Chapter 11-12-13-14

Deep Learning – CNN – Architectures – Batch Normalization

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Deep Learning

In Deep Learning, we represent an image as a Tensor of integers between [0, 255]. For example, an RGB image of resolution 800x600 is a Tensor of shape [800, 600, 3].

We have to take into consideration that the pixel intensities change when we move our camera.

This is one of the main problems when dealing with images in the DL world. Other challenges could be background clutter, poor illumination, occlusion, deformation and so on and so forth.

For classifying an image as a cat or a dog, for example, we can use Machine Learning, which:

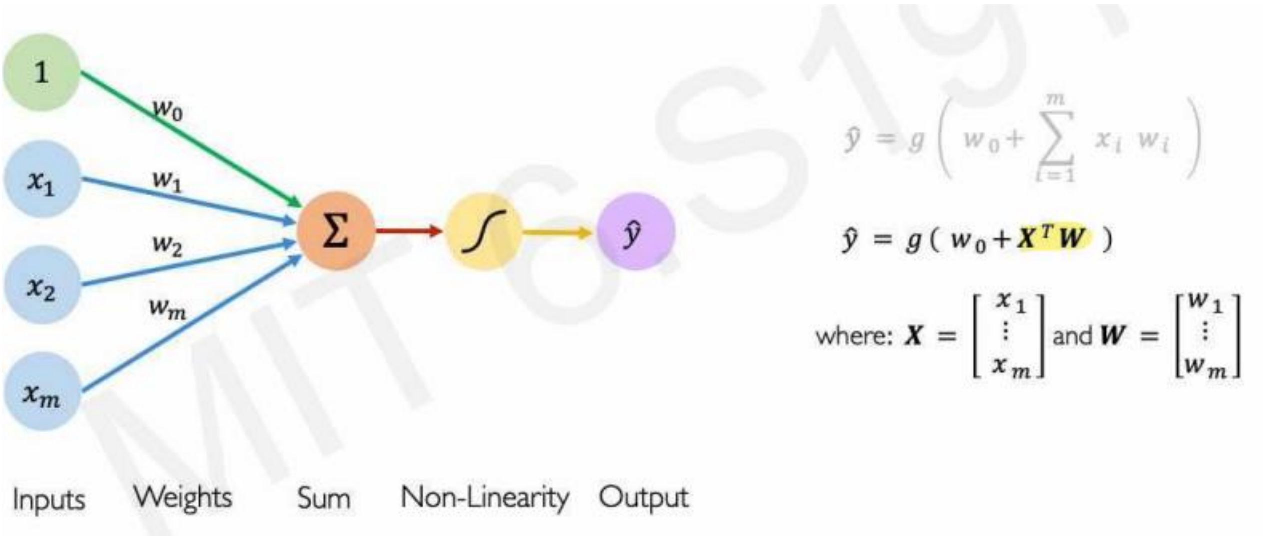
1. Collects a set of images and labels
2. Uses ML Algorithms in order to train a classifier
3. Evaluate this classifier on new images never seen before (test set).

But why, Deep Learning, is preferable over the ML algorithms? Neural Network architectures successfully deliver us the possibility to work with big data (so millions of images in a single Dataset), working with way faster computations (thanks to GPUs) and improved softwares (such as PyTorch / TensorFlow).

1.1 Perceptron

One of the main ML algorithms, which nowadays is still widely used, is called Perceptron.

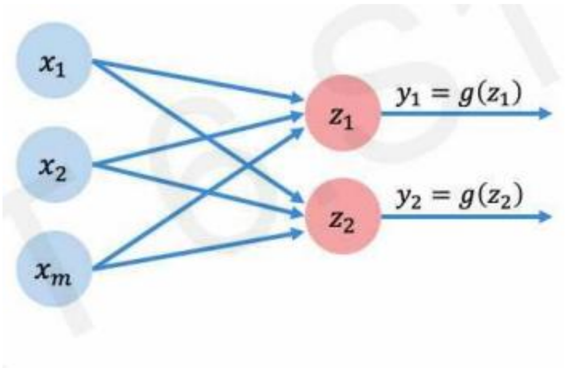
The architecture of the Perceptron is shown in the image below:



1
x_1
x_2
x_m
w_0
w_1
w_2
w_m
Σ
$\hat{y}$
$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$
$\hat{y} = g(w_0 + \boldsymbol{X}^T\boldsymbol{W})$
where: $\boldsymbol{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $\boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$
Inputs
Weights
Sum
Non-Linearity
Output

We can easily see how we can represent our input X as a vector of inputs and the Weight vector as a vector of Weights, such that $X^T W$ is a linear combination of inputs and w_0 is denoted as the bias term.

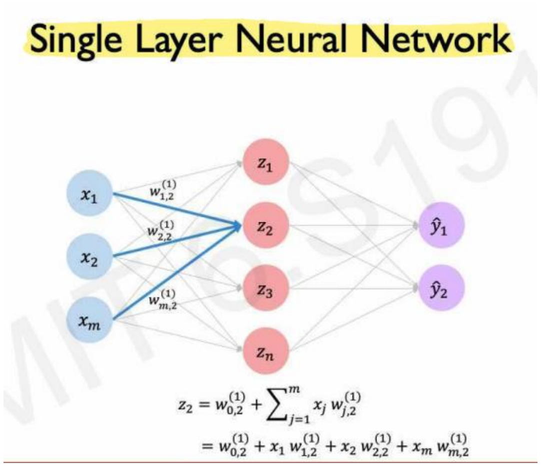
g is any non-linear activation function (such as Sigmoid, ReLU, tanh, etc.) which introduces non-linearity into the network, allowing us to approximate complex functions.



$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j \, w_{j,i}$$

Perceptron can also be stacked multiple times, denoting a Multi-Output Perceptron, where all inputs are densely connected to all outputs. This is the creation of the **Dense** Layers.

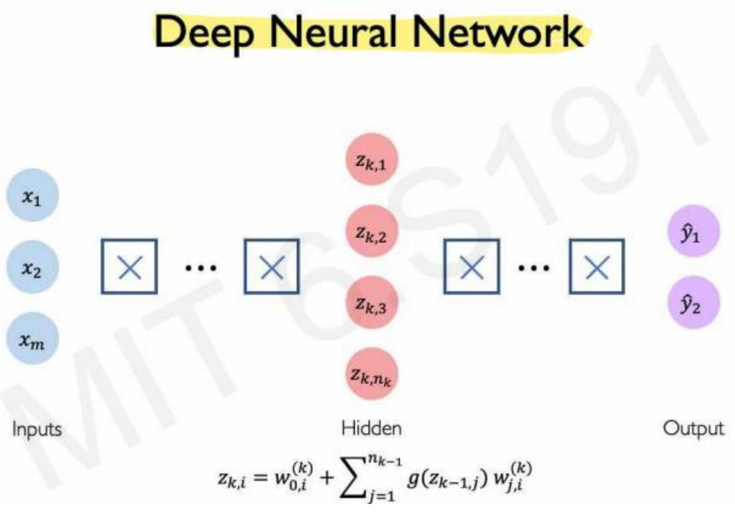
From this idea, we can start thinking about Neural Networks and how they are composed by stacking up multiple connections between input and output. This is a Single Layer Network where, we say, there is only one Hidden Layer.



$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \, w_{j,2}^{(1)}$$
$$= w_{0,2}^{(1)} + x_1 \, w_{1,2}^{(1)} + x_2 \, w_{2,2}^{(1)} + x_m \, w_{m,2}^{(1)}$$

1.2 Deep Neural Network

A deep Neural Network, instead, is formed by multiple Hidden Layers:

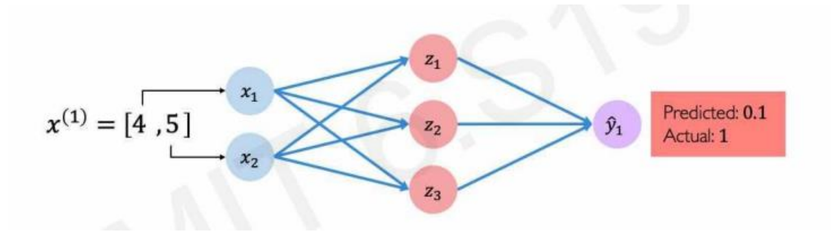


$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) \, w_{j,i}^{(k)}$$

Let’s take for example that we have a problem where we define the number of lectures you attend on the X-Axis and the number of hours spent on the final project on the Y-axis.

Let’s assume that the Neural Network, already has some information regarding possible combinations of X-Y such that I “Pass” the exam or “Fail” the exam (Binary Classification).

1.3 Loss functions



If we take as input the vector $\begin{bmatrix}4\\5\end{bmatrix}$ and we feed this into the network, we might have this kind of result where the predicted value is 0.1 and the true value being 1.

This introduces us to the Loss Function, which is a measure of quantifying cost incurred from incorrect predictions. We define the loss for <u>one single sample</u> as follows:

$$\mathcal{L}(f(x^{(i)};W),y^{(i)})$$

Where the first term $f(x^{(i)};W)$ is the predicted value from the Network, while $y^{(i)}$ is the true value.

If we sum all these costs, we end up with the Cost Function, which is the sum of all the costs, averaged by the number of samples:

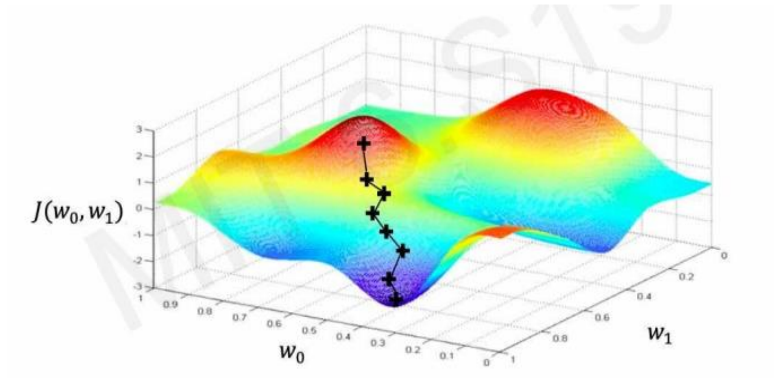
$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^{(i)};W),y^{(i)})$$

We have multiple losses, as per the Binary Cross Entropy Loss, Mean Squared Error (MSE), Mean Absolute Error (MAE) and so on and so forth.

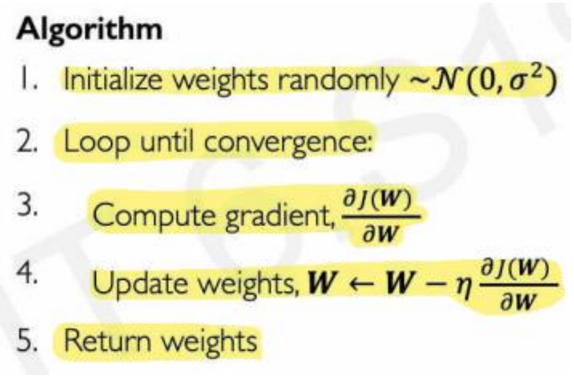
1.4 Loss Optimization

In order to have a proper Network, we must find the network weights which achieve the lowest loss. This is done through finding the argmin of the Loss Function we explained above:

$$\boldsymbol{W}^* = \underset{W}{argmin}\, J(\boldsymbol{W})$$



But how exactly we compute this optimization? Well, this is done through a random pick of weights (w_0, w_1), computation of the gradient w.r.t. the weights $\frac{\partial J(W)}{\partial W}$, taking a small step in the opposite direction of the gradient and we repeat this process until we reach convergence (**Gradient Descent**).



Algorithm

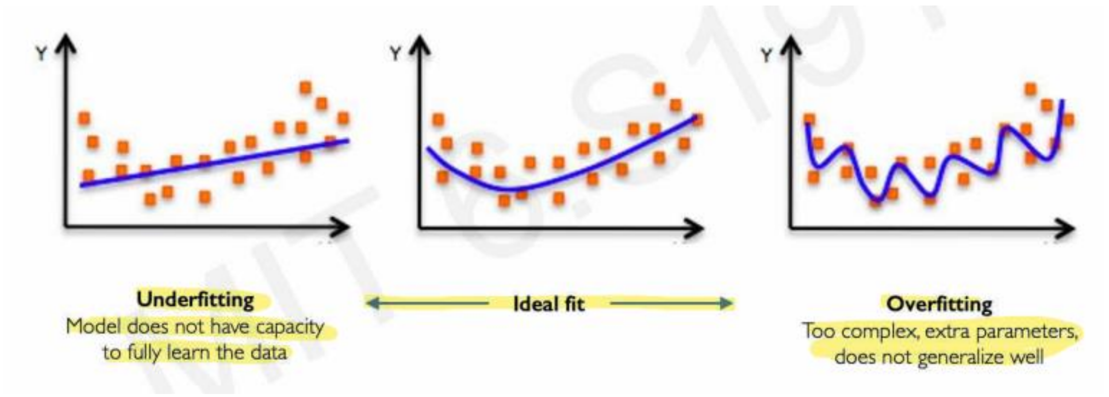
```
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3.     Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4.     Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights
```

We compute the gradient through the backpropagation algorithm, which uses the chain rule to update the gradients w.r.t to previous gradients. This is the key point for understanding that a small change in one weight (for ex. w_2) affects the final loss $\boldsymbol{J}(\boldsymbol{W})$

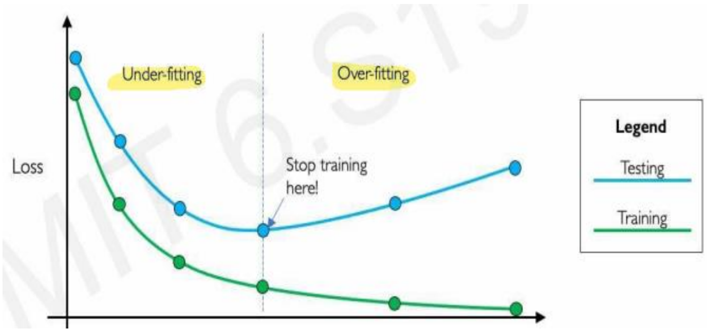
Neural Networks are difficult to train and optimize, that's why we use and adaptive learning rate which adapts itself during training, depending on how large the gradient is or how fast learning is happening or, in the end, given a size of a particular weight. The most used optimizers used are *SGD, Adam, Adadelta, Adagrad.*

One important thing to notice and put particular attention to is the computation of the gradient, since it can be computationally expensive to compute. That's why we use the Stochastic Gradient Descent, which is relatively faster to compute but **very noisy** (since it's stochastic)! We compute this by easily taking a batch of data points and not all the data points at once. This ensures smoother convergence and a more accurate estimation of the gradient.

1.5 Overfitting issue



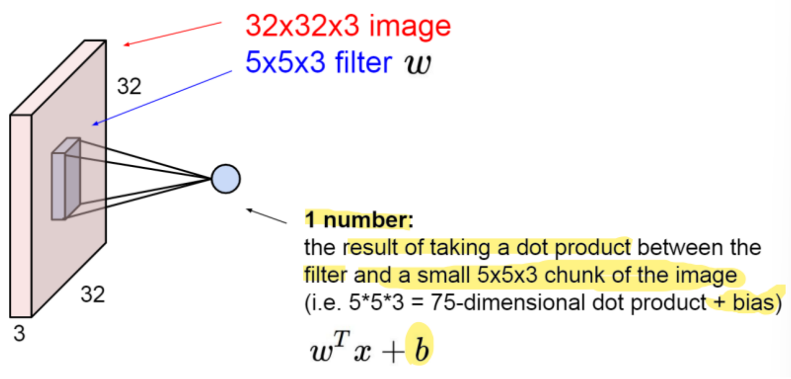
Another really common problem in the NN environment is the problem of overfitting, where our model does not generalize well on new data. We can solve this problem through **Regularization**, which constrains our optimization problem to discourage complex models. This is needed in order to improve generalization of our model on unseen data, as we said earlier.



- **Dropout**: During training, we set some neurons to 0 inside our Hidden Layers.
- **Early Stopping:** We stop training loop before we overfit (based on the patience value, which on the left image is set to **1**).
- **Data augmentation** (rotation, scaling, translation, stretching, flips)

2. CNNs

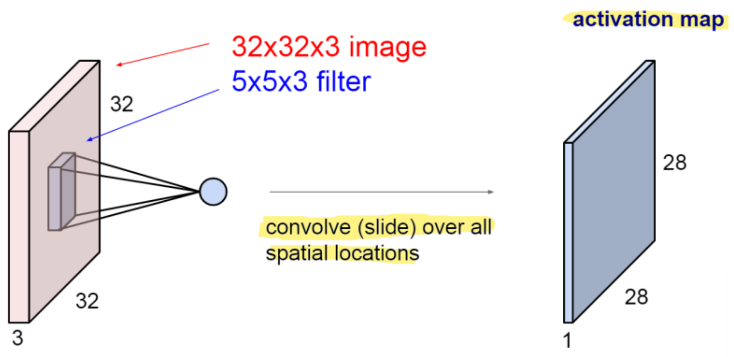
As we know already, a Fully Connected Layer correlates weights with inputs, while in the NN architecture a Convolutional Layer preserves the spatial structure by passing a **window** W of fixed shape through the image. We also refer to the window as "**Filter**".



32x32x3 image
5x5x3 filter w
32
32
3
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)
$w^T x + b$

On the left we find the initial scheme of a Convolutional Layer. The filter always extends the full depth of the input volume (3 channel input -> At least 3 channels for the filter). If we slide this filter over all spatial locations, we have an output denoted as "Activation Map".

We can also stack multiple filters on top of each other. For each one of them, we'll get a corresponding activation map. By applying a Convolutional Layer, followed by a ReLU or any activation function, we are essentially building a Convolutional Neural Network (CNN), which can be really powerful for extracting meaningful features for the final classifier. We call the layer convolutional because it is related to a convolution of two signals (img/filter).

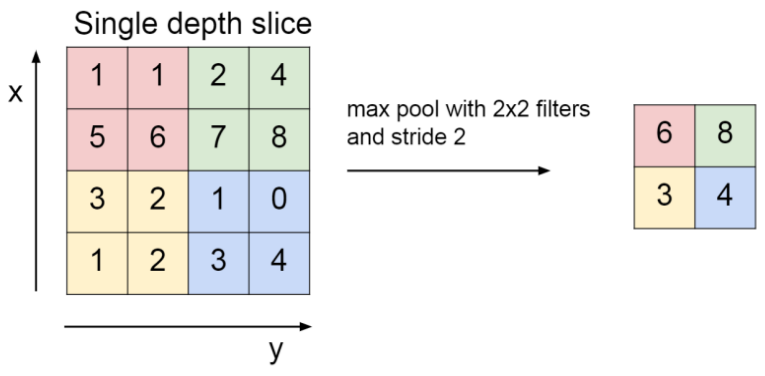


We can apply the stride value to a filter, in order to let the window slide by a fixed amount of pixels (stride). In general, it is common to see Convolutional Layers with stride 1, in order to preserve spatial information.

What happens if we apply a bigger filter which doesn't fit our initial image shape? We can't (in theory), but the Padding operation comes in handy, so that we add a 1-pixel border full of zeros in order to let the window/filter fit inside our image.

If we sum the parameters as **K** (number of filters), **F** (filter size), **S** (stride), **P** (padding) we can ascertain that a Convolutional Layer will produce an output of $W_2\ x\ H_2\ x\ K$.

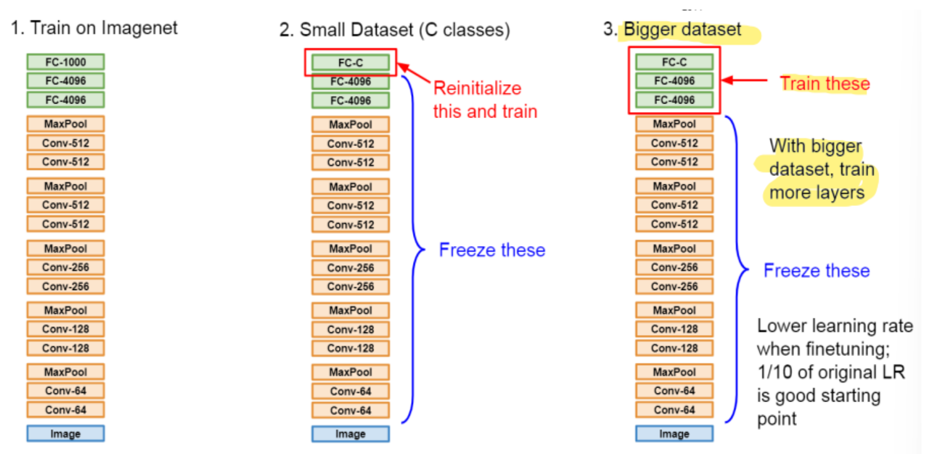
Finally, we explain what is the Pooling operation, a method which makes the representations smaller and more manageable by reducing the spatial size of the image through a fixed window which takes as input the pixels of the image via techniques:



- **Max pooling**: Takes the maximum pixel in a window of fixed shape (the one on the left).
- **Avg pooling**: Takes the average pixel in a window of fixed shape.

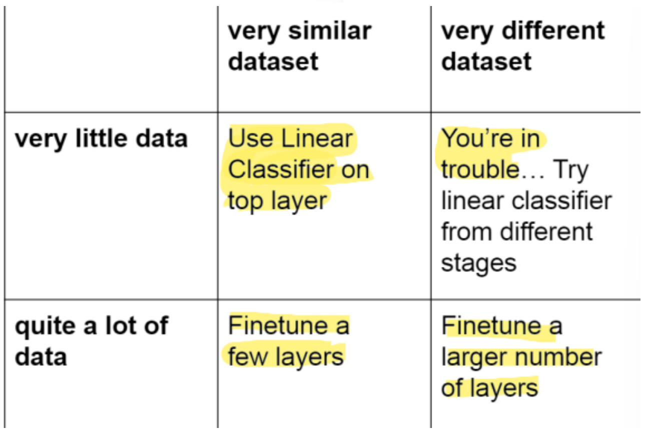
3. Architectures

When we talk about Neural Networks and DL, we can't forget their main architectures and their main methodologies for training a NN.



We are talking about Transfer Learning, where we retrieve a model, freeze its layer, remove the top block and put a block related to our Dataset problem/task.

Concerning *CNN* architectures, we definitely think about **LeNet-5, AlexNet** (first use of CNN to train Computer Vision models), **VGG** (which showed that bigger networks work better), **ResNet** (extremely deep networks!) and so on and so forth (**EfficientNet, MobileNet**, etc.)



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

4. Batch Normalization

Batch Normalization is a technique which speeds up training and standardizes the inputs to a layer for each mini-batch, rescaling data to have a mean of zero and a standard deviation of one. This is done through two learnable parameters γ, β which are respectively σ, μ.

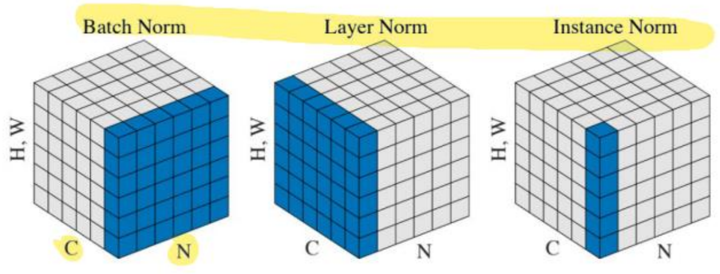
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

It is inserted after a Fully Connected Layer or a Convolutional Layer, before the non-linearity and computed as per the image on the left. There are also multiple variations of the Batch Normalization technique, such as Layer Normalization and Instance Normalization.



Questions by professor:

- **“What is the main invention that enabled 128-Layers?”** -> Batch normalization, ResNet etc.
- **“What is the idea behind fine-tuning?”** -> We have to explain the transfer-learning technique.