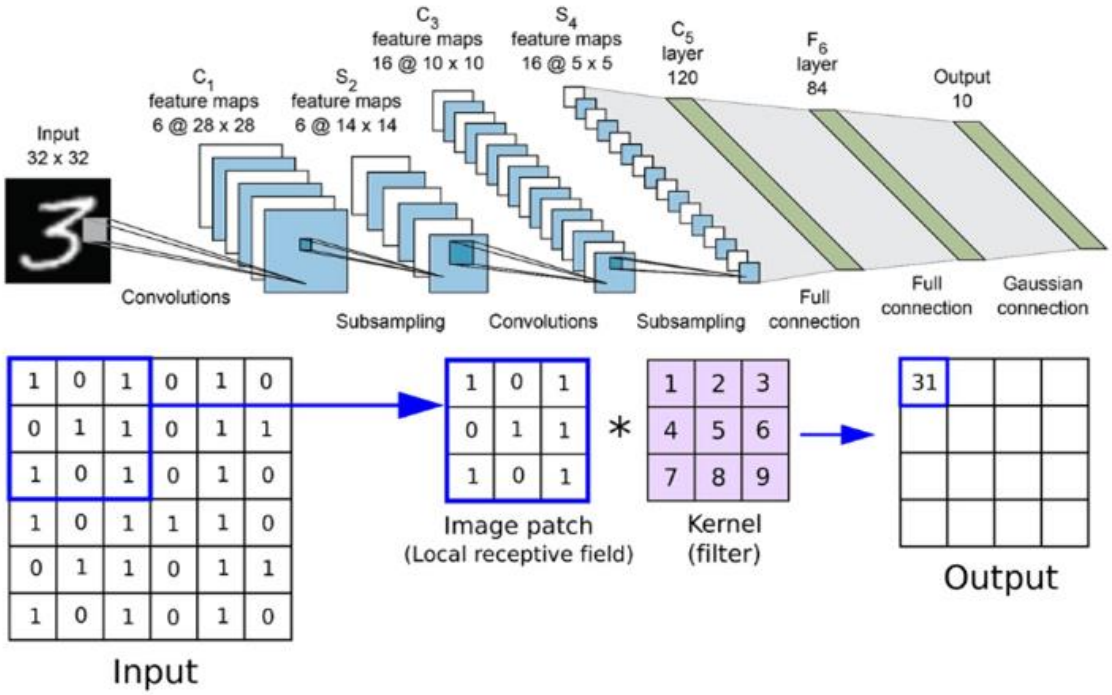
CNN Architectures

CNNs: An Overview



CNNs: An Overview

- If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.
  - So they can never learn to be different features.
  - We break symmetry by initializing the weights to have small random values.

CNNs: An Overview

After propagating the input features forward to the output layer through the various hidden layers consisting of different/same activation functions, we come up with a predicted probability of a sample belonging to the positive class

- Now, the backpropagation algorithm propagates backward from the output layer to the input layer calculating the error gradients on the way.
- Once the computation for gradients of the cost function w.r.t each parameter (weights and biases) in the neural network is done, the algorithm takes a gradient descent step towards the minimum to update the value of each parameter in the network using these gradients.

Vanishing and Exploding Gradient

- During backpropagation, the gradients get smaller and smaller, gradually approaching zero: **Vanishing Gradient**

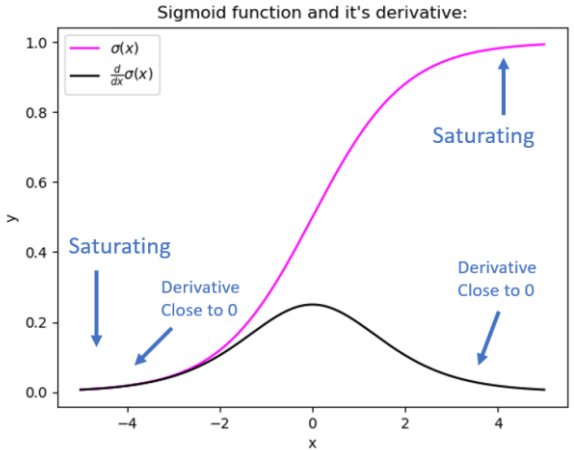
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the *vanishing gradients* problem.

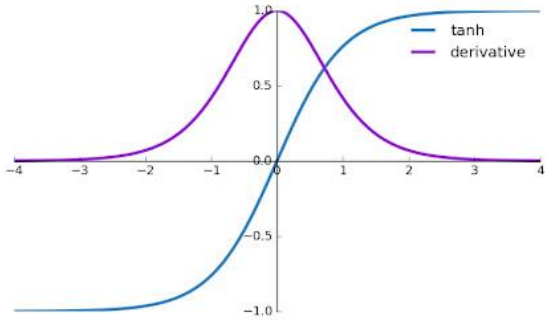
- The gradient becomes too large: **Exploding Gradient**

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the *exploding gradients* problem.

Vanishing gradient

- Weights remain unchanged due to zero gradient; hence model never converges
- Mostly happens with sigmoid and tanh activations
- Sigmoid: squeezes input to range[0,1]
- Too large/too small := 1 or 0 resulting in saturation region
- Derivative of saturating region is zero





Vanishing gradient

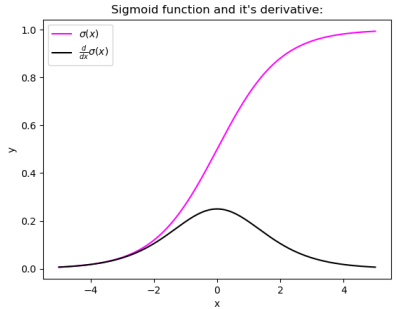
- Huge problem in deep networks
- The effect of multiplying *n* of these small numbers to compute gradients of the early layers in an *n*-layer network, meaning that the gradient decreases exponentially with *n* while the early layers train very slowly and thus the performance of the entire network degrades

Exploding Gradient

- Exploding gradients occur due to the weights in the Neural Network, not the activation function.
- The gradient linked to each weight in the Neural Network is equal to a product of numbers. If this contains a product of values that is greater than one, there is a possibility that the gradients become too large.
- The weights in the lower layers of the Neural Network are more likely to be affected by Exploding Gradient as their associated gradients are products of more values. This leads to the gradients of the lower layers being more unstable, causing the algorithm to diverge.

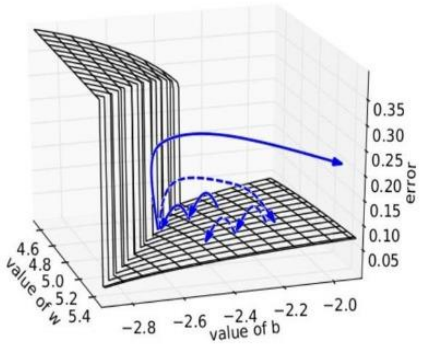
Why do gradients even vanish/explode

- The logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs
- They shrink and transform a larger input space into a smaller output space that lies between the range of [0,1]
- For larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero
- Hence no gradient for backpropagation!



Why do gradients even vanish/explode

- In some cases the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network



How to identify vanishing and exploding gradient

Vanishing

- Large changes are observed in parameters of later layers, whereas parameters of earlier layers change slightly or stay unchanged
- In some cases, weights of earlier layers can become 0 as the training goes
- The model learns slowly and often times, training stops after a few iterations
- Model performance is poor

Exploding

- Contrary to the vanishing scenario, exploding gradients shows itself as unstable, large parameter changes from batch/iteration to batch/iteration
- Model weights can become NaN very quickly
- Model loss also goes to NaN

Solving vanishing gradient problem

- **Initialization techniques:** Ensure proper initialization of weights using methods like Xavier/Glorot or He initialization. This prevents gradients from becoming too small as they propagate through the network.
- **Use of different activation functions:** Employ activation functions like ReLU, Leaky ReLU, or ELU which are less prone to the vanishing gradient problem compared to sigmoid or tanh.
- **Batch normalization:** Normalize activations of each layer to mitigate internal covariate shift, keeping gradients within a certain range and preventing them from vanishing.
- **Proper architecture design:** Avoid excessively deep networks, as deeper networks are more susceptible to vanishing gradients. Utilize architectures with skip connections or shorter pathways to facilitate gradient flow.

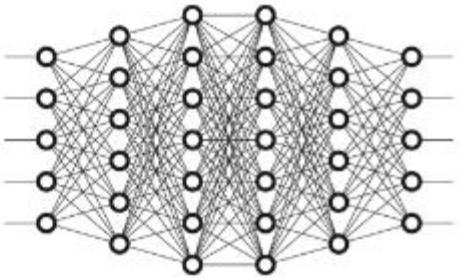
Solving Exploding gradient problem

- **Gradient clipping:** Limit the magnitude of gradients during training by clipping them to a predefined range, preventing them from becoming too large and causing instability.
- **Optimization algorithms:** Use optimization algorithms like Adam, RMSprop, or AdaGrad, which adaptively adjust the learning rate based on the magnitude of gradients to mitigate the impact of exploding gradients.
- **Regularization:** Apply techniques such as L1 or L2 regularization, dropout, or early stopping to prevent overfitting and stabilize training, reducing the likelihood of gradients exploding.
- **Simplify the model:** Reduce the complexity of the neural network architecture by decreasing the number of layers or neurons, which can help mitigate exploding gradients by reducing the parameter space and the potential for large gradients.

Weight initialization methods

- Training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization.
- The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether.

$$W = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1j} \\ W_{21} & W_{22} & \cdots & W_{2j} \\ \vdots & \vdots & \cdots & \vdots \\ W_{n1} & W_{n2} & \cdots & W_{nj} \end{bmatrix}$$



Weight initialization methods

Zero initialisation:

- All weights are set to 0.
- Ineffective as neurons learn same feature during each iteration

Random initialisation: random normal and random uniform

Random normal: weights are drawn from normal distribution randomly

$$\mathbf{W} \sim G\left(0, \sigma^2\right)$$

Random uniform: weights are drawn from uniform distribution randomly

W ~ uniform[a,b]

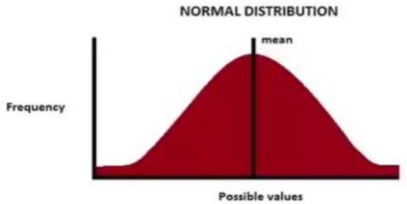
Weight initialization methods

LeCun initialisation: normalize variance

- Prevents vanishing or exploding gradient
- Solves growing variance by replacing with constant variance

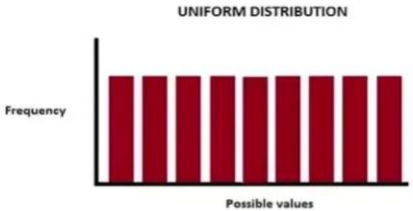
- LeCun Normal Initialization

$$W \sim N\left(0, \sqrt{\frac{1}{n_{in}}}\right)$$



- LeCun Uniform Initialization

$$W \sim U\left(-\sqrt{\frac{1}{n_{in}}}, +\sqrt{\frac{1}{n_{in}}}\right)$$



n_{in}: the number of previous node

Weight initialization methods

Xavier or Glorot initialisation:

- weights such as variance of activations are same across every layer. Prevents gradient from exploding or vanishing
- Used widely for tanh and sigmoid activation functions
- Dying neuron problem: gradient can be 0 while using ReLU activation

$$\mathbf{W} \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right), \quad \mathbf{b} = 0$$

n_i: the number of incoming network connections.

n_{i+1}: the number of outgoing network connections.

