

Дистиллирование данных

Марков В.О.
МГУ ВМК

9 апреля 2020 г.

1 Цели

Изучить статьи по дистилляции данных, подчеркнуть главные моменты и написать реализующие алгоритмы.

2 Основы

Ранее использовали дистилляцию самих нейросетевых моделей. Семейство нейросетей превращали в одну нейросеть. Дистилляция данных, в свою очередь, превращает большой набор данных в значительно более сжатый по объёму набор, точность на котором не хуже точности на исходных данных после тренировки модели.

Плюсы данного подхода

- В уменьшении времени тренировки
- В уменьшении затрат на хранение данных

Также рассмотрим несколько видов инициализации весов и "отравление" классов, базирующееся на методе дистилляции.

3 Обзор алгоритма

Рассмотрим на примере датасет MNIST. Обозначим длину и ширину изображений как $W = 28, H = 28$. Возьмем 10 объектов в качестве будущих эталонов, обозначив $M = 10$. Случайным образом заполним пиксели этих 10 эталонных изображений $\tilde{x} = \{x_{i,j,k}\}_{1,1,1}^{M,W,H}$. Вектор признаков для \tilde{x} обозначим $\tilde{y} = [0, 1, 2, 3, \dots, 9]$. Исходные данные обозначим как x, y для признаков и меток соответственно. Веса будем обозначать за w .

Далее будем считать, что метки закодированы с помощью one-hot.

Формально, для начала, нам необходимо минимизировать функцию потерь по признакам

$$\tilde{x} = \operatorname{argmin}_{\tilde{x}} l(x, y, w - \tilde{\eta} \nabla_w l(\tilde{x}, \tilde{y}, w))$$

Поскольку непонятно, какой градиентный шаг брать при оптимизации весов, то его также будем оптимизировать, получим в итоге:

$$\tilde{x}, \tilde{\eta} = \underset{\tilde{x}, \tilde{\eta}}{\operatorname{argmin}} l(x, y, w - \tilde{\eta} \nabla_w l(\tilde{x}, \tilde{y}, w))$$

Для нескольких проходов(эпох) достаточно воспользоваться рекурсией, а для подсчета градиентов методом backpropagation, что будет показано ниже. В дальнейшем N будет обозначать N-ый шаг рекурсии.

$$l(x, y, w_N - \tilde{\eta} \nabla_{w_N} l(\tilde{x}, \tilde{y}, w_N))$$

4 Tensorflow реализация

4.1 Модель

Модель будет иметь следующие методы:

```

1  import tensorflow as tf
2
3  class Model:
4      def __init__(self, size: int, num_classes: int, sigma=0.2, mu=0):
5          pass
6
7      def forward(self, X, weights=None) -> tf.Tensor:
8          pass
9
10     def flat_weights(self, weights: tf.Tensor, biases: tf.Tensor) -> tf.Tensor:
11         pass
12
13     def unflat_weights(self, weights: tf.Tensor) -> (tf.Tensor, tf.Tensor):
14         pass
15
16     def get_weights(self) -> tf.Tensor:
17         pass
18

```

Метод `forward` принимает на вход также веса, то есть, мы можем перестроить граф вычислений модели предоставив новые веса. Методы `flat_weights`, `unflat_weights` для преобразования весов в одномерный массив и для преобразования из одномерного массива весов в многомерные для каждого слоя соответственно.

4.2 Дистиллятор

```
1 class Distillator:
2     def __init__(self, models: [], shape: (int),
3                   num_classes: int, M: int,
4                   batch_size: int, num_epochs: int, distill_epochs: int,
5                   sigma: float, mu: float):
6         pass
7     def forward(self, model) -> (tf.Tensor, [], []):
8         pass
9     def backward(self, forward_output) -> ([], []):
10        pass
11    def optimizer(self, gradients: []):
12        pass
13    def train(self, X, y):
14        pass
15    def get_result(self):
16        pass
17
```

