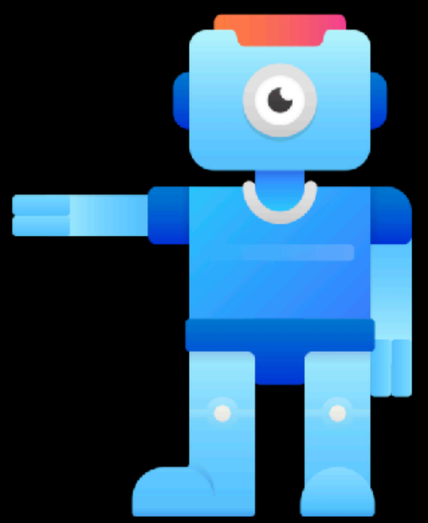


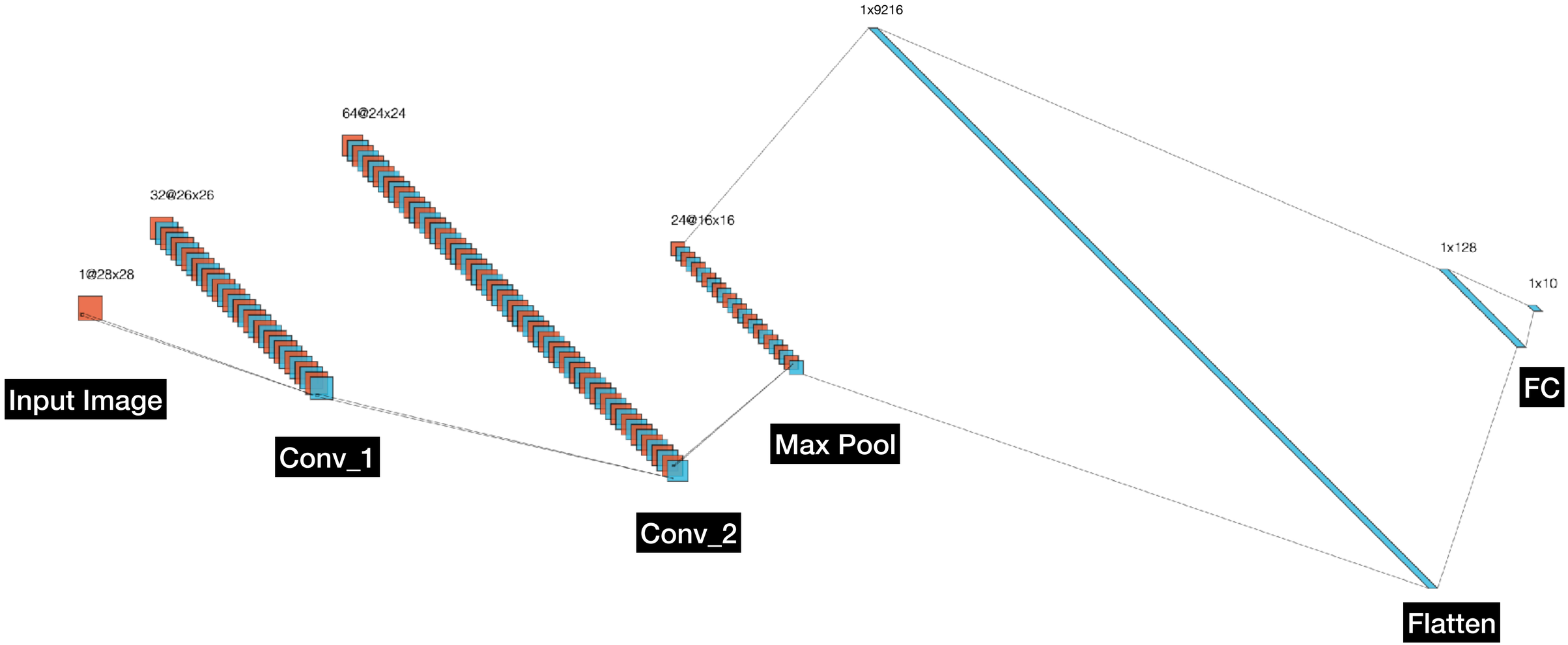
MODERN COMPUTER VISION

BY RAJEEV RATAN

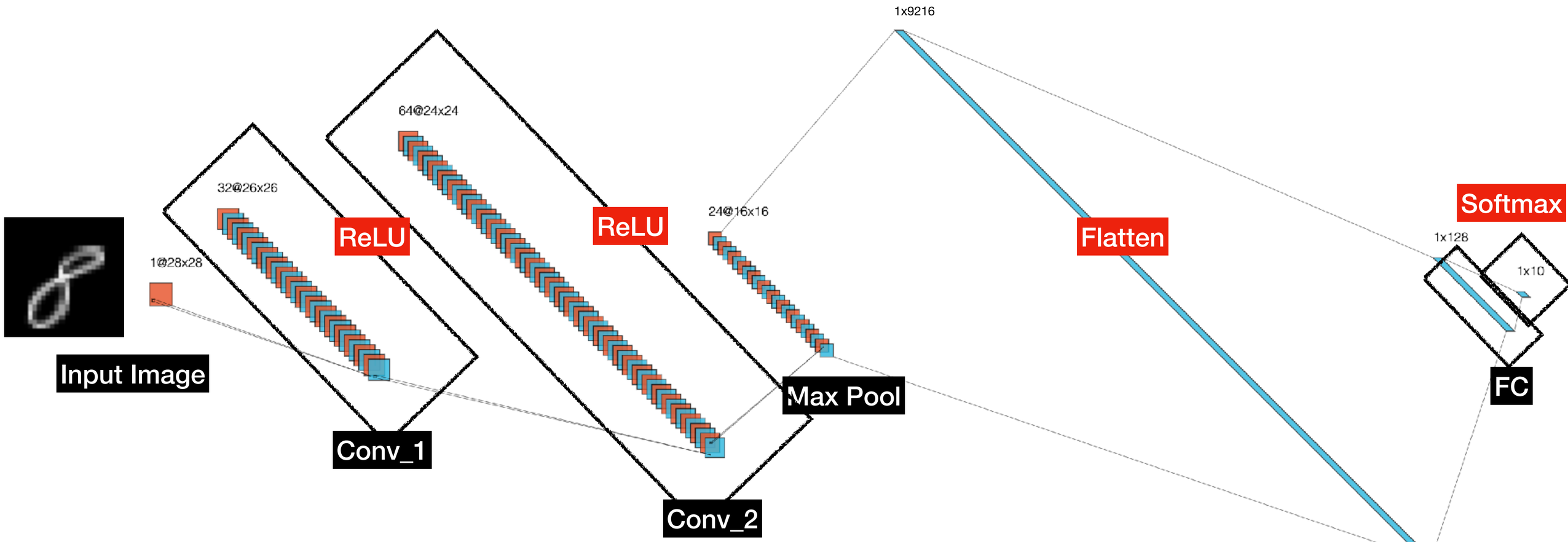
Building a CNN

Let’s put all the pieces together and build a CNN

A Simple CNN

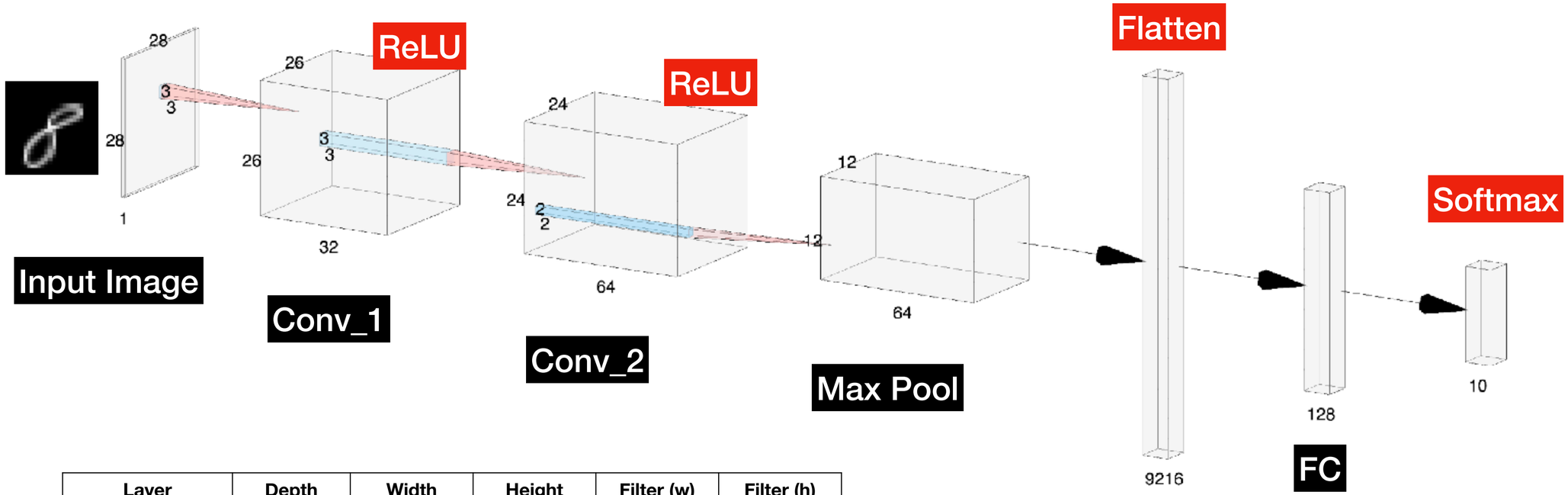

1x9216
64@24x24
32@26x26
24@16x16
1x128
1x10
1@28x28
Input Image
Conv_1
Conv_2
Max Pool
Flatten
FC

A 4 Layer Deep CNN for MNIST



Example of a 4 Layer CNN we can use for the MNIST Dataset

Another Representation

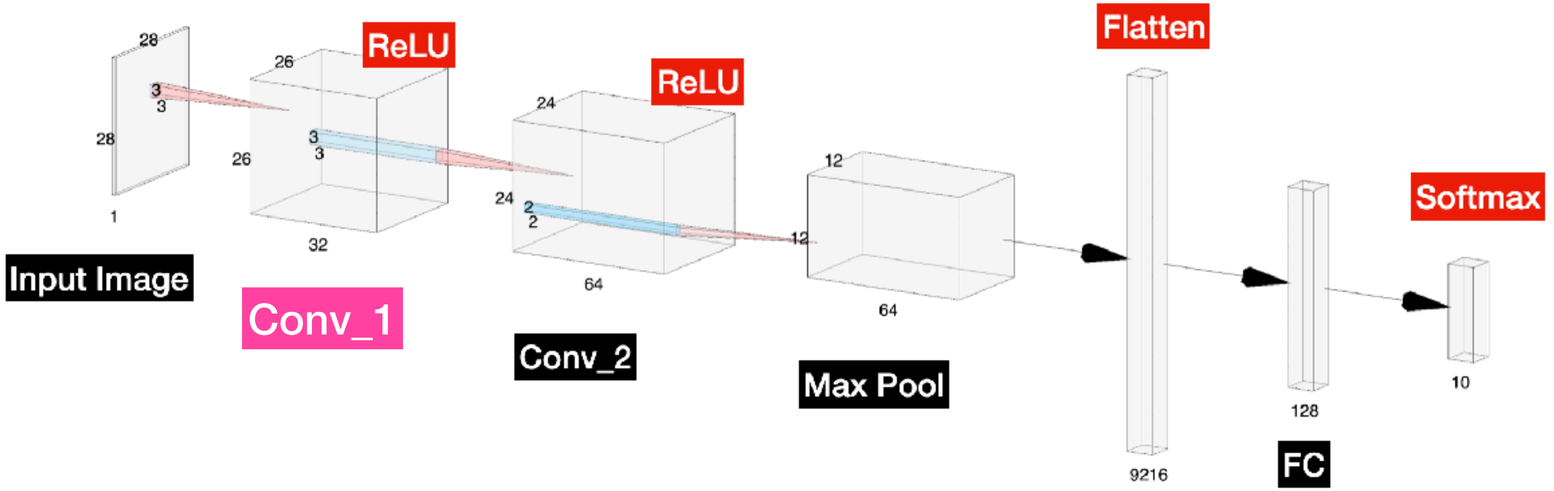


Layer	Depth	Width	Height	Filter (w)	Filter (h)
Input	1	28	28		
Conv_1	32	26	26	3	3
Conv_2	64	24	24	3	3
Max Pool	64	12	12	2	2
Flatten	9216	1	1		
Fully Connected	128	1	1		
Output	10	1	1		

Notes:

- We choose 32 & 64 Filters or Kernels for Conv_1 & Conv_2
- We choose to
- The Feature Maps are shown as Conv_1 and Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2

Calculating the Output Size of Conv_1



Notes:

- We choose 32 Filters or Kernels for Conv_1
- The Feature Maps are shown as Conv_1 & Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2

$$(n \times n) * (f \times f) = (\frac{n + 2p - f}{s} + 1) \times (\frac{n + 2p - f}{s} + 1) = (\frac{28 + (2 \times 0) - 3}{1} + 1) \times (\frac{28 + (2 \times 0) - 3}{1} + 1) = \text{26 x 26}$$

Where

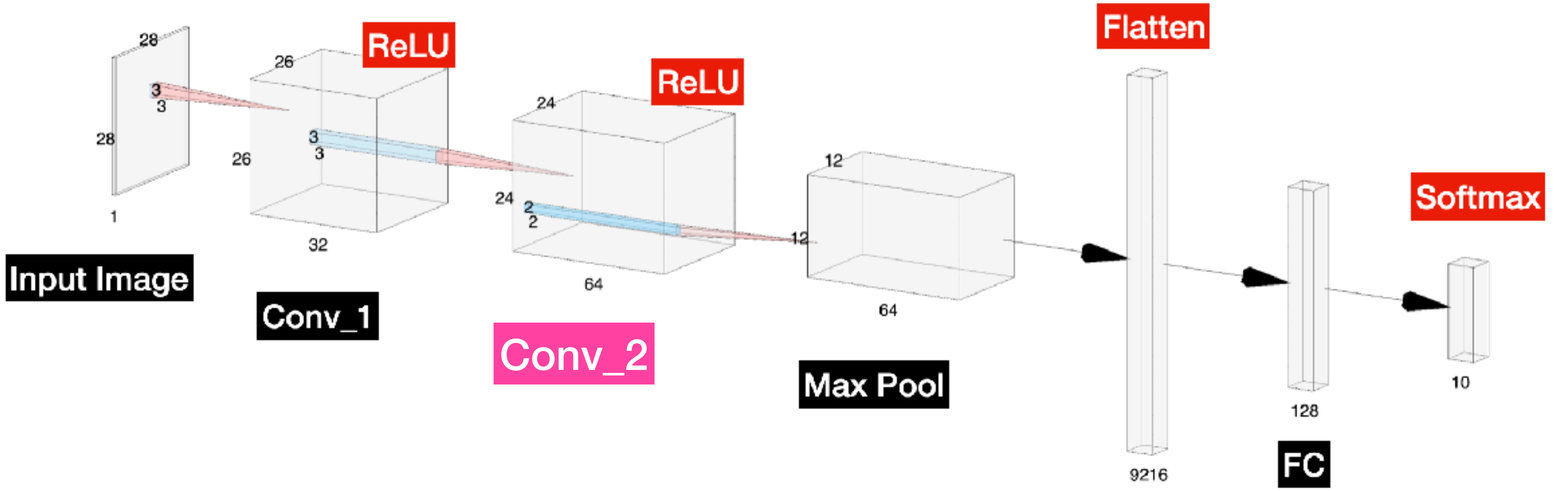
$n = 28$

$f = 3$

$s = 1$

$p = 0$

Calculating the Output Size of Conv_2



Notes:

- We choose 64 Filters or Kernels for Conv_2
- The Feature Maps are shown as Conv_1 & Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2

$$(n \times n) * (f \times f) = (\frac{n + 2p - f}{s} + 1) \times (\frac{n + 2p - f}{s} + 1) = (\frac{26 + (2 \times 0) - 3}{1} + 1) \times (\frac{26 + (2 \times 0) - 3}{1} + 1) = \text{24 x 24}$$

Where

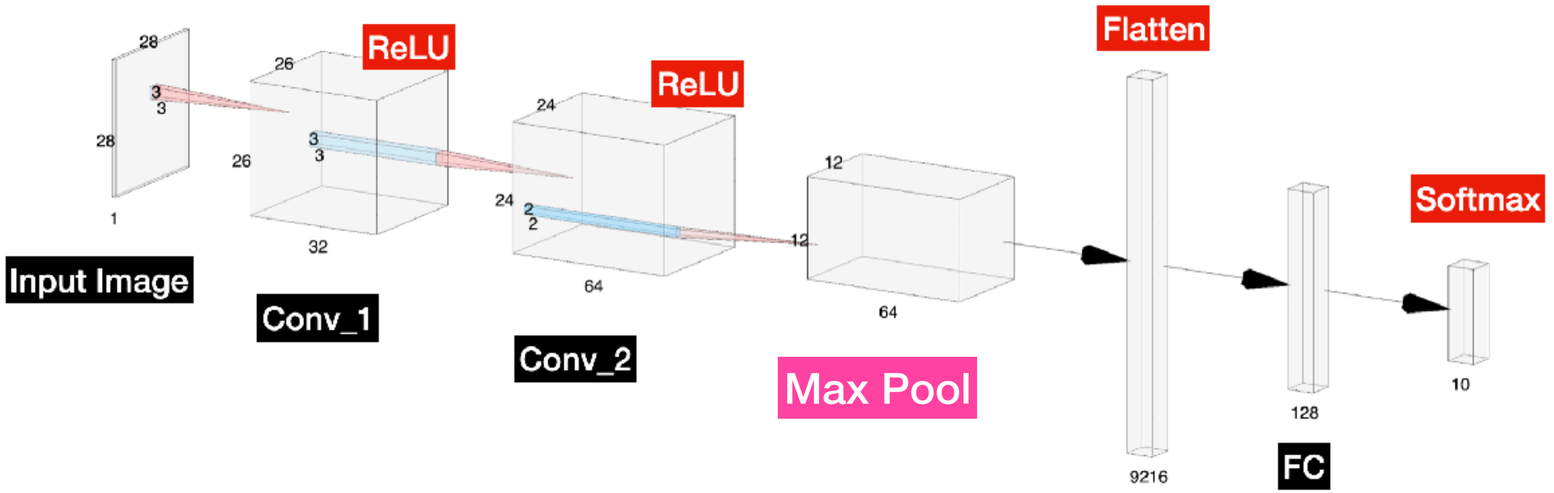
$n = 26$

$f = 3$

$s = 1$

$p = 0$

Calculating the Output Size of the Max Pool Layer



Notes:

- We choose 64 Filters or Kernels for Conv_2
- The Feature Maps are shown as Conv_1 & Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2

$$(n \times n) * (f \times f) = (\frac{n + 2p - f}{s} + 1) \times (\frac{n + 2p - f}{s} + 1) = (\frac{24 + (2 \times 0) - 2}{2} + 1) \times (\frac{24 + (2 \times 0) - 2}{2} + 1) = 12 \times 12$$

12 x 12

Where

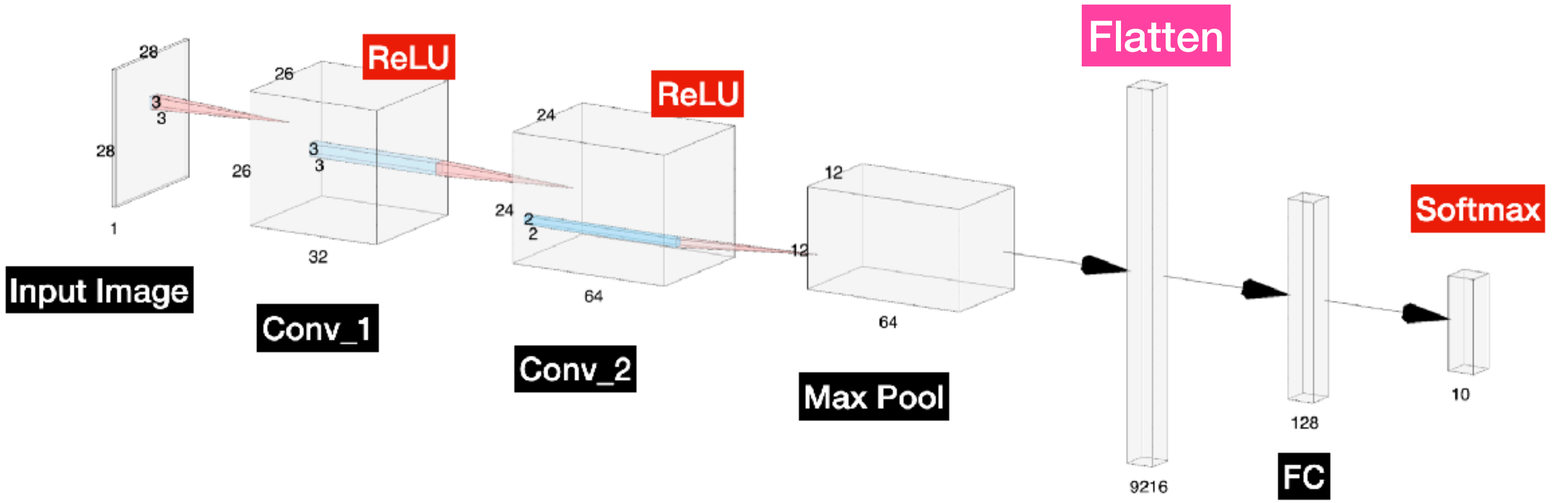
$n = 24$

$f = 2$

$s = 2$

$p = 0$

Calculating the Output Size of Flattened Layer



Notes:

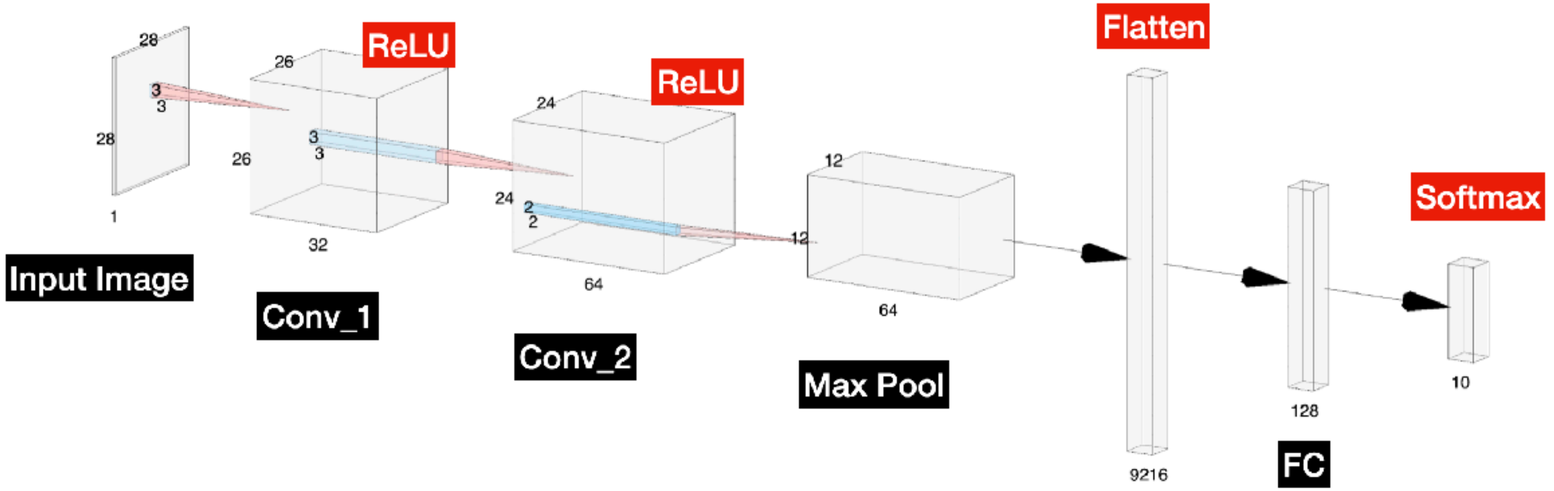
- We choose 64 Filters or Kernels for Conv_2
- The Feature Maps are shown as Conv_1 & Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2

$$12 \times 12 \times 64 = 9216$$

Where

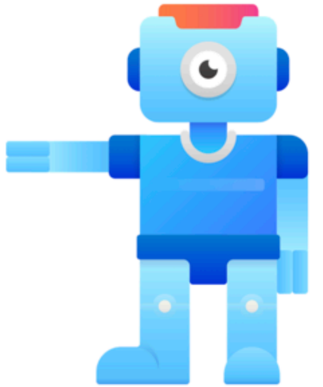
$$n = 12$$

The Rest of the Our CNN



Notes:

- We choose 64 Filters or Kernels for Conv_2
- The Feature Maps are shown as Conv_1 & Conv_2
- Stride is 1
- Padding is 0 (not used)
- Max Pool Stride is 2
- **We choose 128 nodes in our FC Layer**
- **Our Dataset has 10 classes, hence why the final layer has 10 nodes**

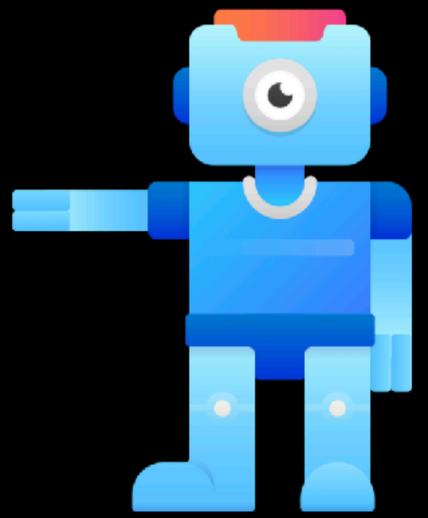


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Parameter Counts in CNNs



MODERN COMPUTER VISION

BY RAJEEV RATAN

Parameter Counts in CNNs

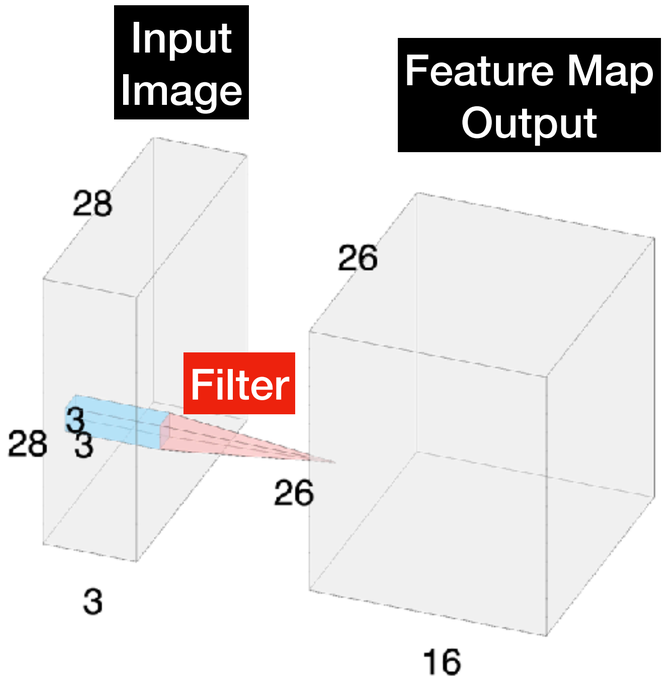
We explore what makes Convolution Neural Networks well suited for images

Parameters

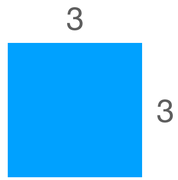
What are Parameters or weights?

- They are the variables that need to be learnt when training a Model
- Often called learnable parameters or weights
- Our hidden layers like the Convolution or Fully Connected Layers have weights

Calculating Learnable Parameters in a Conv Filter



- How many parameters are in 16 Convolution Filters with 3x3 kernels?
- Does the input image dimension matter? No, only it's depth
- All that matters is that we have 16 Conv Filters of 3x3 kernel size



Parameters for One Filter

$$(Height \times Width \times Depth) + bias$$

$$(3 \times 3 \times 1) + 1 = 10$$

Parameters for 16 Filters

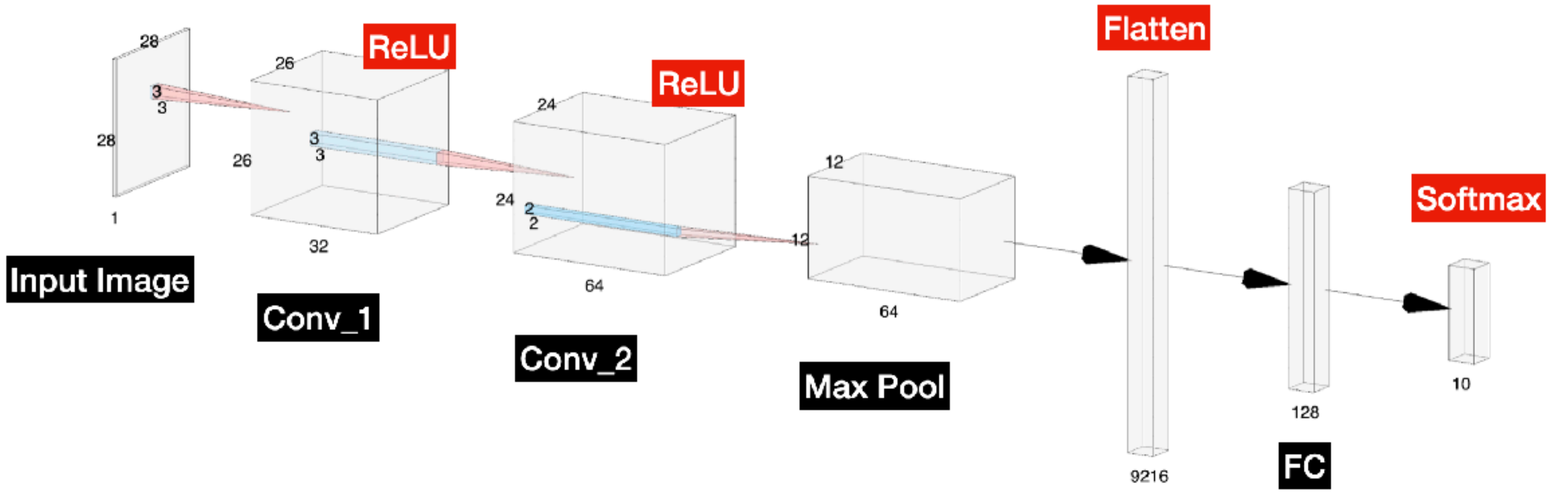
$$((Height \times Width \times Depth) + bias) \times 16$$

$$((3 \times 3 \times 1) + 1) \times 16 = 160$$

Biases

- Biases allow us to shift our activation function (left or right) by adding a constant value
- Biases are per '**neuron**', so per **filter** in our case (shared in the case with colour RGB images as the input)
- These shared biases per 'neuron' allow the filter to detect the same feature
- They are a learnable parameter

Let's Calculate the Number of Parameters in Our Simple CNN



Layer	Parameters
Conv_1 + ReLU	**320**
Conv_2 + ReLU	**18494**
Max Pool	**0**
Flatten	**0**
FC_1	**1,179,776**
FC_2 (Output)	**1,290**
Total	**1,199,882**

Conv_1

$$((Height \times Width \times Depth) + bias) \times N_f$$

$$((3 \times 3 \times 1) + 1) \times 32 = 320$$

Conv_2

$$((Height \times Width \times Depth) + bias) \times N_f$$

$$((3 \times 3 \times 32) + 1) \times 64 = 18{,}494$$

No Trainable Parameters

- Max Pool
- Flatten
- ReLU

Fully Connected/Dense

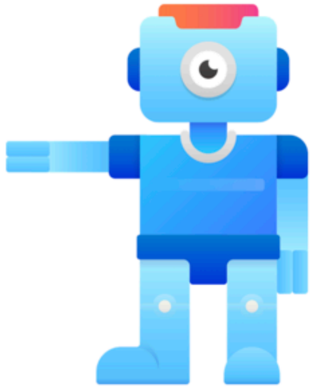
$$(Length + bias) \times N_{nodes}$$

$$(9216 + 1) \times 128 = 1{,}179{,}776$$

Final Output (FC/Dense)

$$(Length + bias) \times N_{nodes}$$

$$(128 + 1) \times 10 = 1{,}290$$

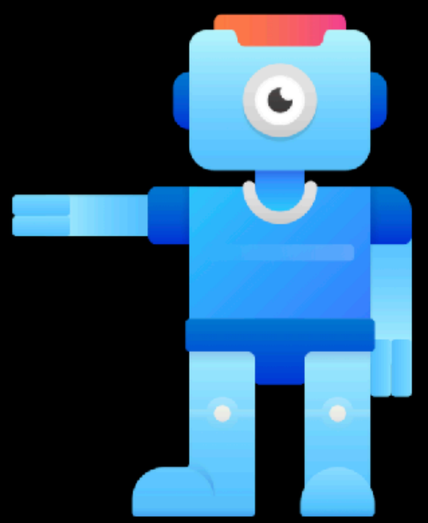


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Why CNNs work so well for Images



MODERN COMPUTER VISION

BY RAJEEV RATAN

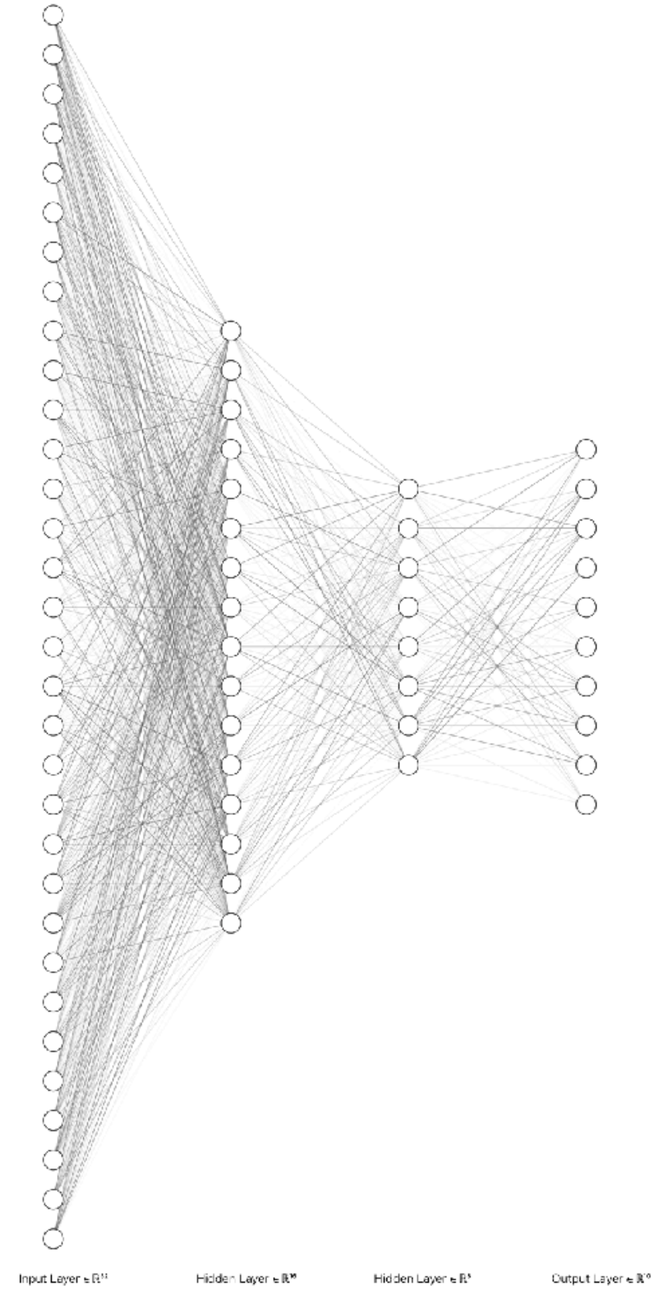
Why CNNs Work So Well For Images

We explore design choices in CNNs and how it relates to their performance in the real world

Standard Neural Networks

A thought experiment…

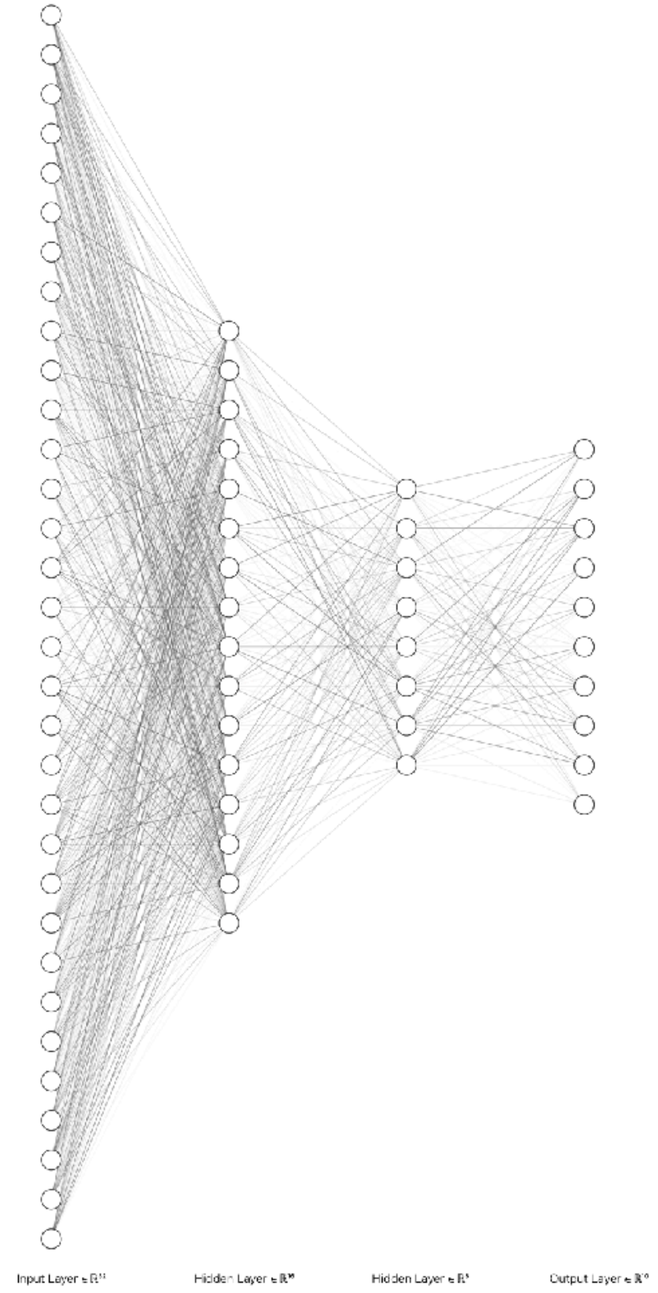
- Standard Neural Networks don’t have Convolution Filter Inputs
- For Images every pixel will be it’s own input
- Therefore, a small image that’s 28 x 28 would have 784 input nodes for our first layer



Standard Neural Networks

A thought experiment…

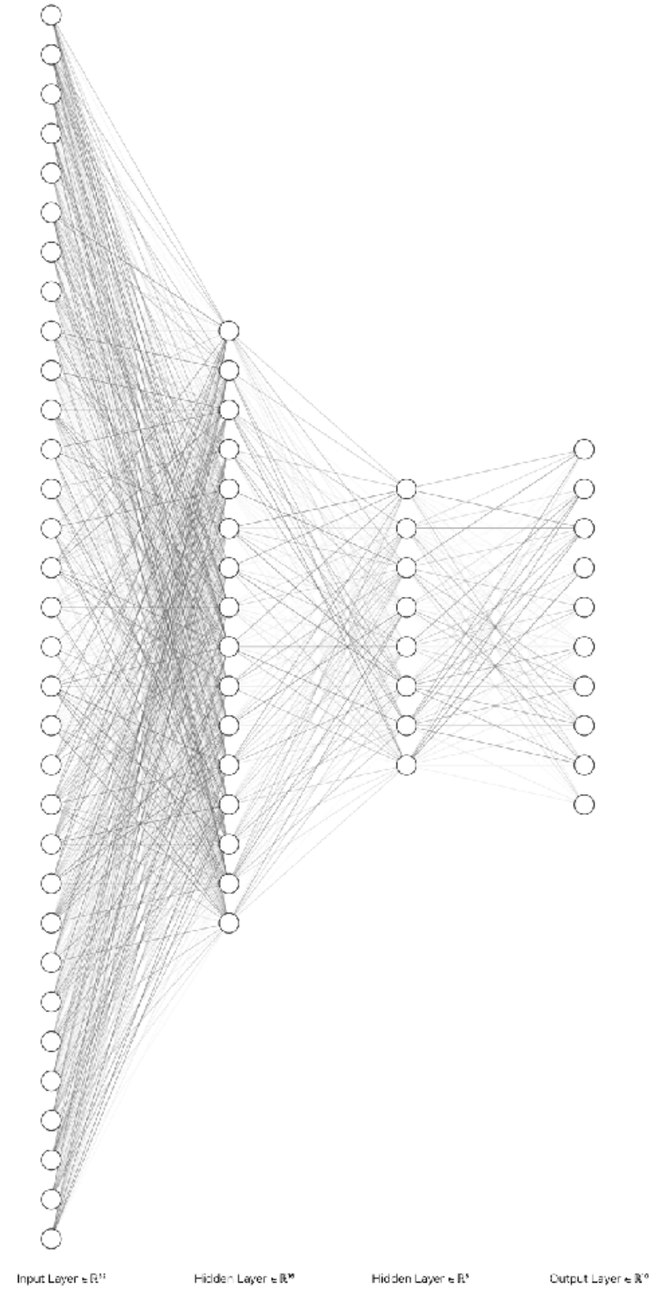
- Our second layer, if we breakdown our previous CNN model would be:
  - 32 Filters for 26 x 26 Feature Maps
  - 13,312
- If they were fully connected, our weight matrix would be:
  - 784 x 13,312 = **10,436,608**
- For just one hidden layer!



Standard Neural Networks

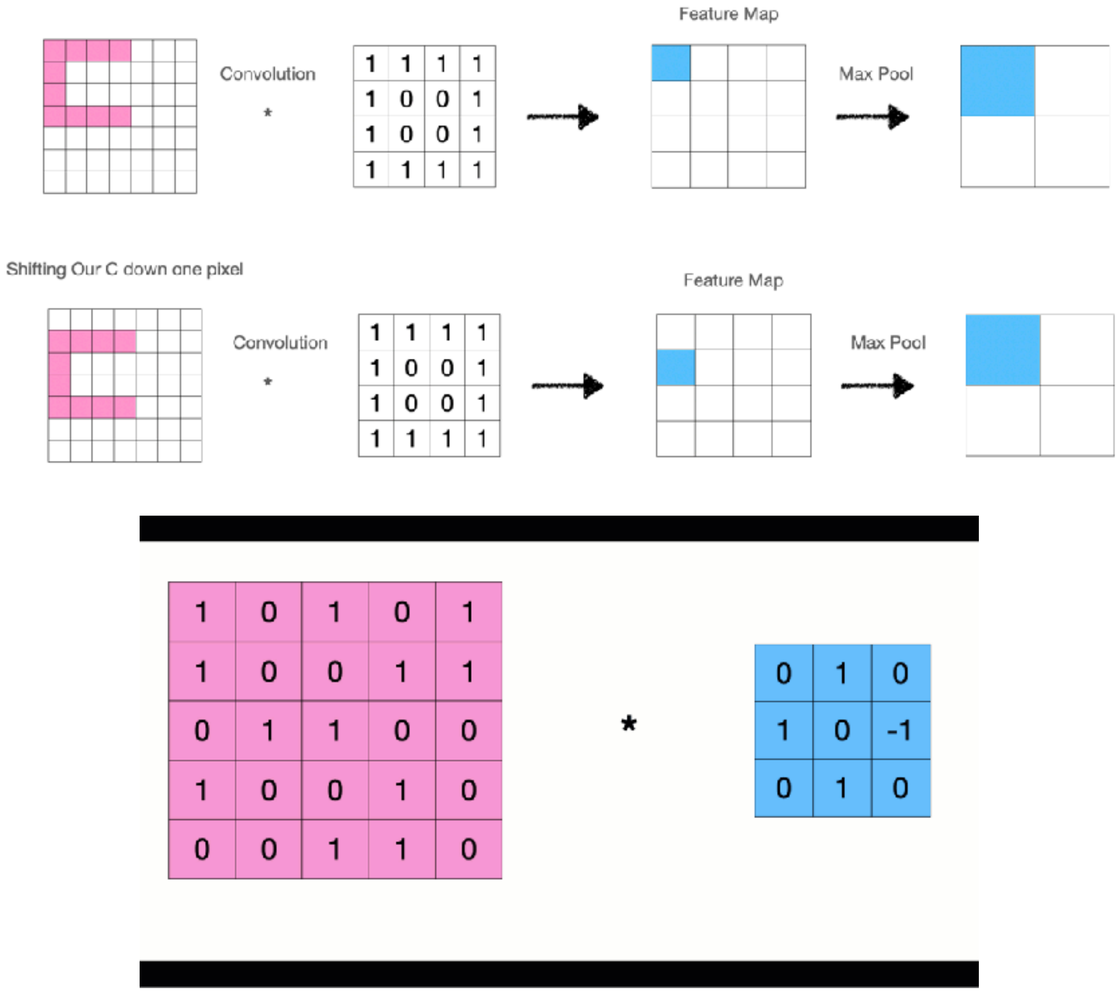
Not Feasible for Image Classification!

- Not scalable to large data inputs
- Overfitting



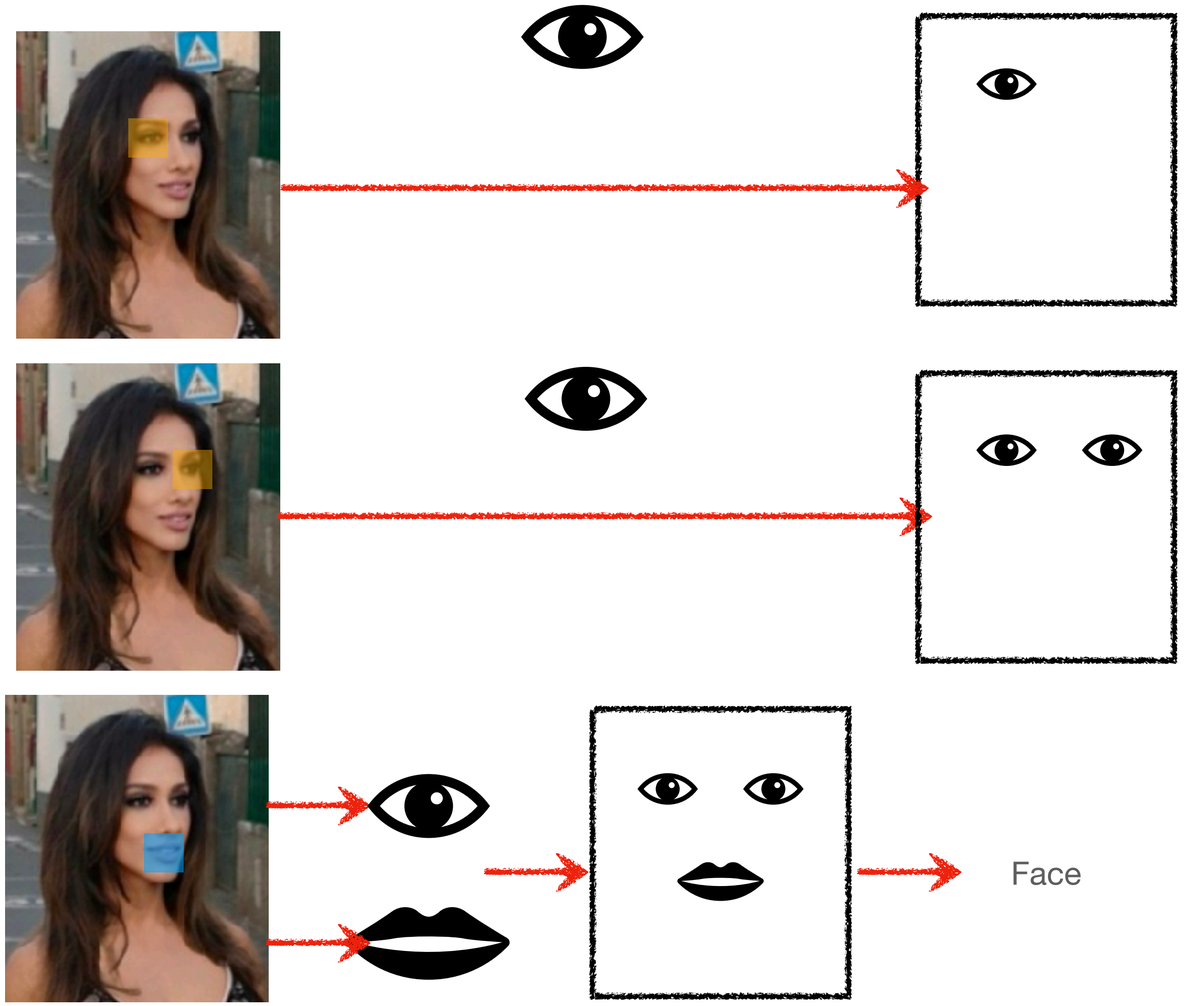
Advantages of Convolution Neural Networks

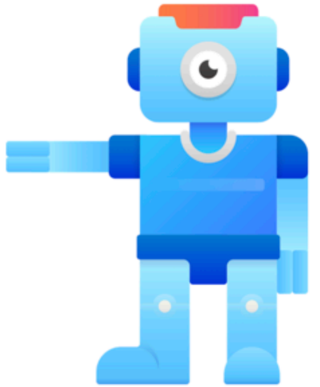
- **Parameter sharing** - where a single filter can be used all parts of an image
- **Sparsity of connections** - As we saw, fully connected layers in a typical Neural Network result in a weight matrix with large number of parameters.
- **Invariance** - Remember our Max Pool Example



Convolution Neural Networks Assumptions

- Low-level features are local
- Features are translational invariant
- High-level features are made up of low-level features



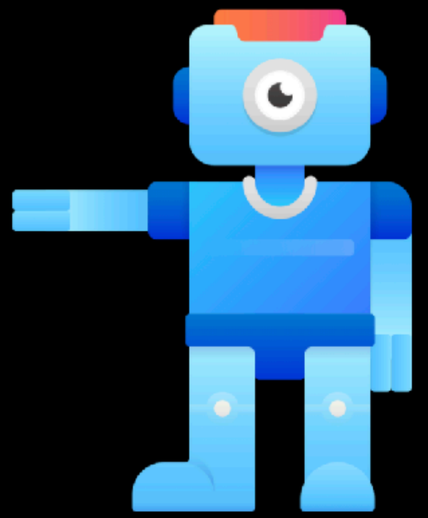


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

How to Train a CNN



MODERN COMPUTER VISION

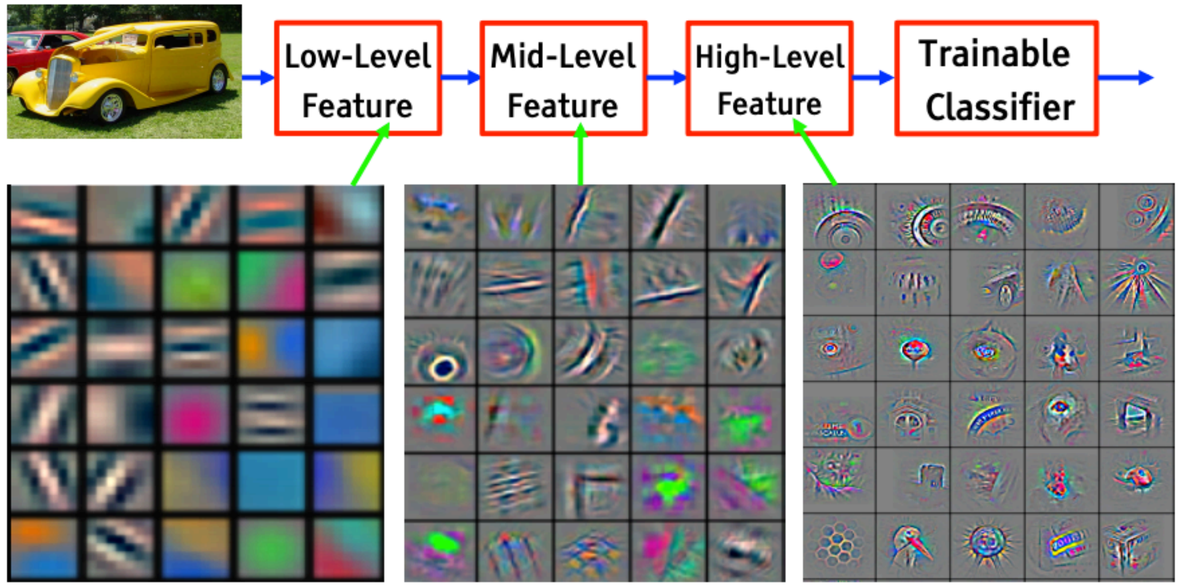
BY RAJEEV RATAN

How to Train a CNN

We now explore a high level view of how the training process works

What Conv Filters Learn

- Typically early layers of our CNN learn **low level** features (like edges, or lines)
- Mid-level layers learn **simple patterns**
- High-level layers learn more **structured complex patterns**
- **How is this done?**

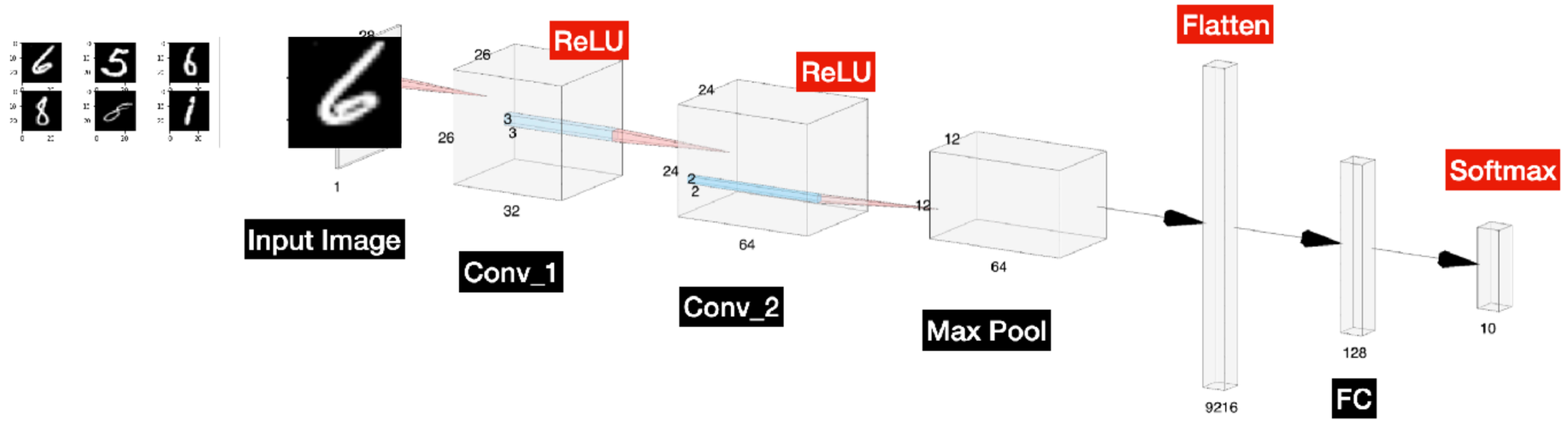


http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf

What Happens During Training?

- **Initialise random weights** values for our trainable parameters
- **Forward propagate** an image or batch of images through our network
- Calculate the **total error**
- Use **Back Propagation** to update our gradients (weights) via Gradient Descent
- **Propagate more images** (or batch) and update weights, until all images have been propagated (one epoch)
- **Repeat a few more epochs** (i.e. passing all image batches through our Network) until our loss reaches satisfactory values

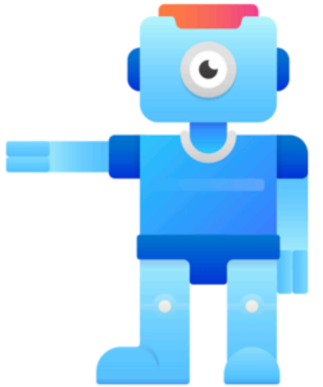
The Training Process



P	6	5	6	8	8	1
0	**0**	0.9	0.83	0.21	0.19	0.62
1	**0.73**	0.8	0.89	0.7	0.92	0.07
2	**0.78**	0.88	0.19	0.39	0.08	0.74
3	**0.37**	0.56	0.07	0.64	0.64	0.9
4	**0.63**	0.25	0.79	0.94	0.52	0.55
5	**0.87**	0.65	0.57	0.63	0.97	0.04
6	**0.67**	0.05	0.45	0.51	0.87	0.51
7	**0.71**	0.66	0.13	0.59	0.86	0.89
8	**0.51**	0.88	0.59	0.01	0.37	0.63
9	**0.24**	0.52	0.79	0.15	0.63	0.78

We need to Learn from our Results

- How correct are our results?
- We need a way tell the model it needs to do better

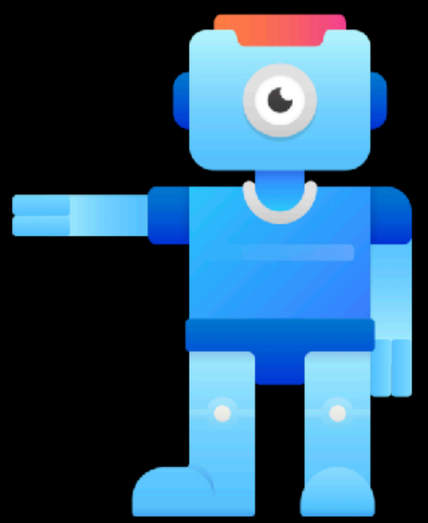


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Loss Functions



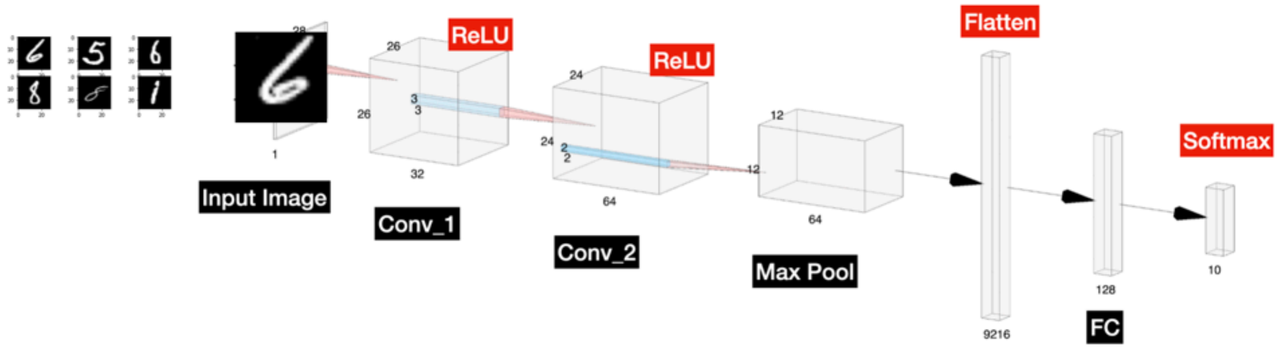
MODERN COMPUTER VISION

BY RAJEEV RATAN

Loss Functions

Loss Functions are essential to training

Quantifying Loss



P	6	5	6	8	8	1
0	0	0.9	0.83	0.21	0.19	0.62
1	0.73	0.8	0.89	0.7	0.92	0.07
2	0.78	0.88	0.19	0.39	0.08	0.74
3	0.37	0.56	0.07	0.64	0.64	0.9
4	0.63	0.25	0.79	0.94	0.52	0.55
5	0.87	0.65	0.57	0.63	0.97	0.04
6	0.67	0.05	0.45	0.51	0.87	0.51
7	0.71	0.66	0.13	0.59	0.86	0.89
8	0.51	0.88	0.59	0.01	0.37	0.63
9	0.24	0.52	0.79	0.15	0.63	0.78

- How bad are the probabilities we predicted?
- How do we quantify the degree our prediction is off by?

Cross Entropy Loss or Categorical Cross Entropy Loss

Class	Predicted Probabilities	Ground Truth
0	0.1	0
1	0.2	0
2	0.1	0
3	0.05	0
4	0.05	0
5	0.05	0
6	0.05	0
7	0.3	1
8	0.05	0
9	0.05	0

- Cross Entropy Loss uses two distributions, our ground truth distribution $p(x)$ and $q(x)$ our predicted distribution.
- $L = -\, y \,.\, log(\hat{y})$
- Where y is the ground truth vector, $\hat{y}$ is the predicted distribution and ‘ . ‘ is the inner product.

Cross Entropy Loss a Simpler Example

Class	Predicted Probabilities	Ground Truth
0	0.3	0
1	0.6	1
2	0.1	0

- $L = -\, y \,.\, log(\hat{y})$
- $L = -\,(0 \times log(0.3) + 1 \times log(0.6) + 0 \times log(0.1)$
- $L = -\,(0 + 1 \times -0.222 + 0) = 0.222$
- **NOTE:**
  - Multi-class log loss rewards/penalises the correct classes only

$$L = \frac{1}{N} \sum_{i-1}^{N} L_i$$

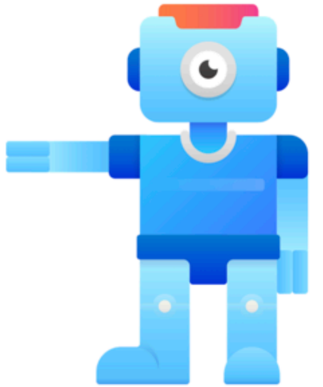
-

Other Loss Functions

- Loss Functions are sometimes called **Cost Functions**
- For Binary Classification problems we use **Binary Cross-Entropy Loss** (same as categorical cross-entropy loss except it uses just one output node)
- For Regressions we often use the Mean Square Error (MSE)
  - Mean Square Error (MSE) = (Target - Predicted)^2
    - $MSE = \frac{1}{n}\sum(Y_i - \hat{Y}_i)^2$
- Other loss functions that are sometimes used:
  - L1, L2
  - Hinge Loss
  - Mean Absolute Error (MAE)

What do we do with our Quantified Loss?

- Updating all the weights of our model is not trivial
- How do we correctly update our weights to minimise loss?
- We use Back Propagation
- And we use the loss value for this!

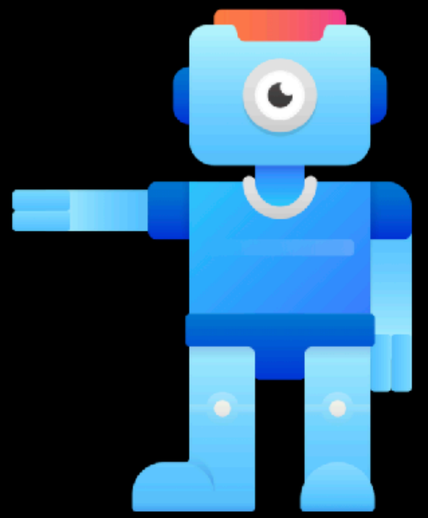


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Back Propagation



MODERN COMPUTER VISION

BY RAJEEV RATAN

Back Propagation

Back Propagation makes Neural Networks Trainable

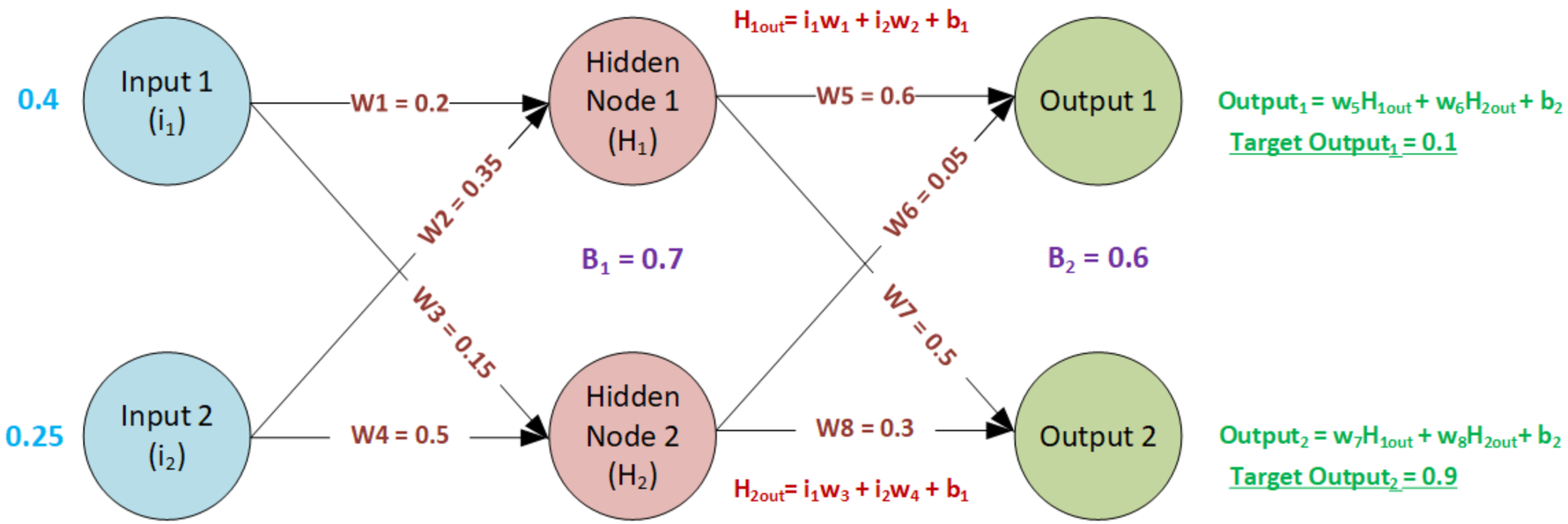
Back Propagation

This is what makes Neural Networks Trainable :)

- The importance of Back Propagation cannot be understated
- Using the loss, it tells us how much to change/update the gradients by so that we reduce the overall loss

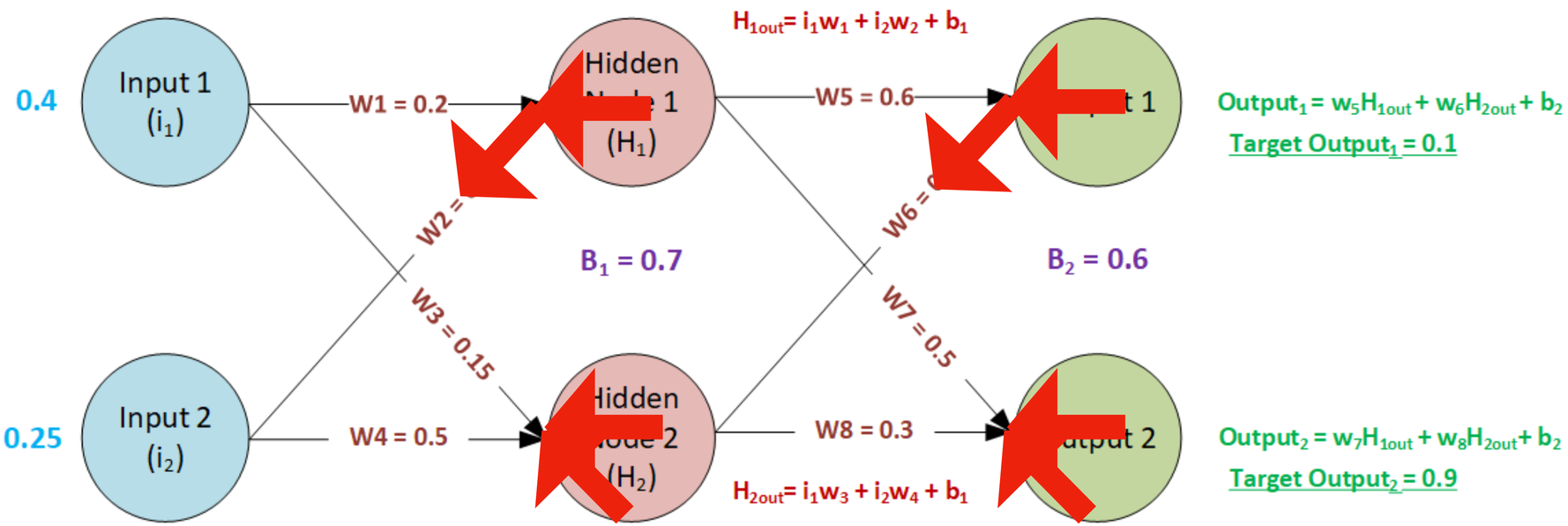
Back Propagation Example

Explained Using Neural Networks



- Let's look at a regular Neural Network above.
- Using the Loss value, Back Propagation can tells us whether a **small increase** of W_5 to 0.6001 or a **small decrease** 0.5999 will lead to a **reduction in the overall loss**

Back Propagation Example



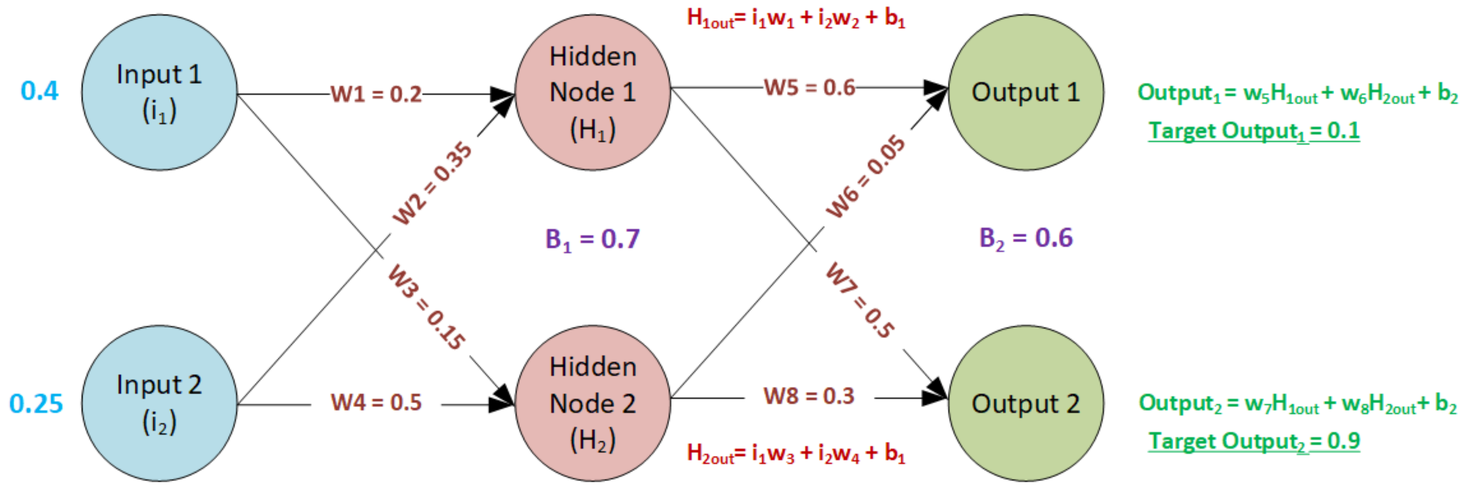
- Moving **right to left**
- Back Propagation gives us the new gradient or weight values for each node so that the overall loss is decreased
- This is done for all nodes

Back Propagation Process

- By **forward** propagating input data we can use back propagation to **lower** the weights to **lower the loss**
- **But**, this simply tunes the weights for that particular input (or batch of inputs)
- We improve **Generalisation** (ability to make good predictions on unseen data) by using all data in our training dataset
- By continuously changing the weights for each data input (or batch of images) we are lowering the overall loss for our training data.

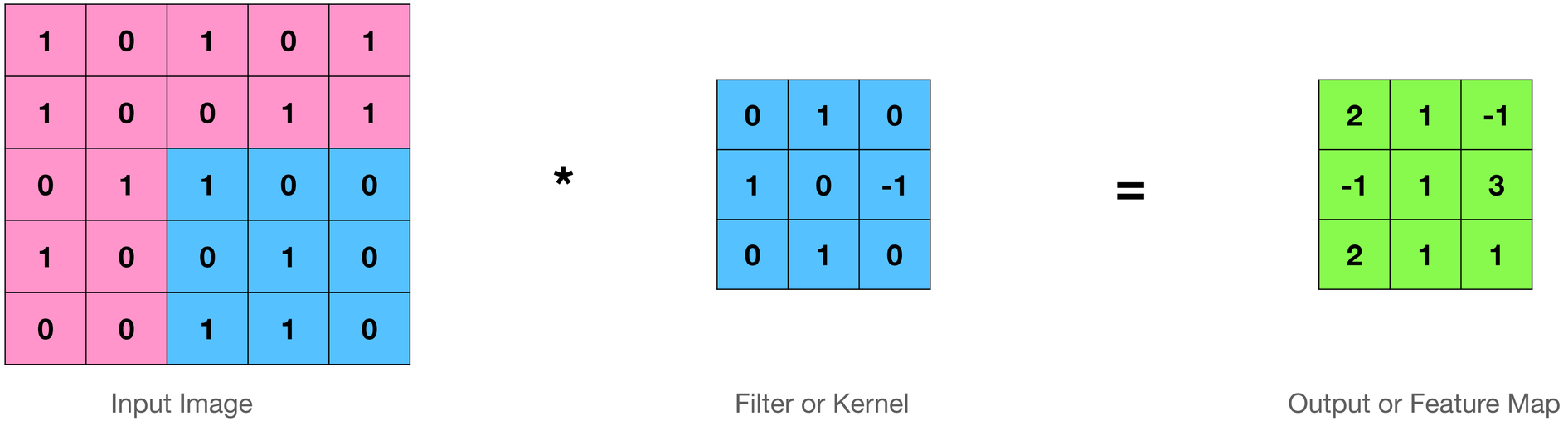
What do our Weights or Gradients Look Like?

Let's look at a Simple Neural Network



- The output from Hidden Node 1 is:
  - $H_1out = i_1w_1 + i_2w_2 + b_1$

For a Convolutional Neural Network

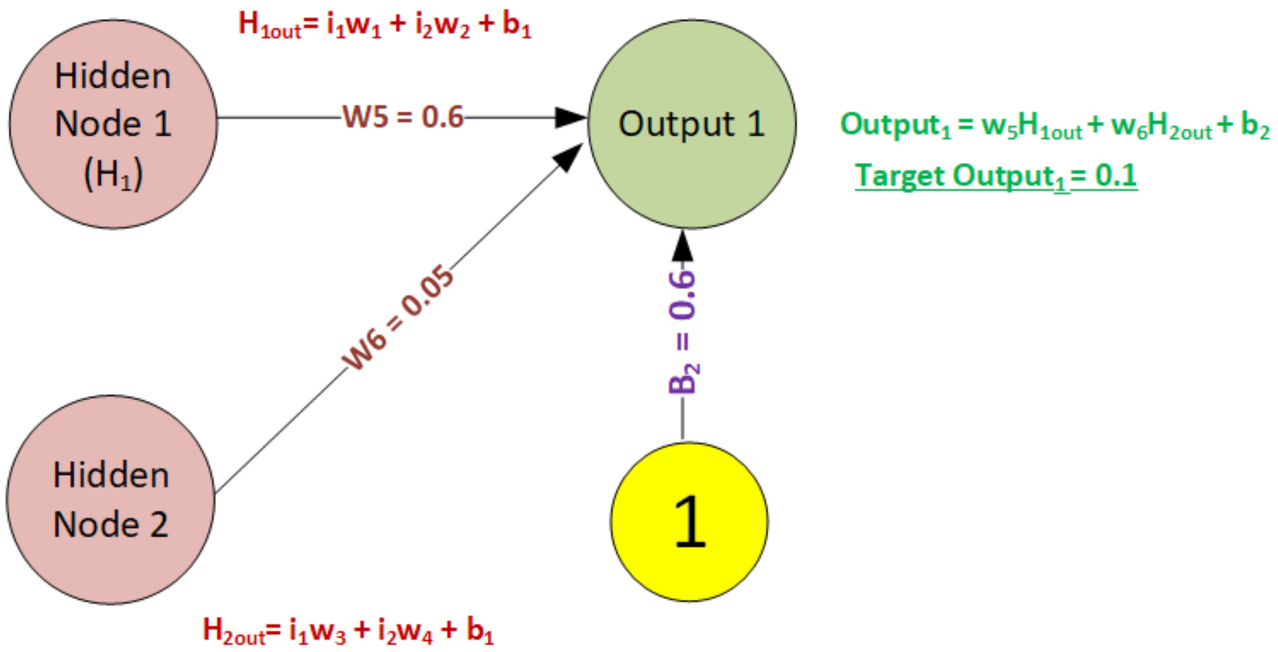


- The values of our Filter/Kernel are the weights!

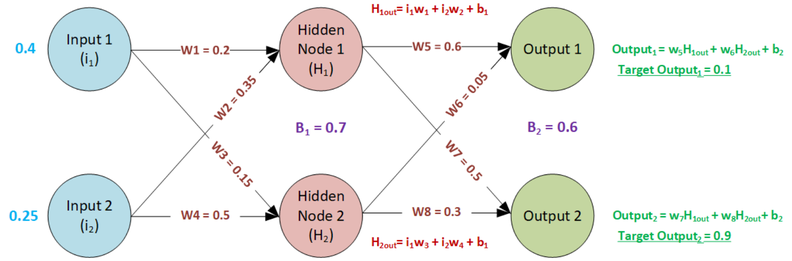
How does Back Propagation Work?

- **Chain Rule!**
- If we have two functions $y = f(u)$ and $u = g(x)$ then the derivative of y is:
  - $$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

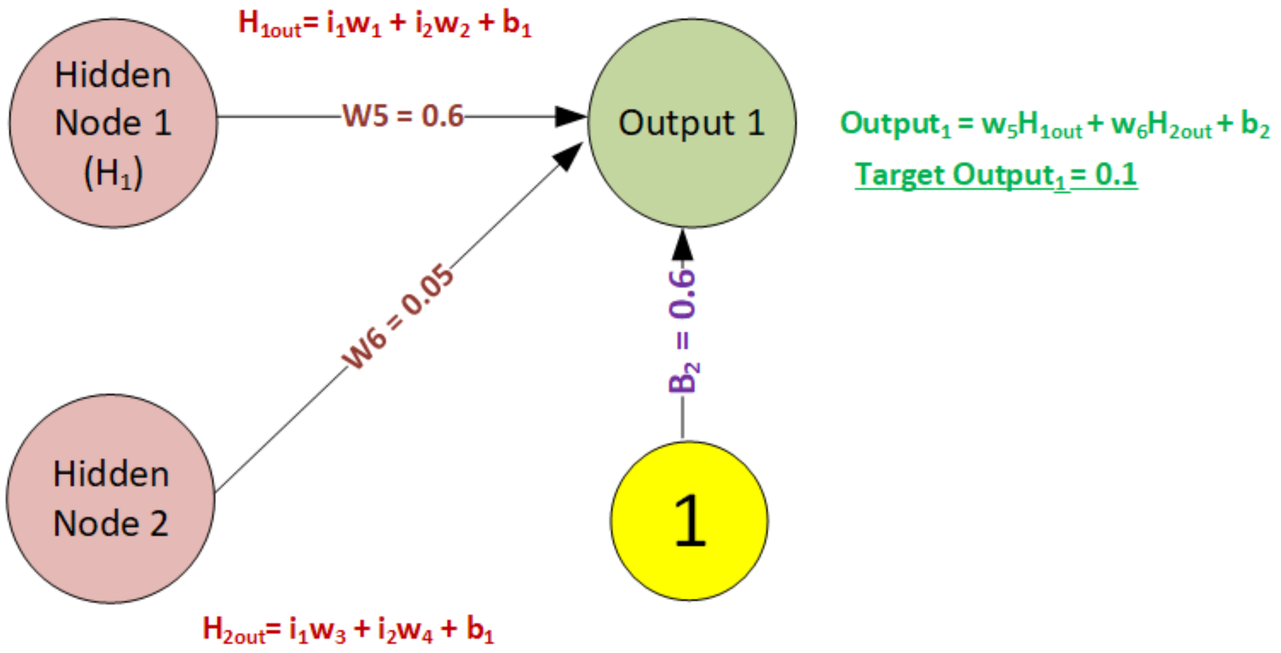
A Simple Back Propagation Example



- We want to know how much changing W_5 changes the **Total Error**.
- That is given by:
  - $\frac{dE_T}{dW_5}$
  - Where E_T is the sum of the error from Outputs 1 and 2 (see below)

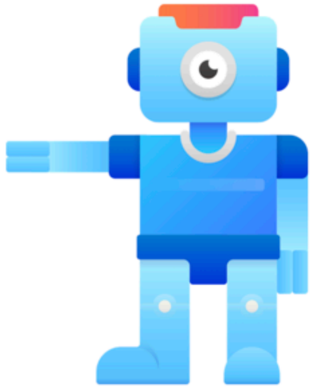


A Simple Back Propagation Example



New $W_5 = -\lambda \times \frac{dE_T}{dW_5}$

- Note we introduced a new parameter λ
- λ is our learning rate
- It controls how a big a jump (positive or negative) we take when updating W_5
- Large learning rates train faster, but can get stuck in a Global Minimum
- Small learning rates train more slowly

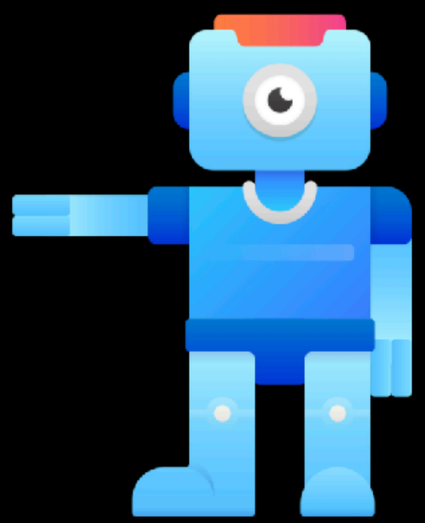


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Gradient Descent



MODERN COMPUTER VISION

BY RAJEEV RATAN

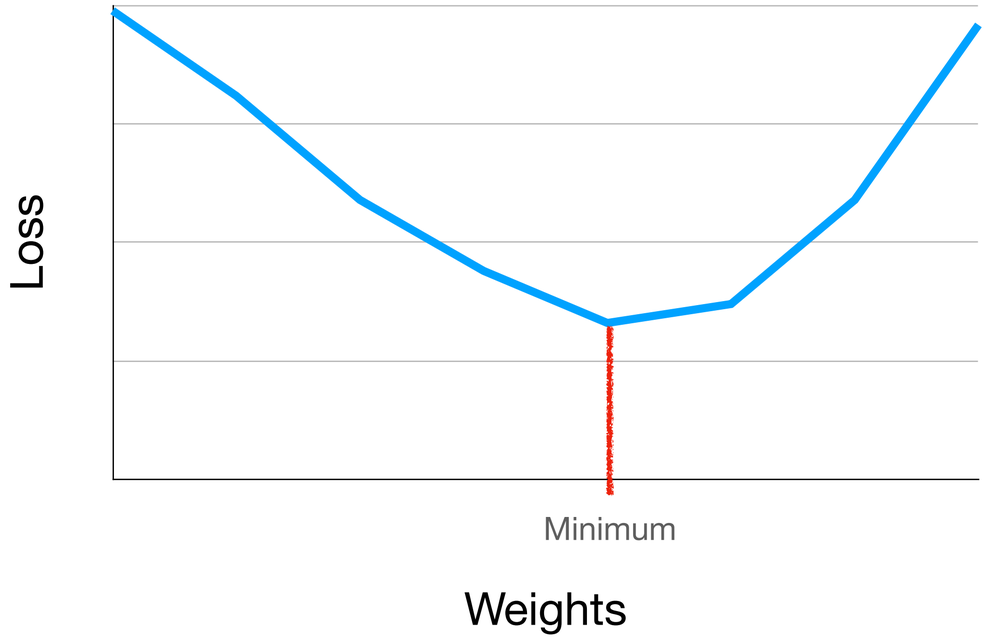
Gradient Descent

Finding the optimal weights

Loss Functions

How do we find the lowest loss?

- Back Propagation is the process we use to update the individual weights or gradients
  - $wx + b$
- Our goal is finding the right value of weights where the loss is lowest
- The method by which we achieve this goal (i.e. updating all weights to lower the total loss) is called **Gradient Descent**
- It's the point at which we find the **optimal weights** such that **loss is near lowest**



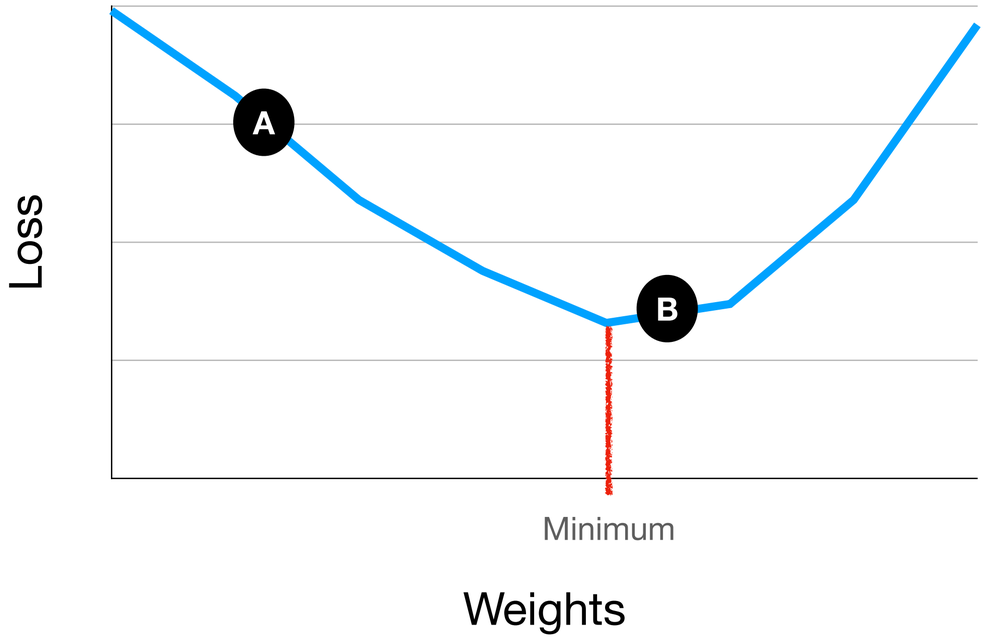
Gradients

Gradients are the derivative of a function

- It tells us the rate of change of one variable with respect to the other e.g.

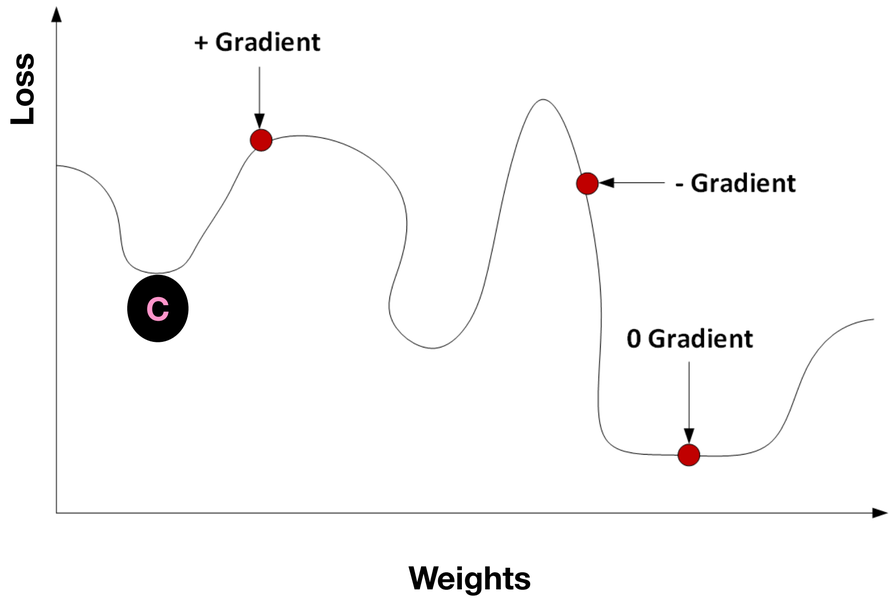
  $Gradient = \frac{dE}{dw}$, where E is the Error or Loss and w is the weight

- A positive gradient means, loss increases if weights increase
- A negative gradient means, loss decreases if weights increase
- At point A, moving right increases our weights and decreases our loss, -ve
- A point B, moving right increases our weights and increases our loss, +ve
- Therefore, the **negative** of our gradient tells us the direction we should be moving



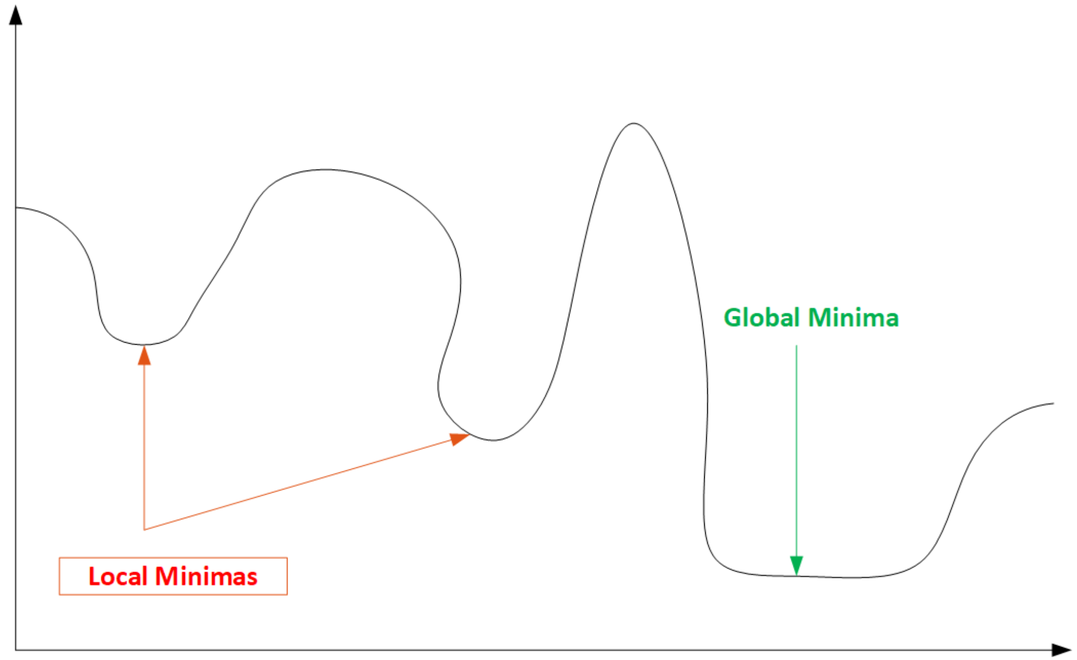
More on Gradients

- The point at which a Gradient is zero means that small changes to the left or right don't change the loss
- In training Neural Networks this is good and bad
- At point, C, very small changes to the left or right don't change the Loss
- This means, our network gets stuck during training.
- This is called getting **stuck in a Local Minima**



Local and Global Minimas

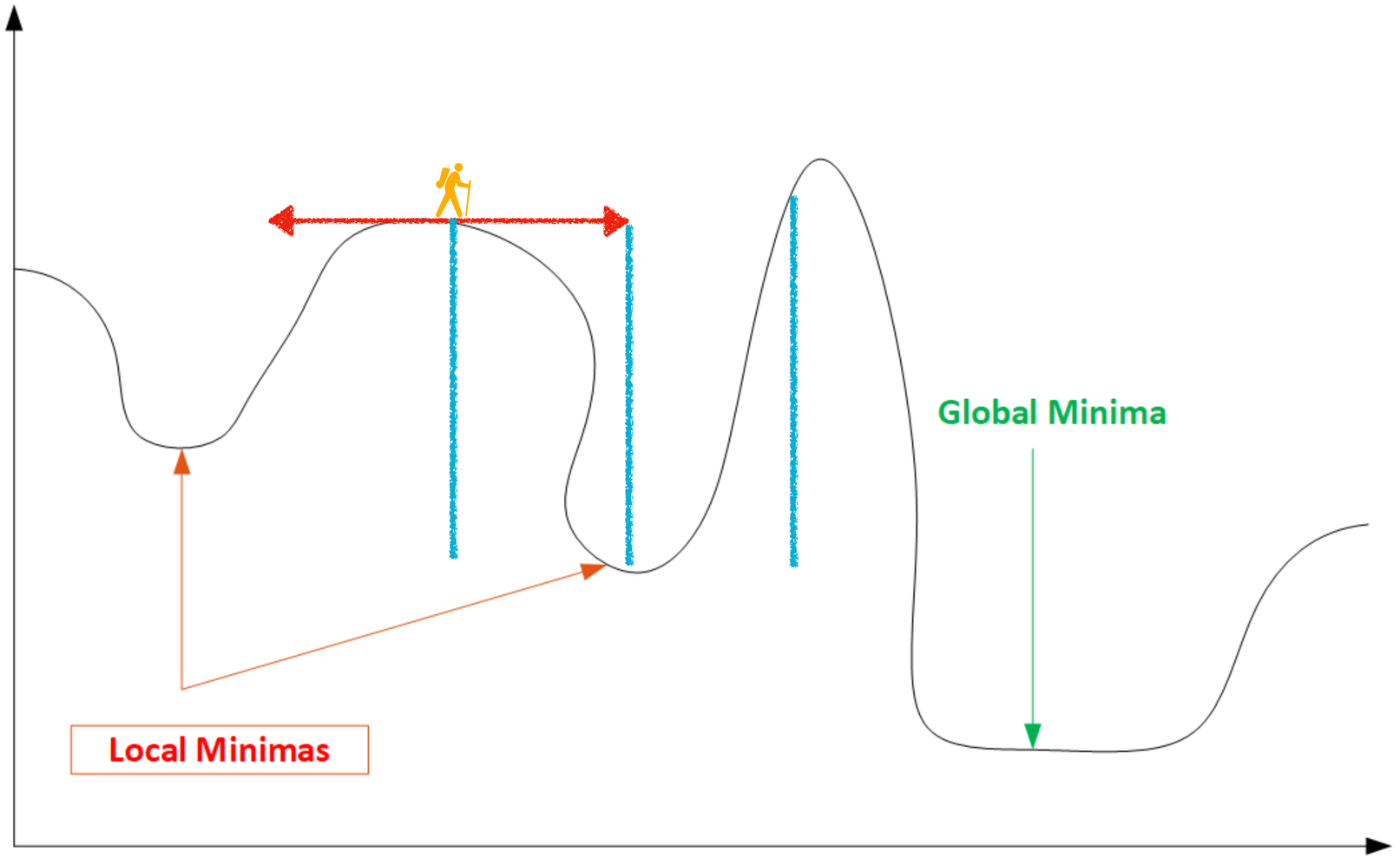
- We always want to find the Global Minima during training.
- That is the point where the combination of weights give the lowest loss



Gradient Descent

- Imagine being a really tiny person and you're traversing down the slope of this old, rough bowl.
- There'd be peaks, valleys, troughs etc.
- How do you know when you're truly at the bottom?
- Possibly take large steps so you don't get stuck in a valley
- But then you risk jumping over the Global Minima

Step Size is Important



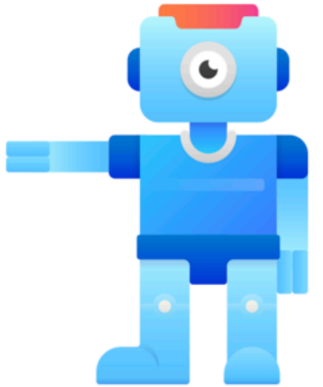
Learning Rates

Recall our Back Propagation Weight Update Formula

- $W_5 = -\lambda \times \frac{dE_T}{dW_5}$
- λ is our learning rate
- Learning Rates allow us to adjust the magnitude of jumps in weights
- Finding an optimal value will avoid us finding Local Minimums and while preventing us from jumping over the Global Minimum.

Gradient Descent Methods

- **Naive Gradient Descent** - Passes the entire dataset through our network then updates the weights.
  - It is computationally expensive and slow.
- **Stochastic Gradient Descent (SGD)** - Updates weights after each data sample (image) is forward propagated through our network.
  - This leads to noisy fluctuating loss values and is also slow to train
- **Mini-Batch Gradient Descent** - Combines both methods, it takes a batch of data points (images) and forward propagates all, then updates the gradients.
  - This leads to faster training and convergence to the Global Minima
  - Batches are typically 8 to 256 in size

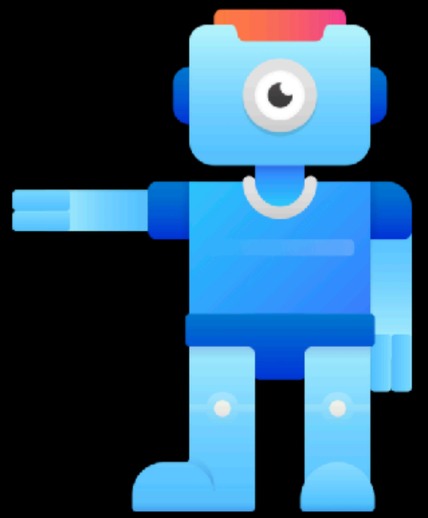


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

Optimisers



MODERN COMPUTER VISION

BY RAJEEV RATAN

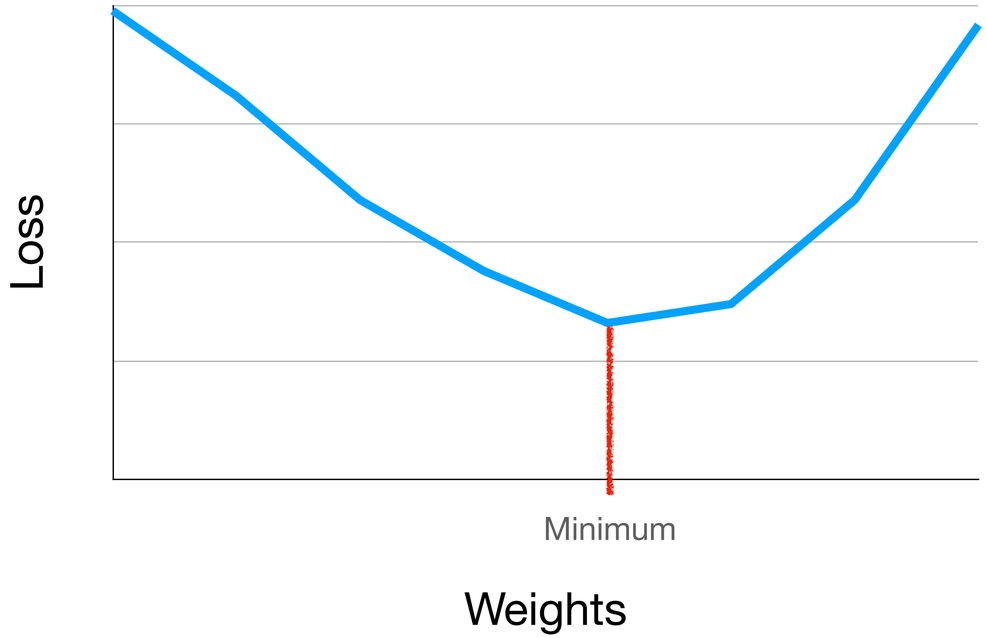
Optimisers & Learning Rate Schedules

Methods or Algorithms used in finding optimal weights

Optimisers

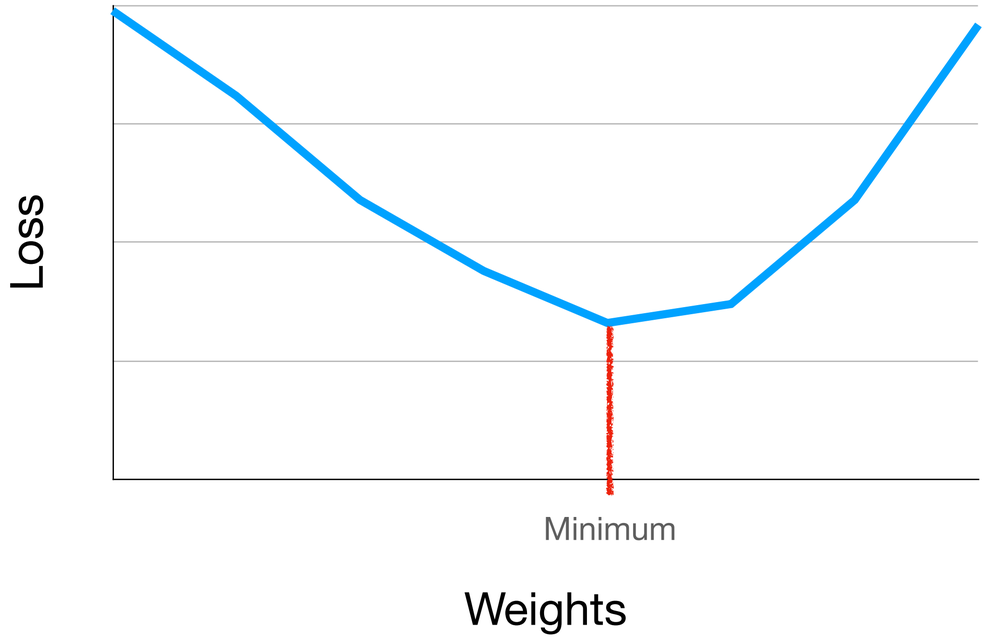
The algorithm we use to update the weights

- You have already been introduced to Gradient Descent which is an example of a first order optimisation algorithm.
- In this section we will explore some alternatives to Stochastic Gradient Descent and take a look at a few other optimisation algorithms.



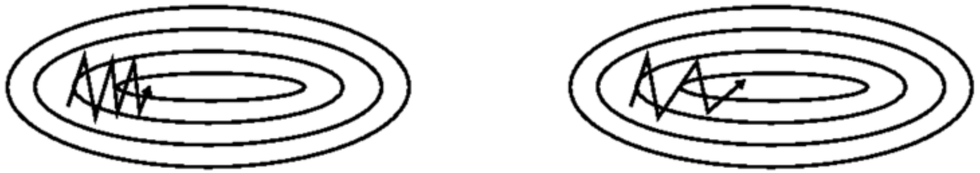
Problems with standard SGD

- Choosing an appropriate Learning Rate (LR), deciding Learning Rate Schedules,
- Using the same learning rate for all parameter updates (as is the case with sparse data), but most importantly SGD is susceptible to getting trapped in Local Minimas or Saddle Points (where one dimensions slopes up and another slopes down).
- To solve some of these issues several other algorithms have been developed including some extensions to SGD which include Momentum and Nestor's Acceleration.



Momentum

- One of the issues with SGD are areas where our hyper-plane is much steeper in one direction.
- This results in SGD oscillating around these slopes making little progress to the minimum point.
- Momentum increases the strength of the updates for dimensions whose gradients switch directions. It also dampens oscillations. Typically we use a Momentum value of 0.9.



(a) SGD without momentum (b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

https://arxiv.org/pdf/1609.04747.pdf

Nesterov's Acceleration

- One problem introduced by Momentum is overshooting the local minimum.
- Nestoerov's Acceleration is effectively a corrective update to the momentum which lets us obtain an approximate idea of where our parameters will be after the update.
- Below we show the corrected Gradient updates (in green)

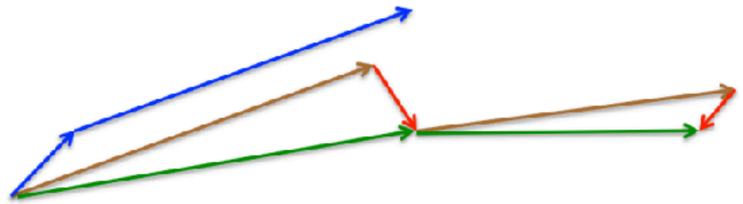
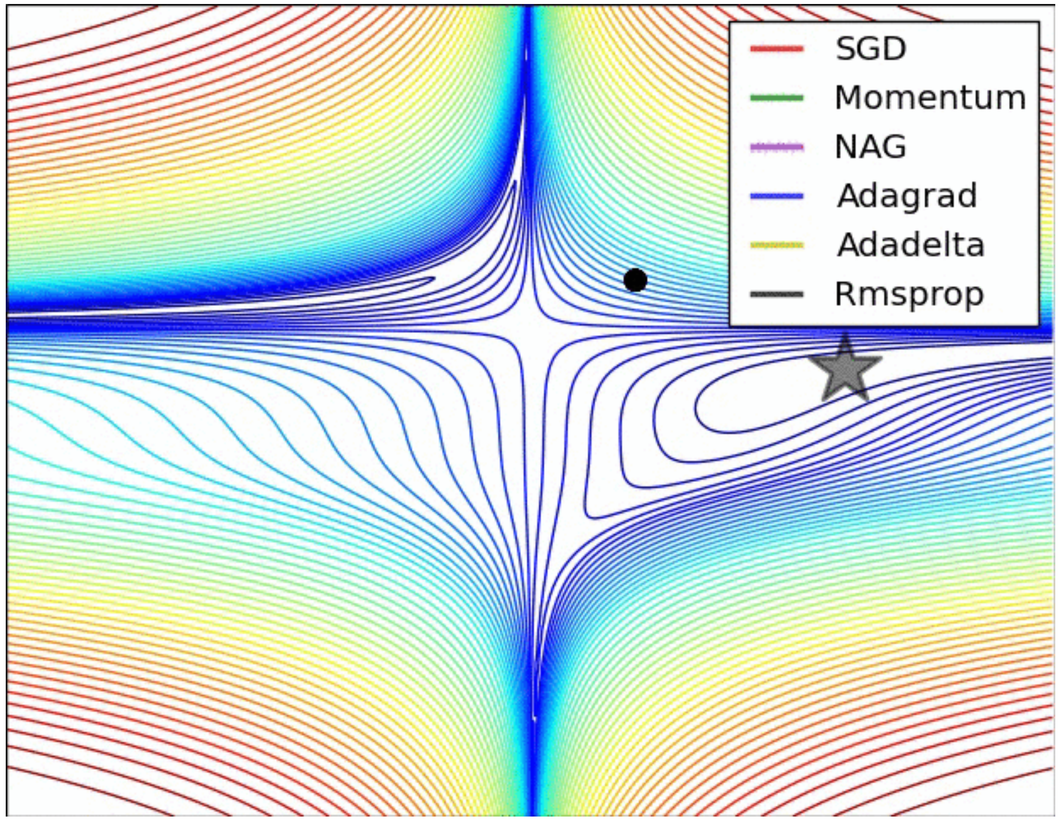


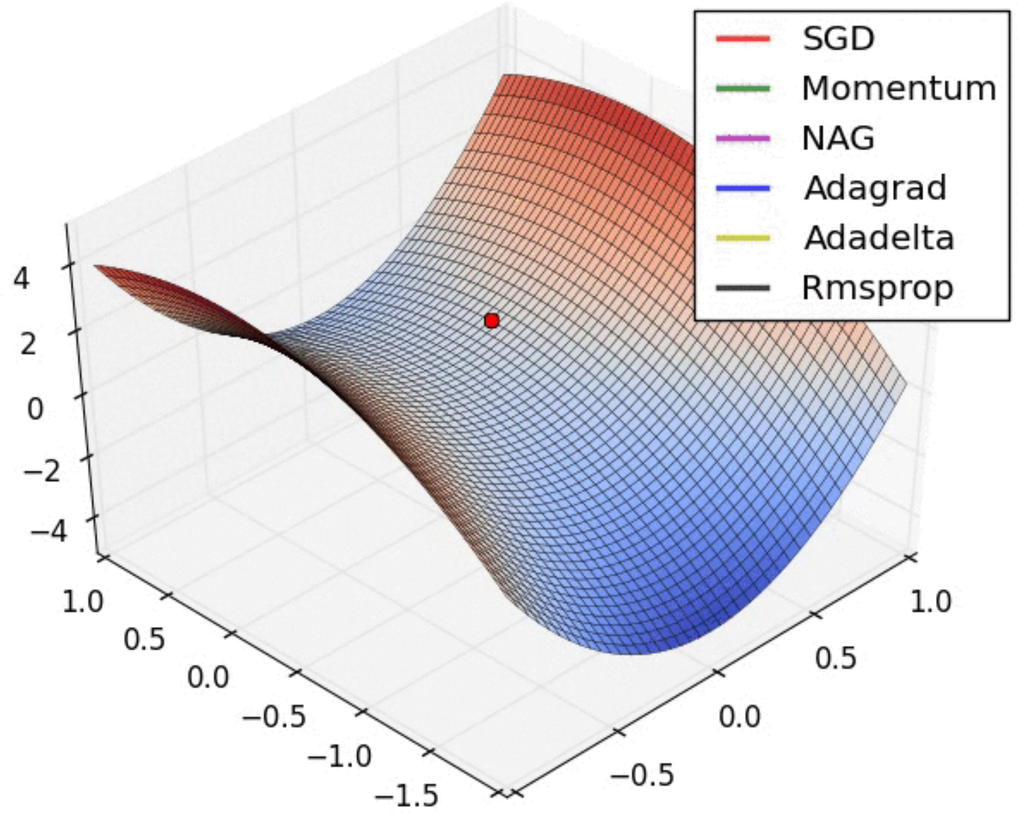
Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

Other Optimisers

- **Adagrad** - Good for Sparse data. It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.
- **Adadelta** - Extension of Adagrad that reduces its aggressive, monotonically decreasing learning rate.
- **Adam** - Adaptive Moment Estimation is a method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients, similar to momentum.
- **RMSProp** - Similar to adadelta,
- **AdaMax** - It is a variant of Adam based on the infinity norm.
- **Nadam** - Nesterov accelerated gradient (NAG) is superior to vanilla momentum. Nadam (Nesterov-accelerated Adaptive Moment Estimation) thus combines Adam and NAG. In order to incorporate NAG into Adam, we need to modify its momentum term
- **AMSGrad** - AMSGrad is an extension to the Adam version of gradient descent that attempts to improve the convergence properties of the algorithm, avoiding large abrupt changes in the learning rate for each input variable

Visual Comparison of some Optimisers

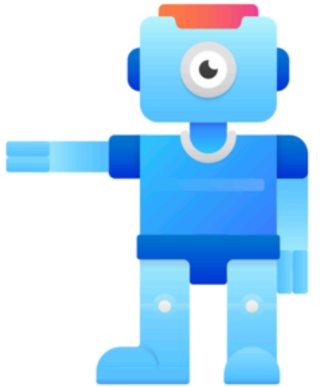




http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/

Learning Rate Schedules

- A preset list of learning rates used for each epoch.
- Progressively reduce over time.
- We use LR Schedules because if our LR is too high, it can overshoot the minimum points (areas of lowest loss).
- Applying a progressively decreasing learning rate allows the network to take smaller steps (when gradients are updated) allowing our network to find the point of lowest loss instead of jumping over it.
- Early in the training process, we can afford to take big steps, however as the decrease in loss slows, it is often better to use smaller learning rates to avoid oscillations.
- Learning rate schedules are simply to implement with our Deep learning libraries (PyTorch or Keras/Tensorflow) as they incorporate a decay parameter in optimisers that support LR schedules, such as SGD.

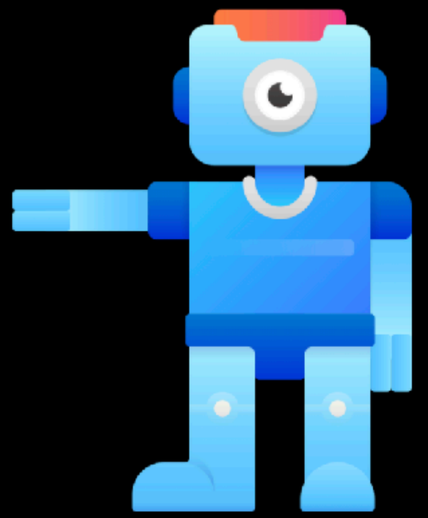


MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

A Review of all CNN Theory



MODERN COMPUTER VISION

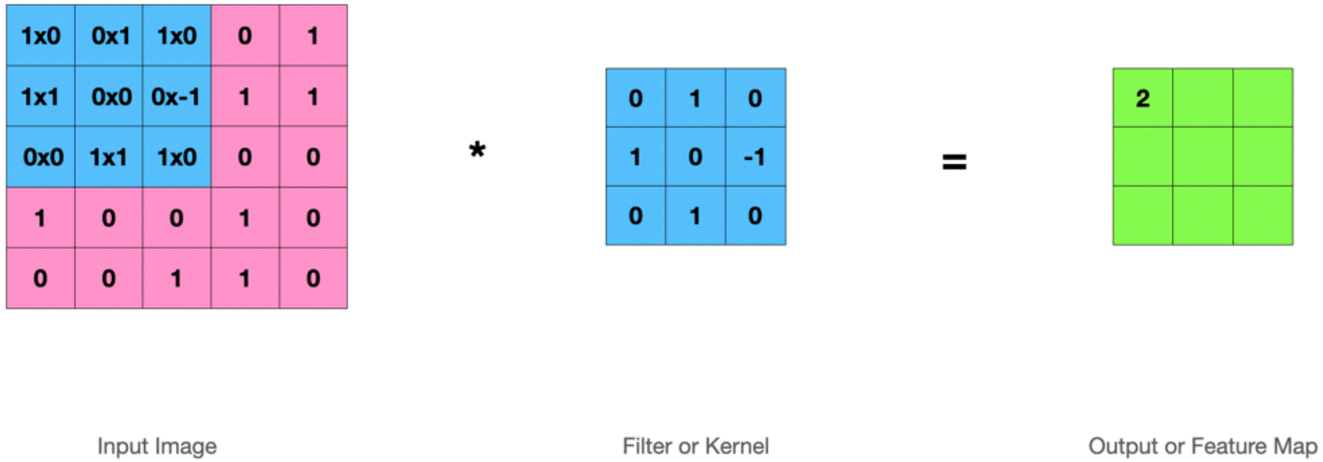
BY RAJEEV RATAN

Summary of Convolutional Neural Networks (CNNs)

Putting it all together

Conv Layers

$$(1 \times 0) + (0 \times 1) + (1 \times 0) + (1 \times 1) + (0 \times 0) + (0 \times -1) + (0 \times 0) + (1 \times 1) + (1 \times 0) = 2$$

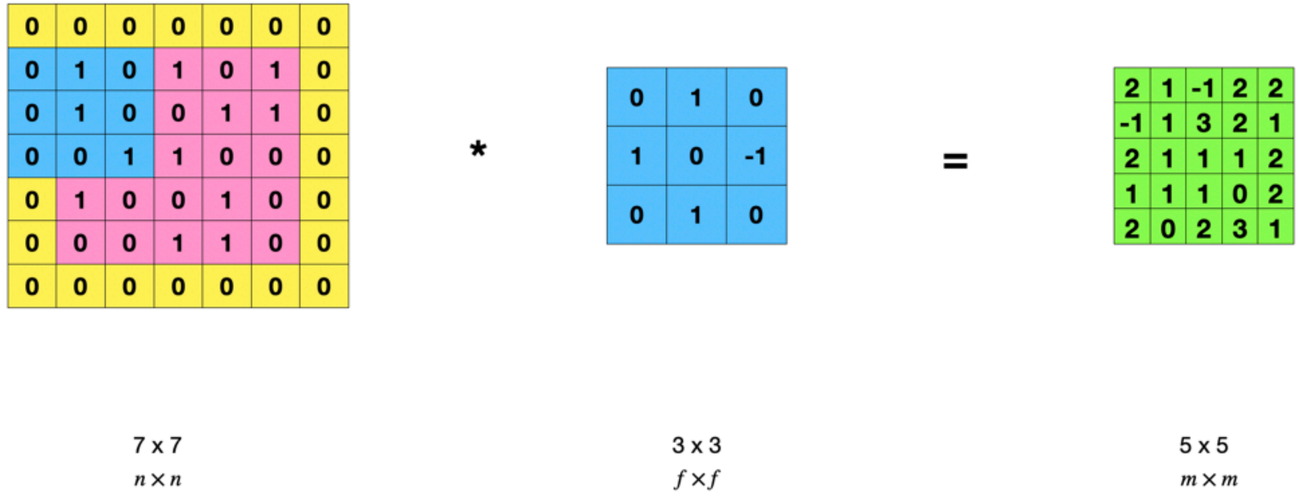


- Convolution Operations occur when we Convolve our Filters with the input image by sliding it over our image
- This produces an output called a Feature Map
- Feature Maps are now the inputs to the next layer of our CNN

Stride, Padding and Kernel Size

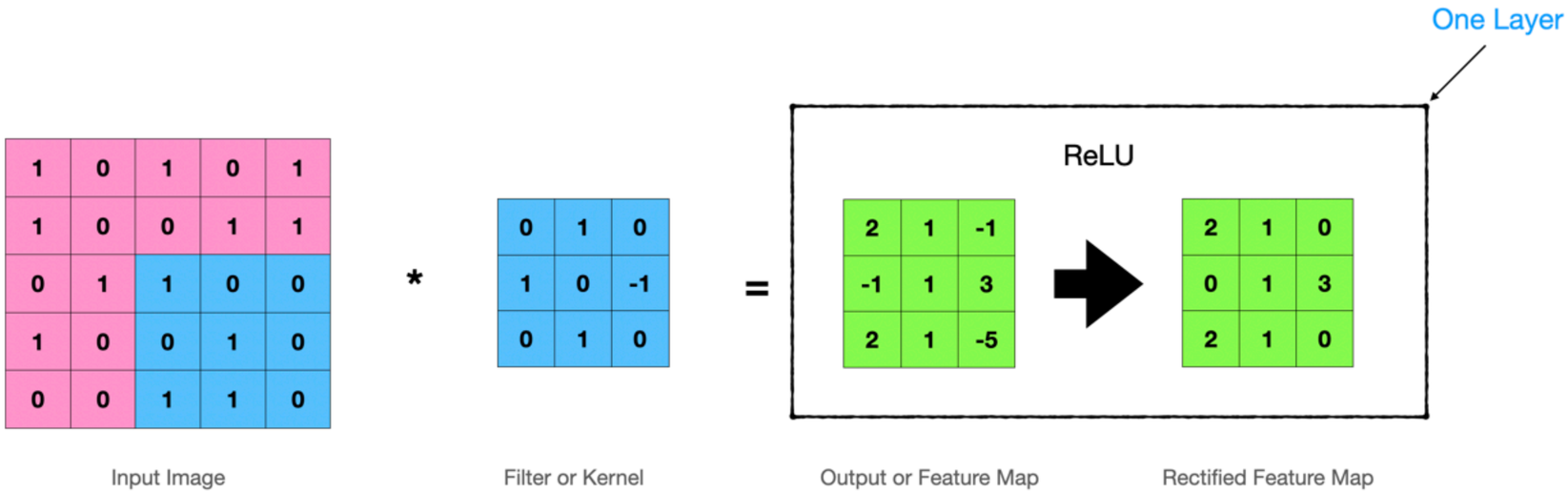
Feature Map Size = $n - f + 1 = m$

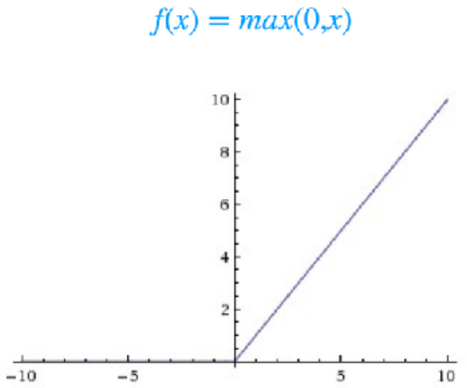
Feature Map Size = $7 - 3 + 1 = 5$



- We use Stride, Padding and Kernel size to control the output size of our Feature Map
- $(n \times n) * (f \times f) = (\frac{n + 2p - f}{s} + 1) \times (\frac{n + 2p - f}{s} + 1)$

Activation Layer ReLU - Adds Non-Linearity to our Network





Max Pooling - Reduce Dimensionality

4	123	1	34
56	99	222	253
45	122	165	12
21	187	133	124

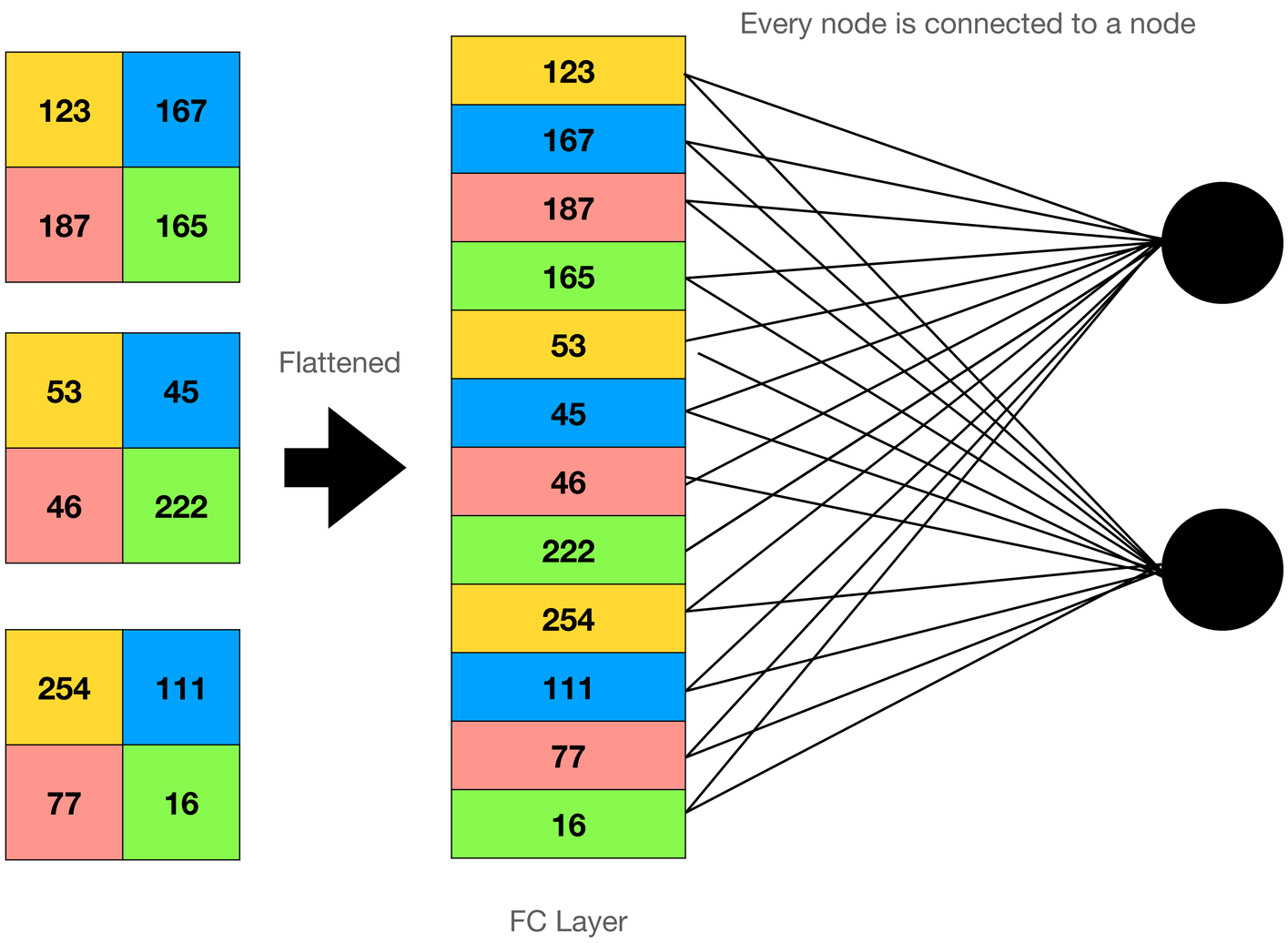
MaxPool Operation

Stride = 2

Kernel = 2x2

123	167
187	165

Fully Connected Layer - Max Pool Layer is Flattened



Softmax Layer

Logits Scores

2.0

1.0

0.1

$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$$

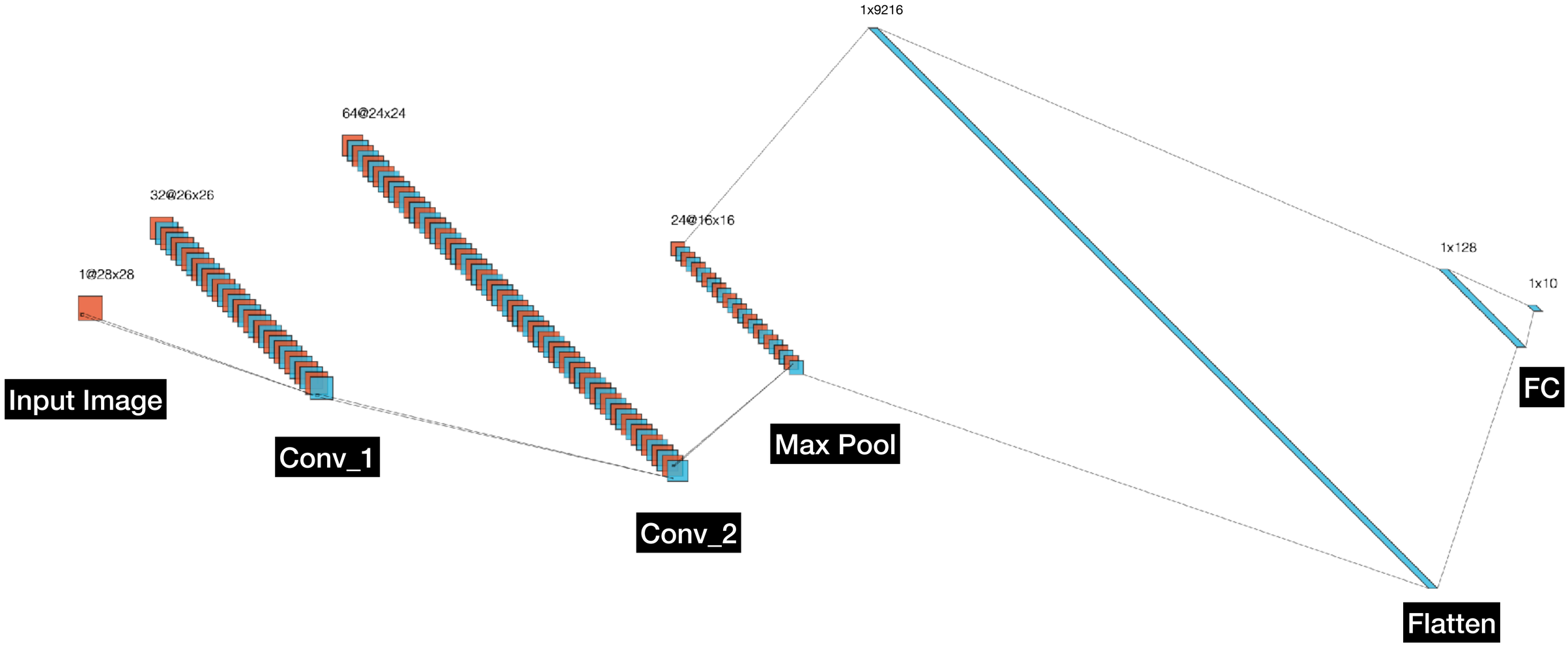
Probabilities

0.7

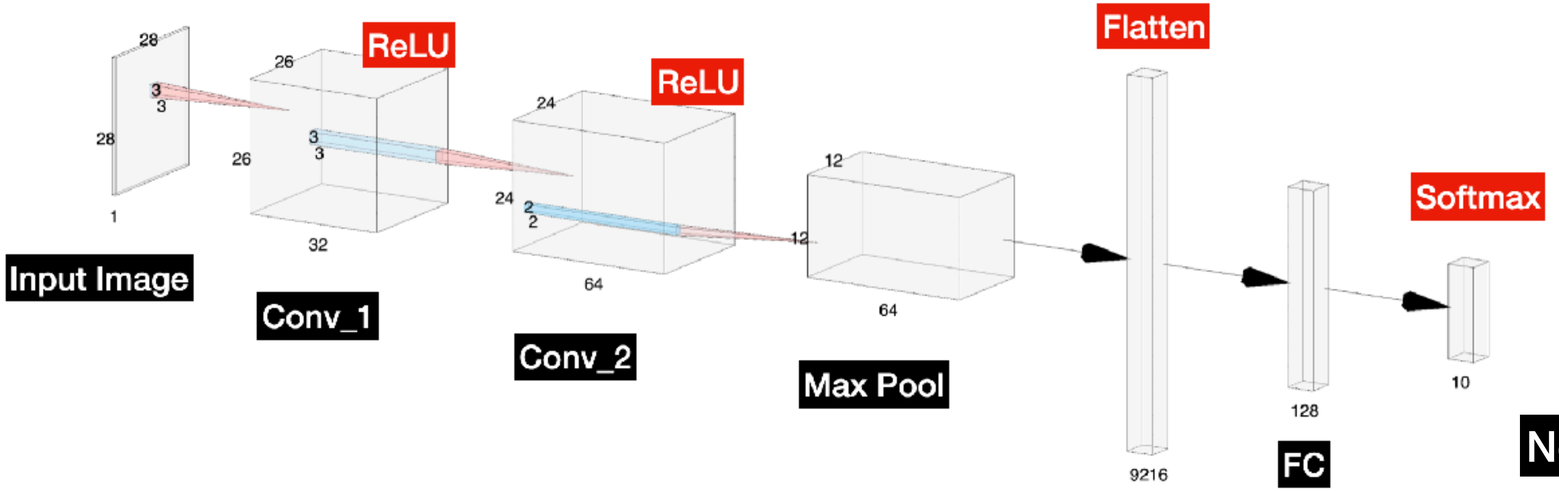
0.2

0.1

Our Basic CNN



Parameters in our Basic CNN



Layer	Parameters
Conv_1 + ReLU	**320**
Conv_2 + ReLU	**18494**
Max Pool	**0**
Flatten	**0**
FC_1	**1,179,776**
FC_2 (Output)	**1,290**
Total	**1,199,882**

Conv_1

$$((Height \times Width \times Depth) + bias) \times N_f$$

$$((3 \times 3 \times 1) + 1) \times 32 = 320$$

Conv_2

$$((Height \times Width \times Depth) + bias) \times N_f$$

$$((3 \times 3 \times 32) + 1) \times 64 = 18{,}494$$

No Trainable Parameters

- Max Pool
- Flatten
- ReLU

Fully Connected/Dense

$$(Length + bias) \times N_{nodes}$$

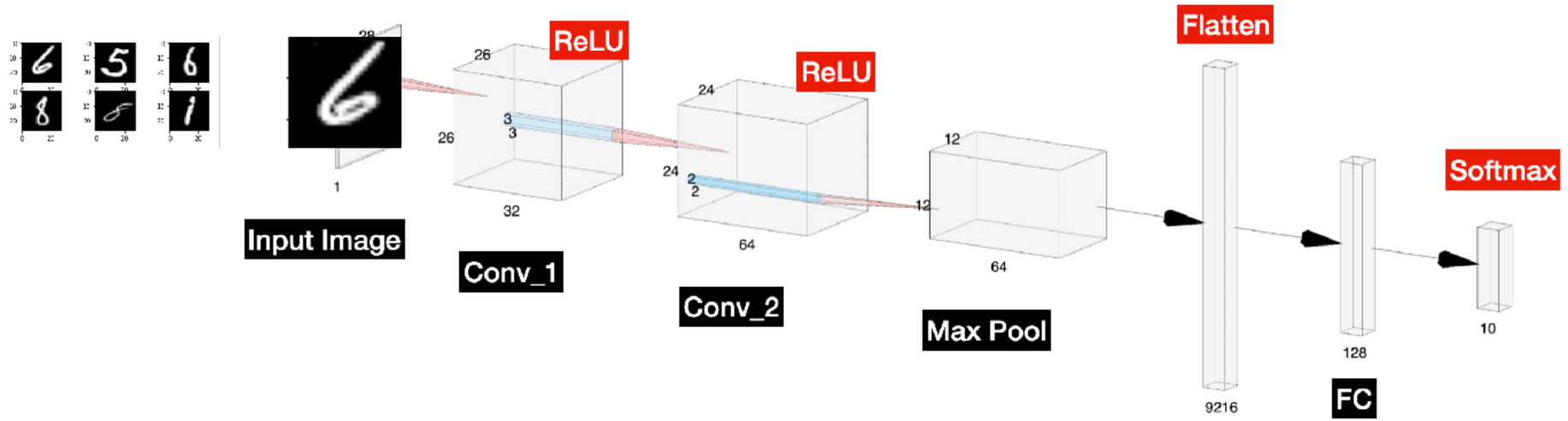
$$(9216 + 1) \times 128 = 1{,}179{,}776$$

Final Output (FC/Dense)

$$(Length + bias) \times N_{nodes}$$

$$(128 + 1) \times 10 = 1{,}290$$

The Training Process



P	6	5	6	8	8	1
0	**0**	0.9	0.83	0.21	0.19	0.62
1	**0.73**	0.8	0.89	0.7	0.92	0.07
2	**0.78**	0.88	0.19	0.39	0.08	0.74
3	**0.37**	0.56	0.07	0.64	0.64	0.9
4	**0.63**	0.25	0.79	0.94	0.52	0.55
5	**0.87**	0.65	0.57	0.63	0.97	0.04
6	**0.67**	0.05	0.45	0.51	0.87	0.51
7	**0.71**	0.66	0.13	0.59	0.86	0.89
8	**0.51**	0.88	0.59	0.01	0.37	0.63
9	**0.24**	0.52	0.79	0.15	0.63	0.78

Overview on Training

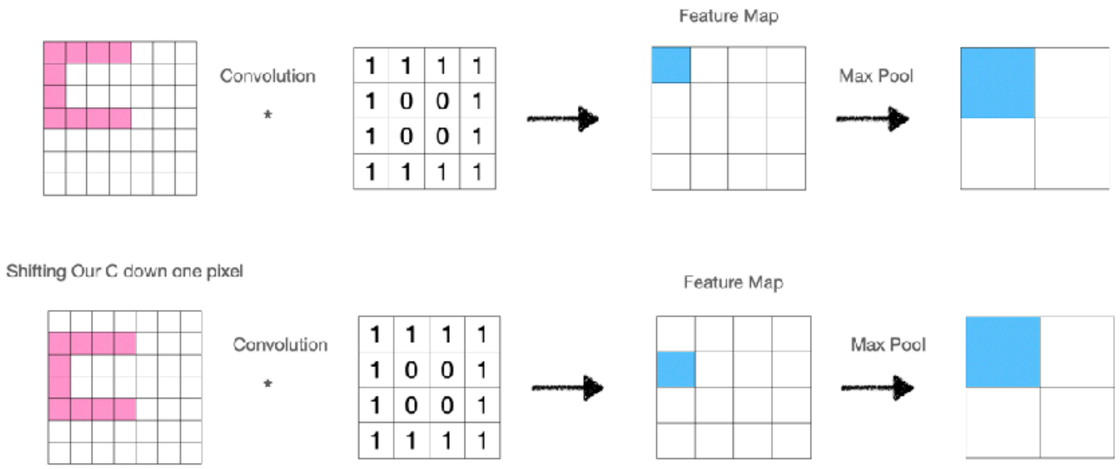
- CNN Model is **designed and defined**
- Weights are **initialised with random values**
- Batches of Images (typically 8 to 256) are **forward propagated** through our CNN Model
- Using **Back Propagation** with **Mini-Batch Gradient Descent** we update the individual weights (right to left)
- Using we update all our weights so that we have a lower loss
- We the entire dataset of images is forward propagated, we've completed an **Epoch**
- We train for **5 to 50** Epochs and stop when Loss stops decreasing

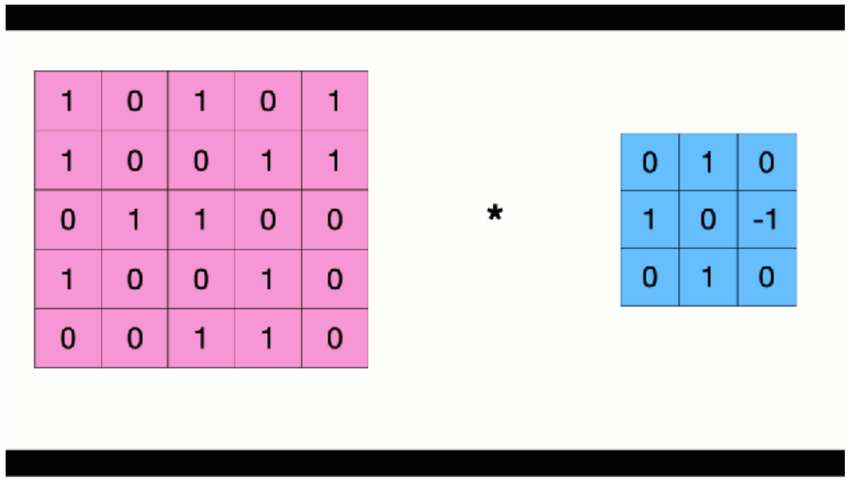
Batches, Mini-Batches, Iterations & Epochs

- We can feed images one at a time, however using mini-batches is better

Advantages of Convolution Neural Networks

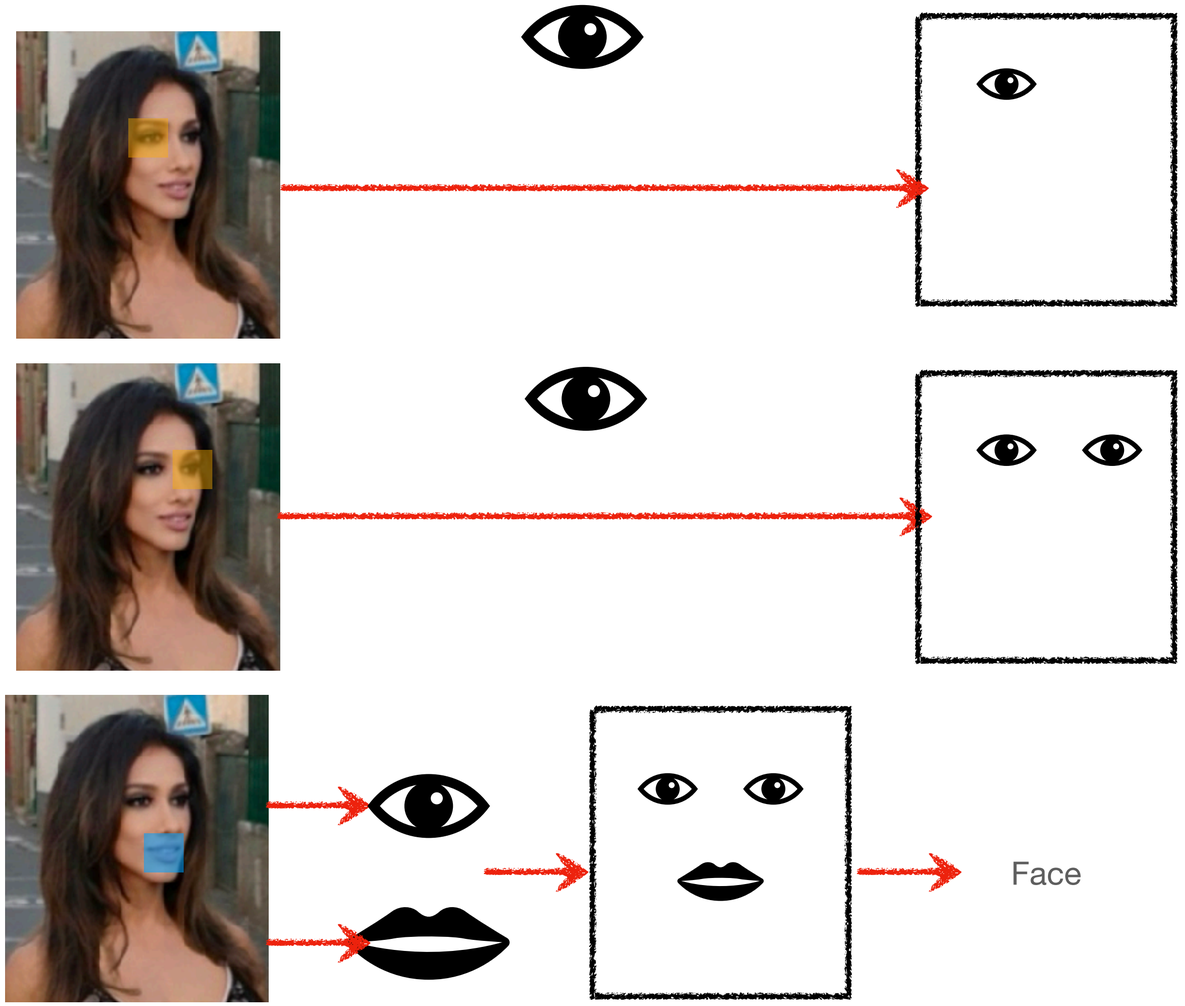
- **Invariance** - Remember our Max Pool Example
- **Parameter sharing** - where a single filter can be used all parts of an image
- **Sparsity of connections** - As we saw, fully connected layers in a typical Neural Network result in a weight matrix with large number of parameters.

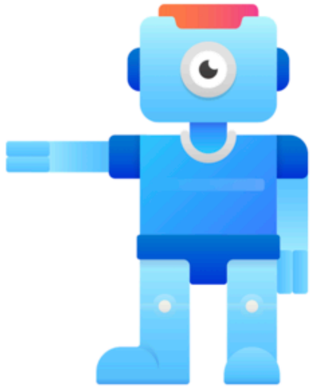




Convolution Neural Networks Assumptions

- Low-level features are local
- Features are translational invariant
- High-level features are made up of low-level features





MODERN COMPUTER VISION

BY RAJEEV RATAN

Next...

History of Deep Learning and AI