



Convolutional Neural Networks

Amir H. Payberah
payberah@kth.se
2021-11-25



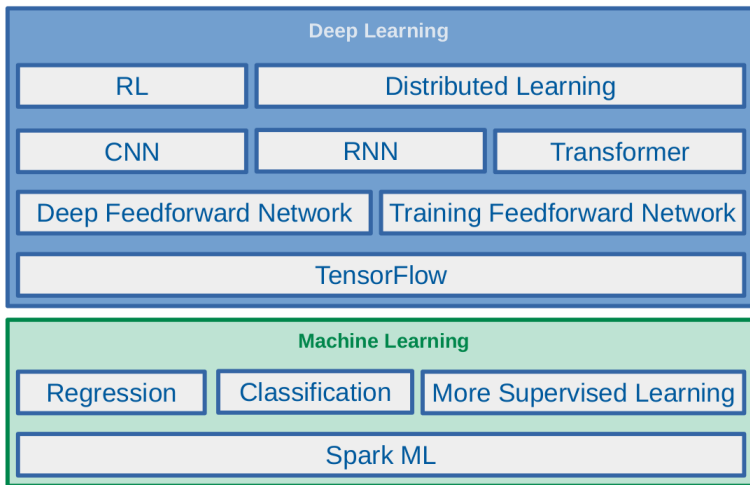


The Course Web Page

`https://id2223kth.github.io`
`https://tinyurl.com/6s5jy46a`

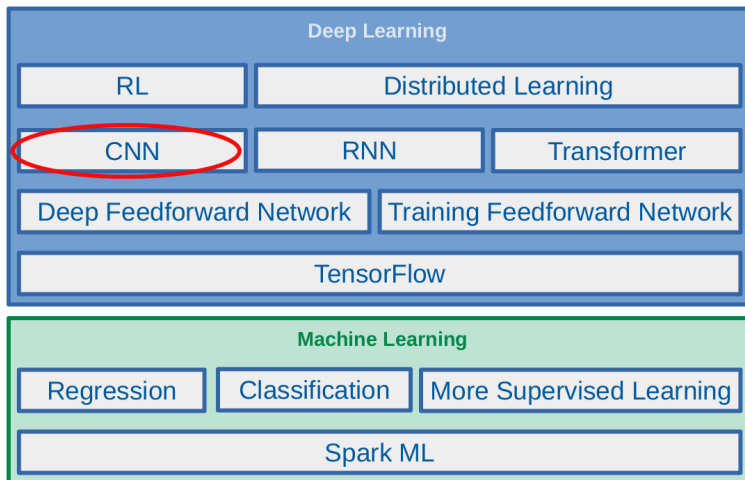


Where Are We?





Where Are We?



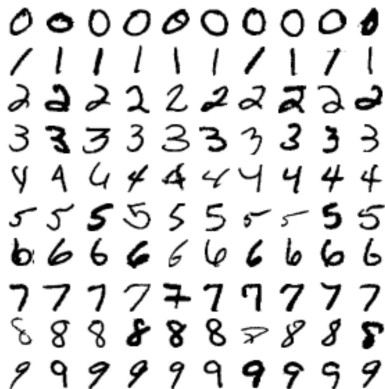


Let's Start With An Example



MNIST Dataset

- ▶ Handwritten digits in the **MNIST** dataset are **28x28 pixel greyscale images**.



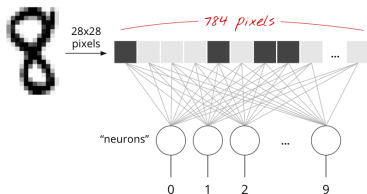
One-Layer Network For Classifying MNIST (1/4)



[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]

One-Layer Network For Classifying MNIST (2/4)

- ▶ Let's make a **one-layer** neural network for **classifying digits**.
- ▶ Each **neuron** in a neural network:
 - Does a **weighted sum** of all of its inputs
 - Adds a **bias**
 - Feeds the result through some **non-linear activation** function, e.g., **softmax**.



One-Layer Network For Classifying MNIST (3/4)

- ▶ Assume we have a batch of 100 images as the **input**.
- ▶ Using the **first column** of the **weights matrix** W , we compute the **weighted sum** of all the **pixels** of the **first image**.

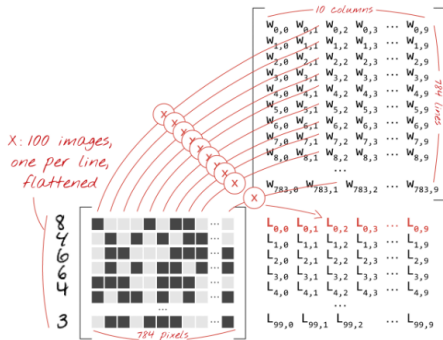
- The **first neuron**:

$$L_{0,0} = w_{0,0}x_0^{(1)} + w_{1,0}x_1^{(1)} + \dots + w_{783,0}x_{783}^{(1)}$$
- The **2nd neuron until the 10th**:

$$L_{0,1} = w_{0,1}x_0^{(1)} + w_{1,1}x_1^{(1)} + \dots + w_{783,1}x_{783}^{(1)}$$

$$\dots$$

$$L_{0,9} = w_{0,9}x_0^{(1)} + w_{1,9}x_1^{(1)} + \dots + w_{783,9}x_{783}^{(1)}$$
- Repeat the operation for the **other 99 images**, i.e., $x^{(2)} \dots x^{(100)}$



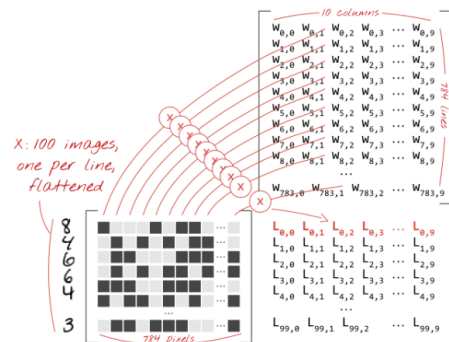
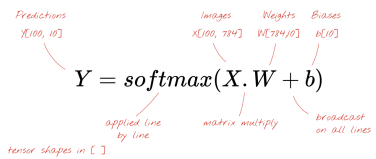
One-Layer Network For Classifying MNIST (4/4)

- ▶ Each neuron must now add its **bias**.
- ▶ Apply the **softmax activation function** for each instance $x^{(i)}$.

▶ For each input instance $x^{(i)}$: $L_i =$

$$\begin{bmatrix} L_{i,0} \\ L_{i,1} \\ \vdots \\ L_{i,9} \end{bmatrix}$$

▶ $\hat{y}_i = \text{softmax}(L_i + b)$



How Good the Predictions Are?

- ▶ Define the cost function $J(W)$ as the **cross-entropy** of **what the network tells us** (\hat{y}_i) and **what we know to be the truth** (y_i), for each instance $x^{(i)}$.
- ▶ Compute the **partial derivatives of the cross-entropy** with respect to all the **weights** and **all the biases**, $\nabla_W J(W)$.
- ▶ Update weights and biases by a **fraction of the gradient** $W^{(\text{next})} = W - \eta \nabla_W J(W)$

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	0	0	0

actual probabilities, "one-hot" encoded

Cross entropy: $-\sum Y_i \cdot \log(\hat{Y}_i)$

computed probabilities

0.1	0.2	0.1	0.3	0.2	0.1	0.9	0.2	0.1	0.1
0	1	2	3	4	5	6	7	8	9

this is a "6"



```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
model.fit(x_train, y_train, batch_size=100, epochs=10)
model.evaluate(x_test, y_test)
```

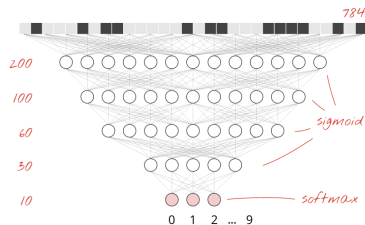



Some Improvement (1/5)

- ▶ Add more layers to **improve the accuracy**.
- ▶ On **intermediate layers** we will use the **sigmoid** activation function.
- ▶ We keep **softmax** as the activation function on the **last layer**.



