New York University: Machine Learning in Economics

Lecture 10: Convolutional Networks

Dr. George Lentzas

Introduction

Convolutional Neural Networks (CNNs) are specialized neural networks for processing data that have a grid-like structure.

Common examples are images (a two dimensional grid of pixels) and time series (a one dimensional grid of ordered values).

CNNs use a mathematical operation called a "convolution" which is a specialized kind of linear operation.

Introduction

A convolution is an operation on two functions of a real valued argument.

The convolution operation is typically denoted with an asterisk,

$$s(t) = (x * w)(t)$$

where

$$s(t) = \int x(\alpha)\, w(t - \alpha)\, d\alpha$$

which is equivalent to

$$s(t) = \int x(t - \alpha)\, w(\alpha)\, d\alpha$$

Introduction

The first argument is usually called the *input* and the second is called the *kernel*. The output is usually referred to as the *feature map*.

In machine learning, the input is usually a tensor (a multidimensional array of data) and the kernel is similarly a tensor of parameters that are *learned by the model*.

In practice convolutions are discrete (sum instead of an integral) and multidimensional. A discrete convolution can be viewed as multiplication by a matrix, with the matrix having several entries constrained to be equal to other entries.

Convolutions for Neural Networks

Convolution merges three key ideas that can help improve a neural network: (i) *sparse interactions*, (ii) *parameter sharing*, and (iii) *equivariant representations*.

An additional advantage is that convolutions allow us to work with inputs of variable sizes.

Sparse Interactions: by making the kernel smaller than the matrix, a convolution creates a feature map that reduces the dimensionality of the problem.

Parameter Sharing: one set of parameters is learned per kernel.

Pooling: allows us to "summarize" the output of the convolution and make it approximately invariant to small translations of the input.

Convolutions for Neural Networks

Convolutional nets ("convnets") work by combining several of these concepts (multiple times).

We start with the input and apply one or more convolutions to generate linear feature maps. Multiple kernels are usually applied to generate different types of features.

An activation function is then applied to the derived features to add non-linerarity to the model.

Usually ReLU activation functions are used. These are applied to element by element to the derived features.

Convolutions for Neural Networks

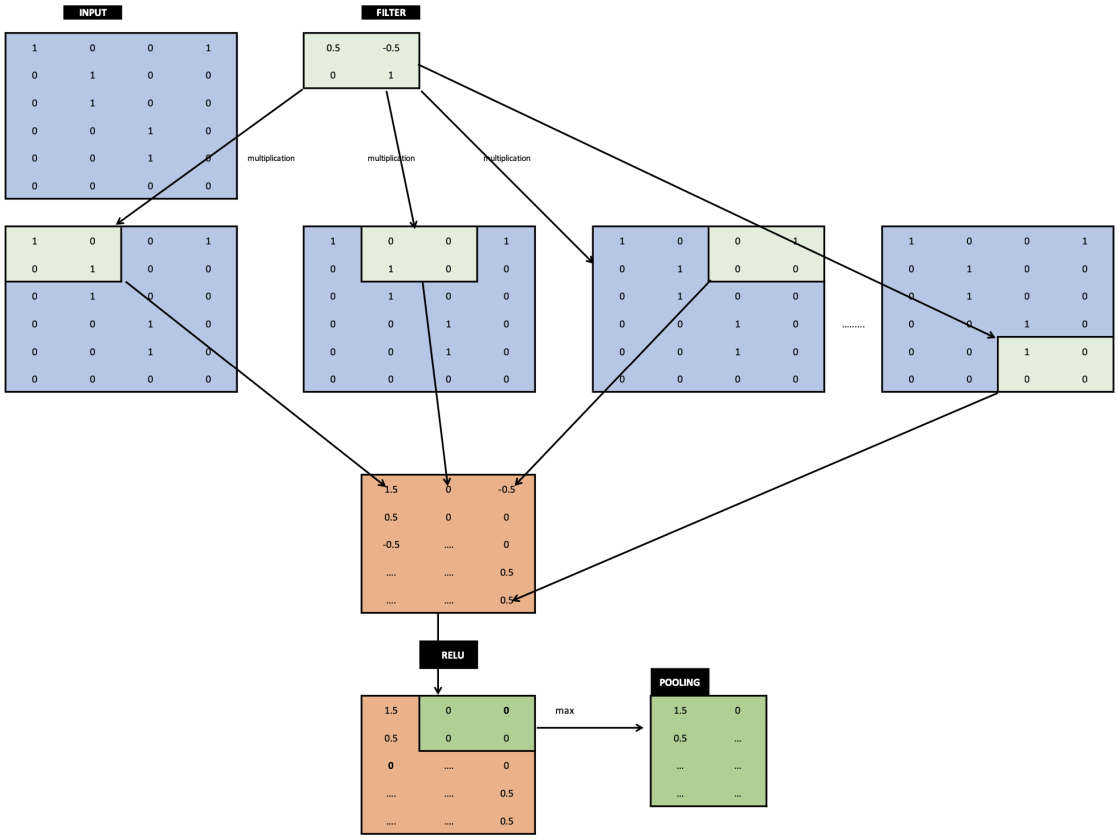
In the next stage *pooling* is applied to the derived non-linear features to reduce the dimensionality, in other words "summarize" the information and make it invariant to small changes.

Popular function choices for the pooling stage include the max operator, the weighted average based on the distance fro the center, norms, etc.

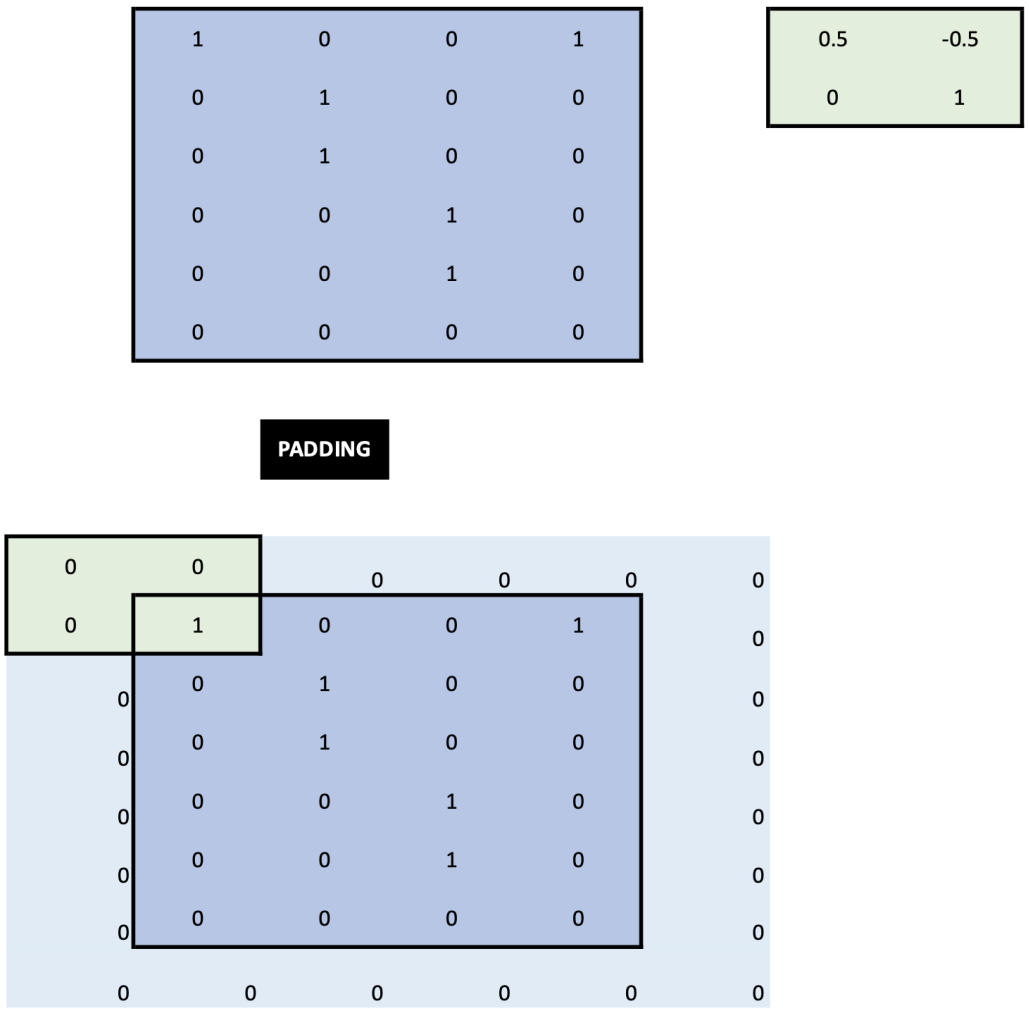
The above procedure is repeated multiple times, each layer detecting different and more complex features.

The last stage aggregates all the derived features into a fully connected MLP structure for classification or regression.

Visualizing a Convolution



Visualizing Pooling



Why do Convolutions Work?

Why do convnets do so well with image classification? This is because convolutional layers learn local patters in the input space (.e.g edges or other shapes) - unlike a fully connected netowrk that tries to learn global patterns. This has two implications:

First, convnets learn patterns that are *translation invariant*. This means that if a convnet learns a pattern in the top left corner of an image (e.g. an edge), it can recognise it elsewhere in the image. This is fundamentally different to how a fully connected network works, which would have to learn the same pattern from scratch if it appeared in a new location. This makes convnets very data efficient for data that posseses the translation invariance property, such as images. Carefule this property does not exist in all datasets!

Second, convents can learn *spacial hierarchies of patterns*. The first convolution can learn simple, local (in the input space) patterns (e.g. edges), the second convolution can learn more complex, larger patterns made from features of the first layer, and so on. This again works well for data that is spatially hierarchical, such as images.

Convolutions for Neural Networks

Convolutions operate over tensors called *feature maps*. These are usually 3D tensors with two spatial axes (*height* and *width*) for the number of pixels (e.g. 28 by 28) and a *depth* axis for the number of channels (e.g. 1 for greyscale and 3 for RGB images).

The convolution extracts small areas from the input (called the input feature map) and applies the *same* transformation to all of them (specifically a tensor product with the same across all extracted areas, learned weight matrix), thus creating an *output feature map* which again has height, width and depth.

The depth now is the dimension of the applied filters (which has to be chosen by the researcher).

Convolutions for Neural Networks

When we apply this to the greyscale MNIST data in the example below the input feature map is $(28, 28, 1)$. We chose to use 64 filters and so the first convolution layerproduces an output feature map of dimension $(26, 26, 64)$

Each of the 64 output channels contains a $(26, 26)$ *response map* of the specific filter over the input.

Hence the term *feature map* which is the 3D tensor of dimension (height, width, n) where *n* is the number of filters and each 2D tensor (height, width, i) is a *response map* of the i^{th} filter applied to the input.

Convolutions for Neural Networks

Convolutions for neural networks are defined by two key parameters:

(i) Size of the filter: this is usually 3×3, 5×5 or 7×7. (Notice it is an odd number, why?)

(ii) Number of filters: this gives the depth of the output feature map (above we used 64, this need to be chosen).

Note that the output width and height may differ from that of the input due to either of (i) *border effects* and (ii) *use of strides*.

Convolutions for Neural Networks

Border effects arise from the fact that the filter has a size larger than 1 which means that there are fewer ways to center the filter than the dimension of the input space.

Padding allows you to deal with this by adding an appropriate number of rows and columns of zeros around the input map.

The distance between two successive windows is called the *stride* and is a parameter of the model that needs to be chosen by the researcher.

A stride larger than 1 means that the feture map is downsized by a factor of 2. Strides other than 1 are generally not used.

Convolutions for Neural Networks

Once the convolution operation has been completed (including applying some non-linear function such as RELU), the next step in a convnet is usually to use some pooling technique to downsample the output feature map.

The most commonly used pooling operator is *max pooling*. This takes a window from the output feature map and returns the maximum value of each channel. Note that this layer has no parameters to learn.

Max pooling is commonly done using 2×2 windows and stride of 2 (non overlapping) which results in dowwnsampling of the feature map by a factor os 2.

The motivation behind *pooling* is two-fold: to reduce the dimensionality of the model (usually by a factor of 2) and to create spatial hierarchies by making succesive layers process larger amount of information.

CNN architectures

So to summarize, when building a convnet you need to choose a number of specific parameters and architecture specifics.

1. Number of filters (each one can produce different types of feature maps.)
2. Size of each of the kernels (typically 3 by 3, 5 by 5 or 7 by 7).
3. Stride: number of observations that the filter slides along when applied to the input. (Larger strides results in smaller feature maps.)
4. Zero-padding: whether to add zeros around the observations at the edge of the matrix.
5. Pooling type: how to summarize the derived features (usually the max operator).
6. Number of convolutional and/or pooling layers.

These are in addition to usual MLP architecture choices which need to be made at the last fully connected layers.

