Deep Learning

Tristan Cazenave

Tristan.Cazenave@dauphine.psl.eu

Backpropagation

A spectrum of machine learning tasks

Typical Statistics------------Artificial Intelligence

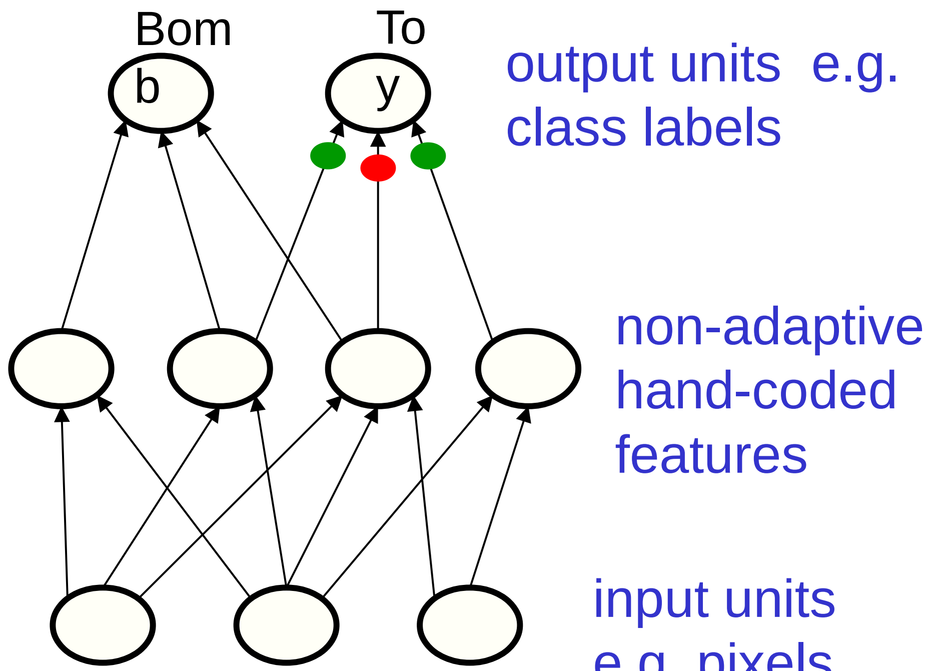
- Low-dimensional data (e.g. less than 100 dimensions)
- Lots of noise in the data
- There is not much structure in the data, and what structure there is, can be represented by a fairly simple model.
- The main problem is distinguishing true structure from noise.

- High-dimensional data (e.g. more than 100 dimensions)
- The noise is not sufficient to obscure the structure in the data if we process it right.
- There is a huge amount of structure in the data, but the structure is too complicated to be represented by a simple model.
- The main problem is figuring out a way to represent the complicated structure so that it can be learned.

Historical background:

First generation neural networks

- Perceptrons (~1960) used a layer of hand-coded features and tried to recognize objects by learning how to weight these features.
  - There was a neat learning algorithm for adjusting the weights.
  - But perceptrons are fundamentally limited in what they can learn to do.

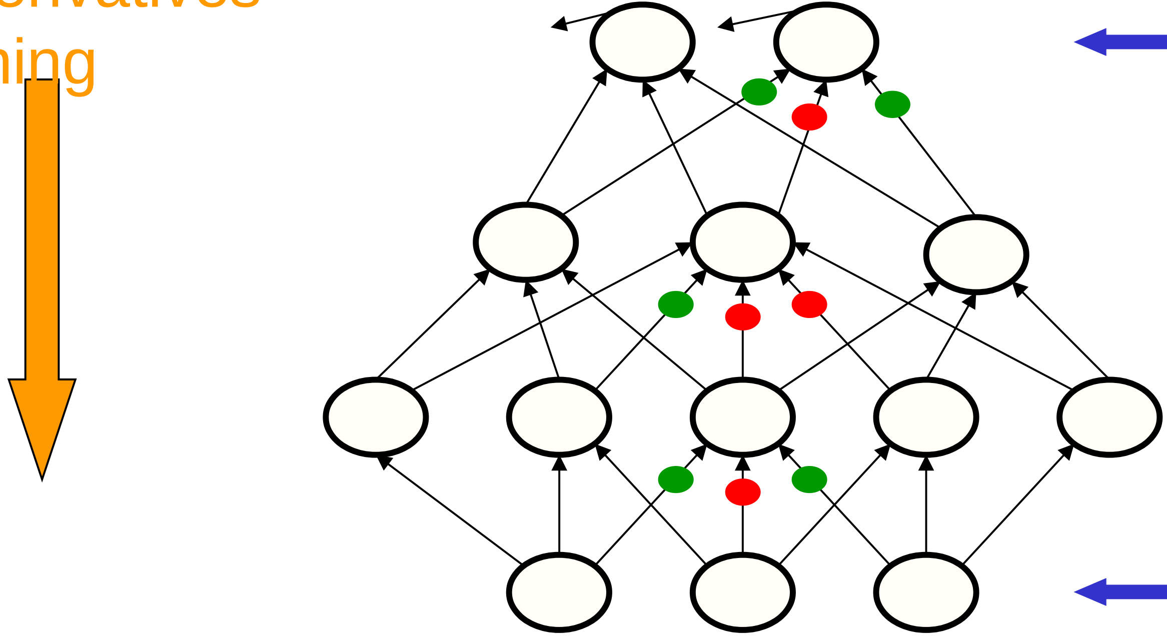

Bom
To
b
y
output units e.g. class labels
non-adaptive hand-coded features
input units e.g. pixels

Sketch of a typical perceptron from the 1960's

Second generation neural networks (~1985)

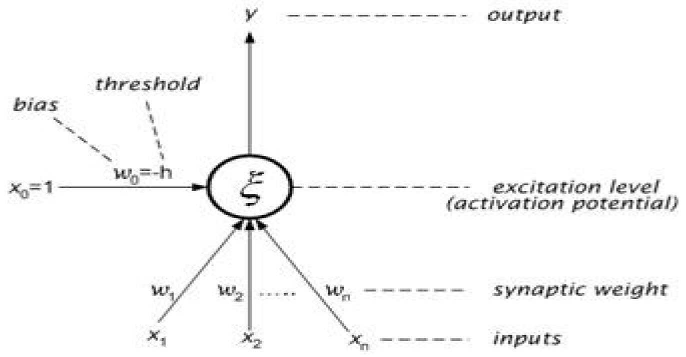
Back-propagate error signal to get derivatives for learning

Compare outputs with correct answer to get error signal



Formalization of a neural network

- Each neuron computes its output y as a linear combinations of its inputs x_i followed by an activation function :



Formalization of a neural network

- Linear combination : $\xi = \sum_{i=1}^{n} w_i x_i$

- Activation function :

$$y = \sigma(\xi) = \begin{cases} 1 \ if \ \xi \geq 0 \\ 0 \ if \ \xi < 0 \end{cases} \ where \ \xi = \sum_{i=0}^{n} w_i x_i$$

Formalization of a neural network

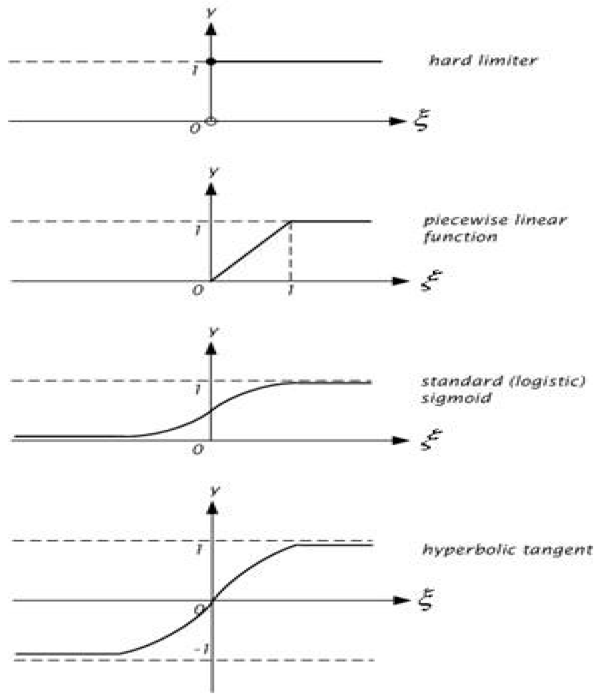
- Activation functions :

$$\sigma(\xi) = \begin{cases} 1 \; if \; \xi \geq 0 \\ 0 \; if \; \xi < 0 \end{cases}$$

$$\sigma(\xi) = \begin{cases} 1 & \xi > 1 \\ \xi & 0 \leq \xi \leq 1 \\ 0 & \xi < 0 \end{cases}$$

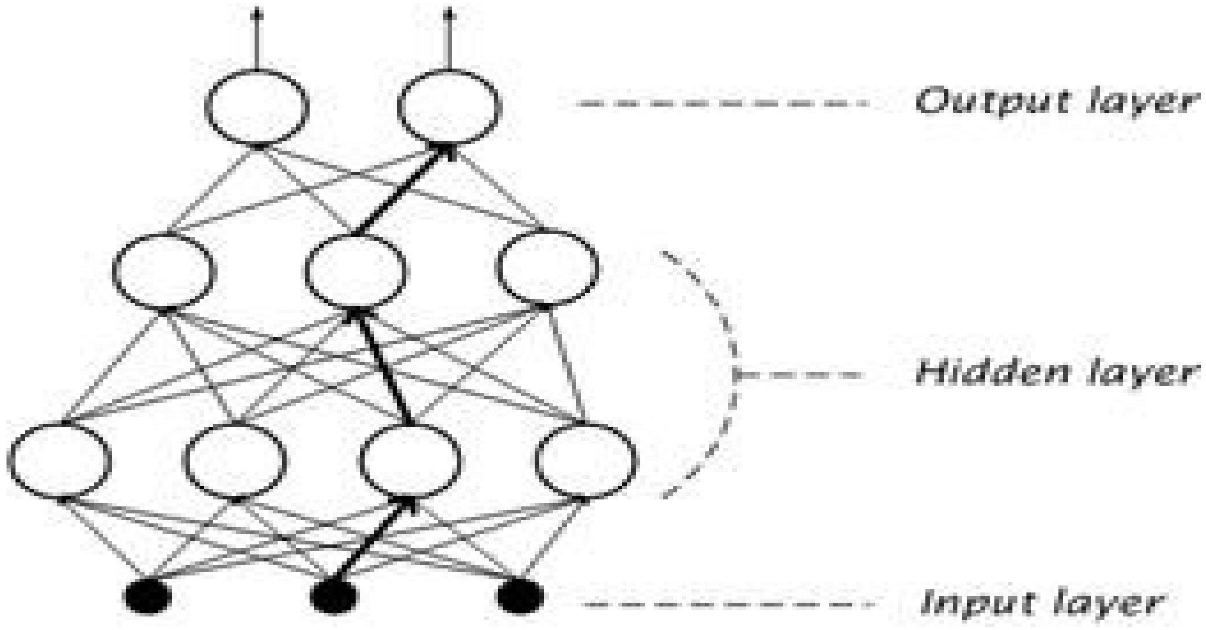
$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}}$$

$$\sigma(\xi) = \tanh\left(\frac{1}{2}\xi\right) = \frac{1 - e^{-\xi}}{1 + e^{-\xi}}$$



Formalization of a neural network

- Architecture :

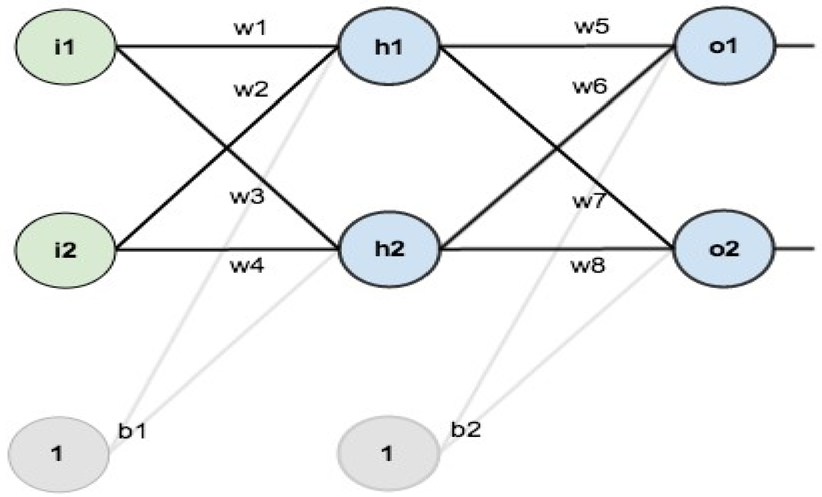


Backpropagation

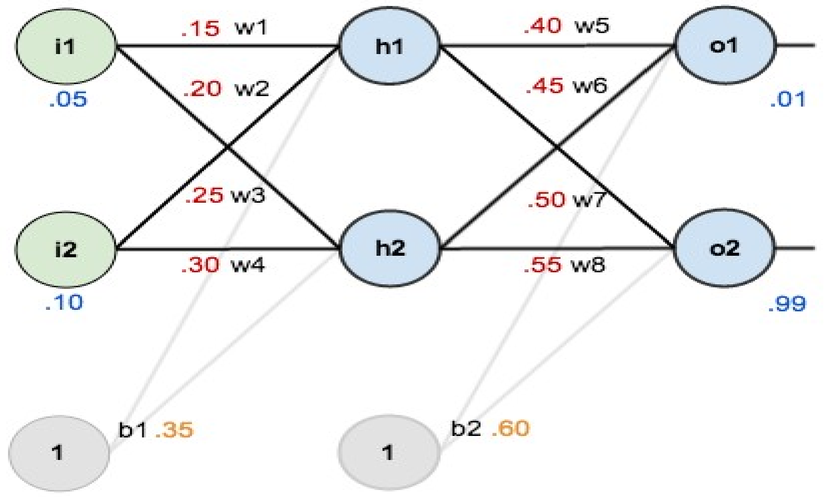
- We are going to explain backpropagation on a simple example.

- We take as example a network with two inputs, two outputs and two hidden neurons .

Backpropagation



Backpropagation



Backpropagation

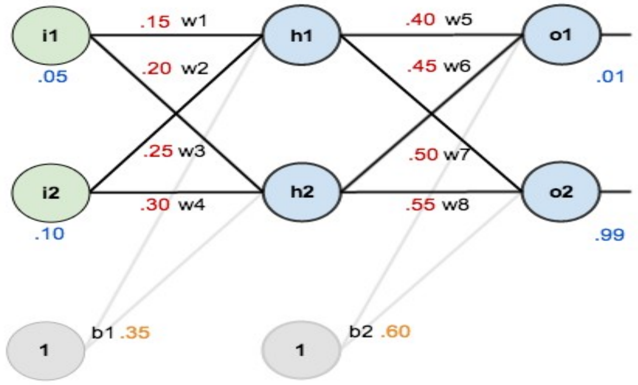
- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

- We are going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

The Forward Pass

- We figure out the total net input to each hidden layer neuron.
- Squash the total net input using an activation function (here we use the sigmoid function).
- Repeat the process with the output layer neurons.

The Forward Pass



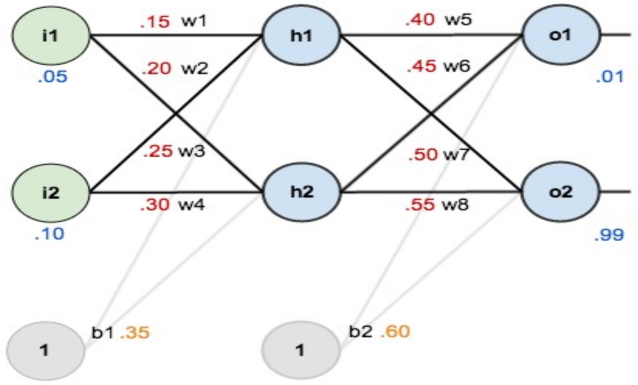
- Here’s how we calculate the total net input for h1:

  net_{h1} = w1 * i1 + w2 * i2 + b1 * 1

  net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775
- We then squash it using the sigmoid function to get the output of h1 :

  out_{h1} = 1/(1+e^-net_{h1}) = 1/(1+e^-0.3775) = 0.593269992
- Carrying out the same process for h2 we get : out_{h2} = 0.596884378

The Forward Pass



- We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.
- Here's the output for o1: $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$

  $out_{o1} = 1/(1+e^{\wedge}-net_{o1}) = 1/(1+e^{\wedge}-1.105905967) = 0.75136507$
- And carrying out the same process for o2 we get: $out_{o2} = 0.772928465$

The Error

- We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

  $E_{total} = \Sigma\ 1/2\ (\text{target} – \text{output})^2$

- For example, the target output for o1 is 0.01 but the neural network output 0.75136507, therefore its error is:

  $E_{o1} = 1/2\ (\text{target}_{o1} - \text{out}_{o1})^2 = 1/2\ (0.01 - 0.75136507)^2 = 0.274811083$

- Repeating this process for o2 we get: $E_{o2} = 0.023560026$
- The total error for the neural network is the sum of these errors:

  $E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$

The Backwards Pass

- Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

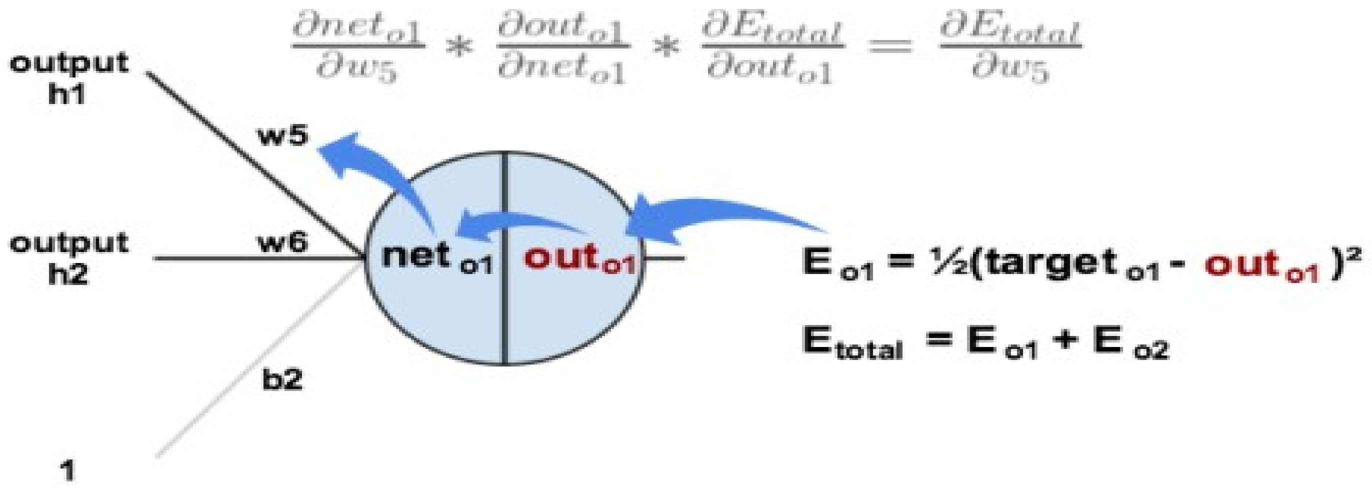
Output Layer

- Consider w5.
- We want to know how much a change in w5 affects the total error:

  $\delta E_{total} / \delta w5$
- $\delta E_{total} / \delta w5$ is read as “the partial derivative of E_{total} with respect to w5“.
- You can also say “the gradient with respect to w5“.
- By applying the chain rule we know that:

  $\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$

Output Layer



$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$
output h1
output h2
1
w5
w6
b2
net_{o1}
out_{o1}
$E_{o1} = ½(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

Output Layer

- We need to figure out each piece in this equation.
- First, how much does the total error change with respect to the output?

$E_{total} = 1/2\ (target_{o1} - out_{o1})^2 + 1/2\ (target_{o2} - out_{o2})^2$

$\delta E_{total} / \delta out_{o1} = -(target_{o1} - out_{o1})$

$\delta E_{total} / \delta out_{o1} = -(0.01 - 0.75136507) = 0.74136507$

Output Layer

- Next, how much does the output of o1 change with respect to its total net input?

  $out_{o1} = 1/(1+e^{\wedge}(-net_{o1}))$

  $\delta out_{o1} / \delta net_{o1} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$

- Finally, how much does the total net input of o1 change with respect to w5?

  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $\delta net_{o1} / \delta w5 = out_{h1} = 0.593269992$

Output Layer

- Putting it all together:

  $\delta E_{total} / \delta w5 = \delta E_{total} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} * \delta net_{o1} / \delta w5$

  $\delta E_{total} / \delta w5 = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$

- To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, η, which we'll set to 0.5):

  $w5 = w5 - \eta * \delta E_{total} / \delta w5 = 0.4 - 0.5 * 0.082167041 = 0.35891648$

Output Layer

- We can repeat this process to get the new weights w6, w7, and w8:

  w6 = 0.408666186

  w7 = 0.511301270

  w8 = 0.561370121

- We perform the actual updates in the neural network after we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

Hidden Layer

- Next, we’ll continue the backwards pass by calculating new values for w1, w2, w3, and w4.

- Big picture, here’s what we need to figure out:

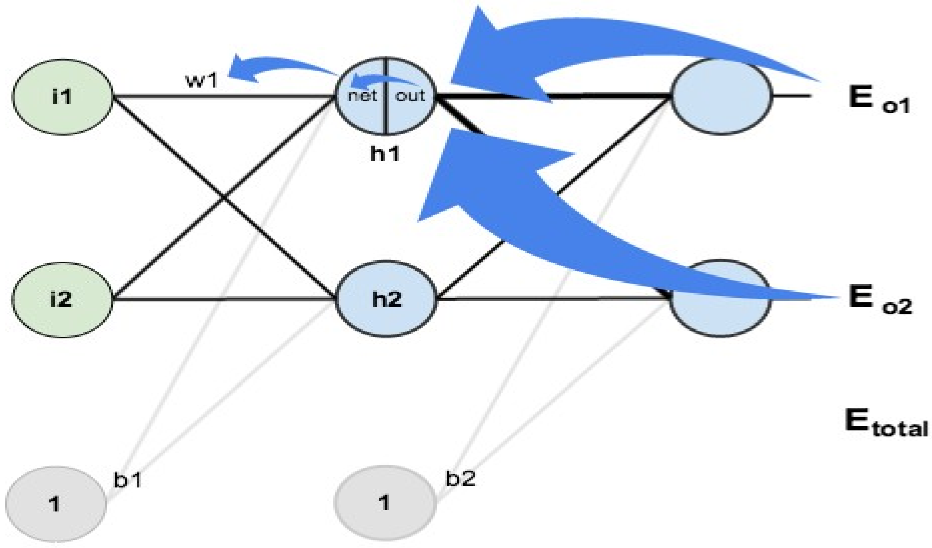
$$\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$$

Hidden Layer

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$E_{total} = E_{o1} + E_{o2}$

Hidden Layer

- We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.
- We know that out_{h1} affects both out_{o1} and out_{o2} therefore the $\delta E_{total} / \delta out_{h1}$ needs to take into consideration its effect on the both output neurons:

$$\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1}$$

Hidden Layer

- Starting with $\delta E_{o1} / \delta out_{h1}$:

  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1}$

- We can calculate $\delta E_{o1} / \delta net_{o1}$ using values we calculated earlier:

  $\delta E_{o1} / \delta net_{o1} = \delta E_{o1} / \delta out_{o1} * \delta out_{o1} / \delta net_{o1} = 0.74136507 * 0.186815602 = 0.138498562$

- And $\delta net_{o1} / \delta out_{h1}$ is equal to w5:

  $net_{o1} = w5 * out_{h1} + w6 * out_{h2} + b2 * 1$

  $\delta net_{o1} / \delta out_{h1} = w5 = 0.40$

- Plugging them in:

  $\delta E_{o1} / \delta out_{h1} = \delta E_{o1} / \delta net_{o1} * \delta net_{o1} / \delta out_{h1} = 0.138498562 * 0.40 = 0.055399425$

Hidden Layer

- Following the same process for $\delta E_{o2} / \delta out_{h1}$, we get:

  $\delta E_{o2} / \delta out_{h1} = -0.019049119$

- Therefore:

  $\delta E_{total} / \delta out_{h1} = \delta E_{o1} / \delta out_{h1} + \delta E_{o2} / \delta out_{h1} = 0.055399425 + -0.019049119 = 0.036350306$

- Now that we have $\delta E_{total} / \delta out_{h1}$, we need to figure out $\delta out_{h1} / \delta net_{h1}$ and then $\delta net_{h1} / \delta w$ for each weight:

  $out_{h1} = 1/(1+e^{\wedge}(-net_{h1}))$

  $\delta out_{h1} / \delta net_{h1} = out_{h1} (1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$

Hidden Layer

- We calculate the partial derivative of the total net input to h1 with respect to w1 the same as we did for the output neuron:

  $net_{h1} = w1 * i1 + w2 * i2 + b1 * 1$

  $\delta net_{h1} / \delta w1 = i1 = 0.05$

- Putting it all together:

  $\delta E_{total} / \delta w1 = \delta E_{total} / \delta out_{h1} * \delta out_{h1} / \delta net_{h1} * \delta net_{h1} / \delta w1$

  $\delta E_{total} / \delta w1 = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$

Hidden Layer

- We can now update w1:

  $w1 = w1 - \eta * \delta E_{total} / \delta w1 = 0.15 - 0.5 * 0.000438568 = 0.149780716$

- Repeating this for w2, w3, and w4 :

  w2 = 0.19956143

  w3 = 0.24975114

  w4 = 0.29950229

Backpropagation

- Finally, we’ve updated all of our weights!
- When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109.
- After this first round of backpropagation, the total error is now down to 0.291027924.
- It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.000035085.
- At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Backpropagation

- Network with two inputs, one output, one hidden layer of one neuron.
- Inputs are 0.1 and 0.5, desired output is 0.2.
- Write code for the forward pass.
- Compute the error.
- Backpropagate the error.
- Train the network.

Backpropagation

- Represent the connections between two layers with a matrix of weights.

- Train a network using matrices.

Keras

Keras

- Keras is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

- Use Keras if you need a deep learning library that:

  Allows for easy and fast prototyping (through user friendliness, modularity, and extensibility).

  Supports both convolutional networks and recurrent networks, as well as combinations of the two.

  Runs seamlessly on CPU and GPU.

Keras

- The core data structure of Keras is a model, a way to organize layers. The simplest type of model is the Sequential model, a linear stack of layers.

```
from keras.models import Sequential
model = Sequential()
```

Keras

- In order to define a network you have to import the libraries for defining the layers and the libraries for the training algorithms :

```
from keras.layers.core import Dense, Activation

from keras.optimizers import SGD
```

Keras

- Example

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

Keras

Stacking layers is as easy as .add():

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(units=64, input_dim=100))
model.add(Activation(‘tanh'))
model.add(Dense(units=10))
model.add(Activation('softmax'))
```

Keras

- Input and output are represented as arrays :

```
import numpy as np

X = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y = np.array([[0.],[1.],[1.],[0.]])
```

Keras

```
sgd = SGD(lr=0.1)

model.compile(loss='binary_crossentropy',
optimizer=sgd)

model.fit(X, y, verbose=1, batch_size=1,
epochs=1000)

print(model.predict(X))
```

Keras

- Cross-entropy loss :
  $C = -\Sigma_n \Sigma_i y_{in} \log o_{in}$
  where o_{in} is your network's output for class i, and $y_{in} = 1$ if example n is of class i and 0 if it has some other class j.
  $dC / do_{in} = -y_{in} / o_{in}$

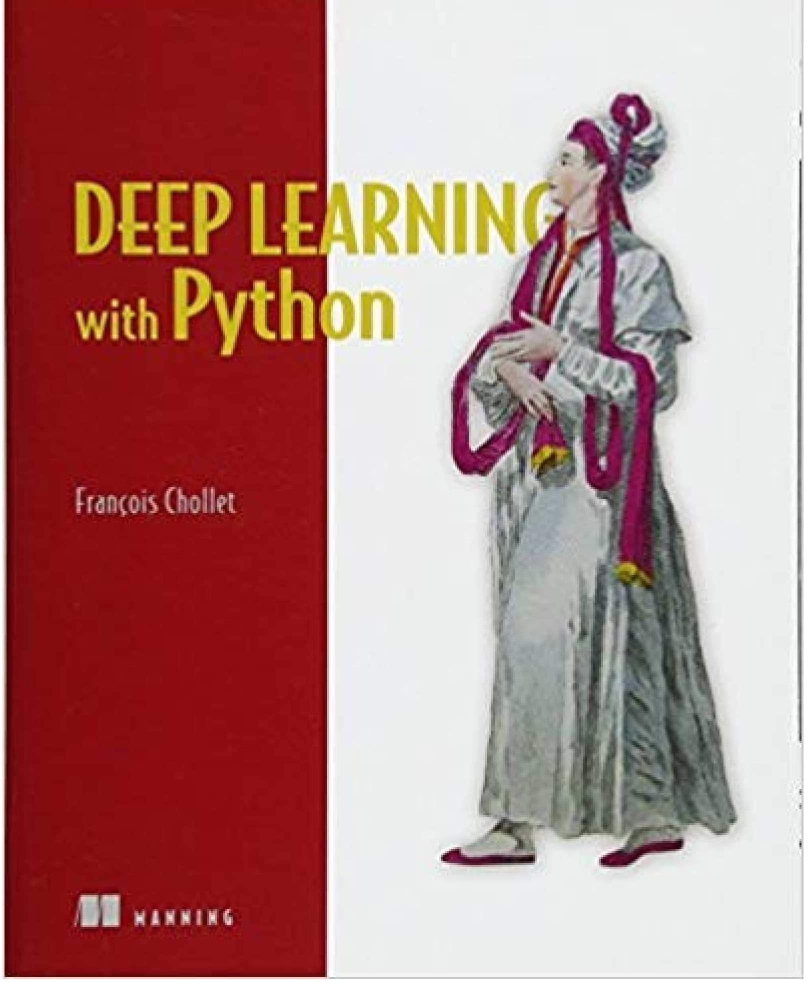
- Binary cross-entropy loss :
  $C = -\Sigma_n \Sigma_i y_{in} \log o_{in} + (1 – y_{in}) \log (1 - o_{in})$

- Mean squared error loss :
  $C' = \Sigma_n \Sigma_i (y_{in} - o_{in})^2$
  $dC' / do_{in} = -2 (y_{in} - o_{in})$.

- For a binary classification binary cross entropy.
- For a multi-class classification softmax + cross entropy.
- For regression mean squared error.

Practical Work

- Implement a two layers XOR network (one hidden layer).
- Make it learn the XOR function with two inputs and one output.



DEEP LEARNING
with Python

François Chollet

MANNING

MNIST

The problem we're trying to solve here is to classify grayscale images of handwritten digits (28 × 28 pixels) into their 10 categories (0 through 9). We'll use the MNIST dataset, a classic in the machine-learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "hello world" of deep learning—it's what you do to verify that your algorithms are working as expected. As you become a machine-learning practitioner, you'll see MNIST come up over and over again in scientific papers, blog posts, and so on. You can see some MNIST samples in figure 2.1.

Classes and labels

In machine learning, a *category* in a classification problem is called a *class*. Data points are called *samples*. The class associated with a specific sample is called a *label*.

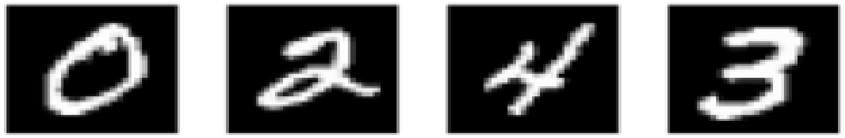


Figure 2.1 MNIST sample digits

MNIST

- Loading the MNIST data :

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

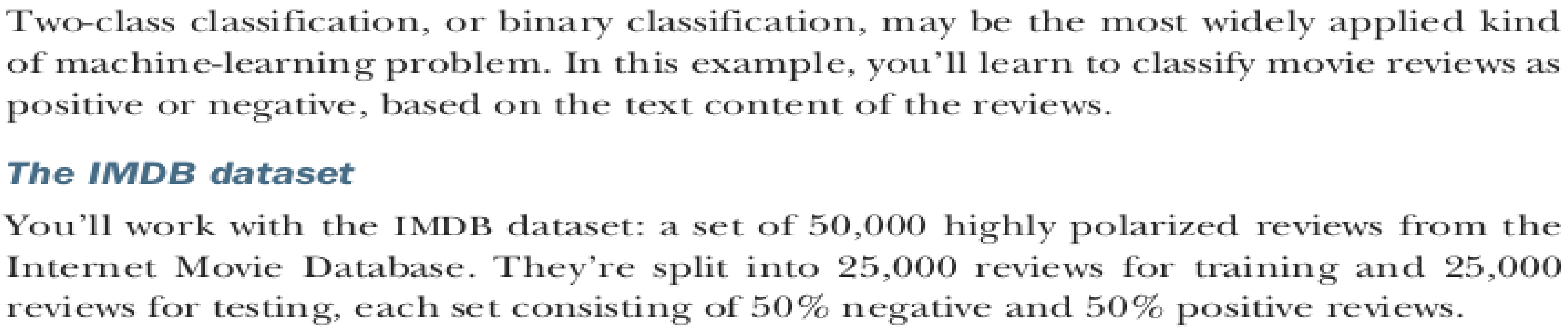
Exercise

Train a small dense network on the MNIST data :

- Prepare the data :
  - Inputs = vectors of real numbers (between 0.0 and 1.0) of size 28*28
  - Outputs = vectors of real numbers of size 10 (nine 0 and a 1 at the index of the label)
- Define the network :
  - Fully connected network with 28*28 inputs and 10 outputs
- Define the loss and the optimizer
- Train the network
- Test the network
- Print an image in the test set and the predicted class

Binary Classification

Binary Classification



Two-class classification, or binary classification, may be the most widely applied kind of machine-learning problem. In this example, you'll learn to classify movie reviews as positive or negative, based on the text content of the reviews.

The IMDB dataset

You'll work with the IMDB dataset: a set of 50,000 highly polarized 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.

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

Binary Classification

- The argument num_words = 10000 means you’ll keep only the top 10,000 most frequently occurring words in the training data. Rare words will be discarded. This allows you to work with vector data of a 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 :

```
train_data[0]

[1, 14, 22, 16, ... 178, 32]

train_labels[0]

1
```

Exercise : Preparing the data

- You can’t feed lists of integers into a neural network. You have to turn your lists into tensors :
  - 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.

Exercise

Train a small dense network on the IMDB data :

- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

Weight Regularization

Weight Regularization

- You may be familiar with the principle of Occam’s razor : given two explanations for something, the explanation most likely to be correct is the simplest one—the one that makes fewer assumptions. This idea also applies to the models learned by neural networks: given some training data and a network architecture, multiple sets of weight values (multiple models) could explain the data. Simpler models are less likely to overfit than complex ones.

- A simple model in this context is a model where the distribution of parameter values has less entropy (or a model with fewer parameters). Thus, a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it’s done by adding to the loss function of the network a cost associated with having large weights.

Weight Regularization

- L2 regularization : The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks.
- In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments :

```
from keras import regularizers
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu',
                                    input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001), activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- regularizer_l2(0.001) means every coefficient in the weight matrix of the layer will add 0.001 * weight_coefficient_value to the total loss of the network. Note that because this penalty is only added at training time, the loss for this network will be much higher at training time than at test time.

Exercise

Compare for the IMDB network the effect of weight regularization on the evolution of training and test errors.

Dropout

Dropout

- Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Geoff Hinton and his students at the University of Toronto.
- Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.
- Let’s say a given layer would normally return a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, [0, 0.5, 1.3, 0, 1.1] .
- The dropout rate is the fraction of the features that are zeroed out; it’s usually set between 0.2 and 0.5.
- At test time, no units are dropped out; instead, the layer’s output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time.

Dropout

- Consider a matrix containing the output of a layer, layer_output , of shape (batch_size, features) .
- At training time, we zero out at random a fraction of the values in the matrix:

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
```

- At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):

```
layer_output *= 0.5
```

- Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it’s implemented in practice :

```
layer_output *= np.random.randint(0, high=2, size=layer_output.shape)
layer_output /= 0.5
```

Tips For Using Dropout

- Generally, use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A probability too low has minimal effect and a value too high results in under-learning by the network.

- Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.

Dropout

- Dropout

  keras.layers.core.Dropout(rate, noise_shape=None, seed=None)

- Applies Dropout to the input.

  Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.

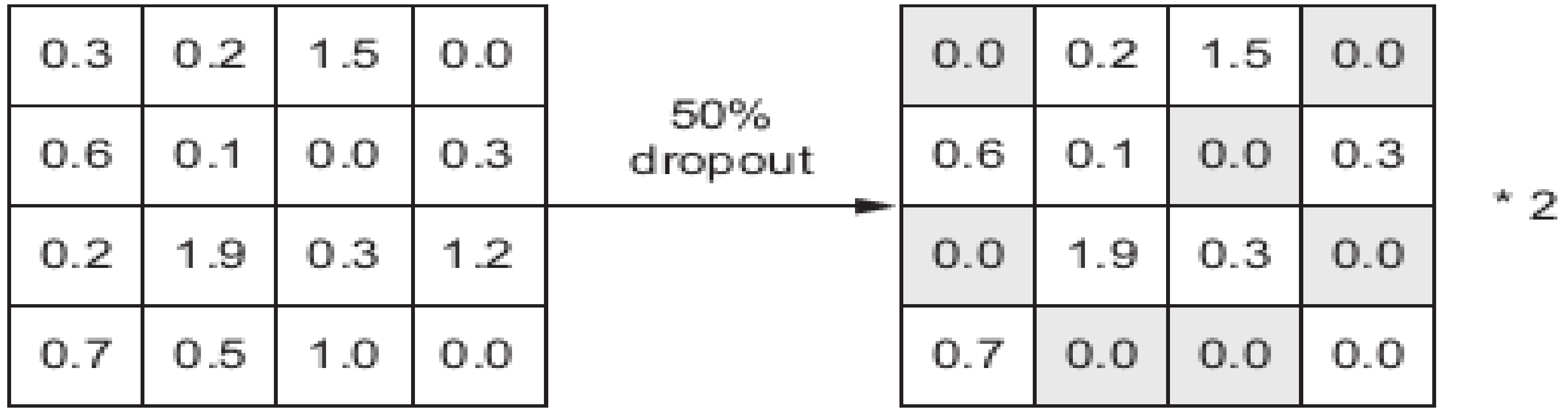
- Arguments

  rate: float between 0 and 1. Fraction of the input units to drop.

  noise_shape: 1D integer tensor representing the shape of the binary dropout mask that will be multiplied with the input. For instance, if your inputs have shape (batch_size, timesteps, features) and you want the dropout mask to be the same for all timesteps, you can use noise_shape=(batch_size, 1, features).

  seed: A Python integer to use as random seed.

Dropout



0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

Dropout

- Add two dropout layers in the IMDB network to see how well they do at reducing overfitting.

Avoid Overfitting

- To recap, these are the most common ways to prevent overfitting in neural networks:
  - Get more training data.
  - Reduce the capacity of the network.
  - Add weight regularization.
  - Add dropout.

Multiclass Classification

Multiclass Classification

- You’ll work with the Reuters dataset, a set of short newswires and their topics, published by Reuters in 1986. It’s a simple, widely used toy dataset for text classification. There are 46 different topics; some topics are more represented than others, but each topic has at least 10 examples in the training set.

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

Exercise

Train a small dense network on the Reuters data :

- Encode the data
- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

Reuters

- Further experiments :
- Train on less epochs
- Try using larger or smaller layers: 32 units, 128 units, and so on.
- The network used two hidden layers. Now try using a single hidden layer, or three hidden layers.

Multiclass classification

Here’s what you should take away from this example:

- 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 so that it will output a probability distribution over the N output classes.
- Categorical crossentropy is almost always the loss function you should use for such problems. It minimizes the distance between the probability distributions output by the network and the true distribution of the targets.
- Encoding the labels via categorical encoding (also known as one-hot encoding) and using categorical_crossentropy as a loss function
- If you need to classify data into a large number of categories, you should avoid creating information bottlenecks in your network due to intermediate layers that are too small.

Regression

Regression

- The Boston Housing Price dataset
- Predict the median price of homes in a given Boston suburb in the mid-1970s, given data points about the suburb at the time, such as the crime rate, the local property tax rate, and so on. The dataset you’ll use has an interesting difference from the two previous examples. It has relatively few data points: only 506, split between 404 training samples and 102 test samples. And each feature in the input data (for example, the crime rate) has a different scale. For instance, some values are proportions, which take values between 0 and 1; others take values between 1 and 12, others between 0 and 100, and so on.

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

Exercise

Train a small dense network on the Boston Housing data :

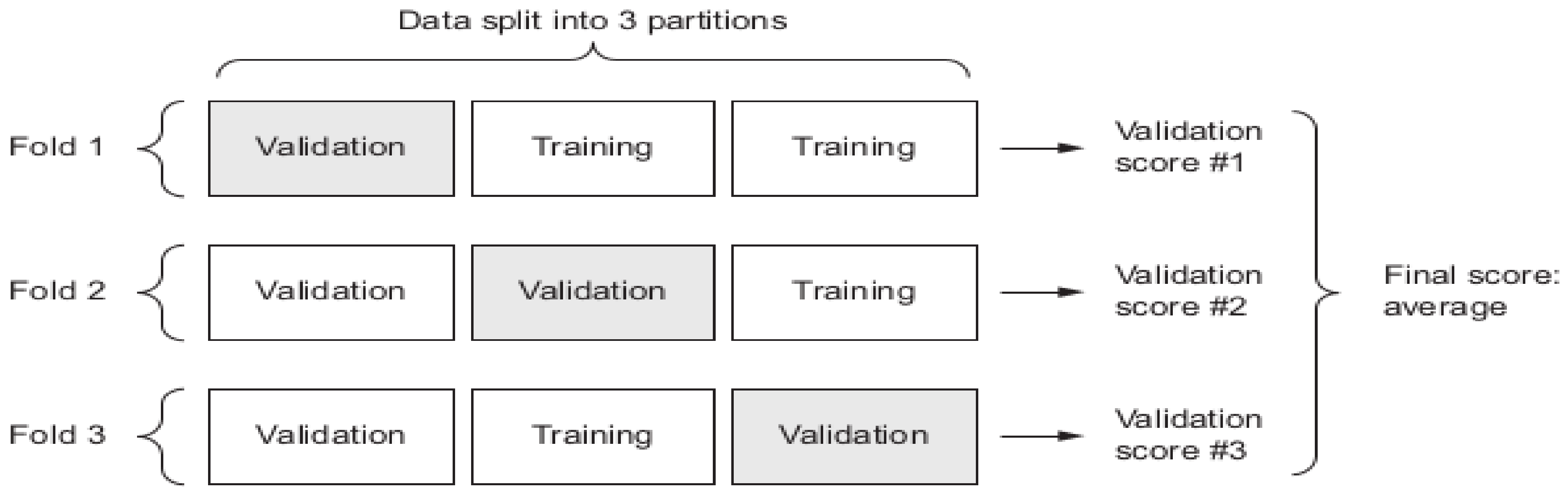
- Encode the data
- Define the network
- Define the loss and the optimizer
- Define a validation set
- Train the network using a validation set

Regression

- In order to find the best hyperparameters (epochs, number of hidden layers, number of neurons,…) we will evaluate the hyperparameters with cross validation.
- The goal is to find hyperparameters that minimize the cross validation error.
- Once the hyperparameters have been chosen, the network with these hyperparameters is initialized and trained on the whole training set.

Regression

- 3-fold cross-validation :



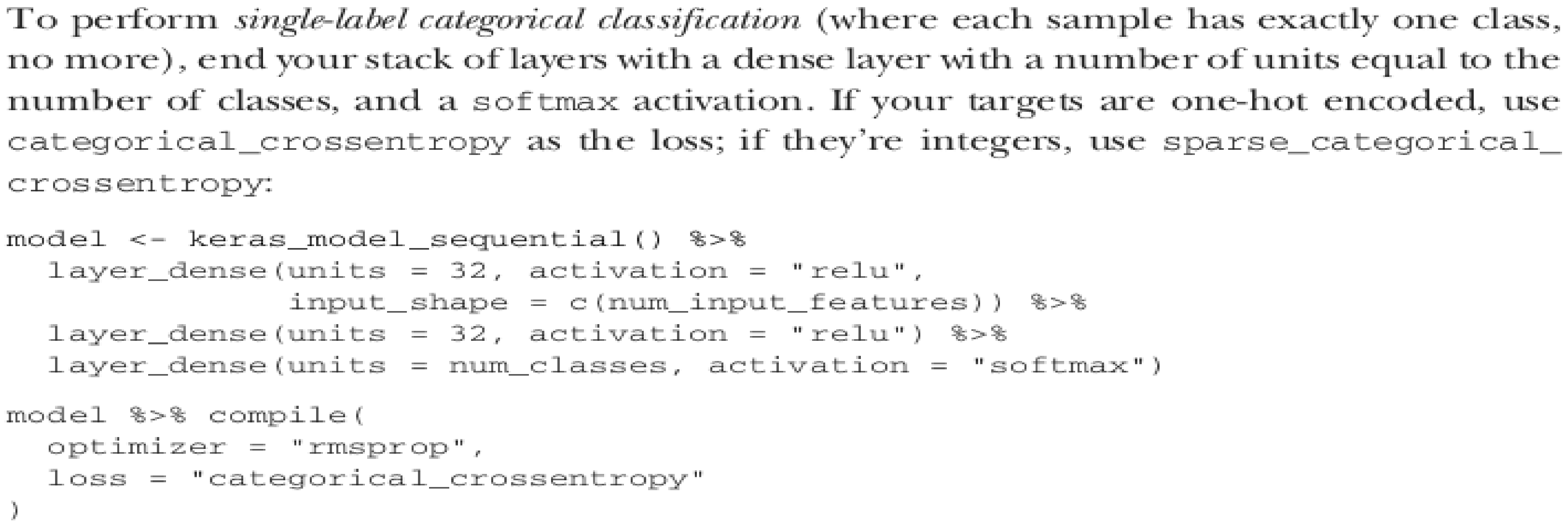
Regression

Here’s what you should take away from this example:

- Regression is done using different loss functions than classification. Mean squared error (MSE) is a loss function commonly used for regression.
- Similarly, evaluation metrics to be used for regression differ from those used for classification; naturally, the concept of accuracy doesn’t apply for regression. A common regression metric is mean absolute error (MAE).
- When features in the input data have values in different ranges, each feature should be scaled independently as a preprocessing step.
- When there is little data available, using K-fold validation is a great way to reliably evaluate a model.
- When little training data is available, it’s preferable to use a small network with few hidden layers (typically only one or two), in order to avoid severe overfitting.

What we have learned

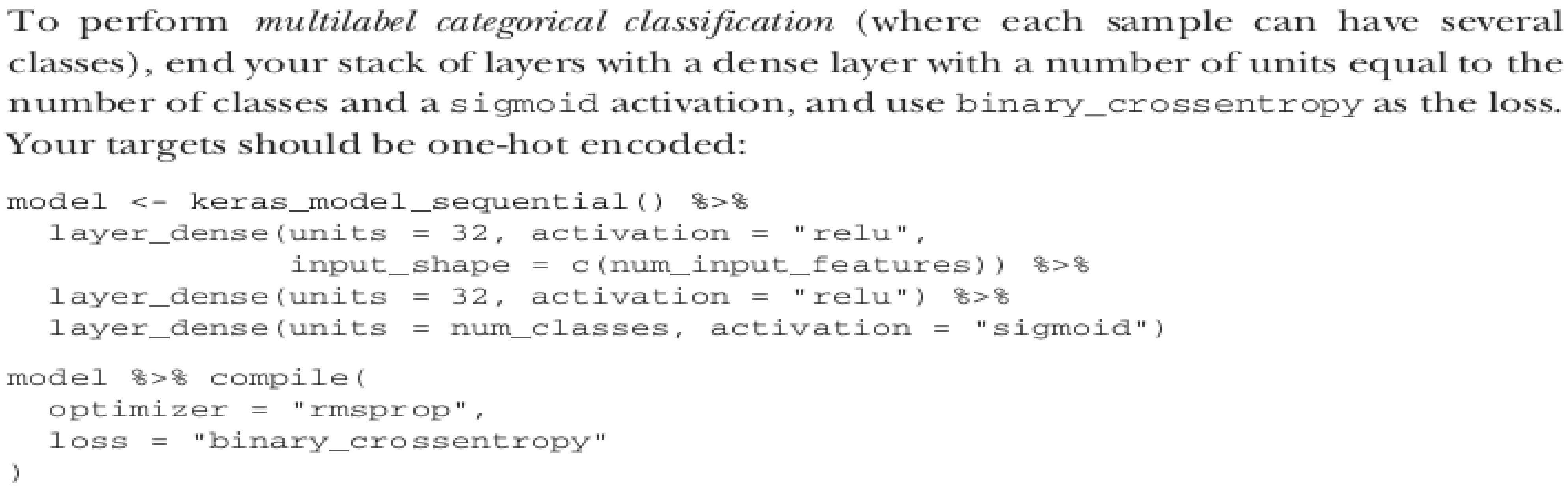
What we have learned



To perform *single-label categorical classification* (where each sample has exactly one class, no more), end your stack of layers with a dense layer with a number of units equal to the number of classes, and a `softmax` activation. If your targets are one-hot encoded, use `categorical_crossentropy` as the loss; if they're integers, use `sparse_categorical_crossentropy`:

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = c(num_input_features)) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "softmax")

model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy"
)
```

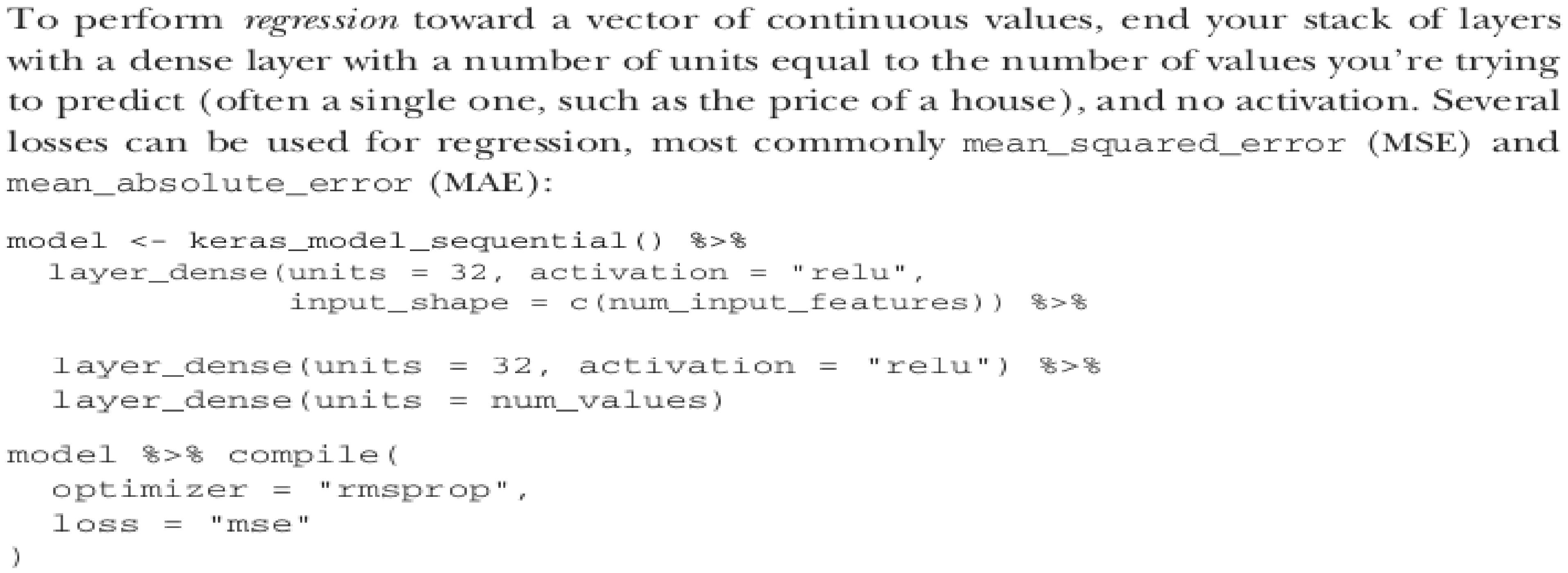
What we have learned



To perform *multilabel categorical classification* (where each sample can have several classes), end your stack of layers with a dense layer with a number of units equal to the number of classes and a `sigmoid` activation, and use `binary_crossentropy` as the loss. Your targets should be one-hot encoded:

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = c(num_input_features)) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = num_classes, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy"
)
```

What we have learned

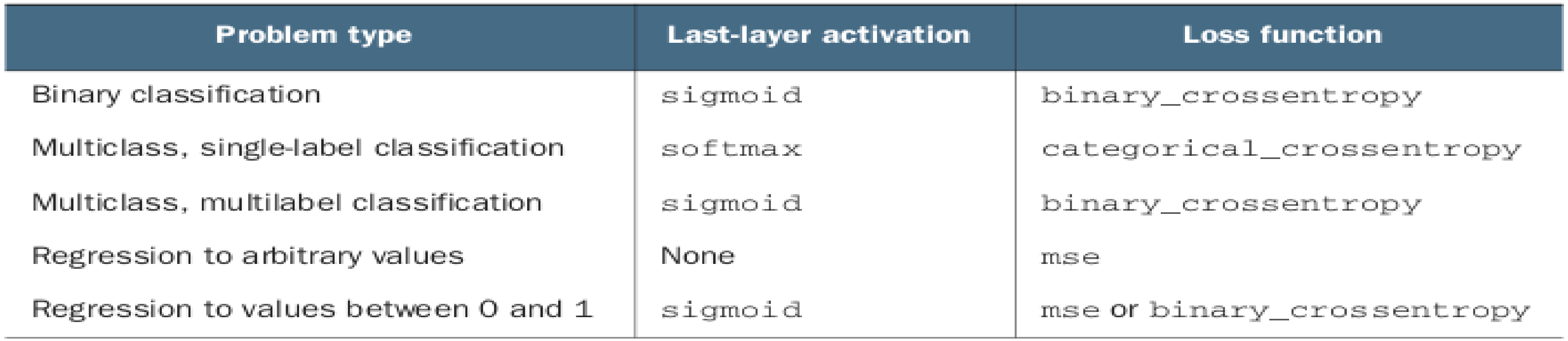


To perform *regression* toward a vector of continuous values, end your stack of layers with a dense layer with a number of units equal to the number of values you're trying to predict (often a single one, such as the price of a house), and no activation. Several losses can be used for regression, most commonly `mean_squared_error` (MSE) and `mean_absolute_error` (MAE):

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = c(num_input_features)) %>%

  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = num_values)

model %>% compile(
  optimizer = "rmsprop",
  loss = "mse"
)
```

What we have learned

- Choosing the right last-layer activation and loss function for your model :



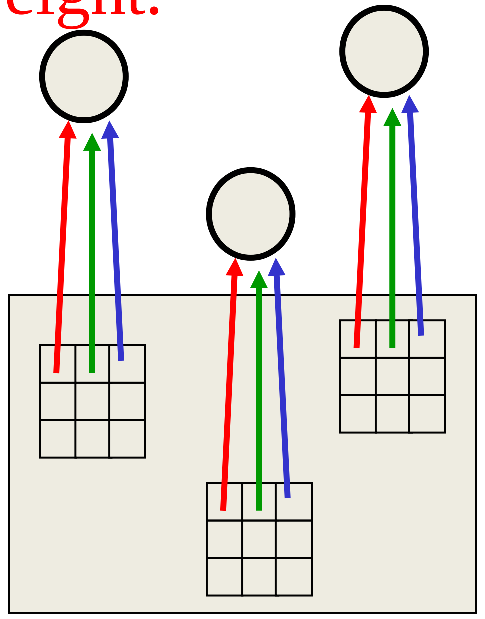
Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Convolutional Neural Networks

The replicated feature approach
(currently the dominant approach for neural networks)

- Use many different copies of the same feature detector with different positions.
  - Could also replicate across scale and orientation (tricky and expensive)
  - Replication greatly reduces the number of free parameters to be learned.
- Use several different feature types, each with its own map of replicated detectors.
  - Allows each patch of image to be represented in several ways.

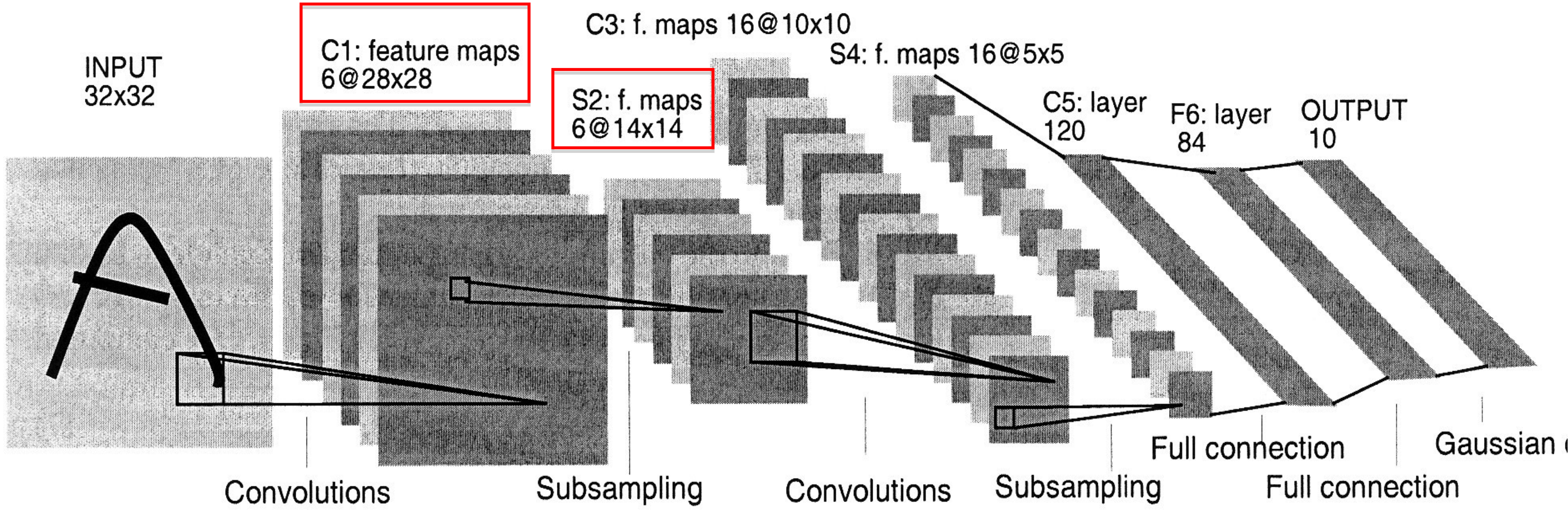
The red connections all have the same weight.

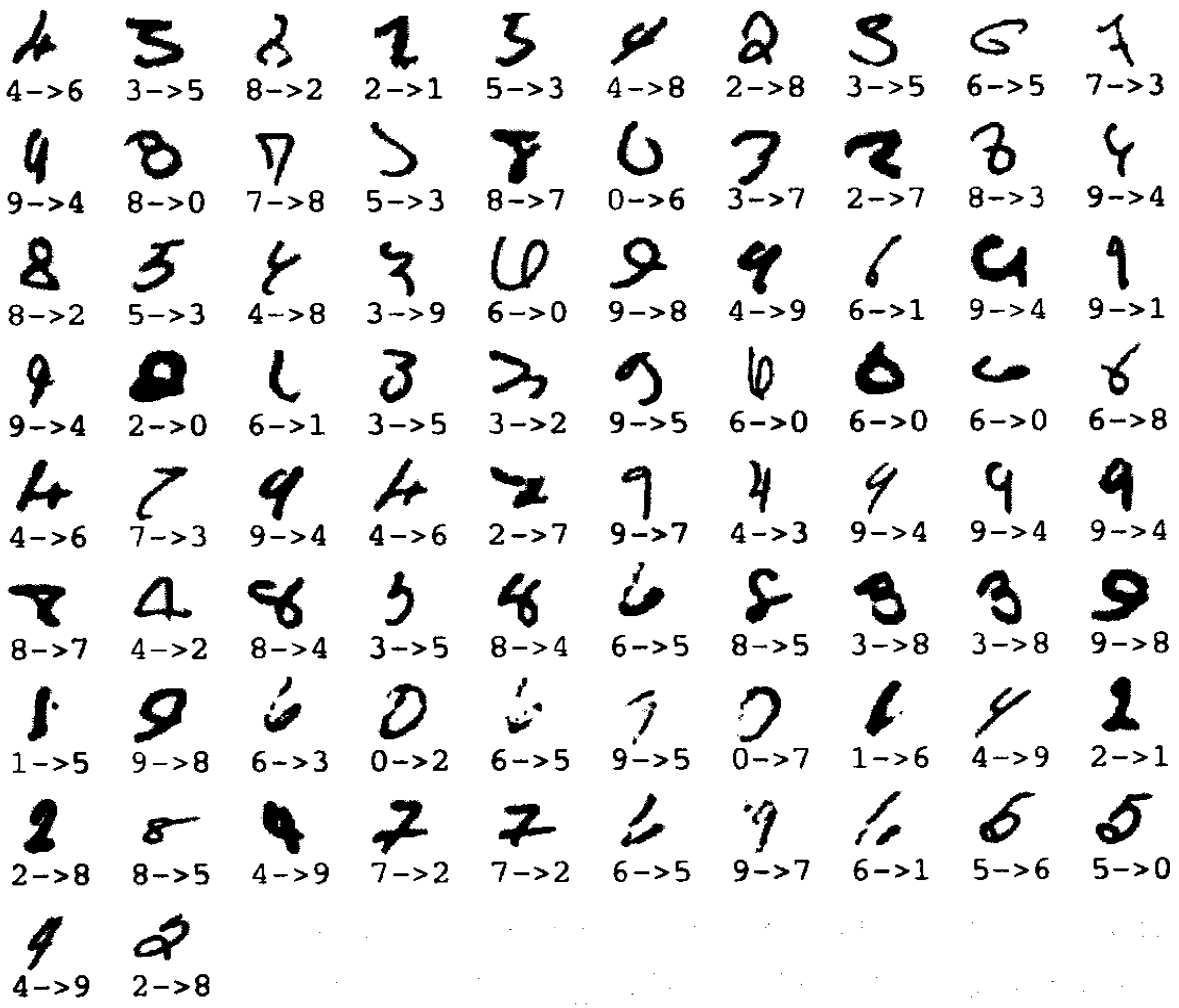


Le Net

- Yann LeCun and his collaborators developed a really good recognizer for handwritten digits by using backpropagation in a feedforward net with:
  - Many hidden layers
  - Many maps of replicated units in each layer.
  - Pooling of the outputs of nearby replicated units.
  - A wide net that can cope with several characters at once even if they overlap.
  - A clever way of training a complete system, not just a recognizer.
- This net was used for reading ~10% of the checks in North America.
- Look the impressive demos of LENET at http://yann.lecun.com

The architecture of LeNet5





The 82 errors made by LeNet5

Notice that most of the errors are cases that people find quite easy.

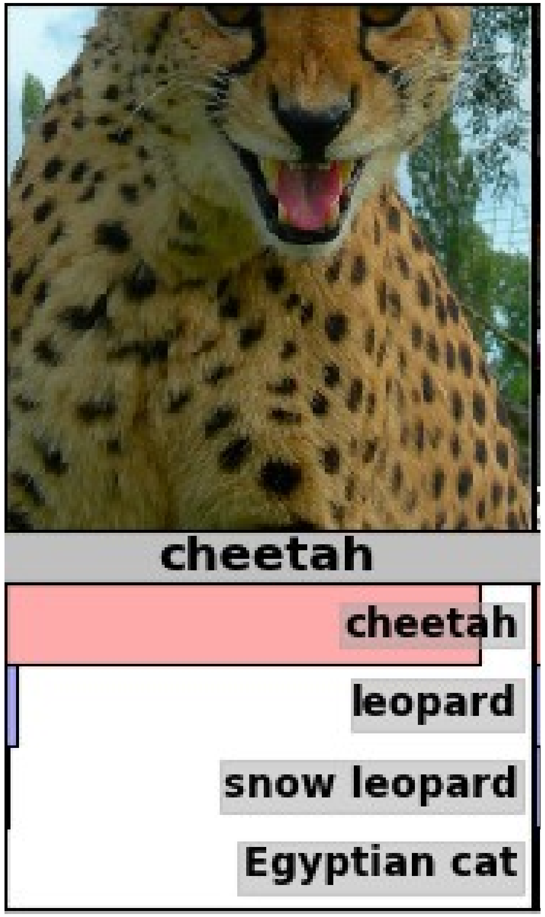
The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.

