```python
import torch # pytorch main library
import torchvision # computer vision utilities
import torchvision.transforms as transforms # transforms used in the pre-processing of the data

import torch.nn as nn
import torch.nn.functional as F

import matplotlib.pyplot as plt
import numpy as np

import torch.optim as optim

# Check if GPU is available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:
print(device)
```

This code imports several libraries, including PyTorch (torch), a library for building machine lear ning models, and torchvision, a library containing computer vision utilities. The torchvision.transf orms module is also imported, which contains preprocessing functions for image data. The code also imports several PyTorch modules, including torch.nn, which contains classes for building n eural networks, and torch.nn.functional, which contains functions for building neural network lay ers. The code also imports the matplotlib and numpy libraries for plotting and numerical operatio ns, respectively.

The code then checks if a GPU (Graphics Processing Unit) is available for use. If a GPU is avail able, the device variable is set to 'cuda:0', which tells PyTorch to run computations on the GPU. If a GPU is not available, the device variable is set to 'cpu', which tells PyTorch to run computati ons on the CPU.

Finally, the code prints the value of the device variable, which should indicate whether the comp utations will be run on the GPU or the CPU.

```python
# Function to get thge statistics of a dataset
def get_dataset_stats(data_loader):
    mean = 0.
    std = 0.
    nb_samples = 0.
    for data in data_loader:
        data = data[0] # Get the images to compute the stgatistics
        batch_samples = data.size(0)
        data = data.view(batch_samples, data.size(1), -1)
        mean += data.mean(2).sum(0)
        std += data.std(2).sum(0)
        nb_samples += batch_samples

    mean /= nb_samples
    std /= nb_samples
    return mean,std

```

```python
# functions to show an image
def imshow(img,stats):
    img = img *stats[1] + stats[0]     # unnormalize
    npimg = img.numpy() # convert the tensor back to numpy
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

This code defines two functions: get_dataset_stats and imshow.

The get_dataset_stats function takes a single argument, data_loader, which is a PyTorch DataLoader object that iterates over a dataset and returns a batch of data. The function initializes three variables: mean, std and nb_samples. The mean and std variables will be used to store the mean and standard deviation of the dataset's data, respectively. The nb_samples variable will be used to count the total number of samples in the dataset.

The function then iterates over the data in the data_loader, which is a batch of data, and for each iteration, it does the following:

1. It extracts the images from the batch of data, which is stored in the variable data.
2. It gets the number of samples in the batch, stored in the variable batch_samples.
3. It reshapes the data from 3D tensor to 2D tensor by using the view method and flattening the last dimension.
4. It accumulates the mean of each channel over all samples in the batch.
5. It accumulates the standard deviation of each channel over all samples in the batch.
6. It increments the total number of samples by the number of samples in the current batch.

After all the batches have been processed, the function calculates the mean and standard deviation by dividing the accumulated mean and standard deviation by the total number of samples, respectively. Finally, the function returns the mean and standard deviation of the dataset's data.

The imshow function takes two arguments, img and stats. The img argument is a PyTorch tensor containing an image, and the stats argument is a tuple of two elements, which are the mean and standard deviation of the dataset's data, respectively. The function does the following:

1. It un-normalizes the image by multiplying it with the standard deviation and adding the mean.
2. It converts the tensor back to a numpy array using the numpy() method.
3. It transposes the numpy array so that the channels are in the last dimension.
4. It shows the image using the plt.imshow() function from the matplotlib library.
5. It shows the plot using the plt.show() function from the matplotlib library.

The imshow function allows to display the image in its original format, thus the mean and std are used to un-normalize the image before displaying it.

```
batch_size = 256

transform = transforms.Compose(
    [transforms.ToTensor()]) # Convert the data to a PyTorch tensor

# Load develpoment dataset
devset = torchvision.datasets.MNIST(root='./data', train=True,
                                       download=True, transform = transform)

train_set_size = int(len(devset) * 0.8)
val_set_size = len(devset) - train_set_size

# Split the development set into train and validation
trainset, valset = torch.utils.data.random_split(devset, [train_set_size,
val_set_size], generator=torch.Generator().manual_seed(42))


# Get the data loader for the train set
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

# Comopute the statistics of the train set
stats = get_dataset_stats(trainloader)
print("Train stats:", stats)
# Pre-processing transforms
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((stats[0]), (stats[1]))])


# Load the development set again using the proper pre-processing transforms
devset = torchvision.datasets.MNIST(root='./data', train=True,
                                       download=True, transform = transform)

# Split the development set into train and validation
trainset, valset = torch.utils.data.random_split(devset, [train_set_size,
val_set_size], generator=torch.Generator().manual_seed(42))

# Get the data loader for the train set
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

# Get the data loader for the test set
valloader = torch.utils.data.DataLoader(valset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

# Get the test set
testset = torchvision.datasets.MNIST(root='./data', train=False,
                                      download=True, transform=transform)

# Get the data loader for the test set
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('zero', 'one', 'two', 'three',
           'four', 'five', 'six', 'seven', 'eight', 'nine')
```

This code is for loading, splitting, normalizing and pre-processing the MNIST dataset for training, validation and testing a model. It starts by defining the batch size as 256, which means that the dataset will be divided into batches of 256 samples each.

Then, it defines a transform using the torchvision.transforms.Compose function, which applies multiple pre-processing steps to the data. In this case, it applies the torchvision.transforms.ToTensor() function, which converts the input data from a PIL image to a PyTorch tensor.

Next, it loads the MNIST dataset, which is a dataset of handwritten digits, using the torchvision.datasets.MNIST function. The root argument is set to './data', which is the directory where the dataset will be downloaded and stored. The train argument is set to True, which means that the dataset will be loaded as a training dataset. The download argument is set to True, which means that the dataset will be downloaded if it is not already present in the specified directory. The transform argument is set to the transform defined earlier.

It then splits the dataset into a train and validation set by using the torch.utils.data.random_split() function. The first argument is the dataset, the second argument is a list containing the size of the train set and the validation set, and the third argument is a generator that will use a manual seed of 42 to generate random numbers.

Next, it creates PyTorch DataLoader objects for the train, validation and test sets using the torch.utils.data.DataLoader function. The first argument is the corresponding dataset, the second argument is the batch size, the third argument is whether to shuffle the data, and the fourth argument is the number of worker threads to use for loading the data.

Then, it computes the statistics of the train set using the get_dataset_stats function, which returns a tuple of mean and std. Then, it normalizes the train, validation and test sets using the computed mean and std and the torchvision.transforms.Normalize() function, which normalizes the data by subtracting the mean and dividing by the standard deviation.

Finally, it defines a tuple of classes containing the names of the digits (zero, one, two, ..., nine) that the model will be trying to classify.

```python
# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images[:8]), stats)
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(8)))
```

This code is used to display some random training images from the MNIST dataset and their corresponding labels.

First, it creates an iterator for the training data using the iter() function and the trainloader object. Then, it uses the next() function to get the next batch of data from the iterator, which is stored in the variables images and labels.

Then, it uses the torchvision.utils.make_grid function to create a grid of images from the batch of images, and the function takes the first 8 images from the batch. It then calls the imshow function with the grid of images and the statistics of the dataset, to display the images in their original format.

Finally, it uses the join() function to concatenate the labels of the images, which are the classes of the digits, and it prints them. The f'{classes[labels[j]]:5s}' is used to format the output as a string of 5 characters and it uses the index of the labels to get the class name from the classes tuple.

```python
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
net.to(device)
```

This is a PyTorch implementation of a simple feedforward neural network. The Net class is a subclass of PyTorch's nn.Module class and it defines the architecture of the network. The __init__ method of the Net class initializes three fully connected layers, fc1, fc2 and fc3, using PyTorch's nn.Linear module. The input and output sizes of these layers are specified as arguments to the nn.Linear module, with the input size of the first layer being 28*28 (the size of a single image in the MNIST dataset) and the output size of the last layer being 10 (the number of classes in the MNIST dataset).

The forward method of the Net class defines the forward pass of the network. It takes an input tensor x and applies a series of operations to it: x is first flattened using torch.flatten so that it can be passed to the first linear layer, fc1. Then, the output of fc1 is passed through a rectified linear unit (ReLU) activation function using F.relu. This process is repeated for the second and third linear layers, fc2 and fc3. Finally, the output of the last linear layer fc3 is returned.

The last line of the code creates an instance of the Net class and assigns it to the variable net. The net.to(device) line moves the model to the specified device (e.g. 'cuda' for GPU)

```python
criterion = nn.CrossEntropyLoss() # Loss function
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) # Optimizer used for training
```

The code above sets up the training process for the neural network defined earlier.

The first line initializes the loss function, which is used to measure the difference between the predicted and actual output during training. In this case, the nn.CrossEntropyLoss function is used, which is a commonly used loss function for classification tasks

The second line initializes the optimizer, which is used to update the network's parameters during training. The optim.SGD function is used, which stands for stochastic gradient descent. The first argument passed to optim.SGD is the parameters of the network that need to be optimized (in this case, net.parameters()), the second argument lr is the learning rate, which determines the step size at which the optimizer makes updates to the parameters during training, a smaller learning rate means the optimizer will make smaller updates and the training process will be slower. The third argument momentum is a parameter used in the optimizer that helps the optimizer converge faster by averaging the gradients across multiple steps.

This optimizer and loss function combination is a common choice for training neural networks, but other options such as Adam, Adagrad, and RMSprop are also commonly used.

```python
nepochs = 20
PATH = './cifar_net.pth' # Path to save the best model

best_loss = 1e+20
for epoch in range(nepochs):  # loop over the dataset multiple times
    # Training Loop
    train_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
    print(f'{epoch + 1},  train loss: {train_loss / i:.3f},', end = ' ')

    val_loss = 0
    # since we're not training, we don't need to calculate the gradients for our outputs
    with torch.no_grad():
        for i, data in enumerate(valloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data[0].to(device), data[1].to(device)

            outputs = net(inputs)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
        print(f'val loss: {val_loss / i:.3f}')

        # Save best model
        if val_loss < best_loss:
            print("Saving model")
            torch.save(net.state_dict(), PATH)

print('Finished Training')
```

This code trains the neural network defined earlier using the training data and the optimizer and loss function initialized earlier. The training process is repeated for a specified number of epochs (in this case, 20).

The outer loop iterates over the number of epochs. In each iteration, the training loss is computed by iterating over the training data using the PyTorch DataLoader, trainloader. For each training sample, the gradients of the network's parameters are set to zero using optimizer.zero_grad(), the forward pass is performed using outputs = net(inputs), the loss is calculated using the loss function criterion(outputs, labels), the gradients are computed using loss.backward(), and the optimizer updates the parameters using optimizer.step(). The training loss is accumulated over the entire training set and printed out at the end of each epoch.

The validation loss is also computed by iterating over the validation data using the PyTorch DataLoader, valloader. The validation loss is accumulated over the entire validation set and printed out at the end of each epoch

If the validation loss is smaller than the best validation loss seen so far, the current model is saved to the specified file path './cifar_net.pth' using the torch.save function.

After the training is complete, the script prints 'Finished Training' to indicate that the training process is done.

```python
# Load the best model to be used in the test set
net = Net()
net.load_state_dict(torch.load(PATH))
```

This code loads the best model that was saved during the training process.

It first creates an instance of the Net class, which defines the architecture of the network. Then, the load_state_dict method is used to load the saved model's parameters, which were previously saved to the file path specified in the training code PATH = './cifar_net.pth'. The torch.load function is used to load the saved model's parameters from the file.

After this point, the network is ready to be used for prediction on new data. For example, the test set can be passed through the network and the accuracy of the model can be calculated on the test set.

```python
dataiter = iter(testloader)
images, labels = next(dataiter)


```

```python
# print images
imshow(torchvision.utils.make_grid(images[:4]), stats)
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
outputs = net(images)
```

This code loads a batch of test data from the testloader (a PyTorch DataLoader that was previously defined and used to load the test data), and assigns the first batch of images and labels to the variables images and labels.

The function imshow(torchvision.utils.make_grid(images[:4]), stats) is used to display the first 4 images in the batch using the torchvision.utils.make_grid function to create a grid of images and the imshow function to display the grid.

The line print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4))) prints the true labels of the first 4 images in the batch. classes is a list of class names and labels[j] is the index of the true class for the j-th image.

The final line outputs = net(images) applies the neural network to the batch of images and assigns the output of the network to the variable outputs. The outputs will be a tensor of shape (batch_size, number of classes) representing the predicted class scores for each image in the batch.

```python
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

This code performs a prediction on the batch of test images using the loaded neural network.

The torch.max(outputs, 1) function is used to find the class with the highest predicted score for each image in the batch. The first element of the returned tuple is the maximum values and the second element is the indices of the maximum values along the specified dimension (in this case, dimension 1, which corresponds to the classes)

The predicted variable contains the class indices of the highest predicted scores. These predicted class indices are then used to print the predicted class names of the first 4 images in the batch using the classes list, which maps class indices to class names.

It is possible to calculate the accuracy of the model by comparing the predicted labels with the true labels and counting the number of correct predictions.

```python
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct / total} %')
```

This code calculates the accuracy of the neural network on the test dataset.

The accuracy is calculated by comparing the predicted labels, obtained by finding the class with the highest predicted score for each image in the batch using torch.max(outputs.data, 1), with the true labels.

The outer loop iterates over the test data using the testloader PyTorch DataLoader. For each batch of test data, the network is run on the images to obtain predicted scores, the predicted class is obtained by taking the argmax of the predicted scores and the predicted class is compared to the true labels.

The number of correct predictions is accumulated in the correct variable, and the total number of predictions is accumulated in the total variable

Finally, the accuracy is calculated by dividing the number of correct predictions by the total number of predictions and printing the result with a message.

It is worth noting that the torch.no_grad() context manager is used when evaluating the model on the test set, since we don't need gradients for the forward pass during evaluation, this will reduce the memory usage and speed up the evaluation process.

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

This code calculates the accuracy of the neural network on the test dataset for each class separately.

It prepares to count the correct and total predictions for each class using a dictionary comprehension.

The outer loop iterates over the test data using the testloader PyTorch DataLoader. For each batch of test data, the network is run on the images to obtain predicted scores, the predicted class is obtained by taking the argmax of the predicted scores. Then it collects the correct predictions for each class by comparing the predicted class to the true label.

Finally, the accuracy for each class is calculated by dividing the number of correct predictions by the total number of predictions for that class, and then printing the result with a message.

This way you can see how well the model is performing on each individual class, which can be useful for identifying potential issues or areas for improvement in the model.