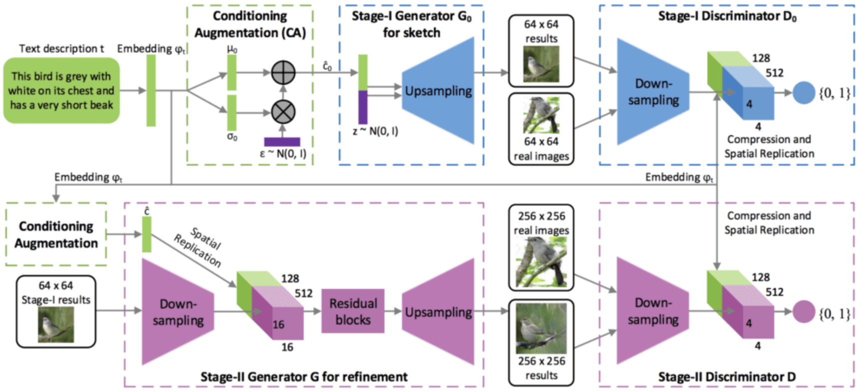
Deep Learning
ISLP Lab Solution



Kunal Gokhe

MT2309

2024

Contents

Single Layer Neural Network on Hitters Data

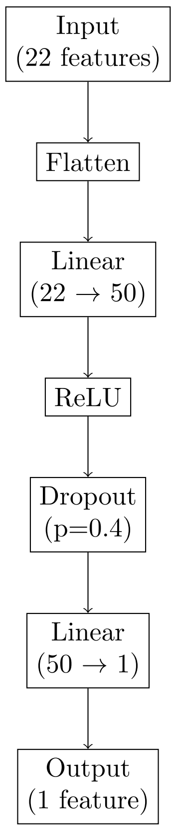
Introduction

In this task, we will design the neural network for predicting the Salary based on the predictor variables. We will design the network with 22 input and single output with single hidden layers with 50 hidden units.

Dataset

The dataset consists of 23 columns and some of the columns contain categorical values, which we have to convert into the dummy variable using a one-hot encoding approach. No. of observations in the dataset are 263. The ratio of splitting dataset into training, validation and test are 80%, 10% and 10% respectively.

Neural Network Design



Code

Code for Importing Libraries

```
# Import Libraries
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split, TensorDataset
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchinfo
from torchmetrics import R2Score,MeanAbsoluteError
from ISLP import load_data
from sklearn.model_selection import train_test_split
```

Selecting NIVIDA-CUDA Device to train on GPU

```
# Select Device for training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Selected Device for Training: ', device)
```

Import Dataset for training model, we also standardize the dataset which will help NN to train better.

```
# Import Data
data=load_data('Hitters').dropna()
# Extract columns of category
catCols=data.select_dtypes(include='category').columns
# One-hot Encoding for Category data
data=pd.get_dummies(data,catCols)
# Compute the mean and standard deviation for each column
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
# Standardize the data
standardized_data = (data - means) / stds
# Split Response and Predictor variable
Xdata=standardized_data[standardized_data.drop(columns='Salary').columns].to_numpy
    ()
Ydata=standardized_data['Salary'].to_numpy()
```

Now, Convert the dataset to the Tensor Dataloader which will be required for training the neural network model, Dataloader used to increase the performance of pre-processing of data if any and which pass the data in batch without sacrificing the performance of machine.

```
# Train, validate and test splits
Xtrain, Xtemp, Ytrain, Ytemp = train_test_split(Xdata, Ydata, test_size=0.2)
Xvalid, Xtest, Yvalid, Ytest = train_test_split(Xtemp, Ytemp, test_size=0.1/0.2)
# Create TensorDatasets
train_dataset = TensorDataset(torch.tensor(Xtrain, dtype=torch.float32), torch.
    tensor(Ytrain, dtype=torch.float32))
valid_dataset = TensorDataset(torch.tensor(Xvalid, dtype=torch.float32), torch.
    tensor(Yvalid, dtype=torch.float32))
test_dataset = TensorDataset(torch.tensor(Xtest, dtype=torch.float32), torch.
    tensor(Ytest, dtype=torch.float32))
# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Define Neural Network containing single hidden layer of 50 units

```
# NN Model
class Model(nn.Module):
    def __init__(self):
        super(Model,self).__init__()
```

```
        self.flatten=nn.Flatten()
        self.sequential=nn.Sequential(
            nn.Linear(22,50),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(50,1)
        )
    def forward(self,x):
        x=self.flatten(x)
        return torch.flatten(self.sequential(x))
# Transfer Model to CUDA Device
model = Model().to(device)
```

Set the Hyperparameters and performance metrics

```
# Hyperparameters and Performance Metrics
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(),lr=0.001)
r2_metric = R2Score().to(device)
mae_metric = MeanAbsoluteError().to(device)
Nepochs = 100
```

The training and validation section is shown below, it will plot the Loss for both on figure and update the figure on each epochs to see the progress of training.

```
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,1)
ax.set_xlabel('Epochs')
ax.set_ylabel('MSE')
ax.set_title('Training and Validation Loss')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Training the Model
trainLoss,validLoss=[],[]
for epoch in range(Nepochs):
    model.train()
    running_loss = 0
    for input,target in train_loader:
        optimizer.zero_grad()
        # Add inputs and labels to device
        input, target = input.to(device), target.to(device)
        # Forward pass
        predictions = model(input)
        # Compute loss
        loss = criterion(predictions,target)
        # Backward pass
        loss.backward()
        optimizer.step()
        # Compute loss
        running_loss += loss.item()
    trainLoss.append(running_loss/len(train_loader))
    print(f'Epoch {epoch+1}/{Nepochs}, Loss: {running_loss/len(train_loader):.4f}')

    model.eval()
    running_loss = 0
    with torch.no_grad():
        for input,target in valid_loader:
```

```
                # Add inputs and labels to device
                input, target = input.to(device), target.to(device)
                predictions = model(input)
                # Compute loss
                loss = criterion(predictions, target)
                # Compute loss
                running_loss += loss.item()
            validLoss.append(running_loss/len(valid_loader))
    # Update Plot
    train_acc_line.set_data(range(1, len(trainLoss) + 1), trainLoss)
    valid_acc_line.set_data(range(1, len(trainLoss) + 1), validLoss)
    ax.relim()
    plt.draw()
    plt.pause(0.1)
plt.show()
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset.

```
# Test Model
model.eval()
testLoss=[]
running_loss = 0
with torch.no_grad():
        for input,target in test_loader:
                # Add inputs and labels to device
                input, target = input.to(device), target.to(device)
                predictions = model(input)
                # Compute loss
                loss = criterion(predictions, target)
                r2_metric.update(predictions, target)
                mae_metric.update(predictions, target)
                # Compute loss
                running_loss += loss.item()
        testLoss=running_loss/len(test_loader)
print('Testing Results')
print(f'R-Squared: {r2_metric.compute():0.4f}')
print(f'MSE: {testLoss:0.4f}')
print(f'MAE: {mae_metric.compute():0.4f}')
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 100

Criterion : MSE Loss

Batch Size : 32

Performance Metric

R-Squared : 0.7640

MSE : 0.3231

MAE : 0.4075

Conclusions

From the above results we can see that the model fit is good as the R-square values is close to 1 but the R-square value can be better, it is good if we have large amount of data to train model and get good prediction on salary. Relatively high value of R-square suggesting that the model fits the data well. It indicates that the neural network has successfully captured a substantial amount of the underlying patterns related to the performance of the hitters while predicting the Salary.

Multilayer Network on the MNIST Digit Data

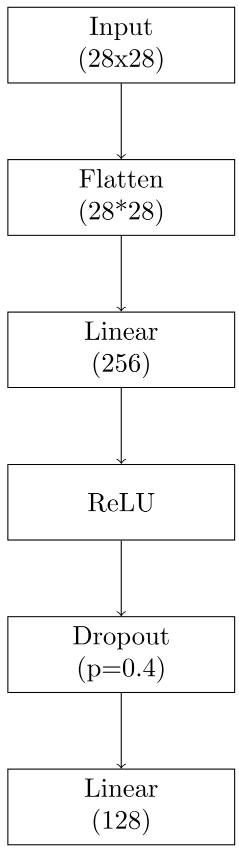
Introduction

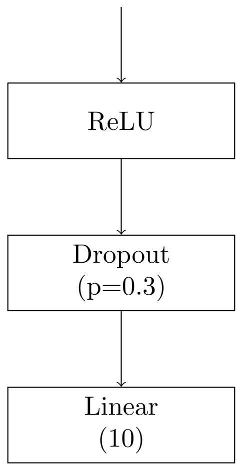
In this section we will create the multilayer network which will be trained on the MNIST digit dataset from 0 to 9. The image size of 28x28 is the input of the model and 10 output represent the probability of occurring that response number label. The model is trained on the GPU for faster computation and with batchsize of 128.

Dataset

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems. The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. The train, valid and test set is splited into 80%, 10% and 10% respectively.

Neural Network Design





Code

Import Libraries

