Tasks

### Write a Python function using TensorFlow to create and compile a simple linear regression model with one dense layer. Train it on the classic California housing datasets.

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Input, Concatenate

from sklearn.datasets import fetch\_california\_housing

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

housing = fetch\_california\_housing()

Xtrain, Xtest, ytrain, ytest = train\_test\_split(housing.data, housing.target, test\_size = .2)

Xtrain, Xval, ytrain, yval = train\_test\_split(Xtrain, ytrain, test\_size=.2)

model = Sequential([

Dense(30, activation = 'relu', input\_shape= Xtrain.shape[1:]),

Dense(1)

])

model.compile(loss='mean\_squared\_error', optimizer='adam')

history = model.fit(Xtrain, ytrain, validation\_data=(Xval, yval), epochs=20)

### Write a Python script using PyTorch to train a convolutional neural network (CNN) on the Cifar 10 dataset. Include data loading, model definition, training loop, and evaluation

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision

import torchvision.transforms as transforms

import torch.nn.functional as F

transform = transforms.Compose(

[transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,

download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4,

shuffle=True, num\_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,

download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch\_size=4,

shuffle=False, num\_workers=2)

classes = ('plane', 'car', 'bird', 'cat',

'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

class Net(nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 6, 5)

self.pool = nn.MaxPool2d(2, 2)

self.conv2 = nn.Conv2d(6, 16, 5)

self.fc1 = nn.Linear(16 \* 5 \* 5, 120)

self.fc2 = nn.Linear(120, 84)

self.fc3 = nn.Linear(84, 10)

def forward(self, x):

x = self.pool(F.relu(self.conv1(x)))

x = self.pool(F.relu(self.conv2(x)))

x = x.view(-1, 16 \* 5 \* 5)

x = F.relu(self.fc1(x))

x = F.relu(self.fc2(x))

x = self.fc3(x)

return x

net = Net()

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(2):

running\_loss = 0.0

for i, data in enumerate(trainloader, 0):

inputs, labels = data

optimizer.zero\_grad()

outputs = net(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()

if i % 2000 == 1999:

print('[%d, %5d] loss: %.3f' %

(epoch + 1, i + 1, running\_loss / 2000))

running\_loss = 0.0

correct = 0

total = 0

with torch.no\_grad():

for data in testloader:

images, labels = data

outputs = net(images)

\_, predicted = torch.max(outputs.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (

100 \* correct / total))

### Create a scikit-learn pipeline that generates a random n-class classification problem and creates a pipeline to compute the mean and standard deviation on a training set so as to be able to later re-apply the same transformation on the testing set. Train the model Fit the dataset using the same pipeline and compute score on the test set.

from sklearn.datasets import make\_classification

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

X, y = make\_classification(random\_state=42)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)

pipe = make\_pipeline(StandardScaler(), LogisticRegression())

pipe.fit(X\_train, y\_train) # apply scaling on training data

Pipeline(steps=[('standardscaler', StandardScaler()),

('logisticregression', LogisticRegression())])

pipe.score(X\_test, y\_test)

### **Write a Python function that uses Hugging Face to train a model(**distilbert/distilbert-base-uncaseddistilbert/distilbert-base-uncased) **using the IMDb Reviews dataset. Its binary classification task. Involve any necessary preprocessing steps if required.**

from transformers import AutoTokenizer

from transformers import DataCollatorWithPadding

from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer

from huggingface\_hub import notebook\_login

import evaluate

import numpy as np

from datasets import load\_dataset

notebook\_login()

imdb = load\_dataset("imdb")

tokenizer = AutoTokenizer.from\_pretrained("distilbert/distilbert-base-uncased")

def preprocess\_function(examples):

return tokenizer(examples["text"], truncation=True)

tokenized\_imdb = imdb.map(preprocess\_function, batched=True)

data\_collator = DataCollatorWithPadding(tokenizer=tokenizer)

accuracy = evaluate.load("accuracy")

def compute\_metrics(eval\_pred):

predictions, labels = eval\_pred

predictions = np.argmax(predictions, axis=1)

return accuracy.compute(predictions=predictions, references=labels)

id2label = {0: "NEGATIVE", 1: "POSITIVE"}

label2id = {"NEGATIVE": 0, "POSITIVE": 1}

model = AutoModelForSequenceClassification.from\_pretrained(

"distilbert/distilbert-base-uncased", num\_labels=2, id2label=id2label, label2id=label2id

)

training\_args = TrainingArguments(

output\_dir="my\_awesome\_model",

learning\_rate=2e-5,

per\_device\_train\_batch\_size=16,

per\_device\_eval\_batch\_size=16,

num\_train\_epochs=2,

weight\_decay=0.01,

eval\_strategy="epoch",

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

push\_to\_hub=True,

)

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=tokenized\_imdb["train"],

eval\_dataset=tokenized\_imdb["test"],

processing\_class=tokenizer,

data\_collator=data\_collator,

compute\_metrics=compute\_metrics,

)

trainer.train()

### 

### Build a TensorFlow model for image classification using the Fashion MNIST dataset and integrate early stopping and model checkpoint callbacks during training.

import tensorflow as tf

import numpy as np

fashion\_mnist = tf.keras.datasets.fashion\_mnist

(train\_images, train\_labels), (test\_images, test\_labels) = fashion\_mnist.load\_data()

model = tf.keras.Sequential([

tf.keras.layers.Flatten(input\_shape=(28, 28)),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(10)

])  
early\_stopping = tf.keras.callbacks.EarlyStopping(

monitor='val\_loss', patience=3, restore\_best\_weights=True

)

model\_checkpoint = tf.keras.callbacks.ModelCheckpoint(

filepath='best\_fashion\_mnist\_model.h5',

monitor='val\_loss',

save\_best\_only=True

)

model.fit(

train\_images, train\_labels,

epochs=20,

validation\_split=0.2,

callbacks=[early\_stopping, model\_checkpoint]

)

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print(f'\nTest accuracy: {test\_acc:.4f}')

### Write a PyTorch class that subclasses torch.utils.data.Dataset to load image files and labels from a folder. Use this class with a DataLoader for batching. Use FashionMNIST dataset.

import torch

from torch.utils.data import Dataset

from torchvision import datasets

from torchvision.transforms import ToTensor

from torch.utils.data import DataLoader

training\_data = datasets.FashionMNIST(

root="data",

train=True,

download=True,

transform=ToTensor()

)

test\_data = datasets.FashionMNIST(

root="data",

train=False,

download=True,

transform=ToTensor()

)

train\_dataloader = DataLoader(training\_data, batch\_size=64, shuffle=True)

test\_dataloader = DataLoader(test\_data, batch\_size=64, shuffle=True)

### 

### Write a Python script that loads the Iris dataset, reduces its dimensionality using PCA, and fits a k-nearest neighbors classifier using scikit-learn

import pylab as pl

from scikits.learn import datasets

from scikits.learn.pca import PCA

iris = load\_iris()

X = iris.data

y = iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

pca = PCA(n\_components=2)

X\_train\_pca = pca.fit\_transform(X\_train\_scaled)

X\_test\_pca = pca.transform(X\_test\_scaled)

knn = KNeighborsClassifier(n\_neighbors=3)

knn.fit(X\_train\_pca, y\_train)

y\_pred = knn.predict(X\_test\_pca)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy}")

### Write a Java program using the Hadoop MapReduce API to count the frequency of each word in a large text file, such as a sample from Project Gutenberg.

**Mapper Code**

import java.io.IOException;

import org.apachehadoop.io.IntWritable;

import org.apache.hadoop.io.LongWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.Mapper;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reporter;

public class CharCountMapper extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable>{

public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,

Reporter reporter) throws IOException{

String line = value.toString();

String tokenizer[] = line.split("");

for(String SingleChar : tokenizer)

{

Text charKey = new Text(SingleChar);

IntWritable One = new IntWritable(1);

output.collect(charKey, One);

}

}

}

**Reducer Code**

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.MapReduceBase;

import org.apache.hadoop.mapred.OutputCollector;

import org.apache.hadoop.mapred.Reducer;

import org.apache.hadoop.mapred.Reporter;

public class CharCountReducer extends MapReduceBase

implements Reducer<Text, IntWritable, Text,

IntWritable> {

public void

reduce(Text key, Iterator<IntWritable> values,

OutputCollector<Text, IntWritable> output,

Reporter reporter) throws IOException

{

int sum = 0;

while (values.hasNext()) {

sum += values.next().get();

}

output.collect(key, new IntWritable(sum));

}

}

**Driver Code**

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapred.FileInputFormat;

import org.apache.hadoop.mapred.FileOutputFormat;

import org.apache.hadoop.mapred.JobClient;

import org.apache.hadoop.mapred.JobConf;

import org.apache.hadoop.mapred.TextInputFormat;

import org.apache.hadoop.mapred.TextOutputFormat;

public class CharCountDriver {

public static void main(String[] args)

throws IOException

{

JobConf conf = new JobConf(CharCountDriver.class);

conf.setJobName("CharCount");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(CharCountMapper.class);

conf.setCombinerClass(CharCountReducer.class);

conf.setReducerClass(CharCountReducer.class);

conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf,

new Path(args[0]));

FileOutputFormat.setOutputPath(conf,

new Path(args[1]));

JobClient.runJob(conf);

}

}

1. **Create a simple Spring Boot REST API that accepts JSON input and returns predictions made by a scikit-learn model trained on the Iris dataset and serialized using joblib.**

**Generalization tasks —**

1. **Create a python script that uses imbalanced-learn to do a binary classification model on Breast Cancer Wisconsin (Diagnostic) Dataset (WDBC), where your major class to minor class ratio is 99:1. You should use techniques like SMOTE to balance the dataset. Use a general model like XGboost. Do an analysis of metrics (like accuracy, precision, recall, f1 score, auc score and confusion matrix) before and after using imbalanced-learn techniques.**
2. **Write a script in python that uses tensorflow and keras library. Your task is to work on Semantic Segmentation using Oxford Pets Dataset and a U-Net Model. The semantic segmentation task would be to differentiate between background and the pet. Also calculate pixel-wise classification accuracy, model loss and IoU.**

Few shot examples

Few shot

### Task 1:

1. Linear Regression of random data using Tensorflow

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

np.random.seed(101)

# Generating random linear data

# There will be 50 data points ranging from 0 to 50

x = np.linspace(0, 50, 50)

y = np.linspace(0, 50, 50)

# Adding noise to the random linear data

x += np.random.uniform(-4, 4, 50)

y += np.random.uniform(-4, 4, 50)

n = len(x) # Number of data points

# Plot of Training Data

plt.scatter(x, y)

plt.xlabel('x')

plt.ylabel('y')

plt.title("Training Data")

plt.show()

X = tf.placeholder("float")

Y = tf.placeholder("float")

W = tf.Variable(np.random.randn(), name = "W")

b = tf.Variable(np.random.randn(), name = "b")

learning\_rate = 0.01

training\_epochs = 1000

# Hypothesis

y\_pred = tf.add(tf.multiply(X, W), b)

