PyTorch with Notes and Examples

## torch

The main PyTorch library for tensor operations and deep learning.

Example:

import torch

## torch.nn

Module for building neural network layers, loss functions, etc.

Example:

import torch.nn as nn

https://pytorch.org/docs/stable/nn.html

## torch.nn.functional

Provides functional versions of layers (like softmax).

Example:

import torch.nn.functional as F

Some of the Functions and their descriptions :

| Function | Description | Example |
| --- | --- | --- |
| F.relu() | Rectified Linear Unit activation | F.relu(x) |
| F.sigmoid() | Sigmoid activation | F.sigmoid(x) |
| F.tanh() | Hyperbolic tangent activation | F.tanh(x) |
| F.cross\_entropy() | Combines log\_softmax + NLLLoss for classification | F.cross\_entropy(logits, targets) |
| F.linear() | Applies a linear transformation | F.linear(x, weight, bias) |
| F.dropout() | Applies dropout for regularization | F.dropout(x, p=0.5, training=True) |
| F.one\_hot() | Converts indices to one-hot vectors | F.one\_hot(torch.tensor([1, 2]), num\_classes=4) |
| F.log\_softmax() | Log version of softmax (useful for stability) | F.log\_softmax(x, dim=1) |

## torch.tensor

Creates a tensor (multi-dimensional array) in PyTorch.

Example:

x = torch.tensor([1.0, 2.0, 3.0])

## torch.device

Specifies the device to use (CPU or GPU).

Example:

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

## torch.no\_grad()

Disables gradient tracking — used during inference to save memory and speed up.

Example:

with torch.no\_grad():  
 output = model(input)  
  
# Why use it?  
# - Saves memory  
# - Speeds up inference  
# - No need to compute gradients during text generation

## torch.optim.Adam

An optimizer that adjusts model weights using gradients.

Adam stands for **Adaptive Moment Estimation**.  
It’s one of the most popular optimizers used to train deep learning models.

Example:

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

## What Does It Do?

When you train a neural network:

* The model makes predictions
* You compute the **loss** (error)
* You use .backward() to compute gradients of the loss w.r.t (“calculate how much the loss changes **with respect to each weight or bias in the model**.”) each parameter
* Then... optimizer.step() applies those gradients to **update the weights**

Adam manages how those gradients are applied using smart momentum and adaptive learning rate strategies.

## torch.nn.Embedding

Maps indices ( in our Case characters) to dense vector representations.

Example:

### Imagine:

You have **100 different items** (like characters or words), and you want to give each one a **vector of 64 numbers.**

embedding = nn.Embedding(num\_embeddings=100, embedding\_dim=64)  
output = embedding(torch.tensor([5])) #Get the embedding for item #5

## torch.nn.LSTM

LSTM stands for **Long Short-Term Memory**.

An LSTM layer for learning from sequences with memory.

It’s a special kind of **Recurrent Neural Network (RNN)** that’s good at:

* Understanding sequences (like text, speech, time-series)
* Remembering important information from earlier in the sequence
* Forgetting what’s not important

Example:

lstm = nn.LSTM(input\_size=64, hidden\_size=128, batch\_first=True)  
output, (h, c) = lstm(torch.randn(2, 10, 64))

What each parameter means:

input\_size=64 → Each item in the sequence has 64 features (e.g., word embeddings)

hidden\_size=128 → The LSTM has 128 hidden neurons (i.e., memory cells)

batch\_first=True → Input and output tensors will be shaped like (batch, sequence, features)

It **remembers** long-term patterns like:

* Grammar in sentences
* Repeated patterns in time series
* Musical notes or rhythm

Unlike simple RNNs, LSTMs are **better at handling long sequences** and **avoiding vanishing gradients** during training.

### Real Example Use Case:

You're generating text one character at a time.

* LSTM takes the previous characters and **remembers context**
* It uses that to guess the next most likely character

### Summary Table

| Term | Meaning |
| --- | --- |
| input\_size | Features per time step (e.g., embedding size) |
| hidden\_size | Memory capacity of the LSTM |
| batch\_first=True | Input/output has shape (batch, sequence, features) |
| output | Hidden states at each time step |
| (h\_n, c\_n) | Final hidden and cell states |

## torch.nn.Linear

A fully connected layer to map from hidden features to output size.

Example:

linear = nn.Linear(128, 10)  
output = linear(torch.randn(1, 128))

nn.Linear creates a **fully connected layer** (also called a **dense layer**) in a neural network.

It does a simple math operation:

output=input×weightsT+bias

It maps input features to output features using **matrix multiplication**.

### What it does:

* nn.Linear(128, 10):
  + Takes an input of size **128**
  + Outputs a vector of size **10**
  + Think of it like:

"Take 128 features and turn them into a score for 10 possible classes."

* torch.randn(1, 128):
  + 1 sample, with 128 input features
* output:
  + Will be a tensor of shape (1, 10) → 10 output values

| Parameter | Meaning |
| --- | --- |
| 128 | Input feature size |
| 10 | Output feature size (e.g., number of classes) |
| linear(...) | Applies the transformation |
| Output shape | (batch\_size, 10) |

## torch.nn.CrossEntropyLoss

A loss function used for classification tasks.

Example:

criterion = nn.CrossEntropyLoss()  
loss = criterion(predictions, targets)

## Why CrossEntropyLoss is perfect for text prediction

| Task | CrossEntropy Use |
| --- | --- |
| Character prediction | Predict next character from char vocabulary |
| Word prediction | Predict next word from word vocabulary |
| Token prediction (LLM) | Predict next token from token vocabulary (e.g., BPE) |
| Model output | Raw logits for each class (char/word/token) |
| Loss compares | Prediction vs actual next token |

## model.train()

Sets the model to training mode (enables dropout, batchnorm, etc.).

Example:

model.train()

## What does model.train() do in PyTorch?

model.train()

It **activates** special behaviors in certain layers — mainly:

* **Dropout**
* **Batch Normalization**

## what are those exactly?

### 1. ****Dropout****

* Randomly **“drops out” (disables)** some neurons during training
* Prevents the model from becoming **too reliant on any one neuron**
* Helps avoid **overfitting**

Example:

nn.Dropout(p=0.5)

* During training: 50% of neurons are randomly dropped each time
* During eval: no neurons are dropped — all are used

### ****2. Batch Normalization****

* Normalizes (rescales) the activations in each mini-batch
* Makes training faster and more stable
* **Behavior changes** in training vs eval:
  + **Training:** uses **batch stats** (mean & variance)
  + **Eval:** uses **running stats** (global averages)

Example:

nn.BatchNorm1d(128)

===============================

import torch.nn as nn

model = nn.Sequential(

nn.Linear(10, 20),

nn.ReLU(),

nn.Dropout(p=0.5), # Dropout layer

nn.BatchNorm1d(20) # Batch Normalization layer

)

model.train() # Turn ON dropout + batchnorm training behavior

## model.eval()

Sets the model to evaluation mode (disables dropout, batchnorm).

Example:

model.eval()

## Real-World Analogy

Your model is like a student:

* **Training mode (**train()**)**:
  + Doing practice quizzes with distractions (dropout)
  + Adjusting techniques based on how things are going (batchnorm uses current batch)
* **Evaluation mode (**eval()**)**:
  + Taking the final exam — no distractions, serious mode
  + Using knowledge accumulated over all practice sessions (batchnorm uses saved stats)

## model.parameters()

Returns all learnable parameters of the model.

Example:

optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

lr stands for **learning rate** — it's a **hyperparameter** that controls **how big a step** the model takes when updating its weights.

## How it works:

During training, the model computes gradients (slopes), and the optimizer adjusts the weights like this:

new\_weight=old\_weight−lr×gradient

So:

* A **higher** lr makes **bigger jumps** in the weight space
* A **lower** lr makes **smaller, careful steps**

| Parameter | Value | Meaning |
| --- | --- | --- |
| lr=0.001 | 0.001 | Learning rate for weight updates |
| Too high | > 0.01 | Unstable training |
| Too low | < 0.00001 | Very slow learning |
| Good start | 0.001 | Works well for Adam in many cases |

## loss.backward()

Computes gradients of the loss with respect to parameters.

Example:

loss.backward()

Go backward from the loss and **compute the gradient** of the loss **with respect to each model parameter**.

These gradients are then used to **update the model’s weights** in the next step.

