Cognitive Computer Vision

Computer Exercise

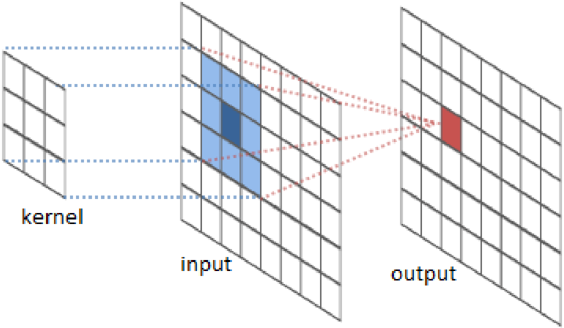
Tuesdays 14.15-15.45, Room D-118/119

Dr. Mikko Lauri
Email: lauri@informatik.uni-hamburg.de

SS 2018

Contents

4. 2D Convolutions



[1]

- Sliding window (kernel) over image, calculate inner product with contents of input.

[1] Image credit: http://jeanvitor.com/convolution-parallel-algorithm-python/

4. 2D Convolutions

```
tf.nn.conv2d(
    input,
    filter,
    strides,
    padding,
    use_cudnn_on_gpu=True,
    data_format='NHWC',
    dilations=[1, 1, 1, 1],
    name=None)
```

- `input` – the input images: tensor of shape $\begin{bmatrix} N & H & W & C \end{bmatrix}$: N images, height H, width W, C channels.
- Example: one grayscale image, shape of `input` is $\begin{bmatrix} 1 & H & W & 1 \end{bmatrix}$

4. 2D Convolutions

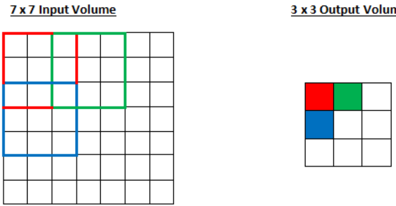
```
tf.nn.conv2d(
    input,
    filter,
    ...)
```

- `filter` – the filtering kernels, tensor of shape $\begin{bmatrix} F_h & F_w & N_i & N_o \end{bmatrix}$: filter height F_h, width F_w, N_i input channels, N_o output channels.
- F_h and F_w typically odd, the middle element is the origin of the kernel.
- Example: one filter operating on single-channel input: $\begin{bmatrix} F_h & F_w & 1 & 1 \end{bmatrix}$
- `filter[i,j,k,l]` is the weight of relative location (i, j) for the kth input channel to produce the lth output channel.

4. 2D Convolutions

```
tf.nn.conv2d(...,
    strides, ...)
```

- `stride`: number of units the filter shifts in each dimension, 1-D tensor with 4 elements: S_b, S_h, S_w, S_c



[2]

Figure: Stride 2 for height and width

[2] Image credit: https://adeshpande3.github.io

4. 2D Convolutions

```
tf.nn.conv2d(...,
    padding, ...)
```

- `padding` – How to pad image around the edges?
- `"SAME"`: Pad the input with zeros around the edges to ensure output is of same shape as input.
- `"VALID"`: no padding. Output shape will be smaller than input depending on the kernel shape.

4. 2D Convolutions

- `input` – the input images: tensor of shape $\begin{bmatrix} N & H & W & C \end{bmatrix}$: N images, height H, width W, C channels.
- `filter` – the filtering kernels, tensor of shape $\begin{bmatrix} F_h & F_w & N_i & N_o \end{bmatrix}$: filter height F_h, width F_w, N_i input channels, N_o output channels.
- `tf.reshape`, `tf.expand_dims`, or NumPy equivalents.

Static and dynamic shape of a tensor `x`

- The *static shape* is known at graph construction time and can be accessed by `x.shape`. Print it out anywhere!
- The *dynamic shape* can be partly or completely known only at runtime. Access by the operation `tf.shape(x)`. Use `sess.run`!
- The symbol `?` is shown for static shape where shape is unknown statically.

4. 2D Convolutions

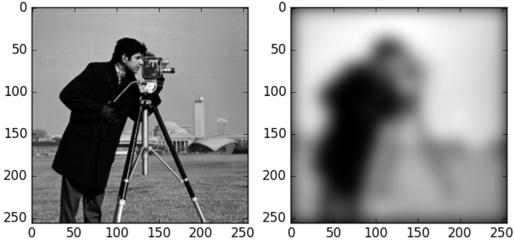


Figure: Left: original, Right: Smoothed with $\sigma = 10$

4. 2D Convolutions

- You will need OpenCV + Python bindings to generate the filter kernel
- Use a classroom computer if you do not have OpenCV on your laptop
- If you have OpenCV on your laptop, to install Python bindings: `sudo apt-get install python-opencv`

1. Variables

- Values of tensors "live" only during a single `Session.run()` call, and are lost afterwards.
- Exception: `tf.Variable` stores data that persists over multiple runs
  - Model parameters
  - Bookkeeping: running averages and other statistics, number of iterations, ...
- Shape specified via `tf.initializer`; some variables are trainable (e.g., model parameters), others are not (e.g., number of iterations)

```
tf.Variable(initial_value=None,
    trainable=True,
    name=None,
    ...)
```

2. Fully-connected layer

- Convenience function for creating a fully-connected (dense) neural network (NN) layer: `tf.layers.dense`

$$y = h(Wx + b)$$

- Input $x \in \mathbb{R}^m$, weights $W \in \mathbb{R}^{m \times k}$, bias $b \in \mathbb{R}^k$, output $y \in \mathbb{R}^k$, activation function $h : \mathbb{R} \to \mathbb{R}$
- Creates variables for W and b, adds ops to calculate y
- Infers the shape of W and b according to static shape of x and user-specified number of output neurons k

2. Fully-connected layer

- All layers in TF optimized to operate over several inputs at once (batch processing).
- `tf.layers.dense` expects an input of shape $\begin{bmatrix} N & m \end{bmatrix}$: number of inputs is N, each with dimensionality m – take this into account when defining the input!
- Same weight W and bias b will be applied to each of the N inputs.

3. Optimization and data logging

- Problem: minimize real-valued $F(\theta)$ where θ is a vector of parameters.
- Gradient descent idea:
  - Given the current parameters θ_n, calculate gradient of F at θ_n.
  - Gradient is a vector that points in the direction of the *steepest increase* of F: update θ_n by moving towards the direction of the *negative* gradient (direction of steepest *decrease*)
  - Repeat until convergence
- Gradient denoted by ∇F, step size $\gamma \in \mathbb{R}$, also called the *learning rate*.

$$\theta_{n+1} = \theta_n - \gamma \nabla F(\theta_n)$$

- There is no guarantee of finding a global optimum!

3. Optimization and data logging

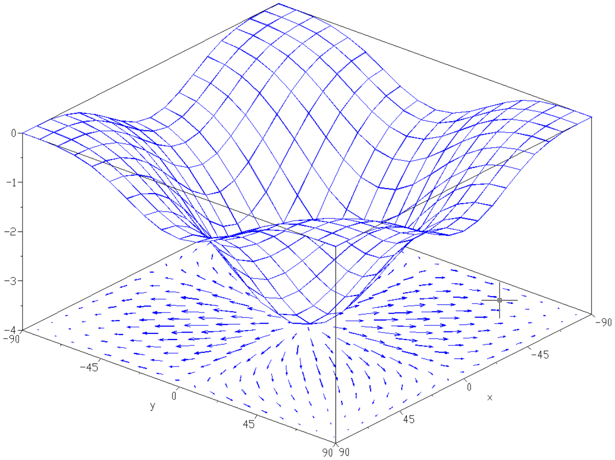


Figure: Gradient at (x, y) illustrated by arrows originating from (x, y).

3. Optimization and data logging

- Linear regression. Given *training data* (x_i, y_i) for $i = 1, \ldots, N$,

$$\min_{a,b \in \mathbb{R}} \sum_{i=1}^{N} (ax_i + b - y_i)^2$$

- a and b make up our parameter vector θ. The expression above is the *loss function* to be minimized with respect to the parameters.

3. Optimization and data logging

- Tensorflow recipe for optimization:
  1. Choose a model architecture.
  2. Set up a computation graph to compute a prediction $\hat{y}_i$ for given input x_i using the chosen architecture.
  3. Define a loss function that compares the prediction to the *ground truth* y_i: $L(\hat{y}_i, y_i)$
  4. Create an optimizer instance, e.g., `tf.train.GradientDescentOptimizer`
  5. Create a *training op* by calling the `minimize(loss)` method of the optimizer
  6. Iteratively run the training op while feeding training data.