

DEEP
SCHOOL

лекция

Quantization

Опять сжимаем

спикер



Дмитрий
Чудаков



Спустил академическую
карьеру на округление чисел

О чём будет лекция

STOP QUANTIZE



*Floats were
not supposed to be
represented as integer*

*They have played us
for absolute fools*

1

Поговорим про что floats, и как их испортить

2

Поговорим про квантование, и как превратить
floats в integer

3

Угорим по продвинутым техникам

Содержание

Floats in DL →

Содержание

Floats in DL → Квантование →

Содержание

Floats in DL → Квантование → **Quantization**
Aware Training →

Содержание

Floats in DL → Квантование → Quantization
Aware Training → **Tricks** →

Содержание

Floats in DL → Квантование → Quantization
Aware Training → Tricks → **Fast Guide**
Torch →

Floats in DL

Посмотрим как работают наши любимые чиселки в компьютере

$$1.2345 = \underbrace{12345}_{\text{мантисса}} \times 10^{\overbrace{-4}^{\text{порядок}}}$$

Используется экспоненциальная запись числа

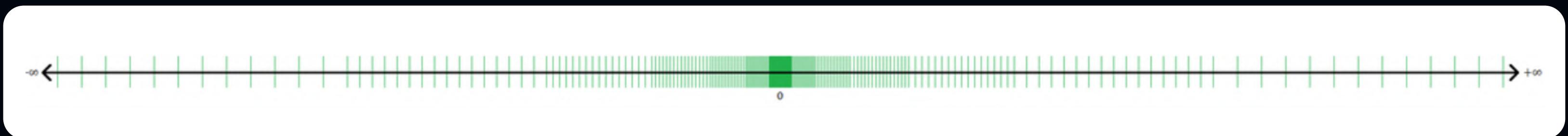
Очень большой диапазон чисел (и маленькие и большие)



Неточное представление чисел



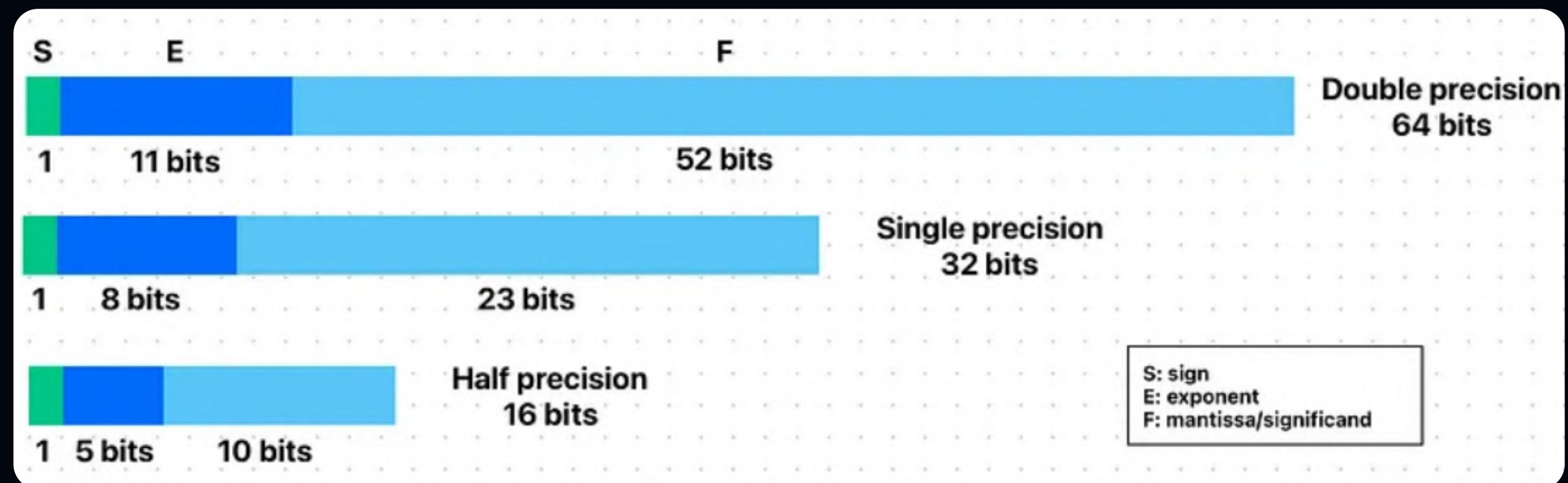
Сетки очень **робастные**, а флоты
очень **избыточные**



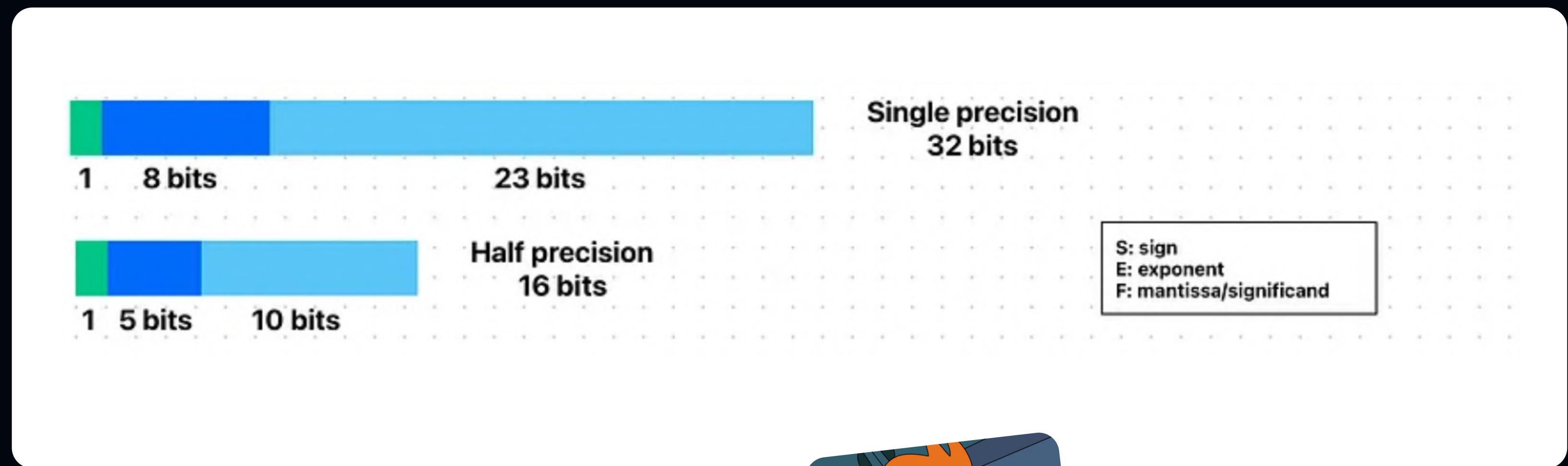
Нормальные
люди →

Странные
люди →

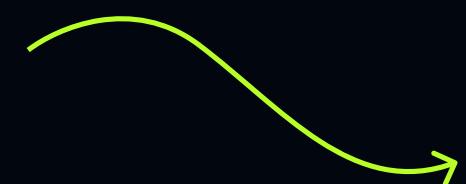
Психи
(Мы) →



Посмотрим как работают наши любимые
чиселки в компьютере

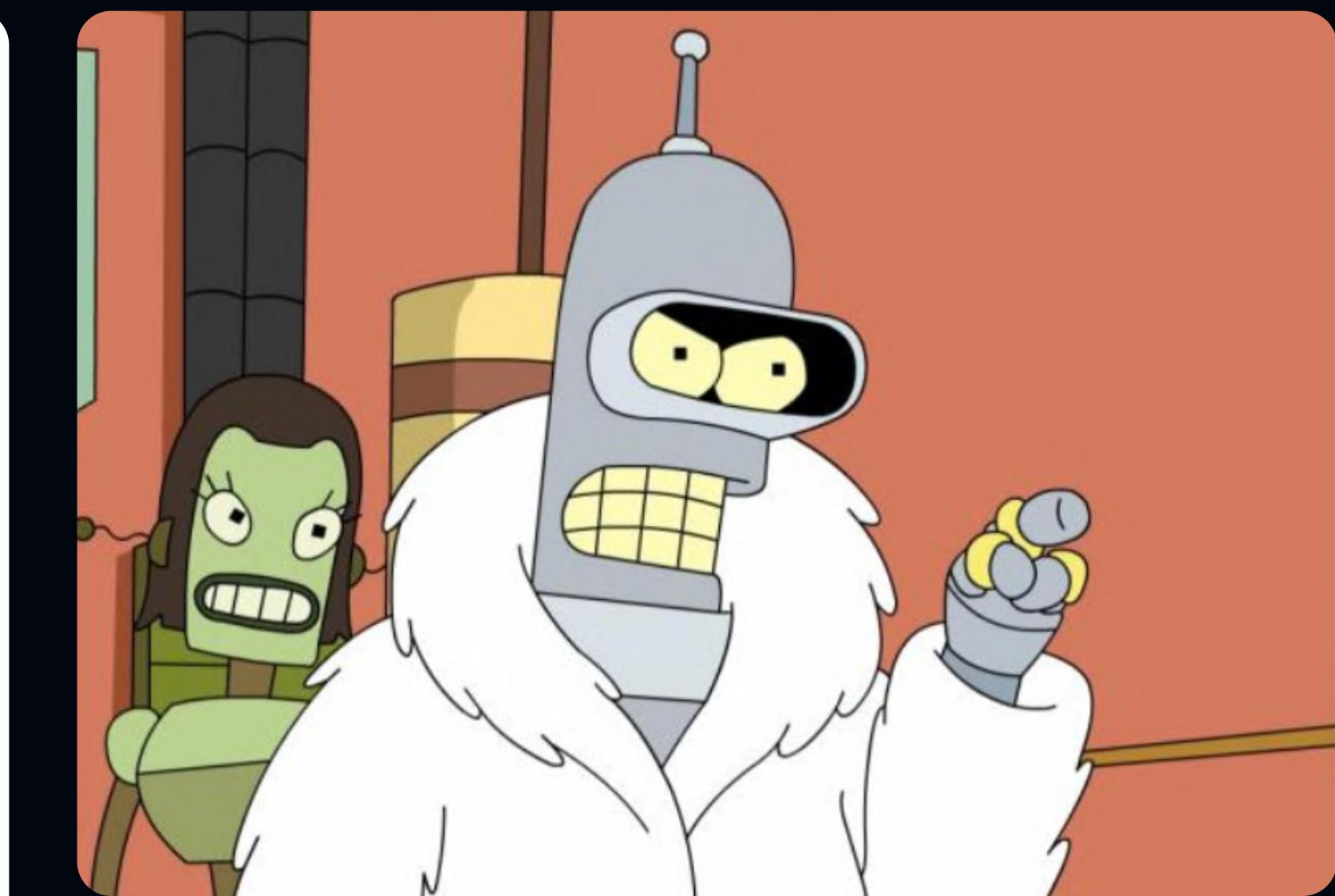
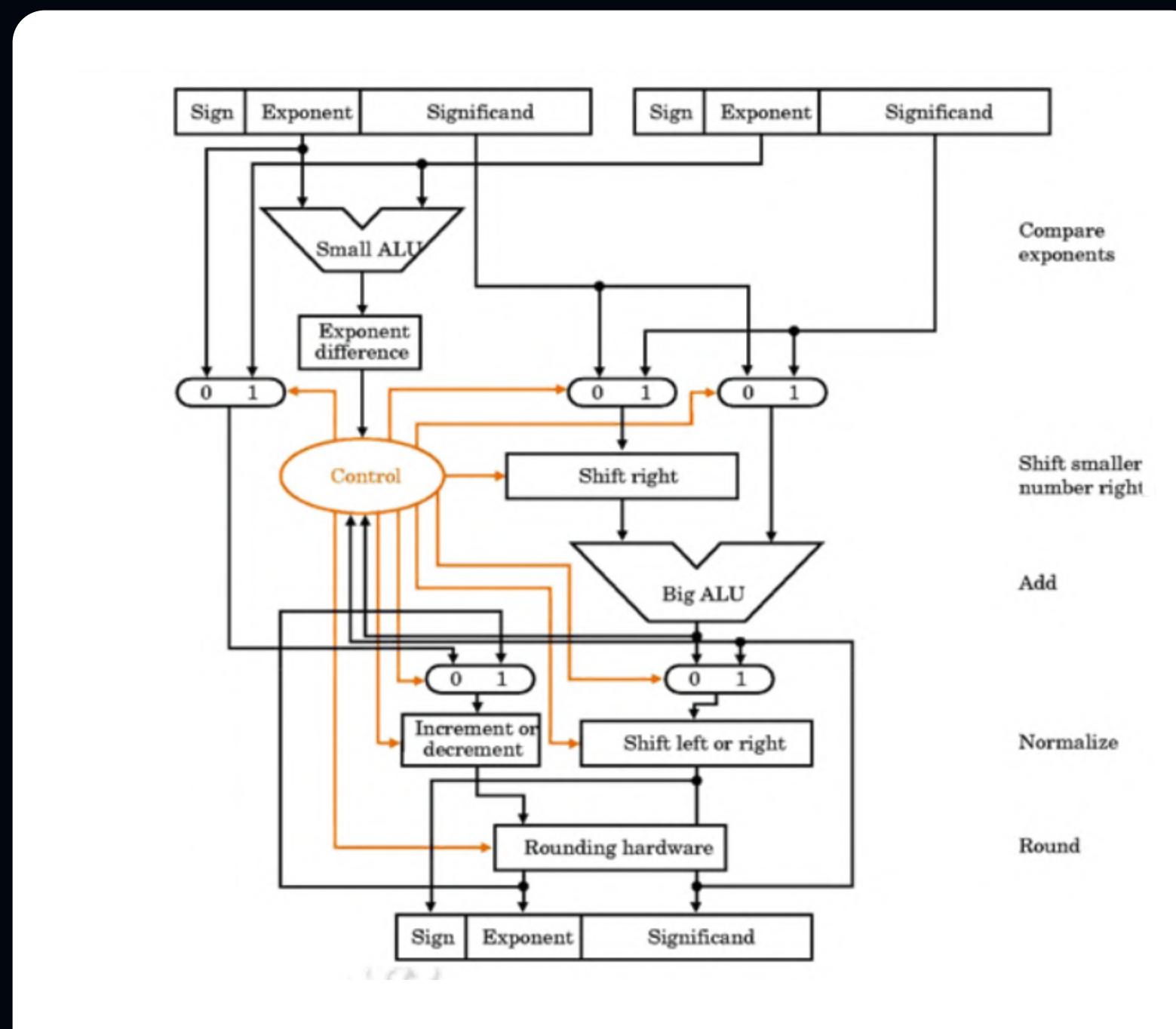


В два раза меньше бит, в два раза
меньше памяти и вычислений!



