HW1

January 24, 2025

Please complete the `NotImplemented` parts of the code cells and write your answers in the markdown cells designated for your response to any questions asked. The tag `# AUTOGRADED` (all caps, with a space after `#`) should be at the beginning of each autograded code cell, so make sure that you do not change that. You are also not allowed to import any new package other than the ones already imported. Doing so will prevent the autograder from grading your code.

For the code submission, run the last cell in the notebook to create the submission zip file. If you are working in Colab, make sure to download and then upload a copy of the completed notebook itself to its working directory to be included in the zip file. Finally, submit the zip file to Gradescope.

After you finish the assignment and fill in your code and response where needed (all cells should have been run), save the notebook as a PDF using the `jupyter nbconvert --to pdf HW1.ipynb` command (via a notebook code cell or the command line directly) and submit the PDF to Gradescope under the PDF submission item. If you cannot get this to work locally, you can upload the notebook to Google Colab and create the PDF there. You can find the notebook containing the instruction for this on Canvas.

If you are running the notebook locally, make sure you have created a virtual environment (using `conda` for example) and have the proper packages installed. We are working with `python=3.10` and `torch>=2`.

Files to be included in submission:

- `HW1.ipynb`
- `mlp_and.yaml`
- `mlp_and.pth`
- `mlp_or.yaml`
- `mlp_or.pth`
- `mlp_xor.yaml`
- `mlp_xor.pth`
- `mlp_xnor.yaml`
- `mlp_xnor.pth`

This homework is fully autograded. The PDF submission is supposed to familiarize you with how to save the notebook properly so all codes and plots and explanations are visible. Future assignments will be partially graded from the PDF submission.

1 Making logic gates with an MLP

Electronic circuits in computers are made up of thousands of logic gates, which take binary inputs (0 and 1 only) and produce a binary output. Shown in the figure are common logic gates. You are asked to design an MLP (Multi-layer Perceptron) to achieve the function of different logic gates. The activation function we use in our MLP is the heaviside step function which **returns 1 for positive inputs and 0 otherwise**.

```
[3]:   # DO NOT CHANGE THIS CELL OR IMPORT ANY ADDITIONAL LIBRARIES IN YOUR CODE
       import os
       import yaml
       import zipfile
       import torch
       from typing import Sequence
```

1.1 Implementing the MLP class

First, you will write a basic MLP class consisting of linear layers and perceptron neurons (step activation functions). In PyTorch, a neural network is implemented as a class that inherits from `torch.nn.Module`, which has the necessary properties and functions to be trained and many more. In this assignment, however, our activation function does not allow for training (why?) so you will assign the proper values to the parameters of the model manually to create the logic gates.

The Multi-Layer Perceptron may or may not have hidden layers. Your code should be able to handle any number of hidden layers, and the number of layers will be based on the hidden dimensions provided by the `.__init__` argument `hidden_dims`.

Example:

```
mlp = MLP(in_dim=3, out_dim=6, hidden_dims=[4,5])
```

creates an MLP with 2 hidden layers and one output layer (3 total layers). You can apply the activation function later in the `.forward` method. These two methods (`.__init__` and `.forward`) are necessary for any model you implement.

The input and output of the model should be a boolean tensor that consist of `True` and `False` values. However, you cannot perform calculations with boolean tensors, and model parameters are defined as float tensors by default. Therefore, make sure you cast the input to `torch.float` data type in the `.forward` method before performing the calculations. This will map `False` to `0.` and `True` to `1.`. After the calculations, cast the data type back to `torch.bool` so the output is boolean again.

Note: The `.forward` method of `nn.Module` can also process batched data. For example, your input tensor can be of shape `(N, in_dim)` and the output will be of shape `(N, out_dim)`. Using batched representation, you can pass several data points and test your model in a single forward pass. In the future assignments and projects, however, it is usually assumed that the input is always batched. Therefore, it is good practice to always have the batch dimension as the first dimension, even if you have a single data point.

You may find the following useful:

- `torch.nn.Linear` to build your model.
- `torch.heaviside`, `torch.ones`, `torch.ones_like`, `torch.zeros`, `torch.zeros_like` (You won't need all of these)
- using `.to()` on a tensor to change the dtype and/or device by specifying the dtype and/or device, or another tensor with the desired dtype and/or device.

If you do not find an activation function as a module in `torch.nn`, you can use the corresponding function directly in the `.forward()` method! Usually, any module without learnable parameters are also available as a function that can be directly called, like `torch.sigmoid()`.

```python
[5]: # AUTOGRADED (60)

class MLP(torch.nn.Module):

    def __init__(
            self,
            in_dim: int,
            out_dim: int,
            hidden_dims: Sequence[int],
            ):
        super().__init__()
        """
        Creates a ModuleList of Linear layers. Do not change self.layers, as
    ↪this will be used by the autograder.
        """
        self.layers = torch.nn.ModuleList() # Just to help you get started
        # Fill in self.layers using the arguments if the __init__ function
        # you can append modules one by one to a ModuleList, just like you
    ↪would with a normal list

        # Go through hidden dimensions
        current_dim = in_dim
        for hd in hidden_dims:
            self.layers.append(torch.nn.Linear(current_dim, hd, bias=True))
            current_dim = hd

        # Final output layer
        self.layers.append(torch.nn.Linear(current_dim, out_dim, bias=True))
        #NotImplemented

    def forward(
            self,
            x: torch.BoolTensor, # (batch_size, in_dim)
            ) -> torch.BoolTensor: # (batch_size, out_dim)
        """
        Implement the forward pass of the MLP.
```

```
        For each layer, pass the output of the previous layer (or the input) to␣
    ↪the linear layer, then apply the step function.
        """
        x = x.to(torch.float)
        for layer in self.layers:
            x = layer(x)
            x = torch.heaviside(x, torch.zeros_like(x))
        x = x.to(torch.bool)
        return x

        #NotImplemented
```

1.2 Find the right paramters to design logical gates

You are required to design the **smallest** model (with the fewest parameters) that behaves like the following logic gates: **AND, OR, XOR, XNOR**. First, you have to instantiate the MLP class with the right dimensions. Then, you manually assign values to the parameters of the model so that it works like the intended gate.

In order to manually assign values to the parameters, to have to access them through `self.layers`, which is a `torch.nn.ModuleList` object and can be indexed like a list. Each item in `self.layers` is a `torch.nn.Module` itself and has its own methods and attributes. In our case, each item is a linear layer. The weight and bias of a linear layer can be accessed using `.weight` and `.bias`. However, you cannot assign them values directly. For that, you have to call `.data` on the parameter object to have the proper access to assign them your desired values. To summarize, if `A` is a `torch.nn.Linear` object, you can assign a certain tensor as its weight or bias, or certain elements of them using `A.weight.data = ...` or `A.weight.data[...] = ...` where you have to choose what to put as `...`.

We recommend that you first figure out a correct set of parameters on paper, then assign them to your model. It may be easier to assign the values element by element to reduce confusion. This assignment helps you get a better understanding about what is happening inside the layers, what each parameter corresponds to, and so on.

Since we have 2 inputs and one output, `in_dim=2` and `out_dim=1`. A and B from the image are each one of the input dimensions. Remember the shape of the input tensors to the model should be `(N,2)` and the output will be `(N,1)` where `N` is the batch size or the number of data points.

You may not need any hidden layers, meaning that `hidden_dims` will be an empty list!

```
[7]: mlp_and_config = dict(
         in_dim = 2,
         out_dim = 1,
         hidden_dims = []
         )

     # the ** operator unpacks the dictionary into keyword arguments
     mlp_and = MLP(**mlp_and_config)
```

```
# ASSIGN PROPER VALUES TO THE MODEL PARAMETERS
with torch.no_grad():
    mlp_and.layers[0].weight.data = torch.tensor([[1.0, 1.0]])
    mlp_and.layers[0].bias.data = torch.tensor([-1.5])
```

```
[9]: # TEST YOUR MODEL
input_tensor = torch.tensor([
    [True, True],
    [True, False],
    [False, True],
    [False, False]
])

output_and = mlp_and(input_tensor)
print("AND Gate output:")
print(output_and)
#NotImplemented
```

```
AND Gate output:
tensor([[ True],
        [False],
        [False],
        [False]])
```

```
[11]: mlp_or_config = dict(
    in_dim = 2,
    out_dim = 1,
    hidden_dims = []
    )

mlp_or = MLP(**mlp_or_config)

# ASSIGN PROPER VALUES TO THE MODEL PARAMETERS
with torch.no_grad():
    mlp_or.layers[0].weight.data = torch.tensor([[1.0, 1.0]])
    mlp_or.layers[0].bias.data = torch.tensor([-0.5])
```

```
[13]: # TEST YOUR mlp_or
output_or = mlp_or(input_tensor)
print("OR Gate output:")
print(output_or)
#NotImplemented
```

```
OR Gate output:
tensor([[ True],
        [ True],
        [ True],
        [False]])
```

```python
[15]: mlp_xor_config = dict(
          in_dim = 2,
          out_dim = 1,
          hidden_dims = [2]
          )

      mlp_xor = MLP(**mlp_xor_config)

      # ASSIGN PROPER VALUES TO THE MODEL PARAMETERS
      with torch.no_grad():
          #hidden Layer
          mlp_xor.layers[0].weight.data = torch.tensor([[1.0, 1.0],
                                                         [1.0, 1.0]])
          mlp_xor.layers[0].bias.data = torch.tensor([-0.5, -1.5])
          #output layer
          mlp_xor.layers[1].weight.data = torch.tensor([[1.0, -2.0]])
          mlp_xor.layers[1].bias.data = torch.tensor([-0.5])
```

```python
[17]: # TEST YOUR mlp_xor
      output_xor = mlp_xor(input_tensor)
      print("XOR Gate output:")
      print(output_xor)
      #NotImplemented
```

```
XOR Gate output:
tensor([[False],
        [ True],
        [ True],
        [False]])
```

```python
[19]: mlp_xnor_config = dict(
          in_dim = 2,
          out_dim = 1,
          hidden_dims = [2]
          )

      mlp_xnor = MLP(**mlp_xnor_config)

      # ASSIGN PROPER VALUES TO THE MODEL PARAMETERS
      with torch.no_grad():
          #hidden Layer
          mlp_xnor.layers[0].weight.data = torch.tensor([[1.0, 1.0],
                                                          [1.0, 1.0]])
          mlp_xnor.layers[0].bias.data = torch.tensor([-0.5, -1.5])
          #output layer
          mlp_xnor.layers[1].weight.data = torch.tensor([[-2.0, 1.0]])
          mlp_xnor.layers[1].bias.data = torch.tensor([1.5])
```

```
[21]:  # TEST YOUR mlp_xnor
       output_xnor = mlp_xnor(input_tensor)
       print("XNOR Gate output:")
       print(output_xnor)
       #NotImplemented
```

```
XNOR Gate output:
tensor([[ True],
        [False],
        [False],
        [ True]])
```

After you create the correct models, run the code below to save your models to be included in the zip file for submission. We recommend you understand this code as well to learn how model configuration and parameters (`state_dict`) are saved.

```
[24]:  # helper functions to save and load config files
       def save_yaml(config: dict, path: str):
           with open(path, 'w') as f:
               yaml.dump(config, f, sort_keys=False, default_flow_style=None)

       def load_yaml(path: str):
           # with safe loader
           with open(path, 'r') as f:
               return yaml.load(f, Loader=yaml.FullLoader)

       def save_mlp(gate: str):
           # eval() is a function that evaluates a string as a python expression
           config: dict = eval(f'mlp_{gate}_config')
           model: torch.nn.Module = eval(f'mlp_{gate}')
           state_dict = model.state_dict()
           # saving config as yaml:
           save_yaml(config, f'mlp_{gate}.yaml')
           # saving state_dict as torch:
           torch.save(state_dict, f'mlp_{gate}.pth')

       def load_mlp(gate: str):
           # to test if model can be loaded without error
           config = load_yaml(f'mlp_{gate}.yaml')
           model = MLP(**config)
           state_dict = torch.load(f'mlp_{gate}.pth')
           model.load_state_dict(state_dict)
           return model

       if __name__ == '__main__':
           for gate in ['and', 'or', 'xor', 'xnor']:
               save_mlp(gate)
               # testing if model can be loaded without error:
```

```
        model = load_mlp(gate)
```

C:\Users\nickc\AppData\Local\Temp\ipykernel_31336\3448251871.py:25:
FutureWarning: You are using `torch.load` with `weights_only=False` (the current
default value), which uses the default pickle module implicitly. It is possible
to construct malicious pickle data which will execute arbitrary code during
unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  state_dict = torch.load(f'mlp_{gate}.pth')

2   Zip submission files

You can run the following cell to zip the generated files for submission.

If you are on Colab, make sure to download and then upload a completed copy of the notebook to
the working directory so the code can detect and include it in the zip file for submission.

```python
[28]: def zip_files(output_filename, *file_paths):
          with zipfile.ZipFile(output_filename, 'w') as zipf:
              for file_path in file_paths:
                  zipf.write(file_path, os.path.basename(file_path))

      gates = ['and', 'or', 'xor', 'xnor']
      files_to_zip = ['HW1.ipynb'] + [f'mlp_{gate}.yaml' for gate in gates] +
       [f'mlp_{gate}.pth' for gate in gates]
      output_zip = 'HW1_submission.zip'
      zip_files(output_zip, *files_to_zip)
```

```
[26]: ls

       Volume in drive C is OS
       Volume Serial Number is 84B4-2710

       Directory of C:\Users\nickc\Deep Learning

      01/22/2025  08:38 PM    <DIR>          .
      01/22/2025  03:03 PM    <DIR>          ..
      01/21/2025  04:16 PM    <DIR>          .ipynb_checkpoints
      01/22/2025  08:38 PM           169,204 HW1.ipynb
      01/22/2025  12:54 PM           177,653 HW1_submission.zip
```

```
01/22/2025  08:38 PM             1,560 mlp_and.pth
01/22/2025  08:38 PM                40 mlp_and.yaml
01/22/2025  08:38 PM             1,554 mlp_or.pth
01/22/2025  08:38 PM                40 mlp_or.yaml
01/22/2025  08:38 PM             2,136 mlp_xnor.pth
01/22/2025  08:38 PM                41 mlp_xnor.yaml
01/22/2025  08:38 PM             2,128 mlp_xor.pth
01/22/2025  08:38 PM                41 mlp_xor.yaml
              10 File(s)        354,397 bytes
               3 Dir(s)  669,695,836,160 bytes free
```

[ ]:

[39]: `!pandoc --version`

```
'pandoc' is not recognized as an internal or external command,
operable program or batch file.
```

[3]: `!jupyter nbconvert --to pdf HW1.ipynb`

```
[NbConvertApp] Converting notebook HW1.ipynb to pdf
[NbConvertApp] Writing 57504 bytes to notebook.tex
[NbConvertApp] Building PDF
Traceback (most recent call last):
  File "C:\Users\nickc\anaconda3\Scripts\jupyter-nbconvert-script.py", line 10,
in <module>
    sys.exit(main())
             ^^^^^^
  File "C:\Users\nickc\AppData\Roaming\Python\Python312\site-
packages\jupyter_core\application.py", line 283, in launch_instance
    super().launch_instance(argv=argv, **kwargs)
  File "C:\Users\nickc\AppData\Roaming\Python\Python312\site-
packages\traitlets\config\application.py", line 1075, in launch_instance
    app.start()
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\nbconvertapp.py",
line 412, in start
    self.convert_notebooks()
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\nbconvertapp.py",
line 590, in convert_notebooks
    self.convert_single_notebook(notebook_filename)
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\nbconvertapp.py",
line 556, in convert_single_notebook
    output, resources = self.export_single_notebook(
                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\nbconvertapp.py",
line 479, in export_single_notebook
    output, resources = self.exporter.from_filename(
                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-
```

```
packages\nbconvert\exporters\templateexporter.py", line 389, in from_filename
    return super().from_filename(filename, resources, **kw)  #
type:ignore[return-value]
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-
packages\nbconvert\exporters\exporter.py", line 203, in from_filename
    return self.from_file(f, resources=resources, **kw)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-
packages\nbconvert\exporters\templateexporter.py", line 395, in from_file
    return super().from_file(file_stream, resources, **kw)  #
type:ignore[return-value]
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-
packages\nbconvert\exporters\exporter.py", line 222, in from_file
    return self.from_notebook_node(
           ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\exporters\pdf.py",
line 200, in from_notebook_node
    self.run_latex(tex_file)
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\exporters\pdf.py",
line 169, in run_latex
    return self.run_command(
           ^^^^^^^^^^^^^^^^^^
  File "C:\Users\nickc\anaconda3\Lib\site-packages\nbconvert\exporters\pdf.py",
line 123, in run_command
    raise OSError(msg)
OSError: xelatex not found on PATH, if you have not installed xelatex you may
need to do so. Find further instructions at
https://nbconvert.readthedocs.io/en/latest/install.html#installing-tex.
```

[ ]: