Deep Canonical Correlation Analysis (DeepCCA)

Srijan Agrawal (0801CS171084)
gs0801cs171084@sgsitsindore.in

December 20 , 2020

Content:

1. Introduction **3**
1.1 Deep Learning 3
1.2 Deep CCA 3
2. Mathematical Formulation **4**
2.1 Formulation 4
2.2 Gradient Formulation 5
3. Algorithm **6**
3.1 DeepCCA Algorithm 6
4. Documentation of API **7**
4.1 DeepCCAModels 7
4. DeepCCA 7
5. Example **9**
5.1 Example 1 9
6. Learning Outcome **11**
A References **12**

Introduction

1.1 Deep Learning

“Deep” networks, having more than two layers, are capable of representing nonlinear functions involving multiple nested high-level abstractions of the kind that may be necessary to accurately model complex real-world data. There has been a resurgence of interest in such models following the advent of various successful unsupervised methods for initializing the parameters(“pretraining”) in such a way that a useful solution can be found (Hinton et al., 2006; Hinton & Salakhutdinov,2006). Contrastive divergence (Bengio & Delalleau,2009) has had great success as a pre-training technique,as have many variants of autoencoder networks, including the denoising autoencoder (Vincent et al., 2008)used in the present work. The growing availability of both data and compute resources also contributes to the resurgence, because empirically the performance of deep networks seems to scale very well with data size and complexity.

1.2 DeepCCA

While deep networks are more commonly used for learning classification labels or mapping to another vector space with supervision, here we use them to learn non-linear transformations of two datasets to a space in which the data is highly correlated. The same properties that may account for deep networks’ success in other tasks—high model complexity,the ability to concisely represent a hierarchy of features for modeling real-world data distributions—could be particularly useful in a setting where the output space is significantly more complex than a single label.

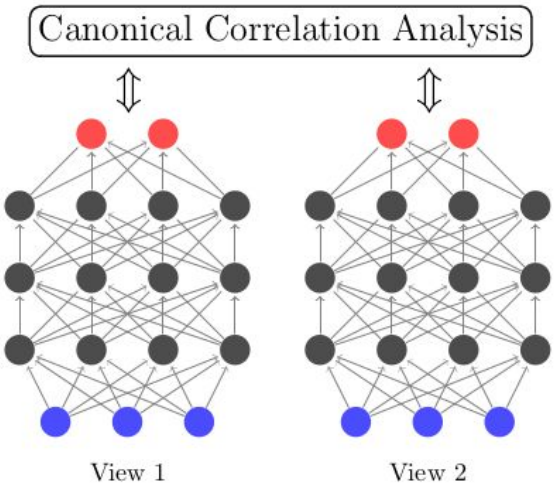
Mathematical Formula

2.1 Formulation

Deep CCA computes representations of the two views by passing them through multiple stacked layers of nonlinear transformation (see Figure 1). Assume for simplicity that each intermediate layer in the network for the first view has c1 units, and the final (output)layer has units. Let $x^1 \in \boldsymbol{R}^{n1}$ be an instance of the first view. The outputs of the first layer for the instance x^1 are $h^1 = s(W_1^1 x_1 + b_1^1) \in \boldsymbol{R}^{c1}$, where $W_1^1 \in R^{c1 \times n1}$ is a matrix of weights, $b_1^1 \in R^{c1}$ is a vector of biases, and $s : \boldsymbol{R} \rightarrow \boldsymbol{R}$ is a nonlinear function applied componentwise. The outputs h_1 may then be used to compute the outputs of the next layer as $h_2 = s(W^1{}_2 h_1 + b^1{}_2) \in \boldsymbol{R}^{c1}$, and so on until the final representation $f_1(x_1) = s(W^1{}_d h_{d-1} + b^1{}_d) \in R^o$ is computed, for a network with layers. Given an instance 2 of the second view, the representation $f_2(x_2)$ is computed the same way, with different parameters W_l^2 and $b^2{}_l$ (and potentially different architectural parameters c_2 and d).

The goal is to jointly learn parameters for both views W_l^v and $b^v{}_l$ such that $corr(f_1(X_1), f_2(X_2))$ is as high as possible. If θ_1 is the vector of all parameters W_l^1 and $b^1{}_l$ of the first view for $l = 1, \ldots, d$, and similarly for θ_2 , then

$$(\theta_1^*, \theta_2^*) = \underset{(\theta_1, \theta_2)}{argmax}\ corr(f_1(X_1; \theta_1), f_2(X_2; \theta_2))$$



To find (θ_1^*, θ_2^*), we follow the gradient of the correlation objective as estimated on the training data. Let $H_1 \in \boldsymbol{R}^{o \times m}$, $H_2 \in \boldsymbol{R}^{o \times m}$ be matrices whose columns are the top-level representations produced by the deep models on the two views, for a training set of size m. Let $\overline{H}_1 = \mathrm{H}_1 - \frac{1}{m} \mathrm{H}_1$ be the centered data matrix (resp. $\overline{H}_2$), and define $\widehat{\Sigma}_{12} = \frac{1}{m-1} \overline{H}_1 \overline{H}_2^T$, and $\widehat{\Sigma}_{11} = \frac{1}{m-1} \overline{H}_1 \overline{H}_1^T + r_1 I$ for regularization constant r_1 (resp. $\widehat{\Sigma}_{22}$). Assume that $r_1 > 0$ so that $\widehat{\Sigma}_{11}$ is positive definite.

