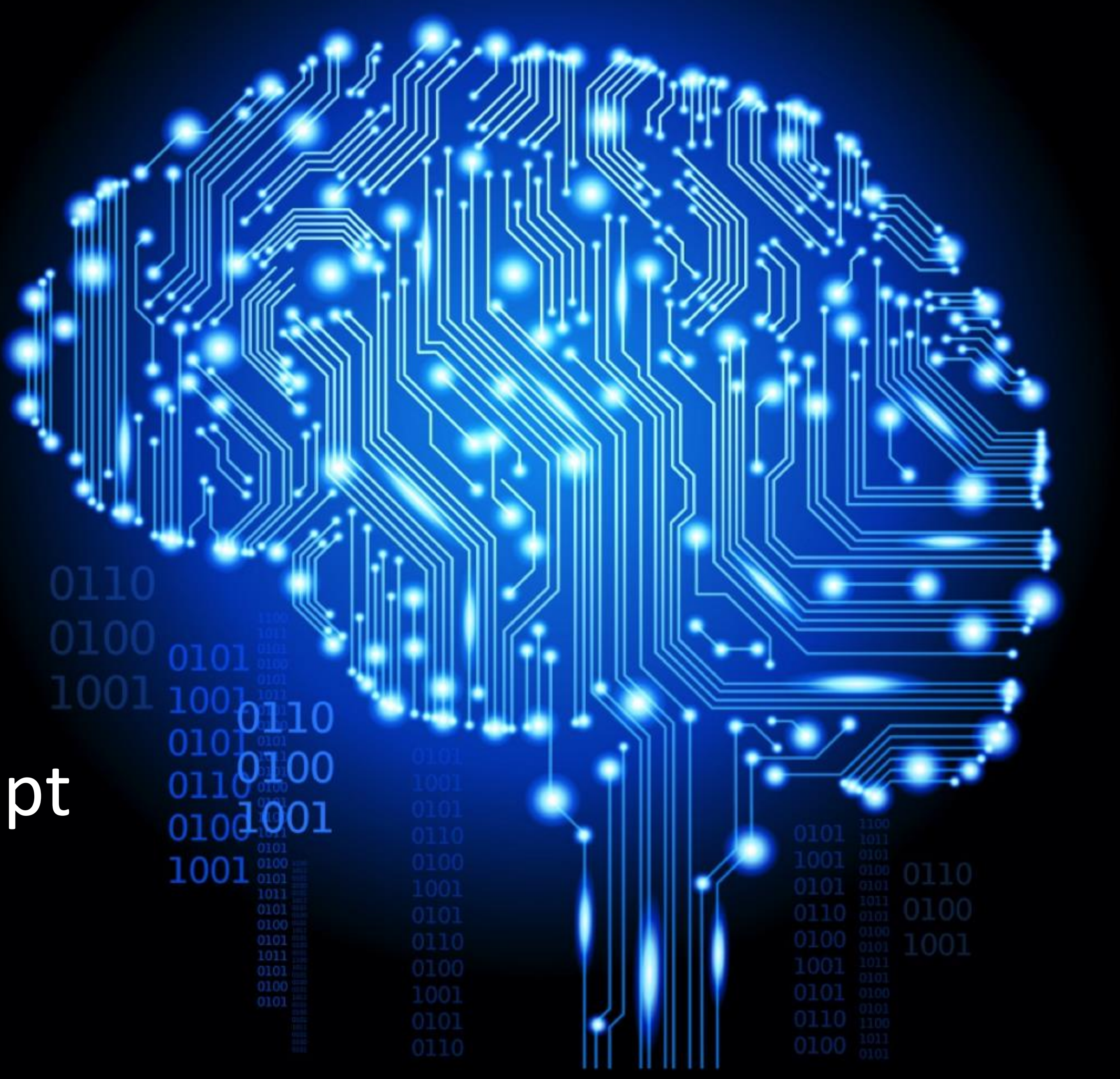
DEEP LEARNING

Spring 2021

MAURO CASTELLI
mcastelli@novaims.unl.pt



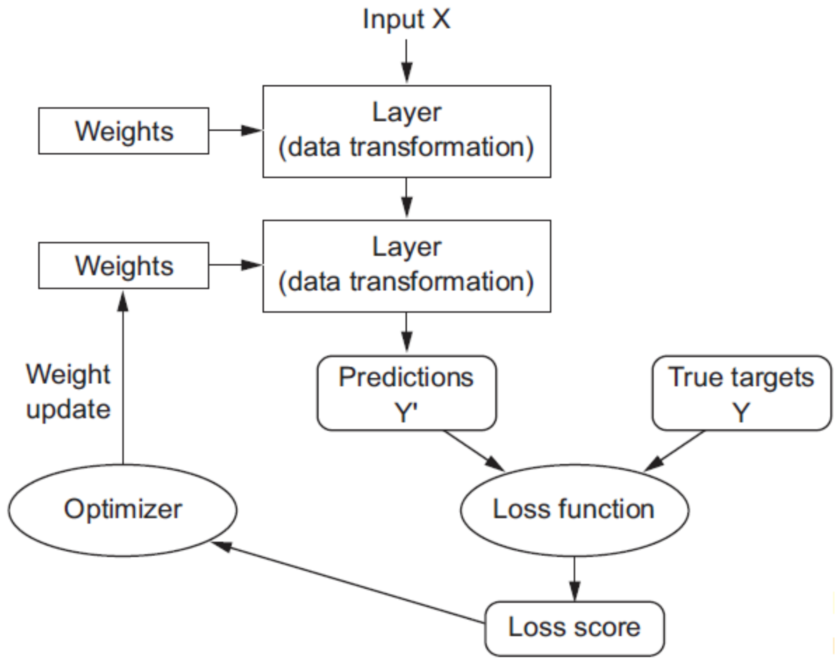
Outline

- Core components of neural networks
- Using neural networks to solve basic classification and regression problems

Core components of neural networks

Training a neural network revolves around the following objects:

- ✓ *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

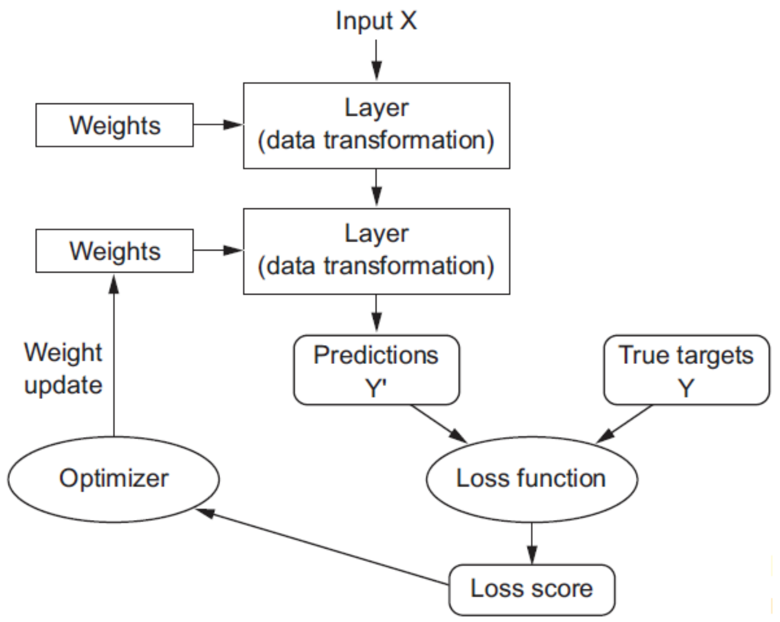


Core components of neural networks: layers

The network, composed of layers that are chained together, maps the input data to predictions.

The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the network's predictions match what was expected.

The optimizer uses this loss value to update the network's weights.



Core components of neural networks: layers

The fundamental data structure in neural networks is the *layer.* A layer is a data-processing module that takes as input one or more tensors and that outputs one or more tensors. Some layers are stateless, but more frequently layers have a state: the layer's *weights*, one or several tensors learned with stochastic gradient descent, which together contain the network's *knowledge*.

Different layers are appropriate for different tensor formats and different types of data processing.

- For instance, simple vector data, stored in 2D tensors of shape (samples, features), is often processed by *densely connected* layers, also called *fully connected* or *dense* layers (the Dense class in Keras).
- Sequence data, stored in 3D tensors of shape (samples, timesteps, features), is typically processed by *recurrent* layers such as an LSTM layer.
- Image data, stored in 4D tensors, is usually processed by 2D convolution layers (Conv2D).

