Recall we can apply several filters to the same input and stack their outputs together. E.g.

- To get a complete convolutional layer we pass the elements of the output through a nonlinearity, usually after adding a bias.
  - Kernel weights $w^{[1]}$, input $a^{[0]}$, bias/offset $b^{[1]}$ (weights $w^{[1]}$ and bias $b^{[1]}$ are unknown parameters that need to be learned).
  - After convolution output is $w^{[1]} * a^{[0]}$
  - Add bias to get $z^{[1]} = w^{[1]} * a^{[0]} + b^{[1]}$
  - Final output $a^{[1]} = g(z^{[1]})$, for nonlinear activation function $g(\cdot)$. Note: $g(\cdot)$ is applied separately to each element of $z^{[1]}$. 

![Diagram](image-url)
Choice of Activation Function $g(\cdot)$

- **ReLU (Rectified Linear Unit)**
  
  $g(x) = \begin{cases} 
  x & x \geq 0 \\
  0 & x < 0 
  \end{cases}$

- Almost universally used nowadays (older choices were sigmoid and tanh). Quick to compute, observed to work pretty well.

- But can lead to “dead” neurons where output is always zero → **leaky ReLU**
Combining Convolutional Layers

* We can use the output from one convolution layer as the input to another convolution layer

* E.g. Suppose input to first layer is $32 \times 32 \times 3$ and convolve this with 16 kernels of size $3 \times 3 \times 3 \rightarrow$ output is $30 \times 30 \times 16$

* Now use this $30 \times 30 \times 16$ output as input to a second layer with 8 kernels of size $3 \times 3 \times 16 \rightarrow$ output is $28 \times 28 \times 8$ tensor

* All layers use ReLU activation function. Stride is 1.

* Typical way of drawing this schematically:

![Diagram showing convolution layers](image)

Notes:

* “conv $3 \times 3, 16$” means convolutional layer with $3 \times 3$ kernel and 16 output channels.

* Number of channels in each kernel must match number of input channels e.g. $3 \times 3 \times 3$ for 3 input channels and $3 \times 3 \times 16$ for 16 input channels, no choice here. So usually abbreviate to $3 \times 3$.

* Depth of cube roughly indicates #output channels.
Combining Convolutional Layers

Some more notes:

* No padding used, so output is smaller than input. Could keep the same using padding.
* Number of kernel weights/parameters for first layer is $16 \times 3 \times 3 \times 3 = 432$, and for second layer $8 \times 3 \times 3 \times 16 = 1152$
* Using equations:
  * Input $a^{[0]}$ to first layer, output is $a^{[1]} = g(w^{[1]} \ast a^{[0]} + b^{[1]})$
  * Input $a^{[1]}$ to second layer, output is $a^{[2]} = g(w^{[2]} \ast a^{[1]} + b^{[2]})$
where $w^{[1]}$, $w^{[2]}$ are layer kernel weights, $b^{[1]}$, $b^{[2]}$ layer bias parameters and $g(\cdot)$ is ReLU.
Pooling layers are used to reduce the size of matrices in tensor

E.g. Suppose want to downsample \(4 \times 4\) matrix to \(2 \times 2\) matrix:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 3 & 2 & 3 \\
3 & 2 & 1 & 4 \\
6 & 1 & 1 & 2 \\
\end{array}
\]

Use \textit{max-pooling} with \(2 \times 2\) block size and stride 2:

1. Partition input matrix into \(2 \times 2\) blocks, stride of 2 means blocks don’t overlap.
2. Calculate value of max element in each block.
3. Use max as value of corresponding output element.
E.g. Max-pooling with $3 \times 3$ block size and stride 2:

\[
\begin{array}{ccc}
1 & 2 & 3 \\
1 & 3 & 2 \\
3 & 2 & 1 \\
6 & 1 & 1 \\
\end{array}
\rightarrow
\begin{array}{c}
3 \\
\end{array}
\rightarrow
\begin{array}{c}
3 \\
\end{array}
\rightarrow
\begin{array}{c}
3 \\
\end{array}
\]

But mostly use stride=block size $\rightarrow$ no overlap between blocks.

Pooling block size and stride must be chosen compatible with size of input matrix.

As well as max-pooling there is average pooling $\rightarrow$ output is average of elements in a block. But rarely used.
**Down-sampling Using Strided Convolution**

* Recall that we can use strides $> 1$ in a convolutional layer → also reduces size of output

* E.g. Applying $2 \times 2$ kernel \[
\begin{pmatrix}
1 & -1 \\
1 & -1
\end{pmatrix}
\] with stride 2:

\[
\begin{array}{cccc}
1 & 1 & 2 & -1 \\
1 & 3 & -1 & 2 \\
3 & 2 & 1 & 4 \\
6 & 1 & 1 & 2
\end{array} \rightarrow
\begin{array}{c}
-3 \\

\end{array}
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 3 & 2 & 3 \\
3 & 2 & 1 & 4 \\
6 & 1 & 1 & 2
\end{array} \rightarrow
\begin{array}{c}
-3 \\
-2
\end{array}
\]

→ for $4 \times 4$ input the output is reduced to $2 \times 2$

* Often works well, e.g. see Striving For Simplicity: The All Convolutional Net [https://arxiv.org/pdf/1412.6806.pdf](https://arxiv.org/pdf/1412.6806.pdf)

* Not quite the same as using $(2,2)$ kernel with stride 1 and same padding followed by $(2,2)$ max-pooling:
  * $(2,2)$ kernel with stride 1 and same padding does 16 convolutions whereas $(2,2)$ kernel with stride 2 calcs only 4 convolutions (so faster, computationally cheaper)
  * Max-pooling combines info from all 4 convolutions involving $2 \times 2$ block whereas $(2,2)$ kernel with stride 2 only uses info from 1 convolution per $2 \times 2$ block (uses less info)
Fully-connected (FC) layer = one layer of MLP. Called dense layer in keras.

Each output is a function of a weighted sum of all of the inputs

- Input is vector $x$ (not a tensor or matrix). Output is $y = f(w^T x)$, $w$ are weights/parameters, $f(\cdot)$ is nonlinear function.

If input is output from a convolution layer, i.e. a tensor, need to flatten it before it can be used as input to FC layer.

- Flattening $\rightarrow$ take all elements of tensor and write them as a list/array
- E.g. two channels $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $\begin{bmatrix} 4 & 5 \\ 6 & 7 \end{bmatrix}$ $\rightarrow$ [1, 2, 3, 4, 4, 5, 6, 7].
**Fully-Connected Layer**

* A FC-layer can have multiple outputs e.g. Input $x$ and two output $y_1 = f(w^T x)$, $y_2 = f(\nu^T x)$. Here $w$ is weight vector for $y_1$, $\nu$ the weight vector for $y_2$.

* If input vector $x$ has $n$ elements and have $m$ outputs then FC-layer has $n \times m$ parameters.
  * Suppose have $h_0 \times w_0 \times c_0$ input and $h_1 \times w_1 \times c_1$ output.
  * Convolution layer has $c_1 \times k \times k \times c_0$ parameters for $k \times k$ kernel
  * FC-layer has $h_0 \times w_0 \times c_0 \times h_1 \times w_1 \times c_1$ parameters
  * $h_0 = w_0 = 32$, $c_0 = 32$, $h_1 = w_1 = 32$, $c_1 = 32$, conv $3 \times 3$ layer has 9216 parameters, FC layer has $10^9$ parameters.

* Common to use FC-layer as the last layer in a ConvNet i.e. the layer which generates the (smallish number of) final outputs.

* How to choose nonlinear function $f(\cdot)$?
  * Common choice: **softmax**.
  * Recall softmax = multi-class logistic regression model.
Convolutional Network Example

MNIST Dataset

* Training data: 60K images of handwritten digits 0-9. Test data 10K images
* Each image is $28 \times 28$ pixels, gray scale
* Task is to predict which digit an image shows.
* Widely studied, relatively easy task. Best performance to date is 99.8% accuracy using ConvNet

1https://en.wikipedia.org/wiki/MNIST_database#cite_note-Gradient-9
Convolutional Network Example

- Uses strides to downsample the image.
  - Input $28 \times 28 \times 1 \rightarrow 13 \times 13 \times 32 \rightarrow 6 \times 6 \times 64$
- Number of channels increases as we move through network ($1 \rightarrow 32 \rightarrow 64$), size of image decreases ($28 \times 28 \rightarrow 13 \times 13 \rightarrow 6 \times 6$)
- We use final softmax layer/logistic regression to map from ConvNet features to final output (flatten step not shown in schematic)
  - Output is $10 \times 1 \rightarrow$ there are 10 classes, corresponding to digits 0-9, elements of output vector are probability of each class. To make prediction pick the class with highest probability.
We’ll use Python keras package for ConvNets (its a front end to tensorflow)

```python
import numpy as np
from tensorflow import keras
from tensorflow.keras import regularizers
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout

num_classes = 10
input_shape = (28, 28, 1)

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

model = keras.Sequential()
#3x3 kernel with stride 2, 32 output channels.
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2,2), input_shape=input_shape, activation="relu"))
#3x3 kernel with stride 2, 64 output channels.
model.add(Conv2D(64, kernel_size=(3, 3), strides=(2,2), activation="relu"))
# use CNN output as input to a Logistic regression classifier. Regularise logistic loss with L2 penalty.
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax',activity_regularizer=regularizers.l2(0.01)))
model.summary()

model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=32, epochs=5, validation_split=0.2)

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss: %f accuracy: %f"%(score[0],score[1]))

Note: use regularisation on FC-layers but usually not on convolutional layers. Why?
## Typical output:

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d (Conv2D)</td>
<td>(None, 13, 13, 32)</td>
<td>320</td>
</tr>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 6, 6, 64)</td>
<td>18496</td>
</tr>
<tr>
<td>flatten (Flatten)</td>
<td>(None, 2304)</td>
<td>0</td>
</tr>
<tr>
<td>dense (Dense)</td>
<td>(None, 10)</td>
<td>23050</td>
</tr>
</tbody>
</table>

Total params: 41,866  
Trainable params: 41,866  
Non-trainable params: 0

Epoch 1/5  
3000/3000 [==================================] - 6s 2ms/step - loss: 0.1927 - accuracy: 0.9447 - val_loss: 0.0916 - val_accuracy: 0.9765

Epoch 2/5  
3000/3000 [==================================] - 6s 2ms/step - loss: 0.0788 - accuracy: 0.9788 - val_loss: 0.0755 - val_accuracy: 0.9814

Epoch 3/5  
3000/3000 [==================================] - 6s 2ms/step - loss: 0.0584 - accuracy: 0.9850 - val_loss: 0.0700 - val_accuracy: 0.9820

Epoch 4/5  
3000/3000 [==================================] - 6s 2ms/step - loss: 0.0466 - accuracy: 0.9882 - val_loss: 0.0723 - val_accuracy: 0.9819

Epoch 5/5  
3000/3000 [==================================] - 6s 2ms/step - loss: 0.0384 - accuracy: 0.9908 - val_loss: 0.0616 - val_accuracy: 0.9858

Test loss: 0.051263 accuracy: 0.987100

* Achieves 98.7% accuracy on test data, model takes about 30s to train

* Baseline for comparison:
  * Logistic regression: 73s to train, achieves 92% accuracy
  * Kernelised SVM: 711s to train, achieves 94% accuracy
Convolutional Network Example

- Can also use dropouts rather than $L_2$ penalty for regularisation → using dropouts is popular in ConvNets