# Mean Squared Error Cost Function

cost = tf.reduce\_sum(tf.pow(y\_pred-Y, 2)) / (2 \* n)

# Gradient Descent Optimizer

optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)

# Global Variables Initializer

init = tf.global\_variables\_initializer()

# Starting the Tensorflow Session

with tf.Session() as sess:

# Initializing the Variables

sess.run(init)

# Iterating through all the epochs

for epoch in range(training\_epochs):

# Feeding each data point into the optimizer using Feed Dictionary

for (\_x, \_y) in zip(x, y):

sess.run(optimizer, feed\_dict = {X : \_x, Y : \_y})

# Displaying the result after every 50 epochs

if (epoch + 1) % 50 == 0:

# Calculating the cost a every epoch

c = sess.run(cost, feed\_dict = {X : x, Y : y})

print("Epoch", (epoch + 1), ": cost =", c, "W =", sess.run(W), "b =", sess.run(b))

# Storing necessary values to be used outside the Session

training\_cost = sess.run(cost, feed\_dict ={X: x, Y: y})

weight = sess.run(W)

bias = sess.run(b)

# Calculating the predictions

predictions = weight \* x + bias

print("Training cost =", training\_cost, "Weight =", weight, "bias =", bias, '\n')

1. Basic regression: Predict fuel efficiency

pip install -q seaborn

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import seaborn as sns

# Make NumPy printouts easier to read.

np.set\_printoptions(precision=3, suppress=True)

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

print(tf.\_\_version\_\_)

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data'

column\_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower', 'Weight',

'Acceleration', 'Model Year', 'Origin']

raw\_dataset = pd.read\_csv(url, names=column\_names,

na\_values='?', comment='\t',

sep=' ', skipinitialspace=True)

dataset = raw\_dataset.copy()

dataset.tail()

dataset.isna().sum()

dataset = dataset.dropna()

dataset['Origin'] = dataset['Origin'].map({1: 'USA', 2: 'Europe', 3: 'Japan'})

dataset = pd.get\_dummies(dataset, columns=['Origin'], prefix='', prefix\_sep='')

dataset.tail()

train\_dataset = dataset.sample(frac=0.8, random\_state=0)

test\_dataset = dataset.drop(train\_dataset.index)

train\_features = train\_dataset.copy()

test\_features = test\_dataset.copy()

train\_labels = train\_features.pop('MPG')

test\_labels = test\_features.pop('MPG')

horsepower = np.array(train\_features['Horsepower'])

horsepower\_normalizer = layers.Normalization(input\_shape=[1,], axis=None)

horsepower\_normalizer.adapt(horsepower)

horsepower\_model = tf.keras.Sequential([

horsepower\_normalizer,

layers.Dense(units=1)

])

horsepower\_model.summary()

horsepower\_model.compile(

optimizer=tf.keras.optimizers.Adam(learning\_rate=0.1),

loss='mean\_absolute\_error')

history = horsepower\_model.fit(

train\_features['Horsepower'],

train\_labels,

epochs=100,

# Suppress logging.

verbose=0,

# Calculate validation results on 20% of the training data.

validation\_split = 0.2)

### Task 2:

1. mnist

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import torch

from torch import optim

from torch import nn

from torch.utils.data import DataLoader

from tqdm import tqdm

# !pip install torchvision

import torchvision

import torch.nn.functional as F

import torchvision.datasets as datasets

import torchvision.transforms as transforms

# !pip install torchmetrics

import torchmetrics

batch\_size = 60

train\_dataset = datasets.MNIST(root="dataset/", download=True, train=True, transform=transforms.ToTensor())

train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)

test\_dataset = datasets.MNIST(root="dataset/", download=True, train=False, transform=transforms.ToTensor())

test\_loader = DataLoader(dataset=test\_dataset, batch\_size=batch\_size, shuffle=True)

class CNN(nn.Module):

def \_\_init\_\_(self, in\_channels, num\_classes):

"""

Building blocks of convolutional neural network.

Parameters:

\* in\_channels: Number of channels in the input image (for grayscale images, 1)

\* num\_classes: Number of classes to predict. In our problem, 10 (i.e digits from 0 to 9).

"""

super(CNN, self).\_\_init\_\_()

# 1st convolutional layer

self.conv1 = nn.Conv2d(in\_channels=in\_channels, out\_channels=8, kernel\_size=3, padding=1)

# Max pooling layer

self.pool = nn.MaxPool2d(kernel\_size=2, stride=2)

# 2nd convolutional layer

self.conv2 = nn.Conv2d(in\_channels=8, out\_channels=16, kernel\_size=3, padding=1)

# Fully connected layer

self.fc1 = nn.Linear(16 \* 7 \* 7, num\_classes)

def forward(self, x):

"""

Define the forward pass of the neural network.

Parameters:

x: Input tensor.

Returns:

torch.Tensor

The output tensor after passing through the network.

