Unit Scaling: Out-of-the-Box Low-Precision Training

Charlie Blake [1] **Douglas Orr** [1] **Carlo Luschi** [1]

1. Abstract

We present *unit scaling*, a paradigm for designing deep learning models that simplifies the use of low-precision number formats. Training in FP16 or the recently proposed FP8 formats offers substantial efficiency gains, but can lack sufficient range for out-of-the-box training. Unit scaling addresses this by introducing a principled approach to model numerics: seeking unit variance of all weights, activations and gradients at initialisation. Unlike alternative methods, this approach neither requires multiple training runs to find a suitable scale nor has significant computational overhead. We demonstrate the efficacy of unit scaling across a range of models and optimisers. We further show that existing models can be adapted to be unit-scaled, training $\text{BERT}_{\text{LARGE}}$ in FP16 and then FP8 with no degradation in accuracy.

2. Introduction

The development of algorithms that efficiently leverage available hardware has been key to the substantial advances seen in deep learning over the last decade (Sutton, 2019; Hooker, 2021).

With the increase in size of state-of-the-art models, hardware-efficiency is also motivated by the need to lower the costs of training. These have grown to become substantial—in terms of money, time, and environmental impact (Strubell et al., 2019; Chowdhery et al., 2022; Luccioni et al., 2022).

However, with the end of Moore's law and Dennard scaling (Esmaeilzadeh et al., 2011; Theis and Wong, 2017), increased transistor density can no longer be relied upon to provide a simple path towards greater efficiency, and other techniques must be leveraged. One such technique is the use of low-precision number formats. The gains to be had here are considerable: compute, memory and bandwidth usage all depend on the bit-width of a format.

[1]Graphcore Research, United Kingdom. Correspondence to: Charlie Blake <charlieb@graphcore.ai>, Douglas Orr <douglaso@graphcore.ai>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

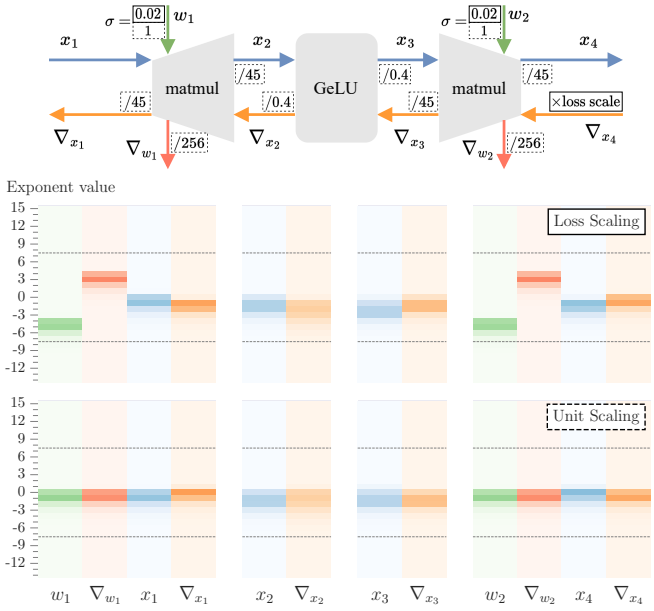

Figure 1. Above: Unit scaling of an FFN layer. We multiply each tensor by a fixed scalar to achieve consistent scale, no longer requiring a loss scale to control the scale of ∇_{x_4}. Hyperparameters here are the same as those in our $\text{BERT}_{\text{LARGE}}$ experiments (Table A.5).

Below: A histogram of exponent values at initialisation for the above FFN, with shade indicating bin density. The y-axis reflects exponent values available in FP16, while dashed lines show the max/min exponents of the FP8 E4 format of Noune et al. (2022).

Unlike inference, where integer quantisation is possible (Jacob et al., 2018), for training, floating point formats are required (Noune et al., 2022; Micikevicius et al., 2022; Kuzmin et al., 2022). The traditional approach of using 32-bit floats is being superseded by mixed precision strategies, which place many values into 16-bit formats (Micikevicius et al., 2018). Furthermore, 8-bit floating-point hardware is becoming available (Graphcore, 2022; Nvidia, 2022), with the potential for accurate 8-bit training already demonstrated (Wang et al., 2018; Sun et al., 2019; Noune et al., 2022; Micikevicius et al., 2022).

However, the use of low-precision formats introduces new difficulties, reducing the absolute range of representable values and increasing quantisation noise. Existing techniques to address these issues either introduce additional overhead or require manual tuning. An approach is needed which is both accurate and places minimal burden on the user.

Figure 2.The signal to noise ratio (SNR) of samples from a normal distribution, quantised in FP16 and FP8, as a function of the distribution's scale.

To this end, we present unit scaling a technique for model design that operates on the principle of ideal scaling at initialisation (unit variance for activations, weights and gradients). This is achieved by considering how each operation in the model affects the variance of different tensors, and introducing fixed scaling factors to counteract changes.

Empirically, we show that unit scaling aligns values much closer to the centre of the representable range than conventional loss scaling (Micikevicius et al., 2018), and removes the need for a scaling hyperparameter to be swept. None of our experiments require dynamic re-scaling of values, indicating robustness to shifting distributions during training.

2.1. Contributions

In this paper we make the following contributions:

1. We provide an analysis of how scale changes as a result of operations within a typical model, and the challenges this introduces for low-precision training.
2. We present unit scaling: a method for combating changes in scale, along with an implementation recipe and code examples.
3. We validate unit scaling empirically across a range of models and optimisers.
4. For the first time, we show training of $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ in FP16 without loss scaling. We then go a step further, training successfully in FP8, still without degradation.

We emphasise that our method works out-of-the-box, with no extra sweeps or hyperparameters, demonstrating the effectiveness of unit scaling for simplifying the use of low-precision formats.

3. Background

3.1. Floating-point formats for deep learning

Definition The conventional representation used for floating point numbers is defined by the IEEE 754 standard (IEEE, 2019). In this standard, a binary floating point format can be defined by specifying the number of exponent bits, E, and the number of mantissa bits, M. A value within such a format is defined by a sign bit, exponent and mantissa value. Each is represented using a bit-string of the requisite length (with values $b_{\text{sign}}, b_{\text{exp}}, b_{\text{mant}}$ respectively), which are interpreted as follows:

$$\begin{aligned} \text{exponent} &= b_{\text{exp}} - \text{bias}, \quad (\text{bias} = 2^{E-1} - 1) \\ \text{mantissa} &= 1 + \frac{b_{\text{mant}}}{2^M}, \\ \text{value} &= (-1)^{b_{\text{sign}}} \cdot 2^{\text{exponent}} \cdot \text{mantissa} \end{aligned}$$

There are also a small number of `special values' which denote bit-strings to which the above interpretation does not apply. These represent infinities, NaN (not-a-number) and a range of `subnormal numbers' which allow for the representation of even smaller (absolute) values.

Common floating point formats used in machine learning that implement the IEEE 754 standard are shown in Table A.1. The term *low precision* typically refers to all formats requiring fewer than 32 bits. More recently, two kinds of FP8 format have been proposed, which we term E4 and E5, i.e. $(E, M) = (4, 3)$ or $(5, 2)$. These are similar to the IEEE 754 standard, but contain differences, especially for the representation of special values. These formats are covered in detail in Appendix B.

Quantisation error Formats with more exponent bits are able to represent a wider range of values, whereas those with more mantissa bits have smaller gaps between represented values. This trade-off between range and precision can be framed in terms of *quantisation error*. This consists of two terms: the loss of accuracy due to values lying outside the absolute range of a format (overflow or underflow) is termed the *clipping error* (or saturation error), whereas the loss of accuracy due to values lying between representable numbers is termed the *rounding error*.

We demonstrate the effect quantisation error has for different formats in Figure 2. This shows the signal to noise ratio (SNR) of normally distributed values $X \sim N(0, \sigma^2)$ quantised in FP16 and FP8 as σ varies. SNR measures the faithful reproduction of an input (signal) versus the error (noise) introduced, defined as $\text{SNR} = E[X^2] / E[(q(X) - X)^2]$, where $q(\cdot)$ is the quantisation function mapping an input to the nearest representable value.

The heights of the SNR curves reflect the level of rounding error incurred by each format, and the widths reflect the range in which they are free of clipping error. With the exception of subnormal numbers (which slope away on the left-hand-side), the height of each format's SNR curve is roughly constant. This reflects the fact that exponents are evenly distributed, giving a relative rounding error that is approximately uniform.

Table 1.A comparison of techniques for low precision training.' ìndicates that this method ideally requires no tuning, but in practice may introduce hyperparameters that need to be swept.

Method	Fine-grained scaling	No tuning required	Adapts during training
Loss scaling	×	×	×
Automatic loss scaling	×	X	X
Automatic per-tensor scaling	X		X
Unit scaling	X	X	×

3.2. Trade-offs of low-precision training

Drawbacks The two common 16-bit formats, FP16 and BFLOAT16, offer different trade-offs: FP16 has more precision, but BFLOAT16 has more range. As a result FP16 is more prone to clipping error, requiring careful scaling, and BFLOAT suffers more from rounding error, which in some cases can degrade model accuracy (e.g. Rae et al., 2021).

For FP8 there is a reduction in both range and precision. For range, the same techniques used to train in FP16 are required, and for precision, the use of FP8 has thus far been restricted to only the inputs of matmul (matrix multiply) operations (Sun et al., 2019; Noune et al., 2022; Micikevicius et al., 2022), with 3 mantissa bits typically required for weights and activations, and 2 mantissa bits for gradients.

Bene ts The potential ef ciency gains when using low-precision formats are substantial. These include memory usage (often a limiting factor for large models), bandwidth usage (the main overhead for low-arithmetic-intensity ops), compute (the main overhead for high-arithmetic-intensity ops) and cross-device communication (a substantial overhead for distributed training).

3.3. Low-precision training techniques

Here we analyse existing techniques for addressing the challenges of low precision training. Table 1 provides a summary of their trade-offs and a comparison with unit scaling.

Mixed precision Mixed precision is the use of multiple number formats with different bit-widths. This differs from the traditional approach of placing all values in FP32, with Micikevicius et al. (2018) showing that most activations, weights and gradients (collectively, *tensors*) can be put in FP16 with no loss in accuracy, with the exception of master weights that are often kept in FP32. Mixed precision training is also possible in BFLOAT16 (Kalamkar et al., 2019).

By `training in FP8' we mean that matmuls are performed in FP8 (inputs are cast down to FP8, with outputs in higher precision) with wider formats typically used elsewhere, following the lead of Sun et al. (2019); Noune et al. (2022) and Micikevicius et al. (2022). FP8 reduces both precision and range, and has not generally been used for other operations as matmuls bene t most from using low-precision formats.

Mixed precision training is complementary to unit scaling—all of our experiments use some form of mixed precision.

Loss scaling Reduced range in FP16 and FP8 is particularly challenging for the backward pass, where standard model-design practices lead to gradients that risk under ow.

To combat this, Micikevicius et al. (2018) have observed that the loss can be multiplied by a scalar to increase the scale of gradients, where weight gradients are then divided by the same scalar in the optimiser. This is valid due to the linearity of the backward pass implicit in the chain rule. Loss scaling is often essential to accurate mixed precision training in FP16 and FP8.

However, there is no theoretical motivation for the choice of loss scale, which instead must be found empirically. This comes with a number of downsides. Firstly, a hyperparameter sweep must be conducted to nd the loss scale value. This can require multiple full runs, as insuf cient loss scales may only become apparent later in training. Secondly, it's not clear ahead-of-time what changes require the loss scale to be re-swept. Thirdly, as loss scaling only applies a single, global scaling factor, it has no mechanism to combat differences in scale between gradient tensors. For some models this difference may be too large for effective training.

Automatic loss scaling The dynamic adjustment of the loss scale during training is termed *automatic loss scaling* (Kuchaiev et al., 2018). This can remove the need to sweep the initial loss scale, and combats shifts in tensor distributions during training.

The combination of automatic loss scaling and automatic selection of number formats, is termed *automatic mixed precision* (PyTorch, 2023). Unit scaling doesn't specify tensors' formats, so can be used in systems that automate it.

Per-tensor scaling To address the inherent scaling dif culties of FP8 training, Micikevicius et al. (2022) propose a per-tensor scaling system, re-scaling locally based on run-time statistics.

Like unit scaling, at the beginning of training this technique may be able to achieve well-scaled tensors throughout the model. However, additional compute, memory, bandwidth and cross-device communication costs may be incurred by the recording of statistics (see Section 8 for a more detailed discussion of the potential compute overheads incurred by each of these schemes).

4. Analysis

For normally distributed tensors we use the term scale to refer to standard deviation. We observe minimal change (relative to the range of our formats) of the mean. Scale therefore characterises the probability of clipping error given a format, as too large or small a scale will lead to values that lie outside of the representable range.

