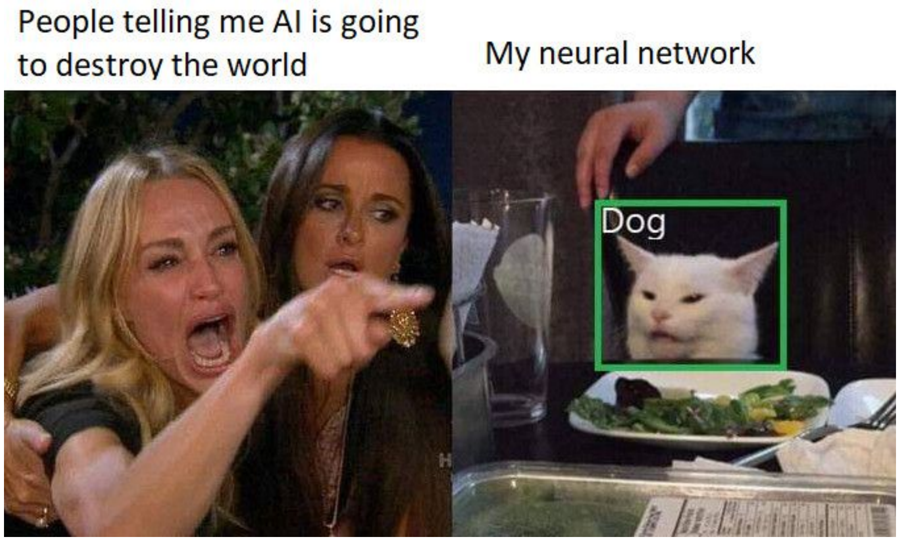
This is CS4048!

GCR:wzj3vua

Convolution Neural Networks

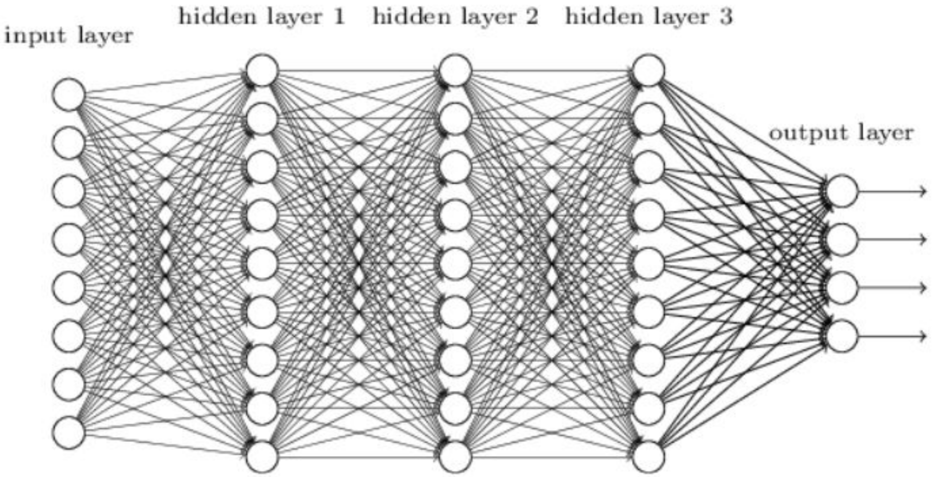


Neural network with Images

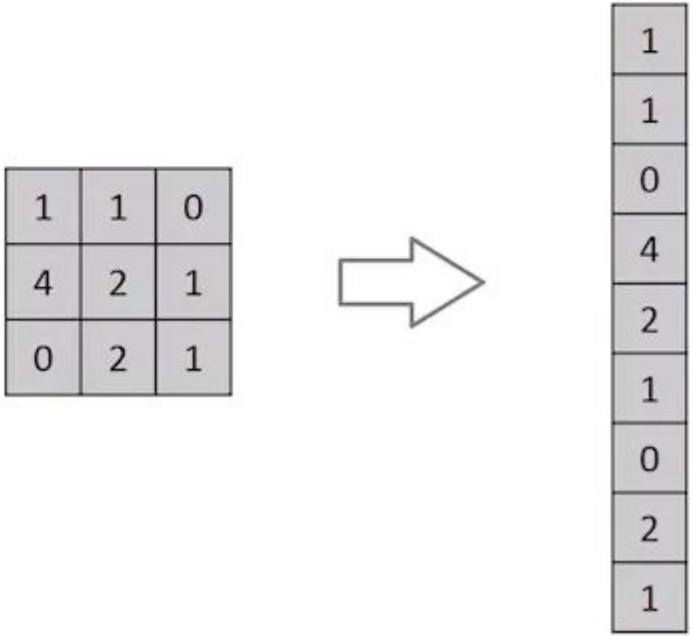
We know it is good to learn a small model.

From this fully connected model, do we really need all the edges?

Can some of these be shared?



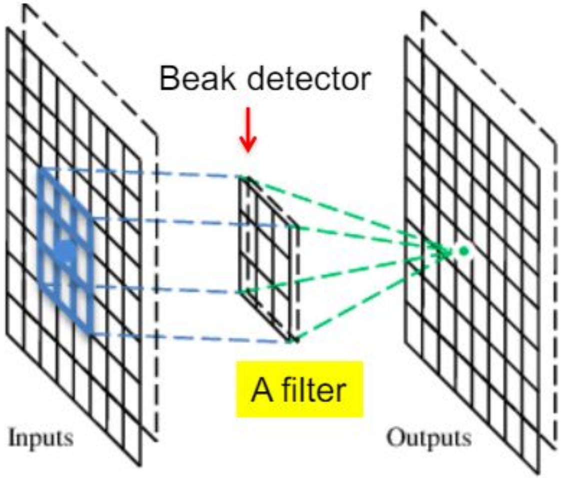
An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh.. not really.

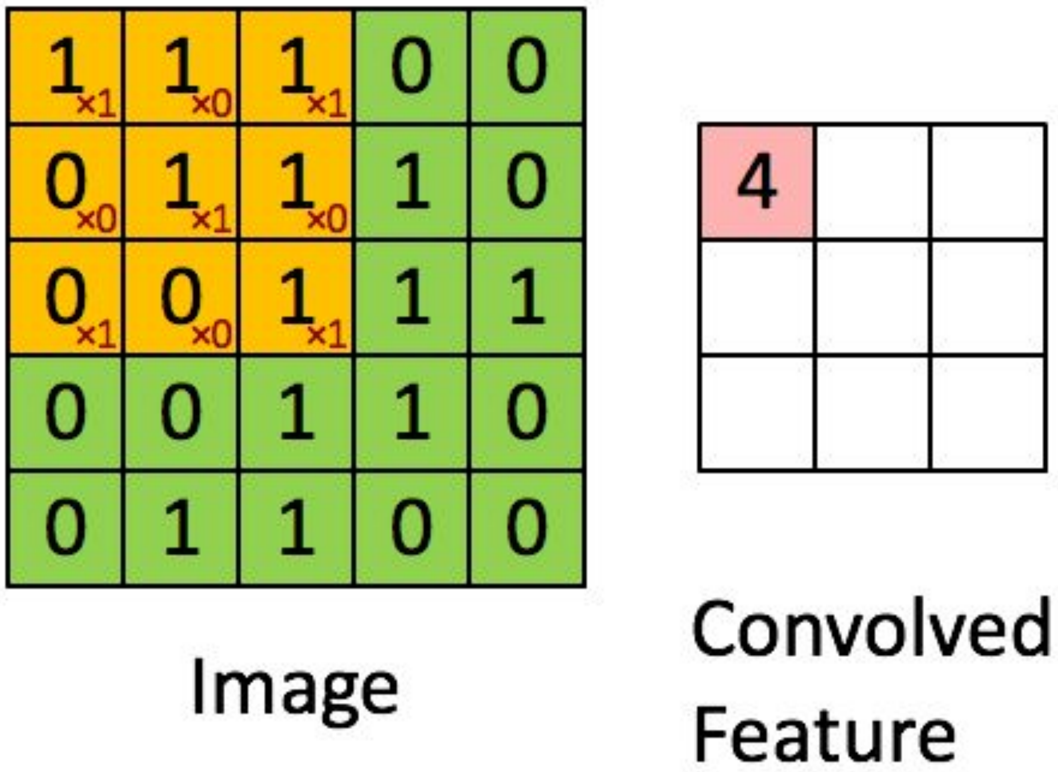


To capture Spatial/temporal dependencies.

CNN

A CNN is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that does convolutional operation.





1 ×1 1 ×0 1 ×1 0 0
0 ×0 1 ×1 1 ×0 1 0
0 ×1 0 ×0 1 ×1 1 1
0 0 1 1 0
0 1 1 0 0
Image
4
Convolved Feature

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

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

⋮ ⋮

Each filter detects a small pattern (3 x 3).

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

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

Dot product

3 -1

6 x 6 image

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

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

6 x 6 image

3 -3

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

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

6 x 6 image

$\boxed{3}$	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
$\boxed{3}$	-2	-2	-1

Convolution

-1	1	-1
-1	1	-1
-1	1	-1

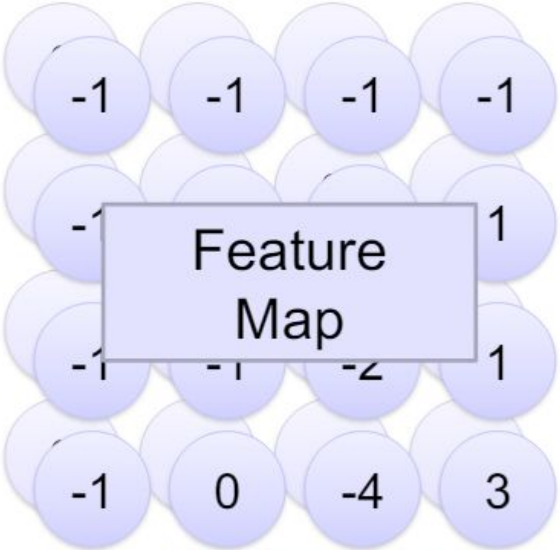
Filter 2

stride=1

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

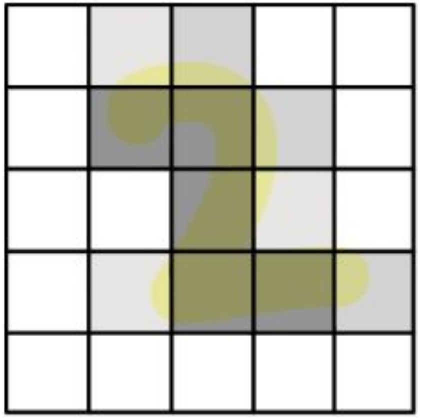
6 x 6 image

Repeat this for each filter



Two 4 x 4 images
Forming 2 x 4 x 4 matrix

For Grey Images



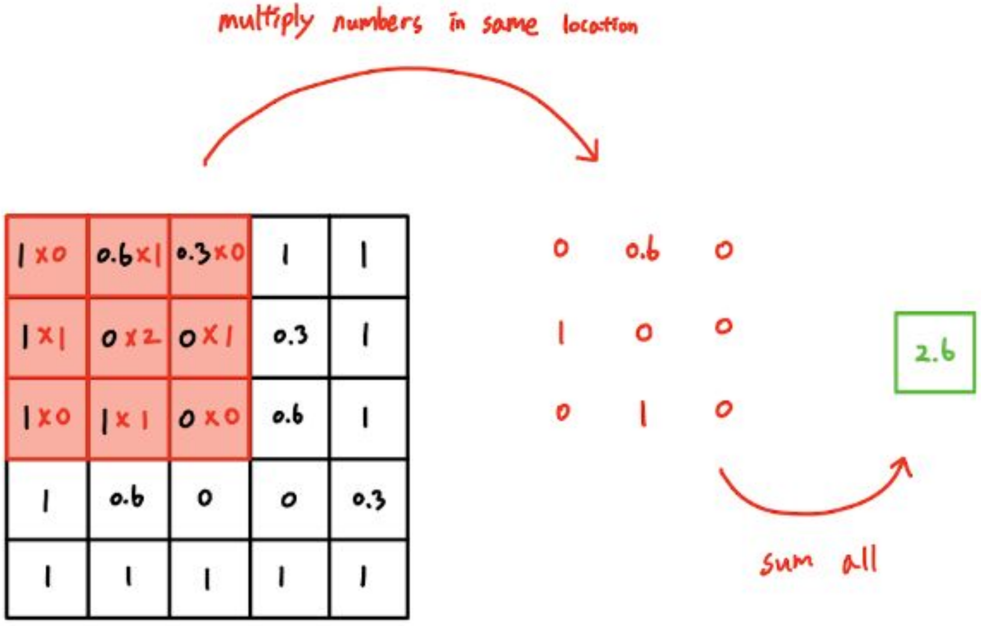
1	0.6	0.3	1	1
1	0	0	0.3	1
1	1	0	0.6	1
1	0.6	0	0	0.3
1	1	1	1	1

For Grey Images

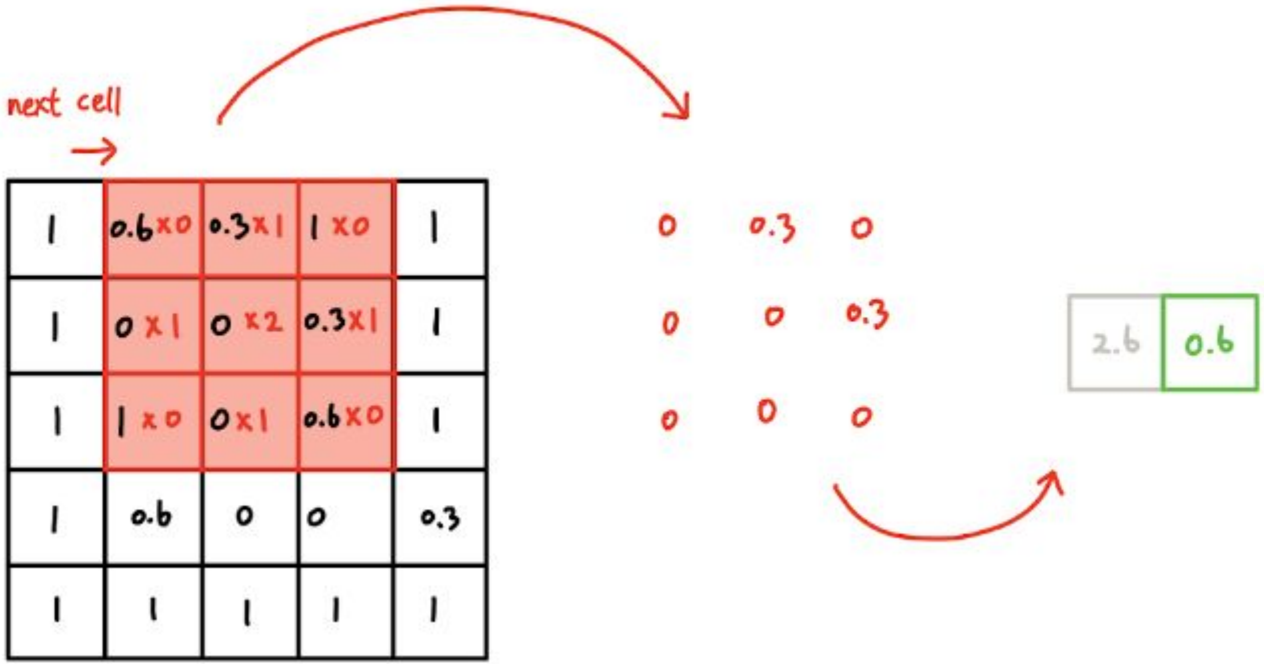
1	0.6	0.3	1	1
1	0	0	0.3	1
1	1	0	0.6	1
1	0.6	0	0	0.3
1	1	1	1	1

0	1	0
1	2	1
0	1	0

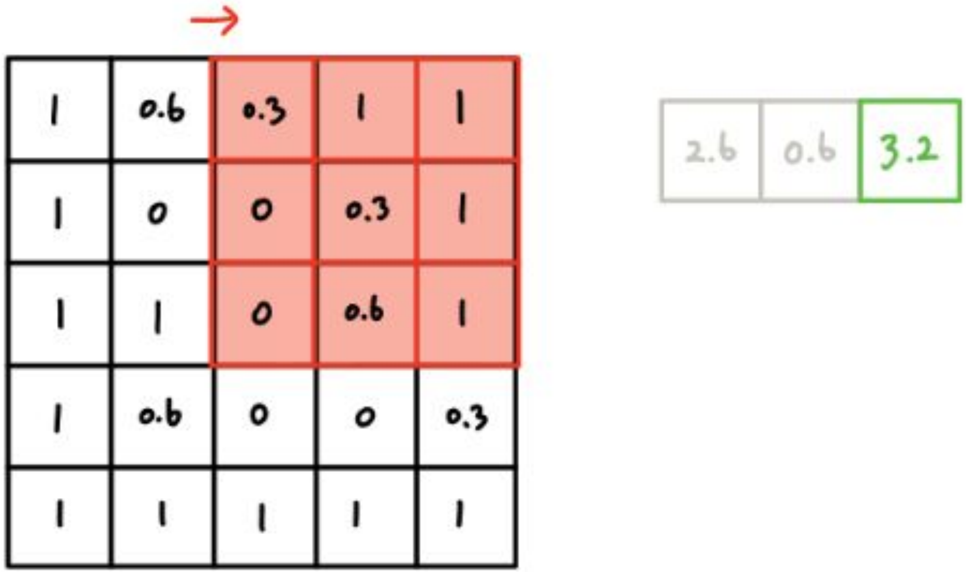
For Grey Images



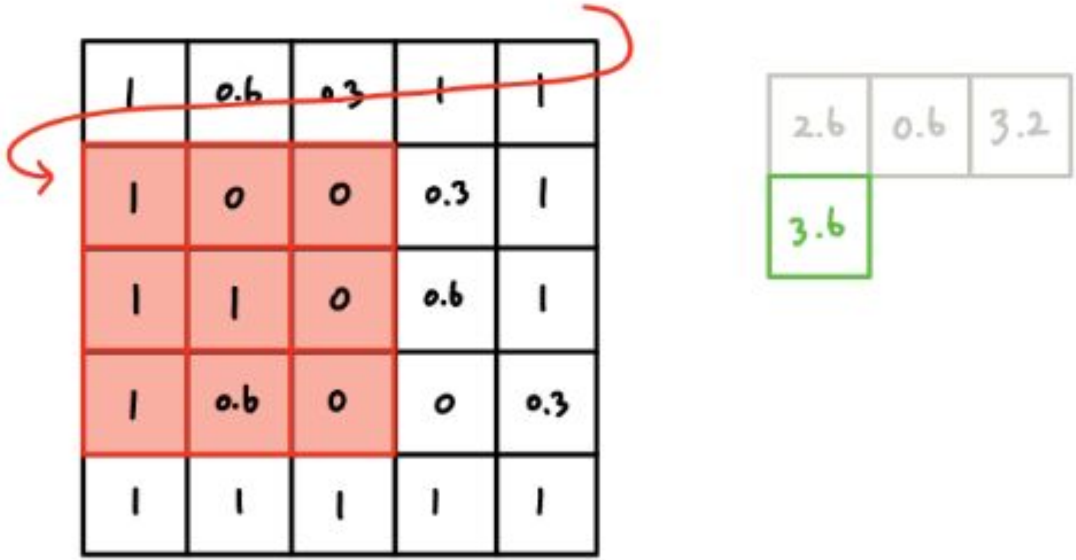
For Grey Images



For Grey Images

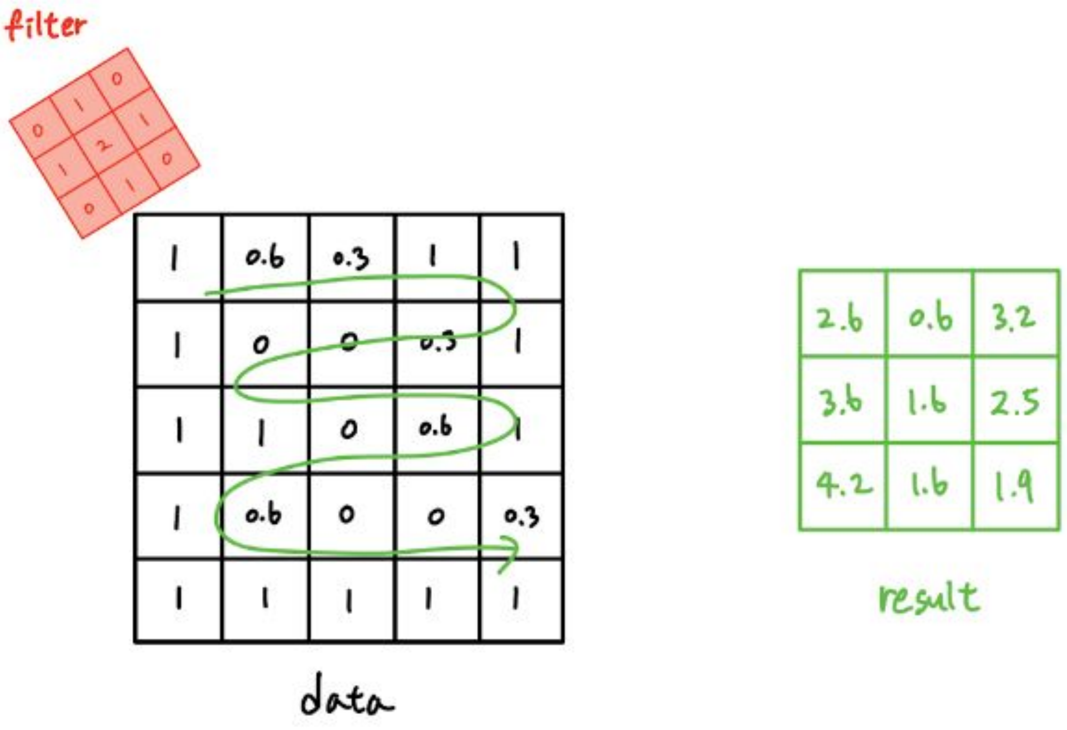


For Grey Images

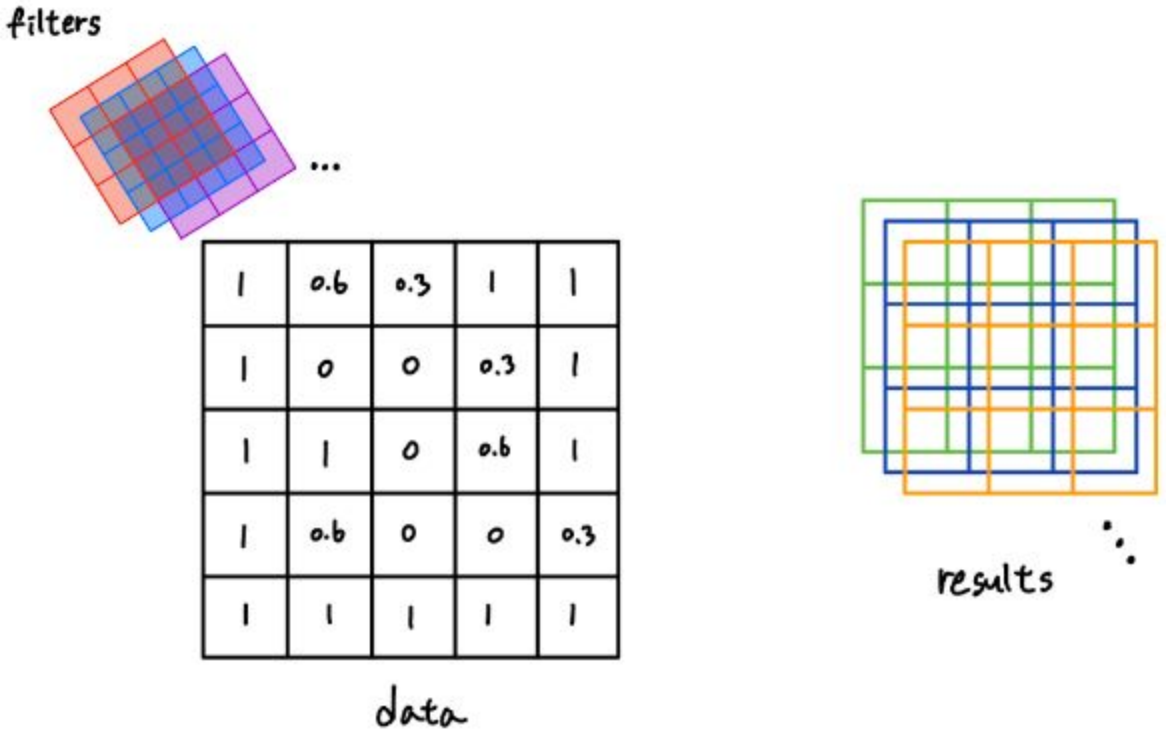


For Grey Images

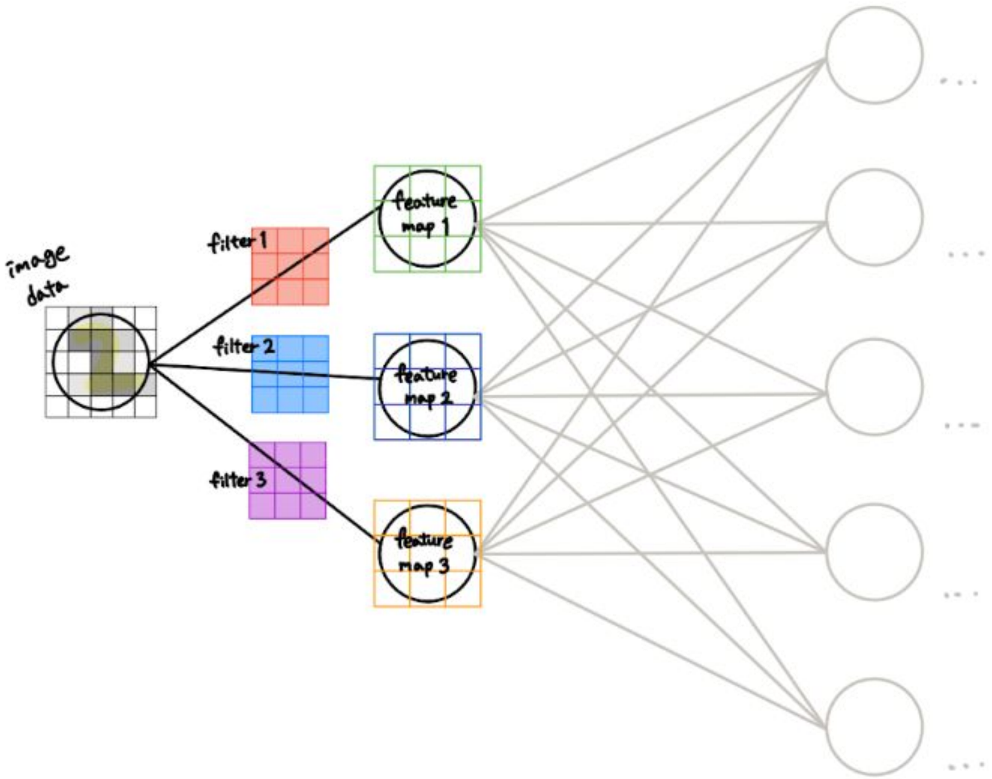
This sliding operation is called **convolution**.*

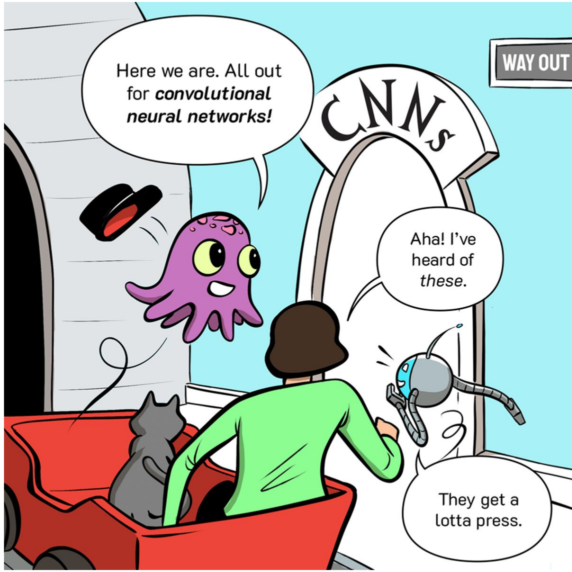


For Grey Images



For Grey Images

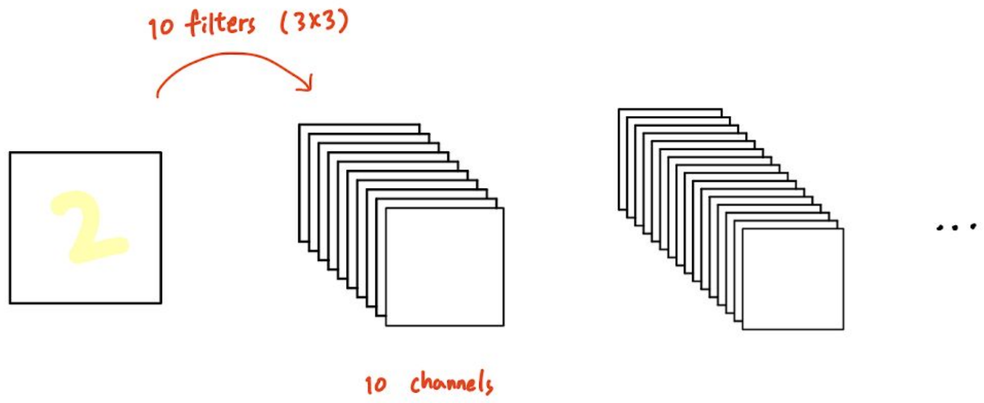




CNN

Result array from a convolution operation using a filter is called a feature map.

The number of feature maps is called the number of channels as well. If you use 10 filters, then there will be 10 output channels.



Color image: RGB 3 channels

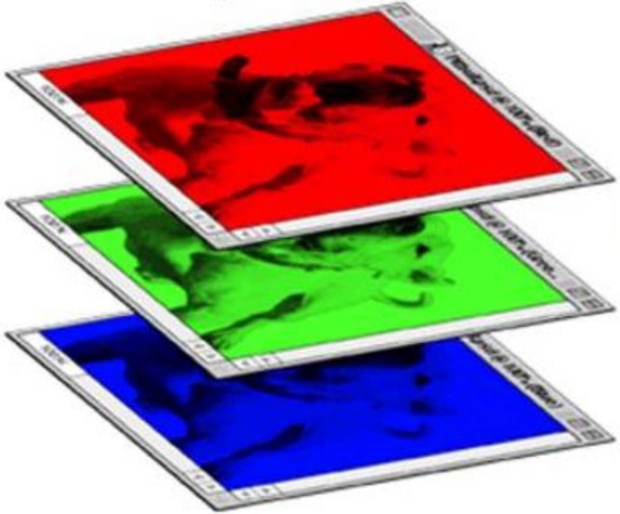
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

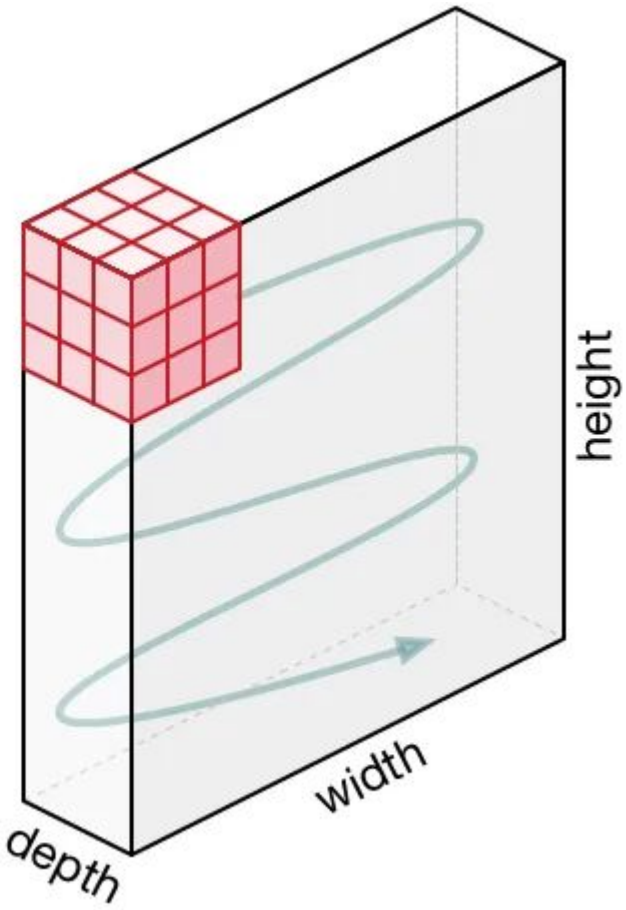
Color image

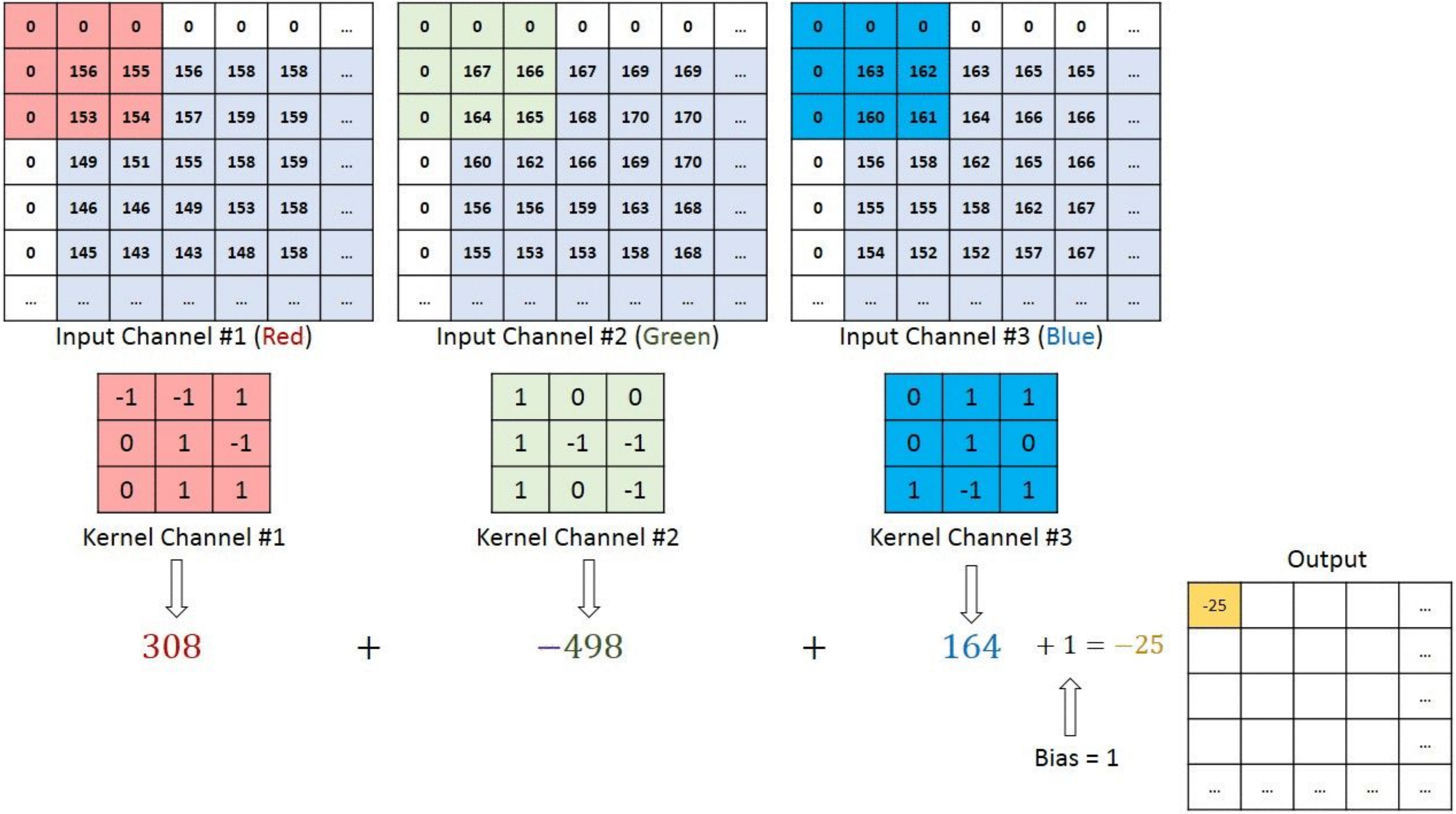


1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

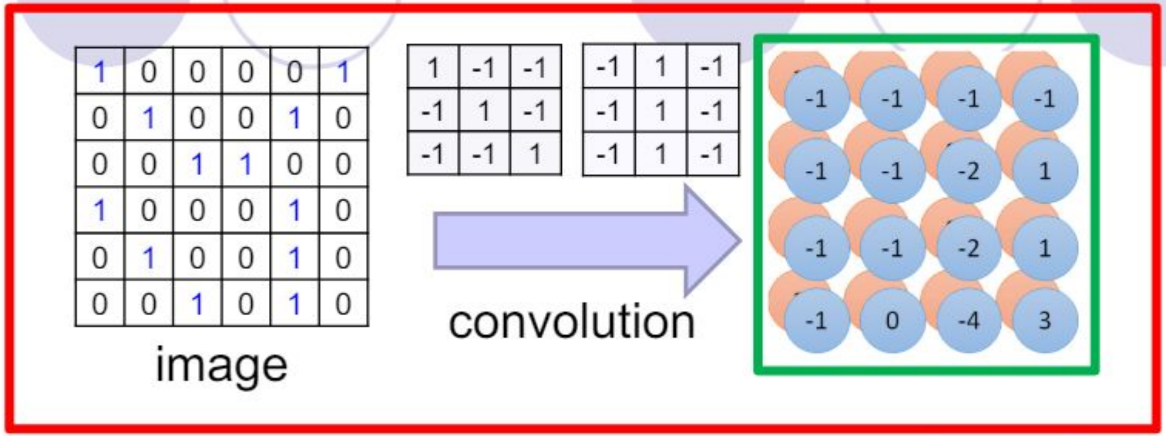
CNN with RGB Images

The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.

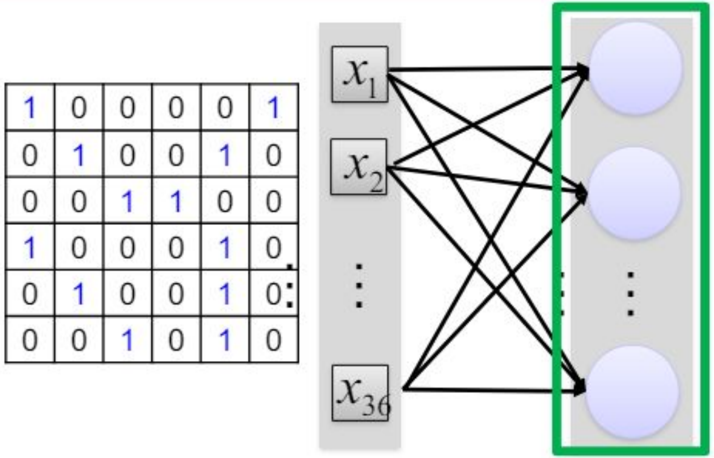



Input Channel #1 (Red)
Input Channel #2 (Green)
Input Channel #3 (Blue)
Kernel Channel #1
Kernel Channel #2
Kernel Channel #3
308
+
−498
+
164
+ 1 = −25
Bias = 1
Output
-25

Convolution v.s. Fully Connected



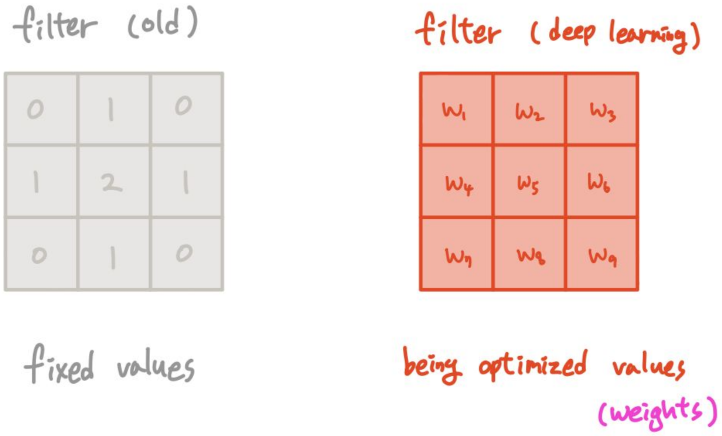
Fully-connected

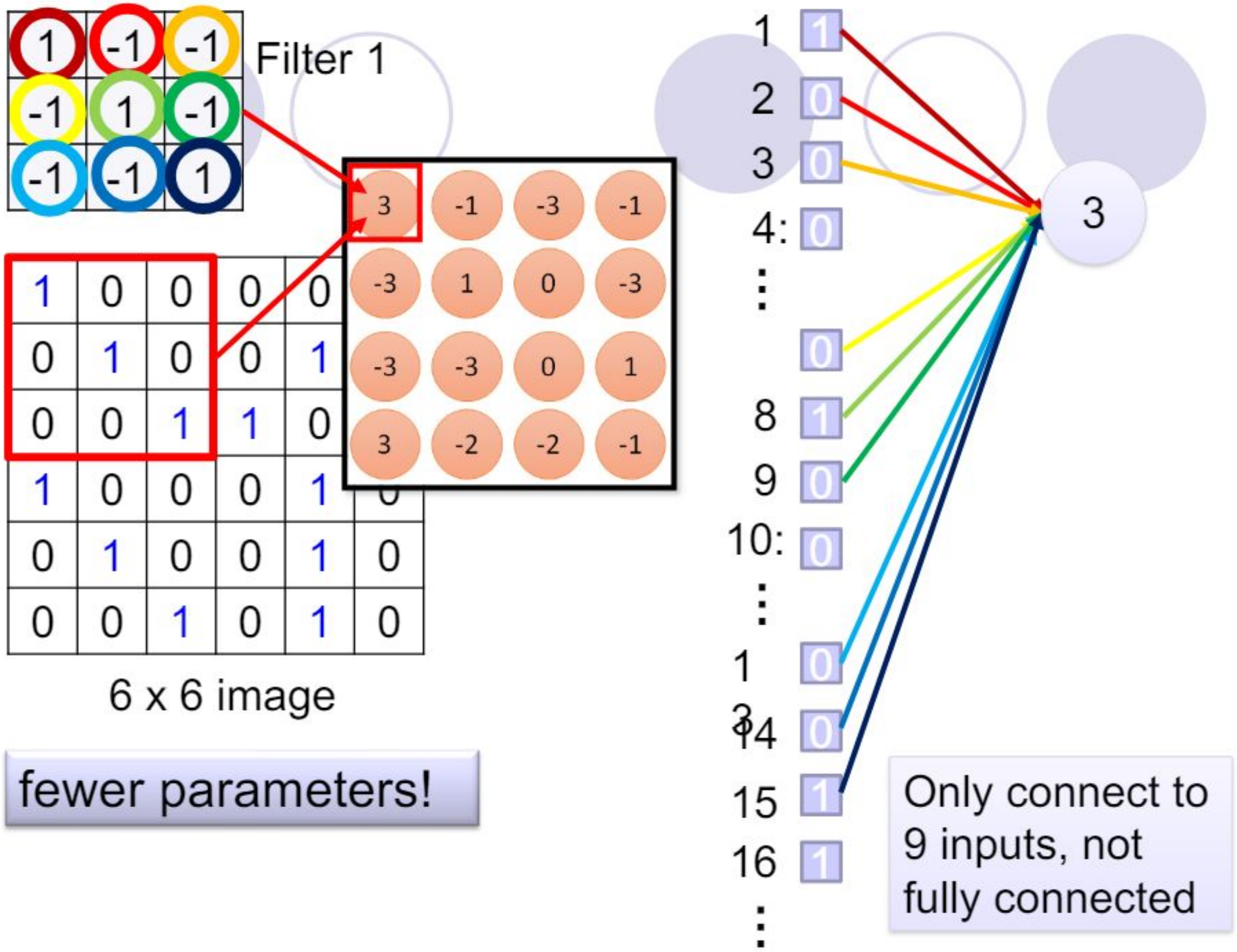


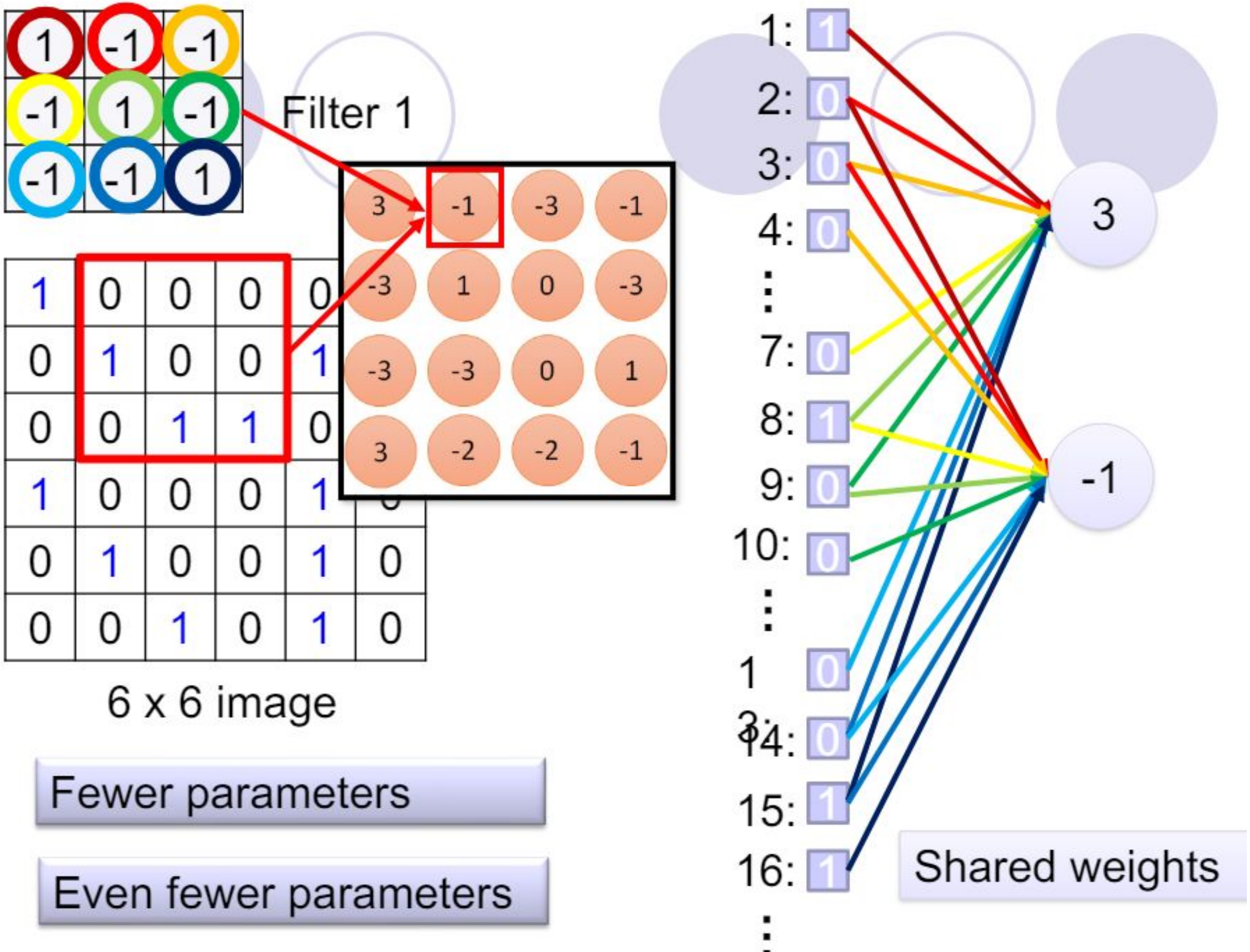
So what kind of values should be in the filters?

Before the era of deep learning, the values in the filter were fixed. Have you ever used "sharpen filter" in a photo editor?

The values in a convolution filter are the weights of a Convolutional Neural Network.




Filter 1
6 x 6 image
fewer parameters!
Only connect to 9 inputs, not fully connected



1 -1 -1
-1 1 -1
-1 -1 1
Filter 1
3 -1 -3 -1
-3 1 0 -3
-3 -3 0 1
3 -2 -2 -1
1 0 0 0 0
0 1 0 0 1
0 0 1 1 0
1 0 0 0 1 0
0 1 0 0 1 0
0 0 1 0 1 0
6 x 6 image
Fewer parameters
Even fewer parameters
1: 1
2: 0
3: 0
4: 0
7: 0
8: 1
9: 0
10: 0
13: 0
14: 0
15: 1
16: 1
3
-1
Shared weights

CNN

That means, we put some random values into the filters and set how many filters we use in each layer before starting training, but the values in the filters change as the training goes on.

By doing so, the values in the filters are getting optimized, more meaningful feature maps are being extracted, so the model makes better prediction.

CNN

The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image.

Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc.

With added layers, the architecture adapts to the High-Level features as well, giving us a network that has a wholesome understanding of images in the dataset, similar to how we would.

- Convolution layer takes an input feature map of dimension $W \times H \times N$ and produces an output feature map of dimension $\widehat{W} \times \widehat{H} \times M$
- Each layer is defined using following parameters:
  - # Input channels (N)
  - # Output channels (M)
  - Kernel size
  - Padding
  - Stride

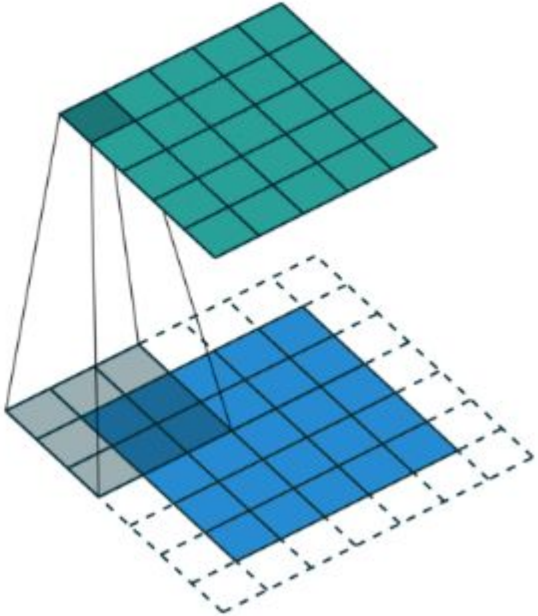


Figure: In this example, 5x5 input is convolved with 3x3 kernel with stride=padding=1 to produce an output of size 5x5.

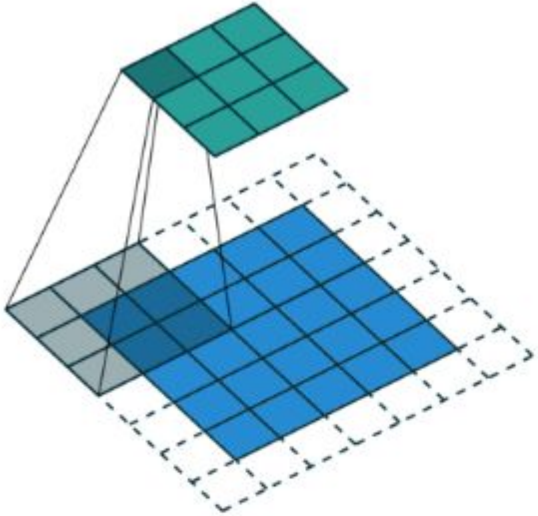
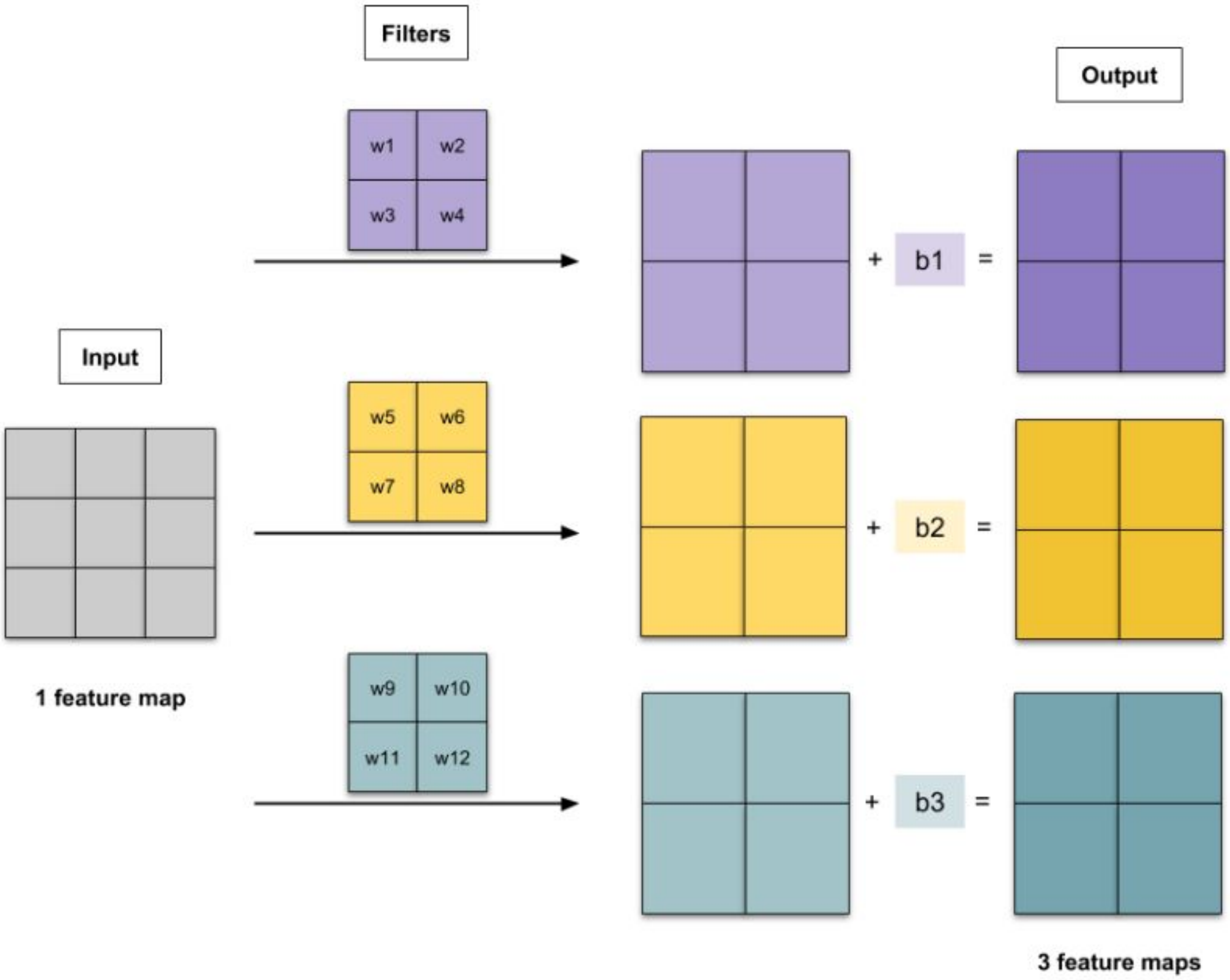
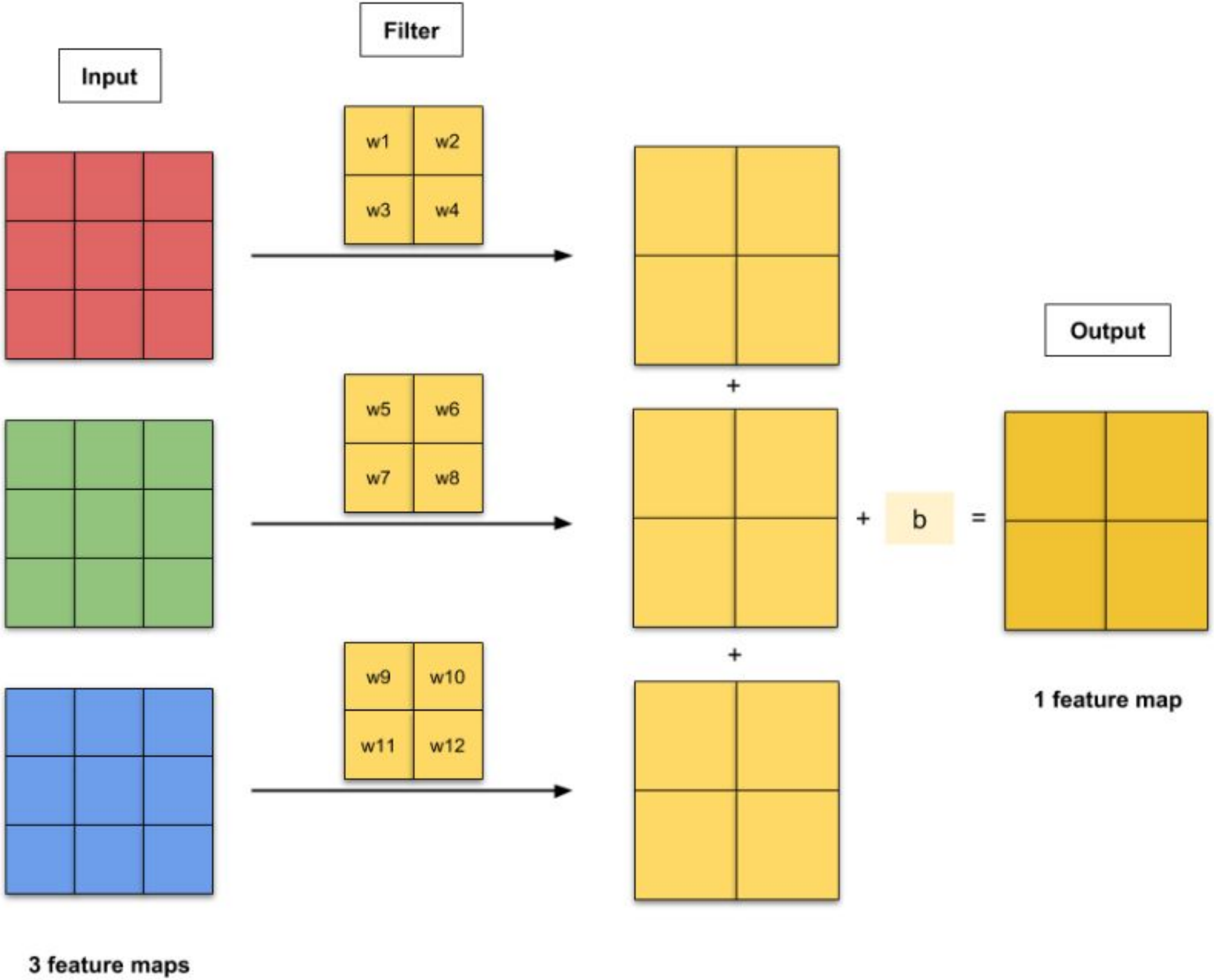
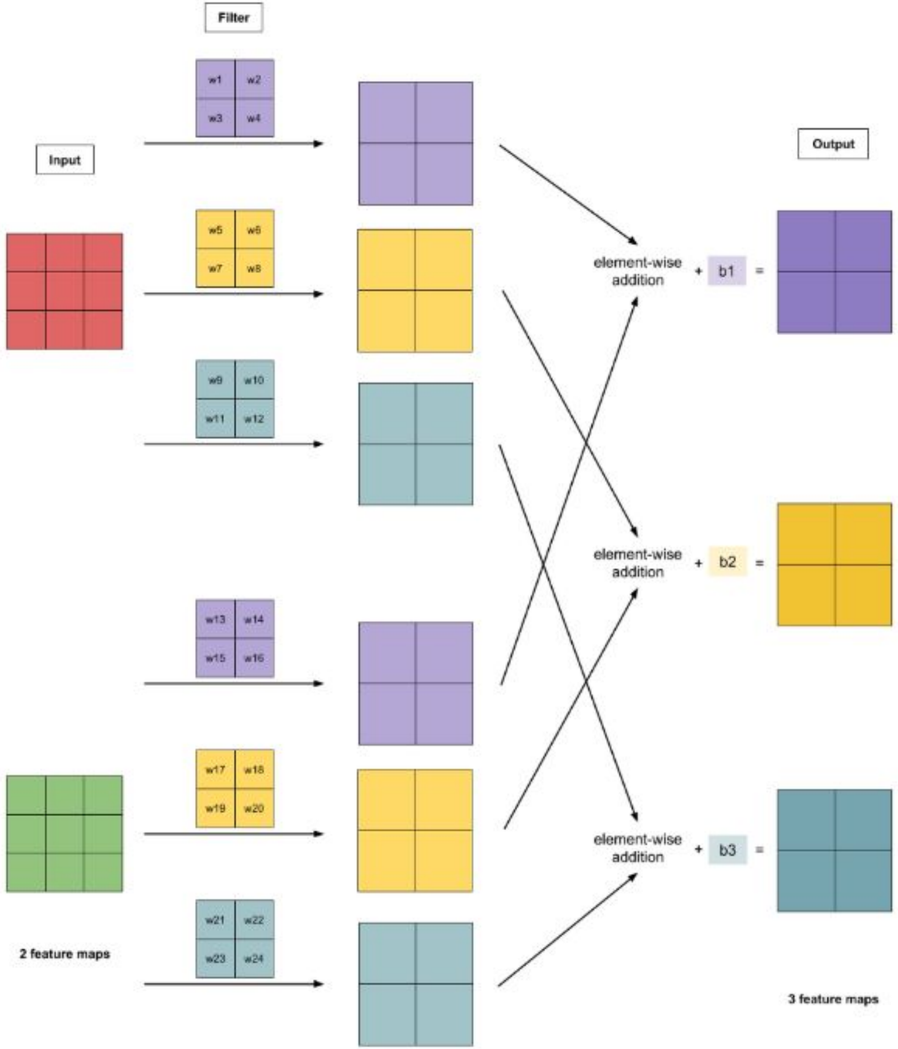


Figure: In this example, 5x5 input is convolved with 3x3 kernel with **stride=2** and padding=1 to produce an output of size 3x3.

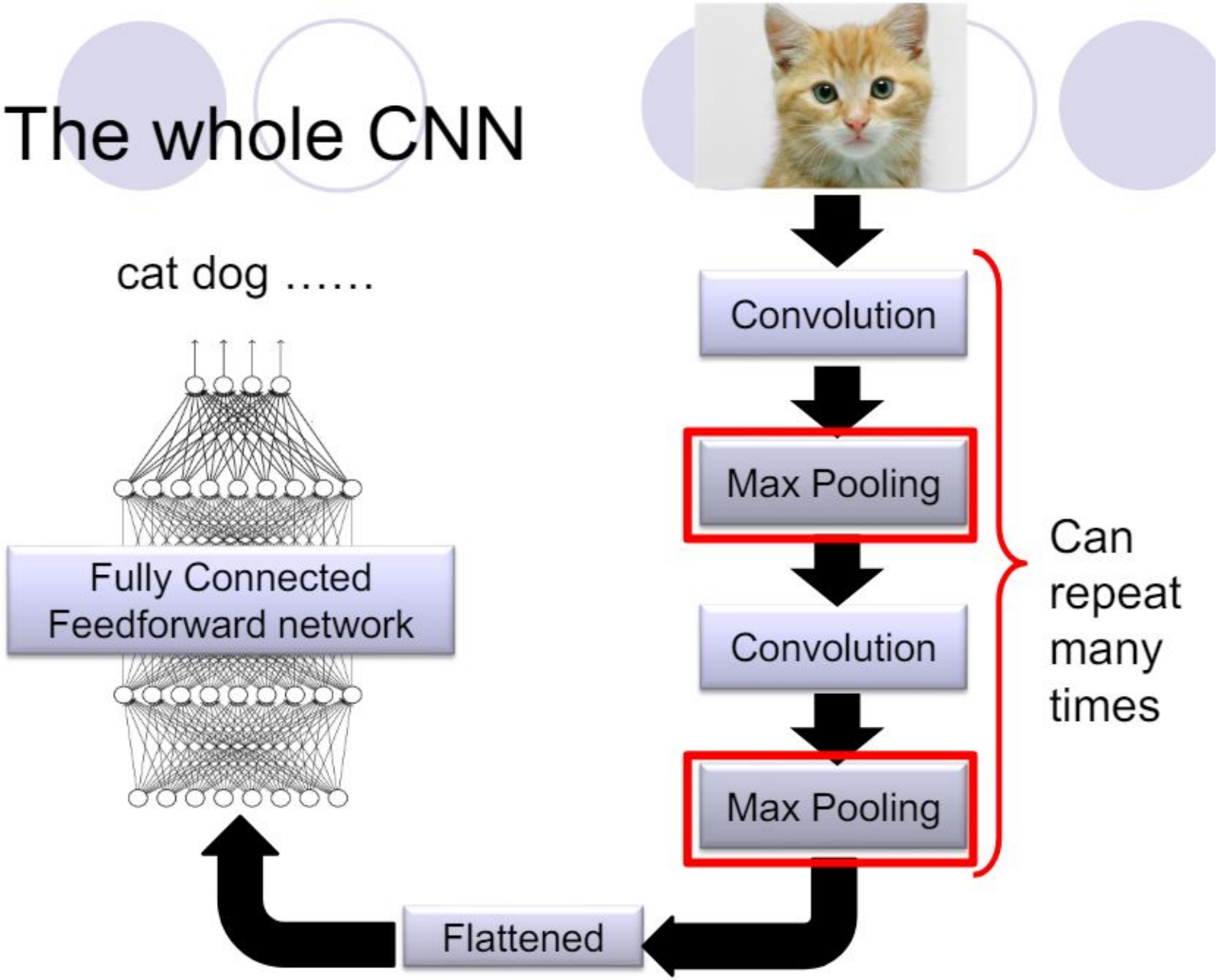

Filters
Output
Input
w1
w2
w3
w4
+
b1
=
w5
w6
w7
w8
+
b2
=
1 feature map
w9
w10
w11
w12
+
b3
=
3 feature maps


Input
Filter
w1
w2
w3
w4
w5
w6
w7
w8
w9
w10
w11
w12
+
+
+
b
=
Output
1 feature map
3 feature maps



Filter
Input
Output
w1
w2
w3
w4
w5
w6
w7
w8
w9
w10
w11
w12
w13
w14
w15
w16
w17
w18
w19
w20
w21
w22
w23
w24
element-wise addition
+ b1 =
element-wise addition
+ b2 =
element-wise addition
+ b3 =
2 feature maps
3 feature maps

The whole CNN



Pooling Layers

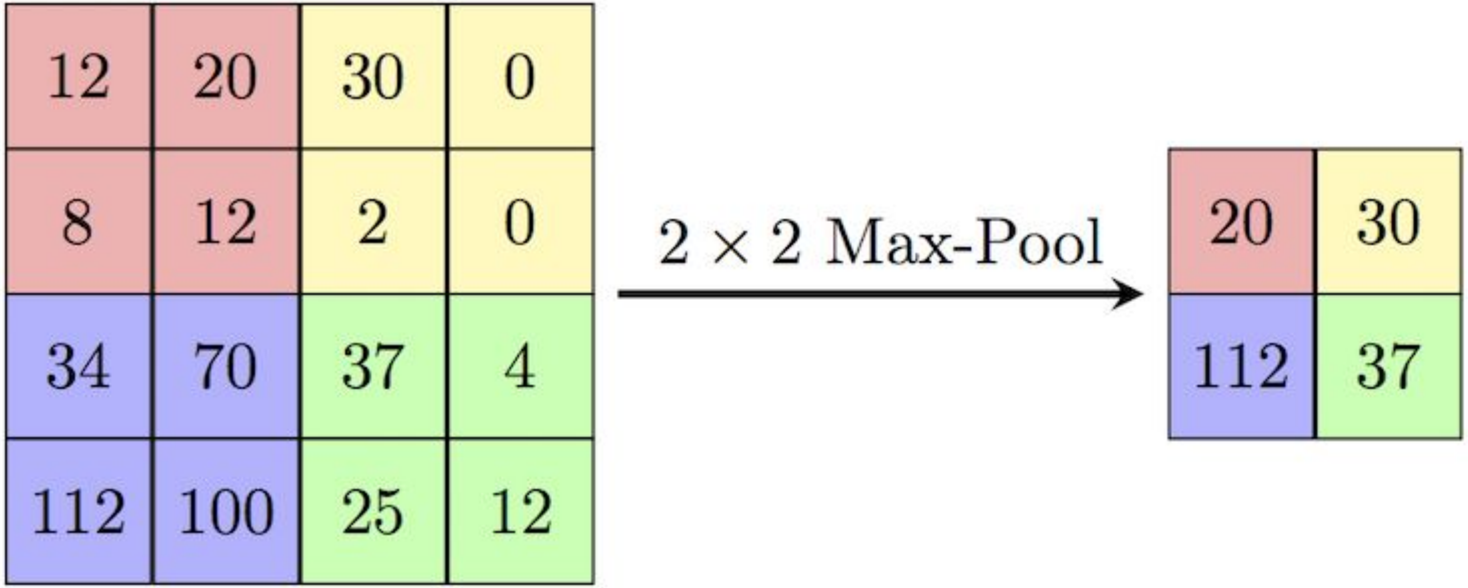
Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature.

Decreases the computational power required to process the data through dimensionality reduction

Extract dominant features.

Max - Min - Avg pooling

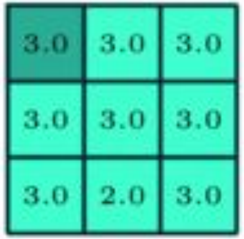
Max Pool



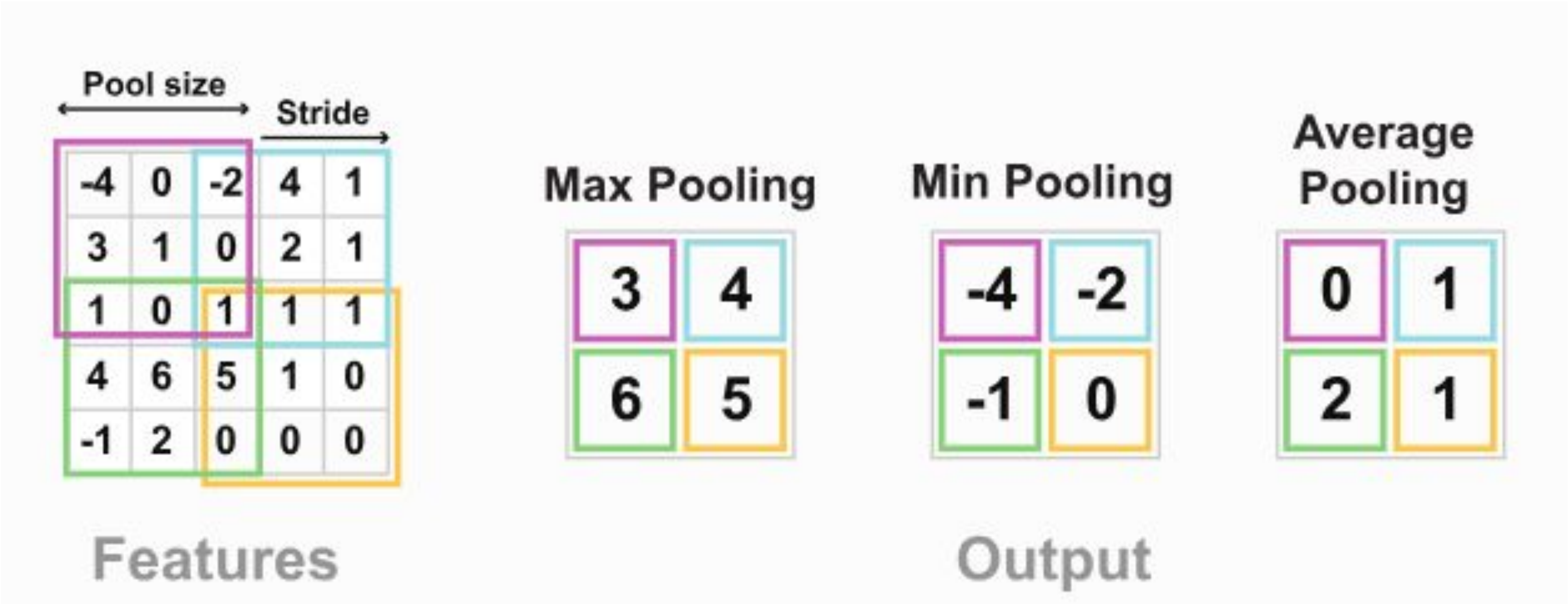
Max Pooling

Max Pooling also performs as a Noise Suppressant.

It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction.



Max Pooling



Pool size
Stride
-4 0 -2 4 1
3 1 0 2 1
1 0 1 1 1
4 6 5 1 0
-1 2 0 0 0
Max Pooling
3 4
6 5
Min Pooling
-4 -2
-1 0
Average Pooling
0 1
2 1
Features
Output

Why Pooling

- Subsampling pixels will not change the object

bird

Subsampling

bird

We can subsample the pixels to make image smaller

→ fewer parameters to characterize the image

CNN

A CNN compresses a fully connected network in following ways:

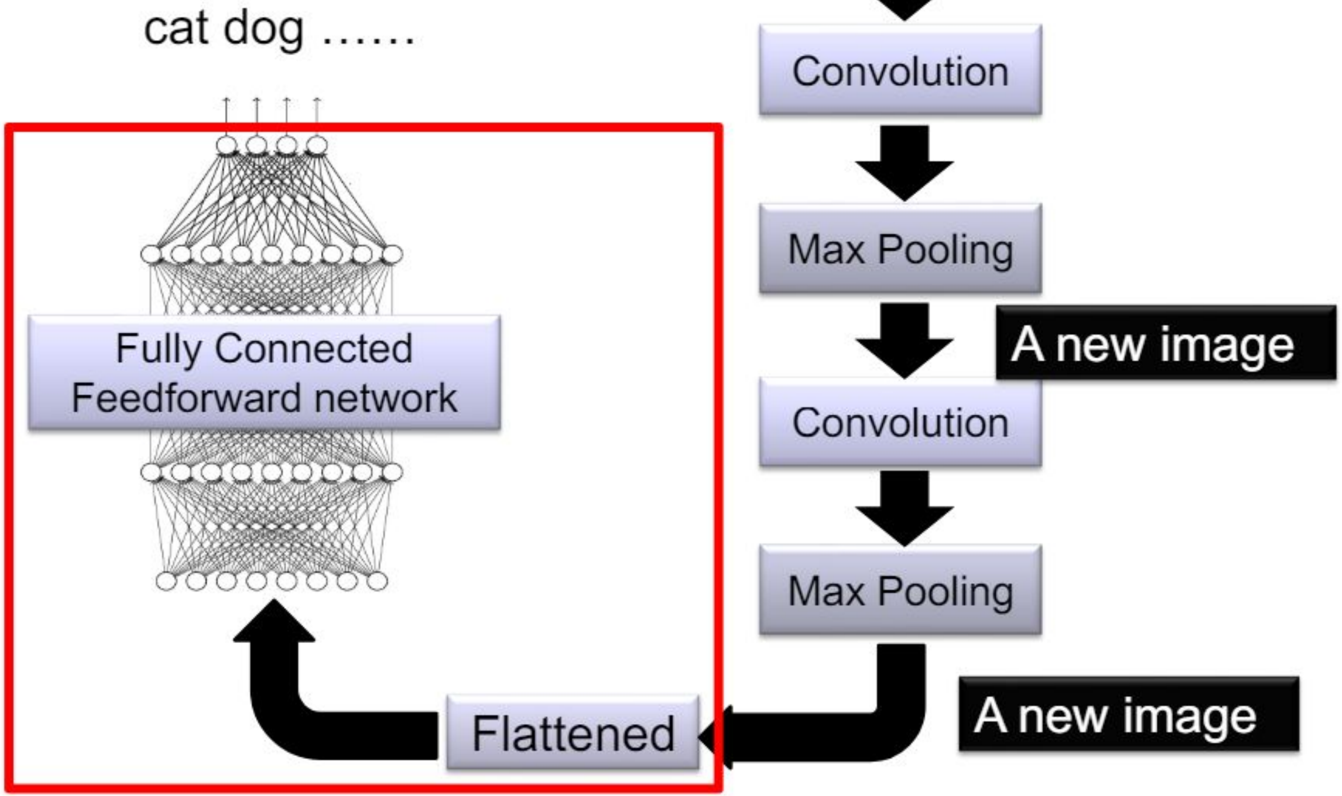
- Reducing number of connections
- Shared weights on the edges
- Max pooling further reduces the complexity

CNN

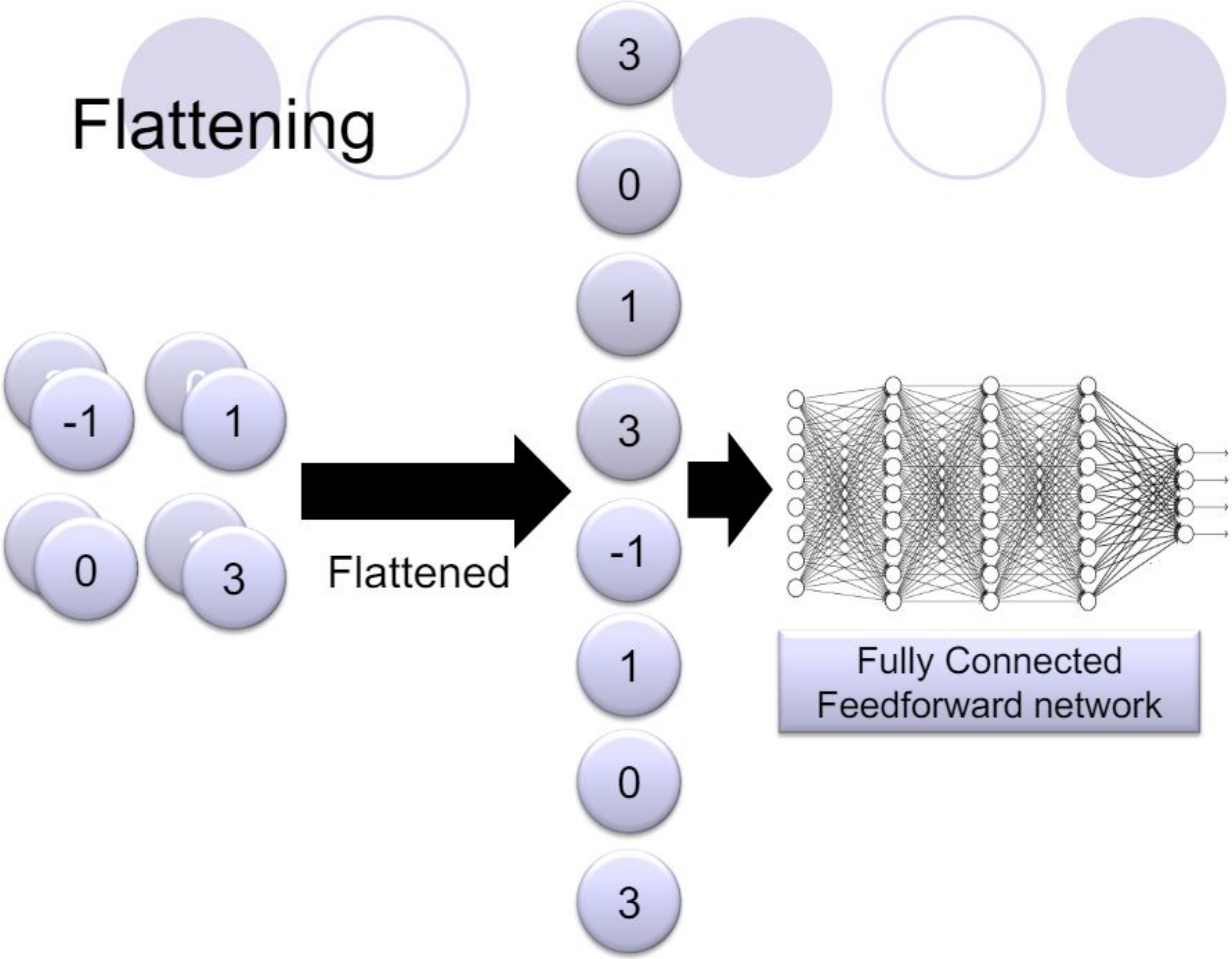
Convolutional Neural Networks is extension of traditional Multi-layer Perceptron, based on 3 ideas:

- Local receive fields
- Shared weights
- Spatial / temporal sub-sampling

The whole CNN

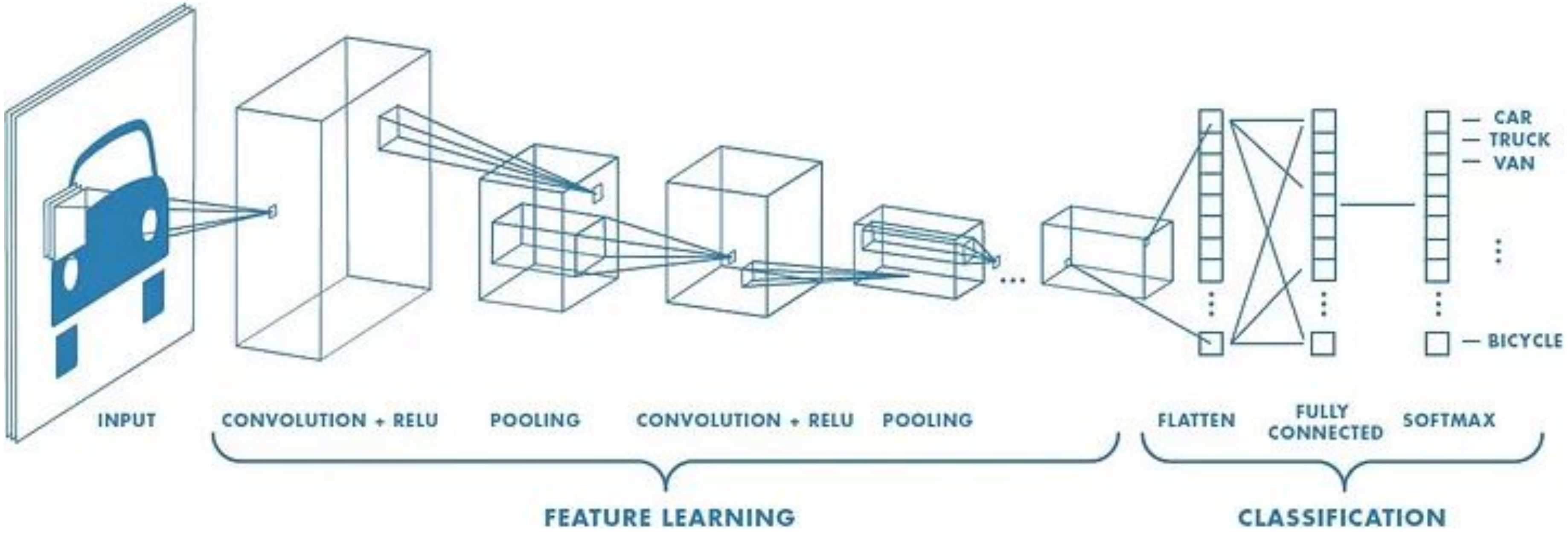


Flattening



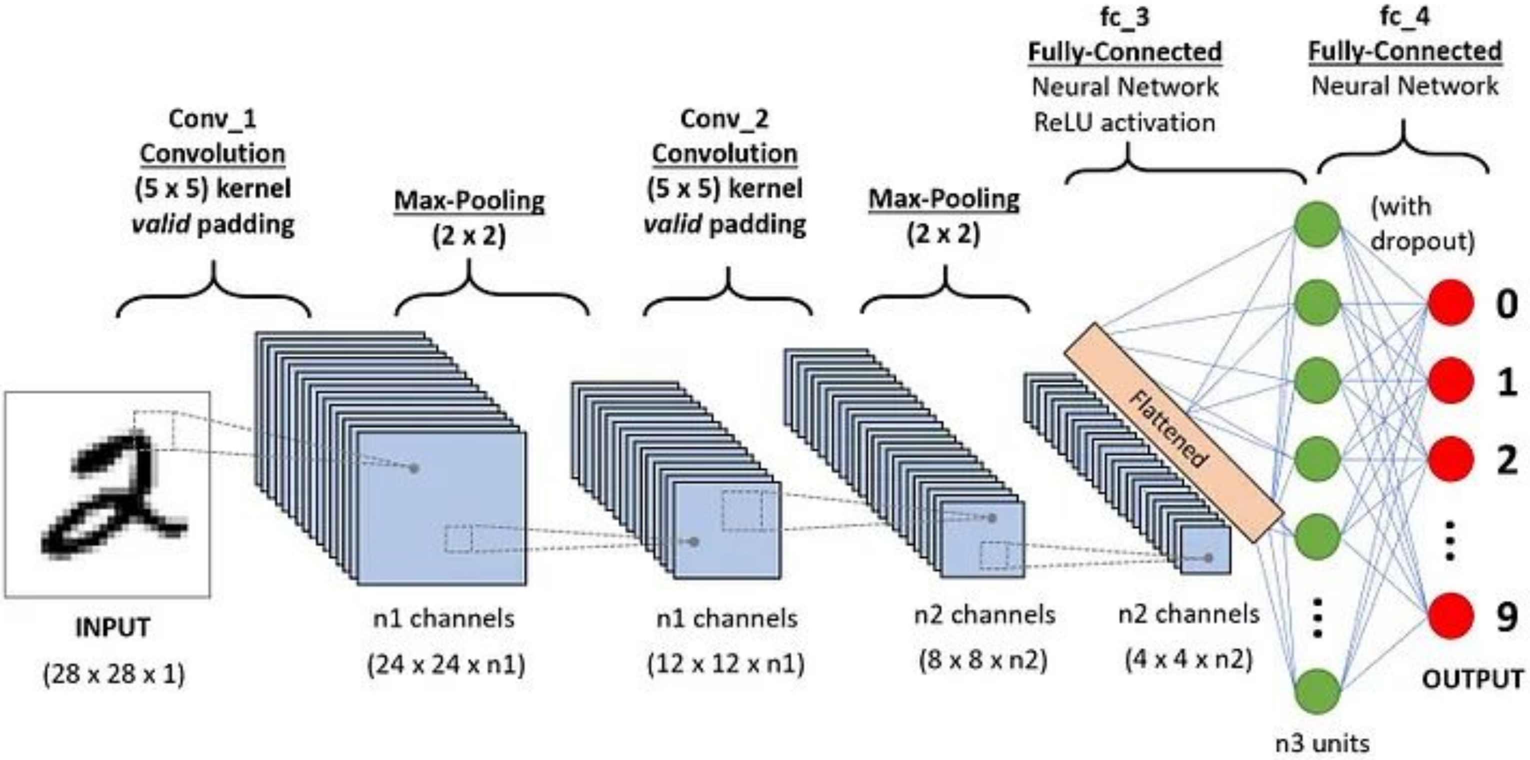
-1
1
0
3
Flattened
3
0
1
3
-1
1
0
3
Fully Connected
Feedforward network

CNN

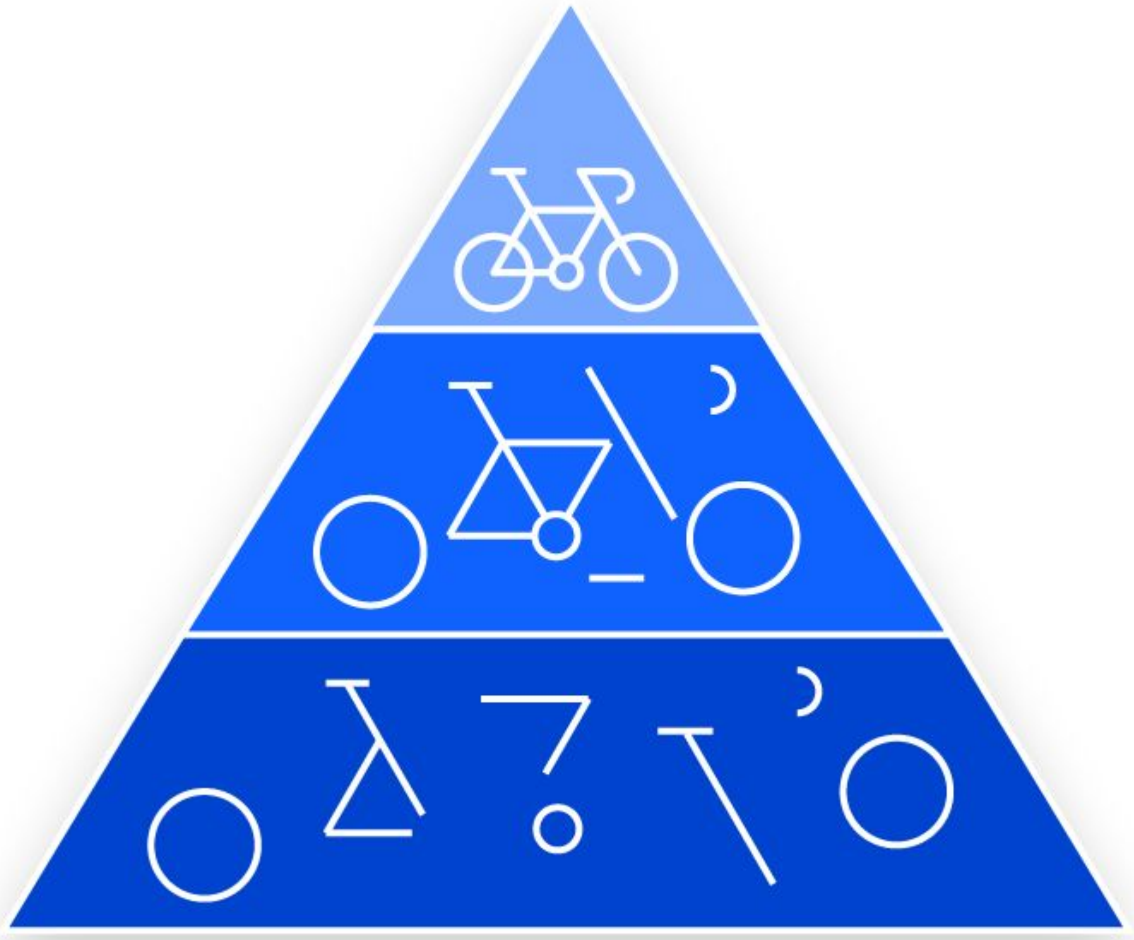


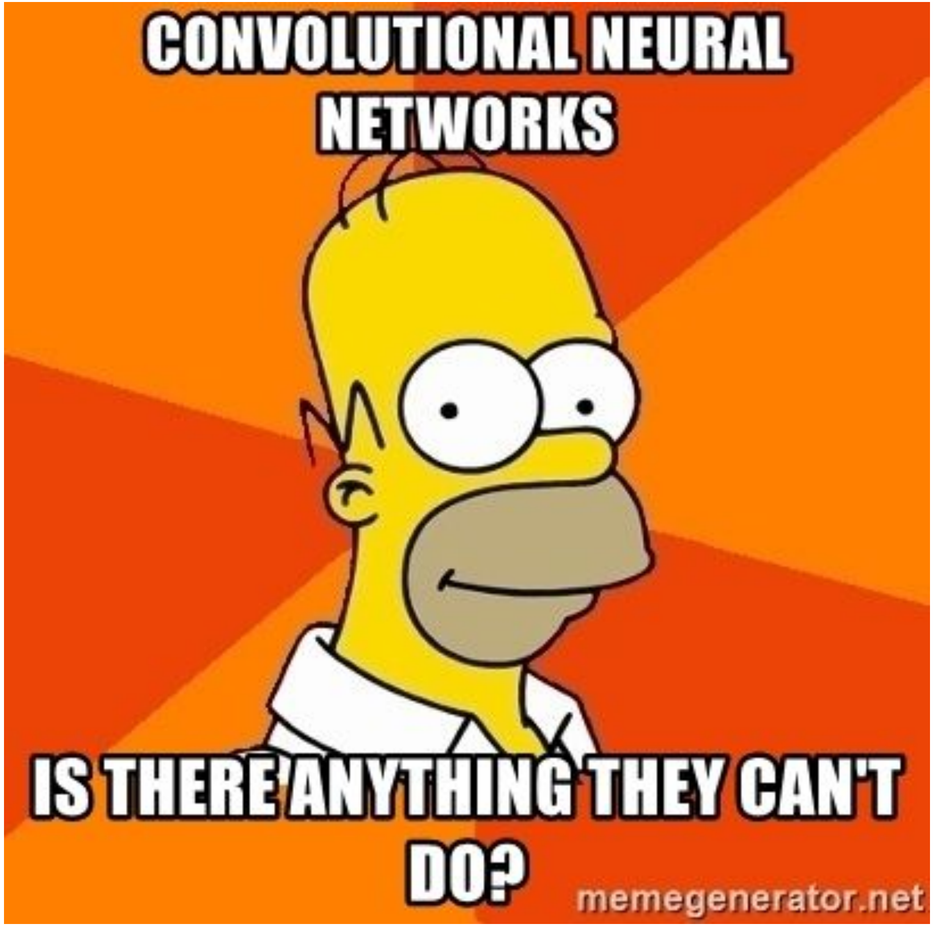
CAR
TRUCK
VAN
BICYCLE
INPUT
CONVOLUTION + RELU
POOLING
CONVOLUTION + RELU
POOLING
FLATTEN
FULLY CONNECTED
SOFTMAX
FEATURE LEARNING
CLASSIFICATION

CNN



fc_3
Fully-Connected
Neural Network
ReLU activation
fc_4
Fully-Connected
Neural Network
Conv_1
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
Conv_2
Convolution
(5 x 5) kernel
valid padding
Max-Pooling
(2 x 2)
(with
dropout)
0
1
2
9
Flattened
INPUT
(28 x 28 x 1)
n1 channels
(24 x 24 x n1)
n1 channels
(12 x 12 x n1)
n2 channels
(8 x 8 x n2)
n2 channels
(4 x 4 x n2)
OUTPUT
n3 units




CONVOLUTIONAL NEURAL NETWORKS
IS THERE ANYTHING THEY CAN'T DO?
memegenerator.net

References

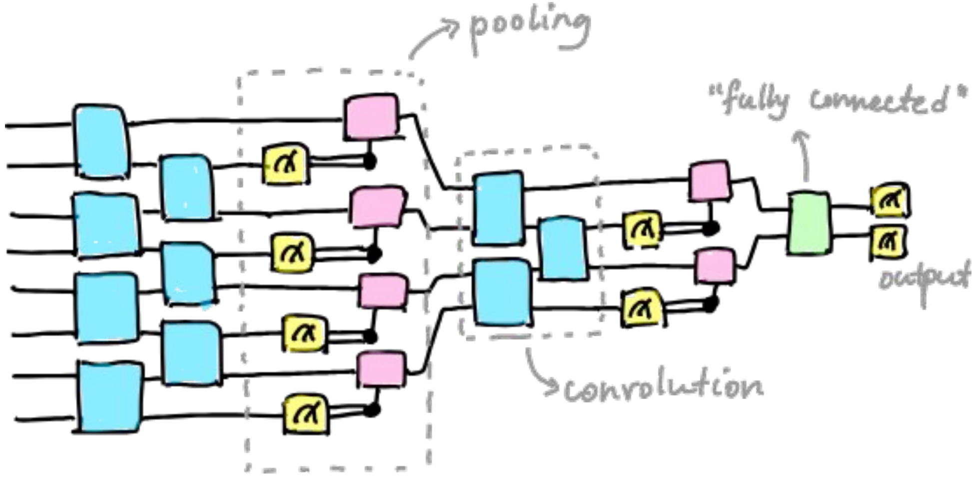
Some slides are taken from Dr Atif Tahir’s slides.

https://www.philgineer.com/2021/11/convolutional-neural-network-explained.html

https://github.com/vdumoulin/conv_arithmetic?source=post_page-----3bd2b1164a53--------------------------------

https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Quantum Convolution Neural Networks



quantum convolutional neural network

Nature

https://www.nature.com › nature physics › articles

Quantum convolutional neural networks

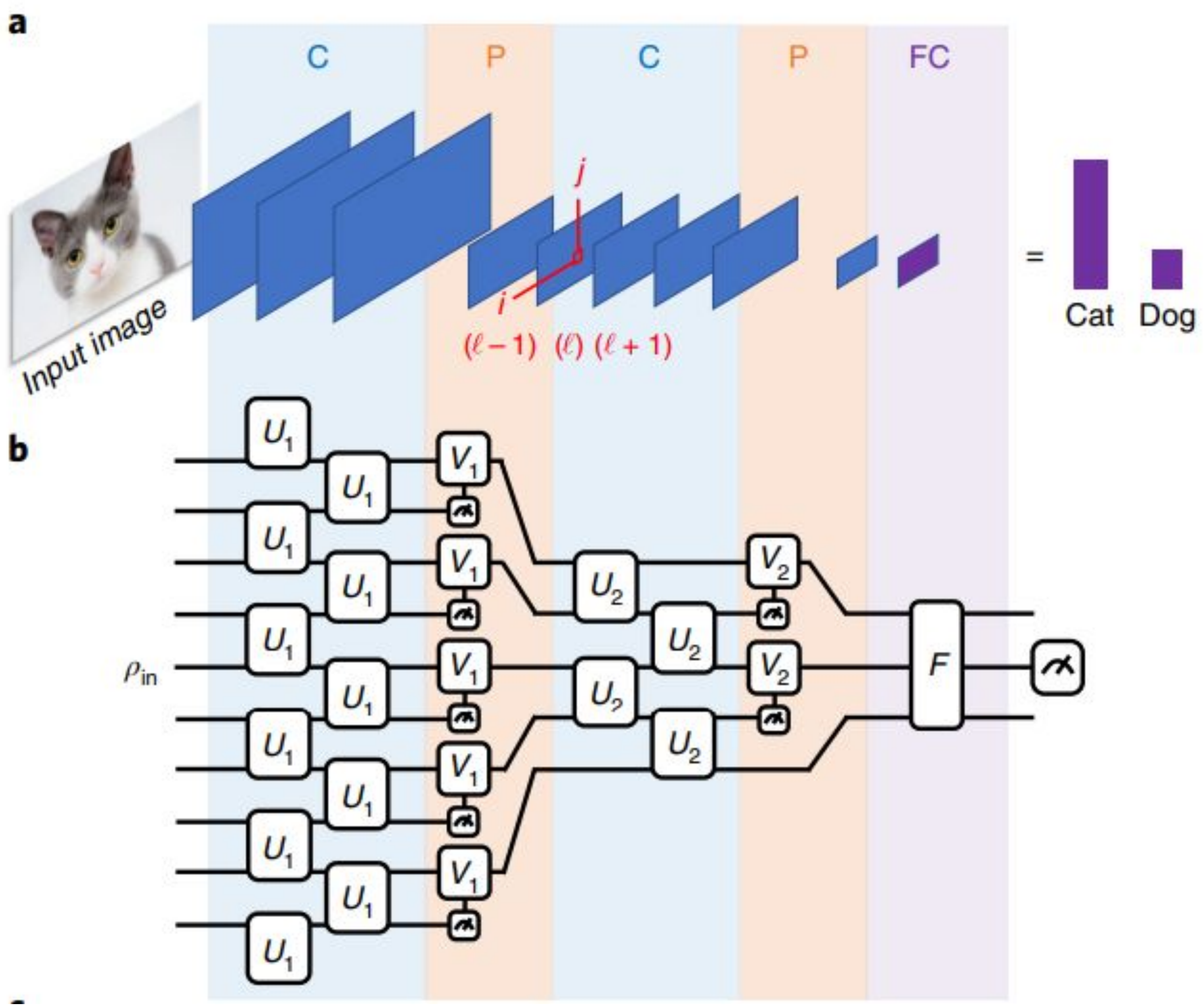
by I Cong · 2019 · Cited by 1206 — We introduce and analyse a **quantum** circuit-based algorithm inspired by **convolutional neural networks**, a highly effective model in machine…

Scholarly articles for **quantum convolutional neural network**

Quantum convolutional neural networks - Cong - Cited by 1206

A tutorial on **quantum convolutional neural networks** (… - Oh - Cited by 131

Quantum convolutional neural network for classical … - Hur - Cited by 209


a
C
P
C
P
FC
Input image
j
i
(ℓ − 1) (ℓ) (ℓ + 1)
=
Cat
Dog
b
ρ_{in}
U_1
V_1
U_2
V_2
F

QCNN

Quantum Convolutional Neural Networks (QCNN) behave in a similar manner to CCNNs.

First, we encode our pixelated image into a quantum circuit using a given feature map, such Qiskit's ZFeatureMap or ZZFeatureMap or others available in the circuit library.

Tabish sagri ~ Hello from my son!

CNN

After encoding our image, we apply alternating convolutional and pooling layers.

By applying these alternating layers, we reduce the dimensionality of our circuit until we are left with one qubit.

We can then classify our input image by measuring the output of this one remaining qubit.

QCNN

Quantum Convolutional Neural Networks (QCNN) behave in a similar manner to CCNNs.

First, we encode our pixelated image into a quantum circuit using a given feature map, such Qiskit’s ZFeatureMap or ZZFeatureMap or others available in the circuit library.

After encoding our image, we apply alternating convolutional and pooling layers. By applying these alternating layers, we reduce the dimensionality of our circuit until we are left with one qubit. We can then classify our input image by measuring the output of this one remaining qubit.

QCNN - Convolution Layer

The Quantum Convolutional Layer will consist of a series of two qubit unitary operators, which recognize and determine relationships between the qubits in our circuit.

$$U^\dagger U = I_4$$

QCNN - Pooling Layer

For the Quantum Pooling Layer, we cannot do the same as is done classically to reduce the dimension, i.e. the number of qubits in our circuit.

Instead, we reduce the number of qubits by performing operations upon each until a specific point and then disregard certain qubits in a specific layer.

It is these layers where we stop performing operations on certain qubits that we call our 'pooling layer'.

QCNN

In the QCNN, each layer contains parameterized circuits, meaning we alter our output result by adjusting the parameters of each layer.

When training our QCNN, it is these parameters that are adjusted to reduce the loss function of our QCNN.

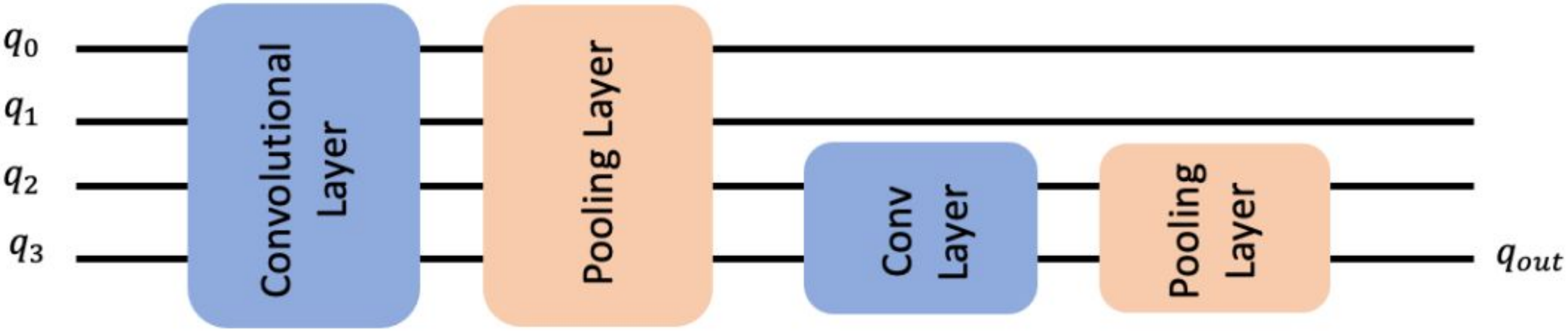
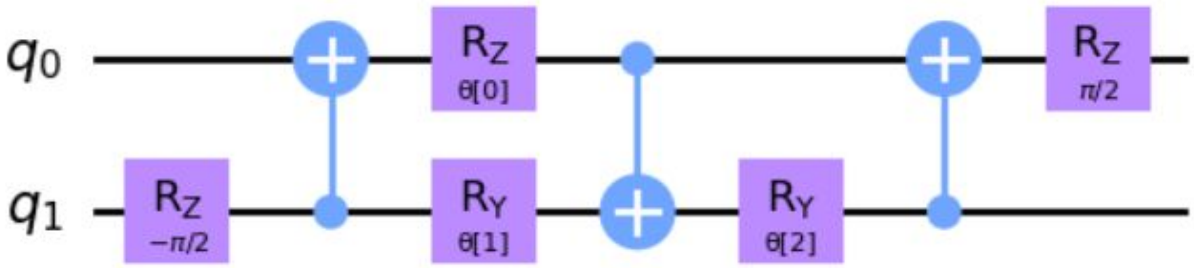


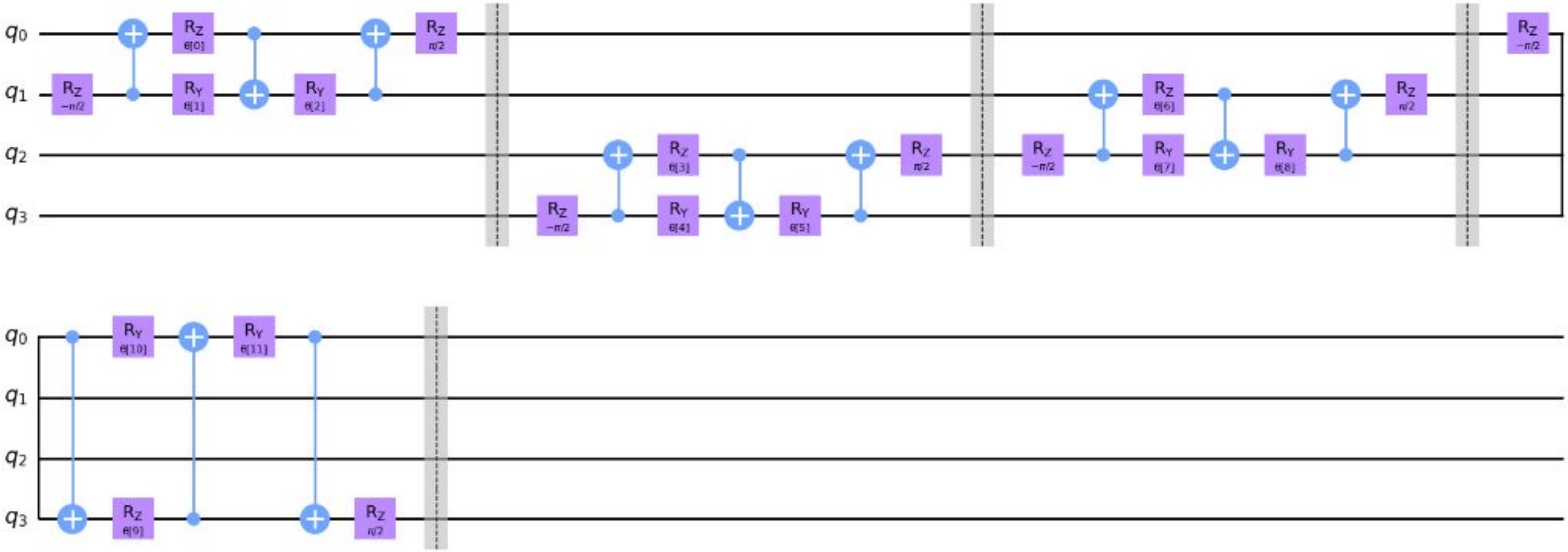
Figure 2: Example QCNN containing four qubits. The first Convolutional Layer acts on all the qubits. This is followed by the first pooling layer, which reduces the dimensionality of the QCNN from four qubits to two qubits by disregarding the first two. The second Convolutional layer then detects features between the two qubits still in use in the QCNN, followed by another pooling layer, which reduces the dimensionality from two qubits to one, which will be our output qubit.

QCNN - Convolution Layer

```python
# We now define a two qubit unitary as defined in [3]
def conv_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)
    target.cx(1, 0)
    target.rz(np.pi / 2, 0)
    return target


# Let's draw this circuit and see what it looks like
params = ParameterVector("θ", length=3)
circuit = conv_circuit(params)
circuit.draw("mpl", style="clifford")
```



QCNN - Convolution Layer



```python
def conv_layer(num_qubits, param_prefix):
    qc = QuantumCircuit(num_qubits, name="Convolutional Layer")
    qubits = list(range(num_qubits))
    param_index = 0
    params = ParameterVector(param_prefix, length=num_qubits * 3)
    for q1, q2 in zip(qubits[0::2], qubits[1::2]):
        qc = qc.compose(conv_circuit(params[param_index : (param_index + 3)]), [q1, q2])
        qc.barrier()
        param_index += 3
    for q1, q2 in zip(qubits[1::2], qubits[2::2] + [0]):
        qc = qc.compose(conv_circuit(params[param_index : (param_index + 3)]), [q1, q2])
        qc.barrier()
        param_index += 3

    qc_inst = qc.to_instruction()

    qc = QuantumCircuit(num_qubits)
    qc.append(qc_inst, qubits)
    return qc


circuit = conv_layer(4, "θ")
circuit.decompose().draw("mpl", style="clifford")
```

Pooling Layer

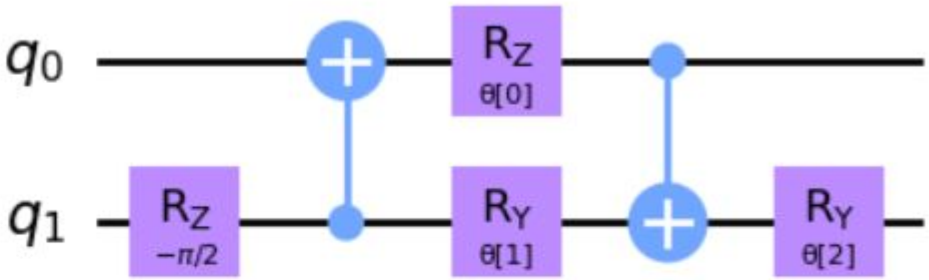
The purpose of a pooling layer is to reduce the dimensions of our Quantum Circuit, i.e. reduce the number of qubits in our circuit, while retaining as much information as possible from previously learned data.

Reducing the amount of qubits also reduces the computational cost of the overall circuit, as the number of parameters that the QCNN needs to learn decreases.

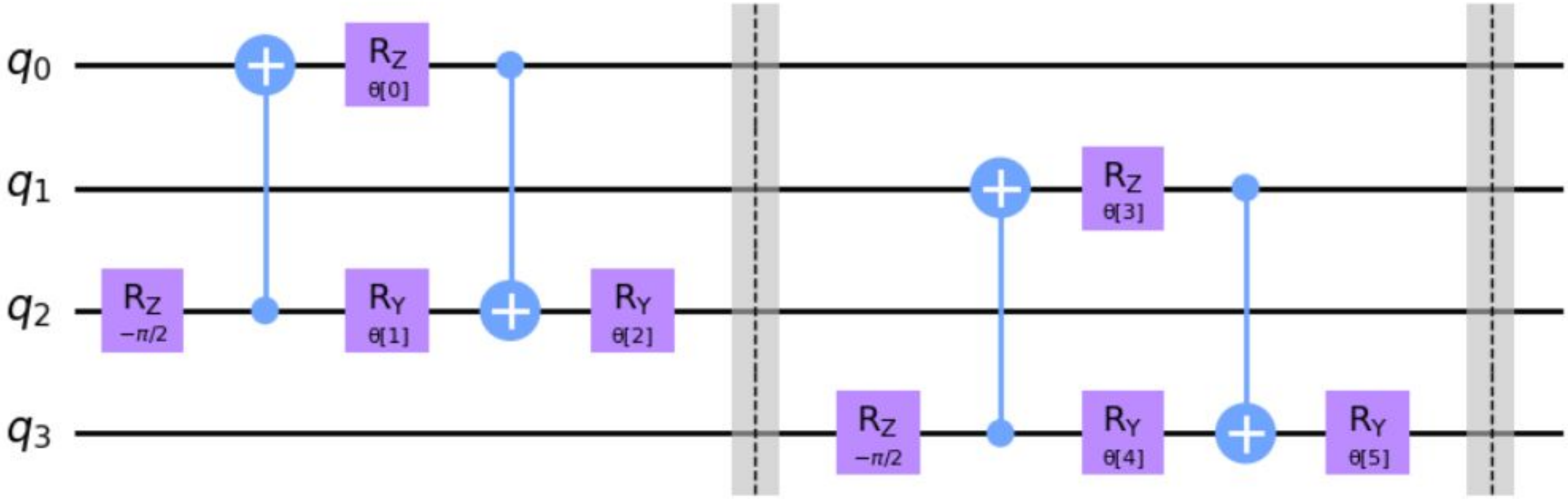
Pooling Layer

```python
def pool_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)

    return target
```

```python
params = ParameterVector("θ", length=3)
circuit = pool_circuit(params)
circuit.draw("mpl", style="clifford")
```

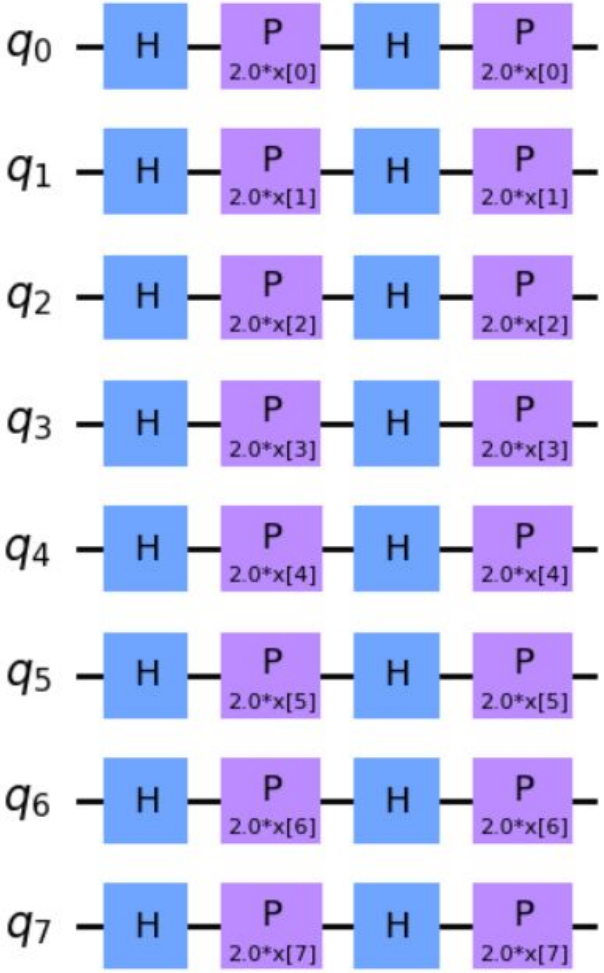


Pooling Layer

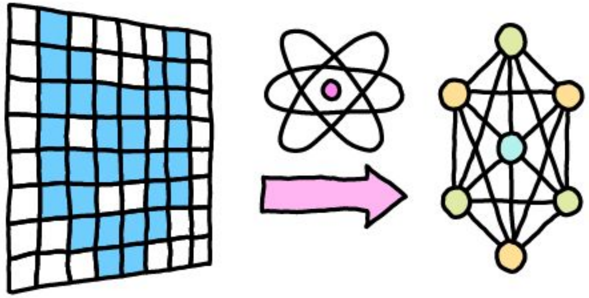


```
def pool_layer(sources, sinks, param_prefix):
    num_qubits = len(sources) + len(sinks)
    qc = QuantumCircuit(num_qubits, name="Pooling Layer")
    param_index = 0
    params = ParameterVector(param_prefix, length=num_qubits // 2 * 3)
    for source, sink in zip(sources, sinks):
        qc = qc.compose(pool_circuit(params[param_index : (param_index + 3)]), [source, sink])
        qc.barrier()
        param_index += 3

    qc_inst = qc.to_instruction()

    qc = QuantumCircuit(num_qubits)
    qc.append(qc_inst, range(num_qubits))
    return qc


sources = [0, 1]
sinks = [2, 3]
circuit = pool_layer(sources, sinks, "θ")
circuit.decompose().draw("mpl", style="clifford")
```

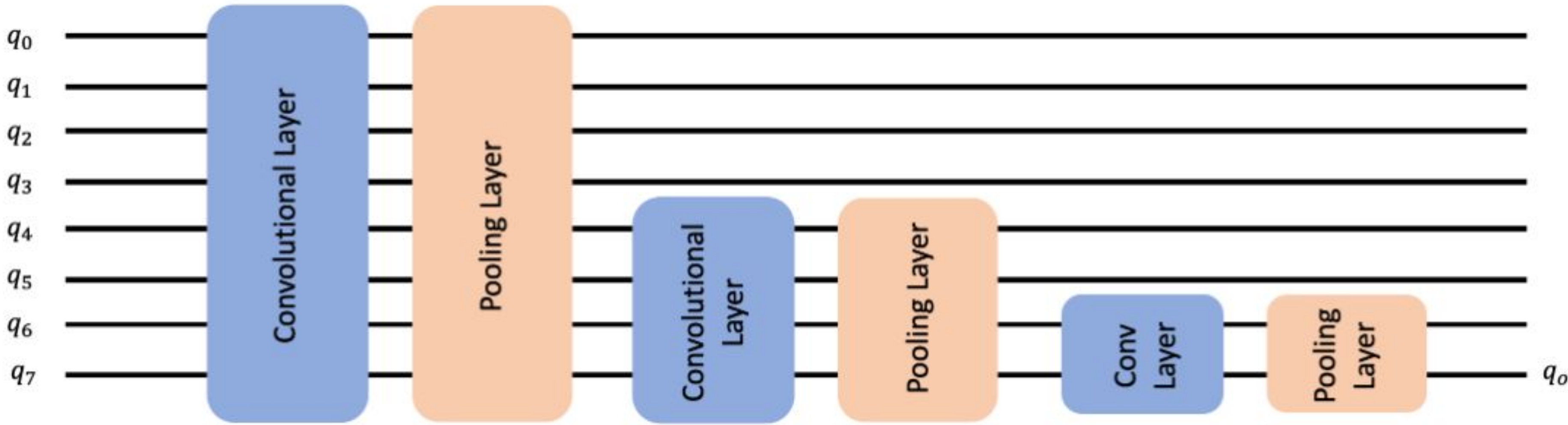
Example



q0 H P 2.0*x[0] H P 2.0*x[0]
q1 H P 2.0*x[1] H P 2.0*x[1]
q2 H P 2.0*x[2] H P 2.0*x[2]
q3 H P 2.0*x[3] H P 2.0*x[3]
q4 H P 2.0*x[4] H P 2.0*x[4]
q5 H P 2.0*x[5] H P 2.0*x[5]
q6 H P 2.0*x[6] H P 2.0*x[6]
q7 H P 2.0*x[7] H P 2.0*x[7]

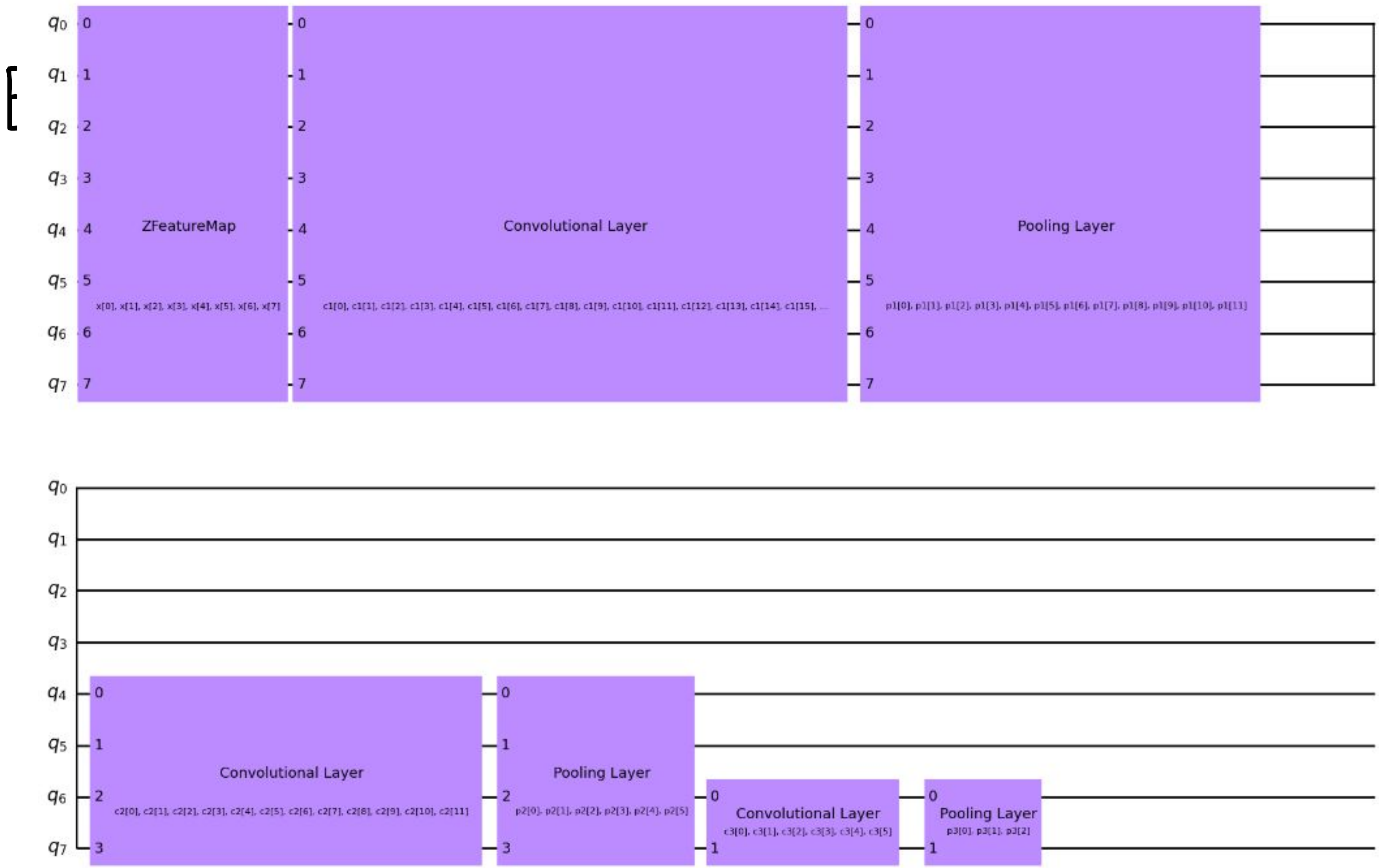


Example



q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
Convolutional Layer
Pooling Layer
Convolutional Layer
Pooling Layer
Conv Layer
Pooling Layer
q_o

```python
feature_map = ZFeatureMap(8)

ansatz = QuantumCircuit(8, name="Ansatz")

# First Convolutional Layer
ansatz.compose(conv_layer(8, "c1"), list(range(8)), inplace=True)

# First Pooling Layer
ansatz.compose(pool_layer([0, 1, 2, 3], [4, 5, 6, 7], "p1"), list(range(8)), inplace=True)

# Second Convolutional Layer
ansatz.compose(conv_layer(4, "c2"), list(range(4, 8)), inplace=True)

# Second Pooling Layer
ansatz.compose(pool_layer([0, 1], [2, 3], "p2"), list(range(4, 8)), inplace=True)

# Third Convolutional Layer
ansatz.compose(conv_layer(2, "c3"), list(range(6, 8)), inplace=True)

# Third Pooling Layer
ansatz.compose(pool_layer([0], [1], "p3"), list(range(6, 8)), inplace=True)

# Combining the feature map and ansatz
circuit = QuantumCircuit(8)
circuit.compose(feature_map, range(8), inplace=True)
circuit.compose(ansatz, range(8), inplace=True)
```



ZFeatureMap
x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]
Convolutional Layer
c1[0], c1[1], c1[2], c1[3], c1[4], c1[5], c1[6], c1[7], c1[8], c1[9], c1[10], c1[11], c1[12], c1[13], c1[14], c1[15], ...
Pooling Layer
p1[0], p1[1], p1[2], p1[3], p1[4], p1[5], p1[6], p1[7], p1[8], p1[9], p1[10], p1[11]
Convolutional Layer
c2[0], c2[1], c2[2], c2[3], c2[4], c2[5], c2[6], c2[7], c2[8], c2[9], c2[10], c2[11]
Pooling Layer
p2[0], p2[1], p2[2], p2[3], p2[4], p2[5]
Convolutional Layer
c3[0], c3[1], c3[2], c3[3], c3[4], c3[5]
Pooling Layer
p3[0], p3[1], p3[2]

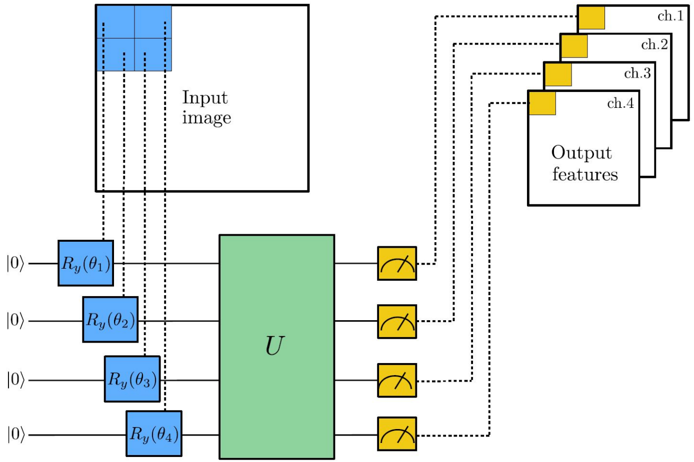
References For QCNN

https://qiskit-community.github.io/qiskit-machine-learning/tutorials/11_quantum_convolutional_neural_networks.html

https://www.ibm.com/topics/convolutional-neural-networks

https://sci-hub.se/https://www.nature.com/articles/s41567-019-0648-8

Quanvolutional Neural Networks



Quanvolutional Neural Networks: Powering Image Recognition with Quantum Circuits

Maxwell Henderson[1]*, Samriddhi Shakya[1], Shashindra Pradhan[1], and Tristan Cook[1]

[1]*QxBranch, Inc., 777 6th St NW, 11th Floor, Washington DC, 20001*

Convolutional neural networks (CNNs) have rapidly risen in popularity for many machine learning applications, particularly in the field of image recognition. Much of the benefit generated from these networks comes from their ability to extract features from the data in a hierarchical manner. These features are extracted using various transformational layers, notably the convolutional layer which gives the model its name. In this work, we introduce a new type of transformational layer called a quantum convolution, or quanvolutional layer. Quanvolutional layers operate on input data by locally transforming the data using a number of random quantum circuits, in a way that is similar to the transformations performed by random convolutional filter layers. Provided these quantum transformations produce meaningful features for classification purposes, then the overall algorithm could be quite useful for near term quantum computing, because it requires small quantum circuits with little to no error correction. In this work, we empirically evaluated the potential benefit of these quantum transformations by comparing three types of models built on the MNIST dataset: CNNs, quantum convolutional neural networks (QNNs), and CNNs with additional non-linearities introduced. Our results showed that the QNN models had both higher test set accuracy as well as faster training compared to the purely classical CNNs.

Quanvolutional

Quanvolutional Layers enhance or replace traditional convolutional layers in CNNs. They operate as follows:

Quantum Circuit Embedding: Input data (e.g., image patches) is encoded into a quantum state using a quantum circuit.

Quantum Transformation: The quantum state undergoes transformations via quantum gates, exploiting properties like superposition and entanglement to extract complex features.

Measurement: The quantum state is measured to produce classical data, which serves as features for the neural network.

Feature Extraction: The classical data from quantum measurements is used as features in the neural network, passed to subsequent layers for further processing and classification.

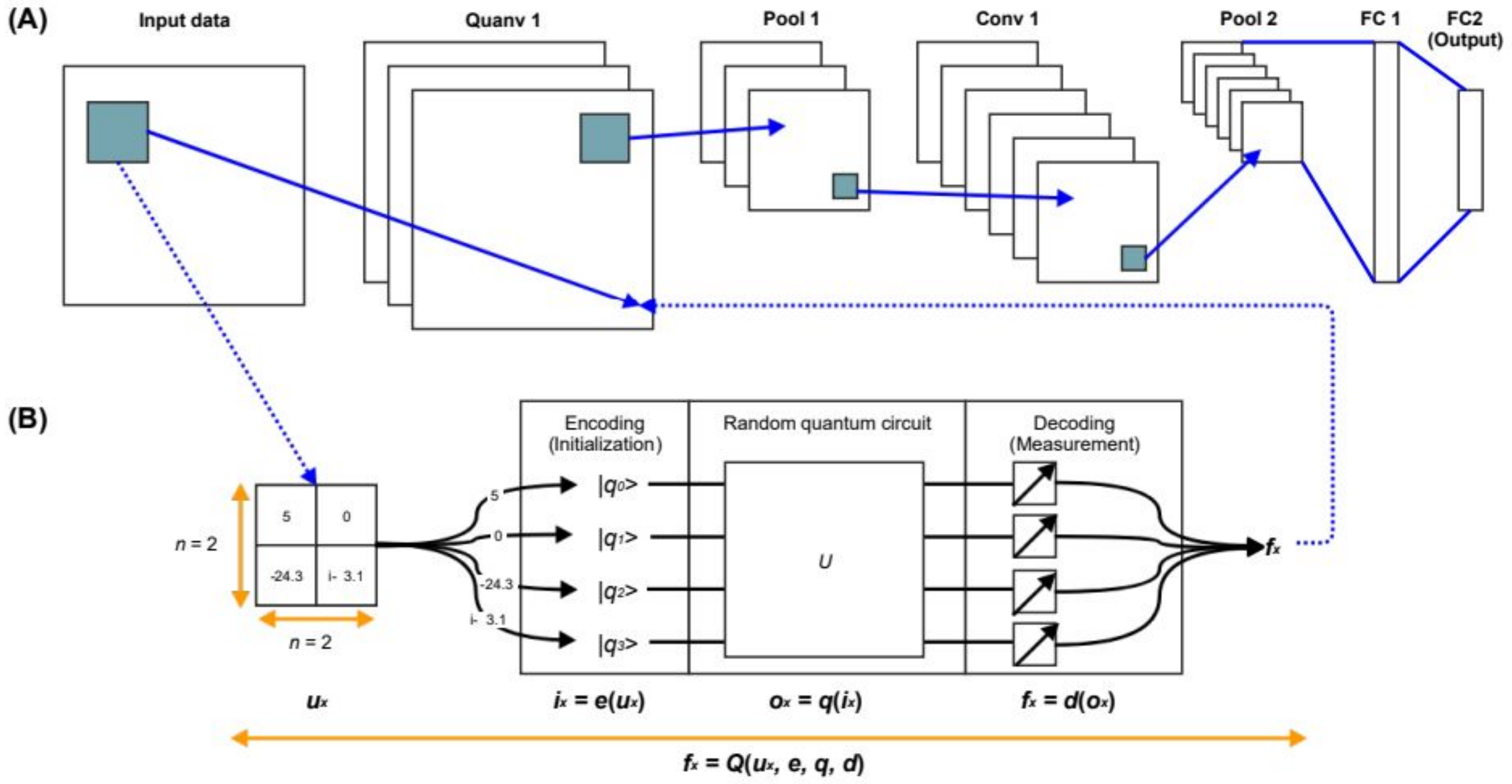


Fig. 1.: A. Simple example of a quanvolutional layer in a full network stack. The quanvolutional layer contains several quanvolutional filters (three in this example) that transform the input data into different output feature maps. B. An in-depth look at the processing of classical data into and out of the random quantum circuit in the quanvolutional filter.

