# Tanawin-st123975-AlexNET

October 20, 2023

[1]:
```
!nvidia-smi
```

```
Wed Oct 18 18:06:44 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   43C    P8    10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

[2]:
```
import torch
import torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
import time
import os
import copy
import torch.nn.functional as F
```

[3]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Using device', device)
```

```
Using device cuda:0
```

[4]:

```python
import matplotlib.pyplot as plt

def plot_data(val_acc_history, loss_acc_history):
    plt.plot(loss_acc_history, label='Validation')
    plt.title('Loss per epoch')
    plt.legend()
    plt.show()

    val_acc_history_cpu = [x.cpu() for x in val_acc_history]
    plt.plot(val_acc_history_cpu, label='Validation')
    plt.title('Accuracy per epoch')
    plt.legend()
    plt.show()
```

[5]:

```python
def plot_data_sub(val_acc_history, loss_acc_history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))

    ax1.plot(loss_acc_history, label='Validation')
    ax1.set_title('Loss per epoch')
    ax1.legend()

    val_acc_history_cpu = [x.cpu() for x in val_acc_history]
    ax2.plot(val_acc_history_cpu, label='Validation')
    ax2.set_title('Accuracy per epoch')
    ax2.legend()

    plt.tight_layout()
    plt.show()
```

[6]:

```python
def plot_results(train_loss_list, valid_loss_list, train_accuracy_list,␣
 ↪valid_accuracy_list, num_epochs, model_name):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    # Plot training and validation loss
    plt.plot(range(1, num_epochs + 1), train_loss_list, label='Train Loss')
    plt.plot(range(1, num_epochs + 1), valid_loss_list, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    # Plot training and validation accuracy
    plt.plot(range(1, num_epochs + 1), train_accuracy_list, label='Train␣
 ↪Accuracy')
    plt.plot(range(1, num_epochs + 1), valid_accuracy_list, label='Validation␣
 ↪Accuracy')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.suptitle(model_name)
    plt.show()
```

[7]:
```
def plot_data_loos_acc(model_name, num_epochs, train_loss_list,␣
 ↪valid_loss_list, train_accuracy_list, valid_accuracy_list):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    # Plot training and validation loss
    plt.plot(range(1, num_epochs + 1), train_loss_list, label='Train Loss')
    plt.plot(range(1, num_epochs + 1), valid_loss_list, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    # Plot training and validation accuracy
    plt.plot(range(1, num_epochs + 1), train_accuracy_list, label='Train␣
 ↪Accuracy')
    plt.plot(range(1, num_epochs + 1), valid_accuracy_list, label='Validation␣
 ↪Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.suptitle(model_name)
    plt.show()
```

[8]:
```
# Set up preprocessing of CIFAR-10 images to 3x224x224 with normalization
# using the magic ImageNet means and standard deviations. You can try
# RandomCrop, RandomHorizontalFlip, etc. during training to obtain
# slightly better generalization.

preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))])

# Download CIFAR-10 and split into training, validation, and test sets
```

```
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                             download=True,␣
 ↪transform=preprocess)

# Split the training set into training and validation sets randomly.
# CIFAR-10 train contains 50,000 examples, so let's split 80%/20%.

train_dataset, val_dataset = torch.utils.data.random_split(train_dataset,␣
 ↪[40000, 10000])

# Download the test set. If you use data augmentation transforms for the␣
 ↪training set,
# you'll want to use a different transformer here.

test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                            download=True, transform=preprocess)

# Dataset objects are mainly designed for datasets that can't fit entirely into␣
 ↪memory.
# Dataset objects don't load examples into memory until their __getitem__()␣
 ↪method is
# called. For supervised learning datasets, __getitem__() normally returns a␣
 ↪2-tuple
# on each call. To make a Dataset object like this useful, we use a DataLoader␣
 ↪object
# to optionally shuffle then batch the examples in each dataset. During␣
 ↪training.
# To keep our memory utilization small, we'll use 4 images per batch, but we␣
 ↪could use
# a much larger batch size on a dedicated GPU. To obtain optimal usage of the␣
 ↪GPU, we
# would like to load the examples for the next batch while the current batch is␣
 ↪being
# used for training. DataLoader handles this by spawining "worker" threads that␣
 ↪proactively
# fetch the next batch in the background, enabling parallel training on the GPU␣
 ↪and data
# loading/transforming/augmenting on the CPU. Here we use num_workers=2 (the␣
 ↪default)
# so that two batches are always ready or being prepared.

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
                                               shuffle=True, num_workers=2)
val_dataloader = torch.utils.data.DataLoader(val_dataset, batch_size=64,
                                             shuffle=False, num_workers=2)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
```

```
                                               shuffle=False, num_workers=2)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

100%|      | 170498071/170498071 [00:10<00:00, 15831297.70it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

[9]:
```
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25,␣
 ↪weights_name='weight_save', is_inception=False):
    '''
    train_model function

    Train a PyTorch model for a given number of epochs.

            Parameters:
                    model: Pytorch model
                    dataloaders: dataset
                    criterion: loss function
                    optimizer: update weights function
                    num_epochs: number of epochs
                    weights_name: file name to save weights
                    is_inception: The model is inception net (Google LeNet) or␣
 ↪not

            Returns:
                    model: Best model from evaluation result
                    val_acc_history: evaluation accuracy history
                    loss_acc_history: loss value history
    '''
    since = time.time()

    val_acc_history = []
    loss_acc_history = []

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        epoch_start = time.time()

        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
```

```
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over the train/validation dataset according to which␣
↪phase we're in

            for inputs, labels in dataloaders[phase]:

                # Inputs is one batch of input images, and labels is a␣
↪corresponding vector of integers
                # labeling each image in the batch. First, we move these␣
↪tensors to our target device.

                inputs = inputs.to(device)
                labels = labels.to(device)

                # Zero out any parameter gradients that have previously been␣
↪calculated. Parameter
                # gradients accumulate over as many backward() passes as we let␣
↪them, so they need
                # to be zeroed out after each optimizer step.

                optimizer.zero_grad()

                # Instruct PyTorch to track gradients only if this is the␣
↪training phase, then run the
                # forward propagation and optionally the backward propagation␣
↪step for this iteration.

                with torch.set_grad_enabled(phase == 'train'):
                    # The inception model is a special case during training␣
↪because it has an auxiliary
                    # output used to encourage discriminative representations␣
↪in the deeper feature maps.
                    # We need to calculate loss for both outputs. Otherwise, we␣
↪have a single output to
                    # calculate the loss on.
                    if is_inception and phase == 'train':
                        # From https://discuss.pytorch.org/t/
↪how-to-optimize-inception-model-with-auxiliary-classifiers/7958
                        outputs, aux_outputs = model(inputs)
```

```
                    loss1 = criterion(outputs, labels)
                    loss2 = criterion(aux_outputs, labels)
                    loss = loss1 + 0.4 * loss2
                else:
                    outputs = model(inputs)
                    loss = criterion(outputs, labels)

                _, preds = torch.max(outputs, 1)

                # Backpropagate only if in training phase

                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            # Gather our summary statistics

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.double() / len(dataloaders[phase].
↪dataset)
        epoch_end = time.time()

        elapsed_epoch = epoch_end - epoch_start

        print('{} Loss: {:.4f} Acc: {:.4f}'.format(phase, epoch_loss,␣
↪epoch_acc))
        print("Epoch time taken: ", elapsed_epoch)

        # If this is the best model on the validation set so far, deep copy␣
↪it

        if phase == 'val' and epoch_acc > best_acc:
            best_acc = epoch_acc
            best_model_wts = copy.deepcopy(model.state_dict())
            torch.save(model.state_dict(), weights_name + ".pth")
        if phase == 'val':
            val_acc_history.append(epoch_acc)
        if phase == 'train':
            loss_acc_history.append(epoch_loss)

    print()

# Output summary statistics, load the best weight set, and return results
```

```
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed // 60,␣
 ↪time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))
    model.load_state_dict(best_model_wts)
    return model, val_acc_history, loss_acc_history
```

[10]:
```
class AlexNetModule(nn.Module):
    '''
    An AlexNet-like CNN