"""

x = F.relu(self.conv1(x)) # Apply first convolution and ReLU activation

x = self.pool(x) # Apply max pooling

x = F.relu(self.conv2(x)) # Apply second convolution and ReLU activation

x = self.pool(x) # Apply max pooling

x = x.reshape(x.shape[0], -1) # Flatten the tensor

x = self.fc1(x) # Apply fully connected layer

return x

x = x.reshape(x.shape[0], -1) # Flatten the tensor

x = self.fc1(x) # Apply fully connected layer

return x

# Define the loss function

criterion = nn.CrossEntropyLoss()

# Define the optimizer

optimizer = optim.Adam(model.parameters(), lr=0.001)

num\_epochs=10

for epoch in range(num\_epochs):

# Iterate over training batches

print(f"Epoch [{epoch + 1}/{num\_epochs}]")

for batch\_index, (data, targets) in enumerate(tqdm(dataloader\_train)):

data = data.to(device)

targets = targets.to(device)

scores = model(data)

loss = criterion(scores, targets)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

# Set up of multiclass accuracy metric

acc = Accuracy(task="multiclass",num\_classes=10)

# Iterate over the dataset batches

model.eval()

with torch.no\_grad():

for images, labels in dataloader\_test:

# Get predicted probabilities for test data batch

outputs = model(images)

\_, preds = torch.max(outputs, 1)

acc(preds, labels)

precision(preds, labels)

recall(preds, labels)

#Compute total test accuracy

test\_accuracy = acc.compute()

print(f"Test accuracy: {test\_accuracy}")

1. Mnizt 2

from torch.nn import Module

from torch.nn import Conv2d

from torch.nn import Linear

from torch.nn import MaxPool2d

from torch.nn import ReLU

from torch.nn import LogSoftmax

from torch import flatten

import matplotlib

matplotlib.use("Agg")

from pyimagesearch.lenet import LeNet

from sklearn.metrics import classification\_report

from torch.utils.data import random\_split

from torch.utils.data import DataLoader

from torchvision.transforms import ToTensor

from torchvision.datasets import KMNIST

from torch.optim import Adam

from torch import nn

import matplotlib.pyplot as plt

import numpy as np

import argparse

import torch

import time

class LeNet(Module):

def \_\_init\_\_(self, numChannels, classes):

# call the parent constructor

super(LeNet, self).\_\_init\_\_()

# initialize first set of CONV => RELU => POOL layers

self.conv1 = Conv2d(in\_channels=numChannels, out\_channels=20,

kernel\_size=(5, 5))

self.relu1 = ReLU()

self.maxpool1 = MaxPool2d(kernel\_size=(2, 2), stride=(2, 2))

# initialize second set of CONV => RELU => POOL layers

self.conv2 = Conv2d(in\_channels=20, out\_channels=50,

kernel\_size=(5, 5))

self.relu2 = ReLU()

self.maxpool2 = MaxPool2d(kernel\_size=(2, 2), stride=(2, 2))

# initialize first (and only) set of FC => RELU layers

self.fc1 = Linear(in\_features=800, out\_features=500)

self.relu3 = ReLU()

# initialize our softmax classifier

self.fc2 = Linear(in\_features=500, out\_features=classes)

self.logSoftmax = LogSoftmax(dim=1)

def forward(self, x):

# pass the input through our first set of CONV => RELU =>

# POOL layers

x = self.conv1(x)

x = self.relu1(x)

x = self.maxpool1(x)

# pass the output from the previous layer through the second

# set of CONV => RELU => POOL layers

x = self.conv2(x)

x = self.relu2(x)

x = self.maxpool2(x)

# flatten the output from the previous layer and pass it

# through our only set of FC => RELU layers

x = flatten(x, 1)

x = self.fc1(x)

x = self.relu3(x)

# pass the output to our softmax classifier to get our output

# predictions

x = self.fc2(x)

output = self.logSoftmax(x)

# return the output predictions

return output

# construct the argument parser and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-m", "--model", type=str, required=True,

help="path to output trained model")

ap.add\_argument("-p", "--plot", type=str, required=True,

help="path to output loss/accuracy plot")

args = vars(ap.parse\_args())

PyTorch: Training your first Convolutional Neural Network (CNN)

# define training hyperparameters

INIT\_LR = 1e-3

BATCH\_SIZE = 64

EPOCHS = 10

# define the train and val splits

TRAIN\_SPLIT = 0.75

VAL\_SPLIT = 1 - TRAIN\_SPLIT

# set the device we will be using to train the model

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# load the KMNIST dataset

print("[INFO] loading the KMNIST dataset...")

trainData = KMNIST(root="data", train=True, download=True,

transform=ToTensor())

testData = KMNIST(root="data", train=False, download=True,

transform=ToTensor())

# calculate the train/validation split

print("[INFO] generating the train/validation split...")

numTrainSamples = int(len(trainData) \* TRAIN\_SPLIT)

numValSamples = int(len(trainData) \* VAL\_SPLIT)

(trainData, valData) = random\_split(trainData,

[numTrainSamples, numValSamples],

generator=torch.Generator().manual\_seed(42))

# initialize the train, validation, and test data loaders

trainDataLoader = DataLoader(trainData, shuffle=True,

batch\_size=BATCH\_SIZE)

valDataLoader = DataLoader(valData, batch\_size=BATCH\_SIZE)

testDataLoader = DataLoader(testData, batch\_size=BATCH\_SIZE)

# calculate steps per epoch for training and validation set

trainSteps = len(trainDataLoader.dataset) // BATCH\_SIZE

valSteps = len(valDataLoader.dataset) // BATCH\_SIZE

# initialize the LeNet model

print("[INFO] initializing the LeNet model...")

model = LeNet(

numChannels=1,

classes=len(trainData.dataset.classes)).to(device)

# initialize our optimizer and loss function

opt = Adam(model.parameters(), lr=INIT\_LR)

lossFn = nn.NLLLoss()

# initialize a dictionary to store training history

H = {

"train\_loss": [],

"train\_acc": [],

"val\_loss": [],

"val\_acc": []

}

# measure how long training is going to take

print("[INFO] training the network...")

