Problem-1-MLP

March 12, 2024

Simple MLP Baseline for Image classification for the CIFAR-10 dataset

[ ]:
```
import torch
import torchvision.datasets as datasets
from torchvision.transforms import v2
import torch.utils.data as dataloader
import torch.nn as nn
import torch.optim as optim
import numpy as np

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda:0
```

[ ]:
```
# Params for wandb sweeps
PROJECT_NAME = 'mlp_cifar10_pytorch'
PROJECT_ENTITY = 'cs20b013-bersilin'

# The 10 classes in the CIFAR-10 dataset
LABELS = {
    0: 'airplane',
    1: 'automobile',
    2: 'bird',
    3: 'cat',
    4: 'deer',
    5: 'dog',
    6: 'frog',
    7: 'horse',
    8: 'ship',
    9: 'truck'
}

# The architecture of the MLP model
ARCH = [500, 250, 100]
DATA_DIR = "../data"
```

[ ]:
```python
def get_transform(mean, std):
    '''
    Returns a transform to convert a CIFAR image to a tensor of type float32
    '''
    return v2.Compose([
        v2.ToImage(),
        v2.ToDtype(torch.float32, scale=True),
        v2.Normalize(mean, std)
    ])
```

[ ]:
```python
def get_dataloader(batch_size: int, val_split: float = 0.2, shuffle: bool =␣
 ↪True):
    '''
    Load the CIFAR-10 dataset

    Normalizes the data using the mean and standard deviation of the training␣
 ↪data
    '''
    train_data = datasets.CIFAR10(root=DATA_DIR, train=True, download=True)
    test_data = datasets.CIFAR10(root=DATA_DIR, train=False, download=True)

    mean = np.array(train_data.data).mean(axis=(0, 1, 2)) / 255
    std = np.array(train_data.data).std(axis=(0, 1, 2)) / 255

    transform = get_transform(mean, std)
    train_data.transform = transform
    test_data.transform = transform

    train_size = int((1 - val_split) * len(train_data))
    val_size = len(train_data) - train_size

    train_data, val_data = dataloader.random_split(train_data, [train_size,␣
 ↪val_size])

    train_loader = dataloader.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=shuffle)
    val_loader = dataloader.DataLoader(val_data, batch_size=batch_size,␣
 ↪shuffle=shuffle)
    test_loader = dataloader.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False)

    return train_data, test_data, train_loader, val_loader, test_loader, mean,␣
 ↪std
```

[ ]:
```python
def show_random_image(dataset: datasets.CIFAR10, index: int = None, mean: np.
 ↪ndarray = None, std: np.ndarray = None):
    '''
```

```
    Shows a random image from the dataset

    If the mean and standard deviation are provided, the image is denormalized
    If the index is provided, the image at that index is shown else a random␣
↪image is shown
    '''
    if index is None:
        index = np.random.randint(0, len(dataset))
    else:
        index = index

    image, label = dataset[index]

    if mean is not None and std is not None:
        # image is (3, 32, 32), std and mean are (3,)
        image = image * std[:, None, None] + mean[:, None, None]

    plot = plt.imshow(image.permute(1, 2, 0).clip(0, 1))
    plt.title(f"True Label: {LABELS[label]}")

    return plot, index, label
```

[ ]:
```
def plot_accuracies(train_acc, val_acc):
    '''
    Plot the training and validation accuracies
    '''
    plot = plt.plot(train_acc, label='Training Accuracy')
    plt.plot(val_acc, label='Validation Accuracy')
    plt.legend()
    plt.title('Training and Validation Accuracies')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')

    return plot
```

[ ]:
```
def plot_losses(train_loss, val_loss):
    '''
    Plot the training and validation losses
    '''
    plot = plt.plot(train_loss, label='Training Loss')
    plt.plot(val_loss, label='Validation Loss')
    plt.legend()
    plt.title('Training and Validation Losses')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')

    return plot
```

[ ]:
```python
# Architecture of the model

class MLP(nn.Module):
    '''
    Multi-layer perceptron model with BatchNorm

    Activation function: ReLU
    Output activation function: Softmax
    '''
    def __init__(self, arch, in_size, out_size, batch_norm: bool = True):
        super(MLP, self).__init__()

        self.batch_norm = batch_norm
        self.sequence  = self.get_layers(arch, in_size, out_size)
        self.fc = nn.Sequential(*self.sequence)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        x = self.softmax(x)
        return x

    def get_layers(self, arch, in_size, out_size):
        '''
        Returns a list of layers for the model
        '''
        layers = []
        layers.append(nn.Linear(in_features=in_size, out_features=arch[0]))
        if self.batch_norm:
            layers.append(nn.BatchNorm1d(arch[0]))
        layers.append(nn.ReLU())

        for i in range(1, len(arch)):
            layers.append(nn.Linear(in_features=arch[i-1],␣
 ↪out_features=arch[i]))
            if self.batch_norm:
                layers.append(nn.BatchNorm1d(arch[i]))
            layers.append(nn.ReLU())

        layers.append(nn.Linear(in_features=arch[-1], out_features=out_size))

        return layers
```

[ ]:
```python
def get_accuracy(model: nn.Module, data_loader: dataloader.DataLoader, device:␣
 ↪torch.device):
    '''
```

```
    Get the accuracy of the model on the data_loader
    '''
    correct, total = 0, 0

    with torch.no_grad():
        for X, y in data_loader:
            X, y = X.to(device), y.to(device)
            preds = model(X)
            _, predicted = torch.max(preds, 1)
            correct += (predicted == y).sum().item()
            total += y.size(0)

    return correct / total
```

[ ]:
```
def get_predicted_labels(model: nn.Module, data_loader: dataloader.DataLoader,␣
 ↪device: torch.device):
    '''
    Get the predicted labels of the model on the data_loader
    '''
    labels = []
    with torch.no_grad():
        for X, y in data_loader:
            X, y = X.to(device), y.to(device)
            preds = model(X)
            _, predicted = torch.max(preds, 1)
            labels.append(predicted)

    return torch.cat(labels)
```

[ ]:
```
# Training the model

def train(configs, train_loader: dataloader.DataLoader, val_loader: dataloader.
 ↪DataLoader, criterion: nn.CrossEntropyLoss,
          optimizer: optim.Optimizer, model: nn.Module, device: torch.device):
    '''
    Train the model
    '''

    if(configs['wandb_log']):
        import wandb
        wandb.init(project=PROJECT_NAME, entity=PROJECT_ENTITY)

    print('Training the model...')
    print('----------------------')

    val_accuracies, train_accuracies = [], []
    val_losses, train_losses = [], []
```

```python
    for epoch in range(configs['num_epochs']):
        model.train()
        running_loss = 0.0

        total_iterations = len(train_loader)

        for i, (inputs, labels) in enumerate(train_loader):
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs) # Forward pass
            loss = criterion(outputs, labels) # Calculate loss
            loss.backward() # Backward pass
            optimizer.step() # Update weights

            running_loss += loss.item()

            if (i != total_iterations-1):
                print(f'Epoch {epoch + 1}, Iteration {i + 1}/{total_iterations}, Loss: {loss.item()}', end='\r')
            else:
                print(f'Epoch {epoch + 1}, Iteration {i + 1}/{total_iterations}, Loss: {loss.item()}')

        print(f'Epoch {epoch + 1} done, Training Loss: {running_loss / len(train_loader)}')
        train_losses.append(running_loss / len(train_loader))

        # Validation loss
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

        print(f'Epoch {epoch + 1}, Validation Loss: {val_loss / len(val_loader)}')
        val_losses.append(val_loss / len(val_loader))

        train_accuracy = get_accuracy(model, train_loader, device)
        val_accuracy = get_accuracy(model, val_loader, device)
```

```
        train_accuracies.append(train_accuracy)
        val_accuracies.append(val_accuracy)

        print(f'Epoch {epoch + 1}, Training Accuracy: {train_accuracy},␣
↪Validation Accuracy: {val_accuracy} \n')

        if configs['wandb_log']:
            wandb.log({'Epoch:': epoch + 1,
                       'Training Loss': running_loss / len(train_loader),
                       'Validation Loss': val_loss / len(val_loader),
                       'Training Accuracy': train_accuracy,
                       'Validation Accuracy': val_accuracy})

    print('Finished Training')
    print('----------------------')

    return model, configs, train_accuracies, val_accuracies, train_losses,␣
↪val_losses
```

Used Wandb to run sweeps to find the best hyperparameters for the model from a set of hyperparameters.