Начнем с конструктора класса:

```
1     def __init__(self, models: [], shape: (int),
2                   num_classes: int, M: int,
3                   batch_size: int, num_epochs: int, distill_epochs: int,
4                   sigma: float, mu: float):
5
6         self.batch_size = batch_size
7         self.num_epochs = num_epochs
8         self.distill_epochs = 3
9         self.models = models
10        self.num_classes = num_classes
11
12        self.learningRateX = tf.constant(1e-2, dtype=tf.float32)
13        self.learningRateW = tf.Variable(1e-2, dtype=tf.float32)
14
15        self.x_real = tf.placeholder(tf.float32, shape=[None, *shape])
16        self.y_real = tf.placeholder(tf.float32, shape=[None, num_classes])
17
18        self.x_distilled = tf.Variable(tf.truncated_normal([M, *shape]
19        ↪ , stddev=sigma, mean=mu))
20        self.y_distilled = np.eye(num_classes)[np.arange(0, num_classes)]
21        self.y_distilled = tf.Variable(self.y_distilled)
```

В конструкторе мы объявляем три переменные: шаг градиента $\tilde{\eta}$, дистиллированные признаки \tilde{x} и их метки \tilde{y} . Пока что будем оптимизировать лишь шаг градиента и признаки. А метки положим равными 0,1...,9 после чего закодируем с помощью one-hot кодирования. Следует уточнить, что в алгоритме у нас будут два прохода - по дистиллированным данным и по реальным данным, отсюда есть необходимость определить значения для эпох каждого цикла.

Также следует заметить, что в инициализациях указаны модели, что подразумевает работу с несколькими моделями. То есть, алгоритм будет проходить по всем моделям, получая градиенты от каждого из них, после чего будем усреднять полученное значение на количество моделей $\frac{1}{models} \sum_{j=1}^{models} \nabla_{\tilde{x}} L(x, y, w_N)$

Где за $\nabla_{\tilde{x}} L_j(x, y, w_N)$ обозначена сумма градиентов, полученных из прохода по каждой модели. Несколько моделей необходимы для того, чтобы дистиллированные данные не зависели от алгоритма, формально, мы берем маотжидание относительно алгоритмов, считая, что дисперсия дистиллированных данных по всем алгоритмам небольшая.

4.3 Train

```

1  def train(self,X,y):
2      sess = tf.Session()
3      self.sess = sess
4      sess.run([self.x_distilled.initializer,
5                self.y_distilled.initializer,
6                self.learningRateWK.initializer])
7
8      self.__weights_initializers = []
9      gradients = []
10     lossSum = 0
11
12     for model in self.models:
13         forwardOutput = self.forward(model)
14         loss,_,_ = forwardOutput
15         lossSum = lossSum+loss
16         gradients.append(self.backward(forwardOutput))
17     self.optimizer(gradients)
18
19     for epoch in np.arange(0,self.numEpochsK):
20         for step_i in np.arange(0,X.shape[0], self.batchSizeK):
21             x_real_batch = X[step_i:step_i + self.batchSizeK,:]
22             y_real_batch = y[step_i:step_i + self.batchSizeK,:]
23
24             sess.run(self.__weights_initializers)
25             sess.run([self.optimize_x,self.optimize_lr],
26                     feed_dict={self.x_real:x_real_batch,
27                               self.y_real:y_real_batch})

```

Сначала мы создаем новую сессию, после чего инициализируем наши три переменные. Далее начинаем проход по всем моделям.

Метод `forward` возвращает значением функции потерь на N-ом шаге $l(x, y, w_N)$, а также список весов w_1, \dots, w_N и их градиентов, умноженных на $\tilde{\eta}$ $\tilde{\eta} \nabla_{w_1} l(\tilde{x}, \tilde{y}, w_1), \dots, \tilde{\eta} \nabla_{w_N} l(\tilde{x}, \tilde{y}, w_N)$.

Все данные значения отправляются в следующий метод `backward`, который возвращает искомые градиенты для $\tilde{\eta}, \tilde{x}$.

