Image Classification via CNN

This tutorial teaches you how to classify images via convolutional neural network (CNN). You will learn how to create, train and evaluate a CNN by PyTorch. Part of this tutorial is adapted from Deep Learning with PyTorch: A 60 Minute Blitz.

1 Install PyTorch

PyTorch is an open source machine learning library, used for applications such as computer vision and natural language processing, primarily developed by Facebook’s AI Research lab. Please install it via following command.

1.1 On macOS

Please open your terminal, and install PyTorch via following command.

```
$ conda install pytorch torchvision torchaudio -c pytorch
```

Then test your installation and you shall see your torch version number, such as `1.10`.

```
$ python -c "import torch; print(torch.__version__)"
1.10.0
```

1.2 On Windows

Please open your anaconda prompt from Start menu, and install PyTorch via following command.

```
(base) C:\Users\%USERNAME%> conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

Then test your installation and you shall see your torch version number, such as `1.10`.

```
(base) C:\Users\%USERNAME%> python -c "import torch; print(torch.__version__)"
1.10.0
```

2 CIFAR10 Dataset

For this tutorial, we will use the CIFAR10 dataset. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of

32x32 pixels in size. Note that PyTorch take channels as first dimension by convention. This convention is different from other platform such as Pillow, Matlab, skimage, etc. They all put channels at last dimension.

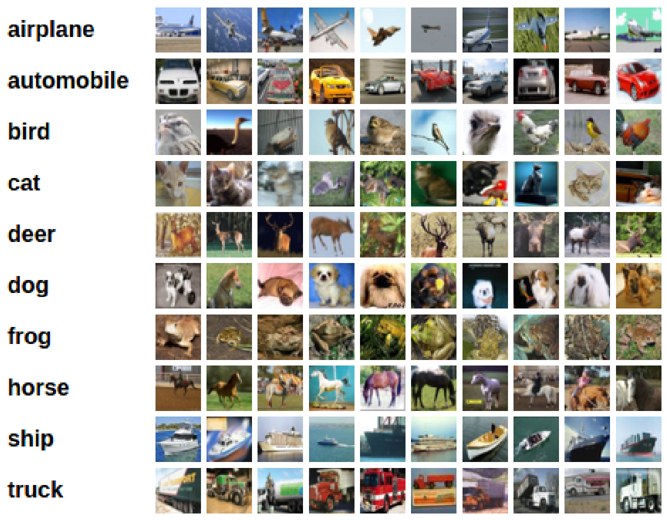


Figure 1. CIFAR10 dataset

We can load CIFAR10 from torchvision. It may take several minutes to download the dataset.

```
from torchvision.datasets import CIFAR10
from torchvision.transforms import ToTensor

trainset = CIFAR10(root='./data', train=True,
                   download=True, transform=ToTensor())

testset = CIFAR10(root='./data', train=False,
                  download=True, transform=ToTensor())
```

The dataset consists of two parts. One is train set, another one is test set. Usually, we train our model (CNN) on train set, and test the model on test set.

- Train: Show CNN the images, and tell it which classes they belong to. In such a way, we can teach it to distinguish different classes.
- Test: Show CNN the images, and ask it which classes they belong to. In such a way, we can test how well the CNN learns.

`trainset.classes` contains the all class names in order.

```
trainset.classes
# ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
         'truck']
```

Train set contains 50000 images. Let’s get the first image in train set, and show it.

```
len(trainset)               # 50000 images
image, label = trainset[0] # get first image and its class id
image.shape                 # 3 x 32 x 32
imshow(image)               # `imshow` is in cifar10.py
trainset.classes[label]     # 'frog'
```

You can see the image is of shape $3 \times 32 \times 32$, which means it has 3 channels, and 32×32 pixels.

Script `cifar10.py` already contains all code you need to load the dataset. In your program, all you need to do is

```
from dataset import load_cifar10, imshow
trainset, testset = load_cifar10()
```

Beside the dataset itself, we also need `DataLoader` objects to help us randomly load image batch by batch. Batch means a small collection of images. Here, we set `batch_size` to 4, so each batch contains 4 images.

```python
from torch.utils.data import DataLoader
trainloader = DataLoader(trainset, batch_size=4, shuffle=True)
testloader = DataLoader(testset, batch_size=4, shuffle=False)
```

Then we may iterate over the `DataLoader`, to get batches until the dataset is exhausted.

```python
for batch in trainloader:
    images, labels = batch
    print(images.shape) # [4, 3, 32, 32]
    print(labels.shape) # [4]
    break
```

`images` is of shape `[4, 3, 32, 32]`, which means it contains 4 images, each has 3 channels, and is of size 32x32. `labels` contains 4 scalars, which are the class IDs of this batch.

3 Create CNN Model

In this tutorial, we implement a simple but famous CNN model, LeNet-5. 5 means it contains 5 (convolutional or fully-connected) layers.

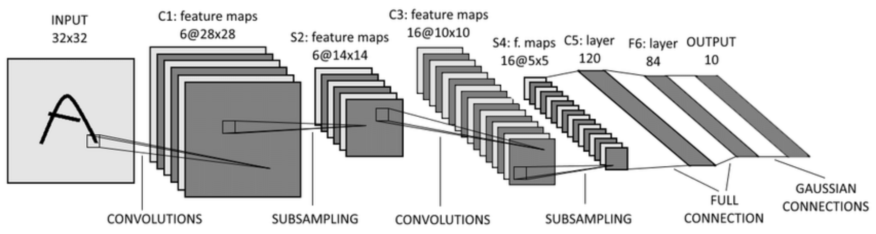


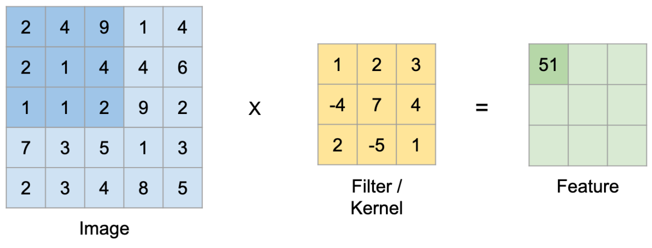
Figure 2. Architecture of LeNet-5

A typical CNN consists of these kinds of layer — convolutional layers, max pooling layers, and fully connected layers.

3.1 Convolutional layers

Convolutional layers are usually the first several layers. They perform a convolution on the output of last layer to extract features from the image. A convolutional layer has three architecture parameters:

- kernel_size $h \times w$: the size of the convolutional kernel.
- in_channels: the number of input channels
- out_channels: the number of output channels



In this layer, beside the convolutional kernel K, we also have a bias b added to each output channel. The formula for output is

$$X' = K * X + b,$$

where $*$ stands for convolution, X and X' are input and output. The total number of trainable parameters in convolutional layer is:

$$\underbrace{h \times w \times \text{in\_channels} \times \text{out\_channels}}_{\text{kernel}} + \underbrace{\text{out\_channels}}_{\text{bias}}$$

The convolution is performed without padding by default, so image size will shrink after convolution. If input image size is $H \times W$ and the kernel size is $h \times w$, the output will be of size

$$(H + 1 - h) \times (W + 1 - w).$$

Then we take channels and batch size into consideration, assume the input tensor has shape [batch_size, in_channels, H, W], then the output tensor will have shape

- input shape: [batch_size, in_channels, H, W]
- output shape: [batch_size, out_channels, H+1-h, W+1-w]

3.2 Activation

The output of convolutional layer and fully connected layer is usually "activated", i.e., transformed by a non-linear function, such as ReLU, sigmoid, tanh, etc. Activation functions are all scalar function. They do not change the tensor shape, but only map each element into a new value. They usually contain no trainable parameters.

In this tutorial, we choose perhaps the most popular activation function, $ReLU(x) = \max(0, x)$.

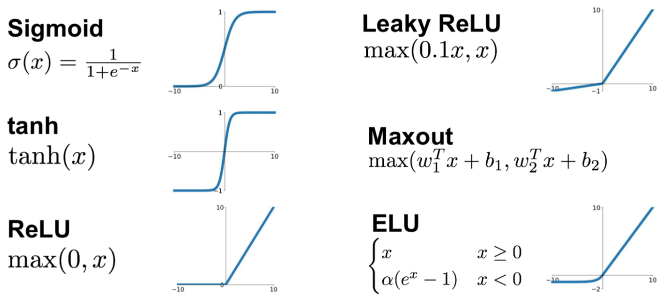


Figure 3. Activation Functions

Here we demonstrate how to create the first convolutional layer of LeNet-5 by PyTorch. This layer has kernel size 5x5 and its output contains 6 channels. Its input is the original RGB images, so `in_channels=3`. The output is activated by ReLU (Original paper uses tanh).

