Coursework 2: Artificial Neural Networks

Introduction to Machine Learning

Deadline: Friday 4th March 2022 (7pm GMT) on CATe

Overview

Please read this manual **THOROUGHLY** as it contains crucial information about the assignment, as well as the support provided.

In this assignment, you will learn to implement and optimise **neural network models** and apply them to solve a regression task. You are expected to submit a **report** (up to 5 pages) answering any questions specified and discussing your implementation and the results of your experiments. You should also submit the **SHA1 token** for the specific commit on your GitLab repository that you wish to be assessed.

This assignment is split into 2 parts:

- **Part 1: Create a neural network mini-library**. You will create a low-level implementation of a multi-layered neural network, including a basic implementation of the backpropagation algorithm. The task also involves implementing necessary functions for data preprocessing, training and evaluation.
- **Part 2: Create and train a neural network for regression**. You will develop and optimise a neural network architecture to predict the price of houses in California using the California House Prices Dataset. For this task, you will have the choice to either use `PyTorch` or the mini-library you have just developed.

The mini-library in Part 1 must be designed using `NumPy`, and will require you to implement a linear layer class, activation functions, a multi-layer network class, a trainer class, and a data preprocessing class. For this part, you may not use libraries that implement automatic differentiation, such as `PyTorch` or `TensorFlow`, but you may use basic python libraries (`os`, `sys`, `math`, etc.). This task will be assessed based on the results of corresponding `LabTS` tests and you do not need to include information about Part 1 in your report.

For the models in Part 2 you must use either the `PyTorch` neural network library or the mini-library that you developed in the first part. This part of the assignment will be primarily assessed on the basis of your report, where you describe your model, evaluation setup, the hyperparameter tuning process and the final results. There are also `LabTS` tests for Part 2 which will contribute to the final coursework score.

Setting up your environment

Please see the guidelines provided on Scientia to set up your machines for the coursework.

It is your own responsibility to ensure that your code runs on `LabTS`. **We reserve the right to reduce your marks by 30% for any bits of your code that cannot be run.**

Part 1: Create a neural network mini-library

In this part, you will implement your own modular neural network mini-library using `NumPy`. This will require you to complete a number of classes in `src/part1_nn_lib.py`. This part of the coursework will be evaluated based on your code only using `LabTS` tests. However, keep in mind that in addition to the public tests you can run by yourself on `LabTS`, the final assessment will be performed using a different set of private tests. Thus, make sure to double-check also components or functions that are not directly tested by the public tests.

A simple dataset (`https://en.wikipedia.org/wiki/Iris_flower_data_set`, given in the Gitlab repository as `iris.dat`) and sample code demonstrating intended usage is provided for debugging purposes as you work through this part of the coursework.

Q1.1: Implement a linear layer:

For this question, you must implement a linear layer (which performs an affine transformation $XW + B$ on a batch of inputs X) by completing the following methods of the `LinearLayer` class:

Constructor: In the constructor, initialise the attributes you need for this class. In particular, `NumPy` arrays representing the learnable parameters of the layer, initialized in a sensible manner (hint: you can use the provided `xavier_init` function). Use the attributes `_W, _b` to refer to your weights matrix and bias respectively.

Forward pass method: Implement the `forward` method to do the following:

- Return the outputs of the layer given a `NumPy` array representing a batch of inputs.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Backward pass method: Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the layer as input, compute the gradient of the function with respect to the parameters of the layer and store them in the relevant attributes defined in the constructor (`_grad_W_current` and `_grad_b_current`).

- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Parameter update method: Implement the `update_params` method to perform one step of gradient descent on the parameters of the layer (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```python
layer = LinearLayer(n_in=3, n_out=42)
# `inputs` shape: (batch_size, 3)
# `outputs` shape: (batch_size, 42)
outputs = layer(inputs)
# `grad_loss_wrt_outputs` shape: (batch_size, 42)
# `grad_loss_wrt_inputs` shape: (batch_size, 3)
grad_loss_wrt_inputs = layer.backward(grad_loss_wrt_outputs)
layer.update_params(learning_rate)
```

Q1.2: Implement activation function classes:

For this question, you must implement the `SigmoidLayer` and `ReluLayer` activation function classes (note: linear activation can be achieved without an activation class). For each of these two classes, complete the following:

Forward pass method: Implement the `forward` method to do the following:

- Returns the element-wise transformation of the inputs using the activation function.
- Store any data necessary for computing the gradients when later performing the backward pass in the `_cache_current` attribute.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Backward pass method: Implement the `backward` method to do the following:

- Compute and return the gradient of the function with respect to the inputs of the layer.
- You are NOT allowed to use Python loops in this method (for efficiency reasons); use vectorized operations instead.

Q1.3: Implement a multi-layer network:

For this question, you must implement a multi-layer network (consisting of stacked linear layers and activation functions) by completing the following methods of the `MultiLayerNetwork` class:

Constructor: Define the following in the constructor:

- Attribute/s containing instances of the `LinearLayer` class and activation classes, as specified by the arguments:
  - `input_dim`: an integer specifying the number of input neurons in the first linear layer,
  - `neurons`: a list specifying the number of output neurons in each linear layer, the length of the list determines the number of linear layers,
  - `activations`: a list of activation functions to apply to the output of each linear layer.
- Store your layer instances in the `_layers` attribute.

Forward pass method: Implement the `forward` method to do the following:

- Return the outputs of the network given a `NumPy` array representing a batch of inputs (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of data needed to compute gradients).

Backward pass method: Implement the `backward` method to do the following:

- Given the gradient of some scalar function with respect to the outputs of the network as input, compute the gradient of the function with respect to the parameters of the network. (note: the instances of the `LinearLayer` classes created in the constructor should automatically handle any storage of computed gradients).
- Return the gradient of the function with respect to the inputs of the network.

Parameter update method: Implement the `update_params` method to perform one step of gradient descent on the parameters of the network (using the stored gradients and the learning rate provided as argument).

Here is an example of how this class can be used:

```
# The following command will create a MultiLayerNetwork object
# consisting of the following stack of layers:
#   - LinearLayer(4, 16)
#   - ReluLayer()
#   - LinearLayer(16, 2)
#   - SigmoidLayer()
```

```
network = MultiLayerNetwork(
    input_dim=4, neurons=[16, 2], activations=["relu", "sigmoid"]
)
# `inputs` shape: (batch_size, 4)
# `outputs` shape: (batch_size, 2)
outputs = network(inputs)
# `grad_loss_wrt_outputs` shape: (batch_size, 2)
# `grad_loss_wrt_inputs` shape: (batch_size, 4)
grad_loss_wrt_inputs = network.backward(grad_loss_wrt_outputs)
network.update_params(learning_rate)
```

Q1.4: Implement a trainer:

For this question, you must implement a "Trainer" class which handles data shuffling, training a given network using minibatch gradient descent on a given training dataset, as well as computing the loss on a validation dataset. To do so, complete the following methods of the `Trainer` class:

Constructor: Define the following in the constructor:

- An attribute (`_loss_layer`) referencing an instance of a loss layer class as specified by the `loss_fun` argument (which can take values `"mse"` or `"bce"`, corresponding to the mean-squared error and binary cross-entropy losses respectively).

Data shuffling: Implement the `shuffle` method to return a randomly reordered version of the data observations provided as arguments.

Main training loop: Implement the `train` method to carry out the training loop for the network. It should loop over the following `nb_epoch` times:

- If `shuffle_flag = True`, shuffle the provided dataset using the `shuffle` method.
- Split the provided dataset into minibatches of size `batch_size` and train the network using minibatch gradient descent.

Computing evaluation loss: Implement the `eval_loss` method to compute and return the loss on the provided evaluation dataset.

Here is an example of how this class can be used:

```
trainer = Trainer(
    network=network,
    batch_size=32,
    nb_epoch=10,
    learning_rate=1.0e-3,
    shuffle_flag=True,
    loss_fun="mse",
)
trainer.train(train_inputs, train_targets)
print("Validation loss = ", trainer.eval_loss(val_inputs, val_targets))
```

Q1.5: Implement a preprocessor:

Data normalization can be crucial for effectively training neural networks. For this question, you will need to implement a "Preprocessor" class which performs min-max scaling such that the data is scaled to lie in the interval $[0, 1]$. Same as before, complete the methods of the `Preprocessor` class.

Constructor: Should compute and store data normalization parameters according to the provided dataset. This function does **not** modify the provided dataset.

Apply method: Complete the `apply` method such that it applies the pre-processing operations to the provided dataset and returns the preprocessed version.

Revert method: Complete the `revert` method such that it reverts the pre-processing operations that have been applied to the provided dataset and returns the reverted dataset. For any dataset `A`, `prep.revert(prep.apply(A))` should return `A`.

Here is an example of how this class can be used:

```
prep = Preprocessor(dataset)
normalized_dataset = prep.apply(dataset)
original_dataset = prep.revert(normalized_dataset)
```

Part 2: Create and train a neural network for regression

In this part, you will work on the California House Prices Dataset.[1] The dataset covers all the block groups in California from the 1990 Census. A block group is the smallest geographical unit for which the US Census Bureau publishes sample data, and on average it includes 1425.5 individuals living in a geographically compact area. The dataset contains ten variables:

1. **longitude:** longitude of the block group
2. **latitude:** latitude of the block group
3. **housing_median_age:** median age of the individuals living in the block group
4. **total_rooms:** total number of rooms in the block group
5. **total_bedrooms:** total number of bedrooms in the block group
6. **population:** total population of the block group

[1]This data was initially featured in the following paper: *Pace, R. Kelley, and Ronald Barry. "Sparse spatial autoregressions." Statistics Probability Letters 33.3 (1997): 291-297.*

7. **households:** number of households in the block group
8. **median_income:** median income of the households comprise in the block group
9. **ocean_proximity:** proximity to the ocean of the block group
10. **median_house_value:** median value of the houses of the block group

In this part of the coursework, you will implement a neural network architecture to infer the median house value of a block group from the value of all other attributes.

You can find the skeleton code for this part in `src/part2_house_value_regression.py`. It defines a `Regressor` class that you will complete in the following questions. This file also defines two functions `save_regressor` and `load_regressor`, to save and load a `Regressor` model using `pickle` functions. You can modify these functions if you need to but make sure to change both of them so they can work in tandem. In this part, you must use either the `PyTorch` neural network library or the mini-library that you developed in the first part. You can find an introductory tutorial for `PyTorch` in the library documentation (https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html).

You can find the data for this part of the coursework in the file `src/housing.csv`. Using these data will be a bit more complicated than for the first part: they contain both numerical and textual entries, but also missing values in some of the columns. You will need to come up with strategies for handling such data fields, both in the training data and in the unseen test data. To read this `csv` file into python3, we use the function `read_csv` from `Pandas` library, as it allows to handle multi-datatype. `Pandas` is a widely-used Python library for data-sciences. It offers a nice interface to handle and visualise large data tables. You can find a quick introduction to `Pandas` here: https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html. In the following steps, we will suggest some functions that may be useful for preprocessing the data. You are also free to use other libraries, such as `SciPy`, for preprocessing or implement your own preprocessor code. Note that the data arguments given during `LabTS` tests for Part 2 will be in the format of `Pandas` DataFrame objects.

The evaluation for this part of the coursework will be based on both your PDF report and `LabTS` test results. Make sure the report includes: 1) a description of your model, along with justification of your choices, 2) description of the evaluation setup, 3) information about the hyperparameter search you performed along with the resulting findings, 4) final evaluation of your best model. **The maximum report length is 5 pages**, including figures and tables. **Marks will be deducted if you go over this page limit**. You are welcome to include additional details in an appendix, but the main report needs to contain the necessary information, and the markers are not required to take the appendix into account.

Q2.1: Implement an architecture for regression

For this question, you will implement a neural network architecture to predict the median house value of a block group by completing the following methods of the Regressor class:

Preprocessor method: Implement the `_preprocessor` method that preprocesses the input and output of your model. It should do the following:

- Handle `Pandas` Dataframe as input, as it will be used for the `LabTS` test, as explained previously.
- Store some parameters used for the preprocessing to apply the same preprocessing method to all inputs of your model. It is important that any values necessary for data processing (e.g. normalising constants, mapping from categorical values to 1-hot vectors, etc.) are created based on the training data and the same values are applied during testing. You can use the boolean *training* parameter to determine whether the input is training data (and new dataset-wide preprocessing values should be calculated) or test/validation data (and existing values should be applied).
- Handle the missing values in the data, for example setting them to a default value. You can use the `Pandas` function `fillna` to do it (`https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html`).
- Handle the textual values in the data, encoding them using one-hot encoding. For this, you can use the `Sklearn` class `LabelBinarizer` (`https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelBinarizer.html`). Remember to store all the parameters you need to be able to re-apply your preprocessing again at test time.
- Eventually normalise the numerical values to improve learning.

Constructor method: Implement the `__init__` method to initialise your neural network model and all the attributes you need. This function takes as input the training data and apply the `_preprocessor` method to them to set the dimensions of the neural network model. You can add any input parameters you need to this method, but make sure to set them with a default value to avoid errors when testing on `LabTS`.

Model-training method: Implement the `fit` method to train your regressor model, performing the following steps number-of-epochs times:

- Perform forward pass though the model given the input.
- Compute the loss based on this forward pass.
- Perform backwards pass to compute gradients of loss with respect to parameters of the model.
- Perform one step of gradient descent on the model parameters.
- You are free to implement any additional steps to improve learning (batch-learning, shuffling...).

Here is an example of how the `Regressor` class can be used to train a regressor model on the data:

```
# Assuming x_train contains the data and y_train the corresponding targets
regressor = Regressor(x_train, nb_epoch = 10)
regressor.fit(x_train, y_train)
save_regressor(regressor)
```

Feel free to define any supplementary methods or classes you may find necessary. Detail your choice of architecture and the methodology you used in your report.

Q2.2: Set up model evaluation

The aim of this question is to propose an evaluation of your model by completing the following method:

Prediction method: Implement the `predict` method to predict the output corresponding to a given input using your model. Try to minimise the number of Python loops you use in this method using Torch tensors' properties.

Evaluation method: Implement the `score` method to print or return indicators on the performance of your regressor model. You are allowed to use utilities from libraries such as `scikit-learn`.

Detail your choice of evaluation and the methodology you used in your report, and propose an analysis of your results. You might want to illustrate your choice using graphs or tables.

Q2.3: Perform hyperparameter tuning

Using the tools you have developed so far, perform a hyperparameter search using a well thought out methodology that you will detail in your report. You are allowed to use utilities from libraries such as `scikit-learn`. Implement this search in the `RegressorHyperParameterSearch` function. Find and save your best performing model, using the provided `save_regressor` function to generate a model file which will be used to load your model on `LabTS`. Detail the methodology you used in your report. You might want to illustrate your choices with graphs or tables showing the impact of the hyperparameters.

Note that the public `LabTS` tests will report your model performance on the same training data that has been provided to you. It is not suggested to directly tune for high performance on this test, as that would result in overfitting to the training data. The final private tests will measure the performance of your saved model on a separate held-out evaluation set.

Deliverables

You will have to submit two items on CATe:

- The SHA1 corresponding to the commit on you gitlab repository (used with `LabTS`). In this repository, you should have completed and pushed:
  - Completed version of `part1_nn_lib.py`.
  - Completed version of `part2_house_value_regression.py`.

  - Uploaded your best model as `part2_model.pickle`. This has to be saved by `save_regressor` and will be loaded by `load_regressor`.
  - Uploaded a `README.md` explaining how to run your code.

- Report (PDF) for Part 2 with a **maximum length of 5 pages**, including figures and tables. The report should include:
  - A description of your model, along with justification of your choices.
  - A description of the evaluation setup.
  - Information about the hyperparameter search you performed along with the resulting findings.
  - The final evaluation of your best model.

Important Notes

1. **Important!** Make sure to evaluate on `LabTS` early, to ensure that your code runs there correctly. You will lose points on any tests where the code fails or does not run.
2. Make sure to have at least one working commit tested on `LabTS` a few days before the deadline. The `LabTS` queues for running tests will likely get long as the deadline approaches and your tests might not complete on time if you don't submit them early enough.
3. The performance test evaluates your pretrained model from the pickle file. For this to work, make sure you have actually committed your pickle file into the github repository.
4. If you create your `part2_model.pickle` file on a machine with incompatible versions of libraries, then you might experience errors on `LabTS`. We recommend creating the pickle file in one of the lab machines, using the environment that is given in the spec. If you want to work on your own machine, make sure your python library versions match the ones given in the `requirements.txt` file on Scientia.
5. Note that LabTS does not have GPUs during testing. Depending on how you've saved your model, it might be looking for a GPU when loading, resulting in warnings or errors. If that happens, try training and saving your model using only your CPU.
6. Avoid committing Python environments, packages, compiled Python files, cache files, etc. to your GitLab repository. These take up unnecessary space and slow down `LabTS`. Use a `.gitignore` file to ignore any local files that you do not wish to be committed.
7. Only one person can be a leader in a given group and submit the coursework on CATe. If multiple people have declared themselves as leaders, they won't be able to add each other to the group. If this happens, someone needs to delete their declaration (and possibly their submission) so that the chosen group leader can add them. Only after that should they sign their declaration.
8. All team members need to sign the coursework declaration. As long as the team leader submitted and signed on time, there is no penalty for the other members to sign late. But all the team members still need to do the signing.

Grading scheme

Part 1 will be evaluated only based on `LabTS` results.

Part 2 will be assessed based on your report along with the `LabTS` tests.

Total = 100 marks.

- Part 1 – *based on LabTS tests only* (50 marks):
  - Linear layer (10 marks)
  - Activation functions (10 marks)
  - Multilayer network (10 marks)
  - Trainer (10 marks)
  - Preprocessor (10 marks)
- Part 2 – *based on LabTS and report* (50 marks):
  - Passing LabTS tests for Part 2 (15 marks)
  - Implement an architecture for regression (14 marks)
  - Set up model evaluation (11 marks)
  - Perform hyperparameter tuning (10 marks)