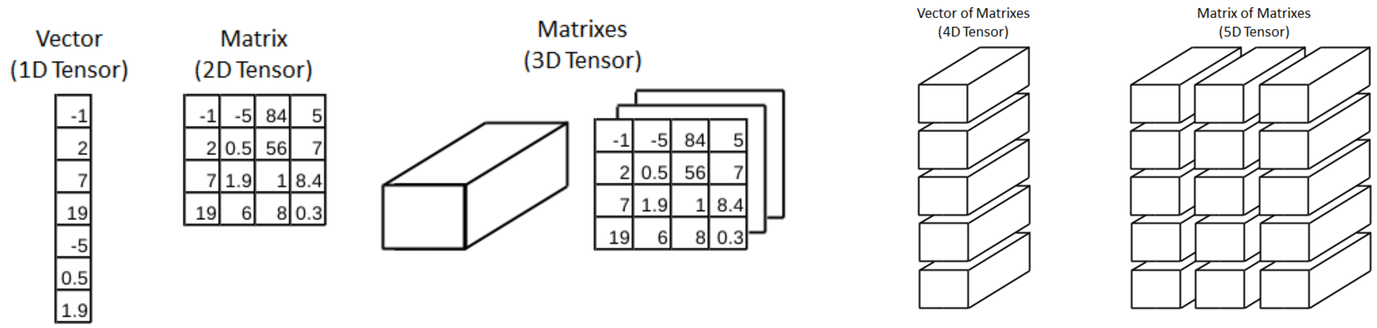
Introduction to Deep Learning
Recitation 0.4

Manigandan Ramadasan

Introduction

What is PyTorch?

- Open-source deep learning framework
- *Tensor* library with CPU and GPU support



Why do we need PyTorch?

- Rich Ecosystem
- Used widely in industries and research
- Makes life easier!
  - Has python APIs for building models, automatic differentiation etc.

TorchText

TorchAudio

PyG

TorchRL

TorchGeo

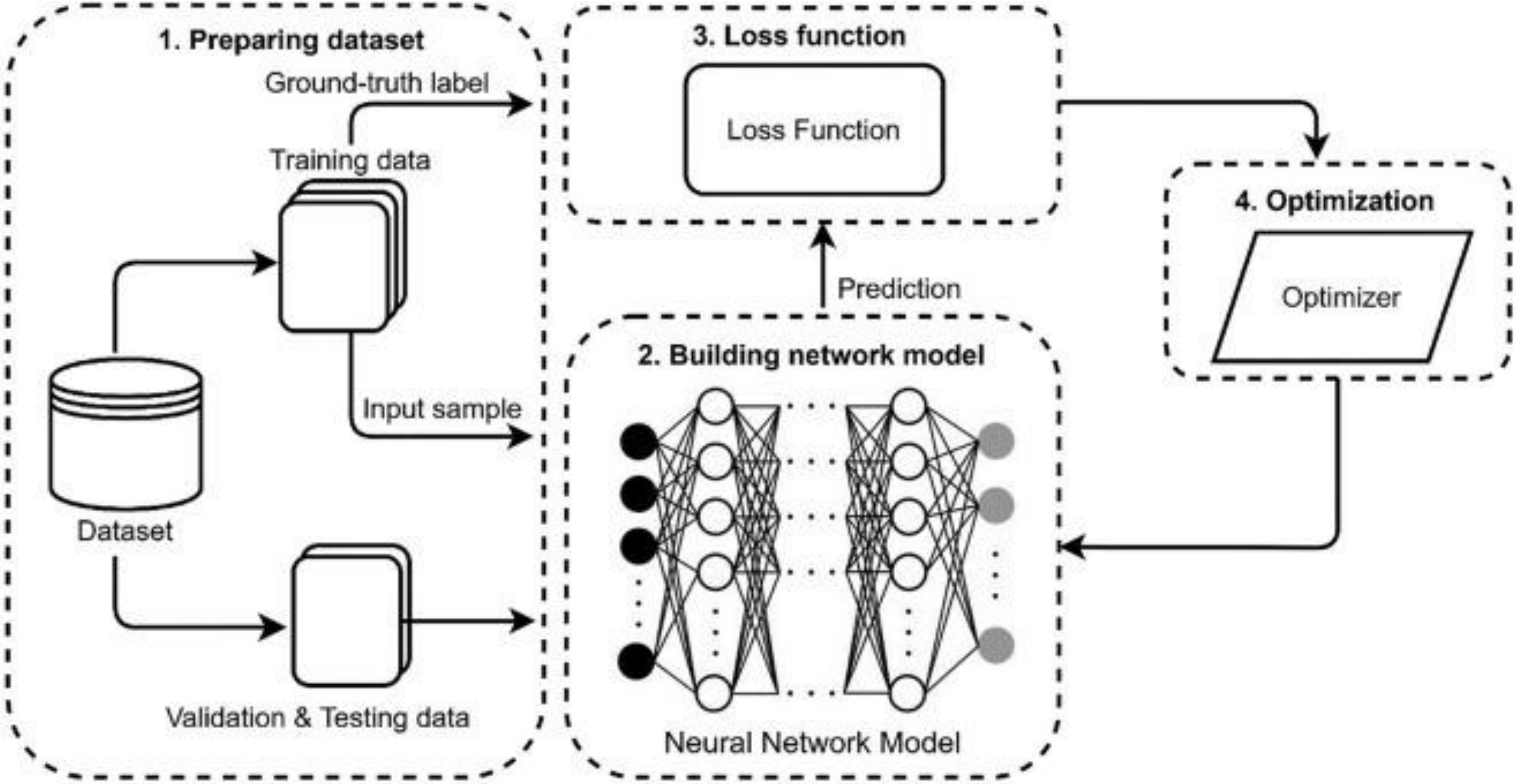
TorchRec

torchvision

Check out the following URL:
https://landscape.pytorch.org/

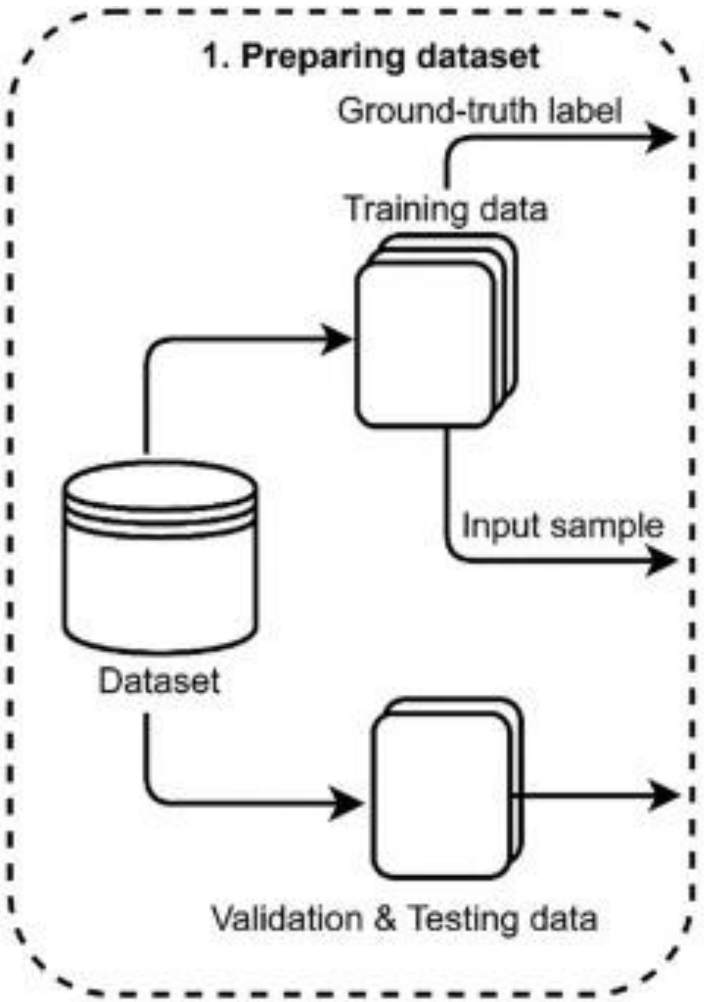
Tensor Image from [3]

A Typical Model Training Process



Training Process Image from [4]

(1) Preparing Dataset



(More about this in future Recitation 0)

torch.utils.data

Created On: Jun 13, 2025 | Last Updated On: Jun 13, 2025

At the heart of PyTorch data loading utility is the `torch.utils.data.DataLoader` class. It represents a Python iterable over a dataset, with support for

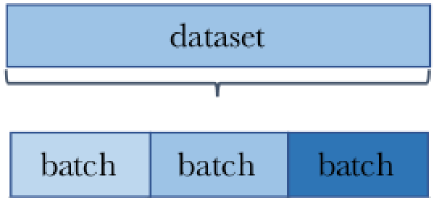
class **torch.utils.data.Dataset** [source]

An abstract class representing a `Dataset`.

All datasets that represent a map from keys to data samples should subclass it. All subclasses should overwrite `__getitem__()`, supporting fetching a data sample for a given key. Subclasses could also optionally overwrite `__len__()`, which is expected to return the size of the dataset by many `Sampler` implementations and the default options of `DataLoader`. Subclasses could also optionally implement `__getitems__()`, for speedup batched samples loading. This method accepts list of indices of samples of batch and returns list of samples.

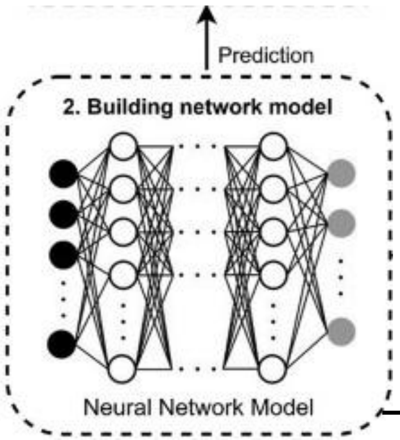
class **torch.utils.data.DataLoader**(*dataset*, *batch_size=1*, *shuffle=None*, *sampler=None*, *batch_sampler=None*, *num_workers=0*, *collate_fn=None*, *pin_memory=False*, *drop_last=False*, *timeout=0*, *worker_init_fn=None*, *multiprocessing_context=None*, *generator=None*, ***, *prefetch_factor=None*, *persistent_workers=False*, *pin_memory_device=''*, *in_order=True*) [source]

Data loader combines a dataset and a sampler, and provides an iterable over the given dataset.



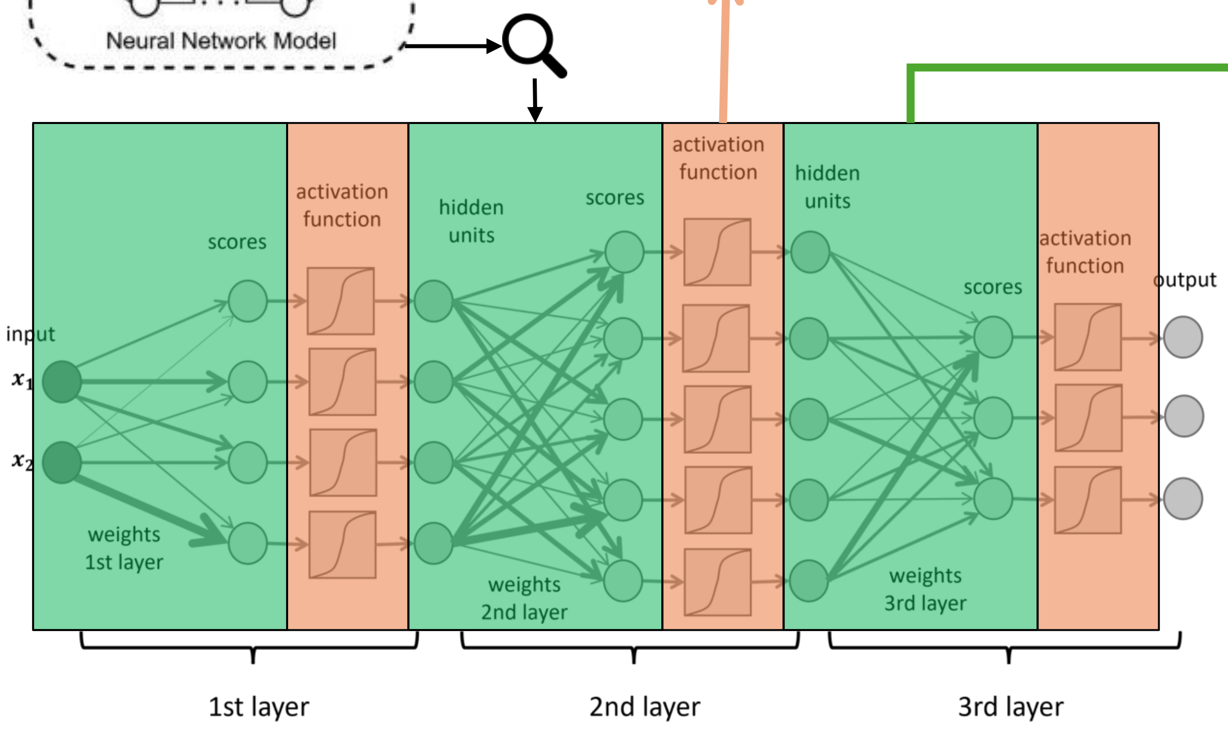
Batch Image from [5]

(2) Building the Neural Network



Non-linear Activations (weighted sum, nonlinearity)

`nn.ELU`	Applies the Exponential Linear Unit (ELU) function, element-wise.
`nn.Hardshrink`	Applies the Hard Shrinkage (Hardshrink) function element-wise.
`nn.Hardsigmoid`	Applies the Hardsigmoid function element-wise.
`nn.Hardtanh`	Applies the HardTanh function element-wise.
`nn.Hardswish`	Applies the Hardswish function, element-wise.
`nn.LeakyReLU`	Applies the LeakyReLU function element-wise.
`nn.LogSigmoid`	Applies the Logsigmoid function element-wise.



Neural Network Image from [6]

Build neural networks by stacking layers!
(and train their connection weights from data)

For more details, refer to:
https://docs.pytorch.org/docs/stable/nn.html

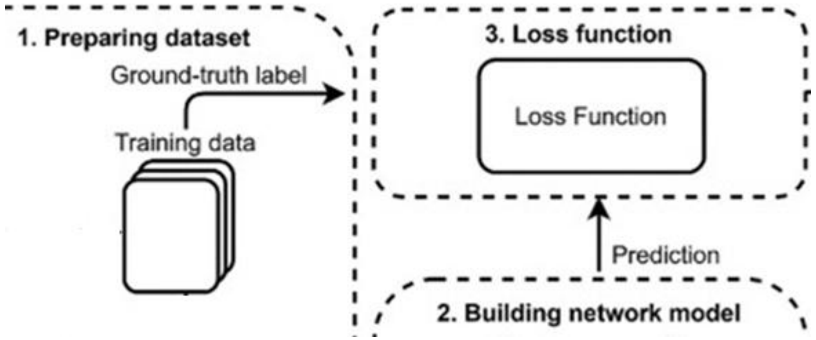
Linear Layers

`nn.Identity`	A placeholder identity operator that is argument-insensitive.
`nn.Linear`	Applies an affine linear transformation to the incoming data: $y = xA^T + b$.
`nn.Bilinear`	Applies a bilinear transformation to the incoming data: $y = x_1^T A x_2 + b$.
`nn.LazyLinear`	A `torch.nn.Linear` module where *in_features* is inferred.

Convolution Layers

`nn.Conv1d`	Applies a 1D convolution over an input signal composed of several input planes.
`nn.Conv2d`	Applies a 2D convolution over an input signal composed of several input planes.
`nn.Conv3d`	Applies a 3D convolution over an input signal composed of several input planes.
`nn.ConvTranspose1d`	Applies a 1D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose2d`	Applies a 2D transposed convolution operator over an input image composed of several input planes.
`nn.ConvTranspose3d`	Applies a 3D transposed convolution operator over an input image composed of several input planes.

