IN4310/3310 - Mandatory 2

April, 2025

Submission due 29th of April 23:59 Oslo time.

1 The mandatory exercise 2

Figure 1: A visualisation generated using the method generate_captioned_images_top_bottom() in generate_captions.py. This image was generated using a model that also trained the encoder (ResNet18) and is a cherry-picked example. Do not expect to get such visualisations from your model.

Learning goals:

- Work with image captioning and outputs beyond simple fixed classes
- Complete a Recurrent Neural Net (RNN) which outputs a sequence of variable length

- Complete a 2-layer RNN
- Complete LSTM cell equation
- Implement a simple attention model on top

File hierarchy:
This mandatory comes with several fully functional code scripts to streamline the work required by the student. These are the only python files you need to modify:

- config.py
- train.py
- model.py

Is it still recommend you familiarize yourself with the workings of the remaining scripts.

1.1 Task 1

NOTE: **DO NOT** change the variable names, argument names or the function definitions.

- In `config.py`, set

  ```
  self.cell_type = 'RNN'
  ```

  inside `class Config(nn.Module)`

- Complete in

  ```
  class ImageCaptionModel(nn.Module):
  ```

  the

  ```
  self.feature_projection
  ```

  as a sequence of

  - dropout with set to zero probability of 0.25
  - a linear layer with `cnn_feature_dim` many inputs and `hidden_size` many outputs.
  - a leaky ReLU

- Complete in

  ```
  class ImageCaptionModel(nn.Module):
  ```

  the remaining parts except the code block inside `if self.use_attention:` in the `forward` method.

- Complete in

  ```
  class CaptionRNN(nn.Module):
  ```

the remaining parts except the code block inside `if self.use_attention:` in the `forward` method.

- Initialize the hidden states with zeros! The content gets infused into the RNN as input only
- Take note of the hidden state shape $(num\_layers, batch\_size, 2 \times self.hidden\_state\_size)$

  Reason: The hidden state tensor in LSTM is the concatenation of the memory cell and the actual hidden state. The first half represents the hidden state, and the second half represents the memory cell state. We do not need cell memory in vanilla RNN but we keep the interface the same for the RNNCell so as to avoid putting in 'if' statements. Check the code in the RNNCell for a better understanding.
- You may set `self.num_layers` to 1 in config.py if you'd like to keep it simple. But if you do not, then take care of the next 2 points.
- Take note that inputs to the RNN cells are different in 1^{st} and 2^{nd} layers!
- Take note of what to put in as dimensions into

  `input_sizes`

  for each layer.
- Train your model.
- **Note:** You'd need to load both PyTorch and matplotlib for running train.py because it plots the loss and evaluation metrics after training.

  ```
  module load PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised

  module load matplotlib/3.4.3-foss-2021b
  ```

On features:

- I use a pre-trained ResNet18's features. It is common to use a ResNet pre-trained on the ImageNet dataset and this is what I use here. You can also train the ResNet18 model (or a bigger model like ResNet50 or ResNet101), but it will consume more time and resources.
- Extracted features from ResNet18 are the stage 4's features right before the Global Average Pooling layer, which gives per image a feature of shape $(512, 7, 7)$, which results in a dataset (`resnet50_features_train_file` in Config) of size around 12 GB.
- The point above is why I am asking you to Global Average Pool features yourself in the code. But we need non-pooled features later for the attention mechanism.

1.2 Task 2

NOTE: **DO NOT** change the variable names, argument names or the function definitions.

- Goal of this task is to replace vanilla RNN with an LSTM.
- We won't be using vanilla (simplified) RNN anymore. So `self.cell_type` be 'LSTM' for this task and the rest of the tasks.
  Set

  ```
  self.cell_type = 'LSTM'
  ```

  in the

  ```
  class Config
  ```

  in config.py
- It is advisable to create a separate copy of model.py with a new filename, and a copy of train.py with a new filename which imports the

  ```
  class ImageCaptionModel
  ```

  from the new copy of model*
- implement an LSTM.

  NOTE: Use the equations from the slides.
- Complete in

  ```
  class LSTMCell(nn.Module):
  ```

  the remaining parts.
- Set

  ```
  self.num_layers = 2
  ```

  in the

  ```
  class Config
  ```

  in config.py if you did not do it for Task 1.
- Take note that inputs to the LSTM cells are different in 1^{st} and 2^{nd} layers!
- Take note of what to put in as dimensions into

  ```
  input_sizes
  ```

  for each layer.

- Take note of the hidden state shape (num_layers, batch_size, 2 * self.hidden_state_size).

  Reason: The hidden state tensor in LSTM is the concatenation of the memory cell and the actual hidden state. The first half represents the hidden state, and the second half represents the memory cell state. You must return the output in the same format: 1^{st} half: new hidden state, 2^{nd} half: new cell state.

- Write the code that can also be used for more than just 2 layers.

- You will run into trouble if you code like this:

  ```
  hiddenstate[a,:,0:b] = rnncell(inputs,hiddenstate)
  ```

  due to backward in place modification errors.

- Train your model.

1.3 Task 3

NOTE: **DO NOT** change the variable names, argument names or the function definitions.

- **Goal:** implement a simple attention model. Refer to the Vision Transformers lecture to understand the attention mechanism. The lecture uses translation as an example, but you can think of each input word in the example as a patch in an image, so instead of attending to different words in the input text, here you're attending to different parts of the image.

- The idea is that instead of Global Average Pooling the features (like we have been doing so far), we will do a weighted average of the features. We will still convert features from shape (batch_size, 512, 7, 7) to (batch_size, 512, 1, 1) but instead of a simple averaging of those 7x7 features, we will do a weighted average using the attention mechanism.

- Use the code with

  ```
  self.cell_type = 'LSTM'
  ```

  and

  ```
  self.num_layers = 2
  ```

  in the

  ```
  class Config
  ```

  in config.py if you did not do it for Task 2.

- Complete in

  ```
  class Attention(nn.Module):
  ```

  the remaining parts.

  The Attention mechanism takes as input:

1. The ResNet18 features (batch, 512, 7, 7) but permuted to (batch, 7, 7, 512) and then flattened to (batch, 49, 512). You perform this permutation and partial flattening inside the ImageCaptionModel's forward function. Hint: You can use `cnn_features.permute()` function for the permutation followed by `cnn_features.view()` for partial flattening.
2. The hidden state of the first layer of LSTM.

- The Attention class then projects both the inputs to the same dimension. Add these two projected vectors followed by a ReLU. It then applies a linear layer on this added vector to get the attention scores. Attention scores are passed through a softmax to get the weights, and finally, the weighted average is performed.
- Complete the remaining parts in the `class ImageCaptionModel`'s forward method and `class CaptionRNN`'s forward method in the `if self.use_attention:` blocks.

1.4 Debugging ugly errors appearing during the backward pass

If you are not sure what shape you have somewhere, then just insert:

```
print(sometensor.shape)

exit()
```

If you run into errors saying that backward detected an inplace modification of a tensor,

```
Warning: Error detected in *
RuntimeError: one of the variables needed for gradient computation
has been modified by an inplace operation.
```

Then the solution is usually to clone but not detach an offending tensor (because detach prevents gradient flow backwards!). https://discuss.pytorch.org/t/backward-error-after-in-place-modification-only-if-using-tanh/78291

If you fail to debug it, but you have really tried it on 2 different days, then reach out to me and we can check your code.

The most common error of this type would occur where you use the hidden state to compute the attention layer in Task4. Use then a .clone():

```
w = attlayer(sometensor.clone())
```

1.5 Running unit tests

We're providing you with some unit tests for LSTM equations. The idea is to help you ensure that your implementation is likely to be correct. We might have missed some cases, so if unit tests pass, it does not guarantee that your equations are correct. However, if unit tests fail, your implementation is definitely incorrect.

We are using Python's pytest library (needs to be installed separately in your Python environment). We recommend you run unit tests locally on your PC by installing pytest as `pip install pytest`. The unit tests won't invoke any model training so they will run pretty fast (less than a minute). The tests are located in the `tests` directory. You can check pytest's documentation (https://docs.pytest.org/en/7.1.x/how-to/usage.html) for different ways of running unit tests via command line.

Steps for running unit tests:

1. Install pytest locally via `pip install pytest`.
2. Open the test file (test_lstm.py) and modify the import statement to import your implementation of the LSTMCell.
3. In the shell/terminal, navigate to the root directory of the project.
4. Run `PYTHONPATH=<PATH TO THE PROJECT> pytest tests/test_lstm.py` to invoke pytest for LSTM. Please note that you might need to replace the slash here with the backward slash depending on your OS, and the PYTHONPATH you set would depend on how you're importing the LSTMCell inside the test file test_lstm.py.

1.6 Deliverables

Read carefully!

- Deliver for each task, one separate training start file. If you have to create separate versions of `train.py` for different tasks, then you can do that too.
- For each task, report loss graphs and the metric validation scores. This has already been implemented for you. You need to load the matplotlib module (along with the PyTorch module) for this to work. Run `module load matplotlib/3.4.3-foss-2021b` for loading matplotlib.
- For each task, report the best BLEU@4 and CIDEr score that you achieved (they could be in different epochs). The code in train.py by default only reports the best CIDEr score. You can modify it to also report the best BLEU@4 score.
- For any 1 task of your choice, deliver the pretrained model.
- Deliver a report as pdf-file with your full name – to identify the submitter
- Write in your report: the training parameters
- Write in your report: 5 example images with predicted captions which you obtained using your code – for one of the models. You have 2 options here. You can either use the generate_captioned_images() method in the generate_captions.py file or the generate_captioned_images_top_bottom method if you are using the attention based model. The second method generates the images visualising the attention maps on the images (the figure 1 in the beginning of this PDF).

- Put everything : codes, saved models, the pdf and everything else you want to add **into one single zip file**. Your zip file's name should have your **username** in it.

Code guidelines:

- **DO NOT** change the variable names, argument names or the function definitions.
- One should only be required to change the path to the data folder in your code. All other paths should be relative to your main folder of your .py files, no absolute paths except for the dataset.
- path manipulations: os.path.join, os.path.basename, os.path.dirname, os.path.is*, , os.makedirs(...)
- Reproducibility: set all involved seeds to fixed values (python, numpy, torch). Check PyTorch's page on how to make PyTorch code reproducible: https://pytorch.org/docs/stable/notes/randomness.html
- A requirements.txt and write it into the pdf for any additional packages beyond numpy, scipy, matplotlib, pytorch and their dependencies
- Your deliverable should work with the following steps:
  - Unpack the zip files
  - Set **one single path** for the root of the dataset. This must be documented. Nothing else should need to set it up.
- Code should run without typing any parameters on the command line!!

```
CUDA_VISIBLE_DEVICES=x TMP=./tmp python blafile.py
```

- Python scripts.

2 GPU Resources and Python interpreters

You can run your code on your own computer (if you have a GPU) or on the GPU servers. You can use UiO's computing cluster: any of the ml-nodes.

We cannot log in directly to the GPU servers and must first log in to the login nodes. To log in to the login nodes, use SSH:

```
ssh <user-name>@login.uio.no
```

To login to the GPU server from the login nodes, use ssh again:

```
ssh <user-name>@ml9.hpc.uio.no
```

Each node has eight GPUs, each with 11 GB of GPU RAM. The critical resource here is the GPU RAM. If your code uses more than is available, it will end with a memory allocation error. With a ResNet-18 model, your code will consume less than 2 GB with a batch size of 16. As mentioned earlier, use the nvidia-smi command to check which GPUs are in use, and how much memory is available on those GPUs. If there are 2 GBs available, you can use it. After logging in to the GPU server, you will need to load Python and the libraries needed for the project before you can run any code.

`module load` is the command for loading any modules. The module we are using is `PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised` and `matplotlib/3.4.3-foss-2021b`. To load the modules, do:

```
module load PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised
module load matplotlib/3.4.3-foss-2021b
```

Do `module list` to see what modules are loaded. And to remove any loaded modules, do `module purge`. You can check if Python is loaded by doing `python --version`, or `which python`. To start running a script, use the following command:

```
CUDA_VISIBLE_DEVICES=x python your_script.py
```

`x` is the GPU that you want to use. This command will, however, end when you log out of SSH. To prevent that from happening, you can do the following:

```
CUDA_VISIBLE_DEVICES=x nohup python yourscript.py > out1.log 2> error1.log
&
```

Let's go through this command: `nohup` starts the command without hangup, `> out.log` redirects the output of the script to the file `out.log`, and `2> error.log` redirects the error messages in the same way, and the & symbol places the job running in the background.

2.1 Using screen command to train your models

`screen` is a better utility than nohup:

- Start a screen session with `screen -S <SCREEN_NAME>`
- Run `module purge` and load the modules in the screen.
- Press `ctrl + a + d` to detach from the screen.
- Do `screen -r <SCREEN_NAME>` to reattach to the screen.

- NOTE: If you got logged out or lost connection or something, then run `screen -d <SCREEN_NAME>` first to detach the screen (because losing connection does not automatically detach the screen) and then run the command above to reattach.
- `screen -ls` lists all the screen sessions you have running.
- `Ctrl + a + c` opens a new screen session while inside a screen.
- `Ctrl + a + n` switches to the next screen session (if you have multiple of them).
- `Ctrl + a + p` switches to the previous screen
- `Ctrl + d` exists the screen, destroying it completely.

2.2 How to clean up your processes to avoid clogging up resources on the server?

- Do not start a script when there are already 4 jobs running on it or when it is foreseeable that your 2.5 Gbyte won't fit into this GPU RAM.
- Ensure that you only have 1 main Python process running at a time. Run `pstree -p -U $USER` to check your process tree. When you're training a model, this should show you a tree with 1 main Python process and some other Python processes spun by it as branches of the tree. If you see some old Python process you ran, then kill the stale process with `kill -9 <PID>` where $\langle PID \rangle$ is the process ID shown in the parentheses. Refer to https://man7.org/linux/man-pages/man1/pstree.1.html for more details on pstree.

To kill a process, first show the processes and IDs that you are running. Do this with the following command:

```
ps -u <user-name>
```

And doing:

```
ps -u <user-name> | grep -i python
```

will only show the Python processes you are running. Both of these commands will show the process id. To kill a process, do:

```
kill -9 <process-id>
```

3 Running and debugging with a remote interpreter

Debugging your code with the method above can be time-consuming as you may need to change the code and deploy/upload it to the server manually with scp/rsync each time. You can use a remote interpreter, i.e., debug your code in an IDE (integrated development environment) on your computer, but the code is actually running on the GPU server. We recommend PyCharm. You can use others, but the group teachers may not be able to give you support for the debugging. To do this, we will create a bash script that will load Python

and the modules needed (from the lmod system). We will then point to this bash script (as our Python interpreter) in PyCharm or another IDE of your choice. Your code can then be deployed and debugged without having to use the method above.

To set this up follow these steps:

3.1 Step-by-step to setup the remote interpreter

- Open one terminal and run the following command:

  `ssh -J username@login.uio.no username@ml9.hpc.uio.no`

  This will take you directly to the ML9 node.

- While in ML9, create a new bash file, e.g. using `vim`, and paste this content inside:

  ```
  #!/bin/bash
  source ~/.bash_profile
  module load PyTorch-bundle/1.10.0-MKL-bundle-pre-optimised
  # Run the Python interpreter with the passed arguments
  exec python "$@"
  ```

  Check this link on how to use vim to create and save files.

- Make that script executable by you, by running `chmod 700 your_bash_script.sh`

- Open another terminal and run the following command:

  `ssh -L 6000:ml9.hpc.uio.no:22 <user-name>@login.uio.no`

  This will allow you to tunnel ssh from your local machine to ML9 through the login node, so that you can use the remote interpreter.

- On the bottom left corner of PyCharm, click on the interpreter you have, e.g. "Python 3.x", "Add New Interpreter", "On SSH..."

- For host, type "localhost", for the port, use 6000 (the same number we used in the ssh command above), and enter your username, then click Next.

- PyCharm will perform some checks, when it finishes, click "Next"

- For the Environment, choose "Existing", then for the interpreter, click on the three horizontal dots on the right, and browse through ML9 and choose the bash script you created in step 1. The path will be something like:

  `/itf-fi-ml/home/username/your_bash_script.sh`

- For the sync folders, click on the browse icon on the right, go to your username directory on ML9, create a folder there, e.g. `mand2`, and then choose that folder. Then, Local Path should be your root directory where your local project is, and Remote Path should be the folder you created. This will allow PyCharm to sync files between these two directories. Then, click Create.

With these steps, you should be able to upload any Python file you have on your local machine, by right clicking anywhere inside the file, Deployment, "upload to ...". Afterwards, you should be able to run files remotely from PyCharm for debugging purposes.

3.2 I use my own GPUs, where to get the data?

in the folder /itf-fi-ml/shared/courses/IN3310/mandatory2_data

- ...

Writing a mandatory exercise ... you can guess how long it took for that pdf, now it is your time :).