## How it works:

1. **Forward Pass** → compute predictions
2. **Loss Calculation** → how wrong was the prediction?
3. loss.backward() → computes gradients (derivatives) of the loss w.r.t each weight
4. optimizer.step() → adjusts weights using those gradients

## optimizer.step()

Applies gradient updates to the model parameters.

Example:

optimizer.step()

### Analogy:

You're hiking down a mountain in the dark:

* loss.backward() tells you the **steepest direction downhill**
* optimizer.step() is when you **actually walk in that direction**

for input, target in dataset:

optimizer.zero\_grad() # clear old gradients

output = model(input) # forward pass

loss = criterion(output, target) # compute loss

loss.backward() # compute gradients

optimizer.step() # update weights

## optimizer.zero\_grad()

Resets gradients before the next backward pass.

Example:

optimizer.zero\_grad()

This **clears the old gradients** from the **previous training step**.

In PyTorch, **gradients accumulate** by default — so if you don’t reset them, they’ll **add up** and mess up your weight updates.

## clip\_grad\_norm\_

Prevents exploding gradients by limiting their norm.

Example:

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1.0)

This function **limits ("clips") the size of the gradients** before you update the weights.

It prevents the **gradients from becoming too large**, which can make your model unstable or even crash during training.

## F.softmax

Applies softmax to a tensor to obtain probabilities.

Example:

probs = F.softmax(logits, dim=1)

## Analogy

Imagine your model gives scores for each class like **votes**.

Softmax turns the votes into **percentages**, making them easier to interpret and use.

## to(device)

Moves model or tensor to CPU or GPU.

Example:

model.to(device)  
tensor.to(device)

## torch.save

Saves the model's learned parameters to a file.

Example:

torch.save(model.state\_dict(), 'model.pt')

## torch.load

Loads model parameters from a file.

Example:

model.load\_state\_dict(torch.load('model.pt'))

## argparse

Library for parsing command-line arguments.

Example:

import argparse  
parser = argparse.ArgumentParser()  
args = parser.parse\_args()

what a **model** and **weights** are, with a real-world example.

## Imagine You buidling a Pizza Delivery Robot

You’re building a robot that can guess:

“How long will it take to deliver this pizza?”

It depends on:

* Distance to customer (in km)
* Time of day (rush hour vs not)

## You Build a Model

A **model** is just a formula that takes **inputs** and gives an **output**.

You start with something simple:

delivery\_time = (distance × w1) + (time\_of\_day × w2) + b

* distance and time\_of\_day are your **inputs**
* w1 and w2 are your **weights**
* b is your **bias (adjustment)**
* delivery\_time is the **predicted output**

## What are ****weights****?

* Think of weights like **importance sliders**:
  + How much should “distance” affect delivery time?
  + How much should “rush hour” affect it?

They start off **random**, but your robot:

* Makes a guess
* Sees how wrong it was (loss)
* Adjusts the weights to be a little better next time

This is how it **learns**.

## In PyTorch:

import torch.nn as nn

model = nn.Linear(2, 1)

This means:

* 2 inputs (distance, time of day)
* 1 output (predicted delivery time)

It **automatically creates**:

* 2 weights (w1, w2)
* 1 bias (b)

And trains them using **gradient descent**.

## A Mini Example in Code:

import torch

import torch.nn as nn

# Our "robot"

model = nn.Linear(2, 1)

# Example input: distance=3km, time\_of\_day=1 (rush hour)

inputs = torch.tensor([[3.0, 1.0]])

# Make a prediction

output = model(inputs)

print("Predicted delivery time:", output.item())

You can then train it with many real examples and it will adjust the weights to predict better.

## Final Takeaway

| Concept | Meaning |
| --- | --- |
| **Model** | A smart formula that gives you predictions |
| **Weight** | How important each input is |
| **Bias** | A baseline adjustment |
| **Training** | Teaching the model by adjusting weights |

Resources :

* https://pytorch.org/tutorials/intermediate/char\_rnn\_generation\_tutorial.html
* https://pytorch.org/tutorials/
* https://www.geeksforgeeks.org/implementing-recurrent-neural-networks-in-pytorch/
* https://pytorch.org/tutorials/intermediate/char\_rnn\_classification\_tutorial