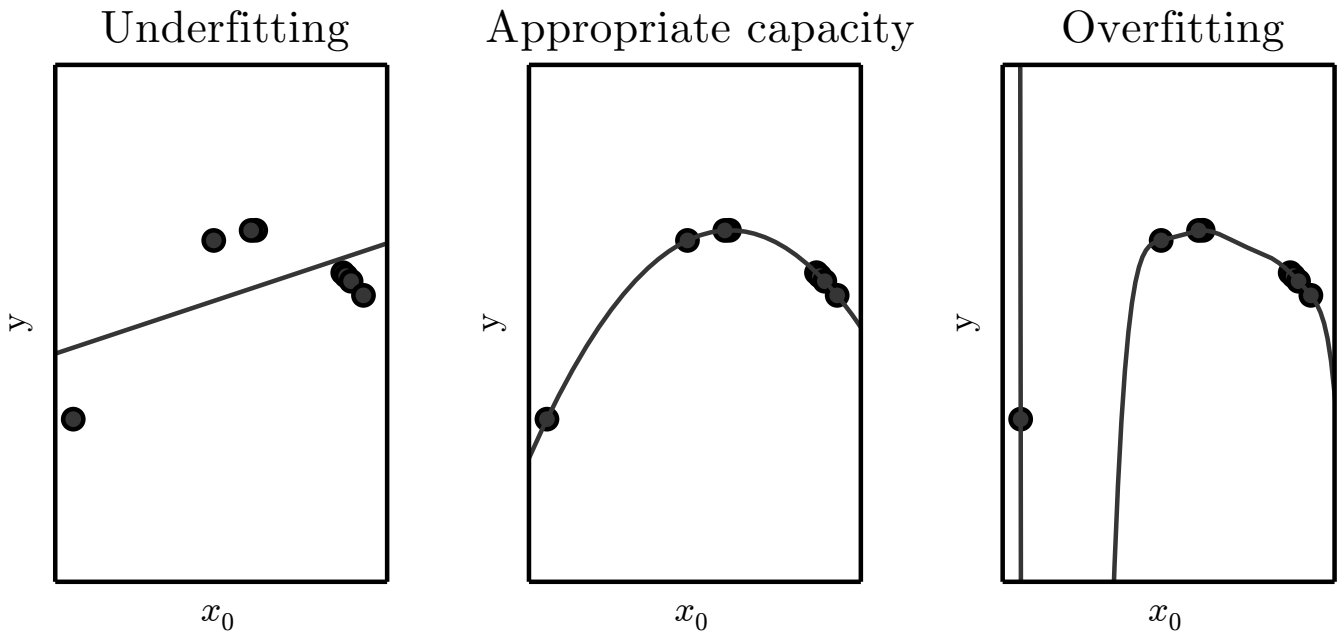


Figure: Here true model is quadratic, so the quadratic function (middle) fits well. Source: DL.