# loop over our epochs

for e in range(0, EPOCHS):

# set the model in training mode

model.train()

# initialize the total training and validation loss

totalTrainLoss = 0

totalValLoss = 0

# initialize the number of correct predictions in the training

# and validation step

trainCorrect = 0

valCorrect = 0

# loop over the training set

for (x, y) in trainDataLoader:

# send the input to the device

(x, y) = (x.to(device), y.to(device))

# perform a forward pass and calculate the training loss

pred = model(x)

loss = lossFn(pred, y)

# zero out the gradients, perform the backpropagation step,

# and update the weights

opt.zero\_grad()

loss.backward()

opt.step()

# add the loss to the total training loss so far and

# calculate the number of correct predictions

totalTrainLoss += loss

trainCorrect += (pred.argmax(1) == y).type(

torch.float).sum().item()

# switch off autograd for evaluation

with torch.no\_grad():

# set the model in evaluation mode

model.eval()

# loop over the validation set

for (x, y) in valDataLoader:

# send the input to the device

(x, y) = (x.to(device), y.to(device))

# make the predictions and calculate the validation loss

pred = model(x)

totalValLoss += lossFn(pred, y)

# calculate the number of correct predictions

valCorrect += (pred.argmax(1) == y).type(

torch.float).sum().item()

# calculate the average training and validation loss

avgTrainLoss = totalTrainLoss / trainSteps

avgValLoss = totalValLoss / valSteps

# calculate the training and validation accuracy

trainCorrect = trainCorrect / len(trainDataLoader.dataset)

valCorrect = valCorrect / len(valDataLoader.dataset)

# update our training history

H["train\_loss"].append(avgTrainLoss.cpu().detach().numpy())

H["train\_acc"].append(trainCorrect)

H["val\_loss"].append(avgValLoss.cpu().detach().numpy())

H["val\_acc"].append(valCorrect)

# print the model training and validation information

print("[INFO] EPOCH: {}/{}".format(e + 1, EPOCHS))

print("Train loss: {:.6f}, Train accuracy: {:.4f}".format(

avgTrainLoss, trainCorrect))

print("Val loss: {:.6f}, Val accuracy: {:.4f}\n".format(

avgValLoss, valCorrect))

### Task 3:

1. one

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.linear\_model import LogisticRegression

pipeline = Pipeline([

('scaler', StandardScaler()),

('pca', PCA(n\_components=2)),

('classifier', LogisticRegression())

])

1. two

pipe = Pipeline([

('scaler', StandardScaler()),

('selector', VarianceThreshold()),

('classifier', KNeighborsClassifier())

])

pipe.fit(X\_train, y\_train)

print('Training set score: ' + str(pipe.score(X\_train,y\_train)))

print('Test set score: ' + str(pipe.score(X\_test,y\_test)))

### Task 4:

1. In this example we learn to identify past vs present tense in <50 examples.

from typing import Callable, Dict, Sequence, Text

import numpy

import tensorflow

import toolz

import transformers

from toolz import curried

import gamla

def \_make\_model\_from\_input\_ids\_and\_masks(

input\_ids\_in: tensorflow.keras.layers.Input,

input\_masks\_in: tensorflow.keras.layers.Input,

transformer\_model,

):

model = toolz.pipe(

transformer\_model(input\_ids\_in, attention\_mask=input\_masks\_in)[0],

tensorflow.keras.layers.Bidirectional(

tensorflow.keras.layers.LSTM(

50, return\_sequences=True, dropout=0.1, recurrent\_dropout=0.1

)

),

tensorflow.keras.layers.GlobalMaxPool1D(),

tensorflow.keras.layers.Dense(50, activation="relu"),

tensorflow.keras.layers.Dropout(0.2),

tensorflow.keras.layers.Dense(1, activation="sigmoid"),

lambda layers: tensorflow.keras.Model(

inputs=[input\_ids\_in, input\_masks\_in], outputs=layers

),

)

for layer in model.layers[:3]:

layer.trainable = False

return model

def \_make\_model(max\_sequence\_length: int, transformer\_model):

return \_make\_model\_from\_input\_ids\_and\_masks(

tensorflow.keras.layers.Input(

shape=(max\_sequence\_length,), name="input\_ids", dtype="int32"

),

tensorflow.keras.layers.Input(

shape=(max\_sequence\_length,), name="attention\_mask", dtype="int32"

),

transformer\_model,

)

\_DISTILBERT\_MODEL = "distilbert-base-uncased"

def \_make\_model\_and\_encoder(max\_sequence\_length: int):

config = transformers.DistilBertConfig(dropout=0.2, attention\_dropout=0.2)

config.output\_hidden\_states = False

return (

\_make\_model(

max\_sequence\_length,

transformers.TFDistilBertModel.from\_pretrained(

\_DISTILBERT\_MODEL, config=config

),

),

transformers.DistilBertTokenizer.from\_pretrained(

\_DISTILBERT\_MODEL,

do\_lower\_case=True,

add\_special\_tokens=True,

max\_length=128,

pad\_to\_max\_length=True,

),

)

\_iterable\_to\_numpy\_array = toolz.compose\_left(

tuple, lambda data: numpy.asarray(data, dtype="int32"),

)

def \_encode(encoder: Callable[[Text], Dict[Text, numpy.array]]):

"""Warning - due to weirdness of `merge\_with` this will throw exception if iterable has only one element."""

