Trainer

An appendix to the Mathematical Exploration of Janav Nagapatla and Aiden Lim
Contains: main.py, dataset.py, helpers.py, layer.py, network.py, export.py

This code is made publicly available at: https://github.com/jnagapatla-x-aidenlims/trainer

Part 1 of a series of 2 programs

Take note that these programs require NumPy to run

```python
# Import structures from other files
from dataset import Dataset
from helpers import sigmoid, sigmoid_prime, error
from layer import Layer
from network import Network
from export import export

# Print manifest
print("\033c", end="", flush=True)
print("Year 4 Mathematical Exploration 2024: Trainer", end="\033[K\n\a")
print("Janav Nagapatla and Aiden Lim", end="\033[K\n")
print("All rights reserved", end="\033[K\n")

# Import datasets
print("", end="\033[K\n")
print("Importing datasets", end="\033[K\n")

print("> (1/2) Training Dataset", end="\033[K\n")
training_dataset: Dataset = Dataset("train-images.idx3-ubyte",
                                    "train-labels.idx1-ubyte")

print("> (2/2) Testing Dataset", end="\033[K\n")
testing_dataset: Dataset = Dataset("t10k-images.idx3-ubyte",
                                   "t10k-labels.idx1-ubyte")

print("> Successfully imported 2 out of 2 datasets", end="\033[K\n")

# Initialise a network
print("", end="\033[K\n")
print("Creating network", end="\033[K\n")

network: Network = Network(error,
                           [
                               Layer(784, 30, sigmoid, sigmoid_prime),
                               Layer(30, 10, sigmoid, sigmoid_prime)
                           ])

print("> Successfully initialised network with random weights", end="\033[K\n")
```

```python
# Evaluate the network
print("", end="\033[K\n")
print("Evaluating initial performance", end="\033[K\n")

network.evaluate(testing_dataset)

# Train the network
print("", end="\033[K\n")
print("Training network", end="\033[K\n")

network.train(training_dataset, testing_dataset, 30, 10, 3.0)

# Reevaluate the network
print("", end="\033[K\n")
print("Evaluating final performance", end="\033[K\n")

network.evaluate(testing_dataset)

# Save the network
print("", end="\033[K\n")
print("Saving model", end="\033[K\n")
filename: str = input("> Name of model (press return to abort): \a")
print("", end="\033[K\033[F")

if not filename:
    exit("> The program has been aborted.\033[K")

export(network, filename)
```

```python
# Import Python libraries
import numpy as np
from typing import BinaryIO


class Image:
    """
    Stores an image and its label from the MNIST datasets from: http://yann.lecun.com/exdb/mnist/
    """

    def __init__(self,
                 label: int,
                 pixels: np.ndarray) -> None:
        """
        Stores the input list into the object
        """

        self.label: int = label
        self.pixels: np.ndarray = pixels

    def __str__(self) -> str:
        """
        Returns a textual representation of the image
        """

        plot: str = ""

        for i, pixel in zip(range(1, 785), self.pixels):
            plot += ("█" if pixel else " ") + ("" if i % 28 else "\n")

        return plot

    def __int__(self) -> int:
        """
        Returns a numeric representation of the image
        """

        return self.label
```

```python
class Dataset:
    """
    Stores a representation of the MNIST datasets as a list of Image objects
    """

    def __init__(self,
                 images_file: str,
                 labels_file: str) -> None:
        """
        Converts an MNIST dataset file from http://yann.lecun.com/exdb/mnist/ to a dataset structure
        An adaptation of Joseph Redmon's MNIST-to-CSV code from https://pjreddie.com/projects/mnist-in-csv/
        """

        images: BinaryIO = open(images_file, "rb")
        labels: BinaryIO = open(labels_file, "rb")

        images.read(4)

        self.size: int = int.from_bytes(images.read(4))

        images.read(8)
        labels.read(8)

        self.images: list[Image] = [Image(ord(labels.read(1)), np.array([ord(images.read(1)) / 255
                                                                          for _ in range(784)]).reshape(784, 1))
                                    for _ in range(self.size)]

        images.close()
        labels.close()

    def __int__(self) -> int:
        """
        Returns a numeric representation of the length of the dataset
        """

        return self.size
```

```
if __name__ == "__main__":
    match input("Training or Testing Dataset: "):
        case "Training":
            dataset: Dataset = Dataset("train-images.idx3-ubyte",
                                       "train-labels.idx1-ubyte")
        case "Testing":
            dataset: Dataset = Dataset("t10k-images.idx3-ubyte",
                                       "t10k-labels.idx1-ubyte")
        case _:
            exit("That is not a suitable dataset.\033[K")

    try:
        index: int = int(input(f"Which image do you want to plot (1-{dataset.size}): "))

        print(dataset.images[index - 1])
        print(dataset.images[index - 1].label)
    except (ValueError, IndexError):
        exit("That is not a suitable image.\033[K")
```

```python
# Import Python libraries
import numpy as np


def sigmoid(value: np.ndarray) -> np.ndarray:
    """
    Returns logistic sigmoid at value
    """

    return 1.0 / (1.0 + np.exp(-value))


def sigmoid_prime(value: np.ndarray) -> np.ndarray:
    """
    Returns the derivative of logistic sigmoid at value
    """

    return (1.0 / (1.0 + np.exp(-value))) * (1 - (1.0 / (1.0 + np.exp(-value))))


def error(received: np.ndarray,
          answer: int) -> np.ndarray:
    """
    Returns a list of the difference between expected values and predicted values
    """
    desired = np.zeros((10, 1))
    desired[answer] = 1.0

    return received.reshape(10, 1) - desired


if __name__ == "__main__":
    exit("This script cannot be run on its own.\033[K")
```

```python
# Import Python libraries
import numpy as np
from typing import Callable


class Layer:
    """
    A representation of a connected layer
    Requires the number of input and output neurones of the layer and its desired activation function + derivative
    """

    def __init__(self,
                 input_size: int,
                 output_size: int,
                 activation: Callable[[np.ndarray], np.ndarray],
                 activation_prime: Callable[[np.ndarray], np.ndarray]) -> None:
        """
        Generates random weights and biases (as a starting point) or the layer
        """

        self.input_size: int = input_size
        self.output_size: int = output_size

        self.activation: Callable[[np.ndarray], np.ndarray] = activation
        self.activation_prime: Callable[[np.ndarray], np.ndarray] = activation_prime

        self.weights: np.ndarray = np.random.randn(output_size, input_size)
        self.biases: np.ndarray = np.random.randn(output_size, 1)

    def forward(self,
                previous: np.ndarray) -> np.ndarray:
        """
        Conducts forward propagation and returns the output neurones
        """

        return self.activation(np.dot(self.weights, previous) + self.biases)
```

```python
    def nonactivated(self,
                     previous: np.ndarray) -> np.ndarray:
        """
        Conducts forward propagation without activation and returns the output neurones
        """

        return np.dot(self.weights, previous) + self.biases

    def update(self,
               weight_derivatives: np.ndarray,
               bias_derivatives: np.ndarray,
               rate: float,
               size: int) -> None:
        """
        Updates all weights and biases of the layer
        """

        self.weights = self.weights - rate * weight_derivatives / size
        self.biases = self.biases - rate * bias_derivatives / size


if __name__ == "__main__":
    exit("This script cannot be run on its own.\033[K")
```

```python
# Import Python libraries
import numpy as np
from typing import Callable
from random import shuffle

# Import structures from other files
from dataset import Image, Dataset
from layer import Layer


class Network:
    """
    A representation of a network
    Takes in the layers of the network and their parameters
    """

    def __init__(self,
                 error: Callable[[np.ndarray, int], np.ndarray],
                 layers: list[Layer]) -> None:
        """
        Creates a set of layers
        """

        self.error: Callable[[np.ndarray, int], np.ndarray] = error
        self.layers: list[Layer] = layers
        self.size: int = len(self.layers)

    def predict(self,
                image: Image) -> np.ndarray:
        """
        Forwards the image through the network of layers and returns the output of the last layer
        """

        evaluation = image.pixels
        for layer in self.layers:
            evaluation = layer.forward(evaluation)

        return evaluation
```

```python
def evaluate(self,
             dataset: Dataset,
             silent: bool = False) -> float:
    """
    Evaluates the network based on a given dataset
    """

    correct: int = 0

    for image, i in zip(dataset.images, range(1, dataset.size + 1)):
        evaluation: int = int(np.argmax(self.predict(image)))
        if evaluation == image.label:
            correct += 1

        if not silent:
            print(f"> Image {i} / {dataset.size}", end="\033[K\n")
            print(f"> Success {correct / i:.2%}", end="\033[K\n")
            print(f"> |{"█" * round(i / dataset.size * 50)}{"-" * (50 - round(i / dataset.size * 50))}|",
                  end="\033[K\033[F\033[F")

    if not silent:
        print(f"> Out of {dataset.size}, {correct} images were predicted accurately", end="\033[K\n")
        print(f"> That is a {correct / dataset.size:.2%} success rate", end="\033[K\n")

    return correct / dataset.size

def train(self,
          training_dataset: Dataset,
          testing_dataset: Dataset,
          epochs: int,
          size: int,
          rate: float) -> None:
    """
    Trains the network and updates weights for a number of epochs
    """

    for epoch in range(1, epochs + 1):
        shuffle(training_dataset.images)
```

```
            batches = [training_dataset.images[i:i + size] for i in range(0, training_dataset.size, size)]
            for batch in batches:
                self.batch(batch, rate)

            print(f"> Epoch {epoch} / {epochs}", end="\033[K\n")
            print(f"> Success {self.evaluate(testing_dataset, True):.2%}", end="\033[K\n")
            print(f"> |{"█" * round(epoch / epochs * 50)}{"-" * (50 - round(epoch / epochs * 50))}|",
                  end="\033[K\033[F\033[F")

        print(f"> After {epochs} epochs, success reached {self.evaluate(testing_dataset, True):.2%}",
              end="\033[K\n")

    def batch(self,
              batch: list[Image],
              rate: float) -> None:
        """
        Trains the network and updates weights for a number of epochs
        """

        weight_gradient = [np.zeros(layer.weights.shape) for layer in self.layers]
        bias_gradient = [np.zeros(layer.biases.shape) for layer in self.layers]

        for image in batch:
            weight_microgradient, bias_microgradient = self.backprop(image)
            weight_gradient = [wg + wmg for wg, wmg in zip(weight_gradient, weight_microgradient)]
            bias_gradient = [bg + bmg for bg, bmg in zip(bias_gradient, bias_microgradient)]

        for layer in range(self.size):
            self.layers[layer].update(weight_gradient[layer], bias_gradient[layer], rate, len(batch))

    def backprop(self,
                 image: Image) -> tuple[list[np.ndarray], list[np.ndarray]]:
        """
        Returns a tuple representing the gradient for the error for an image
        """

        weight_gradient = [np.zeros(layer.weights.shape) for layer in self.layers]
        bias_gradient = [np.zeros(layer.biases.shape) for layer in self.layers]
```

```
        activations = [image.pixels]
        preactivations = []

        for layer in self.layers:
            preactivations.append(layer.nonactivated(activations[-1]))
            activations.append(layer.activation(preactivations[-1]))

        delta = (self.error(activations[-1], image.label) *
                 self.layers[-1].activation_prime(preactivations[-1]))
        weight_gradient[-1] = np.dot(delta, activations[-2].T)
        bias_gradient[-1] = delta

        for layer in range(2, self.size + 1):
            delta = (np.dot(self.layers[-layer + 1].weights.T, delta) *
                     self.layers[-layer].activation_prime(preactivations[-layer]))
            weight_gradient[-layer] = np.dot(delta, activations[-layer - 1].T)
            bias_gradient[-layer] = delta

        return weight_gradient, bias_gradient


if __name__ == "__main__":
    exit("This script cannot be run on its own.\033[K")
```

```python
# Import Python libraries
from typing import TextIO

# Import structures from other files
from network import Network


def export(network: Network,
           filename: str) -> None:
    """
    Exports the weights and activations of the network and saves the result to the given filename
    """

    try:
        config: TextIO = open(f"{filename}.networkconfig", "x")
    except FileExistsError:
        exit("The file already exists.\033[K")

    print(f"> Created {config.name}", end="\033[K\n")

    config.write(f"Model: {filename.split("/")[-1]}\n")
    config.write("Program: Year 4 Mathematical Exploration 2024\n")
    config.write("Authors: Aiden Lim and Janav Nagapatla\n")
    print("> Manifest written", end="\033[K\n")

    config.write("--- Begin Network Configuration ---\n")

    for layer in network.layers:
        config.write("> New Layer\n")
        config.write(f"    > Input Neurones: {layer.input_size}\n")
        config.write(f"    > Output Neurones: {layer.output_size}\n")
        config.write(f"    > Activation Function: {layer.activation.__name__}\n")

        config.write(f"    > Weights:\n")
        for o in range(layer.output_size):
            for i in range(layer.input_size):
                config.write(f"        > {layer.weights[o, i]}\n")
```

```
        config.write(f"    > Biases:\n")
        for o in range(layer.output_size):
            config.write(f"        > {layer.biases[o, 0]}\n")

        print("> Layer written", end="\033[K\n")

    config.write("--- End Network Configuration ---")

    print(f"> The network configuration has been saved to {config.name}", end="\033[K\n")


if __name__ == "__main__":
    exit("This script cannot be run on its own.\033[K")
```

To God Be The Glory
The Best Is Yet To Be