Efficient Neural Network Training/Inferencing

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

March 23, 2020

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Machine Learning

Tom Mitchell: *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*

Learning Paradigms

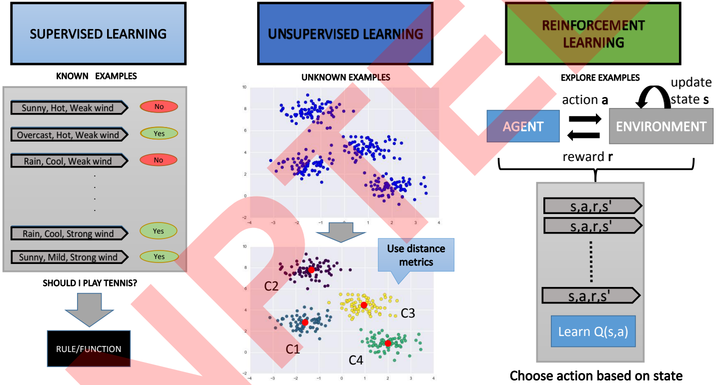


Figure: Types of Learning

Machine Learning

- Task **T**: Learning Rules/Functions from Examples.
- Each example is characterized by a vector of real numbers or boolean variables.
- Each example may have a target label (Supervised Learning) or no label (Unsupervised Learning).
- Examples are not available explicitly and some agent-environment interaction mechanism must be envisaged to extract meaningful examples (Reinforcement Learning).
- Experience **E**: More examples means better performance for task **T**!

What is Performance **P**?

- Performance **P** refers to some metric for assessing the quality of the rule/function learned.
- We restrict our discussion to Supervised Learning Problems.
- Supervised learning problems can be Classification (The target labels are discrete or categorical) or Regression (The target labels are continuous values).

Supervised Learning Workflow

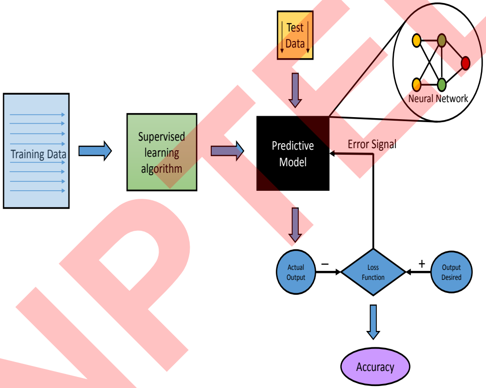


Figure: Training/Testing

Supervised Learning Algorithm

- Characterize the model to be learned by some parameter θ
- Define a loss function between predicted output and actual output. (function of θ and inputs)
- Update θ so that the loss function is minimized.
- The more the loss function is closer to the minima, the more closer is the predicted output to the actual output
- This is also known as parameter estimation.

Linear Regression

- Given a dataset $\mathcal{D} = \{(x_i, y_i), x_i, y_i \in \mathbb{R}, i \in [1, n]\}$, learn a function $f_\theta(x) = y$ that can predict any unknown x_j not in the dataset, with a label y_j with reasonable accuracy.
- The function f_θ is of the form $f(x) = wx + b$.
- The parameters to be tuned are the slope w and the intercept b.

Linear Regression

- Linear Regression can also handle multidimensional linear models of the form $y = w_1 x_1 + w_2 x_2 + \cdots w_n x_n + b$ where there are multiple x values.
- Geometrically, this is equivalent to fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

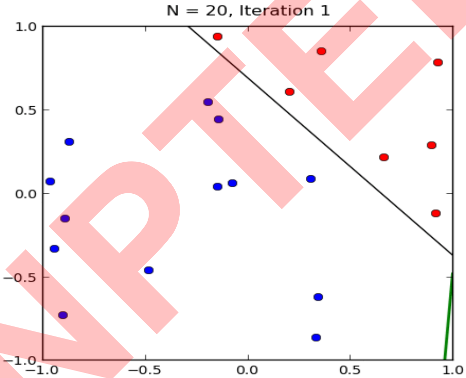
Perceptron: Linear Binary Classifier

- Given a dataset $\mathcal{D} = \{(x_i, y_i), x_i \in \mathbb{R}^d, y_i \in \{0, 1\}, i \in [1, n]\}$, learn a function $f_\theta(x) = y$ that can predict any unknown x_j not in the dataset, with a label y_j with reasonable accuracy.
- The function f_θ is of the form $f(x_i) = g(\sum_{j=1}^{d} w_j x_i[j] + b)$ where g is an activation function.

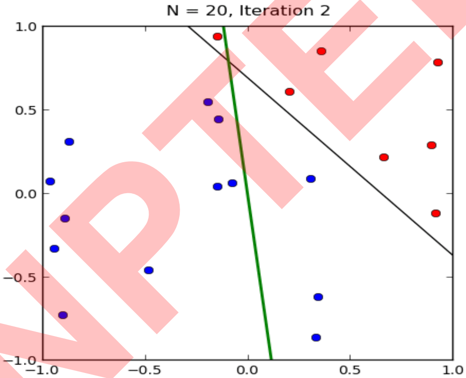
Perceptron: Linear Binary Classifier

- Given a dataset $\mathcal{D} = \{(x_i, y_i), x_i \in \mathbb{R}^d, y_i \in \{0, 1\}, i \in [1, n]\}$, learn a function $f_\theta(x) = y$ that can predict any unknown x_j not in the dataset, with a label y_j with reasonable accuracy.
- The function f_θ is of the form $f(x_i) = g(\sum_{j=1}^{d} w_j x_i[j] + b)$ where g is an activation function.

