Исследование эффекта гроккинга

За основу взята статья Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets:
https://arxiv.org/pdf/2201.02177.pdf (https://arxiv.org/pdf/2201.02177.pdf)

Выполнили: Грозный Сергей, Мельник Юрий 209 гр.

1. Вступление

В данной статье описывается т.н. эффект гроккинга: нейросеть резко переходит от качества случайного угадывания к идеальному качеству, причём случается это сильно после точки переобучения.

Авторы данной работы наблюдают этот эффект на данных вида aob=c, где "a","b","c" - числа, а "o" - некая операция. Состовляется таблица, где строки и столбцы это всевозможные значения "a" и "b", в ячейках которой хранятся соответствующие этим "a" и "b" - "c". Далее, случайным образом стираются некоторые ячейки(то есть разбиваем выборку на train и test(пустые ячейки)). Задача состоит в том, чтобы заполнить пустые ячейки в соответствии с выше описанной операцией.

В этой научной работе авторы наблюдали этот эффект на многих операциях, но мы остановимся на нескольких из них. Тип нейросети - трансформер, в качестве оптимизатора будем использовать AdamW, поскольку данный эффект наиболее отчетливо наблюдается при его использовании.

2. Программная реализация

Библиотеки:

Ввод [16]:
```
from torch import nn
import torch
import numpy as np
from torch.nn.functional import cross_entropy
from torch.optim import AdamW, Adam
from torch.optim.lr_scheduler import LambdaLR
from net import Grokformer  # net - файл с реализацией трансформера
from tqdm import tqdm
import math
import matplotlib.pyplot as plt
```

Функция генерации данных:

p - деление по модулю p
function - операция

Ввод [17]:
```
def create_data_p(p: int, function):
    x = torch.arange(p)  # 0..p
    y = torch.arange(1, p)  # 1..p
    x, y = torch.cartesian_prod(x, y).T  # декартово произведение x и y
    result = function(x, y) % p
    return torch.stack([x, y, result]).T
```

Ввод [18]:
```
def prod(a, b):  # a*b
    return a * b
```

Ввод [19]:
```
def sinm(a, b):  # целая часть модуля синуса от a+b
    return (abs(torch.sin(a+b))*sinp).to(int)
```

Ввод [20]:
```
def nesim(a, b):  # несимметричная функция a*b+b*b
    return (a*b+b*b)
```

Ввод [21]:
```
p = 97
device = torch.device("cuda:0")  # "cpu" - процессор, "cuda:0" - видеокарта
train_ratio = 0.4  # какая доля выборки уйдет на train
batch_size = 512
budget = 10000  # регулирует кол-во эпох
sinp = 3*p  # множитель для функции синуса, чтобы результат был от 0 до sinp
func = prod  # операция
```

Авторы статьи в качестве входных параметров для трансформера использовали токены "a","o","b","=","c", но мы будем использовать только "a", "b", "c". Как нам кажется, токены "o" и "=" никакой ценности для нейросети не несут.

Ввод [22]:
```
# 1, 2, 3 столбец - "a", "b", "c" соответственно
example = create_data_p(p, func)
print(example)
```

```
tensor([[ 0,  1,  0],
        [ 0,  2,  0],
        [ 0,  3,  0],
        ...,
        [96, 94,  3],
        [96, 95,  2],
        [96, 96,  1]])
```

Перемешиваем выборку и разбиваем на train и val:

Ввод [23]:
```
data = create_data_p(p, func)
data = data.to(device)
data_index = torch.randperm(data.shape[0], device=device)
split = int(data.shape[0] * train_ratio)
training_set = data[data_index[:split]]
validation_set = data[data_index[split:]]
```

Ввод [24]:
```
training_set
```

Out[24]:
```
tensor([[90, 15, 89],
        [46, 29, 73],
        [21, 55, 88],
        ...,
        [32, 34, 21],
        [22, 67, 19],
        [19, 82,  6]], device='cuda:0')
```