[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]



Some Improvement (2/5)

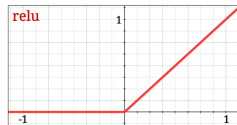
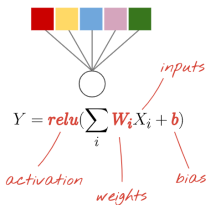
- ▶ Network **initialization**. e.g., using **He** initialization.
- ▶ Better **optimizer**, e.g., using **Adam** optimizer.



[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]

Some Improvement (3/5)

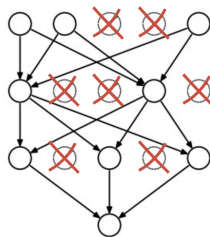
- ▶ Better activation function, e.g., using $\text{ReLU}(z) = \max(0, z)$.



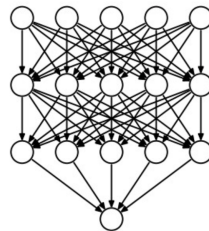
[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]

Some Improvement (4/5)

- ▶ Overcome **overfitting**, e.g., using **dropout**.



TRAINING
rate=0.4



EVALUATION
rate=0

[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]

Some Improvement (5/5)

- ▶ Start fast and **decay** the learning rate exponentially.
- ▶ You can do this with the `tf.keras.callbacks.LearningRateScheduler` callback.



[<https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd>]



```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, kernel_initializer="he_normal", activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

```
# lr decay function
def lr_decay(epoch):
    return 0.01 * math.pow(0.6, epoch)

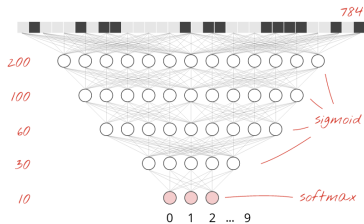
# lr schedule callback
lr_decay_callback = tf.keras.callbacks.LearningRateScheduler(lr_decay, verbose=True)

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'],
              callbacks=[lr_decay_callback])
```

```
model.fit(x_train, y_train, batch_size=100, epochs=10)
model.evaluate(x_test, y_test)
```

Vanilla Deep Neural Networks Challenges (1/2)

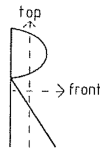
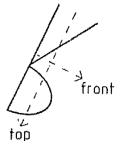
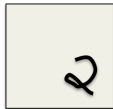
- ▶ Pixels of each image were flattened into a single vector (really bad idea).



- ▶ Vanilla deep neural networks do not scale.
 - In MNIST, images are black-and-white 28x28 pixel images: $28 \times 28 = 784$ weights.
- ▶ Handwritten digits are made of shapes and we discarded the shape information when we flattened the pixels.

Vanilla Deep Neural Networks Challenges (2/2)

- ▶ Difficult to **recognize objects**.
- ▶ **Rotation**
- ▶ **Lighting**: objects may **look different** depending on the level of **external lighting**.
- ▶ **Deformation**: objects can be deformed in a variety of **non-affine ways**.
- ▶ **Scale variation**: visual classes often exhibit **variation in their size**.
- ▶ **Viewpoint invariance**.





Tackle the Challenges

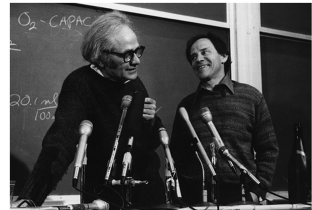
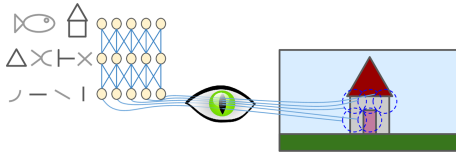
- ▶ Convolutional neural networks (CNN) can tackle the vanilla model challenges.
- ▶ CNN is a type of neural network that can take advantage of shape information.
- ▶ It applies a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification.



Filters and Convolution Operations

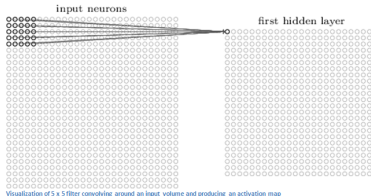
Brain Visual Cortex Inspired CNNs

- ▶ 1959, David H. Hubel and Torsten Wiesel.
- ▶ Many neurons in the visual cortex have a **small local receptive field**.
- ▶ They **react** only to visual stimuli located in a **limited region** of the visual field.



Receptive Fields and Filters

- ▶ Imagine a **flashlight** that is shining over the top left of the image.
- ▶ The **region** that it is shining over is called the **receptive field**.
- ▶ This **flashlight** is called a **filter**.
- ▶ A filter is a **set of weights**.
- ▶ A **filter** is a **feature detector**, e.g., straight edges, simple colors, and curves.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[<https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks>]

Filters Example (1/3)

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter



Original image



Visualization of the filter on the image

[<https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks>]

Filters Example (2/3)



Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

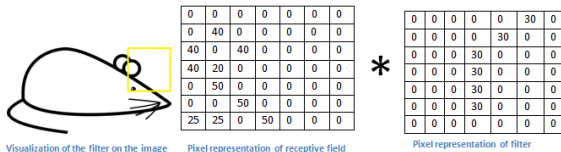
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = $(50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600$ (A large number!)

[<https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks>]

Filters Example (3/3)

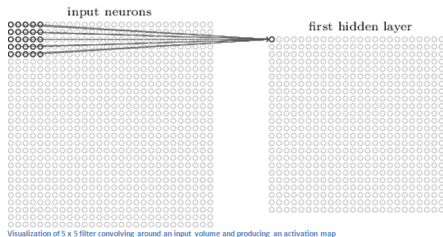


Multiplication and Summation = 0

[<https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks>]

Convolution Operation

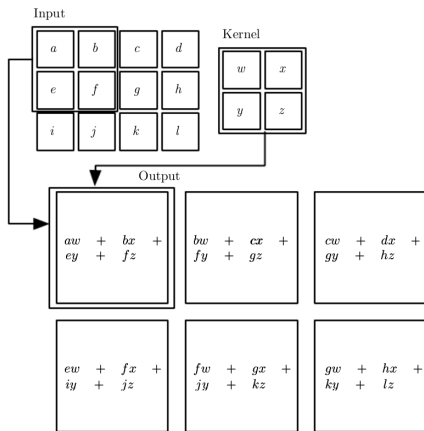
- ▶ **Convolution** takes a **filter** and **multiplying** it over the entire area of an input image.
- ▶ Imagine this **flashlight (filter)** **sliding across all the areas** of the input image.



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

[<https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks>]

Convolution Operation - 2D Example

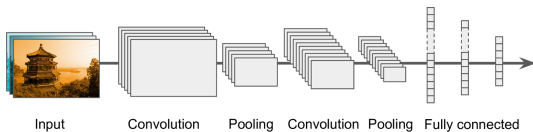




Convolutional Neural Network (CNN)

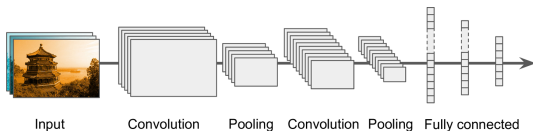
CNN Components (1/2)

- ▶ **Convolutional layers:** apply a specified number of **convolution filters** to the image.
- ▶ **Pooling layers:** **downsample the image** data extracted by the convolutional layers to **reduce the dimensionality** of the feature map in order to decrease processing time.
- ▶ **Dense layers:** a **fully connected layer** that performs **classification** on the features extracted by the convolutional layers and downsampled by the pooling layers.

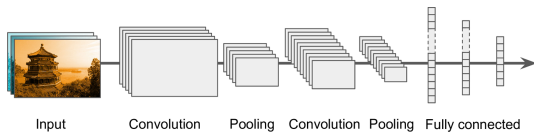


CNN Components (2/2)

- ▶ A **CNN** is composed of a **stack of convolutional modules**.
- ▶ Each **module** consists of a **convolutional layer** followed by a pooling layer.
- ▶ The **last module** is followed by **one or more dense layers** that perform **classification**.
- ▶ The **final dense layer** contains a **single node for each target class** in the model, with a **softmax** activation function.

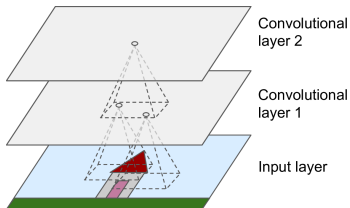


Convolutional Layer



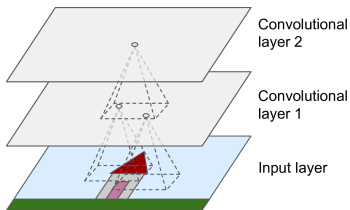
Convolutional Layer (1/4)

- ▶ Sparse interactions
- ▶ Each neuron in the convolutional layers is **only** connected to pixels in its **receptive field** (not every single pixel).



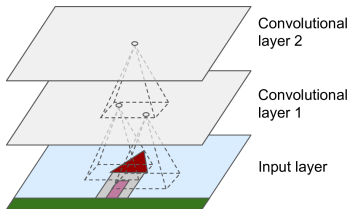
Convolutional Layer (2/4)

- ▶ Each neuron applies **filters** on its **receptive field**.
 - Calculates a **weighted sum** of the input pixels in the receptive field.
- ▶ Adds a **bias**, and feeds the result through its **activation function** to the next layer.
- ▶ The **output** of this layer is a **feature map (activation map)**



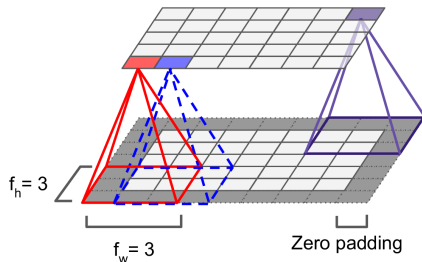
Convolutional Layer (3/4)

- ▶ Parameter sharing
- ▶ All neurons of a convolutional layer reuse the same weights.
- ▶ They apply the same filter in different positions.
- ▶ Whereas in a fully-connected network, each neuron had its own set of weights.



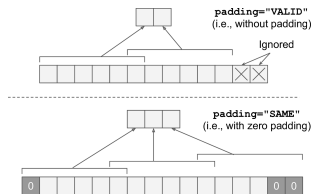
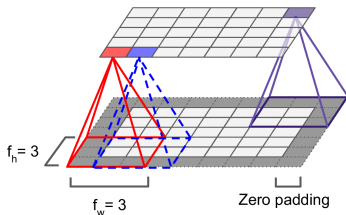
Convolutional Layer (4/4)

- ▶ Assume the filter size (kernel size) is $f_w \times f_h$.
 - f_h and f_w are the height and width of the receptive field, respectively.
- ▶ A neuron in row i and column j of a given layer is connected to the outputs of the neurons in the previous layer in rows i to $i + f_h - 1$, and columns j to $j + f_w - 1$.



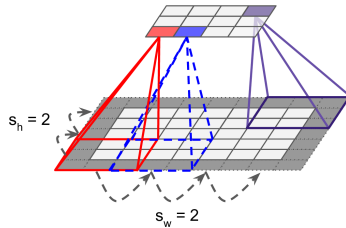
Padding

- ▶ What will happen if you apply a 5x5 filter to a 32x32 input volume?
 - The output volume would be 28x28.
 - The spatial dimensions decrease.
- ▶ **Zero padding**: in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs.
- ▶ In TensorFlow, padding can be either **SAME** or **VALID** to have zero padding or not.



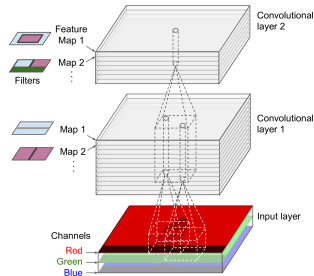
Stride

- ▶ The **distance** between **two consecutive receptive fields** is called the **stride**.
- ▶ The stride controls **how the filter convolves** around the input volume.
- ▶ Assume s_h and s_w are the **vertical and horizontal strides**, then, a neuron located in **row i** and **column j** in a layer is connected to the outputs of the neurons in the **previous layer** located in **rows $i \times s_h$ to $i \times s_h + f_h - 1$** , and **columns $j \times s_w$ to $j \times s_w + f_w - 1$** .



Stacking Multiple Feature Maps

- ▶ Up to now, we represented each convolutional layer with a **single feature map**.
- ▶ Each convolutional layer can be composed of **several feature maps** of equal sizes.
- ▶ Input images are also composed of **multiple sublayers**: **one per color channel**.
- ▶ A **convolutional layer simultaneously** applies **multiple filters** to its inputs.

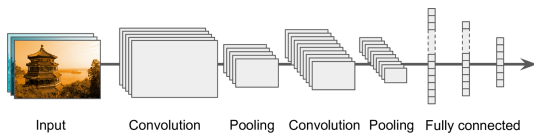




Activation Function

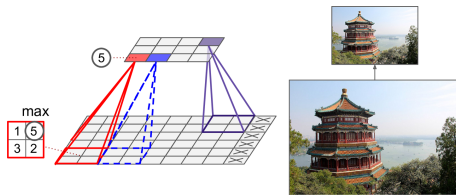
- ▶ After calculating a **weighted sum** of the input pixels in the **receptive fields**, and adding **biases**, each neuron feeds the result through its **ReLU activation function** to the next layer.
- ▶ The purpose of this activation function is to add **non linearity** to the system.

Pooling Layer



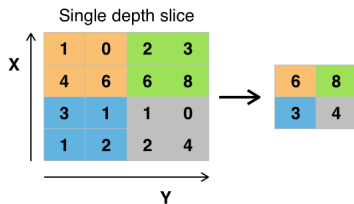
Pooling Layer (1/2)

- ▶ After the activation functions, we can apply a **pooling layer**.
- ▶ Its goal is to **subsample (shrink)** the input image.
 - To **reduce** the computational load, the memory usage, and the number of parameters.



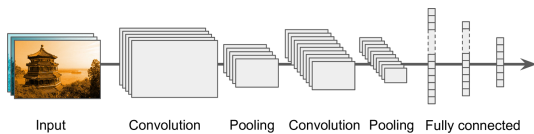
Pooling Layer (2/2)

- ▶ Each neuron in a pooling layer is connected to the outputs of a **receptive field** in the previous layer.
- ▶ A pooling neuron has **no weights**.
- ▶ It **aggregates the inputs** using an aggregation function such as the **max** or **mean**.



Example of Maxpool with a 2x2 filter and a stride of 2

Fully Connected Layer





Fully Connected Layer

- ▶ This layer takes an input from the **last convolution module**, and outputs an **N** dimensional vector.
 - **N** is the **number of classes** that the model has to choose from.
- ▶ For example, if you wanted a **digit classification** model, **N would be 10**.
- ▶ Each number in this **N** dimensional vector represents the **probability of a certain class**.



Flattening

- ▶ We need to **convert the output** of the convolutional part of the CNN into a **1D feature vector**.
- ▶ This operation is called **flattening**.
- ▶ It gets the **output of the convolutional layers**, **flattens** all its structure to create a **single long feature vector** to be used by the **dense layer** for the final classification.

Example

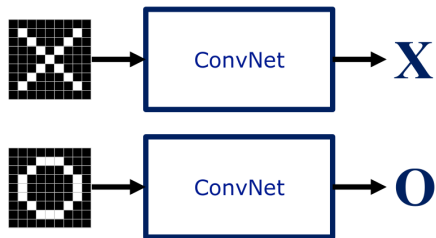
A Toy ConvNet: X's and O's

A two-dimensional
array of pixels

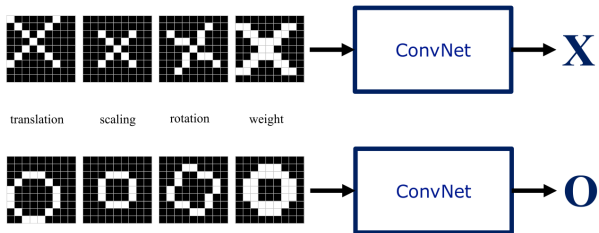


X or **O**

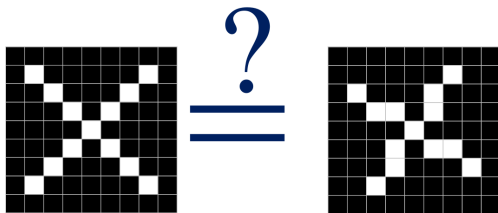
For Example



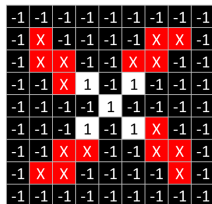
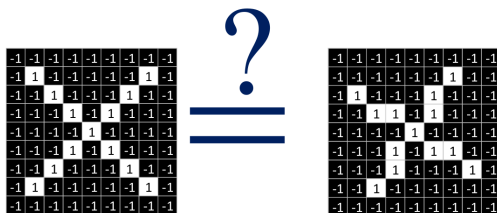
Trickier Cases



Deciding is Hard



