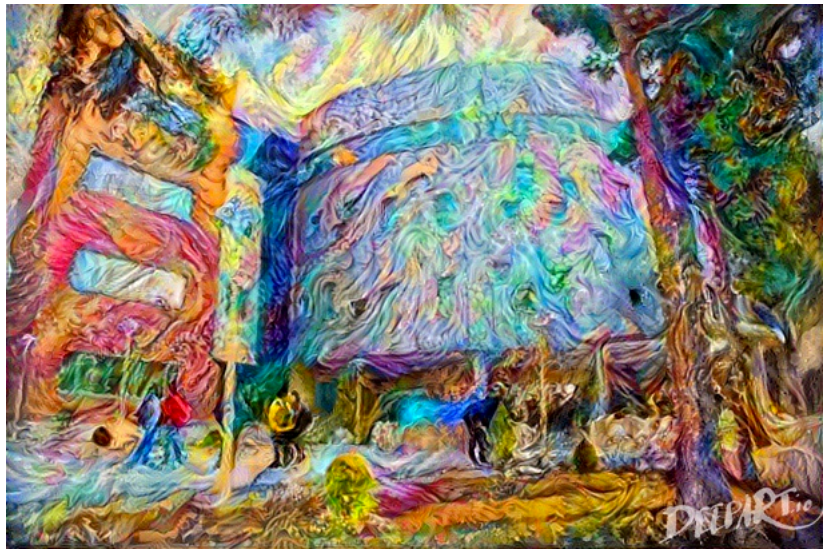


Getting Started with Neural Networks

Mohammad Taher Pilehvar

Introduction to Artificial Intelligence, 97

http://iust-courses.github.io/ai97/

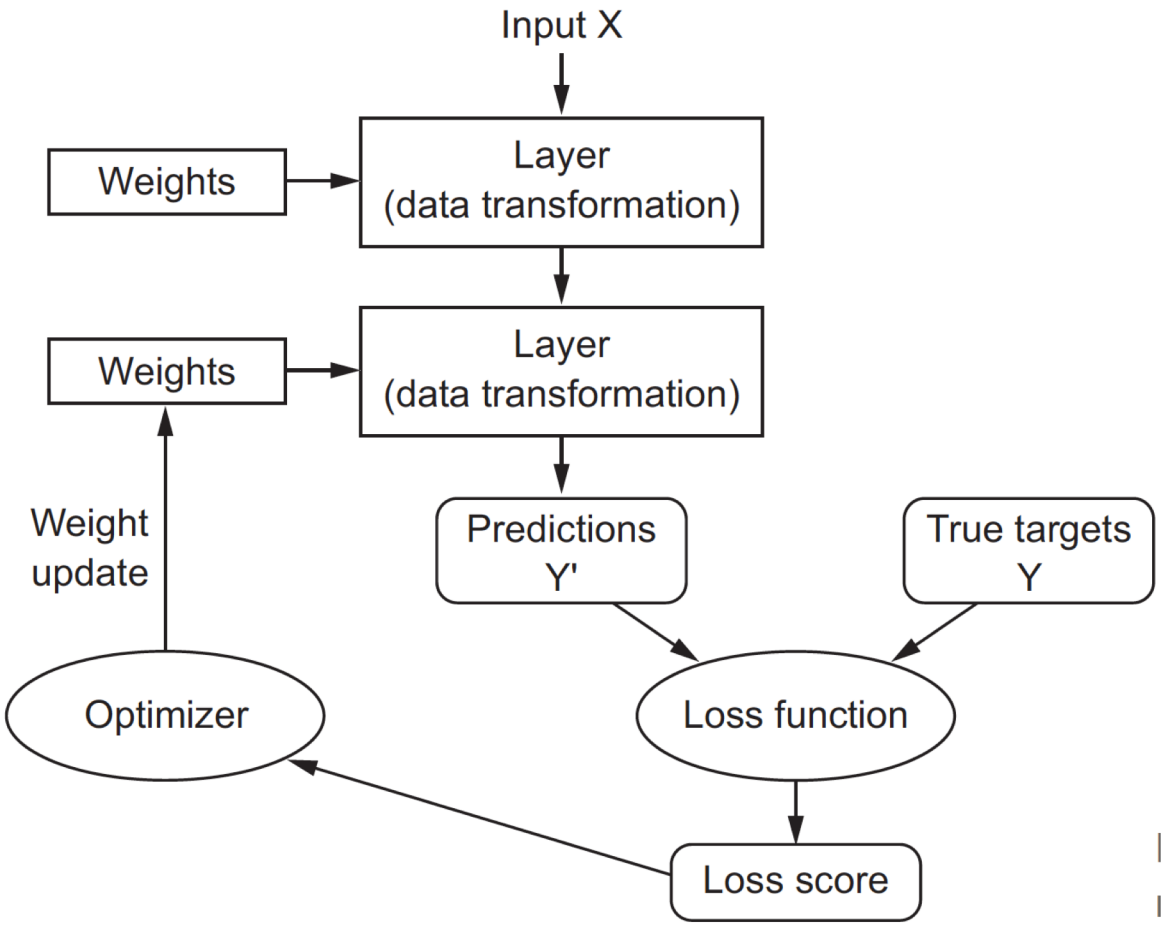
Today

- Core components of neural networks
- Introduction to Keras
- Neural networks for basic classification and regression

Anatomy of a neural network

- *Layers*, which are combined into a network (or model)
- The *input data* and corresponding targets
- The *loss* function, which defines the feedback signal used for learning
- The *optimizer*, which determines how learning proceeds

Anatomy of a neural network



Layers

- A layer is a data-processing module that takes as input one or more tensors and that outputs one or more tensors.

- Different layers for different types of data:
  - *Dense or fully connected:* simple vector data, stored in 2D tensors of shape (samples, features
  - *Recurrent:* sequence data, stored in 3D tensors of shape (samples, timesteps, features)
  - *Convolution:* image data, stored in 4D tensors

Layers: compatibility

- A dense layer with 32 output units

```
from keras import layers
layer = layers.Dense(32, input_shape=(784,))
```

- Accepts input 2D tensors only, with dimension 784
  - Batch dimension is unspecified; any value is accepted