Example: A Convolutional Net for MNIST

Lets look at an example of a convnet for MNIST digit classification (same dataset as last week). First lets load and prepare the data.

```
library(keras)
# load the data
mnist <- dataset_mnist()

#notice this compact syntax
c(c(train_images, train_labels), c(test_images, test_labels)) %<-% mnist

train_images <- array_reshape(train_images, c(60000, 28, 28, 1))
train_images <- train_images / 255

test_images  <- array_reshape(test_images, c(10000, 28, 28, 1))
test_images  <- test_images / 255

train_labels <- to_categorical(train_labels)
test_labels  <- to_categorical(test_labels)
```

A Convolutional Net for MNIST

Now lets build a simple convnet. Below we have 3 convolutional layers and two pooling layers.

```
model <- keras_model_sequential() %>%
  # convolution layer
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
                activation = "relu", padding = "same", # we add padding
                input_shape = c(28, 28, 1)) %>% # the format of MNIST data point
  # pooling layer
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # convolution layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu", padding = "same") %>%
  # pooling layer
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  # convolution layer
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu", padding = "same")
```

Example: A Convolutional Net for MNIST

```
model

## Model
## Model: "sequential"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## conv2d (Conv2D)              (None, 28, 28, 32)        320
## _________________________________________________________________
## max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
## _________________________________________________________________
## conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
## _________________________________________________________________
## max_pooling2d_1 (MaxPooling2D) (None, 7, 7, 64)        0
## _________________________________________________________________
## conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
## =================================================================
## Total params: 55,744
## Trainable params: 55,744
## Non-trainable params: 0
## _________________________________________________________________
```

A Convolutional Net for MNIST

Now lets add two fully connected layers and the output layer.

```
model <- model %>%
  layer_flatten() %>% # flatten the data to pass to a dense layer
  # dense layer
  layer_dense(units = 100, activation = "relu") %>%
  # add dropout
  layer_dropout(rate = 0.3) %>%
  # output layer
  layer_dense(units = 10, activation = "softmax")
```

Example: A Convolutional Net for MNIST

```
model

## Model
## Model: "sequential"
## _________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =================================================================
## conv2d (Conv2D)              (None, 28, 28, 32)        320
## _________________________________________________________________
## max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
## _________________________________________________________________
## conv2d_1 (Conv2D)            (None, 14, 14, 64)        18496
## _________________________________________________________________
## max_pooling2d_1 (MaxPooling2D) (None, 7, 7, 64)        0
## _________________________________________________________________
## conv2d_2 (Conv2D)            (None, 7, 7, 64)          36928
## _________________________________________________________________
## flatten (Flatten)            (None, 3136)              0
## _________________________________________________________________
## dense (Dense)                (None, 100)               313700
## _________________________________________________________________
## dropout (Dropout)            (None, 100)               0
## _________________________________________________________________
## dense_1 (Dense)              (None, 10)                1010
## =================================================================
## Total params: 370,454
## Trainable params: 370,454
## Non-trainable params: 0
## _________________________________________________________________
```

A Convolutional Net for MNIST

Finally lets compile anf fit the model. We choose a number of epochs and mini-batch comparable to what we used last week.

```
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

model %>% fit(
  train_images, train_labels,
  epochs = 15, batch_size = 65
)

test_results <- model %>% evaluate(test_images, test_labels); test_results

## $loss
## [1] 0.04157078
##
## $accuracy
## [1] 0.9926
```

How does the model compare to the fully connected neural network?

Resources

There are a number of excellent articles and blogs about Convolutional Nets:

`www.towardsdatascience.com/deep-dive-into-convolutional-networks-48db75969fdf`

`www.ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/`

`setosa.io/ev/image-kernels/`

`www.cs.ryerson.ca/~aharley/vis/conv/flat.html`

Acknowledgements

These notes are based on and follow closely the excellent book "Deep Learning with R" by Franois Chollet (the creator of Keras) with J.J. Allaire.

These notes are also based on and follow closely the excellent tutorials at `wwww.keras.rstudio.tutorials`.

There are a numerous useful Keras related resources online, including at `www.keras.io`.

Additionally, you can refer to the excellent books for more examples and details: "The Deep Learning with R" and "Deep Learning with Python", the latter also by Franois Chollet.

A Deep Learning with Keras cheat sheet is available here `https://github.com/rstudio/cheatsheets/raw/master/keras.pdf`.