École national Supérieure d'Informatique, Algiers, Algeria
2CSSID (2025/2026)
Machine Learning (ML)

Workshop 03: Neural Networks' tools (Pytorch)

Mr. Abdelkrime ARIES

Get familiar with a tool to create MLPs and train them.
Learn how to create custom MLPs using this tool.
Understand how this tool works by programming low level MLPs

1 Data Preparation

We will use the **satellite** dataset:

- Import the training dataset.
- Separate the input data and normalize it by dividing by **255**.
- Separate the output data and encode it in OneHot format using **LabelBinarizer** from **scikit-learn**.
- Transform **X_train** and **Y_train** to **torch.Tensor**.
- Redo the same with test dataset, except do not transform **Y_test** to **torch.Tensor**.

2 High Level

In this section, we will create a project to classify satellite images. We will try to use high level APIs to create an MLP. In PyTorch, the training loop is not handled implicitly; you have to define it yourself. But, you can use some other APIs such as pytorch-lightning.

2.1 Sequential model

Create a sequential model (**nn1**) composed of:

- a dense layer having a shape (number of features of the past dataset, 10);
- a ReLU activation function;
- a dense layer having a shape (10, 10);
- a ReLU activation function;
- a dense layer having a shape (10, number of classes);
- and a Softmax activation function with dim=1.

Print it to get its structure.

2.2 Model training

- Use cross entropy as loss function.
- Define Adam as an optimization function by passing **nn1.parameters()** and a learning rate
- Train **nn1** with a loop over a range of iterations of your choice.
  - Apply a forward pass by passing **X_train** into **nn1**.
  - Calculate the loss.
  - Set gradients to 0 (**optimizer.zero_grad()**).
  - Apply a backward pass using the instruction **loss.backward()**.
  - Apply an optimization step using the instruction **optimizer.step()**.
  - Print the loss.

2.3 Model testing

- Generate the estimations of **X_test** using **nn1**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

3 High level with a custom class

We want to create a custom model which adds layers one after the other. It verifies if the layers are compatibles. Then, when finished, we cannot add any more layers (we lock it).

3.1 Custom Layer

- Create a class called **MyLayer** which inherits from **torch.nn.Layer**.
- Its obligatory arguments are: the number of inputs and the number of outputs.
- Its optional arguments are: a boolean for bias (True by default) and a string of activation function to choose from 'relu', 'sigmoid' and 'linear' ('linear' by default).
- Use **assert** to verify that both the number of inputs and outputs are more than 0.
- Add the **forward** method.

- You have some unitary tests, adapt them to your model to test it.

3.2 Custom Net

- Create a class called **MyMLP** which inherits from **torch.nn.Module**.
- It has no arguments; but, it contains 2 attributes: a lock initialized to False and a list to store our layers. Do not use **list**; use **torch.nn.ModuleList** instead. This is because the latter let the module access the trainable parameters in the list.
- Add a method **add_layer** which has one argument: a layer.
  - It raises an exception when the object is locked.
  - It raises an exception when the previous layer's output is not as the current layer's input.
  - In case this is the first layer to add, no need to check.
  - It adds the layer when all is right
  - Then, it returns **self**; this allows us to call this method in one line
- Add a method **compile** which has these optional arguments: number of inputs of the output layer (=1), number of outputs of the output layer (=1), bias of the output layer (=True), a boolean indicating if it is a multi-class classification (=False), and learning rate (=1.).
  - If there are past layers, it modifies the number of inputs of the output layer to be the output of the last layer.
  - The output layer will have a sigmoid activation function
  - If it is multi-class and the number of outputs is more than 1, this function must be softmax.
  - The loss function must be BCE.
  - If it is multi-class and the number of outputs is more than 1, The loss function must be CE.
  - The optimizer must be Adam, taking this model's parameters and the bias.
  - Lock the model.
- Add the **forward** method taking **X** as argument.
- Add the **backward** method taking **X, Y** as arguments:
  - It applies a forward pass.
  - It calculates a loss using the output and **Y**.
  - It initializes the gradients of the optimizer to zero (**.zero_grad()**).
  - It applies a backward pass on the loss.
  - It applies a step of the optimizer.
  - It returns the loss as numpy (use **.detach().numpy()**) .
- Add a **fit** method taking **X, Y, epochs** as arguments. For each iteration, it shows the cost.
- Override **__call__** which returns the backward pass's result.

3.3 Model training

- Create a model (**nn2**) composed of:
  - a hidden layer having a shape (number of features of the past dataset, 10) and ReLU activation function;
  - a hidden layer having a shape (10, 10) and a ReLU activation function;
  - an output layer having a shape (10, number of classes) for multi-class classification.
- Print it to get its structure.
- Fit it on **satellite** train dataset.

3.4 Model testing

- Generate the estimations of **X_test** using **nn2**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

4 Low Level

4.1 Activation functions

- Create classes **SimpleSigmoid**, **SimpleReLU** and **SimpleSoftmax** which inherit from **torch.nn.Module**.
- No constructor is required.
- Add the **forward** method which takes **X** as argument.

4.2 Loss functions

- Create classes **SimpleBCE** and **SimpleCE** which inherit from **torch.nn.Module**.
- No constructor is required.
- Add the **forward** method which takes **X, Y** as arguments.

4.3 Optimization functions

- Create a class **SimpleGD** which inherits from **optim.Optimizer**.
- The constructor takes **params** and **lr** as arguments.
- It passes them to the super class as **super().__init__(params, defaults={'lr': lr})**
- Add the **step** method which takes no arguments. This function must update all parameters based on their gradients and their learning rate.
  - **self.groups** contains a list of parameters' groups.
  - each group is a dictionary containing, among others, a key '**lr**' and another '**params**'.
  - the value of '**lr**' is the learning rate of this group of parameters.
  - the value of '**params**' is a list of tensors. Since these tensors are 'trainable', they have an attribute **data** containing theirs current values, and another **grad** containing their current gradients. These gradients are coming from **loss.backward()** which will calculate the gradients of all the trainable parameters contributing to the loss and for each stores the gradient in its **grad** attribute.

4.4 Custom Layer

- Copy **MyLayer** and rename it **SimpleLayer**, but it must inherit from **torch.nn.Module**.
- Create an attribute **W** of type **nn.parameter.Parameter** initialized to zeros.
- Create an attribute **W** initialized to zeros. It can be only transformed to **nn.parameter.Parameter** if bias=True. In this case, it will be subscribed into the module's parameters; thus can be recovered using the method **parameters**.
- Replace all activation functions by our simple versions.
- Add a method **randomize** which randomizes all trainable parameters using a normal law (**torch.normal**) having a mean of 0 and a standard deviation of 1. Try to update their **data** attribute.

4.5 Custom Net

- Copy **MyMLP** and rename it into **SimpleMLP**.
- Modify it to handle our new structures.
- Add a method **randomize** which randomizes all layers.

4.6 Model training

- Create a model (**nn3**) composed of:
  - a hidden layer having a shape (number of features of the past dataset, 10) and ReLU activation function;
  - a hidden layer having a shape (10, 10) and a ReLU activation function;
  - an output layer having a shape (10, number of classes) for multi-class classification.
- Randomize it.
- Print its parameters. The method **parameters** returns an iterator, so you have to cast it into a list.
- Fit it on **satellite** train dataset.

4.7 Model testing

- Generate the estimations of **X_test** using **nn3**.
- Transform these estimations into classes using the method **inverse_transform** of our **LabelBinarizer**.
- Use these to print a classification report.

Appendix

- pandas.read_csv: `https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html`
- sklearn.preprocessing.LabelBinarizer: `https://scikit-learn.org/1.5/modules/generated/sklearn.preprocessing.LabelBinarizer.html`
- torch.Tensor: `https://pytorch.org/docs/main/tensors.html`
- torch.nn.Sequential: `https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html`
- torch.nn.Linear: `https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear`
- torch.nn.ReLU: `https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html#relu`
- torch.nn.Softmax: `https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html#softmax`
- torch.optim.Adam: `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#adam`

- torch.nn.CrossEntropyLoss: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html#crossentropyloss
- sklearn.metrics.classification_report: https://scikit-learn.org/1.5/modules/generated/sklearn.metrics.classification_report.html
- assert: https://docs.python.org/3/reference/simple_stmts.html#assert
- torch.nn.Sigmoid: https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html#torch.nn.Sigmoid
- Exception: https://docs.python.org/3/library/exceptions.html#Exception
- torch.nn.Module: https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module
- torch.nn.ModuleList: https://pytorch.org/docs/stable/generated/torch.nn.ModuleList.html#torch.nn.ModuleList
- torch.nn.BCELoss: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html#bceloss
- torch.exp: https://pytorch.org/docs/stable/generated/torch.exp.html#torch-exp
- torch.where: https://pytorch.org/docs/stable/generated/torch.where.html#torch-where
- torch.Tensor.view: https://pytorch.org/docs/stable/generated/torch.Tensor.view.html
- torch.sum: https://pytorch.org/docs/stable/generated/torch.sum.html#torch.sum
- torch.mean: https://pytorch.org/docs/stable/generated/torch.mean.html#torch-mean
- torch.log: https://pytorch.org/docs/stable/generated/torch.log.html#torch-log
- torch.optim.Optimizer: https://pytorch.org/docs/stable/optim.html#torch.optim.Optimizer
- torch.nn.parameter.Parameter: https://pytorch.org/docs/stable/generated/torch.nn.parameter.Parameter.html#torch.nn.parameter.Parameter
- torch.zeros: https://pytorch.org/docs/stable/generated/torch.zeros.html#torch-zeros
- torch.normal: https://pytorch.org/docs/stable/generated/torch.normal.html#torch-normal
- torch.matmul: https://pytorch.org/docs/stable/generated/torch.matmul.html#torch-matmul