Ideal scaling Given we are able to influence the scale of tensors at the start of training, the questions arises—what scale should we aim for? As suggested by Figure 2, we argue that unit scale, $= 1$ is a `sweet spot' representing a sensible compromise between several competing factors. We address this question further in Appendix C.

Is scale predictable? The ability to predict the scales of tensors in a deep learning model would give us a powerful tool to address clipping error. This is hard in general, but the problem is simpler at initialisation. Before any training steps, parameters are drawn from known initialisation distributions, so if the input distribution is known, analysis or simulation can derive the scale of each tensor.

A further simplification is to make local distributional assumptions for a single layer in the model and consider the propagation of scale through the model. This permits a methodical analysis: first, characterise the scaling effect of each operation independently; second, propagate scales through the computational graph, forwards and backwards. We provide an example of such analysis in Appendix E.1.

Scaling at initialisation Since the initial distribution of parameters is directly controlled by the model designer, the dominant approach to scaling is to select initial parameter variance to trade off forward and backward pass variance scaling (Glorot and Bengio, 2010; He et al., 2015).

Such schemes were developed to avoid exploding/vanishing gradients in deep multilayer perceptrons. As such, they do not seek to constrain the scale of parameters and parameter gradients. They are also limited to computations where scale factors can be moved into trainable parameters.

Example: BERT (Devlin et al., 2019) BERT's initialisation scheme does not use the rules of Glorot and Bengio (2010), instead initialising all non-bias parameters from $N(0; (0{:}02)^2)$. It also adopts a scaling factor from the Transformer (Vaswani et al., 2017), which scales the product of activation matrices $QK^>, Q; K \; 2 \; R^{s \; d}$ by $1=\sqrt{p}\overline{d}$.

We instrument the model to record histograms of all tensors at the start and end of training, and plot the results in Figures A.4 and A.6. In light of this analysis, we can understand loss scaling as simply enacting a shift of the x and grad_w histograms by $\log_2(\text{loss scale})$ bits to the right, trading off under flow and over flow globally across gradient tensors.

BERT with loss scaling illustrates the drawbacks of having just three scales: weight initialisation scale, loss scale, and $QK^>$ scale. These are not sufficient to centre most tensors' distributions in the representable range.

5. Unit Scaling

Based on our analysis of the scaling within typical models and the limitations of existing methods for managing scale, we present *unit scaling*. A model is said to be unit-scaled if its activations, weight and gradients have approximately unit variance at initialisation.

We achieve this by inserting scaling factors into the forward and backward passes. Like loss scaling, our modification of the backward pass still ensures correct gradients up to a constant multiplicative factor. However, unlike loss scaling, unit scaling determines these scales based on a set of rules for each operation, rather than a single hyperparameter to be found empirically, or via an adaptive algorithm.

The scales chosen enable each operation to approximately preserve the variance of its inputs. This effect then propagates through the model, giving global unit-scaling. By concentrating values in approximately the centre of the exponent range at initialisation, we give tensors headroom to potentially shift during training without going out-of-range.

Unit scaling does not address the issue of adapting scales during training. We anticipate that unit scale is sufficient to avoid numerical instability for many models, and observe this in all our experiments. We leave to further work a full investigation of where dynamic re-scaling is required, and how to integrate such a scheme into unit scaling.

5.1. A framework for scaling computational graphs

Computational Graphs We take our model to be represented by the differentiable function $f_{\text{model}}(x_1; \ldots; x_m)$, itself a composition of differentiable functions $f_1; \ldots; f_n$.

We can describe the structure of such a model using a directed acyclic graph (DAG) denoted $G = (V; E)$, with the property that the vertex $x_i \; 2 \; V$ corresponds to the function f_i for each $i \; 2 \; f1; \ldots; ng$, and where the vector-valued

output of function f_a used as an input to function f_b is represented by the edge $(v_a, v_b) \in E$.

This kind of graph is commonly known as a computational graph, with vertices as nodes and their corresponding functions as ops.

Forward and backward graphs We refer to the computational graph corresponding to f_{model} as the forward graph.

In deep learning we typically apply reverse-mode automatic differentiation to the forward graph to create a second computational graph whose output nodes represent the partial derivatives of the model with respect to its inputs: $\frac{\partial f_{model}}{\partial x_i}, \forall i \in [1..m]$. We call this the backward graph.

The backward graph mirrors the structure of the forward graph, but with edge directions reversed. Thus each op f in the forward graph corresponds to a new f_{grad} in the backward graph. This op computes the gradient of the model up to f by calculating the product of the incoming gradient g from the previous grad op and the partial derivatives of f evaluated at its inputs: $f_{grad}(x_1, ..., x_k, g)_j = g^\top \frac{\partial f}{\partial x_j}(x_1, ..., x_k), \forall j \in [1..k]$.

Scaled ops Given an op $f(x_1, ..., x_k)$, we define the scaled op $f^*(x_1, ..., x_k, \alpha, \alpha_1, ..., \alpha_k)$ with scaling factors $\alpha, \alpha_1, ..., \alpha_k \in \mathbb{R}^+$, such that:

$$f^* = \alpha f(x_1, ..., x_k),$$

$$f^*_{grad}(x_1, ..x_k, g)_i = \alpha \alpha_i f_{grad}(x_1, ..x_k, g)_i, \forall i \in [1..k].$$

Proposition 5.1. For any scaled op, there is an equivalent unscaled op with the same training dynamics under a first-order optimiser.

We demonstrate this for SGD and Adam in Appendix E.2.

Scaled computational graph A scaled computational graph is one where every op f in the forward graph is replaced by a scaled equivalent f^*, with the backward graph then generated to produce f^*_{grad} for each f^*_{grad}, using any choice of scaling factors.

If we can show that a scaled computational graph represents a scaled op, by Proposition 5.1, we are within a reparameterisation of regular training. Unfortunately, this is not true for scaled computational graphs in general, for example $h(x) = x + f(x)$ is not a scaled op for some choices of the scaled op f^* and when $\alpha \neq 1$ (see Appendix E.3).

Constraint-scaled computational graphs We denote the set of edges in the forward graph that are cut-edges[1] as $C \subseteq E$. A constraint-scaled computational graph is a scaled computational graph where we restrict the scaling factors of ops that consume non-cut-edge variables in the following way: for any edge $e \notin C$, we require the op consuming the variable x_e to have scaling factors $= \alpha_e$.

Theorem 5.2. A constraint-scaled computational graph itself represents a scaled op.

Proven in Appendix E.4. This is sufficient to show that we've achieved the property we set out to: valid gradients, up to a constant multiplicative factor.

5.2. A scaling strategy for unit variance

Unit scaled computational graphs We define a unit-scaled computational graph as an instance of a constraint-scaled computational graph, with scales selected via the following:

1. Initially set aside any scale constraints, and calculate the scaling factors that give each op expected unit variance outputs (this process is covered below).
2. Now resolve any scale constraints by taking each constrained group $\{\alpha_1, ..., \alpha_l\}$ and selecting the geometric mean $(\alpha_1 \cdots \alpha_l)^{\frac{1}{l+1}}$.

This compromise is necessary to ensure valid gradients, but diverges from strict unit scale. In practice though, we observe that the scales going into our geometric mean are often similar enough to preserve approximate unit variance.

Selecting scaling factors Assuming unit-scaled inputs to $y = f(x_i, ..., x_k)$, derive the output scale σ_Y and set the forward scaling factor $\alpha = 1/\sigma_Y$. Repeat this process for $x'_i = f_{grad}(...)_i, \forall i \in [1..k]$, to obtain the gradient scale $\sigma_{x'_i}$ and set the backward scaling factor $\alpha_i = 1/\sigma_{x'_i}$. (See Table A.2 for the scaling factors of common ops.)

Note that our assumption of unit-scaled inputs above is justified by inductive reasoning: we assume that a given op has unit-scaled inputs, which allows us to unit scale its outputs. In this way, unit scale propagates through the graph. The base-cases here are the model's initial inputs, corresponding to parameters and input data. As we initialise parameters to have unit scale, the only extra step we require is to normalise the input data.

5.3. Weighted addition

For the most part, the scale of tensors at initialisation in unscaled deep learning models does not play a critical role. A notable exception is when tensors of different scales are added, for example residual layers, losses and positional encodings.

[1] A cut-edge is an edge in the equivalent undirected graph where the number of connected components increases upon its deletion.

```
def scaled(X, alpha=1, beta=1):
  # Forward: Y = X * alpha
  # Backward: grad_X = grad_Y * beta

def scaled_projection(X, W):
  (b, _), (m, n) = X.shape, W.shape
  alpha = beta_X = (m * n) ** -(1/4)
  beta_W = b ** -(1/2)
  X = scaled(X, beta=beta_X)
  W = scaled(W, beta=beta_W)
  return scaled(matmul(X, W), alpha)
```

```
class FFN(nn.Module):
  def __init__(self, d, h):
    super().__init__()
    self.norm = LayerNorm(d)
    sigma = (d * h) ** -(1/4)
    self.W_1 = Parameter(
      randn(d, h) * sigma)
    self.W_2 = Parameter(
      randn(h, d) * sigma)

  def forward(self, X):
    Z = self.norm(X)
    Z = matmul(Z, self.W_1)
    Z = gelu(Z)
    Z = matmul(Z, self.W_2)
    return X + Z
```

```
class ScaledFFN(nn.Module):
  def __init__(self, d, h, tau):
    super().__init__()
    self.norm = ScaledLayerNorm(d)
    self.W1 = Parameter(randn(d, h))
    self.W2 = Parameter(randn(h, d))
    self.tau = tau

  def forward(self, X):
    a = (1 - self.tau) ** (1/2)
    b = self.tau ** (1/2)
    Z = self.norm(scaled(X, beta=b))
    Z = scaled_projection(Z, self.W1)
    Z = scaled_gelu(Z)
    Z = scaled_projection(Z, self.W2)
    return X * a + scaled(Z, b)
```

Figure 3.PyTorch examples. Left: Scaled projection op, which implicitly constrains α_X. Center vs Right: Unscaled vs scaled Transformer FFN layers. Changes: a) initialise weights with unit scale, b) replace unscaled with scaled ops, c) replace residual add with interpolation according to τ, moving the backward pass scale as in Section 5.2. See Figure A.2 for the implementation of scaled_matmul and further ops.

If we naïvely convert these add ops to unit-scaled equivalents, they place equal weight on their inputs, which can be detrimental to performance. We propose using weighted add (Table A.2) to resolve this. This introduces new hyperparameters into the model, which can be chosen by design principle, empirically by sweep, or selected to match a reference model (see Appendix H).

For residual layers, there are existing design principles in literature. We consider the following residual layers based on NF-ResNets (Brock et al., 2021):

default: $x_{l+1} = x_l + f(x_l)$ (not suitable for unit scaling)

xed ( ): $x_{l+1} = \sqrt{1}\, x_l + \sqrt{\ }\, f(x_l)$

running-mean: $x_{l+1} = \sqrt{l/(l+1)}\, x_l + \sqrt{1/(l+1)}\, f(x_l)$

An issue with these weighting rules is that they may produce small gradient scales in the residual branch, which isn't a cut-edge so can't be independently rescaled. To resolve this, we perform a special-case rewrite to replace $f(x)$ with $\mathrm{id}(f(\mathrm{id}(x; 1; \ )); \ ; 1)$, where $\mathrm{id}(x; \ ; \ )$ is the scaled identity function. This maintains unit scale for the backward pass f_{grad}, while preserving G as a scaled op.

5.4. Recipe

We now outline a high-level recipe for a unit-scaled model:

1. Initialise non-bias parameters with unit variance.
2. Calculate scaling factors for all scaled ops.
3. Identify non-cut-edges, and constrain the ops consuming them to have $=$ by taking the geometric mean.
4. Replace adds with weighted adds.

Unconstrained scaling factors are as outlined in Appendix G. Identifying cut-edges may sound challenging, but in practice is similar across models. The set of cut-edges commonly contains parameters and any encoder/decoder layers (anything before/after a stack of residual layers). After applying this recipe, training and inference proceed as usual.

To align a unit-scaled model with an existing model, there are some additional considerations. We cover these in Appendix H. One notable difference is that unit scaled models have different effective optimiser step sizes across their parameters versus unscaled models.[2] While this difference can be compensated by per-tensor step size modifiers, it means that the training dynamics may be different by default.

