DATA MINING AND MACHINE LEARNING

ASSIGNMENT 4

1.[5 pts] (Refer to p. 54 in deep learning slide) Suppose we use MSE(mean squared error) error function in this RNN. Compute the error of first step (input='h')

The error for the first step (input='h') can be computed as follows:

The image shows a character-level language model example with an input sequence of "hello". For the first step, the input character is 'h'.

The error values for each target character are given in the "error" rows of the diagram:

For the target character 'h', the error is 1.0

For the target character 'e', the error is 0.5

For the target character 'l', the error is 0.1

For the target character 'l', the error is 0.2

For the target character 'o', the error is -0.3

To compute the total error for the first step (input='h'), we need to take the mean squared error (MSE) of these individual errors.

The MSE is calculated as the average of the squared errors:

MSE = (1.0^2 + 0.5^2 + 0.1^2 + 0.2^2 + (-0.3)^2) / 5

= (1.0 + 0.25 + 0.01 + 0.04 + 0.09) / 5

= 1.39 / 5

= 0.278

2. [3 pts] In p. 55 in deep learning slide, suppose the output of RNN is not forwarded as an input to the next step. In this case, does the RNN always generates the same output in each step? Explain in detail "USING" the formula of RNN node.

Therefore, the error for the first step (input='h') using the MSE error function is 0.278

Based on the data in the image, if the output of the RNN is not forwarded as an input to the next step, the RNN will not always generate the same output in each step.

The key points are:

1. The image shows a character-level language model example, where the input sequence is "hello".
2. The RNN architecture consists of an input layer, a hidden layer, and an output layer (softmax).

The formula for the RNN node is:

$h_t = f(W_{xh} * x_t + W_{hh} * h_{t-1} + b_h)$

$y_t = g(W_{hy} * h_t + b_y)$

Where:

- h_t is the hidden state at time step t
- x_t is the input at time step t
- W_{xh} is the weight matrix between input and hidden layer
- W_{hh} is the weight matrix between hidden layers (across time steps)
- b_h is the bias of the hidden layer
- y_t is the output at time step t
- W_{hy} is the weight matrix between hidden and output layer
- b_y is the bias of the output layer
- f and g are activation functions (e.g., tanh, softmax)

3. If the output y_t is not fed back as input x_{t+1} for the next time step, the hidden state h_t will still depend on the previous hidden state h_{t-1} and the current input x_t.
4. This means that the hidden state h_t, and consequently the output y_t, will change at each time step, even if the input x_t is the same (e.g., the first character 'h' in the sequence "hello").

Therefore, without the output feedback, the RNN will not generate the same output in each step, as the hidden state will evolve differently based on the current input and the previous hidden state.

The key difference is that without the output feedback, the RNN cannot maintain a consistent internal state and will generate different outputs at each time step, even for the same input character

3.[3 pts] In semi-supervised learning, we use both labeled and unlabeled data. During the training process we repeatedly predict (and also update) the label of unlabeled data and use them in supervised learning. In this case using cross entropy error function and using KL divergence have the same result? Explain it in detail "USING" the formula of cross-entropy and KL divergence.

In semi-supervised learning, both labeled and unlabeled data are used to improve the model's performance. When we predict the labels of unlabeled data, we can use various loss functions to measure how well our predictions align with the true distributions of the data. Two common measures are the cross-entropy loss and Kullback-Leibler (KL) divergence. So they relate to each other as follows:

Cross-Entropy Loss

The cross-entropy loss is a measure of the difference between two probability distributions: the true distribution P and the predicted distribution Q. The formula for cross-entropy H(P,Q) is given by:

$$\mathbf{H(P,Q) = -\sum_x P(x) \log(Q(x))}$$

Where:

- P(x) is the true distribution (often represented as one-hot encoded labels for labeled data).
- Q(x) is the predicted distribution (the output of the model).

KL Divergence

The KL divergence measures how one probability distribution diverges from a second, expected probability distribution. The formula for KL divergence $D_{KL}(P||Q)$ is:

$$\mathbf{D_{KL}(P||Q) = \sum_x P(x) \log(P(x)/Q(x))}$$

Relationship Between Cross-Entropy and KL Divergence

To see how cross-entropy and KL divergence relate, we can rewrite the KL divergence formula:

$$D_{KL}(P||Q)=\sum_x P(x)\log(P(x))-\sum_x P(x)\log(Q(x))$$

The first term $\sum_x P(x)\log(P(x))$ is a constant with respect to Q (it only depends on the true distribution P). Therefore, we can express the KL divergence in terms of cross-entropy:

$$D_{KL}(P||Q)=\text{Constant}-H(P,Q)$$

This shows that minimizing the KL divergence $D_{KL}(P||Q)$ is equivalent to maximizing the cross-entropy H(P,Q) (up to a constant).

In Semi-Supervised Learning:

In semi-supervised learning, when we use both labeled and unlabeled data, we often predict labels for the unlabeled data and use these predictions in our training. If we use cross-entropy loss for the labeled data and incorporate predictions for the unlabeled data, we are effectively minimizing the KL divergence between the true distribution of the labeled data and our model's predicted distribution.

1. **For Labeled Data**: We compute the cross-entropy loss directly using the true labels.
2. **For Unlabeled Data**: We treat the predicted labels as a distribution Q and compute the cross-entropy with respect to the true distribution P.

Conclusion

So, while cross-entropy and KL divergence measure different aspects of probability distributions, they are closely related in the context of semi-supervised learning. When using cross-entropy to update the model based on both labeled and unlabeled data, we are effectively minimizing the KL divergence, leading to similar outcomes in terms of model performance. Therefore, using either can yield comparable results in this framework, depending on how the predictions are formulated and updated during training.

