DEEP LEARNING FOR SIGNAL & IMAGE PROCESSING

Dr. Mithun Kumar Kar

School of Artificial Intelligence

Amrita Vishwa Vidyapeetham Coimbatore

What Is Deep Learning?

- Deep learning is another name for a multilayer artificial neural network or multilayer perceptron.
- By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity.
- Deep learning neural networks are able to automatically learn arbitrary complex mappings from inputs to outputs and support multiple inputs and outputs.

What Is Deep Learning?

- Deep Learning: a class of machine learning techniques, where many layers of information processing stages in hierarchical supervised architectures are exploited for unsupervised feature learning and for pattern analysis/classification.
- Deep learning is the name we often use for "deep neural networks" composed of several layers.

Definitions

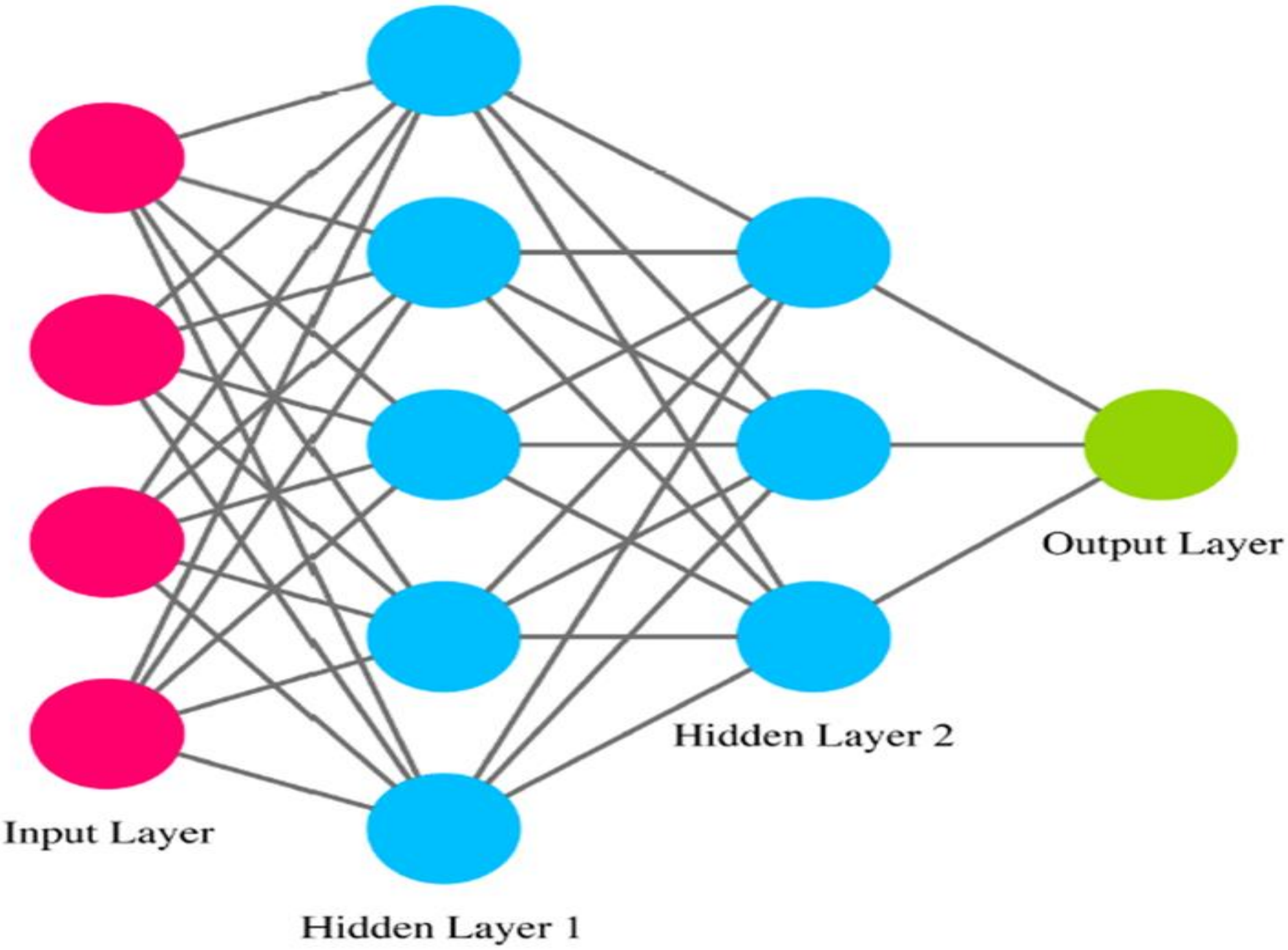
- Definition 1: A class of machine learning techniques that exploit many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification.

- Definition 2: A sub-field within machine learning that is based on algorithms for learning multiple levels of representation in order to model complex relationships among data.
- Higher-level features and concepts are thus defined in terms of lower-level ones, and such a hierarchy of features is called a deep architecture.

MLP

- A multilayer perceptron consists of at least three types of layers: input layer, hidden layers, and output layer.
- Input layer: The first layer of a neural network is called the input layer. This layer takes the input from the external source. The inputs to this layer are the features.
- Hidden layer: The layers of neurons between the input and output layers are called hidden layers.

MLP



MLP

- Output layer: The final layer of the neural network is the output layer. The output layer gets its input from the last hidden layer.
- For regression problems when the network has to predict a continuous value, such as the closing price of stocks, the output node has only one neuron.
- For classification problems when the network has to predict one of many classes, the output layer has as many neurons as the number of all possible classes.

Types of Deep Learning Network Models

- Convolutional Neural Networks (CNNs).
- Multilayer Perceptrons (MLP).
- Recurrent Neural Networks (RNNs).

Types of Computer Vision Problems

- Object detection
- Classification
- Regression
- Segmentation
- Prediction

Convolutional Neural Network (CNN)

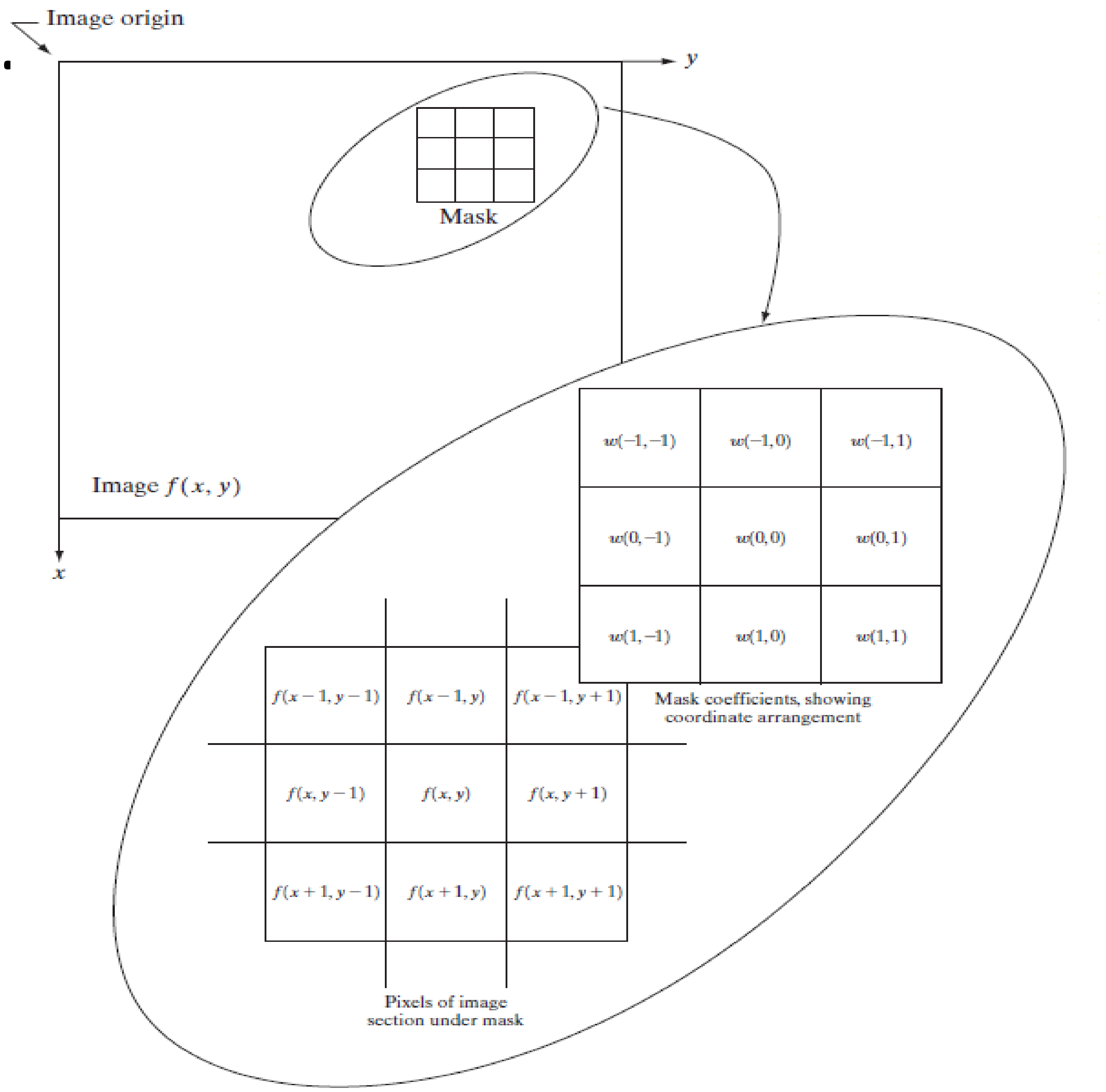
- Convolutional neural networks (CNN) are artificial neural network architectures consist of several convolutional layers that allow spatial convolution of input image with different filter kernels to compute various feature maps.
- In CNN, filters or kernels acting as weights slide across the total input image during convolution to create the next layer, and this new layer is called a feature map.
- The same filters can be used to repeat this operation to create new layers of feature maps

Convolutional Neural Network (CNN)

- The input and output feature maps have different dimensions depending on the dimension of image channels, i.e. 1 or 3 respectively for gray scale and color images.
- Multiple filters can be applied across a set of input image slices where each filter will generate a distinctive output slice.
- These slices highlight the features detected by the filters.
- At each layer, the input image is convolved with a set of kernels or masks with added biases to generate a new feature map.

Convolution in an Image

- **Spatial Filtering**.



Convolution in an Image

- In general, linear filtering of an image f of size M*N with a filter mask of size m*n is given by the expression:

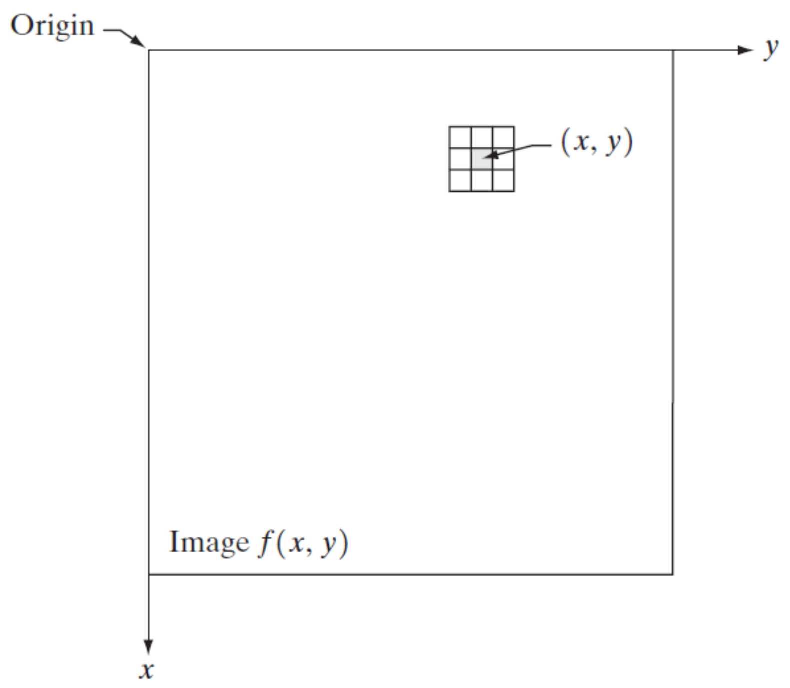
$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) f(x + s, y + t)$$

where, $a = (m - 1)/2$ and $b = (n - 1)/2$. To generate a complete filtered image this equation must be applied for $x = 0, 1\ 2, \ldots, M - 1$ and $y = 0, 1, 2, \ldots, N - 1$.

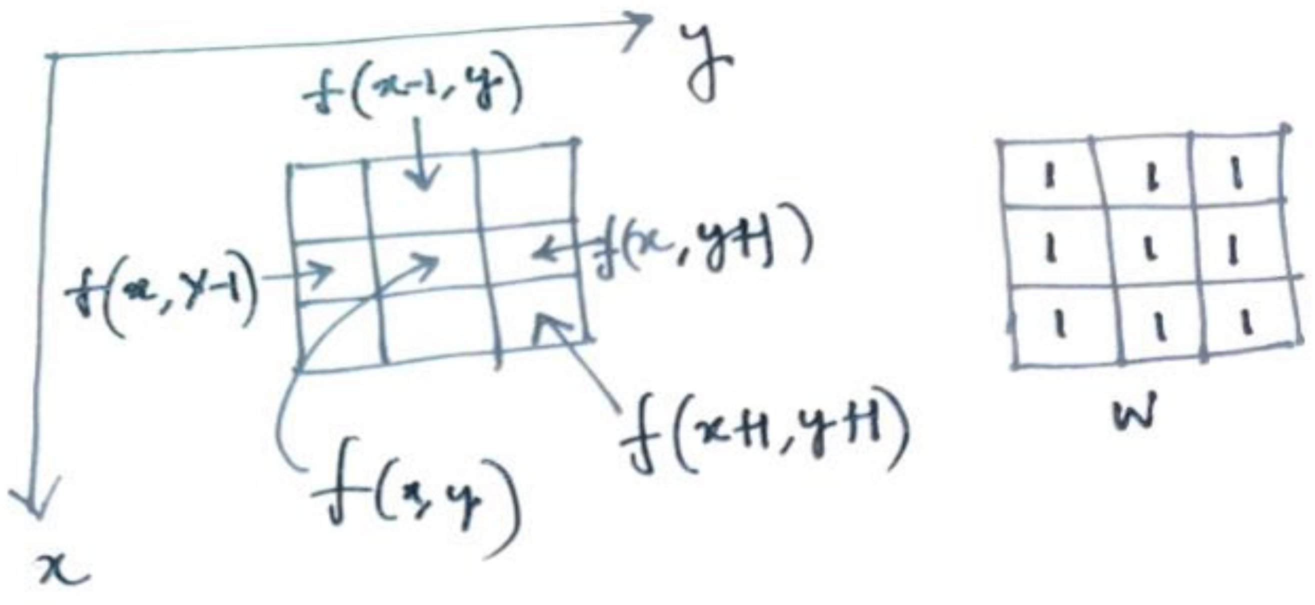
Smoothing Spatial Filters

- Two examples of averaging filters.

$$\frac{1}{9} \times \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \qquad \frac{1}{16} \times \begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$



Smoothing Spatial Filters



$$\text{Averaging} = P(x,y) = f(x,y) + f(x-1,y-1) + f(x-1,y) + f(x-1,y+1) + \cdots + f(x+1,y+1)$$

$P = fW$ in matrix form.

Sharpening Spatial Filters

- The derivatives of a digital function are defined in terms of differences.
- A basic definition of the first-order derivative of a one-dimensional function f(x) is the difference

$$\frac{\partial f}{\partial x} = f(x+1) - f(x).$$

- Similarly, we define a second-order derivative as the difference

$$\frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x).$$

Sharpening Spatial Filters

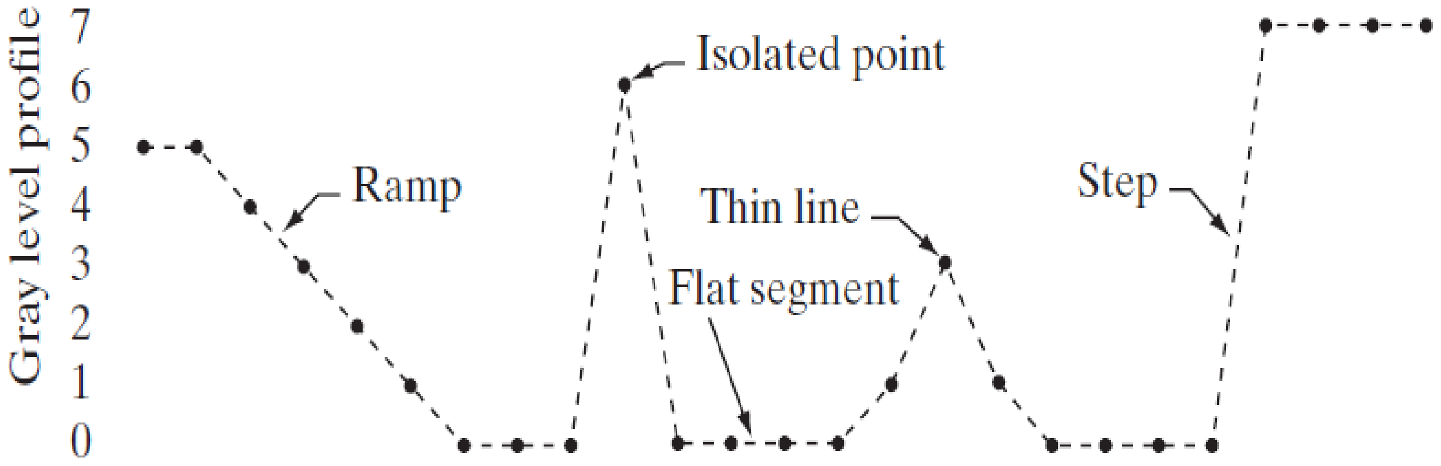


Image strip	5	5	4	3	2	1	0	0	0	6	0	0	0	0	1	3	1	0	0	0	0	7	7	7	7	•	•
First Derivative		−1	−1	−1	−1	−1	0	0	6	−6	0	0	0	1	2	−2	−1	0	0	0	7	0	0	0			
Second Derivative		−1	0	0	0	0	1	0	6	−12	6	0	0	1	1	−4	1	1	0	0	7	−7	0	0			

Sharpening Spatial Filters

- It can be shown that the simplest second derivative operator is the **Laplacian**, which, for a function (image) f(x, y) of two variables, is

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

- Now $$\frac{\partial^2 f}{\partial^2 x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

$$\frac{\partial^2 f}{\partial^2 y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$$

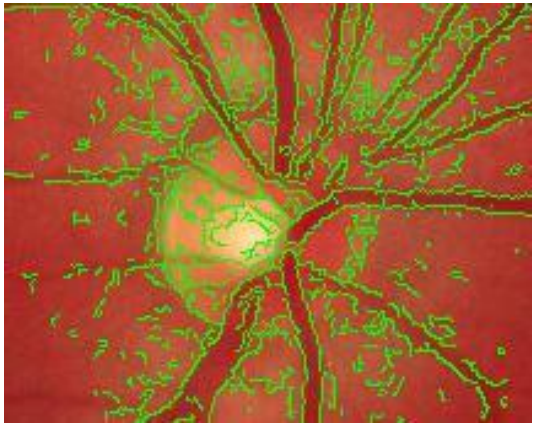
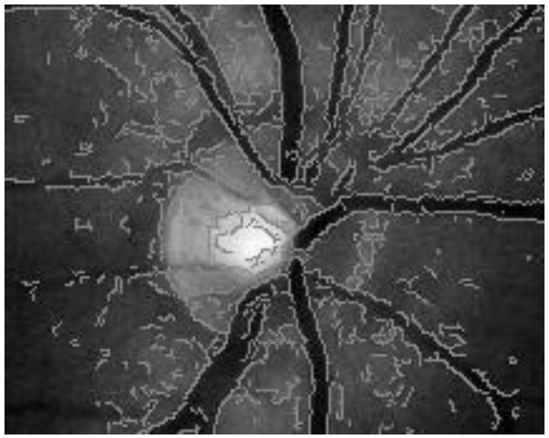
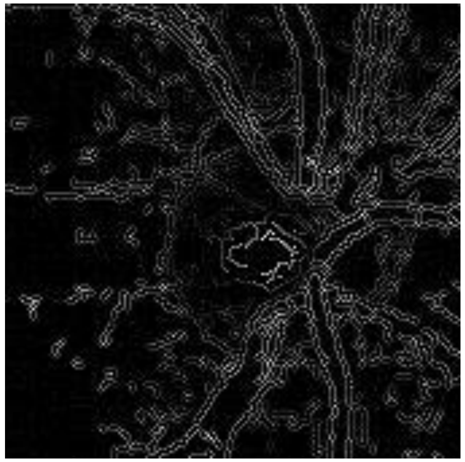
$$\nabla^2 f = [f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1)] - 4f(x, y).$$

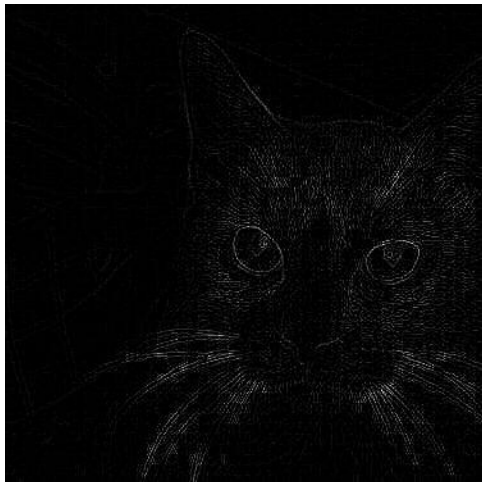
Sharpening Spatial Filters

- Now if we multiply this kernel with f(x,y) we got the same result. This equation can be implemented using the mask shown below.

0	1	0
1	–4	1
0	1	0

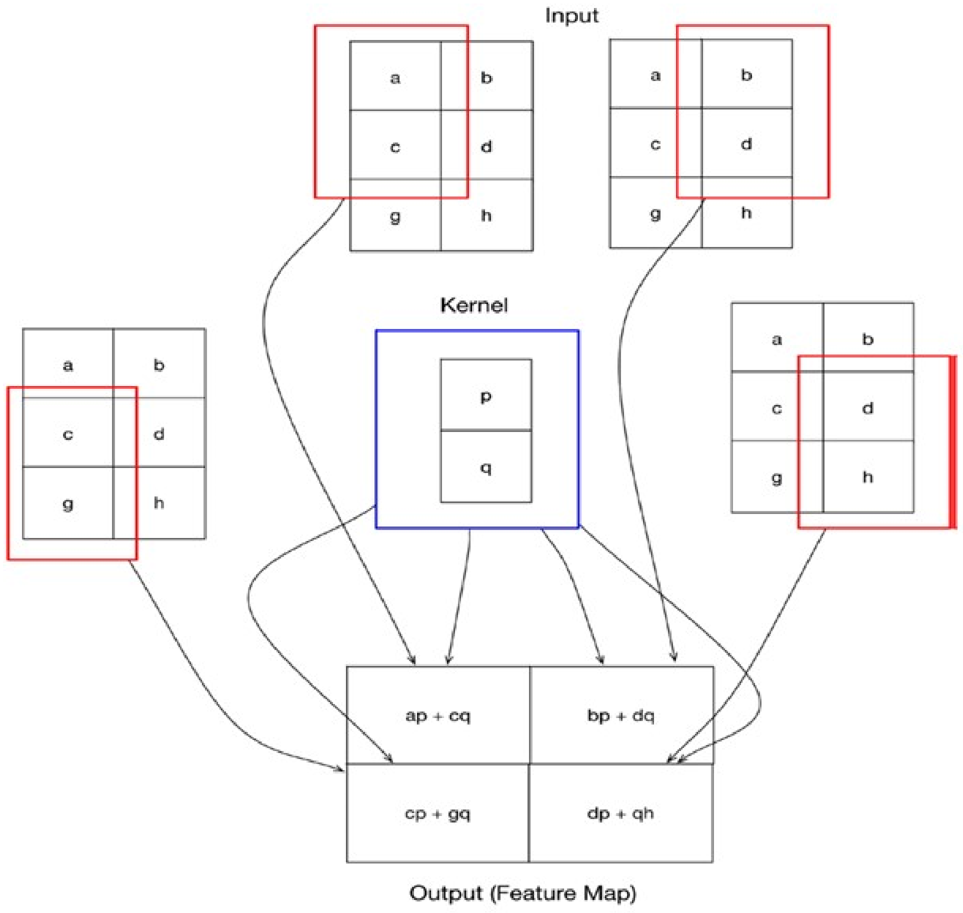
Implementation



CNN

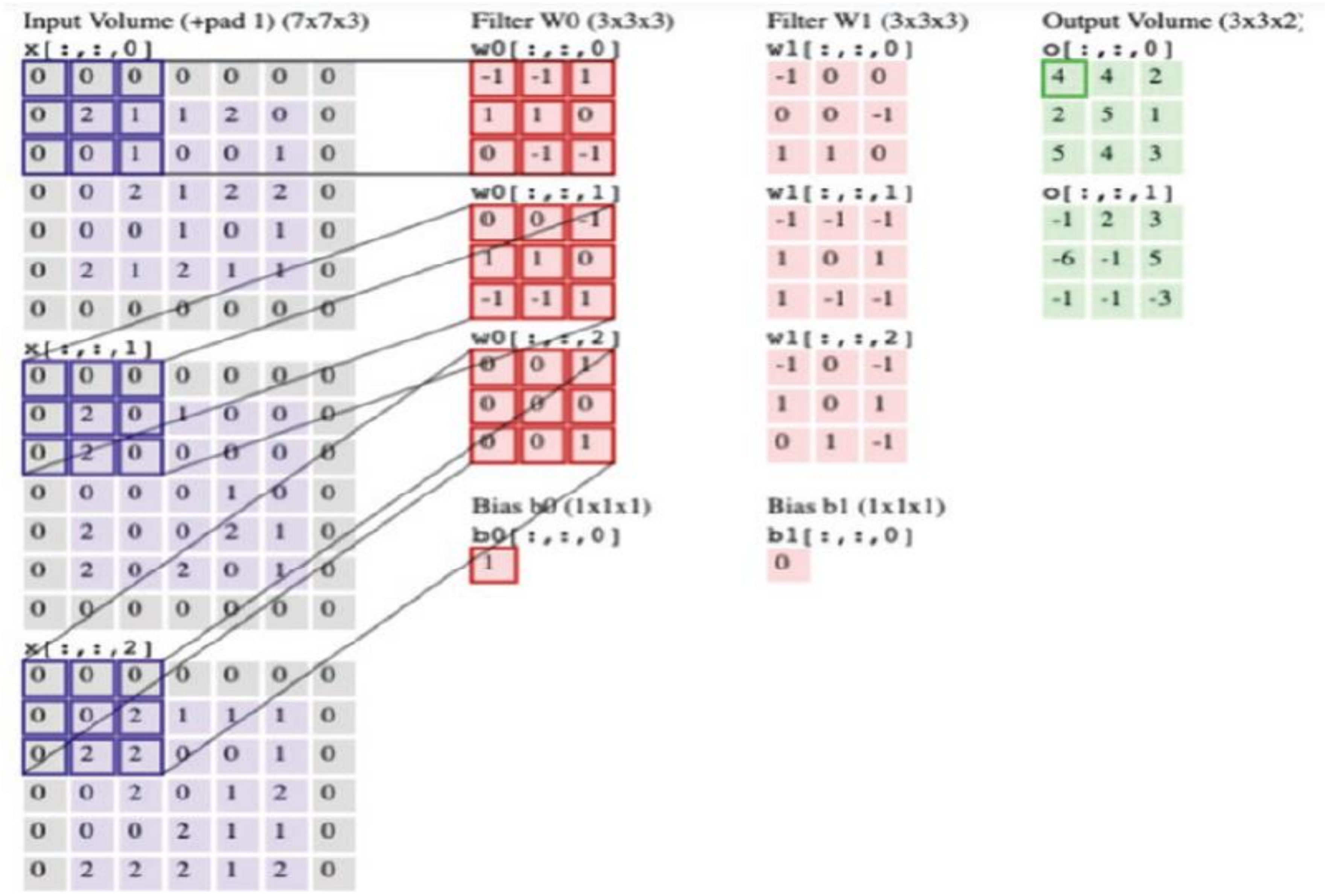
- The shape of the image after convolution is changed wrt the size of kernel used, shifting parameters and pooling.
- **General operations:**
- Convolution layer
- Pooling layer
- Batch normalization layer
- Fully connected layer



Components of Convolution Neural Networks

- Input layer
- Convolution layer
- Activation function for CNNs
- Pooling layer
- Batch normalization layer
- Fully connected layers

Convolutional Neural Network (CNN) Layer



Convolutional Neural Network (CNN) Layer

- **weight connections**: Each input feature to a neuron is multiplied by a weight.
- Think of weight as the contribution or significance of an input feature.
- The higher the weight, the more the contribution of the feature.
- The training objective of a neural network is to calculate the most optimized weights for each input feature for each connection to neurons of each layer

Convolutional Neural Network (CNN) Layer

- In a CNN, only the size of the filters is specified; the weights are initialized to arbitrary values before the start of training. The weights of the filters are learned through the CNN training process.
- Some of the terms with which one should be familiar while defining the convolution layer
- **Filter size**
- **Stride:** The stride determines the number of pixels to move in each spatial direction while performing convolution.

Convolutional Neural Network (CNN) Layer

- **Padding:** Padding is an approach that appends zeroes to the boundary of an image to control the size of the output of convolution.
- The convolved output image length L′ along a specific spatial dimension is given by

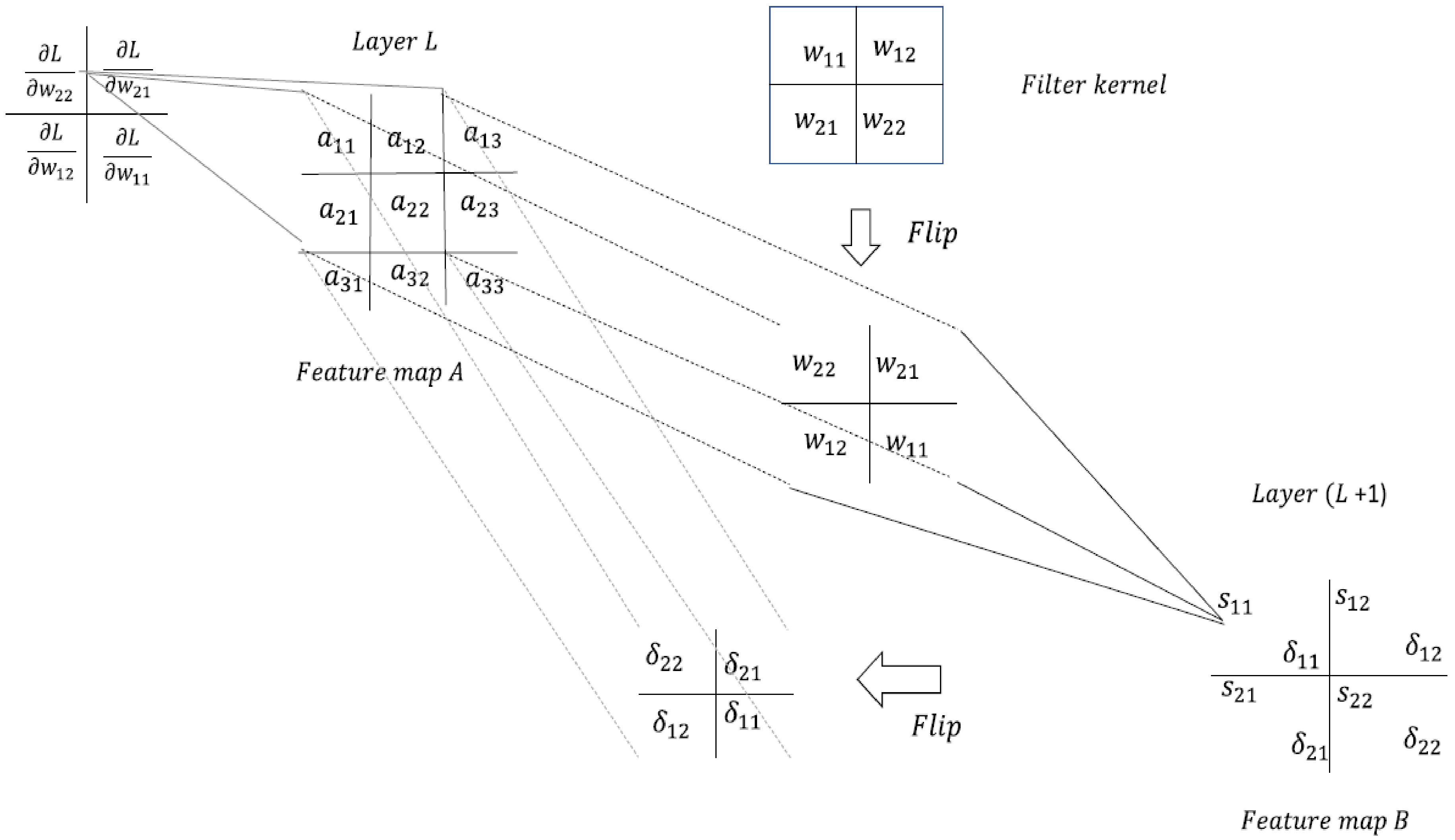
$$L' = \frac{L - K + 2P}{S} + 1$$

- Where L is the length of the input image in a specific dimension.
- K is the length of the kernel/filter in a specific dimension.

Convolutional Neural Network (CNN) Layer

- P-- Zeroes padded along a dimension in either end
- S-- Stride of the convolution
- **In pytorch this is done by**
- CLASS **torch.nn.Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*, *device=None*, *dtype=None*)

Backpropagation Through the Convolutional Layer



$\frac{\partial L}{\partial w_{22}}$
$\frac{\partial L}{\partial w_{21}}$
$\frac{\partial L}{\partial w_{12}}$
$\frac{\partial L}{\partial w_{11}}$
Layer L
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Feature map A
w_{11} w_{12}
w_{21} w_{22}
Filter kernel
Flip
w_{22} w_{21}
w_{12} w_{11}
Layer (L +1)
s_{11} s_{12}
δ_{11} δ_{12}
s_{21} s_{22}
δ_{21} δ_{22}
Feature map B
Flip
δ_{22} δ_{21}
δ_{12} δ_{11}

Backpropagation Through the Convolutional Layer

- In generalized way

$$s_{ij} = \sum_{n=1}^{2}\sum_{m=1}^{2} w_{(3-m)(3-n)} * a_{(i-1+m)(j-1+n)}$$

- Now, let the gradient of the cost function *L* with respect to the net input *s*ij be denoted by

$$\frac{\partial L}{\partial s_{ij}} = \delta_{ij}$$

- Let’s compute the gradient of the cost function with respect to the weight w22

Backpropagation through the Convolutional Layer

- .

$$\frac{\partial \mathrm{L}}{\partial w_{22}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \frac{\partial \mathrm{L}}{\partial s_{ij}} \frac{\partial s_{ij}}{\partial w_{22}} = \sum_{j=1}^{2}\sum_{i=1}^{2} \delta_{ij} \frac{\partial s_{ij}}{\partial w_{22}}$$

$$\frac{\partial s_{11}}{\partial w_{22}} = a_{11}, \ \frac{\partial s_{12}}{\partial w_{22}} = a_{12}, \ \frac{\partial s_{13}}{\partial w_{22}} = a_{21}, \ \frac{\partial s_{14}}{\partial w_{22}} = a_{22}$$

$$\frac{\partial \mathrm{L}}{\partial w_{22}} = \delta_{11} * a_{11} + \delta_{12} * a_{12} + \delta_{21} * a_{21} + \delta_{22} * a_{22}$$

$$\frac{\partial \mathrm{L}}{\partial w_{21}} = \delta_{11} * a_{12} + \delta_{12} * a_{13} + \delta_{21} * a_{22} + \delta_{22} * a_{23}$$

Backpropagation through the Convolutional Layer

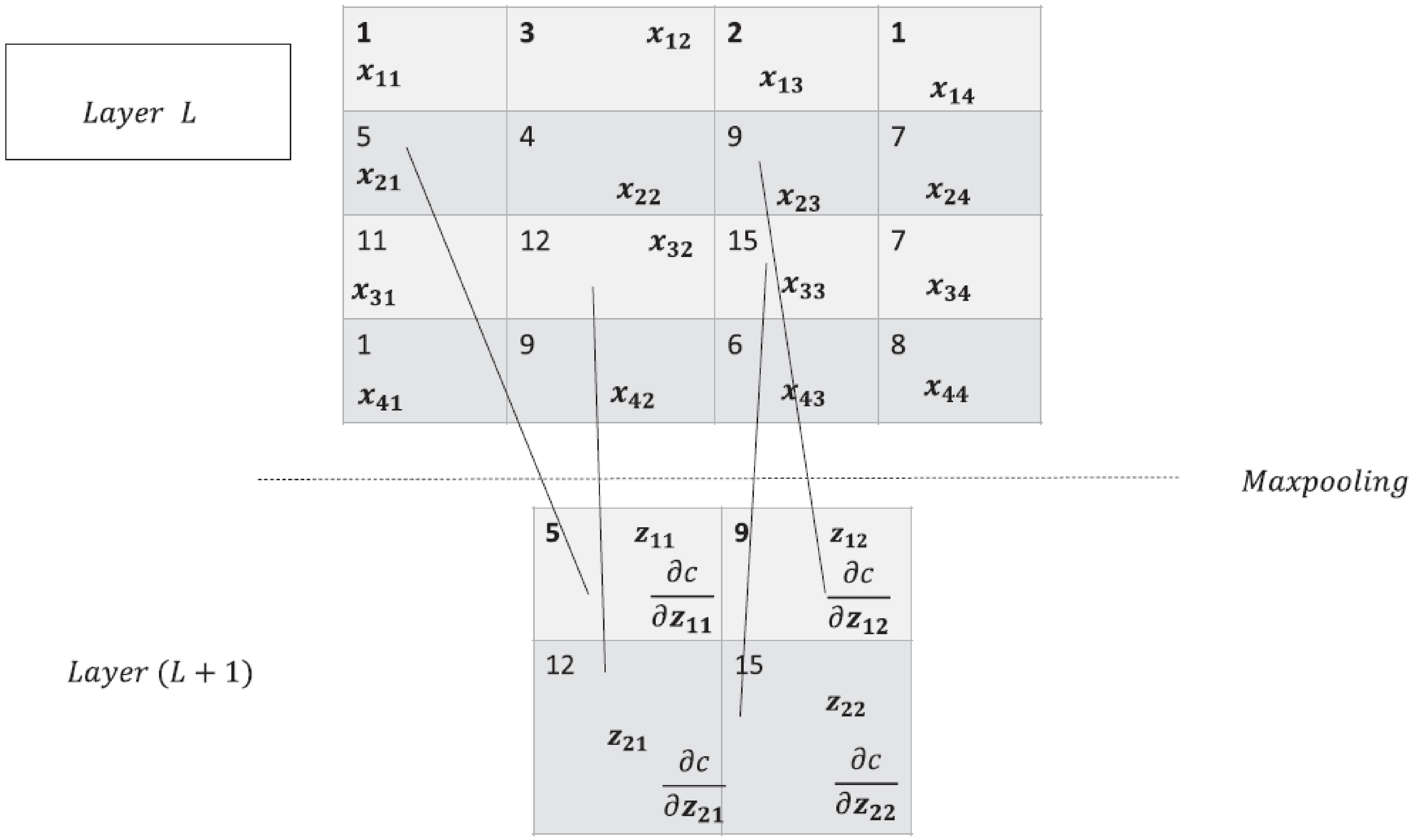
- Generalizing we get

$$\frac{\partial L}{\partial w_{ij}} = \sum_{n=1}^{2}\sum_{m=1}^{2}\delta_{mn} * a_{(i-1+m)(j-1+n)}$$

$$\begin{bmatrix} \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{21}} \\ \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{11}} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} (\mathrm{x}) \begin{bmatrix} \delta_{11} & \delta_{12} \\ \delta_{21} & \delta_{22} \end{bmatrix}$$

- In terms of the layers, one can say the flip of the gradient matrix turns out to be a cross-correlation of the gradient at the (L +1) layer with the outputs of the feature map at layer L.

Backpropagation Through the Pooling Layers



Average pooling

Layer L

1 x_{11}	**3** x_{12}	**2** x_{13}	**1** x_{14}
5 x_{21}	4 x_{22}	9 x_{23}	7 x_{24}
11 x_{31}	12 x_{32}	15 x_{33}	7 x_{34}
1 x_{41}	9 x_{42}	6 x_{43}	8 x_{44}

Average *pooling*

Layer $(L+1)$

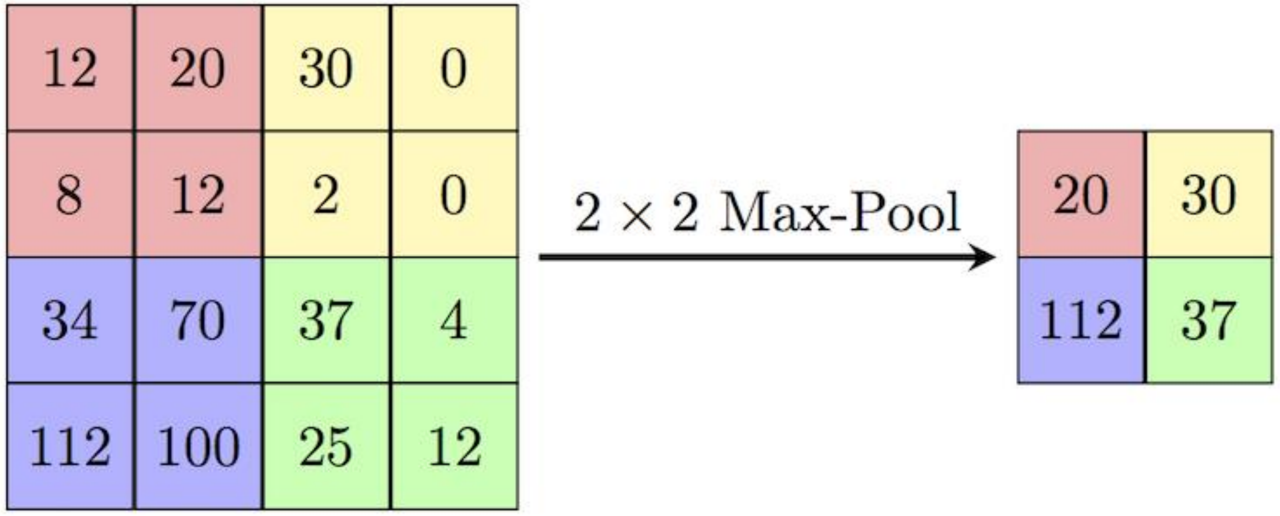
3.25 z_{11} $\frac{\partial c}{\partial z_{11}}$	**4.75** z_{12} $\frac{\partial c}{\partial z_{12}}$
8.25 z_{21} $\frac{\partial c}{\partial z_{21}}$	**9** z_{22} $\frac{\partial c}{\partial z_{22}}$

Activation function for CNNs

- Linear Activation Function
- Rectified Linear Unit
- Leaky ReLU
- Sigmoid
- Tanh
- Softmax

Pooling layer

- **Max Pooling** is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a down sampled (pooled) feature map. It is usually used after a convolutional layer.



Batch normalization layer

- **Batch normalization** is used to reduce internal covariate shift and for faster convergence.
- The batch normalized activation is given

$$a_i = \frac{a_i - a_{mean}}{\sqrt{\sigma_b^2 + c}}$$

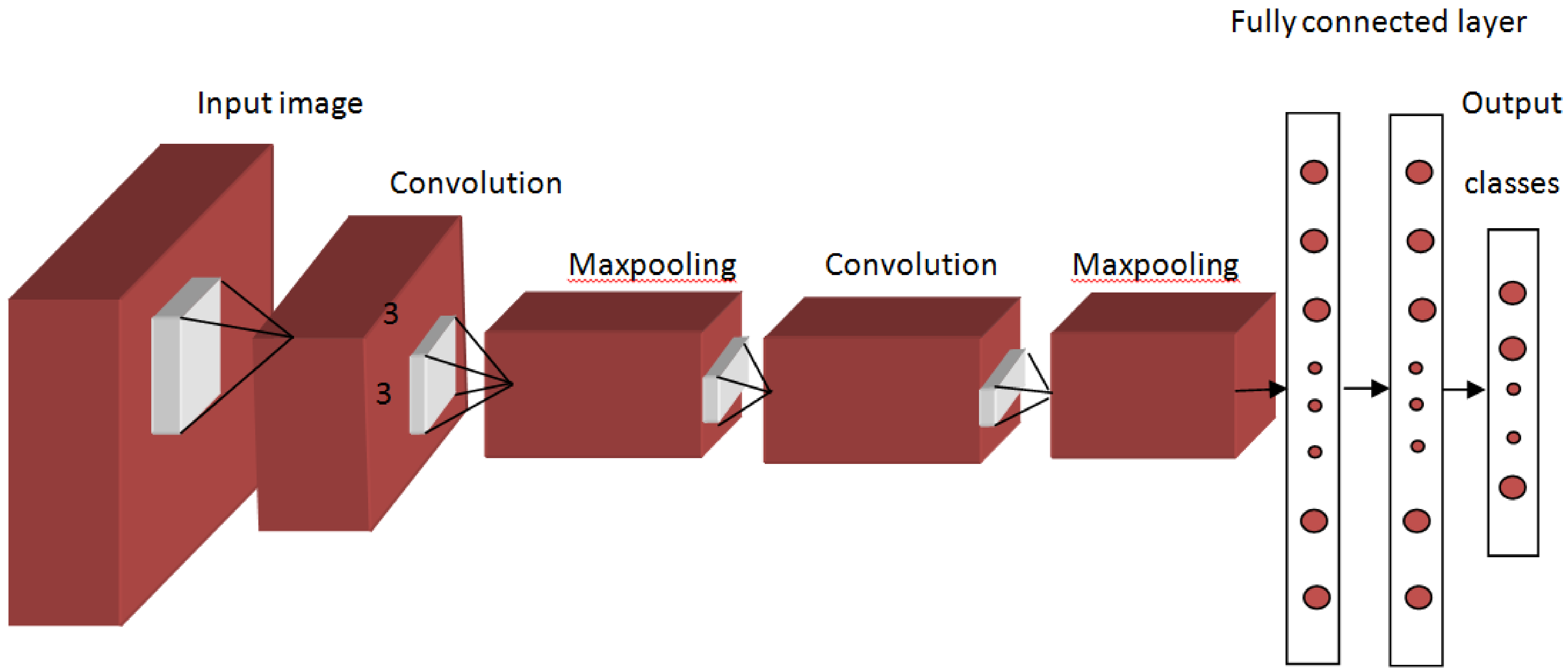
- where a_{mean} represents mini batch mean, σ_b^2 is the mini batch variance and c is a numeric constant used for numerical stability.

Fully connected layer

- **Fully connected layer**: Here each neuron in one layer is connected to other neurons in the subsequent layer.
- The fully connected layer integrates the various features extracted in the previous convolutional and pooling layers and maps them to specific classes or outcomes.
- Limiting the number of fully connected layers balances computational efficiency and generalization ability with the capability to learn complex patterns.

CNN

- CNN as classifier: CNN is used to classify the image data in to different classes.

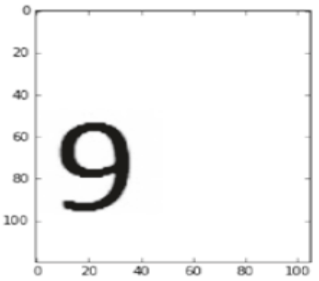


Weight Sharing Through Convolution and Its Advantages

- Weight sharing through convolution greatly reduces the number of parameters in the convolutional neural network.
- In cases of convolution, as in this scenario, we just need to learn the weights for the specific filter kernel. Since the filter size is relatively small with respect to the image, the number of weights is reduced significantly.
- The convolution operation provides translational equivariance.

Weight Sharing Through Convolution and Its Advantages

- if a feature A in an input produces a specific feature B in the output, then even if feature A is translated around in the image, feature B would continue to be generated at different locations of the output.



$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

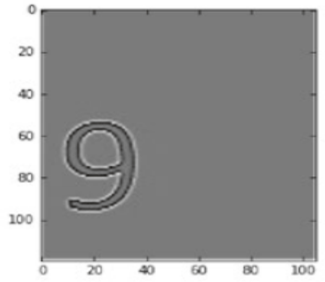
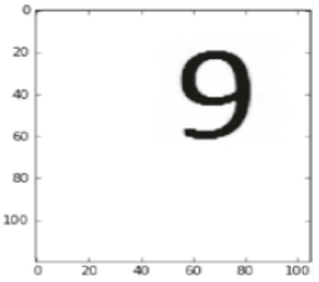


Image (A) with digit 9 *Filter kernel* *Convolved output image (C)*



$$* \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} =$$

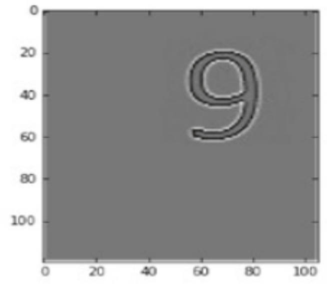


Image (B) with digit 9 translated *Filter kernel* *Convolved output image (D)*

Weight Sharing Through Convolution and Its Advantages

- Translation Invariance Due to Pooling:

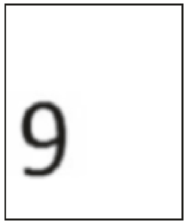


Image A

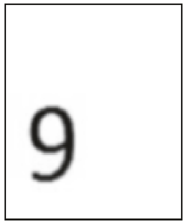


Image B

Convolution with same filter H

0	**0**	**0**	**0**
100	40	0	0
80	20	0	0
0	0	0	0

P

0	**0**	**0**	**0**
50	100	40	0
20	80	20	0
0	0	0	0

P'

Output Feature maps

Max Pooling with 2×2 receptive field

100	**0**
80	0

M

100	**40**
80	20

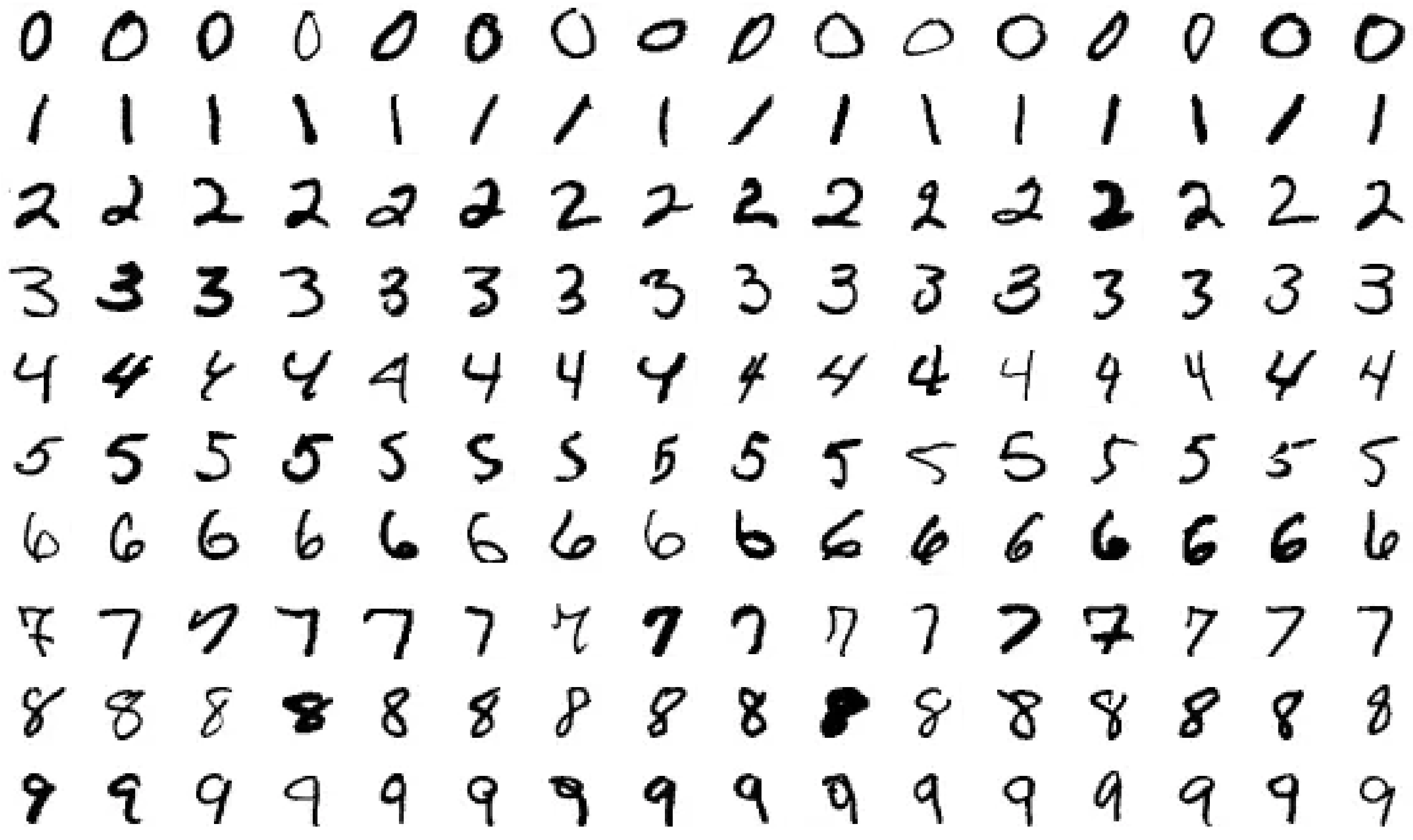
M'

Output after Max Pooling

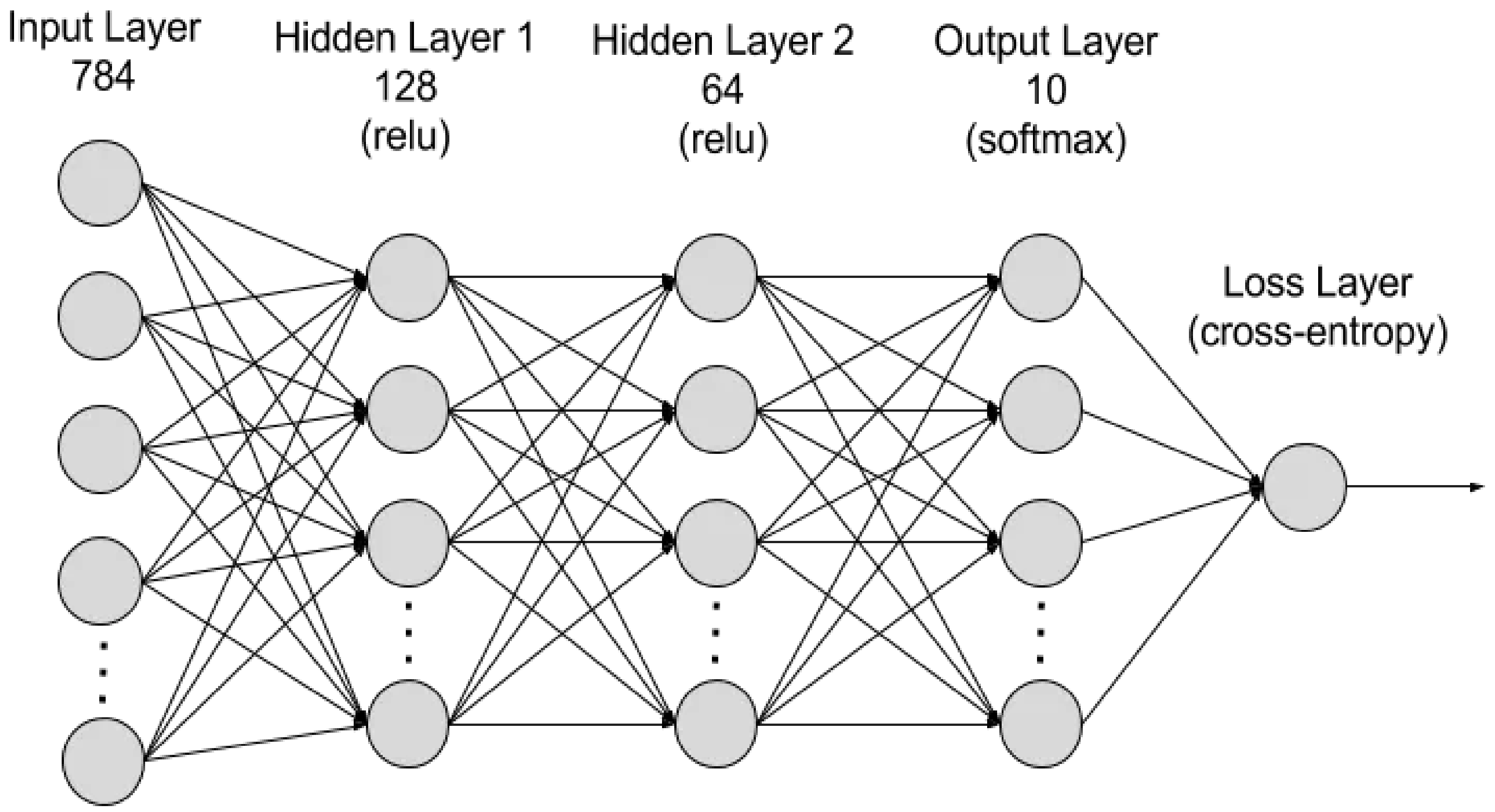
A classification problem using CNN in pytorch

- Load and normalize the train and test data.
- Define the Convolutional Neural Network (CNN).
- Define the loss function and optimizer.
- Train the model on the train data.
- Validate the model using validation data.
- Test the model on the test data

Digit classification(MNIST database)



Digit classification model



Digit classification

```
transform = transforms.Compose([transforms.ToTensor(),
                              transforms.Normalize((0.5,), (0.5,)),
                              ])
# defining the training and testing set
trainset = datasets.MNIST('./data', download=True, train=True, transform=transform)
testset = datasets.MNIST('./', download=True, train=False, transform=transform)

# defining trainloader and testloader
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)

# shape of training data
dataiter = iter(trainloader)
images, labels = dataiter.next()

print(images.shape)
print(labels.shape)
# visualizing the training images
plt.imshow(images[0].numpy().squeeze(), cmap='gray')
# shape of validation data
dataiter = iter(testloader)
images, labels = dataiter.next()
print(images.shape)
print(labels.shape)
```

Digit classification

```
class Net(nn.Module):
  def __init__(self):
     super(Net, self).__init__()

     self.cnn_layers = nn.Sequential (
        nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(4),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
        # Defining another 2D convolution layer
        nn.Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(4),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2, stride=2),
     )

     self.linear_layers = nn.Sequential(
        nn.Linear(4 * 7 * 7, 10)
     )

  # Defining the forward pass
  def forward(self, x):
     x = self.cnn_layers(x)
     x = x.view(x.size(0), -1)
     x = self.linear_layers(x)
     return x
# defining the model
model = Net()
```

Digit classification

```
•   optimizer = optim.Adam(model.parameters(), lr=0.01)
•   # defining the loss function
•   criterion = nn.CrossEntropyLoss()
•   # checking if GPU is available
•   if torch.cuda.is_available():
•       model = model.cuda()
•       criterion = criterion.cuda()
•
•   print(model)
•   for i in range(10):
•       running_loss = 0
•       for images, labels in trainloader:

•           if torch.cuda.is_available():
•             images = images.cuda()
•             labels = labels.cuda()

•           # Training pass
•           optimizer.zero_grad()
•
•           output = model(images)
•           loss = criterion(output, labels)
•
•           #This is where the model learns by backpropagating
•           loss.backward()
•
•           #And optimizes its weights here
•           optimizer.step()
•
•           running_loss += loss.item()
•       else:
•           print("Epoch {} - Training loss: {}".format(i+1, running_loss/len(trainloader)))
```

Digit classification

```
•  correct_count, all_count = 0, 0
•  for images,labels in testloader:
•    for i in range(len(labels)):
•      if torch.cuda.is_available():
•          images = images.cuda()
•          labels = labels.cuda()
•      img = images[i].view(1, 1, 28, 28)
•      with torch.no_grad():
•          logps = model(img)

•
•      ps = torch.exp(logps)
•      probab = list(ps.cpu()[0])
•      pred_label = probab.index(max(probab))
•      true_label = labels.cpu()[i]
•      if(true_label == pred_label):
•        correct_count += 1
•      all_count += 1

•  print("Number Of Images Tested =", all_count)
•  print("\nModel Accuracy =", (correct_count/all_count))
```

Image Classification

- CIFAR-10 consists of 60,000 tiny 32 × 32 color (RGB) images, labeled with an integer corresponding to 1 of 10 classes: airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).

Image Classification

- Load and normalize the CIFAR10 training and test datasets using torchvision
- Define a Convolutional Neural Network
- Define a loss function
- Train the network on the training data
- Test the network on the test data

LENet

- Yann LeCun et al. created

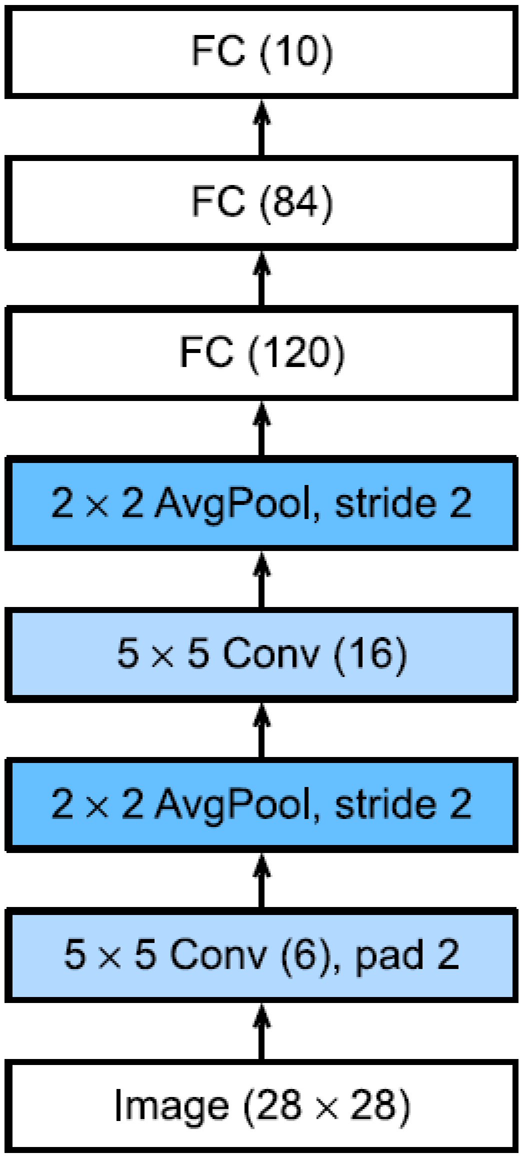
the initial form of LeNet in 1989.

- Paper: Backpropagation Applied

to Handwritten Zip Code

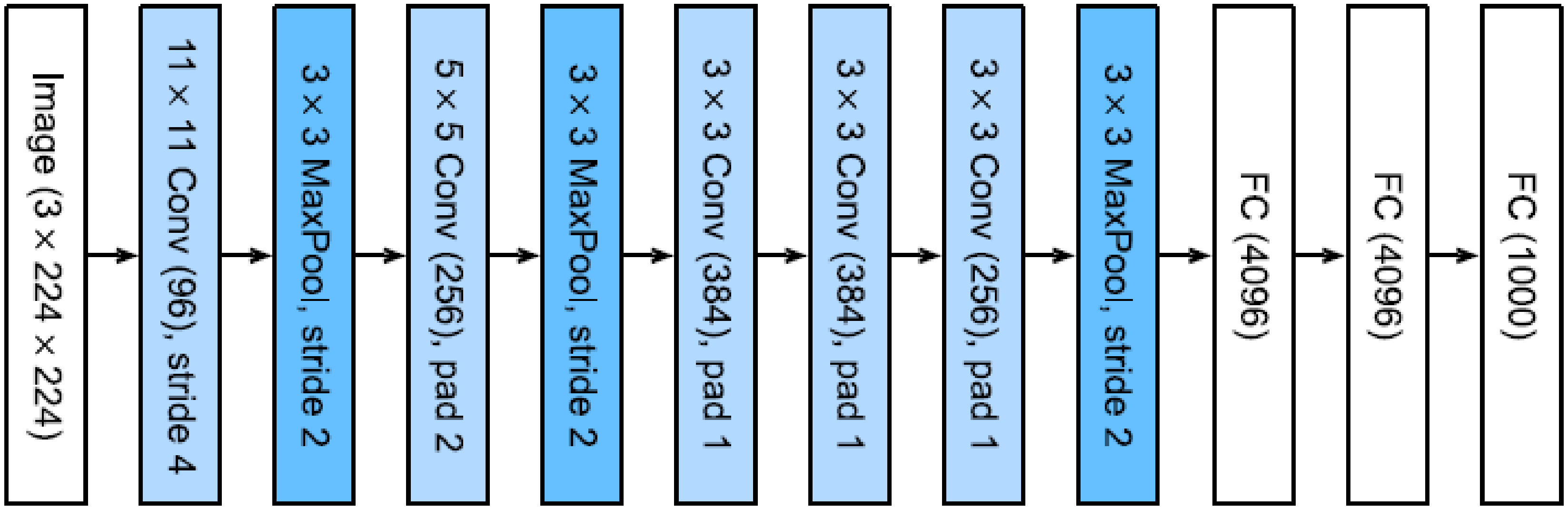
Recognition

- 1994 MNIST database was
- Developed.



AlexNet

- AlexNet is the name of a convolutional neural network (CNN) architecture, designed by Alex Krizhevsky in collaboration with Ilya Sutskever and Geoffrey Hinton.



Performance increasing of models

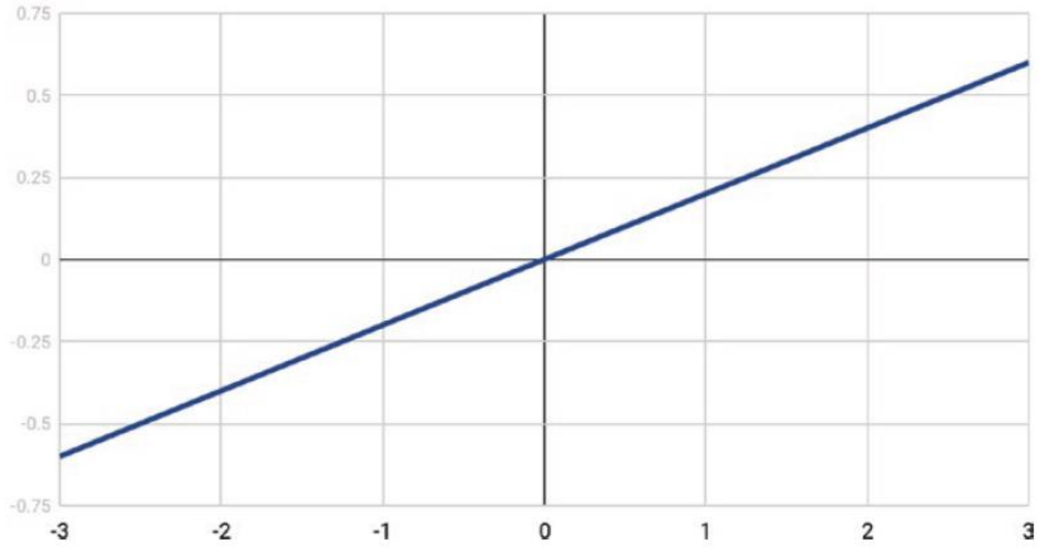
- Tune hyper-parameters
- Use different optimizers
- Image data augmentation
- Try more complex architectures
- Deal with overfitting
- Find more data
- Use Regularizer

Activation Functions used in CNN

- **Linear Activation Function:**
- The linear activation function calculates the neuron output by multiplying weights to inputs as

$$f(x) = x_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

- The output of linear activation function varies from $-\infty$ to ∞.



Linear activation function graph

Activation Functions used in CNN

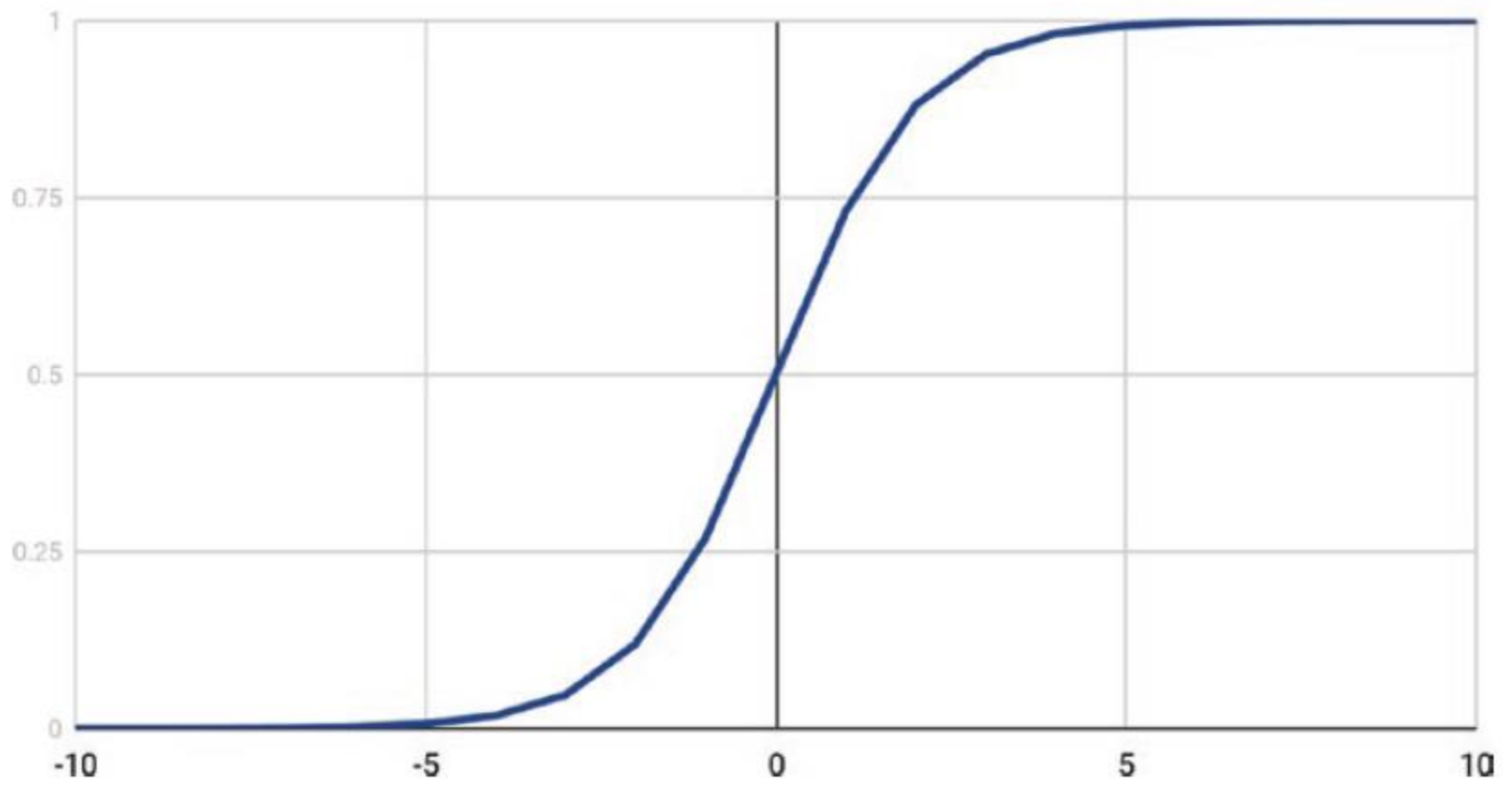
- **Drawbacks:**
- The gradient descent requires calculating a first-order derivative of the input, which, in the case of linear activation, is a constant. The first derivative of a constant is a zero.
- In other words, a linear activation function turns your network into just one layer. That means your network can learn only the linear dependencies of inputs to output.

Activation Functions used in CNN

- **Sigmoid or Logistic Activation Function:**
- The sigmoid function always yields a value between 0 and 1. It is represented by
$$f(x) = \frac{1}{1+e^{-x}}$$
- This makes the output smooth without many jumps as the input value fluctuates.
- The other advantage is that this is a nonlinear function and does not generate a constant value from a first-order derivative.
- This makes it suitable for deep learning with backpropagation that updates weights based on gradient descent

Activation Functions used in CNN

- Sigmoid or Logistic Activation Function:



Sigmoid activation function graph

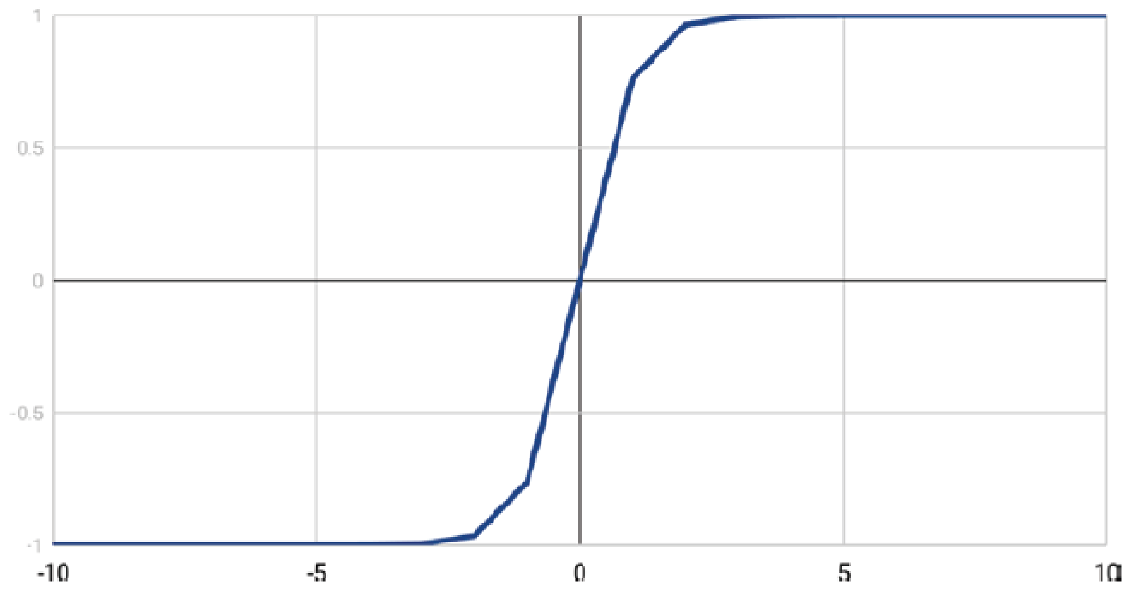
Activation Functions used in CNN

- **Sigmoid or Logistic Activation Function**:
- The disadvantage of the sigmoid function is that the output does not change between large or small input values, which make it unsuitable for cases where the feature vector contains large or small values.
- One way to overcome this disadvantage is to normalize your feature vector to have values between -1 and 1 or between 0 and 1.
- The S-shaped curve is not centered at zero.

Activation Functions used in CNN

- **Tanh/Hyperbolic Tangent**:
- Tanh is similar to the sigmoid activation function except that Tanh is zero-centered

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



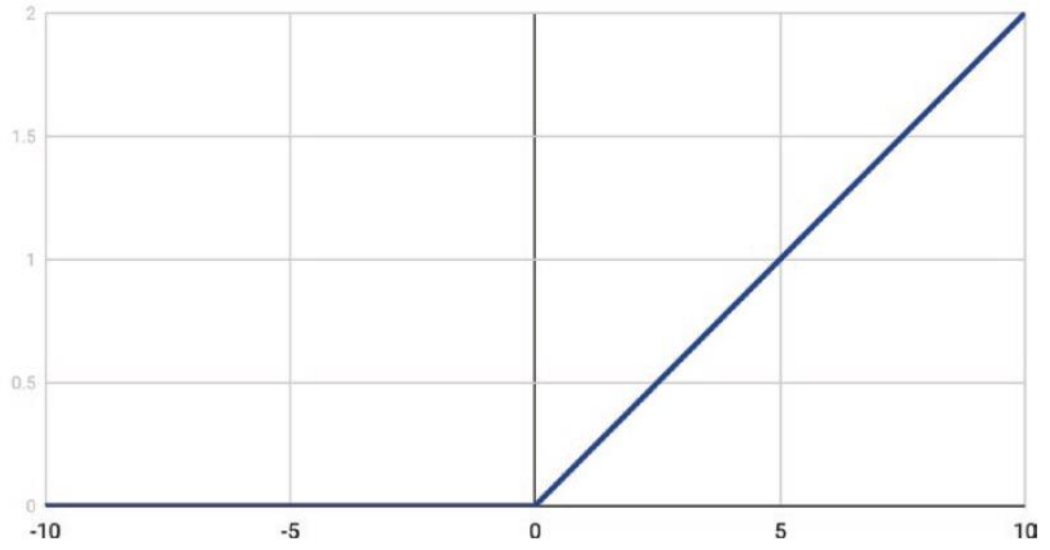
TanH activation function graph (zero-centered)

Activation Functions used in CNN

- **Rectified Linear Unit**: The rectified linear unit (ReLu) determines the neuron output based on the following expression

$$f(x) = (0, \max)$$

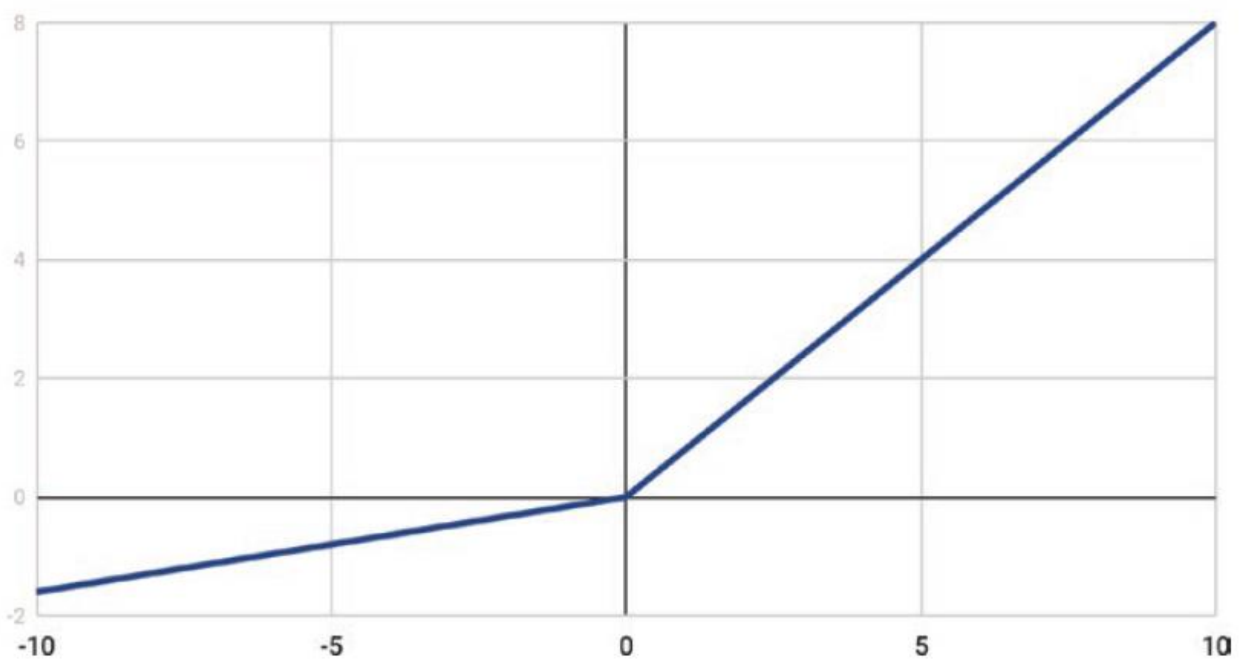
- If the value of x is positive, ReLU takes that value as an output; otherwise, it outputs as zero.



ReLU activation graph (with value ranges between 0 and infinity)

Activation Functions used in CNN

- **Leaky ReLU**: Leaky ReLU provides a slight variation of ReLU. Instead of making the negative value of x to zero, it multiplies the negative value of x by a small number such as 0.01.



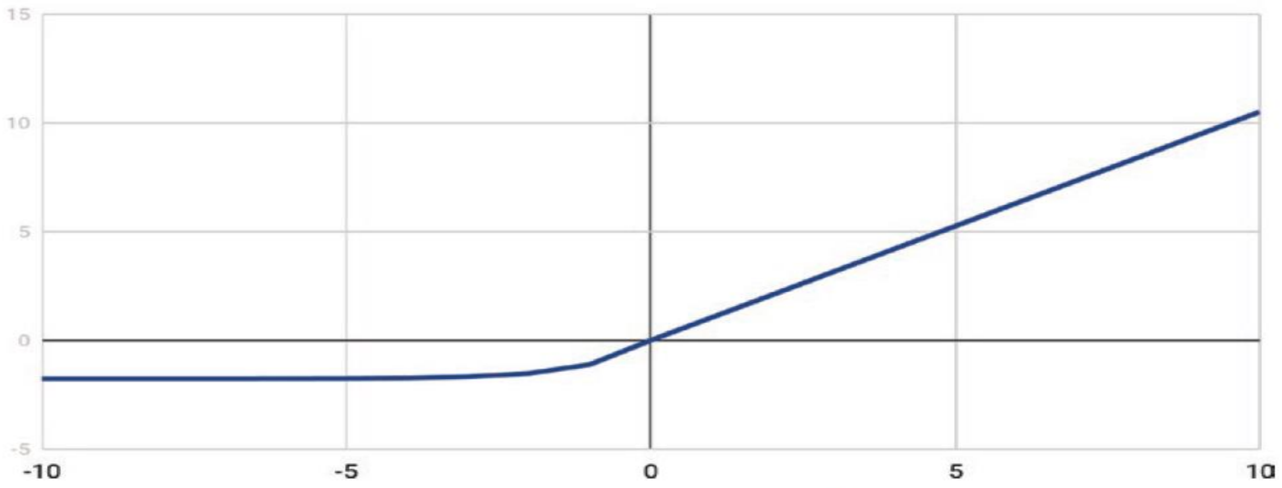
Leaky ReLU graph (modified ReLU by taking negative value multiplied with a small number)

Activation Functions used in CNN

- **Scaled Exponential Linear Unit**: A scaled exponential linear unit (SELU) computes neuron outputs using the following equation

$$f(\alpha, x) = \lambda \begin{cases} \alpha\left(e^{x} - 1\right) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

where the value of λ = 1.05070098 and the value of ***α*** = 1.67326324. These values are fixed and do not change during backpropagation.



SELU activation graph

Activation Functions used in CNN

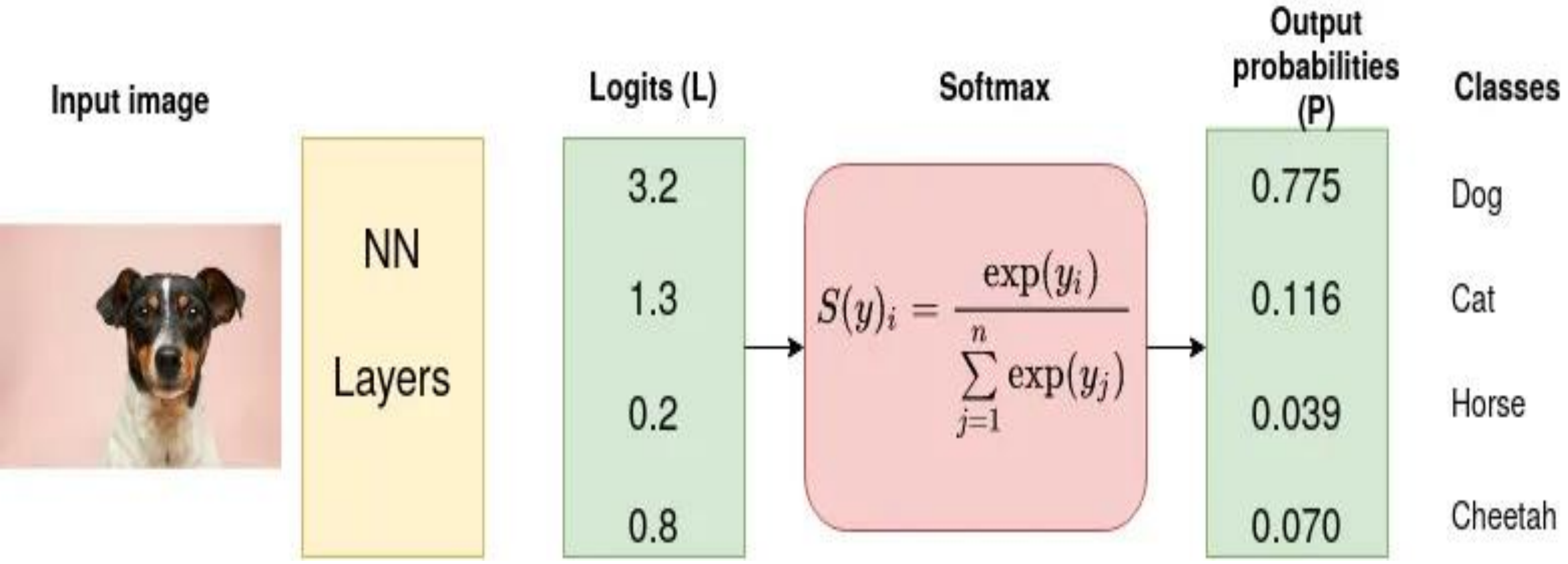
- **Softmax:** Softmax is a function that takes an input vector of real numbers, normalizes it into a probabilit distribution, and generates outputs in the range (0,1) with the sum of output values equal to 1.
- It is most often used as the activation for the last layer (output layer) of a classification neural network. The result is interpreted as the prediction probability of each class.
- The softmax transformation is calculated using the following formula:

$$\sigma(\mathrm{x})_i = \frac{e^{\mathrm{x}_i}}{\sum_{j=1}^{K} e^{\mathrm{x}_j}} \text{ for } i = 1,\ldots,K \text{ and } \mathrm{x} = (\mathrm{x}_1,\cdots,\mathrm{x}_K) \in \mathbb{R}^K$$

Softmax Output Layers

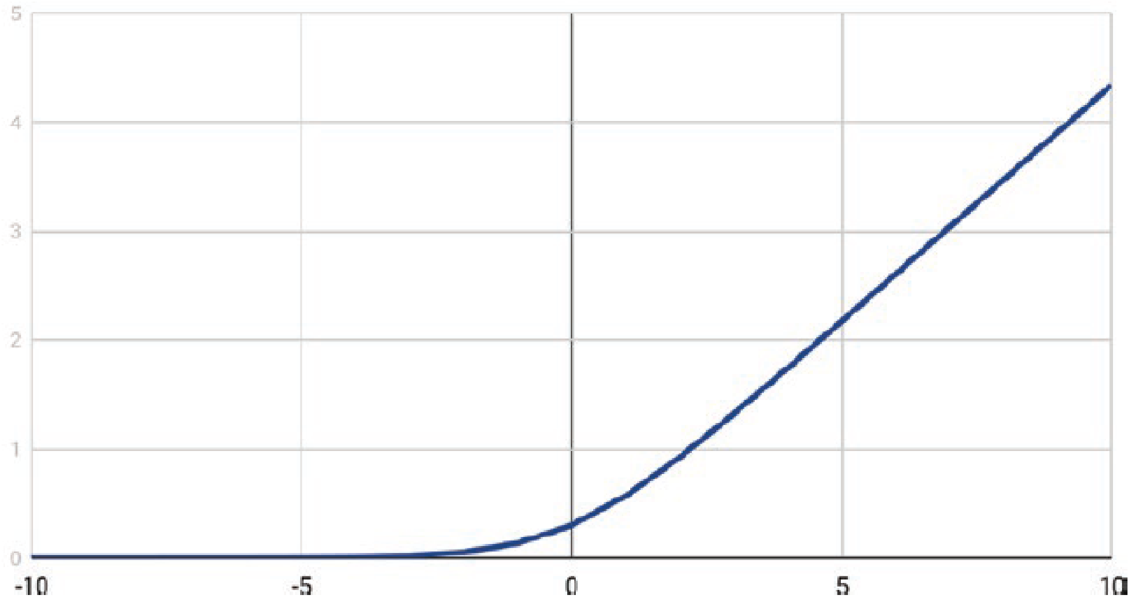
- We want our output vector to be a probability distribution over a set of mutually exclusive labels.
- Build a neural network to recognize handwritten digits from the MNIST dataset.
- The desired output vector is of the form below $\left[p_0, p_1, p_2, \ldots p_9 \right]$ where $\sum_{i=0}^{9} p_i = 1$.
- This is achieved by using a special output layer called a softmax layer.

Softmax Output Layers



Activation Functions used in CNN

- **Soft plus Activation Function**: The soft plus activation function applies smoothing to the activation function value x.
- It uses the log of exponent. $f(x) = \log(1 + e^x)$
- Soft plus is also
- called the
- SmoothReLU
- function.



Softplus activation graph

Loss functions used in CNN models

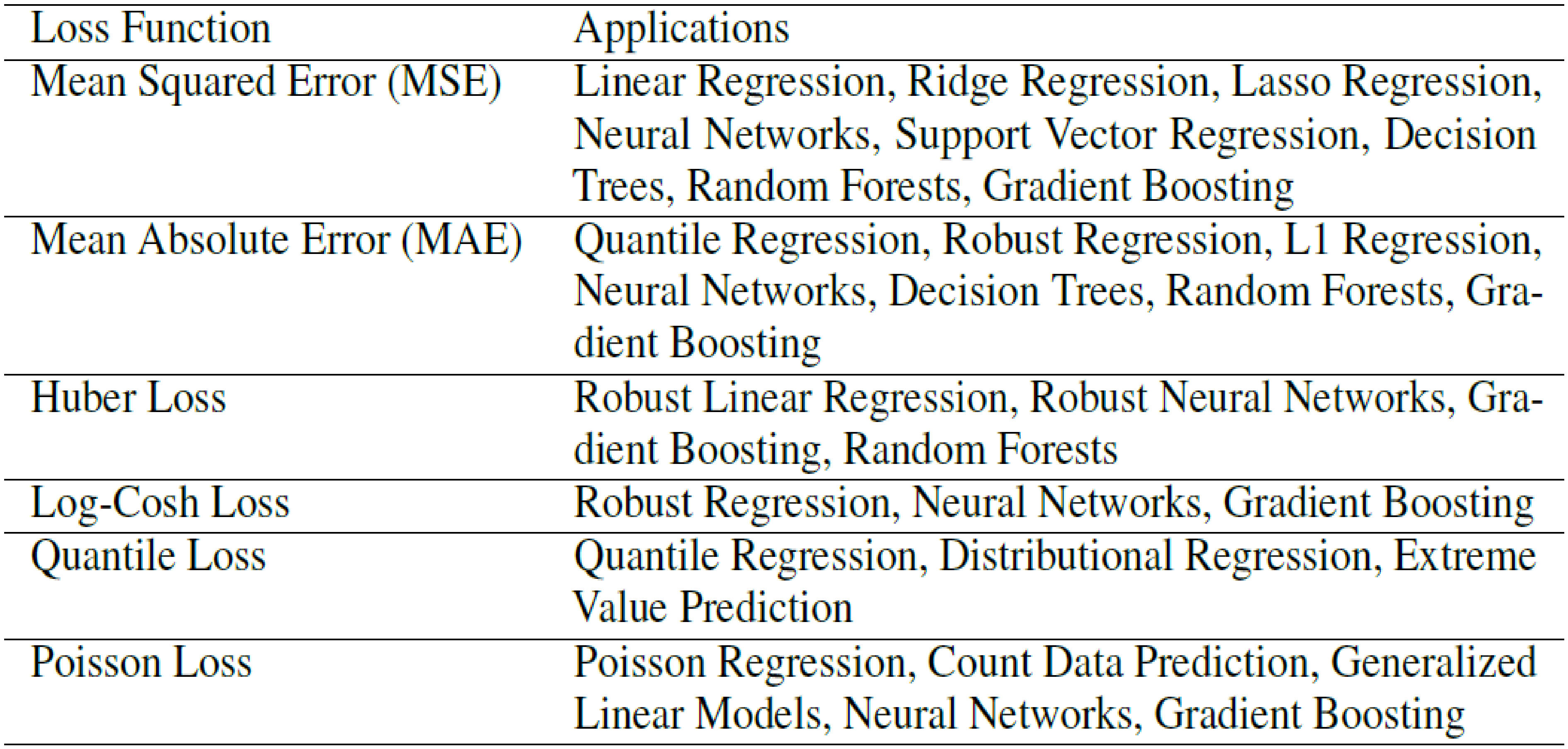
- The loss function in a neural network quantifies the difference between the expected outcome and the outcome produced by the machine learning model.
- From the loss function, we can derive the gradients which are used to update the weights.
- The average over all losses constitutes the cost.
- The loss, calculated by a loss function, indicates the difference between the outputs of the model (predicted label) and the actual labels that belong to the input.

Types of loss functions

- Regression based loss functions.
- Classification based loss functions.

Common loss functions used for regression

- Loss Functions and their applications in regression tasks.



Loss Function	Applications
Mean Squared Error (MSE)	Linear Regression, Ridge Regression, Lasso Regression, Neural Networks, Support Vector Regression, Decision Trees, Random Forests, Gradient Boosting
Mean Absolute Error (MAE)	Quantile Regression, Robust Regression, L1 Regression, Neural Networks, Decision Trees, Random Forests, Gradient Boosting
Huber Loss	Robust Linear Regression, Robust Neural Networks, Gradient Boosting, Random Forests
Log-Cosh Loss	Robust Regression, Neural Networks, Gradient Boosting
Quantile Loss	Quantile Regression, Distributional Regression, Extreme Value Prediction
Poisson Loss	Poisson Regression, Count Data Prediction, Generalized Linear Models, Neural Networks, Gradient Boosting

Mean Squared Error (MSE)

- The Mean Square Error (MSE) measures the average of the squared differences between the predicted values and the true values. The MSE loss function can be defined mathematically as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2,$$

where n is the number of samples, y_i is the true value of the i_{th} sample and $\hat{y_i}$ is the predicted value of the i_{th} sample.

Mean Squared Error (MSE)

- MSE is also called L2 norm or L2 loss.
- The MSE loss function has the following properties:
- **Non-negative**: A value of 0 indicates a perfect fit, while larger values correspond to higher discrepancies between predictions and actual values.
- **Quadratic:** MSE is a quadratic function of the prediction errors, which means it places more emphasis on larger errors than smaller ones. This property makes it sensitive to outliers
- **Differentiable**:This property allows for the efficient computation of gradients, which is essential for optimization algorithms like gradient descent

Mean Squared Error (MSE)

- **Convex**: MSE is a convex function, which means it has a unique global minimum, hence suited for regression problems.
- **Scale-dependent**: The value of MSE depends on the scale of the target variable, making it difficult to compare the performance of models across different problems or target variable scales
- However, it is not robust to outliers due to the square of the error term. Thus if the data includes outliers, it is better to use another loss function.

Mean Absolute Error (MAE)

- The Mean Absolute Error (MAE) is another commonly used loss function in regression problems.
- It measures the average of the absolute differences between the predicted values and the true values.
- The MAE loss can be defined as

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|,$$

- where n is the number of samples, y_i and $\hat{y_i}$ are the true and predicted value of the i_{th} sample.

Mean Absolute Error (MAE)

- The MAE loss function has the following properties:
- **Non-negative:**
- **Linear:** MAE is a linear function of the prediction errors, which treats all errors equally regardless of their magnitude. This property makes MAE less sensitive to outliers than MSE.
- **Robust:** Due to its linear nature and reduced sensitivity to outliers, MAE is considered a more robust loss function than MSE.
- **Non-differentiable:** Although MAE is continuous, it is not differentiable when the prediction error is zero due to the absolute value function.

Mean Absolute Error (MAE)

- **Convex:** MAE is a convex function, which means it has a uniq Like the MSE, the MAE is non-convex for Deep neural networks due to the multiple layers with non-linear activation functions.ue global minimum.
- **Scale-dependent:** Like MSE, the value of MAE depends on the scale of the target variable, making it difficult to compare the performance of models across different problems or target variable scales.

Huber Loss

- The Huber loss combines the properties of both Mean Squared Error (MSE) and Mean Absolute Error (MAE).
- Huber loss is designed to be more robust to outliers than MSE while maintaining smoothness and differentiability.

$$L(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{for } |y-\hat{y}| \le \delta \\ \delta(|y-\hat{y}| - \frac{1}{2}\delta) & \text{otherwise,} \end{cases}$$

where y is the true value, $\hat{y_i}$ is the predicted value, and δ is a user-specified threshold value

Huber Loss

- When the error is small, the Huber loss function behaves like the MSE loss function, and when the error is large, the Huber loss function behaves like the MAE loss function. This property makes the Huber loss function more robust to outliers than the MSE loss function.
- The Huber loss function is differentiable, which makes it suitable for use in gradient-based optimization algorithms such as stochastic gradient descent (SGD). It is commonly used in linear regression and time series forecasting, as it can handle outliers and noise in the data.

Log-Cosh Loss

- The Log-Cosh loss function is smooth and differentiable. It is commonly used in regression problems where the data may contain outliers or noise.

$$L(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\log(\cosh(y_i - \hat{y}_i)),$$

- where y is the true value, $\hat{y_i}$ is the predicted value and n is the number of samples.
- One of the advantages of the log-cosh loss function is that it is less sensitive to outliers than the mean squared error (MSE), as it is not affected by extreme data values. However, it is more sensitive to small errors than the Huber loss

Root Mean Squared Error (RMSE)

- The Root Mean Square Error (RMSE) is the square root of the mean squared error (MSE) defined as

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2},$$

- where y_i is the true value, $\hat{y_i}$ is the predicted value, and n is the number of samples.
- The RMSE measures the average deviation of the predictions from the true values.

Classification Loss Functions

- Several loss functions can be used for classification tasks, depending on the specific problem and algorithm.
- **Binary Cross-Entropy Loss (BCE):**
- The Binary Cross Entropy (BCE), also known as log loss, is a commonly used loss function for binary classification problems.
- It measures the dissimilarity between the predicted probability of a class and the true class label.
- Cross-entropy measures the dissimilarity between
- two probability distributions.

Binary Cross-Entropy Loss (BCE)

- In binary classification the true class has a value of 1, and the other class has a value of 0.
- The predicted probability is represented by a vector of predicted probabilities for each class.
- The loss function is defined as

$$L(y,p) = -(y\log(p) + (1-y)\log(1-p))$$

- where y is the true class label (0 or 1) and p is the predicted probability of the positive class.
- The loss function is minimized when the predicted probability p equals the true class label y.

Weighted Binary Cross Entropy (WBCE)

- BCE is sensitive to the class imbalance problem, which occurs when the number of samples of one class is significantly greater than the other.
- This can be overcome by using Weighted Binary Cross Entropy loss function, which is represented by

$$L = -(w_i \cdot y log(p) + w_i(1-y) log(1-p))$$

- where w_i is the weight assigned to the i^{th} sample, y is the true label, and p is the predicted probability of the positive class.

Categorical Cross-entropy Loss (CCE)

- The Categorical Cross Entropy (CCE), also known as the negative log-likelihood loss or Multi-class log loss, is a function used for multi-class classification tasks.
- It measures the dissimilarity between the predicted probability distribution and the true distribution.
- The formula for categorical cross-entropy loss is expressed as

$$L = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{C} y_{i,j} log(p_{i,j})$$

Categorical Cross-entropy Loss (CCE)

- where N is the number of samples, C is the number of classes, y is the true label, and p is the predicted probability of the true class.
- Given the predicted probability distribution, it is defined as the average negative log-likelihood of the true class.
- The true label is a one-hot encoded vector in traditional categorical cross-entropy loss, where the element corresponding to the true class is one, and all other elements are 0.

Sparse Categorical Cross-entropy Loss

- Variation of the categorical cross-entropy loss used for multi-class classification tasks where the classes are encoded as integers rather than one-hot encoded vectors.
- Given that the true labels are provided as integers, we directly select the correct class using the provided label index instead of summing over all possible classes.
- Thus the loss for each example is calculated as

$$H(y, \hat{y}) = -\log(\hat{y}_{i,y_i})$$

Sparse Categorical Cross-entropy Loss

- The final sparse categorical cross-entropy loss is the average over all the samples

$$H(Y, \hat{Y}) = -\frac{1}{n}\sum_{i=1}^{n} \log(\hat{y}_{i,y_i}),$$

- where y_i the true class of the ith sample and $\hat{y}_{i,y_i}$ is the predicted probability of the ith sample for the correct class y_i .

Pytorch loss functions

- **nn.L1Loss:** Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = \left| x_n - y_n \right|,$$

- **nn.MSELoss:** Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = \left(x_n - y_n \right)^2$$

Pytorch loss functions

- **nn.CrossEntropyLoss**:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

- **nn.NLLLoss**: The negative log likelihood loss.
- **nn.BCELoss**: Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

Pytorch loss functions

- **nn.HuberLoss:**

$$l_n = \begin{cases} 0.5(x_n - y_n)^2, & \text{if } |x_n - y_n| < delta \\ delta * (|x_n - y_n| - 0.5 * delta), & \text{otherwise} \end{cases}$$

- **nn.SmoothL1Loss:**

$$l_n = \begin{cases} 0.5(x_n - y_n)^2 / beta, & \text{if } |x_n - y_n| < beta \\ |x_n - y_n| - 0.5 * beta, & \text{otherwise} \end{cases}$$

Regularization

- Regularization can be simply seen as any modification to the model (or its training process) that intends to improve the error on the unseen data (at the cost of the error on the training data) by systematically limiting the capacity of the model.
- Regularization is a set of techniques that can prevent over-fitting in neural networks and thus improve the accuracy of a Deep Learning model when facing completely new data from the problem domain.

Regularization

- Regularization is any supplementary technique that aims at making the model generalize better, i.e. produce better results on the test set.
- A neural network is a function $f_w : x \mapsto y$ with trainable weights $w \in W$.
- Training the network means finding a weight configuration w^* by minimizing a loss function $\mathcal{L} : W \to \mathbb{R}$ as follows:

$$w^* = \arg\min_w \mathcal{L}(w).$$

Regularization

- Usually in regularization, the loss function takes the form of expected risk:

$$\mathcal{L} = \mathbb{E}_{(x,t)\sim P}\Big[E\big(f_w(x), t\big) + R(\ldots)\Big],$$

- where we identify two parts, an error function E and a regularization term R.
- The error function depends on the targets and assigns a penalty to model predictions according to their consistency with the targets.
- The regularization term assigns a penalty to the model based on other criteria. It may depend on anything except the targets.

Regularization

- The expected risk cannot be minimized directly since the data distribution P is unknown. Instead, a **training set D** sampled from the distribution is given.
- The minimization of the expected risk can be then approximated by minimizing the empirical risk $\hat{\mathcal{L}}$:

$$\arg\min_{w} \frac{1}{|\mathcal{D}|} \sum_{(x_i,t_i)\in\mathcal{D}} E(f_w(x_i), t_i) + R(\ldots)$$

Regularization

- In the minimization of the empirical risk, we can identify the following elements that are responsible for the value of the learned weights, and thus can contribute to regularization:
- D: The training set,
- f: The selected model,
- E: The error function,
- R: The regularization term,
- The optimization procedure itself.

Regularization via data(The training set)

- The quality of a trained model depends largely on the training data and it is possible to employ regularization via data.
- This is done by **applying some transformation** to the training set D, resulting in a new set $\mathcal{D}_{R}$.
- Other methods allow **generating new samples** to create a larger, possibly infinite, augmented dataset.
- The goal of regularization via data is either one of them, or the other, or both. They both rely on **transformations with (stochastic) parameters**:

Regularization via data(The training set)

- An example of a transformation with stochastic parameters is the corruption of inputs by Gaussian noise.

$$\tau_\theta(x) = x + \theta, \quad \theta \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$$

- Where τ_θ is a transformation function with stochastic parameters.
- The stochasticity of the transformation parameters is responsible for generating new samples, i.e. data augmentation.

Regularization

- **Effect on the data representation:**
- Representation-preserving transformations: Preserve the feature space and attempt to preserve the data distribution
- Representation-modifying transformations: Map the data to a different representation(different distribution or even new feature space) that may disentangle the underlying factors of the original representation and make the learning problem easier.

Regularization

- Target-preserving data augmentation**:** These methods use stochastic transformations in input and hidden-feature spaces, while preserving the original target t.
- **Examples of Regularization via data**.
- Gaussian noise on input and hidden units.
- Dropout
- Batch normalization
- Training with adversarial examples.
- Inter- and extrapolation in hidden-feature space.

Dropout Layers and Regularization

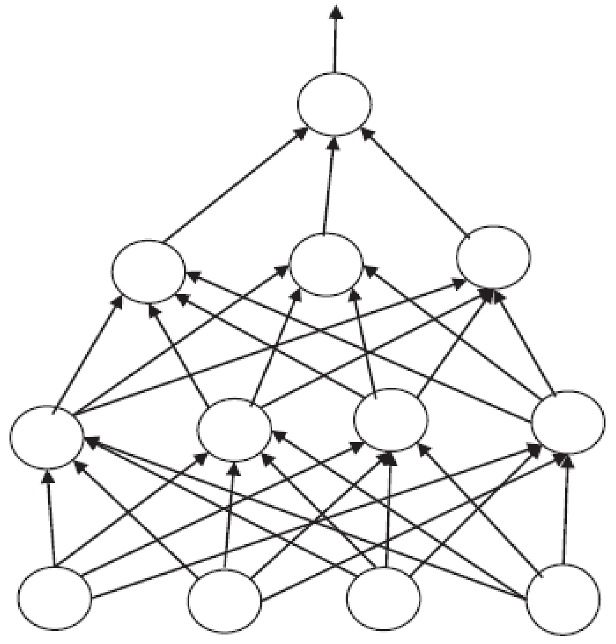
- **Dropout:**
- Dropout is an activity to regularize weights in the fully connected layers of a convolutional neural network to avoid over fitting.
- A specified proportion of neural network units, both hidden and visible, is randomly dropped at training time for each training sample in a mini batch so that the remaining neurons can learn important features all by themselves and not rely on cooperation from other neurons.

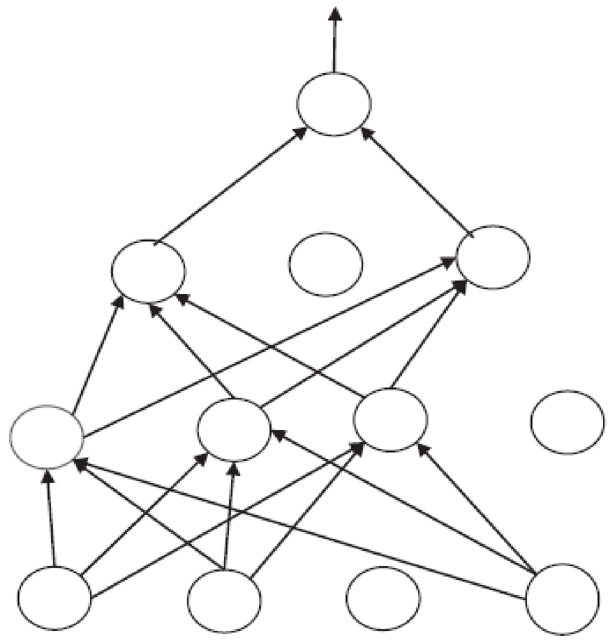
Dropout Layers and Regularization

- When the neuron units are dropped randomly, each such setting of remaining available neurons produces a different network.
- So, training a neural network with dropout is equivalent to training a set of different neural networks where each network very seldom gets trained, if at all.
- At test time, it is not possible to compute the predictions from all such possible networks.

Dropout Layers and Regularization

- For a convolutional neural network, the units in the fully connected layers and their corresponding incoming and outgoing connections are usually dropped. Hence, the different filter-kernel weights do not require any adjustment while predicting for the test dataset


Neural Network without dropout


Neural Network with three units dropped

Regularization via the network architecture

- A network architecture **f** can be selected to have certain properties or match certain assumptions in order to have a regularizing effect.
- **Model fitting or Mapping**: The decision about the number of layers and units, which allows the mapping to be neither too simple nor too complex (thus avoiding under-fitting and over-fitting).
- **Weight sharing:** Reusing a certain trainable parameter in several parts of the network is referred to as weight sharing. Example is weight sharing in autoencoders.

Regularization via the network architecture

- **Activation functions**: Choosing the right activation function is quite important.
- Rectified linear units (ReLUs) improved the performance of many deep architectures both in the sense of training times and accuracy.
- **Multi-task learning**:
- Meta-learning, where multiple tasks from the same domain are learned sequentially, using previously gained knowledge as bias for new tasks.
- Transfer learning, where knowledge from one domain is transferred into another domain.

Regularization via the network architecture

- **Model selection**: The selection of the number of units in a specific network architecture.
- The best among several trained models (e.g. with different architectures) can be selected by evaluating the predictions on a validation set.

Regularization via the error function

- The error function E can also have a regularizing effect.
- Different loss functions can be selected for better optimization of the error function.
- Dice coefficient optimization (Milletari et al., 2016) which is robust to class imbalance.

Regularization via regularization term (R)

- Regularization can be achieved by adding a regularize R into the loss function L.
- The regularization term is independent of the targets. Instead, it is used to encode other properties of the desired model, to provide inductive bias.
- The independence of R from t has an important implication: it allows additionally using unlabeled samples (semi-supervised learning) to improve the learned model based on its compliance with some desired properties.

Regularization via regularization term (R)

$$\mathcal{L} = \mathbb{E}_{(x,t)\sim P}\left[E\left(f_w(x), t\right) + R(\dots)\right],$$

$$\arg\min_{w} \frac{1}{|\mathcal{D}|} \sum_{(x_i,t_i)\in\mathcal{D}} E\left(f_w(x_i), t_i\right) + R(\dots)$$

- A classical regularizer is weight decay

$$R(w) = \lambda \frac{1}{2}\|w\|_2^2$$

where λ is a weighting term controlling the importance of the regularization over the consistency. That is, we leave the biases unregularized, and penalize half the squared L2 or the L1 norm.

Regularization via regularization term (R)

- Another common prior assumption that can be expressed via the regularization term is “smoothness” of the learned mapping.
- If $x_1 \approx x_2,$ then $f_w(x_1) \approx f_w(x_2).$
- Smoothness regularization term can be expressed by the following loss term

$$R(f_w, x) = \|J_{f_w}(x)\|_F^2$$

- Where $\|\cdot\|_F$ term denotes the Frobenius norm and $J_{f_w}(x)$ is the Jacobian of the neural network input-to-output mapping f_w .

Regularization via optimization

- Stochastic gradient descent (SGD) is the most frequently used optimization algorithm in the context of deep neural networks.
- Stochastic gradient descent is an iterative optimization algorithm using the following update rule:

$$w_{t+1} = w_t - \eta_t \nabla_w \mathcal{L}(w_t, d_t)$$

- Where $\nabla \mathcal{L}(w_t, d_t)$ is the gradient of the loss $\mathcal{L}$ evaluated on a mini-batch d_t from the training Set $\mathcal{D}$.

Regularization via optimization

- It is frequently used in combination with momentum and other techniques to improve the convergence speed.

Initialization and warm-start methods

- These methods affect the initial selection of

  the model weights.
- Sampling the initial weights from a carefully tuned distribution.
- Another option is pre-training on different data, or with a different objective, or with partially different architecture.
- Transfer learning.

Update methods

- **Update rules:**
- Momentum, Nesterov’s accelerated gradient method, AdaGrad, AdaDelta, RMSProp, Adam etc.
- Learning rate schedules
- Online batch selection
- **Gradient and weight filters:**
- Dropout corresponds to optimization steps in subspaces of weight space.
- Anneal SGD
- Annealed noise on targets.

Termination methods

- There are numerous possible stopping criteria and selecting the right moment to stop the optimization procedure.
- It may improve the generalization by reducing the error caused by the discrepancy between the minimizers of expected and empirical risk.
- Termination using a validation set.
- Termination without using a validation set and depend on fixed number of iterations.
- Early stopping.

Optimization Algorithms (Optimizers)

- The learning objective of a neural network is to determine the most optimized weights and biases.
- To determine the ideal weights, the learning algorithm optimizes the loss function so that it finds weights that make the loss function have the minimum value.
- **The mathematical function that optimizes the loss function is called the optimization algorithm or optimizer.**

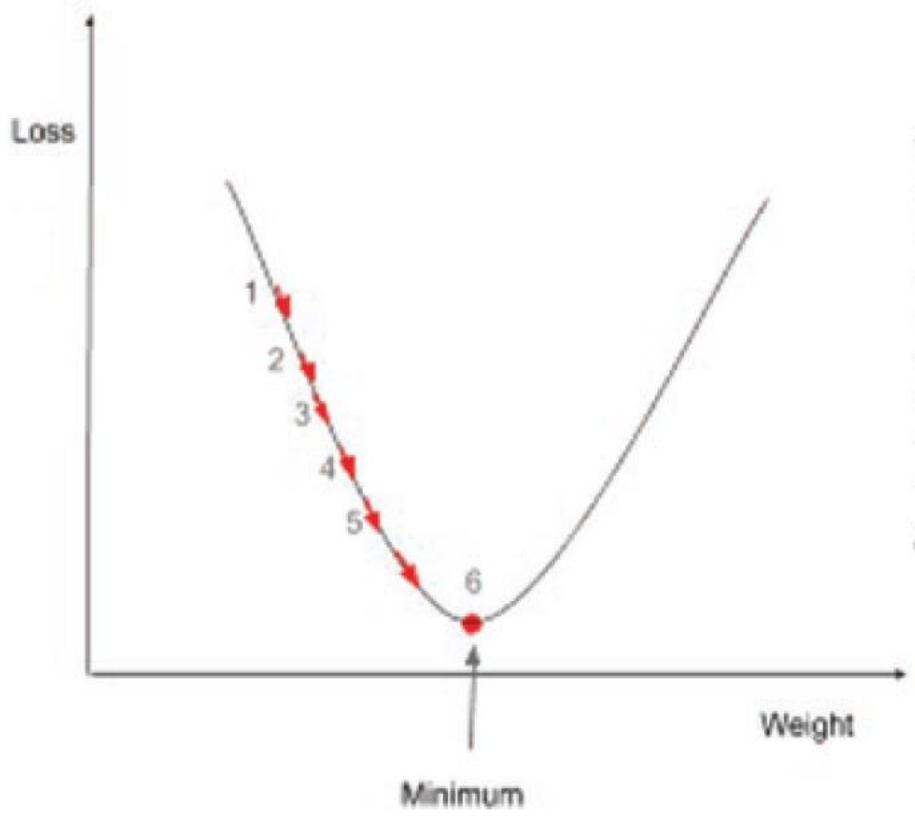
Gradient Descent

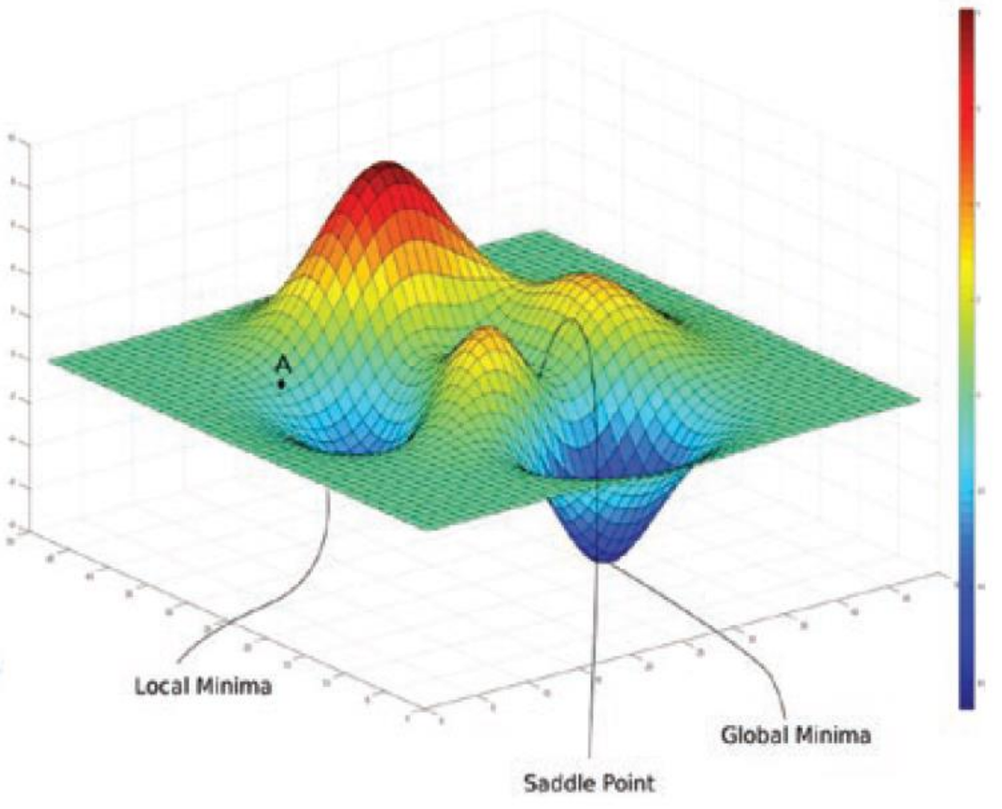
- Gradient descent is an optimization algorithm that finds weights where the loss function (also known as cost function) is zero or minimum.
- From calculus, we know that the first derivative of a function at a point gives the slope or gradient of the function at that point.
- The derivative is calculated to get the gradient to determine which direction along the curve to move to get the new set of weights

$$x_{k+1} = x_k - \alpha_k g_k$$

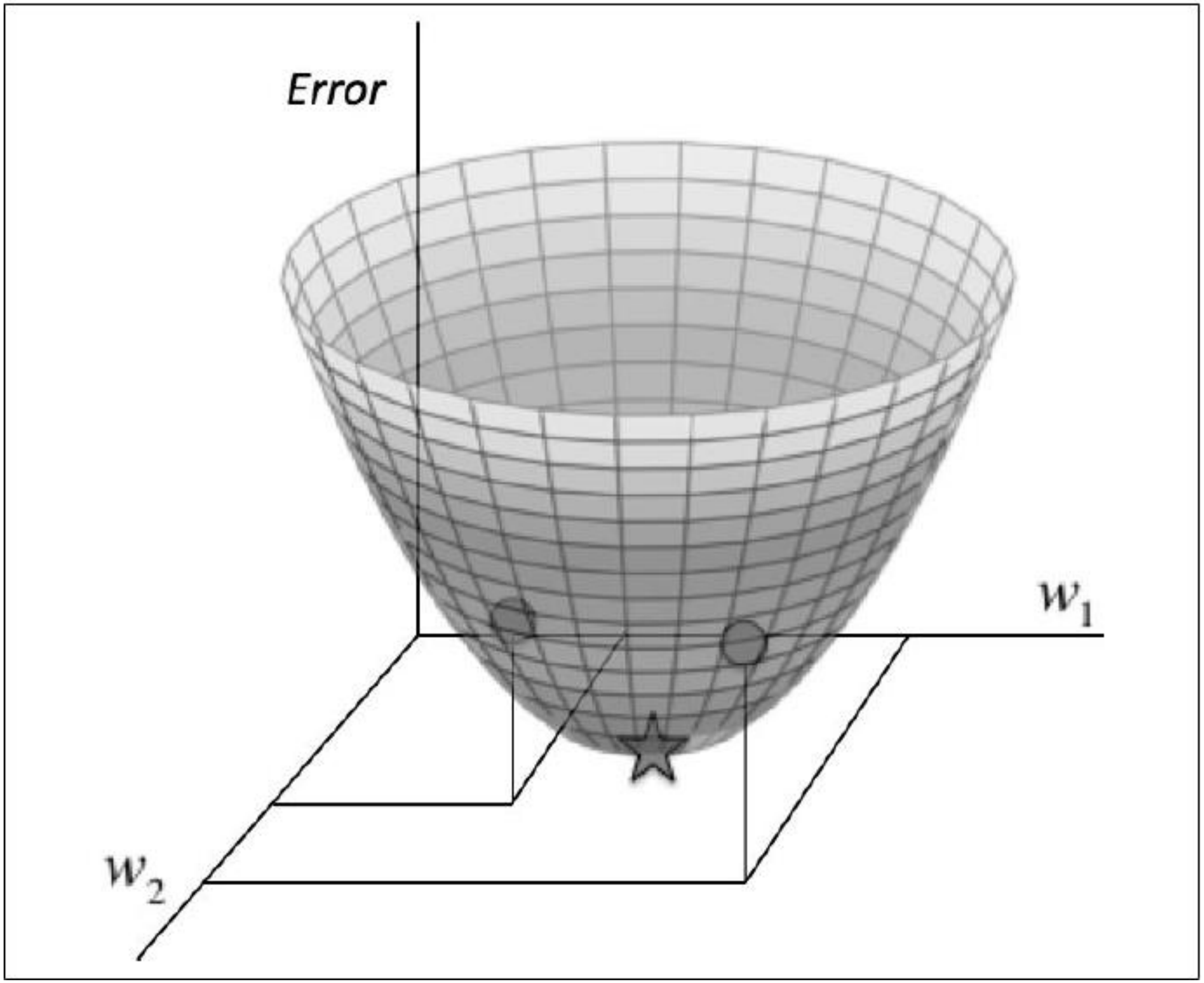
Gradient Descent

- The learning rate determines the size of the steps through which the gradient descends the curve to reach the minimum point.



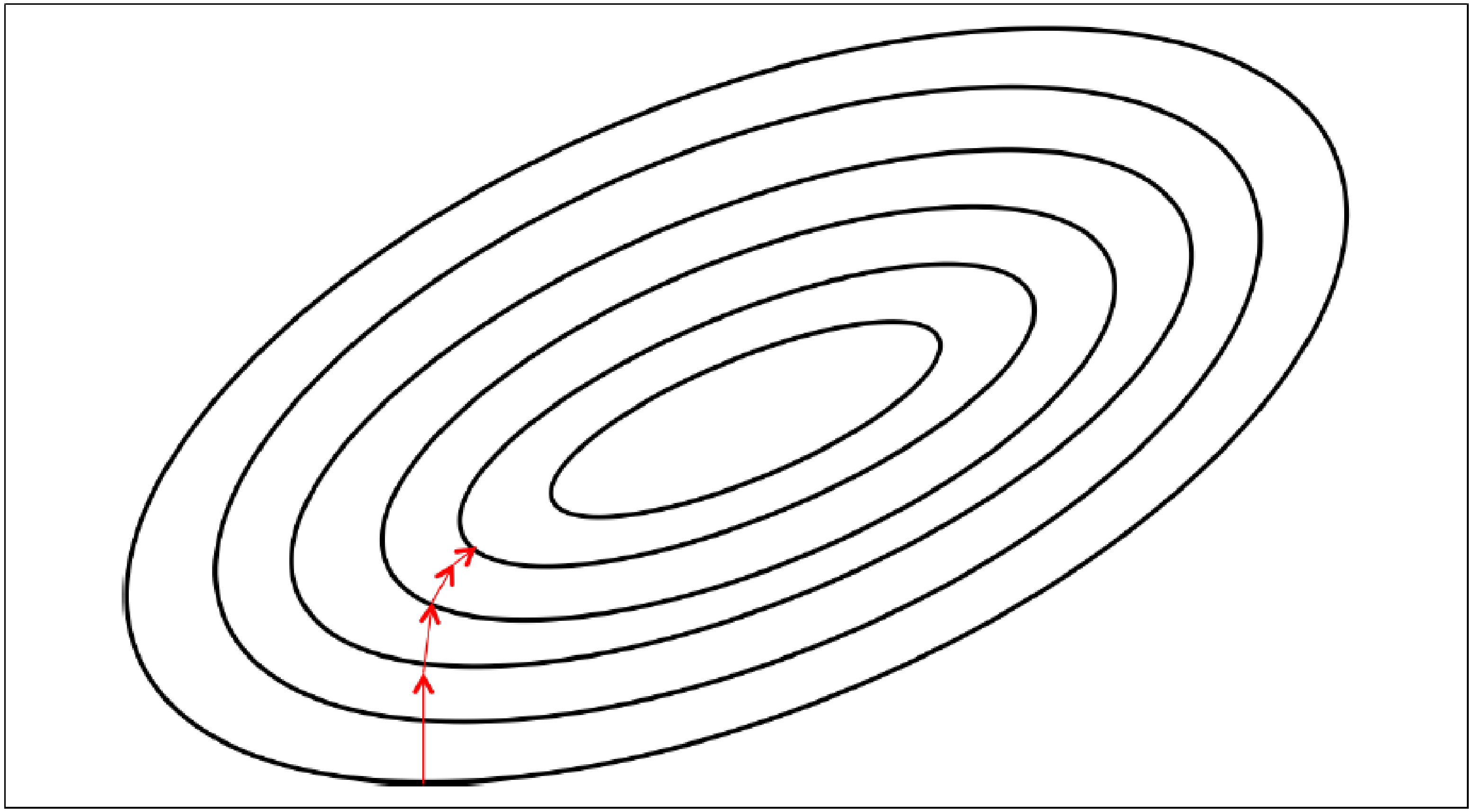


Gradient Descent



Gradient Descent

- We can also conveniently visualize this surface as a set of elliptical contours, where the minimum error is at the center of the ellipses.



Gradient Descent

- The closer the contours are to each other, the steeper the slope. In fact, it turns out that the direction of the steepest descent is always perpendicular to the contours. This direction is expressed as a vector known as the gradient.
- **By evaluating the gradient at our current position, we can find the direction of steepest descent, and we can take a step in that direction**.
- This algorithm is known as gradient descent, and we’ll use it to tackle the problem of training the deep models.

Gradient Descent

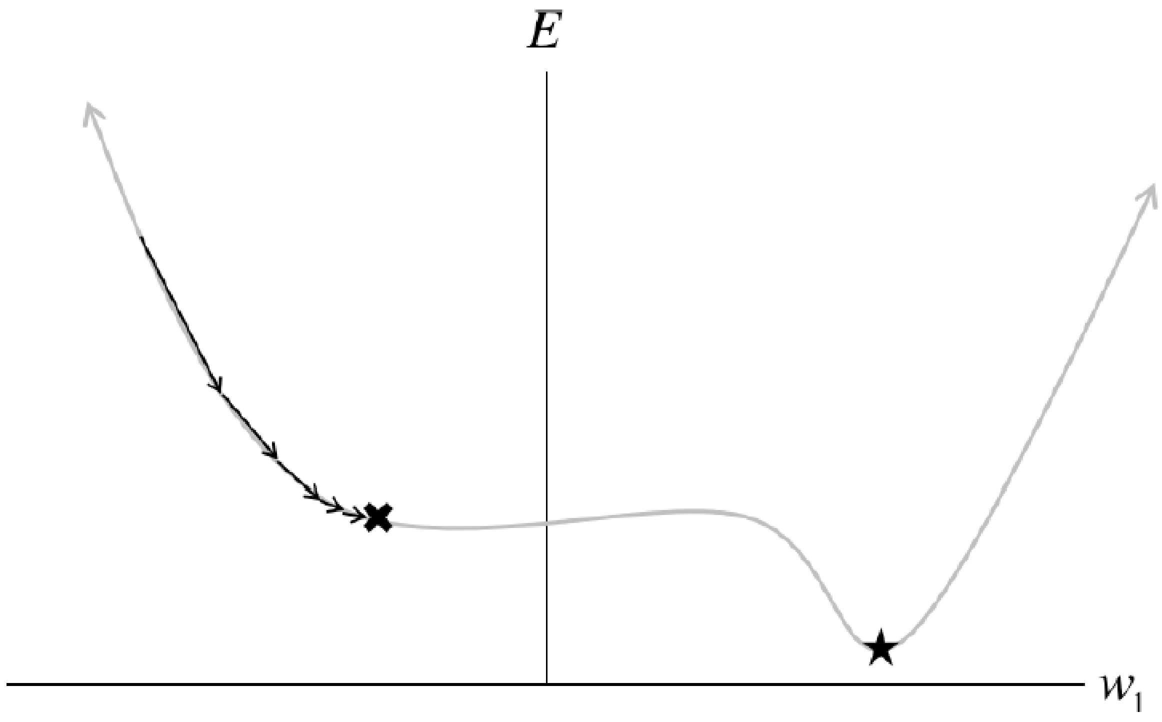
- We can update the weights as

$$w_k = w_k - \alpha_k \nabla F(w_k) = w_k - \alpha_k \frac{\partial F}{\partial w_k}$$

- Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data.
- Gradient descent computes the gradient of the cost function with respect to the model parameters using the entire training dataset in each iteration.

Gradient Descent

- The idea behind gradient descent is that we use our entire dataset to compute the error surface and then follow the gradient to take the path of steepest descent.



Stochastic Gradient Descent

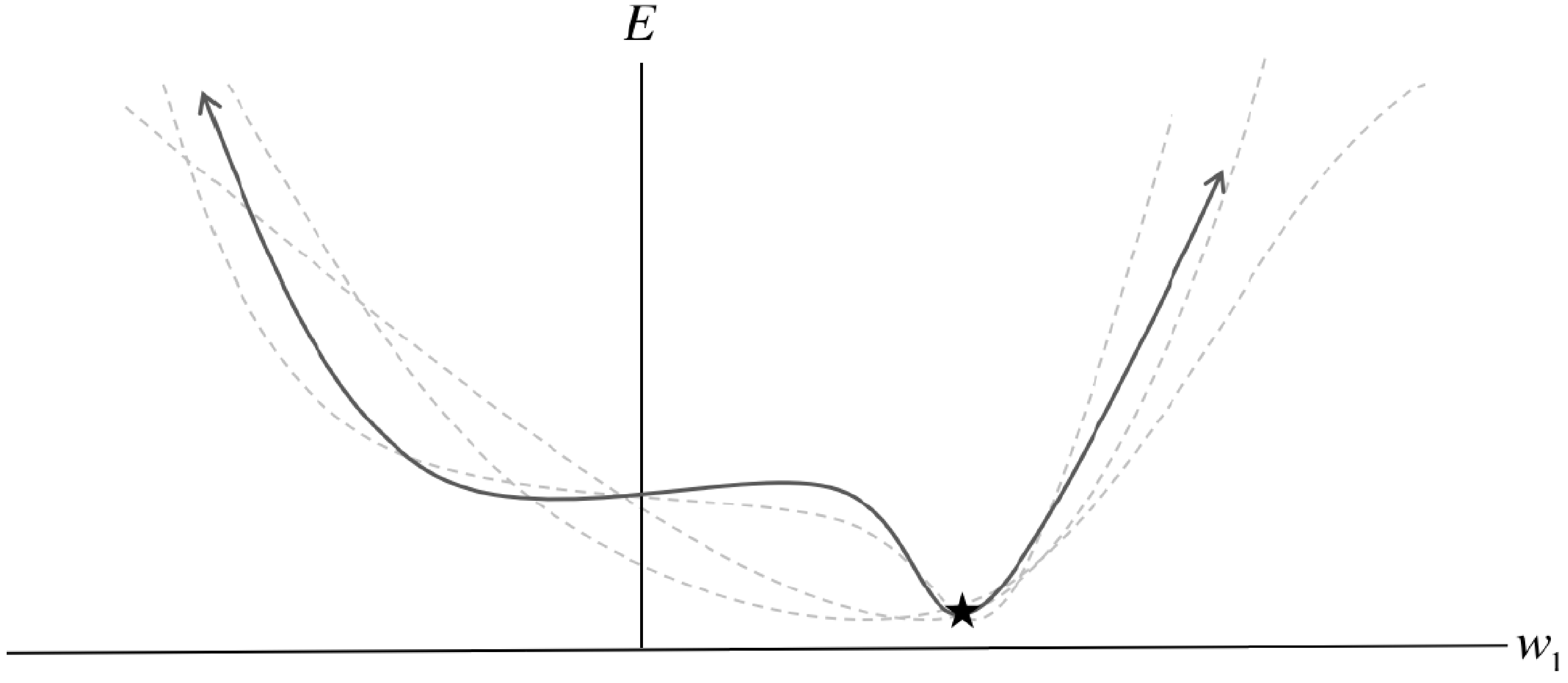
- Gradient descent computes the gradients of the entire training examples in every step and every iteration. This are lots of computations, and they take time to converge.
- Stochastic gradient descent (SGD) overcomes these problems and computes the gradients of a small subset of a training set that can easily fit in the memory.
- Generally, the weight updates in SGD are computed for a few training examples as opposed to a single example because **this reduces the variances in the weights that lead to stable convergences.**

Stochastic Gradient Descent(SGD)

- Stochastic gradient descent computes the gradient using only a single training example or a small subset of examples in each iteration.
- At each iteration, our error surface is estimated only with respect to a single example. This approach is illustrated by Figure, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.

Stochastic Gradient Descent(SGD)

- Here instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.



Stochastic Gradient Descent(SGD)

- The steepest descent algorithm for the stochastic gradient descent is

$$w_{i,j}^{m}(k+1) = w_{i,j}^{m}(k) - \alpha \frac{\partial \tilde{F}}{\partial w_{i,j}^{m}}$$

- The major pitfall of stochastic gradient descent, however, is that looking at the error incurred one example at a time may not be a good enough approximation of the error surface. This, in turn, could potentially make gradient descent take a significant amount of time.

Stochastic Gradient Descent(SGD)

- One way to combat this problem is using **mini-batch gradient descent.** In mini-batch gradient descent, at every iteration, we compute the error surface with respect to some subset of the total dataset (instead of just a single example). This subset is called a mini-batch.
- Here we have to control two hyper parameters:

  the learning rate, mini-batch size.
- Mini-batches strike a balance between the efficiency of batch gradient descent and the local-minim avoidance afforded by stochastic gradient descent.

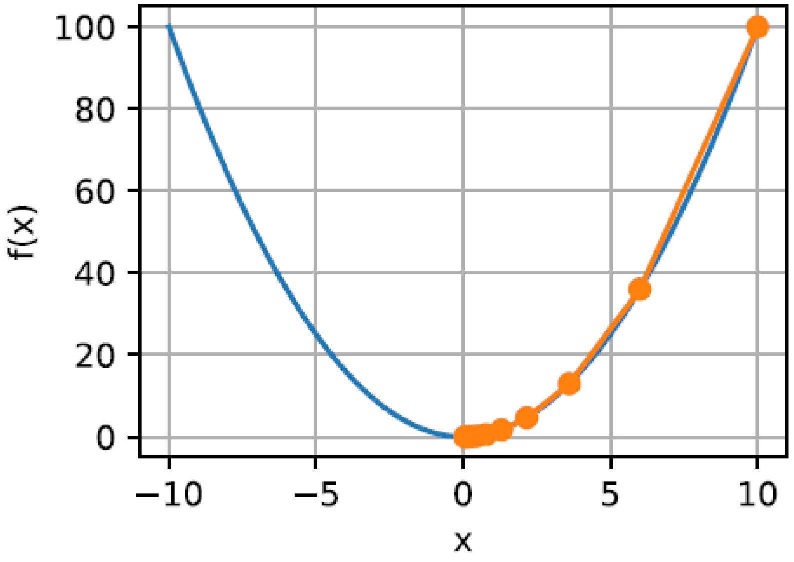
Stochastic Gradient Descent(SGD)

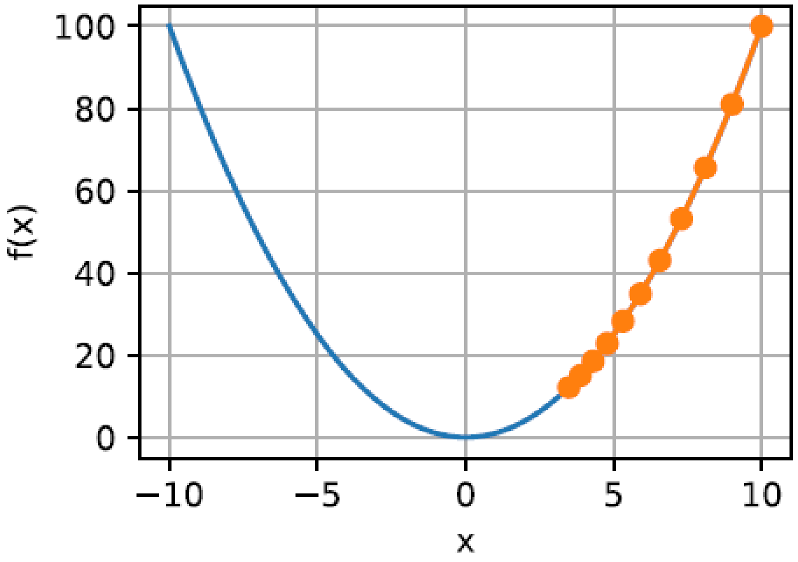
```
•   import numpy as np
•   import torch
•   import matplotlib.pylab as plt

•   def f(x): # Objective function
•       return x ** 2
•   def f_grad(x): # Gradient (derivative) of the objective function
•       return 2 * x

•   def gd(eta, f_grad):
•       x = 5.0
•       results = [x]
•       for i in range(5):
•           x -= eta * f_grad(x)
•           results.append(float(x))
•       print(f'epoch 10, x: {x:f}')
•       return results

•   results = gd(0.01, f_grad)
```

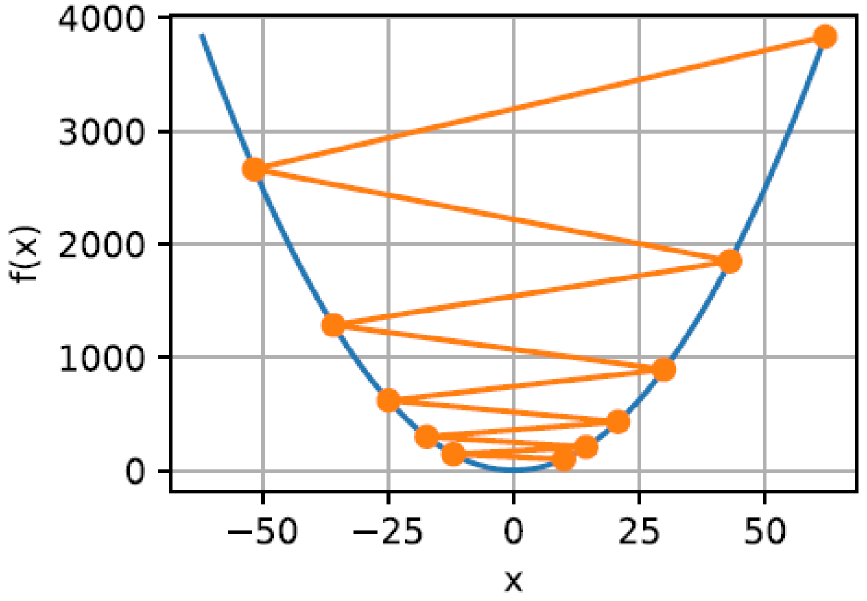
Stochastic Gradient Descent(SGD)

- import numpy as np
- from matplotlib.pyplot import plot
- results = gd(0.2, f_grad)
- n = max(abs(min(results)), abs(max(results)))
- xvals = np.linspace(-n,n,6) #100 points from 0 to 6 in ndarray
- yvals = list(map(f, xvals)) #evaluate f for each point in xvals
- xvals1 = np.linspace(-n,0,6)
- yvals1=list(map(f, results)) #evaluate f for each point in xvals
- plot(xvals, yvals)
- plot(xvals1, yvals1,color='orange', linestyle='-', marker='o')
- plt.show()
- plt.figure()

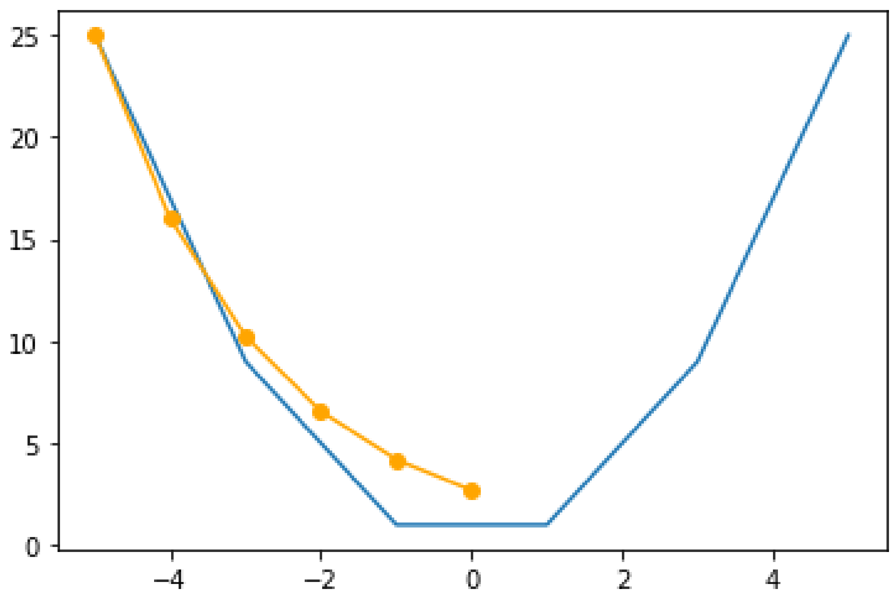
Different learning rates of SGD

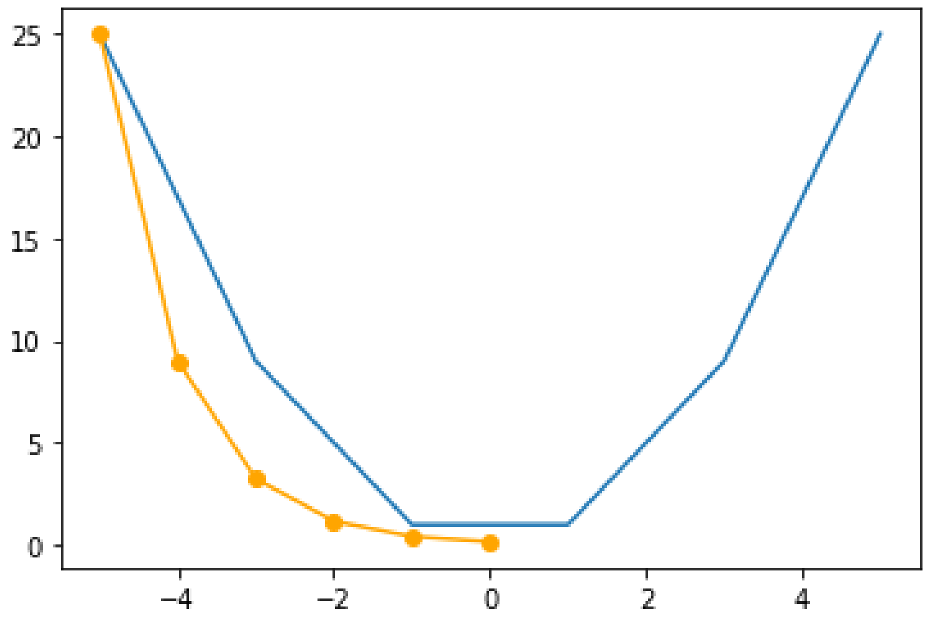


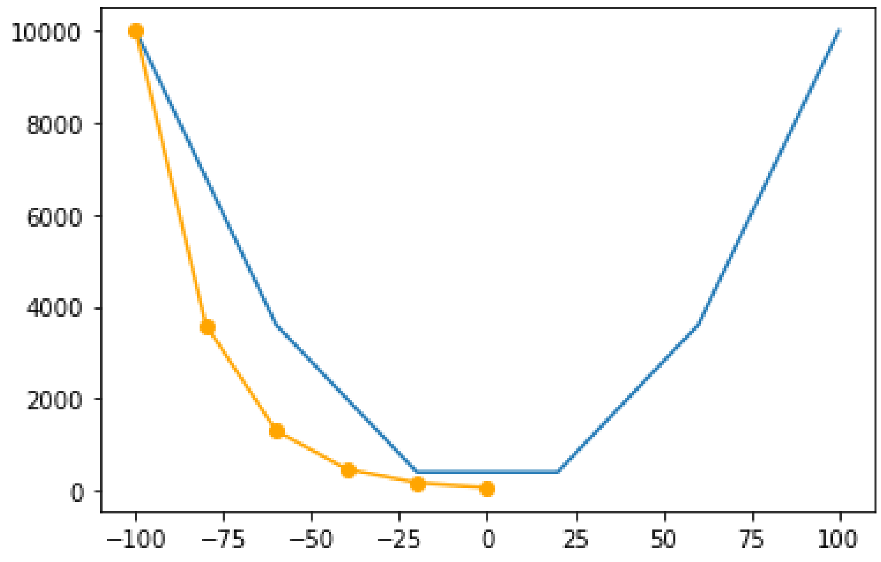


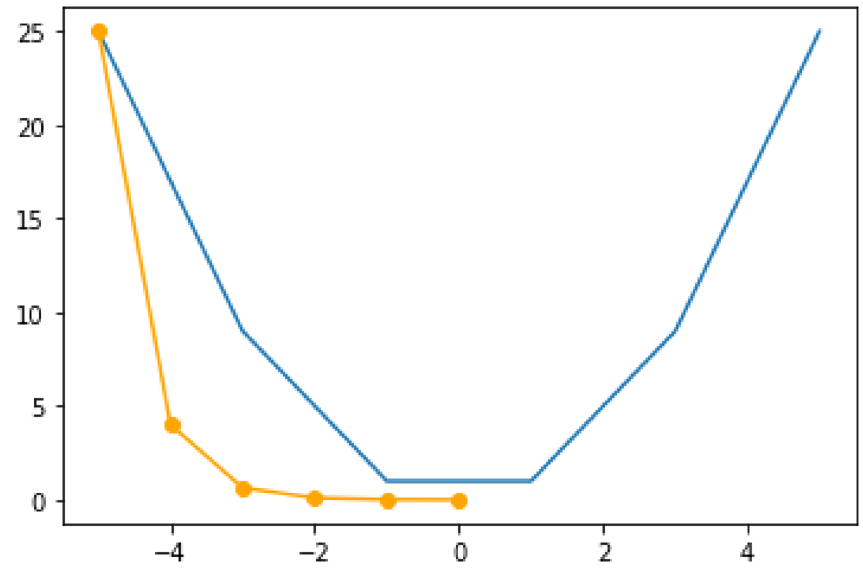


Different learning rates of SGD









The Challenges with Gradient Descent

- Local minima
- Flat Regions in the Error Surface
- When the Gradient Points in the Wrong Direction

Momentum-Based Optimization

- Fundamentally, the problem of an ill-conditioned Hessian matrix manifests itself in the form of gradients that fluctuate wildly.
- One way to think about how we might tackle this problem is by investigating how a ball rolls down a hilly surface. (acceleration and velocity)
- The momentum is added to the SGD by keeping track of an exponentially weighted decay of past gradients

Momentum-Based Optimization

- we use the momentum hyper-parameter m to determine what fraction of the previous velocity to retain in the new update, and add this “memory” of past gradients to our current gradient.

$$v_i = m v_i - \alpha g_i$$

$$w_k = w_{k-1`} + v_i$$

- Momentum smooths volatility in the step sizes during a random walk using an exponentially weighted moving average.
- **Gradient descent with momentum** leverages the past gradients to calculate an exponentially weighted average of the gradients to further smoothen the parameter updates.

Gradient Descent with Momentum

- The update process can be simplified using the following equations
- First, we compute an exponentially weighted average of the past gradients as v_t

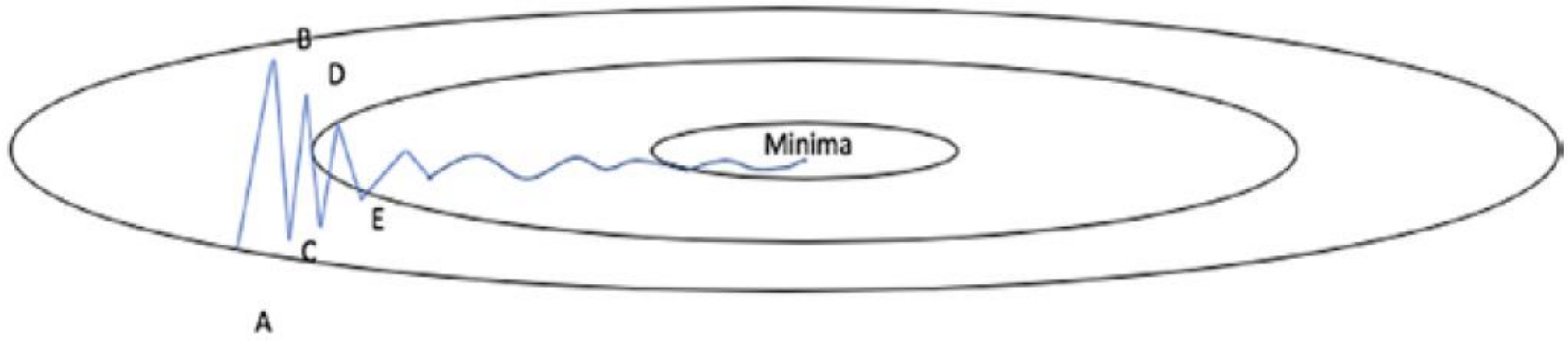
$$v_t = mv_{t-1} + \alpha g_t$$

$$v_t = mv_{t-1} + (1-m)g_t$$

- v_t : is the new weight update done at iteration **t**
- m: Momentum constant
- g_t **:** is the gradient at iteration **t**

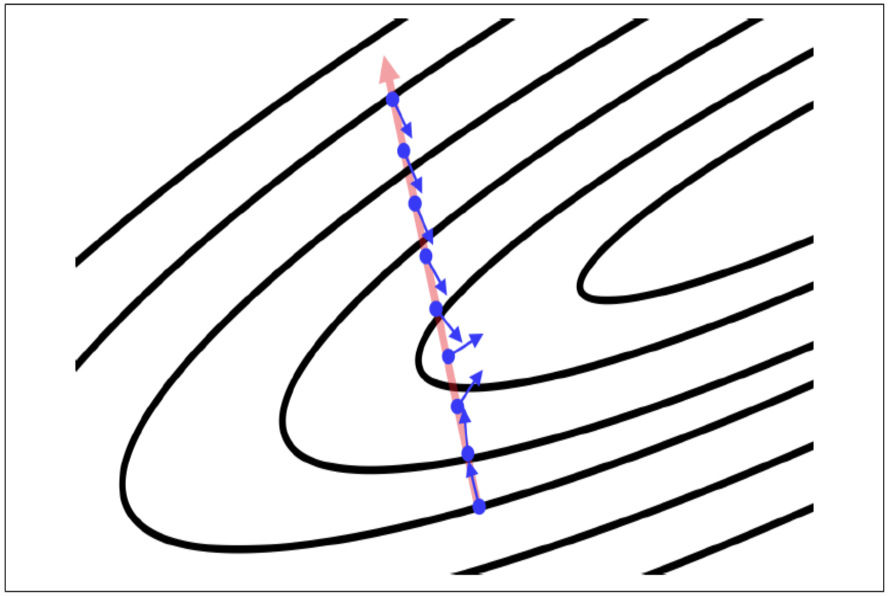
Gradient Descent with Momentum

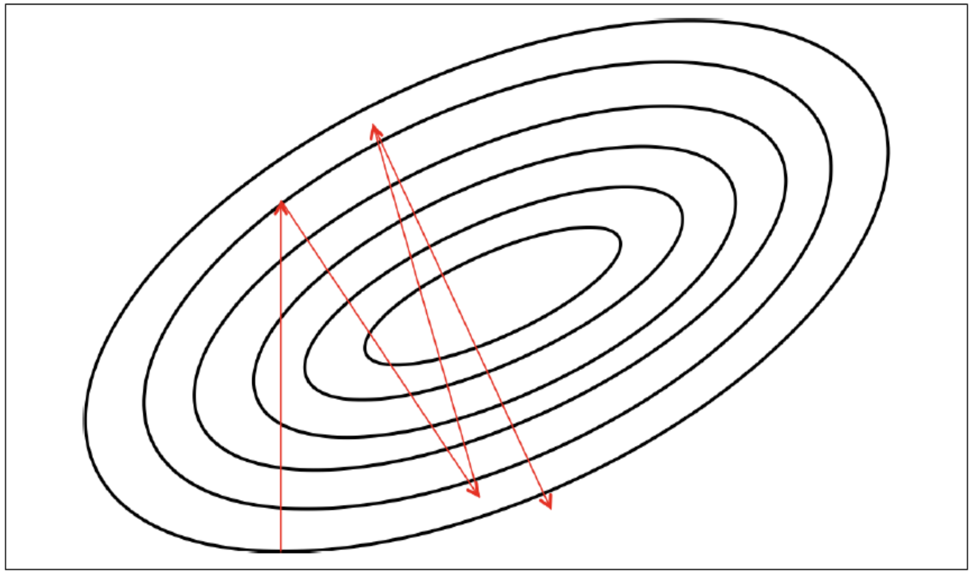
- By leveraging the exponentially weighted averages of the gradients, instead of directly using the gradients, the incremental steps are smoother and faster and thus overcome the problems with oscillating around the minima.



Learning Rate Adaptation

- A major challenge for training deep networks is appropriately selecting the learning rate.
- A learning rate that is too small doesn't learn quickly enough, but a learning rate that is too large may have difficulty converging as we approach a local minimum or region that is ill-conditioned.





Learning Rate Adaptation

- The basic concept behind learning rate adaptation is that the optimal learning rate is appropriately modified over the span of learning to achieve good convergence properties.
- **AdaGrad, RMSProp, and Adam**, are the three of the most popular adaptive learning rate algorithms.

AdaGrad—Accumulating Historical Gradients

- AdaGrad attempts to adapt the global learning rate over time using an **accumulation of the historical gradients.**
- We initialize a gradient accumulation vector $r_0 = 0$
- At every step, we **accumulate the square of all the gradient parameters** as follows

$$\mathbf{r}_i = \mathbf{r}_{i-1} + \mathbf{g}_i \odot \mathbf{g}_i$$

- Then we compute the update as usual, except our global learning rate is divided by the square root of the gradient accumulation vector

AdaGrad—Accumulating Historical Gradients

- The updation is given by

$$\theta_i = \theta_{i-1} - \frac{\epsilon}{\delta \oplus \sqrt{\mathbf{r}_i}} \odot \mathbf{g}$$

- Note that we add a tiny number δ (~10–7) to the denominator in order to prevent division by zero.
- Here g is the gradient vector and θ is the error function in terms of weight.

Exponentially Weighted Moving Average of Gradients(RMSprop)

- Flat regions may force AdaGrad to decrease the learning rate before it reaches a minimum. The conclusion is that simply using accumulation of gradients isn't sufficient.
- The update to the gradient accumulation vector is now as follows:

$$\mathbf{r}_i = \rho \mathbf{r}_{i-1} + (1-\rho)\mathbf{g}_i \odot \mathbf{g}_i$$

- The decay factor ρ determines how long we keep old gradients. The smaller the decay factor, the shorter the effective window.

Adam—Combining Momentum and RMSProp

- The basic idea is as follows. We want to keep track of an exponentially weighted moving average of the gradient (essentially the concept of velocity in classical momentum), which we can express as follows:

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1-\beta_1)\mathbf{g}_i$$

- This is our approximation of what we call the first moment of the gradient that is $\mathbb{E}[\mathbf{g}_i]$..

Adam—Combining Momentum and RMSProp

- Similarly to RMSProp, we can maintain an exponentially weighted moving average of the historical gradients. This is our estimation of second moment of the gradient that is $\mathbb{E}[\mathbf{g}_i \odot \mathbf{g}_i]$

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1 - \beta_2)\mathbf{g}_i \odot \mathbf{g}_i$$

- We can then use these corrected moments to update the parameter vector, resulting in the final Adam update:

Adam—Combining Momentum and RMSProp

- It is given by

$$\theta_i = \theta_{i-1} - \frac{\epsilon}{\delta \oplus \sqrt{\tilde{\mathbf{v}}_i}} \tilde{\mathbf{m}}_i$$

- Where

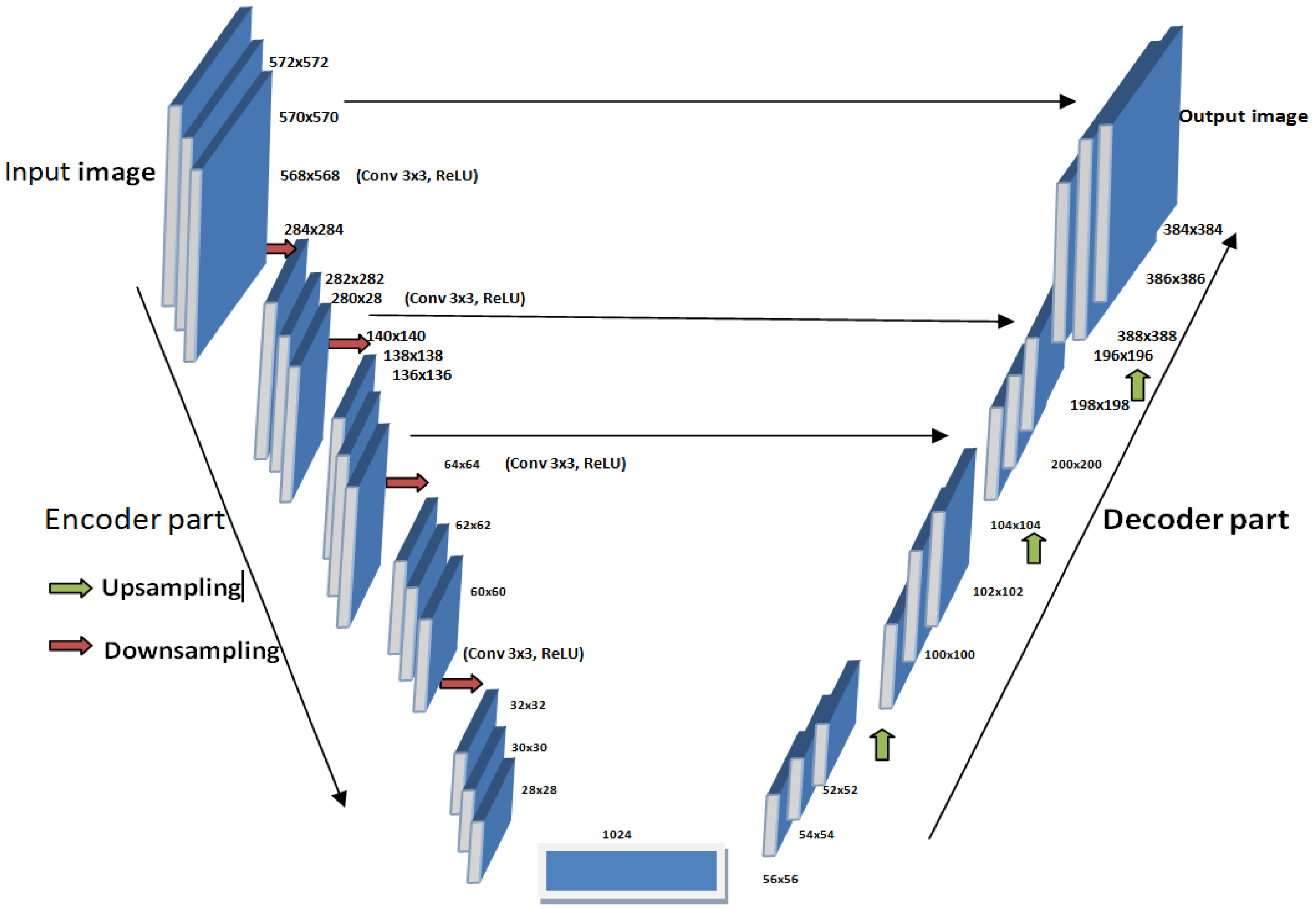
$$\tilde{\mathbf{m}}_i = \frac{m_i}{1 - \beta_1^i} \qquad \tilde{\mathbf{v}}_i = \frac{\tilde{\mathbf{v}}_i}{1 - \beta_2^i}$$

- Recently, Adam has gained popularity because of its corrective measures against the zero initialization bias (a weakness of RMSProp) and its ability to combine the core concepts behind RMSProp with momentum more effectively

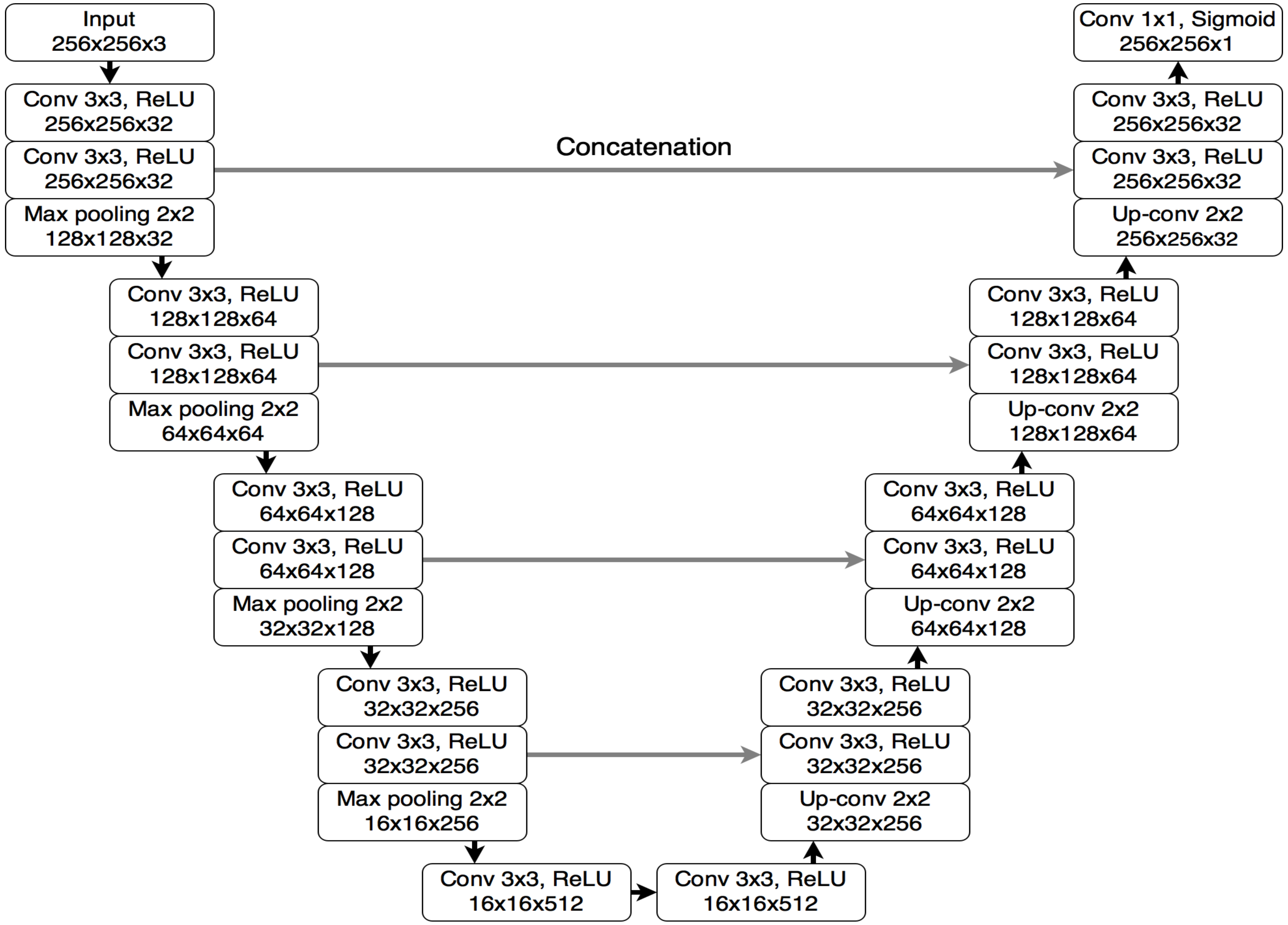
UNet Architecture

- U-Net is an encoder-decoder architecture first proposed by Ronneberger et al. (2015), that have been used to segment biomedical images.
- The network has a U-shaped architecture which consists of two paths: one is a contracting path and the other one is a symmetric expanding path.
- Contracting path has general CNN structure consists of recurring layers of convolutions, followed by a rectified linear unit (ReLU) and a maxpooling operation.
- On the other hand, expanding path facilitates accurate localization of high resolution features.

U-Net Architecture



U-Net Architecture



Transfer Learning

- It’s an incredibly powerful technique in deep learning circles called transfer learning, whereby a network trained for one task (e.g., ImageNet) is adapted to another (fish versus cats).
- Reusing the weights and parameters from a pre-trained model to train a new model is called transfer learning.
- One way to quickly get results (and often also get by with much less data) is to start not from random initializations but from a network trained on some task with related data. This is called transfer learning.

Transfer Learning for Image Classification using Torch vision

- Why would you do this?
- It turns out that an architecture trained on Image-Net already knows an awful lot about images, and in particular, quite a bit about whether something is a cat or a fish (or a dog or a whale).
- When pretrained is true , the weights of the algorithm are already tuned for a particular ImageNet classification problem of predicting 1,000 different categories, which include cars, ships, fish, cats, and dogs are utilized in the new model.

Transfer Learning

- For example, suppose a model is trained for image classification on the ImageNet dataset. In that case, we can take this model and “re-train” it to recognize classes it was never trained to recognize in the first place.
- you can make direct use of a well-trained model by freezing the parameters, changing the output layer, and fine-tuning the weights.

Transfer Learning

- **Transfer learning via feature extraction**: We remove the FC layer head from the pre-trained network and replace it with a softmax classifier. This method is super simple as it allows us to treat the pre-trained CNN as a feature extractor and then pass those features through a Logistic Regression classifier.

Transfer Learning

- **Transfer learning via fine-tuning**: When applying fine-tuning, we again remove the FC layer head from the pre-trained network, and initialized a FC layer head on top of the original body of the network.
- The weights in the body of the CNN are frozen, and then we train the new layer head.
- We may then choose to unfreeze the body of the network and train the entire network.

Transfer Learning

- These weights are stored and shared with the model that we are using for the use case.
- When we say to use pre-trained weights we mean use the layers which hold the representations to identify cats but discard the last layer (dense and output) and instead add fresh dense and output layers with random weights. So our predictions can make use of the representations already learned.

Transfer Learning

- Algorithms tend to work better when started with fine-tuned weights, rather than when started with random weights. So, for our use case, we'll start with pretrained weights.
- The ResNet algorithm cannot be used directly, as it is trained to predict one of the 1,000 categories.
- For our use case, we need to predict only one of the two categories of dogs and cats. To achieve this, we take the last layer of the ResNet model, which is a linear layer, and change the output features to 4.

Transfer Learning

- These two major transfer learning scenarios
- 1- **Finetuning the ConvNet**: Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.
- **2-ConvNet as fixed feature extractor**: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

Finetuning the ConvNet

- Load a pretrained model and reset final fully connected layer.
- Observe that all parameters are being optimized.

ConvNet as fixed feature extractor

- Here, we need to freeze all the network except the final layer.
- We need to set (requires_grad = False) to freeze the parameters so that the gradients are not computed in backward().
- Parameters of newly constructed modules have requires_grad=True by default.
- Observe that only parameters of final layer are being optimized as opposed to before.

Transfer Learning

The three major Transfer Learning scenarios:

1- ConvNet as fixed feature extractor.

2- Fine-tuning the ConvNet.

3- Pretrained models.

Transfer Learning

- **ConvNet as fixed feature extractor**: Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer’s outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset.
- The last layer can be trained with a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset according to the application.

Transfer Learning

- **Fine-tuning the ConvNet**.
- The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the back propagation.
- It is possible to fine-tune all the layers of the ConvNet, or it’s possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network.

Transfer Learning

- **Pretrained models:**
- Use others pretrained models.

Transfer Learning

- **When and how to fine-tune?**
- How do you decide what type of transfer learning you should perform on a new dataset?
- **New dataset is small and similar to original dataset**.
- Since the data is small, it is not a good idea to fine-tune the ConvNet due to overfitting concerns. Since the data is similar to the original data, we expect higher-level features in the ConvNet to be relevant to this dataset as well. Hence, the best idea might be to train a linear classifier on the CNN codes.

Transfer Learning

- **New dataset is large and similar to the original dataset**. Since we have more data, we can have more confidence that we won’t overfit if we were to try to fine-tune through the full network.
- **New dataset is small but very different from the original dataset.** Since the data is small, it is likely best to only train a linear classifier. Since the dataset is very different, it might not be best to train the classifier form the top of the network, which contains more dataset-specific features. Instead, it might work better to train the SVM classifier from activations somewhere earlier in the network.

Transfer Learning

- **New dataset is large and very different from the original dataset.** Since the dataset is very large, we may expect that we can afford to train a ConvNet from scratch. However, in practice it is very often still beneficial to initialize with weights from a pretrained model.