University of Michigan
EECS 442: Computer Vision
Fall 2022. Instructor: Andrew Owens.

Problem Set 4: Backpropagation

Posted: Wednesday, September 27, 2023 **Due:** Wednesday, October 4, 2023
Submission instructions:

- Canvas:
  - Colab notebook (`.ipynb`) containing solutions and visualizations for 4.1(b), 4.1(c) and 4.2. Before submitting, please make sure to rename the file to `<uniqname>_<umid>.ipynb`.
- Gradescope
  - Written solutions for 4.1(a)
  - `.pdf` version of Colab notebook (conversion instructions here). If this instruction does not work, look into this additional method. For your convenience, we have included the PDF conversion script at the end of the notebook.

The starter code can be found here.

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Problem 4.1 *Understanding backpropagation*

Recall that we can represent a neural network as a computation graph, which allows us to compute its gradients in a systematic way. The following diagram is an example of the equation $f(x, y, z) = (x + y)z$. The corresponding code for the forward and backward of this diagram is also shown below.

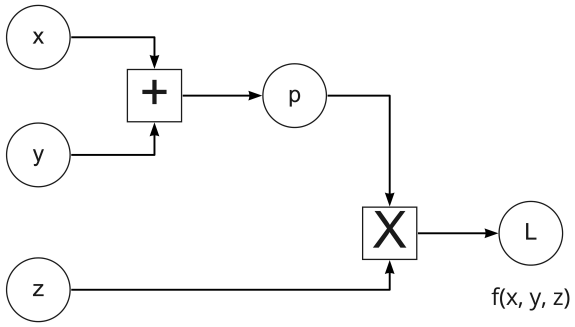

Figure 1: Computation graph for $f(x, y, z) = (x + y)z$

```python
def f(x, y, z):
    ###forward pass###
    p = x + y
    L = p * z

    ###backward pass###
    grad_L = 1
    grad_z = grad_L * p
    grad_p = grad_L * z
    grad_x = grad_p * 1
    grad_y = grad_p * 1
    return L, (grad_x, grad_y, grad_z)
```

Sometimes one piece of data is used as input to multiple operations in a computation graph. In such cases, you can modify the graph to include an explicit copy operator that returns multiple copies of its input to make logic cleaner. You can compute separate gradients for all the copies at first. When you meet the copy operator, just take the sum of separate gradients. The computation graph of the equation $f(x, y, z) = (x + y)(y + z)$ is shown in Figure 2. The modified version of computation graph with copy node is shown in Figure 3. We also include the code for this computation graph with copy node.

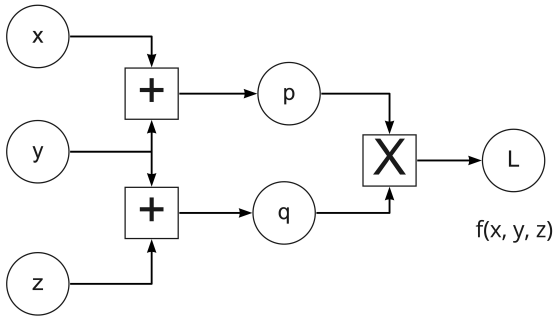

Figure 2: Computation graph for $f(x, y, z) = (x + y)(y + z)$

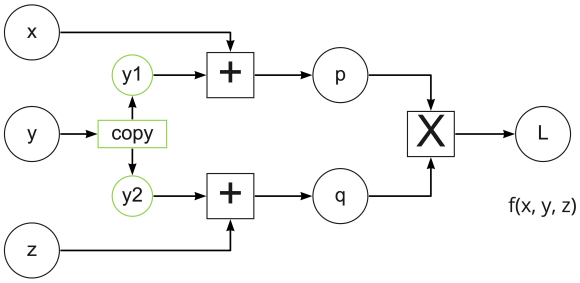

Figure 3: Computation graph with copy node

```python
def f(x, y, z):
    ###forward pass###
    y1 = y
    y2 = y
    p = x + y1
    q = z + y2
    L = p * q

    ###backward pass###
    grad_L = 1
    grad_p = grad_L * q
```

```
    grad_q = grad_L * p
    grad_x = grad_p * 1
    grad_y1 = grad_p * 1
    grad_z = grad_q * 1
    grad_y2 = grad_q * 1
    grad_y = grad_y1 + grad_y2
    return L, (grad_x, grad_y, grad_z)
```

(a) **(1 point)** Given the input $\vec{x} = [x_0, x_1, x_2], \vec{w} = [w_0, w_1, w_2, w_3]$, draw a computation graph for $f(\vec{x}, \vec{w}) = \frac{1}{1+\exp(-(w_0 x_0 + w_1/x_1 - w_2 x_2 + w_3))}$.

Note: Please use the following operations: $+, \times, -, +1, \times(-1), \exp, \frac{1}{x}$.

(b) **(1 point)** Please implement the code for forward and backward pass of computation graph in (a).

(c) **(2 point)** Please implement the code for forward and backward pass of computation graph shown below.

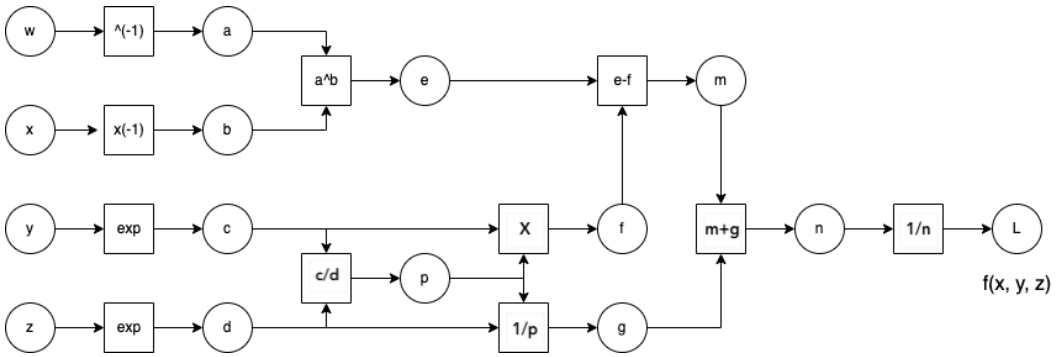

Figure 4: Computation graph for problem 4.1 (c).

Problem 4.2 *Multi-layer perceptron*

In this problem, you'll train a two-layer neural network (a multi-layer perceprton) to recognize objects in tiny images. The model and codebase will be very similar to the linear classifier that you trained in PS3. However, instead of providing you with the formula for the gradients, you will calculate them yourself using backpropagation.

Our network will have two fully connected layers (i.e. linear layers), and a softmax layer to perform classification. As in PS3, we'll train the network to minimize cross-entropy loss. The network uses a ReLU nonlinearity after the first fully connected layer. In other words, the network has the following architecture:
1) input, 2) fully connected layer, 3) ReLU, 4) fully connected layer, 5) softmax.

More concretely, for an image $\mathbf{x}$, we compute the unnormalized class probability (scores), $\mathbf{s} = (s_1, s_2, ..., s_C)$, as follows:

$$\mathbf{s} = \mathbf{W}_2 \text{ relu}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2, \tag{1}$$

where C is the total number of classes and $\mathbf{W}_i, \mathbf{b}_i$ are the parameters of the fully connected layers.

The softmax loss, L_i, for a single image with label y, can be calculated as,

$$L_i(\mathbf{s}, y) = -\log \frac{e^{s_y - \max_k(s_k)}}{\sum_{j=1}^{C} e^{s_j - \max_k(s_k)}} \tag{2}$$

This softmax loss has been provided for you already in part (a).

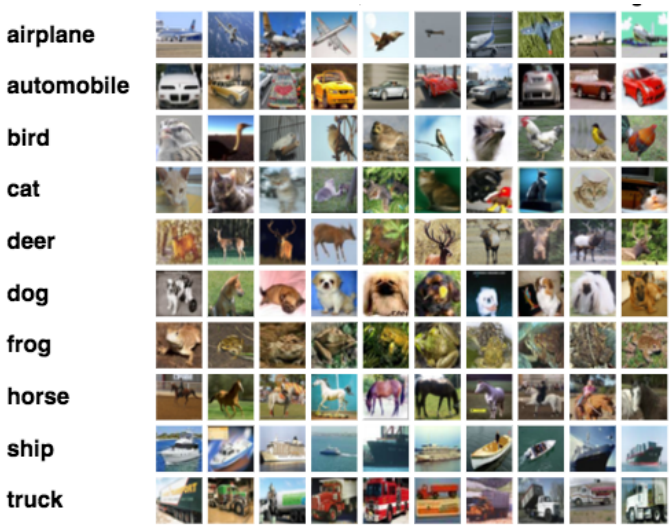

Figure 5: The CIFAR-10 dataset, which we'll be classifying in this problem. [1, 2]

The outputs of the second fully-connected layer are the scores for each class. You should *not* use a deep learning library (e.g., PyTorch) for this problem: instead, you will implement it from scratch.

(a) **(2 points)** Implement the fully connected and ReLU layers. Softmax layer has already been implemented in the provided code.

Layer reference:

- Fully connected layer:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{3}$$

- ReLU:

$$y = \begin{cases} x, & x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

(b) **(1 point)** Finish implementing the SoftmaxClassifier class. The structure has been mentioned above.

Hint: It might be helpful to use the function `np.random.normal`.

(c) **(1 point)** Set the model hyperparameters (learning_rate, lr_decay, batch_size) by yourself and run the given code. You will get full points if you obtain at least 45% accuracy on the test set.

(d) **(1 point)** Improve your optimization gradient descent method by adding *momentum*. Implement `SGD_Momentum` function and set up the remaining model hyper-parameters same as what you used for (c), then run the given code.

Hint: Recall that this learning procedure has the following update rule.

$$\mathbf{v}' \leftarrow \beta\mathbf{v} + \eta\nabla_\theta\mathcal{L}(\theta)$$
$$\theta' \leftarrow \theta - \mathbf{v}'$$

where η is the learning rate, β is a scalar in range $[0, 1)$, θ are the network parameters, $\nabla_\theta\mathcal{L}(\theta)$ is the gradient of the loss with respect to the parameters, η is the learning rate, and $\mathbf{v}$ is velocity vector (initialized as $\vec{0}$).

(e) **(0.5 point)** Run the code provided and print out test accuracy in Colab Notebook. The test accuracy of SGD and SGD Momentum model should be at least 45%.

(f) **(0.5 point)** Plot and compare the training and validation accuracy, using 1) gradient descent alone, and 2) SGD Momentum. Which model trains more quickly? Is the ultimate validation accuracy different? Report your observation in Colab.

(g) (**advanced, optional**) Add L2 regularization, also known as *weight decay*. Apply the regularization only to the weight matrices in the fully connected layers (not to the biases). Your code should be added to the `SoftmaxClassifier` class from 4.2(b). Specifically, add a loss:

$$\mathcal{L}_{reg} = \lambda \left(\|\mathbf{W}_1\|^2 + \|\mathbf{W}_2\|^2 \right), \tag{5}$$

to the parameters θ, where λ is a constant that represents the relative importance of the regularization. For this problem, set $\lambda = 0.01$. Your test accuracy should be at least 46% if implemented correctly.

(h) (**advanced, optional**) Train the model after adding the L2 regularization on the weight matrices (with the same training setup as 4.2(d)) and report the test accuracy.

(i) (**advanced, optional**) Plot and compare the training and validation accuracy for SoftmaxClassifier with and without L2 regularization. How does the gap between training accuracy and validation accuracy change after adding L2 regularization?

(j) (**advanced, optional**) Choose better hyperparameters, and run the training code. You will get full points if you obtain at least 50% test accuracy (the higher accuracy is due to the improved performance from regularization).

Acknowledgements. Part of the homework and the starter code are taken from a previous EECS 442 class taught by David Fouhey and Justin Johnson, which itself was adapted from CS231n at Stanford University by Fei-Fei Li, Justin Johnson and Serena Yeung. Please feel free to similarly re-use our problems while similarly crediting us.

References

[1] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. *University of Toronto*, 2009.

[2] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 2008.