What Computers See



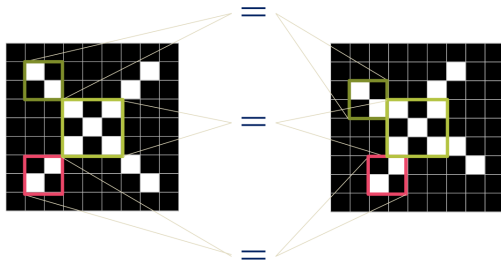
Computers are Literal

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

\neq

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

ConvNets Match Pieces of the Image



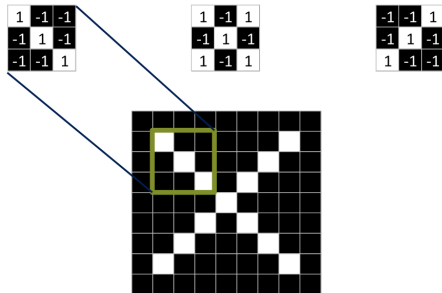
Filters Match Pieces of the Image

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

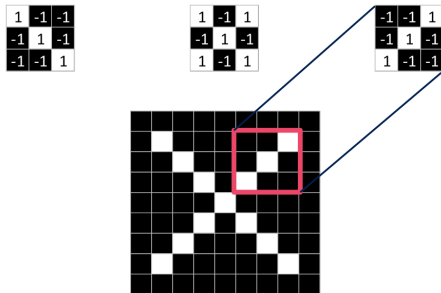
1	-1	1
-1	1	-1
1	-1	1

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

Filters Match Pieces of the Image



Filters Match Pieces of the Image

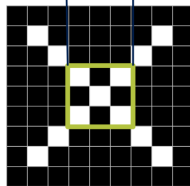


Filters Match Pieces of the Image

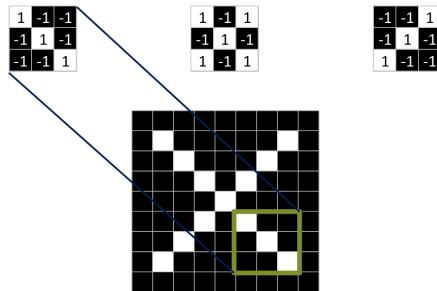
1	-1	-1
-1	1	-1
-1	-1	1

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

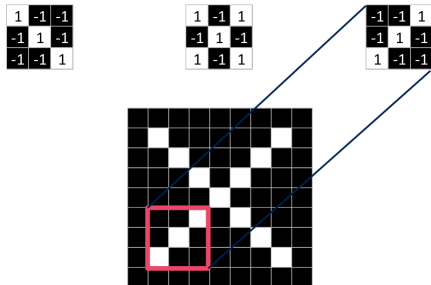
-1	-1	1
-1	1	-1
1	-1	-1



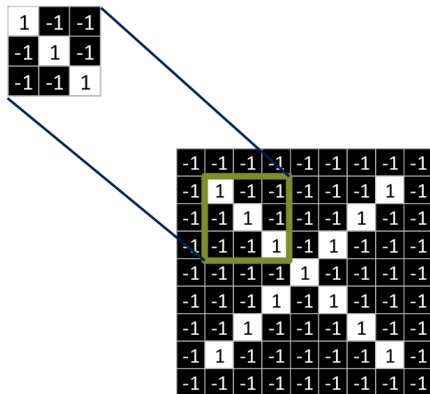
Filters Match Pieces of the Image



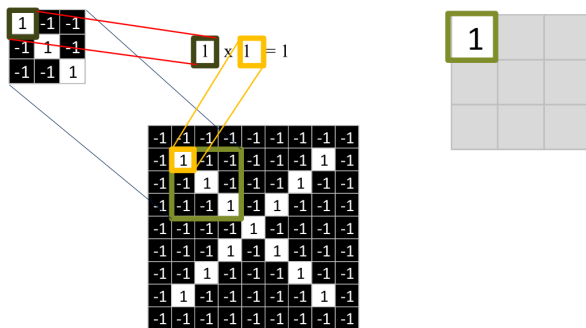
Filters Match Pieces of the Image



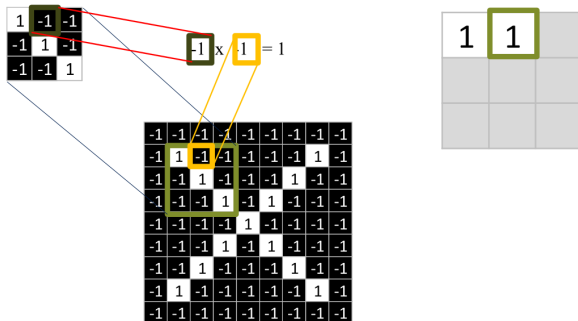
Filtering: The Math Behind the Match



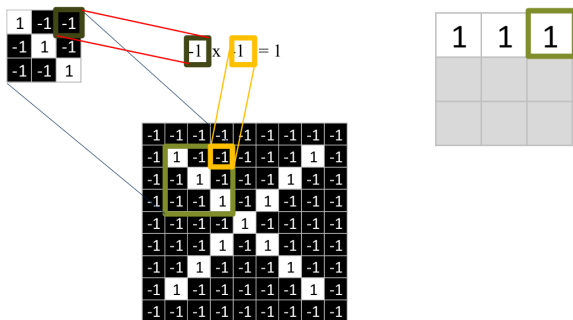
Filtering: The Math Behind the Match



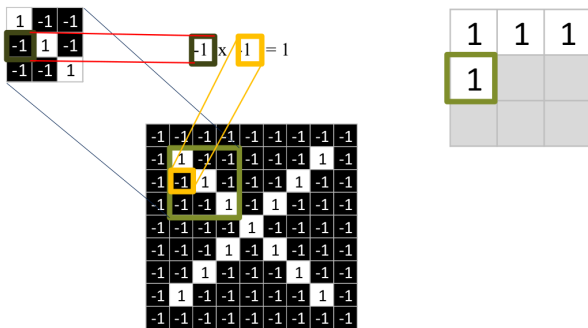
Filtering: The Math Behind the Match



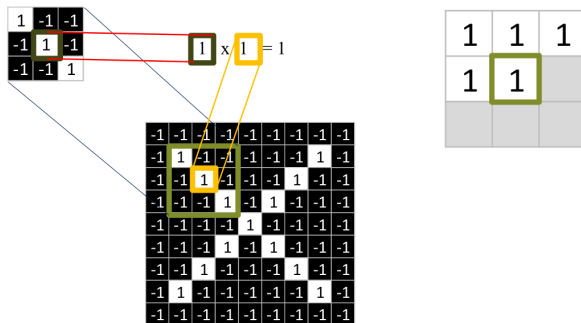
Filtering: The Math Behind the Match



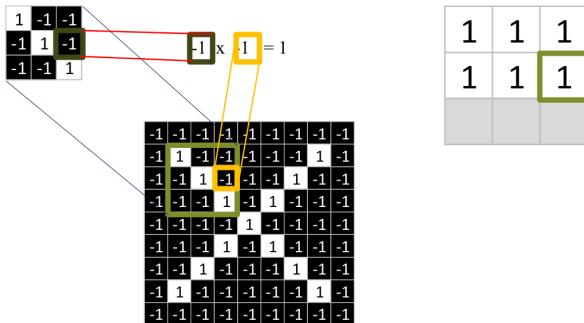
Filtering: The Math Behind the Match



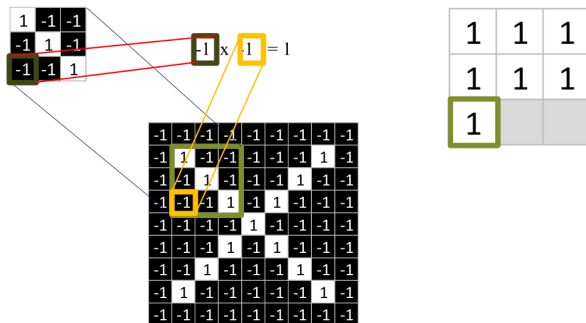
Filtering: The Math Behind the Match



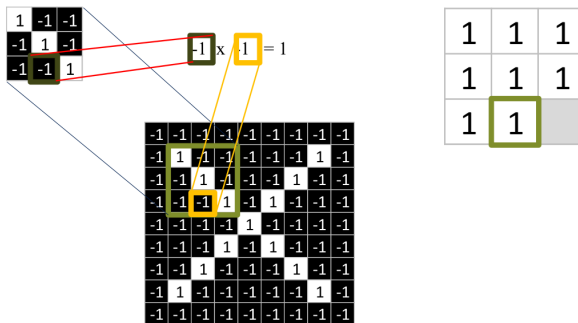
Filtering: The Math Behind the Match



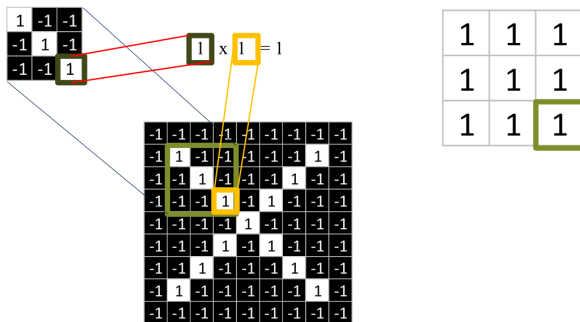
Filtering: The Math Behind the Match



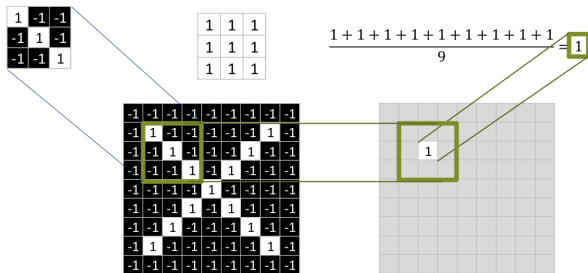
Filtering: The Math Behind the Match



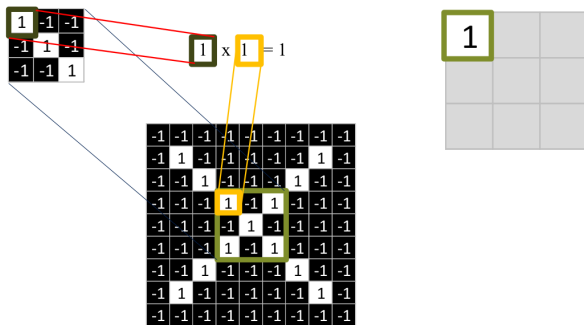
Filtering: The Math Behind the Match



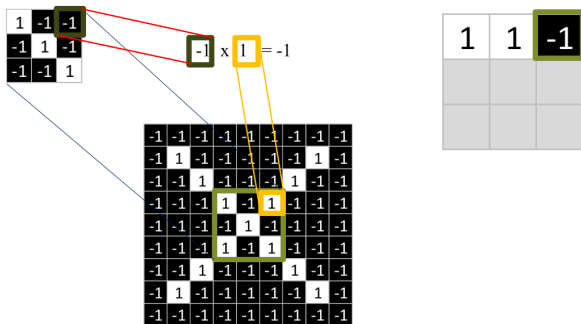
Filtering: The Math Behind the Match



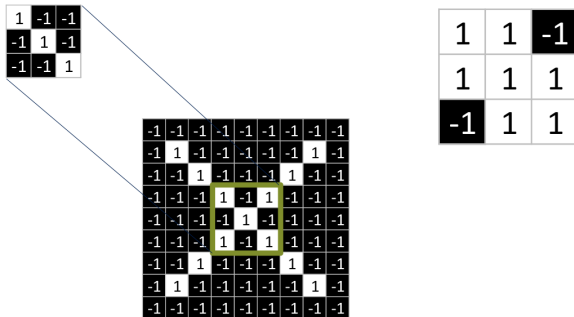
Filtering: The Math Behind the Match



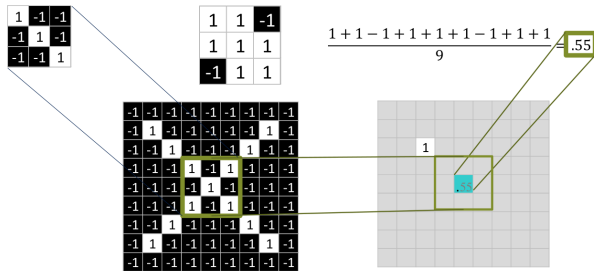
Filtering: The Math Behind the Match



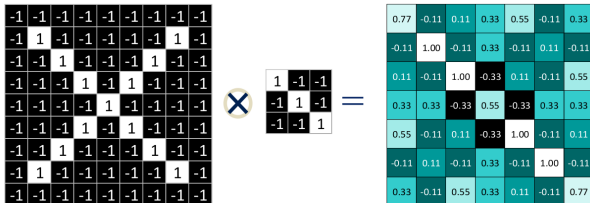
Filtering: The Math Behind the Match



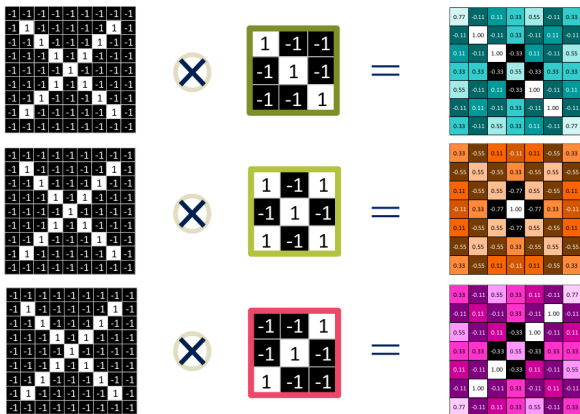
Filtering: The Math Behind the Match



Convolution: Trying Every Possible Match

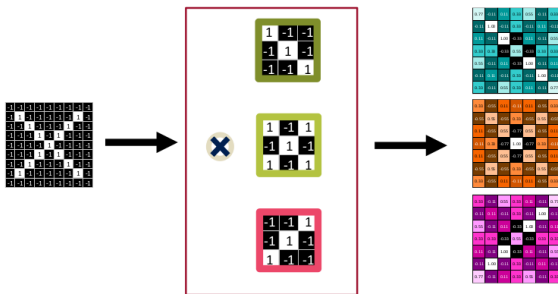


Three Filters Here, So Three Images Out

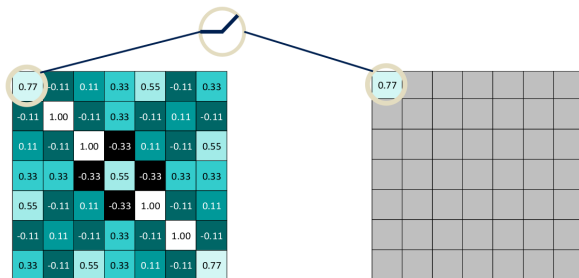


Convolution Layer

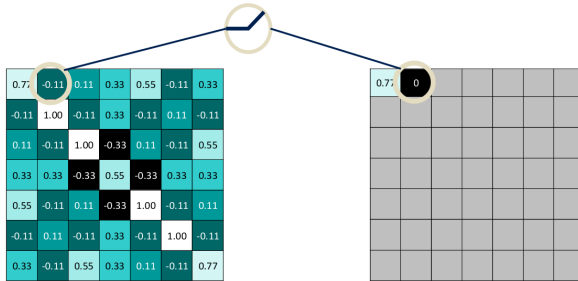
- ▶ One image becomes a **stack of filtered images**.



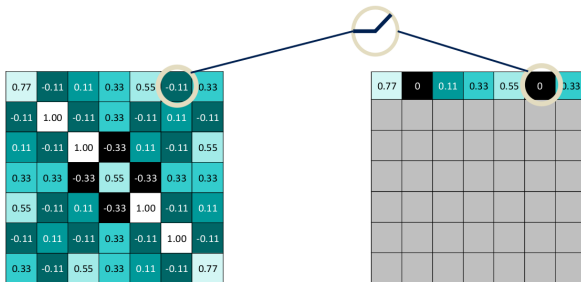
Rectified Linear Units (ReLUs)



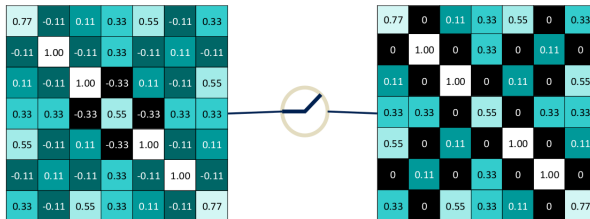
Rectified Linear Units (ReLUs)



Rectified Linear Units (ReLUs)

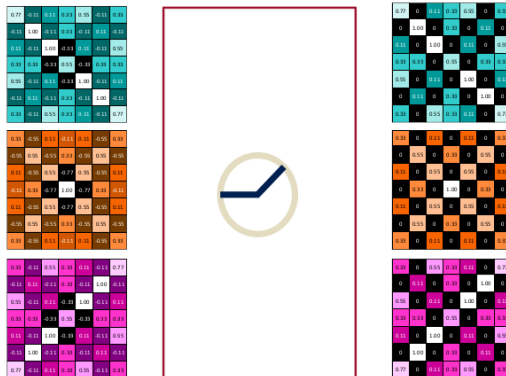


Rectified Linear Units (ReLUs)

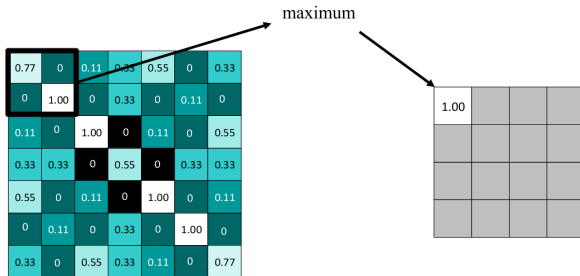


ReLU Layer

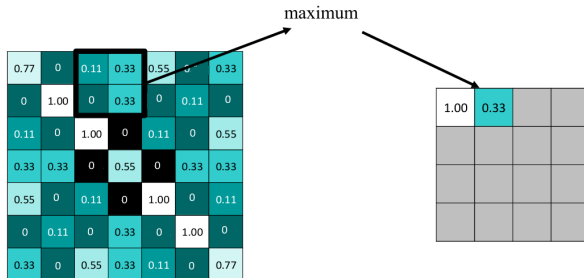
- ▶ A stack of images becomes a stack of images with **no negative values**.



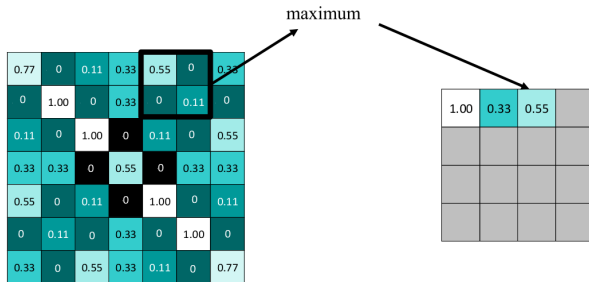
Pooling: Shrinking the Image Stack



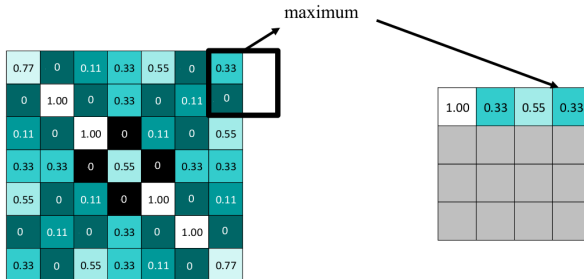
Pooling: Shrinking the Image Stack