```python
model = keras.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), strides=(2,2), input_shape=input_shape, activation="relu"))
model.add(Conv2D(64, kernel_size=(3, 3), strides=(2,2), activation="relu"))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax'))
```

- Again, note that use regularisation on FC-layers but usually not on convolutional layers.
Convolutional Network Example

An alternative (but very similar) architecture:

- Use “same” padding in conv layers → output is same size as input.
- Use max-pool to downsample, stride=kernel size=2
- \(28 \times 28 \times 1 \rightarrow 28 \times 28 \times 32 \rightarrow 14 \times 14 \times 32 \rightarrow 14 \times 14 \times 64 \rightarrow 8 \times 8 \times 64\)
- Using same padding plus max-pool like this is currently popular ... but that might well change

Python keras code:

```python
model = keras.Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=input_shape, padding="same",activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(3, 3), padding="same", activation="relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(num_classes, activation='softmax',activity_regularizer=regularizers.l2(0.01)))
model.summary()
```
## Convolutional Network Example

* Typical output:

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d (Conv2D)</td>
<td>(None, 28, 28, 32)</td>
<td>320</td>
</tr>
<tr>
<td>max_pooling2d</td>
<td>(None, 14, 14, 32)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 14, 14, 64)</td>
<td>18496</td>
</tr>
<tr>
<td>max_pooling2d_1</td>
<td>(None, 7, 7, 64)</td>
<td>0</td>
</tr>
<tr>
<td>flatten (Flatten)</td>
<td>(None, 3136)</td>
<td>0</td>
</tr>
<tr>
<td>dense (Dense)</td>
<td>(None, 10)</td>
<td>31370</td>
</tr>
</tbody>
</table>

Total params: 50,186
Trainable params: 50,186
Non-trainable params: 0

Epoch 1/5
3000/3000 [==============================] - 21s 7ms/step - loss: 0.1490 - accuracy: 0.9565 - val_loss: 0.0627 - val_accuracy: 0.9854

Epoch 2/5
3000/3000 [==============================] - 22s 7ms/step - loss: 0.0570 - accuracy: 0.9850 - val_loss: 0.0527 - val_accuracy: 0.9886

Epoch 3/5
3000/3000 [==============================] - 22s 7ms/step - loss: 0.0432 - accuracy: 0.9898 - val_loss: 0.0567 - val_accuracy: 0.9849

Epoch 4/5
3000/3000 [==============================] - 22s 7ms/step - loss: 0.0345 - accuracy: 0.9920 - val_loss: 0.0504 - val_accuracy: 0.9877

Epoch 5/5
3000/3000 [==============================] - 21s 7ms/step - loss: 0.0284 - accuracy: 0.9941 - val_loss: 0.0474 - val_accuracy: 0.9901

Test loss: 0.044836 accuracy: 0.989100

* Achieves 98.9% accuracy on test data
* Takes 100s to train (longer than when use strides to downsample, why?)
Cross-validation

* Training by minimising cost function and using cross-validation to select hyperparameters (not just regularisation penalty but also number of convolutional output channels etc) is best practice
* But ...
* ... it often takes ages to train ConvNets. Even in above v easy example it takes a minute or so, with bigger networks and more data training can easily take days even with a good GPU rig
* So $k$-fold cross-validation usually impractical, just takes too long
* Instead often just keep a hold-out test set and use that to evaluate hyperparameter choices. Also often only evaluate only a few hyperparameter values as otherwise takes too long.
* Its not great, but we have little choice. Also means you can see many conflicting/random views on web for how to approach the same ML task.