Assignment 11

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- Convolutional neural networks for classification (VGG architecture)
- All the things that go into training a deep learning model:
  - Data augmentation
  - Learning rate scheduling
  - Early stopping
  - Hyperparameter tuning

Part 1 of 3: Implement the VGG architecture for CIFAR-10

You will need to both implement this model (in part 1) and spend time optimizing its training (in part 2) while sharing server resources with your classmates. As always, start early and don't be afraid to chat with each other, come to office hours, and use assistive tools like Copilot.

Instructions:

In the `models.py` file, implement the VGG class by filling in the `__init__` and `forward` methods. There are four sub-types of the VGG architecture, each with a different number of layers, named `VGG11`, `VGG13`, `VGG16`, and `VGG19`. As we discussed in class, this is done here using a common design principle in PyTorch: defining a generic VGG base class, then creating sub-classes that specialize the architecture.

The "plan" part of the architecture specifies the order and width of the convolutional part of the network. Everywhere that the "plan" is an integer, that means that the next layer(s) should be:

```
nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
nn.ReLU(),
```

where `in_channels` is however many channels was output by the previous layer, and `out_channels` is the integer specified in the plan. Using a `kernel_size` of 3 and `padding` of 1 will preserve the height and width of the feature map through the convolutional layers. Everywhere that the "plan" is the string "M", that means that the next layer should be:

```
nn.MaxPool2d(kernel_size=2, stride=2)
```

which will reduce the height and width of the feature map by half.

After the convolutional plan is done, the output should have size `(512, 2, 2)`. Flattening this gives a `2048` dimensional vector. This vector is then fed into a multi-layer perceptron (MLP) (a few `nn.Linear` layers followed by `ReLU` activations). Remember to **not** put a `ReLU` activation on the output/logits! The `num_classes`, `readout_width`, and `readout_depth` parameters specify the architecture of the MLP readout.

Note 1: we are doing this on CIFAR-10 rather than ImageNet, so we will be implementing a slightly smaller version of the original published VGG architecture. Pay attention to the number of layers, the number of filters in each layer, and the size of the fully connected layers at the end.

Review how PyTorch custom modules work here. If you're having trouble implementing the VGG architecture, a good way to sanity-check what's happening in your model is to add print statements or break points in the `forward` method of the model. You can then run the model on some random image data and see what the output shape is at each layer. This will help you debug the model architecture.

Part 2 of 3: Implement a few modern training tricks

Despite what you may have heard, training deep learning models is not just about using a big fancy model, and throwing data at it. You should try training one of each of the 4 models, monitoring them in tensorboard, and seeing how they do on CIFAR10. You'll hopefully notice that VGG11 does ok, but the deeper models are actually harder to train without additional tricks.

This part of the assignment will guide you through implementing a few common and useful tricks that will help your models train more effectively and repeatably.

Instructions:

1. Switch from the SGD optimizer to the `Adam` optimizer in `train.py`. The `Adam` optimizer has emerged as a popular choice for training deep learning models because it is more robust to hyperparameter choices and often converges faster than SGD. This is not *always* the case, but it will work well for VGG. You may use the default parameters for `Adam`.
2. Modify your model architecture to include a `BatchNorm2d` layer after each convolutional layer but before the `ReLU` layer. This will help stabilize the training process and reduces the instability of training deeper models. After this change, you should see the VGG19 model train much more effectively than before. **This paper** can give more details about what's happening if you're curious.
3. Add data augmentation by modifying the `get_transform` function in `datasets.py`. You should only augment the training data, not the test data! You should think about which augmentation methods make sense for the CIFAR classes and try out a few form the `torchvision.transforms` module.
4. Implement a learning rate scheduler in `train.py` using the `torch.optim.lr_scheduler.StepLR` class. Set its `step_size` to the `lr_decay_every` parameter and its `gamma` to `lr_decay_amt`. This will update the learning rate every `lr_decay_every` epochs by multiplying it by `lr_decay_amt`. Make sure you save and load the scheduler state_dict along with the model state_dict so that you can resume training from a checkpoint. Also make sure that you call the `scheduler.step()` method after each epoch to update the learning rate. Saving and loading the scheduler state is what will allow the training to resume properly if it crashes.
5. Implement early stopping in `train.py`. This is a simple but effective technique to prevent overfitting. A `EarlyStopper` class has been provided for you in `early_stopping.py`. The idea is to keep track of the validation loss and stop training when the validation loss stops improving. It includes "debouncing" or "patience" so that it only stops if the validation loss has

consistently failed to improved for some number of epochs in a row. You should instantiate the `EarlyStopper` with its patience set to `stopper_patience` and `update()` it after each epoch, breaking out of the training loop if it signals that you should do so.

Just like the learning rate scheduler, make sure you save and load the early stopper state_dict to "checkpoint_latest.pt" so that you can resume training from a checkpoint.

In addition to early stopping, we'll keep track of the best model so far and save it to disk in a "checkpoint_best.pt" file. So, each training run will result in two checkpoints on disk: "checkpoint_latest.pt" (for resuming training) and "checkpoint_best.pt" (a record of model parameters at the point during training with the minimum validataion loss). You'll see in `plot.py` that we're loading only the "checkpoint_best.pt" files for plotting.

Part 3 of 3: Run and plot

As you experiment with the various strategies above, keep track of what hyperparameters worked well (what learning rate, batch size, stopper patience, etc.) and what didn't. When you're ready to do your final training runs, move or delete all of your old logs and checkpoints so that you can start fresh (you may have to delete files simply due to your server space quotas). Then, train 12 total models: 3 versions of each of the 4 VGG architectures (VGG11, VGG13, VGG16, VGG19). You can do this with a bash script like the `launch_training.sh` script we've provided.

Consider running it in a `screen` terminal session on the server, which will allow you to disconnect from the server and leave it running. (Look up tutorials on how to use `screen` or come to office hours if you're not sure how to do this.)

Finally, run `plot.py` to show the performance vs number of parameters for your training runs.

Include an image of the plot generated by `plot.py` in the `images/` directory.

What we're looking for: a general upward trend in performance as the number of parameters increases. If your deeper models are not performing better than your shallower models, you may need to adjust some hyperparameters or training strategies.

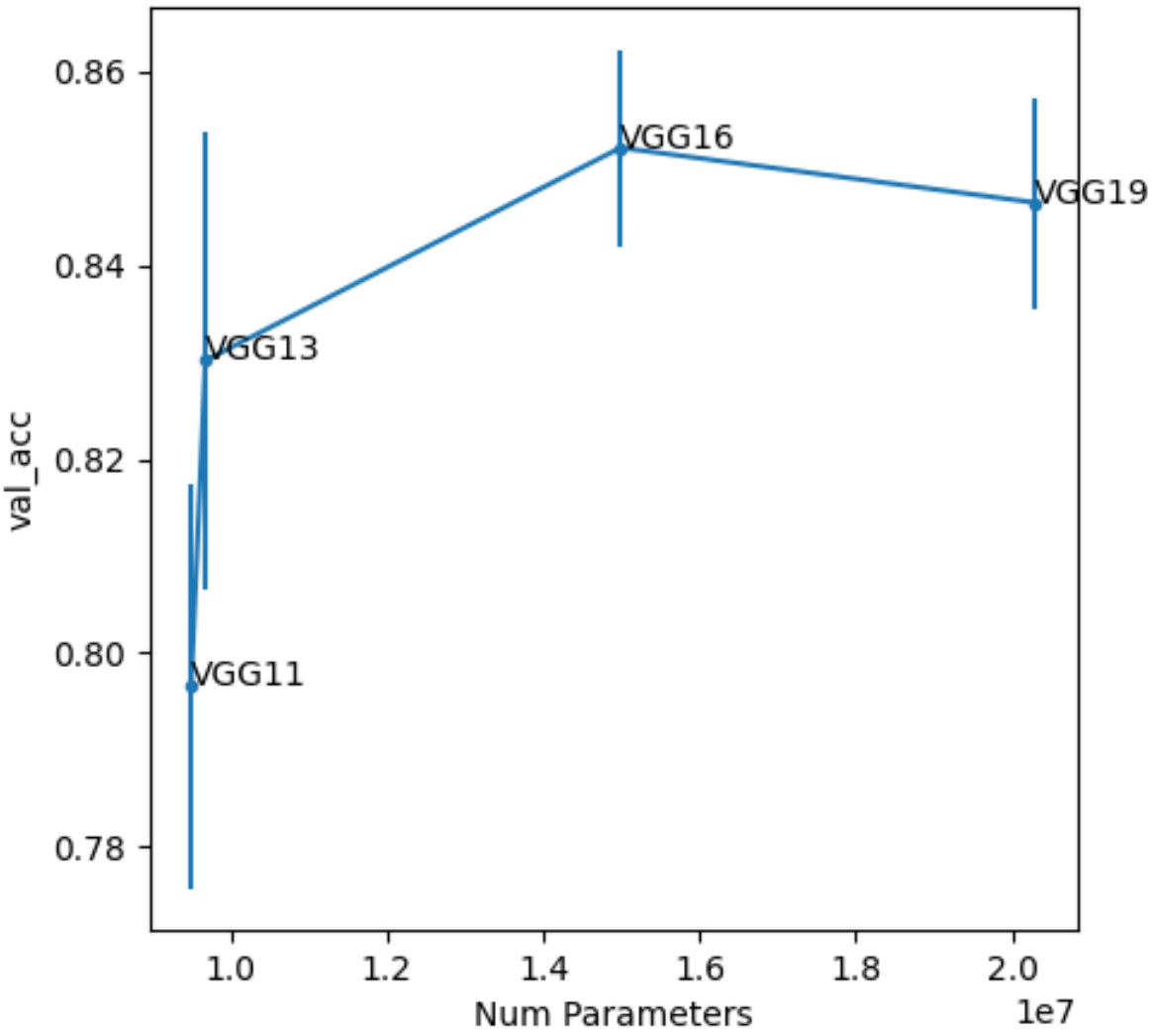

Figure 1: example output of `plot.py`

Upload your trained model checkpoints for us to validate them:

The checkpoint files make the zip file bigger than Gradescope can handle, so they cannot be included them in the zip. Instead, use the provided `submit_checkpoints.sh` script to copy your checkpoint files from wherever you created them on the department servers into a location where we can access them. These will be validated separately after the deadline and won't appear as part of your autograder score.

The submission script automates the process of submitting model checkpoint files for grading. It identifies all files named `checkpoint_*.pt` in the specified directory, validates their existence, and copies them to the designated submission directory on the course server. Usage Instructions:

1. Run on a Course Server: Ensure you are logged into one of the approved servers (`granger`, `weasley`, or `lovegood`).
2. Navigate to Your Directory: Change to the directory containing your checkpoints (e.g., `~/csci631/assignments/assignment-11/src/` or maybe in `/tmp/pycharm_project_XYZ/src`
3. Execute the Script:

   bash ./submit_checkpoints.sh

   Replace `<path_to_checkpoint_dir>` with the relative or absolute path to the directory containing your checkpoints.
4. Confirm Submission: The script will list all checkpoints it finds and prompt you for confirmation.
5. Check Submission: Once completed, the script will display the submitted files.

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.