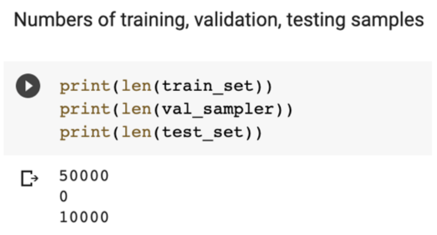
DS340W – Lab2
Daehoon Gwak

# Part A: ResNet for Image Classification on CIFAR-10

## 1. Data preparation

For this part, we have used the CIFAR-10 dataset. We have used 10 classes and the size of the images are 3X32X32. Then we divided up the dataset into training, validation, and testing sets and get the samples. Especially, in this Lab, we have 50,000 samples for training set and 10,000 samples for testing set and don`t assign any sample for validation.

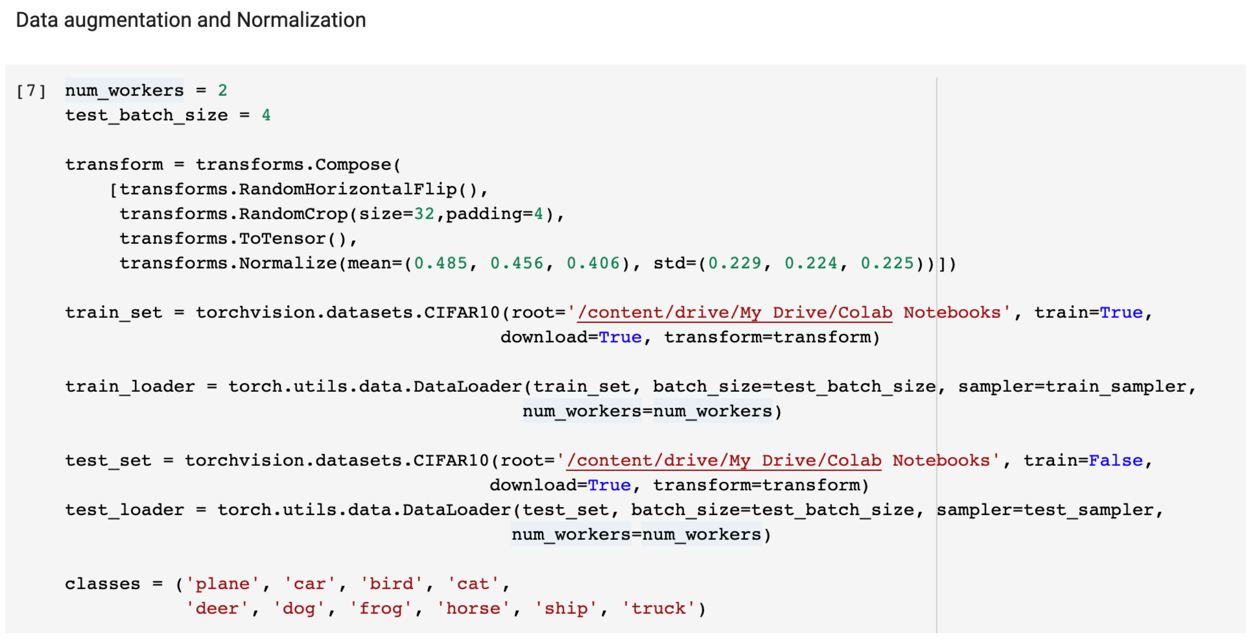

Numbers of training, validation, testing samples

```
print(len(train_set))
print(len(val_sampler))
print(len(test_set))
```

```
50000
0
10000
```

*Figure 1. Number of samples*

Then, we load the dataset using augmentation and normalization method.