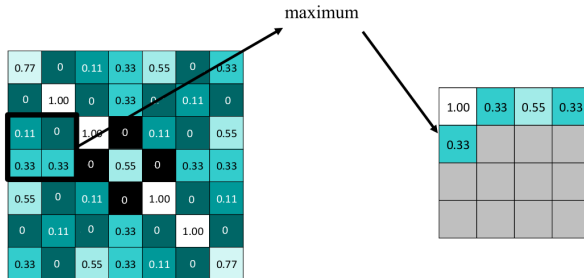
Pooling: Shrinking the Image Stack



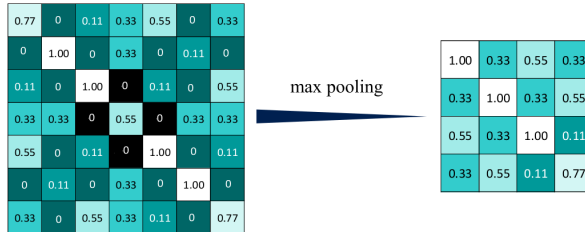
Pooling: Shrinking the Image Stack



Pooling: Shrinking the Image Stack



Pooling: Shrinking the Image Stack

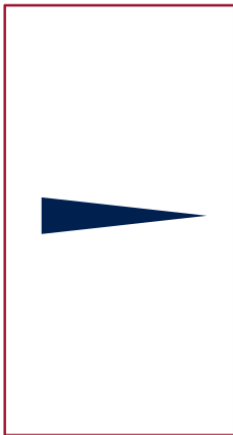


Repeat For All the Filtered Images

0.77	0	0.33	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.33	0
0.33	0	1.00	0	0.33	0	0.33
0.33	0.33	0	0.55	0	0.33	0.33
0.33	0	0.33	0	1.00	0	0.33
0	0.33	0	0.33	0	1.00	0
0.33	0	0.33	0.33	0.33	0	0.77

0.33	0	0.33	0	0.33	0	0.33
0	0.55	0	0.33	0	0.33	0
0.33	0	0.33	0	0.55	0	0.33
0	0.33	0	1.00	0	0.33	0
0.33	0	0.33	0	0.55	0	0.33
0	0.55	0	0.33	0	0.33	0
0.33	0	0.33	0	0.33	0	0.33

0.33	0	0.33	0.33	0.33	0	0.77
0	0.33	0	0.33	0	1.00	0
0.33	0	0.33	0	1.00	0	0.33
0.33	0.33	0	0.55	0	0.33	0.33
0.33	0	1.00	0	0.33	0	0.55
0	1.00	0	0.33	0	0.33	0
0.77	0	0.33	0.33	0.33	0	0.33



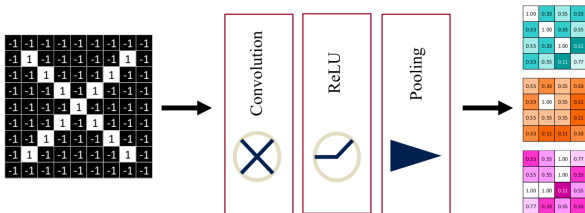
1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

0.55	0.33	0.55	0.33
0.33	1.00	0.55	0.11
0.55	0.55	0.55	0.11
0.33	0.11	0.11	0.33

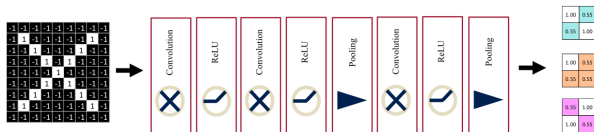
0.33	0.55	1.00	0.77
0.55	0.55	1.00	0.33
1.00	1.00	0.11	0.55
0.77	0.33	0.55	0.33

Layers Get Stacked

- ▶ The output of one becomes the input of the next.

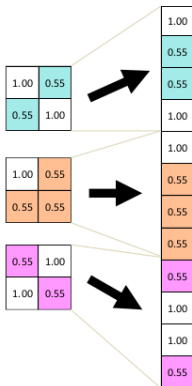


Deep Stacking



Fully Connected Layer

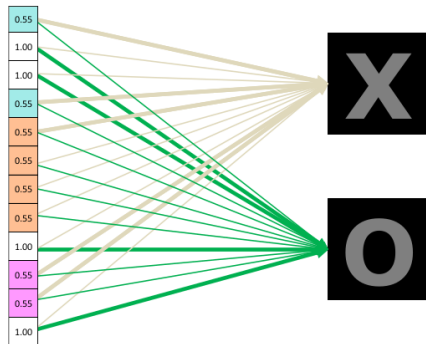
- ▶ **Flattening** the outputs before giving them to the **fully connected layer**.



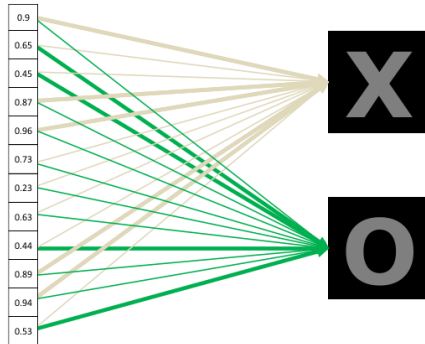
Fully Connected Layer



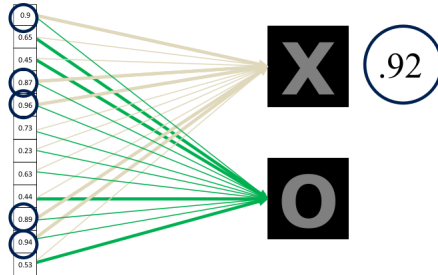
Fully Connected Layer



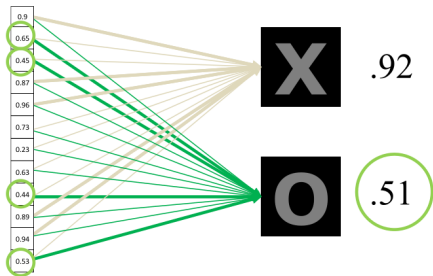
Fully Connected Layer



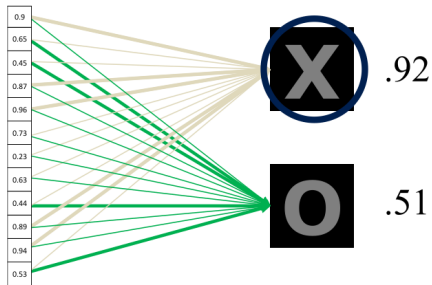
Fully Connected Layer



Fully Connected Layer

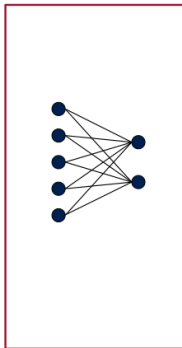


Fully Connected Layer

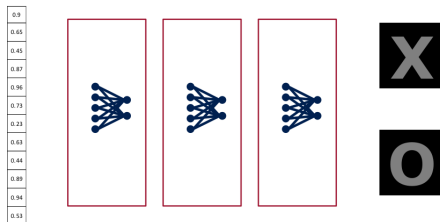


Fully Connected Layer

0.9
0.65
0.45
0.87
0.96
0.73
0.23
0.63
0.44
0.89
0.94
0.53



Fully Connected Layer



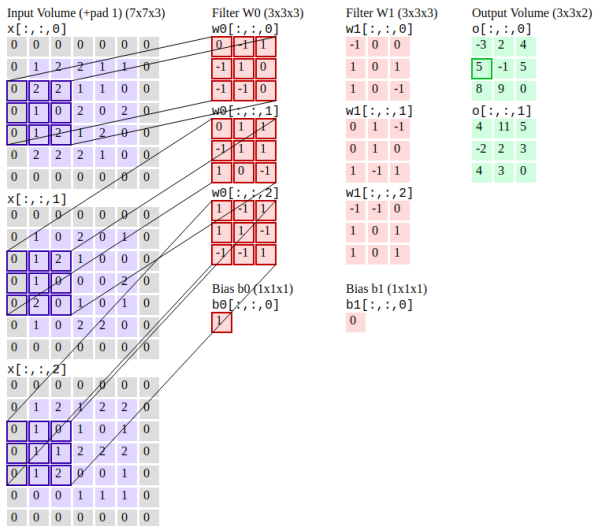
Putting It All Together





One more example

- ▶ A **conv layer**.
- ▶ Computes **2 feature maps**.
- ▶ Filters: **3x3** with stride of 2.
- ▶ Input tensor shape: **[7, 7, 3]**.
- ▶ Output tensor shape: **[3, 3, 2]**.



[<http://cs231n.github.io/convolutional-networks>]





CNN in TensorFlow



CNN in TensorFlow (1/7)

- ▶ A CNN for the MNIST dataset with the following network.
- ▶ Conv. layer 1: computes 32 feature maps using a 5x5 filter with ReLU activation.
- ▶ Pooling layer 1: max pooling layer with a 2x2 filter and stride of 2.
- ▶ Conv. layer 2: computes 64 feature maps using a 5x5 filter.
- ▶ Pooling layer 2: max pooling layer with a 2x2 filter and stride of 2.
- ▶ Dense layer: densely connected layer with 1024 neurons.
- ▶ Softmax layer



CNN in TensorFlow (2/7)

- ▶ **Conv. layer 1**: computes **32 feature maps** using a **5x5 filter** with ReLU activation.
- ▶ Padding **same** is added to **preserve width and height**.
- ▶ Input tensor shape: `[batch_size, 28, 28, 1]`
- ▶ Output tensor shape: `[batch_size, 28, 28, 32]`

```
# MNIST images are 28x28 pixels, and have one color channel: [28, 28, 1]
```

```
tf.keras.layers.Conv2D(kernel_size=5, filters=32, activation='relu', padding='same',  
                        input_shape=[28, 28, 1])
```



CNN in TensorFlow (3/7)

- ▶ **Pooling layer 1:** max pooling layer with a 2x2 filter and stride of 2.
- ▶ Input tensor shape: [batch_size, 28, 28, 32]
- ▶ Output tensor shape: [batch_size, 14, 14, 32]

```
tf.keras.layers.MaxPooling2D(pool_size=2, strides=2)
```



CNN in TensorFlow (4/7)

- ▶ **Conv. layer 2**: computes **64 feature maps** using a **5x5 filter**.
- ▶ Padding **same** is added to **preserve width and height**.
- ▶ Input tensor shape: `[batch_size, 14, 14, 32]`
- ▶ Output tensor shape: `[batch_size, 14, 14, 64]`

```
tf.keras.layers.Conv2D(kernel_size=5, filters=64, activation='relu', padding='same')
```



CNN in TensorFlow (5/7)

- ▶ **Pooling layer 2:** max pooling layer with a 2x2 filter and stride of 2.
- ▶ Input tensor shape: `[batch_size, 14, 14, 64]`
- ▶ Output tensor shape: `[batch_size, 7, 7, 64]`

```
tf.keras.layers.MaxPooling2D(pool_size=2, strides=2)
```



CNN in TensorFlow (6/7)

- ▶ **Flatten** tensor into a batch of vectors.
 - Input tensor shape: `[batch_size, 7, 7, 64]`
 - Output tensor shape: `[batch_size, 7 * 7 * 64]`

```
tf.keras.layers.Flatten()
```

- ▶ **Dense layer**: densely connected layer with **1024 neurons**.
 - Input tensor shape: `[batch_size, 7 * 7 * 64]`
 - Output tensor shape: `[batch_size, 1024]`

```
tf.keras.layers.Dense(1024, activation='relu')
```

- ▶ **Softmax layer**: softmax layer with **10 neurons**.
 - Input tensor shape: `[batch_size, 1024]`
 - Output tensor shape: `[batch_size, 10]`

```
tf.keras.layers.Dense(10, activation='softmax')
```



CNN in TensorFlow (7/7)

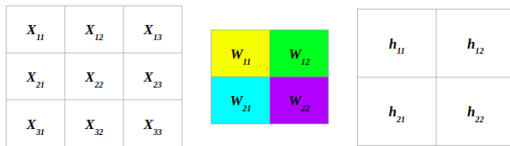
```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(kernel_size=5, filters=32, activation='relu', padding='same',
                           input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPooling2D(pool_size=2, strides=2),
    tf.keras.layers.Conv2D(kernel_size=5, filters=64, activation='relu', padding='same'),
    tf.keras.layers.MaxPooling2D(pool_size=2, strides=2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1024, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```



Training CNNs

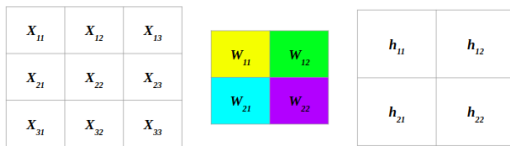
Training CNN (1/4)

- ▶ Let's see how to use **backpropagation** on a **single convolutional layer**.
- ▶ Assume we have an input X of size 3×3 and a **single filter** W of size 2×2 .
- ▶ **No padding** and **stride = 1**.
- ▶ It generates an **output** H of size 2×2 .



Training CNN (2/4)

► Forward pass



$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

Training CNN (3/4)

► Backward pass

► E is the error: $E = E_{h_{11}} + E_{h_{12}} + E_{h_{21}} + E_{h_{22}}$

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

w_{11}	w_{12}
w_{21}	w_{22}

h_{11}	h_{12}
h_{21}	h_{22}

$$\frac{\partial E}{\partial w_{11}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial w_{11}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial w_{11}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial w_{11}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial w_{11}}$$

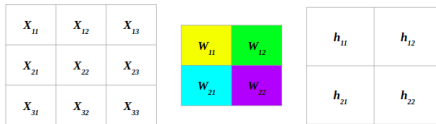
$$\frac{\partial E}{\partial w_{12}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial w_{12}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial w_{12}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial w_{12}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial w_{12}}$$

$$\frac{\partial E}{\partial w_{21}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial w_{21}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial w_{21}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial w_{21}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial w_{21}}$$

$$\frac{\partial E}{\partial w_{22}} = \frac{\partial E_{h_{11}}}{\partial h_{11}} \frac{\partial h_{11}}{\partial w_{22}} + \frac{\partial E_{h_{12}}}{\partial h_{12}} \frac{\partial h_{12}}{\partial w_{22}} + \frac{\partial E_{h_{21}}}{\partial h_{21}} \frac{\partial h_{21}}{\partial w_{22}} + \frac{\partial E_{h_{22}}}{\partial h_{22}} \frac{\partial h_{22}}{\partial w_{22}}$$

Training CNN (4/4)

- Update the wights W



$$W_{11}^{(\text{next})} = W_{11} - \eta \frac{\partial E}{\partial W_{11}}$$

$$W_{12}^{(\text{next})} = W_{12} - \eta \frac{\partial E}{\partial W_{12}}$$

$$W_{21}^{(\text{next})} = W_{21} - \eta \frac{\partial E}{\partial W_{21}}$$

$$W_{22}^{(\text{next})} = W_{22} - \eta \frac{\partial E}{\partial W_{22}}$$

Summary



Summary

- ▶ Receptive fields and filters
- ▶ Convolution operation
- ▶ Padding and strides
- ▶ Pooling layer
- ▶ Flattening, dropout, dense



Reference

- ▶ Tensorflow and Deep Learning without a PhD
<https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist>
- ▶ Ian Goodfellow et al., Deep Learning (Ch. 9)
- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 14)

Questions?