```
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split, TensorDataset
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchinfo
from torchmetrics import R2Score,MeanAbsoluteError
from ISLP import load_data
from sklearn.model_selection import train_test_split
```

Selecting NIVIDA-CUDA Device to train on GPU

```
# Select Device for training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Selected Device for Training: ', device)
```

Augment Data and Add to Dataloader

```
 Data Augmentation Transform
transform = transforms.Compose([
    transforms.ToTensor()
])

# Load MNIST training and test datasets
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
train_dataset,valid_dataset=random_split(train_dataset,[int(0.8*len(train_dataset)),int(0.2*len(train_dataset))])
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
# Data loaders for batching
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=128, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False)
```

Create Multilayer Neural Network to train MNIST Dataset

```python
# NN Model
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.layer1 = nn.Sequential(
        nn.Flatten(),
        nn.Linear(28*28, 256),
        nn.ReLU(),
        nn.Dropout(0.4))
        self.layer2 = nn.Sequential(
        nn.Linear(256, 128),
        nn.ReLU(),
        nn.Dropout(0.3))
        self._forward = nn.Sequential(
        self.layer1,
        self.layer2,
        nn.Linear(128, 10))
    def forward(self, x):
        return self._forward(x)

# Transfer Model to CUDA Device
model = Model().to(device)
torchinfo.summary(model)
```

Set the Hyperparameters and performance metrics

```python
# Hyperparameters and Performance Metrics
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),lr=0.001)
Nepochs = 10
```

The training and validation section is shown below, it will plot the Loss for both on figure and update the figure on each epochs to see the progress of training.

```python
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Training and Validation Accuracy')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Training the Model
trainAcc,validAcc=[],[]
for epoch in range(Nepochs):
    model.train()
    correct_train = 0
    total_train = 0
    running_loss = 0
    for input,target in train_loader:
        optimizer.zero_grad()
        # Add inputs and labels to device
        input, target = input.to(device), target.to(device)
        # Forward pass
        predictions = model(input)
        # Compute loss
        loss = criterion(predictions,target)
```

```
        # Backward pass
        loss.backward()
        optimizer.step()
        # Compute loss
        running_loss += loss.item()
        # Compute Accuracy
        _, predicted = torch.max(predictions.data, 1)
        total_train += target.size(0)
        correct_train += (predicted == target).sum().item()
    # Store Training Accuracy
    trainAcc.append(correct_train / total_train * 100)
    print(f'Epoch {epoch+1}/{Nepochs}, Loss: {running_loss/len(train_loader):.4f}'
    )

    model.eval()
    running_loss = 0
    correct_valid = 0
    total_valid = 0
    with torch.no_grad():
        for input,target in valid_loader:
            # Add inputs and labels to device
            input, target = input.to(device), target.to(device)
            predictions = model(input)
            # Compute loss
            loss = criterion(predictions, target)
            # Compute loss
            running_loss += loss.item()
            # Compute Accuracy
            _, predicted = torch.max(predictions.data, 1)
            total_valid += target.size(0)
            correct_valid += (predicted == target).sum().item()
        validAcc.append(correct_valid / total_valid * 100)
    # Update Plot
    train_acc_line.set_data(range(1, len(trainAcc) + 1), trainAcc)
    valid_acc_line.set_data(range(1, len(trainAcc) + 1), validAcc)
    ax.relim()
    plt.draw()
    plt.pause(0.1)
plt.show()
print('Training Accuracy: ',trainAcc[-1])
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset.

```
# Test Model
model.eval()
testLoss=[]
correct_test = 0
total_test = 0
running_loss = 0
with torch.no_grad():
        for input,target in test_loader:
            # Add inputs and labels to device
            input, target = input.to(device), target.to(device)
            predictions = model(input)
            # Compute loss
            loss = criterion(predictions, target)
            # Compute loss
            running_loss += loss.item()
            total_test += target.size(0)
            _, predicted = torch.max(predictions.data, 1)
            correct_test += (predicted == target).sum().item()
```

```
        testLoss=running_loss/len(test_loader)
print('Testing Accuracy: ',correct_test / total_test * 100)
```

We will also show the random image of data and plot the actual and predicted labels as shown below.

```
# Move images back to CPU for plotting
images = input.cpu().numpy()
# Plot the images and their predictions
fig = plt.figure(figsize=(12, 12))
for i in range(9):
    plt.subplot(3, 3, i+1)
    plt.imshow(np.squeeze(images[i]), cmap='gray')
    plt.title(f"Pred: {predicted[i].item()}, Actual: {target[i].item()}")
    plt.axis('off')
plt.show()
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 10

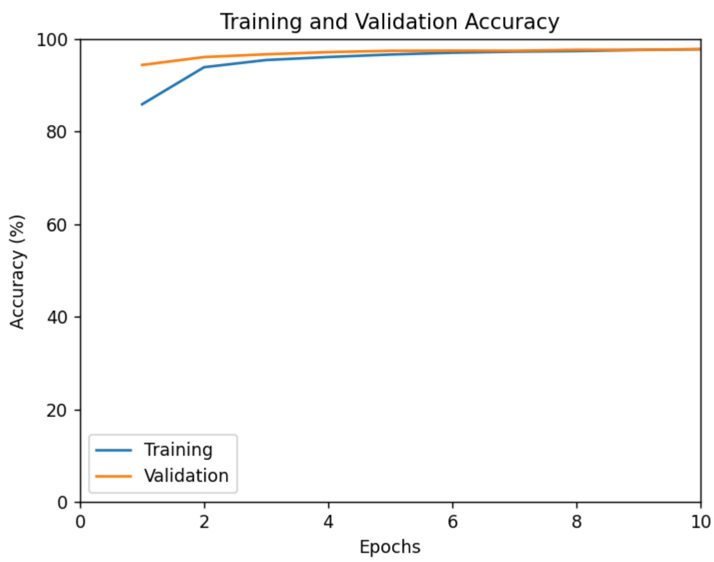
Criterion : Cross Entropy Loss

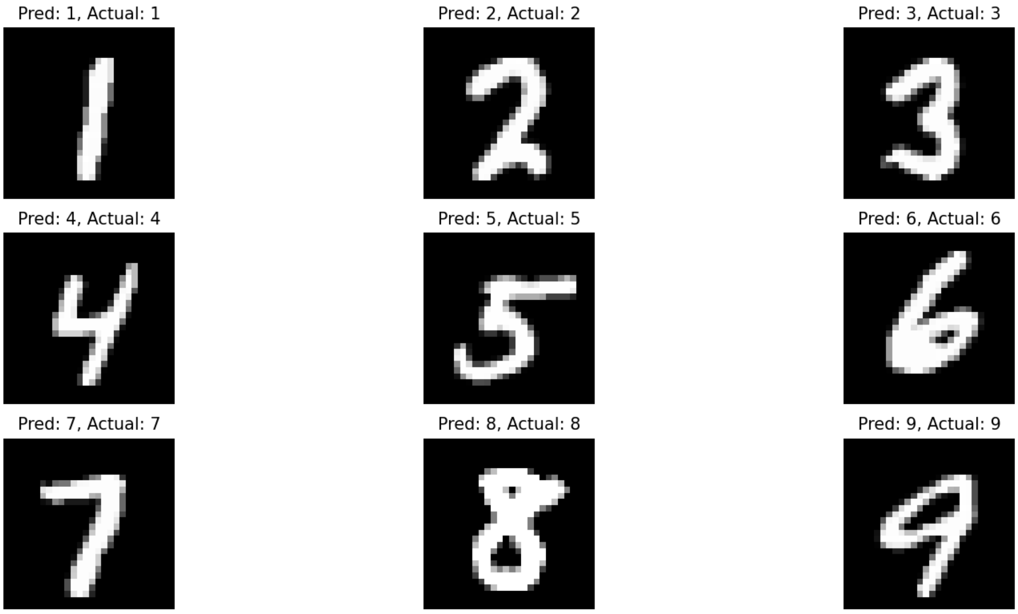
Batch Size : 128

Performance Metric

Training Accuracy : 97.80

Testing Accuracy : 97.86





Conclusions

From the above results, we can say that the model is performing very well in predicting the number from the given image, the accuracy of the model is 97% in just 10 epochs which gives us good results. The testing accuracy of the model is also promising which is 97.86%. So, we can concluded that the designed NN model gives the good results for the given MNIST dataset.

Convolutional Neural Networks

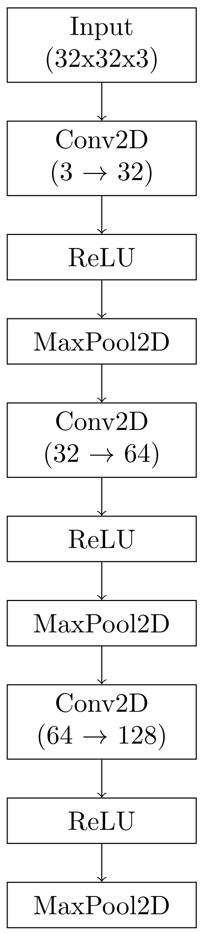
Introduction

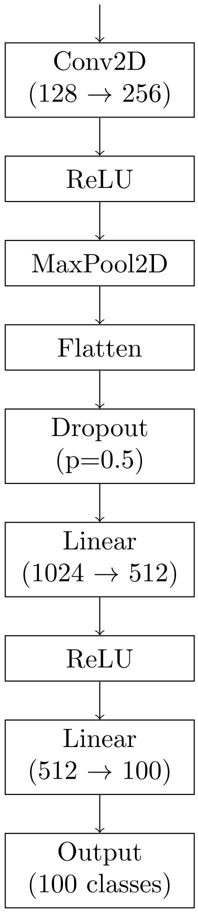
In this part we will design the CNN Model to classify the collection of the natural image from the dataset, we will use CIFAR100 dataset which contains 100 class of similar kind of image, we will design CNN model which takes the input image and at output we get the predicted class.

Dataset

The dataset consists of 50,000 training images and 10,000 testing image. out of the training we have use 10,000 for validating the model. The input image is first augmented using the transform by giving the random horizontal and vertical flips after that the Dataloader will handle the image with batchsize of 128.

Neural Network Design





Code

Import Libraries

