Department of Electronic & Telecommunication Engineering

University of Moratuwa

EN3160 - Image Processing and Machine Vision

Assignment 03

Neural Networks

210325M - Kuruppu M.P.

Date - 2024.11.11

Github Repository Link : https://github.com/MIHIRAJA-KURUPPU/Neural-Networks-Assignment

1 Question 01

Define the Network with a Hidden Layer and Sigmoid Activation

```
class SimpleNN(nn.Module):
    def __init__(self, Din, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.hidden = nn.Linear(Din, hidden_size)  # Hidden layer
        self.sigmoid = nn.Sigmoid()  # Sigmoid activation
        self.output = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        x = self.hidden(x)
        x = self.sigmoid(x)  # Apply sigmoid activation
        x = self.output(x)
        return x
Din = 3 * 32 * 32  # Input size (flattened CIFAR-10 image size)
output_size = 10  # Output size (CIFAR-10 classes)
hidden_size = 100  # Number of neurons in the hidden layer
model = SimpleNN(Din, hidden_size, output_size)
```

Listing 1: Define the Network

First we initialize the CIFAR-10 dataset with two transformation: the conversion to tensor format and normalization with mean 0.5 and standard deviation 0.5. Then we define the neural network model with one hidden layer with 100 nodes and a sigmoid activation function. The input is flattened CIFAR-10 image, and the output consists of ten classes.

Loss and Optimizer & Training Loop

```
criterion = nn.CrossEntropyLoss()  # Classification loss
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
iterations = 10
loss_history = []
for epoch in range(iterations):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        inputs, labels = inputs.view(inputs.size(0), -1),labels
        optimizer.zero_grad()  # Zero the parameter gradients
        outputs = model(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Loss computation
        loss.backward()  # Backpropagation
        optimizer.step()  # Update weights
        running_loss += loss.item()
    average_loss = running_loss / len(trainloader)
```

```
    loss_history.append(average_loss)
    print(f"Epoch {epoch+1}/{iterations}, Loss: {average_loss:.4f}")
```

Listing 2: Training Loop

The model uses cross-entropy loss suitable for classification tasks and the SGD optimizer with momentum to update weights.
This training loop iterates over the dataset multiple times, computing the loss, performing backpropagation, and updating weights using the optimizer. The average loss for each epoch is recorded in `loss_history`.The loss history recorded during training, is visualized providing insight into model convergence.

Training and Test Accuracy Calculation

The accuracy of the model is calculated on both the training and test sets by comparing the model's predictions with the true labels.

```
Files already downloaded and verified
Files already downloaded and verified
Epoch 1/10, Loss: 1.8433
Epoch 2/10, Loss: 1.6893
Epoch 3/10, Loss: 1.6249
Epoch 4/10, Loss: 1.5746
Epoch 5/10, Loss: 1.5259
Epoch 6/10, Loss: 1.4836
Epoch 7/10, Loss: 1.4474
Epoch 8/10, Loss: 1.4095
Epoch 9/10, Loss: 1.3782
Epoch 10/10, Loss: 1.3474
```