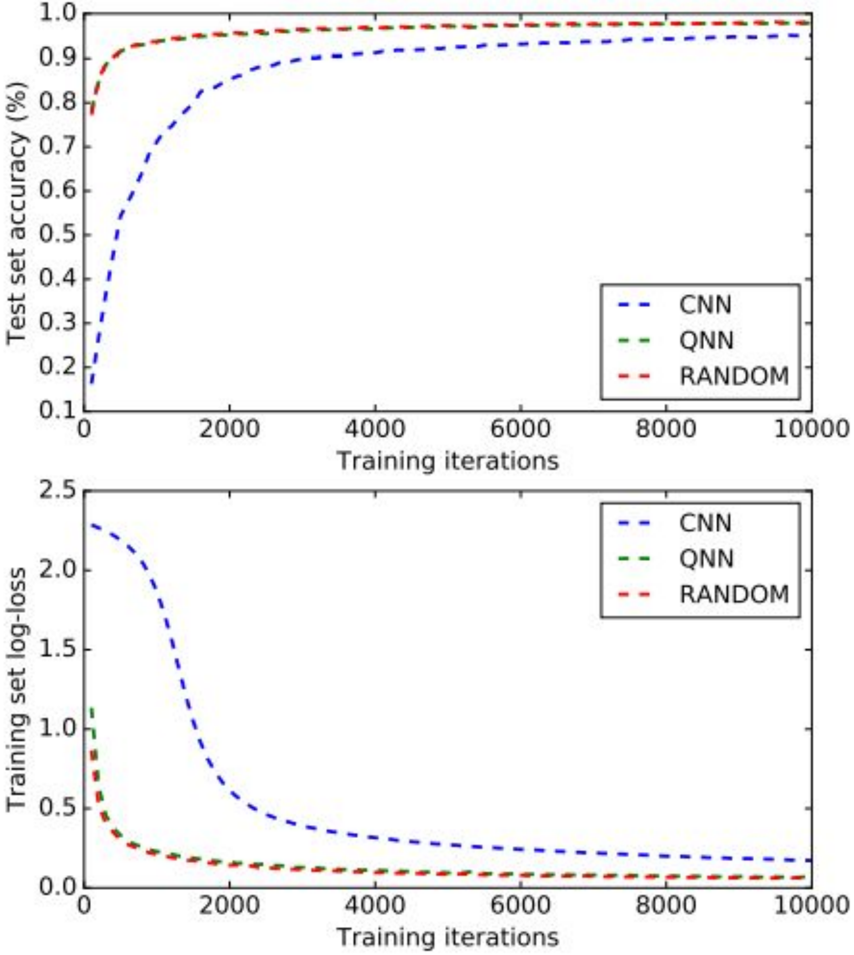
Equations for a Quanvolutional Layer

$$i_x = e(u_x).$$

$$o_x = q(i_x) = q(e(u_x)).$$

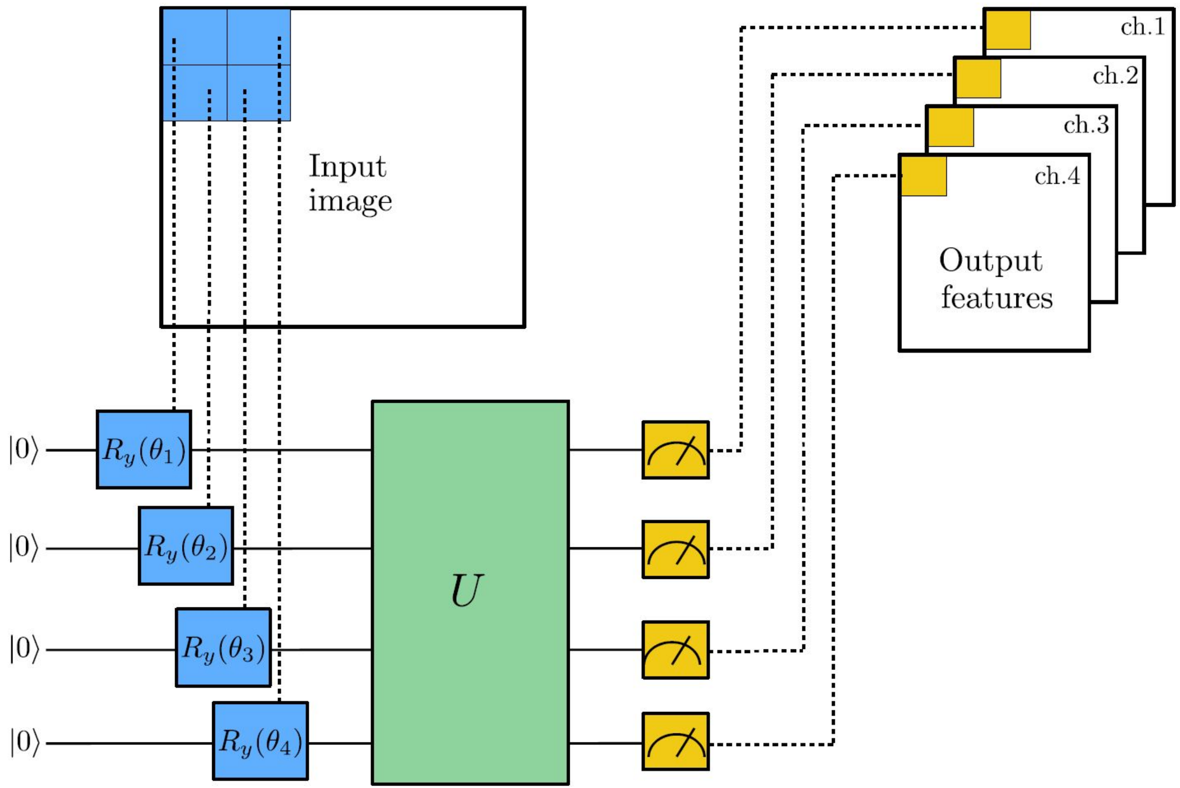
$$f_x = d(o_x) = d(q(e(u_x)))$$

$$f_x = Q(u_x, e, q, d).$$



Test set accuracy (%)
Training iterations
CNN
QNN
RANDOM
Training set log-loss
Training iterations
CNN
QNN
RANDOM

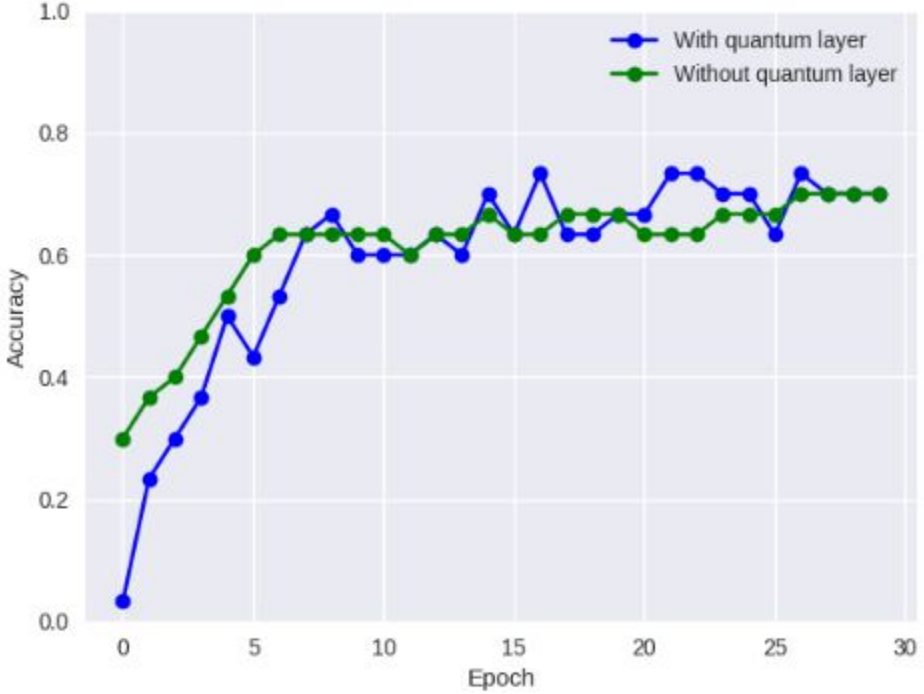
Quanvolutional Neural Network



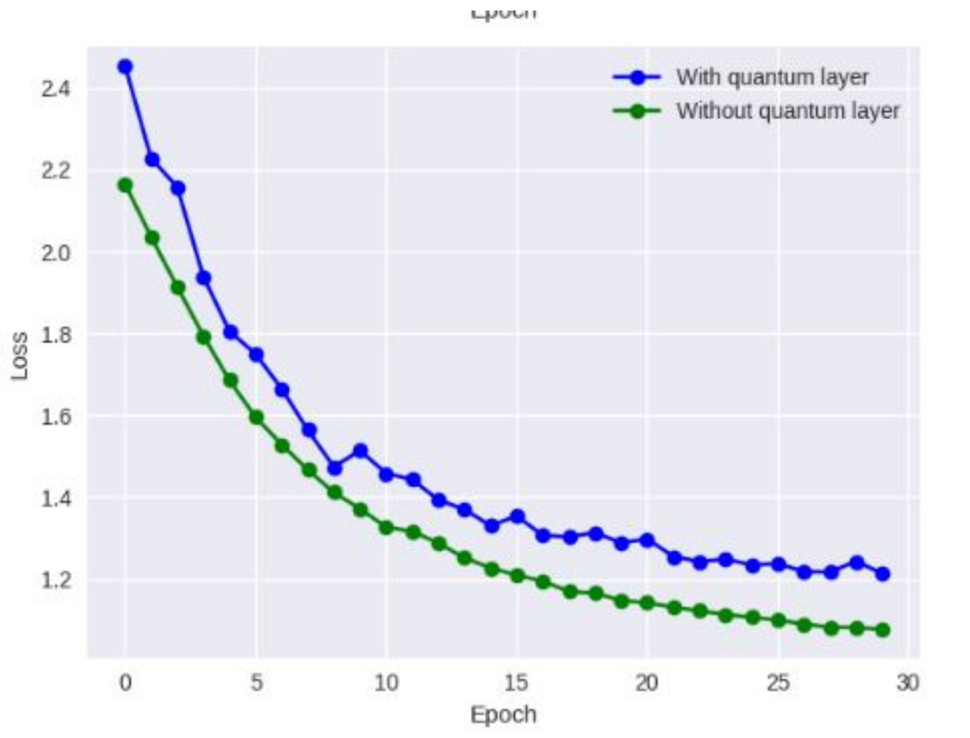
Quanvolutional Layer

1. A small region of the input image, in our example a 2×2 square, is embedded into a quantum circuit. In this demo, this is achieved with parametrized rotations applied to the qubits initialized in the ground state.
2. A quantum computation, associated to a unitary U is performed on the system. The unitary could be generated by a variational quantum circuit or, more simply, by a random circuit as proposed .
3. The quantum system is finally measured, obtaining a list of classical expectation values. The measurement results could also be classically post-processed but, for simplicity, in this demo we directly use the raw expectation values.
4. Analogously to a classical convolution layer, each expectation value is mapped to a different channel of a single output pixel.
5. Iterating the same procedure over different regions, one can scan the full input image, producing an output object which will be structured as a multi-channel image.
6. The quantum convolution can be followed by further quantum layers or by classical layers.

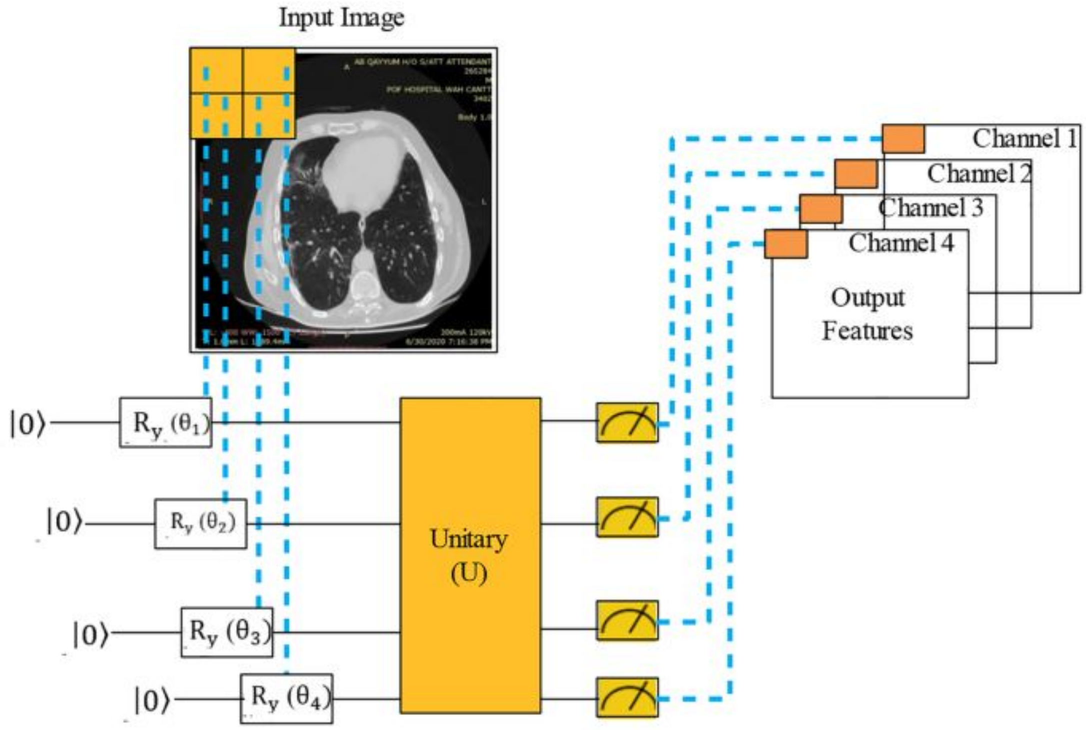
Results Generated By PennyLane



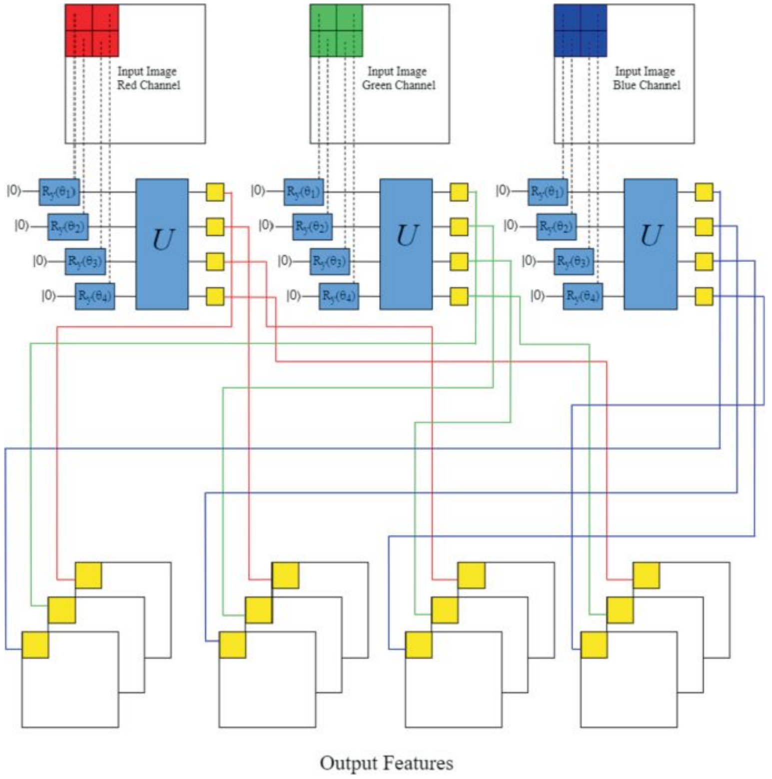
Results Generated By PennyLane



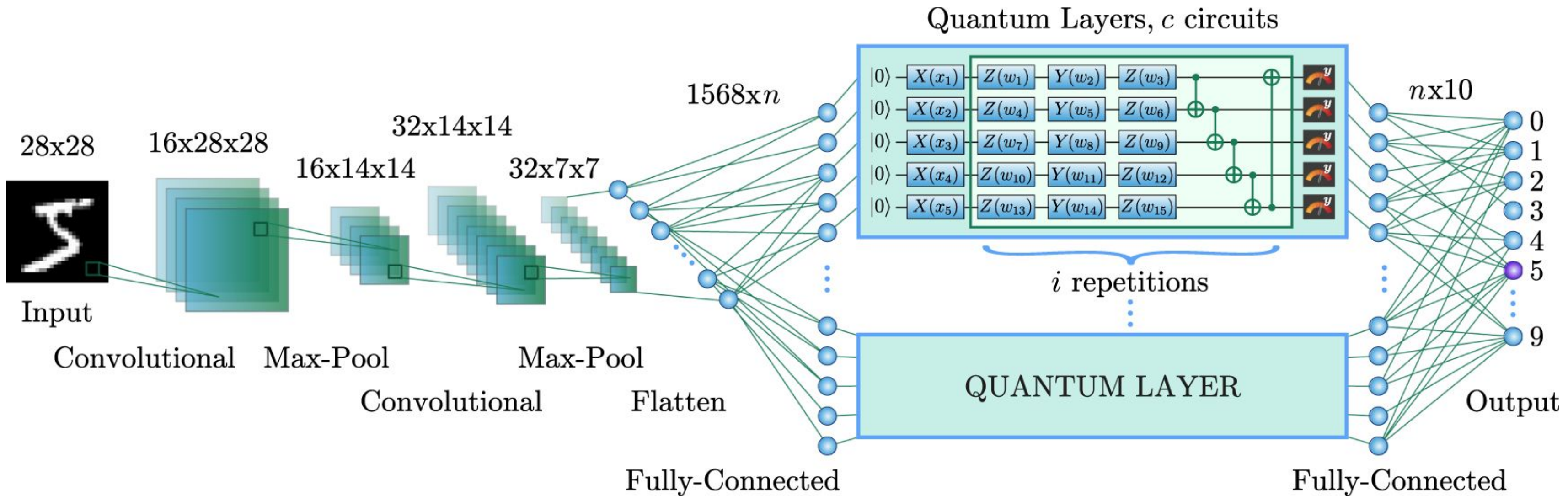
Quantum Machine Learning Architecture for COVID-19 Classification Based on Synthetic Data Generation



GQNN: Greedy Quanvolutional Neural Network Model



Quantum machine learning for image classification



References For QCNN

https://arxiv.org/pdf/1904.04767

https://pennylane.ai/qml/demos/tutorial_quanvolution/

https://github.com/viventriglia/Quantum_Neural_Network_QNN

https://link.springer.com/chapter/10.1007/978-3-031-12413-6_31