Cornell University
CS 5670: Introduction to Computer Vision
Fall 2025. Instructor: Andrew Owens

Problem Set 2: Machine Learning

Posted: Monday, September 15, 2025	**Due:** Monday, September 29, 2025

Please submit your Colab notebook to Gradescope as a `.pdf` file. Please convert your Colab notebook to PDF. For your convenience, we have included the PDF conversion script at the end of the notebook.

The starter code can be found at:
https://colab.research.google.com/drive/1AAXdHBC7C2fgwU0Nw6AO6t5sua79On-8?usp=sharing

Please see **GPU Tips** at the end of this document to better understand how to efficiently use GPUs for this homework set (and all future homeworks that require GPUs). **tldr:** your code will run *very* slow if you run out of GPU hours early and you have to train your models on CPUs. Please follow the tips we provide to avoid this!

We recommend editing and running your code in Google Colab, although you are welcome to use your local machine instead.

Problem 2.1 *Nearest Neighbor Classification*

In this problem, we will implement the k-nearest neighbor algorithm to recognize objects in tiny images. We will use images from CIFAR-10 [2], a small dataset (by today's standards) with 60,000 32×32 color images across 10 classes for image classification (Fig. 1). The code for loading and pre-processing the dataset has been provided for you. For this problem, we will subsample the dataset for less compute time.

Note: There is a `DEBUG` flag in the starter code that you can set to `True` while you are debugging your code. When the flag is set, only 20% of the training set will be loaded, so the rest of the code should take less time to run. However, before reporting the answers to questions, please remember to set the flag back to `False`, and to rerun the cells! There is also an option to run the code with a different image size, which you are welcome to experiment with (again, please set this back to the default before submitting!).

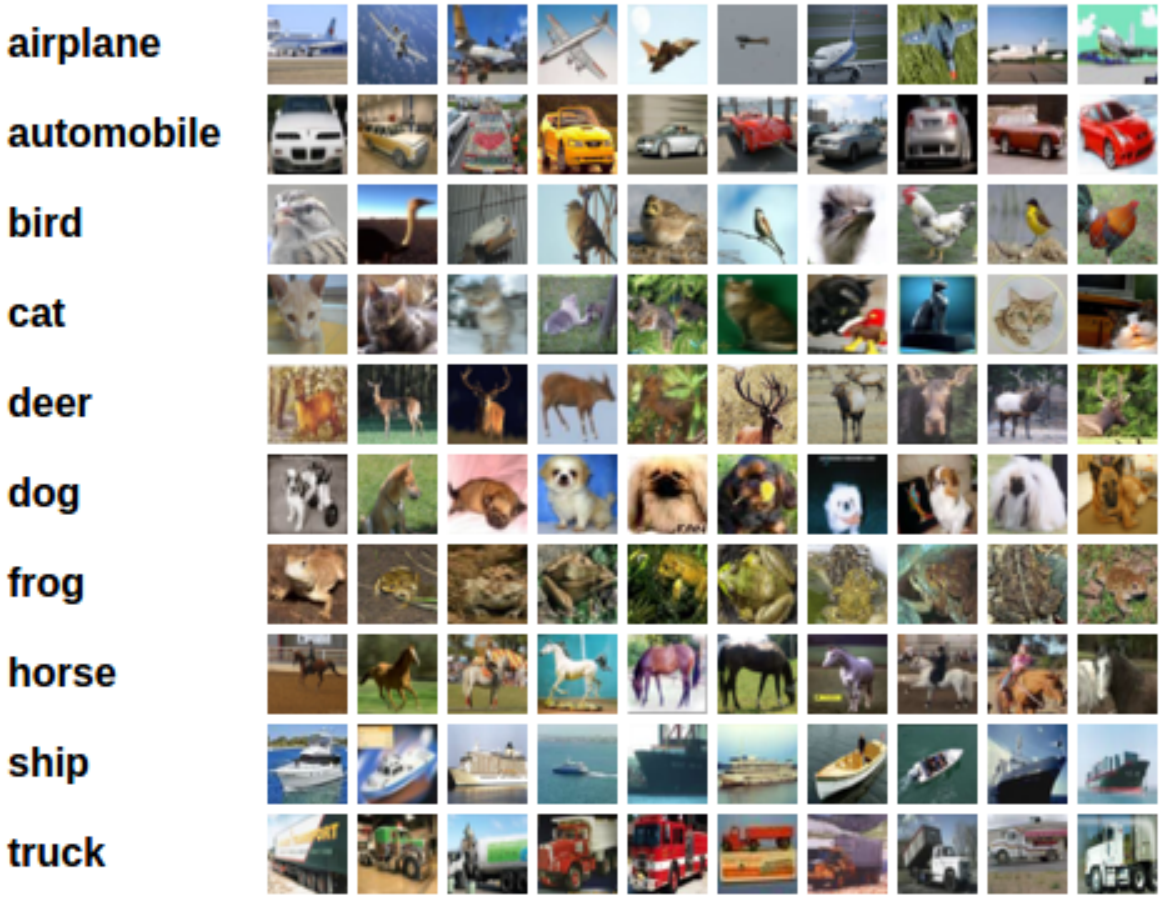

Figure 1: A sample of images from CIFAR-10 [2]

(a) For the class `KNearestNeighbor` defined in the notebook, please finish implementing the following methods:

i. **(1 point)** Please read the header for the method `compute_distance_two_loops` and understand its inputs and outputs. Fill the remainder of the method as indicated in the notebook, to compute the L2 distance between the images in the test set and the images in the training set. The L2 distance is computed as the square root of the sum of the squared differences between the corresponding pixels of the two images.
Hint: You may use `np.linalg.norm` to compute the L2 distance.

ii. **(1 point)** It will be important in subsequent problem sets to write fast *vectorized* code: that is, code that operates on multiple examples at once, using as few `for` loops as possible. As practice, please complete the methods `compute_distance_one_loops` which computes the L2 distance only using a single for loop (and is thus partially vectorized) and `compute_distance_no_loops` which computes the L2 distance without using any loops and is thus fully vectorized.
Hint: $||x - y||^2 = ||x||^2 + ||y||^2 - 2x^T y$

iii. **(1 point)** Complete the implementation of `predict_labels` to find the k nearest neighbors for each test image.
Hint: It might be helpful to use the function `np.argsort`.

(b) **(0 points)** Run the subsequent cells, so that we can check your implementation above.

You will use `KNearestNeighbor` to predict the labels of test images and calculate the accuracy of these predictions. We have implemented the code for $k = 1$ and $k = 3$. For $k = 1$, you should expect to see approximately 33% test accuracy.

(c) **(1 point)** Find the best value for k using *grid search* on the validation set: for each value of k, calculate the accuracy on the validation set, then choose the highest one. Report the highest accuracy and the associated k in the provided cell below in the notebook. Also, please run the code that we've provided which uses the best k to calculate accuracy on the test set, and to see some visualizations of the nearest neighbors.

(optional, 0 points) Run the provided cells below to see the effects of normalization on the accuracy.

Problem 2.2 *Linear classifier with Multinomial Logistic (Softmax) Loss*

In this problem, we will train a linear classifier using the softmax (multinomial logistic) loss (Equation 2) for image classification (Figure 1), using stochastic gradient descent.

(a) **(3 points) Estimating the loss and gradients**. Complete the implementation of the `softmax_loss_naive` function and its gradients using the formulae we have provided, following its specification. Please note that we are calculating the loss on a *minibatch* of N images. The inputs are $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ...(\mathbf{x}_N, y_N)$ where $\mathbf{x}_i$ represents the i-th image in the batch, and y_i is its corresponding label.

We first calculate the scores for each object class, i.e. the unnormalized probability that the image is of a particular class. We'll denote the scores ***for a single image*** as $\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_C$ where C is the total number of classes, and compute them as, $\mathbf{s} = Wx_i$. The softmax loss ***for a single image***, L_i can be defined as,

$$L_i(\mathbf{s}, y) = -\log \frac{e^{s_{y_i}}}{\sum_{j=1}^{C} e^{s_j}} \tag{1}$$

The total loss $\mathcal{L}$ for all images in the *minibatch* can then be calculated by averaging the losses over all of the individual examples:

$$\mathcal{L}(W) = \frac{1}{N} \sum_{i=1}^{N} L_i. \tag{2}$$

Caution: When you exponentiate large numbers in your softmax layer, the result could be quite large, resulting in values of `inf`. To avoid these numerical issues, you can first subtract the maximum score from each scores as shown below:

$$L_i = -\log \frac{e^{s_{y_i} - \max_k(s_k)}}{\sum_{j=1}^{C} e^{s_j - \max_k(s_k)}} \tag{3}$$

Gradients We provide the formulae for the gradients, $\frac{\partial L}{\partial W}$, which will also be returned by `softmax_loss_naive`:

$$\frac{\partial L_i}{\partial W_{y_i}} = \left(\frac{e^{s_{y_i} - \max_k(s_k)}}{\sum_{j=1}^{C} e^{s_j - \max_k(s_k)}} - 1 \right) x_i \tag{4}$$

$$\frac{\partial L_i}{\partial W_j} = \left(\frac{e^{s_j - \max_k(s_k)}}{\sum_{j=1}^{C} e^{s_j - \max_k(s_k)}} \right) x_i, \quad j \neq y_i \tag{5}$$

As described in the notebook, after implementing this, please run the indicated cells for loss check and gradient check and make sure you get the expected values.

(b) **(3 points)** For the `LinearClassifier` class defined in the notebook, please complete the implementation of the following:

i. **Stochastic gradient descent.** Read the header for the method `train` and fill in the portions of the code as indicated, to sample random elements from the training data to form batched inputs and perform parameter update using gradient descent. (Loss and gradient calculation has already been taken care of by us) .

ii. **Running the classifier.** Similarly, write the code to implement `predict` method which returns the predicted classes by the linear classifier.

(c) **(optional, 0 points)** Please run the rest of the code that we have provided, which uses `LinearClassifier` to train on the training split of the dataset and obtain the accuracies on the training and validation sets. Observe the accuracy on the test set, which should be around 38%.

(d) **(optional, 0 points)** Finally, please refer to the visualizations of the learned classifiers. In these visualizations, we treat the classifier weights as though they were an image, and plot them. You may observe some interesting patterns in the way that each classifier distributes its weight.

(e) **(optional, 0 points)** (i) Show that Equation 1 is equivalent to Equation 3. That is, subtracting the largest score does not change the result of softmax. (ii) Explain why this may reduce numerical issues during training..

Problem 2.3 *Neural Networks*

In this problem, you will implement and train several neural networks with PyTorch for image classification: a simple Multi-layer Perceptron (MLP), a convolutional network (CNN), and a CNN with residual connections [1]. A residual connection is a shortcut that lets the network pass information directly from earlier layers to later ones, making it easier to train very deep networks without losing important information.

We will still use the CIFAR-10 dataset for this problem. Below is an outline for your implementation. For more detailed instructions, please refer to the notebook.

(a) **(2 points)** We often train deep neural networks on very large datasets. Because it is impossible to fit the whole dataset into the RAM, loading the data one batch at a time

is a common practice. We also often need to preprocess the data, which includes common preprocessing steps like normalizing the pixel values, resizing the images to have a consistent size, and converting `NumPy` arrays to PyTorch tensors. As the first step of this problem set, please fill in the indicated part for building data loaders with the specified data preprocessing steps (for the specific preprocessing you need to perform, please see the comments in the provided notebook). You may also find the PyTorch tutorial[1] on data loading to be helpful.

(b) **(1 points)** Construct the `MLP` model, consisting of several linear layers and ReLU activations. Make sure the neural network you build has the same architectures as the ones we give in the notebook before you start training them.

(c) **(2 points)** Implement the training and validation loops. For the training loop, you will need to implement the following steps: i) compute the outputs for each minibatch using the neural networks you build, ii) calculate the loss, iii) update the parameters using the SGD optimizer (with momentum). Please use PyTorch's built-in automatic differentiation, rather than implementing backpropagation yourself. The validation loop is almost identical to the training loop except that we do not perform gradient update to the model parameters; we'll simply report the loss and accuracy. Please see the notebook as well for further instructions.

(d) **(1 point)** Train the `MLP` network and visualize training and validation accuracy history for the model. Note: the model is very small and can be trained entirely on CPU. Training will take about 5 minutes on Colab with *CPU*.

(e) **(2 points)** Construct the `CNN` model, with convolutional layers, batch normalization, ReLU activations, global average pooling, and a final linear layer. Train the network and visualize training and validation accuracy history for the model. Make sure the neural network you build has the same architectures as the ones we give in the notebook before you start training them. Note: training will take about 5 minutes on Colab with T4 GPU.

(f) **(1 points)** Construct the `Residual CNN` model. The architecture is very similar to a CNN, with the addition of residual connections. Train the network and visualize training and validation accuracy history for the model. Make sure the neural network you build has the same architectures as the ones we give in the notebook before you start training them. Note: training will take about 5 minutes on Colab with T4 GPU.

(g) **(0 points)** Please run the code we provide to compute top-k accuracy of the three models.

(h) **(optional, 0 points)** Since we use Global Average Pooling (GAP), we can easily compute the Class Activation Map (CAM) [3] to visualize where the network attends for a given category. Complete the relevant blocks in the notebook. The implementation of CAM has already been provided to you.

[1] https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

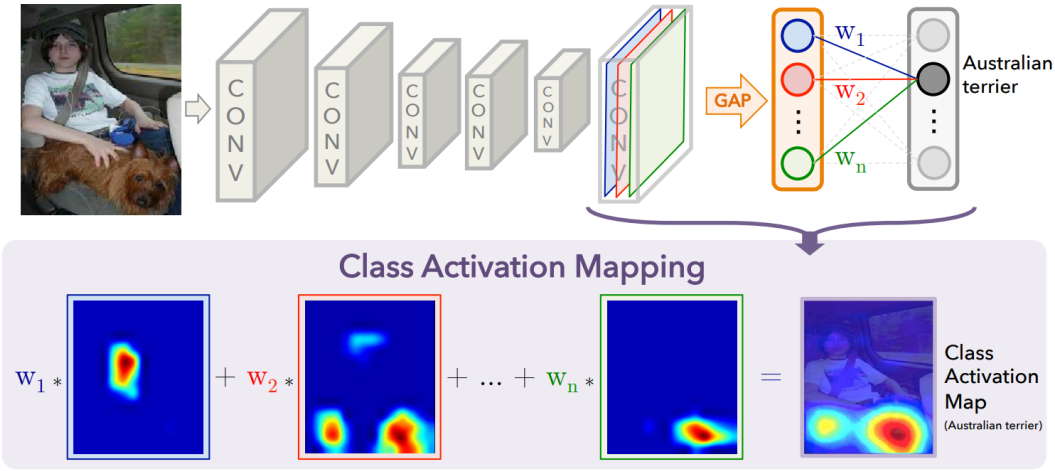

Figure 2: An overview of Class Activation Map (CAM). Figure source: Zhou et al. [3]

1 GPU Tips

For the remainder of the class, we will be using GPUs with Google Colab. We use Colab, since we believe that it is the best possible option (given the fact that high quality GPUs are very expensive). However, there are a number of challenges due to the limits that Colab puts on GPU usage for free accounts. To help you get some information on this and make the most use of the free limits we have compiled a list of best practices to follow.

