PyTorch for Absolute Beginners: Zero to Hero Guide

Table of Contents

---

1. What is PyTorch?

PyTorch is a powerful, open-source machine learning library developed by Facebook (Meta). Think of it as a toolkit that makes building and training neural networks much easier.

Why PyTorch?

- **Easy to learn**: Pythonic syntax that feels natural
- **Dynamic**: You can change your network structure on the fly
- **Powerful**: Used by researchers and companies worldwide
- **Great community**: Lots of tutorials, examples, and help available

Real-world applications:

- Image recognition (like photo tagging on social media)
- Natural language processing (chatbots, translation)
- Recommendation systems (Netflix, Spotify suggestions)
- Self-driving cars
- Medical diagnosis

---

2. Installation and Setup

Installing PyTorch

```bash
# For CPU only (good for learning)
pip install torch torchvision torchaudio

# For GPU (if you have NVIDIA GPU)
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
```

Verify Installation

```python
```

```python
import torch
print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")

# This should print your PyTorch version
# CUDA shows if you can use GPU acceleration
```

Practice Exercise 1: Run the code above and note down your PyTorch version. Don't worry if CUDA shows False - CPU is fine for learning!

---

3. Understanding Tensors

Tensors are the fundamental building blocks of PyTorch. Think of them as multi-dimensional arrays that can run on GPUs.

What are Tensors?

```
python
```

```python
import torch

# Scalar (0D tensor) - just a single number
scalar = torch.tensor(5)
print(f"Scalar: {scalar}")
print(f"Shape: {scalar.shape}")

# Vector (1D tensor) - like a list of numbers
vector = torch.tensor([1, 2, 3, 4])
print(f"Vector: {vector}")
print(f"Shape: {vector.shape}")

# Matrix (2D tensor) - like a spreadsheet
matrix = torch.tensor([[1, 2], [3, 4], [5, 6]])
print(f"Matrix:\n{matrix}")
print(f"Shape: {matrix.shape}")

# 3D tensor - like a stack of matrices
tensor_3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(f"3D Tensor:\n{tensor_3d}")
print(f"Shape: {tensor_3d.shape}")
```

Creating Tensors

```python
```

```python
# Create tensors with specific values
zeros = torch.zeros(3, 4)  # 3x4 tensor of zeros
ones = torch.ones(2, 3)    # 2x3 tensor of ones
random = torch.randn(2, 2) # 2x2 tensor with random values

print("Zeros:\n", zeros)
print("Ones:\n", ones)
print("Random:\n", random)

# Create tensor from Python list
from_list = torch.tensor([[1, 2, 3], [4, 5, 6]])
print("From list:\n", from_list)

# Create tensor with specific data type
float_tensor = torch.tensor([1, 2, 3], dtype=torch.float32)
int_tensor = torch.tensor([1, 2, 3], dtype=torch.int64)
print(f"Float tensor: {float_tensor}, dtype: {float_tensor.dtype}")
print(f"Int tensor: {int_tensor}, dtype: {int_tensor.dtype}")
```

Practice Exercise 2: Create the following tensors:

1. A 5x5 matrix of zeros
2. A 1D tensor with values [10, 20, 30, 40, 50]
3. A 2x3 matrix of random numbers
4. A 3x3 identity matrix (hint: use `torch.eye()`)

---

4. Basic Tensor Operations

Mathematical Operations

python

```python
# Create sample tensors
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# Basic arithmetic
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)

# Matrix operations
matrix_a = torch.tensor([[1, 2], [3, 4]])
matrix_b = torch.tensor([[5, 6], [7, 8]])

print("Matrix addition:\n", matrix_a + matrix_b)
print("Matrix multiplication:\n", torch.matmul(matrix_a, matrix_b))
# or use @ operator
print("Matrix multiplication (@ operator):\n", matrix_a @ matrix_b)
```

