Course: DD2424 - Assignment 1

In this assignment you will train and test a one 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.

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:

$$\mathbf{s} = W\mathbf{x} + \mathbf{b} \tag{1}$$

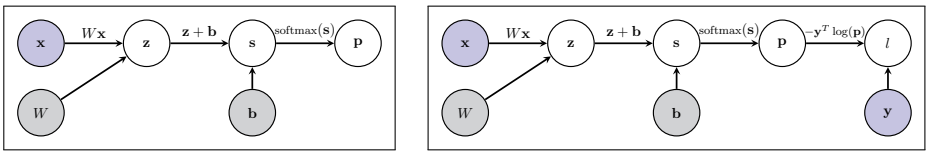
$$\mathbf{p} = \text{SOFTMAX}(\mathbf{s}) \tag{2}$$

where the matrix W has size $K \times d$, the vector $\mathbf{b}$ is $K \times 1$ and SOFTMAX is defined as

$$\text{SOFTMAX}(\mathbf{s}) = \frac{\exp(\mathbf{s})}{\mathbf{1}^T \exp(\mathbf{s})} \tag{3}$$

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

$$k^* = \arg \max_{1 \le k \le K} \{p_1, \ldots, p_K\} \tag{4}$$



a) Classification function **b) Loss function**

Figure 1: Computational graph of the classification and loss function that is applied to each input $\mathbf{x}$ in this assignment.

The classifier's parameters W and $\mathbf{b}$ are what we have to learn from the 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. Mathematically this cost function is

$$J(\mathcal{D}, \lambda, W, b) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y) \in \mathcal{D}} l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b}) + \lambda \sum_{i,j} W_{ij}^2 \tag{5}$$

where

$$l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b}) = -\log(p_y) \tag{6}$$

and $\mathbf{p}$ has been calculated using equations (1, 2). (Note if the label is encoded as one-hot representation then the cross-entropy loss is defined as $-\mathbf{y}^T \log(\mathbf{p}) = \log(p_y)$.) The optimization problem we have to solve is

$$W^*, \mathbf{b}^* = \arg\min_{W,\mathbf{b}} J(\mathcal{D}, \lambda, W, b) \tag{7}$$

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

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

$$W^{(t+1)} = W^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, W, \mathbf{b})}{\partial W} \right|_{W=W^{(t)}, \mathbf{b}=\mathbf{b}^{(t)}} \tag{8}$$

$$\mathbf{b}^{(t+1)} = \mathbf{b}^{(t)} - \eta \left. \frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, W, \mathbf{b})}{\partial \mathbf{b}} \right|_{W=W^{(t)}, \mathbf{b}=\mathbf{b}^{(t)}} \tag{9}$$

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

$$\frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, W, \mathbf{b})}{\partial W} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y)\in\mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b})}{\partial W} + 2\lambda W \tag{10}$$

$$\frac{\partial J(\mathcal{B}^{(t+1)}, \lambda, W, \mathbf{b})}{\partial \mathbf{b}} = \frac{1}{|\mathcal{B}^{(t+1)}|} \sum_{(\mathbf{x},y)\in\mathcal{B}^{(t+1)}} \frac{\partial l_{\text{cross}}(\mathbf{x}, y, W, \mathbf{b})}{\partial \mathbf{b}} \tag{11}$$

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.

Before Starting

I assume you will complete the assignment in *Python* and use the standard *numpy* library for the required mathematical and matrix operations. You will also use *PyTorch* to calculate *ground truth* gradients which will be used as reference for your gradient calculations. We will not use the GPU implementations of *PyTorch* thus you can install and run it without GPU support. Check out Get Started to figure out how to get *PyTorch* working within your environment. You can complete the assignment in another

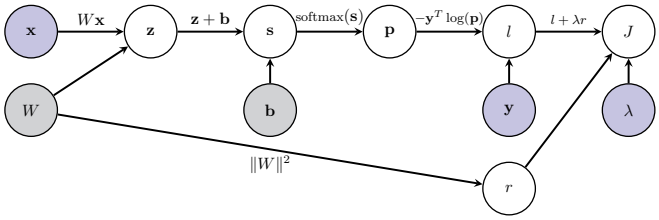


Figure 2: Computational graph of the cost function applied to a mini-batch containing one training example $\mathbf{x}$. If you have a mini-batch of size greater than one, then the loss computations are repeated for each entry in the mini-batch (as in the above graph), but the regularization term is only computed once.

programming language. If you do though I will not answer programming specific questions and you will also probably have to find a way to display, plot and graph your results. Another comment is that I'm not a sophisticated python programmer so feel free for the more experienced programmers to come up with better overall structures for the implementation and/or also optimize for speed, readability etc.