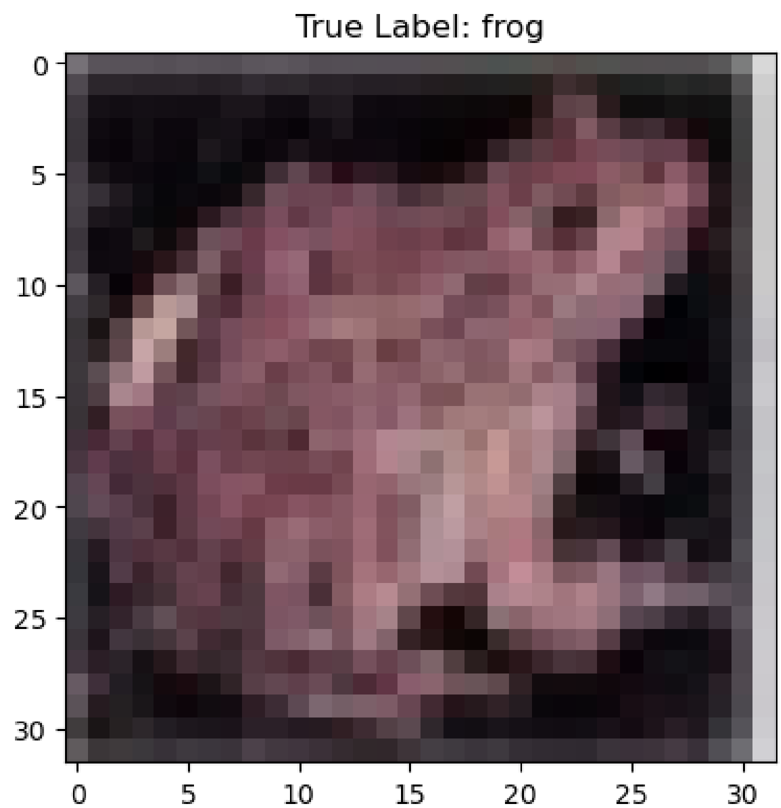
Link: MLP-Sweep

[ ]:
```
# The best hyperparameters found using wandb sweeps

best_configs = {
    'batch_norm': True,
    'learning_rate': 0.007,
    'num_epochs': 20,
    'momentum': 0.87,
    'wandb_log': False,
    'batch_size': 50
}
```

[ ]:
```
train_data, test_data, train_loader, val_loader, test_loader, mean, std =␣
↪get_dataloader(best_configs['batch_size'])
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[ ]:
```
# Show a random image from the dataset

plot, index, label = show_random_image(train_data, mean=mean, std=std)
```




```
[ ]: model = MLP(ARCH, 3*32*32, 10, best_configs['batch_norm']).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=best_configs['learning_rate'],␣
 ↪momentum=best_configs['momentum'])

print(model)
```

```
MLP(
  (fc): Sequential(
    (0): Linear(in_features=3072, out_features=500, bias=True)
    (1): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): Linear(in_features=500, out_features=250, bias=True)
    (4): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU()
    (6): Linear(in_features=250, out_features=100, bias=True)
    (7): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (8): ReLU()
```

```
    (9): Linear(in_features=100, out_features=10, bias=True)
  )
  (softmax): Softmax(dim=1)
)
```

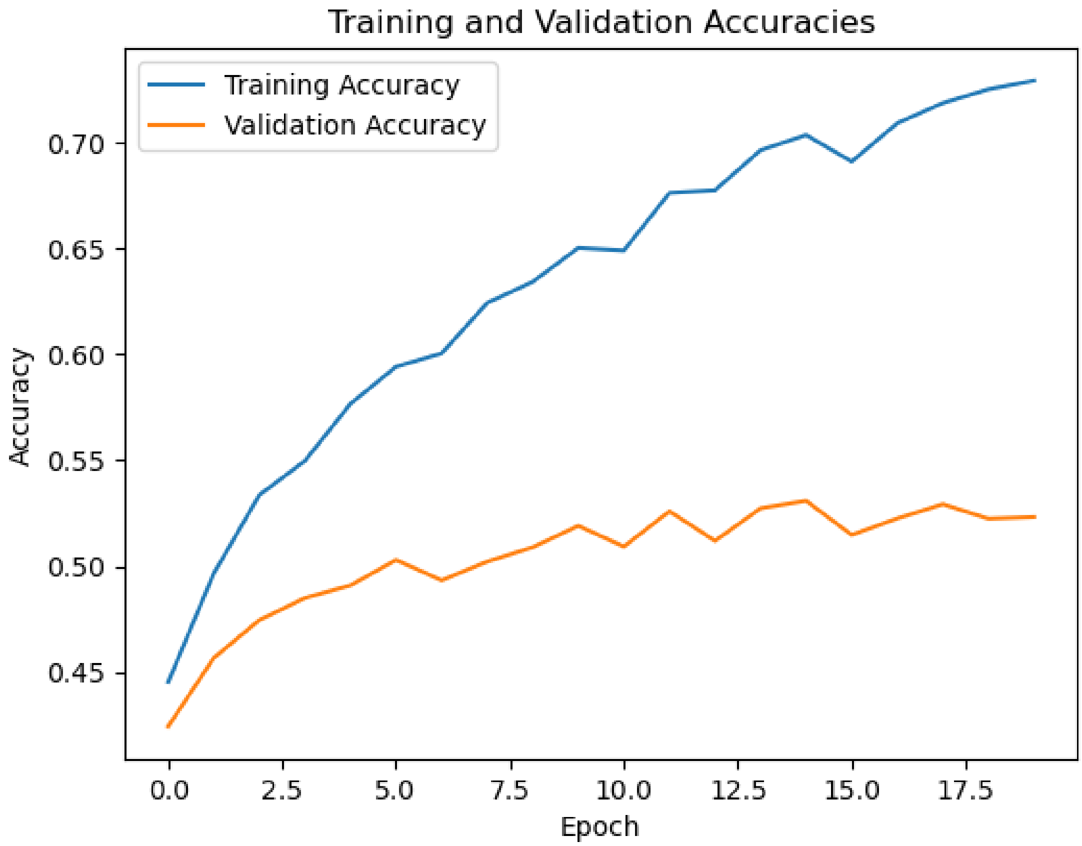
[ ]:
```
model, configs, train_acc, val_acc, train_losses, val_losses =␣
 ↪train(best_configs, train_loader, val_loader, criterion, optimizer, model,␣
 ↪device)

test_accuracy = get_accuracy(model, test_loader, device)
print(f'Test Accuracy: {test_accuracy}')
```

```
Training the model…
---------------------
Epoch 1, Iteration 800/800, Loss: 2.0156481266021738
Epoch 1 done, Training Loss: 2.1178781358897685
Epoch 1, Validation Loss: 2.0467407459020617
Epoch 1, Training Accuracy: 0.445225, Validation Accuracy: 0.4243

Epoch 2, Iteration 800/800, Loss: 2.0131654739379883
Epoch 2 done, Training Loss: 2.02337217181921
Epoch 2, Validation Loss: 2.0096182239055635
Epoch 2, Training Accuracy: 0.496525, Validation Accuracy: 0.4566

Epoch 3, Iteration 800/800, Loss: 2.0257201194763184
Epoch 3 done, Training Loss: 1.9827061545848848
Epoch 3, Validation Loss: 1.9895015108585357
Epoch 3, Training Accuracy: 0.533575, Validation Accuracy: 0.4745

Epoch 4, Iteration 800/800, Loss: 2.0387582778930664
Epoch 4 done, Training Loss: 1.9548261186480522
Epoch 4, Validation Loss: 1.9780973184108734
Epoch 4, Training Accuracy: 0.5497, Validation Accuracy: 0.4849

Epoch 5, Iteration 800/800, Loss: 1.8857461214065552
Epoch 5 done, Training Loss: 1.935291922390461
Epoch 5, Validation Loss: 1.967222598195076
Epoch 5, Training Accuracy: 0.576675, Validation Accuracy: 0.4909

Epoch 6, Iteration 800/800, Loss: 1.9440379142761235
Epoch 6 done, Training Loss: 1.9166350585222245
Epoch 6, Validation Loss: 1.959997730255127
Epoch 6, Training Accuracy: 0.594175, Validation Accuracy: 0.5029

Epoch 7, Iteration 800/800, Loss: 1.8273791074752808
Epoch 7 done, Training Loss: 1.9030521786212922
Epoch 7, Validation Loss: 1.9664887797832489
Epoch 7, Training Accuracy: 0.60045, Validation Accuracy: 0.4933
```

```
Epoch 8, Iteration 800/800, Loss: 1.9059983491897583
Epoch 8 done, Training Loss: 1.8858307473361493
Epoch 8, Validation Loss: 1.9552596139907836
Epoch 8, Training Accuracy: 0.624325, Validation Accuracy: 0.5021

Epoch 9, Iteration 800/800, Loss: 1.9185059070587158
Epoch 9 done, Training Loss: 1.8745500588417052
Epoch 9, Validation Loss: 1.9494841235876084
Epoch 9, Training Accuracy: 0.634275, Validation Accuracy: 0.5089

Epoch 10, Iteration 800/800, Loss: 1.8405575752258314
Epoch 10 done, Training Loss: 1.8614854721724987
Epoch 10, Validation Loss: 1.9427410674095154
Epoch 10, Training Accuracy: 0.650325, Validation Accuracy: 0.519

Epoch 11, Iteration 800/800, Loss: 1.7570589780807495
Epoch 11 done, Training Loss: 1.852667957097292
Epoch 11, Validation Loss: 1.9469806122779847
Epoch 11, Training Accuracy: 0.649075, Validation Accuracy: 0.5092

Epoch 12, Iteration 800/800, Loss: 1.8654538393020635
Epoch 12 done, Training Loss: 1.8404364612698556
Epoch 12, Validation Loss: 1.933016836643219
Epoch 12, Training Accuracy: 0.676275, Validation Accuracy: 0.5259

Epoch 13, Iteration 800/800, Loss: 1.8267539739608765
Epoch 13 done, Training Loss: 1.8298608508706093
Epoch 13, Validation Loss: 1.9446801519393921
Epoch 13, Training Accuracy: 0.67755, Validation Accuracy: 0.5119

Epoch 14, Iteration 800/800, Loss: 1.7870601415634155
Epoch 14 done, Training Loss: 1.8214345306158066
Epoch 14, Validation Loss: 1.9321112126111983
Epoch 14, Training Accuracy: 0.696475, Validation Accuracy: 0.5273

Epoch 15, Iteration 800/800, Loss: 1.7862341403961182
Epoch 15 done, Training Loss: 1.8118319979310036
Epoch 15, Validation Loss: 1.9285208147764206
Epoch 15, Training Accuracy: 0.703575, Validation Accuracy: 0.5308

Epoch 16, Iteration 800/800, Loss: 1.8201119899749756
Epoch 16 done, Training Loss: 1.8030758649110794
Epoch 16, Validation Loss: 1.9415590167045593
Epoch 16, Training Accuracy: 0.6911, Validation Accuracy: 0.5147

Epoch 17, Iteration 800/800, Loss: 1.7824553251266482
Epoch 17 done, Training Loss: 1.7951377731561662
```

```
Epoch 17, Validation Loss: 1.9351833313703537
Epoch 17, Training Accuracy: 0.7092, Validation Accuracy: 0.5225

Epoch 18, Iteration 800/800, Loss: 1.8446632623672485
Epoch 18 done, Training Loss: 1.7885868905484676
Epoch 18, Validation Loss: 1.9284369814395905
Epoch 18, Training Accuracy: 0.718725, Validation Accuracy: 0.5292

Epoch 19, Iteration 800/800, Loss: 1.7742190361022957
Epoch 19 done, Training Loss: 1.7802397894859314
Epoch 19, Validation Loss: 1.9333221793174744
Epoch 19, Training Accuracy: 0.725225, Validation Accuracy: 0.5224

Epoch 20, Iteration 800/800, Loss: 1.8311920166015625
Epoch 20 done, Training Loss: 1.7744938723742962
Epoch 20, Validation Loss: 1.9366289907693863
Epoch 20, Training Accuracy: 0.7293, Validation Accuracy: 0.5232

Finished Training
---------------------
Test Accuracy: 0.5249
```

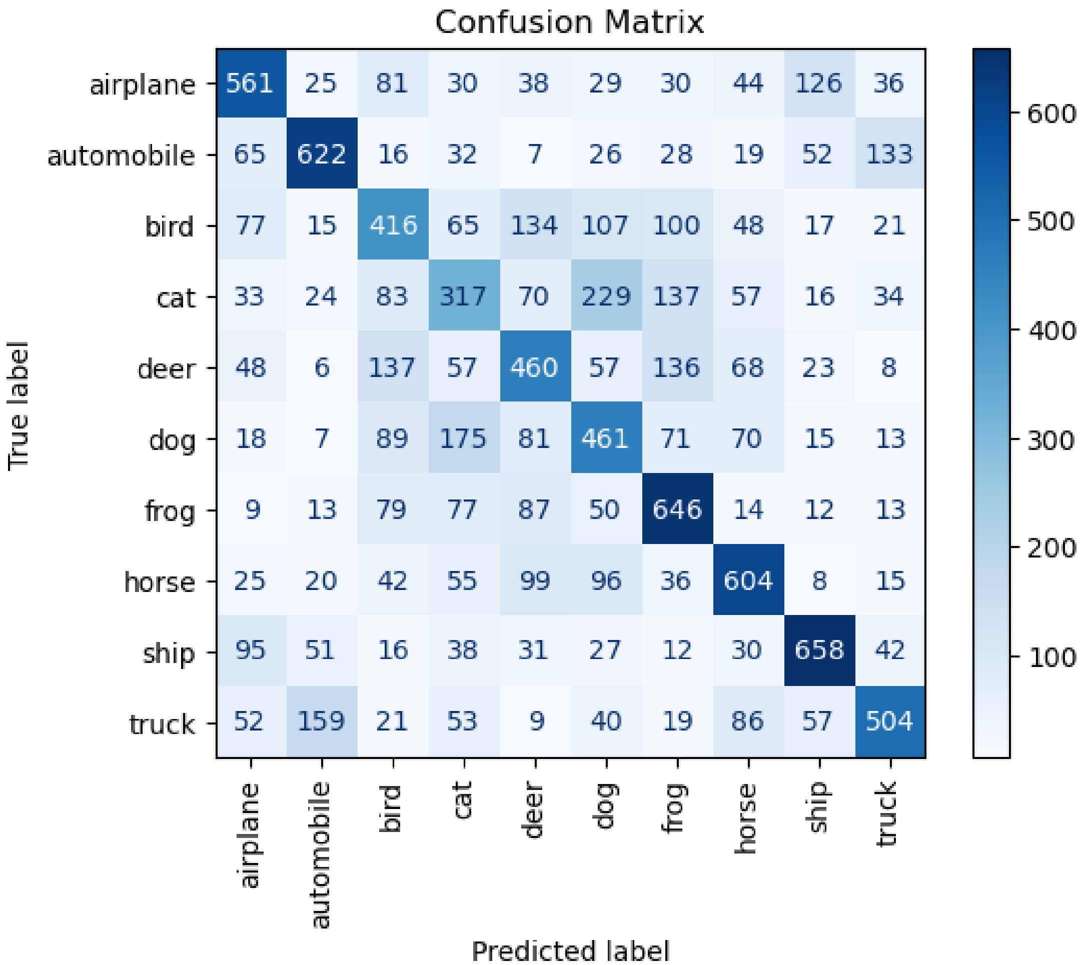
[ ]:
```
plot = plot_accuracies(train_acc, val_acc)
plt.show()
```



[ ]:
```
plot2 = plot_losses(train_losses, val_losses)
plt.show()
```

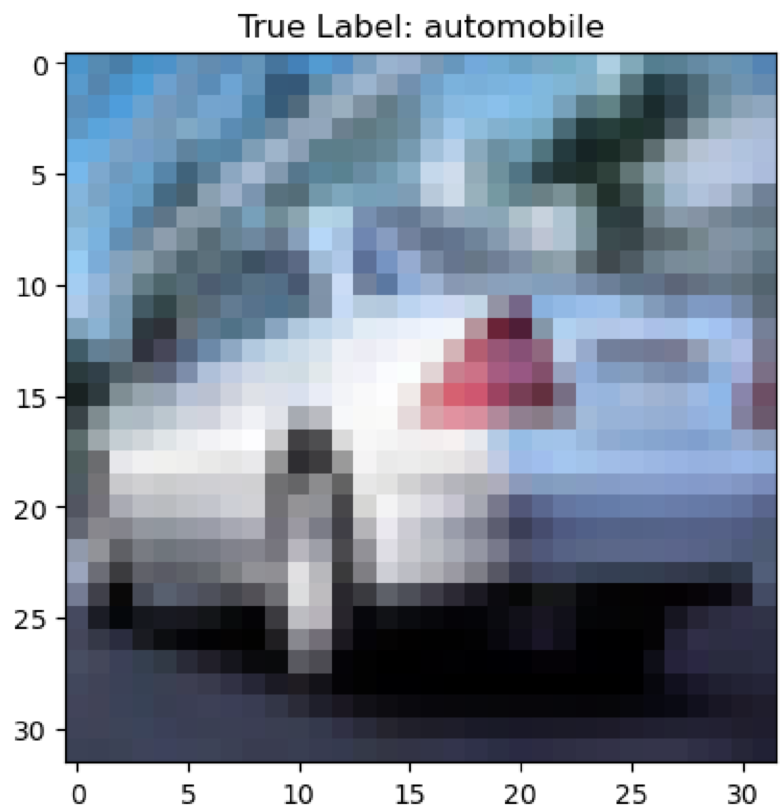
[ ]:

```
# Confusion Matrix

predicted_labels = get_predicted_labels(model, test_loader, device)
true_labels = test_data.targets

cm = confusion_matrix(true_labels, predicted_labels.cpu().numpy())
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=LABELS.
 ↪values())
disp.plot(cmap='Blues', xticks_rotation='vertical')

plt.title('Confusion Matrix')
plt.show()
```

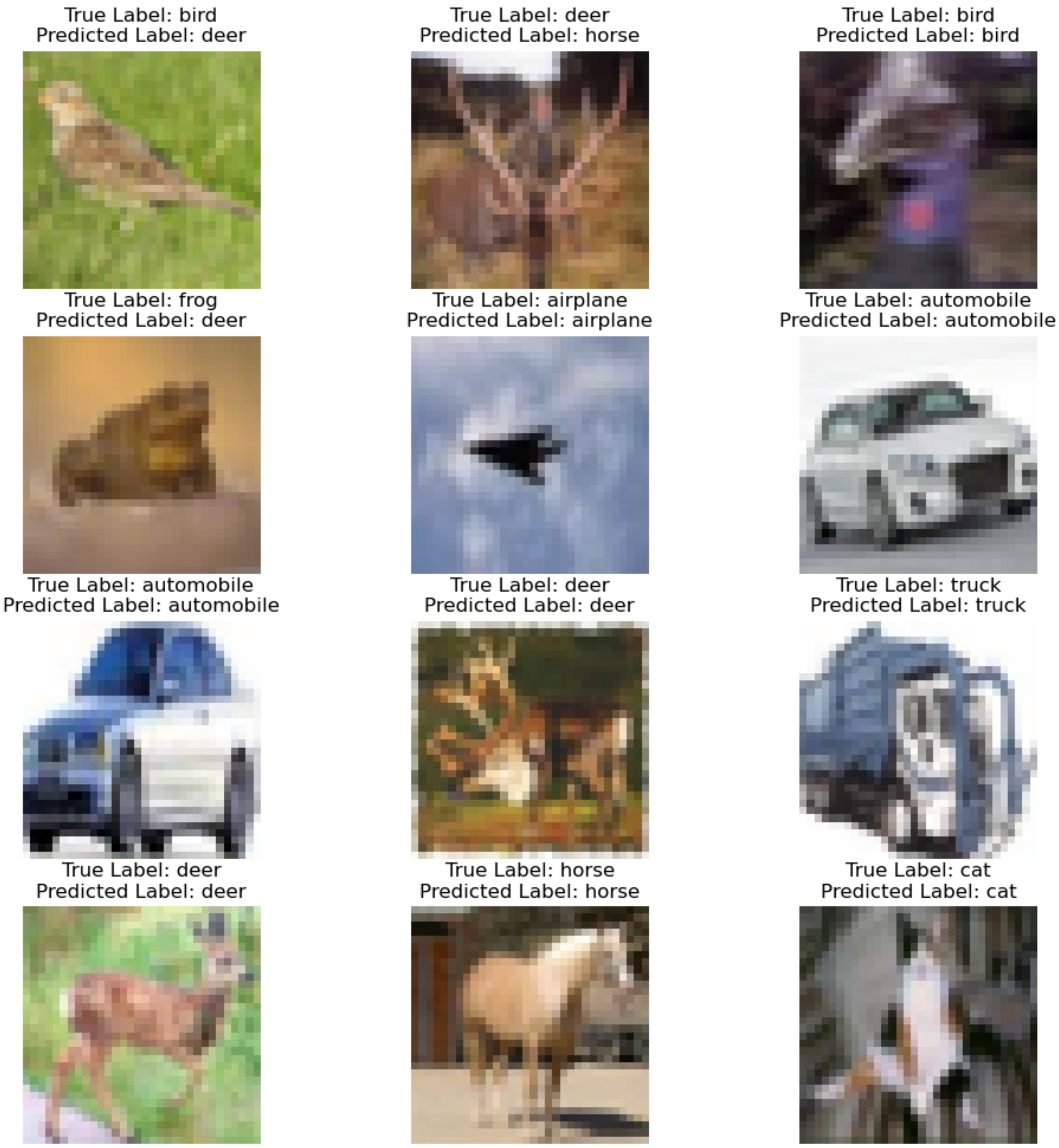


[ ]:
```
# Show an random image and its predicted label

plot, index, label = show_random_image(test_data, mean=mean, std=std)

predicted_label = predicted_labels[index].item()
print(f'True Label: {LABELS[label]}')
print(f'Predicted Label: {LABELS[predicted_label]}')
```

```
True Label: automobile
Predicted Label: ship
```



[ ]:
```
# Take a random 10 images and show their true and predicted labels in a 5*2 grid

fig, ax = plt.subplots(4, 3, figsize=(12, 12))
axes = ax.flatten()

for i in range(12):
    plot, index, label = show_random_image(test_data, mean=mean, std=std)
    predicted_label = predicted_labels[index].item()
    axes[i].set_title(f'True Label: {LABELS[label]}\nPredicted Label:␣
 ↪{LABELS[predicted_label]}')
    axes[i].axis('off')
    axes[i].imshow(plot.get_array().clip(0, 1))
```


True Label: bird
Predicted Label: deer
True Label: deer
Predicted Label: horse
True Label: bird
Predicted Label: bird
True Label: frog
Predicted Label: deer
True Label: airplane
Predicted Label: airplane
True Label: automobile
Predicted Label: automobile
True Label: automobile
Predicted Label: automobile
True Label: deer
Predicted Label: deer
True Label: truck
Predicted Label: truck
True Label: deer
Predicted Label: deer
True Label: horse
Predicted Label: horse
True Label: cat
Predicted Label: cat