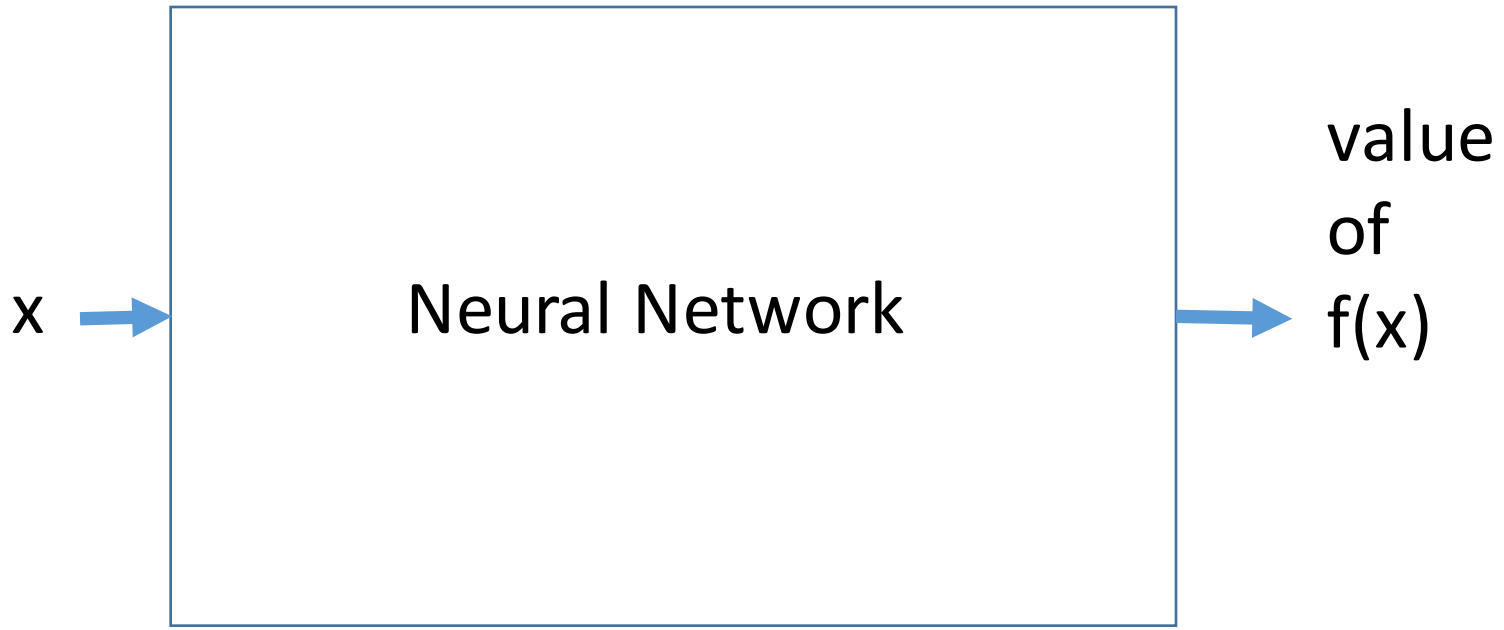
# CSCE 636 Neural Networks (Deep Learning)

Lecture 4: Getting Started with Neural Networks

Anxiao (Andrew) Jiang

# What a neural network does: learn a function

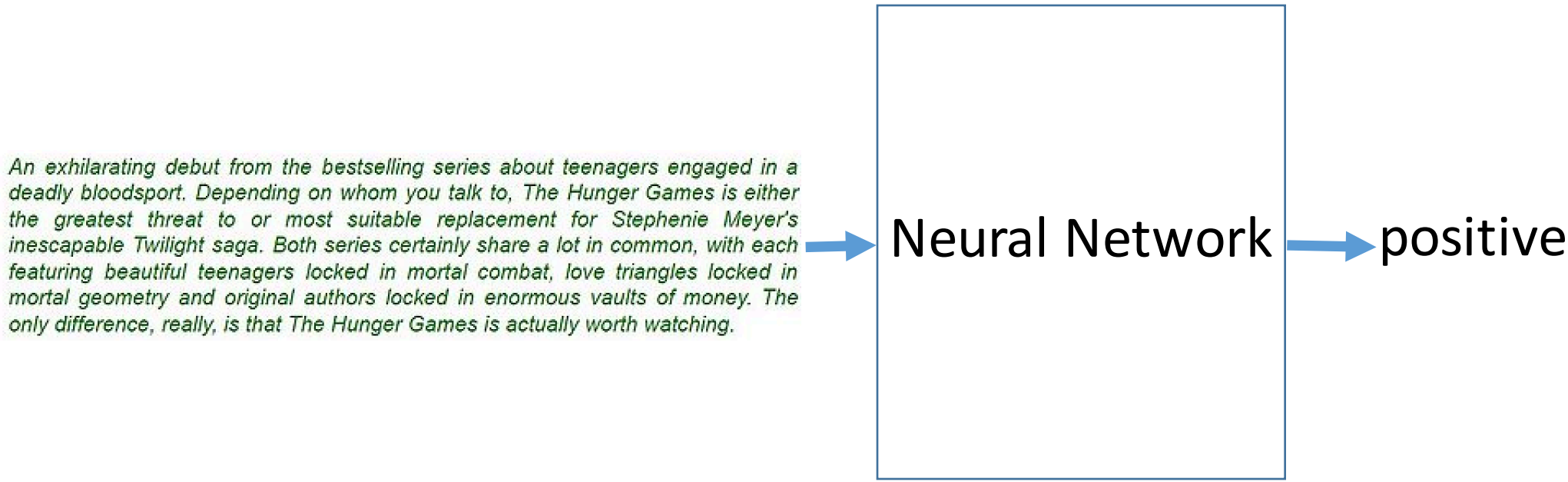



The neural network learns the function f(x), either exactly or approximately.

# Binary Classification

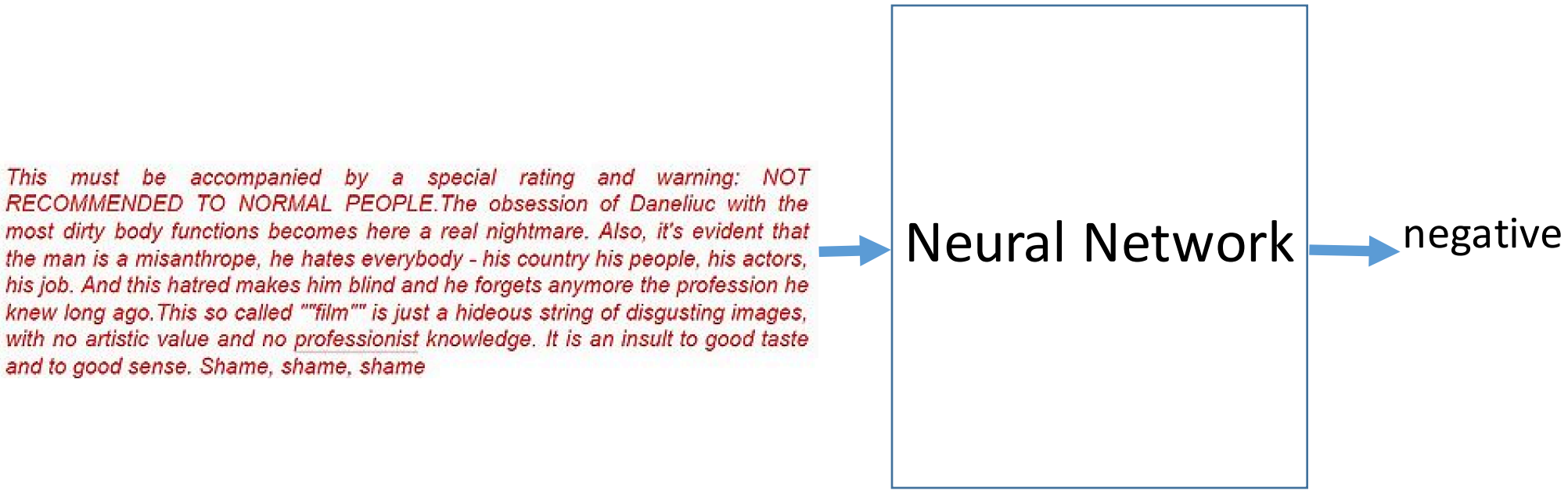
# Application: Classifying Movie Reviews

Task: Classify a movie review as positive or negative.

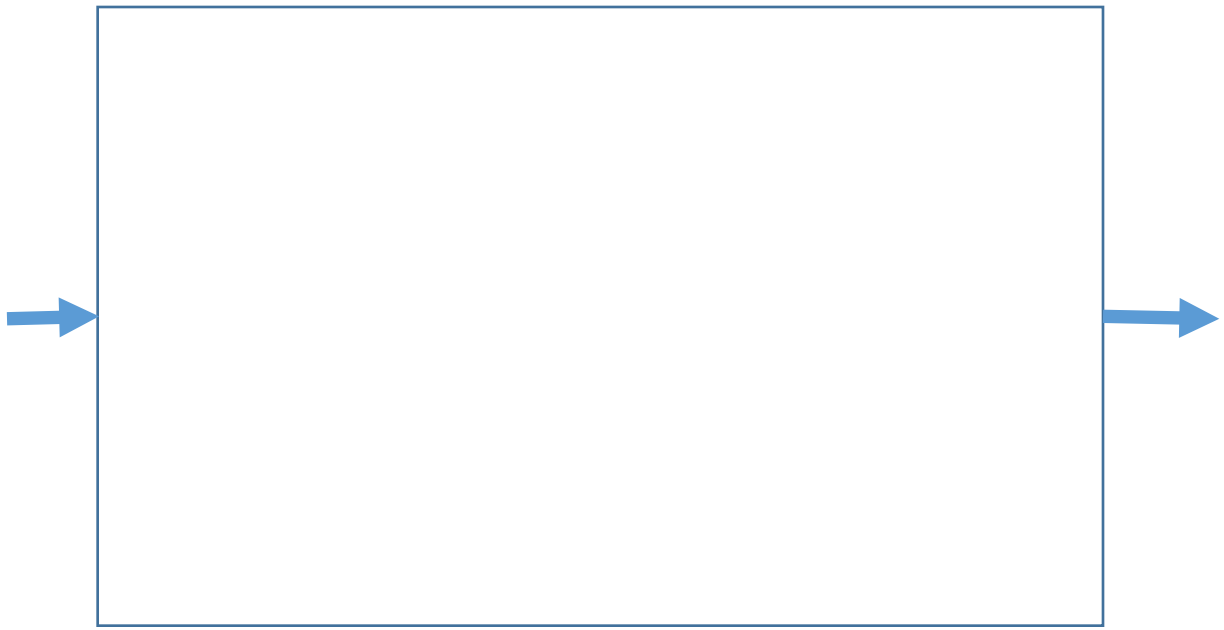


# Application: Classifying Movie Reviews

Task: Classify a movie review as positive or negative.



# How to start?



# Step 1: Load the dataset

IMDB Dataset: 50,000 high polarized reviews from Internet Movie Database, along with their "positive/negative" labels.

The partition: 25,000 reviews in training set, 25,000 reviews in test set. Training set and test set both have 50% negative and 50% positive reviews.

# Step 1: Load the dataset

**Listing 3.1 Loading the IMDB dataset**

```
from keras.datasets import imdb

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

Only keep the 10,000 most frequent words

train_data and test_data: Each is a list of 25,000 reviews; each review is a list of word indices (encoding a sequence of words) in [0,9999].

Shape of data: 2-dimensional.

train_labels and test_labels: Each is a list of 25,000 labels (of value 0 or 1).

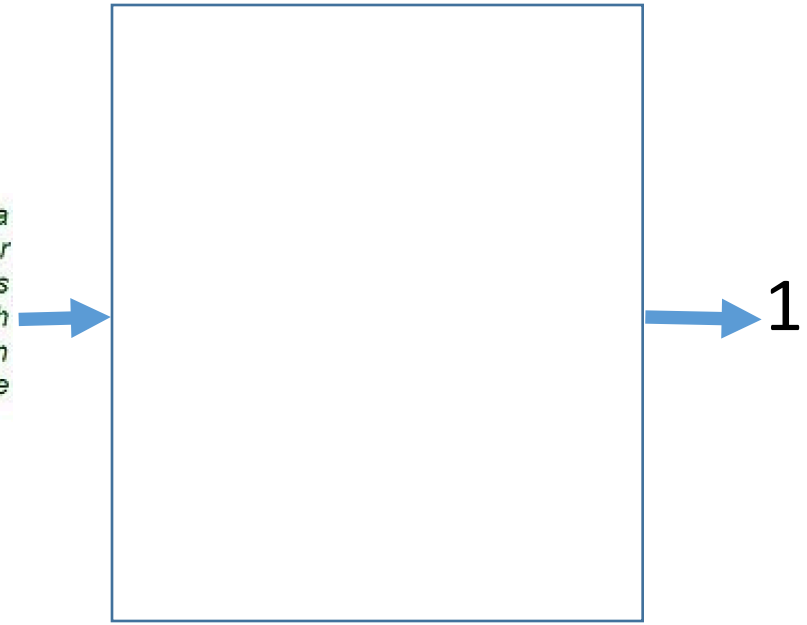
0 stands for: negative

1 stands for: positive

Shape of data: 1-dimensional.

# IMDB dataset

*An exhilarating debut from the bestselling series about teenagers engaged in a deadly bloodsport. Depending on whom you talk to, The Hunger Games is either the greatest threat to or most suitable replacement for Stephenie Meyer's inescapable Twilight saga. Both series certainly share a lot in common, with each featuring beautiful teenagers locked in mortal combat, love triangles locked in mortal geometry and original authors locked in enormous vaults of money. The only difference, really, is that The Hunger Games is actually worth watching.*



# Step 2: Prepare the data

- One-hot encode the lists (of train_data and test_data) to turn them into vectors of 0s and 1.

- For example: turn the review [3,5] (as a sequence of word indices) into a 10,000-dimensional vector that are all 0s except for indices 3 and 5 (which would be 1s).

- Note: the order of words in the review is lost. This is model is called "bag of words" model.

One-hot encode train_data and test_data

**Listing 3.2 Encoding the integer sequences into a binary matrix**

```
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)
```

Creates an all-zero matrix of shape (len(sequences), dimension)

Sets specific indices of results[i] to 1s

Vectorized training data

Vectorized test data

x_train and x_test: 2-dimensional tensor (array) of shape 25000 x 10000.
Dimension 0: number of samples.
Dimension 1: size of vocabulary.

Turn train_labels and test_labels from lists into arrays (as a data structure type)

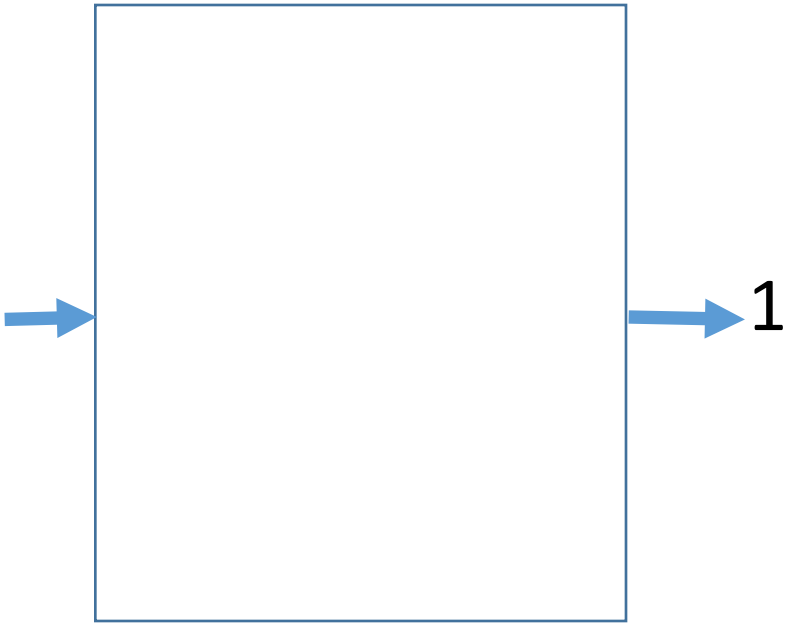
```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

y_train and y_test: 1-dimensional tensor (array) of shape (25000, ).
Dimension 0: number of samples.

# IMDB dataset

*An exhilarating debut from the bestselling series about teenagers engaged in a deadly bloodsport. Depending on whom you talk to, The Hunger Games is either the greatest threat to or most suitable replacement for Stephenie Meyer's inescapable Twilight saga. Both series certainly share a lot in common, with each featuring beautiful teenagers locked in mortal combat, love triangles locked in mortal geometry and original authors locked in enormous vaults of money. The only difference, really, is that The Hunger Games is actually worth watching.*

