# CS540 Spring 2025 Homework 7

## 1 Assignment Goals

- Implement and train LeNet-5 [3], a simple convolutional neural network (CNN).
- Understand and count the number of trainable parameters in the model.
- Explore different training configurations by varying hyper-parameters like batch size, learning rate and number of training epochs.
- Design and customize your own deep network for scene recognition.

## 2 Summary

Your implementation in this assignment **might take one or two hours to run**. We ***highly*** recommend starting working on this assignment early! In this homework, we will explore building deep neural networks, specifically Convolutional Neural Networks (CNNs), using PyTorch. Helper code is provided in this assignment. If you have compute limitations, you can train and evaluate your model on CSL servers (see instructions in HW6 and at the end of this assignment). Alternatively, if you have a sufficiently powerful CPU, you can also run this on your machine. Go through Submission details in Section 7 carefully.

## 3 Packages Needed for this Project

You are only allowed to use Python3 standard library as well as numpy, torch, torchvision, and tqdm. We recommend using the conda package manager as suggested in the PyTorch tutorial to easily set up the correct environment on the CSL machine:

### 3.1 Install Conda

Documentation reference (Miniconda) here.

#### 3.1.1 On CSL

```
>>> ssh <userid>@best-linux.cs.wisc.edu
>>> mkdir -p ~/miniconda3
>>> wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh -O
  ~/miniconda3/miniconda.sh
>>> bash ~/miniconda3/miniconda.sh -b -u -p  /miniconda3
>>> rm ~/miniconda3/miniconda.sh
>>> source ~/miniconda3/bin/activate
>>> conda init --all
```

You may need to restart your terminal or try `source ~/.bashrc`.

#### 3.1.2 On your machine

Alternatively, if you can run the code on your own machines, follow the instructions depending on your OS:

- MacOS
- Windows
- Linux

### 3.2 Use conda to set up environment for this HW

Installing the Pytorch environment may take a while:

```
>>> conda create -n "cse540-hw7" pytorch torchvision torchaudio anaconda::tqdm cpuonly
  -c pytorch
>>> conda activate cse540-hw7
```

## 4 Dataset

You will implement LeNet and design your own CNN model on CIFAR100 [2], a scene recognition dataset from Alex Krizhevsky and Geoffrey Hinton (Nobel Price winner!) at the University of Toronto. The CIFAR100 dataset has 100 classes containing 600 tiny images each. There are 500 training images and 100 testing images per class. Each image is 32x32 in size. You can always assume this input resolution.

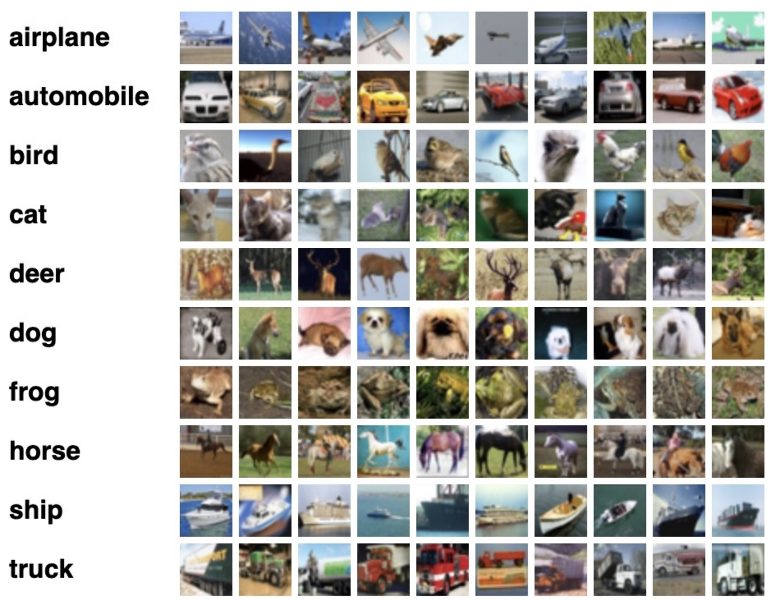


Figure 1: Examples of images in the CIFAR100 dataset.

### 4.1 Helper Code

We provide helper methods in `train_cifar100.py`, `eval_cifar100.py` and `dataloader.py`, and skeleton code in `student_code.py`. See comments in these files for more details.

Before beginning the training procedure, we define the dataloader, model, optimizer, image transforms and criterion. We use `train_model()` and `test_model()` methods for training and evaluating the CNN, which is similar to what we did in HW6. We provide the helper code on canvas (`HW7.zip`). Download and unzip this file.

If you are using CSL, here is how to copy `HW7.zip` contents to the CSL server:

```
>>> unzip HW7.zip
>>> scp -r HW7 <userid>@best-linux.cs.wisc.edu:<path_you_want>
```

Our data loader will try to download the full dataset the first time you run `python train_cifar100.py`, and you should see:

```
>>> Loaded trainset: 1563
>>> Loaded testset: 313
>>> ...
>>> ValueError: optimizer got an empty parameter list
```

The `ValueError` is because you have not yet written code for `LeNet()` in `student_code`. **Think:** What do the numbers 3125 and 313 represent? Hint: how do the dataloaders load the data? If this process works, skip to Section 5.

**Backup setup:** If the CIFAR100 website is down, you will get a download error. Follow these instructions to set up the dataset manually:

1. Download the backup CIFAR100 zip file from this link

2. **If you are using CSL Server:** Copy the downloaded zip file to the CSL server via scp. Unpack the CIFAR100 zip file (`data.zip`) by running the command below.

   ```
   >>> scp data.zip <userid>@best-linux.cs.wisc.edu:<path_you_want>
   >>> unzip data.zip
   ```

   If your CSL account do not have unzip command installed, do the following:

   ```
   >>> sudo apt-get install unzip
   ```

3. **If you are using your own machine:** Unpack the zip file by running the command below.

   ```
   >>> unzip data.zip
   ```

4. Now, you will have a `data/` directory, which consist of `cifar-100-python.tar.gz`. You can then run `train_cifar100.py`, and our code will generate the following: (1) `data/cifar100/images/` directory, (2) `data/cifar100/train.txt` label file, and (3) `data/cifar100/test.txt` label file.

   Once you have this all set up, you can delete `data.zip`

# 5 Program Specification

Implement the following in `student_code.py`:

1. `class LeNet()`: define the network layers in `__init__()` and the forward process in `forward()`.

2. `count_model_params()`: return the number of trainable parameters in the model

## 5.1 Creating LeNet-5 ([30] points)

**Background:** LeNet [3] was one of the first convolutional neural networks (CNNs), and its success was foundational in furthering research into deep learning for computer vision. While we are implementing an existing architecture, it might be helpful to think about *why* early researchers chose certain kernel sizes, padding, and strides which we learned about conceptually in class.

**Implementation:** In LeNet-5, you should use the following layers in this order (see Figure 2):

1. One `conv` layer with the 6 output channels, kernel size = 5, stride = 1, followed by a ReLU [1, 4] activation and a 2D max pool operation (kernel size = 2 and stride = 2).
2. One `conv` layer with 16 output channels, kernel size = 5, stride = 1, followed by a ReLU activation and a 2D max pool operation (kernel size = 2 and stride = 2).
3. A `flatten` layer to convert the 3D tensor to a 1D tensor.
4. A `linear` layer with output dimension = 256, followed by a ReLU activation.
5. A `linear` layer with output dimension = 128, followed by a ReLU activation.
6. A `linear` layer with output dimension = number of classes (in our case, 100).

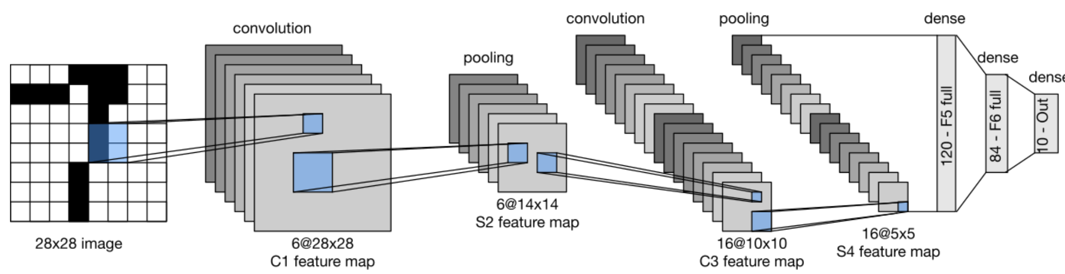


Figure 2: LeNet-5 Architecture.

Implement the model with the `LeNet()` class in `student_code.py`. You are expected to create the model following this PyTorch tutorial, which is different from using `nn.Sequential()` as we did in the last HW. In addition, given a batch of inputs with shape [N, C, W, H], where N is the batch size, C is the input channel and W, H are the width and height of the image (both 32 in our case), **you are expected to return** both the output of the model (`torch.Tensor`) along with the shape of the intermediate outputs for the above 6 stages. The shape should be a Python dictionary with the keys = [1, 2, 3, 4, 5, 6] (integers) denoting each stage, where the corresponding value is a list that denotes the shape of the intermediate outputs.

**Hint:** The expected model has the following form:

```
class LeNet(nn.Module):
    def __init__(self, input_shape=(32, 32), num_classes=100): super(LeNet
    , self).__init__()
        # certain definitions
    def forward(self, x):
        shape_dict = {}
        # certain operations
        return out, shape_dict
```

`shape_dict` should have the following form: `{1: [a, b, c, d], 2:[e, f, g, h], ... , 6: [x, y]}` The linear layer and conv layers have bias terms. You need to use `torch.nn.conv2d` to create a convolutional layer. The method parameters allow us to specify the details of the conv layer eg. the input/output dimensions, padding, stride, and kernel size. More information can be found in the documentation.

## 5.2 Count the number of trainable parameters of LeNet-5 ([20] points)

**Background:** As discussed in the lecture, fully connected models (like what we created in HW6) are **dense**, with many trainable parameters. After finishing this section, think about the number of parameters (also sometimes called model size) in the CNN model compared to the number of parameters in a fully connected model of similar depth (similar number of layers). Especially, how does the difference in size impact efficiency and accuracy?

**Implementation:** In this part, you are expected to return the number of trainable parameters of the LeNet model you created in Section 5.1. You have to fill in `count_model_params()` in `student_code.py`. The function output should be in the unit of Million (1e6) ie. how many millions of trainable parameters are in your implementation of LeNet. Please do not use any external libraries (see Section 3) which directly calculate the number of parameters (other libraries, such as NumPy can be used as helpers)

**Hint:** You can use the `model.named_parameters()` to get the name and corresponding parameters of a torch model. **Please do not round your result**.

## 5.3 Training LeNet-5 under different configurations ([50] points)

**Background:** A large part of creating neural networks is designing the architecture (part 1). However, there are other ways of tuning the neural net to change its performance. In this section, we can see how batch size, learning rate, and number of epochs impact how well the model learns. As you get your results, it might be helpful to think about how and why the changes have impacted the training.

**Implementation:** Based on the LeNet-5 model created in Section 5.1, in this section you are expected to train the LeNet-5 model under different configurations. You will use similar implementations of `train_model` and `test_model` as you did for HW6 (which we provide in `student_code.py`). When you run the training script `train_cifar100.py`, the script will save two files in the `outputs/` folder.

- checkpoint.pth.tar is the model checkpoint at the latest epoch.
- model_best.pth.tar is the model weights that has highest accuracy on the validation set.

Our code supports resuming from a previous checkpoint, so you can pause training and resume later. This can be achieved by running `python train_cifar100.py --resume ./outputs/checkpoint.pth.tar`. This is also very helpful if your training is interrupted for any reason.

**Evaluation:** After training, you can evaluate your model on the val set with the `eval_cifar100.py` script we provide. This script will grab a pre-trained model and evaluate it on the val set of 10K images. For example, you can run `python eval_cifar100.py --load ./outputs/model_best.pth.tar`. The output shows the validation accuracy and also the model evaluation time in seconds (see an example below).

```
=> Loading from cached file ./data/cifar100/cached\_val.pkl
=> loading checkpoint './outputs/model\_best.pth.tar'
=> loaded checkpoint './outputs/model\_best.pth.tar' (epoch x)
Evaluting the model ...
[Test set] Epoch: xx, Accuracy: xx.xx %
Evaluation took 2.26 sec
```

You can run this script a few times to see the average runtime of your model. Please train the model under the following configurations:

1. The default configuration provided in the code, which means you do not have to make modifications.
2. Set the batch size to 8, the remaining hyper-params are same as the default configuration.
3. Set the batch size to 16, the remaining hyper-params are same as the default configuration.

4. Set the learning rate to 0.05, the remaining hyper-params are same as the default configuration.
5. Set the learning rate to 0.01, the remaining hyper-params are same as the default configuration.
6. Set the epochs to 20, the remaining hyper-params are same as the default configuration.
7. Set the epochs to 5, the remaining hyper-params are same as the default configuration.

After training, you are expected to get the validation accuracy using the best model (`model_best.pth.tar`), then save the output into a results.txt file, where the accuracy of each configuration above is placed in each newline, in order. Your .txt file will end up looking like this:

```
11.11
22.22
33.33
...
```

These exact accuracy will probably not align well with your results. They are just for illustration purposes. Follow the submission details in Section 7.

# 6 Help with Training

## 6.1 Profiling Your Model (Optional)

You might find that the training or evaluation of your model is a bit slower than expected. Fortunately, PyTorch has its own profiling tool. Here is a quick tutorial of using PyTorch profiler. You can easily inject the profiler into train_cifar100.py to inspect the runtime and memory consumption of different parts of your model. A general principle is that a deep (many layers) and wide (many feature channels) network will train much slower. It is your design choice to balance between efficiency and accuracy.

## 6.2 Training on CSL

You are required to train and evaluate your models on the CSL machines. You should find a way to allow your remote session to remain active if you are disconnected from CSL. In this case, we recommend using tmux, a terminal multiplexer for Unix-like systems. tmux is already installed on CSL. To use tmux, simply type `tmux` in the terminal. Now you can run your code in a tmux session. And the session will remain active even if you are disconnected.

- If you want to detach a tmux session without closing it, press "ctrl + b" then "d" (detach) within a tmux session. This will exit to the terminal while keeping the session active. Later you can re-attach the session.
- If you want to enter an active tmux session, type "tmux a" to attach to the last session in the terminal (outside of tmux).
- If you want to close a tmux session, press "ctrl + b" then "x" (exit) within a tmux session. You won't be able to enter this session again. Please make sure that you close your tmux sessions after this assignment.
- Here is a brief tutorial about this powerful tool, tmux.

# 7 Submission Notes

Please submit two files named `student_code.py` and `results.txt` (From Section 5.3) to Gradescope. Do *not* submit a Jupyter notebook `.ipynb` file. Be sure to remove all debugging output before submission. Failure to remove debugging output may be penalized.

- No code should be put outside the function definitions (except for import statements; helper functions are allowed).
- The validation accuracy for different configurations must be in .txt format and named `results.txt`

This assignment is due on April 8th at 11:59 PM. **We highly recommend starting early. We highly suggest submitting a version well before the deadline** (at least one hour before) and check the content/format of the submission to make sure it's the right version. You can then update your submission until the deadline, if needed.

**Good luck and happy (deep) learning!**

# References


[1] Alston S Householder. A theory of steady-state activity in nerve-fiber networks: I. definitions and preliminary lemmas. *The bulletin of mathematical biophysics*, 3:63–69, 1941.

[2] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.