VGG-16 Image Recognition

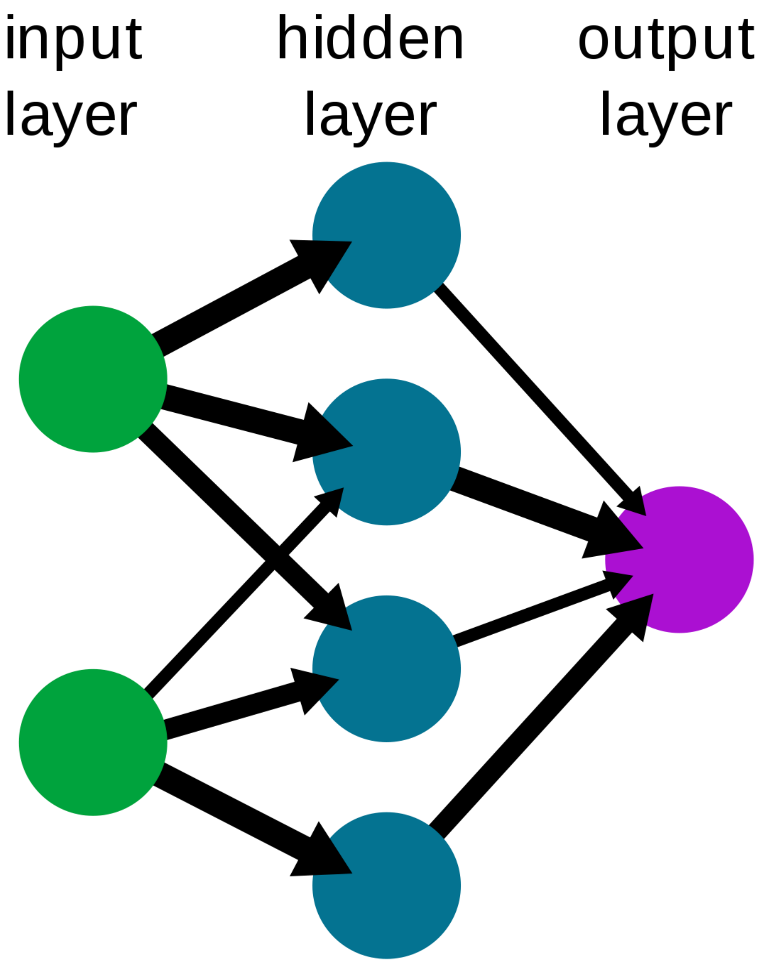
Presented by:

- Aswhin Rathore
- Lazarus S F
- Likitha
- Manudeep
- Sai Krishna Teja Damaraju

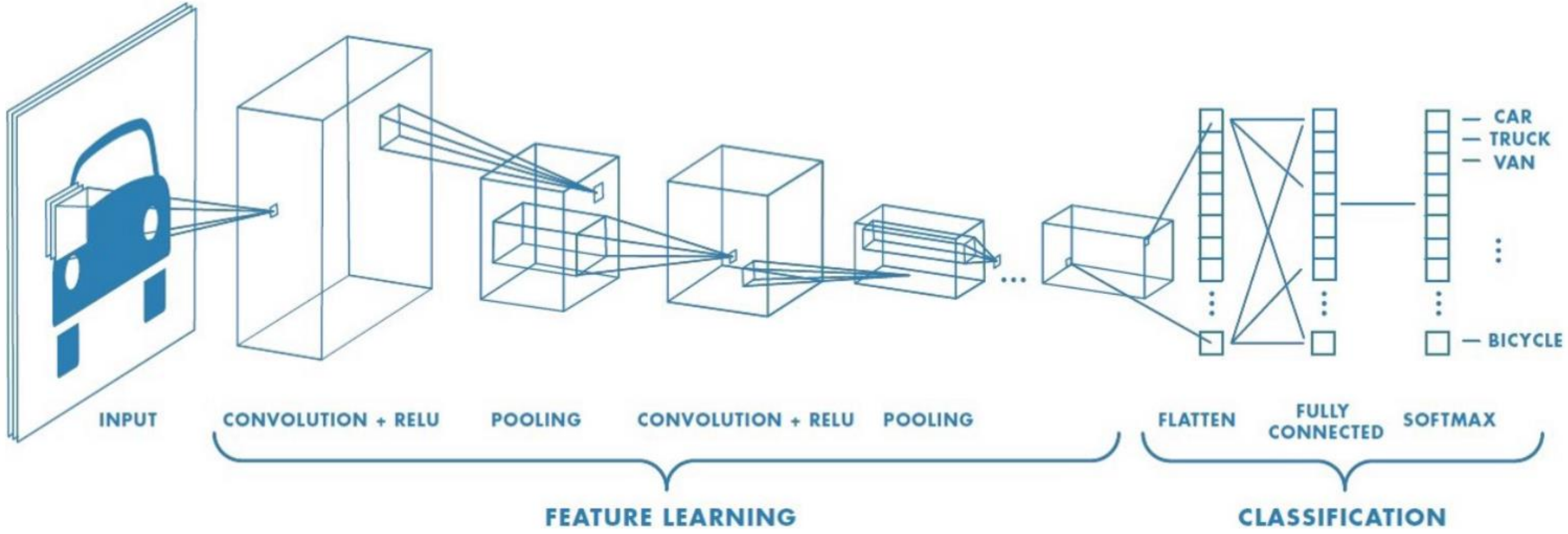
Neural network

- A set of algorithms, modeled loosely after the human brain, that are designed to recognize patterns.
- It comprised of node layers, containing an input layer, one or more hidden layers, and an output layer.
- A Convolutional Neural Network(**CNN**), is a class of neural network that specializes in processing data that has a grid-like topology, such as an image.

A simple neural network

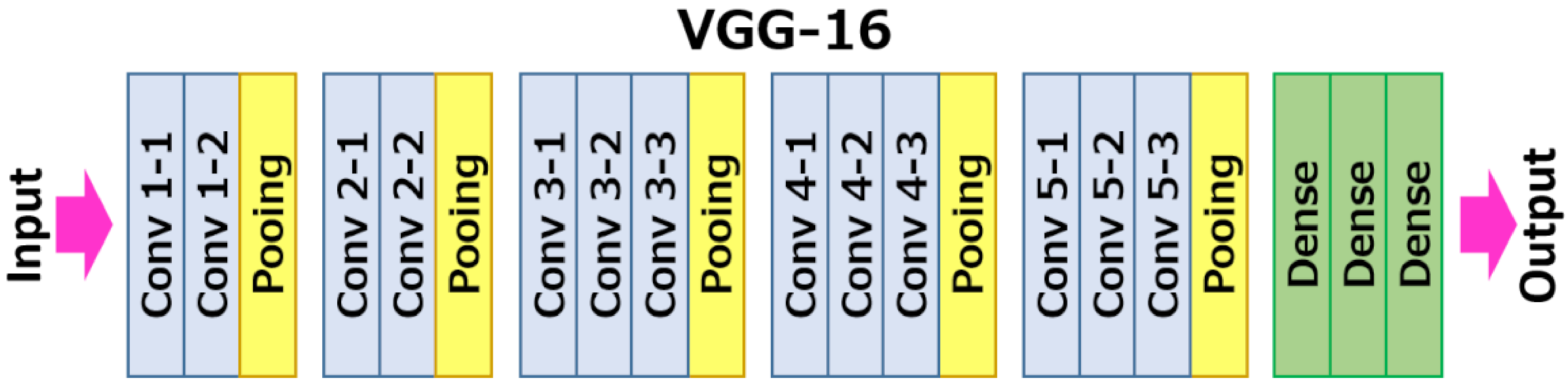


CNN Architecture

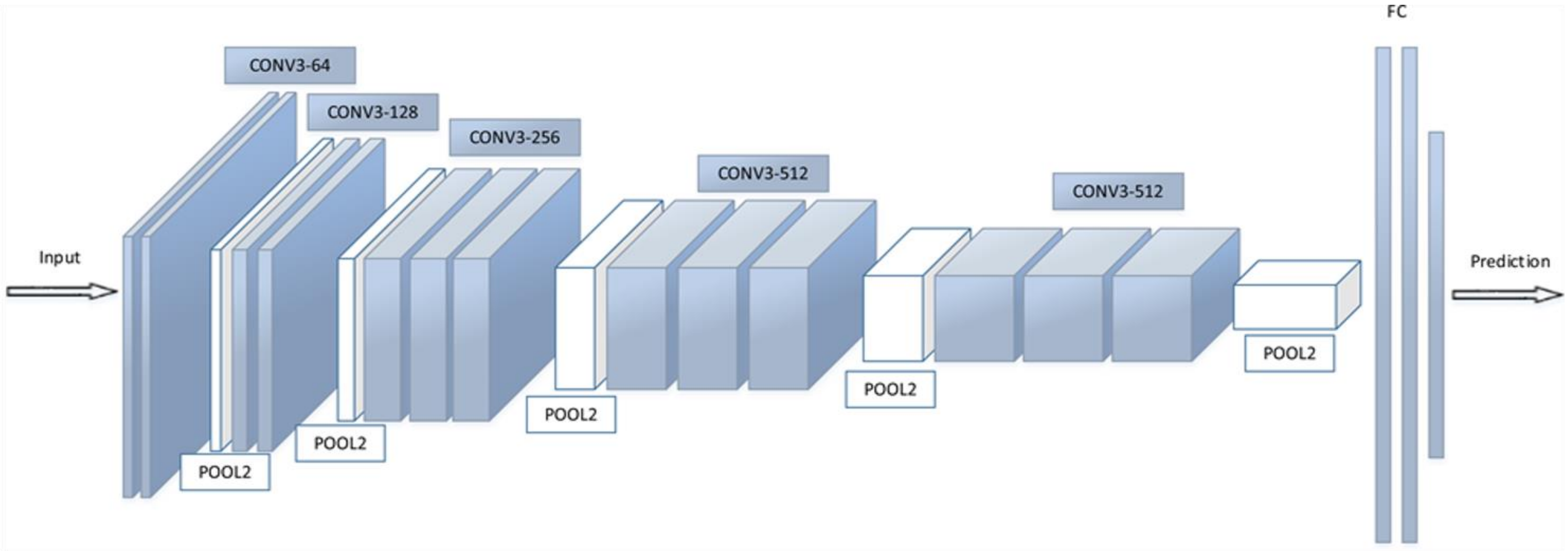


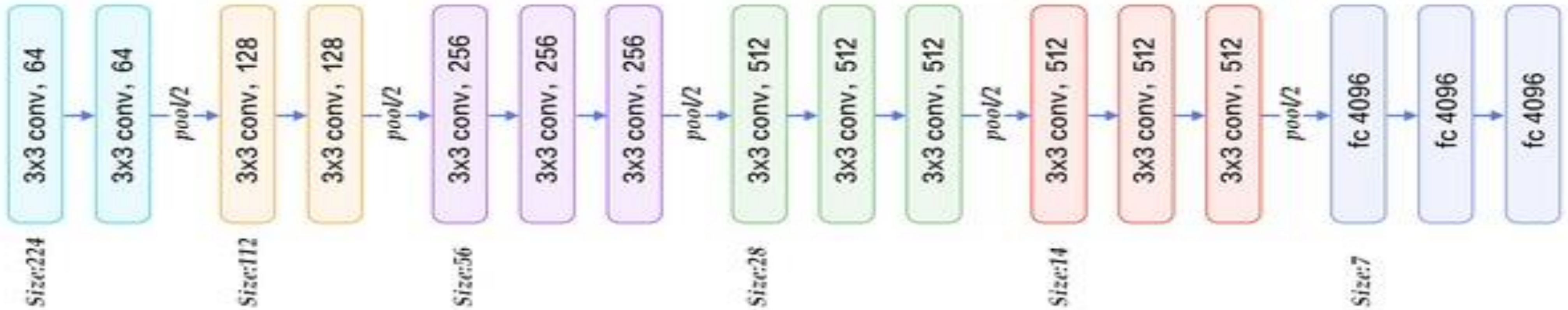
VGG16 – What problem it solves

- **VGG16** is a CNN for Large-scale image recognition.
- It is 16 layers deep.
- It has 13 convolutional layers followed by 3 fully connected layers. The width of the network starts at a small value of 64 and increases by a factor of 2 after every pooling layer. It achieves the top-5 accuracy of 92.3 % on ImageNet Dataset.



VGG-16 Architecture


FC
CONV3-64
CONV3-128
CONV3-256
CONV3-512
CONV3-512
Input
Prediction
POOL2
POOL2
POOL2
POOL2
POOL2


3x3 conv, 64
3x3 conv, 64
pool/2
3x3 conv, 128
3x3 conv, 128
pool/2
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool/2
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool/2
fc 4096
fc 4096
fc 4096
Size:224
Size:112
Size:56
Size:28
Size:14
Size:7

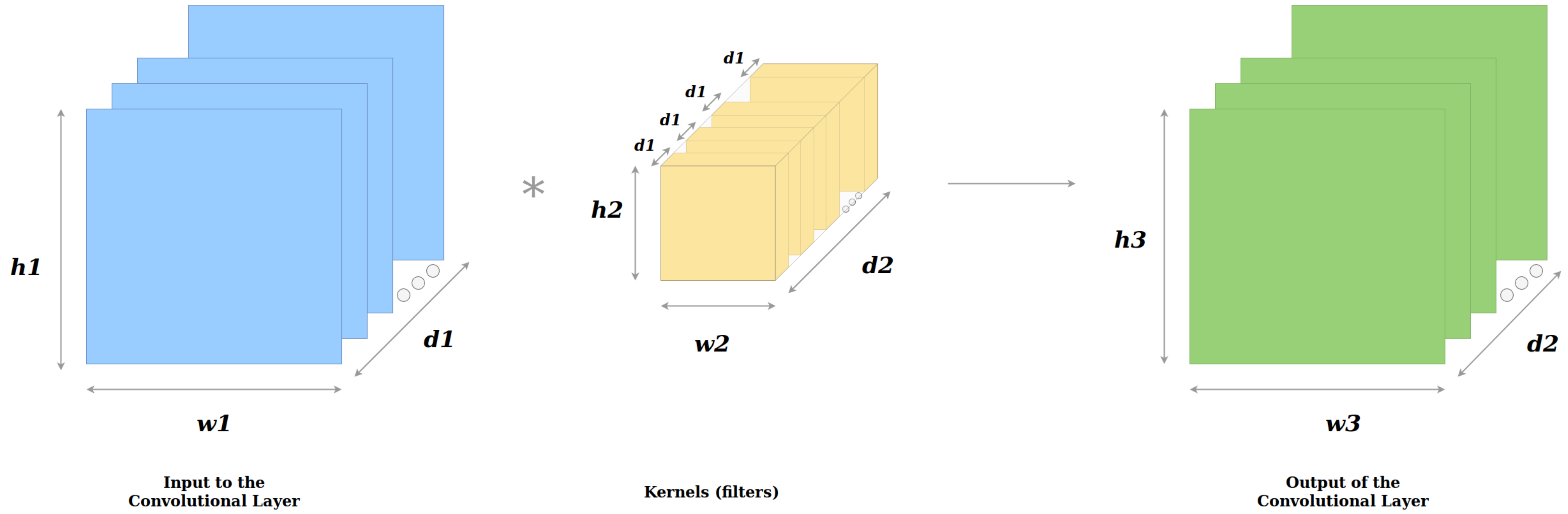
Code : Layers

1. Convolution2D(inputs_channel=1, num_filters=64, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv1')
   + ReLu()
2. Convolution2D(inputs_channel=64, num_filters=64, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv2')
   + ReLu()
   +Maxpooling2D(pool_size=2, stride=2, name='maxpool1')
3. Convolution2D(inputs_channel=64, num_filters=128, kernel_size=3, padding=1, stride=1, learning_rate=lr,name='conv3')
   + Relu()
4. Convolution2D(inputs_channel=128, num_filters=128, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv4')
   + Relu()
   + Maxpooling2D(pool_size=2, stride=2, name='maxpool2')
5. Convolution2D(inputs_channel=128, num_filters=256, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv5')
   + Relu()
6. Convolution2D(inputs_channel=256, num_filters=256, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv6')
   + Relu()
7. Convolution2D(inputs_channel=256, num_filters=256, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv7')
   + Relu()
   Maxpooling2D(pool_size=2, stride=2, name='maxpool3')
8. Convolution2D(inputs_channel=256, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv8')
   + ReLu()
9. Convolution2D(inputs_channel=512, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv9')
   + ReLu()

Code : Layers

10. **Convolution2D**(inputs_channel=512, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv10’)
    + **ReLu**()
    +**Maxpooling2D**(pool_size=2, stride=2, name='maxpool4’)
11. **Convolution2D**(inputs_channel=512, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv11’)
    + **ReLu**()
12. **Convolution2D**(inputs_channel=512, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv12’)
    + **ReLu**()
13. **Convolution2D**(inputs_channel=512, num_filters=512, kernel_size=3, padding=1, stride=1, learning_rate=lr, name='conv13’)
    + **ReLu**()
    + **Maxpooling2D**(pool_size=2, stride=2, name='maxpool5’)
    +**Flatten**()
14. **FullyConnected**(num_inputs=2048, num_outputs=4096, learning_rate=lr, name='fc1')
    + **ReLu**()
15. **FullyConnected**(num_inputs=4096, num_outputs=4096, learning_rate=lr, name='fc2’)
    + **ReLu**()
16. **FullyConnected**(num_inputs=4096, num_outputs=10, learning_rate=lr, name='fc3’)
    + **Softmax**()

Convolution layer



Convolution layer

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

Filter / Kernel

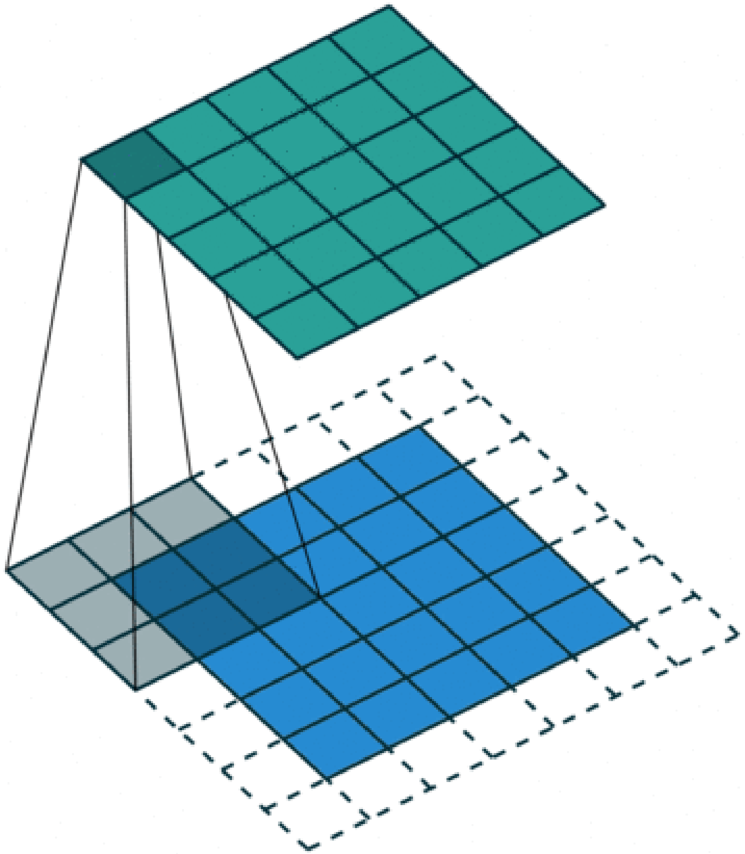
1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

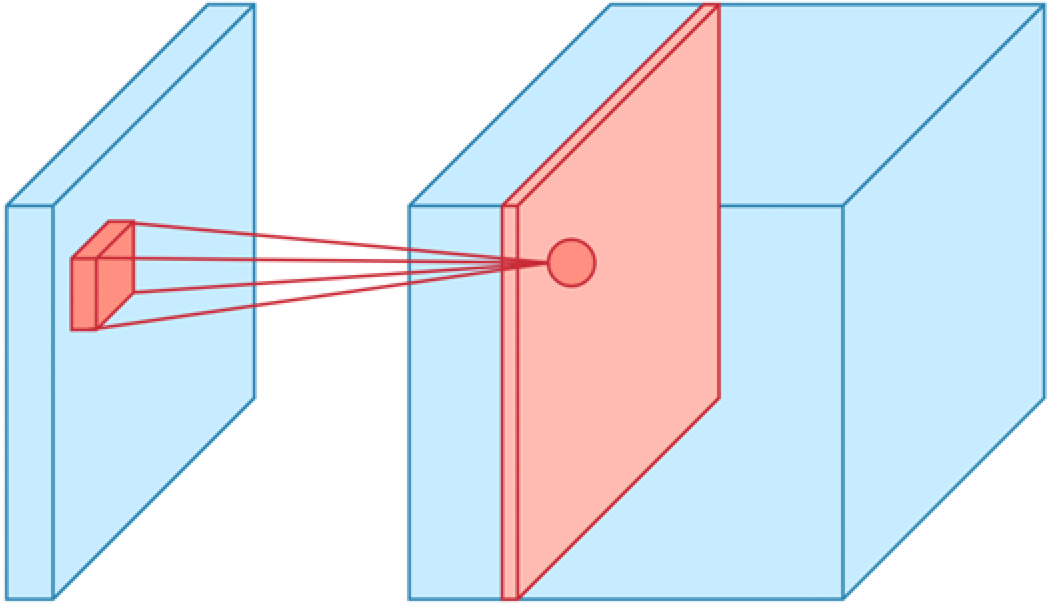
Input x Filter

4		

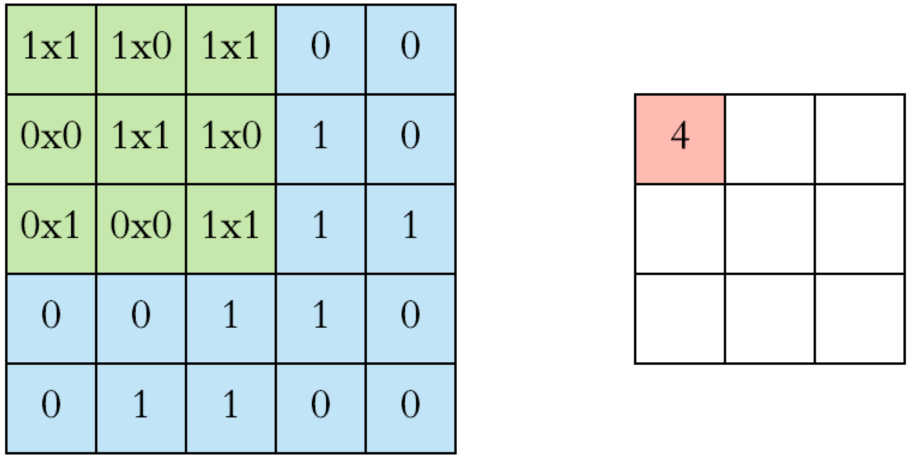
Feature Map

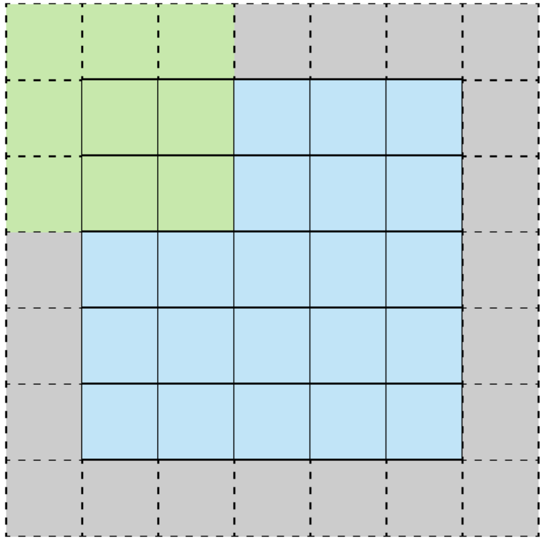
Convolution layer – Calculation of output using 2D convolution



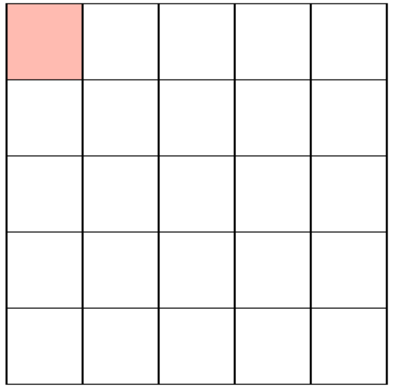


Convolution layer - Calculation of output using 2D convolution



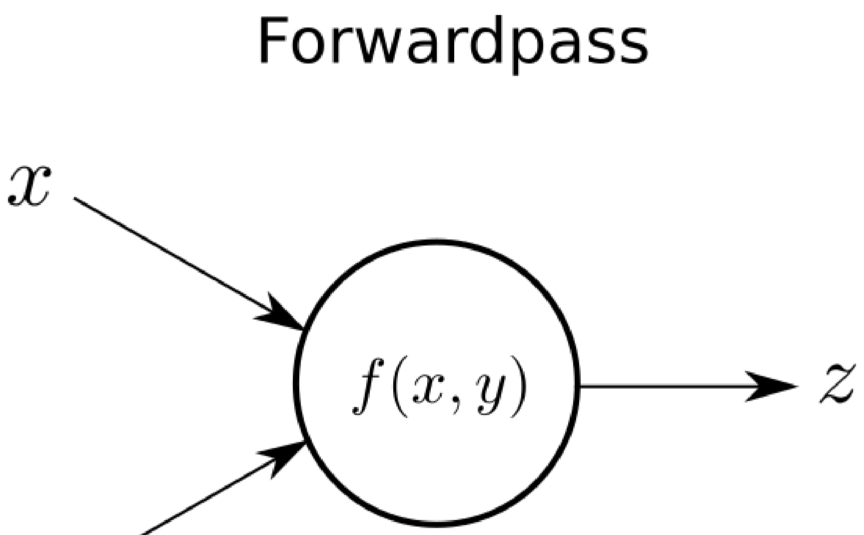


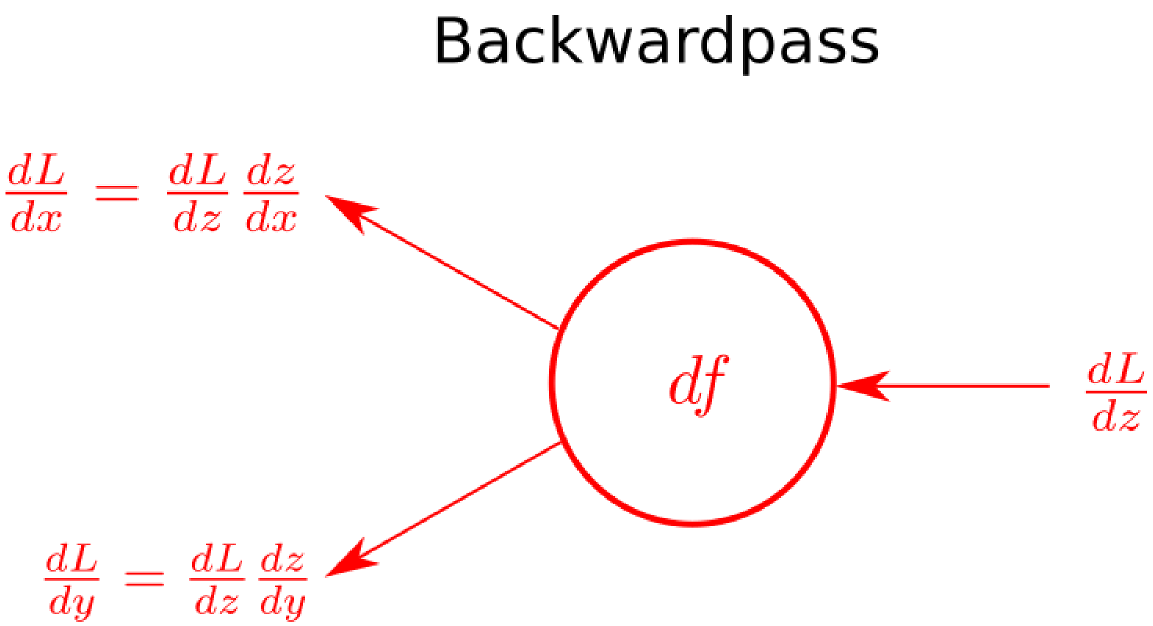
Stride 1 with Padding



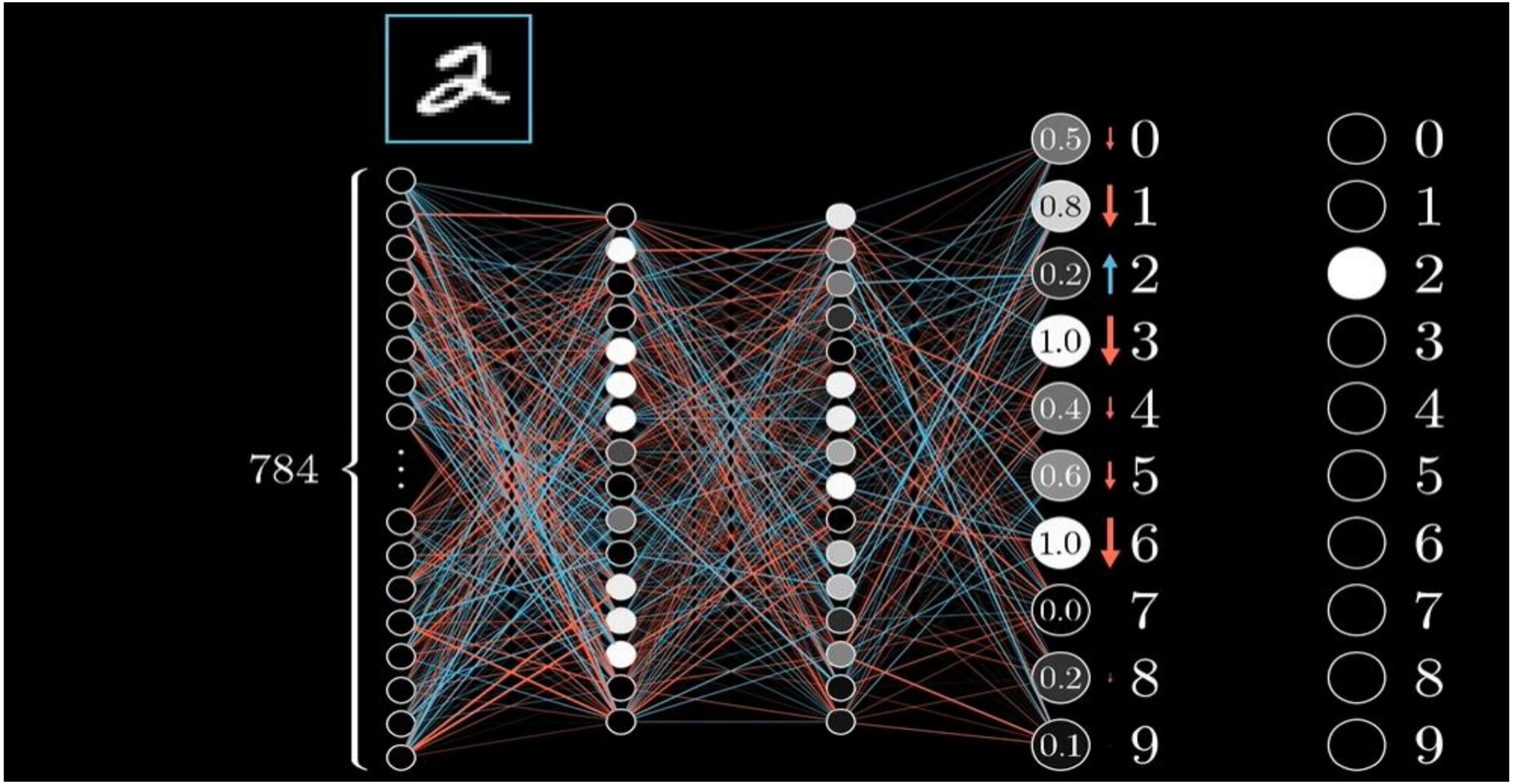
Feature Map

Convolution layer – Back Propogation

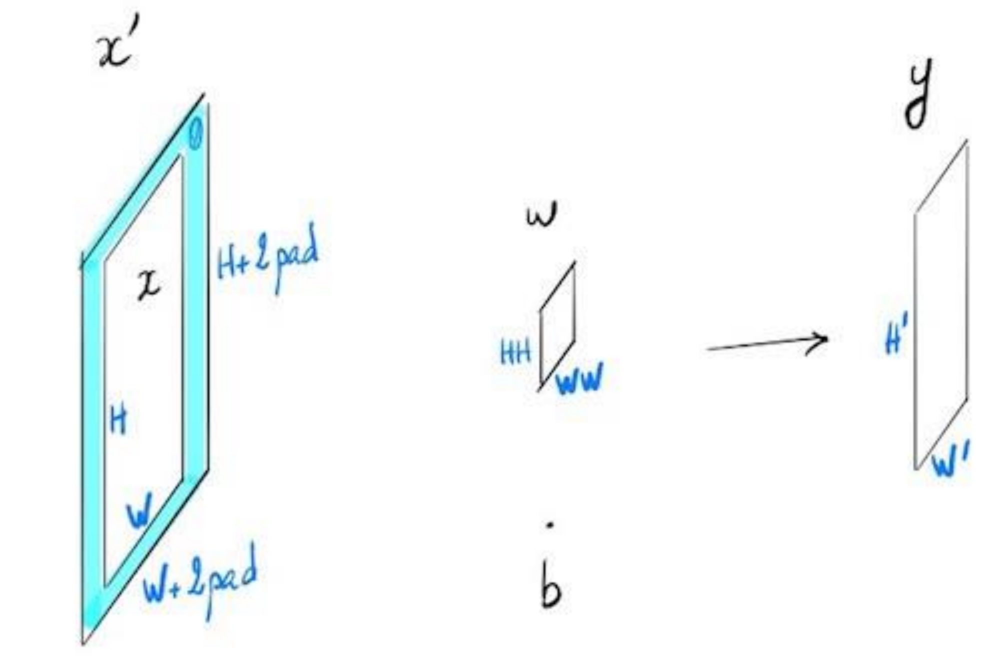




Convolution layer – Loss function



Convolution layer – Back Propogation



$$\forall (i,j) \in [1, H'] \times [1, W']$$

$$y_{ij} = \left(\sum_{k=1}^{HH} \sum_{l=1}^{WW} w_{kl} x'_{si+k-1, sj+l-1} \right) + b \quad (1)$$

Gradient of our cost function L with respect to y:

$$dy = \left(\frac{\partial L}{\partial y_{ij}} \right)$$

We are looking for

$$dx = \frac{\partial L}{\partial x}, dw = \frac{\partial L}{\partial w}, db = \frac{\partial L}{\partial b}$$

$$db = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial b} = dy \cdot \frac{\partial y}{\partial b}$$

Convolution layer – Back Propogation

$$dw = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w} = dy \cdot \frac{\partial y}{\partial w}$$

$$\frac{\partial y}{\partial w} = \begin{bmatrix} \frac{\partial y_1}{\partial w_1} & \frac{\partial y_1}{\partial w_2} \\ \frac{\partial y_2}{\partial w_1} & \frac{\partial y_2}{\partial w_2} \\ \frac{\partial y_3}{\partial w_1} & \frac{\partial y_3}{\partial w_2} \end{bmatrix}$$

$$dw_1 = x_1 dy_1 + x_2 dy_2 + x_3 dy_3$$
$$dw_2 = x_2 dy_1 + x_3 dy_2 + x_4 dy_3$$

$$dx = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial x} = dy^T \cdot \frac{\partial y}{\partial x}$$

$$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} & \frac{\partial y_1}{\partial x_4} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} & \frac{\partial y_2}{\partial x_4} \\ \frac{\partial y_3}{\partial x_1} & \frac{\partial y_3}{\partial x_2} & \frac{\partial y_3}{\partial x_3} & \frac{\partial y_3}{\partial x_4} \end{bmatrix}$$

$$\begin{aligned} dx_1 &= w_1 dy_1 \\ dx_2 &= w_2 dy_1 + w_1 dy_2 \\ dx_3 &= w_2 dy_2 + w_1 dy_3 \\ dx_4 &= w_2 dy_3 \end{aligned}$$

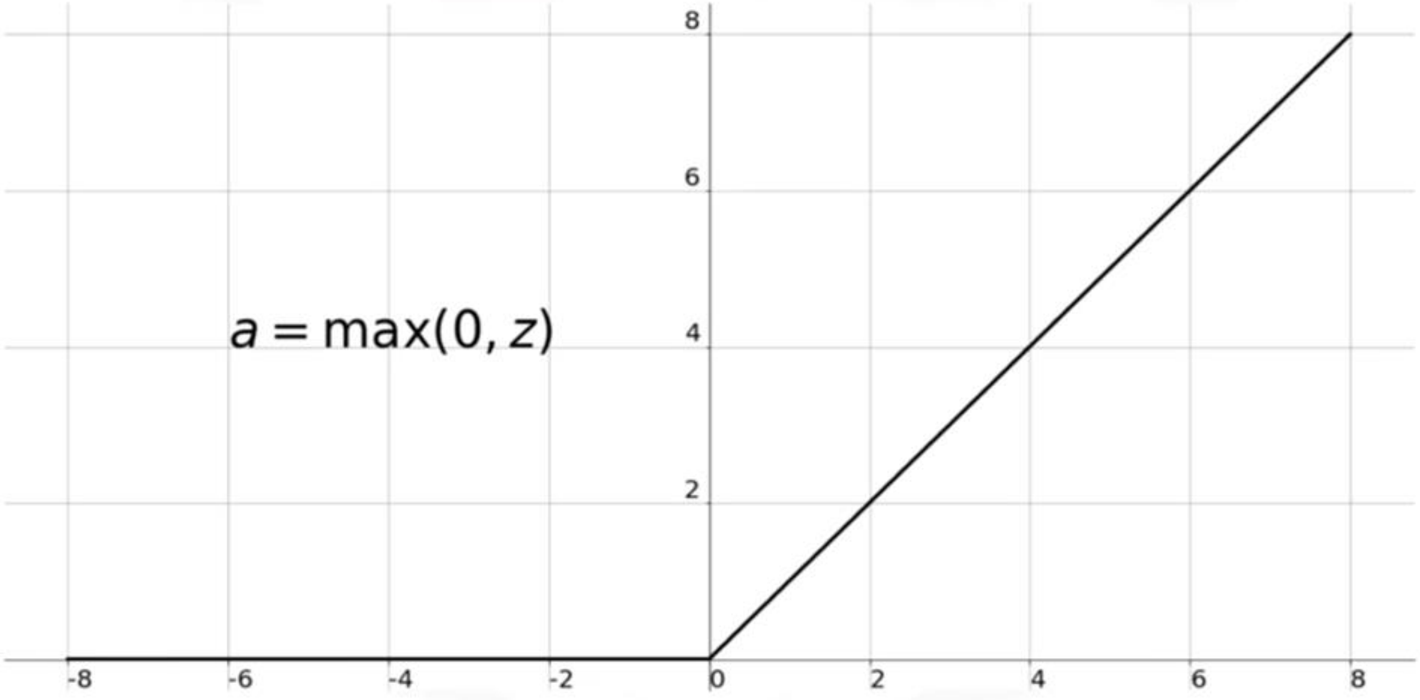
$$db = \sum_{i=1}^{3} \sum_{j=1}^{3} dy_{ij}$$

$$dw = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} * \begin{bmatrix} dy_{11} & dy_{12} & dy_{13} \\ dy_{21} & dy_{22} & dy_{23} \\ dy_{31} & dy_{32} & dy_{33} \end{bmatrix} = x * dy$$

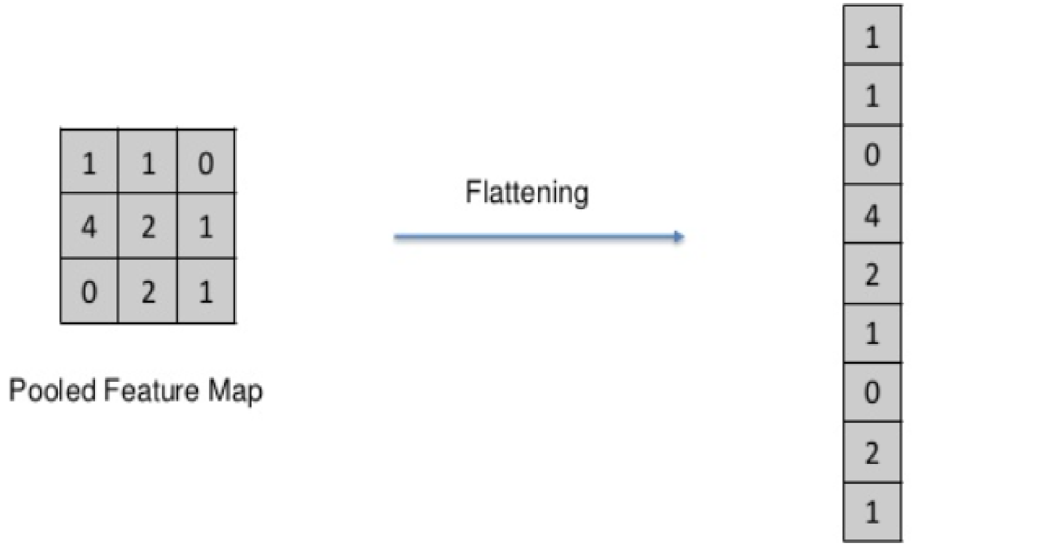
$$dx = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & dy_{11} & dy_{12} & dy_{13} & 0 \\ 0 & dy_{21} & dy_{22} & dy_{23} & 0 \\ 0 & dy_{31} & dy_{32} & dy_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{bmatrix} = dy\_0 * w'$$

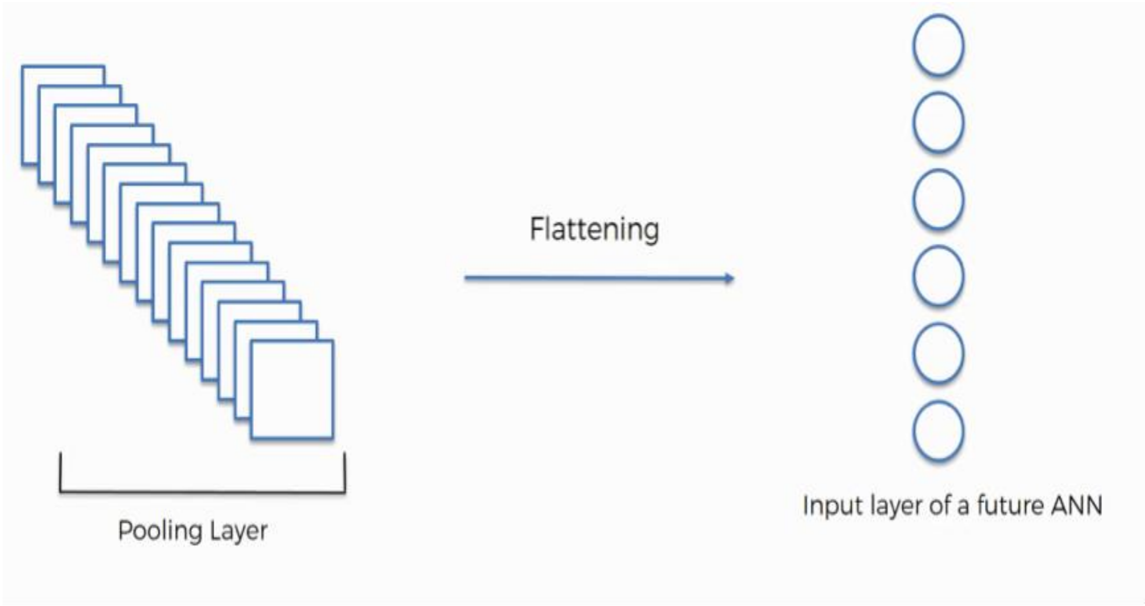
Convolution layer – Activation function

ReLU Function

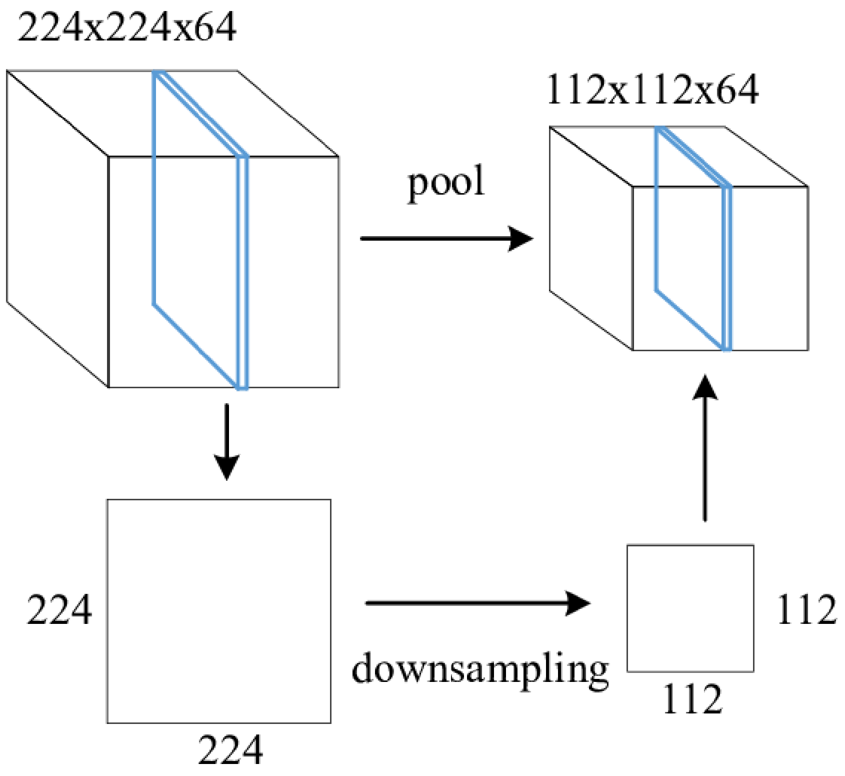


Convolution layer – Flattening





Pooling



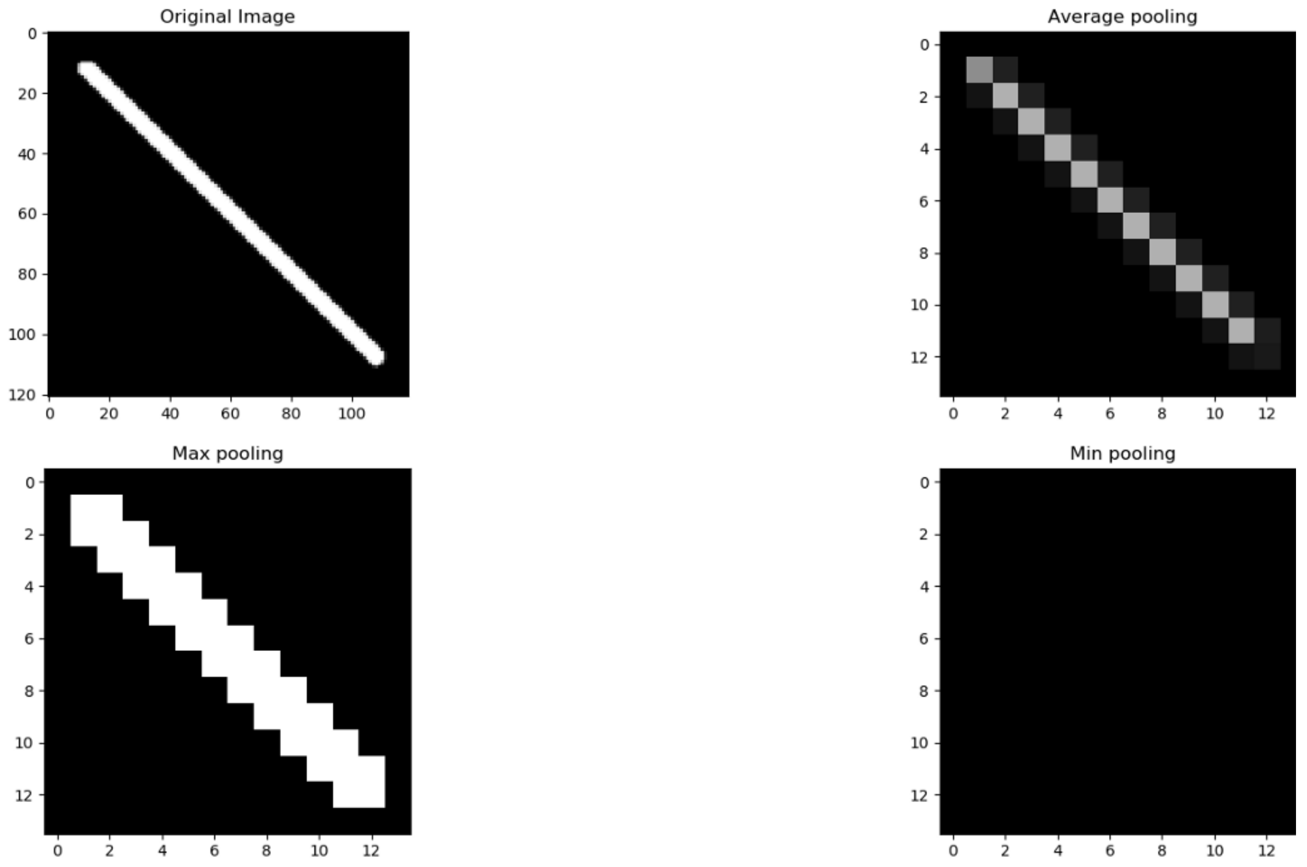
- to "accumulate" features from maps generated by convolving a filter over an image.
- to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network.
- it reduces the computational cost by reducing the number of parameters to learn

Pooling

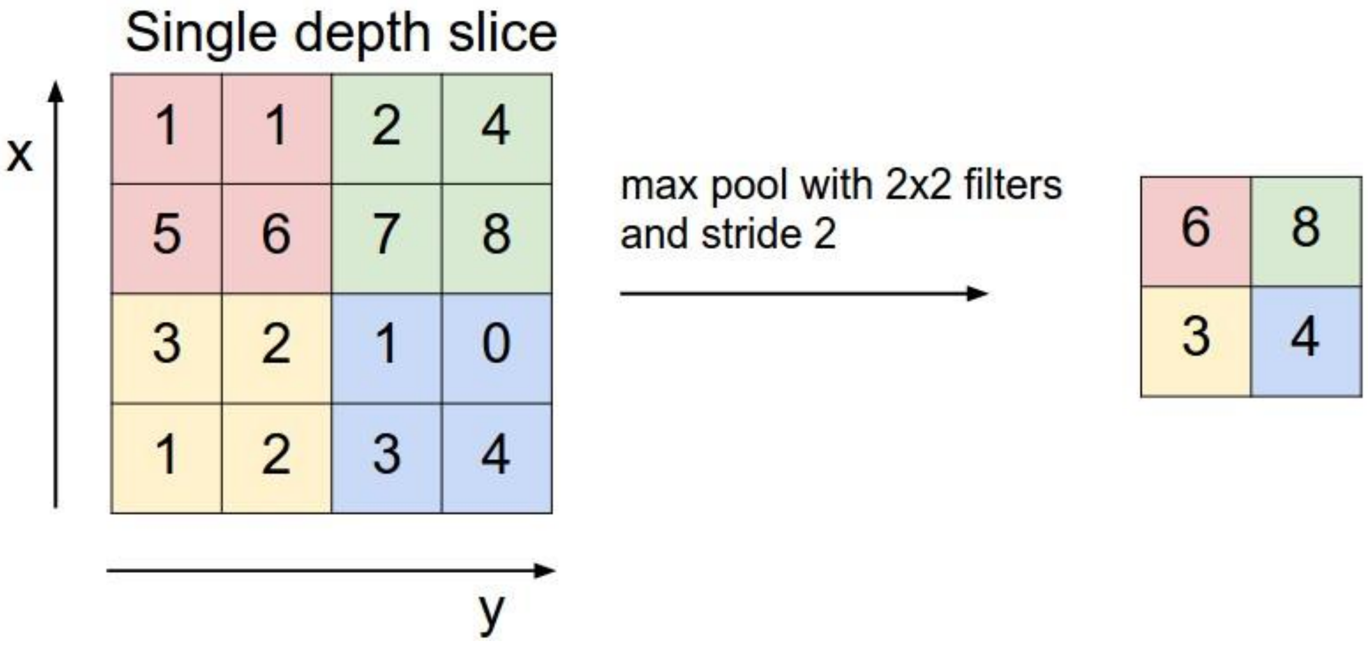
Max pooling selects the brighter pixels from the image. It is useful when the background of the image is dark and we are interested in only the lighter pixels of the image.

Similarly, **min pooling** is used in the other way round.

Average pooling method smooths out the image and hence the sharp features may not be identified when this **pooling** method is used.

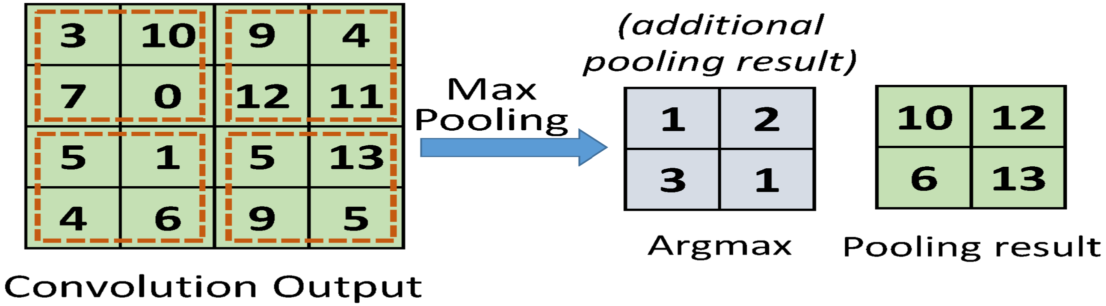


Max Pooling Forward Propagation

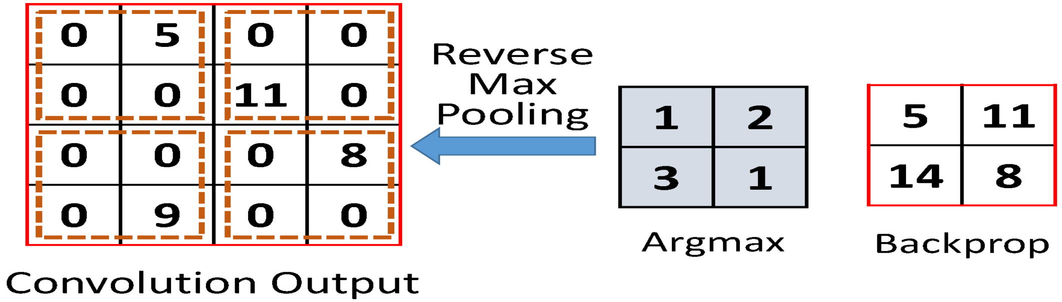


- Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map.

Max Pooling Backward Propagation



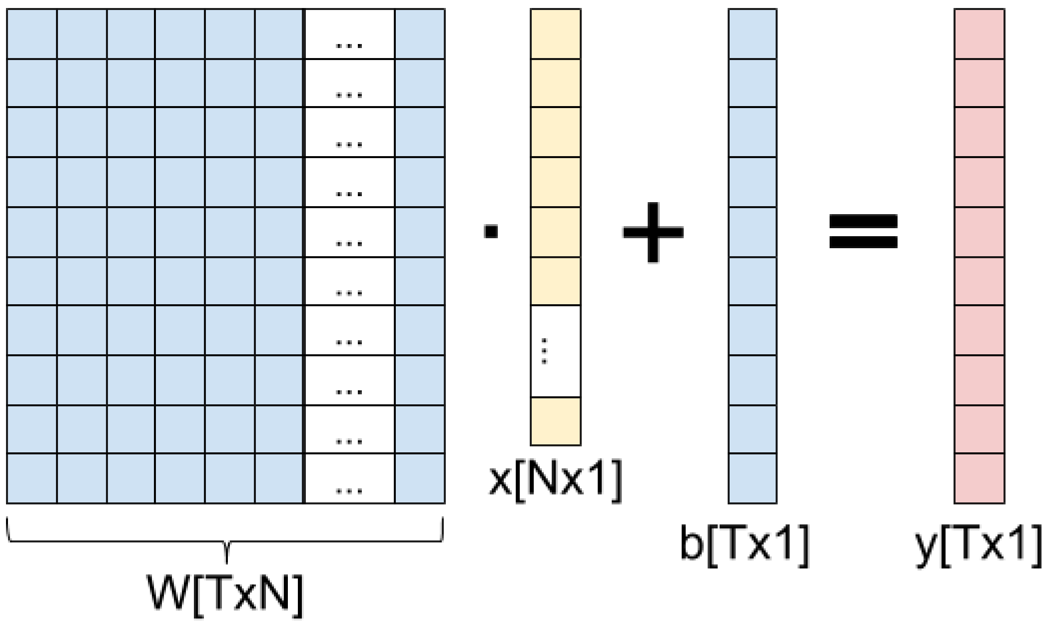
(a)



(b)

- To keep track of the "winning unit" its index noted during the forward pass and used for gradient routing during backpropagation.
- the error is just assigned to where it comes from - the "winning unit" because other units in the previous layer's pooling blocks did not contribute to it hence all the other assigned values of zero.

Fully Connected layer Forward Propagation

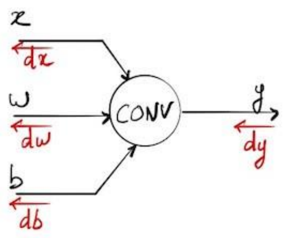


- After feature extraction we need to **classify the data into various classes**, this can be done using a fully connected (FC) neural network.

- Fully connected layers connect every neuron in one layer to every neuron in another layer.

Fully Connected layer
Backward Propagation

- We are calculating gradients: dw, dx, db
- Using these gradients, we are updating weights and bias.



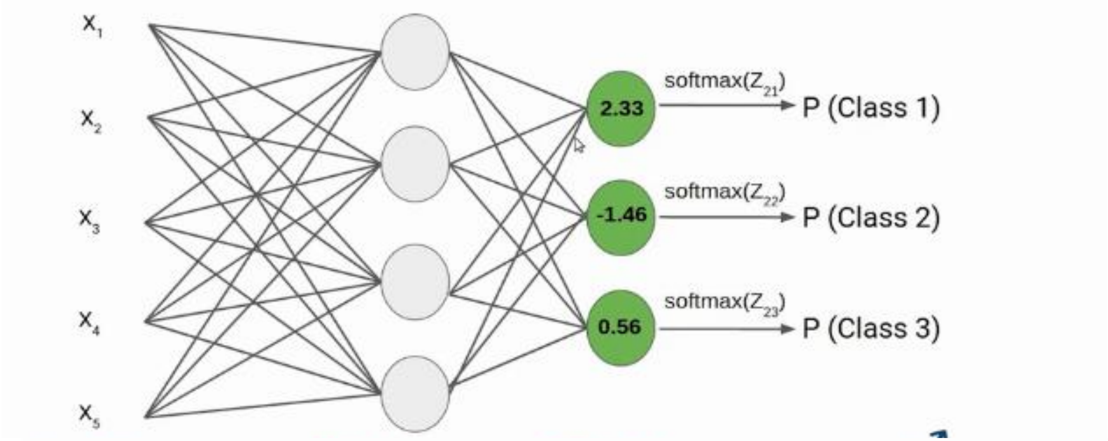
$$\frac{\partial L}{\partial x_1} = dout_{y1}.w11$$
$$\frac{\partial L}{\partial x_2} = dout_{y1}.w12$$
$$\frac{\partial L}{\partial x_3} = dout_{y1}.w13$$

$$\frac{\partial L}{\partial w_{11}} = dout_{y1}.x1$$
$$\frac{\partial L}{\partial w_{12}} = dout_{y1}.x2$$
$$\frac{\partial L}{\partial w_{13}} = dout_{y1}.x3$$

$$\frac{\partial L}{\partial b_1} = dout_{y1}$$

softmax

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$



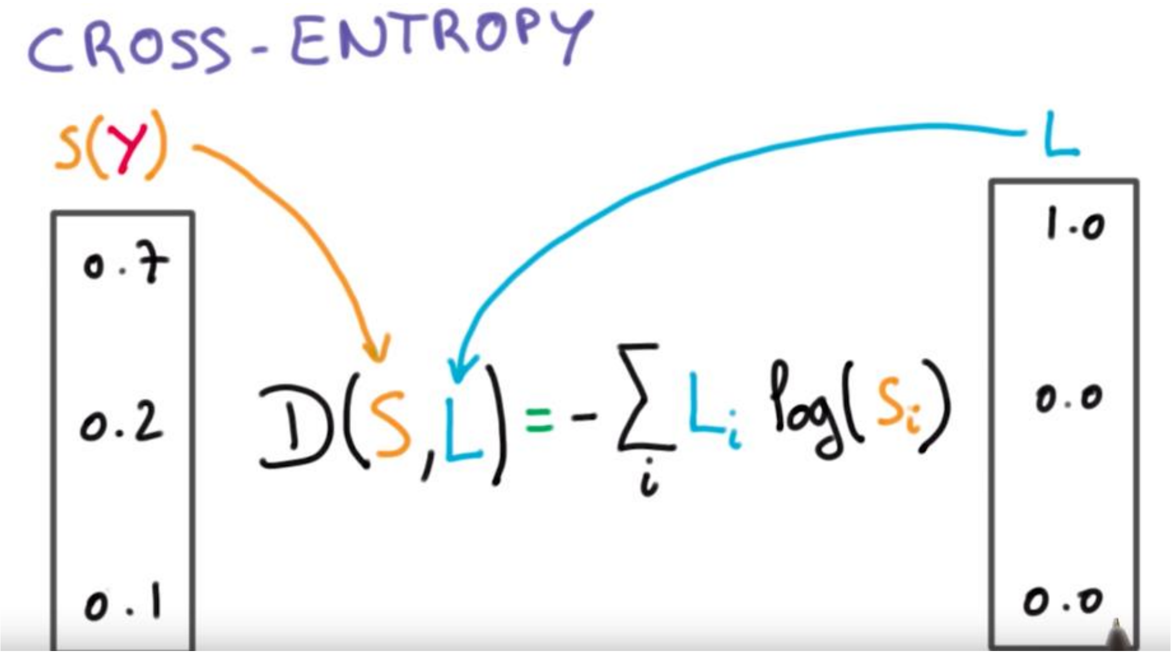
Example :

$$2.33 \rightarrow P\ (\text{Class } 1) = \frac{\exp(2.33)}{\exp(2.33) + \exp(-1.46) + \exp(0.56)} = 0.83827314$$

$$-1.46 \rightarrow P\ (\text{Class } 2) = \frac{\exp(-1.46)}{\exp(2.33) + \exp(-1.46) + \exp(0.56)} = 0.01894129$$

$$0.56 \rightarrow P\ (\text{Class } 3) = \frac{\exp(0.56)}{\exp(2.33) + \exp(-1.46) + \exp(0.56)} = 0.14278557$$

Loss function Cross Entropy



- Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1.
- predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value.

1,2,3 ...Let's get going...!

- Dataset details
- Feature Engineering
- Training and testing parameters
- Results
- Scope and plan for Optimization
- Drawbacks
- Conclusion

Dataset details

- MNIST – Numbered (0-9) images dataset
- Image characteristic: Gray scale**
- Total number of images: 60000
- Output classes: 10 (0-9)
- 6000 images of each class

** VGG16 is known to work great for RGB images as well

Feature Engineering

- **Zooming** into the images – 5 Max-Pooling layers.
  - 28x28 images to 64x64 images
- **Shuffle** the dataset (training images and training labels)
  - Make sure every batch will have all variety of images.
- **Normalization datasets**
  - Alter numeric columns of the dataset to a common scale, without distorting differences in the ranges of values or losing information so that the learning and prediction will not be biased.
- **One-Hot encode of labels**
  - 6 -> [0,0,0,0,0,0,1,0,0,0]

Training parameters and Model run

- model.train(X_train, y_train, batchSize, epochs, 'MagicNumbersWB.pkl')

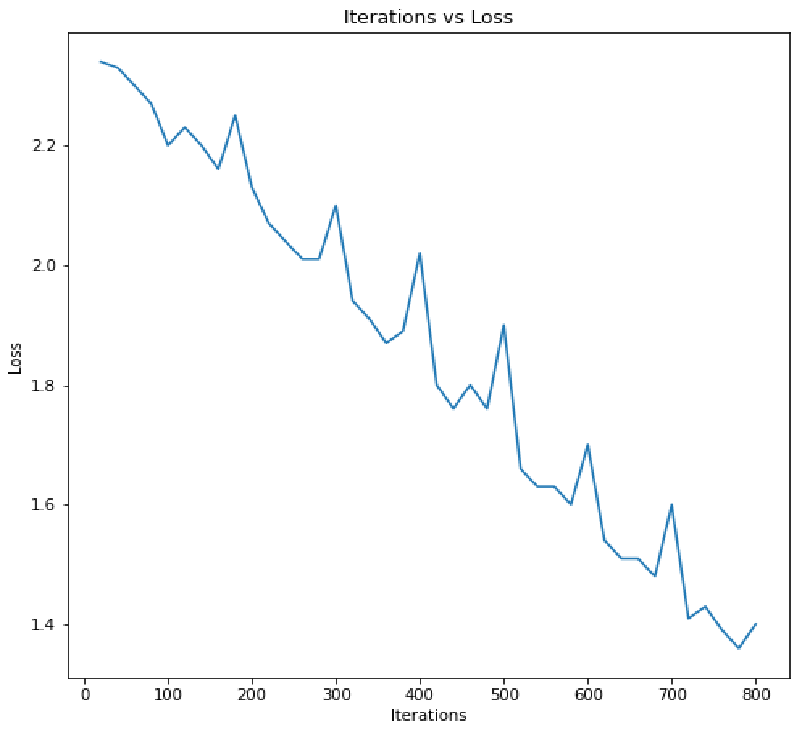
  - Train images dataset size : 100 (shuffled dataset containing all varieties of data)*
  - BatchSize : 20 **
  - Epochs: 8 **
  - MagicNumbersWB.pkl ??
  - Load calculated weights and biases to MagicNumbersWB.pkl.

- model.run(X_test, y_test, 'MagicNumbersWB.pkl')

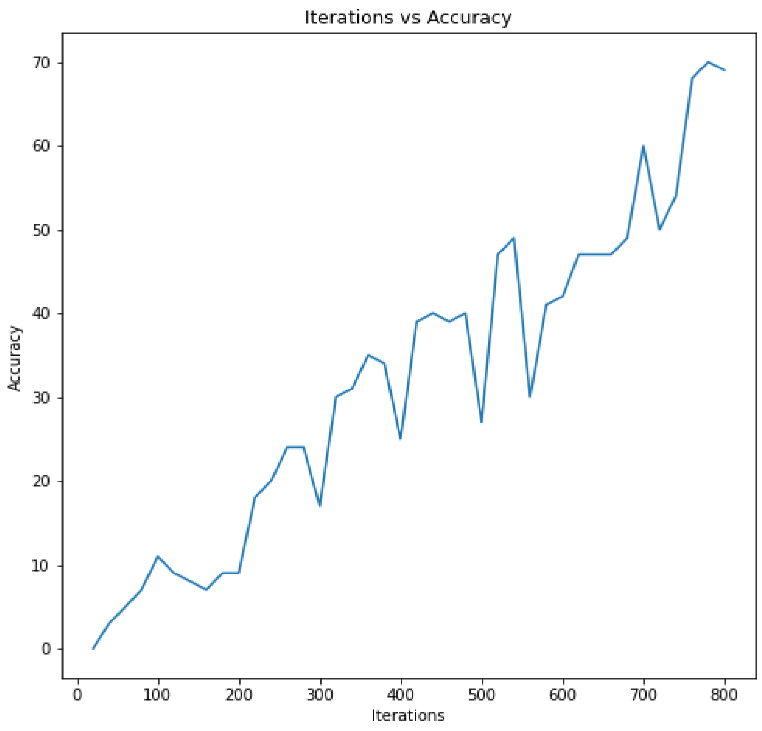
  - Load calculated weights and biases to all layers
  - How do we load weights and biases ??
    - wb = read from 'MagicNumbersWB.pkl' (pickle.load)
    - layers[x].loadWeights(wb[x]['layerX.weights'], wb[x][layerX.bias'])
  - Forward pass on all layers.
  - Compare output of softMax (prediction with label)

** small dataset size and epochs due to time constraints, will be running better dataset sizes and epochs next week

Results



Implemented VGG16 Net
Train accuracy : ~70%
Test accuracy : 60 %

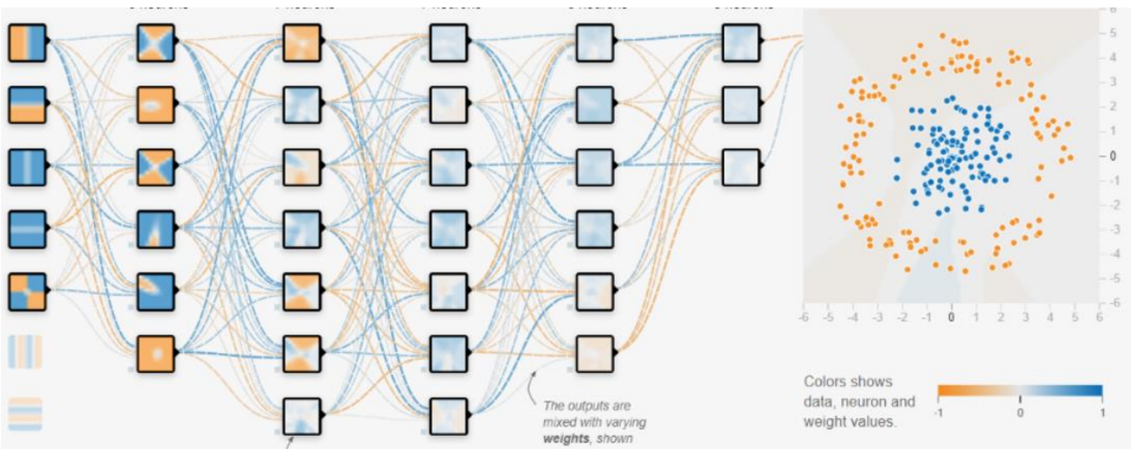


Scope for Optimization

- Drop out
- ADAM technique
- Data Augmentation

Drop Out

Neural Network



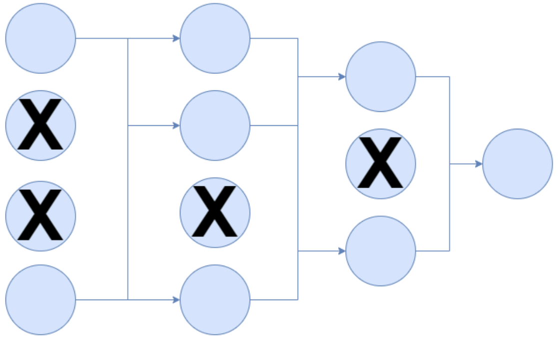
More Neurons $\propto$ ***Representaion capacity***

&

Deeper networks $\propto$ **Overfitting**

Drop out

Drop out networks $\propto$ **Accuracy**



- **Randomly drop interconnecting neurons withing the network**
- **0.5 probability for every neuron (to be dropped) guarantees a fresh network every run.**
- **Neural network with drop out technique, is possibly an average of all possible different neuron connection combinations.**

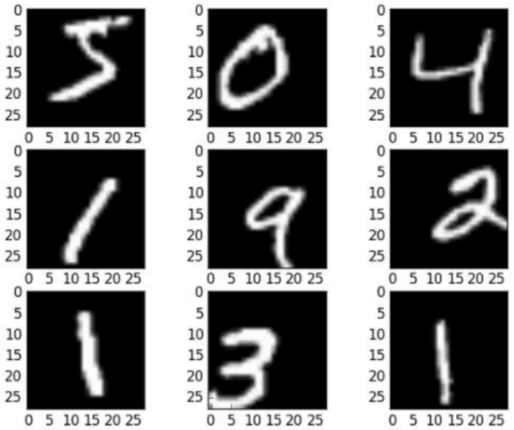
ADAM Optimization

- Adaptive Momentum Estimate
  - Regular gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training.
  - *ADAM computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients.*
- *ADAM is known to be a combination of :*
  - **Adaptive Gradient Algorithm** (AdaGrad) : per-parameter learning rate to improve performance on problems with sparse gradients (natural language and computer vision problems)
  - **Root Mean Square Propagation** (RMSProp) : per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight. This means the algorithm does well on online and other noisy patterns.

Source : https://machinelearningmastery.com/adam-optimization-from-scratch/

Data Augmentation

Random Shift



Random Flip

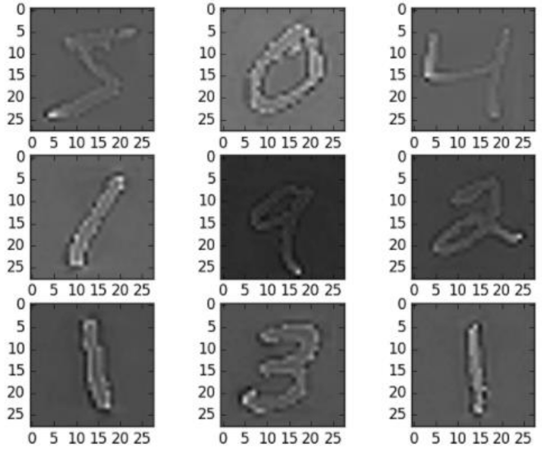
Data augmentation ??

Imagine all the things you could do in photoshop with a picture!

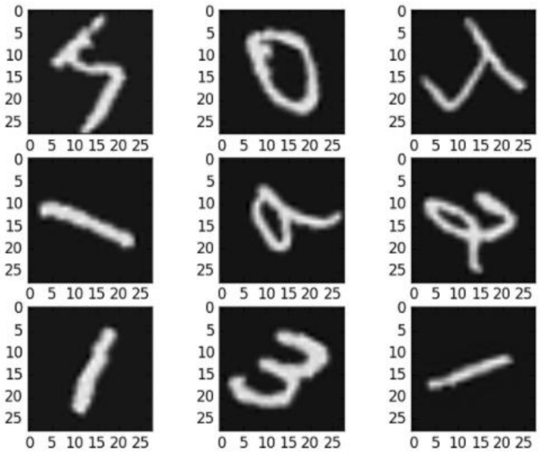
- Flipping (both vertically and horizontally)
- Rotating
- Zooming and scaling

and more...

Random Whitening



Random Rotation



```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

Drawbacks

Major drawbacks with VGGNet:

- It is *very slow* to train.
- The network architecture weights themselves are quite large (concerning disk/bandwidth).
- Due to its depth and number of fully-connected nodes, VGG16 is over 533MB, making the deployment a tiring task

Conclusion

- Implemented VGG16 Net
  - Train accuracy : 70 %
  - Test accuracy : 60 %

- Next steps:
  - Run the training for 10000 images and more epochs
  - Implement possible optimizations

Thank you !

Open to Questions