4. [3 pts/ea] In sigmoid function, explain the following "USING" the graphs of activation functions.

1) When the initial weight values are very high, sigmoid may cause vanishing gradient. Explain it in detail

To know how high initial weight values in a sigmoid activation function can lead to the vanishing gradient problem, we are supposed to look at the properties of the sigmoid function and how it behaves with respect to its inputs.

Sigmoid Function

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The graph of the sigmoid function is an S-shaped curve that maps any real-valued number into the range (0, 1).

Graph of the Sigmoid Function can be represented as

- The function approaches 0 as x approaches negative infinity.
- The function approaches 1 as x approaches positive infinity.
- The steepest part of the curve is around x=0, where the function has its maximum slope.

Vanishing Gradient Problem

High Initial Weights

1. Effect of High Weights: When the initial weights of the neurons are very high, the inputs to the sigmoid function become large (either positive or negative). For example, if the weighted sum $z=w \cdot x$ is large, then $\sigma(z)$ will be very close to either 0 or 1.
2. Saturation of the Sigmoid:

   If z is very large (positive), $\sigma(z)$ approaches 1.

   If z is very small (negative), $\sigma(z)$ approaches 0.

   In both cases, the derivative of the sigmoid function, which is given by:

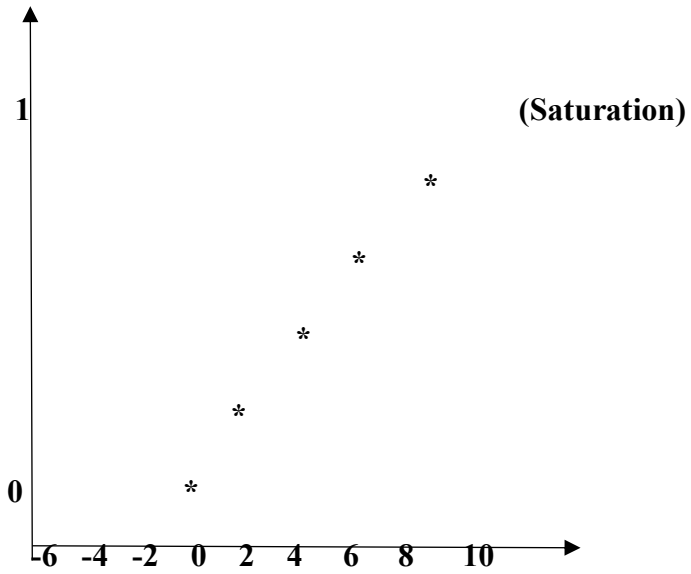
   $\sigma'(x)=\sigma(x)(1-\sigma(x))$ becomes very small because:

   $\sigma(0)=0.5$ gives the maximum slope of 0.25.

   For z near 0, the gradient is acceptable, but for large z (positive or negative), the gradient approaches 0.
3. Gradient Propagation: During backpropagation, the gradients are multiplied through each layer. If the gradients are very small (due to the saturation of the sigmoid), the product of many small gradients can lead to very tiny updates to the weights. This phenomenon is known as the vanishing gradient problem.

Graphical Illustration

The conceptual representation of how high weights affect the sigmoid function and lead to the vanishing gradient problem



In the graph above, the steepest part of the sigmoid curve is around 0. As the input moves away from 0 (either direction), the curve flattens out, leading to a very small gradient.

Conclusion

Therefore, when initial weight values are very high, the inputs to the sigmoid function can cause the outputs to saturate at either 0 or 1. This saturation results in very small gradients during the backpropagation step, leading to minimal updates to the weights. Consequently, this can significantly slow down or even halt the learning process, which is the essence of the vanishing gradient problem.

Using alternative activation functions like ReLU can help mitigate this issue, as they do not suffer from saturation in the same way as sigmoid.

2) When all weight values are normalized around N(0,1), entire network may become a linear model. Explain in detail.

When all weight values in a neural network are initialized around a normal distribution N(0,1) it can lead to a situation where the entire network behaves like a linear model. The reasons behind this phenomenon is

Neural Network Basics

A neural network consists of layers of neurons, where each neuron applies a linear transformation followed by a non-linear activation function. The general operation of a neuron can be expressed as:

$y=f(Wx+b)$

Where:

- W is the weight matrix.
- x is the input vector.
- b is the bias term.
- f is the activation function (e.g., sigmoid, ReLU).

Linear Transformation

1. **Linear Combination**: The term Wx+b represents a linear combination of the inputs. If we have multiple layers, the output of one layer becomes the input to the next layer.
2. **Effect of Activation Functions**: The purpose of activation functions is to introduce non-linearity into the model. This non-linearity allows the network to learn complex patterns in the data. Common activation functions include:

   Sigmoid

   Tanh

   ReLU (Rectified Linear Unit)

Normalization of Weights

When weights are initialized to be drawn from a normal distribution N(0,1):

1. **Mean and Variance**: The weights will have a mean of 0 and a variance of 1. This means that for a large number of weights, they will be symmetrically distributed around 0.
2. **Layer Outputs**: If the weights are close to zero, the output of the linear transformation Wx+b will also be close to zero for many inputs, especially if the inputs are also normalized. This leads to the activation function outputs being very close to their linear regions.

Linear Model Behavior

1. **Activation Functions in Linear Regions**:

   For activation functions like sigmoid or tanh, when the input to the function is near zero (due to small weights), the output will be near the linear part of the function.

