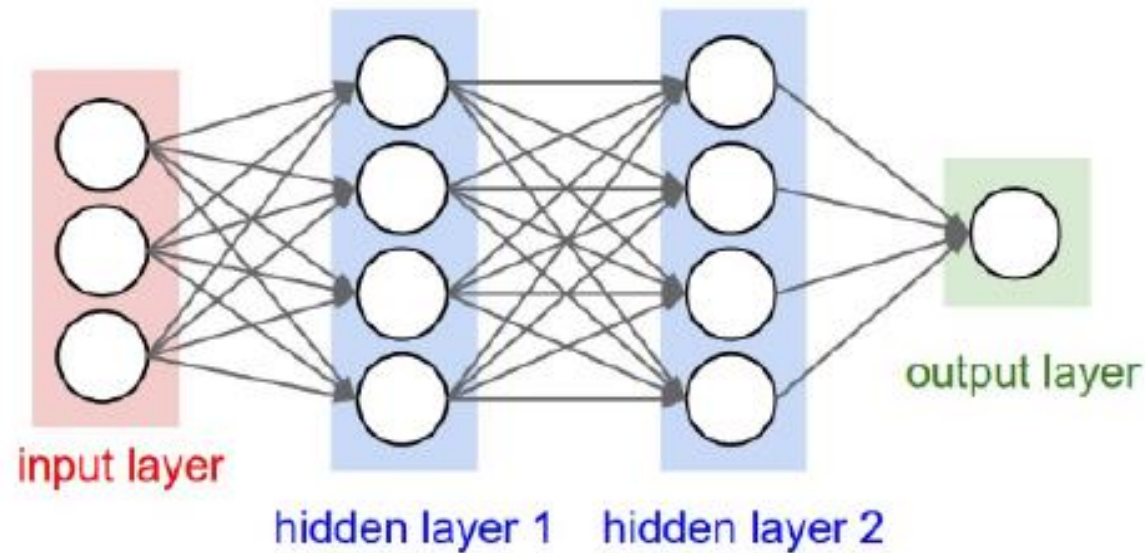


Introduction to Deep Learning

APT3025: APPLIED MACHINE LEARNING

Neural Networks

- An artificial neural network is a model learned from data, just like a decision tree is learned from data. A basic artificial neural network is depicted below.



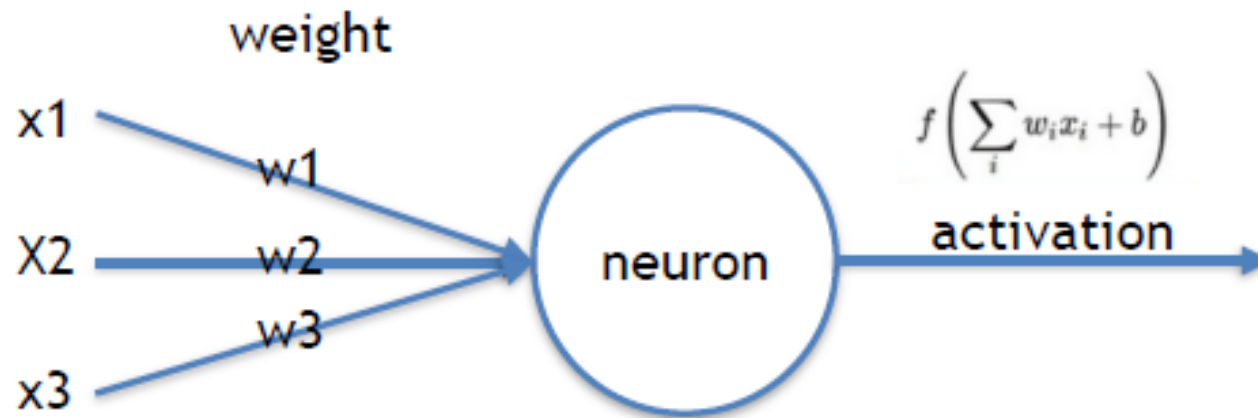
Deep learning is the modern term for artificial neural networks, which are deep, i.e., have many hidden layers

Learning in Neural Networks

- Forward pass: feed training data through the network, transforming each input in some way, and finally getting predictions at the output layer.
- Calculate the prediction error.
- Back propagation: pass the error back through the network and adjust each weight, guided by gradient descent, so that the error is reduced.
- Repeat until the network error is low enough.

Activation

- Each neuron takes the weighted sum of its inputs and applies to it a certain function, known as activation.
- If the result is greater than or equal to a certain threshold, the neuron outputs a 1. Otherwise it outputs a zero.



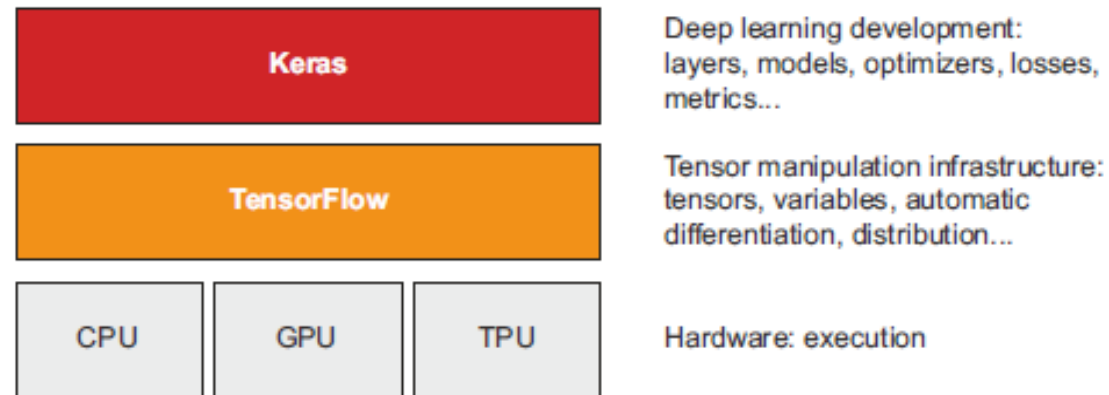
Classifying handwritten digits

- Let's look at an image classification problem.
- We will try to classify grayscale images of handwritten digits (28×28 pixels) into their 10 categories (0 through 9). We'll use the MNIST dataset, which is very well-known in machine learning.
- Here are some of the digits in MNIST.



Classifying handwritten digits

- We will use Keras, which is a wrapper for the low level TensorFlow deep learning library, both from Google.



```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Inspecting the data

- The images are encoded as NumPy arrays, and the labels are an array of digits, ranging from 0 to 9.

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)

>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

The Workflow

- First, we'll feed the neural network the training data, `train_images` and `train_labels`.
- The network will then learn to associate images and labels.
- Finally, we'll ask the network to produce predictions for `test_images`, and we'll verify whether these predictions match the labels from `test_labels`.

The Network Architecture

- Let's build the network.

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

The Network Architecture

- The core building block of neural networks is the layer. You can think of a layer as a filter for data: some data goes in, and it comes out in a more useful form.
- Specifically, layers extract representations out of the data fed into them—hopefully, representations that are more meaningful for the problem at hand.
- Most of deep learning consists of chaining together simple layers that will implement a form of progressive data distillation.

The Network Architecture

- Here, our model consists of a sequence of two Dense layers, which are densely connected (also called fully connected) neural layers.
- The second (and last) layer is a 10-way softmax classification layer, which means it will return an array of 10 probability scores (summing to 1).
- Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

Getting the network ready

- To make the model ready for training, we need to pick three more things as part of the compilation step:
- An optimizer—The mechanism through which the model will update itself based on the training data it sees, so as to improve its performance.
- A loss function—How the model will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction.
- Metrics to monitor during training and testing—Here, we'll only care about accuracy (the fraction of the images that were correctly classified)

Getting the data ready

- Before training, we'll preprocess the data by reshaping it into the shape the model expects and scaling it so that all values are in the $[0, 1]$ interval.
- Previously, our training images were stored in an array of shape (60000, 28, 28) of type uint8 with values in the $[0, 255]$ interval.
- We'll transform it into a float32 array of shape (60000, 28 * 28) with values between 0 and 1.

```
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
```

Training the model

- We're now ready to train the model, which in Keras is done via a call to the model's `fit()` method—we fit the model to its training data.
- Two quantities are displayed during training: the loss of the model over the training data, and the accuracy of the model over the training data. We quickly reach an accuracy of 0.989 (98.9%) on the training data.

```
>>> model.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [=====] - 5s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

Using the model to make predictions

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

Using the model to make predictions

- Each number of index i in that array corresponds to the probability that digit image `test_digits[0]` belongs to class i .
- This first test digit has the highest probability score (0.99999106, almost 1) at index 7, so according to our model, it must be a 7:

```
>>> predictions[0].argmax()  
7
```

```
>>> predictions[0][7]  
0.99999106
```

We can check that the test label agrees:

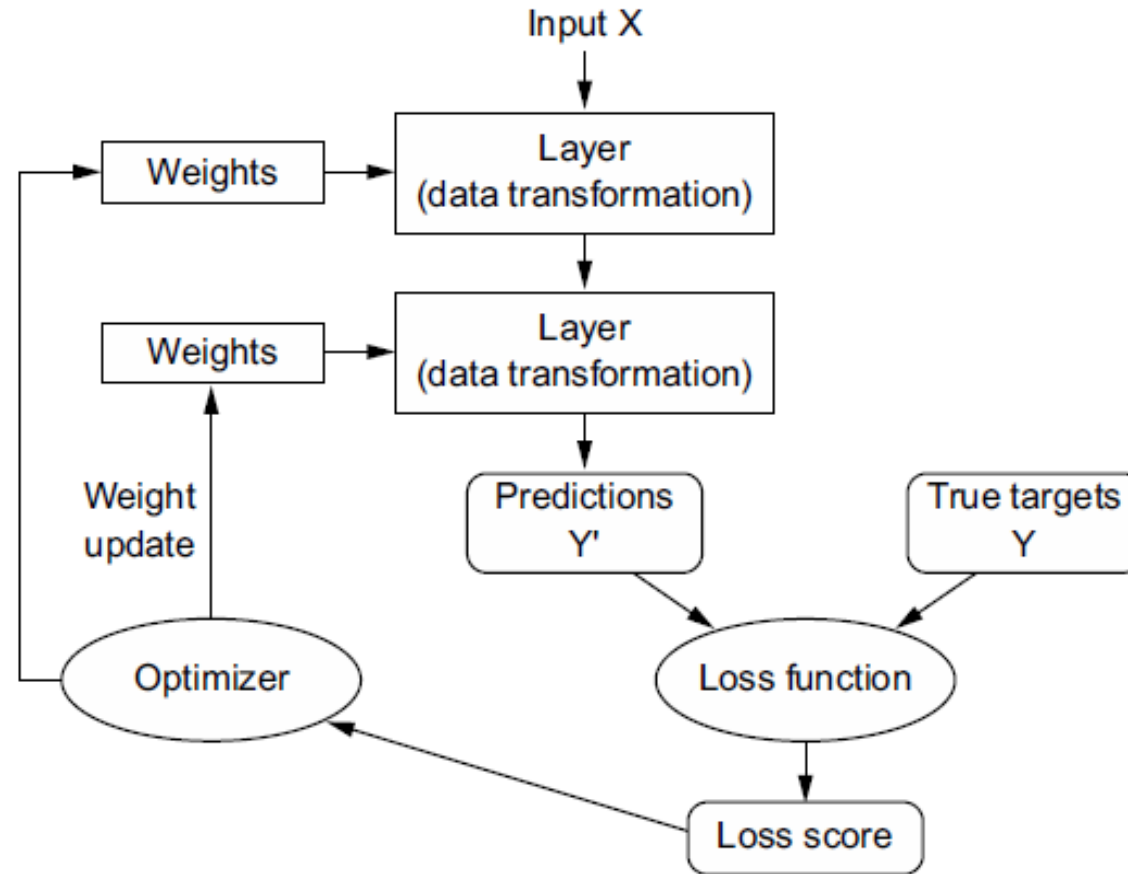
```
>>> test_labels[0]  
7
```


Evaluating the model on new data

- We can see that the model is slightly overfitting: 98.9% accuracy on the training data versus 97.85% accuracy on the test data.

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

Relationships between neural network components



Classifying movie reviews

- The IMDB dataset is a set of 50,000 reviews from the Internet Movie Database.
- They're split into 25,000 reviews for training and 25,000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.
- Like MNIST, IMDB is available from Keras.
- It has already been preprocessed: the reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

Loading the data

- The argument `num_words=10000` means you'll only keep the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This keeps the size manageable.
- Many of the discarded words only occur in a single sample, and thus can't be meaningfully used for classification.

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

Loading the data

- The problem is to learn a model from the data which can then be used to classify future reviews as either positive or negative.
- 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
```

Preparing the data

- You can't directly feed lists of integers into a neural network. They all have different lengths, but a neural network expects to process contiguous batches of data.
- One solution is to multi-hot encoding, which will convert a list (review) into a vector of zeros and ones.
- This would mean, for instance, turning the sequence [8, 5] into a 10,000-dimensional vector that would be all 0s except for indices 8 and 5, which would be 1s.

Multi-hot encoding the integer lists

```
import numpy as np
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        for j in sequence:
            results[i, j] = 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[0]
array([ 0.,  1.,  1., ...,  0.,  0.,  0.] )
```

Vectorize the labels

- The labels are integers and should be converted to floats:

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


Choosing an architecture for the model

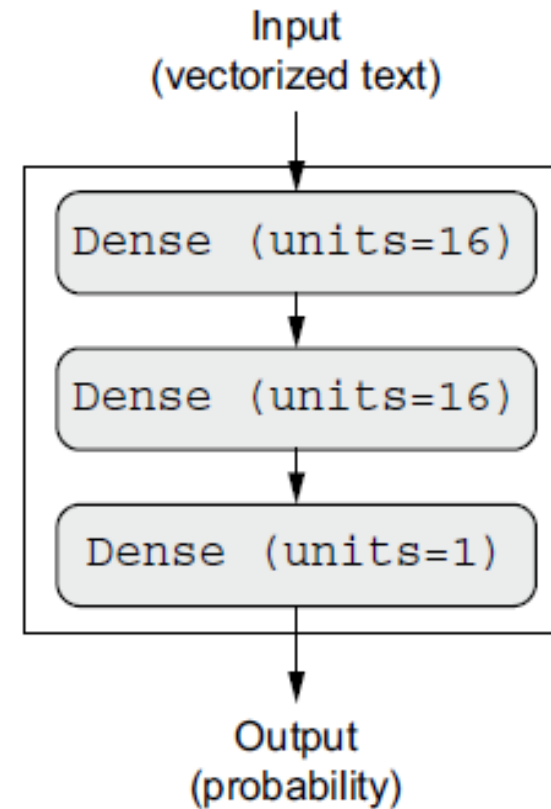
- The input data is vectors, and the labels are scalars (1s and 0s)
- This type of problem is one of the simplest in deep learning.
- A type of model that performs well on such a problem is a plain stack of densely connected (Dense) layers with relu activations.

Choosing an architecture for model

- Experience has shown that the following architecture choices are good:
- Two intermediate layers with 16 units each
- A third layer that will output the scalar prediction regarding the sentiment of the current review

Choosing an architecture for the model

- There are two key architecture decisions to be made about such a stack of Dense layers:
 - How many layers to use
 - How many units to choose for each layer



Model Definition

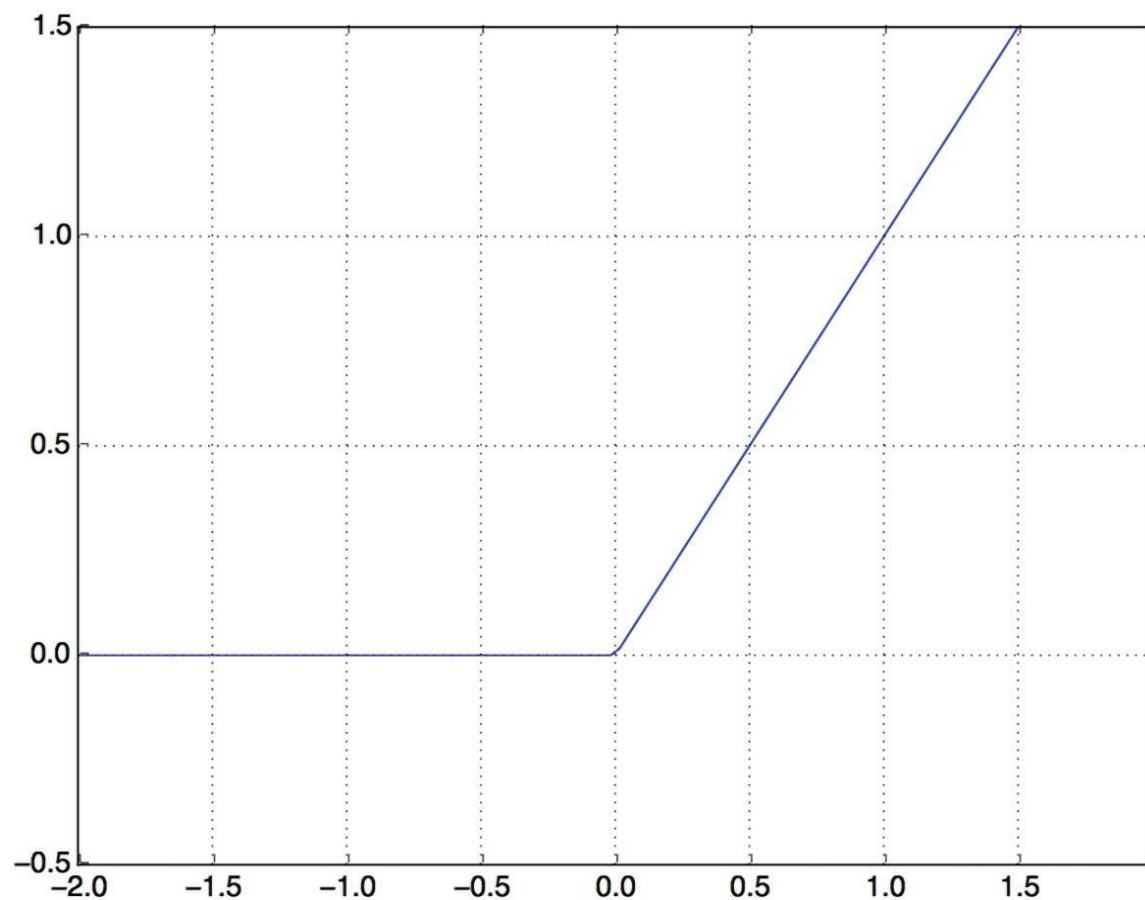
```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

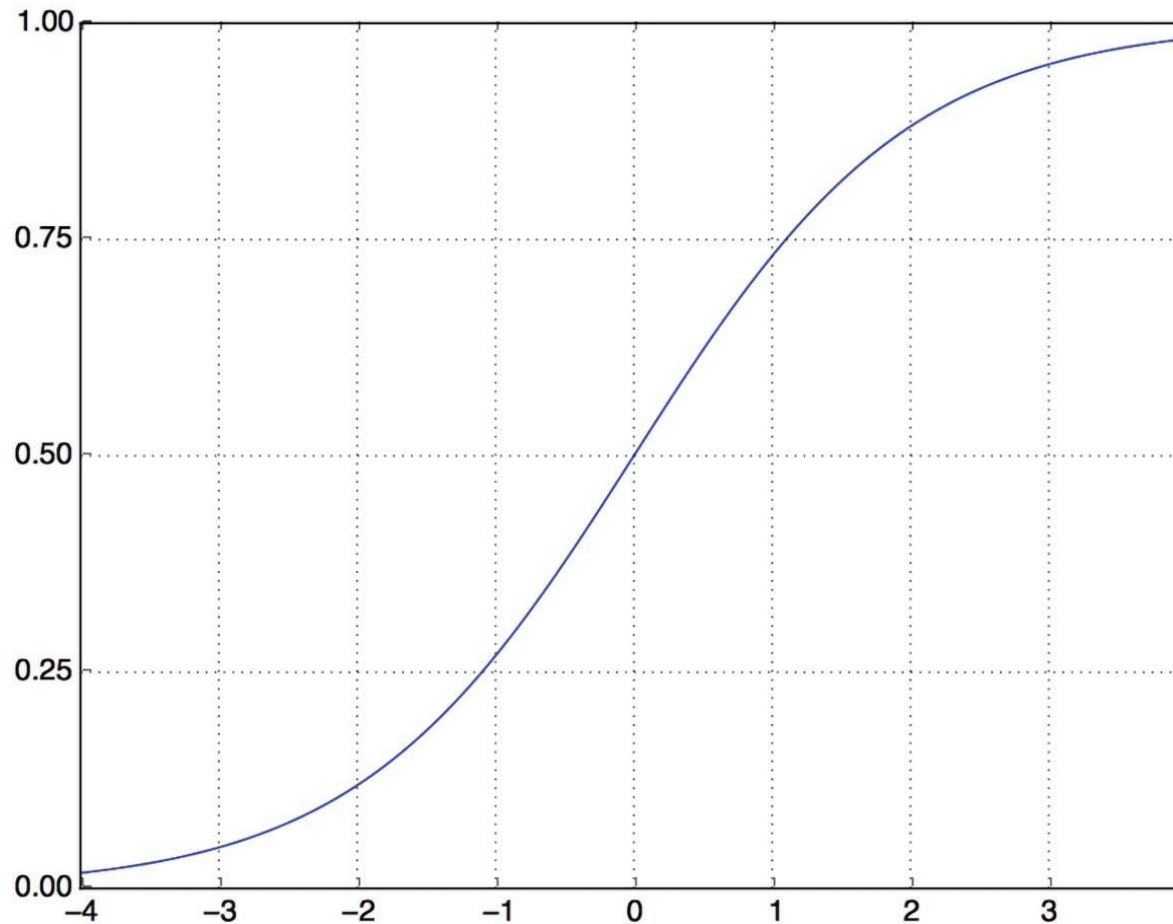
Understanding the architecture

- The intermediate layers use relu as their activation function, and the final layer uses a sigmoid activation so as to output a probability (a score between 0 and 1 indicating how likely the sample is to have the target “1”: how likely the review is to be positive).
- A relu (rectified linear unit) is a function meant to zero out negative values, whereas a sigmoid “squashes” arbitrary values into the $[0, 1]$ interval outputting something that can be interpreted as a probability

The rectified linear unit function (relu)



The sigmoid function



Choosing a loss function

- Finally, you need to choose a loss function and an optimizer.
- Because you're facing a binary classification problem and the output of your model is a probability (you end your model with a single-unit layer with a sigmoid activation), it's best to use the `binary_crossentropy` loss.

Compiling the model

- Here's the step where we configure the model with the rmsprop optimizer and the binary_crossentropy loss function.
- Note that we'll also monitor accuracy during training.

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

Validation

- A deep learning model should never be evaluated on its
- training data—it's standard practice to use a validation set to monitor the accuracy of the model during training.
- Here, we'll create a validation set by setting apart 10,000 samples from the original training data.

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

- We will now train the model for 20 epochs (20 iterations over all samples in the training data) in mini-batches of 512 samples.
- At the same time, we will monitor loss and accuracy on the 10,000 samples that we set apart.
- We do so by passing the validation data as the `validation_data` argument.

Training the model

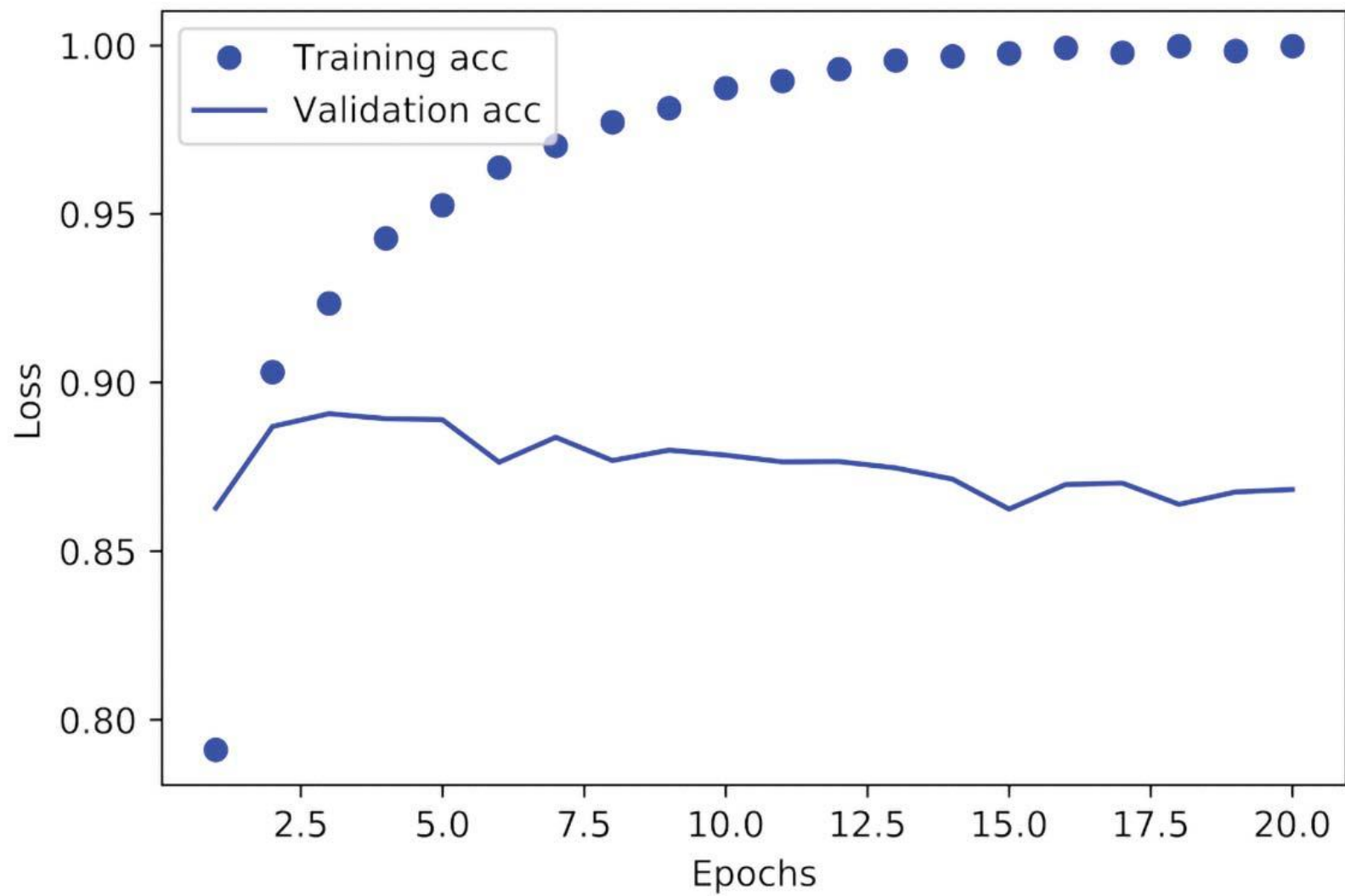
```
history = model.fit(partial_x_train,  
                    partial_y_train,  
                    epochs=20,  
                    batch_size=512,  
                    validation_data=(x_val, y_val))  
  
>>> history_dict = history.history  
>>> history_dict.keys()  
[u"accuracy", u"loss", u"val_accuracy", u"val_loss"]
```

Plotting 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(loss_values) + 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()
```


← "bo" is for
"blue dot."

← "b" is for
"solid blue line."

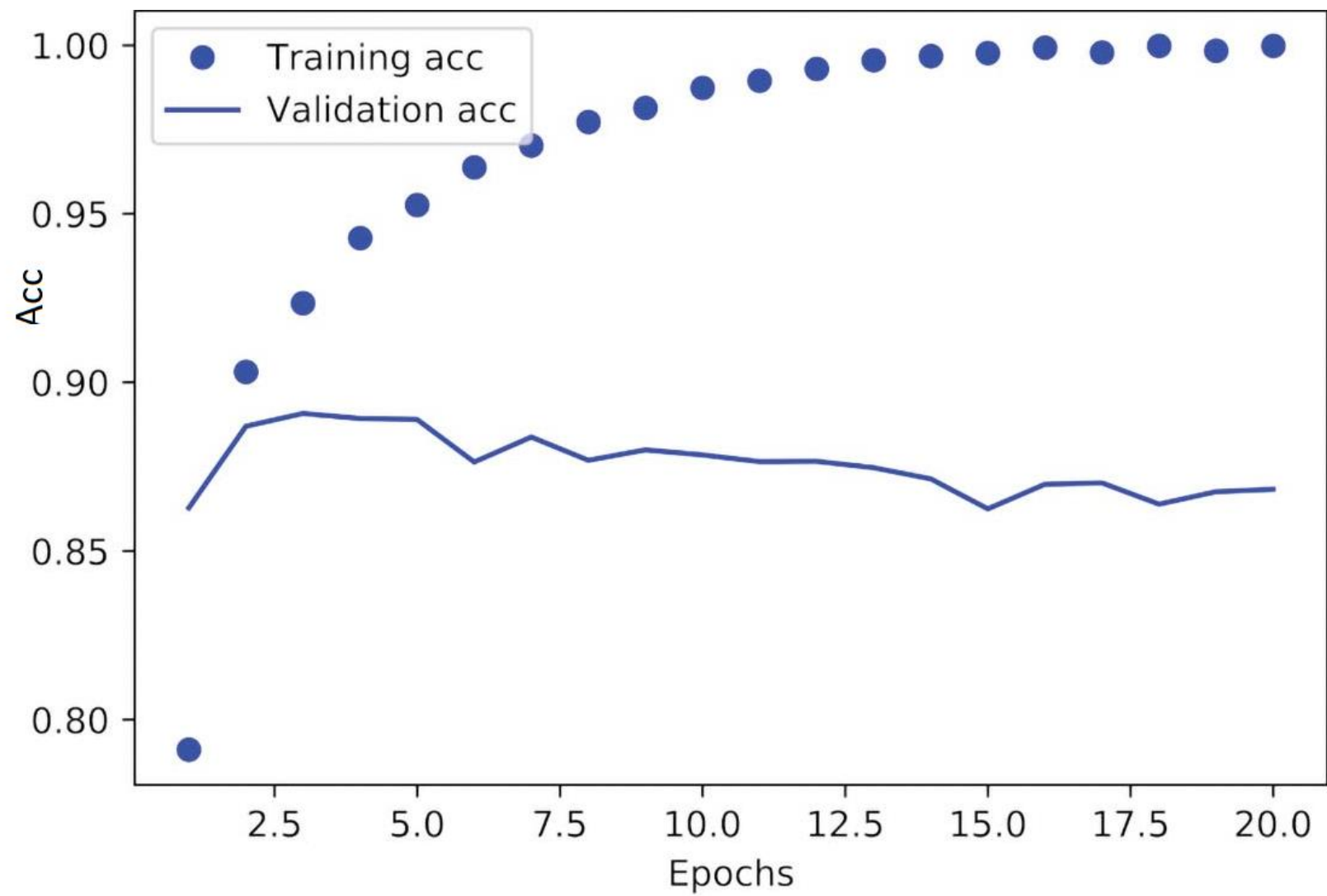


Plotting the training and validation accuracy

```
plt.clf()
acc = history_dict["accuracy"]
val_acc = history_dict["val_accuracy"]
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("Accuracy")
plt.legend()
plt.show()
```



Clears the figure



Overfitting

- Performance on the validation data shows that after the fourth epoch, the model begins to overfit.
- One of the things you can do to try and avoid overfitting is to stop training at the fourth epoch
- Let's train a new model for four epochs and evaluate it on the test data.

Retraining a model and stopping early

```
model = keras.Sequential([
    layers.Dense(16, activation="relu"),
    layers.Dense(16, activation="relu"),
    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)

>>> results
[0.2929924130630493, 0.88327999999999995]
```

Use the model to make predictions

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

Reference

- Deep Learning with Python by François Chollet, 2021 Manning Publications