# EECS 442 Computer Vision: Homework 4

## Instructions

- This homework is **due at 11:59:59 p.m. on Wednesday March 18th, 2020**.
- The submission includes two parts:
  1. **To Gradescope:** a `pdf` file as your write-up. Please reading the grading checklist for each part before you submit it.
     You might like to combine several files to make a submission. Here is an example online link for combining multiple PDF files: https://combinepdf.com/.
     **Please mark where each question is on gradescope.**
  2. **To Canvas:** a `zip` file including all your code.
- The write-up must be an electronic version. **No handwriting, including plotting questions.** LaTeX is recommended but not mandatory.

## 1 Optimization and Fitting [20 pts]

For Problem 1, you will work on the same problem as you did in HW3 1.2.6. Instead of using RANSAC to fit a homography, you need to fit an affine transformation

$$\mathbf{y} = \mathbf{S}\mathbf{x} + \mathbf{t}$$

by gradient descent.

1. (15 pts) Implement the Linear layer and L2 loss in `layers.py`. In forwards propagation, you'll store all inputs in `cache` and save it for backwards propagation. The functions are shown below.
   - Linear layer.
     $$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$
     .
   - L2 loss layer.
     $$L(x, \text{label}) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \text{label}_i)^2,$$
2. (5 pts) Implement the gradient descent in `fitting.py`, report the hyperparameters you choose and the $\mathbf{W}$. Include the figure in your report as well.

## 2 Softmax Classifier with One Layer Neural Network [39 pts]

For Problem 2 and Problem 3, you will implement a softmax classifier from scratch to classify images. **You cannot use any deep learning libraries such as PyTorch in this part.**

Implement the ReLU layer and softmax layer in `layers.py`. These functions are shown below:

- ReLU layer.

$$y = \begin{cases} x, & x \geq 0, \\ 0, & \text{otherwise.} \end{cases}$$

- Softmax layer.

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}}$$

$$L(\mathbf{y}, label) = -\sum_{i=1}^{C} [[i == label]] \log(y_i)$$

**Note:** When you exponentiate even large–ish numbers in your softmax layer, the result could be quite large and numpy would return `inf`. To prevent this case, you can calculate softmax layer with max subtraction:

$$y_i = \frac{e^{x_i - max(\mathbf{x})}}{\sum_{j=1}^{C} e^{x_j - max(\mathbf{x})}}$$

It's also not hard to see these two softmax equations are in fact equivalent since

$$\frac{e^{x_i - max(\mathbf{x})}}{\sum_{j=1}^{C} e^{x_j - max(\mathbf{x})}} = \frac{e^{x_i}/e^{max(\mathbf{x})}}{\sum_{j=1}^{C} e^{x_j}/e^{max(\mathbf{x})}} = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}}$$

In `softmax.py`, you need implement your network. You're only allowed to use layer functions you implement. However, you're encouraged to implement additional features in `layers.py` and use it here. Filling in all `TODO`s in skeleton codes will be sufficient.

You'll use L2 regularization in your neural network to prevent overfitting. In order to simplify the expression of gradient, use the formula $\frac{1}{2}\|x\|^2$ for L2 regularization, where the $2$ and the $\frac{1}{2}$ will cancel out when you take the gradient. **Don't include bias term in your regularization**.

After making sure your network works, you need train it on CIFAR–10 [1] dataset, which is available at https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz. CIFAR–10 has 10 classes, 50000 training images, and 10000 test images. You need split the training set/validation set on the training images by yourself. After decompressing the downloaded dataset, you can use the provided python inferface to read the CIFAR–10 dataset in `train.py`, although you're free to modify it. Preprocess your images and tune the model hyperparameters in `train.py`.

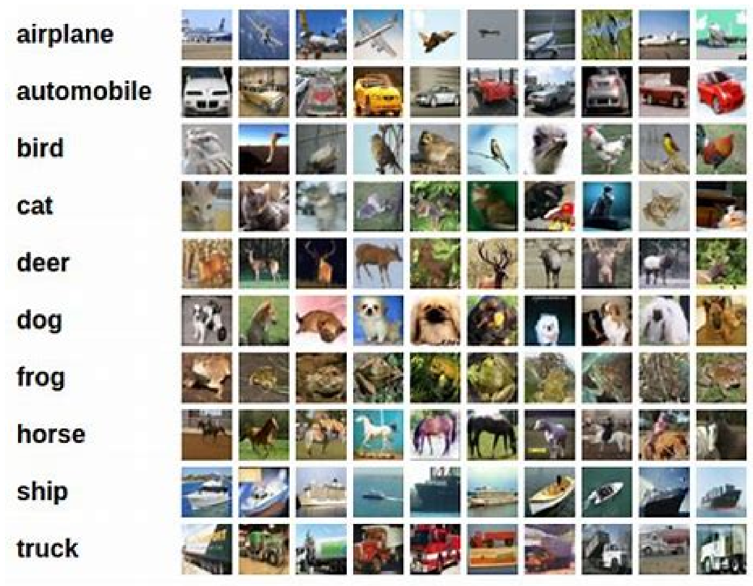

Figure 1: Example images from CIFAR10 dataset [1]

This task is open-ended. However, here are some suggestions:

- When working on the backward function in `layer.py`, you need to calculate the gradients of the objective function. To check your implementation, you can take use of this:

$$\frac{\partial f(\cdots, x_i, \cdots)}{\partial x_i} = \lim_{\delta \to 0} \frac{f(\cdots, x_i + \delta, \cdots) - f(\cdots, x_i, \cdots)}{\delta}$$

  where $f : [x_1, \cdots, x_n] \in \mathbb{R}^n \mapsto \mathbb{R}$. So, you can derive some approximation for the gradients by setting some small value $\delta$. You can compare the gradient your implementation gets with the numerical value of the approximation.

- Image preprocessing is important, you can either normalize them to zero mean and unit standard deviation, or simply scale them to the range [0,1].

- Training neural networks can be hard and slow. Start early!

**Grading checklist:**

1. For all layers, we will have a series of tests that automatically test whether your functions are written correctly. We won't use edge cases. You do not need to include any discussion of your code for part one unless you have decided to implement some extra features.

2. Your report should detail the architecture you used to train on CIFAR–10. Include information on hyperparameters chosen for training and a plot showing both training and validation accuracy across iterations.

3. Report the accuracy on the test set of CIFAR–10. You should only evaluate your model on the test set once. All hyperparameter tuning should be done on the validation set. We were able to get **40%** accuracy on the test set.

4. Include discussion of the parameter you chose for experimentation, and include your results and analysis. In terms of discussion, we expect to see plots and data instead of a general description. You could still get credits if you cannot achieve **40%** accuracy but with a detailed discussion.

5. Please follow the spec and the code instructions strictly. Don't modify the starter code and remember to make your code for all problems could run (remember to uncomment your code in submission if you comment it for debugging). **We'll use an autograder to run you code so make sure you understand the requirement of the API correctly. The submission file should be structured the same as the starter code**. The first step is to make sure the initialization is implemented correctly.

## 3 Softmax Classifier with Hidden Layers [22 pts]

Continue to work on `softmax.py`, add a hidden layer with $N$ dimension on your neural network if `hidden_dim` is set to a positive integer N, there's no hidden layer if `hidden_dim=None`. Use ReLU as your activation function.

Use this model to do the classification as you did in Problem 2 again. We were able to get **50%** accuracy on the test set. Also include the number of hidden dimension in your report. The grading checklist is the same as the one for Problem 2.

Once you finish the training, save the model with highest test accuracy for your next problem, we've already provided `save/load` functions for you in `softmax.py`.

## 4 Fooling Images [19 pts]

Fooling images is a good way to see where the model works and fails, it's highly related to another important field of machine learning: adversarial attack and robustness. In the following example, a few spots on the image will make the model to misclassify this ship image as airplane although it still looks like a ship.

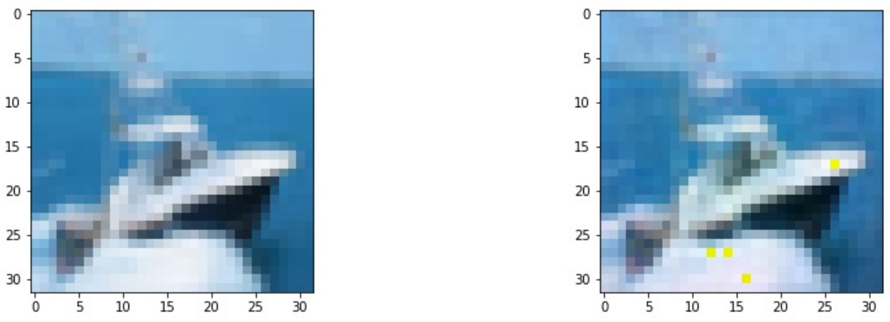

Figure 2: Original ship image (left) and the fooling image classified as airplane (right).

In this part, you will use gradient ascent to generate a fooling image from a correctly classified image from CIFAR-10.

1. (5 pts) Finish the remaining code in softmax.py, return the gradient of loss with respect to the input if `return_dx` flag is True (You function should return a gradient matrix with the same shape as

your input, in part 2 and 3 your function will return a dictionary, in this part only a matrix should be returned).

2. (15 pts) Gradient ascent is similar to gradient descent. You'll fix the parameters of the model and compute the gradient of the classification score with respect to the input image and update the image iteratively. In this problem, you will use a fooling class to compute the gradient of loss with respect to the input image instead, in which way you're doing gradient descent to minimize the loss but gradient ascent to maximize the classification score of that fooling class.

   You will implement the following steps in `fooling_image.py`

   - load the trained model you get from the last problem.
   - load an image that is correctly classified by your model, choose a different class as the fooling class, fix the model parameters and compute the gradient of the loss with respect to your input image.
   - update your input image with the gradient, repeat this process until your model classifies the input image as the fooling class.

   Include the original image, the fooling image and their difference in your report, you can magnify the difference if it's too small. **Comment** on the robustness of your model.

## References

[1] Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

## Acknowledgement

Part of the homework and the starter code are taken from previous EECS442 by David Fouhey and CS231n at Stanford University by Fei-Fei Li, Justin Johnson and Serena Yeung. Please feel free to similarly re-use our problems while similarly crediting us.