As one-hot encoded vector



# Step 3: Build neural network

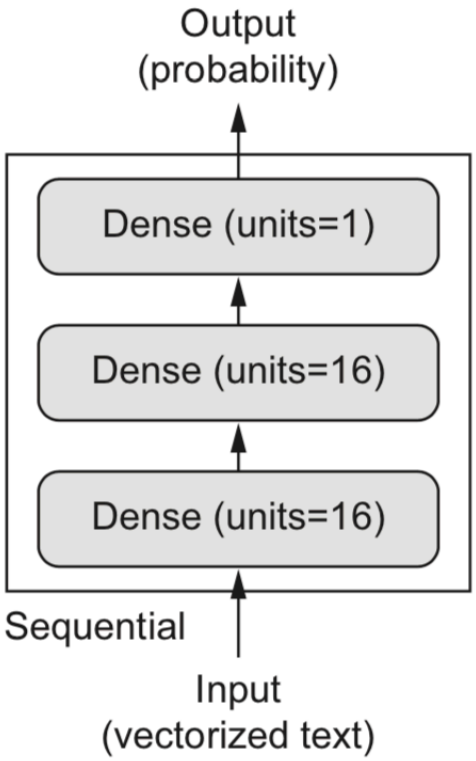
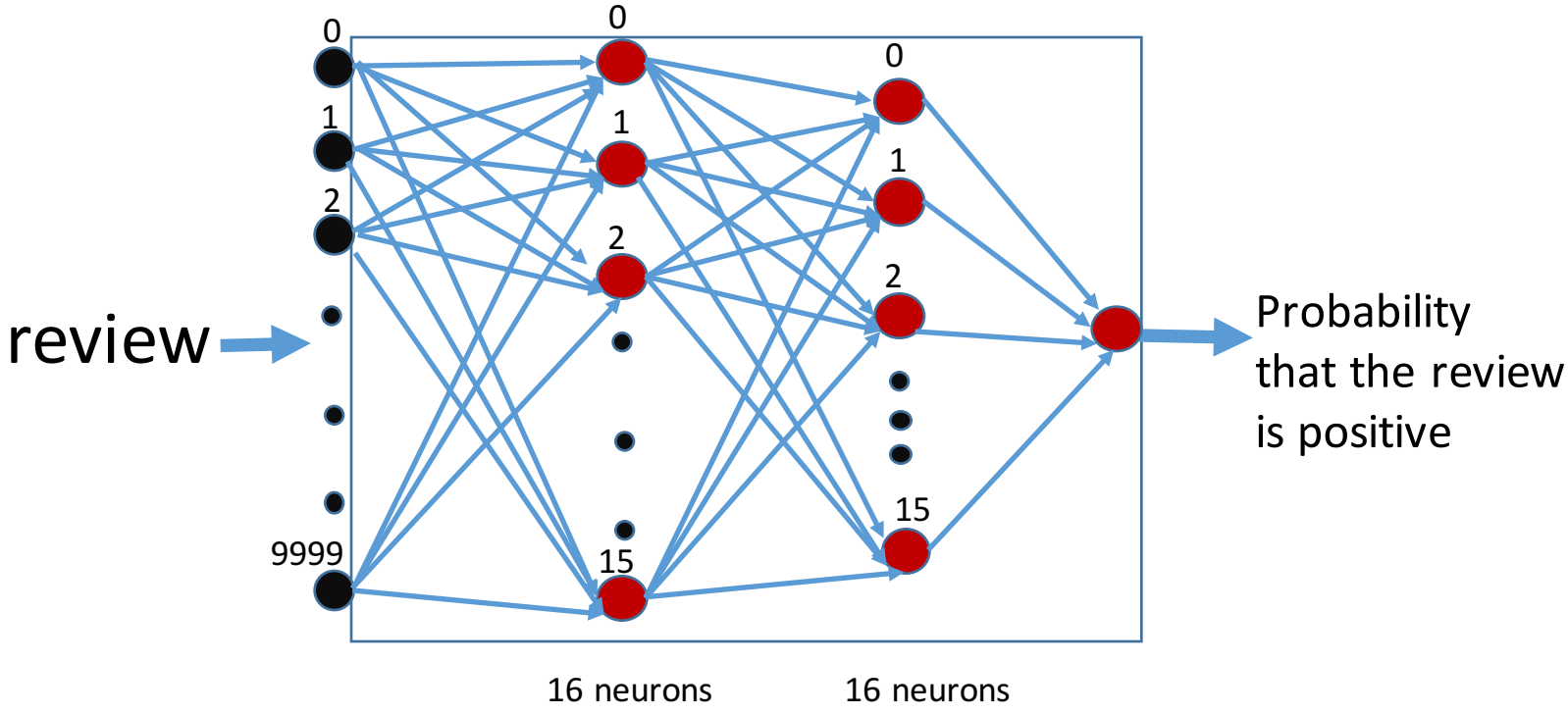