В чём
подвох?

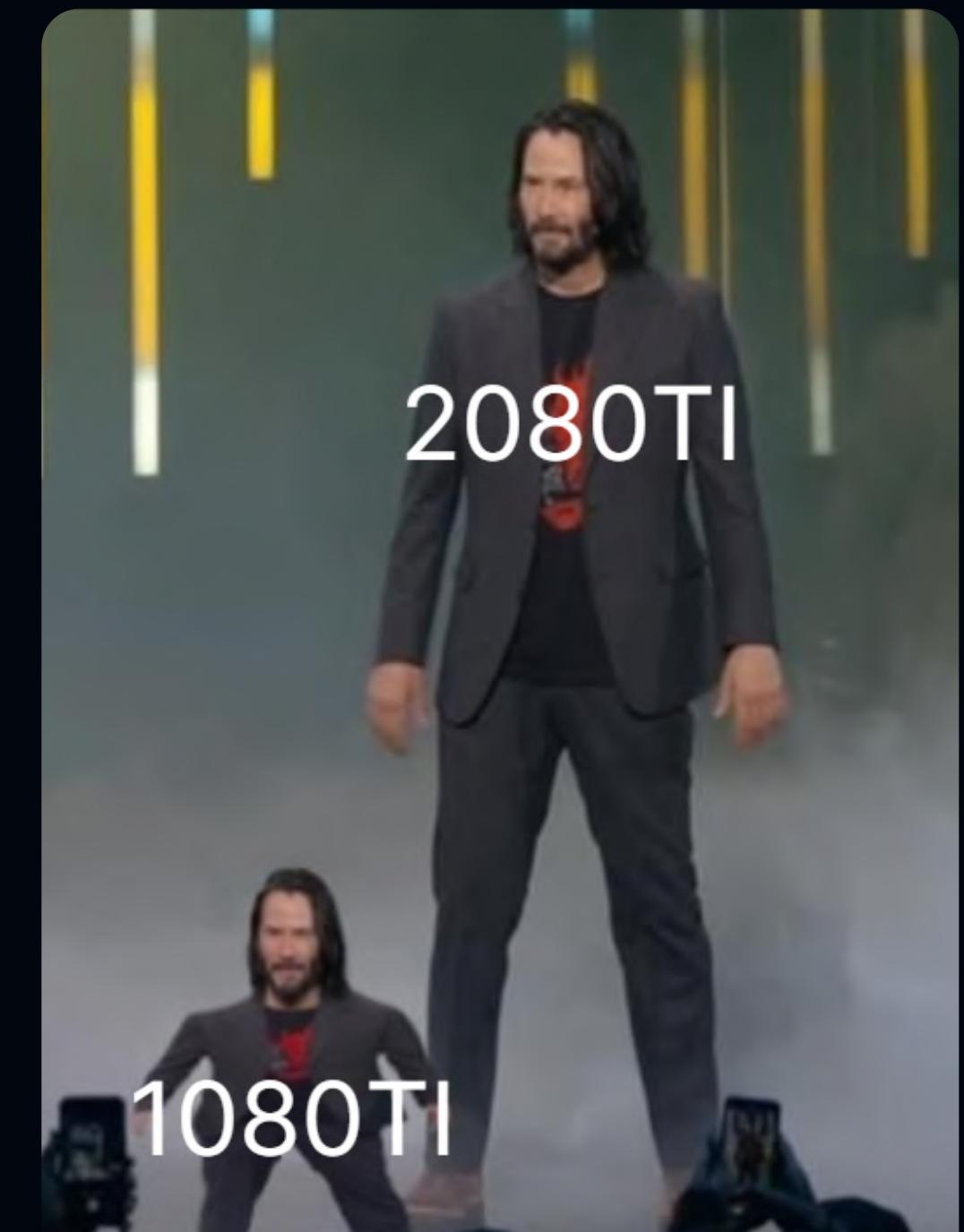
Чиселки умножаются на проце,
а там всё трудно



Этот парень сгибает только Floats32

Видеокарта (да любое железо) должны поддерживать FP16 операции!

	Supported CUDA Core Precisions								Supported Tensor Core Precisions									
	FP8	FP16	FP32	FP64	INT1	INT4	INT8	TF32	BF16	FP8	FP16	FP32	FP64	INT1	INT4	INT8	TF32	BF16
NVIDIA Tesla P4	No	No	Yes	Yes	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No
NVIDIA P100	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
NVIDIA Volta	No	Yes	Yes	Yes	No	No	Yes	No	No	No	Yes	No						
NVIDIA Turing	No	Yes	Yes	Yes	No	No	Yes	No	No	No	Yes	No	No	Yes	Yes	Yes	No	No
NVIDIA A100	No	Yes	Yes	Yes	No	No	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
NVIDIA H100	No	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes	No	Yes	No	No	Yes	Yes	Yes



А обучение?
Просто будем учить в fp16?

А обучение? Просто будем учить в fp16?

Увы, нас покарает underflow

1

Не все слои считаем в FP16. Усреднения в батчнормах, average pooling и прочие в FP32 (по сути `conv` и `matmul` только в fp16)

2

Храним копию весов в FP32 и применяем FP16 градиент к ним (иначе lr1e-10 нас покарает)

3

Делаем loss scaling (GradScaling)

А обучение? Просто будем учить в fp16?

1 Умножим loss на большое число

2 Посчитаем градиенты

3 Поделим градиенты на большое число

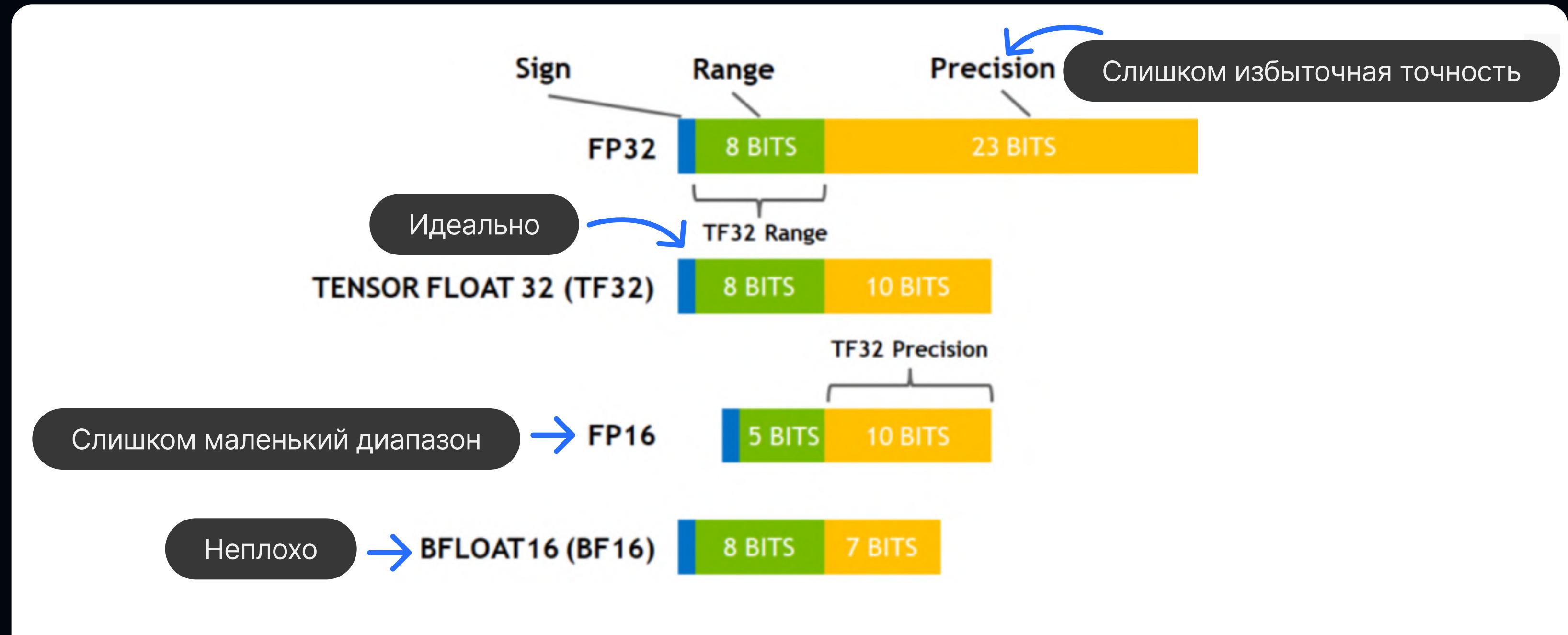
```
# Constructs a ``scaler`` once, at the beginning of the convergence run, using default arguments.  
# If your network fails to converge with default ``GradScaler`` arguments, please file an issue.  
# The same ``GradScaler`` instance should be used for the entire convergence run.  
# If you perform multiple convergence runs in the same script, each run should use  
# a dedicated fresh ``GradScaler`` instance. ``GradScaler`` instances are lightweight.  
scaler = torch.cuda.amp.GradScaler()  
  
for epoch in range(0): # 0 epochs, this section is for illustration only  
    for input, target in zip(data, targets):  
        with torch.autocast(device_type=device, dtype=torch.float16):  
            output = net(input)  
            loss = loss_fn(output, target)  
  
        # Scales loss. Calls ``backward()`` on scaled loss to create scaled gradients.  
        scaler.scale(loss).backward()  
  
        # ``scaler.step()`` first unscales the gradients of the optimizer's assigned parameters.  
        # If these gradients do not contain ``inf``'s or ``NaN``'s, optimizer.step() is then called,  
        # otherwise, optimizer.step() is skipped.  
        scaler.step(opt)  
  
        # Updates the scale for next iteration.  
        scaler.update()  
  
    opt.zero_grad() # set_to_none=True here can modestly improve performance
```

А обучение? Просто будем учить в fp16?

На трансформерах улетаем в космос,
там проявляет себя экономия
на **запихивание тензора в память**



А можно проще?



Не ожидали?

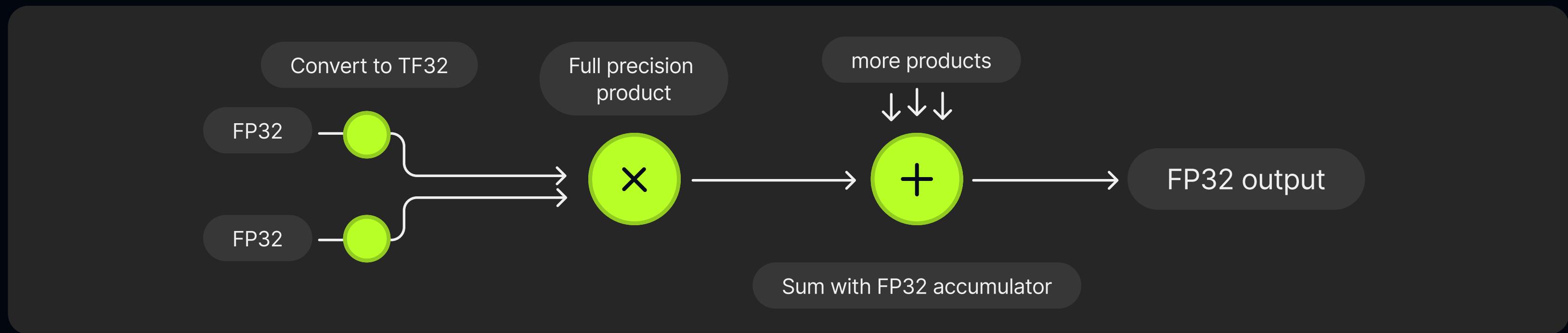
TensorFloat-32 (TF32) on Ampere (and later) devices

Starting in PyTorch 1.7, there is a new flag called `allow_tf32`. This flag defaults to `True` in PyTorch 1.7 to PyTorch 1.11, and `False` in PyTorch 1.12 and later. This flag controls whether PyTorch is allowed to use the TensorFloat32 (TF32) tensor cores, available on NVIDIA GPUs since Ampere, internally to compute matmul (matrix multiplies and batched matrix multiplies) and convolutions.

DL frameworks

TF32 is the default mode for AI on A100 when using the NVIDIA optimized deep learning framework containers for TensorFlow, PyTorch, and MXNet, starting with the 20.06 versions available at [NGC](#). TF32 is also enabled by default for A100 in framework repositories starting with [PyTorch 1.7](#), [TensorFlow 2.4](#), as well as nightly builds for [MXNet 1.8](#). Deep learning researchers can use the framework repositories and containers listed earlier to train single-precision models with benefits from TF32 Tensor Cores.