Core components of neural networks: layers

You can think of layers as the LEGO bricks of deep learning, a metaphor that is made explicit by frameworks like Keras. Building deep-learning models in Keras is done by clipping together compatible layers to form useful data-transformation pipelines.

The notion of layer compatibility here refers specifically to the fact that every layer will only accept input tensors of a certain shape and will return output tensors of a certain shape.

Consider the following example:

```
from keras import layers
layer = layers.Dense(32, input_dim = 100)
```

We're creating a layer that will only accept as input tensors of 100 components (the independent variables). This layer will return a tensor where the first dimension has been transformed to be 32.

Thus this layer can only be connected to a downstream layer that expects 32-dimensional vectors as its input.

Core components of neural networks: layers

When using Keras, you don’t have to worry about compatibility, because the layers you add to your models are dynamically built to match the shape of the incoming layer. For instance, suppose you write the following:

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(layers.Dense(32, input_dim=100))
model.add(layers.Dense(32))
```

The second layer didn’t receive an input shape argument—instead, it automatically inferred its input shape as being the output shape of the layer that came before.

Core components of neural networks: model as a network of layers

A deep-learning model is a directed, acyclic graph of layers. The most common instance is a linear stack of layers, mapping a single input to a single output.

The topology of a network defines a *hypothesis space*.

By choosing a network topology, you constrain your hypothesis space to a specific series of tensor operations, mapping input data to output data. What you'll then be searching for is a good set of values for the weight tensors involved in these tensor operations.

Loss functions and optimizers

Once the network architecture is defined, you still have to choose two more things:

- ✓ *Loss function (objective function)*—The quantity that will be minimized during training. It represents a measure of success for the task at hand.
- ✓ *Optimizer*—Determines how the network will be updated based on the loss function. It implements a specific variant of stochastic gradient descent (SGD).

Choosing the right objective function for the right problem is extremely important: your network will take any shortcut it can, to minimize the loss; so if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

Loss functions and optimizers

Fortunately, for common problems such as classification, regression, and sequence prediction, there are simple guidelines you can follow to choose the correct loss.

For instance, you’ll use binary crossentropy for a two-class classification problem, categorical crossentropy for a many-class classification problem, mean squared error for a regression problem, and so on.

Binary Classification

Our first Keras model ☺

- Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem.

- In this example, we’ll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

Our first Keras model ☺

IMDB dataset: a set of 50000 reviews from the Internet Movie Database.

They’re split into 25000 reviews for training and 25000 reviews for testing, each set consisting of 50% negative and 50% positive reviews.

Just like the MNIST dataset, the IMDB dataset comes packaged with 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.

Our first Keras model ☺

The following code will load the dataset (when you run it the first time, ~ 80 MB of data will be downloaded to your machine).

Loading the IMDB dataset

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

The argument `num_words=10000` means you'll only keep the top 10000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of manageable size.