Capacity, Overfitting, Underfitting

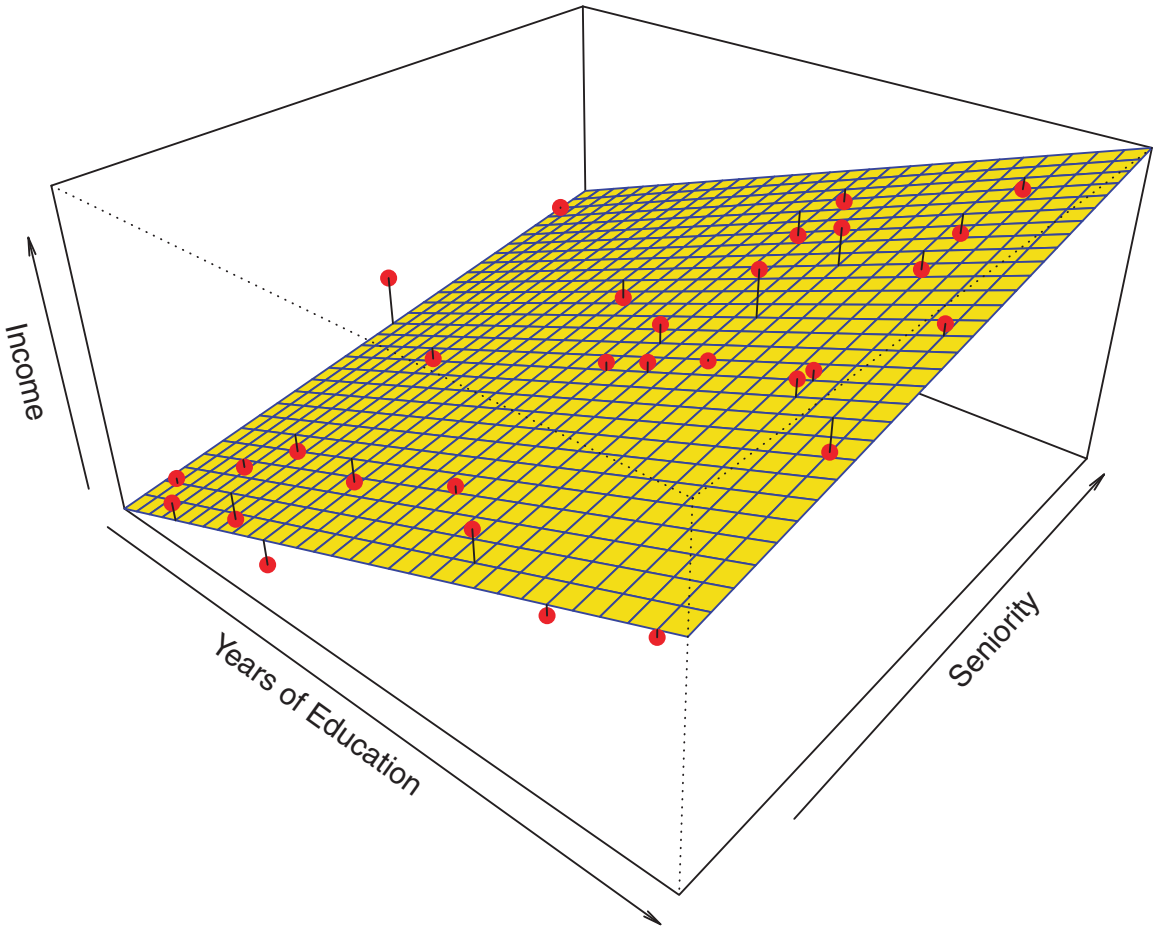


Figure: Underfitting model. Source: ISL.

Capacity, Overfitting, Underfitting

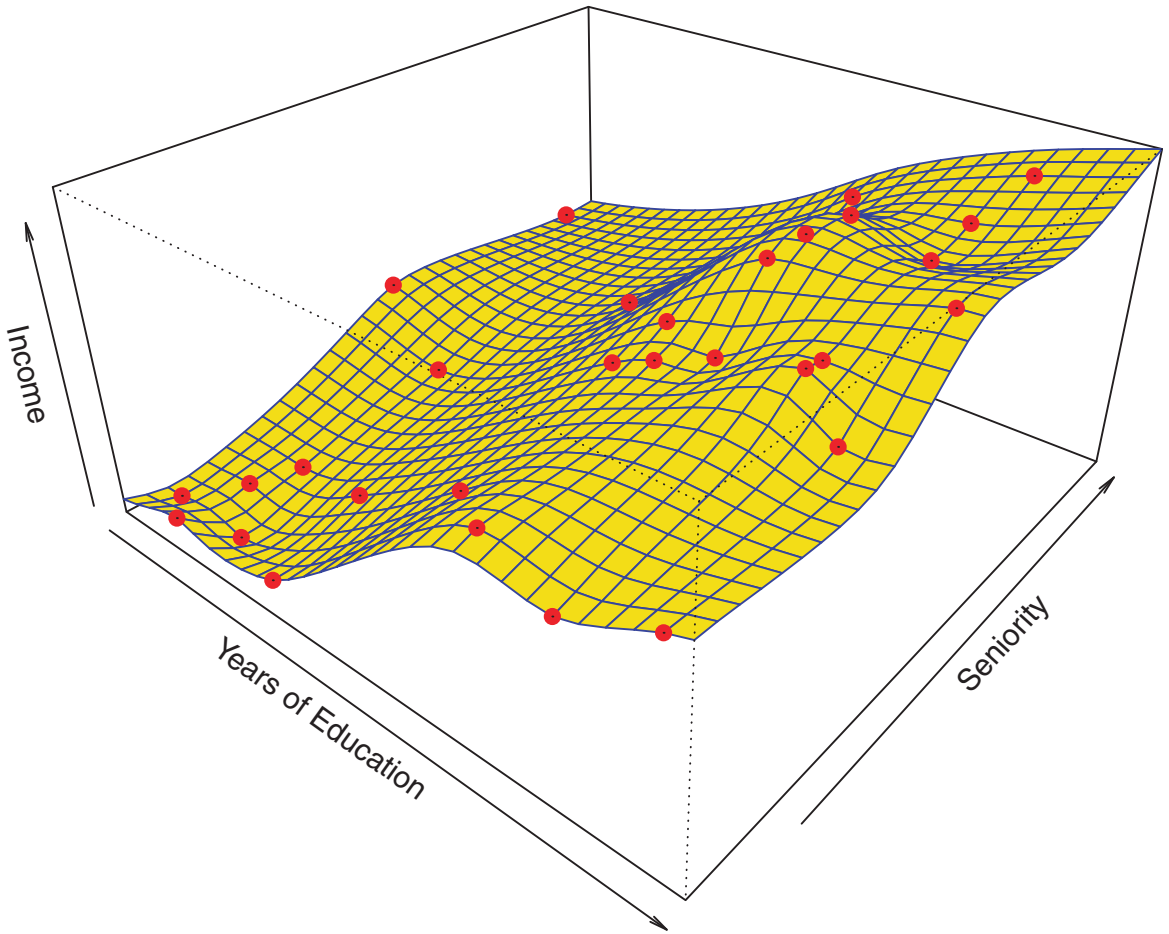


Figure: Overfitting model with zero training error. Source: ISL.

