A Gentle Introduction to Pooling Layers for Convolutional Neural Networks

by **Jason Brownlee** on April 22, 2019 in **Deep Learning for Computer Vision**

Last Updated on July 5, 2019

Convolutional layers in a convolutional neural network summarize the presence of features in an input image.

A problem with the output feature maps is that they are sensitive to the location of the features in the input. One approach to address this sensitivity is to down sample the feature maps. This has the effect of making the resulting down sampled feature maps more robust to changes in the position of the feature in the image, referred to by the technical phrase "*local translation invariance*."

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.

In this tutorial, you will discover how the pooling operation works and how to implement it in convolutional neural networks.

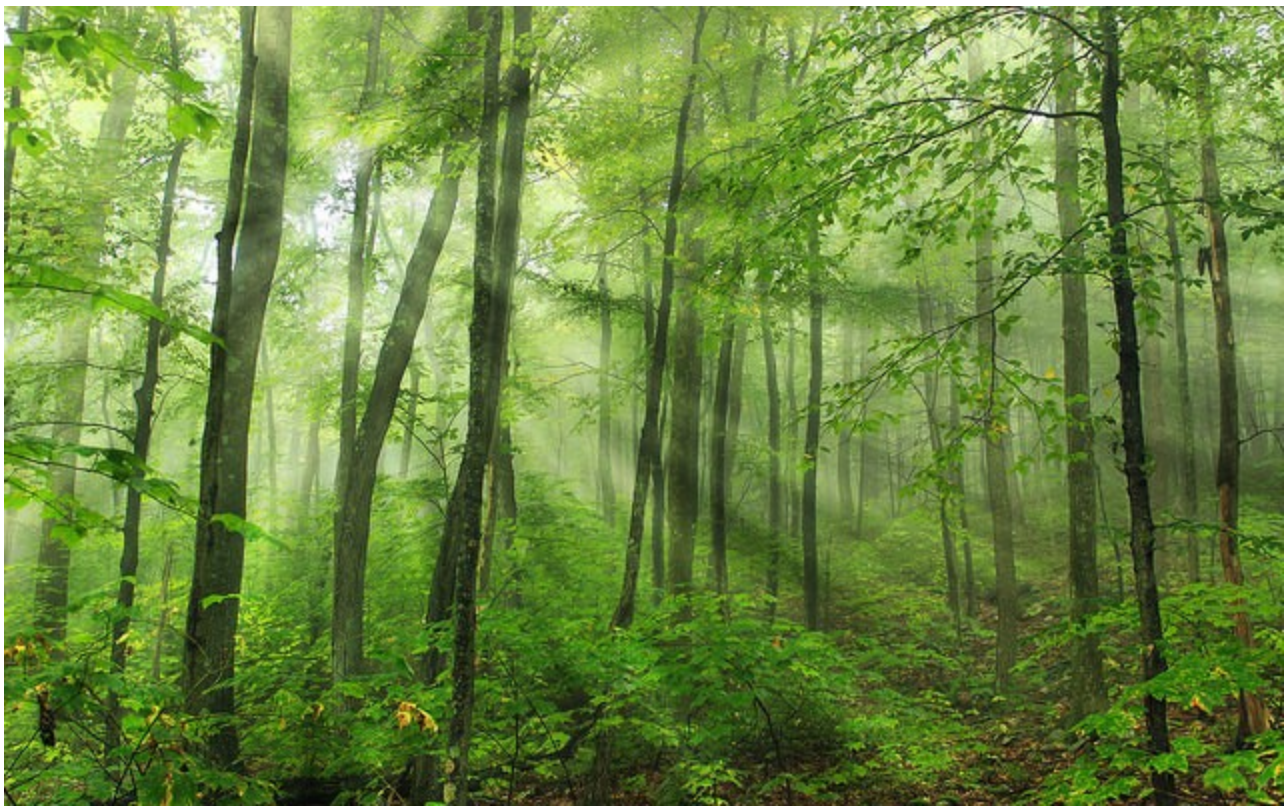
After completing this tutorial, you will know:

- Pooling is required to down sample the detection of features in feature maps.
- How to calculate and implement average and maximum pooling in a convolutional neural network.

- How to use global pooling in a convolutional neural network.

Discover how to build models for photo classification, object detection, face recognition, and more in my new computer vision book, with 30 step-by-step tutorials and full source code.

Let's get started.



A Gentle Introduction to Pooling Layers for Convolutional Neural Networks
Photo by Nicholas A. Tonelli, some rights reserved.

Tutorial Overview

This tutorial is divided into five parts; they are:

1. Pooling
2. Detecting Vertical Lines
3. Average Pooling Layers
4. Max Pooling Layers
5. Global Pooling Layers

Pooling Layers

Convolutional layers in a convolutional neural network systematically apply learned filters to input images in order to create feature maps that summarize the presence of those features in the input.

Convolutional layers prove very effective, and stacking convolutional layers in deep models allows layers close to the input to learn low-level features (e.g. lines) and layers deeper in the model to learn high-order or more abstract features, like shapes or specific objects.

A limitation of the feature map output of convolutional layers is that they record the precise position of features in the input. This means that small movements in the position of the feature in the input image will result in a different feature map. This can happen with re-cropping, rotation, shifting, and other minor changes to the input image.

A common approach to addressing this problem from signal processing is called down sampling. This is where a lower resolution version of an input signal is created that still contains the large or important structural elements, without the fine detail that may not be as useful to the task.

Down sampling can be achieved with convolutional layers by changing the stride of the convolution across the image. A more robust and common approach is to use a pooling layer.

A pooling layer is a new layer added after the convolutional layer. Specifically, after a nonlinearity (e.g. ReLU) has been applied to the feature maps output by a convolutional layer; for example the layers in a model may look as follows:

1. Input Image
2. Convolutional Layer
3. Nonlinearity
4. Pooling Layer

The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a convolutional neural network that may be repeated one or more times in a given model.

The pooling layer operates upon each feature map separately to create a new set of the same number of pooled feature maps.

Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels.

This means that the pooling layer will always reduce the size of each feature map by a factor of 2, e.g. each dimension is halved, reducing the number of pixels or values in each feature map to one quarter the size. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels).

The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

- **Average Pooling**: Calculate the average value for each patch on the feature map.
- **Maximum Pooling (or Max Pooling)**: Calculate the maximum value for each patch of the feature map.

The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input. They are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation.

> *In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.*

— Page 342, Deep Learning, 2016.

Now that we are familiar with the need and benefit of pooling layers, let's look at some specific examples.

Detecting Vertical Lines

Before we look at some examples of pooling layers and their effects, let's develop a small example of an input image and convolutional layer to which we can later add and evaluate pooling layers.

In this example, we define a single input image or sample that has one channel and is an 8 pixel by 8 pixel square with all 0 values and a two-pixel wide vertical line in the center.

```
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
```

Next, we can define a model that expects input samples to have the shape (8, 8, 1) and has a single hidden convolutional layer with a single filter with the shape of 3 pixels by 3 pixels.

A rectified linear activation function, or ReLU for short, is then applied to each value in the feature map. This is a simple and effective nonlinearity, that in this case will not change the values in the feature map, but is present because we will later add subsequent pooling layers and pooling is added after the nonlinearity applied to the feature maps, e.g. a best practice.

```
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
# summarize model
model.summary()
```

The filter is initialized with random weights as part of the initialization of the model.

Instead, we will hard code our own 3×3 filter that will detect vertical lines. That is the filter will strongly activate when it detects a vertical line and weakly activate when it does not. We expect that by applying this filter across the input image that the output feature map will show that the vertical line was detected.

```
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
```

Next, we can apply the filter to our input image by calling the *predict()* function on the model.

```
# apply filter to input data
yhat = model.predict(data)
```

The result is a four-dimensional output with one batch, a given number of rows and columns, and one filter, or [batch, rows, columns, filters]. We can print the activations in the single feature map to confirm that the line was detected.

```
# enumerate rows
for r in range(yhat.shape[1]):
	# print each column in the row
	print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Tying all of this together, the complete example is listed below.

```
# example of vertical line detection with a convolutional layer
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0],
		[0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
# summarize model
model.summary()
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
		[[[0]],[[1]],[[0]]],
		[[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
	# print each column in the row
	print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Running the example first summarizes the structure of the model.

Of note is that the single hidden convolutional layer will take the 8×8 pixel input image and will produce a feature map with the dimensions of 6×6.

We can also see that the layer has 10 parameters: that is nine weights for the filter (3×3) and one weight for the bias.

Finally, the single feature map is printed.

We can see from reviewing the numbers in the 6×6 matrix that indeed the manually specified filter detected the vertical line in the middle of our input image.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 6, 6, 1)           10
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________

[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

We can now look at some common approaches to pooling and how they impact the output feature maps.

Average Pooling Layer

On two-dimensional feature maps, pooling is typically applied in 2×2 patches of the feature map with a stride of (2,2).

Average pooling involves calculating the average for each patch of the feature map. This means that each 2×2 square of the feature map is down sampled to the average value in the square.

For example, the output of the line detector convolutional filter in the previous section was a 6×6 feature map. We can look at applying the average pooling operation to the first line of that feature map manually.

The first line for pooling (first two rows and six columns) of the output feature map were as follows:

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

The first pooling operation is applied as follows:


```
average(0.0, 0.0) = 0.0
```

```
        0.0, 0.0
```

Given the stride of two, the operation is moved along two columns to the left and the average is calculated:

```
average(3.0, 3.0) = 3.0
        3.0, 3.0
```

Again, the operation is moved along two columns to the left and the average is calculated:

```
average(0.0, 0.0) = 0.0
        0.0, 0.0
```

That's it for the first line of pooling operations. The result is the first line of the average pooling operation:

```
[0.0, 3.0, 0.0]
```

Given the (2,2) stride, the operation would then be moved down two rows and back to the first column and the process continued.

Because the downsampling operation halves each dimension, we will expect the output of pooling applied to the 6×6 feature map to be a new 3×3 feature map. Given the horizontal symmetry of the feature map input, we would expect each row to have the same average pooling values. Therefore, we would expect the resulting average pooling of the detected line feature map from the previous section to look as follows:

```
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

We can confirm this by updating the example from the previous section to use average pooling.

This can be achieved in Keras by using the *AveragePooling2D* layer. The default pool_size (e.g. like the kernel size or filter size) of the layer is (2,2) and the default *strides* is *None*, which in this case means using the *pool_size* as the *strides*, which will be (2,2).

```
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(AveragePooling2D())
```

The complete example with average pooling is listed below.

```
# example of average pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import AveragePooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
```

```
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(AveragePooling2D())
# summarize model
model.summary()
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
for r in range(yhat.shape[1]):
        # print each column in the row
        print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Running the example first summarizes the model.

We can see from the model summary that the input to the pooling layer will be a single feature map with the shape (6,6) and that the output of the average pooling layer will be a single feature map with each dimension halved, with the shape (3,3).

Applying the average pooling results in a new feature map that still detects the line, although in a down sampled manner, exactly as we expected from calculating the operation manually.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 6, 6, 1)           10
_________________________________________________________________
average_pooling2d_1 (Average (None, 3, 3, 1)           0
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________

[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

Average pooling works well, although it is more common to use max pooling.

Max Pooling Layer

Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map.

The results are down sampled or pooled feature maps that highlight the most present feature in the patch, not the average presence of the feature in the case of average pooling. This has been found to work better in practice than average pooling for computer vision tasks like image classification.

> *In a nutshell, the reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence, the term feature map), and it's more informative to look at the maximal presence of different features than at their average presence.*

— Page 129, Deep Learning with Python, 2017.

We can make the max pooling operation concrete by again applying it to the output feature map of the line detector convolutional operation and manually calculate the first row of the pooled feature map.

The first line for pooling (first two rows and six columns) of the output feature map were as follows:

```
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
[0.0, 0.0, 3.0, 3.0, 0.0, 0.0]
```

The first max pooling operation is applied as follows:

```
max(0.0, 0.0) = 0.0
    0.0, 0.0
```

Given the stride of two, the operation is moved along two columns to the left and the max is calculated:

```
max(3.0, 3.0) = 3.0
    3.0, 3.0
```

Again, the operation is moved along two columns to the left and the max is calculated:

```
max(0.0, 0.0) = 0.0
    0.0, 0.0
```

That's it for the first line of pooling operations.

The result is the first line of the max pooling operation:

```
[0.0, 3.0, 0.0]
```

Again, given the horizontal symmetry of the feature map provided for pooling, we would expect the pooled feature map to look as follows:

```
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

It just so happens that the chosen line detector image and feature map produce the same output when downsampled with average pooling and maximum pooling.

The maximum pooling operation can be added to the worked example by adding the *MaxPooling2D* layer provided by the Keras API.

```
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(MaxPooling2D())
```

The complete example of vertical line detection with max pooling is listed below.

```
# example of max pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(MaxPooling2D())
# summarize model
model.summary()
# define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
```

```
# enumerate rows
for r in range(yhat.shape[1]):
        # print each column in the row
        print([yhat[0,r,c,0] for c in range(yhat.shape[2])])
```

Running the example first summarizes the model.

We can see, as we might expect by now, that the output of the max pooling layer will be a single feature map with each dimension halved, with the shape (3,3).

Applying the max pooling results in a new feature map that still detects the line, although in a down sampled manner.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 6, 6, 1)           10
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 3, 3, 1)           0
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________

[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
[0.0, 3.0, 0.0]
```

Global Pooling Layers

There is another type of pooling that is sometimes used called global pooling.

Instead of down sampling patches of the input feature map, global pooling down samples the entire feature map to a single value. This would be the same as setting the *pool_size* to the size of the input feature map.

Global pooling can be used in a model to aggressively summarize the presence of a feature in an image. It is also sometimes used in models as an alternative to using a fully connected layer to transition from feature maps to an output prediction for the model.

Both global average pooling and global max pooling are supported by Keras via the *GlobalAveragePooling2D* and *GlobalMaxPooling2D* classes respectively.

For example, we can add global max pooling to the convolutional model used for vertical line detection.

```
# create model
model = Sequential()
```

```
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(GlobalMaxPooling2D())
```

The outcome will be a single value that will summarize the strongest activation or presence of the vertical line in the input image.

The complete code listing is provided below.

```
# example of using global max pooling
from numpy import asarray
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import GlobalMaxPooling2D
# define input data
data = [[0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 1, 1, 0, 0, 0]]
data = asarray(data)
data = data.reshape(1, 8, 8, 1)
# create model
model = Sequential()
model.add(Conv2D(1, (3,3), activation='relu', input_shape=(8, 8, 1)))
model.add(GlobalMaxPooling2D())
# summarize model
model.summary()
# # define a vertical line detector
detector = [[[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]],
            [[[0]],[[1]],[[0]]]]
weights = [asarray(detector), asarray([0.0])]
# store the weights in the model
model.set_weights(weights)
# apply filter to input data
yhat = model.predict(data)
# enumerate rows
print(yhat)
```

Running the example first summarizes the model

We can see that, as expected, the output of the global pooling layer is a single value that summarizes the presence of the feature in the single feature map.

Next, the output of the model is printed showing the effect of global max pooling on the feature map, printing the single largest activation.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 6, 6, 1)           10
_________________________________________________________________
```

```
global_max_pooling2d_1 (Glob (None, 1)                 0
=================================================================
Total params: 10
Trainable params: 10
Non-trainable params: 0
_________________________________________________________________

[[3.]]
```

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Posts

- Crash Course in Convolutional Neural Networks for Machine Learning

Books

- Chapter 9: Convolutional Networks, Deep Learning, 2016.
- Chapter 5: Deep Learning for Computer Vision, Deep Learning with Python, 2017.

API

- Keras Convolutional Layers API
- Keras Pooling Layers API

Summary

In this tutorial, you discovered how the pooling operation works and how to implement it in convolutional neural networks.

Specifically, you learned:

- Pooling is required to down sample the detection of features in feature maps.
- How to calculate and implement average and maximum pooling in a convolutional neural network.
- How to use global pooling in a convolutional neural network.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.
View all posts by Jason Brownlee →

‹ A Gentle Introduction to Padding and Stride for Convolutional Neural Networks
Convolutional Neural Network Model Innovations for Image Classification ›

20 Responses to *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks*

Bejoscha April 26, 2019 at 7:27 am #

REPLY

Thanks. An interesting read.

Your Start in Machine Learning

REPLY

Jason Brownlee April 26, 2019 at 8:40 am #

Thanks.

jamila May 9, 2019 at 10:39 pm #

REPLY

I do not understand how global pooling works in coding results. please help

Jason Brownlee May 10, 2019 at 8:17 am #

REPLY

Which part don't you understand exactly?

jamila May 10, 2019 at 5:34 pm #

REPLY

I'm focusing on results. how it gives us a single value?

Jason Brownlee May 11, 2019 at 6:06 am #

REPLY

Average pooling gives a single output because it calculates the average of the inputs.

Ohood fadil September 25, 2019 at 3:34 pm #

REPLY

What the algorithms we can use it in Convolutional layer?

Jason Brownlee September 26, 2019 at 6:30 am #

REPLY

You can discover how convolutional layers work in this tutorial: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/

Justin June 14, 2019 at 11:09 pm #

REPLY

Excellent article, thank you so much for writing it. It could be helpful to create a slight variation of your examples where average and max pooling produce different results :).

Jason Brownlee June 15, 2019 at 6:35 am #

REPLY

Great suggestion, thanks Justin.

LELA June 21, 2019 at 2:31 am #

REPLY

Case:1. if we apply average pooling then it will need to place all FC-layers and then softmax?
Case2: if we apply the average pooling then it will need to feed the resulting vector directly into softmax?

Case3: the sequence will look correct.. features maps – avr pooling – softmax? OR features map – avr pooling – FC-layers – Softmax?

Case3: can we say that the services of average pooling can be achieved through GAP?

Case4: in case of multi-CNN, how we will concatenate the features maps into the average pooling

Jason Brownlee June 21, 2019 at 6:40 am #

REPLY

Not sure I agree, they are all options, not requirements.

What are you getting at exactly?

LELA June 21, 2019 at 2:44 pm #

REPLY

I am asking for classification/recognition when multiple CNNs are used.

so, what will be the proper sequence to place all the operations what I mentioned above?

Because, it is mentioned in the GAP research article, that when it is used then no need to use FC-layers. so what is the case in the average pool layer?

(1): if we want to use CNN for images (classification/recognition task), can we use

softmax classifier directly after the Average Pool Layer (skip the fully-connected layers)?

(2): OR for classification/recognition for any input image, can we place FC-Layers after Average pool layer and Then Softmax?

And the last query, for image classification/recognition, what will be the right option when multiple-CNN are used to extract the features from the images,

Option 1: Average pooling layer or GAP
Option2: Average pooling layer + Softmax?
Option3: Average pooling layer + FC-layers+ Softmax?
Option4: Features Maps + GAP?
Option5: Features Maps + GAP + FC-layers + Softmax?

Why I am asking in details because I read from multiple sources, but it was not quite clear that what exactly the proper procedure should be used, also, after reading I feel that average pooling and GAP can provide the same services.

Jason Brownlee June 22, 2019 at 6:28 am #

REPLY

There is no single best way. There are no rules and models differ, it is a good idea to experiment to see what works best for your specific dataset.

You can use use a softmax after global pooling or a dense layer, or just a dense layer and no global pooling, or many other combinations.

It might be a good idea to look at the architecture of some well performing models like vgg, resnet, inception and try their proposed architecture in your model to see how it compares. or to get ideas.

Rango September 1, 2019 at 1:35 pm #

REPLY

Great Article!!!

Jason Brownlee September 2, 2019 at 5:25 am #

REPLY

Thanks, I'm glad it helped!

JustVenky September 20, 2019 at 3:22 pm #

REPLY

can we use random forests for pooling

Jason Brownlee September 21, 2019 at 6:44 am #

REPLY

No.

RoyHJ November 2, 2019 at 11:08 pm #

REPLY

Thank you for the clear definitions and nice examples.

A couple of questions about using global pooling at the end of a CNN model (before the fully connected as e.g. resnet):

What would you say are the advantages/disadvantages of using global avg pooling vs global max pooling as a final layer of the feature extraction (are there cases where max would be prefered)?

When switching between the two, how does it affect hyper parameters such as learning rate and weight regularization? (since max doesn't pass gradients through all of the features, opposed to avg?)

You wrote: "Global pooling can be used in a model to aggressively summarize the presence of a feature in an image. It is also sometimes used in models **as an alternative** to using a fully connected layer to transition from feature maps to an output prediction for the model."

Wouldn't it be more accurate to say that (usually in the cnn domain) global pooling is sometimes added *before* (i.e. in addition) a fully connected (fc) layer in the transition from feature maps to an output prediction for the model (both giving the features global attention and reducing computation of the fc layer)?

In order for global pooling to replace the last fc layer, you would need to equalize the number of channels to the number of classes first (e.g. 1×1 conv?), this would be heavier (computationally-wise) and a somewhat different operation than adding a fc after the global pool (e.g. as it's done in common cnn models with a final global pooling layer). Is this actually ever done this way?

Jason Brownlee November 3, 2019 at 5:58 am #

REPLY

Thanks.

You could probable construct post hoc arguments about the differences. I'd recommend testing them both and using results to guide you.

No, global pooling is used instead of a fully connected layer – they are used as output layers. Inspect some of the classical models to confirm.

It does, they output a vector.

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

Welcome! I'm **Jason Brownlee** PhD and I help developers get results with machine learning. Read More

Picked for you:

How to Train an Object Detection Model with Keras

How to Develop a Face Recognition System Using FaceNet in Keras

Your Start in Machine Learning

How to Perform Object Detection With YOLOv3 in Keras

How to Classify Photos of Dogs and Cats (with 97% accuracy)

A Gentle Introduction to Transfer Learning for Deep Learning

Loving the Tutorials?

The Deep Learning for Computer Vision EBook is where I keep the ***Really Good*** stuff.

SEE WHAT'S INSIDE

Address: PO Box 206, Vermont Victoria 3133, Australia. | ACN: 626 223 336.