MIDDLE EAST TECHNICAL UNIVERSITY

DI504 Foundations of Deep Learning

Deep Learning Layers

METU

Welcome again!

This is DI504, Foundations of Deep Learning,

- We previously talked about feature spaces and score functions, which are basically the fundamentals concept of machine learning.
- Then we jumped into loss function. We learned what ANNs are and how to optimized them. Then about CNNs.
- Now, it is time to get deeper in to CNN-based architectures.

Layers Types in AlexNet

In AlexNet we saw:

- Convolutional layers
- Pooling Layers
- Fully Connected Layers
- Classification Layer (soft-max)

- Any other types?

Layers Types in AlexNet

In AlexNet we saw:

- Convolutional layers
- Pooling Layers
- Fully Connected Layers
- Classification Layer (soft-max)

METU

Layers Types

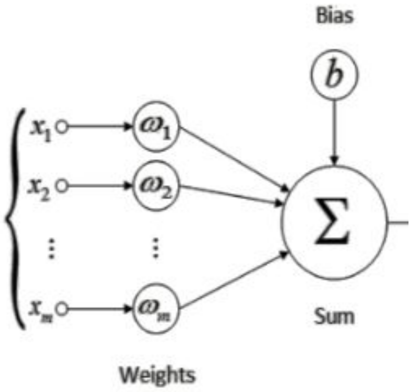
- Convolutional deep architectures today may include different types of layers for different purposes.
- Let’s go over these layers types. Knowing different layer types will help us understand different deep architectures.
- So what are these types:

Layers Types

Let’s categorize layers first:

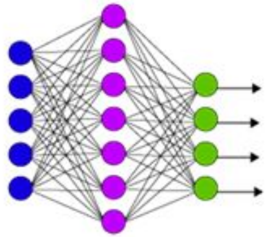
- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

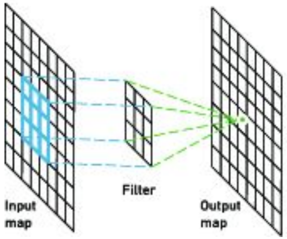
Weight Multiplication Layers



This is the fundamental layer type in neural nets

- Input values are multiplied with weights, and all multiplications are summed up.
  - Fully-Connected Layer:
    - Every node at a layer is connected to every node at the next layer.
    - Each connection/line represents a multiplication.
  - Convolutional Layer:
    - Limited connections, thus limited multiplications
- There are some other types as well.





Conv Layer Types

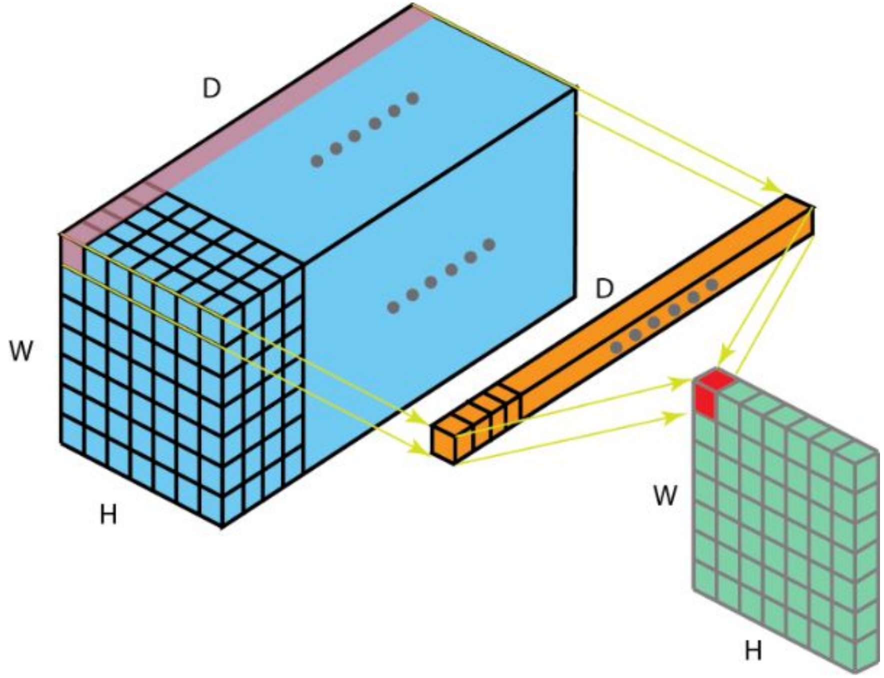
We have different types of convolutional layers:

- Simple Conv. Layer (the one we learned)
- 1x1 Conv Layer
- Transposed Conv. (a.k.a deconvolution) Layer
- Dilated (Atrous) Convolution Layer
- Grouped Conv. Layer
- And others... such as shuffled conv, flattened conv, seperable conv, etc, (which we will not cover)

1x1 Conv Layer

Filter size is simply: 1x1xDepth

- (if 3D, then 1x1x1) (or 1x1x...x1x1 depending on the spatial dimensions)
- Why helpful?
- If the input layer has multiple channels/filters, actually 1x1 conv scrambles them all up!



1x1 Conv Layer

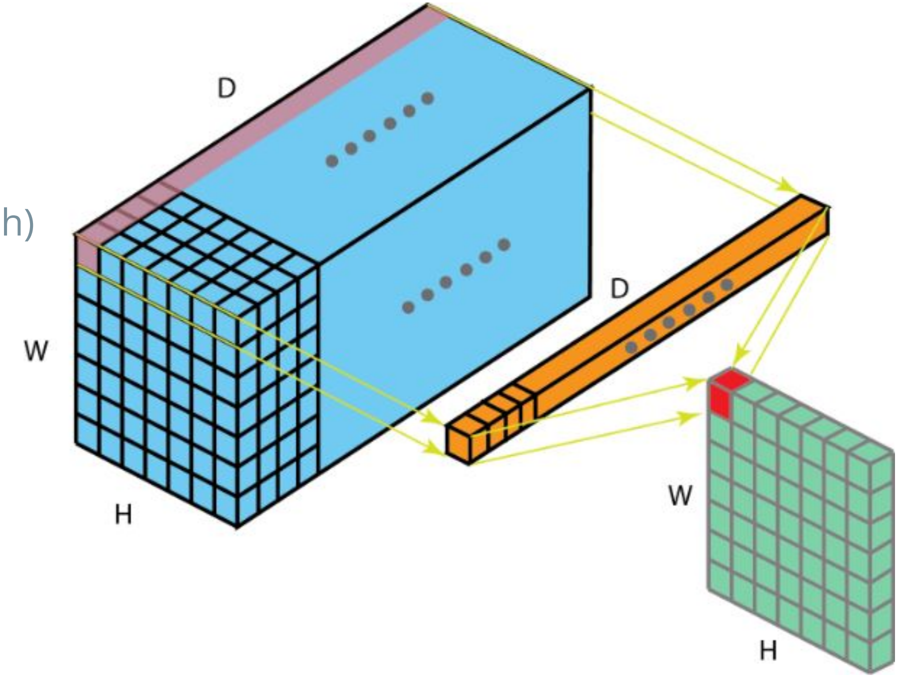
It changes the filter depth

- Input layer WxH x D (depth)
- Filter size 1x1 x D x K (must be the same depth)

The output size will be:

- WxHxK

K=1 in the figure →



1x1 conv filters are used to change the dimensionality in the “filter space”.