Figure 1: Loss after each iteration

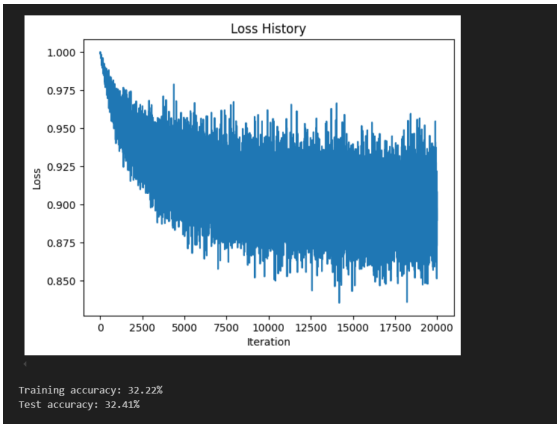


Figure 2: single dense layer network

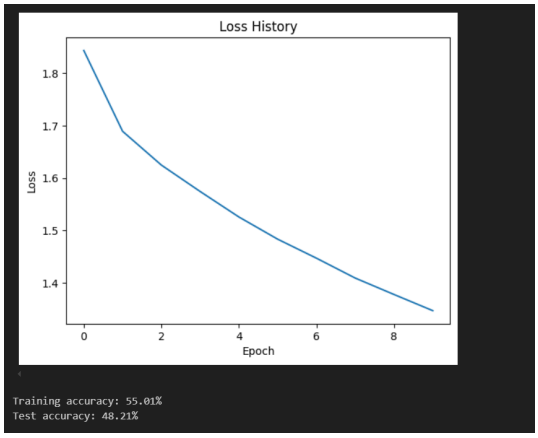


Figure 3: After the modification

Model Performance Comparison

The single-layer model, which consists of only an input and output layer, achieved the following results after 20 epochs of training:

- **Training Accuracy**: 32.22%
- **Test Accuracy**: 32.41%

The performance of the single-layer model was relatively low, with both training and test accuracies around 32%. The absence of a hidden layer limits the model's ability to capture complex patterns within the CIFAR-10 dataset.

The multi-layer model includes an additional hidden layer with a sigmoid activation function. This model was trained for 10 epochs.

- **Training Accuracy**: 55.01%
- **Test Accuracy**: 48.21%

The multi-layer model demonstrated a significant improvement in performance. The training accuracy increased to 55.01% and the test accuracy to 48.21%. The introduction of the hidden layer enabled the model to learn more complex features, leading to better generalization.

2 Question 02

```
train_val_dataset = datasets.MNIST(root="./datasets/", train=True
    , download=False, transform=transforms.ToTensor())
test_dataset = datasets.MNIST(root="./datasets", train=False,
    download=False, transform=transforms.ToTensor())

imgs = torch.stack([img for img, _ in train_val_dataset], dim=0)
mean = imgs.view(1, -1).mean(dim=1) #Calculate mean
std = imgs.view(1, -1).std(dim=1)    # Calculate std
mnist_transforms = transforms.Compose([transforms.ToTensor(),
    transfoms.Normalize(mean=mean, std=std)])

train_val_dataset = datasets.MNIST(root="./datasets/", train=True
    , download=False, transform=mnist_transforms)
test_dataset = datasets.MNIST(root="./datasets/", train=False,
    download=False, transform=mnist_transforms)
```

Here we load the MNIST dataset (both training and test sets) and compute the mean and standard deviation of all pixel values in the training dataset. Then we create a transformation pipeline that converts images to tensors and normalizes them using the computed mean and standard deviation. Then the transformation pipeline is applied to both the training and test datasets.

```
train_size = int(0.9 * len(train_val_dataset))
val_size = len(train_val_dataset) - train_size

```

```
train_dataset, val_dataset = torch.utils.data.random_split(
    dataset=train_val_dataset, lengths=[train_size, val_size])
from torch.utils.data import DataLoader

BATCH_SIZE = 32

train_dataloader = DataLoader(dataset=train_dataset, batch_size=
    BATCH_SIZE, shuffle=True)
val_dataloader = DataLoader(dataset=val_dataset, batch_size=
    BATCH_SIZE, shuffle=True)
test_dataloader = DataLoader(dataset=test_dataset, batch_size=
    BATCH_SIZE, shuffle=False)
```

Here we split the dataset (train_val_dataset) into training and validation sets, with 90% of the data used for training and 10% for validation. It then creates DataLoader instances for the training, validation, and test datasets, each of which loads data in batches of 32.

```
from torch import nn
class LeNet5V1(nn.Module):
    def __init__(self):
        super().__init__()
        self.feature = nn.Sequential(
            # First convolutional block
            nn.Conv2d(in_channels=1, out_channels=6, kernel_size
                =5, stride=1, padding=2),
            nn.Tanh(),  # Activation function
            nn.AvgPool2d(kernel_size=2, stride=2),

            # Second convolutional block
            nn.Conv2d(in_channels=6, out_channels=16, kernel_size
                =5, stride=1),  # Input: 14x14 -> Output: 10x10
            nn.Tanh(),  # Activation function
            nn.AvgPool2d(kernel_size=2, stride=2),  #10x10 -> 5x5
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),  # Flatten the output to a 1D vector
            nn.Linear(in_features=16*5*5, out_features=120),
            nn.Tanh(),  # Activation function
            nn.Linear(in_features=120, out_features=84),
            nn.Tanh(),  # Activation function
            nn.Linear(in_features=84, out_features=10),  # Output
                layer: 84 -> 10 (for 10 classes))
    def forward(self, x):
        # Pass input through the convolutional layers followed by
            fully connected layers
        return self.classifier(self.feature(x))

model_lenet5v1 = LeNet5V1()
```

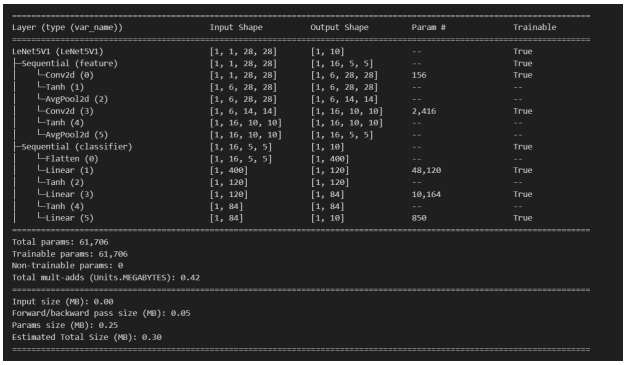
Feature Extraction (Convolutional Layers)

Convolutional layers and pooling layers are responsible for detecting patterns and reducing the spatial dimensions of the image.

- The first convolutional layer takes a grayscale image, applies 6 filters, each having a kernel size of 5x5. A stride of 1 ensures that the kernel moves one pixel at a time. In LeNet-5 architecture input is shown as 32x32. But in our dataset images are 28x28. So input images (28x28) have been padded with 2 pixels all sides to make it a 32x32 dimension.
- The second convolutional layer takes the 6-channel output from the previous layer and applies 16 filters, each with a kernel size of 5x5, to extract higher-level features.
- activation function: The Tanh activation function is applied after each convolutional layer. This function introduces non-linearity and enables the network to learn complex patterns.
- The average pooling layer with a kernel size of 2x2 and a stride of 2 is applied after each convolutional layer. Pooling reduces the spatial dimensions of the feature map by a factor of 2.

Classifier (Fully Connected Layers)

- Flattening the Output: This layer flattens the 3D output from the convolutional layers into a 1D vector.
- Fully Connected Layers: The first fully connected layer takes the flattened output (400 features) and reduces it to 120 features. The second fully connected layer further reduces the 120 features to 84 features. The final output layer reduces the 84 features to 10 features, corresponding to the 10 possible output classes (digits 0-9).



Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
LeNet5V1 (LeNet5V1)	[1, 1, 28, 28]	[1, 10]	--	True
├─Sequential (feature)	[1, 1, 28, 28]	[1, 16, 5, 5]	--	True
│ └─Conv2d (0)	[1, 1, 28, 28]	[1, 6, 28, 28]	156	True
│ └─Tanh (1)	[1, 6, 28, 28]	[1, 6, 28, 28]	--	--
│ └─AvgPool2d (2)	[1, 6, 28, 28]	[1, 6, 14, 14]	--	--
│ └─Conv2d (3)	[1, 6, 14, 14]	[1, 16, 10, 10]	2,416	True
│ └─Tanh (4)	[1, 16, 10, 10]	[1, 16, 10, 10]	--	--
│ └─AvgPool2d (5)	[1, 16, 10, 10]	[1, 16, 5, 5]	--	--
├─Sequential (classifier)	[1, 16, 5, 5]	[1, 10]	--	True
│ └─Flatten (0)	[1, 16, 5, 5]	[1, 400]	--	--
│ └─Linear (1)	[1, 400]	[1, 120]	48,120	True
│ └─Tanh (2)	[1, 120]	[1, 120]	--	--
│ └─Linear (3)	[1, 120]	[1, 84]	10,164	True
│ └─Tanh (4)	[1, 84]	[1, 84]	--	--
│ └─Linear (5)	[1, 84]	[1, 10]	850	True

Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.42

Input size (MB): 0.00
Forward/backward pass size (MB): 0.05
Params size (MB): 0.25
Estimated Total Size (MB): 0.30