По умолчанию уже и не считают в FP32



Operations

TF32 mode accelerates single-precision convolution and matrix-multiply layers, including linear and fully connected layers, recurrent cells, and attention blocks. TF32 does not accelerate layers that operate on non-FP32 tensors, such as 16-bits,

Рецептики по Floats

1

По умолчанию всё и так в TF32

2

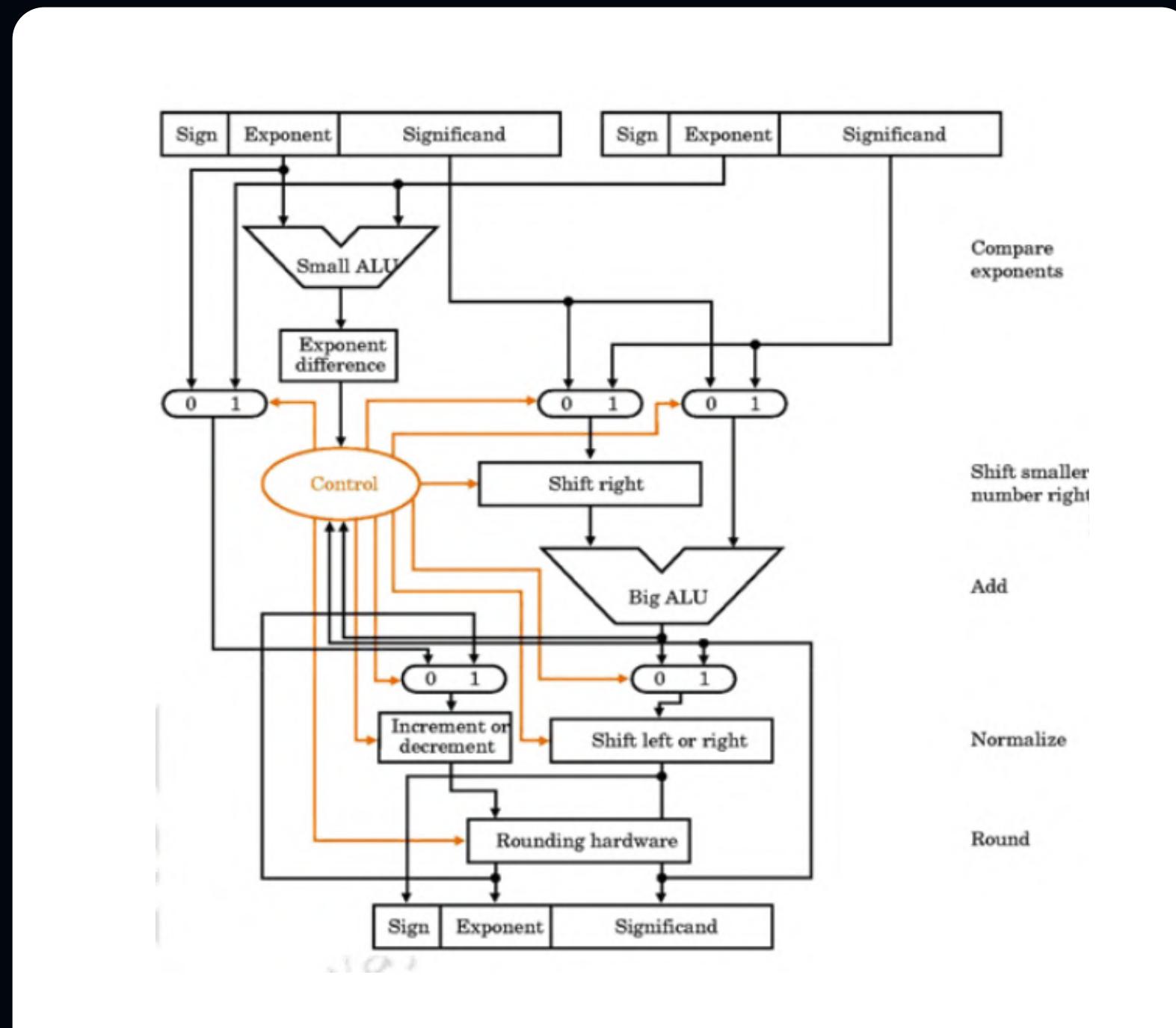
Если нужно ускориться, можно конвернуть в FP16.
Если хотим учить\дообучать используем automatic
mixed precision и grad scaling

3

Можем учить в bfloat16, torch autocast поддерживает.
Только для него не нужен grad scaling. Pytorch
предупреждают что модели обученные в bf16
потом не дообучить в fp16

Квантование

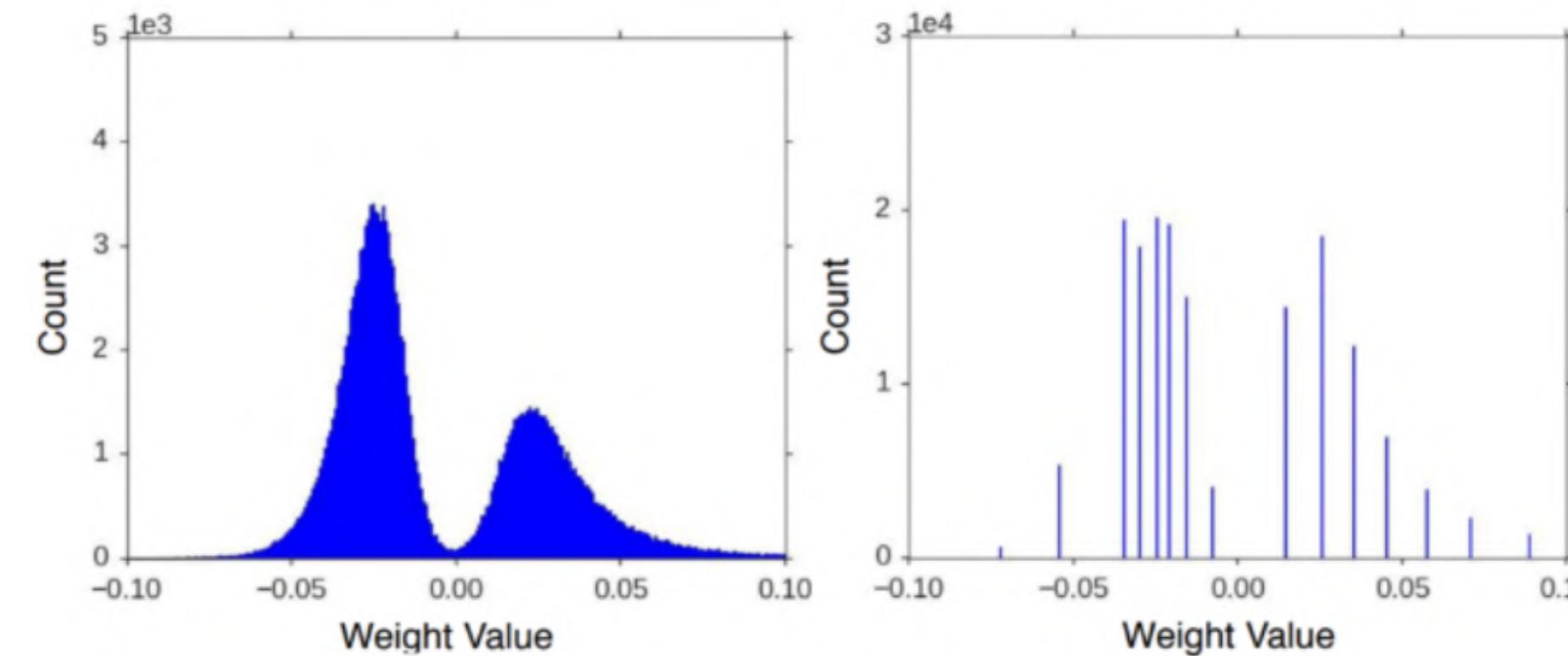
Почему Float16 не всему голова?



- 1 Для Float нужна нормальная поддержка процессора
- 2 Меньше fp16 уже начинаются проблемы с точностью и обучением
- 3 Все что дальше fp16 из коробки не работает

А давайте считать тогда всё integer!

→ Смапим участок реальных чисел на целочисленный диапазон



Source: Han et al

А давайте считать тогда всё integer!

$$\text{clamp}(r; a, b) := \min(\max(r, a), b)$$

$$s(a, b, n) := \frac{b - a}{n - 1}$$

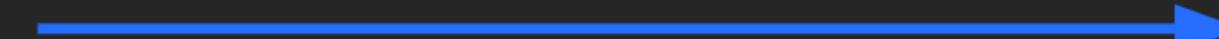
$$q(r; a, b, n) := \left[\frac{\text{clamp}(r; a, b) - a}{s(a, b, n)} \right]$$

Regular Float32 network

Floating point arithmetic

Quantized Int8 network

Integer arithmetic



А давайте считать тогда всё integer!

$$\text{clamp}(r; a, b) := \min(\max(r, a), b)$$

$$s(a, b, n) := \frac{b - a}{n - 1}$$

$$q(r; a, b, n) := \left[\frac{\text{clamp}(r; a, b) - a}{s(a, b, n)} \right]$$

Взяли интервал от **a** до **b** и натянули его на интервал [0, 255]

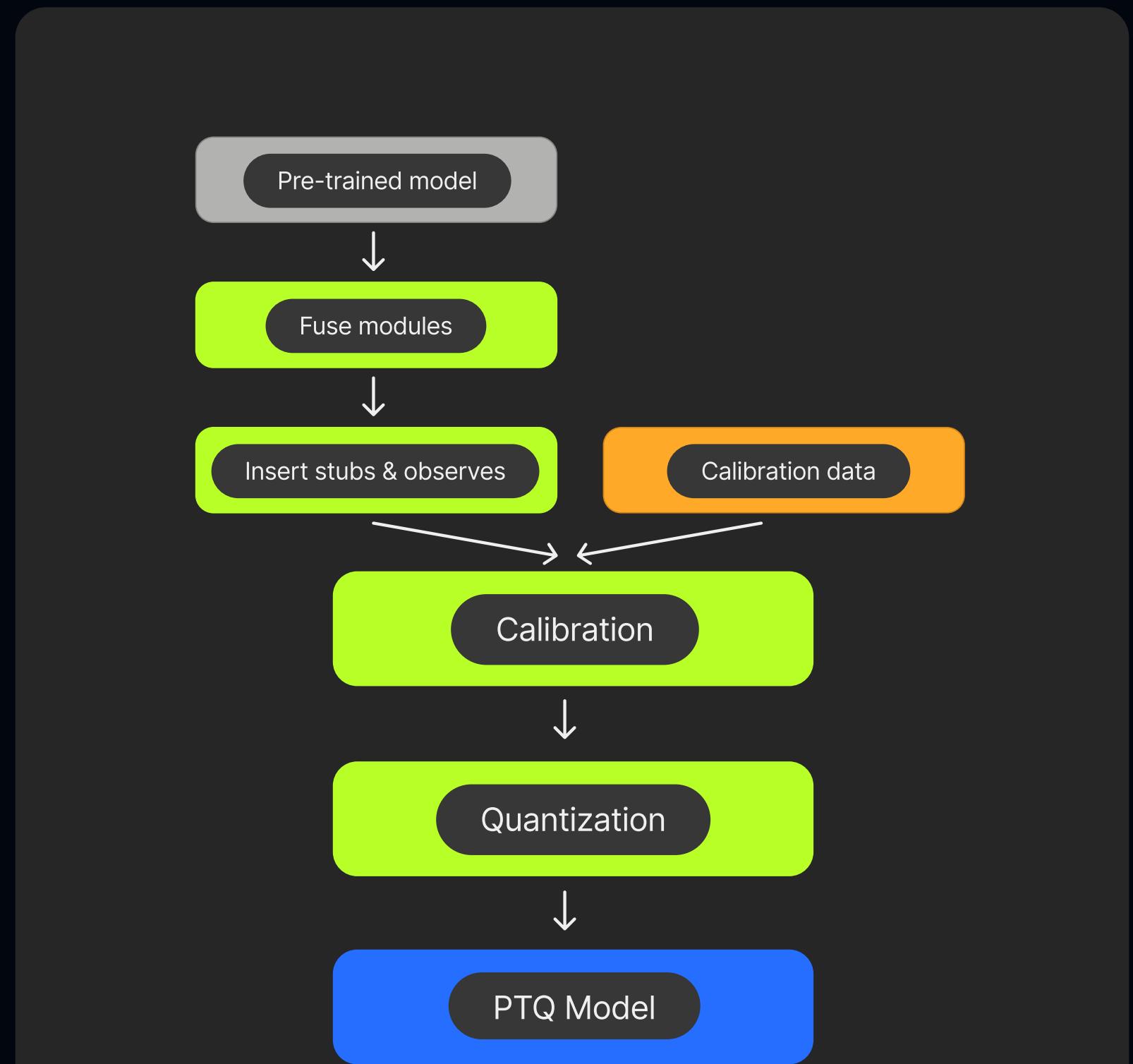
Что нам необходимо для квантования?

1 Какой отрезок квантовать — **левый** и **правый** порог квантования. **a** и **b**

2 На каком железе (фреймворки будем считать сеть) — Схема квантования

Общий пайплайн квантования

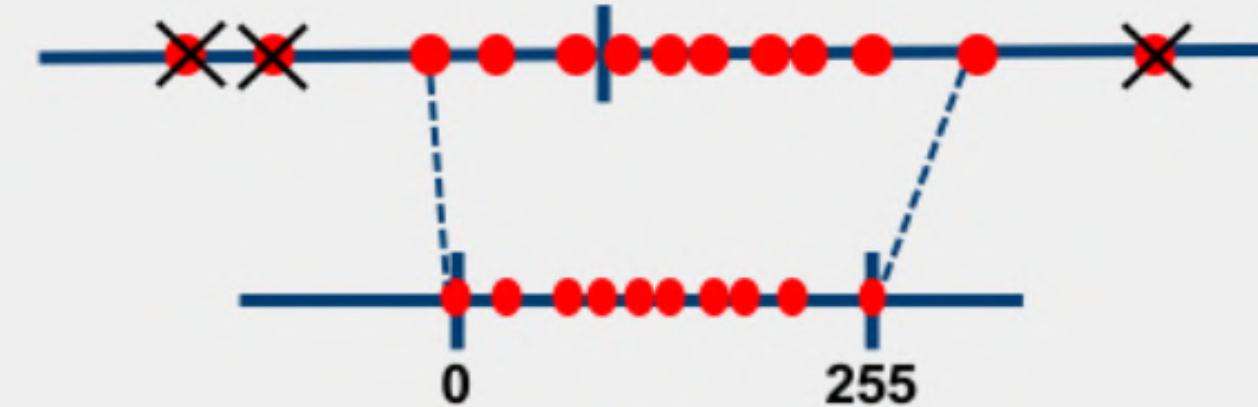
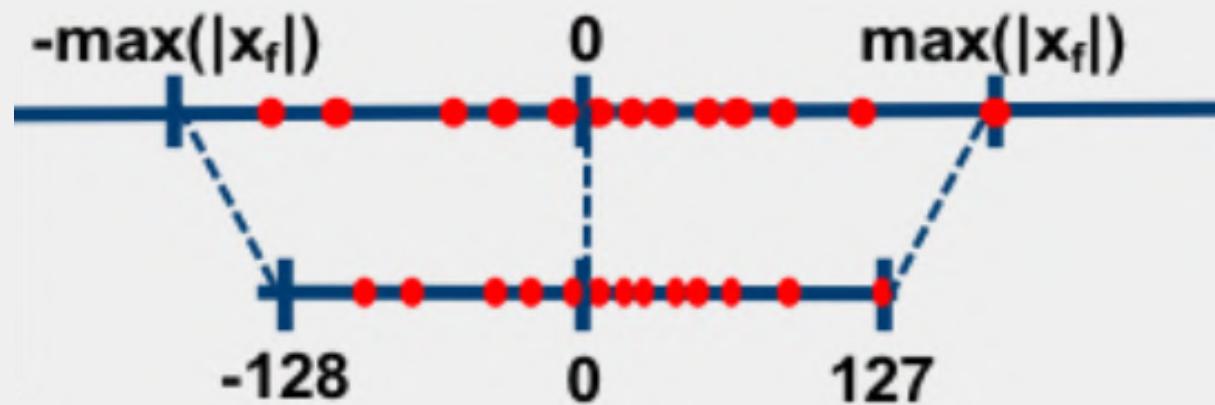
- Убираем мусор
- Вставляем наблюдателей, чтобы набрать статистики для активаций
- Прогоняем данные, т.е. калибуруем
- Квантуем



Что такое схема квантования и почему она отличается?