    Attributes
    ----------
    num_classes : int
        Number of classes in the final multinomial output layer
    features : Sequential
        The feature extraction portion of the network
    avgpool : AdaptiveAvgPool2d
        Convert the final feature layer to 6x6 feature maps by average pooling␣
 ↪if they are not already 6x6
    classifier : Sequential
        Classify the feature maps into num_classes classes
    '''
    def __init__(self, num_classes: int = 10) -> None:
        super().__init__()
        self.num_classes = num_classes
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
```

```
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

[11]:
```
alexnet = AlexNetModule(10)
alexnet = alexnet.to(device)
```

[12]:
```
# Using CrossEntropyLoss for multinomial classification (Because we have 10␣
 ↪classes)
criterion = nn.CrossEntropyLoss()
# parameters = weights
params_to_update = alexnet.parameters()
# Use scholastic gradient descent for update weights in model with learning␣
 ↪rate 0.001 and momentum 0.9
optimizer = optim.SGD(params_to_update, lr=0.001, momentum=0.9)
```

[13]:
```
dataloaders = { 'train': train_dataloader, 'val': val_dataloader }
```

[14]:
```
best_model, val_acc_history, loss_acc_history = train_model(alexnet,␣
 ↪dataloaders, criterion, optimizer, 50, 'alex_sequential_lr_0.001_bestsofar')
```

```
Epoch 0/49
----------
train Loss: 2.3004 Acc: 0.1150
Epoch time taken:  46.182127475738525
val Loss: 2.2915 Acc: 0.1286
Epoch time taken:  55.58980894088745

Epoch 1/49
----------
train Loss: 2.1255 Acc: 0.2124
Epoch time taken:  39.23051118850708
val Loss: 1.9126 Acc: 0.3181
Epoch time taken:  48.654441356658936

Epoch 2/49
----------
train Loss: 1.7878 Acc: 0.3419
Epoch time taken:  39.86481976509094
```

```
val Loss: 1.6680 Acc: 0.3956
Epoch time taken:  49.335160970687866

Epoch 3/49
----------
train Loss: 1.6090 Acc: 0.4045
Epoch time taken:  39.651798486709595
val Loss: 1.5213 Acc: 0.4407
Epoch time taken:  49.1481192111969

Epoch 4/49
----------
train Loss: 1.4825 Acc: 0.4563
Epoch time taken:  39.87878394126892
val Loss: 1.4104 Acc: 0.4838
Epoch time taken:  49.25736212730408

Epoch 5/49
----------
train Loss: 1.3792 Acc: 0.4986
Epoch time taken:  39.78267049789429
val Loss: 1.3928 Acc: 0.5068
Epoch time taken:  49.22571921348572

Epoch 6/49
----------
train Loss: 1.2929 Acc: 0.5350
Epoch time taken:  39.82869625091553
val Loss: 1.2501 Acc: 0.5524
Epoch time taken:  49.314812898635864

Epoch 7/49
----------
train Loss: 1.2095 Acc: 0.5669
Epoch time taken:  39.75265026092529
val Loss: 1.1304 Acc: 0.5974
Epoch time taken:  49.308868646621704

Epoch 8/49
----------
train Loss: 1.1366 Acc: 0.5994
Epoch time taken:  39.71128034591675
val Loss: 1.1084 Acc: 0.6044
Epoch time taken:  49.03447198867798

Epoch 9/49
----------
train Loss: 1.0549 Acc: 0.6270
```

```
Epoch time taken:  39.68074607849121
val Loss: 0.9724 Acc: 0.6611
Epoch time taken:  48.994832277297974

Epoch 10/49
----------
train Loss: 0.9799 Acc: 0.6567
Epoch time taken:  39.662819147109985
val Loss: 0.9289 Acc: 0.6701
