





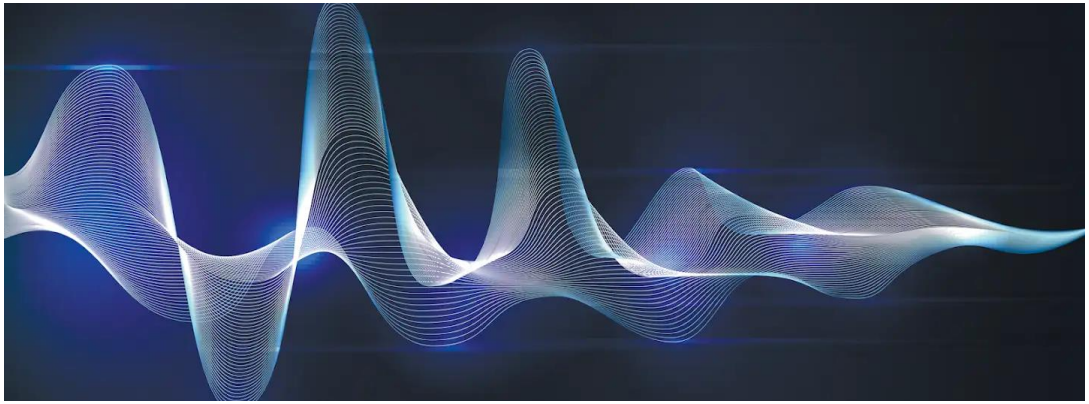
# Machine Learning

## - 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.

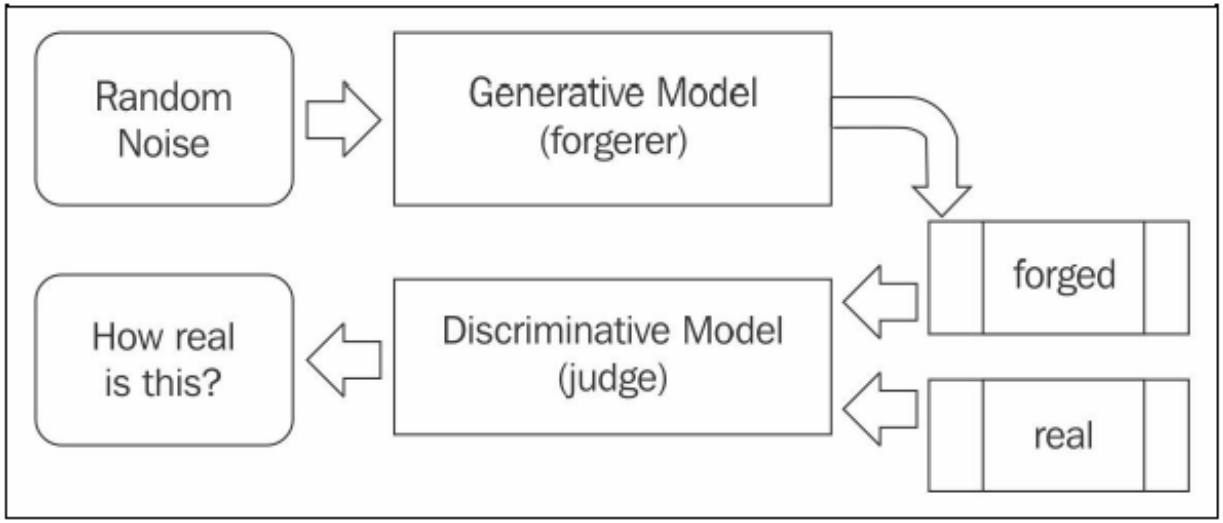
# What is GAN?

- 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.

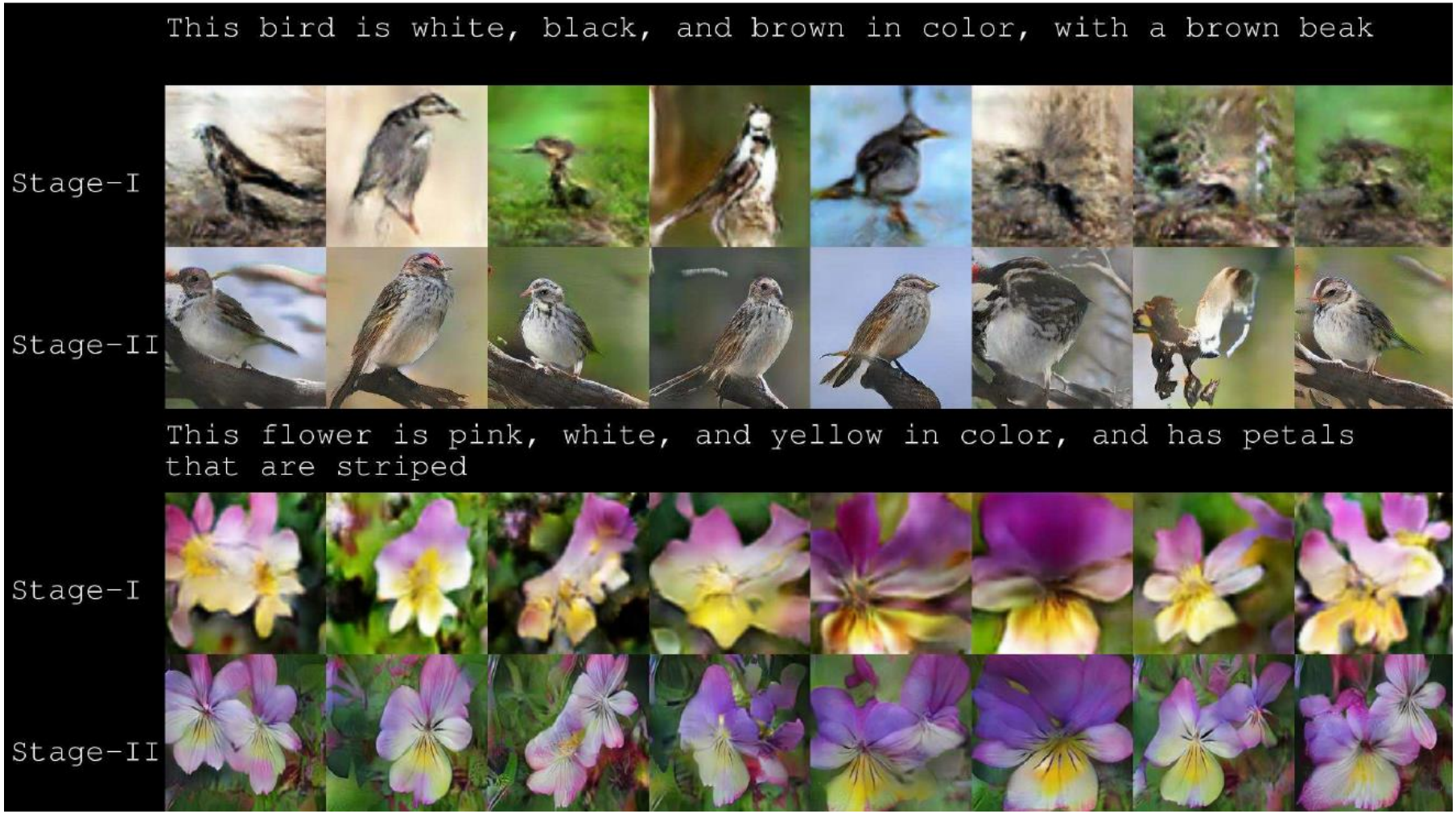
- At the end, the generator will learn how to produce forged images that are indistinguishable from real ones:

# Some GAN Applications

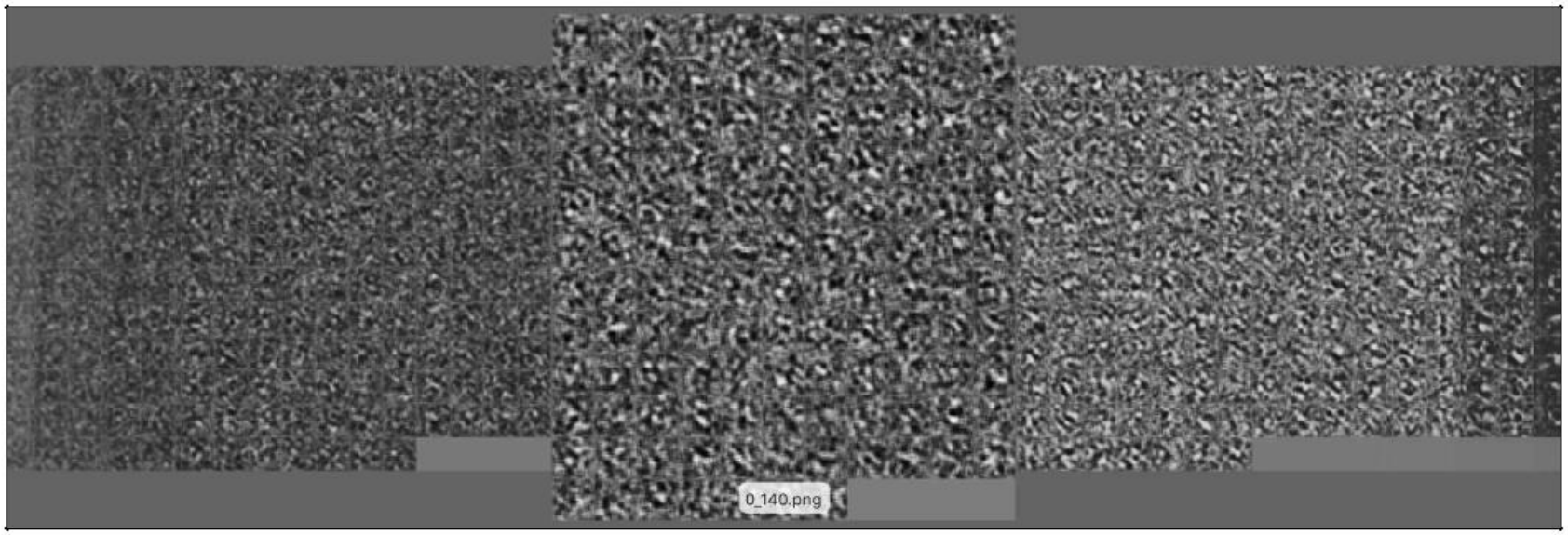
- We have seen that the generator learns how to forge data. This means that it learns how to create new synthetic data, which is created by the network, that looks real and like it was created by humans.

- *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



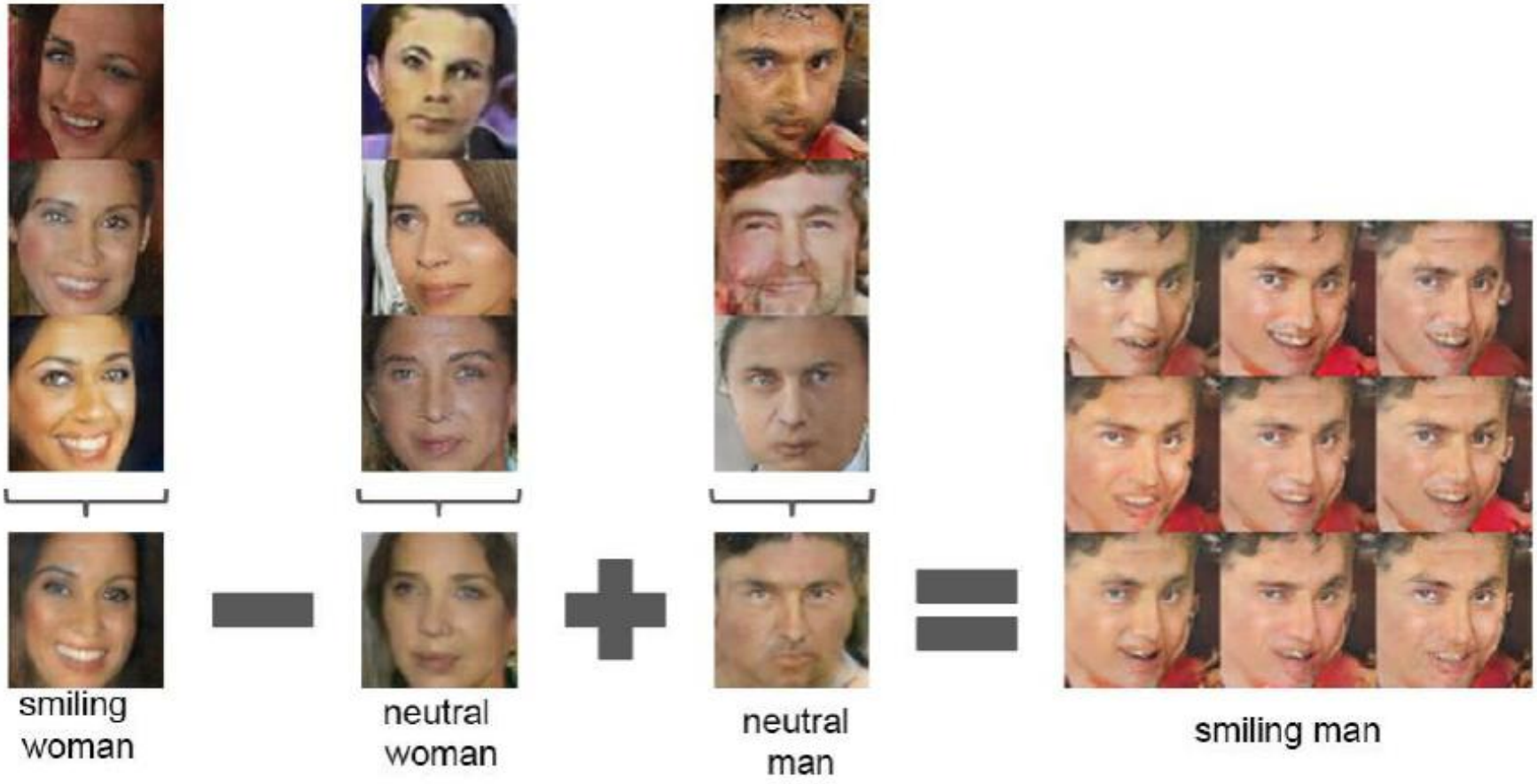
# After few iterations

# At the end

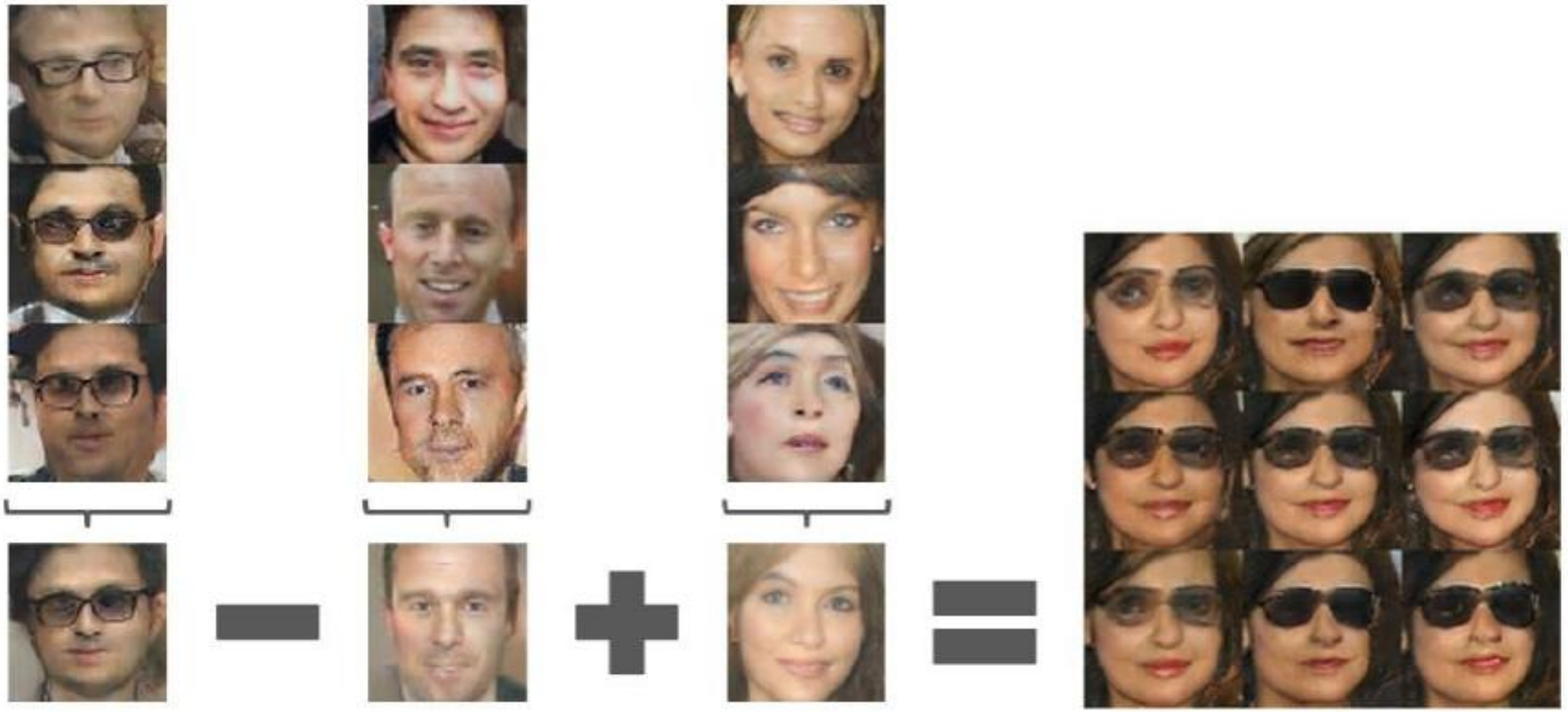
# Arithmetic on Faces

- 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

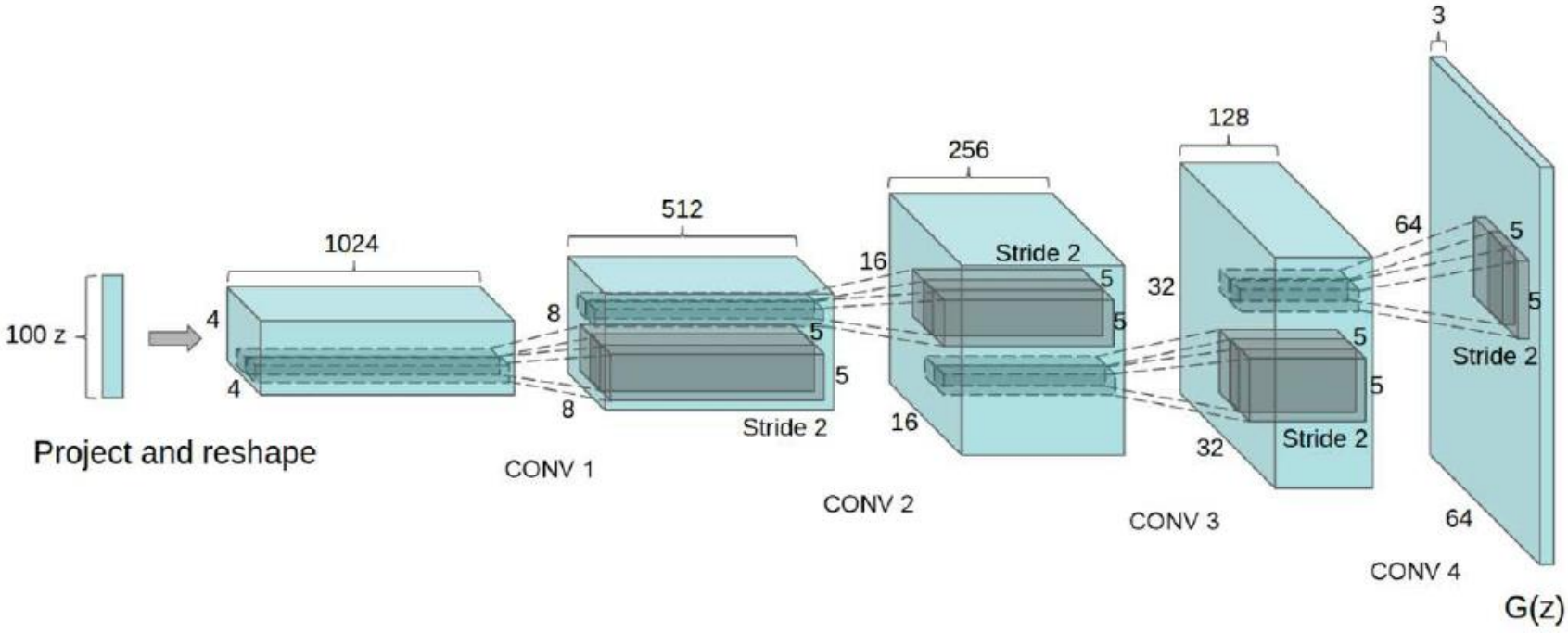



Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution $Z$ is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a $64 \times 64$ 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"])
```

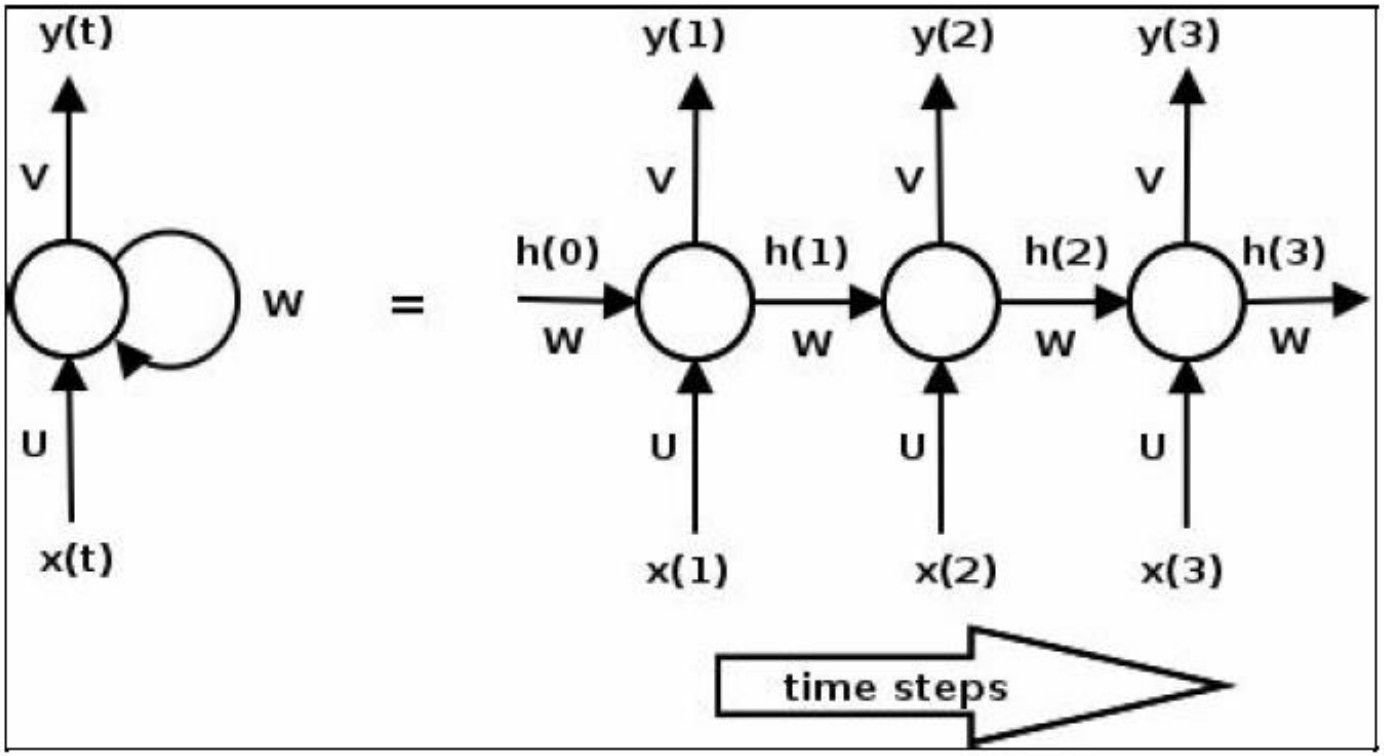
# MNIST

- 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

# RNN




$$\mathbf{h}_t = \tanh(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t)$$
$$\mathbf{y}_t = \mathrm{softmax}(\mathbf{V}\mathbf{h}_t)$$

# SimpleRNN with Keras — generating text

- RNNs have been used extensively by the natural language processing (NLP) community for various applications.
- One such application is building language models. That allows us to predict the probability of a word in a text given the previous words.
- Language models are important for various higher level tasks such as machine translation, spelling correction, and so on.
- In language modeling, our input is typically a sequence of words and the output is a sequence of predicted words.
- The training data used is existing unlabeled text, where we set the label yt at time t to be the input xt+1 at time t+1.

# RNN

- For our first example of using Keras for building RNNs, we will train a character based language model on the text of Alice in Wonderland to predict the next character given 10 previous characters.

- We have chosen to build a character-based model here because it has a smaller vocabulary and trains quicker.

- The idea is the same as using a word-based language model, except we use characters instead of words. We will then use the trained model to generate some text in the same style.

# Code

- Imports
- Read the text and clean up

- Since we are building a character-level RNN, our vocabulary is the set of characters that occur in the text. There are 42 of them in our case. Since we will be dealing with the indexes to these characters rather than the characters themselves, the following code snippet creates the necessary lookup tables

# Input and Labels

- The next step is to create the input and label texts. We do this by stepping through the text by a number of characters given by the STEP variable (1 in our case) and then extracting a span of text whose size is determined by the SEQLEN variable (10 in our case).

- The next character after the span is our label character:

# Example

- Using the preceding code, the input and label texts for the text it turned into a pig would look like this:

- **it turned -> i**
- **t turned i -> n**
- **turned in -> t**
- **turned int -> o**
- **urned into ->**
- **rned into -> a**
- **ned into a ->**
- **ed into a -> p**
- **d into a p -> i**
- **into a pi -> g**

# Vectorize the input

- Each row of the input to the RNN corresponds to one of the input texts shown previously.
- There are SEQLEN characters in this input, and since our vocabulary size is given by nb_chars, we represent each input character as a one-hot encoded vector of size (nb_chars).
- Thus each input row is a tensor of size (SEQLEN and nb_chars).
- Our output label is a single character, so similar to the way we represent each character of our input, it is represented as a one-hot vector of size (nb_chars). Thus, the shape of each label is nb_chars

# Build the model

- We define the RNN's output dimension to have a size of 128. This is a hyper-parameter that needs to be determined by experimentation.
- In general, if we choose too small a size, then the model does not have sufficient capacity for generating good text, and you will see long runs of repeating characters or runs of repeating word groups.
- On the other hand, if the value chosen is too large, the model has too many parameters and needs a lot more data to train
- We want to return a single character as output, not a sequence of characters, so return_sequences=False. We have already seen that the input to the RNN is of shape (SEQLEN and nb_chars). In addition, we set unroll=True because it improves performance on the TensorFlow backend.

# Model

- The RNN is connected to a dense (fully connected) layer. The dense layer has (nb_char) units, which emits scores for each of the characters in the vocabulary. The activation on the dense layer is a softmax, which normalizes the scores to probabilities.
- The character with the highest probability is chosen as the prediction. We compile the model with the categorical cross-entropy loss function, a good loss function for categorical outputs, and the RMSprop optimizer

# Results

- Generating the next character or next word of text is not the only thing you can do with this sort of model. This kind of model has been successfully used to generate classical music (for more information refer to the article: *DeepBach: A Steerable Model for Bach Chorales Generation*, by G.
- Hadjeres and F. Pachet, arXiv:1612.01010, 2016

- Generating fake Wikipedia pages,
- Algebraic geometry proofs, and Linux source code in his blog post at: *The Unreasonable Effectiveness of Recurrent Neural Networks* at http://karpathy.github.io/2015/05/21/rnn-effectiveness/.