EECS 553 HW6

Lingqi Huang

October 2024

1 Problem 1

1.1 (a)

The completed code would be shown below:

```
def compute_train_statistics(self):
    # TODO (part a): compute per-channel mean and std with respect to self.train_dataset

    x_mean = np.mean(self.train_dataset, axis=(0, 1, 2))  # per-channel mean
    x_std = np.std(self.train_dataset, axis=(0, 1, 2))  # per-channel std
    return x_mean, x_std

def get_transforms(self):
    if self.if_resize:
        # TODO (part a): fill in the data transforms
        transform_list = [
            # resize the image to 32x32x3
            transforms.Resize((32, 32)),
            # convert image to PyTorch tensor
            transforms.ToTensor(),
            # normalize the image (use self.x_mean and self.x_std)
            transforms.Normalize(mean = self.x_mean, std = self.x_std)
        ]
    else:
        # TODO (part f): fill in the data transforms
        # Note: Only change from part a) is there is no need to resize the image
            transforms_list = [
            # convert image to PyTorch tensor
            transforms.ToTensor(),
            # normalize the image (use self.x_mean and self.x_std)
            transforms.Normalize(mean = self.x_mean, std = self.x_std)
        ]
    transform = transforms.Compose(transform_list)
    return transform
```

(1): If we resize the image, we could run our algorithm faster because we have less weights or parameters to learn. However, this may loss efficiency because lower the size of image would loss information, and thus decrease the accuracy of prediction. If we normalize the image, we could make the training more stable and could make the algorithm more likely to converge. However, incorrect normalization could introduce bias.

(2): This is because if we compute the mean and sd on validation set, it is a kind of cheating because we are using what we already have to train the model, where in fact the validation set should be unobservable. This may violate the principle and integrity of training model because we are using data that we should never seen.

Thus, by running the code, we can find that the mean of x is [0.5016, 0.4561, 0.3824], and the sd of x is [0.2462, 0.2361, 0.2391].

1.2 (b)

(1): If we connecting the input and the convolutional layer1, we have totally (1 + 5 * 5 * 3) * 16 = 1216 parameters. If we connecting convolutional layer2, we have totally (1 + 5 * 5 * 16) * 32 = 12832 parameters. If we connecting convolutional layer3, we have (1 + 5 * 5 * 32) * 64 = 51264 parameters. If we connecting layer4, we have (1 + 5 * 5 * 64) * 128 = 204928 parameters. If we connect layer5, we will have (128 * 2 * 2 + 1) * 64 = 32832 parameters. If we connect layer6, we have (64 + 1) * 5 = 325 parameters. Thus, we have totally 1216 + 12832 + 51264 + 204928 + 32832 + 325 = 303397 parameters.

(2): If we set all initial weight to be 0, we will produce the same output during the foward pass and then receive the same gradient when start backward propagation algorithm. Also, if we initialize all weight to be zero, this may cause gradient vanishing problem.

1.3 (c)

```python
def __init__(self):
    super().__init__()

    # TODO (part c): define layers
    self.conv1 = nn.Conv2d(3, 16, 5, stride=2, padding=2)  # convolutional layer 1
    self.conv2 = nn.Conv2d(16, 32, 5, stride=2, padding=2) # convolutional layer 2
    self.conv3 = nn.Conv2d(32, 64, 5, stride=2, padding=2) # convolutional layer 3
    self.conv4 = nn.Conv2d(64, 128, 5, stride=2, padding=2) # convolutional layer 4
    self.fc1 = nn.Linear(128 * 2 * 2, 64)    # fully connected layer 1
    self.fc2 =  nn.Linear(64, 5)  # fully connected layer 2 (output layer)

    self.init_weights()

def init_weights(self):
    for conv in [self.conv1, self.conv2, self.conv3, self.conv4]:
        C_in = conv.weight.size(1)
        nn.init.normal_(conv.weight, 0.0, 1 / math.sqrt(5 * 2.5 * C_in))
        nn.init.constant_(conv.bias, 0.0)

    # TODO (part c): initialize parameters for fully connected layers
    for fc in [self.fc1, self.fc2]:
        if fc == self.fc1:
            f_in = fc.weight.size(1)
            nn.init.normal_(fc.weight, 0.0, 1/math.sqrt(256))
            nn.init.constant_(fc.bias, 0)
        if fc == self.fc2:
            f_in = fc.weight.size(1)
            nn.init.normal_(fc.weight, 0.0, 1/math.sqrt(32))
            nn.init.constant_(fc.bias, 0)


def forward(self, x):
    N, C, H, W = x.shape

    # TODO (part c): forward pass of image through the network
    z = F.relu(self.conv1(x))
    z = F.relu(self.conv2(z))
    z = F.relu(self.conv3(z))
    z = F.relu(self.conv4(z))

    z = z.view(z.size(0), -1)

    z = F.relu(self.fc1(z))
    z = self.fc2(z)

    return z
```

And we find that the output the code is same as what we get from part(b).

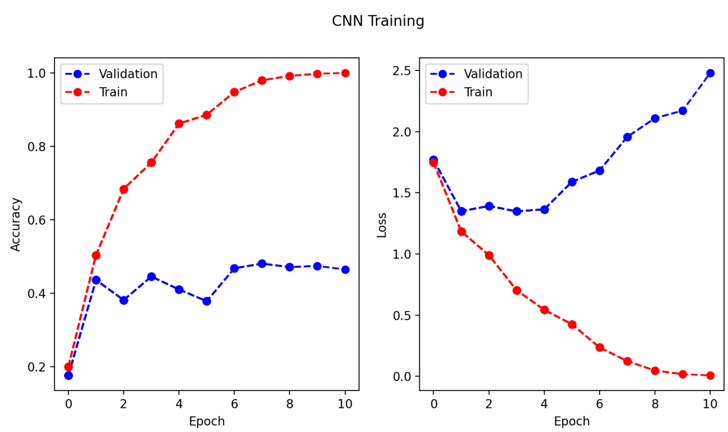
```
if __name__ == '__main__':
    net = CNN()
    print(net)
    print('Number of CNN parameters: {}'.format(count_parameters(net)))
    dataset = DogDataset()
    images, labels = next(iter(dataset.train_loader))
    print('Size of model output:', net(images).size())
```

```
CNN(
  (conv1): Conv2d(3, 16, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (conv3): Conv2d(32, 64, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (conv4): Conv2d(64, 128, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
  (fc1): Linear(in_features=512, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=5, bias=True)
)
Number of CNN parameters: 303397
Size of model output: torch.Size([4, 5])
```

1.4 (d)

```
def predictions(logits):
    """
    Compute the predictions from the model.
    Inputs:
        - logits: output of our model based on some input, tensor with shape=(batch_size, num_classes)
    Returns:
        - pred: predictions of our model, tensor with shape=(batch_size)
    """
    # TODO (part d): compute the predictions

    pred = torch.argmax(logits, dim = 1)


    return pred
# torch.zeros(logits.size(0))


def accuracy(y_true, y_pred):
    """
    Compute the accuracy given true and predicted labels.
    Inputs:
        - y_true: true labels, tensor with shape=(num_examples)
        - y_pred: predicted labels, tensor with shape=(num_examples)
    Returns:
        - acc: accuracy, float
    """
    # TODO (part d): compute the accuracy
    correct = (y_true == y_pred).sum().item()
    total = y_true.size(0)

    acc = correct / total
    return acc
```

```python
def train(config, dataset, model):
    # Data loaders
    train_loader, val_loader = dataset.train_loader, dataset.val_loader

    if 'use_weighted' not in config:
        # TODO (part d): define loss function
        criterion = nn.CrossEntropyLoss()
    else:
        # TODO (part h): define weighted loss function
        criterion = None

    # TODO (part d): define optimizer
    learning_rate = config['learning_rate']
    momentum = config['momentum']
    optimizer = optim.SGD(model.parameters(), lr = learning_rate, momentum = momentum)

    # Attempts to restore the latest checkpoint if exists
    print('Loading model...')
    force = config['ckpt_force'] if 'ckpt_force' in config else False
    model, start_epoch, stats = checkpoint.restore_checkpoint(model, config['ckpt_path'], force=force)

    # Create plotter
    plot_name = config['plot_name'] if 'plot_name' in config else 'CNN'
    plotter = Plotter(stats, plot_name)

    # Evaluate the model
    _evaluate_epoch(plotter, train_loader, val_loader, model, criterion, start_epoch)

    # Loop over the entire dataset multiple times
    for epoch in range(start_epoch, config['num_epoch']):
        # Train model on training set
        _train_epoch(train_loader, model, criterion, optimizer)

        # Evaluate model on training and validation set
        _evaluate_epoch(plotter, train_loader, val_loader, model, criterion, epoch + 1)

        # Save model parameters
        checkpoint.save_checkpoint(model, epoch + 1, config['ckpt_path'], plotter.stats)

    print('Finished Training')

    # Save figure and keep plot open
    plotter.save_cnn_training_plot()
    plotter.hold_training_plot()
```



We finally find that at Epoch 10, the validation loss is about 2.239, the validation accuracy is about 0.474, the train loss is about 0.0083, and the train accuracy is 1.0.

1.5 (e)

(1) : It seems that at first the training loss and the validation loss will both decrease. Then after certain number of epochs, the training loss will continue to decrease but the validation loss will then become higher and higher. If we continue, the training loss will then decrease to 0 but validation error may continue to increase.

(2) : It seems that we should stop as epoch of 4 according the graph. If we trying to maximize the training accuracy, it may cause overfitting that our model performs very well on the training set, but poorly on the validation set.

1.6 (f)

```python
def get_transforms(self):
    if self.if_resize:
        # TODO (part a): fill in the data transforms
        transform_list = [
            # resize the image to 32x32x3
            transforms.Resize((32, 32)),
            # convert image to PyTorch tensor
            transforms.ToTensor(),
            # normalize the image (use self.x_mean and self.x_std)
            transforms.Normalize(mean = self.x_mean, std = self.x_std)
        ]
    else:
        # TODO (part f): fill in the data transforms
        # Note: Only change from part a) is there is no need to resize the image
            transform_list = [
            # convert image to PyTorch tensor
            transforms.ToTensor(),
            # normalize the image (use self.x_mean and self.x_std)
            transforms.Normalize(mean = self.x_mean, std = self.x_std)
        ]
    transform = transforms.Compose(transform_list)
    return transform
```