Метод `optimizer` делает оптимизацию по полученным градиентам. Всё это статическая часть кода, на этом этапе мы создали весь граф вычислений алгоритма, далее в динамической части мы разбиваем реальную выборку на батчи и проходим несколько эпох по исходным данным. На каждой шаге(step) инициализируем заново случайным образом веса модели, после чего запускаем оптимизацию для $\tilde{\eta}, \tilde{x}$.

4.4 Forward

Данный метод имеет следующую реализацию:

```

1  def forward(self,model):
2      model.forward(self.x_distilled)
3      weights = model.get_weights()
4      (new_weights_list,gradients_list) = [weights],[]
5
6      for epoch in np.arange(0,self.distill_epochs):
7          output = model.forward(self.x_distilled,weights=weights)
8          loss = reduce_mean(softmax_cross_entropy(logits=output,
9                                labels=self.y_distilled))
10         dloss_dw = tf.gradients(loss,weights)
11         dloss_dw = tf.squeeze(dloss_dw)
12         new_weights = weights - self.learningRateW*dloss_dw
13
14         new_weights_list.append(new_weights)
15         gradients_list.append(self.learningRateW*dloss_dw)
16         weights = new_weights
17
18     self.__weights_initializers += model.initializers
19     output = model.forward(self.x_real,weights=weights)
20     loss = reduce_mean(softmax_cross_entropy(logits=output,
21                                               labels=self.y_real))
22     return(loss,new_weights_list,gradients_list)

```

Метод получает на вход конкретную модель, прогоняет дистиллированные данные через эту модель, получает веса, инициализированные случайным образом, которые будут первые в нашей рекурсии w_1 .

В цикле прогоняем дистиллированные данные с новыми весами w_1, \dots, w_{N-1} на каждой итерации, для упрощения, будем для всех моделей брать многоклассовую кроссэнтропию из tensorflow. Далее идет подсчет градиента:

$$\nabla_{w_{N-1}} l(\tilde{x}, \tilde{y}, w_{N-1})$$

После чего делаем новый градиентный шаг:

$$w_N = w_{N-1} - \tilde{\eta} \nabla_{w_{N-1}} l(\tilde{x}, \tilde{y}, w_{N-1})$$

После цикла сохраняем инициализаторы весов, делаем проход модели, но уже на реальных данных, получая значение функции потерь:

$$l(x, y, w_N)$$

Заметим, что w_N имеет зависимость от $\tilde{x}, \tilde{\eta} w_{N-1}$, в обратном проходе необходимо будет проводить дифференцирование по данным тензорам.

4.5 Backward

Данный метод имеет следующую реализацию:

```

1  def backward(self, forward_output):
2      (loss, new_weights_list,
3       gradients_list) = forward_output
4
5      dloss_dw_last, = tf.gradients(loss, new_weights_list[-1])
6      x_distilled_grad = tf.zeros_like(self.x_distilled)
7      lr_grad = 0
8
9      for weights, dloss_dw in reversed(list(zip(new_weights_list, gradients_list))):
10         dloss_dw_last = dloss_dw_last * -1
11         gradients = tf.gradients(dloss_dw,
12                                  [self.x_distilled,
13                                   self.learningRateWK,
14                                   weights],
15                                  grad_ys=dloss_dw_last)
16         (dloss_dx, dloss_dlr, ddloss_ddw) = gradients
17
18         x_distilled_grad += dloss_dx
19         lr_grad += dloss_dlr
20
21         dloss_dw_last = dloss_dw_last + ddloss_ddw
22     return x_distilled_grad, lr_grad

```

Сначала берем дифференцирование по самым последним весам w_N :

$$\frac{\partial l(x, y, w_N)}{\partial w_N}$$

. После чего идем обратно по подсчитанным градиентам и весам w_{N-1}, \dots, w_1 и их градиентов, умноженных на $\tilde{\eta}$

$$\tilde{\eta} \nabla_{w_{N-1}} l(\tilde{x}, \tilde{y}, w_{N-1}), \dots, \tilde{\eta} \nabla_{w_1} l(\tilde{x}, \tilde{y}, w_1)$$

Примечание

В силу того, что в списке весов больше, чем соответствующих градиентов, получим, что проход начнется с шага **N-1**

Если рассмотреть первый проход, то наша цель получить:

$$\begin{cases} \frac{\partial l(x, y, w_N)}{\partial \tilde{x}} = -\tilde{\eta} \frac{\partial l(x, y, w_N)}{\partial w_N} \frac{\partial l(\tilde{x}, \tilde{y}, w_{N-1})}{\partial \tilde{x}, \partial w_{N-1}} \\ \frac{\partial l(x, y, w_N)}{\partial \tilde{\eta}} = -\tilde{\eta} \frac{\partial l(x, y, w_N)}{\partial w_N} \frac{\partial l(\tilde{x}, \tilde{y}, w_{N-1})}{\partial \tilde{\eta}, \partial w_{N-1}} \end{cases}$$

Отсюда ясно, почему мы производим смену знака у $\tilde{\eta} \nabla_{w_N} l(\tilde{x}, \tilde{y}, w_N)$.

Далее берем градиенты:

- $$\tilde{\eta} \frac{\partial l(\tilde{x}, \tilde{y}, w_{N-1})}{\partial \tilde{x}, \partial w_{N-1}}$$
- $$\tilde{\eta} \frac{\partial l(\tilde{x}, \tilde{y}, w_{N-1})}{\partial \tilde{\eta}, \partial w_{N-1}}$$
- $$\tilde{\eta} \frac{\partial l(\tilde{x}, \tilde{y}, w_{N-1})}{\partial w_{N-1}, \partial w_{N-1}}$$

И умножаем их на $-\frac{\partial l(x, y, w_N)}{\partial w_N}$ по правилам дифференцирования сложной функции, как показано в системе выше.

Полученные градиенты для искоемых переменных аккумулируем, а градиент для весов используем для обновления $-\frac{\partial l(x, y, w_N)}{\partial w_N}$ и подготовки к следующему шагу.

Наконец, возвращаем градиенты искоемых переменных.

4.6 Optimizer

Данный метод имеет следующую простейшую реализацию:

```
1 def optimizer(self, gradients):
2     x_grad_sum = tf.zeros_like(self.x_distilled)
3     lr_grad_sum = 0
4     for x_grad, lr_grad in gradients:
5         x_grad_sum = x_grad_sum + x_grad
6         lr_grad_sum = lr_grad_sum + lr_grad
7     x_grad = x_grad_sum / len(self.models)
8     lr_grad = lr_grad_sum / len(self.models)
9     self.optimize_x = tf.assign_sub(self.x_distilled,
10                                     self.learningRateX * x_grad)
11     self.optimize_lr = tf.assign_sub(self.learningRateW,
12                                     self.learningRateX * lr_grad)
```

Проходимся по всем градиентам, суммируя их, далее делим на количество моделей, как было описано выше и обновляем искоемые переменных $\tilde{x}, \tilde{\eta}$.

Последний метод служит простым вычислением данных через сессию.

5 Однослойная нейросеть

Для начала возьмем вместо многослойной нейросети однослойную - логистическую регрессию. Как говорилось выше, рассмотрим датасет MNIST. Прогоним данные через наш дистиллятор и после примерно 10 секунд получим, что данные аппроксимируются визуально верно:

Если посмотреть на $\tilde{x} : \tilde{y} = 3$, то получим:

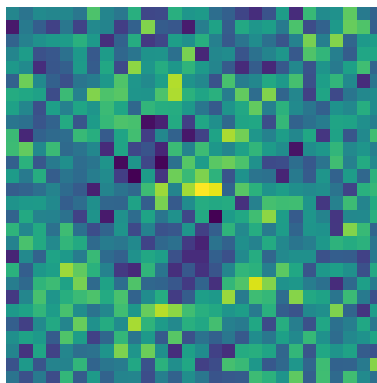


Рис. 1: $\tilde{x} : \tilde{y} = 3$

В таблице представлены зависимости времени и значение функции потерь: