PyTorch

Tutorial II

Course CS 419

Instructor Prof Abir De

Aditya Pande

[22m2108 @iitb.ac.in]

Ninad Gandhi

[22m2151 @iitb.ac.in]

</ 0. Recap />

Previously Discussed

1. MNIST Dataset
2. Neural Network Architecture
    a. Activations
    b. Mathematical Form
    c. Probability and Logits

Previously Discussed: Activations

$\underline{Sigmoid}$

$$\sigma(x) = \frac{1}{1 + e^{-(x)}}$$

$\underline{Tanh}$

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$\underline{ReLU}$

$$\sigma(x) = max(x, 0)$$

$\underline{LeakyReLU}$

$$\sigma(x) = max(0.1x, x)$$

[Visualization]

Previously Discussed: Mathematical Form

$$y_i = f(x_i)$$

$$y_i = W_1^T \cdot x_i + b_1$$

$$h_1 = \sigma_1(W_1^T \cdot x_i + b_1)$$

$$h_2 = \sigma_2(W_2^T \cdot (h_1) + b_2)$$

$$\vdots$$

$$h_l = \sigma_l(W_l^T \cdot (h_{l-1}) + b_l)$$

$$logits = W_{out}^T \cdot (h_l) + b_{out}$$

Next in line…

1. Tensors
2. Datasets and Dataloaders
3. Batching
4. Model Architecture
5. Forward Pass
6. Optimizer
7. Loss Objective
8. Device: GPU v CPU
9. Train Trajectory
10. Early Stopping
11. Reproducibility

</ 1. Tensors />

Tensor: What is a tensor?

A torch tensor is a multi-dimensional array used for data and computations in PyTorch

```python
import torch

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

# Print the tensor
print("Tensor x:", x)

# Perform basic operations
y = x + 2.0          # Addition
z = x * 3.0          # Multiplication

# Print results
print("x + 2:", y)   # Output: tensor([3.0, 4.0, 5.0])
print("x * 3:", z)   # Output: tensor([3.0, 6.0, 9.0])
```

</ TENSOR : go in BATCHES />

Always use BATCHES !!!!!!!!!

- Groups of data samples processed together
- Batch processing speeds up training and improves efficiency by enabling parallel computation and reducing memory usage

```
any_data_tensor .shape == [batch_size, .,
.]
```

</ TENSOR : Attributes : .shape />

How to access attribute ?

An attribute in is a characteristic or of that tensor

An attribute can be accessed with .

```python
import torch

# Create a larger tensor
tensor = torch.randn(10, 20, 30)  # A tensor with shape (10, 20, 30)

# Print the shape of the tensor
print("Shape of the tensor:", tensor.shape)
```

</ Dataloader />

- handles batching, shuffling, and loading data efficiently for training and evaluation

- It works with a Dataset to streamline data handling during model training

```python
import torch
from torch.utils.data import DataLoader, TensorDataset

# Generate random data
num_samples = 1000
num_features = 20
# Random features
X = torch.randn(num_samples, num_features)
 # Random 10-ary labels
y = torch.randint(0, 10, (num_samples,))

# Create a TensorDataset from the random data
dataset = TensorDataset(X, y)

# Create DataLoader
batch_size = 64
data_loader = DataLoader(dataset, batch_size=batch_size,
                         shuffle=True, num_workers=2)
```

Tensor: Methods

- transpose(T)
- squeeze
- unsqueeze
- size
- log, log10, log2, …
- mean, median, mode, std
- nelement, element_size
- dtype: fp16, fp32, fp64
- relu, sigmoid, tanh, softmax
- Topk, sort, sum

Tensor: Methods

- transpose(T)
- squeeze
- unsqueeze
- size
- log, log10, log2, …
- mean, median, mode, std
- nelement, element_size
- dtype: fp16, fp32, fp64
- relu, sigmoid, tanh, softmax
- Topk, sort, sum
- flatten, reshape
- view
- max, min
- logsumexp
- ndim
- norm
- Permute
- multinomial, random
- requires_grad

Tensor: Methods (more)

- abs
- abs_
- absolute
- absolute_
- acos
- acos_
- acosh
- acosh_
- add
- add_
- addbmm
- addbmm_
- addcdiv
- addcdiv_
- addcmul
- addcmul_
- addmm
- addmm_
- addmv
- addmv_
- addr
- addr_
- adjoint
- align_as
- align_to
- all
- allclose
- amax
- amin
- aminmax
- angle
- any
- apply_
- arccos
- arccos_
- arccosh
- arccosh_
- arcsin
- arcsin_
- arcsinh
- arcsinh_
- arctan
- arctan2
- arctan2_
- arctan_
- arctanh
- arctanh_
- argmax
- argmin
- argsort
- argwhere
- as_strided
- as_strided_
- as_strided_scatter
- as_subclass
- asin
- asin_
- asinh
- asinh_
- atan
- atan2
- atan2_
- atan_
- atanh
- atanh_
- backward
- baddbmm
- baddbmm_
- bernoulli
- bernoulli_
- bfloat16
- bincount
- bitwise_and
- bitwise_and_
- bitwise_left_shift
- bitwise_left_shift_
- bitwise_not
- bitwise_not_
- bitwise_or
- bitwise_or_
- bitwise_right_shift
- bitwise_right_shift_
- bitwise_xor
- bitwise_xor_
- bmm
- bool
- broadcast_to
- byte
- cauchy_
- ccol_indices
- double
- ceil
- ceil_
- cfloat
- chalf
- char
- cholesky
- cholesky_inverse
- cholesky_solve
- chunk
- clamp
- clamp_
- clamp_max
- lamp_max_
- clamp_min
- clamp_min_
- clip
- clip_
- clone
- coalesce
- col_indices
- conj
- conj_physical
- conj_physical_
- contiguous
- copy_
- copysign
- copysign_
- corrcoef
- cos
- cos_
- cosh
- cosh_
- count_nonzero
- cov
- cpu
- cross
- crow_indices
- cuda
- cummax
- cummin
- cumprod
- cumprod_
- cumsum
- cumsum_
- data
- data_ptr
- deg2rad
- deg2rad_
- dense_dim
- dequantize
- det
- detach
- detach_
- device
- diag
- diag_embed
- diagflat
- diagonal
- diagonal_scatter
- diff
- digamma
- digamma_
- dim
- dim_order
- dist
- div
- div_
- divide
- divide_
- dot
- double
- dsplit
- dtype
- eig
- element_size
- eq
- eq_
- equal
- erf
- erf_
- erfc
- erfc_
- erfinv
- erfinv_
- exp
- exp2
- exp2_
- exp_
- expand
- expand_as
- expm1
- expm1_
- exponential_
- fill_
- fill_diagonal_
- fix
- fix_
- flatten
- flip
- fliplr
- flipud
- float
- float_power
- float_power_
- floor
- floor_
- floor_divide
- floor_divide_
- fmax
- fmin
- fmod
- fmod_
- frac
- frac_
- frexp
- gather
- gcd
- gcd_
- ge
- ge_
- geometric_
- geqrf
- ger
- get_device
- grad
- grad_fn
- greater
- greater_
- greater_equal
- greater_equal_
- gt
- gt_
- half
- hardshrink
- has_names
- heaviside
- heaviside_
- histc
- histogram
- hsplit
- hypot
- hypot_
- i0
- i0_
- igamma
- igamma_
- igammac
- igammac_
- imag
- index_add
- index_add_
- index_copy
- index_copy_
- index_fill
- index_fill_
- index_put
- index_put_
- index_reduce
- index_reduce_
- index_select
- indices
- inner
- int
- int_repr
- inverse
- ipu
- is_coalesced
- is_complex
- s_conj
- is_contiguous
- is_cpu
- is_cuda
- is_distributed
- is_floating_point
- is_inference
- is_ipu
- is_leaf
- is_maia
- is_meta
- is_mkldnn
- is_mps
- is_mtia
- is_neg
- is_nested
- is_nonzero
- is_pinned
- is_quantized
- is_same_size
- is_set_to
- is_shared
- is_signed
- is_sparse
- is_sparse_csr
- is_vulkan
- is_xla
- is_xpu
- isclose…

</ 2. Dataset and Dataloaders />

Built-in Datasets

Torchvision: (https://pytorch.org/vision/stable/datasets.html)

- MNIST
- CIFAR-10
- CIFAR-100
- ImageNet

Torchaudio: (https://pytorch.org/audio/stable/datasets.html)

- VoxCeleb1Identification
- CMUARCTIC

</ MNIST : dataset example />

```python
# 1. Data Loading and Preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert image to tensor
    transforms.Normalize((0.5,), (0.5,))  # Normalize data
])

# Download and load training data
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=2)

# Download and load test data
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False, num_workers=2)
```

Custom Dataset

Object type: torch.utils.data.Dataset

Implement 3 methods:

1. __init__()
2. __len__()
3. __getitem__()

```python
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

Dataloaders

- The DataLoader class in PyTorch handles *batching*, *shuffling*, and *parallel* data loading using multiple workers.

Dataloaders

- The DataLoader class in PyTorch handles *batching*, *shuffling*, and *parallel* data loading using multiple workers.
- Dataloader wraps around the Dataset object.

Dataloaders

- The DataLoader class in PyTorch handles *batching*, *shuffling*, and *parallel* data loading using multiple workers.
- Dataloader wraps around the Dataset object.
- You must split the dataset into train test and val before using the dataloader.

</ MNIST : dataloader example />

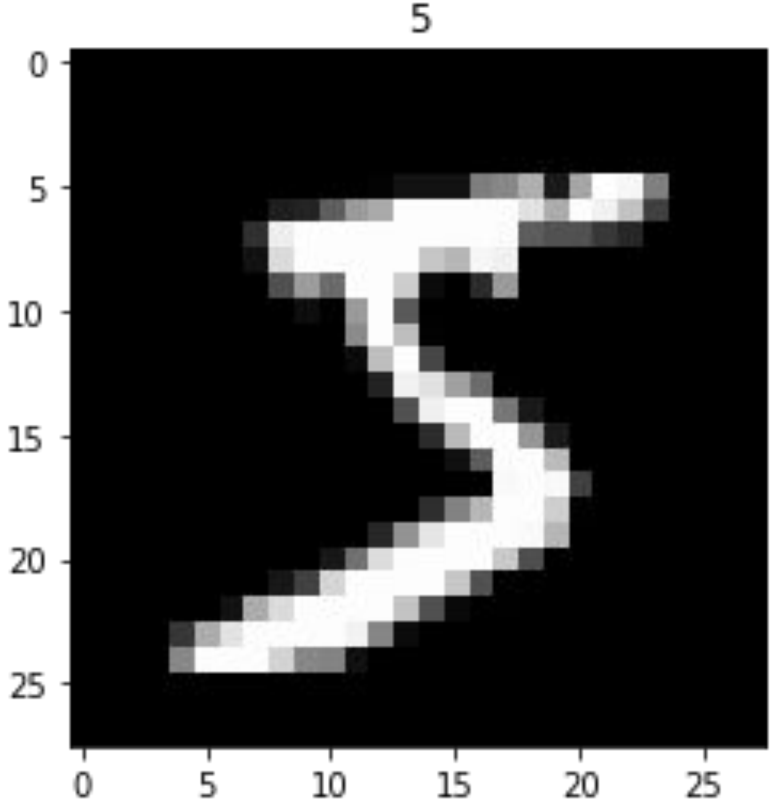
- handles batching, shuffling, and loading data efficiently for training and evaluation

- It works with a Dataset to streamline data handling during model training

