Additional materials

Mutual Information estimator

François Fleuret

https://fleuret.org/ee559/

Thu Jan 17 11:42:25 UTC 2019

idiap RESEARCH INSTITUTE

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

An important quantity from information theory is the mutual information

$$\begin{aligned}\mathbb{I}(A;B) &= \mathbb{H}(A) + \mathbb{H}(B) - \mathbb{H}(A,B) \\ &= \mathbb{H}(A) - \mathbb{H}(A \mid B).\end{aligned}$$

An important quantity from information theory is the mutual information

$$\begin{aligned}\mathbb{I}(A;B) &= \mathbb{H}(A) + \mathbb{H}(B) - \mathbb{H}(A,B) \\ &= \mathbb{H}(A) - \mathbb{H}(A \mid B).\end{aligned}$$

Which is also

$$\mathbb{I}(A;B) = \mathbb{D}_{KL}\left(\mu_{A,B} \,\|\, \mu_A \otimes \mu_B\right)$$

where

$$\mathbb{D}_{KL}(p\|q) = \int \log\left(\frac{p(x)}{q(x)}\right) p(x) dx.$$

An important quantity from information theory is the mutual information

$$\begin{aligned}\mathbb{I}(A;B) &= \mathbb{H}(A) + \mathbb{H}(B) - \mathbb{H}(A,B) \\ &= \mathbb{H}(A) - \mathbb{H}(A \mid B).\end{aligned}$$

Which is also

$$\mathbb{I}(A;B) = \mathbb{D}_{KL}\left(\mu_{A,B} \,\|\, \mu_A \otimes \mu_B\right)$$

where

$$\mathbb{D}_{KL}(p\|q) = \int \log\left(\frac{p(x)}{q(x)}\right) p(x) dx.$$

Which translates in the finite case into

$$\mathbb{I}(A;B) = \sum_{i,j} P(A=i, B=j) \log\left(\frac{P(A=i, B=j)}{P(A=i)P(B=j)}\right).$$

An important quantity from information theory is the mutual information

$$\begin{aligned}\mathbb{I}(A;B) &= \mathbb{H}(A) + \mathbb{H}(B) - \mathbb{H}(A,B) \\ &= \mathbb{H}(A) - \mathbb{H}(A \mid B).\end{aligned}$$

Which is also

$$\mathbb{I}(A;B) = \mathbb{D}_{KL}\left(\mu_{A,B} \,\|\, \mu_A \otimes \mu_B\right)$$

where

$$\mathbb{D}_{KL}(p\|q) = \int \log\left(\frac{p(x)}{q(x)}\right) p(x) dx.$$

Which translates in the finite case into

$$\mathbb{I}(A;B) = \sum_{i,j} P(A=i, B=j) \log\left(\frac{P(A=i, B=j)}{P(A=i)P(B=j)}\right).$$

In the continuous case, the standard approach to estimating it is with parametric density models, which is unsatisfactory for real-world high-dimension signals.

Belghazi et al. (2018) propose to use the Donsker-Varadhan representation of the KL divergence (Donsker and Varadhan, 1983) to leverage deep models:

$$\mathbb{D}_{KL}(p\|q) = \sup_{T:\Omega\to\mathbb{R}} \mathbb{E}_p\left[T\right] - \log \mathbb{E}_q\left[e^T\right].$$

Belghazi et al. (2018) propose to use the Donsker-Varadhan representation of the KL divergence (Donsker and Varadhan, 1983) to leverage deep models:

$$\mathbb{D}_{KL}(p\|q) = \sup_{T:\Omega\to\mathbb{R}} \mathbb{E}_p\left[T\right] - \log \mathbb{E}_q\left[e^T\right].$$

We can derive this expression in the finite case.

We have

$$\mathbb{E}_p\left[T\right] - \log \mathbb{E}_q\left[e^T\right] = \sum_i p_i t_i - \log \sum_i q_i e^{t_i}.$$