Data augmentation and Normalization

```
[7] num_workers = 2
    test_batch_size = 4

    transform = transforms.Compose(
        [transforms.RandomHorizontalFlip(),
         transforms.RandomCrop(size=32,padding=4),
         transforms.ToTensor(),
         transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225))])

    train_set = torchvision.datasets.CIFAR10(root='/content/drive/My Drive/Colab Notebooks', train=True,
                                            download=True, transform=transform)

    train_loader = torch.utils.data.DataLoader(train_set, batch_size=test_batch_size, sampler=train_sampler,
                                              num_workers=num_workers)

    test_set = torchvision.datasets.CIFAR10(root='/content/drive/My Drive/Colab Notebooks', train=False,
                                           download=True, transform=transform)
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=test_batch_size, sampler=test_sampler,
                                             num_workers=num_workers)

    classes = ('plane', 'car', 'bird', 'cat',
               'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

*Figure 2. Data augmentation and Normalization*

For data augmentation, We`ve used *torchvision.transforms* functions including *RandomHorizontalFlip(), RandomCrop()*. *RandomHorizontalFlip* is to flip the image horizontally and *RandomCrop* is to crop the image randomly with size 32. For normalization, We`ve used transforms.Normalize function with the given mean and standard deviation values.

## 2. Network setting

Our next step is to construct the Network and we adopted the ResNet model(also known as Residual neural network). Previously, people have used multiple layers to optimize the model, however, it ruins the model when we use the deeper model. So, people now use this ResNet model which uses network layers to fit a residual mapping, instead of directly trying to fit a desired underlying mapping. Below Figure 3 is the diagram for our Lab's ResNet architecture.

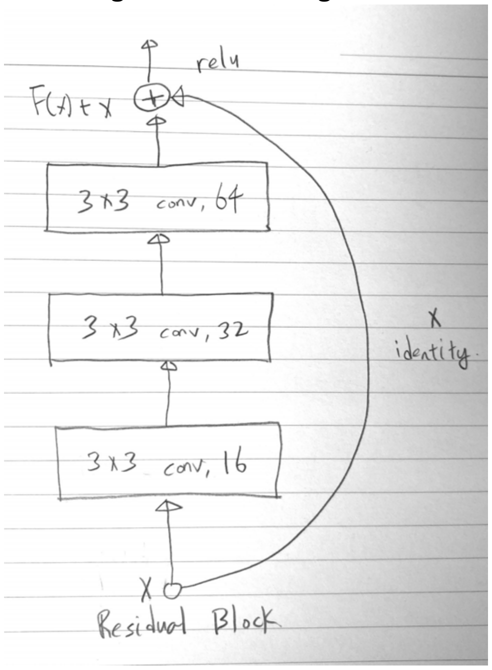


*Figure 3. ResNet Architecture*

Then, we defined the loss function by using Cross-Entropy loss and the optimizer with Adam. Cross-Entropy loss function checks the performance of the classification where its output is a probability between 0 and 1. As the predicted probability increases to 1, the loss value decreases exponentially and vice versa. We can also write in mathematical form as:

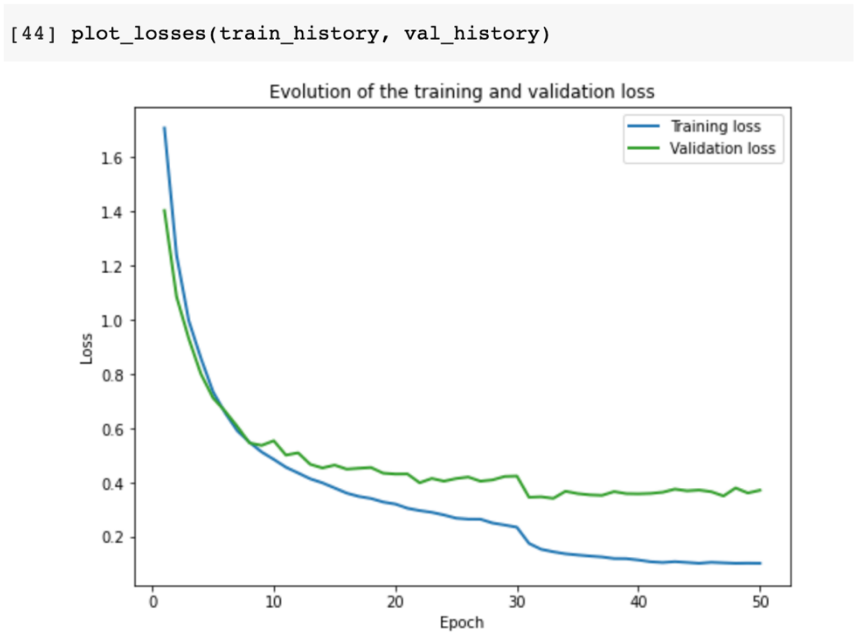
$$\mathrm{loss}(\mathbf{x}, \mathrm{class}) = \mathrm{weight}[\mathrm{class}] \left( -\mathbf{x}[\mathrm{class}] + \log \left( \sum_{j} \exp(\mathbf{x}[j]) \right) \right)$$

*Figure 4. Cross Entropy loss function*

For the optimizer, as we mentioned above, we used Adam method. This method is a technique which implements the adaptive learning rate where the classic stochastic gradient descent method remains the single learning rate.

## 3. Network training

For the training procedure, we used annealed learning rate scheme. In other words, we scheduled the learning rate by using *torch.optim.lr_scheduler.MultiStepLR* function to improve the accuracy. To be specific, we initiate with learning rate with 0.01 multiplying it by a decay factor of 0.1 after 30 and 40 epochs respectively by setting milestone parameter. Also, we set the batch size equals to 128 to accelerate the computation, which took each epoch to train about 20 seconds and total number of epochs were 50 to get the best accuracy. Then we compare the training loss and validation loss which shows in Figure 5, and we could get the lowest training loss of 0.09 and validation loss of 0.34.

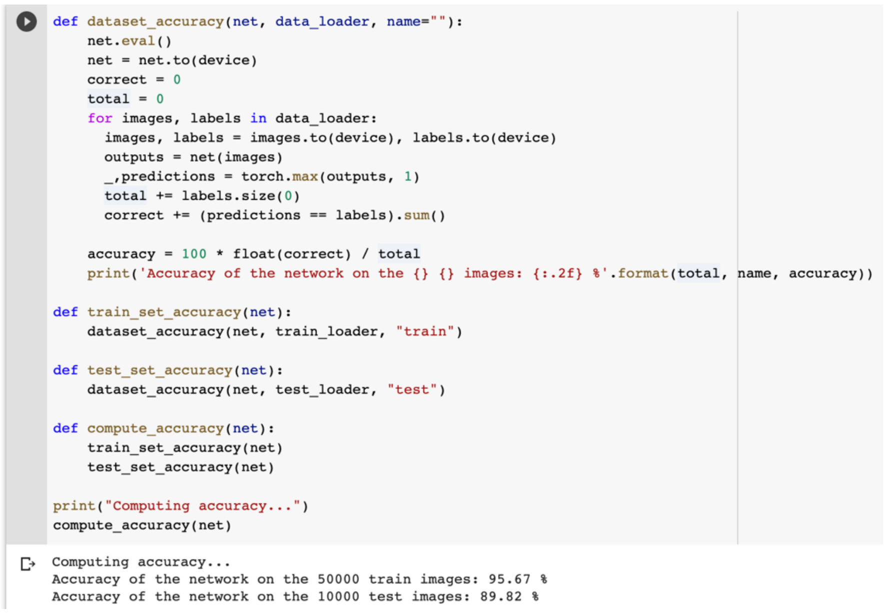


```
[44] plot_losses(train_history, val_history)
```

*Figure 5. Training and validation loss as the Epochs increase*

## 4. Network testing

After we finished the training procedure, we could get the best testing accuracy 89.82% as Figure 6 shows, which is about 90%.

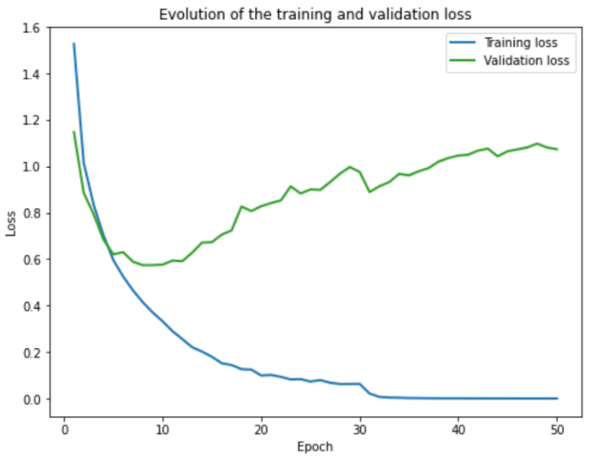


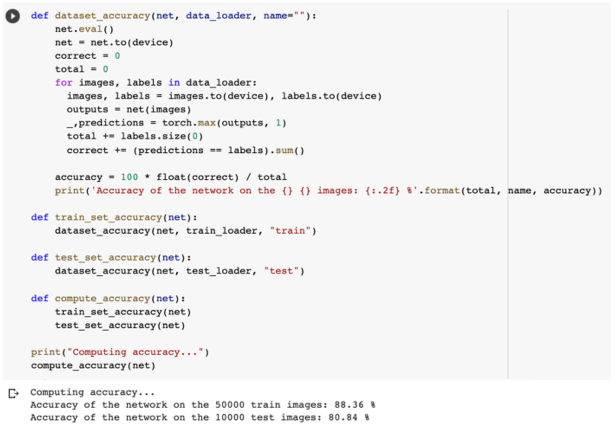
```
def dataset_accuracy(net, data_loader, name=""):
    net.eval()
    net = net.to(device)
    correct = 0
    total = 0
    for images, labels in data_loader:
      images, labels = images.to(device), labels.to(device)
      outputs = net(images)
      _,predictions = torch.max(outputs, 1)
      total += labels.size(0)
      correct += (predictions == labels).sum()

    accuracy = 100 * float(correct) / total
    print('Accuracy of the network on the {} {} images: {:.2f} %'.format(total, name, accuracy))

def train_set_accuracy(net):
    dataset_accuracy(net, train_loader, "train")

def test_set_accuracy(net):
    dataset_accuracy(net, test_loader, "test")

def compute_accuracy(net):
    train_set_accuracy(net)
    test_set_accuracy(net)

print("Computing accuracy...")
compute_accuracy(net)
```

```
Computing accuracy...
Accuracy of the network on the 50000 train images: 95.67 %
Accuracy of the network on the 10000 test images: 89.82 %
```

*Figure 6. Best testing accuracy: 89.82%, about 90%*

Now, let`s compare the testing accuracy when we remove the data augmentation and normalization.





```
def dataset_accuracy(net, data_loader, name=""):
    net.eval()
    net = net.to(device)
    correct = 0
    total = 0
    for images, labels in data_loader:
      images, labels = images.to(device), labels.to(device)
      outputs = net(images)
      _,predictions = torch.max(outputs, 1)
      total += labels.size(0)
      correct += (predictions == labels).sum()

    accuracy = 100 * float(correct) / total
    print('Accuracy of the network on the {} {} images: {:.2f} %'.format(total, name, accuracy))

def train_set_accuracy(net):
    dataset_accuracy(net, train_loader, "train")

def test_set_accuracy(net):
    dataset_accuracy(net, test_loader, "test")

def compute_accuracy(net):
    train_set_accuracy(net)
    test_set_accuracy(net)

print("Computing accuracy...")
compute_accuracy(net)
```

```
Computing accuracy...
Accuracy of the network on the 50000 train images: 88.36 %
Accuracy of the network on the 10000 test images: 80.84 %
```

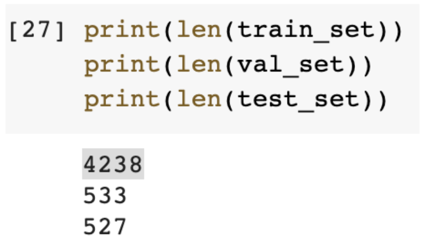
*Figure 7. Plot of the losses and the accuracy without the augmentation and normalization*

When we look at the Figure 7, we can clear observe that the validation loss increases and the testing accuracy is about 80.84%, which is far low compared to the previous result. So, we can say that the data augmentation and normalization improve the accuracy of the model. It is because classification loss can be less sensitive to small changes to the weights, which is easier to optimize.

## Part B: Experiments of KaoKore dataset

### 1. Data preparation

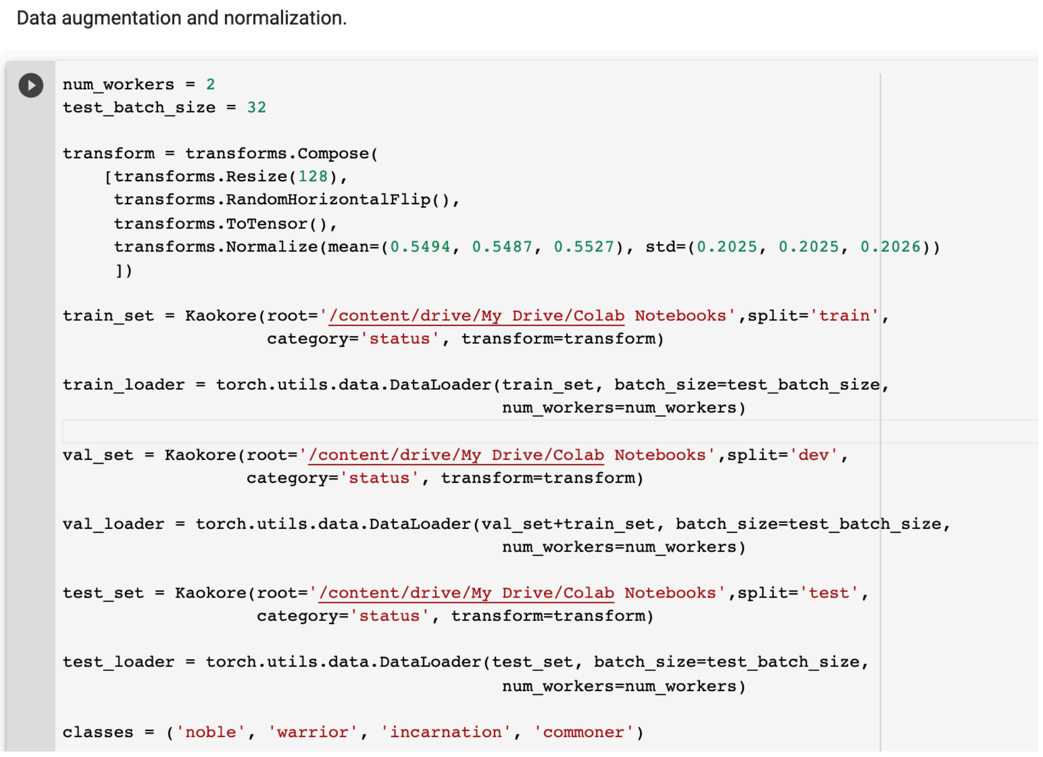
For this part, we have used the KaoKore dataset. In this dataset, we used only the status category and it contains four classes. Also, the size of the images is 256 X 256. Then we divided up the dataset into training, validation, and testing sets and get the samples. Especially, in this Lab, we have 4238 samples for training set, 533 samples for validation set, and 527 samples for testing set.



