Course: DD2424 - Assignment 2

In this assignment you will train and test a two layer network with multiple outputs to classify images from the CIFAR-10 dataset. You will train the network using mini-batch gradient descent applied to a cost function that computes the cross-entropy loss of the classifier applied to the labelled training data and an L_2 regularization term on the weight matrix.

The overall structure of your code for this assignment should mimic that from **Assignment 1**. You will have more parameters than before and you will have to change the functions that **1**) evaluate the network (the forward pass) and **2**) compute the gradients (the backward pass). We will also be paying more attention to how to search for good parameter settings for the network's regularization term and the learning rate. Welcome to the nitty gritty of training neural networks!

Background 1: *Mathematical background*

The mathematical details of the network 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:

$$\mathbf{s}_1 = W_1\mathbf{x} + \mathbf{b}_1 \tag{1}$$
$$\mathbf{h} = \max(0, \mathbf{x}) \tag{2}$$
$$\mathbf{s} = W_2\mathbf{h} + \mathbf{b}_2 \tag{3}$$
$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{4}$$

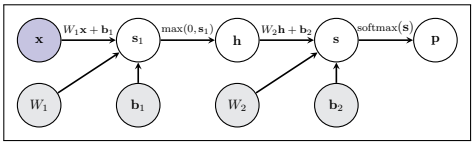
where the matrix W_1 and W_2 have size $m \times d$ and $K \times m$ respectively and the vectors $\mathbf{b}_1$ and $\mathbf{b}_2$ have sizes $m \times 1$ and $K \times 1$. SOFTMAX is defined as

$$\text{SOFTMAX}(\mathbf{s}) = \frac{\exp(\mathbf{s})}{\mathbf{1}^T \exp(\mathbf{s})} \tag{5}$$

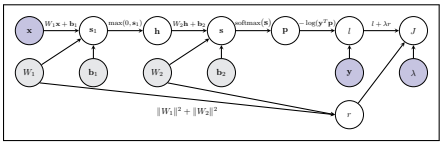
The predicted class corresponds to the label with the highest probability:

$$k^* = \arg \max_{1 \le k \le K} \{p_1, \ldots, p_K\} \tag{6}$$

We have to learn the parameters $W_1, W_2, \mathbf{b}_1$ and $\mathbf{b}_2$ from our labelled training data. The parameters W and $\mathbf{b}$ of our classifier are what we have to learn by exploiting labelled training data. Let $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$, with each $y_i \in \{1, \ldots, K\}$ and $\mathbf{x}_i \in \mathbb{R}^d$, represent our labelled training data. In the lectures we have described how to set the parameters by minimizing the cross-entropy loss plus a regularization term on W_1 and W_2. To simplify the



a) Classification function



b) Cost function

Figure 1: For this assignment the computational graph of the classification function applied to an input $\mathbf{x}$ and the computational graph of the cost function applied to a mini-batch of size 1.

notation we group the parameters of the model as $\boldsymbol{\Theta} = \{W_1, W_2, \mathbf{b}_1, \mathbf{b}_2\}$. The cost function is

$$J(\mathcal{D}, \lambda, \boldsymbol{\Theta}) = \frac{1}{|\mathcal{D}|} \sum_{\mathbf{x},y \in \mathcal{D}} l_{\text{cross}}(\mathbf{x}_i, y_i, \boldsymbol{\Theta}) + \lambda \sum_{l=1}^{2} \sum_{i,j} W_{l,ij}^2 \tag{7}$$

where

$$l_{\text{cross}}(\mathbf{x}, y, \boldsymbol{\Theta}) = -\log(p_y) \tag{8}$$

and $\mathbf{p}$ has been calculated using equations (1-4). (Note if the label is encoded by a one-hot representation then the cross-entropy loss is defined as $-\log(\mathbf{y}^T\mathbf{p})$.) The optimization problem we have to solve is

$$\boldsymbol{\Theta}^* = \arg\min_{\boldsymbol{\Theta}} J(\mathcal{D}, \lambda, \boldsymbol{\Theta}) \tag{9}$$

In this assignment (as described in the lectures) we will solve this optimization problem via mini-batch gradient descent (with momentum).

For mini-batch gradient descent we begin with a sensible random initialization of the parameters $W, \mathbf{b}$ and we then update our estimate for the parameters with for $k = 1, 2$

$$W_k^{(t+1)} = W_k^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, \boldsymbol{\Theta})}{\partial W_k} \right|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}^{(t)}} \tag{10}$$

$$\mathbf{b}_k^{(t+1)} = \mathbf{b}_k^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, \boldsymbol{\Theta})}{\partial \mathbf{b}_k} \right|_{\boldsymbol{\Theta}=\boldsymbol{\Theta}^{(t)}} \tag{11}$$

where η is the learning rate and $\mathcal{B}^{(t+1)}$ is called a mini-batch and is a random subset of the training data $\mathcal{D}$ and for $k = 1, 2$:

$$\frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, \boldsymbol{\Theta})}{\partial W_k} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y) \in \mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, \boldsymbol{\Theta})}{\partial W_k} + 2\lambda \sum_{i,j} W_{k,ij} \tag{12}$$

$$\frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, \boldsymbol{\Theta})}{\partial \mathbf{b}_k} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y) \in \mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, \boldsymbol{\Theta})}{\partial \mathbf{b}_k} \tag{13}$$

To compute the relevant gradients for the mini-batch, we then have to compute the gradient of the loss w.r.t. each training example in the mini-batch. You should refer to the lecture notes for the explicit description of how to compute these gradients.

Background 2: *Speeding up training: Add momentum to training*

The vanilla version of mini-batch gradient descent with a sensible learning-rate is painfully slow for the size of the network and data we will use in this exercise. To speed up training, we must add a momentum term in the update step. This is achieved as follows. You initialize a momentum vector (matrix) $\mathbf{v}_0$ for each parameter of the network ($\mathbf{v}_0$ has the same dimension as the parameter vector/matrix and is initialized to have zero in all its entries) and then at each time step t:

$$\mathbf{v}_t = \rho \mathbf{v}_{t-1} + \eta \frac{\partial J}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t$$

where η is the learning rate as in standard mini-batch gradient descent, $\rho \in [0, 1]$ and $\boldsymbol{\theta}$ is a generic placeholder to represent one of the parameters of the model. Typically ρ is set to .9 or .99.

For this assignment you will not have an adaptive learning rate, but you will potentially decay the learning rate by some factor, say $\sim .95$, after each epoch. The Additional material for lecture 3 from Standford's course **Convolutional Neural Networks for Visual Recognition** has a small section describing common ways to anneal the learning rate.

Exercise 1: *Read in the data & initialize the parameters of the network*

For this assignment (to begin) you will just use the data in the file `data_batch_1.mat` for training, the file `data_batch_2.mat` for validation and the file `test_batch.mat` for testing. You have already written a function for Assignment 1 to read in the data and pre-process it slightly. For this assignment it helps if we apply a little more pre-processing to the raw input data. You should transform it to have zero mean. If `X` is the $d \times n$ image data matrix (each column corresponds to an image) then perform

```
mean_X = mean(X, 2);
X = X - repmat(mean_X, [1, size(X, 2)]);
```

(**Note** You should only compute the mean for the training data and then keep a record of this mean and subtract it from the input vectors in the

validation and test sets. You have to assume that you will only see your test data one item at a time not in bunches.)

Next you have to set up the data structure for the parameters of the network and to initialize their values. In the assignment we will just focus on a network that has `m=50` nodes in the hidden layer. As `W1` and `W2` will have different sizes, as will `b1` and `b2`, I recommend you use `cell arrays` to store these matrices and vectors. To set their initial values I typically set the bias vectors to zero and the entries in the weight matrices are random draws from a Gaussian distribution with mean 0 and standard deviation `.001`. (You can use the *Matlab* function `randn` to generate these random draws). You should probably write a separate function to initialize the parameters as you will be initializing your network frequently as you perform grid searches for good values of your hyper-parameters.

Exercise 2: *Compute the gradients for the network parameters*

Next you will write functions to compute the gradient of your two-layer network w.r.t. its W and $\mathbf{b}$ parameters. As before I suggest you re-use much of your code from `Assignment1.m` You will need to write (update) the following functions (that you wrote previously):

- Compute the network function, to apply the function defined in figure 1 (**a**), on a mini-batch of data and that returns the final $\mathbf{p}$ values and the intermediary activation values. The latter will be needed to compute the gradients.
- Compute the gradients of the cost function for a mini-batch of data given the values computed from the forward pass.

Once you have written the code to compute the gradients the next step is debugging. Download code from the KTH Social that computes the gradient vectors numerically. Note there are two versions **1**) a slower but more accurate version based on the *centered difference* formula and **2**) a faster but less accurate based on the *finite difference method*. As in Assignment 1 you will probably have to make small changes to the function to make it compatible with your own code, especially if you have not stored the `W`'s and `b`'s as cell arrays. You should use the same procedures as described in `Assignment1` to check the gradients. As in that assignment, when checking the gradients, you should probably reduce the dimensionality of the input vector (both for speed and numerical precision issues). I also found that `h`$\approx$ `1e-5` gave the best precision for the numerical gradients.

Once you have convinced yourself that your analytic gradient computations are correct then you can move forward with the following sanity check. Try

and train your network on a small amount of the training data (say 100 examples) with regularization turned off (`lambda`=0) and check if you can overfit to the training data and get a very low loss on the training data after training for a sufficient number of epochs (∼200) and with a reasonable η. Being able to achieve this indicates that your gradient computations and mini-batch gradient descent algorithm are okay.

Exercise 3: *Add momentum to your update step*

Up until now you have trained your networks with vanilla mini-batch gradient descent. To help speed up training times you should add momentum terms into your mini-batch update steps. Once you have implemented this, you should repeat the sanity check that your network can fit to a small amount of training data. You can set `rho` to values in $\{.5, .9, .99\}$ and check that you can still learn and that learning is now faster. You can also decay your learning rate by a factor of `decay_rate` (i.e. `eta = eta*decay_rate`) after each epoch of training. Typically the decay rate is set to ∼`.95`. At this stage we are now ready to train the network for real.

Exercise 4: *Training your network*

There is quite a bit of babysitting required to training a network. First, you have to find the range of effective values for the learning rate. Next you must perform coarse to fine random searches to discover *good/effective* values for these terms (in tandem). (Check out Random Search for Hyper-Parameter Optimization why it is probably better to perform random searches over regular grid searches when some of your hyper-parameters are more influential on the final result.) For these experiments you can set the momentum term `rho=.9` in the update step.

Find a reasonable range of values for the learning rate. Set the regularization term to a small value say `.000001`. Your randomly initialized network should perform similarly to a random guesser and thus the training loss before you start learning should be ≈`2.3` (≈ ln(.1)). You want to perform a quick search by hand to find the rough bounds for reasonable values of the learning rate. During this stage you should print out the training cost/lost after every epoch. If the learning rate is too small, then after each epoch you will find that the training loss barely changes. While if the learning rate is too large, then learning is unstable and you will probably get `NaN`s and/or very high loss values. You should only need ∼5 epochs to check these properties. After some quick experiments you should find the rough range for the learning rate you should cross-validate (or validate). Figure 2 is a cartoon representation of how the loss evolves given different

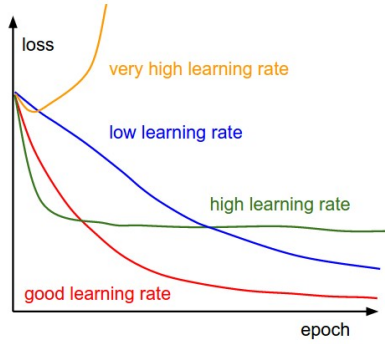


Figure 2: Above shows the effect of typical effect of different learning rates on evolution of the training loss during training. Figure taken from Convolutional Neural Networks for Visual Recognition.

qualitative values of the learning rate.

Coarse-to-fine random search to set `lambda` and `eta`. At this point you may need to restructure/re-organize your code so that you can cleanly and easily call function(s) to initialize your network, perform training and then check the learnt network's best performance on the validation set. To perform your random search you'll need to train your network from a random initialization and measure its performance (via the accuracy on the validation set) multiple times as the hyper-parameters of `eta` and `lambda` vary. You should first perform a coarse search over the range of feasible learning-rates you have identified in tandem with a search over a very broad range of values for `lambda`. Here you should only run a few epochs of training to get rough idea of what parameter settings work. Search for the hyper-parameters on a log scale, for example to generate one random sample for the learning rate uniformly in the range of `10^e_min` to `10^e_max`

```
e = e_min + (e_max - e_min)*rand(1, 1);
eta = 10^e;
```

You should perhaps try somewhere between `50-100` different pairings for `lambda` and `eta` generated by uniform sampling. Save all the parameter settings tried and their resulting best scores on the validation set to a file. Inspect this file after finishing the coarse search and see what parameter ranges gave the best results. Repeat the random search but with your search adjusted to a narrower range and possibly run training for a few more epochs than before. Once again save the results and look for the best parameter settings. You could do another round of random search or just use the good found parameter settings, train the model for longer and see what final performance you get on the test set. You should be getting performances

of at least >44% for your good settings given 10 epochs of training and just using the first batch of training data.

Note: during training if the training cost is ever > 3×original training cost, then abort training and break out early.

To complete the assignment:

To pass the assignment you need to upload to bilda:

1. The code for your assignment 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 were bug free.
   - ii) Comment on how much faster the momentum term made your training.
   - iii) State the range of the values you searched for `lambda` and `eta`, the number of epochs used for training during the coarse search and the hyper-parameter settings for the 3 best performing networks you trained.
   - iv) 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 the 3 best performing networks you trained.
   - v) For your best found hyper-parameter setting (according to performance on the validation set), train the network on all the training data (all the batch data), except for 1000 examples in a validation set, for ∼30 epochs. Plot the training and validation cost after each epoch of training and then report the learnt network's performance on the test data.

Exercise 5: *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 Assignment 2's network (a 2-layer fully connected network) on CIFAR-10. 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 and the learning rate.

(c) You could also explore whether having more hidden nodes improves the final classification rate. One would expect that with more hidden nodes then the amount of regularization would have to increase.

(d) Play around with different approaches to annealing the learning rate. For example you can keep the learning rate fixed over several epochs then decay it by a factor of 10 after n epochs.

(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?

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.

I'm interested in finding out what is the best possible accuracy for a 2-layer fully-connected network on CIFAR-10. Please do fill in your best result (enter as a percentage) in the form for DD2424 Assignment 2 Leaderboard and also give a brief synopsis of your network and training regime. To see what other results students have achieved here is a link to the current version of the leaderboard.

At the submission deadline **11:59pm on 24th of April** the student within `.3%` of the top accuracy on the leaderboard will get 1 bonus point (if test performance $> 53\%$).