1. Triplet Loss

1-1. Triplet Loss

Triplet loss for representations of time series. Optimized for training sets where all time series have the same length. Takes as input a tensor as the chosen batch to compute the loss, a PyTorch module as the encoder, a 3D tensor (`B`, `C`, `L`) containing the training set, where `B` is the batch size, `C` is the number of channels and `L` is the length of the time series, as well as a boolean which, if True, enables to save GPU memory by propagating gradients after each loss term, instead of doing it after computing the whole loss. The triplets are chosen in the following manner. First the size of the positive and negative samples are randomly chosen in the range of lengths of time series in the dataset. The size of the anchor time series is randomly chosen with the same length upper bound but the the length of the positive samples as lower bound. An anchor of this length is then chosen randomly in the given time series of the train set, and positive samples are randomly chosen among subseries of the anchor. Finally, negative samples of the chosen length are randomly chosen in random time series of the train set.
`@param compared_length` Maximum length of randomly chosen time series. If None, this parameter is ignored.
`@param nb_random_samples` Number of negative samples per batch example.
`@param negative_penalty` Multiplicative coefficient for the negative sample loss.

```
import torch
import numpy

class TripletLoss(torch.nn.modules.loss._Loss):
    def __init__(self, compared_length, nb_random_samples, negative_penalty):
        super(TripletLoss, self).__init__()
        self.compared_length = compared_length
        if self.compared_length is None:
            self.compared_length = numpy.inf
        self.nb_random_samples = nb_random_samples
        self.negative_penalty = negative_penalty

    def forward(self, batch, encoder, train, save_memory=False):
        batch_size = batch.size(0)
        train_size = train.size(0)
        length = min(self.compared_length, train.size(2))

        # For each batch element, we pick nb_random_samples possible random
        # time series in the training set (choice of batches from where the
        # negative examples will be sampled)
        samples = numpy.random.choice(
            train_size, size=(self.nb_random_samples, batch_size)
        )
        samples = torch.LongTensor(samples)

        # Choice of length of positive and negative samples
        length_pos_neg = numpy.random.randint(1, high=length + 1)

        # We choose for each batch example a random interval in the time
        # series, which is the 'anchor'
        random_length = numpy.random.randint(
```

```
            length_pos_neg, high=length + 1
        )  # Length of anchors
        beginning_batches = numpy.random.randint(
            0, high=length - random_length + 1, size=batch_size
        )  # Start of anchors

        # The positive samples are chosen at random in the chosen anchors
        beginning_samples_pos = numpy.random.randint(
            0, high=random_length - length_pos_neg + 1, size=batch_size
        )  # Start of positive samples in the anchors
        # Start of positive samples in the batch examples
        beginning_positive = beginning_batches + beginning_samples_pos
        # End of positive samples in the batch examples
        end_positive = beginning_positive + length_pos_neg

        # We randomly choose nb_random_samples potential negative samples for
        # each batch example
        beginning_samples_neg = numpy.random.randint(
            0, high=length - length_pos_neg + 1,
            size=(self.nb_random_samples, batch_size)
        )

        representation = encoder(torch.cat(
            [batch[
                j: j + 1, :,
                beginning_batches[j]: beginning_batches[j] + random_length
            ] for j in range(batch_size)]
        ))  # Anchors representations

        positive_representation = encoder(torch.cat(
            [batch[
                j: j + 1, :, end_positive[j] - length_pos_neg: end_positive[j]
            ] for j in range(batch_size)]
        ))  # Positive samples representations

        size_representation = representation.size(1)
        # Positive loss: -logsigmoid of dot product between anchor and positive
        # representations
        loss = -torch.mean(torch.nn.functional.logsigmoid(torch.bmm(
            representation.view(batch_size, 1, size_representation),
            positive_representation.view(batch_size, size_representation, 1)
        )))

        # If required, backward through the first computed term of the loss and
        # free from the graph everything related to the positive sample
        if save_memory:
            loss.backward(retain_graph=True)
            loss = 0
            del positive_representation
            torch.cuda.empty_cache()

        multiplicative_ratio = self.negative_penalty / self.nb_random_samples
        for i in range(self.nb_random_samples):
            # Negative loss: -logsigmoid of minus the dot product between
            # anchor and negative representations
            negative_representation = encoder(
                torch.cat([train[samples[i, j]: samples[i, j] + 1][
                    :, :,
```

```
                    beginning_samples_neg[i, j]:
                    beginning_samples_neg[i, j] + length_pos_neg
                ] for j in range(batch_size)])
            )
            loss += multiplicative_ratio * -torch.mean(
                torch.nn.functional.logsigmoid(-torch.bmm(
                    representation.view(batch_size, 1, size_representation),
                    negative_representation.view(
                        batch_size, size_representation, 1
                    )
                ))
            )
            # If required, backward through the first computed term of the loss
            # and free from the graph everything related to the negative sample
            # Leaves the last backward pass to the training procedure
            if save_memory and i != self.nb_random_samples - 1:
                loss.backward(retain_graph=True)
                loss = 0
                del negative_representation
                torch.cuda.empty_cache()

        return loss
```

1-2. Triplet Loss with Varying Length

```
class TripletLossVaryingLength(torch.nn.modules.loss._Loss):
    def __init__(self, compared_length, nb_random_samples, negative_penalty):
        super(TripletLossVaryingLength, self).__init__()
        self.compared_length = compared_length
        if self.compared_length is None:
            self.compared_length = numpy.inf
        self.nb_random_samples = nb_random_samples
        self.negative_penalty = negative_penalty

    def forward(self, batch, encoder, train, save_memory=False):
        batch_size = batch.size(0)
        train_size = train.size(0)
        max_length = train.size(2)

        # For each batch element, we pick nb_random_samples possible random
        # time series in the training set (choice of batches from where the
        # negative examples will be sampled)
        samples = numpy.random.choice(
            train_size, size=(self.nb_random_samples, batch_size)
        )
        samples = torch.LongTensor(samples)

        # Computation of the lengths of the relevant time series
        with torch.no_grad():
            lengths_batch = max_length - torch.sum(
                torch.isnan(batch[:, 0]), 1
            ).data.cpu().numpy()
            lengths_samples = numpy.empty(
                (self.nb_random_samples, batch_size), dtype=int
            )
            for i in range(self.nb_random_samples):
                lengths_samples[i] = max_length - torch.sum(
                    torch.isnan(train[samples[i], 0]), 1
                ).data.cpu().numpy()

        # Choice of lengths of positive and negative samples
        lengths_pos = numpy.empty(batch_size, dtype=int)
        lengths_neg = numpy.empty(
            (self.nb_random_samples, batch_size), dtype=int
        )
        for j in range(batch_size):
            lengths_pos[j] = numpy.random.randint(
                1, high=min(self.compared_length, lengths_batch[j]) + 1
            )
            for i in range(self.nb_random_samples):
                lengths_neg[i, j] = numpy.random.randint(
                    1,
                    high=min(self.compared_length, lengths_samples[i, j]) + 1
                )

        # We choose for each batch example a random interval in the time
        # series, which is the 'anchor'
        random_length = numpy.array([numpy.random.randint(
            lengths_pos[j],
```

```
            high=min(self.compared_length, lengths_batch[j]) + 1
        ) for j in range(batch_size)])  # Length of anchors
        beginning_batches = numpy.array([numpy.random.randint(
            0, high=lengths_batch[j] - random_length[j] + 1
        ) for j in range(batch_size)])  # Start of anchors

        # The positive samples are chosen at random in the chosen anchors
        # Start of positive samples in the anchors
        beginning_samples_pos = numpy.array([numpy.random.randint(
            0, high=random_length[j] - lengths_pos[j] + 1
        ) for j in range(batch_size)])
        # Start of positive samples in the batch examples
        beginning_positive = beginning_batches + beginning_samples_pos
        # End of positive samples in the batch examples
        end_positive = beginning_positive + lengths_pos

        # We randomly choose nb_random_samples potential negative samples for
        # each batch example
        beginning_samples_neg = numpy.array([[numpy.random.randint(
            0, high=lengths_samples[i, j] - lengths_neg[i, j] + 1
        ) for j in range(batch_size)] for i in range(self.nb_random_samples)])

        representation = torch.cat([encoder(
            batch[
                j: j + 1, :,
                beginning_batches[j]: beginning_batches[j] + random_length[j]
            ]
        ) for j in range(batch_size)])  # Anchors representations

        positive_representation = torch.cat([encoder(
            batch[
                j: j + 1, :,
                end_positive[j] - lengths_pos[j]: end_positive[j]
            ]
        ) for j in range(batch_size)])  # Positive samples representations

        size_representation = representation.size(1)
        # Positive loss: -logsigmoid of dot product between anchor and positive
        # representations
        loss = -torch.mean(torch.nn.functional.logsigmoid(torch.bmm(
            representation.view(batch_size, 1, size_representation),
            positive_representation.view(batch_size, size_representation, 1)
        )))

        # If required, backward through the first computed term of the loss and
        # free from the graph everything related to the positive sample
        if save_memory:
            loss.backward(retain_graph=True)
            loss = 0
            del positive_representation
            torch.cuda.empty_cache()

        multiplicative_ratio = self.negative_penalty / self.nb_random_samples
        for i in range(self.nb_random_samples):
            # Negative loss: -logsigmoid of minus the dot product between
            # anchor and negative representations
            negative_representation = torch.cat([encoder(
                train[samples[i, j]: samples[i, j] + 1][
```

```
                    :, :,
                    beginning_samples_neg[i, j]:
                    beginning_samples_neg[i, j] + lengths_neg[i, j]
                ]
            ) for j in range(batch_size)])
            loss += multiplicative_ratio * -torch.mean(
                torch.nn.functional.logsigmoid(-torch.bmm(
                    representation.view(batch_size, 1, size_representation),
                    negative_representation.view(
                        batch_size, size_representation, 1
                    )
                ))
            )
            # If required, backward through the first computed term of the loss
            # and free from the graph everything related to the negative sample
            # Leaves the last backward pass to the training procedure
            if save_memory and i != self.nb_random_samples - 1:
                loss.backward(retain_graph=True)
                loss = 0
                del negative_representation
                torch.cuda.empty_cache()

        return loss
```

2. [Encoder 1] Causal CNN

2-1. Chomp1d

Removes the last elements of a time series. Takes as input a three-dimensional tensor (`B`, `C`, `L`) where `B` is the batch size, `C` is the number of input channels, and `L` is the length of the input. Outputs a three-dimensional tensor (`B`, `C`, `L - s`) where `s` is the number of elements to remove.
`@param chomp_size` Number of elements to remove.

```
class Chomp1d(torch.nn.Module):
    def __init__(self, chomp_size):
        super(Chomp1d, self).__init__()
        self.chomp_size = chomp_size

    def forward(self, x):
        return x[:, :, :-self.chomp_size]
```

2-2. Squeeze Channels

Squeezes, in a three-dimensional tensor, the third dimension.

```
class SqueezeChannels(torch.nn.Module):
    def __init__(self):
        super(SqueezeChannels, self).__init__()

    def forward(self, x):
        return x.squeeze(2)
```

2-3. Causal Convolution Block

Causal convolution block, composed sequentially of two causal convolutions (with leaky ReLU activation functions), and a parallel residual connection.
Takes as input a three-dimensional tensor (`B`, `C`, `L`) where `B` is the batch size, `C` is the number of input channels, and `L` is the length of the input. Outputs a three-dimensional tensor (`B`, `C`, `L`).
`@param in_channels` Number of input channels.
`@param out_channels` Number of output channels.
`@param kernel_size` Kernel size of the applied non-residual convolutions.
`@param dilation` Dilation parameter of non-residual convolutions.
`@param final` Disables, if True, the last activation function.

```
class CausalConvolutionBlock(torch.nn.Module):

    def __init__(self, in_channels, out_channels, kernel_size, dilation,
                 final=False):
        super(CausalConvolutionBlock, self).__init__()

        # Computes left padding so that the applied convolutions are causal
        padding = (kernel_size - 1) * dilation

        # First causal convolution
        conv1 = torch.nn.utils.weight_norm(torch.nn.Conv1d(
            in_channels, out_channels, kernel_size,
            padding=padding, dilation=dilation
        ))
        # The truncation makes the convolution causal
        chomp1 = Chomp1d(padding)
        relu1 = torch.nn.LeakyReLU()

        # Second causal convolution
        conv2 = torch.nn.utils.weight_norm(torch.nn.Conv1d(
            out_channels, out_channels, kernel_size,
            padding=padding, dilation=dilation
        ))
        chomp2 = Chomp1d(padding)
        relu2 = torch.nn.LeakyReLU()

        # Causal network
        self.causal = torch.nn.Sequential(
            conv1, chomp1, relu1, conv2, chomp2, relu2
        )

        # Residual connection
        self.upordownsample = torch.nn.Conv1d(
            in_channels, out_channels, 1
        ) if in_channels != out_channels else None

        # Final activation function
        self.relu = torch.nn.LeakyReLU() if final else None

    def forward(self, x):
        out_causal = self.causal(x)
        res = x if self.upordownsample is None else self.upordownsample(x)
```

```
        if self.relu is None:
            return out_causal + res
        else:
            return self.relu(out_causal + res)
```

2-4. Causal CNN Encoder

Encoder of a time series using a causal CNN: the computed representation is the output of a fully connected layer applied to the output of an adaptive
max pooling layer applied on top of the causal CNN, which reduces the length of the time series to a fixed size. Takes as input a three-dimensional tensor (`B`, `C`, `L`) where `B` is the batch size, `C` is the number of input channels, and `L` is the length of the input. Outputs a three-dimensional tensor (`B`, `C`).
`@param in_channels` Number of input channels.
`@param channels` Number of channels manipulated in the causal CNN.
`@param depth` Depth of the causal CNN.
`@param reduced_size` Fixed length to which the output time series of the causal CNN is reduced.
`@param out_channels` Number of output channels.
`@param kernel_size` Kernel size of the applied non-residual convolutions.

```
class CausalCNNEncoder(torch.nn.Module):
    def __init__(self, in_channels, channels, depth, reduced_size,
                 out_channels, kernel_size):
        super(CausalCNNEncoder, self).__init__()
        causal_cnn = CausalCNN(
            in_channels, channels, depth, reduced_size, kernel_size
        )
        reduce_size = torch.nn.AdaptiveMaxPool1d(1)
        squeeze = SqueezeChannels()  # Squeezes the third dimension (time)
        linear = torch.nn.Linear(reduced_size, out_channels)
        self.network = torch.nn.Sequential(
            causal_cnn, reduce_size, squeeze, linear
        )

    def forward(self, x):
        return self.network(x)
```

3. [Encoder 2] LSTM Encoder

Encoder of a time series using a LSTM, ccomputing a linear transformation of the output of an LSTM. Takes as input a three-dimensional tensor (`B`, `C`, `L`) where `B` is the batch size, `C` is the number of input channels, and `L` is the length of the input. Outputs a three-dimensional tensor (`B`, `C`). Only works for one-dimensional time series.

```
class LSTMEncoder(torch.nn.Module):
    def __init__(self):
        super(LSTMEncoder, self).__init__()
        self.lstm = torch.nn.LSTM(
            input_size=1, hidden_size=256, num_layers=2
        )
        self.linear = torch.nn.Linear(256, 160)

    def forward(self, x):
        return self.linear(self.lstm(x.permute(2, 0, 1))[0][-1])
```