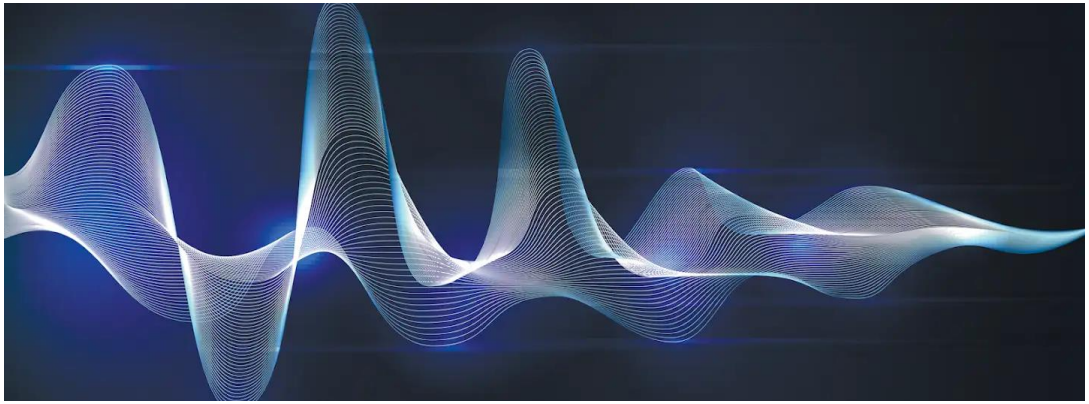
Machine Learning - Day 4 –

GAN

Georges Sakr - ESIB

Day 4: Generative Adversarial Networks and WaveNet



Georges Sakr
ESIB

Outline

- What is GAN?
- Deep convolutional GAN
- Applications of GAN

GANs

- Today we will discuss **generative adversarial networks** (**GANs**) and WaveNets.

- GANs have been defined as *the most interesting idea in the last 10 years in ML* by Yann LeCun, one of the fathers of deep learning.

- GANs are able to learn how to reproduce synthetic data that looks real.

GANs

- Computers can learn how to paint and create realistic images.
- The idea was originally proposed by Ian Goodfellow in 2016
- WaveNet is a deep generative network proposed by Google DeepMind to teach computers how to reproduce human voices and musical instruments, both with impressive quality.

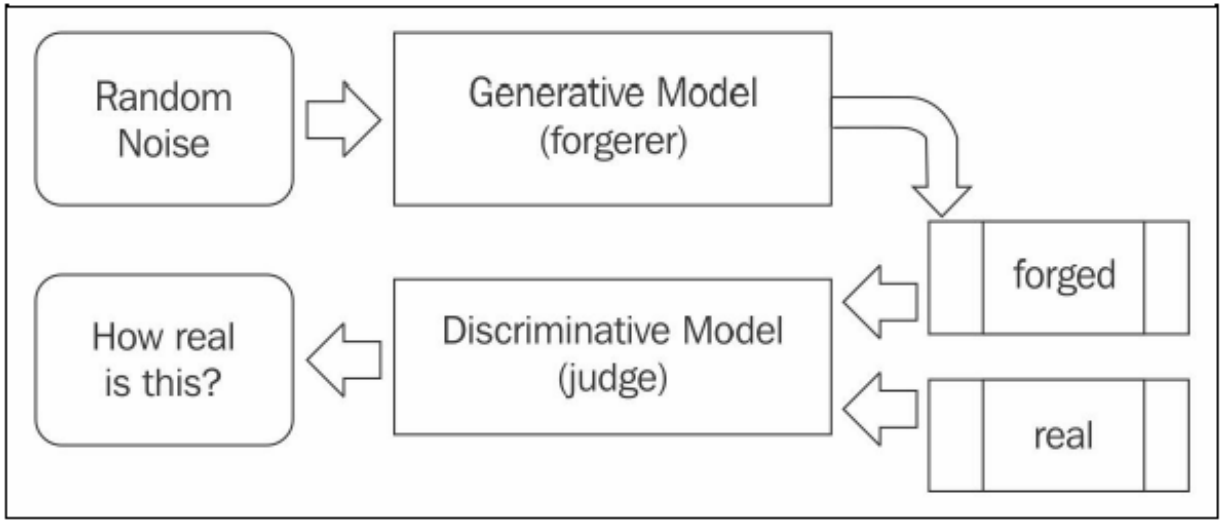
What is GAN?

- The key intuition of GAN can be easily considered as analogous to *art forgery*, which is the process of creating works of art that are falsely credited to other, usually more famous, artists.

- GANs train two neural nets simultaneously, as shown in the next diagram.

What is GAN?

- The generator *G(Z)* makes the forgery, and the discriminator *D(Y)* can judge how realistic the reproductions based on its observations of authentic pieces of arts and copies are.



What is GAN?

- *D(Y)* takes an input, *Y,* (for instance, an image) and expresses a vote to judge how real the input. (0-Real, 1-Forged)

- *G(Z)* takes an input from a random noise, *Z*, and trains itself to fool *D* into thinking that whatever *G(Z)* produces is real.

