I'm not a robot

Introduction:Neural networks have revolutionized the field of artificial intelligence, enabling powerful applications such as image recognition, natural language processing, and speech synthesis. PyTorch, a popular deep-learning framework, provides an intuitive and flexible platform for building and training neural networks. In this tutorial, we will walk you through the process of creating a basic neural network using PyTorch, explaining each step along the way. By the end, you will have a solid understanding of how neural networks work and be ready to explore more complex architectures.Prerequisites:To follow this tutorial, you should have a basic understanding of Python programming and some familiarity with linear algebra and calculus concepts. Additionally, make sure you have PyTorch installed on your machine. You can install it via pip by running the command: pip install torch.Step 1:Importing the necessary libraries Lets start by importing the required libraries. PyTorch is our primary dependency, but we will also import the torchvision library for convenient access to popular datasets and models.import torchimport torch.nn as nnimport torch.optim as optimimport torchvision.datasets as datasetsimport torchvision.transforms as transformsStep 2:Loading and Preprocessing the Data For this tutorial, we will use the MNIST dataset, which consists of grayscale images of handwritten digits (09). Well load the dataset and apply some basic preprocessing, including normalization and resizing.# Define transformation pipelinetransform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])# Load the MNIST training settrain_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)# Create a data loadertrain_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)Step 3:Building the Neural Network Architecture Next, we define the architecture of our neural network. For this tutorial, we will create a simple feedforward network with two fully connected layers and a ReLU activation function.class NeuralNetwork(nn.Module): def __init__(self): super(NeuralNetwork, self).__init__() self.fc1 = nn.Linear(784, 128) self.fc2 = nn.Linear(128, 10) self.relu = nn.ReLU() def forward(self, x): x = x.view(x.size(0), -1) x = self.relu(self.fc1(x)) x = self.fc2(x) return x# Create an instance of the NeuralNetwork classmodel = NeuralNetwork()Step 4:Defining the Loss Function and Optimizer To train our neural network, we need to specify a loss function and an optimizer. For this classification task, we will use the cross-entropy loss and the stochastic gradient descent (SGD) optimizer.criterion = nn.CrossEntropyLoss()optimizer = optim.SGD(model.parameters(), lr=0.01)Step 5:Training the Neural Network Now, we can proceed to train our neural network. We will iterate over the training dataset for a specified number of epochs, performing forward and backward passes to update the models weights.num_epochs = 10for epoch in range(num_epochs): for batch_idx, (data, targets) in enumerate(train_loader): optimizer.zero_grad() # Forward pass outputs = model(data) loss = criterion(outputs, targets) # Backward pass loss.backward() optimizer.step() if batch_idx % 100 == 0: print(f"Epoch {epoch+1}/{num_epochs}, Step {batch_idx}/{len(train_loader)}, Loss: {loss.item():.4f}")Step 6:Evaluating the Model After training, its essential to evaluate the performance of our model on unseen data. Here, we will use the MNIST test set to calculate the accuracy of our model.test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)def test_model(): model.eval() correct = 0 total = 0 with torch.no_grad(): for data, targets in test_loader: outputs = model(data) _, predicted = torch.max(outputs.data, 1) total += targets.size(0) correct += (predicted == targets).sum().item() print(f"Test Accuracy: {(100 * correct / total):.2f}%")# Call the test_model function to evaluate the trained modeltest_model()Conclusion:Congratulations! You have successfully built and trained a basic neural network using PyTorch. We covered the essential steps, from loading and preprocessing data to defining the network architecture, training, and evaluating the model. Neural networks offer endless possibilities, and PyTorch makes it accessible to implement and experiment with various architectures. Keep exploring and refining your models to tackle exciting challenges in the field of deep learning!Remember, this tutorial only scratched the surface of what you can achieve with neural networks and PyTorch. As you delve deeper into the world of deep learning, youll encounter more complex architectures, advanced optimization techniques, and powerful applications. Happy coding!Have you ever wondered what really goes into building a neural network? At first glance, it might seem like magic, but its actually a series of simple, logical steps that anyone can learn. In this tutorial, were going to demystify the process and show you how to build a neural network from scratch using PyTorch.Understanding the fundamentals of neural networks is more than just a cool skill its the foundation for mastering machine learning. Whether youre a beginner or looking to deepen your understanding, building a model from the ground up is the perfect way to solidify your knowledge.Why PyTorch? Its one of the most beginner-friendly yet powerful deep learning frameworks out there, trusted by researchers and industry professionals alike. With PyTorch, you get the flexibility to dive into the nuts and bolts of neural networks while enjoying its intuitive API.By the end of this blog, youll have built your own neural network step-by-step, learned how to train it, and gained the confidence to experiment further. Lets roll up our sleeves and get started!Setting Up the EnvironmentBefore we dive into building our neural network, lets set up the perfect environment for coding: Google Colab. Why Colab? Its free, runs on the cloud, and comes pre-installed with PyTorch and GPU support, making it ideal for machine learning projects without the need for local installation hassles.Step 1: Access Google ColabOpen your browser and go to Google Colab.Sign in with your Google account (if you arent already signed in).Click on New Notebook to start your project.Step 2: Verify GPU AvailabilityTo speed up training, well use Colabs GPU support:Go to the Runtime menu.Select Change runtime type.Set Hardware accelerator to GPU and save.You can verify the GPU setup by running this code in a Colab cell:import torchprint("GPU Available:", torch.cuda.is_available())print("GPU Name:", torch.cuda.get_device_name(0) if torch.cuda.is_available() else "None")If you see a GPU name, youre good to go!Step 3: Import PyTorchThe best part? PyTorch is pre-installed on Colab. Simply import it:import torchimport torch.nn as nnimport torch.optim as optimNo extra installation required!Step 4: Organizing Your NotebookTo keep your work organized:Create sections for Data Preparation, Model Building, Training, and Evaluation.Use Colabs text cells to add notes and explanations for each section.With Colab, youre ready to focus entirely on building and training your neural network without worrying about setup complications. Lets move on to understanding the basics of neural networks!Understanding the BasicsBefore building our neural network, lets break down its core concepts. Think of a neural network as a recipe in the kitchen its all about tweaking the ingredients (inputs), adjusting the method (layers), and tasting the dish (output) until its just right. Heres how it all fits together:1. Input LayerThe input layer is where it all begins its like gathering your ingredients. Each ingredient is a feature from your dataset, such as a pixel value in an image or a numerical attribute in a spreadsheet. The input layer takes this raw data and passes it to the next step.2. Hidden LayersHidden layers are like mixing and combining ingredients in a recipe. They take the raw inputs, process them using mathematical operations, and pass the result forward. These layers add complexity to the network, allowing it to capture patterns and relationships in the data.Each hidden layer consists of:Neurons: Individual units that process input values and produce outputs.Weights and Biases: The secret sauce that determines how strongly each input influences the output. These are learned during training.3. Output LayerThe output layer is like tasting your dish it gives the final result. For example:In classification problems, it might predict a label (e.g., cat vs. dog).In regression problems, it might predict a number (e.g., house price).The number of neurons in the output layer corresponds to the number of target classes or outputs.4. Activation FunctionsThink of activation functions as the flavor enhancers in your recipe. They decide whether a neurons output should be passed forward or not. Without them, a neural network would just be a stack of linear equations.Common activation functions include:ReLU (Rectified Linear Unit): Passes positive values as-is, setting negative values to zero.Sigmoid: Squeezes outputs into a range between 0 and 1, useful for probabilities.Softmax: Converts outputs into probabilities for multi-class classification.5. Forward Propagation and Loss CalculationForward propagation is like following a recipe step-by-step:Inputs pass through the network layer by layer.At each layer, calculations (using weights, biases, and activation functions) transform the inputs.The output layer produces a prediction.The loss function evaluates how off the prediction is compared to the true value, like tasting your dish and deciding if its too salty or bland.Common loss functions:Mean Squared Error (MSE): For regression problems.Cross-Entropy Loss: For classification problems.6. Backpropagation and OptimizationBackpropagation is where the magic happens its like tweaking your recipe after a taste test:The network calculates the error (loss) and determines which parts of the recipe (weights and biases) need adjustment.Gradients (like feedback on taste) are computed to show how much each parameter contributed to the error.An optimizer (e.g., SGD or Adam) adjusts the weights and biases to reduce the error in the next iteration.This process repeats until the dish your neural network is perfectly cooked and ready to serve.By thinking of neural networks as recipes, you can see that theyre not so intimidating after all. With this foundation, youre ready to start building your own in PyTorch!Defining the ProblemTo make our neural network project more engaging and practical, lets tackle a real-life problem: Text Classification. Specifically, well classify news articles into categories like politics, sports, or technology using the AG News dataset a popular benchmark dataset for text classification tasks.Problem StatementWe aim to build a neural network that can classify news articles into one of four categories:WorldSportsBusinessScience/TechnologyStep 1: Know About the DatasetDataset: AG News (available in PyTorchs torchtext library)Description: Contains 120,000 training samples and 7,600 test samples of news headlines and short descriptions, along with their corresponding labels.Input: Text data (news headlines and descriptions)Output: A category label (1 to 4)Well use Hugging Faces datasets library to load and preprocess the AG News dataset.from datasets import load_datasetfrom transformers import AutoTokenizerimport torch# Load the AG News datasetdataset = load_dataset("ag_news")Step 4: Tokenise the Dataset# Load a tokenizer (e.g., BERT tokenizer)tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")# Tokenize the datasetdef tokenize_function(example): return tokenizer(example["text"], padding="max_length", truncation=True, max_length=128)tokenized_datasets = dataset.map(tokenize_function, batched=True)# Format the dataset for PyTorchtokenized_datasets = tokenized_datasets.remove_columns(["text"])tokenized_datasets = tokenized_datasets.rename_column("label", "labels")tokenized_datasets.set_format("torch")Load the Tokenizertokenizer = AutoTokenizer.from_pretrained("bert-base-uncased"): Loads a pre-trained tokenizer (BERT tokenizer here) to convert text into tokenized input that the model can process.Tokenize the DatasetThe tokenize_function uses the tokenizer to:Pad sequences to a fixed length (padding="max_length").Truncate long sequences (truncation=True).Limit sequence length to 128 