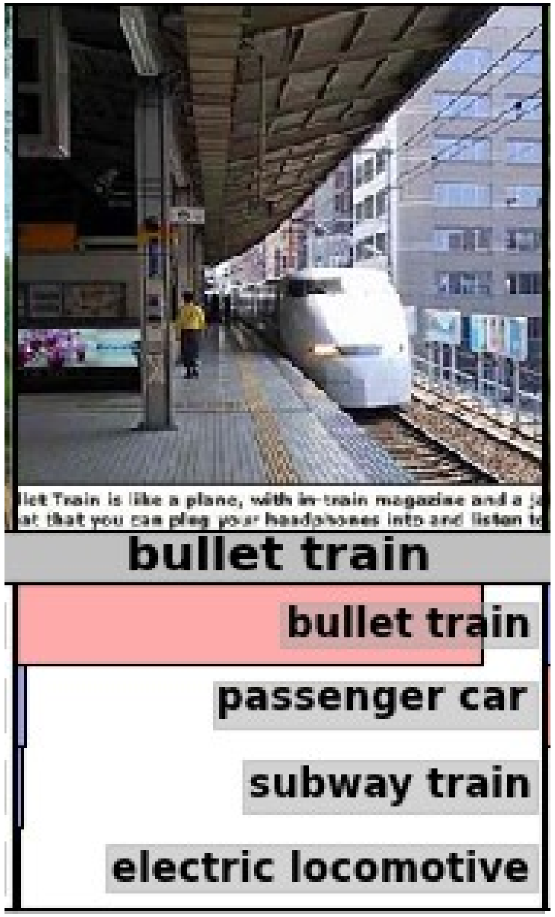
The ILSVRC-2012 competition on ImageNet

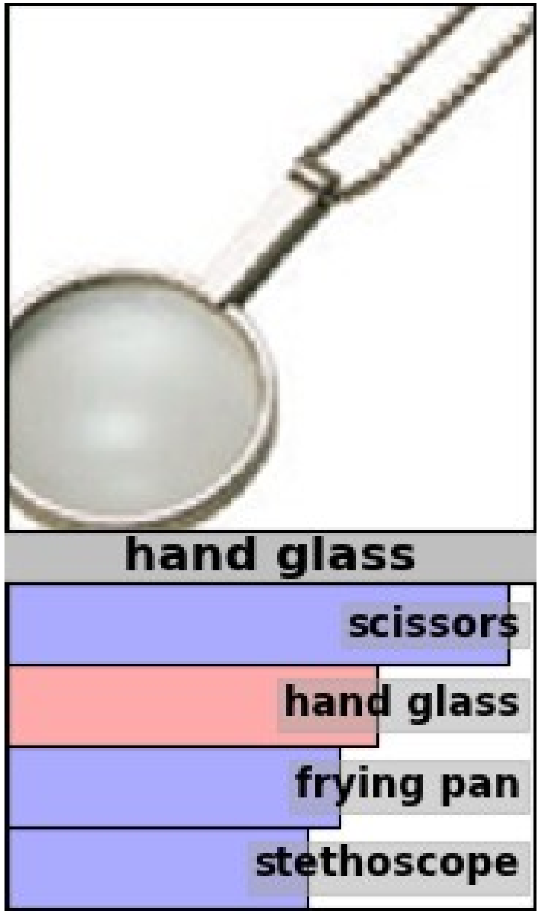
- The dataset has 1.2 million high-resolution training images.
- The classification task:
  - Get the “correct” class in your top 5 bets. There are 1000 classes.
- The localization task:
  - For each bet, put a box around the object. Your box must have at least 50% overlap with the correct box.

- Some of the best existing computer vision methods were tried on this dataset by leading computer vision groups from Oxford, INRIA, XRCE, …
  - Computer vision systems use complicated multi-stage systems.
  - The early stages are typically hand-tuned by optimizing a few parameters.

Examples from the test set (with the network's guesses)







Error rates on the ILSVRC-2012 competition

	classification	classification &localization
University of Toronto (Alex Krizhevsky)	16.4%	34.1%
University of Tokyo	26.1%	53.6%
Oxford University Computer Vision Group	26.9%	50.0%
INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe)	27.0%	
University of Amsterdam	29.5%	

A neural network for ImageNet

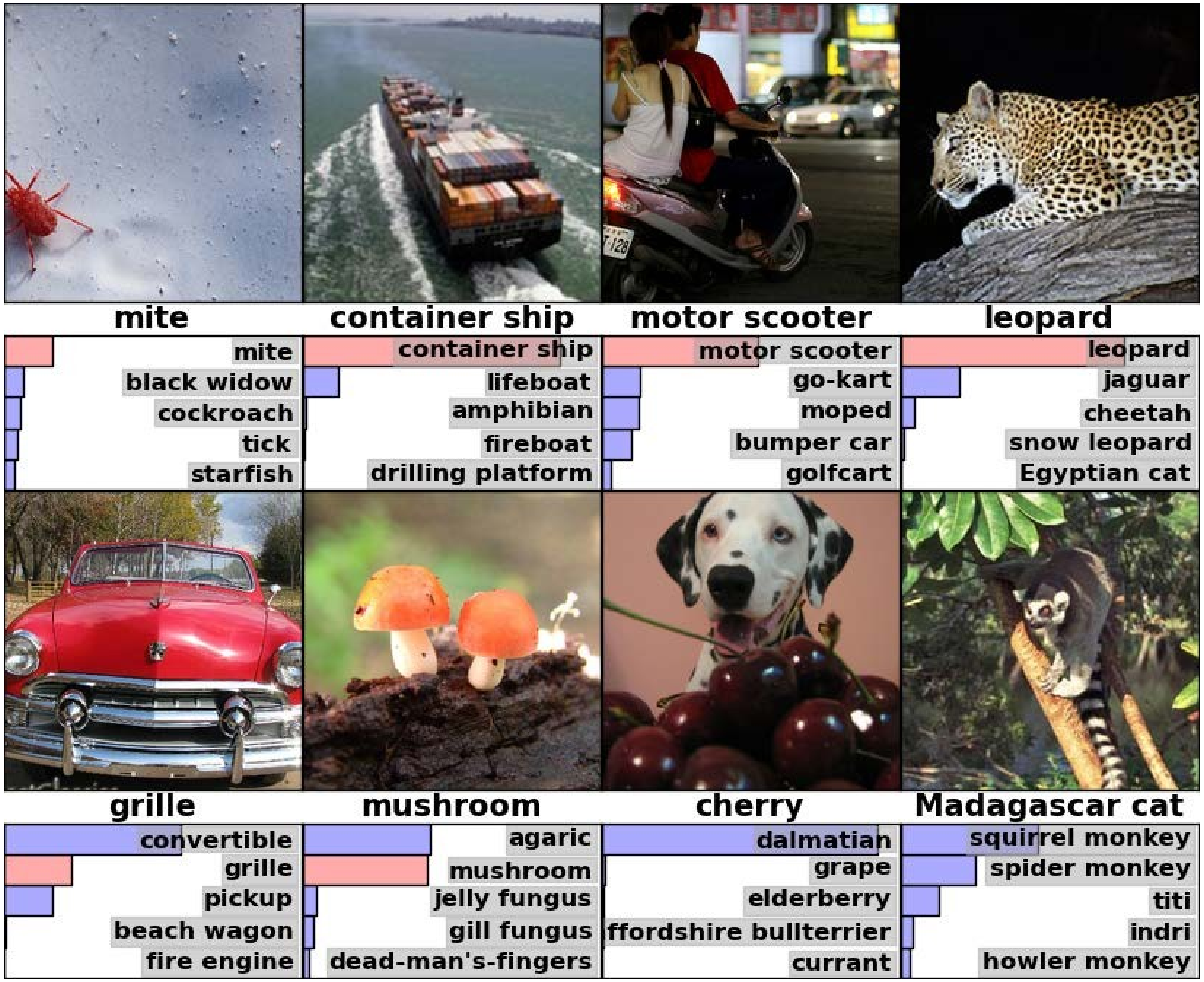
- Alex Krizhevsky (NIPS 2012) developed a very deep convolutional neural net of the type pioneered by Yann Le Cun. Its architecture was:
  - 7 hidden layers not counting some max pooling layers.
  - The early layers were convolutional.
  - The last two layers were globally connected.

- The activation functions were:
  - Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.
  - Competitive normalization to suppress hidden activities when nearby units have stronger activities. This helps with variations in intensity.

Tricks that significantly improve generalization

- Train on random 224x224 patches from the 256x256 images to get more data. Also use left-right reflections of the images.
  - At test time, combine the opinions from ten different patches: The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.

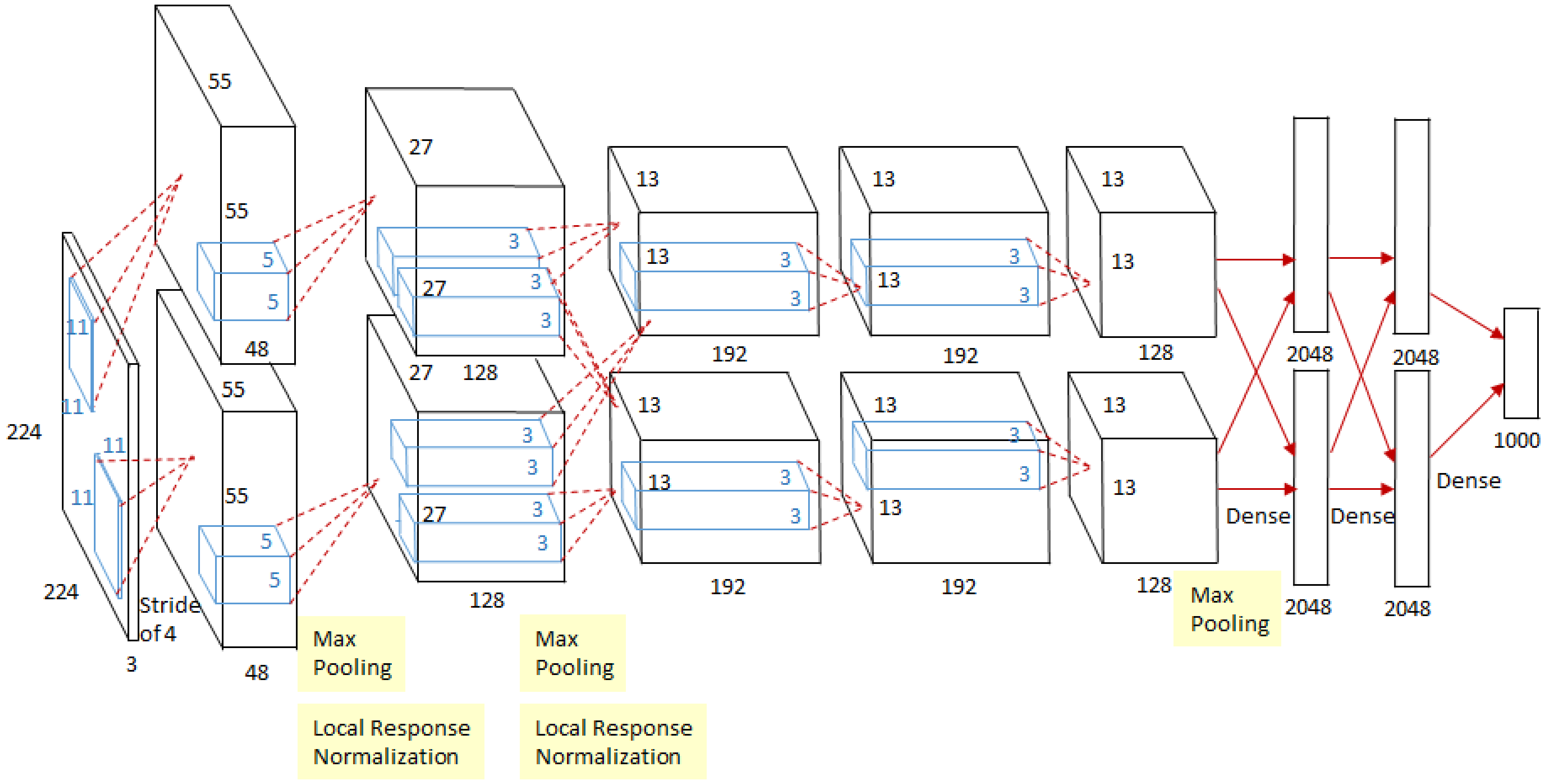
- Use “dropout” to regularize the weights in the globally connected layers (which contain most of the parameters).
  - Dropout means that half of the hidden units in a layer are randomly removed for each training example.
  - This stops hidden units from relying too much on other hidden units.



Some more examples of how well the deep net works for object recognition.

The hardware required for Alex's net

- He uses a very efficient implementation of convolutional nets on two Nvidia GTX 580 Graphics Processor Units (over 1000 fast little cores)
  - GPUs are very good for matrix-matrix multiplies.
  - GPUs have very high bandwidth to memory.
  - This allows him to train the network in a week.
  - It also makes it quick to combine results from 10 patches at test time.
- We can spread a network over many cores if we can communicate the states fast enough.
- As cores get cheaper and datasets get bigger, big neural nets will improve faster than old-fashioned (*i.e.* pre Oct 2012) computer vision systems.


224
224
3
11
11
11
11
Stride of 4
55
55
48
5
5
55
55
48
5
5
Max Pooling
Local Response Normalization
27
27
128
3
3
3
27
128
3
3
3
3
Max Pooling
Local Response Normalization
13
13
3
3
192
13
13
3
3
192
13
13
3
3
192
13
13
3
3
192
13
13
128
13
13
128
Max Pooling
Dense
Dense
Dense
2048
2048
2048
2048
1000

Convolutional Networks

- Conv2D

  keras.layers.convolutional.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)

  2D convolution layer (e.g. spatial convolution over images).

  This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. If use_bias is True, a bias vector is created and added to the outputs. Finally, if activation is not None, it is applied to the outputs as well.

  When using this layer as the first layer in a model, provide the keyword argument input_shape (tuple of integers, does not include the sample axis), e.g. input_shape=(128, 128, 3) for 128x128 RGB pictures in data_format="channels_last".

Convolutional Networks

- Input shape

  4D tensor with shape: (samples, rows, cols, channels)

- Output shape

  4D tensor with shape: (samples, new_rows, new_cols, filters)
  rows and cols values might have changed due to padding.

Convolutional Networks

- ReLU layer :
- ReLU is the abbreviation of Rectified Linear Units. This is a layer of neurons that applies the non-saturating activation function f (x) = max (0, x).
- Compared to other functions the usage of ReLU is preferable, because it results in the neural network training several times faster, without making a significant difference to generalisation accuracy.
- activation = ‘relu’ in the layer parameters.

Convolutional Networks

- Softmax :
- Applies the Softmax function to an n-dimensional input Tensor, rescaling them so that the elements of the n-dimensional output Tensor lie in the range (0,1) and sum to 1.
- Softmax is defined as exp(xi) / Σ exp(xj)
- activation = ‘softmax‘ in the layer parameters.

Convolutional Networks

- Reshape

  keras.layers.core.Reshape(target_shape)

  Reshapes an output to a certain shape.

- Arguments

  target_shape: target shape. Tuple of integers, does not include the samples dimension (batch size).

- Input shape

  Arbitrary, although all dimensions in the input shaped must be fixed. Use the keyword argument input_shape (tuple of integers, does not include the samples axis) when using this layer as the first layer in a model.

- Output shape

  (batch_size,) + target_shape

Convolutional Networks

- Example

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

Convolutional Networks

- Flatten

  keras.layers.core.Flatten()

  Flattens the input. Does not affect the batch size.