Схема квантования говорит о том, как именно мы квантуем числа, куда округляем значения, какие именно слои мы считаем в интах.

Что такое схема квантования и почему она отличается?



1. Симметричное и Ассиметричное квантование. Т.е. Симметричный ли отрезок квантования относительно нуля

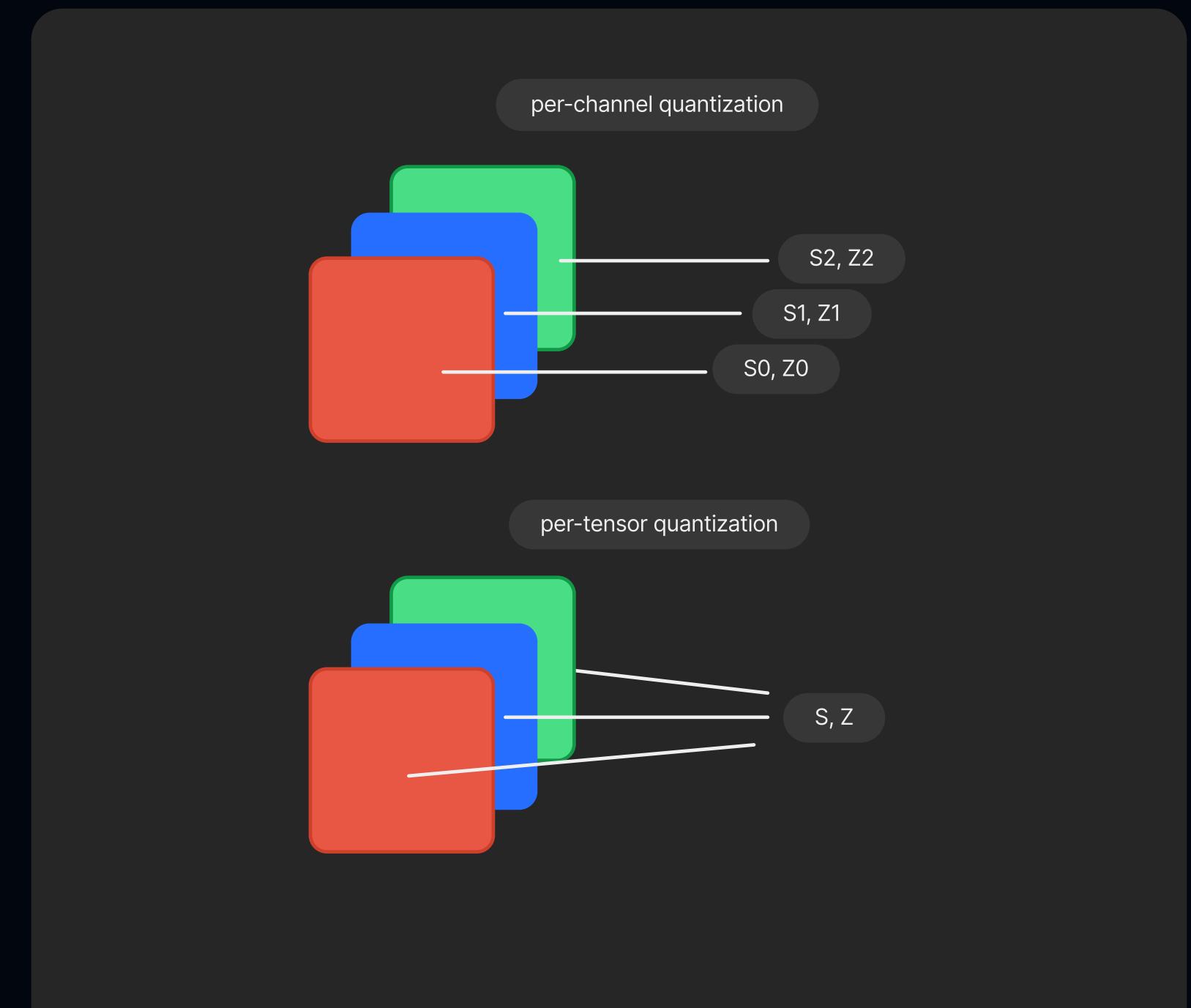
2. Векторное или скалярное квантования весов. Векторное квантование = независимые пороги для каждого отдельного фильтра

Per channel quantization и per tensor quantization

Что такое схема квантования и почему она отличается?

Векторное или скалярное квантования весов. Векторное квантование = независимые пороги для каждого отдельного фильтра

Per channel quantization и per tensor quantization



Что это нам даёт?

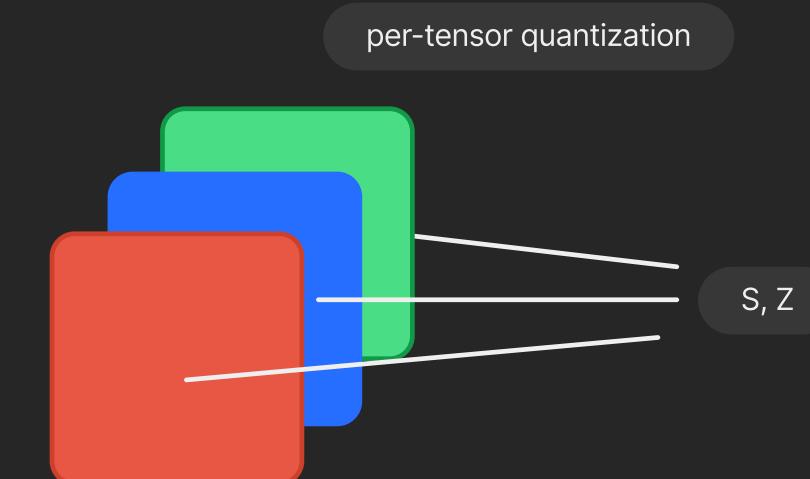
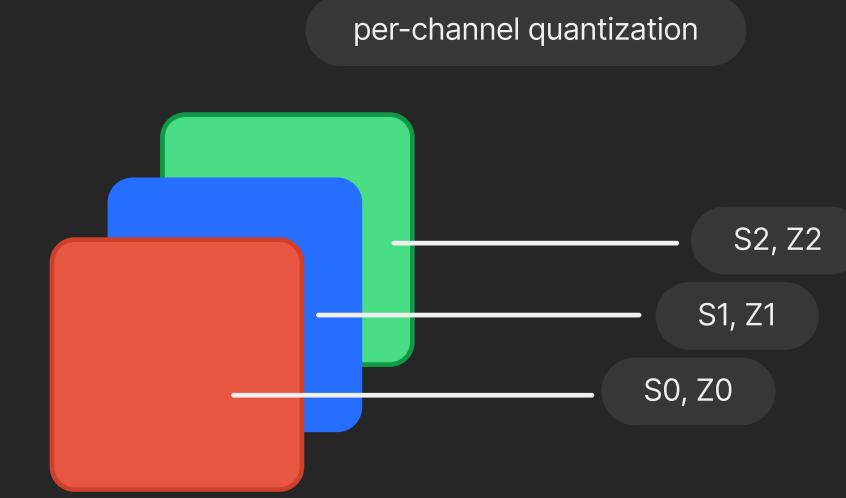
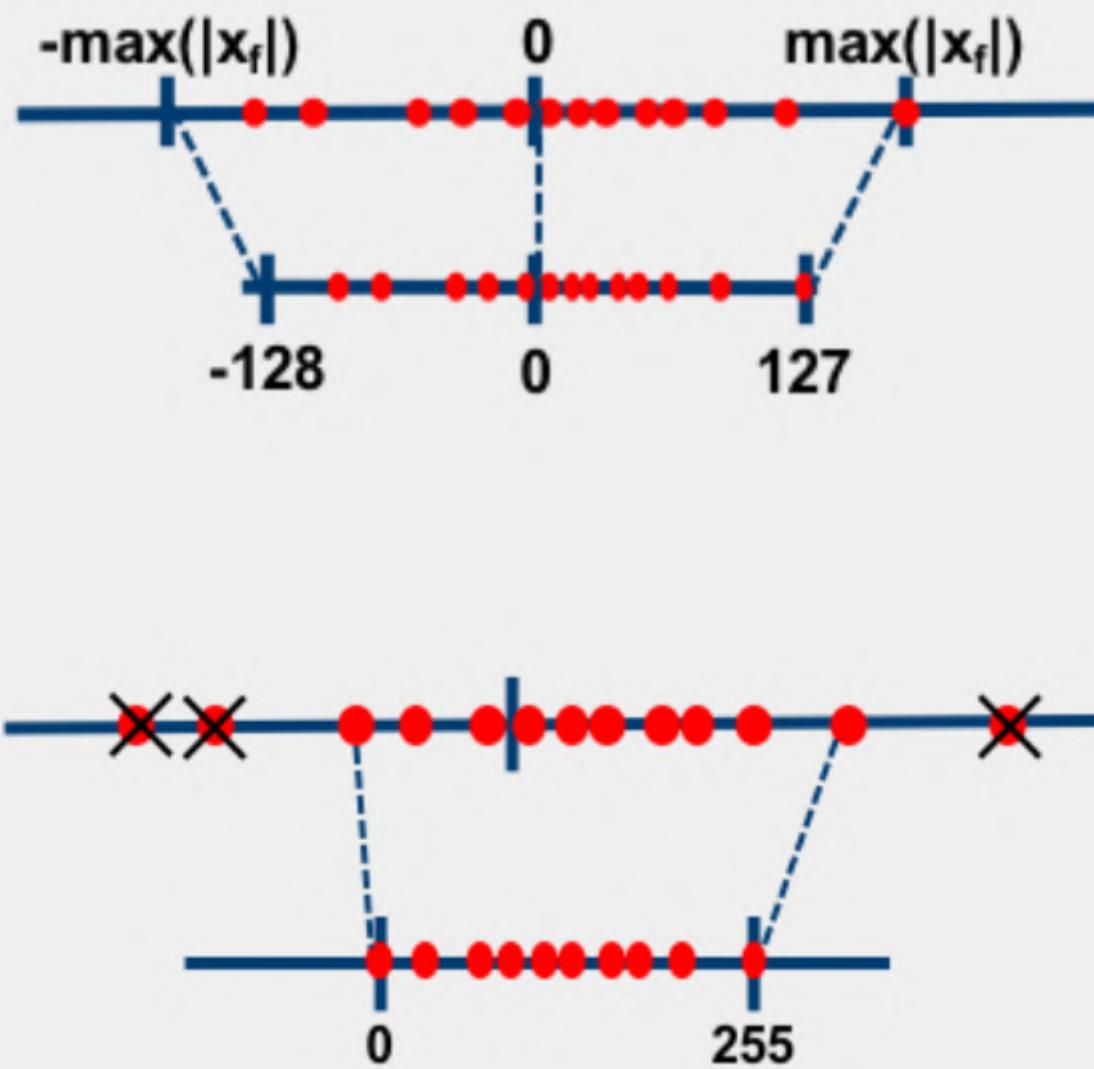
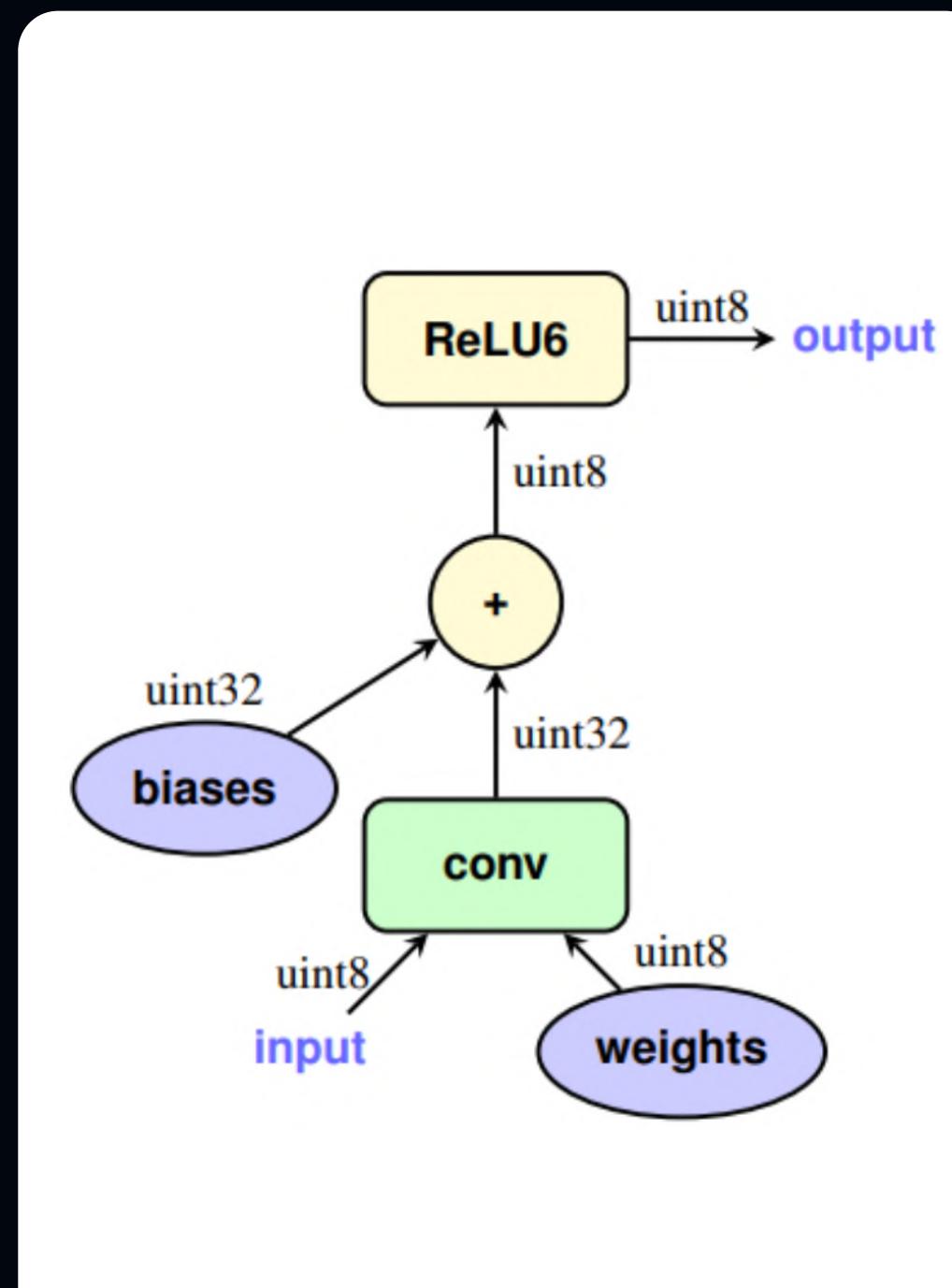


