**PyTorch**

|  |  |
| --- | --- |
| **Функция** | **Объяснение** |
| x = torch.empty(3, 5) | создать неинициализированный тензор |
| x = torch.rand(3, 5) | создать инициализированный рандомно тензор |
| x = torch.zeros((3.5), dtype=torch.long) | тензор нулей |
| x = torch.tensor( [1., 2., 3.] ) | создать тензор с известными значениями |
| x = x.new\_ones( (3, 5), dtype=…)  y = torch.randm\_like(x, dtype=…) | повторное использование |
| x.size() | аналогично arr.shape |
| **res** = x+y <==> torch.add(x, y, out=**res**) | поэлеметное сложение |
| y.add\_(x) | к тензору y прибавить элементы тензора x |
| x.t\_() | транспонирование x (меняется исходный тензор) |
| Любые операции, заканч. \_ , изменяют исходный тензор | |
| операции типа x[:, 5:, 0] в торче аналогичны Numpy | |
| y = x.view(new\_size) | аналогично arr.reshape |
| torch.argmax(tensor, dim=1, keepdims=True/False) | argmax |
| x.item() | если тензор имеет один элемент, получить его |
| self.conv1.**weight.data()** | вернуть веса данного слоя |
| self.conv1.weight.**grad**.data() | вернуть градиент для весов данного слоя |
| Другие подобные операции: https://pytorch.org/docs/torch | |
| y = x.numpy() | из тензора в Numpy ndarray |
| x = torch.from\_numpy(y) | из Numpy ndarray в тензор |
| только если тензор на ЦПУ, и изменение одного приведет к изменению другого | |
| if torch.cuda.is\_available():  device = torch.device(“cuda”)  x = torch.ones\_like(y, device=device)  y = y.to(device) | если ГПУ активировано  узнать устройство  создать тензор x, как y, только на ГПУ  перенести y на ГПУ |
| tensor.device | устройство расположения тензора |
| next(model.parameters()).device | Устройство расположения модели |
| device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")  model.to(device) | автоматический выбор устройства |
| torch.cuda.memory\_allocated() | Занимаемая память (в байтах) на GPU на данный момент |
| torch.cuda.get\_device\_properties('cuda:0').total\_memory | Полный объем памяти данного устройства |
| torch.cuda.max\_memory\_allocated() | Макс. занимаемая память (в байтах) на GPU с момента запуска скрипта |
| torch.cuda.reset\_max\_memory\_allocated() | Сбросить макс. значение |
| self.conv = nn.Conv2d(channels\_in, channels\_out, kernel) | инициализация сверточного слоя |
| self.lin1 = nn.Linear(vector\_in, vector\_out) | инициализация dense слоя |
| nn.Dropout(p=0.2) | dropout |
| F.one\_hot() | to categorical |
| F.max\_pool2d(F.relu(self.conv1(x)), (2, 2)) | пример последовательной обработки: за сверточным слоем следует ReLU, затем вычисляется MaxPooling |
| torch.LayerNorm(dim) | Нормализация значений в тензоре. dim — по каким измерениям нормализовать |
| tensor.size() | arr.shape |
| size = x.size()**[1:]** | можем узнать размерность на выходе слоя **(не считая размер батча)** |
| print(tensor) | выведется содержимое тензора и устройств |
| Torch.eye(N) | Получить единичную матрицу N\*N |
| criterion = nn.MSELoss()  loss = criterion(predict, true) # *tensor* | вычисление лосса |
| learning\_rate = 0.01  for f in net.parameters():  f.data.sub\_(f.grad.data \* learning\_rate) | в общем случае, веса обновляются вот так |
| # *create your optimizer*  optimizer = optim.SGD(net.parameters(), lr=0.01)  for …  optimizer.zero\_grad() # *zero the gradient*  output = net(input)  loss = criterion(output, target)  loss.backward()  optimizer.step() # *Does the update* | net.paramtres() – список обучаемых параметров  выбираем оптимизатор from torch.optim as optim  для каждого шага обучения:  1)обнуляем градиент!  2)вычисляем pred для данного input  3)вычисляем loss  4)обр распр-е ошибки - градиентный спуск  5)оптимизатор обновляет веса |
| loss.backward() | вычисляет производную ошибки по каждому **параметру модели x**, который имеет *requires\_grad=True*. И записывает значения в **x**.grad. Это просто подсчет градиентов. Он не обновляет веса |
| optimizer.step() | обновляет веса, используя x.grad **x**\_new = **x** - lr \* x.grad |
| optimizer.zero\_grad() | **x**.grad = 0 для каждого параметра модели  Важно вызывать ***до loss.backward*** |
| **weight\_decay** - еще один параметр у optimizer (1e-4) | это аналогия Regularization Loss - штраф за слишком большое значение весов: loss = loss + **weight\_decay** \* L2norm\_weights |
| scheduler = StepLR(optimizer, step\_size=500, gamma=0.1, last\_eposh=-1)  scheduler.step() # *счетчик шагов* | каждые 500 шагов будет понижать learning rate в 10 раз. Можно поставить ограничение last\_epoch, после которой не будет изменений lr |
| loss.backward() **clip\_grad\_norm\_**(model.parametres(), **max\_norm=0.25**)  optimizer.step()  *Очень важный параметр* | Помимо исчезновения (vanishing) градиента возникает также взрыв (explode) градиента. Он усиливается от слоя к слою. **clip\_grad\_norm** нормирует градиент всех весов (либо конкретных), предотвращая слишком большое изменение. Другими словами, чтобы все время выполнять небольшие обновления весов, уменьшаем скорости обучения только для больших градиентов. |
| for p in model.parameters():  p.data.add\_(p.grad.data, alpha=-lr) | вместо optimizer.step() веса можно обновлять так. Если лосс не падает, то возможно надо увеличить max\_norm |
| x = some\_tensor # *req\_grad=True*  with torch.no\_grad():  y = x \*\* 2 # *req\_grad=False* | все переменные и/или слои в данном блоке будут не обучаемы. Даже если входные тензоры (х) были req\_grad=True. Это уменьшает расход памяти и времени. Это рекомендуется для прогона тестовых батчей, тогда как eval() для единичный предсказаний |
| model.eval() | для предсказаний (Где нет батчей). Он лишь отключает слои dropout, batch\_norm. |
| nn.ReLU(**inplace**=True) | **inplace**=True, перезаписывает переменную в графе, которая была до слоя ReLU. То есть отдельный узел активации не создается |
| act = nn.Softmax(dim=1)  probs = act(logits) | logits = tensor (Batch\_sz, n\_classes)  probs => tensor (Batch\_sz, n\_classes) |
| loss = torch.nn.CrossEntropyLoss()  loss\_value = loss(logits, label) | **CrossEntropy принимает НЕ выходы softmax**, а именно logits.  label - одно число - номер истинного класса |
| some\_tensor.**data** | Поскольку любой тензор хранит в себе также операции, мы можем извлечь именно **данные** |
| some\_tensor.data.**cpu()** | Перенести значение на **ЦПУ** |
| torchvision.datasets.ImageFolder(.) | Интересный метод, погугли |
| with torch.set\_grad\_enabled(**Training**): | Если **Training**, то вычисления будут сохранять в себе граф, который нужен для оптимизатора. Иначе только значения |
| torch.save(model.state\_dict(), PATH) | Сохранение модели. Предпочтительно брать именно state\_dict().  save(model) сохраняет в себе все пути, имена классов и др. зависимости. При несоответствии чего-либо будет ошибка |
| DataLoader.num\_workers | Если **0**, значит главный процесс будет извлекать батч. Если **n**, значит **n** других процессов будут извлекать и обрабатывать каждый свой батч. Перемешиваются они так же независимо. Если данных не так много, лучше использовать **n**=1. При использовании **n**>1, рекомендуется pin\_memory=True |
| DataLoader.pin\_memory | Если мы загружаем и обрабатываем данные на ЦПУ, затем переводим их на ГПУ, то это займет меньше времени, если pin\_memory=True. Как я понял, запоминает расположение тензоров на ГПУ. Эффекта не будет, если данных мало или каждый батч разного размера или и батчи уже на ГПУ. |
| **writer** = SummaryWriter(log\_dir=“logs“, „comment“, purge\_step=None) | Для логов в TensorBoard. Будет так: log\_dir/  Если log\_dir=None, то runs/datetime\_comment/  purge\_step — с какого step продолжить логгирование |
| Чтобы несколько кривых отображались в одном графике writer\_train = SummaryWriter(log\_dir=“logs/train“)  writer\_test = SummaryWriter(log\_dir=“logs/test“)  А сами **title** у графиков были **одинаковы** | |
| https://pytorch.org/docs/master/tensorboard.html | |
| writer.add\_image(„image“, …) |  |
| writer.add\_graph(model, input) | Добавить граф и input Tensor (or ls of Tensors) |
| writer.add\_scalar(„**title**“, value, step) | Добавить скаляр |
| Writer.add\_scalars("**title**", **dict\_**)  **dict\_** = {"sin": np.sin(step),  "cos": np.cos(step)} | Можно добавить несколько кривых на график |
| Writer.close() |  |

*Autograph* – это важный механизм, который проводит автоматическое дифф-ние над тензорами. То есть он двигает обучение.

Применив к тензору .frad\_fn() можно проследить его вычисления.

Если мы хотим создать переменные и тензоры без необходимости дифф-ния (необучаемые переменные), мы укываем в параметре *requires\_grad=False* или заносим в блок *with torch.np\_grad():*

*Class MyNet(torch.nn.Module):*

*def \_\_init\_\_(self):*

*super(MyNet, self).\_\_init\_\_()*

*self.conv = torch.nn.Conv2d(in\_ch, out\_ch, kernel=, padding=, stride=)*

*self.fc = torch.nn.Linear(in, out)*

*self.act = torch.nn.Tanh()*

*def forward(x):*

*x = self.conv(x)*

*x = self.fc(x)*

*return self.act(x)*

*def forward(input) –* для вычисления ответа сети (для предсказания или дальнейшего измерения ошибки). Тут определяются любые операции над тензорами, а не только стандартные слои.

*Data distribution —* разделяет на потоки обработку данных, подачу их в сеть и обратное распространение ошибки. Разве что веса корректируются синхронно.

*Data Parallel —* разделяет на потоки обработку батчей и отправку их в сеть. Однако, остальное происходит синхронно. Более предпочтительно

**Структура проекта на PyTorch**

- CNN\_model.py # *модель(и), как на странице выше*

- train.py # *здесь есть параметры обучения*

- test.py # *тесты из Test DataSet (no grad and model.train())*

*-* eval.py # *тесты из других источников (например, .wav) (model.eval)*

*-* data.py # *организация данных (пример: tutorial/torch\_data.py)*

*-* best\_model.pt # *должно быть указано, что это за модель в dev/*

*по завершению разработки* *добавить в .hgignore*

*-* preprocess.py # *набор функций для подготовки (и проверки) сырого сигнала*

*-* preprocess.sh # *shell скрипт*

*-* utils.py # *вспомогательные иструменты (например, расчет accuracy)*

*-* net.py # *стандартная структура (Class: eval\_wav, eval\_batch, train, load\_model)*

*-* data/ # *пример того, как организованы данные (здесь же README.txt)*

# *здесь также мб 2-3 тестовые записи, на которых можно быстро проверить работоспособность модели*

*-* torch\_serve/ # *для серверного использования (пока что только для единичного образца)*

*-* MyHandler.py # *многие функции берутся из eval.py*

# *другие модули типа utils, data, eval подключается из проекта*

- debug.py # *для отладки перед упаковкой в архив.*

*по завершению разработки добавить в .hgignore*

*-* test\_request.py # *тестовый скрипт обработки запроса*

- trace\_model.py

*-* traced\_model.pt

- dev/ # *develop - папка, которая НЕ ВХОДИТ в итоговый проект. Она добавляется в* ***.hgignore****. Здесь хранятся результаты тестов, базовые модели-прототипы, краш-тесты, всякие временные файлы (preprocessed\_sound.wav) и т. п.*

\* \_\_init\_\_.py # *по желанию*

*-* README.txt

Замечание: **всевозможные проверки** сырой записи/записей в методе eval\_wavs проводятся при параметре **check\_sound**=True. Отдельно устанавливается параметр **preprocessing**=True/False, который использует ffmpeg при несовпадении sample rate, codec, n\_channels. В этом случае автоматически **check\_sound**=True. Чтобы сэкономить время и не выполнять эти проверки, нужно установить **check\_sound**=False. Однако в любом случае (в тч при подаче на вход numpy сэмпла вместо wavpath) нужно выполнить проверку на мин. длину записи.

- В методе *eval\_wavs* для вывода текстов ошибок должен быть параметр logger, который либо равен объекту (вывод в консоль/файл), либо None (без вывода вообще)

- В методе *postprocess* неверному входу будет соответствовать не просто None, а строка ошибки.