Understanding Neural Networks
Part II: Convolutional Layers and Collaborative Filters

Nick Winovich
Department of Mathematics
Purdue University
July 2018
1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
While fully-connected layers provide an effective tool for analyzing general data, the associated dense weight matrices can be inefficient to work with. Fully-connected layers also have no awareness of spatial information (consider reindexing the dataset inputs).

When working with data which is spatially structured (e.g. images, function values on a domain, etc.), convolutional layers provide an efficient, spatially aware approach to data processing.

Another key advantage to using convolutional layers is the fact that hardware accelerators, such as GPUs, are capable of applying the associated convolutional filters extremely efficiently by design.
The key concept behind convolutional network layers is that of filters/kernels. These filters consist of small arrays of trainable weights which are typically arranged as squares or rectangles.

- Though shaped like matrices, the multiplication between filter weights and input values is performed element-wise.

- Filters are designed to slide across the input values to detect spatial patterns in local regions; by combining several filters in series, patterns in larger regions can also be identified.
Example: Convolutional Layer (with Stride=2)

\[ y_{11} = f\left( w_{11} x_{11} + w_{12} x_{12} + w_{21} x_{21} + w_{22} x_{22} + b \right) \]
Example: Convolutional Layer (with Stride=2)

\[ y_{12} = f \left( w_{11} x_{13} + w_{12} x_{14} + w_{21} x_{23} + w_{22} x_{24} + b \right) \]
Example: Convolutional Layer (with Stride=2)

\[ y_{21} = f \left( w_{11} x_{31} + w_{12} x_{32} 
+ w_{21} x_{41} + w_{22} x_{42} + b \right) \]
Example: Convolutional Layer (with Stride=2)

\[ y_{22} = f \left( w_{11} x_{33} + w_{12} x_{34} ight. \]
\[ + w_{21} x_{43} + w_{22} x_{44} + b \)
The bias term and activation function have been omitted for brevity
Floating Point Operation Count

For a convolutional layer with filter of size $k \times k$ applied to a two dimensional input array with resolution $R \times R$, we have:

- $k^2 R^2$ multiplication ops between filter weights and inputs
- $(k^2 - 1) R^2$ addition ops to sum the $k^2$ values in each position
- $R^2$ addition ops for adding the bias term $b$ to each entry

$$\approx 2 k^2 R^2$$ FLOPs

The true FLOP count depends on the choice of stride and padding; but the count is generally close to the upper-bound given above.
Transposed convolutional layers play a complementary role to standard convolutional layers and are commonly used to increase the spatial resolution of data/features.

As the name suggests, the matrix which defines this network layer is precisely the transpose of a standard convolutional layer.

\[
\begin{align*}
y_{11} &= f(w_{11} \cdot x_{11} + b) \\
y_{12} &= f(w_{12} \cdot x_{11} + b) \\
y_{21} &= f(w_{21} \cdot x_{11} + b) \\
y_{22} &= f(w_{22} \cdot x_{11} + b)
\end{align*}
\]
Matrix Representation

\[
\begin{bmatrix}
    y_{11} \\
    y_{12} \\
    y_{13} \\
    y_{14} \\
    y_{21} \\
    y_{22} \\
    y_{23} \\
    y_{24} \\
    y_{31} \\
    y_{32} \\
    y_{33} \\
    y_{34} \\
    y_{41} \\
    y_{42} \\
    y_{43} \\
    y_{44}
\end{bmatrix}
= 
\begin{bmatrix}
    w_{11} & 0 & 0 & 0 \\
    w_{12} & 0 & 0 & 0 \\
    0 & w_{11} & 0 & 0 \\
    0 & w_{12} & 0 & 0 \\
    w_{21} & 0 & 0 & 0 \\
    w_{22} & 0 & 0 & 0 \\
    0 & w_{21} & 0 & 0 \\
    0 & w_{22} & 0 & 0 \\
    0 & 0 & w_{11} & 0 \\
    0 & 0 & w_{12} & 0 \\
    0 & 0 & 0 & w_{11} \\
    0 & 0 & 0 & w_{12} \\
    0 & 0 & w_{21} & 0 \\
    0 & 0 & w_{22} & 0 \\
    0 & 0 & 0 & w_{21} \\
    0 & 0 & 0 & w_{22}
\end{bmatrix}
\begin{bmatrix}
    x_{11} \\
    x_{12} \\
    x_{21} \\
    x_{22}
\end{bmatrix}
\]

* The bias term and activation function have been omitted for brevity
Up until now, we have only discussed convolutional layers between two arrays with a single channel. A convolutional layer between an input array with $N$ channels and an output array with $M$ channels can be defined by a collection of $N \cdot M$ distinct filters, with weight matrices $W^{(n,m)}$ for $n \in \{1, \ldots, N\}$ and $m \in \{1, \ldots, M\}$, which correspond to the connections between input and output channels.

Each output channel is also assigned a bias term, $b^{(m)} \in \mathbb{R}$ for $m \in \{1, \ldots, M\}$, and the final outputs for channel $m$ are given by:

$$y^{(m)} = f \left( \sum_n W^{(n,m)} x^{(n)} + b^{(m)} \right)$$

The weight matrices $W^{(n,m)}$ typically correspond to filter weights $w^{(n,m)}$ of the same shape; we will see later how to generalize this.
A convolutional layer between an input array with $N$ channels and an output feature array with $M$ channels therefore consists of:

$$k^2 M N \text{ weights} + M \text{ biases}$$

Moreover, a calculation analogous to that used for the single channel case shows that the FLOP count for the layer is:

$$\approx 2 k^2 R^2 M N \text{ FLOPs}$$

**Note:** The filter size $k$ must be kept relatively small in order to maintain a manageable number of trainable variables and FLOPs.
While small filters may appear capable of only local detection, when used in series much larger patterns can be also be found.

The *receptive fields*, or regions of influence, for feature values later in the network are much larger than those at the beginning.
Hardware accelerators, such as GPUs, leverage the availability of thousands of cores to quickly compute the matrix-vector products associated with a convolutional layer in parallel.

Weight matrices for convolutional layers are extremely sparse, highly structured, and have only a handful of distinct values.

Specialized libraries exist with GPU-optimized implementations of the computational “primitives” used for these calculations:

**cuDNN: Efficient Primitives for Deep Learning**

It is possible to train networks using half-precision (i.e. 16-bit) fixed-point number representations without losing the accuracy achieved by single-precision floating-point representations.

This is possible in part due to the use of stochastic rounding:

\[
\text{Round}(x) = \begin{cases} 
\lfloor x \rfloor & \text{with probability } 1 - \frac{x - \lfloor x \rfloor}{\varepsilon} \\
\lfloor x \rfloor + \varepsilon & \text{with probability } \frac{x - \lfloor x \rfloor}{\varepsilon}
\end{cases}
\]
When defining convolutional layers, it is also necessary to specify how quickly, and to what extent, the filter slides across the inputs; these properties are controlled by *stride* and *padding* parameters.

- **A horizontal stride $I$ and vertical stride $J$ results in a filter which moves across rows in steps of $I$, e.g. $x_{1,1}, x_{1,1+I}, x_{1,1+2I}$, etc., and skips down rows by steps of $J$ once the current row ends.**

- Padding is used to determine which positions are admissible for the filter (e.g. when should the filter proceed to the next row).
  - **Same padding**: zeros are added to pad the array if necessary.
  - **Valid padding**: the filter is only permitted to continue to positions where all of its values fit entirely inside the array.
Example: Stride=1 with Valid Padding
Example: Stride=1 with Valid Padding
Example: Stride=1 with Valid Padding
Example: Stride=1 with Valid Padding
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding

![Diagram showing an example of Stride=1 with Same Padding in neural networks.](image-url)
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding

SIAM@Purdue 2018 - Nick Winovich
Understanding Neural Networks: Part II
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding
Example: Stride=1 with Same Padding
Same Padding

Same padding ensures that every input value is included, but also adds zeros near the boundary which are not in the original input.

Valid Padding

Valid padding only uses values from the original input; however, when the data resolution is not a multiple of the stride, some boundary values are ignored entirely in the feature calculation.
Additional references for visualizing and understanding the concepts of stride and padding in convolutional layers are:

- A guide to convolution arithmetic for deep learning by Vincent Dumoulin and Francesco Visin (2016)
  
  https://arxiv.org/abs/1603.07285

- The associated GitHub page with animations and source files:
  
  https://github.com/vdumoulin/conv_arithmetic/
Outline

1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
As was shown earlier, convolutional layers with non-trivial stride result in a reduction in spatial resolution. In some applications, performance can be improved by instead using a convolution with stride 1 followed by a dedicated downsampling procedure:

- **Max Pooling**: filter shape, strides, and padding are specified and the maximum value under the filter is returned for each position.

- **Average Pooling**: essentially the same as max pooling, but returns the average of the values under the filter.
Similarly, transposed convolutional layers can be used to increase the spatial resolution. However, it may be helpful to instead use a convolution with stride 1 and a dedicated upsampling procedure:

- **Bilinear/Bicubic Interpolation**: used to perform upsampling when the result is expected to have smooth, continuous values

- **Nearest-neighbor Interpolation**: useful for upsampling when the result is expected to have sharp boundaries or discontinuities
As the spatial resolution of features is decreased/downsampled, the channel count is typically increased to help avoid reducing the overall size of the information stored in features too rapidly.
- Similarly, the channel counts of features are typically decreased whenever the spatial resolution is increased/upsampled.
# Input Shape = [None, 64, 64, 1]

# CONV: [None, 64, 64, 1] --> [None, 64, 64, 4]
h = tf.layers.conv2d(x, 4, 3, padding="same",
                      activation=tf.nn.relu)

# POOL: [None, 64, 64, 4] --> [None, 32, 32, 4]
h = tf.layers.max_pooling2d(h, 3, 2, padding="same")

# CONV: [None, 32, 32, 4] --> [None, 30, 30, 8]
h = tf.layers.conv2d(h, 8, 3, padding="valid",
                      activation=tf.nn.relu)

# POOL: [None, 30, 30, 8] --> [None, 15, 15, 8]
h = tf.layers.max_pooling2d(h, 2, 2, padding="same")
# Example Implementation: Transposed Convolution

```python
# Shortened names for brevity
cnv2d_transpose = tf.layers.conv2d_transpose
lrelu = tf.nn.leaky_relu

# Input Shape = [None, 4, 4, 128]

# TCONV: [None, 4, 4, 128] --> [None, 8, 8, 64]
h = cnv2d_transpose(x, 64, 3, strides=(2, 2),
                     padding="same", activation=lrelu)

# TCONV: [None, 8, 8, 64] --> [None, 17, 17, 32]
h = cnv2d_transpose(h, 32, 3, strides=(2, 2),
                     padding="valid", activation=lrelu)
```
# Shortened names for brevity
bilinear = tf.image.ResizeMethod.BILINEAR
lrelu = tf.nn.leaky_relu

# Input Shape = [None, 4, 4, 128]

# CONV: [None, 4, 4, 128] --> [None, 4, 4, 64]
h = tf.layers.conv2d(x, 64, 3, padding="same",
activation=lrelu)

# INTERP: [None, 4, 4, 64] --> [None, 8, 8, 64]
h = tf.image.resize_images(h, [8,8], method=bilinear)
Outline

1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
Outline

1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
Network layers can be systematically organized in blocks, or modules, which facilitate collaboration between different filters.

These modules provide a multi-scale, multimodal approach to processing input data and features throughout the network.
Inception v1 Block (naïve version)

Diagram from *Going deeper with convolutions*
Using 1x1 Filters for Dimension Reduction

- The pooling layer in this “naïve” version of the module produces features with the same number of channels as the original input.

- To balance the impact of each component in the module, it is natural to assign this channel count to features from each layer; when this channel count is relatively high, however, the layers with larger filters can become prohibitively expensive.

- Alternatively, $1 \times 1$ convolutional layers can be used as a form of dimension reduction to help limit the computational demand and balance the size of features produced by each component.
Inception v1 Block (with dimension reduction)

Diagram from *Going deeper with convolutions*

Diagram showing the components and connections of an Inception v1 Block, including 1x1 convolutions, 3x3 convolutions, 5x5 convolutions, and 1x1 convolutions, along with a 3x3 max pooling layer.
Factoring Large Filters for Improved Efficiency

While dimension reduction can be used to improve efficiency in part, the large filter sizes still pose a problem. A compromise between the full expressiveness of large filters and the efficiencies of small filters is to “factor” the larger filters into smaller, more efficient ones.

