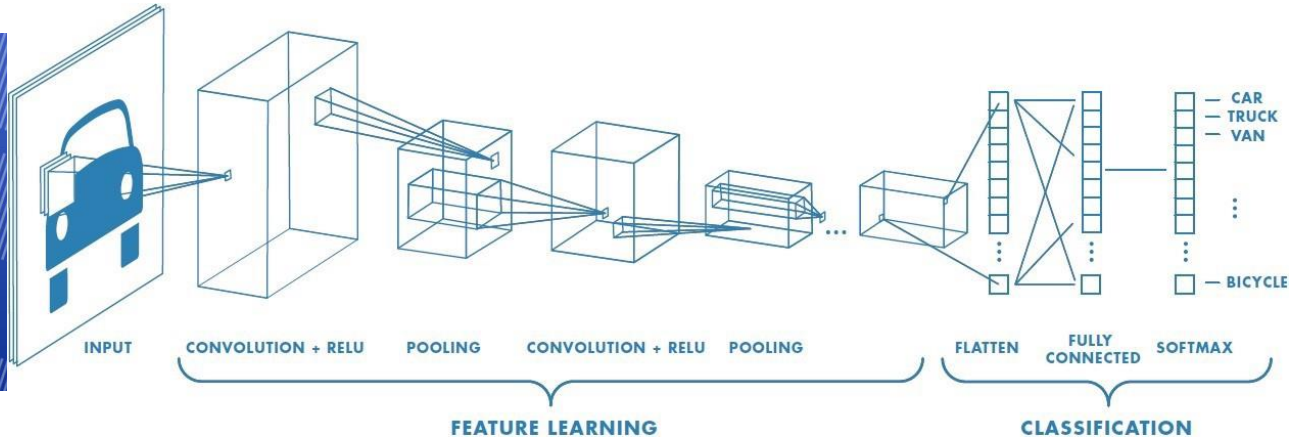
Machine Learning - Day 2 -

Georges Sakr - ESIB

Day 2: Deep Learning with ConvNets



Georges Sakr
ESIB

Outline

- Deep convolutional neural networks
- Image classification
- CIFAR Dataset

So Far

- We discussed dense nets, in which each layer is fully connected to the adjacent layers.
- We applied those dense networks to classify the MNIST handwritten characters dataset.
- However, this strategy does not leverage the spatial structure and relations of each image. In particular, this piece of code transforms the bitmap representing each written digit into a flat vector,

- `#X_train is 60000 rows of 28x28 values --> reshaped in 60000 x 784`
- `X_train = X_train.reshape(60000, 784)`
- `X_test = X_test.reshape(10000, 784)`

Convolution Neural Networks - Humans

- Convolutional neural networks (also called ConvNet) leverage spatial information and are therefore very well suited for classifying images.

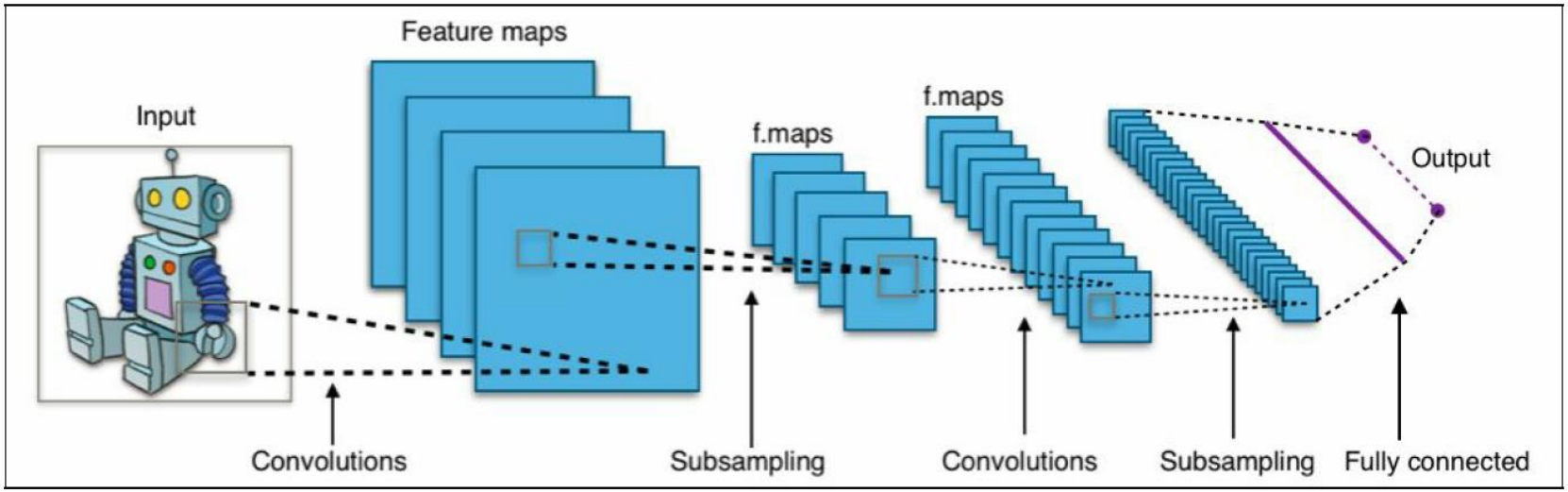
- Our vision is based on multiple cortex levels, each one recognizing more and more structured information.