- This layer can be connected to a layer that accepts inputs of dimension 32

Layers: compatibility

- Keras automatically takes care of compatibility

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_shape=(784,)))
model.add(layers.Dense(48))
```

  - No need to mention the `input_shape` for the second layer
  - It is automatically inferred

Network of layers

- A deep-learning model is a directed, acyclic graph of layers.
- Remember, in machine learning:
  - we are in search for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal.

- By choosing a network topology, you constrain your space of possibilities.

- Then we search for a good set of values for the weight tensors involved in these tensor operations

- Picking the right network architecture is more an art than a science!

Network of layers + *loss* function + optimizer

- *Loss function* (*objective function*): The quantity that will be minimized during training.
  - It represents a measure of success for the task at hand.
  - Very important; e.g., "maximizing the average well-being of all humans alive."

- *Optimizer*: Determines how the network will be updated based on the *loss* function.
  - Implements a specific variant of stochastic gradient descent.

Keras

https://keras.io

- CPU + GPU
- Easy and quick for prototyping deep-learning models
- Supports a wide range of architectures
- Very popular, 200K users
  - Google, Netflix, Uber, CERN, Yelp, Square, Kaggle

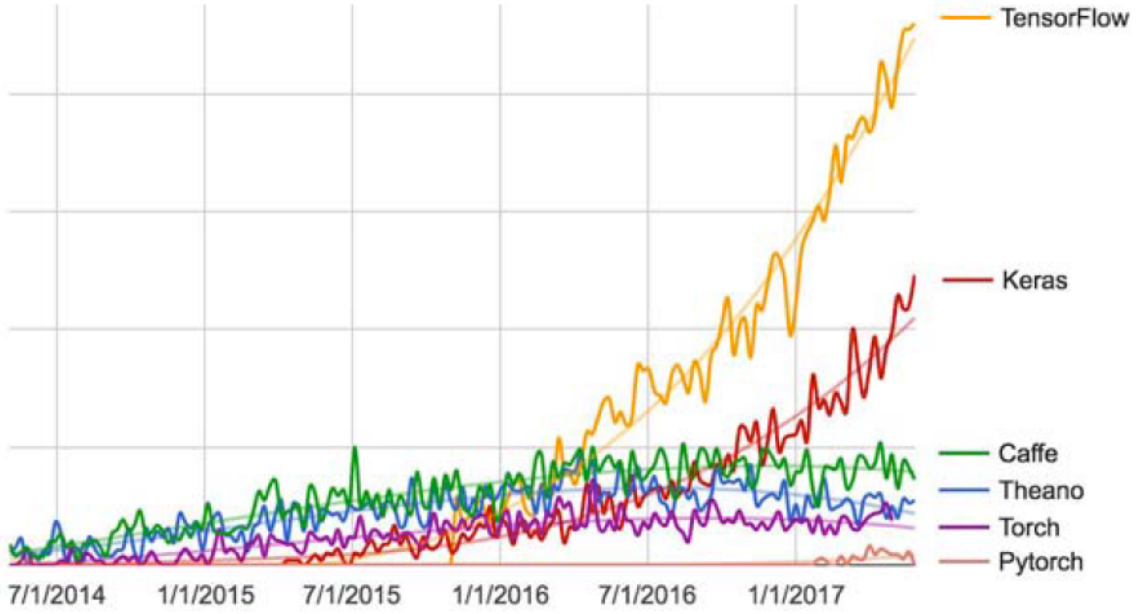
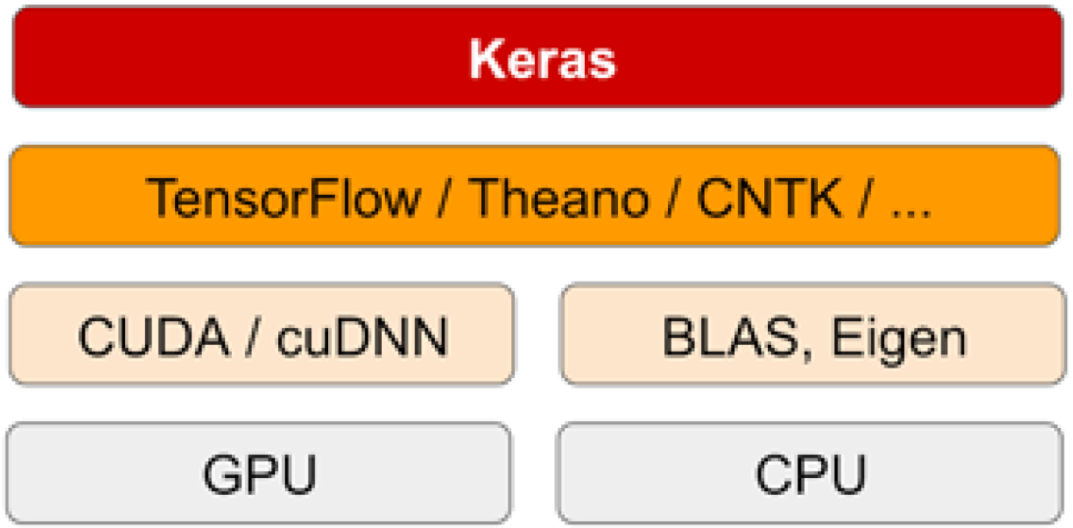


Figure 3.2 Google web search interest for different deep-learning frameworks over time

Deep learning frameworks

- TensorFlow
  - Google
- Microsoft Cognitive Toolkit - CNTK
- Theano
  - MILA lab at Université de Montréal
- PyTorch
  - Facebook

Deep learning frameworks



Keras: workflow

1. Define your training *data*: input tensors and target tensors.
2. Define a network of *layers* (or model) that maps your inputs to your targets.
3. Configure the learning process by choosing a *loss* function, an *optimizer*, and some *metrics* to monitor.
4. Iterate on your training data by calling the `fit()` method of your model.

Defining a model

- Sequential

```
model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax’))
```

- Funcional API

```
input_tensor = layers.Input(shape=(784,))
x = layers.Dense(32, activation='relu')(input_tensor)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = models.Model(inputs=input_tensor, outputs=output_tensor)
```

Compilation and training

- Compilation: specify the *optimizer* and *loss* function(s) that the model should use, as well as the *metrics* you want to monitor during training.