   For example, for sigmoid: $\sigma(x) \approx 0.5$ when $x \approx 0$

   For tanh: $\tanh(x) \approx 0$ when $x \approx 0$

2. **Composition of Linear Functions**: When you stack multiple layers that are all producing outputs near their linear regions, the overall function can be approximated as a linear combination of the inputs. Mathematically, if each layer behaves linearly, then:

   $\mathbf{y = f(W_n f(W_n - 1 f(...f(W1_x + b1) + b2...) + b_n)}$ can be simplified to a linear function of x.

3. **Loss of Non-linearity**: As a result, regardless of how many layers are in the network, if all weights are initialized to be small (as with N(0,1)), the entire network may behave like a single linear transformation. This means the network cannot learn complex patterns or perform tasks that require non-linear decision boundaries.

Conclusion

Therefore, initializing all weight values around a normal distribution N(0,1) can lead to a situation where the outputs of the neurons are close to their linear regions. This results in the network effectively becoming a linear model, as the non-linearities introduced by activation functions become negligible. For neural networks to capture complex relationships in data, appropriate weight initialization strategies (like Xavier or He initialization) and careful scaling of inputs are crucial to maintain the non-linear dynamics necessary for learning

3) In gradient descent, using tanh usually makes it converge faster than using sigmoid. Explain the reason.

In gradient descent optimization, using the hyperbolic tangent function (tanh) often leads to faster convergence compared to using the sigmoid function. This difference in convergence speed can be attributed to several key factors related to the properties of these activation functions.

1. Output Range

- **Sigmoid Function**: The sigmoid function outputs values in the range (0,1). This means that all outputs are positive and centered around 0.5.

  $$\sigma(x) = \frac{1}{1+e^{-x}}$$

- **Tanh Function**: The tanh function outputs values in the range (−1,1). This allows it to be centered around 0, providing both negative and positive outputs.

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

2. Zero-Centered Output

- **Impact of Non-Zero Centering**: Since the sigmoid function outputs are not zero-centered, the gradients during backpropagation can consistently push weights in one direction (either positive or negative) depending on the input. This can lead to inefficient updates and slower convergence, especially when the inputs to the neurons are also centered around zero.
- **Zero-Centered Tanh**: In contrast, the tanh function is zero-centered. This means that the outputs can be both negative and positive, allowing the gradients to be distributed more symmetrically. This helps in balancing the weight updates, leading to more efficient convergence.

3. Gradient Behavior

- **Gradient Saturation**: Both sigmoid and tanh functions can suffer from the vanishing gradient problem, but the saturation effects are more pronounced in the sigmoid function. When the input to the sigmoid function is far from zero (either very positive or very negative), the gradient becomes very small:

  $$\sigma'(x)=\sigma(x)(1-\sigma(x))$$

Near the extremes (0 or 1), the derivative approaches 0.

- **Tanh Gradient**: The derivative of the tanh function is:

  $$\tanh'(x)=1-\tanh^2(x)$$

The gradient is larger than that of the sigmoid function for the same input values, especially around the origin. This means that during backpropagation, the updates to the weights are generally more substantial with tanh, allowing for faster learning.

4. Convergence Speed

- **Faster Learning**: Because tanh has a steeper gradient near the origin and is zero-centered, the weights can be updated more effectively. This generally leads to a more rapid reduction in loss during training, resulting in faster convergence.
- **Reduced Risk of Vanishing Gradients**: With tanh, while it still faces the vanishing gradient problem, the risk is reduced compared to sigmoid, particularly in deeper networks. This helps in maintaining more effective gradient flow through the network.

Conclusion

Therefore, using the tanh activation function in gradient descent typically leads to faster convergence compared to the sigmoid function due to its zero-centered output, steeper gradients near the origin, and reduced risk of saturation effects. These properties allow for more effective weight updates, enabling the model to learn more quickly and efficiently.

5. [3 pts/ea] Neural network with many hidden layers and each hidden node is using ReLU.

1) Explain why ReLU sometimes has the effect of Dropout.

ReLU (Rectified Linear Unit) is a popular activation function used in neural networks, particularly in deep learning. It is defined as:

$$ReLU(x)=max(0,x)$$

This means that ReLU outputs zero for any negative input and passes positive input unchanged. While ReLU is primarily used to introduce non-linearity into the model, it can exhibit behavior similar to dropout under certain conditions.

1. Sparsity in Activations

- Zero Outputs: Since ReLU outputs zero for any negative input, many neurons can become inactive (outputting zero) for a given input. This results in a sparse activation pattern, where only a subset of neurons are active at any given time.
- Effect Similar to Dropout: Dropout is a regularization technique where, during training, a random subset of neurons is "dropped out" (set to zero) to prevent overfitting. The sparsity created by ReLU can mimic this behavior, as a significant number of neurons may not contribute to the output for certain inputs.

2. Dynamic Neuron Activation

- Random Inactivation: The randomness in which neurons become inactive can vary from one input to another. Depending on the input data, different neurons may activate or deactivate, leading to a dynamic network structure. This variability can help the model generalize better, similar to how dropout prevents reliance on any specific set of neurons.

3. Mitigating Overfitting

- Regularization Effect: The inherent sparsity induced by ReLU can act as a form of implicit regularization. By having many neurons output zero, the model is less likely to overfit to the training data, as not all neurons are being utilized for every input. This can lead to improved generalization to unseen data.
- Encouraging Robust Features: With ReLU, the network may learn to rely on different subsets of features for different inputs, encouraging the model to learn more robust and diverse representations.

4. Comparison to Dropout

- Dropout Implementation: Dropout explicitly sets a random subset of neurons to zero during training, which forces the network to learn redundant representations. This is a deliberate and controlled method of regularization.
- ReLU Behavior: In contrast, the zeroing out of activations in ReLU is a natural consequence of its definition. While it can lead to similar effects as dropout, it is not controlled or random in the same way; rather, it depends on the input values.

Conclusion

Therefore, ReLU can sometimes have effects similar to dropout due to its ability to produce sparse activations by outputting zero for negative inputs. This leads to a dynamic selection of active neurons, which can help mitigate overfitting and encourage robust feature learning. While both techniques aim to improve generalization, they operate through different mechanisms—ReLU through its activation properties and dropout through explicit neuron inactivation.

2) Explain why output values may explode (grows exponentially)

Output values in a neural network may explode, growing exponentially, due to several factors related to the architecture, initialization, and training process. The key reasons are as follows:

1. Weight Initialization

- Large Initial Weights: If the weights of the network are initialized with large values, the outputs can quickly grow during forward propagation. In deep networks, this can lead to exponentially increasing activations as each layer multiplies the input by these large weights.

2. Activation Functions

- Unbounded Activations: Some activation functions, such as the exponential function, can produce outputs that grow very quickly. For instance, if a neuron uses an exponential activation function, even small positive inputs can lead to very large outputs.
- ReLU Activation: While ReLU itself does not output large values directly, if the inputs to ReLU are large (due to previous layers having large outputs), the outputs can also be large. This can happen if the weights are not properly controlled.

3. Deep Networks and Layer Stacking

- Accumulation of Values: In deep networks, each layer's output becomes the input for the next layer. If weights are not properly scaled, the repeated multiplication of large values through many layers can lead to exponential growth of outputs. This phenomenon is often referred to as the "exploding gradient problem."

4. Gradient Descent and Backpropagation

- Exploding Gradients: During backpropagation, if the gradients are large, weight updates can become excessively large, causing the weights to grow rapidly. This can lead to a situation where the outputs of the network increase exponentially with each training step.

5. Lack of Regularization

- Overfitting: Without proper regularization techniques (like dropout, L2 regularization, or batch normalization), the network may learn to produce extreme outputs as it tries to fit the training data too closely, leading to instability in the output values.

6. Learning Rate Issues

- High Learning Rate: If the learning rate is set too high, the weight updates during training can be excessively large, causing the weights to explode. This can lead to outputs that grow exponentially as the network tries to adjust to the loss function.

7. Loss Function Characteristics

- Loss Function Sensitivity: Some loss functions can be sensitive to large output values. For example, using mean squared error (MSE) with large predictions can lead to large gradients, which further exacerbate the issue of exploding outputs.

Conclusion

Therefore, output values in a neural network may explode due to factors such as large weight initialization, unbounded activation functions, the accumulation of values in deep networks, exploding gradients during backpropagation, high learning rates, and lack of regularization. These issues can lead to instability in training and poor model performance. To mitigate these problems, techniques such as careful weight initialization, gradient clipping, using appropriate activation functions, and regularization methods are often employed.

3) ReLU function sometime has the effect of regularization (making the model compact) especially when many input values are negative. Explain this in detail using the formula of ReLU.

The ReLU (Rectified Linear Unit) activation function is defined mathematically as:

$$\text{ReLU}(x)=\max(0,x)$$

This means that for any input x, if x is negative, the output will be zero; if x is positive, the output will be equal to x. This simple but effective property of ReLU can lead to regularization effects, particularly when many input values are negative.

1. Sparsity of Activations

- **Zeroing Out Negative Inputs**: When many input values to a neuron are negative, the ReLU function outputs zero for those inputs. This results in a sparse activation pattern, where only a subset of neurons are active (i.e., producing non-zero outputs) for a given input.

- **Effect on Model Complexity**: This sparsity effectively reduces the number of active parameters during training, as many neurons contribute nothing to the output. The model becomes "compact" because it relies on fewer active neurons, which can help prevent overfitting by discouraging reliance on every feature.

2. Dynamic Neuron Activation

- **Variable Neuron Contribution**: Since the activation of neurons depends on the input values, different subsets of neurons may be activated for different inputs. This dynamic selection of active neurons means that the model learns to use a diverse set of features rather than all features at once.
- **Encouraging Robust Features**: By only activating certain neurons for specific inputs, the model is encouraged to learn more robust and generalized features, as it cannot rely on any single neuron or small group of neurons for all predictions.

3. Mitigation of Overfitting

- **Regularization Effect**: The inherent sparsity introduced by ReLU can act as a form of implicit regularization. With many neurons outputting zero, the model has fewer parameters effectively contributing to the learning process, which can help it generalize better to unseen data.
- **Reduced Complexity**: A model with fewer active neurons at any given time has a reduced effective capacity, which can help in avoiding overfitting to the training data. This is especially beneficial in scenarios where the training dataset is small relative to the model complexity.

4. Gradient Flow and Training Stability

- **Gradient Behavior**: During backpropagation, when ReLU outputs zero, the gradients for those neurons are also zero. This means that the weights associated with inactive neurons do not get updated. This can lead to certain neurons becoming permanently inactive (often referred to as "dying ReLU"), which further contributes to model compactness.
- **Focus on Active Neurons**: The network is forced to focus on learning the weights associated with the active neurons, which can lead to more efficient learning and a more compact representation of the data.

5. Comparison to Other Activation Functions

- **Contrast with Sigmoid and Tanh**: Unlike sigmoid or tanh functions, which produce outputs across a continuous range (including small positive values), ReLU's behavior of zeroing out negative inputs creates a stark contrast in neuron activation. This makes ReLU particularly effective in creating compact models.

Conclusion

Therefore, the ReLU activation function can have a regularization effect by introducing sparsity in the activation of neurons, especially when many input values are negative. This sparsity leads to a more compact model, reduces the effective number of parameters, encourages robust feature learning, and mitigates overfitting. The unique properties of ReLU, including its zeroing out of negative inputs and dynamic activation patterns, make it a powerful tool for building efficient neural networks.

5.[3 pts] If we use batch gradient with a large dataset and error function is mean squared error (not sum squared error), we may have a vanishing gradient. Explain the reason USING the formula of batch update gradient.

When using batch gradient descent with a large dataset and the mean squared error (MSE) as the error function, the potential for vanishing gradients arises due to the way gradients are computed and updated in the learning process.

1. Mean Squared Error (MSE) Definition

The mean squared error is defined as:

$$\mathbf{E}=\frac{1}{n}\sum_{i=1}^{N}(y_i-\hat{y}_i)^2$$

where:

- E is the mean squared error,
- N is the number of samples in the batch,
- y_i is the true value,
- $\hat{y}_i$ is the predicted value.

2. Gradient Calculation

To update the weights using gradient descent, we compute the gradient of the error with respect to the weights. The gradient of the MSE with respect to a weight w_j is given by:

$$\frac{\partial E}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N} -2(y_i - \hat{y}_i)\frac{\partial \hat{y}_i}{\partial w_j}$$

This formula shows that the gradient depends on the error term $(y_i - \hat{y}_i)$ and the derivative of the predicted output with respect to the weights.

3. Vanishing Gradient Problem

The vanishing gradient problem occurs when the gradients become very small as they are propagated backward through the network during training.

- Activation Functions: If the neural network uses activation functions like sigmoid or tanh, their derivatives can become very small for inputs that are far from the origin (either very positive or very negative). For example, the derivative of a sigmoid function approaches zero as the input moves away from zero.
- Gradient Propagation: During backpropagation, the gradients are multiplied by the derivatives of the activation functions at each layer. If these derivatives are small (due to the reasons mentioned), the gradient of the loss with respect to the weights can diminish exponentially as it moves backward through the layers.

4. Batch Update Formula

In batch gradient descent, the weights are updated using the computed gradient as follows:

$$\omega_j \leftarrow \omega_j - \eta\frac{\partial E}{\partial w_j}$$

where η is the learning rate.

If the gradient $\frac{\partial E}{\partial w_j}$ becomes very small (due to vanishing gradients), the update step $\eta\frac{\partial E}{\partial w_j}$ will also be very small. This leads to minimal changes in the weights, effectively stalling the learning process.

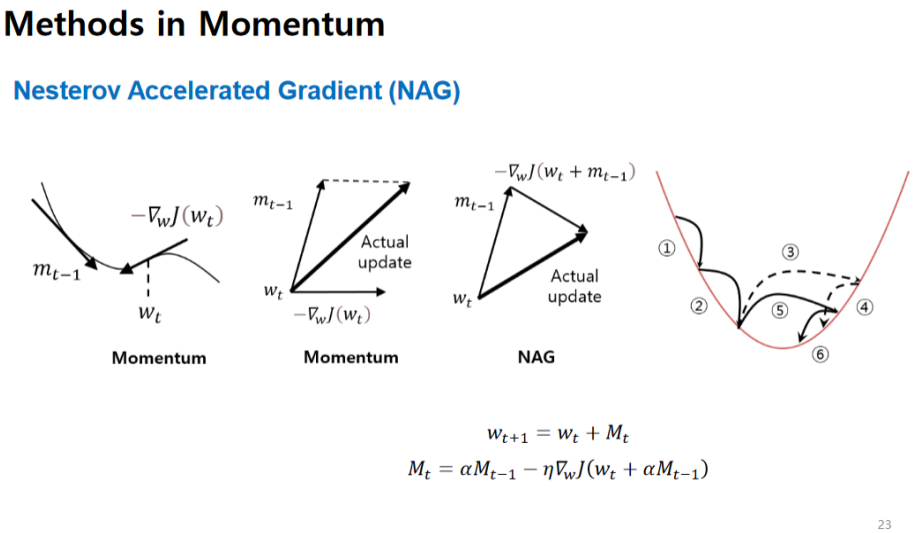
5. Impact of Large Datasets

- Large Datasets: When using a large dataset, the averaging effect in MSE can exacerbate the vanishing gradient problem. The gradients computed from a large number of samples can converge to very small values, particularly if the dataset contains many instances where the predictions are close to the true values.
- Overall Effect: As a result, the model may struggle to learn effectively, as the updates to the weights become negligible, and the training process can stagnate.

Conclusion

Therefore, when using batch gradient descent with mean squared error, the potential for vanishing gradients arises due to the small derivatives of activation functions, especially in deep networks. The formula for the gradient update shows that if the gradients diminish, the weight updates will also be minimal, leading to slow or stalled learning. This phenomenon is particularly pronounced in large datasets where the averaging of errors can lead to very small gradients, compounding the issue.

7. [4 pts] In p. 23 in "gradient-descent" slide, explain the rightmost picture.



The rightmost picture in the "Nesterov Accelerated Gradient (NAG)" section illustrates the key steps of the NAG algorithm.

1. The current weight is denoted as w_t.
2. The momentum term m_{t-1} is computed based on the previous momentum and the gradient of the loss function with respect to the previous weight w_{t-1}.

3. The "Actual update" step shows how the new weight w_{t+1} is computed by taking a step in the direction of the sum of the current weight w_t and the momentum term m_t.
4. The momentum term m_t is updated using the formula shown at the bottom of the image, which includes the previous momentum term m_{t-1} and the gradient of the loss function with respect to the current weight w_t

This NAG algorithm is designed to accelerate the convergence of gradient descent by incorporating a momentum term that helps the optimization process overcome local minima and saddle points more effectively compared to standard gradient descent.

8. [4 pts] In p. 29 in "gradient-descent" slide, explain why the method uses 'n_in' parameter.

The "n_in" parameter of the "gradient-descent" slide refers to the number of input features or dimensions in the dataset.

The reason why the method uses the "n_in" parameter is related to the initialization of the weights in the neural network.

When training a neural network, the weights are typically initialized to small random values. The scale of these initial weights can have a significant impact on the performance and convergence of the training process.

If the initial weights are too small, the gradients computed during backpropagation may become very small, leading to a vanishing gradient problem and slow learning. Conversely, if the initial weights are too large, the gradients may become too large, leading to an exploding gradient problem and numerical instability.

To address this issue, the method uses the "n_in" parameter to scale the initial weights based on the number of input features. Specifically, the initial weights are typically sampled from a normal distribution with a standard deviation inversely proportional to the square root of the number of input features, i.e., 1/sqrt(n_in).

This initialization scheme, known as the Xavier or Glorot initialization, helps to ensure that the initial activations and gradients have appropriate scales, which can improve the stability and convergence of the training process.

By using the "n_in" parameter, the method can automatically adjust the scale of the initial weights based on the dimensionality of the input data, making the training more robust and less sensitive to the specific choice of initial weights.

9. [4 pts] Explain why we can NOT use the gradient method in hyperparameter optimization. Explain this USING MSE error function with L2 regularization.

The gradient method, such as gradient descent, is typically not suitable for hyperparameter optimization, even when using an error function like Mean Squared Error (MSE) with L2 regularization.

The key reason is that hyperparameters are not part of the neural network's trainable parameters, and therefore, the gradients with respect to the hyperparameters cannot be computed in the same way as for the weights and biases.

Let's consider the MSE error function with L2 regularization:

$$\mathbf{E = (1/N) * \Sigma (y_i - f(x_i, w))^2 + \lambda * \Sigma w_j^2}$$

Where:

- N is the number of training samples
- y_i is the true label for the i-th sample
- $f(x_i, w)$ is the model's prediction for the i-th sample, given the input x_i and the model parameters w
- w_j is the j-th model parameter (weight or bias)
- λ is the L2 regularization hyperparameter

The gradients of the error function E with respect to the model parameters w can be computed using backpropagation. However, the hyperparameter λ is not a part of the model parameters w, and therefore, the gradient $\partial E/\partial \lambda$ cannot be computed in the same way.

The hyperparameters, such as the learning rate, regularization strength, or the number of layers, are not directly involved in the forward and backward propagation of the neural network. They are external to the model and control the training process itself.

As a result, the gradient-based optimization methods, like gradient descent, cannot be directly applied to optimize the hyperparameters. The gradients with respect to the hyperparameters are not available, and the error function is not differentiable with respect to the hyperparameters.

Instead, other optimization techniques, such as grid search, random search, or Bayesian optimization, are typically used for hyperparameter tuning. These methods explore the hyperparameter space in a more systematic or guided way, without relying on gradient information.

10. For the following confusion matrix

	Pred Pos	**Pred Neg**
Actual Pos	**50**	**10**
Actual Neg	**5**	**100**

Based on the given confusion matrix, we can calculate the following performance metrics:

1. Accuracy: Accuracy = (True Positives + True Negatives) / (True Positives + True Negatives + False Positives + False Negatives)

   Accuracy = (50 + 100) / (50 + 10 + 5 + 100)

   = 150 / 165

   = 0.9090 or 90.90%

2. Precision: Precision = True Positives / (True Positives + False Positives)

   Precision = 50 / (50 + 5)

   = 50 / 55

   = 0.9091 or 90.91%

3. Recall (True Positive Rate): Recall = True Positives / (True Positives + False Negatives)

   Recall = 50 / (50 + 10)

= 50 / 60

= 0.8333 or 83.33%

4. False Positive Rate: False Positive Rate = False Positives / (False Positives + True Negatives)

   False Positive Rate = 5 / (5 + 100)

   = 5 / 105

   = 0.0476 or 4.76%

5. F1-score: F1-score = 2 * (Precision * Recall) / (Precision + Recall)

   F1-score = 2 * (0.9091 * 0.8333) / (0.9091 + 0.8333)

   = 1.5152 / 1.7424

   = 0.8684 or 86.84%

6. Specificity (True Negative Rate) = True Negatives / (True Negatives + False Positives)
   Specificity = 100 / (100 + 5)
   = 0.9524 or 95.24%

Therefore, the key performance metrics calculated from the given confusion matrix are:

- Accuracy: 90.90%
- Precision: 90.91%
- Recall (True Positive Rate): 83.33%
- False Positive Rate: 4.76%
- F1-score: 86.84%
- Specificity: 95.24%

2) [6 pts] draw a ROC graph using the following values.

True Labels: [1, 0, 1, 0]

Predicted Probabilities: [0.8, 0.3, 0.6]

For the ROC curve, we need to calculate the True Positive Rate (TPR) and False Positive Rate (FPR) at different thresholds.

Given:

- True Labels: [1, 0, 1, 0]
- Predicted Probabilities: [0.8, 0.3, 0.6]

Steps:

1. Sort predicted probabilities in descending order with corresponding true labels:

Sorted predicted probabilities:[0.8,0.6,0.3]

True labels:[1,1,0]

2. Plot points on the ROC curve by considering various thresholds.

We will calculate TPR and FPR at each threshold as follows:

Threshold = 0.8:

Predicted positive: Only the first instance (probability 0.8) is predicted as positive.

TP = 1 (because the true label for this instance is 1)

FP = 0 (no false positives)

FN = 2 (remaining instances are false negatives)

TN = 0 (no true negatives)

$TPR = \frac{TP}{TP+FN} = \frac{1}{1+2} = 0.333$

$FPR = \frac{FP}{FP+TN} = \frac{0}{0+0} = 0$

Threshold = 0.6:

Predicted positive: The first two instances (probabilities 0.8 and 0.6) are predicted as positive.

TP = 2 (both the first and second instances are true positives)

FP = 0 (no false positives)

FN = 1 (remaining instance is a false negative)

TN = 0 (no true negatives)

$TPR = \frac{TP}{TP+FN} = \frac{2}{2+1} = 0.667$

$FPR = \frac{FP}{FP+TN} = \frac{0}{0+0} = 0$

Threshold = 0.3:

Predicted positive: All three instances are predicted as positive.

TP = 2 (the first two instances are true positives)

FP = 1 (the third instance is a false positive)

FN = 0 (no false negatives)

TN = 1 (one true negative)

$TPR = \frac{TP}{TP+FN} = \frac{2}{2+0} = 1$

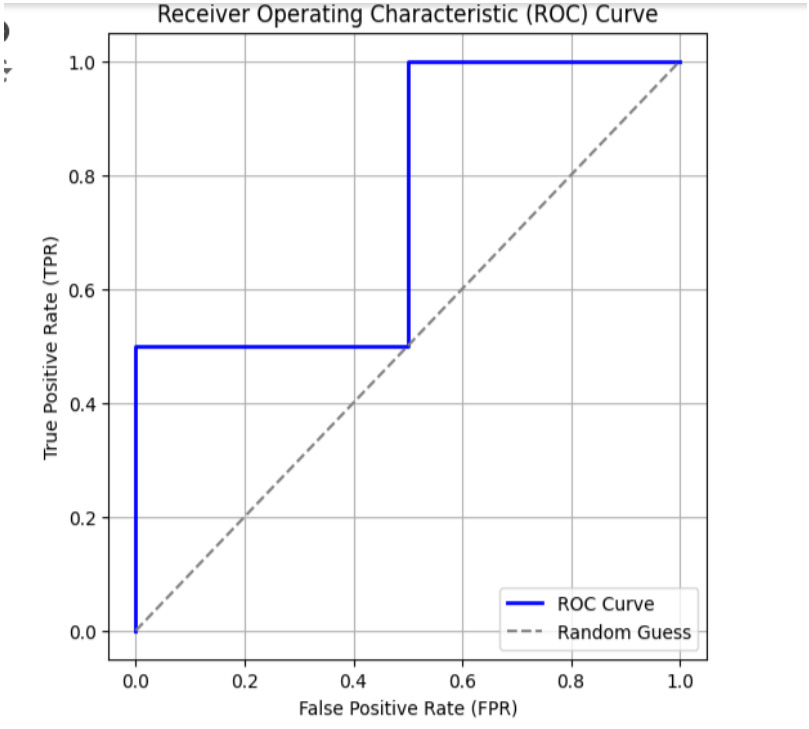
$FPR = \frac{FP}{FP+TN} = \frac{1}{1+1} = 0.5$

ROC Curve Points:

FPR = 0, TPR = 0.333

FPR = 0, TPR = 0.667

FPR = 0.5, TPR = 1



After the execution of the code the graph is given as above.

11. [8 pts] Using the following table, verify whether algo 1 and algo 2 are statistically similar or different using paired t-test.

experiments	Algo 1	Algo 2
1	70	75
2	65	68
3	80	82
4	90	94
5	60	66

To determine whether Algo 1 and Algo 2 are statistically similar or different using a paired t-test, we can follow these steps:

1. Calculate the differences between the paired observations.
2. Compute the mean and standard deviation of the differences.
3. Perform the paired t-test.

Step 1: Calculate Differences

Experiment	Algo 1	Algo 2	Difference (Algo 1 - Algo 2)
1	70	75	-5
2	65	68	-3
3	80	82	-2
4	90	94	-4
5	60	66	-6

Step 2: Calculate Mean and Standard Deviation of Differences

- **Differences**: -5, -3, -2, -4, -6
- **Mean of Differences** ($\bar{d}$):

  $(\bar{d}) = \frac{-5-3-2-4-6}{5} = \frac{-20}{5} = -4$
- **Standard Deviation of Differences** (S_d):

  $(S_d) = \sqrt{\frac{\sum(d_1-d)^2}{n-1}}$

  Where d_i are the differences and n is the number of pairs.

d_1 &= -5 \quad (d_1 - \bar{d})^2 = (-5 + 4)^2 = 1

d_2 &= -3 \quad (d_2 - \bar{d})^2 = (-3 + 4)^2 = 1

d_3 &= -2 \quad (d_3 - \bar{d})^2 = (-2 + 4)^2 = 4

d_4 &= -4 \quad (d_4 - \bar{d})^2 = (-4 + 4)^2 = 0

d_5 &= -6 \quad (d_5 - \bar{d})^2 = (-6 + 4)^2 = 4

Now, summing these squared differences: $\sum(d_i - d)^2$=1+1+4+0+4=10

Then, calculate $S_d = \sqrt{\frac{10}{5-1}} = \sqrt{\frac{10}{4}}$ = √2.5 ≈1.58

Step 3: Perform the Paired T-Test

The t-statistic is calculated as:

$t = \frac{\bar{d}}{s_d/\sqrt{n}} = \frac{-4}{1.58/\sqrt{5}} = \frac{-4}{0.7071}$ = -5.66

Step 4: Determine Degrees of Freedom and Critical Value

- **Degrees of Freedom (df)**: n−1=5−1=4

Using a t-table for df = 4, we can find the critical value for a two-tailed test at a significance level of 0.05. The critical t-value is approximately ±2.776.

Conclusion

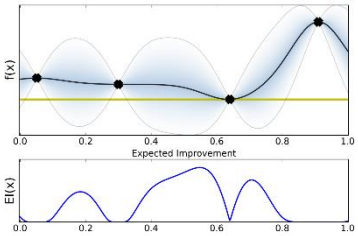
- **Calculated t-value**: -5.66
- **Critical t-values**: ±2.776

Since the absolute value of the calculated t-value (-5.66) is greater than the critical value (±2.776), we reject the null hypothesis.

Result

There is significant statistical evidence to conclude that Algo 1 and Algo 2 are different.

12. [5 pts] In the following acquisition function in Bayesian optimization, explain why the X values near X=5.5 is the best choice for the next parameter value.



In Bayesian optimization, the acquisition function is used to determine the next point to sample based on the current model of the objective function. The choice of the next parameter value is influenced by the trade-off between exploration and exploitation.

Why X values near X=5.5 are the Best Choice:

1. **Exploitation of Known Information**:

   If the acquisition function indicates that the expected value of the objective function is high near X=5.5, it suggests that there is a region of the parameter space where the function has been evaluated and yielded favorable results. Sampling in this area can lead to improved results based on existing knowledge.

2. **Uncertainty Reduction**:

   The acquisition function often balances the expected improvement with the uncertainty in the model. If the uncertainty is lower near X=5.5, it means that the model is more confident about the performance of the objective function in this area. This can make it a prime candidate for the next sample.

3. **Gradient Information**:

   If the acquisition function shows a steep gradient or rapid improvement near X=5.5, it indicates that small changes around this value could yield significant improvements in the objective function. This is a strong signal to sample in that vicinity.

4. **Local Optima**:

   If previous evaluations of the objective function around X=5.5 have shown promising results, the algorithm may favor this area to exploit this local optimum further.

5. **Trade-off Between Exploration and Exploitation**:

   The acquisition function is designed to balance exploration (trying new, uncertain areas) and exploitation (sampling known good areas). If X=5.5 is identified as a good candidate, it suggests that the function is likely to yield better results with less risk, making it a strategic choice.

Conclusion

Choosing X values near 5.5 is advantageous because it leverages existing information to maximize the expected improvement while minimizing uncertainty. This approach is fundamental in Bayesian optimization, where the goal is to efficiently find the optimum of an expensive-to-evaluate function.

13. [5 pts] In Bayesian optimization, the functions generated in Gaussian process are smooth (e.g., the functions in p. 24) Explain the reason USING the following kernel function.

$$\boldsymbol{k(x_i,\ x_j) = \exp(-\lambda * \|x_i\ - x_j\|^2)}$$

The kernel function provided is the Radial Basis Function (RBF) kernel, also known as the Gaussian kernel:

$$\boldsymbol{k(x_i,\ x_j) = \exp(-\lambda * \|x_i\ - x_j\|^2)}$$

where λ>0 is a hyperparameter that controls the scale of the similarity between $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$

Here's why this kernel leads to smooth functions in Gaussian Process (GP) regression

Where:

- $\boldsymbol{k(x_i,\ x_j)}$ is the kernel function value between the input vectors $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$
- **λ (lambda)** is a hyperparameter that controls the smoothness of the function.
- $\boldsymbol{\|x_i\ - x_j\|}$ is the Euclidean distance between the input vectors $\boldsymbol{x_i}$ and $\boldsymbol{x_j}$

The reason for using this Gaussian kernel function in Bayesian optimization is that it generates smooth functions in the Gaussian process. This smoothness property is desirable because it reflects the assumption that the underlying objective function being optimized is also smooth.

The Gaussian kernel function has the following properties that contribute to the smoothness of the generated functions:

1. Continuity: The Gaussian kernel function is a continuous function, which means that small changes in the input vectors x_i and x_j result in small changes in the kernel function value $k(x_i, x_j)$. This continuity translates to the smoothness of the generated functions.

2. Differentiability: The Gaussian kernel function is infinitely differentiable, which means that it has derivatives of all orders. This property allows the Gaussian process to model functions with smooth gradients, which is important for optimization algorithms that rely on gradient information.

3. Locality: The Gaussian kernel function exhibits a locality property, where the kernel function value decreases rapidly as the distance between the input vectors increases. This means that nearby points in the input space have a stronger influence on the function value compared to distant points, contributing to the smoothness of the generated functions.

By using the Gaussian kernel function, the Bayesian optimization algorithm can generate smooth functions that reflect the underlying properties of the objective function being optimized. This smoothness is advantageous for optimization algorithms, as it allows them to efficiently navigate the function landscape and converge to the optimal solution.