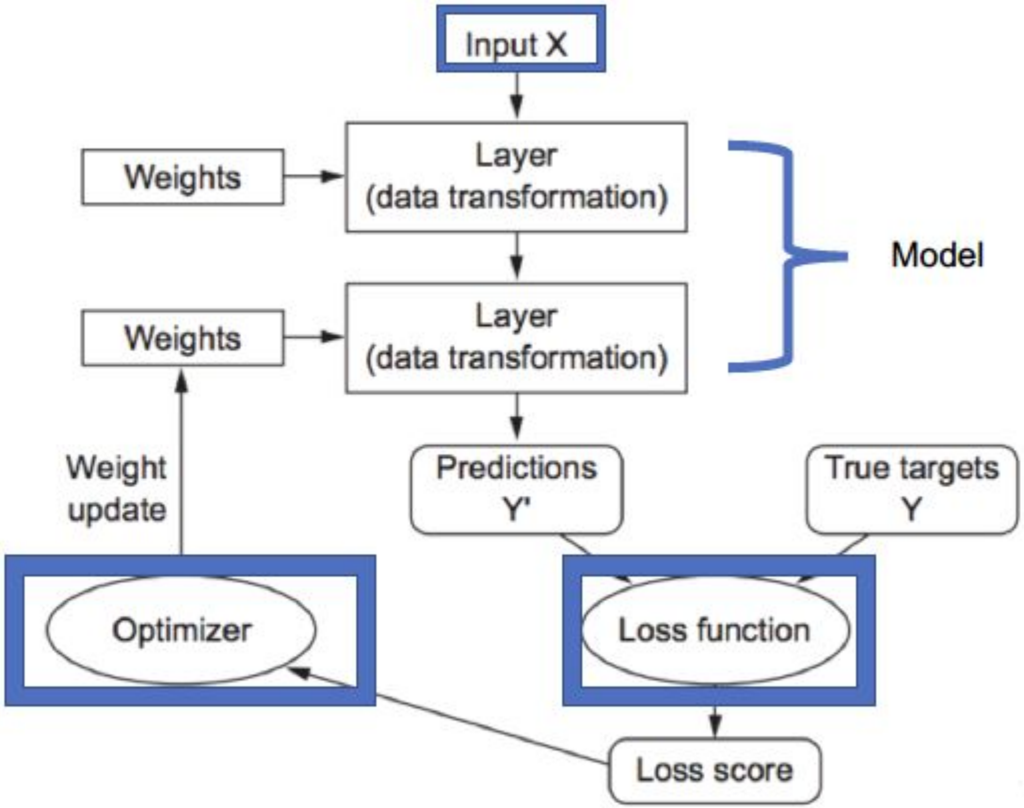
BST 261: Data Science II
Lecture 3
Feedforward networks in Python with Keras

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2019

Neural Network Architecture



Generic Feedforward Network

```
# 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)
```

Generic Feedforward Network

train_data: training examples (matrix of feature vectors; $\mathbf{X}_{train}$)

train_labels: training labels ($\mathbf{y}_{train}$)

test_data: test examples used to measure performance of network ($\mathbf{X}_{test}$)

test_labels: test set labels ($\mathbf{y}_{test}$)

Optimizing algorithms: rmsprop, sgd, adagrad, adam, etc.

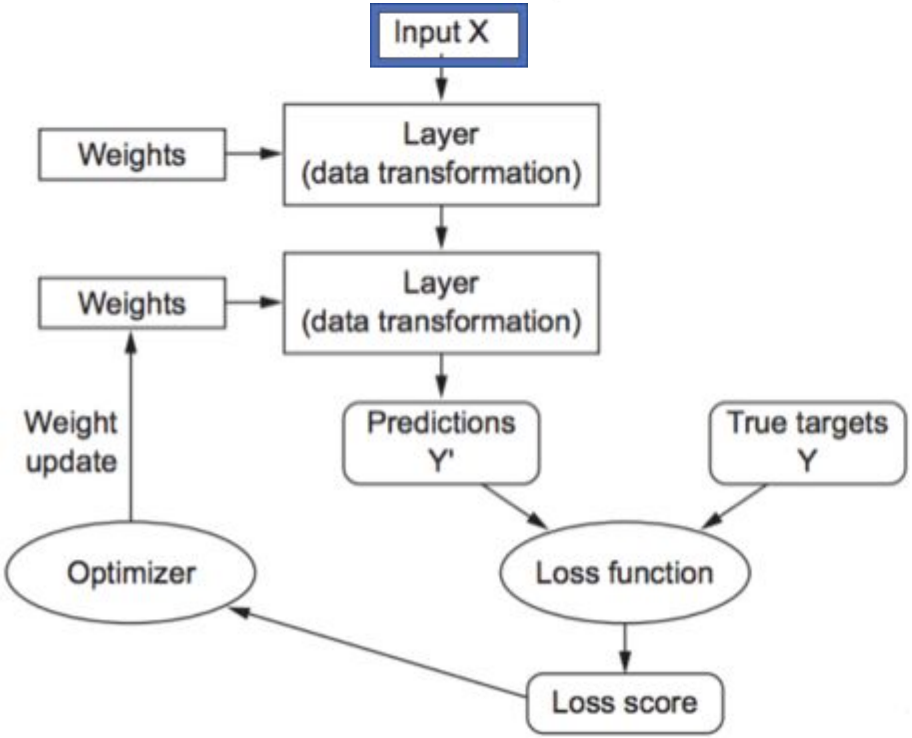
Loss function options: mse, mae, categorical_crossentropy, etc.

Performance measure options: accuracy, mae, etc.

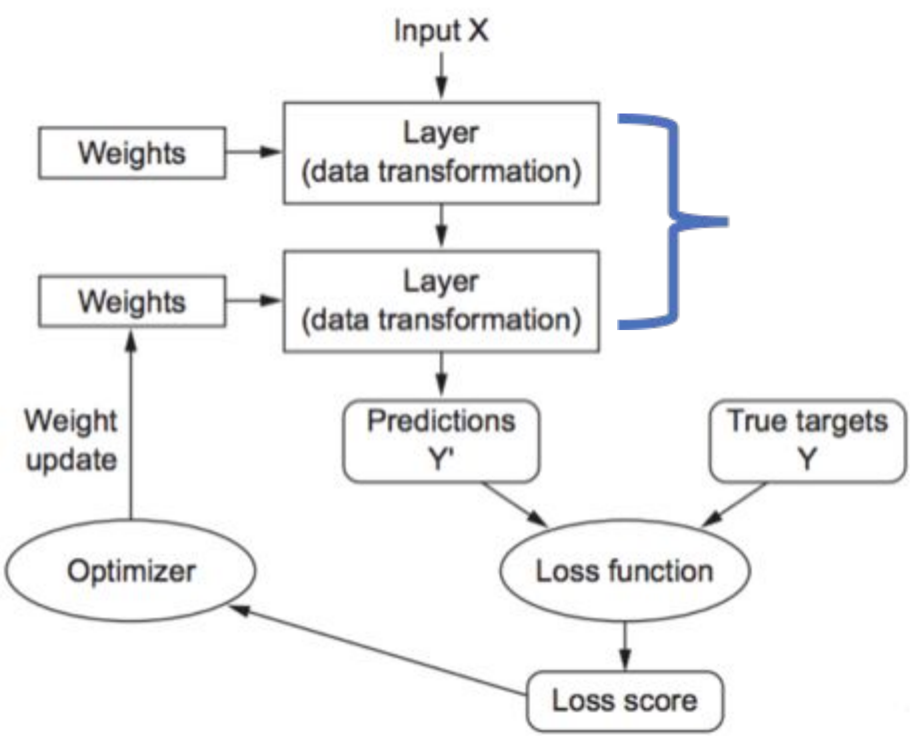
Here:

- ***c*** = the number of hidden units (neurons) in a hidden layer
- ***d*** = the number of units (neurons) in the output layer
- ***e*** = the number of epochs (iterations) over entire training data set
- ***b*** = the bath size (how many training examples to optimize as once)

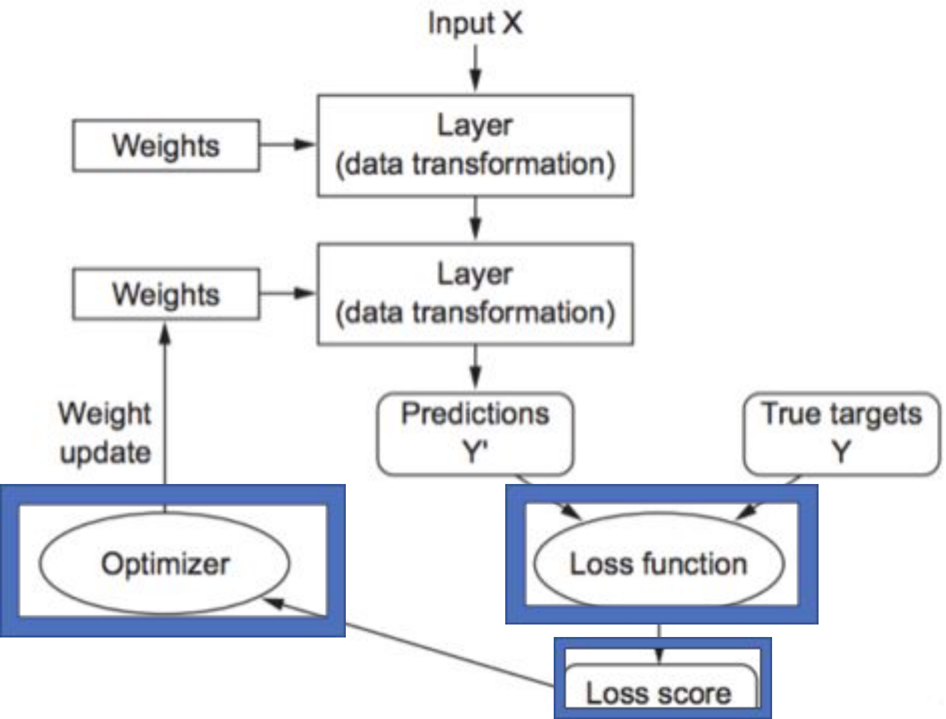
```python
# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=(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



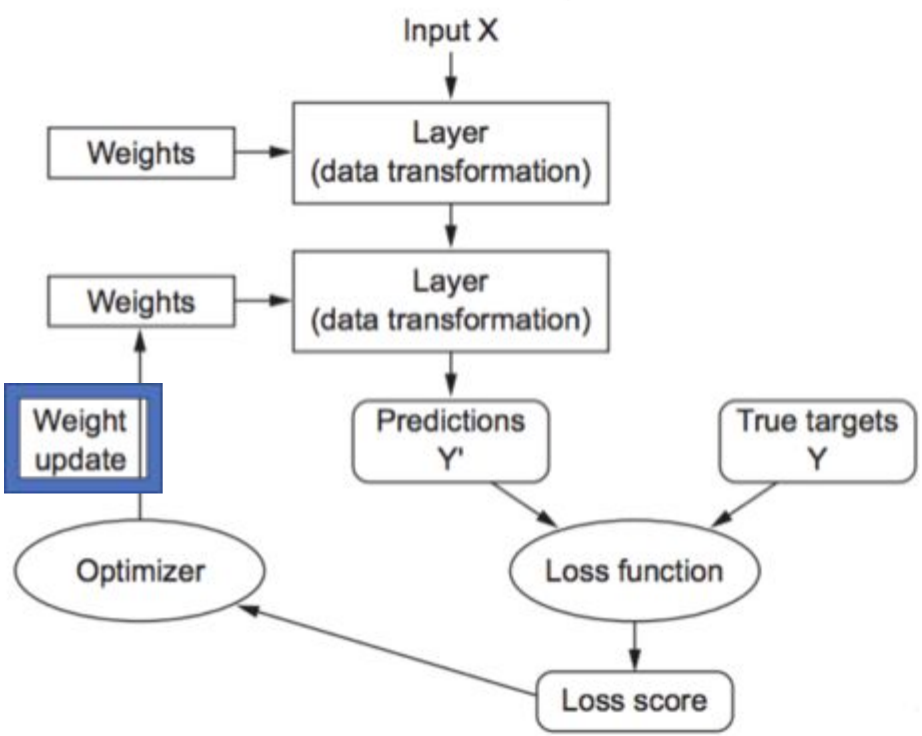
```python
# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=(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



```python
# 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)
```

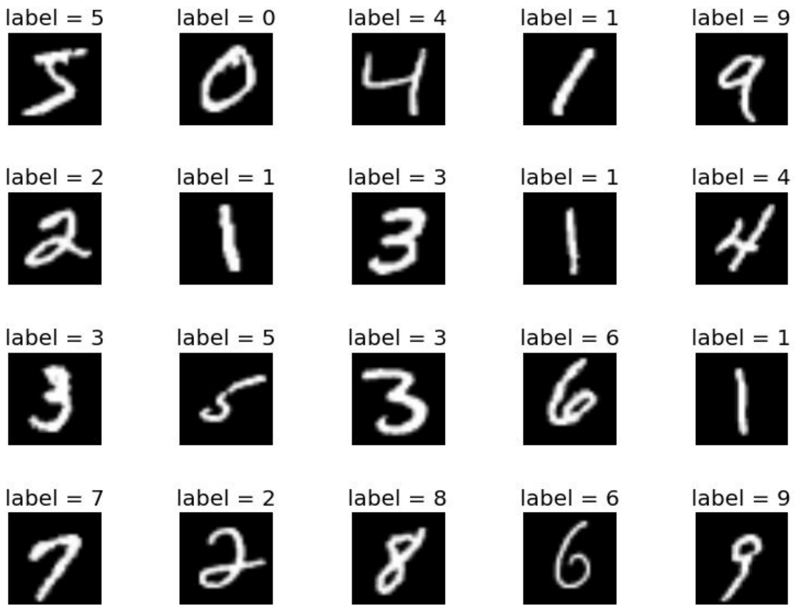


```python
# Format data to be fed into the network
(train_data, train_labels), (test_data, test_labels) = load_data()
# This tells keras you want a linear stack of layers
network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(c, activation='activation function', input_shape=(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

- The [MNIST data set]() includes handwritten digits with corresponding labels
- Training set: 60,000 images of handwritten digits and corresponding labels
  - Each digit is represented as a 28 x 28 matrix of grayscale 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



MNIST Data Example

Data wrangling

- We'll get into RGB images later, but for grayscale images, we need to transform the matrix of values into a vector of values. Neural networks also require "small" numbers - numbers between -1 and 1 are best (this ensures the gradients don't "blowup" or "vanish"
  - Reshape each image from a 28 x 28 matrix of grayscale values 0 - 255 to a vector of length 28*28 = 784 of values 0 - 1 (divide each by 255)

- We now have 10 classes (categories; the digits 0-9)
  - We need to have multiclass labels that tell the network which digit the example is
  - Reshape each corresponding image label to a vector of length 10 of values 0 or 1
  - Example: the digit 3 would be represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
  - You can think of this as "dummy coding" the labels

Activation and Loss Function Choices

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

Softmax function

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Softmax units are used as outputs when predicting a discrete variable *y* with *k* possible values

- In this setting, which can be seen as a generalization of the Bernoulli distribution, we need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = P(y = i|x)$

- We require that each $\hat{y}_i$ lie in the [0, 1] interval and that the entire vector sums to 1

- We first compute $z = w^T x + b$ as usual
- Here, $z_i = log[\tilde{P}(y = i|x)]$ represents an unnormalized log probability for class *i*
- The softmax function then exponentiates and normalizes *z* to obtain $\hat{\mathbf{y}}$

Softmax function

- In this case we want to maximize

$$log[P(y=i;z)] = log[\mathrm{softmax}(z)_i] = z_i - log\sum_j exp(z_j)$$

- The first term shows that the input always has a direct contribution to the loss function

- Because $log\sum_j exp(z_j) \approx max_j z_j$, the negative log-likelihood loss function always strongly penalizes the most active incorrect prediction

MNIST Data Example

Network Architecture

- Let’s start with 2 layers:
  - 1 hidden and 1 output layer
  - Hidden layer will have 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, where each element is a probability between 0 and 1 of the image being classified as that digit)
  - Example: [0, 0.3, 0, 0, 0, 0, 0, 0.7, 0, 0] - the highest probability corresponds to a label of 7, so the network would classify this image as a 7
  - **rmsprop optimization algorithm**
  - **categorical_crossentropy loss function**
  - **accuracy performance measure** (the proportion of times the correct class is chosen)

MNIST Data Example

```
(train_data, train_labels), (test_data, test_labels) = mnist.load_data()

network = models.Sequential()
# Layer 1 (Hidden layer)
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28, )))
# Layer 2 (Output layer)
network.add(layers.Dense(10, activation='softmax'))

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

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

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

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

network.fit(train_data, train_labels, epochs=5, batch_size=128)
test_loss, test_acc = network.evaluate(test_data, test_labels)
print('test accuracy:', test_acc)
```

MNIST Data Example

```
Epoch 1/5
60000/60000 [==============================] - 8s 135us/step - loss: 0.2576 - acc: 0.9244
Epoch 2/5
60000/60000 [==============================] - 7s 119us/step - loss: 0.1030 - acc: 0.9695
Epoch 3/5
60000/60000 [==============================] - 8s 131us/step - loss: 0.0678 - acc: 0.9798
Epoch 4/5
60000/60000 [==============================] - 8s 130us/step - loss: 0.0493 - acc: 0.9852
Epoch 5/5
60000/60000 [==============================] - 7s 123us/step - loss: 0.0370 - acc: 0.9892
10000/10000 [==============================] - 1s 54us/step
test accuracy: 0.9778
```

MNIST Data Example

```
Epoch 1/5
60000/60000 [==============================] - 8s 135us/step - loss: 0.2576 - acc: 0.9244
Epoch 2/5
60000/60000 [==============================] - 7s 119us/step - loss: 0.1030 - acc: 0.9695
Epoch 3/5
60000/60000 [==============================] - 8s 131us/step - loss: 0.0678 - acc: 0.9798
Epoch 4/5
60000/60000 [==============================] - 8s 130us/step - loss: 0.0493 - acc: 0.9852
Epoch 5/5
60000/60000 [==============================] - 7s 123us/step - loss: 0.0370 - acc: 0.9892
10000/10000 [==============================] - 1s 54us/step
test accuracy: 0.9778
```

Training set accuracy

IMDb Data Example

The [IMDb data set] is a set of movie reviews that have been labeled as either positive or negative, based on the text content of the reviews

- 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

IMDb Data Example

Data Wrangling

- 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 x 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

Activation and Loss Function Choices

Task	Last-layer activation	Loss function
Binary classification	sigmoid	Binary cross-entropy
Multiclass, single-label classification	softmax	Categorical cross-entropy
Multiclass, multilabel classification	sigmoid	Binary cross-entropy
Regression to arbitrary values	None	Mean square error (MSE)
Regression to values between 0 and 1	sigmoid	MSE or binary cross-entropy

IMDb Data Example

Network Architecture

- 3 layers
  - 2 hidden layers and 1 output layer
  - Hidden layers have 16 hidden units each and a **relu activation function**
  - Output layer has 1 unit (the probability a review is positive)
- **Sigmoid activation function**
- **rmsprop optimization algorithm**
- **binary_crossentropy loss function**
- **accuracy performance measure** (proportion of times the correct class is chosen)

IMDb Data Example

```
import keras
from keras.datasets import imdb
import numpy as np
from keras import models
from keras import layers

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)

y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')

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'))

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

IMDb Data 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))

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

history is a dictionary that contains data about what happened during training.

It contains 4 entries: the training loss and accuracy and the validation loss and accuracy