return toolz.compose\_left(

curried.map(encoder),

gamla.star(curried.merge\_with(toolz.identity)),

curried.valmap(\_iterable\_to\_numpy\_array),

)

def train(positive: Sequence[Text], negative: Sequence[Text], epochs: int):

max\_sentence\_length = max(map(len, toolz.concat([positive, negative])))

model, encoder = \_make\_model\_and\_encoder(max\_sentence\_length)

model.compile(loss="mean\_squared\_error")

sentence\_encoder = \_encode(

lambda sentence: encoder.encode\_plus(

sentence,

add\_special\_tokens=True,

max\_length=max\_sentence\_length,

pad\_to\_max\_length=True,

return\_attention\_mask=True,

return\_token\_type\_ids=True,

)

)

X = sentence\_encoder(toolz.concat([positive, negative]))

y = \_iterable\_to\_numpy\_array(

toolz.concat([map(gamla.just(1), positive), map(gamla.just(0), negative)])

)

n\_train = int((len(positive) + len(negative)) \* 0.9)

trainX, testX = (

toolz.valmap(lambda d: d[:n\_train, :], X),

toolz.valmap(lambda d: d[n\_train:, :], X),

)

trainy, testy = y[:n\_train], y[n\_train:]

model.fit(

x=trainX, y=trainy, validation\_data=(testX, testy), epochs=epochs,

)

return model, sentence\_encoder

\_filter\_empty\_strip = toolz.compose\_left(

curried.map(str.strip), curried.filter(toolz.identity), curried.take(1000), tuple,

)

"""

Example usage; training a classifier to differentiate present and past tense.

model, encoder = bert\_seq2seq.train(

[

"go",

"walk",

"bring",

"think",

"build",

"drink",

"said",

"clean",

"do",

"bring",

"place",

"break",

"kick",

"code",

"type",

"kill",

"scare",

"make",

"bake",

"run",

],

[

"placed",

"did",

"brought",

"say",

"drank",

"went",

"walked",

"brought",

"thought",

"built",

"cleaned",

"broke",

"kicked",

"coded",

"typed",

"killed",

"scared",

"made",

"baked",

"ran",

],

epochs=50,

)

model.predict(

encoder(["run", "ran", "go", "went", "laugh", "laughed", "smoke", "smoked"])

)

1. distilbert-base-uncased-finetuned-sst-2-english, a distilled version of the BERT model fine-tuned on the Stanford Sentiment Treebank (SST-2)

import gradio as gr

from transformers import pipeline

def load\_model():

# Load a pre-trained HuggingFace pipeline for sentiment analysis

model\_pipeline = pipeline("sentiment-analysis", model="distilbert-base-uncased-finetuned-sst-2-english")

return model\_pipeline

def classify\_text(model, text):

# Use the loaded model to classify text

result = model(text)

return result

def main():

# Load the model

model = load\_model()

# Define the Gradio interface

interface = gr.Interface(

fn=lambda text: classify\_text(model, text),

inputs=gr.Textbox(lines=2, placeholder="Enter Text Here..."),

outputs="json",

title="Text Classification with HuggingFace",

description="This interface uses a HuggingFace model to classify text sentiments. Enter a sentence to see its classification."

)

# Launch the Gradio app

interface.launch()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Task 5

1. Img recognition using tensorflow

import matplotlib.pyplot as plt

import numpy as np

import os

import PIL

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras.models import Sequential

import pathlib

dataset\_url = "https://storage.googleapis.com/download.tensorflow.org/example\_images/flower\_photos.tgz"

data\_dir = tf.keras.utils.get\_file('flower\_photos', origin=dataset\_url, untar=True)

data\_dir = pathlib.Path(data\_dir)

data\_dir = data\_dir / "flower\_photos"

if not os.path.exists(data\_dir):

raise FileNotFoundError(f"Dataset directory not found at: {data\_dir}")

train\_ds = tf.keras.utils.image\_dataset\_from\_directory(

data\_dir,

validation\_split=0.2,

subset="training",

seed=123,

image\_size=(180, 180),

batch\_size=32)

val\_ds = tf.keras.utils.image\_dataset\_from\_directory(

data\_dir,

validation\_split=0.2,

subset="validation",

seed=123,

image\_size=(180,180),

batch\_size=32)

class\_names = train\_ds.class\_names

print(class\_names)

num\_classes = len(class\_names)

model = Sequential([

layers.Rescaling(1./255, input\_shape=(180,180, 3)),

layers.Conv2D(16, 3, padding='same', activation='relu'),

layers.MaxPooling2D(),

layers.Conv2D(32, 3, padding='same', activation='relu'),

layers.MaxPooling2D(),

layers.Conv2D(64, 3, padding='same', activation='relu'),

layers.MaxPooling2D(),

layers.Flatten(),

layers.Dense(128, activation='relu'),

layers.Dense(num\_classes)

])

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(

from\_logits=True),

metrics=['accuracy'])

model.summary()

epochs=10

history = model.fit(

train\_ds,

validation\_data=val\_ds,

epochs=epochs

)

1. CIFAR img classification

import tensorflow as tf

# Display the version

print(tf.\_\_version\_\_)

# other imports

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout

from tensorflow.keras.layers import GlobalMaxPooling2D, MaxPooling2D

from tensorflow.keras.layers import BatchNormalization

from tensorflow.keras.models import Model

# Load in the data

cifar10 = tf.keras.datasets.cifar10

# Distribute it to train and test set

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

print(x\_train.shape, y\_train.shape, x\_test.shape, y\_test.shape)

# Reduce pixel values

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# flatten the label values

y\_train, y\_test = y\_train.flatten(), y\_test.flatten()

# number of classes

K = len(set(y\_train))

# calculate total number of classes

# for output layer

print("number of classes:", K)

# Build the model using the functional API

# input layer

i = Input(shape=x\_train[0].shape)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(i)

x = BatchNormalization()(x)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)

x = BatchNormalization()(x)

x = MaxPooling2D((2, 2))(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = BatchNormalization()(x)

x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = BatchNormalization()(x)

x = MaxPooling2D((2, 2))(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

x = BatchNormalization()(x)

x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

x = BatchNormalization()(x)

x = MaxPooling2D((2, 2))(x)

x = Flatten()(x)

x = Dropout(0.2)(x)

# Hidden layer

x = Dense(1024, activation='relu')(x)

x = Dropout(0.2)(x)

# last hidden layer i.e.. output layer

x = Dense(K, activation='softmax')(x)

model = Model(i, x)

# model description

model.summary()

# Compile

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

# Fit

r = model.fit(

x\_train, y\_train, validation\_data=(x\_test, y\_test), epochs=50)

# Fit with data augmentation

# Note: if you run this AFTER calling

# the previous model.fit()

# it will CONTINUE training where it left off

batch\_size = 32

data\_generator = tf.keras.preprocessing.image.ImageDataGenerator(

width\_shift\_range=0.1, height\_shift\_range=0.1, horizontal\_flip=True)

train\_generator = data\_generator.flow(x\_train, y\_train, batch\_size)

steps\_per\_epoch = x\_train.shape[0] // batch\_size

r = model.fit(train\_generator, validation\_data=(x\_test, y\_test),

steps\_per\_epoch=steps\_per\_epoch, epochs=50)

# label mapping

labels = '''airplane automobile bird cat deerdog frog horseship truck'''.split()

# select the image from our test dataset

image\_number = 0

# display the image

plt.imshow(x\_test[image\_number])

# load the image in an array

n = np.array(x\_test[image\_number])

# reshape it

p = n.reshape(1, 32, 32, 3)

# pass in the network for prediction and

# save the predicted label

predicted\_label = labels[model.predict(p).argmax()]

# load the original label

original\_label = labels[y\_test[image\_number]]

# display the result

print("Original label is {} and predicted label is {}".format(

original\_label, predicted\_label))

### Task 6:

1. Creating custom dataset using pytorch

import os

import pandas as pd

from torchvision.io import read\_image

class CustomImageDataset(Dataset):

def \_\_init\_\_(self, annotations\_file, img\_dir, transform=None, target\_transform=None):

self.img\_labels = pd.read\_csv(annotations\_file)

self.img\_dir = img\_dir

self.transform = transform

self.target\_transform = target\_transform

def \_\_len\_\_(self):

return len(self.img\_labels)

def \_\_getitem\_\_(self, idx):

img\_path = os.path.join(self.img\_dir, self.img\_labels.iloc[idx, 0])

image = read\_image(img\_path)

label = self.img\_labels.iloc[idx, 1]

if self.transform:

image = self.transform(image)

if self.target\_transform:

label = self.target\_transform(label)

return image, label

1. Custom dataset and data loader for random data

import torch

from torch.utils.data import Dataset, DataLoader

import torch.nn as nn

import torch.optim as optim

import numpy as np

# Custom Dataset class

class CustomDataset(Dataset):

def \_\_init\_\_(self, data, labels):

self.data = data

self.labels = labels

def \_\_len\_\_(self):

return len(self.data)

def \_\_getitem\_\_(self, idx):

sample = self.data[idx]

label = self.labels[idx]

return sample, label

# Prepare data

data = np.random.randn(100, 3, 32, 32) # 100 samples of 3x32x32 images

labels = np.random.randint(0, 10, size=(100,)) # 100 labels in the range 0-9

# Create Dataset

dataset = CustomDataset(data, labels)

# Create DataLoader

dataloader = DataLoader(dataset, batch\_size=4, shuffle=True, num\_workers=2)

# Define a simple CNN model

class SimpleCNN(nn.Module):

def \_\_init\_\_(self):

super(SimpleCNN, self).\_\_init\_\_()

self.conv1 = nn.Conv2d(3, 16, 3, 1)

self.conv2 = nn.Conv2d(16, 32, 3, 1)

self.fc1 = nn.Linear(32\*6\*6, 128)

self.fc2 = nn.Linear(128, 10)

def forward(self, x):

x = torch.relu(self.conv1(x))

x = torch.max\_pool2d(x, 2, 2)

x = torch.relu(self.conv2(x))

x = torch.max\_pool2d(x, 2, 2)

x = torch.flatten(x, 1)

x = torch.relu(self.fc1(x))

x = self.fc2(x)

return x

model = SimpleCNN()

# Define loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop

num\_epochs = 5

for epoch in range(num\_epochs):

for batch\_data, batch\_labels in dataloader:

# Convert numpy arrays to torch tensors

batch\_data = torch.tensor(batch\_data, dtype=torch.float32)

batch\_labels = torch.tensor(batch\_labels, dtype=torch.long)

# Zero the parameter gradients

optimizer.zero\_grad()

# Forward pass

outputs = model(batch\_data)

loss = criterion(outputs, batch\_labels)

# Backward pass and optimization

loss.backward()

optimizer.step()

print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}')

### Task 7:

1. Pca and knn

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets

from sklearn.decomposition import PCA

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier, NeighborhoodComponentsAnalysis

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

n\_neighbors = 3

random\_state = 0

# Load Digits dataset

X, y = datasets.load\_digits(return\_X\_y=True)

# Split into train/test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.5, stratify=y, random\_state=random\_state

)

dim = len(X[0])

n\_classes = len(np.unique(y))

# Reduce dimension to 2 with PCA

pca = make\_pipeline(StandardScaler(), PCA(n\_components=2, random\_state=random\_state))

# Reduce dimension to 2 with LinearDiscriminantAnalysis

lda = make\_pipeline(StandardScaler(), LinearDiscriminantAnalysis(n\_components=2))

# Reduce dimension to 2 with NeighborhoodComponentAnalysis

nca = make\_pipeline(

StandardScaler(),

NeighborhoodComponentsAnalysis(n\_components=2, random\_state=random\_state),

)

# Use a nearest neighbor classifier to evaluate the methods

knn = KNeighborsClassifier(n\_neighbors=n\_neighbors)

1. Using generated data pca + log reg

import numpy as np

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

data = {

'Height': [170, 165, 180, 175, 160, 172, 168, 177, 162, 158],

'Weight': [65, 59, 75, 68, 55, 70, 62, 74, 58, 54],

'Age': [30, 25, 35, 28, 22, 32, 27, 33, 24, 21],

'Gender': [1, 0, 1, 1, 0, 1, 0, 1, 0, 0] # 1 = Male, 0 = Female

}

df = pd.DataFrame(data)

print(df)

X = df.drop('Gender', axis=1)

y = df['Gender']

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(df)

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X\_scaled)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_pca, y, test\_size=0.3, random\_state=42)

model = LogisticRegression()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

### Task 8:

1. Map-reduce word count general

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

StringTokenizer itr = new StringTokenizer(value.toString());

while (itr.hasMoreTokens()) {

word.set(itr.nextToken());

context.write(word, one);

}

}

}

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

1. Sample data map reduce fun

package hadoop;

import java.util.\*;

import java.io.IOException;

import java.io.IOException;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.conf.\*;

import org.apache.hadoop.io.\*;

import org.apache.hadoop.mapred.\*;

import org.apache.hadoop.util.\*;

public class ProcessUnits {

//Mapper class

public static class E\_EMapper extends MapReduceBase implements

Mapper<LongWritable ,/\*Input key Type \*/

Text, /\*Input value Type\*/

Text, /\*Output key Type\*/

IntWritable> /\*Output value Type\*/

{

//Map function

public void map(LongWritable key, Text value,

OutputCollector<Text, IntWritable> output,

Reporter reporter) throws IOException {

String line = value.toString();

String lasttoken = null;

StringTokenizer s = new StringTokenizer(line,"\t");

String year = s.nextToken();

while(s.hasMoreTokens()) {

lasttoken = s.nextToken();

}

int avgprice = Integer.parseInt(lasttoken);

output.collect(new Text(year), new IntWritable(avgprice));

}

}

//Reducer class

public static class E\_EReduce extends MapReduceBase implements Reducer< Text, IntWritable, Text, IntWritable > {

//Reduce function

public void reduce( Text key, Iterator <IntWritable> values,

OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

int maxavg = 30;

int val = Integer.MIN\_VALUE;

while (values.hasNext()) {

if((val = values.next().get())>maxavg) {

output.collect(key, new IntWritable(val));

}

}

}

}

//Main function

public static void main(String args[])throws Exception {

JobConf conf = new JobConf(ProcessUnits.class);

conf.setJobName("max\_eletricityunits");

conf.setOutputKeyClass(Text.class);

conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(E\_EMapper.class);

conf.setCombinerClass(E\_EReduce.class);

conf.setReducerClass(E\_EReduce.class);

conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(conf, new Path(args[0]));

FileOutputFormat.setOutputPath(conf, new Path(args[1]));

JobClient.runJob(conf);

}

}

APIs

Tensorflow

* import tensorflow as tf
* from tensorflow import keras
* from tensorflow.keras.models import Sequential
* from tensorflow.keras.layers import Dense, Input, Concatenate

Sklearn

* from sklearn.datasets import fetch\_california\_housing
* from sklearn.model\_selection import train\_test\_split
* from sklearn.preprocessing import StandardScaler
* from sklearn.datasets import make\_classification
* from sklearn.linear\_model import LogisticRegression
* from sklearn.pipeline import make\_pipeline
* from scikits.learn import datasets
* from scikits.learn.pca import PCA

Pytorch

* import torch
* import torch.nn as nn
* import torch.optim as optim
* import torchvision
* import torchvision.transforms as transforms
* import torch.nn.functional as F
* from torch.utils.data import Dataset
* from torchvision import datasets
* from torchvision.transforms import ToTensor
* from torch.utils.data import DataLoader

Misc

* import numpy as np
* import pandas as pd
* import matplotlib.pyplot as plt
* import evaluate

Hugging face

* from transformers import AutoTokenizer
* from transformers import DataCollatorWithPadding
* from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer
* from huggingface\_hub import notebook\_login
* from datasets import load\_dataset

Java Hadoop

* import java.io.IOException;
* import org.apache.hadoop.io.IntWritable;
* import org.apache.hadoop.io.LongWritable;
* import org.apache.hadoop.io.Text;
* import org.apache.hadoop.mapred.MapReduceBase;
* import org.apache.hadoop.mapred.Mapper;
* import org.apache.hadoop.mapred.OutputCollector;
* import org.apache.hadoop.mapred.Reporter;
* import java.util.Iterator;
* import org.apache.hadoop.mapred.Reducer;
* import org.apache.hadoop.fs.Path;
* import org.apache.hadoop.mapred.FileInputFormat;
* import org.apache.hadoop.mapred.FileOutputFormat;
* import org.apache.hadoop.mapred.JobClient;
* import org.apache.hadoop.mapred.JobConf;
* import org.apache.hadoop.mapred.TextInputFormat;
* import org.apache.hadoop.mapred.TextOutputFormat;