```
model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

- Learning:
  - pass Numpy arrays of input data and targets

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Setting up Keras

- Preferably a GPU

- Unix: highly recommended!
  - Ubuntu is a good distribution

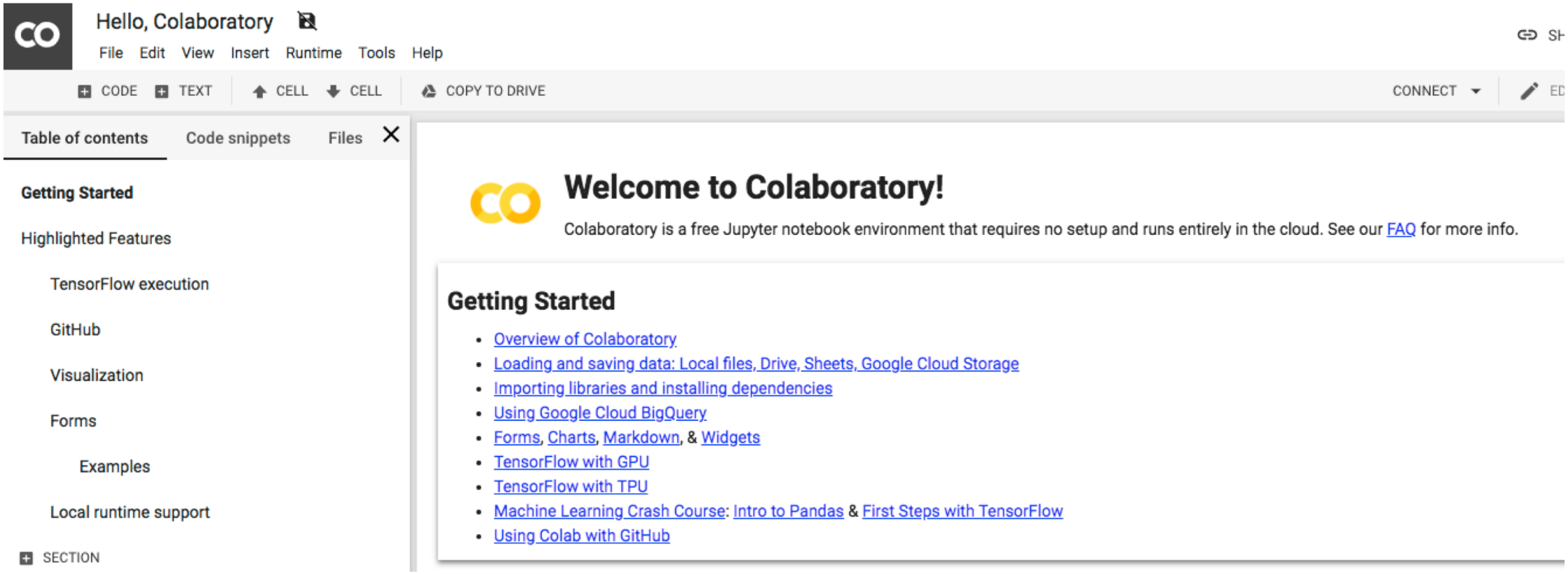
- IDE
  - PyCharm
  - Jupyter notebook

Jupyter notebooks

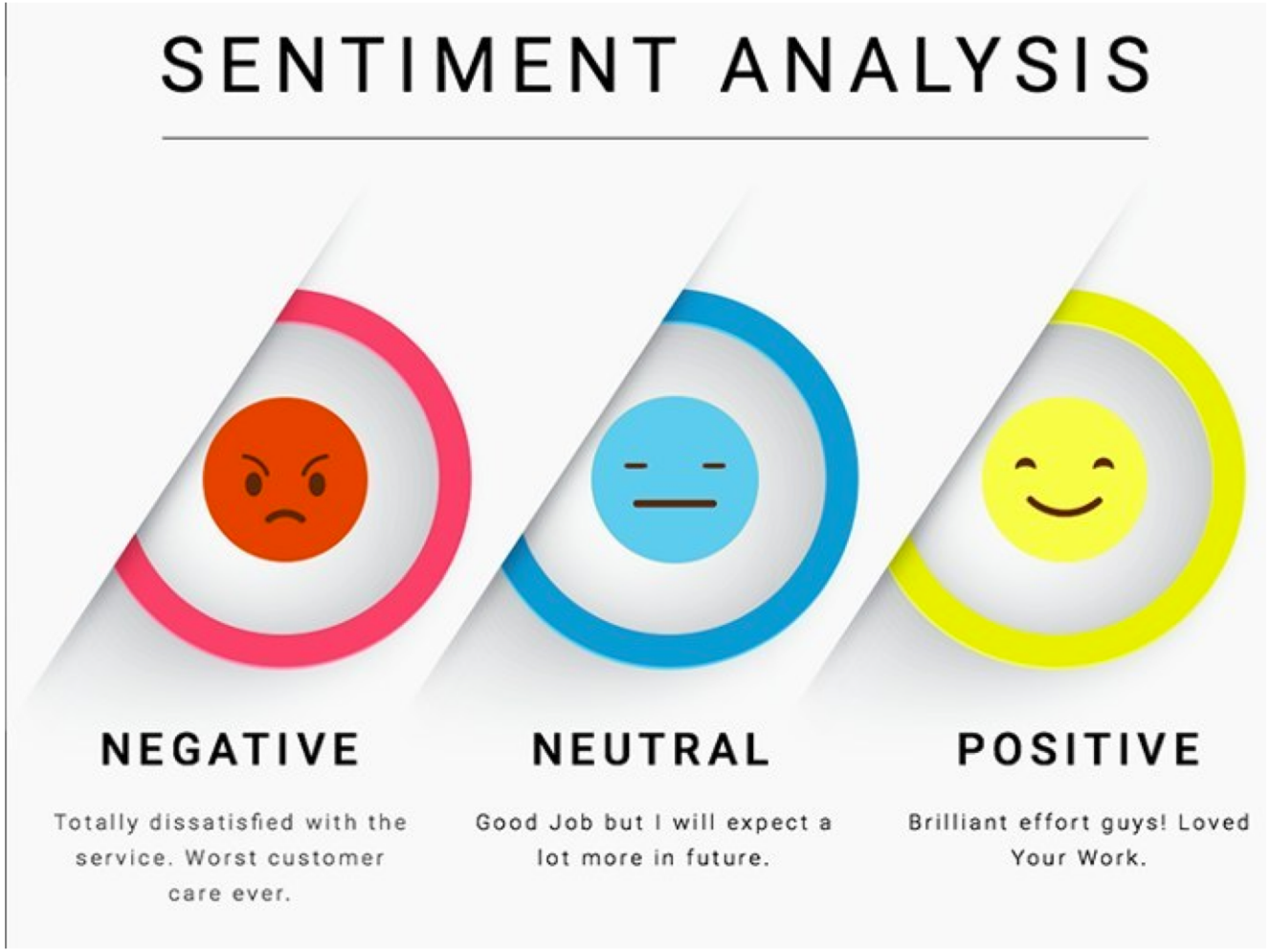
https://jupyter.org

- Break up long experiments into smaller pieces that can be executed independently
- Annotate your code
- Highly interactive

Google Colaboratory



Exercise 1: Classifying movie reviews



Exercise 1: Classifying movie reviews

- IMDB dataset
- 50,000 highly polarized reviews
- 25K training, 25K test, each 50-50%
  - Why use separate training and test sets?

- IMDB comes preloaded with Keras

```
from keras.datasets import imdb
(train_data, train_labels), (test_data, test_labels) =
             imdb.load_data(num_words=10000)
```

IMDB dataset

- The variables `train_data` and `test_data` are lists of reviews
- Each review is a list of word indices (encoding a sequence of words).
- `train_labels` and `test_labels` are lists of 0s and 1s, where 0 stands for *negative* and 1 stands for *positive*:

```
>>> train_data[0]
[1, 14, 22, 16, ... 178, 32]
>>> train_labels[0]
1

>>> max([max(sequence) for sequence in train_data])
9999
```

IMDB dataset

- Check one review instance:

```
word_index = imdb.get_word_index()
reverse_word_index = dict(
      [(value, key) for (key, value) in word_index.items()])
decoded_review = ' '.join(
      [reverse_word_index.get(i - 3, '?') for i in train_data[0]])
```

Preparing data

- You can't feed lists of integers into a neural network
  - We need tensors!