**Figure 3.6 The three-layer network**

**Listing 3.3 The model definition**

```
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'))
```



# Activation function: ReLU

$$f(x) = x^+ = \max(0, x)$$

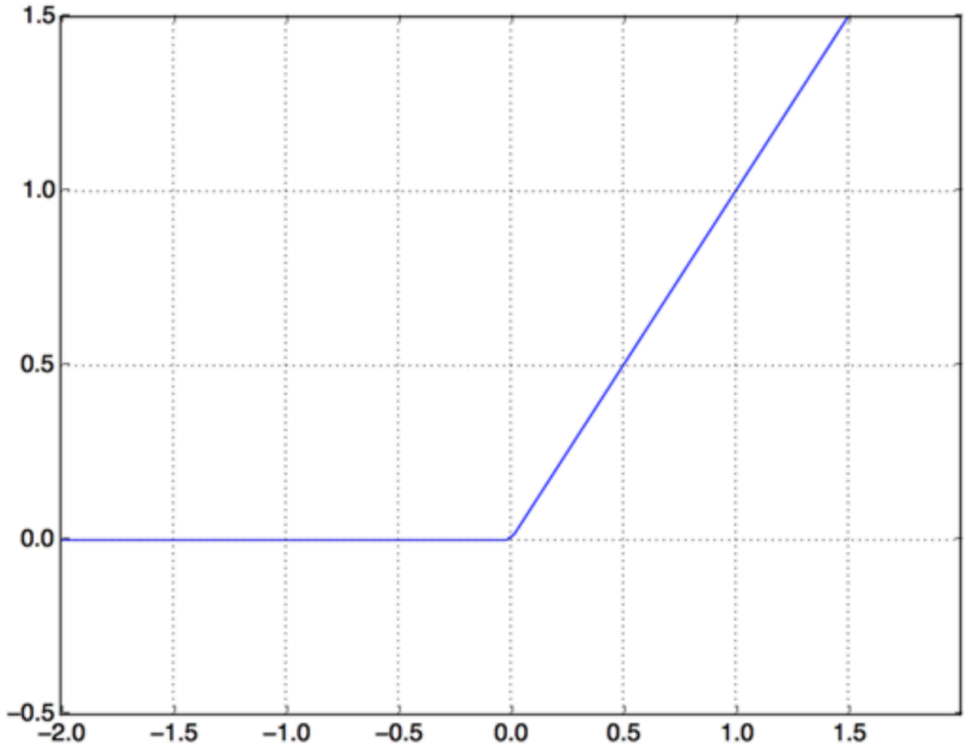



**Figure 3.4 The rectified linear unit function**

# Activation function: sigmoid

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

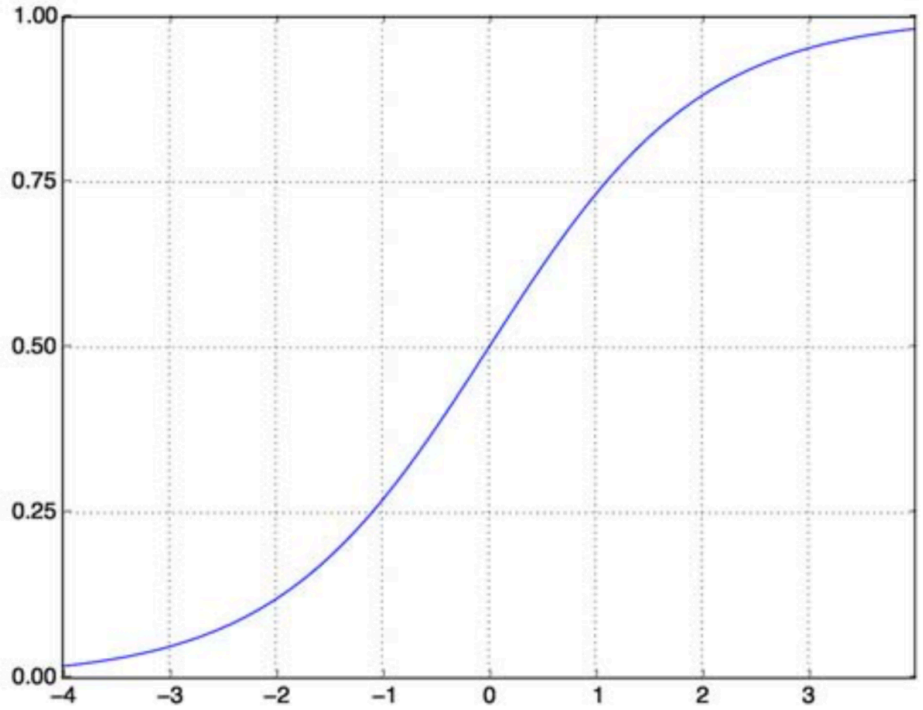
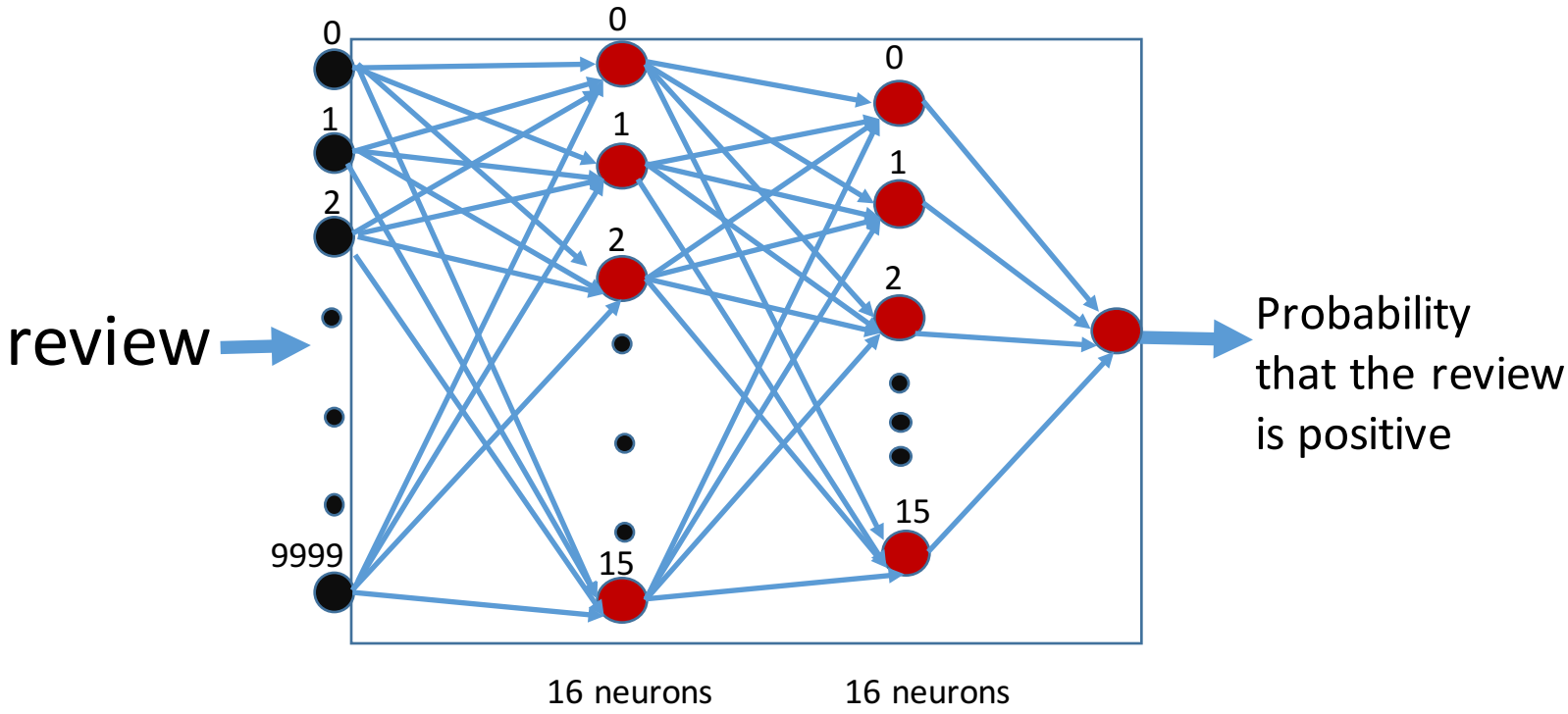