5.5. Example

Using the unit scaling recipe, we first build a scaled op, and then a full scaled layer. Consider a scaled projection op with learnable weights:

$$\mathrm{matmul}(X; W) = X\,W$$
$$\mathrm{matmul}_{\text{grad}}(X; W; G)_1 = {}_1\, G\, W^{>}$$
$$\mathrm{matmul}_{\text{grad}}(X; W; G)_2 = {}_2\, X^{>} G;$$

for input $X \in \mathbb{R}^{b\ m}$, weight $W \in \mathbb{R}^{m\ n}$, output $\mathbb{R}^{b\ n}$ and incoming gradients $G \in \mathbb{R}^{b\ n}$.

Assuming large b, m, n, the analysis of Appendix E.1 gives unconstrained scaling factors $= m^{-\frac{1}{2}}$, $_1 = n^{-\frac{1}{2}}$, $_2 = b^{-\frac{1}{2}}$. Typically, the edge connecting the weights W is a cut-edge, while the edge connecting in the inputs X is not. Given that assumption, we constrain $= {}_1$, satisfied by setting both to the geometric mean of the unconstrained values: $= {}_1 = (m\ n)^{\frac{1}{4}}$. We leave $_2$ unchanged.

We show code for the above in Figure 3, which also gives a scaled layer for the Transformer FFN of Figure 1.

[2] For instance, a larger effective step size for bias parameters when using unit scaling. Effective step size considers the effect of an optimiser update on model output, rather than parameters.

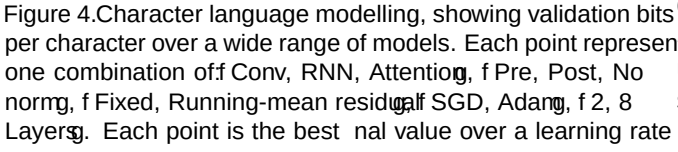

Figure 4. Character language modelling, showing validation bits per character over a wide range of models. Each point represents one combination of: {Conv, RNN, Attention}, {Pre, Post, No norm}, {Fixed, Running-mean residual}, {SGD, Adam}, {2, 8 Layers}. Each point is the best final value over a learning rate sweep.

6. Results

6.1. Character language modelling

Experimental Setup To evaluate unit scaling for multiple model architectures and optimisers, we perform small-scale experiments on WikiText-103 raw character language modelling (Merity et al., 2017). We train causal language models, using cross entropy loss during training and evaluate on bits per character (BPC). All models follow the pattern of a Transformer decoder layer (Vaswani et al., 2017), with the following variants:

Sequence layer type: Attention, RNN and Convolution.

Norm placement: PreNorm, PostNorm and NoNorm.

Residual scaling: default, fixed and running-mean (as defined in Section 5.2).

Over the product of these settings, we compare the performance of regular (baseline) and unit scaling in both FP32 and FP16. For this, we also evaluate the regular model in FP16 with loss scaling. For full hyperparameters and details, see Appendix J.1.

Results The above configurations amount to a 2092-run sweep, the results of which are shown in Figure 4. First, these demonstrate the need for scaling when using FP16. This is due to gradient underflow, since loss scaling with a factor of 2048 resolves the issue. Second, they demonstrate that unit scaling, despite changing the training behaviour of the model beyond just numerics, matches or even slightly improves upon baseline performance in almost all cases. Finally, they show that no tuning is necessary when switching unit scaling to FP16.

We also explore the effect of using different residual scaling schemes, with results shown in Figure A.3. We find that performance is not sensitive to the choice of scheme, and suggest that running-mean or fixed are reasonable choices when using unit scaling.

6.2. Masked language modelling

Experimental setup To evaluate unit scaling against a standard baseline known for challenging numerics, where loss scaling is conventionally required (Lin et al., 2020), we train unit-scaled $\text{BERT}_{\text{BASE}}$ and $\text{BERT}_{\text{LARGE}}$ models.

We use the standard BERT masked language model pre-training objective over English Wikipedia articles, and demonstrate downstream performance on SQuAD v1.1 and SQuAD v2.0 (Rajpurkar et al., 2016; 2018). We follow the unit scaling recipe, along with our guide on aligning a unit scaled model with a regular model (Appendix H).

Full hyperparameters and details are covered in Appendix J.2. Note that we do not sweep any additional hyperparameters for our unit-scaled BERT (or character language models) relative to the baselines.

Results We report our results in Table 2. For unit scaling in FP16, we are able to attain the same performance as the baseline model, and whereas the baseline requires sweeping a loss scale, unit scaling works in all cases out-of-the-box. Due to differences in the effective optimiser step size across parameters (Section 5.4), our regular and unit-scaled models aren't exactly equivalent, but deviations in their downstream performance are minor ($\text{BERT}_{\text{BASE}}$ is slightly below the baseline, and $\text{BERT}_{\text{LARGE}}$ is slightly above).

For FP8, we build on the results of Noune et al. (2022) who demonstrate the training of loss-scaled BERT in FP8 with no degradation relative to FP16. We show that the same can also be achieved with unit scaling, with no additional techniques required to make FP8 work over FP16—we simply quantise our matmul inputs into FP8 and are able to train accurately. These results represent the first time $\text{BERT}_{\text{BASE}}$ or $\text{BERT}_{\text{LARGE}}$ have been trained in either FP16 or FP8 without requiring a form of loss scaling.

To highlight the precise effects of unit scaling, we show histograms for activations, weights and gradients for unit-scaled FP16 BERT. These can be found in Figures A.5, A.7, alongside equivalent plots for a regular FP16 BERT.

Table 2. Downstream performance of regular and unit-scaled BERT models. We pretrain 3 models for every model-method-format combination, then fine-tune 5 SQuAD v1.1 and 5 v2.0 runs for each (i.e. 15 runs per downstream task). The values shown represent the mean across the 15 runs, with ± indicating the standard deviation across the mean scores of the 3 sub-groups. † published result from Devlin et al. (2019). ‡ published result from Noune et al. (2022); this model also adds an activation scale alongside the loss scale.

Model	Method	Precision	SQuAD v1.1 EM	SQuAD v1.1 F1	SQuAD v2.0 EM	SQuAD v2.0 F1
Base	No Scaling †	FP32	80.8	88.5	—	—
	Loss Scaling	FP16	80.55 (±0.16)	88.19 (±0.16)	73.36 (±0.27)	76.47 (±0.23)
	Unit Scaling	FP16	79.96 (±0.31)	87.86 (±0.44)	72.31 (±0.60)	75.70 (±0.53)
	Unit Scaling	FP8	80.15 (±0.18)	88.04 (±0.12)	72.28 (±0.02)	75.67 (±0.01)
Large	No Scaling †	FP32	84.1	90.9	78.7	81.9
	Loss Scaling	FP16	84.23 (±0.20)	90.93 (±0.14)	77.52 (±0.63)	80.54 (±0.61)
	Loss Scaling ‡	FP8	83.40 (±0.23)	90.69 (±0.16)	—	—
	Unit Scaling	FP16	85.67 (±0.10)	92.14 (±0.08)	79.94 (±0.10)	82.97 (±0.09)
	Unit Scaling	FP8	85.22 (±0.03)	91.77 (±0.10)	79.29 (±0.31)	82.29 (±0.29)

The code used in these experiments can be found at https://github.com/graphcore-research/unit-scaling-demo, alongside a separate notebook implementing a unit-scaled NanoGPT model. We recommend this resource for those looking to understand unit scaling through a simple example implementation.

For those interested in using unit scaling in their own models, we also provide a PyTorch library: https://graphcore-research.github.io/unit-scaling. The documentation includes a practical guide to developing and optimising a unit-scaled model. This implementation should be considered a definitive reference for unit scaling.

7. Related Work

Variance scaling analysis Klambauer et al. (2017) and Peiwen and Changsheng (2022) propose activation functions that encourage unit-variance activations and gradients, which are complementary to unit scaling. He et al. (2016) introduce residual networks, using skip connections and explicit normalisation to stabilise forward and backward passes. Variants on normalisation (Ioffe and Szegedy, 2015; Ba et al., 2016; Labatie et al., 2021; Salimans and Kingma, 2016) are complementary to unit scaling, which considers the norm of the gradients as well as activations and does not constrain activation norms after initialisation. Alternative residual schemes (Zhang et al., 2019; Brock et al., 2021) can be incorporated into unit-scaled models, although the residual layer output variance should not be allowed to grow with depth.

The reparameterisation implied by unit scaling is also used by Jacot et al. (2018), later broadened by Yang and Hu (2020) and exploited by Yang et al. (2022) in their work analysing the training behaviour of deep networks. Motivated by low-precision computation rather than training dynamics, unit scaling applies scaling factors locally throughout the compute graph, but the effect on training hyperparameter scaling is similar.

FP8 inference Although there has been little hardware support for FP8 training, accelerated 8-bit inference is increasingly common via the use of integer quantisation (Jacob et al., 2018) to the INT8 format. This process typically results in degraded accuracy, requiring additional techniques such as quantisation-aware training (see Nagel et al. (2021) for a thorough discussion on this topic). Though recent efforts have been made to improve efficient INT8 quantisation (Yao et al., 2022; Park et al., 2022; Dettmers et al., 2022; Xiao et al., 2022), the use of FP8 enables accelerated inference in the same format as training, promising a substantial improvement in the simplicity and accuracy of 8-bit inference (Kuzmin et al., 2022).

8. Discussion

Compute overhead Unit scaling relies solely on the addition of scaling operations of the form X, where $$ is a fixed scalar and X is a tensor. These scaling factors can be fused into the preceding ops (e.g. via torch.jit, torch.compile or jax.jit). By doing this we observe that the increase in memory-access cost is negligible. For models with reasonably large hidden sizes, the compute overhead is also minimal. For example, the FLOPs required to train our unit-scaled $\text{BERT}_{\text{LARGE}}$ are only 0.2% greater than the baseline (explained further in Appendix I.2). Basic loss scaling operates on a similar principle, and only introduces a single scaling factor. From this we conclude that both techniques have low overall overhead, assuming a fused implementation.

Automatic loss scaling has an additional feature which increases overhead: its requirement to occasionally discard batches. This assumes that re-scaling is determined by tracking gradient over ows (the standard approach, as used in PyTorch (2023)). When over ows occur, batches must not be used to update parameters. The overhead of dropping batches is tolerable for FP16 but may not be for FP8 (Micikevicius et al., 2022).

Proposed automatic per-tensor scaling schemes take a different approach, and have potential to add overhead in other areas (how much depends largely on software and hardware characteristics). Micikevicius et al. (2022) reject scaling based on gradient over ows, instead opting for heuristics based on properties of the tensors being scaled. Their preferred training heuristic is not speci ed, but for inference they choose between max, percentile, and minimum MSE methods. These approaches trade-off overhead for accuracy. At one extreme, max is likely easy to fuse but may be distorted by outliers; at the other extreme minimum MSE may be more robust but is challenging to implement ef ciently (e.g. Sakr et al. (2022)). Distributed training adds further challenges, potentially requiring the communication of statistics across devices to keep scales synchronised.

It remains to be seen whether effective automatic scaling methods can be implemented ef ciently given these complexities. This will likely be an important future research objective. In contrast unit scaling, with xed precomputed scaling factors, offers a simpler alternative.

Broader impact The potential for unit scaling to simplify the use of 8-bit number formats may lead to increased adoption, and in turn facilitate training larger models. At scale, new capabilities emerge (Wei et al., 2022), potentially exacerbating known harms (Weidinger et al., 2021) such as toxicity (Nadeem et al., 2020), misinformation (Lin et al., 2021), privacy concerns (Carlini et al., 2021) and environmental damage (Strubell et al., 2019). To mitigate these outcomes, a variety of methods have been proposed, including reinforcement learning from human (Ouyang et al., 2022) or AI (Bai et al., 2022) feedback, anti-experts (Liu et al., 2021) and baked-in safety models (Xu et al., 2020), all of which are applicable to unit-scaled models.

Conclusion We have demonstrated that unit scaling addresses the complexities of low-precision training, providing a simpler and more granular solution. This is demonstrated by our training of $\text{BERT}_{\text{LARGE}}$ for the rst time without loss scaling, in FP16 and even FP8. The community's transition to FP8 training will see new capabilities emerge as a result of improved ef ciency, and this transition can be accelerated by unit scaling.

Acknowledgements

We would like to thank the following people for their contributions to the paper at the various stages of its development: Daniel Justus, Alberto Cattaneo, Andrew Fitzgibbon, Paul Balanca, Luke Prince, Ivan Chelombiev, Luka Ribar and Zach Eaton-Rosen.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. arXiv preprint arXiv:1607.06450, 2016.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. arXiv preprint arXiv:2212.08073, 2022.

Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. Understanding and overcoming the challenges of ef cient transformer quantization. arXiv preprint arXiv:2109.12948, 2021.

