Course: DD2424 - Assignment 3

In this assignment you will train and test k-layer networks with multiple outputs to classify images (once again) from the CIFAR-10 dataset. You will upgrade your code from Assignment 2 in two significant ways:

1. Generalize your code so that you can train and test k-layer networks.
2. Incorporate batch normalization into the k-layer network both for training and testing.

The overall structure of your code for this assignment should mimic that from **Assignment 2**. You will mainly just have to modify the functions that implement the forward and backward passes. Before the explicit instructions for the assignment, we present the mathematical details that you will need to complete the assignment. As in the previous assignment we will train our networks by minimizing a cost function, a weighted sum of the cross-entropy loss on the labelled training data and L_2 regularization of the weight matrices see equation (18) for the general form, using mini-batch gradient descent.

Background 1: *k-layer network*

The mathematical details of the first network you will implement are as follows. Given an input vector, $\mathbf{x}$, of size $d \times 1$ our classifier outputs a vector of probabilities, $\mathbf{p}$ $(K \times 1)$, for each possible output label.

for $l = 1, 2, \ldots, k-1$

$$\mathbf{s}^{(l)} = W_l \, \mathbf{x}^{(l-1)} + \mathbf{b}_l \tag{1}$$

$$\mathbf{x}^{(l)} = \max(0, \mathbf{s}^{(l)}) \tag{2}$$

and then finally

$$\mathbf{s} = W_k \, \mathbf{x}^{(k-1)} + \mathbf{b}_k \tag{3}$$

$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{4}$$

The equations for the gradient computations of the back-propagation algorithm for a k-layer network are given in Lecture 4 (you should download a recent version of the notes as multiple typos have been fixed since when the lecture was given). Note the equations in the lecture notes compute the

gradient for a mini-batch of size 1. So you will have to up-grade them to an mini-batch of arbitrary size.

Background 2: *k-layer network with Batch Normalization*

You will discover that training a network with >3 layers for our problem becomes difficult if you are not careful with your random initialization. Therefore the second part of the assignment will be devoted to implementing batch normalization to overcome this limitation. We now give the explicit mathematical details for batch normalization for a k-layer network.

At test time it is assumed that you have a pre-computed

- $\boldsymbol{\mu}^{(l)}$ - an estimated mean for the unnormalized scores $\mathbf{s}^{(l)}$ at layer l (has the same size as $\mathbf{s}^{(l)}$),
- $\mathbf{v}^{(l)}$ - the vector containing the estimated variance for each dimension of $\mathbf{s}^{(l)}$.

You then apply batch normalization with quantities with these equations:

for $l = 1, 2, \ldots, k-1$

$$\mathbf{s}^{(l)} = W_l \, \mathbf{x}^{(l-1)} + \mathbf{b}_l \tag{5}$$

$$\hat{\mathbf{s}}^{(l)} = \text{BatchNormalize}(\mathbf{s}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) \tag{6}$$

$$\mathbf{x}^{(l)} = \max(0, \hat{\mathbf{s}}^{(l)}) \tag{7}$$

and then finally

$$\mathbf{s} = W_k \, \mathbf{x}^{(k-1)} + \mathbf{b}_k \tag{8}$$

$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{9}$$

where

$$\text{BatchNormalize}(\mathbf{s}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) = \left(\text{diag}(\mathbf{v}^{(l)} + \epsilon)\right)^{-\frac{1}{2}} \left(\mathbf{s}^{(l)} - \boldsymbol{\mu}^{(l)}\right) \tag{10}$$

and $\epsilon > 0$ is a small number that is present to ensure you don't divide by 0.

Forward pass of BN for back-propagation training

During the forward pass of BN training for each mini-batch you also normalize the scores at each layer, but you compute the mean and variances of the un-normalized scores from the data in the mini-batch. In more detail assume that we have a mini-batch of data $\mathcal{B} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. At

each layer $1 \leq l \leq k-1$ you must make the following computations. Compute the un-normalized scores at the current layer l for each example in the mini-batch

$$\mathbf{s}_i^{(l)} = W_l \, \mathbf{x}_i^{(l-1)} + \mathbf{b}_l \quad \text{for } i = 1, \ldots, n \tag{11}$$

Then compute the mean and variances of these un-normalized scores

$$\boldsymbol{\mu}^{(l)} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{s}_i^{(l)} \tag{12}$$

$$v_j^{(l)} = \frac{1}{n} \sum_{i=1}^{n} \left(s_{ij}^{(l)} - \mu_j^{(l)} \right)^2 \quad \text{for } j = 1, \ldots, m_l \tag{13}$$

where m_l is the dimension of the scores at layer l. Given the computed mean and variances, we can now normalize the scores for the mini-batch and subsequently apply ReLu. So for $i = 1, \ldots, n$:

$$\hat{\mathbf{s}}_i^{(l)} = \text{BatchNormalize}(\mathbf{s}_i^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) \tag{14}$$

$$\mathbf{x}_i^{(l)} = \max(0, \hat{\mathbf{s}}_i^{(l)}) \tag{15}$$

The final layer is then applied as usual, for $i = 1, \ldots, n$:

$$\mathbf{s}_i = W_k \, \mathbf{x}_i^{(k-1)} + \mathbf{b}_k \tag{16}$$

$$\mathbf{p}_i = \text{SOFTMAX}(\mathbf{s}_i) \tag{17}$$

Backward pass of BN for back-propagation training

As we have applied score normalization during the forward pass we have to compensate for this in the backward pass of the back-propagation algorithm. As per usual let J represent the cost function for the mini-batch that is

$$J(\mathcal{B}, \lambda, \boldsymbol{\Theta}) = \frac{1}{n} \sum_{i=1}^{n} l_{\text{cross}}(\mathbf{x}_i, y_i, \boldsymbol{\Theta}) + \lambda \sum_{i=1}^{k} \|W_i\|^2 \tag{18}$$

Then the backward pass of the back-prop algorithm is defined as:

The gradient computations for our mini-batch $\mathcal{B}$ after having completed the forward pass and kept a record of $\mathbf{s}_1^{(l)}, \ldots \mathbf{s}_n^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}$ for $l = 1, \ldots, k-1$ and $\mathbf{p}_1, \ldots \mathbf{p}_n$.

- for $i = 1, \ldots, n$

$$\mathbf{g}_i = -(\mathbf{y}_i - \mathbf{p}_i)^T \tag{19}$$

- The gradients of J w.r.t. bias vector $\mathbf{b}_k$ and W_k

$$\frac{\partial J}{\partial \mathbf{b}_k} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}_i, \qquad \frac{\partial J}{\partial W_k} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}_i^T \mathbf{x}_i^{(k-1)T} + 2\lambda W_k \tag{20}$$

- Propagate the gradient vector $\mathbf{g}_i$ to the previous layer:
  for $i = 1, \ldots, n$

$$\mathbf{g}_i = \mathbf{g}_i W_k \tag{21}$$

$$\mathbf{g}_i = \mathbf{g}_i \text{ diag}(\text{Ind}(\hat{\mathbf{s}}_i^{(k-1)} > 0)) \tag{22}$$

- for $l = k-1, \ldots, 1$

  1. $\mathbf{g}_1, \ldots, \mathbf{g}_n = \text{BatchNormBackPass}(\mathbf{g}_1, \ldots, \mathbf{g}_n, \mathbf{s}_1^{(l)}, \ldots, \mathbf{s}_n^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)})$
  2. The gradient of J w.r.t. bias vector $\mathbf{b}_l$

$$\frac{\partial J}{\partial \mathbf{b}_l} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}_i \tag{23}$$

  3. Gradient of J w.r.t. weight matrix W_l

$$\frac{\partial J}{\partial W_l} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}_i^T \mathbf{x}_i^{(l-1)T} + 2\lambda W_l \tag{24}$$

  4. Propagate the gradient vector $\mathbf{g}_i$ to the previous layer (if $l > 1$).
     for $i = 1, \ldots, n$

$$\mathbf{g}_i = \mathbf{g}_i W_l \tag{25}$$

$$\mathbf{g}_i = \mathbf{g}_i \text{ diag}(\text{Ind}(\hat{\mathbf{s}}_i^{(l-1)} > 0)) \tag{26}$$

where the function `BatchNormBackPass` implements the equations given in the last slide of lecture 4. Remember that each $\mathbf{g}_i$ sent into `BatchNormBackPass` represents $\frac{\partial J}{\partial \hat{\mathbf{s}}_i^{(l)}}$ while each $\mathbf{g}_i$ returned by `BatchNormBackPass` represents $\frac{\partial J}{\partial \mathbf{s}_i^{(l)}}$.

Exponential moving average for batch means and variances

While training your network with batch normalization you should keep a exponential moving average estimate of the mean and variances for the unnormalized scores for each layer that will be used during test time. You can achieve this by setting, after each forward pass (which generates a new $\boldsymbol{\mu}^{(l)}$ and $\mathbf{v}^{(l)}$) of the mini-batch gradient descent algorithm for $l = 1, \ldots, k-1$

$$\boldsymbol{\mu}_{\text{av}}^{(l)} = \alpha \boldsymbol{\mu}_{\text{av}}^{(l)} + (1-\alpha)\boldsymbol{\mu}^{(l)} \tag{27}$$

$$\mathbf{v}_{\text{av}}^{(l)} = \alpha \mathbf{v}_{\text{av}}^{(l)} + (1-\alpha)\mathbf{v}^{(l)} \tag{28}$$

where $\alpha \in (0, 1)$ and typically $\alpha \approx .99$. You can initialize $\boldsymbol{\mu}_{\text{av}}^{(l)}$ to be equal to the $\boldsymbol{\mu}^{(l)}$ obtained from the very first mini-batch update step and similarly for $\mathbf{v}^{(l)}$.

Exercise 1: *Upgrade Assignment 2 code to train & test k-layer networks*

In Assignment 2 you wrote code to train and test a 2-layer neural network. For the first part of this assignment you should upgrade your code from Assignment 2 so that you can train and test a k-layer network. If you have a decent architecture for your code, this should not involve too much coding. You will need to refine the functions and data structures that you use

1. to store and initialize the parameters of your network,
2. to apply the network to input vectors and keep a record of the intermediary scores when you apply the network (the $\mathbf{x}^{(l)}$'s in equation (2)) (**forward pass**),
3. to compute the gradient of the cost function for a mini-batch relative to the parameters of the network using the gradient equations in the lectures notes (**backward pass**).

When you have upgraded your code you should debug the gradient computations and check them numerically as previously. You should start with a 2-layer network, then a 3-layer network and then finally a 4-layer network. You'll probably notice that the discrepancy between the analytic and the numerical gradients increases for the earlier layers as the gradient is back-propagated through the network. Re-read the relevant section of the Additional material for lecture 3 from Standford's course **Convolutional Neural Networks for Visual Recognition** to get all the tips and potential issues. But remember to make your checks initially with `lambda=0` and also with a much reduced dimensional input data to avoid numerical precision issues. You should train using the basic mini-batch gradient descent with momentum.

Once you have convinced yourself that your analytic gradient computations are bug free then you should continue with the assignment.

Exercise 2: *Can I train a 3-layer network?*

First check, with the new version of your code, you can replicate the (default) results you achieved in Assignment 2 with a 2-layer network with 50 nodes in the hidden layer. If the answer is yes, then your next task is to try and train a 3-layer network with 50 and 30 nodes in the first and second hidden layer respectively with the same learning parameters. What happens after a few epochs of training? Are you learning anything? What happens if you play around with the learning rate `eta`? What happens if you use He initialization?

What you'll find is that it is tricky to train a 3-layer network for this dataset using a random initialization of the weights and mini-batch gradient descent

with momentum. Once you have convinced yourself of this fact, you are ready to face the task of overcoming this problem by implementing *batch normalization*.

Exercise 3: *Implement batch normalization*

You have seen firsthand that training networks with more than 2-layers is difficult. In the lecture notes I told you *batch normalization* overcomes this problem. Now it's your turn to implement it.

First, consider the **forward pass** where you apply the network to the input data in a mini-batch. You will have written, for the first part of this assignment, a function that evaluates the network on a mini-batch of input data and returns the probability score and the intermediary activations (for each hidden layer) for each example in the mini-batch. In batch normalization you will need to augment your code so that it implements equations (11) - (17) (and returns the intermediary vectors needed by the backward pass). In the first version of your new function you should write it assuming the layer means and variances are computed from the mini-batch data sent into the function. You will, however, also call this function at test time and in this case it is assumed that the un-normalized scores are normalized by known pre-computed means and variances that have been estimated during training. Thus you should write a final version of the function so that it can take a variable number of inputs depending on whether you send it pre-computed means and variances or not. You can do this in `Matlab` using the `varargin` cell structure. Use the `help` command to get more details.

Note: If you store your un-normalized scores for the lth layer in the matrix `scores` of size `m` $\times$ `n` where `n` is the number of examples in the mini-batch then this `matlab` code will compute the variance for each dimension:

```
var_scores = var(scores, 0, 2);
```

The matlab function `var` computes the variances by dividing the relevant sum-of-squares quantities by `n-1`, however, in the original batch normalization paper it is assumed the variance is computed by dividing by `n` instead. The back-propagation equations in lecture 4 assume the latter therefore you will have to compensate for this fact by applying:

```
var_scores = var_scores * (n-1) / n;
```

Next up is implementation of the **backward pass**. You should upgrade the functions in the first part of the assignment to implement equations (19)-(26). Note you should probably write a separate function to implement

`BatchNormBackPass`. Once you have completed this then it is time to check your analytic gradient computations as per usual. Just a couple of tips:

- When you compute the loss in the numerical calculation of the gradient you have to apply the network function to the mini-batch data. When you do this you have to apply batch normalization and you should, as in your analytic gradient computations, compute the un-normalized means and variances from the mini-batch data.
- Make sure your mini-batch has size >1. You want to make sure your mean and variance computations are okay.

You should check with a 2-layer network (with 50 hidden nodes) and then a 3-layer network (with 50 and 30 hidden nodes respectively). After you have convinced yourself that your gradient computations are okay then you should move on to training your network. (The numerical gradient computations from Assignment 2 should be sufficient for Assignment 3.)

There is just one upgrade you need to make in the top level function implementing the mini-batch gradient descent learning algorithm (with momentum). You need to keep an exponential moving average of the batch mean and variances for the un-normalized scores for each layer of your network as defined by equations (27) and (28). You should use these moving averages when you compute the cost and accuracy on the training and validation sets after each epoch.

You should train a 3-layer network with 50 and 30 nodes in the first and second hidden layers respectively. You should train as in Assignment 2 by performing a coarse-to-fine search for good values of learning rate `eta` and the regularization penalty `lambda`. After you have found a good setting for the hyper-parameters you should train a network for `20` epochs and see what test accuracy this network can achieve.

One of the stated *pros* of batch normalization is that training becomes more stable and that higher learning rates can be used as opposed to when batch normalization is not used. I would like you to explore if this is true for (the special case of) a 2-layer network. Thus the second experiment you should run is to train a 2-layer network with 50 hidden nodes (the same architecture from Assignment 2) and experiment whether you can train the network to get the same performance but check whether convergence to this performance can be achieved in *significantly* fewer update steps because you can use a higher learning rate.

To complete the assignment:

To pass the assignment you need to upload to Canvas:

1. The code for your k-layer network trained and tested with batch normalization assembled into one file.

2. A brief pdf report with the following content:

   i) State how you checked your analytic gradient computations (with some accompanying numbers) and whether you think that your gradient computations are bug free for your k-layer network with batch normalization.

   ii) Include graphs of the evolution of the loss function when you tried to train your 3-layer network without batch normalization and with batch normalization.

   iii) State the range of the values you searched for `lambda` and `eta`, the number of epochs used for training during the fine search, and the hyper-parameter settings for your best performing 3-layer network you trained with batch normalization. Also state the test accuracy achieved by network.

   iv) Plot the training and validation loss for your 2-layer network with batch normalization with 3 different learning rates (small, medium, high) for 10 epochs and make the same plots for a 2-layer network with no batch normalization.

Exercise 4: *Optional for bonus points*

1. **Optimize the performance of the network**

   It would be interesting to discover what is the best possible performance achievable by a k-layer fully connected network on CIFAR-10. From a quick search of the web it seems the best performance of a fully connected network on CIFAR-10 is 78%. The details of this network are available at How far can we go without convolution: Improving fully connected networks by Lin, Memisevic and Konda.

   Here are some tricks/avenues you can explore to help bump up performance:

   (a) Train for a longer time and use your validation set to make sure you don't overfit or keep a record of the best model before you begin to overfit.

   (b) Do a more exhaustive random search to find *good* values for the amount of regularization, the learning rate.

   (c) Do a more thorough search to find a good network architecture. Does making the network deeper improve performance?

   (d) It has been empirically reported in several works that you get better performance by the final network if you apply batch normalization to the scores after the non-linear activation function has been applied. You could investigate whether this is the case.

(e) Apply dropout to your training if you have a high number of hidden nodes and you feel you need more regularization.

(f) Augment your training data by applying small random geometric and photometric jitter to the original training data. You can do this on the fly by applying a random jitter to each image in the mini-batch before doing the forward and backward pass.

Bonus Points Available: 2 points (if you complete at least 3 (beyond using all the training data) improvements - you can follow my suggestions, think of your own or some combination of the two.)

To get the bonus point you must submit

(a) Your code.

(b) Pdf document reporting on your trained network with the best test accuracy, what improvements you made and which ones brought the largest gains (if any!).

Train network using a different activation to ReLu Use one of the other activation functions described in the lecture notes and build your network based on this. See how it changes the network's performance. Does it make things better or worse? Did you have to implement the slightly more involved version of batch normalization, where you also scale and shift the activation scores? If you don't, with simple batch normalization you run the risk that for sigmoid activation functions you constrain the inputs to the linear regime of the sigmoid. Check out the journal version of the Batch Normalization paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift by Ioffe and Szegedy for details.

Bonus Points Available: 2 points

To get the bonus points you must submit

(a) Your code for computing the gradients.

(b) Pdf document comparing the test accuracy of the network trained with the non-ReLu activation function compared to the same network with a ReLu activation function for several sensible training parameter settings.