SAMSUNG

Samsung Innovation Campus

Artificial Intelligence Course

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 9.

Various Deep Learning Topics **(Deep Learning Techniques for Video and Language Intelligence)**

Artificial Intelligence Course

Chapter Description

Chapter objectives

- Build and train the convolutional neural networks for image classification.
- Convolutional neural networks for detecting locally correlated patterns in images.
- Build and train the recurrent neural networks for time series forecast and natural language processing.
- Brief overview on the LSTM.
- AutoEncoders for dimensional reduction and feature extraction.
- Brief introduction to the generative adversarial networks (GAN).

Chapter contents

✔ Unit 1. About the CNN model

✔ Unit 2. Recurrent Neural Network (RNN) for Sequential Data Modeling

✔ Unit 3. Generative Adversarial Neural Network to Create Non-Existent Images

Unit 1.

About the CNN model

1.1. About the Convolutional Neural Network (CNN)

1.2. Components of Convolutional Neural Network (CNN)

1.3. Building Deep Convolution Neural Network Using Basic Components

1.4. Building Convolutional Neural Network using TensorFlow

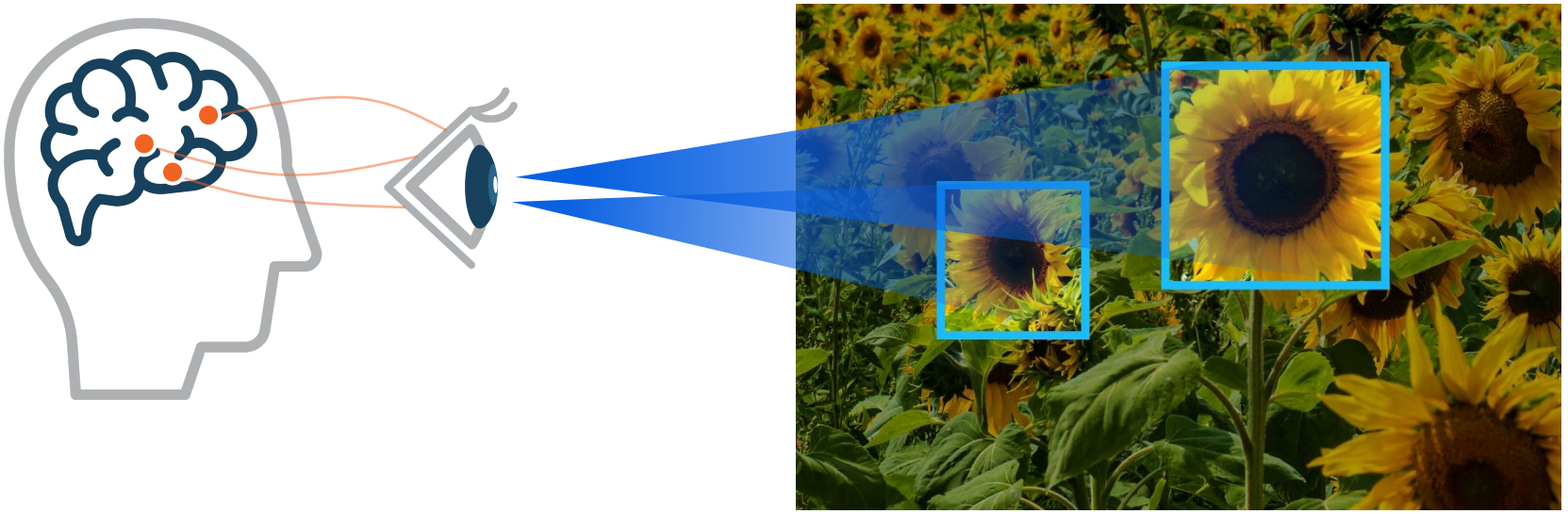
Convolutional Neural Network (CNN)

About the Convolutional Neural Network (CNN):

- CNN is a deep neural network commonly used for image classification.
- Background:
  - In 1990s, Yann LeCun's LeNet-5 was one of the pioneering examples of the CNN.
  - In 2012, Alex Krizhevsky's AlexNet winner of the ImageNet challenge.
  - In 2014, C. Szegedy's GoogleNet winner of the ImageNet challenge.

About the Convolutional Neural Network (CNN):

- Also has a biological origin.
- In the brain, the visual cortex recognizes an image in localized patches.



Convolution filter:

- Given two functions f and g, the convolution is denoted using an asterisk: $f * g$.
- If f and g are functions of a continuous variable, the convolution is an integral:

$$(f * g) = (x) \int f(x')g(x - x')dx'$$

- If f and g are functions of a discrete variable, the convolution is a sum:

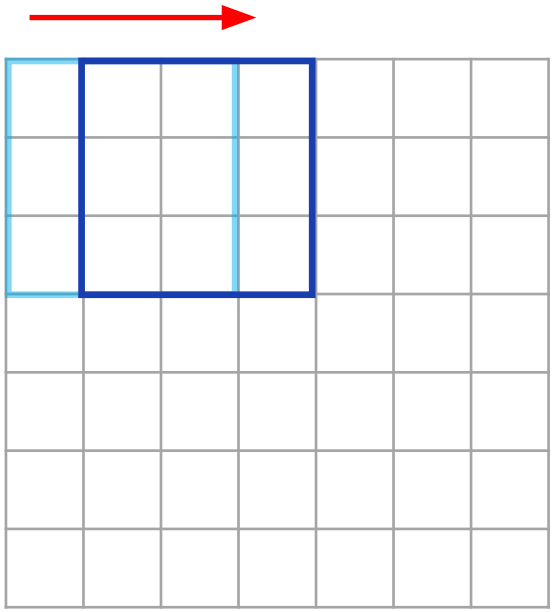
$$(f * g)(i) = \sum_{i'} f(i')g(i - i')$$

  - f can be the so-called 'kernel' that acts like a filter and g be the image.
  - When a 2D image G is filtered by a finite size 2D kernel K, we have:

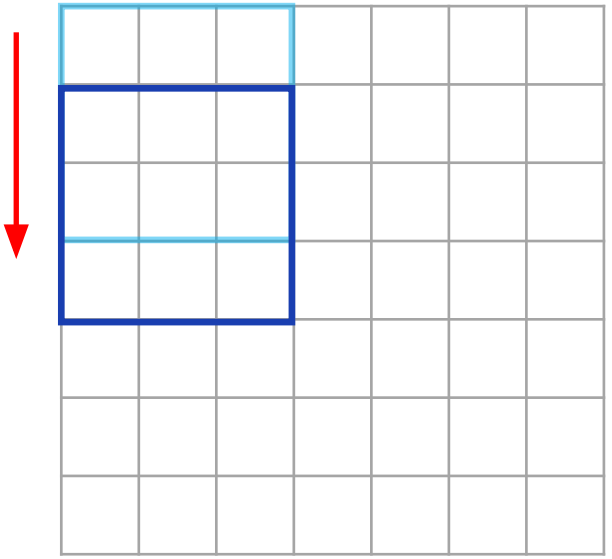
$$(K * G)(i, j) = \sum_{(i',j') \in K} K(i', j')G(i + i', j + j')$$

Convolution filter:

- Moving the filter (kernel) through the image $\Leftrightarrow$ Moving (i, j) of $(K * G)(i, j)$.

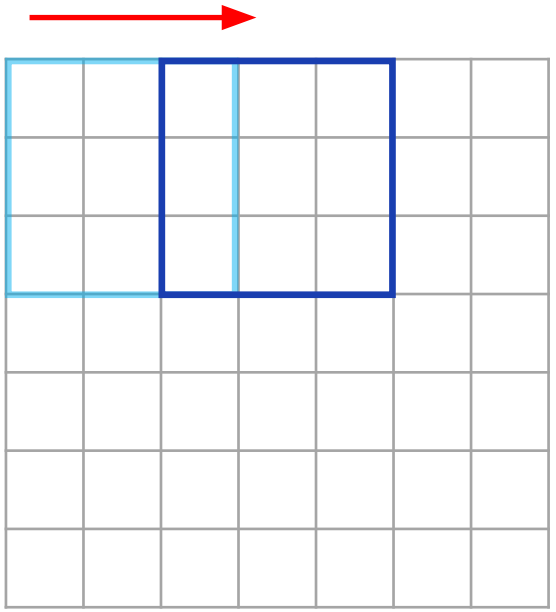


Stride = (1,1)

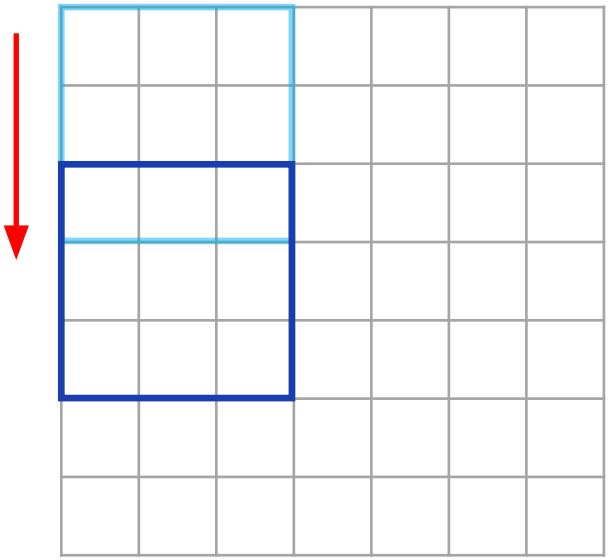


Convolution filter:

- Moving the filter (kernel) through the image $\Leftrightarrow$ Moving (i, j) of $(K * G)(i, j)$.

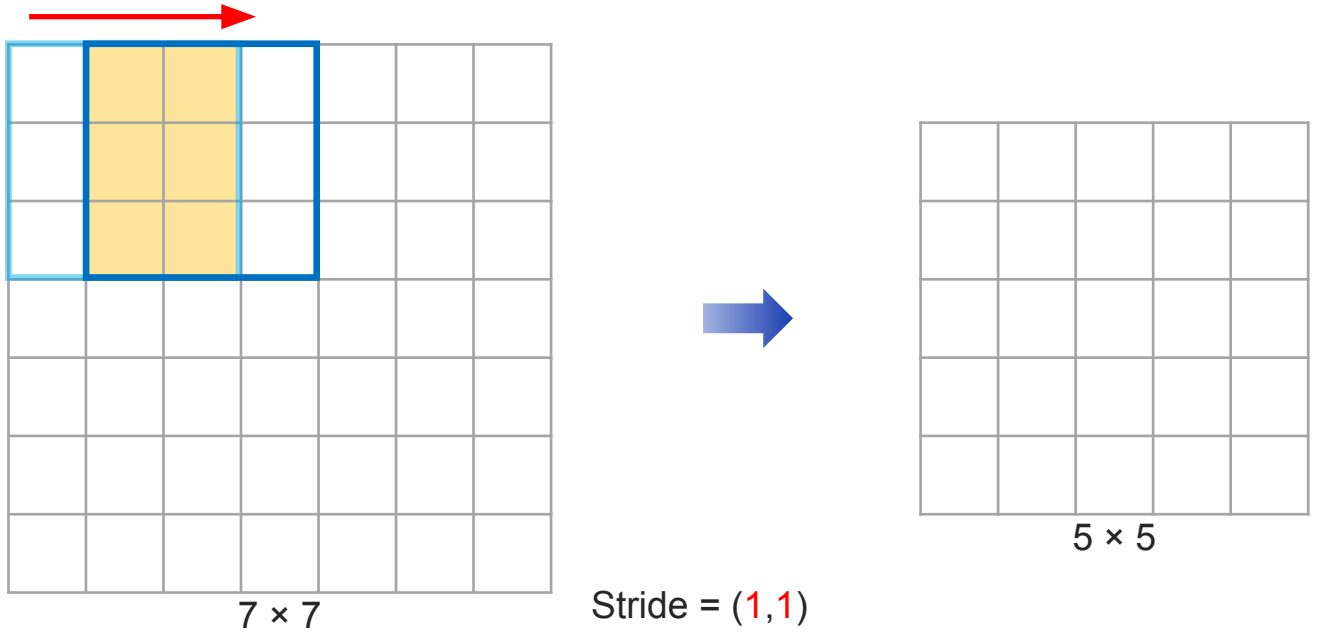


Stride = (2,2)



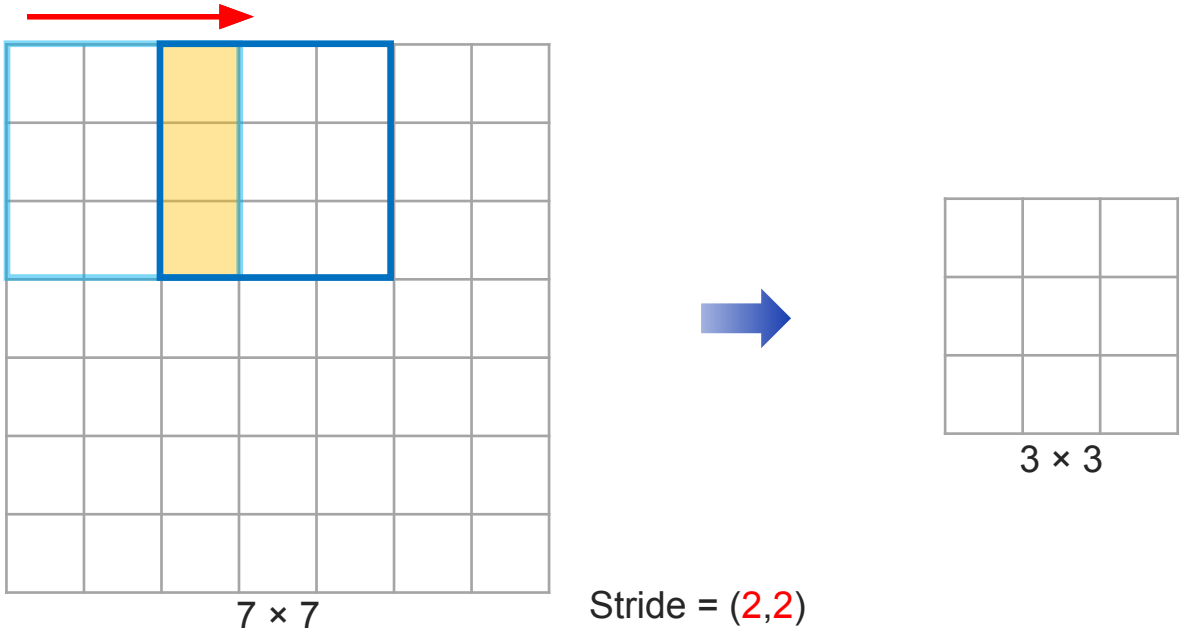
Convolution filter:

- The filtered image suffers from a undesirable side effect: size reduction.



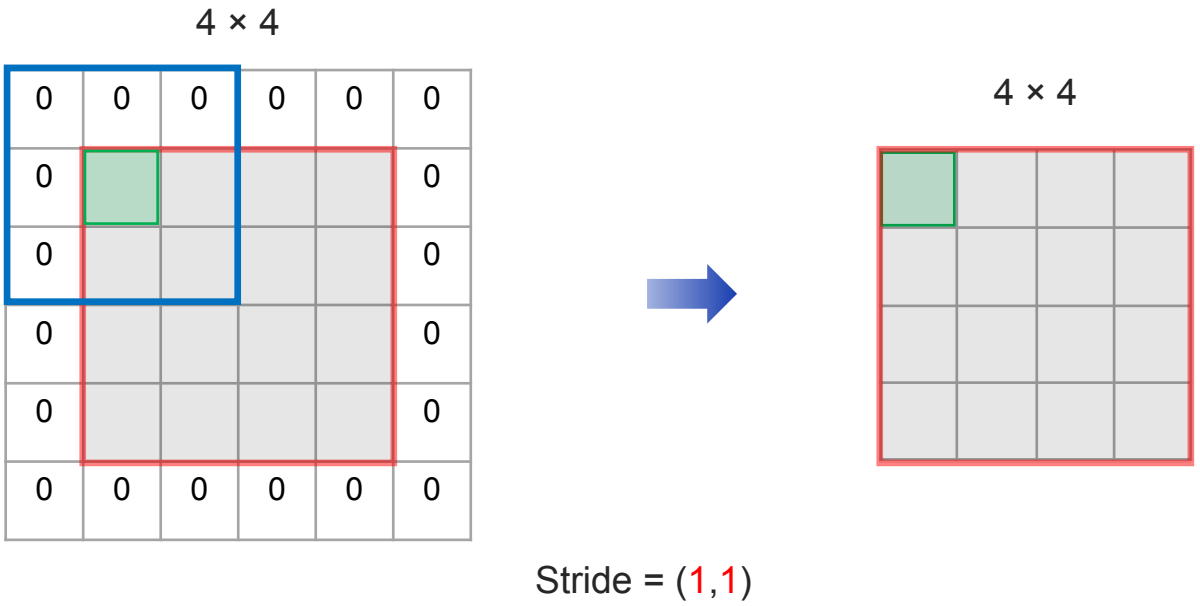
Convolution filter:

- The filtered image suffers from a undesirable side effect: size reduction.

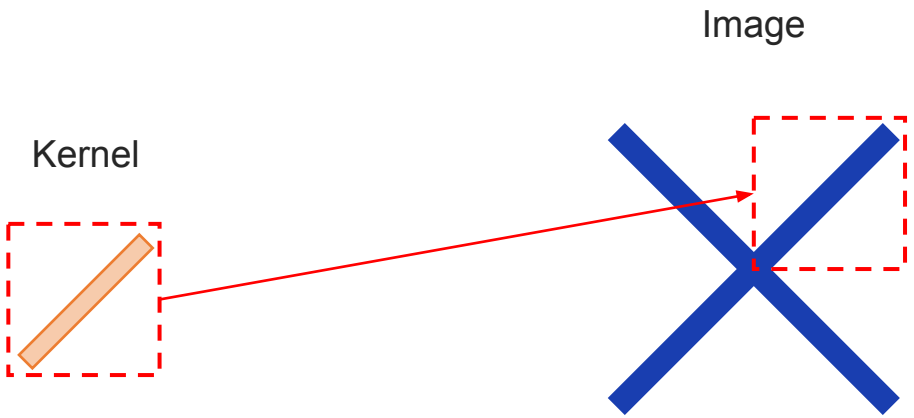


Convolution filter:

- Padding: additional cells with 0 are attached to the outer boundary, so that the size is maintained.



Convolution filters can be used to detect local patterns:



Convolution filters can be used to detect local patterns:

Kernel matrix

0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0

$*$

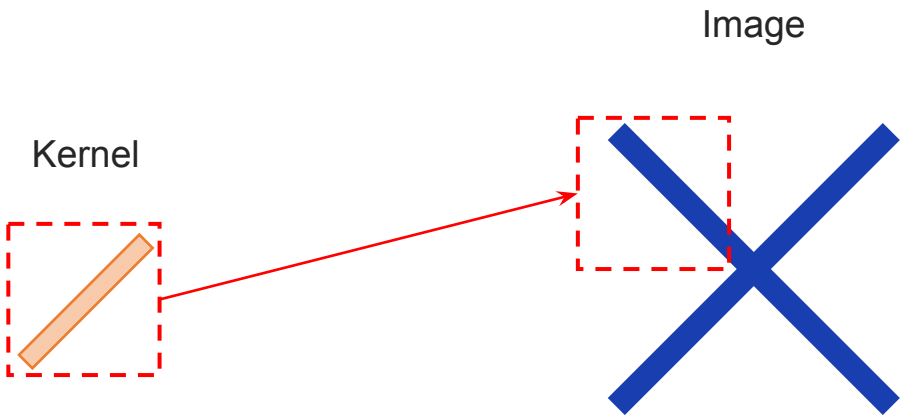
Local image matrix

0	0	0	0	0	0	255
0	0	0	0	0	255	0
0	0	0	0	255	0	0
0	0	0	255	0	0	0
0	0	255	0	0	0	0
0	255	0	0	0	0	0
255	0	0	0	0	0	0

- Apply the left kernel to the right image by convolution.

  $\rightarrow 1 \times 255 + \cdots + 1 \times 255 = 7 \times 255 = 1785$ (several pixels match).

Convolution filters can be used to detect local patterns:



Convolution filters can be used to detect local patterns:

Kernel matrix

0	0	0	0	0	0	1
0	0	0	0	0	1	0
0	0	0	0	1	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
1	0	0	0	0	0	0

$*$

Local image matrix

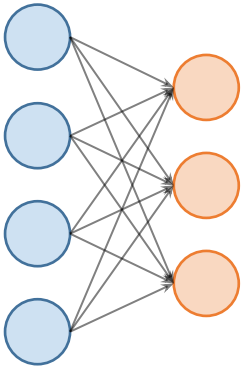
255	0	0	0	0	0	0
0	255	0	0	0	0	0
0	0	255	0	0	0	0
0	0	0	255	0	0	0
0	0	0	0	255	0	0
0	0	0	0	0	255	0
0	0	0	0	0	0	255

- Apply the left kernel to the right image by convolution.

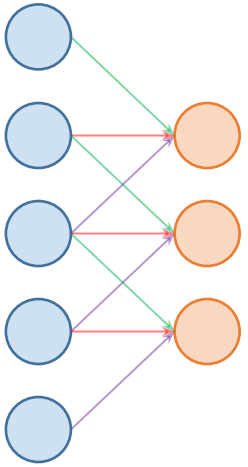
  $\rightarrow 1 \times 255 = 255$ (there is only one match).