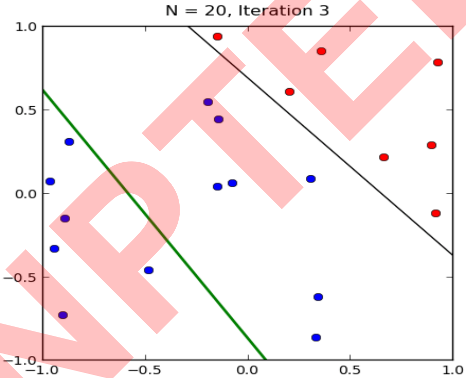
Perceptron Example



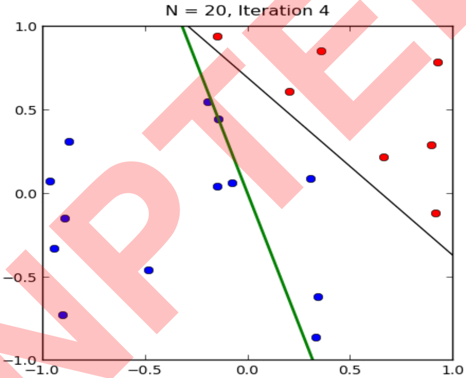
Perceptron Example



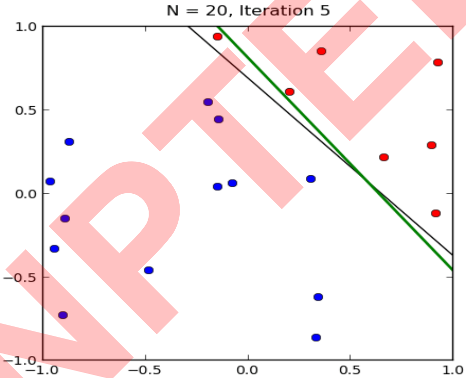
Perceptron Example



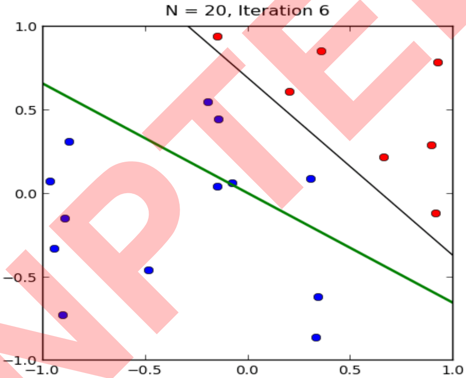
Perceptron Example



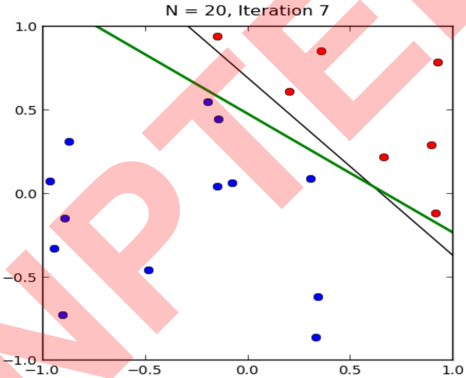
Perceptron Example



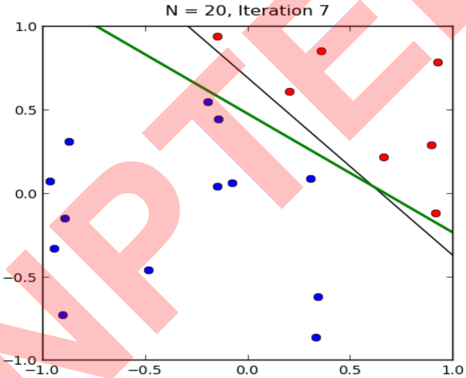
Perceptron Example



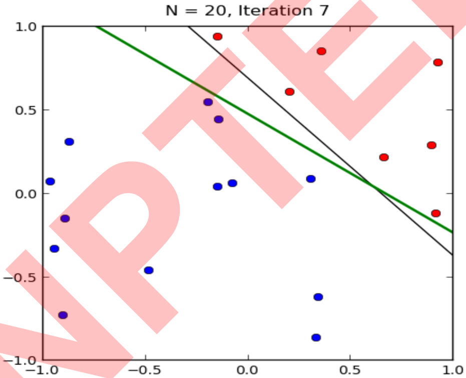
Perceptron Example



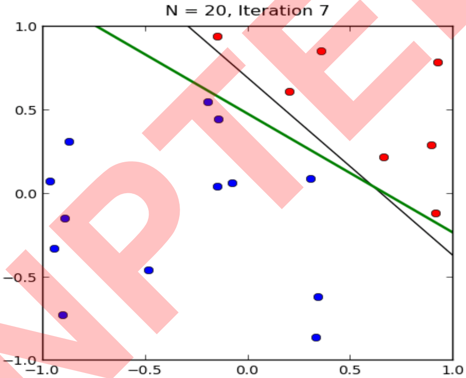
Perceptron Example



Perceptron Example



Perceptron Example



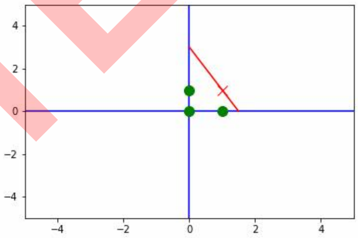
Perceptron Algorithm

Let $\mathbf{x} = [x_1, x_2, \cdots x_n, 1]$ and $\mathbf{w} = [w_1, w_2, \cdots, w_n, b]$
The value of the dot product $\mathbf{w}.\mathbf{x}$ is the same as $\sum_{j=1}^{n} w_j * x_j$.

```
while convergence is not reached
    for i in range(len(y)):
        y_pred = heaviside(w.x)
        dw = (y[i] - y_pred) * eta * x[i]
        w = w + dw
```

Perceptron NAND Example

x1	x2	y
0	0	1
0	1	1
1	0	1
1	1	0



XOR ?

Building Block of a NN

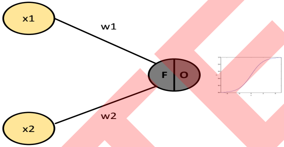
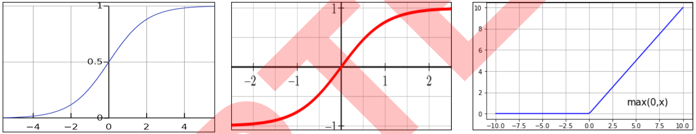


Figure: Perceptron

- The yellow nodes are inputs while the grey node is a neuron.
- The edges (synapses) have weights
- The incoming value to the neuron is $f = \sum w_i x_i$
- The outgoing value is a nonlinear function of f i.e. $o = \sigma(f)$

Activation Functions

SIGMOID | HYPERBOLIC TANGENT | RECTIFIED LINEAR UNIT



$$y = \frac{1}{1 + e^{-x}}$$

$$y = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$y = \max(0, x)$$

Figure: Linear/Non-linear functions

Multilayer Perceptron/ FeedForward Network

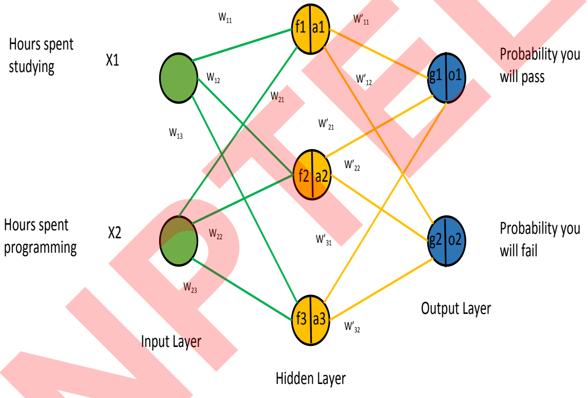


Figure: Classification

Multilayer Perceptron/ FeedForward Network

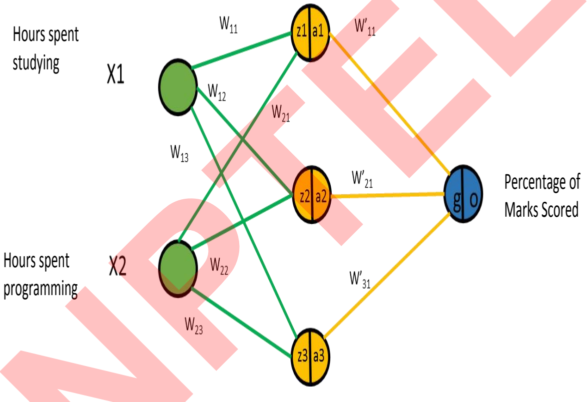


Figure: Regression

Note

- The objective of this course is to get you acquainted with the computation involved while training and testing a neural network.
- We shall not discuss core ML principles which should be followed while designing neural networks for various problem domains.

Neural Networks

- Each neuron of a layer accumulates a weighted sum of the inputs from the previous layer.
- Each neuron applies an activation function to its input and propagates the output to a neuron of the next layer..
- This results in a series of linear and non-linear transformations from the input layer to the output layer.
- The predicted output value for every input example is a function of the input feature values and the weights and activation in the network

Feedforward NN Forward Propagation

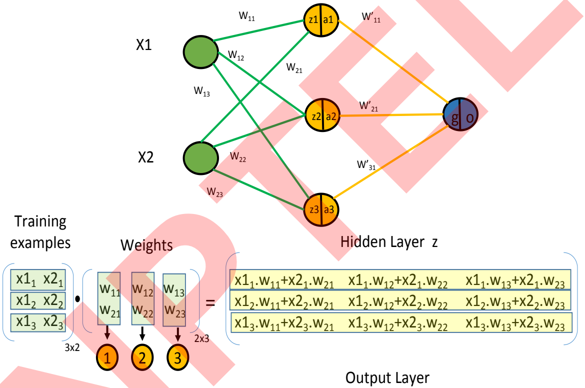
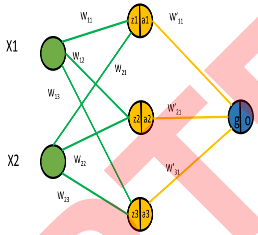


Figure: Feedforward propagation

Ref: Youtube video from Welch Labs

Feedforward NN Forward Propagation



Hidden Layer z

$$\begin{pmatrix} x1_1.w_{11}+x2_1.w_{21} & x1_1.w_{12}+x2_1.w_{22} & x1_1.w_{13}+x2_1.w_{23} \\ x1_2.w_{11}+x2_2.w_{21} & x1_2.w_{12}+x2_2.w_{22} & x1_2.w_{13}+x2_2.w_{23} \\ x1_3.w_{11}+x2_3.w_{21} & x1_3.w_{12}+x2_3.w_{22} & x1_3.w_{13}+x2_3.w_{23} \end{pmatrix}_{3x3} \Rightarrow$$

Hidden Layer a

$$\begin{pmatrix} \sigma(x1_1.w_{11}+x2_1.w_{21}) & \sigma(x1_1.w_{12}+x2_1.w_{22}) & \sigma(x1_1.w_{13}+x2_1.w_{23}) \\ \sigma(x1_2.w_{11}+x2_2.w_{21}) & \sigma(x1_2.w_{12}+x2_2.w_{22}) & \sigma(x1_2.w_{13}+x2_2.w_{23}) \\ \sigma(x1_3.w_{11}+x2_3.w_{21}) & \sigma(x1_3.w_{12}+x2_3.w_{22}) & \sigma(x1_3.w_{13}+x2_3.w_{23}) \end{pmatrix}_{3x3} = \begin{pmatrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \\ a_{11}\ a_{31}\ a_{31} \end{pmatrix}_{3x3}$$

Figure: Feedforward propagation

Feedforward NN Forward Propagation

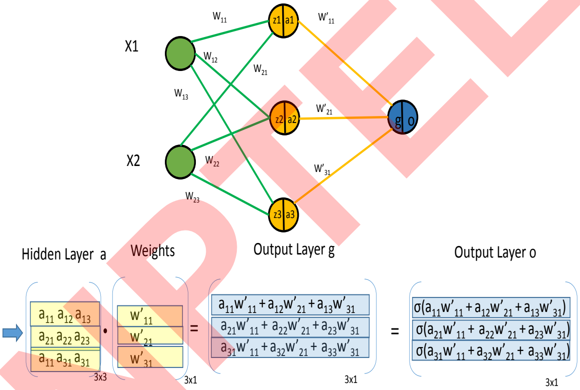
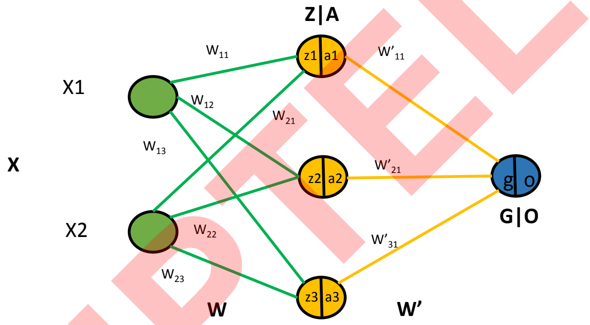


Figure: Feedforward propagation

Neural Networks



Z = XW → **A = f(Z)** → **G = AW'** → **O = f(G)**

Figure: Feedforward propagation

The feedforward propagation can therefore be expressed as a series of matrix

Neural Networks

- Given the structure and weights of a neural network, we now know how to compute the predicted output value for an input example.
- The structure of the network i.e. the number of layers and number of neurons per layer are referred as hyperparameters (to be decided by the user).
- The weights are the actual parameters (θ) which will be learned during the course of training.
- Training involves the minimization of a cost/loss function.