- Two options:
  1. Use `Embedding` layer
     - Same length: pad your sentences.

  2. One-hot encoding
     - Vectors of 0's and 1's
       - turning the sequence [3, 5] into a 10,000-dimensional vector that would be all 0s except for indices 3 and 5, which would be 1s
     - Use `Dense` layer

Preparing data (one-hot encoding)

```
import numpy as np
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)
```

```
>>> x_train[0]
array([ 0., 1., 1., ..., 0., 0., 0.])

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

Building the network

- The input data is vectors, and the labels are scalars (1s and 0s)

- The argument being passed to each `Dense` layer (16) is the number of hidden units of the layer.

  A hidden unit is a dimension in the representation space of the layer

Dense layer

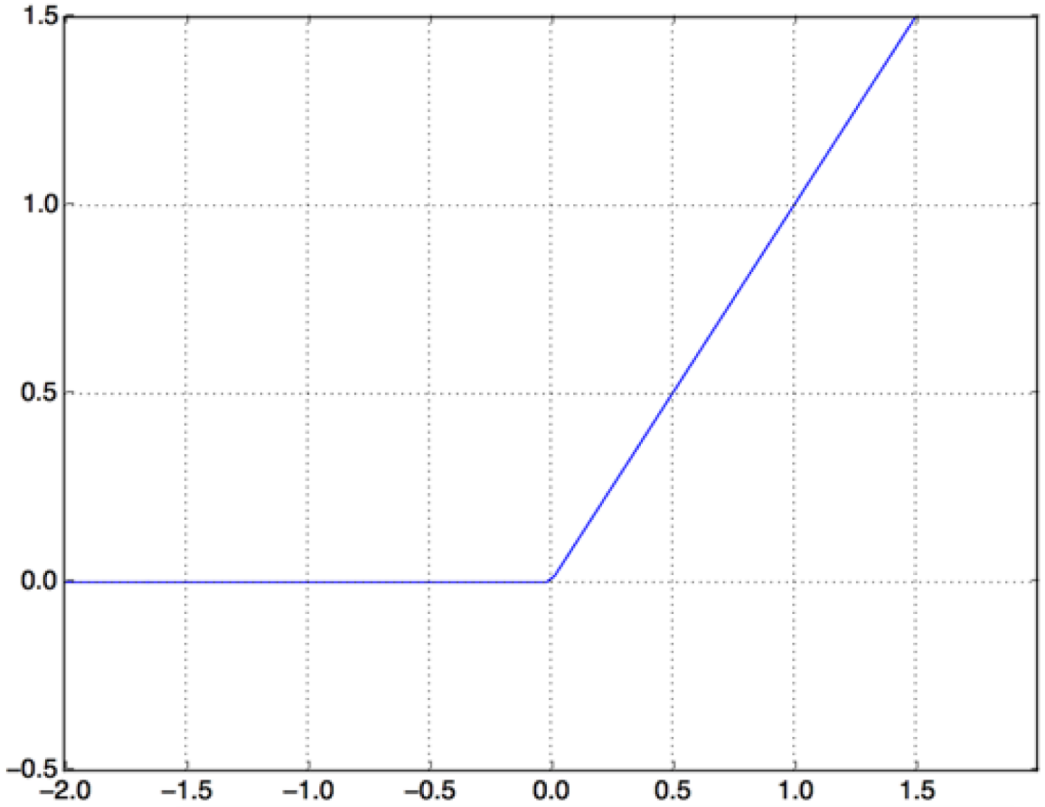
```
output = relu(dot(W, input) + b)
```

- 16 hidden units: the weight matrix W will have shape `(input_dimension, 16)`

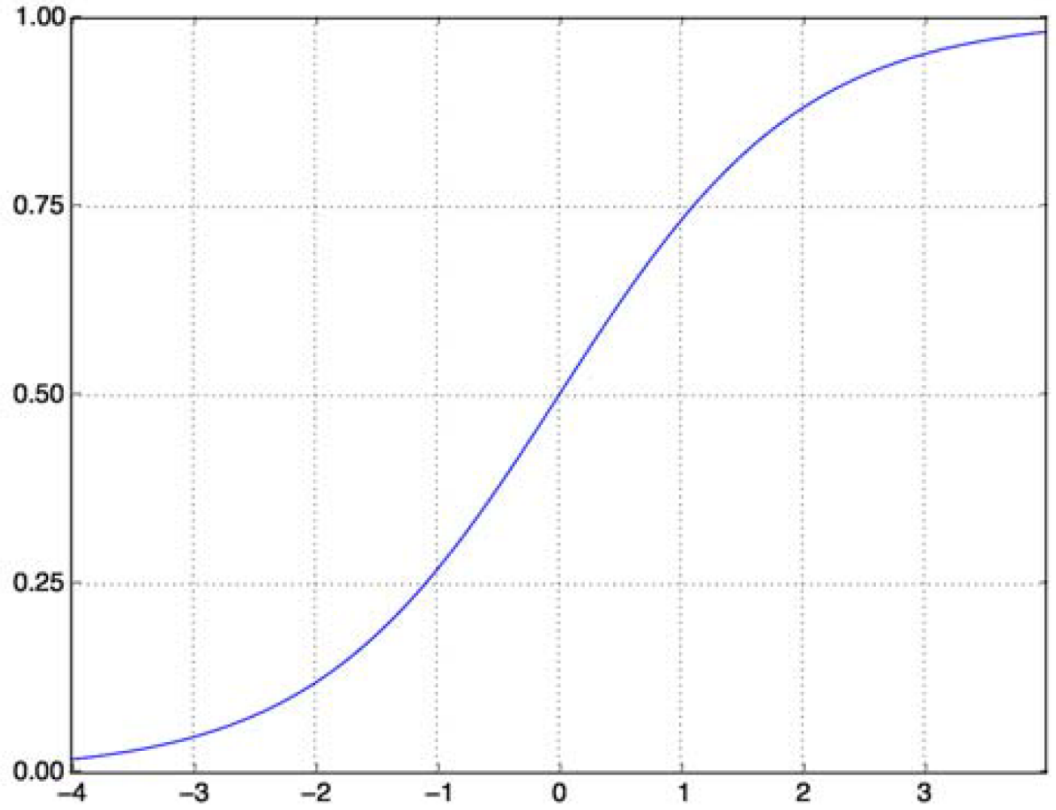
- The dot product with W will project the input data onto a 16-dimensional representation space

- Two decisions:
  - How many layers to use?
  - How many hidden units to choose for each layer?

ReLU and Sigmoid activation function



zero out negative values



“squashes” values into the [0, 1] interval

Activation function

- Without an activation function, the Dense layer would consist of two linear operations—a dot product and an addition:

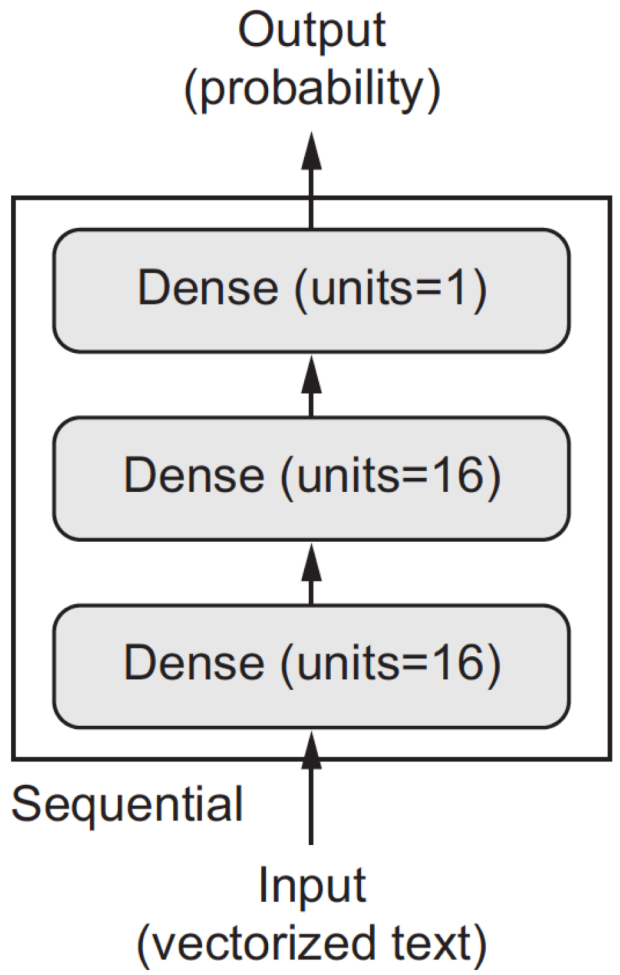
```
output = dot(W, input) + b
```

- It can only learn linear transformations of the input data
  - Too restricted and wouldn't benefit from multiple layers of representations

- You need a non-linearity
  - In order to get access to a much richer hypothesis space

Network architecture

