COMP2211 Exploring Artificial Intelligence

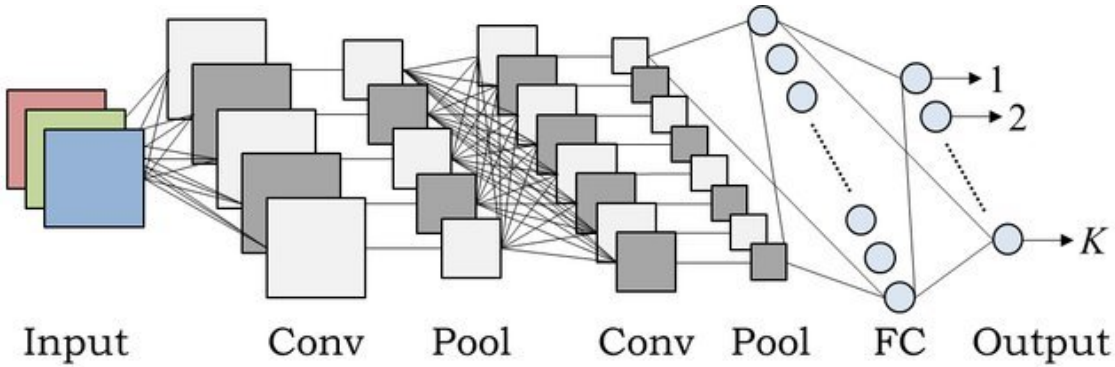
Convulutional Neural Network

Huiru Xiao

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

Overview

- In this topic, we will continue to discuss how to implement an AI for image recognition and classification.
- This time, we use Convolutional Neural Networks (CNNs) to train the AI agent to know how to recognize/classify images.
- CNNs can also be used to train AI agents to solve other problems, such as natural language processing, speech recognition, recommendation systems, image segmentation, medical image analysis, financial time series, etc.



Convolutional Neural Network

Convolutional Neural Network

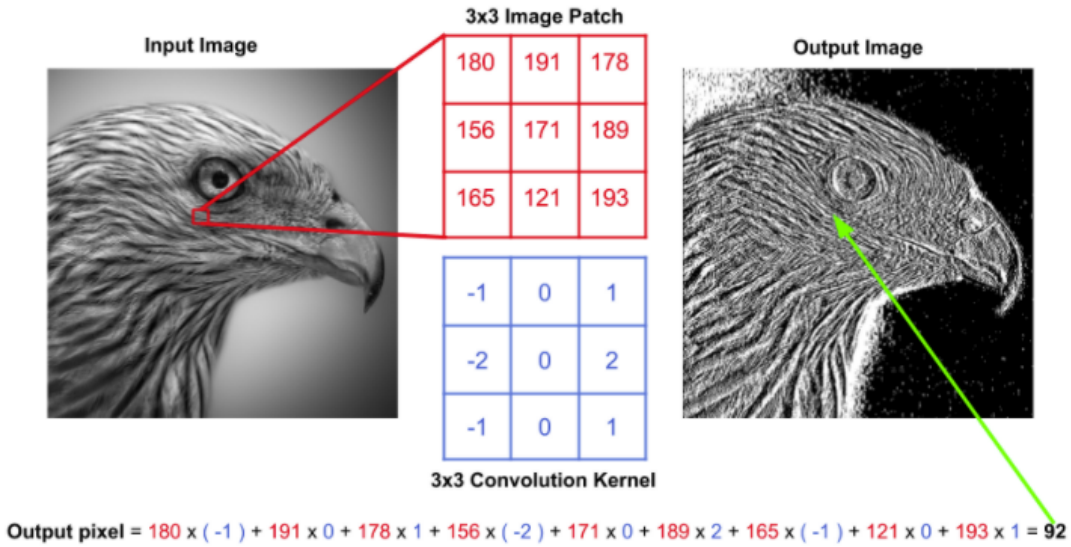
- Convolutional neural networks were developed in the late 1980s and then forgotten about due to the lack of processing power.
- With the powerful Graphical Processing Units (GPUs), the research on CNNs and deep learning was given new life.
- CNNs are used in many applications:
  - Object recognition in images and videos (e.g., image search in Google, tagging friends' faces on Facebook, adding filters in Snapchat and tracking movement in Kinect).
  - Natural language processing (speech recognition in Google Assistant or Amazon's Alexa)
  - Playing games (the defeat of the World 'Go' champion by DeepMind at Google)
  - Medical innovation (from drug discovery to prediction of disease)

What are Convolutional Neural Networks?

- A Convolutional Neural Network (CNN or ConvNet) is very much related to the standard Artificial Neural Network (ANN) that we have previously encountered.

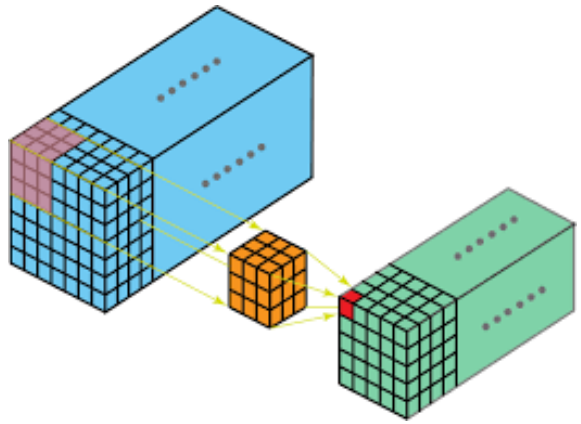
Definition

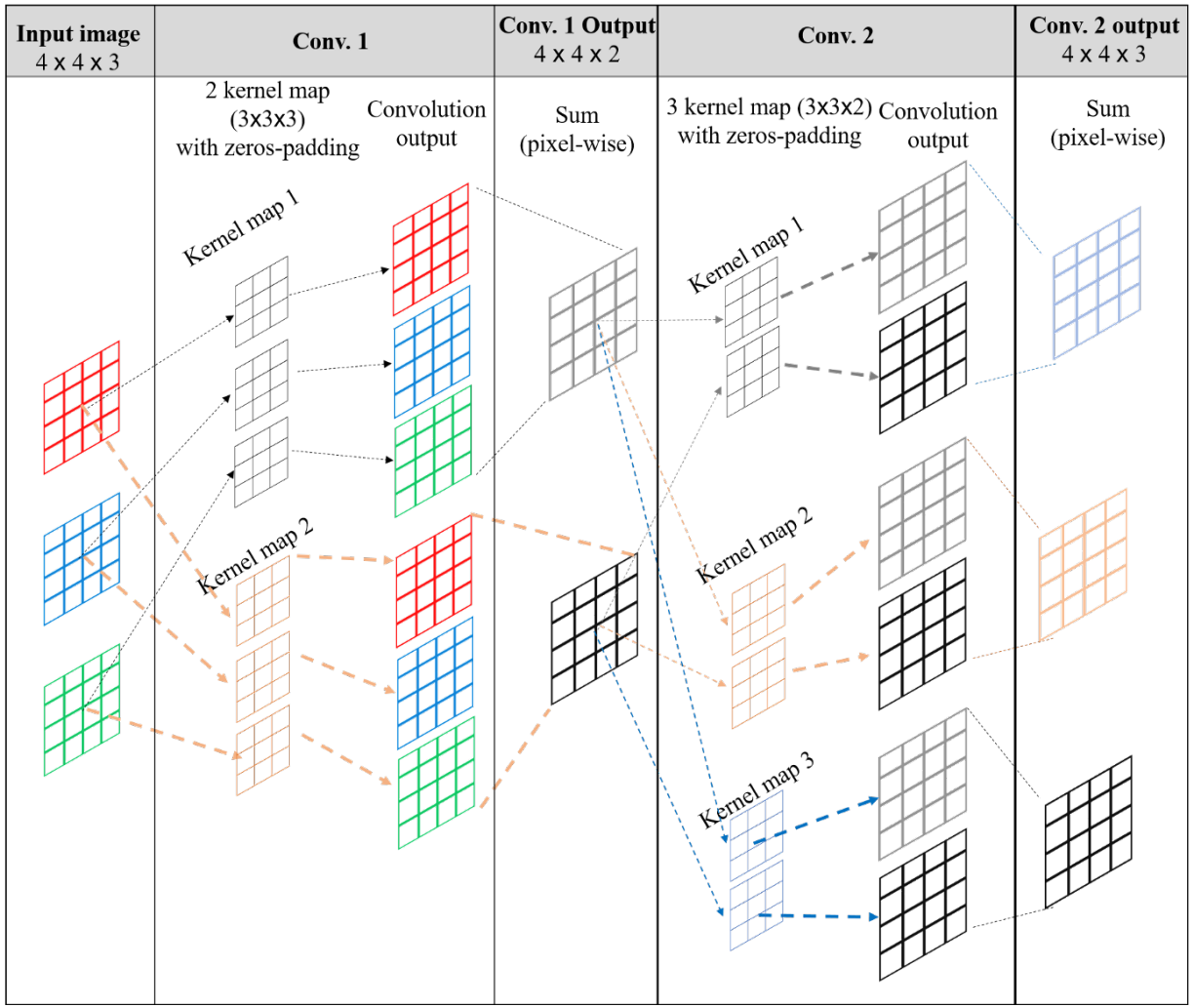
A convolutional neural network is a neural network with a convolution operation in at least one of the layers.



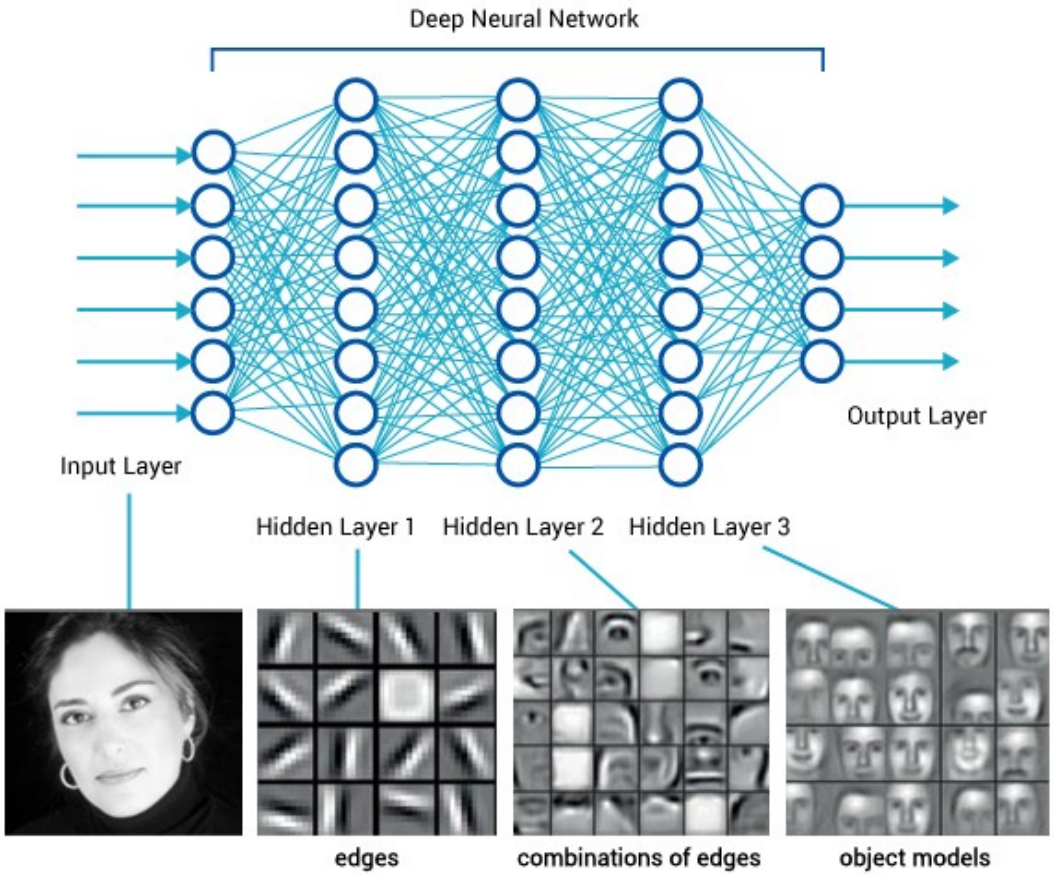
How Does Convolution Feed Into CNNs?

- First, we should recognize that every pixel in an image is a feature, and that means it represents an input node.
- The result from each convolution is placed into the next layer in a hidden node. Each feature or pixel of the convolved image is a node in the hidden layer.
- So, what are the numbers in the kernel? They are weights connecting the feature of the input image and the node of the hidden layer. These weights that connect to the nodes need to be learned exactly as in a regular neural network.

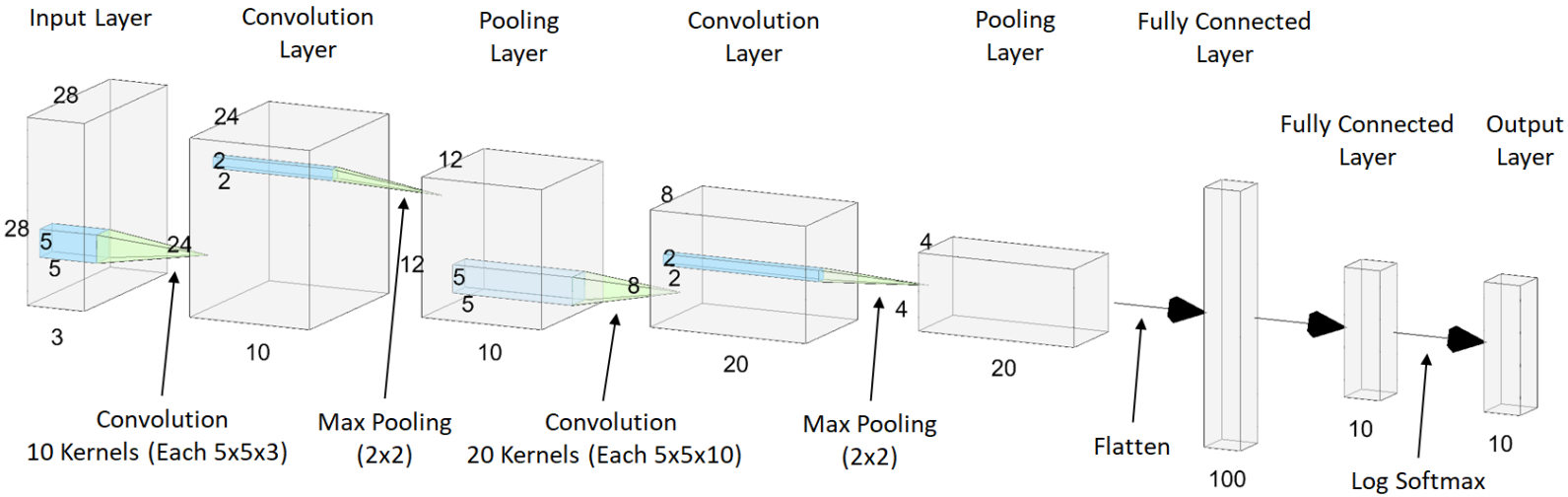



Input image
4 x 4 x 3
Conv. 1
2 kernel map
(3x3x3)
with zeros-padding
Convolution
output
Kernel map 1
Kernel map 2
Conv. 1 Output
4 x 4 x 2
Sum
(pixel-wise)
Conv. 2
3 kernel map (3x3x2)
with zeros-padding
Convolution
output
Kernel map 1
Kernel map 2
Kernel map 3
Conv. 2 output
4 x 4 x 3
Sum
(pixel-wise)

Intuitive Idea



Convolutional Neural Network



- Three main types of layers to build a Convolutional Neural Network (CNN/ConvNet).
  - Input layer
  - Convolutional layer
  - Pooling layer
  - Fully-connected layer
  - Output layer