tokens (max_length=128).dataset.map(tokenize_function, batched=True): Applies the tokenizer to the entire dataset in batches for efficiency.Remove Unused Columnstokenized_datasets.remove_columns(["text"]): Removes the original text column since we now have tokenized inputs.Rename Label Columntokenized_datasets.rename_column("label", "labels"): Renames the label column to match PyTorch conventions for compatibility.Format Dataset for PyTorchtokenized_datasets.set_format("torch"): Converts the dataset into PyTorch tensors, making it ready for DataLoader usage.Step 5: Create Data LoadersWell use PyTorchs DataLoader to prepare batches of data for training and evaluation.from torch.utils.data import DataLoader# Create DataLoaderstrain_dataloader = DataLoader(tokenized_datasets["train"], batch_size=32, shuffle=True)test_dataloader = DataLoader(tokenized_datasets["test"], batch_size=32)The DataLoader in PyTorch is a utility that simplifies the process of feeding data to your model in batches during training and evaluation.Batchingbatch_size=32: Splits the dataset into batches of 32 samples for efficient processing.Shufflingshuffle=True (for training): Randomizes the order of the data to reduce overfitting and improve generalization.Automatic BatchingCombines the tokenized input and labels into structured batches that the model can process.Train/Test Splittrain_dataloader: For training the model.test_dataloader: For evaluating the model on unseen data (no shuffling needed here).The DataLoader is essential for handling large datasets efficiently and is compatible with PyTorch training loops.Step 6: Define the Neural NetworkHeres a custom PyTorch neural network for text classification.import torch.nn as nnclass TextClassifier(nn.Module): def __init__(self, vocab_size, embed_dim, num_classes): super(TextClassifier, self).__init__() self.embedding = nn.Embedding(vocab_size, embed_dim) self.fc1 = nn.Linear(embed_dim, 128) # First hidden layer self.relu = nn.ReLU() # Activation function self.fc2 = nn.Linear(128, num_classes) # Output layer def forward(self, input_ids): # Embed the input tokens embedded = self.embedding(input_ids).mean(dim=1) # Mean pooling # Pass through the network x = self.fc1(embedded) x = self.relu(x) output = self.fc2(x) return output# Initialize the modelvocab_size = tokenizer.vocab_sizeembed_dim = 128num_classes = 4model = TextClassifier(vocab_size, embed_dim, num_classes)Heres a concise explanation of theTextClassifier:Inheriting from nn.ModuleThe TextClassifier class extends PyTorchs nn.Module, enabling it to act as a trainable neural network.Initialization (__init__)nn.Embedding(vocab_size, embed_dim): Creates an embedding layer to transform input token IDs into dense vectors of size embed_dim.nn.Linear(embed_dim, 128): First fully connected (dense) layer with 128 neurons.nn.ReLU(): Activation function to add non-linearity.nn.Linear(128, num_classes): Output layer mapping the 128 features to the number of classes (e.g., 4 for AG News).Forward Pass (forward)self.embedding(input_ids): Maps input token IDs to embeddings..mean(dim=1): Applies mean pooling to summarize the embeddings across tokens in a sequence.self.fc1(embedded): Passes the pooled embedding through the first dense layer.self.relu(x): Applies the ReLU activation function.self.fc2(x): Outputs the logits for the number of classes.Model Initializationvocab_size: Size of the tokenizer's vocabulary.embed_dim: Dimension of the word embeddings (set to 128).num_classes: Number of output classes (e.g., 4 for text classification).This model takes tokenized input IDs, converts them into embeddings, and predicts the class probabilities through a series of transformations. Its simple yet effective for basic text classification tasks.Step 7: Define the optimiser and loss functioncriterion = nn.CrossEntropyLoss()optimizer = torch.optim.Adam(model.parameters(), lr=5e-4)Loss Function: nn.CrossEntropyLoss()Purpose: Measures the difference between the predicted logits and the true class labels.Usage: Suitable for classification tasks where the output is a probability distribution across multiple classes.Input:Predictions (logits) from the model: Shape (batch_size, num_classes).True labels: Shape (batch_size), containing integer class indices.Output: A scalar loss value used to guide the models parameter updates.Optimizer: torch.optim.AdamPurpose: Updates the models parameters to minimize the loss function.Why Adam?Combines the benefits of momentum (smoother updates) and adaptive learning rates (faster convergence).Parameters:model.parameters(): Specifies the model's trainable parameters.lr=5e-4: Learning rate (controls the step size for parameter updates).Step 8: Define the training loopdevice = torch.device("cuda" if torch.cuda.is_available() else "cpu")model.to(device)num_epochs = 3for epoch in range(num_epochs): model.train() total_loss = 0 for batch in train_dataloader: # Move data to device input_ids = batch["input_ids"].to(device) labels = batch["labels"].to(device) # Forward pass outputs = model(input_ids) loss = criterion(outputs, labels) # Backward pass and optimization optimizer.zero_grad() loss.backward() optimizer.step() total_loss += loss.item() print(f"Epoch {epoch + 1}, Loss: {total_loss / len(train_dataloader)}")torch.device("cuda" if torch.cuda.is_available() else "cpu"): Automatically uses a GPU if available; otherwise, defaults to the CPU.model.to(device): Moves the model to the selected device for computation.Outer Loop (Epochs):num_epochs = 3: Runs the entire dataset through the model three times to learn patterns.model.train(): Sets the model to training mode, enabling operations like dropout (if present).Inner Loop (Batches):Iterates through the train_dataloader, processing one batch of data at a time.Data Handlingbatch["input_ids"].to(device) and batch["labels"].to(device): Moves the batch data (input IDs and labels) to the same device as the model (GPU or CPU).Forward Passoutputs = model(input_ids): Feeds the input IDs through the model to compute predictions.loss = criterion(outputs, labels): Calculates the classification error between predictions (outputs) and true labels (labels) using the cross-entropy loss.Backward Pass and