```
[27] print(len(train_set))
     print(len(val_set))
     print(len(test_set))

     4238
     533
     527
```

*Figure 8. Number of samples*

Then, we load the dataset using data augmentation and normalization method.



Data augmentation and normalization.

```
num_workers = 2
test_batch_size = 32

transform = transforms.Compose(
    [transforms.Resize(128),
     transforms.RandomHorizontalFlip(),
     transforms.ToTensor(),
     transforms.Normalize(mean=(0.5494, 0.5487, 0.5527), std=(0.2025, 0.2025, 0.2026))
     ])

train_set = Kaokore(root='/content/drive/My Drive/Colab Notebooks',split='train',
                    category='status', transform=transform)

train_loader = torch.utils.data.DataLoader(train_set, batch_size=test_batch_size,
                                           num_workers=num_workers)

val_set = Kaokore(root='/content/drive/My Drive/Colab Notebooks',split='dev',
                  category='status', transform=transform)

val_loader = torch.utils.data.DataLoader(val_set+train_set, batch_size=test_batch_size,
                                         num_workers=num_workers)

test_set = Kaokore(root='/content/drive/My Drive/Colab Notebooks',split='test',
                   category='status', transform=transform)

test_loader = torch.utils.data.DataLoader(test_set, batch_size=test_batch_size,
                                          num_workers=num_workers)

classes = ('noble', 'warrior', 'incarnation', 'commoner')
```

*Figure 9. Data augmentation and Normalization*

For data augmentation, We`ve used *torchvision.transforms* functions including *RandomHorizontalFlip(),Resize()*. *RandomHorizontalFlip* is to flip the image horizontally and *Resize()* is to resize the image to accelerate the training and save GPU memory and we used *size=128*. For normalization, We`ve used transforms.Normalize function with values (0.5494, 0.5487, 0.5527) for mean and (0.2025, 0.2025, 2026) for standard deviation. The values came from normalizing trainset, validation set, and testing set respectively.

## 2. Network training

For the training procedure, we used annealed learning rate scheme. In other words, we scheduled the learning rate by using *torch.optim.lr_scheduler.MultiStepLR* function to improve the accuracy. To be specific, we initiate with learning rate with 0.001 multiplying it by a decay factor of 0.1 after 30 and 40 epochs respectively by setting milestone parameter. Also, we set the batch size equals to 128 to accelerate the computation, which took each epoch to train about 20 seconds and total number of epochs were 50 to get the best accuracy. Then, we compare the training loss and validation loss which shows in Figure 10, and we could get the lowest training loss of 0.25 and validation loss of 0.58.

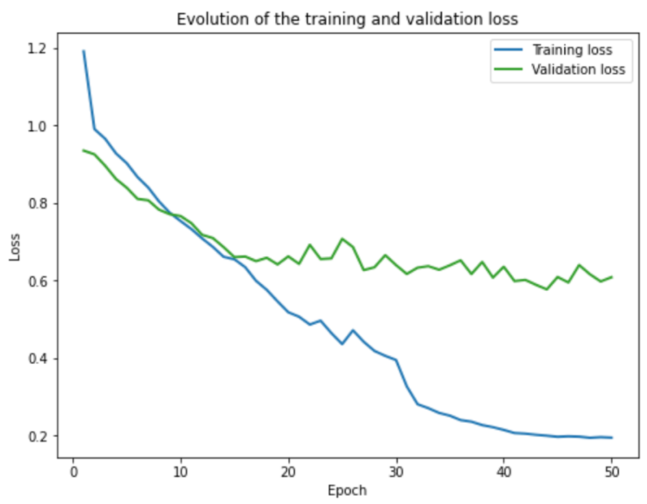
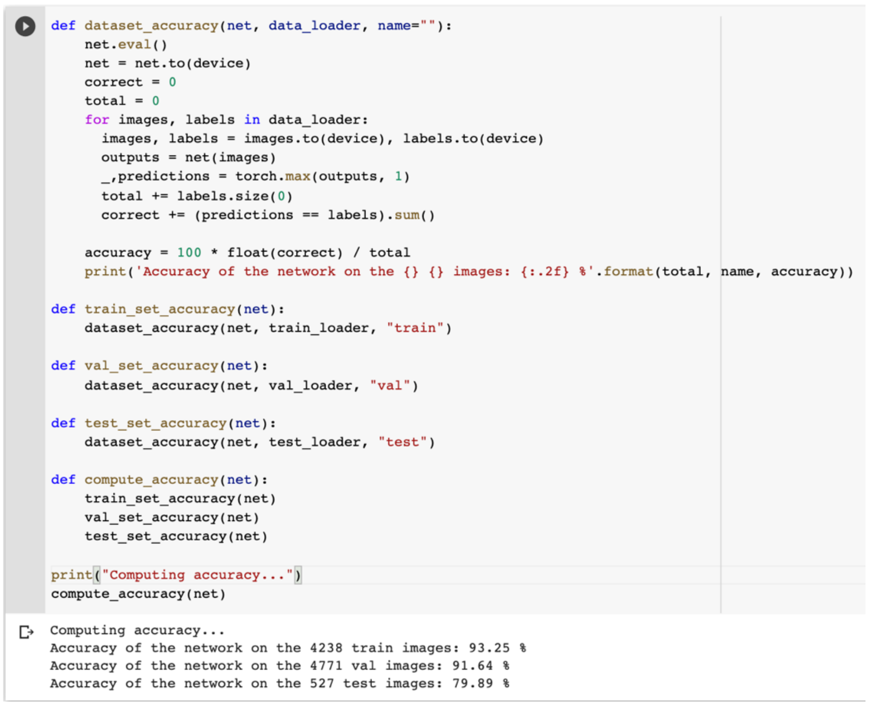


*Figure 10. Training and validation plot*

## 3. Network testing

After we finished the training procedure, we could get the best testing accuracy 79.89% as Figure 11 shows, which is about 80%.



```python
def dataset_accuracy(net, data_loader, name=""):
    net.eval()
    net = net.to(device)
    correct = 0
    total = 0
    for images, labels in data_loader:
      images, labels = images.to(device), labels.to(device)
      outputs = net(images)
      _,predictions = torch.max(outputs, 1)
      total += labels.size(0)
      correct += (predictions == labels).sum()

    accuracy = 100 * float(correct) / total
    print('Accuracy of the network on the {} {} images: {:.2f} %'.format(total, name, accuracy))

def train_set_accuracy(net):
    dataset_accuracy(net, train_loader, "train")

def val_set_accuracy(net):
    dataset_accuracy(net, val_loader, "val")

def test_set_accuracy(net):
    dataset_accuracy(net, test_loader, "test")

def compute_accuracy(net):
    train_set_accuracy(net)
    val_set_accuracy(net)
    test_set_accuracy(net)

print("Computing accuracy...")
compute_accuracy(net)
```

```
Computing accuracy...
Accuracy of the network on the 4238 train images: 93.25 %
Accuracy of the network on the 4771 val images: 91.64 %
Accuracy of the network on the 527 test images: 79.89 %
```

*Figure 11. Best testing accuracy: 79.89%, about 80%*