EECS-553 Homework 4

Vipransh Sinha

April 27, 2025

Exercise 1 Solution

a) One example of a simple 1D dataset that is linearly seperable with offset but not serpable without offest is $x_1 = -1, y_1 = -1$ and $x_2 = 1, y_2 = 1$. With an offset, we can easily seperate this dataset with a hyperplane at $x = 0$. Without an offest, the hyperplane must go through the origin, and thus unable to seperate -1 and 1, leading to the garunteed misclassification of at least one of the points.

b) Given the problem:

$$\begin{aligned} \min_{w,\xi} \quad & \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \\ s.t. \quad & y_i(\langle w, x_i\rangle) \geq 1 - \xi_i, \xi_i \geq 0 \end{aligned}$$

We get the lagrangian of:

$$L(w, \xi, \alpha, \beta) = \frac{1}{2}\|w\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i\langle w, x_i\rangle - 1 + \xi_i) - \sum_{i=1}^{n}\beta_i\xi_i$$

Taking the gradient and setting it to 0, we get the following equations:

$$\begin{aligned} \nabla_w L &= w - \sum_{i=1}^{n}\alpha_i y_i x_i = 0 \\ \nabla_{\xi_i} L &= \frac{C}{n} - \alpha_i - \beta_i = 0 \end{aligned}$$

This gives us:

$$\begin{aligned} w &= \sum_{i=1}^{n}\alpha_i y_i x_i \\ \alpha_i + \beta_i &= \frac{C}{n} \end{aligned}$$

Substituting back into the Lagrangian we get:

$$\begin{aligned} \max_\alpha \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle \\ s.t \quad & 0 \leq \alpha_i \leq \frac{C}{n} \end{aligned}$$

This is the dual quadratic program.

c) Setting all inner products to kernel functions we get:

$$\begin{aligned} \max_\alpha \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ s.t \quad & 0 \leq \alpha_i \leq \frac{C}{n} \end{aligned}$$

This gives us a final classifier of

$$f(x) = \sum_{i=1}^{n} \alpha_i y_i k(x_1, x)$$

Exercise 2 Solution

a) After running the code we get the computed statistics of [0.50161401 0.45612724 0.38243697] for mean and [0.24604 0.23611233 0.23892746] for std. The tradeoff for resizing and normalizing images is that we get a reduced computational load (due to smaller images) and helps the model converge faster due to normalized images, but we also lose fine details in the images from resizing and wrong normalization could cause huge errors. We compute mean and std only on training images as the model only learns from the training images, so the normalization should only be based on these set of images. Using the validation dataset could cause information to leak to the model, causing possible overfitting. Code for part a:

```
def compute_train_statistics(self):
        # TODO (part a): compute per-channel mean and std with respect to self.train_dat
        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):
        mean = self.x_mean.tolist()
        std = self.x_std.tolist()

        if self.if_resize:
            # TODO (part a): fill in the data transforms
            transform_list = [
```

```
            transforms.Resize((32,32)),
            transforms.ToTensor(),
            transforms.Normalize(mean=mean, std=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
            # normalize the image (use self.x_mean and self.x_std)
            transforms.ToTensor(),
            transforms.Normalize(mean=mean, std=std)
        ]
    transform = transforms.Compose(transform_list)
    return transform
```

b) There are $75 \times 16 + 16 = 1216$ parameters in conv layer 1, $400 \times 32 + 32 = 12832$ parameters in conv layer 2, $800 \times 64 + 64 = 51264$ parameters in conv layer 3, $1600 \times 128 + 128 = 204928$ parameters in conv layer 4, $32768 + 64 = 32832$ parameters in fully connected layer 1, and $320 + 5 = 325$ parameters in fully connected layer 2. Summing these we get a total of 303,397 total parameters in this CNN. We randomly initialize these parameters to allow for proper learning in the model. For example, if the parameters were all initialized to 0, the gradient would not move, and thus learning would not be possible.

c) Here is my code for part c:

```
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(512,64)   # 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
        nn.init.normal_(self.fc1.weight, mean=0.0,std=math.sqrt(1.0/256))
        nn.init.constant_(self.fc1.bias, 0.0)
        nn.init.normal_(self.fc2.weight,mean=0.0,std=math.sqrt(1.0/32))
        nn.init.constant(self.fc2.bias, 0.0)

    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(N, -1)
        z = F.relu(self.fc1(z))
        z = self.fc2(z)

        return z
```

d) For part d I get the following graph is below. Code for part 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
    Returns:
        - pred: predictions of our model, tensor with shape=(batch_size)
    """
    return torch.argmax(logits, dim=1)


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
    """
    return (y_true == y_pred).float().mean().item()
```

```
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 = torch.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 = torch.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']

    # 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()
```

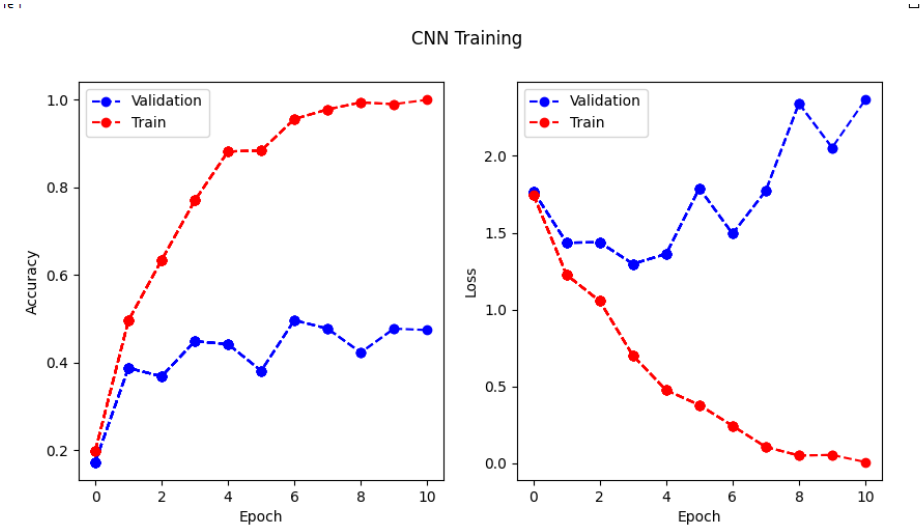

Figure 1: Part d output

```
plotter.hold_training_plot()
```

We find the final Validation Loss to be 2.36, the final validation accuracy to be 0.47, the final train loss to be 0.0089, and the final train accuracy to be 1.0 after running train.py.

e) As the training loss decreases, the validation loss increases. I predict that if we keep training the model, the training loss will reach 0, and the validation loss would flatten out at some value. We should stop training our model at the local minima of the training loss. We should not try to maximiza the trainig accuracy, as this just means we are over-fitting the model, which is not the goal. We want our model to work on all and new datasets.

f) We find a validation loss of 0.1799, a validation accuracy of 0.9262, a train loss of 0.1150, and a train accuracy of 0.9559. Here is the plot generated and the code used:

```
def get_transforms(self):
        mean = self.x_mean.tolist()
        std = self.x_std.tolist()

        if self.if_resize:
            # TODO (part a): fill in the data transforms
            transform_list = [
                transforms.Resize((32,32)),
                transforms.ToTensor(),
                transforms.Normalize(mean=mean, std=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
                # normalize the image (use self.x_mean and self.x_std)
                transforms.ToTensor(),
                transforms.Normalize(mean=mean, std=std)

            ]
        transform = transforms.Compose(transform_list)
        return transform

def load_pretrained(num_classes=5):
    """
    Load a ResNet-18 model from `torchvision.models` with pre-trained weights. Freeze al
    final layer by setting the flag `requires_grad` for each parameter to False. Replace
    with another fully connected layer with `num_classes` many output units.
    Inputs:
        - num_classes: int
    Returns:
        - model: PyTorch model
    """
    # TODO (part f): load a pre-trained ResNet-18 model
    model = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)


    for param in model.parameters():
        param.requires_grad = False
    in_features = model.fc.in_features
    model.fc = torch.nn.Linear(in_features, num_classes)
    return model
```

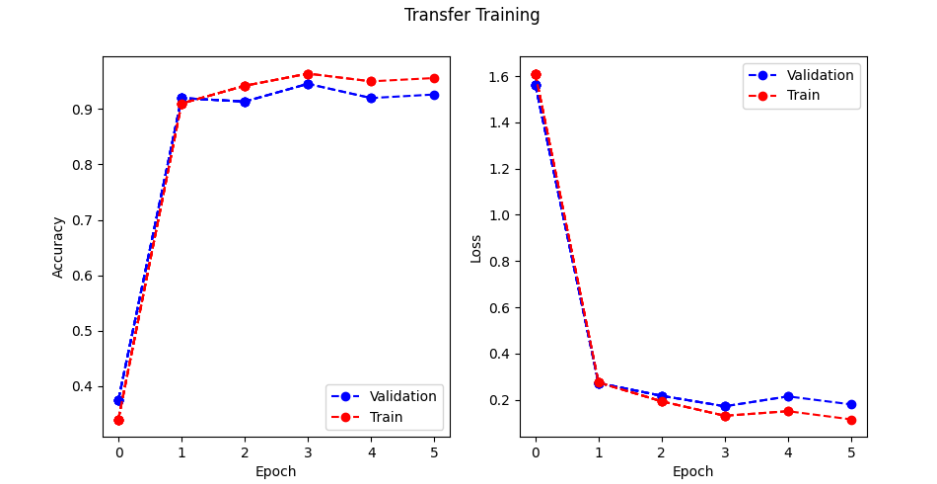

Figure 2: Part f Plot