# Basic Network & CNN

May 19, 2024

Implementation of `activations.ReLU`:

```
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.array(Z, copy=True)
        dZ[Z<0], dZ[Z>=0] = 0, 1
        return dY * dZ
```

Implementation of `layers.FullyConnected`:

```
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))  # Takes the shape of the design matrix
        b = np.zeros((1, self.n_out))  # Initialize bias to zero, 1xn^(l+1)
        Z = []
        X = []

        self.parameters = OrderedDict({"W": W, "b": b}) # DO NOT CHANGE THE KEYS
        self.cache = OrderedDict({"Z": Z, "X": X})  # cache for backprop
        self.gradients = OrderedDict({"W": np.zeros_like(W), "b": np.zeros_like(b)})  # parame
                                                                                        # MUST HAVE THE SA

        ### 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)
```

```python
        """
        # initialize layer parameters if they have not been initialized
        if self.n_in is None:
            self._init_parameters(X.shape)

        ### BEGIN YOUR CODE ###
        W, b = self.parameters["W"], self.parameters["b"]
        # perform an affine transformation and activation
        Z = X @ W + b
        out = self.activation(Z)
        # store information necessary for backprop in `self.cache`
        self.cache["Z"], self.cache["X"] = Z, X
        ### 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
        W, b = self.parameters["W"], self.parameters["b"]
        Z, X = self.cache["Z"], self.cache["X"]
        # 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 = X.T @ dLdZ
        dX = dLdZ @ W.T
        dLdb = np.sum(dLdZ, axis=0, keepdims=True)
        # store the gradients in `self.gradients`
        # the gradient for self.parameters["W"] should be stored in
        # self.gradients["W"], etc.
        self.gradients["W"], self.gradients["b"] = dLdW, dLdb
```

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

        return dX
```

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 ###
        shifted_Z = Z - np.max(Z, axis=-1, keepdims=True)
        exp = np.exp(shifted_Z)
        return exp / np.sum(exp, axis=-1, keepdims=True)

    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 ###
        softmax = self.forward(Z)
        dZ = dY - np.sum(dY * softmax, axis=-1, keepdims=True)
        return dZ * softmax
```

Implementation of `losses.CrossEntropy`:

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

```python
    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 ###
        return -(np.sum(Y * np.log(Y_hat))) / Y.shape[0]

    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_hat)/Y.shape[0])
```

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.
        for layer in self.layers:
            X = layer.forward(X)
        return X
```

Implementation of `models.NeuralNetwork.backward`:

```
    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.
        loss = self.loss.forward(target, out)   # Y, Y_hat
        dLdY = self.loss.backward(target, out)
        for layer in reversed(self.layers):   # Backpropagate
            dLdY = layer.backward(dLdY)
        return loss
```

Implementation of `models.NeuralNetwork.predict`:

```
    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.
        Y_hat = self.forward(X)
        loss = self.backward(Y, Y_hat)
        return Y_hat, loss
```

Implementation of `layers.Conv2D.forward`:

```
    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 ###
        pad_h, pad_w = self.pad
        out_rows = (in_rows + 2*pad_h - kernel_height) // self.stride + 1 # formula from disc
        out_cols = (in_cols + 2*pad_w - kernel_width) // self.stride + 1  # formula from disc
        # Padding for four axis
        X_pad = np.pad(X, pad_width=((0, 0), (pad_h, pad_h), (pad_w, pad_w), (0, 0)), mode='co
        # implement a convolutional forward pass
        Z = np.empty((n_examples, out_rows, out_cols, out_channels), dtype=X.dtype)
        for i in range(out_rows):
            for j in range(out_cols):
```

```
                i_start, j_start = i*self.stride, j*self.stride
                i_end, j_end = i_start+kernel_height, j_start+kernel_width
                X_slice = X_pad[:,i_start:i_end,j_start:j_end,:]
                np.einsum('ijcn, bijc->bn', W, X_slice, out=(Z[:,i,j,:]))
        # X_pad: (16, 18, 18, 32)    W: (3, 3, 3, 32)     b: (1, 32)     Z: (16, 16, 16, 32)
        Z += b
        out = self.activation.forward(Z)
        # cache any values required for backprop
        self.cache['Z'], self.cache['X'] = Z, X
        self.cache['X_pad'] = X_pad
        self.cache['out_rows'], self.cache['out_cols'] = out_rows, out_cols
        ### END YOUR CODE ###

        return out
```

Implementation of `layers.Conv2D.backward`:

```
    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 ###
        W, b = self.parameters["W"], self.parameters["b"]
        Z, X = self.cache['Z'], self.cache['X']
        pad_h, pad_w = self.pad
        X_pad = self.cache['X_pad']
        out_rows, out_cols = self.cache['out_rows'], self.cache['out_cols']
        # perform a backward pass
        dLdZ = self.activation.backward(Z, dLdY)
        dX = np.zeros_like(X_pad)

        for i in range(out_rows):
            for j in range(out_cols):
                i_start, j_start = i*self.stride, j*self.stride
                i_end, j_end = i_start+W.shape[0], j_start+W.shape[1]
                X_slice = X_pad[:,i_start:i_end,j_start:j_end,:]
                self.gradients['W'] += np.einsum('bn, bijc->ijcn', dLdZ[:,i,j,:], X_slice)
                dX[:,i_start:i_end,j_start:j_end,:] += np.einsum('bn, ijcn->bijc', dLdZ[:,i,j,
```

```python
    self.gradients['b'] = np.sum(dLdZ, axis=(0, 1, 2)).reshape(1, -1)
    dX = dX[:, pad_h:pad_h+X.shape[1], pad_w:pad_w+X.shape[2],:]    # Remove padding
    ### END YOUR CODE ###

    return dX
```

Implementation of `layers.Pool2D.forward`:

```python
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 ###
    kernel_height, kernel_width = self.kernel_shape
    batch_size, in_rows, in_cols, channels = X.shape
    pad_h, pad_w = self.pad
    out_rows = (in_rows + 2*pad_h - kernel_height) // self.stride + 1
    out_cols = (in_cols + 2*pad_w - kernel_width) // self.stride + 1

    X_pad = np.pad(X, pad_width=((0, 0), (pad_h, pad_h), (pad_w, pad_w), (0, 0)), mode='co
    X_pool = np.zeros((batch_size, out_rows, out_cols, channels))
    # implement the forward pass
    for i in range(out_rows):
        for j in range(out_cols):
            i_start, j_start = i*self.stride, j*self.stride
            i_end, j_end = i_start+kernel_height, j_start+kernel_width
            X_pool[:,i,j,:] = self.pool_fn(X_pad[:,i_start:i_end,j_start:j_end,:], axis=(1
    # cache any values required for backpro
    self.cache['X_pad'], self.cache['X_shape'] = X_pad, X.shape
    ### END YOUR CODE ###

    return X_pool
```

Implementation of `layers.Pool2D.backward`:

```python
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 ###
        ker_h, ker_w = self.kernel_shape
        X_pad = self.cache['X_pad']
        batch_size, in_rows, in_cols, channels = self.cache['X_shape']
        batch_size, out_rows, out_cols, channels = dLdY.shape
        pad_h, pad_w = self.pad
        dX = np.zeros_like(X_pad)
        # perform a backward pass
        for i in range(out_rows):
            for j in range(out_cols):
                i_start, j_start = i*self.stride, j*self.stride
                i_end, j_end = i_start+ker_h, j_start+ker_w
                if self.mode == 'max':
                    X_i = X_pad[:,i_start:i_end,j_start:j_end,:]    # Partition of X_pad
                    X_i_flat = X_i.reshape(batch_size, ker_h*ker_w, channels)   # Flatten spat
                    X_i_indices = self.arg_pool_fn(X_i_flat, axis=1)    # Find the argmax of t
                    batch_idx, channel_idx = np.indices((batch_size, channels))
                    # Create a mask as same shape as X_i_flat
                    mask = np.zeros_like(X_i_flat)
                    mask[batch_idx, X_i_indices, channel_idx] = 1     # Set max position to be
                    mask = mask.reshape(batch_size, ker_h, ker_w, channels)    # Reshape back
                    dX[:,i_start:i_end,j_start:j_end,:] += mask * dLdY[:,i:i+1,j:j+1,:]
                else:
                    dX[:,i_start:i_end,j_start:j_end,:] += dLdY[:,i:i+1,j:j+1,:] / (ker_h*ker_

        dX = dX[:, pad_h:pad_h+in_rows, pad_w:pad_w+in_cols, :]    # Remove padding
        ### END YOUR CODE ###

        return dX
```

### 0.0.1 Activation Function Implementations:

Implementation of `activations.Linear`:

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

```python
    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
        """
        return dY
```

Implementation of `activations.Sigmoid`:

```python
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 1 / (1 + np.exp(-Z))
```

```python
    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 ###
        sigmoid = self.forward(Z)
        return dY * (sigmoid * (1-sigmoid))
```

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(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.array(Z, copy=True)
dZ[Z<0], dZ[Z>=0] = 0, 1
return dY * dZ
```