- First, we see single pixels; then from them, we recognize simple geometric forms. And then... more and more sophisticated elements such as objects, faces, human bodies, animals, and so on.

CNN – Description

- A **deep convolutional neural network** (**DCNN**) consists of many neural network layers.
- Two different types of layers, convolutional and pooling, are typically alternated. The depth of each filter increases from left to right in the network.
- The last stage is typically made of one or more fully connected layers

Figure


Feature maps
Input
f.maps
f.maps
Output
Convolutions
Subsampling
Convolutions
Subsampling
Fully connected

Local Receptive Fields

- If we want to preserve spatial information, then it is convenient to represent each image with a matrix of pixels.
- To encode the local structure, connect a submatrix of adjacent input neurons (pixel) into one single hidden neuron belonging to the next layer.
- That single hidden neuron represents one local receptive field. Note that this operation is named convolution and it gives the name to this type of network.

LRF - Stride

- We can encode more information by having overlapping submatrices.
- Suppose that the size of each single submatrix is 5 x 5 and that those submatrices are used with MNIST images of 28 x 28 pixels. Then we will be able to generate 23 x 23 local receptive field neurons in the next hidden layer.
- In fact it is possible to slide the submatrices by only 23 positions before touching the borders of the images.
- In Keras, the size of each single submatrix is called **stride length**, and this is a hyperparameter that can be fine-tuned during the construction of our nets.

Feature Map

- Let's define the feature map from one layer to another layer. Of course, we can have multiple feature maps that learn independently from each hidden layer.

- For instance, we can start with 28 x 28 input neurons for processing MINST images and then recall *k* feature maps of size 23 x 23 neurons each (again with a stride of 5 x 5) in the next hidden layer.

Shared Weights and Biases

- Let's suppose that we want to move away from the pixel representation in a row by gaining the ability to detect the same feature independently from the location where it is placed in the input image.

- A simple intuition is to use the same set of weights and bias for all the neurons in the hidden layers. In this way, each layer will learn a set of position-independent features derived from the image.

- Assuming that the input image has shape *(256, 256)* on three channels with *tf* (TensorFlow) ordering, this is represented as *(256, 256, 3)*

Keras

- In Keras, if we want to add a convolutional layer with dimensionality of the output 32 and extension of each filter 3 x 3, we will write:

- model = Sequential()
- model.add(Conv2D(32, (3, 3), input_shape=(256, 256, 3))

