SCALIFY: scale propagation for efficient low-precision LLM training

Paul Balança, Sam Hosegood, Carlo Luschi, Andrew Fitzgibbon

GRAPHCORE

Contributions

Scalify formalizes tensor scaling for low-precision training and inference.

- **End-to-end scale propagation**: model forward & backward passes and optimizer update.
- **FP8** as just another datatype: `t.astype(jnp.float8_e4m3)`
- Robust **scaled FP16** master weights and optimizer state.
- Minimized **dynamic rescaling** for FP8 training.
- **Model invariance**: full computation graph transformation.

Scale propagation

In **Scalify** every tensor is a **ScaledArray** s.t.

$$X = X_d \cdot X_s$$

```
@dataclass
class ScaledArray:
    """Scaled array generic representation.
        data: Main data tensor.
        scale: Scale tensor
            Usually scalar and power-of-two.
    """
    data: Array
    scale: Array
```

End-to-end scale propagation requires JAX LAX (or Pytorch ATen) primitives scaled implementation, i.e. for each:

$$(Y_1, \ldots, Y_m) = f(X_1, \ldots, X_n)$$

an equivalent scaled operation:

$$(Y_{1,d}, Y_{1,s}, \ldots, Y_{m,d}, Y_{m,s}) \quad = f_{\text{scaled}}(X_{1,d}, X_{1,s}, \ldots, X_{n,d}, X_{n,s})$$

Scaled primitives are implemented following *unit-scaling* rules, i.e. assuming inputs $\mathbf{E}[X_{i,d}^2] \simeq 1$ then outputs $\mathbf{E}[Y_{j,d}^2] \simeq 1$.

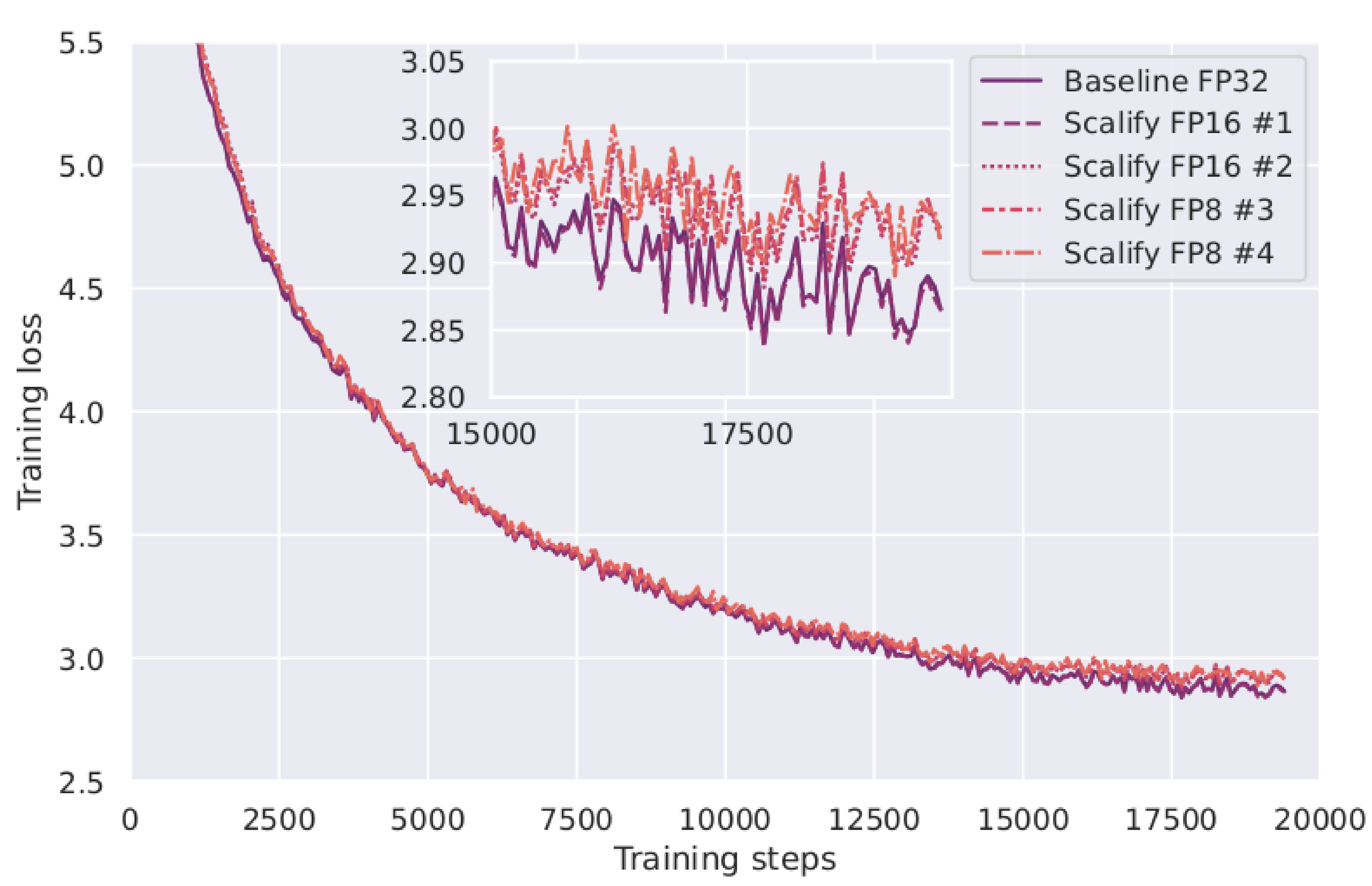
Why *unit-scaling* assumption? $\mathbf{E}[X_{i,d}^2] \simeq 1$

- Maintains high SNR for low-precision formats FP8 & FP16.
- Can represent large outliers.

Experiments

Training 168M GPT-2 model on WikiText-103 showing:

- Out-the-box replacement of FP16 loss scaling.
- FP8 forward & backward matmuls with E4M3 and E5M2.
- Scaled FP16 master weights (instead of FP32).
- Scaled FP16 optimizer state (with additional weight gradients rescaling).
- Reduced dynamic rescaling to `LayerNorm` gradients.



How to Scalify a training loop?

Seamless integration with JAX & ecosystem (Flax, Optax, ...) as a white box explicit approach to tensor scaling.

Using Scalify in practice:

- Model and optimizer states as **ScaledArray.**
- **Scalify** computational graph: forward + backward + optimizer update.
- (Optional) **dynamic rescaling** of gradients, states, ...

```
import jax_scaled_arithmetics as jsa

# Scalify transform on FWD + BWD + optimizer.
# Propagating scale in the computational graph.
@jsa.scalify
def update(state, data, labels):
    # Forward and backward pass on the NN model.
    loss, grads = jax.grad(model)(state, data, labels)
    # Optimizer applied on scaled state.
    state = optimizer.apply(state, grads)
    return loss, state

# Model + optimizer state.
state = (model.init(...), optimizer.init(...))
# Transform state to scaled array(s)
sc_state = jsa.as_scaled_array(state)

for (data, labels) in dataset:
    # If necessary (e.g. images), scale input data.
    data = jsa.as_scaled_array(data)
    # State update, with full scale propagation.
    sc_state = update(sc_state, data, labels)
    # Optional dynamic rescaling of state.
    sc_state = jsa.dynamic_rescale(sc_state)
```

General linear layer

General linear layer supporting FP8 E4M3 in forward and E5M2 in backward passes.

```
def general_linear_layer(
        x: Array, w: Array, bias: Array, *,
        fwd_dtype: DType, bwd_dtype: DType):
    # Forward casting. No-op on gradient.
    w = cast_on_forward(w, fwd_dtype)
    x = cast_on_forward(x, fwd_dtype)
    # Matrix multiplication, with
    # potentially different output dtype.
    out = jnp.dot(x, w)
    # Backward casting. No-op on activation.
    out = cast_on_backward(out, bwd_dtype)
    # Adding bias, using output precision.
    out = out + bias
    return out
```

Activation layers $\quad f(X) = X \cdot g(X)$

Custom identity scale propagation based on of the common gating decomposition of activation functions.

$$f_{\text{scaled}}(X_d, X_s) = (X_d \cdot g(X_d \cdot X_s), X_s)$$

Normalization layers

Implicitly resetting tensor scale to 1 (before affine correction).

$$\frac{X - \mathbf{E}[X]}{\sqrt{\mathbf{Var}[X] + \varepsilon}} = \frac{X_d - \mathbf{E}[X_d]}{\sqrt{\mathbf{Var}[X_d] + \frac{\varepsilon}{X_s}}}$$

$$\simeq \frac{X_d - \mathbf{E}[X_d]}{\sqrt{\mathbf{Var}[X_d] + \varepsilon}}.$$

Experiment	Matmul (GEMM)	Master state	State grads.	Optimizer state	Dynamic rescaling	Training loss
FP32 baseline #0	FP32	FP32	FP32	FP32	x	2.87 ± 0.12
Scalify FP16 #1	FP16	FP32	FP16	FP32	x	2.87 ± 0.12
Scalify FP16 #2	FP16	FP16	FP16	FP32	`LayerNorm` bwd	2.92 ± 0.12
Scalify FP8 #3	FP8	FP16	FP8	FP32	`LayerNorm` bwd	2.92 ± 0.12
Scalify FP8 #4	FP8	FP16	FP8	FP16	`LayerNorm` bwd & grads	2.93 ± 0.12

Read the paper **Try JAX Scalify**