Reshaping Tensors

python

```
# Create a tensor
x = torch.tensor([[1, 2, 3, 4], [5, 6, 7, 8]])
print("Original shape:", x.shape)
print("Original tensor:\n", x)

# Reshape to different dimensions
reshaped = x.view(4, 2)  # 4 rows, 2 columns
print("Reshaped (4, 2):\n", reshaped)

reshaped2 = x.view(8)  # Flatten to 1D
print("Flattened:", reshaped2)

# -1 means "figure out this dimension automatically"
auto_reshape = x.view(-1, 1)  # Make it a column vector
print("Auto-reshaped to column:\n", auto_reshape)
```

Indexing and Slicing

```python
```

```python
# Create a 3x4 matrix
matrix = torch.tensor([[1, 2, 3, 4],
                       [5, 6, 7, 8],
                       [9, 10, 11, 12]])

print("Original matrix:\n", matrix)

# Access elements
print("Element at [1, 2]:", matrix[1, 2])  # Row 1, Column 2
print("First row:", matrix[0, :])          # All columns in row 0
print("Last column:", matrix[:, -1])        # All rows in last column
print("First 2 rows:", matrix[:2, :])       # First 2 rows, all columns

# Modify elements
matrix[0, 0] = 999
print("After modification:\n", matrix)
```

Practice Exercise 3:

1. Create two 3x3 matrices with random values
2. Add them together
3. Multiply them using matrix multiplication
4. Reshape the result into a 1D tensor
5. Extract the first 5 elements

---

5. Automatic Differentiation (Autograd)

This is where PyTorch becomes magical! Autograd automatically calculates gradients (derivatives) for you, which is essential for training neural networks.

Understanding Gradients

python

```python
# Create a tensor that requires gradients
x = torch.tensor(2.0, requires_grad=True)
print(f"x = {x}")

# Define a function
y = x**2 + 3*x + 1
print(f"y = x² + 3x + 1 = {y}")

# Calculate the gradient (derivative)
y.backward()  # This computes dy/dx
print(f"Gradient dy/dx = {x.grad}")

# Mathematical check: dy/dx = 2x + 3 = 2(2) + 3 = 7 ✓
```

More Complex Example

python

```python
# Multiple variables
x = torch.tensor(1.0, requires_grad=True)
y = torch.tensor(2.0, requires_grad=True)

# More complex function
z = x**2 + y**3 + x*y
print(f"z = x² + y³ + xy = {z}")

# Calculate gradients
z.backward()
print(f"∂z/∂x = {x.grad}")  # Should be 2x + y = 2(1) + 2 = 4
print(f"∂z/∂y = {y.grad}")  # Should be 3y² + x = 3(4) + 1 = 13
```

Vector Gradients

```python
# Reset gradients (important!)
if x.grad is not None:
    x.grad.zero_()
if y.grad is not None:
    y.grad.zero_()

# Vector operations
x = torch.tensor([1.0, 2.0], requires_grad=True)
y = torch.sum(x**2)  # Sum of squares

y.backward()
print(f"x = {x}")
print(f"y = sum(x²) = {y}")
print(f"Gradient: {x.grad}")  # Should be [2*1, 2*2] = [2, 4]
```

Practice Exercise 4:

1. Create a tensor x = 3.0 with requires_grad=True
2. Define $y = x^3 - 2x^2 + x - 5$
3. Calculate the gradient
4. Verify your answer: $dy/dx = 3x^2 - 4x + 1$

---

6. Building Neural Networks

Neural networks in PyTorch are built using the `torch.nn` module. Let's start simple!

Your First Neural Network

```python
```

```python
import torch
import torch.nn as nn

# Define a simple neural network
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        # Define layers
        self.layer1 = nn.Linear(2, 4)  # Input: 2 features, Output: 4 neurons
        self.layer2 = nn.Linear(4, 1)  # Input: 4 neurons, Output: 1 neuron
        self.activation = nn.ReLU()    # Activation function

    def forward(self, x):
        # Define how data flows through the network
        x = self.layer1(x)
        x = self.activation(x)
        x = self.layer2(x)
        return x

# Create the network
net = SimpleNet()
print(net)

# Test with dummy data
input_data = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
output = net(input_data)
print(f"Input: {input_data}")
print(f"Output: {output}")
```