We define $\forall i,\ T(i) = t_i$, and we have, $\forall j$,

$$\frac{\partial}{\partial t_j}\left(\sum_i p_i t_i - \log \sum_i q_i e^{t_i}\right) = 0$$

$$\Rightarrow\ p_j - \frac{q_j e^{t_j}}{\sum_i q_i e^{t_i}} = 0$$

$$\Rightarrow\ p_j \sum_i q_i e^{t_i} = q_j e^{t_j}$$

$$\Rightarrow\ t_j = \log \frac{p_j}{q_j} + \underbrace{\log \sum_i q_i e^{t_i}}_{\alpha}$$

from which

$$\begin{aligned}
\sum_i p_i t_i - \log \sum_i q_i e^{t_i} &= \sum_i p_i \left(\log \frac{p_i}{q_i} + \alpha\right) - \log \sum_i q_i \exp\left(\log \frac{p_i}{q_i} + \alpha\right) \\
&= \sum_i p_i \left(\log \frac{p_i}{q_i} + \alpha\right) - \log \sum_i e^{\alpha} q_i \frac{p_i}{q_i} \\
&= \sum_i p_i \log \frac{p_i}{q_i} + \alpha - \log e^{\alpha} \\
&= \mathbb{D}_{KL}(p\|q).
\end{aligned}$$

Given pairs of samples

$$x_n \sim p, y_n \sim q, \;\; n = 1, \ldots, N$$

we can replace the sup over all mappings with the optimization of a model:

$$\hat{\mathbb{D}}_{KL}\left(\mu_X \,\|\, \mu_Y\right) = \sup_w \underbrace{\frac{1}{N}\sum_{n=1}^{N} f(x_n; w)}_{\hat{\mathbb{E}}_{\mu_X}[f(X;w)]} - \underbrace{\log\left(\frac{1}{N}\sum_{n=1}^{N}\exp(f(y_n; w))\right)}_{\log \hat{\mathbb{E}}_{\mu_Y}\left[e^{f(Y;w)}\right]}$$

And given pairs of samples i.i.d $\sim \mu_{A,B}$

$$(a_n, b_n), n = 1, \ldots, N,$$

with σ a random permutation, we can use

$$\left(a_n, b_{\sigma(n)}\right), n = 1, \ldots, N$$

as i.i.d samples $\sim \mu_A \otimes \mu_B$, from which we have

$$\hat{\mathbb{I}}(A; B) = \sup_w \underbrace{\frac{1}{N}\sum_{n=1}^{N} f(a_n, b_n; w)}_{\hat{\mathbb{E}}_{\mu_{A,B}}[f(A,B;w)]} - \underbrace{\log\left(\frac{1}{N}\sum_{n=1}^{N} \exp(f(a_n, b_{\sigma(n)}; w))\right)}_{\log \hat{\mathbb{E}}_{\mu_A \otimes \mu_B}\left[e^{f(A,B;w)}\right]}$$

$$\hat{\mathbb{I}}(A;B) = \sup_w \frac{1}{N}\sum_{n=1}^{N} f(a_n, b_n; w) - \log\left(\frac{1}{N}\sum_{n=1}^{N}\exp(f(a_n, b_{\sigma(n)}; w))\right)$$

```
for e in range(nb_epochs):

    perm_sigma = torch.randperm(input_b.size(0))
    input_br = input_b[perm_sigma]

    for ba, bb, bbr in zip(input_a.split(batch_size),
                           input_b.split(batch_size),
                           input_br.split(batch_size)):

        # We want the max
        loss = - ( model(ba, bb).mean() - model(ba, bbr).exp().mean().log() )
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

A simple experiment with

- a_n a MNIST image from a fixed subset of classes $\mathscr{C}$,
- b_n another MNIST image of same class.

We have $\mathbb{I}(A; B) = \log(|\mathscr{C}|)$.

```
class NetForImagePair(nn.Module):
    def __init__(self):
        super(NetForImagePair, self).__init__()
        self.features_a = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size = 5),
            nn.MaxPool2d(3), nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size = 5),
            nn.MaxPool2d(2), nn.ReLU(),
        )

        self.features_b = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size = 5),
            nn.MaxPool2d(3), nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size = 5),
            nn.MaxPool2d(2), nn.ReLU(),
        )

        self.fully_connected = nn.Sequential(
            nn.Linear(256, 200),
            nn.ReLU(),
            nn.Linear(200, 1)
        )

    def forward(self, a, b):
        a = self.features_a(a).view(a.size(0), -1)
        b = self.features_b(b).view(b.size(0), -1)
        x = torch.cat((a, b), 1)
        return self.fully_connected(x)
```

After each epoch, we print the current estimated MI in base 2, and the "true" one, which is the entropy of the class. Note that the classes are not exactly balanced.

```
./mine_mnist.py --data image_pair --mnist_classes='4,5,8,9'
1 0.3970 1.9991
2 1.0787 1.9991
3 1.3101 1.9991
4 1.5023 1.9991
5 1.5645 1.9991
6 1.6162 1.9991
7 1.6721 1.9991
8 1.6218 1.9991
9 1.7119 1.9991
10 1.7441 1.9991
20 1.8763 1.9991
30 1.9179 1.9991
40 1.9271 1.9991
50 1.9223 1.9991
test 2.0831 1.9985
```

```
./mine_mnist.py --data image_pair --mnist_classes='0,2,3,7,4,5,8,9'
1 0.7786 2.9989
2 1.6795 2.9989
3 1.9775 2.9989
4 2.1605 2.9989
5 2.2594 2.9989
6 2.3503 2.9989
7 2.3935 2.9989
8 2.5045 2.9989
9 2.5187 2.9989
10 2.5903 2.9989
20 2.7748 2.9989
30 2.8146 2.9989
40 2.8787 2.9989
50 2.8312 2.9989
test 2.8946 2.9987
```

A more complicated experiment with

- a_n a MNIST image from a fixed subset of classes $\mathscr{C}$,
- c_n its class in $\{0, \ldots, 9\}$,
- $\alpha_n \sim \mathscr{U}([0, 10])$,
- $\epsilon_n \sim \mathscr{U}([0, 1/2])$,
- $b_n = (\alpha_n, \alpha_n + \epsilon_n + c_n) \in \mathbb{R}^2$.

We have $\mathbb{I}(A; B) = \log(|\mathscr{C}|)$.

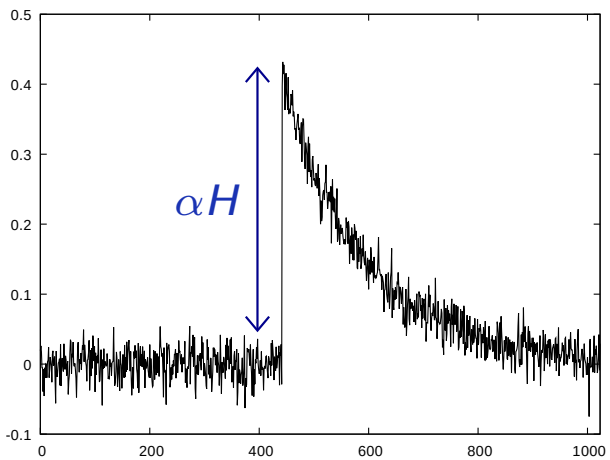
```
class NetForImageValuesPair(nn.Module):
    def __init__(self):
        super(NetForImageValuesPair, self).__init__()
        self.features_a = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size = 5),
            nn.MaxPool2d(3), nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size = 5),
            nn.MaxPool2d(2), nn.ReLU(),
        )

        self.features_b = nn.Sequential(
            nn.Linear(2, 32), nn.ReLU(),
            nn.Linear(32, 32), nn.ReLU(),
            nn.Linear(32, 128), nn.ReLU(),
        )

        self.fully_connected = nn.Sequential(
            nn.Linear(256, 200),
            nn.ReLU(),
            nn.Linear(200, 1)
        )

    def forward(self, a, b):
        a = self.features_a(a).view(a.size(0), -1)
        b = self.features_b(b).view(b.size(0), -1)
        x = torch.cat((a, b), 1)
        return self.fully_connected(x)
```

```
./mine_mnist.py --data image_values_pair --mnist_classes='4,5,8,9'
1 0.6008 1.9991
2 1.0139 1.9991
3 1.1663 1.9991
4 1.3079 1.9991
5 1.4031 1.9991
6 1.4222 1.9991
7 1.4933 1.9991
8 1.5533 1.9991
9 1.5327 1.9991
10 1.5451 1.9991
20 1.7382 1.9991
30 1.8183 1.9991
40 1.8690 1.9991
50 1.8883 1.9991
test 1.8895 1.9985
```

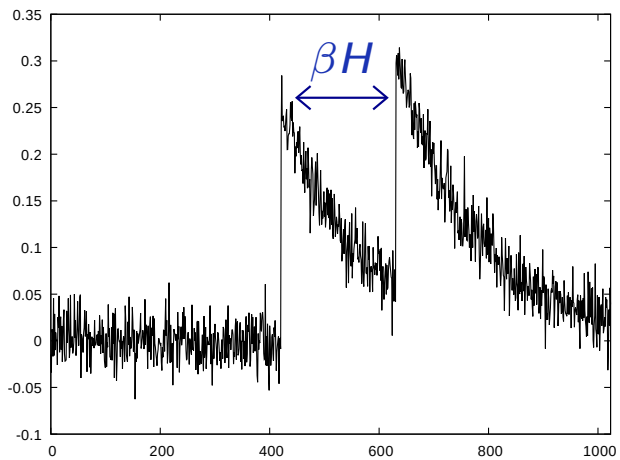
```
./mine_mnist.py --data image_values_pair --mnist_classes='0,2,3,7,4,5,8,9'
1 1.0164 2.9989
2 1.6143 2.9989
3 1.8919 2.9989
4 2.0464 2.9989
5 2.2027 2.9989
6 2.2775 2.9989
7 2.3232 2.9989
8 2.4043 2.9989
9 2.3774 2.9989
10 2.4653 2.9989
20 2.6591 2.9989
30 2.7886 2.9989
40 2.8272 2.9989
50 2.8878 2.9989
test 2.9768 2.9987
```

An example with sequences.

$H \sim \mathcal{U}\{1, \ldots, Q\}$, A is a sequences with a peak at a random location, of amplitude αH, and B is a sequence with two peaks at random locations βH apart, with Q, α, β constants.



A



B

We have $\mathbb{I}(A; B) = \log(Q)$.

```
class NetForSequencePair(nn.Module):

    def feature_model(self):
        kernel_size = 11
        pooling_size = 4
        return  nn.Sequential(
            nn.Conv1d(      1, self.nc, kernel_size = kernel_size),
            nn.AvgPool1d(pooling_size),
            nn.LeakyReLU(),
            nn.Conv1d(self.nc, self.nc, kernel_size = kernel_size),
            nn.AvgPool1d(pooling_size),
            nn.LeakyReLU(),
            nn.Conv1d(self.nc, self.nc, kernel_size = kernel_size),
            nn.AvgPool1d(pooling_size),
            nn.LeakyReLU(),
            nn.Conv1d(self.nc, self.nc, kernel_size = kernel_size),
            nn.AvgPool1d(pooling_size),
            nn.LeakyReLU(),
        )
```

```
def __init__(self):
    super(NetForSequencePair, self).__init__()

    self.nc = 32
    self.nh = 256

    self.features_a = self.feature_model()
    self.features_b = self.feature_model()

    self.fully_connected = nn.Sequential(
        nn.Linear(2 * self.nc, self.nh),
        nn.ReLU(),
        nn.Linear(self.nh, 1)
    )

def forward(self, a, b):
    a = a.view(a.size(0), 1, a.size(1))
    a = self.features_a(a)
    a = F.avg_pool1d(a, a.size(2))

    b = b.view(b.size(0), 1, b.size(1))
    b = self.features_b(b)
    b = F.avg_pool1d(b, b.size(2))

    x = torch.cat((a.view(a.size(0), -1), b.view(b.size(0), -1)), 1)
    return self.fully_connected(x)
```

```
./mine_mnist.py --data sequence_pair --nb_classes=16 --nb_epochs=200
1 -0.0000 3.9993
2 0.0000 3.9984
3 0.0007 3.9982
4 0.0223 3.9990
5 0.2610 3.9989
6 0.6664 3.9993
7 0.9216 3.9991
8 1.1370 3.9983
9 1.3446 3.9991
10 1.4704 3.9991
20 2.2416 3.9988
30 2.7459 3.9981
40 2.9052 3.9992
50 3.0245 3.9987
100 3.4294 3.9992
150 3.5365 3.9989
200 3.4900 3.9993
test 3.5740 3.9980
```

```
./mine_mnist.py --data sequence_pair --nb_classes=16 --nb_epochs=200 --independent
1 -0.0000 3.9993
2 -0.0000 3.9992
3 -0.0000 3.9991
4 -0.0000 3.9989
5 -0.0000 3.9991
6 -0.0000 3.9988
7 -0.0000 3.9985
8 -0.0000 3.9988
9 -0.0000 3.9985
10 -0.0000 3.9983
20 -0.0000 3.9987
30 -0.0000 3.9986
40 -0.0000 3.9988
50 -0.0000 3.9985
100 -0.0000 3.9989
150 0.0000 3.9993
200 0.0000 3.9989
test 0.0000 3.9991
```

The end

References

M. Belghazi, A. Baratin, S. Rajeswar, S. Ozair, Y. Bengio, A. Courville, and R. Hjelm. Mine: Mutual information neural estimation. *CoRR*, abs/1801.04062, 2018.

M. Donsker and S. Varadhan. Asymptotic evaluation of certain Markov process expectations for large time. *Communications on Pure and Applied Mathematics*, 36(2): 183–212, 1983.