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. As in Assignment 2 you will train your network with mini-batch gradient descent and cyclical learning rates. 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 an L_2 regularization of the weight matrices see equation (20) 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

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

$$\mathbf{p} = \textsc{softmax}(\mathbf{s}) \tag{4}$$

The equations for the gradient computations of the back-propagation algorithm for a k-layer network are given in Lecture 4. I suggest you implement the efficient version of the backward pass at it will make your computations much faster.

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

You will discover that training a network is tricky as its number of layers increase. A proper initialization of the weights is key and $\eta_{\max}$ in the cyclic learning rate approach may be relatively small and thus training is slow. The second part of the assignment is therefore devoted to implementing batch normalization to overcome these limitations and also to get a feel for the effect of this process on training. 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 $\mathbf{s}^{(l)}$.

It is also assumed that you have learnt during training extra parameters $\boldsymbol{\gamma}_1, \ldots, \boldsymbol{\gamma}_{k-1}$ and $\boldsymbol{\beta}_1, \ldots, \boldsymbol{\beta}_{k-1}$ to scale and shift each entry of the normalized activations at each layer. Batch normalization, followed by a scale and shift, is then implemented at test time with these equations:

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

$$\begin{aligned}
\mathbf{s}^{(l)} &= W_l\, \mathbf{x}^{(l-1)} + \mathbf{b}_l && (5)\\
\hat{\mathbf{s}}^{(l)} &= \text{BatchNormalize}(\mathbf{s}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) && (6)\\
\tilde{\mathbf{s}}^{(l)} &= \boldsymbol{\gamma}_l \odot \hat{\mathbf{s}}^{(l)} + \boldsymbol{\beta}_l && (7)\\
\mathbf{x}^{(l)} &= \max(0, \tilde{\mathbf{s}}^{(l)}) && (8)
\end{aligned}$$

and then finally

$$\begin{aligned}
\mathbf{s} &= W_k\, \mathbf{x}^{(k-1)} + \mathbf{b}_k && (9)\\
\mathbf{p} &= \textsc{softmax}(\mathbf{s}) && (10)
\end{aligned}$$

where

$$\text{BatchNormalize}(\mathbf{s}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) = \left(\text{diag}(\mathbf{v}^{(l)} + \epsilon)\right)^{-\frac{1}{2}} \left(\mathbf{s}^{(l)} - \boldsymbol{\mu}^{(l)}\right) \qquad (11)$$

and $\epsilon > 0$ is a small number, of the order of magnitude of `Matlab`'s `eps` constant, 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

$$\mathbf{s}_i^{(l)} = W_l\, \mathbf{x}_i^{(l-1)} + \mathbf{b}_l \quad \text{for } i = 1, \ldots, n \tag{12}$$

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

$$\boldsymbol{\mu}^{(l)} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{s}_i^{(l)} \tag{13}$$

$$v_j^{(l)} = \frac{1}{n} \sum_{i=1}^{n} \left(s_{ij}^{(l)} - \mu_j^{(l)} \right)^2 \quad \text{for } j = 1, \ldots, m_l \tag{14}$$

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 $i = 1, \ldots, n$:

$$\hat{\mathbf{s}}_i^{(l)} = \text{BatchNormalize}(\mathbf{s}_i^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}) \tag{15}$$

$$\tilde{\mathbf{s}}_i^{(l)} = \boldsymbol{\gamma}_l \odot \hat{\mathbf{s}}_i^{(l)} + \boldsymbol{\beta}_l \tag{16}$$

$$\mathbf{x}_i^{(l)} = \max(0, \tilde{\mathbf{s}}_i^{(l)}) \tag{17}$$

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

$$\mathbf{s}_i = W_k\, \mathbf{x}_i^{(k-1)} + \mathbf{b}_k \tag{18}$$

$$\mathbf{p}_i = \textsc{softmax}(\mathbf{s}_i) \tag{19}$$

Backward pass of BN for back-propagation training

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

$$J(\mathcal{B}, \lambda, \boldsymbol{\Theta}) = \frac{1}{n} \sum_{i=1}^{n} l_{\text{cross}}(\mathbf{x}_i, y_i, \boldsymbol{\Theta}) + \lambda \sum_{i=1}^{k} \|W_i\|^2 \tag{20}$$

From the forward pass of the back-prop algorithm you should store

$$X_{\text{batch}}^{(l)} = \left(\mathbf{x}_1^{(l)}, \mathbf{x}_2^{(l)}, \ldots, \mathbf{x}_n^{(l)}\right), \quad S_{\text{batch}}^{(l)} = \left(\mathbf{s}_1^{(l)}, \mathbf{s}_2^{(l)}, \ldots, \mathbf{s}_n^{(l)}\right), \quad \hat{S}_{\text{batch}}^{(l)} = \left(\hat{\mathbf{s}}_1^{(l)}, \hat{\mathbf{s}}_2^{(l)}, \ldots, \hat{\mathbf{s}}_n^{(l)}\right)$$

and $\boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}$ for the intermediary layers $l = 1, \ldots, (k-1)$ and then also the final probability vectors output for each example in the batch:

$$P_{\text{batch}} = (\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n)$$

Given these quantities it is then possible to compute the gradient for all the parameters that have to be learnt in the network:

- Propagate the gradient through the loss and softmax operations
$$G_{\text{batch}} = -(Y_{\text{batch}} - P_{\text{batch}}) \tag{21}$$
- The gradients of J w.r.t. bias vector $\mathbf{b}_k$ and W_k
$$\frac{\partial J}{\partial W_k} = \frac{1}{n} G_{\text{batch}} X_{\text{batch}}^{(k-1)^T} + 2\lambda W_k, \quad \frac{\partial J}{\partial \mathbf{b}_k} = \frac{1}{n} G_{\text{batch}} \mathbf{1}_n \tag{22}$$
- Propagate G_{batch} to the previous layer
$$G_{\text{batch}} = W_k^T G_{\text{batch}} \tag{23}$$
$$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}\left(X_{\text{batch}}^{(k-1)} > 0\right) \tag{24}$$
- For $l = k-1, k-2, \ldots, 1$
  1. Compute gradient for the scale and offset parameters for layer l:
  $$\frac{\partial J}{\partial \boldsymbol{\gamma}_l} = \frac{1}{n}\left(G_{\text{batch}} \odot \hat{S}_{\text{batch}}^{(l)}\right)\mathbf{1}_n, \quad \frac{\partial J}{\partial \boldsymbol{\beta}_l} = \frac{1}{n} G_{\text{batch}} \mathbf{1}_n \tag{25}$$
  2. Propagate the gradients through the scale and shift
  $$G_{\text{batch}} = G_{\text{batch}} \odot \left(\boldsymbol{\gamma}_l \mathbf{1}_n^T\right) \tag{26}$$
  3. Propagate G_{batch} through the batch normalization
  $$G_{\text{batch}} = \texttt{BatchNormBackPass}\left(G_{\text{batch}}, S_{\text{batch}}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}\right) \tag{27}$$
  4. The gradients of J w.r.t. bias vector $\mathbf{b}_l$ and W_l
  $$\frac{\partial J}{\partial W_l} = \frac{1}{n} G_{\text{batch}} X_{\text{batch}}^{(l-1)^T} + 2\lambda W_l, \quad \frac{\partial J}{\partial \mathbf{b}_l} = \frac{1}{n} G_{\text{batch}} \mathbf{1}_n \tag{28}$$
  5. If $l > 1$ propagate G_{batch} to the previous layer
  $$G_{\text{batch}} = W_l^T G_{\text{batch}} \tag{29}$$
  $$G_{\text{batch}} = G_{\text{batch}} \odot \text{Ind}\left(X_{\text{batch}}^{(l-1)} > 0\right) \tag{30}$$

where the function $\texttt{BatchNormBackPass}\left(G_{\text{batch}}, S_{\text{batch}}^{(l)}, \boldsymbol{\mu}^{(l)}, \mathbf{v}^{(l)}\right)$ corresponds to the following steps:

$$\boldsymbol{\sigma}_1 = \left((v_1^{(l)} + \epsilon)^{-.5}, \ldots, (v_m^{(l)} + \epsilon)^{-.5}\right)^T \tag{31}$$
$$\boldsymbol{\sigma}_2 = \left((v_1^{(l)} + \epsilon)^{-1.5}, \ldots, (v_m^{(l)} + \epsilon)^{-1.5}\right)^T \tag{32}$$
$$G_1 = G_{\text{batch}} \odot \left(\boldsymbol{\sigma}_1 \mathbf{1}_n^T\right) \tag{33}$$
$$G_2 = G_{\text{batch}} \odot \left(\boldsymbol{\sigma}_2 \mathbf{1}_n^T\right) \tag{34}$$
$$D = S_{\text{batch}}^{(l)} - \boldsymbol{\mu}^{(l)} \mathbf{1}_n^T \tag{35}$$
$$\mathbf{c} = (G_2 \odot D)\,\mathbf{1}_n \tag{36}$$
$$G_{\text{batch}} = G_1 - \frac{1}{n}(G_1 \mathbf{1}_n)\,\mathbf{1}_n^T - \frac{1}{n} D \odot \left(\mathbf{c}\mathbf{1}_n^T\right) \tag{37}$$

assuming $\mathbf{v}^{(l)} = (v_1^{(l)}, \ldots, v_m^{(l)})^T$. Remember that the ith column of G_{batch} sent into `BatchNormBackPass` represents $\frac{\partial J}{\partial \hat{\mathbf{s}}_i^{(l)}}$ while the ith column of G_{batch} returned by `BatchNormBackPass` represents $\frac{\partial J}{\partial \mathbf{s}_i^{(l)}}$.

You should note that the network's bias parameters $\mathbf{b}_l$ for $l = 1, \ldots, k-1$ are superfluous when using batch normalization as you will subtract away these biases when you normalize. These bias parameters will be estimated as effectively zero vectors when you train.

Exponential moving average for batch means and variances

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

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

$$\boldsymbol{\mu}_{\text{av}}^{(l)} = \alpha \boldsymbol{\mu}_{\text{av}}^{(l)} + (1-\alpha)\boldsymbol{\mu}^{(l)} \tag{38}$$

$$\mathbf{v}_{\text{av}}^{(l)} = \alpha \mathbf{v}_{\text{av}}^{(l)} + (1-\alpha)\mathbf{v}^{(l)} \tag{39}$$

where $\alpha \in (0,1)$ and typically $\alpha \approx .9$ (in this assignment as our training is shorter than usual and this requires a smaller value for α). 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 4) (**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. As before please only do numerical checks on networks with a small number of nodes in each layer and a much reduced dimensional input data ($d \approx 10$) to avoid numerical precision issues. 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. You can 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`. 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 multi-layer networks?*

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 using mini-batch gradient descent with a cyclic learning rate. If the answer is yes, then your next task is to train a 3-layer network with 50 and 50 nodes in the first and second hidden layer respectively with the same learning parameters. You can use the following hyper-parameter setting `n_batch=100`, `eta_min = 1e-5`, `eta_max = 1e-1`, `lambda=.005`, two cycles of training and `n_s = 5 * 45,000 / n_batch`. You should use a careful initialization such as Xavier or He initialization. With these settings my trained network after two cycles got a test accuracy of ∼52%. I also randomly shuffle the order of the training data after each epoch. Now consider a 9-layer network whose number of nodes at the hidden layers are [50, 30, 20, 20, 10, 10, 10, 10]. This network has the same number of weight parameters as the earlier network. Train the network with the same hyper-parameter settings as before and see what happens to performance.

For the deeper network its performance dropped by quite a bit. Generally as a network becomes deeper it becomes harder to train when training with variants of mini-batch gradient descent and using a more standard decay of the learning rate. The technique of *batch normalization* is a way to overcome this difficulty.

Exercise 3: *Implement batch normalization*

You have seen that training networks with many layers can be tricky. 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 (12) - (19) (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 of 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 a batch ar the lth layer in the matrix `scores` (this would correspond to $S^{(l)}_{\text{batch}}$ in the mathematical description) 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 the lecture slides 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 (21)-(30). 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 50 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 have to be augmented with the numerical gradient computations w.r.t. the parameters $\boldsymbol{\gamma}_l$ and $\boldsymbol{\beta}_l$.)

There is just one upgrade you need to make in the top level function implementing the mini-batch gradient descent learning algorithm. 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 (38) and (39). 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 50 nodes in the first and second hidden layers respectively. You should train as in Assignment 2 with cyclic learning rate. I achieved quite good results (when using `45,000` training examples) with He initialization and hyper-parameter settings of `eta_min = 1e-5`, `eta_max = 1e-1`, `lambda=.005`, two cycles of training and `n_s = 5 * 45,000 / n_batch`. With this set-up I was able to achieve test accuracies of $\sim$53.5%. (To reach this level of accuracy when using BN it seems to be important that you shuffle the order of your training samples after each epoch. I think the reason for this is that it ensures you have different combinations of training examples in your batches over epochs and this is good for regularization and estimating the mean and standard deviation of the activations at each layer.) You should perform a proper some coarse-to-fine search to find a good value for `lambda`. After you have found a good setting for `lambda`, you should train a network for 3 cycles and see what test accuracy this network can achieve.

Now reconsider the 9-layer network whose number of nodes at the hidden layers are [50, 30, 20, 20, 10, 10, 10, 10] respectively. Train the network with the same hyper-parameter settings as your 3-layer network and see what happens to performance. Not bad! Hopefully this result will convince you that *batch normalization* is a good thing!

The frequently stated *pros* of batch normalization are that training becomes more stable, higher learning rates can be used as opposed to when batch normalization is not used and it acts as a form of regularization. I would like you to explore if you can get some experimental evidence that is consistent with one of these stated pros. You will train your 3-layer network with 50 nodes at each hidden layer and basic hyper-parameter setting will `eta_min = 1e-5`, `eta_max = 1e-1`, `lambda=.005`, two cycles of training with `n_s =`

`5 * 45,000 / n_batch`.

Sensitivity to initialization For each training regime instead of using He initialization, initialize each weight parameter to be normally distributed with `sigmas` equal to the same value `sig` at each layer. For three runs set `sig=1e-1, 1e-3` and 1e-4 respectively and train the network with and without BN and see the effect on the final test accuracy. Use `n_s = 2 * 45,000 / n_batch` if training is slow on your machine and you want to complete training with fewer update steps.

s53.

To complete the assignment:

To pass the assignment you need to upload to bilda:

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 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 train the 3-layer network with and without batch normalization with the given default parameter setting.
    iii) Include graphs of the evolution of the loss function when you train the 9-layer network with and without batch normalization with the given default parameter setting.
    iv) State the range of the values you searched for `lambda` when you tried to optimize the performance of the 3-layer network trained with batch normalization, and the `lambda` settings for your best performing 3-layer network. Also state the test accuracy achieved by this network.
    v) Include the loss plots for the training with Batch Norm Vs no Batch Norm for the experiment related to *Sensitivity to initialization* and comment on your experimental findings.

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) Do a more exhaustive random search to find *good* values for the amount of regularization.

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

   (c) 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. You will have to update your forward and backward pass of the back-prop algorithm accordingly.

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

   (e) 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: 1 bonus point for each non-trivial improvement (capped at 4 bonus points, 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!).