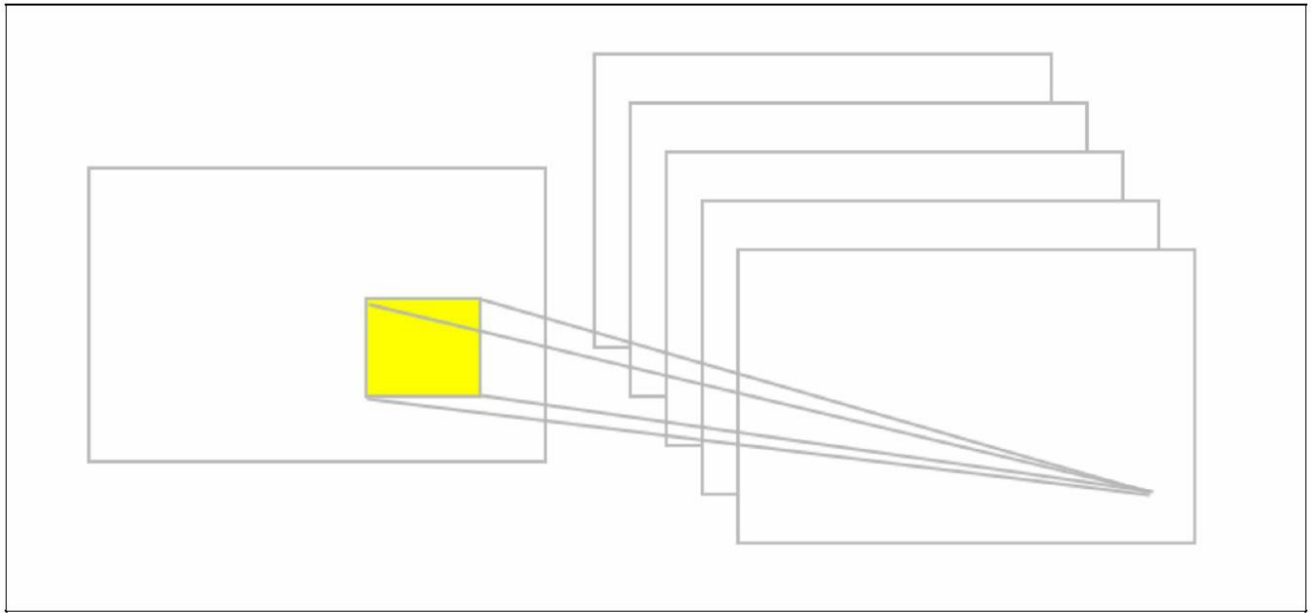
Or

- model = Sequential()
- model.add(Conv2D(32, kernel_size=3, input_shape=(256, 256, 3))

What that means?

- This means that we are applying a 3 x 3 convolution on a 256 x 256 image with three input channels (or input filters), resulting in 32 output channels (or output filters).

- An example of convolution is provided in the following diagram:



Pooling Layer

- Let's suppose that we want to summarize the output of a feature map. Again, we can use the spatial contiguity of the output produced from a single feature map and aggregate the values of a submatrix into a single output value that synthetically describes the *meaning* associated with that physical region.

Max Pooling

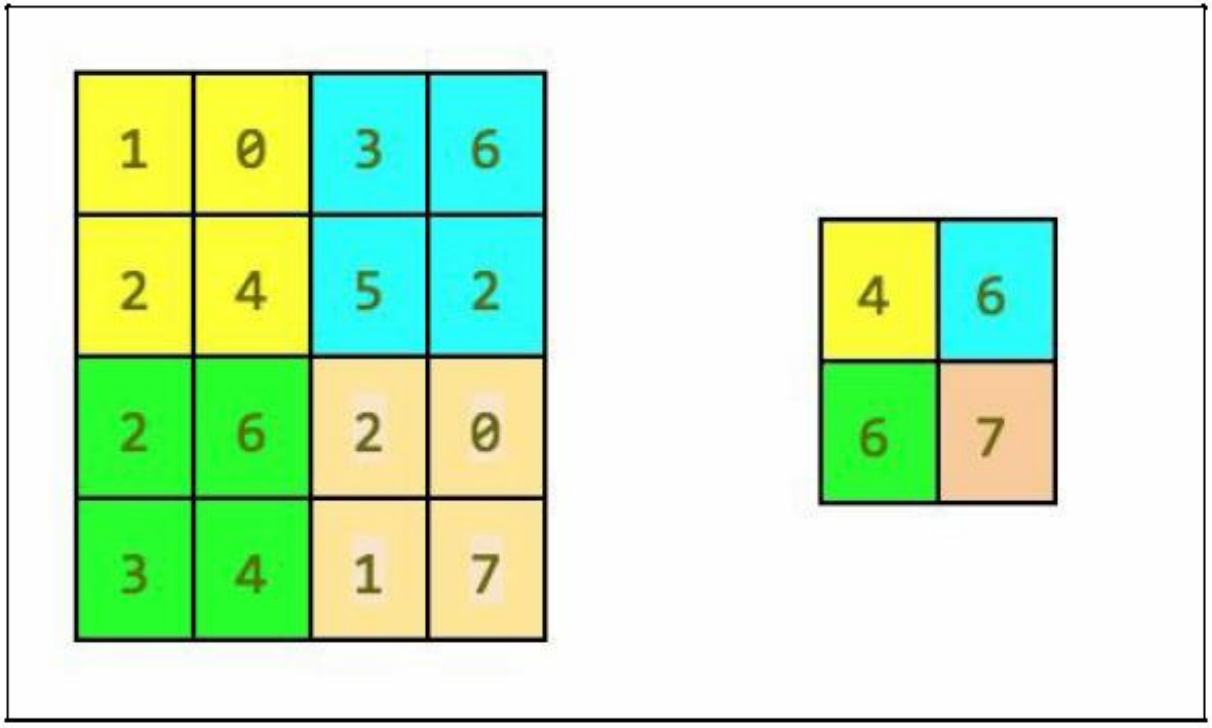
- One easy and common choice is *max-pooling*, which simply outputs the maximum activation as observed in the region.

- In Keras, if we want to define a max-pooling layer of size 2 x 2, we will write:

- `>> model.add(MaxPooling2D(pool_size = (2, 2)))`

Max-Pooling example

- An example of max-pooling is shown in the following diagram:



Average pooling

- Another choice is average pooling, which simply aggregates a region into the average values of the activations observed in that region.

- Note that Keras implements a large number of pooling layers and a complete list is available at:
- https://keras.io/layers/pooling/

- In short, all pooling operations are nothing more than a summary operation on a given region.

ConvNets summary

- So far, we have described the basic concepts of ConvNets.
- CNNs apply convolution and pooling operations in one dimension for audio and text data along the time dimension
- In two dimensions for images along the (height x width) dimensions
- Three dimensions for videos along the (height x width x time) dimensions.

ConvNet summary 2

- For images, sliding the filter over input volume produces a map that gives the responses of the filter for each spatial position.

- ConvNet has multiple filters stacked together which learn to recognize specific visual features independently of the location in the image. Those visual features are simple in the initial layers of the network, and then more and more sophisticated deeper in the network.

Application LeNet

- Yann le Cun proposed a family of ConvNets named LeNet trained for recognizing MNIST handwritten characters with robustness to simple geometric transformations and to distortion.

- The key intuition here is to have low-layers alternating convolution operations with max-pooling operations.

- The convolution operations are based on carefully chosen local receptive fields with shared weights for multiple feature maps. Then, higher levels are fully connected layers based on a traditional MLP with hidden layers and softmax as the output layer.

LeNet code in Keras

- `keras.layers.convolutional.Conv2D(filters, kernel_size, padding='valid')`

- filters is the number of convolution kernels to use (for example, the dimensionality of the output),

- kernel_size is an integer or tuple/list of two integers, specifying the width and height of the 2D convolution window (can be a single integer to specify the same value for all spatial dimensions),

LeNet Code

- padding='same' means that padding is used. There are two options: padding='valid' means that the convolution is only computed where the input and the filter fully overlap, and therefore the output is smaller than the input, while padding='same' means that we have an output that is the *same* size as the input, for which the area around the input is padded with zeros.

LeNet Code

- In addition, we use a MaxPooling2D module:

```
keras.layers.pooling.MaxPooling2D(pool_size=(2, 2),
strides=(2, 2))
```

- Pool_size=(2, 2) is a tuple of two integers representing the factors by which the image is vertically and horizontally downscaled. So *(2, 2)* will halve the image in each dimension,
- strides=(2, 2) is the stride used for processing.