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*.

Loading the IMDB dataset

Example:

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

Data Preparation

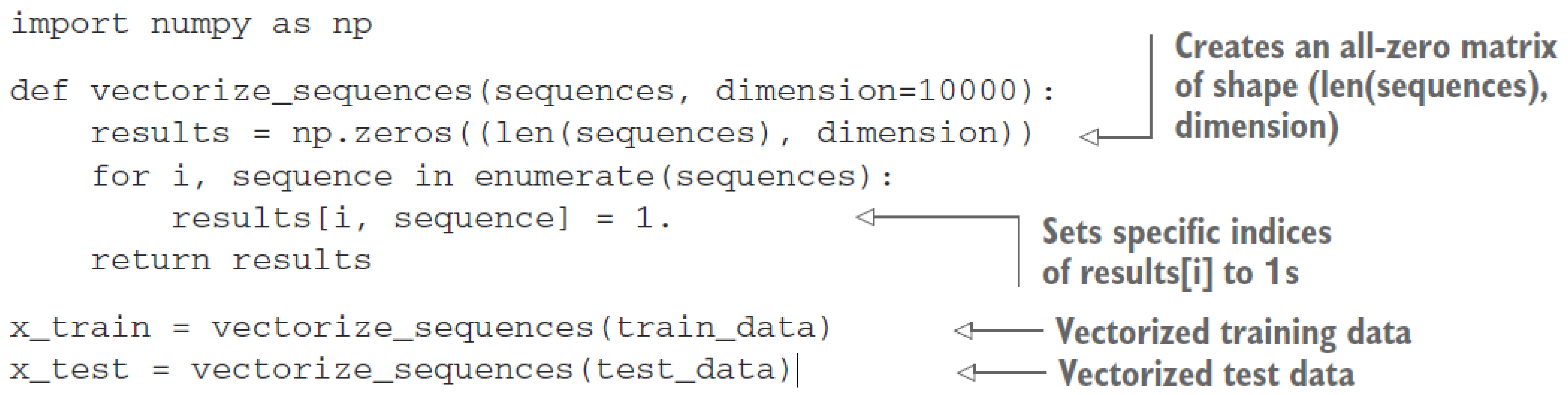
You can’t feed lists of integers into a neural network. You have to turn your lists into tensors. There are two ways to do that:

✓ Using an Embedding layer (commonly used for NLP problems).

✓ One-hot encode your lists to turn them into vectors of 0s and 1s. This would mean, for instance, 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. Then you could use as the first layer in your network a Dense layer, capable of handling floating-point vector data.

Let’s go with the latter solution to vectorize the data.

Data Preparation: 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

Data Preparation

Here’s what the samples look like now:

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

We should also vectorize the labels, which is straightforward:

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

Now the data is ready to be fed into a neural network.

Building the (not so deep) network

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

A type of network that performs well on such a problem is a simple stack of fully connected (Dense) layers with relu activations: `Dense(16,activation='relu')`.

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.

Such Dense layer with a relu activation implements the following chain of operations:

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

Building the network

Having 16 hidden units means 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 (and then you’ll add the bias vector b and apply the relu operation).

You can intuitively understand the dimensionality of your representation space as “how much freedom you’re allowing the network to have when learning internal representations.”

Having more hidden units (a higher-dimensional representation space) allows your network to learn more-complex representations, but it makes the network more computationally expensive and may lead to learning unwanted patterns (patterns that will improve performance on the training data but not on the test data).

Building the network

There are two key architecture decisions to be made about such a stack of Dense layers:

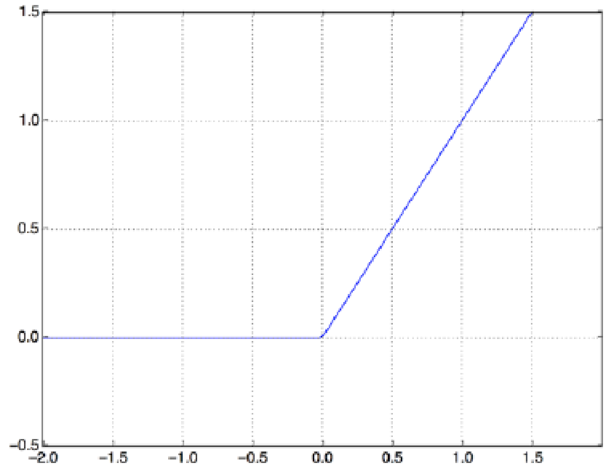
- ➢ How many layers to use
- ➢ How many hidden units to choose for each layer

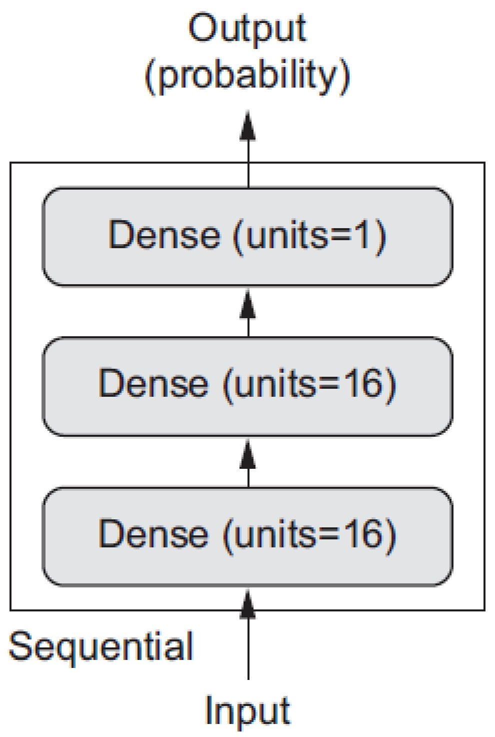
We’ll learn formal principles to guide you in making these choices. For the time being, you’ll have to trust me with the following architecture choice:

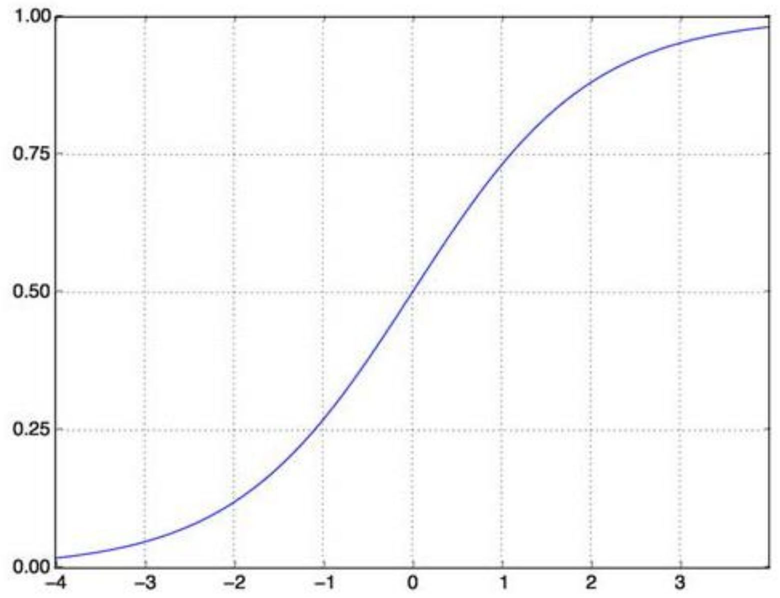
- ✓ Two intermediate layers with 16 hidden units each
- ✓ A third layer that will output the scalar prediction regarding the sentiment of the current review

Building the network

The intermediate layers will use relu as their activation function, and the final layer will use 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).


The rectified linear unit function




The sigmoid function

Aside: What are activation functions, and why are they necessary?

Without an activation function like relu (also called a *non-linearity*), the Dense layer would consist of two linear operations—a dot product and an addition: output = dot(W, input) + b

So the layer could only learn *linear transformations* (affine transformations) of the input data: the *hypothesis space* of the layer would be the set of all possible linear transformations of the input data into a 16-dimensional space. Such a hypothesis space is too restricted and adding more layers wouldn't extend the hypothesis space.

In order to get access to a much richer hypothesis space that would benefit from deep representations, you need a non-linearity, or activation function. relu is the most popular activation function in deep learning.

Model Definition in Keras