(3) Loss Functions



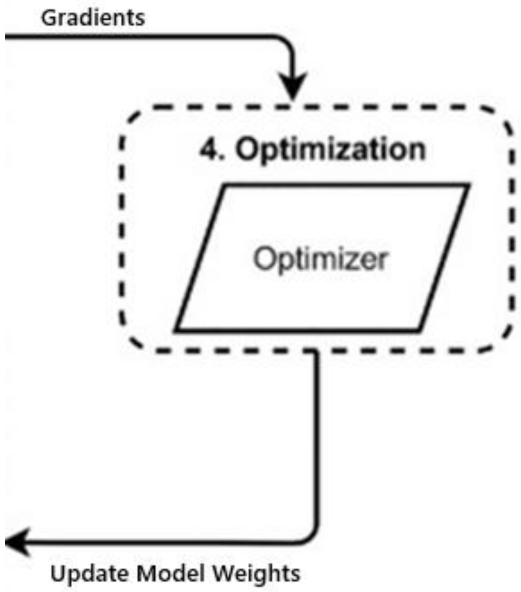
- Measure how far predictions are from the true labels
- Compute gradients – *loss.backward()*

More on loss functions at: https://docs.pytorch.org/docs/stable/nn.html#loss-functions

Loss Functions

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.
`nn.CTCLoss`	The Connectionist Temporal Classification loss.
`nn.NLLLoss`	The negative log likelihood loss.
`nn.PoissonNLLLoss`	Negative log likelihood loss with Poisson distribution of target.
`nn.GaussianNLLLoss`	Gaussian negative log likelihood loss.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss.
`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
`nn.BCEWithLogitsLoss`	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
`nn.MarginRankingLoss`	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch or 0D *Tensors*, and a label 1D mini-batch or 0D *Tensor* y (containing 1 or -1).

(4) Optimizers



Take the calculated gradients and update model parameters – *optimizer.step()*

More about *torch.optim* at: https://docs.pytorch.org/docs/stable/optim.html

torch.optim

Created On: Jun 13, 2025 | Last Updated On: Aug 24, 2025

`torch.optim` is a package implementing various optimization algorithms.

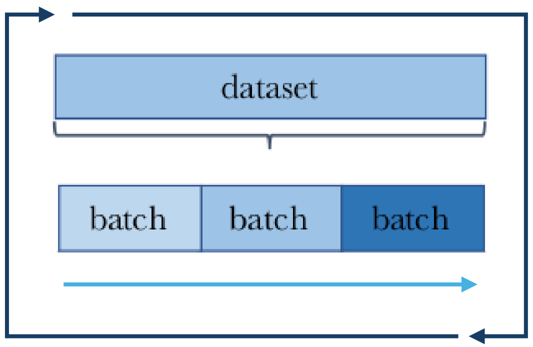
Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can also be easily integrated in the future.

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adafactor`	Implements Adafactor algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm, where weight decay does not accumulate in the momentum nor variance.
`SparseAdam`	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.

Putting everything together

1. Data Preparation
   a) Load Data
   b) Dataset (apply transforms, etc.)
   c) DataLoader – gives batches of *(inputs, labels)*
2. Build the neural network – Stack layers!
3. Define loss and optimizer
4. Train the neural network by going through the dataset *multiple times*
   a) Get model predictions – *model(one batch of inputs)*
   b) Compute the loss – *loss(prediction, labels)*
   c) Clear/Zero out old gradients – *optimizer.zero_grad()*
   d) Compute new gradients – *loss.backward()*
   e) Tell optimizer to update model weights – *optimizer.step()*
5. Test the trained neural network



References

[1] Previous course iteration's slides

[2] PyTorch. (2024). PyTorch documentation — PyTorch 2.7 documentation. Pytorch.org. https://docs.pytorch.org/docs/stable/index.html

[3] Roman, V. (2020, January 19). Deep Learning: Introduction to Tensors & TensorFlow | Towards Data Science. Towards Data Science. https://towardsdatascience.com/deep-learning-introduction-to-tensors-tensorflow-36ce3663528f/

[4] Neural Network Training - an overview | ScienceDirect Topics. (n.d.). Www.sciencedirect.com. https://www.sciencedirect.com/topics/engineering/neural-network-training

[5] Päpper, M. (2022, February 28). Rethinking Batch in BatchNorm. Read. Hack. Learn. Repeat. https://www.paepper.com/blog/posts/rethinking-batch-in-batchnorm/

[6] *Deep Learning: How do deep neural networks work?» Lamarr-Blog. (2021, April 21). Lamarr Institute for Machine Learning and Artificial Intelligence. https://lamarr-institute.org/blog/deep-neural-networks/*