Andy Brock, Soham De, Samuel L. Smith, and Karen Simonyan. High-performance large-scale image recognition without normalization. 38th International Conference on Machine Learning, ICML 2021, 2021.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting training data from large language models. 30th USENIX Security Symposium (USENIX Security 21), pages 2633–2650, 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, and Sebastian et al. Gehrmann. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311, 2022.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a xed-length context. Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, 2019.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. arXiv preprint arXiv:2208.07339, 2022.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. 2019 Conference Of The North American Chapter Of The Association

For Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), 2019.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. Proceedings of the 38th annual international symposium on Computer architecture, pages 365–376, 2011.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 13th International Conference on Artificial Intelligence and Statistics, AISTATS 2010, 2010.

Graphcore. Graphcore launches C600 PCIe card for AI compute. https://www.graphcore.ai/posts/graphcore-launches-c600-pcie-card-for-ai-compute, 2022. (Online: accessed 25 January 2023).

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. IEEE International Conference on Computer Vision, ICCV 2015, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2016, 2016.

Sara Hooker. The hardware lottery. Communications of the Association for Computing Machinery, 2021.

Xiao Shi Huang, Felipe Perez, Jimmy Ba, and Maksims Volkovs. Improving transformer optimization through better initialization. Proceedings of the 37th International Conference on Machine Learning, 2020.

Computer Society IEEE. IEEE standard for floating-point arithmetic. IEEE Std 754-2019, pages 1–84, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 32nd International Conference on Machine Learning, ICML 2015, 2015.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2018, 2018.

Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. Advances in Neural Information Processing Systems 31, NeurIPS 2018, 2018.

Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the graphcore ipu architecture via microbenchmarking. arXiv preprint arXiv:1912.03413, 2019.

Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFLOAT16 for deep learning training. arXiv preprint arXiv:1905.12322, 2019.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 3rd International Conference on Learning Representations, ICLR 2015, 2015.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. Advances in Neural Information Processing Systems 30, NeurIPS 2017, 2017.

Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Jason Li, Huyen Nguyen, Carl Case, and Paulius Micikevicius. Mixed-precision training for nlp and speech recognition with openseq2seq. arXiv preprint arXiv:1805.10387, 2018.

Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. Fp8 quantization: The power of the exponent. arXiv preprint arXiv:2208.09225, 2022.

Antoine Labatie, Dominic Masters, Zach Eaton-Rosen, and Carlo Luschi. Proxy-normalizing activations to match batch normalization while removing batch dependence. Advances in Neural Information Processing Systems 34, NeurIPS 2021, 2021.

Jiahuang Lin, Xin Li, and Gennady Pekhimenko. Multi-node BERT-pretraining: Cost-efficient approach. arXiv preprint arXiv:2008.00177, 2020.

Stephanie Lin, Jacob Hilton, and Owain Evans. TruthfulQA: Measuring how models mimic human falsehoods. arXiv preprint arXiv:2109.07958, 2021.

Alisa Liu, Maarten Sap, Ximing Lu, Swabha Swayamdipta, Chandra Bhagavatula, Noah A Smith, and Yejin Choi. Dexperts: Decoding-time controlled text generation with experts and anti-experts. arXiv preprint arXiv:2105.03023, 2021.

Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. Estimating the carbon footprint of bloom, a 176b parameter language model. arXiv preprint arXiv:2211.02001, 2022.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. 5th International Conference on Learning Representations, ICLR 2017, 2017.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. 6th International Conference on Learning Representations, ICLR 2018, 2018.

Paulius Micikevicius, Dusan Stosic, Patrick Judd, John Kamalu, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning. arXiv preprint arXiv:2209.05433, 2022.

Moin Nadeem, Anna Bethke, and Siva Reddy. StereoSet: Measuring stereotypical bias in pretrained language models. arXiv preprint arXiv:2004.09456, 2020.

Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. arXiv preprint arXiv:2106.08295, 2021.

Badreddine Noune, Philip Jones, Daniel Justus, Dominic Masters, and Carlo Luschi. 8-bit numerical formats for deep neural networks. arXiv preprint arXiv:2206.02915, 2022.

Nvidia. Nvidia H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core, 2022. (Online: accessed 25 January 2023).

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155, 2022.

Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuQmm: Quantized matmul for efficient inference of large-scale generative language models. arXiv preprint arXiv:2206.09557, 2022.

Yuan Peiwen and Zhu Changsheng. Normalized activation function: Toward better convergence. arXiv preprint arXiv:2208.13315, 2022.

PyTorch. Automatic mixed precision package - torch.amp. https://pytorch.org/docs/stable/amp.html, 2023. (Online: accessed 25 January 2023).

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training Gopher. arXiv preprint arXiv:2112.11446, 2021.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. arXiv preprint arXiv:1606.05250, 2016.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. arXiv preprint arXiv:1806.03822, 2018.

Charbel Sakr, Steve Dai, Rangha Venkatesan, Brian Zimmer, William Dally, and Brucek Khailany. Optimal clipping and magnitude-aware differentiation for improved quantization-aware training. 39th International Conference on Machine Learning, ICML 2022, 2022.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. Advances in Neural Information Processing Systems 29, NeurIPS 2016, 2016.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. arXiv preprint arXiv:1906.02243, 2019.

Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. Advances in Neural Information Processing Systems 32, NeurIPS 2019, 2019.

Richard S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. (Online: accessed 25 January 2023).

Tesla. A guide to tesla's configurable floating point formats & arithmetic. https://tesla-cdn.thron.com/static/MXMU3S_tesla-dojo-technology_1WDVZN.pdf, 2021. (Online: accessed 25 January 2023).

Thomas N. Theis and H.-S. Philip Wong. The end of Moore's law: A new beginning for information technology. Computing in Science & Engineering, 19(2):41–50, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in Neural Information Processing Systems 30, NeurIPS 2017, 2017.

Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit oating point numbers. arXiv preprint arXiv:1812.08011, 2018.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. arXiv preprint arXiv:2206.07682, 2022.

Laura Weidinger, John Mellor, Maribeth Rauh, Conor Grifn, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. Ethical and social risks of harm from language models. arXiv preprint arXiv:2112.04359, 2021.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144, 2016.

Guangxuan Xiao, Ji Lin, Mickaël Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and ef cient post-training quantization for large language models. arXiv preprint arXiv:2211.10438, 2022.

XLA and TensorFlow teams. XLA – TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, 2017. (Online: accessed 26 January 2023).

Jing Xu, Da Ju, Margaret Li, Y-Lan Boureau, Jason Weston, and Emily Dinan. Recipes for safety in open-domain chatbots. arXiv preprint arXiv:2010.07079, 2020.

Greg Yang and Edward J. Hu. Feature learning in in nite-width neural networks. arXiv preprint arXiv:2011.14522, 2020.

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor programs V: Tuning large neural networks via zero-shot hyperparameter transfer. arXiv preprint arXiv:2203.03466, 2022.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Ef cient and affordable post-training quantization for large-scale transformers. arXiv preprint arXiv:2206.01861, 2022.

Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. arXiv preprint arXiv:1904.00962, 2019.

Hongyi Zhang, Yann N. Dauphin, and Tengyu Ma. Residual learning without normalization via better initialization. 7th International Conference on Learning Representations, ICLR 2019, 2019.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutn k, and Jürgen Schmidhuber. Recurrent highway networks. 34th International Conference on Machine Learning, ICML 2017, 2017.

A. Floating point format specification

Table A.1. Common floating point formats for deep learning. E refers to the number of exponent bits, and M the number of mantissa bits of a given format. Max exp. and Min exp. refer to the maximum and minimum values that can be represented by the exponent, excluding special values. E5 (a) and E4 (a) refer to the FP8 formats introduced by Noune et al. (2022), whereas E5 (b) and E4 (b) refer to those introduced by Micikevicius et al. (2022).

Format	E	M	Max exp.	Min exp.
FP32	8	23	127	-126
TF32	8	10	127	-126
BFLOAT16	8	7	127	-126
FP16	5	10	15	-14
FP8 E5 (a)	5	2	15	-15
FP8 E5 (b)	5	2	15	-14
FP8 E4 (a)	4	3	7	-7
FP8 E4 (b)	4	3	8	-6

B. Proposed FP8 formats

Here we analyse the recently-proposed FP8 formats. We cover two proposals for 8-bit floating point formats (Noune et al., 2022; Micikevicius et al., 2022) (other proposals include Tesla (2021); Kuzmin et al. (2022)), each of which introduce one format with 4 exponent bits and a second format with 5. We refer to these here as E4 and E5 respectively (with the implication that the remaining bits represent the sign and mantissa).

To compensate for the low number of representable values, all of the proposed formats except the Micikevicius et al. (2022) E5 format deviate from the IEEE 754 standard by reducing the number of special values available. Both Noune et al. (2022) formats also increment the IEEE 754 bias by one. This slightly alters the maximum and minimum (absolute normal) values that each can represent.

FP8 formats in the literature are sometimes presented as having an explicit bias value, to be defined by the user (Noune et al., 2022; Kuzmin et al., 2022). The bias is subtracted from the exponent, just as in the IEEE 754 standard. This approach is equivalent to multiplying by 2^{bias}, and hence is no different from using a scaling factor to control the range of values represented. Micikevicius et al. (2022) explore both interpretations, with a preference for the scaling-factor viewpoint which aligns better with software implementations, whereas the exponent-bias viewpoint is more hardware aligned and in practice is likely to restrict bias values to integers.

These caveats aside, the proposed FP8 formats do not differ significantly from a standard-compliant 8-bit format.

C. Is unit standard deviation the correct criterion?

Here we justify the position that aiming for unit standard deviation of normally distributed tensors at initialisation is a sensible strategy.

When considering the scale of a floating-point tensor, we aim to keep absolute values within the upper and lower absolute normal bounds defined by a given format.

To analyse the absolute values generated by a normal distribution, we instead consider a folded normal distribution with zero mean and unit variance. Here, the central 90% of all probability mass falls within $[2^{-4}; 2^{1}]$.

As a point of comparison, for an IEEE 754 float the absolute range of normal values we can represent is approximately $[2^{2^{E-1}}; 2^{2^{E}-1}]$, giving a centre-point (in log-space) of 2^{1}.

From the perspective of clipping error, one might suggest scaling values to be as close as possible to this point, as we are equidistant from the upper and lower boundaries.

Hence, we can conclude that unit standard deviation will concentrate most values very near to, but slightly below the centre of the numerical range. Whether centrality within the normal floating-point range is the correct criterion for normally-distributed tensors during the training of deep learning models is a much harder question to answer.

In favour of sub-central scaling, is the argument that the subnormal values provides us with extra range at the lower end of the spectrum, albeit with reduced precision. Additionally, under flow in deep learning models tends to be less detrimental to training than over flow.

In favour of super-central scaling, is the argument that we might expect values such as gradients to decrease in magnitude during the course of training (our results in Section K suggest that this is true for BERT's grad_W values, though not for grad_X), and so we ought to up-scale values to compensate.

In light of these arguments, we argue that in situations where we can control scale, aiming for unit scaling is a sensible compromise. If we wished to precisely align the 90%-probability-mass range with the centre point calculated above, we might aim for a slightly larger scale. But given the confounding factors outlined, the difference is small enough that $= 2^{0}$ is still a strong choice, and keeps us aligned with other techniques in the literature with the same aim (e.g. Glorot and Bengio (2010)).

D. Unit scaling and emergent outliers

Recent work on inference quantisation for large language models (> 1B parameters) has highlighted the importance of

special techniques for accommodating outliers. These are large-magnitude values concentrated in particular sequence-elements (Bondarenko et al., 2021) or feature-dimensions (Dettmers et al., 2022), emerging as model size increases.

The main difficulty with accommodating tensors with outliers is that "a single outlier can reduce the quantisation precision of all other values" (Dettmers et al., 2022). These outliers have been shown to degrade INT8 quantisation accuracy at the 6.7B parameter model size and above, which leads to a key question: what impact do we expect outliers to have for unit scaling when applied to models of this size?

Firstly, we do not expect unit scaling to have a significant effect on the *magnitude* of outliers. This is because outliers occur in activation tensors, and these typically have a similar scale in unit and non-unit-scaled models (primarily due to the frequent presence of layernorms, which control scale).

However, we still expect unit scaling to be less impaired by outliers than the examples seen in recent literature. The key consideration here is that unit scaling is a *training* method and uses floating-point formats. In contrast, the literature on emergent outliers has all been in the integer quantisation setting.

Integer formats lack the dynamic range for training (Noune et al., 2022), and the same problem arises in the presence of outliers. We anticipate that using FP8 over INT8 will mitigate the difficulties presented to unit scaling by outliers. An analysis of the relative SNRs of the formats is insightful:

We first make some assumptions about the problem setting. We take the work of Dettmers et al. (2022) as our starting point, who show that the median outlier magnitude is 60 as accuracy begins to degrade. The distribution of non-outlier values is not clear, though the authors define non-outliers to have a magnitude of < 6. Hence, we assume that these have remained approximately unit-scaled.

