Transfer NLP

Release 0.0.3

Peter Martigny

May 15, 2019

NOTES

`Transfer NLP` is a framework built on top of PyTorch which goal is to achieve 2 kinds of Transfer:

- **easy transfer of code**: the framework should be modular enough so that you don't have to re-write everything each time you experiment with a new architecture / a new kind of task
- **easy transfer learning**: the framework should be able to easily interact with pre-trained models and manipulate them in order to fine-tune some of their parts.

You can try the library on this Colab Notebook., which shows how to use the framework on several examples. All examples on these notebooks embed in-cell Tensorboard training monitoring!

CHAPTER

ONE

INSTALLATION

From source:

You can clone the source from github and run

```
python setup.py install
```

1.1 Concepts

1.1.1 Experiment

The **essence** of the framework is the class *ExperimentConfig*, a class which enables to define an experiment based on a json file. An experiment will contain all the components that you might need: - Data loader - Model - Optimizer - Trainer - . . .

Launching experiments from json config files has two main advantages:

- **reproducibility**: when you are happy with the outcome of an experiment, the json file you used defines it entirely, so it is really easy to reproduce
- **ablation studies**: when experimenting with new architectures, it is becoming a standards practice to assess the importance of some model components to the outcome.

Using json files facilitates this process, where you just have to remove some components from the json file and run the experiment again.

```
from transfer_nlp.plugins.config import ExperimentConfig

# Defining an experiment and starting the training pipeline
experiment_config = {...}  # Config dictionary with components defining your␣
↪experiment
experiment = ExperimentConfig(experiment_config)
experiment['trainer'].train()

# Using the trained model to make predictions on some inputs
predictor = experiment['predictor']
json_input = {'inputs': []}
results = predictor.json_to_json(input_json=input_json)
```

1.1.2 Json file

The class `ExperimentConfig` has been designed so that an experiment can be instantiated from any kind of objects you might need. The experiment instantiator is able to deal with 3 kinds of inputs from the json files:

- **Simple parameters**: these are simple user-defined values, such as:

```
experiment_config = {"lr": 0.01,
                     "seed": 1,
                     "num_epochs": 1}
```

- **simple lists**: this is the same as simple parameters, but using lists, e.g.:

```
experiment_config = {"layer_sizes": [10, 50, 10]}
```

- **complex configuration**: here you can instantiate an object from any class. The framework will require the json file to contain the name of the used class, e.g.:

```
experiment_config = {"lr": 0.01,
                     "model": {"_name": "MyClassifier"}}
```

When creating an instance of the class, *ExperimentConfig* will check for the hyperparameters. If it does not find them and the class defines default parameters, those will be used. Otherwise, an exception will be thrown. So in this example if the *MyClassifier* class takes *input_dim* and *output_dim* as hyperparameters, you would define the experiment as:

```
experiment_config = {"input_dim": 10000,
                     "output_dim": 5,
                     "model": {"_name": "MyClassifier"}}
```

or:

```
experiment_config = {"model": {"_name": "MyClassifier",
                     "input_dim": 10000,
                     "output_dim": 5}}
```

If one of your objects takes another complex object as initialization parameter, *ExperimentConfig* can build it recursively, e.g.:

```
experiment_config = {
"my_dataset_splits": {
"_name": "SurnamesDatasetMLP",
"data_file": "$HOME/surnames/surnames_with_splits.csv",
"batch_size": 128,
"vectorizer": {
  "_name": "SurnamesVectorizerMLP",
  "data_file": "$HOME/surnames/surnames_with_splits.csv"
}
}
```

The framework encourages the use of this nesting definition for clarity. However, in this example if the object *vectorizer* was needed to initialize another object in your experiment, you should isolate this multi-use object. Objects which will use it will call a reference to that object using the common *$* notation. This enables to not defining different objects when we don't need them.

```
experiment_config = {

  "common_object": {
```

(continues on next page)

(continued from previous page)

```
  "_name": "MyCommonObject",
  "some_parameter": "foo/bar"
  },
  "complex_object_A": {
"_name": "ComplexObjectA",
"common_object": "$common_object"
},
      "complex_object_B": {
"_name": "ComplexObjectB",
"common_object": "$common_object"
}
}
```

To let Transfer NLP know about your custom classes, you add them to a registry. The framework does not require using separate registries for some fixed set of components, such as Models, Optimizers, etc.. There is an only one registry of classes, where you need to add your custom classes to use the framework.

Let's say you have a fancy model class that extends the PyTorch neural network module class. The only thing you need to do is add the class to the registry using the *@register_plugin* decorator:

```
import torch
from transfer_nlp.plugins.config import register_plugin

@register_plugin
class MyClassifier(torch.nn.Module):
    def __init__(self, input_dim: int, ouput_dim: int):

        super(MyClassifier, self).__init__()

    def forward(self, input_tensor):
        # Do complex transofmrations
        return result
```

Finally, to enable the sharing of experiment configuration files, we can use environment variables for paths parameters, and the framework will automatically replace them:

```
experiment_config = {
"my_dataset_splits": {
"_name": "SurnamesDatasetMLP",
"data_file": "$HOME/surnames/surnames_with_splits.csv",
"batch_size": 128,
"vectorizer": {
  "_name": "SurnamesVectorizerMLP",
  "data_file": "$HOME/surnames/surnames_with_splits.csv"
}
}
experiment = ExperimentConfig(path, HOME=str(Path.home() / 'data'))  # Changes $HOME␣
↪to a custom folder
```

1.1.3 Final thoughts

In the core design of Transfer NLP, the framework allows any kind of experiment to be instantiated, run, checkpointed, monitored, etc... The framework is not PyTorch-specific at its core, which make it easy to extend to objects using other machine learning backends such as tensorflow. Although the framework allows this flexibility, we will start focusing on PyTorch for next steps on our end. You are very welcome to contribute with Tensorflow building blocks to run easily-customizable experiments! In the long-run we hope that Transfer NLP becomes backend-agnostic and can enable any kind of ML experiments.

1.2 Data Management Components

1.2.1 Vocabularies

We provide classes to build vocabularies over datasets. These classes do not take into account the nature of the symbols whith which you are filling a dictionary. Hence, whether you want to use vocabularies for tokens, characters, BPE, etc.., you can still use the vocabulary classes coupled with a vectorizer of your choice.

1.2.2 Vectorizers

Vectorizers take string inputs and converts hem to lists of symbnols. When implementing your vectorizer, you need to build the vocabularies that you need for your experiment, and set these vocabularies as vectorizer attributes. You also need to implement the *vectorize* method, which turns a string input into a list of numbers representing the symbols you choose to use to represent the text.

1.2.3 Loaders

Data Loaders splits te dataset into train, validation and test sets, and creates the appropriate PyTorch DataLoaders.

1.3 Modeling Components

While the framework is flexible enough to deal with any kind of objects, here are some baseline components that you can use:

1.3.1 Models

A model extends the PyTorch *torch.nn.Module* class. You only have to define implement the __*init*__ and the *forward* classes. Your model class will have hyperparameters (which are used at object creation), and parameters for the *forward* method (used when __*call*__ is called). The parameters that the *forward* method expects should match the parameters yield by the PyTorch batch iterator. For example:

```python
import torch
from transfer_nlp.plugins.config import register_plugin

@register_plugin
class MyClassifier(torch.nn.Module):
    def __init__(self, input_dim: int, ouput_dim: int):

        super(MyClassifier, self).__init__()

    def forward(self, input_tensor: torch.tensor):
        # Do complex transofmrations
        return result
```

In this example, you need to set your data loader to yield batches with the key *"input_tensor"*. If the *forward* method has default parameters that do not appear in the batch, they will be used, otherwise tyey will be replaced by the values from the batch

1.3.2 Optimizers

Optimizers allows for moving the model parameters in the direction of their gradients, following the strategy proper of a certain optimizer. The framework registry comes with all PyTorch optimizers so you should be good to go for most cases, e.g.:

```
experiment_config = {
                        "optimizer": {"_name": "Adam",
                                      "params": "model_params"
                                      }
                     }
```

However, if you want to use a custom Optimizer, you need to extend the *torch.optim.Optimizer* class and register it to the registry. For example, if we want to use the optimizer used for BERT, we can use this implementation and register it like this:

```
@register_plugin
class BertAdam(Optimizer):

    def __init__(self, params, lr=required, warmup=-1, t_total=-1, schedule='warmup_
↪linear',
                 b1=0.9, b2=0.999, e=1e-6, weight_decay=0.01,
                 max_grad_norm=1.0):

        super(BertAdam, self).__init__(params, defaults)
    def step(self, closure=None):
        # Compute the loss
        return loss


experiment_config = {
                        "optimizer": {"_name": "BertAdam",
                                      "params": "model_params"
                                      }
                     }
```

1.4 Trainer Components

While the framework is flexible enough to deal with any kind of trainers, we encourage the use of a framework to manage your training loops. We found that Ignite provides everything we could expect from a training management system.

Ignite defines 6 classes of events, defining a training loop:

- STARTED: start the training loop
- EPOCH_STARTED: start an epoch
- ITERATION_STARTED: start processing of one batch
- ITERATION_COMPLETED: complete processing of one batch

- EPOCH_COMPLETED: complete a full epoch
- COMPLETED: complete the training loop

Ignite allows to perform some actions at each of these events, by simply adding events.

Here are some examples of events you can do:

- Track metrics and log them on the terminal
- Log metrics, parameters norms, histograms, distributions, etc.. to Tensorboard (via TensorboardX)
- Learning schedulers: adapt the learning rates at different times of the training. A good example is the Cyclical learning rate scheduling, which has proven successful in models like ULMFit
- Model checkpointing: save your model periodically if it improves
- Early stopping: stop training when no learning is ever observed
- Terminate on NaNs: terminates the training when nans or infinite values are encountered.
- Timers
- . . .

We provide a *BasicTrainer* class which should set you up for most cases in the supervised single task setting. For more complex settings like multi-task learning, you might want to change the *_update* and *_inference* methods to fit several tasks objectives / loss functions.

1.5 Frequently Asked Questions

1.6 vocabulary

class `transfer_nlp.loaders.vocabulary.`**`Vocabulary`**(*token2id: Dict = None*, *add_unk: bool = True*, *unk_token: str = '<UNK>'*)

1.7 vectorizer

class `transfer_nlp.loaders.vectorizers.`**`Vectorizer`**(*data_file: str*)

1.8 loader

class `transfer_nlp.loaders.loaders.`**`DatasetSplits`**(*train_set: torch.utils.data.Dataset*, *train_batch_size: int*, *val_set: torch.utils.data.Dataset*, *val_batch_size: int*, *test_set: torch.utils.data.Dataset = None*, *test_batch_size: int = None*)

This file contains an abstract CustomDataset class, on which we can build up custom dataset classes.

In your project, you will have to customize your data loader class. To let the framework interact with your class, you need to use the decorator @register_dataset, just as in the examples in this file

class `transfer_nlp.loaders.loaders.`**`DataFrameDataset`**(*df*)

class transfer_nlp.loaders.loaders.**DatasetHyperParams**(*vectorizer: transfer_nlp.loaders.vectorizers.Vectorizer*)

1.9 config

class transfer_nlp.plugins.config.**ExperimentConfig**(*experiment: Union[str, pathlib.Path, Dict], **env*)

This file contains all necessary plugins classes that the framework will use to let a user interact with custom models, data loaders, etc…

The Registry pattern used here is inspired from this post: https://realpython.com/primer-on-python-decorators/

class transfer_nlp.plugins.config.**ConfigFactoryABC**

class transfer_nlp.plugins.config.**ParamFactory**(*param*)

Factory for simple parameters

class transfer_nlp.plugins.config.**PluginFactory**(*cls, param2config_key: Optional[Dict[str, str]], *args, **kwargs*)

Factory for complex objects creation

exception transfer_nlp.plugins.config.**UnconfiguredItemsException**(*items*)

1.10 trainers

class transfer_nlp.plugins.trainers.**BasicTrainer**(*model: torch.nn.Module, dataset_splits: transfer_nlp.loaders.loaders.DatasetSplits, loss: torch.nn.Module, optimizer: torch.optim.Optimizer, metrics: Dict[str, ignite.metrics.Metric], experiment_config: transfer_nlp.plugins.config.ExperimentConfig, device: str = None, num_epochs: int = 1, seed: int = None, cuda: bool = None, loss_accumulation_steps: int = 4, scheduler: Any = None, regularizer: transfer_nlp.plugins.regularizers.RegularizerABC = None, gradient_clipping: float = 1.0, output_transform=None, tensorboard_logs: str = None, embeddings_name: str = None, finetune: bool = False*)

freeze_and_replace_final_layer()

Freeze al layers and replace the last layer with a custom Linear projection on the predicted classes Note: this method assumes that the pre-trained model ends with a *classifier* layer, that we want to learn :return:

train()

Launch the ignite training pipeline If fine-tuning mode is granted in the config file, freeze all layers, replace classification layer by a Linear layer and reset the optimizer :return:

This class contains the abstraction interface to customize runners. For the training loop, we use the engine logic from pytorch-ignite

Check experiments for examples of experiment json files

class transfer_nlp.plugins.trainers.**TrainingMetric**(*metric: ignite.metrics.Metric*)

1.11 predictors

class transfer_nlp.plugins.predictors.**PredictorABC**(*vectorizer: transfer_nlp.loaders.vectorizers.Vectorizer, model: torch.nn.Module*)

decode(**args, **kwargs*) → List[Dict]
Return an output dictionary for every example in the batch :param args: :param kwargs: :return:

forward(*batch: Dict[str, Any]*) → torch.tensor
Do the forward pass :param batch: :return:

json_to_data(*input_json: Dict*) → Dict
Transform a json entry into a data example, which is the same that what the __getitem__ method in the data loader, except that this does not output any expected label as in supervised setting :param input_json: :return:

json_to_json(*input_json: Dict*) → Dict[str, Any]
Full prediction: input_json –> data example –> predictions –> json output :param input_json: :return:

output_to_json(**args, **kwargs*) → Dict[str, Any]
Convert the result into a proper json :param args: :param kwargs: :return:

predict(*batch: Dict[str, Any]*) → List[Dict]
Decode the output of the forward pass :param batch: :return:

1.12 regularizers

class transfer_nlp.plugins.regularizers.**RegularizerABC**

1.13 Surnames Classification

A use case that arise very often in the book NLP with PyTorch is that of surnames classification: a dataset of names from different countries is provided and the task is to predict the country.

1.13.1 Vectorizer

The most straigthforward to represent a surname is to get its one-hot character encoding:

```
import pandas as pd
import numpy as np
from transfer_nlp.loaders.vocabulary import Vocabulary


@register_plugin
```

(continues on next page)

(continued from previous page)

```
class MyVectorizer(Vectorizer):

    def __init__(self, data_file: str):

        super().__init__(data_file=data_file)

        df = pd.read_csv(data_file)
        data_vocab = Vocabulary(unk_token='@')
        target_vocab = Vocabulary(add_unk=False)

        # Add surnames and nationalities to vocabulary
        for index, row in df.iterrows():
            surname = row.surname
            nationality = row.nationality
            data_vocab.add_many(tokens=surname)
            target_vocab.add_token(token=nationality)

        self.data_vocab = data_vocab
        self.target_vocab = target_vocab

    def vectorize(self, input_string: str) -> np.array:

        encoding = np.zeros(shape=len(self.data_vocab), dtype=np.float32)
        for character in surname:
            encoding[self.data_vocab.lookup_token(token=character)] = 1

        return encoding
```

1.13.2 Data loader

Let's create a data loader and have the PyTorch loaders set for train, vaildation and test categories.

```
from transfer_nlp.loaders.loaders import DatasetSplits, DataFrameDataset,␣
↪DatasetHyperParams

@register_plugin
class MyDataLoader(DatasetSplits):

    def __init__(self, data_file: str, batch_size: int, dataset_hyper_params:␣
↪DatasetHyperParams):
        self.df = pd.read_csv(data_file)
        self.vectorizer: Vectorizer = dataset_hyper_params.vectorizer

        self.df['x_in'] = self.df.apply(lambda row: self.vectorizer.vectorize(row.
↪surname), axis=1)
        self.df['y_target'] = self.df.apply(lambda row: self.vectorizer.target_vocab.
↪lookup_token(row.nationality), axis=1)

        train_df = self.df[self.df.split == 'train'][['x_in', 'y_target']]
        val_df = self.df[self.df.split == 'val'][['x_in', 'y_target']]
        test_df = self.df[self.df.split == 'test'][['x_in', 'y_target']]

        super().__init__(train_set=DataFrameDataset(train_df), train_batch_size=batch_
↪size,
                         val_set=DataFrameDataset(val_df), val_batch_size=batch_size,
```

(continues on next page)

(continued from previous page)

```
                            test_set=DataFrameDataset(test_df), test_batch_size=batch_
↪size)
```

1.13.3 Model

A simple modeling approach is to take the character one-hot encoding as input to a multi-layer perceptron:

```
import torch

@register_plugin
class ModelHyperParams(ObjectHyperParams):

    def __init__(self, dataset_splits: DatasetSplits):
        super().__init__()
        self.input_dim = len(dataset_splits.vectorizer.data_vocab)
        self.output_dim = len(dataset_splits.vectorizer.target_vocab)


@register_plugin
class MultiLayerPerceptron(torch.nn.Module):

    def __init__(self, model_hyper_params: ObjectHyperParams, hidden_dim: int):
        super(MultiLayerPerceptron, self).__init__()

        self.input_dim = model_hyper_params.input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = model_hyper_params.output_dim

        self.fc1 = torch.nn.Linear(in_features=self.input_dim, out_features=hidden_
↪dim)
        self.fc2 = torch.nn.Linear(in_features=hidden_dim, out_features=self.output_
↪dim)

    def forward(self, x_in: torch.tensor) -> torch.tensor:
        """
        Linear -> ReLu -> Linear (+ softmax if probabilities needed)
        :param x_in: size (batch, input_dim)
        :return:
        """
        intermediate = torch.nn.functional.relu(self.fc1(x_in))
        output = self.fc2(intermediate)

        if self.output_dim == 1:
            output = output.squeeze()

        return output
```

1.13.4 Predictor

To use the model in inference mode, we create a specific predictor object:

```
from transfer_nlp.plugins.predictors import PredictorABC, PredictorHyperParams
from transfer_nlp.plugins.config import register_plugin
```

(continues on next page)

(continued from previous page)

```
@register_plugin
class MyPredictor(PredictorABC):

    def __init__(self, predictor_hyper_params: PredictorHyperParams):
        super().__init__(predictor_hyper_params=predictor_hyper_params)

    def json_to_data(self, input_json: Dict):
        return {
            'x_in': torch.tensor([self.vectorizer.vectorize(input_string=input_
↪string) for input_string in input_json['inputs']])}

    def output_to_json(self, outputs: List) -> Dict[str, Any]:
        return {
            "outputs": outputs}

    def decode(self, output: torch.tensor) -> List[Dict[str, Any]]:
        probabilities = torch.nn.functional.softmax(output, dim=1)
        probability_values, indices = probabilities.max(dim=1)
        return [{
            "class": self.vectorizer.target_vocab.lookup_index(index=int(res[1])),
            "probability": float(res[0])} for res in zip(probability_values, indices)]
```

1.13.5 Experiment

Now that all classes are properly designed, we can define an experiment in a config file and have it trained:

```
  from transfer_nlp.plugins.config import ExperimentConfig

  experiment_config = {
"predictor": {
  "_name": "MLPPredictor",
  "data": "$my_dataset_splits",
  "model": "$model"
},
"my_dataset_splits": {
  "_name": "SurnamesDatasetMLP",
  "data_file": "$HOME/surnames/surnames_with_splits.csv",
  "batch_size": 128,
  "vectorizer": {
    "_name": "SurnamesVectorizerMLP",
    "data_file": "$HOME/surnames/surnames_with_splits.csv"
  }
},
"model": {
  "_name": "MultiLayerPerceptron",
  "hidden_dim": 100,
  "data": "$my_dataset_splits"
},
"optimizer": {
  "_name": "Adam",
  "lr": 0.01,
  "alpha": 0.99,
  "params": {
    "_name": "TrainableParameters"
```

(continues on next page)

(continued from previous page)

```
  }
},
"scheduler": {
  "_name": "ReduceLROnPlateau",
  "patience": 1,
  "mode": "min",
  "factor": 0.5
},
"trainer": {
  "_name": "BasicTrainer",
  "model": "$model",
  "dataset_splits": "$my_dataset_splits",
  "loss": {
    "_name": "CrossEntropyLoss"
  },
  "optimizer": "$optimizer",
  "gradient_clipping": 0.25,
  "num_epochs": 5,
  "seed": 1337,
  "regularizer": {
    "_name": "L1"
  },
  "tensorboard_logs": "$HOME/surnames/tensorboard/mlp",
  "metrics": {
    "accuracy": {
      "_name": "Accuracy"
    },
    "loss": {
      "_name": "LossMetric",
      "loss_fn": {
        "_name": "CrossEntropyLoss"
      }
    }
  }
}

  # Configure the experiment
  experiment = ExperimentConfig(experiment_config)
  # Launch the training loop
  experiment['trainer'].train()
  # Use the predictor for inference
  input_json = {"inputs": ["Zhang", "Mueller", "Rastapopoulos"]}
  output_json = experiment['predictor'].json_to_json(input_json=input_json)
```

1.14 License

1.15 Contact

Contact peter.martigny@gmail.com

1.16 Help

Contact peter.martigny@gmail.com

1.17 Join the Feedly Lab Slack

Join the Feedly Lab Slack on this link.

PYTHON MODULE INDEX

INDEX