GETTING STARTED WITH PYTORCH: A BEGINNER'S GUIDE TO DEEP LEARNING

Youva ADDAD

Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, Caen, FRANCE

youva.addad@unicaen.fr

GREYC Electronics and Computer Science Laboratory

Normandie Université

UNICAEN

ENSI CAEN ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

cnrs

RÉGION NORMANDIE

Outline

What is Deep Learning?

Machine Learning is turning things (data) into numbers and finding patterns in those numbers.


Traditional programming
Inputs
Rules
Output
1. Cut vegetables
2. Season chicken
3. Preheat oven
4. Cook chicken for 30-minutes
5. Add vegetables
Starts with
Makes
Machine learning algorithm
Inputs
Output
Rules
1. Cut vegetables
2. Season chicken
3. Preheat oven
4. Cook chicken for 30-minutes
5. Add vegetables
Starts with
Figures out

YouTube

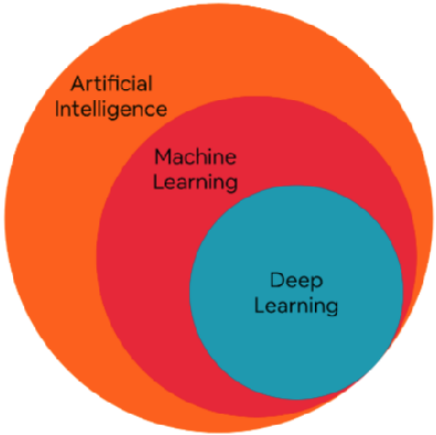
Yashaswi Kulshreshtha commented on your video

2020 Machine Learning Roadmap

Yashaswi Kulshreshtha

I think you can use ML for literally anything as long as you can convert it into numbers and program it to find patterns. Literally it could be anything any input or output from the universe

Machine Learning vs. Deep Learning



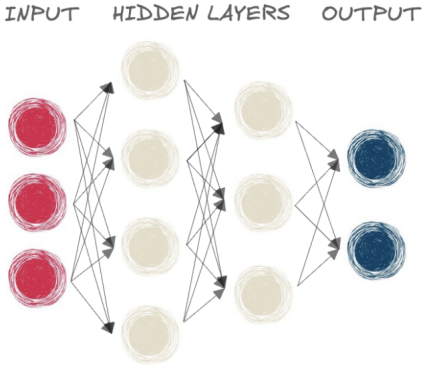
What is deep learning

- Deep learning is everywhere:
  - Language translation
  - Self-driving cars
  - Medical diagnostics
  - Chatbots
- Traditional machine learning: relies on hand-crafted **feature engineering**
- Deep learning: enables **feature learning** from raw data

What is deep learning

DEEP LEARNING

What is deep learning



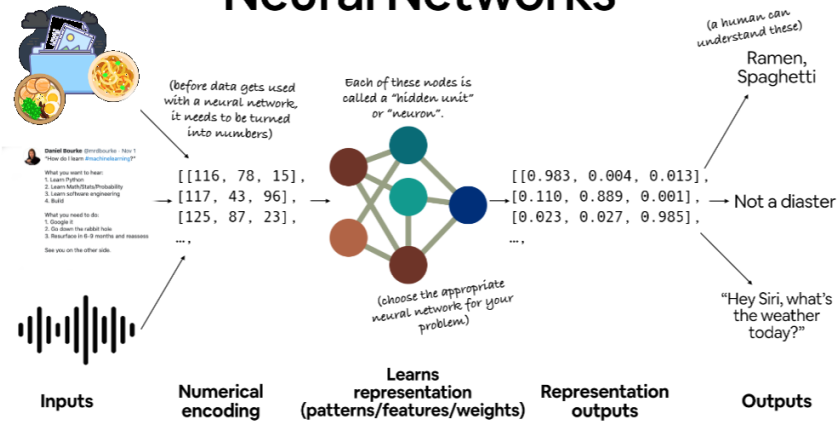
What is deep learning

- Inspired by connections in the human brain.
- Neurons $\Longrightarrow$ neural networks.
- Layered Structure: The term ”deep” refers to the presence of multiple layers in the network.
- Models require large amount of data.
- At least 100,000s data points.

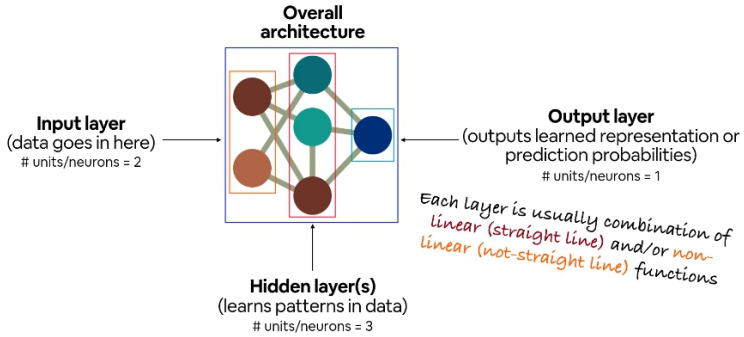
Outline

What is Neural Networks?

Neural Networks



Anatomy of Neural Networks



Note: "patterns" is an arbitrary term, you'll often hear "embedding", "weights", "feature representation", "feature vectors" all referring to similar things.

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

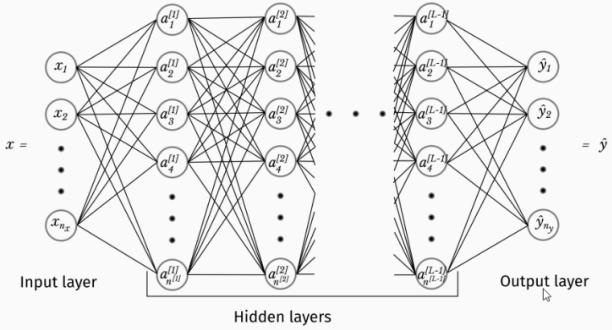
$$a_k^{[L]} = s(z_k^{[L]})$$

$$= \hat{y}_k$$

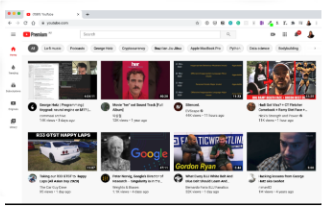
for

$$k = 1, \ldots, n_y,$$

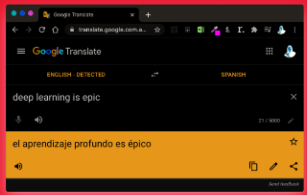
$$= 1, \ldots, n^{[L]}.$$



(some) Deep Learning Use Cases



Recommendation



Translation

"Hey Siri, who's the biggest big dog of them all?"

Speech recognition

Sequence to sequence (seq2seq)

Computer Vision

To: daniel@mrdbourke.com
Hey Daniel,

This deep learning course is incredible! I can't wait to use what I've learned!

Not spam

To: daniel@mrdbourke.com
Hey daniel...

COongratu1ations! U win $1139239230

Spam

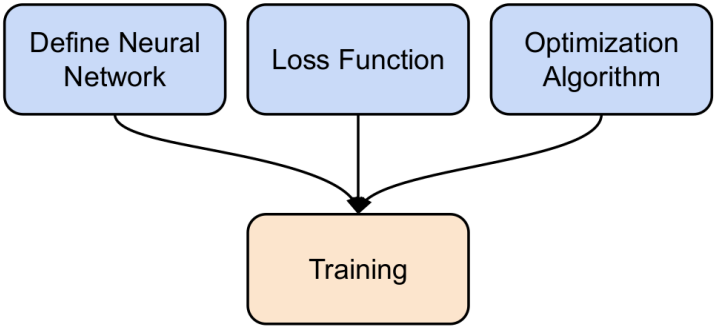
Natural Language Processing (NLP)

Classification/regression

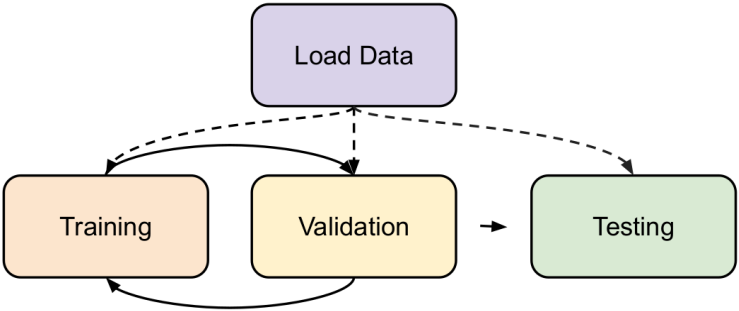
Outline

Deep Learning Workflow

Training Neural Networks



Training & Testing Neural Networks



Outline


PyTorch

What is PyTorch?

- Most popular research deep learning framework.
- Intuitive and user-friendly.
- N-dimensional Tensor computation (like NumPy) on GPUs.
- Create fast Python deep learning code for single or multi-GPU use.
- Automatic differentiation for training deep neural networks.
- End-to-end: preprocess, model, and deploy in app/cloud.
- Access to various pre-built deep learning models (Torch Hub/Timm).

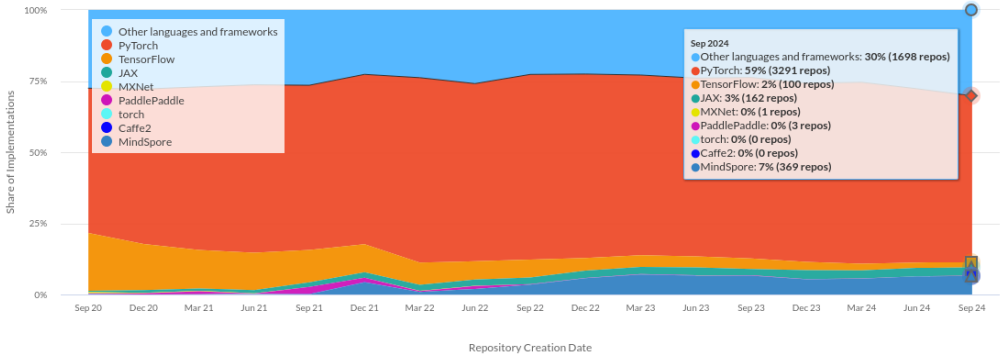
What is PyTorch?

Trends

Quarter | 2020-09-01 to 2024-09-30

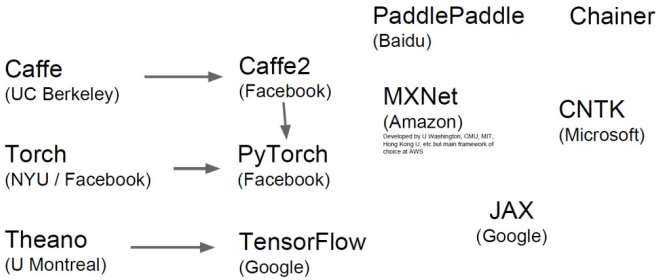
Frameworks

Paper Implementations grouped by framework



Various Framework

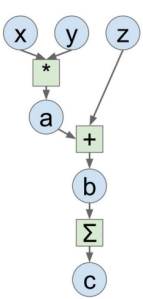
- Various Deep Learning Frameworks



- Focus on PyTorch in this session.

- Preview of Numpy & PyTorch & Tensorflow

Computation Graph



Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                   feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D),requires_grad=True)
y = torch.rand((N, D),requires_grad=True)
z = torch.rand((N, D),requires_grad=True)

a =x * y
b =a + z
c=torch.sum(b)

c.backward()
```

Outline

Installing PyTorch and Importing Torch

Step 1: Installing PyTorch

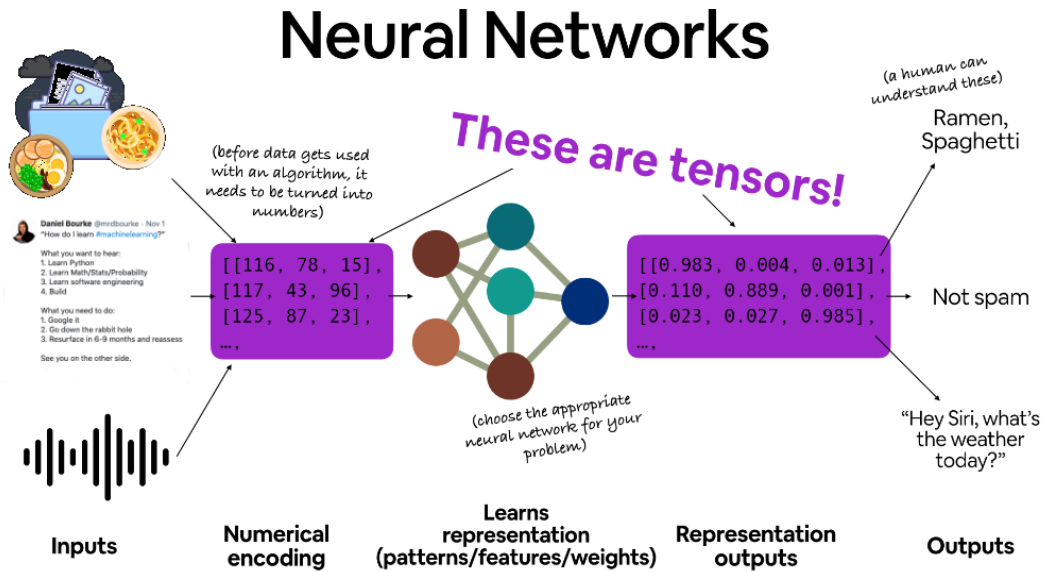
- Using pip:
  pip install torch torchvision torchaudio
- Using conda:
  conda install pytorch torchvision torchaudio -c pytorch

Step 2: Importing Torch

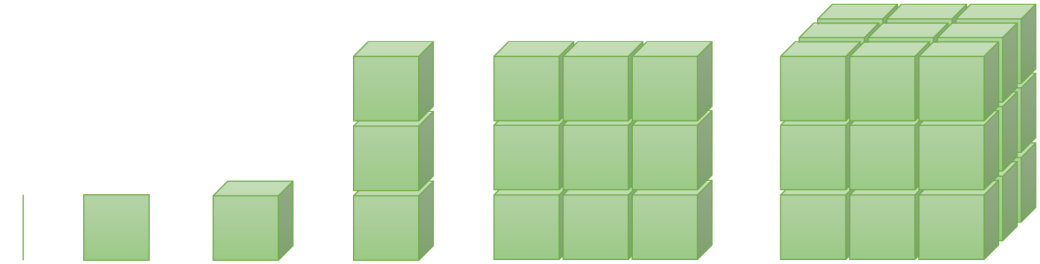
- Once installed, you can import PyTorch in Python with:
  import torch

Outline

What is a Tensor



Vector, Matrix and Tensor



PyTorch Tensor Shape Convention

- 2D Tensor (Typical Simple Setting)
  - $|t|$ = (batch size, dim)

PyTorch Tensor Shape Convention

- 3D Tensor (Typical Computer Vision)
  - $|t|$ = (batch size, height, width)

PyTorch Tensor Shape Convention

- 3D Tensor (Typical Natural Language Processing)
  - $|t|$ = (batch size, length, dim)

Outline

Initializing Tensors in PyTorch

- **Directly from Data**

```
import torch

# Initializing directly from Python list
data = [[1, 2], [3, 4]]
tensor_from_data = torch.tensor(data)
print(tensor_from_data)
```

- **From a NumPy Array**

```
import numpy as np

# Create a NumPy array
np_array = np.array([[5, 6], [7, 8]])
tensor_from_numpy = torch.from_numpy(np_array)
print(tensor_from_numpy)
```

Initializing Tensors in PyTorch

- **Random Initialization**

```
# Random tensor with specific size
random_tensor = torch.rand(3, 3)
print(random_tensor)
```

```
# Random tensor from a normal distribution N(0,1) with specific
    size
random_tensor = torch.randn(3, 3)
print(random_tensor)
```

```
# Random integers generated uniformly between low (inclusive)
   and high (exclusive). with specific size
random_tensor = torch.randint(low=0, high=20, (3, 3))
print(random_tensor)
```

Initializing Tensors in PyTorch

Tensors with Zeros and Ones

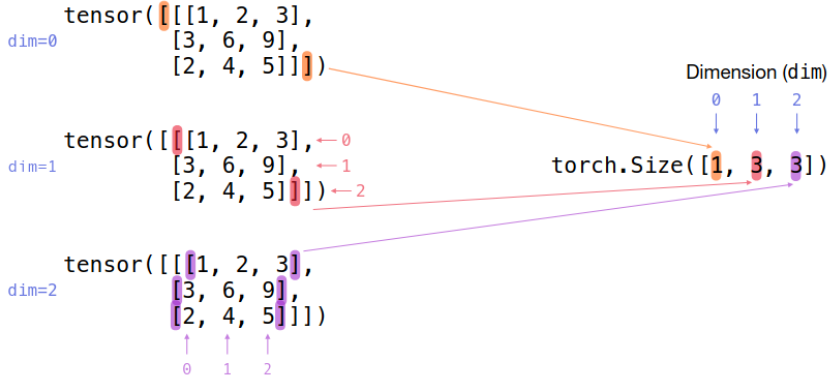
```
# Tensor initialized with zeros
zeros_tensor = torch.zeros(2, 2)

# Tensor initialized with ones
ones_tensor = torch.ones(2, 3)
```

Creating with the Same Shape and Dtype

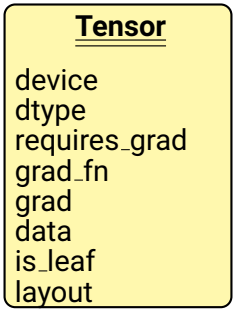
```
# New tensor with the same shape and data type as
   original_tensor
same_shape_tensor = torch.ones_like(original_tensor)

# New tensor filled with zeros, same shape and dtype
same_shape_zeros = torch.rand_like(original_tensor)
```

Tensor dimensions

```
import torch

lst = torch.as_tensor([[[1, 2, 3], [3, 6, 9], [2, 4, 5]]])
# Printing the size of the tensor => lst.shape == lst.size()
print("Tensor size:", lst.size()) => torch.Size([1, 3, 3])
```



PyTorch Tensor as Object



Printing All Attributes of a Tensor

Tensor Attributes in PyTorch

```
import torch

# Create a tensor with random values
lst = torch.rand(3, 7)

# Print all relevant attributes of the tensor
print("Tensor:", lst.data)
print("Shape:", lst.shape) => torch.Size([3, 7])
print("Data type:", lst.dtype) => torch.float32
print("Device:", lst.device) => device(type='cpu')
print("Requires grad:", lst.requires_grad) => False
print("Gradient generated the tensor:", lst.grad_fn) => None
print("The gradient corresponding:", lst.grad) => None
```

Operations on Tensors

- Point-wise operations

```
# Common arithmetic functions in PyTorch
import torch

# Addition
result_add = torch.add(x, y)  # or x + y

# Subtraction
result_sub = torch.sub(x, y)  # or x - y

# Multiplication
result_mul = torch.mul(x, y)  # or x * y

# Division
result_div = torch.div(x, y)  # or x / y
```

Operations on Tensors

```
# Power
result_pow = torch.pow(x, y)  # or x ** y

# Square root
result_sqrt = torch.sqrt(x)

# Exponential
result_exp = torch.exp(x)

# Logarithm
result_log = torch.log(x)
```

Operations on Tensors

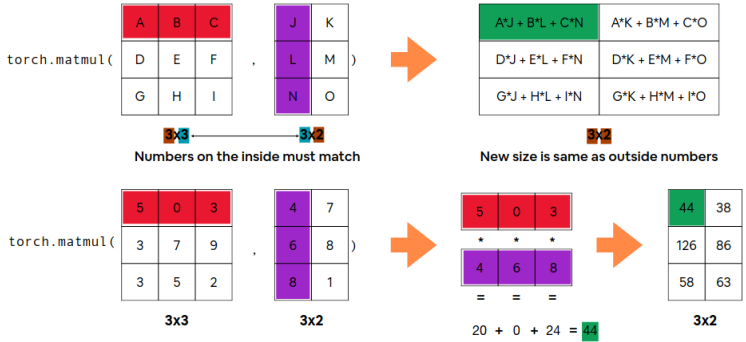
- Reduction operations

```
#Summation
result_sum = torch.sum(x)

#Mean
result_sum = torch.mean(x)

#std
result_sum = torch.std(x)

#var
result_sum = torch.var(x)
```

Operations on Tensors

- Matrix, vector multiplication

```
# Matrix multiplication
result_matmul = torch.matmul(x, y)  # or x @ y

#Batch matrix multiplication
result_matmul = torch.bmm(x, y)
```



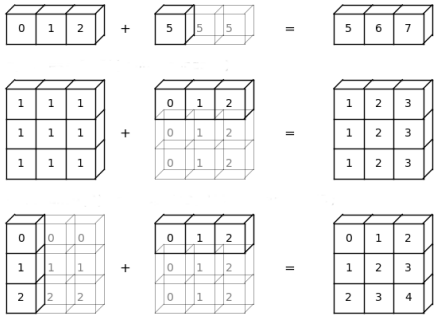
Operations on Tensors

- Comparison operation

```
#maximum
result_sum = torch.max(x)

#minimum
result_sum = torch.min(x)

#topk
result_sum = torch.topk(x, k=2)

#sort
result_sum = torch.sort(x)
```

Broadcasting in PyTorch

- **Broadcasting**: A way of performing operations on tensors of different shapes by automatically expanding dimensions to match each other.
- Two tensors are “broadcastable” if the following rules hold:
  - Each tensor has at least one dimension.
  - When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.



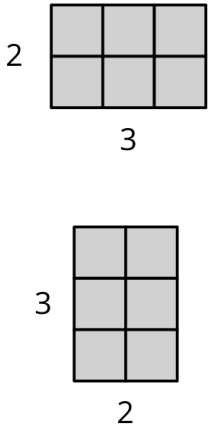
```
x=torch.empty(5,7,3)
y=torch.empty(5,7,3)
# same shapes are always broadcastable

x=torch.empty((0,))
y=torch.empty(2,2)
# x and y are not broadcastable, because x does not have at
    least 1 dimension

# can line up trailing dimensions
x=torch.empty(5,3,4,1)
y=torch.empty(  3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist
```

```
x=torch.empty(5,2,4,1)
y=torch.empty(  3,1,1)
# x and y are not broadcastable, because in the 3rd trailing
    dimension 2 != 3
```

Transpose

- Transpose: transpose two specified dimensions

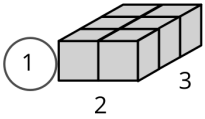
```
# torch.transpose(input, dim0, dim1)
x = torch.zeros([2, 3])
print(x.shape) # torch.Size([2, 3])
x = x.transpose(0, 1)
print(x.shape) #torch.Size([3, 2])
```

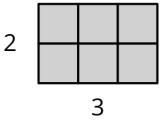


Squeeze

- Squeeze: remove the specified dimension with length = 1

```
>>> x = torch.zeros([1, 2, 3])
>>> x.shape
torch.Size([1, 2, 3])
>>> x = x.squeeze(0)
                    (dim = 0)
>>> x.shape
torch.Size([2, 3])
```



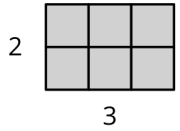


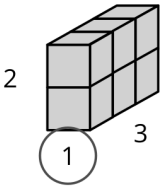
Unsqueeze

- Unsqueeze: expand a new dimension

```
>>> x = torch.zeros([2, 3])
>>> x.shape
torch.Size([2, 3])
>>> x = x.unsqueeze(1)
>>> x.shape
torch.Size([2, 1, 3])
```

(dim = 1)

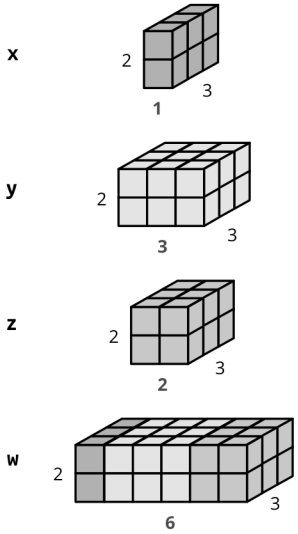




Cat

- Cat: concatenate multiple tensors

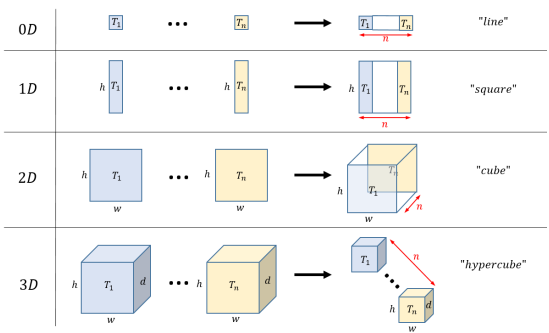
```
# torch.cat(tensors, dim=0)
x = torch.zeros([2, 1, 3])
y = torch.zeros([2, 3, 3])
z = torch.zeros([2, 2, 3])
w = torch.cat([x, y, z], dim=1)
#torch.Size([2, 6, 3])
print(w.shape)
```



Stack

- Stack: combine multiple tensors along a new dimension

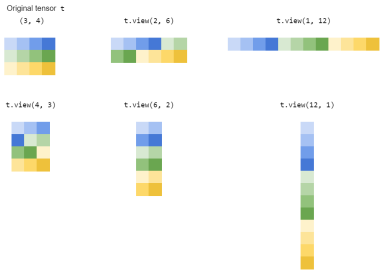
```
# torch.stack(tensors,
    dim=0)
x = torch.tensor([1, 2])
y = torch.tensor([3, 4])
z = torch.tensor([5, 6])
w = torch.stack([x, y, z
    ], dim=0)
# torch.Size([3, 2])
print(w.shape)

v = torch.stack([x, y, z
    ], dim=1)
# torch.Size([2, 3])
print(v.shape)
```



View and Reshape in PyTorch

- **View**: Returns a new tensor with the same data but a different shape. It requires the tensor to be contiguous in memory.
- **Reshape**: Similar to `view`, but can handle non-contiguous tensors.
- Use when you want to modify the dimensions of a tensor without changing its underlying data.

```
# Original tensor
x = torch.arange(12)
print(x.shape)  # torch.Size([12])

# Using view
y = x.view(3, 4)  # Reshape to 3x4

# Using reshape
z = x.reshape(2, 6)  # Reshape to 2x6
```



PyTorch Device

Overview of Device Assignment in PyTorch

In PyTorch, tensors can be placed on various devices, such as:

- **CPU (Central Processing Unit)**: Default device for tensor operations.
- **GPU (Graphics Processing Unit)**: Used for accelerating computations. Requires CUDA or other supported backends.
- **Other Devices**: Emerging support includes devices like TPUs or other accelerators (often requires specialized libraries).

Checking Device Availability

- `torch.cuda.is_available()` checks if a CUDA-capable GPU is available.
- `torch.device()` allows explicit specification of the device (e.g., `"cpu"`, `"cuda"`, `"cuda:1"`).

Outline

PyTorch Device

```
import torch

# Select device based on availability
device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")

# Create tensor directly on the device
tensor = torch.rand(3, 3, device=device)

# Alternative approach for general devices
other_device_tensor = tensor.to("cpu")  # Move tensor to CPU
```

- Using `"cuda:0"`, `"cuda:1"`, etc., specifies which GPU to use in multi-GPU systems.
- Future releases and libraries may support more specialized devices for further acceleration.

Outline

Pytorch Data types (must be float)

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	torch.float32 or torch.float	torch.FloatTensor	torch.cuda.FloatTensor
64-bit floating point	torch.float64 or torch.double	torch.DoubleTensor	torch.cuda.DoubleTensor
16-bit floating point	torch.float16 or torch.half	torch.HalfTensor	torch.cuda.HalfTensor
8-bit integer (unsigned)	torch.uint8	torch.ByteTensor	torch.cuda.ByteTensor
8-bit integer (signed)	torch.int8	torch.CharTensor	torch.cuda.CharTensor
16-bit integer (signed)	torch.int16 or torch.short	torch.ShortTensor	torch.cuda.ShortTensor
32-bit integer (signed)	torch.int32 or torch.int	torch.IntTensor	torch.cuda.IntTensor
64-bit integer (signed)	torch.int64 or torch.long	torch.LongTensor	torch.cuda.LongTensor
Boolean	torch.bool	torch.BoolTensor	torch.cuda.BoolTensor

Type Conversion in PyTorch

```
# Create a tensor with random float32 values
float_tensor = torch.rand(3, 3, dtype=torch.float32)

# Convert the tensor to int8
int8_tensor = float_tensor.to(torch.int8)
#Same Result
int8_tensor = float_tensor.char()
```

Outline

Understanding `requires_grad` in PyTorch

Overview of `requires_grad`

- PyTorch's `requires_grad` attribute determines if a tensor will have its gradients calculated during backpropagation.
- Tensors with `requires_grad=True` track operations for automatic differentiation, crucial for training models with gradient-based optimization.

Key Rules

- By default, tensors are created with `requires_grad=False`.
- When performing operations involving tensors with `requires_grad=True`, the result also has `requires_grad=True`.
- Gradients are accumulated in the `.grad` attribute after calling `.backward()` on the computational graph.

Understanding `requires_grad` in PyTorch

```
import torch

# Create a tensor with requires_grad=True
x = torch.tensor([[1., 0.], [-1., 1.]], requires_grad=True)
print("Tensor x:", x)

# Perform operations on the tensor
y = x.pow(2)
print("Tensor y:", y)
#torch.tensor([[1., 0.], [1., 1.]], grad_fn=<PowBackward0>)
y_sum = y.sum()  # Summarize for scalar output
y_sum.backward()  # Backpropagate to compute gradients
```

- `x` is a tensor with `requires_grad=True`, so PyTorch tracks operations on it.
- The result `y` automatically inherits `requires_grad=True`.
- Calling `y_sum.backward()` computes gradients for `x`, stored in `x.grad`.

Tensors – Gradient Calculation

1
```
>>> x = torch.tensor([[1., 0.], [-1., 1.]], requires_grad=True)
```
2
```
>>> z = x.pow(2).sum()
```
3
```
>>> z.backward()
```
4
```
>>> x.grad
tensor([[ 2.,  0.],
        [-2.,  2.]])
```

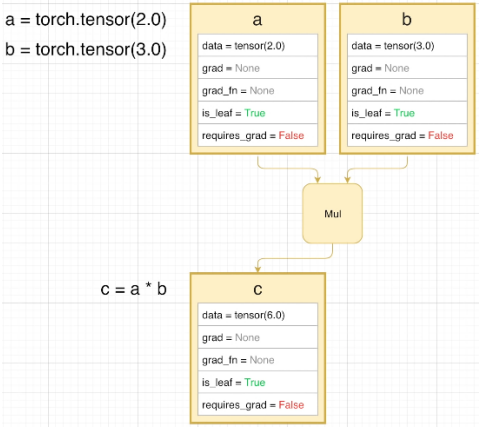
1
$$x = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

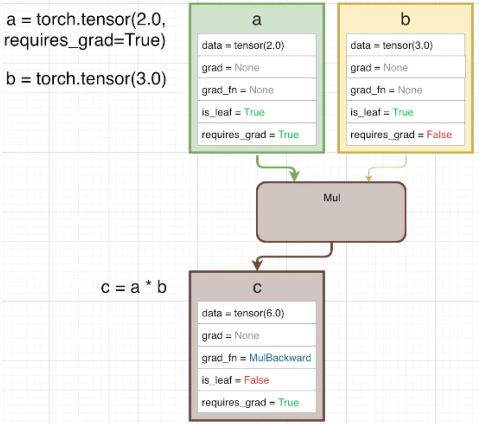
2
$$z = \sum_i \sum_j x_{i,j}^2$$

3
$$\frac{\partial z}{\partial x_{i,j}} = 2x_{i,j}$$

4
$$\frac{\partial z}{\partial x} = \begin{bmatrix} 2 & 0 \\ -2 & 2 \end{bmatrix}$$

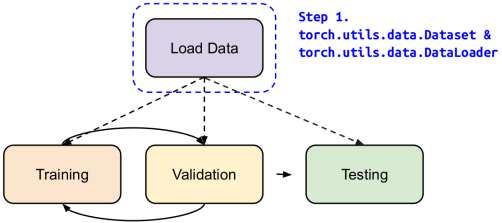
Tensors – Gradient Calculation

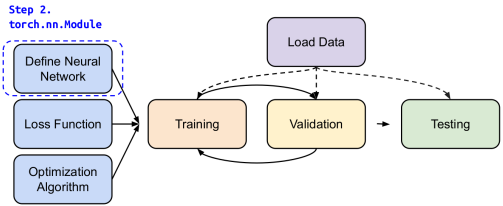


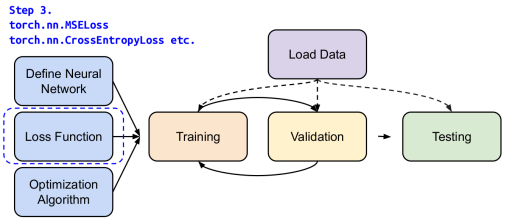


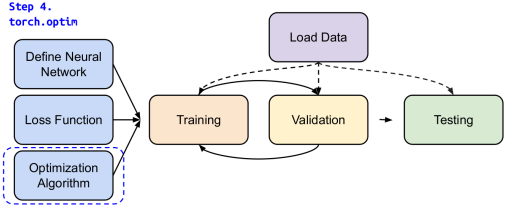
Training Step

Training Step

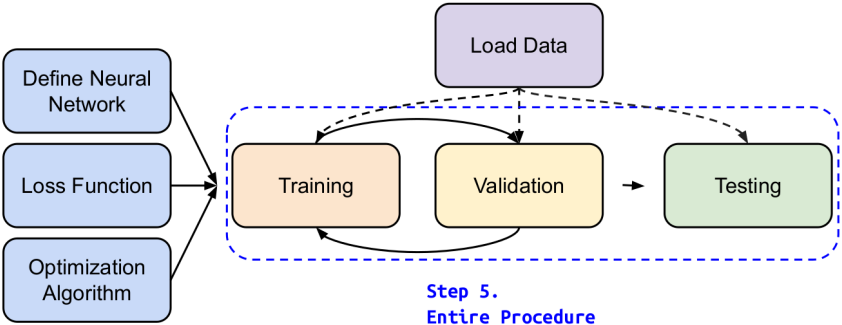






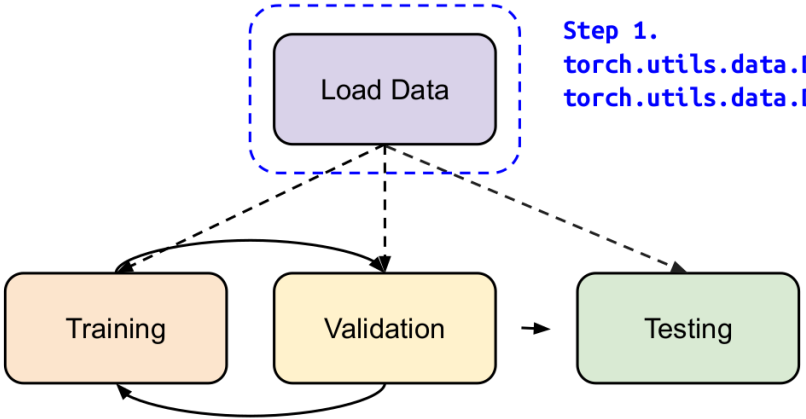


Training Step



Outline

Dataset & Dataloader



Dataset & Dataloader

- **Dataset**: stores data samples and expected values
- **Dataloader**: groups data in batches, enables multiprocessing

- **Dataset**: A class to represent a dataset. You can use built-in datasets like `torchvision.datasets` or create a custom dataset by subclassing `torch.utils.data.Dataset`.
- **DataLoader**: Wraps a dataset and provides an iterable for easy batching, shuffling, and loading data during training.

- dataset = MyDataset(file)
- dataloader = DataLoader(dataset, batch_size, shuffle=True)

Dataset & Dataloader

```
class MyDataset(Dataset):
    def __init__(self, file):
        self.data = ...

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return len(self.data)
```

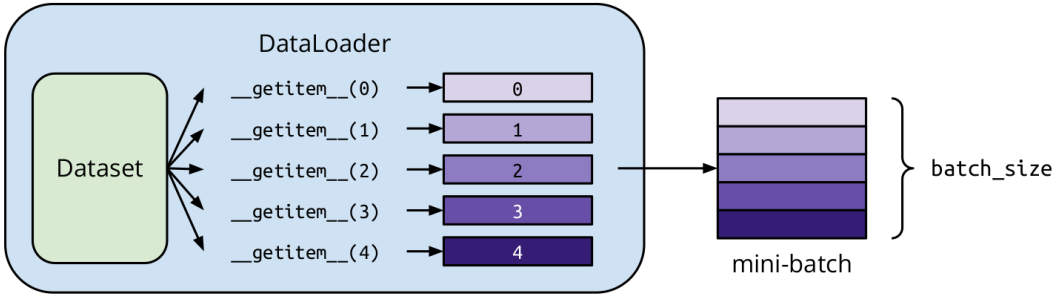
Read data & preprocess

Returns one sample at a time

Returns the size of the dataset

Dataset & Dataloader

```
dataset = MyDataset(file)

dataloader = DataLoader(dataset, batch_size=5, shuffle=False)
```

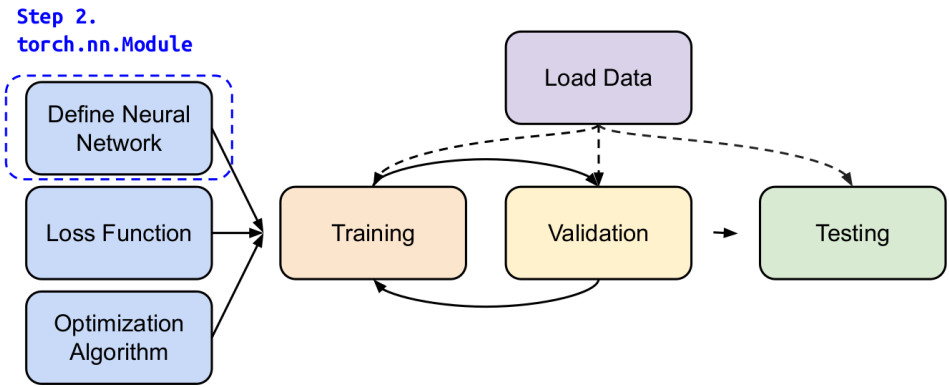


Creating a Custom Dataset for your files

```python
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

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

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        sample = {"image": image, "label": label}
        return sample
```

Outline

Neural Network



Neural Network

```
import torch
from torch import nn
import torch.nn.functional as F
```

The `torch.nn` Module in PyTorch: Purpose

The `torch.nn` module abstracts away many low-level details of neural network implementation, offering tools to:

- Define **layers** like fully connected (linear), convolutional, or recurrent layers.
- Apply **activation functions** such as ReLU, Sigmoid, or Softmax.
- Configure **loss functions** for training, like `MSELoss` or `CrossEntropyLoss`.
- Manage **weights** and **parameters** of a model for easy updates and optimizations.

The `torch.nn` Module in PyTorch: Key Components

a. Layers: The building blocks of a neural network, transforming inputs to outputs using weights and biases:

- `nn.Linear`: Fully connected layers that perform $y = xW^T + b$.
- `nn.Conv2d`: Processes spatial data (e.g., images) with kernel-based filters.
- `nn.LSTM`, `nn.GRU`: Handle sequential data like time series or text.

b. Activation Functions: Introduce non-linearity to model complex patterns:

- Examples: `nn.ReLU`, `nn.Sigmoid`, `nn.Softmax`.

c. Loss Functions: Evaluate how well the model's predictions align with targets:

- Examples: `nn.CrossEntropyLoss` (classification), `nn.MSELoss` (regression).

d. Container Modules: Organize and stack layers in a structured way:

- `nn.Sequential`: Defines a simple chain of layers.
- `nn.Module`: The base class for all models, allowing custom architectures.

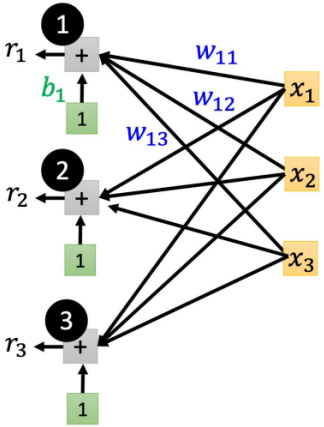
The `torch.nn` Module in PyTorch: Workflow

Define and train a model using `torch.nn`:

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc = nn.Linear(10, 5)  # Layer: input 10, output 5

    def forward(self, x):
        return self.fc(x)  # Forward pass

# Instantiate the model
model = MyModel()
```

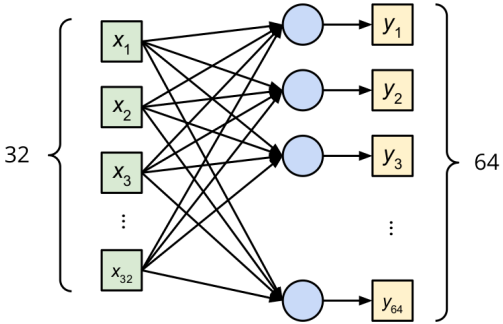
Linear Layer (Fully-connected Layer)

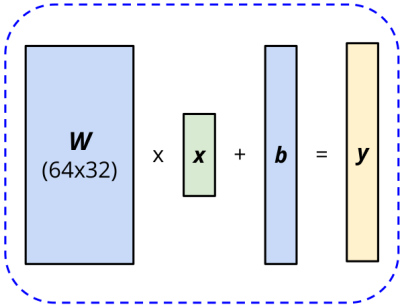
nn.Linear(in_features, out_features, bias=True)



$$\boldsymbol{b} + \boldsymbol{W}\,\boldsymbol{x}$$

Linear Layer (Fully-connected Layer)

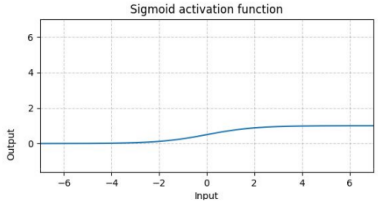




Non-Linear Activation Functions

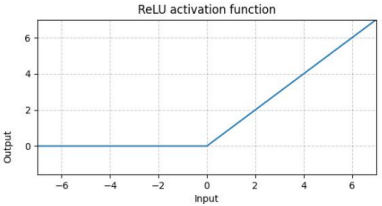
- Sigmoid Activation

```
nn.Sigmoid()
```



- ReLU Activation

```
nn.ReLU()
```



Create a Custom Neural Network

```
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.net(x)
```

Initialize your model & define layers

Compute output of your NN

Create a Custom Neural Network

```python
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(10, 32),
            nn.Sigmoid(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.net(x)
```

=

```python
import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer1 = nn.Linear(10, 32)
        self.layer2 = nn.Sigmoid(),
        self.layer3 = nn.Linear(32,1)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        return out
```

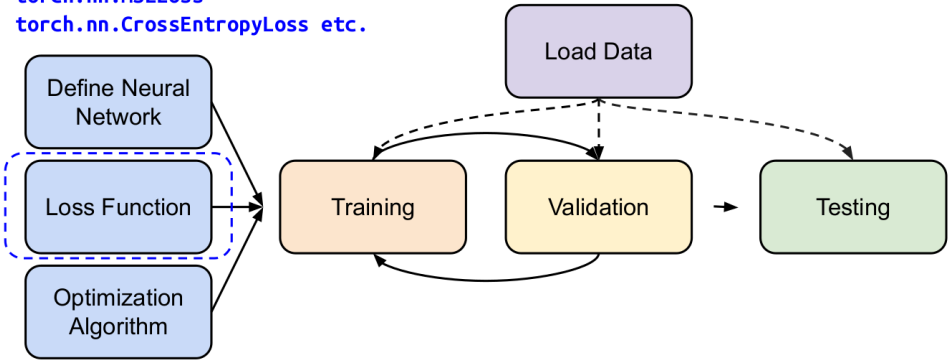
Create Model

```python
import torch

# Define the device (GPU if available, else CPU)
device = "cuda" if torch.cuda.is_available() else "cpu"

# Create an instance of MyModel and move it to the device
model = MyModel().to(device)

print(f"Model is running on: {device}")
```

Outline

Loss

Step 3.
torch.nn.MSELoss
torch.nn.CrossEntropyLoss etc.



What is a Loss Function?

A loss function measures prediction error; `torch.nn` provides options for various tasks.

General Usage: The loss is calculated by comparing the model's predictions (`model_output`) with the actual target values (`expected_value`).

```
import torch
import torch.nn as nn

model_output = torch.tensor([2.5, 3.0, 4.5])  # Predicted
    values
expected_value = torch.tensor([3.0, 3.0, 4.0])  # True values

# Select a loss function
criterion = nn.MSELoss()  # Example: MSE Loss

# Compute the loss
loss = criterion(model_output, expected_value)
```

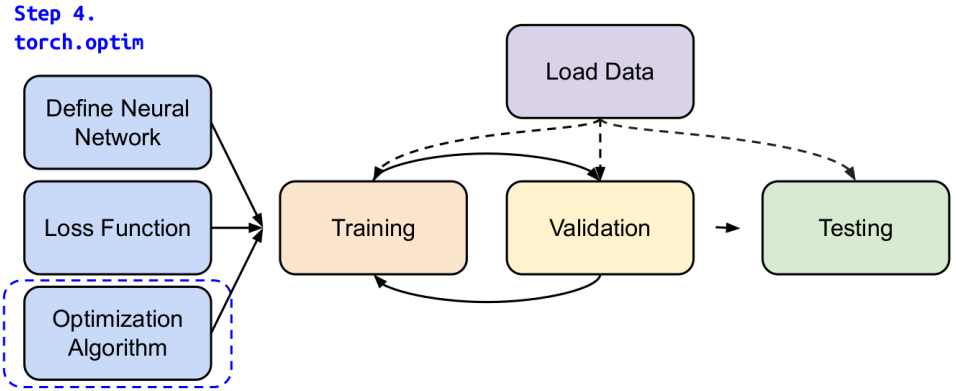
Common Loss Functions in PyTorch:

Here are some of the most commonly used loss functions in PyTorch, based on the task:

- **Mean Squared Error (MSE Loss)**: For regression tasks, minimizes the squared differences between predicted and true values. *criterion = nn.MSELoss()*
- **Cross Entropy Loss**: For classification tasks, compares predicted probabilities (logits) with true class labels. *criterion = nn.CrossEntropyLoss()*
- **Binary Cross Entropy (BCE Loss)**: For binary classification tasks, measures the difference between predicted probabilities and actual labels. *criterion = nn.BCELoss()*
- **Negative Log-Likelihood (NLL Loss):** Commonly used for classification tasks with pre-applied log-softmax. *criterion = nn.NLLLoss()*
- **Huber Loss**: Combines MSE and MAE, used when you want to be robust to outliers. *criterion = nn.HuberLoss()*

Outline

Optimizers


Step 4.
torch.optim
Define Neural Network
Loss Function
Optimization Algorithm
Load Data
Training
Validation
Testing

What is an Optimizer?

An optimizer adjusts model parameters to minimize the loss function by updating weights based on gradients computed during backpropagation. **General Usage:**

The optimizer is initialized and used in the training loop to update parameters and minimize the loss.

What is an Optimizer?

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

model = MyModel() # Define a simple model
criterion = nn.MSELoss()#Define Loss

# Define the optimizer: Stochastic Gradient Descent (SGD)
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training loop example
for data, target in dataloader:
    optimizer.zero_grad()  # Reset gradients
    output = model(data)   # Get predictions
    loss = criterion(output, target)  # Compute loss
    loss.backward()        # Backpropagation
    optimizer.step()       # Update parameters
```

Common Optimizers in PyTorch:

Here are some of the most commonly used optimizers in PyTorch:

- **Stochastic Gradient Descent (SGD)**: A basic optimizer using gradients from mini-batches of data. *optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)*
- **Adam Optimizer**: An adaptive learning rate method combining the benefits of both SGD and RMSProp. *optimizer = optim.Adam(model.parameters(), lr=0.001)*
- **Adagrad Optimizer**: Adjusts the learning rate for each parameter based on historical gradients. *optimizer = optim.Adagrad(model.parameters(), lr=0.01)*
- **RMSprop Optimizer**: Divides the learning rate by an exponentially decaying average of squared gradients. *optimizer = optim.RMSprop(model.parameters(), lr=0.01)*
- **Adadelta Optimizer**: Restricts the window of accumulated gradients, improving performance. *optimizer = optim.Adadelta(model.parameters(), lr=1.0)*

Loss in a Training Loop

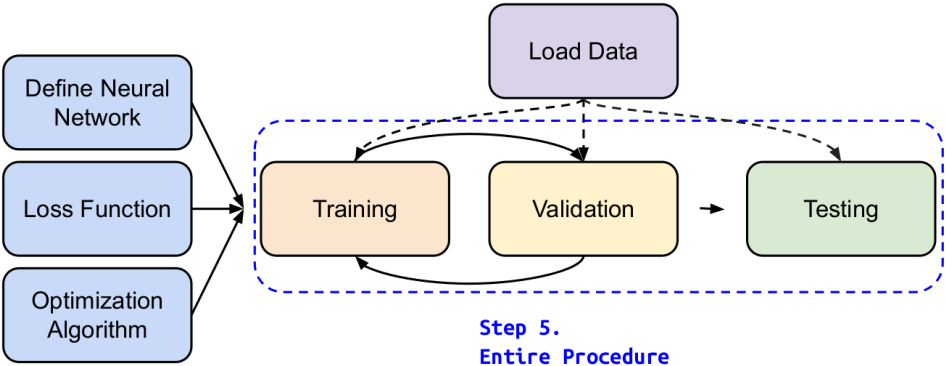
```
for data, target in dataloader:
    optimizer.zero_grad()  # Reset gradients
    output = model(data)   # Get predictions
    loss = criterion(output, target)  # Compute loss
    loss.backward()        # Backpropagation
    optimizer.step()       # Update parameters
```

For every batch of data:

1. Call **optimizer.zero_grad()** to reset gradients of model parameters.
2. Call **loss.backward()** to backpropagate gradients of prediction loss.
3. Call **optimizer.step()** to adjust model parameters.

Outline

Entire Training



Training Setup

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

# Define device (CPU or CUDA)
device = "cuda" if torch.cuda.is_available() else "cpu"

# Example dataset and model (already defined elsewhere)
dataset = MyDataset()  # Assume MyDataset is already defined
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

model = MyModel().to(device)  # Assume MyModel is already
    defined
criterion = nn.MSELoss()  # Loss function
optimizer = optim.SGD(model.parameters(), lr=0.01)  # Optimizer
```

Training Loop

```
# Put the model in training mode
model.train()

# Training loop
for epoch in range(10):  # 10 epochs
    for data, target in dataloader:
        data, target = data.to(device), target.to(device)  #
            Move data to device
        optimizer.zero_grad()  # Reset gradients
        output = model(data)   # Forward pass
        loss = criterion(output, target)  # Compute loss
        loss.backward()        # Backpropagation
        optimizer.step()       # Update parameters
    print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

Validation Loop

```
# Validation phase
model.eval()  # Set model to evaluation mode

val_loss = 0
with torch.no_grad():  # No need to track gradients during
    validation
     for data, target in val_dataloader:
         data, target = data.to(device), target.to(device)  #
             Move data to device
         output = model(data)  # Forward pass
         loss = criterion(output, target)  # Compute loss
         val_loss += loss.item()  # Accumulate loss for the
             validation set

print(f"Epoch {epoch+1} Validation Loss: {val_loss / len(
    val_dataloader)}")
```

- **model.eval()**
  - Changes the behavior of certain layers (e.g., dropout, batch normalization).
  - Ensures dropout is turned off and batch normalization uses accumulated statistics during evaluation.

- **with** `torch.no_grad()`
  - Prevents the computation of gradients during the forward pass.
  - Usually used during validation/testing to avoid unnecessary memory usage and to prevent accidental training on validation/testing data.

Saving and Loading Trained Models

Saving a Model:

- In PyTorch, you can save the model's state_dict (parameters) or the entire model.
- It's recommended to save the state_dict for better flexibility and portability.

```
torch.save(model.state_dict(), 'model.pth')  # Save model
    parameters
```

Loading a Model:

- To load the model, you need to initialize the model structure first and then load the state_dict.

```
model = MyModel()  # Initialize model structure
model.load_state_dict(torch.load('model.pth'))  # Load saved
    parameters
```

Getting Models from PyTorch Hub and Other Sources

PyTorch Hub: PyTorch Hub provides pre-trained models that can be used directly for inference or fine-tuning. It allows easy access to many popular models like ResNet, BERT, etc.

Example - Loading a Model from PyTorch Hub:

```
import torch

# Load a pre-trained model from PyTorch Hub (e.g., ResNet50)
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet50',
    pretrained=True)

# Set model to evaluation mode
model.eval()
```

Other Sources: You can also load models from other sources like GitHub or custom URLs. Simply follow the same principles as loading from local files, and ensure to load the correct model architecture and weights.

Using Custom URLs:

```
model = torch.load('https://your_model_url.com/model.pth')  # Load from a URL
model.eval()
```

Important Notes:

- Always ensure to call `model.eval()` after loading a pre-trained model for correct evaluation behavior.
- You may fine-tune these models for specific tasks by adjusting their final layers.

Merci
Pour votre attention

Des Questions ?

youva.addad@unicaen.fr