Capacity, Overfitting, Underfitting

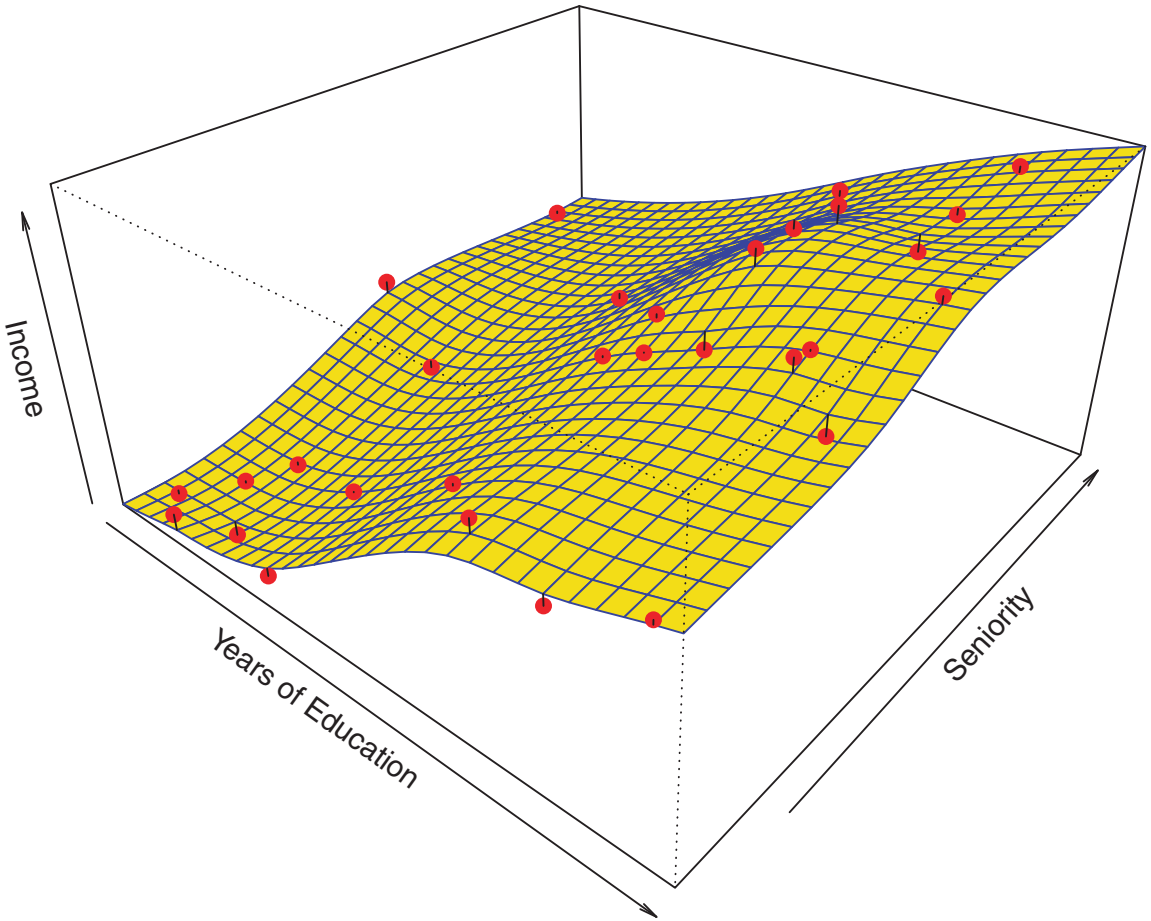


Figure: Appropriate model capacity. Source: ISL.

Capacity, Overfitting, Underfitting

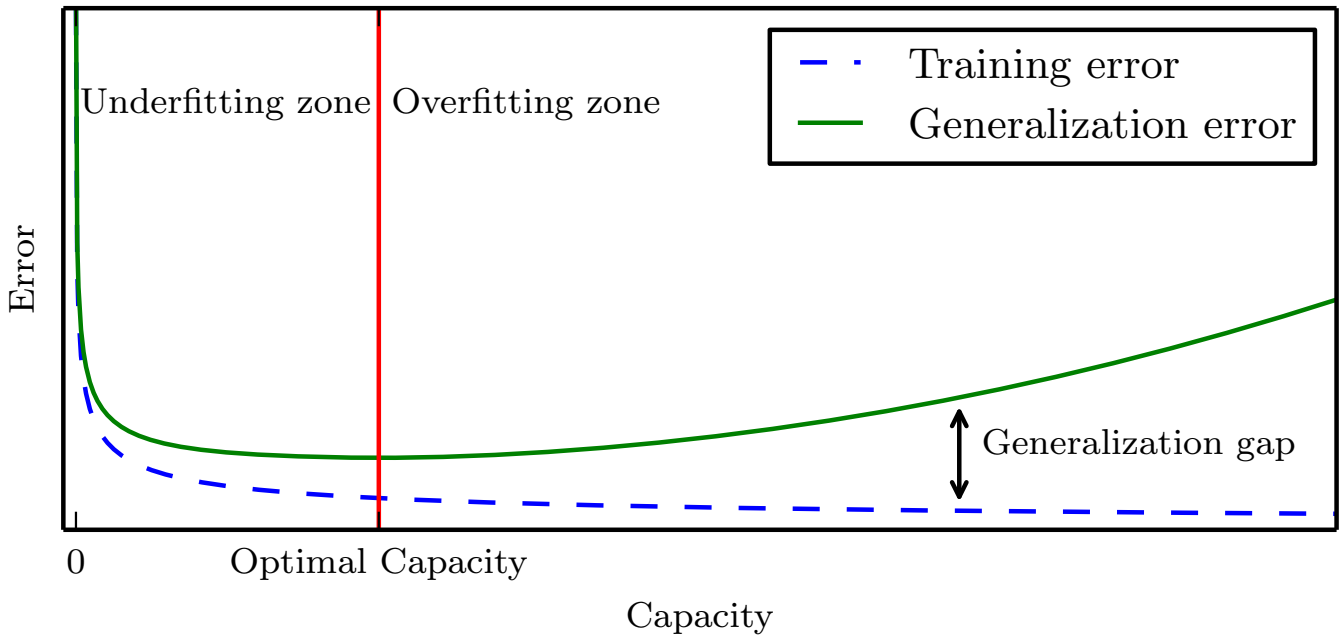
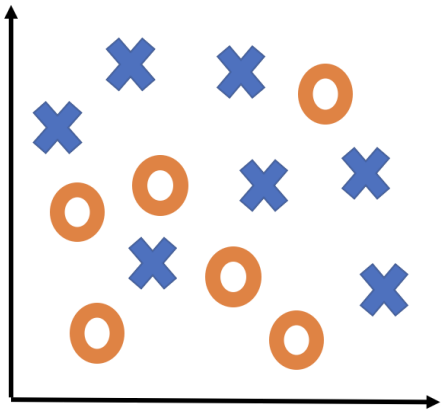
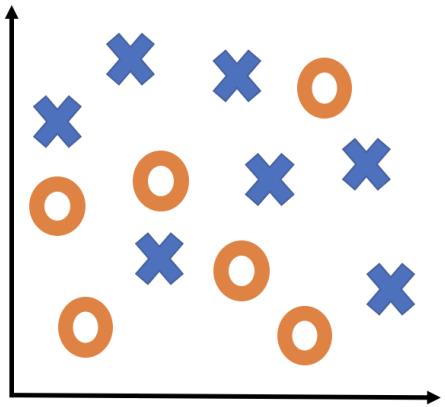


Figure: Typical relationship between capacity and error. Source: DL.