Figure 4: Model Summery

```python
from torchmetrics import Accuracy

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=model_lenet5v1.parameters(),
    lr=0.001)
accuracy = Accuracy(task='multiclass', num_classes=10)

from tqdm.notebook import tqdm
from torch.utils.tensorboard import SummaryWriter

from datetime import datetime
import os

# Experiment tracking
timestamp = datetime.now().strftime("%Y-%m-%d")
experiment_name = "MNIST"
model_name = "LeNet5V1"
log_dir = os.path.join("runs", timestamp, experiment_name,
    model_name)
writer = SummaryWriter(log_dir)

EPOCHS = 12

for epoch in tqdm(range(EPOCHS)):
    # Training loop
    train_loss, train_acc = 0.0, 0.0
    for X, y in train_dataloader:
        model_lenet5v1.train()

        y_pred = model_lenet5v1(X)

        loss = loss_fn(y_pred, y)
        train_loss += loss.item()

        acc = accuracy(y_pred, y)
        train_acc += acc

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    train_loss /= len(train_dataloader)
    train_acc /= len(train_dataloader)

    # Validation loop
    val_loss, val_acc = 0.0, 0.0
    model_lenet5v1.eval()
```

```
    with torch.inference_mode():
        for X, y in val_dataloader:
            y_pred = model_lenet5v1(X)

            loss = loss_fn(y_pred, y)
            val_loss += loss.item()

            acc = accuracy(y_pred, y)
            val_acc += acc

        val_loss /= len(val_dataloader)
        val_acc /= len(val_dataloader)

    writer.add_scalars(main_tag="Loss", tag_scalar_dict={"train/
       loss": train_loss, "val/loss": val_loss}, global_step=
       epoch)
    writer.add_scalars(main_tag="Accuracy", tag_scalar_dict={"
       train/acc": train_acc, "val/acc": val_acc}, global_step=
       epoch)

    print(f"Epoch: {epoch}| Train loss: {train_loss: .5f}| Train
       acc: {train_acc: .5f}| Val loss: {val_loss: .5f}| Val acc:
        {val_acc: .5f}")
```

This code trains and evaluates a LeNet-5-based model on MNIST using PyTorch. It initializes cross-entropy loss, Adam optimizer, and an accuracy metric for multiclass classification.The training loop iterates over epochs, computing loss and accuracy for each batch, applying backpropagation to update model weights.

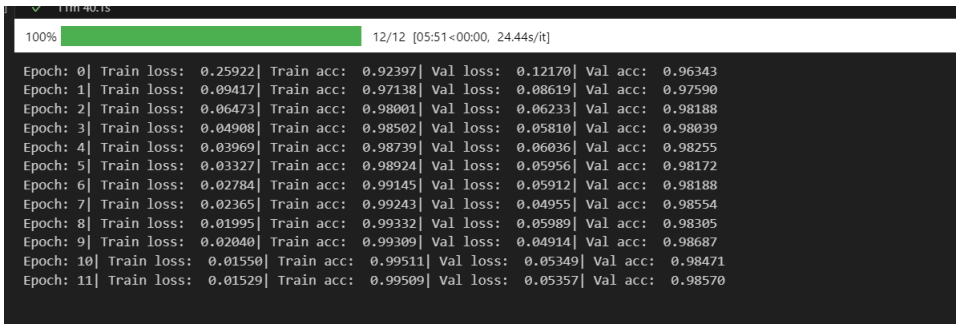


Figure 5: Loss after each iteration

Test Accuracy

The test accuracy was nearly 98.6% which suggests that the model is able to generalize on unseen digits well.

```
test_loss, test_acc = 0, 0

model_lenet5_v1_mnist_loaded.to(device)

model_lenet5_v1_mnist_loaded.eval()
with torch.inference_mode():
    for X, y in test_dataloader:
        X, y = X.to(device), y.to(device)
        y_pred = model_lenet5_v1_mnist_loaded(X)

        test_loss += loss_fn(y_pred, y)
        test_acc += accuracy(y_pred, y)

    test_loss /= len(test_dataloader)
    test_acc /= len(test_dataloader)

print(f"Test loss: {test_loss: .5f}| Test acc: {test_acc: .5f}")
✓ 9.8s
Test loss:  0.05078| Test acc:  0.98622
```