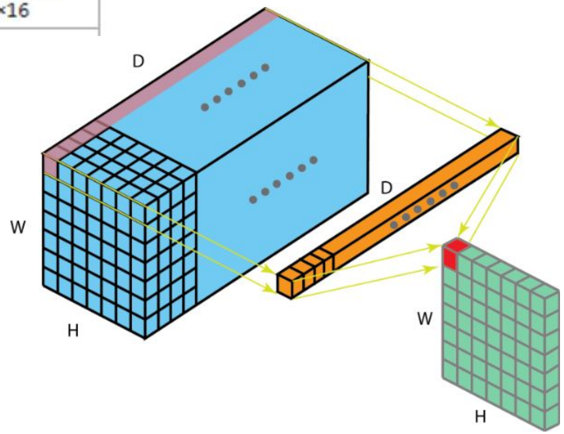
- Number of weights:
- 1 x 1 x D x K
- Number of biases: K

1x1 Conv Layer (example)

	Name	Type	Activations	Learnables
1	imageinput 100x100x8 images with 'zerocenter' normalization	Image Input	100×100×8	-
2	conv_1 32 3x3x8 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	25×25×32	Weights 3×3×8×32 Bias 1×1×32
3	conv_2 16 1x1x32 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	25×25×16	Weights 1×1×32×16 Bias 1×1×16

Sample 1x1 layer in a network, seen above.

1x1 conv layer does not change the receptive field.



Transposed (de)convolution Layer

Deconv is simply upsampling with weight multiplication.

- Input layer has spatial dimensions of size M x N
- You need k·M x k·N (k>1)

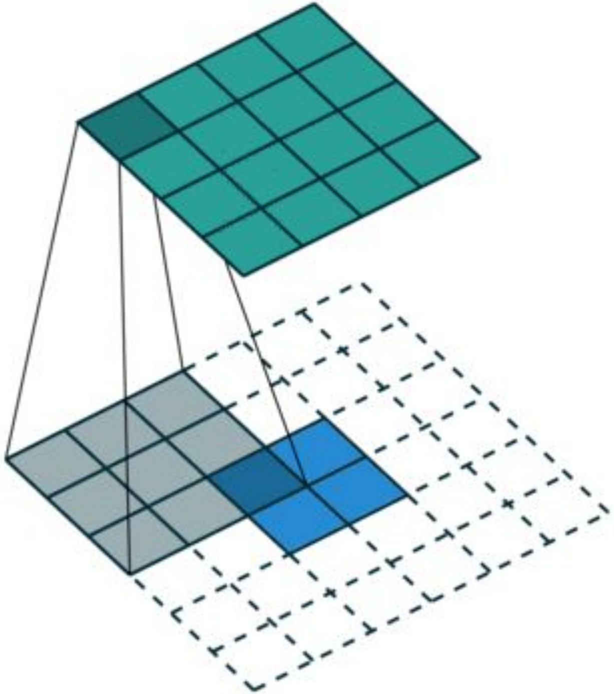
So Deconvolution does upsampling and it is like a sampling layer, but there are weights!

Remember the formula:

$$O = \frac{M - f + (p_{m-start} + p_{m-end})}{stride_m} + 1$$

if stride>1, then we have smaller sized output.

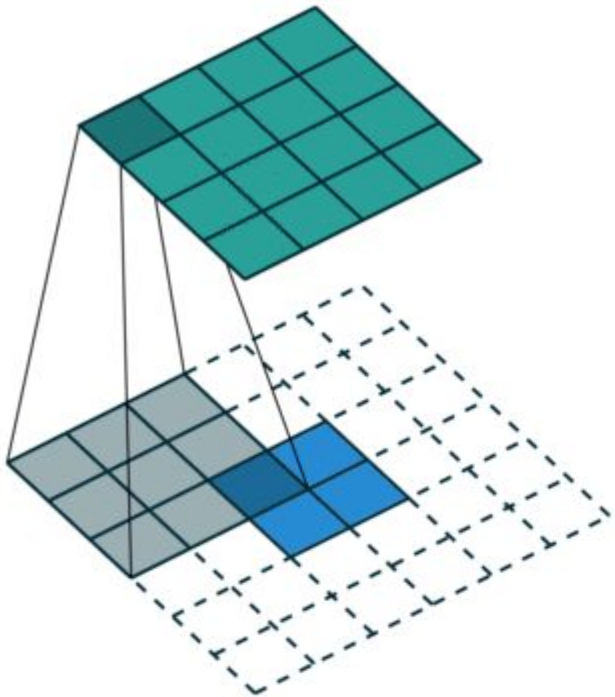
what if stride<1 ?



$$O = \frac{M - f + (p_{m-start} + p_{m-end})}{stride_m} + 1$$

Transposed (de)convolution Layer

- The transposed convolution is also known as deconvolution, or fractionally strided convolution in the literature.
- Deconvolution” is less appropriate, since transposed convolution is not the real deconvolution as defined in signal / image processing.
- There is something called deconvolution in signal processing. Technically speaking, deconvolution in signal processing reverses the convolution operation.



$$O = \frac{M - f + (p_{m-start} + p_{m-end})}{stride_m} + 1$$

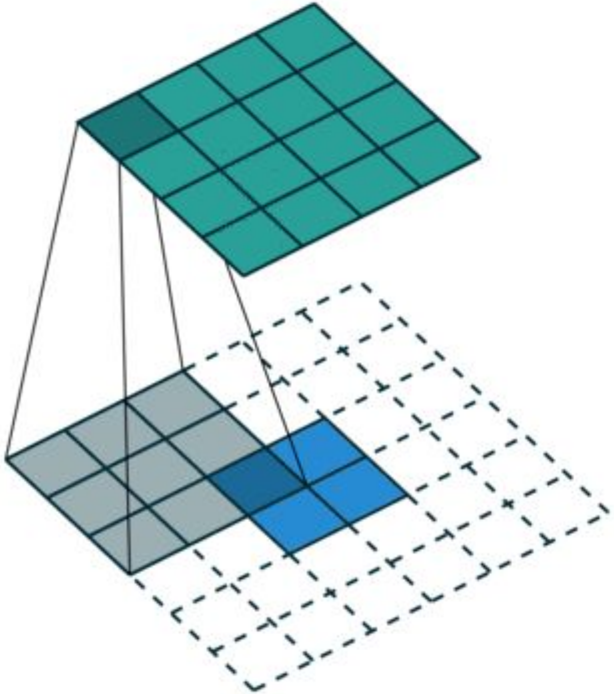
Transposed (de)convolution Layer

It is a means for upsampling. When you upsample you «interpolate» data.

- Nearest neighbor interpolation
- Bi-linear interpolation
- Bi-cubic interpolation, which one?

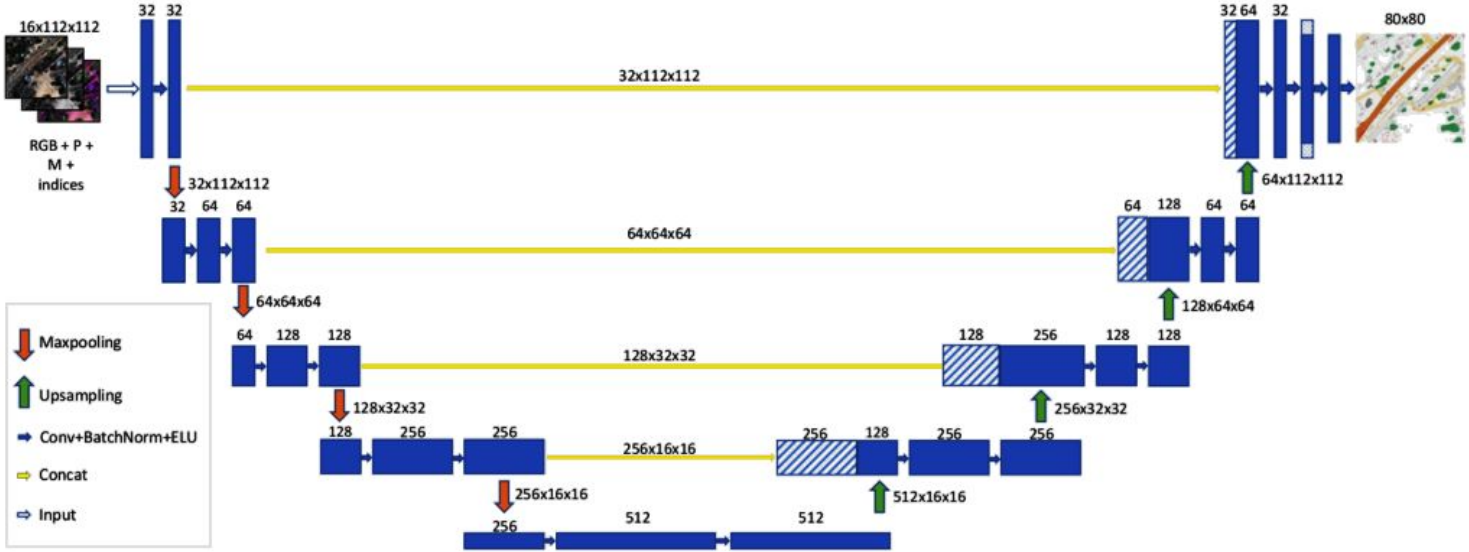
None of them! Deconv does it in a more clever (data driven) way ?