Colaboratory or simply Colab is part of the Google suite of products. Put simply, Colab is a free Jupyter notebook environment that runs entirely in the cloud. You can read more about it here: `https://colab.research.google.com/`. We use Colab with Python environment and to speed up model training. We will be using a very important feature of Colab which is GPU access.

Colab usually provides T4 GPU for free accounts. Very rarely you'll get a A100 or L4, which are much faster. Colab has time and usage limits for GPU access and the rules for these limits are vague so we have to be careful using them. The general observation is that you start off with a fixed (4 to 8hr, could be lower) bank. Once you exhaust that, you will not longer be able to use GPU for upto 24hrs. Once the counter resets, you get lower hours of usage each time. So how do we maximize our free GPU resources?

(1) **Use CPU for debugging**

The majority of the implementation in most of the assignments can be completed in CPU; you should only switch to GPU once you are confident about your code and ready to train your finals models. The notebook we give you might connect to a GPU runtime by default. You can change between CPU and GPU instances using Runtime → Change Runtime Type; this resets the notebook backend, so you'll need to rerun all cells after switching instance types. You will also need to change `device = torch.device('cuda')` to `device = torch.device('cpu')` Please make sure you don't have any `.cuda()` statements but rather `.to(device)`

Also, please note that **you do not need the GPU for problems 2.1 and 2.2, so make**

sure to run in CPU-only mode for these problems.

(2) **Use other Google accounts**

If you have multiple Google accounts, please note that each one has a separate limit.

Enable cell execution notifications: Colab can notify you of completed executions even if you switch to another tab, window, or application. You can enable it via Tools → Settings → Site → Show desktop notifications (and allow browser notifications once prompted) to check it out. This is helpful when training models so you don't leave GPU connected and idle for long times.

(3) **Debugging tips for GPU**

```
RuntimeError: CUDA device-side assert triggered
```

Do not rely on the line where this error is raised. Even the error traceback says this: RuntimeError: CUDA error: device-side assert triggered CUDA kernel errors might be asynchronously reported at some other API call, so the stacktrace below might be incorrect. For debugging consider passing `CUDA_LAUNCH_BLOCKING=1`

The stack trace might be incorrect. For better debugging, re-run your code as

```
CUDA_LAUNCH_BLOCKING=1 python script.py
```

or add

```
import os
os.environ["CUDA_LAUNCH_BLOCKING"] = 1
```

at the top of your code (temporarily) for debugging. The traceback you get now will point you to the actual line where the error occurs.

(4) **Non-Colab options**

We highly recommend using Colab. However, there are several other options.

If you have a GPU in your local machine and can set it up to use within Jupyter notebooks, you can use it to run the notebooks. In our experience, this approach is also less reliable, so we recommend using Colab when possible. Also, since the problem sets are designed for Colab, running them here will require modifications. You can also connect Colab to a local runtime with GPU. Instructions here: https://research.google.com/colaboratory/local-runtimes.html. Please note that the instructors will not be able to help debug environment setup and code issues on non-Colab systems.

(5) **Colab Pro**

Google offers paid premium upgrades: Colab Pro and Pro Plus which are $10/month and $50/month respectively. These upgrades have access to faster GPUs and for longer. If you

need more access, you can consider these options. All assignments will be thoroughly tested to ensure that it can completed within the limits of free Colab resources.

Acknowledgements. Some of the homework and the starter code was taken from previous CS231n course at Stanford University by Fei-Fei Li, Justin Johnson and Serena Yeung.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[2] A. Krizhevsky, V. Nair, , and G. Hinton. Cifar, 2009. URL https://www.cs.toronto.edu/~kriz/cifar.html.

[3] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929, 2016.