```
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'))
```



Loss function and optimizer

- Binary classification
  - `binary_crossentropy`
  - `mean_squared_error`

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

- `Optimizer`
  - `rmsprop`

- `You can customize:`

```
model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy’,
              metrics=['accuracy'])
```

Train the model (monitor on development set)

Monitor during training the accuracy of the model on data it has never seen before

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]
y_val = y_train[:10000]
partial_y_train = y_train[10000:]

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

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

results = model.evaluate(x_test, y_test)
```

Training history

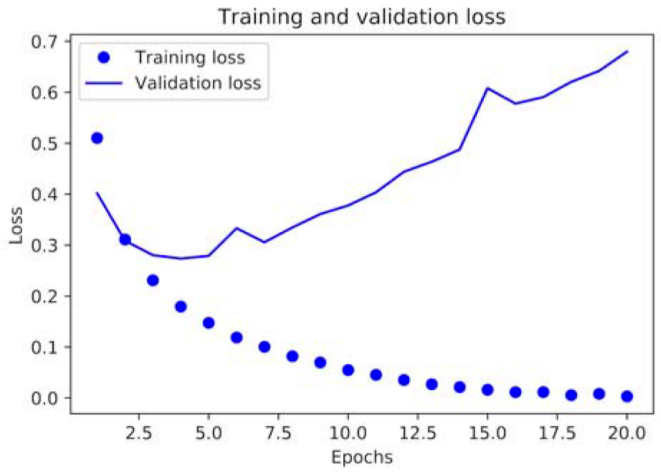
`model.fit()` returns a `History` object

- Contains data about everything that happened during training

```
>>> history_dict = history.history
>>> history_dict.keys()
[u'acc', u'loss', u'val_acc', u'val_loss']
```

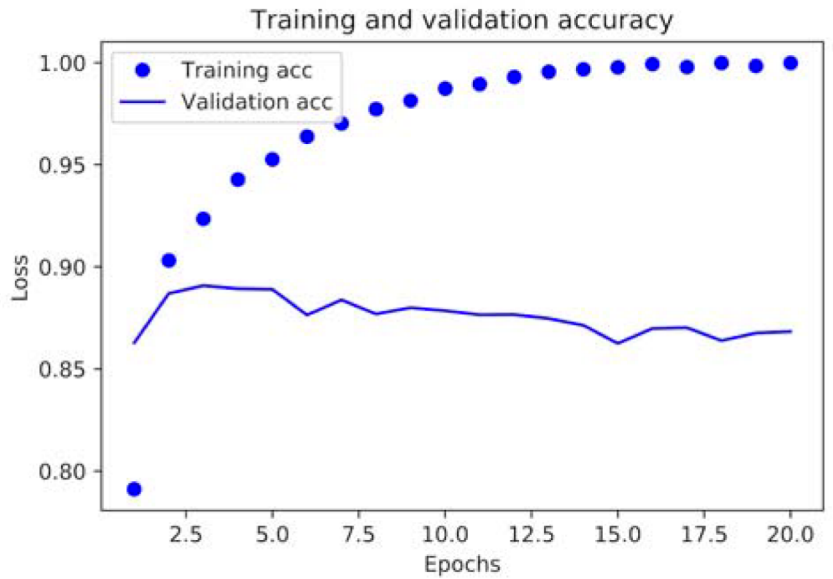
Plot the training and validation loss

```
import matplotlib.pyplot as plt

history_dict = history.history
loss_values = history_dict['loss']
val_loss_values = history_dict['val_loss’]

epochs = range(1, len(acc) + 1)

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

plt.show()
```



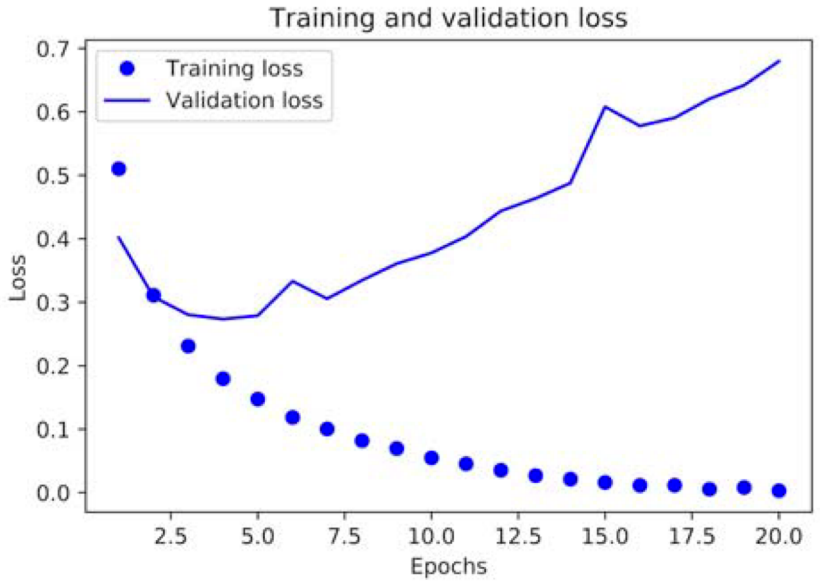
Plot the training and validation accuracy

