PyTorch Fundementals

Prepared By:

H. Fuat Alsan

PyTorch

- Very Pythonic
- Efficient Tensor operations
- Autograd (Automatic Differentiation)
- Ready-to-use deep learning layers and functions
- Handles complex CUDA operations for us
- Uses cuDNN (CUDA Deep Neural Network)

PyTorch Important Components

- Tensors
  - Represent data and math operations
  - Very similar to NumPy arrays
- Dataset
  - Represent all available data
  - Transforms (data augmentation)
- Dataloader
  - Creates batches from the datasets
- Autograd (Automatic Differentiation)
  - Useful for gradient descent, backpropagation
- Model and Layers (nn.Module)
- Optimizers
- Loss Functions
- Learning Rate (LR) schedulers

Datasets

- torch.utils.data.Dataset
- **Built-in Datasets (torchvision.datasets)**
  - MNIST
  - CIFAR10
- **Imagefolder**
  - Useful for classification
- **Custom Dataset Class**
  - Most flexiable
- See: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

Datasets

- Train/validation/test split

Data transformation & augmentations

Download & load built-in dataset for training and testing

```
from torchvision import datasets, transforms

transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,)),
])

train_dataset = datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

test_dataset = datasets.MNIST(
    root='./data',
    train=False,
    transform=transform
)
```

ImageFolder

```
├── train
  ├── class1
    |     ├── 1.jpg
    |     ├── 2.jpg
  ├── class2
    |     ├── 1.jpg
    |     ├── 2.jpg
├── valid
  ├── class1
    |     ├── 1.jpg
    |     ├── 2.jpg
  ├── class2
    |     ├── 1.jpg
    |     ├── 2.jpg
```

(Root data folder structure)

ImageFolder

root_folder/train/dogs/001.png
root_folder/train/dogs/002.png
...
root_folder/val/dogs/001.png
root_folder/val/dogs/002.png
...
root_folder/train/cats/001.png
root_folder/train/cats/002.png
...
root_folder/val/cats/001.png
root_folder/val/cats/002.png

train_dataset = torchvision.datasets.ImageFolder(root='train’, train_transforms)
valid_dataset = torchvision.datasets.ImageFolder(root='val’, val_transforms)
(AUTOMATICALLY ASSINGS CLASS LABELS USING FOLDER STRUCTURE)

Custom Dataset Class

Subclass of Dataset

Total lenght of dataset

Function used for fetching data

Reading image file

Convert to tensors

Apply data augmentation

```python
class BasicSegmentationDataset(Dataset):
    def __init__(self, df_full, class_colors, class_mapping, transform=None, reverse=True):
        self.df_full = df_full
        self.class_colors = class_colors
        self.class_mapping = class_mapping
        self.transform = transform
        self.reverse = reverse

        self.xml_files = df_full.filename.unique()

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

    def __getitem__(self, idx):
        # id_rooms.png (target image)
        selected_xml_file = self.xml_files[idx]
        selected_img_file = selected_xml_file[:-4] + '.tiff'

        image_1 = Image.open(selected_img_file).convert('RGB')
        # for grayscale
        #image_2 = Image.open(os.path.join(self.file_dir, selected_img_file)).convert('L')
        #image_2 = Image.open(os.path.join(self.file_dir, selected_img_file_2)).convert('RGB')

        image_1 = np.array(image_1).astype(np.float32) # ORIGINAL IMAGE
        #image_2 = swap_pixels(image_1.copy()).astype(np.float32) # MASK
        _df = self.df_full[self.df_full.filename == selected_xml_file]
        image_2 = self.draw_masks(image_1.copy(), _df)

        # Feature scaling
        image_1 = normalize_image(image_1)
        image_2 = normalize_image(image_2)

        # convert to tensor
        # (Width, Height, Channel) -> (Channel, Width, Height)
        image_1 = torch.from_numpy(image_1.copy().transpose((2,0,1)))
        image_2 = torch.from_numpy(image_2.copy().transpose((2,0,1)))
        # for grayscale (repeats grayscale channel over RGB channels)
        #image_2 = torch.from_numpy(image_2.copy()).unsqueeze(0).repeat(3, 1, 1)

        # Apply image transformation (if any)
        if self.transform is not None:
            both_images = torch.cat((image_1.unsqueeze(0), image_2.unsqueeze(0)), 0)
            transformed_images = self.transform(both_images)

            image_1 = transformed_images[0]
            image_2 = transformed_images[1]

        return image_1, image_2
```

Dataset Bacthing

Pass dataset

Batch size

```
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=32,
    shuffle=True,
    num_workers=6,
    pin_memory=False,
)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=32,
    shuffle=True,
    num_workers=6,
    pin_memory=False,
)
```

Add batch dimension to the data

Original: (1, 28, 28)
Batched: (32, 1, 28, 28)

```
ex_img_batch, ex_target_batch = next(iter(train_loader))
print(ex_img_batch.shape)
print(ex_target_batch.shape)
```

```
torch.Size([32, 1, 28, 28])
torch.Size([32])
```

PyTorch nn.Module Hierarchy

- **nn.Module** is the base class for everything in PyTorch
- nn.Module2 **(contains)** nn.Module1 **(contains)** nn.Parameter **(contains)** tensors
- Commonly used layers:
  - nn.Linear
  - nn.Conv2d
  - nn.ConvTranspose2d
  - nn.ReLU
  - nn.Sigmoid

Basic nn.Module

Subclass of nn.Module
super().__init__() must be called!

__init__() is special function that defines the module itself

nn.Module can contain other nn.Module(s)

Forward function is called when forward pass on this module is done

```python
class SimpleLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        self.fc1 = nn.Linear(in_features, out_features)

        self.act_fn = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.act_fn(x)
        return x
```

There is no need for defining backward pass function, autograd handles that for us

Another nn.Module

__init__() function takes arguments to define the module

Other nn.Modules

Forward pass

```python
class DownsampleBlock(nn.Module):
    def __init__(self, in_channels, out_channels, down_ratio):
        super().__init__()

        # Conv layer with kernel_size=(3, 3) and padding=1
        # This layer normally doesn't change the width and heigth
        # However, setting stride=down_ratio we downsample the image
        # Also: channels are converted fro in_channels to out_channels
        self.conv = nn.Conv2d(
            in_channels=in_channels,
            out_channels=out_channels,
            kernel_size=3,
            stride=down_ratio, # divide by down_ratio
            padding=1,
            bias=False
        )

        self.norm = nn.BatchNorm2d(out_channels)

        self.relu = nn.ReLU()

    def forward(self, x_in):
        # (N, in_channels, H, W)
        x_features = self.conv(x_in)

        # (N, out_channels, H//down_ratio, W//down_ratio)
        # doesn't change dims, only feature scaling
        x_features = self.norm(x_features)

        # (N, out_channels, H//down_ratio, W//down_ratio)
        # doesn't change dims, only non-linear activation
        x_features = self.relu(x_features)

        # final dims: (N, out_channels, H//down_ratio, W//down_ratio)
        return x_features
```

nn.Module with No Learnable Parameters

Does not have any other nn.Module(s)

Performs a very basic forward pass operation

```
class Flatten(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        # (N, CHANNELS, HEIGHT, WIDTH)
        batch_size = x.shape[0]
        # Merge into single dimension
        # (N, CHANNELS*HEIGHT*WIDTH)
        x = x.reshape(batch_size, -1)
        return x
```

nn.Module with No Learnable Parameters

```python
class UpscaleLayer(nn.Module):
    def __init__(self, scale_factor):
        super().__init__()
        self.scale_factor = scale_factor

    def forward(self, x):
        # (N, CHANNELS, HEIGHT, WIDTH)
        # Perform nearest-neighbor interpolation to upscale the input
        # Assigns the value of the nearest input pixel
        x_upscaled = nn.functional.interpolate(x, scale_factor=self.scale_factor, mode='nearest')
        # x_upscaled dims:
        # (N, CHANNELS, HEIGHT*scale_factor, WIDTH*scale_factor)
        return x_upscaled
```