Besides invoking *Python* commands, you will be required to run a few operating system commands. For these commands I will assume your computer's operating system is either linux or unix. If otherwise, you'll have to fend for yourself. But all the non-*Python* commands needed are more-or-less trivial.

The notes for this assignment, and those to follow, will give you pointers about which *Python* commands to use. However, I will not give detailed explanations about their usage. I assume you have some previous experience with *Python* and *numpy*, are aware of many of the in-built functions, how to manipulate vectors and matrices and how to write your own functions etc.

Background 2: *Getting Started*

Set up your environment

Create a new directory to hold all the *Python* files you will write for this course:

```
$ mkdir DirName
$ cd DirName
$ mkdir Datasets
```

Download the CIFAR-10 dataset stored in its *Python* format from this link. Move the `cifar-10-python.tar.gz` file to the `Datasets` directory you have

just created, `untar`the file and then move up to the parent directory.

```
$ mv cifar-10-python.tar.gz DirName/Datasets
$ cd DirName/Datasets
$ tar xvfz cifar-10-python.tar.gz
$ cd ..
```

Background 3: *Useful Display Function*

There are several options for displaying images. A straight forward way is with `matplotlib`. If you have a `numpy` array, for example `im` with size `(32, 32, 3)` and entries between 0 and 1 then you can display the image with this code

```
import matplotlib.pyplot as plt

......
plt.imshow(im)
plt.show()
```

Note if you are running the code from the command line and want the programme to continue to run and not stop while the graphic window is open you should replace the `plt.show()` with `plt.pause(0.1)` that is

```
import matplotlib.pyplot as plt

.....
plt.imshow(im)
plt.pause(0.1)
```

To display some of the images from the `CIFAR-10` dataset you can use the following sequence of commands to display 5 training images:

```
# Load a batch of training data
cifar_dir = 'DirName/Datasets/cifar-10-batches-mat/'
with open(cifar_dir + 'data_batch_1', 'rb') as fo:
    dict = pickle.load(fo, encoding='bytes')
# Extract the image data and cast to float from the dict dictionary
X = dict[b'data'].astype(np.float64) / 255.0
X = X.transpose()
nn = X.shape[1]
# Reshape each image from a column vector to a 3d array
X_im = X.reshape((32, 32, 3, nn), order='F')
X_im = np.transpose(xx, (1, 0, 2, 3))
```

```
# Display the first 5 images
ni = 5
fig, axs = plt.subplots(1, 5, figsize=(10, 5))
for i in range(ni):
     axs[i].imshow(X_im[:, :, :, i])
     axs[i].axis('off')
plt.pause(0.1)
```

This sequence of commands loads the first training file containing image and label data. You access the image data with the dictionary `dict[b'data']` and we store it as `X`. It has size 10000×3072. Each row of `dict[b'data']` corresponds to an image of size $32 \times 32 \times 3$ that has been flattened into a row vector. We transpose `X` so each column corresponds to a an image and then to allow display of the underlying images we rearrange the entries to create a 4d tensor of size $(32 \times 32 \times 3 \times 10000)$ using the command `reshape`. After reshaping the array, the rows and columns of each image still need to be permuted and this is achieved with the `transpose` command. Now you have the images in the 2D array format matplotlib expects when displaying an image. In the above code we display the first 5 images using the `subplot` command.

You have looked at some of the CIFAR-10 images. Now it is time to start writing some code.

Exercise 1: *Training a multi-linear classifier*

For this assignment you will just use data in the file `data_batch_1` for training, the file `data_batch_2` for validation and the file `test_batch` for testing. Create a file `Assignment1.py`. In this file you will write the code for this assignment and the necessary (sub-)functions. Here are my recommendations for which functions to write and the order in which to write them:

1. Write a function that reads in the data from a CIFAR-10 batch file and returns the image and label data in separate files. Make sure to convert your image data to `float32` or `float64` format. I would suggest the function has the following input and outputs

   ```
   function [X, Y, y] = LoadBatch(filename)
   ```

   where

   - `X` contains the image pixel data, has size `d`×`n`, is of type `float32` or `float64` and has entries between 0.0 and 1.0. `n` is the number of images (10000) and `d` the dimensionality of each image (3072=32×32×3).
   - `Y` is `K`×`n` (`K`= # of labels = 10) and contains the one-hot representation of the label for each image. As `Y` will be used in the back-prop calculations it will need to have the same type as `X`.

- `y` is a vector of length `n` containing the label for each image. CIFAR-10 encodes the labels as integers between 0-9. As this array will be used for indexing it can remain as a list of integers.

This function will not be long. You just need to read a batch of data and then rearrange and cast to the type the quantities `dict[b'data']` and `dict[b'labels']` assuming that `dict` is the dictionary you read in from the pickle file.

Top-level: Read in and store the training, validation and test data.

2. Next we should pre-process the raw input data as this helps training. You should transform training data to have zero mean. If `trainX` is the `d`×`n` image data matrix (each column corresponds to an image) for the training data then

```
mean_X = np.mean(X, axis=1).reshape(d, 1)
std_X = np.std(trainX, axis=1).reshape(d, 1)
```

Important both `mean_X` and `std_X` have size `d`×`1`. You normalize w.r.t to each dimension.

You should normalize the training, validation and test data with respect to the mean and standard deviation values computed from the training data as follows. If `X` is an `d`×`n` image data matrix then you can normalize `X` as

```
X = X - mean_X
X = X / std_X
```

Note the above relies on broadcasting. In general *numpy* can do all sorts of broadcasting! So please double check when you have broadcasting that you expected to happen happened!

Top-level: Compute the mean and standard deviation vector for the training data and then normalize the training, validation and test data w.r.t. the training mean and standard deviation vectors.

3. **Top-Level**: After reading in and pre-processing the data, you can initialize the parameters of the model `W` and `b` as you now know what size they should be. It is handy to store these network parameters in a dictionary. `W` has size `K`×`d` and `b` is `K`×`1`. Initialize each entry to have Gaussian random values with zero mean and standard deviation .01. Here is some example code to set up the random number generation.

```
rng = np.random.default_rng()
# get the BitGenerator used by default_rng
BitGen = type(rng.bit_generator)
# use the state from a fresh bit generator
seed = 42
```

```
rng.bit_generator.state = BitGen(seed).state
init_net = {}
init_net['W'] = .01*rng.standard_normal(size = (K, d))
init_net['b'] = np.zeros((K, 1))
```

4. Write a function that applies the network function, i.e. equations (1, 2), to multiple images and returns the results. I would suggest the function has the following inputs and outputs

   ```
   P = ApplyNetwork(X, network)
   ```

   where

   - each column of `X` corresponds to an image and it has size `d`×`n`.
   - `network` is a dictionary with keys `'W'` and `'b'` that correspond to the parameters of the network.
   - each column of `P` contains the probability for each label for the image in the corresponding column of `X`. `P` has size `K`×`n`.

   Top-level: Check the function runs on a subset of the training data given a random initialization of the network's parameters:
   `P = ApplyNetwork(trainX[:, 0:100], init_net)`.

5. Write the function that computes the loss function given by equation (5) for a set of images. I suggest the function has the following inputs and output (note you could instead send in the data `X` and the network parameters `network` if you have not already pre-computed `P` with `ApplyNetwork`)

   ```
   L = ComputeLoss(P, y)
   ```

   where

   - each column of `P` is the probability of each class for the corresponding column of the data `X` and has size `K`×`n`.
   - `y` is (`1`×`n`) and corresponds to the ground truth label of each image whose predicted labels are contained in `P`.
   - `L` is a scalar corresponding to the mean cross-entropy loss of the network's predictions relative to the ground truth labels.

6. Write a function that computes the accuracy of the network's predictions given by equation (4) on a set of data. Remember the accuracy of a classifier for a given set of examples is the percentage of examples for which it gets the correct answer. At a high level you want to have function with these inputs and output (note you could instead send in the data `X` and the network parameters `network` if you have not already pre-computed `P` with `ApplyNetwork`)

```
acc = ComputeAccuracy(P, y)
```

where

- each column of `P` contains the probability of each class for the corresponding column of input data matrix `X` and has size `K`×`n`
- `y` is the vector of ground truth labels of length `n`.
- `acc` is a scalar value containing the accuracy.

7. Write the function that evaluates, for a mini-batch, the gradients of the cost function w.r.t. `W` and `b`, that is equations (10, 11). The function should have these inputs and outputs:

```
grads = BackwardPass(X, Y, P, network, lam)
```

where

- each column of `X` corresponds to an image and it has size `d`×`n`.
- each column of `Y` (`K`×`n`) is the one-hot ground truth label for the corresponding column of `X`.
- each column of `P` contains the probability for each label for the image in the corresponding column of `X`. `P` has size `K`×`n`.
- `lam` is the λ parameter in the cost function (we can't use the name `lambda` as this is name protected within `python`)
- `grads` is a dictionary with the gradients. The keys should be consistent with those of `network`. Then `grad['W']` is the gradient of the cost J relative to `W` and has size `K`×`d`. And `grad['b']` is the gradient vector of the cost J relative to `b` and has size `K`×`1`.

At this stage it might convenient, and make the code more readable and extendable, to also use a dictionary to store the outputs from the forward pass (ie `P` for this simple example) etc. However, this is not important for this assignment as `P` is the only data one has to keep track of from the forward pass.

Be sure to check out how you can efficiently compute the gradient for a batch from the last slide of Lecture 3. This can lead to a much faster implementation (> 3 times faster) than looping through each training example in the batch.

Everyone makes mistakes when computing gradients. Therefore you must always check your analytic gradient computations against numerical estimations of the gradients! For our assignments we will not actually compute the gradients numerically but will instead use `PyTorch`'s automatic differentiation to act as our generator of ground truth gradients. For this assignment download code, `torch_gradient_computations.py`, from the Canvas webpage. This code contains the function

```
ComputeGradsWithTorch
```

to compute the gradient vectors via `PyTorch`. Its input variables have the same format as for the rest of your code. The conversion of numpy arrays to torch tensors happens inside the function. You will probably have to make small changes to make it compatible with your code. You will also have to install `PyTorch`. As we will not use GPU implementations both these tasks should be fairly straightforward. As the number of training examples in `X` is large as is the input dimension you should do your checks on just a small batch of training data and also a much reduced input dimension. Here is a snippet code to show the type of calculations you should be performing for the checks:

```
d_small = 10
n_small = 3
lam = 0
small_net['W'] = .01*rng.standard_normal(size = (10, d_small))
small_net['b'] = np.zeros((10, 1))

X_small = trainX[0:d_small, 0:n_small]
Y_small = trainY[:, 0:n_small]

P = ApplyNetwork(X_small, small_net)
my_grads = BackwardPass(X_small, Y_small, P, small_net, lam)

torch_grads = ComputeGradsWithTorch(X_small, train_y[0:n_small], small_net)
```

After you have computed the gradients via your code and `PyTorch` you should check they have produced the same output. You can start by examining their absolute differences and declaring, if all these absolutes difference are small (<`1e-6`), then they have produced the same result. However, when the gradient vectors have small values this approach may fail. A more reliable method is to compute the relative error between a numerically computed gradient value g_n and an analytically computed gradient value g_a

$$\frac{|g_a - g_n|}{\max(\text{eps}, |g_a| + |g_n|)} \quad \text{where eps a very small positive number}$$

and check this is small. There are potentially more issues that can plague numerical gradient checking (especially when you start to train deep rectifier networks), so I suggest you read the relevant section of the Additional material for lecture 3 from Standford's course **Convolutional Neural Networks for Visual Recognition** for a more thorough exposition especially for the subsequent assignments.

Do not continue with the rest of this assignment until you are sure your analytic gradient code is correct. If you are having problems, set the seed of the random number generator with the command `rng` to ensure at each test `W` and `b` have the same values and double/triple check that you have a correct implementation of the gradient equations from the lecture notes.

When you have verified the gradient of the cross-entropy loss is okay then you should upgrade `ComputeGradsWithTorch` to also include the L_2 regularization term on the weight matrix and compute the gradient

w.r.t. the cost `J`. Thus you will need to pass the value for the `lam` parameter, compute the `cost` and compute the backward pass relative to the `cost` as oppose to the `loss` i.e.

```
cost = loss + lam * torch.sum(torch.multiply(W, W))
cost.backward()
```

8. Once you have the gradient computations debugged you are now ready to write the code to perform the mini-batch gradient descent algorithm to learn the network's parameters where the updates are defined in equations (8, 9). You have a couple of parameters controlling the learning algorithm (for this assignment you will just implement the most vanilla version of the mini-batch gradient descent algorithm, with no adaptive tuning of the learning rate or momentum terms):

   - `n_batch` the size of the mini-batches
   - `eta` the learning rate
   - `n_epochs` the number of runs through the whole training set.

   As the images in the CIFAR-10 dataset are in random order, the easiest to generate each mini-batch is to just run through the images sequentially. Let `n_batch` be the number of images in a mini-batch. Then for one epoch (a complete run through all the training images), you can generate the set of mini-batches with this snippet of code:

   ```
   for j in range(n/n_batch):
      j_start = j*n_batch
      j_end = (j+1)*n_batch
      inds = j_start:j_end
      Xbatch = Xtrain[:, inds]
      Ybatch = Ytrain[:, inds]
   ```

   A slight upgrade of this default implementation is to randomly shuffle your training examples before each epoch. One efficient way to do this is via the command `rng.permutation(n)` which when given the input `n` returns a vector containing a random permutation of the integers `0, ..., (n-1)`. I suggest the mini-batch learning function has these inputs and outputs

   ```
   trained_net = MiniBatchGD(X, Y, GDparams, init_net, lam)
   ```

   where `X` contains all the training images, `Y` the labels for the training images, `init_net` is a dictionary with the keys `'W'`, `'b'` containing the initial values for the network's parameters, `lam` is the regularization factor in the cost function and

   - `GDparams` is dictionary containing the parameter values `n_batch`, `eta` and `n_epochs`

   The output is a dictionary containing the values of the network's trained parameters. It should have the same keys as `init_net`. Note

`Python` rarely makes explicit copies of variables. Thus if you want to ensure `init_net` is kept unchanged after calling `MiniBatchGD` you should at the beginning of this function make a deep copy of `init_net`, that is:

```
trained_net = copy.deepcopy(init_net)
```

One other tip is that you may also want to pass the random generator `rng` to the function `MiniBatchGD` if you shuffle the data order after each epoch then you can make this experiment repeatable by setting the seed beforehand.

For my initial experiments I set `n_batch=100`, `eta=.001`, `n_epochs=20` and `lam=0`. To help you debug I suggest that after each epoch you compute the cost function and print it out (and save it) on all the training data. For these parameter settings you should see that the training cost decreases for each epoch. After the first epoch my cost score on all the training data was `1.972` where I had set the random number seed generator to `42` and I had initialized the weight matrix before the bias vector. In figure 8 you can see the training cost score when I run these parameter settings for 40 epochs. The cost score on the validation set is plotted in red in the same figure.

(Note: in `Tensorflow` and other software packages they count in the number of update steps as opposed to the number of training epochs. Given `n` training images and mini-batches of size `n_batch` then one epoch will result in `n/n_batch` update steps of the parameter values. Obviously, the performance of the resulting network depends on the number of update steps and how much training data is used. Therefore to make fair comparisons one should make sure the number of update steps is consistent across runs - for instance if you change the size of the mini-batch, number of training images, etc.. you may need to run more or fewer epochs of training.)

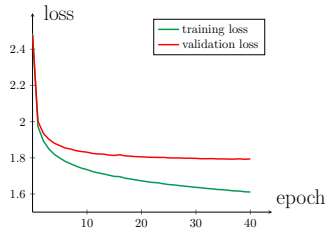


Figure 3: The graph of the training and validation loss computed after every epoch. The network was trained with the following parameter settings: `n_batch=100`, `eta=.001`, `n_epochs=40` and `lam=0`.

When you have finished training you can compute the accuracy of

your learnt classifier on the test data. My network achieves (after 40 epochs) an accuracy of `39.13%` (with a random shuffling of the training example at the beginning of each epoch). This performance is much better than random but not great. Hopefully, we will achieve improved accuracies in the assignments to come when we build and train more complex networks.

After training you can also visualization the weight matrix `W` as an image and see what *class templates* your network has learnt. Figure 4 shows the templates my network learnt. Here is a code snippet to re-arrange each row of `W` (assuming `W` is a `10×d` matrix) into a set of images that can be displayed:

```
Ws = trained_net['W'].transpose().reshape((32, 32, 3, 10), order='F')
W_im = np.transpose(Ws, (1, 0, 2, 3))
for i in range(10):
      w_im = W_im[:, :, :, i]
      w_im_norm = (w_im - np.min(w_im)) / (np.max(w_im) - np.min(w_im))
```

Then you can use `plt.imshow` to display each `w_im_norm` and you can save the image with the command `plt.imsave`.

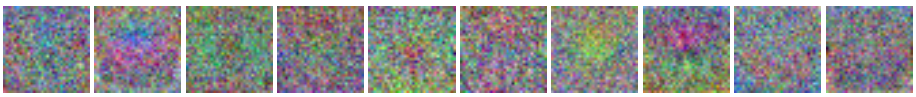


Figure 4: The learnt `W` matrix visualized as class template images. The network was trained with the following parameter settings: `n_batch=100`, `eta=.001`, `n_epochs=40` and `lam=0`.

To complete the assignment:

In the DD2424 Canvas Assignment 1 page upload:

- The final (and cleaned up) version of the code you have written for the assignment. If you have written the code in multiple files please place them in one file for the uploaded version. This will make life easier for me and the TAs! Note we will not run your code.
- A pdf document containing a short report. Please **do not zip** your code and pdf document into on file and upload the zip file. Then the graders have to download the zip file and uncompress it and this is time consuming. Here you should state whether you successfully managed to write the functions to correctly compute the gradient analytically, what tests you ran to check against the numerically computed gradient and the results of these tests. You should include the following plots/figures

1. Graphs of the loss and the cost function (when `lam = 0`, these two quantities are the same) on the training data and the validation data after each epoch of the mini-batch gradient descent algorithm.
2. Images representing the learnt weight matrix after the completion of training.

for the following parameter settings

- `lam=0`, `n_epochs=40`, `n_batch=100`, `eta=.1`
- `lam=0`, `n_epochs=40`, `n_batch=100`, `eta=.001`
- `lam=.1`, `n_epochs=40`, `n_batch=100`, `eta=.001`
- `lam=1`, `n_epochs=40`, `n_batch=100`, `eta=.001`

You should also report the final test accuracy your network achieves after each of these training runs. You should also make a short comment on the effect of increasing the amount of regularization and the importance of the correct learning rate.

Exercise 2: *Optional for Bonus Points*

1. **Improve performance of the network** It would be interesting to discover if it is possible to improve on the performance achieved in the first part of the assignment. Here are some tricks/avenues you can explore to help bump up performance (most of the tricks you can use for higher capacity networks and they will probably have much more of an effect for the higher capacity networks where over-fitting is more of a problem).:
   (a) Use all the available training data for training (all five batches minus a small subset of the training images for a validation set). Decrease the size of the validation set down to ~ 1000.
   (b) Augment your training data by flipping an image horizontally with a .5 probability each time you encounter it during training. You can implement flipping efficiently by pre-computing the indices of the entries of the image vector that you have to switch. Given a Cifar-10 image stored as a column vector, `xx`, of size 3072×1 then one can flip the image horizontally with

```
xx_flipped = xx[inds_flip]
```

   where

```
aa = np.int32(np.arange(32)).reshape((32, 1))
bb = np.int32(np.arange(31, -1, -1)).reshape((32, 1))
vv = np.tile(32*aa, (1, 32))

ind_flip = vv.reshape((32*32, 1)) + np.tile(bb, (32, 1))
inds_flip = np.vstack((ind_flip, 1024+ind_flip))
inds_flip = np.vstack((inds_flip, 2048+ind_flip))
```

   Training with this augmentation should at the very least reduce the amount of over-fitting (gap between training and test performance). Also you should probably reduce the amount of L_2 regularization you apply by reducing the value of λ when you combine it with data-augmentation.