Figure 6: Test Accuracy

3 Question 03

```
data_dir = 'hymenoptera_data'
zip_path = os.path.join(data_dir, 'hymenoptera_data.zip')
if not os.path.exists(data_dir):
    # If the directory does not exist, create it
    os.makedirs(data_dir, exist_ok=True)
    # Download the dataset zip file
    url = 'https://download.pytorch.org/tutorial/hymenoptera_data.zip'
    print("Downloading the dataset...")
    urllib.request.urlretrieve(url, zip_path)
    # Unzip the dataset into the specified directory
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(data_dir)

    # Remove the zip file after extraction to save space
    os.remove(zip_path)

    print(f"Dataset downloaded and extracted to '{data_dir}'")
else:
    print(f"Dataset already exists in '{data_dir}'. Skipping download.")
```

This code checks if the dataset directory already exists. If not, it downloads, extracts, and then deletes the zip file.

```
# Data augmentation and normalization for training
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
```

```
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229,
            0.224, 0.225])]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229,
            0.224, 0.225])]),}

data_dir = 'hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir,
    x), data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
    batch_size=4, shuffle=True, num_workers=4) for x in ['train',
    'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', '
    val']}
class_names = image_datasets['train'].classes
device = torch.device("cuda:0" if torch.cuda.is_available() else
    "cpu")
```

This code sets up data augmentation and normalization transformations for training and validation data. It loads the images, creates DataLoader objects for efficient batching, and assigns class names and dataset sizes.

```
def train_model(model, criterion, optimizer, scheduler,
    num_epochs=25):
    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        best_model_params_path = os.path.join(tempdir, '
            best_model_params.pt')
        torch.save(model.state_dict(), best_model_params_path)
        best_acc = 0.0
        for epoch in range(num_epochs):
            print(f'Epoch {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 data.
```

```
                for inputs, labels in dataloaders[phase]:
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    # zero the parameter gradients
                    optimizer.zero_grad()
                    # forward
                    with torch.set_grad_enabled(phase == 'train'):
                        outputs = model(inputs)
                        _, preds = torch.max(outputs, 1)
                        loss = criterion(outputs, labels)
                        # backward + optimize only if in training
                        if phase == 'train':
                            loss.backward()
                            optimizer.step()

                    # statistics
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds == labels.data)
                if phase == 'train':
                    scheduler.step()

                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() / dataset_sizes[phase]

                print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

                # deep copy the model
                if phase == 'val' and epoch_acc > best_acc:
                    best_acc = epoch_acc
                    torch.save(model.state_dict(), best_model_params_path)

            print()

        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
        print(f'Best val Acc: {best_acc:4f}')

        # load best model weights
        model.load_state_dict(torch.load(best_model_params_path, weights_only=True))
    return model
```

The `train_model` function trains a model over multiple epochs, managing both training and validation phases.

1. Model Checkpoints : Saves the best model parameters to a temporary directory based on validation accuracy.

2. Training and Validation Phases: For each epoch, the model performs forward and backward passes, updating weights, and tracking cumulative loss and accuracy.

3. Best Model Selection: After training, the model reloads the parameters achieving the highest validation accuracy.

3.1 Finetuning the ConvNet

```
model_ft = models.resnet18(weights='IMAGENET1K_V1')
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 2.
model_ft.fc = nn.Linear(num_ftrs, 2)
model_ft = model_ft.to(device)
criterion = nn.CrossEntropyLoss()
# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001,
   momentum=0.9)
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
    gamma=0.1)
model_ft = train_model(model_ft, criterion, optimizer_ft,
   exp_lr_scheduler,
    num_epochs=25)
```

The code initializes a fine-tuning process for the pretrained ResNet-18 model.

1. The ResNet-18 model is loaded with ImageNet weights, with its final fully connected layer adapted to output two classes.

2. A cross-entropy loss function is defined, and an SGD optimizer is configured to update all model parameters, with a learning rate of 0.001 and momentum of 0.9. The learning rate is decreased by a factor of 0.1 every 7 epochs to facilitate convergence.

3. train_model function is called to train the model over 25 epochs.

The model learned well during training and performed well on new data, with the best accuracy on the validation set being 93.46%

```
Epoch 20/24
----------
train Loss: 0.2802 Acc: 0.8770
val Loss: 0.2083 Acc: 0.9346

Epoch 21/24
----------
train Loss: 0.2549 Acc: 0.8934
val Loss: 0.2341 Acc: 0.9281

Epoch 22/24
----------
train Loss: 0.2718 Acc: 0.8648
val Loss: 0.2048 Acc: 0.9216

Epoch 23/24
----------
train Loss: 0.2693 Acc: 0.8770
val Loss: 0.2291 Acc: 0.9281

Epoch 24/24
----------
train Loss: 0.2855 Acc: 0.8852
val Loss: 0.2145 Acc: 0.9216

Training complete in 22m 21s
Best val Acc: 0.934641
```

Figure 7: Loss and accuracy after each iteration in Finetuning the ConvNet

3.2 ConvNet as fixed feature extractor

```
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')
for param in model_conv.parameters():
    param.requires_grad = False
# Parameters of newly constructed modules have requires_grad=True
    by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)
model_conv = model_conv.to(device)
criterion = nn.CrossEntropyLoss()
# Observe that only parameters of final layer are being optimized
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001,
   momentum=0.9)
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size
   =7, gamma=0.1)
model_conv = train_model(model_conv, criterion, optimizer_conv,
   exp_lr_scheduler, num_epochs=25)
```

The pre-trained ResNet-18 model is used, with only the final layer being trained to fit the binary classification. The other layers are frozen to retain the learned features from ImageNet. The training is done with a small learning rate of 0.001 and momentum of 0.9. A learning rate scheduler is used to reduce the learning rate every 7 epochs.
The model learned well during training and performed well on new data, with the best accuracy on the validation set being 96.07%

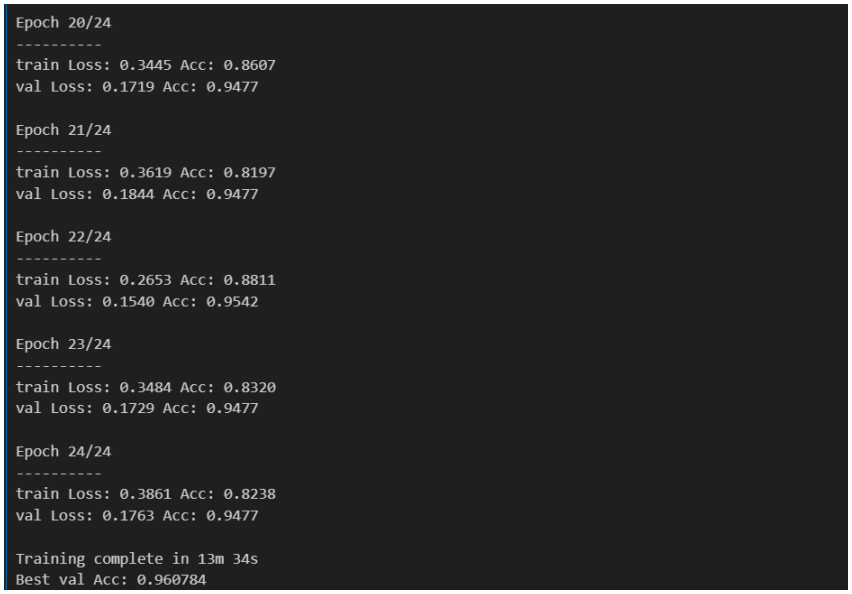


Figure 8: Loss and accuracy after each iteration in ConvNet as fixed feature extractor

3.3 Comparison of Fine-Tuned Model and Feature Extractor Model

Metric	Fine-Tuned Model	Feature Extractor Model
Training Accuracy	98.77%	92.62%
Validation Accuracy	93.46%	96.08%

Table 1: Comparison of Fine-Tuned Model and Feature Extractor Model Accuracy

- **Training Accuracy:**
  - The Fine-Tuned Model has a much higher training accuracy of 98.77%, indicating that it has learned well from the training data and can predict it accurately.
  - The Feature Extractor Model has a lower training accuracy of 92.62%, which is expected as it only trains the final layer while keeping the rest of the model frozen.
- **Validation Accuracy:**
  - The Fine-Tuned Model has lower validation accuracy of 93.46%, may be because of slight overfitting, as it performs well on the training data but generalizes lower on unseen validation data.
  - The Feature Extractor Model has a higher validation accuracy of 96.08%, suggesting better generalization to unseen data, possibly as it depends on pre-trained features.