Task_1

January 5, 2024

1 TASK 1

1.1 Libraries

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.2 ReLU layer

[ ]:
```
# ReLU layer class
class ReLU:
    '''
    A class representing the Rectified Linear Unit (reLu) activation function.
    '''
    def __init__(self):
        self.input = None # placeholder for storing the input to the layer

    def forward_pass(self, input_data):
        self.input = input_data # store the input to use it in the backward pass
        return np.maximum(0, input_data) # apply the relu function: if x is␣
↪negative, max(0, x) will be 0; otherwise, will be x

    def backward_pass(self, output_gradient):
        '''
        Compute the backward pass through the reLu activation function.

        The method calculates the gradient of the reLu function with respect
        to its input 'x', given the gradient of the loss function with respect
        to the output of the relu layer ('gradient_values').

        Parameters:
        - gradient_values (numpy.ndarray): The gradient of the loss function␣
↪with respect
                                            to the output of the relu layer.

        Returns:
        - numpy.ndarray: The gradient of the loss function with respect to the
                         input of the relu layer.
```

```
        '''
        # apply the derivative of the relu function: if the input is negative,␣
↪the derivative is 0; otherwise, the derivative is 1
        return output_gradient * (self.input > 0)
        #return output_gradient * np.where(self.input > 0, 1.0, 0.0)
```

1.3 Sigmoid Layer

[ ]:
```
# Sigmoid layer class
class Sigmoid:
    '''
    A class representing the Sigmoid activation function.
    '''
    def __init__(self):
        self.output = None # placeholder for storing the output of the forward␣
↪pass

    def forward_pass(self, input_data):
        self.output = 1 / (1 + np.exp(-input_data)) # apply the sigmoid␣
↪function: f(x) = 1 / (1 + exp(-x))
        return self.output

    def backward_pass(self, output_gradient):
        '''
        Computes the backward pass of the Sigmoid activation function.

        Given the gradient of the loss function with respect to the output of␣
↪the
        Sigmoid layer ('output_gradient'), this method calculates the gradient␣
↪with respect
        to the Sigmoid input.

        Parameters:
        - output_gradient (numpy.ndarray): The gradient of the loss function␣
↪with respect
                                           to the output of the Sigmoid layer.

        Returns:
        - numpy.ndarray: The gradient of the loss function with respect to the
                         input of the Sigmoid layer.
        '''
        return output_gradient * (self.output * (1 - self.output))
```

1.4 Softmax Layer

[ ]:
```
# Softmax layer class
class Softmax:
    '''
    A class representing the Softmax activation function.
    '''

    def forward_pass(self, input_data):
        '''
        Computes the forward pass of the Softmax activation function.

        Parameters:
        - input_data (numpy.ndarray): A numpy array containing the input data␣
↪to which the Softmax
                                  function should be applied.

        Returns:
        - numpy.ndarray: The result of applying the Softmax function to␣
↪'input_data', with the
                              same shape as 'input_data'.
        '''
        exp_values = np.exp(input_data - np.max(input_data, axis=1,␣
↪keepdims=True)) # Shift the input data to avoid numerical instability in␣
↪exponential calculations
        output = exp_values / np.sum(exp_values, axis=1, keepdims=True)
        return output

    def backward_pass(self, dvalues):
        # The gradient of loss with respect to the input logits
        # directly passed through in case of softmax + categorical cross-entropy
        return dvalues
```

1.5 Dropout Layer

[ ]:
```
class Dropout:
    def __init__(self, probability):
        self.probability = probability

    def forward_pass(self, input_data):
        self.mask = np.random.binomial(1, 1-self.probability, size=input_data.
↪shape) / (1-self.probability)
        return input_data * self.mask

    def backward_pass(self, output_gradient):
        return output_gradient * self.mask
```