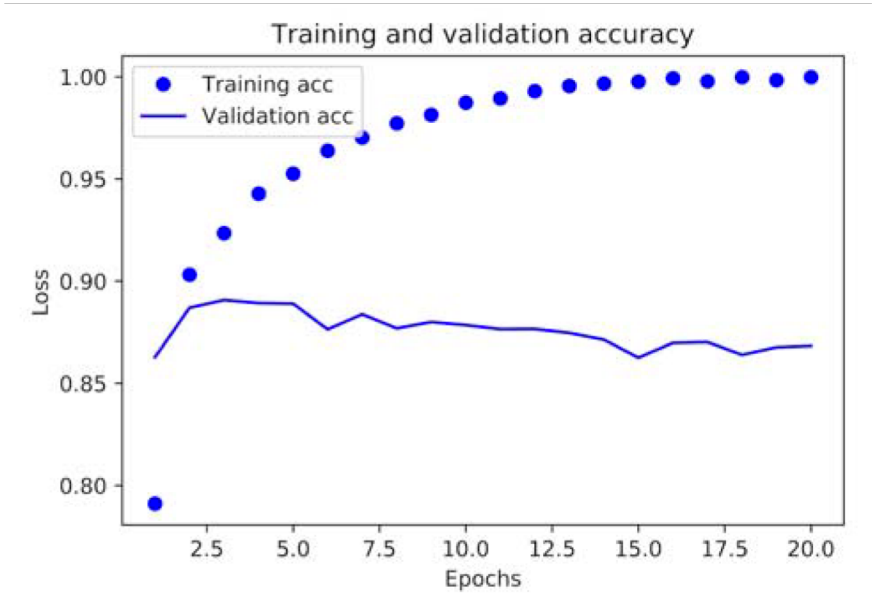
```
plt.clf()
acc_values = history_dict['acc']
val_acc_values = history_dict['val_acc’]

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

plt.show()
```



Overfitting

- A model that performs better on the training data isn't necessarily a model that will do better on data it has never seen before
- After the second epoch, you're overoptimizing on the training data





Generate predictions on new data

```
>>> model.predict(x_test)
array([[ 0.98006207]
       [ 0.99758697]
       [ 0.99975556]
       ...,
       [ 0.82167041]
       [ 0.02885115]
       [ 0.65371346]], dtype=float32)
```

Further experiments

- Try using one or three hidden layers
- Try using layers with more hidden units or fewer hidden units
- Try using the `mse` loss function instead of `binary_crossentropy`
- Try using the `tanh` activation instead of `relu`

Wrap up: Classifying movie reviews

- Preprocessing of raw input data
- Stack of `Dense` layers
- In a binary classification, your network should end with a `Dense` layer with one unit and a sigmoid activation
  - The output of your network should be a scalar between 0 and 1, encoding a probability.
- Optimizer and loss function
- Importance of avoiding overfitting

Exercise II: Classifying newswire

- Multiclass classification problem

- Classify Reuters newswires into 46 mutually exclusive topics
  - *Single-label, multiclass*

Loading the data

```
from keras.datasets import reuters
(train_data, train_labels), (test_data, test_labels) =
                            reuters.load_data(num_words=10000)
```

```
>>> len(train_data)
8982
>>> len(test_data)
2246

>>> train_data[10]
[1, 245, 273, 207, 156, 53, 74, 160, 26, 14, 46, 296, 26, 39, 74, 2979,
3554, 14, 46, 4689, 4329, 86, 61, 3499, 4795, 14, 61, 451, 4329, 17, 12]
```

Checking the data

```
word_index = reuters.get_word_index()
reverse_word_index = dict([(value, key) for (key, value) in
word_index.items()])
decoded_newswire = ' '.join([reverse_word_index.get(i - 3, '?') for i in
      train_data[0]])
```

- Labels are integers [0, 45]

```
>>> train_labels[10]
3
```

Vectorize the data

- Same as last exercise:

```
import numpy as np

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)
```

Vectorize the labels

- Integer tensor or one-hot encoding

```
def to_one_hot(labels, dimension=46):
      results = np.zeros((len(labels), dimension))
      for i, label in enumerate(labels):
            results[i, label] = 1.
      return results

one_hot_train_labels = to_one_hot(train_labels)
one_hot_test_labels = to_one_hot(test_labels)
```

- There is a built-in function in Keras to do this:

```
from keras.utils.np_utils import to_categorical
one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Build the network

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

Two modifications

- You end the network with a `Dense` layer of size 46

- The last layer uses a *softmax* activation.
  - The network will output a probability distribution over the 46 different output classes
  - For every input sample, the network will produce a 46-dimensional output vector, where output[i] is the probability that the sample belongs to class i.
  - The 46 scores will sum to 1.

Loss function

`categorical_crossentropy`

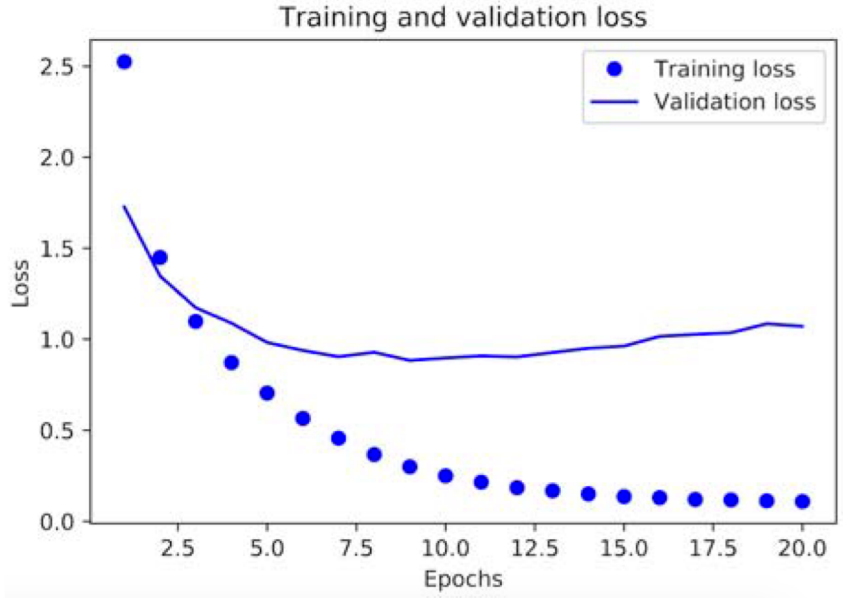
- Measures the distance between two probability distributions:
  - between the probability distribution output by the network and the true distribution of the labels

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

Train the model

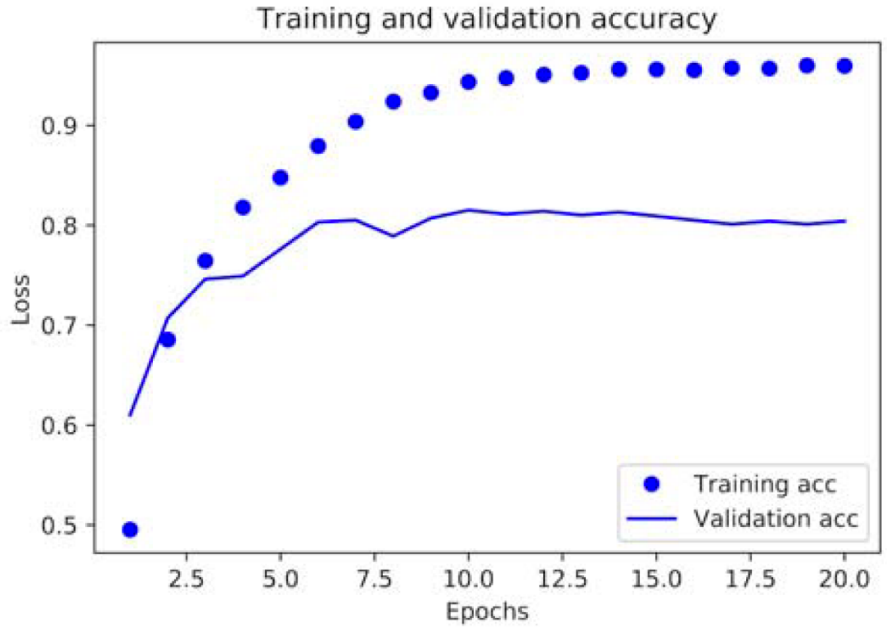
```
x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]

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

Plot loss




```
import matplotlib.pyplot as plt

loss = history.history['loss']
val_loss = history.history['val_loss’]

epochs = range(1, len(loss) + 1)

plt.plot(epochs, 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()

plt.show()
```

Plot accuracy




```
plt.clf()

acc = history.history['acc']
val_acc = history.history['val_acc’]

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

plt.show()
```

Retrain from scratch

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax’))

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

model.fit(partial_x_train, partial_y_train,
              epochs=9,
              batch_size=512,
              validation_data=(x_val, y_val))

results = model.evaluate(x_test, one_hot_test_labels)


>>> results
[0.956521343144807, 0.7969723953695458]
```

Labels as integer tensors?

- Only need to change loss function:

`sparse_categorical_crossentropy`

```
y_train = np.array(train_labels)
y_test = np.array(test_labels)

model.compile(optimizer='rmsprop',
loss='sparse_categorical_crossentropy', metrics=['acc'])
```

- Mathematically the same as categorical_crossentropy; it just has a different interface

Information bottleneck

- Intermediate layers should not be significantly smaller than the final one (46)
- For instance, having a 4-dimensional intermediate layer would reduce the accuracy from 79% to 71%
- The network is unable to cram all necessary information into this representation

Wrap up: multiclass classification

- If you're trying to classify data points among N classes, your network should end with a `Dense` layer of size N.
- In a single-label, multiclass classification problem, your network should end with a *softmax* activation
  - It will output a probability distribution over the N output classes.
- Two choices for handling labels:
  - One-hot encoding with `categorical_crossentropy` loss
  - Integer encoding with `sparse_categorical_crossentropy` loss
- Avoid information bottlenecks

Exercise 3: predict house prices

Regression problem

- predicting a continuous value instead of a discrete label

- Boston Housing Price dataset
  - Relatively small: 404 training, 102 test
  - Features of data have different scales
  - Targets are the median values of owner-occupied homes

```
from keras.datasets import boston_housing
(train_data, train_targets),
(test_data, test_targets)
      = boston_housing.load_data()
```

```
>>> train_data.shape
(404, 13)
>>> test_data.shape
(102, 13)
>>> train_targets
[ 15.2, 42.3, 50. ... 19.4, 19.4, 29.1]
```

Preparing the data

- Problematic to feed values that all take wildly different ranges
  - Makes learning more difficult

- Solution:
  - Feature-wise Normalization
    - Subtract the mean of the feature and divide by the standard deviation
    - Feature is centered around 0 and has a unit standard deviation

```
mean = train_data.mean(axis=0)
train_data -= mean
std = train_data.std(axis=0)
train_data /= std
test_data -= mean
test_data /= std
```

The model

- Small data, small network

```
from keras import models
from keras import layers
def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu’,
                           input_shape=(train_data.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(1))
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
    return model
```

Notes

- The network ends with a single unit and no activation
  - Activation constrains the range of output values, e.g., a network with sigmoid learns to generate values in [0,1]

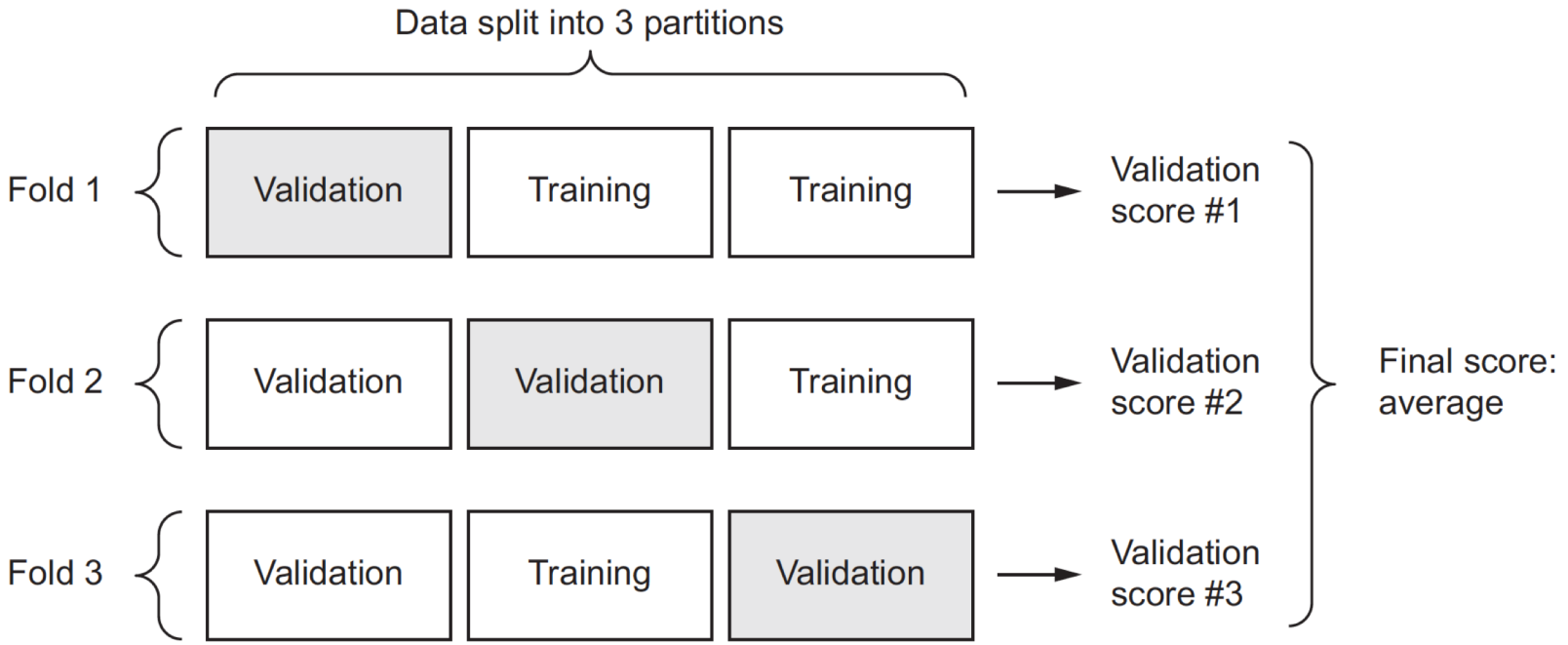
- Loss:
  - Mean squared error
    - The square of the difference between the predictions and the targets

- Evaluation metric:
  - Mean absolute eror (MAE)

    It's the absolute value of the difference between the predictions and the targets.

K-fold cross validation

For small datasets, the validation scores might have a high variance with regard to the validation split



K-fold cross validation

```
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate([train_data[:i * num_val_samples],
              train_data[(i + 1) * num_val_samples:]], axis=0)

    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
       train_targets[(i + 1) * num_val_samples:]], axis=0)

    model = build_model()
    model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)
    val_mse, val_mae = model.evaluate(val_data, val_targets, verbose=0)
    all_scores.append(val_mae)
```

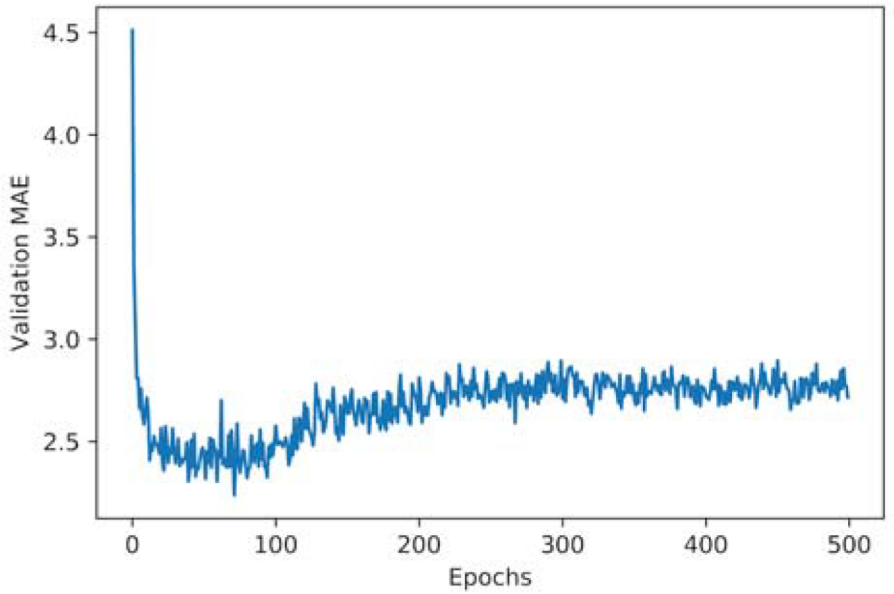
K-fold cross validation – plot

```
for i in range(k):
    print('processing fold #', i)
    val_data = train_data[i * num_val_samples: (i + 1) * num_val_samples]
    val_targets = train_targets[i * num_val_samples: (i + 1) * num_val_samples]

    partial_train_data = np.concatenate([train_data[:i * num_val_samples],
              train_data[(i + 1) * num_val_samples:]], axis=0)

    partial_train_targets = np.concatenate([train_targets[:i * num_val_samples],
       train_targets[(i + 1) * num_val_samples:]], axis=0)

    model = build_model()
    history = model.fit(partial_train_data, partial_train_targets,
              epochs=num_epochs, batch_size=1, verbose=0)

    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

K-fold cross validation – plot (II)

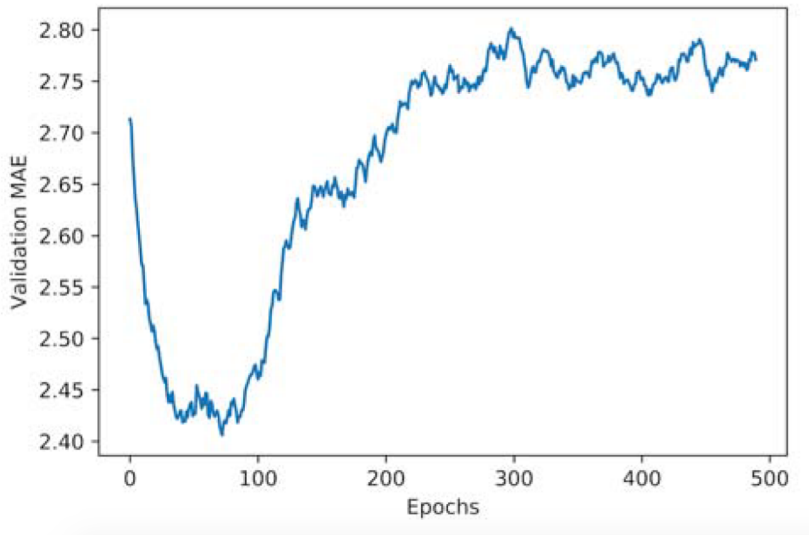
```
average_mae_history = [
      np.mean([x[i] for x in all_mae_histories]) for i in range(num_epochs)]
```

```
import matplotlib.pyplot as plt

plt.plot(range(1, len(average_mae_history) + 1),
                    average_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



K-fold cross validation – plot (III)

```
def smooth_curve(points, factor=0.9):
      smoothed_points = []
      for point in points:
            if smoothed_points:
                  previous = smoothed_points[-1]
                  smoothed_points.append
      (previous * factor + point * (1 - factor))
            else:
                  smoothed_points.append(point)
      return smoothed_points

smooth_mae_history = smooth_curve(average_mae_history[10:])

plt.plot(range(1, len(smooth_mae_history) + 1), smooth_mae_history)
plt.xlabel('Epochs')
plt.ylabel('Validation MAE')
plt.show()
```



Wrap up: Regression exercise

- Regression is done using different loss functions, e.g., MSE

- Evaluation metrics differ as well
  - No more *accuracy*
  - MAE
- Feature normalization
- Little data
  - Small network
  - K-fold validation

- Questions?