Classifying movie reviews: a binary classification example

Material: http://bit.ly/rladies-sf-dl

IMDB dataset

50,000 highly polarized reviews from the Internet Movie Database.

They’re split into:

- 25,000 reviews for training
- 25,000 reviews for testing
- each set consisting of 50% negative and 50% positive reviews.
- The reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary.

Build your network

Steps

1. Data Preprocessing

2. Constructing the Model

3. Compile And Fit The Model

4. Evaluating Your Model

5. Predict Labels of New Data

6. Fine-tuning Your Model

7. Saving, Loading or Exporting Your Model

1) Data Preprocessing

DONE!

See

`00-data-preparation-imdb.Rmd`

2) Constructing the model

2) Constructing the model

1) To start constructing a model, you should first initialize a sequential model with the `keras_model_sequential()`

2) Define how many layers (and its arguments)

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = "relu",
              input_shape = ncol(x_train)) %>%
  layer_dense(units = 16,
              activation = "relu") %>%
  layer_dense(units = 1,
              activation = "sigmoid")
```

Define a model

First layer

Second layer

Third layer

Layers

```
model <- keras_model_sequential() %>%
```

```
  layer_dense(units = 16,
              activation = "relu",
              input_shape = ncol(x_train)) %>%
  layer_dense(units = 16,
              activation = "relu") %>%
```

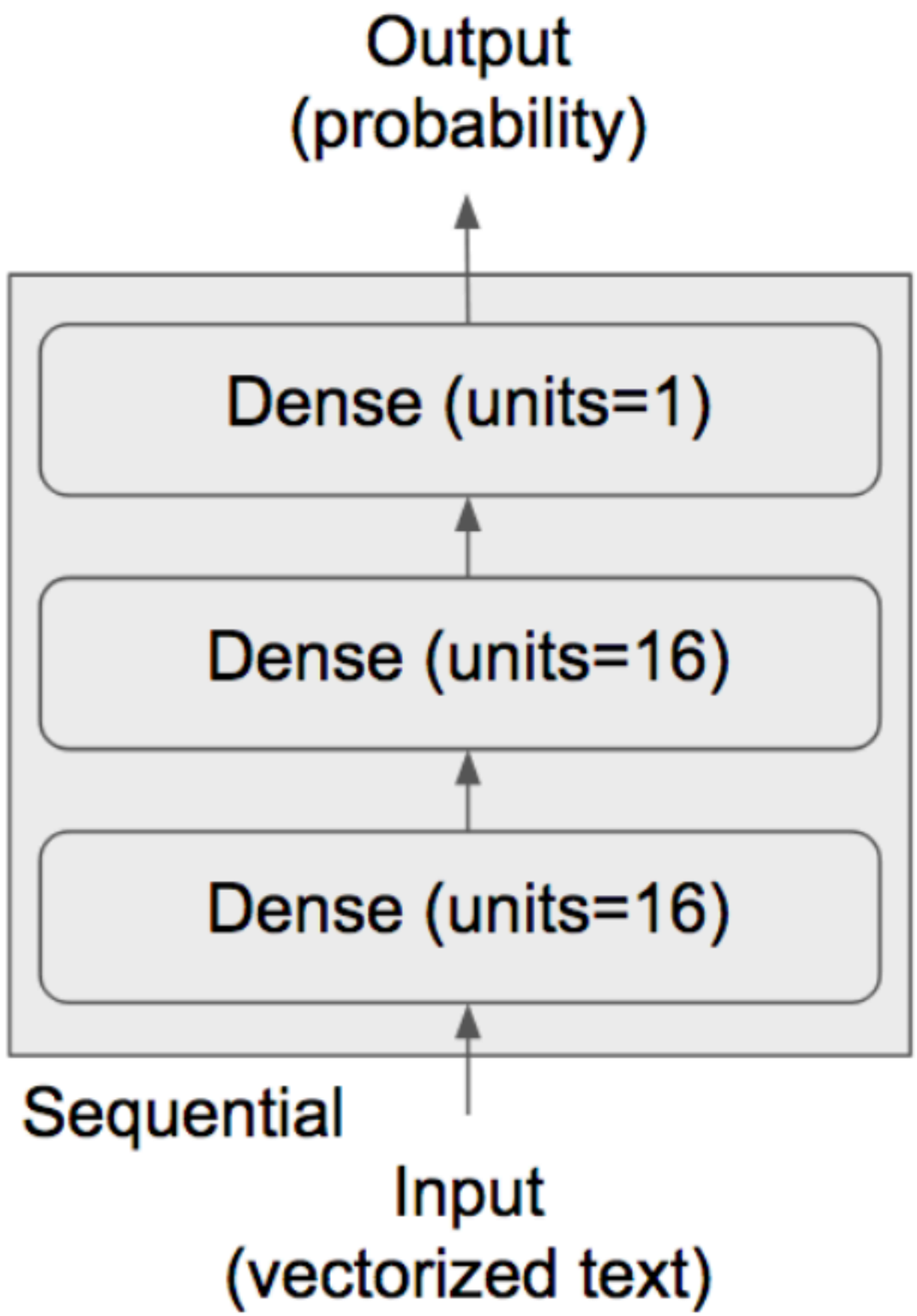
two intermediate layers with 16 hidden units each and **relu** as their "activation function"

3rd layer

output is the scalar prediction regarding the sentiment of the current review.

```
  layer_dense(units = 1,
              activation = "sigmoid")
```

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", i.e. how likely the review is to be positive).


Output
(probability)
Dense (units=1)
Dense (units=16)
Dense (units=16)
Sequential
Input
(vectorized text)

Model

```
> model
```

```
Model
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                160016
_________________________________________________________________
dense_2 (Dense)              (None, 16)                272
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,305
Trainable params: 160,305
Non-trainable params: 0
_________________________________________________________________
```

3) Compile and Fit The Model

Compile the Model

For **compile** we need:

1) **loss function**

Since we are facing a binary classification problem and the output of our network is a probability it is best to use the **binary_crossentropy loss**.

Crossentropy is usually the best choice when you are dealing with models that output **probabilities**. Crossentropy measures the "distance" between probability distributions, or in our case, between the ground-truth distribution and our predictions.

Compile the Model

2) optimizer*

3) metric

```
model %>% compile(
optimizer = "rmsprop",
loss = "binary_crossentropy",
metrics = c(“accuracy")
)
```

* optimizer specifies the exact way in which the gradient of the loss will be used to update parameters: for instance, it could be the RMSProp optimizer, SGD with momentum, and so on. More info: https://keras.io/optimizers/#rmsprop

Fit the Model - Validating the Approach

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 10,000 samples from the original training data.

```
validation_indices <- 1:10000

# Validation Set
x_validation <- x_train[validation_indices,]
y_validation <- y_train[validation_indices]

# Training Set
partial_x_train <- x_train[-validation_indices,]
partial_y_train <- y_train[-validation_indices]
```

Fit the Model

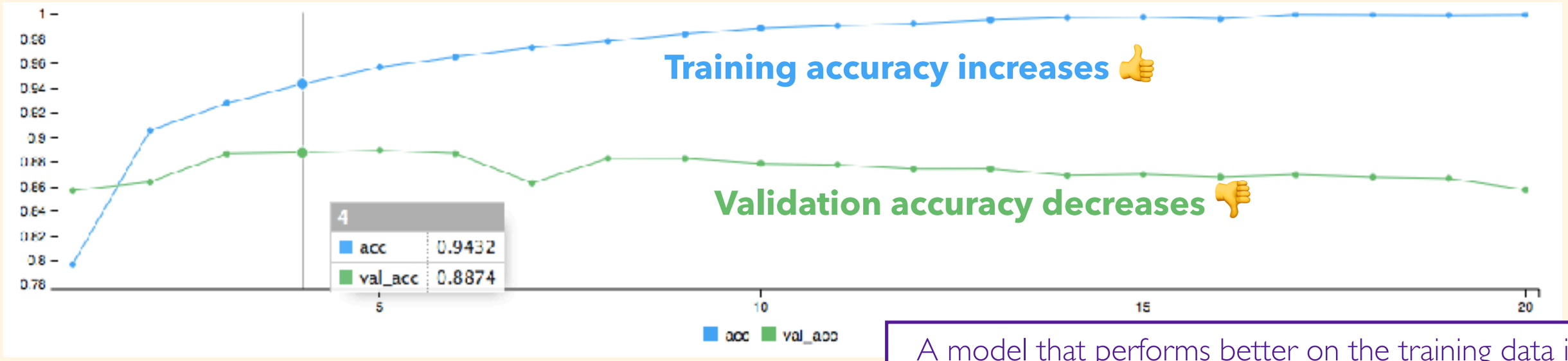
```
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_validation, y_validation)
)
```

20 iterations over all samples in the x_train and y_train tensors

The batch size defines the number of samples that going to be propagated through the network.
By doing this, you optimize the efficiency because you make sure that you don't load too many input patterns into memory at the same time.

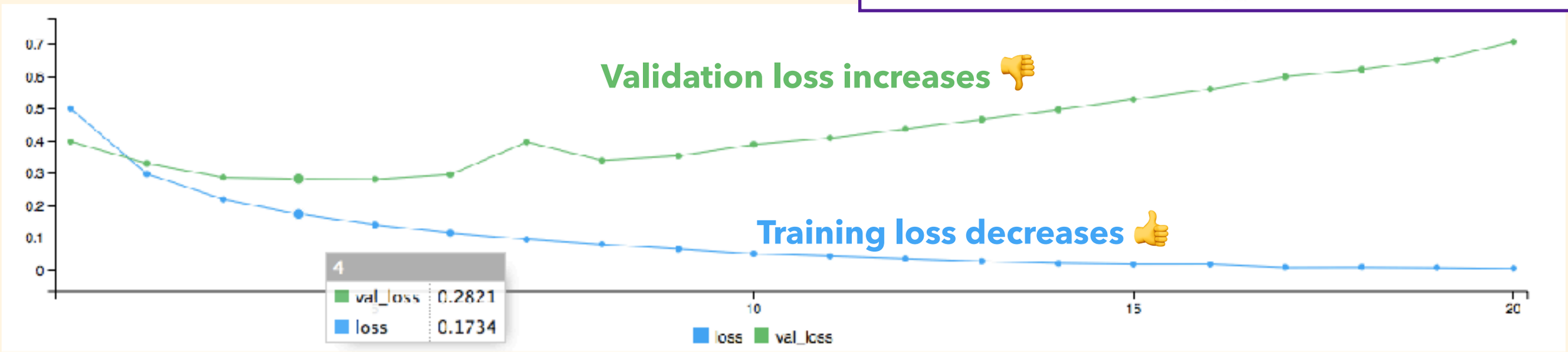
```
Train on 15000 samples, validate on 10000 samples
Epoch 1/20
15000/15000 [==============================] - 4s 256us/step - loss: 0.1427 - acc: 0.9509 - val_loss: 0.1428 - val_acc: 0.9515
Epoch 2/20
15000/15000 [==============================] - 2s 133us/step - loss: 0.1176 - acc: 0.9611 - val_loss: 0.1550 - val_acc: 0.9440
Epoch 3/20
15000/15000 [==============================] - 2s 149us/step - loss: 0.0951 - acc: 0.9721 - val_loss: 0.2155 - val_acc: 0.9165
Epoch 4/20
15000/15000 [==============================] - 2s 116us/step - loss: 0.0814 - acc: 0.9755 - val_loss: 0.1866 - val_acc: 0.9303
Epoch 5/20
15000/15000 [==============================] - 2s 116us/step - loss: 0.0683 - acc: 0.9811 - val_loss: 0.2188 - val_acc: 0.9187
Epoch 6/20
15000/15000 [==============================] - 2s 158us/step - loss: 0.0550 - acc: 0.9854 - val_loss: 0.2619 - val_acc: 0.9071
Epoch 7/20
15000/15000 [==============================] - 2s 112us/step - loss: 0.0439 - acc: 0.9911 - val_loss: 0.3127 - val_acc: 0.8969
Epoch 8/20
15000/15000 [==============================] - 2s 105us/step - loss: 0.0385 - acc: 0.9913 - val_loss: 0.2702 - val_acc: 0.9129
Epoch 9/20
15000/15000 [==============================] - 2s 106us/step - loss: 0.0288 - acc: 0.9949 - val_loss: 0.2980 - val_acc: 0.9098
Epoch 10/20
15000/15000 [==============================] - 2s 109us/step - loss: 0.0239 - acc: 0.9957 - val_loss: 0.3212 - val_acc: 0.9055
Epoch 11/20
15000/15000 [==============================] - 2s 116us/step - loss: 0.0171 - acc: 0.9980 - val_loss: 0.3486 - val_acc: 0.9036
Epoch 12/20
15000/15000 [==============================] - 2s 165us/step - loss: 0.0145 - acc: 0.9981 - val_loss: 0.3851 - val_acc: 0.8997
Epoch 13/20
15000/15000 [==============================] - 2s 139us/step - loss: 0.0124 - acc: 0.9979 - val_loss: 0.4069 - val_acc: 0.8969
Epoch 14/20
15000/15000 [==============================] - 2s 111us/step - loss: 0.0067 - acc: 0.9998 - val_loss: 0.4472 - val_acc: 0.8924
Epoch 15/20
15000/15000 [==============================] - 3s 172us/step - loss: 0.0085 - acc: 0.9985 - val_loss: 0.4602 - val_acc: 0.8955
Epoch 16/20
15000/15000 [==============================] - 2s 130us/step - loss: 0.0045 - acc: 0.9998 - val_loss: 0.5037 - val_acc: 0.8893
Epoch 17/20
15000/15000 [==============================] - 2s 120us/step - loss: 0.0028 - acc: 0.9999 - val_loss: 0.5206 - val_acc: 0.8917
Epoch 18/20
15000/15000 [==============================] - 2s 144us/step - loss: 0.0034 - acc: 0.9997 - val_loss: 0.5523 - val_acc: 0.8901
Epoch 19/20
15000/15000 [==============================] - 2s 127us/step - loss: 0.0036 - acc: 0.9992 - val_loss: 0.5788 - val_acc: 0.8890
Epoch 20/20
15000/15000 [==============================] - 2s 119us/step - loss: 0.0010 - acc: 1.0000 - val_loss: 0.6015 - val_acc: 0.8870
```

Accuracy



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.
This is a case of ***overfitting***: after the 2nd epoch, you're over-optimizing on the training data, and you end up learning representations that are specific to the training data and don't generalize to data outside of the training set.

Loss



Note that your own results may vary slightly due to a different random initialization of your network.

4) Evaluate the Model

Retrain the Model

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

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = "relu",
              input_shape = c(10000)) %>%
  layer_dense(units = 16,
              activation = "relu") %>%
  layer_dense(units = 1,
              activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

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

Build the Model

Compile

Fit

Evaluate

Evaluate the model on the test data.

```
results <- model %>% evaluate(x_test, y_test)
```

```
> results
$loss
[1] 0.2916625

$acc
[1] 0.884
```

Accuracy: 88%

5) Predictions on new data

Using a trained network to generate predictions on new data

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

```
model %>% predict(x_test[1:10,])

       [,1]
 [1,] 0.1922472
 [2,] 0.9998888
 [3,] 0.8777745
 [4,] 0.8634432
 [5,] 0.9594250
 [6,] 0.9000725
 [7,] 0.9998257
 [8,] 0.0138099
 [9,] 0.9733002
[10,] 0.9950176
```

As you can see, the network is confident for some samples (0.99 or more, or 0.01 or less) but less confident for others (0.8, 0.2).

Further experiments

Room for improvement

- We used 2 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: 32 units, 64 units, and so on.

- Try using the `mse` loss function instead of `binary_crossentropy`.

- Try using the `tanh` activation (an activation that was popular in the early days of neural networks) instead of `relu`.

Wrapping Up

Take aways

- You usually need to do quite a bit of **preprocessing** on your raw data in order to be able to feed it—as tensors—into a neural network. Sequences of words can be encoded as binary vectors, but there are other encoding options, too.

- Stacks of dense layers with `relu` activations can solve a wide range of problems (including sentiment classification), and you'll likely use them frequently.

- In a binary classification problem, 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.

Take aways

- 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 enough 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.🚨