Machine Learning

UAS

Zalva Ihilani Pasha

1103194182

TELKOM UNIVERSITY

BANDUNG

2023

Contents

Tensor Basics 3
Autograd 5
Backpropagation 6
Gradient Descent 7
  Manually 7
  Auto 8
Training Pipeline 10
  No model 10
  Model 11
Linear Regression 13
Logistic Regression 14
Dataset and Dataloader 15
Dataset Transforms 16
Softmax and Crossentropy 17
Activation Functions 18
  Base 18
  Plot Activations 19
Feed Forward Net 20
CNN 21
Transfer Learning 22
Tensorboard 23
Save & Load Models 24

Tensor Basics

1. Introduction

PyTorch is a popular open-source deep learning framework that provides a flexible and efficient way to build and train neural networks. At the core of PyTorch are tensor operations, which enable efficient numerical computations on multi-dimensional arrays. This technical report aims to provide an overview of tensor operations in PyTorch through a series of code examples.

2. Tensor Creation

In PyTorch, tensors are the primary data structure used for storing and manipulating numerical data. Tensors can have different dimensions, ranging from scalars (0-dimensional tensors) to higher-dimensional tensors. The code demonstrates the creation of tensors with various dimensions using different initialization methods.

- **torch.empty(size)**: Creates an uninitialized tensor with the specified size. Examples are shown for scalar, vector, matrix, and 3D tensor.
- **torch.rand(size)**: Generates a tensor filled with random numbers from a uniform distribution between 0 and 1.
- **torch.zeros(size)**: Creates a tensor filled with zeros.
- **torch.ones(size)**: Creates a tensor filled with ones.
- **torch.tensor(data)**: Constructs a tensor from existing data, such as a Python list or NumPy array. The size of the tensor is inferred from the data.

3. Tensor Properties

Once tensors are created, it is essential to understand their properties, such as size and data type. The code demonstrates how to access these properties using various methods.

- **tensor.size()**: Returns the size of the tensor, indicating the number of elements in each dimension.
- **tensor.dtype**: Returns the data type of the tensor.
- **tensor.requires_grad**: Specifies whether the tensor requires gradient calculations for optimization purposes. If **requires_grad** is set to **True**, PyTorch will track operations on this tensor to compute gradients during backpropagation.

4. Tensor Operations

PyTorch provides a rich set of mathematical operations that can be applied to tensors. The code examples illustrate basic element-wise operations on tensors.

- Addition: Performs element-wise addition of two tensors using the **+** operator or **torch.add()** function.
- Subtraction: Performs element-wise subtraction of two tensors using the **-** operator or **torch.sub()** function.
- Multiplication: Performs element-wise multiplication of two tensors using the ***** operator or **torch.mul()** function.

- Division: Performs element-wise division of two tensors using the **/** operator or **torch.div()** function.

5. Tensor Slicing

Slicing allows accessing specific elements or subsets of a tensor. The code demonstrates slicing operations on tensors.

- **tensor[:, index]**: Selects all rows from the tensor and a specific column indicated by **index**.
- **tensor[index, :]**: Selects a specific row indicated by **index** and all columns from the tensor.
- **tensor[row_index, col_index]**: Selects a specific element at the intersection of **row_index** and **col_index**.
- **tensor.item()**: Retrieves the actual value of a tensor with only one element.

6. Tensor Reshaping

Reshaping operations allow changing the dimensions of a tensor without modifying its data. The code showcases the usage of **torch.view()** to reshape tensors.

- **tensor.view(size)**: Returns a new tensor with the same data but a different shape specified by **size**. The size can be explicitly provided or inferred using **-1** for automatic calculation.

7. Tensor and NumPy Interoperability

PyTorch provides seamless integration with NumPy, a popular numerical computing library in Python. The code demonstrates how to convert tensors to NumPy arrays and vice versa.

- **tensor.numpy()**: Converts a PyTorch tensor to a NumPy array.
- **torch.from_numpy(array)**: Creates a PyTorch tensor from a NumPy array.
- Caution should be exercised when modifying tensors or arrays after conversion, as they may share the same memory location.

8. Tensor on GPU

PyTorch allows for GPU acceleration to leverage the computational power of graphics cards. The code demonstrates how to move tensors between CPU and GPU.

- **torch.cuda.is_available()**: Checks if a GPU is available for computation.
- **tensor.to(device)**: Moves a tensor to the specified device (CPU or GPU) for computation.
- **tensor.to("cuda")**: Alternative syntax for moving a tensor to the GPU.
- **tensor.to("cpu")**: Moves a tensor back to the CPU.
- Caution should be exercised when working with tensors on different devices, as certain operations may not be compatible or may require additional conversions.

In conclusion, this technical report provided an overview of tensor operations in PyTorch, covering tensor creation, properties, basic operations, slicing, reshaping, interoperability with NumPy, and GPU utilization. Understanding these fundamental concepts and operations is crucial for effectively working with PyTorch and developing deep learning models.

Autograd

1. Introduction

PyTorch is a popular deep learning framework that provides automatic differentiation capabilities through the **autograd** package. This technical report explores the concept of automatic differentiation and gradient computation in PyTorch, highlighting the usage of gradients for optimization purposes.

2. Automatic Differentiation

The **autograd** package in PyTorch allows for automatic differentiation, which enables the calculation of gradients for tensors involved in computations. The code examples illustrate the usage of automatic differentiation and its key components.

- **requires_grad=True**: By setting the **requires_grad** attribute of a tensor to **True**, PyTorch tracks all operations performed on that tensor for gradient computation.
- **grad_fn**: When a tensor is created as a result of an operation, it is associated with a **grad_fn** attribute that references the function responsible for creating the tensor.
- **backward()**: After completing the desired computation, calling the **backward()** function computes the gradients automatically using backpropagation.
- **grad**: The gradients of a tensor are accumulated in the **grad** attribute, which represents the partial derivative of the function with respect to the tensor.

3. Gradients for Non-Scalar Tensors

The computation of gradients becomes more involved when dealing with non-scalar tensors, i.e., tensors with more than one element. The code demonstrates how to handle gradients for non-scalar tensors.