1.6 Dense Layer Class

[ ]:
```
# Dense layer class
class Layer:
    def __init__(self, input_size, output_size, l1=0.0, l2=0.0):
        self.weights = 0.01 * np.random.normal(0, 1/np.sqrt(input_size),␣
↪(input_size, output_size)) # Normal distribution initialisation
        self.biases = np.full((1, output_size), 0.001) # Initialise biases with␣
↪a small positive value
        self.velocity_weights = np.zeros_like(self.weights) # Initialise␣
↪(weights) velocity terms for momentum optimization
        self.velocity_biases = np.zeros_like(self.biases) # Initialise (biases)␣
↪velocity terms for momentum optimization
        self.l1 = l1 # L1 regularization coefficient (default 0.0).
        self.l2 = l2 # L2 regularization coefficient (default 0.0).
        self.input = None

    def forward_pass(self, input_data):
        self.input = input_data
        return np.dot(input_data, self.weights) + self.biases

    def backward_pass(self, output_gradient, learning_rate, optimizer='GD',␣
↪momentum=0.9):
        '''
        Computes the backward pass of the Dense layer.

        Parameters:
        - output_gradient: The gradient of the loss function with respect to␣
↪the output of the layer.

        - learning_rate: A hyperparameter that controls how much the weights␣
↪and biases are updated during training.

        - optimizer: Specifies the optimization technique to use. Can be 'GD'␣
↪for standard Gradient Descent or 'Momentum' for Gradient Descent with␣
↪Momentum.

        - momentum: A hyperparameter representing the momentum coefficient,␣
↪typically between 0 (no momentum) and 1.

        Returns:
        - numpy.ndarray: the gradient of the loss with respect to the layer's␣
↪inputs (which will be passed back to the previous layer in the network).
        '''
        # Regularization terms
        l1_reg = self.l1 * np.sign(self.weights)
        l2_reg = self.l2 * self.weights
```

```
        weights_gradient = np.dot(self.input.T, output_gradient) + l1_reg +␣
↪l2_reg
        input_gradient = np.dot(output_gradient, self.weights.T)
        biases_gradient = np.sum(output_gradient, axis=0, keepdims=True)

        if optimizer == 'GD':
            # Update weights and biases
            self.weights += learning_rate * weights_gradient
            self.biases += learning_rate * biases_gradient
        elif optimizer == 'Momentum':
            # Momentum update for weights and biases
            self.velocity_weights = momentum * self.velocity_weights +␣
↪learning_rate * weights_gradient
            self.velocity_biases = momentum * self.velocity_biases +␣
↪learning_rate * biases_gradient

            # Update weights and biases using velocity
            self.weights += self.velocity_weights
            self.biases += self.velocity_biases

        return input_gradient
```