- So, *G* and *D* play an opposite game; hence the name *adversarial training*. Note that we train *G* and *D* in an alternating manner, where each of their objectives is expressed as a loss function optimized via a gradient descent.

What is GAN

- The generative model learns how to forge more successfully, and the discriminative model learns how to recognize forgery more successfully.

- The discriminator network (usually a standard convolutional neural network) tries to classify whether an input image is real or generated.

- The important new idea is to backpropagate through both the discriminator and the generator to adjust the generator's parameters in such a way that the generator can learn how to fool the the discriminator.

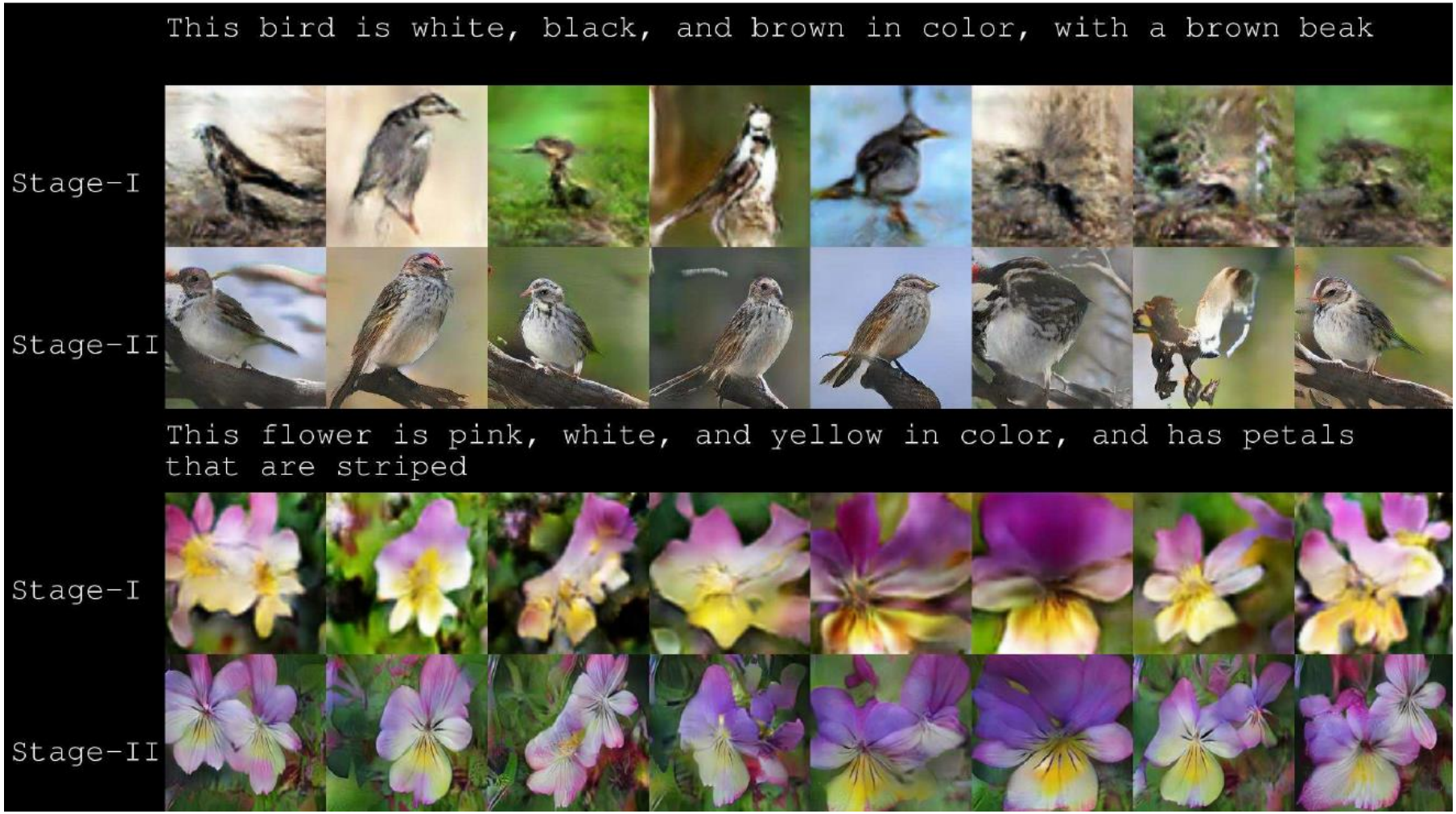
- At the end, the generator will learn how to produce forged images that are indistinguishable from real ones:

Some GAN Applications

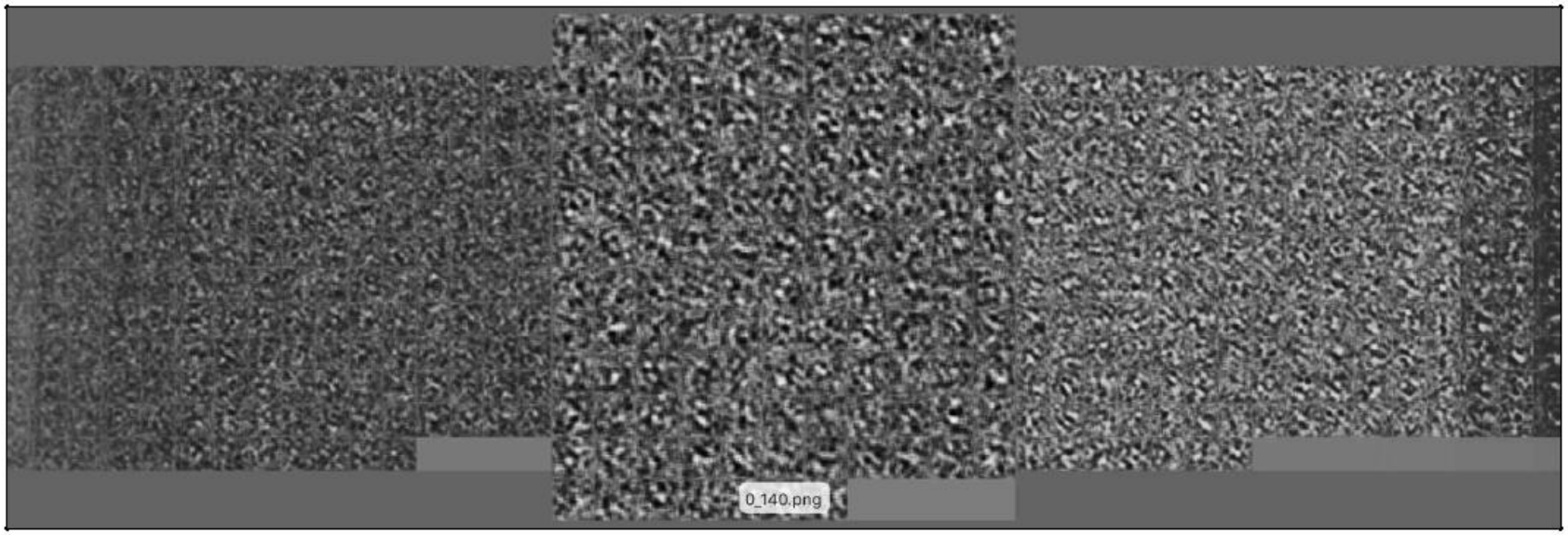
- We have seen that the generator learns how to forge data. This means that it learns how to create new synthetic data, which is created by the network, that looks real and like it was created by humans.

- *StackGAN: Text to Photo-Realistic Image Synthesis with Stacked Generative Adversarial Networks*, Here, a GAN has been used to synthesize forged images starting from a text description. The results are impressive. The first column is the real image in the test set, and the rest of the columns contain images generated from the same text description by Stage-I and Stage-II of StackGAN.

Result


This bird is white, black, and brown in color, with a brown beak
Stage-I
Stage-II
This flower is pink, white, and yellow in color, and has petals that are striped
Stage-I
Stage-II

Forging MNIST



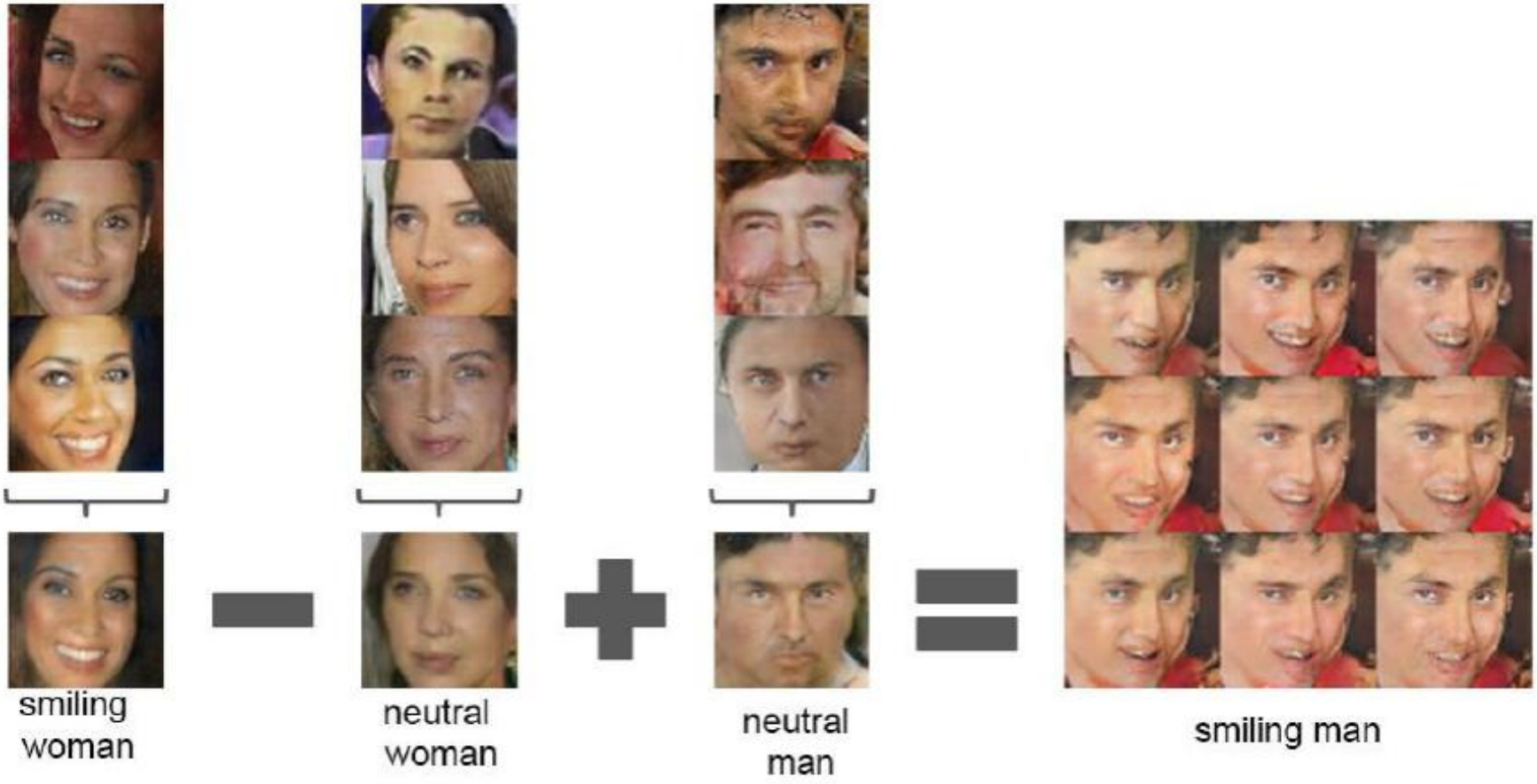
After few iterations

At the end

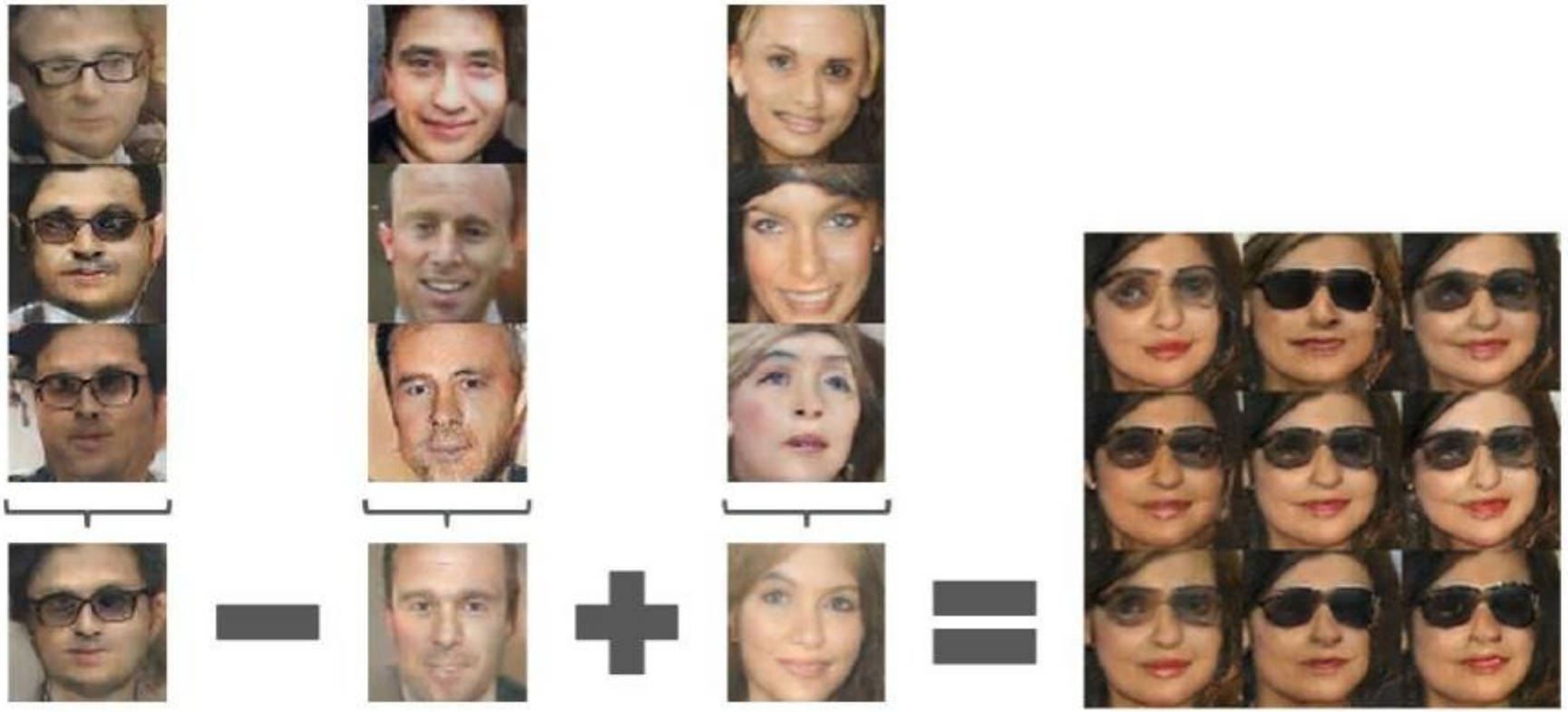
Arithmetic on Faces

- One of the coolest uses of GAN is arithmetic on faces in the generator's vector *Z*.
- In other words, if we stay in the space of synthetic forged images, it is possible to see things like this:
- *[smiling woman] - [neutral woman] + [neutral man] = [smiling man]*
- Or like this:
- *[man with glasses] - [man without glasses] + [woman without glasses] = [woman with glasses]*

