1. Fill in the input shape (XXXaXXX)

The input shape for a grayscale image of height 224 and width 224 is: input_shape=(224,224,1)\text{input_shape} = (224, 224, 1)input_shape=(224,224,1)pythonCopy codemodel.add(Conv2D(input_shape=(224, 224, 1), filters=64, kernel_size=(3, 3), padding="same", activation="relu"))

B. Fill in the conv layer type to be used: (XXXbXXX) The layer that is used next should be a Conv2D layer:pythonCopy codemodel.add(Conv2D(filters=128, kernel_size=(3, 3), padding="same", activation="relu"))

C. Fill in the Loss function to be used: (XXXcXXX) Since this is a classification problem with 10 classes, the loss function to use is categorical_crossentropy:pythonCopy codemodel.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

D. At what level we will have to use the Flatten Layer The Flatten layer is used to convert the 2D matrix into a 1D vector before feeding it into the Dense (fully connected) layer. This should be done at Level 3, just before the Dense layer:pythonCopy code# Before adding Dense layers, we add Flattenmodel.add(Flatten())

E. Calculate the number of neurons in the final dense Layer The final dense layer should have neurons equal to the number of output classes, which is 10:pythonCopy codemodel.add(Dense(10))

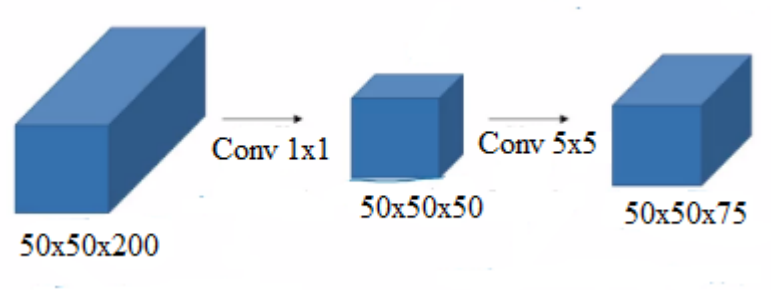
F. Calculate the number of parameters in layer 4 labelled above The layer 4 mentioned is:pythonCopy codemodel.add(Conv2D(filters=128, kernel_size=(3, 3), padding="same", activation="relu"))

To calculate the number of parameters:Previous layer output shape: (height,width,channels)=(112,112,64)(height, width, channels) = (112, 112, 64)(height,width,channels)=(112,112,64) (since it follows a max pooling layer with pool size (2, 2))Filters: 128Kernel size: 3×33 \times 33×3Input channels: 64Formula for parameters in Conv2D: Number of parameters=(kernel height×kernel width×input channels+bias)×number of filters\text{Number of parameters} = (\text{kernel height} \times \text{kernel width} \times \text{input channels} + \text{bias}) \times \text{number of filters}Number of parameters=(kernel height×kernel width×input channels+bias)×number of filtersNumber of parameters=(3×3×64+1)×128=(576+1)×128=577×128=73856\text{Number of parameters} = (3 \times 3 \times 64 + 1) \times 128 = (576 + 1) \times 128 = 577 \times 128 = 73856Number of parameters=(3×3×64+1)×128=(576+1)×128=577×128=73856So, the number of parameters in the 4th Conv2D layer is 73,856.

Summary of Answers**A**: input_shape=(224,224,1)\text{input_shape} = (224, 224, 1)input_shape=(224,224,1)**B**: Conv2D\text{Conv2D}Conv2D**C**: categorical_crossentropy\text{categorical_crossentropy}categorical_crossentropy**D**: Level 3 (before adding Dense layers)**E**: Number of neurons in the final dense layer is 10**F**: Number of parameters in layer 4 is 73,856

Question – 2

ii.



A. In the following GoogLeNet step, 1x1 operators are first used to process 50x50 images of depth 200 and produce 50x50 images of depth 50, and then 5x5 operators are used to output final 50x50 images of depth 75. a) What is the padding size used in the first step and padding size in the last step? b) How many multiplication operations are needed in the first and final steps? B. How is the vanishing gradient problem mitigated in deep layers of Resnet? C. In faster R-CNN for object detection and classification,

Part A: GoogLeNet Step

a) Padding Size in the First and Last Steps

First Step (Conv 1x1)

Input dimensions: 50×50×200

Filter size: 1×1

Output dimensions: 50×50×50

Since the input and output spatial dimensions remain the same (from 50×50 to 50×50), padding is likely **0**. This means no padding is applied since the 1x1 convolution doesn't alter spatial dimensions.

Last Step (Conv 5x5)

Input dimensions: 50×50×50

Filter size: 5×5

Output dimensions: 50×50×75

For the spatial dimensions to remain 50×50 after applying a 5×5 filter, padding is required. The padding size pp can be determined using the formula:
Output size=(Input size−Filter size+2p)/Stride+1

Plugging in the values:50=(50−5+2p)/1+1

Solving for pp:49=45+2p⇒2p=4⇒p=249 = 45 + 2p \quad \Rightarrow \quad 2p = 4 \quad \Rightarrow \quad p = 249=45+2p⇒2p=4⇒p=2

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

Padding size in the last step is **2**.

b) Number of Multiplication Operations

First Step (Conv 1x1)

Input dimensions: 50×50×200

Filter size: 1×1×200

Number of filters: 50 (output depth)

The number of multiplication operations in this step:

Total operations=Output dimensions×Filter size =

(50×50×50)×(1×1×200)=50×50×50×200=25,000,000

Last Step (Conv 5x5)

Input dimensions: 50×50×50

Filter size: 5×5×50

Number of filters: 75 (output depth)

The number of multiplication operations in this step:

Total operations=Output dimensions×Filter size =(50×50×75)×(5×5×50)

=50×50×75×25×50=468,750,000

Part B: Vanishing Gradient Problem in ResNet

The vanishing gradient problem, where gradients become too small for effective learning in deep networks, is mitigated in ResNet through the use of **residual connections** (or skip connections). These connections allow the gradient to bypass certain layers and be directly backpropagated to earlier layers, maintaining its magnitude. This enables deep networks to learn effectively without suffering from diminishing gradient magnitudes, as the gradient can flow more easily through the network.

Part C: Faster R-CNN for Object Detection and Classification

Faster R-CNN is an improvement over the original R-CNN for object detection. It integrates a **Region Proposal Network (RPN)** to propose regions of interest (ROIs) directly from the convolutional feature maps. This is much faster than traditional methods like selective search. The RPN generates a set of candidate bounding boxes, which are then used for both classification and bounding box regression. The architecture is end-to-end, meaning that both the RPN and the object detection network are trained together, resulting in faster and more accurate object detection. The RPN shares the same convolutional layers as the object detection network, allowing for efficient computation.

Q-3 Refer to the following recurrent neural network with one input unit x, two hidden units h, and one output unit y. All of the units use the hard threshold activation function, i.e., output is 1 if total weighted input is >=bias, else 0.

Note, at time t, ht = step(Wht-1 + Ux - bh) and yt = step(vht - by).

U, v and W, by and bhT are given by v = [1 -1] and by = 0.5 bhT=[−0.5 0.5]W=[1 -11 -1]U=[-1 -1]

A. The inputs are presented to this RNN as binary sequences from left to right. What will be the output string for the input string {0, 1, 0, 0}. Assume initial hidden state values are 0's.

B. What is the key limitation of RNN overcome by the long short term memory (LSTM) model?

C. Suppose an RNN is used to predict next character in the input string 'abracadabra' (no upper case needs to be considered). If one hot encoding is used for representing input and output, how many nodes will be needed in the input/output layer? If characters are presented from left to right, write the first input and target output vectors.

Part A: Output String for Input {0,1,0,0}Given the RNN configuration:Initial hidden state: h 0 =[0,0]Input vector xWeight matrices:Bias terms:

Output step by step:

For t=1 (input $x1$=0):

Hidden state h 1 =step($Wh0+U\cdot x1-bh$)

$h1$=step([1−1 1−1]·[0 0]+[−1−1]·0−[−0.50.5])

h 1 =step([11 −1−1]·[00]+[−1−1]·0−[−0.50.5])$h1$=step([00]−[−0.50.5])

=step([0.5−0.5])h 1

=step([00]−[−0.50.5])

=step([0.5−0.5])

$h1$=[10]

h 1 =[0 Output $y1$:$y1$=step($v\cdot h1-by$)y 1 =step(v·h 1 −b y)$y1$=step([1−1]·[10]−0.5)y 1 =step([1 −1]·[10 −0.5)$y1$=step(1−0.5)=step(0.5)=1y 1 =step(1−0.5)=step(0.5)=1

For t=2t=2 (input $x2$=1x 2 =1):Hidden state $h2$h 2 :$h2$=step($W\cdot h1+U\cdot x2-bh$)h 2 =step(W·h 1 +U·x 2 −b h)$h2$=step([1−11−1]·[10]+[−1−1]·1−[−0.50.5])h 2 =step([11 −1−1]·[10]+[−1−1]·1−[−0.50.5])$h2$=step([11]+[−1−1]−[−0.50.5])h 2 =step([11]+[−1−1]−[−0.50.5])$h2$=step([0.5−0.5])h 2 =step([0.5−0.5])$h2$=[10]h 2 =[10]Output $y2$y 2 :$y2$=step($v\cdot h2-by$)y 2 =step(v·h 2 −b y)$y2$=step(1−0.5)=1y 2 =step(1−0.5)=1

For t=3t=3 (input x3=0x 3 =0):Hidden state h3h 3 :h3=step($W \cdot h2+U \cdot x3-bh$)h 3 =step(W·h 2 +U·x 3 −b h)h3=step([1−11−1]·[10]+[−1−1]·0−[−0.50.5])h 3 =step([11 −1−1]·[10]+[−1−1]·0−[−0.50.5])h3=step([11]−[−0.50.5])h 3 =step([11]−[−0.50.5])h3=step([1.50.5])h 3 =step([1.50.5])h3=[11]h 3 =[11]Output y3y 3 :y3=step(1−1−0.5)=step(−0.5)=0y 3 =step(1−1−0.5)=step(−0.5)=0

For t=4t=4 (input x4=0x 4 =0):Hidden state h4h 4 :h4=step($W \cdot h3+U \cdot x4-bh$)h 4 =step(W·h 3 +U·x 4 −b h)h4=step([1−11−1]·[11]+[−1−1]·0−[−0.50.5])h 4 =step([11 −1−1]·[11]+[−1−1]·0−[−0.50.5])h4=step([00]−[−0.50.5])h 4 =step([00]−[−0.50.5])h4=[10]h 4 =[10]Output y4y 4 :y4=step(1−0.5)=1y 4 =step(1−0.5)=1Final Output String: {1,1,0,1}{1,1,0,1}

Part B: Limitation of RNN Overcome by LSTM

The key limitation of traditional RNNs is the vanishing gradient problem, where gradients become extremely small during backpropagation through time, especially in deep networks or for long sequences. This makes it difficult for RNNs to learn long-term dependencies effectively. LSTM (Long Short-Term Memory) networks address this by introducing mechanisms like gates (input gate, forget gate, output gate) and memory cells that control the flow of information. These mechanisms allow LSTM networks to retain and use information over long periods, enabling them to learn long-term dependencies much better than standard RNNs.

Part C: RNN for Character Prediction in "abracadabra"Input/Output Layer Size:If we use one-hot encoding for the input and output, the number of nodes in both the input and output layers would correspond to the number of unique characters in the string "abracadabra".Unique characters: {a, b, r, c, d}

Total unique characters: 5

So, the input/output layer will have 5 nodes.

First Input and Target Output Vectors:

For the string "abracadabra", presented from left to right:First input vector corresponds to the character 'a'.One-hot encoding for 'a': [1, 0, 0, 0, 0]

Target output vector corresponds to the next character 'b'.One-hot encoding for 'b': [0, 1, 0, 0, 0]Thus, the first input vector is [1,0,0,0,0] and the target output vector is [0,1,0,0,0][0,1,0,0,0].

A. **For what choice of activation functions in hidden and output nodes, autoencoder weights represent the principal component vectors?**

B. Gradient ascent is used in training the generative module of the generative adversarial network (GAN). True/False

C. What types of convolution are used in the pooling layers of CNN based GANs?

D. Why is KL divergence used for regularization in variational autoencoders?

E. A dataset X is given. During training, how are the input and output data specified for a denoising autoencoder?

Let's address each of these questions one by one.

A. Activation Functions for Autoencoder Weights to Represent Principal Component Vectors

For an autoencoder's weights to represent the principal component vectors (which is essentially performing Principal Component Analysis, PCA), the following conditions are required:

- **Linear Activation Function**: Both the hidden layer and the output layer should use linear activation functions. This means that the autoencoder is just a linear model without any non-linear transformations.
- **Squared Error Loss Function**: The autoencoder should be trained using the mean squared error (MSE) loss function.

When the hidden layer and output layer are linear, the autoencoder essentially learns to project the input data onto a lower-dimensional space, capturing the directions of maximum variance in the data—these are the principal components.

B. Gradient Ascent in Training the Generative Module of GAN

False. In Generative Adversarial Networks (GANs), the generative module (or generator) is trained using **gradient descent**, not gradient ascent. The goal of the generator is to minimize a loss function (often related to how well it can "fool" the discriminator into classifying generated samples as real). Gradient descent is used to find the set of parameters that minimize this loss function.

C. Types of Convolution in Pooling Layers of CNN-Based GANs

Pooling layers in CNN-based GANs typically do not use any convolutions themselves; they perform downsampling. However, when discussing the types of convolutions associated with these layers:

- **Strided Convolutions**: Instead of traditional pooling, many GAN architectures use strided convolutions (in the generator, these might be transposed convolutions, also called deconvolutions) to reduce the spatial dimensions (in the discriminator) or to upsample the image (in the generator).
- **Average Pooling or Max Pooling**: In some cases, traditional pooling operations like average pooling or max pooling are used in the discriminator to reduce the spatial dimensions of the feature maps.

D. Why is KL Divergence Used for Regularization in Variational Autoencoders (VAEs)?

Kullback-Leibler (KL) Divergence is used in Variational Autoencoders (VAEs) to measure the difference between the learned distribution q(z|x)q(z|x)q(z|x) (the encoder's output) and the prior distribution p(z)p(z)p(z), which is typically a standard normal distribution. This regularization term in the loss function encourages the encoder to learn a distribution that is close to the prior, ensuring that the latent space is well-behaved and that the generative model can produce realistic samples even when generating from arbitrary points in the latent space.

Mathematically, the loss function in a VAE is composed of two terms:

1. **Reconstruction Loss**: Measures how well the autoencoder reconstructs the input data.
2. **KL Divergence**: Regularizes the distribution of the latent variables to be close to a standard normal distribution.

This combination allows the VAE to generate new data that is similar to the training data while maintaining a structured and continuous latent space.

E. Input and Output Data in Training a Denoising Autoencoder

In a **denoising autoencoder**, the goal is to train the network to remove noise from the input data.

