Дистиллирование данных

Марков В.О.
МГУ ВМК

April 10, 2020

1 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

Рассмотрим несоклько видов инициализации весов:

1. На каждой эпохе будем брать случайным оборазом инициализированные веса
2. Будем работать с фиксированными чатсично предобученными весами

Также рассмотрим "отравление" классов, базирующееся на методе дистилляции.

2 Обзор алгоритма

Обозначим длину и ширину изображений как W, H. Возьмем M объектов в качестве будущих эталонов. Случайным образом заполним пиксели этих M эталонных изображений $\tilde{x} = \{x_{i,j,k}\}_{1,1,1}^{M,W,H}$. Вектор признаков для $\tilde{x}$ обозначим $\tilde{y} = [0, 1, 2, 3, ..., K]$. 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

$$\tilde{x} = \underset{\tilde{x}}{\operatorname{argmin}}\, l(x, y, w - \tilde{\eta}\, \nabla_w l(\tilde{x}, \tilde{y}, w)) \tag{1}$$

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

$$\tilde{x}, \tilde{\eta} = \underset{\tilde{x}, \tilde{\eta}}{\operatorname{argmin}}\, l(x, y, w - \tilde{\eta}\, \nabla_w l(\tilde{x}, \tilde{y}, w)) \tag{2}$$

В дальнейшем рассмотрим минимизацию не только по $\tilde{x}, \tilde{\eta}$, но и по метка $\tilde{y}$:

$$\tilde{x}, \tilde{\eta}, \tilde{y} = \underset{\tilde{x},\tilde{\eta},\tilde{y}}{\operatorname{argmin}} \, l(x, y, w - \tilde{\eta} \nabla_w l(\tilde{x}, \tilde{y}, w)) \tag{3}$$

Веса изначально будем инициализировать случайным образом на каждой эпохе.

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

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

.

3 Tensorflow реализация

3.1 Модель

Модель будет иметь следующие методы:

```
import tensorflow as tf

class Model:
        def __init__(self,size:int,num_classes:int,sigma=0.2,mu=0):
                pass

        def forward(self,X, weights=None)->tf.Tensor:
                pass

        def flat_weights(self,weights:tf.Tensor,biases:tf.Tensor)->tf.Tensor:
                pass

        def unflat_weights(self,weights:tf.Tensor)->(tf.Tensor,tf.Tensor):
                pass

        def get_weights(self)->tf.Tensor:
                pass

```

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 в массива весов в многомерные для каждого слоя соотвественно.

3.2 Дистиллятор

```
class Distillator:
        def __init__(self,models:[],shape:(int),
                        num_classes:int,M:int,
                        batch_size:int,num_epochs:int,distill_epochs:int,
                        sigma:float,mu:float):
                pass
        def forward(self,model)->(tf.Tensor,[],[]):
                pass
        def backward(self,forward_output)->([],[]):
                pass
        def optimizer(self,gradients:[]):
                pass
        def train(self,X,y):
                pass
        def  get_result(self):
                pass

```

Начнем с конструктора класса:

```
        def __init__(self,models:[],shape:(int),
                                num_classes:int,M:int,
                                batch_size:int,num_epochs:int,distill_epochs:int,
                                sigma:float,mu:float):
        self.batch_size = batch_size
        self.num_epochs = num_epochs
        self.distill_epochs = distill_epochs
        self.models = models
        self.num_classes = num_classes



        self.learningRateX = tf.constant(1e-2,dtype=tf.float32)
        self.learningRateW = tf.Variable(1e-2,dtype=tf.float32)

        self.x_real = tf.placeholder(tf.float32, shape=[None, *shape])
        self.y_real = tf.placeholder(tf.float32, shape=[None, num_classes])

        self.x_distilled = tf.Variable(tf.truncated_normal([M, *shape]
        ↪  ,stddev=sigma,mean=mu))
        self.y_distilled = np.eye(num_classes)[np.arange(0,num_classes)]
        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 дистилированные данные не зависили от алгоритма, формально, мы берем маотжидание относительно алгоритмов, считая, что дисперсия дистилированных данных по всем алгоритмам небольшая.

3.3 Train

```
    def train(self,X,y):
                sess = tf.Session()
                self.sess = sess
                sess.run([self.x_distilled.initializer,
                        self.y_distilled.initializer,
                        self.learningRateWK.initializer])

                self.__weights_initializers = []
                gradients = []
                lossSum = 0

                for model in self.models:
                        forwardOutput = self.forward(model)
                        loss,_,_  = forwardOutput
                        lossSum = lossSum+loss
                        gradients.append(self.backward(forwardOutput))
                self.optimizer(gradients)

                for epoch in np.arange(0,self.numEpochsK):
                        for step_i in np.arange(0,X.shape[0], self.batchSizeK):
                                x_real_batch = X[step_i:step_i + self.batchSizeK,:]
                                y_real_batch = y[step_i:step_i + self.batchSizeK,:]

                                sess.run(self.__weights_initializers)
                                sess.run([self.optimize_x,self.optimize_lr],
                                    feed_dict={self.x_real:x_real_batch,
                                                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 списком весов $w_1, ..., w_N$ и их градиентов, умноженных на $-\tilde{\eta}$: -$\tilde{\eta}\,\nabla_{w_1} l(\tilde{x}, \tilde{y}, w_1), ..., -\tilde{\eta}\,\nabla_{w_N} l(\tilde{x}, \tilde{y}, w_N)$.

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 бачи и проходим несоклько эпох по исходным данным. На каждой шаге(step) инициализируем заново случайным образом веса модели, после чего запускаем оптимизацию для $\tilde{\eta}, \tilde{x}$.

3.4 Forward

Данный метод имеет следующую реализацию:

```
def forward(self,model):
      model.forward(self.x_distilled)
      weights = model.get_weights()
      (new_weights_list,gradients_list) = [weights],[]

      for epoch in np.arange(0,self.distill_epochs):
              output = model.forward(self.x_distilled,weights=weights)
              loss = reduce_mean(softmax_cross_entropy(logits=output,
                              labels=self.y_distilled))
              dloss_dw = tf.gradients(loss,weights)
              dloss_dw = tf.squeeze(dloss_dw)
              new_weights = weights - self.learningRateW*dloss_dw

              new_weights_list.append(new_weights)
              gradients_list.append(-self.learningRateW*dloss_dw)
              weights = new_weights

      self.__weights_initializers += model.initializers
      output = model.forward(self.x_real,weights=weights)
      loss = reduce_mean(softmax_cross_entropy(logits=output,
                              labels=self.y_real))
      return(loss,new_weights_list,gradients_list)
```

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

В цикле прогоняем дистиллированные данные с новыми весами $w_1, ..., w_{N-1}$ 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:

$$l(x, y, w_N) \tag{4}$$

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

3.5 Backward

Данный метод имеет следующую реализацию:

```
def backward(self,forward_output):
        (loss,new_weights_list,
                gradients_list)  =  forward_output

        dloss_dw_last, = tf.gradients(loss,new_weights_list[-1])
        x_distilled_grad = tf.zeros_like(self.x_distilled)
        lr_grad = 0

        for weights,dloss_dw in reversed(list(zip(new_weights_list,gradients_list))):
                gradients = tf.gradients(dloss_dw,
                                [self.x_distilled,
                                self.learningRateWK,
                                weights],
                                grad_ys=dloss_dw_last)
                (dloss_dx,dloss_dlr,ddloss_ddw) = gradients

                x_distilled_grad += dloss_dx
                lr_grad += dloss_dlr

                dloss_dw_last = dloss_dw_last+ddloss_ddw
        return x_distilled_grad,lr_grad
```

Сначала берем дифференцирование (4) по самомым последним весам w_N:

$$\frac{\partial l(x, y, w_N)}{\partial w_N} \tag{5}$$

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

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

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:

$$\begin{cases} \frac{\partial l(x,y,w_N)}{\partial \tilde{x}} = -\tilde{\eta}\frac{\partial l(x,y,w_N)}{\partial w_N}\frac{\partial^2 l(\tilde{x},\tilde{y},w_{N-1})}{\partial \tilde{x},\partial w_{N-1}} \\ \frac{\partial l(x,y,w_N)}{\partial \tilde{\eta}} = -\tilde{\eta}\frac{\partial l(x,y,w_N)}{\partial w_N}\frac{\partial^2 l(\tilde{x},\tilde{y},w_{N-1})}{\partial \tilde{\eta},\partial w_{N-1}} \end{cases} \tag{6}$$

Далее берем градиенты:

$$\nabla_{\tilde{x}}\left[-\tilde{\eta}\frac{\partial l(\tilde{x},\tilde{y},w_{N-1})}{\partial w_{N-1}}\right] \tag{7}$$

$$\nabla_{\tilde{\eta}}\left[-\tilde{\eta}\frac{\partial l(\tilde{x},\tilde{y},w_{N-1})}{\partial w_{N-1}}\right] \tag{8}$$

$$\nabla_{w_{N-1}}\left[-\tilde{\eta}\frac{\partial l(\tilde{x},\tilde{y},w_{N-1})}{\partial w_{N-1}}\right] \tag{9}$$

Полученные градиенты для искомых переменных аккумулируем, а градиент для весов используем для обновления (5) и подготовки ко второму шагу. Последующие шаги проводятся аналогично.

Наконец, возвращаем градиенты искомых переменных.

3.6 Optimizer

Данный метод имеет следующую простейшую реализацию:

```
    def optimizer(self,gradients):
                x_grad_sum = tf.zeros_like(self.x_distilled)
                lr_grad_sum = 0
                for x_grad,lr_grad in gradients:
                        x_grad_sum = x_grad_sum+x_grad
                        lr_grad_sum= lr_grad_sum+lr_grad
                x_grad = x_grad_sum/len(self.models)
                lr_grad = lr_grad_sum/len(self.models)
                self.optimize_x = tf.assign_sub(self.x_distilled,
                                self.learningRateX*x_grad)
                self.optimize_lr = tf.assign_sub(self.learningRateW,
                                self.learningRateX*lr_grad)
```

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

Последний метод служит простым вычислением данных через сессию.

4 Однослойная нейросеть

Для начала возьмем вместо многослойной нейросети однослойную - логистическую регрессию. Рассмотрим датасет MNIST, $W = H = 28$, $\tilde{y} = [0, 1, 2, 3, ..., 9]$. Прогоним данные через наш дистиллятор и после примерно 10 секунд получим, что данные аппроксимируются визуально верно:

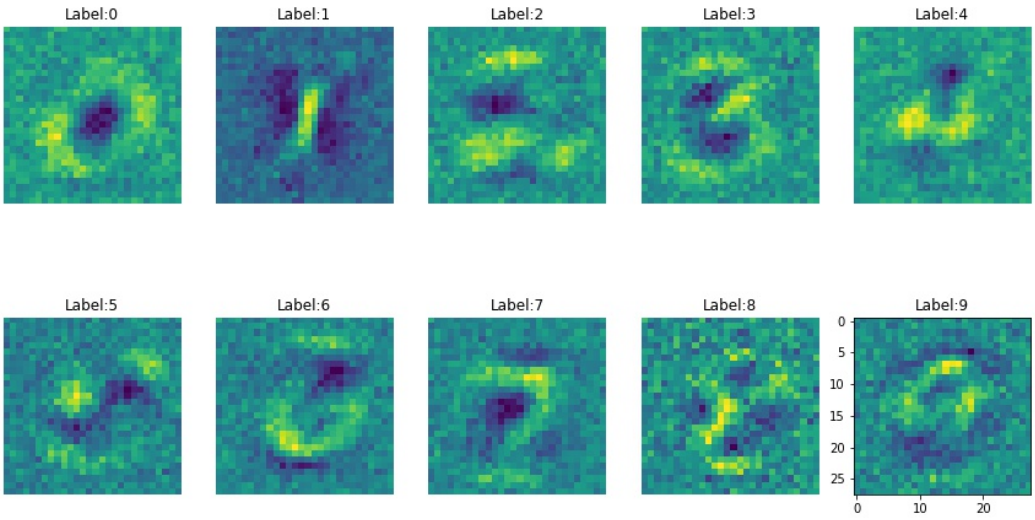

Figure 1: Визуализация $\tilde{x}$

Посомтрим на зависимость значений функции потерь от эпох:

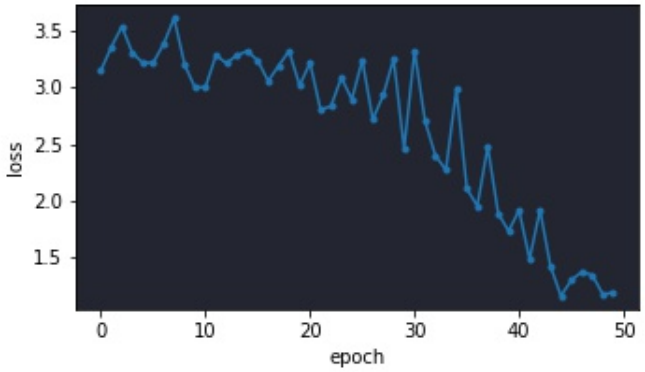

Figure 2: Зависимость значений функции потерь от эпох