Name: _____

CNN Backpropagation

4. (10 points) Conne von Lucien has many pictures from her trip to Flatland and wants to determine which ones have her in the image. All of the pictures are arrays of size 4x1, with array values of either 0 or 1. Conne looks like the vector [1, 0, 1] in one dimension, so if a picture contains the pattern [1, 0, 1] anywhere inside it, it should be classified as a positive example, otherwise as a negative example.

   Fortunately, you learned about CNNs and have helped Conne by designing the following network architecture with three layers:

   1. A convolutional layer with one filter W that is size 3x1, and stride 1, and a single bias w_0 (where the output pixel corresponds to the input pixel that the filter is centered on). Input values of 0 should be assumed beyond the boundaries of the input.
   2. A max-pooling layer P with size 2x1 and stride 2.
   3. A fully connected layer $\sigma(\cdot)$ with a single output unit having a sigmoidal activation function.

   (a) What is the shape of the output of each layer?

   Solution: 4 x 1

   Solution: 2 x 1

   Solution: 1 x 1, scalar

   (b) What loss function is most appropriate here, especially if you want your neural network package to be useful with few modifications, to other Flatland visitors (who may appear as longer vectors)?

   √ **A. NLL loss**

   ◯ B. Hinge loss

   ◯ C. Quadratic loss

(c) We can express the loss function as $L(\sigma(P), y)$ where P is the output from the max pooling layer of the CNN and y is the true label for the input. Given $\frac{dL}{dP}$, derive the update rule for w_1 if the filter is composed of $W = [w_1, w_2, w_3]^T$ with bias w_0, and step size is η.

Solution:

Consider Z to be the outputs of layer 1, $Z = [z_1, z_2, z_3, z_4]^T$.

$$
\begin{aligned}
z_1 &= w_1 \cdot 0 + w_2 x_1 + w_3 x_2 + w_0 \\
z_2 &= w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0 \\
z_3 &= w_1 x_2 + w_2 x_3 + w_3 x_4 + w_0 \\
z_4 &= w_1 x_3 + w_2 x_4 + w_3 \cdot 0 + w_0
\end{aligned}
$$

$$
\begin{aligned}
P &= [p_1, p_2]^T \\
p_1 &= \max(z_1, z_2) \\
\frac{dp_1}{dw_1} &= 0 \text{ if } z_1 > z_2 \text{ else } x_1 \\
p_2 &= \max(z_3, z_4) \\
\frac{dp_2}{dw_1} &= x_2 \text{ if } z_3 > z_4 \text{ else } x_3 \\
\frac{dP}{dw_1} &= \left[\frac{dp_1}{dw_1}, \frac{dp_2}{dw_1}\right]^T
\end{aligned}
$$

$$
w_1 := w_1 - \eta \frac{dL}{dP}^T \frac{dP}{dw_1}
$$

(d) Given $\frac{dL}{dP}$, provide the update rule for w_0, the bias to the filter.

Solution:

$$
\frac{dP}{dw_0} = [1, 1]^T
$$

$$
w_0 := w_0 - \eta \frac{dL}{dP}^T \frac{dP}{dw_0}
$$

(e) Conne decides to use the neural network code as written by a 6.036 student for the 6.036 homework (and that actually was a correct implementation) to train her CNN using SGD. The `sgd` procedure may be called multiple times from elsewhere (e.g., to implement multiple epochs of SGD). Conne thinks she has a better `sgd` python procedure than that given in the package; her code is:

```
1  def sgd(nn, X, Y, iters=100, lrate=0.005):
2      D, N = X.shape
3      sum_loss = 0
4      for k in range(iters):
5          Xt = X[:, k:k+1]
6          Yt = Y[:, k:k+1]
7          Ypred = nn.forward(Xt)
8          sum_loss += nn.loss.forward(Ypred, Yt)
9          err = nn.loss.backward()
10         nn.backward(err)
11         nn.sgd_step(lrate)
```

Here, `nn` is an instance of the `Sequential` class implementing the CNN. She knows from the unit tests that the `nn` routines function properly. In particular, `nn.forward` properly computes the predicted outputs `Ypred` from input data `Xt`, `nn.loss.forward` also properly computes the forward loss, `nn.loss.backward` properly computes the backward loss, `nn.backward` properly computes the backward gradients, and `nn.sgd_step` properly applies an SGD update step with the specified learning rate `lrate`. And the N sets of dimension D input data `X`, and labels `Y` are known to be correct.

However, Conne's procedure consistently gives poor results (and occasionally throws errors), compared with the 6.036 student's correct SGD routine, when run with identical arguments.

Why? Specify the line(s) which have errors, and describe how the code should be improved to do as well as the correct implementation of the 6.036 student:

Solution: Lines 5 and 6

Solution:
The SGD algorithm needs a *random* data point to be selected for the gradient computation. Thus, the `Xt` and `Yt` assignments should draw from a randomly chosen `j`, e.g.:

```
1      for k in range(iters):
2          j = np.random.randint(N)
3          Xt = X[:, j:j+1]
4          Yt = Y[:, j:j+1]
5      ...
```

Note that Conne's code may throw errors when `iters` $\geq N$.