```
import torch.nn as nn
# convolutional layer 1
conv_layer1 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(5,5)),
    nn.ReLU(),
)
```

3.3 Pooling layers (subsampling)

Pooling usually follows a convolutional layer. There are two kinds of pooling layer — maximum pooling and average pooling. Maximum pooling computes the maximum of small local patches, while average pooling computes the average of small local patches.

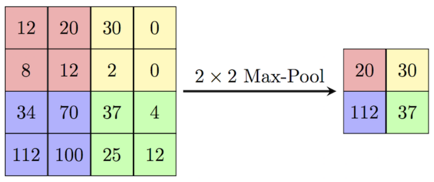


Figure 4. Max pooling with kernel size 2x2

The kernel size of a pooling layer is the size of local patches. Assume the input image is of size $H \times W$ and the kernel size is $h \times w$, the output of pooling layer will be of size

$$\frac{H}{h} \times \frac{W}{w}.$$

Then we take channels and batch size into consideration, input tensor and output tensor will have shape:

- input shape: [batch_size, in_channels, H, W],
- output shape: [batch_size, in_channels, H/h, W/w].

Pooling layers do not change the number of channels and do not contain any trainable parameters.

This code snip demonstrates how to create a 2x2 max pooling layer.

```
max_pool = nn.MaxPool2d(kernel_size=(2,2))
```

3.4 Fully connected (FC) layers

Fully connected (FC) layers are usually the last several layers. They take the features conv layers produced and output the final classification result. Before go into FC layers, we need to “flatten” the intermediate representation produced by convolutional layers. The output of CNN is a 4D tensor of shape [batch_size, channels, H, W]. After flattened, it becomes a 2D tensor of shape [batch_size, channels*H*W]. This 2D tensor is exactly what FC layers consumes as input.

4D tensor of shape [batch_size, channels, H, W]

|
flatten
|
v
2D tensor of shape [batch_size, channels*H*W]

A FC layer has two architecture parameter — input features and output features:

- in_features: the number of input features,
- out_features: the number of output features.

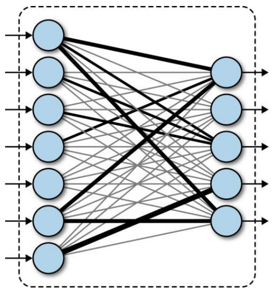


Figure 5. FC layer with 7 input features and 5 output features

The input and output of FC layers are of shape:

- input shape: [batch_size, in_features]
- output shape: [batch_size, out_features]

The formula for output is

$$X' = \Theta X + b$$

where Θ is weights, and b is biases. Because there is a weight between any input feature and any output feature, Θ is of shape $\text{in\_features} \times \text{out\_features}$. Number of biases is equal to the number of output features. Each output feature is added by a bias. In total, the number of trainable parameters in a FC layer is

$$\underbrace{\text{in\_features} \times \text{out\_features}}_{\text{weights } \Theta} + \underbrace{\text{out\_features}}_{\text{bias}}.$$

This example shows how to create a FC layer in PyTorch. The created FC layer has 120 input features and 84 output features, and its output is activated by ReLU.

```
fc_layer = nn.Sequential(
    nn.Linear(in_features=120, out_features=84),
    nn.ReLU(),
)
```

The last layer of our CNN is a little bit special. First, it is not activated, i.e., no ReLU. Second, its output features must be equal to the number of classes. Here, we have 10 classes in total, so its output features must be 10.

```
output_layer = nn.Linear(in_features=84, out_features=10)
```

3.5 Create LeNet-5

LeNet-5 is a simple but famous CNN model. It contains 5 (convolutional or fully-connected) layers. Here, we choose it as our CNN model. Its architecture is shown in figure 6.

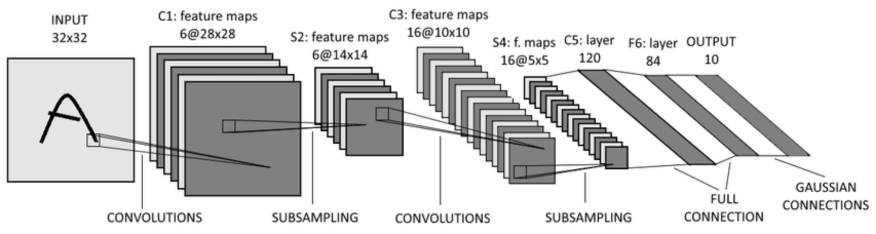


Figure 6. Architecture of LeNet-5

The layers of LeNet-5 are summarized here:

0. Input image: 3x32x32
1. Conv layer:
   - kernel_size: 5x5
   - in_channels: 3
   - out_channels: 6
   - activation: ReLU
2. Max pooling:
   - kernel_size: 2x2
3. Conv layer:
   - kernel_size: 5x5
   - in_channels: 6
   - out_channels: 16
   - activation: ReLU
4. Max pooling:
   - kernel_size: 2x2
5. FC layer:
   - in_features: 16*5*5
   - out_features: 120
   - activation: ReLU
6. FC layer:
   - in_features: 120
   - out_features: 84
   - activation: ReLU
7. FC layer:
   - in_features: 84
   - out_features: 10 (number of classes)

`model.py` create LeNet-5 by PyTorch. First, we create the 2 convolutional layers:

```
import torch.nn as nn

# convolutional layer 1
conv_layer1 = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(5,5)),
    nn.ReLU()),
)
# convolutional layer 2
conv_layer2 = nn.Sequential(
    nn.Conv2d(in_channels=6, out_channels=16, kernel_size=(5,5)),
    nn.ReLU()),
)
```

Then, the 3 fully connected layers:

```
# fully connected layer 1
fc_layer1 = nn.Sequential(
    nn.Linear(in_features=16*5*5, out_features=120),
    nn.ReLU(),
)
# fully connected layer 2
fc_layer2 = nn.Sequential(
    nn.Linear(in_features=120, out_features=84),
    nn.ReLU(),
)
# fully connected layer 3
fc_layer3 = nn.Linear(in_features=84, out_features=10)
```

Finally, combine them as LeNet-5. Don't forget flatten layer before FC layers.

```
LeNet5 = nn.Sequential(
    conv_layer1,
    nn.MaxPool2d(kernel_size=(2,2)),
    conv_layer2,
    nn.MaxPool2d(kernel_size=(2,2)),
    nn.Flatten(), # flatten
    fc_layer1,
    fc_layer2,
    fc_layer3
)
```

4 Train the Model

After create the network, we will teach it how to distinguish images between different classes. Intuitively, the teaching is achieved by showing it the images in train set, and telling it which classes they belong to. The network will gradually learn the concepts, such as 'bird', 'cat', 'dog', etc., just like how human children learn. This part of code is in `train.py`.

First, we import our model `LeNet5`, and define the loss function and optimization method. Here, we use cross-entropy loss, which is designed for classification tasks. This loss measure how similar your prediction is to the correct answer (ground truth). The closer your prediction is to the correct one, the smaller this loss is. To minimize this loss, we need an optimizer. Here, we use stochastic gradient descent (SGD) method as optimizer.

```
from model import LeNet5
model = LeNet5

loss_fn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

When training a network, the most important parameter is learning rate. In above example, learning rate `lr` is 0.001. To train model successfully, you need a proper learning rate. If learning rate is too small, your loss will converge very slowly. If learning rate is too big, loss may not converge at all.

Then we start training. The training usually takes minutes to hours. Once you finish looping over the dataset one time, you finished one epoch. A successful train usually has multiple epochs. Following example trains the network for 10 epochs.

```
# training
num_epoch = 10
for epoch in range(num_epoch):
    running_loss = 0.0
    for i, batch in enumerate(trainloader, 0):
        # get the images; batch is a list of [images, labels]
        images, labels = batch

        optimizer.zero_grad() # zero the parameter gradients

        # get prediction
        outputs = model(images)

        # compute loss
        loss = loss_fn(outputs, labels)

        # reduce loss
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 500 == 499:  # print every 500 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 500))
            running_loss = 0.0

print('Finished Training')
```

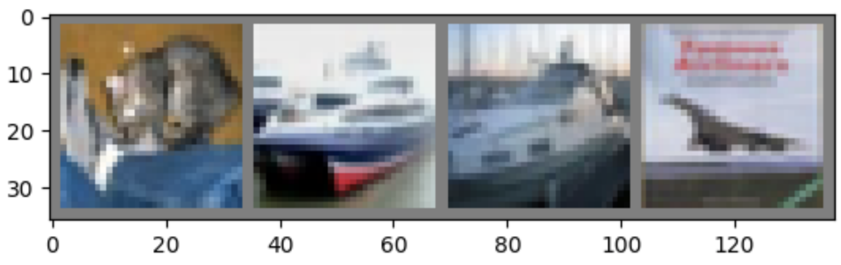
5 Test the Model

After training, our model can classify images now. At first, we show it several images in test set to see if it can correctly recognize them.

```
dataiter = iter(testloader)
images, labels = dataiter.next()
predictions = model(images).argmax(1)

# show some prediction result
classes = trainset.classes
print('GroundTruth: ', ' '.join('%5s' % classes[i] for i in labels))
print('Prediction: ', ' '.join('%5s' % classes[i] for i in predictions))
imshow(torchvision.utils.make_grid(images.cpu()))
```

You will see images and output like this. Due to randomness, your results may be different.



```
GroundTruth:    cat  ship  ship plane
Prediction:    cat  ship airplane airplane
```

Next, let us look at how the model performs on the whole dataset.

```
@torch.no_grad()
def accuracy(model, data_loader):
    model.eval()
    correct, total = 0, 0
    for batch in data_loader:
        images, labels = batch
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    return correct / total

train_acc = accuracy(model, trainloader) # accuracy on train set
test_acc = accuracy(model, testloader)  # accuracy on test set

print('Accuracy on the train set: %f %%' % (100 * train_acc))
print('Accuracy on the test set: %f %%' % (100 * test_acc))
```

The output looks like

```
Accuracy on the train set: 72.17 %
Accuracy on the test set: 61.37 %
```

Since we trained for only 10 epochs, the accuracy is not very high.

Finally, we can save our model to a file:

```
torch.save(LeNet5.state_dict(), 'model.pth')
```

6 Load Predefined CNNs

Besides build your own CNN from scratch, you may use predefined networks in PyTorch Hub. Predefined models have two major advantages.

- These models were searched out and tested by previous researchers, and usually performs better than the one you build from scratch.
- All of them are already pre-trained for specific vision tasks, such as image classification, object detection, face recognition. Usually, you only need to fine-tune the model a little bit to fit your dataset. If you are lucky enough, some model may perfect fit the task you are working on without further training.

Here we take ResNet18 as an example to show how to use predefined models. You may read its document for more information.

First, load ResNet18 from PyTorch Hub.

```
model = torch.hub.load('pytorch/vision:v0.10.0',
                       'resnet18',
                        pretrained=True)
```

You may print the model to check its architecture.

```
>>> print(model)
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): ...
  (relu): ...
  (maxpool): ...
  (layer1): ...
  (layer2): ...
  (layer3): ...
  (layer4): ...
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

The out is quite long, but we only care about its first layer and output layer. Its first layer `conv1` has `in_channels=3`, which means it was designed for colour images. If you want to apply it to grey images, the first layer need to replaced by one with `in_channels=1`.

```
model.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
          bias=False)
```

Its output layer `fc` has `out_features=1000`, which means it was trained on a Dataset with `1000` classes. If you want to apply it to other dataset, such as CIFAR10, the output layer need to replaced by one with `out_features=10`, because CIFAR10 only have 10 classes.

```
model.fc = nn.Linear(in_features=512, out_features=10, bias=True)
```

According to the document of ResNet, the model expect input images of shape `224x224`. Before we feed images into the model, we need to resize images to 224x224.

```
from torchvision import transforms
preprocess = transforms.Resize(256)

for i, batch in enumerate(trainloader, 0):
    images, labels = batch

    # resize to fit the input size of resnet18
    images = preprocess(images)

    # feed into model
    optimizer.zero_grad()
    outputs = model(images)

    # compute loss, back propagation, etc.
    ...
```

Which model should I use?

First, you need to know the task you are working on. For example, if you are developing an object detection system, then you should only consider those model developed for object detection, such as YOLOv5. For image recognition, you should consider ResNet, AlexNet, etc.

Second, the predefined model usually have several variants of different size. For example, ResNet has five variants – ResNet18, ResNet34, ResNet50, ResNet101, ResNet152, which contain 18, 34, 50, 101, 152 layers perspectively. Bigger models have more parameters and modelling capability, but consumes more memory, more computational resources and more power. Generally, You should choose big model for difficult tasks and big datasets, and choose small model for easy tasks and small datasets.

7 GPU Acceleration

GPU acceleration plays an important role in reducing training time of CNNs. Modern CNNs usually contains tons of trainable parameters and are extremely computationally hungry. It takes hours to days, or even weeks to training a perfect CNN model. GPU acceleration technique could speed up training by 10-100 times, compared to CPU. Figure 7 shows a typical GPU acceleration performance. The acceleration is even more significant with large batch-size.

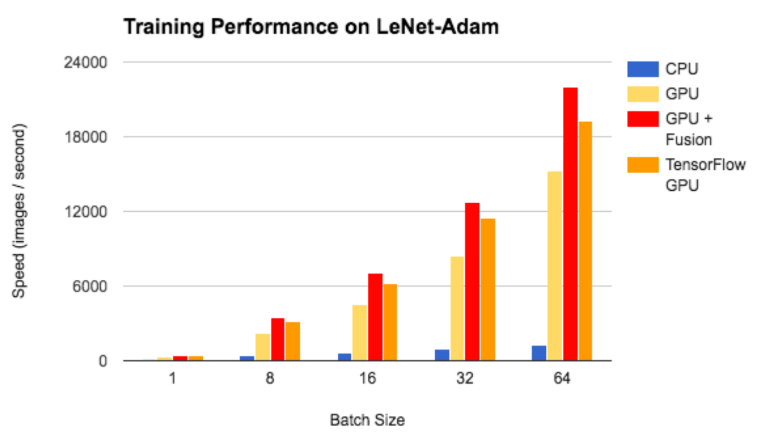


Figure 7. Typical GPU acceleration against CPU.

7.1 Install CUDA

To get GPU acceleration, you need to have a computer with NVIDIA GPU equipped and have `cudatoolkit` installed. If your own PC does not have NVIDIA GPU, but you still want GPU acceleration, you may use computers in COMP laboratories. All PCs in COMP laboratories are equipped with NVIDIA GPU. Please use following command to install `cudatoolkit`.

```
$ conda install cudatoolkit=11.3 -c pytorch
```

Then test your `PyTorch` to check your installation.

```
$ python -c "import torch; print(torch.cuda.is_available())"
True
```

You shall see output `True`.

7.2 Adapt your code for GPU

You also need to modify your code to get the acceleration. Specifically, you need to move your model and data to GPU.

Move model to GPU:

```
device = torch.device('cuda:0') # get your GPU No. 0
model = model.to(device)        # move model to GPU
```

Move data to GPU:

```
# get some image from loader
dataiter = iter(testloader)
images, labels = dataiter.next()

# move it to GPU
images = images.to(device)
labels = labels.to(device)
```

Get the prediction as usual, but the computation is done by GPU and thus faster.

```
# get prediction as usual
predictions = model(images).argmax(1).detach()

# or perform one-step training, if you are training the model
optimizer.zero_grad()
outputs = model(images)
loss = loss_fn(outputs, labels)
loss.backward()
optimizer.step()
```

Finally, if you want to print the result, you may transfer the result back to CPU:

```
# transfer results back to CPU and so we can print it
predictions = predictions.cpu()
print(predictions)
```

However, above code only works on PCs with GPU. To let our code be able to work, no matter GPU is equipped or not, we usually define `device` as:

```
if torch.cuda.is_available():
    # If GPU is available, use gpu.
    device = torch.device('cuda:0')
else:
    # If not, use cpu.
    device = torch.device('cpu')
```

8 Assignment

8.1 Handwritten digit recognition

Modify `train.py` , and `model.py` to train a CNN to recognize hand-written digits in MNIST datasets.

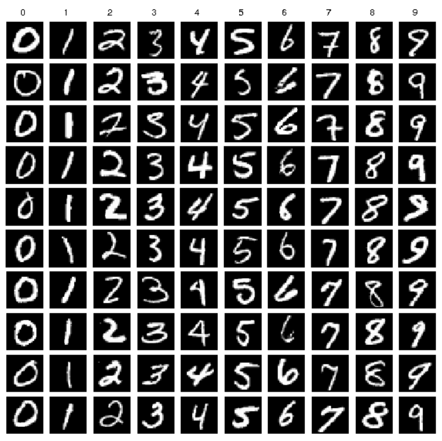


Figure 8. Example images from MNIST

MNIST contains images of digits 0-9 written by human. The task is to recognize which digit the image represent. All image are grey scale (only 1 channel), and contains 28x28 pixels.

Your CNN should contain following layers in order.

0. Input image: 1x28x28
1. Conv layer:
   - kernel_size: 5x5
   - out_channels: 16
   - activation: ReLU
2. Max pooling:
   - kernel_size: 2x2
3. Conv layer:
   - kernel_size: 3x3
   - out_channels: 32
   - activation: ReLU
4. Max pooling:
   - kernel_size: 2x2
5. Conv layer:
   - kernel_size: 1x1
   - out_channels: 8
   - activation: ReLU
6. FC layer:
   - out_features: 64
   - activation: ReLU
7. FC layer:
   - out_features: ?? (to be inferred by you)
   - activation: None

Tasks:

1. Answer following questions first:
   - Assume we set `batch_size=8` , what shape do above 7 layers' inputs and outputs have?
   - How many trainable parameters each layer contains?
2. Train a CNN to recognize hand-written digits in MNIST datasets.
   - Modify `model.py` to create a CNN with architecture specified above.
   - Modify `train.py` and `dataset.py` to train it on MNIST dataset. Your model should achieve an accuracy higher than 95% on test set. Usually, more than 3 epochs are enough to achieve this accuracy. Save your model to file `model.pth` .

8.2 Fashion-MNIST

MNIST is too easy. Convolutional nets can achieve 99%+ accuracy on MNIST. Fashion-MNIST, containing ten different classes of clothes, is more challenging than MNIST. Your task is to load predefined CNN - ResNet18 from PyTorch Hub, and train it to achieve 90+% classification accuracy on Fashion-MNIST. Please modify `fashion_mnist.py` to complete this part.

Tips:

- You may use `load_fashion_mnist()` in `dataset.py` to load Fashion-MNIST.
- You need to resize images in Fashion-MNIST to fit the input size of ResNet18. The input and output layer of the network also need modification to fit Fashion-MNIST.
- Please save your model to `fashion_mnist.pth`.

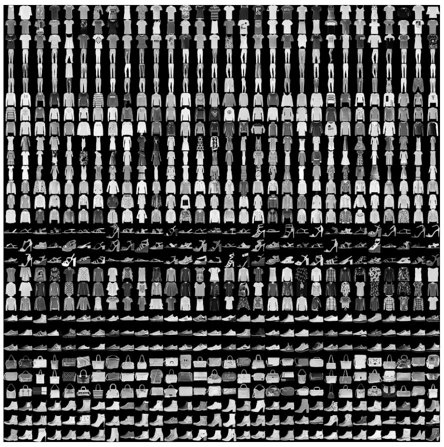


Figure 9. Example images from Fashion-MNIST

8.3 Submission instruction

Your submission should include:

1. An report, containing
   - your analysis and answer to above questions.
   - screenshot of your program output (please print your student ID in the output log to verify the screenshot is from your work).
2. All python source file.
3. The saved models – `model.pth` and `fashion_mnist.pth`.

Important Note: Please carry out your project independently! Your answer should NOT be identical or significantly similar to the answers from any of your classmates. Once your answer is found suspicious, all students with similar answers will be subject to investigation of academic dishonesty and may receive penalty for any misconducts.