CS 189 Introduction to Machine Learning
Summer 2019 Marc Khoury & Brijen Thananjeyan HW6

Due: Monday, August 5 at 11:59 pm

Deliverables:

1. Submit your predictions for the test sets to Kaggle as early as possible. Include your Kaggle scores in your write-up (see below). The Kaggle competition for this assignment can be found at

   - `https://www.kaggle.com/t/636053ea2f674c13be4e9d3978933aea`

2. Submit a PDF of your homework, **with an appendix listing all your code**, to the Gradescope assignment entitled "Homework 6 Write-Up". In addition, please include, as your solutions to each coding problem, the specific subset of code relevant to that part of the problem. You may typeset your homework in LaTeX or Word (submit PDF format, **not** .doc/.docx format) or submit neatly handwritten and scanned solutions. **Please start each question on a new page.** If there are graphs, include those graphs in the correct sections. **Do not** put them in an appendix. We need each solution to be self-contained on pages of its own.

   - In your write-up, please state with whom you worked on the homework.
   - In your write-up, please copy the following statement and sign your signature next to it. (Mac Preview and FoxIt PDF Reader, among others, have tools to let you sign a PDF file.) We want to make it *extra* clear so that no one inadvertently cheats.

     "I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted."

3. Submit all the code needed to reproduce your results to the Gradescope assignment entitled "Homework 6 Code". Yes, you must submit your code twice: in your PDF write-up following the directions as described above so the readers can easily read it, and once in compilable/interpretable form so the readers can easily run it. Do **NOT** include any data files we provided. Please include a short file named README listing your name, student ID, and instructions on how to reproduce your results. Please take care that your code doesn't take up inordinate amounts of time or memory. If your code cannot be executed, your solution cannot be verified.

1 Modular Fully-Connected Neural Networks

First, we will establish some notation for this problem. We define

$$h_{i+1} = \sigma(z_i) = \sigma(W_i h_i + b_i).$$

In this equation, W_i is an $n_{i+1} \times n_i$ matrix that maps the input h_i of dimension n_i to a vector of dimension n_{i+1}, where n_{i+1} is the size of layer $i+1$. The vector b_i is the bias vector added after the matrix multiplication, and σ is the nonlinear function applied element-wise to the result of the matrix multiplication and addition. $z_i = W_i h_i + b_i$ is a shorthand for the intermediate result within layer i before applying the activation function σ. Each layer is computed sequentially where the output of one layer is used as the input to the next. To compute the derivatives with respect to the weights W_i and the biases b_i of each layer, we use the chain rule starting with the output of the network and propagate backwards through the layers, which is where the backprop algorithm gets its name.

In this problem, we will implement fully-connected networks with a modular approach. This means different layer types are implemented individually, which can then be combined into models with different architectures. This enables code re-use, quick implementation of new networks and easy modification of existing networks.

1.1 Layer Implementations

Each layer's implementation will have two defining functions:

1. `forward` This function has as input the output h_i from the previous layer, and any relevant parameters, such as the weights W_i and bias b_i. It returns an output h_{i+1} and a `cache` object that stores intermediate values needed to compute gradients in the backward pass.

```
def forward(h, w):
    """ example forward function skeleton code with h: inputs, w: weights"""
    # Do computations...
    z = # Some intermediate output
    # Do more computations...
    out = # the output
    cache = (h, w, z, out) # Values needed for gradient computation
    return out, cache
```

2. `backward` This function has input: upstream derivatives and the `cache` object. It returns the local gradients with respect to the inputs and weights.

```
def backward(dout, cache):
    """ example backward function skeleton code with dout: derivative of loss with respect to outputs and
    ↪  cache from the forward pass """
    # Unpack cache
    h, w, z, out = cache
    # Use values in cache, along with dout to compute derivatives
    dh = # Derivative of loss with respect to a
    dw = # Derivative of loss with respect to w
    return dh, dw
```

Your layer implementations should go into the provided `layers.py` script. The code is clearly marked with TODO statements indicating what to implement and where.

When implementing a new layer, it is important to manually verify correctness of the forward and backward passes. Typically, the gradients in the backward pass are checked against numerical gradients. We provide a test script `starter_code.ipynb` for you to use to check each of layer implementations, which handles the gradient checking. Please see the comments of the code for how to appropriately use this script.

In your write-up, provide the following for each layer you've implemented.

1. Listings of (the relevant parts of) your code.
2. Written justification/derivation for the derivatives in your backward pass for *all* the layers that you implement.
3. The output of running numerical gradient checking.
4. Answers to any inline questions.

1.1.1 Fully-Connected (fc) Layer

In `layers.py`, you are to implement the forward and backward functions for the fully-connected layer. The fully-connected layer performs an affine transformation of the input: $\text{fc}(h) = Wa + b$. Write your fc layer for a general input h that contains a mini-batch of B examples, each of which is of shape $(d_1, \cdots, d_k)$.

1.1.2 Activation Functions

In `layers.py`, implement the forward and backward passes for the ReLU activation function

$$\sigma_{\text{ReLU}}(\gamma) = \begin{cases} 0 & \gamma < 0 \\ \gamma & \text{otherwise} \end{cases}$$

Note that the activation function is applied element-wise to a vector input.

There are many other activation functions besides ReLU, and each activation function has its advantages and disadvantages. One issue commonly seen with activation functions is vanishing gradients, i.e., getting zero (or close to zero) gradient flow during backpropagation. Which of activation functions (among: linear, ReLU, tanh, sigmoid) experience this problem? Why? What types of one-dimensional inputs would lead to this behavior?

1.1.3 Softmax Loss

In subsequent parts of this problem, we will train a network to classify the digits in MNIST. Therefore, we will need the softmax loss, which is comprised of the softmax activation followed by the cross-entropy loss. It is a minor technicality, but worth noting that the softmax is just the squashing function that enables us to apply the cross-entropy loss. Nevertheless, it is a commonly used shorthand to refer to this as the softmax loss.

The softmax function has the desirable property that it outputs a probability distribution. For this reason, many classification neural networks use the softmax. Technically, the softmax activation takes in C input

numbers and outputs C scores which represents the probabilities for the sample being in each of the possible C classes. Formally, suppose $s_1 \cdots s_C$ are the C input scores; the outputs of the softmax activations are

$$t_i = \frac{e^{s_i}}{\sum_{k=1}^{C} e^{s_k}}$$

for $i \in [1, C]$. The cross-entropy loss is

$$E = -\log t_c,$$

where c is the correct label for the current example.

Since the loss is the last layer within a neural network, and the backward pass of the layer is immediately calculated after the foward pass, `layers.py` merges the two steps with a single function called `softmax_loss`.

You have to be careful when you implement this loss, otherwise you will run into issues with numerical stability. Let $m = \max_{i=1}^{C} s_i$ be the max of the s_i. Then

$$E = -\log t_c = \log \frac{e^{s_c}}{\sum_{k=1}^{C} e^{s_k}} = \log \frac{e^{s_c - m}}{\sum_{k=1}^{C} e^{s_k - m}} = -(s_c - m) + \log \sum_{k=1}^{C} e^{s_k - m}.$$

We recommend using the rightmost expression to avoid numerical problems.

Finish the softmax loss in `layers.py`.

1.2 Two-layer Network

Now, you will use the layers you have written to implement a two-layer network (also referred to as a one *hidden* layer network) that classifies MNIST handwritten digits. You should implement the following network architecture: input - fc layer - ReLU activation - fc layer - softmax loss. Implement the class `FullyConnectedNet` in `fc_net.py`. Note that this class supports multi-layer networks, not just two-layer networks. You will need this functionality in the next part. To help you train your model, we have already implemented a solver, which encapsulates all the logic necessary for training. See `starter_code.ipynb` for an example of how to use the solver.

For your part, you will need to instantiate a model of your two-layer network, load your training and validation data, and use a `Solver` instance to train your model. Explore different hyperparameters including the learning rate, learning rate decay, batch size, the hidden layer size, and the weight_scale initialization for the parameters. Report the results of your exploration, including what parameters you explored and which set of parameters gave the best validation accuracy.

1.3 Multi-layer Network

Now you will implement a fully-connected network with an arbitrary number of hidden layers. Use the same code as before and try different number of layers (1 hidden layer to 4 hidden layers) as well as different number of hidden units. Include in your write-up what kinds of models you have tried, their hyperparameters, and their training and validation accuracies. Report which architecture works best.

2 Convolution and Backprop Revisited

In this problem, we will explore how image masking can help us create useful high-level features that we can use instead of raw pixel values. We will walk through how discrete 2D convolution works and how we can use the backprop algorithm to compute derivatives through this operation.

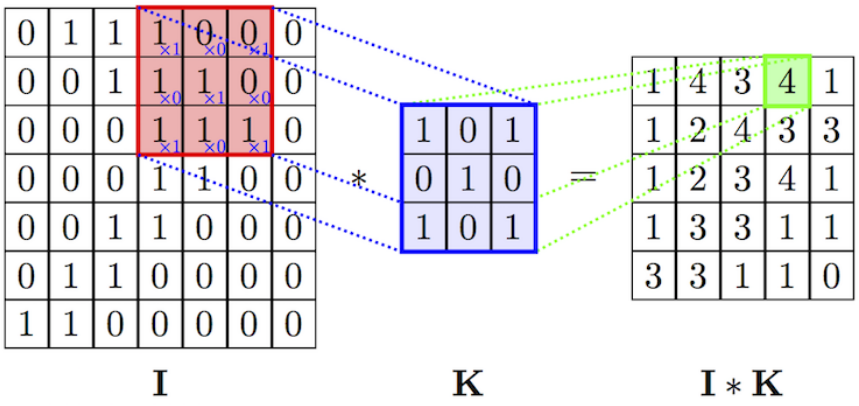

Figure 1: Figure showing an example of one convolution.

(a) To start, let's consider convolution in one dimension. Convolution can be viewed as a function that takes a signal $I[]$ and a mask $G[]$, and the discrete convolution at point t of the signal with the mask is

$$(I * G)[t] = \sum_{k=-\infty}^{\infty} I[k]G[t-k]$$

If the mask $G[]$ is nonzero in only a finite range, then the summation can be reduced to just the range in which the mask is nonzero, which makes computing a convolution on a computer possible.

As an example, we can use convolution to compute a derivative approximation with finite differences. The derivative approximation of the signal is $I'[t] \approx (I[t+1] - I[t-1])/2$. Design a mask $G[]$ such that $(I * G)[t] = I'[t]$.

(b) Convolution in two dimensions is similar to the one-dimensional case except that we have an additional dimension to sum over. If we have some image $I[x, y]$ and some mask $G[x, y]$, then the convolution at the point (x, y) is

$$(I * G)[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} I[m, n]G[x-m, y-n]$$

or equivalently,

$$(I * G)[x, y] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} G[m, n]I[x-m, y-n],$$

because convolution is commutative.

In an implementation, we'll have an image I that has three color channels I_r, I_g, I_b each of size $W \times H$ where W is the image width and H is the height. Each color channel represents the intensity of red, green, and blue for each pixel in the image. We also have a mask G with finite support. The mask also has three color channels, G_r, G_g, G_b, and we represent these as a $w \times h$ matrix where w and h are the width and height of the mask. (Note that usually $w \ll W$ and $h \ll H$.) The output $(I * G)[x, y]$ at point (x, y) is

$$(I * G)[x, y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{r,g,b\}} I_c[x+a, y+b] \cdot G_c[a, b]$$

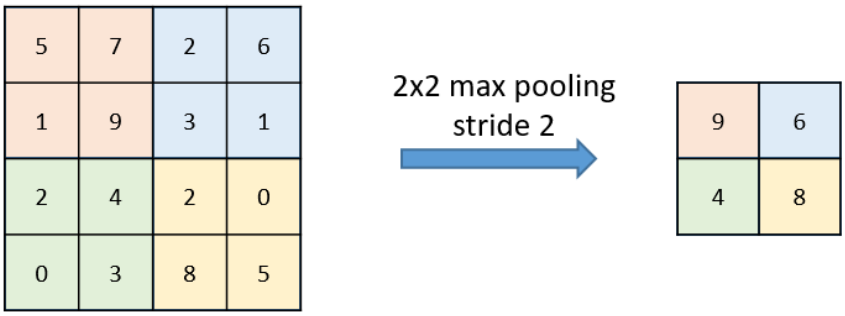

Figure 2: Figure showing an example of one maxpooling.

In this case, the size of the output will be $(1 + W - w) \times (1 + H - h)$, and we evaluate the convolution only within the image I. (For this problem we will not concern ourselves with how to compute the convolution along the boundary of the image.) To reduce the dimension of the output, we can do a strided convolution in which we shift the convolutional mask by s positions instead of a single position, along the image. The resulting output will have size $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$.

Write pseudocode to compute the convolution of an image I with a set of masks G and a stride of s. Hint: to save yourself from writing low-level loops, you may use the operator $*$ for element-wise multiplication of two matrices (which is not the same as matrix multiplication) and invent other notation when convenient for simple operations like summing all the elements in the matrix.

(c) Masks can be used to identify different types of features in an image such as edges or corners. Design a mask G that outputs a large value for vertically oriented edges in image I. By "edge," we mean a vertical line where a black rectangle borders a white rectangle. (We are not talking about a black line with white on both sides.)

(d) Although handcrafted masks can produce edge detectors and other useful features, we can also learn masks (sometimes better ones) as part of the backpropagation algorithm. These masks are often highly specific to the problem that we are solving. Learning these masks is a lot like learning weights in standard backpropagation, but because the same mask (with the same weights) is used in many different places, the chain rule is applied a little differently and we need to adjust the backpropagation algorithm accordingly. In short, during backpropagation each weight w in the mask has a partial derivative $\frac{\partial L}{\partial w}$ that receives contributions from every patch of image where w is applied.

Let L be the loss function or cost function our neural network is trying to minimize. Given the input image I, the convolution mask G, the convolution output $R = I * G$, and the partial derivative of the error with respect to each scalar in the output, $\frac{\partial L}{\partial R[i,j]}$, write an expression for the partial derivative of the loss with respect to a mask weight, $\frac{\partial L}{\partial G_c[x,y]}$, where $c \in \{r, g, b\}$. Also write an expression for the derivative of $\frac{\partial L}{\partial I_c[x,y]}$.

(e) Sometimes, the output of a convolution can be large, and we might want to reduce the dimensions of the result. A common method to reduce the dimension of an image is called max pooling. This method works similar to convolution in that we have a mask that moves around the image, but instead of multiplying the mask with a subsection of the image, we take the maximum value in the subimage. Max pooling can also be thought of as downsampling the image but keeping the largest activations for each channel from the original input. To reduce the dimension of the output, we can do a strided max pooling in which we shift the max pooling mask by s positions instead of a single position, along the

input. Given a mask size of $w \times h$, and a stride s, the output will be $\lfloor 1 + (W - w)/s \rfloor \times \lfloor 1 + (H - h)/s \rfloor$ for an input image of size $W \times H$.

Let the output of a max pooling operation be an array R. Write a simple expression for element $R[i, j]$ of the output.

(f) Explain how we can use the backprop algorithm to compute derivates through the max pooling operation. (A plain English answer will suffice; equations are optional.)