If we want our network to learn how to up-sample optimally, we use the transposed convolution.



Transposed (de)convolution Layer

Using Deconv, we build encoder/decoder architectures for semantic segmentation.
(like UNET - we'll see them later)



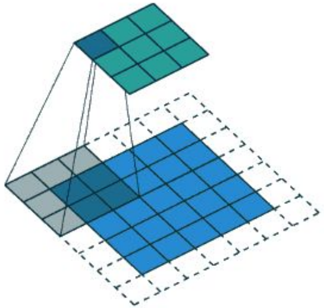
Dilated (atrous) Convolution Layer

When the filter grows, the number of parameters also grows.

How to obtain a larger filter, with less number of filter coefficients (parameters)!

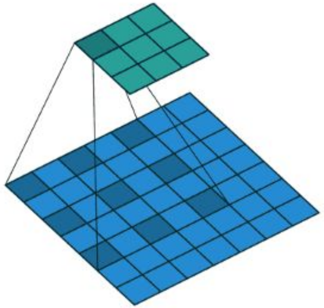
- Filter below has 3x3 coefficients, but it behaves like 5x5
- The empty points are called the dilation parameter (d)
- $f\_new = f + (f - 1) \cdot d$

Larger receptive field, with less number of parameters



$$O = \frac{M - (f_{new} = 5) + (p_{m-start} + p_{m-end})}{stride_m} + 1$$

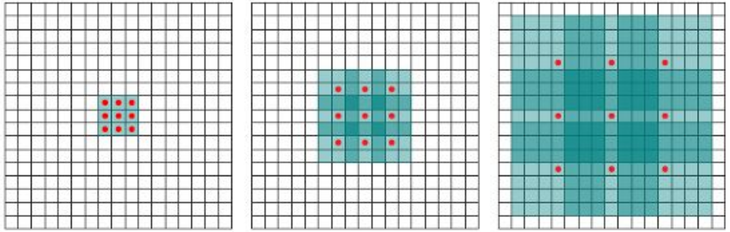
dilation parameter = 1



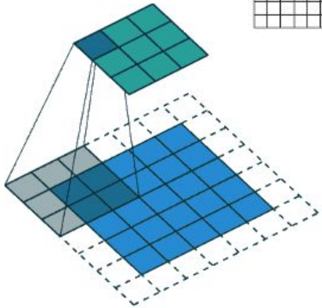
Dilated (atrous) Convolution Layer

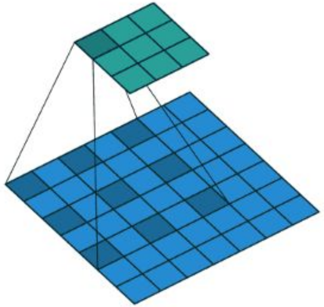
Dilated convolution is a very important tool for optimizing deep conv nets.

It helps us grow the receptive field, with less calculation requirement.



dilation parameter = 1





$$O = \frac{M - (f_{new} = 5) + (p_{m-start} + p_{m-end})}{stride_m} + 1$$

Group Convolution Layer

In group convolution we group the input depth and the filters into groups.

For example:

- the input is M x N x K
- The filter is f x g x L
- We have 2 groups
- So the number of weights is
    - f x g x K x L
- Number of bias is
    - L single for each

5	pool1 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5x5x48 convolutions with stride [1 1] and padding [2 2 2 2]	Grouped Convolution	27×27×256	Weights 5×5×48×128×2 Bias 1×1×128×2

Group Convolution Layer

In group convolution we group the input depth and the filters into groups.

Why ?

- Less number of operations
- Again for optimizing the model.

5	pool1 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 2 groups of 128 5x5x48 convolutions with stride [1 1] and padding [2 2 2 2]	Grouped Convolution	27×27×256	Weights 5×5×48×128×2 Bias 1×1×128×2

Layers Types

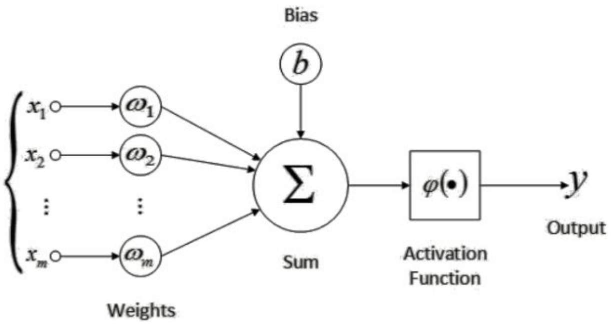
Let’s categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

Activation Layers

The results of the multiplication layer are always fed to an activation layer.

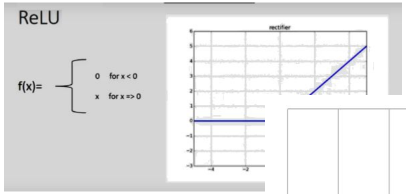
- The activation layer provides «nonlinearity».
- Different types:
    - ReLU
    - *sigmoid()*
    - *tanh()*
    - others...

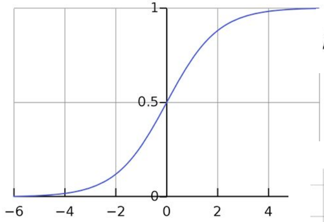


Activation Layers

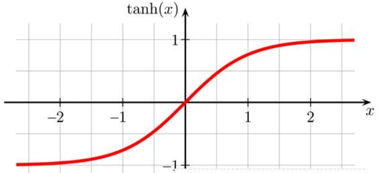
The results of the multiplication layer are always fed to an activation layer.

- The activation layer provides «nonlinearity».
- Different types:
  - ReLU: (stop if negative, pass if positive)
  - *sigmoid()*: (bounded between 0 to +1)
  - *tanh()*: (bounded between -1 to +1)
  - others…





$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$



$$\tanh x = \frac{\sinh x}{\cosh x}$$

They DO NOT CHANGE THE INPUT/OUTPUT SIZE !!!

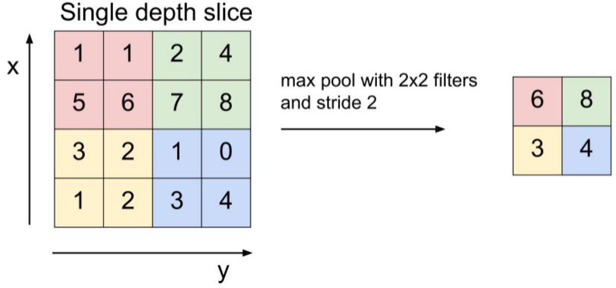
Layers Types

Let's categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

8	norm2 cross channel normalization with 5 channels per element	Cross Channel Normalization	27×27×256	-
9	pool2 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-

Sampling Layers

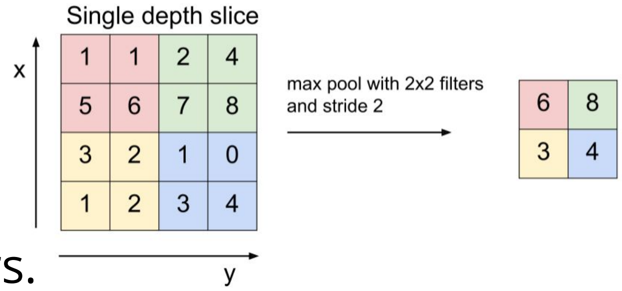


These layers change sample the input layer

- input layer has N nodes
  - Downsampling by 2 makes it N/2 nodes
  - Upsampling by 2 makes it 2 x N nodes.
  - Does not change the depth! Only the spatial dimensions.
- Some sampling layers do downsampling, some do upsampling
  - Max-pool (downsample by selecting the maximum value)
  - Average-pool (downsample by averaging)
  - Unpool: upsample (by a criteria)

8	norm2 cross channel normalization with 5 channels per element	Cross Channel Normalization	27×27×256	-
9	pool2 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-

Sampling Layers



The mathematics (Input/Output relation) of Sampling layers is the same as convolutional layers.

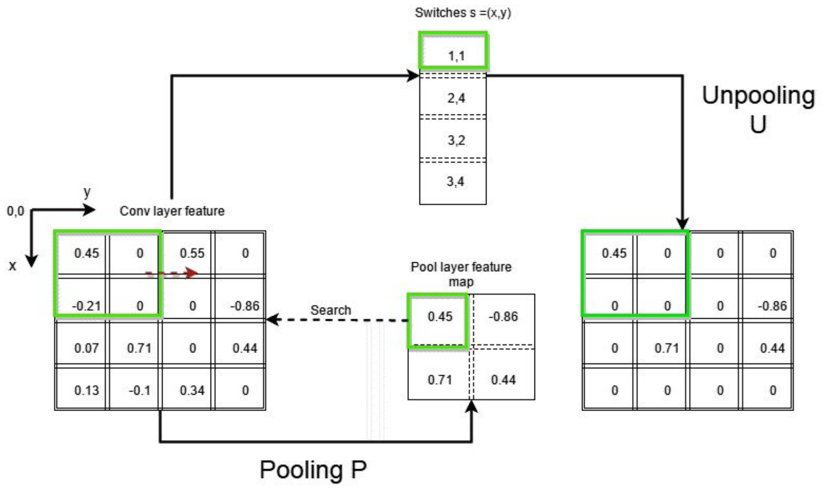
- (Remember the formula) The output size of a 2D (conv/sampling) layer with input $M \times N$ and with filter size $k \times l \times F$, with padding of P (on both sides) and a stride of S, will be $W \times H \times F$:

$$W = \frac{M - k + 2P}{S} + 1, \; H = \frac{N - l + 2P}{S} + 1,$$

Sampling Layers

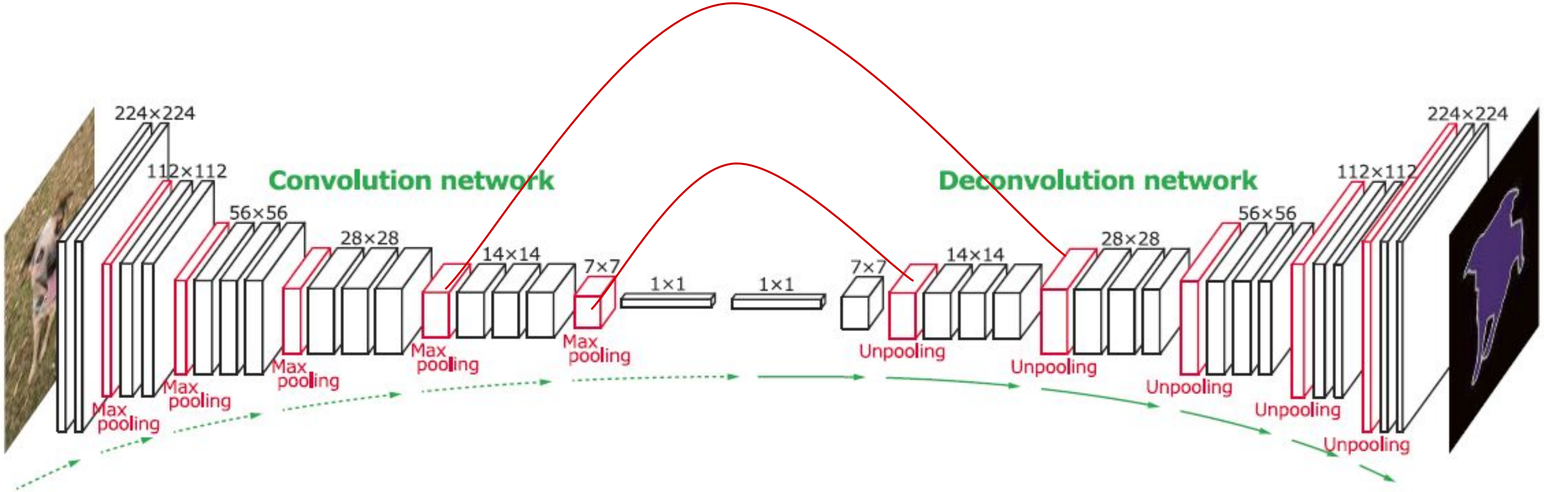
These layers change sample the input layer

- input layer has N nodes
  - Downsampling by 2 makes it N/2 nodes
  - Upsampling by 2 makes it 2 x N nodes.
  - Does not change the depth! Only the spatial dimensions.
- Some sampling layers do downsampling, some do upsampling
  - Max-pool (downsample by selecting the maximum value)
  - Average-pool (downsample by averaging)
  - Unpool: upsample (by a criteria)



Sampling Layers

- Unpooling can be used instead of deconvolution (bc it is cheaper).
- But how to pass that information?



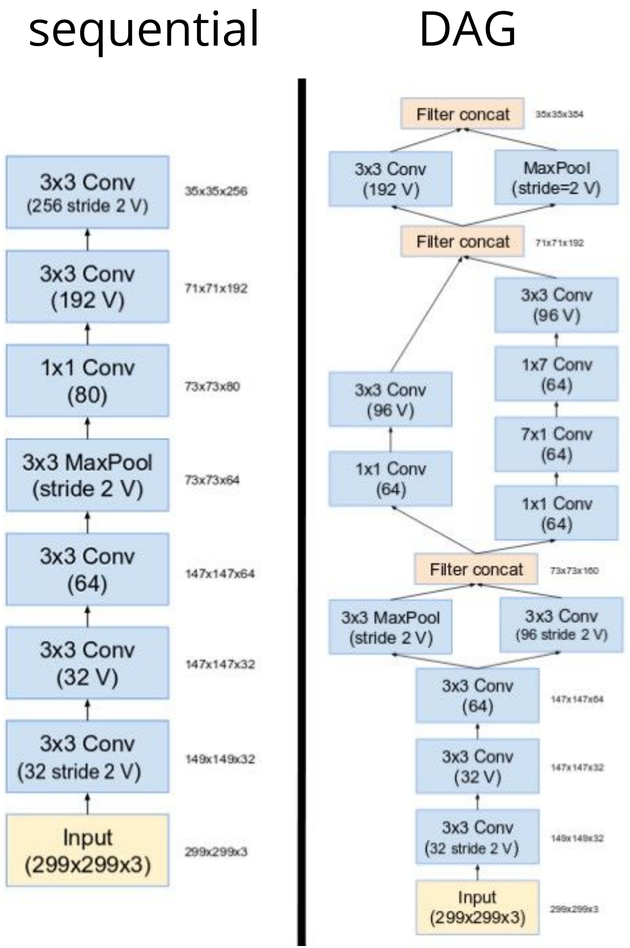
Layers Types

Let’s categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

Combination Layers

- Combination layers combine (or split) output of layer to another layer which is not sequentially the next.
- Hence, combination layers convert sequential (simple/serial) networks (such as AlexNet), into Directed Acyclic Graphs (DAGs).

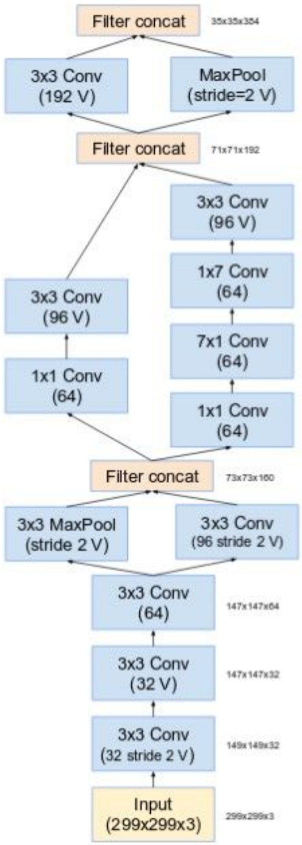


Combination Layers

There are different ways to combine/split layers.

- The most common combination layers (i.e. widely used in many architectures)
    - Addition Layer
    - Multiplication Layer
    - Concatenation Layer
        - Spatial concat, depth concat,...
    - Split Layer
- When combination layers combine different parts of the network. They are also called “skip connections”.

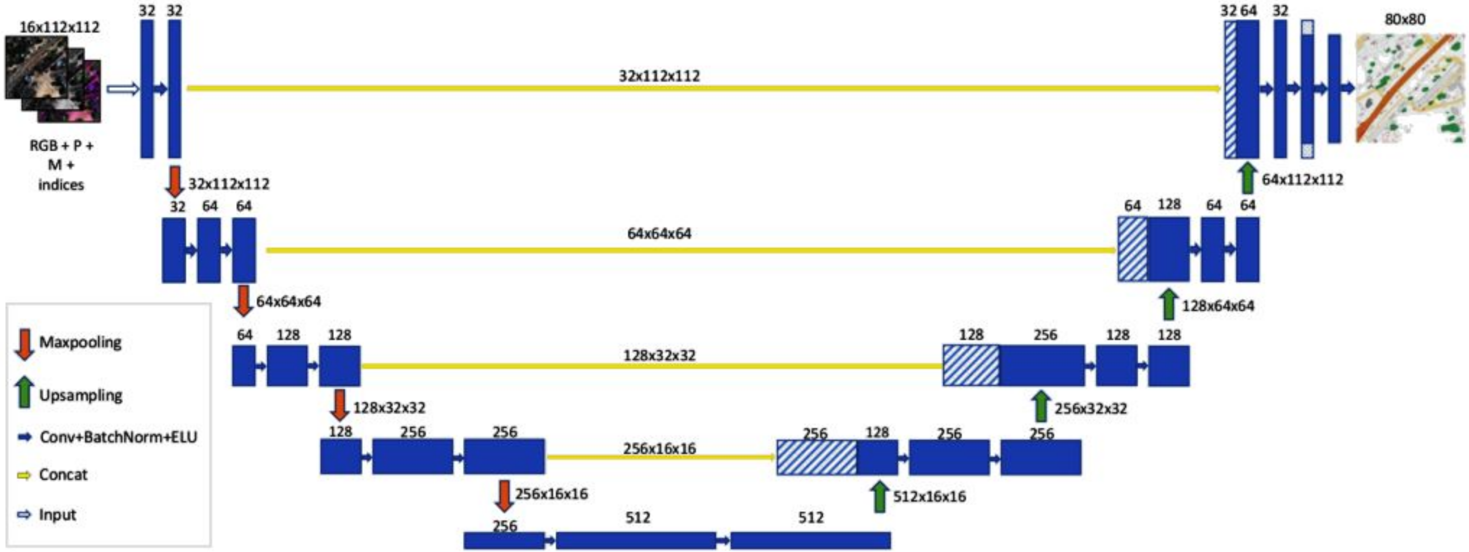
DAG



Combination Layers

For example:

- UNet uses skip connections with a “Depth Concatenation” layer.



Layers Types

Let’s categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

Input Layers

These are the first layers of a neural net.

- We feed our signal/data to these layers.
- They are simple. If the input has N values, then the output is N as well.
- The purpose of these layers are some sort of pre-processing/transformation, depending on the signal/problem/model.
- A common type of transformation at the input layer is normalization.
    - Zero-mean
    - Unity-variance

Input Layers

Zero-mean mean input normalization.

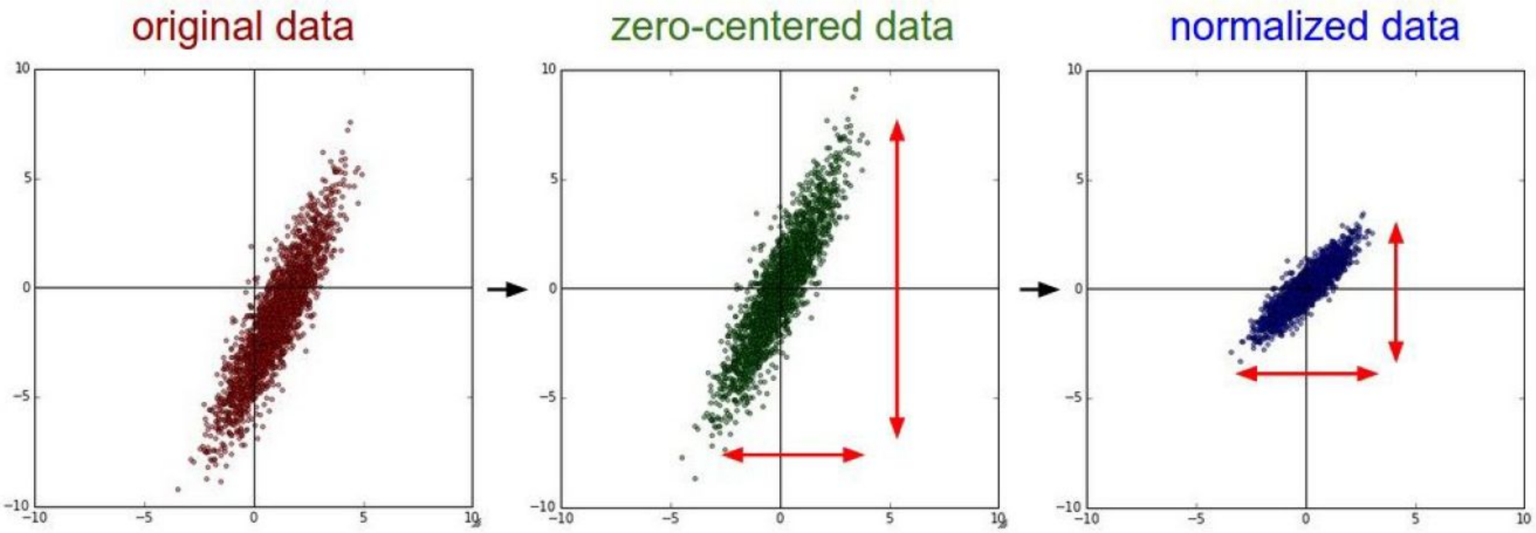
- We just subtract the mean image (of the training set) from every input.
  - So input pixels become both negative and positive valued.
- So if you are using a pretrained network (like AlexNet) the guys (Alex et al.) who trained the network (who magically found the weights for us) also did this:
  - They took the average of the training set.
- Average per channel or per pixel (it is per pixel in AlexNet)
  - Practice shows that it does not make much a difference.

Input Layers

Zero-mean mean input normalization.

- But Why?
- Normalizing the input signals around a zero mean, helps (speeds up) the optimization process (complex math, will not get into detail).
- Normalization allows us to use larger learning rates, bc normalized inputs reduce the risk of exploding gradient.
  - Exploding gradient: the problem of back-propagated gradient becoming too large.