Epoch time taken:  48.96351957321167

Epoch 11/49
----------
train Loss: 0.9171 Acc: 0.6793
Epoch time taken:  39.73200082778931
val Loss: 0.9510 Acc: 0.6713
Epoch time taken:  49.05087065696716

Epoch 12/49
----------
train Loss: 0.8506 Acc: 0.7047
Epoch time taken:  39.63900017738342
val Loss: 0.8111 Acc: 0.7157
Epoch time taken:  48.97172427177429

Epoch 13/49
----------
train Loss: 0.7982 Acc: 0.7208
Epoch time taken:  39.67841720581055
val Loss: 0.8103 Acc: 0.7213
Epoch time taken:  48.96117901802063

Epoch 14/49
----------
train Loss: 0.7483 Acc: 0.7383
Epoch time taken:  39.697455167770386
val Loss: 0.8279 Acc: 0.7143
Epoch time taken:  48.96851992607117

Epoch 15/49
----------
train Loss: 0.7130 Acc: 0.7515
Epoch time taken:  39.62378001213074
val Loss: 0.7433 Acc: 0.7433
Epoch time taken:  48.95343351364136

Epoch 16/49
----------
```

```
train Loss: 0.6545 Acc: 0.7719
Epoch time taken:  39.654545068740845
val Loss: 0.6879 Acc: 0.7637
Epoch time taken:  48.967679262161255

Epoch 17/49
----------
train Loss: 0.6244 Acc: 0.7815
Epoch time taken:  39.68966102600098
val Loss: 0.7094 Acc: 0.7567
Epoch time taken:  48.92105221748352

Epoch 18/49
----------
train Loss: 0.5892 Acc: 0.7940
Epoch time taken:  39.57251262664795
val Loss: 0.6837 Acc: 0.7694
Epoch time taken:  48.84906554222107

Epoch 19/49
----------
train Loss: 0.5578 Acc: 0.8058
Epoch time taken:  39.6605269908905
val Loss: 0.6555 Acc: 0.7721
Epoch time taken:  48.91392254829407

Epoch 20/49
----------
train Loss: 0.5233 Acc: 0.8162
Epoch time taken:  39.64619731903076
val Loss: 0.6497 Acc: 0.7785
Epoch time taken:  48.90301156044006

Epoch 21/49
----------
train Loss: 0.4922 Acc: 0.8287
Epoch time taken:  39.64341449737549
val Loss: 0.6432 Acc: 0.7850
Epoch time taken:  48.89917349815369

Epoch 22/49
----------
train Loss: 0.4620 Acc: 0.8377
Epoch time taken:  39.5831298828125
val Loss: 0.6117 Acc: 0.7901
Epoch time taken:  48.8268301486969

Epoch 23/49
```

```
----------
train Loss: 0.4323 Acc: 0.8475
Epoch time taken:  39.593567848205566
val Loss: 0.6208 Acc: 0.7914
Epoch time taken:  48.907570123672485

Epoch 24/49
----------
train Loss: 0.4065 Acc: 0.8580
Epoch time taken:  39.61204743385315
val Loss: 0.6145 Acc: 0.7959
Epoch time taken:  48.834168434143066

Epoch 25/49
----------
train Loss: 0.3838 Acc: 0.8644
Epoch time taken:  39.58351993560791
val Loss: 0.5909 Acc: 0.8029
Epoch time taken:  48.88575458526611

Epoch 26/49
----------
train Loss: 0.3554 Acc: 0.8755
Epoch time taken:  39.69348931312561
val Loss: 0.5750 Acc: 0.8081
Epoch time taken:  49.076382875442505

Epoch 27/49
----------
train Loss: 0.3263 Acc: 0.8847
Epoch time taken:  39.57463502883911
val Loss: 0.6132 Acc: 0.8055
Epoch time taken:  48.74088501930237

Epoch 28/49
----------
train Loss: 0.3042 Acc: 0.8922
Epoch time taken:  39.795562982559204
val Loss: 0.6251 Acc: 0.8007
Epoch time taken:  49.195048570632935

Epoch 29/49
----------
train Loss: 0.2861 Acc: 0.8987
Epoch time taken:  39.726269006729126
val Loss: 0.5992 Acc: 0.8107
Epoch time taken:  49.07827043533325
```

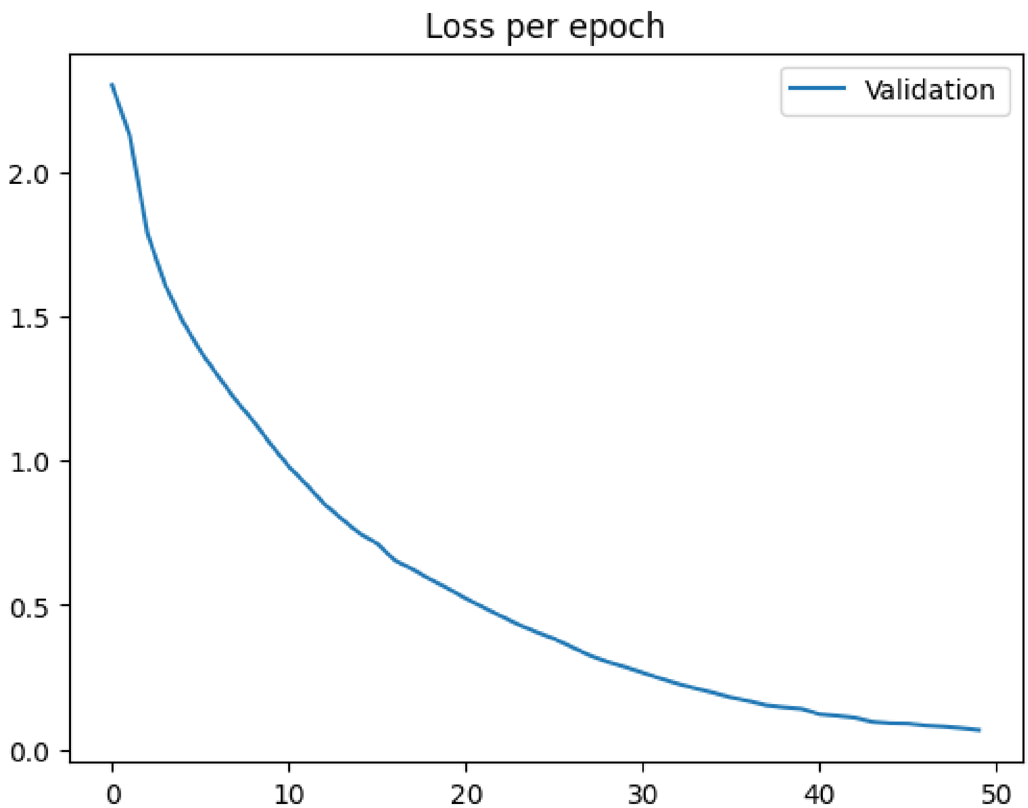
```
Epoch 30/49
----------
train Loss: 0.2665 Acc: 0.9066
Epoch time taken:  39.79297113418579
val Loss: 0.5766 Acc: 0.8111
Epoch time taken:  49.268059968948364

Epoch 31/49
----------
train Loss: 0.2465 Acc: 0.9120
Epoch time taken:  39.71680569648743
val Loss: 0.5972 Acc: 0.8112
Epoch time taken:  49.2406792640686

Epoch 32/49
----------
train Loss: 0.2278 Acc: 0.9187
Epoch time taken:  39.655497550964355
val Loss: 0.6200 Acc: 0.8099
Epoch time taken:  49.143606185913086

Epoch 33/49
----------
train Loss: 0.2119 Acc: 0.9247
Epoch time taken:  39.7529513835907
val Loss: 0.6263 Acc: 0.8116
Epoch time taken:  49.08042860031128

Epoch 34/49
----------
train Loss: 0.1971 Acc: 0.9313
Epoch time taken:  39.74892735481262
val Loss: 0.6040 Acc: 0.8152
Epoch time taken:  49.13806366920471

Epoch 35/49
----------
train Loss: 0.1807 Acc: 0.9363
Epoch time taken:  39.847148418426514
val Loss: 0.6638 Acc: 0.8157
Epoch time taken:  49.20997667312622

Epoch 36/49
----------
train Loss: 0.1682 Acc: 0.9400
Epoch time taken:  39.69161868095398
val Loss: 0.6365 Acc: 0.8156
Epoch time taken:  49.17864203453064
```

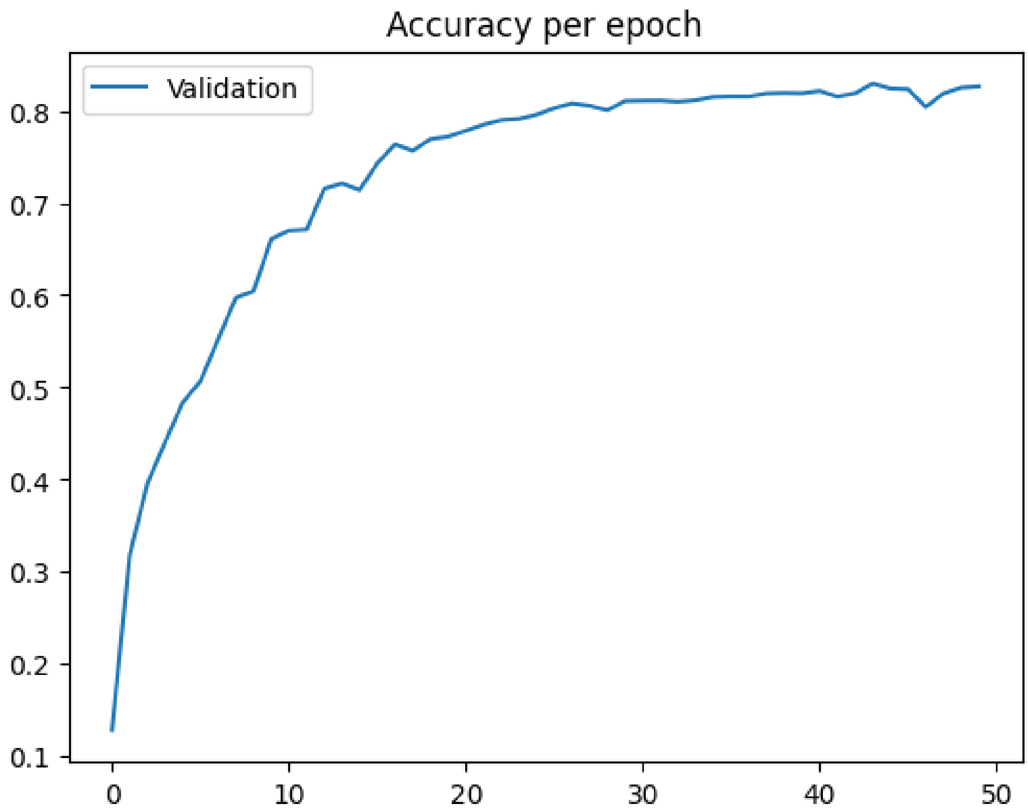
```
Epoch 37/49
----------
train Loss: 0.1535 Acc: 0.9454
Epoch time taken:  39.61636471748352
val Loss: 0.6382 Acc: 0.8189
Epoch time taken:  49.04148578643799

Epoch 38/49
----------
train Loss: 0.1465 Acc: 0.9494
Epoch time taken:  39.78605127334595
val Loss: 0.6545 Acc: 0.8194
Epoch time taken:  49.185128688812256

Epoch 39/49
----------
train Loss: 0.1414 Acc: 0.9496
Epoch time taken:  39.815269231796265
val Loss: 0.6378 Acc: 0.8191
Epoch time taken:  49.38883662223816

Epoch 40/49
----------
train Loss: 0.1228 Acc: 0.9565
Epoch time taken:  39.729148387908936
val Loss: 0.6445 Acc: 0.8216
Epoch time taken:  49.22160983085632

Epoch 41/49
----------
train Loss: 0.1174 Acc: 0.9581
Epoch time taken:  39.78662848472595
val Loss: 0.6860 Acc: 0.8155
Epoch time taken:  49.34826111793518

Epoch 42/49
----------
train Loss: 0.1107 Acc: 0.9611
Epoch time taken:  39.76258969306946
val Loss: 0.6811 Acc: 0.8190
Epoch time taken:  49.17836260795593

Epoch 43/49
----------
train Loss: 0.0963 Acc: 0.9655
Epoch time taken:  39.71380352973938
val Loss: 0.6899 Acc: 0.8295
```

```
Epoch time taken:  49.1376793384552

Epoch 44/49
----------
train Loss: 0.0915 Acc: 0.9684
Epoch time taken:  39.70981693267822
val Loss: 0.6816 Acc: 0.8243
Epoch time taken:  49.164443254470825

Epoch 45/49
----------
train Loss: 0.0903 Acc: 0.9685
Epoch time taken:  39.8269362449646
val Loss: 0.6693 Acc: 0.8235
Epoch time taken:  49.235639572143555

Epoch 46/49
----------
train Loss: 0.0837 Acc: 0.9702
Epoch time taken:  39.72202396392822
val Loss: 0.8013 Acc: 0.8042
Epoch time taken:  49.16241669654846

Epoch 47/49
----------
train Loss: 0.0798 Acc: 0.9723
Epoch time taken:  39.705278158187866
val Loss: 0.7443 Acc: 0.8189
Epoch time taken:  49.24738335609436

Epoch 48/49
----------
train Loss: 0.0745 Acc: 0.9744
Epoch time taken:  40.08659815788269
val Loss: 0.6897 Acc: 0.8252
Epoch time taken:  49.576655626297

Epoch 49/49
----------
train Loss: 0.0681 Acc: 0.9777
Epoch time taken:  39.84947729110718
val Loss: 0.7387 Acc: 0.8265
Epoch time taken:  49.34030294418335

Training complete in 41m 20s
Best val Acc: 0.829500
```

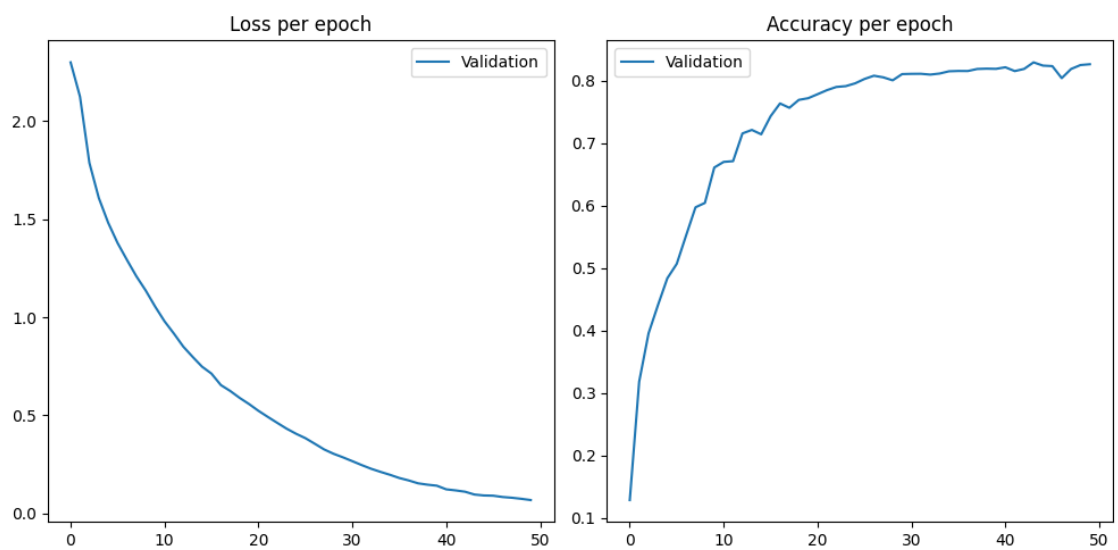
[15]:

```
plot_data(val_acc_history, loss_acc_history)
```



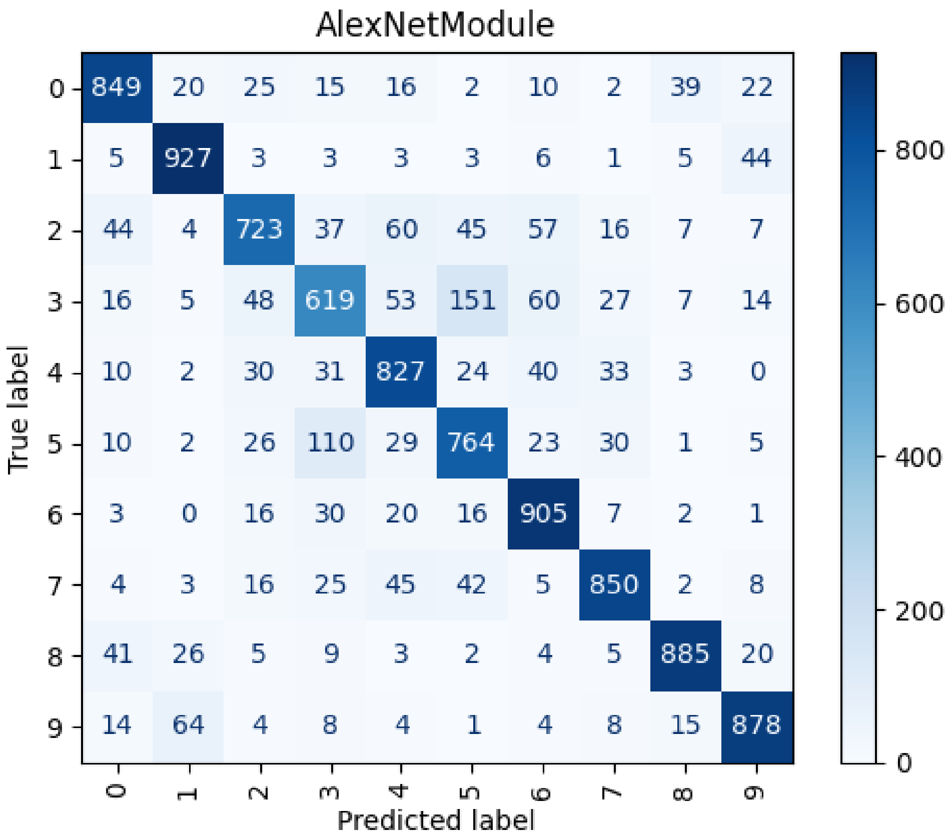



[16]:
```
plot_data_sub(val_acc_history, loss_acc_history)
```



[18]:

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np

loaded_model = AlexNetModule()
loaded_model.load_state_dict(torch.load('/content/alex_sequential_lr_0.
↪001_bestsofar.pth'))
loaded_model.to(device)  # Move the loaded model to the same device as the data

# Test the loaded model
def test_loaded_model(model, test_dataloader, device):
    model.eval()  # Set the model to evaluation mode
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in test_dataloader:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    return all_preds, all_labels

# Test the loaded model
all_preds, all_labels = test_loaded_model(loaded_model, test_dataloader, device)

# Generate confusion matrix
cm = confusion_matrix(all_labels, all_preds)

# Display the confusion matrix using ConfusionMatrixDisplay
cmd = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.
↪unique(all_labels))
cmd.plot(cmap='Blues', xticks_rotation='vertical')
plt.title("AlexNetModule")
# Show the plot
plt.show()
```


AlexNetModule
True label
Predicted label
0
1
2
3
4
5
6
7
8
9
849 20 25 15 16 2 10 2 39 22
5 927 3 3 3 3 6 1 5 44
44 4 723 37 60 45 57 16 7 7
16 5 48 619 53 151 60 27 7 14
10 2 30 31 827 24 40 33 3 0
10 2 26 110 29 764 23 30 1 5
3 0 16 30 20 16 905 7 2 1
4 3 16 25 45 42 5 850 2 8
41 26 5 9 3 2 4 5 885 20
14 64 4 8 4 1 4 8 15 878
800
600
400
200
0