Concept Bottleneck Models: Dot Training Example

This very short notebook will showcase how to set up a Concept Embedding Model (CEM) using our library and train it on the Dot dataset proposed in our CEM NeurIPS 2022 paper.

Our example is composed by four different steps:

1. Loading the dataset of interest in a format that can be "digested" by our models.
2. Instantiating a CEM with the embedding size and encoder/decoder architectures we want to use.
3. Training the CEM on the Dot dataset.
4. Evaluating the CEM's task accuracy, concept AUC, and concept alignment score (CAS).

Step 1: Load Data

As a first step, we will show you how one can generate a dataset from scratch that is compatible with how our training pipeline is set.

In practice, you can train any CEM (or CBM variant) using our library as long as your dataset is structured such that:

1. It is contained within a Pytorch DataLoader object.
2. Every sample contains is a tuple with three elements in it: the sample $x \in R^n$, the task label $y \in \{0, \cdots, L-1\}$, and a vector of k binary concept annotations $c \in \{0,1\}^k$ (in that order).

Below, we show how we do this for the Dot dataset. For details on the actual dataset, please refer to our paper.

```
import numpy as np
import torch
from pytorch_lightning import seed_everything

# We first create a simple helper function to sample random labeled
instances
# from the Dot dataset:
def generate_dot_data(size):
    # sample from normal distribution
    emb_size = 2
    # Generate the latent vectors
    v1 = np.random.randn(size, emb_size) * 2
    v2 = np.ones(emb_size)
    v3 = np.random.randn(size, emb_size) * 2
    v4 = -np.ones(emb_size)
    # Generate the sample
```

```
    x = np.hstack([v1+v3, v1-v3])

    # Now the concept vector
    c = np.stack([
        np.dot(v1, v2).ravel() > 0,
        np.dot(v3, v4).ravel() > 0,
    ]).T
    # And finally the label
    y = ((v1*v3).sum(axis=-1) > 0).astype(np.int64)

    # We NEED to put all of these into torch Tensors (THIS IS VERY
IMPORTANT)
    x = torch.FloatTensor(x)
    c = torch.FloatTensor(c)
    y = torch.Tensor(y)
    return x, y, c


# We then use our helper function to generate DataLoaders with the
correct
# number of samples in them. We use a separate function for this to
avoid
# repeating code to generate the different folds of our dataset:
def data_generator(
    dataset_size,
    batch_size,
    seed=None,
):
    # For the sake of determinism, let's always first seed everything
    # so that things can be recreated
    seed_everything(seed)
    x, y, c = generate_dot_data(dataset_size)
    data = torch.utils.data.TensorDataset(x, y, c)
    dl = torch.utils.data.DataLoader(
        data,
        batch_size=batch_size,
    )
    return dl

# Finally, we generate our training, testing, and validation folds
with
# different random seeds
train_dl = data_generator(
    dataset_size=int(3000 * 0.7),
    batch_size=256,
    seed=42,
)
test_dl = data_generator(
    dataset_size=int(3000 * 0.2),
    batch_size=256,
```

```
    seed=43,
)
val_dl = data_generator(
    dataset_size=int(3000 * 0.1),
    batch_size=256,
    seed=44,
)
```

Step 2: Create CEM Model

Now that we have our dataset in the correct `DataLoader` format, we can proceed to construct our CEM object. For this, we will simply import our `ConceptEmbeddingModel` object from the `cem` library. We can then instantiate a CEM by indicating:

1. The number of concepts `n_concepts` in the dataset we will train it on (e.g., 2 for the Dot dataset).
2. The number of output tasks/labels `n_tasks` in the dataset of interest (e.g., 1 for the binary task in the Dot dataset).
3. The size `emb_size` of each concept embedding.
4. The weight `concept_loss_weight` to use for the concept prediction loss during training of the CEM (e.g., in our paper we set this value to 1 for the Dot dataset).
5. The `learning_rate` and `optimizer` to use during training (e.g., "adam" or "sgd").
6. The probability `training_intervention_prob` to perform a random intervention at training time via RandInt (we recommend setting this to 0.25).
7. The model architecture `c_extractor_arch` to use for the latent code generator (i.e., the model that generates a latent representation to learn embeddings from the input samples).
8. The model `c2y_model` to use as a label predictor **after** all concept embeddings have been generated by a CEM.

The only non-trivial parameters to set for this instantiation are the model architectures for the latent code generator (passed via the `c_extractor_arch` argument) and for the label predictor (passed via) the `c2y_model` argument.

The first of these arguments, namely the latent code generator `c_extractor_arch`, must be provided as a simple Python function that takes as an input a named argument `output_dim` and generates a model that maps inputs from your task of interest to a latent code with shape `output_dim`. For our Dot example, we will do this via a simple MLP (although in practice you can do use an arbitrarily complex model):

```
def latent_code_generator_model(output_dim):
    if output_dim is None:
        output_dim = 128
    return torch.nn.Sequential(*[
        # 4 because Dot has inputs with 4 features in them
        torch.nn.Linear(4, 128),
        torch.nn.LeakyReLU(),
```

```
        torch.nn.Linear(128, 128),
        torch.nn.LeakyReLU(),
        torch.nn.Linear(128, output_dim),
    ])
```

The second of these arguments, namely the label predictor `c2y_model`, must be any valid Pytorch model that takes as an input as many activations as the CEM's bottleneck (i,e., `n_concepts` * `emb_size`) and generates `n_tasks` outputs, one for each output label in our dataset's downstream task. If not provided, or if set to `None`, then by default we will simply attach a linear mapping after the CEM's bottleneck to obtain the output label prediction. In practice, this is how a CEM is usually constructed.

```
# We simply import our CEM class (the same can be done with CBMs to
easily train
# any of their variants)
from cem.models.cem import ConceptEmbeddingModel

# And generate the actual model
cem_model = ConceptEmbeddingModel(
  n_concepts=2, # Number of training-time concepts. Dot has 2
  n_tasks=1, # Number of output labels. Dot is binary so it has 1.
  emb_size=128,  # We will use an embedding size of 128
  concept_loss_weight=1,  # The weight assigned to the concept
prediction loss relative to the task predictive loss.
  learning_rate=1e-3,  # The learning rate to use during training.
  optimizer="adam",  # The optimizer to use during training.
  training_intervention_prob=0.25, # RandInt probability. We recommend
setting this to 0.25.
  c_extractor_arch=latent_code_generator_model, # Here we provide our
generating function for the latent code generator model.
  c2y_model=None,  # We will let the API simply add a linear layer
from the concept bottleneck to the downstream task labels
)
print(cem_model)
```

Step 3: Train the CEM

Now that we have both the dataset and the model defined, we can train our CEM using Pytorch Lightning's wrappers for ease. This should be very simple via Pytorch Lightning's `Trainer` once the data has been generated:

```
import pytorch_lightning as pl

trainer = pl.Trainer(
    accelerator="gpu",  # Change to "cpu" if you are not running on a
GPU!
    devices="auto",
    max_epochs=500,  # The number of epochs we will train our model
```

```
for
    check_val_every_n_epoch=5,  # And how often we will check for
validation metrics
    logger=False,  # No logs to be dumped for this trainer
)

# train_dl and val_dl are datasets previously built...
trainer.fit(cem_model, train_dl, val_dl)
```

For more details on all the things you may add/configure to the Trainer for more control, please refer to the official documentation.

Part 4: Evaluate Model

Once the CEM has been trained, you can evaluate it with test data to generate the learnt embeddings, the predicted concepts, and the predicted task labels!

A CEM or CBM model can be called with any input sample of shape `(batch_size, ...)` using Pytorch's functional API:

```
(c_pred, c_embs, y_pred) = cem_model(x)
```

Where:

1. `c_pred` is a $(\text{batch\_size}, k)$-dimensional vector where the i-th dimension indicates the probability that the i-th concept is on.
2. `c_embs` is a $(\text{batch\_size}, k \cdot \text{emb\_size})$-dimensional tensor representing the CEM's bottleneck. This corresponds to all concept embeddings concatenated in the same order as given in the training annotations (so reshaping it to $(\text{batch\_size}, k, \text{emb\_size})$ will allow you to access each concept's embedding directly).
3. `y_pred` is a $(\text{batch\_size}, L)$-dimensional vector where the i-th dimension is proportional to the probability that the i-th label is predicted for the current sample (the model outputs logits by default). If the downstream task is binary, then the CEM will output a (batch_size)-dimensional vector where each entry is the logit of the probability of the downstream class being 1.

This allows us to compute some metrics of interest. Below, we will use PytorchLightning's API to be able to run inference in batches in a GPU to obtain all test activations.

Before doing this, we will turn our test dataset into numpy arrays as they will be easily easier to work with if we want to compute custom metrics:

```
# Before anything, however, let's get the underlying numpy arrays of
our
# test dataset as they will be easier to work with
x_test, y_test, c_test = [], [], []
for (x, y, c) in test_dl:
    x_test.append(x)
```

```
    y_test.append(y)
    c_test.append(c)
x_test = np.concatenate(x_test, axis=0)
y_test = np.concatenate(y_test, axis=0)
c_test = np.concatenate(c_test, axis=0)
```

Now we are ready to generate the concept, label, and embedding predictions for the test set using our trained CEM:

```
# We will use a Trainer object to run inference in batches over our
test
# dataset
trainer = pl.Trainer(
    accelerator="gpu",
    devices="auto",
    logger=False, # No logs to be dumped for this trainer
)
batch_results = trainer.predict(cem_model, test_dl)

# Then we combine all results into numpy arrays by joining over the
batch
# dimension
c_pred = np.concatenate(
    list(map(lambda x: x[0].detach().cpu().numpy(), batch_results)),
    axis=0,
)
c_embs = np.concatenate(
    list(map(lambda x: x[1].detach().cpu().numpy(), batch_results)),
    axis=0,
)
# Reshape them so that we have embeddings (batch_size, k, emb_size)
c_embs = np.reshape(c_embs, (c_test.shape[0], c_test.shape[1], -1))

y_pred = np.concatenate(
    list(map(lambda x: x[2].detach().cpu().numpy(), batch_results)),
    axis=0,
)
```

And compute all the metrics of interest:

```
##########
## Compute test task accuracy
##########

from scipy.special import expit
from sklearn.metrics import accuracy_score

# Which allows us to compute the task accuracy (we explicitly perform
a
```

```python
# sigmoidal operation as CEMs always return logits)
task_accuracy = accuracy_score(y_test, expit(y_pred) >=0.5)
print(f"Our CEM's test task accuracy is {task_accuracy*100:.2f}%")

##########
## Compute test concept AUC
##########

from scipy.special import expit
from sklearn.metrics import roc_auc_score

# Which allows us to compute the task accuracy (we explicitly perform
a
# sigmoidal operation as CEMs always return logits)
concept_auc = roc_auc_score(c_test, c_pred)
print(f"Our CEM's test concept AUC is {concept_auc*100:.2f}%")

##########
## Compute test concept alignment score
##########

from cem.metrics.cas import concept_alignment_score

cas, _ = concept_alignment_score(
    c_vec=c_embs,
    c_test=c_test,
    y_test=y_test,
    step=5,
    progress_bar=False,
)
print(f"Our CEM's concept alignment score (CAS) is {cas*100:.2f}%")
```