Activation Function Implementations:

Implementation of `activations.Linear` :

```
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  derivative of loss w.r.t. the output of this layer
            same shape as `Z`

        Returns
        -------
        derivative of loss w.r.t. input of this layer
        """
        return dY
```

Implementation of `activations.Sigmoid` :

```
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  derivative of loss w.r.t. the output of this layer
            same shape as `Z`

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

Implementation of `activations.ReLU` :

```python
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(Z,0)

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

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

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

        return dY*dZ
```

Implementation of `activations.SoftMax` :

```python
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 ###
         # Subtract the maximum value of each row for numerical stability
        Z -= np.max(Z, axis=1, keepdims=True)

        # Exponentiate the result
        exp_Z = np.exp(Z)

        # Normalize each row by dividing by the sum of all exponentiated values
        softmax_Z = exp_Z / np.sum(exp_Z, axis=1, keepdims=True) + 1e-9

        return softmax_Z

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

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

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

        S = self.forward(Z)  # calculate the output of the layer (softmax function applied to Z)
        N = Z.shape[0]  # number of samples in the input batch
        dZ = np.zeros_like(Z)  # initialize gradient with zeros

        # loop over each sample in the batch
        for i in range(N):
            # compute the Jacobian matrix of the softmax function at S[i]
            J = np.diag(S[i]) - np.outer(S[i], S[i])

            # multiply the Jacobian matrix with the derivative of the loss w.r.t. the output
            # of the layer to get the derivative of the loss w.r.t. the input to the layer
            dZ[i] = np.dot(J, dY[i])

        return dZ
```

Layer Implementations:

Implementation of `layers.FullyConnected` :

```python
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)) # adding one to the input dimension for the bias term
        b = np.zeros((1,self.n_out))

        self.parameters = OrderedDict({"W": W, "b": b})
        self.cache: OrderedDict = OrderedDict()  # cache for backprop
        self.gradients: OrderedDict = OrderedDict({"W": np.zeros_like(self.parameters["W"]), "b": np.zeros_like(self.par
ameters["b"])})# parameter gradients initialized to zero
                                           # MUST HAVE THE SAME KEYS AS `self.parameters`

        ### 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 ###
        Z = np.dot(X,self.parameters["W"])+self.parameters["b"]
        Y = self.activation.forward(Z)
        # store information necessary for backprop in `self.cache`
        self.cache['X'] = X
        self.cache['Z'] = Z
        self.cache['Y'] = Y

        ### END YOUR CODE ###

        return Y

    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  derivative of the loss with respect to the output of this layer
```

```
            shape (batch_size, output_dim)

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

        # unpack the cache
        X = self.cache['X']
        Z = self.cache['Z']

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


        # compute the gradients of the loss w.r.t. all parameters as well as the
        # input of the layer

        dLdZ = self.activation.backward(Z, dLdY)
        dLdW = np.dot(X.T, dLdZ)
        dLdb = np.sum(dLdZ, axis=0)
        dX = np.dot(dLdZ, W.T)


        # store the gradients in `self.gradients`
        # the gradient for self.parameters["W"] should be stored in
        # self.gradients["W"], etc.
        self.gradients['W'] = dLdW
        self.gradients['b'] = dLdb

        ### END YOUR CODE ###

        return dX
```

Implementation of `layers.Pool2D` :

```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

        ### 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

        ### END YOUR CODE ###

        return dX
```

Implementation of `layers.Conv2D.__init__` :

```
    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)
```

Implementation of `layers.Conv2D._init_parameters` :

```
    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})
        self.cache = OrderedDict({"Z": [], "X": []})
        self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)})

        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.")
```

Implementation of `layers.Conv2D.forward` :

```python
    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

        if self.pad == "same":
            pad_rows = int(np.ceil((self.stride*(in_rows-1) - in_rows + kernel_height)/2))
            pad_cols = int(np.ceil((self.stride*(in_cols-1) - in_cols + kernel_width)/2))
        elif self.pad == "valid":
            pad_rows, pad_cols = (0, 0)
        else:
            pad_rows, pad_cols = self.pad

        X_padded = np.pad(X, ((0,0), (pad_rows, pad_rows),
                              (pad_cols, pad_cols), (0,0)), mode='constant')

        out_rows = int(np.ceil(float(in_rows + 2*pad_rows - kernel_height + 1) / float(self.stride)))
        out_cols = int(np.ceil(float(in_cols + 2*pad_cols - kernel_width + 1) / float(self.stride)))
        out = np.zeros((n_examples, out_rows, out_cols, out_channels))

        for r in range(out_rows):
            for c in range(out_cols):
                h_start = r*self.stride
                h_end = h_start + kernel_height
                w_start = c*self.stride
                w_end = w_start + kernel_width