**Figure 3.5 The sigmoid function**


review
0
1
2
9999
0
1
2
15
0
1
2
15
Probability that the review is positive
16 neurons
16 neurons

# Step 3: choose loss function, optimizer, and target metrics

**Listing 3.4 Compiling the model**

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

# Binary cross-entropy

$$-(y \log(p) + (1 - y) \log(1 - p))$$

Correct Output Probability (0 or 1)

NN's Output Probability (between 0 and 1)

# Alternative codes

**Listing 3.5 Configuring the optimizer**

```
from keras import optimizers

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

Learning rate

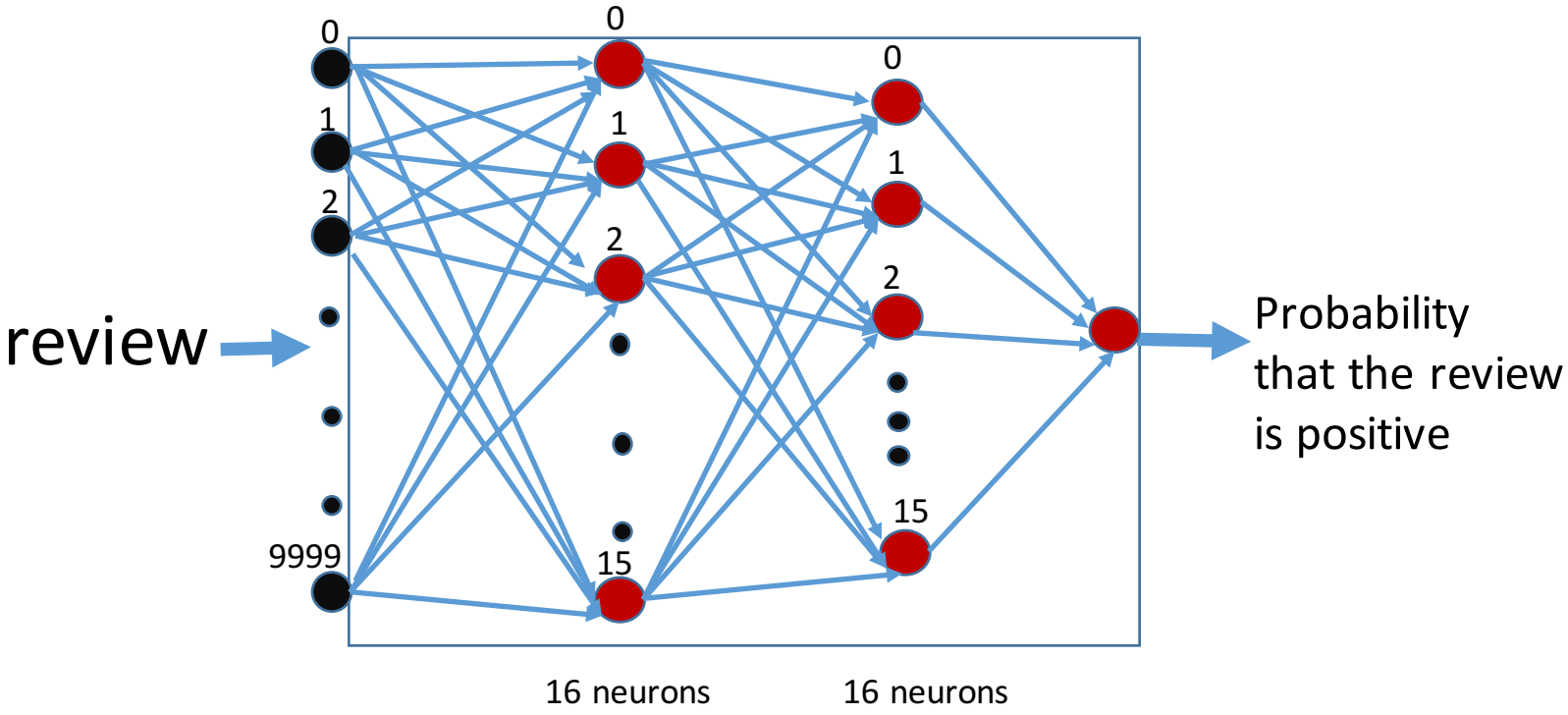
**Listing 3.6 Using custom losses and metrics**

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

The "Teacher":

Loss function: binary cross-entropy
Optimizer: RMSProp
Target Metric: Accuracy

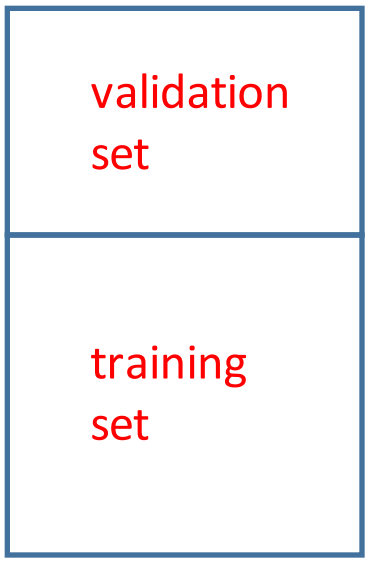