Understanding the Components

```python
# Let's break down what each part does

# 1. Linear Layer (Fully Connected)
linear = nn.Linear(3, 2)  # 3 inputs → 2 outputs
print("Linear layer weights shape:", linear.weight.shape)
print("Linear layer bias shape:", linear.bias.shape)

# 2. Activation Functions
relu = nn.ReLU()        # ReLU: max(0, x)
sigmoid = nn.Sigmoid()  # Sigmoid: 1/(1+e^(-x))
tanh = nn.Tanh()        # Tanh: (e^x - e^(-x))/(e^x + e^(-x))

# Test activation functions
test_input = torch.tensor([-2.0, -1.0, 0.0, 1.0, 2.0])
print("Input:", test_input)
print("ReLU:", relu(test_input))
print("Sigmoid:", sigmoid(test_input))
print("Tanh:", tanh(test_input))
```

A More Complete Example

```python
```

```python
class BetterNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(BetterNet, self).__init__()

        # Define layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

        # Activation and dropout
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.1)  # Helps prevent overfitting

    def forward(self, x):
        # Layer 1
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)

        # Layer 2
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout(x)

        # Output layer (no activation for regression)
        x = self.fc3(x)
        return x

# Create network: 10 inputs → 20 hidden → 20 hidden → 1 output
model = BetterNet(10, 20, 1)
print(f"Model has {sum(p.numel() for p in model.parameters())} parameters")
```

```python
# Test with random data
test_input = torch.randn(5, 10)  # 5 samples, 10 features each
output = model(test_input)
print(f"Input shape: {test_input.shape}")
print(f"Output shape: {output.shape}")
```

Practice Exercise 5: Create a neural network with:

- Input size: 5
- First hidden layer: 10 neurons with ReLU
- Second hidden layer: 8 neurons with ReLU
- Output layer: 3 neurons Test it with random input data.

---

7. Training Your First Model

Now let's learn how to actually train a neural network! We'll solve a simple regression problem.

The Training Process

```
python
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# Step 1: Create synthetic data
# Let's learn the function y = 2x + 1 + noise
torch.manual_seed(42)  # For reproducible results

n_samples = 100
X = torch.randn(n_samples, 1)  # Input features
y = 2 * X + 1 + 0.1 * torch.randn(n_samples, 1)  # Target values

print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")
print(f"First 5 samples:")
print(f"X: {X[:5].flatten()}")
print(f"y: {y[:5].flatten()}")
```

Define the Model

```
python
```

```python
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(1, 1)  # 1 input, 1 output

    def forward(self, x):
        return self.linear(x)

# Create model, loss function, and optimizer
model = LinearRegression()
criterion = nn.MSELoss()  # Mean Squared Error
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Stochastic Gradient Descent

print("Initial parameters:")
for name, param in model.named_parameters():
    print(f"{name}: {param.data}")
```

Training Loop

python

```
# Training parameters
num_epochs = 1000
losses = []

# Training loop
for epoch in range(num_epochs):
    # Forward pass
    predictions = model(X)
    loss = criterion(predictions, y)

    # Backward pass
    optimizer.zero_grad()  # Clear gradients
    loss.backward()        # Calculate gradients
    optimizer.step()       # Update parameters

    # Store loss for plotting
    losses.append(loss.item())

    # Print progress
    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

print("\nFinal parameters:")
for name, param in model.named_parameters():
    print(f"{name}: {param.data}")

print(f"\nTarget was: y = 2x + 1")
print(f"Model learned: y = {model.linear.weight.item():.3f}x + {model.linear.bias.item():.3f}")
```

Visualize Results

```python
# Plot training loss
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(losses)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)

# Plot predictions vs actual
plt.subplot(1, 2, 2)
with torch.no_grad():  # Don't calculate gradients for inference
    predictions = model(X)

plt.scatter(X.numpy(), y.numpy(), alpha=0.5, label='Actual')
plt.scatter(X.numpy(), predictions.numpy(), alpha=0.5, label='Predicted')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.title('Actual vs Predicted')
plt.grid(True)

plt.tight_layout()
plt.show()
```

Making Predictions

```python
```

```python
# Test on new data
new_X = torch.tensor([[0.5], [1.0], [1.5], [2.0]])

model.eval()  # Set to evaluation mode
with torch.no_grad():
    new_predictions = model(new_X)

print("Predictions on new data:")
for i in range(len(new_X)):
    x_val = new_X[i].item()
    pred = new_predictions[i].item()
    actual = 2 * x_val + 1  # True function
    print(f"X={x_val:.1f}: Predicted={pred:.3f}, Actual={actual:.1f}")
```

Practice Exercise 6:

1. Create data for the function $y = 0.5x^2 + 3x - 2$
2. Design a neural network to learn this function
3. Train it and visualize the results
4. How well does it perform?

---

8. Working with Data

Real-world data doesn't come as nice tensors. Let's learn how to handle datasets properly.

DataLoader and Datasets

```
python
```

```python
from torch.utils.data import Dataset, DataLoader
import numpy as np

class CustomDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Create sample data
np.random.seed(42)
n_samples = 1000
X_data = np.random.randn(n_samples, 5)  # 5 features
y_data = np.sum(X_data, axis=1) + np.random.randn(n_samples) * 0.1

# Create dataset and dataloader
dataset = CustomDataset(X_data, y_data)
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

print(f"Dataset size: {len(dataset)}")
print(f"Number of batches: {len(dataloader)}")

# Look at one batch
for batch_X, batch_y in dataloader:
    print(f"Batch X shape: {batch_X.shape}")
```

```
    print(f"Batch y shape: {batch_y.shape}")
    break
```

Training with Batches

```
python
```

```
class MultiFeatureNet(nn.Module):
    def __init__(self):
        super(MultiFeatureNet, self).__init__()
        self.fc1 = nn.Linear(5, 10)
        self.fc2 = nn.Linear(10, 5)
        self.fc3 = nn.Linear(5, 1)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x.squeeze()  # Remove extra dimension

# Create model, loss, optimizer
model = MultiFeatureNet()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training with batches
num_epochs = 50
train_losses = []

for epoch in range(num_epochs):
    epoch_loss = 0.0

    for batch_X, batch_y in dataloader:
        # Forward pass
        predictions = model(batch_X)
```

```
        loss = criterion(predictions, batch_y)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    # Average loss for this epoch
    avg_loss = epoch_loss / len(dataloader)
    train_losses.append(avg_loss)

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Average Loss: {avg_loss:.4f}')

# Plot training progress
plt.plot(train_losses)
plt.title('Training Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
```

Data Preprocessing

```
python
```

```python
# Common preprocessing techniques
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

# Split data into train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_data, y_data, test_size=0.2, random_state=42
)

# Normalize features (important for neural networks!)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Before scaling:")
print(f"X_train mean: {X_train.mean(axis=0)}")
print(f"X_train std: {X_train.std(axis=0)}")

print("\nAfter scaling:")
print(f"X_train_scaled mean: {X_train_scaled.mean(axis=0)}")
print(f"X_train_scaled std: {X_train_scaled.std(axis=0)}")

# Create datasets
train_dataset = CustomDataset(X_train_scaled, y_train)
test_dataset = CustomDataset(X_test_scaled, y_test)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Practice Exercise 7:

1. Create a dataset with 3 features and 1000 samples
2. Split it into train/test sets (80/20)
3. Normalize the features
4. Create DataLoaders with batch size 16
5. Train a model and evaluate on test set

---

9. Complete Example: Image Classification

Let's put everything together with a real computer vision task using the famous MNIST dataset (handwritten digits).

Loading MNIST Data

python

```python
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# Define data transformations
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert PIL image to tensor
    transforms.Normalize((0.1307,), (0.3081,))  # Normalize with MNIST mean/std
])

# Download and load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

print(f"Training samples: {len(train_dataset)}")
print(f"Test samples: {len(test_dataset)}")
```

```python
# Look at the data
import matplotlib.pyplot as plt

# Get one batch of training data
dataiter = iter(train_loader)
images, labels = next(dataiter)

print(f"Batch shape: {images.shape}")  # [batch_size, channels, height, width]
print(f"Labels shape: {labels.shape}")

# Visualize some images
fig, axes = plt.subplots(2, 4, figsize=(10, 5))
for i in range(8):
    row, col = i // 4, i % 4
    axes[row, col].imshow(images[i].squeeze(), cmap='gray')
    axes[row, col].set_title(f'Label: {labels[i]}')
    axes[row, col].axis('off')
plt.tight_layout()
plt.show()
```

Define CNN Model

```
python
```

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)  # 28x28 -> 28x28
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # 28x28 -> 28x28

        # Pooling layer
        self.pool = nn.MaxPool2d(2, 2)  # 28x28 -> 14x14, then 14x14 -> 7x7

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # 64 channels * 7x7 pixels
        self.fc2 = nn.Linear(128, 10)  # 10 classes (digits 0-9)

        # Activation and dropout
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        # Convolutional layers with pooling
        x = self.pool(self.relu(self.conv1(x)))  # 28x28 -> 14x14
        x = self.pool(self.relu(self.conv2(x)))  # 14x14 -> 7x7

        # Flatten for fully connected layers
        x = x.view(-1, 64 * 7 * 7)

        # Fully connected layers
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
```

```
        return x

# Create model
model = CNN()
print(model)

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'Total parameters: {total_params:,}')
print(f'Trainable parameters: {trainable_params:,}')
```

Training the CNN

```
python
```

```
# Training setup
criterion = nn.CrossEntropyLoss()  # For classification
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training function
def train_model(model, train_loader, criterion, optimizer, num_epochs=5):
    model.train()  # Set to training mode

    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0

        for batch_idx, (images, labels) in enumerate(train_loader):
            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Statistics
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if batch_idx % 200 == 199:  # Print every 200 mini-batches
                print(f'Epoch [{epoch+1}/{num_epochs}], '
                      f'Step [{batch_idx+1}/{len(train_loader)}], '
```

```
                f'Loss: {running_loss/200:.4f}')
            running_loss = 0.0

        # Print epoch results
        accuracy = 100 * correct / total
        print(f'Epoch [{epoch+1}/{num_epochs}] - Accuracy: {accuracy:.2f}%')

# Train the model
train_model(model, train_loader, criterion, optimizer, num_epochs=5)
```

Evaluate the Model

```
python
```

```python
def evaluate_model(model, test_loader):
    model.eval()  # Set to evaluation mode
    correct = 0
    total = 0
    class_correct = list(0. for i in range(10))
    class_total = list(0. for i in range(10))

    with torch.no_grad():
        for images, labels in test_loader:
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # Per-class accuracy
            c = (predicted == labels).squeeze()
            for i in range(labels.size(0)):
                label = labels[i]
                class_correct[label] += c[i].item()
                class_total[label] += 1

    # Overall accuracy
    accuracy = 100 * correct / total
    print(f'Overall Test Accuracy: {accuracy:.2f}%')

    # Per-class accuracy
    for i in range(10):
        if class_total[i] > 0:
            class_acc = 100 * class_correct[i] / class_total[i]
            print(f'Accuracy of digit {i}: {class_acc:.2f}%')
```

```
# Evaluate the model
evaluate_model(model, test_loader)
```

Visualize Predictions

```python
def visualize_predictions(model, test_loader, num_images=8):
    model.eval()

    # Get one batch of test data
    dataiter = iter(test_loader)
    images, labels = next(dataiter)

    # Make predictions
    with torch.no_grad():
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

    # Plot images with predictions
    fig, axes = plt.subplots(2, 4, figsize=(12, 6))
    for i in range(num_images):
        row, col = i // 4, i % 4
        axes[row, col].imshow(images[i].squeeze(), cmap='gray')

        # Color: green if correct, red if wrong
        color = 'green' if predicted[i] == labels[i] else 'red'
```