Bias-Variance Tradeoff, Logistic Regression, and Neural Networks

CS145: Introduction to Data Mining

Spring 2024

Contents

1 Introduction

This lecture covers fundamental concepts in machine learning, including the bias-variance tradeoff, logistic regression, and neural networks. We will explore the mathematical formulations, optimization techniques, and practical considerations for each topic. By the end of this lecture, you will have a solid understanding of these key concepts and their applications in data mining and machine learning.

2 Bias-Variance Tradeoff

The bias-variance tradeoff is a fundamental concept in machine learning that aims to balance the model's ability to fit the training data (bias) and its ability to generalize to new, unseen data (variance).

2.1 Bias

Bias refers to the error introduced by approximating a real-world problem with a simplified model. A model with high bias is typically too simple to capture the underlying patterns in the data, leading to underfitting.

2.2 Variance

Variance refers to the model's sensitivity to small fluctuations in the training data. A model with high variance is overly complex and fits the noise in the training data, leading to overfitting and poor generalization to new data.

2.3 Decomposition of Mean Squared Error (MSE)

The mean squared error (MSE) can be decomposed into bias, variance, and irreducible error terms:

$$\text{MSE} = \text{Bias}[f(x)]^2 + \text{Var}[f(x)] + \sigma^2 \tag{1}$$

where $f(x)$ is the learned model, and σ^2 is the irreducible error due to noise in the data.

To derive this decomposition, let's consider a true model $y = f + \epsilon$, where f is the true underlying function and ϵ is the noise with zero mean and variance σ^2. The expected squared error can be written as:

$$\begin{aligned}
\text{MSE} &= \mathbb{E}[(y - f(x))^2] \\
&= \mathbb{E}[(f + \epsilon - f(x))^2] \\
&= \mathbb{E}[(f - f(x))^2] + 2\mathbb{E}[(f - f(x))\epsilon] + \mathbb{E}[\epsilon^2] \\
&= \mathbb{E}[(f - f(x))^2] + 0 + \sigma^2 \\
&= \mathbb{E}[(f - \mathbb{E}[f(x)] + \mathbb{E}[f(x)] - f(x))^2] + \sigma^2 \\
&= (\mathbb{E}[f(x)] - f)^2 + \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2] + \sigma^2 \\
&= \text{Bias}[f(x)]^2 + \text{Var}[f(x)] + \sigma^2
\end{aligned}$$

2.4 Bias-Variance Tradeoff

The goal of machine learning is to find a model that minimizes both bias and variance simultaneously. However, in practice, there is often a tradeoff between bias and variance:

- Increasing model complexity typically reduces bias but increases variance.
- Decreasing model complexity typically increases bias but reduces variance.

The optimal model complexity is the one that achieves the best balance between bias and variance, minimizing the overall MSE.

2.5 Underfitting and Overfitting

Underfitting occurs when a model has high bias and low variance. The model is too simple to capture the underlying patterns in the data, resulting in poor performance on both training and test data.

Overfitting occurs when a model has low bias and high variance. The model is too complex and fits the noise in the training data, resulting in excellent performance on the training data but poor generalization to new, unseen data.

3 Logistic Regression

Logistic regression is a popular algorithm for binary classification problems. It models the probability of an instance belonging to a particular class given its features.

3.1 Sigmoid Function

The sigmoid function, also known as the logistic function, is used to map the output of a linear model to a probability between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

where z is the output of the linear model.

3.2 Logistic Regression Model

In logistic regression, the probability of an instance $\boldsymbol{x}$ belonging to class 1 is modeled as:

$$P(y = 1 \mid \boldsymbol{x}, \boldsymbol{\beta}) = \sigma(\boldsymbol{x}^\mathsf{T}\boldsymbol{\beta}) = \frac{1}{1 + e^{-\boldsymbol{x}^\mathsf{T}\boldsymbol{\beta}}} \tag{3}$$

where $\boldsymbol{\beta}$ is the vector of model parameters.

3.3 Maximum Likelihood Estimation (MLE)

The model parameters $\boldsymbol{\beta}$ are estimated using maximum likelihood estimation (MLE). The likelihood function for logistic regression is:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} P(y_i \mid \boldsymbol{x}_i, \boldsymbol{\beta})^{y_i} (1 - P(y_i \mid \boldsymbol{x}_i, \boldsymbol{\beta}))^{1-y_i} \tag{4}$$

The log-likelihood is:

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \log(P(y_i \mid \boldsymbol{x}_i, \boldsymbol{\beta})) + (1 - y_i) \log(1 - P(y_i \mid \boldsymbol{x}_i, \boldsymbol{\beta})) \tag{5}$$

The MLE estimates of $\boldsymbol{\beta}$ are obtained by maximizing the log-likelihood using optimization techniques such as gradient descent.

3.4 Extension to Multiclass Classification

Logistic regression can be extended to handle multiclass classification problems using the softmax function. In a K-class problem, the probability of an instance $\boldsymbol{x}$ belonging to class k is:

$$P(y = k \mid \boldsymbol{x}, \boldsymbol{W}) = \frac{e^{\boldsymbol{x}^\mathsf{T}\boldsymbol{\beta}_k}}{\sum_{j=1}^{K} e^{\boldsymbol{x}^\mathsf{T}\boldsymbol{\beta}_j}} \tag{6}$$

where $\boldsymbol{W} = [\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_K]$ is the matrix of model parameters.

The log-likelihood for multiclass logistic regression is:

$$\ell(\boldsymbol{W}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbf{1}\{y_i = k\} \log(P(y_i = k \mid \boldsymbol{x}_i, \boldsymbol{W})) \tag{7}$$

where $\mathbf{1}\{\cdot\}$ is the indicator function.

4 Neural Networks

Neural networks are powerful machine learning models inspired by the structure and function of biological neurons. They consist of interconnected layers of nodes (neurons) that process and transform input data to make predictions or decisions.

4.1 Multilayer Perceptron (MLP)

A multilayer perceptron (MLP) is a type of feedforward neural network with one or more hidden layers between the input and output layers. Each neuron in an MLP computes a weighted sum of its inputs, applies a nonlinear activation function, and passes the result to the next layer.

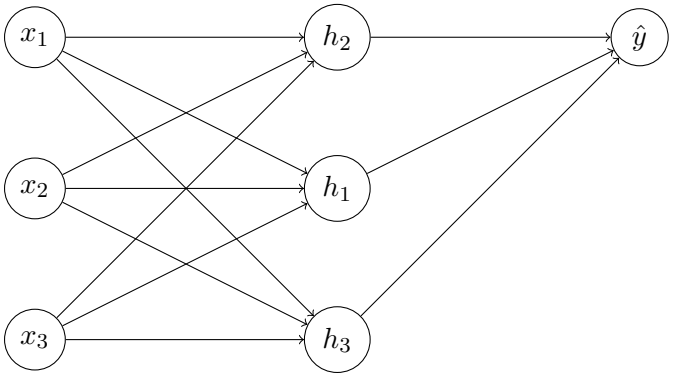

Figure 1: A simple multilayer perceptron with one hidden layer.

4.2 Nonlinearities (Activation Functions)

Nonlinear activation functions are crucial for neural networks to learn complex, nonlinear relationships in the data. Some popular activation functions include:

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent (tanh): $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Rectified Linear Unit (ReLU): $\text{ReLU}(z) = \max(0, z)$

4.3 Forward Propagation

Forward propagation is the process of computing the outputs of a neural network given its inputs. For an MLP with L layers, the forward propagation equations are:

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \tag{8}$$

$$\boldsymbol{a}^{(l)} = f^{(l)}(\boldsymbol{z}^{(l)}) \tag{9}$$

where $\boldsymbol{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weights and biases of layer l, $\boldsymbol{a}^{(l-1)}$ is the output of the previous layer (or the input for the first layer), and $f^{(l)}$ is the activation function of layer l.

4.4 Backpropagation

Backpropagation is an efficient algorithm for computing the gradients of the loss function with respect to the model parameters. It uses the chain rule to propagate the gradients from the output layer back to the input layer.

For an MLP with a loss function $J(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents all model parameters, the backpropagation equations are:

$$\frac{\partial J}{\partial \boldsymbol{z}^{(L)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(L)}} \odot f'^{(L)}(\boldsymbol{z}^{(L)}) \tag{10}$$

$$\frac{\partial J}{\partial \boldsymbol{W}^{(l)}} = \frac{\partial J}{\partial \boldsymbol{z}^{(l)}} \boldsymbol{a}^{(l-1)\mathsf{T}} \tag{11}$$

$$\frac{\partial J}{\partial \boldsymbol{b}^{(l)}} = \frac{\partial J}{\partial \boldsymbol{z}^{(l)}} \tag{12}$$

$$\frac{\partial J}{\partial \boldsymbol{a}^{(l-1)}} = \boldsymbol{W}^{(l)\mathsf{T}} \frac{\partial J}{\partial \boldsymbol{z}^{(l)}} \tag{13}$$

where $\odot$ denotes element-wise multiplication, and $f'^{(l)}$ is the derivative of the activation function of layer l. The gradients are computed iteratively from the output layer back to the input layer.

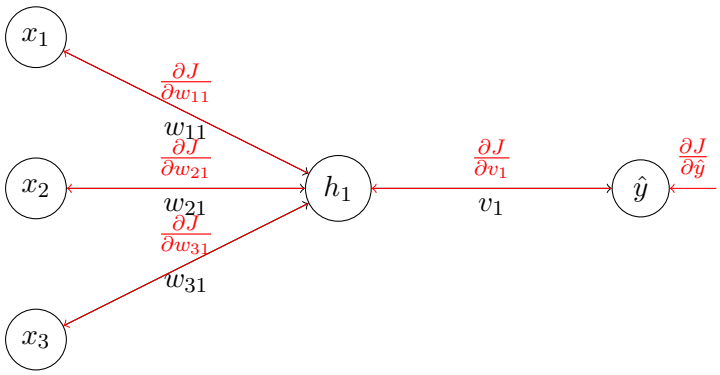

Figure 2: Gradient flow for a single neuron during backpropagation.

Example. Now, let's derive the chain rule formulation for backpropagation using the Mean Squared Error (MSE) as the objective function. Given an input vector $\boldsymbol{x} = [x_1, x_2, x_3]^\mathsf{T}$, the weight matrix for the hidden layer $\boldsymbol{W}$, and the weight vector for the output layer $\boldsymbol{v}$, we can compute the output of the neural network as follows:

$$\boldsymbol{z} = \boldsymbol{W}^\mathsf{T}\boldsymbol{x} \tag{14}$$

$$\boldsymbol{h} = \sigma(\boldsymbol{z}) \tag{15}$$

$$\hat{y} = \boldsymbol{v}^\mathsf{T}\boldsymbol{h} \tag{16}$$

where σ is the activation function applied element-wise to the hidden layer activations. The MSE loss function for a single training example (x, y) can be written as:

$$J = \frac{1}{2}(\hat{y} - y)^2 \tag{17}$$

To compute the gradients using backpropagation, we first calculate the gradient of the loss with respect to the output:

$$\frac{\partial J}{\partial \hat{y}} = \hat{y} - y \tag{18}$$

Next, we compute the gradient of the loss with respect to the output layer weights:

$$\frac{\partial J}{\partial \boldsymbol{v}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{v}} = (\hat{y} - y)\boldsymbol{h} \tag{19}$$

To compute the gradient of the loss with respect to the hidden layer weights, we first calculate the gradient of the loss with respect to the hidden layer activations:

$$\frac{\partial J}{\partial \boldsymbol{h}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{h}} = (\hat{y} - y)\boldsymbol{v} \tag{20}$$

Then, we compute the gradient of the loss with respect to the hidden layer pre-activations:

$$\frac{\partial J}{\partial \boldsymbol{z}} = \frac{\partial J}{\partial \boldsymbol{h}} \odot \sigma'(\boldsymbol{z}) \tag{21}$$

where $\odot$ denotes element-wise multiplication and σ' is the derivative of the activation function. Finally, we compute the gradient of the loss with respect to the hidden layer weights:

$$\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{x} \left(\frac{\partial J}{\partial \boldsymbol{z}} \right)^{\mathsf{T}} \tag{22}$$

The weights can then be updated using gradient descent:

$$\boldsymbol{v} := \boldsymbol{v} - \alpha \frac{\partial J}{\partial \boldsymbol{v}} \tag{23}$$

$$\boldsymbol{W} := \boldsymbol{W} - \alpha \frac{\partial J}{\partial \boldsymbol{W}} \tag{24}$$

where α is the learning rate. This process is repeated for multiple epochs, using the entire training dataset, until the network converges or a stopping criterion is met.

4.5 Vanishing and Exploding Gradients

Deep neural networks can suffer from the vanishing or exploding gradient problem, which makes training difficult. Vanishing gradients occur when the gradients become extremely small as they are propagated back through the network, making it hard for the parameters in the early layers to learn. Exploding gradients occur when the gradients become extremely large, causing unstable updates and divergence.

To mitigate these problems, several techniques can be used:

- Careful initialization of weights (e.g., Xavier or He initialization)
- Using activation functions with stable gradients (e.g., ReLU)
- Gradient clipping to prevent exploding gradients
- Batch normalization to normalize activations and stabilize training

4.6 Batch Normalization

Batch normalization is a technique that normalizes the activations of a layer for each mini-batch during training. It helps stabilize the training process, reduces the sensitivity to the choice of initialization, and allows for higher learning rates.

Given a mini-batch of activations $\boldsymbol{x} = (x_1, \ldots, x_m)$, batch normalization performs the following

transformation:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{25}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{26}$$

$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{27}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{28}$$

where $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are the mean and variance of the mini-batch, ϵ is a small constant for numerical stability, and γ and β are learnable parameters that allow the network to adjust the scale and shift of the normalized activations if needed.

Batch normalization is typically applied before the nonlinear activation function and can significantly improve the training speed and generalization performance of deep neural networks.

5 Discussions

In this lecture, we covered several key concepts in machine learning, including the bias-variance tradeoff, logistic regression, and neural networks. Understanding these concepts is crucial for developing effective machine learning models and avoiding common pitfalls such as underfitting, overfitting, and training instability.

The bias-variance tradeoff highlights the importance of finding the right balance between model complexity and generalization performance. Logistic regression provides a simple yet powerful method for binary and multiclass classification problems. Neural networks, particularly deep neural networks, offer a flexible and expressive framework for learning complex, nonlinear relationships in data.

Techniques such as careful initialization, suitable activation functions, gradient clipping, and batch normalization can help mitigate the challenges associated with training deep neural networks, such as vanishing and exploding gradients.

As you continue your journey in data mining and machine learning, keep these concepts in mind and apply them to build robust and effective models for various real-world problems.