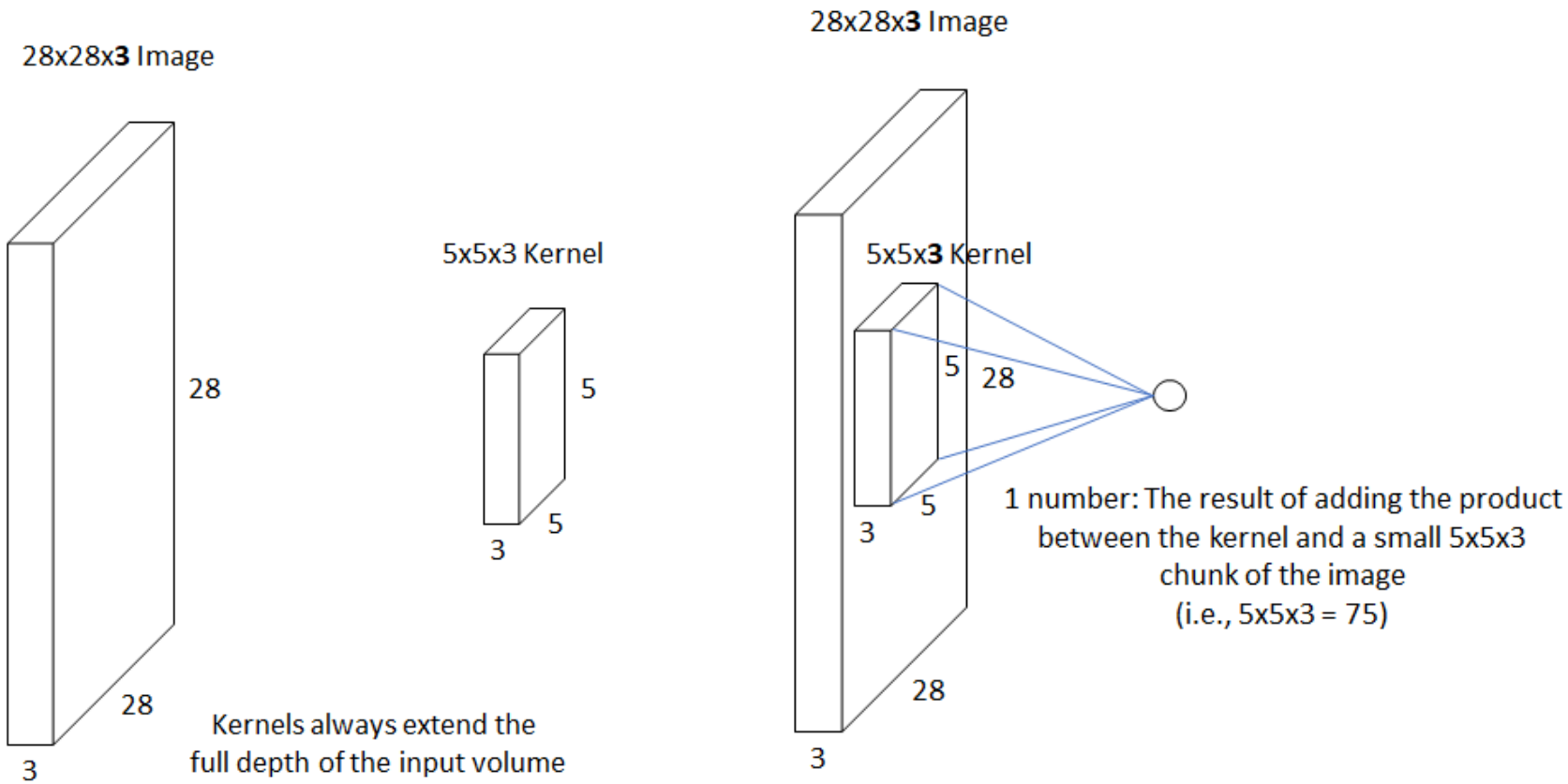
Input Layer

- The input image is placed into this layer. It can be a single-layer 2D image (grayscale), 3-channel 2D images (RGB color) or 3D.

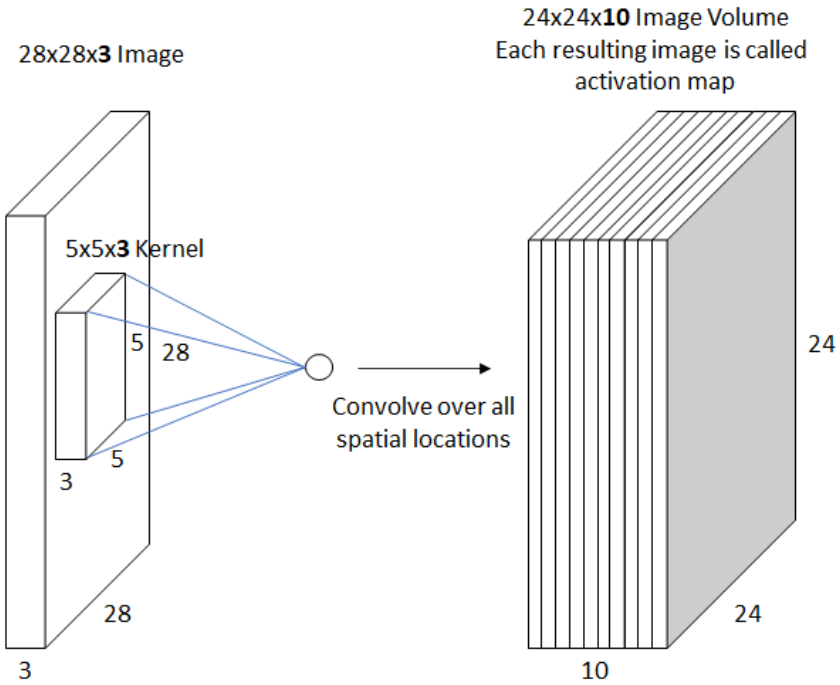
Note: In the example on the last page, the input images are 3-channel 2D images in 28 pixels by 28 pixels (RGB Color).

- Kernels need to be learned, are in the same depth as the input.
Note: In the example on the last page, the depth of kernels applied on the input layer is 3 in size, 5 by 5.

Convolutional Layer



Convolutional Layer



- Input:
  - Image: $28 \times 28 \times 3$
  - 10 Kernels: $10 \times 5 \times 5 \times 3$
- Assumption: Ignore those pixels that do not have 24 neighboring pixels.
- Output: $24 \times 24 \times 10$

Stride

Stride is the amount of movement between applications of the kernel to the input image.

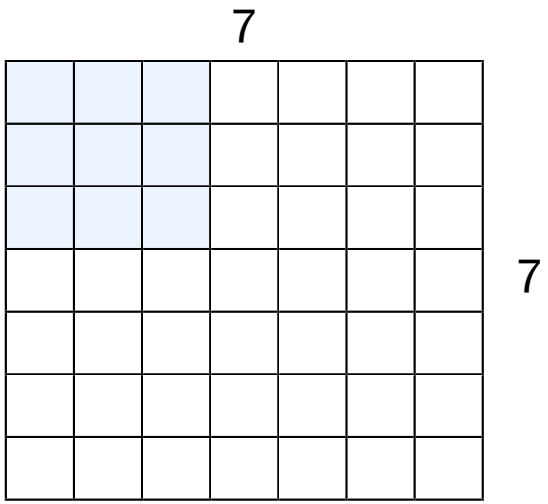
- The default stride or strides in 2D is (1,1) for the height and the width movement.
- The kernel is moved across the image left to right, top to bottom, with a one-pixel column change on the horizontal movements, then a one-pixel row change on the vertical movements.
- The stride can be changed, which has an effect both on how to kernel is applied to the image, and in turn, the size of the resulting output.
- The stride can be specified in Keras on the Conv2D layer via the 'stride' argument and specified as a tuple with height and width.

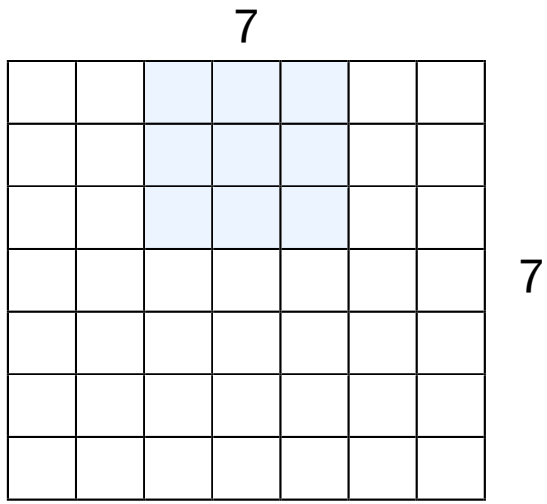
Why setting stride to a larger number?

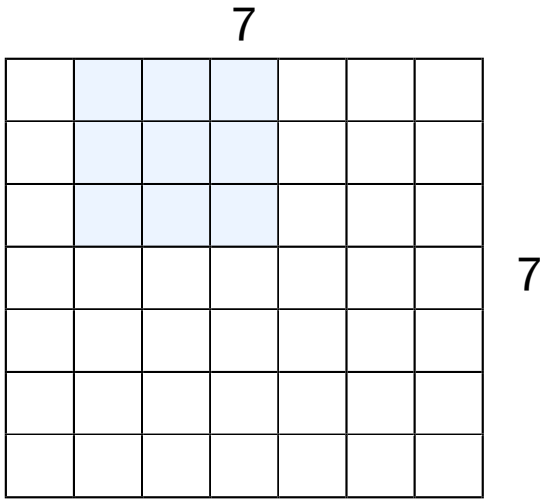
- Larger strides lead to lesser overlaps which means lower output volume.
- Lesser memory needed for output.
- It avoids overfitting especially in case of image processing having a large number of attributes.

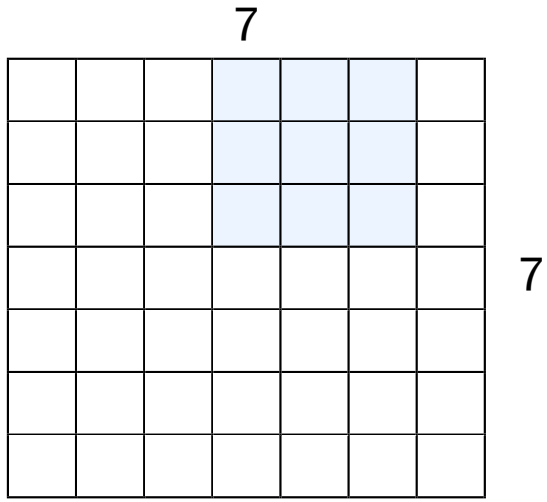
A Closer Look at Spatial Dimensions

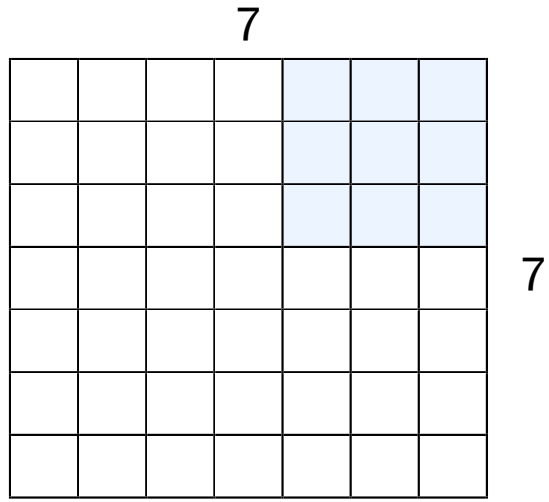
- Suppose an input image is in size 7 by 7 and convolution is performed using a 3 by 3 kernel.





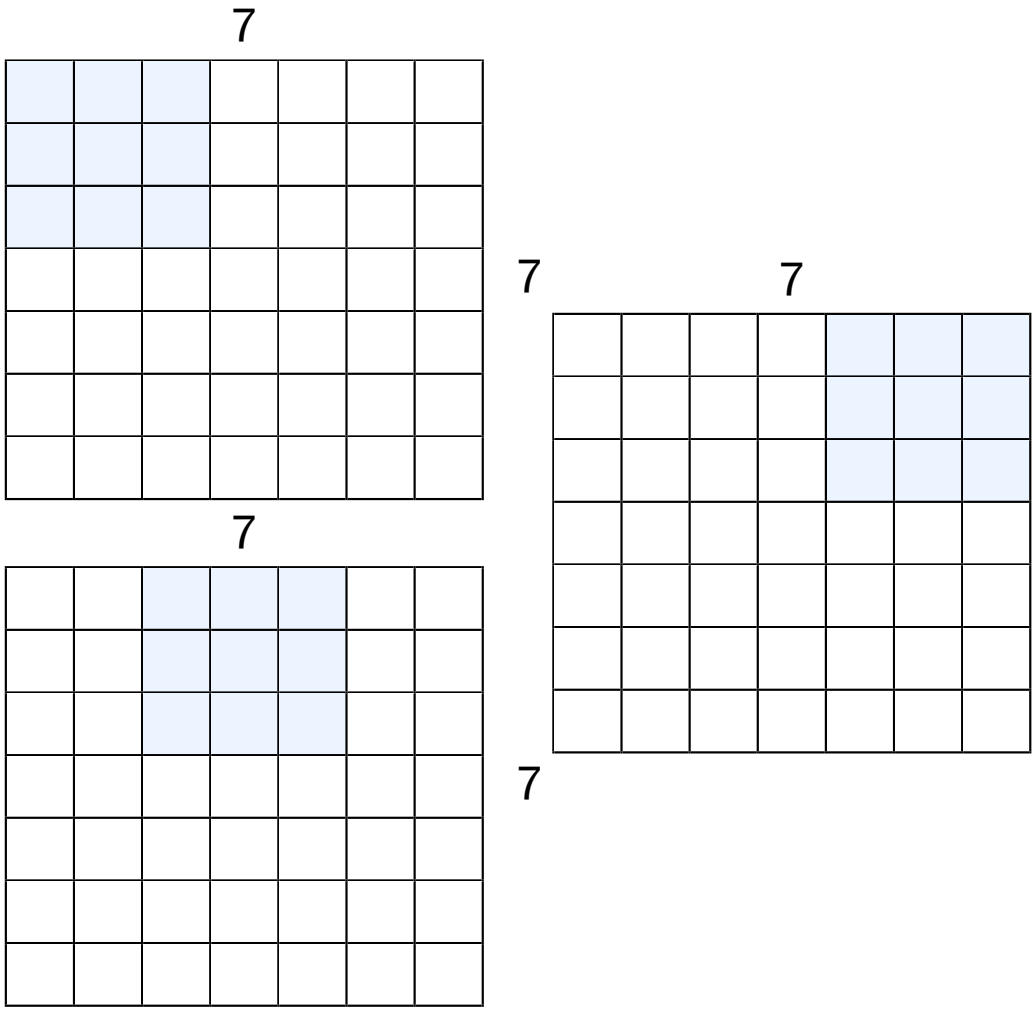






That is why the output is 5×5.

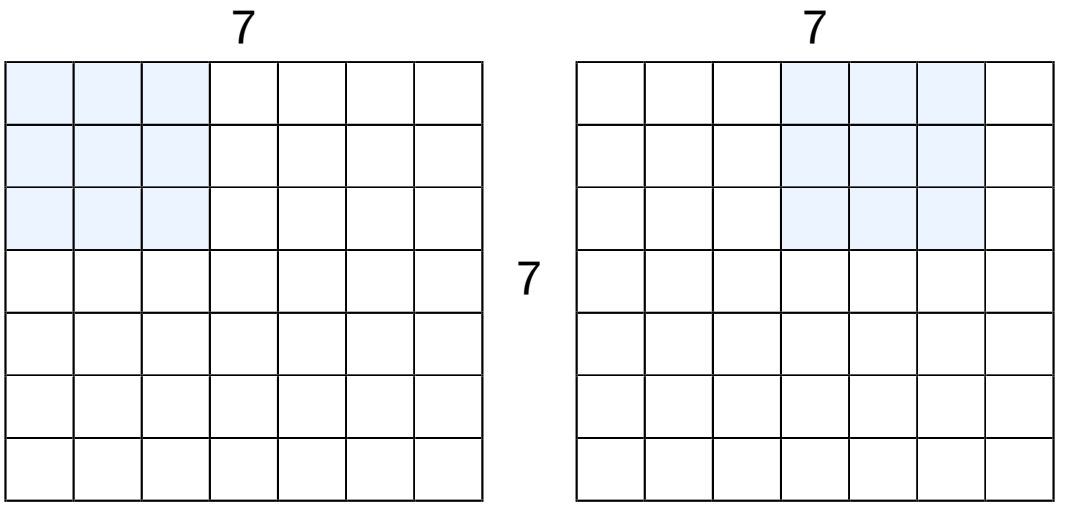
A Closer Look at Spatial Dimensions with Stride



When the stride is 2, we move the kernels to 2 pixels at a time and so on.

That is why the output is 3×3.

A Closer Look at Spatial Dimensions with Stride



How about a 7×7 input image and a 3×3 kernel applied with stride 3?

We cannot apply 3×3 kernel with stride 3 since it does not fit (it does not reach the input image boundary).

Useful Formula for Determining the Size of Output Image

Output size = (Size of image dimension - Size of kernel dimension) / Stride + 1

Example: Size of image dimension = 7, Size of kernel dimension = 3

- Output size:
  - Stride 1 $\Rightarrow$ Output size = (7 - 3)/1 + 1 = 5
  - Stride 2 $\Rightarrow$ Output size = (7 - 3)/2 + 1 = 3
  - Stride 3 $\Rightarrow$ Output size = (7 - 3)/3 + 1 = 2.33 (Does not fit!)

Zero Padding

- When a kernel convolves a given input image, the dimensions of the output image are reduced.
- For example:
  - The input image size is 28×28.
  - Convolving the image using a 3×3 kernel, the output image size is 26×26. Since we ignore those pixels without 8-neighbors.

How to preserve the size of the output image? Zero padding!!!