```python
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchinfo
```

Selecting NIVIDA-CUDA Device to train on GPU

```python
# Select Device for training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Selected Device for Training: ',device)
```

Data Augmentation and Create Dataloader

```python
# Data Augmentation Transform
```

```
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor()
])
# Import Dataset
train_dataset = torchvision.datasets.CIFAR100(root='./data', train=True, download=True, transform=transform)
train_dataset,valid_dataset=random_split(train_dataset,[int(0.8*len(train_dataset)),int(0.2*len(train_dataset))])
test_dataset = torchvision.datasets.CIFAR100(root='./data', train=False, download=True, transform=transform)
# Create DataLoader
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=False)
```

Define Neural Network containing the CNN Model

```
# Create NN Model
class BuildingBlock(nn.Module):
    def __init__(self,in_channels,out_channels):
        super(BuildingBlock , self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels,out_channels=out_channels , kernel_size =(3 ,3),padding='same')
        self.activation = nn.ReLU ()
        self.pool = nn.MaxPool2d(kernel_size =(2 ,2))
    def forward(self , x):
        return self.pool(self.activation(self.conv(x)))

class CIFARModel(nn.Module):
    def __init__(self):
        super(CIFARModel , self).__init__()
        sizes = [(3 ,32) ,
        (32 ,64) ,
        (64 ,128) ,
        (128 ,256)]
        self.conv = nn.Sequential (*[ BuildingBlock(in_ , out_) for in_ , out_ in sizes ])
        self.output = nn.Sequential(nn.Dropout (0.5) ,
        nn.Linear (2*2*256 , 512) ,
        nn.ReLU (),
        nn.Linear (512, 100))
    def forward(self , x):
        val = self.conv(x)
        val = torch.flatten(val , start_dim =1)
        return self.output(val)
# Model Summary
model = CIFARModel().to(device)
torchinfo.summary(model,input_size=(128,3,32,32),col_names=['input_size','output_size','num_params'])
```

Set the Hyperparameters and performance metrics

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
Nepochs=50
```

The training and validation section is shown below, it will plot the Accuracy for both on figure and update the figure on each epochs to see the progress of training.

```python
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Training and Validation Accuracy')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Train Model
trainAcc,validAcc=[],[]
for epoch in range(Nepochs):
        model.train()
        correct_train = 0
        total_train = 0
        running_loss = 0
        for i, (inputs, labels) in enumerate(train_loader):
            # Add inputs and labels to device
            inputs, labels = inputs.to(device), labels.to(device)
            # Zero the parameter gradients
            optimizer.zero_grad()
            # Forward pass
            outputs = model(inputs)
            # Compute Loss
            loss = criterion(outputs, labels)
            # Backward pass
            loss.backward()
            optimizer.step()
            # Compute Accuracy
            _, predicted = torch.max(outputs.data, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

            # Compute Batch Loss
            running_loss += loss.item()
            # Print every 100 mini-batches
            if i % 100 == 99:
                print(f'Epoch [{epoch+1}/{Nepochs}], Step [{i+1}/{len(train_loader
    )}], Loss: {running_loss/100:.4f}')
                running_loss = 0
        # Store Training Accuracy
        trainAcc.append(correct_train / total_train * 100)

        # Validate Model
        model.eval()
        correct_valid = 0
        total_valid = 0
        with torch.no_grad():
            for inputs, labels in valid_loader:
                # Add inputs and labels to device
                inputs, labels = inputs.to(device), labels.to(device)
                # Forward pass
                outputs = model(inputs)
                # Compute Accuracy
                _, predicted = torch.max(outputs.data, 1)
                total_valid+= labels.size(0)
                correct_valid += (predicted == labels).sum().item()
            # Store Validation Accuracy
            validAcc.append(correct_valid / total_valid * 100)
```

```python

        # Update Plot
        train_acc_line.set_data(range(1,len(trainAcc)+1),trainAcc)
        valid_acc_line.set_data(range(1, len(trainAcc) + 1),validAcc)
        plt.draw()
        plt.pause(0.1)
plt.show()
print('Training Accuracy: ',trainAcc[-1])
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset.

```python
# Test Model
model.eval()
correct_test = 0
total_test = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        # Add inputs and labels to device
        inputs, labels = inputs.to(device), labels.to(device)
        # Forward pass
        outputs = model(inputs)
        # Compute Accuracy
        _, predicted = torch.max(outputs.data, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()
print('Testing Accuracy: ',correct_test / total_test * 100)
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 50

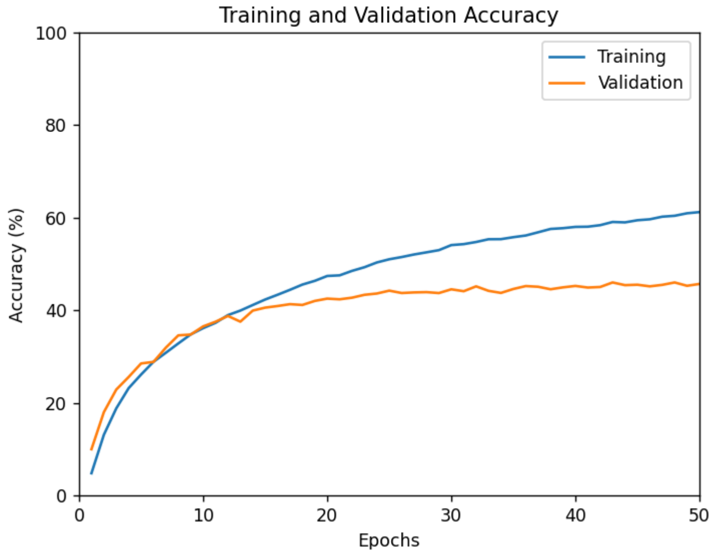
Criterion : Cross Entropy Loss

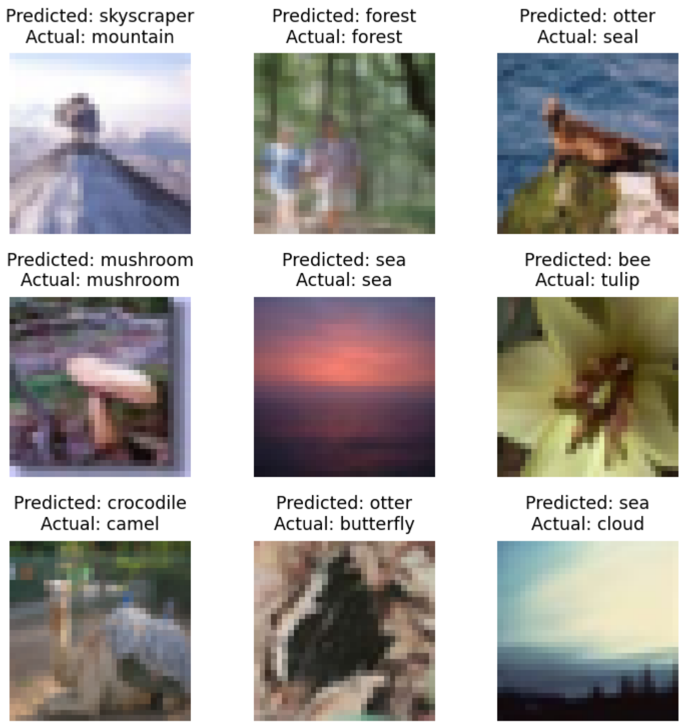
Batch Size : 128

Performance Metric

Training Accuracy : 61.21

Testing Accuracy : 45.68





Conclusions

We concluded that using the developed CNN Model we got a decent accuracy of 67.21% but the testing accuracy is not good enough to use the model, the reason for the poor performance may be due to lesser data size for each classes in dataset or we can apply some more dense layer to get the good results. From the tested results we can see that some images get easily able to predict but it is also tricky to predict from some classes where the feature is very similar.

Using Pretrained CNN Models

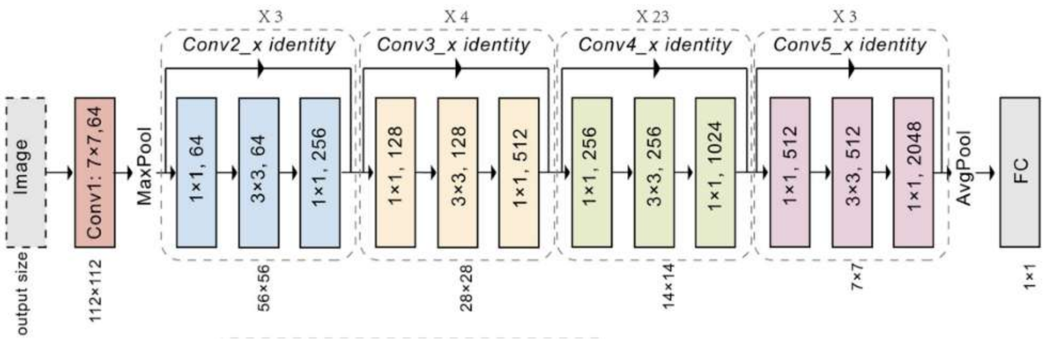
Introduction

In this task we are going to use the pre-trained model ResNet-50 to predict the class of the given image. we import the model with trained weights for the imagenet dataset. using the few images we will test the model and check how the ResNet-50 Model performs.

Dataset

Here we are not going to train the model but as far the dataset is concerned, ResNet-50 model is trained on the ImageNet dataset consisting of 1000 classes and 14 Million images. We are going to use few image to test the Model response.

Neural Network Design



Code

Import Libraries

```
# Import Libraries
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split, Dataset
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torchinfo
from PIL import Image
```

Selecting NIVIDA-CUDA Device to train on GPU

```
# Select Device for training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Selected Device for Training: ',device)
```

Data Augmentation and DataLoader

```python
# Data Augmentation Transform
transform = transforms.Compose([
    transforms.Resize((232,232)),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    # transforms.Normalize([0.485 ,0.456 ,0.406] ,[0.229 ,0.224 ,0.225])
])
# Image Dataset Loader
class ImageDataset(Dataset):
    def __init__(self, image_folder, transform=None):
        self.image_folder = image_folder
        self.image_files = [f for f in os.listdir(image_folder) if f.endswith(('jpg','png'))]
        self.transform = transform

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_folder, self.image_files[idx])
        image = Image.open(img_path)  # Open image
        if self.transform:
            image = self.transform(image)  # Apply transformation
        return image

# Import Dataset
test_dataset = ImageDataset('./data/book_images',transform=transform)
# Create DataLoader
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1, shuffle=False)
```

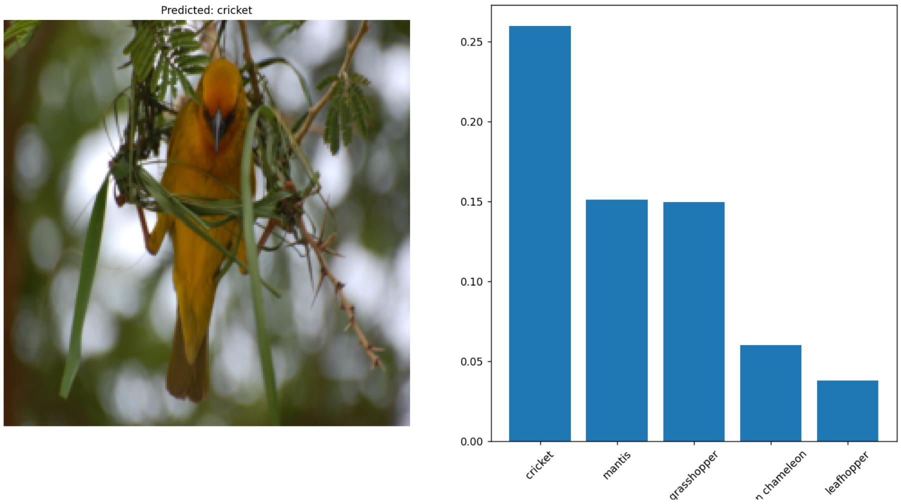
ResNet-50 Model

```python
# Resnet50 Model
model=torchvision.models.resnet50(pretrained=True).to(device)
# Extract Categories of ImageNet
category=torchvision.models.ResNet50_Weights.DEFAULT.meta['categories']
```

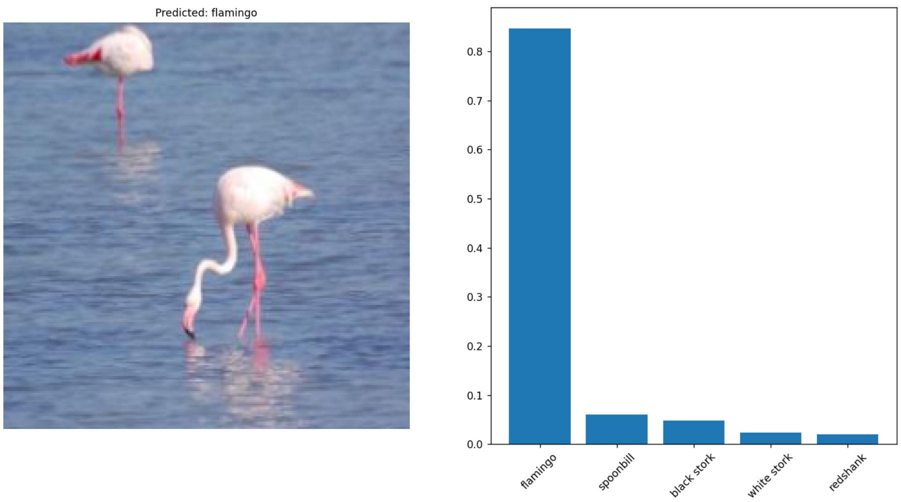
Evaluate model

```python
# Test Model
model.eval()
for i,inputs in enumerate(test_loader):
    # Add inputs and labels to device
    inputs = inputs.to(device)
    # Forward pass
    outputs = model(inputs)
    prediction = outputs.squeeze(0).softmax(0)
    classid = prediction.argmax().item()
    predicted_label = category[classid]
    # Find 5 strongest prediction
    nclassid=prediction.argsort(descending=True)[0:5].cpu().numpy()
    npredict_label= [category[i] for i in nclassid]
    npred=[prediction[i].item() for i in nclassid]
    img = torchvision.utils.make_grid(inputs.cpu()).numpy()
    fig, axes = plt.subplots(1, 2, figsize=(12,8))
    axes[0].imshow(np.transpose(img, (1, 2, 0)))
    axes[0].set_title(f'Predicted: {predicted_label}', fontsize=10)
    axes[0].axis('off')
    axes[1].bar(npredict_label,npred)
    axes[1].tick_params(axis='x',rotation = 45)
    plt.show()
```