- Example

```
model = Sequential()
model.add(Convolution2D(64, 3, 3,
          border_mode='same',
          input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

MNIST

```
from keras.datasets import mnist

from keras.layers import Dense, Flatten
from keras.layers import Conv2D

# the data, shuffled and split between train and test sets
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Practical Work

- Implement a convolutional network for MNIST.
- 3x3 filters
- 32 planes for the first layer
- 64 planes for the second layer
- Fully connected layer with 128 neurons
- 10 classes for the output layer
- ReLU
- Softmax

Max Pooling

Max Pooling

- Max pooling is a sample-based discretization process. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing it's dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

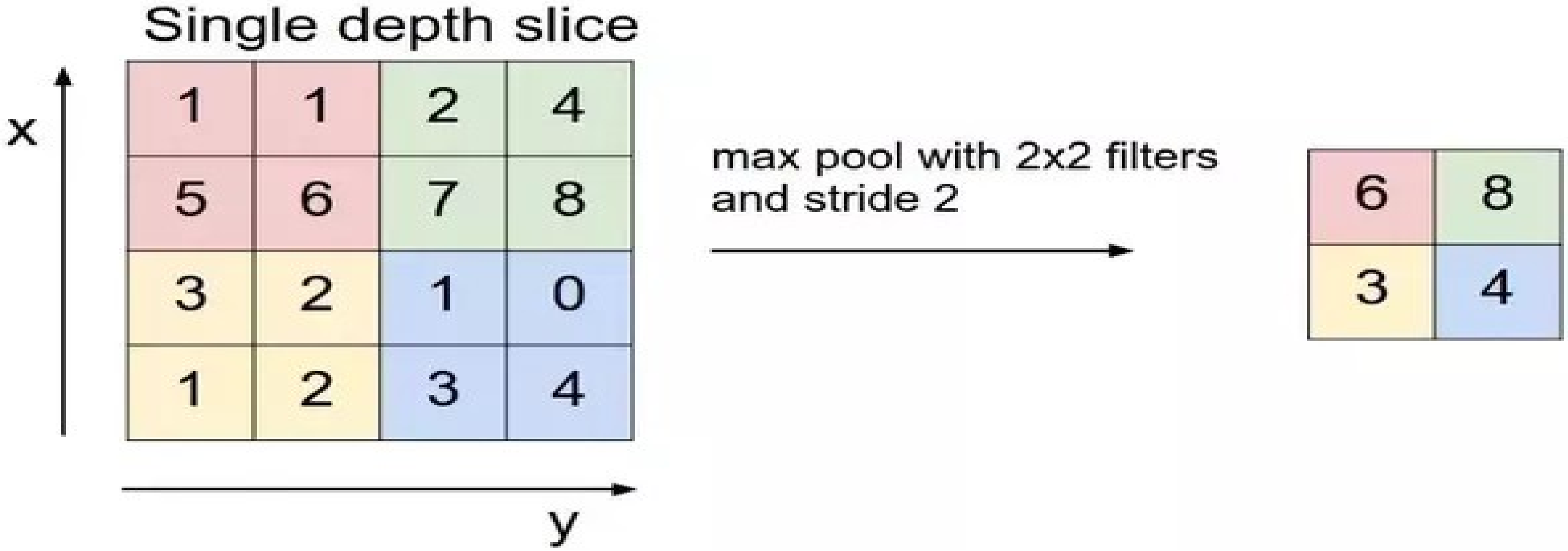
- This is done to in part to help over-fitting by providing an abstracted form of the representation. As well, it reduces the computational cost by reducing the number of parameters to learn and provides basic translation invariance to the internal representation.

- Max pooling is done by applying a max filter to non-overlapping sub-regions of the initial representation.

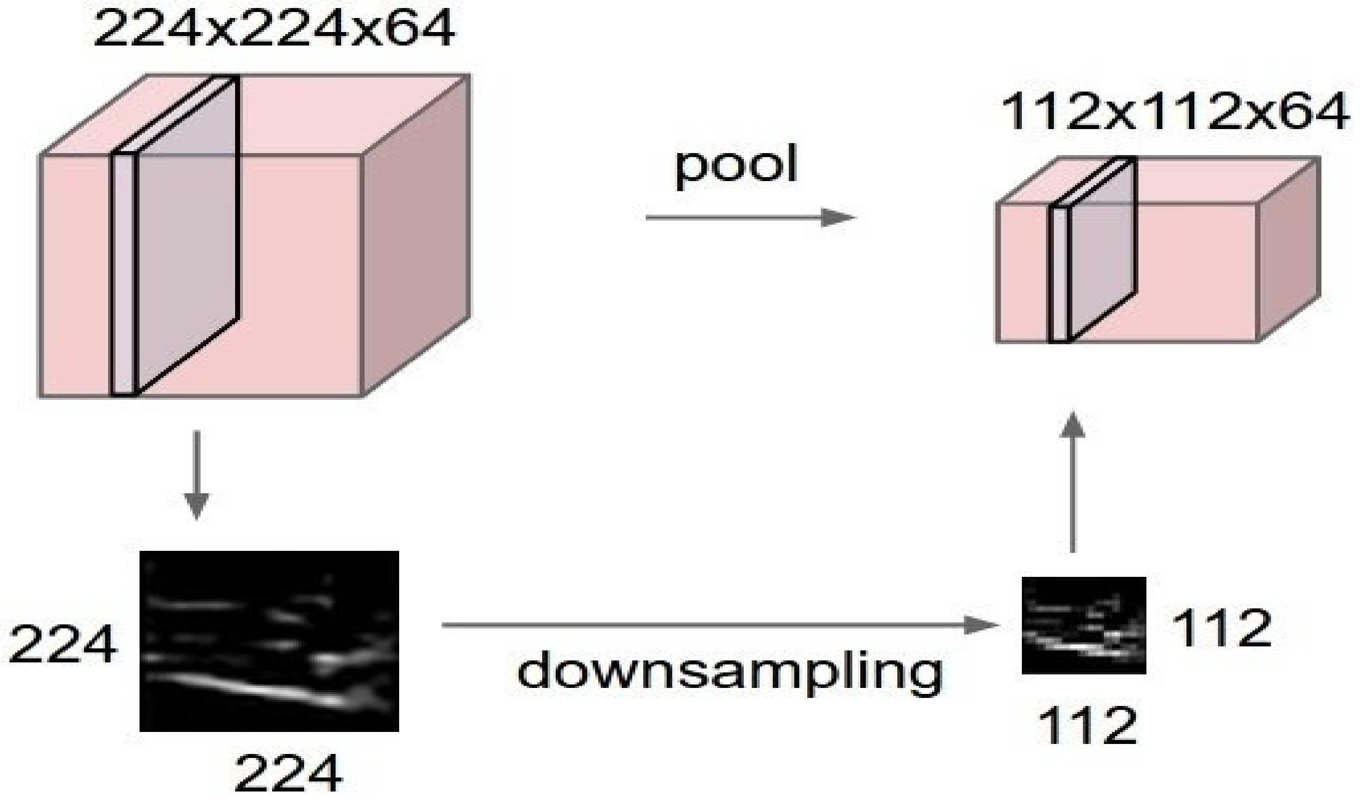
Max Pooling

- For example a 4x4 matrix representing our initial input.
- We run a 2x2 filter over our input.
- Use a stride of 2 (meaning the (dx, dy) for stepping over our input will be (2, 2)) and won't overlap regions.
- For each of the regions represented by the filter, we will take the max of that region and create a new, output matrix where each element is the max of a region in the original input.

Max Pooling


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters and stride 2
6 8
3 4

Max Pooling



Max Pooling

- MaxPooling2D

  keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)

  Max pooling operation for spatial data.

- Arguments

  pool_size: integer or tuple of 2 integers, factors by which to downscale (vertical, horizontal). (2, 2) will halve the input in both spatial dimension. If only one integer is specified, the same window length will be used for both dimensions.

  strides: Integer, tuple of 2 integers, or None. Strides values. If None, it will default to pool_size.

  padding: One of "valid" or "same" (case-insensitive).

  data_format: A string, one of channels_last (default) or channels_first. The ordering of the dimensions in the inputs. channels_last corresponds to inputs with shape (batch, height, width, channels) while channels_first corresponds to inputs with shape (batch, channels, height, width). It defaults to the image_data_format value found in your Keras config file at ~/.keras/keras.json. If you never set it, then it will be "channels_last".

Max Pooling

- Input shape

  4D tensor with shape: (batch_size, rows, cols, channels)

- Output shape

  4D tensor with shape: (batch_size, pooled_rows, pooled_cols, channels)

Dropout

- Dropout is a technique where randomly selected neurons are ignored during training.
- Their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.
- You can imagine that if neurons are randomly dropped out of the network during training, that other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.
- The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data.

Practical Work

- Add Dropout and MaxPooling to the MNIST network.
- Train a network on the CIFAR10 image dataset.
- Add Dropout and MaxPooling to the CIFAR10 network.

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

The Functional API

The Functional API

- The Functional API is a way to create models that is more flexible than Sequential:
- It can handle:
  - models with non-linear topology,
  - models with shared layers,
  - models with multiple inputs or outputs.

The Functional API

- It's based on the idea that a deep learning model is usually a directed acyclic graph (DAG) of layers.
- The Functional API a set of tools for building graphs of layers.

The Functional API

- Consider the following model:

  (input: 784-dimensional vectors)

  [Dense (64 units, relu activation)]

  [Dense (64 units, relu activation)]

  [Dense (10 units, softmax activation)]

  (output: probability distribution over 10 classes)

- It is a simple graph of 3 layers.

The Functional API

- To build this model with the functional API, you would start by creating an input node:

```
from tensorflow import keras
inputs = keras.Input(shape=(784,))
```

- Here we just specify the shape of our data: 784-dimensional vectors.
- Note that the batch size is always omitted, we only specify the shape of each sample.
- For an input meant for images of shape (32, 32, 3), we would have used:

```
img_inputs = keras.Input(shape=(32, 32, 3))
```

- What gets returned, inputs, contains information about the shape and dtype of the input data that you expect to feed to your model:

```
inputs.shape = TensorShape([None, 784])
inputs.dtype = tf.float32
```

The Functional API

- You create a new node in the graph of layers by calling a layer on this inputs object:

```
from tensorflow.keras import layers

dense = layers.Dense(64, activation='relu')
x = dense(inputs)
```

- The "layer call" action is like drawing an arrow from "inputs" to this layer we created.
- We're "passing" the inputs to the dense layer, and out we get x.

The Functional API

- Let's add a few more layers to our graph of layers:

```
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)
```

- At this point, we can create a Model by specifying its inputs and outputs in the graph of layers:

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

The Functional API

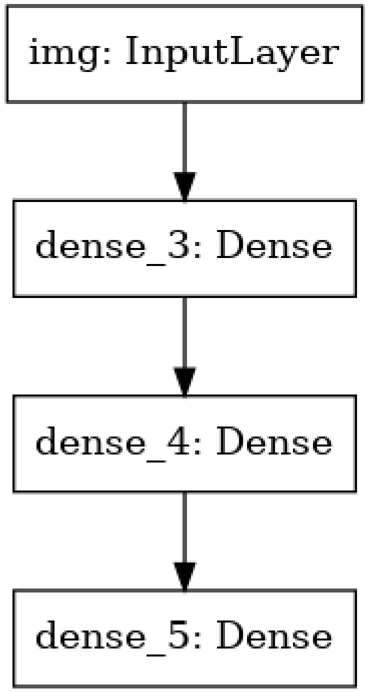
- To recap, here is our full model definition process:

```
inputs = keras.Input(shape=(784,), name='img')
x = layers.Dense(64, activation='relu')(inputs)
x = layers.Dense(64, activation='relu')(x)
outputs = layers.Dense(10, activation='softmax')(x)

model = keras.Model(inputs=inputs, outputs=outputs,
name='mnist_model')
```

The Functional API

- We can also plot the model as a graph:

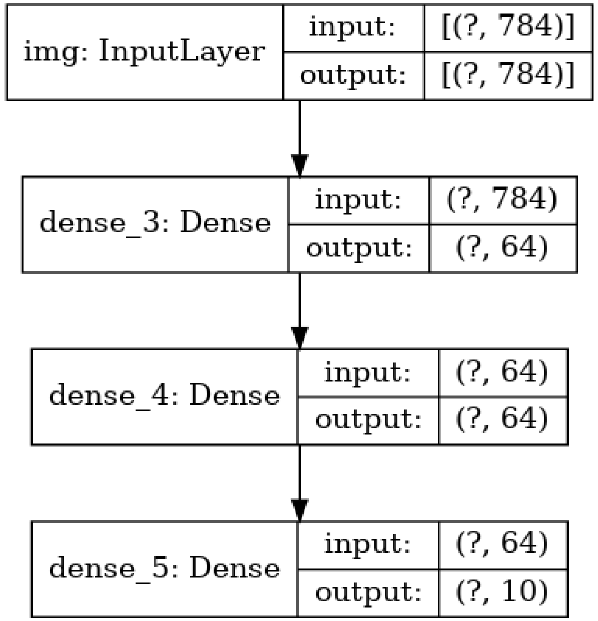
```
keras.utils.plot_model(model, 'my_first_model.png')
```



The Functional API

- And optionally display the input and output shapes of each layer in the plotted graph:

  keras.utils.plot_model(model, 'my_first_model_with_shape_info.png', show_shapes=True)



The Functional API

- Training, evaluation, and inference work exactly in the same way for models built using the Functional API as for Sequential models.
- Here is a quick demonstration.
- Here we load MNIST image data, reshape it into vectors, fit the model on the data (while monitoring performance on a validation split), and finally we evaluate our model on the test data:

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

model.compile(loss='sparse_categorical_crossentropy',
              optimizer=keras.optimizers.RMSprop(),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,
                    batch_size=64,
                    epochs=5,
                    validation_split=0.2)
test_scores = model.evaluate(x_test, y_test, verbose=2)
print('Test loss:', test_scores[0])
print('Test accuracy:', test_scores[1])
```

The Functional API

- Saving and serialization work exactly in the same way for models built using the Functional API as for Sequential models.

- The standard way to save a Functional model is to call model.save() to save the whole model into a single file.

- You can later recreate the same model from this file, even if you no longer have access to the code that created the model.

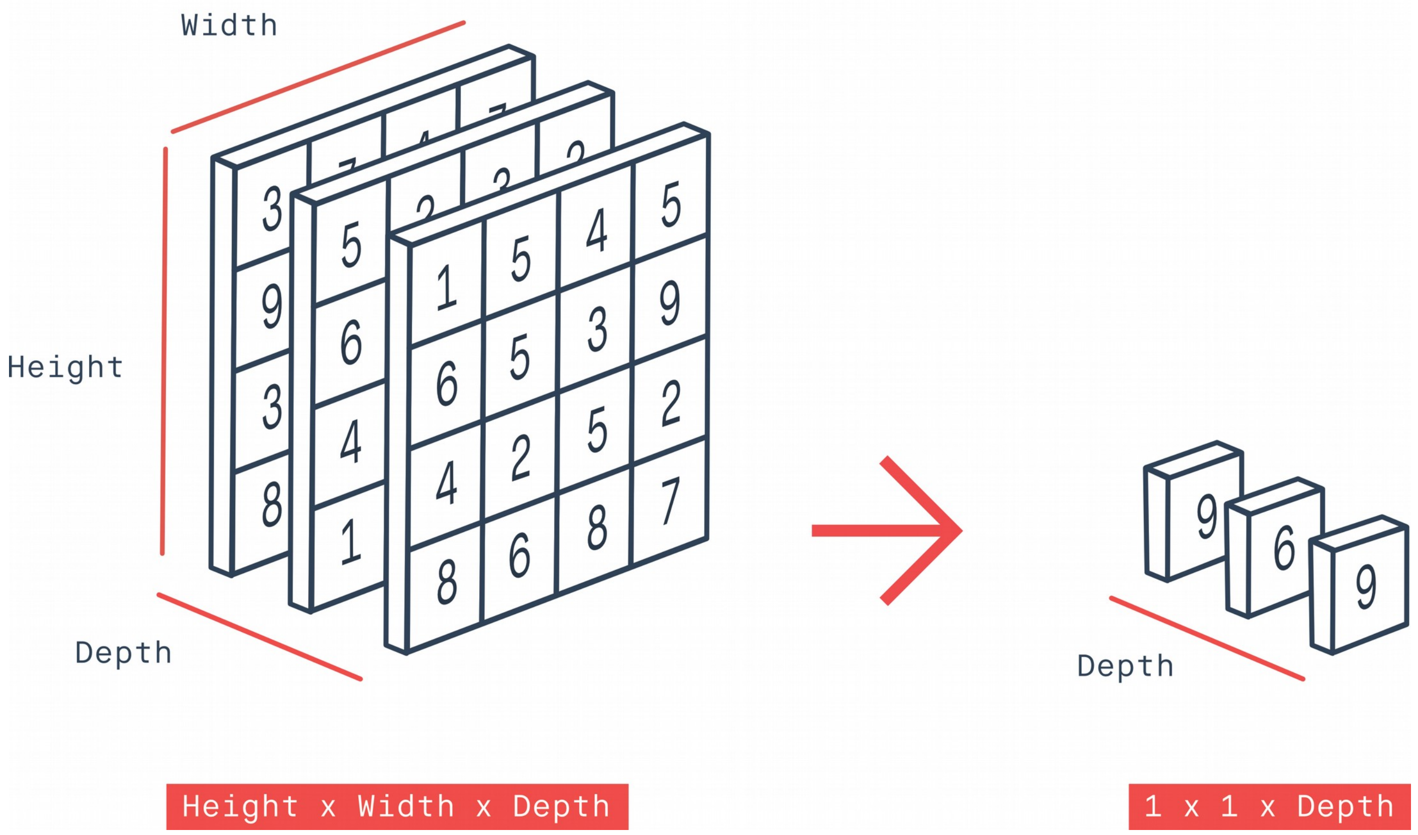
- This file includes: - The model's architecture - The model's weight values (which were learned during training) - The model's training config (what you passed to compile), if any - The optimizer and its state, if any (this enables you to restart training where you left off)

```
model.save('path_to_my_model.h5')
# Recreate the exact same model purely from the file:
model = keras.models.load_model('path_to_my_model.h5')
```

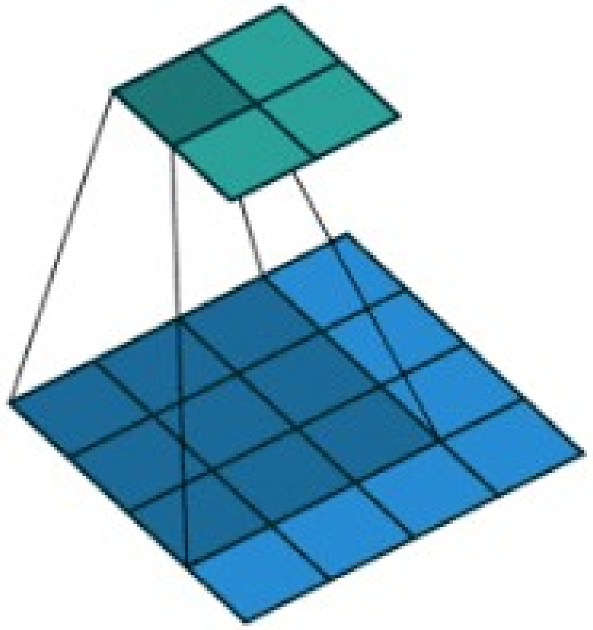
Exercise

- Train a MNIST network defined with the functional API.

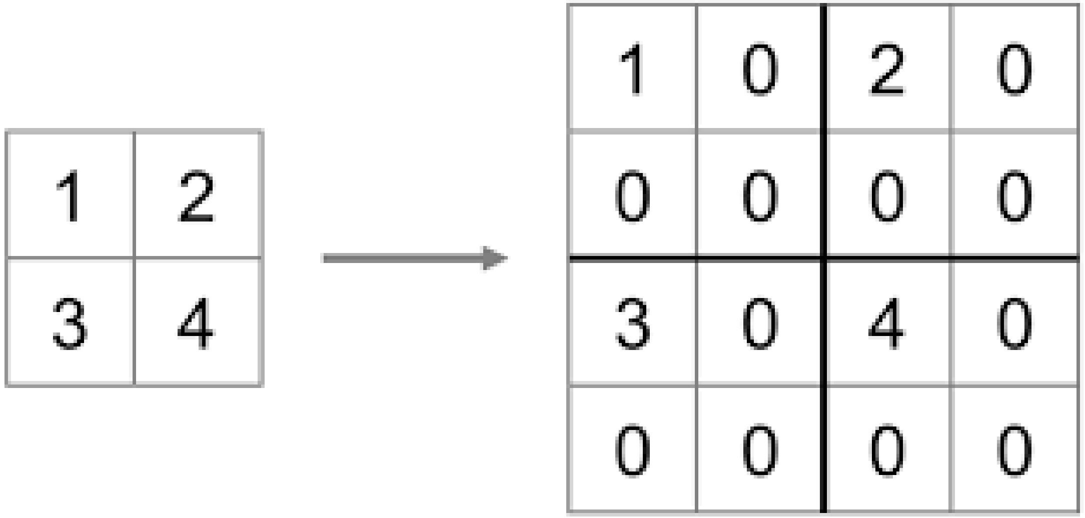
GlobalMaxPooling2D



Conv2DTranspose



UpSampling2D



- Nearest-Neighbor: Copies the value from the nearest pixel.
- Bilinear: Uses all nearby pixels to calculate the pixel's value, using linear interpolations.

Autoencoder

- In the functional API, models are created by specifying their inputs and outputs in a graph of layers. That means that a single graph of layers can be used to generate multiple models.

- In the example below, we use the same stack of layers to instantiate two models: an encoder model that turns image inputs into 16-dimensional vectors, and an end-to-end autoencoder model for training.

```
encoder_input = keras.Input(shape=(28, 28, 1), name='img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')
encoder.summary()
```

Autoencoder

```
x = layers.Reshape((4, 4, 1))(encoder_output)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu')(x)
x = layers.UpSampling2D(3)(x)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
decoder_output = layers.Conv2DTranspose(1, 3, activation='relu')(x)

autoencoder = keras.Model(encoder_input, decoder_output, name='autoencoder')
autoencoder.summary()
```

- Note that we make the decoding architecture strictly symmetrical to the encoding architecture, so that we get an output shape that is the same as the input shape (28, 28, 1).

- The reverse of a Conv2D layer is a Conv2DTranspose layer, and the reverse of a MaxPooling2D layer is an UpSampling2D layer.

Autoencoder

- You can treat any model as if it were a layer, by calling it on an Input or on the output of another layer.
- Note that by calling a model you aren't just reusing the architecture of the model, you're also reusing its weights.
- Let's see this in action. Here's a different take on the autoencoder example that creates an encoder model, a decoder model, and chain them in two calls to obtain the autoencoder model:

```
encoder_input = keras.Input(shape=(28, 28, 1), name='original_img')
x = layers.Conv2D(16, 3, activation='relu')(encoder_input)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.MaxPooling2D(3)(x)
x = layers.Conv2D(32, 3, activation='relu')(x)
x = layers.Conv2D(16, 3, activation='relu')(x)
encoder_output = layers.GlobalMaxPooling2D()(x)

encoder = keras.Model(encoder_input, encoder_output, name='encoder')
encoder.summary()
```

Autoencoder

```
decoder_input = keras.Input(shape=(16,), name='encoded_img')
x = layers.Reshape((4, 4, 1))(decoder_input)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
x = layers.Conv2DTranspose(32, 3, activation='relu')(x)
x = layers.UpSampling2D(3)(x)
x = layers.Conv2DTranspose(16, 3, activation='relu')(x)
decoder_output = layers.Conv2DTranspose(1, 3, activation='relu')(x)

decoder = keras.Model(decoder_input, decoder_output, name='decoder')
decoder.summary()

autoencoder_input = keras.Input(shape=(28, 28, 1), name='img')
encoded_img = encoder(autoencoder_input)
decoded_img = decoder(encoded_img)
autoencoder = keras.Model(autoencoder_input, decoded_img, name='autoencoder')
autoencoder.summary()
```

Exercise

- Train a an autoencoder for MNIST with the functional API.

Ensemble

- As you can see, model can be nested: a model can contain submodels (since a model is just like a layer).
- A common use case for model nesting is ensembling. As an example, here's how to ensemble a set of models into a single model that averages their predictions:

```
def get_model():
  inputs = keras.Input(shape=(128,))
  outputs = layers.Dense(1, activation='sigmoid')(inputs)
  return keras.Model(inputs, outputs)

model1 = get_model()
model2 = get_model()
model3 = get_model()

inputs = keras.Input(shape=(128,))
y1 = model1(inputs)
y2 = model2(inputs)
y3 = model3(inputs)
outputs = layers.average([y1, y2, y3])
ensemble_model = keras.Model(inputs=inputs, outputs=outputs)
```

Multiple Inputs and Outputs

- Models with multiple inputs and outputs
- The functional API makes it easy to manipulate multiple inputs and outputs. This cannot be handled with the Sequential API.
- Let's say you're building a system for ranking custom issue tickets by priority and routing them to the right department.

- Your model will have 3 inputs:

  Title of the ticket (text input)
  Text body of the ticket (text input)
  Any tags added by the user (categorical input)

- It will have two outputs:

  Priority score between 0 and 1 (scalar sigmoid output)
  The department that should handle the ticket (softmax output over the set of departments)

Multiple Inputs and Outputs

- Let's built this model in a few lines with the Functional API.

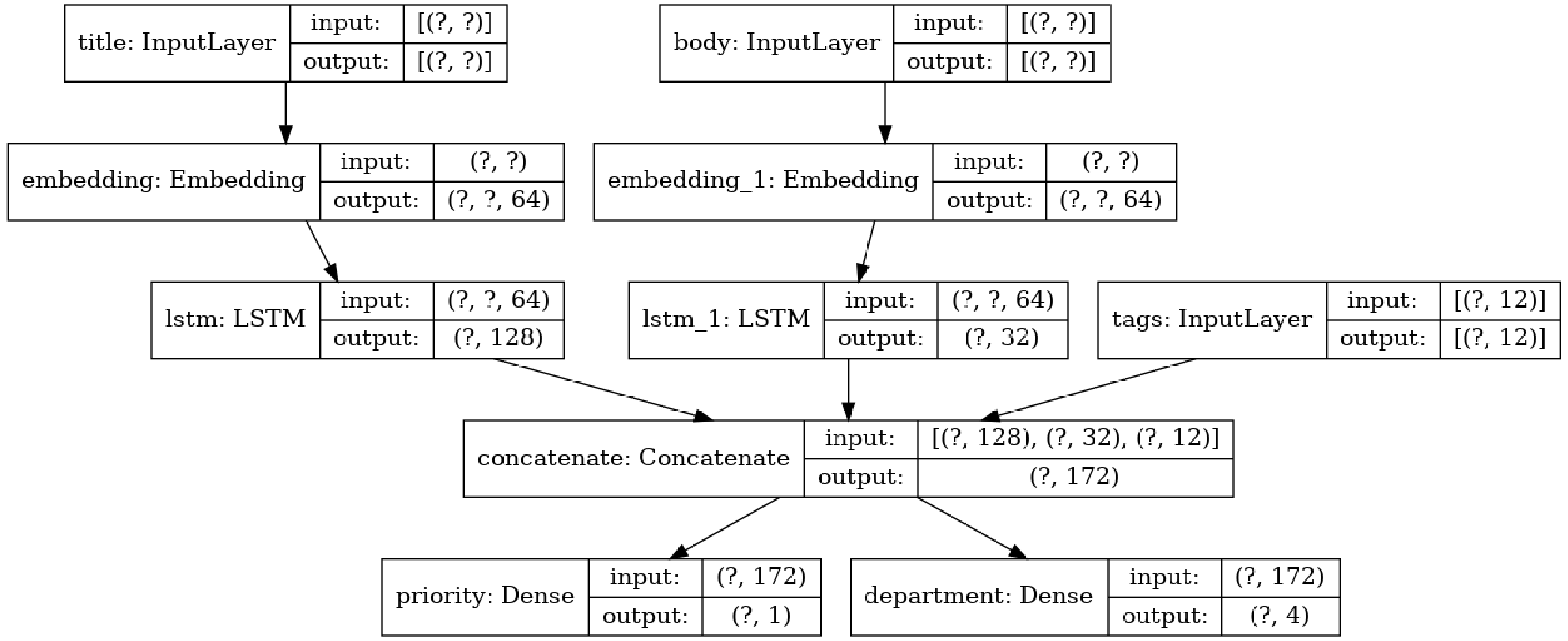
```
num_tags = 12  # Number of unique issue tags
num_words = 10000  # Size of vocabulary obtained when preprocessing text data
num_departments = 4  # Number of departments for predictions

title_input = keras.Input(shape=(None,), name='title')  # Variable-length sequence of ints
body_input = keras.Input(shape=(None,), name='body')  # Variable-length sequence of ints
tags_input = keras.Input(shape=(num_tags,), name='tags')  # Binary vectors of size `num_tags`

# Embed each word in the title into a 64-dimensional vector
title_features = layers.Embedding(num_words, 64)(title_input)
# Embed each word in the text into a 64-dimensional vector
body_features = layers.Embedding(num_words, 64)(body_input)
```

Multiple Inputs and Outputs

```
# Reduce sequence of embedded words in the title into a single 128-dimensional vector
title_features = layers.LSTM(128)(title_features)
# Reduce sequence of embedded words in the body into a single 32-dimensional vector
body_features = layers.LSTM(32)(body_features)

# Merge all available features into a single large vector via concatenation
x = layers.concatenate([title_features, body_features, tags_input])

# Stick a logistic regression for priority prediction on top of the features
priority_pred = layers.Dense(1, activation='sigmoid', name='priority')(x)
# Stick a department classifier on top of the features
department_pred = layers.Dense(num_departments, activation='softmax', name='department')(x)

# Instantiate an end-to-end model predicting both priority and department
model = keras.Model(inputs=[title_input, body_input, tags_input],
                    outputs=[priority_pred, department_pred])
```

Multiple Inputs and Outputs

```
keras.utils.plot_model(model, 'multi_input_and_output_model.png', show_shapes=True)
```



title: InputLayer | input: [(?, ?)] | output: [(?, ?)]
body: InputLayer | input: [(?, ?)] | output: [(?, ?)]
embedding: Embedding | input: (?, ?) | output: (?, ?, 64)
embedding_1: Embedding | input: (?, ?) | output: (?, ?, 64)
lstm: LSTM | input: (?, ?, 64) | output: (?, 128)
lstm_1: LSTM | input: (?, ?, 64) | output: (?, 32)
tags: InputLayer | input: [(?, 12)] | output: [(?, 12)]
concatenate: Concatenate | input: [(?, 128), (?, 32), (?, 12)] | output: (?, 172)
priority: Dense | input: (?, 172) | output: (?, 1)
department: Dense | input: (?, 172) | output: (?, 4)

Multiple Inputs and Outputs

- When compiling this model, we can assign different losses to each output.
- You can even assign different weights to each loss, to modulate their contribution to the total training loss.