Ввод [25]:
```
validation_set
```

Out[25]:
```
tensor([[53, 45, 57],
        [84, 66, 15],
        [90, 55,  3],
        ...,
        [46, 49, 23],
        [57, 23, 50],
        [92, 78, 95]], device='cuda:0')
```

Ввод [26]:
```
# используется модель из файла net.py
net = Grokformer(num_embeddings=99, embedding_dim=128, device=device)
optimizer = AdamW(net.parameters(), lr=1e-3, weight_decay=1., betas=(0.9, 0.98))
scheduler = LambdaLR(optimizer, lambda epoch: min(epoch/10, 1.))  # регулятор learning rate
```

Ввод [27]:
```
# кол-во шагов оптимизации за 1 эпоху
steps_per_epoch = math.ceil(training_set.shape[0] / batch_size)
```

Ввод [28]:
```
train_acc, val_acc, train_loss, val_loss = [], [], [], []
s = np.array([])  # сумма норм весов на каждой эпохе
```

Ввод [29]:
```
for epoch in tqdm(range(int(budget) // steps_per_epoch)):

    # на каждой эпохе перемешиваем train
    training_set = training_set[torch.randperm(training_set.shape[0]), :]

    for data, is_train in [(training_set, True), (validation_set, False)]:

        total_acc = 0
        total_loss = 0
        net.train(is_train)

        dl = torch.split(data, batch_size, dim=0)  # делим на батчи
        for input in dl:  # input - 1 батч
            input = input.to(device)  # используем видеокарту
            with torch.set_grad_enabled(is_train):
                logits = net(input[:, :-1])  # предсказание
                loss = cross_entropy(
                    logits, input[:, -1].flatten().to(torch.long))
                total_loss += loss.item() * input.shape[0]

            if is_train:  # пересчитываем веса, вычисляя градиенты; обновляем lr
                net.zero_grad()
                loss.backward()
                optimizer.step()
                scheduler.step()

            acc = (logits.argmax(-1) == input[:, -1]).float().mean()
            total_acc += acc.item()*input.shape[0]

        if is_train:
            # Считаем сумму норм весов модели
            # ********************************************************************
            f = 0
            for name, parameter in net.named_parameters():
                if ("weight" in name) and ("norm" not in name) and ("emb" not in name):
                    f += torch.norm(parameter, p=2).item()
            s = np.append(s, f)
            # ********************************************************************

            train_acc.append(total_acc / training_set.shape[0])
            train_loss.append(total_loss / training_set.shape[0])
        else:
            val_acc.append(total_acc / validation_set.shape[0])
            val_loss.append(total_loss / validation_set.shape[0])
    if (epoch + 1) % 100 == 0:  # каждые 100 эпох обновляем графики точности и ошибки
        steps = torch.arange(len(train_acc)).numpy() * steps_per_epoch
        plt.plot(steps, train_acc, label="train")
        plt.plot(steps, val_acc, label="val")
        plt.legend()
        plt.title("x*y mod 97 (training on 40% of data)")
        plt.xlabel("Optimization Steps")
        plt.ylabel("Accuracy")
        plt.xscale("log", base=10)
        # графики сохраняются в папке figures
        plt.savefig("figures/acc.png", dpi=150)
        plt.close()

        plt.plot(steps, train_loss, label="train")
        plt.plot(steps, val_loss, label="val")
        plt.legend()
        plt.title("x*y mod 97 (training on 40% of data)")
        plt.xlabel("Optimization Steps")
        plt.ylabel("Loss")
        plt.xscale("log", base=10)
        plt.savefig("figures/loss.png", dpi=150)
        plt.close()
```

```
100%|██████████████████████████████████| 1250/1250 [01:54<00:00, 10.90it/s]
```