Arithmetic on Faces



Arithmetic on faces



Deep convolutional generative adversarial networks

- The **deep convolutional generative adversarial networks** (**DCGAN**) are introduced in the paper:

  Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, by A. Radford, L. Metz, and S. Chintala in 2015

- The generator uses a 100-dimensional, uniform distribution space, *Z,* which is then projected into a smaller space by a series of vis-a-vis convolution operations. An example is shown in the following figure:

Network

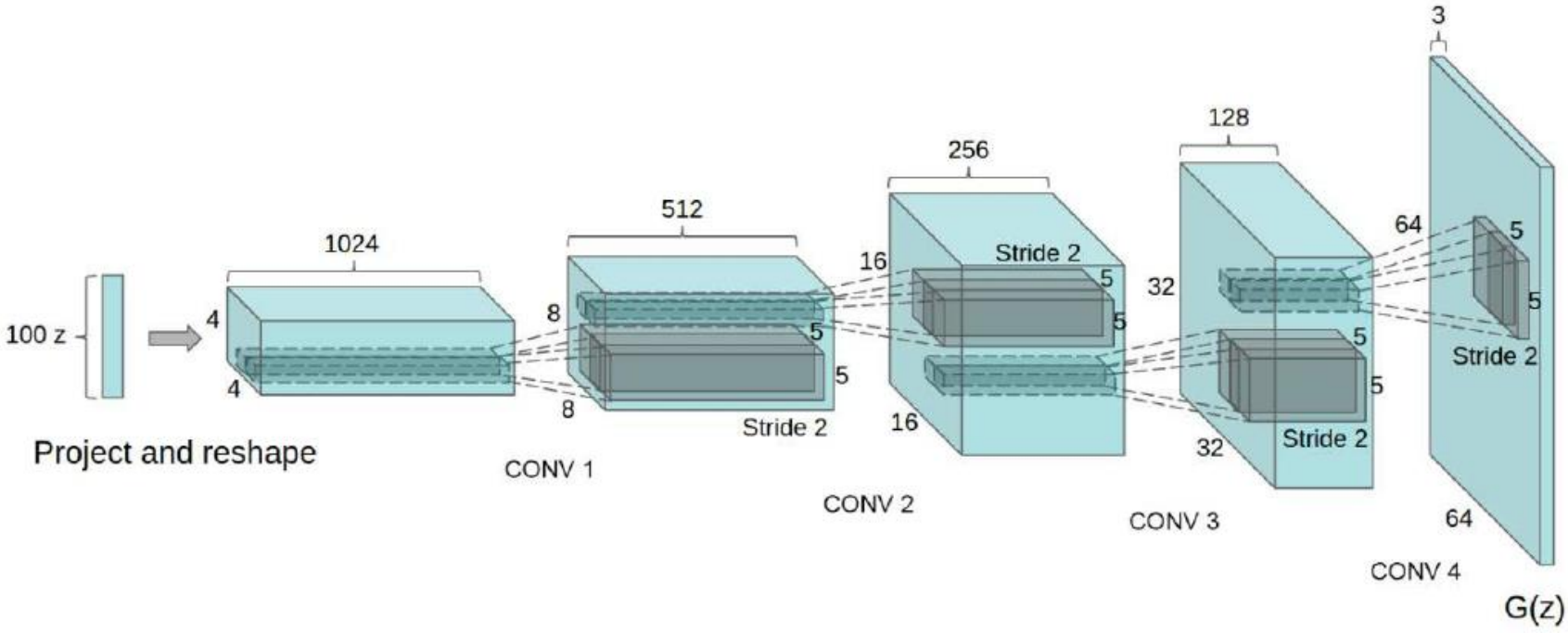


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

Generator Code

```
def generator_model():
    model = Sequential()
    model.add(Dense(input_dim=100, output_dim=1024))
    model.add(Activation('tanh'))
    model.add(Dense(128*7*7))
    model.add(BatchNormalization())
    model.add(Activation('tanh'))
    model.add(Reshape((128, 7, 7), input_shape=(128*7*7,)))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(64, (5, 5), border_mode='same'))
    model.add(Activation('tanh'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Conv2D(1, (5, 5), border_mode='same'))
    model.add(Activation('tanh'))
    return model
```

Discriminator

```
def discriminator_model():
    model = Sequential()
    model.add(Conv2D(64, 5, 5, border_mode='same',
input_shape=(1, 28, 28)))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, 5, 5))
    model.add(Activation('tanh'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(1024))
    model.add(Activation('tanh'))
    model.add(Dense(1))
    model.add(Activation('sigmoid'))
    return model
```

Code

- For a chosen number of epochs, the generator and discriminator are in turn trained by using binary_crossentropy as loss function. At each epoch, the generator makes a number of predictions (for example, it creates forged MNIST images) and the discriminator tries to learn after mixing the prediction with real MNIST images.

- After 32 epochs, the generator learns to forge this set of handwritten numbers.

- No one has programmed the machine to write but it has learned how to write numbers that are indistinguishable from the ones written by humans.

Results

Code

- `pip install keras_adversarial`

- If the generator G and the discriminator D are based on the same model, M, then they can be combined into an adversarial model; it uses the same input, M

```
adversarial_model = AdversarialModel(base_model=M,
player_params=[generator.trainable_weights,
discriminator.trainable_weights],
player_names=["generator", "discriminator"])
```

Code

- If the generator G and the discriminator D are based on the two different models, then it is possible to use this API call:

- ```
  adversarial_model = adversarialModel(
  player_models=[gan_g, gan_d],
  player_params=[generator.trainable_weights,
  discriminator.trainable_weights],
  player_names=["generator", "discriminator"])
  ```
  ```

MNIST

- Adversarial models train for multiplayer games. Given a base model with *n* targets and *k* players, create a model with *n***k* targets, where each player optimizes loss on that player's targets.

- In addition, simple_gan generates a GAN with the given gan_targets. Note that in the library, the labels for generator and discriminator are opposite; intuitively, this is a standard practice for GANs:

- …code

Decision Trees

- Decision trees are statistical data mining technique that express independent attributes and a dependent attributes logically AND in a tree shaped structure.
- Classification rules, extracted from decision trees, are IF-THEN expressions and all the tests have to succeed if each rule is to be generated.
- Decision tree usually separates the complex problem into many simple ones and resolves the sub problems
- Decision trees are predictive decision support tools that create mapping from observations to possible consequences.

Random Forest

- Random forest model is an ensemble of decision trees.
- Ensemble methods aggregates their predictions in determining the class label for a data point.
- Ensembles perform well when individual members are dissimilar, and random forests obtain variation among individual trees using two sources for randomness as follows:
  - Obtain a bootstrap sample of N cases.
  - At each node, randomly select a subset of attributes. Determine the best split at the node from this reduced set of b attributes
  - Grow the full tree without pruning
  - Random forests are computationally efficient since each tree is built independently of the others. With large number of trees in the ensemble, they are also noted to be robust to over fitting and noise in the data.

Artificial Neural Network Model

- **Fraud detection methods** based on neural network are the most popular ones.
- The advantages of neural networks over other techniques are that these models are able to learn from the past and thus, improve results as time passes.
- They can also extract rules and predict future activity based on the current situation.
- By employing neural networks, effectively, banks can detect fraudulent use of a card, faster and more efficiently.
- Among the reported credit card fraud studies most have focused on using neural networks.

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:

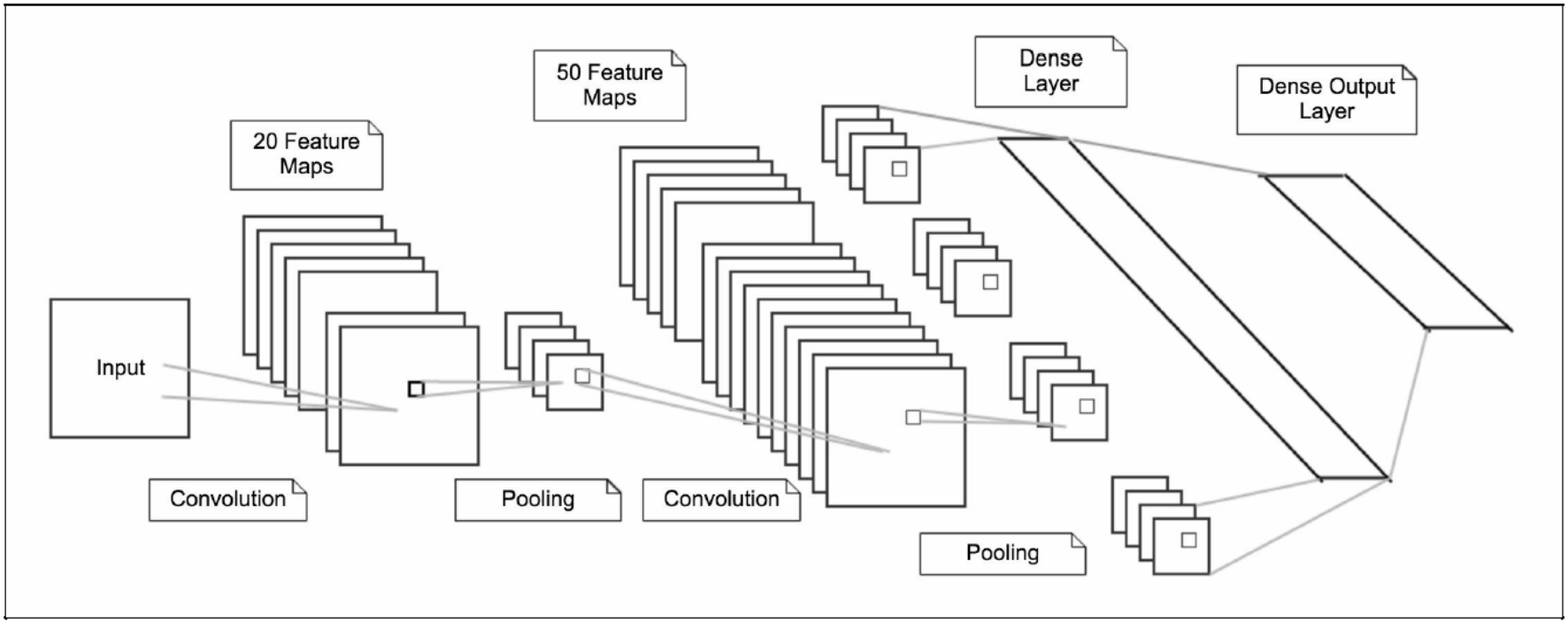
Stage 2

- 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

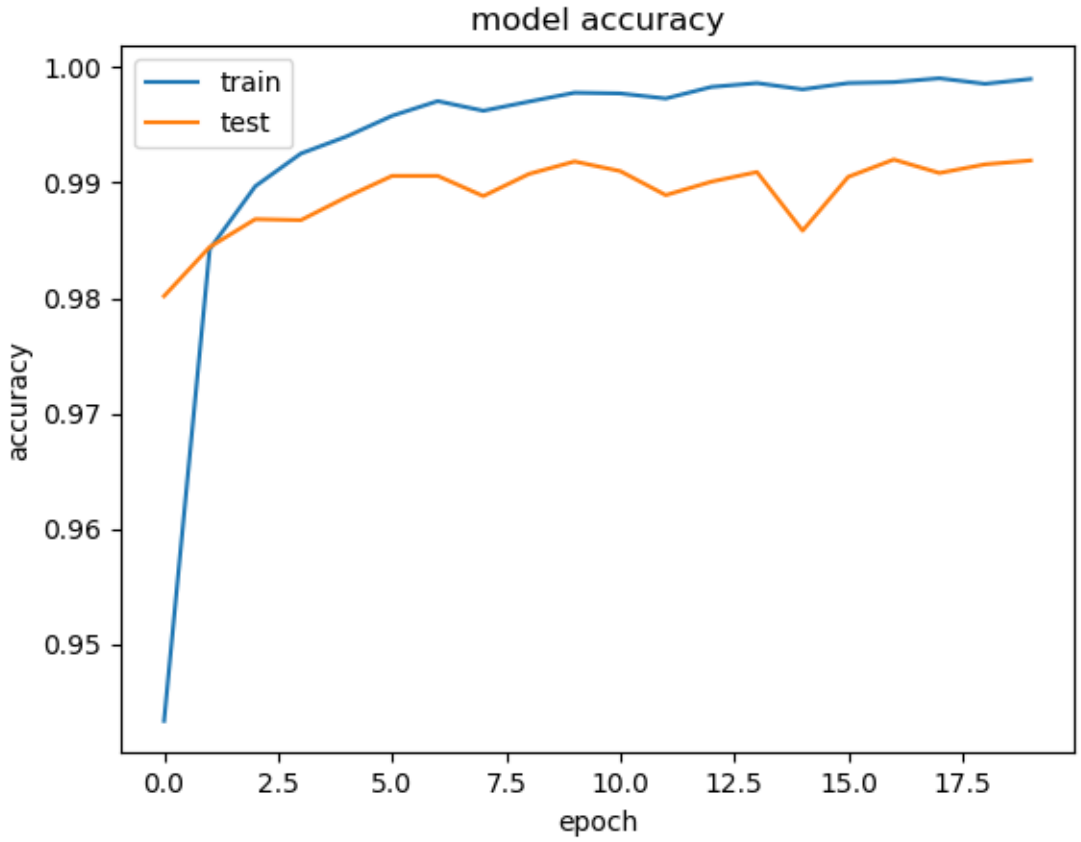
- Now we need some additional code for training the network, but this is very similar to what we have already described in Day 1.

Results

Anaconda Prompt

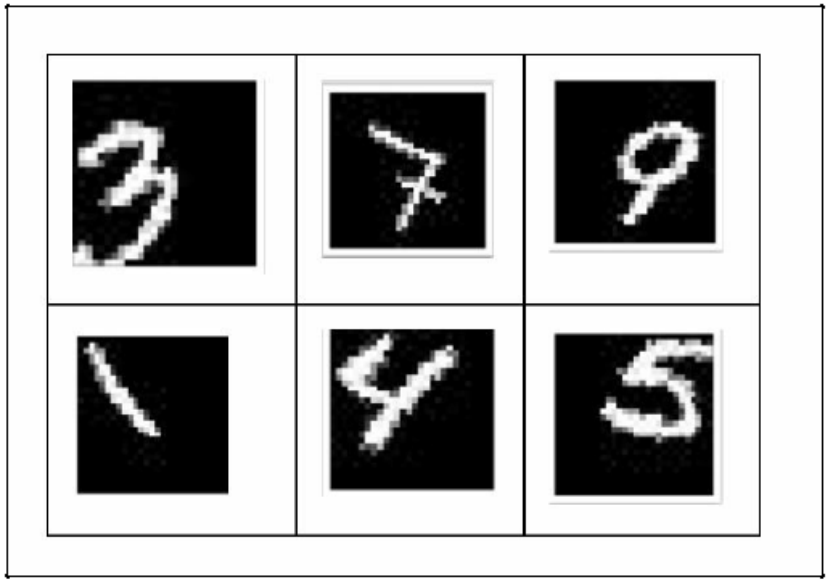
```
48000/48000 [==============================] - 2s 45us/step - loss: 0.0060 - acc: 0.9979 - val_loss: 0.0455 - val_acc: 0
.9902
Epoch 13/20
48000/48000 [==============================] - 2s 45us/step - loss: 0.0058 - acc: 0.9981 - val_loss: 0.0681 - val_acc: 0
.9853
Epoch 14/20
48000/48000 [==============================] - 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

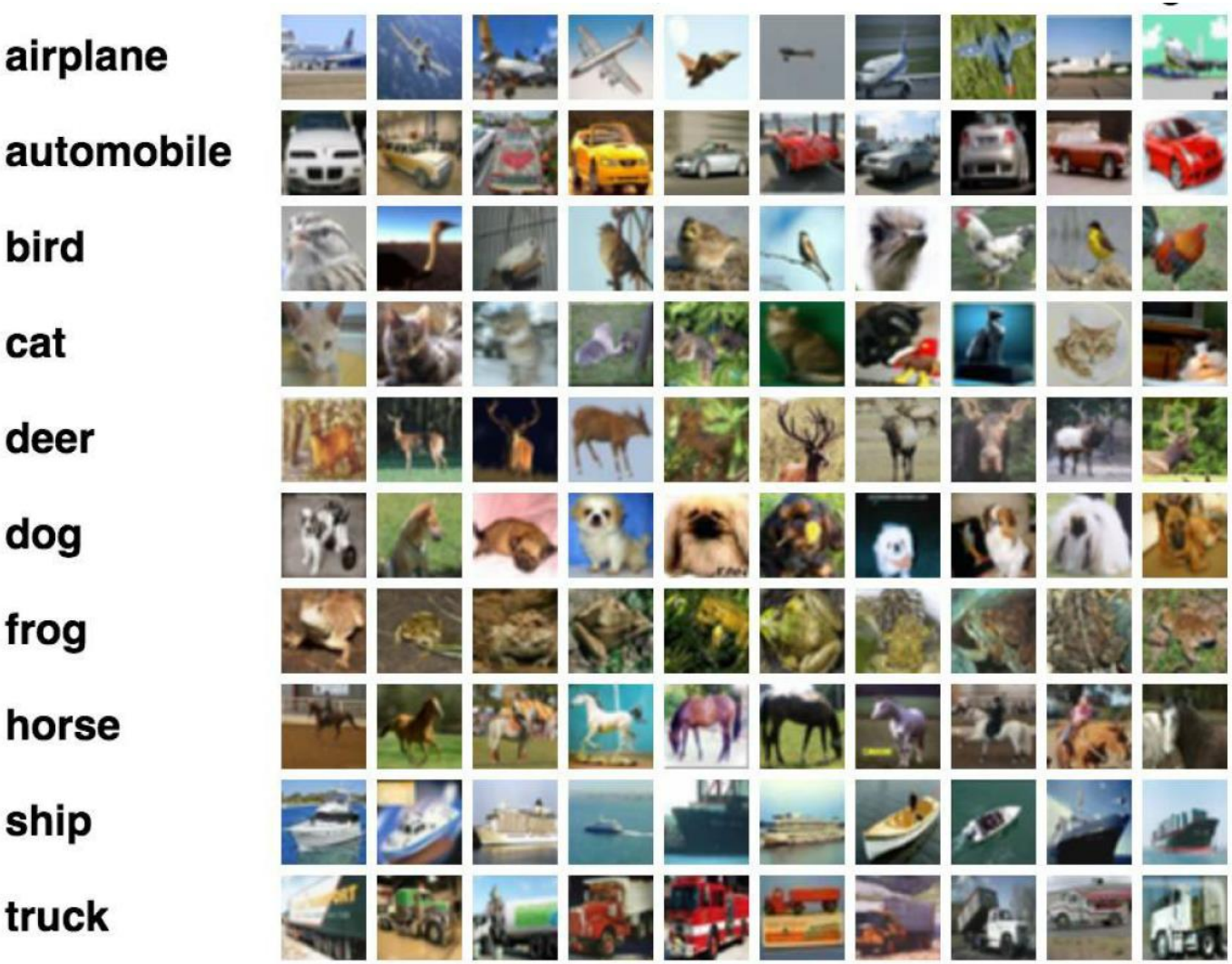
CIFAR-10

- 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%