```python
import torch
from torch.utils.data import DataLoader, TensorDataset

# Generate random data
num_samples = 1000
num_features = 20
# Random features
X = torch.randn(num_samples, num_features)
 # Random 10-ary labels
y = torch.randint(0, 10, (num_samples,))

# Create a TensorDataset from the random data
dataset = TensorDataset(X, y)

# Create DataLoader
batch_size = 64
data_loader = DataLoader(dataset, batch_size=batch_size,
                         shuffle=True, num_workers=2)
```

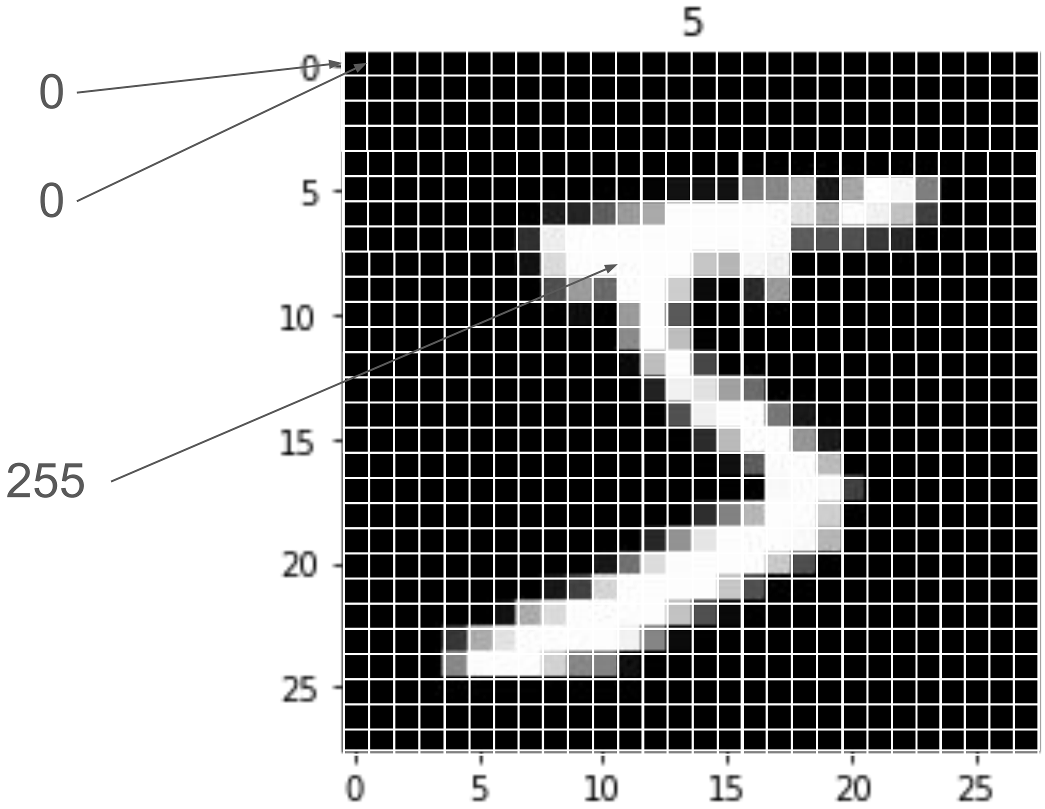
</ 3. Batching />

Batching: One Example

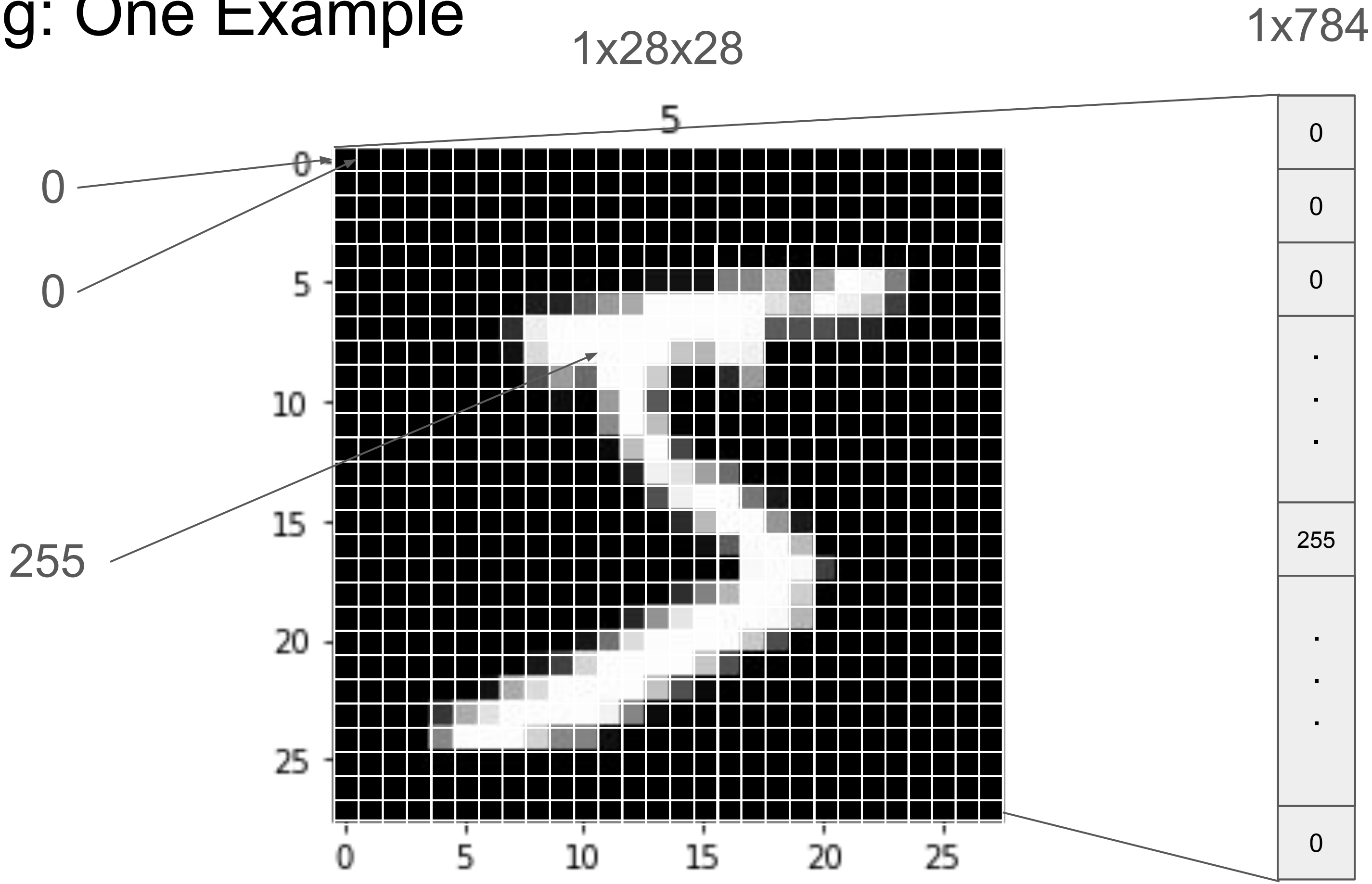


Batching: One Example

1x28x28



Batching: One Example



Batching: Sending multiple examples

0	0	0	0	117	0		0	0
0	0	0	0	140	0		0	0
0	120	0	50	20	0		0	0
. . .	. . .	. . .	. . .	. . .	. . .		. . .	. . .
255	255	200	0	255	255	• • •	255	255
. . .	. . .	. . .	. . .	. . .	. . .		. . .	. . .
0	0	0	0	0	0		0	0

Bx784

</ 4. Model Architecture />

Architecture: Mathematical Form

$$h_1 = \sigma_1(W_1^T \cdot x_i + b_1)$$

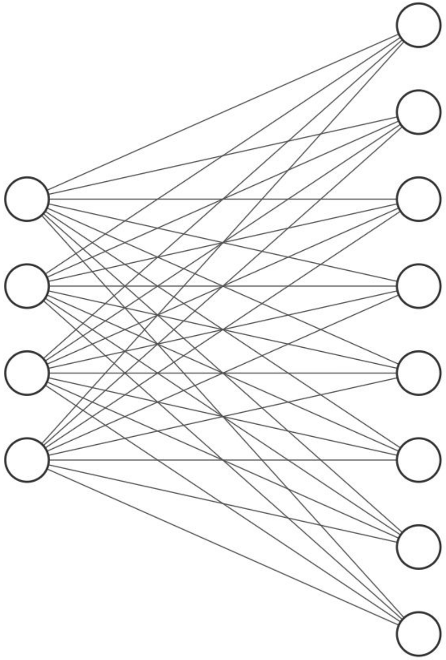
$$h_2 = \sigma_2(W_2^T \cdot (h_1) + b_2)$$

$$\vdots$$

$$h_l = \sigma_l(W_l^T \cdot (h_{l-1}) + b_l)$$

$$logits = W_{out}^T \cdot (h_l) + b_{out}$$

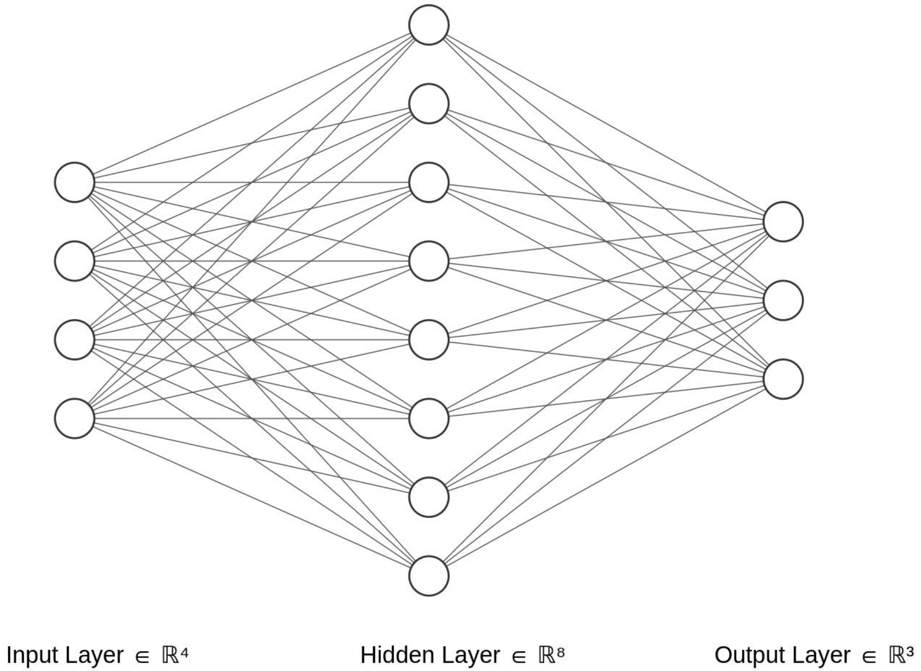
</ Building Block : nn.Linear(in_dim, out_dim) />



Input Layer $\in \mathbb{R}^4$ Output Layer $\in \mathbb{R}^8$

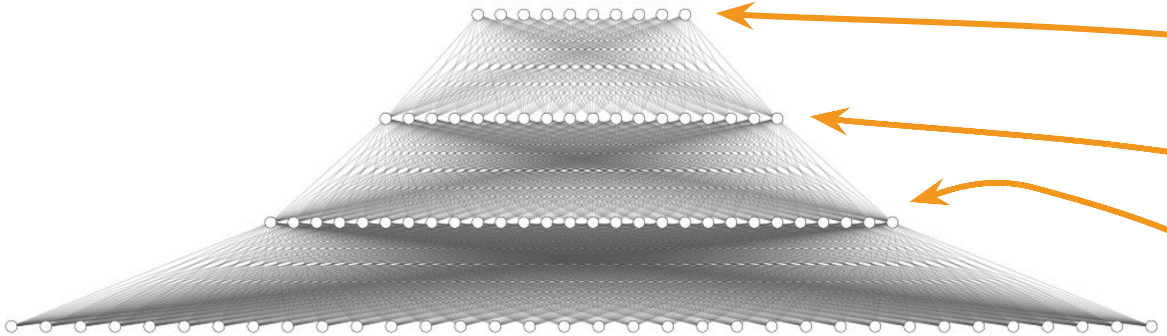
```python
layer = nn.Linear(4, 8)
out = layer(x)
```

</ Building Block : Linear & Activations />




```
layer1 = nn.Linear(4, 8)
layer2 = nn.Linear(8, 3)
out = layer(x)
out = torch.relu(out)
out = torch.relu(out)
```

</ Building Block : OUR MODEL />



```
self.fc1 = nn.Linear(28*28, 128)
self.fc2 = nn.Linear(128, 64)
self.fc3 = nn.Linear(64,  10)
```

</ 5. Forward Method />

</ .forward() />

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64,  10)

    def forward(self, x):
        x = x.view(-1, 28*28)

        h1 = torch.relu(self.fc1(x))
        h2 = torch.relu(self.fc2(h1))
        # ...
        h3 = self.fc3(h2)

        return x
```

$$h_1 = \sigma_1(W_1^T \cdot x_i + b_1)$$

$$h_2 = \sigma_2(W_2^T \cdot (h_1) + b_2)$$

$$\vdots$$

$$h_l = \sigma_l(W_l^T \cdot (h_{l-1}) + b_l)$$

$$logits = W_{out}^T \cdot (h_l) + b_{out}$$

</ 6. Loss Objective />

</ Loss : Cross Entropy />

For K-class classification using the softmax function, the cross-entropy loss $J(\theta)$ is given by:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log \left(h_\theta(x^{(i)})_k \right)$$

```
criterion = nn.CrossEntropyLoss()
```

Where:

- m is the number of training examples.
- K is the number of classes.
- $y_k^{(i)}$ is a binary indicator (0 or 1) that represents whether the class label k is the correct classification for the i-th example.
- $h_\theta(x^{(i)})_k$ is the predicted probability of the i-th example belonging to class k, given by the softmax function:

$$h_\theta(x^{(i)})_k = \frac{\exp(\theta_k^T x^{(i)})}{\sum_{j=1}^{K} \exp(\theta_j^T x^{(i)})}$$

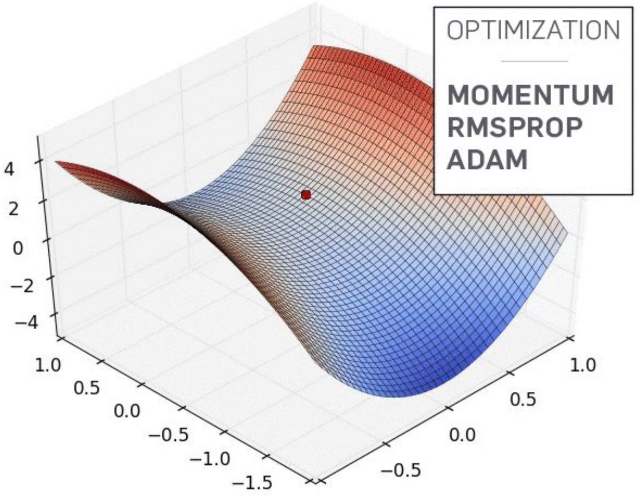
</ Loss : Gradient Descent />

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta J(\theta)$$

```
loss.backward()
```

</ 7. Optimizer />

Various Optimizers




```python
optimizer = optim.Adam(model.parameters(), lr=3e-4)
```

https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

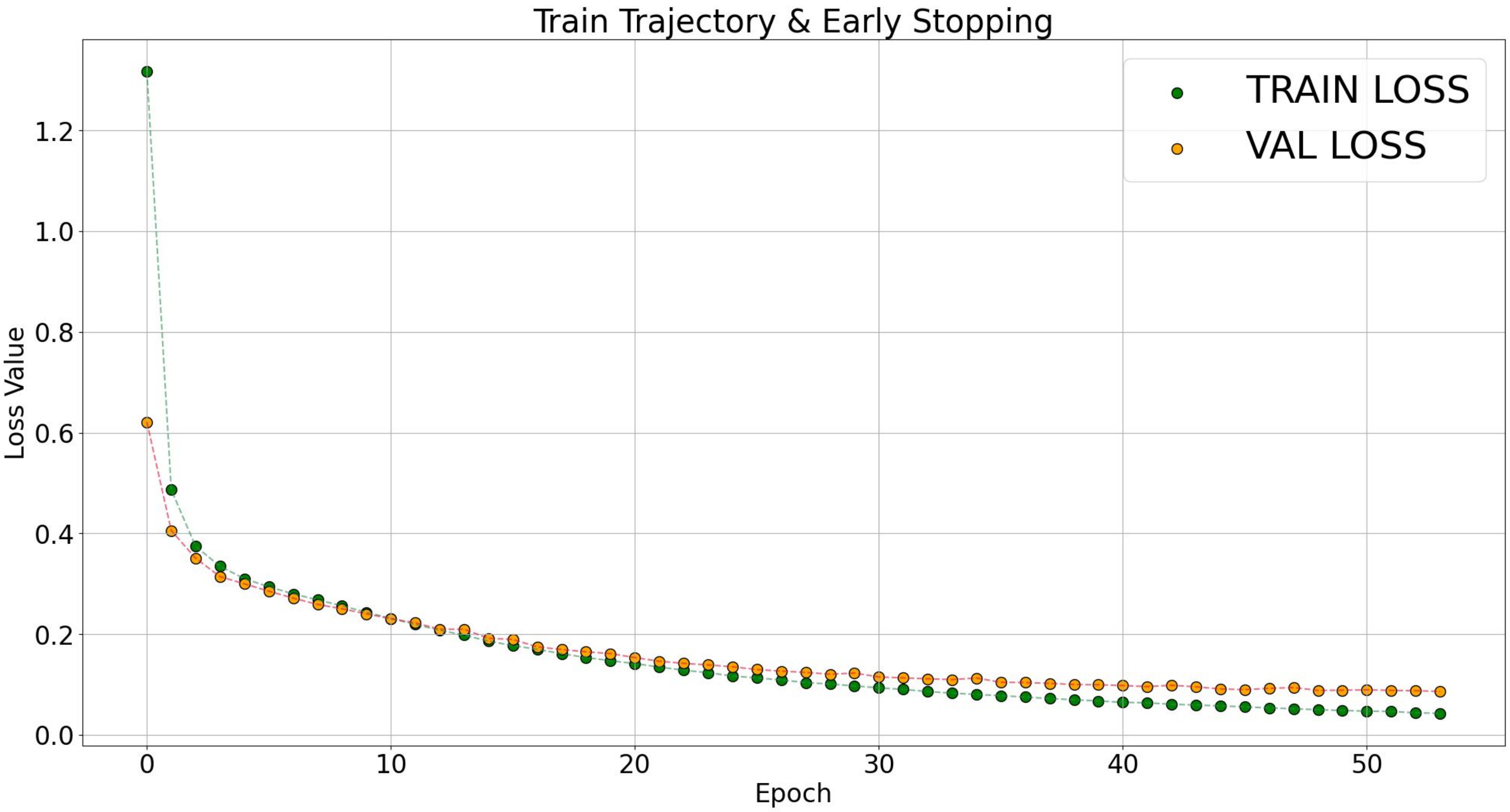
</ Loss : Gradient Descent />

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta J(\theta)$$

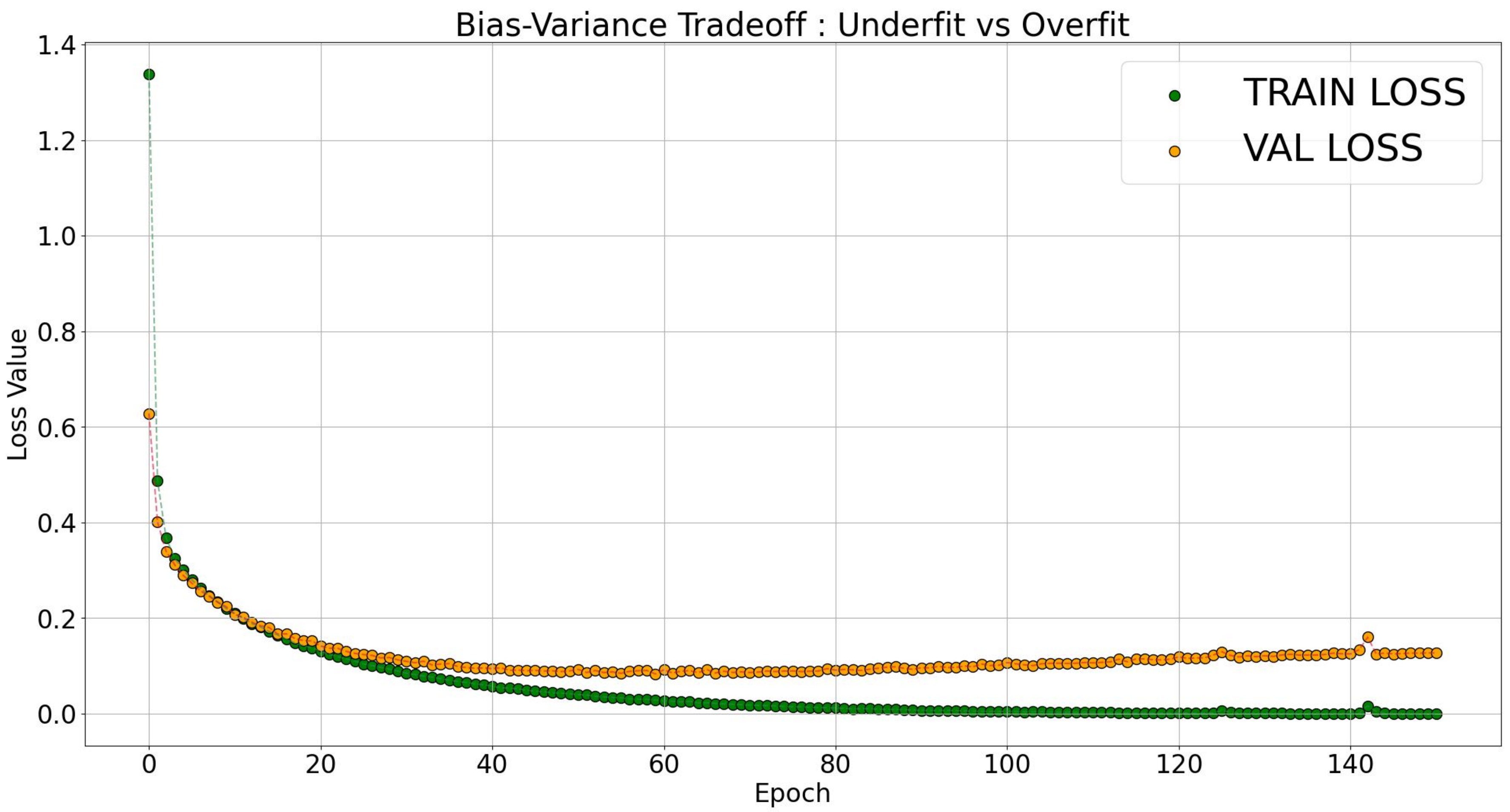
```
loss.backward()

optimizer.step()
```

</ 8. Device: GPU v CPU />

</ 9. Train Trajectory />



Train Trajectory & Early Stopping
TRAIN LOSS
VAL LOSS
Loss Value
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
Epoch



Bias-Variance Tradeoff : Underfit vs Overfit
TRAIN LOSS
VAL LOSS
Loss Value
Epoch
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0
20
40
60
80
100
120
140

</ 10. Reproducibility />

Random Number Generators

- **torch**, **numpy**, and **random** libraries have their own random number generators.

Random Number Generators

- **torch**, **numpy**, and **random** libraries have their own random number generators.
- **Initializing models** generates parameter vectors with random weights.

Random Number Generators

- **torch**, **numpy**, and **random** libraries have their own random number generators.
- **Initializing models** generates parameter vectors with random weights.
- **Shuffling** of data also uses an RNG.

Random Number Generators

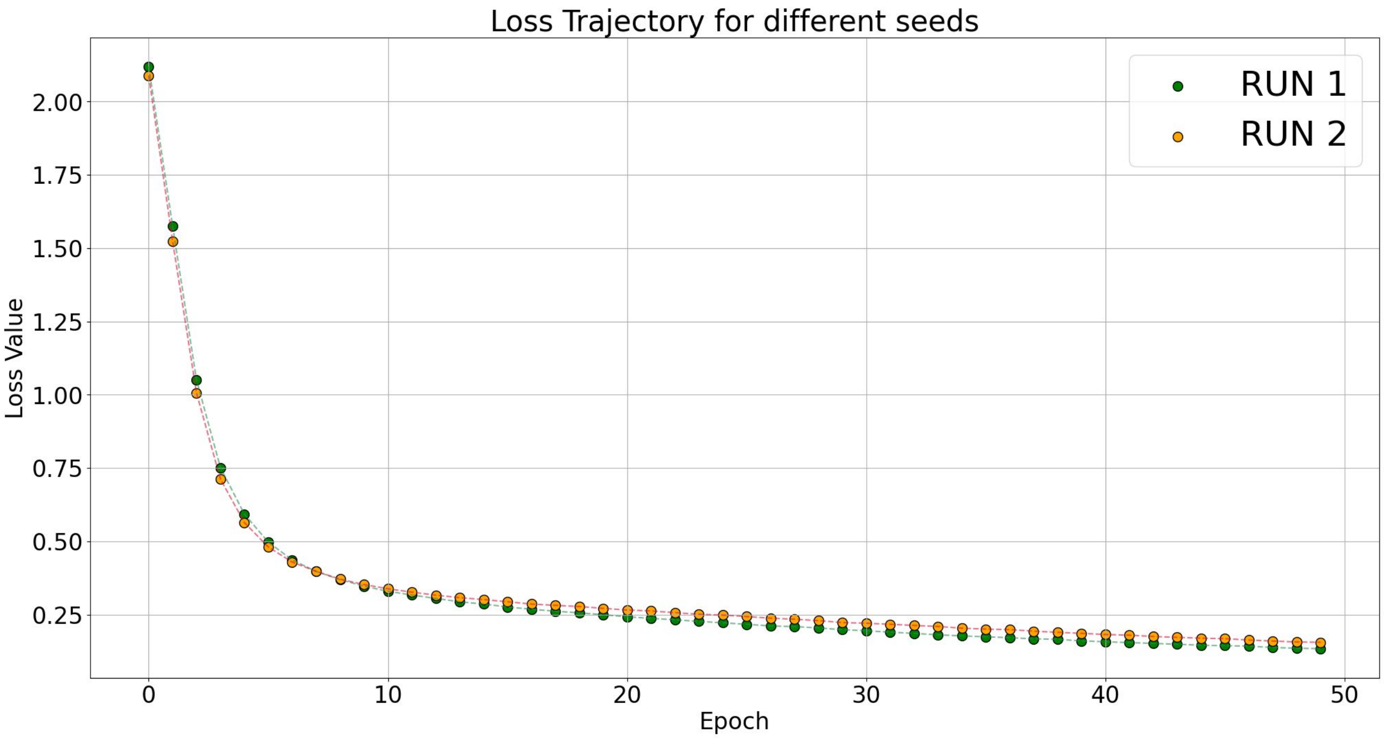
- **torch**, **numpy**, and **random** libraries have their own random number generators.
- **Initializing models** generates parameter vectors with random weights.
- **Shuffling** of data also uses an RNG.
- **Dropout** uses a RNG.

Random Number Generators

- **torch**, **numpy**, and **random** libraries have their own random number generators.
- **Initializing models** generates parameter vectors with random weights.
- **Shuffling** of data also uses an RNG.
- **Dropout** uses a RNG.

- These RNGs are used in the backend implicitly.
- What are the consequences?

Multiple train runs through the epochs



How to reproduce runs?

Initialize the random seeds for each RNG

- One can write a function as shown.
- Seeding everything is a good practice in machine learning.
- Reproducibility is a big aspect of machine learning research so that people can verify your work.

```
def seed_everything(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
```

Non Determinism (Pro stuff)

- It is possible that the training runs are not reproducible even after seeding.
- Non-Deterministic code is because of the hardware and non-deterministic algorithms used by torch in the background.

```
void nll_loss2d_forward_out_cuda_template(
    Tensor& output,
    Tensor& total_weight,
    const Tensor& input,
    const Tensor& target,
    const c10::optional<Tensor>& weight_opt,
    int64_t reduction,
    int64_t ignore_index) {
  // See Note [Writing Nondeterministic Operations]
  // Nondeterministic because of atomicAdd usage in 'sum' or 'mean' reductions.
  if (reduction != at::Reduction::None) {
    at::globalContext().alertNotDeterministic("nll_loss2d_forward_out_cuda_template");
  }
```

https://github.com/pytorch/pytorch/blob/8f1c3c68d3aba5c8898bfb3144988aab6776d549/aten/src/ATen/native/cuda/NLLLoss2d.cu#L236-L240

Non Determinism (Pro stuff)

- It is possible that the training runs are not reproducible even after seeding.
- Non-Deterministic code is because of the hardware and non-deterministic algorithms used by torch in the background.
- Sometimes asking pytorch to use deterministic algorithms can solve problems. For example, NLLLoss becomes deterministic.

```
torch.use_deterministic_algorithms(True)
```

Non Determinism (Pro stuff)

- It is possible that the training runs are not reproducible even after seeding.
- Non-Deterministic code is because of the hardware and non-deterministic algorithms used by torch in the background.
- Sometimes asking pytorch to use deterministic algorithms can solve problems. For example, NLLLoss becomes deterministic.

- But you may still have other dependencies that introduce non-determinism. :(

</ 11. Important Points/>

These are the things you should do in practice

Always…

These are the things you should do in practice

Always:

1. Use git.

These are the things you should do in practice

Always:

1. Use git.
2. Write modular code.

These are the things you should do in practice

Always:

1. Use git.
2. Write modular code.
3. Document your code.

These are the things you should do in practice

Always:

1. Use git.
2. Write modular code.
3. Document your code.
4. Seed everything before training.

These are the things you should do in practice

Always:

1. Use git.
2. Write modular code.
3. Document your code.
4. Seed everything before training.
5. Reuse the available implementations. (Don’t reinvent the wheel)

These are the things you should do in practice

Always:

1. Use git.
2. Write modular code.
3. Document your code.
4. Seed everything before training.
5. Reuse the available implementations. (Don’t reinvent the wheel)
6. Use chatGPT to code.

…

</ Thank You />