Author: Benjamin Smidt
Created: September 19, 2022
Last Updated: September 19, 2022

Assignment 2: Full Connected Networks

Note to reader.

This is my work for assignment three of Michigan's course EECS 498: Deep Learning for Computer Vision. This document is meant to be used as a reference, explanation, and resource for the assignment, not necessarily a comprehensive overview of Neural Networks. Furthermore, this document is well-informed and thoroughly researched but may not be perfect. If there's a typo or a correction needs to be made, feel free to email me at benjamin.smidt@utexas.edu so I can fix it. Thank you! I hope you find this document helpful.

1 Functions

1.1 Linear.Forward

The *Forward* function, implemented in the *Linear* class is simple. We aren't doing anything different from the matrix multiplication and bias computed in the two layer neural network we built last assignment. See A2: Two Layer Neural Net for an in depth explanation of how this matrix multiplication was computed.

1.2 Linear.Backward

The *Backward* function, implemented in the *Linear* class certainly isn't as simply as the forward function. However, I explained how to compute the gradient of a matrix multiplication for x, w, and b in great length in the last assignment A2: Two Layer Neural Net.

1.3 ReLU.Forward

Just look up the documentation for torch.clamp() if you're confused. It's just setting every value less than zero equal to zero and leaving positive values alone.

1.4 ReLU.Backward

This is getting repetitive. Again, see A2: Two Layer Neural Net for information on ReLU. I explain, in depth, both the forward and backward pass of ReLU including the gradient and how to compute it.

1.5 Linear-ReLU.forward and Linear-ReLU.backward

These functions are just utilizing the functions we just created. Although, it's worth noting how beautiful the modularity of this code truly is. With no extra work we were able to easily define a common function. Obviously, I will not be explaining object oriented programming (OOP). That is literally a course in and of itself. However, I did just want to stop for a second and marvel at the beauty that OOP has produced here.

1.6 Softmax and SVM

Since we spent a LOT of time deriving, explaining, and programming the SVM and Softmax algorithms in assigment 2, they were nice enough to gift us this implementation. To be clear, the following code is NOT my own. It is freely available from the EECS 498 Course Website in assignment 3. If you'd like to see my implementation (which is almost certainly slightly worse), you can check out A2: Softmax or A2: Multiclass SVM.

SVM-Loss-Gradient

```
def svm_loss(x, y):
    """
    Computes the loss and gradient using for multiclass SVM classification.
    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C
    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    N = x.shape[0]
    correct_class_scores = x[torch.arange(N), y]
    margins = (x - correct_class_scores[:, None] + 1.0).clamp(min=0)
    margins[torch.arange(N), y] = 0.
    loss = margins.sum() / N
    num_pos = (margins > 0).sum(dim=1)
    dx = torch.zeros_like(x)
    dx[margins > 0] = 1.
    dx[torch.arange(N), y] -= num_pos.to(dx.dtype)
    dx /= N
    return loss, dx
```

Let's quickly review how this code computes SVM loss.

$$L_i = \sum_{j \neq y_i}^{C} max(0, x_j - x_{y_i} + \Delta) \tag{1}$$

First, we get the correct class scores. Then we compute the margins by subtracting each value from the correct score in it's example, adding Δ (=1), and finally taking the max of that value and zero. Remember, this has the interpretation of SVM wanting the difference between the correct class score and every other score to be $\geq \Delta$. Lastly, we set all the correct class scores equal to zero and average the losses over each example.

The derivatives of SVM with respect to x_j and x_{y_i} are as follows.

$$\nabla_{x_{y_i}} L_i = -\sum_{j \neq y_i}^{C} \mathbb{1}(x_j - x_{y_i} + \Delta > 0) \tag{2}$$

$$\nabla_{x_j} L_i = \sum_{j \neq y_i}^{C} \mathbb{1}(x_j - x_{y_i} + \Delta > 0) \tag{3}$$

This is somewhat remiscient of the ReLU function since we have to deal with the $max()$ function. Regarding code though, for each example we simply sum the number of terms with a margin greater than zero. Then we set the gradient value at a given example N equal to this value, giving us the gradient with respect to x_{y_i}. For the gradient with respect to x_j, we simply set all the incorrect scores index values to 1 if their margin exceeds 0 and (leave it at) zero otherwise. Finally we divide all the values by N since we average the values over N examples (if we summed them then we wouldn't divide all the values by N).

Softmax-Loss-Gradient

```
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.
    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for
      the jth class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label
      for x[i] and 0 <= y[i] < C
    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """
    shifted_logits = x - x.max(dim=1, keepdim=True).values
    Z = shifted_logits.exp().sum(dim=1, keepdim=True)
    log_probs = shifted_logits - Z.log()
    probs = log_probs.exp()
    N = x.shape[0]
    loss = (-1.0 / N) * log_probs[torch.arange(N), y].sum()
    dx = probs.clone()
    dx[torch.arange(N), y] -= 1
    dx /= N
    return loss, dx
```

For softmax, our loss is:

$$L_i = -log(\frac{e^{x_{y_i}}}{\sum_{j=1}^{C} e^{x_j}}) = -x_{y_i} + log(\sum_{j=1}^{C} e^{x_j}) \tag{4}$$

In the code, we first normalize the everything to improve numerical stability. We can shift everything by any constant K and the loss won't change.

$$\frac{Ke^{x_{y_i}}}{K\sum_{j=1}^{C} e^{x_j}} = \frac{e^{x_{y_i}+logK}}{\sum_{j=1}^{C} e^{x_j+logK}} \tag{5}$$

For simplicity, we often choose $logK$ to be the negative of the max value in each example e^x. This prevents any single e^{x_j} value from becoming too large and subsequently producing a NaN or infinity.

Anyways, we compute the loss function by adding the correct class score values to the log of the sum of the scores in each example. Then we multiply by negative 1 and average over N examples. Personall, I like to distribut the negative 1 instead of multiplying at the end as seen in my loss function equation above, but it's all the same at the end of the day.

Lasly, here are the derivatives.

$$\frac{\partial L_i}{\partial x_{y_i}} = \frac{e^{x_{y_i}}}{\sum_{j=1}^{C} e^{x_j}} - 1 \tag{6}$$

$$\frac{\partial L_i}{\partial x_j} = \frac{e^{x_j}}{\sum_{j=1}^{C} e^{x_j}} \tag{7}$$

To compute the gradient we simply reuse the *probs* variable which stores all the values of both gradients we need. We simply alter the correct scores indices by subtracting 1 and finish by dividing by N since we average the loss over N examples.

1.7 TwoLayerNet Class

2 References