                X_slice = X_padded[:, h_start:h_end, w_start:w_end, :]
                out[:, r, c, :] = self.activation.forward(np.tensordot(X_slice,
                                                      W, axes=([1,2,3], [0,1,2])) + b)

        self.cache = {"Z": X_padded, "X": X}

        ### END YOUR CODE ###

        return out
```

Implementation of `layers.Conv2D.backward` :

```python
    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  derivative of loss with respect to output of this layer
              shape (batch_size, out_rows, out_cols, out_channels)

        Returns
        -------
        derivative 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 = self.parameters["W"]
        b = self.parameters["b"]
        X_padded = self.cache["Z"]
        X = self.cache["X"]
        kernel_height, kernel_width, in_channels, out_channels = W.shape

        batch_size, out_rows, out_cols = dLdY.shape[:-1]

        dX = np.zeros_like(X_padded)
        dLdW = np.zeros_like(W)
        dLdb = np.zeros_like(b)

        for r in range(out_rows):
            for c in range(out_cols):
                h_start = r*self.stride
                h_end = h_start + kernel_height
                w_start = c*self.stride
                w_end = w_start + kernel_width
                X_slice = X_padded[:, h_start:h_end, w_start:w_end, :]

                for i in range(batch_size):
                    dX[i, h_start:h_end, w_start:w_end, :] += np.tensordot(dLdY[i, r, c, :], W, axes=[0, 3])

                dLdW += np.tensordot(X_slice, dLdY[:, r, c, :], axes=[0, 0])

                dLdb += np.sum(dLdY[:, r, c, :], axis=0)

        if self.pad == "same":
            pad_rows = int(np.ceil((self.stride*(X.shape[1]-1) - X_padded.shape[1] + kernel_height)/2))
            pad_cols = int(np.ceil((self.stride*(X.shape[2]-1) - X_padded.shape[2] + kernel_width)/2))
            dX = dX[:, pad_rows:-pad_rows, pad_cols:-pad_cols, :]
        elif self.pad == "valid":
            dX = dX[:, kernel_height-1:-kernel_height+1:self.stride, kernel_width-1:-kernel_width+1:self.stride, :]
        else:
            dX = dX[:, self.pad[0]:-self.pad[0], self.pad[1]:-self.pad[1], :]

        self.gradients["W"] = dLdW
        self.gradients["b"] = dLdb


        ### END YOUR CODE ###
        return dX
```

Loss Function Implementations:

Implementation of `losses.CrossEntropy` :

```python
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
        """
        ### YOUR CODE HERE ###

        num_samples = Y.shape[0]
        num_classes = Y.shape[1]

        # Avoid division by zero by clipping Y_hat
        epsilon = 1e-8
        Y_hat = np.clip(Y_hat, epsilon, 1 - epsilon)


        # Calculate the cross-entropy loss
        loss = -1/num_samples * np.sum(Y * np.log(Y_hat))

        return loss

    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 derivative of the cross-entropy loss with respect to the vector of
        predictions, `Y_hat`
        """
        # Compute the number of samples in the batch

        m = Y.shape[0]
        epsilon = 1e-8

        # Compute the gradient of the loss with respect to Y_hat
        grad = -Y / ((m * Y_hat)+epsilon)

        return grad
```

Implementation of `losses.L2`:

```python
class L2(Loss):
    """Mean squared error loss."""

    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:
        """Compute the mean squared error loss for predictions `Y_hat` given
        regression targets `Y`.

        Parameters
        ----------
        Y      vector of regression targets of shape (batch_size, 1)
        Y_hat  vector of predictions of shape (batch_size, 1)

        Returns
        -------
        a single float representing the loss
        """
        ### YOUR CODE HERE ###
        return ...

    def backward(self, Y: np.ndarray, Y_hat: np.ndarray) -> np.ndarray:
        """Backward pass for mean squared error loss.

        Parameters
        ----------
        Y      vector of regression targets of shape (batch_size, 1)
        Y_hat  vector of predictions of shape (batch_size, 1)

        Returns
        -------
        the derivative of the mean squared error with respect to the last layer
        of the neural network
        """
        ### YOUR CODE HERE ###
        return ...
```

Model Implementations:

Implementation of `models.NeuralNetwork.forward` :

```python
    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.
        output = X
        for layer in self.layers:
            output = layer.forward(output)
        # Return the output of the last layer.
        return output
```

Implementation of `models.NeuralNetwork.backward` :

```python
    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.
        loss = self.loss(target, out)

        # Backpropagate through the network's layers.
        grad = self.loss.backward(target, out)
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

        # Return the loss.
        return loss
```

Implementation of `models.NeuralNetwork.predict` :

```python
    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
        Y_hat = self.forward(X)

        # Get the loss
        L = self.backward(Y_hat, Y)

        return Y_hat, L
```

In [ ]: