COMP0088
Introduction to Machine Learning
Lab Assignment 6

Matthew Caldwell

November 13, 2021

Task Summary

1 Training and Testing **4**
1.1 Train a PyTorch model 4
1.2 Evaluate a PyTorch model 5

2 Multi-Layer Perceptron **6**
2.1 Initialise an MLP 6
2.2 Perform the forward pass for an MLP 6

3 Convolutional Neural Network **7**
3.1 Initialise a CNN 7
3.2 Perform the forward pass for a CNN model 7

4 Recurrent Neural Network **8**
4.1 Initialise an RNN 8
4.2 Perform the forward pass for an RNN 8

Week 6 Notes

In general it is rather unlikely that you will want to implement neural network models from scratch, as you did in week 5. Instead you will typically use components provided by a deep learning framework. In this weeks exercises you will do just that, building several different neural network models using PyTorch. These models will also be tested on somewhat more complex data than in previous weeks, doing image classification on several standard datasets.

The assignment script is `week_6.py`, generating output file `week_6.pdf`. There are some potentially-useful command line options, which you can list using:

```
$ python week_6.py -h
```

In particular, the `--data` (`-D`) option allows you to choose between the following different image datasets:

- MNIST classic dataset of handwritten digits at 28×28 resolution (60k train, 10k test)
- USPS smaller dataset of handwritten digits at 8×8 pixels (7k train, 2k test)

- Fashion-MNIST dataset of clothing images the same size as MNIST
- Kuzushiji-MNIST dataset of Japanese Hiragana characters the same size as MNIST
- CIFAR10 dataset of full colour images of animals and vehicles at 32×32 resolution (60k train, 6k test)

Only the first letter of each dataset name is needed to specify it to `-D`. The default is USPS.

Neural networks can be computationally intensive. Larger images and larger numbers of samples will train and validate more slowly. One way to speed things up is to reduce the number of classes used. All of the above datasets have 10 classes, but you can tell the script to use only a subset using the `--classes` (`-c`) option. So for example, the following command would tell the script to use only the first two classes of CIFAR10:

```
$ python week_6.py -D cifar -c 2
```

The `--model` (`-M`) option specifies which model type to build:

- `linear`: a simple logistic regression model with no hidden layers. The implementation of this is provided for you, and all other types fall back to this until you have implemented them.
- `mlp`: a mullti-layer perceptron (to be implemented in Task 2)
- `cnn`: a convolutional network (to be implemented in Task 3)
- `rnn`: a vanilla RNN
- `lstm`: an RNN with LSTM units
- `gru`: an RNN with GRU untis

All three RNN types are to be implemented in Task 4.

The `--layers` (`-y`) option adds extra configuration for the model, in a similar way to week 5. You provide a comma-separated list of numbers (without spaces). The interpretation of these differs by model type. For MLPs, it specifies the numbers of neurons in the hidden layers. For CNNs, it specifies the number of filters in each convolutional layer. For RNNs it specifies the size of the hidden state, and optionally the number of RNN layers to stack.

E.g., the following would tell the script to build a CNN with 3 convolutional layers having 8, 16 and 32 filters respectively:

```
$ python week_6.py -M cnn -y 8,16,32
```

Once you have implemented all the model classes, play around with different configurations and see how changing the model type and structure affect the behaviour. There are significant differences in the difficulty of classifying the different data sets—you are likely to be able to achieve much higher accuracy on MNIST than on CIFAR10, for example.

Example output from a completed assignment is shown in Fig. 1. Plotting functionality is provided, so your output should look similar, though the contents may vary quite a bit with different runtime configurations.

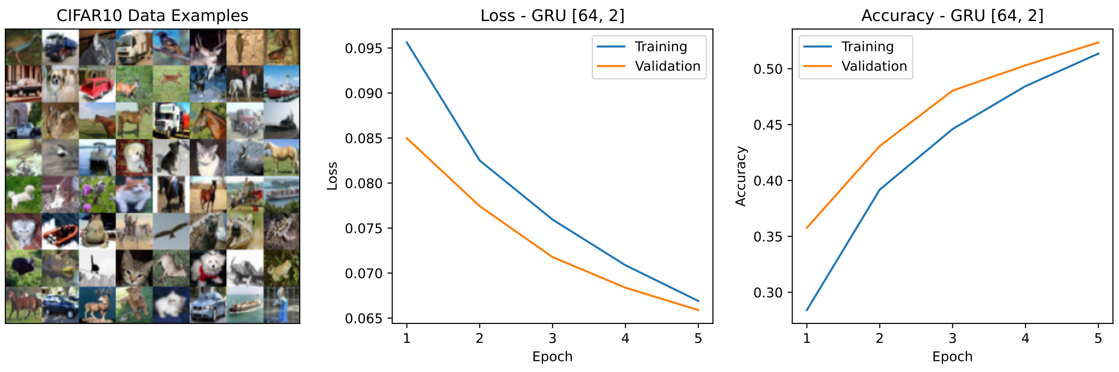

Figure 1: Example output from the `week_6.py` script

1 Training and Testing

Both training and testing require you to run through the data in mini-batches, processing each in turn. The data will be wrapped in a PyTorch DataLoader object which you can iterate over directly:

```
for X, y in dataloader:
    # ... do stuff
```

Many model layers behave differently during training and at test time—for example dropout layers only switch neurons off during training. So before invoking a model you should tell it which mode it is in. To set it to training mode call `model.train()`, to set it to evaluation mode call `model.eval()`.

To do a forward pass through the model and make predictions, you just invoke it on the data:

```
preds = model(X)
```

You can then invoke the loss function on the predictions:

```
loss = loss_function(preds, y)
```

As mentioned in lecture 6.4, although PyTorch objects look like functions or data, they are actually proxy objects that participate in operation graphs behind the scenes, potentially on a GPU. To actually get at the data you need to call an accessor function, such as `.item()`:

```
total_loss += loss.item()
```

(You can also convert Tensors to NumPy arrays with `.numpy()`.)

When you call the loss function on the predictions, the loss you get back is a Tensor connected at the end of the operation graph for your model. To perform the backward pass you just call `backward()` on this:

```
loss.backward()
```

This will propagate all the way back through the model, calculating the gradients. Before doing this, you should call `zero_grad()` on the optimiser to start from a clean slate. After the backward pass you tell the optimiser to update the parameters by calling `optimiser.step()`.

When evaluating, you don't need the model to calculate gradients. You can tell it not to by wrapping it in a `no_grad` context manager:

```
with torch.no_grad():
    # do stuff ...
```

Training even a single model epoch can take time, so you may want to print progress messages along the way just to reassure yourself that it is doing something.

1.1 Train a PyTorch model

Implement the following function in `week_6.py`:

```python
def train_epoch(model, dataloader, loss_function, optimiser):
    """
    Train a model on a single epoch of data.

    # Arguments
        model: a pytorch model (eg one of the above nn.Module subclasses)
        dataloader: a pytorch dataloader that will iterate over the dataset
            in batches
        loss_function: a pytorch loss function tensor
        optimiser: optimiser to use for training

    # Returns:
        loss: mean loss over the whole epoch
        accuracy: mean prediction accuracy over the epoch
    """
    # TODO: implement this
    return None, None
```

The return values here should be ordinary Python numbers, not PyTorch tensors.

1.2 Evaluate a PyTorch model

Implement the following function in `week_6.py`:

```python
def test_epoch(model, dataloader, loss_function):
    """
    Evaluate a model on a dataset.

    # Arguments
        model: a pytorch model (eg one of the above nn.Module subclasses)
        dataloader: a pytorch dataloader that will iterate over the dataset
            in batches
        loss_function: a pytorch loss function tensor

    # Returns:
        loss: mean loss over the whole epoch
        accuracy: mean prediction accuracy over the epoch
    """
    # TODO: implement this
    return None, None
```

Again, the return values here should be numbers, not tensors.

2 Multi-Layer Perceptron

All of the models in these exercises are going to have a fairly similar structure: they'll be classes inheriting from torch.nn.Module, with an `__init__` method to initialise the layers and a `forward` method to perform the forward pass. (The backward pass will take care of itself.)

The MLP will do much the same as what you implemented by hand last week, but built out of PyTorch layers—specifically, nn.Linear and nn.ReLU. You can wrap these in an nn.Sequential model. As noted in the docstring, you will need to use nn.Flatten to turn the image inputs into simple vectors.

2.1 Initialise an MLP

Implement the following method for the `MLP` class in `week_6.py`:

```
def __init__(self, input=INPUT_SIZE, spec=DEFAULT_MLP, output=NUM_CLASSES):
    """
    Initialise the multi-layer perceptron with the specified arrangement
    of fully-connected layers, using ReLU activation on each layer.

    Note that the data passed to this model (as for all models this
    week) will be *image* data, so the first layer in the model
    should be a Flatten layer, to turn the C * W * H pixel arrays
    into one dimensional ones.

    # Arguments
        input: the input size for the first hidden layer
        spec: a list of sizes for the intermediate layers
        output: the output size for the final layer
    """
    super().__init__(input, output)
    # TODO: implement this
```

2.2 Perform the forward pass for an MLP

Implement the following method for the `MLP` class in `week_6.py`:

```
def forward(self, x):
    """
    Execute a forward pass of the model.

    # Arguments
        x: the input data to the first layer

    # Returns
        output tensor from the last layer
    """

    # TODO: remove line below and implement this
    return super().forward(x)
```

3 Convolutional Neural Network

For a CNN, in addition to the previously mentioned layer classes, you will need nn.Conv2d. Use the same kernel size, padding and stride for all layers. Use the output dimension calculation from the lectures to keep track of the output dimensions of the convolutional layers so that you'll know how many outputs there are from the final convolutional layer. Flatten these outputs to go into a final fully-connected layer to produce the number of class prediction logits specified by the `output` argument.

3.1 Initialise a CNN

Implement the following method for the `CNN` class in `week_6.py`:

```
def __init__(self, input=INPUT_SHAPE, spec=DEFAULT_CNN, output=NUM_CLASSES,
             kernel=3, stride=2, padding=1):
    """
    Initialise the CNN with the specified arrangement of 2D convolutional
    layers and ReLU activations, with a final fully-connected layer to
    do the classification.

    # Arguments
        input: the input shape for the first convolutional layer
        spec: a list of numbers of channels for the convolutional layers
        output: the output size for the final fully-connected layer
        kernel: kernel size for all layers
        stride: convolution stride for all layers
        padding: amount of padding to add around each layer before convolving
    """
    super().__init__(np.prod(input), output)
    assert(len(spec) > 0)
    # TODO: implement this
```

3.2 Perform the forward pass for a CNN model

Implement the following method for the `MLP` class in `week_6.py`:

```
def forward(self, x):
    """
    Execute a forward pass of the model.

    # Arguments
        x: the input data to the first layer

    # Returns
        output tensor from the last layer
    """

    # TODO: remove line below and implement this
    return super().forward(x)
```

4 Recurrent Neural Network

The PyTorch layers for the three different recurrent units are nn.RNN, nn.LSTM and nn.GRU. All three have very similar interfaces and can be used interchangeably here according to the `unit_type` argument.

You will need to convert the image inputs into a stack of vector inputs. One possible way, as described in the docstring, is to treat the image rows as the input vectors, in which case the sequence length will be the number of those rows. (Other conversions are possible if you feel like experimenting.) You'll need to specify the input size when you create the recurrent unit, and also reshape the input data appropriately in your `forward` method. You may also find it helpful to specify `batch_first=True` when creating the unit.

As with the CNN, you'll need to pass the RNN outputs through a fully connected layer to generate the `output` class logits.

4.1 Initialise an RNN

Implement the following method for the `RNN` class in `week_6.py`:

```
def __init__(self, input=INPUT_SHAPE, spec=DEFAULT_RNN, output=NUM_CLASSES,
             unit_type='lstm'):
    """
    Initialise the RNN with the specified stack of recurrent layers,
    with a final fully-connected layer to do the classification.

    # Arguments
        input: the input shape for the image data.
            we assume [C, H, W] ordering and will process as a
            sequence of H inputs of size C*W
        spec: a list specifying the hidden size of each layer (at element 0)
            and optionally the number of such layers to stack (at element 1)
        output: the output size for the final fully-connected layer
        unit_type: what type of recurrent layers to use
            'lstm': use nn.LSTM
            'gru': use nn.GRU
            (anything else): use nn.RNN
    """
    super().__init__(np.prod(input), output)
    # TODO: implement this
```

4.2 Perform the forward pass for an RNN

Implement the following method for the `RNN` class in `week_6.py`:

```
def forward(self, x):
    """
    Execute a forward pass of the model.

    # Arguments
        x: the input data to the first layer
    # Returns
        output tensor from the last layer
    """

    # TODO: remove line below and implement this
    return super().forward(x)
```