ProtoTorch Documentation

Release 0.4.4

Jensun Ravichandran

Jun 18, 2021

CONTENTS

1 A short tutorial for the `prototorch.models` plugin **1**

1.1 Introduction . . . 1
1.2 Basics . . . 1
1.2.1 Building Models . . . 1
1.2.2 Data . . . 2
1.2.3 Training . . . 3
1.2.4 From data to a trained model - a very minimal example . . . 3
1.3 Advanced . . . 4
1.3.1 Initializing prototypes with a subset of a dataset (along with transformations) . . . 4
1.4 FAQs . . . 5
1.4.1 How do I Retrieve the prototypes and their respective labels from the model? . . . 5
1.4.2 How do I make inferences/predictions/recall with my trained model? . . . 5

2 Models **7**

2.1 Unsupervised Methods . . . 9
2.2 Classical Learning Vector Quantization . . . 9
2.3 Generalized Learning Vector Quantization . . . 9
2.4 Probabilistic Models . . . 9
2.5 Classification by Component . . . 9

3 Visualization **11**

4 Bibliography **13**

5 Abstract Models **15**

6 About **17**

7 Library **19**

8 Customizable **21**

Bibliography **23**

CHAPTER

ONE

A SHORT TUTORIAL FOR THE PROTOTORCH.MODELS PLUGIN

1.1 Introduction

This is a short tutorial for the models plugin of the ProtoTorch framework.

ProtoTorch provides torch.nn modules and utilities to implement prototype-based models. However, it is up to the user to put these modules together into models and handle the training of these models. Expert machine-learning practioners and researchers sometimes prefer this level of control. However, this leads to a lot of boilerplate code that is essentially same across many projects. Needless to say, this is a source of a lot of frustration. PyTorch-Lightning is a framework that helps avoid a lot of this frustration by handling the boilerplate code for you so you don't have to reinvent the wheel every time you need to implement a new model.

With the prototorch.models plugin, we've gone one step further and pre-packaged commonly used prototype-models like GMLVQ as Lightning-Modules. With only a few lines to code, it is now possible to build and train prototype-models. It quite simply cannot get any simpler than this.

1.2 Basics

First things first. When working with the models plugin, you'll probably need `torch`, `prototorch` and `pytorch_lightning`. So, we recommend that you import all three like so:

[1]:
```
import prototorch as pt
import pytorch_lightning as pl
import torch
```

1.2.1 Building Models

Let's start by building a `GLVQ` model. It is one of the simplest models to build. The only requirements are a prototype distribution and an initializer.

[2]:
```
model = pt.models.GLVQ(
    hparams=dict(distribution=[1, 1, 1]),
    prototype_initializer=pt.components.Zeros(2),
)
```

[3]:
```
print(model)
```

```
GLVQ(
  (proto_layer): LabeledComponents(components.shape: (3, 2))
  (acc_metric): Accuracy()
)
```

The key `distribution` in the `hparams` argument describes the prototype distribution. If it is a Python list, it is assumed that there are as many entries in this list as there are classes, and the number at each location of this list describes the number of prototypes to be used for that particular class. So, `[1, 1, 1]` implies that we have three classes with one prototype per class. If it is a Python tuple, a shorthand of `(num_classes, prototypes_per_class)` is assumed. If it is a Python dictionary, the key-value pairs describe the class label and the number of prototypes for that class respectively. So, `{0: 2, 1: 2, 2: 2}` implies that we have three classes with labels `{1, 2, 3}`, each equipped with two prototypes. If however, the dictionary contains the keys `"num_classes"` and `"prototypes_per_class"`, they are parsed to use their values as one might expect.

The `prototype_initializer` argument describes how the prototypes are meant to be initialized. This argument has to be an instantiated object of some kind of ComponentInitializer. If this is a DimensionAwareInitializer, this only requires a dimension arugment that describes the vector dimension of the prototypes. So, `pt.components.Zeros(2)` creates 2d-vector prototypes all initialized to zeros.

It is also possible to use a ClassAwareInitializer. However, this type of initializer requires data to be instantiated.

For a full list of available models, please check the prototorch_models documentation.

1.2.2 Data

The preferred way to working with data in `torch` is to use the Dataset and Dataloader API. There a few pre-packaged datasets available under `prototorch.datasets`. See here for a full list of available datasets.

[4]:
```
train_ds = pt.datasets.Iris(dims=[0, 2])
```

[5]:
```
type(train_ds)
```

[5]:
```
prototorch.datasets.iris.Iris
```

[6]:
```
train_ds.data.shape, train_ds.targets.shape
```

[6]:
```
((150, 2), (150,))
```

Once we have such a dataset, we could wrap it in a `Dataloader` to load the data in batches, and possibly apply some transformations on the fly.

[7]:
```
train_loader = torch.utils.data.DataLoader(train_ds, batch_size=2)
```

[8]:
```
type(train_loader)
```

[8]:
```
torch.utils.data.dataloader.DataLoader
```

[9]:
```
x_batch, y_batch = next(iter(train_loader))
print(f"{x_batch=}, {y_batch=}")
```

```
x_batch=tensor([[5.1000, 1.4000],
        [4.9000, 1.4000]]), y_batch=tensor([0., 0.])
```

This perhaps seems like a lot of work for a small dataset that fits completely in memory. However, this comes in very handy when dealing with huge datasets that can only be processed in batches.

1.2.3 Training

If you're familiar with other deep learning frameworks, you might perhaps expect a `.fit(...)` or `.train(...)` method. However, in PyTorch-Lightning, this is done slightly differently. We first create a trainer and then pass the model and the Dataloader to `trainer.fit(...)` instead. So, it is more functional in style than object-oriented.

[10]:
```
trainer = pl.Trainer(max_epochs=2, weights_summary=None)
```

```
GPU available: False, used: False
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
TPU available: False, using: 0 TPU cores
```

[11]:
```
trainer.fit(model, train_loader)
```

```
/home/blackfly/pyenvs/pt/lib/python3.9/site-packages/pytorch_lightning/utilities/
↪distributed.py:69: UserWarning: you defined a validation_step but have no val_
↪dataloader. Skipping val loop
  warnings.warn(*args, **kwargs)
```

```
Validation sanity check: 0it [00:00, ?it/s]
```

```
/home/blackfly/pyenvs/pt/lib/python3.9/site-packages/pytorch_lightning/utilities/
↪distributed.py:69: UserWarning: The dataloader, train dataloader, does not have many␣
↪workers which may be a bottleneck. Consider increasing the value of the `num_workers`␣
↪argument` (try 6 which is the number of cpus on this machine) in the `DataLoader` init␣
↪to improve performance.
  warnings.warn(*args, **kwargs)
```

```
Training: 0it [00:00, ?it/s]
```

1.2.4 From data to a trained model - a very minimal example

[12]:
```
train_ds = pt.datasets.Iris(dims=[0, 2])
train_loader = torch.utils.data.DataLoader(train_ds, batch_size=32)

model = pt.models.GLVQ(
    dict(distribution=(3, 2), lr=0.1),
    prototype_initializer=pt.components.SMI(train_ds),
)

trainer = pl.Trainer(max_epochs=50, weights_summary=None)
trainer.fit(model, train_loader)
```

```
GPU available: False, used: False
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
TPU available: False, using: 0 TPU cores
```

```
Validation sanity check: 0it [00:00, ?it/s]
```

```
Training: 0it [00:00, ?it/s]
```

1.3 Advanced

1.3.1 Initializing prototypes with a subset of a dataset (along with transformations)

[13]:
```
import prototorch as pt
import pytorch_lightning as pl
import torch
from torchvision import transforms
from torchvision.datasets import MNIST
```

[14]:
```
from matplotlib import pyplot as plt
```

[15]:
```
train_ds = MNIST(
    "~/datasets",
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.RandomHorizontalFlip(p=1.0),
        transforms.RandomVerticalFlip(p=1.0),
        transforms.ToTensor(),
    ]),
)
```

[16]:
```
s = int(0.05 * len(train_ds))
init_ds, rest_ds = torch.utils.data.random_split(train_ds, [s, len(train_ds) - s])
```

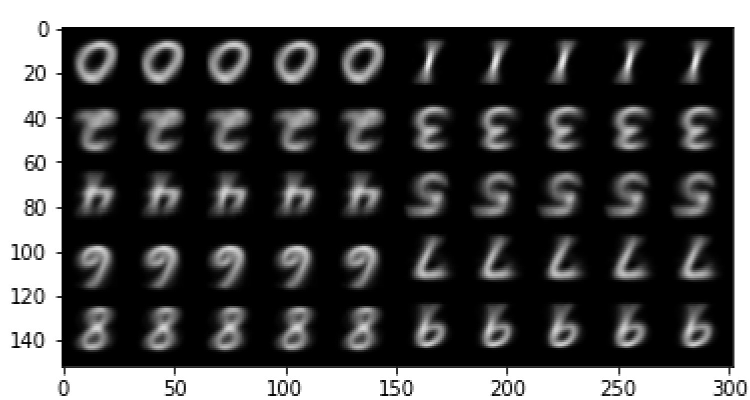
[17]:
```
init_ds
```

[17]:
```
<torch.utils.data.dataset.Subset at 0x7fd9c9c5b8e0>
```

[18]:
```
model = pt.models.ImageGLVQ(
    dict(distribution=(10, 5)),
    prototype_initializer=pt.components.SMI(init_ds),
)
```

[19]:
```
plt.imshow(model.get_prototype_grid(num_columns=10))
```

[19]:
```
<matplotlib.image.AxesImage at 0x7fd9c8173a00>
```



1.4 FAQs

1.4.1 How do I Retrieve the prototypes and their respective labels from the model?

For prototype models, the prototypes can be retrieved (as `torch.tensor`) as `model.prototypes`. You can convert it to a NumPy Array by calling `.numpy()` on the tensor if required.

```
>>> model.prototypes.numpy()
```

Similarly, the labels of the prototypes can be retrieved via `model.prototype_labels`.

```
>>> model.prototype_labels
```

1.4.2 How do I make inferences/predictions/recall with my trained model?

The models under prototorch.models provide a `.predict(x)` method for making predictions. This returns the predicted class labels. It is essential that the input to this method is a `torch.tensor` and not a NumPy array. Model instances are also callable. So, you could also just say `model(x)` as if `model` were just a function. However, this returns a (pseudo)-probability distribution over the classes.

Example

```
>>> y_pred = model.predict(torch.Tensor(x_train))  # returns class labels
```

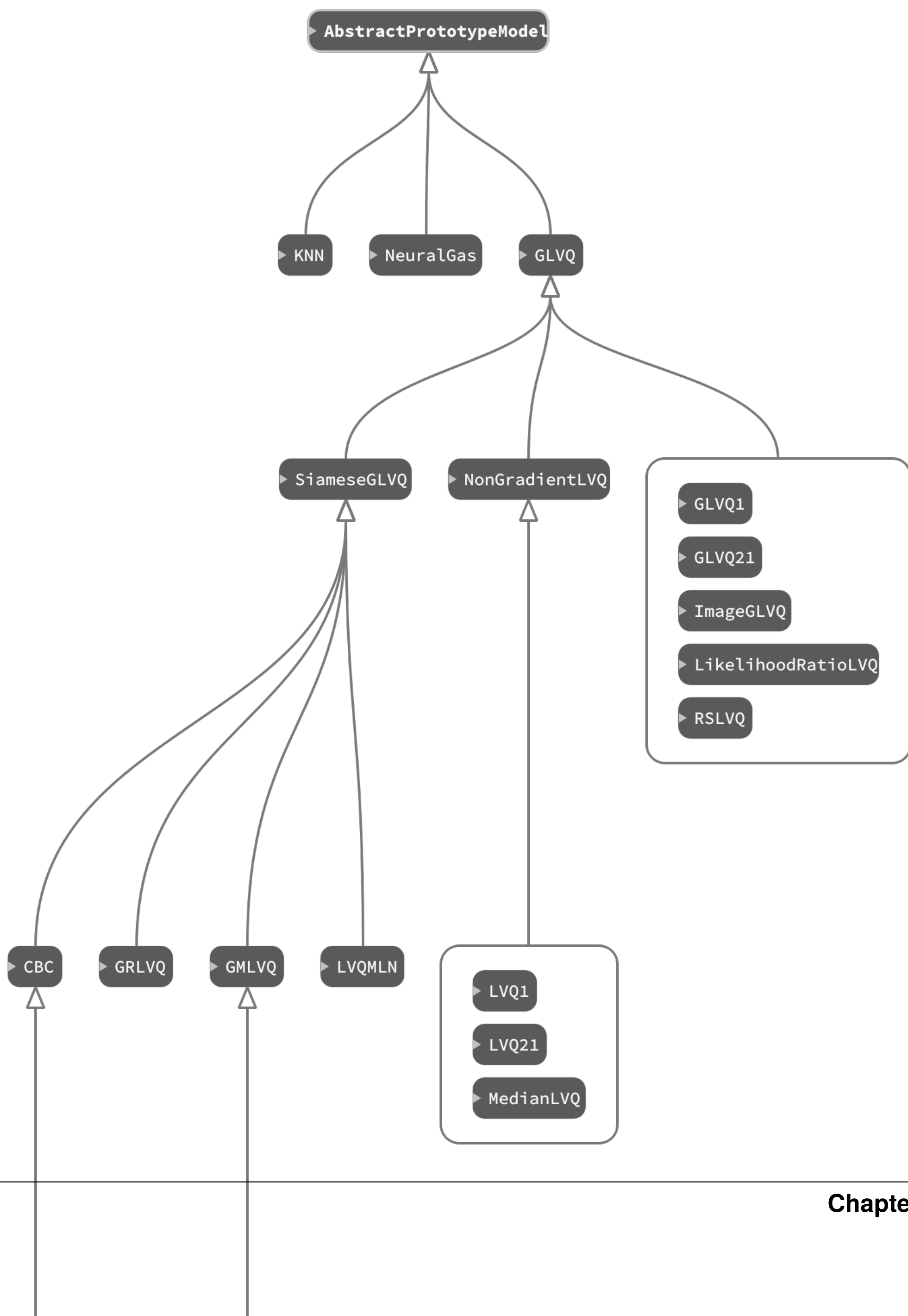
or, simply

```
>>> y_pred = model(torch.Tensor(x_train))  # returns probabilities
```

CHAPTER

TWO

MODELS



2.1 Unsupervised Methods

2.2 Classical Learning Vector Quantization

Original LVQ models introduced by Kohonen [1989]. These heuristic algorithms do not use gradient descent.

It is also possible to use the GLVQ structure as shown by Sato and Yamada [1996] in chapter 4. This allows the use of gradient descent methods.

2.3 Generalized Learning Vector Quantization

Sato and Yamada [1996] presented a LVQ variant with a cost function called GLVQ. This allows the use of gradient descent methods.

The cost function of GLVQ can be extended by a learnable dissimilarity. These learnable dissimilarities assign relevances to each data dimension during the learning phase. For example GRLVQ [Hammer and Villmann, 2002] and GMLVQ [Schneider *et al.*, 2009] .

The dissimilarity from GMLVQ can be interpreted as a projection into another dataspace. Applying this projection only to the data results in LVQMLN

The projection idea from GMLVQ can be extended to an arbitrary transformation with learnable parameters.

2.4 Probabilistic Models

Probabilistic variants assume, that the prototypes generate a probability distribution over the classes. For a test sample they return a distribution instead of a class assignment.

The following two algorihms were presented by Seo and Obermayer [2003] . Every prototypes is a center of a gaussian distribution of its class, generating a mixture model.

Villmann *et al.* [2018] proposed two changes to RSLVQ: First incooperate the winning rank into the prior probability calculation. And second use divergence as loss function.

2.5 Classification by Component

The Classification by Component (CBC) has been introduced by Saralajew *et al.* [2019] . In a CBC architecture there is no class assigned to the prototypes. Instead the dissimilarities are used in a reasoning process, that favours or rejects a class by a learnable degree. The output of a CBC network is a probability distribution over all classes.

CHAPTER THREE

VISUALIZATION

Visualization is very specific to its application. PrototorchModels delivers visualization for two dimensional data and image data.

The visulizations can be shown in a seperate window and inside a tensorboard.

CHAPTER

FOUR

BIBLIOGRAPHY

CHAPTER FIVE

ABSTRACT MODELS

CHAPTER

SIX

ABOUT

Prototorch Models is a Plugin for Prototorch. It implements common prototype-based Machine Learning algorithms using PyTorch-Lightning.

CHAPTER

SEVEN

LIBRARY

Prototorch Models delivers many application ready models. These models have been published in the past and have been adapted to the Prototorch library.

CHAPTER

EIGHT

CUSTOMIZABLE

Prototorch Models also contains the building blocks to build own models with PyTorch-Lightning and Prototorch.

BIBLIOGRAPHY

[HV02] Barbara Hammer and Thomas Villmann. Generalized relevance learning vector quantization. *Neural Networks*, 15(8):1059–1068, 2002. doi:https://doi.org/10.1016/S0893-6080(02)00079-5.

[Koh89] Teuvo Kohonen. *Self-Organization and Associative Memory*. Springer Berlin Heidelberg, 1989. doi:10.1007/978-3-642-88163-3.

[SHR+19] Sascha Saralajew, Lars Holdijk, Maike Rees, Ebubekir Asan, and Thomas Villmann. Classification-by-components: probabilistic modeling of reasoning over a set of components. In *Advances in Neural Information Processing Systems*, volume 32. 2019. URL: https://proceedings.neurips.cc/paper/2019/file/dca5672ff3444c7e997aa9a2c4eb2094-Paper.pdf.

[SY96] Atsushi Sato and Keiji Yamada. Generalized learning vector quantization. *Advances in neural information processing systems*, pages 423–429, 1996. URL: http://papers.nips.cc/paper/1113-generalized-learning-vector-quantization.pdf.

[SBH09] Petra Schneider, Michael Biehl, and Barbara Hammer. Adaptive Relevance Matrices in Learning Vector Quantization. *Neural Computation*, 21(12):3532–3561, 12 2009. doi:10.1162/neco.2009.11-08-908.

[SO03] Sambu Seo and Klaus Obermayer. Soft Learning Vector Quantization. *Neural Computation*, 15(7):1589–1604, 07 2003. doi:10.1162/089976603321891819.

[VKSV18] Andrea Villmann, Marika Kaden, Sascha Saralajew, and Thomas Villmann. Probabilistic learning vector quantization with cross-entropy for probabilistic class assignments in classification learning. In *Artificial Intelligence and Soft Computing*. Springer International Publishing, 2018.