Same as before but different forward pass operation

Full Model

Each of these are nn.Module layers

Convolutional part

Fully connected (nn.Linear) part

```
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)

        self.max_pool = nn.MaxPool2d(kernel_size=2)

        self.relu = nn.ReLU()

        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        #######################
        # Convolutional Part
        #######################
        print(f'Input dims: {x.shape}')
        x = self.conv1(x) # (N, 1, 28, 28) -> (N, 32, 26, 26)
        print(f'After conv1 {x.shape}')
        x = self.relu(x) # no dim change
        x = self.conv2(x) # (N, 32, 26, 26) -> (N, 64, 24, 24)
        print(f'After conv2 {x.shape}')
        x = self.relu(x) # no dim change
        x = self.max_pool(x) # (N, 64, 24, 24) -> (N, 64, 12, 12)
        print(f'After maxpool {x.shape}')
        #######################
        #######################

        #######################
        ## Fully Connected Part
        #######################
        x = torch.flatten(x, 1) # (N, 64, 12, 12) -> (N, 64*12*12) -> (N, 9216)
        x = self.fc1(x) # (N, 9216) -> (N, 128)
        x = self.relu(x) # no dim change
        logits = self.fc2(x) # (N, 128) - (N, 10)
        #######################
        #######################

        return logits
```