Loss Function

- Define a loss function $J(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - o_i)^2$ where y_i and o_i are the actual outputs and predicted outputs of input example i respectively.
- Recall the feedforward propagation equations.
  $\mathbf{Z} = \mathbf{XW}, \mathbf{A} = f(\mathbf{Z}), \mathbf{G} = \mathbf{AW}', \mathbf{O} = f(\mathbf{G})$
- Therefore, $J = \sum \frac{1}{2}(\mathbf{Y} - f(f(\mathbf{XW})\mathbf{W'})^2)$ where the summation operation is over the elements of the column vector obtained from the loss function.
-

Minimizing Loss Function

- Find values of **W** and **W'** so that J(w) is minimized.
- Compute $\frac{\partial J}{\partial \mathbf{W}}$ and $\frac{\partial J}{\partial \mathbf{W'}}$
- Instead of setting the partial derivatives to zero and finding a solution, we perform numerical gradient descent.
- Update the weights **W** and **W'** in the direction of the steepest gradient descent

Gradient Descent

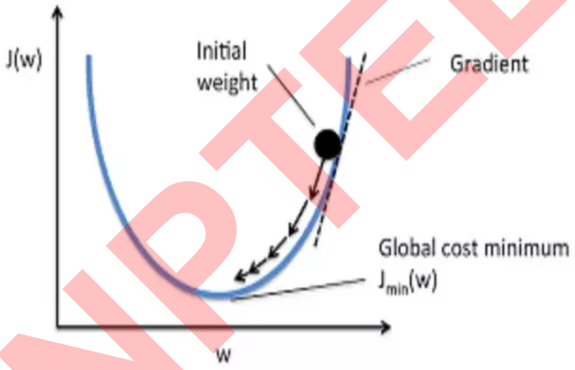


Figure: Obtaining minimum J

Gradient Descent

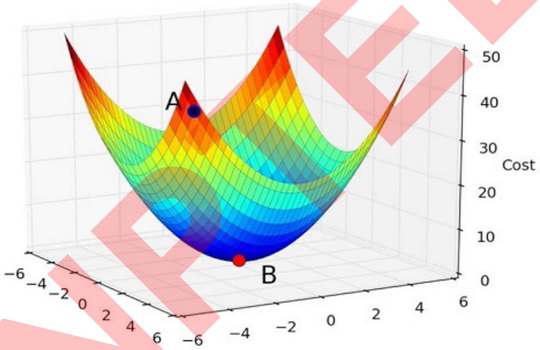


Figure: Obtaining minimum J

Training Using Backpropagation

- Perform feedforward propagation to obtain predicted output values for each input example.
- Compute loss function $J(w)$
- Compute $\frac{\partial J}{\partial \mathbf{W}}$ for each weight matrix
- Update weights of every weight matrix W with the gradient.
- Repeat steps 1-3 until there is no change in gradient.

Computing Partial Derivatives w.r.t a Matrix

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{w_{11}} & \frac{\partial J}{w_{12}} & \frac{\partial J}{w_{13}} & \cdots & \frac{\partial J}{w_{1n}} \\ \frac{\partial J}{w_{21}} & \frac{\partial J}{w_{22}} & \frac{\partial J}{w_{23}} & \cdots & \frac{\partial J}{w_{2n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{w_{d1}} & \frac{\partial J}{w_{d2}} & \frac{\partial J}{w_{d3}} & \cdots & \frac{\partial J}{w_{dn}} \end{bmatrix}$$

The dimensions of the weight matrix and its gradients will be the same.

Neural Networks

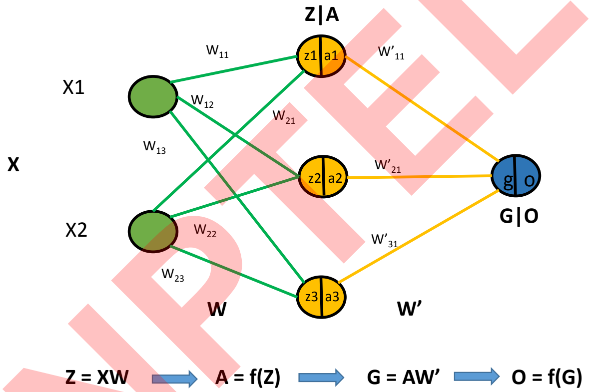


Figure: Feedforward propagation

We first compute $\frac{\partial J}{\partial \mathbf{W'}}$

Chain Rule

$$\frac{\partial J}{\partial \mathbf{W'}} = \frac{\partial}{\partial \mathbf{W'}} \sum \frac{1}{2}(y_i - o_i)^2 = -\sum (y_i - o_i) \frac{\partial o_i}{\partial \mathbf{G}} \frac{\partial G}{\partial \mathbf{W'}}$$

$$= -\sum (y_i - o_i) f'(g_i) \frac{\partial G}{\partial \mathbf{W'}}$$

Consider input example 1 and one weight say w'_{11}
The derivative is $-(y_1 - o_1) f'(g1) \frac{\partial}{\partial w'_{11}} (w'_{11} a_{11} + w'_{21} a_{12} + w'_{31} a_{13})$
$= \delta_1^1 a_{11}$
For input example 2, the gradient would be $= \delta_2^1 a_{21}$
For input example 3, the gradient would be $= \delta_3^1 a_{31}$

Computing Partial Derivatives w.r.t a Matrix

$$\frac{\partial J}{\partial W'} = \begin{bmatrix} \frac{\partial J}{w'_{11}} \\ \frac{\partial J}{w'_{21}} \\ \frac{\partial J}{w'_{31}} \end{bmatrix} = \begin{bmatrix} \delta_1^1 a_{11} + \delta_2^1 a_{21} + \delta_3^1 a_{31} \\ \delta_1^1 a_{12} + \delta_2^1 a_{22} + \delta_3^1 a_{32} \\ \delta_1^1 a_{13} + \delta_2^1 a_{23} + \delta_3^1 a_{33} \end{bmatrix}$$

This can be expressed as $\mathbf{A}^T \delta^1$

Computing gradient w.r.t W

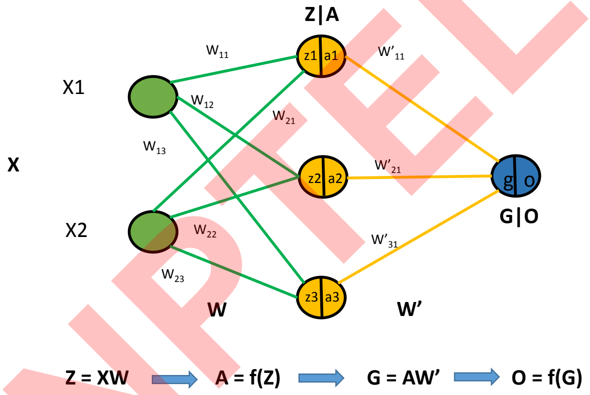


Figure: Feedforward propagation

In a similar fashion we compute $\frac{\partial J}{\partial \mathbf{W}}$

Chain Rule

$$\frac{\partial J}{\partial \mathbf{W}} = -(Y - O)\frac{\partial O}{\partial \mathbf{G}}\frac{\partial G}{\partial \mathbf{W}}$$

$$= -(Y - O)f'(G)\frac{\partial G}{\partial \mathbf{A}}\frac{\partial A}{\partial \mathbf{W}}$$

$$= \delta^1 \mathbf{W'}^T \frac{\partial A}{\partial \mathbf{W}}$$

$$= \delta^1 \mathbf{W'}^T \frac{\partial A}{\partial \mathbf{Z}}\frac{\partial Z}{\partial \mathbf{W}}$$

$$= \delta^1 \mathbf{W'}^T f'(\mathbf{Z})\frac{\partial Z}{\partial \mathbf{W}}$$

$$= \mathbf{X}^T \delta^1 \mathbf{W'}^T f'(\mathbf{Z}) \text{ (Derive!)}$$

$$= \mathbf{X}^T \delta^2 \text{ where } \delta^2 = \delta^1 \mathbf{W'}^T f'(\mathbf{Z})$$

Backward Propagation Delta Rule

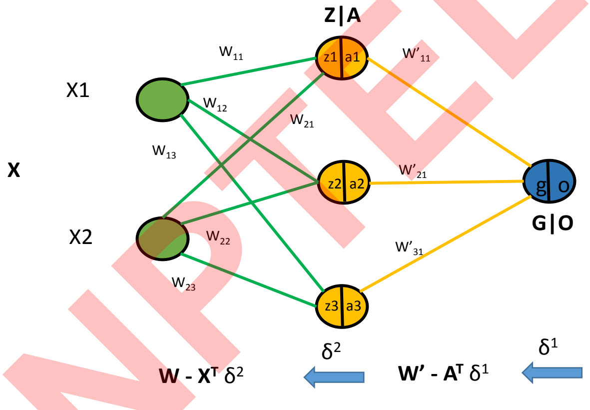


Figure: Backward propagation

Summary

- Feedforward propagation can be performed by a series of linear and non linear transformations involving matrix operations starting from the input layer.
- Backpropagation also involves a series of linear and non linear transformations involving matrix operations starting from the output layer.
- Each operation and transformation exhibits parallelism and scope for optimizations using a GPU.

Building a DL Library

A DL Library should have support for the following

- Provide constructs for specifying a network.
- Provide efficient routines for feedforward, backpropagation and gradient computation.
- Provide routines for training and testing.
- Should support parallel and distributed processing for the computation passes.

DL Libraries like Tensorflow and Theano use a Computational Graph Abstraction for encoding a neural network.

Computation Graph

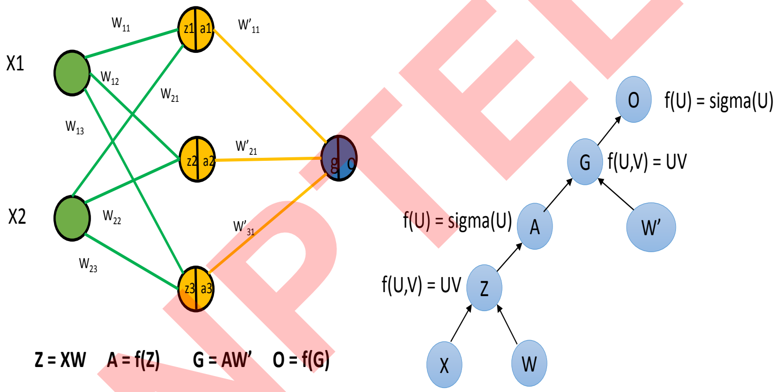


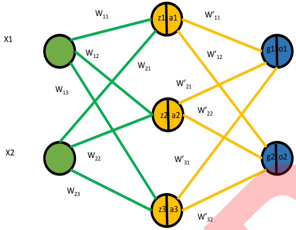
Figure: MLP vs CG

Computation Graph

A graph that denotes the functional description of the required computation.

- A node with no incoming edge is a tensor, matrix, vector or scalar value.
- A node with an incoming edge is a function of the edge's tail node. computation.
- An edge represents a data dependency between nodes.
- A node knows how to compute its value and the value of its derivative w.r.t each incoming edges's tail node.

Computation Graph



J is the loss function

Z = XW A = f(Z) G = AW' O = f(G)

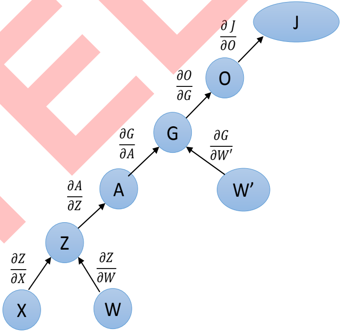


Figure: MLP vs CG

Computations for a CG

- **Forward Computation:** Loop over each node in topological order and compute the value of the node given its inputs.
- **Backward Computation** Loop over each node in reverse topological order and compute the derivative of the final goal node with respect to each incoming edge's tail node.

Backpropagation: Gradient w.r.t W

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial O}\frac{\partial O}{\partial G}\frac{\partial G}{\partial A}\frac{\partial A}{\partial Z}\frac{\partial Z}{\partial W}$$

J is the loss function

Z = XW A = f(Z) G = AW' O = f(G)

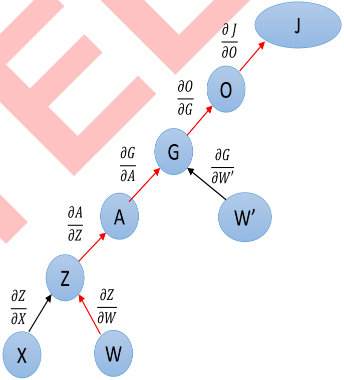


Figure: Computing gradients on CG

Backpropagation: Gradient w.r.t W'

$$\frac{\partial J}{\partial W'} = \frac{\partial J}{\partial O}\frac{\partial O}{\partial G}\frac{\partial G}{\partial W'}$$

J is the loss function

Z = XW A = f(Z) G = AW' O = f(G)

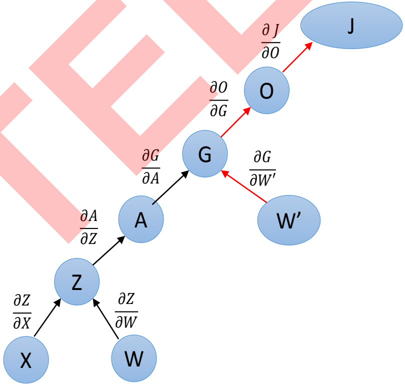


Figure: Computing gradients on CG

Convolutional Neural Network

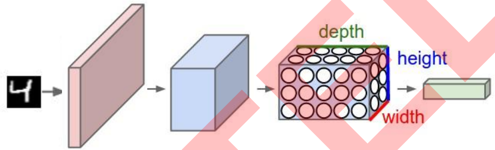


Figure: Example: Digit Classification

- Convolutional Neural Networks are very similar to vanilla Neural Networks discussed before and are used primarily for image classifcation tasks.
- An end to end CNN network expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other.
- Unlike vanilla Neural Networks, CNNs have neurons arranged in 3 dimensions: width, height, depth.

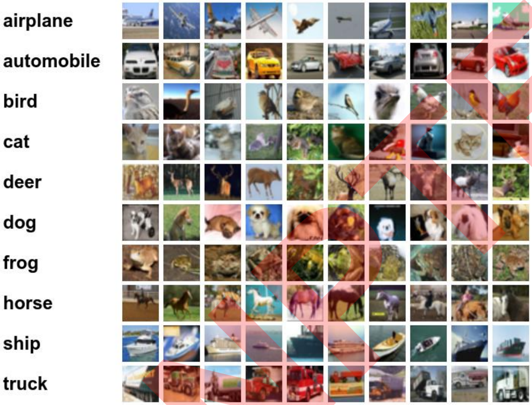
Convolutional Neural Network

A CNN is typically made up of the following layers

1. Input Layer
2. Convolutional Layer
3. Pooling Layer
4. RELU
5. Fully Connected Layer

The input and output for each of the layers 1-4 represent 3D image volumes.
The fully connected layer is typically used at the end where the 3D image volume is flattened and fed as input.

Classification Problem: Example



CIFAR 10 Dataset

Multi-class classification problem

INPUT: 3 x 32 x 32

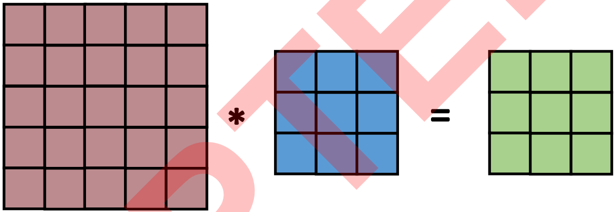
Raw pixel values for 32 x 32 images with three colour channels R,G,B

Figure: 10-Class Classification Problem

Convolution Layer

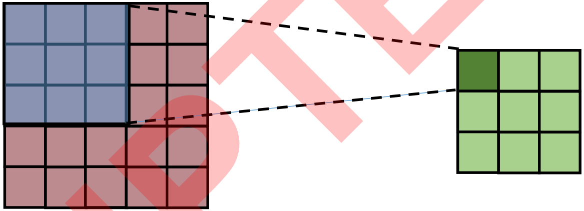
- The Convolution layer is the core building block of a CNN and is computationally expensive.
- The input to the layer is a 3D image volume. The output to the layer is also a 3D image volume
- The dimensions of the output are dictated by three hyper-parameters - depth, stride and zero-padding.

2D Convolution Example

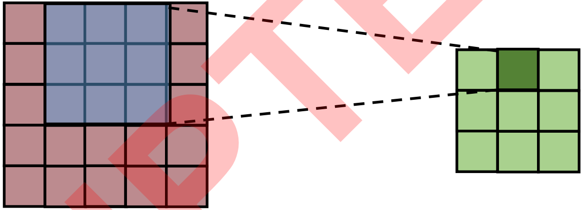


Given a 2-D input image I and a 2-D weight filter(mask) W, the convolution operation slides the filter over the image, computes a neighborhood operation (weighted sum) over the elements of I and produces a 2-D output image.