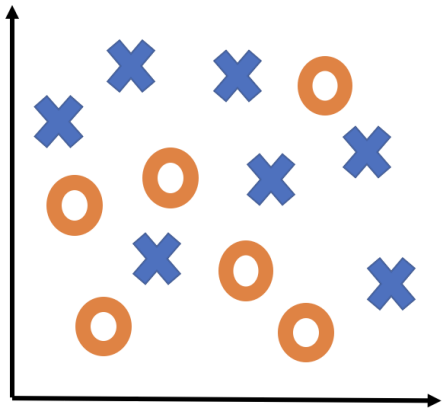
Bias and Variance



High Bias



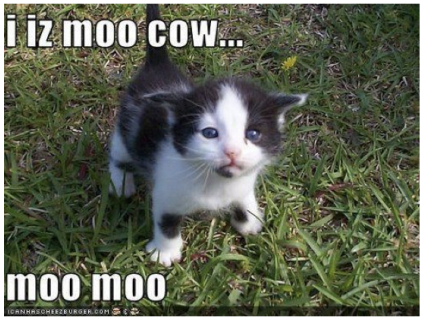
"Just Right"



High Variance

Bias and Variance

Example:
Cat classification



y = 1

y = 0

Training set error

Validation set error

Human error:
Optimal (Bayes) error:

Basic Recipe for Machine Learning

High Bias?
(training data performance)

Bigger Network
- Limiting factor = computing time
- Train longer
- Network architecture

High Variance?
(validation data performance)

More Data
- If possible
- Regularization
- Network architecture

Done

Regularization

Regularization

- One of the easiest ways to fight overfitting is with regularization
- We will focus on 3 different methods:
  - Reducing the size of the network
  - Weight regularization
  - Dropout

Reducing network size

- The simplest way to prevent overfitting is to reduce the network's size: the number of learnable parameters in the model
- This is determined by the number of layers and units in each of those layers
- The number learnable parameters is what we referred to before as a model's 'capacity'
- As we saw before, we need to find a 'just right' solution that doesn't under or overfit
- There isn't a formula for this - you must train several different networks and evaluate them all
- Start with relatively few layers and units, and work your way up until you see diminishing returns in terms of validation loss

IMDB Classification Example

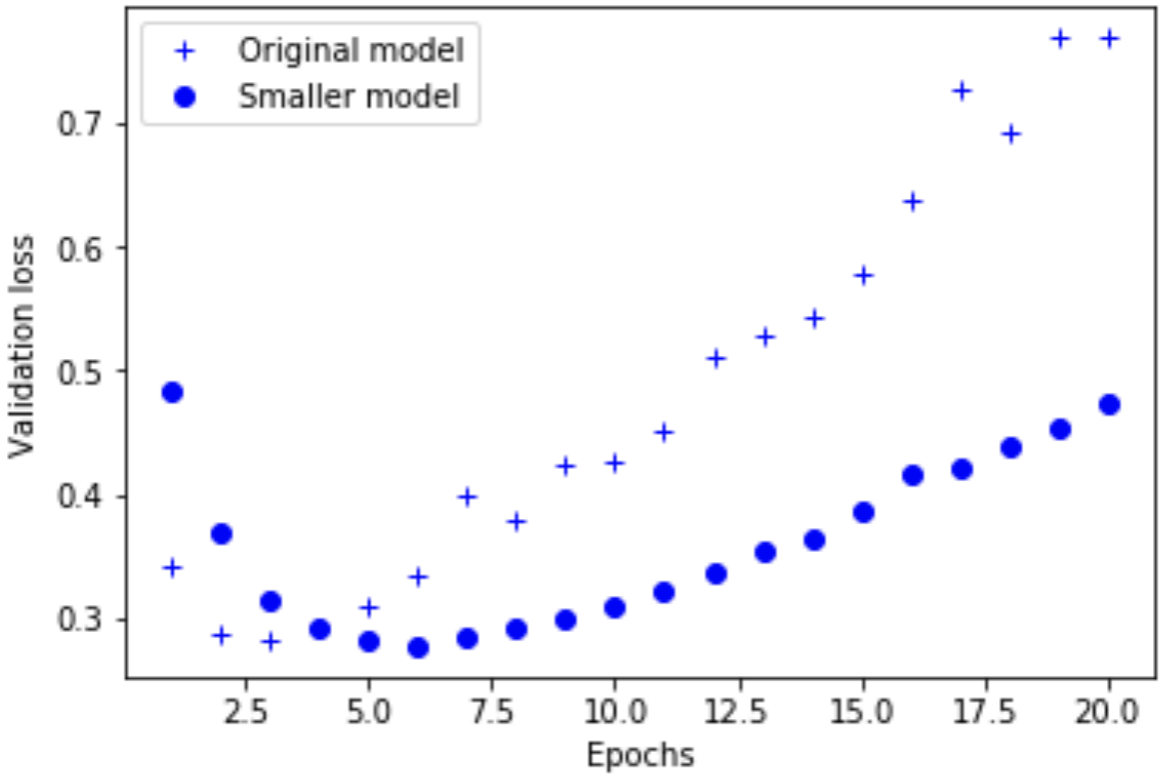
- Original Model:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- Lower Capacity Model:

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

IMDB Classification Example



IMDB Classification Example

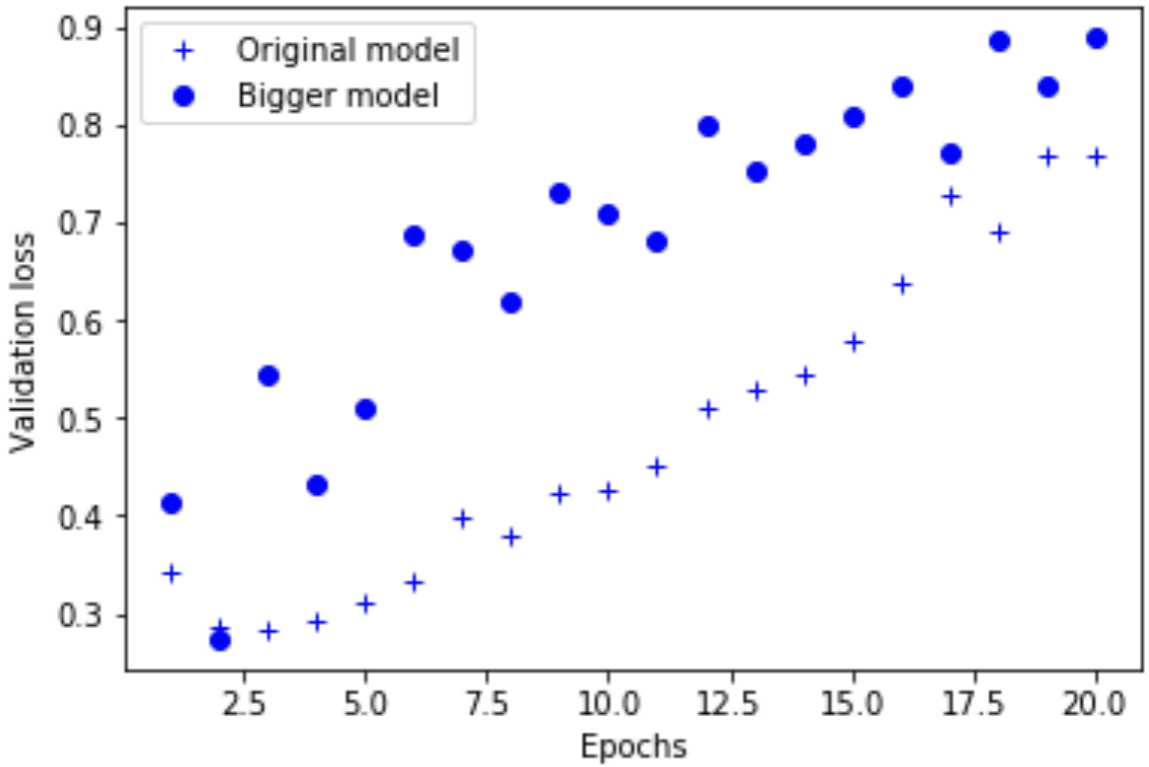
- Original Model:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- Higher Capacity Model:

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

IMDB Classification Example



IMDB Classification Example