The total correlation of the top k components of H_1 and H_2 is the sum of the top k singular values of the matrix $T = \widehat{\Sigma}_{11}^{-1/2} \widehat{\Sigma}_{12} \widehat{\Sigma}_{12}^{-1/2}$.

If we take k = o, then this is exactly the matrix trace norm of T , or

$$corr(H_1, H_2) = ||T||_{tr} = tr(T^T T)^{1/2}$$

2.2 Gradient Formulation

The parameters W_l^v and $b^v{}_l$ of DCCA are trained to optimize this quantity using gradient based optimization. To compute the gradient of $corr(H_1, H_2)$ with respect to all parameters W_l^v and $b^v{}_l$, we can compute its gradient with respect to H_1 and H_2 and then use backpropagation. If the singular value decomposition of T is $T = UDV^T$, then

$$\frac{\partial corr(H_1, H_2)}{\partial H_1} = \frac{1}{m-1}(2 \nabla_{11} \overline{H}_1 + \nabla_{12} \overline{H}_2)$$

where

$$\nabla_{12} = \widehat{\Sigma}_{11}^{-1/2} U V^T \widehat{\Sigma}_{22}^{-1/2}$$

and

$$\nabla_{11} = -\frac{1}{2} \widehat{\Sigma}_{11}^{-1/2} U D U^T \widehat{\Sigma}_{22}^{-1/2}$$

and $\partial \mathrm{corr}(\mathrm{H}_1, \mathrm{H}_2)/\partial \mathrm{H}_2$ has a symmetric expression.

Algorithm

In this section, we will give an overview of the Deep CCA Algorithm.

3.1 DeepCCA Algorithm

Algorithms 1 : DeepCCA algorithm

Input : multiview training datasets $X_1 \in R^{n \times d_1}$, $Y \in R^{n \times d_2}$
Where d_1 and d_2 are the dimensionality of X and Y views.
Output : Reduced Dimensions of input Matrices.

---

1. Construct $\theta_1 = (W_1^1, b_1^1), (W_2^1, b_2^1) \ldots (W_l^1, b_l^1)$, $\theta_2 = (W_1^2, b_1^2) \ldots (W_l^2, b_l^2)$, where l is the number of layers and θ is the parameters in the Multilayer perceptron model.
2. Compute $H_1 = f_1(X_1 ; \theta_1)$ and $H_2 = f_2(X_2 ; \theta_2)$ as model output .
3. Compute $\overline{H}_1$ and $\overline{H}_2$ as the mean centered of H_1 and H_2.
4. Covariance matrix:
   $\hat{C}_{12} = (\overline{H}_1 \overline{H}_2^T)/(m-1)$.
5. Compute C_{11} and C_{22} .
6. Compute matrix $T = \widehat{C}_{11}^{-1/2} \widehat{C}_{12} \widehat{C}_{12}^{-1/2}$.
7. *Compute* $corr(H_1, H_2) = ||T||_{tr}$.
8. Compute Gradient of $corr(H_1, H_2)$ *w.r.t.* H_1 and H_2.
9. Backpropagate error and update weights for n iterations .
10. Obtain H_1 and H_2 from model output .

Documentation of API

4.1 DeepCCAModels

class **DeepCCA.DeepCCAModels.Model**(*layer_sizes1: list*, *layer_sizes2: list*, *input_size1: int*, *input_size2: int*, *outdim_size: int*, *use_all_singular_values: bool = False*, *device: torch.device = device(type='cpu')*)

Bases: torch.nn.modules.module.Module

__init__(*layer_sizes1: list*, *layer_sizes2: list*, *input_size1: int*, *input_size2: int*, *outdim_size: int*, *use_all_singular_values: bool = False*, *device: torch.device = device(type='cpu')*)

model initialization

Parameters

- **layer_sizes1** (*list*) – list of layer shape of view 1
- **layer_sizes2** (*list*) – list of layer shape of view 1
- **input_size1** (*int*) – input dimension of view 1
- **input_size2** (*int*) – input dimension of view 2
- **outdim_size** (*int*) – output dimension of data use_all_singular_values (bool, optional): specifies if all the singular values should get used to calculate the correlation or just the top outdim_size ones. Defaults to False.
- **device** (*torch.device, optional*) – device type GPU/CPU. Defaults to torch.device('cpu').

4.2 DeepCCA

class **DeepCCA.main.DeepCCA**(*model: DeepCCA.DeepCCAModels.Model*, *outdim_size: int*, *epoch_num: int*, *batch_size: int*, *learning_rate: float*, *reg_par: float*, *linearcca: bool = False*, *device=device(type='cpu')*)

Bases: object

__init__(*model: DeepCCA.DeepCCAModels.Model*, *outdim_size: int*, *epoch_num: int*, *batch_size: int*, *learning_rate: float*, *reg_par: float*, *linearcca: bool = False*, *device=device(type='cpu')*)

initialize object

Parameters

- **model** (*Model*) – Pytorch Model
- **outdim_size** (*int*) – size of output dimensions
- **epoch_num** (*int*) – Number of iterations on data
- **batch_size** (*int*) – size of batch while training
- **learning_rate** (*float*) – Learning rate of the model
- **reg_par** (*float*) – regularization parameter
- **linearcca** (*bool, optional*) – apply linear cca on model output. Defaults to False.
- **device** (*[type], optional*) – [select device type GPU / CPU. Defaults to torch.device('cpu').

fit(*x1: torch.Tensor*, *x2: torch.Tensor*, *vx1: torch.Tensor = None*, *vx2: torch.Tensor = None*, *tx1: torch.Tensor = None*, *tx2: torch.Tensor = None*, *checkpoint: str = 'checkpoint.model'*)

train model with the given dataset

Parameters

- **x1** (*torch.Tensor*) – training data of view 1
- **x2** (*torch.Tensor*) – training data of view 2
- **vx1** (*torch.Tensor, optional*) – validation data of view 1. Defaults to None.
- **vx2** (*torch.Tensor, optional*) – validation data of view 2. Defaults to None.
- **tx1** (*torch.Tensor, optional*) – testing data of view 1. Defaults to None.
- **tx2** (*torch.Tensor, optional*) – testing data of view 2. Defaults to None.
- **checkpoint** (*str, optional*) – model weights saving location. Defaults to 'checkpoint.model'.

transform(*x1: torch.Tensor*, *x2: torch.Tensor*, *use_linear_cca: bool = False*) → list

get output of the model

Parameters

- **x1** (*torch.Tensor*) – view 1 data
- **x2** (*torch.Tensor*) – view 2 data
- **use_linear_cca** (*bool, optional*) – apply linear cca on model output. Defaults to False.

Returns

List containing transformed matrices .

Example

5.1 Example 1

```
%load_ext autoreload
%autoreload 2
import torch
import torch.nn as nn
import numpy as np
from torch.utils.data import BatchSampler, SequentialSampler
torch.set_default_tensor_type(torch.DoubleTensor)

from DeepCCA import DeepCCA , Model

device = torch.device('cuda')
print("Using", torch.cuda.device_count(), "GPUs")
# the path to save the final learned features
save_to = './new_features.gz'
# the size of the new space learned by the model (number of the new
features)
outdim_size = 10
# size of the input for view 1 and view 2
input_shape1 = 784
input_shape2 = 784
# number of layers with nodes in each one
layer_sizes1 = [1024, 1024, 1024, outdim_size]
layer_sizes2 = [1024, 1024, 1024, outdim_size]
# the parameters for training the network
learning_rate = 1e-3
epoch_num = 1
batch_size = 800
# apply linear CCA on model output
linear_cca= False

# the regularization parameter of the network
```

```
# seems necessary to avoid the gradient exploding especially when
non-saturating activations are used
reg_par = 1e-5
# specifies if all the singular values should get used to calculate the
correlation or just the top outdim_size ones
# if one option does not work for a network or dataset, try the other one
use_all_singular_values = False
# if a linear CCA should get applied on the learned features extracted
from the networks
# it does not affect the performance on noisy MNIST significantly
apply_linear_cca = True
# end of parameters section
############

data1 = torch.randn((100,784))
data2 = torch.randn((100,784))

model = Model(layer_sizes1, layer_sizes2, input_shape1,input_shape2,
outdim_size, use_all_singular_values).double()

deepcca = DeepCCA(model,  outdim_size , epoch_num , batch_size ,
learning_rate , reg_par , linear_cca  )


deepcca.fit(data1 , data2)
outputs = deepcca.transform(data1,data2, linear_cca)
```

Learning Outcome

1. Studied CCA , Lagrange Optimisation , use of Eigenvalue Decomposition to find Inverse square root of a matrix.
2. Studied and implemented DeepCCA.
3. Implementation of Feedforward Network in PyTorch framework.

References

1. http://proceedings.mlr.press/v28/andrew13.pdf
2. https://github.com/Michaelvll/DeepCCA
3. https://arxiv.org/pdf/1907.01693.pdf