AI on NVIDIA Jetson Nano
(Day 3 - 5)

Updated at 3 Sep 2021

Outline

- Face Mask Detection Project (Quickstart)
- Tensors
- Datasets & DataLoaders
- Transforms
- Build the Neural Networks
- Optimizing Model Parameters
- Save and Load the Model

Prerequisites

- Jetson Nano Developer Kit
- Computer with Internet Access and SD card port
- microSD Memory Card (32GB UHS-I mininum)
- Compatible 5V 4A Power Supply with 2.1mm DC barrel connector
- 2-pin jumper
- USB cable (Micro-B to Type-A)
- Logitech C270 Webcam (Optional)

Face Mask Detection Project

- Linux Packages

```
$sudo apt-get install libopenblas-base libopenblas-dev libblas-dev
libatlas-base-dev gfortran libffi-dev
```

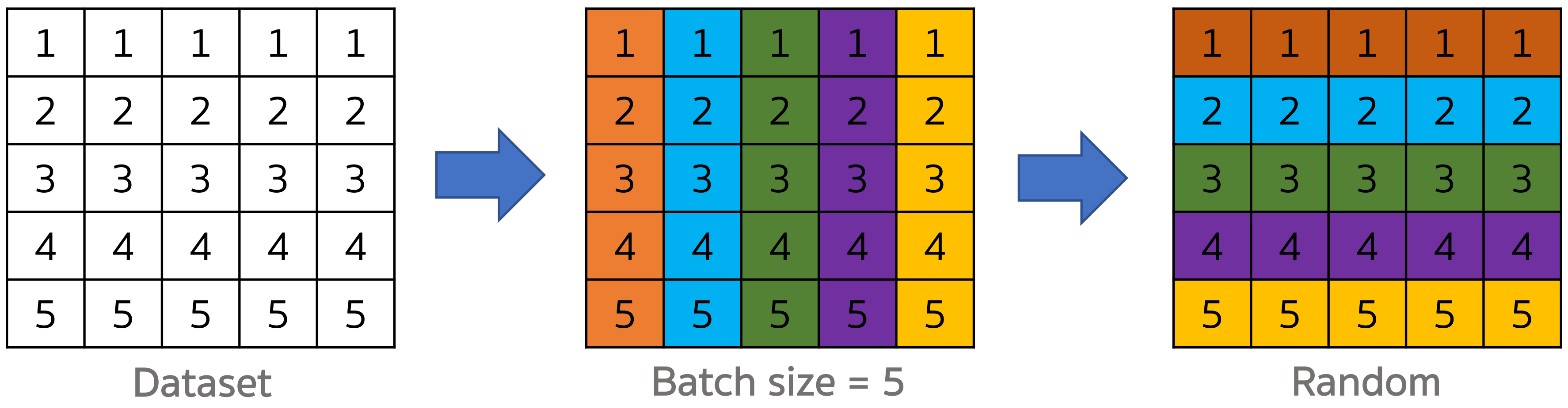
- Python Libraries

```
$pip3 install seaborn
$pip3 install ipywidgets
$pip3 install tqdm
$pip3 install scikit-learn
```

Epochs vs Batch Size vs Iterations

- Epochs
  - One epoch is when an entire dataset is passed forward and backward through the neural network only once
- Batch Size
  - Total number of training examples present in a single batch
- Iterations
  - Iterations is the number of batches needed to complete one epoch

Batch Size



1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
Dataset
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
Batch size = 5
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
Random

Iterations

- Training dataset = 1,000 pics
- Batch size = 10

$$\frac{1{,}000}{10} = 100 \text{ iterations}$$

REPL Prompt

- aka Read Evaluate Print Loop
- Python is an interpreter language. It means it executes the code line by line.
- Python provides a Python Shell, which is used to execute a single Python command and display the result.

Tensors

- Tensor are a specialized data structure that are very similar to arrays and matrices.
- Use tensors to encode the inputs and outputs of a model
- Tensors are similar to NumPy, except that tensors can run on GPUs.

```
import torch
import numpy as np

print(torch.__version__)
```

Tensors (cont'd)

- Initializing a Tensor
  - Directly from data

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
```

  - From a NumPy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

  - From another tensor

```
x_ones = torch.ones_like(x_data)
print(f”Ones Tensor: \n {x_ones} \n”)

x_rand = torch.rand_like(x_data, dtype=torch.float)
print(f”Random Tensor: \n {x_rand} \n”)
```

Tensors (cont'd)

- With random or constant values

```
shape = (2, 3, )
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f”Random Tensor: \n {rand_tensor} \n”)
print(f”Ones Tensor: \n {ones_tensor} \n”)
print(f”Zeros Tensor: \n {zeros_tensor} \n”)
```

Tensors (cont'd)

- Attributes of a Tensor

```
tensor = torch.rand(3, 4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Tensors (cont'd)

- Operations on Tensors

```
if torch.cuda.is_available():
    tensor = tensor.to('cuda')
```

  - Standard numpy-like indexing and slicing

```
tensor = torch.ones(4, 4)
print("First row: ", tensor[0])
print("First column: ", tensor[:, 0])
print("Last column: ", tensor[..., -1])
tensor[:, 1] = 0
print(tensor)
```

Tensors (cont'd)

- Joining tensors

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
```

- Arithmetic operations
  - Compute the matrix multiplication between two tensors

```
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(tensor)
torch.matmul(tensor, tensor.T, out=y3)

z1 = tensor * tensor
z2 = tensor.mul(tensor)
z3 = tensor.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

Tensors (cont’d)

- Single-element tensors

```
agg = tensor.sum()
agg_item = agg.item()

print(agg_item, type(agg_item))
```

- In-place operations

```
print(tensor, “\n”)
tensor.add_(5)
print(tensor)
```

Tensors (cont’d)

- Tensor to NumPy array

```
t = torch.ones(5)
print(f”t: {t}”)
n = t.numpy()
print(f”n: {n}”)
```

- A change in the tensor reflects in the NumPy array

```
t.add_(1)
print(f”t: {t}”)
print(f”n: {n}”)
```

Tensors (cont’d)

- NumPy array to Tensor

```
n = np.ones(5)
t = torch.from_numpy(n)
```

- A change in the NumPy array reflects in the tensor

```
np.add(n, 1, out=n)
print(f”t: {t}”)
print(f”n: {n}”)
```

Datasets & Dataloaders

- PyTorch provides two data primitives:
  - torch.utils.data.DataLoader
  - torch.utils.data.Dataset
- Pre-loaded datasets
  - Image Datasets (such as Fashion-MNIST)
  - Text Datasets (such as DBpedia)
  - Audio Datasets (such as YESNO)

Fashion MNIST

- Fashion-MNIST is a dataset of Zalando’s article images consisting of a training set of 60,000 examples and a test set of 10,000 examples.
- Each example is a 28x28 grayscale image, associated with a label from 10 classes.

Loading a Dataset

- The FashionMNIST dataset with the following parameters
  - **root** is the path where the train/test data is stored,
  - **train** specifies training or test dataset
  - **download**=True downloads the data from the internet if it's not available at root
  - **transform** and **target_transform** specify the feature and label transformations

Loading a Dataset (cont'd)

```python
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Iteraing and Visualizing the Dataset

- The FashionMNIST dataset with the following parameters
  - **root** is the path where the train/test data is stored,
  - **train** specifies training or test dataset
  - **download**=True downloads the data from the internet if it’s not available at root
  - **transform** and **target_transform** specify the feature and label transformations

Iteraing and Visualizing the Dataset (cont'd)

```python
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}

figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

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_t
        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)
        return image, label
```

tshirt1.jpg, 0
tshirt2.jpg, 0
....
skirt99.jpg, 9

Preparing you data for training with DataLoaders

- DataLoader is an iterable that abstracts this complexity for us in an easy API

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Iterate through the DataLoader

```
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Transforms

- The FashionMNIST features are in PIL Image format, and the labels are integers.
- For training, we need the features as normalized tensor, and the labels as one-hot encoded tensors.
- To make these transformations, we use ToTensor and Lambda

```python
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).scatter_(0, torch.tensor(y), value=1))
)
```

Transforms (cont'd)

- ToTensor()
  - ToTensor() converts a PIL image or Numpy ndarray into FloatTensor and scales the image's pixel intensity values in the range [0., 1.]
- Lambda Transforms

```
target_transform = Lambda(lambda y: torch.zeros(
    10, dtype=torch.float).scatter_(dim=0, index=torch.tensor(y), value=1))
```

Build the Neural Network

- Build a neural network to classify images in the FashionMNIST dataset

```python
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Build the Neural Network (cont'd)

- Get Device for Training
  - To be able to train the model on GPU

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Using {} device'.format(device))
```

Build the Neural Network (cont'd)

- Define the Class

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

```
model = NeuralNetwork().to(device)
print(model)
```

Build the Neural Network (cont'd)

- Model Layers (Example)
  - For example, we will take a sample minibatch of 3 images of size 28 x 28

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

  - Flatten layer to convert each 2D 28x28 image into a contiguous array of 784 pixels values

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

Build the Neural Network (cont'd)

- Model Layers
  - Linear layer is a module that applies a linear transformation on the input using its stored weights and biases

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

  - Non-linear activations are what create the complex mappings between the model's inputs and outputs (we use ReLU)

```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

Build the Neural Network (cont'd)

- Model Layers
  - Sequential is an ordered container of modules.

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

  - Softmax: The last linear layer of the neural network return logits

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

Optimizing Model Parameters

- Hyperparameters
  - Number of Epochs
  - Batch Size
  - Learning Rate
- Optimization Loop
  - Train Loop
  - Test Loop
- Loss Function
  - We pass our model's output logit to nn.CrossEntropyLoss, which will normalize the logits and compute the predict error.

Optimizing Model Parameters (cont'd)

- Optimizer
  - Optimization is the process of adjust model parameters to reduce model error in each training step.
  - In this example we use Stochatic Gradient Descent (SGD) algorithm.
  - Available in PyTorch such as ADAM and RMSProp

Saving and Loading Models with Shapes

- Saving

```
torch.save(model, 'model.pth')
```

- Loading

```
model = torch.load('model.pth')
```

References

- Pytorch Tutorials
  - https://pytorch.org/tutorials/index.html
- Epoch vs Batch Size vs Iterations
  - https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9
- Fashion MNIST
  - https://research.zalando.com/project/fashion_mnist/fashion_mnist/