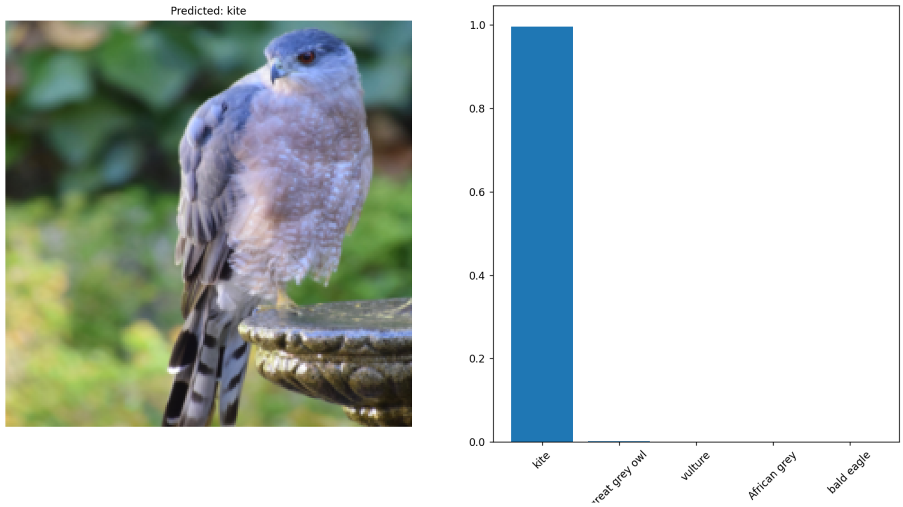
Results



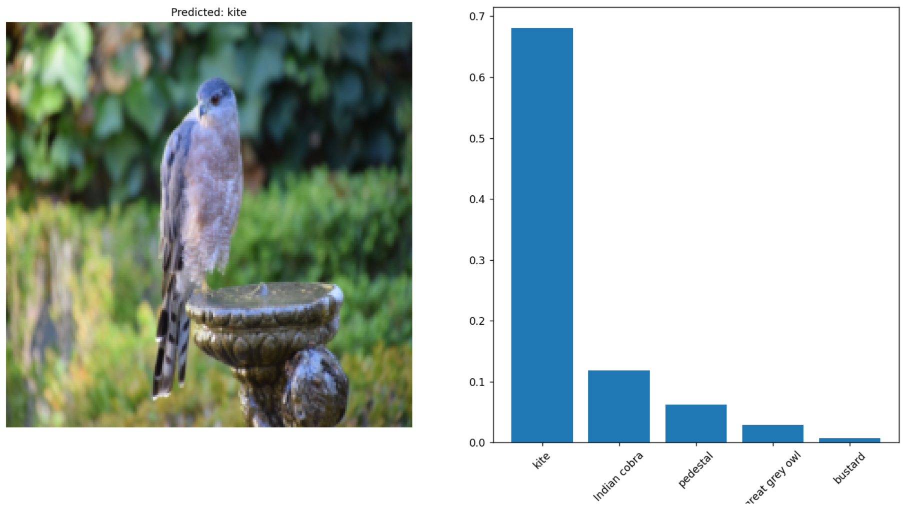
Predicted: cricket
0.25
0.20
0.15
0.10
0.05
0.00
cricket
mantis
grasshopper
chameleon
leafhopper



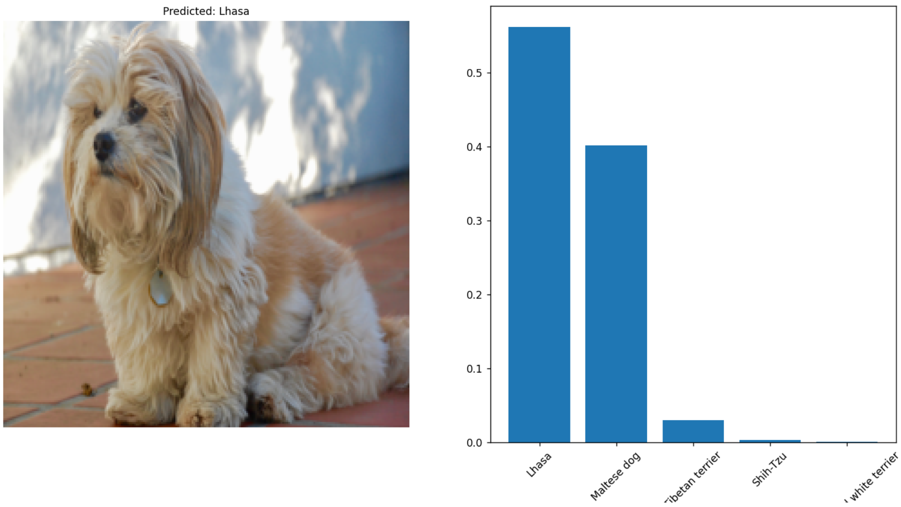
Predicted: flamingo
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
flamingo
spoonbill
black stork
white stork
redshank

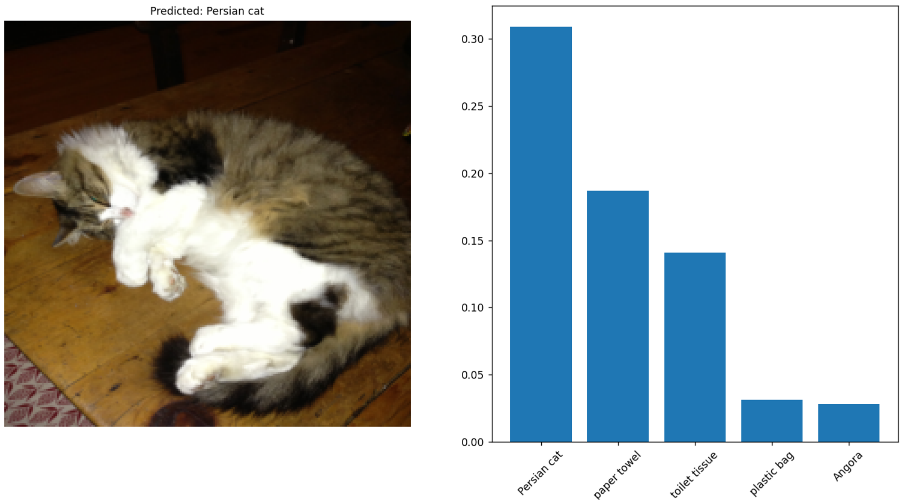


Predicted: kite
1.0
0.8
0.6
0.4
0.2
0.0
kite
reat grey owl
vulture
African grey
bald eagle



Predicted: kite
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
kite
Indian cobra
pedestal
reat grey owl
bustard





Conclusions

We have predict the class of the image using the ResNet-50 pre-trained network, we can see that it model is able to works well for the given inputs which is used for testing. in the first image instead of cape weaver it is predicted as cricket, flamingo image is predicted correctly, In 3rd and 4th image it predicts the wrong which should be Cooper's hawk. So, the ResNet-50 done the reasonable job for classifying the given image.

IMDB Document Classification

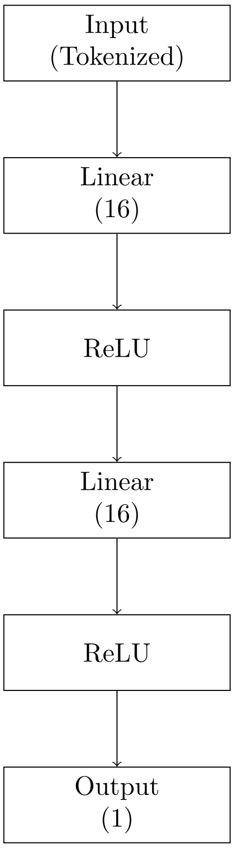
Introduction

In this we will create the NN model which can predict the sentiment based on the given text review of movie. for this task we are going to use IMDb dataset which has review of movie with sentiment of review. In this we use tokenization of word and Vocabulary library to embed the text data for training the model.

Dataset

We have a training set and test set, each with 25,000 examples, and each balanced with regard to sentiment.The resulting training feature matrix X has dimension 25,000 10,000, but only 1.3% of the binary entries are non zero. We split off a validation set in ratio of 10% from the 25,000 training observations.

Neural Network Design



Code

Import Libraries

```
# Import Libraries
```

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchtext import data, datasets  # Use legacy for compatibility
import torchinfo
import spacy
```

Selecting NIVIDA-CUDA Device to train on GPU

```python
# Select Device for training
device = torch.device('cpu')
print('Selected Device for Training: ', device)
```

Tokenizing Words

```python
# Load the SpaCy English tokenizer
spacy_en = spacy.load('en_core_web_sm')
# Define a function to tokenize the text using SpaCy
def tokenize_en(text):
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

Create DataLoader for Text Dataset

```python
# Define fields for text and labels
TEXT = data.Field(tokenize=tokenize_en, lower=True)
LABEL = data.LabelField(dtype=torch.float)
# Load IMDb dataset
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
train_data, valid_data = train_data.split(split_ratio=0.8)
# Build the vocabulary for the text and labels
TEXT.build_vocab(train_data, max_size=10000, vectors="glove.6B.100d", unk_init=
    torch.Tensor.normal_)
LABEL.build_vocab(train_data)
# Create an iterator for training data
train_iterator,valid_iterator, test_iterator = data.BucketIterator.splits((
    train_data,valid_data,test_data), batch_size=64, device=device)
```

Define Neural Network

```python
# NN Model
class IMDBModel(nn.Module):
    def __init__(self, input_size):
        super(IMDBModel, self).__init__()
        self.dense1 = nn.Linear(input_size, 16)
        self.activation = nn.ReLU()
        self.dense2 = nn.Linear(16, 16)
        self.output = nn.Linear(16, 1)

    def forward(self, x):
        val = x.mean(dim=0)
        for _map in [self.dense1, self.activation, self.dense2, self.activation,
    self.output]:
            val = _map(val)
        return torch.flatten(val)

model = IMDBModel(100)  # Input size of 100 (as we are using GloVe 100d vectors)
model = model.to(device)
```

Set the Hyperparameters and performance metrics

```
# Hyperparameters and Performance Metrics
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(),lr=0.001)
Nepochs = 50
```

The training and validation section is shown below, it will plot the Accuracy for both on figure and update the figure on each epochs to see the progress of training.

```
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,100)
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Training and Validation Accuracy')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Training the Model
trainAcc,validAcc=[],[]
for epoch in range(Nepochs):
    model.train()
    correct_train = 0
    total_train = 0
    running_loss = 0
    for batch in train_iterator:
        optimizer.zero_grad()
        # Get inputs and labels
        text = batch.text
        text = text.to(device)
        labels = batch.label.to(device)
        # Get embeddings from GloVe vectors
        embedded = TEXT.vocab.vectors[text]
        embedded = embedded.to(device)

        # Forward pass
        predictions = model(embedded)

        # Compute loss
        loss = criterion(predictions, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        # Compute loss
        running_loss += loss.item()

        # Compute accuracy
        sigmoid_predictions = torch.sigmoid(predictions)
        binary_predictions = (sigmoid_predictions >= 0.5).float()

        correct_train += (binary_predictions == labels).sum().item()
        total_train += labels.size(0)
    trainAcc.append(correct_train / total_train*100)
    print(f'Epoch {epoch+1}/{Nepochs}, Loss: {running_loss:.4f}')

    model.eval()
    correct_valid = 0
```

```
    total_valid = 0
    with torch.no_grad():
        for batch in valid_iterator:
            text = batch.text
            text = text.to(device)
            labels = batch.label.to(device)

            embedded = TEXT.vocab.vectors[text]
            embedded = embedded.to(device)

            predictions = model(embedded)

            # Compute accuracy
            sigmoid_predictions = torch.sigmoid(predictions)
            binary_predictions = (sigmoid_predictions >= 0.5).float()
            correct_valid += (binary_predictions == labels).sum().item()
            total_valid += labels.size(0)
        validAcc.append(correct_valid / total_valid * 100)

    # Update Plot
    train_acc_line.set_data(range(1, len(trainAcc) + 1), trainAcc)
    valid_acc_line.set_data(range(1, len(trainAcc) + 1), validAcc)
    plt.draw()
    plt.pause(0.1)
plt.show()
print('Training Accuracy: ',trainAcc[-1])
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset.

```
# Test Model
model.eval()
correct_test = 0
total_test = 0
with torch.no_grad():
        for batch in test_iterator:
            text = batch.text
            text = text.to(device)
            labels = batch.label.to(device)

            embedded = TEXT.vocab.vectors[text]
            embedded = embedded.to(device)

            predictions = model(embedded)

            # Compute accuracy
            sigmoid_predictions = torch.sigmoid(predictions)
            binary_predictions = (sigmoid_predictions >= 0.5).float()
            correct_test += (binary_predictions == labels).sum().item()
            total_test += labels.size(0)
print('Testing Accuracy: ',correct_test / total_test * 100)
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 50

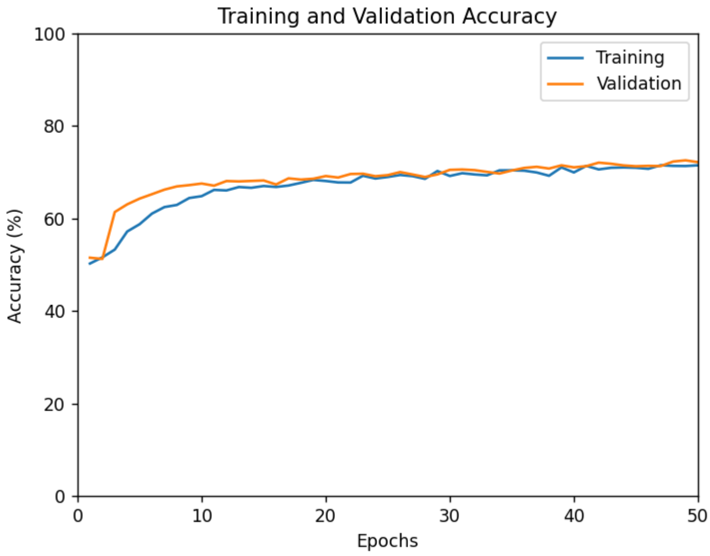
Criterion : BCE With Logits Loss

Batch Size : 64

Performance Metric

Training Accuracy : 71.505

Testing Accuracy : 72.48



Conclusions

From the above results, we can say that model is working sufficiently well to predict that the review of the movie is positive or negative, the training accuracy is 71% and testing accuracy is 72% for just 50 epochs which is good enough for the prediction, more number of data can be incorporated to get the better prediction performance.

Recurrent Neural Networks

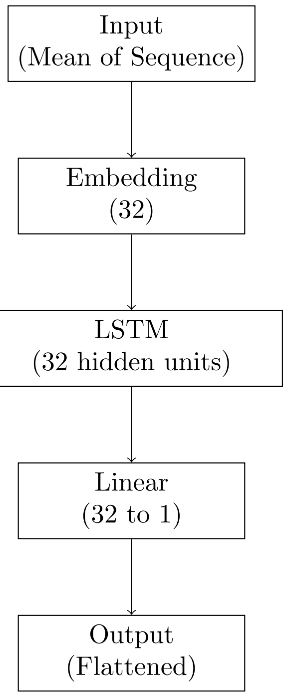
Introduction

In this section we will use the same IMDb dataset and train it using LSTM network and we will compare the performance of both the NN model on same dataset. The last model give 71% accuracy for 50 Epochs.

Dataset

We have a training set and test set, each with 25,000 examples, and each balanced with regard to sentiment.The resulting training feature matrix X has dimension 25,000 10,000, but only 1.3% of the binary entries are non zero. We split off a validation set in ratio of 10% from the 25,000 training observations.