```
from keras import models
from keras import layers

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

Model Definition in Keras

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

Crossentropy is a quantity from the field of Information Theory that measures the distance between probability distributions or, in this case, between the ground-truth distribution and your predictions.

Model Definition in Keras

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

You’re passing your optimizer, loss function, and metrics as strings, which is possible because `rmsprop`, `binary_crossentropy`, and `accuracy` are packaged as part of Keras.

It is also possible to configure the parameters of your optimizer or pass a custom loss function or metric function (refer to Keras documentation).

The importance of the validation set

In order to monitor during training the accuracy of the model on data it has never seen before, you'll create a validation set by setting apart 10000 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:]
```

Training the Model

We'll now train the model for 20 epochs (20 iterations over all samples in the `x_train` and `y_train` tensors), in mini-batches of 512 samples. At the same time, you'll monitor loss and accuracy on the 10000 samples that you set apart. You do so by passing the validation data as the `validation_data` argument.

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

Training the Model

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']
```

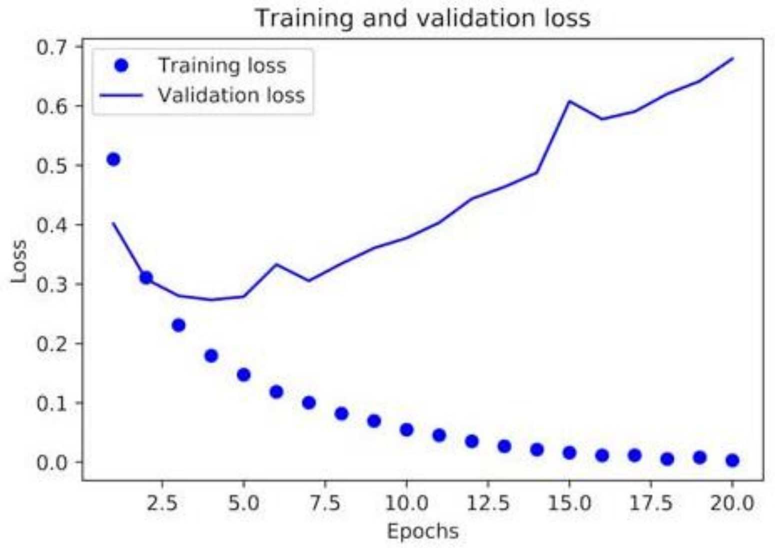
The dictionary contains four entries: one per metric that was being monitored during training and during validation.

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(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()
```

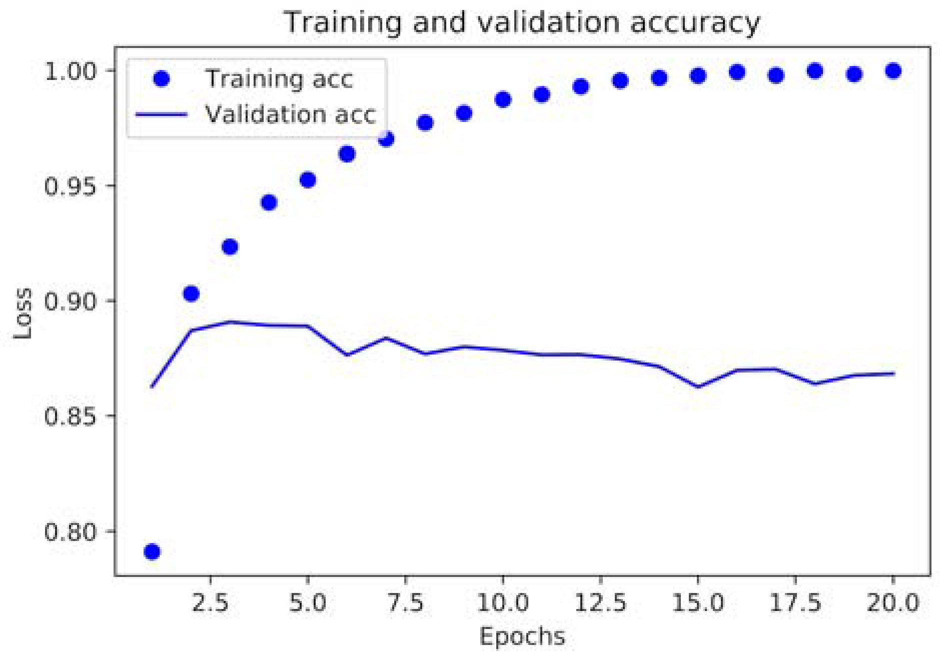
"bo" is for "blue dot."

"b" is for "solid blue line."



Plotting the training and validation accuracy

```
plt.clf()                    <—— Clears the figure
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()
```



Observations

As you can see, the training loss decreases with every epoch, and the training accuracy increases with every epoch. But that isn't the case for the validation loss and accuracy: they seem to peak at the fourth epoch.

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. In precise terms, what you're seeing is *overfitting.*

In this case, to prevent overfitting, you could stop training after three epochs. In general, you can use a range of techniques to mitigate overfitting.

Evaluation on Test Data

Let’s train a new network from scratch for four epochs and then evaluate it on the test data.

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_dim = 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)
```

The final results are as follows:

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

Evaluation on Test Data

You can generate the likelihood of reviews being positive by using the predict method:

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

Further Experiments

- ❖You used two hidden layers. Try using one or three hidden layers, and see how doing so affects validation and test accuracy.

- ❖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`.

Take home messages

We usually need to do some preprocessing on your raw data in order to be able to feed it into a neural network.

Stacks of Dense layers with relu activations can solve a wide range of problems, and you'll likely use them frequently.

In a binary classification problem (two output classes), 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.

With such a scalar sigmoid output on a binary classification problem, the loss function you should use is binary_crossentropy.

The rmsprop optimizer is generally a good choice, whatever your problem. That's one less thing for you to worry about.

As they get better on their training data, neural networks eventually start overfitting and end up obtaining increasingly worse results on data they've never seen before. Be sure to always monitor performance on data that is outside of the training set.