Optimizationoptimizer.zero_grad(): Resets gradients from the previous iteration to prevent accumulation.loss.backward(): Computes the gradients of the loss with respect to the model parameters via backpropagation.optimizer.step(): Updates the model parameters using the computed gradients.Loss Trackingtotal_loss += loss.item(): Accumulates the total loss for the epoch to monitor training progress.print(f"Epoch {epoch + 1}, Loss: {total_loss / len(train_dataloader)}"): Logs the average loss per epoch to track improvement.Step 7: Evaluate the Modelmodel.eval()correct = 0total = 0with torch.no_grad(): for batch in test_dataloader: input_ids = batch["input_ids"].to(device) labels = batch["labels"].to(device) outputs = model(input_ids) predictions = torch.argmax(outputs, dim=1) correct += (predictions == labels).sum().item() total += labels.size(0)accuracy = correct / totalprint(f"Test Accuracy: {accuracy:.4f}")Evaluation Modemodel.eval(): Puts the model in evaluation mode, disabling dropout and training-specific behaviors. torch.no_grad(): Disables gradient computation to save memory and speed up evaluation.Loop Through Test DataIterates through the test_dataloader to process the test dataset in batches.Data Handlinginput_ids = batch["input_ids"].to(device) and labels = batch["labels"].to(device): Move the tokenized input IDs and ground truth labels to the same device (GPU or CPU).Forward Passoutputs = model(input_ids): Passes the input IDs through the model to compute logits (unnormalized scores for each class).Predictionspredictions = torch.argmax(outputs, dim=1): Finds the class with the highest score (logit) for each input, resulting in predicted class indices.Accuracy Calculation(predictions == labels).sum().item(): Counts the correct predictions in the current batch. total += labels.size(0): Tracks the total samples processed. accuracy = correct / total: Computes overall accuracy as the ratio of correct predictions to total samples.Outputprint(f"Test Accuracy: {accuracy:.4f}"): Displays the model's accuracy on the test dataset as a decimal with four places.This loop evaluates model performance by comparing predictions to true labels, calculating accuracy efficiently without computing gradients.Step 8: Run the model on a new example datadef classify_text(text): model.eval() inputs = tokenizer(text, return_tensors="pt", padding="max_length", truncation=True, max_length=128).to(device) with torch.no_grad(): outputs = model(inputs["input_ids"]) prediction = torch.argmax(outputs, dim=1).item() categories = ["World", "Sports", "Business", "Science/Technology"] return categories[prediction]# Example usagetext = "French veteran Gael Monfils becomes the oldest player to win an ATP Tour singles title."print(f"Predicted Category: {classify_text(text)}")Output:Predicted Category: SportsAdjusting Hyperparameters:Learning Rate: Fine-tune to balance convergence speed and stability. A smaller rate improves precision, while a larger rate speeds up training but may miss optimal solutions.Batch Size: Larger batches offer smoother gradients but require more memory. Smaller batches may generalize better.Epochs: Increase epochs for better learning but watch for overfitting (monitor validation performance).Model Architecture:Add Layers: Deeper networks can capture more complex patterns but require more data and careful regularization.Activation Functions: Experiment with ReLU, LeakyReLU, or GELU for better non-linearity and learning dynamics.Overfitting vs. Underfitting:Overfitting: Model performs well on training data but poorly on unseen data, indicating it has memorized patterns. Mitigation: Add dropout, use regularization (e.g., L2), or gather more data.Underfitting: Model fails to capture patterns in training data, often due to insufficient complexity or training. Solution: Add layers, increase training time, or adjust hyperparameters.By iterating on these strategies, you can systematically enhance your models performance and generalization.ConclusionBuilding a neural network from scratch in PyTorch is a hands-on way to understand deep learning. From dataset preprocessing to model training and optimization, weve covered the essentials to get you started. Along the way, we explored tips to improve performance and tackle challenges like overfitting and underfitting.Now its your turn to experiment! Adjust hyperparameters, modify the architecture, or try a new dataset. Share your results, insights, or questions, and keep exploring PyTorchs vast potential. Happy coding and learning!Link to colab Notebook: reading on basics of Pytorch: This tutorial shows how to use PyTorch to create a basic neural network for classifying handwritten digits from the MNIST dataset. Neural networks, which are central to modern AI, enable machines to learn tasks like regression, classification, and generation. With PyTorch, you'll learn how to design and train a neural network in Python to classify these handwritten numbers.Building Neural Network using PyTorch PyTorch offers two primary methods for building neural networks: using the nn.Module class or the nn.Sequential container.Using nn.Module: To create a custom network, subclass the nn.Module class and define the __init__ and forward functions. The __init__ method sets up the layers and parameters, while the forward function defines how input flows through the network and produces output.Using nn.Sequential: This container allows you to specify layers in a list. The layers are automatically connected in the order provided.Steps to Implement a Neural Network in PyTorch:1. Import Required Modules: Bring in necessary libraries like torch, torch.nn, and torch.optim.2. Define the Network Architecture: Specify the number and types of layers, activation functions, and output size. You can either subclass torch.nn.Module for custom layers or use preset layers like torch.nn.Linear, torch.nn.Conv2d, or torch.nn.LSTM.3. Set the Loss Function: Choose a loss function based on your task (e.g., torch.nn.MSELoss for regression, torch.nn.CrossEntropyLoss for classification).4. Choose an Optimizer: Specify an optimizer like torch.optim.SGD, torch.optim.Adam, or torch.optim.RMSprop to adjust the networks weights