Дистиллирование данных

Марков В.О.
МГУ ВМК

April 13, 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 согласно некоторому распределению $p(w|\sigma, \mu)$
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. В общем случае будем говорить, что имеем дело с объектами из $\mathbb{R}^D$. 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.

Наша цель основная цель - получить M синтезированных объектов из заданного датасета, которые будут минимизировать некоторую функцию потерь, причем эти объекты в идеальном случае не зависят от весов и модели. Для оптимизации по весам и данным будем использовать градиентный спуск.Сначала сделаем один градиентный шаг по весам $w_1 = w - \tilde{\eta}\,\nabla_w l(\tilde{x}, \tilde{y}, w)$, после чего w_1 подставим в функцию потерь посчитанную на реальных данных $l(x, y, w_1)$. Эта функция теперь неявно зависит от $\tilde{x}$, поэтому мы можем минимизировать её значение по $\tilde{x}$:

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

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 $\tilde{y}$:

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

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.

$$\tilde{x}, \tilde{\eta} = \underset{\tilde{x},\tilde{\eta}}{\operatorname{argmin}}\, l(x, y, \underbrace{w_{N-1}} - \tilde{\eta}\, \nabla_w l(\tilde{x}, \tilde{y}, \underbrace{w_{N-1}})) \tag{4}$$

Распишем градиент, к примеру, для переменной $\tilde{x}$ поподробнее:

$$\frac{d\; l(x, y, w_N(\tilde{x}, \tilde{y}, w_{N-1}))}{d\; \tilde{x}} = \underbrace{\frac{d\; l(x, y, w_N)}{d\; w_N} \frac{\partial w_N}{\partial \tilde{x}}}_{\text{Первый шаг}} + \underbrace{\frac{d\; l(x, y, w_{N-1})}{d\; w_{N-1}} \frac{\partial w_{N-1}}{\partial \tilde{x}}}_{\text{Второй шаг}} + ... + \underbrace{\frac{d\; l(x, y, w_1)}{d\; w_1} \frac{\partial w_1}{\partial \tilde{x}}}_{\text{N-ый шаг}} \tag{5}$$

Мы разбили поиск искомого градиента на N шагов. Это делается для того, чтобы частично самим реализовать метод backpropagation, в следующих разделах станет более понятно почему мы можем оптимизировать данный метод, вместо того, чтобы дать tensorflow или pytorch самостоятельно реализовать обратный проход.

Стоит уточнить, что символом d обозначается полная частная производная, а символом ∂ частная производная.

Заметим, также, что для любого N справедливо:

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

Если подставить это в уравнение (5), то получим готовую формулу для оптимизированного метода backpropagation, чем воспользуемся ниже.

И заметим, что пока непонятно, какое брать распределение весов и каким брать параметр M.

В оригинальной статье по дистилляции данных вначале рассматривают простейший случай с однослойной сетью со среднеквадратичной функцией ошибки, где легко выводят, что значение M должно удовлетворять неравенству $M \geq D$, в таком случае, по крайней мере, для данной простой модели наши дистилированные данные не зависят от весов. Для более сложных моделей мы можем полагать, что должно выполняться аналогичное ограничение и понятно, что хотелось бы брать значение M минимально возможным.

3 Tensorflow реализация

3.1 Модель

Модель будет иметь следующие методы:

```python
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

```python
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, это еще один подход к избавлению зависимости дистилированных данных от весов. 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 списокк 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 = model.loss(logits=output,labels=self.y_distilled)
             dloss_dw = tf.gradients(loss,weights)
             dloss_dw = tf.squeeze(dloss_dw)
             new_weights = tf.stop_gradient(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 = model.loss(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, инициализированные 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:

$$l(x, y, w_N) \tag{7}$$

Останавливаем градиент

В 12 строчке мы останавливаем градиент, чтобы веса w_{N+1} не зависили от $w_N, \tilde{x}, \tilde{\eta}$. Это сделано для того, чтобы в методе backward мы самостоятельно проделали обратный проход для оптимизации, в силу того, что мы явно знаем зависимость w_{N+1} от указанных переменных и имеем возможность явно выписать соответствующие градиенты.

3.5 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                 gradients = tf.gradients(dloss_dw,
11                                 [self.x_distilled,
12                                 self.learningRateWK,
13                                 weights],
14                                 grad_ys=dloss_dw_last)
15                 (dloss_dx,dloss_dlr,ddloss_ddw) = gradients
16
17                 x_distilled_grad += dloss_dx
18                 lr_grad += dloss_dlr
19
20                 dloss_dw_last = dloss_dw_last+ddloss_ddw
21         return x_distilled_grad,lr_grad
```

Сначала берем дифференцирование (7) по самым последним весам w_N:

$$\frac{\partial l(x, y, w_N)}{\partial w_N} \tag{8}$$

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

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

Здесь нужно вспомнить уравнение (5) , в котором и подчеркнут каждый шаг в backward.

Примечание!

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

Если рассмотреть первый шаг, то наша цель получить:

$$\begin{cases} \underbrace{\frac{d\ l(x,y,w_N)}{d\ w_N}\frac{\partial w_N}{\partial \tilde{x}}}_{\text{Первый шаг}} = \overbrace{-\tilde{\eta}\frac{\partial l(x,y,w_N)}{\partial w_N}}^{\text{См. 5 и 20 строчки}} \quad \overbrace{\frac{\partial^2 l(\tilde{x},\tilde{y},w_{N-1})}{\partial \tilde{x},\partial w_{N-1}}}^{\text{Получаем в 10-14 строках кода}} \\ \underbrace{\frac{d\ l(x,y,w_N)}{d\ w_N}\frac{\partial w_N}{\partial \tilde{\eta}}}_{\text{Первый шаг}} = -\tilde{\eta}\frac{\partial l(x,y,w_N)}{\partial w_N} \overbrace{\frac{\partial^2 l(\tilde{x},\tilde{y},w_{N-1})}{\partial \tilde{\eta},\partial w_{N-1}}}^{\text{Аналогично}} \end{cases} \tag{9}$$

Полученные градиенты для искомых переменных аккумулируем, а градиент для весов используем для обновления (8) и подготовки ко второму шагу. Последующие шаги проводятся аналогично.

Почему мы аккумулируем градиенты?

В уравнении (5) расписаны все шаги по правилам дифференцирования, на каждом шаге мы поулчаем очередное слагаемое, учитывая, что отключили градиент у всех весов. Тем самым, мы не выделяем память под данные, а лишь обновляем значения искомых градиентов.

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

4.1 Random initialization

Для начала возьмем вместо многослойной нейросети однослойную - логистическую регрессию. Рассмотрим датасет MNIST, $W = H = 28$, $\tilde{y} = [0, 1, 2, 3, ..., 9]$. Также возьмем нормальное распределение весов с параметры $\sigma =$ **0.2,** $\mu =$ **0** .

Прогоним данные через наш дистиллятор и после примерно 10 секунд получим, что данные аппроксимируются визуально верно:

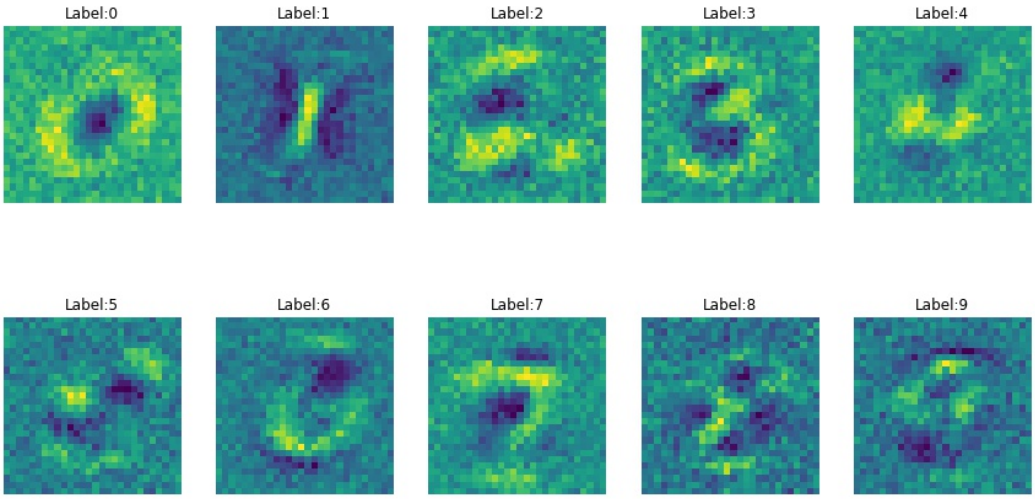

Figure 1: Визуализация $\tilde{x}$

Посомтрим на зависимость значений функции потерь от эпох:

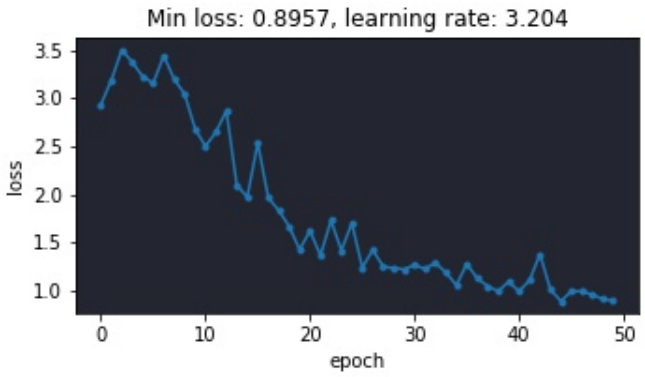

Figure 2: Зависимость значений функции потерь от эпох

Попробуем увеличить дисперсию у весов, положив $\sigma = \mathbf{0.7}$, $\mu = \mathbf{0}$

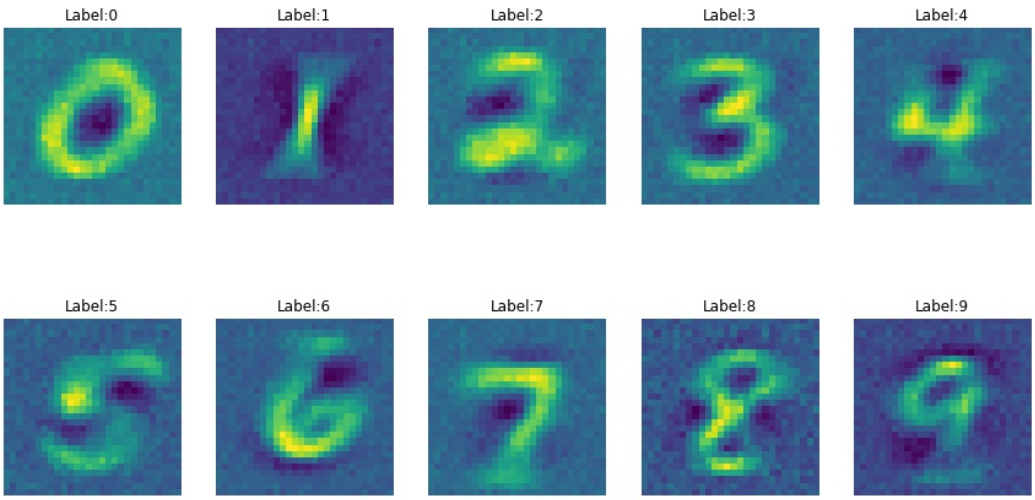

Figure 3: Визуализация $\tilde{x}$

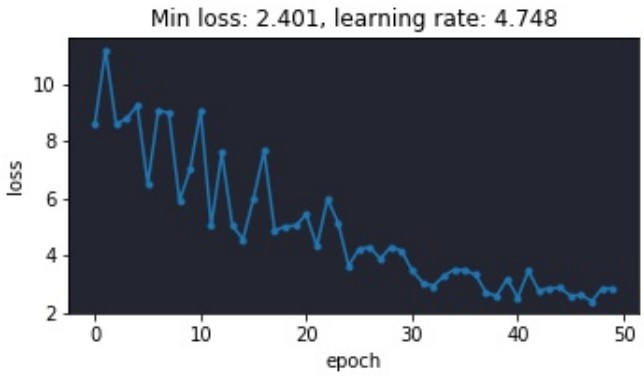

Figure 4: Зависимость значений функции потерь от эпох

Вероятно, из-за того, что веса генерируются слишком большими, функция потерь также имеет относительно большие значения, в то время как $\tilde{x}, \tilde{\eta}$ "сильнее" стремятся минимизировать потери.

Возьмем Xavier инициализацию, тогда получим следующий график:

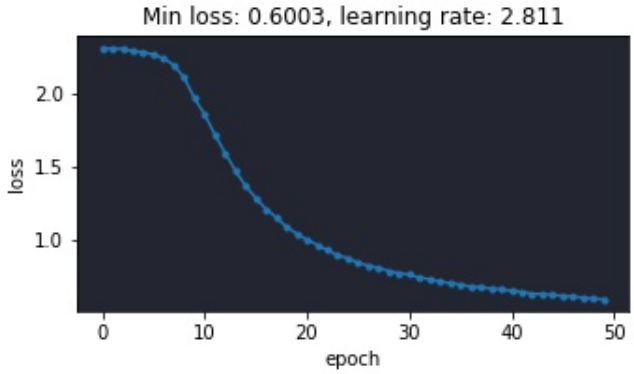

Figure 5: Зависимость значений функции потерь от эпох

Поставив 10 эпох на дистилляцию, получим максимальную точность на тестовом наборе **88.5%**:

```
model = Model(num_classes=10)

output = model.forward(x_distilled)
loss = reduce_mean(softmax_cross_entropy(logits=output,labels=y_distilled))
optimizer_w =
    tf.train.GradientDescentOptimizer(learning_rate=learningRateW).minimize(loss)

sess.run(model.initializers)
for i in range(10):
        sess.run(optimizer_w)
out = model.forward(X_test)
y_pred = tf.argmax(tf.nn.softmax(out),axis=1)
```

5 Простейшая сверточная сеть

5.1 Random initialization

Возьмем теперь в качестве модели(класс Model) данную сверточную нейросеть:

```
conv1 = tf.nn.conv2d(x, self.weights[0], strides=[1, 1, 1, 1], padding='VALID')
conv1 = tf.nn.bias_add(conv1, self.biases[0])
conv1 = tf.nn.tanh(conv1)
conv1 = tf.nn.avg_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

flatten = tf.flatten(conv1)
dense1 = tf.add(tf.matmul(flatten, self.weights[1]), self.biases[1])
dense1 = tf.nn.tanh(dense1)

out = tf.add(tf.matmul(dense1, self.weights[2]), self.biases[2])
```

Проблемы с $\tilde{\eta}$

Я заметил, что при инициализации весов с большим значением σ в распределении можно увидеть, что значение функции потерь уходит в бесконечность и становится неопределенным.
Из экспериментов удалось выяснить - эта проблема связана с тем, что с увеличениием количества весов(а в данной сверточной сети значительно больше весов относительной логистической регрессии) и достаточно большого разброса, поулчаем большой градиент у $\tilde{\eta}$, то есть

$$\frac{d\ l(x, y, w_N)}{d\ \tilde{\eta}} \partial\tilde{\eta}\ >>\ 1$$

В свою очередь, отсюда следует, что при итерации по весам, особенно с несколькими эпохами дистилляции, веса увеличиваются, с увеличением весов градиент $\tilde{\eta}$ становится еще больше и так далее. В связи с этим я ограничил изменение $\tilde{\eta}$ в местоде **optimizer**.

```
self.minLrW = tf.constant(1e-5,dtype=tf.float64)
self.maxLrW = tf.constant(1.,dtype=tf.float64)
new_lr = self.learningRateW-self.learningRateX*lr_grad
new_lr = tf.math.minimum(self.maxLrW,new_lr)
new_lr = tf.math.maximum(self.minLrW,new_lr)
self.optimize_lr = tf.assign(self.learningRateW,new_lr)
```

При инициализации Xavier обучение будет достаточно долгим из-за маленьких градиентов, поэтому возьмем нормальное распределение весов с параметрами σ= **0.05,** μ= **0** . Количество дистиллирующих эпох поставим равным **10**. Результаты:

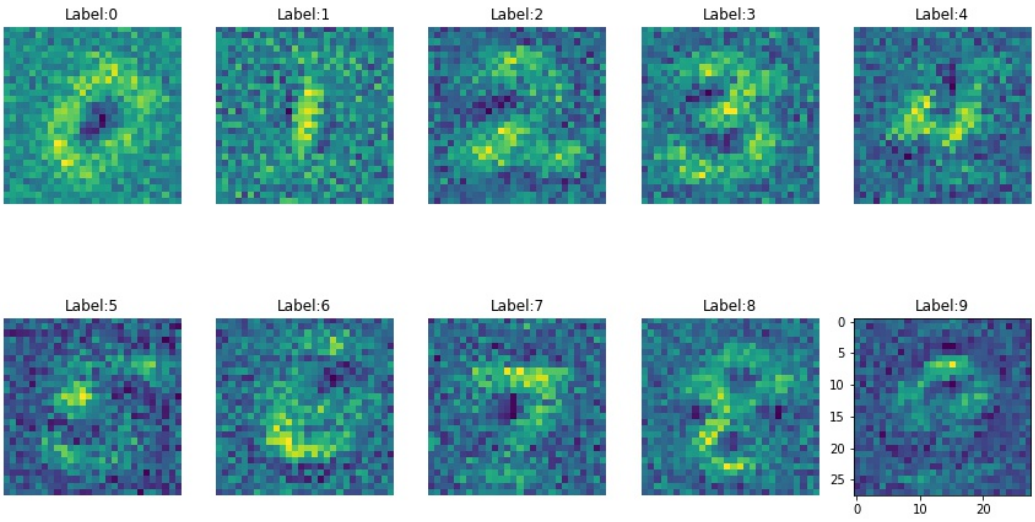

Figure 6: Визуализация $\tilde{x}$

Посомтрим на зависимость значений функции потерь от эпох:

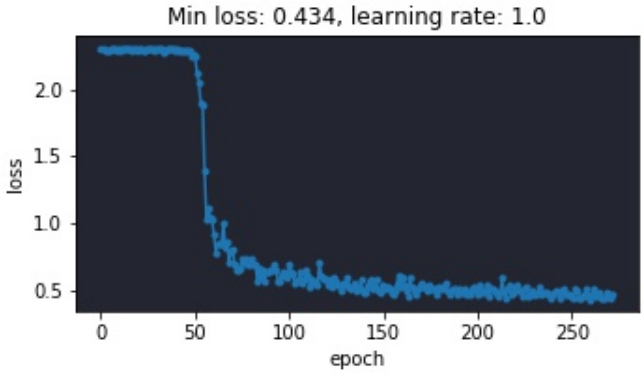

Figure 7: Зависимость значений функции потерь от эпох

На тесте получим точность **86** $\pm$**1.6** %. Очеивдно, чем больше весов(параметров модели), тем менее статичным будет дистллятор при случайной инициализации этих весов. У авторов в оригинальной статье взята сеть LeNet, которая дает качество порядка **80** $\pm$**6** %.