Neural Network Design



Code

Import Libraries

```python
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split
import torch.nn as nn
import torch.optim as optim
```

```
import torchvision
import torchvision.transforms as transforms
from torchtext import data, datasets
import torchinfo
import spacy
```

Selecting NIVIDA-CUDA Device to train on GPU

```
# Select Device for training
device = torch.device('cpu')
print('Selected Device for Training: ', device)
```

Tokenizing Words and create vocabulary

```
# Load the SpaCy English tokenizer
spacy_en = spacy.load('en_core_web_sm')
# Define a function to tokenize the text using SpaCy
def tokenize_en(text):
    return [tok.text for tok in spacy_en.tokenizer(text)]
# Define fields for text and labels
TEXT = data.Field(tokenize=tokenize_en, lower=True)
LABEL = data.LabelField(dtype=torch.float)
# Load IMDb dataset
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
train_data, valid_data = train_data.split(split_ratio=0.8)
# Build the vocabulary for the text and labels
TEXT.build_vocab(train_data, max_size=10000, vectors="glove.6B.100d", unk_init=
    torch.Tensor.normal_)
LABEL.build_vocab(train_data)
# Create an iterator for training data
train_iterator,valid_iterator, test_iterator = data.BucketIterator.splits((
    train_data,valid_data,test_data), batch_size=64, device=device)
```

Create NN Model using LSTM Architecture

```
# NN Model
class LSTMModel(nn.Module):
    def __init__(self, input_size):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(input_size, 32)
        self.lstm = nn.LSTM(input_size=100,
                        hidden_size=32,
                        batch_first=True)
        self.dense = nn.Linear(32, 1)
    def forward(self, x):
        val, (h_n, c_n) = self.lstm(x.mean(dim=0))
        return torch.flatten(self.dense(val))
model = LSTMModel(100)  # Input size of 100 (as we are using GloVe 100d vectors)
model = model.to(device)
```

Set the Hyperparameters and performance metrics

```
# Hyperparameter and Performance Metrics
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(),lr=0.001)
Nepochs = 50
```

The training and validation section is shown below, it will plot the Accuracy for both on figure and update the figure on each epochs to see the progress of training.

```
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,100)
```

```
ax.set_xlabel('Epochs')
ax.set_ylabel('Accuracy (%)')
ax.set_title('Training and Validation Accuracy')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Training the Model
trainAcc,validAcc=[],[]
for epoch in range(Nepochs):
    model.train()
    correct_train = 0
    total_train = 0
    running_loss = 0
    for batch in train_iterator:
        optimizer.zero_grad()
        # Get inputs and labels
        text = batch.text
        text = text.to(device)
        labels = batch.label.to(device)
        # Get embeddings from GloVe vectors
        embedded = TEXT.vocab.vectors[text]
        embedded = embedded.to(device)

        # Forward pass
        predictions = model(embedded)

        # Compute loss
        loss = criterion(predictions, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        # Compute loss
        running_loss += loss.item()

        # Compute accuracy
        sigmoid_predictions = torch.sigmoid(predictions)
        binary_predictions = (sigmoid_predictions >= 0.5).float()

        correct_train += (binary_predictions == labels).sum().item()
        total_train += labels.size(0)
    trainAcc.append(correct_train / total_train*100)
    print(f'Epoch {epoch+1}/{Nepochs}, Loss: {running_loss:.4f}')

    model.eval()
    correct_valid = 0
    total_valid = 0
    with torch.no_grad():
        for batch in valid_iterator:
            text = batch.text
            text = text.to(device)
            labels = batch.label.to(device)

            embedded = TEXT.vocab.vectors[text]
            embedded = embedded.to(device)

            predictions = model(embedded)

            # Compute accuracy
```

```
            sigmoid_predictions = torch.sigmoid(predictions)
            binary_predictions = (sigmoid_predictions >= 0.5).float()
            correct_valid += (binary_predictions == labels).sum().item()
            total_valid += labels.size(0)
        validAcc.append(correct_valid / total_valid * 100)

    # Update Plot
    train_acc_line.set_data(range(1, len(trainAcc) + 1), trainAcc)
    valid_acc_line.set_data(range(1, len(trainAcc) + 1), validAcc)
    plt.draw()
    plt.pause(0.1)
plt.show()
print('Training Accuracy: ',trainAcc[-1])
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset.

```
# Test Model
model.eval()
correct_test = 0
total_test = 0
with torch.no_grad():
        for batch in test_iterator:
            text = batch.text
            text = text.to(device)
            labels = batch.label.to(device)

            embedded = TEXT.vocab.vectors[text]
            embedded = embedded.to(device)

            predictions = model(embedded)

            # Compute accuracy
            sigmoid_predictions = torch.sigmoid(predictions)
            binary_predictions = (sigmoid_predictions >= 0.5).float()
            correct_test += (binary_predictions == labels).sum().item()
            total_test += labels.size(0)
print('Testing Accuracy: ',correct_test / total_test * 100)
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 50

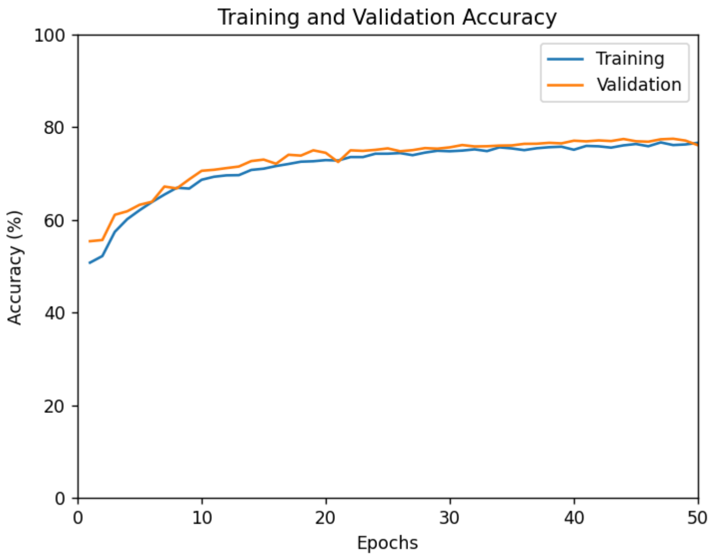
Criterion : BCE With Logits Loss

Batch Size : 64

Performance Metric

Training Accuracy : 76.645

Testing Accuracy : 76.896



Conclusions

We can observed that there is increase in performance of 5% as compare to the previous method which use embedding. We still have a somewhat "entry level" RNN in spite of this additional LSTM complexity. With a different model size, different regularization, and more hidden layers, we could probably get marginally better results. Nevertheless, parameter optimization and exploring various architectures are laborious due to the lengthy training times of LSTM models.

Time Series Prediction

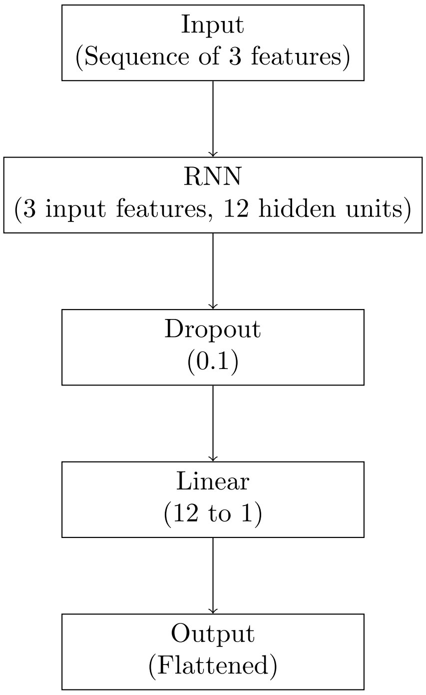
Introduction

Time series forecasting involves predicting future values of a given series based on its historical values. In this part we will takes the NYSE dataset with three feature and 5 lagged sequence and try to forecast the log volume of the NYSE stock.

Dataset

Historical trading statistics from the New York Stock Exchange. Daily values of the normalized log trading volume, DJIA return, and log volatility are given for a 24-year period from 1962–1986. The training, validation, and testing set are split in 80:10:10 ratio respectively.

Neural Network Design



Code

Import Libraries

```
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import torch
from torch.utils.data import DataLoader, random_split, TensorDataset
import torch.nn as nn
import torch.optim as optim
import torchvision
```

```python
import torchvision.transforms as transforms
from torchtext import data, datasets
import torchinfo
import spacy
from ISLP import load_data
from sklearn.model_selection import train_test_split
```

Selecting NIVIDA-CUDA Device to train on GPU

```python
# Select Device for training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Selected Device for Training: ', device)
```

Import and Standardize dataset

```python
# Import Dataset
NYSE = load_data('NYSE').dropna()
cols = ['DJ_return', 'log_volume', 'log_volatility']
data = NYSE[cols].values
# Compute the mean and standard deviation for each column
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
# Standardize the data
standardized_data = (data - means) / stds
```

Create lagged Feature and Splits the datasets

```python
# Function to create lagged feature
def create_rnn_input(data,num_lags=5):
    N = data.shape[0] - num_lags
    rnn_input = np.zeros((N, num_lags, data.shape[1]))
    next_volume = np.zeros(N)
    for i in range(N):
        rnn_input[i] = data[i:i + num_lags]
        next_volume[i] = data[i + num_lags, 1]
    return rnn_input,next_volume

# Train, valid and test splot
Xdata,Ydata = create_rnn_input(standardized_data,num_lags=5)
Xtrain, Xtemp, Ytrain, Ytemp = train_test_split(Xdata, Ydata, test_size=0.2)
Xvalid, Xtest, Yvalid, Ytest = train_test_split(Xtemp, Ytemp, test_size=0.1/0.2)
```

Create DataLoader

```python
# Create TensorDatasets
train_dataset = TensorDataset(torch.tensor(Xtrain, dtype=torch.float32), torch.
    tensor(Ytrain, dtype=torch.float32))
valid_dataset = TensorDataset(torch.tensor(Xvalid, dtype=torch.float32), torch.
    tensor(Yvalid, dtype=torch.float32))
test_dataset = TensorDataset(torch.tensor(Xtest, dtype=torch.float32), torch.
    tensor(Ytest, dtype=torch.float32))
# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Define NN Model using RNN

```python
# NN Model
class NYSEModel(nn.Module):
    def __init__(self):
        super(NYSEModel, self).__init__()
        self.rnn = nn.RNN(3,12,batch_first=True)
        self.dense = nn.Linear(12, 1)
        self.dropout = nn.Dropout(0.1)
```

```python
    def forward(self, x):
        val, h_n = self.rnn(x)
        val = self.dense(self.dropout(val[:,-1]))
        return torch.flatten(val)
# Transfer to CUDA Device
model = NYSEModel().to(device)
```

Set the Hyperparameters and performance metric

```python
# Hyperparameters and Performance Metrics
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(),lr=0.01)
Nepochs = 50
```

The training and validation section is shown below, it will plot the Accuracy for both on figure and update the figure on each epochs to see the progress of training.

```python
# open Figure for Accuracies
fig,ax=plt.subplots()
ax.set_xlim(0,Nepochs)
ax.set_ylim(0,1)
ax.set_xlabel('Epochs')
ax.set_ylabel('MSE')
ax.set_title('Training and Validation Loss')
train_acc_line, = ax.plot([], [], label='Training')
valid_acc_line, = ax.plot([], [], label='Validation')
plt.legend()
plt.draw()

# Training the Model
trainLoss,validLoss=[],[]
for epoch in range(Nepochs):
    model.train()
    running_loss = 0
    for input,target in train_loader:
        optimizer.zero_grad()
        # Add inputs and labels to device
        input, target = input.to(device), target.to(device)
        # Forward pass
        predictions = model(input)
        # Compute loss
        loss = criterion(predictions,target)
        # Backward pass
        loss.backward()
        optimizer.step()
        # Compute loss
        running_loss += loss.item()

    trainLoss.append(running_loss/len(train_loader))
    print(f'Epoch {epoch+1}/{Nepochs}, Loss: {running_loss/len(train_loader):.4f}')

    model.eval()
    running_loss = 0
    with torch.no_grad():
        for input,target in valid_loader:
            # Add inputs and labels to device
            input, target = input.to(device), target.to(device)
            predictions = model(input)
            # Compute loss
            loss = criterion(predictions, target)
            # Compute loss
            running_loss += loss.item()
```

```
        validLoss.append(running_loss/len(valid_loader))
    # Update Plot
    train_acc_line.set_data(range(1, len(trainLoss) + 1), trainLoss)
    valid_acc_line.set_data(range(1, len(trainLoss) + 1), validLoss)
    ax.relim()
    plt.draw()
    plt.pause(0.1)
plt.show()
```

Testing of the trained model is the essential part of the post-designing process, here we check how the model is performed on the test dataset

```
# Test Model
model.eval()
testLoss=[]
running_loss = 0
with torch.no_grad():
        for input,target in test_loader:
            # Add inputs and labels to device
            input, target = input.to(device), target.to(device)
            predictions = model(input)
            # Compute loss
            loss = criterion(predictions, target)
            r2_metric.update(predictions, target)
            mae_metric.update(predictions, target)
            # Compute loss
            running_loss += loss.item()
        testLoss=running_loss/len(test_loader)
print('Testing Results')
print(f'R-Squared: {r2_metric.compute():0.4f}')
print(f'MSE: {testLoss:0.4f}')
print(f'MAE: {mae_metric.compute():0.4f}')
```

We will also forecast the trained model to see the output of the future volume of the stock.

```
# Forecasting Model
plt.plot(range(0,len(Yvalid)),Yvalid*stds[1]+means[1])
plt.plot(range(len(Yvalid)+1,len(Yvalid)+len(predictions.cpu().numpy())),
    predictions.cpu().numpy()*stds[1]+means[1])
plt.xlabel('Time Step')
plt.ylabel('Log(Trading Volume)')
plt.legend(['Past Data','Forecasting'])
plt.show()
```

Hyperparameters and Performance Metrics

Hyperparameters

Optimizer : ADAM

Epochs : 50

Criterion : MSE Loss

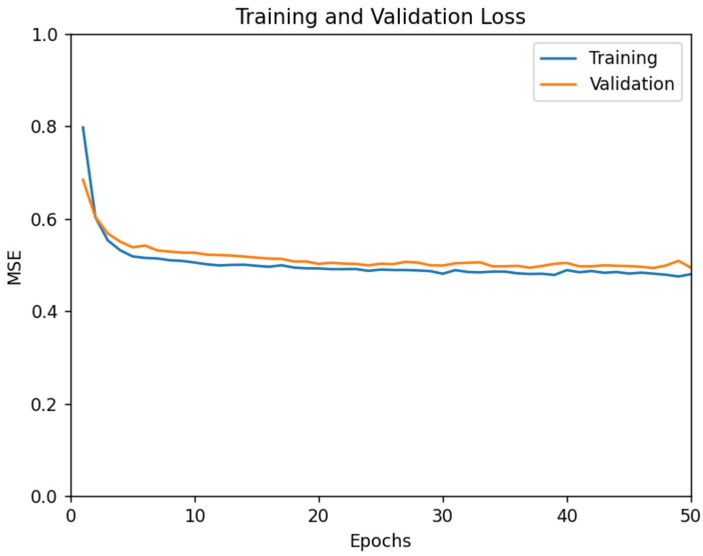
Batch Size : 64

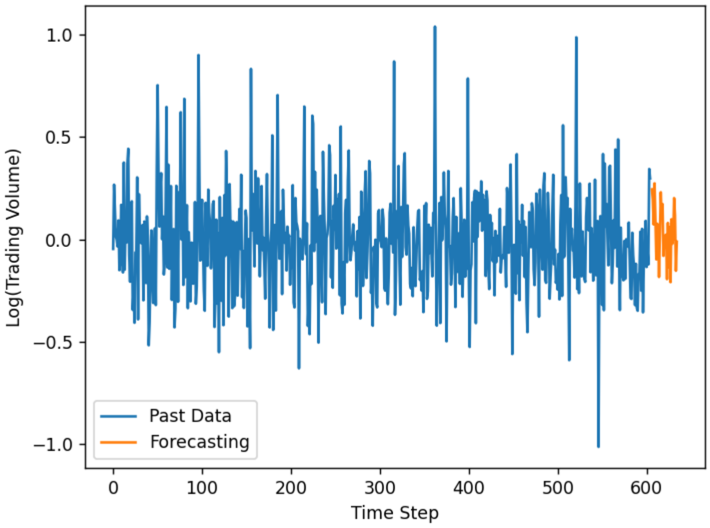
Performance Metric

R-Squared : 0.5298

MSE : 0.4838

MAE : 0.5257





Conclusions

We have fit the model and tried to get the forecasting based on the previous data. The R-squared values came to be 0.52 which is on the lower side, closer to value 1 treated as the good performance model and we have MSE of 0.4838.