```
model.compile(optimizer=keras.optimizers.RMSprop(1e-3),
              loss=['binary_crossentropy', 'categorical_crossentropy'],
              loss_weights=[1., 0.2])
```

Multiple Inputs and Outputs

- Since we gave names to our output layers, we could also specify the loss like this:

```
model.compile(optimizer=keras.optimizers.RMSprop(1e-3),
              loss={'priority': 'binary_crossentropy',
                    'department': 'categorical_crossentropy'},
              loss_weights=[1., 0.2])
```

Multiple Inputs and Outputs

- We can train the model by passing lists of Numpy arrays of inputs and targets:

```
import numpy as np

# Dummy input data
title_data = np.random.randint(num_words, size=(1280, 10))
body_data = np.random.randint(num_words, size=(1280, 100))
tags_data = np.random.randint(2, size=(1280, num_tags)).astype('float32')
# Dummy target data
priority_targets = np.random.random(size=(1280, 1))
dept_targets = np.random.randint(2, size=(1280, num_departments))

model.fit({'title': title_data, 'body': body_data, 'tags': tags_data},
          {'priority': priority_targets, 'department': dept_targets},
          epochs=2,
          batch_size=32)
```

Multiple Inputs and Outputs

- When calling fit with a Dataset object, it should yield either a tuple of lists like :

  ([title_data, body_data, tags_data], [priority_targets, dept_targets])

- or a tuple of dictionaries like :

  ({'title': title_data, 'body': body_data, 'tags': tags_data}, {'priority': priority_targets, 'department': dept_targets}).

Exercise

- Train the ticket model on the randomly generated data.

Exercise

- Write a convolutional model that takes six 19x19 planes as input and that outputs a vector of 362 with a softmax (the policy) and an output of 1 (the value).

- Train it on randomly generated data with different losses for the policy (categorical cross entropy) and the value (mse).

Residual Networks

- A toy resnet model
- In addition to models with multiple inputs and outputs, the Functional API makes it easy to manipulate non-linear connectivity topologies : models where layers are not connected sequentially.
- This also cannot be handled with the Sequential API (as the name indicates).
- A common use case for this is residual connections.

Residual Networks

AlphaGo

David Silver

Aja Huang

AlphaGo

Fan Hui is the european Go champion and a 2p professional Go player :

AlphaGo Fan won 5-0 against Fan Hui in November 2015.

Nature, January 2016.

AlphaGo

Lee Sedol is among the strongest and most famous 9p Go player :

AlphaGo Lee won 4-1 against Lee Sedol in march 2016.

AlphaGo

Ke Jie is the world champion of Go according to Elo ratings :

AlphaGo Master won 3-0 against Ke Jie in may 2017.

Golois

- In 2016 I replicated the AlphaGo experiments with the policy and value networks.
- Golois policy network scores 58.54% on the test set (57.0% for AlphaGo).
- Policy alone (instant play) is ranked 4d on kgs instead of 3d for AlphaGo.
- The improvement is the use of a residual network for the policy.

Residual Networks

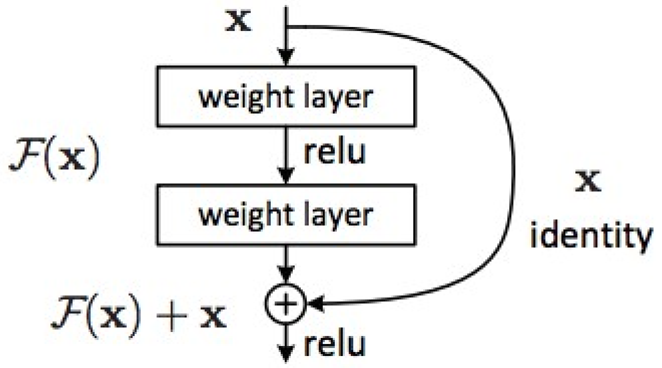
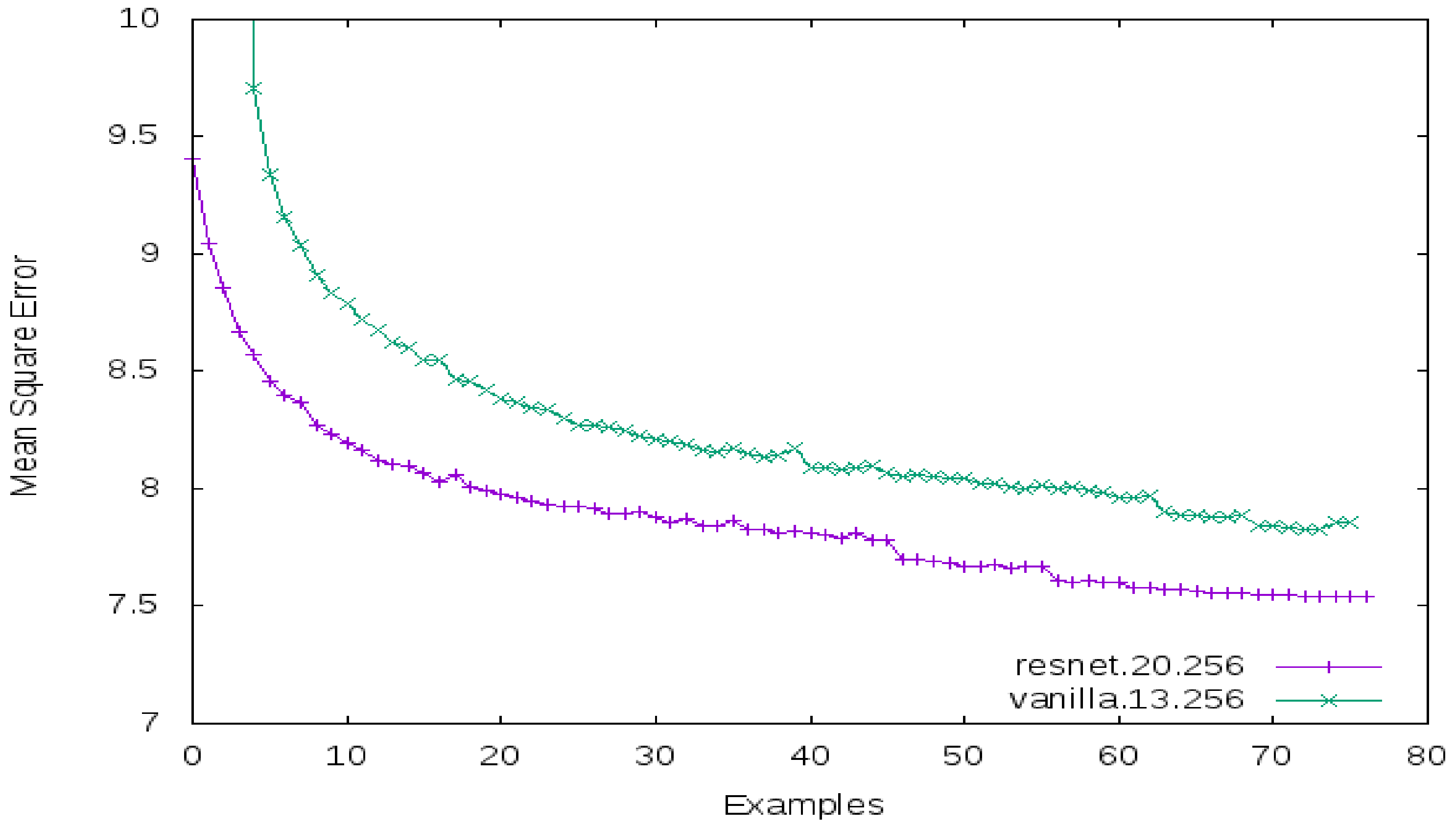
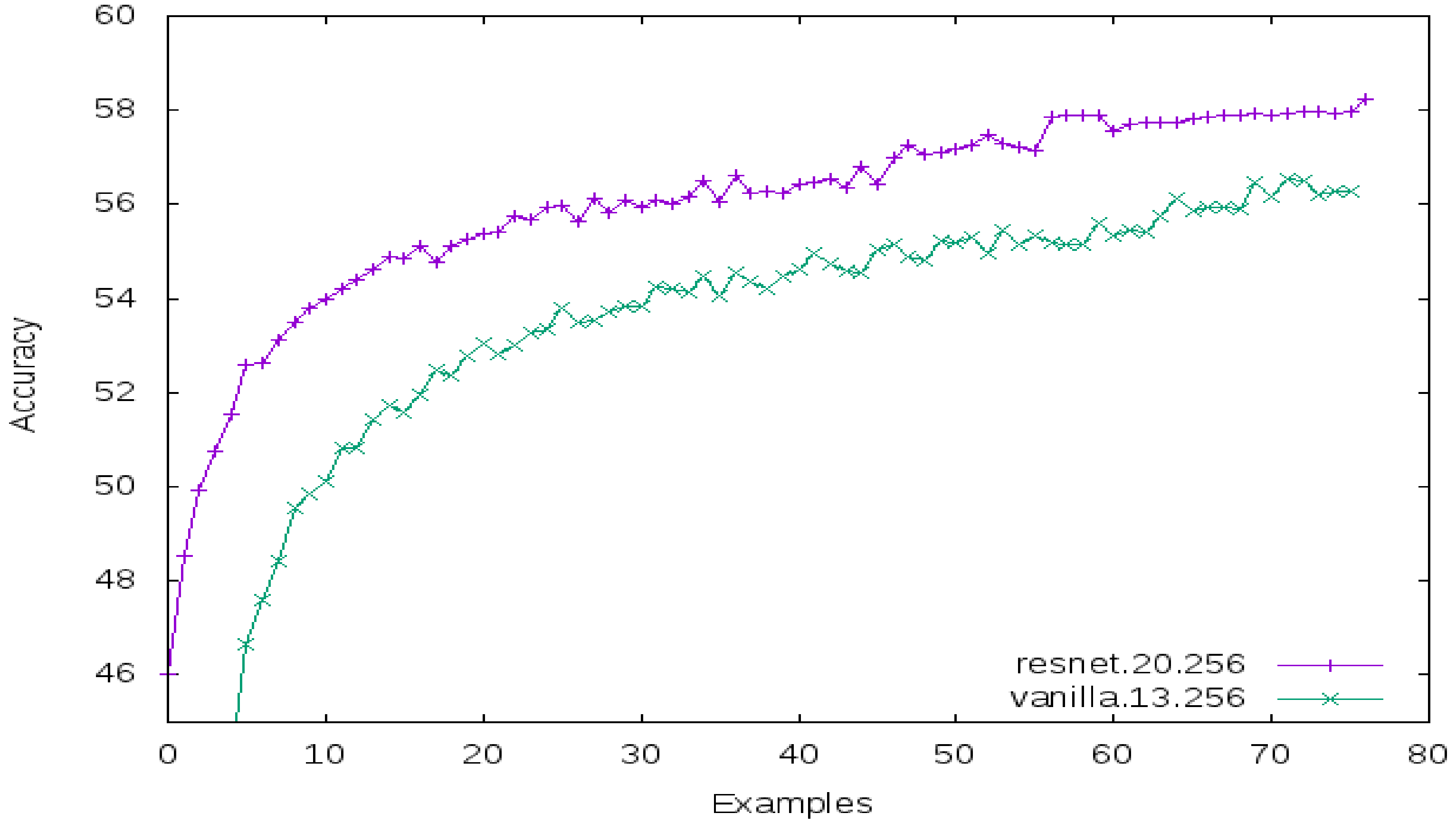


Figure 2. Residual learning: a building block.

Evolution of the error



Evolution of the accuracy



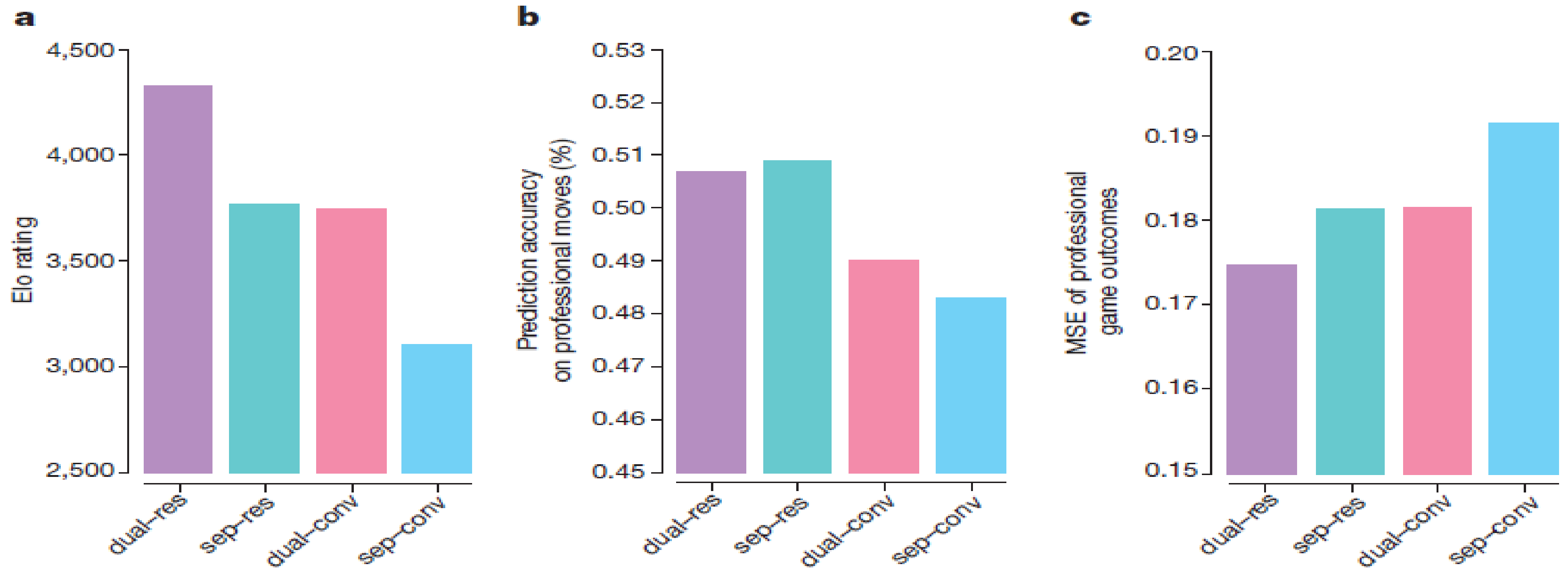
AlphaGo

AlphaGo Zero learns to play Go from scratch playing against itself.

After 40 days of self play it surpasses AlphaGo Master.

Nature, 18 october 2017.

AlphaGo Zero



Residual Networks

Defining a residual model with layers :

```
from tensorflow.keras import layers
input = tensorflow.keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(32, 1, activation='relu', padding=’same’)(input)
ident = x
x = layers.Conv2D(32, (3, 3), activation='relu', padding=’same’)(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding=’same’)(x)
x = layers.add([ident,x])
model = keras.models.Model(inputs=input, outputs=x)
```

Residual Networks

Train a standard convolutional network on the CIFAR10 image dataset.

Compare it to a residual network.

Shared Layers

- Another good use for the functional API are models that use shared layers. Shared layers are layer instances that get reused multiple times in a same model: they learn features that correspond to multiple paths in the graph-of-layers.

- Shared layers are often used to encode inputs that come from similar spaces (say, two different pieces of text that feature similar vocabulary), since they enable sharing of information across these different inputs, and they make it possible to train such a model on less data.

- If a given word is seen in one of the inputs, that will benefit the processing of all inputs that go through the shared layer.

Shared Layers

- To share a layer in the Functional API, just call the same layer instance multiple times.
- For instance, here's an Embedding layer shared across two different text inputs:

```
# Embedding for 1000 unique words mapped to 128-dimensional vectors
shared_embedding = layers.Embedding(1000, 128)

# Variable-length sequence of integers
text_input_a = keras.Input(shape=(None,), dtype='int32')

# Variable-length sequence of integers
text_input_b = keras.Input(shape=(None,), dtype='int32')

# We reuse the same layer to encode both inputs
encoded_input_a = shared_embedding(text_input_a)
encoded_input_b = shared_embedding(text_input_b)
```

Transfer Learning

- Because the graph of layers you are manipulating in the Functional API is a static datastructure, it can be accessed and inspected.
- This means that we can access the activations of intermediate layers ("nodes" in the graph) and reuse them elsewhere.
- This is extremely useful for feature extraction.
- This is a VGG19 model with weights pre-trained on ImageNet:

```
from tensorflow.keras.applications import VGG19
vgg19 = VGG19()
```

Transfer Learning

- And these are the intermediate activations of the model, obtained by querying the graph datastructure:

```
features_list = [layer.output for layer in vgg19.layers]
```

- We can use these features to create a new feature-extraction model, that returns the values of the intermediate layer activations -- and we can do all of this in 3 lines.

```
feat_extraction_model = keras.Model(inputs=vgg19.input, outputs=features_list)
img = np.random.random((1, 224, 224, 3)).astype('float32')
extracted_features = feat_extraction_model(img)
```

Exercise

- Reuse the first layers of VGG19 to train a model with a different head on CIFAR10.

```
from tensorflow.keras.applications.vgg16 import VGG16

model = VGG16(
    classes=10,
    input_shape=(32,32,3)
)
```

Defining New Layers

- It's easy to extend the API by creating your own layers.
- All layers subclass the Layer class and implement:
  - A call method, that specifies the computation done by the layer.
  - A build method, that creates the weights of the layer
  - (note that this is just a style convention; you could create weights in __init__ as well).

Defining New Layers

Here's a simple implementation of a Dense layer:

```
class CustomDense(layers.Layer):

  def __init__(self, units=32):
    super(CustomDense, self).__init__()
    self.units = units

  def build(self, input_shape):
    self.w = self.add_weight(shape=(input_shape[-1], self.units),
                             initializer='random_normal',
                             trainable=True)
    self.b = self.add_weight(shape=(self.units,),
                             initializer='random_normal',
                             trainable=True)

  def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b

inputs = keras.Input((4,))
outputs = CustomDense(10)(inputs)

model = keras.Model(inputs, outputs)
```

Exercise

- Write a class for a residual block and test it on CIFAR10.

Project

- www.lamsade.dauphine.fr/~cazenave
- Train a network to play Go
- Submit trained networks
- Tournament of networks