To represent values in INT8 we will assume that they are scaled throughout such that outliers are (just) within the range of the format. This involves dividing by the expected maximum outlier value, and multiplying by the maximum INT8 value (127). We will assume a maximum outlier value of 3 the median, giving a scaling of $127 / (3 \cdot 60)$. To represent values in FP8 (E4) we do not need to re-scale values to accommodate outliers as the maximum FP8 E4 value is already larger than the maximum outlier, at 240.

Having scaled INT8 to accommodate outliers, the key question is what effect this has on the representation of non-outlier values. As observed in the literature, the "range of the quantisation distribution is too large so that most quantisation bins are empty and small quantisation values are quantised to zero, essentially extinguishing information" (Dettmers et al., 2022).

Figure A.1. The signal to noise ratio (SNR) of a quantised normal distribution, as a function of the distribution's scale. This plot is the same as Figure 2, but with the addition of scaled INT8 quantisation and vertical lines for outliers and non-outliers.

We model this scenario, calculating an SNR for the non-outlier values of only 2.03 (this raises to 14.8 if we scale for the median outlier rather than the max). In contrast, the SNR calculated in FP8 E4 is 635x higher, at $1.29 \cdot 10^3$. This is due to the exponential distribution of values in floating-point formats, which gives it a small number of large values (suitable for outliers) and a large number of small values (suitable for non-outliers).

This can be observed in Figure A.1, where we plot the SNR for this INT8 quantisation applied to a normally distributed tensor across different scales. Although INT8 gives a good representation of the outlier values (as does FP8 E4), the non-outlier values have low signal. One challenge for FP8 is the scenario in which outlier magnitude increases; in this case we would have to either re-scale or switch to the less precise E5 format.

Another way of viewing this is to look at the number of quantisation bins each format makes use of in this setting. For INT8 the lower 95% of non-outlier values are assigned to just 3 out of 256 quantisation bins. In contrast, for FP8 90 bins are utilised.

This modelling gives us cause for optimism when applying unit scaling in the presence of outliers, though we acknowledge there may still be challenges.

E. Theoretical results

E.1. Example – scaling analysis

We reproduce a simple version of the scaling analysis of Glorot and Bengio (2010), for a multilayer perceptron (MLP).

Consider an MLP which transforms inputs X_0 to outputs X_L using $X_{l+1} = f(X_l W_l)$ for $l \in [0, \ldots, L-1]$, where $f(\cdot)$ is an elementwise activation function. We separate the analysis of a single layer into $Z = XW$ and $Y = f(Z)$.

Projection First, $Z = XW$, where $Z \in \mathbb{R}^{b\ n}$, $X \in \mathbb{R}^{b\ m}$, $W \in \mathbb{R}^{m\ n}$, and X, W each have independently distributed elements with zero mean and variance σ_X^2 and σ_W^2 respectively. The values in Z follow $Z_{ik} = \sum_j X_{ij} W_{jk}$, which is a sum over m uncorrelated products, each with variance $\sigma_X^2 \sigma_W^2$. Then, by the variance of an independent sum, the output variance $\sigma_Z^2 = m \sigma_X^2 \sigma_W^2$.

When computing the partial derivative of a scalar loss with respect to X, $\partial_X L = (\partial_Z L) W^\top$, assuming $\partial_Z L$ is zero mean with variance $\sigma_{\partial_Z L}^2$ and is not correlated with W,[3] then by same reasoning as above $\sigma_{\partial_X L}^2 = n \sigma_{\partial_Z L}^2 \sigma_W^2$. And again $\sigma_{\partial_W L}^2 = b \sigma_{\partial_Z L}^2 \sigma_X^2$.

Activation Consider $f(Z) = \text{relu}(Z) = \max(Z, 0)$, with $Z \sim \mathcal{N}(0, 1)$. Then, in the forward pass $\mathbb{P}(f(Z) = y) = \frac{1}{2}\delta(y) + H(y)\, P_N(y)$, where $P_N(\cdot)$ is the pdf of a standard normal distribution, and $H(\cdot)$ is the Heaviside step function. This gives variance $\sigma_Y^2 = \frac{1}{2}(1 - 1/\pi)$. In the backward pass, $\mathbb{P}(\partial_Z L = z') = \frac{1}{2}\delta(z') + \frac{1}{2} P_N(z')$, with variance $\sigma_{\partial_Z L}^2 = \frac{1}{2}$.

He et al. (2015) note that the activation function can break the local distributional assumption for the first step: for example, the ReLU function $h(Z) = \max(Z, 0)$ does not produce zero mean output, invalidating our previous assumption on X_l. However, the corrections for such invalid assumptions are often small, and can be ignored for sake of expedience, permitting local scaling analysis.

For an example of extending scale analysis to training, Huang et al. (2020) consider the training dynamics of a Transformer under Adam, using this to derive an initialisation scheme that avoids vanishing updates.

E.2. Proofs in support of Proposition 5.1

For two common choices of optimiser, SGD and Adam, we show that there is an unscaled model with identical training dynamics as any unit-scaled model.

E.2.1. SGD

We define a model as an op with scalar output and a subset of inputs denoted as trainable parameters, written $f(\ldots; x_{j\,2\,1:::k})$.

A training trajectory is defined as a sequence $\theta_i^{(t)}$ for all parameters in a model, given initial settings $\theta_i^{(0)}$ and optimiser.

[3] This is likely to be a very bad assumption, since W was used to generate Z and therefore $\partial_Z L$. But it is hard to avoid this assumption without doing a global analysis of the model.

For SGD,

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \eta \frac{\partial f(\ldots)}{\partial \theta_i}; \quad = \theta_i^{(t)} - \eta\, f_{\text{grad}}(\ldots; 1)_i;$$

where η is a constant learning rate hyperparameter. We define the trajectory under a scaled op similarly, using f_{grad}:

$$\theta_i^{\prime(t+1)} = \theta_i^{\prime(t)} - \eta\, f_{\text{grad}}(\ldots; 1)_i.$$

Proposition E.1. For any scaled op with training trajectory $\theta_i^{\prime(t)}$ under SGD, there exists an equivalent unscaled op with training trajectory $\theta_i^{(t)} = \sqrt{\alpha_i}\, \theta_i^{\prime(t)}$.

We consider the evolution of the following unscaled op under SGD on $\hat{f}$:

$$\hat{f}(\ldots; x_{i\,2\,1:::n}; x_{j\,2\,1:::k}), \qquad f(\sqrt{\alpha_i}\, x_{i\,2\,1:::n}; x_{j\,2\,1:::k}).$$

Applying the chain rule to obtain gradients,

$$\frac{\partial \hat{f}(\ldots; x_{i\,2\,1:::n}; \ldots)}{\partial x_i} = \sqrt{\alpha_i}\, \frac{\partial f(\sqrt{\alpha_{i'}}\, x_{i'\,2\,1:::n}; \ldots)}{\partial x_i}.$$

Substituting to get the evolution of $\hat{f}$ under SGD,

$$\hat{\theta}_i^{(t+1)} = \hat{\theta}_i^{(t)} - \eta \sqrt{\alpha_i}\, \sqrt{\alpha_i}\, \frac{\partial f(\sqrt{\alpha_{i'}}\, \theta_{i'\,2\,1:::n}^{(t)}; \ldots)}{\partial \theta_i}.$$

We can now use the definition of α as follows and obtain

$$\sqrt{\alpha_i}\, \sqrt{\alpha_i} = \alpha_i;$$

$$\theta_i^{\prime(t+1)} = \theta_i^{\prime(t)} - \eta\, f_{\text{grad}}(\theta_{i\,2\,1:::n}^{\prime(t)}; \ldots; 1)_i.$$

Therefore if the initial condition $\theta_i^{(0)} = \sqrt{\alpha_i}\, \theta_i^{\prime(0)}$ is satisfied, then $\theta_i^{(t)} = \sqrt{\alpha_i}\, \theta_i^{\prime(t)}$ thereafter.

E.2.2. Adam

As noted by Kingma and Ba (2015), Adam is invariant to diagonal rescaling of the gradients. Defining the function adam that computes a single update thus:

$$\theta^{(t+1)} = \text{adam}(\theta^{(t)}, \frac{\partial f}{\partial \theta});$$

invariance to diagonal rescaling gives

$$\text{adam}(\theta^{(t)}, \frac{\partial f}{\partial \theta}) = \text{adam}(\theta^{(t)}, s \frac{\partial f}{\partial \theta});$$

for any positive-valued scaling vector $s \in (\mathbb{R}^+)^{j\ j}$ that is constant over all timesteps t.

Proposition E.2. For any scaled op with training trajectory $\theta_i^{(t)}$ under Adam with $\epsilon = 0$, there exists an equivalent unscaled op with the same training trajectory.

Consider the unscaled op $\hat{f}(\ldots) = f(\ldots)$. This follows the trajectory

$$\theta_i^{(t+1)} = \text{adam}(\theta_i^{(t)}, \frac{\partial f}{\partial \theta_i}).$$

Now consider the scaled op f with the same η, β_1, β_2. This follows:

$$\begin{aligned}\alpha_i \theta_i^{\prime(t+1)} &= \text{adam}(\alpha_i \theta_i^{\prime(t)}, \alpha_i \frac{\partial f}{\partial \theta_i}); \\ &= \text{adam}(\alpha_i \theta_i^{\prime(t)}, \frac{\alpha_i}{\beta_i} \frac{\partial f}{\partial \theta_i}).\end{aligned}$$

Therefore if $\theta_i^{\prime(0)} = \theta_i^{(0)}$, we conclude $\theta_i^{\prime(t)} = \theta_i^{(t)}$.

E.3. Example – a scaled computational graph does not necessarily represent a scaled op

Let $f(x_1, \ldots, x_n)$ be an unscaled operation with values in $\mathbb{R}^n$ and consider the scaled computational graph defined by $x + f(x, \alpha_1, \ldots, \alpha_n)$. If this scaled computational graph represented a scaled op $h(x_1, \ldots, x_n)$ for some function $h(x_1, \ldots, x_n)$, there would exist scalars $\beta_0, \beta_1, \ldots, \beta_n$ such that:

$$\begin{aligned}\beta_0 h(x) &= x + f(x, \alpha_1, \ldots, \alpha_n); \\ \beta_i g_i &= \frac{\partial h(\ldots)}{\partial x_i} = g_i + f_{\text{grad}}(x, \alpha_1, \ldots, g)_i \quad \forall i \in \{1, \ldots, n\}.\end{aligned}$$

Consider $f(x) = x^2$, so that

$$\begin{aligned}f(x, \alpha_1) &= x^2; \\ f_{\text{grad}}(x, \alpha_1, g)_i &= 2\alpha_1 x_i g_i \quad \forall i \in \{1, \ldots, n\}.\end{aligned}$$

This implies

$$\frac{\beta_0}{\beta_i} g_i (1 + 2\alpha_1 x_i) = g_i + 2\alpha_1 g_i x_i \quad \forall i \in \{1, \ldots, n\}.$$

Assuming $g_i \neq 0$, in the case $x_i \neq 0$ these two expressions cannot be made to match by any choice of (β_0, β_i). Therefore the scaled graph does not implement a scaled op.

E.4. Proof of Theorem 5.2

We first define how a computational graph represents an op. Then we show that an unscaled graph correctly represents an unscaled op. Finally, we proceed to show that a constraint-scaled graph with a single output correctly represents a scaled op.

Graph – op We adopt a generalisation of the earlier definition of an op, to permit multiple outputs. An op defines mappings from k vector-valued inputs to m vector-valued outputs via $f(x_{i\,2\,1:::k})_{j\,2\,1:::m}$, and corresponding gradient mappings,

$$f_{\text{grad}}(x_{i\,\circ 2\,1:::k}; g_{j\,\circ 2\,1:::m})_i, \quad \sum_j g_j \frac{\partial f(x_{i\,\circ 2\,1:::k})_j}{\partial x_i}.$$

We use f_G to denote the op represented by the computational graph G. To evaluate the function and the vector Jacobian product $f_{\text{grad};G}$, we assign inputs and outputs to edges in the graph[4]. Define a list of input edges, $\text{in}_{i\,2\,1:::k} \in E$, and output edges, $\text{out}_{j\,2\,1:::m} \in E$.

Define the forward value of an edge using $z : E \to \mathbb{R}^{(\ldots)}$, via the recursive relations:

$$\begin{aligned}z(\text{in}_i) &, \quad x_i; \\ z((u, v)) &, \quad f_u(f\, z((w, u)) \mid (w, u) \in E)_v; \\ f_G(x_{i\,2\,1:::k})_j &, \quad z(\text{out}_j);\end{aligned}$$

where $f_u(\ldots)_v$ evaluates node u's output corresponding to the edge (u, v).

Similarly, define the backward value of an edge using $h : E \to \mathbb{R}^{(\ldots)}$ via:

$$\begin{aligned}h(\text{out}_j) &, \quad g_j; \\ h((u, v)) &, \quad f_{\text{grad};v}(f\, z((u', v)) g; f\, h((v, r)) g)_u; \\ f_{\text{grad};G}(\ldots; g_{j\,2\,1:::m})_i &, \quad h(\text{in}_i);\end{aligned}$$

where $f_{\text{grad};v}(\ldots)_u$ evaluates the grad op for node v for the input $x_{v;u}$ corresponding to the edge (u, v). Note that we use the shorthand $f\, z((u', v)) g$ to denote $f\, z((u', v)) \mid (u', v) \in E g$.

Unscaled graph – op To show that $(f_G; f_{\text{grad};G})$ represent an op, we must show they are consistent with the definition of f_{grad}. We expand the backward value using the definition of $f_{\text{grad};v}$,

$$h((u, v)) = \sum_w h((v, w)) \frac{\partial f_v(f\, z((u', v)) g)_w}{\partial x_{v;u}}.$$

Using the base case for $h(\text{out}_j)$ and the chain rule,

$$h((u, v)) = \sum_w \sum_q h((w, q)) \frac{\partial f_w(\ldots)_q}{\partial x_{w;v}} \left(\frac{\partial f_v(\ldots)_w}{\partial x_{v;u}} \right)^{\top},$$

$$h((u, v)) = \sum_j g_j \frac{\partial f_{G;v}(\ldots)_j}{\partial x_{v;u}}.$$

Therefore $h(\text{in}_i)$ gives the correct gradient, so G correctly represents an op.

[4] It is often natural to assign inputs and outputs to nodes, but we use edges in our analysis for notational convenience. Such edges imply the existence of `dummy' nodes.

Constraint-scaled graph – scaled op Again, generalising the earlier definition to multiple outputs,

$$f \;(x_{i\,2\,1:::k})_j \;,\quad f\,(x_{i\,2\,1:::k})_j \;;$$
$$f_{\text{grad}}\,(x_{i\,0 2\,1:::k}\,;g_{j\,0 2\,1:::m})_i \;,\quad \sum_j g_j^{>}\,\frac{@f(x_{i\,0 2\,1:::k})_j}{@x_i}\,:$$

Note that all outputs are scaled using a single value $_v$. Using the same definitions for z and h,

$$h((u;v)) = \sum_w h((v;w))^{>}\,\frac{@f_v\,(f\,z((u^0;v))\,g)_w}{@x_{;u}}\,;$$
$$= \frac{_{v;u}}{_v}\sum_w h((v;w))^{>}\,\frac{@f_v\,(f\,z((u^0;v))\,g)_w}{@x_{;u}}\,:$$

In order to apply the chain rule here, we must first deal with the scale ratio $\frac{_{v;u}}{_v}$. To do this, we define the unscaled backward value $\hat{h}$, in terms of a single reachable output out and a rescaling function $s : E \;E \to \;R$, thus:

$$\hat{h}((u;v)) \;,\quad \frac{h((u;v))}{s((u;v);out)}\,;$$
$$s(a;b) \;,\quad \prod_{(u;v)\,2E^{\,cut(\,a;b\,)}} \frac{_{v;u}}{_v}\,;$$

where $E^{cut(\,a;b)}$ is the set of edges where, after the removal of any one, there is no path connecting the head of a and the head of b in G. We observe this property for adjacent edges:

$$\frac{s((v;w);out)}{s((u;v);out)} = \begin{cases} \frac{_v}{_{v;u}} & \text{if } (u;v) \text{ is a cut-edge} \\ 1 & \text{otherwise} \end{cases}\,;$$

which follows directly from the definition of s. Now we substitute into our grad,

$$\hat{h}((u;v)) = \sum_w (u;v;w)\;\;\hat{h}((v;w))^{>}\,\frac{@f_v\,(:::)_w}{@x_{;u}}\,;$$
$$(u;v;w) \;,\quad \frac{_{v;u}}{_v}\;\;\frac{s((v;w);out)}{s((u;v);out)}\,:$$

Consider two cases:

Case 1: $(u;v)$ is not a cut-edge. The rules of constraint-scaled computation graphs ensure $_{v;u} = \;_v$. From the aforementioned property, $s((u;v);out) = s((v;w);out)$. So we conclude $(u;v;w) = 1$.

Case 2: $(u;v)$ is a cut-edge. From the same property, we conclude $(u;v;w) = 1$.

Since in either case, $(u;v;w) = 1$, we can simplify:

$$\hat{h}((u;v)) = \sum_w \hat{h}((v;w))^{>}\,\frac{@f_v\,(:::)_w}{@x_{;u}}\,;$$

which is the correct form for the chain rule and induction from the base case as previously, noting that $s(out;out) = 1$ so $\hat{h}(out) = g$. We can therefore conclude that $\hat{h}$ gives true gradients and:

$$\hat{h}((u;v)) = g^{>}\,\frac{@f_{G;v}\,(:::)}{@x_{;u}}\,;$$
$$h((u;v)) = s((u;v);out)\;\;\hat{h}((u;v))\,:$$

So G represents a scaled op with $h = s(in_i\,;out)$.

F. Constraint-scaled computational graphs for other schemes

For sake of comparison, it can be instructive to consider other scaling schemes within the constraint-scaled computational graph framework.

Glorot initialisation (Glorot and Bengio, 2010) For a layer $Y = f\,(XW)$, consider the scales $_Y$ and $_{r\,X\,L}$, ignoring $_{r\,W\,L}$. Apply full constraints, and typically use arithmetic mean rather than geometric mean to combine scales. Finally, push the combined scale into the initialisation of W, so that no multiplication is required at execution time.

Loss scaling (Micikevicius et al., 2018) Introduce a single scaled identity op before the loss. $f\,(x) = \;x$, $f_{\text{grad}}\,(x;g) = \;g$. Since this edge is always a cut-edge, set $= 1$, and use $$ to generate gradients that all share a single scale. Unlike unit scaling, there are no local distributional assumptions that can inform the choice of loss scale—it must be chosen empirically or heuristically.

Scaled dot product self attention (Vaswani et al., 2017) When computing the similarity matrix $A = QK^{>}$; $Q;K \;2\;R^{s\;\;d}$, consider the scale $_A$, ignoring $_{r\,Q\,L}$, $_{r\,K\,L}$. Apply fully constrained scaling, yielding $= \;_1 = \;_2 = \frac{1}{\sqrt{d}}$. This is perhaps the best pre-existing example of a commonly employed scheme similar to unit scaling.

G. Unit scaled ops compendium

Unit scaling relies on the correct selection of the scaling factors $\;;\;_i\,;:::;\;_k$ for a given op. These scaling factors are derived from an analysis of the scaling of a given operation and its corresponding grad op, as outlined in Section 5.2, with an example of analysing the scaling of a multilayer perceptron given in Appendix E.

To avoid practitioners having to analyse the scaling characteristics of each op in their model by hand, we provide a reference for common ops in Table A.2, giving scaled versions of each op alongside necessary scaling factors.

We provide further details on the derivation of certain non-trivial scaled operations below.

Activations We calculate the scaling of ReLU analytically, based on the analysis in Appendix E.1. The other activation functions given are not amenable to the same procedure, so we calculate their scaling empirically (this is done through the use of short programs, which only need consider functions in isolation rather than within a larger model).

Softmax (followed by matmul) We make the simplifying assumption in our analysis that the output of a softmax over s normally-distributed elements is uniformly $1/s$. In practice, there is some variance across output elements but this is small enough to ignore for our purposes.

This deviates from our standard unit scaling assumption of zero mean and unit variance, with $1/s$ mean and zero variance instead. Hence we require a different strategy for scaling softmax if we wish to still propagate unit scale.

We assume in this scenario that the softmax is followed by a matmul (as in multi-head self-attention). Based on this assumption, we scale by a factor of s, meaning the output is approximately a vector of ones.

From the perspective of the subsequent matmul, its ideal choice of scaling factor is then identical to the scaling factor it would have required if its input were sampled from a unit normal distribution: $m^{-\frac{1}{2}}$, where m is the size of the dimension reduced over. The subsequent matmul op can then be implemented using our standard scaling without any special-case behaviour.

We also find through empirical analysis that the backward pass of softmax requires s scaling, though in this direction it generates normally distributed values, conforming to our standard assumption.

Softmax cross-entropy We now consider a softmax going into a cross-entropy function, treating this composition as a single operation: $\text{softmax\_xent}(x, t) = -\log \text{softmax}(x)_t$ (where t is the index of the target label), and assume that this is the final layer in a model used to generate a loss.

On this basis, we need not consider forward scaling, and focus on the backward operation $x' = \text{softmax\_xent}_{\text{grad}}(x, t)$ and the calculation of its scaling factor $= 1/\sigma(x')$.

Assuming again that at the beginning of training the output of the softmax over s inputs is uniformly $1/s$, the gradient of softmax cross-entropy is given by,

$$x'_i = \text{softmax\_xent}_{\text{grad}}(x, t)_i = \begin{cases} \frac{1}{s} - 1; & \text{if } i = t \\ \frac{1}{s}; & \text{otherwise} \end{cases}$$

where $x \in \mathbb{R}^s$.

To calculate $\sigma(x')$ we first observe that,

$$\begin{aligned} \mathbb{E}[x'] &= \frac{1}{s}\left(\frac{1}{s} - 1\right) + (s-1)\frac{1}{s} \\ &= 0 \end{aligned}$$

from which we derive,

$$\begin{aligned} \sigma(x')^2 &= \mathbb{E}\left[(x')^2\right] - \mathbb{E}[x']^2 \\ &= \frac{1}{s}\left(\frac{1}{s} - 1\right)^2 + (s-1)\left(\frac{1}{s}\right)^2 - 0 \\ &= \frac{1}{s}\cdot\frac{1 - 2s + s^2 + s - 1}{s^2} \\ &= \frac{s-1}{s^2} \end{aligned}$$

This gives us our scaling factor, $= s/\sqrt{s-1}$.

H. Aligning unit scaling with existing models

Our presentation of unit scaling in Section 5 assumes the design of a model from scratch. However, we anticipate there will be cases in which practitioners will wish to unit scale existing models, such that their unit scaled model and base model are either equivalent or similar enough to give matching performance.

Here we outline the additional considerations required to do so. We follow this approach for our BERT experiments in Section 6.2.

H.1. Activation functions

We take 'activation function' to mean any non-linear element-wise function in a model. Due to non-linearity, the behaviour of an activation function $f(x)$ depends on the scale of its input. Therefore a base model's activation functions may not have the same effect on their inputs as a unit scaled version, as the unit scaled model alters the scale of inputs.

To counter this, one can introduce a scaling factor immediately before an activation function (temporarily breaking unit scale), and a second un-scaling factor immediately afterwards (restoring unit scale):

$$\hat{f}(x) = f(s_1 x)\, s_2,$$

where $\hat{f}$ is our new 'aligned' activation function, x is assumed to be normally distributed with unit scale (not necessarily true for x in the base model), and $s_1, s_2 \in \mathbb{R}$ are our new scaling factors.

Table A.2.Table of unit scaling factors, based on simple distributional assumptions on inputs and gradients, most often that they are unit normal.

Op	Unit scaling factors
LINEAR	
$\text{matmul}(X^{b\ m}; W^{m\ n})^{b\ n} = XW$	$= m^{\frac{1}{2}},\ _X = n^{\frac{1}{2}},\ _W = b^{\frac{1}{2}}$
$\text{sum}(x) = \sum_{i=1}^{n} x_i$	$= n^{\frac{1}{2}},\ = 1$
$\text{weighted\_add}(x_{i\,2\,1:::n}; _{i\,2\,1:::n}) = \sum_{i=1}^{n} _i x_i$	$= \sum_i {}_i^2 {}_i^{\frac{1}{2}},\ _i = {}_i^1$
ACTIVATIONS	
$\text{relu}(x) = \max(x; 0)$	$= \sqrt{2=(1 - 1=)},\ = \sqrt{2}$
$\text{gelu}(x) = x\ (\ x)$	$= 1:701,\ = 1:481$
$\tanh(x) = (e^{2x} - 1)=(e^{2x} + 1)$	$= 1:593,\ = 1:467$
$\text{sigmoid}(x) = (1 + e^{\ x})^{\ 1}$	$= 4:802,\ = 4:722$
OTHER	
$\text{softmax}(x)_i = e^{x_i} = \sum_{j=1}^{s} e^{x}_{j}$	$= s,\ = s$
$\text{softmax\_xent}(x; t) = \log \text{softmax}(x)_t$	$= 1,\ = s = \sqrt{s - 1}$
$\text{layer\_norm}(X^{b\ n}; w; c)_{ij} = c_j + w_j\ (X_{ij} \quad _i) = _i,$ $\ldots\ _i = \frac{1}{n}\sum_{j=1}^{n} X_{ij},\ _i = \sqrt{\frac{1}{n}\sum_{j=1}^{n} X_{ij}^2 \quad {}_i^2}$	$= 1,\ _x = 1,\ _w = \ _c = b^{\frac{1}{2}}$

We select the first scaling factor such that $s_1 = \ (x)$, giving identical-scale inputs to both activation functions: $(s_1 \hat{x}) = \ (x)$.

The second scaling factor is selected to restore unit scale: $s_2 = \frac{1}{(f(x))}$, giving,

$$\hat{(f}(\hat{x})) = \frac{f(\ (x)\ \hat{x})}{(f(x))};$$
$$= 1:$$

All that remains is the estimation of (x) and $(f(x))$ in the base model. This can be done either analytically (by stepping through operations in the base model and calculating the expected scale at each) or empirically (via instrumentation of the base model). The latter method tends to be simpler and less error-prone, but the former is more mathematically rigorous and has the advantage of generating scaling factors that are a function of the model's hyperparameters.

Note that although we temporarily break the assumption of unit scale in the above analysis, in practice scaling factors here are close enough to 1 that this momentary mis-scaling is negligible from a numerics perspective.

H.2. Softmax functions

The above analysis also applies to softmax functions. Although softmax is not an element-wise function, the same approach is still valid and $s_1; s_2$ should be chosen in the same way.

Note that the standard softmax function is sometimes introduced with a `temperature' scalar, T, by which all inputs are divided. Hence our method can be seen as tuning the effective temperature of the softmax to align the unit scaled model with the base model.

H.3. Residual weighted add

In Section 5.3 we recommended that practitioners introduce a weighted addition into their models between residual and skip branches, in order to actively select how much each contributes to the output.

A typical unscaled base model implicitly makes this choice via the scaling effect of the residual branch (i.e. the ratio of $(f(x)) = \ (x)$, which typically $\ 1$).

For our unit-scaled model to be equivalent to the base model, we need the output of our addition to be equal up to a constant (unit) scaling factor.

Taking a $\text{xed}(\ )$ residual layer, this means we must maintain: $\sqrt{1 \quad}\hat{x} + \sqrt{\ }\hat{f}(\hat{x}) = \ (x + f(x))$, where $\hat{f}(\ )$ is the residual branch and $\hat{x}$ the input in our unit-scaled model.

Thanks to unit scaling, we have $\hat{x} = x = \ (x)$ and $\hat{f}(\hat{x}) =$

$f(x) = \sqrt{\text{Var}(f(x))}$ giving,

$$\hat{f}(\hat{x}) = \sqrt{\frac{1}{\text{Var}(x) + \text{Var}(f(x))}} \frac{x}{\sqrt{\text{Var}(x)}} + \sqrt{\frac{\text{Var}(f(x))}{\text{Var}(f(x))}}$$

Our desired form requires the terms multiplying x and $f(x)$ to be equal, meaning:

$$\sqrt{\frac{1}{\text{Var}(x) + \text{Var}(f(x))}} = \sqrt{\frac{\text{Var}(f(x))}{\text{Var}(f(x))}}$$
$$= \frac{\text{Var}(f(x))}{\text{Var}(x) + \text{Var}(f(x))},$$

giving,

$$= \sqrt{\frac{1}{\text{Var}(x) + \text{Var}(f(x))}},$$

and recalling that our original de nition of $\hat{f}$ ixed( ) residual layer ensures that this still maintains a unit-scaled output.

Hence to align the residual add operation with a base model, we need rst need to use ixed( ) residual layer, and secondly calculate $\text{Var}(x)$ and $\text{Var}(f(x))$ for the base model, plugging them into the above equation for .

This calculation of in the base model can again be done analytically or empirically. For typical models, the correct value of is the same across layers.

H.4. Shared parameters

Weights used in multiple operations in the forward pass sum the weight gradients coming from those operations in the backward pass.

The same argument used for the residual add applies to the alignment of this summation too: for a unit-scaled model to be equivalent it must match the ratio of scales going into this sum as in the base model. Unit scaling will normalise these all to have $= 1$, but this is not guaranteed in the base model.

The same analysis as used for the residual add op can be applied here, with the same outcome. The calculation of the scale of residual branches in the base model should be substituted with the scale of each weight gradient. In the case that the weight gradient is used more than twice, the above argument will have to be generalised to multiple operands.

H.5. Example: aligning BERT

We follow the steps above in our experiments for Section 6.2, where we align unit-scaled BERT models against standard baseline models, to match performance.

Here we outline how we apply the above rules in practice, along with a few additional considerations required due to speci cs relating to the BERT architecture.

Where these rules require the calculation of standard deviation of tensors in the base model, we always calculate them analytically, rather than relying on empirical measurements (though we have then used empirical measurements to check the correctness of our calculations).

Embedding layer BERT contains three separate embeddings: a general word embedding, along with segment and positional embeddings. These are all combined using a summation at the beginning of the model. For unit scaling we must implement this using:

$$x_{\text{emb}} = \text{weighted\_add}\left(x_{\text{word}}, x_{\text{seg}}, x_{\text{pos}}; \sqrt{\tfrac{1}{3}}, \sqrt{\tfrac{1}{3}}, \sqrt{\tfrac{1}{3}}\right).$$

Weights are equal here as the initial scales of the embeddings in the base model are unchanged from their initialisation, and all are initialised with the same scale.

FFN For the FFN, alignment need not be considered for the matmul and layernorm ops, which we scale using the set of scaling factors for common ops given in Table A.2. For the gelu activation function, we must follow the alignment process outlined above, applying scaling factors immediately before and after.

Multi-head self-attention For multi-head self attention, we employ the rule for aligning softmax (followed by a matmul) given above. Again, matmuls do not require alignment with the base model. We note that in the particular case of the matmul with the V tensor, our standard distributional assumption of independent elements no longer strictly holds, due to correlation across the sequence dimension introduced by the segment embedding. This requires a slight correction to ensure unit scaling is maintained.

Residual connection Both the FFN and multi-head self-attention layers are residuals, and as such employ the rule above for aligning weighted addition with a base model.

Loss heads We train BERT according to the standard procedure of using two heads: one for the masked-language-modelling (MLM) task, and one for the next-sentence-prediction (NSP) task. The NSP head uses a tanh activation function which requires alignment, and the MLM head reuses the weights of the word embedding for a matmul, requiring the above rule for aligning shared parameters. Each head is terminated by a softmax cross-entropy, that we also tune to match the base model.

Sequence length considerations Care must be taken when unit-scaling sequence-based models to account for the role of the sequence dimension. For many ops this effectively becomes an extra batch dimension, and must be handled as such when applying unit scaling.

In our experiments we use padding to compensate for uneven-length input-sequences. In this case the value used for our sequence calculations is not the length of the sequence dimension, but the average number of non-padding tokens in a sequence (for our experiments, this was approximately 77% of the padded length).

One additional complication specific to BERT, is that the gradients flowing back into the final transformer layer are sparse, as they only come via the subset of tokens used in the two heads (specifically, the [CLASS] token, and those tokens masked for the MLM head). As a result, backwards pass sequence length calculations for this layer must be adapted to assume a smaller sequence length, according to the level of sparsity in the gradient.

I. Implementation

Unit scaling is straightforward to implement in deep learning frameworks such as PyTorch, JAX and TensorFlow, that support user-defined custom gradient autograd operations. A convenient way to do this is via a scaled identity op, $\mathrm{id}(x; \alpha, \beta)$, which can be used to implement scaled ops without defining custom gradients for each.

I.1. Code examples

We show an example implementations in Figure 3, with additional code listings in Figure A.2, demonstrating basic tools for constructing unit-scaled models in PyTorch. Note

`scaled` is the basic building block of unit-scaled models. It enables independent control of forward and backward pass scaling factors, and as such must be used with care—it could be used to define a scaled graph with incorrect constraints leading to gradients that are inconsistent with the forward pass of the model.

`scaled_matmul` demonstrates how to combine multiple constraints using geometric mean.

`scaled_gelu` implements only fully constrained scaling, for brevity. When scales are fully constrained, custom gradients via `scaled` are optional. Note that it may still be useful in certain situations for improving the scale of intermediate values.

`ScaledLayerNorm` uses the usual assumption for scaled layers: weights are cut-edges, activations are not. This permits independent scales for the weight and bias parameters.

```python
class ScaledGrad(autograd.Function):
    @staticmethod
    def forward(ctx, X, alpha, beta):
        ctx.save_for_backward(
            tensor(beta, dtype=X.dtype))
        return alpha * X

    @staticmethod
    def backward(ctx, grad_Y):
        beta, = ctx.saved_tensors
        return beta * grad_Y, None, None

def scaled(X, alpha=1, beta=1):
    # Forward: Y = X * alpha
    # Backward: grad_X = grad_Y * beta
    return ScaledGrad.apply(X, alpha, beta)

def scaled_matmul(
    A, B, constrain_A=True, constrain_B=True,
):
    (m, k), (_, n) = A.shape, B.shape
    alpha = k ** -(1/2)
    beta_A = n ** -(1/2)
    beta_B = m ** -(1/2)

    if constrain_A and constrain_B:
        alpha = beta_A = beta_B = \
            (alpha * beta_A * beta_B) ** (1/3)
    elif constrain_A:
        alpha = beta_A = (alpha * beta_A) ** (1/2)
    elif constrain_B:
        alpha = beta_B = (alpha * beta_B) ** (1/2)

    A = scaled(A, beta=beta_A)
    B = scaled(B, beta=beta_B)
    return scaled(matmul(A, B), alpha)

def scaled_gelu(X):
    return 1.5876 * gelu(X)

class ScaledLayerNorm(nn.LayerNorm):
    def forward(self, x):
        beta = (
            np.prod(self.normalized_shape)
            / x.nelement()
        ) ** 0.5
        return nn.functional.layer_norm(
                x,
                self.normalized_shape,
                scaled(self.weight, beta=beta),
                scaled(self.bias, beta=beta),
                self.eps,
        )
```

Figure A.2. Definition of `scaled` in PyTorch, as a custom autograd function. Additional scaled ops and layers required for a Transformer FFN. See Table A.2 for a reference of scaling factors.

I.2. Computational overhead

Unit scaling typically introduces one extra function invocation per invocation in the equivalent unscaled model. For example, matmul typically involves 3 function invocations during training, corresponding to 1 forward, 2 backward functions (one for each input). Using unit scaling, there are 3 additional rescaling function invocations of the form $f(x; \alpha) = \alpha x$, where $\alpha \in \mathbb{R}, x \in \mathbb{R}^n$.

FLOPs Considering the typical theoretical metric for computational effort, floating point operations (FLOPs), the overhead appears much smaller. For the matmul op with forward pass $\text{matmul} : \mathbb{R}^{b \times n} \times \mathbb{R}^{n \times m} \to \mathbb{R}^{b \times m}$, the amount of computational effort due to 3 matmul is $6bnm$ (note this is 2 because multiply and add are counted separately), while rescaling consumes $bn + nm + bm$. Therefore the ratio of rescaling to matmul ops follows:

$$\frac{\text{FLOP}_{\text{rescaling}}}{\text{FLOP}_{\text{matmul}}} = \frac{1}{6}(b^{-1} + m^{-1} + n^{-1}).$$

Note that this is bounded above by $\frac{1}{2}(\min(b, n, m))^{-1}$. For the matmuls that dominate compute in many models, this minimum dimension corresponds to the hidden size.

There are also operations other than matmuls that require scaling, but contribute negligible FLOPs. To simplify analysis, we'll assume that there are $(\text{ops\_per\_matmul} - 1)$ additional ops for every matmul in the model.

So we write $\text{FLOP}_{\text{matmul+}} = \text{FLOP}_{\text{matmul}}$ and $\text{FLOP}_{\text{rescaling+}} = \text{ops\_per\_matmul} \cdot \text{FLOP}_{\text{rescaling}}$. This gives the following adjusted estimate for the FLOP overhead of unit scaling a model:

$$\frac{\text{FLOP}_{\text{rescaling}}}{\text{FLOP}_{\text{unscaled}}} = \frac{\text{ops\_per\_matmul}}{2 \cdot \text{hidden\_size}}.$$

In the example of $\text{BERT}_{\text{LARGE}}$, we set $\text{hidden\_size} = 1024$, pessimistically estimate $\text{ops\_per\_matmul} = 4$, and obtain a FLOP overhead of 0.2%.