using gradients and learning rates.5. Train the Network: Perform forward and backward passes through the data, and update the weights using the optimizer. Monitor training progress by tracking the loss and additional metrics (e.g., accuracy).This structure simplifies building and training neural networks in Python with PyTorch.Implementing Feedforward Neural Network for MNIST using PyTorchLet's implement a Feedforward Neural Network (FNN) for classifying handwritten digits from the MNIST dataset using PyTorch. Step 1: Import the Necessary LibrariesWe start by importing the necessary PyTorch libraries, which include torch, torch.nn for building the model, torch.optim for the optimizer, and torchvision for dataset handling and image transformations. Python import torchimport torch.nn as nnimport torch.nn.functional as Fimport torch.optim as optimfrom torchvision import datasets, transformsimport matplotlib.pyplot as pltimport numpy as npStep 2: Define Hyperparameters and TransformationsWe set hyperparameters like batch_size, num_epochs, and learning_rate for training. A transformation pipeline is applied to MNIST images: converting them to tensors and normalizing the pixel values. Python batch_size = 64num_epochs = 10learning_rate = 0.01 transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,)) # Mean and Std of MNIST dataset])Step 3: Load and Prepare the DatasetThe MNIST dataset is loaded for both training and testing. We use DataLoader to manage batching and shuffling of data during training. Python train_dataset = datasets.MNIST(root='.', train=True, download=True, transform=transform)test_dataset = datasets.MNIST(root='.', train=False, download=True, transform=transform) train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)Step 4: Define the Neural Network ModelWe define a simple feedforward neural network with two fully connected layers. The first layer takes the flattened image (28x28 pixels) and outputs 512 features. The second layer outputs 10 classes (digits 0-9). Python class Net(nn.Module): def __init__(self): super(Net, self).__init__() self.fc1 = nn.Linear(28*28, 512) self.fc2 = nn.Linear(512, 10) def forward(self, x): x = x.view(-1, 28*28) # Flatten the image x = F.relu(self.fc1(x)) # ReLU activation x = self.fc2(x) # Output layer return xStep 5: Define the Loss Function, Optimizer, and Model InstanceWe use CrossEntropyLoss as the loss function for multi-class classification. The optimizer used is Stochastic Gradient Descent (SGD). Python device = torch.device("cuda" if torch.cuda.is_available() else "cpu")model = Net().to(device) criterion = nn.CrossEntropyLoss()optimizer = optim.SGD(model.parameters(), lr=learning_rate)Output:Net((fc1): Linear(in_features=784, out_features=512, bias=True) (fc2): Linear(in_features=512, out_features=10, bias=True))Step 6: Define the Training and Test LoopsThe training loop processes the batches, computes the gradients, and updates the model parameters. The test loop evaluates the model on the test dataset. Python def accuracy(outputs, labels): _, preds = torch.max(outputs, 1) return torch.sum(preds == labels).item() / len(labels) def train(model, device, train_loader, criterion, optimizer, epoch): model.train() running_loss = 0.0 running_acc = 0.0 for i, (inputs, labels) in enumerate(train_loader): inputs, labels = inputs.to(device), labels.to(device) optimizer.zero_grad() outputs = model(inputs) loss = criterion(outputs, labels) loss.backward() optimizer.step() running_loss += loss.item() running_acc += accuracy(outputs, labels) if (i + 1) % 200 == 0: print(f'Epoch {epoch}, Batch {i+1}, Loss: {running_loss / 200:.4f}, Accuracy: {running_acc / 200:.4f}') running_loss = 0.0 running_acc = 0.0 def test(model, device, test_loader, criterion): model.eval() test_loss = 0.0 test_acc = 0.0 with torch.no_grad(): for inputs, labels in test_loader: inputs, labels = inputs.to(device), labels.to(device) outputs = model(inputs) loss = criterion(outputs, labels) test_loss += loss.item() test_acc += accuracy(outputs, labels) print(f'Test Loss: {test_loss / len(test_loader):.4f}, Test Accuracy: {test_acc / len(test_loader):.4f}')Step 7: Train, Test, and Visualize ResultsThe model is trained for a set number of epochs and then tested. Additionally, a few sample predictions are visualized using matplotlib. Python for epoch in range(1, num_epochs + 1): train(model, device, train_loader, criterion, optimizer, epoch) test(model, device, test_loader, criterion) # Visualize sample images with predictionssamples, labels = next(iter(test_loader))samples = samples.to(device)outputs = model(samples)_, preds = torch.max(outputs, 1) samples = samples.cpu().numpy()fig, axes = plt.subplots(3, 3, figsize=(8, 8))for i, ax in enumerate(axes.ravel()): ax.imshow(samples[i].squeeze(), cmap='gray') ax.set_title(f'Label: {labels[i]}, Prediction: {preds[i]}') ax.axis('off')plt.tight_layout()plt.show()Output:Epoch 1, Batch 200, Loss: 1.1144, Accuracy: 0.7486Epoch 1, Batch 400, Loss: 0.4952, Accuracy: 0.8739Epoch 1, Batch 600, Loss: 0.3917, Accuracy: 0.8903Epoch 1, Batch 800, Loss: 0.3515, Accuracy: 0.9042Test Loss: 0.3018, Test Accuracy: 0.9155. . . Epoch 10, Batch 200, Loss: 0.1112, Accuracy: 0.9679Epoch 10, Batch 400, Loss: 0.1120, Accuracy: 0.9707Epoch 10, Batch 600, Loss: 0.1158, Accuracy: 0.9681Epoch 10, Batch 800, Loss: 0.1138, Accuracy: 0.9688Test Loss: 0.1145, Test Accuracy: 0.9665The output contains loss and accuracy at regular intervals along with the test accuracy. The final part of the code will show some test images along with their true labels and predicted labels. PyTorch is a Python package that provides two high-level features:Tensor computation (like NumPy) with strong GPU accelerationDeep neural networks built on a tape-based autograd systemYou can reuse your favorite Python packages such as NumPy, SciPy, and Cython to extend PyTorch when needed.Our trunk health (Continuous Integration signals) can be found at hud.pytorch.org. Learn the basics of PyTorchAt a granular level, PyTorch is a library that consists of the following components:ComponentDescriptiontorchA Tensor library like NumPy, with strong GPU supporttorch.autogradA tape-based automatic differentiation library that supports all differentiable Tensor operations in torchtorch.jitA compilation stack (TorchScript) to create serializable and optimizable models from PyTorch codetorch.nnA neural networks library deeply integrated with autograd designed for maximum flexibilitytorch.multiprocessingPython multiprocessing, but with magical memory sharing of torch Tensors across processes. Useful for data loading and Hogwild trainingtorch.utilsDataLoader and other utility functions for convenienceUsually, PyTorch is used either as:A replacement for NumPy to use the power of GPUs.A deep learning research platform that provides maximum flexibility and speed.Elaborating Further: If you use NumPy, then you have used Tensors (a.k.a. ndarray).PyTorch provides Tensors that can live either on the CPU or the GPU and accelerates thecomputation by a huge amount.We provide a wide variety of tensor routines to accelerate and fit your scientific computation needssuch as slicing, indexing, mathematical operations, linear algebra, reductions.And they are fast! PyTorch has a unique way of building neural networks: using and replaying a tape recorder.Most frameworks such as TensorFlow, Theano, Caffe, and CNTK have a static view of the world.One has to build a neural network and reuse the same structure again and again.Changing the way the network behaves means that one has to start from scratch.With PyTorch, we use a technique called reverse-mode auto-differentiation, which allows you tochange the way your network behaves arbitrarily with zero lag or overhead. Our inspiration comesfrom several research papers on this topic, as well as current and past work such astorch-autograd,autograd,Chainer, etc.While this technique is not unique to PyTorch, it's one of the fastest implementations of it to date.You get the best of speed and flexibility for your crazy research. PyTorch is not a Python binding into a monolithic C++ framework.It is built to be deeply integrated into Python.You can use it naturally like you would use NumPy / SciPy / scikit-learn etc.You can write your new neural network layers in Python itself, using your favorite librariesand use packages such as Cython and Numba.Our goal is to not reinvent the wheel where appropriate. PyTorch is designed to be intuitive, linear in thought, and easy to use.When you execute a line of code, it gets executed. There isn't an asynchronous view of the world.When you drop into a debugger or receive error messages and stack traces, understanding them is straightforward.The stack trace points to exactly where your code was defined.We hope you never spend hours debugging your code because of bad stack traces or asynchronous and opaque execution engines. PyTorch has minimal framework overhead. We integrate acceleration librariessuch as Intel MKL and NVIDIA (cuDNN, NCCL) to maximize speed.At the core, its CPU and GPU Tensor and neural network backendsare mature and have been tested for years.Hence, PyTorch is quite fast whether you run small or large neural networks.The memory usage in PyTorch is extremely efficient compared to Torch or some of the alternatives.We've written custom memory allocators for the GPU to make sure thatyour deep learning models are maximally memory efficient.This enables you to train bigger deep learning models than before. Writing new neural network modules, or interfacing with PyTorch's Tensor API was designed to be straightforwardand with minimal abstractions.You can write new neural network layers in Python using the torch APIor your favorite NumPy-based libraries such as SciPy.If you want to write your layers in C/C++, we provide a convenient extension API that is efficient and with minimal boilerplate.No wrapper code needs to be written. You can see a tutorial here and an example here. Commands to install binaries via Conda or pip wheels are on our website: Python wheels for NVIDIA's Jetson Nano, Jetson TX1/TX2, Jetson Xavier NX/AGX, and Jetson AGX Orin are provided here and the L4T container is published hereThey require JetPack 4.2 and above, and @dusty-nv and @ptrblck are maintaining them. If you are installing from source, you will need:Python 3.9 or laterA compiler that fully supports C++17, such as clang or gcc (gcc 9.4.0 or newer is required, on Linux)Visual Studio or Visual Studio Build Tool (Windows only)* PyTorch CI uses Visual C++ BuildTools, which come with Visual Studio Enterprise,Professional, or Community Editions. You can also install the build tools from . The build tools do notcome with Visual Studio Code by default.An example of environment setup is shown below: $ source /bin/activate$ conda create -y -n $ conda activate $ source \Scripts\activate.bat$ conda create -y -n $ conda activate $ call "C:\Program Files\Microsoft Visual Studio\\Community\VC\Auxiliary\Build\vcvarsall.bat" x64 If you want to compile with CUDA support, select a supported version of CUDA from our support matrix, then install the following:NVIDIA CUDANVIDIA cuDNN v8.5 or aboveCompiler compatible with CUDANote: You could refer to the cuDNN Support Matrix for cuDNN versions with the various supported CUDA, CUDA driver and NVIDIA hardwareIf you want to disable CUDA support, export the environment variable USE_CUDA=0.Other potentially useful environment variables may be found in setup.py.If you are building for NVIDIA's Jetson platforms (Jetson Nano, TX1, TX2, AGX Xavier), Instructions to install PyTorch for Jetson Nano are available here If you want to compile with ROCm support, installAMD ROCm 4.0 and above installationROCm is currently supported only for Linux systems.By default the build system expects ROCm to be installed in /opt/rocm. If ROCm is installed in a different directory, the ROCM_PATH environment variable must be set to the ROCm installation directory. The build system automatically detects the AMD GPU architecture. Optionally, the AMD GPU architecture can be explicitly set with the PYTORCH_ROCM_ARCH environment variable AMD GPU architectureIf you want to disable ROCm support, export the environment variable USE_ROCM=0.Other potentially useful environment variables may be found in setup.py. If you want to compile with Intel GPU support, follow these If you want to disable Intel GPU support, export the environment variable USE_XPU=0.Other potentially useful environment variables may be found in setup.py. git clone pytorch# if you are updating an existing checkoutgit submodule syncgit submodule update --init --recursive Commonconda install cmake ninja# Run this command from the PyTorch directory after cloning the source code using the Get the PyTorch Source section belowpip install -r requirements.txtOn Linuxpip install mkl-static mkl-include# CUDA only: Add LAPACK support for the GPU if needed# magma installation: run with active conda environment. specify CUDA version to install.ci/docker/common/install_magma_conda.sh 12.4 # (optional) If using torch.compile with inductor/triton, install the matching version of triton# Run from the pytorch directory after cloning# For Intel GPU support, please explicitly `export USE_XPU=1` before running command.make tritonOn MacOS# Add this package on intel x86 processor machines onlypip install mkl-static mkl-include# Add these packages if torch.distributed is neededconda install pkg-config libuvOn Windowspip install mkl-static mkl-include# Add these packages if torch.distributed is needed.# Distributed package support on Windows is a prototype feature and is subject to changes.conda install -c conda-forge libuv=1.39 On LinuxIf you're compiling for AMD ROCm then first run this command:# Only run this if you're compiling for ROCmpython tools/amd_build/build_amd.pyInstall PyTorchexport CMAKE_PREFIX_PATH="${CONDA_PREFIX:-'$(dirname $(which conda))/../'}:${CMAKE_PREFIX_PATH}"python setup.py developOn macOS On WindowsIf you want to build legacy python code, please refer to Building on legacy code and CUDACPU-only buildsIn this mode PyTorch computations will run on your CPU, not your GPU. Note on OpenMP: The desired OpenMP implementation is Intel OpenMP (iomp). In order to link against iomp, you'll need to manually download the library and set up the building environment by tweaking CMAKE_INCLUDE_PATH and LIB. The instruction here is an example for setting up both MKL and Intel OpenMP. Without these configurations for CMake, Microsoft Visual C OpenMP runtime (vcomp) will be used.CUDA based buildIn this mode PyTorch computations will leverage your GPU via CUDA for faster number crunchingNVTX is needed to build Pytorch with CUDA.NVTX is a part of CUDA distributive, where it is called "Nsight Compute". To install it onto an already installed CUDA run CUDA installation once again and check the corresponding checkbox.Make sure that CUDA with Nsight Compute is installed after Visual Studio.Currently, VS 2017 / 2019, and Ninja are supported as the generator of CMake. If ninja.exe is detected in PATH, then Ninja will be used as the default generator, otherwise, it will use VS 2017 / 2019. If Ninja is selected as the generator, the latest MSVC will get selected as the underlying toolchain.Additional libraries such asMagma, oneDNN, a.k.a. MKLDNN or DNNL, and Sccache are often needed. Please refer to the installation-helper to install them.You can refer to the build_pytorch.bat script for some other environment variables configurationscmd :: Set the environment variables after you have downloaded and unzipped the mkl package,:: else CMake would throw an error as `Could NOT find OpenMP`.set CMAKE_INCLUDE_PATH={Your directory}\mkl\includeset LIB={Your directory}\mkl\lib;%LIB% :: Read the content in the previous section carefully before you proceed.:: [Optional] If you want to override the underlying toolset used by Ninja and Visual Studio with CUDA, please run the following script block.:: "Visual Studio 2019 Developer Command Prompt" will be run automatically.:: Make sure you have CMake >= 3.12 before you do this when you use the Visual Studio generator.set CMAKE_GENERATOR_TOOLSET_VERSION=14.27set DISTUTILS_USE_SDK=1for /f "usebackq tokens=*" %i in (`"%ProgramFiles(x86)%\Microsoft Visual Studio\Installer\vswhere.exe" -version [15^,17^) -products * -latest -property installationPath`) do call "%i\VC\Auxiliary\Build\vcvarsall.bat" x64 -vcvars_ver=%CMAKE_GENERATOR_TOOLSET_VERSION% :: [Optional] If you want to override the CUDA host compilerset CUDAHOSTCXX=C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.27.29110\bin\HostX64\x64\cl.exepython setup.py developIntel GPU buildsIn this mode PyTorch with Intel GPU support will be built.Please make sure the common prerequisites as well as the prerequisites for Intel GPU are properly installed and the environment variables are configured prior to starting the build. For build tool support, Visual Studio 2022 is required.Then PyTorch can be built with the command::: CMD Commands::: Set the CMAKE_PREFIX_PATH to help find corresponding packages:: %CONDA_PREFIX% only works after `conda activate custom_env` if defined CMAKE_PREFIX_PATH (set "CMAKE_PREFIX_PATH=%CONDA_PREFIX%\Library;%CMAKE_PREFIX_PATH%") else (set "CMAKE_PREFIX_PATH=%CONDA_PREFIX%\Library")python setup.py develop You can adjust the configuration of cmake variables optionally (without building first), by doingthe following. For example, adjusting the pre-detected directories for CuDNN or BLAS can be donewith such a step.On Linuxexport CMAKE_PREFIX_PATH="${CONDA_PREFIX:-'$(dirname $(which conda))/../'}:${CMAKE_PREFIX_PATH}"python setup.py build --cmake-onlyccmake build # or cmake-gui buildOn macOSexport CMAKE_PREFIX_PATH="${CONDA_PREFIX:-'$(dirname $(which conda))/../'}:${CMAKE_PREFIX_PATH}"MACOSX_DEPLOYMENT_TARGET=10.9 CC=clang CXX=clang++ python setup.py build --cmake-onlyccmake build # or cmake-gui build You can also pull a pre-built docker image from Docker Hub and run with docker v19.03+docker run --gpus all --rm -ti --ipc=host pytorch/pytorch:latestPlease note that PyTorch uses shared memory to share data between processes, so if torch multiprocessing is used (e.g.for multithreaded data loaders) the default shared memory segment size that container runs with is not enough, and youshould increase shared memory size either with --ipc=host or --shm-size command line options to nvidia-docker run. NOTE: Must be built with a docker version > 18.06The Dockerfile is supplied to build images with CUDA 11.1 support and cuDNN v8.You can pass PYTHON_VERSION=x.y make variable to specify which Python version is to be used by Miniconda, or leave itunset to use the default.make -f docker.Makefile# images are tagged as docker.io/${your_docker_username}/pytorchYou can also pass the CMAKE_VARS="..." environment variable to specify additional CMake variables to be passed to CMake during the build.See setup.py for the list of available variables. To build documentation in various formats, you will need Sphinxand the pytorch_sphinx_theme2.Before you build the documentation locally, ensure torch isinstalled in your environment. For small fixes, you can install thenightly version as described in Getting Started.For more complex fixes, such as adding a new module and docstrings forthe new module, you might need to install torch from source.See Docstring Guidelinesfor docstring conventions.cd docs/pip install -r requirements.txtmake htmlmake serveRun make to get a list of all available output formats.If you get a katex error run npm install katex. If it persists, trynpm install -g katexNoteIf you installed nodejs with a different package manager (e.g.,conda) then npm will probably install a version of katex that is notcompatible with your version of nodejs and doc builds will fail.A combination of versions that is known to work is node@6.13.1 andkatex@0.13.18. To install the latter with npm you can runnpm install -g katex@0.13.18NoteIf you see a numpy incompatibility error, run: When you make changes to the dependencies run by CI, edit the.ci/docker/requirements-docs.txt file. To compile a PDF of all PyTorch documentation, ensure you havetexlive and LaTeX installed. On macOS, you can install them using:brew install --cask mactexTo create the PDF:Run: This will generate the necessary files in the build/latex directory.Navigate to this directory and execute:make LATEXOPTS="-interaction=nonstopmode"This will produce a pytorch.pdf with the desired content. Run thiscommand one more time so that it generates the correct tableof contents and index.NoteTo view the Table of Contents, switch to the Table of Contentsview in your PDF viewer. Installation instructions and binaries for previous PyTorch versions may be foundon our website. Three-pointers to get you started: Forums: Discuss implementations, research, etc. Issues: Bug reports, feature requests, install issues, RFCs, thoughts, etc.Slack: The PyTorch Slack hosts a primary audience of moderate to experienced PyTorch users and developers for general chat, online discussions, collaboration, etc. If you are a beginner looking for help, the primary medium is PyTorch Forums. If you need a slack invite, please fill this form: No-noise, a one-way email newsletter with important announcements about PyTorch. You can sign-up here: Page: Important announcements about PyTorch. brand guidelines, please visit our website at pytorch.org Typically, PyTorch has three minor releases a year. Please let us know if you encounter a bug by filing an issue.We appreciate all contributions. If you are planning to contribute back bug-fixes, please do so without any further discussion.If you plan to contribute new features, utility functions, or extensions to the core, please first open an issue and discuss the feature with us.Sending a PR without discussion might end up resulting in a rejected PR because we might be taking the core in a different direction than you might be aware of.To learn more about making a contribution to Pytorch, please see our Contribution page. For more information about PyTorch releases, see Release page. PyTorch is a community-driven project with several skillful engineers and researchers contributing to it.PyTorch is currently maintained by Soumith Chintala, Gregory Chanan, Dmytro Dzhulgakov, Edward Yang, and Nikita Shulga with major contributions coming from hundreds of talented individuals in various forms and means.A non-exhaustive but growing list needs to mention: Trevor Killeen, Sasank Chilamkurthy, Sergey Zagoruyko, Adam Lerer, Francisco Massa, Alykhan Tejani, Luca Antiga, Alban Desmaison, Andreas Koepf, James Bradbury, Zeming Lin, Yuandong Tian, Guillaume Lample, Marat Dukhan, Natalia Gimelshein, Christian Sarofeen, Martin Raison, Edward Yang, Zachary Devito.Note: This project is unrelated to hughperkins/pytorch with the same name. Hugh is a valuable contributor to the Torch community and has helped with many things Torch and PyTorch. PyTorch has a BSD-style license, as found in the LICENSE file. On May 17, the PyTorch Meetup was successfully held in Hangzhou, drawing nearly 60 developers and industry experts from companies including Huawei, Tencent, Ant Group, and ByteDance. The event focused Read More PyTorch/XLA is a Python package that uses the XLA deep learning compiler to enable PyTorch deep learning workloads on various hardware backends, including Google Cloud TPUs, GPUs, and AWS Inferentia/Trainium. Read More Mixture-of-Experts (MoE) is a popular model architecture for large language models (LLMs). Although it reduces computation in training and inference by activating fewer parameters per token, it imposes additional challenges Read More How can financial brands set themselves apart through visual storytelling? Our experts explainhow.Learn MoreThe Motorsport Images Collections captures events from 1895 to todays most recentcoverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of EditorsPicks.Browse Editors' FavoritesHow can financial brands set themselves apart through visual storytelling? Our experts explainhow.Learn MoreThe Motorsport Images Collections captures events from 1895 to todays most recentcoverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of EditorsPicks.Browse Editors' FavoritesHow can financial brands set themselves apart through visual storytelling? Our experts explainhow.Learn MoreThe Motorsport Images Collections captures events from 1895 to todays most recentcoverage.Discover The CollectionCurated, compelling, and worth your time. Explore our latest gallery of EditorsPicks.Browse Editors' Favorites

Simple neural network pytorch. Physics informed neural network a simple tutorial with pytorch. Pytorch simple neural network example. Pytorch vs tensorflow for beginners. Simple pytorch.