- Zero padding occurs when we add a border of pixels all with value zero around the boundaries of the input images.
- For implementation of CNN using Keras, there are two categories of padding:
  - valid (i.e. no padding)
  - same (i.e. pad the original input with zeros before we convolve so that the output size is the same size as the input size)

Zero Padding

0	0	0	0	0	0	0	0	0
0								0
0								0
0								0
0								0
0								0
0								0
0								0
0	0	0	0	0	0	0	0	0

Suppose we do padding with a 1-pixel width border and with stride 1. What is the output image size?

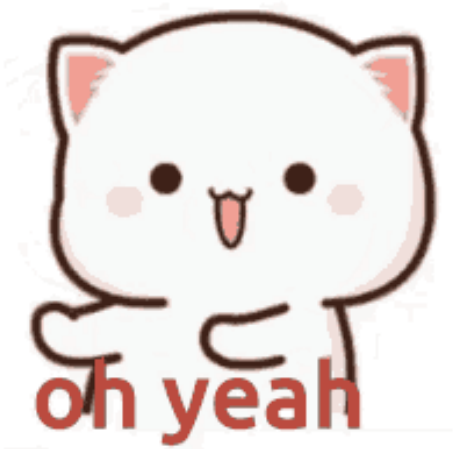
Example:

- Size of image dimension = 7
- Size of kernel dimension = 3
- Pad with 1 pixel border
- Output image size = ((7 + 2) - 3) / 1 + 1 = 7

- In general, it is common to see convolutional layers with stride 1, kernels of size K×K, and zero-padding with (K-1)/2. It will preserve size spatially.
- Padded border size
  - If K = 3, zero pad with (3 - 1) / 2 = 1 (1-pixel border)
  - If K = 5, zero pad with (5 - 1) / 2 = 2 (2-pixel border)
  - If K = 7, zero pad with (7 - 1) / 2 = 3 (3-pixel border)

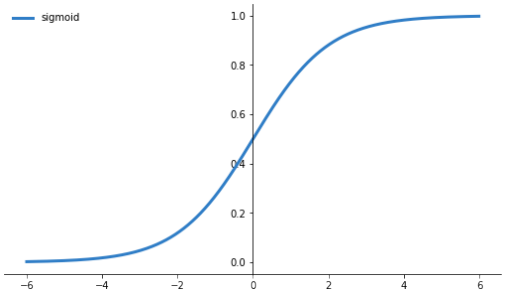
All-in-One Example

- Input image size $= 32 \times 32 \times 3$
- 10 Kernels, each of size $5 \times 5 \times 3$
- Stride $= 1$
- Pad with 2 pixels border
- Output size $=$ ($(32 + 4)$ - 5) / $1 + 1 = 32$
- Output volume size $= 32 \times 32 \times 10$
- Number of parameters in this layer
  - Each kernel has $5 \times 5 \times 3 + 1 = 76$ parameters ($+1$ for bias)
  - We have 10 kernels, and there are $76 \times 10 = 760$ parameters.

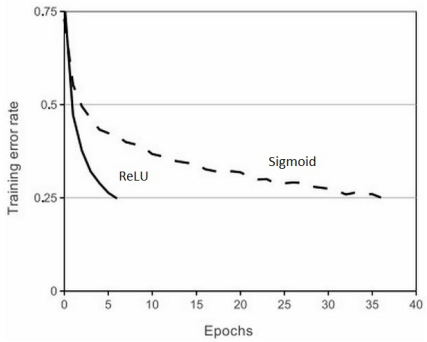


Non-linearity

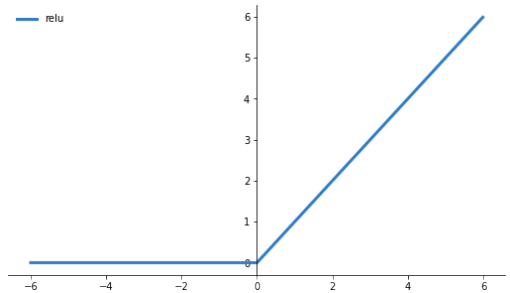
- Convolution is a linear operation. We need non-linearity; otherwise, 2 convolution layers would be no more powerful than 1.
- The non-linearity is NOT its own distinct layer of CNN, but comes as part of the convolution layer as it is done in the neurons (just like a normal artificial neural network).
- In a neural network, we use different activation functions, typically using sigmoid. But rectified linear unit (ReLU) is more popular for CNN as it does not require any expensive computation. It has been shown to speed up the convergence of stochastic gradient descent algorithms.



$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$



Convergence Rates



$$ReLU(x) = max(0, x)$$

Pooling Layer

- The pooling layer is key to ensuring that the subsequent layers of the CNN can pick up larger-scale detail than just edges and curves.
- It does this by merging pixel regions in the convolved image together (shrinking the image) before attempting to learn kernels on it.
- Two popular ones:
  - Max pooling: Done by applying a max filter to non-overlapping subregions of the initial representation.
  - Average pooling: Done by applying an average filter to non-overlapping subregions of the initial representation.

12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

Max pooling

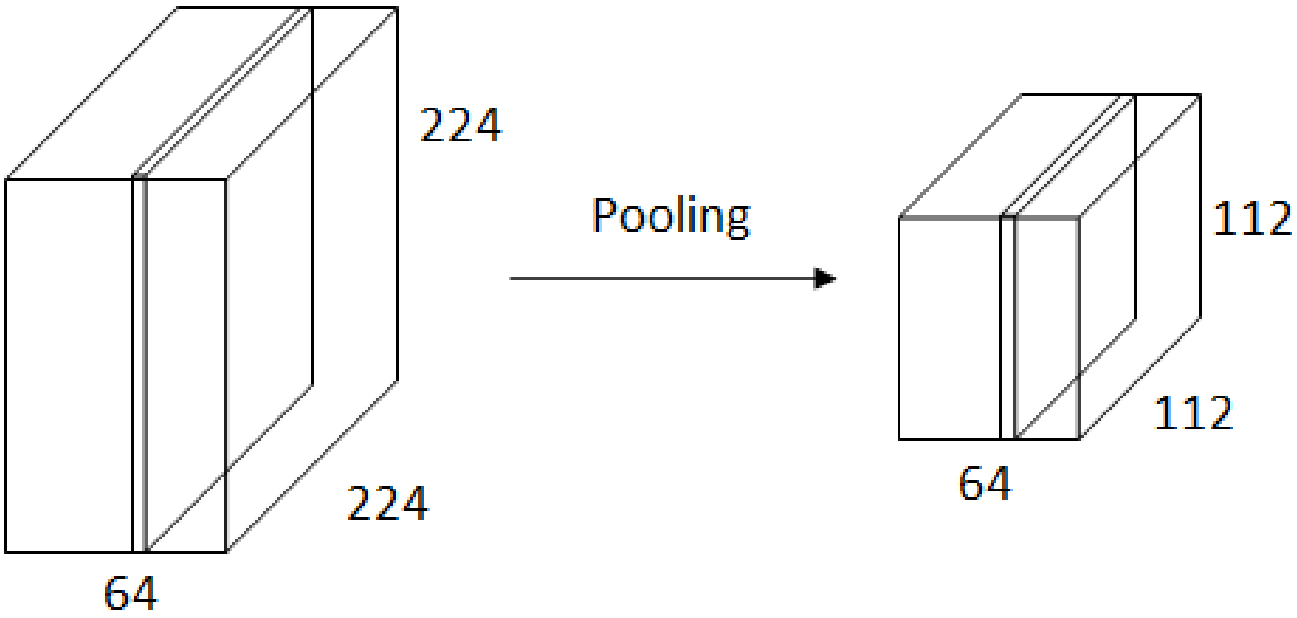
20	30
112	37

Average pooling

13	8
79	20

Pooling can also tolerate local distortion and translation of patterns of interest and reduce the computational load, memory usage, and the number of parameters.

Pooling



Fully-connected (Dense) Layer

- The fully-connected layer is a traditional multi-layer perceptron that uses a softmax activation function in the output layer.
- The term "fully-connected" implies that every neuron in the previous layer is connected to every neuron on the next layer.
- The output from the convolutional and pooling layer represents high-level features of the input image.
- The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset.

Summary

Convolutional + Pooling layers act as feature extractors from the input image, while the fully-connected layer acts as a classifier.

Overall Training Process of CNN

1. We initialize all kernels and parameters/weights with random values.
2. The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with the forward propagation in a fully-connected layer) and finds the output probabilities for each class.
3. Calculate the total error at the output layer.
4. Use backpropagation to update all kernel values/weights and parameter values to minimize the output error.
5. Repeat steps 2-4 with all images in the training set.

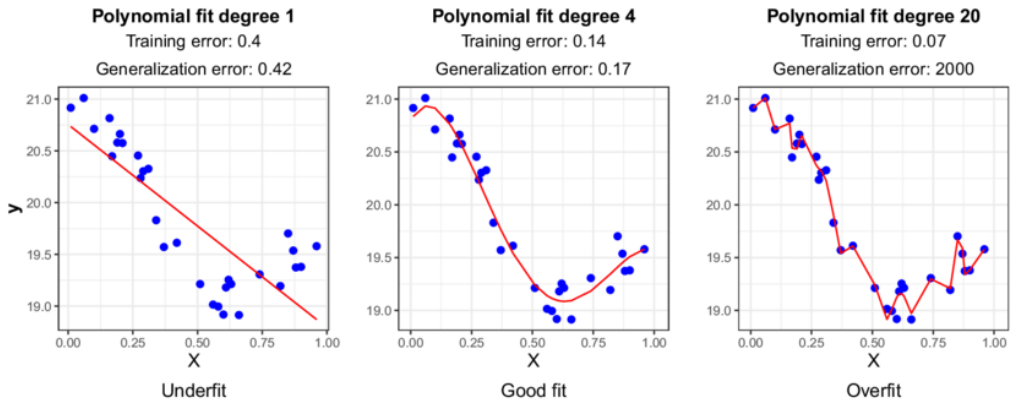
Recall: Overfitting and Underfitting Problem

Overfitting

Overfitting refers to a model that was trained too much on the particulars of the training data (when the model learns the noise in the dataset). A model that is overfitted will not perform well on new, unseen data.

Underfitting

Underfitting typically refers to a model that has not been trained sufficiently. This could be due to insufficient training time or a model that was simply not trained properly. A model that is under fitted will perform poorly on the training data and new, unseen data alike.



Dropout Layer

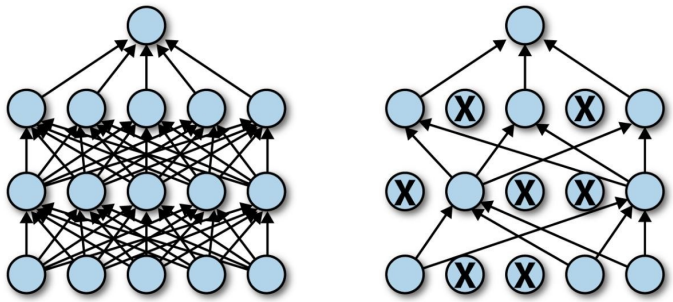
- Large neural network trained on relatively small datasets can overfit the training data.
- This has the effect of the model learning the statistical noise in the training data, which results in poor performance when the model is evaluated on new data, e.g. a test dataset.
- Generalization error increases due to overfitting.

Dropout is a simple way to prevent neural networks from overfitting.

- Dropout is a regularization method. During training, some number of layer outputs are randomly ignored or "dropped out".
- Dropout has the effect of forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
- Because the outputs of a layer under dropout are randomly subsampled. it has the effect of reducing the capacity or thinning the network during training. As such, a wider network, e.g. more nodes, may be required when using dropout.

How to Dropout?

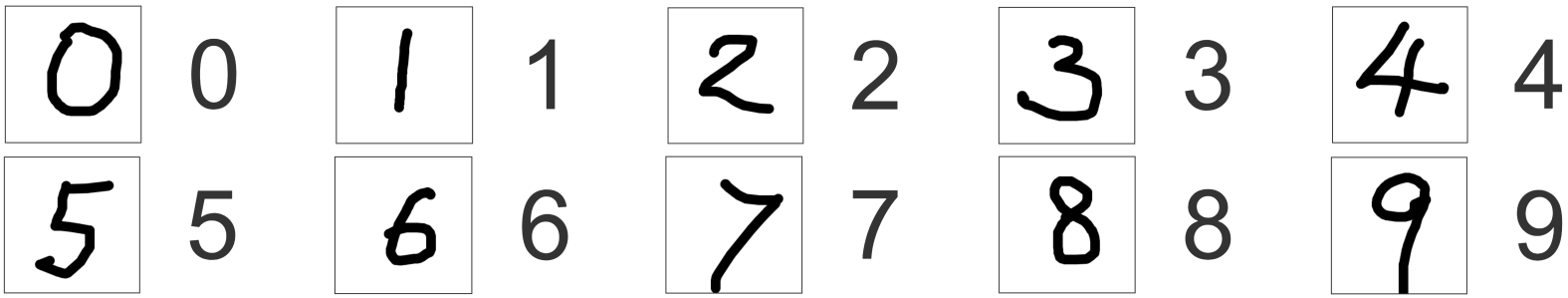
- Dropout is implemented per-layer in a neural network.
- It can be used with most types of layers, such as dense fully-connected layers, convolutional layers, etc.
- Dropout may be implemented on any or all hidden layers in the network and the input layer.
- It is not used on the output layer.
- A parameter is introduced that specifies the probability at which the outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained.
- A common value is a probability of 0.5 for retraining the output of each node in a hidden layer.



Handwritten Digits Recognition using CNN

Handwritten Digits Recognition using CNN

We will build a CNN model to recognize/classify handwritten digits.



Dataset

- We use the Modified National Institute of Standards and Technology (MNIST) dataset.
- This dataset contains two sets of samples:
  - Training data: 60000 28 pixel $\times$28 pixel images of handwritten digits from 0 to 9.
  - Testing data: 10000 28 pixel $\times$ 28 pixel images.

Procedures

1. Import required libraries and define global variables
2. Load and prepare the data
3. Build the model
4. Compile the model
5. Train the model
6. Evaluate the model accuracy
7. Save the model
8. Use the model

Please refer to the lecture on MLP for data exploration.

1. Import Required Libraries and Define Global Variables

```
import numpy as np                                  # Import numpy library
import matplotlib.pyplot as plt                     # Import mathplot library
import datetime                                     # Import datetime library
from keras.datasets import mnist                    # Import MNIST dataset
from keras.models import Sequential                 # Import Sequential class
from keras.layers import Conv2D, MaxPooling2D       # Import Conv2D, MaxPooling2D class
from keras.layers import Dense, Dropout, Flatten    # Import Dense, Dropout, Flatten class
from keras.utils import to_categorical              # Import numpy-related utilities
from keras.callbacks import TensorBoard             # Import TensorBoard class
from keras.models import load_model                 # Import load\_model method
from tensorflow.keras.utils import plot_model       # Import plot\_model method
from tensorflow.math import confusion_matrix        # Import confusion matrix method
# Import spare categorical crossentroy loss
from tensorflow.keras.metrics import categorical_crossentropy
batch_size = 128                # Number of samples per gradient update
num_classes = 10                # Number of classes in the dataset
epochs = 1                      # Number of epochs to train the model
img_rows, img_cols = 28, 28     # Image dimensions
```

2. Load and Prepare the Data

```
# Load data
# x_train is a NumPy array of grayscale image data with shapes (60000, 28, 28)
# y_train is a NumPy array of digit labels (in range 0-9) with shape (60000,)
# x_test is a NumPy array of grayscale image data with shapes (10000, 28, 28)
# y_test is a NumPy array of digit labels (in range 0-9) with shape (10000,)
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.reshape(60000,28,28,1) # Reshape the data to 4-dimension
x_test = x_test.reshape(10000,28,28,1)   # Reshape the data to 4-dimension

# There are 10 classes and classes are represented as unique integers
# To do so, transforming the integer into a 10 element binary vector with
# a 1 for the index of the class value, and 0 values for all other classes
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

One-Hot Encoding

- to_categorical() converts class vector (integers from 0 to number of classes) to binary class matrix, for use of categorical_crossentropy.
- Example: Assuming the labeled dataset has a total of six classes (0 to 5), y_train is the true label array.

```
y_train = [1, 0, 3, 4, 5, 0, 2, 1]
to_categorical(y_train, num_classes=6)

# Output
# array([[ 0.,  1.,  0.,  0.,  0.,  0.],
#        [ 1.,  0.,  0.,  0.,  0.,  0.],
#        [ 0.,  0.,  0.,  1.,  0.,  0.],
#        [ 0.,  0.,  0.,  0.,  1.,  0.],
#        [ 0.,  0.,  0.,  0.,  0.,  1.],
#        [ 1.,  0.,  0.,  0.,  0.,  0.],
#        [ 0.,  0.,  1.,  0.,  0.,  0.],
#        [ 0.,  1.,  0.,  0.,  0.,  0.]])
```

When to Use One-Hot Encoding?

In a situation where data has no relation to each other.

- The order of integers treated as a significant characteristic by machine learning algorithm. In other words, a large number will be interpreted as better or more significant than a smaller number.
- While this is useful in some ordinal scenarios, certain input data lacks a ranking for categorical values, which can cause problems with predictions and performance.
- That is why we use a one-hot encoding.

Benefits of One-hot Encoding

Training data is more usable and expressive as a result of one-hot encoding, and it can be re-scaled easily.

Question

Why is one-hot encoding applied to labels?

3. Build the Model

- Instead of building the model from scratch, we will use the software library, Keras, instead.
- Layers
  - Layer 1: Convolutional layer with 32 3×3 kernels and ReLU activation (Input size: 28×28×1)
  - Layer 2: Convolutional layer with 64 3×3 kernels and ReLU activation
  - Layer 3: Max Pooling layer with a 2×2 kernel.
  - Layer 4: Dropout layer with a rate 0.25
    It randomly sets input units to 0 with a frequency of 0.25 at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1-rate) such that the sum over all inputs is unchanged.
  - Layer 5: Flatten layer that will flatten 2D image into 1D
  - Layer 6: Dense layer with 128 neurons and ReLU activation
  - Layer 7: Dropout layer with a rate 0.5
  - Layer 8: Dense layer with 10 neurons (1 per class) and Softmax activation

```
model = Sequential() # Create a Sequential object
# Add a convolutional layer with 32 kernels, each of size 3x3
# Use ReLU activation function, padding="valid", strides=(1,1)
# Specify the input size to this convolutional layer: (28,28,1)
# Note: Input size needs to be specified for the first layer only
model.add(Conv2D(filters=32, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
# Add another convolutional layer with 64 kernels, each of size 3x3
# Use ReLU activation function, padding="valid", strides=(1,1)
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu'))
# Add a max pooling layer of size 2 x 2
model.add(MaxPooling2D(pool_size=(2, 2)))
# Add a dropout layer to prevent a model from overfitting
model.add(Dropout(0.25))
# Add a flatten layer to convert the pooled data to a single column
# that is passed to the fully-connected layer
model.add(Flatten())
# Add a dense layer (fully-connected layer) and use ReLU activation function
model.add(Dense(units=128, activation='relu'))
# Add a dropout layer tpo prevent a model from overfitting
model.add(Dropout(0.5))
# Add a dense layer (fully-connected layer) and use Softmax activation function
model.add(Dense(units=num_classes, activation='softmax'))
```

- Print model summary

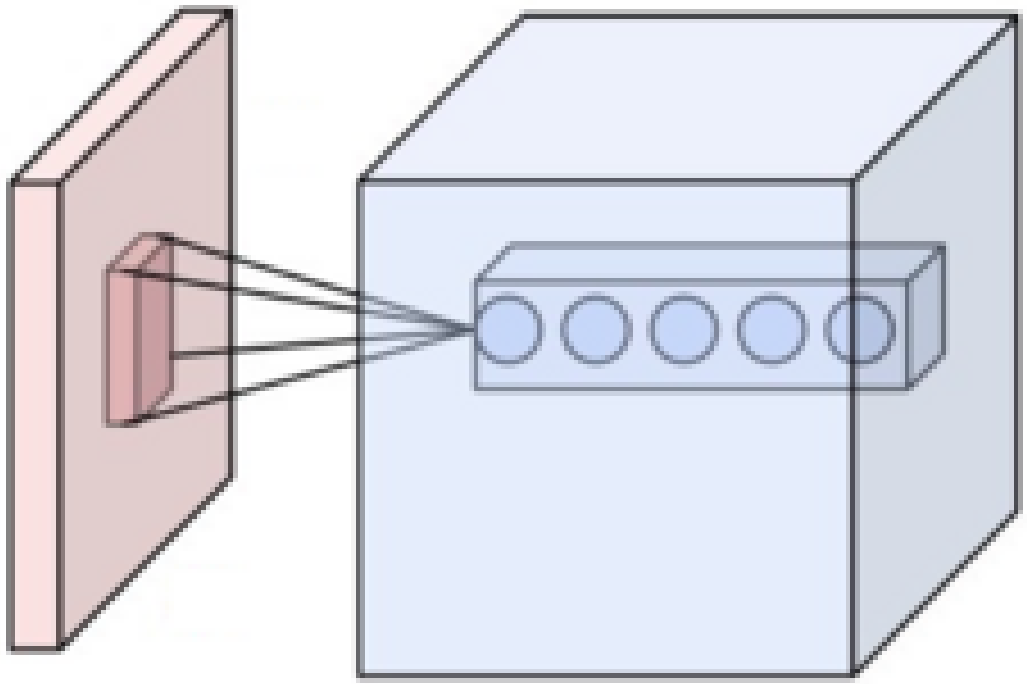
```
model.summary()
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d (Conv2D)              (None, 26, 26, 32)        320

 conv2d_1 (Conv2D)            (None, 24, 24, 64)        18496

 max_pooling2d (MaxPooling2D)  (None, 12, 12, 64)       0

 dropout (Dropout)            (None, 12, 12, 64)        0

 flatten (Flatten)            (None, 9216)              0

 dense (Dense)                (None, 128)               1179776

 dropout_1 (Dropout)          (None, 128)               0

 dense_1 (Dense)              (None, 10)                1290

=================================================================
Total params: 1,199,882
Trainable params: 1,199,882
Non-trainable params: 0
_________________________________________________________________
```

Question: How to calculate $\#param$ in a conv layer?

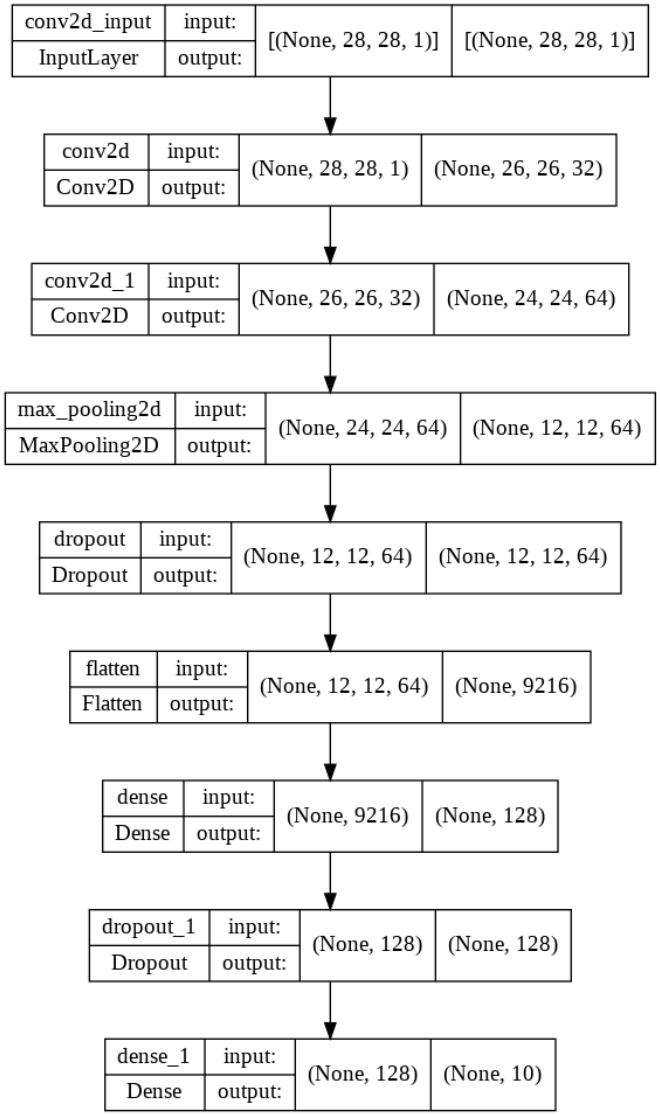
In a conv layer,

$$\begin{aligned} \#param &= \#weights + \#biases \\ &= \text{kernel size} \cdot \text{kernel depth} \cdot \#\text{kernels} + \#\text{kernels} \\ &= (\text{kernel size} \cdot \text{kernel depth} + 1) \cdot \#\text{kernels} \end{aligned}$$



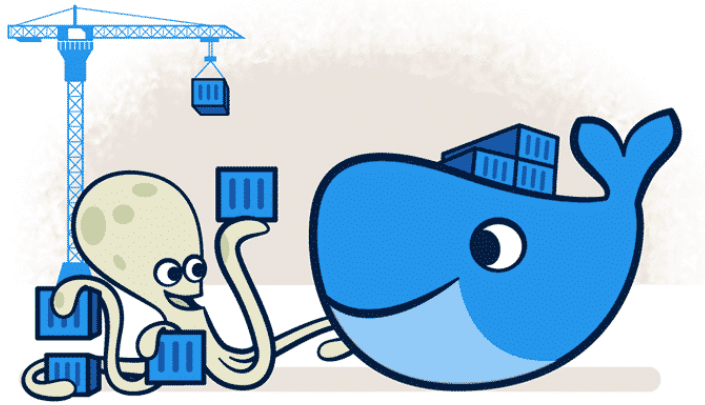
- Plot the model

```
plot_model(model, show_shapes=True, show_layer_names=True)
```



4. Compile the Model

```
# Compile the model, i.e., configures the model for training
# Use crossentropy loss function since there are two or more label classes.
# Use adam algorithm (a stochastic gradient descent method)
# Use accuracy as metric, i.e., report on accuracy
model.compile(optimizer='adam', # Default learning rate is 0.001
              loss=categorical_crossentropy,
              metrics=['accuracy'])
```



5. Train the Model

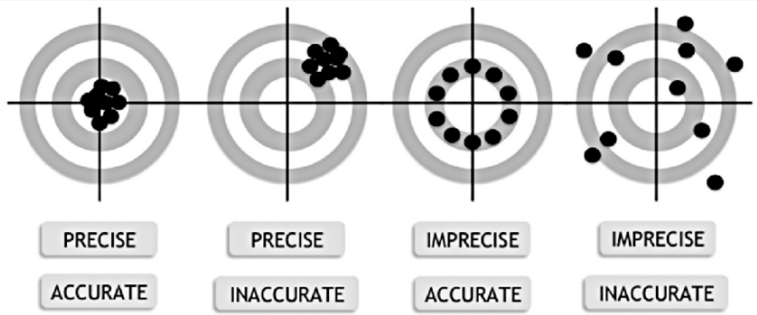
```
# Create TensorBoard object to track experiment metrics like loss and
# accuracy, visualizing the model graph, etc.
log_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# Fit the model, i.e., train the model
# Specify training data and labels
# Speicfy batch size, i.e., number of samples per gradient update
# Specify validation data, i.e., data on which to evaluate the loss
# Write TensorBoard logs after every batch of training to monitor our metrices
training_history = model.fit(x_train, y_train,
                             batch_size=batch_size,
                             epochs=epochs,
                             validation_data=(x_test, y_test),
                             callbacks=[tensorboard_callback]
)
```

6. Evaluate the Model Accuracy

```
# Evaluate the model
# Specify test data and labels
# Set verbose to 0, i.e., slient (no progress bar)
validation_loss, validation_accuracy = model.evaluate(x_test, y_test, verbose=0)
# Print loss and accuracy
print('Validation loss:', validation_loss)
print('Validation accuracy:', validation_accuracy)
```

Output

```
Validation loss: 0.07229845970869064
Validation accuracy: 0.9776999950408936
```



7. Save the Model

- Save the entire model to an HDFS (Hadoop Distributed File System) file.
- The .h5 extension of the file indicates that the model should be saved in Keras format as an HDFS file.

```
model_name = 'digits_recognition_cnn.h5'
model.save(model_name, save_format='h5')

loaded_model = load_model(model_name)
```

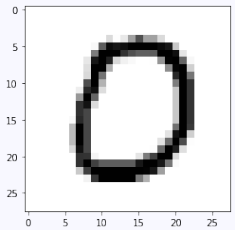
8. Use the Model

- To use the model, we call predict() function.

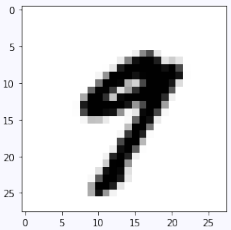
```
image_index = (int)(input("Enter an image index: ")) # Get image index from user
# Show the image in greyscale
plt.imshow(x_test[image_index].reshape(28,28),cmap='Greys')
# Use the model to do prediction by specifying the image.
# Get back a numpy array of prediction
prediction = loaded_model.predict(x_test[image_index].reshape(1,28,28,1))
# Print the predicted result, i.e., the one with maximum value
print('Predicted result:', prediction.argmax())
```

Output

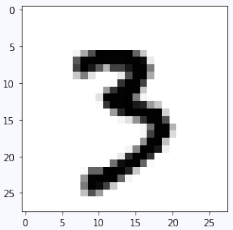
```
Enter an image index: 10
Predicted result: 0
```



```
Enter an image index: 20
Predicted result: 9
```



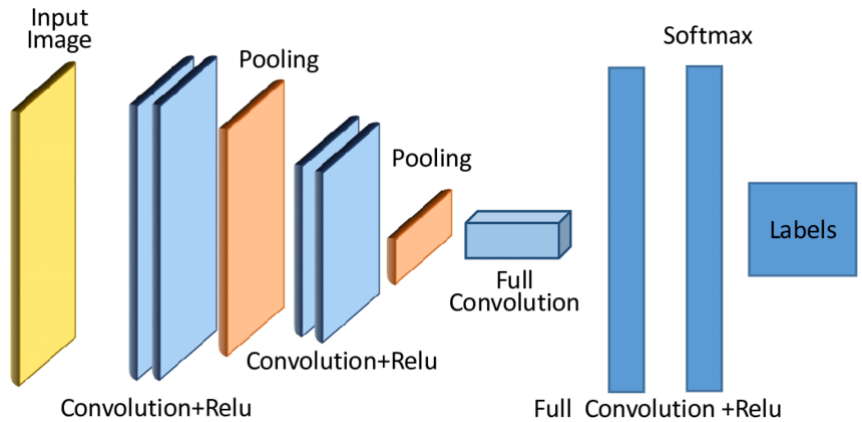
```
Enter an image index: 30
Predicted result: 3
```



Typical CNN Architecture

Typical CNN Architecture

- Typical CNN models for classification tasks stack different types of layers as follows:
  - A few convolutional layers (usually with ReLU activation)
  - Then, a pooling layer
  - Then, a few convolutional layers (usually with ReLU activation) and a pooling layer, and so on
  - Then, one or more fully connected layers (usually with ReLU activation)
  - Then, a final fully connected softmax layer



Typical CNN Architecture

- Generally, the convolutional and pooling layers are for feature learning, and the fully-connected layers are for classification.
- Doing through the convolutional layers, the feature maps typically get smaller but deeper (i.e. more feature maps per layer).
- The fully connected layers form an ordinary feedforward neural network to which the transformed inputs are fed.
- Instead of using a convolutional layer with a large kernel size (e.g. 9×9 with 81 parameters), it is better to stack two convolutional layers with smaller kernels (e.g. 3×3) with 18 parameters for the two layers together.

Practice Problem

- Given a 5×5 input image

5	1	3	7	8
2	4	6	1	7
3	4	9	7	2
4	9	4	5	9
7	9	8	7	9

  and the 3×3 kernel

3	0	0
0	0	0
0	0	-3

- Estimate the output of convolving the input image with the kernel. Assume zero-padding with 1 pixel border is performed.
  (Remark: Don't forget to flip the kernel.)

Answer:

12	18	3	21	0
12	12	18	-3	-21
27	6	3	9	-3
27	15	9	0	-21
0	-12	-27	-12	-15

Practice Problem

- Suppose we have a Convolutional Neural Network (CNN):
  - Input layer: An input color image of size $128 \times 128 \times 3$.
  - First convolutional layer: 5 different kernels, each of size $3 \times 3 \times 3$ (no zero padding and stride of size 5).
  - First max-pooling layer: Size 2×2.
  - Second convolutional layer: 3 different kernels, each of size $5 \times 5 \times 5$ (zero padding of size 2 and stride of size 2).
  - ...

1. What is the dimension of the output of the first convolutional layer, and how many weights must be learned for this layer.
2. What is the main purpose of doing pooling during training in a deep neural network? Also, what is the dimension of the output of the first pooling layer?
3. What is the dimension of the output of the second convolutional layer, and how many weights must be learned for this layer?

Answer:

1. The dimension of the output of the first convolutional layer is $26 \times 26 \times 5$. As each unit in the convolutional layer is connected to $5 \times 3 \times 3 \times 3 = 135$ units, these weights are shared across all units in this layer. So, the number of weights to be learned is 135.
2. The main purpose of doing pooling during training is to make sure that the subsequent layers of the CNN can pick up larger-scale detail. The dimension of the output of the pooling layer is $13 \times 13 \times 5$.
3. The dimension of the output of the second convolutional layer is $7 \times 7 \times 3$. As each unit in the convolutional layer is connected to $3 \times 5 \times 5 \times 5 = 375$ units, these weights are shared across all units in this layer. So, the number of weights to be learned is 375.

That's all!

Any question?

Welcome Back!

Acknowledgments

- The lecture notes are developed based on Dr. Desmond Tsoi's lecture slides.