- **backward(gradient)**: When a tensor is non-scalar, the **backward()** function requires an additional **gradient** argument, which specifies the shape of the gradient tensor.
- Vector-Jacobian Product: PyTorch's **autograd** engine computes partial derivatives by applying the chain rule, resulting in a vector-Jacobian product.

4. Controlling Gradient Tracking

In certain scenarios, it may be necessary to stop tracking the history of computations for a tensor or exclude specific operations from gradient computation. The code showcases various techniques for controlling gradient tracking.

- **requires_grad_(False)**: Modifying the **requires_grad** attribute in-place to **False** stops tracking the gradients for a tensor.
- **detach()**: Calling **detach()** on a tensor creates a new tensor with the same content but without gradient computation.
- **with torch.no_grad():**: Wrapping code within a **with torch.no_grad():** block temporarily stops gradient tracking for the enclosed operations.

5. Optimizing with Gradients

The gradients computed through automatic differentiation are crucial for optimizing models during training. The code provides an example of how to use gradients for model optimization.

- **zero_()**: Before each optimization step, it is important to zero out the gradients of the variables being optimized using **zero_()** to avoid accumulation of gradients from previous steps.
- Optimization Loop: The code demonstrates a simple optimization loop where the gradients are used to update the weights of the model.

6. Using Optimizers

PyTorch provides optimizers that automate the process of updating model parameters using gradients. The code mentions the usage of optimizers and provides an example of optimization with the SGD optimizer.

- **torch.optim.SGD**: The SGD optimizer is initialized with the parameters to be optimized and a learning rate (**lr**).
- **optimizer.step()**: The **step()** function updates the parameters based on the gradients and learning rate.
- **optimizer.zero_grad()**: The **zero_grad()** function is called to reset the gradients of the optimized parameters before the next optimization step.

In conclusion, this technical report provided an overview of automatic differentiation and gradient computation in PyTorch using the **autograd** package. Understanding these concepts is crucial for leveraging PyTorch's capabilities in training deep learning models and optimizing their parameters.

Backpropagation

1. Introduction

Gradient descent is a widely used optimization algorithm for adjusting the parameters of machine learning models. This technical report explores the implementation of gradient descent in PyTorch, showcasing how to compute gradients, update weights, and perform multiple optimization steps.

2. Initialization and Forward Pass

The code starts by initializing the input variables (**x** and **y**) and the parameter to be optimized (**w**). The **requires_grad=True** attribute is set for **w** to track its operations for gradient computation.

A forward pass is then performed to compute the predicted output (**y_predicted**) using the current value of **w**. The loss function is calculated as the squared difference between the predicted output and the true output (**loss**).

3. Backward Pass and Gradient Computation

The backward pass is executed to compute the gradients of the loss function with respect to the parameter **w**. By calling **loss.backward()**, PyTorch automatically calculates the gradients using the chain rule and stores them in the **grad** attribute of **w**.

The computed gradient (**w.grad**) represents the derivative of the loss function with respect to **w**, indicating the direction and magnitude of the steepest descent.

4. Weight Update and Optimization Step

After computing the gradients, the code proceeds to update the weights using the gradient descent optimization algorithm. The learning rate (**0.01** in this case) determines the step size for the weight update.

To ensure that the weight update operation does not create a computational graph for gradient computation, it is wrapped in a **torch.no_grad()** block. This operation modifies the value of **w** directly.

5. Zeroing Gradients

Before proceeding to the next optimization step, it is crucial to zero out the gradients of the optimized parameter (**w**) using **w.grad.zero_()**. This step prevents the gradients from accumulating across multiple optimization steps and ensures accurate gradient computation in subsequent passes.

6. Iterative Optimization

The optimization process typically involves multiple iterations, where forward and backward passes, weight updates, and gradient zeroing are performed iteratively.

The code provided showcases a single optimization step. To continue optimizing, the same process can be repeated by performing the forward and backward passes, updating the weights, and zeroing the gradients in a loop.

7. Conclusion

This technical report demonstrated the implementation of gradient descent optimization in PyTorch using the example of a single parameter (**w**). By computing gradients, updating weights, and performing multiple optimization steps, PyTorch enables the efficient training of machine learning models. Understanding these optimization techniques is essential for effectively utilizing PyTorch's capabilities in model parameter tuning.

Gradient Descent

Manually

1. Introduction

Linear regression is a widely used technique for modeling the relationship between variables. This technical report focuses on performing linear regression manually, without using any automatic differentiation libraries. The code demonstrates the steps involved in initializing the model, defining the forward pass, calculating the loss, updating the weights, and training the model.

2. Problem Description

The problem at hand is to find the relationship between the input variable **X** and the output variable **Y**, where **Y** is twice the value of **X**. The objective is to learn the weight parameter **w** in the linear equation **f = w * X** using manual computation of gradients.

3. Implementation Steps

The code begins by importing the necessary libraries, including NumPy, which provides powerful numerical computing capabilities.

4. Model Definition

The model's forward pass is defined as a simple linear equation **forward(x) = w * x**. In this case, the weight parameter **w** is initialized as zero.

5. Loss Function

The loss function used for linear regression is the Mean Squared Error (MSE), defined as **loss(y, y_pred) = ((y_pred - y)^2).mean()**. This function measures the average squared difference between the predicted values **y_pred** and the true values **y**.

6. Gradient Calculation

To update the weights during training, the gradients of the loss function with respect to the weight parameter **w** need to be computed. The gradient is manually calculated using the formula **dJ/dw = 1/N * 2x(w*x - y)**, where **J** represents the loss function, **N** is the number of data points, **x** is the input variable, **w** is the weight parameter, and **y** is the true output.

7. Training Loop

The training loop consists of multiple iterations (epochs). Within each epoch, the forward pass is performed to obtain the predicted values **y_pred**. The loss is calculated using the MSE function.

Gradients are computed manually using the gradient formula, and the weights are updated by subtracting the product of the learning rate and the gradients from the current weights.

During training, the weights **w** and the loss **l** are printed at every 2nd epoch to track the optimization progress.

8. Results

The code prints the prediction of the model before training for an input value of **5** using the initial weight value of **0.0**. After training, the code prints the prediction for the same input value using the learned weights.