Схема квантования Tensorflow V1

Ещё где-то такое бывает



$$r = S(q - Z)$$

$$S_3 \left(q_3^{(i,k)} - Z_3 \right) = \sum_{j=1}^N S_1 \left(q_1^{(i,j)} - Z_1 \right) S_2 \left(q_2^{(j,k)} - Z_2 \right)$$

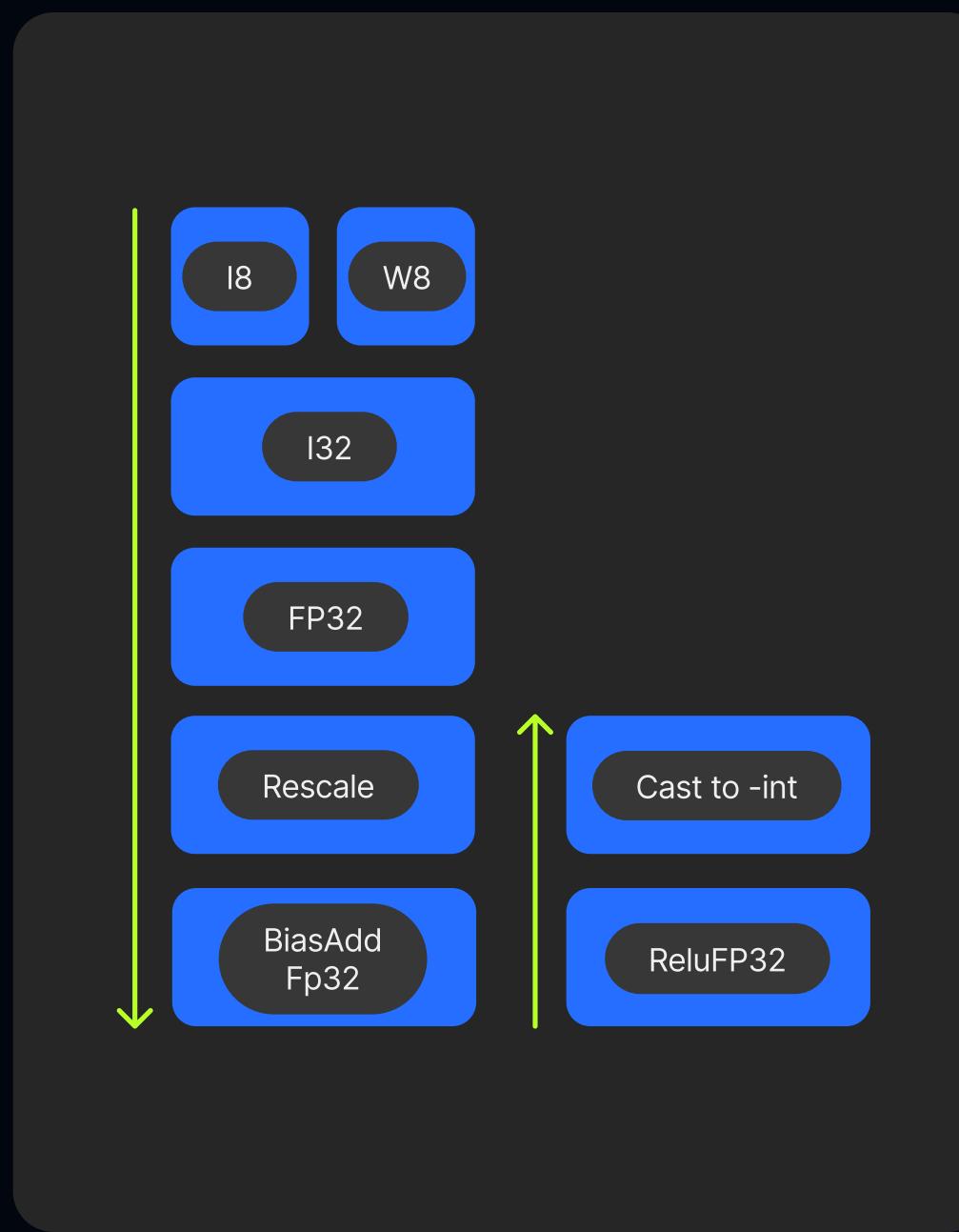
$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^N \left(q_1^{(i,j)} - Z_1 \right) \left(q_2^{(j,k)} - Z_2 \right)$$

$$M := \frac{S_1 S_2}{S_3}$$

$$M = 2^{-n} M_0$$

TensorRT

Per-Tensor скейлы

**Pseudocode for the INT8 conv kernel**

```

// I8 input tensors: I8_input, I8_weights,
// F32 bias (original bias from the F32 model)
// F32 scaling factors: input_scale, output_scale, weights_scale[K]
// I8 output tensors: I8_output

I32_gemm_out = I8_input * I8_weights          // Compute INT8 GEMM (DP4A)

F32_gemm_out = (float)I32_gemm_out           // Cast I32 GEMM output to F32 float

// At this point we have F32_gemm_out which is scaled by ( input_scale * weights_scale[K] ),
// but to store the final result in int8 we need to have scale equal to "output_scale", so we have to rescale:
// (this multiplication is done in F32, *_gemm_out arrays are in NCHW format)
For i in 0, ... K-1:
    rescaled_F32_gemm_out[ :, i, :, :] = F32_gemm_out[ :, i, :, :] * [ output_scale / (input_scale * weights_scale[ i ] ) ]

// Add bias, to perform addition we have to rescale original F32 bias so that it's scaled with "output_scale"
rescaled_F32_gemm_out _with_bias = rescaled_F32_gemm_out + output_scale * bias

// Perform ReLU (in F32)
F32_result = ReLU(rescaled_F32_gemm_out _with_bias)

// Convert to INT8 and save to global
I8_output = Saturate( Round_to_nearest_integer( F32_result ) )

```

Pytorch X86

```
# original model
# all tensors and computations are in floating point
previous_layer_fp32 -- linear_fp32 -- activation_fp32 -- next_layer_fp32
/
linear_weight_fp32

# statically quantized model
# weights and activations are in int8
previous_layer_int8 -- linear_with_activation_int8 -- next_layer_int8
/
linear_weight_int8
```

Pytorch X86

```
validator.evaluate()
# batch accuracy 0.70650: 100%|██████████| 157/157 [00:29<00:00,  5.30it/s] Это если фьюзить батчнормы
# batch accuracy 0.70650: 100%|██████████| 157/157 [00:39<00:00,  4.01it/s] А это если не фьюзить
# В 1.3 раза быстрее на хаяву!
```

```
50     validator.evaluate()
51     # Вот это реально злодейский буст по скорости. в 6 раз!
52     # А если ещё учитывать фьюзинг то это 8 раз. ОГОНЬ!
53     # batch accuracy 0.62980: 100%|██████████| 157/157 [00:05<00:00, 29.81it/s]
54
```

Пример ускорения торча x86

Точность просела, что делать?

```
validator.evaluate()
# batch accuracy 0.70650: 100%|██████████| 157/157 [00:29<00:00,  5.30it/s] Это если фьюзить батчнормы
# batch accuracy 0.70650: 100%|██████████| 157/157 [00:39<00:00,  4.01it/s] А это если не фьюзить
# В 1.3 раза быстрее на халюву!
```

```
50      validator.evaluate()
51      # Вот это реально злодейский буст по скорости. в 6 раз!
52      # А если ещё учитывать фьюзинг то это 8 раз. ОГОНЬ!
53      # batch accuracy 0.62980: 100%|██████████| 157/157 [00:05<00:00, 29.81it/s]
54
```

Точность просела, что делать?

Дообучение

→ Quantization Aware Training

Tricks

→ Rescaling, мудрёная калибровка и т.д.

Quantization Aware Training

Что нам мешает обучать квантованные сетки?

$$\text{clamp}(r; a, b) := \min(\max(r, a), b)$$

$$s(a, b, n) := \frac{b - a}{n - 1}$$

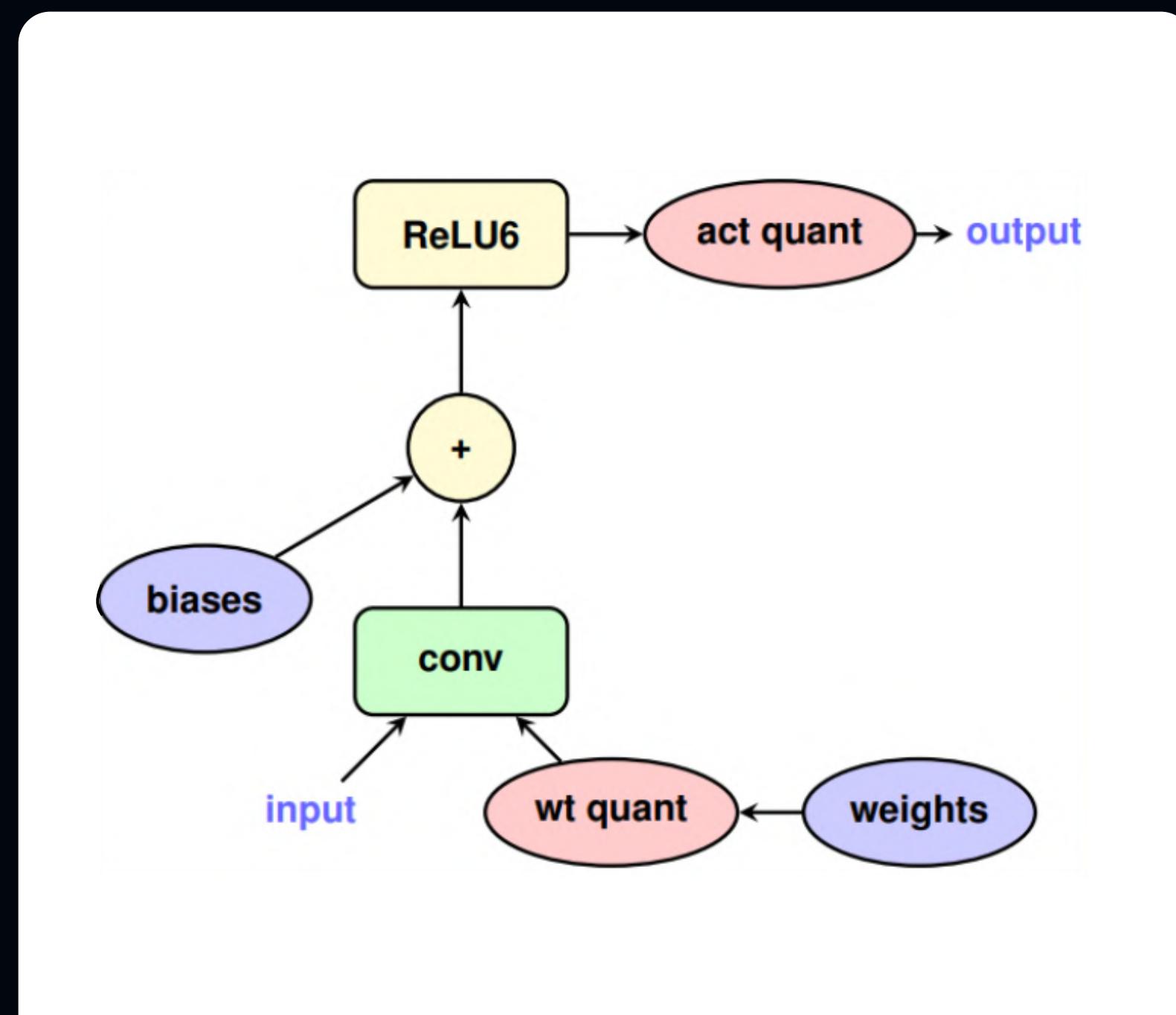
$$q(r; a, b, n) := \left[\frac{\text{clamp}(r; a, b) - a}{s(a, b, n)} \right]$$

Не дифференцируемая операция

Большой диапазон значений

Аппроксимируем

Нам нужно сделать Float32 сетку, которая будет себя вести как квантованная!



Квантуем и сразу же **деквантуем**.
Диапазон чисел тот же, но мы наши
числа уже клипнули и округлили.
Т.е. **Дискретизовали**

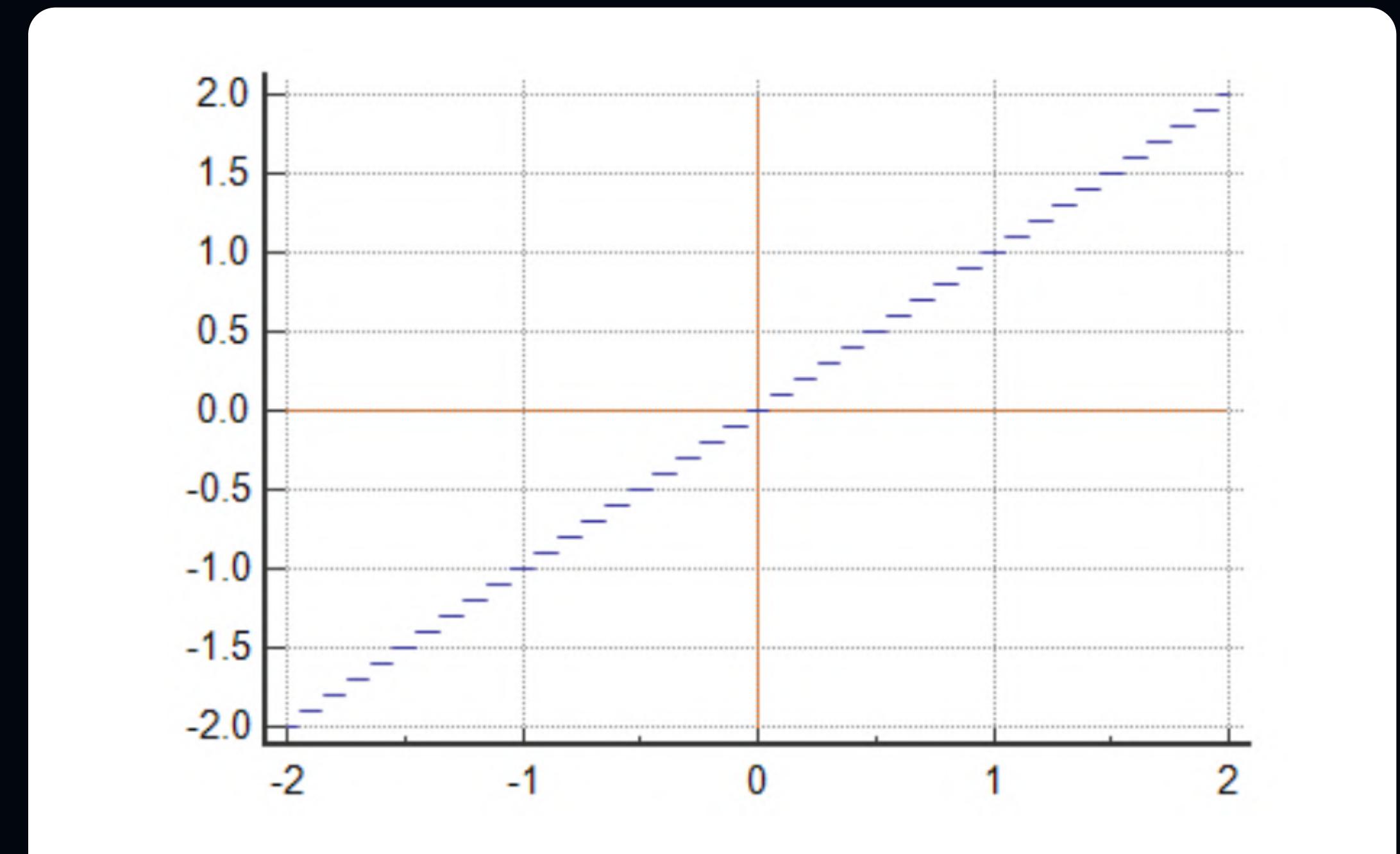
$$q(r; a, b, n) := \left\lfloor \frac{\text{clamp}(r; a, b) - a}{s(a, b, n)} \right\rfloor s(a, b, n) + a$$

Аппроксимируем

Знакомьтесь —
это Round

(

Какой у него градиент?



Аппроксимируем

Просто заменим линейной
функцией на Backward pass.

«Побросим» градиент через
не дифференцируемую функцию

```
class DifferentiableRound(torch.autograd.Function):

    @staticmethod
    def forward(ctx, input):
        return torch.round(input)

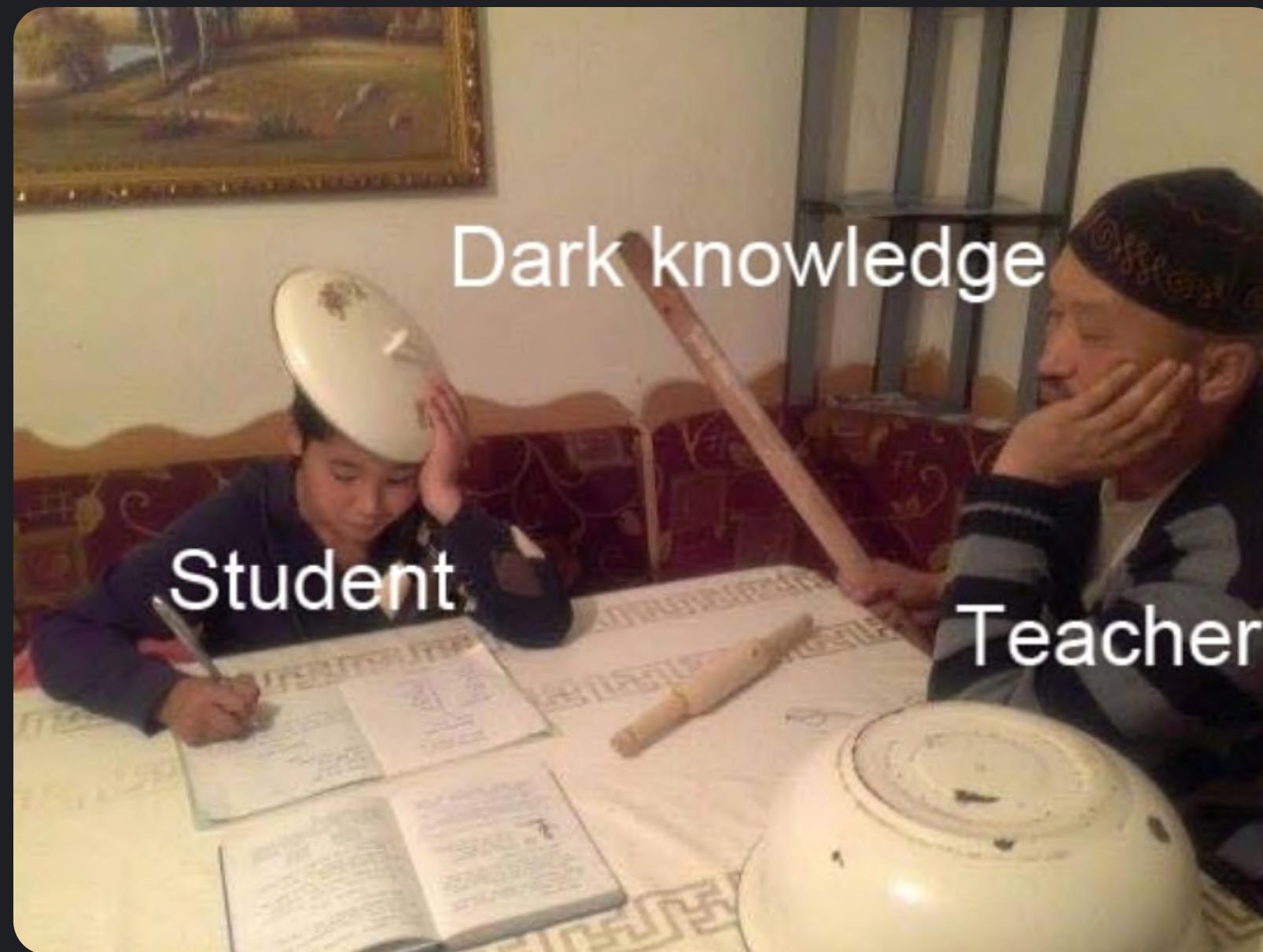
    @staticmethod
    def backward(ctx, grad_output):
        grad_input = grad_output.clone()
        return grad_input
```

→ Straight through estimator



Дальше просто
дообучаем сетку

→ Лучше дистиллировать квантованную
сетку из оригинальной



Tricks

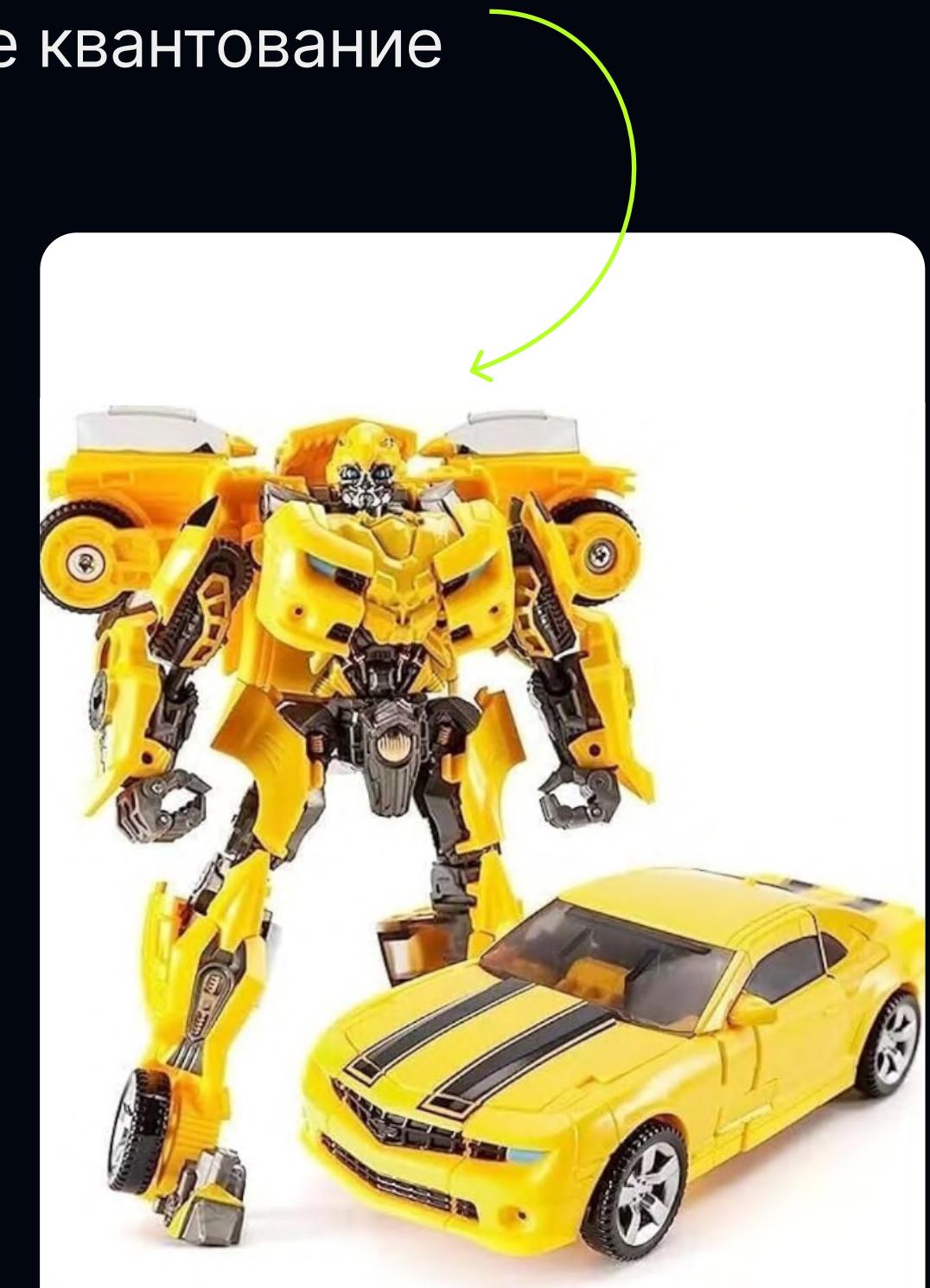
Рассмотрим специфические случаи

Dynamic Quantization

```
# original model
# all tensors and computations are in floating point
previous_layer_fp32 -- linear_fp32 -- activation_fp32 -- next_layer_fp32
/
linear_weight_fp32

# dynamically quantized model
# linear and LSTM weights are in int8
previous_layer_fp32 -- linear_int8_w_fp32_inp -- activation_fp32 -- next_layer_fp32
/
linear_weight_int8
```

Этот парень не любит десептиконов
и статическое квантование



Рассмотрим специфические случаи

Dynamic Quantization

```
# original model  
# all tensors and computations are in floating point  
previous_layer_fp32 -- linear_fp32 -- activation_fp32 -- next_layer_fp32  
    /  
linear_weight_fp32  
  
# dynamically quantized model  
# linear and LSTM weights are in int8  
previous_layer_fp32 -- linear_int8_w_fp32_inp -- activation_fp32 -- next_layer_fp32  
    /  
linear_weight_int8
```

→ А где выигрыш то? Конвертации все съедят, пороги активаций каждый раз новые пересчитываются

Как правило, огромным трансформерам хорошо из-за экономии памяти

Но часто накладные операции сводят на нет все прирост

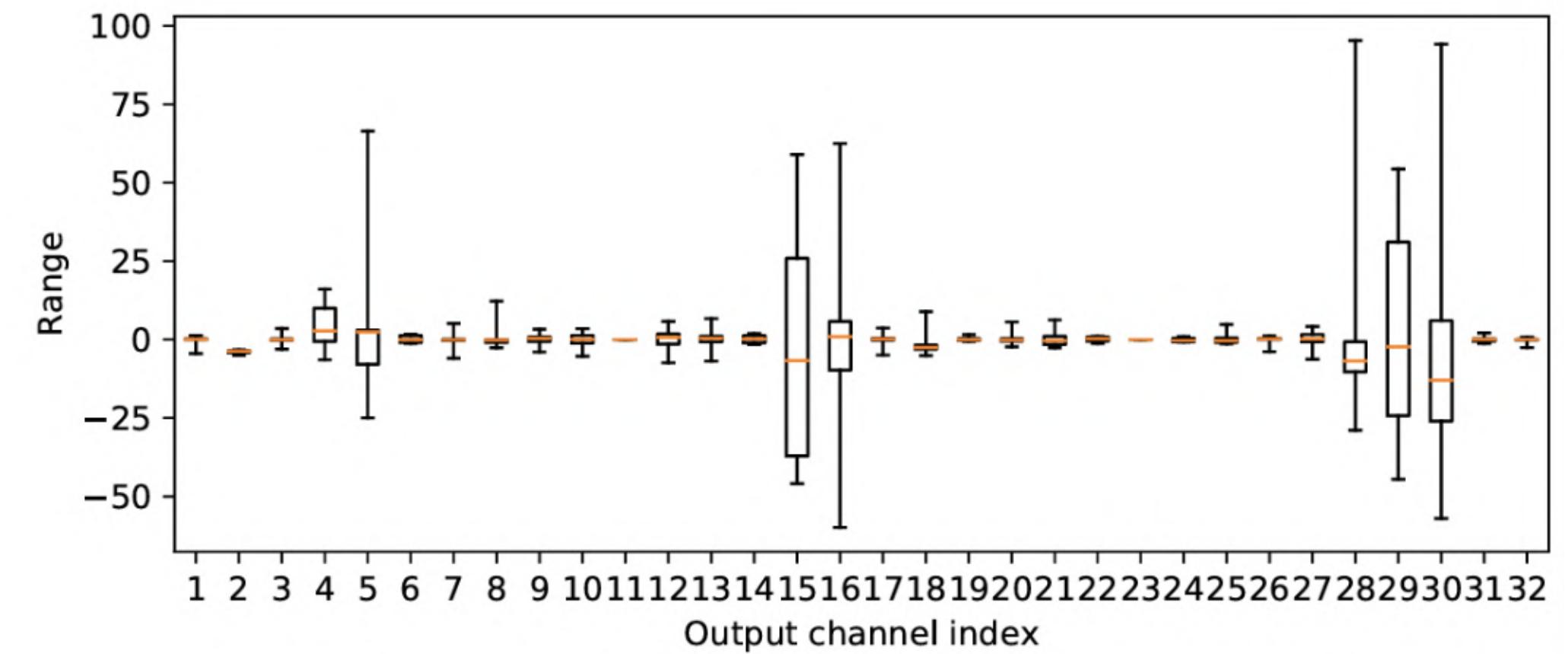
Arithmetic in the quantized model is done using vectorized INT8 instructions. Accumulation is typically done with INT16 or INT32 to avoid overflow. This higher precision value is scaled back to INT8 if the next layer is quantized or converted to FP32 for output.

Рассмотрим
специфические случаи

Rescaling

Очень разные распределения на каналах в DWS Conv,
а мы хотим на мобилки

Per tensor



Рассмотрим специфические случаи

Rescaling

Можем перемасштабиро-
вать веса в двух соседних
конвах, чтобы выровнить
распределение

$$\mathbf{y} = f(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) \quad (5)$$

$$= f(\mathbf{W}^{(2)} \hat{\mathbf{S}} f(\mathbf{S}^{-1} \mathbf{W}^{(1)} \mathbf{x} + \mathbf{S}^{-1} \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) \quad (6)$$

$$= f(\widehat{\mathbf{W}}^{(2)} \hat{f}(\widehat{\mathbf{W}}^{(1)} \mathbf{x} + \widehat{\mathbf{b}}^{(1)}) + \mathbf{b}^{(2)}) \quad (7)$$

Рассмотрим специфические случаи

Rescaling

$$\mathbf{y} = f(\mathbf{W}^{(2)} f(\mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) \quad (5)$$

$$= f(\mathbf{W}^{(2)} \hat{\mathbf{S}} f(\mathbf{S}^{-1} \mathbf{W}^{(1)} \mathbf{x} + \mathbf{S}^{-1} \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}) \quad (6)$$

$$= f(\widehat{\mathbf{W}}^{(2)} \hat{f}(\widehat{\mathbf{W}}^{(1)} \mathbf{x} + \widehat{\mathbf{b}}^{(1)}) + \mathbf{b}^{(2)}) \quad (7)$$

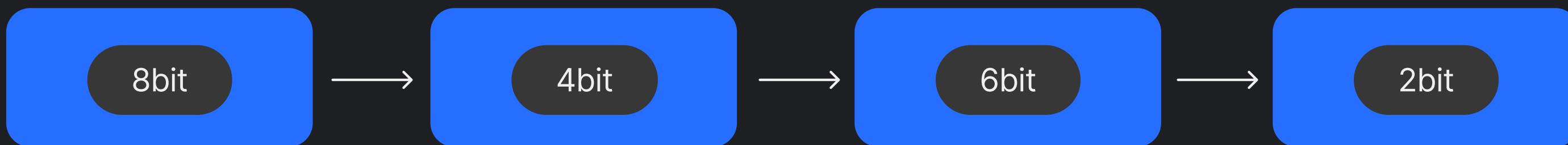
$$\widehat{\mathbf{W}}^{(2)} = \mathbf{W}^{(2)} \mathbf{S}, \quad \widehat{\mathbf{W}}^{(1)} = \mathbf{S}^{-1} \mathbf{W}^{(1)}, \quad \widehat{\mathbf{b}}^{(1)} = \mathbf{S}^{-1} \mathbf{b}^{(1)}$$

Диапазон
квантования →

$$2 \cdot \max_j |\widehat{\mathbf{W}}_{ij}^{(1)}| \xrightarrow{\mathbf{S}_i = \frac{1}{\mathbf{r}_i^{(2)}} \sqrt{\mathbf{r}_i^{(1)} \mathbf{r}_i^{(2)}}}$$

Mixed Precision Quantization

→ Используем разные битности для разных слоёв, чтобы выжать максимум



Mixed Precision Quantization

→ Как выбрать кому сколько бит?

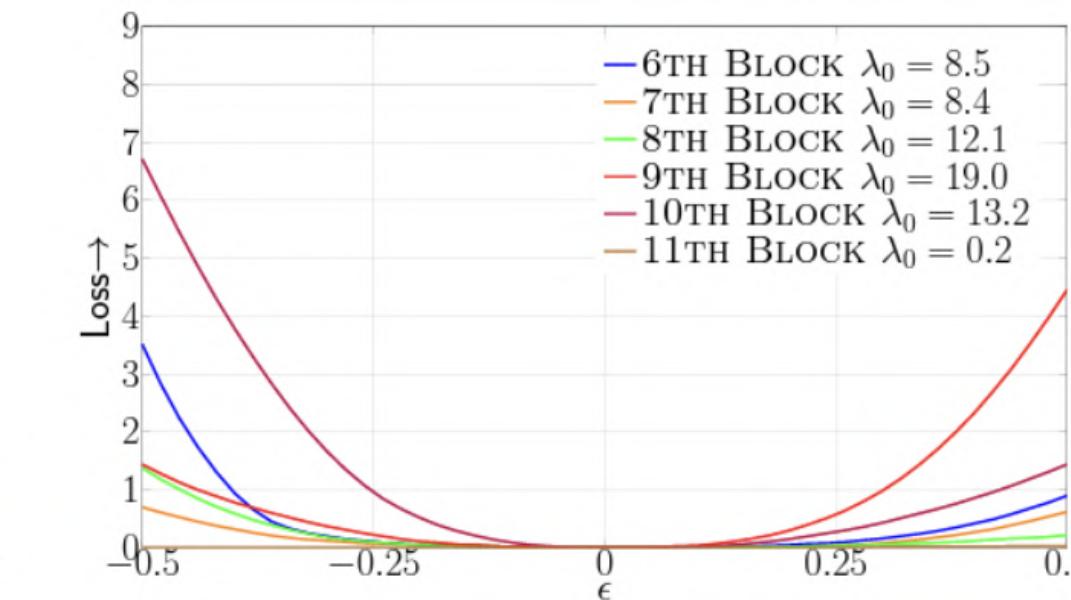
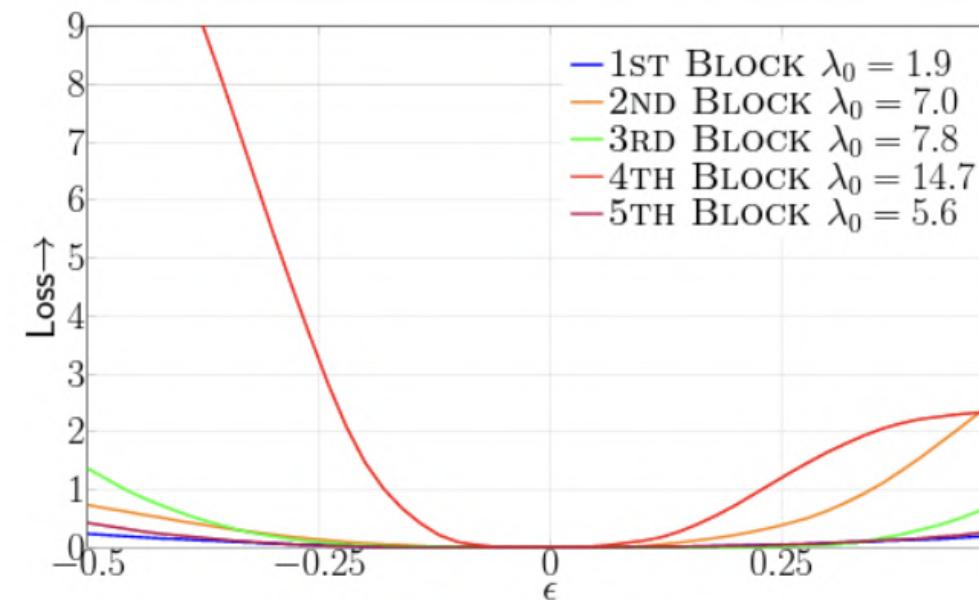
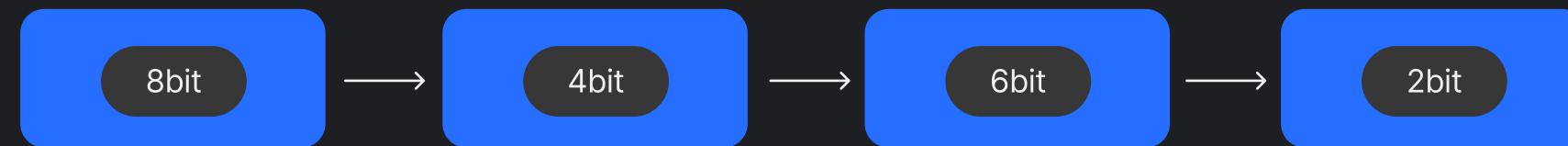
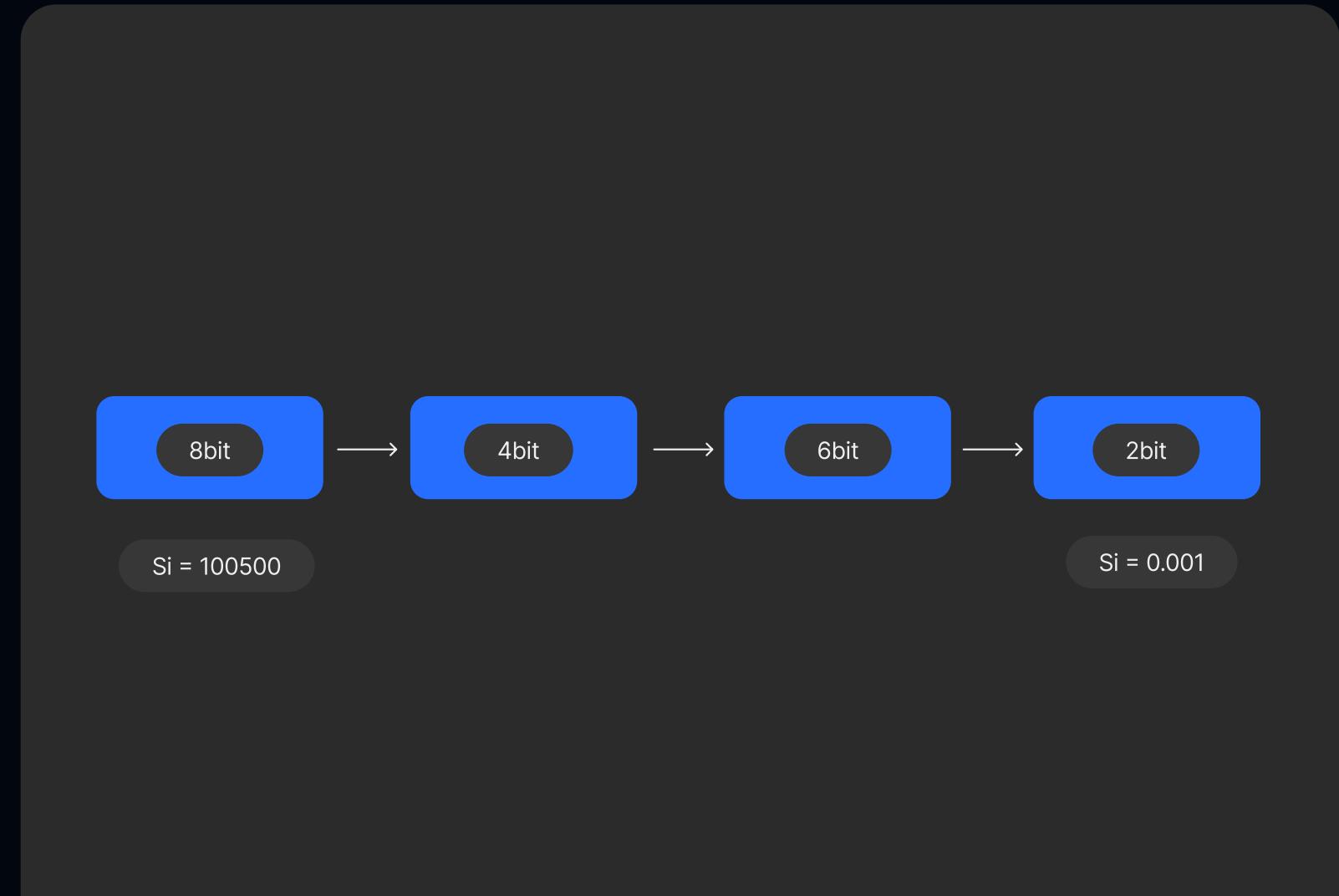


Fig. 2: 1-D loss landscape for different blocks of ResNet20 on Cifar-10. The landscape is plotted by perturbing model weights along the top Hessian eigenvector of each block, with a magnitude of ϵ (i.e., $\epsilon = 0$ corresponds to no perturbation).

Mixed Precision Quantization

Самое большое собственное число
/ потребление памяти →



→ Посчитаем собственные значения
Гессиана (и то костыльно)

$$S_i = \lambda_i / n_i,$$

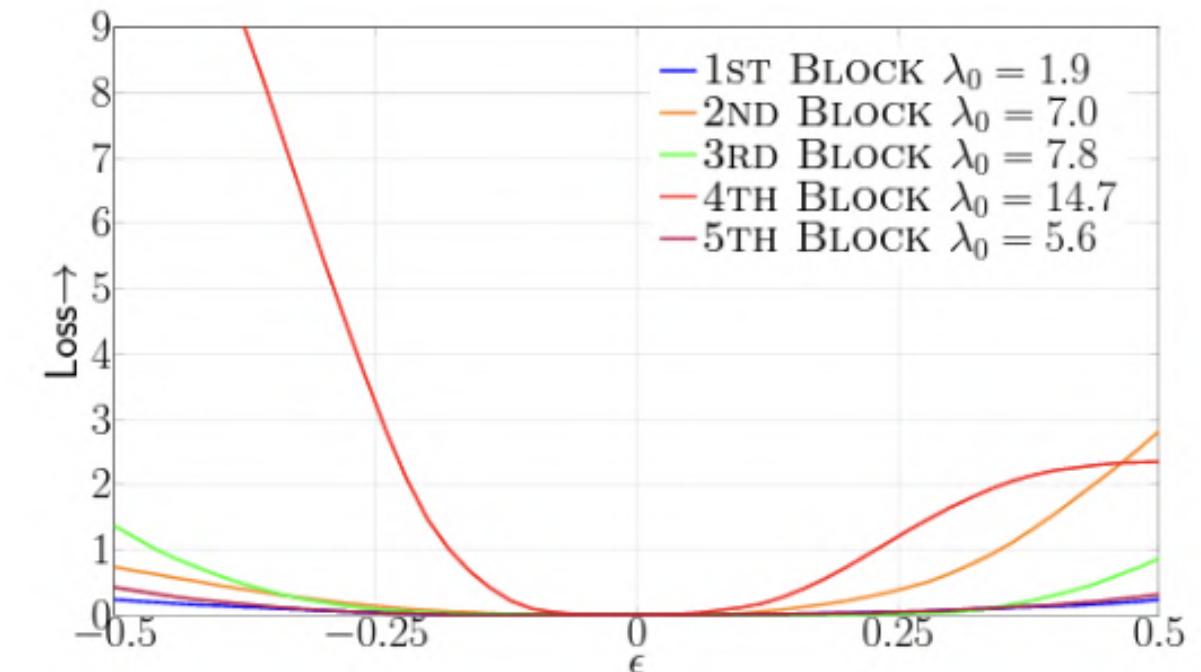


Fig. 2: 1-D loss landscape for different blocks of ResNet20 on along the top Hessian eigenvector of each block, with a magni

Mixed Precision Quantization

Дообучать слои
будем по очереди
→

$$S_i = \lambda_i / n_i,$$

→ Посчитаем собственные значения
Гессиана (и то костыльно)

$$\Omega_i = \lambda_i \|Q(W_i) - W_i\|_2^2$$

1-ый

3-ый

30-ый

100-ый

Omega_i = 100500

Omega_i = 1

Mixed Precision Quantization

$$\frac{\partial(g_i^T v)}{\partial W_i} = \frac{\partial g_i^T}{\partial W_i} v + g_i^T \frac{\partial v}{\partial W_i} = \frac{\partial g_i^T}{\partial W_i} v = H_i v,$$

Algorithm 1: Power Iteration for Eigenvalue Computation

Input: Block Parameter: W_i .

Compute the gradient of W_i by backpropagation, *i.e.*,

$$g_i = \frac{dL}{dW_i}.$$

Draw a random vector v (same dimension as W_i).

Normalize v , $v = \frac{v}{\|v\|_2}$

for $i = 1, 2, \dots, n$ **do**

// Power Iteration

 Compute $gv = g_i^T v$

// Inner product

 Compute Hv by backpropagation, $Hv = \frac{d(gv)}{dW_i}$

// Get Hessian vector product

 Normalize and reset v , $v = \frac{Hv}{\|Hv\|_2}$

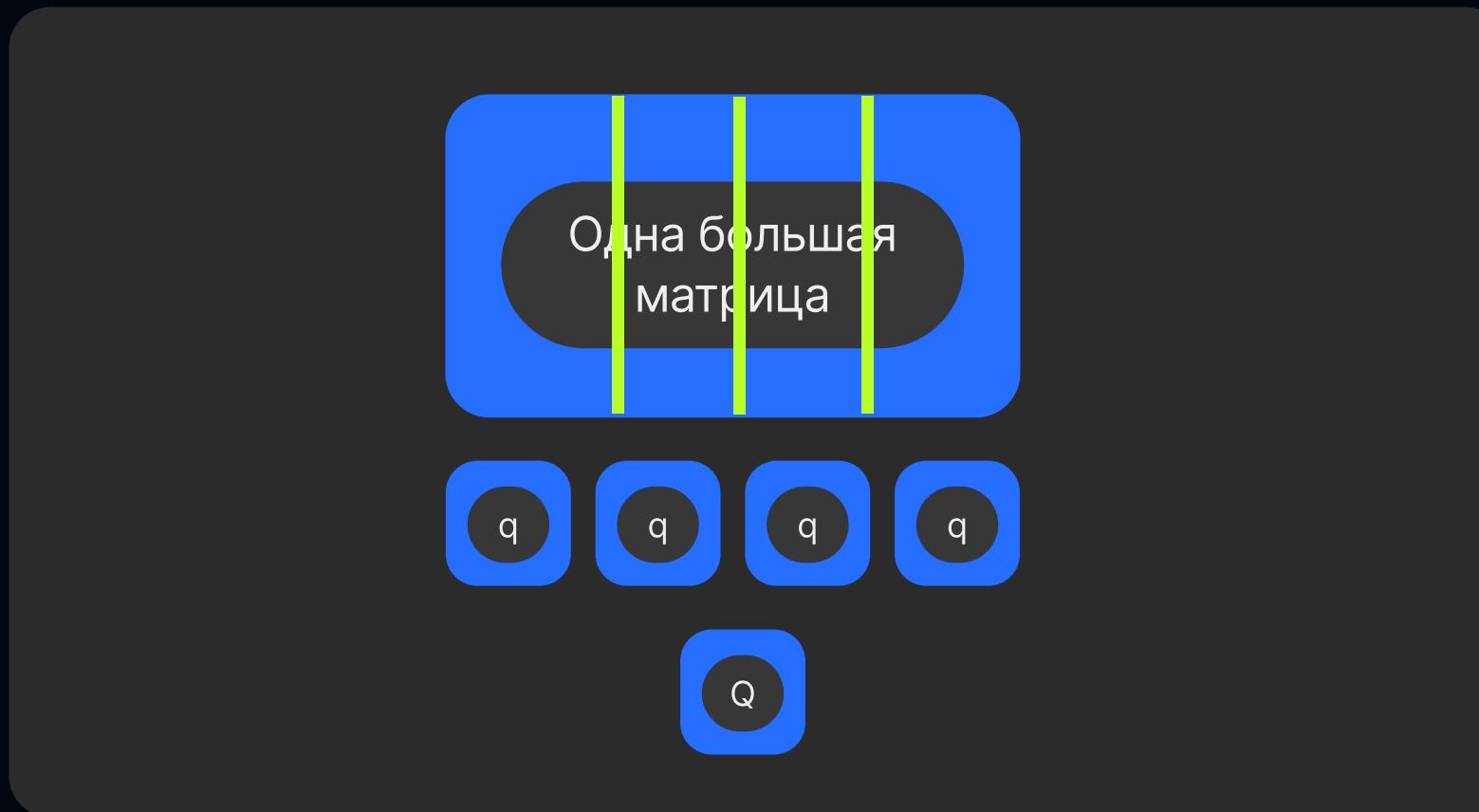
→ Посчитаем собственные значения
Гессиана через Randomized Linear Algebra

$$\text{Tr}(H) \approx \frac{1}{m} \sum_{i=1}^m z_i^T H z_i$$

Hutchinson algorithm

QLORA

→ Тюним квантованные
сетки



$$\mathbf{X}^{\text{BF16}} \text{doubleDequant}\left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}\right)$$

$$\text{doubleDequant}\left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}\right) = \text{dequant}\left(\text{dequant}\left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}\right), \mathbf{W}^{\text{4bit}}\right) = \mathbf{W}^{\text{BF16}}$$

QLORA QLORA – NF4 != FP4

- 1 Нормализуем Веса, [-1, 1]
- 2 Распихнуть веса в контейнер так, чтобы минимизировать ошибку деквантования для нормальных весов $N(0, 1)$ (один раз посчитали)
- 3 При деквантовании достаём веса из 4-битного контейнера обратно, домнажаем на скайл, которым нормализовывали

Это медленно, но жутко экономно по памяти

E NormalFloat 4-bit data type

The exact values of the NF4 data type are as follows:

```
[-1.0, -0.6961928009986877, -0.5250730514526367,  
-0.39491748809814453, -0.28444138169288635, -0.18477343022823334,  
-0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725,  
0.24611230194568634, 0.33791524171829224, 0.44070982933044434,  
0.5626170039176941, 0.7229568362236023, 1.0]
```

Fast Guide Torch

1 Делаем Prepare

2 Делаем Calibrate

3 Делаем Convert

```
49     prepared_model = deepcopy(model)
50     # Вся магия происходит здесь
51     # Torch FX позволяет нам вместо monkey patching
52     # Редактировать граф во время рантайма
53     # А значит мы можем вставить перед нодой нашу ноду для калибровки
54     prepared_model.eval()
55     example_input = next(iter(data_loader))
56     prepared_model = prepare_fx(
57         model=prepared_model,
58         qconfig_mapping=QCONFIG_MAPPING,
59         example_inputs=(example_input,),|# Click here to see the highlighted code
60     )
61     print(prepared_model.print_readable())
62     # До FX все подобные трюки использовали monkey patching
63     # И Тайные знания о структуре модели, некоторые писали свои трейсеры
64     # Потому что normally квантовать, не зная как связанные между собой слои, не получится.
65     # Например пайторчи предлагали явно самим встраивать узлы квантования и деквантования в начале и в конце сети
66     # Потому что автоматически понять что это нужно сделать нельзя было.
67
68     device = torch.device(device)
69
70     prepared_model.eval()
71     prepared_model.to(device)
72
73     with torch.no_grad():
74         # Для калибровки сильно много данных не надо.
75         for image, _ in islice(data_loader, num_batches):
76             prepared_model(image.to(device))
77
78         # Собственно в данной строке и происходит ускорение и квантование
79         # Моделька до этого была обычной, просто собирала необходимые статистики
80         # А вот теперь мы заменяем операции на квантованные в int8
81     prepared_model.cpu()
82     quantized_model = convert_fx(prepared_model)
83     return quantized_model
```

Квантование в торче всё ещё в бете, что-то меняется

The screenshot shows the PyTorch documentation website. The top navigation bar includes links for Get Started, Ecosystem, Edge, Blog, Tutorials, Docs (which is currently selected), Resources, and Git. A sidebar on the left provides links to Community, Developer Notes, Language Bindings, Python API, and specific torch modules like torch, torch.nn, and torch.nn.functional. The main content area displays the 'QUANTIZATION' page, which features a prominent red 'WARNING' banner stating 'Quantization is in beta and subject to change.' Below this, the 'Introduction to Quantization' section defines quantization as techniques for performing computations and storing tensors at lower bitwidths than floating point precision. The URL of the page is shown at the bottom: [pytorch.org/docs/stable/quantization.html#quantization-api-summary ↗](https://pytorch.org/docs/stable/quantization.html#quantization-api-summary).

[pytorch.org/docs/stable/quantization.html#quantization-api-summary ↗](https://pytorch.org/docs/stable/quantization.html#quantization-api-summary)

Для квантования надо парсить граф, поэтому для торча разные варианты квантования

	Eager Mode Quantization	FX Graph Mode Quantization	PyTorch 2 Export Quantization
Release Status	beta	prototype (maintainence)	prototype
Operator Fusion	Manual	Automatic	Automatic
Quant/DeQuant Placement	Manual	Automatic	Automatic

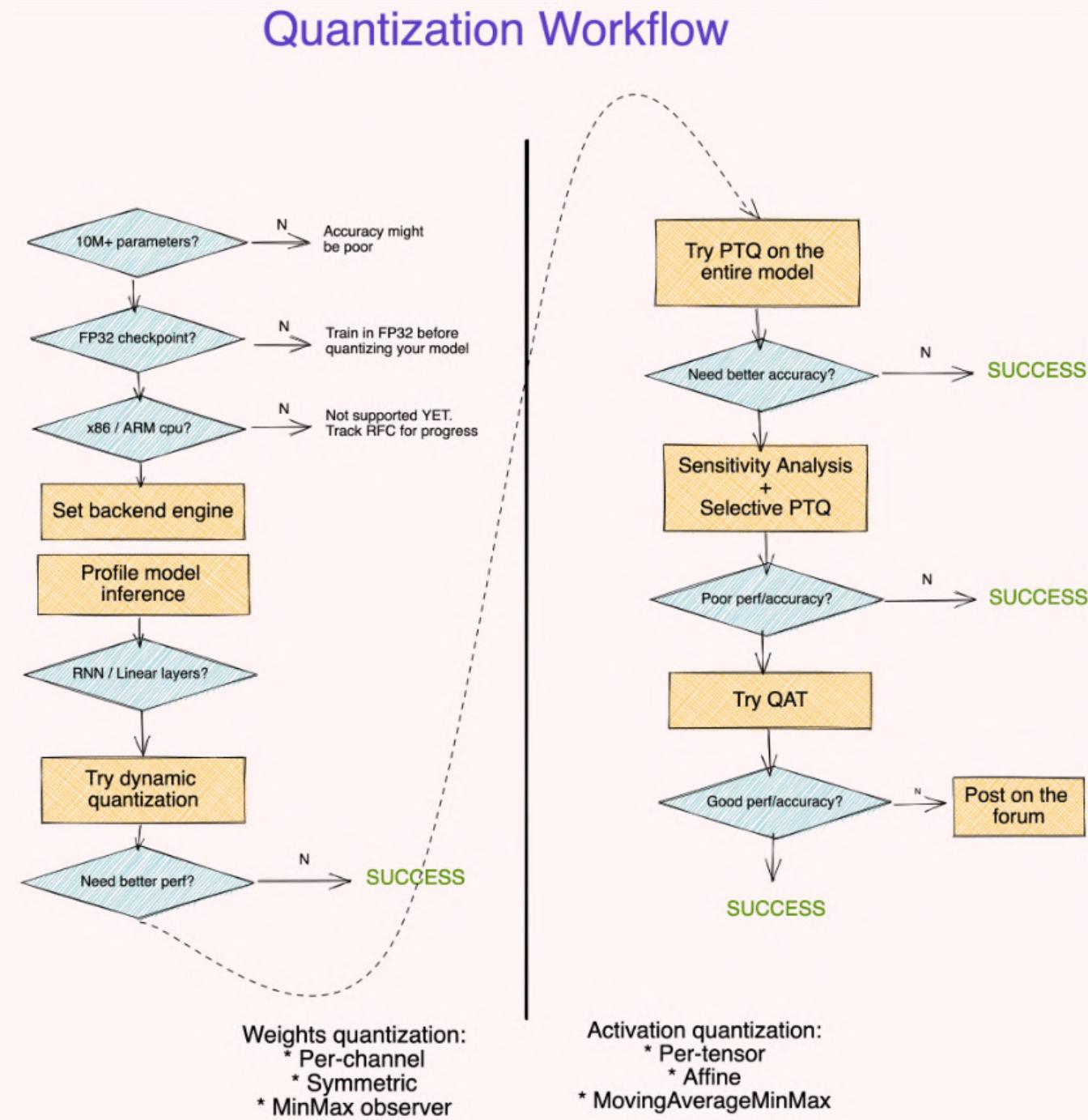
При должной удаче из торча можно экспортнуть уже дообученую после QAT модель

- 1 Схема квантования в торче совпадает с целевым фреймворком
- 2 Пороги для весов MinMax, для активаций калибровка такая же как в целевом фреймворке
- 3 Убираем FakeQuantization, отправляем модель в целевой фреймворк (как правило через ONNX) и калибуруем

При должной удаче из торча можно
экспортнуть уже дообученую после
QAT модель

У вашего фреймворка уже может быть
написан **Экспорт из торча!**

Итог



1

Сначала вкатываемся
в условный FP16, выбираем
целевой фреймворк

2

Если нужен экстрем,
квантуем. Но сначала
выбираем фреймворк!

3

Советую сразу морально
готовиться к QAT

