How tt-2 Is Trained, and Training on Every Byte a Fixed Number of Times

Claude and MJC

2026-11-26

tt-2 is a function: given up to 128 bytes, it gives a probability to each of the 256 values of the byte after them. Its 115,008 numbers are set by training on the first 10^7 bytes of enwik9. Section 1 is how that is done now. Section 2 is a change with two parts, proposed by MJC: train only the predictions that compression will use, and give every byte the same number of them. Section 3 is what is left to decide.

1 Training now

A window is 129 consecutive bytes of the training text, from a random offset. The model is run on the first 128, and position t of the window predicts byte $t+1$ from bytes $0..t$. So one window gives 128 predictions, from $1, 2, \ldots, 128$ bytes of context. They cost one run because each head reads only positions at or before its own: the output at t is exactly what the model would give if handed bytes $0..t$ alone. Nothing is carried from one window to another.

A batch is 32 windows, each from its own offset, drawn uniformly and independently from the 10^7 bytes; they may overlap. That is 4,096 predictions, and the batch's *loss* is their average cost, $-\ln p(\text{true byte})$.

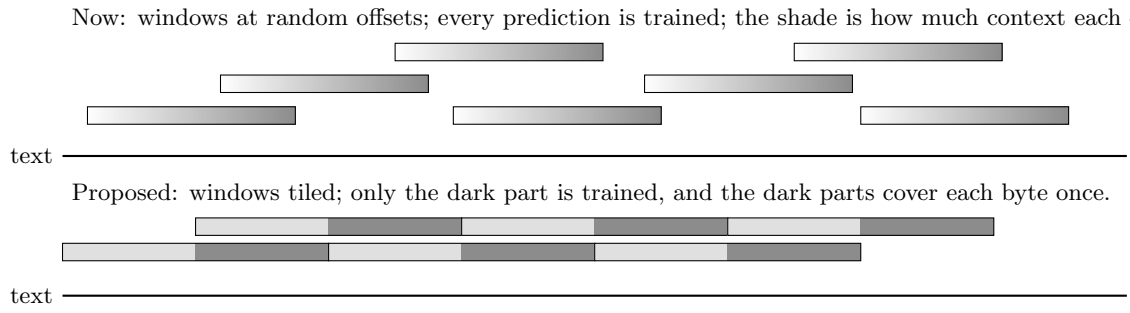
A step changes every number once:

1. The *gradient*: the exact derivative of the batch's loss with respect to each of the 115,008 numbers (by the chain rule; "backpropagation" is the name for computing it efficiently).
2. *Clipping*: if the gradient, as one vector of 115,008 numbers, is longer than 1, it is scaled to length 1.
3. The update ("AdamW"): each number keeps a running average of its gradient (each step keeps 0.9 of the old average) and of its gradient squared (0.95). It moves by $-\text{lr} \times (\text{average gradient})/\sqrt{\text{average square}}$, a step of about lr whatever the size of its gradients. Each entry of a matrix or of E is also moved toward 0 by $\text{lr} \times 0.1$ times its value (*weight decay*); the gains are not.

The learning rate lr rises in a straight line from 0 to 0.003 over the first 200 steps, then falls along half a cosine to 0.0003 at the last step.

Epochs. "8 epochs" sets only the total: 8×10^7 predictions, 19,531 steps of 4,096. The offsets being random, the number of times a byte is predicted is close to Poisson with mean 8 (about $e^{-8} \times 10^7 \approx 3{,}400$ bytes are never predicted), and each time its place in the window, so the length of context it is predicted from, is uniform on 1..128.

The start. Before step 0 the numbers are random (normal; the gains are 1). A *seed* fixes them and the sequence of offsets; with the seed and the number of threads fixed, a run is repeated exactly. tt-2 and tt-2-s2 differ only in seed. tt-2 took 95 minutes on two cores.



What this means for us. Two things in it are at odds with how the model will be used and read.

- *Compression uses one prediction per byte*, from its full context: the last 128 bytes for tt-2, the last 31 for tt-2-span16. Training spends most of its predictions on less than that. The scale of the effect is seen in tt-2-ctx32, trained on windows of 32: it got better at short contexts and worse at long ones, 0.062 bpc behind tt-2 with 32 bytes visible (the previous paper).
- *How many times a pattern was trained on is random.* A pattern that occurs 5 times in the text was predicted, in training, a number of times that is a sum of 5 Poisson draws, from random context lengths. Relating what the model learned from it to how often it was seen carries that error.

2 The proposal: tiled windows, full contexts, a fixed number of passes

Let R be the reach: how many bytes before t can affect the prediction at t (30 for tt-2-span16, 127 for tt-2). In a window of 128, the positions $R..127$ have their full context, so their predictions are *exactly* the ones compression makes. The rule:

1. Tile the text with windows of 128 that overlap by R bytes (step $128 - R$). Train only positions $R..127$ of each window; in the first window, which starts the file, all of them, since compression has nothing more for those bytes either. Then **every byte is predicted exactly once per pass, from its full context**, the same prediction compression would make.
2. A *pass* is every window once. Train for a fixed number of passes, P. The windows are put in a shuffled order, a new one each pass, from the seed.
3. Batches hold about 4,096 trained predictions, as now; learning rate, clipping and the update as now.

	reach R	trained positions per window	windows per pass	time for $P = 8$, estimated
tt-2-span16	30	98 (of 128)	102,041	2.3 hours
tt-2, exact	127	1	10^7	about 200 hours
tt-2, from 64 to 128	–	64	156,249	3.2 hours

Times are scaled from the runs so far (95 minutes for tt-2, 106 for tt-2-span16, per 625,000 windows).

For tt-2-span16 the rule is exact and cheap: 98 of every 128 predictions are used, and each is the compression prediction. **For tt-2 it is exact only at one position per window**, 128 times the work per prediction, which is out of reach on this machine. The third row is the nearest affordable form: train positions 63..127, so each byte is predicted from between 64 and 128 bytes, always the same number for a given byte. It is not exact, but measured on tt-2 the cost with 64 visible is within 0.002 of 128 on training text and 0.0003 on unseen (64 visible: 1.9311 and 2.1859; 128: 1.9333 and 2.1862). Its compression would then use the same 64..128 rule.

What the change makes exact. After P passes, every occurrence of a pattern was predicted exactly P times, and appeared in the context of exactly the predictions that follow it within the reach, P times each. Counts in the text and counts in training become the same number times P.

What it does not. What the model takes from a pattern still depends on more than how many times it was seen: on when (the learning rate falls by 10× over the run), on what else was in the same batch, and on the random start. Fixing the seed fixes all of these, so the run is repeatable, but two seeds will differ. That difference is the size of this remaining term, and two seeds of the current scheme differ by 0.002 bpc on training text overall. The proposal removes the part of the error that comes from the counts; the rest can be measured with a second seed if we need it.

What we expect. Better on full contexts and worse on short ones than now, as tt-2-ctx32 was in the other direction. Measured with `tt ctx` as in the previous paper, on the same bytes, so the two schemes can be compared directly.

3 To decide

1. **Which model**: tt-2 (128 bytes, trained with the 64..128 rule) or tt-2-span16 (31 bytes, trained exactly). MJC leans to tt-2. The change to training favours tt-2-span16, the only one of the two for which "trained on every byte P times, as compression predicts it" holds exactly; it also costs 0.022 bpc on training text (previous paper). Either can be trained: the code change is the same for both.

2. **How many passes**, P. Now it is 8; we propose keeping 8, so the result is comparable to tt-2.