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:

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

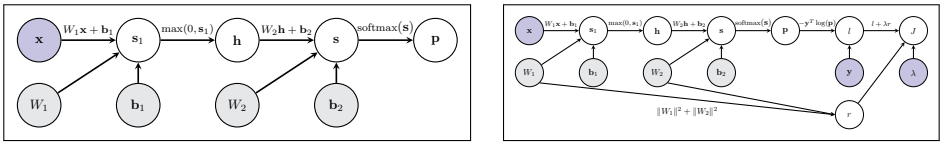
where the matrix W_1 and W_2 have size $m \times d$ and $K \times m$ respectively and the vectors $\mathbf{b}_1$ and $\mathbf{b}_2$ have sizes $m \times 1$ and $K \times 1$. SOFTMAX is defined as

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

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. 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 notation we group the parameters of



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.

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 as the label is encoded by a one-hot representation then the cross-entropy loss $-\mathbf{y}^T \log(\mathbf{p}) = -\log(p_y)$.) 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 cyclic learning rates.

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 $k = 1, 2$:

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

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

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. Once again I would advise you to implement the *Matlab* efficient version as it results in significant speed ups.

Background 2: *What learning rate to use with SGD?*

There is not really one optimal learning-rate when training a neural network with vanilla mini-batch gradient descent. Choose a too small learning rate and training will take too long and too large a learning rate may result in training diverging. Ideally, one should have an adaptive learning rate which changes to match the local shape of the cost surface at the current estimate of the network's parameters. Many variants of mini-batch training try to achieve this - ADAM, mini-batch with a momentum term etc. - and these variants are covered in the lectures. For this assignment though we will explore the rather recent idea of exploiting *cyclical learning rates* [Smith, 2015] as this approach eliminates much of the trial-and-error associated with finding a good learning rate and some of the costly hyper-parameter optimization over multiple parameters associated with training with momentum. It also empirically seem to work well when training relatively small networks as in these assignments. The main idea of *cyclical learning rates* is that during training the learning rate is periodically changed in a systematic fashion from a small value to a large one and then from this large value back to the small value. And this process is then repeated again and again until training is stopped. See figure 2 for an illustration of a typical example of how the learning rate is scheduled to change periodically during training. This is the schedule you will implement.

Assume that you have defined a minimum $\eta_{\min}$ and a maximum $\eta_{\max}$ learning rate. $\eta_{\min}$ and $\eta_{\max}$ define the range of learning rates where learning occurs without divergence. (Note, in general, these values will be affected by λ, network architecture and parameter initialization.) Let η_t represent the learning rate at the tth update step. One complete cycle will take $2n_s$ update steps, where n_s is known as the *stepsize*. When $t = 2ln_s$ then $\eta_t = \eta_{\min}$ and when $t = (2l + 1)n_s$ then $\eta_t = \eta_{\max}$ for $l = 0, 1, \ldots$. A rule of thumb is to set $n_s = k \lfloor n/n_{\text{batch}} \rfloor$ with k being an integer between 2 and 8 and n is the total number of training examples and n_{batch} in the number of examples in a batch. For a triangular learning rate schedule have

1. At iteration t of training if $2ln_s \leq t \leq (2l + 1)n_s$ for some $l \in$

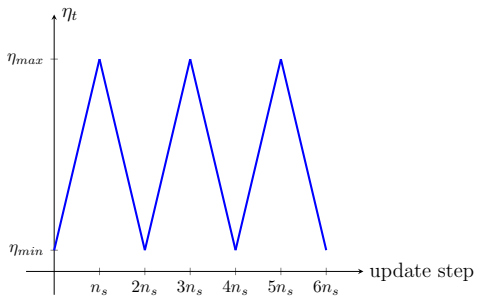


Figure 2: **Schedule for the cyclic learning rate.** The graph above plots the learning rate η_t at each update step. Initially $\eta_1 = \eta_{\min}$ and its value increases linearly until it has a maximum value of $\eta_{\max}$ when $t = n_s$. Then the η_t decreases linearly until it has a value of $\eta_1 = \eta_{\min}$ again when $t = 2n_s$. The cycle can be repeated as many times as one likes and in this example it is repeated three times. For most practical applications the number of cycles is ≥ 2 and ≤ 10. The positive integer n_s is known as the stepsize and is usually chosen so that one cycle of training corresponds to a multiple of epochs of training.

$\{0, 1, 2, \ldots\}$ set

$$\eta_t = \eta_{\min} + \frac{t - 2ln_s}{n_s} (\eta_{\max} - \eta_{\min}) \tag{14}$$

while if $(2l + 1)n_s \leq t \leq 2(l + 1)n_s$ for some $l \in \{0, 1, 2, \ldots\}$ set

$$\eta_t = \eta_{\max} - \frac{t - (2\,l + 1)n_s}{n_s} (\eta_{\max} - \eta_{\min}) \tag{15}$$

2. Update the current estimate of the parameters with

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \left. \frac{\partial J}{\partial \boldsymbol{\theta}} \right|_{\boldsymbol{\theta}_{t-1}}$$

Normally training is run for a set number of complete cycles and is stopped when the learning rate is at its smallest, that is $t = 2ln_s$ for some $l \geq 2$. For this assignment I will give you values for $\eta_{\min}$, $\eta_{\max}$ and n_s that work well for the default network used in the assignment. You can read [Smith, 2015], the paper which forms the basis for this assignment, to get guidelines and tests about how to set $\eta_{\min}$ and $\eta_{\max}$.

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` for training, the file `data_batch_2` for validation and the file `test_batch` for

testing. You have already written a function for Assignment 1 to read in the data and pre-process it. For this assignment we will apply the same pre-processing as before. Remember you should transform it to have zero mean. If `trainX` is the $d \times n$ image data matrix (each column corresponds to an image) for the training data then

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

Then you should normalize the training, validation and test data with respect to this mean and standard deviations. If `X` is an $d \times n$ image data matrix then you can normalize `X` as

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

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 a `list` to store these weight matrices and bias vectors within a dictionary, that is if `L` is the number of layers then like where :

```
net_params = {}
net_params['W'] = [None] * L
net_params['b'] = [None] * L
```

To set the initial values of `net_params['b'][i]` and `net_params['W'][i]` for `i=0,...,(L-1)`, 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 `1/sqrt(d)` for layer 1 and `1/sqrt(m)` for layer 2. You can see the instructions from Assignment 1 to how how to do this. You should probably write a separate function to initialize the parameters as you will be initializing your network frequently when you perform a grid search for a good value of `lambda`.

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 weight and bias parameters. As before I suggest you reuse much of your code from `Assignment1.py` 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 **p** values and

the intermediary activation values. Once again I would recommend using a container such as a dictionary to store this forward-pass data so it can be easily passed to the backward pass function to compute the gradients. For the sample code in this document I refer to this dictionary as `fp_data`.

- 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 the file `torch_gradient_computations.py` from the Canvas page. This file contains skeleton code to compute the gradients via torch. The lines missing are those that compute the scores for input data `X` corresponding to equations (1-3). You need to use `torch` operations to compute them instead of `numpy` ones. To help you out here is one way to compute the ReLu function on each entry of the `m`×`n` torch array `H`:

```
apply_relu = torch.nn.ReLU()
H = apply_relu(H)
```

The number of training examples in `X` is large as is the input dimension. To avoid numerical issues 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 = 5
n_small = 3
m = 6
lam = 0
small_net['W'][0] = (1/np.sqrt(d_small))*rng.standard_normal(size = (m, d_small))
small_net['b'][0] = np.zeros((m, 1))
small_net['W'][1] = (1/np.sqrt(m))*rng.standard_normal(size = (10, m))
small_net['b'][1] = np.zeros((10, 1))
X_small = trainX[0:d_small, 0:n_small]
Y_small = trainY[:, 0:n_small]
fp_data = ApplyNetwork(X_small, small_net)
my_grads = BackwardPass(X_small, Y_small, fp_data, small_net, lam)
torch_grads = ComputeGradsWithTorch(X_small, train_y[0:n_small], small_net)
```

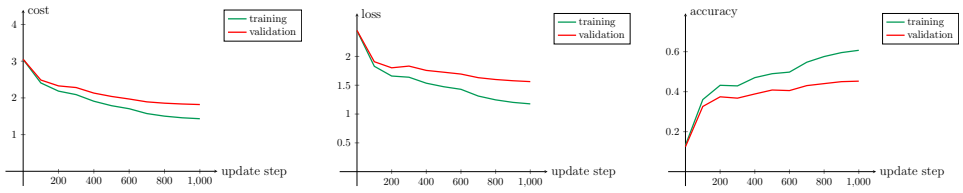
After you have computed the gradients (with no regularization) via your code and `PyTorch` you should check they have produced the same output and follow the guidelines described in the first assignment. You can add in L_2 regularization when you are convinced all the loss gradients are correct in both your code and the `pytorch` code.

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 (`lam`=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: *Train your network with cyclical learning rates*

Up until now you have trained your networks with vanilla mini-batch gradient descent. To help speed up training times and avoid time-consuming searches for good values of η you will now implement mini-batch gradient descent training where the learning rate at each update step is defined by equations (14) and (15) and where you have set `eta_min = 1e-5`, `eta_max = 1e-1` and `n_s=500` and the batch size to `100`. To help you debug, figure 3 shows the training and validation loss/cost I achieved when `lam=.01` and I ran training for one cycle that is from `t=1` until `t=2*n_s`. Once you have convinced yourself that you have a bug free implementation of the cyclic scheduled learning rate then you are ready to somewhat *optimize* the performance of your network.



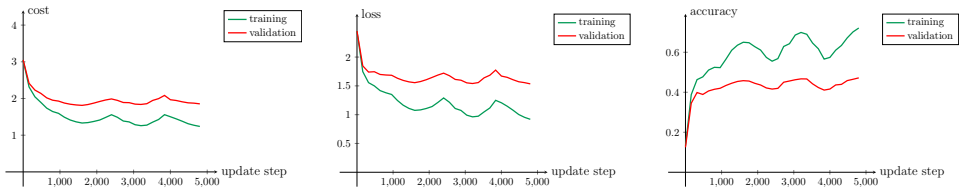
Cost plot **Loss plot** **Accuracy plot**

Figure 3: **Training curves (cost, loss, accuracy) for one cycle of training.** In this example one batch of the training data is used. The hyper-parameter settings of the training algorithm are `eta_min = 1e-5`, `eta_max = 1e-1`, `lam=.01` and `n_s=500`. The last parameter setting implies, as the batch size is 100, one full cycle corresponds to 10 epochs of training. In this simple example only one cycle of training is performed, but already at the end of training a test accuracy of `46.29%` is achieved. Please note my curves are relatively smooth because I plot the loss/cost/accuracy scores 10 times per cycle as opposed to plotting these quantities at every update step.

Exercise 4: *Train your network for real*

Now you should run your training algorithm for more cycles (say 3) and for a larger `n_s=800`. For reference the performance curves I obtained with these parameter settings are shown in figure 4. I measured my performance on the whole training and validation set 9 times per cycle. At the moment

you have not optimized the value of regularization term `lam` at all.



Cost plot **Loss plot** **Accuracy plot**

Figure 4: **Training curves (cost, loss, accuracy) for three cycles of training.** In this example one batch of the training data is used. The hyper-parameter settings of the training algorithm are `eta_min = 1e-5`, `eta_max = 1e-1`, `lam=.01` and `n_s=800`. In the loss and accuracy plots you can clearly see how the loss and accuracy vary as the η_t varies. After the three cycles of training a test accuracy of `48.11%` is achieved.

Coarse-to-fine random search to set `lam`. 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 `lam` varies. You should first perform a coarse search over a very broad range of values for `lam`. To perform this search you should use most of the training data available and the rest for the validation. This is because more data and increasing the value of `lam` are both forms of regularization. When you have less training data you will need a higher `lam` and when you have more training data you will need a lower `lam`. Thus you should perform your search using the same ballpark amount of data that you will use when you train your final network. Thus for this part of the assignment you should load all 5 training batches and use all for training except for `5000` images that should be used as your validation set. When you train each network you should only run 2 cycles (1 cycle could also potentially work) of training and you should set `n_s = 2 * np.floor(n / n_batch)` to get a good idea of performance for a given `lam`. Search for `lam` on a log scale, for example to generate one random sample for the learning rate uniformly in the range of `10^l_min` to `10^l_max`

```
l = l_min + (l_max - l_min) * rng.random()
lam = 10**l;
```

In my experiments for the coarse search I set `l_min=-5` and `l_max=-1` and actually used a uniform grid with eight different values. 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. Next perform a random search but with your search adjusted to a narrower range focused on the good settings found in the coarse setting and possibly run training for a few more cycles 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 best `lam` found, and then train the network using most of the training data, for more cycles and for a larger `n_s` and see what final performance you get on the test set. You should be getting performances of >`50%` for your good settings given ≥ 2 cycles of training (or even just one cycle of training). For reference I was able to train networks that achieved test accuracies $\sim 52\%$ without too much exhaustive searching.

To complete the assignment:

To pass the assignment you need to upload to Canvas:

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. Give evidence for these conclusions.
   - ii) The curves for the training and validation loss/cost when using the cyclical learning rates with the default values, that is replicate figures 3 and 4. Also comment on the curves.
   - iii) State the range of the values you searched for `lam`, the number of cycles 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 `lam`, the number of cycles 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 `lam` 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 $\sim$3 cycles. Plot the training and validation loss plots and then report the learnt network's performance on the test data.

Exercise 5: *Optional for bonus points*

For Assignment 2 I will award at most 5 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) Explore whether having significantly 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.

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

   (c) Apply data augmentation during training - random mirroring as described in the bonus part of assignment and also random translations. As a hint if you translate your image `xx` by a positive integer translations `tx` and `ty` then by computing the following indices (conceptually not hard but a little tricky to get right):

   ```
   aa = np.arange(32).reshape((32, 1))
   vv = np.tile(32*aa, (1, 32-tx))
   bb1 = np.arange(tx, 32, 1).reshape((32-tx, 1))
   bb2 = np.arange(0, 32-tx, 1).reshape((32-tx, 1))

   ind_fill = vv.reshape((32*(32-tx), 1)) + np.tile(bb1, (32, 1))
   ii = np.transpose(np.nonzero(ind_fill > ty*32+1))
   ind_fill = ind_fill[ii[0,0]:]

   ind_xx = vv.reshape((32*(32-tx), 1)) + np.tile(bb2, (32, 1))
   ii = np.transpose(np.nonzero(ind_xx < 1024-ty*32))
   ind_xx = ind_xx[0:ii[-1, 0]+1]

   inds_fill = np.vstack((ind_fill, 1024+ind_fill))
   inds_fill = np.vstack((inds_fill, 2048+ind_fill))

   inds_xx = np.vstack((ind_xx, 1024+ind_xx))
   inds_xx = np.vstack((inds_xx, 2048+ind_xx))
   ```

   and applying them to produce the shifted image

   ```
   xx_shifted[inds_fill] = xx[inds_xx]
   ```

   Note the code above has to be changed a bit if either `tx` or `ty` is negative. So please visualize your before and after images to ensure there are no bugs. Even if you pre-compute the indices for all the different (`tx, ty`) pairs given `-3 <= tx <= 3` and `-3 <= ty <= 3` there will be a slow down in your training if you apply a shift randomly for every image in your batch, as there will be accessing different parts of memory. So perhaps use this augmentation somewhat judiciously to avoid too much of a slow down.

   (d) In the basic assignment cyclical learning rates with sgd was the optimizer used. Potentially using Nesterov Momentum with straightforward linear decay on the learning rate could perform just as well or better given a fixed number of update steps could potential outperform the basic assignment. (See Budgeted Training:Rethinking Deep Neural Network Training Under Resource Constraints, M, Li, E. Yumer, and D. Ramanan, ICLR 2020 for

evidence. Their default implementation has: base learning rate 0.1, momentum 0.9, weight decay 0.0005 and a batch size 128, but this is for a ResNet as opposed to a fully connected network.)

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

2. Given that you have implemented data-augmentation and defined a network with more hidden nodes, do some semi-extensive testing (amount of l2 regularization, number of cycles used for training, annealing of the max learning rate as you train, have a long training run etc...) to see what level of performance you can get with a *wide* 2-layer network and data-augmentation.

   Bonus Points Available: 2 points

3. Cyclical learning rates are good for fast training. But for your 2-layer network would an Adam optimizer perform just as well or better? For these bonus points implement an Adam optimizer. Then do some testing (amount of l2 regularization and data-augmentations, have a long training run etc...) to see what level of performance you can achieve by training a *wide* 2-layer network with data-augmentation and an Adam optimizer. I would recommend using the suggested settings for Adam's parameters.

   Bonus Points Available: 3 points

To get the bonus point(s) you must upload the following to the Canvas assignment page *Assignment 2 Bonus Points*:

1. Your code.
2. You can get at most 5 points for Assignment 2.
3. 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 (if any!). (Exercise 5.1)
   - Summarizes the training and search you completed and the final test accuracies you achieve. (Exercise 5.2)

References

[Smith, 2015] Smith, L. N. (2015). Cyclical learning rates for training neural networks. *arXiv:1506.01186 [cs.CV]*.