Other large models should behave in a similar manner, so we conclude that the theoretical FLOP overhead of unit scaling is small for large models. Actual performance will depend on many other factors, and we anticipate that FLOP-based measures are likely to be optimistic in predicting runtime overhead on typical deep learning hardware. However, we expect the efficiency gains of low-precision formats to vastly outweigh the scaling overhead.

Fusing scale factors We anticipate substantial efficiency gains from fusing the fixed scale factors from unit scaling into preceding ops. This yields two potential benefits. First, fusing avoids the communication overhead of an extra round-trip to memory. Second, it may permit low-precision outputs and even intermediate values. This may be particularly valuable for distributed aggregation ops, where partial results are aggregated on separate workers before sharing them to compute a final result.

Transformations implementing automatic fusing of ops are widely available using optimising compilers such as XLA (XLA and TensorFlow teams, 2017). These are particularly effective at fusing consecutive elementwise ops, which should encompass most unit scaling factors (since matmul outputs are typically first used in add or activation functions).

J. Additional experimental details and results

J.1. Character language modelling

The WikiText-103 raw dataset consists of approximately 500 million characters of text extracted from Wikipedia articles. We do not perform any additional preprocessing beyond that of the published dataset. All results correspond to the best value over a learning rate sweep starting from a low value, with step $2^{1/2}$. A complete set of hyperparameters used is shown in Table A.3.

Mixed precision When running in FP16, all activations, parameters and gradients are stored in FP16. Optimiser state is also stored in FP16, with the exception of Adam's second moment state, which is stored in FP32 since squared values are more prone to clipping.

Model architectures All models are based on causal Transformer-like stacks that interleave contextual (i.e. token-mixing) layers and FFN layers. Input tokens are embedded by indexing into a trainable embedding table, and output token probabilities are generated by $\text{softmax}(W_{\text{proj}} \cdot \text{layernorm}(x_L) + b_{\text{proj}})$, where x_L is the final hidden state from the Transformer stack.

The basic unscaled layer definition follows:

$$x_{l+1} = \text{res}(\text{seq}; \text{res}(\text{context}; x_l))$$
$$\text{res}^{\text{NoNorm}}(f; z) = \text{interp}(z; f(z))$$
$$\text{res}^{\text{PreNorm}}(f; z) = \text{interp}(z; f(\text{layernorm}(z)))$$
$$\text{res}^{\text{PostNorm}}(f; z) = \text{layernorm}(\text{interp}(z; f(z)))$$
$$\text{interp}^{\text{default}}(a; b) = a + b$$
$$\text{interp}^{\text{fixed}}(a; b; \alpha) = \sqrt{1-\alpha}\, a + \sqrt{\alpha}\, b$$
$$\text{interp}^{\text{mean}}(a; b; l) = \sqrt{l/(l+1)}\, a + \sqrt{1/(l+1)}\, b$$
$$\text{seq}(z) = W_2 \max(0; W_1 z + b_1) + b_2$$

The contextual layers are as follows:

1. $\text{context}^{\text{Attention}}$: multi-head dot product self attention using causal masking (Vaswani et al., 2017), with

Figure A.3.Comparison of residual scaling approaches. We observe (a) for regular models, default scaling performs similarly to xed interpolation = 0:5; (b) in most cases, running-mean scaling is similar or better than xed interpolation. The exception is 2-layer attention models, where we hypothesise that running mean places too much weight on the rst layer, which is detrimental in such a shallow model.

relative-positional encoding using sinusiodal bases following Dai et al. (2019),

2. context^{Conv} : 1D grouped causal convolution with relu nonlinearity,

3. context^{RNN} : recurrent highway network (Zilly et al., 2017) with tied transform and carry gatesx_{t+1} = (1 $g(x_t)$) x_t + $g(x_t)$ $f(x_t)$, where$g(x)$ is a projection with sigmoid nonlinearity, and$f(x)$ is a projection with tanh nonlinearity.

When applying unit scaling, we also reduce the learning rate for non-projection parameters by$\sqrt{\text{hidden_size}}$to compensate for the relative step size increase implied by unit scaling.

Additional results Test set results, with multiple runs per learning rate are shown in Table A.4. These support the main ndings shown for the wider sweep of Figure 4: unit-scaled models perform comparably to regular models, and can be trained in FP16 without modi cation or additional hyperparameter selection.

Figure A.3 shows the effect of employing residual scaling schemes described in Section 5.2. This supports the claim that xed and running-mean residual scaling are viable alternatives to default scaling, since both perform well in regular and unit-scaled models.

J.2. Masked language modelling

We follow the standard practice of splitting BERT pre-training into two phases. For the rst phase we use a sequence length of 128 tokens, and for the second we use 384. Tokens are derived using the WordPiece tokeniser (Wu et al.,

Table A.3.Character language modelling hyperparameters.

Parameter	Value
Sequence length	256 characters
Sequence mask	32 characters
Batch size	2048 characters
Training duration	2^{19} steps
Learning rate decay half-life	2^{16} steps
Adam($_1$; $_2$)	(0.9, 0.999)
SGD momentum	0.9
Vocabulary size	5008 characters (100% coverage, no OOV)
Hidden size	128
FFN size	512
Depth	[2, 8] layers
Attention heads	2
Attention head size	64
Relative positional frequency components	128 bases, period [1 ...1024] characters
Convolution kernel size	7
Convolution group size	16
Typical learning rate ranges:	
Regular, SGD	$2^{-8}:::2^{-4}$
Regular, Adam	$2^{-12}:::2^{-8}$
Unit, SGD	$2^{-14}:::2^{-10}$
Unit, Adam	$2^{-8}:::2^{-4}$

Table A.4.Character language modelling, test BPC with 3 runs per learning rate. The best learning rate is chosen according to validation BPC. 95% con dence interval is0:010. All models use PreNorm and 8 layers, except where noted.

Model	Regular FP32	Unit scaling FP32	Unit scaling FP16
Attention (PostNorm)	1.548	1.540	1.540
Attention	1.582	1.562	1.567
Convolution	1.625	1.620	1.622
RNN (2 layers)	1.674	1.677	1.673

Table A.5. BERT pre-training hyperparameters.

Parameter	Value
Sequence length	[128, 384] tokens (phase 1/2)
Depth	[12, 24] (base/large)
Hidden size	[768, 1024] (base/large)
FFN size	[3072, 4096] (base/large)
Attention heads	[12, 16] (base/large)
Attention head size	64
Vocabulary size	30400
Total batch size	[16320, 4080] seqs (ph. 1/2)
Micro-batch size	[8, 2] (phase 1/2)
Data-parallel count	4
Gradient accumulation count	510
Training duration	[28266, 8437] steps (ph. 1/2)
Learning rate	[0.0045, 0.0015] (phase 1/2)
Warmup steps	[2827, 275] steps (phase 1/2)
Learning rate decay	linear
Optimiser	LAMB
LAMB Beta1	0.9
LAMB Beta2	0.999
LAMB epsilon	1e-06
Weight decay	0.01
Weight init std	0.02 (unit scaling=n/a)
Loss scaling	[512, 512, 32768, 128] (base phase 1/2, large phase 1/2; unit scaling=n/a)

2016), with a vocabulary of 30400 tokens. Our masking approach is consistent with that used in Devlin et al. (2019). A complete set of pretraining hyperparameters used is shown in Table A.5.

Mixed precision For FP16, we follow the same approach here as in our character language modelling experiments (appendix J.1), storing all tensors and optimiser state in FP16, apart from the optimiser second moment state which is stored in FP32 (note, we use the LAMB optimiser (You et al., 2019) here over Adam).

For FP8, we modify our FP16 mixed precision strategy by quantising the inputs to all matmul operations. Note that our experiments do not utilise hardware FP8 support; we instead simulate FP8 training by quantising from FP16 to the set of supported values in a given FP8 format. In this, we are following the approach taken by Noune et al. (2022) and Micikevicius et al. (2022). As recommended in both studies, we also use E4 for activations and weights, and E5 for all gradients. Again, following the precedent set in these studies, the one matmul operation we exclude from FP8 quantisation is the vocabulary embedding matmul, which has been known to cause numerical instabilities.

Hardware & distributed training Models were trained on IPU hardware (Jia et al., 2019), using either Bow Pod16 or IPU-POD16 Classic machines. On each machine we distribute training across 16 IPUs, using 4-way model parallelism and 4-way pipeline parallelism, with gradient accumulation across pipeline stages.

K. Histograms of tensor-scaling within BERT

To give readers a better intuitive sense of how loss scaling and unit scaling operate for a standard model, we provide histograms of absolute tensor values taken from FP16 $\text{BERT}_{\text{BASE}}$.

Figures A.4 and A.5 show the beginning of training for loss and unit scaling respectively, and Figures A.6 and A.7 show the end of training.

We use 9 transformer layers rather than the standard 12 in order to accommodate the overheads of tracking histograms across all tensors in the model. For the sake of concision we omit histograms of the middle layers, which are substantially similar to layers 0 and 7 in both the forward and backward pass. A small number of numerically insigni cant ops are also omitted.

The rst two gures can be understood as the full-model equivalent to the plot in Figure 1, with the second two showing how values shift as a result of training. The x-axis is labelled slightly differently to Figure 1, showing the log of absolute values rather than the exponent value, but by

the de nition of oating point values given in Section 3.1, these two are approximately equivalent. We also have a special bin for the range $2^{-24}; 2^{-14}$, which represents all subnormal values in the FP16 range, and bins on either end to hold zero and in nity values.

There are some surprising features in the shapes of these plots, resulting from the design of BERT. We provide a brief analysis here of our key plot: Figure A.5 (unit scaling at initialisation).

K.1. Analysis of Figure A.5

Impact of unit scaling A comparison with Figure A.4 demonstrates the effectiveness of unit scaling. Whereas the loss-scaled model has to tune a hyperparameter to centre the two gradient sub-plots, unit scaling does this naturally. Furthermore, values in the unit-scaled model are typically closer to the centre of the range. Loss scaling also has the problem of very large grad_x values in its NSP and MLM heads.

Effect of aligning with regular BERT As outlined in Appendix H.5, we take a range of measures to align our unit scaled model more closely with the regular BERT base model, so that their performance is similar. This has the impact of temporarily mis-scaling certain operations. This can be seen most clearly in the case of gelu, which requires a scaling factor for alignment, but as a result is slightly below unit-scale in the diagram.

Sparse gradients for layer 8 The grad_x values for layer 8 in all plots have most of their values set to zero. This is a consequence of sparse gradients owing back into this layer from the NSP and MLM heads, as described in Appendix H.5. The cross-sequence mixing of gradients in the multi-head self-attention layer has the effect of removing this sparsity, giving a strong signal for all subsequent layers.

Three groups of gradient scales Our nal observation is somewhat subtle, but key to understanding both the shape of the grad_x plots, and the particular dif culties encountered when training BERT in low-precision.

We note that in the grad_x plots there are in effect three separate `columns' visible: a strong signal (i.e. many values) on the left, a faint signal through the centre, and a very small number of values on the right. This is a consequence of BERT's design, rather than of any scaling technique.

The right-hand column is a result of the natural up-scaling of gradients owing from BERT's NSP head. BERT naturally has larger gradients owing out of this head. Note that these gradients are sparse, representing only a single token-gradient in each sequence, but the signal is kept alive throughout the layers by the residual connection, resulting in this feature of the plot.

The central column comes out of the MLM head in a similar fashion. This is still sparse, but contains more token-gradients and hence gives a stronger signal. Finally the main left-hand column results from the mixing of gradients in the multi-head self-attention layer. This removes sparsity in the tensor, giving a stronger signal. However, the attention mechanism in BERT naturally lowers the scale of values, meaning this third signal is shifted to the left.

The existence of these three groups of gradients creates a trimodal distribution of exponent values. As most values are still concentrated in the left-hand column, our assumption of a single normal distribution is still suf cient, but we effectively have to balance the positions of these three columns, meaning that the backward pass does not fall into a single, neat column.

Figure A.4.A histogram of absolute values in regular $\text{BERT}_{\text{BASE}}$ at initialisation. Here a loss scale of 2^{15} was required for stable training. We can understand loss scaling in light of this plot as enacting a shift of the x and grad w histograms by $\log_2(\text{loss scale})$ to the right.

Figure A.5.A histogram of absolute values in unit-scaled $\text{BERT}_{\text{BASE}}$ at initialisation. Unit scaling naturally places values in approximately the centre of the range without requiring a tuned hyperparameter. See Appendix K.1 for specific details of this plot.

Figure A.6. A histogram of absolute values in **regular** $\text{BERT}_{\text{BASE}}$ at the **end of training**. Compare with figure A.4 to see the shift in distributions during training and the implications for numerics.