1.7 NeuralNetwork Wrapper Class

[ ]:
```
# Neural Network wrapper class
class NeuralNetwork:
    def __init__(self):
        self.layers = [] # placeholder for storing the layers of the network so␣
↪we can propagate the infomation in a sequential order
        self.loss_history = [] # placeholder to store the (train) loss for␣
↪printing/plotting
        self.val_loss_history = [] #placeholder to store the loss function␣
↪calculated on the validation set for printing/plotting
        self.accuracy_history = [] #placeholder to store the (train) accuracy␣
↪for printing/plotting
        self.val_accuracy_history = [] #placeholder to store the accuracy␣
↪calculated on the validation set for printing/plotting

    def add_layer(self, layer):
        '''
        Add the layer to the network
        '''
        self.layers.append(layer)

    def forward_pass(self, input_data):
```

```python
        '''
        Performs a forward pass through the network.
        It sequentially passes the input data through each layer, transforming
↪it according to each layer's operation.
        '''
        for layer in self.layers:
            input_data = layer.forward_pass(input_data)
        return input_data

    def prediction(self, input_data):
        '''
        Performs a forward pass through the network ignoring the dropout.
        '''
        for layer in self.layers:
            if not isinstance(layer, Dropout):
                input_data = layer.forward_pass(input_data)
        return input_data

    def compute_accuracy(self, predictions, labels):
        '''
        Computes the accuracy of predictions by comparing them with the true
↪labels.
        Accuracy is computed as the proportion of correct predictions to the
↪total number of predictions.
        '''
        return np.mean(predictions == labels)

    def backward_pass(self, output_gradient, learning_rate, optimizer='GD',
↪momentum=0.9):
        '''
        Performs the backward pass (backpropagation) for training.
        It propagates the gradient of the loss function backward through the
↪network, updating weights in the process if the layer is a dense one.
        '''
        for layer in reversed(self.layers):
            if isinstance(layer, Layer):
                output_gradient = layer.backward_pass(output_gradient,
↪learning_rate, optimizer, momentum)
            else:
                output_gradient = layer.backward_pass(output_gradient)

    def compute_categorical_cross_entropy_loss(self, y_pred, y_true):
        '''
        Computes the categorical cross entropy loss
        '''
```

```
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7) # Clip predictions to␣
↪prevent log(0)

        # Calculate the negative log of the probabilities of the correct class
        # Multiply with the one-hot encoded true labels and sum across classes
        loss = np.sum(y_true * -np.log(y_pred_clipped), axis=1)

        # Average loss over all samples
        return np.mean(loss)

    def compute_categorical_cross_entropy_gradient(self, y_pred, y_true):
        '''
        Calculates the gradient of the categorical cross entropy loss with␣
↪respect to the network's output, assuming that the output layer is the␣
↪softmax activation function.

        Parameters:
        - y_pred: Output of the softmax activation function.

        - y_true: One-hot encoded label array.
        '''
        # Assuming y_true is one-hot encoded and y_pred is the output of softmax
        y_pred_gradient = (y_pred - y_true) / len(y_pred)
        return y_pred_gradient

    def train(self, X_train, y_train, epochs=100, learning_rate=0.001,␣
↪optimizer='GD', momentum=0.9, batch_size=32, validation_split = 0.2, verbose␣
↪= 1):
        '''
        Conducts the training process over a specified number of epochs.

        Parameters:
        - X_train: The input features of the training data.

        - y_train: The target output (labels) of the training data.

        - epochs: The number of times the entire training dataset is passed␣
↪forward and backward through the neural network.

        - learning_rate: The step size at each iteration while moving toward a␣
↪minimum of the loss function.

        - optimizer: Specifies the optimization technique to use. Can be 'GD'␣
↪for standard Gradient Descent or 'Momentum' for Gradient Descent with␣
↪Momentum.
```

```
        - momentum: A hyperparameter representing the momentum coefficient,␣
↪typically between 0 (no momentum) and 1.

        - batch_size: The number of training examples used in one iteration.

        - validation_split: Fraction of the training data to be used as␣
↪validation data.

        - verbose: The mode of verbosity (0 = silent, 1 = update every 10␣
↪epochs, 2 = update every epoch).

        '''
        val_sample_size = int(len(X_train) * validation_split) # calculate␣
↪validation sample size based on validation split parameter

        # Shuffles the indices of the training data to ensure random␣
↪distribution
        indices = np.arange(len(X_train))
        np.random.shuffle(indices)
        X_train, y_train = X_train[indices], y_train[indices]

        X_train, y_train = X_train[val_sample_size:], y_train[val_sample_size:]␣
↪# splits the data into new training set.
        X_val, y_val = X_train[:val_sample_size], y_train[:val_sample_size] #␣
↪splits the data into new validation set.

        n_samples = len(X_train)

        for epoch in range(epochs):
            # Shuffles the indices of the training data at the beginning of␣
↪each epoch to improve generalisation
            indices = np.arange(n_samples)
            np.random.shuffle(indices)
            X_train = X_train[indices]
            y_train = y_train[indices]

            # Processing of the training data in batches
            for start_idx in range(0, n_samples, batch_size):
                end_idx = min(start_idx + batch_size, n_samples)
                batch_x = X_train[start_idx:end_idx]
                batch_y = y_train[start_idx:end_idx]

                output = self.forward_pass(batch_x) # forward pass to get the␣
↪output predictions
                loss_gradient = self.
↪compute_categorical_cross_entropy_gradient(batch_y, output)
```

```python
                self.backward_pass(loss_gradient, learning_rate, optimizer, momentum) # backward pass to update the network's weights

            # Calculate training loss for the epoch
            output = self.forward_pass(X_train)
            train_loss = self.compute_categorical_cross_entropy_loss(output, y_train)
            self.loss_history.append(train_loss)

            # Calculate training accuracy
            train_predictions = self.predict(X_train)
            train_accuracy = self.compute_accuracy(train_predictions, np.argmax(y_train, axis=1))
            self.accuracy_history.append(train_accuracy)

            # Calculate validation loss for the epoch
            val_output = self.prediction(X_val)  # ensure dropout is not applied
            val_loss = self.compute_categorical_cross_entropy_loss(val_output, y_val)
            self.val_loss_history.append(val_loss)

            # Calculate validation accuracy
            val_predictions = self.predict(X_val)
            val_accuracy = self.compute_accuracy(val_predictions, np.argmax(y_val, axis=1))
            self.val_accuracy_history.append(val_accuracy)

            # Printing
            if verbose == 1:
                if epoch % 10 == 0:
                    print(f"Epoch {epoch}/{epochs} --- Train Loss: {train_loss} --- Val Loss: {val_loss} --- Train Acc: {train_accuracy:.2f} --- Val Acc: {val_accuracy:.2f}")
            elif verbose == 2:
                print(f"Epoch {epoch}/{epochs} --- Train Loss: {train_loss} --- Val Loss: {val_loss} --- Train Acc: {train_accuracy:.2f} --- Val Acc: {val_accuracy:.2f}")
            epoch += 1

    def predict(self, X_test):
        '''
        Uses the trained network to make predictions on new data (X_test).
        '''
        output = self.prediction(X_test) # use prediction method to avoid dropout
```

```
        predictions = np.argmax(output, axis=1) # convert probabilities to␣
↪class predictions
        return predictions

    def plot_loss(self):
        '''
        Plots the loss history stored in self.loss_history over the epochs.
        '''
        plt.plot(self.loss_history, label = 'Train Loss')
        plt.plot(self.val_loss_history, label = 'Val Loss')
        plt.title("Loss over Epochs")
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.legend()
        plt.show()

    def plot_accuracy(self):
        plt.plot(self.accuracy_history, label='Train Accuracy')
        plt.plot(self.val_accuracy_history, label='Val Accuracy')
        plt.title("Accuracy over Epochs")
        plt.xlabel("Epoch")
        plt.ylabel("Accuracy")
        plt.legend()
        plt.show()
```

1.8 Standardisation Function

[ ]:
```
def standardize_data(X):
    # Calculate the mean and standard deviation for each feature
    means = X.mean(axis=0)
    stds = X.std(axis=0)

    # Avoid division by zero in case of a constant feature
    stds[stds == 0] = 1

    # Standardize each feature
    X_standardized = (X - means) / stds
    return X_standardized
```

1.9 MNIST dataset classification

[ ]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load dataset
digits = load_digits()
```

```
X, y = digits.data, digits.target

# Standardize the features
X = standardize_data(X)

# One-hot encode the labels
one_hot_encoder = OneHotEncoder(sparse=False)
y = one_hot_encoder.fit_transform(y.reshape(-1, 1))

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Create the neural network model
network = NeuralNetwork()
network.add_layer(Layer(64, 128))  # 64 inputs (8x8 images)
network.add_layer(ReLU())
network.add_layer(Dropout(0.25))
network.add_layer(Layer(128, 32))
network.add_layer(ReLU())
#network.add_layer(Layer(64, 32, l2=0.01))
#network.add_layer(ReLU())
network.add_layer(Layer(32, 10))  # 10 classes
network.add_layer(Softmax())

# Train the network
network.train(X_train, y_train, epochs=1000, learning_rate=0.01,␣
↪optimizer='Momentum', momentum=0.9, batch_size=64)

network.plot_loss()
network.plot_accuracy()

# Evaluate the performance of the model
y_pred = network.predict(X_test)
y_test = np.argmax(y_test, axis=1) # transoform back the One-Hot encoded array␣
↪of the labels

accuracy = np.mean(y_pred == y_test)
print(f"\nAccuracy: {accuracy:.2f}")
```

1.10 Dropout Experiment

[ ]:
```
from sklearn.datasets import load_breast_cancer

dataset = load_breast_cancer()

X = dataset.data
```

```
y = dataset.target

# Standardize the features
X = standardize_data(X)

# One-hot encode the labels
one_hot_encoder = OneHotEncoder(sparse=False)
y = one_hot_encoder.fit_transform(y.reshape(-1, 1))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,␣
↪random_state=42)
```

[ ]:
```
# Create the neural network model without regularisers
network = NeuralNetwork()
network.add_layer(Layer(X_train.shape[1], 128))
network.add_layer(ReLU())
network.add_layer(Layer(128, 32))
network.add_layer(ReLU())
network.add_layer(Layer(32, 2))
network.add_layer(Sigmoid())

# Train the network
network.train(X_train, y_train, epochs=2500, batch_size=X_train.shape[0],␣
↪optimizer='Momentum', learning_rate=0.1)

network.plot_loss()
```

[ ]:
```
# Create the neural network model without regularisers
network = NeuralNetwork()
network.add_layer(Layer(X_train.shape[1], 128))
network.add_layer(ReLU())
network.add_layer(Dropout(0.25))
network.add_layer(Layer(128, 32))
network.add_layer(ReLU())
network.add_layer(Dropout(0.25))
network.add_layer(Layer(32, 2))
network.add_layer(Sigmoid())