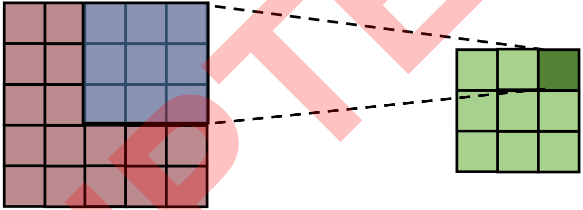
2D Convolution Example



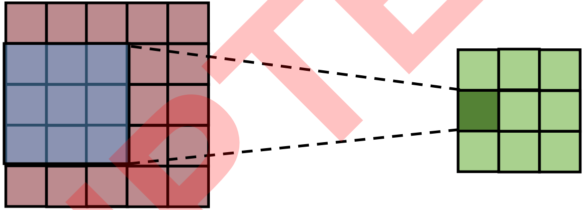
2D Convolution Example



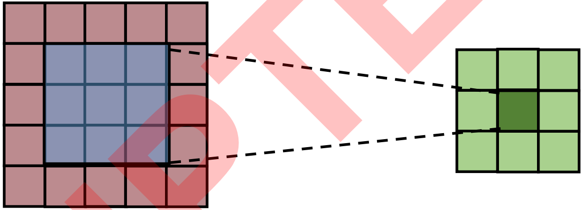
2D Convolution Example



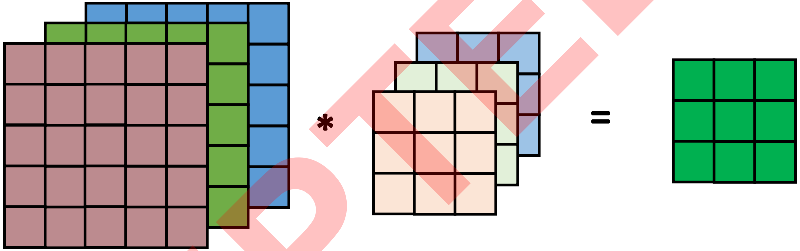
2D Convolution Example



2D Convolution Example

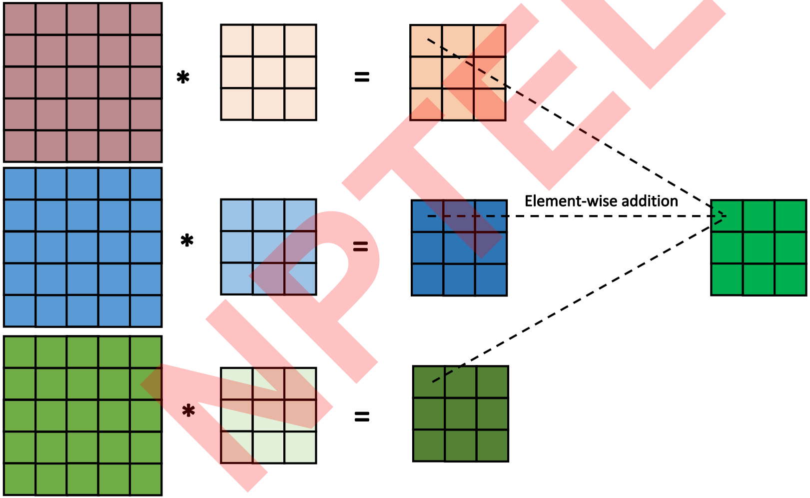


3D Convolution Example

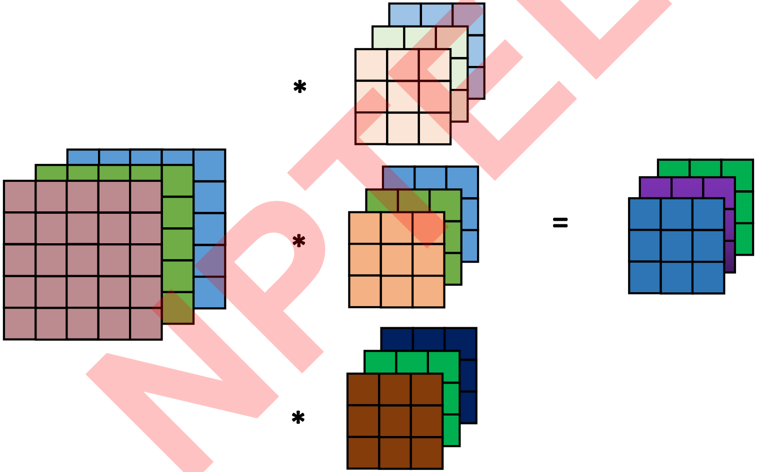


Given a 3-D input image I and a 3-D weight filter(mask) W, the convolution operation slides the filter over the image, computes a neighborhood operation (weighted sum) over the elements of I and produces a 2-D output image.

3D Convolution Example



Convolution Layer



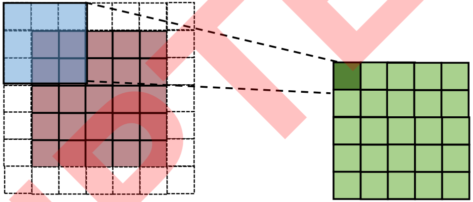
Convolution Layer: Parameters and Hyper-parameters

- The Convolutional layer has a 3D weight matrix or an array of 2D weight filters which represent the learnable **parameters**. .
- The dimensions of the output are dictated by four **hyper-parameters** - number of filters, filter dimensions, stride and zero-padding.

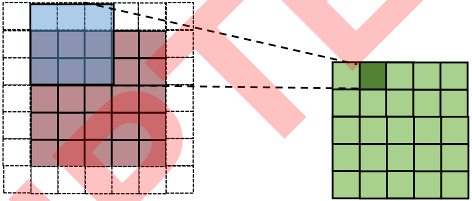
Convolution Layer: Spatial Transformations

- The Convolutional layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- The hyper-parameters for the layer are as follows.
  - M - Number of filters
  - F - Filter Size
  - S - Stride
  - P - Zero Padding
- The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
  - $C' = M$
  - $H' = 1 + (H - F + 2 * P)/S$
  - $W' = 1 + (W - F + 2 * P)/S$

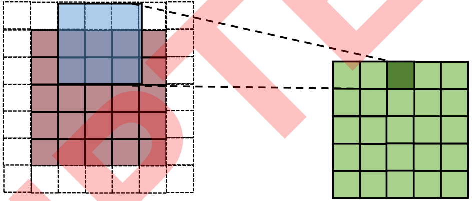
Padding $P = 1$



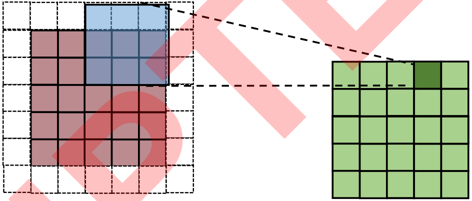
Padding



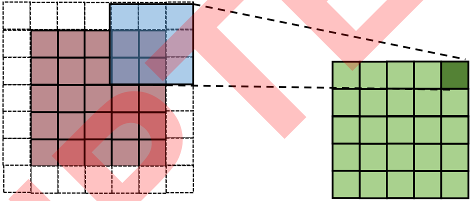
Padding



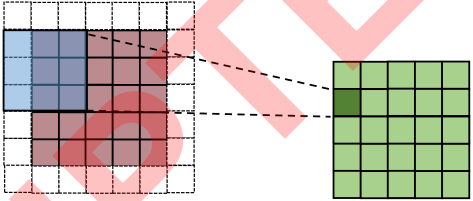
Padding



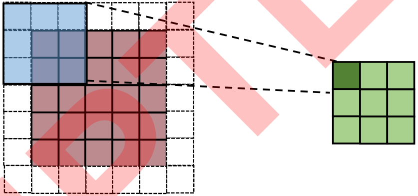
Padding



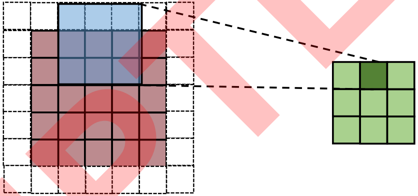
Padding



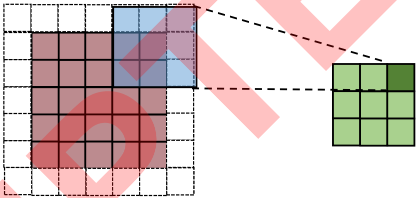
Stride $S = 2$



Stride



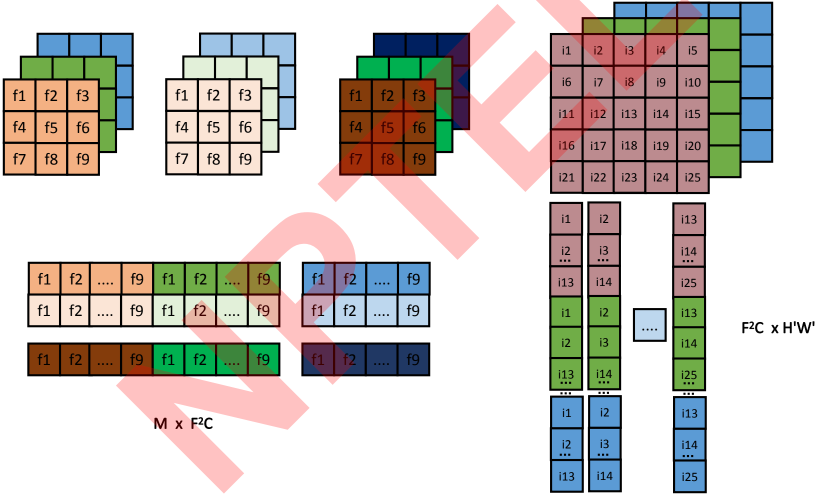
Stride



Convolution and Matrix Multiplication

- The convolution operation essentially performs dot products between the filter weights and patches of the input image.
- A common approach is to leverage this fact and formulate the operation of the convolutional layer as a matrix multiplication.

Convolution and Matrix Multiplication



Im2Col Operation

```
#define CUDA_KERNEL_LOOP(i, n) \
for (int i = blockIdx.x * blockDim.x + threadIdx.x; \
i < (n); \
i += blockDim.x * gridDim.x)
template <typename Dtype>
__global__ void im2col_gpu_kernel(const int n, const Dtype* data_im,
const int height, const int width, const int kernel_h, const int kernel_w,
const int pad_h, const int pad_w, const int stride_h, const int stride_w,
const int height_col, const int width_col, Dtype* data_col)
{
CUDA_KERNEL_LOOP(index, n) {
 const int h_index=index/width_col;
 const int h_col=h_index%height_col; const int w_col=index%width_col;
 const int c_im = h_index/height_col; const int c_col=c_im*kernel_h*kernel_w;
 const int h_offset = h_col*stride_h-pad_h;
 const int w_offset = w_col*stride_w-pad_w;
 Dtype* data_col_ptr = data_col;
 data_col_ptr+=(c_col* height_col+h_col)*width_col + w_col;
```

Im2Col Operation

```
const Dtype* data_im_ptr = data_im;
data_im_ptr += (c_im * height + h_offset) * width + w_offset;
for (int i=0;i<kernel_h;++i) {
 for (int j = 0; j < kernel_w; ++j) {
   int h_im = h_offset + i;  int w_im = w_offset + j;
   *data_col_ptr = (h_im >= 0 && w_im >= 0 && h_im < height && w_im < width)
       ?
                                data_im_ptr[i * width + j ] : 0;
     data_col_ptr += height_col * width_col;
  }
 }
}
}
```

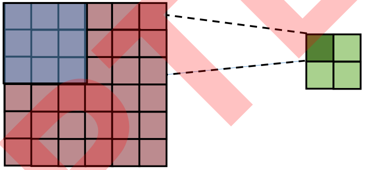
Convolution and Matrix Multiplication

- The disadvantage of the im2col approach is extra memory, since some values in the 3D input are replicated multiple times in the columns.
- However, the benefit is that there are many very efficient implementations of **G**eneral **M**atrix **M**ultiplication that we can take advantage of (cuBLAS API).
- We'll discuss certain GEMM optimizations later.

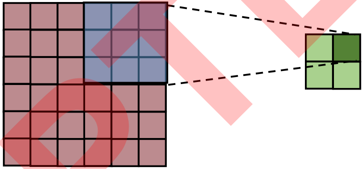
Pooling Layer

- The pooling layer typically applies a spatial downsampling transformation.
- The input and outputs for this layer are again 3D image volumes.
- The pooling layer does not have any learnable parameters and is purely a transformation operation in the context of CNNs.
- Pooling operations are typically - i)Max pooling and ii) Average pooling

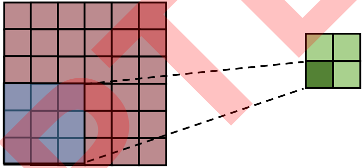
Pooling Operation



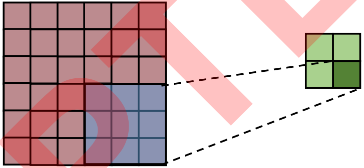
Pooling Operation



Pooling Operation



Pooling Operation



Pooling Layer: Spatial Transformations

- The Pooling layer takes as input a 3D image volume of dimensions $C \times H \times W$.
- The hyper-parameters for the layer are as follows.
  - F - Filter Size
  - S - Stride
- The layer produces a 3D volume of dimensions $C' \times H' \times W'$ where
  - $C' = C$
  - $H' = 1 + (H - F)/S$
  - $W' = 1 + (W - F)/S$

Pooling Kernel

```
template <typename Dtype>
__global__ void MaxPoolForward(const int nthreads,
const Dtype* const bottom_data, const int num, const int channels,
const int height, const int width, const int pooled_height,
const int pooled_width, const int kernel_h, const int kernel_w,
const int stride_h, const int stride_w, const int pad_h, const int pad_w,
Dtype* const top_data, int* mask, Dtype* top_mask) {
 CUDA_KERNEL_LOOP(index, nthreads) {
  const int pw = index % pooled_width;
  const int ph = (index / pooled_width) % pooled_height;
  const int c = (index / pooled_width / pooled_height) % channels;
  const int n = index / pooled_width / pooled_height / channels;
  int hstart = ph * stride_h - pad_h; int wstart = pw * stride_w - pad_w;
  const int hend = min(hstart + kernel_h, height);
  const int wend = min(wstart + kernel_w, width);
  hstart = max(hstart, 0); wstart = max(wstart, 0);
  Dtype maxval = -FLT_MAX; int maxidx = -1; //FLT_MAX --> max float
  const Dtype* const bottom_slice=bottom_data+(n*channels+c)*height*width;
```

Pooling Kernel

```
for (int h = hstart; h < hend; ++h) {
  for (int w = wstart; w < wend; ++w) {
    if (bottom_slice[h * width + w] > maxval) {
     maxidx = h * width + w;
     maxval = bottom_slice[maxidx];
    }
  }
}
top_data[index] = maxval;
if (mask) {
  mask[index] = maxidx;
} else {
 top_mask[index] = maxidx;
}
}
}
```

RELU Activation Function

```
__global__ void ReLUForward(const int n, const Dtype* in, Dtype* out,
Dtype negative_slope) {
  CUDA_KERNEL_LOOP(index, n) {
   out[index] = in[index] > 0 ? in[index] : in[index] * negative_slope;
 }
}
```

Rectified **L**inear **U**nit is the standard activation function used in CNNs.

End to End CNN