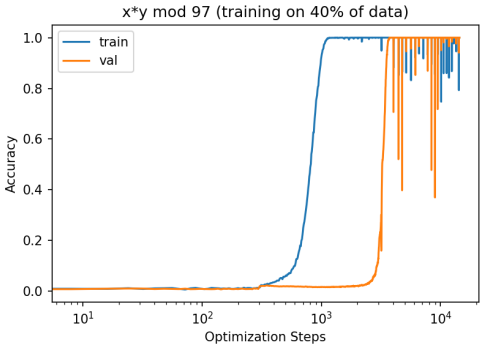
Построение графика суммы норм весов:

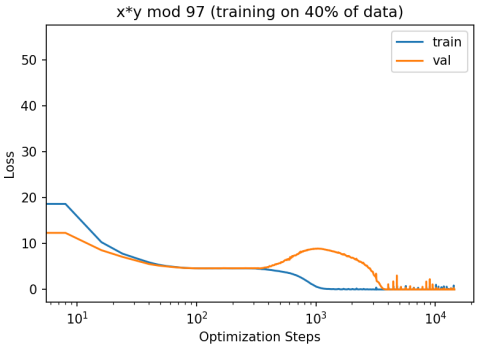
Ввод [30]:
```
plt.plot([x*steps_per_epoch for x in range(len(s))], s)
plt.xscale("log", base=10)
plt.title("x*y mod 97 (training on 40% of data)")
plt.xlabel("Optimization Steps")
plt.ylabel("Weights")
plt.savefig("figures/nweights.png", dpi=150)
plt.close()
```

3. Подтверждение результатов, полученных авторами статьи

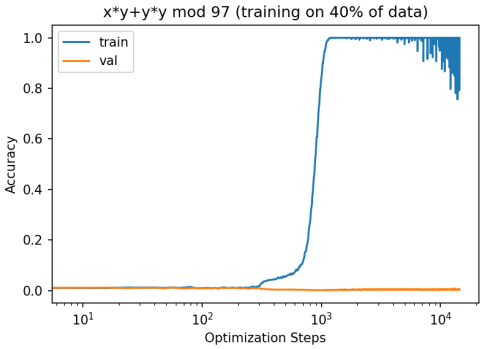
В первую очередь мы решили проверить результаты, полученные авторами статьи, а именно:

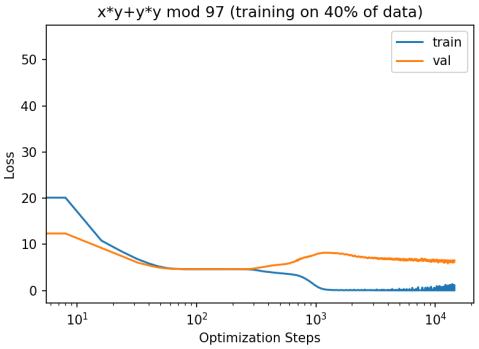
1. Эффект действительно наблюдается на данных из статьи.
2. На симметричных функциях данный эффект чаще наблюдается, чем на несимметричных.
3. С увеличением доли обучающей выборки увеличивается скорость наблюдения эффекта.





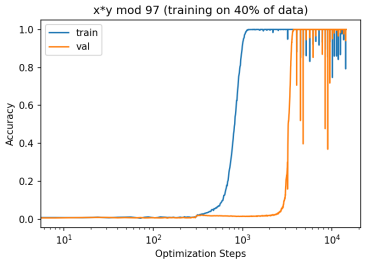
Как и было заявлено, на симметричной функции эффект наблюдается.

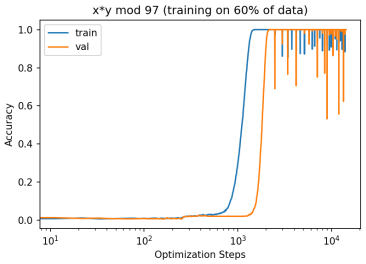


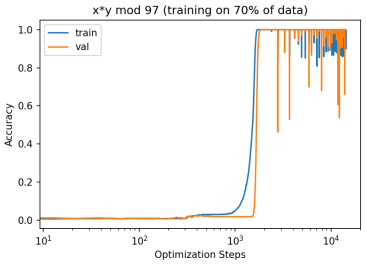


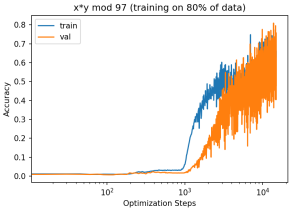
А на несимметричной функции добиться нужного эффекта не удалось.

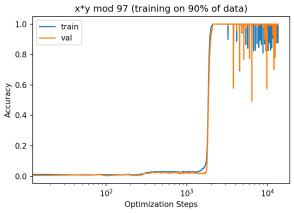
Теперь будем увеличивать долю обучающей выборки для симметричной операции.



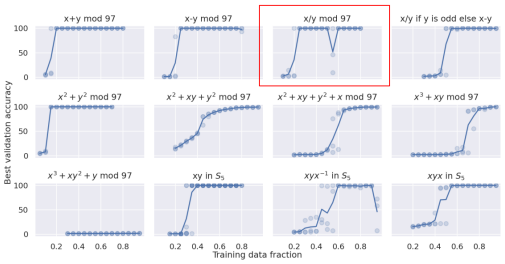






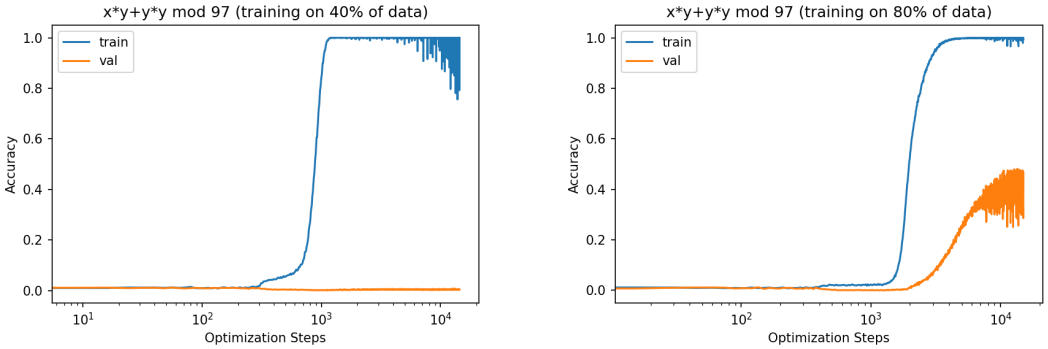


Действительно, с ростом доли обучающей выборки модель быстрее выходит на хорошую обобщающую способность после переобучения. Также стоит отметить случай, когда обучающая выборка составляет 80% от всей выборки, здесь как и на результатах, полученных авторами, наблюдается неожиданное ухудшение точности.



Отметим, что для операций "x/y" и "xy" по модулю простого числа результат совпадает.

Проверим данное наблюдение на несимметричной функции:

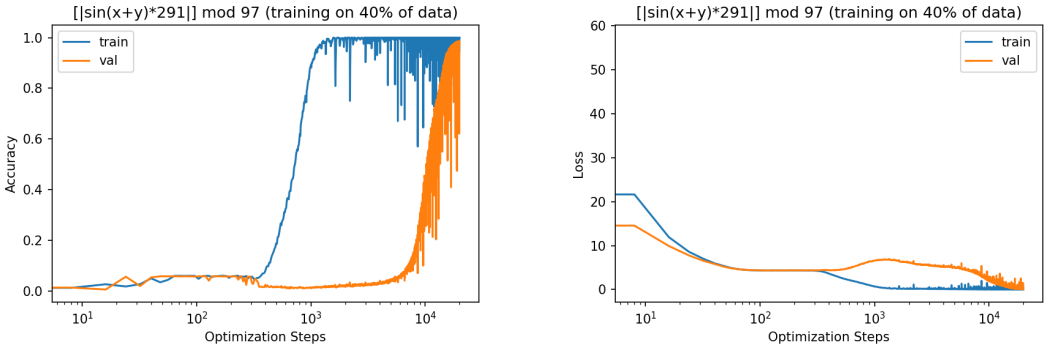


Несмотря на то, что при увеличении доли обучающей выборки имеются предпосылки для наблюдения эффекта гроккинга (правый рисунок), достигнуть желаемой точности не удается.

4. Анализ эффекта

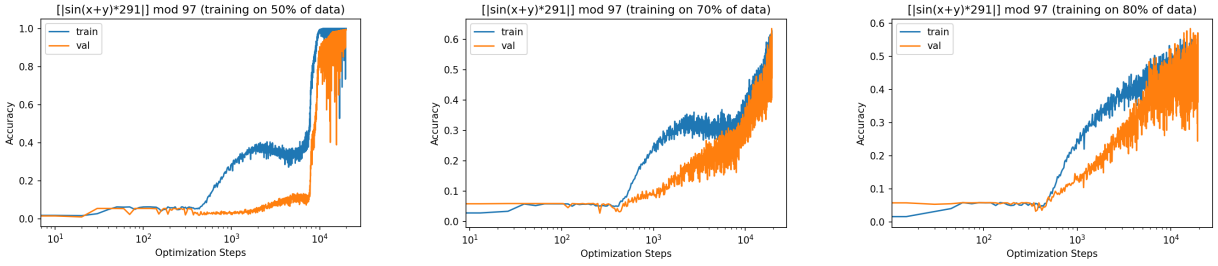
Другая операция

Рассмотрим операцию, которая не использовалась в статье. Попробуем на какую-нибудь симметричную операцию навесить относительно сложную функцию, например синус. Для того, чтобы работать с целыми неотрицательными числами, берем целую часть от модуля синуса, причем, чтобы деление по модулю на 97 имело смысл, умножим результат на некоторую константу "sinp", большую 97. Значение константы регулируется в программе.



Даже на такой, казалось бы, сложной функции эффект наблюдается.

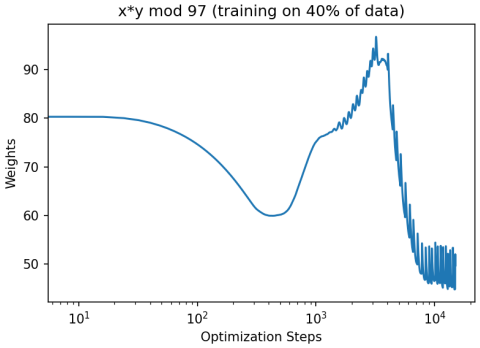
Попробуем увеличить долю обучающей выборки:

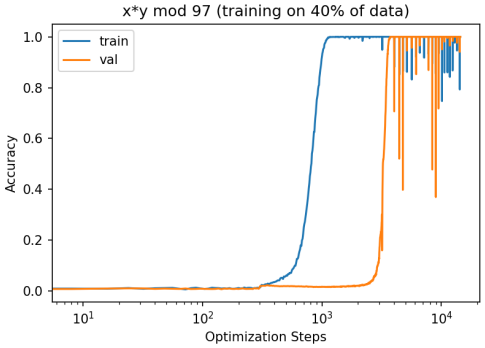


Заметим, что на 50% эффект гроккинга наблюдается не так отчетливо, причем у модели начинают появляться трудности даже на обучающей выборке. При дальнейшем увеличении нашего параметра эффект становится все менее различимым и не достигается необходимая точность.

Анализ весов

Проанализируем значения суммы норм весов модели на разных шагах оптимизации:



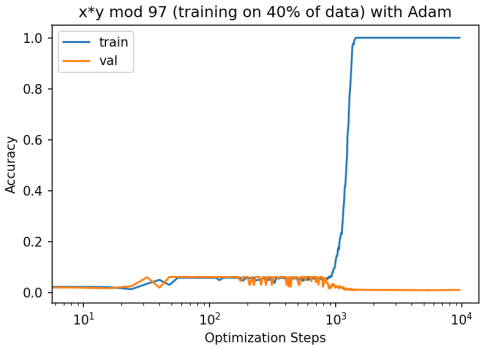


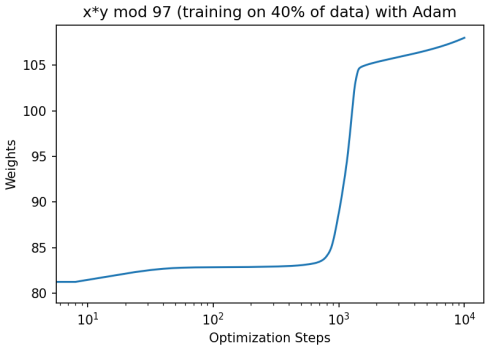
Отметим, что рост значения весов соответствует началу увеличения точности на обучающей выборке(отрезок от 0.75*10^3 до 10^3 шагов оптимизации). Далее при достижении точности на train, равной 1, и дальнейшем переобучении просходит немонотонный рост весов (отрезок от 10^3 до 0.5*10^4) вплоть до точки максимума, которой соответствует началу роста точности на валидационной выборке. После происходит резкое уменьшение значений весов, которое сопровождается увеличением точности на val вплоть до 1 (луч от 0.5*10^4 и далее).

Другими словами происходит следующее: веса сначала улетают в большие значения, переобучаясь, а потом спускаются вниз вдоль своего рода "направляющего вектора", который выдает одинаковые ответы, но имеет более низкий модуль весов.

Adam

Попробуем убрать weight decay, заменив AdamW на Adam. Сравним графики изменения весов для этих оптимизаторов.



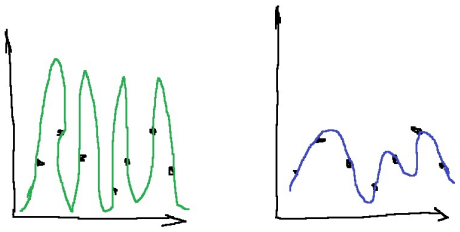


Невооруженным глазом видно, что для Adam значения весов модели на протяжении всего времени увеличиваются из-за отсутсвии L2 регуляризации, которая "штрафует" за большие значения весов. То есть не будет точки, начиная с которой значения весов будут уменьшатся, что соответствует началу гроккинга(по нашему предположению). Из этого можно сделать вывод, что weight decay играет ключевую роль в этом эффекте.

5. Гипотезы и предположения

Основываясь на результатах, описанных в статье и своих собственных, мы можем сделать следующие предположения:

1. С ростом доли обучающей выборки модель склоняется в пользу "выучивания" операции, а не пытается "запомнить" все данные, как происходит при переобучении.
2. Немаловажную роль играет L2-регуляризация, которая добавляет "гладкости" предсказанию. Это можно наблюдать на рисунке ниже: левый график соответствует переобучению модели, что ведет к потери точности на тестовой выборке; правый - переобучение также присутствует, но засчет L2 регуляризации график проходит "гладко" через все точки и складывается впечатление, что модель выучила правило.



3. В некоторый момент после наступления переобучения засчет наличия weight decay, штраф за веса начинает превалировать над ошибкой модели, что заставляет ее искать другое решение с точки зрения выгоды относительно весов, которое в конечном счете оказывается оптимальным.
4. Прослеживается схожесть с раннее открытым явлением "double descent" и поэтому, вероятно, они имеют одинаковую природу возникновения.

Итоги

В ходе проделанной работы мы познакомились с основами глубинного обучения, разобрались в устройстве нейросети трансформер, изучили библиотеку pytorch, что позволило нам воспроизвести эффект гроккинга и сделать предположения относительно причин его возникновения.