- This factorization can be approximated by using a series/tower of consecutive convolutional layers with smaller filters.
- By construction, the resulting component produces features with receptive fields identical to those of the original layer.
Diagram from *Rethinking the Inception Architecture for Computer Vision*
Definition for `inception_v2(x, chans, name)`

```python
conv2d = tf.layers.conv2d; lrelu = tf.nn.leaky_relu;

""" 1x1 CONV + 3x3 CONV """
h1 = conv2d(x, chans, 1, activation = lrelu,
            padding = "same", name = name + "_1a")
h1 = conv2d(h1, chans, 3, activation = lrelu,
            padding = "same", name = name + "_1b")

""" 1x1 CONV + 3x3 CONV + 3x3 CONV """
h2 = conv2d(x, chans, 1, activation = lrelu,
            padding = "same", name = name + "_2a")
h2 = conv2d(h2, chans, 3, activation = lrelu,
            padding = "same", name = name + "_2b")
h2 = conv2d(h2, chans, 3, activation = lrelu,
            padding = "same", name = name + "_2c")
```
Definition for inception_v2(x, chans, name)

""" 3x3 MAX POOL + 1x1 CONV """

```python
h3 = tf.layers.max_pooling2d(x, 3, 1, padding = "same")
h3 = conv2d(h3, chans, 1, activation = lrelu,
            padding = "same", name = name + "_3")
```

""" 1x1 CONV """

```python
h4 = conv2d(x, chans, 1, activation = lrelu,
            padding = "same", name = name + "_4")
```

```python
h = tf.concat([h1,h2,h3,h4],3)
```
“If our main goal is to factorize the linear part of the computation, would it not suggest to keep linear activations in the first layer? We have ran several control experiments (for example see figure 2) and using linear activation was always inferior to using rectified linear units in all stages of the factorization.” (Rethinking the Inception Architecture)

- The motivation of “factoring” large filters suggests only using activations for the final layer of each series/tower in a block

- Including activation functions in the intermediate block layers as well tends to improve the network’s performance in practice
Outline

1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks
Residual Learning


Instead of training layers to produce the full set of features $H(x)$ directly, we can design network layers to learn residual changes:

$$\mathcal{F}(x) = H(x) - x$$

- This can be done by including shortcuts, or skip connections, which allow features to pass through without modification.
- These skip connections provide a way for the network to determine how many active layers are actually necessary.
Diagram from *Deep Residual Learning for Image Recognition*
Define ResNet block with 2-layer shortcut

```python
def resnet_block(x, chans, kernel_size):
    # Layer 1
    r = tf.layers.conv2d(x, chans, kernel_size,
                        padding="same", use_bias=False)
    r = tf.layers.batch_normalization(r)
    r = tf.nn.relu(r)
    # Layer 2
    r = tf.layers.conv2d(r, chans, kernel_size,
                        padding="same", use_bias=False)
    r = tf.layers.batch_normalization(r)
    # Shortcut
    h = tf.nn.relu(tf.add(r, x))
    return h
```
1. Convolutional Neural Networks
   - Convolutional Layers
   - Strides and Padding
   - Pooling and Upsampling

2. Advanced Network Design
   - Collaborative Filters
   - Residual Blocks
   - Dense Convolutional Blocks


“DenseNets exploit the potential of the network through feature reuse, yielding condensed models that are easy to train and highly parameter efficient.” *(Huang et al.)*

A variation on the underlying idea behind skip connections is provided by passing the unmodified features of several previous network layers to the current layer all at once
Diagram from *Densely Connected Convolutional Networks*
The input to the $l^{th}$ layer of a dense block consists of features from all previous layers: $[x_0, x_1, \ldots, x_{l-1}]$.

The new features $x_l$ produced by the $l^{th}$ block layer are the outputs of the $3 \times 3$ convolution.

These new features are concatenated with the previous features and passed to the next layer.
Implementation of DenseNet Blocks

""" BN-ReLU-Conv layers within DenseNet blocks """

def block_layer(x, chans):
    h = tf.layers.batch_normalization(x)
    h = tf.nn.relu(h)
    h = tf.layers.conv2d(h, chans, 3, padding = "same")
    return tf.concat([x,h], 3)

""" Define a DenseNet block with k layers """

def block(x, chans, k):
    for i in range(0,k):
        x = block_layer(x, chans)
    return x