# Train the network
network.train(X_train, y_train, epochs=2500, batch_size=X_train.shape[0],␣
↪optimizer='Momentum', learning_rate=0.1)

network.plot_loss()
```

1.11 Optimizer Experiment

[ ]:
```
dataset = load_digits()

X = dataset.data
y = dataset.target

# Standardize the features
X = standardize_data(X)

# One-hot encode the labels
one_hot_encoder = OneHotEncoder(sparse=False)
y = one_hot_encoder.fit_transform(y.reshape(-1, 1))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

[ ]:
```
# Create the neural network model with Momentum optimizer (mini-batch)
network = NeuralNetwork()
network.add_layer(Layer(64, 128))  # 64 inputs (8x8 images)
network.add_layer(ReLU())
network.add_layer(Dropout(0.25))
network.add_layer(Layer(128, 32))
network.add_layer(ReLU())
network.add_layer(Layer(32, 10))  # 10 classes
network.add_layer(Softmax())

# Train the network
network.train(X_train, y_train, epochs=100, learning_rate=0.1,␣
↪optimizer='Momentum', momentum=0.5, batch_size=16)

network.plot_loss()
network.plot_accuracy()
```

[ ]:
```
# Evaluate the performance of the model
y_pred = network.predict(X_test)
y_test_ = np.argmax(y_test, axis=1) # transoform back the One-Hot encoded array␣
↪of the labels

accuracy = np.mean(y_pred == y_test_)
print(f"\nAccuracy: {accuracy:.2f}")
```

[ ]:
```
# Create the neural network model with GD optimizer (mini-batch variant)
network = NeuralNetwork()
network.add_layer(Layer(64, 128))  # 64 inputs (8x8 images)
network.add_layer(ReLU())
network.add_layer(Dropout(0.25))
```

```
network.add_layer(Layer(128, 32))
network.add_layer(ReLU())
network.add_layer(Layer(32, 10))  # 10 classes
network.add_layer(Softmax())

# Train the network
network.train(X_train, y_train, epochs=100, learning_rate=0.1, optimizer='GD',␣
↪batch_size=16)

network.plot_loss()
network.plot_accuracy()
```

[ ]:
```
# Evaluate the performance of the model
y_pred = network.predict(X_test)
y_test_ = np.argmax(y_test, axis=1) # transoform back the One-Hot encoded array␣
↪of the labels

accuracy = np.mean(y_pred == y_test_)
print(f"\nAccuracy: {accuracy:.2f}")
```

2 Next Step (Out of Scope)

- optimization of hyperparameters (random search and grid search function?)

[ ]:
```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load dataset
digits = load_digits()
X, y = digits.data, digits.target

# Standardize the features
X = standardize_data(X)

# One-hot encode the labels
one_hot_encoder = OneHotEncoder(sparse=False)
y = one_hot_encoder.fit_transform(y.reshape(-1, 1))
```

[ ]:
```
class RandomSearch:
    def __init__(self, network, param_grid, n_iter=10):
        self.network = NeuralNetwork
        self.param_grid = param_grid
        self.n_iter = n_iter

    def sample_params(self):
        sampled_params = {}
```

```
        for param, values in self.param_grid.items():
            sampled_params[param] = np.random.choice(values)
        return sampled_params

    def evaluate(self, X_train, y_train, X_val, y_val, params):
        network = self.network()

        config = params['layer_configs']
        # Add the first Dense layer
        network.add_layer(Layer(64, config['layer1_nodes'],␣
↪l1=config['layer1_l1'], l2=config['layer1_l2']))
        network.add_layer(ReLU())

        network.add_layer(Dropout(0.25))

        print(config['layer1_nodes'])
        # Add the second Dense layer
        network.add_layer(Layer(config['layer1_nodes'], config['layer2_nodes'],␣
↪l1=config['layer2_l1'], l2=config['layer2_l2']))
        network.add_layer(ReLU())

        # Add the output Softmax layer
        network.add_layer(Layer(config['layer2_nodes'], 10))
        network.add_layer(Softmax())

        network.train(X_train, y_train, epochs=params['epochs'],␣
↪learning_rate=params['learning_rate'],
                      optimizer=params['optimizer'],␣
↪momentum=params['momentum'], batch_size=params['batch_size'])

        y_pred = network.predict(X_val)
        y_val = np.argmax(y_val, axis=1)
        accuracy = np.mean(y_pred == y_val)

        return accuracy

    def search(self, X, y):
        best_params = None
        best_accuracy = 0

        for _ in range(self.n_iter):
            params = self.sample_params()
            X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.
↪2, random_state=42)
            accuracy = self.evaluate(X_train, y_train, X_val, y_val, params)

            if accuracy > best_accuracy:
```

```
                best_accuracy = accuracy
                best_params = params

            print(f"Params: {params}, Accuracy: {accuracy}")

        return best_params, best_accuracy
```