9. Conclusion

This technical report demonstrated linear regression using manual computation of gradients. Although manual gradient computation can be cumbersome and prone to errors, it provides a clear understanding of the underlying mathematics and optimization process involved in linear regression.

However, manual computation becomes impractical for complex models with a large number of parameters. In such cases, leveraging automatic differentiation libraries like PyTorch, as shown in the previous example, simplifies the implementation and improves efficiency.

The code presented in this report serves as a learning resource for understanding the fundamentals of linear regression and the manual computation of gradients.

Auto

1. Introduction

Linear regression is a popular machine learning technique used for modeling the relationship between variables. This technical report demonstrates linear regression using PyTorch's autograd feature, which enables automatic computation of gradients. The code showcases the steps involved in setting up the model, defining the forward pass, calculating the loss, updating the weights, and training the model.

2. Problem Description

The problem at hand is to determine the relationship between an input variable **X** and an output variable **Y**, where **Y** is twice the value of **X**. The objective is to learn the weight parameter **w** in the linear equation **f = w * X** using autograd to compute the gradients automatically.

3. Implementation Steps

The code begins by importing the necessary libraries, including PyTorch, which provides automatic differentiation capabilities through autograd.

4. Model Definition

The model's forward pass is defined as a simple linear equation **forward(x) = w * x**, where **w** represents the weight parameter. In this case, the weight parameter **w** is initialized as zero and requires gradients to be computed during training.

5. Loss Function

The loss function used for linear regression is the Mean Squared Error (MSE), defined as **loss(y, y_pred) = ((y_pred - y)^2).mean()**. This function measures the average squared difference between the predicted values **y_pred** and the true values **y**.

6. Gradient Calculation

With PyTorch's autograd feature, the gradients of the loss function with respect to the weight parameter **w** are automatically computed during the backward pass. These gradients capture the sensitivity of the loss to changes in **w**, enabling efficient weight updates.

7. Training Loop

The training loop consists of multiple iterations (epochs). Within each epoch, the forward pass is performed to obtain the predicted values **y_pred**. The loss is calculated using the MSE function.

Gradients are computed automatically during the backward pass using the **backward()** method on the loss tensor. The weights are then updated using an optimization step that subtracts the product of the learning rate and the gradients from the current weights.

After each optimization step, the gradients are zeroed using **zero_()** to avoid accumulation in subsequent iterations.

The code prints the current weight value **w** and the loss **l** at every 10th epoch to monitor the training progress.

8. Results

The code prints the prediction of the model before training for an input value of **5** using the initial weight value of **0.0**. After training, the code prints the prediction for the same input value using the learned weights.

9. Conclusion

This technical report showcased linear regression using PyTorch's autograd feature, which automates the computation of gradients. By leveraging automatic differentiation, the code simplifies the implementation process and enables efficient training of the model.

The automatic gradient computation provided by autograd eliminates the need for manual calculation and reduces the risk of errors in gradient derivation. It also allows for seamless integration with complex models and optimization algorithms.

The code presented in this report serves as a valuable resource for understanding the application of autograd in linear regression. By utilizing PyTorch's autograd feature, researchers and practitioners can focus on model design and training without the burden of manual gradient computation.

Training Pipeline

No model

1. Introduction

Linear regression is a widely used machine learning technique for modeling the relationship between variables. This technical report demonstrates the implementation of linear regression using PyTorch, focusing on the design of the model, construction of the loss function, and selection of an optimizer. The code showcases the training loop, which includes forward pass computation, backward pass for gradient calculation, and weight updates.

2. Problem Description

The problem at hand involves predicting the output variable **Y** given an input variable **X**, where the relationship between **X** and **Y** is defined by the equation **Y = 2 * X**. The objective is to train a linear regression model to learn the weight parameter **w** in the equation **f = w * X** using a specified loss function and optimizer.

3. Implementation Steps

The code begins by importing the necessary libraries, including PyTorch and the **torch.nn** module, which provides various tools for neural network implementation.

4. Model Design

In this case, the linear regression model consists of a single weight parameter **w**. The model's forward pass is defined by the function **forward(x)**, which computes the output **y_pred** by multiplying the input **x** with the weight **w**. The weight **w** is initialized as zero and requires gradients to be computed during training.

5. Loss Function

The loss function used for linear regression is Mean Squared Error (MSE). In this code, the **nn.MSELoss()** function from PyTorch's **torch.nn** module is employed to calculate the MSE loss between the predicted values **y_predicted** and the true values **Y**.

6. Optimizer Selection

For weight updates during training, the Stochastic Gradient Descent (SGD) optimizer is employed. The optimizer is initialized with the weight parameter **w** as the parameter to optimize and a specified learning rate.

7. Training Loop

The training loop iterates for a specified number of epochs. Within each epoch, the forward pass is performed to obtain the predicted values **y_predicted**. The loss is calculated using the MSE loss function.

Gradients are computed automatically during the backward pass using the **backward()** method on the loss tensor. The optimizer's **step()** function is then called to update the weights based on the gradients. After updating the weights, the gradients are zeroed using the optimizer's **zero_grad()** function to prevent accumulation in subsequent iterations.

The code prints the current weight value **w** and the loss **l** at every 10th epoch to monitor the training progress.

8. Results

The code prints the prediction of the model before training for an input value of **5** using the initial weight value of **0.0**. After training, the code prints the prediction for the same input value using the learned weights.

9. Conclusion

This technical report presented the implementation of linear regression using PyTorch, focusing on the construction of the loss function and the selection of an optimizer. By utilizing PyTorch's built-in loss functions and optimizers, researchers and practitioners can easily define and optimize models for linear regression tasks.

The combination of the MSE loss function and the SGD optimizer provides a robust and efficient framework for training linear regression models. Researchers can extend this implementation to more complex models and larger datasets by incorporating additional layers and adapting the optimizer and loss function accordingly.

The code presented in this report serves as a valuable resource for understanding the application of loss functions and optimizers in linear regression. By leveraging PyTorch's capabilities, practitioners can efficiently train linear regression models and apply them to a wide range of real-world problems.

Model

1. Introduction

Linear regression is a widely used technique in machine learning for predicting a continuous output variable based on one or more input features. This technical report presents the implementation of a linear regression model using PyTorch. The report focuses on the design of the model, construction of the loss function, and selection of an optimizer. The code also includes a training loop for iteratively updating the model's parameters.

2. Problem Description

The problem addressed here involves predicting the output variable **Y** given the input variable **X**, where the relationship between **X** and **Y** follows the equation **Y = 2 * X**. The objective is to train a linear regression model to learn the weight parameter **w** in the equation **f = w * X**, using a specified loss function and optimizer.

3. Implementation Steps

The code begins by importing the necessary libraries, including PyTorch and the **torch.nn** module, which provides tools for implementing neural networks.

4. Model Design

In this case, the linear regression model is designed using the built-in **nn.Linear** class from PyTorch. The input size of the model is determined by the number of features in the input data, and the output size is also set to the number of features. The model implements the forward pass, which applies the linear transformation **f = w * X** to the input **X**.

An alternative approach is provided as commented code, defining a custom class **LinearRegression** that inherits from **nn.Module** and implements the forward pass in a separate function.

5. Loss Function and Optimizer

The loss function chosen for linear regression is Mean Squared Error (MSE). The **nn.MSELoss()** function from PyTorch is used to calculate the MSE loss between the predicted values **y_predicted** and the true values **Y**.

For weight updates during training, the Stochastic Gradient Descent (SGD) optimizer is selected. The optimizer is initialized with the model's parameters and a specified learning rate.

6. Training Loop

The training loop iterates for a specified number of epochs. Within each epoch, the forward pass is performed using the model to obtain the predicted values **y_predicted**. The loss is then calculated using the MSE loss function.

The gradients are automatically computed during the backward pass by calling the **backward()** method on the loss tensor. The optimizer's **step()** function is used to update the model's parameters based on the gradients. After updating the parameters, the gradients are zeroed using the optimizer's **zero_grad()** method to prevent accumulation in subsequent iterations.

The code also prints the current weight value **w** and the loss **l** at every 10th epoch to monitor the training progress.

7. Results

The code prints the prediction of the model before training for an input value of **5** using the initial weight value. After training, the code prints the prediction for the same input value using the learned weights.

8. Conclusion

This technical report presented the implementation of a linear regression model using PyTorch, focusing on the design of the model, construction of the loss function, and selection of an optimizer. By leveraging the capabilities of PyTorch, researchers and practitioners can easily define and optimize linear regression models.

The combination of the MSE loss function and the SGD optimizer provides an effective framework for training linear regression models. The code can be extended to handle more complex models or larger datasets by adapting the architecture and incorporating additional layers.

The presented code serves as a valuable resource for understanding the application of models, loss functions, and optimizers in linear regression. By utilizing PyTorch's capabilities, practitioners can efficiently train linear regression models and apply.

Linear Regression

1. Introduction

Linear regression is a fundamental supervised learning technique used for predicting continuous output values based on input features. This technical report presents the implementation of linear regression using PyTorch. The code demonstrates how to prepare the data, design a linear regression model, define the loss function and optimizer, and train the model to make accurate predictions.

2. Problem Description

The code addresses a synthetic regression problem, where the goal is to predict a continuous output value based on a single input feature. The dataset is generated using the **make_regression** function from the **sklearn** library. The generated data includes 100 samples with one feature and some added noise.

3. Implementation Steps

The code begins by importing the necessary libraries, including **torch**, **torch.nn**, **numpy**, **sklearn**, and **matplotlib.pyplot**.

4. Data Preparation

The **make_regression** function from **sklearn.datasets** is used to generate the synthetic dataset. The input features **X** and the target values **y** are stored as NumPy arrays. These arrays are then converted to PyTorch tensors using the **torch.from_numpy** function. The target tensor **y** is reshaped to have a shape of **(n_samples, 1)** to match the expected shape by the model.

5. Model Design

A linear regression model is designed using the **nn.Linear** class from PyTorch. The input size of the model is determined by the number of features in the input data, and the output size is set to 1 since we aim to predict a single continuous value. The model represents a linear relationship **f = wx + b**, where **w** is the weight and **b** is the bias term.

6. Loss Function and Optimizer

The mean squared error (MSE) loss function is selected for this linear regression task. The **nn.MSELoss** class from PyTorch is used to compute the MSE loss between the predicted values **y_predicted** and the true values **y**.

To optimize the model's parameters, the stochastic gradient descent (SGD) optimizer is employed. The **torch.optim.SGD** class is used, which takes the model parameters and a learning rate as input.

7. Training Loop

The training loop iterates for a specified number of epochs. Within each epoch, the forward pass is performed by calling the model with the input data **X** to obtain the predicted values **y_predicted**. The loss is then computed by comparing the predicted values with the true values using the MSE loss function.

The gradients are automatically computed during the backward pass by calling the **backward()** method on the loss tensor. The optimizer's **step()** function is used to update the model's parameters based on the gradients. After each update, the optimizer's **zero_grad()** method is called to reset the gradients to zero.

The code also prints the loss value every 10th epoch to monitor the training progress.

8. Results

After the training loop, the code uses the trained model to make predictions on the input data **X**. The predicted values are detached from the computational graph and converted to a NumPy array for visualization purposes. The code then plots the input data points as red dots and the predicted regression line as a blue line using **matplotlib.pyplot**.

9. Conclusion

This technical report presented the implementation of linear regression using PyTorch. By following the steps outlined in the code, researchers and practitioners can build and train linear regression models efficiently.

The provided code can be extended and modified to handle more complex regression tasks with multiple input features. Additionally, different loss functions, optimizers, and model architectures can be explored to improve the performance of the linear.

Logistic Regression

1. Introduction

Logistic regression is a widely used classification algorithm for predicting binary outcomes. This technical report presents the implementation of logistic regression using PyTorch. The code demonstrates how to prepare the data, design a logistic regression model, define the loss function and optimizer, train the model, and evaluate its performance.

2. Problem Description

The code tackles a binary classification problem using the Breast Cancer Wisconsin (Diagnostic) dataset from the **sklearn.datasets** module. The dataset contains features extracted from breast mass images and their corresponding diagnosis labels. The goal is to predict whether a given breast mass is benign (0) or malignant (1) based on the provided features.

3. Implementation Steps

The code begins by importing the necessary libraries, including **torch**, **torch.nn**, **numpy**, **sklearn.datasets**, **sklearn.preprocessing**, and **sklearn.model_selection**.

4. Data Preparation

The breast cancer dataset is loaded using **datasets.load_breast_cancer()** from **sklearn.datasets**. The features **X** and labels **y** are obtained from the dataset. The data is then split into training and test sets using **train_test_split()** from **sklearn.model_selection**. Additionally, the features are standardized using **StandardScaler** from **sklearn.preprocessing**.

The data is converted to PyTorch tensors using **torch.from_numpy()**. The labels are reshaped to have a shape of **(n_samples, 1)** to match the expected shape by the model.

5. Model Design

A logistic regression model is designed by creating a custom class **Model** that inherits from **nn.Module**. The model consists of a linear layer followed by the sigmoid activation function. The

linear layer maps the input features to a single output, representing the log-odds of the positive class (malignant). The sigmoid activation function squashes the log-odds to a probability between 0 and 1.

6. Loss Function and Optimizer

The binary cross-entropy loss (BCELoss) function is chosen for logistic regression. The **nn.BCELoss** class from PyTorch is used to compute the loss between the predicted probabilities **y_pred** and the true labels **y_train**.

For optimization, the stochastic gradient descent (SGD) optimizer is utilized. The **torch.optim.SGD** class is used, which takes the model parameters and a learning rate as input.

7. Training Loop

The training loop iterates for a specified number of epochs. Within each epoch, the forward pass is performed by calling the model with the training data **X_train** to obtain the predicted probabilities **y_pred**. The loss is then computed by comparing the predicted probabilities with the true labels using the BCE loss function.

The gradients are automatically computed during the backward pass by calling the **backward()** method on the loss tensor. The optimizer's **step()** function is used to update the model's parameters based on the gradients. After each update, the optimizer's **zero_grad()** method is called to reset the gradients to zero.

The code also prints the loss value every 10th epoch to monitor the training progress.

8. Evaluation

After training, the code uses the trained model to make predictions on the test data **X_test**. The predictions are rounded to the nearest integer using the **round()** function. The accuracy of the model is calculated by comparing the predicted labels with the true labels.

9. Conclusion

This technical report presented the implementation of logistic regression using PyTorch. By following the steps outlined in the code, researchers and practitioners can build and train logistic regression models effectively.

The provided code can be extended and modified to handle multi-class classification problems by adjusting the model architecture and the loss function accordingly. Additionally, different optimization algorithms and regularization techniques can be explored to improve the model's performance on various datasets.

Dataset and Dataloader

1. Introduction

The purpose of this technical report is to demonstrate the usage of Dataset and DataLoader in PyTorch. Dataset and DataLoader are important components in PyTorch for efficiently handling large datasets and performing batch computations during training.

2. Dataset

In this code snippet, a custom dataset called WineDataset is implemented by inheriting the Dataset class. The WineDataset class reads data from a CSV file and initializes the dataset by loading the

features and labels into tensors. The dataset provides support for indexing, allowing users to access individual samples by implementing the **__getitem__** method. The number of samples in the dataset is obtained using the **__len__** method.

3. DataLoader

To load the dataset in batches, the DataLoader class is utilized. The WineDataset is passed as the dataset argument to the DataLoader. Additional parameters such as batch_size, shuffle, and num_workers can be specified. The batch_size determines the number of samples used in each forward/backward pass, while shuffle randomizes the order of samples during training. The num_workers parameter controls the number of subprocesses to use for data loading, enabling faster loading with multiple processes.

4. Training Loop

A dummy training loop is provided to illustrate the usage of the DataLoader. The loop iterates over the DataLoader, allowing for easy iteration through the dataset in batches. For each batch, the inputs and labels are unpacked and can be used for training or other computations. The loop also demonstrates the calculation of the total number of iterations based on the batch size and dataset size.

5. Famous Datasets in torchvision.datasets

The code snippet also showcases the availability of famous datasets in the **torchvision.datasets** module. For example, the MNIST dataset is loaded using the **torchvision.datasets.MNIST** class. The dataset is downloaded, transformed into tensors using **torchvision.transforms.ToTensor()**, and then loaded into a DataLoader.

6. Conclusion

This technical report provided an overview of Dataset and DataLoader in PyTorch. By utilizing these components, researchers and practitioners can efficiently handle large datasets, perform batch computations during training, and easily access individual samples for analysis and model development. The code snippet serves as a useful reference for implementing custom datasets and loading data in batches using DataLoader.

Dataset Transforms

1. Introduction

The purpose of this technical report is to showcase the usage of dataset transforms in PyTorch. Dataset transforms are operations that can be applied to data during the creation of a dataset. These transforms enable preprocessing, data augmentation, and conversion of data to a suitable format for training or analysis.

2. Dataset Transforms

In this code snippet, the WineDataset class is extended to include the option of applying transforms to the data. Transforms can be specified during the initialization of the dataset by passing a transform object as an argument. The transform object must implement the **__call__** method, which applies the desired transformations to the sample.

3. Custom Transforms

Custom transforms can be created by defining a class that implements the **__call__** method. Two custom transforms are showcased in the code snippet:

- **ToTensor**: Converts the input and target ndarrays to PyTorch tensors using **torch.from_numpy()**. This transform allows seamless integration of the dataset with PyTorch's tensor-based operations.
- **MulTransform**: Multiplies the input data by a given factor. This transform showcases the ability to perform custom operations on the data.

4. Transform Composition

Transforms can be composed using the **torchvision.transforms.Compose** class. By creating a composition of transforms, multiple operations can be applied to the data in a sequential manner. In the code snippet, a composition is created using the **ToTensor** transform followed by the **MulTransform** transform with a factor of 4. This composition allows for both conversion to tensors and multiplication of the input data.

5. Usage and Results

The code snippet demonstrates the usage of dataset transforms by applying different transforms to the WineDataset. The data is initially loaded without any transforms, and the types of the features and labels are printed. Then, the dataset is created with the **ToTensor** transform, and the types of the transformed features and labels are printed. Finally, the dataset is created with the composition of **ToTensor** and **MulTransform**, and the types and values of the transformed features and labels are printed.

6. Conclusion

This technical report provided an overview of dataset transforms in PyTorch. By using transforms, researchers and practitioners can preprocess data, perform data augmentation, and convert data to suitable formats for training or analysis. The code snippet and the custom transforms showcased the flexibility and extensibility of dataset transforms in PyTorch.

Softmax and Crossentropy

1. Introduction

This technical report focuses on two fundamental concepts in deep learning: softmax and cross-entropy. Softmax is a mathematical function used to convert a vector of real numbers into a probability distribution. Cross-entropy, on the other hand, is a loss function that measures the dissimilarity between predicted and target probability distributions. Both softmax and cross-entropy play crucial roles in training classification models.

2. Softmax

Softmax is a function that takes a vector of real numbers as input and outputs a probability distribution over the classes. In the provided code snippet, both a numpy implementation and the PyTorch implementation of softmax are showcased. The softmax function exponentiates each element of the input vector and normalizes it by dividing it by the sum of all exponentiated elements. This operation ensures that the output probabilities range between 0 and 1 and sum up to 1. Softmax is commonly used in the output layer of classification models to obtain class probabilities.

3. Cross-Entropy

Cross-entropy is a loss function used to measure the dissimilarity between predicted and target probability distributions. It is widely used in classification tasks. The code snippet provides a numpy implementation of cross-entropy. It calculates the loss by taking the negative sum of the element-wise multiplication between the target probability distribution (one-hot encoded) and the logarithm of the predicted probability distribution. The PyTorch implementation of cross-entropy is also showcased using the **nn.CrossEntropyLoss** class. The PyTorch implementation combines the softmax operation and the negative log-likelihood loss into a single function.

4. Usage and Results

The code snippet demonstrates the usage of softmax and cross-entropy in both numpy and PyTorch. It includes examples of binary classification and multiclass classification scenarios. For binary classification, a neural network with a sigmoid activation function is used, and the binary cross-entropy loss (**nn.BCELoss**) is employed. For multiclass classification, a neural network with a ReLU activation function is used, and the cross-entropy loss (**nn.CrossEntropyLoss**) is utilized.

5. Conclusion

Softmax and cross-entropy are essential components of deep learning models, particularly in classification tasks. Softmax enables the conversion of raw output scores into interpretable probabilities. Cross-entropy provides a way to measure the dissimilarity between predicted and target probability distributions, allowing the model to optimize its parameters through backpropagation. This technical report provided an overview of softmax and cross-entropy, along with their implementations in both numpy and PyTorch.

Activation Functions

Base

1. Introduction

This technical report focuses on activation functions in deep learning, which play a crucial role in introducing non-linearity to neural networks. Activation functions are applied to the output of each neuron, enabling neural networks to learn complex patterns and make non-linear transformations. This report discusses various activation functions and their implementations in PyTorch.

2. Activation Functions

The code snippet demonstrates the usage of several popular activation functions:

2.1 Softmax Softmax is a function that converts a vector of real numbers into a probability distribution. It exponentiates each element of the input vector and normalizes it by dividing it by the sum of all exponentiated elements. Softmax is commonly used in the output layer of multi-class classification models to obtain class probabilities.

2.2 Sigmoid Sigmoid is a function that maps the input to a value between 0 and 1. It is often used in binary classification problems to produce a probability-like output. The sigmoid function has a smooth, S-shaped curve.

2.3 Tanh Tanh, or hyperbolic tangent, is a function similar to the sigmoid function but maps the input to a value between -1 and 1. Like the sigmoid, it is also used to introduce non-linearity to neural networks.

2.4 ReLU ReLU, or Rectified Linear Unit, is a widely used activation function. It returns the input directly if it is positive, and zero otherwise. ReLU is known for its simplicity and ability to mitigate the vanishing gradient problem.

2.5 Leaky ReLU Leaky ReLU is a variant of the ReLU function that allows small negative values when the input is negative. It introduces a small slope for negative values, preventing the "dying ReLU" problem where neurons can get stuck in a state of zero activation.

3. Implementation

The code snippet demonstrates the implementation of activation functions in PyTorch using both functional and module-based approaches. In the functional approach, the activation functions are directly applied to the output tensors in the forward pass. In the module-based approach, activation functions are initialized as modules within the neural network class and applied in the forward method.

4. Conclusion

Activation functions are essential components in deep learning models as they introduce non-linearity and enable neural networks to learn complex patterns. This technical report provided an overview of several commonly used activation functions, including softmax, sigmoid, tanh, ReLU, and leaky ReLU. The implementation of these activation functions in PyTorch was also showcased. Understanding activation functions and their properties is crucial for effectively designing and training neural networks.

Plot Activations

1. Introduction

This technical report focuses on plotting various activation functions commonly used in deep learning. Activation functions introduce non-linearity to neural networks, enabling them to learn complex patterns. The code snippet provided demonstrates the plotting of different activation functions using NumPy and Matplotlib.

2. Activation Functions and Plots

The code snippet plots the following activation functions:

2.1 Sigmoid The sigmoid function is a smooth, S-shaped curve that maps the input to a value between 0 and 1. It is widely used in binary classification problems to produce a probability-like output.

2.2 TanH The TanH (hyperbolic tangent) function is another S-shaped curve that maps the input to a value between -1 and 1. TanH is often used in neural networks to introduce non-linearity and has outputs centered around zero.

2.3 ReLU ReLU (Rectified Linear Unit) is a popular activation function that returns the input directly if it is positive, and zero otherwise. ReLU helps alleviate the vanishing gradient problem and has become a standard choice for many neural network architectures.

2.4 Leaky ReLU Leaky ReLU is a variant of ReLU that allows small negative values when the input is negative. It introduces a small slope for negative values, preventing neurons from becoming "dead." Leaky ReLU can be beneficial when dealing with sparse or negative inputs.

2.5 Binary Step The binary step function is a simple activation function that outputs either 0 or 1 based on the input's sign. It is not commonly used in deep learning models due to its lack of differentiability, but it is included here for reference.

3. Plots and Analysis

The code snippet generates plots for each activation function, showcasing their respective shapes and properties. Each plot includes labeled axes, a title, and appropriate tick marks for clarity. The x-axis represents the input values, while the y-axis represents the output values.

4. Conclusion

Activation functions are essential components in deep learning models, enabling the introduction of non-linearity and improving the models' learning capabilities. This technical report demonstrated the plotting of several popular activation functions, including sigmoid, TanH, ReLU, Leaky ReLU, and the binary step function. By visualizing these activation functions, researchers and practitioners can gain insights into their characteristics and make informed decisions when designing neural network architectures.

Feed Forward Net

1. Introduction

This technical report focuses on implementing and training a feedforward neural network for the MNIST classification task. The MNIST dataset consists of grayscale images of handwritten digits (0-9) and serves as a benchmark dataset for evaluating machine learning models' performance. The code snippet provided demonstrates the complete pipeline, including data loading, model definition, training, and evaluation.

2. Model Architecture

The feedforward neural network used in this code snippet consists of one hidden layer with ReLU activation and a final output layer. The input layer size is 784, corresponding to the flattened 28x28 grayscale images in the MNIST dataset. The hidden layer size is set to 500, and the number of output classes is 10, representing the digits 0-9. The model's architecture is defined in the **NeuralNet** class, utilizing the **nn.Module** base class provided by PyTorch.

3. Data Loading and Preprocessing

The code snippet utilizes the torchvision library to download and load the MNIST dataset. The dataset is split into training and testing sets. Data loaders are employed to efficiently load the data in batches during training and evaluation. The images are transformed to tensors and normalized using the **transforms.ToTensor()** function.

4. Training the Model

The model is trained using the Adam optimizer and cross-entropy loss function. The training loop iterates over the training dataset for the specified number of epochs. Each batch of images and labels is passed through the network, and the loss is calculated. The optimizer's gradients are updated using backpropagation, and the process is repeated until all batches are processed. The loss is printed periodically to monitor the training progress.

5. Evaluating the Model

After training, the model is evaluated on the test dataset. The evaluation is performed in a no-gradient mode using **torch.no_grad()** to save memory. The accuracy is calculated by comparing the predicted labels to the true labels, and the result is printed.

6. Conclusion

The code snippet provides a complete implementation of a feedforward neural network for the MNIST classification task. By following this pipeline, researchers and practitioners can train and evaluate their own models on the MNIST dataset or similar image classification tasks. The provided code can serve as a foundation for exploring more advanced neural network architectures and optimization techniques to achieve higher accuracy on the MNIST dataset.

CNN

1. Introduction

This technical report presents the implementation and training of a Convolutional Neural Network (CNN) for the CIFAR-10 image classification task. The CIFAR-10 dataset consists of 60,000 32x32 color images categorized into 10 classes. The provided code snippet demonstrates the complete pipeline, including data loading, model architecture, training, and evaluation.

2. Model Architecture

The CNN used in this code snippet consists of two convolutional layers, each followed by a max-pooling layer, and three fully connected layers. The convolutional layers are responsible for learning spatial hierarchies and capturing relevant features from the input images. The max-pooling layers reduce the spatial dimensions, aiding in translation invariance. The fully connected layers act as classifiers, mapping the learned features to the 10 output classes. The architecture is defined in the **ConvNet** class, derived from the **nn.Module** base class provided by PyTorch.

3. Data Loading and Preprocessing

The code snippet utilizes the torchvision library to download and load the CIFAR-10 dataset. The images are transformed to tensors and normalized to have a range of [-1, 1] using the **transforms.Compose** function. The dataset is split into training and testing sets, and data loaders are employed to efficiently load the data in batches during training and evaluation.

4. Training the Model

The model is trained using stochastic gradient descent (SGD) optimization and the cross-entropy loss function. The training loop iterates over the training dataset for the specified number of epochs. Each batch of images and labels is passed through the network, and the loss is calculated. The optimizer's gradients are updated using backpropagation, and the process is repeated until all batches are processed. The loss is printed periodically to monitor the training progress.

5. Evaluating the Model

After training, the model is evaluated on the test dataset. The evaluation is performed in a no-gradient mode using **torch.no_grad()** to save memory. The accuracy is calculated by comparing the predicted labels to the true labels. Additionally, the accuracy for each individual class is calculated. The overall accuracy and per-class accuracy are printed.

6. Conclusion

The provided code snippet demonstrates the implementation and training of a CNN for the CIFAR-10 classification task. By following this pipeline, researchers and practitioners can train and evaluate their own CNN models on the CIFAR-10 dataset or similar image classification tasks. The code can be used as a starting point for further exploration, such as experimenting with different architectures, optimization algorithms, or data augmentation techniques, to improve the model's performance on the CIFAR-10 dataset.

Transfer Learning

1. Introduction

This technical report presents the implementation and training of a transfer learning approach for image classification using a pretrained ResNet-18 model. Transfer learning allows us to leverage the knowledge learned from a large dataset (e.g., ImageNet) and apply it to a new dataset with limited labeled examples. The provided code snippet demonstrates two transfer learning scenarios: finetuning the entire network and using the convolutional base as a fixed feature extractor.

2. Data Loading and Preprocessing

The code snippet uses the **torchvision.datasets.ImageFolder** class to load the image dataset from the **hymenoptera_data** directory. The dataset is divided into training and validation sets. Data augmentation techniques, such as random resized crop and random horizontal flip, are applied during training to improve generalization. The images are normalized using the mean and standard deviation values provided.

3. Visualizing the Data

The **imshow** function is provided to visualize a batch of training data. It shows a grid of images along with their corresponding class labels. This visualization helps in understanding the dataset and verifying the correctness of the data loading and preprocessing steps.

4. Training the Model

The **train_model** function implements the training loop for the transfer learning scenarios. The function takes a model, criterion (loss function), optimizer, scheduler, and the number of epochs as inputs. It trains the model using the training dataset and evaluates it on the validation dataset at each epoch. The training loop performs forward and backward passes, updates the model's weights, and prints the loss and accuracy for each phase (train or val). The best model based on validation accuracy is saved and returned.

5. Transfer Learning: Finetuning the ConvNet

In the first transfer learning scenario, the entire ResNet-18 model is loaded with pretrained weights. The final fully connected layer is reset and replaced with a new linear layer that maps the learned features to the specific classification task. Only the parameters of the new layer are optimized during training, while the rest of the network is frozen. The model is trained using stochastic gradient descent (SGD) optimization and the provided learning rate scheduler.

6. Transfer Learning: Using ConvNet as Fixed Feature Extractor

In the second transfer learning scenario, the pretrained ResNet-18 model is loaded, and all the parameters are frozen except for the final layer. This approach treats the convolutional base as a fixed feature extractor. The gradients are not computed for the frozen layers during the backward

pass, and only the parameters of the final layer are optimized. This technique is useful when the new dataset is small and similar to the original dataset on which the model was pretrained.

7. Conclusion

The provided code snippet demonstrates the application of transfer learning for image classification using a pretrained ResNet-18 model. By following this pipeline, researchers and practitioners can effectively utilize transfer learning to improve the performance of their models on new image datasets with limited labeled examples. The code can be extended and customized for other transfer learning scenarios and different pretrained models available in the torchvision library.

Tensorboard

1. Introduction

This technical report presents the implementation of TensorBoard integration in a simple MNIST classification task using a fully connected neural network. TensorBoard is a powerful visualization tool provided by TensorFlow that allows users to track and visualize various aspects of their machine learning models. In this report, we demonstrate the use of TensorBoard for visualizing images, model graphs, training loss, accuracy, and precision-recall curves.

2. Loading and Preprocessing Data

The code snippet loads the MNIST dataset using the torchvision library. It separates the dataset into training and testing sets and creates data loaders for efficient batch processing. Additionally, it displays a subset of example images from the test set using Matplotlib.

3. TensorBoard Integration: Visualizing Images

To visualize the MNIST images in TensorBoard, the code snippet uses the **torchvision.utils.make_grid** function to create a grid of example images. It then adds the image grid to the TensorBoard writer using **writer.add_image**. This allows us to visualize a grid of images in TensorBoard, providing a visual representation of the dataset.

4. Model Definition

The code defines a fully connected neural network model using the **NeuralNet** class. It consists of one hidden layer with ReLU activation and an output layer. The **forward** method defines the forward pass of the model.

5. TensorBoard Integration: Visualizing Model Graph

To visualize the model graph in TensorBoard, the code snippet uses **writer.add_graph** to add the model graph to the TensorBoard writer. This visual representation of the model helps in understanding the network architecture and the flow of data through the layers.

6. Model Training and Logging

The code trains the model using the MNIST training dataset. It performs the forward and backward passes, updates the model's weights, and calculates the training loss and accuracy. After every 100 steps, it prints the current loss and adds the training loss and accuracy values to TensorBoard using **writer.add_scalar**. This allows us to visualize the training progress and monitor the loss and accuracy trends over time.

7. TensorBoard Integration: Logging Metrics

To log additional metrics, the code calculates the precision-recall curves for each class using the model's outputs and adds them to TensorBoard using **writer.add_pr_curve**. This visualization helps in assessing the model's performance for each class in terms of precision and recall.

8. Model Evaluation and Accuracy Calculation

After training, the code evaluates the trained model on the MNIST test dataset. It calculates the accuracy of the model on the test set and prints the result. This evaluation helps in understanding the model's performance on unseen data.

9. Conclusion

The provided code snippet demonstrates the integration of TensorBoard in a simple MNIST classification task. By incorporating TensorBoard, researchers and practitioners can effectively track and visualize various aspects of their models, including images, model graphs, training loss, accuracy, and precision-recall curves. TensorBoard provides valuable insights into the model's behavior and performance, aiding in model analysis and decision-making. The code can be extended and customized to integrate additional TensorBoard functionalities and adapt to other machine learning tasks and datasets.

Save & Load Models

1. Introduction

This technical report presents different methods for saving and loading models in PyTorch. Saving and loading models is crucial for preserving trained models, transferring them across different devices, and resuming training from a saved state. This report outlines two common approaches: saving the entire model and saving only the model's state_dict.

2. Saving the Entire Model

The code snippet demonstrates two ways to save the entire model. The first approach uses the **torch.save** function to save the entire model object directly to a file. This method is straightforward but may result in compatibility issues if the model architecture changes between saving and loading. The second approach, recommended by PyTorch, saves only the model's **state_dict** using **torch.save**. This method isolates the model's learnable parameters and ensures compatibility when loading the state_dict into a model with the same architecture.

3. Loading the Entire Model

To load the entire model, the code snippet uses **torch.load** to load the saved model file. The loaded model can then be used for inference or further training. It is important to call **model.eval()** after loading to set the model in evaluation mode, especially if the model contains layers like dropout or batch normalization that behave differently during training and inference.

4. Saving and Loading Only the state_dict

The code snippet demonstrates saving and loading only the **state_dict** of the model, which contains all the learnable parameters. This method is more flexible because it allows for easy parameter transfer between different model architectures or instances. By saving and loading only the **state_dict**, the model architecture can be modified or redefined without affecting the saved parameters.

5. Saving and Loading Model Checkpoints

The code snippet illustrates how to save and load model checkpoints, which include both the **state_dict** and optimizer state. Saving checkpoints allows for resuming training from a specific epoch or iteration. The checkpoint dictionary contains the model's **state_dict**, the optimizer's **state_dict**, and additional information such as the current epoch or iteration. Loading a checkpoint involves loading the **state_dict** and optimizer state into the model and optimizer instances, respectively.

6. Saving and Loading Models on Different Devices

The code snippet provides examples of saving and loading models on different devices, such as between CPU and GPU. It demonstrates the necessary steps to ensure compatibility when transferring models between devices. The **to()** function is used to move the model to the desired device, and the **map_location** argument is used during loading to specify the target device.

7. Conclusion

The provided code snippet highlights various methods for saving and loading models in PyTorch. Choosing the appropriate method depends on the use case and requirements. Saving and loading the entire model object is convenient but may have compatibility issues, while saving and loading only the **state_dict** provides more flexibility. Model checkpoints are useful for resuming training, and transferring models between devices requires careful consideration of device placement and conversion. Researchers and practitioners can utilize these techniques to save, load, and transfer models effectively in their PyTorch projects.