```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
from abc import ABC, abstractmethod


class Activation(ABC):
    """Abstract class defining the common interface for all activation methods."""

    def __call__(self, Z):
        return self.forward(Z)

    @abstractmethod
    def forward(self, Z):
        pass


def initialize_activation(name: str) -> Activation:
    """Factory method to return an Activation object of the specified type."""
    if name == "linear":
        return Linear()
    elif name == "sigmoid":
        return Sigmoid()
    elif name == "tanh":
        return TanH()
    elif name == "arctan":
        return ArcTan()
    elif name == "relu":
        return ReLU()
    elif name == "softmax":
        return SoftMax()
    else:
        raise NotImplementedError("{} activation is not implemented".format(name))


class Linear(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for f(z) = z.

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        return Z

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for f(z) = z.

        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        gradient of loss w.r.t. input of this layer
```

```python
        """
        return dY


class Sigmoid(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for sigmoid function:
        f(z) = 1 / (1 + exp(-z))

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        ### YOUR CODE HERE ###
        return ...

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for sigmoid.

        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        gradient of loss w.r.t. input of this layer
        """
        ### YOUR CODE HERE ###
        return ...


class TanH(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for f(z) = tanh(z).

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        return 2 / (1 + np.exp(-2 * Z)) - 1

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for f(z) = tanh(z).

        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer

        Returns
        -------
        gradient of loss w.r.t. input of this layer
        """
        fn = self.forward(Z)
```

```python
        return dY * (1 - fn ** 2)


class ReLU(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for relu activation:
        f(z) = z if z >= 0
               0 otherwise

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        ### YOUR CODE HERE ###
        return np.maximum(0, Z)


    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for relu activation.

        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        gradient of loss w.r.t. input of this layer
        """
        ### YOUR CODE HERE ###
        dZ = np.where(Z <= 0, 0, 1)
        return dY * dZ



class SoftMax(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z: np.ndarray) -> np.ndarray:
        """Forward pass for softmax activation.
        Hint: The naive implementation might not be numerically stable.

        Parameters
        ----------
        Z  input pre-activations (any shape)

        Returns
        -------
        f(z) as described above applied elementwise to `Z`
        """
        ### YOUR CODE HERE ###
        m = np.max(Z, axis=-1, keepdims=True)
        mm = Z - m
        denom = np.sum(np.exp(mm), axis=-1, keepdims=True)
        numer = np.exp(mm)

        return np.divide(numer, denom)

    def backward(self, Z: np.ndarray, dY: np.ndarray) -> np.ndarray:
        """Backward pass for softmax activation.
```

```
        Parameters
        ----------
        Z   input to `forward` method
        dY  gradient of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        gradient of loss w.r.t. input of this layer
        """
        ### YOUR CODE HERE ###
        probs = self.forward(Z)
        res = []
        for i, pvector in enumerate(probs):
            pdiag = np.diag(pvector)
            reshaped = pvector.reshape((-1, 1))
            jacobian = pdiag - np.dot(reshaped, reshaped.T)
            grad = dY[i] @ jacobian
            res.append(grad)
        return np.array(res)


class ArcTan(Activation):
    def __init__(self):
        super().__init__()

    def forward(self, Z):
        return np.arctan(Z)

    def backward(self, Z, dY):
        return dY * 1 / (Z ** 2 + 1)
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
import math
from neural_networks.utils import normalize, standardize
from neural_networks.utils import integers_to_one_hot


def initialize_dataset(
    name,
    batch_size=50,
):
    if name == "iris":
        training_set = np.load('datasets/iris/iris_train_data.npy')
        training_labels = np.load('datasets/iris/iris_train_labels.npy')

        validation_set = np.load('datasets/iris/iris_val_data.npy')
        validation_labels = np.load('datasets/iris/iris_val_labels.npy')

        test_set = np.load('datasets/iris/iris_test_data.npy')
        test_labels = np.load('datasets/iris/iris_test_labels.npy')

        dataset = Dataset(
            training_set=training_set,
            training_labels=training_labels,
            validation_set=validation_set,
            validation_labels=validation_labels,
            test_set=test_set,
            test_labels=test_labels,
            batch_size=batch_size,
        )
        return dataset
    elif name == "mnist":
        training_set = np.load('datasets/mnist/mnist_train_data.npy')
        training_labels = np.load('datasets/mnist/mnist_train_labels.npy')

        validation_set = np.load('datasets/mnist/mnist_val_data.npy')
        validation_labels = np.load('datasets/mnist/mnist_val_labels.npy')

        training_set = training_set.astype(np.float32) / 255.0
        validation_set = validation_set.astype(np.float32) / 255.0

        training_labels = integers_to_one_hot(training_labels, 9)
        validation_labels = integers_to_one_hot(validation_labels, 9)

        dataset = Dataset(
            training_set=training_set,
            training_labels=training_labels,
            validation_set=validation_set,
            validation_labels=validation_labels,
            batch_size=batch_size,
        )
        return dataset
    else:
        raise NotImplementedError


class Data:
    def __init__(
        self,
        data,
        batch_size=50,
        labels=None,
        out_dim=None,
```

```
    ):

        self.data_ = data
        self.labels = labels
        self.out_dim = out_dim
        self.iteration = 0
        self.batch_size = batch_size
        self.n_samples = data.shape[0]
        self.samples_per_epoch = math.ceil(self.n_samples / batch_size)

    def shuffle(self):
        idxs = np.arange(self.n_samples)
        np.random.shuffle(idxs)

        self.data_ = self.data_[idxs]
        if self.labels is not None:
            self.labels = self.labels[idxs]

    def sample(self):
        if self.iteration == 0:
            self.shuffle()

        low = self.iteration * self.batch_size
        high = self.iteration * self.batch_size + self.batch_size

        self.iteration += 1
        self.iteration = self.iteration % self.samples_per_epoch

        if self.labels is not None:
            return self.data_[low:high], self.labels[low:high]
        else:
            return self.data_[low:high]

    def reset(self):
        self.iteration == 0


class Dataset:
    def __init__(
        self,
        training_set,
        training_labels,
        batch_size,
        validation_set=None,
        validation_labels=None,
        test_set=None,
        test_labels=None,
    ):

        self.batch_size = batch_size
        self.n_training = training_set.shape[0]
        self.n_validation = validation_set.shape[0]
        self.out_dim = training_labels.shape[1]

        self.train = Data(
            data=training_set,
            batch_size=batch_size,
            labels=training_labels,
            out_dim=self.out_dim,
        )

        if validation_set is not None:
            self.validate = Data(
                data=validation_set,
                batch_size=batch_size,
                labels=validation_labels,
                out_dim=self.out_dim,
            )

        if test_set is not None:
```

```
        self.test = Data(
            data=test_set,
            batch_size=batch_size,
            labels=test_labels,
            out_dim=self.out_dim,
        )
```

```python
"""
Author: Sophia Sanborn, Sagnik Bhattacharya
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas, github.com/sagnibak
"""

import numpy as np
from abc import ABC, abstractmethod

from neural_networks.activations import initialize_activation
from neural_networks.weights import initialize_weights
from collections import OrderedDict

from typing import Callable, List, Literal, Tuple, Union


class Layer(ABC):
    """Abstract class defining the `Layer` interface."""

    def __init__(self):
        self.activation = None

        self.n_in = None
        self.n_out = None

        self.parameters = {}
        self.cache = {}
        self.gradients = {}

        super().__init__()

    @abstractmethod
    def forward(self, z: np.ndarray) -> np.ndarray:
        pass

    def clear_gradients(self) -> None:
        self.cache = OrderedDict({a: [] for a, b in self.cache.items()})
        self.gradients = OrderedDict(
            {a: np.zeros_like(b) for a, b in self.gradients.items()}
        )

    def forward_with_param(
        self, param_name: str, X: np.ndarray,
    ) -> Callable[[np.ndarray], np.ndarray]:
        """Call the `forward` method but with `param_name` as the variable with
        value `param_val`, and keep `X` fixed.
        """

        def inner_forward(param_val: np.ndarray) -> np.ndarray:
            self.parameters[param_name] = param_val
            return self.forward(X)

        return inner_forward

    def _get_parameters(self) -> List[np.ndarray]:
        return [b for a, b in self.parameters.items()]

    def _get_cache(self) -> List[np.ndarray]:
        return [b for a, b in self.cache.items()]

    def _get_gradients(self) -> List[np.ndarray]:
        return [b for a, b in self.gradients.items()]


def initialize_layer(
    name: str,
    activation: str = None,
    weight_init: str = None,
```

```python
    n_out: int = None,
    kernel_shape: Tuple[int, int] = None,
    stride: int = None,
    pad: int = None,
    mode: str = None,
    eps: float = 1e-8,
    momentum: float = 0.95,
    keep_dim: str = "first",
) -> Layer:
    """Factory function for layers."""
    if name == "fully_connected":
        return FullyConnected(
            n_out=n_out, activation=activation, weight_init=weight_init,
        )

    elif name == "conv2d":
        return Conv2D(
            n_out=n_out,
            activation=activation,
            kernel_shape=kernel_shape,
            stride=stride,
            pad=pad,
            weight_init=weight_init,
        )

    elif name == "batchnorm1d":
        return BatchNorm1D(eps=eps, momentum=momentum,)

    elif name == "pool2d":
        return Pool2D(kernel_shape=kernel_shape, mode=mode, stride=stride, pad=pad)

    elif name == "flatten":
        return Flatten(keep_dim=keep_dim)

    else:
        raise NotImplementedError("Layer type {} is not implemented".format(name))


class FullyConnected(Layer):
    """A fully-connected layer multiplies its input by a weight matrix, adds
    a bias, and then applies an activation function.
    """

    def __init__(
        self, n_out: int, activation: str, weight_init="xavier_uniform"
    ) -> None:

        super().__init__()
        self.n_in = None
        self.n_out = n_out
        self.activation = initialize_activation(activation)

        # instantiate the weight initializer
        self.init_weights = initialize_weights(weight_init, activation=activation)

    def _init_parameters(self, X_shape: Tuple[int, int]) -> None:
        """Initialize all layer parameters (weights, biases)."""
        self.n_in = X_shape[1]

        ### BEGIN YOUR CODE ###

        W = self.init_weights((self.n_in, self.n_out))
        b = np.zeros((1, self.n_out))

        self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
        self.cache: OrderedDict = OrderedDict({"Z":[], "X":[]})  # cache for backprop
        self.gradients: OrderedDict = OrderedDict({"W": np.zeros_like(W), "b":np.zeros_li
ke(b)})  # parameter gradients initialized to zero
                                                   # MUST HAVE THE SAME KEYS AS 'self.parameters'
```

```python
        ### END YOUR CODE ###

    def forward(self, X: np.ndarray) -> np.ndarray:
        """Forward pass: multiply by a weight matrix, add a bias, apply activation.
        Also, store all necessary intermediate results in the `cache` dictionary
        to be able to compute the backward pass.

        Parameters
        ----------
        X  input matrix of shape (batch_size, input_dim)

        Returns
        -------
        a matrix of shape (batch_size, output_dim)
        """
        # initialize layer parameters if they have not been initialized
        if self.n_in is None:
            self._init_parameters(X.shape)

        ### BEGIN YOUR CODE ###

        # perform an affine transformation and activation
        W, b = self.parameters["W"], self.parameters["b"]
        Z = X @ W + b

        out = self.activation(Z)

        self.cache["Z"], self.cache["X"] = Z, X

        # store information necessary for backprop in `self.cache`

        ### END YOUR CODE ###

        return out

    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """Backward pass for fully connected layer.
        Compute the gradients of the loss with respect to:
            1. the weights of this layer (mutate the `gradients` dictionary)
            2. the bias of this layer (mutate the `gradients` dictionary)
            3. the input of this layer (return this)

        Parameters
        ----------
        dLdY  gradient of the loss with respect to the output of this layer
              shape (batch_size, output_dim)

        Returns
        -------
        gradient of the loss with respect to the input of this layer
        shape (batch_size, input_dim)
        """
        ### BEGIN YOUR CODE ###

        # unpack the cache

        # compute the gradients of the loss w.r.t. all parameters as well as the
        # input of the layer
        Z = self.cache["Z"]
        X = self.cache["X"]
        W = self.parameters["W"]
        dZ = self.activation.backward(Z, dLdY)
        dW = X.T @ dZ
        db = dZ.sum(axis=0, keepdims=True)
        dX = dZ @ W.T

        self.gradients["W"] = dW
        self.gradients["db"] = db

        # store the gradients in `self.gradients`
```

```
        # the gradient for self.parameters["W"] should be stored in
        # self.gradients["W"], etc.

        ### END YOUR CODE ###

        return dX


class BatchNorm1D(Layer):
    def __init__(
        self,
        # n_in: int,
        mode: str = "train",
        weight_init: str = "xavier_uniform",
        eps: float = 1e-8,
        momentum: float = 0.9,
    ) -> None:
        super().__init__()

        # self.n_in = None
        self.mode = mode

        # instantiate the weight initializer
        self.init_weights = initialize_weights(weight_init,)

        self.eps = eps
        self.momentum = momentum

    def _init_parameters(self, X_shape: Tuple[int, int]) -> None:
        """Initialize all layer parameters (weights, biases)."""
        self.n_in = X_shape[1]

        ### BEGIN YOUR CODE ###

        gamma = self.init_weights(X_shape)
        beta = np.zeros((1, self.n_in))

        self.parameters = OrderedDict({"gamma": gamma, "beta": beta}) # DO NOT CHANGE THE
 KEYS
        self.cache = OrderedDict({"X": [], "X_hat": [],
                                  "mu": [], "var": [],
                                  "running_mu": np.zeros((1,self.n_in)), "running_var": n
p.zeros((1,self.n_in))})
        # cache for backprop
        self.gradients: OrderedDict = OrderedDict({"gamma": [], "beta": []})  # parameter
 gradients initialized to zero
                                                     # MUST HAVE THE SAME KEYS AS 'self.parameters'

        ### END YOUR CODE ###

    def forward(self, X: np.ndarray, mode="train") -> np.ndarray:
        """ Forward pass for 1D batch normalization layer.
        Allows taking in an array of shape (B, C) and performs batch normalization over i
t. Bill's sidenote: I think we can
        make it to include cases of it being (B, C, L), but is it really necessary?

        We use Exponential Moving Average to update the running mean and variance. with a
lpha value being equal to self.gamma

        You should set the running mean and running variance to the mean and variance of
the first batch after initializing it.
        You should also not
        """
        ### BEGIN YOUR CODE ###

        # implement a batch norm forward pass

        # cache any values required for backprop

        ### END YOUR CODE ###
```

```
        if mode == "train":
            mu, var = np.mean(X, axis = 0), np.var(X, axis = 0)
            self.cache["mu"].append(mu)
            self.cache["var"].append(var)
            h_hat = (X - mu) / np.sqrt(var + self.eps)
            z = self.parameters["gamma"] * h_hat + self.parameters["beta"]

            self.cache["running_mu"] = self.momentum * self.cache["running_mu"] + (1 - se
lf.momentum) * mu
            self.cache["running_var"] = self.momentum * self.cache["running_var"] + (1 -
self.momentum) * var

            self.gradients["gamma"].append(h_hat)
            self.gradients["beta"].append(np.ones_like(X) * 1)
        else:
            mu = self.cache["running_mu"]
            var = self.cache["running_var"]

            h_hat = (X - mu) / np.sqrt(var + self.eps)

            z = self.parameters["gamma"] * h_hat + self.parameters["beta"]

        return z

    def backward(self, dY: np.ndarray) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        """
        Backward method for batch normalization layer. You don't need to implement this t
o get full credit, although it is
        fun to do so if you have the time.
        """

        ### BEGIN YOUR CODE ###

        # implement backward pass for batchnorm.

        ### END YOUR CODE ###


        return dX

class Conv2D(Layer):
    """Convolutional layer for inputs with 2 spatial dimensions."""

    def __init__(
        self,
        n_out: int,
        kernel_shape: Tuple[int, int],
        activation: str,
        stride: int = 1,
        pad: str = "same",
        weight_init: str = "xavier_uniform",
    ) -> None:

        super().__init__()
        self.n_in = None
        self.n_out = n_out
        self.kernel_shape = kernel_shape
        self.stride = stride
        self.pad = pad

        self.activation = initialize_activation(activation)
        self.init_weights = initialize_weights(weight_init, activation=activation)

    def _init_parameters(self, X_shape: Tuple[int, int, int, int]) -> None:
        """Initialize all layer parameters and determine padding."""
        self.n_in = X_shape[3]

        W_shape = self.kernel_shape + (self.n_in,) + (self.n_out,)
        W = self.init_weights(W_shape)
        b = np.zeros((1, self.n_out))
```

```
        self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
        self.cache = OrderedDict({"Z": [], "X": []}) # cache for backprop
        self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)}) # pa
rameter gradients initialized to zero
                                                                                     # MU
ST HAVE THE SAME KEYS AS 'self.parameters'

        if self.pad == "same":
            self.pad = ((W_shape[0] - 1) // 2, (W_shape[1] - 1) // 2)
        elif self.pad == "valid":
            self.pad = (0, 0)
        elif isinstance(self.pad, int):
            self.pad = (self.pad, self.pad)
        else:
            raise ValueError("Invalid Pad mode found in self.pad.")

    def forward(self, X: np.ndarray) -> np.ndarray:
        """Forward pass for convolutional layer. This layer convolves the input
        `X` with a filter of weights, adds a bias term, and applies an activation
        function to compute the output. This layer also supports padding and
        integer strides. Intermediates necessary for the backward pass are stored
        in the cache.

        Parameters
        ----------
        X  input with shape (batch_size, in_rows, in_cols, in_channels)

        Returns
        -------
        output feature maps with shape (batch_size, out_rows, out_cols, out_channels)
        """
        if self.n_in is None:
            self._init_parameters(X.shape)

        W = self.parameters["W"]
        b = self.parameters["b"]

        kernel_height, kernel_width, in_channels, out_channels = W.shape
        n_examples, in_rows, in_cols, in_channels = X.shape
        kernel_shape = (kernel_height, kernel_width)

        ### BEGIN YOUR CODE ###

        # implement a convolutional forward pass

        # cache any values required for backprop

        # don't pad n_examples, pad rows and cols, don't pad channels
        X_pad = np.pad(X, pad_width=((0,0), (self.pad[0], self.pad[0]), (self.pad[1], sel
f.pad[1]), (0, 0)), mode="constant")

        out_rows = (X_pad.shape[1] - kernel_height) // self.stride + 1
        out_cols = (X_pad.shape[2] - kernel_width) // self.stride + 1

        # make an empty Z that we can store our post-convolution values in
        Z = np.zeros((n_examples, out_rows, out_cols, out_channels))

        for row in range(out_rows):
            height_top = row * self.stride
            height_bottom = height_top + kernel_height
            for col in range(out_cols):
                width_left = col * self.stride
                width_right = width_left + kernel_width

                window = X_pad[:, height_top:height_bottom, width_left:width_right, :]
                Z[:, row, col, :] = np.einsum("bhwc,hwcf->bf", window, W)


        Z = Z + b
```

```
        out = self.activation(Z)

        self.cache["Z"] = Z
        self.cache["X"] = X

        ### END YOUR CODE ###

        return out

    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """Backward pass for conv layer. Computes the gradients of the output
        with respect to the input feature maps as well as the filter weights and
        biases.

        Parameters
        ----------
        dLdY  gradient of loss with respect to output of this layer
              shape (batch_size, out_rows, out_cols, out_channels)

        Returns
        -------
        gradient of the loss with respect to the input of this layer
        shape (batch_size, in_rows, in_cols, in_channels)
        """
        ### BEGIN YOUR CODE ###

        # perform a backward pass
        W, b = self.parameters["W"], self.parameters["b"]
        Z, X = self.cache["Z"], self.cache["X"]

        X_pad = np.pad(X, pad_width=((0,0), (self.pad[0], self.pad[0]), (self.pad[1], sel
f.pad[1]), (0, 0)), mode="constant")
        dX_pad = np.zeros_like(X_pad)

        kernel_height, kernel_width, in_channels, out_channels = W.shape
        n_examples, in_rows, in_cols, in_channels = X.shape
        out_rows = (X_pad.shape[1] - kernel_height) // self.stride + 1
        out_cols = (X_pad.shape[2] - kernel_width) // self.stride + 1

        dZ = self.activation.backward(Z, dLdY)

        # sum over d1, d2, and n of dZ
        db = np.sum(dZ, axis=(0,1,2)).reshape(1, -1)
        dW = np.zeros_like(W)

        for row in range(out_rows):
            height_top = row * self.stride
            height_bottom = height_top + kernel_height
            for col in range(out_cols):
                width_left = col * self.stride
                width_right = width_left + kernel_width
                # update our dx_pad tensor by adding gradients
                dX_grad = np.einsum("bf, hwcf->bhwc", dZ[:, row, col, :], W)
                dX_pad[:, height_top:height_bottom, width_left:width_right, :]  += dX_gra
d

                dW_grad = np.einsum('bhwc,bf->hwcf', X_pad[:, height_top:height_bottom, w
idth_left:width_right, :], dZ[:, row, col, :])
                dW += dW_grad

        self.gradients["W"] = dW
        self.gradients["b"] = db

        # adjust our dX_pad to correct dimensions
        dX = dX_pad[:, self.pad[0]:in_rows+self.pad[0], self.pad[1]:in_cols+self.pad[1],
:]
        ### END YOUR CODE ###

        return dX
```

```python
class Pool2D(Layer):
    """Pooling layer, implements max and average pooling."""

    def __init__(
        self,
        kernel_shape: Tuple[int, int],
        mode: str = "max",
        stride: int = 1,
        pad: Union[int, Literal["same"], Literal["valid"]] = 0,
    ) -> None:

        if type(kernel_shape) == int:
            kernel_shape = (kernel_shape, kernel_shape)

        self.kernel_shape = kernel_shape
        self.stride = stride

        if pad == "same":
            self.pad = ((kernel_shape[0] - 1) // 2, (kernel_shape[1] - 1) // 2)
        elif pad == "valid":
            self.pad = (0, 0)
        elif isinstance(pad, int):
            self.pad = (pad, pad)
        else:
            raise ValueError("Invalid Pad mode found in self.pad.")

        self.mode = mode

        if mode == "max":
            self.pool_fn = np.max
            self.arg_pool_fn = np.argmax
        elif mode == "average":
            self.pool_fn = np.mean

        self.cache = {
            "out_rows": [],
            "out_cols": [],
            "X_pad": [],
            "p": [],
            "pool_shape": [],
        }
        self.parameters = {}
        self.gradients = {}

    def forward(self, X: np.ndarray) -> np.ndarray:
        """Forward pass: use the pooling function to aggregate local information
        in the input. This layer typically reduces the spatial dimensionality of
        the input while keeping the number of feature maps the same.

        As with all other layers, please make sure to cache the appropriate
        information for the backward pass.

        Parameters
        ----------
        X  input array of shape (batch_size, in_rows, in_cols, channels)

        Returns
        -------
        pooled array of shape (batch_size, out_rows, out_cols, channels)
        """
        ### BEGIN YOUR CODE ###

        # implement the forward pass

        # cache any values required for backprop

        self.cache["X"] = X
        n_examples, in_rows, in_cols, in_channels = X.shape
        kernel_height, kernel_width = self.kernel_shape
```

```
        out_rows = int((in_rows + 2 * self.pad[0] - kernel_height) / self.stride + 1)
        out_cols = int((in_cols + 2 * self.pad[1] - kernel_width) / self.stride + 1)

        X_pad = np.pad(X, pad_width=((0,0), (self.pad[0], self.pad[0]), (self.pad[1], sel
f.pad[1]), (0,0)), mode="constant")
        X_pool = np.zeros((n_examples, out_rows, out_cols, in_channels))

        for row in range(out_rows):
            height_top = row * self.stride
            height_bottom = height_top + kernel_height
            for col in range(out_cols):
                width_left = col * self.stride
                width_right = width_left + kernel_width
                X_pool[:, row, col, :]  += self.pool_fn(X_pad[:, height_top:height_bottom
, width_left:width_right, :], axis=(1, 2))

        self.cache["X_pad"] = X_pad



        ### END YOUR CODE ###

        return X_pool

    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        """Backward pass for pooling layer.

        Parameters
        ----------
        dLdY  gradient of loss with respect to the output of this layer
              shape (batch_size, out_rows, out_cols, channels)

        Returns
        -------
        gradient of loss with respect to the input of this layer
        shape (batch_size, in_rows, in_cols, channels)
        """
        ### BEGIN YOUR CODE ###

        # perform a backward pass

        X = self.cache["X"]

        n_examples, in_rows, in_cols, in_channels = X.shape
        kernel_height, kernel_width = self.kernel_shape

        # can't use out_rows and out_cols in my cache? its an empty list
        out_rows = int((in_rows + 2 * self.pad[0] - kernel_height) / self.stride + 1)
        out_cols = int((in_cols + 2 * self.pad[1] - kernel_width) / self.stride + 1)

        X_pad = self.cache["X_pad"]
        dX = np.zeros_like(X_pad)

        for row in range(out_rows):
            height_top = row * self.stride
            height_bottom = height_top + kernel_height
            for col in range(out_cols):
                width_left = col * self.stride
                width_right = width_left + kernel_width

                if self.mode == "max":
                    window = X_pad[:, height_top:height_bottom, width_left:width_right, :
]
                    flat_window = window.reshape(n_examples, kernel_width*kernel_height,
in_channels)

                    # make a mask so we know which elements in tensor r maxes
                    indices = np.argmax(flat_window, axis=1)
                    mask = np.zeros_like(flat_window)
                    num_idx, channel_idx = np.indices((n_examples, in_channels))
```

```
                    mask[num_idx, indices, channel_idx] = 1

                    # reshape mask to our X_pad tensor's dimensions
                    mask = mask.reshape(n_examples, kernel_height, kernel_width, in_chann
els)
                    dX[:, height_top:height_bottom, width_left:width_right, :] += mask *
dLdY[:, row:row+1, col:col+1, :]
                else:
                    dX[:, height_top:height_bottom, width_left:width_right, :] += dLdY[:,
 row:row+1, col:col+1, :] / (kernel_height * kernel_width)

        # get rid of padding
        dX = dX[:, self.pad[0]:in_rows + self.pad[0], self.pad[1]:in_cols + self.pad[1],
:]

        return dX





        ### END YOUR CODE ###

        return gradX

class Flatten(Layer):
    """Flatten the input array."""

    def __init__(self, keep_dim: str = "first") -> None:
        super().__init__()

        self.keep_dim = keep_dim
        self._init_params()

    def _init_params(self):
        self.X = []
        self.gradients = {}
        self.parameters = {}
        self.cache = {"in_dims": []}

    def forward(self, X: np.ndarray, retain_derived: bool = True) -> np.ndarray:
        self.cache["in_dims"] = X.shape

        if self.keep_dim == -1:
            return X.flatten().reshape(1, -1)

        rs = (X.shape[0], -1) if self.keep_dim == "first" else (-1, X.shape[-1])
        return X.reshape(*rs)

    def backward(self, dLdY: np.ndarray) -> np.ndarray:
        in_dims = self.cache["in_dims"]
        gradX = dLdY.reshape(in_dims)
        return gradX
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
import matplotlib.pyplot as plt
import pickle
import os


class Logger:
    def __init__(
        self,
        model_name,
        model_args,
        data_args,
        save=False,
        plot=False,
        save_dir="experiments/",
    ):

        self.model_name = model_name
        self.model_args = model_args
        self.data_args = data_args
        self.save = save
        self.save_dir = save_dir + model_name + "/"
        self.plot = plot
        self.counter = 0
        self.log = {}

        if not os.path.isdir(save_dir):
            os.mkdir(save_dir)

        if not os.path.isdir(self.save_dir):
            os.mkdir(self.save_dir)

        with open(self.save_dir + "model_args", "wb") as f:
            pickle.dump(self.model_args, f)

        with open(self.save_dir + "data_args", "wb") as f:
            pickle.dump(self.data_args, f)

    def push(self, log):
        if self.counter == 0:
            self.log = {k: {} for k in log.keys()}

            # self.log = {k: [] if k != "params" else {} for k in log.keys()}
            if "params" in log.keys():

                self.log["params"] = {
                    k: {"max": [], "min": []} for k in log["params"].keys()
                }

            self.log["loss"] = {"train": [], "validate": []}
            self.log["error"] = {"train": [], "validate": []}

        self.counter += 1
        for k, v in log.items():
            if k == "params":
                for param, vals in v.items():
                    self.log["params"][param]["max"].append(vals["max"])
                    self.log["params"][param]["min"].append(vals["min"])

            else:
                self.log[k]["train"].append(v["train"])
                self.log[k]["validate"].append(v["validate"])
```

```python
        if self.save:
            with open(self.save_dir + "log", "wb") as f:
                pickle.dump(self.log, f)
            if self.plot:
                self._plot()

    def reset(self):
        self.log = {}
        self.counter = 0

    def _plot(self):
        for k, v in self.log.items():
            if k == "params":
                for param, vals in v.items():
                    plt.figure(figsize=(15, 10))
                    plt.plot(vals["max"], label="{}_max".format(param))
                    plt.plot(vals["min"], label="{}_min".format(param))
                    plt.legend()
                    plt.xlabel("epochs")
                    plt.ylabel(param)
                    plt.title(self.model_name)
                    plt.savefig(self.save_dir + param)
                    plt.close()
            else:
                plt.figure(figsize=(15, 10))
                plt.plot(v["train"], label="training")
                plt.plot(v["validate"], label="validation")
                plt.legend()
                plt.xlabel("epochs")
                plt.ylabel(k)
                plt.title(self.model_name)
                plt.savefig(self.save_dir + k)
                plt.close()
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
from abc import ABC, abstractmethod


class Loss(ABC):
    @abstractmethod
    def forward(self):
        pass

    @abstractmethod
    def backward(self):
        pass


def initialize_loss(name: str) -> Loss:
    if name == "cross_entropy":
        return CrossEntropy(name)
    else:
        raise NotImplementedError("{} loss is not implemented".format(name))


class CrossEntropy(Loss):
    """Cross entropy loss function."""

    def __init__(self, name: str) -> None:
        self.name = name

    def __call__(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
        return self.forward(Y, Y_hat)

    def forward(self, Y: np.ndarray, Y_hat: np.ndarray) -> float:
        """Computes the loss for predictions `Y_hat` given one-hot encoded labels
        `Y`.

        Parameters
        ----------
        Y      one-hot encoded labels of shape (batch_size, num_classes)
        Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

        Returns
        -------
        a single float representing the loss
        """
        B = Y.shape[0]
        return np.sum(Y * np.log(Y_hat))/-B

    def backward(self, Y: np.ndarray, Y_hat: np.ndarray) -> np.ndarray:
        """Backward pass of cross-entropy loss.
        NOTE: This is correct ONLY when the loss function is SoftMax.

        Parameters
        ----------
        Y      one-hot encoded labels of shape (batch_size, num_classes)
        Y_hat  model predictions in range (0, 1) of shape (batch_size, num_classes)

        Returns
        -------
        the gradient of the cross-entropy loss with respect to the vector of
        predictions, `Y_hat`
        """
        ### YOUR CODE HERE ###
        return -Y / (Y.shape[0] * Y_hat)
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

from abc import ABC, abstractmethod
import numpy as np

from neural_networks.losses import initialize_loss
from neural_networks.optimizers import initialize_optimizer
from neural_networks.layers import initialize_layer
from collections import OrderedDict
import pickle
from tqdm import tqdm
import pandas as pd

# imports for typing only
from neural_networks.utils import AttrDict
from neural_networks.datasets import Dataset
from typing import Any, Dict, List, Sequence, Tuple


def initialize_model(name, loss, layer_args, optimizer_args, logger=None, seed=None):

    return NeuralNetwork(
        loss=loss,
        layer_args=layer_args,
        optimizer_args=optimizer_args,
        logger=logger,
        seed=seed,
    )


class NeuralNetwork(ABC):
    def __init__(
        self,
        loss: str,
        layer_args: Sequence[AttrDict],
        optimizer_args: AttrDict,
        logger=None,
        seed: int = None,
    ) -> None:

        self.n_layers = len(layer_args)
        self.layer_args = layer_args
        self.logger = logger
        self.epoch_log = {"loss": {}, "error": {}}

        self.loss = initialize_loss(loss)
        self.optimizer = initialize_optimizer(**optimizer_args)
        self._initialize_layers(layer_args)

    def _initialize_layers(self, layer_args: Sequence[AttrDict]) -> None:
        self.layers = []
        for l_arg in layer_args[:-1]:
            l = initialize_layer(**l_arg)
            self.layers.append(l)

    def _log(self, loss: float, error: float, validation: bool = False) -> None:

        if self.logger is not None:
            if validation:

                self.epoch_log["loss"]["validate"] = round(loss, 4)
                self.epoch_log["error"]["validate"] = round(error, 4)
                self.logger.push(self.epoch_log)
                self.epoch_log = {"loss": {}, "error": {}}
```

```
        else:
            self.epoch_log["loss"]["train"] = round(loss, 4)
            self.epoch_log["error"]["train"] = round(error, 4)

    def save_parameters(self, epoch: int) -> None:
        parameters = {}
        for i, l in enumerate(self.layers):
            parameters[i] = l.parameters
        if self.logger is None:
            raise ValueError("Must have a logger")
        else:
            with open(
                self.logger.save_dir + "parameters_epoch{}".format(epoch), "wb"
            ) as f:
                pickle.dump(parameters, f)

    def forward(self, X: np.ndarray) -> np.ndarray:
        """One forward pass through all the layers of the neural network.

        Parameters
        ----------
        X  design matrix whose must match the input shape required by the
           first layer

        Returns
        -------
        forward pass output, matches the shape of the output of the last layer
        """
        ### YOUR CODE HERE ###
        # Iterate through the network's layers.
        Y = X
        for layer in self.layers:
            Y = layer.forward(Y)
        return Y

    def backward(self, target: np.ndarray, out: np.ndarray) -> float:
        """One backward pass through all the layers of the neural network.
        During this phase we calculate the gradients of the loss with respect to
        each of the parameters of the entire neural network. Most of the heavy
        lifting is done by the `backward` methods of the layers, so this method
        should be relatively simple. Also make sure to compute the loss in this
        method and NOT in `self.forward`.

        Note: Both input arrays have the same shape.

        Parameters
        ----------
        target   the targets we are trying to fit to (e.g., training labels)
        out      the predictions of the model on training data

        Returns
        -------
        the loss of the model given the training inputs and targets
        """
        ### YOUR CODE HERE ###
        # Compute the loss.
        # Backpropagate through the network's layers.
        L = self.loss.forward(target, out)
        dLdY = self.loss.backward(target, out)

        for layer in self.layers[::-1]:
            dLdY = layer.backward(dLdY)
        return L

    def update(self, epoch: int) -> None:
        """One step of gradient update using the derivatives calculated by
        `self.backward`.

        Parameters
        ----------
```

```
        epoch  the epoch we are currently on
        """
        param_log = {}
        for i, layer in enumerate(self.layers):
            for param_name, param in layer.parameters.items():
                if param_name != "null":  # FIXME: possible change needed to 'is not'
                    param_grad = layer.gradients[param_name]
                    # Optimizer needs to keep track of layers
                    delta = self.optimizer.update(
                        param_name + str(i), param, param_grad, epoch
                    )
                    layer.parameters[param_name] -= delta
                    if self.logger is not None:
                        param_log["{}{}".format(param_name, i)] = {}
                        param_log["{}{}".format(param_name, i)]["max"] = np.max(param)
                        param_log["{}{}".format(param_name, i)]["min"] = np.min(param)
            layer.clear_gradients()
        self.epoch_log["params"] = param_log

    def error(self, target: np.ndarray, out: np.ndarray) -> float:
        """Only calculate the error of the model's predictions given `target`.

        For classification tasks,
            error = 1 - accuracy

        For regression tasks,
            error = mean squared error

        Note: Both input arrays have the same shape.

        Parameters
        ----------
        target  the targets we are trying to fit to (e.g., training labels)
        out     the predictions of the model on features corresponding to
                `target`

        Returns
        -------
        the error of the model given the training inputs and targets
        """
        # classification error
        if self.loss.name == "cross_entropy":
            predictions = np.argmax(out, axis=1)
            target_idxs = np.argmax(target, axis=1)
            error = np.mean(predictions != target_idxs)

        # Error!
        else:
            raise NotImplementedError(
                "Error for {} loss is not implemented".format(self.loss)
            )

        return error

    def train(self, dataset: Dataset, epochs: int) -> None:
        """Train the neural network on using the provided dataset for `epochs`
        epochs. One epoch comprises one full pass through the entire dataset, or
        in case of stochastic gradient descent, one epoch comprises seeing as
        many samples from the dataset as there are elements in the dataset.

        Parameters
        ----------
        dataset  training dataset
        epochs   number of epochs to train for
        """
        # Initialize output layer
        args = self.layer_args[-1]
        args["n_out"] = dataset.out_dim
        output_layer = initialize_layer(**args)
        self.layers.append(output_layer)
```

```
        for i in range(epochs):
            training_loss = []
            training_error = []
            for _ in tqdm(range(dataset.train.samples_per_epoch)):
                X, Y = dataset.train.sample()
                Y_hat = self.forward(X)
                L = self.backward(np.array(Y), np.array(Y_hat))
                error = self.error(Y, Y_hat)
                self.update(i)
                training_loss.append(L)
                training_error.append(error)
            training_loss = np.mean(training_loss)
            training_error = np.mean(training_error)
            self._log(training_loss, training_error)

            validation_loss = []
            validation_error = []
            for _ in range(dataset.validate.samples_per_epoch):
                X, Y = dataset.validate.sample()
                Y_hat = self.forward(X)
                L = self.loss.forward(Y, Y_hat)
                error = self.error(Y, Y_hat)
                validation_loss.append(L)
                validation_error.append(error)
            validation_loss = np.mean(validation_loss)
            validation_error = np.mean(validation_error)
            self._log(validation_loss, validation_error, validation=True)

            print("Example target: {}".format(Y[0]))
            print("Example prediction: {}".format([round(x, 4) for x in Y_hat[0]]))
            print(
                "Epoch {} Training Loss: {} Training Accuracy: {} Val Loss: {} Val Accura
cy: {}".format(
                    i,
                    round(training_loss, 4),
                    round(1 - training_error, 4),
                    round(validation_loss, 4),
                    round(1 - validation_error, 4),
                )
            )

    def test(
        self, dataset: Dataset, save_predictions: bool = False
    ) -> Dict[str, List[np.ndarray]]:
        """Makes predictions on the data in `datasets`, returning the loss, and
        optionally returning the predictions and saving both.

        Parameters
        ----------
        dataset  test data
        save_predictions  whether to calculate and save the predictions

        Returns
        -------
        a dictionary containing the loss for each data point and optionally also
        the prediction for each data point
        """
        test_log = {"loss": [], "error": []}
        if save_predictions:
            test_log["prediction"] = []
        for _ in range(dataset.test.samples_per_epoch):
            X, Y = dataset.test.sample()
            Y_hat, L = self.predict(X, Y)
            error = self.error(Y, Y_hat)
            test_log["loss"].append(L)
            test_log["error"].append(error)
            if save_predictions:
                test_log["prediction"] += [x for x in Y_hat]
        test_loss = np.mean(test_log["loss"])
```

```
        test_error = np.mean(test_log["error"])
        print(
            "Test Loss: {} Test Accuracy: {}".format(
                round(test_loss, 4), round(1 - test_error, 4)
            )
        )
        if save_predictions:
            with open(self.logger.save_dir + "test_predictions.p", "wb") as f:
                pickle.dump(test_log, f)
        return test_log

    def predict(self, X: np.ndarray, Y: np.ndarray) -> Tuple[np.ndarray, float]:
        """Make a forward and backward pass to calculate the predictions and
        loss of the neural network on the given data.

        Parameters
        ----------
        X  input features
        Y  targets (same length as `X`)

        Returns
        -------
        a tuple of the prediction and loss
        """
        ### YOUR CODE HERE ###
        # Do a forward pass. Maybe use a function you already wrote?
        # Get the loss. Remember that the `backward` function returns the loss.
        pred = self.forward(X)
        return (pred, self.backward(target=Y, out=pred))
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
from abc import ABC, abstractmethod
from neural_networks.schedulers import initialize_scheduler


def initialize_optimizer(
    name,
    lr,
    lr_scheduler=None,
    momentum=None,
    clip_norm=None,
    lr_decay=None,
    staircase=None,
    stage_length=None,
):
    if name == "SGD":
        return SGD(
            lr=lr,
            lr_scheduler=lr_scheduler,
            momentum=momentum,
            clip_norm=clip_norm,
            lr_decay=lr_decay,
            staircase=staircase,
            stage_length=stage_length,
        )
    else:
        raise NotImplementedError


class Optimizer(ABC):
    def __init__(self):
        self.lr = None
        self.lr_scheduler = None


class SGD(Optimizer):
    def __init__(
        self,
        lr,
        lr_scheduler,
        momentum=0.0,
        clip_norm=None,
        lr_decay=0.9,
        stage_length=None,
        staircase=None,
    ):
        self.lr = lr
        self.lr_scheduler = initialize_scheduler(
            lr_scheduler,
            lr=lr,
            decay=lr_decay,
            stage_length=stage_length,
            staircase=staircase,
        )
        self.momentum = momentum
        self.clip_norm = clip_norm
        self.cache = {}

    def update(self, param_name, param, param_grad, epoch):
        if param_name not in self.cache:
            self.cache[param_name] = np.zeros_like(param)
```

```
        if self.clip_norm is not None:
            if np.linalg.norm(param_grad) > self.clip_norm:
                param_grad = (
                    param_grad * self.clip_norm / np.linalg.norm(param_grad)
                )

        lr = self.lr_scheduler(epoch)
        delta = (
            self.momentum * self.cache[param_name]
            + lr * param_grad
        )
        self.cache[param_name] = delta
        return delta
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
from abc import ABC, abstractmethod
import math


def initialize_scheduler(name, lr, decay=None, stage_length=None, staircase=None):
    if name == "constant":
        return Constant(lr=lr)
    elif name == "exponential":
        return Exponential(
            lr=lr, decay=decay, stage_length=stage_length, staircase=None
        )
    else:
        raise NotImplementedError("{} scheduler is not implemented".format(name))


class Scheduler(ABC):
    def __call__(self, epoch):
        return self.scheduled_lr(epoch)

    @abstractmethod
    def scheduled_lr(self, epoch=None):
        pass


class Constant(Scheduler):
    def __init__(self, lr=0.01):
        self.lr = lr

    def scheduled_lr(self, epoch):
        return self.lr


class Exponential(Scheduler):
    def __init__(self, lr=0.01, decay=0.9, stage_length=1000, staircase=False):
        self.lr = lr
        self.decay = decay
        self.stage_length = stage_length
        self.staircase = staircase

    def scheduled_lr(self, epoch):
        if self.staircase:
            stage = math.floor(epoch / self.stage_length)
        else:
            stage = epoch / self.stage_length

        return self.lr * self.decay ** stage
```

```python
import numpy as np
from numpy.linalg import norm
from typing import Callable


class AttrDict(dict):
    def __init__(self, *args, **kwargs):
        super(AttrDict, self).__init__(*args, **kwargs)
        self.__dict__ = self


def integers_to_one_hot(integer_vector, max_val=None):
    integer_vector = np.squeeze(integer_vector)
    if max_val == None:
        max_val = np.max(integer_vector)
    one_hot = np.zeros((integer_vector.shape[0], max_val + 1))
    for i, integer in enumerate(integer_vector):
        one_hot[i, integer] = 1.0
    return one_hot


def center(X, axis=0):
    return X - np.mean(X, axis=axis)


def normalize(X, axis=0, max_val=None):
    X -= np.min(X, axis=axis)
    if max_val is None:
        X /= np.max(X, axis=axis)
    else:
        X /= max_val
    return X


def standardize(X, axis=0):
    mean = np.mean(X, axis=axis)
    std = np.std(X, axis=axis)
    X -= mean
    X /= std + 1e-10
    return X


def check_gradients(
    fn: Callable[[np.ndarray], np.ndarray],
    grad: np.ndarray,
    x: np.ndarray,
    dLdf: np.ndarray,
    h: float = 1e-6,
) -> float:
    """Performs numerical gradient checking by numerically approximating
    the gradient using a two-sided finite difference.

    For each position in `x`, this function computes the numerical gradient as:
        numgrad = fn(x + h) - fn(x - h)
                  ---------------------
                           2h

    Next, we use the chain rule to compute the derivative of the input of `fn`
    with respect to the loss:
        numgrad = numgrad @ dLdf

    The function then returns the relative difference between the gradients:
        ||numgrad - grad||/||numgrad + grad||

    Parameters
    ----------
    fn       function whose gradients are being computed
    grad     supposed to be the gradient of `fn` at `x`
    x        point around which we want to calculate gradients
    dLdf     derivative of
```

```
    h         a small number (used as described above)

    Returns
    -------
    relative difference between the numerical and analytical gradients
    """
    # ONLY WORKS WITH FLOAT VECTORS
    if x.dtype != np.float32 and x.dtype != np.float64:
        raise TypeError(f"`x` must be a float vector but was {x.dtype}")

    # initialize the numerical gradient variable
    numgrad = np.zeros_like(x)

    # compute the numerical gradient for each position in `x`
    it = np.nditer(x, flags=["multi_index"], op_flags=["readwrite"])
    while not it.finished:
        ix = it.multi_index
        oldval = x[ix]
        x[ix] = oldval + h
        pos = fn(x).copy()
        x[ix] = oldval - h
        neg = fn(x).copy()
        x[ix] = oldval

        # compute the derivative, also apply the chain rule
        numgrad[ix] = np.sum((pos - neg) * dLdf) / (2 * h)
        it.iternext()

    return norm(numgrad - grad) / norm(numgrad + grad)
```

```python
"""
Author: Sophia Sanborn
Institution: UC Berkeley
Date: Spring 2020
Course: CS189/289A
Website: github.com/sophiaas
"""

import numpy as np
from abc import ABC, abstractmethod
import math


def initialize_weights(name, activation=None, mode="fan_in"):
    if name == "zeros":
        return Zeros()
    elif name == "ones":
        return Ones()
    elif name == "identity":
        return Identity()
    elif name == "uniform":
        return Uniform()
    elif name == "normal":
        return Normal()
    elif name == "constant":
        return Constant()
    elif name == "sparse":
        return Sparse()
    elif name == "he_uniform":
        return HeUniform(activation=activation, mode=mode)
    elif name == "he_normal":
        return HeNormal(activation=activation, mode=mode)
    elif name == "xavier_uniform":
        return XavierUniform(activation=activation)
    elif name == "xavier_normal":
        return XavierNormal(activation=activation)
    else:
        raise NotImplementedError


def _calculate_gain(activation, param=None):
    """
    Adapted from https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.calculate_gai
n
    """
    linear_fns = [
        "linear",
        "conv2d",
    ]
    if (
        activation in linear_fns
        or activation == "sigmoid"
        or activation == "softmax"
    ):
        return 1.0
    elif activation == "tanh":
        return 5.0 / 3.0
    elif activation == "relu":
        return math.sqrt(2.0)
    else:
        return 1.0


def _get_fan(shape, mode="sum"):
    fan_in, fan_out = shape[0], shape[-1]
    if mode == "fan_in":
        return fan_in
    elif mode == "fan_out":
        return fan_out
    elif mode == "sum":
```

```python
        return fan_in + fan_out
    elif mode == "separate":
        return fan_in, fan_out
    else:
        raise ValueError("Mode must be one of fan_in, fan_out, sum, or separate")


class WeightInitializer(ABC):
    @abstractmethod
    def __call__(self):
        pass


class Zeros(WeightInitializer):
    def __call__(self, shape):
        W = np.zeros(shape=shape)
        return W


class Ones(WeightInitializer):
    def __call__(self, shape):
        W = np.ones(shape=shape)
        return W


class Identity(WeightInitializer):
    def __call__(self, shape):
        fan_in, fan_out = _get_fan(shape, mode="separate")
        if fan_in != fan_out:
            raise ValueError(
                "Weight matrix shape must be square for identity initialization"
            )
        W = np.identity(n=fan_in)
        return W


class Uniform(WeightInitializer):
    def __init__(self, low=-1.0, high=1.0):
        self.low = low
        self.high = high

    def __call__(self, shape):
        W = np.random.uniform(self.low, self.high, size=shape)
        return W


class Normal(WeightInitializer):
    def __init__(self, mean=0, std=1.0):
        self.mean = mean
        self.std = std

    def __call__(self, shape):
        W = np.random.normal(self.mean, self.std, size=shape)
        return W


class Constant(WeightInitializer):
    def __init__(self, val=0.5):
        self.val = val

    def __call__(self, shape):
        W = np.full(shape, self.val)
        return W


class Preset(WeightInitializer):
    def __call__(self, preset_matrix):
        return preset_matrix
```

```python
class Sparse(WeightInitializer):
    def __init__(self, sparsity=0.1, std=0.01):
        self.sparsity = sparsity
        self.std = std

    def __call__(self, shape):
        n_rows, n_cols = shape
        n_zeros = int(math.ceil(n_rows * self.sparsity))

        W = np.random.normal(0, self.std, size=shape)
        for col_idx in range(n_cols):
            row_idx = np.arange(n_rows)
            np.random.shuffle(row_idx)
            zero_idx = row_idx[:n_zeros]
            W[zero_idx, col_idx] = 0
        return W


class XavierUniform(WeightInitializer):
    def __init__(self, activation=None):
        self.activation = activation

    def __call__(self, shape):
        fan = _get_fan(shape, mode="sum")
        gain = _calculate_gain(self.activation)
        std = gain * math.sqrt(2.0 / (fan))
        a = math.sqrt(3.0) * std
        W = np.random.uniform(-a, a, size=shape)
        return W


class XavierNormal(WeightInitializer):
    def __init__(self, activation=None):
        self.activation = activation

    def __call__(self, shape):
        fan = _get_fan(shape, mode="sum")
        gain = _calculate_gain(self.activation)
        std = gain * math.sqrt(2.0 / (fan))
        W = np.random.normal(0, std, size=shape)
        return W


class HeUniform(WeightInitializer):
    def __init__(self, activation=None, mode="fan_in"):
        self.activation = activation
        self.mode = mode

    def __call__(self, shape):
        fan = _get_fan(shape, mode=self.mode)
        gain = _calculate_gain(self.activation)
        std = gain / math.sqrt(fan)
        a = math.sqrt(3.0) * std
        W = np.random.uniform(-a, a, size=shape)
        return W


class HeNormal(WeightInitializer):
    def __init__(self, activation=None, mode="fan_in"):
        self.activation = activation
        self.mode = mode

    def __call__(self, shape):
        fan = _get_fan(shape, mode=self.mode)
        gain = _calculate_gain(self.activation)
        std = gain / math.sqrt(fan)
        W = np.random.normal(0, std, size=shape)
        return W
```

```
"""
Step 1: Define layer arguments

- Define the arguments for each layer in an attribute dictionary (AttrDict).
- An attribute dictionary is exactly like a dictionary, except you can access the values
as attributes rather than keys...for cleaner code :)
- See layers.py for the arguments expected by each layer type.
"""

from neural_networks.utils import AttrDict

conv1 = AttrDict(
    {
        "name": "conv2d",
        "n_out": 6,
        "kernel_shape": (5, 5),
        "stride": 1,
        "pad": "same",
        "activation": "relu",
        "weight_init": "he_uniform",
    }
)

pool1 = AttrDict(
    {
        "name": "pool2d",
        "kernel_shape": (2, 2),
        "mode": "max",
        "stride": 2,
        "pad": "valid"
    }
)

conv2 = AttrDict(
    {
        "name": "conv2d",
        "n_out": 16,
        "kernel_shape": (5, 5),
        "stride": 1,
        "pad": "valid",
        "activation": "relu",
        "weight_init": "he_uniform",
    }
)

pool2= AttrDict(
    {
        "name": "pool2d",
        "kernel_shape": (2, 2),
        "mode": "max",
        "stride": 2,
        "pad": "valid"
    }
)

flatten = AttrDict(
    {
        "name": "flatten"
    }
)


fc1 = AttrDict(
    {
        "name": "fully_connected",
        "activation": "relu",
        "weight_init": "he_uniform",
        "n_out": 120,
    }
```

```
)

fc2 = AttrDict(
    {
        "name": "fully_connected",
        "activation": "relu",
        "weight_init": "he_uniform",
        "n_out": 84,
    }
)

fc_out = AttrDict(
    {
        "name": "fully_connected",
        "activation": "softmax",  # Softmax for last layer for classification
        "weight_init": "he_uniform",
        "n_out": None
        # n_out is not defined for last layer. This will be set by the dataset.
    }
)

"""
Step 2: Collect layer argument dictionaries into a list.

- This defines the order of layers in the network.
"""

layer_args = [conv1, pool1, conv2, pool2, flatten, fc1, fc2, fc_out]

"""
Step 3: Define model, data, and logger arguments

- The list of layer_args is passed to the model initializer.
"""

optimizer_args = AttrDict(
    {
        "name": "SGD",
        "lr": 0.01,
        "lr_scheduler": "constant",
        "lr_decay": 0.99,
        "stage_length": 1000,
        "staircase": True,
        "clip_norm": 1.0,
        "momentum": 0.9,
    }
)

model_args = AttrDict(
    {
        "name": "feed_forward",
        "loss": "cross_entropy",
        "layer_args": layer_args,
        "optimizer_args": optimizer_args,
        "seed": 0,
    }
)

data_args = AttrDict(
    {
        "name": "mnist",
        "batch_size": 16,
    }
)

log_args = AttrDict(
    {"save": True, "plot": True, "save_dir": "experiments/",}
)

"""
```

```
Step 4: Set random seed

Warning! Random seed must be set before importing other modules.
"""

import numpy as np

np.random.seed(model_args.seed)

"""
Step 5: Define model name for saving
"""

model_name = "{}_{}layers_{}-lr{}_mom{}_seed{}".format(
    model_args.name,
    len(layer_args),
    fc1["n_out"],
    optimizer_args.lr,
    optimizer_args.momentum,
    model_args.seed,
)

"""
Step 6: Initialize logger, model, and dataset

- model_name, model_args, and data_args are passed to the logger for saving
- The logger is passed to the model.
"""

from neural_networks.models import initialize_model
from neural_networks.datasets import initialize_dataset
from neural_networks.logs import Logger


logger = Logger(
    model_name=model_name,
    model_args=model_args,
    data_args=data_args,
    save=log_args.save,
    plot=log_args.plot,
    save_dir=log_args.save_dir,
)


model = initialize_model(
    name=model_args.name,
    loss=model_args.loss,
    layer_args=model_args.layer_args,
    optimizer_args=model_args.optimizer_args,
    logger=logger,
)


dataset = initialize_dataset(
    name=data_args.name,
    batch_size=data_args.batch_size,
)


"""
Step 7: Train model!
"""

epochs = 5

print(
    "Training {} neural network on {} with {} for {} epochs...".format(
        model_args.name, data_args.name, optimizer_args.name, epochs
    )
)
```

```
print("Optimizer:")
print(optimizer_args)

model.train(dataset, epochs=epochs)
```

```
"""
Step 1: Define layer arguments

- Define the arguments for each layer in an attribute dictionary (AttrDict).
- An attribute dictionary is exactly like a dictionary, except you can access the values
as attributes rather than keys...for cleaner code :)
- See layers.py for the arguments expected by each layer type.
"""

from neural_networks.utils import AttrDict

fc1 = AttrDict(
    {
        "name": "fully_connected",
        "activation": "relu",
        "weight_init": "xavier_uniform",
        "n_out": 128,
    }
)

fc_out = AttrDict(
    {
        "name": "fully_connected",
        "activation": "softmax",  # Softmax for last layer for classification
        "weight_init": "xavier_uniform",
        "n_out": None
        # n_out is not defined for last layer. This will be set by the dataset.
    }
)

"""
Step 2: Collect layer argument dictionaries into a list.

- This defines the order of layers in the network.
"""

layer_args = [fc1, fc_out]

"""
Step 3: Define model, data, and logger arguments

- The list of layer_args is passed to the model initializer.
"""

optimizer_args = AttrDict(
    {
        "name": "SGD",
        "lr": 0.01,
        "lr_scheduler": "constant",
        "lr_decay": 0.99,
        "stage_length": 1000,
        "staircase": True,
        "clip_norm": 1.0,
        "momentum": 0.9,
    }
)

model_args = AttrDict(
    {
        "name": "feed_forward",
        "loss": "cross_entropy",
        "layer_args": layer_args,
        "optimizer_args": optimizer_args,
        "seed": 0,
    }
)

data_args = AttrDict(
    {
```

```
        "name": "iris",
        "batch_size": 25,
    }
)

log_args = AttrDict(
    {"save": True, "plot": True, "save_dir": "experiments/",}
)

"""
Step 4: Set random seed

Warning! Random seed must be set before importing other modules.
"""

import numpy as np

np.random.seed(model_args.seed)

"""
Step 5: Define model name for saving
"""

model_name = "{}_{}layers_{}-lr{}_mom{}_seed{}".format(
    model_args.name,
    len(layer_args),
    fc1["n_out"],
    optimizer_args.lr,
    optimizer_args.momentum,
    model_args.seed,
)

"""
Step 6: Initialize logger, model, and dataset

- model_name, model_args, and data_args are passed to the logger for saving
- The logger is passed to the model.
"""

from neural_networks.models import initialize_model
from neural_networks.datasets import initialize_dataset
from neural_networks.logs import Logger


logger = Logger(
    model_name=model_name,
    model_args=model_args,
    data_args=data_args,
    save=log_args.save,
    plot=log_args.plot,
    save_dir=log_args.save_dir,
)


model = initialize_model(
    name=model_args.name,
    loss=model_args.loss,
    layer_args=model_args.layer_args,
    optimizer_args=model_args.optimizer_args,
    logger=logger,
)


dataset = initialize_dataset(
    name=data_args.name,
    batch_size=data_args.batch_size,
)


"""
```

```
Step 7: Train model!
"""

epochs = 100

print(
    "Training {} neural network on {} with {} for {} epochs...".format(
        model_args.name, data_args.name, optimizer_args.name, epochs
    )
)

print("Optimizer:")
print(optimizer_args)

model.train(dataset, epochs=epochs)
model.test(dataset)
```