As you begin to fit larger models you need to apply more regularization to allow longer training runs without over-fitting. These longer runs are required to train the best possible version of your network! In this training scenario a simple and complimentary approach to other regularization techniques such as data-augmentation, weight decay, is *label smoothing*. Label smoothing works as follows. Assume you have a labelled training example (X, y) where $y \in \{1, \ldots, K\}$. Let $\mathbf{y}$ represent the one-hot encoding of label y. This is the usual target output for cross-entropy training. However, this target vector does not necessarily need to be a one-hot encoded vector. One can instead spread some of the probability from the ground truth class to the other classes. Let ϵ be small number in $[0, 1)$. The ith entry of the label smoothed target vector $\mathbf{y}_{\text{smooth}}$ is defined by

$$y_{\text{smooth},i} = \begin{cases} 1-\epsilon & \text{if } i = y \\ \epsilon/(K-1) & \text{otherwise} \end{cases} \tag{23}$$

Typically we set $\epsilon = .1$. Once you have defined $\mathbf{y}_{\text{smooth}}$ then you only have to make a minimal change in your training algorithm. In the backward-pass to compute the gradients when you propagate the gradient through cross-entropy loss and softmax operations you should make this replacement:

$$-(\mathbf{y} - \mathbf{p}) \qquad \Longrightarrow \qquad -(\mathbf{y}_{\text{smooth}} - \mathbf{p}) \tag{24}$$

Thus label smoothing also has the benefit that it has minimal computational overhead per update iteration. Though, of course, similar to over regularization techniques if applied then longer training is required as you have made the training task harder.

Background 6: *Cyclical learning rates with increasing step sizes*

In Assignment 2 you trained with cyclical learning rates. For most of the experiments you will use this optimizer again for this assignment. However, you will apply a small *upgrade* to this algorithm for longer training runs to help with efficiency, that is we will have shorter cycles at the start of training. The upgrade is that the number of update steps per cycle is doubled after each cycle. Let $n_{i,s}$ be the number of steps for the ith cycle then

$$n_{i+1,s} = 2\, n_{i,s} \tag{25}$$

This schedule of the learning rate is an approximation to the cosine with warm restarts schedule of [Loshchilov and Hutter, 2017]. For an even more accurate approximation you should also decay $\eta_{\max}$ for each new cycle but in the basic assignment we will not do this.

Exercise 1: *Write code to implement the convolution efficiently*

For this assignment to be a success you must have a bug free and fast implementation of the convolution applied at the first layer. To ensure this happens you will write slow but straightforward code as your first implementation of the convolutional layer - a dot-product between the convolution filter and each non-overlapping sub-patch of the input image. I have provided debugging information on the Canvas assignment page to ensure a bug free implementation. Read in the file `debug_info.npz` and extract the image data `X` and the convolution filter `Fs`:

```
debug_file = 'debug_conv_info.npz'
load_data = np.load(debug_file)
X = load_data['X']
Fs = load_data['Fs']
```

`X` contains the image data from 5 `cifar10` images and has size `3072`×`n` where `n=5` and `Fs` has size `f`×`f`×`3`×`nf` where `f=4` and `nf=2`. The images in `X` have been flattened and to convert them back into images of size `32`×`32`×`3`×`n` apply

```
X_ims = np.transpose(X.reshape((32, 32, 3, n), order='F'), (1, 0, 2, 3))
```

For each image `X_ims[:, :, :, i]` an easy way to compute the convolution applied with stride `f` between `X_ims[:, :, :, i]` and the `k`th filter `F[:, :, :, k]` is to have nested `for` loops to visit each sub-patch. At each sub-patch compute the dot product between the sub-patch of `X_ims[:, :, :, i]` of size `f`×`f`×`3` and `F[:, :, :, k]`. The dot-product can be computed the with `numpy` commands of `np.multiply` and `np.sum`. There are `32/f` × `32/f` sub-patches to visit. Write a function which for each image, each sub-window and each filter computes all the appropriate dot-products and puts the result in a 4d array of size `32/f`× `32/f`×`nf`×`n`. To debug you should compare your result to `load_data['conv_outputs']`.

Once your code produces the correct numbers, it will act as your *ground truth* code and you can begin to write a more efficient implementation via matrix multiplication. This more efficient implementation first requires constructing the multi-dimensional array `MX` of size `n_p` × `f*f*3` × `n` which contains the M_X defined as in equation (10), for each input image `X_ims[:, :, :, i]`. To construct `MX` you should allocate the memory by initializing an array of zeros of appropriate size. Then for each image `X_ims[:, :, :, i]`, you should iterate through each non-overlapping sub-region (as previously), let `X_patch` be the pixel data extracted from the `l`th sub-region, and set:

```
MX[l, :, i] = X_patch.reshape((1, f*f*3), order='C')
```

I visited the sub-patches in the order where the second dimension of `X_ims[:, :, :, i]` changes first and then the first dimension. This is to be consistent with how I flatten `load_data['conv_outputs']`. Importantly, `MX` just needs to be computed once at the beginning of training and stored. When training `MX` will be accessed continually and it does not change during training as it is completely defined by the input data and the size and stride of the convolution filters and these quantities do not change.

You are almost ready to compute the convolution as a matrix multiplication. First, though you have to flatten each filter via:

```
Fs_flat = Fs.reshape((f*f*3, nf), order='C')
```

where `Fs_flat` has size `f*f*3`$\times$`nf`. The convolutions can now be computed for each image as a matrix multiplication see in equation (9)

```
for i in range(n):
   conv_outputs_mat[:, :, i] = np.matmul(MX[:, :, i], Fs_flat)
```

where `conv_outputs_mat` storing the result has size `n_p` $\times$ `n_f` $\times$ `n`. You should compare this output to that from your dot-product implementation. First, thought you have to reshape `conv_outputs` to have the same shape as `conv_outputs_mat` that is

```
conv_outputs_flat = conv_outputs.reshape((n_p, nf, n), order='C')
```

The last step to make the convolution computations even faster is to remove the `for` loop over images using an Einstein summation:

```
conv_outputs_mat = np.einsum('ijn, jl ->iln', MX, Fs_flat, optimize=True)
```

You should check this produces the same output as before. The moment it does you can feel like a deep learning guru! Using `einsum` is hardcore! It is okay if you do not fully understand the workings of the `einsum` operator, to be honest I just did some pattern matching and interpolated to our use case, but more details are available at the numpy einsum help page. For our test case `MX` is small and only contains data from 5 images so there is probably no speed up using the Einstein summation. But during training `MX` will contain data from a training batch of ~ 100 images and on my laptop (M1 Macbook pro) machine the `einsum` implementation gives a > 3 times speed up over the `for` loop implementation.

Exercise 2: *Compute gradients*

With the fast convolution computations in place, you are ready to tackle writing the code for the bare bones of the back-prop algorithm. First you have to write the *forward pass* and then the *backward pass*. Initially, you will check your gradient computations using debugging information provided. But after clearing these checks you will clean up your code, add bias terms to your network function and use `pytorch` to do a more thorough job of debugging your gradient calculations.

Forward Pass You are ready to write the forward pass of the back-prop algorithm for the network described in equations (1 - 5). The function you write should have as input the `MX` representation of the input data and the parameters of your network. These parameters are the convolution filters of the first layer and the weight matrices for the two subsequent layers. You have written the code to apply the first convolutional layer. Then you have to apply the ReLu activation function to the output of the convolution and then reshape the output array so you can apply the first fully connected layer. This corresponds to:

```
conv_flat = np.fmax(conv_outputs_mat.reshape((n_p*nf, n), order='C'), 0)
```

(Note I don't think this code vectorizes the output of the convolution in the same order as equation (13). This does not matter as the critical issue is that the ordering is consistent within your code and this is why I'm being very explicit about this in the code description.) Given `conv_flat` you can proceed with the fully connected layers of the network as in Assignment 2 to finally produce the probability vector for each input image. The function you write should return the information needed for the backward pass. These required quantities are the final probabilities and the intermediary outputs at each layer (ie those corresponding to equations (2), (3) and (5).) To help you debug your code the debugging file you loaded previously, which had the parameters for the convolution filters, also contains the arrays for the rest of the network's parameters:

```
load_data['W1'] (size:  nh × n_p*nf),
load_data['W2'] (size:  10 × nh),
load_data['b1'] (size:  nh × 1),
load_data['b2'] (size:  10 × 1)
```

and the values for the arrays your forward-pass should return

```
load_data['conv_flat'] (size:  n_p*nf × n),
load_data['X1'] (size:nh × n),
load_data['P'] (size:  10 × n)
```

My naming convention tries to be consistent with equations (1-5) and hopefully from the context you can figure out to which arrays they correspond.

Backward Pass Next up is the backward pass code. The target labels for the given debug data are in `load_data['Y']` ($10 \times$ `n`). Propagating the gradient from the loss node through the fully-connected layers and to the weight matrices `W2` and `W1` is the same as in assignment 2 though, of course, the input to first fully-connected layer is the output of the convolutional layer as opposed to the original input data. Let `G_batch` be the `n_p*n_f`×`n` array containing the gradient information after back-prop to the `h` node. At this stage in the forward-pass we had just performed a flattening/reshaping operation. We need to undo this operation w.r.t. `G_batch` and obtain an array `G` of size `n_p`×`n_f`×`n`

```
GG = G_batch.reshape((n_p, nf, n), order='C')
```

Given `GG` and `MX` you can now compute the sum in equation (22) and compute the gradient for `Fs_flat`. Write the code for this and you can check your answer against `load_data['grad_Fs_flat']`.

The last step to make your gradient computations fast is to remove the `for` loop over images in equation (22) and replace with an Einstein summation:

```
MXt = np.transpose(MX, (1, 0, 2))
grad_Fs_flat = np.einsum('ijn, jln ->il', MXt, GG, optimize=True)
```

Clean up your code! You have written the essential code for applying the network function and computing the gradients. At this point you probably need to clean up your code and come up with good ways of storing your network's parameters etc., You will also need to write and/or upgrade existing code

- to pre-compute, save and load the `MX` representation for all the training data and test data,
- to initialize the network's parameters etc. (You should use He initialization for both the convolution filters and the fully connected layers.)
- and of course integrate your new forward and backward algorithms into your training.

A few words of advice:

- Regarding `MX` if you are having trouble fitting everything into memory then you should use `np.float32` as the numerical type for each number (input data and network parameters) and to be honest this is the default type used in deep learning. The precision of doubles is not needed.
- As you apply the convolution via matrix multiplication you should store your network's convolutional filters as an array of size `3*f*f` × `nf`.

Check gradient computations To double check your gradient computations are okay, you should upgrade the `torch` gradient code supplied for Assignment 2, to compute the gradients for your new network. Here you should emulate in `torch` the forward pass you have just written and also compute the loss. Use the `for` loop implementation, equation (21), to apply the convolutional filters to each input image to avoid any nuisance complications caused by differences in calling and the ordering of inputs and outputs to `torch`'s and `numpy`'s version of `einsum`. The total number of training examples is large. Thus you should compare your gradients to the `torch` gradients computed on just a small subset of training images, small `nf` ($<$10) and small `nf` ($<$10). After completing this and checking for agreement between the two sets of gradient computations you should:

- Upgrade your network and gradient calculations to also include a bias vector for the convolutional layer. This vector will have `nf` entries.
- Add an L2 regularization for the weights of the fully connected layers and the convolutional filters.

After these upgrades you will need to check the gradient calculations again to ensure you have a bug free implementation.

Exercise 3: *Train small networks with cyclic learning rates*

You are now ready to start training your first ConvNets. For training we will use the cyclic learning rates optimizer as in Assignment 2. Do, if possible, re-use your code. The initial network you will train has architecture:

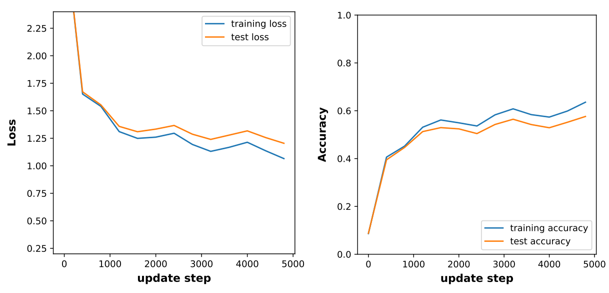
- **Network architecture**: `f=4`, `nf=10`, `nh=50`

with the hyper-parameters for training and L2 regularization set to

- **Cyclic learning rates hyper-parameters**:
  `n_cycles=3`, `step=800`, `eta_min=1e-5`, `eta_max=1e-1`, `n_batch=100`
- **Regularization**: `lam=.003`

With this setting my final model achieved a test performance of ∼`57.61%` and took under 50 seconds to train on a M1 mac book pro when I use `49,000` training examples, see figure 2 for the training curves. Note I had some variation in my training run times. Please train this network and check you can get a similar level of performance in a reasonable running time.

Even if we keep the number of layers of this network fixed, there are too many hyper-parameters w.r.t. number of filters, number of hidden node, size



Loss curves　　　　Accuracy curves

Figure 2: The training and tests curves for the first ConvNet ($f = 4, n_f = 10, n_h = 50$). The final test accuracy achieved for this network is `57.61%` when trained with three cycles of cyclical learning rate and `step=800`. This accuracy probably exceeds most if not all the models you trained in Assignment 1 and 2.

of filters, number of updates applied during training, amount of regularization applied etc, to fully explore the best configuration given your limited computational resources. You will just, therefore, run a small subset of experiments to get some feel for how changing some factors while keeping others fixed impacts test performance. Our initial set of experiments is to see the impact of the filter size, `f`, and the number of filters, `n_f`, on test performance and run time. Given the same training set up as before you should run the following architectures:

- **Network architecture 1**: `f=2`, `nf=3`, `nh=50`
- **Network architecture 2**: `f=4`, `nf=10`, `nh=50`
- **Network architecture 3**: `f=8`, `nf=40`, `nh=50`
- **Network architecture 4**: `f=16`, `nf=160`, `nh=50`

The number of filters, `n_f`, has be chosen to keep the number of outputs from the convolution layer constant (or at least approximately the same) for each network. Remember you have already trained **architecture 2**. Keep a record of the final test performance and also the training time for each architecture and make two bar charts. One show the final test performance of each model and the other displaying the run time.

Train for longer The results you just obtained indicate working with filters of size `f=16` is going to be too slow and with `f=2` we minimize the effect of the convolutional layer. Therefore `f` ∈ {`4, 8`} is probably the best option for our set-up. Now you should investigate the benefit of training the networks with architecture 2 and 3 for longer. You should upgrade the basic cyclic learning rate algorithm to the algorithm with increasing cycle length introduced in background section 6. Set `step_1 = 800`, still with `n_cycles=3` and train the two models. Keep a record of the final test accuracies achieved and the plots of the training and test loss.

As you can see training for more updates improves results! At least one of the networks should have given final test performance of >`60%`. Hurrah and well done! With longer training the bigger jump in performance was for architecture 3. But was this caused by the bigger filter size or that architecture 3 is significantly wider, i.e. it applies more filters at the first layer than architecture 2? Let's get some indication whether layer width is a critical factor. For architecture 2 bump up the number of filters to `n_f=40` and re-run the previous experiment. Keep a record of the final test accuracies achieved and the plots of the training and test loss. As you can see increasing width helps alot. There is lots of anecdotal and empirical evidence that increasing the width of a neural network helps training and capacity. If you are interested check out this paper [Tan and Le, 2019] which introduces the EfficientNet and giving a rule of thumb scaling laws of how best you should increase the width, depth and resolution of the your network architecture if given more resources.

Exercise 4: *Larger networks and regularization with label smoothing*

As training with `f=4` is generally quicker we will now investigate how increasing the capacity of the network by making each layer wider increases the capacity and performance of your network. But, of course, when you begin to train larger models and have longer training times then over-fitting, particularly in relation to the loss, is much more likely. And if you over-fit quickly w.r.t. the loss then it is not possible to train for many update steps. In this part of the assignment you will generate some results to help you get a feel for the issue. But remember we are only barely scratching the surface here. Consider the network:

- **Network architecture 5**: `f=4`, `nf=40`, `nh=300`

Train this network with `n_cycle=4` and `step_1=800` and with L2 regularization `lam=.0025`. (Note this run will take some time $\sim$ 10 minutes for me) We reduce the L2 regularization over-fit more quickly than with a higher L2 regularization. Plot the training and test losses for this architecture, regularization and length of training. It is obvious for this set-up the loss has over-fit. To regularize your model, instead of applying more L2 regularization, implement label smoothing as in background section 5. Re-run the same experiment but with label smoothing. Keep a record of the loss plots plus the final test accuracy achieved and report on the qualitative difference of the loss plots for the two approaches.

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. Please also report the training time for the initial three layer ConvNet you train in Exercise 3 you train.
   ii) The bar charts for the final test accuracy and training times for the 4 network architecture trained with short training runs and varying `f` and `n_f` in Exercise 3.
   iii) The curves for the training and test loss asked for in the **Train for longer** part of Exercise 3. To keep running times down I only computed the test loss and accuracy sparsely for these curves. I computed the performance metrics at every j*`step/2` th update iteration for $j = 0, 1, 2, \ldots$. I did the same for the training loss and used a big chunk of the training data to have a less noisy estimate than the estimate from the batch.
   iv) The curves for the training and test loss for network architecture 5 when no label smoothing and label smoothing is applied. Please comment on the qualitative difference in the curves.
   v) Imagine you want to boost performance of your three layer network even further and you have more available compute than now but it is not unlimited. What would be the next set of experiments you would run and why? Remember the goal here is to investigate factor(s) that you feel, given the experiments from this assignment and the previous assignments, would give the most bang for your buck w.r.t. final test accuracy! There is no perfect answer here I'm just curious to find out what you think would be import architecture and/or training issue to investigate.

Exercise 5: *Optional for bonus points*

For Assignment 3 I will award at most 4 bonus points (the special bonus points are not included in this calculation).

1. **Push performance of the network**. It would be fun to discover how high the performance of Assignment 3's network (a 3-layer network with an initial patchify layer) on CIFAR-10 can be pushed. There

are lots of options to try to make gains! Make the network wider, use data-augmentation, try to find the right balance between the amount of L2 regularization and label smoothing, decay `eta_max` in the cyclical learning rate algorithm, concatenate the output from convolution filters of multiple sizes (just a crazy idea), ... From my not so extensive and slightly haphazard investigations the maximum I achieved was `67.26`%. The *tricks* I used are the ones I describe in assignment! But, in general, data-augmentation and making the network wider is usually a good bet!

Bonus Points Available: 4 points (if you complete at least 3 improvements) - you can follow my suggestions, think of your own or some combination of the two. I'll also award an additional extra bonus point if you get a test accuracy ≥`68`% and two if you achieve ≥`70`%. Note for this assignment because training takes a relatively long time, I'm okay with the test set becoming your validation set! Thus any final result you get will be a slight over-estimate of the true test error. Note for the project such behaviour will not be tolerated!

2. **Compare the speed of your training to using pytorch** Use the `torch` function `torch.nn.functional.conv2d` to compute the convolutions and its auto-diff computations for training where the computations are calculated on the CPU. Please compare the training timings of your implementation versus the `torch` implementation for this network across a variety of filter widths and number of filters. These test do not need to be exhaustive. We just want to get an indication if your implementation is faster/slower than the in-built `pytorch` functions for this particular network and if the speed-up / slow-down is correlated with the filter size etc..

   Bonus Points Available: 3 points

To get the bonus point(s) you must upload the following to the Canvas assignment page *Assignment 3 Bonus Points*:

1. Your code.
2. You can get at most 4 points for Assignment 3.
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 times for pytorch Vs your training code for several network architectures and the general conclusions you can draw from these results. (Exercise 5.2)

References

[Loshchilov and Hutter, 2017] Loshchilov, I. and Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts. In 5th International Conference on Learning Representations. In *Proceedings of the International Conference on Learning Representations (ICLR).*

[Tan and Le, 2019] Tan, M. and Le, Q. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In *Proceedings of the International Conference of Machine Learning (ICML).*