(c) Do a grid search to find *good* values for the amount of L_2 regularization, the learning rate and the batch size. There is some empirical evidence that training with smaller batch sizes when using SGD training leads to better generalization. Though there is also more empirical evidence that with larger batch sizes you can increase the learning rate and not suffer as much with over-fitting... However, due to the simplicity of the model not too much over-fitting will be observed in any case.

(d) Play around with decaying the learning rate by a factor of 10 after every nth epoch or when the validation seems to plateau. This is known as step decay.

Bonus Points Available: 1 point (if you complete at least 3 improvements - you can follow my suggestions and/or think of your own or some combination.)

2. **Train network - multiple binary cross-entropy losses**

In the assignment you just completed the softmax operation was used to turn the output scores $\mathbf{s}$ into a probability vector $\mathbf{p}$. This operation ensures the entries of $\mathbf{p}$ sum to one and explicitly assumes that only one class is present in the image. However, this is not the only option to turn $\mathbf{s}$ into a vector of probabilities $\mathbf{p}$. One can instead interpret that each entry p_i in $\mathbf{p}$ should correspond to the probability that class i is present in the image and is independent of the probabilities for the other classes. In this interpretation each $0 \leq p_i \leq 1$ for $i = 1, \ldots, K$, but there is no constraint that the p_i's sum to one. When this weaker assumption is made, the sigmoid function, $\sigma : \mathbb{R} \to [0, 1]$:

$$\sigma(s) = \frac{1}{1 + \exp(-s)} = \frac{\exp(s)}{\exp(s) + 1} \tag{12}$$

is usually used to map a score to a number between 0 and 1. The sigmoid is applied element wise to $\mathbf{s}$

$$\mathbf{p} = \sigma(\mathbf{s}) \tag{13}$$

Given this interpretation of the probability vector, it is usual to use K binary cross-entropy losses for training the network. For an input $\mathbf{x}$ with one-hot encoding, $\mathbf{y}$, of its label this multiple binary cross-entropy loss is defined as:

$$l_{\text{multiple bce}}(\mathbf{x}, \mathbf{y}) = -\frac{1}{K} \sum_{k=1}^{K} \left[(1 - y_k) \log(1 - p_k) + y_k \log(p_k) \right] \tag{14}$$

For this bonus point assignment you should train your network, with the softmax operation replaced by the sigmoid function, using the multiple binary cross-entropy loss. Check what the new gradient is and if you have to change your code or not. In the report:

- Please write down the expression for $\partial l_{\text{multiple bce}} / \partial \mathbf{s}$

- Record the final test accuracy you achieve training with this loss where a classification prediction is still based on the highest output probability as defined in equation (4). Note for training you may have to increase the learning rate because of the $1/K$ factor in the loss.
- For the test data make a histogram plot of the probability for the ground truth class for the examples correctly and those incorrectly classified. Check if there is a qualitative difference between these histograms when training is performed with the new training procedure versus the softmax + cross-entropy training. Also check if there is more or less over-fitting by looking at the training and validation loss plots.

Bonus Points Available: 2 points

To get the bonus point(s) you must upload the following to the Canvas page *Assignment 1 Bonus Points*:

1. Your code.
2. A Pdf document which
   - reports on your trained network with the best test accuracy, what improvements you made and which ones brought the largest gains. (Exercise 2.1)
   - compares the test accuracy of the network trained with the multiple binary cross-entropy loss compared to the cross-entropy loss and if more or less over-fitting occurs with this loss compare to the cross-entropy loss (Exercise 2.2).