Classification Model

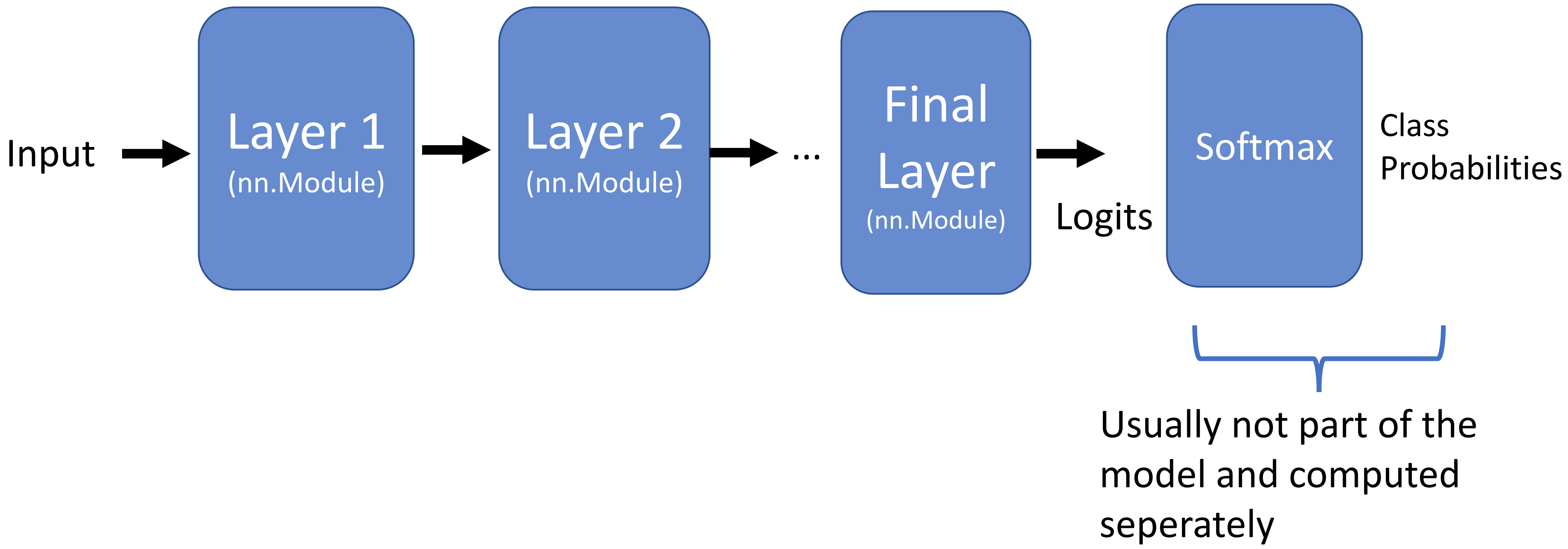
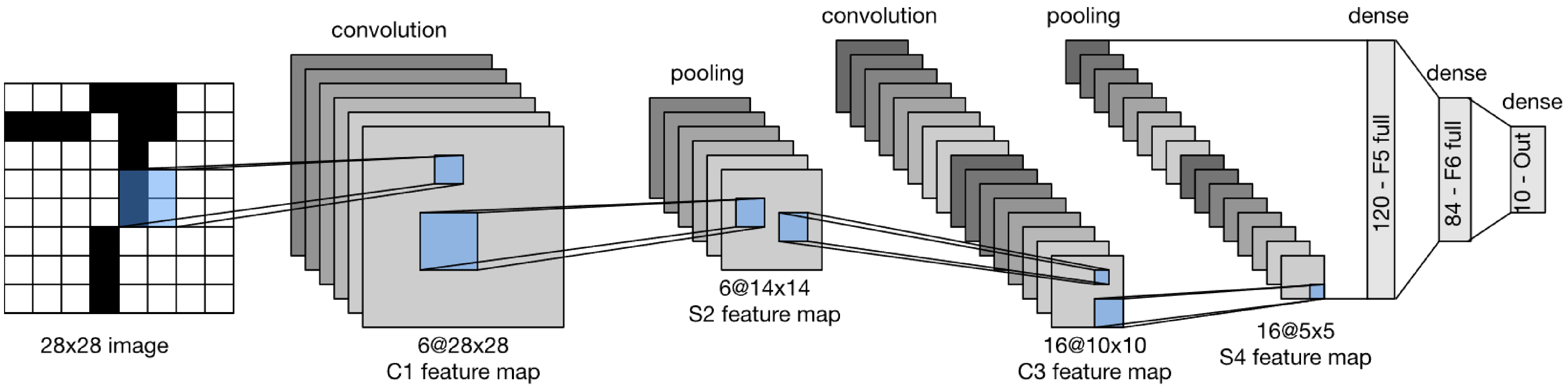
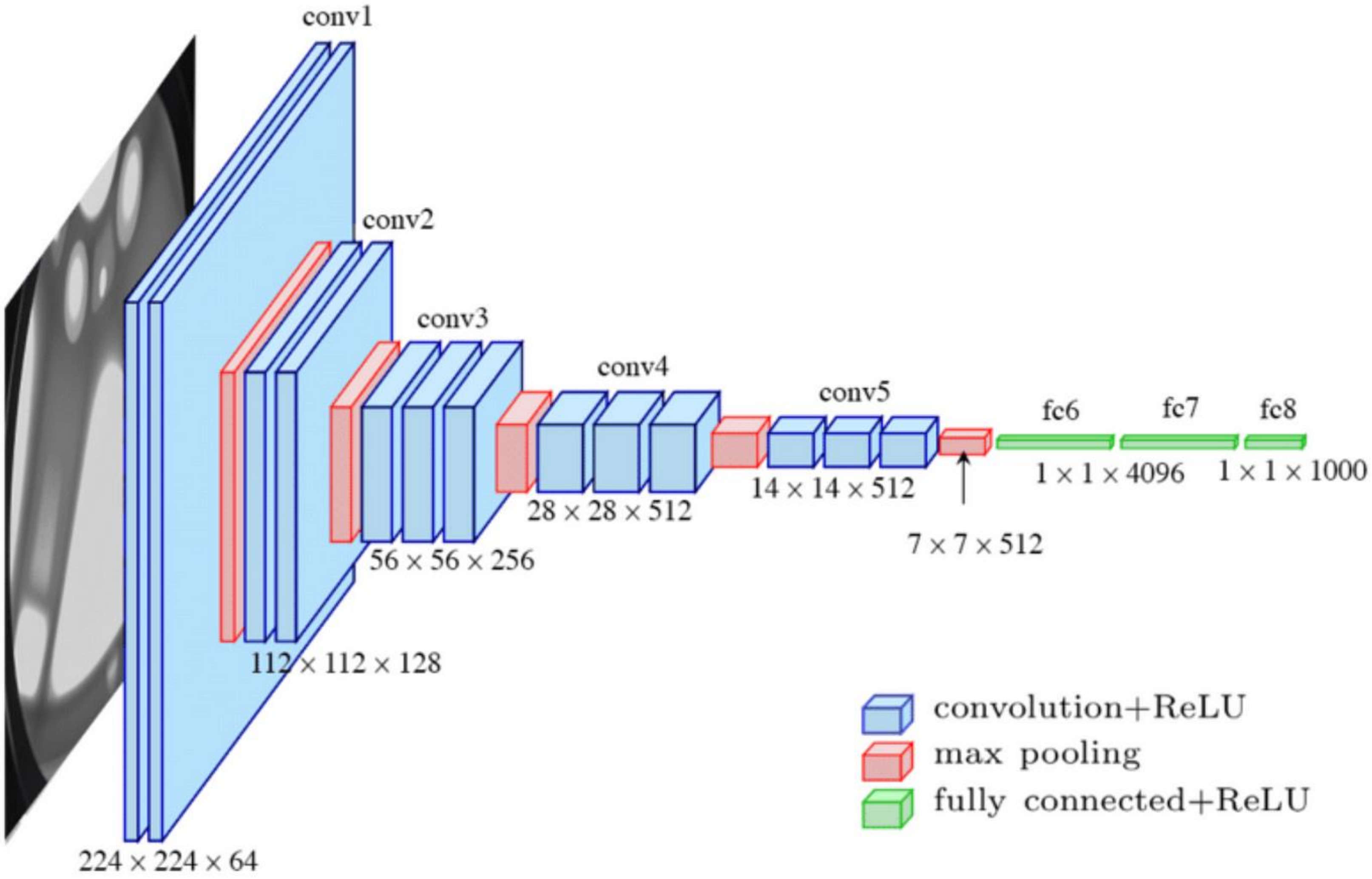


Image Classifier for MNIST Dataset



A Modern Image Classifier (VGG16)



Training Classifiers

- In regression we had continuos values, we used MSE
- However, this is not the case in classification
- In classification, we have fixed classes
  - Each class is represented by an ingeter ID
  - For ex: Cat/Dog classifier
  - Cat -> 0
  - Dog -> 1
- In classification, we use a loss function called **cross entropy**
  - Actually it is **negative log-likelihood** loss

Training Classifiers

- **Batch iteration:** single cycle of batch data
  - **batch_size** amount of data is used for every iteration
  - Last iteration might have less data (dataset is not fully divisible by batch size)
    - (Use **drop_last=True** in dataloader to ignore this)
- **Epoch:** single cycle of full dataset
  - All batch iterations are completed
- **Backpropagation**
  - **Foward pass:** Model predicts, loss is computed
  - **Backward pass:** Using loss, gradients are computed and parameters are updated

Training Loop

Switch model to training mode

Loop over all batches

Move data to GPU (if available)

Forward pass and compute loss

Backward pass, compute gradients and update model parameters

```
def train(model, train_loader, optimizer, criterion, epoch):
    model.train()

    for batch_idx, (img, target) in enumerate(train_loader):
        img, target = img.to(device), target.to(device)

        # Zero gradients, perform a backward pass, and update the weights.
        # In PyTorch, gradients are accumulated, you need to reset gradients in each loop
        optimizer.zero_grad()

        # Forward pass
        preds = model(img)
        loss = criterion(preds, target)

        # Compute gradients
        loss.backward()
        # Update gradients
        optimizer.step()
```

Test/Vadilation Loop

Disable Autograd

Switch model to eval mode

Loop over all batches

Move data to GPU (if available)

Forward pass and compute loss

Compute accuracy

```
@torch.no_grad()
def test(model, test_loader, criterion):
    model.eval()

    test_loss = 0
    correct = 0

    for img, target in test_loader:
        img, target = img.to(device), target.to(device)

        preds = model(img)
        test_loss += criterion(preds, target)

        pred_max = preds.argmax(dim=1, keepdim=True)  # get the index of the max probable class
        correct += pred_max.eq(target.view_as(pred_max)).sum().item()

    test_loss /= len(test_loader.dataset)

    test_acc = 100.0 * correct / len(test_loader.dataset)

    return test_loss
```

There is no backward pass in testing/validation

Loss & Optimizer

Pass model parameters

Learning rate

Initialize loss function
(called criterion)

```
# ConvNet() is defined somewhere
model = ConvNet()

optimizer = torch.optim.SGD(
    model.parameters(),
    lr=0.02
)

# Negative log-likehood with softmax
criterion = nn.CrossEntropyLoss()
```

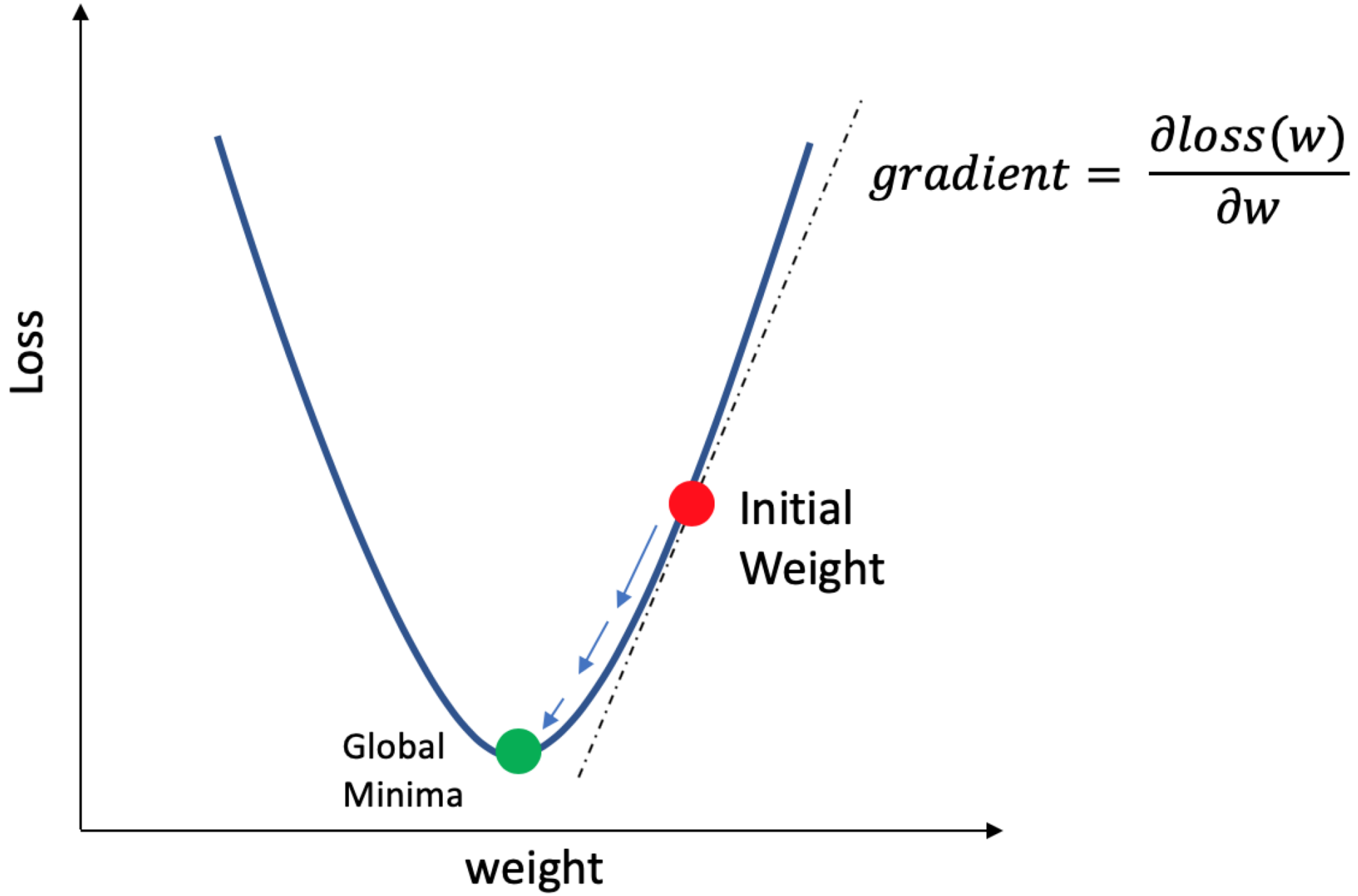
Epoch Loop

```
NUM_EPOCHS = 10

# Move model to GPU (if available)
model.to(device)

for epoch in range(1, NUM_EPOCHS+1):
    train_loss, train_acc = train(model, train_loader, optimizer, criterion, epoch)
    test_loss, test_acc = test(model, test_loader, criterion)

     print(f'Epoch: {epoch}, Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}, Test Loss: {test_loss:.4f}, Test Acc: {test_acc:.2f}')
```

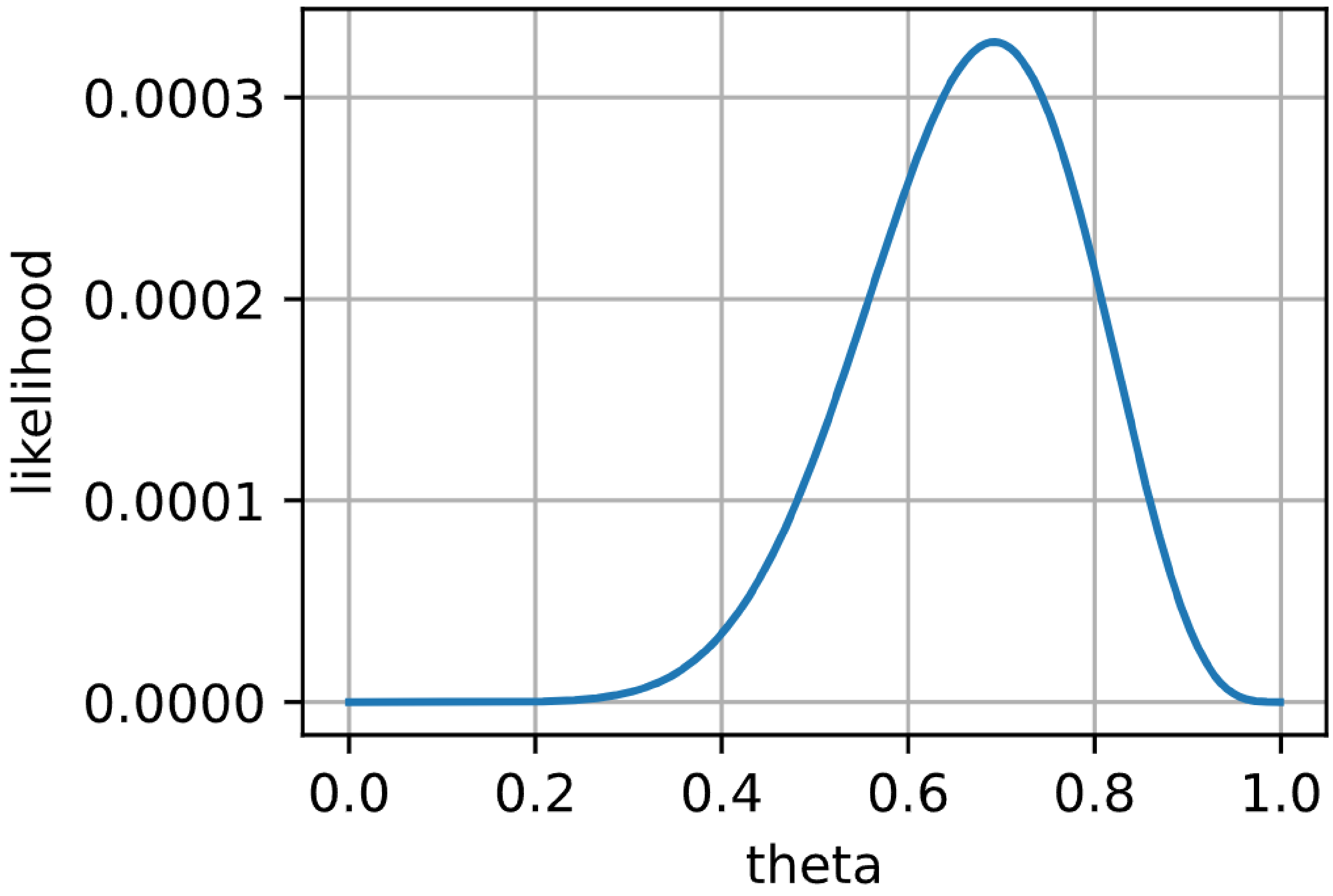
Likelihood Function As Loss

- We wish to increase the probabilty of predicting a class
- Likelihood function is useful for measuring that
- Log likelihood: turns multiplication into addition
  - Faster computation -> Negative log likelihood
- Negative log likelihood
  - Likelihood: better when increased
  - Gradient descent is designed for reducing
  - Hence we add «negative sign» to log likehood
  - This way, we increase the likelihood

Loss vs Likelihood



Loss vs Likelihood



Classification Loss Function

- Binay Case:
- y_i: class id (0 or 1, each indicating a class)
- $\hat{y}_i$: probability of class (predicted by the model)

$$\text{Loss} = -\frac{1}{\substack{\text{output}\\ \text{size}}} \sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

BCE PyTorch Implementation

- Binary Cross Entropy (BCE)
- Automically flattens the data if tensor.dim() > 2
  - For ex: segmentation mask have (N, 10, height, width)
  - Useful for computing loss on segmentation masks
  - (N, flatted_mask_actual), (N, flatted_mask_predected)

Multiclass Classification (Cont.)

$$logloss = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij}\log(p_{ij})$$

- N is the number of rows
- M is the number of classes

Save & Load Model

Model parameters represented as Python dictionary

Optimizer inner state represented as Python dictionary

Filename

```python
# SAVE MODEL AND OPTIMIZER STATE DICT
# SAVE FILENAME 'convnet_checkpoint.pt'
torch.save({
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict()
},
    'convnet_checkpoint.pt'
)
```

Load model state dict

Do this only if training will NOT continue

```python
# LOAD PRE-TRAINED MODEL
model = ConvNet()

checkpoint = torch.load('convnet_checkpoint.pt')
model.load_state_dict(checkpoint['model_state_dict'], strict=True)

# SWITCH MODEL TO PREDICTION ONLY MODE
# (OPTIONAL)
model.eval()
```