[ ]:
```
param_grid = {
    'layer_configs': [
        {
            'layer1_nodes': layer1_nodes,
            'layer1_l1': layer1_l1,
            'layer1_l2': layer1_l2,
            'layer2_nodes': layer2_nodes,
            'layer2_l1': layer2_l1,
            'layer2_l2': layer2_l2
        }
        for layer1_nodes in [32, 64, 128]  # Possible node counts for the first␣
 ↪Dense layer
        for layer1_l1 in [0.0, 0.01]       # L1 regularization for the first␣
 ↪Dense layer
        for layer1_l2 in [0.0, 0.01]       # L2 regularization for the first␣
 ↪Dense layer
        for layer2_nodes in [16, 32, 64]   # Possible node counts for the␣
 ↪second Dense layer
        for layer2_l1 in [0.0, 0.01]       # L1 regularization for the second␣
 ↪Dense layer
        for layer2_l2 in [0.0, 0.01]       # L2 regularization for the second␣
 ↪Dense layer
    ],
    'learning_rate': [0.01, 0.1, 0.5],
    'epochs': [100, 500, 1000],
    'optimizer': ['GD', 'Momentum'],
    'momentum': [0.5, 0.9],
    'batch_size': [16, 32, 64]
}
```

[ ]:
```
random_search = RandomSearch(NeuralNetwork, param_grid, n_iter=100)
best_params, best_accuracy = random_search.search(X, y)
print(f"\nBest Params: {best_params}, Best Accuracy: {best_accuracy}")
```

Task_2

January 5, 2024

[ ]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import os
import cv2
from PIL import Image


from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
```

[ ]:
```python
# Check if GPU is available
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

device
```

[ ]:
```python
# Dataset directories
training_path = './Training/'
testing_path = './Testing'
os.listdir(testing_path)
```

[ ]:
```python
# Custom Dataset Class for loading and retrieving samples
from torchvision.io import read_image

class TumorDataset(Dataset):
    def __init__(self, img_dir, transform=None):
        self.dataset_dir = img_dir
        self.category_dir = os.listdir(img_dir)
        self.data = self.load_data()
        self.transform = transform
```

```
    def __len__(self):
        return len(self.data)

    def load_data(self):
        data = []
        for i , category in enumerate(self.category_dir):
            category_path = os.path.join(self.dataset_dir , category)
            for file_name in os.listdir(category_path):
                img_path = os.path.join(category_path , file_name)
                data.append([img_path , i])
        return data

    def __getitem__(self, idx):
        img_path , label = self.data[idx]
        image = Image.open(img_path).convert("RGB")
        label = torch.tensor(label)
        if self.transform:
            tansformed_image = self.transform(image)

        return tansformed_image.to(device) , label.to(device)
```

[ ]:
```
# Transformations to the data
transform = transforms.Compose([
    transforms.Resize((300, 300)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.1858, 0.1858, 0.1859), (0.1841, 0.1841, 0.1841))
])
```

[ ]:
```
# Create separate train, test and validation datasets
train_data = TumorDataset(training_path,  transform)
test_data = TumorDataset(testing_path, transform )
```

[ ]:
```
train_size = int(0.8 * len(train_data))
val_size = len(train_data) - train_size
train_data, val_data = torch.utils.data.random_split(train_data, [train_size,␣
 ↪val_size])
```

[ ]:
```
print(len(train_data))
print(len(val_data))
print(len(test_data))
```

[ ]:
```
# Dataloader for each set, to create batches
train_loader = DataLoader(dataset = train_data, batch_size = 16, shuffle=True,␣
 ↪num_workers=0)
```

```
val_loader = DataLoader(dataset = val_data, batch_size = 16, shuffle=True,␣
 ↪num_workers=0)
test_loader = DataLoader(dataset = test_data, batch_size = 16, shuffle=True,␣
 ↪num_workers=0)
```

[ ]:
```
# https://discuss.pytorch.org/t/computing-the-mean-and-std-of-dataset/34949/2
# Calculate meman and variance of dataset
mean = 0.
std = 0.
for images, _ in val_loader:
    batch_samples = images.size(0) # batch size (the last batch can have␣
 ↪smaller size!)
    images = images.view(batch_samples, images.size(1), -1)
    mean += images.mean(2).sum(0)
    std += images.std(2).sum(0)

for images, _ in train_loader:
    batch_samples = images.size(0) # batch size (the last batch can have␣
 ↪smaller size!)
    images = images.view(batch_samples, images.size(1), -1)
    mean += images.mean(2).sum(0)
    std += images.std(2).sum(0)

for images, _ in test_loader:
    batch_samples = images.size(0) # batch size (the last batch can have␣
 ↪smaller size!)
    images = images.view(batch_samples, images.size(1), -1)
    mean += images.mean(2).sum(0)
    std += images.std(2).sum(0)

mean /= len(train_loader.dataset + val_loader.dataset + test_loader.dataset)
std /= len(train_loader.dataset + val_loader.dataset + test_loader.dataset)


print(mean)
print(std)
```

[ ]:
```
# Dataset visualization
trainimages, trainlabels = next(iter(train_loader))

print(f"Feature batch shape: {trainimages.size()}")
print(f"Labels batch shape: {trainlabels.size()}")
for i in range(5):
    img = trainimages[i].squeeze().cpu()
    label = trainlabels[i]
    plt.imshow(img.T, cmap="gray")
    plt.show()
```

```
    print(f"Label: {label}")
```

[ ]:
```
# CNN Architecture without backbone
import torchvision.models as models
from torchvision.models import resnet50, ResNet50_Weights, vgg16


cnn = models.vgg16(pretrained=True)

modules = list(cnn.children())[:-1]
cnn = torch.nn.Sequential(*modules)
for param in cnn.parameters():
    param.requires_grad = False

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3)
        self.conv2 = nn.Conv2d(16, 36, kernel_size=3)
        self.fc1 = nn.Linear(191844, 1024)
        self.fc2 = nn.Linear(1024, 4)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = x.view(x.shape[0],-1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

[ ]:
```
# CNN Architecture
import torchvision.models as models
from torchvision.models import resnet50, ResNet50_Weights, vgg16


cnn = models.vgg16(pretrained=True)

modules = list(cnn.children())[:-1]
cnn = torch.nn.Sequential(*modules)
for param in cnn.parameters():
    param.requires_grad = False

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.cnn = cnn
```

```
        self.fc1 = nn.Linear(25088, 1024)
        self.fc2 = nn.Linear(1024, 4)

    def forward(self, x):
        x = self.cnn(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

[ ]:
```
# Model instantiation
model = CNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr = 0.001)
```

[ ]:
```
%%time
train_losses = []
valid_losses = []

for epoch in range(1, 20 + 1):
    train_loss = 0.0
    valid_loss = 0.0

    # Start training
    model.train()
    for data, target in train_loader:
        data = data.to(device)
        target = target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * data.size(0)
    # Validation
    model.eval()
    for data, target in val_loader:

        data = data.to(device)
        target = target.to(device)

        output = model(data)

        loss = criterion(output, target)

        valid_loss += loss.item() * data.size(0)
```

```
    train_loss = train_loss/len(train_loader.sampler)
    valid_loss = valid_loss/len(val_loader.sampler)
    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch, train_loss, valid_loss))
```

1 Evaluation

[ ]:
```
model.eval()
predictions = []
truth = []
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        predictions.extend(predicted.cpu())
        truth.extend(labels.cpu())
        correct += (predicted == labels).sum().item()


print('Test Accuracy of the model: {} %'.format(100 * correct / total))
```

[ ]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,␣
 ↪recall_score, precision_score

print('Test Accuracy of the model: {} %'.format(100 * correct / total))
cm = confusion_matrix(truth, predictions)
print('Precision: ' + str(precision_score(truth, predictions, average='macro')))
print('Recall: ' + str(recall_score(truth, predictions, average='macro')))
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot()
plt.show()
```

[ ]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

plt.plot(train_losses, label='Training loss')
plt.plot(valid_losses, label='Validation loss')
```

```
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend(frameon=False)
```