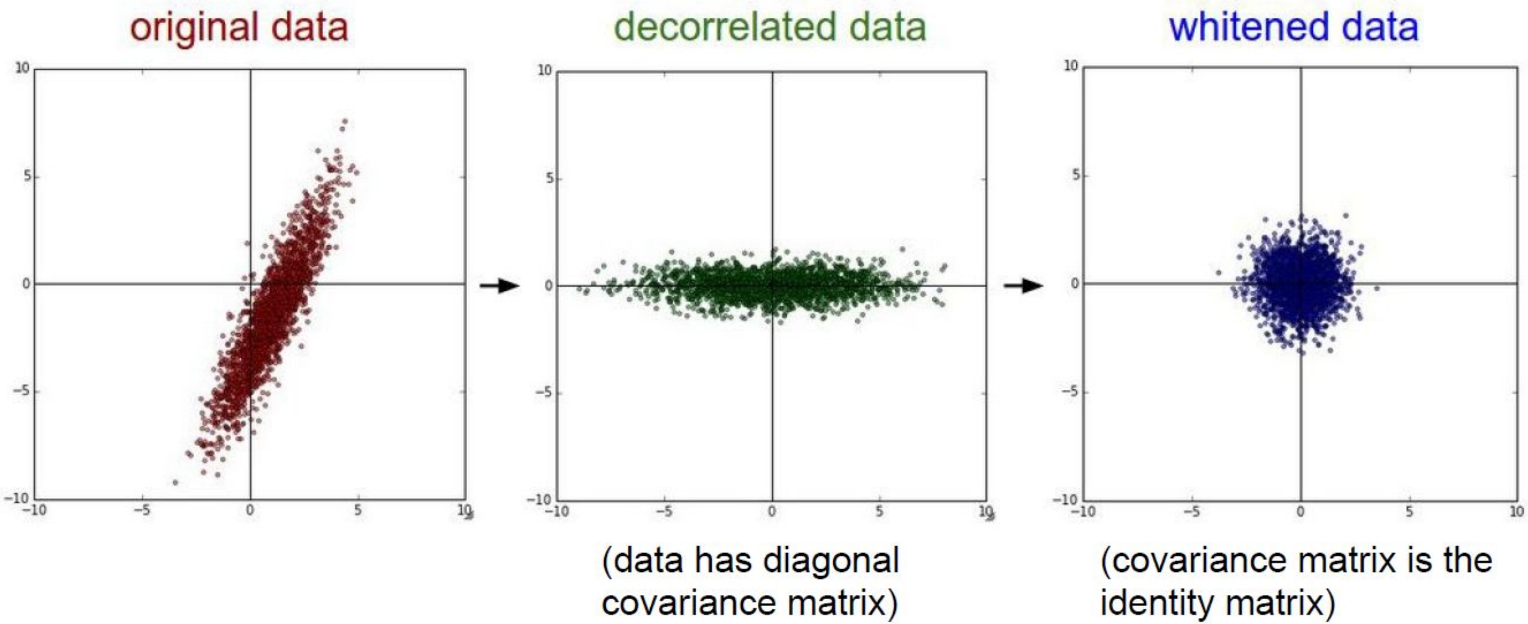
Input Layers




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

Input Layers



Layers Types

Let's categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

Output Layers

These are the last layers of a neural net.

Output layers provide the “score function” Depending on the problem type (classification, regression etc) the layer type changes.

- Classification problems usually use Soft-Max (or similar),
  - in order to create a probability distribution function.
- Regression problems usually use a sigmoid (scaled if necessary).
  - Or sometimes no output layer. Just the output of the previous dense layer.

Layers Types

Let's categorize layers first:

- Weight Multiplication Layers (conv, dense, deconv, groupConv, etc)
- Activation Layers (ReLU, sigmoid, tanh, etc)
- Sampling layers (maxpool, avgpool, unpool, etc)
- Combination Layers (concat, skip connections, etc)
- Input Layers (input normalization/shaping/processing)
- Output Layers (classification-softmax, regression-sigmoid, etc)
- Utility layers (dropout, batch-norm, etc)

Utility Layers

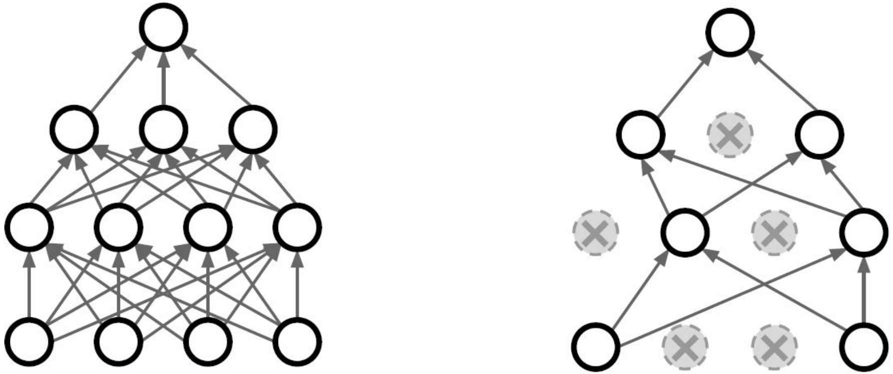
- Utility layers are utilized only for training.
- So when the training ends, and you have your trained weights, you may (or may not) need these layers.
- Most common types are:
  - Drop-out
  - Batch Normalization

METU

Drop-Out Layer

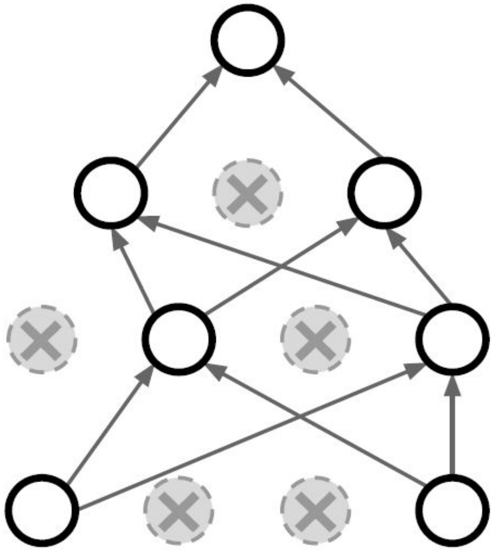
Drop-out is a type of “Regularization”.

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



Drop-Out Layer

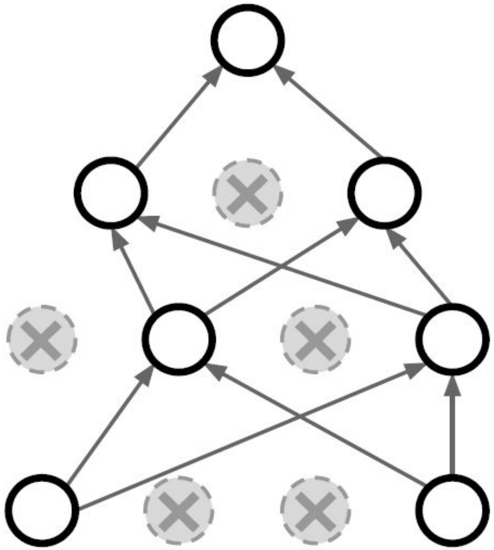
Drop-out is a type of "Regularization".



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Drop-Out Layer

During testing no drops occur!



- Instead the activations are multiplied by the drop-out probability.
- So during testing (inference), the drop-out layer is a scaling layer.

Batch Normalization

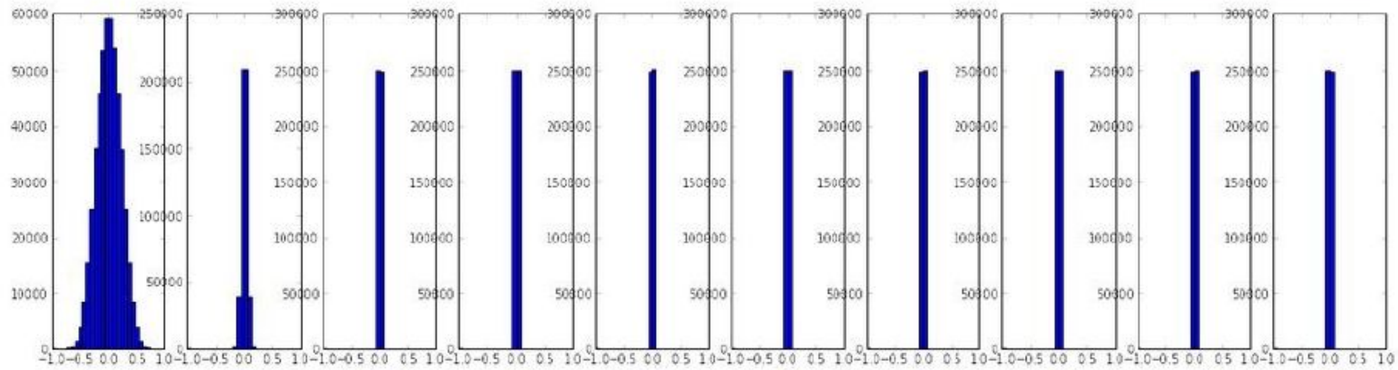
But before that, weight initialization!

- Proper initialization is an active area of research…
  - Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010
  - Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013
  - Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014
  - Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015
  - Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015
  - All you need is a good init, Mishkin and Matas, 2015

Weight Initialization

What happens when we randomly initialize weights?

- e.g.: Small random numbers (gaussian with zero mean and 1e-2 standard deviation)
  - This works fine for small networks,
  - but has problems with deeper networks.



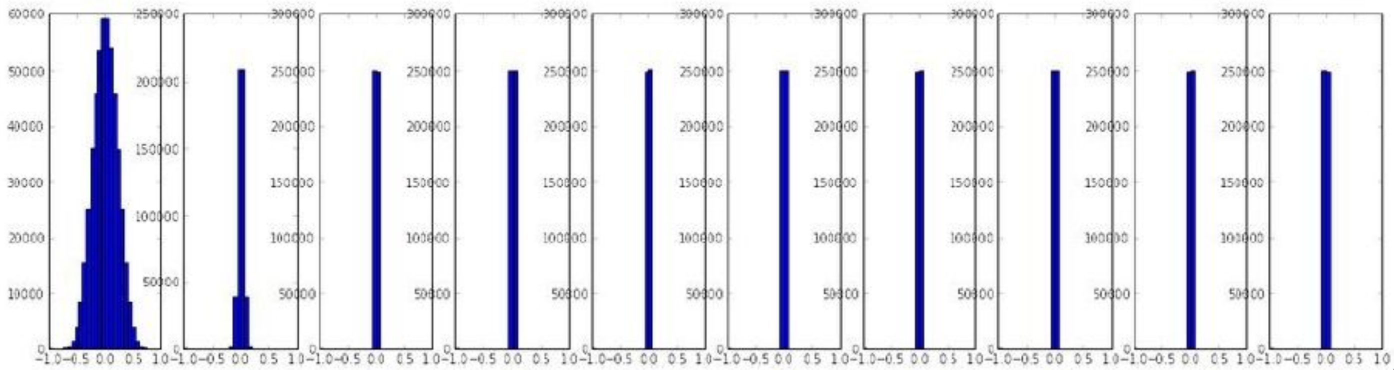
METU

Weight Initialization

$$f\left(\sum_i w_i x_i + b\right)$$

What happens when we randomly initialize weights?

- e.g.: Small random numbers (gaussian with zero mean and 1e-2 standard deviation)
  - This works fine for small networks,
  - but has problems with deeper networks.



activations die…

Error, Loss, L, all the same…

Derivative of Error with respect to weight

Old weight

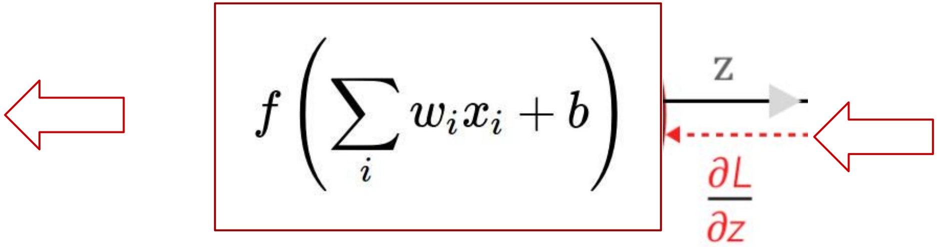
$$^{*}W_x = W_x - a\left(\frac{\partial Error}{\partial W_x}\right)$$

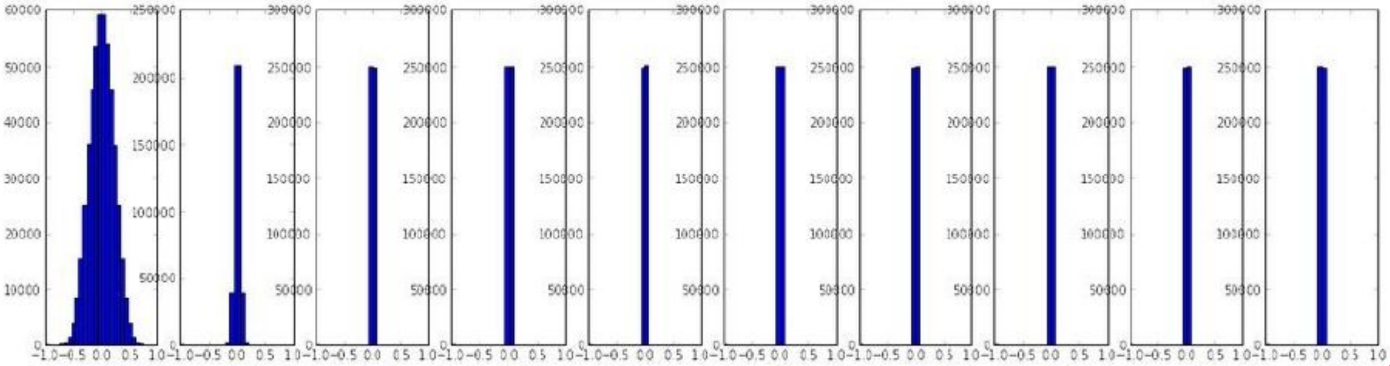
New weight

Learning rate

Weight Initialization

Gradients die because of the backward pass!





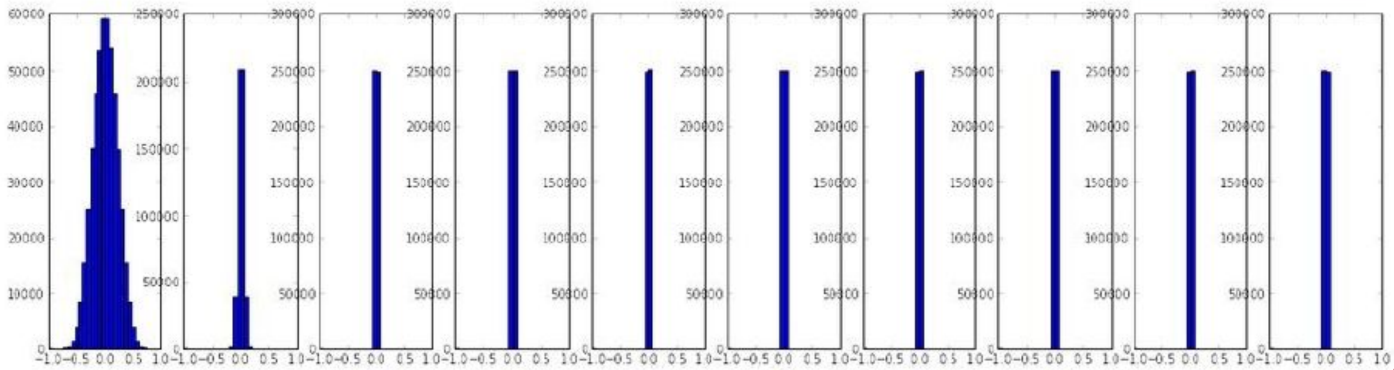
activations die…

Weight Initialization

$$f\left(\sum_i w_i x_i + b\right)$$

What happens when we randomly initialize weights?

- e.g.: Small random numbers (gaussian with zero mean and 1e-2 standard deviation)
  - This works fine for small networks,
  - but has problems with deeper networks.



activations die...

Weight Initialization

With better initialization, we can make them live longer (i.e. deeper)

- There are different weight init methods (available in libs)
- AT this level, it is difficult to indicate the best init method from scratch.
- Go find a similar architecture to yours, and imitate them.

WEIGHT INITIALIZATION TECHNIQUES FOR DEEP LEARNING ALGORITHMS IN REMOTE SENSING: RECENT TRENDS AND FUTURE PERSPECTIVES

Wadii Boulila[1,2], Maha Driss[1,2], Mohamed Al-Sarem[2], Faisal Saeed[2], Moez Krichen[3,4]

[1]RIADI Laboratory, National School of Computer Sciences, University of Manouba, Tunisia
[2]IS Department, College of Computer Science and Engineering, Taibah University, Saudi Arabia
[3]CS Department, Faculty of CSIT, Al-Baha University, Saudi Arabia
[4]ReDCAD Laboratory, University of Sfax, Tunisia

INITIALIZATION IN RS DOMAIN

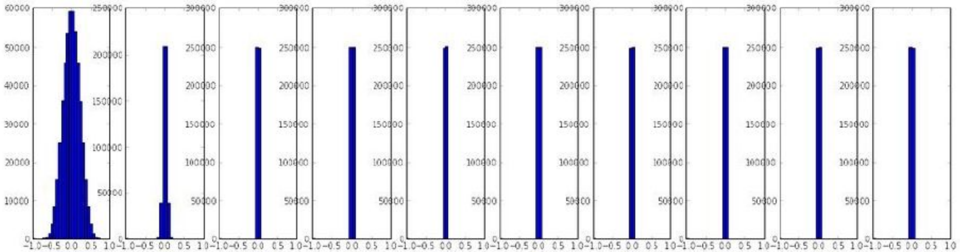
Reference	Application case study	DL model	Weight initialization method
[10]	Semantic segmentation of small objects	CNN	He
[11]	Classification of hyperspectral image	CDL	Auto-encoder
[12]	Automatic target recognition in synthetic aperture radar	CNN	Random distribution
[13]	Single image super-resolution	CNN	He
[14]	Surface water mapping	CNN	He
[15]	Classification of oceanic eddies	CNN	Gaussian distribution
[16]	Hyperspectral image classification	CNN	MSRA (for Microsoft Research Asia)
[17]	Infrastructure Quality Assessment	CNN	Xavier
[18]	Remote sensing image fusion	CNN	He
[19]	Polarimetric synthetic aperture radar image classification	Complex-valued deep fully CNN	A new method is proposed
[20]	Hyperspectral image classification	CNN	Random distribution
[21]	Multi-label aerial image classification	CNN BiLSTM	Xavier
[22]	Building change detection	DBN ELM	Random distribution
[23]	Multispectral image classification	RNN LSTM	Xavier
[24]	Rice lodging canopy	CNN	Xavier
[25]	Road extraction	PSNet	A new method is proposed
[26]	Land cover change detection	LSTM CNN	Random distribution

Batch Normalization

How to make them (activations) live longer?

- Batch Normalization is an effort to make these activations *linger* for deeper architectures.
- The idea is to make the make the network more stable during training.
- How?

Batch Normalization



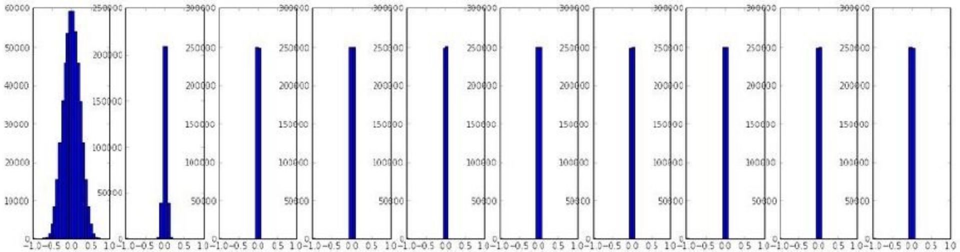
“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

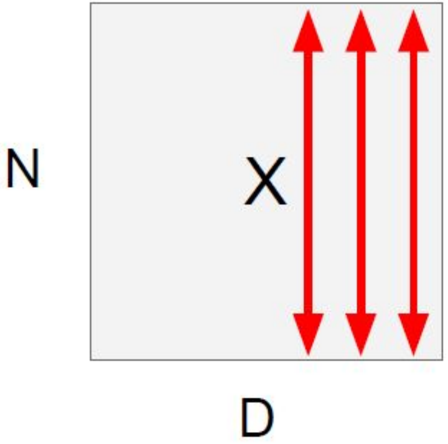
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization



"you want unit gaussian activations?
just make them so."

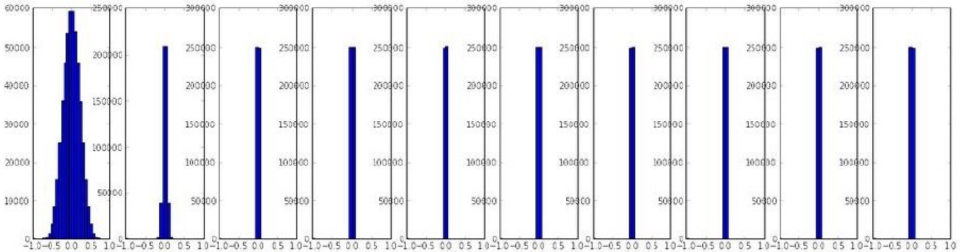


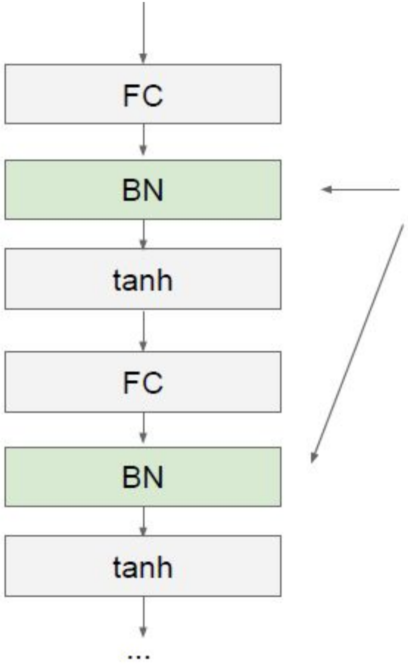
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization



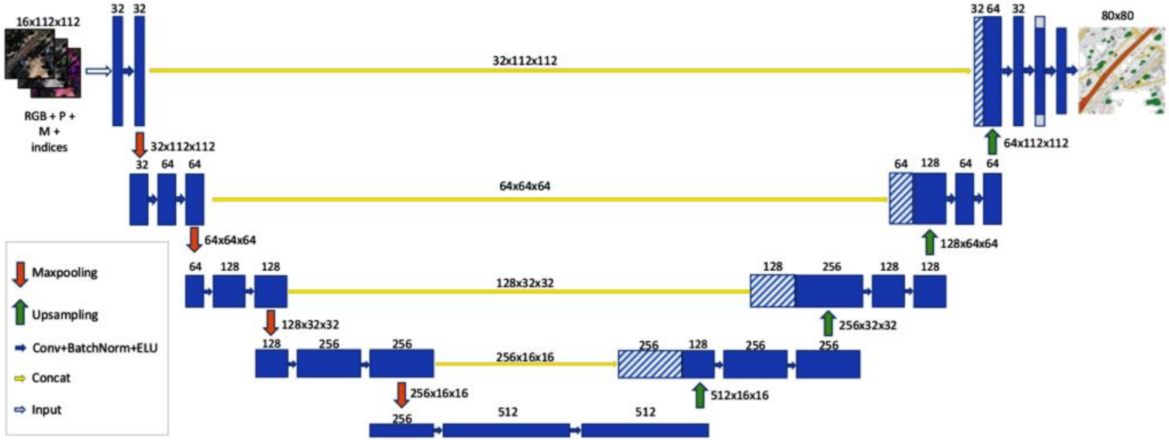


Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Live Code!

- Let's go over the entire UNet and let's visualise architecture.
- For this implementation I will be using MATLAB. But there are ways for it in Python as well.



What will we do next week?

- Starting with next week...
    - ➢ training...
- Following weeks
    - ➢ Introduction to Sequence Models, RNNs
    - ➢ LSTMs, and many applications of deep learning.