Stage 1

- We have a first convolutional stage with ReLU activations followed by a max-pooling.
- Our net will learn 20 convolutional filters, each one of which has a size of 5 x 5.
- The output dimension is the same one of the input shape, so it will be 28 x 28.
- Note that since the Convolution2D is the first stage of our pipeline, we are also required to define its input_shape.
- The max-pooling operation implements a sliding window that slides over the layer and takes the maximum of each region with a step of two pixels vertically and horizontally:

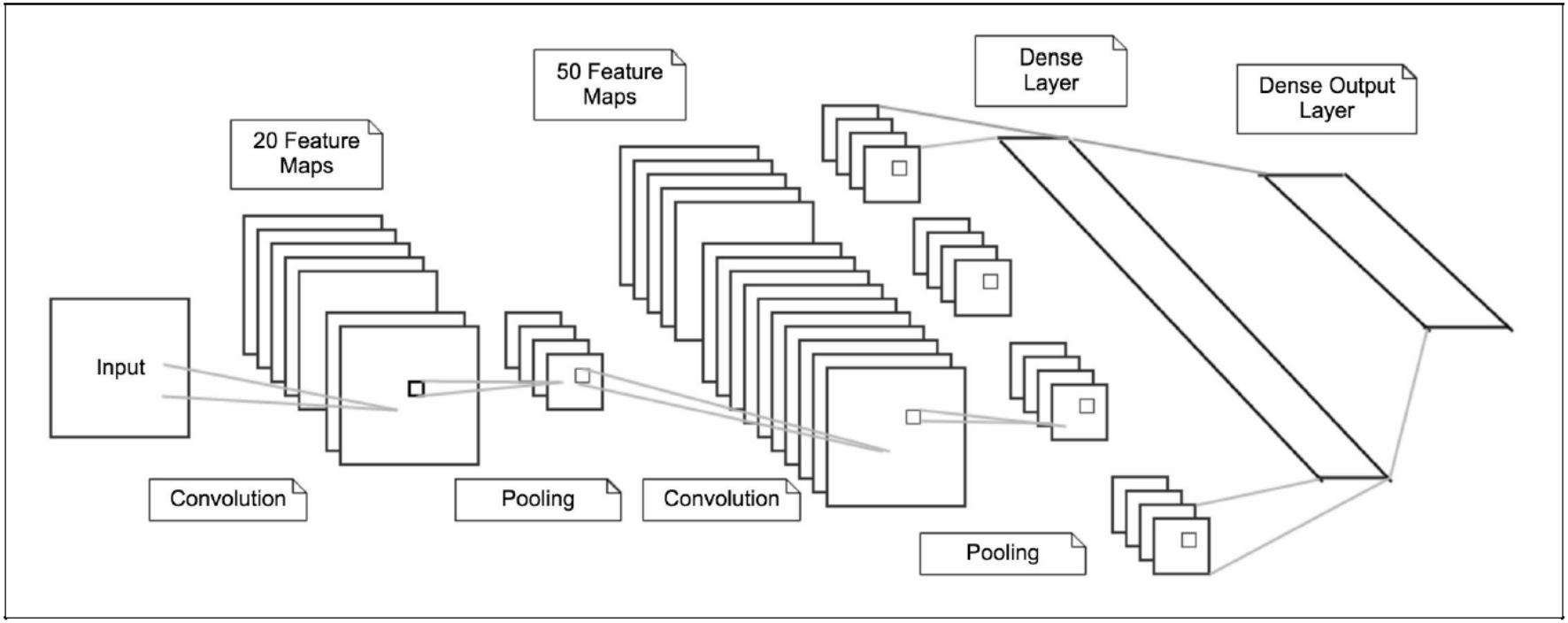
Stage 2

- Then a second convolutional stage with ReLU activations followed, again by a max-pooling.
- In this case, we increase the number of convolutional filters learned to 50 from the previous 20.
- Increasing the number of filters in deeper layers is a common technique used in deep learning:

Stage 3

- Then we have a pretty standard flattening and a dense network of 500 neurons, followed by a softmax classifier with 10 classes:

Congratulations



Training the network

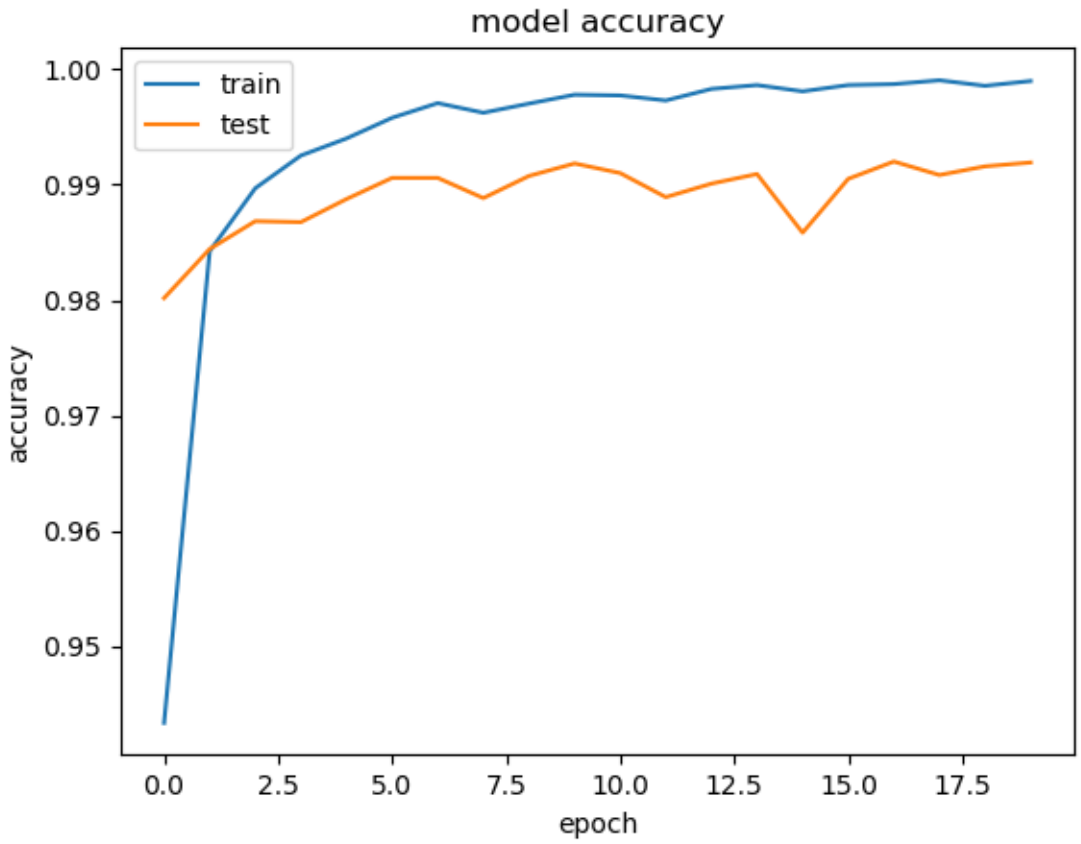
- Now we need some additional code for training the network, but this is very similar to what we have already described in Day 1.

Results

Anaconda Prompt

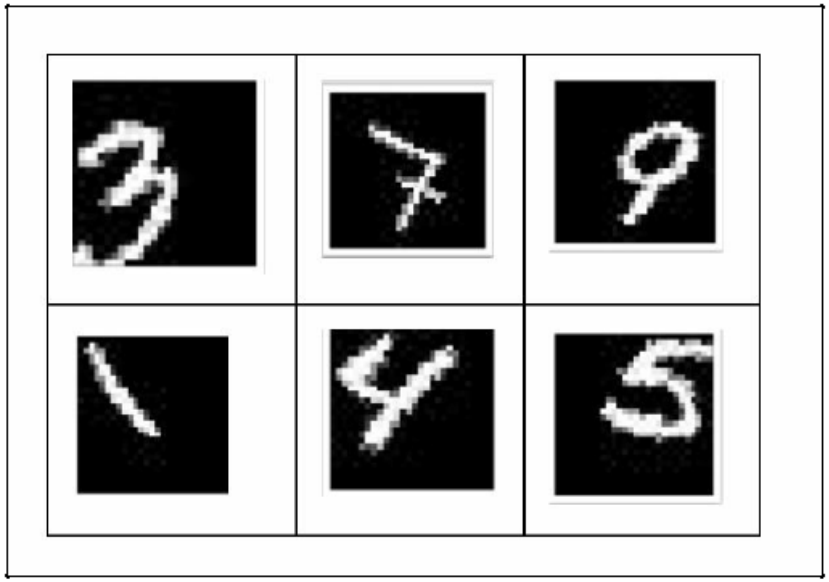
```
48000/48000 [==============================] - 2s 45us/step - loss: 0.0060 - acc: 0.9979 - val_loss: 0.0455 - val_acc: 0
.9902
Epoch 13/20
48000/48000 [==============================] - 2s 45us/step - loss: 0.0058 - acc: 0.9981 - val_loss: 0.0681 - val_acc: 0
.9853
Epoch 14/20
48000/48000 [==============================] - 2s 46us/step - loss: 0.0060 - acc: 0.9979 - val_loss: 0.0488 - val_acc: 0
.9887
Epoch 15/20
48000/48000 [==============================] - 2s 46us/step - loss: 0.0045 - acc: 0.9985 - val_loss: 0.0369 - val_acc: 0
.9910
Epoch 16/20
48000/48000 [==============================] - 2s 45us/step - loss: 0.0024 - acc: 0.9993 - val_loss: 0.0424 - val_acc: 0
.9923
Epoch 17/20
48000/48000 [==============================] - 2s 45us/step - loss: 1.4300e-04 - acc: 1.0000 - val_loss: 0.0394 - val_ac
c: 0.9932
Epoch 18/20
48000/48000 [==============================] - 2s 45us/step - loss: 5.7362e-05 - acc: 1.0000 - val_loss: 0.0392 - val_ac
c: 0.9932
Epoch 19/20
48000/48000 [==============================] - 2s 46us/step - loss: 0.0029 - acc: 0.9991 - val_loss: 0.0582 - val_acc: 0
.9878
Epoch 20/20
48000/48000 [==============================] - 2s 46us/step - loss: 0.0105 - acc: 0.9969 - val_loss: 0.0402 - val_acc: 0
.9914
10000/10000 [==============================] - 0s 43us/step
Test score: 0.029490891997788003
Test accuracy: 0.9921
```

Accuracy


model accuracy
train
test
accuracy
epoch
1.00
0.99
0.98
0.97
0.96
0.95
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

Results Interpretation

- Let's see some of the MNIST images just to understand how good the number 99.2% is! For instance, there are many ways in which humans write a 9, one of them appearing in the following diagram. The same holds for 3, 7, 4, and 5. The number **1** in this diagram is so difficult to recognize that probably even a human will have issues with it:

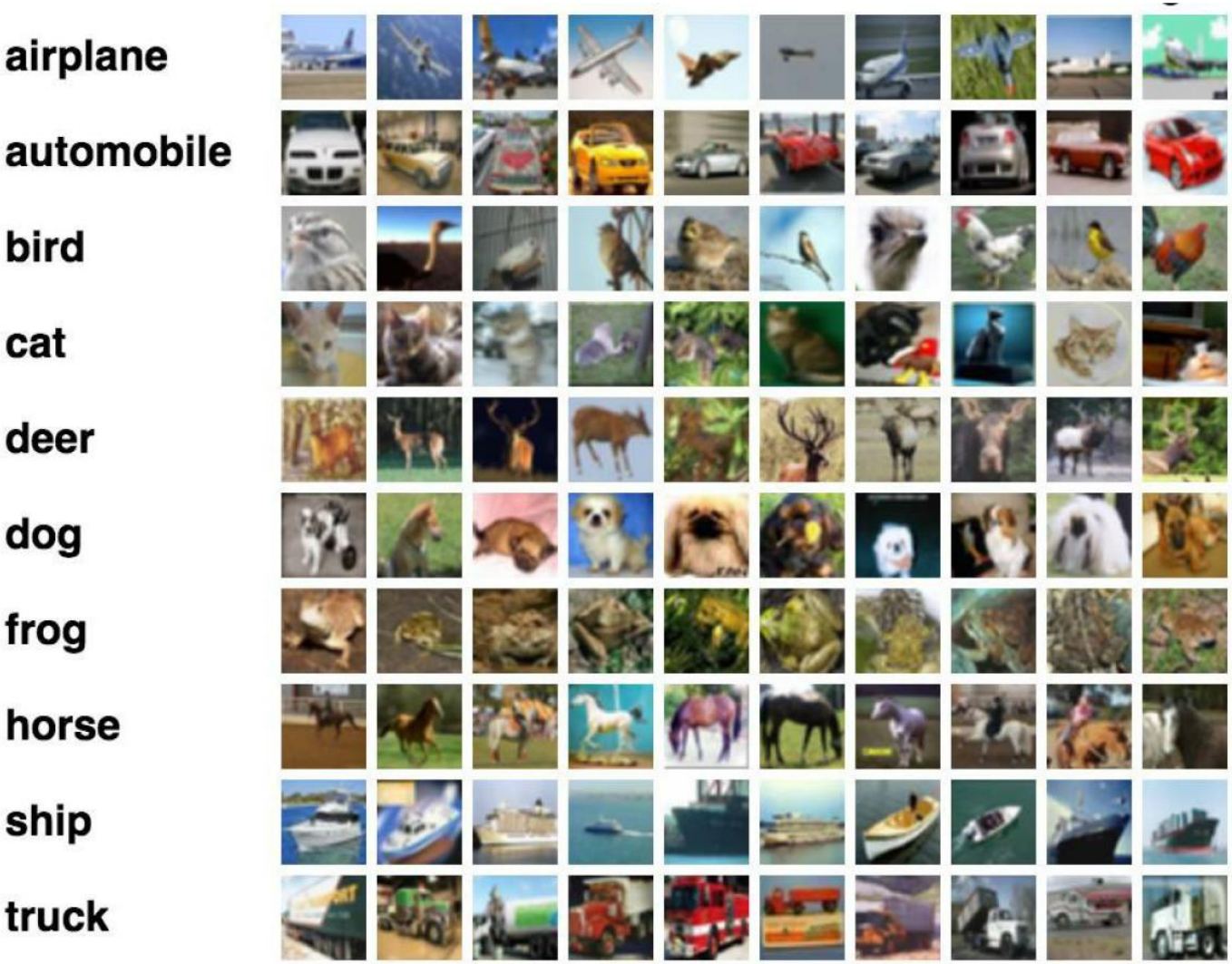


Recognizing CIFAR-10 images with deep learning

CIFAR-10

- The CIFAR-10 dataset contains 60,000 color images of 32 x 32 pixels in 3 channels divided into 10 classes.
- Each class contains 6,000 images. The training set contains 50,000 images, while the test sets provides 10,000 images.
- This image taken from the CIFAR repository describes a few random examples from the 10 classes:

Examples


airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Code

- The goal is to recognize previously unseen images and assign them to one of the 10 classes.
- Let us define a suitable deep net.
- First of all we import a number of useful modules, define a few constants, and load the dataset:

Network Design

- Our net will learn 32 convolutional filters, each of which with a 3 x 3 size. The output dimension is the same one of the input shape, so it will be 32 x 32 and activation is ReLU.

- After that we have a max-pooling operation with pool size 2 x 2 and a dropout at 25%

- The next stage in the deep pipeline is a dense network with 512 units and ReLU activation followed by a dropout at 50% and by a softmax layer with 10 classes as output, one for each category

Train

- After defining the network, we can train the model. In this case, we split the data and compute a validation set in addition to the training and testing sets.

- The training is used to build our models, the validation is used to select the best performing approach, while the test set is to check the performance of our best models on fresh unseen data

Improving accuracy with a deeper network

- One way to improve the performance is to define a deeper network with multiple convolutional operations. In this example, we have a sequence of modules:

- *conv+conv+maxpool+dropout+conv+conv+maxpool*

  32 3x3 | 0.25 | 64 3x3

- *dense+dropout+dense*

  512 | 0.5

Improving the CIFAR-10 performance with data augmentation

- Another way to improve the performance is to generate more images for our training. The key intuition is that we can take the standard CIFAR training set and augment this set with multiple types of transformations including:
- rotation, rescaling, horizontal/vertical flip, zooming, channel shift, and many more. Let us see the code:

Augment and Train

- Now we can apply this intuition directly for training. Using the same ConvNet defined previously we simply generate more augmented images and then we train.

- For efficiency, the generator runs in parallel to the model. This allows an image augmentation on the CPU and in parallel

- to training on the GPU. Here is the code:

Code

```
datagen.fit(X_train)
# train
history = model.fit_generator(datagen.flow(X_train, Y_train,
batch_size=BATCH_SIZE), samples_per_epoch=X_train.shape[0],
epochs=NB_EPOCH, verbose=VERBOSE)
score = model.evaluate(X_test, Y_test)
print("Test score:", score[0])
print('Test accuracy:', score[1])
```

Accuracy

- Each iteration is now more expensive because we have more training data. So let us run for 50 iterations only and see that we reach an accuracy of 78.3%