- **Input Data**: The input to the denoising autoencoder is a **noisy version** of the original dataset X'X'X'. This noisy data is typically generated by adding some form of noise (e.g., Gaussian noise, salt-and-pepper noise) to the clean data.
- **Output Data**: The output data is the **original clean dataset** XXX. The autoencoder is trained to reconstruct the clean version of the data from its noisy counterpart.

During training, the autoencoder learns to map noisy inputs to their clean versions, effectively learning a denoising function.

4.

SL.No		Activation Shape	Activation Size	# Parameters
1.	Input Layer:	(32, 32, 3)	3072	0
2.	CONV1 (f=5, s=1)	(28, 28, 8)	6272	608
3.	POOL1	(14, 14, 8)	1568	0
4.	CONV2 (f=5, s=1)	(10, 10, 16)	1600	3216
5.	POOL2	(5, 5, 16)	400	0
6.	FC3	(120, 1)	120	48120
7.	FC4	(84, 1)	84	10164
8.	Softmax	(10, 1)	10	850

The first **input layer** has no parameters. Input layer has nothing to learn, at it's core, what it does is just provide the input image's shape. So no learnable parameters here. Thus number of **parameters = 0**.Parameters in the second **CONV1(filter shape =5*5, stride=1) layer is: ((shape of width of filter*shape of height filter*number of filters in the previous layer+1)*number of filters) = (((5*5*3)+1)*8) = 608.**The third **POOL1 layer** has no parameters. This has got no learnable parameters because all it does is calculate a specific number, no backprop learning involved! Thus number of **parameters = 0**.Parameters in the fourth **CONV2(filter shape =5*5, stride=1) layer is**: **((shape of width of filter * shape of height filter * number of filters in the previous layer+1) * number of filters) = (((5*5*8)+1)*16) = 3216.**5. The fifth **POOL2 layer** has no parameters. You know why.6. Parameters in the Sixth **FC3 layer is((current layer c*previous layer p)+1*c) = 120*400+1*120= 48120.**7. Parameters in the Seventh **FC4 layer is: ((current layer c*previous layer p)+1*c) = 84*120+1* 84 = 10164.**8. The Eighth **Softmax layer has ((current layer c*previous layer p)+1*c) parameters = 10*84+1*10 = 850.**

Explain the following concepts in machine learning A. Xavier initialization. B. Style GAN C. Re-parameterization and its application

Let's break down each of these concepts in machine learning:

A. Xavier Initialization

Xavier Initialization (also known as Glorot Initialization) is a method for initializing the weights of neural networks. It is particularly useful in ensuring that the weights start with the right scale, which can help in preventing vanishing or exploding gradients during training.

- Purpose: When training a deep neural network, the scale of the initial weights can greatly affect how quickly and effectively the network converges. If weights are initialized too large, activations can grow exponentially through the layers, leading to exploding gradients. If weights are too small, activations can shrink, leading to vanishing gradients.
- Formula: Xavier initialization sets the initial weights WWW using a random distribution with a specific variance:

$$W \sim \mathcal{N}(0, \frac{2}{n_{\text{in}} + n_{\text{out}}})$$

or

$$W \sim U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right)$$

where n in is the number of input units in the weight tensor, and N out is the number of output units.

- Effect: By balancing the variance of weights across layers, Xavier initialization helps maintain the scale of the gradients during backpropagation, thus facilitating more stable and faster training.

B. Style GAN

Style GAN (Generative Adversarial Network) is an advanced type of GAN architecture introduced by NVIDIA in 2018. It is particularly known for its ability to generate high-resolution images with remarkable control over the style and features of the output images. Here's a breakdown of the key concepts:

- Style-Based Generator Architecture: Unlike traditional GANs where the latent vector zzz is directly fed into the input of the generator, Style GAN first maps zzz to an intermediate latent space www using a fully connected network called a mapping network. The vector www is then used to control the style at each layer of the generator through adaptive instance normalization (AdaIN).
- Control Over Style: The key innovation of Style GAN is its ability to control different aspects of the generated image (like colors, textures, and patterns) at different scales. For example, coarse styles can control high-level aspects like pose or hair style, while fine styles can control details like texture or lighting.
- Progressive Growing: Style GAN typically employs progressive growing, where the model starts by generating low-resolution images and gradually increases the resolution as training progresses. This helps the model learn to generate coherent images more effectively.
- Applications: Style GAN is widely used in tasks requiring high-quality image generation, such as creating photorealistic portraits, art, or synthetic datasets for training other machine learning models.

C. Re-parameterization and Its Application

Re-parameterization is a technique commonly used in machine learning, particularly in the training of Variational Autoencoders (VAEs). The purpose of re-parameterization is to allow backpropagation through stochastic nodes, which is essential for optimizing models that involve random variables.

- Problem with Stochastic Nodes: In a VAE, the encoder outputs a distribution (mean and variance) rather than a single point. Sampling directly from this distribution to obtain a latent variable zzz introduces randomness, which makes it impossible to directly compute gradients during backpropagation.
- Re-parameterization Trick: The re-parameterization trick solves this by expressing the random variable zzz as a deterministic function of two components: the deterministic part (mean and variance) and a random noise drawn from a known distribution (usually a standard normal distribution). This allows gradients to flow through the deterministic part, making the model trainable.

For example, if zzz is drawn from a normal distribution N(μ,σ2)\mathcal{N}(\mu, \sigma^2)N(μ,σ2), we can re-parameterize as:

z=μ+σ×ϵz = \mu + \sigma \times \epsilonz=μ+σ×ϵ

where ϵ∼N(0,1)\epsilon \sim \mathcal{N}(0, 1)ϵ∼N(0,1) is a noise term.

- Application: The re-parameterization trick is primarily used in Variational Autoencoders (VAEs), enabling them to be trained via gradient-based optimization methods. This allows VAEs to learn a continuous and structured latent space from which new samples can be generated, making them powerful tools for generative modeling.

These concepts are crucial in understanding the mechanisms behind training and improving the performance of modern neural networks and generative models.

Consider following training data points. [6] Class Green (2,2), (2,4), (4,2), (4,3), (6,9), (8,8), (11,8), (8,10), (9,10) Class Red (1,8), (3,6), (3,8), (4,11), (5,4), (5,6), (6,3), (6,6), (6,12), (7,4), (9,12) Suggest a suitable neural network architecture to obtain 100% training accuracy. Justify your answer.

To achieve 100% training accuracy for the given classification problem, we need to choose a neural network architecture that is capable of separating the two classes (Green and Red) based on the provided coordinates.

Problem Analysis

- **Input Data**: The input consists of 2D points (x, y).
- **Classes**: There are two classes, Green and Red, with 9 and 11 data points respectively.

Given that we need to achieve 100% training accuracy, the network must be powerful enough to perfectly classify all training points. This suggests that a complex decision boundary may be required, especially if the classes are not linearly separable.

Suitable Neural Network Architecture

1. **Input Layer**:
    - **Size**: 2 neurons (for the 2D coordinates xxx and yyy).

2. **Hidden Layers**:
    - **Number of Layers**: Start with 2 hidden layers.
    - **Number of Neurons**: Use around 4 to 6 neurons per layer.
    - **Activation Function**: Use ReLU (Rectified Linear Unit) as the activation function, which helps in learning non-linear decision boundaries.

Rationale: Two hidden layers with ReLU activation allow the network to learn complex, non-linear decision boundaries. The choice of 4-6 neurons per layer gives the network enough capacity to model the potentially complex interactions between the features without overfitting.

3. **Output Layer**:
    - **Size**: 1 neuron (for binary classification).
    - **Activation Function**: Sigmoid activation function to produce a probability output that can be thresholded at 0.5 for classification into Green or Red.
4. **Loss Function**:
    - **Binary Cross-Entropy**: Suitable for binary classification tasks.
5. **Optimizer**:
    - **Adam Optimizer**: A good default choice for training deep networks due to its adaptive learning rate properties.

Justification

- **Two Hidden Layers**: Generally, one hidden layer can classify linearly separable data, but to capture more complex decision boundaries, two layers provide greater flexibility.
- **4-6 Neurons per Layer**: This number is typically sufficient for a dataset of this size and complexity. It avoids underfitting (too few neurons) or overfitting (too many neurons).
- **ReLU Activation**: ReLU helps in creating complex decision boundaries due to its piecewise linear nature, making it easier for the network to approximate non-linear functions.
- **Sigmoid in Output Layer**: This allows the output to be interpreted as a probability, which is essential for binary classification tasks.

Potential Issues & Alternatives

- **Overfitting**: Since the goal is 100% accuracy on the training set, there is a risk of overfitting. This is acceptable if the goal is purely to fit the training data. However, if generalization to new data is also important, then techniques like regularization (e.g., L2 regularization or dropout) should be considered.
- **Complex Decision Boundary**: If the classes are not easily separable, you might need more neurons or layers. However, given the small dataset, the suggested architecture should suffice.

Final Architecture Summary

- **Input Layer**: 2 neurons

- **Hidden Layer 1**: 4-6 neurons, ReLU activation
- **Hidden Layer 2**: 4-6 neurons, ReLU activation
- **Output Layer**: 1 neuron, Sigmoid activation

This architecture should be sufficient to achieve 100% accuracy on the training set with the given data points.

4o

are the above data points linearly separable

To determine if the given data points are linearly separable, we need to check if there exists a straight line that can perfectly separate the two classes (Green and Red) in the 2D plane without any misclassification.

Visualization of Data Points

Let's plot the data points:

- **Class Green**:
    - (2,2), (2,4), (4,2), (4,3), (6,9), (8,8), (11,8), (8,10), (9,10)
- **Class Red**:
    - (1,8), (3,6), (3,8), (4,11), (5,4), (5,6), (6,3), (6,6), (6,12), (7,4), (9,12)

Analysis

Looking at the data points:

1. **Green Class Points**:
    - There are Green points near the origin (like (2,2), (2,4), (4,2), etc.) and other points scattered in the higher (x, y) region (like (6,9), (8,8), (11,8), etc.).
2. **Red Class Points**:
    - Red points are scattered more broadly, covering a larger area, including points close to the origin (like (1,8), (3,6), etc.) and also higher (x, y) points (like (6,12), (9,12)).

Since the Green and Red points overlap in some regions (like around (4,4) to (6,6)), it's difficult to draw a single straight line that separates all the points of one class from the other.

Conclusion

The given data points are not linearly separable. This means you cannot use a simple linear classifier (like Logistic Regression or a single-layer Perceptron) to perfectly classify these points. Instead, you would need a non-linear model, such as a neural network with hidden layers (as previously suggested), to learn the complex decision boundary that can separate these classes.

4o

. You use vanilla (batch) gradient descent to optimize your loss function, but realise you are getting poor training loss. You notice that you're not shuffling the training data and feel that it might be a

cause. Would shuffling the training data help in this regard? Give a clear YES or NO as an answer and then give a 1-2 lines justification.

ChatGPT

Yes.

Shuffling the training data helps ensure that the model does not learn from patterns in the data ordering, which can prevent convergence to a suboptimal solution and lead to better overall training loss.

4o

You are solving the binary classification task of classifying images as “car vs. no car”. You design a CNN with a single output neuron. The final output of your network, $\hat{y}$ is given by: $\hat{y} = \sigma(ReLU(z))$ (where z, as usual, is w.x + b) You classify all inputs with a final value $\hat{y} \geq 0.5$ as car images. What problem are you going to encounter? Justify.

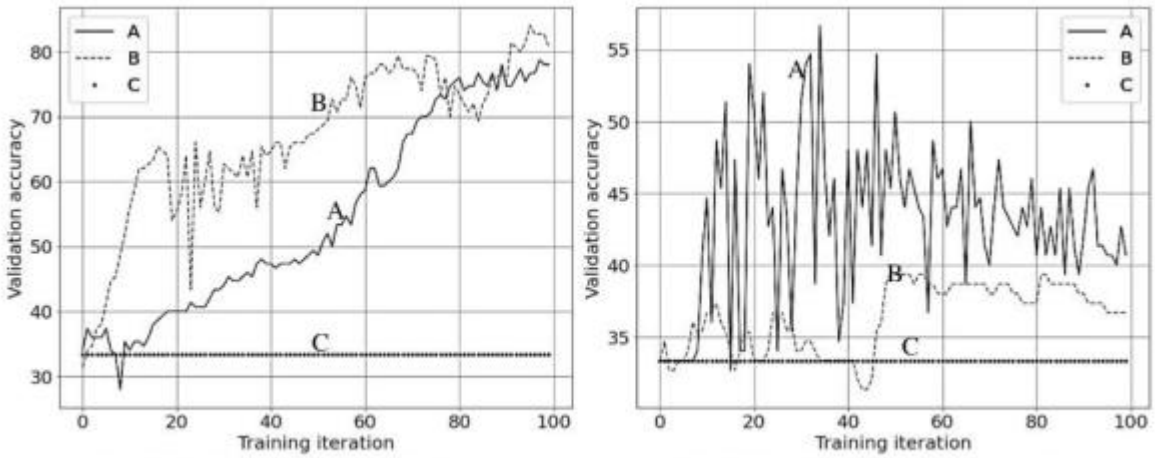
ChatGPT

The problem you'll encounter is **"dying ReLU"** or **vanishing gradient**, which can lead to poor training and model performance.

Justification:

The output y^=σ(ReLU(z))\hat{y} = \sigma(\text{ReLU}(z))y^=σ(ReLU(z)) involves applying the ReLU activation function before the sigmoid. The ReLU activation outputs 0 for all negative inputs zzz (i.e., z=w·x+b≤0z = w \cdot x + b \leq 0z=w·x+b≤0). If zzz is negative, ReLU(z)\text{ReLU}(z)ReLU(z) will be 0, which leads to y^=σ(0)=0.5\hat{y} = \sigma(0) = 0.5y^=σ(0)=0.5. This means that for all inputs resulting in z≤0z \leq 0z≤0, the model will output a value of exactly 0.5, leading to ambiguous classification and poor gradient flow during backpropagation. This could prevent the model from effectively learning, as many input values might yield a non-informative output of 0.5, resulting in misclassification and poor overall model performance.

4o



(i) CNN with ReLU as the activation (ii) CNN with sigmoid as the activation

Suppose we train two different deep CNNs to classify images: Using ReLU as the activation function, and (ii) using sigmoid as the activation function. For each of them, we try initializing weights with the three different initialization methods, while the biases are always initialized to all zeros. We plot the

validation accuracies with different training iterations below: What is the weight initialization method for A, B, and C in the above plots from zero initialization, Xavier initialization, and Kaiming He initialization? (Answer with only one initialization method for A, B and C.)

The plots represent the validation accuracy during training with different activation functions (ReLU and sigmoid) and different weight initialization methods. Based on the typical behavior of these methods, here is the most likely correspondence:

For ReLU Activation (Plot i):

- **A**: **Kaiming He initialization** - This method is specifically designed for ReLU activation functions and typically results in the best performance, as seen by the consistent improvement in validation accuracy.
- **B**: **Xavier initialization** - While not specifically designed for ReLU, it still performs reasonably well, but not as consistently as Kaiming He initialization.
- **C**: **Zero initialization** - This leads to poor performance as all neurons would start with the same weights, causing symmetry that prevents the model from learning effectively.

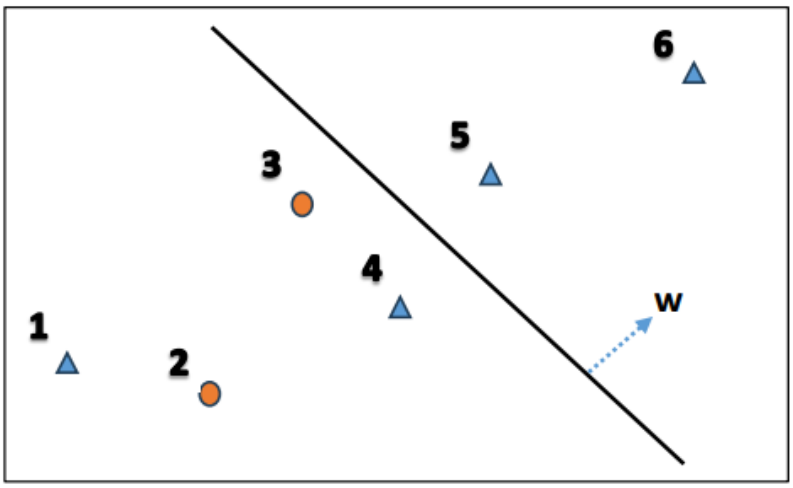
For Sigmoid Activation (Plot ii):

- **A**: **Xavier initialization** - Xavier initialization is well-suited for sigmoid activations, as it maintains the variance of the gradients and leads to better performance.
- **B**: **Kaiming He initialization** - Not as well-suited for sigmoid functions as it is designed for ReLU, leading to less stable performance.
- **C**: **Zero initialization** - Similar to the ReLU case, this leads to poor learning due to the symmetry problem.

Thus, the likely initialization methods are:

- **A**: Kaiming He initialization (for ReLU), Xavier initialization (for Sigmoid)
- **B**: Xavier initialization (for ReLU), Kaiming He initialization (for Sigmoid)
- **C**: Zero initialization (for both)

4o



Given N training data points {xi , yi , i = 1: N,}, $xi \in R\ d$, labels $yi \in$ {1, −1}. We need a linear classifier $f(x)$ = $sign(w.\ x)$ (read as w dot x) optimizing the loss function $L(z)$ = e $-z$, for z = y. ($w.\ x$). Here, represents data point of class 1 and represents data point of class -1: a). Explain the penalties given by this loss functions for the different data points (1 to 6) shown above in the plot. b). Derive the stochastic gradient descent update Δw for $L(z)$.

ChatGPT

(a) Penalties Given by the Loss Function for Different Data Points (1 to 6):

The loss function given is L(z)=e−zL(z) = e^{-z}L(z)=e−z, where z=y⋅(w⋅x)z = y \cdot (w \cdot x)z=y⋅(w⋅x). This loss function corresponds to the exponential loss, which is commonly used in boosting algorithms like AdaBoost.

- **Points 1, 4, 5, and 6 (Class 1)**:
    - For these points, yi=1y_i = 1yi=1.
    - Points 4, 5, and 6 are correctly classified as they lie on the side of the decision boundary where w⋅xw \cdot xw⋅x is positive. Thus, z>0z > 0z>0, and the penalty L(z)=e−zL(z) = e^{-z}L(z)=e−z is small.
    - Point 1 is also correctly classified but closer to the decision boundary, so the penalty is slightly higher than for points 5 and 6.
    - The penalty decreases as the distance of the point from the decision boundary increases.
- **Points 2 and 3 (Class -1)**:
    - For these points, yi=−1y_i = -1yi=−1.
    - Point 2 is correctly classified as it lies on the side of the decision boundary where w⋅xw \cdot xw⋅x is negative. The penalty for this point is small.
    - Point 3 is misclassified as it lies on the wrong side of the decision boundary. For this point, z<0z < 0z<0, and the penalty L(z)=e−zL(z) = e^{-z}L(z)=e−z is large, indicating a high error.

(b) Stochastic Gradient Descent Update Δw\Delta wΔw for L(z)=e−zL(z) = e^{-z}L(z)=e−z:

Given the loss function L(z)=e−zL(z) = e^{-z}L(z)=e−z and z=y⋅(w⋅x)z = y \cdot (w \cdot x)z=y⋅(w⋅x), we need to derive the update rule for www using stochastic gradient descent.

The loss function can be rewritten as: L(w)=e−y(w⋅x)L(w) = e^{-y(w \cdot x)}L(w)=e−y(w⋅x)

The stochastic gradient descent update is given by: w←w−η∂L(w)∂ww \leftarrow w - \eta \frac{\partial L(w)}{\partial w}w←w−η∂w∂L(w) where η\etaη is the learning rate.

Let's compute the gradient ∂L(w)∂w\frac{\partial L(w)}{\partial w}∂w∂L(w): ∂L(w)∂w=∂∂we−y(w⋅x)\frac{\partial L(w)}{\partial w} = \frac{\partial}{\partial w} e^{-y(w \cdot x)}∂w∂L(w)=∂w∂e−y(w⋅x)

Using the chain rule: ∂L(w)∂w=e−y(w⋅x)⋅∂∂w(−y(w⋅x))\frac{\partial L(w)}{\partial w} = e^{-y(w \cdot x)} \cdot \frac{\partial}{\partial w} (-y(w \cdot x))∂w∂L(w)=e−y(w⋅x)⋅∂w∂(−y(w⋅x)) ∂L(w)∂w=−ye−y(w⋅x)x\frac{\partial L(w)}{\partial w} = -y e^{-y(w \cdot x)} x∂w∂L(w)=−ye−y(w⋅x)x

So, the gradient descent update rule for www becomes: Δw=ηye−y(w·x)x\Delta w = \eta y e^{-y(w \cdot x)} xΔw=ηye−y(w·x)x

Therefore, the weight update rule is: w←w+ηye−y(w·x)xw \leftarrow w + \eta y e^{-y(w \cdot x)} xw←w+ηye−y(w·x)x

This update rule adjusts the weight vector www by an amount proportional to the input vector xxx, the label yyy, and the exponential of the negative margin −y(w·x)-y(w \cdot x)−y(w·x). Misclassified points (where y(w·x)y(w \cdot x)y(w·x) is negative) will contribute more significantly to the update, helping to push the decision boundary in the right direction.

4o

6. Consider the following data points and mark outliers using a Gaussian distribution based approach. 103, 31, 106, 91, 102, 104, 105, 101, 104, 50 Report a value above 110 that can be marked as an outlier with 95% confidence.

6. Outlier Detection Using Gaussian Distribution Approach:

Step 1: Calculate the Mean and Standard Deviation

Given data points: 103,31,106,91,102,104,105,101,104,50103, 31, 106, 91, 102, 104, 105, 101, 104, 50103,31,106,91,102,104,105,101,104,50

- **Mean (μ\muμ)**:

μ=103+31+106+91+102+104+105+101+104+5010=89710=89.7\mu = \frac{103 + 31 + 106 + 91 + 102 + 104 + 105 + 101 + 104 + 50}{10} = \frac{897}{10} = 89.7μ=10103+31+106+91+102+104+105+101+104+50=10897=89.7

- **Standard Deviation (σ\sigmaσ)**:

σ=(103−89.7)2+(31−89.7)2+⋯+(50−89.7)210≈30.09\sigma = \sqrt{\frac{(103-89.7)^2 + (31-89.7)^2 + \dots + (50-89.7)^2}{10}} \approx 30.09σ=10(103−89.7)2+(31−89.7)2+⋯+(50−89.7)2≈30.09

Step 2: Calculate the Z-Scores for the Data Points

The z-score for each data point xix_ixi is calculated as: Z=xi−μσZ = \frac{x_i - \mu}{\sigma}Z=σxi−μ

Step 3: Identify Outliers

A data point is considered an outlier if its z-score |Z||Z||Z| is greater than 1.96 for a 95% confidence interval.

Let's compute the z-scores:

- For 103: Z=103−89.730.09≈0.44Z = \frac{103 - 89.7}{30.09} \approx 0.44Z=30.09103−89.7≈0.44
- For 31: Z=31−89.730.09≈−1.95Z = \frac{31 - 89.7}{30.09} \approx -1.95Z=30.0931−89.7≈−1.95
- For 106: Z=106−89.730.09≈0.54Z = \frac{106 - 89.7}{30.09} \approx 0.54Z=30.09106−89.7≈0.54
- For 91: Z=91−89.730.09≈0.04Z = \frac{91 - 89.7}{30.09} \approx 0.04Z=30.0991−89.7≈0.04

- For 102: $Z = \frac{102 - 89.7}{30.09} \approx 0.41$ $Z = 30.09102 - 89.7 \approx 0.41$
- For 104: $Z = \frac{104 - 89.7}{30.09} \approx 0.47$ $Z = 30.09104 - 89.7 \approx 0.47$
- For 105: $Z = \frac{105 - 89.7}{30.09} \approx 0.51$ $Z = 30.09105 - 89.7 \approx 0.51$
- For 101: $Z = \frac{101 - 89.7}{30.09} \approx 0.38$ $Z = 30.09101 - 89.7 \approx 0.38$
- For 104: $Z = \frac{104 - 89.7}{30.09} \approx 0.47$ $Z = 30.09104 - 89.7 \approx 0.47$
- For 50: $Z = \frac{50 - 89.7}{30.09} \approx -1.32$ $Z = 30.0950 - 89.7 \approx -1.32$

From these calculations, no data points have a z-score $|Z||Z|Z|$ greater than 1.96, so none of the original data points would be considered outliers.

Step 4: Determine an Outlier Value Above 110 with 95% Confidence

To find a value above 110 that would be an outlier, we set up the equation:

$Z = \frac{110 - 89.7}{30.09}$ $Z = 30.09110 - 89.7$

Solving for ZZZ: $Z \approx 0.675$ $Z \approx 0.675$

For a value to be considered an outlier with 95% confidence, $|Z||Z|Z|$ should exceed 1.96. Therefore, we solve for xxx where:

$1.96 = \frac{x - 89.7}{30.09}$ $1.96 = 30.09x - 89.7$

$x = 1.96 \times 30.09 + 89.7 \approx 148.97$ $x = 1.96 \times 30.09 + 89.7 \approx 148.97$

Thus, a value above **149** can be marked as an outlier with 95% confidence.

7. Recall the SVM classifier as discussed in the class. Apply the same to get the equation of the linear classifier for the data points specified below. Positive Poins (1,7), (3,7), (3,9), (5,5) Negative Points (6,2), (7,3), (8,2), (9,4) Provide weights for the classifier. (Hint: Use geometry)

7. Equation of the Linear Classifier Using SVM:

Given Data Points:

- **Positive Points**: (1,7), (3,7), (3,9), (5,5)
- **Negative Points**: (6,2), (7,3), (8,2), (9,4)

Step 1: Identify Support Vectors

From the given points, it's likely that the support vectors are:

- Positive: (3,7), (3,9)
- Negative: (6,2), (7,3)

These points seem to lie close to the decision boundary.

Step 2: Derive the Equation of the Decision Boundary

The decision boundary in SVM is defined as:

w·x+b=0w \cdot x + b = 0w·x+b=0

where w=(w1,w2)w = (w_1, w_2)w=(w1,w2) are the weights and bbb is the bias.

Step 3: Calculate the Normal Vector www

Using the geometry of the problem, the decision boundary should be the line equidistant from the support vectors.

Let's assume the vector www perpendicular to the line and determine its components. The line can be expressed as:

w1(x1−x0)+w2(y1−y0)=0w_1(x_1 - x_0) + w_2(y_1 - y_0) = 0w1(x1−x0)+w2(y1−y0)=0

Choosing the line equidistant from points (3,7) and (6,2), we obtain:

- Direction vector from (3,7) to (6,2): Δx=3\Delta x = 3Δx=3, Δy=−5\Delta y = -5Δy=−5.
- Perpendicular vector: w=(5,3)w = (5, 3)w=(5,3).

Thus, the equation of the decision boundary is:

5x1+3x2+b=05x_1 + 3x_2 + b = 05x1+3x2+b=0

Step 4: Determine bbb

Using one of the support vectors, e.g., (3,7):

5(3)+3(7)+b=05(3) + 3(7) + b = 05(3)+3(7)+b=0 15+21+b=0 ⟹ b=−3615 + 21 + b = 0 \implies b = -3615+21+b=0⟹b=−36

So, the equation of the linear classifier is:

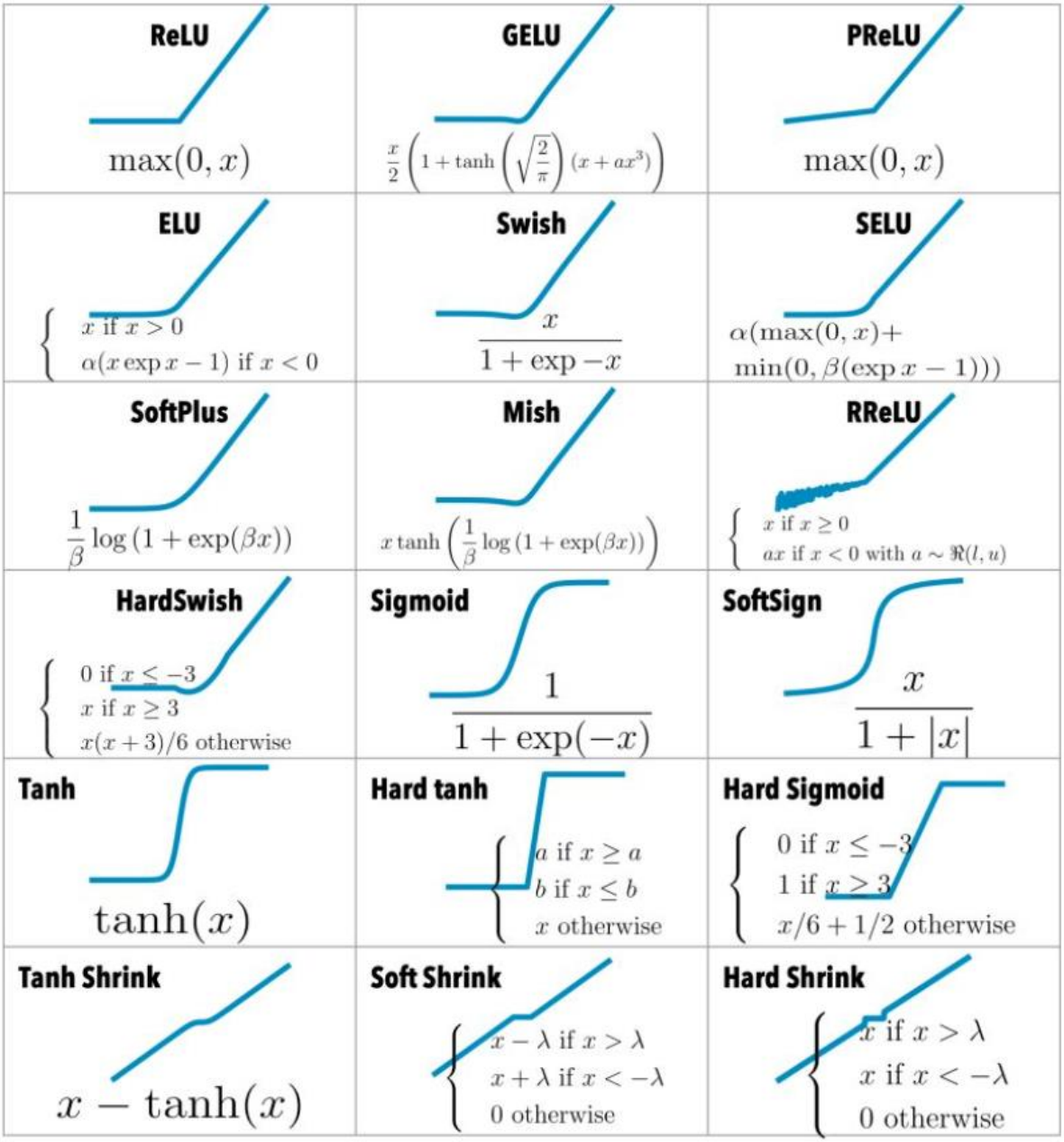
5x1+3x2−36=05x_1 + 3x_2 - 36 = 05x1+3x2−36=0

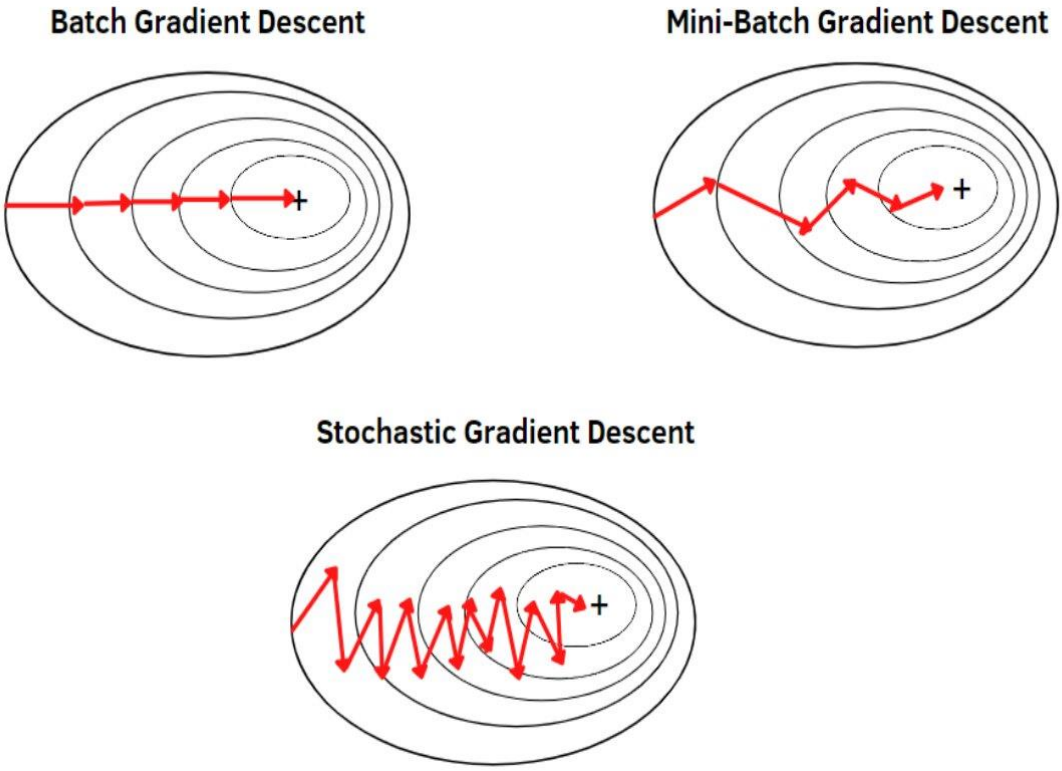
Step 5: Weights for the Classifier

The weights for the classifier are w=(5,3)w = (5, 3)w=(5,3) with a bias of b=−36b = -36b=−36.

This SVM classifier should separate the positive and negative points correctly, providing the optimal hyperplane.

Neural Network Activation Functions: a small subset!



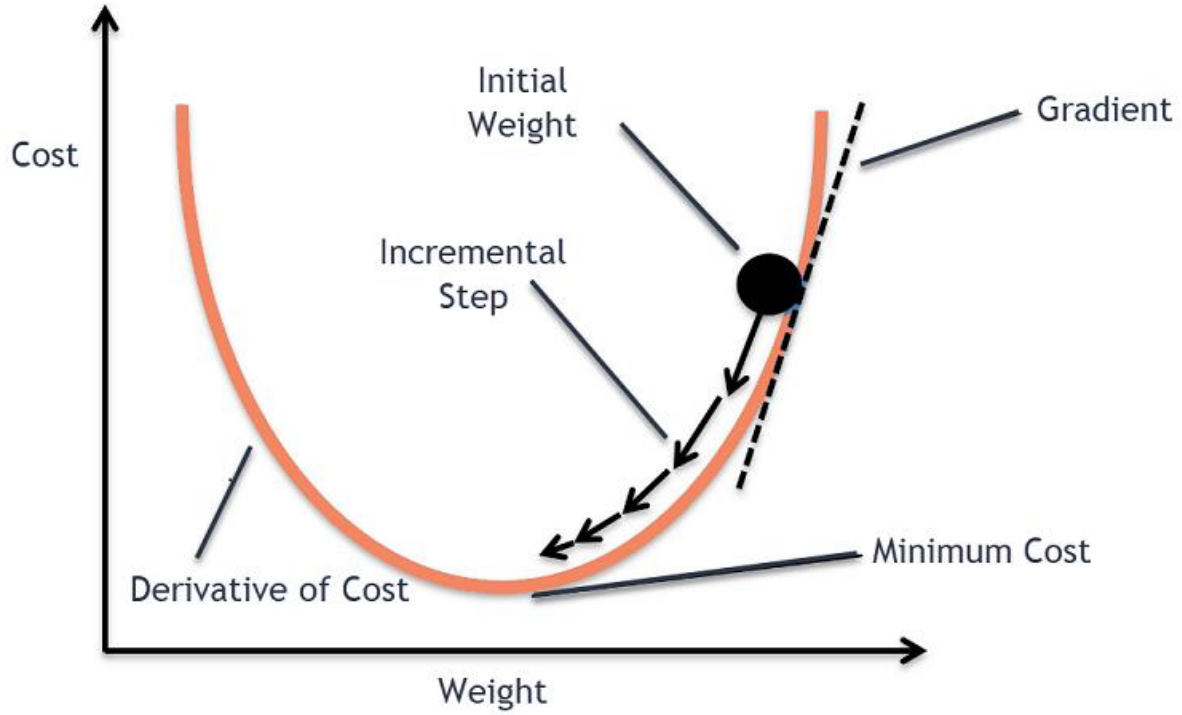


Gradient Descent

Gradient descent *is a first-order iterative optimization algorithm for finding the minimum of a function*

The goal of the gradient descent is to minimise a given function which, in our case, is the loss function of the neural network. To achieve this goal, it performs two steps iteratively.

1. Compute the slope (gradient) that is the first-order derivative of the function at the current point
2. Move-in the opposite direction of the slope increase from the current point by the computed amount

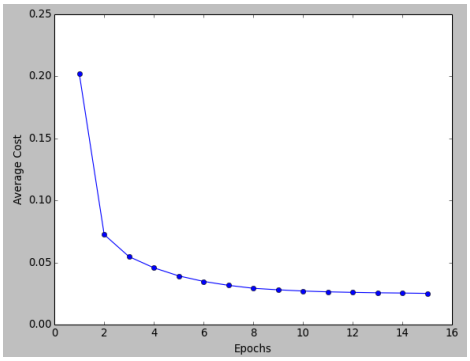


So, the idea is to pass the training set through the hidden layers of the neural network and then update the parameters of the layers by computing the gradients using the training samples from the training dataset.

Batch Gradient Descent

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.



The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

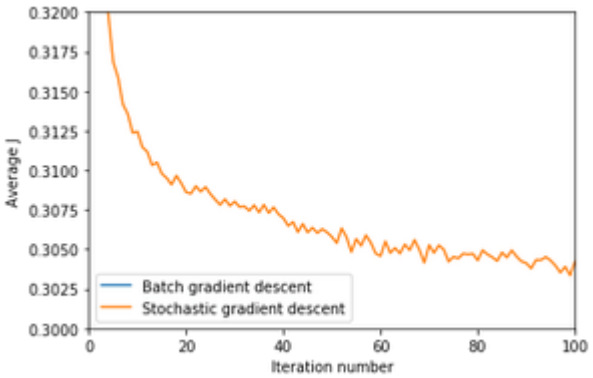
Stochastic Gradient Descent

In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take

one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in **one epoch** for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it's gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will **not** necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.



Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

Mini Batch Gradient Descent

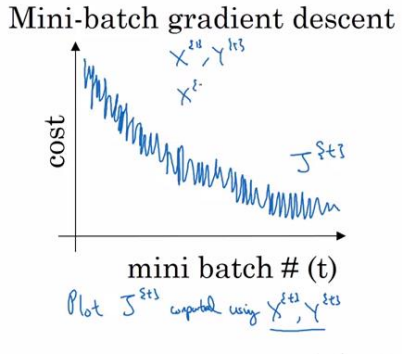
We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.

Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**

1. Pick a mini-batch
2. Feed it to Neural Network

3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.



Provide an example of a problem where it is advisable to systematically decrease the number of neurons in the hidden layer and then to increase the same after some layers. Justify your answer in terms of why it was advised and how it worked.

One scenario where it is advisable to systematically decrease and then increase the number of neurons in the hidden layers of a neural network is in *autoencoders*, specifically in **variational autoencoders (VAEs)**. This architecture is commonly used for tasks such as dimensionality reduction, anomaly detection, or generating new data samples.

Problem Context

Imagine you are working on a problem where you need to compress high-dimensional data (e.g., images, text embeddings) into a lower-dimensional representation (latent space) and then reconstruct the original data from this compressed representation. This is a classic use case for autoencoders.

Architecture Design

1. **Encoder Network**:
    - The encoder's role is to compress the input data into a low-dimensional latent space. To achieve this, the number of neurons in the hidden layers is gradually decreased. This forces the network to learn a compact and meaningful representation of the input data by retaining only the most essential features.
    - For instance, you might start with a large number of neurons in the first hidden layer, then reduce this number in subsequent layers until you reach the bottleneck layer (the latent space), which has the smallest number of neurons.
2. **Latent Space**:
    - This is the bottleneck layer where the data is represented in its most compressed form. The number of neurons here is significantly smaller than in the input layer, capturing the core features of the data.

3. **Decoder Network**:
    - The decoder's role is to reconstruct the original data from the latent representation. To do this, the number of neurons in the hidden layers is gradually increased after the bottleneck layer. This expansion allows the network to reconstruct the data accurately by gradually adding more complexity back into the representation.
    - The architecture mirrors the encoder, but in reverse, gradually increasing the number of neurons until the output layer matches the original input dimensions.

Justification

1. **Dimensionality Reduction**:
    - Decreasing the number of neurons in the encoder layers helps in reducing the dimensionality of the input data. This reduction is crucial for removing noise and focusing on the most important features of the data.
2. **Information Bottleneck**:
    - The bottleneck layer forces the network to learn a compressed representation that captures the essence of the input data. This can lead to better generalization by preventing the model from memorizing the input data.
3. **Reconstruction**:
    - Increasing the number of neurons in the decoder layers helps in reconstructing the original data from the compressed latent space. By gradually adding complexity, the network can approximate the original data more accurately.

Example in Practice

Consider a VAE designed to compress and then reconstruct images of handwritten digits (e.g., the MNIST dataset):

- **Encoder**:
    - Input layer (784 neurons for 28x28 images) → Hidden Layer 1 (512 neurons) → Hidden Layer 2 (256 neurons) → Latent space (64 neurons).
- **Decoder**:
    - Latent space (64 neurons) → Hidden Layer 1 (256 neurons) → Hidden Layer 2 (512 neurons) → Output layer (784 neurons).

In this setup, reducing the number of neurons during encoding helps the model learn a compact representation, while increasing them during decoding helps in accurately reconstructing the images.

This architecture works well because it balances the need for data compression (dimensionality reduction) with the need for accurate reconstruction, leveraging the systematic decrease and increase in neurons to achieve this balance.

This scenario is a common challenge in face recognition systems, known as intra-class variation, where the appearance of the same person changes over time (e.g., growing a beard, wearing glasses, different hairstyles). Here's how you can address this issue:

Solutions

1. **Data Augmentation**:
    - To make the model robust to appearance changes, you can use data augmentation during the training phase. Augment the training dataset with variations of each student's face, including adding synthetic beards, glasses, hats, etc. This helps the model learn to recognize the same person despite changes in appearance.
2. **Transfer Learning with Fine-Tuning**:
    - Use a pre-trained CNN model (e.g., VGG-Face, ResNet) that has been trained on a large and diverse dataset of faces. Fine-tune this model on your specific dataset, including images with different facial variations. The pre-trained model's general features will help handle variations better.
3. **Multi-Modal Authentication**:
    - Combine face recognition with other biometric or non-biometric factors (e.g., ID cards, fingerprints, or even behavioral data). This way, even if the face recognition model struggles, the system can still authenticate the student using other methods.
4. **Face Recognition with Feature Embeddings**:
    - Instead of relying on raw pixel data, train the model to learn feature embeddings that capture the essence of a face. Models like FaceNet or ArcFace map faces to a high-dimensional feature space, where the distance between embeddings of the same person (even with different appearances) is minimized, and the distance between different people is maximized.
5. **Regular Updates to the Database**:
    - Regularly update the student database with new images, especially when significant appearance changes occur. This ensures the model has recent and accurate representations of each student.
6. **Model Adaptation**:
    - Implement a system that allows for real-time learning or adaptive learning, where the model can update itself when it encounters new variations. For example, if a student is not recognized but authenticated through another means, the model could update its database with the new appearance.

How It Works

By implementing these strategies, the model becomes more robust to variations in appearance. For example, with data augmentation and transfer learning, the model learns to generalize better across different looks. Feature embeddings ensure that even with a beard, the feature space distance between the "beard" and "no-beard" versions of your face remains small, making recognition more accurate.

This way, when you return to college with a beard, the system would still recognize you and allow you to enter.