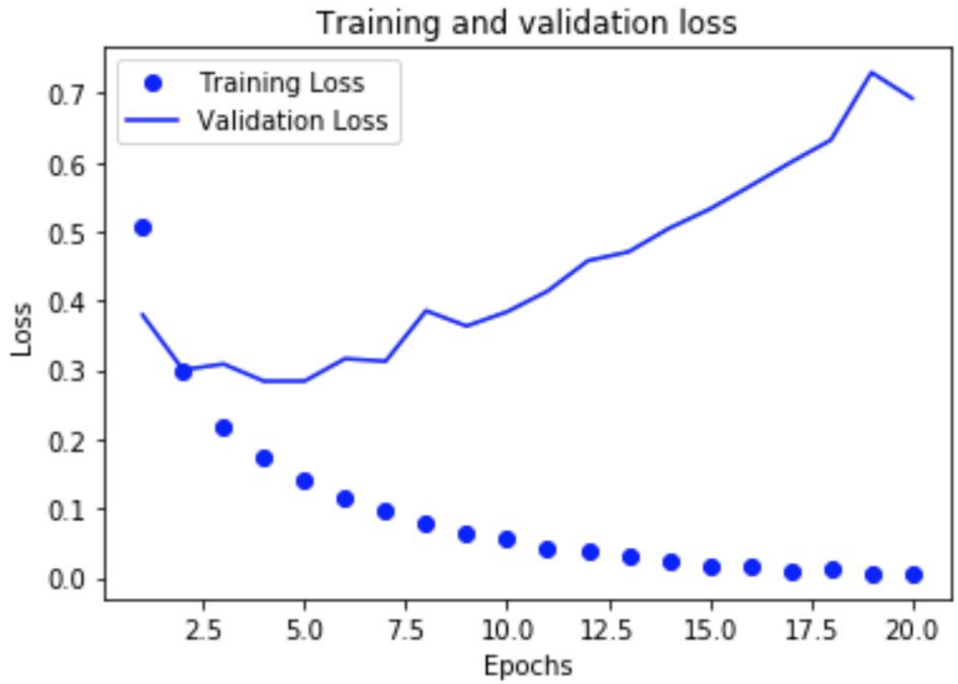
IMDb Data Example

```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 5s 363us/step - loss: 0.5084 - acc: 0.7813 - val_loss: 0.3797 - val_acc: 0.8684
Epoch 2/20
15000/15000 [==============================] - 3s 170us/step - loss: 0.3004 - acc: 0.9047 - val_loss: 0.3003 - val_acc: 0.8897
Epoch 3/20
15000/15000 [==============================] - 2s 109us/step - loss: 0.2179 - acc: 0.9285 - val_loss: 0.3087 - val_acc: 0.8711
Epoch 4/20
15000/15000 [==============================] - 1s 98us/step - loss: 0.1751 - acc: 0.9438 - val_loss: 0.2839 - val_acc: 0.8832
Epoch 5/20
15000/15000 [==============================] - 1s 96us/step - loss: 0.1427 - acc: 0.9542 - val_loss: 0.2841 - val_acc: 0.8871
Epoch 6/20
15000/15000 [==============================] - 1s 95us/step - loss: 0.1150 - acc: 0.9650 - val_loss: 0.3171 - val_acc: 0.8773
Epoch 7/20
15000/15000 [==============================] - 1s 99us/step - loss: 0.0980 - acc: 0.9705 - val_loss: 0.3126 - val_acc: 0.8846
Epoch 8/20
15000/15000 [==============================] - 2s 118us/step - loss: 0.0807 - acc: 0.9763 - val_loss: 0.3858 - val_acc: 0.8650
Epoch 9/20
15000/15000 [==============================] - 2s 102us/step - loss: 0.0661 - acc: 0.9820 - val_loss: 0.3634 - val_acc: 0.8783
Epoch 10/20
15000/15000 [==============================] - 2s 104us/step - loss: 0.0563 - acc: 0.9850 - val_loss: 0.3840 - val_acc: 0.8796
Epoch 11/20
15000/15000 [==============================] - 2s 104us/step - loss: 0.0435 - acc: 0.9899 - val_loss: 0.4146 - val_acc: 0.8784
Epoch 12/20
15000/15000 [==============================] - 2s 105us/step - loss: 0.0380 - acc: 0.9921 - val_loss: 0.4548 - val_acc: 0.8684
Epoch 13/20
15000/15000 [==============================] - 1s 99us/step - loss: 0.0300 - acc: 0.9929 - val_loss: 0.4702 - val_acc: 0.8726
Epoch 14/20
15000/15000 [==============================] - 2s 118us/step - loss: 0.0247 - acc: 0.9943 - val_loss: 0.5041 - val_acc: 0.8716
Epoch 15/20
15000/15000 [==============================] - 1s 94us/step - loss: 0.0190 - acc: 0.9967 - val_loss: 0.5314 - val_acc: 0.8705
Epoch 16/20
15000/15000 [==============================] - 1s 97us/step - loss: 0.0169 - acc: 0.9967 - val_loss: 0.5644 - val_acc: 0.8685
Epoch 17/20
15000/15000 [==============================] - 1s 93us/step - loss: 0.0115 - acc: 0.9987 - val_loss: 0.5989 - val_acc: 0.8667
Epoch 18/20
15000/15000 [==============================] - 2s 103us/step - loss: 0.0123 - acc: 0.9977 - val_loss: 0.6313 - val_acc: 0.8677
Epoch 19/20
15000/15000 [==============================] - 2s 106us/step - loss: 0.0063 - acc: 0.9997 - val_loss: 0.7389 - val_acc: 0.8518
Epoch 20/20
15000/15000 [==============================] - 2s 108us/step - loss: 0.0062 - acc: 0.9996 - val_loss: 0.6937 - val_acc: 0.8657
```

Training and Validation Loss

```
plt.plot(epochs, train_loss, 'bo', label = 'Training Loss')
plt.plot(epochs, val_loss, 'b', label = 'Validation Loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

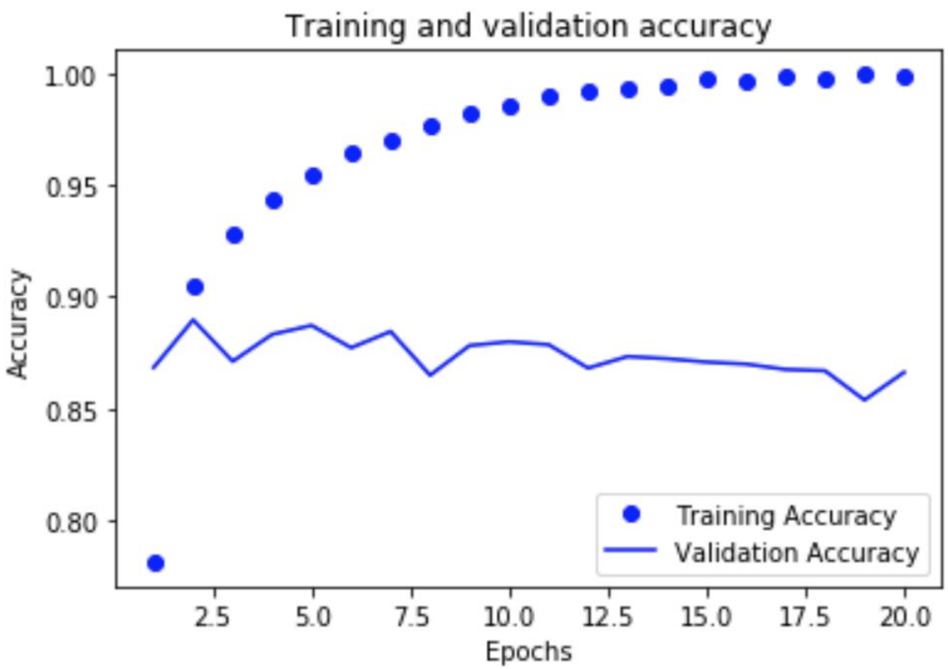
<matplotlib.legend.Legend at 0xb2c30f2b0>



Training and Validation Accuracy

```
plt.plot(epochs, train_acc, 'bo', label = 'Training Accuracy')
plt.plot(epochs, val_acc, 'b', label = 'Validation Accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xb209d0b70>
```



Test Set Accuracy

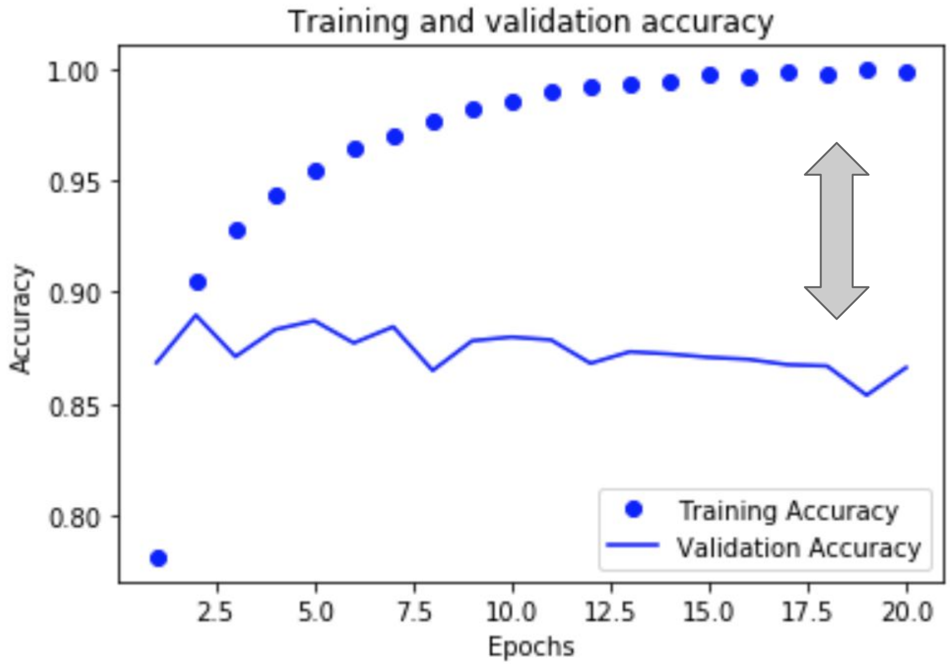
```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
print(results)
```

```
Epoch 1/4
25000/25000 [==============================] - 2s 70us/step - loss: 0.4584 - acc: 0.8133
Epoch 2/4
25000/25000 [==============================] - 2s 61us/step - loss: 0.2630 - acc: 0.9095
Epoch 3/4
25000/25000 [==============================] - 1s 59us/step - loss: 0.2005 - acc: 0.9282
Epoch 4/4
25000/25000 [==============================] - 1s 59us/step - loss: 0.1685 - acc: 0.9388
25000/25000 [==============================] - 1s 45us/step
[0.298845004286766, 0.88256]
```

How do we make this model better?

```
plt.plot(epochs, train_acc, 'bo', label = 'Training Accuracy')
plt.plot(epochs, val_acc, 'b', label = 'Validation Accuracy')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xb209d0b70>
```



There is a big difference in the training accuracy and validation set accuracy - a sign of overfitting.

How do we combat this?

Regularization: reducing network size

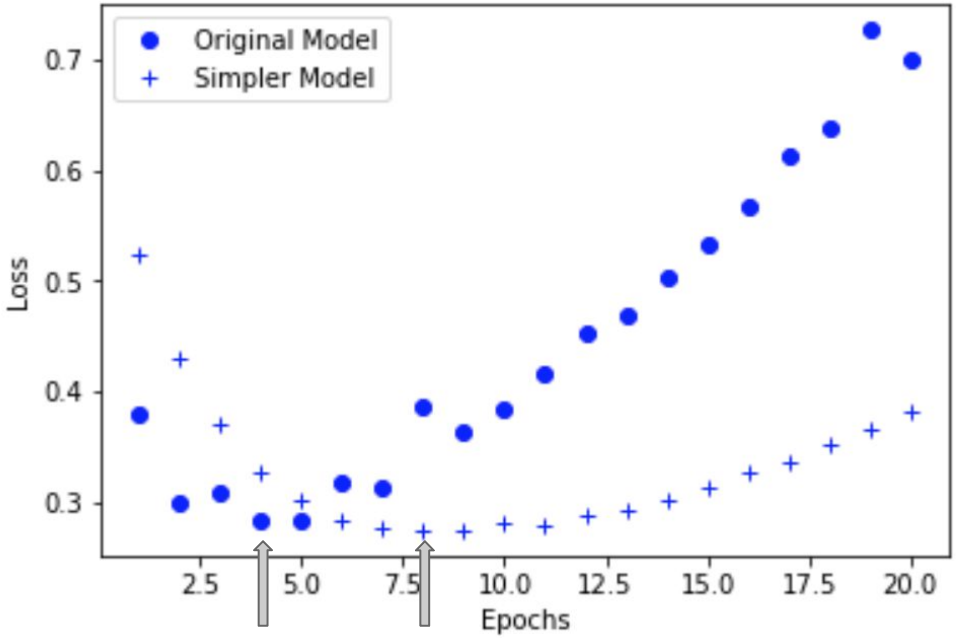
When we are battling overfitting, one option is to simplify the model. Let's compare the performance we get from a simpler model. Here we have simplified the model by reducing the number of hidden units in each hidden layer.

```
# Original Model
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'))

# Simpler, lower capaciy 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'))
```

Regularization: reducing network size

```
plt.plot(epochs, val_loss, 'bo', label = 'Original Model')
plt.plot(epochs, val_loss2, 'b+', label = 'Simpler Model')
plt.title('')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xb2ebcc240>
```



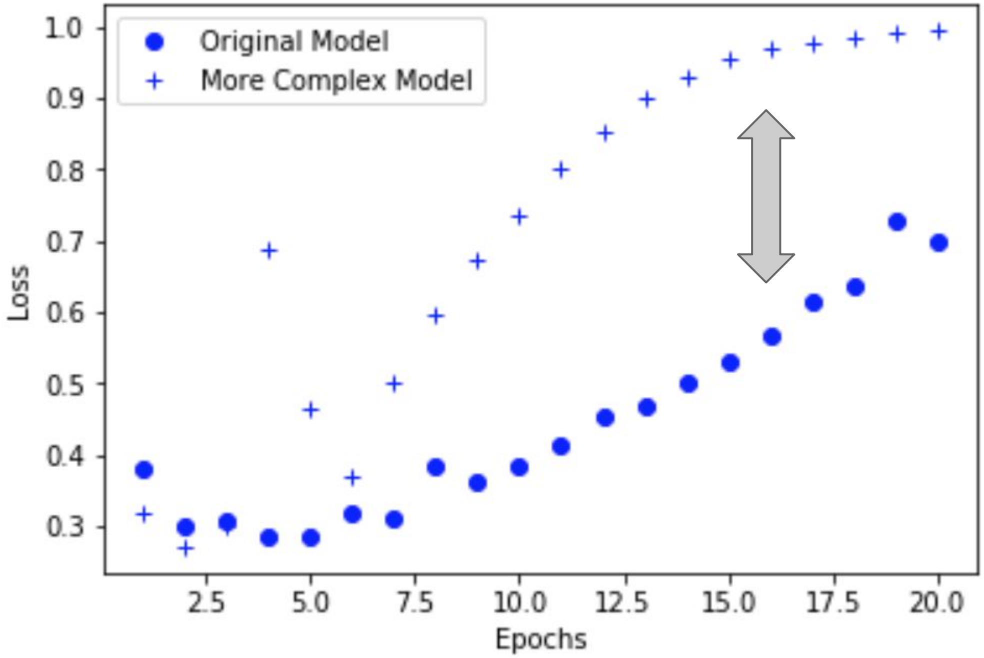
The smaller network performs better after training for more epochs. Rather than a maximum accuracy (here, minimum loss) at epoch 4, the smaller network has maximum accuracy at epoch 8.

What happens if we make the model more complex?

```
# Original Model
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'))
# More complex, higher capaciy model
model3 = models.Sequential()
model3.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model3.add(layers.Dense(512, activation='relu'))
model3.add(layers.Dense(1, activation='sigmoid'))

model3.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
plt.plot(epochs, val_loss, 'bo', label = 'Original Model')
plt.plot(epochs, val_loss3, 'b+', label = 'More Complex Model')
plt.title('')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xb2ef71b38>
```



The original model performs better than the more complex model with many more hidden nodes

Regularization: weight regularization

```
from keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu', input_shape=(10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                          activation='relu'))
l2_model.add(layers.Dense(1, activation='sigmoid'))
```

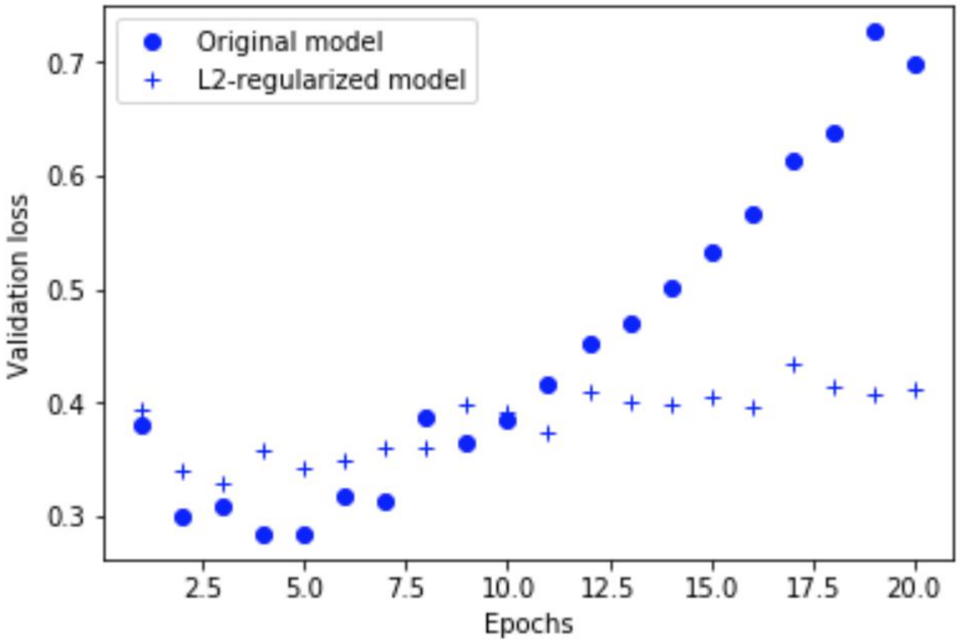
```
l2_model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy',
                 metrics=['acc'])
```

```
l2_model_hist = l2_model.fit(x_train, y_train,
                             epochs=20,
                             batch_size=512,
                             validation_data=(x_test, y_test))
```

Regularization: weight regularization

```
l2_model_val_loss = l2_model_hist.history['val_loss']

plt.plot(epochs, val_loss, 'bo', label='Original model')
plt.plot(epochs, l2_model_val_loss, 'b+', label='L2-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xc3c607588>
```



The L2-regularized model is much more resistant to overfitting - the validation loss starts to increase at a much slower rate

Regularization: adding dropout

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='relu'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))

dpt_model.compile(optimizer='rmsprop',
                  loss='binary_crossentropy',
                  metrics=['acc'])
```

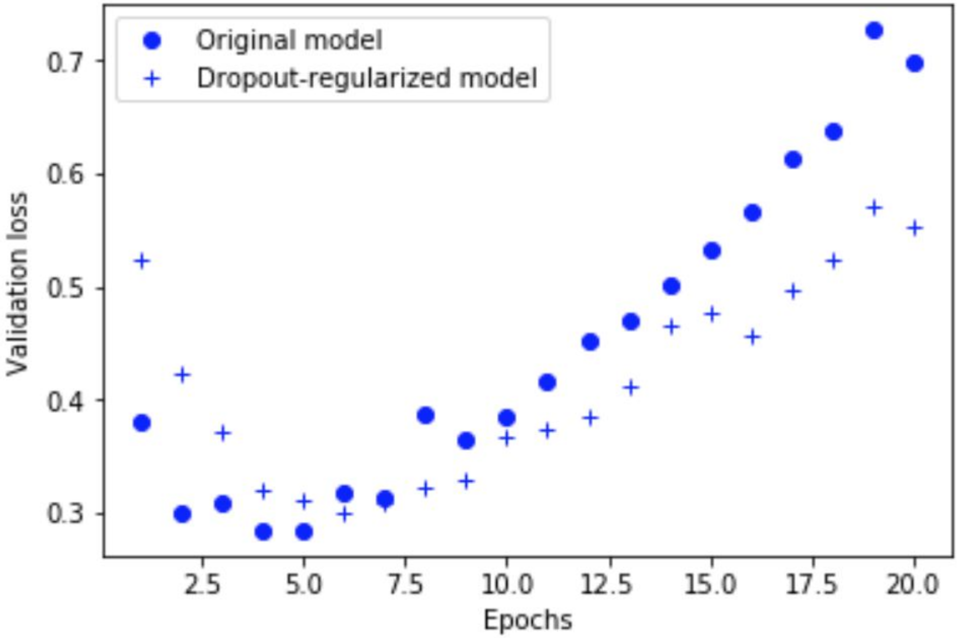
```
dpt_model_hist = dpt_model.fit(x_train, y_train,
                               epochs=20,
                               batch_size=512,
                               validation_data=(x_test, y_test))
```

The 0.5 indicates a 50% probability of dropping out a unit. Typically, 20% is used in practice, but you can try different values and see what performs best.

Regularization: adding dropout

```
dpt_model_val_loss = dpt_model_hist.history['val_loss']

plt.plot(epochs, val_loss, 'bo', label='Original model')
plt.plot(epochs, dpt_model_val_loss, 'b+', label='Dropout-regularized model')
plt.xlabel('Epochs')
plt.ylabel('Validation loss')
plt.legend()
```

```
<matplotlib.legend.Legend at 0xc3af9acc0>
```



The dropout model is slightly better than the original model but does not control for overfitting as well as the L2 network