Convolution filters in a neural network:

- It is possible to implement a convolution filter with the neural network.
- The edges only connect locally ⇒ just like when a kernel is applied.
- The weights are the kernel elements ⇒ can be trained with data!



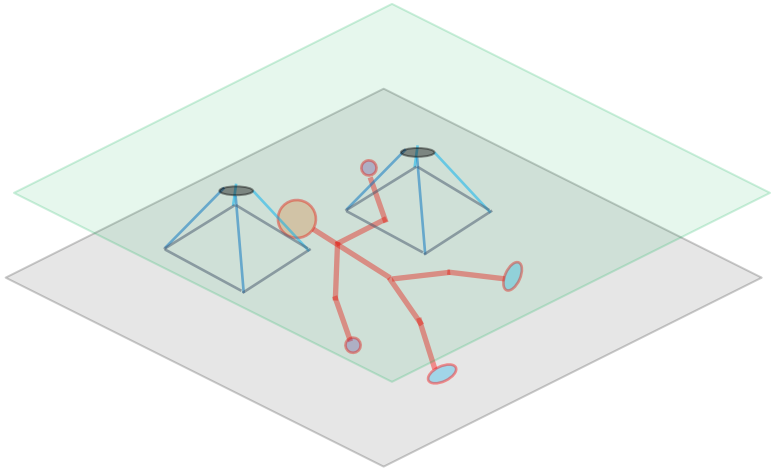
Fully Connected



Convolution

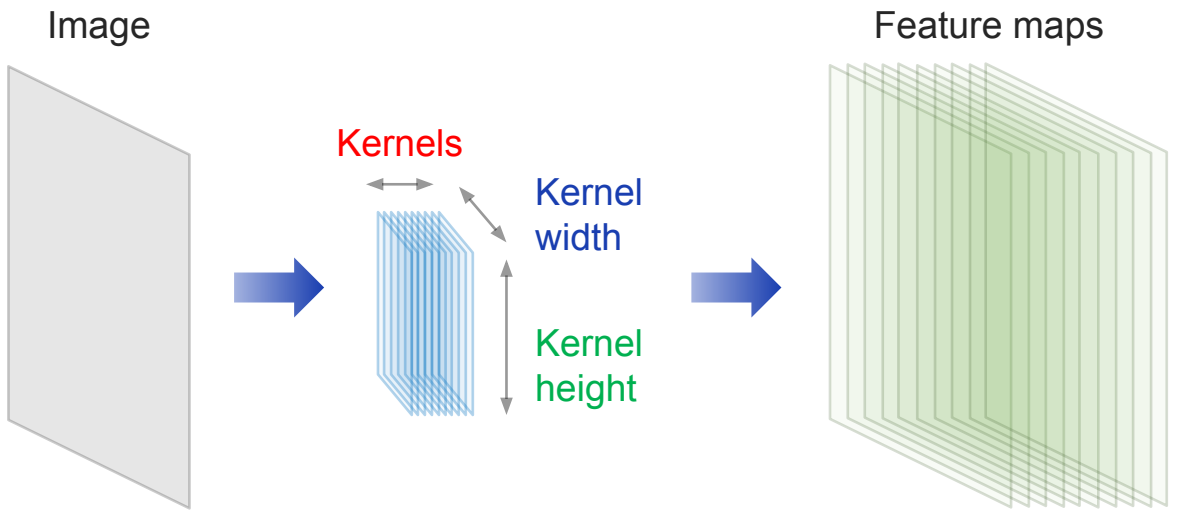
Convolution layer:

- The image is scanned with a kernel and the next layer (feature map) is produced.



Convolution layer:

- Usually we have a set of kernels (filters) with a fixed size $width \times height$.
- A kernel produces a feature map; so as many feature maps are produced as the number of kernels.
- The color channels (Red, Green, Blue) can be convoluted with separate kernels.



Pooling (sub-sampling):

- Move the kernel and summarize. Pooling helps in preventing the overfitting problem.

Ex Max-pooling

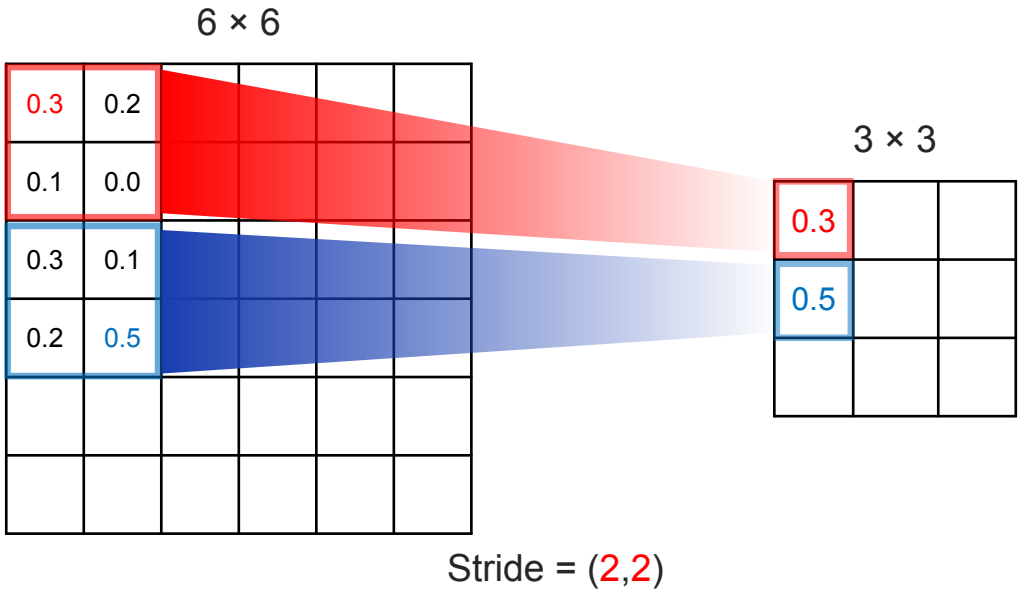
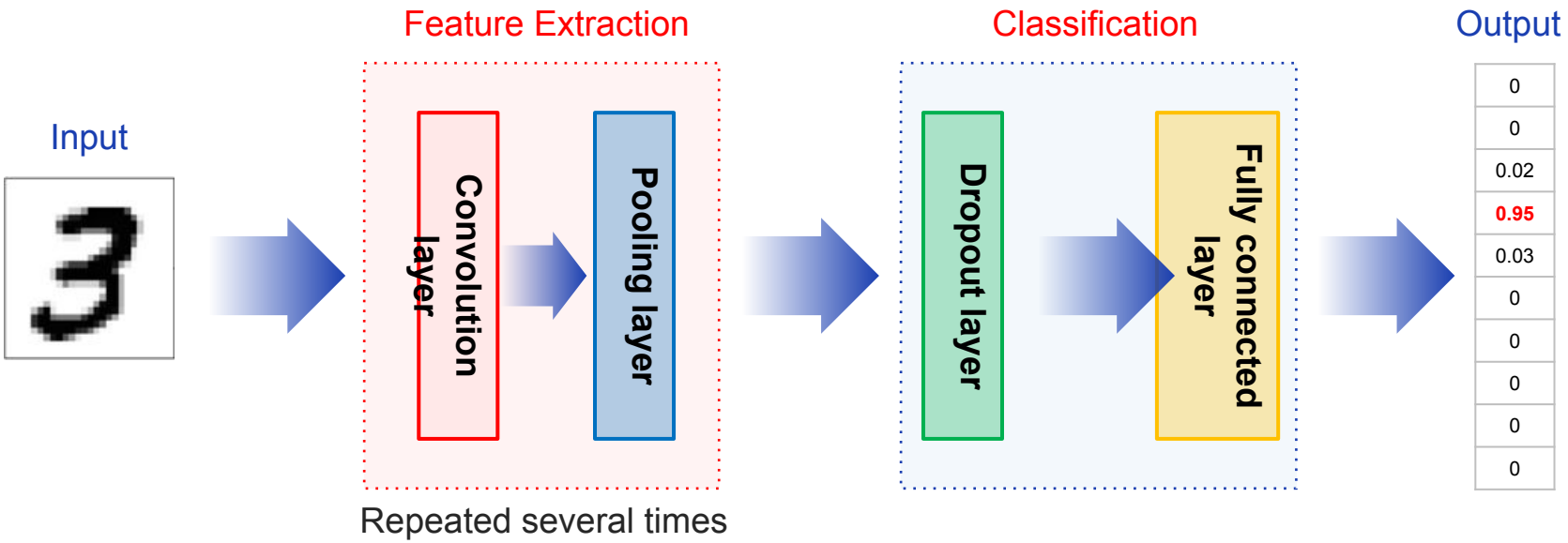


Diagram of a Convolutional Neural Network (CNN):

- The convolution and pooling layers are repeated several times.
- There is a fully connected layer just before the output.



Coding Exercise #0701

Follow practice steps on ‘0701.ipynb' file

Coding Exercise #0702

Follow practice steps on ‘0702.ipynb' file

Unit 1.

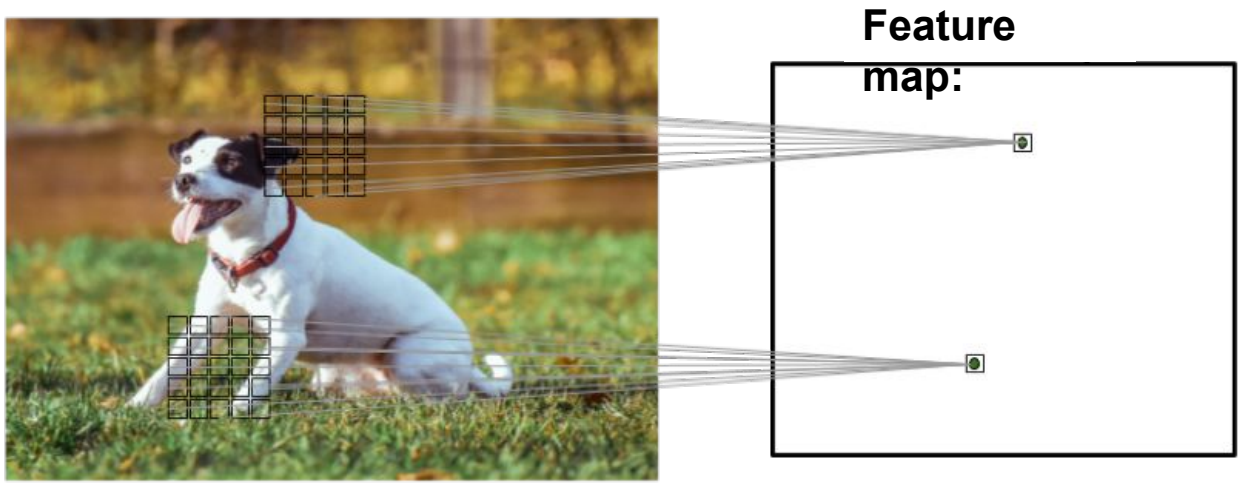
About the CNN model

- 1.1. About the Convolutional Neural Network (CNN)
- 1.2. Components of Convolutional Neural Network (CNN)
- 1.3. Building Deep Convolution Neural Network Using Basic Components
- 1.4. Building Convolutional Neural Network using TensorFlow

Components of Convolutional Neural Network (CNN)

CNN and learning specific classes

- CNN can automatically learn the most useful features for the task from the original data.
- The figure below shows a feature map generated from an input image.



1.2. Components of Convolutional Neural Network (CNN)

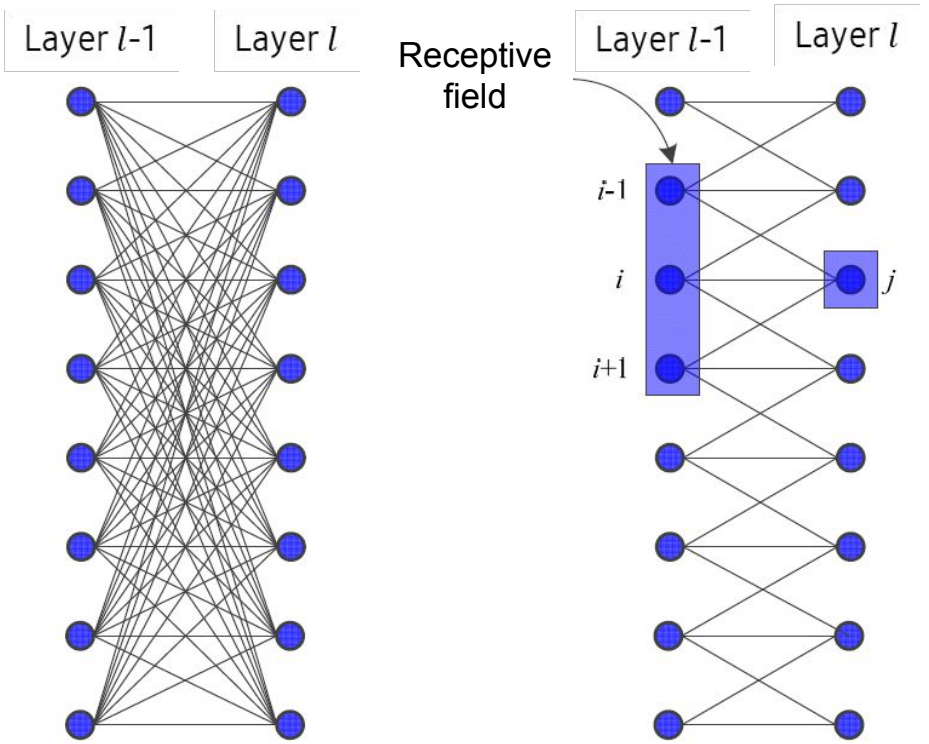
DMLP and CNN

- DMLP
  - High complexity due to fully connected structure
  - Slow learning rate and possibility of overfitting
- CNN
  - Partially connected structure (sparse connection) using convolution operation to reduce complexity.
  - Convolution operation extracts useful features

Suitable for data of grid structure (audio, video etc.)

Its receptive field is analogous to human vision.

Able to process inputs of variable size



(a) DMLP (fully connected)

(b) CNN (partially connected)

Partially connected structure and the receptive field of the CNN

Convolution Operation

- Convolution is a linear operation that multiplies the corresponding values and produces their sum.

u is a kernel, *z* is an input, *s* is an output (feature map)

$$s(i) = z \circledast u = \sum_{x=-(h-1)/2}^{(h-1)/2} z(i+x)u(x)$$

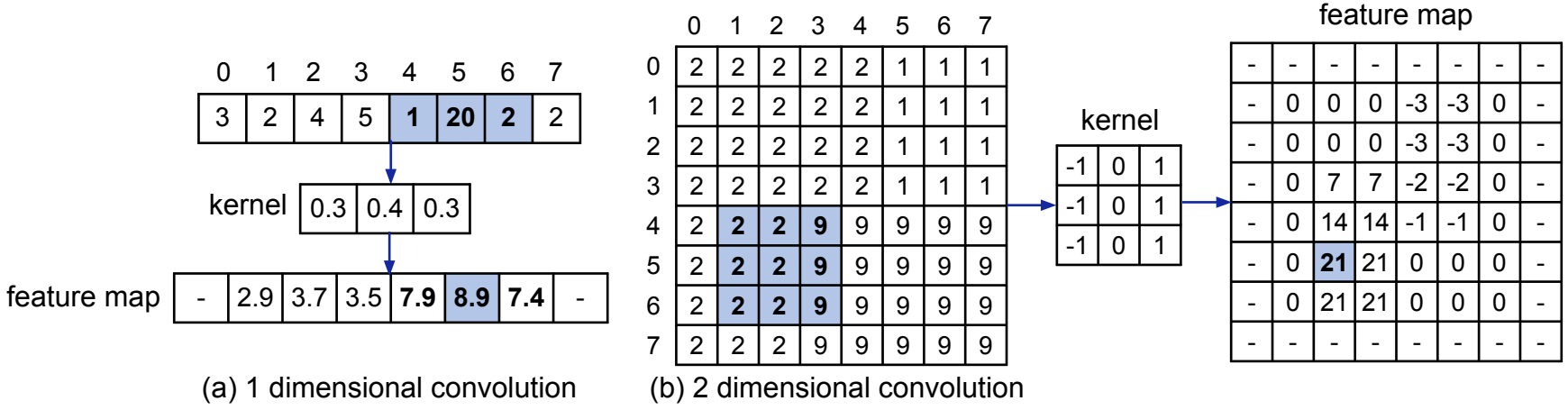
← **1 dimensional input**

$$s(j,i) = z \circledast u = \sum_{y=-(h-1)/2}^{(h-1)/2} \sum_{x=-(h-1)/2}^{(h-1)/2} z(j+y, i+x)u(y,x)$$

← **2 dimensional input**

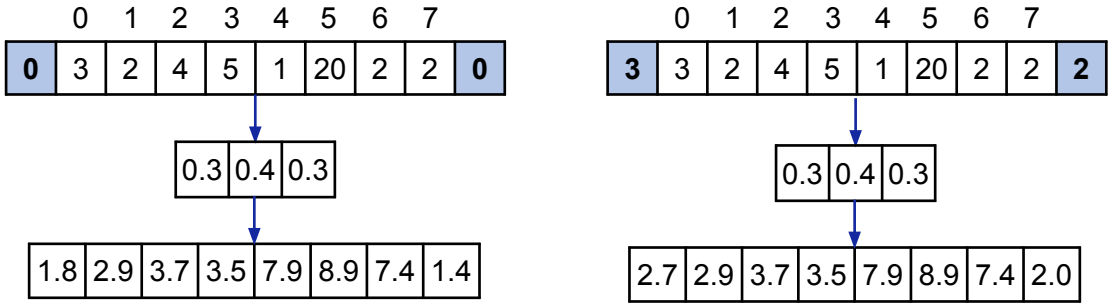
Convolution Operation

- Convolution is a simple linear operation that produces the sum of all the multiplied results of each corresponding units.
- On the left figure, if you execute convolution operation that adds all the results of the multiplication of corresponding units of the values of the receptive field (1,20,2) and the kernel (0.3, 0.4, 0.3), the result is 1*(0.3) + 20*0.4 +2*0.3=8.9. If you move the kernel from left to right, such operation is performed on all nodes, and the operation is complete.
- In two-dimensions, convolution operation is similar to that of one-dimension. The receptive field of the node is $\begin{pmatrix} 2 & 2 & 9 \\ 2 & 2 & 9 \\ 2 & 2 & 9 \end{pmatrix}$ and the kernel is $\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$, so the result of the operation is 21. If you move the kernel and apply the same operation to all nodes, you obtain a feature map on the right.



Padding

- Operation at the border of data is impossible, since the kernel protrudes outside the data's area.
- In the previous slide, areas impossible for operation was denoted with -. If a kernel's size is h, and h is 3, the number of the nodes becomes 2 in the border. In deep neural networks, convolution layers are repeated multiple times. Then, some nodes are omitted to a great degree.
- To prevent such a problem, it performs what is called padding as shown below. It pads a required amount of 0 or duplicates the value of the adjacent node.



(a) Padding 0 (b) Padding the adjacent value

Padding (grey colored nodes are padded nods)

Exercise : computing convolution output shape

In [7]:
```
def conv1d(x, w, p=0, s=1):
    w_rot = np.array(w[::-1])
    x_padded = np.array(x)
    if p > 0:
        zero_pad = np.zeros(shape=p)
        x_padded = np.concatenate(
            [zero_pad, x_padded, zero_pad])
    res = []
    for i in range(0, int((len(x_padded) - len(w_rot)) / s) + 1, s):
        res.append(np.sum(
            x_padded[i:i+w_rot.shape[0]] * w_rot))
    return np.array(res)


## Test:
x = [1, 3, 2, 4, 5, 6, 1, 3]
w = [1, 0, 3, 1, 2]

print('Conv1d rendered:',
      conv1d(x, w, p=2, s=1))

print('NumPy Result:',
      np.convolve(x, w, mode='same'))
```

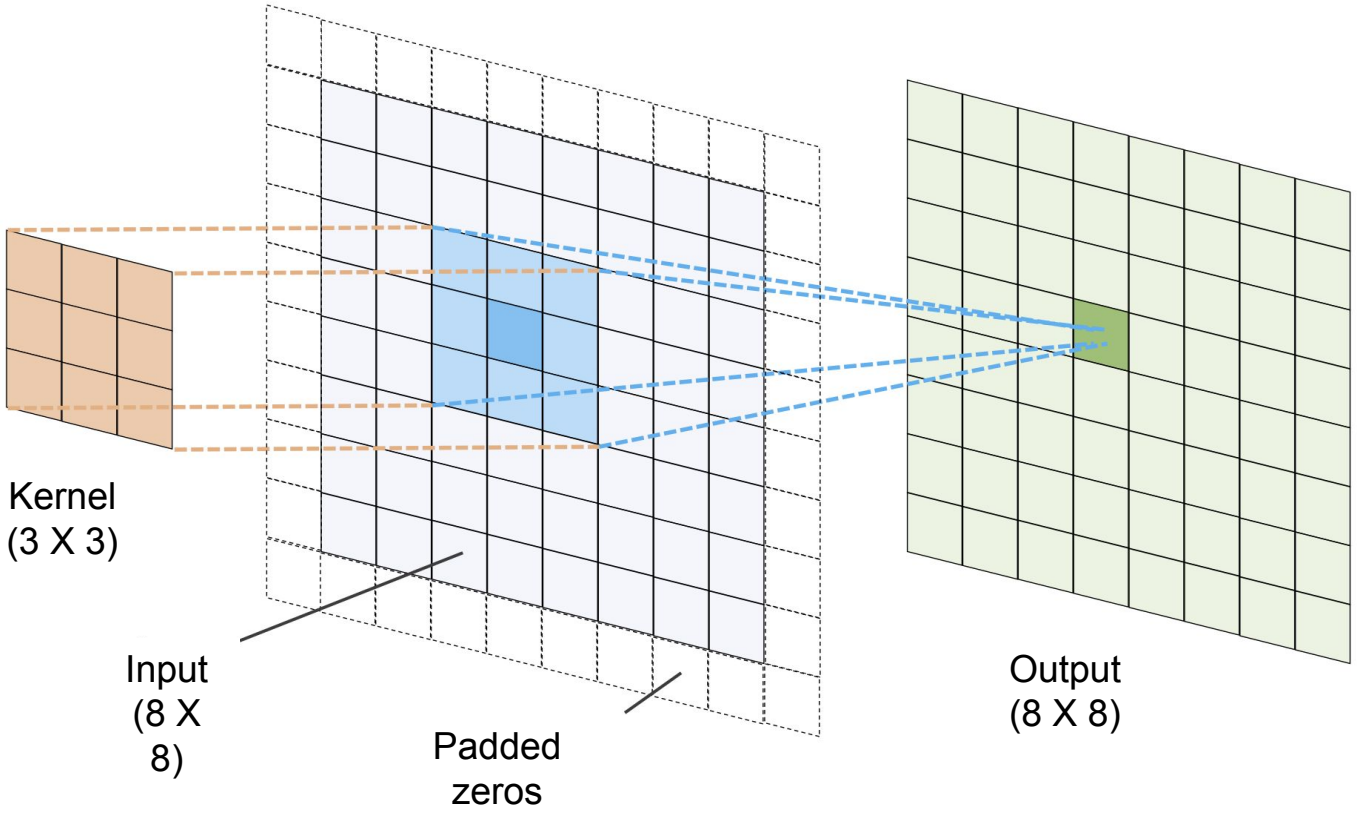
```
Conv1d rendered: [ 5. 14. 16. 26. 24. 34. 19. 22.]
  NumPy Result: [ 5 14 16 26 24 34 19 22]
```

[31]:
```
import tensorflow as tf
import numpy as np

print    ('TensorFlow version:', tf.__version__)
print('Numpy Version:', np.__version__)
```

```
TensorFlow version : 2.5.0
  Numpy Version:  1.19.2
```

2D discrete convolution

- The figure below shows a convolution of 8 x 8 input matrix and 3 x 3 kernel.
- p=1 and zeros are padded in the input matrix. Thus, 2D convolution generates an output of 8 x 8 size.



2D discrete convolution

- The code below is to implement 2D discrete convolution with the con2D function.

In [11]:
```
import scipy.signal


def conv2d(X, W, p=(0, 0), s=(1, 1)):
    W_rot = np.array(W)[::-1,::-1]
    X_orig = np.array(X)
    n1 = X_orig.shape[0] + 2*p[0]
    n2 = X_orig.shape[1] + 2*p[1]
    X_padded = np.zeros(shape=(n1, n2))
    X_padded[p[0]:p[0]+X_orig.shape[0],
    p[1]:p[1]+X_orig.shape[1]] = X_orig

    res = []
    for i in range(0, int((X_padded.shape[0] -
                           W_rot.shape[0])/s[0])+1, s[0]):
        res.append([])
        for j in range(0, int((X_padded.shape[1] -
                               W_rot.shape[1])/s[1])+1, s[1]):
            X_sub = X_padded[i:i+W_rot.shape[0],
                             j:j+W_rot.shape[1]]
            res[-1].append(np.sum(X_sub * W_rot))
    return(np.array(res))
```

2D discrete convolution

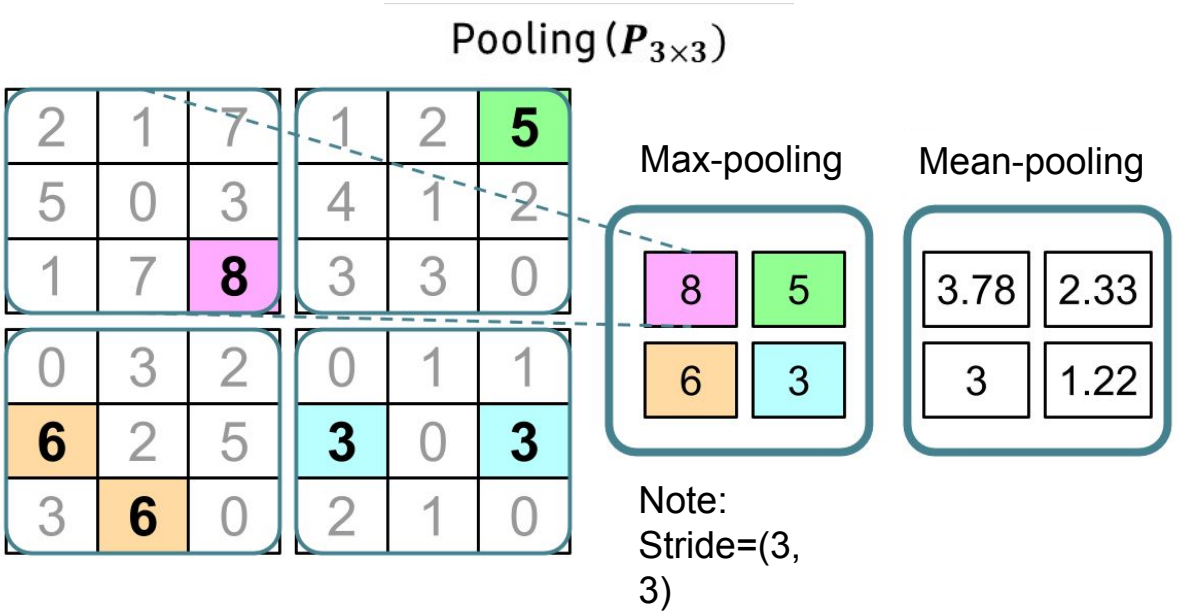
- In scipy.signal, the function scipy.signal.convolve2d computes 2D discrete convolution.

```
X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]

print('Rendering Conv2d:\n',
    conv2d(X, W, p=(1, 1), s=(1, 1)))


print('scipy results:\n',
    scipy.signal.convolve2d(X, W, mode='same'))
```

```
Rendering
Conv2d:
 [[11. 25. 32. 13.]
 [19. 25. 24. 13.]
 [13. 28. 25. 17.]
 [11. 17. 14.  9.]]
scipy results:
 [[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

Sub-sampling

- Sub-sampling is a typical pooling operation of two kinds that are applied to CNN
- Those are maxpooling and mean-pooling (or average-pooling)
- Generally, the pooling layer is denoted as Pn1×n2.
- The subscript denotes the maximum value and the size of the adjacent pixel for mean-pooling.
- The number of the adjacent pixels is the size of of pooling



Benefits of Pooling

- Pooling increases computation efficiency since it reduces the feature's size.
- Reduction of features also reduces chances of overfitting
- Pooling is still a central component of CNN structure, but some CNN structure is developed without using pooling layers.
- They use convolution layer of stride 2 instead of pooling layer to reduce the feature map's size.
- A convolutional layer of stride 2 can be considered as a pooling layer with learning weight.
- For more information, refer to theses that compare various CNN structures that use and don't use pooling layers.

Unit 1.

About the CNN model

1.1. About the Convolutional Neural Network (CNN)

1.2. Components of Convolutional Neural Network (CNN)

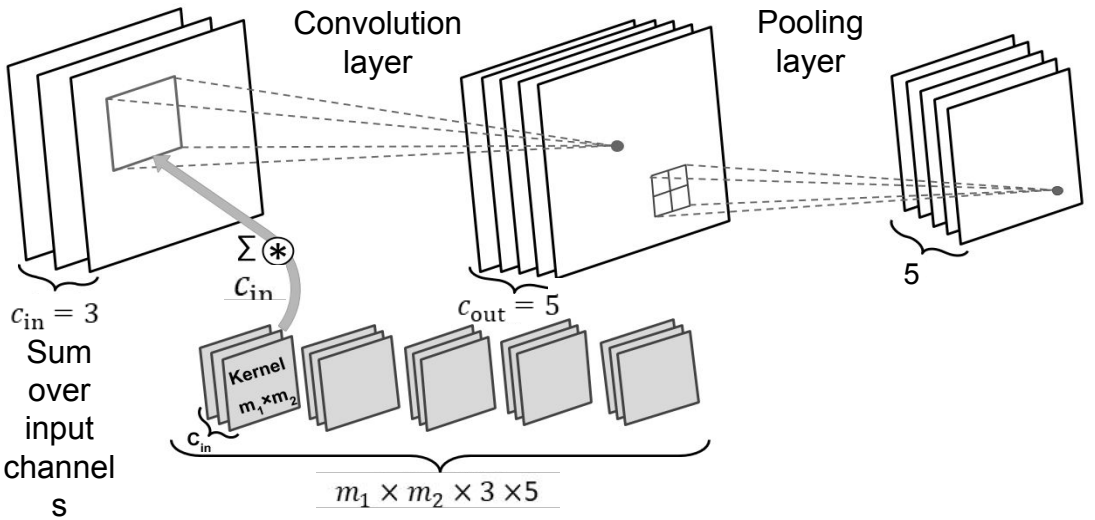
1.3. Building Deep Convolution Neural Network Using Basic Components

1.4. Building Convolutional Neural Network using TensorFlow

Building Deep Convolutional Neural Network using Basic Components

Managing mutiple inputs or color channels

- Let's formulate a CNN computation in the exercise below which includes convolution layer and pooling layer.
- In the exercise there are three input channels.
- The kernel tensor is 4 dimensional.
- Each kernel matrix has a size of m1×m2 . Each input matrix has one kernel, so total three kernels.
- Total five kernel tensors to make five input feature maps.
- There is a pooling layer to sub-sample the feature map.



Regularization to prevent overfitting in neural networks (1/3)

- For both fully connected neural network and CNN, it is difficult to determine the size of the network.
- For quality performance, you need to tune the size of the weight matrix and the number of layers.
- The capacity of a network indicates how complex a function can be approximated.
- Networks with a relatively small number of parameters are prone to underfitting due to their small capacity.
- The performance deteriorates because it cannot learn the inherent structure of the dataset.
- However, a large network can easily become overfitted.
- If such network learns the training data, it will work fine on the training dataset, but it will not in other datasets.

Regularization to prevent overfitting in neural networks (2/3)

- In actual machine learning problems, you cannot know the appropriate size of the network beforehand.
- You can solve such problem with a method described below.
- Frist, construct a relatively large network (one larger than actually needed) so that it works smoothly with a training dataset.
- Then, to prevent overfitting, apply more than one regularization method to improve generalization performance on new datasets such as other test datasets.
- This can prevent or reduce effect of overfitting because regularization L1 and regularization L2 adds penalty to the loss function, thus reducing the size of the weight during training.
- Both L1 regularization and L2 regularization can be used for neural networks, but L2 is used more frequently.
- Then, there are methods such as dropout to regularize neural networks.

Regularization to prevent overfitting in neural networks (3/3)

- With Keras API, as in the figure below, you can easily insert L2 penalty to the loss function with kernel regularizer parameter. (this is automatically reflected on the loss function)

In [20]:
```
from tensorflow import keras


conv_layer = keras.layers.Conv2D(
    filters=16, kernel_size=(3, 3),
    kernel_regularizer=keras.regularizers.l2(0.001))

fc_layer = keras.layers.Dense(
    units=16, kernel_regularizer=keras.regularizers.l2(0.001))
```

Loss functions for classification

- In the previous chapters, you have learned many activation functions such as ReLu, sigmoid, tanh etc.
- Some activation functions like ReLu uses hidden layer in the middle of the neural network to add non-linearity.
- Functions such as sigmoid (for binary classification) and softmax (for multiclass classification) prints the probability of class inclusion by being added to the last layer (output layer).
- If simoid or somftmax activation function is not included in the output layer, the model will compute logit instead of probability of class inclusion.
- If classification is the main issue, you should choose a suitable loss function for model training according to the type of the problem (binary or multiclass) and type of the output (logit or probability)
- **Binary cross-entropy** is a loss function for binary classification (with one output unit).
- **Categorical cross-entropy** is a loss function for multiclass classification
- Keras API offers two options for categorial cross-entropy loss.
- In one option, the label is provided in one-hot representation (for example [0, 0, 1, 0]. In the other, an integer label (for example y=2)
- For the latter, the term is also referred to as ‘sparse’ expression in Keras.

Loss functions for classification

- The following chart shows three loss functions usable in Keras for multi-class classification in one-hot encoded labels, and multi-class classification for integer labels.
- The three loss functions offer two options, either to receive logit or class inclusion possibility for the probability value.

Loss function	Usage	Example	
		Using probabilities *from_logits=False*	Using logits *from_logits=True*
`BinaryCrossentropy`	Binary Classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
`CategoricalCrossentropy`	Multiclass Classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
`Sparse CategoricalCrossentropy`	Multiclass Classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

Loss functions for classification

- It is generally preferred to compute cross entropy loss by logit instead of class inclusion probability, for numerical stability.
- By using logit as an input for the loss function and designating logits=True, the corresponding TensorFlow function will efficiently compute derivate of the loss function for the loss and weight.
- This is because you do not need to compute certain mathematical sections when logit is provided as input.

The following code shows how to use three loss functions when either logit or class inclusion probability was given as an input.

In [55]:
```
from distutils.version import LooseVersion as Version


 ######## Binary cross entropy
bce_probas = tf.keras.losses.BinaryCrossentropy(from_logits=False)
bce_logits = tf.keras.losses.BinaryCrossentropy(from_logits=True)

logits = tf.constant([0.8])
probas = tf.keras.activations.sigmoid(logits)

tf.print(
    'BCE (probability): {:.4f}'.format(
    bce_probas(y_true=[1], y_pred=probas)),
    '(logit): {:.4f}'.format(
    bce_logits(y_true=[1], y_pred=logits)))
```

The following code shows how to use three loss functions when either logit or class inclusion probability was given as an input.

```
 ######## Categorial cross entropy
cce_probas = tf.keras.losses.CategoricalCrossentropy(
    from_logits=False)
cce_logits = tf.keras.losses.CategoricalCrossentropy(
    from_logits=True)

logits = tf.constant([[1.5, 0.8, 2.1]])
probas = tf.keras.activations.softmax(logits)

if Version(tf.__version__) >= '2.3.0':
    tf.print(
        'CCE (probability): {:.4f}'.format(
        cce_probas(y_true=[[0, 0, 1]], y_pred=probas)),
        '(logit): {:.4f}'.format(
        cce_logits(y_true=[[0, 0, 1]], y_pred=logits)))
else:
    tf.print(
        'CCE (probability): {:.4f}'.format(
        cce_probas(y_true=[0, 0, 1], y_pred=probas)),
        '(logit): {:.4f}'.format(
        cce_logits(y_true=[0, 0, 1], y_pred=logits)))
```

The following code shows how to use three loss functions when either logit or class inclusion probability was given as an input.

```
  ######## Sparse categorial cross entropy
sp_cce_probas = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=False)
sp_cce_logits = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True)

tf.print(
    'Sparse CCE (probability): {:.4f}'.format(
    sp_cce_probas(y_true=[2], y_pred=probas)),
    '(logit): {:.4f}'.format(
    sp_cce_logits(y_true=[2], y_pred=logits)))
```

```
BCE (probability): 0.3711  (Logit): 0.3711
CCE (probability): 0.5996  (Logit): 0.5996
Sparse CCE (probability): 0.5996  (Logit): 0.5996
```

Loss functions for classification

- At times, categorical cross entropy is used for binary classification.
- Ordinarily, a model returns an output value for each sample in binary classification.
- This output can be seen as probability P [class=1] of the positive class (for example class 1).
- In binary classification problems, P[class=0] = 1 - P[class=1]
- You do not need to add output units to generate probability of negative class.
- Oftentimes, a sample receives two outputs, each identified as probability for each class (P[class=0] or P[class=1])
- In such case, it is recommended to normalize the output (that is, make the sum 1) by using softmax function (instead of logistic sigmoid)
- Here, categorical cross entropy is the appropriate loss function.

Unit 1.

About the CNN model

1.1. About the Convolutional Neural Network (CNN)

1.2. Components of Convolutional Neural Network (CNN)

1.3. Building Deep Convolution Neural Network Using Basic Components

1.4. Building Convolutional Neural Network using TensorFlow

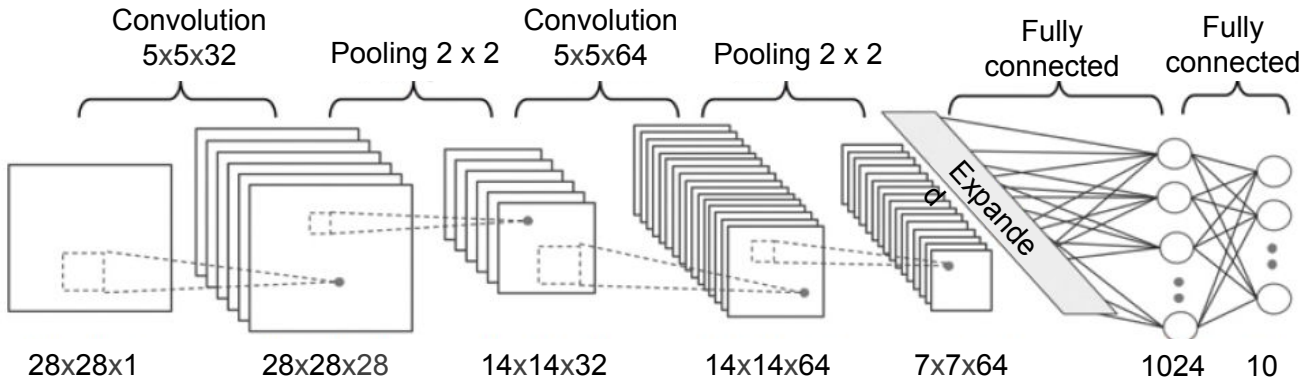
Building Convolutional Neural Network using TensorFlow

Building Convolutional Neural Network using TensorFlow

- So far, you have built a neural network for handwritten number recognition using TensorFlow estimator and used various TensorFlow APIs.
- Achieved 89% accuracy using DNNClassifier which has two hidden layers.
- Here, let’s build CNN for handwritten number recognition and check if it performs better than MLP (DNNClassifier)
- Fully connected aptly solves this problem.
- For applications with functions such as recognizing handwritten bank account numbers, a small mistake could lead to a big loss.
- It is best to reduce errors as much as possible.

Multi-layer CNN struture

- The network structure for this exercise is shown in the figure below.
- The input is a 28 x 28 greyscale image.
- Dimension of the input tensor is batchsize x 28 x 28 x 1 considering the number of the channels (1 since it's a greyscale image) and layout of the batch of the input image.
- Input data passes two convolutional layers with 5 x 5 kernel.
- The first convolution prints 32 feature mas, and the second prints 64 feature maps.
- Each convolutional layer is followed by maxpooling operation P2×2 for subsampling.
- Then, output of the fully connected layer is delivered to the second fully connected layer, which is the final softmax layer.
- The next slide shows the network structure you are building.



1.4. Building Convolutional Neural Network using TensorFlow

Multi-layer CNN structure

- Each layer's tensor dimension is as follows.

```
input: [batchsize×28×28×1]
convolution_1: [batchsize×28×28×32]
pooling_1: [batchsize×14×14×32]
convolution_2: [batchsize×14×14×64]
pooling_2: [batchsize×7×7×64]
fully connected_1: [batchsize×1024]
fully connected and softmax layer: [batchsize×10]
```

- It maintained the dimension of the feature map and the input on the same level by using strides=1 on the convolutional layer.
- The pooling layer sub sampled the image by using strides=2 and reduced the size of the output feature map.
- Let's build this neural network using TensorFlow Keras API.

Loading and Preprocessing Data

- MNIST Dataset

In [24]:
```
import tensorflow_datasets as tfds
import pandas as pd

import matplotlib.pyplot as plt
```

In [25]:
```
mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
datasets = mnist_bldr.as_dataset(shuffle_files=False)
print(datasets.keys())
mnist_train_orig, mnist_test_orig = datasets['train'], datasets['test']
```

```
dict_keys(['train', 'test'])
```

- You've learned two methods to load dataset using tensorflow_datasets module.
- One method is following three steps, and the other, simpler, method is using load function that encompasses these three steps.
- Three steps for loading MNIST dataset is as above.

Loading and Preprocessing Data

- You can split test dataset and validation dataset as follows. (1/2)

In [26]:
```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
NUM_EPOCHS = 20
```

In [27]:
```
mnist_train = mnist_train_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

mnist_test = mnist_test_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

tf.random.set_seed(1)

mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
                                  reshuffle_each_iteration=False)

mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

- MNIST dataset is offered as train dataset and test dataset.
- Here, you will also create a validation dataset from the test dataset.
- In the third step, you designated shuffle_files=False parameter in .as_dataset() method.

Loading and Preprocessing Data

- You can split test dataset and validation dataset as follows. (2/2)

In [26]:
```
BUFFER_SIZE = 10000
BATCH_SIZE = 64
NUM_EPOCHS = 20
```

In [27]:
```
mnist_train = mnist_train_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

mnist_test = mnist_test_orig.map(
    lambda item: (tf.cast(item['image'], tf.float32)/255.0,
                  tf.cast(item['label'], tf.int32)))

tf.random.set_seed(1)

mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
                                  reshuffle_each_iteration=False)

mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

- This prevents initial shuffling.
- This is because it has to split test dataset into smaller test datasets and validation datasets.

Building CNN using TensorFlow Keras API

- Stack convolution, pooling, dropout and fully connected layers by using Keras Sequential class to build CNN in TensorFlow. Kears layer API offers class for each of these layers.
- To build a layer with Conv2D class, designate the number of filters (number of output feature maps) and size of the kernel. In addition, there are more parameters that can modify the operations of the convolutional layer. The most widely used are stride (initial value is 1 in x, y dimensions) and padding. Padding can be either designated as same padding or valid padding.

Conv2D: tf.keras.layers.Conv2D
- filters
- kernel_size
- strides
- padding

MaxPool2D: tf.keras.layers.MaxPool2D
- pool_size
- strides
- padding

Dropout: tf.keras.layers.Dropout2D
- rate

tf.keras.layers.Conv2D: a two-dimensional convolution layer

tf.keras.layers.MaxPool2D and
tf.keras.layers.AvgPool2D
: sub-sampling (maxpooling and meanpooling) layers

- tf.keras.layers.Dropout
  : a layer executing dropout for regularization.

Building CNN using TensorFlow Keras API

- Now that you have learned about class, let's build the CNN model from the previous slide.
- The following code uses Sequential class, and adds convolution and pooling layer

  - Added two convolutional layer to the model.
  - Each convolutional layer uses 5x5 kernel and 'same' padding
  - As mentioned before, using padding='same' preserves the feature map's space directional dimension (vertical and horizontal dimension)
  - Input and output have same height and width (the number of the channels change depending on the number of filters.
  - Max pooling layer of size 2 x2 and stride 2 reduces space directional dimension to its half.
  - (If strides parameter is not designated on MaxPool2D, it is initialized with the size of pooling)

Building CNN using TensorFlow Keras API

In [28]:
```
model = tf.keras.Sequential()

model.add(tf.keras.layers.Conv2D(
    filters=32, kernel_size=(5, 5),
    strides=(1, 1), padding='same',
    data_format='channels_last',
    name='conv_1', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), name='pool_1'))
```

```
model.add(tf.keras.layers.Conv2D(
    filters=64, kernel_size=(5, 5),
    strides=(1, 1), padding='same',
    name='conv_2', activation='relu'))

model.add(tf.keras.layers.MaxPool2D(pool_size=(2, 2), name='pool_2'))
```

Building CNN using TensorFlow Keras API

- In this stage, you can manually compute the size of the feature map, but using Keras API is more convenient.
  - The output shape computed as in response to the input shape designated by a tuple is (16, 7, 7, 64)
  - This is a feature map with 64 channels and height and width of 7 x 7.
  - First dimension is the batch dimension which is randomly designated as 16 here.
  - Instead, it can be designated as None as in input_shape=(None, 28, 28, 1)

```
In [29]: model.compute_output_shape(input_shape=(16, 28, 28, 1))
Out[29]: TensorShape([16, 7, 7, 64])

In [30]: model.add(tf.keras.layers.Flatten())

         model.compute_output_shape(input_shape=(16, 28, 28, 1))
Out[30]: TensorShape([16, 3136])
```

  - The next layer to be added is a dense (or fully connected layer)
  - It is necessary to build classifier on top of convolution and pooling layer.
  - Input of this layer is rank 2, that is [batch size x number of input units]
  - Output of the previous layer must be spread in accordance with the dense layer.

Building CNN using TensorFlow Keras API

- As you can see from the result of compute_output_shape method, input dimension of the dense layer is well set up.
- Then, add two dense layers and insert a dropout layer between the two.

```
In [31]: model.add(tf.keras.layers.Dense(
             units=1024, name='fc_1',
             activation='relu'))

         model.add(tf.keras.layers.Dropout(
             rate=0.5))

         model.add(tf.keras.layers.Dense(
             units=10, name='fc_2',
             activation='softmax'))
```

- fc_2, which is the fully connected layer, has ten output units that correspond to ten class labels from MNIST dataset.
- Moreover, it uses softmax activation function to acquire class inclusion probability of the input sample.
- Sum of each sample's probability is 1. (one sample can belong to only one class)
- Referring to the 'loss functions for classification' section, what loss function should be used here?
- Multi-class classification using integer (sparse) label (not one-hot encoded label) uses SparseCategoricalCrossentropy.

Building CNN using TensorFlow Keras API

- The following code summons build () method for variable time delay, and then summons compile () method.

```
>>> tf.random.set_seed(1)
>>> model.build(input_shape=(None, 28, 28, 1))
>>> model.compile(
...     optimizer=tf.keras.optimizers.Adam(),
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(),
...     metrics=['accuracy'])
```

- This model is trained by fit () method.
- If you use a method well-prepared for training and evaluation (evaluate() and predict() method), it automatically configurates the mode of the dropout layer, and aptly controls the scale of the output of hidden units.

Building CNN using TensorFlow Keras API

- Let’s train this CNN model by using this validation dataset that was made to monitor learning processes.

In [34]:
```
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy']) # same as `tf.keras.metrics.SparseCategoricalAccuracy(name='accuracy')`

history = model.fit(mnist_train, epochs=NUM_EPOCHS,
                    validation_data=mnist_valid,
                    shuffle=True)
```

```
Epoch 1/20
782/782 [==============================] - 55s 69ms/step - loss: 0.1374 - accuracy: 0.9572 - val_loss: 0.0479 - val_accura
cy: 0.9848
Epoch 2/20
782/782 [==============================] - 54s 69ms/step - loss: 0.0452 - accuracy: 0.9858 - val_loss: 0.0424 - val_accura
cy: 0.9880
Epoch 3/20
782/782 [==============================] - 53s 68ms/step - loss: 0.0297 - accuracy: 0.9906 - val_loss: 0.0408 - val_accura
cy: 0.9882
Epoch 4/20
782/782 [==============================] - 58s 73ms/step - loss: 0.0223 - accuracy: 0.9930 - val_loss: 0.0381 - val_accura
cy: 0.9888
Epoch 5/20
782/782 [==============================] - 57s 73ms/step - loss: 0.0183 - accuracy: 0.9942 - val_loss: 0.0409 - val_accura
```

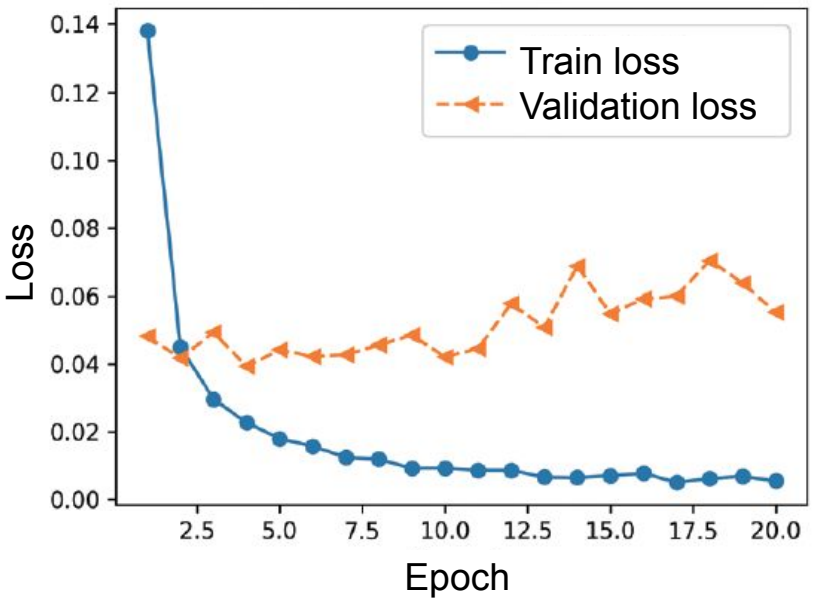
Building CNN using TensorFlow Keras API

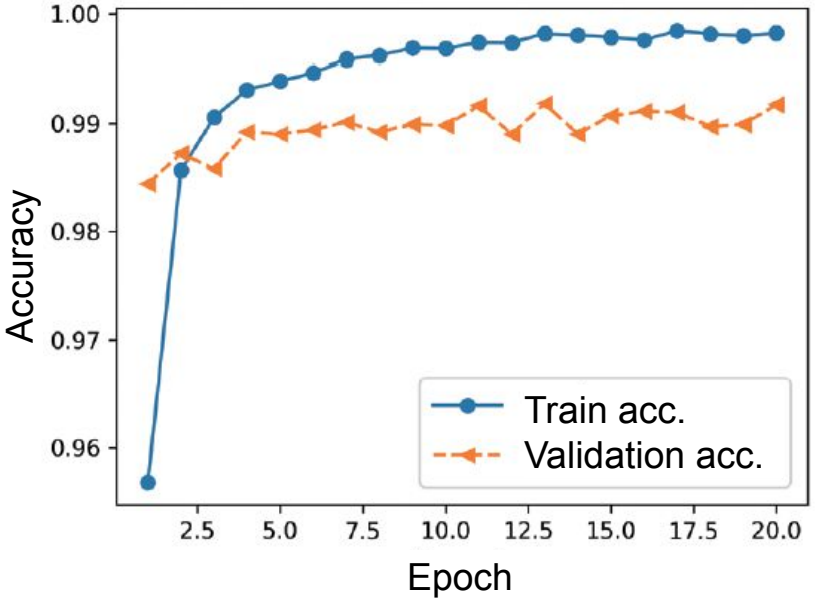
- You can draw a learning curve after the 20th training epoch.

In [35]:
```
hist = history.history
x_arr = np.arange(len(hist['loss'])) + 1

fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(1, 2, 1)
ax.plot(x_arr, hist['loss'], '-o', label='Train loss')
ax.plot(x_arr, hist['val_loss'], '--<', label='Validation loss')
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Loss', size=15)
ax.legend(fontsize=15)
ax = fig.add_subplot(1, 2, 2)
ax.plot(x_arr, hist['accuracy'], '-o', label='Train acc.')
ax.plot(x_arr, hist['val_accuracy'], '--<', label='Validation acc.')
ax.legend(fontsize=15)
ax.set_xlabel('Epoch', size=15)
ax.set_ylabel('Accuracy', size=15)

#plt.savefig('images/15_12.png', dpi=300)
plt.show()
```

Building CNN using TensorFlow Keras API

- Loss and accuracy of train and validation dataset





Building CNN using TensorFlow Keras API

- As you have seen the previous chapter, you summon .evaluate method to evaluate a trained model in a test dataset

```
In [36]: test_results = model.evaluate(mnist_test.batch(20))
         print('\n test accuracy {:.2f}%'.format(test_results[1]*100))

         500/500 [==============================] - 4s 8ms/step - loss: 0.0503 - accuracy: 0.9924

         test accuracy  99.24%
```

- This CNN model reached 99.24% accuracy

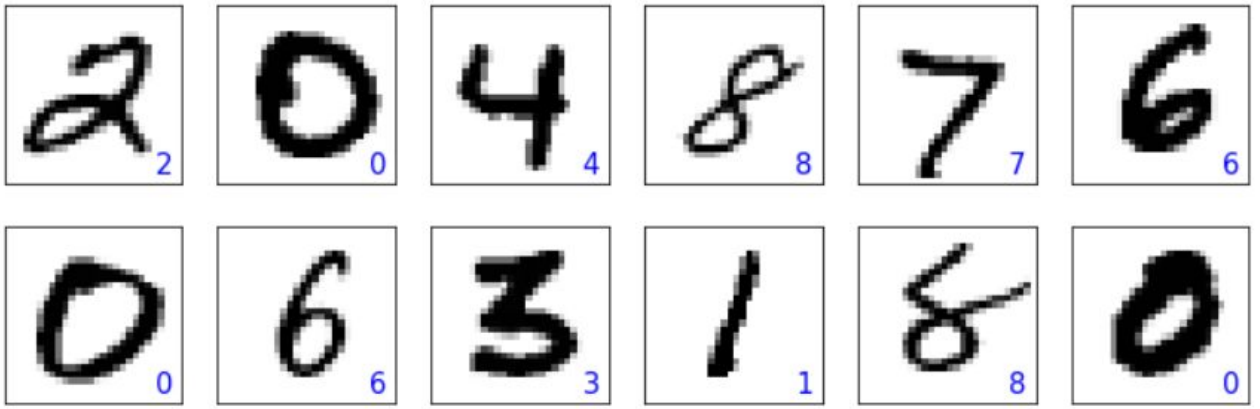
Building CNN using TensorFlow Keras API

- Lastly, after obtaining the prediction results in the form of class inclusion probability, you can modify the predicted label by finding the unit of maximum probability using tf.argmax function.
- Let's execute this on 12 batch samples and draw a graph of input and predicted labels.

In [37]:
```
batch_test = next(iter(mnist_test.batch(12)))

preds = model(batch_test[0])

tf.print(preds.shape)
preds = tf.argmax(preds, axis=1)
print(preds)

fig = plt.figure(figsize=(12, 4))
for i in range(12):
    ax = fig.add_subplot(2, 6, i+1)
    ax.set_xticks([]); ax.set_yticks([])
    img = batch_test[0][i, :, :, 0]
    ax.imshow(img, cmap='gray_r')
    ax.text(0.9, 0.1, '{}'.format(preds[i]),
            size=15, color='blue',
            horizontalalignment='center',
            verticalalignment='center',
            transform=ax.transAxes)

#plt.savefig('images/15_13.png', dpi=300)
plt.show()
```

Building CNN using TensorFlow Keras API

```
TensorShape([12, 10])
tf.Tensor([2 0 4 8 7 6 0 6 3 1 8 0], shape=(12,), dtype=int64)
```



Building CNN using TensorFlow Keras API

In [ ]:
```
import os

if not os.path.exists('models'):
    os.mkdir('models')


model.save('models/mnist-cnn.h5')
```

Unit 2.

Recurrent Neural Network (RNN) for Sequential Data Modeling

2.1. About the Recurrent Neural Network (RNN)

2.2. Long Short Term Memory Cell

Recurrent Neural Network (RNN)

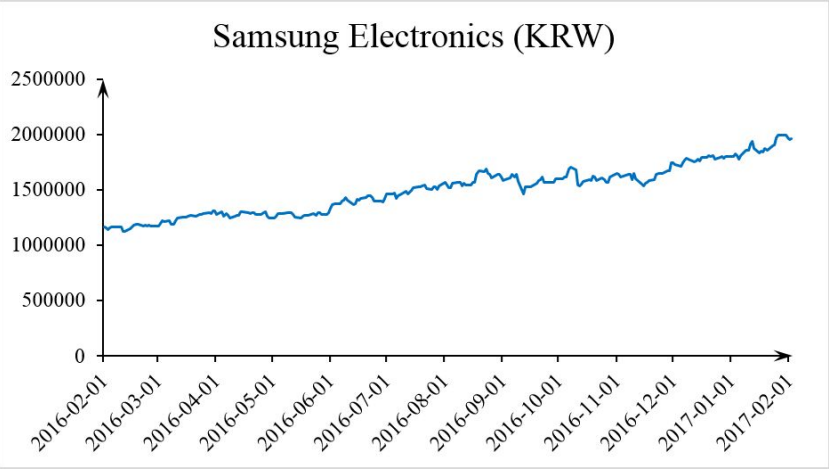
About the Recurrent Neural Network (RNN):

- RNN is a deep neural network commonly used with sequence data.
- In a sequence, there is an implicit ordering; steps are time ordered.
- We need a model that can take into account the auto-correlation.

About the Recurrent Neural Network (RNN):

- Examples of the sequence data:

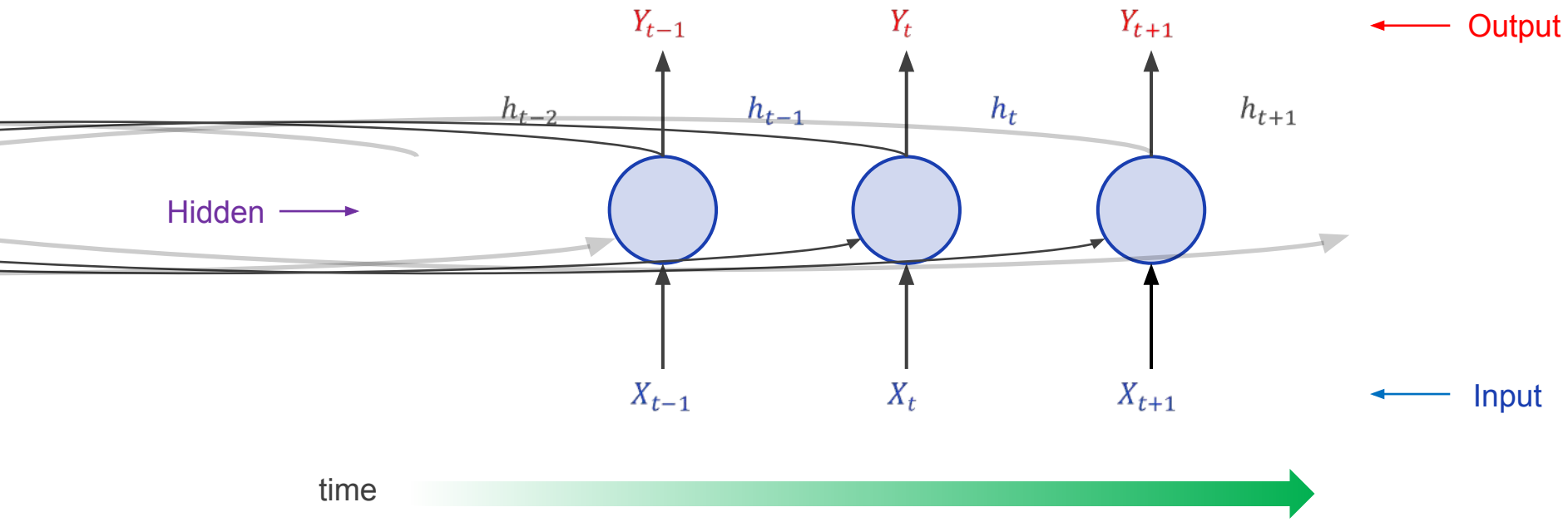
a) Time series.



b) Natural language both written and spoken.

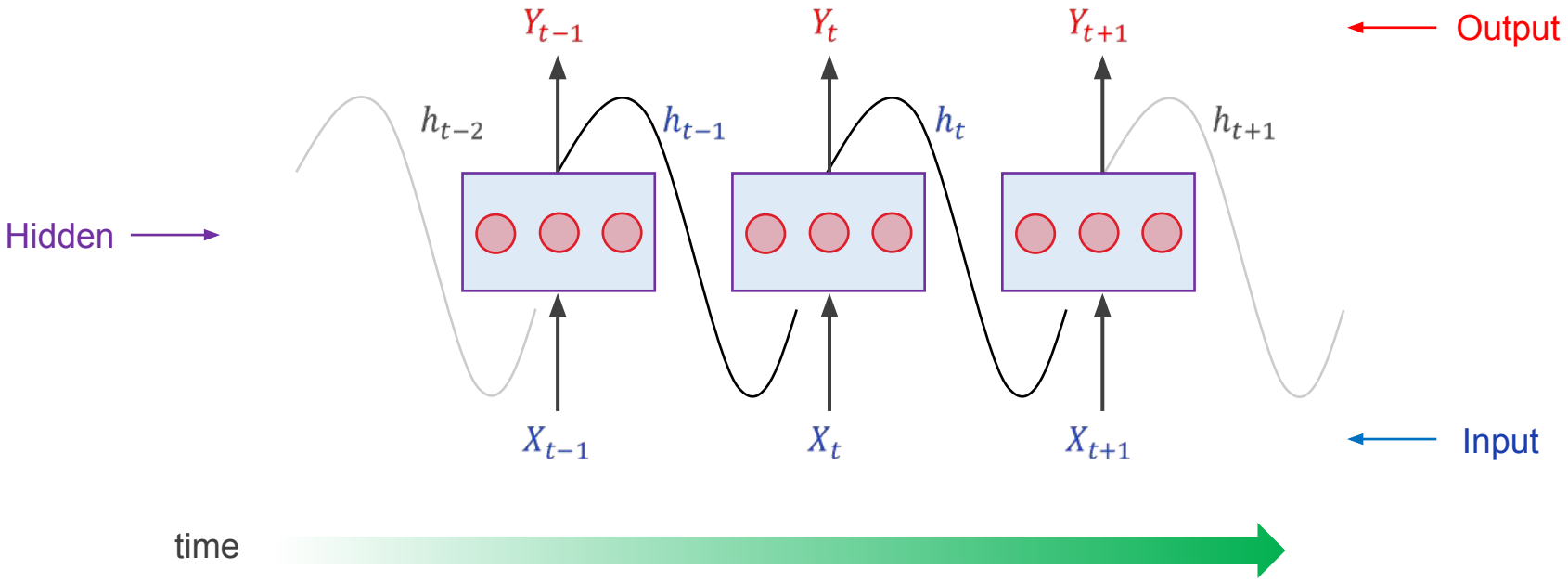
"Machine learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead."

About the Recurrent Neural Network (RNN):



At each time step, there are two inputs: one from the previous step and another from outside.

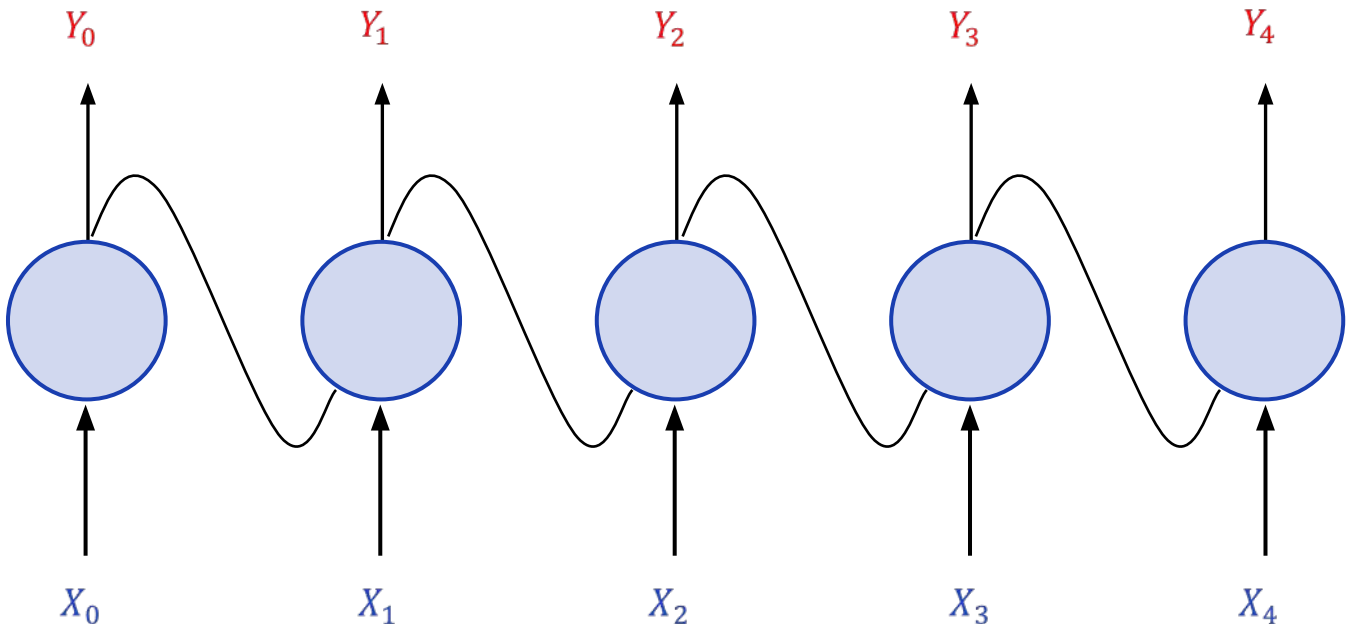
About the Recurrent Neural Network (RNN):



At each time step, we can have a layer with several nodes.

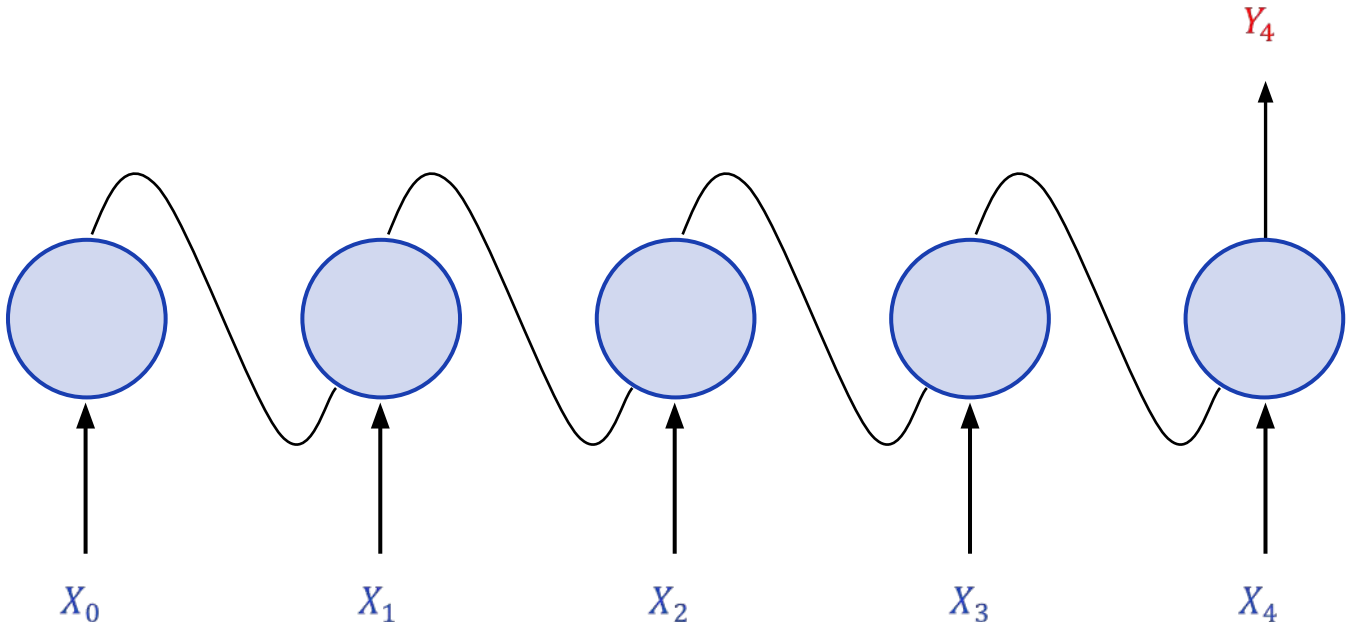
Sequence in and Sequence out:

- Can be useful for time series prediction, machine translation, etc.



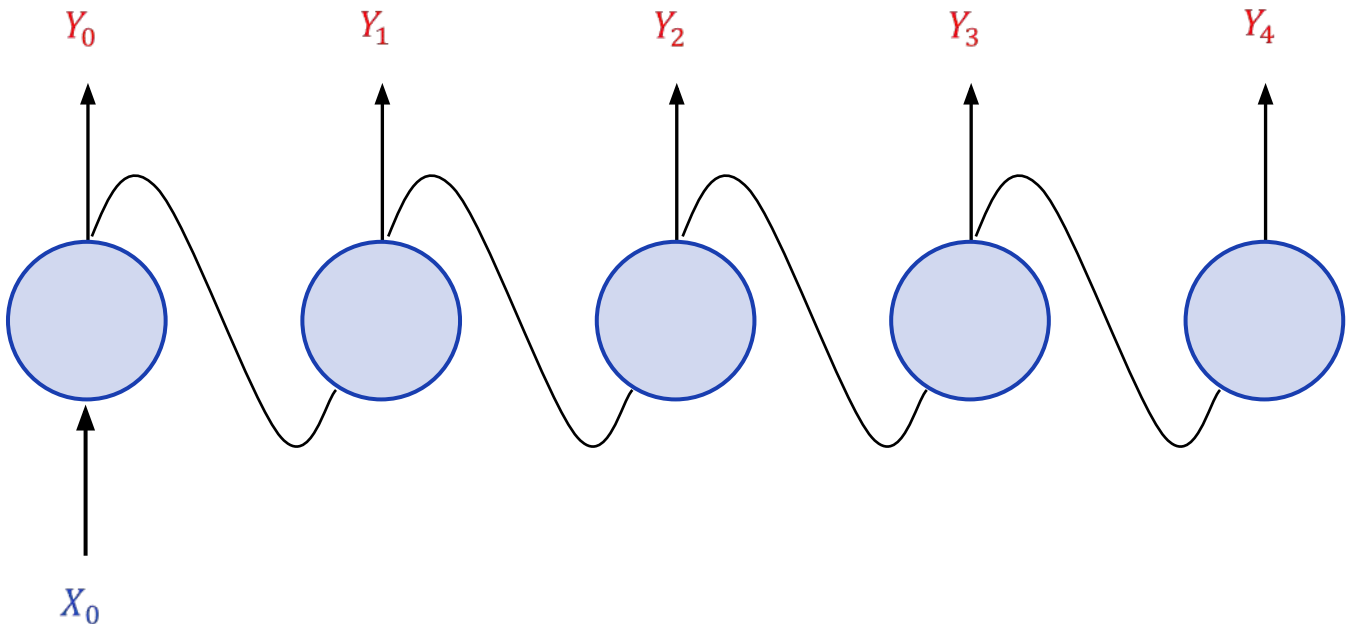
Sequence in and Vector out:

- Can be useful for classification (labeling) of text documents.



Vector in and Sequence out:

- Can be useful for generating text sequences given a vector input.



Unit 2.

Recurrent Neural Network (RNN) for Sequential Data Modeling

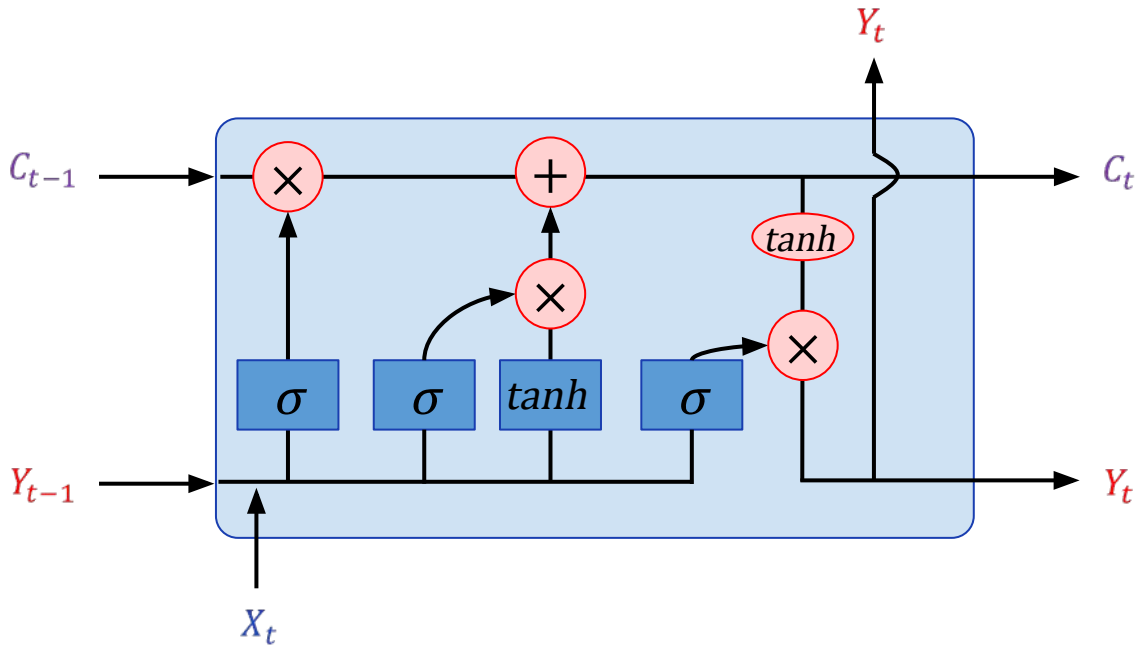
2.1. About the Recurrent Neural Network (RNN)

2.2. Long Short Term Memory Cell

Long Short Term Memory Cell

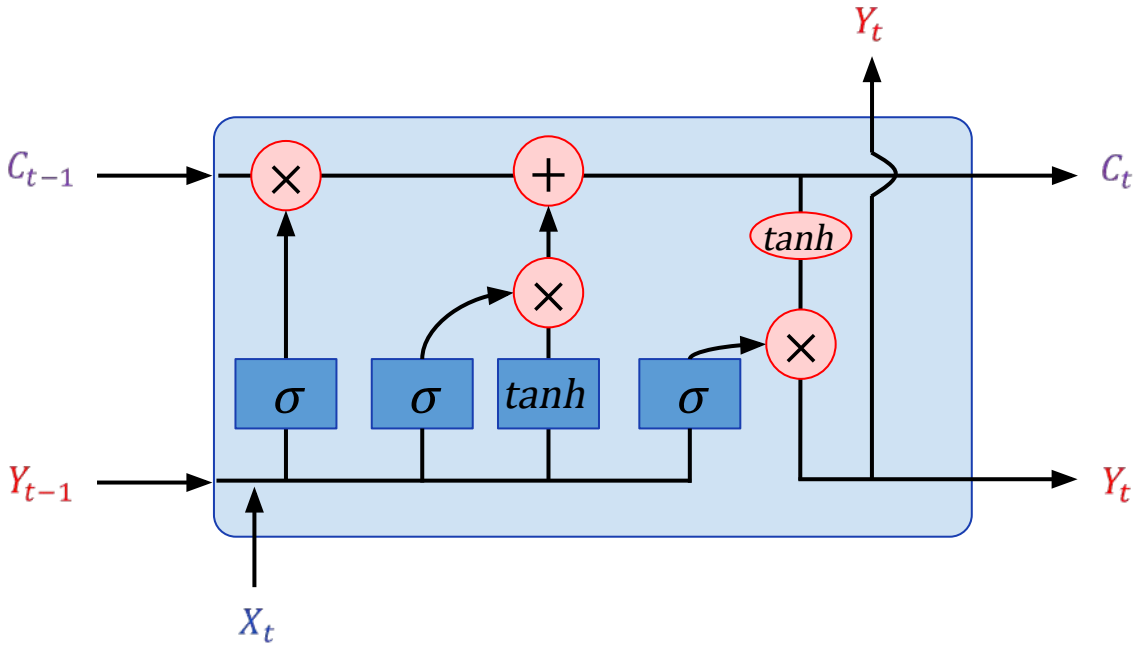
Long Short Term Memory (LSTM) cell:

- Improves upon the regular RNN cells and can hold a long standing correlation.



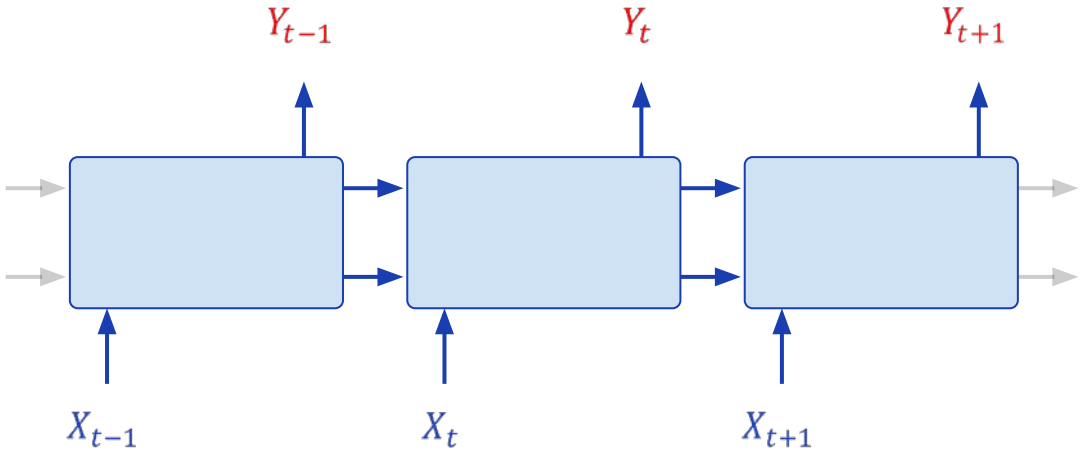
Long Short Term Memory (LSTM) cell:

- C_t = "cell state" and σ= Sigmoid function.



Long Short Term Memory (LSTM) cell:

- LSTM cells can form a sequence just like regular RNN cells.



Unit 3.

Generative Adversarial Neural Network to Create Non-Existent Images

3.1. AutoEncoder

3.2. About the Generative Adversarial Networks (GAN)

3.3. Generative Adversarial Networks Exercise

AutoEncoder

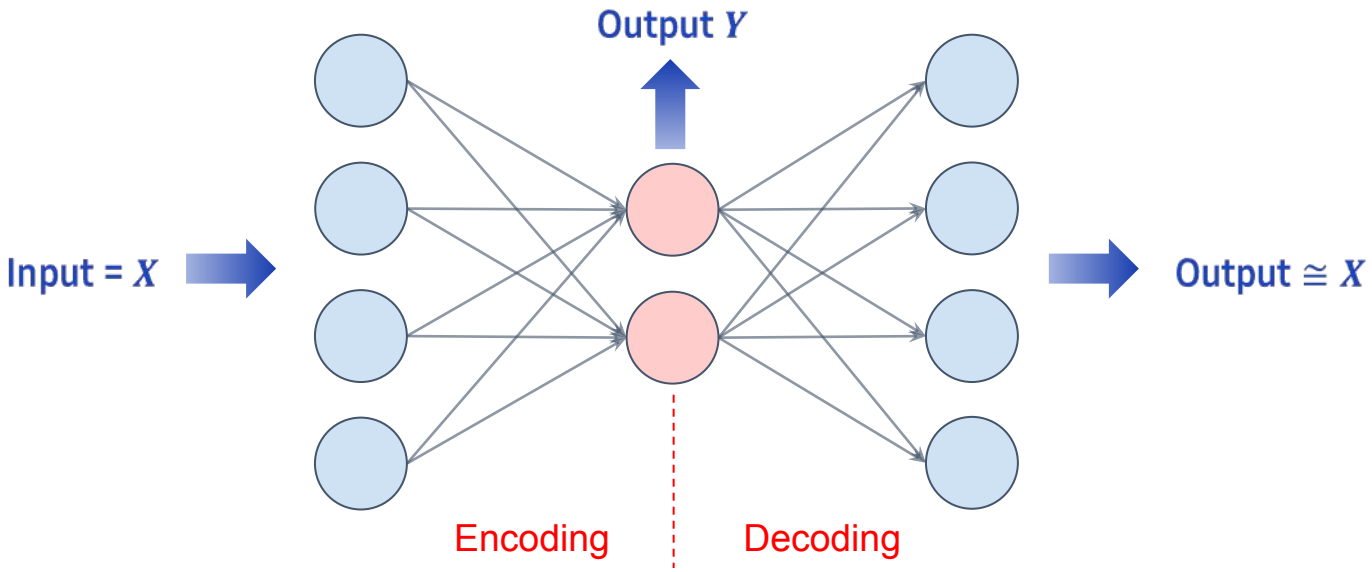
About the AutoEncoder:

- Before studying the mechanisms of GAN, you have to learn the autoencoder which compresses and decompresses train data.
- A basic autoencoder cannot generate new data, but its underlying mechanism helps understanding GAN.
- The main objective of GAN is to synthesize a new data that has the same distribution as the training dataset.
- Since the original form of GAN does not need label data, it is considered as an unsupervised learning type.
- The extension of the original GAN can be seen both as unsupervised and supervised learning method.

AutoEncoder

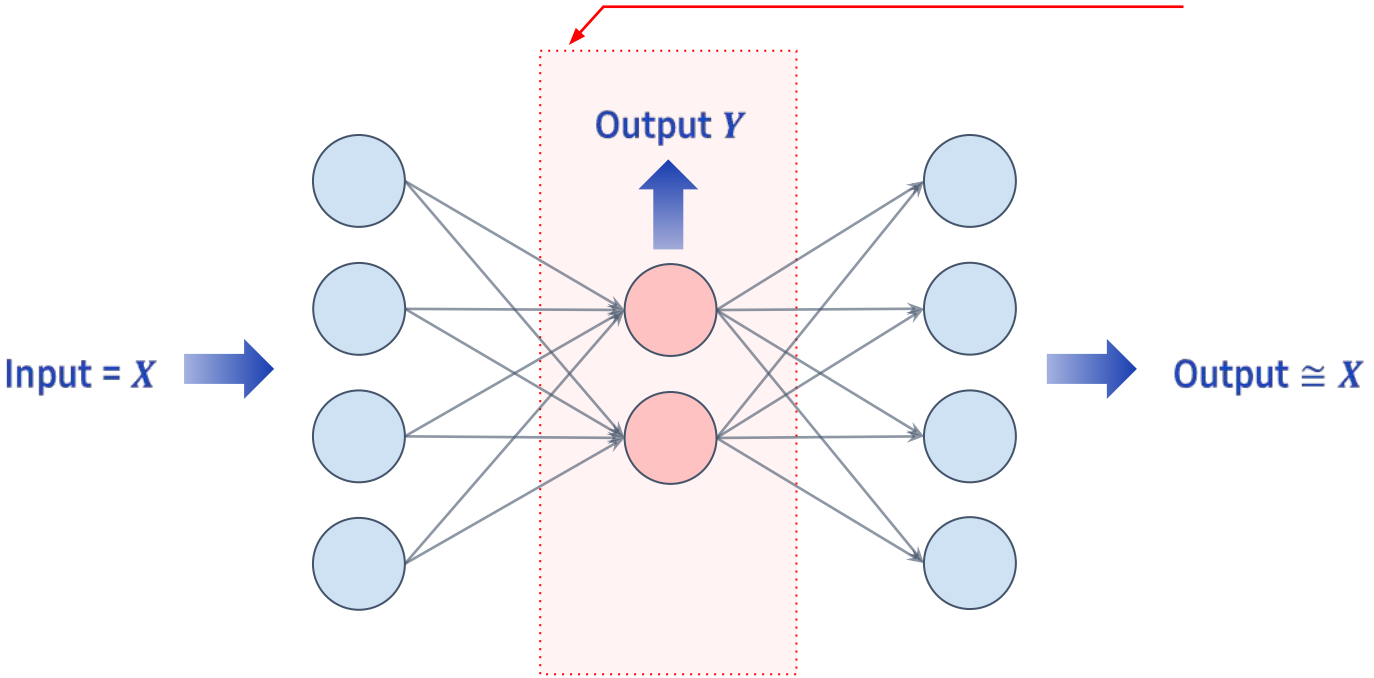
About the AutoEncoder:

- Can be used to “compress” the data by dimensional reduction.
- Composed of symmetric encoding and decoding parts.



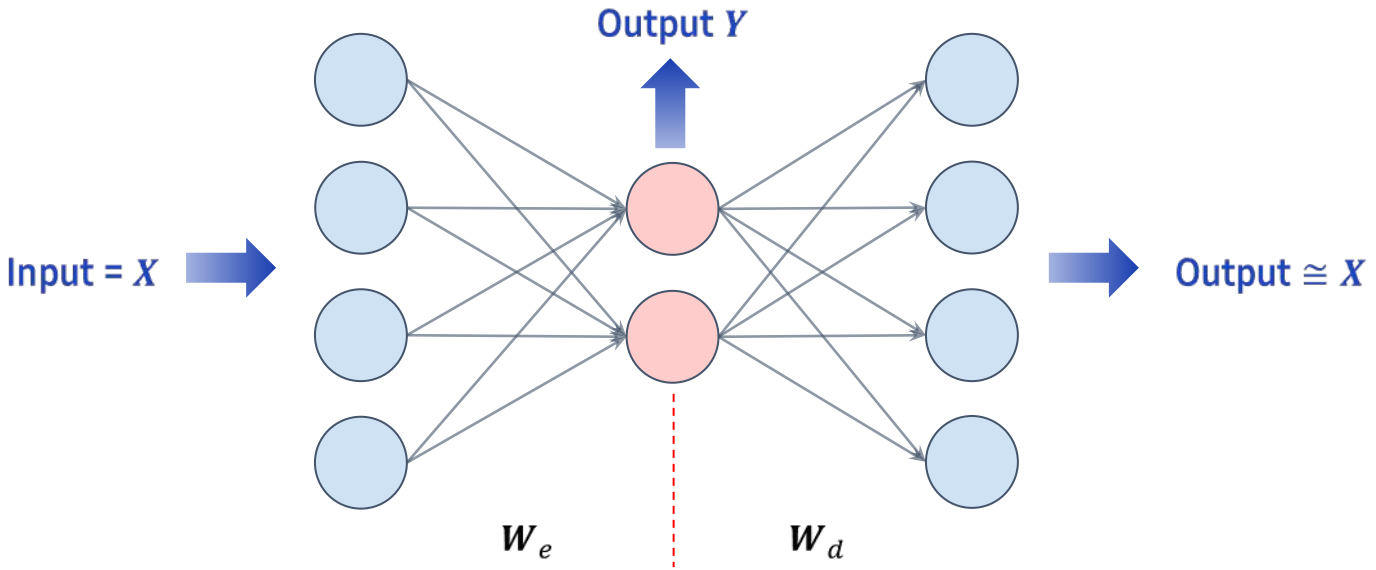
About the AutoEncoder:

- Number of input nodes = Number of output nodes > Number of nodes in the hidden layer.



About the AutoEncoder:

- The encoding weight matrix $\boldsymbol{W}_e$ and the decoding weight matrix $\boldsymbol{W}_d$ can be related: $\boldsymbol{W}_d = \boldsymbol{W}_e^t$.



Let's closely examine the structure of the autoencoder one more time.

- An autoencoder is built by connecting two neural networks, encoder neural network and decoder neural network.

Encoder neural network

- Encoder neural network encodes p dimensional vector z (that is, $z \in R^p$), which is received from d dimensional input feature vector related to sample x (that is, $x \in R^d$). In other words, the encoder's role is to learn how to model $z = f(x)$ function.
- The encoded vector z is called a latent vector or latent feature expression.
- Generally the dimension of a latent vector is smaller than that of an input sample. That is, $p < d$.
- It can be said that the encoder compresses data.

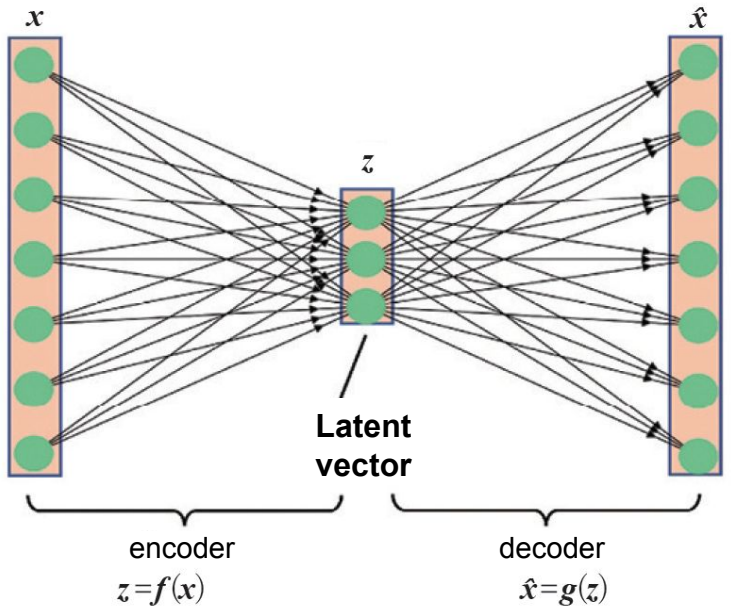
Decoder neural network

- The decoder decompresses $\hat{x}$ from low-dimensional latent vector z.
- Decoder can be considered as function $\hat{x} = g(z)$.

3.1. AutoEncoder

Let’s closely examine the structure of the autoencoder one more time.

- In this figure, encoder and decoder is composed of its own fully connected layer.
- An autoencoder without a hidden layer (like multi-layer neural networks), but it can be built into a deep autoencoder by inserting many non-linear hidden layers.
- Then, it can learn more effective data compression and reconstructing functions.
- In this section, autoencoders use fully connected layers.
- However, when using images, you can use convolutional layers instead of fully connected layers.



Coding Exercise #0703a

Follow practice steps on ‘ex_0703a.ipynb' file

Coding Exercise #0703b

Follow practice steps on ‘ex_0703b.ipynb' file

Unit 3.

Generative Adversarial Neural Network to Create Non-Existent Images

3.1. AutoEncoder

3.2. About the Generative Adversarial Networks (GAN)

3.3. Generative Adversarial Networks Exercise

Generative Adversarial Networks (GAN)

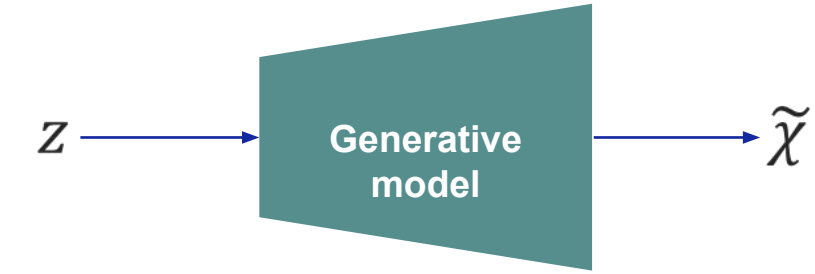
About the Generative Adversarial Networks (GAN):

- Can you distinguish the real pictures from the fake ones?

https://arxiv.org/abs/1812.04948

Generative Model for new synthetic data

- Autoencoder is the definite model
- That is, once the autoencoder is trained, you can reconstruct input x from its low-dimensional compressed version.
- Although the autoencoder can reconstruct an input by transforming compressed expression, it cannot generate new data
- However, a generative model (a latent expression) can generate a new sample $\hat{x}$ from random vector z.
- A generative model expressed in a diagram is as follows.
- Random vector z is easy to sample because it is created from a simple distribution of which features are perfectly known.
- For example, you can sample each unit of z from an even distribution of range [-1, 1] (written as zi~Uniform(-1,1)) or standard normal distribution (written as zi~Normal(μ = 0, σ2= 1)



Generative Model for new synthetic data

- Observing the generative model, you can notice that the decoder of the autoencoder resembles a generative model.
- Both receive latent vector z as input, and create an output in the space of x ($\hat{x}$ is a reconstruction of input x in the autoencoder, while it is a synthetic sample in a generative model)
- The big difference between the two models is that the distribution of z in the autoencoder is unknown, but it is perfectly known in the generative model.
- Autoencoder can be generalized to become a generative model.
- VAE is one of the methods to do so.

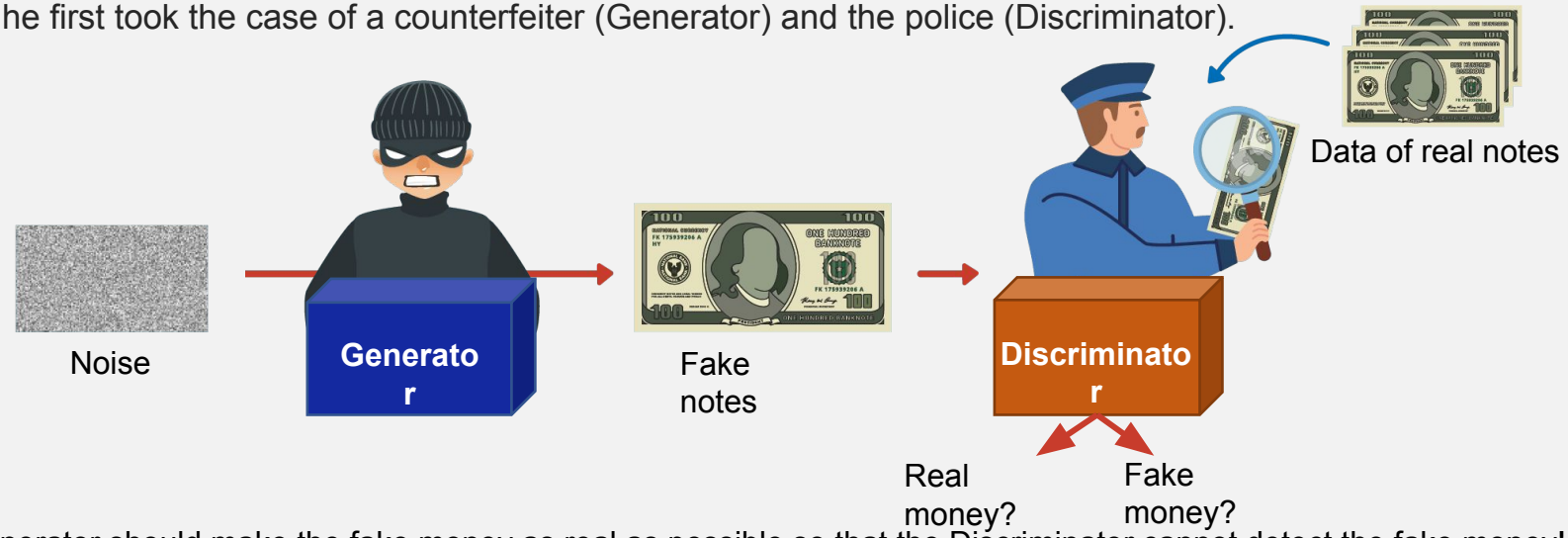
Generative Model for new synthetic data

- In VAE, once it receives input sample x, the encoder network computes mean μ and variance σ2 of the latent vector distribution.
- While training VAE, it modifies the neural network so that the mean and variance (zero-mean, unit-variance) adjusts to standard normal distribution.
- After training the VAE model, separate the encoder, and use the decoder neural network to generate a new sample $x\hat{}$ by inserting z vector which was randomly sampled form a Gaussian distribution.
- There are other generative models such as autoregressive model and normalizing flow model.
- Recently, GAN is the most popular generative model architecture in deep learning.

Generating new sample with GAN

- Let's assume a generative neural network that generates output image x from an input random vector z sampled from a known distribution.
- This is called generator (G) and $\hat{x}$ =G(z) is a generated output.
- Then, suppose the objective of the model is to generate images of faces, buildings, animals, handwritten numbers like MNIST etc.

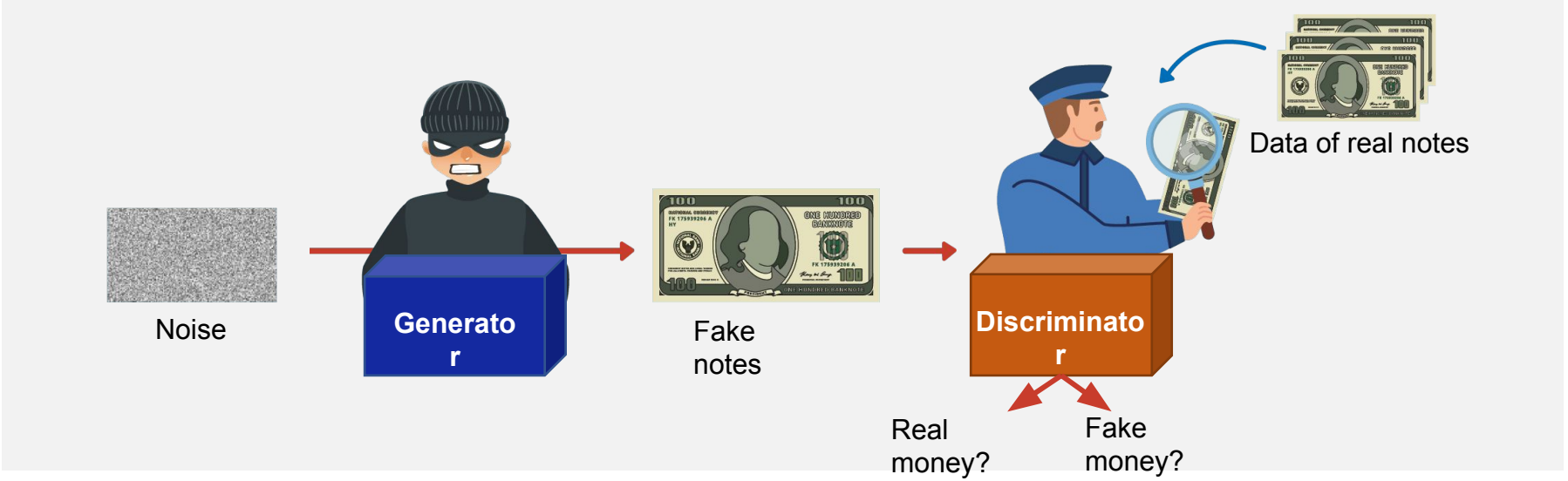
Generative Adversarial Networks (GAN) was introduced by Ian Goodfellow in 2014. In order to easily explain GAN, he first took the case of a counterfeiter (Generator) and the police (Discriminator).



Generator should make the fake money as real as possible so that the Discriminator cannot detect the fake money!

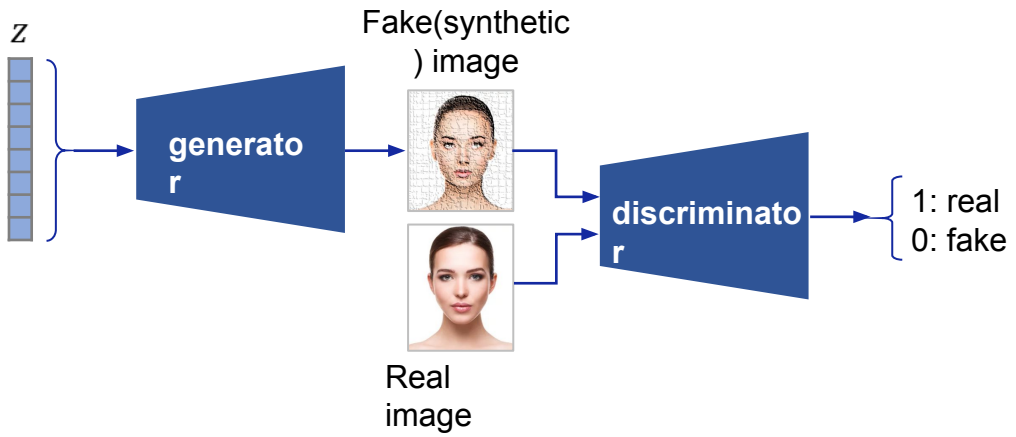
Generating new sample with GAN

- As always, initialize the neural network with a random weight value.
- The output image is like a white noise when the weight is not yet learned.
- Let's suppose a function that evaluates the quality of an image. (called evaluation function)
- To improve the quality of the generated image, this function can send feedbacks to the neural network to modify the weight value.
- Likewise, you can train the generator based on feedbacks from the evaluation function.
- The generator will learn to improve the output to resemble the real image.



Generating new sample with GAN

- The evaluation function mentioned in the paragraph before simplifies image generating process. The real question is ‘Is there an all-purpose function to evaluate the quality of an image, and if so, how can it be defined?’
- A human-being can evaluate an image from the neural network output.
- A result from the brain cannot be backpropagated to the neural network (yet).
- If a human brain can evaluate the quality of a synthetic image, can we create a neural network model that performs a similar task?
- This is the central idea behind GAN.
- As you can see next, GAN model is composed of another neural network called discriminator (D).
- Discriminator is a classifier that learns how to distinguish synthetic image $x\hat{}$ from a real image x.



Generating new sample with GAN

- In GAN, the generator and discriminator are trained together.
- First, after initializing the weight of the model, the generator generates an image unlike the real image.
- Likewise, the discriminator has poor ability to distinguish synthetic fake images from real images.
- As time passes, (that is, through training) the two neural networks improve with mutual interaction.
- The two neural networks execute an adversarial game.
- The generator learns how to improve its output to deceive the discriminator
- The discriminator is trained to better distinguish synthetic images.

Generating new sample with GAN

- The objective function of GAN in the original thesis is as follows.

$$V(\theta^{(D)}, \theta^{(G)}) = E_{x\sim p_{data}(x)}[\log(D(x))] + E_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

  - V(θ(D), θ(G)) is called a value function
  - This can be considered as a compensation
  - You need to maximize this value for the discriminator (D) and minimize this value for the generator (G)

    ➡ $\min_G \max_D V(\theta^{(D)}, \theta^{(G)})$

  - D(x) is a probability that shows whether the input sample x is real or fake.
  - $E_{x\sim p_{data}(x)}[\log D(x)]$ shows expected value of the expression in the bracket in regards to the sample from the data distribution (distribution from the real sample
  - $E_{z\sim p_z(z)}[\log(1-D(G(z)))]$ shows expected value of the expression in the bracket in regards to the input vector z

Understanding generator and discriminator loss function in GAN

- Training GAN models with such evaluation function needs two steps of optimization.
  - (1) Maximize compensation for the discriminator
  - (2) minimize compensation for the generator

- A practical GAN training method alternates these two steps
  - (1) Fixate the weight value of one neural network and optimize the weight value of the other.
  - (2) Fixate the second neural network and optimize the fist.

- Iterate this in every training repetition.
- Let's suppose you are fixating the generator neural network and optimizing the discriminator.
- Both terms of the value function V(θ(D), θ(G)) contributes to the optimization of the discriminator.
- The first term is loss related to the real sample
- The second term is loss related to the fake sample
- If you fix G, the objective function serves to maximize V(θ(D) θ(G))
- That is, make the discriminator better distinguish between real and fake images

Understanding generator and discriminator loss function in GAN

- After optimizing the discriminator by using the loss from the real and fake samples, fixate the discriminator and optimize the generator.
- In this case, only the second term of V(θ(D), θ(G)) contributes to the generator's gradient.
- When D is fixed, the objective function that minimizes V(θ(D), θ(G)) can be written as $\min_G E_{z\sim p_z(z)}\left[\log\left(1-D(G(z))\right)\right]$
- According to Goodfellow's GAN thesis, log(1-D(G(z))) function presents a gradient loss issue in the initial training process.
- Therefore, in the initial phase, output G(z) does not look like a real sample.
- It is certain that D(G(z)) will approach near 0.
- Such phenomenon is called saturation.

To solve such a problem, you can write the objective function intended for minimization $\min_G E_{z\sim p_z(z)}[\log(1-D(G(z)))]$ as $\max_G E_{z\sim p_z(z)}[\log(D(G(z)))]$

- Modifying the expression as such means swapping the images of real and fake samples during generator training and finding the minimal value of the general function. In other words, changing the label to 1 even though it is 0 because the synthetic sample is fake.

Then, instead of maximizing $\max_G E_{z\sim p_z(z)}[\log(D(G(z)))]$
you can minimize binary cross entropy loss with the new label.

Understanding generator and discriminator loss function in GAN

- Introduce general optimization process for GAN training
- Let's see data labels usable for GAN training
- You can use binary cross entropy loss function if the discriminator is a binary classifier (that is, the class label for fake and real images are 0 and 1)
- An answer label for discriminator loss can be discerned as follows.

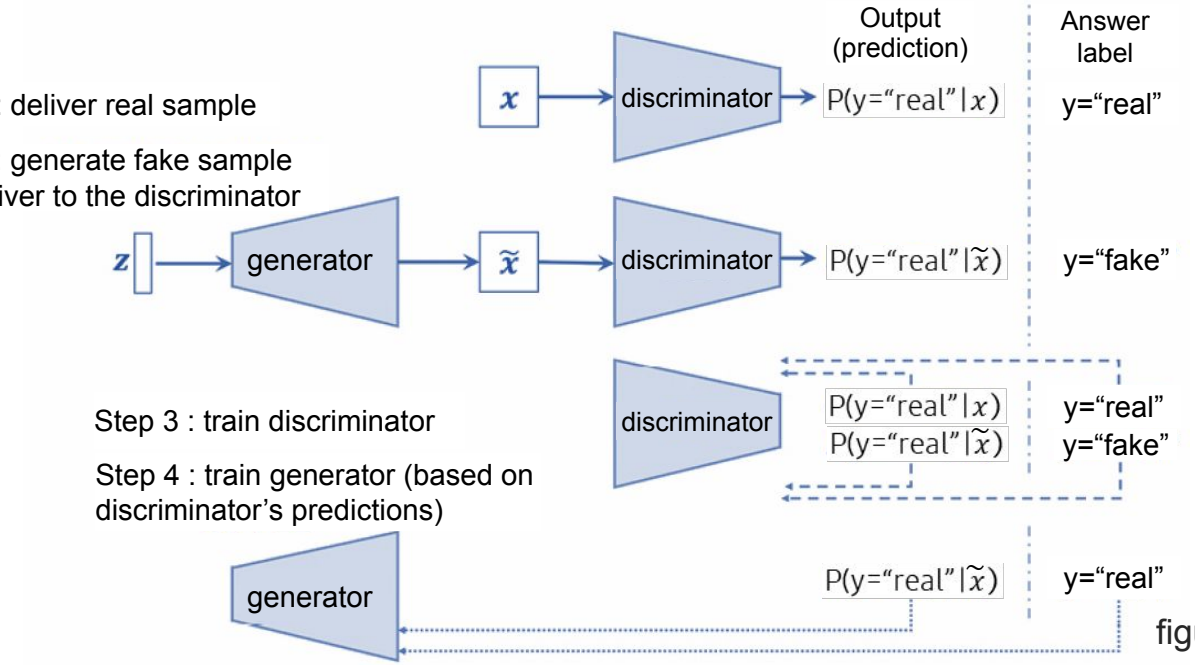


figure GAN training process

Understanding generator and discriminator loss function in GAN

- What is the label for training the generator?
- If the discriminator does not classify the generator's output as real, a penalty can be imposed on the generator since its duty is to generate real-like images.
- When computing the loss function of the generator, you assume that the label for the generator's output is 1.

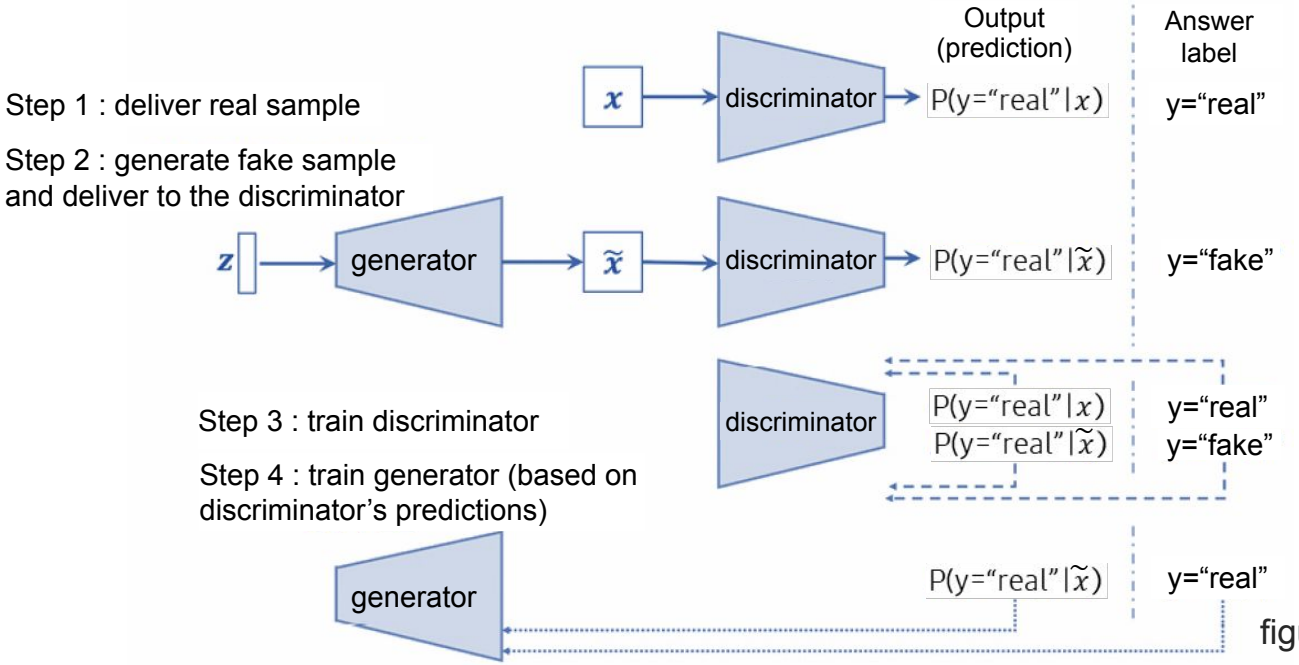


figure GAN training process

Unit 3.

Generative Adversarial Neural Network to Create Non-Existent Images

3.1. AutoEncoder

3.2. About the Generative Adversarial Networks (GAN)

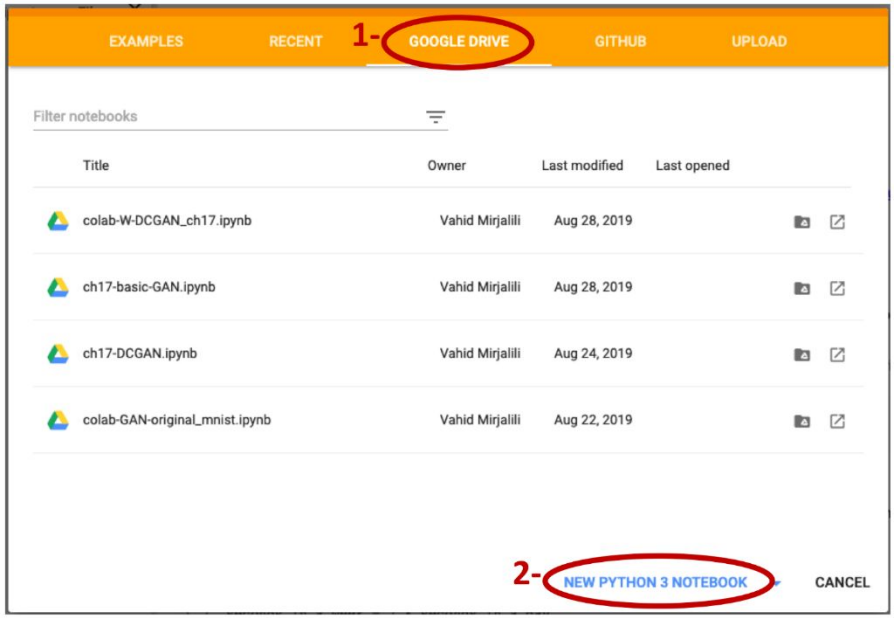
3.3. Generative Adversarial Networks Exercise

Convolutional Neural Network (CNN)

Training GAN model in Google Colab

- This chapter's code exercise requires resources higher than a laptop without GPU or computing power of a workstation.
- If you have a computer with NVIDA GPU, and have installed cuDNN library, you can boost computation speed.
- Here we will use Google Colab, a free cloud computing service, because such high-performance computing resources are generally not available.
- Google Colab offers jupyer notebook instance on cloud.
- This notebook can be saved in Google Drive of GitHub.
- Colab offers many computing resources such as CPU, GPU and even TPU, but limits its operation time to 12 hours.
- The notebook is shutdown after 12 hours of operation.
- Code exercise in this chapter only requires max 2 – 3 hours.
- For projects that run for more than 12 hours, it is recommended that you save mid-results as checkpoint files.

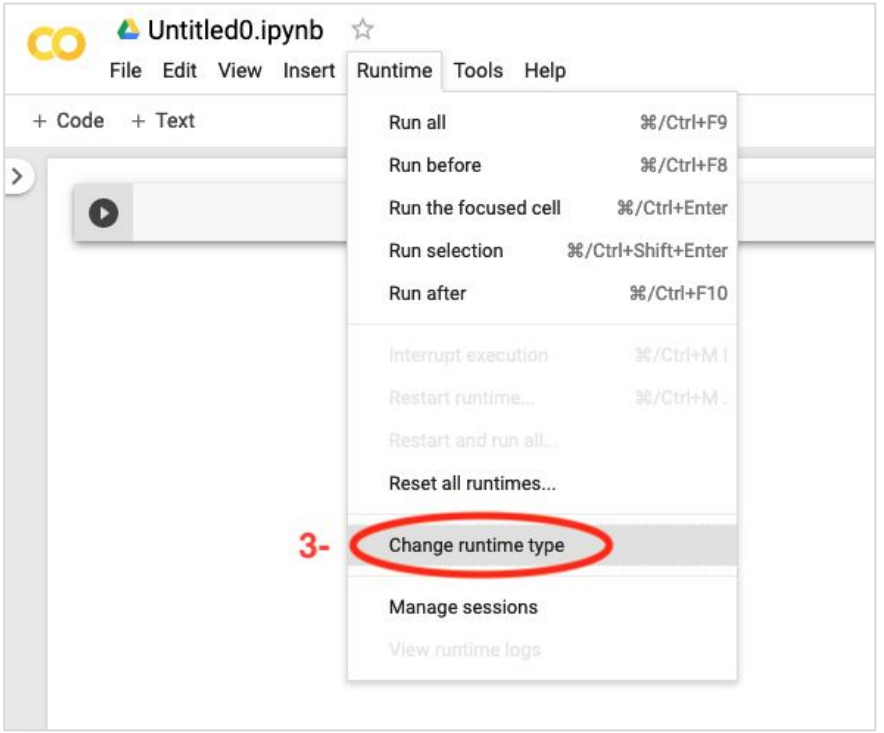
Training GAN model in Google Colab



- Google Colab is simple to use.
- If you visit https://colab.research.google.com, it will ask you where to save jupyter notebook.
- Click Google Drive tab in this window.
- By doing this, you will be able to save notebook in Google Drive.
- Then, click new notebook link at the end to make a new notebook.

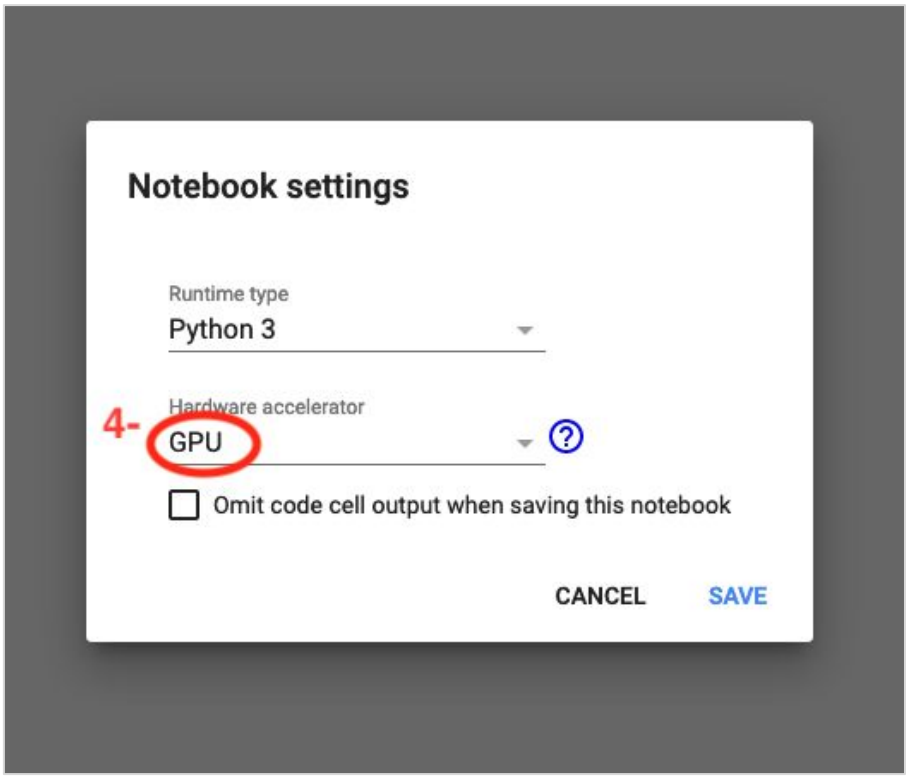
Building GAN model from the ground

- As the next step, let's see how to run the notebook's code using GPU



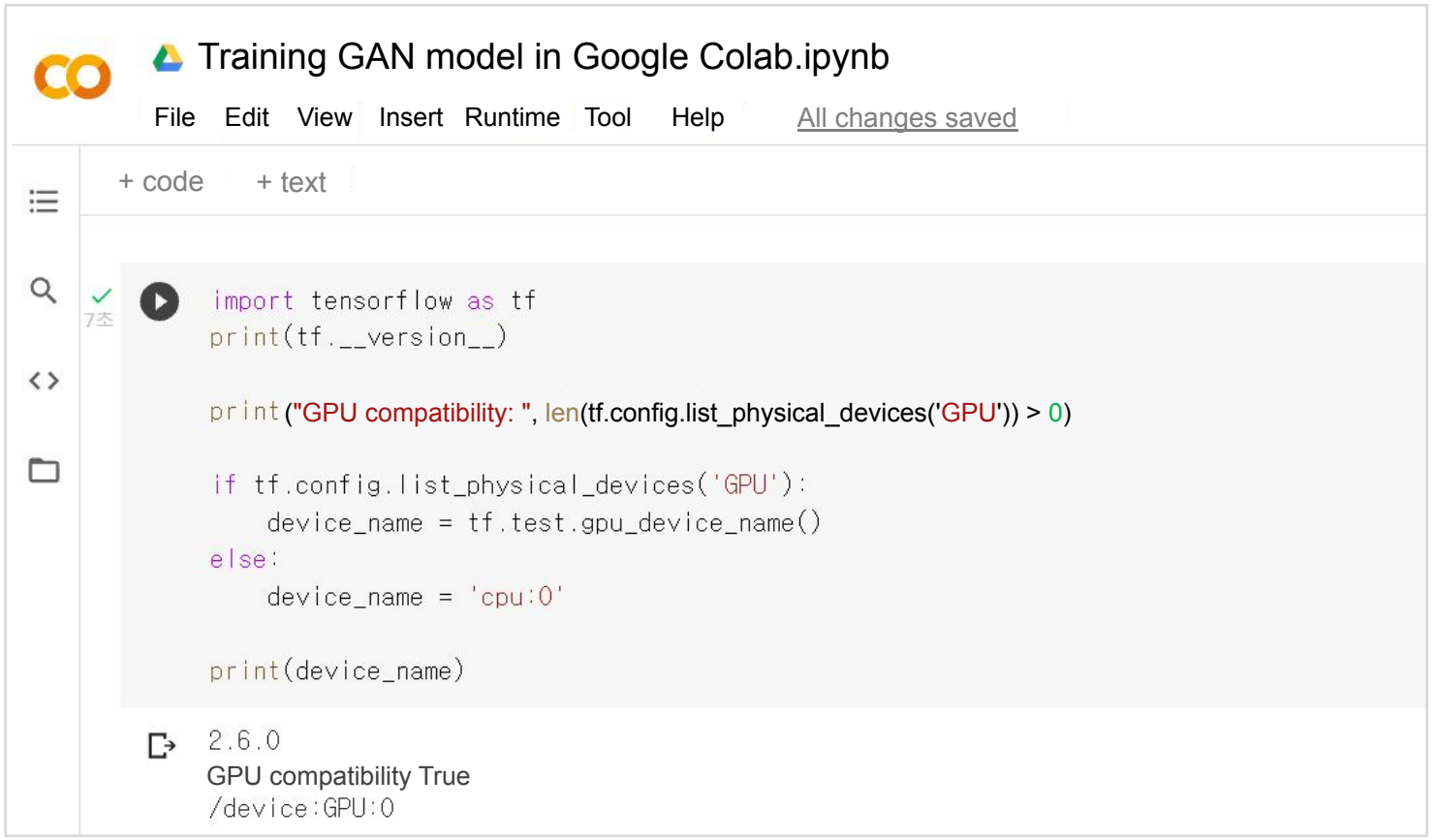
Building GAN model from the ground

- Choose Runtime in the menu bar, and click Runtime type change and select GPU as in the figure



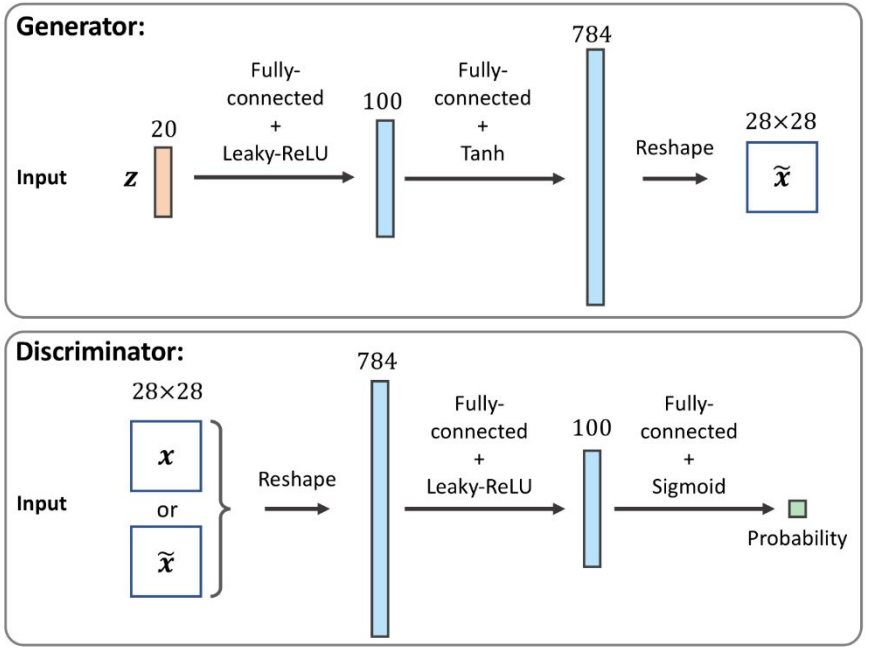
Training GAN model in Google Colab

- Let’s check GPU compatiblity for the installed TensorFlow version using the following code.



Building generator and discriminator neural networks

- Let's build first GAN model's generator and discriminator as a fully connected neural network with one or more hidden layers. (refer to the figure below)
- This model is the original GAN version, and is called vanilla GAN.



- This model uses LeakyReLu activation function for each hidden layer.
- ReLu produces sparse gradient. However, this is not suitable for cases when gradients are needed in all range of the input values.
- In the discriminator neural network, dropout layer follows the hidden layer.
- Generator's output layer uses hyperbolic tangent (tanh) activation function (Using tanh activation function in the generator neural network helps the learning process)
- The discriminator's output layer does not have an activation function in order to compute the logit (that is, it uses linear activation function)
- Or it could use sigmoid activation function to return the probability as an output.

Let's define two helper functions for the two neural networks

- This function builds a model with Keras Sequential class, and the code with the forementioned layer is as follows.

```
[5]  import tensorflow as tf
     import tensorflow_datasets as tfds
     import numpy as np
     import matplotlib.pyplot as plt
```

```
## Define the generator function:
def make_generator_network(
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=784):
    model = tf.keras.Sequential()
    for i in range(num_hidden_layers):
        model.add(
            tf.keras.layers.Dense(
                units=num_hidden_units,
                use_bias=False)
            )
        model.add(tf.keras.layers.LeakyReLU())

    model.add(tf.keras.layers.Dense(
        units=num_output_units, activation='tanh'))
    return model
```

```
## Define the discriminator function:
def make_discriminator_network(
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=1):
    model = tf.keras.Sequential()
    for i in range(num_hidden_layers):
        model.add(tf.keras.layers.Dense(units=num_hidden_units))
        model.add(tf.keras.layers.LeakyReLU())
        model.add(tf.keras.layers.Dropout(rate=0.5))

    model.add(
        tf.keras.layers.Dense(
            units=num_output_units,
            activation=None)
        )
    return model
```

Building generator and discriminator neural network

```
image_size = (28, 28)
z_size = 20
mode_z = 'uniform'  # 'uniform' vs. 'normal'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)

gen_model = make_generator_network(
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))

gen_model.build(input_shape=(None, z_size))
gen_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               2000
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 784)               79184
=================================================================
Total params: 81,184
Trainable params: 81,184
Non-trainable params: 0
_________________________________________________________________
```

- It configurates to train the next model.
- MNIST image size is 28 x 28 pixels. (MNIST file has only one color channel since it is a black and white image)
- Then, set the size of input vector z as 20, and randomly initialize the weight value from even distribution.
- We'll use fully connected layers and a single hidden layer with 100 units in each neural network to build a very simple GAN to serve as an example.
- Generate and initialize two neural networks in the following code
- Load summary () method from Keras

Building generator and discriminator neural network

```
disc_model = make_discriminator_network(
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)

disc_model.build(input_shape=(None, np.prod(image_size)))
disc_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 100)               78500
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dropout (Dropout)            (None, 100)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 101
=================================================================
Total params: 78,601
Trainable params: 78,601
Non-trainable params: 0
_________________________________________________________________
```

Define test dataset

- As the next step, load MNIST dataset and apply necessary preprocessing steps.
- The range of pixel values of the synthetic image is (-1, 1) since the output layer of the generator uses tanh activation function.
- The range of input MNIST image pixels is [0,255] (tf.unit8 data type from TensorFlow)
- In the preprocessing stage, change the dytpe of the input image tensor from tf.uint8 to tf.float32 by using tf.image.convert_image_dtype function.

```
mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
mnist = mnist_bldr.as_dataset(shuffle_files=False)

def preprocess(ex, mode='uniform'):
    image = ex['image']
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.reshape(image, [-1])
    image = image*2 - 1.0
    if mode == 'uniform':
        input_z = tf.random.uniform(
            shape=(z_size,), minval=-1.0, maxval=1.0)
    elif mode == 'normal':
        input_z = tf.random.normal(shape=(z_size,))
    return input_z, image
```

```
Downloading and preparing dataset mnist/3.0.1 (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root/tensorflow_datasets/mnist/3.0.1...
WARNING:absl:Dataset mnist is hosted on GCS. It will automatically be downloaded to your
local data directory. If you'd instead prefer to read directly from our public
GCS bucket (recommended if you're running on GCP), you can instead pass
`try_gcs=True` to `tfds.load` or set `data_dir=gs://tfds-data/datasets`.

Dl Completed...: 100% 4/4 [00:00<00:00, 9.93 file/s]

Dataset mnist downloaded and prepared to /root/tensorflow_datasets/mnist/3.0.1. Subsequent calls will reuse this data.
```

Define test dataset

- Loading this function also changes pixel intensity range to [0, 1]
- Multiply 2 and subtract 1 to adjust pixel intensity range to [-1, 1]
- Also create a random vector z based on a random distribution (standard or normal distribution widely used in this code)
- Here, collectively return input vector z and the extracted image from train dataset for convenience

```
mnist_bldr = tfds.builder('mnist')
mnist_bldr.download_and_prepare()
mnist = mnist_bldr.as_dataset(shuffle_files=False)

def preprocess(ex, mode='uniform'):
    image = ex['image']
    image = tf.image.convert_image_dtype(image, tf.float32)
    image = tf.reshape(image, [-1])
    image = image*2 - 1.0
    if mode == 'uniform':
        input_z = tf.random.uniform(
            shape=(z_size,), minval=-1.0, maxval=1.0)
    elif mode == 'normal':
        input_z = tf.random.normal(shape=(z_size,))
    return input_z, image
```

```
Downloading and preparing dataset mnist/3.0.1 (download: 11.06 MiB, generated: 21.00 MiB, total: 32.06 MiB) to /root/tensorflow_datasets/mnist/3.0.1...
WARNING:absl:Dataset mnist is hosted on GCS. It will automatically be downloaded to your
local data directory. If you'd instead prefer to read directly from our public
GCS bucket (recommended if you're running on GCP), you can instead pass
`try_gcs=True` to `tfds.load` or set `data_dir=gs://tfds-data/datasets`.

Dl Completed...: 100% 4/4 [00:00<00:00, 9.93 file/s]

Dataset mnist downloaded and prepared to /root/tensorflow_datasets/mnist/3.0.1. Subsequent calls will reuse this data.
```

Let's examine the dataset object from previous

```
[9]  mnist_trainset = mnist['train']

     print('before preproceessing: ')
     example = next(iter(mnist_trainset))['image']
     print('dtype: ', example.dtype, ('minimum: {} maximum: {}'.format(np.min(example), np.max(example)))

     mnist_trainset = mnist_trainset.map(preprocess)

     print('after preproceessing: ')
     example = next(iter(mnist_trainset))[0]
     print('dtype: ', example.dtype, 'minimum: {} maximum: {}'.format(np.min(example), np.max(example)))
```

```
after preproceessing:
dtype:  <dtype: 'uint8'> minimum: 0 maximum: 255
before preproceessing:
dtype:  <dtype: 'float32'>  minimum: -0.8737728595733643 maximum: 0.9460210800170898
```

In the following code, let's print the input vector and shape of the image array by extracting a batch

```
mnist_trainset = mnist_trainset.batch(32, drop_remainder=True)
input_z, input_real = next(iter(mnist_trainset))
print('input-z -- shape :', input_z.shape)
print('input-real -- shape :', input_real.shape)
```

```
input-z -- shape : (32, 20)
input-real -- shape : (32, 784)
```

In addition, let's run front-propagation computation of the generator and discriminator to understand the general data flow.

- First, return output g_output by inserting the batch of input vector z onto the generator
- This is the batch of the fake sample
- Obtain d_logits_fake, a batch logit of the fake sample by inserting the above batch onto the discriminator model.
- Then, obtain d_logits_real, logit for the real image, by inserting the preprocessed image from the dataset object onto the discriminator model.

Two logits, d_logits_fake and d_logits_real, are used to compute the loss function during training

```
g_output = gen_model(input_z)
print('Generator output -- shape:', g_output.shape)

d_logits_real = disc_model(input_real)
d_logits_fake = disc_model(g_output)
print('discriminator (real) -- shape:', d_logits_real.shape)
print('discriminator (fake) -- shape:', d_logits_fake.shape)
```

```
(Generator output -- shape: (32, 784)
(discriminator (real) -- shape: (32, 1)
(discriminator (fake) -- shape: (32, 1)
```

Training GAN model

- Next, create a BinaryCrossentropy class object for the loss function to compute generator and discriminator loss of the batch processed before.
- For this operation, an answer label is necessary for each output.
- Let's create a vector filled with 1 for the generator
- The size of this vector is identical to that of the vector d_logits_fake which contains the logit value of the generated image.

```
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)

## Generator loss
g_labels_real = tf.ones_like(d_logits_fake)
g_loss = loss_fn(y_true=g_labels_real, y_pred=d_logits_fake)
print('Generator loss:  {:.4f}'.format(g_loss))
```

Generator loss: 0.6775

Training GAN model

- The discriminator needs two losses.
- Compute the loss that detects fake sample by using d_logits_fake, and compute the loss that detects real sample by using d_logits_real.

```
## Generator lossx
d_labels_real = tf.ones_like(d_logits_real)
d_labels_fake = tf.zeros_like(d_logits_fake)

d_loss_real = loss_fn(y_true=d_labels_real, y_pred=d_logits_real)
d_loss_fake = loss_fn(y_true=d_labels_fake, y_pred=d_logits_fake)
print('discriminator loss: real {:.4f} fake {:.4f}'
      .format(d_loss_real.numpy(), d_loss_fake.numpy()))
```

discriminator loss: real 0.3724 fake 0.7138

- We’re computing various loss clauses step by step to understand the overall idea of the GAN model training

Final Training (1/7)

```
import time


num_epochs = 100
batch_size = 64
image_size = (28, 28)
z_size = 20
mode_z = 'uniform'
gen_hidden_layers = 1
gen_hidden_size = 100
disc_hidden_layers = 1
disc_hidden_size = 100

tf.random.set_seed(1)
np.random.seed(1)
```

```
if mode_z == 'uniform':
    fixed_z = tf.random.uniform(
        shape=(batch_size, z_size),
        minval=-1, maxval=1)
elif mode_z == 'normal':
    fixed_z = tf.random.normal(
        shape=(batch_size, z_size))


def create_samples(g_model, input_z):
    g_output = g_model(input_z, training=False)
    images = tf.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0
```

Final Training (2/7)

```
## Prepare dataset
mnist_trainset = mnist['train']
mnist_trainset = mnist_trainset.map(
    lambda ex: preprocess(ex, mode=mode_z))

mnist_trainset = mnist_trainset.shuffle(10000)
mnist_trainset = mnist_trainset.batch(
    batch_size, drop_remainder=True)
```

Final Training (3/7)

```
## Prepare model
with tf.device(device_name):
    gen_model = make_generator_network(
        num_hidden_layers=gen_hidden_layers,
        num_hidden_units=gen_hidden_size,
        num_output_units=np.prod(image_size))
    gen_model.build(input_shape=(None, z_size))

    disc_model = make_discriminator_network(
        num_hidden_layers=disc_hidden_layers,
        num_hidden_units=disc_hidden_size)
    disc_model.build(input_shape=(None, np.prod(image_size)))
```

```
## Loss function and optimizer
loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
g_optimizer = tf.keras.optimizers.Adam()
d_optimizer = tf.keras.optimizers.Adam()
```

Final Training (4/7)

Execute such operation in the for loop statement in the final code to build a GAN model and to iterate training

```
all_losses = []
all_d_vals = []
epoch_samples = []

start_time = time.time()
for epoch in range(1, num_epochs+1):
    epoch_losses, epoch_d_vals = [], []
    for i,(input_z,input_real) in enumerate(mnist_trainset):

        ## Calculate the loss of generator.
        with tf.GradientTape() as g_tape:
            g_output = gen_model(input_z)
            d_logits_fake = disc_model(g_output, training=True)
            labels_real = tf.ones_like(d_logits_fake)
            g_loss = loss_fn(y_true=labels_real, y_pred=d_logits_fake)

        # Calculate the gradient of g_loss.
        g_grads = g_tape.gradient(g_loss, gen_model.trainable_variables)

        # Optimizer: Apply gradients.
        g_optimizer.apply_gradients(
            grads_and_vars=zip(g_grads, gen_model.trainable_variables))
```

Final Training (6/7)

Execute such operation in the for loop statement in the final code to build a GAN model and to iterate training

```
## Calculate the loss of discriminator
with tf.GradientTape() as d_tape:
    d_logits_real = disc_model(input_real, training=True)

    d_labels_real = tf.ones_like(d_logits_real)

    d_loss_real = loss_fn(
        y_true=d_labels_real, y_pred=d_logits_real)

    d_logits_fake = disc_model(g_output, training=True)
    d_labels_fake = tf.zeros_like(d_logits_fake)

    d_loss_fake = loss_fn(
        y_true=d_labels_fake, y_pred=d_logits_fake)

    d_loss = d_loss_real + d_loss_fake

## Calculate the gradient of d_loss.
d_grads = d_tape.gradient(d_loss, disc_model.trainable_variables)
```

Final Training (7/7)

Execute such operation in the for loop statement in the final code to build a GAN model and to iterate training

```
        ## Optimizer: Apply 'gradients'
        d_optimizer.apply_gradients(
            grads_and_vars=zip(d_grads, disc_model.trainable_variables))

        epoch_losses.append(
            (g_loss.numpy(), d_loss.numpy(),
             d_loss_real.numpy(), d_loss_fake.numpy()))

        d_probs_real = tf.reduce_mean(tf.sigmoid(d_logits_real))
        d_probs_fake = tf.reduce_mean(tf.sigmoid(d_logits_fake))
        epoch_d_vals.append((d_probs_real.numpy(), d_probs_fake.numpy()))
    all_losses.append(epoch_losses)
    all_d_vals.append(epoch_d_vals)
    print(
        'epoch {:03d} | time {:.2f} min |  Average loss >>'
        ' Generator/Discriminator {:.4f}/{:.4f} [Discriminator-Real: {:.4f} Discriminator-Fake: {:.4f}]'
        .format(
            epoch, (time.time() - start_time)/60,
            *list(np.mean(all_losses[-1], axis=0))))
    epoch_samples.append(
        create_samples(gen_## Optimizer: Apply 'gradients'
```

```
...
Epoch 001 | Time 0.66 min | Average loss >> Generator/Discriminator 2.8728/0.2887 [Discriminator-Real: 0.0332 Discriminator-Fake: 0.2555]
Epoch 002 | Time 1.25 min | Average loss >> Generator/Discriminator 5.1191/0.3697 [Discriminator-Real: 0.1236 Discriminator-Fake: 0.2460]
Epoch 003 | Time 1.83 min | Average loss >> Generator/Discriminator 3.0876/0.7191 [Discriminator-Real: 0.3233 Discriminator-Fake: 0.3967]
Epoch 004 | Time 2.43 min | Average loss >> Generator/Discriminator 2.3511/0.7997 [Discriminator-Real: 0.4111 Discriminator-Fake: 0.3887]
```

Training GAN model

- Also, it is helpful to print the average probability of the real sample and fake sample computed by the discriminator for each iteration
- If this probability is close to 0.5, that means the discriminator cannot distinguish between real and fake images.

```
import itertools


fig = plt.figure(figsize=(16, 6))

## Loss graph
ax = fig.add_subplot(1, 2, 1)
g_losses = [item[0] for item in itertools.chain(*all_losses)]
d_losses = [item[1]/2.0 for item in itertools.chain(*all_losses)]
plt.plot(g_losses, label='Generator loss', alpha=0.95)
plt.plot(d_losses, label='Discriminator loss', alpha=0.95)
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

epochs = np.arange(1, 101)
epoch2iter = lambda e: e*len(all_losses[-1])
epoch_ticks = [1, 20, 40, 60, 80, 100]
newpos = [epoch2iter(e) for e in epoch_ticks]
ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)
```

```
## Print discriminator
ax = fig.add_subplot(1, 2, 2)
d_vals_real = [item[0] for item in itertools.chain(*all_d_vals)]
d_vals_fake = [item[1] for item in itertools.chain(*all_d_vals)]
plt.plot(d_vals_real, alpha=0.75, label=r'Real: $D(\mathbf{x})$')
plt.plot(d_vals_fake, alpha=0.75, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)
```

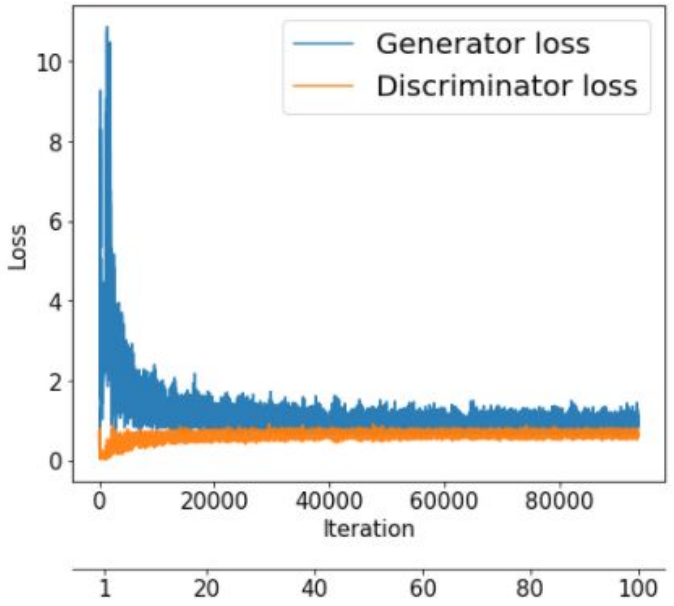
Training GAN model

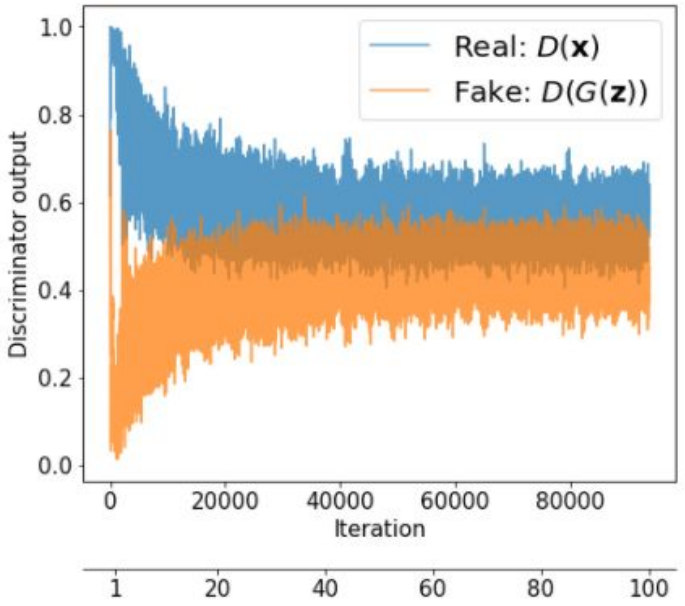
- Also, it is helpful to print the average probability of the real sample and fake sample computed by the discriminator for each iteration
- If this probability is close to 0.5, that means the discriminator cannot distinguish between real and fake images.

```
## Print discriminator
ax = fig.add_subplot(1, 2, 2)
d_vals_real = [item[0] for item in itertools.chain(*all_d_vals)]
d_vals_fake = [item[1] for item in itertools.chain(*all_d_vals)]
plt.plot(d_vals_real, alpha=0.75, label=r'Real: $D(\mathbf{x})$')
plt.plot(d_vals_fake, alpha=0.75, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

ax2 = ax.twiny()
ax2.set_xticks(newpos)
ax2.set_xticklabels(epoch_ticks)
ax2.xaxis.set_ticks_position('bottom')
ax2.xaxis.set_label_position('bottom')
ax2.spines['bottom'].set_position(('outward', 60))
ax2.set_xlabel('Epoch', size=15)
ax2.set_xlim(ax.get_xlim())
ax.tick_params(axis='both', which='major', labelsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)

plt.show()
```

Training GAN model

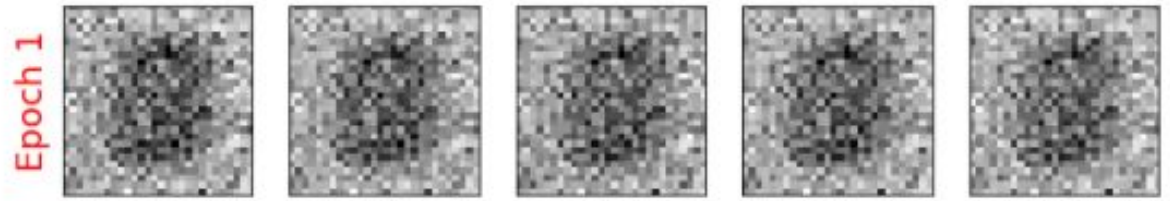
- Also, it is helpful to print the average probability of the real sample and fake sample computed by the discriminator for each iteration
- If this probability is close to 0.5, that means the discriminator cannot distinguish between real and fake images.



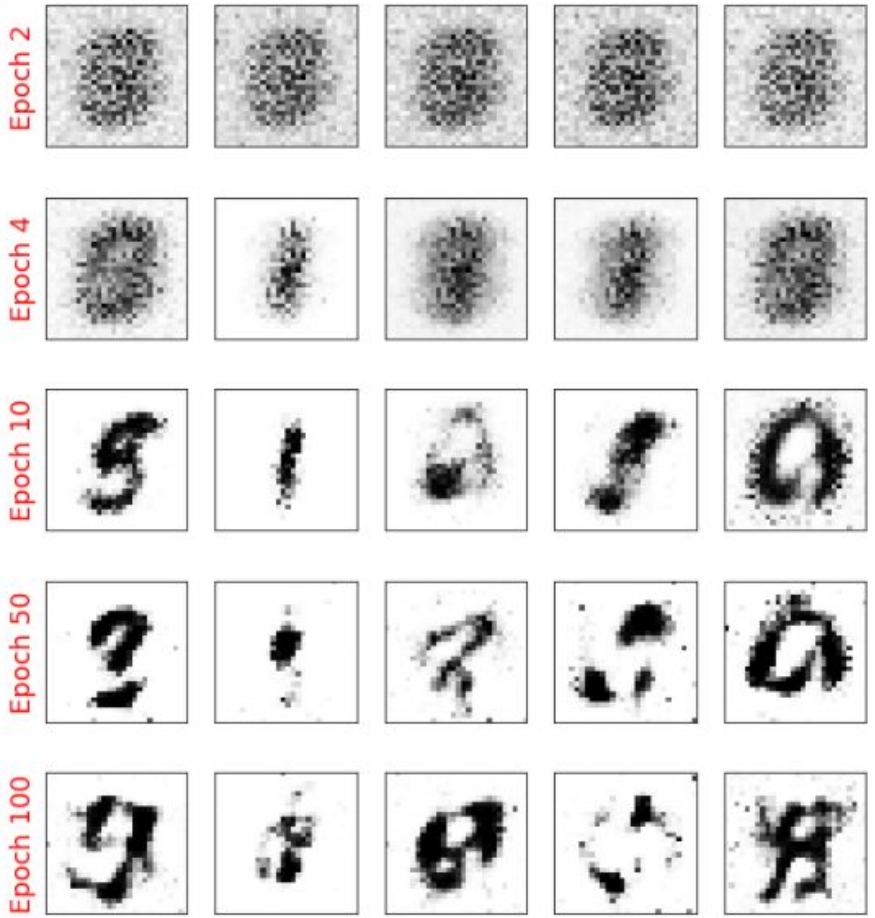


Training GAN model

In [25]:
```
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, 'Epoch {}'.format(e),
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)

        image = epoch_samples[e-1][j]
        ax.imshow(image, cmap='gray_r')

plt.show()
```



Training GAN model




End of
Document