Weight initialization methods

He or Kaiming initialisation:

- draws weights from gaussian distribution and dampens vanishing gradient problem
- Used mainly with ReLU activation functions

$$\mathbf{W} \sim G\left(0, \frac{2}{n_i}\right), \quad \mathbf{b} = 0$$

n_i: the number of incoming network connections.

Weight initialization methods

Initialization method	Pros.	Cons.
All-zeros / constant	Simplicity	Symmetry problem leading neurons to learn the same features
Random	Improves the symmetry-breaking process	- A saturation may occur leading to a vanishing gradient - The slope or gradient is small, which can cause the gradient descent to be slow
LeCun	Solves growing variance and gradient problems	- Not useful in constant-width networks - Takes into account the forward propagation of the input signal - This method is not useful when the activation function is non-differentiable
Xavier	Decreases the probability of the gradient vanishing/exploding problem	- This method is not useful when the activation function is non-differentiable - Dying neuron problem during the training
He	Solves dying neuron problems	- This method is not useful for layers with differentiable activation function such as ReLU or LeakyReLU

Architectures

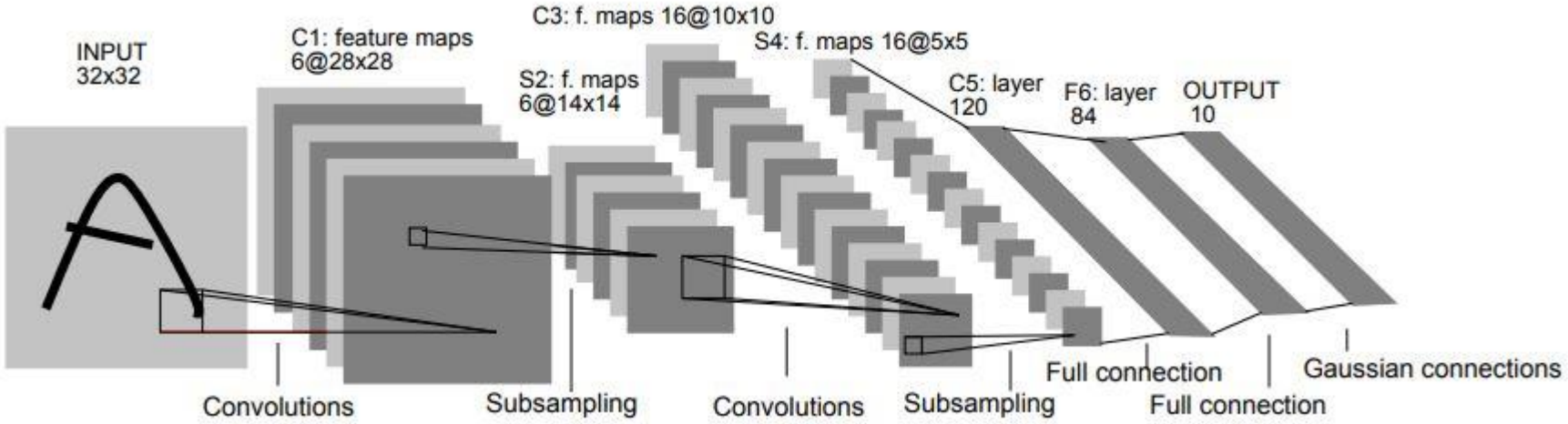
Various models are introduced:

- LeNet-5
- AlexNet
- VGG
- ResNet
- Inception

LeNet-5

- Introduced by Yann LeCun and his team in the 1990s
- First deep CNN architecture
- Designed for handwritten digit recognition
- LeNet-5 CNN architecture has seven layers. Three convolutional layers, two subsampling layers, and two fully linked layers make up the layer composition.
- Can even run on CPU

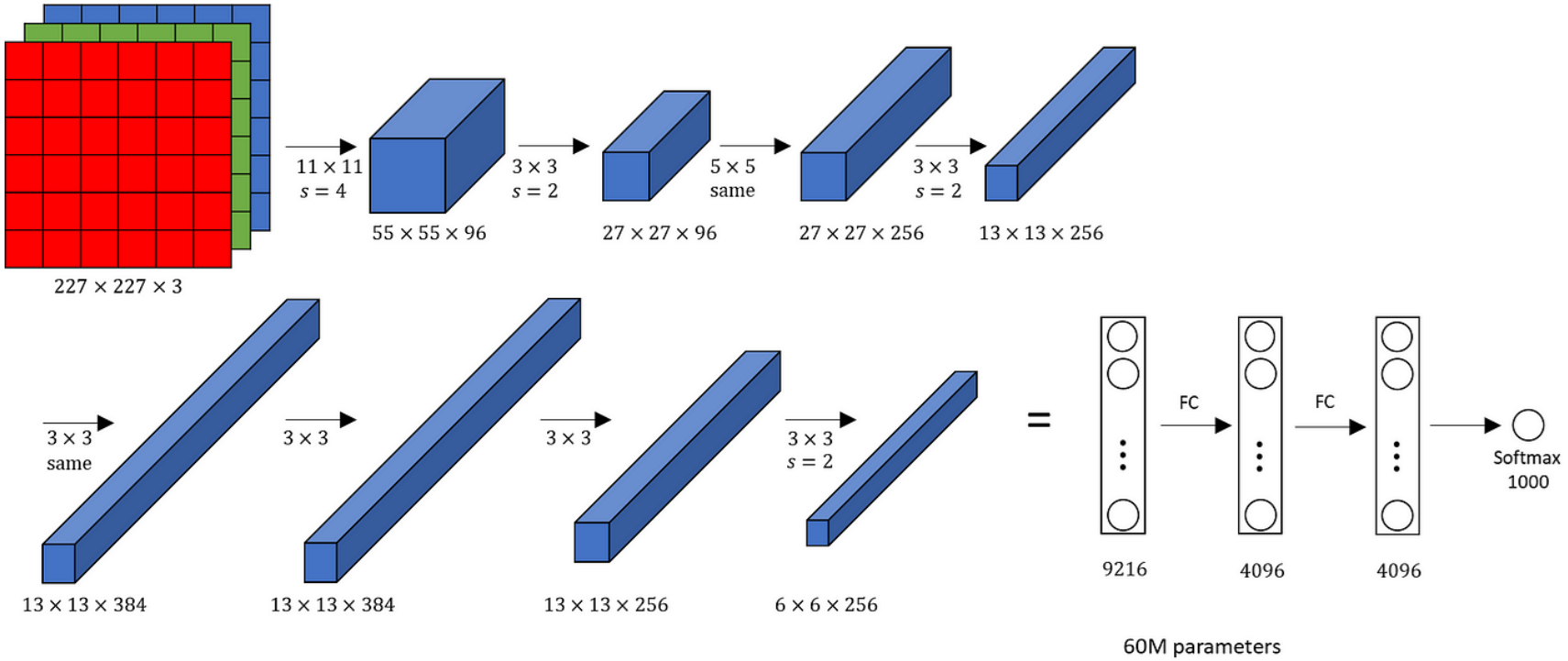
LeNet-5


INPUT
32x32
C1: feature maps
6@28x28
S2: f. maps
6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer
120
F6: layer
84
OUTPUT
10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections

AlexNet

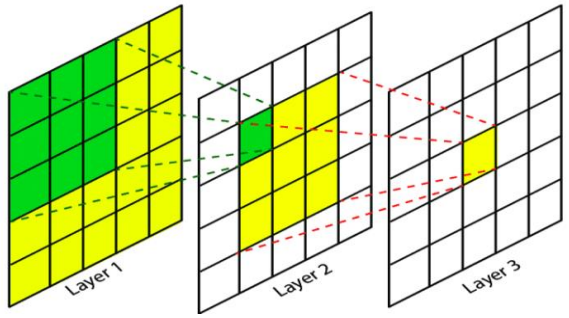
- First to use GPU for improved performance
- Non linear activation ReLU was used in each convolution layer
- Fixed Input size
- Designed to be used with large-scale image datasets
- AlexNet is composed of 5 convolutional layers with a combination of max-pooling layers, 3 fully connected layers, and 2 dropout layers
- Output activation:softmax

AlexNet


227 × 227 × 3
11 × 11
s = 4
55 × 55 × 96
3 × 3
s = 2
27 × 27 × 96
5 × 5
same
27 × 27 × 256
3 × 3
s = 2
13 × 13 × 256
3 × 3
same
13 × 13 × 384
3 × 3
13 × 13 × 384
3 × 3
13 × 13 × 256
3 × 3
s = 2
6 × 6 × 256
=
FC
FC
Softmax
1000
9216
4096
4096
60M parameters

VGG

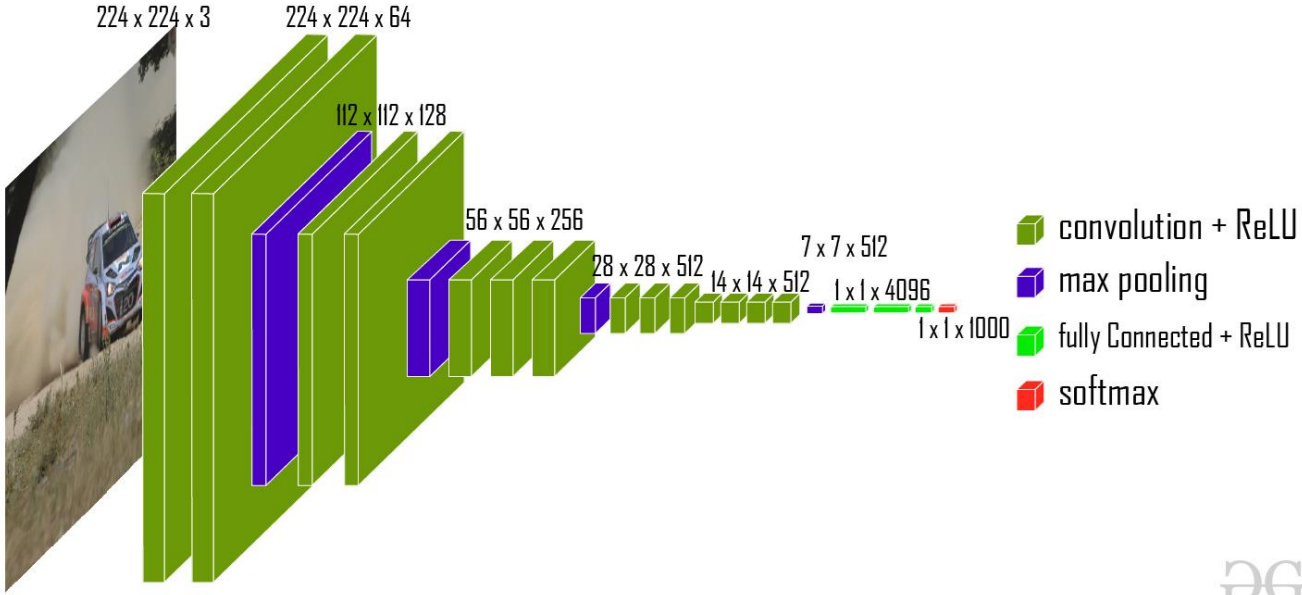
- The Visual Geometry Group (VGG) at Oxford University proposed the VGGNet architecture
- Small receptive field with 3x3 kernels
- Receptive Field (**RF**) is defined as **the size of the region in the input that produces the feature**



VGG

Convolution stride=1 to preserve maximum information

Various depth models, i.e. VGG 16, 19



ResNet

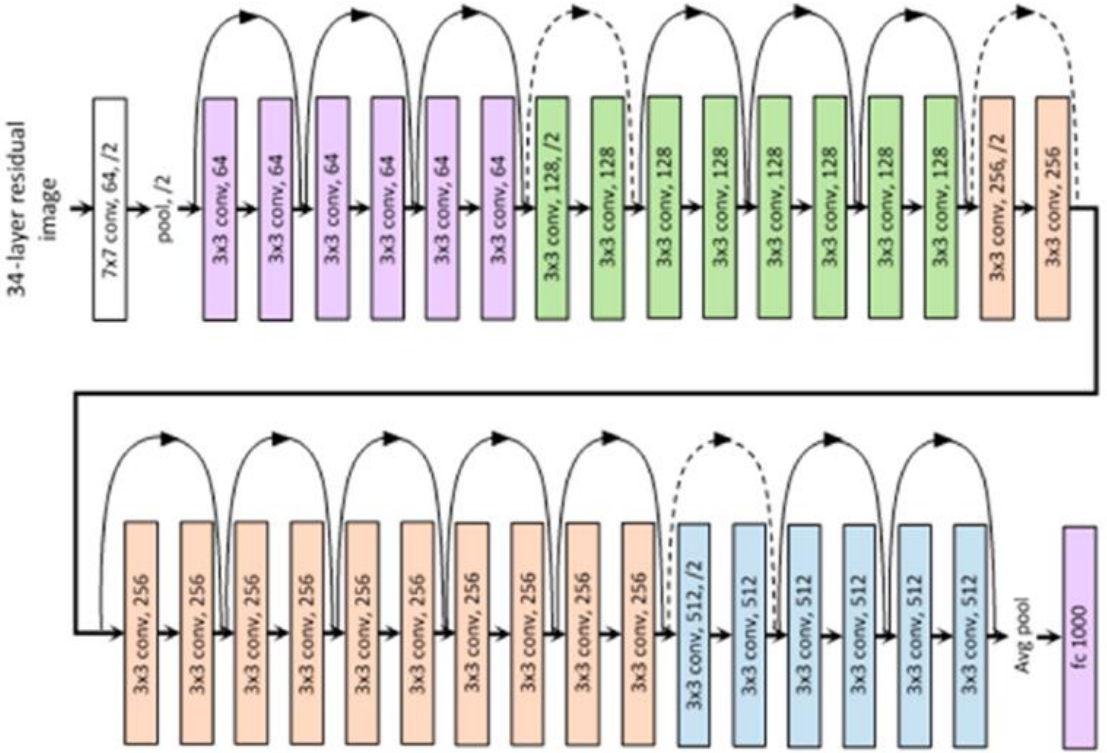
Tackles vanishing gradient problem

Proposes shortcut connections that bypass one or more layer allowing the gradient to flow more easily during backpropagation

facilitated the training of extremely deep networks, reaching hundreds of layers.

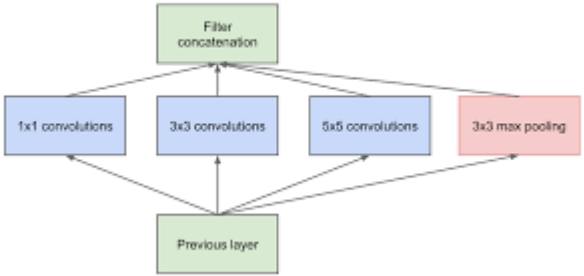
Different versions: ResNet -18.50, 101, 152

ResNet

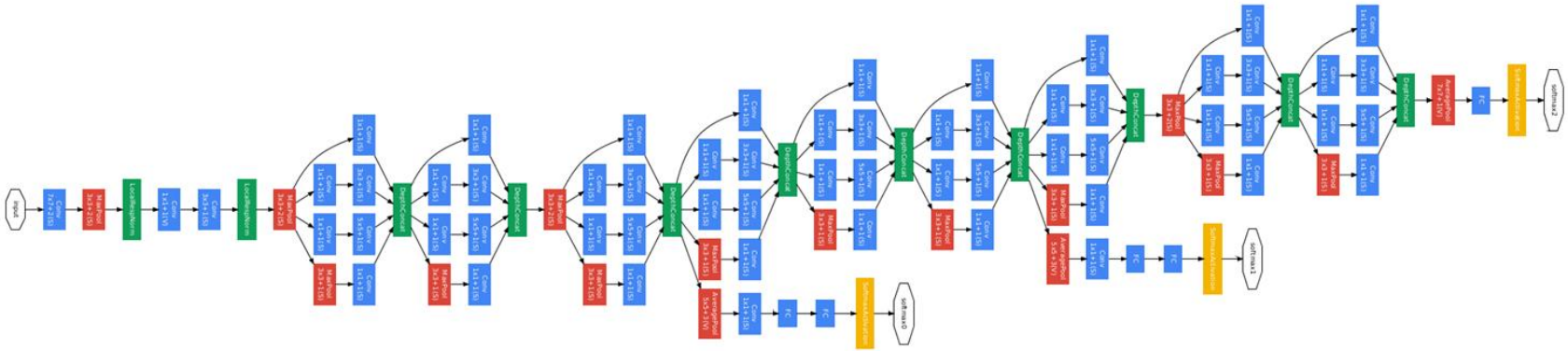

34-layer residual
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
Avg pool
fc 1000

GoogLeNet/Inception

- Employs parallel convolution operation with different kernel sizes\
- Captures both local and global information

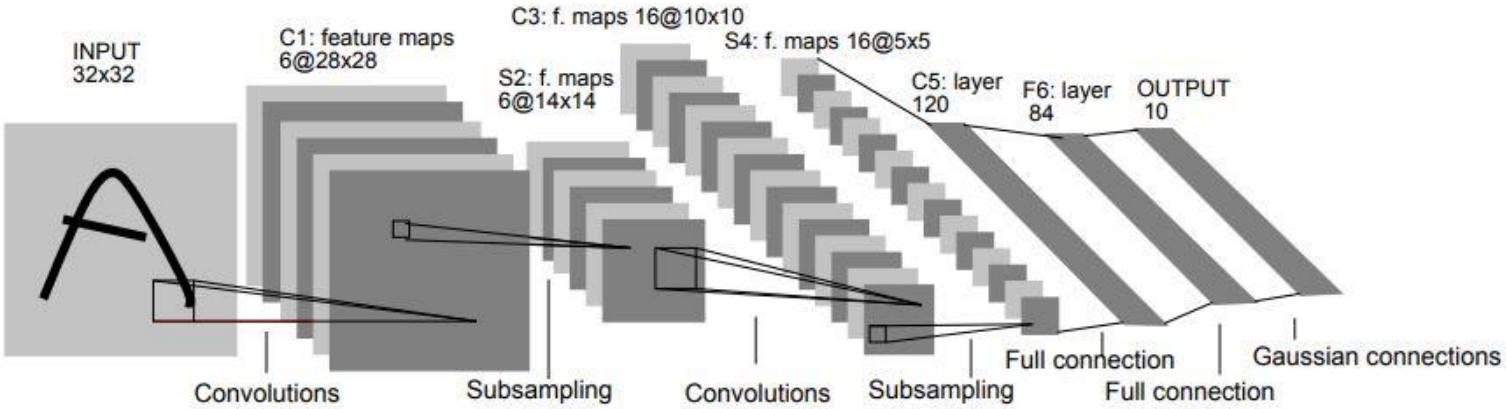


GoogLeNet



Computing Parameters

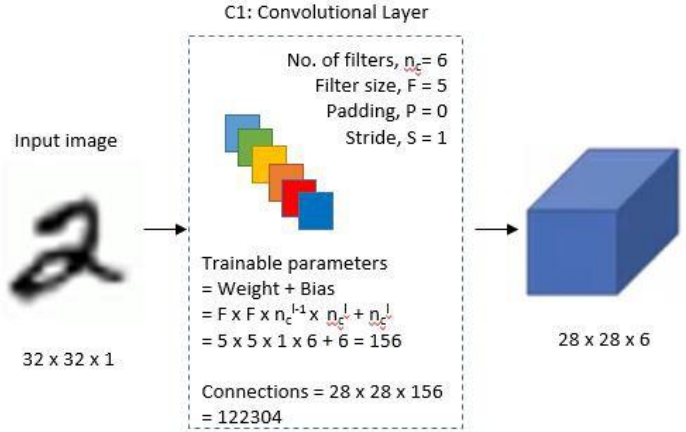
Let’s consider LeNet 5 architecture



Computing Parameters

First Layer:

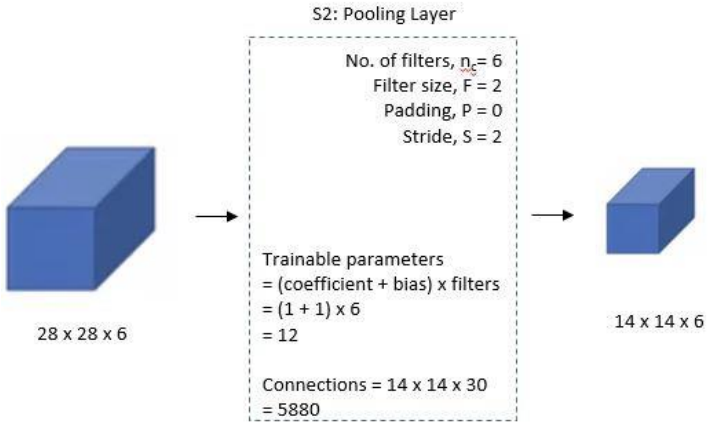
The input for LeNet-5 is a 32×32 grayscale image which passes through the first convolutional layer with 6 feature maps or filters having size 5×5 and a stride of one. The image dimensions changes from 32x32x1 to 28x28x6.



Computing Parameters

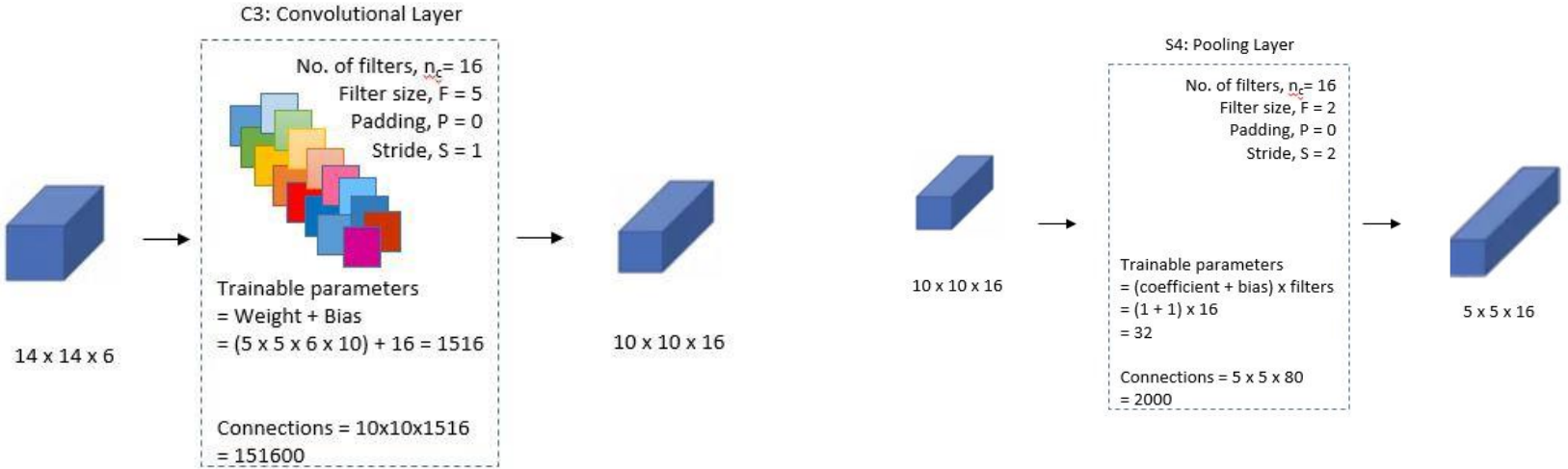
Second Layer:

Then the LeNet-5 applies average pooling layer or sub-sampling layer with a filter size 2×2 and a stride of two. The resulting image dimensions will be reduced to 14x14x6.



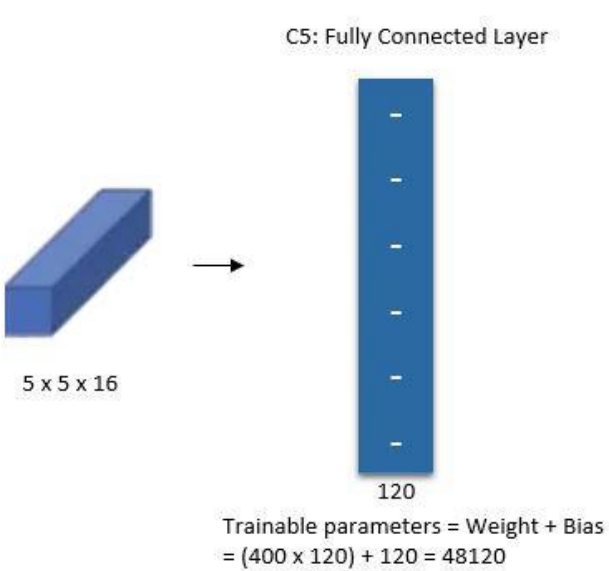
Computing Parameters

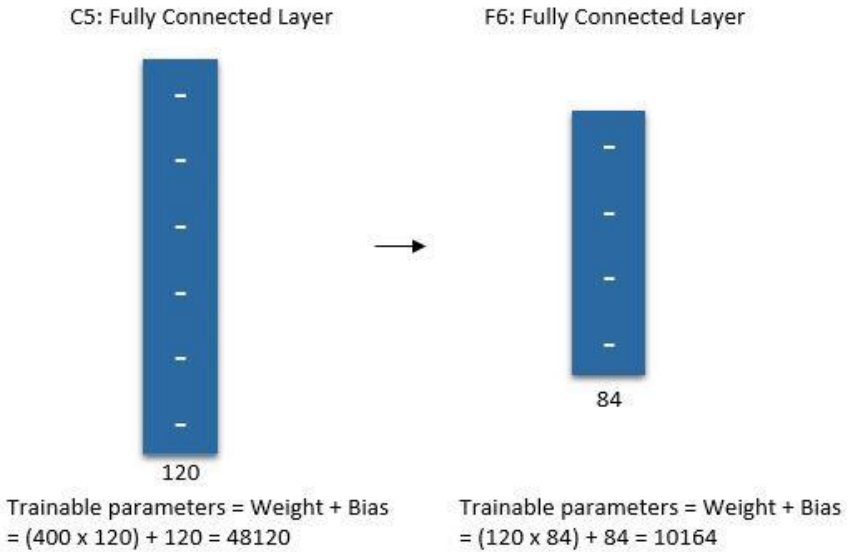
Similarly C3 is third layer and performs convolution and S4 is fourth layer performing average pooling



Computing Parameters

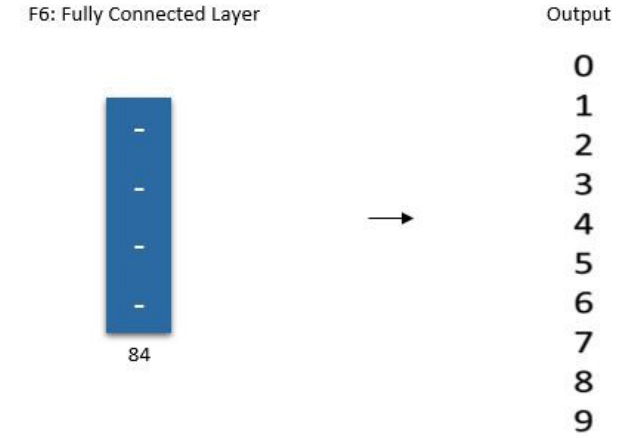
Fifth layer(C5) and sixth layer(C6) is a fully connected layer





Computing Parameters

Output layer

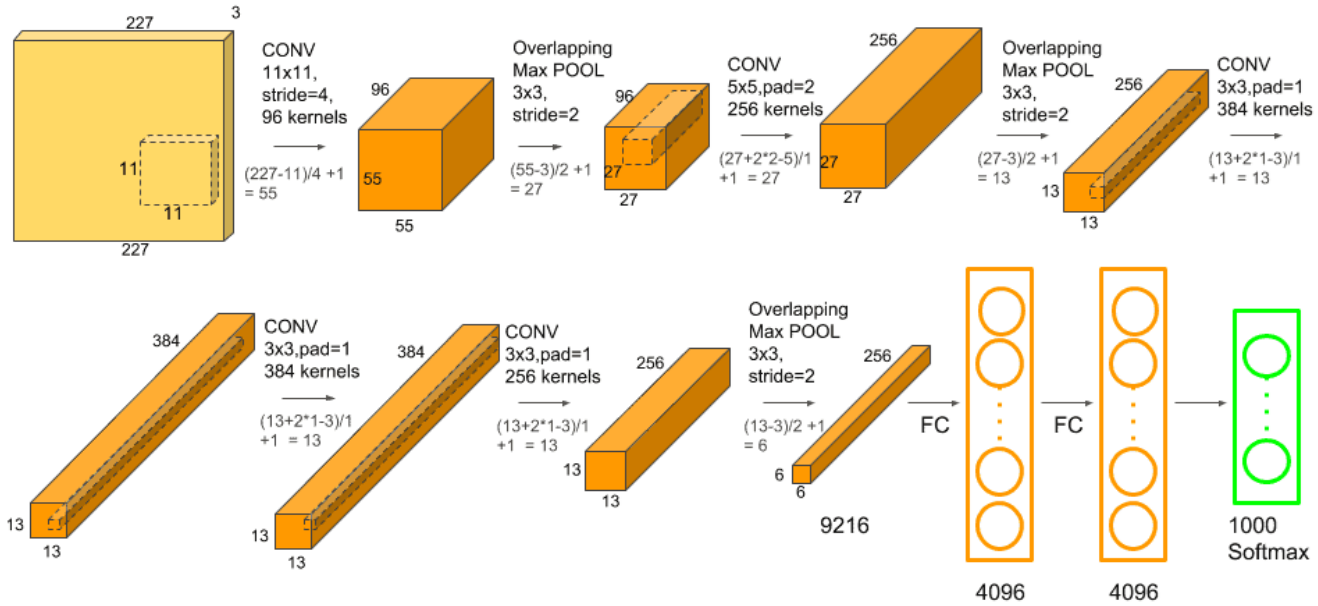


Trainable parameters = Weight + Bias
= (120 x 84) + 84 = 10164

Summary

Layer Name	Input W×H×D	Kernel W×H×D/S	Output W×H×D	Params	Mults
C1: conv2d	32×32×1	5×5×6	28×28×6	1×5×5×6+6 =156	28×28×1×5×5×6 =117,600
S2: pool/2	28×28×6	2×2/2	14×14×6	0	0
C3: conv2d	14×14×6	5×5×16	10×10×16	6×5×5×16+16 =2,416	10×10×6×5×5×16 =240,000
S4: pool/2	10×10×16	2×2/2	5×5×16	0	0
C5: conv2d	5×5×16	5×5×120	1×1×120	16×5×5×120+120 =48,120	1×1×16×5×5×120 =48,000
F6: conv2d	1×1×120	1×1×84	1×1×84	120×1×1×84+84 =10,164	120×84 =10,080
F7: conv2d	1×1×84	1×1×10	1×1×10	84×1×1×10+10 =850	84×40 =840
			Total	61,706	416,520

AlexNet Parameter



AlexNet Parameter

AlexNet Network - Structural Details													
Input			Output			Layer	Stride	Pad	Kernel size		in	out	# of Param
227	227	3	55	55	96	conv1	4	0	11	11	3	96	34944
55	55	96	27	27	96	maxpool1	2	0	3	3	96	96	0
27	27	96	27	27	256	conv2	1	2	5	5	96	256	614656
27	27	256	13	13	256	maxpool2	2	0	3	3	256	256	0
13	13	256	13	13	384	conv3	1	1	3	3	256	384	885120
13	13	384	13	13	384	conv4	1	1	3	3	384	384	1327488
13	13	384	13	13	256	conv5	1	1	3	3	384	256	884992
13	13	256	6	6	256	maxpool5	2	0	3	3	256	256	0
						fc6			1	1	9216	4096	37752832
						fc7			1	1	4096	4096	16781312
						fc8			1	1	4096	1000	4097000
Total													**62,378,344**