# Partition training set into “training set” and “validation set”

**Listing 3.7 Setting aside a validation set**

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

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




10,000 reviews and their labels.
Shape: 10000 x 10000

15,000 reviews and their labels.
Shape: 15000 x 10000

During training:

1) Use “training set” to train NN
2) Use “validation set” to monitor the performance of NN.

# Step 4: Train the neural network

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

# Step 4: Train the neural network

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

Note that the call to `model.fit()` returns a `History` object. This object has a member `history`, which is a dictionary containing data about everything that happened during training. Let's look at it:

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

Corresponding to every key in the dictionary, there is a list that records the accuracy or loss for every epoch.

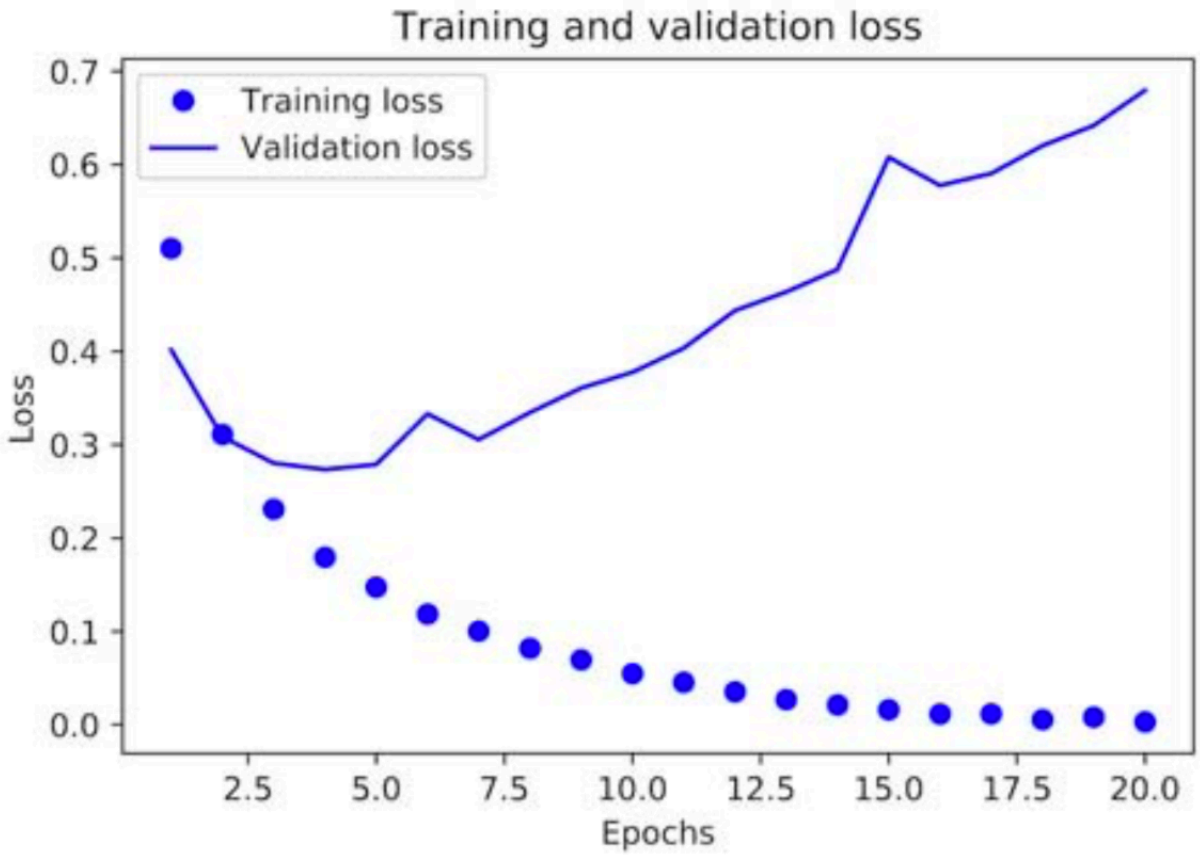



**Figure 3.7 Training and validation loss**

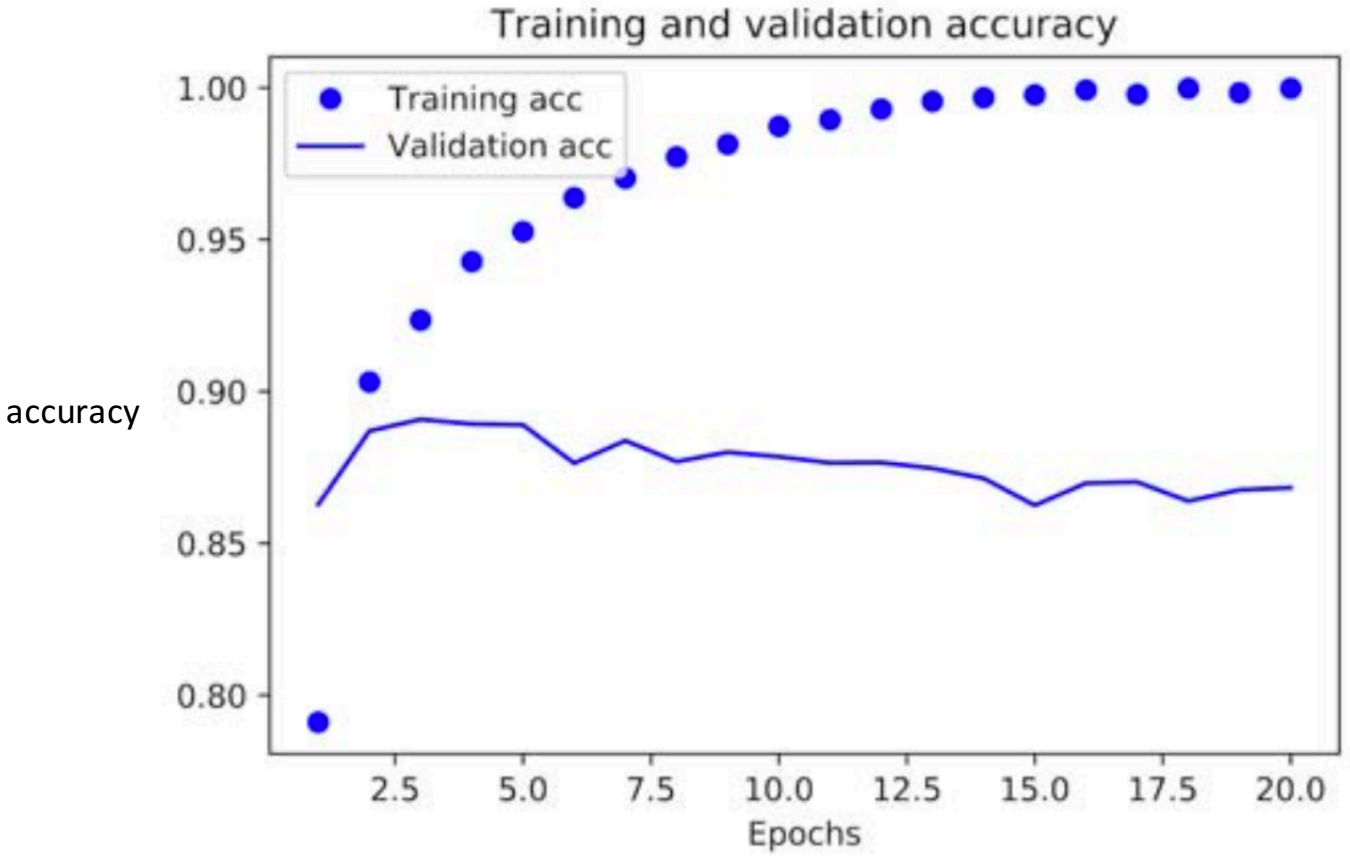



**Figure 3.8 Training and validation accuracy**

NN starts to overfit after about 4 epochs

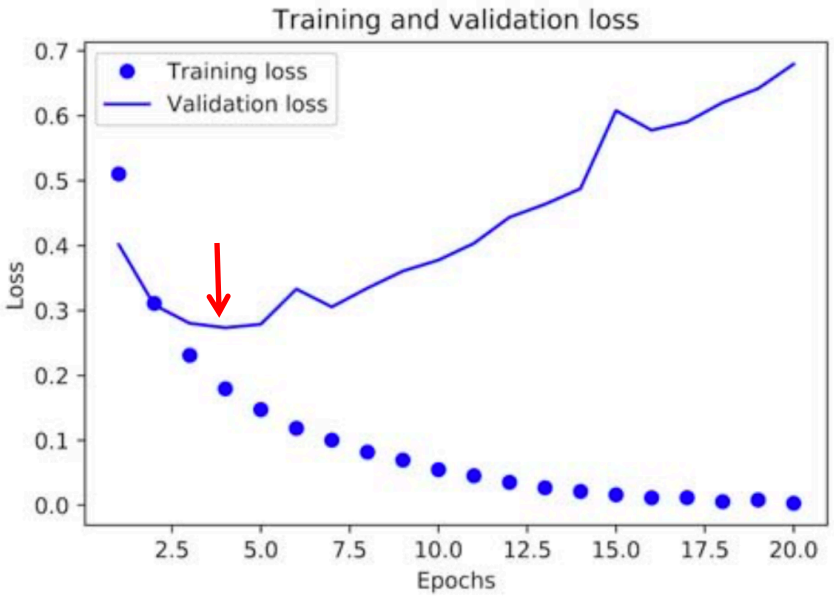



Figure 3.7 Training and validation loss

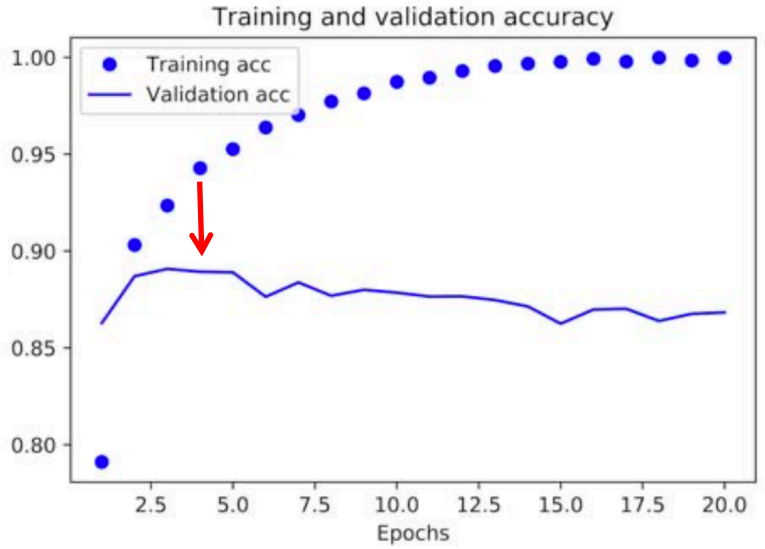



Figure 3.8 Training and validation accuracy

# Train a new network from scratch for 4 epochs, using all the training data

```
model.fit(x_train, y_train, epochs=4, batch_size=512)
```

We don't have to use validation set here.
We can use validation set as part of the training data, too.

# Step 5: Test the trained neural network

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

Show test results:

```
>>> results
[0.2929924130630493, 0.8832799999999995]
```

loss

Accuracy: about 88%

# Step 6: Use trained network for prediction