NT-Xent (Normalized Temperature-Scaled Cross-Entropy) Loss Explained and Implemented in PyTorch

An intuitive explanation of the NT-Xent loss with a step-by-step explanation of the operation and our implementation in PyTorch

Dhruv Matani · Follow
Published in Towards Data Science · 14 min read · Jun 13, 2023

133

Co-authored with Naresh Singh.

$$\mathbb{l}_{i,j} = -\log \frac{\exp(\mathrm{sim}(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum_{k=1}^{2N} 1_{[k \neq i]} \exp(\mathrm{sim}(\mathbf{z}_i, \mathbf{z}_k)/\tau)}$$

Formula for NT-Xent loss. Source: Papers with code (CC-BY-SA)

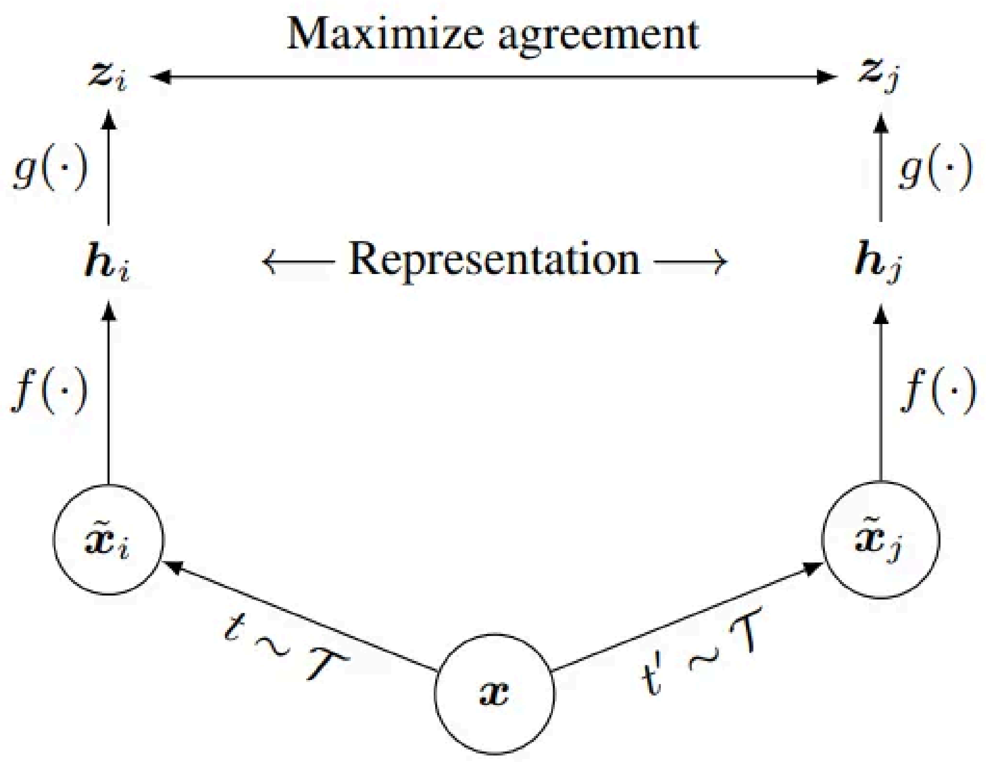
. . .

Introduction

Recent advances in self-supervised learning and contrastive learning have excited researchers and practitioners in Machine Learning (ML) to explore this space with renewed interest.

In particular, the SimCLR paper that presents a simple framework for contrastive learning of visual representations has gained a lot of attention in the self-supervised and contrastive learning space.

The central idea behind the paper is very simple — allow the model to learn if a pair of images were derived from the same or different initial image.



A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize agreement using a contrastive loss. After training is completed, we throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and representation $\boldsymbol{h}$ for downstream tasks.

Figure 1: The high-level idea behind SimCLR. Source: SimCLR paper

The SimCLR approach encodes each input image ***i*** as a feature vector ***zi***. There are 2 cases to consider:

1. **Positive Pairs:** The same image is augmented using a different set of augmentations, and the resulting feature vectors ***zi*** and ***zj*** are compared. These feature vectors are forced to be similar by the loss function.

2. **Negative Pairs:** Different images are augmented using a different set of augmentations, and the resulting feature vectors ***zi*** and ***zk*** are compared. These feature vectors are forced to be dissimilar by the loss function.

The rest of this article will focus on explaining and understanding this loss function, and its efficient implementation using PyTorch.

. . .

The NT-Xent Loss

At a high level, the contrastive learning model is fed 2N images, originating from N underlying images. Each of the N underlying images is augmented using a random set of image augmentations to produce 2 augmented images. This is how we end up with 2N images in a single train batch fed to the model.

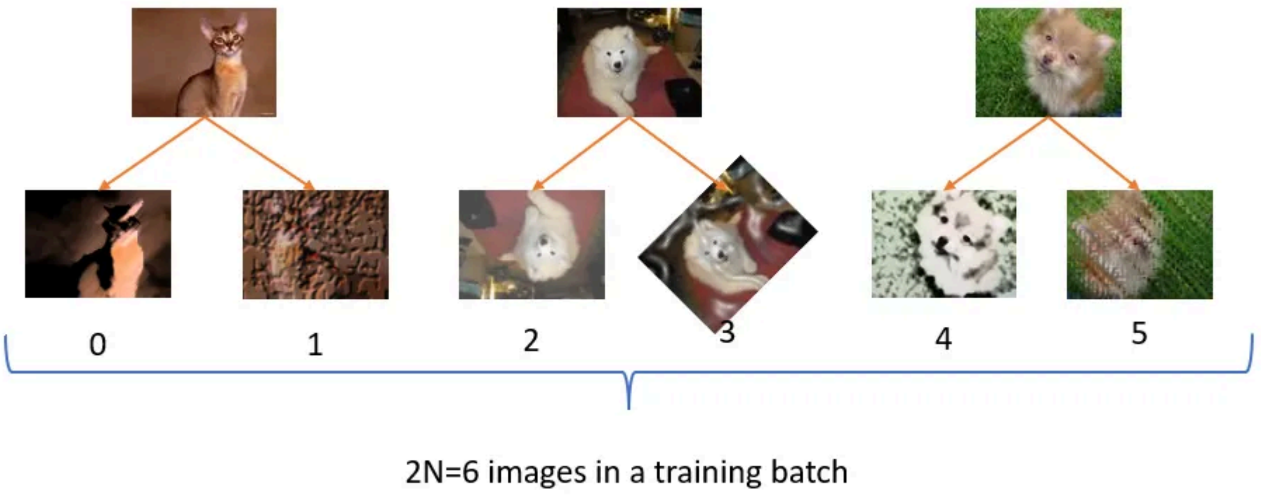


Figure 2: A batch of 6 images in a single training batch for contrastive learning. The number below each image is the index of that image in the input batch when fed into a contrastive learning model. Image Source: Oxford Visual Geometry Group (CC-SA).

In the following sections, we will dive deep into the following aspects of the NT-Xent loss.

1. The effect of temperature on SoftMax and Sigmoid
2. A simple and intuitive interpretation of the NT-Xent loss
3. A step-by-step implementation of NT-Xent in PyTorch
4. Motivating the need for a multi-label loss function (NT-BXent)
5. A step-by-step implementation of NT-BXent in PyTorch

All the code for steps 2–5 can be found in this notebook. The code for step-1 can be found in this notebook.

To understand all the moving parts of the contrastive loss function we'll be studying in this article, we need to first understand the effect of temperature on the SoftMax and Sigmoid activation functions.

Typically, temperature scaling is applied to the input to SoftMax or Sigmoid to either smooth out or accentuate the output of those activation functions. The input logits are divided by the temperature before passing into the activation functions. You can find all the code for this section in this notebook.

SoftMax: For SoftMax, a high temperature reduces the variance in the output distribution which results in softening of the labels. A low temperature increases the variance in the output distribution and makes the maximum value stand out over the other values. See the charts below for the effect of temperature on SoftMax when fed with the input tensor [0.1081, 0.4376, 0.7697, 0.1929, 0.3626, 2.8451].

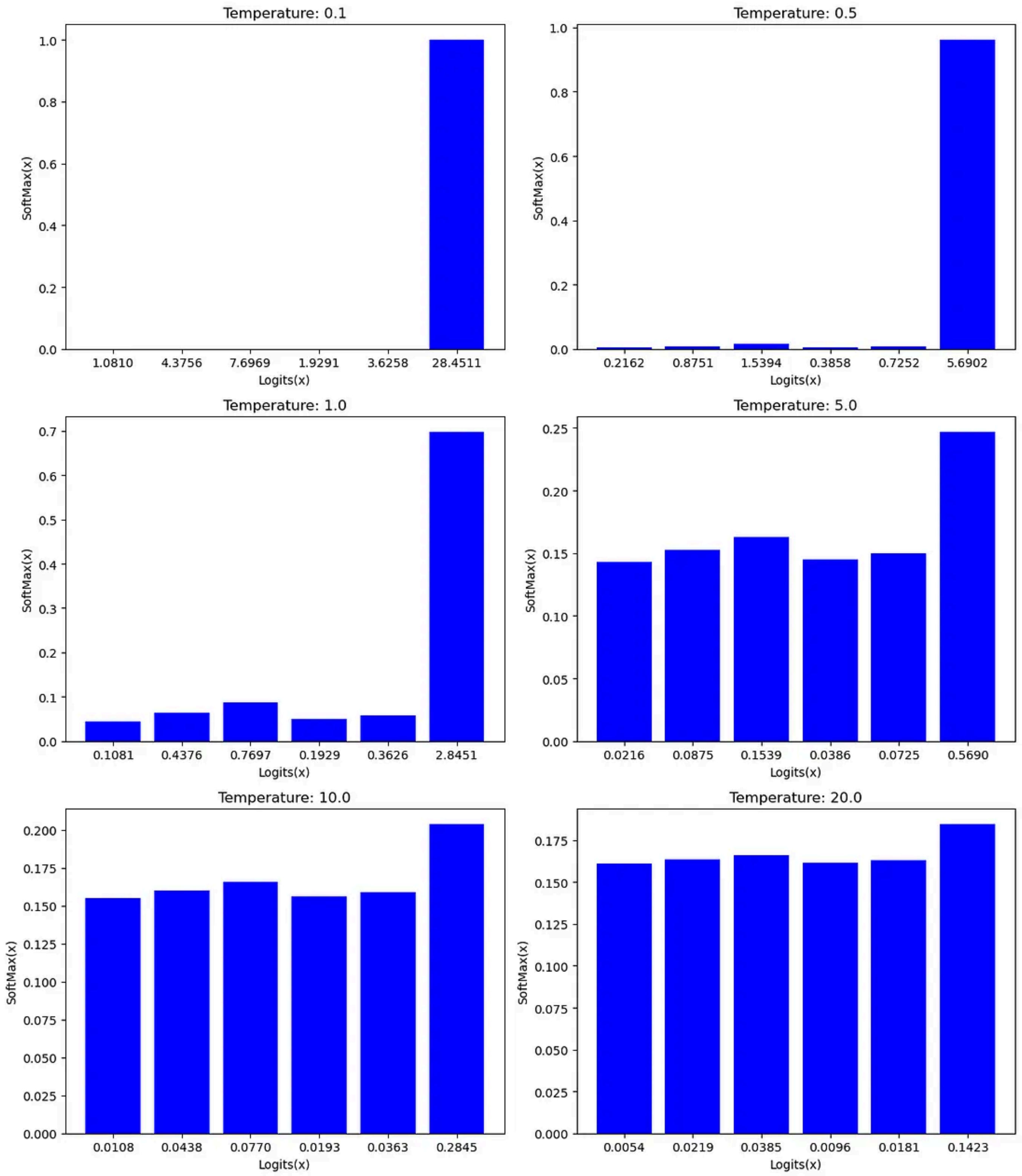


Figure 3: Effect of temperature on SoftMax. Source: Author(s)

Sigmoid: For Sigmoid, a high-temperature results in an output distribution that is pulled towards 0.0, whereas a low temperature stretches the inputs to higher values, stretching the outputs to be closer to either 0.0 or 1.0 depending on the unsigned magnitude of the input.

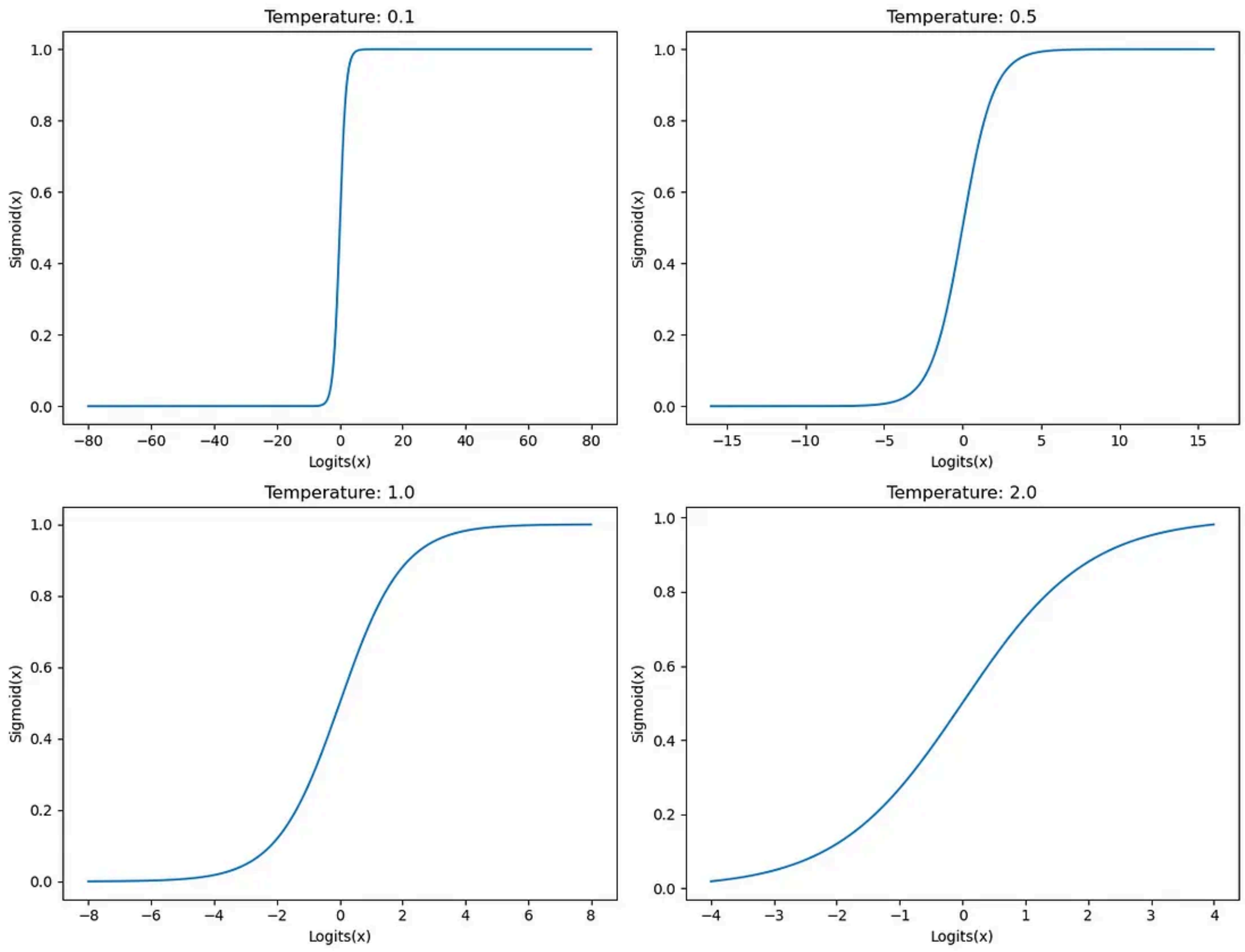


Figure 4: Effect of temperature on Sigmoid. Source: Author(s)

Now that we understand the effect of various temperature values on the SoftMax and Sigmoid functions, let’s see how this applies to our understanding of the NT-Xent loss.

. . .

Interpreting the NT-Xent loss

The NT-Xent loss is understood by understanding the individual terms in the name of this loss.

1. Normalized: Cosine similarity produces a normalized score in the range [-1.0 to +1.0]
2. Temperature-scaled: The all-pairs cosine similarity is scaled by a temperature before computing the cross-entropy loss
3. Cross-entropy loss: The underlying loss is a multi-class (single-label) cross-entropy loss

As mentioned above, we assume that for a batch of size 2N, the feature vectors at the following indices represent positive pairs (0, 1), (2, 3), (4, 5), (6, 7), … and the rest of the combinations represent negative pairs. This is an important factor to keep in mind throughout the interpretation of the NT-Xent loss as it relates to SimCLR.

Now that we understand what the terms mean in the context of the NT-Xent loss, let’s take a look at the mechanical steps needed to compute the NT-Xent loss on a batch of feature vectors.

1. The all-pairs Cosine Similarity score is computed for each of the 2N vectors produced by the SimCLR model. This results in $(2N)^2$ similarity scores represented as a 2N x 2N matrix
2. Comparison results between the same value (i, i) are discarded (since a distribution is perfectly similar to itself and can’t possibly allow the model to learn anything useful)

3. Each value (cosine similarity) is scaled by a temperature parameter τ (which is a hyper-parameter)
4. Cross-entropy loss is applied to each row of the resulting matrix above. The following paragraph explains more in detail
5. Typically, the mean of these losses (one loss per element in a batch) is used for backpropagation

The way that the cross-entropy loss is used here is semantically slightly different from how it's used in standard classification tasks. In classification tasks, a final "classification head" is trained to produce a one-hot-probability vector for each input, and we compute the cross-entropy loss on that one-hot-probability vector since we're effectively computing the difference between 2 distributions. This video explains the concept of cross-entropy loss beautifully. In the NT-Xent loss, there isn't a 1:1 correspondence between a trainable layer and the output distribution. Instead, a feature vector is computed for each input, and we then compute the cosine similarity between every pair of feature vectors. The trick here is that since each image is similar to exactly 1 other image in the input batch (positive pair) (if we ignore the similarity of a feature vector with itself), we can consider this to be a classification-like setting where the probability distribution of the similarity probability between images represents a classification task where one of them will be close to 1.0 and the rest will be close to 0.0.

Now that we have a solid overall understanding of the NT-Xent loss, we should be in great shape to implement these ideas in PyTorch. Let's get going!

. . .

Implementation of NT-Xent loss in PyTorch

All the code in this section can be found in this notebook.

Code Reuse: Many implementations of the NT-Xent loss seen online implement all the operations from scratch. Furthermore, some of them implement the loss function inefficiently, preferring to use for loops instead of GPU parallelism. Instead, we will use a different approach. We'll implement this loss in terms of the standard cross-entropy loss that PyTorch already provides. To do this, we need to massage the predictions and ground-truth labels in a format that cross_entropy can accept. Let's see how to do this below.

Predictions Tensor: First, we need to create a PyTorch tensor that will represent the output from our contrastive learning model. Let's assume that our batch size is 8 (2N=8), and our feature vectors have 2 dimensions (2 values). We'll call our input variable *"x"*.

```
x = torch.randn(8, 2)
```

Cosine Similarity: Next, we'll compute the all-pairs cosine similarity between every feature vector in this batch and store the result in the variable named *"xcs"*. If the line below seems confusing, please read the details on this page. This is the "normalize" step.

```
xcs = F.cosine_similarity(x[None,:,:], x[:,None,:], dim=-1)
```

As mentioned above, we need to ignore the self-similarity score of every feature vector since it doesn't contribute to the model's learning and will be an unnecessary nuisance later on when we want to compute the cross-entropy loss. For this purpose, we'll define a variable *"eye"* which is a matrix with the elements on the principal diagonal having a value of 1.0 and the rest being 0.0. We can create such a matrix using the following command.

```
eye = torch.eye(8)
```

Now let's convert this into a boolean matrix so that we can index into the *"xcs"* variable using this mask matrix.

```
eye = eye.bool()
```

Let's clone the tensor *"xcs"* into a tensor named *"y"* so that we can reference the "xcs" tensor later.

```
y = xcs.clone()
```

Now, we will set the values along the principal diagonal of the all-pairs cosine similarity matrix to *-inf* so that when we compute the softmax on each row, this value will contribute nothing.

```
y[eye] = float("-inf")
```

The tensor *"y"* scaled by a temperature parameter will be one of the inputs (predictions) to the cross-entropy loss API in PyTorch. Next, we need to compute the ground-truth labels (target) that we need to feed to the cross-entropy loss API.

Ground Truth labels (Target tensor): For the example we are using (2N=8), this is what the ground-truth tensor should look like.

> *tensor([1, 0, 3, 2, 5, 4, 7, 6])*

That's because the following index pairs in the tensor *"y"* contain positive pairs.

> *(0, 1), (1, 0)*
>
> *(2, 3), (3, 2)*
>
> *(4, 5), (5, 4)*
>
> *(6, 7), (7, 6)*

To interpret the index pairs above, we look at a single example. The pair (4, 5) means that column 5 at row 4 is supposed to be set to 1.0 (positive pair), which is what the tensor above is also saying. Great!

To create the tensor above, we can use the following PyTorch code, which stores the ground-truth labels in the variable *“target”*.

```
target = torch.arange(8)
target[0::2] += 1
target[1::2] -= 1
```

cross-entropy Loss: We have all the ingredients we need to compute our loss! The only thing that remains to be done is to call the cross_entropy API in PyTorch.

```
loss = F.cross_entropy(y / temperature, target, reduction="mean")
```

The variable “loss” now contains the computed NT-Xent loss. Let’s wrap all the code in a single python function below.

```
def nt_xent_loss(x, temperature):
  assert len(x.size()) == 2

  # Cosine similarity
  xcs = F.cosine_similarity(x[None,:,:], x[:,None,:], dim=-1)
  xcs[torch.eye(x.size(0)).bool()] = float("-inf")

  # Ground truth labels
  target = torch.arange(8)
  target[0::2] += 1
  target[1::2] -= 1
```

```
    # Standard cross-entropy loss
    return F.cross_entropy(xcs / temperature, target, reduction="mean")
```

The code above works as long as each feature vector has exactly one positive pair in the batch when training our contrastive learning model. Let's take a look at how to handle multiple positive pairs in a contrastive learning task.

. . .

A multi-label loss for contrastive learning: NT-BXent

In the SimCLR paper, every image ***i*** has exactly 1 similar pair at index ***j***. This makes cross-entropy loss a perfect choice for the task since it resembles a multi-class problem. Instead, if we have M > 2 augmentations of the same image fed into the contrastive learning model's single training batch, then each batch would have image M-1 similar pairs for image ***i***. This task would resemble a multi-label problem.

The obvious choice would be to replace *cross-entropy loss* with *binary cross-entropy loss*. Hence the name NT-BXent loss, which stands for Normalized Temperature-scaled Binary cross-entropy Loss.

The formulation below shows the loss ***Li*** for the element ***i***. The σ in the formula below stands for the Sigmoid function.

$$l_{ij} = -y_{ij}.\log\sigma(s_{ij}/\tau) - (1 - y_{ij}).\log\sigma((1 - s_{ij})/\tau)$$

$$L_i^{weighted} = \frac{1}{N_{pos}}\Sigma_{j=1}^{N} 1_{ij}^{pos} l_{ij} + \frac{1}{N_{neg}}\Sigma_{j=1}^{N} 1_{ij}^{neg} l_{ij}$$

Figure 5: Formulation for the NT-BXent loss. Image source: Author(s) of this article

To avoid the class imbalance problem, we weigh the positive and negative pairs by the inverse of the number of positive and negative pairs in our mini-batch. The final loss in the mini-batch used for backpropagation will be the mean of the losses of each sample in our mini-batch.

Next, let’s focus our attention on our implementation of the NT-BXent loss in PyTorch.

. . .

Implementation of NT-BXent loss in PyTorch

All the code in this section can be found in this notebook.

Code Reuse: Similar to our implementation of the NT-Xent loss, we shall re-use the Binary Cross-entropy (BCE) loss method provided by PyTorch. The setup of our ground-truth labels will be similar to that of a multi-label classification problem where BCE loss is used.

Predictions Tensor: We’ll use the same (8, 2) predictions tensor as we used for the implementation of the NT-Xent loss.

```
x = torch.randn(8, 2)
```

Cosine Similarity: Since the input tensor *x* is same, the all-pairs cosine similarity tensor *xcs* will also be the same. Please see this page for a detailed explanation of what the line below does.

```
xcs = F.cosine_similarity(x[None,:,:], x[:,None,:], dim=-1)
```

To ensure that the loss from the element at position *(i, i)* is *0*, we'll need to perform some gymnastics to have our *xcs* tensor contain a value *1* at every index *(i, i)* after Sigmoid is applied to it. Since we'll be using BCE Loss, we will mark the self-similarity score of every feature vector with the value ***infinity*** in tensor *xcs*. That's because applying the sigmoid function on the *xcs* tensor, will convert infinity to the value *1*, and we will set up our ground-truth labels so that every position *(i, i)* in the ground-truth labels has the value *1*.

Let's create a masking tensor that has the value ***True*** along the principal diagonal (*xcs* has self-similarity scores along the principal diagonal), and ***False*** everywhere else.

```
eye = torch.eye(8).bool()
```

Let’s clone the tensor *“xcs”* into a tensor named *“y”* so that we can reference the *“xcs”* tensor later.

```
y = xcs.clone()
```

Now, we will set the values along the principal diagonal of the all-pairs cosine similarity matrix to *infinity* so that when we compute the sigmoid on each row, we get 1 in these positions.

```
y[eye] = float("inf")
```

The tensor *“y”* scaled by a temperature parameter will be one of the inputs (predictions) to the BCE loss API in PyTorch. Next, we need to compute the ground-truth labels (target) that we need to feed to the BCE loss API.

Ground Truth labels (Target tensor): We will expect the user to pass to us the pair of all (x, y) index pairs which contain positive examples. This is a departure for what we did for the NT-Xent loss, since the positive pairs were implicit, whereas here, the positive pairs are explicit.

In addition to the locations provided by the user, we will set all the diagonal elements as positive pairs as explained above. We will use the PyTorch tensor indexing API to pluck out all the elements at those locations and set them to 1, whereas the rest are initialized to 0.

```
target = torch.zeros(8, 8)
pos_indices = torch.tensor([
  (0, 0), (0, 2), (0, 4),
  (1, 4), (1, 6), (1, 1),
  (2, 3),
  (3, 7),
  (4, 3),
  (7, 6),
])
# Add indexes of the principal diagonal as positive indexes.
# This will be useful since we will use the BCELoss in PyTorch,
# which will expect a value for the elements on the principal
# diagonal as well.
pos_indices = torch.cat([pos_indices, torch.arange(8).reshape(8, 1).expand(-1, 2
# Set the values in the target vector to 1.
target[pos_indices[:,0], pos_indices[:,1]] = 1
```

Binary cross-entropy (BCE) Loss: Unlike the NT-Xent loss, we can't simply call the torch.nn.functional.binary_cross_entropy_function, since we want to weigh the positive and negative loss based on how many positive and negative pairs the element at index i has in the current mini-batch.

The first step though is to compute the element-wise BCE loss.

```
temperature = 0.1
loss = F.binary_cross_entropy((y / temperature).sigmoid(), target, reduction="no
```

We'll create a binary mask of positive and negative pairs and then create 2 tensors, loss_pos and loss_neg that contain only those elements from the computed loss that correspond to the positive and negative pairs.

```
target_pos = target.bool()
target_neg = ~target_pos
# loss_pos and loss_neg below contain non-zero values only for those elements
# that are positive pairs and negative pairs respectively.
loss_pos = torch.zeros(x.size(0), x.size(0)).masked_scatter(target_pos, loss[tar
loss_neg = torch.zeros(x.size(0), x.size(0)).masked_scatter(target_neg, loss[tar
```

Next, we'll sum up the positive and negative pair loss (separately) corresponding to each element i in our mini-batch.

```
# loss_pos and loss_neg now contain the sum of positive and negative pair losses
# as computed relative to the i'th input.
loss_pos = loss_pos.sum(dim=1)
loss_neg = loss_neg.sum(dim=1)
```

To perform weighting, we need to track the number of positive and negative pairs corresponding to each element i in our mini-batch. Tensors *"num_pos"* and *"num_neg"* will store these values.

```
# num_pos and num_neg below contain the number of positive and negative pairs
# computed relative to the i'th input. In an actual setting, this number should
# be the same for every input element, but we let it vary here for maximum
# flexibility.
num_pos = target.sum(dim=1)
num_neg = target.size(0) - num_pos
```

We have all the ingredients we need to compute our loss! The only thing that we need to do is weigh the positive and negative loss by the number of positive and negative pairs, and then average the loss across the mini-batch.

```
def nt_bxent_loss(x, pos_indices, temperature):
    assert len(x.size()) == 2

    # Add indexes of the principal diagonal elements to pos_indices
    pos_indices = torch.cat([
        pos_indices,
        torch.arange(x.size(0)).reshape(x.size(0), 1).expand(-1, 2),
    ], dim=0)

    # Ground truth labels
    target = torch.zeros(x.size(0), x.size(0))
    target[pos_indices[:,0], pos_indices[:,1]] = 1.0

    # Cosine similarity
    xcs = F.cosine_similarity(x[None,:,:], x[:,None,:], dim=-1)
    # Set logit of diagonal element to "inf" signifying complete
    # correlation. sigmoid(inf) = 1.0 so this will work out nicely
    # when computing the Binary cross-entropy Loss.
    xcs[torch.eye(x.size(0)).bool()] = float("inf")

    # Standard binary cross-entropy loss. We use binary_cross_entropy() here and
    # binary_cross_entropy_with_logits() because of
    # https://github.com/pytorch/pytorch/issues/102894
    # The method *_with_logits() uses the log-sum-exp-trick, which causes inf an
    # to result in a NaN result.
    loss = F.binary_cross_entropy((xcs / temperature).sigmoid(), target, reducti

    target_pos = target.bool()
    target_neg = ~target_pos

    loss_pos = torch.zeros(x.size(0), x.size(0)).masked_scatter(target_pos, loss
    loss_neg = torch.zeros(x.size(0), x.size(0)).masked_scatter(target_neg, loss
    loss_pos = loss_pos.sum(dim=1)
    loss_neg = loss_neg.sum(dim=1)
    num_pos = target.sum(dim=1)
    num_neg = x.size(0) - num_pos

    return ((loss_pos / num_pos) + (loss_neg / num_neg)).mean()

pos_indices = torch.tensor([
    (0, 0), (0, 2), (0, 4),
    (1, 4), (1, 6), (1, 1),
    (2, 3),
    (3, 7),
    (4, 3),
    (7, 6),
])
for t in (0.01, 0.1, 1.0, 10.0, 20.0):
    print(f"Temperature: {t:5.2f}, Loss: {nt_bxent_loss(x, pos_indices, temperat
```

Prints.

> *Temperature: 0.01, Loss: 62.898780822753906*
>
> *Temperature: 0.10, Loss: 4.851151943206787*
>
> *Temperature: 1.00, Loss: 1.0727109909057617*
>
> *Temperature: 10.00, Loss: 0.9827173948287964*
>
> *Temperature: 20.00, Loss: 0.982099175453186*

. . .

Conclusion

Self-supervised learning is an upcoming field in deep learning and allows one to train models on unlabeled data. This technique lets us work around the requirement of labeled data at scale.

In this article, we learned about loss functions for contrastive learning. The first one, named NT-Xent loss, is used for learning on a single positive pair per input in a mini-batch. We introduced the NT-BXent loss which is used for learning on multiple (> 1) positive pairs per input in a mini-batch. We learned to interpret them intuitively, building on our knowledge of cross-entropy loss and binary cross-entropy loss. Finally, we implemented them both efficiently in PyTorch.

Pytorch Self Supervised Learning Loss Function Cross Entropy Binary Cross Entropy

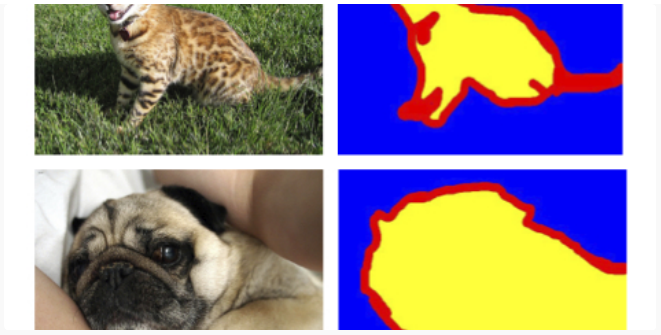
Written by Dhruv Matani

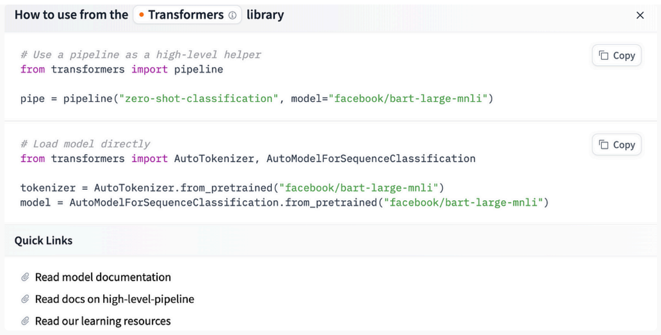
Follow

202 Followers · Writer for Towards Data Science

Machine Learning, PyTorch, CNNs, Transformers, Vision, Speech, Text AI. On-Device AI, Model Optimization, ML and Data Infrastructure. My views are my own.

More from Dhruv Matani and Towards Data Science




How to use from the Transformers library

```
# Use a pipeline as a high-level helper
from transformers import pipeline

pipe = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
```

```
# Load model directly
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained("facebook/bart-large-mnli")
model = AutoModelForSequenceClassification.from_pretrained("facebook/bart-large-mnli")
```

Quick Links

- Read model documentation
- Read docs on high-level-pipeline
- Read our learning resources

Dhruv Matani in Towards Data Science

Efficient Image Segmentation Using PyTorch: Part 1

Concepts and Ideas

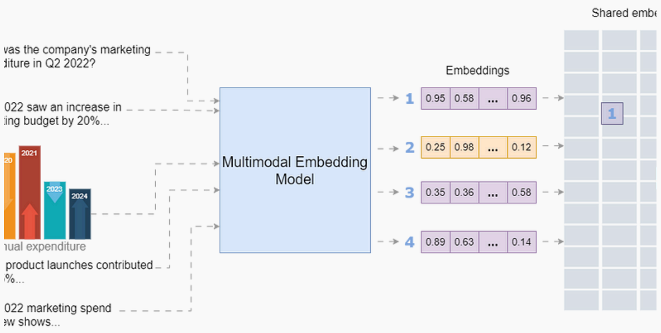
Jun 26, 2023 134

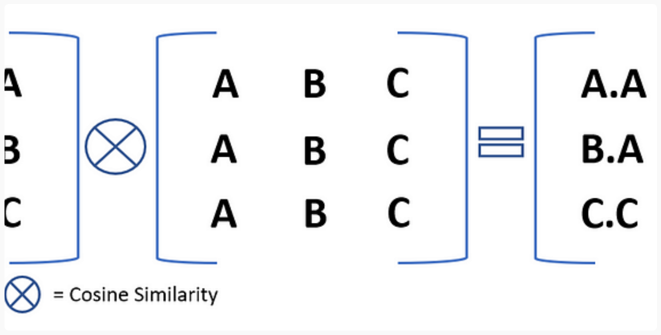
Stephanie Kirmer in Towards Data Science

Choosing and Implementing Hugging Face Models

Pulling pre-trained models out of the box for your use case

1d ago 326 4





Umair Ali Khan in Towards Data Science

Multimodal AI Search for Business Applications

Enabling businesses to extract real value from their data

1d ago 276 4

Dhruv Matani

All Pairs Cosine Similarity in PyTorch

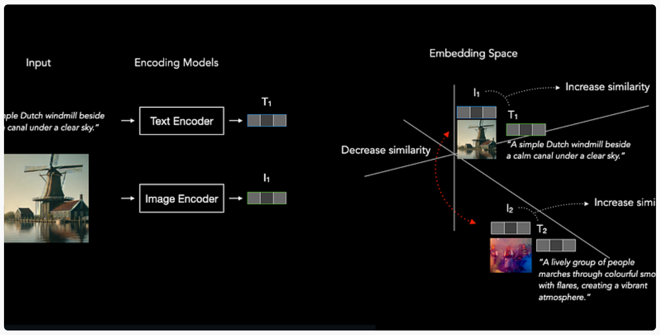
PyTorch defines a cosine_similarity function to compute pairwise cosine similarity…

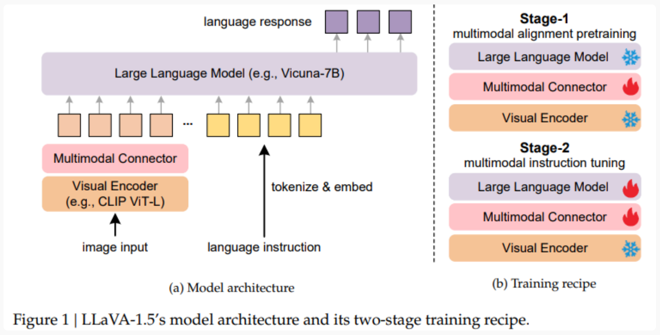
Jun 7, 2023 125 2

See all from Dhruv Matani

See all from Towards Data Science

Recommended from Medium




language response
Large Language Model (e.g., Vicuna-7B)
Multimodal Connector
Visual Encoder (e.g., CLIP ViT-L)
image input
tokenize & embed
language instruction
(a) Model architecture
Stage-1 multimodal alignment pretraining
Stage-2 multimodal instruction tuning
Large Language Model
Multimodal Connector
Visual Encoder
(b) Training recipe
Figure 1 | LLaVA-1.5's model architecture and its two-stage training recipe.

Frederik vom Lehn in Self-Supervised Learning

Understanding CLIP for vision language models

How do vision language models work? A very detailed guide.

Sep 29 102

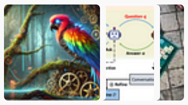
Don Moon in Byte-Sized AI

Multi-Modal Vision Language Models: Architecture and Key…

Understanding multi-modal vision language models

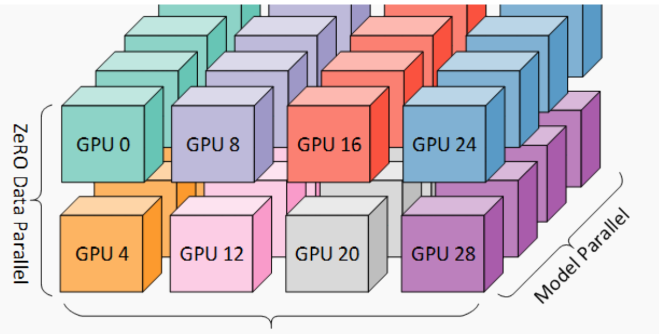
May 22 7

Lists



Natural Language Processing

1789 stories · 1391 saves

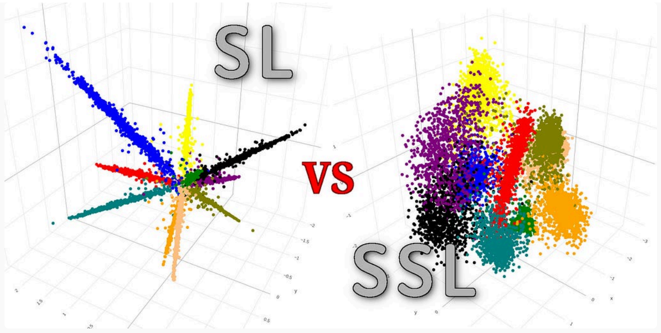


Jasper Zhong

How Does LLM Training Scale to Over 10,000 GPUs? The 4D...

As large language models (LLMs) like GPT-4 and Claude-3.5 continue to push the...

Jul 2 9

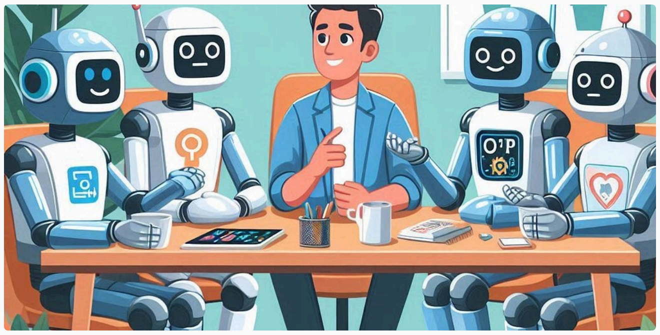


Challenge Enthusiast

Self-supervised learning demonstrated

Self-supervised learning is a machine learning approach that enables models to learn from...

Jun 28 19

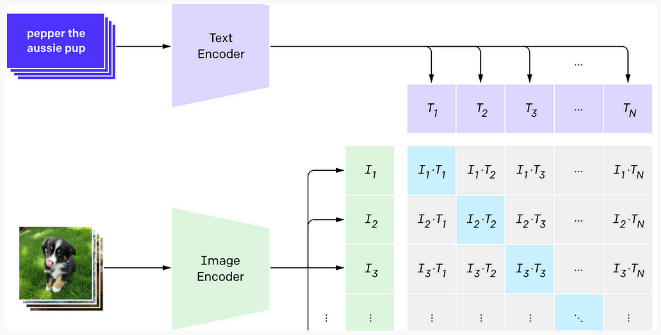


Murat Şimşek in Towards AI

The Next Evolution in Artificial Intelligence: Agentic AI

From Reactive Systems to Autonomous Teams, Agentic AI is almost everywhere soon.

4d ago 57


pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_2 \cdot T_1$ $I_2 \cdot T_2$ $I_2 \cdot T_3$ ... $I_2 \cdot T_N$
$I_3 \cdot T_1$ $I_3 \cdot T_2$ $I_3 \cdot T_3$ ... $I_3 \cdot T_N$

heping_LU

SigLIP vs. CLIP: The Sigmoid Advantage

Enhancing Quality and Efficiency in Language-Image Pre-Training

Sep 25

See more recommendations