Author: Benjamin Smidt
Created: September 19, 2022
Last Updated: September 23, 2022

Assignment 3: Full Connected Networks

Note to reader.

This is my work for assignment three of Michigan's course EECS 498: Deep Learning for Computer Vision. The majority of explanations and understanding are derived from Justin Johnson's Lectures and Stanford's CS 231N Lecture Notes. This document is meant to be used as a reference, explanation, and resource for the assignment, not necessarily a comprehensive overview of Neural Networks. 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.

Contents

1 Gradient and Loss Functions **3**
- 1.1 Linear.Forward . . . 3
- 1.2 Linear.Backward . . . 3
- 1.3 ReLU.Forward . . . 3
- 1.4 ReLU.Backward . . . 3
- 1.5 Linear-ReLU.forward and Linear-ReLU.backward . . . 3
- 1.6 Softmax and SVM . . . 4
  - 1.6.1 SVM-Loss-Gradient . . . 4
  - 1.6.2 Softmax-Loss-Gradient . . . 5

2 End-to-End Neural Networks **7**
- 2.1 TwoLayerNet Class . . . 7
  - 2.1.1 Initialization . . . 7
  - 2.1.2 Loss . . . 7
  - 2.1.3 Gradient . . . 8
- 2.2 FullyConnectedNet Class . . . 8
  - 2.2.1 Initialization . . . 8
  - 2.2.2 Loss . . . 8
  - 2.2.3 Gradient . . . 9

3 Update Rules and Regularization **9**
- 3.1 SGD + Momentum . . . 9
- 3.2 RMSProp . . . 10
- 3.3 Adam . . . 11
- 3.4 Dropout . . . 11

4 References **13**

1 Gradient and Loss 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.

1.6.1 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 of the correct score 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 in our loss function (if we summed them in our loss function then we wouldn't divide by N).

1.6.2 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. Personally, I like to distribute the negative 1 instead of multiplying at the end as seen in my loss function equation above (Eq. 4), 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 score indices by subtracting 1 and finish by dividing by N since, again, we average the loss over N examples.

2 End-to-End Neural Networks

2.1 TwoLayerNet Class

2.1.1 Initialization

We use a two layer neural net class to to easily define our neural net. We begin by initilalizing a dictionary, *self.params*, and a float *self.reg*. *self.reg* is, of course, our regularization constant λ and *self.params* stores our changeable parameters in our network. Since we're creating a two later network with an architecture of fully connected layers Linear - ReLU - Linear - Softmax (same as A2), we'll need four parameters. $W1$ and $b1$ our for our first Linear operation and $W2$ and $b2$ are for our second linear operation.

We initialize each of the weight matrices $W1$ and $W2$ using a random normal distribution with mean 0 and standard deviation *weight-scale*, a user input value into the TwoLayerNet Class that defaults to $1 * 10^{-3}$

2.1.2 Loss

We can easily define our forward pass through the network using the classes we just created. In particular we call *Linear-ReLU.forward* with $W1$ and $b1$ and pass the resulting hidden matrix to *Linear.forward* with $W2$ and $b2$. Finally, we pass the outputted scores matrix to *softmax-loss*, the function defined and reviewed in the previous page, and add our regularization loss to the data loss computed by *softmax-loss*.

2.1.3 Gradient

Similar to our forward pass, we easily define our backward pass using the functions and classes we've created. The key here is really understanding what's being input and returned from our classes and what data from our forward pass (*cache* in the code) needs to be passed to our function computing the backward pass. It's pretty simply, so I'll leave it to you to see how it's implemented in the code. Everything is nicely labeled so there shouldn't be any surprises. Just don't forget to add the gradient for your regularization loss!

2.2 FullyConnectedNet Class

2.2.1 Initialization

When writing this I have NOT implemented different update rules (SGD + Momentum, RMSProp, Adam, Dropout). I'll explain those later in this document since the first implementation of FullyConnectedNet did not include implementing these various update rules.

So, for initialization, I essentially just did the exact same process as the two layer network except I generalized the hidden layers. I had to hard code the first and last layers since the first layer is the only one that uses *input-dim* (number of dimensions for each example in X) and the last layer is the only one that uses *num-classes*. There is some python specific syntax I used for storing an arbitrary number of parameters in *self.params*. You should be able to easily look up Python documentation for what they do though. I'm just iteratively labelling W and b with a number to keep track of the parameters, nothing too exciting.

2.2.2 Loss

Things can get a little hairy here but the idea is still the exact same as the two layer network we just implemented. I iteratively pass the output of the *Linear-ReLU.forward()* function as the input of the next *Linear-ReLU.forward()* function until I reach the very last layer. To compute the final *scores* matrix we need to remember to use *Linear* without *ReLU*. We can then easily pass the scores to softmax which kindly returns our loss and *d-scores* to begin backpropagating.

Some easily forgetten things I need to mention. First, we need to store all the *caches* or "weights" that we're computing throughout the network. We will need them for backpropagation. Second, it's very easy to forget to add regularization. This must be done iteratively as well and is easily computed with the for loop we're already using in our forward propagation.

2.2.3 Gradient

Finally, we get to backprop. I've yet to mention this for previous assignments but I do want to point out the if-statement that catches whether we're in "train" or "test" mode. Obviously, if we're in test mode we can't do backprop because we have no "correct answers" with which to compute our loss function. It's an important detail that can be easily missed, so yeah.

Again, we iteratively backpropagate through our network in the same manner we've done for our two layer network. The last layer (first computation) must be done first since it's the only layer using *Linear* while the rest can be computed in our for-loop using *Linear-ReLU*. As a reminder, don't forget the regularization (I'm bad about that :( ).

3 Update Rules and Regularization

Disclaimer: this section in particular was pulled from CS 231N: Training NN III. I rephrase the ideas stated in these lecture notes but I do so to facilitate my own understanding. These are not wholly my explanations as these gradient update rules can be quite tricky to interpret.

3.1 SGD + Momentum

Physics. Everything always comes back to physics. This idea is simply yet very powerful. To improve stochastic gradient descent (SGD), we can add "momentum", a fancy word for saying: the longer our gradient has been moving in the same direction along a given axis, the larger we want that step size to be along that axis. In the same vein, more momentum should make it more difficult to reverse the velocity along that axis.

Why does this work? Well, it gives more weight to the directions that are consistently updating in the same direction. By doing so, oscillating updates that you might find in vanila SGD are massively reduced. If every update

the direction along an axis is changing directions, then the momentum will push its update to be close to zero since the oscillating updates aren't getting us anywhere meaningful. Essentially, adding momentum uses the gradient history to give more weight to gradient updates that have consistently been in the same direction. This improves SGD tremendously.

Let's take a look at the code to see how we actually implement momentum.

```
v = mu * v - learning_rate * dx
x += v
```

v is our velocity and mu is a hyperparameter usually referred to as *momentum*. You can see, we're adding our current gradient update to our previous velocity, scaled by our momentum. Then we add our velocity to our current coordinate position for our loss function. I don't want to dive too much into the interpretation of this but hopefully you can see that we're using our gradient update to update our *velocity*. Said differently, we constantly iterate the direction we're "already moving" rather than using the gradient update as our direction for each update.

As a quick aside, Nesterov Momentum is a similar idea as momentum. It uses a look ahead though to try and improve the momentum update. You can find more info here.

3.2 RMSProp

Before I explain RMSProp (Root Mean Squared Propagation), it's important to mention Adagrad (Adaptive Gradient). Adagrad is a method that uses an adaptive learning rate to update our weights proposed by Duchi et al.. It does so by dividing our learning rate by the square root of the sum of the squared history of gradients.

```
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

This has the effect of reducing the learning rate when the gradient is large, which is good because we don't need a large learning rate if the gradient is already large. The problem with Adagrad (for Deep Learning) is that the factor we divide our learning rate by it is monotonically increasing, meaning we can only ever decrease the learning rate. This becomes problematic for

deep learning as Adagrad too often squashes our learning rate before we finish learning. As a quick aside, *eps* is a small constant to ensure we don't divide by 0.

An improved variant of Adagrad is RMSProp. Our dividing factor is not monotonically increasing, producing a less aggressive learning rate decay. In practice this works much better. Funnily enough, the citation for RMSProp isn't a research paper but instead lecture 6, slide 29 of Geoff Hinton's Coursera class.

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

I won't expand on this much but hopefully it isn't too difficult to see that because *cache* is multiplied by the decay rate, *cache* is not monotonically increasing. It can decrease depending on the size of *dx* and the decay rate, allowing for our learning rate increase, particularly if the gradient update is quite small (which is what we want).

3.3 Adam

Adam is the default optimization used for updating gradients. The essential idea is that it combines both momentum with the adaptive gradient ideas from Adagrad/RMSprop (Adagrad + Momentum = Adam). Here's the code below.

```
m = beta1*m + (1-beta1)*dx
mt = m / (1-beta1**t)
v = beta2*v + (1-beta2)*(dx**2)
vt = v / (1-beta2**t)
x += - learning_rate * mt / (np.sqrt(vt) + eps
```

I don't currently have time to dive deep into the guts of Adam to give a good explanation. For application purposes it's important to know that it usually includes bias correction term in the beginning and that it can be worth it to compare to Nesterov Momentum.

3.4 Dropout

Finally, the last section. As in the last sections regarding update rules, I don't have the time to become an expert to explain this stuff but I can give the general gist. It's worth noting though that topics such as dropout are still currently being explored and we don't yet fully understand why it works.

Dropout is a technique that aims to limit a model from overfitting. We want our deep learning network to learn from out training network but not learn its unique attributes such that it doesn't perform well on new data. A few methods have been introduced to prevent overfitting: regularization and smaller models (less capacity and/or less depth). However, we know larger networks tend to perform better. Dropout allows us to take advantage of the larger capacity and/or depth of a network but stills preventing overfitting. Thus, we get better predictions from our larger network without worrying about overfitting the data.

The technique is simple. During each pass through the network we randomly choose (with some probability p) nodes in each layer and set them to zero. This effectively doesn't allow the network to learn from those nodes for that particular batch, leaving the network to find more paths to come to the same solution. Since the network can't hyperoptimize any particular node or path for a given input, we force it generalize the weights more which prevents overfitting.

```
Dropout.forward()
    dropout_mask = (torch.rand(x.shape) > p) / p
    x = x * mask

Dropout.backward()
    dx = dout * dropout_mask
```

There are some technicalities when performing dropout. We don't perform dropout at test time since we want the whole network available for new predictions. However, this can create some slight differences in the expected output of the network. The expected output of a given node during training is px since we trained it on $px + (1 - p)0$. The $1 - p$ is our dropout rate, which corresponds to us setting the node to zero (so the expected value of that term is zero). During test time though, our expected value is p.

This is problematic. We need our test and train outputs to have the same expected value. The solution? A fancy term called "inverted dropout" which

is super simple. During training, we divide our node value by probability p. This scales up the expected output back to x since $\frac{px+(1-p)0}{p} = x$. And yeah, that's pretty much all I have for dropout.

One last thing is that if you take a look at my script, droput doesn't perform that well using adam but performs well (or better at least) with SGD+Momentum. Actually, the validation accuracy as a whole performs considerably better with SGD+Momentum. But relative to the other two parameters, dropout performs worse with adam and better with SGD+Momentum. I'm trying to trouble shoot why this is the case, something could be wrong with my adam optimizer. Regardless though, it was worth noting.

*Update: I found the bug that was messing with my Adam implementation (it was very silly). Anyways, the results are as expected using Adam. It's interesting to note though how well Adam fit the training set compared to SGD+Momentum (difference of over 30 percent) despite SGD+Momentum having a slightly larger training accuracy.

4 References

1. CS 231N: NN Training II
2. CS 231N: NN Training III