# Supplemental Material for "Towards Packing: 2x NLP BERT Acceleration"

Anonymous, Authors

## A Packing SQUaD 1.1

We tokenized SQUaD [19] for BERT [5] with maximum sequence length 384 and visualized the histogram over the sequence length (Figure 5). The distribution looks similar to the Wikipedia dataset but is slightly less skewed. However, the maximum sequence length only had an occurrence of $1.2\%$ compared to $23.5\%$. Hence, the theoretical un-padding speedup is 2.232. In Table 2, we can see that SPFHP does not concatenate more than 3 samples and obtains $97.54\%$ efficiency in contrast to a maximally used depth of 16 with $99.60\%$ efficiency on Wikipedia, because of the less skewed distribution. Note that we have less than $90'000$ samples. Hence, NNLSHP is less efficient because the rounding in the residuals has a much larger impact compared to more than 16 million sequences in the Wikipedia dataset.

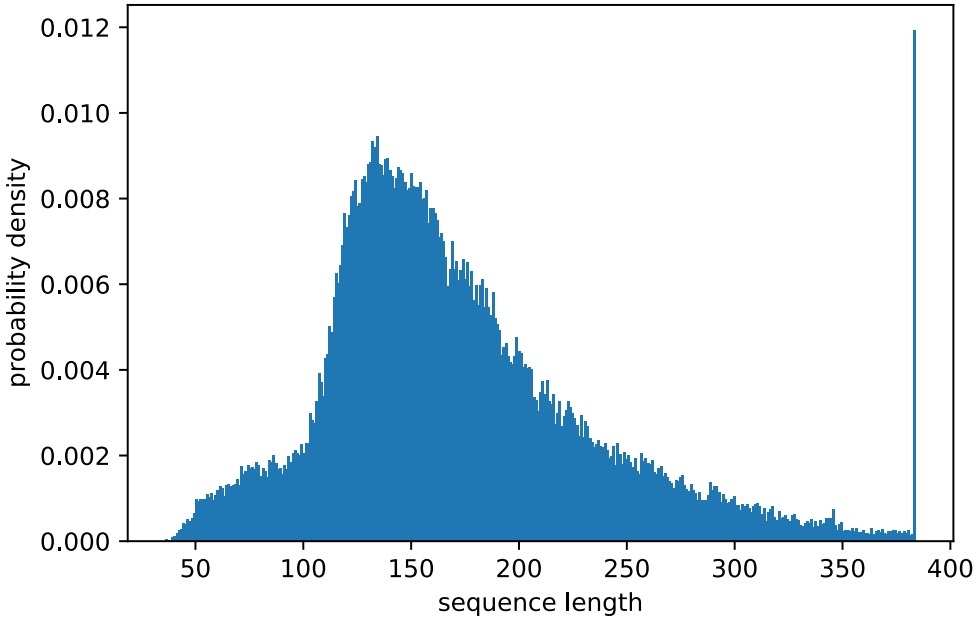


Figure 5: SQUaD 1.1 BERT pre-training dataset sequence length histogram for maximum sequence length of 384.

Table 2: Performance results of proposed packing algorithms for SQUaD 1.1 BERT pre-training.

| packing depth | packing algorithm | # strategies used | # packs | # tokens | # padding tokens | efficiency (%) | packing factor |
|---|---|---|---|---|---|---|---|
| 1 | none | 348 | 88641 | 34038144 | 18788665 | 44.801 | 1.000 |
| 2 | SPFHP | 348 | 45335 | 17408640 | 2159161 | 87.597 | 1.955 |
| 3 | NNLSHP | 398 | 40808 | 15670272 | 420793 | 97.310 | 2.172 |
| 3/max | SPFHP | 344 | 40711 | 15633024 | 383545 | 97.547 | 2.177 |

## B Packing GLUE

To explore a variety of datasets and emphasize that skewed distributions are common, we explored all datasets in the GLUE benchmark [23, 22] that came with training data. We loaded the datasets using the HuggingFace dataset loading API [36]. For preprocessing, we followed the implementation in the HuggingFace transformers repository [35] [2] and extracted the respective data processing snippets to obtain tokenized data with a maximum sequence length of 128. The histogram of the sequence length for each of the included datasets is displayed in Figure 6 and the packing results are given in Table 3. Each dataset benefits from packing. The lower the mean, the higher the packing factors are that can be reached but with a higher packing depth.

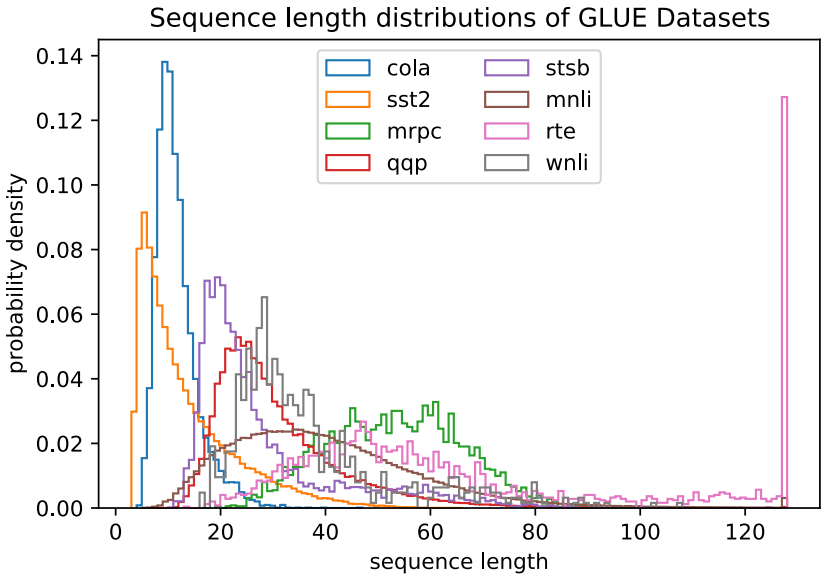


Figure 6: GLUE dataset sequence length histograms for maximum sequence length of 128.

Table 3: Performance results of proposed packing algorithms for the GLUE dataset. Only the baseline and the SPFHP packing results without limiting the packing depth are displayed.

| data name | packing depth | # strategies used | # packs | # tokens | # padding tokens | efficiency (%) | packing factor |
|---|---|---|---|---|---|---|---|
| cola | 1 | 34 | 8551 | 1094528 | 997669 | 8.849 | 1.000 |
| cola | 13/max | 29 | 913 | 116864 | 20005 | 82.882 | 9.366 |
| sst2 | 1 | 64 | 67349 | 8620672 | 7723633 | 10.406 | 1.000 |
| sst2 | 15/max | 64 | 7691 | 984448 | 87409 | 91.121 | 8.757 |
| mrpc | 1 | 77 | 3668 | 469504 | 274214 | 41.595 | 1.000 |
| mrpc | 4/max | 74 | 1606 | 205568 | 10278 | 95.000 | 2.284 |
| qqp | 1 | 123 | 363846 | 46572288 | 35448844 | 23.884 | 1.000 |
| qqp | 5/max | 123 | 97204 | 12442112 | 1318668 | 89.402 | 3.743 |
| stsb | 1 | 85 | 5749 | 735872 | 575993 | 21.726 | 1.000 |
| stsb | 6/max | 83 | 1367 | 174976 | 15097 | 91.372 | 4.206 |
| mnli | 1 | 124 | 392702 | 50265856 | 34636487 | 31.093 | 1.000 |
| mnli | 8/max | 124 | 123980 | 15869440 | 240071 | 98.487 | 3.167 |
| rte | 1 | 112 | 2490 | 318720 | 152980 | 52.002 | 1.000 |
| rte | 4/max | 108 | 1330 | 170240 | 4500 | 97.357 | 1.872 |
| wnli | 1 | 72 | 635 | 81280 | 57741 | 28.960 | 1.000 |
| wnli | 6/max | 63 | 192 | 24576 | 1037 | 95.780 | 3.307 |

[2] https://github.com/huggingface/transformers/blob/master/examples/text-classification/run_glue.py

# C Further learning curves

This section provides further learning curves related to Section 4.2.

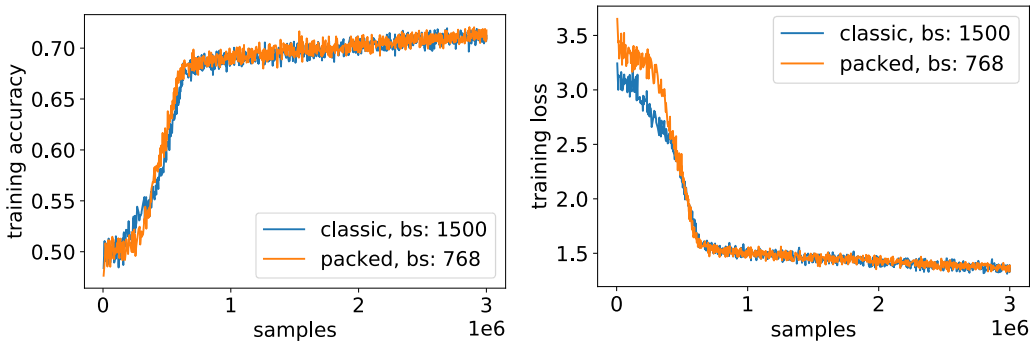


Figure 7: Comparison of learning curves for packed and unpacked processing with **reduced batch size** for the packed approach.

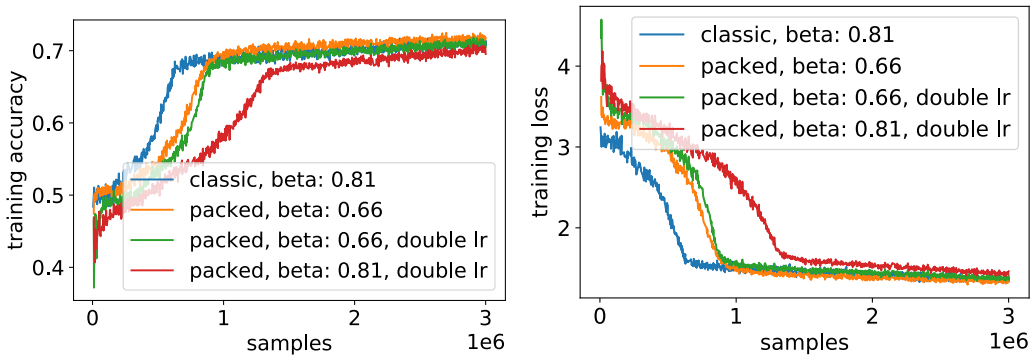


Figure 8: Comparison of learning curves for packed and unpacked processing with **heuristics** applied.

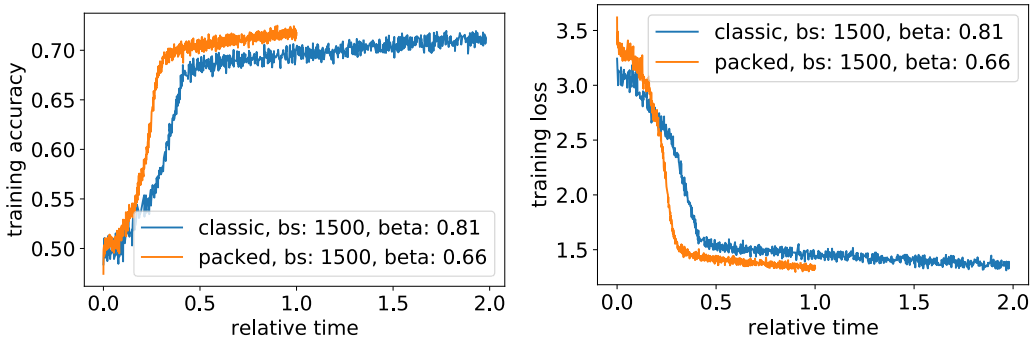


Figure 9: Comparison of learning curves for packed and unpacked processing in the **optimized setup**.

# D Technical background on packing

## D.1 Canonical packing problem

The bin-packing problem deals with the assignment of items into bins of a fixed capacity such that the number of utilized bins is minimized. In the canonical formulation of the packing problem a vector $s(i)$ of length $n$ is used to represent the items being packed, where $s(i)$ denotes the length of the i-th sequence/item. The allocation of items into bins is tracked through the assignment matrix $B$, where $B_{ij} \in \{0, 1\}$ states whether the i-th sequence should be placed into the j-th bin. In the worst case scenario, every item is assigned to its own bin, thus $B \in \mathbb{R}^{n \times n}$. Notably, $s$ grows linearly in the number of sequences/items being packed and $B$ grows with the square. To mask out unused bins $y_j \in \{0, 1\}$, denotes whether the j-th bin is being used. The optimization objective is to minimize the sum of $y_j$ while making sure to assign each $s_i$ to exactly one bin and not exceeding the maximum bin capacity $s_m$ for each bin. This problem formulation is well known as bin-packing [13].

$$
\begin{aligned}
\min_{y \in \{0,1\}^n, B \in \{0,1\}^{n \times n}} \quad & \sum_{j=1}^{n} y_j && && \text{Minimize the number of bins.} \\
\text{s.t.} \quad & \sum_{j=1} b_{ij} = 1 && \forall i && \text{Assign each length/sequence to only one bin.} \\
& \sum_{i=1}^{n} s(i) b_{ij} \leq s_m y_j && \forall j && \text{Cumulative length cannot exceed capacity.}
\end{aligned}
\tag{1}
$$

Bin-packing is a strongly NP-complete [13] problem. Producing an exact and optimal solution is possible with a variety of existing algorithms, for example with the branch-and-cut-and-price algorithm [27]. However, given that we want to apply it for very large $n$ (16M for the Wikipedia dataset) an approximate approach is required.

## D.2 Approximate bin-packing problem

Approximate packing approaches are divided into online and offline algorithms [11]. Online algorithms process incoming sequences one-by-one in a streaming fashion, whereas offline algorithms have a holistic view of all samples to be packed but typically still operate on a per sample basis. This results in best case time and memory complexities of at least $O(n \log(n))$ and solutions that can sometimes be far from optimal, especially for the online algorithms which do not have access to a holistic view of the datasets. The simplest online approach (next-fit) would be to keep a single open bin at any given time. An incoming sequence is added to this open bin if it fits, otherwise the bin is closed (can never be appended to again) and a new one is opened to accommodate the new sequence [11]. In the case of the Wikipedia pre-training dataset almost $25\%$ of the sequences are of length $512$, which makes this approach very inefficient since bins would frequently be closed because the incoming sequence did not fit. More specifically, this approach is not able to efficiently combine one long sequence with one shorter sequence, when the number of long sequences is large. The algorithms that come closest to the approaches proposed in this paper are the online harmonic-k algorithm [31], which creates harmonic sized bins for the assignment decision, and the offline Modified First Fit Decreasing method [12, 26], which sorts the data, groups it into $4$ size categories and defines a strategy adjusted to these sizes.

In our approaches, we make three major simplifications. We make the problem of bin packing less dependent on $n$ by operating on the histogram of sequence lengths with bin size 1. Hence, we replace $s(i)$ by its histogram $b$ and the bin assignment $y, B$ by a mixture of strategies $x$, where the set of all available packing strategies is modeled as the matrix $A$ (see also Section D.4.2).

Then, we do not solve the full packing problem but focus on a fixed packing depth (in other words the well known 3-partition problem). Last but not least, we solve the limited depth packing problem only approximately either with a non-negativity-constrained linear least squares [2] (NNLS) followed by rounding to nearest integer solution or by applying Worst-Fit [12, 26] to the histogram, sorted from largest to smallest (in contrast to using an unsorted dataset). An exact solution would not be appropriate, since the 3-partition problem is strongly NP-complete [28] as well.

### D.3 Definitions

In this section, we standardize the terms used throughout our methods. Firstly, the terms *pack* and *bin* may be used interchangeably. Secondly, the presented packing schemes impose a limit on how many sequences can be packed into any given bin. This limit is referred to as the maximum *packing depth*. For simplicity, we require the different sequence lengths in a pack to always add up exactly to the bin capacity $s_m$ (we can always generate a padding sequence of just the right length to fill-up the bin). A *packing strategy* is a sorted list of sequence lengths, for example $[5, 7, 500]$, such that the total sequence length is no more than $s_m$ and the number of sequences in the pack does not exceed the maximum *packing depth*. The output of a packing scheme is typically as set of *packing strategies* and the corresponding *repeat count* for each strategy stating how many times each strategy should be repeated in order to cover the entire dataset. The strategy *repeat count* is also referred to as the *mixture* of strategies. The objective of the packing algorithm is to jointly design a set of packing strategies and their repeat counts, such that the amount of *padding* is (approximately) minimized. The presence of *padding* in the packs can either be implicit or explicit. For instance for $s_m = 512$ the strategy [2, 508] has an implicit padding of 2 (needed to fill the pack up to the $s_m$). Alternatively, the strategy repeat count may over-subscribe a particular sequence length leading to explicit packing. For instance constructing a pack of [4, 508] may require a new *padding* sequence of length 4 be constructed, if there are not enough sequences of that length in the dataset. The packing algorithms, we present, use both representations.

### D.4 Non-negative least squares histogram-packing

The first algorithm proposed in this paper is suitable for settings where it is desirable to achieve a high packing efficiency with a limited packing depth. The algorithm is deterministic and has three major components described in Sections D.4.1, D.4.2 and D.4.3.

#### D.4.1 Enumerating packing strategies of fixed packing depth

Listing all unique ways of packing up to a maximum *packing depth* can be achieved through dynamic programming. We only consider packing at most up to 3 sequences per pack. This is the smallest packing depth that can eliminate the need for most padding on the Wikipedia dataset. Increasing the depth to 4, increases the size of the packing problem drastically and yields no throughput benefit [3]. With only two sequences, packing would be not as efficient since the distribution on sequence length is not symmetric. We use dynamic programming to enumerate all feasible ways/strategies that up to $M$ sequences of length $1 - 512$ can be packed into a bin of length $512$. For example, a packing strategy may be $[512]$ or $[6, 506]$ or $[95, 184, 233]$. To avoid listing the same strategy multiple times, we enforce the sequence lengths within a pack to occur in sorted order, for example, $[95, 184, 233]$ is equivalent to $[184, 95, 233]$ and should only be listed once. This reduces the search space as well as the space of potential solutions by a factor of $6$ approximately and thus significantly accelerates the optimization process. If you had the same strategy repeated $6$ times instead of having just one instance of that strategy with weight $X$, you will have six instances with weight $x/6$ (for example, or any other distribution). This would conflict with integer rounding of the solutions and with convergence of optimization algorithms.

#### D.4.2 Constructing the packing matrix

The number of rows in the packing matrix is equal to the number of different sequence length categories. For instance, if we are using a granularity of 1 token to distinguish between different sequence lengths, then there are "maximum sequence length" rows. Each column of the matrix corresponds to a valid packing strategy (given the depth of packing). An example packing matrix for fitting up to 3 sequences into sequence length $8$ is given in Table 4. Each column of the matrix represents a packing strategy. For instance, the first column represents the strategy [1, 1, 6] of packing two length-1 sequences and one length-6 sequence together to form a pack of length 8. The number of strategies (and columns in the matrix) is discussed in Section D.5. For a packing depth of 3 and maximum sequence length, we obtain around $\frac{s_m^2+6s_m+12}{12}$ strategies. For depth 4, around $\frac{s_m(s_m+4)(2s_m+1)}{288}$ more get added.

[3] For data distributions that are more skewed than Wikipedia this might look different.

Table 4: Example packing matrix for sequence length 8. Columns represent different kinds of packs. Rows represent the number of sequences in these packs with a certain length. The last column represents a pack with only a single sequence of length six.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 2 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 2 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 2 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

### D.4.3 Solution of the NNLS approximate packing problem

A solution of the packing problem is the mixture of packing strategies $x$ that minimizes the amount of padding in the packed dataset. We solve directly for the mixture (positive real numbers) and recover the padding as the negative portion of the residual (see Section D.4.4).

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \|A \cdot x - b\|^2 \\ \text{s.t.} \quad & x \geq 0 \end{aligned} \tag{2}$$

The solution vector $x$ will represent the mixture of the columns of $A$, in other words the mixture of valid packing strategies such that $A \cdot x$ is as close as possible (in the least squares sense) to the histogram of sequence lengths $b$. We obtain a solution with a non-negative least squares implementation [30, 34] Interestingly in the case of sequence length 512 only 634 out of the 22102 available packing strategies of depth up to 3 are used (3%).

### D.4.4 Padding as the residuals of the packing problem

We compute the residuals of the least squares solution (after rounding the mixture to integer) as:

$$r = b - A \cdot \mathit{round}(x) \tag{3}$$

The negative portion of the residuals represents sequences that we are "short". That is, there is a deficit of those sequences and we are over-subscribing to them. The positive portion of the residuals represents sequences which have failed to be packed. Typically, there is a deficit of short sequences and a surplus of long sequences as demonstrated by the following plot.

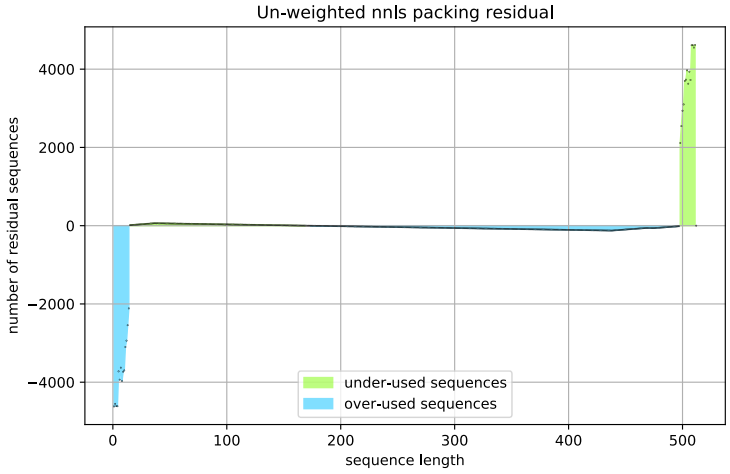


Figure 10: Visualization of the residual of the NNLS packing problem

In total, there are $n = 16\text{'}279\text{'}552$ sequences in the Wikipedia pre-training dataset. After the non-negative least squares packing (and rounding to integer solution) there are $56\text{'}799$ unpacked sequences left un-packed (about $0.352\%$). The residual on sequence lengths 1 to 8 are

$[-4620, -4553, -4612, -4614, -3723, -3936, -3628, -3970]$. These negative residuals imply that we need to add this many sequences of their corresponding sequence length to realize the mixture of packing strategies. In total the first iteration introduces $7.9410^6$ tokens of padding. In contrast large sequence lengths have a positive residual (a surplus of unused sequences). For sequence lengths $504$ to $512$ the values are $[3628, 3936, 3724, 4613, 4612, 4553, 4619, 0]$. Note that sequence length $512$ has a residual of $0$ since they do not need packing. Intermediate sequence lengths typically have non-zero (but much smaller) residuals.

The detailed code for the algorithm is provided in Listing 3.

### D.4.5 Residual weighting

A natural extension of the non-negative least squares problem introduced in Section D.4.3 is to weight the residuals on different sequence length differently.

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \|(wA) \cdot x - (wb)\|^2 \\ \text{s.t.} \quad & x \geq 0 \end{aligned} \tag{4}$$

We should not significantly penalize a deficit in short sequence lengths (smaller than $8$ tokens) as adding up to $8$ tokens of padding is not much overhead. Similarly, a surplus in long sequences is not worrisome because the amount of padding needed to achieve a sequence length of $512$ is small. Reducing the weight of the residual on the first $8$ tokens to $0.09$ leads to the following residual plot shown on the right in Figure 11. In this case the residual is almost entirely shifted to the shorter sequences and the positive residual on the longer sequences has virtual disappeared.

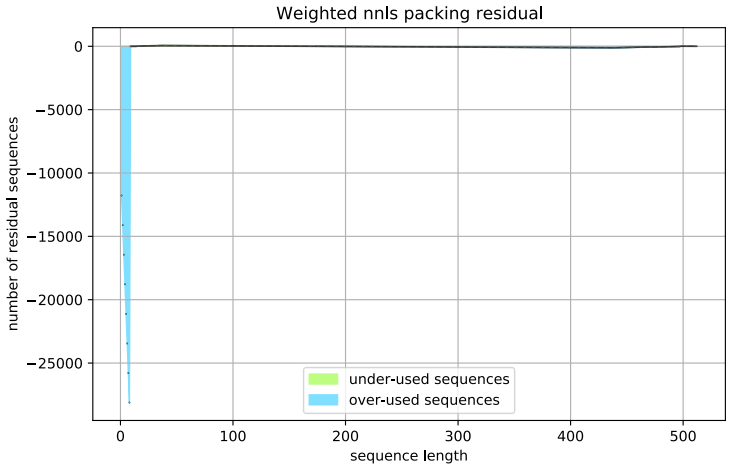


Figure 11: Visualization of the weighted residual of the NNLS packing problem

## D.5 Complexity analysis of the proposed packing approaches

Since approximate packing algorithms have a complexity of at least $O(n \log(n))$ and we would like to be able to tackle datasets with 2K million samples, we will discuss the complexity of our packing algorithms in this section. The complexity depends on the maximum sequence length $s_m$, the number of samples $n$, and the packing depth $d$.

To create the histogram, we have to iterate over the data once ($O(n)$). Our histograms will be binned by size 1, meaning one bin for each sequence length. Hence, a dictionary can be generated ($O(s_m)$) and used for the sorting ($O(1)$). The respective histogram vector has dimension $s_m$.

### D.5.1 Analysis of non-negative least-squares histogram-packing

For a packing depth of one, there is only the strategy $[s_m]$. For a packing depth of two, we add the strategies $[1, s_m - 1], ..., [s_m - \lfloor \frac{s_m}{2} \rfloor]$ which results in an additional $\lfloor \frac{s_m}{2} \rfloor$ potential strategies.

Following the dynamic programming approach, the number of possible additional strategies of depth three can be calculated with

$$
\begin{aligned}
\text{\# potential strategies} &= \sum_{j=1}^{\lfloor \frac{s_m}{3} \rfloor} \sum_{i=j}^{\lfloor \frac{s_m - j}{2} \rfloor} 1 = \sum_{j=1}^{\lfloor \frac{s_m}{3} \rfloor} \left\lfloor \frac{s_m - j}{2} \right\rfloor - (j-1) \\
&\approx \sum_{j=1}^{\lfloor \frac{s_m}{3} \rfloor} \frac{s_m}{2} - \frac{3}{2} j \approx \frac{s_m}{2}\frac{s_m}{3} - \frac{3}{2}\frac{s_m/3(s_m/3+1)}{2} \\
&\approx \left\lceil \frac{s_m^2}{12} \right\rceil
\end{aligned}
\tag{5}
$$

Note that for $s_m = 512$ the approximation is exact. This means that our strategy matrix $A$ has the dimensions $s_m \times \left( \left\lceil \frac{s_m^2}{12} \right\rceil + \lfloor \frac{s_m}{2} \rfloor + 1 \right)$. So it contains 11‘316‘224 numbers which is still much smaller than $n$. Note that the original data matrix $B$ had $n^2$ entries, which all needed to be optimized together with the $n$ bin assignments $y$. We now have only $\left\lceil \frac{s_m^2}{12} \right\rceil + \lfloor \frac{s_m}{2} \rfloor$ free variables in the strategy vector $x$. Also note that $A$ can be precomputed when $s_m$ is known and is independent of the number of samples. Given a problem matrix with dimension $i \times j$, Luo et al. [32] indicate that the asymptotic complexity of most solution approaches is $O(ij^2)$, whereas they propose an $O(ij)$ solution. Since we use the standard SciPy implementation [30], our estimated total complexity for NNLSHP is $O(n + s_m^5)$.

For $s_m = 2048$, the estimate would be $350'540$ potential strategies which is still far less than the number of samples.

For packing depth $4$, we calculate [37]:

$$
\begin{aligned}
&\sum_{k=1}^{\lfloor \frac{s_m}{4} \rfloor} \sum_{j=k}^{\lfloor \frac{s_m - k}{3} \rfloor} \sum_{i=j}^{\lfloor \frac{s_m - j - k}{2} \rfloor} 1 \\
&\approx \sum_{k=1}^{\lfloor \frac{s_m}{4} \rfloor} \sum_{j=k}^{\lfloor \frac{s_m - k}{3} \rfloor} \frac{s_m - k + 2 - 3j}{2} \\
&\approx \sum_{k=1}^{\lfloor \frac{s_m}{4} \rfloor} \frac{1}{12}(s + 4 - 4k)(s + 3 - 4k) \\
&\approx \frac{1}{288} s(2s^2 + 9s + 4) \\
&= \frac{1}{288} s(s+4)(2s+1)
\end{aligned}
\tag{6}
$$

So with $s_m = 512$, there would be around $940$K strategies. In our implementation, this number of strategies would be too high to create the problem matrix. One alternatives to simplify would be to not use the exact length of sequences but to only consider even numbers for the sequence length and round up. That way arbitrary sequence length could also be handled and the limiting factor would be the complexity of the attention layer in BERT which does not scale well with the sequence length.

### D.5.2 Analysis of shortest-pack-first histogram-packing

The complexity calculation of SPFHP is straightforward. We go over the whole data once for the histogram sorting. Next, we iterate over each of the $s_m$ bins in the histogram. Lastly, we iterate over all strategies that were encountered so far. It can be proven that, at each iteration, the number of strategies can be maximally increased by one. In each step, we potentially add a sequence to existing strategies but a new strategy is opened up only in the final step, when we either create a new strategy or we split one of the existing strategies into two. Hence, the number of strategies is bounded by $s_m$ and the overall complexity is bounded by $O(n + s_m^2)$.

# E Theorem on LAMB hyperparameter correction heuristic

With packing, the effective batch size changes and hence hyperparameters of the LAMB optimizer [25] need to be adjusted. For a packed dataset with a packing factor $p$, we update the decay parameters as: $\overline{\beta_1} := \beta_1^p$, $\overline{\beta_2} := \beta_2^p$. For instance if $\beta_1 = 0.81$ for the un-packed dataset, then for a packed dataset with an average of 2 sequences per sample one should use a value of $0.81^2 \approx 0.66$ instead. Assuming no or only minor changes in gradients and $p$ being a natural number, we can prove that this heuristic is the exact solution to make sure that momentum and velocity in LAMB are unaffected by packing. This can be proven by mathematical induction. Note that $p \geq 1$ by definition.

**Theorem 1.** *For any $p \in \mathbb{N}$ and assuming that respective gradients on a batch of $b$ random samples are (approximately) the same, choosing*

$$\overline{\beta_1} := \beta_1^p \tag{7}$$

$$\overline{\beta_2} := \beta_2^p. \tag{8}$$

*as hyperparameters in the LAMB optimizer ensures that the momentum and velocity after $p$ separate update steps are the same as with one packed update step with $p \times b$ samples.*

*Proof.*

- *Base Case*:
  For $p = 1$ the left and right side of the equation are the same which matches exactly the unpacked case. Hence, the theorem holds for $p = 1$.
- *Inductive hypothesis*: Suppose the theorem holds for all values of $p$ up to some $k$, $k \geq 1$.
- *Inductive proposition*: The theorem holds for $p = k + 1$.
- *Proof of the inductive step*: Let $l$ be the loss function, $w_t$ the weight vector after $t$ updates, and $x_1^t, \ldots, x_b^t$ the respective underlying data to calculate the gradient $g_t$. For a single update step in LAMB with batch size $b$ samples, we compute the gradient

  $$g_t = \frac{1}{b} \sum_{i=1}^{b} \frac{\partial l}{\partial w}(x_i^t, w^t). \tag{9}$$

  Since $g_1 \approx g_2 \approx \ldots \approx g_{k+1}$, We have with the inductive hypothesis and the definitions in LAMB:

  $$m_k = \beta_1^k m_0 + (1 - \beta_1^k) g_1 \tag{10}$$

  $$v_k = \beta_2^k v_0 + (1 - \beta_2^k) g_1^2 \tag{11}$$

  Now we can calculate (with $g_1 \approx g_{k+1}$)

  $$\begin{aligned} m_{k+1} &= \beta_1 m_k + (1 - \beta_1) g_{k+1} && (12)\\ &\approx \beta_1 \left(\beta_1^k m_0 + (1 - \beta_1^k) g_1\right) + (1 - \beta_1) g_1 && (13)\\ &= \beta_1^{k+1} m_0 + (1 - \beta_1^{k+1}) g_1 && (14) \end{aligned}$$

  The calculation for $v_k$ is the same. As reference for a packed update with $p = k + 1$ with $\overline{\beta_1}$ and $\overline{\beta_2}$, we would get

  $$g = \frac{1}{pb} \sum_{j=1}^{p} \sum_{i=1}^{b} \frac{\partial l}{\partial w}(x_i^j, w^1) = \frac{1}{p} \sum_{j=1}^{p} \left( \frac{1}{b} \sum_{i=1}^{b} \frac{\partial l}{\partial w}(x_i^j, w^1) \right) \approx \frac{1}{p} \sum_{j=1}^{p} g_1 = g_1 \tag{15}$$

  since we are calculating gradients over $b$ samples which are assumed to be approximately the same. Consequently, the updates for momentum and velocity would be

  $$\overline{m_k} = \overline{\beta_1} m_0 + (1 - \overline{\beta_1}) g_1 \tag{16}$$

  $$\overline{v_k} = \overline{\beta_2} v_0 + (1 - \overline{\beta_2}) g_1^2. \tag{17}$$

  Hence, $\overline{\beta_1} = \beta_1^{k+1}$ and $\overline{\beta_2} = \beta_2^{k+1}$ is required to map to the formula with the consecutive updates (for the same amount of data).

- *Conclusion*: The theorem holds for any $p \in \mathbb{N}$.

□

Since we proved that the formulas $\beta_1 := \beta_1^p,\ \beta_2 := \beta_2^p$. hold for all $p \in \mathbb{N}$, $p \geq 1$, it is safe to assume that it is an appropriate heuristic for all $p \in \mathbb{R}$, $p \geq 1$.

# F Un-padding scaling estimate

Firstly, we retrieve the per-batch processing time for an un-padding implementation running pre-training on the Wikipedia dataset from [17]. These processing times were obtained using 8 GPUs each with a per-device batch size of 32. We also retrieve the throughput numbers for the same system running with padding from [33] and use that as the baseline to compare the un-padded throughput against.

The throughput on the 8 GPU system is effectively limited by the slowest of the eight batches being processed in parallel. The Gumbel distribution is particularly suited to modelling the maximum or minimum value of a fixed size collection of i.i.d. samples (in this case batches). We observe that on 8 GPUs the throughput (i.e. speed-up) distribution indeed closely resembles a Gumbel distribution with $\alpha_1 = 1.6$ and $\beta_8 = 0.13$ as shown in Figure 12.

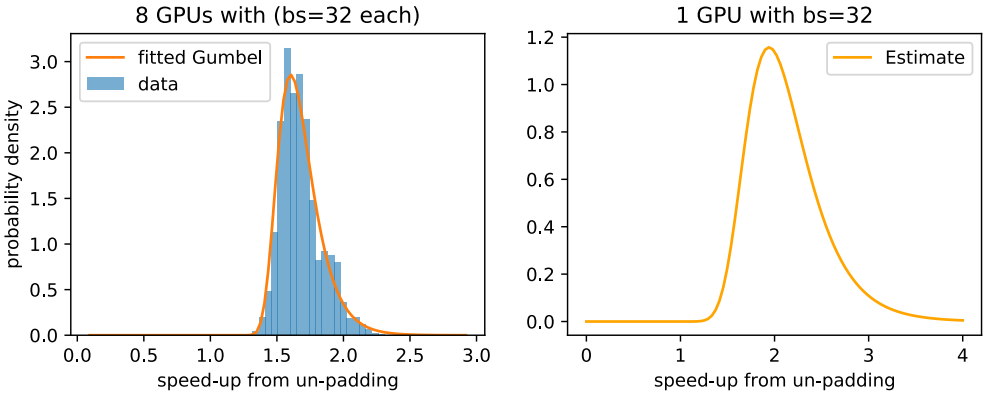


Figure 12: Left: Speed-up from un-padding on 8 GPUs closely resembles a Gumbel distribution. Right: statistical estimate of speed-up distribution on a 1 GPU system running un-padding

We can extrapolate the performance on the 8 GPU system to larger clusters by recognizing that the processing time for each cluster is effectively determined by the slowest batch being processed. Specifically, we could randomly sample (without replacement) two processing times for the 8 GPU system, and record the max of the two as the processing time for a system of 16 GPUs. However, this simple approach is too sensitive to outliers in the data and would result in an under-estimate of the performance of un-padding on large systems. We mitigate the effect of outliers in the data by avoiding directly sampling the processing times. Instead, we fit a Gumbel distribution to the processing times of a single batch of size 32 running on one GPU. To perform the fit, we observe that the cdf on one GPU ($P_1$) is related to the cdf on 8 GPUs ($P_8$) through [29](section 1.3):

$$(1 - P_8(s)) = (1 - P_1(s))^8 \tag{18}$$

In other words, if the speed-up on the cluster is larger than $s$, this implies that the speed-up on every GPUs in the cluster was at least $s$. Assuming $P_1$ is Gumbel and given the 8 GPU Gumbel parameters $\alpha_8$ and $\beta_8$, we need to fit two parameters, $\alpha_1$ and $\beta_1$. Consequently for the median ($s = \alpha_8 - \beta_8 \ln(\ln(2))$, $P_8(s) = 0.5$), we have:

$$0.5 = (1 - P_1(\alpha_8 - \beta_8 \ln(\ln(2))))^8 \,. \tag{19}$$

And since $P_8$ is Gumbel, we also have an equation for the mode ($s = \alpha_8$, $P_8(s) = e^{-1}$):

$$(1 - e^{-1}) = (1 - P_1(\alpha_8))^8 \,. \tag{20}$$

We solve these two non-linear equations simultaneously using the standard SciPy optimization package.

Listing 2: Infer Gumble distribution parameters.

```
import numpy as np
from scipy import stats, optimize
alpha_8 = 1.6038
beta_8 = 0.1288
```

```
def g(x):
    alpha_1, beta_1 = x
    dist = stats.gumbel_r(loc=alpha_1, scale=beta_1)
    # Equations for median and mode
    median = alpha_8 - beta_8*np.log(np.log(2))
    equation1 = 0.5 - dist.sf(median)**n_gpu
    mode = alpha_8
    equation2 = (1-np.exp(-1)) - dist.sf(mode)**n_gpu
    return (equation1**2 + equation2**2)

res = optimize.minimize(g, [alpha_8, beta_8], method="Nelder-Mead")
alpha_1, beta_1 = res.x
```

The resulting estimated speed-up Gumbel distribution for a single device has $\alpha = 1.94$, $\beta = 0.108$ and is shown in Figure 12 [right]. To simulate the performance of a cluster of size $n$ with a batch size of 32 per device, we take the minimum over $n$ samples from this distribution. Repeating this process to generate many samples allows us to estimate the expected speed-up for any given cluster size. Unfortunately, we cannot make any statistical inference about the processing times of individual sequences since the data is only provided at the granularity of 32 sequences per batch, and it is not clear how much of the computation is done in parallel and how much in serial.

# G Fine-tuned longest-pack-first histogram-packing

In the main paper, we focused on SPFHP due its simplicity. In this section, we analyse the effect of applying the "Best-Fit" algorithm [11]. Here, the longest pack that still fits the sequence is chosen instead of the shortest one. In contrast to SPFHP, we additionally consider splitting the histogram count, if it can fit multiple times. A simple example is sequence length $256$, where we divide the respective histogram count by 2 to create the optimal pack with strategy $[256, 256]$ instead of the strategy $[256]$. This latter strategy would be complemented by other sequences but would probably not result in an optimal packing. The implementation of this approach is much more complex than the SPFHP implementation. The code is provided in Listing 8 and the results in Table 5.

| pack. depth | # strat. used | # packs | # tokens | # padding tokens | efficiency (%) | pack. factor |
|---|---|---|---|---|---|---|
| 1 | 508 | 16279552 | 8335130624 | 4170334451 | 49.967 | 1.000 |
| 2 | 634 | 10099081 | 5170729472 | 1005933299 | 80.546 | 1.612 |
| 3 | 648 | 9090154 | 4654158848 | 489362675 | 89.485 | 1.791 |
| 4 | 671 | 8657119 | 4432444928 | 267648755 | 93.962 | 1.880 |
| 8 | 670 | 8207569 | 4202275328 | 37479155 | 99.108 | 1.983 |
| 16 | 670 | 8140006 | 4167683072 | 2886899 | 99.931 | 2.000 |
| 29/max | 670 | 8138483 | 4166903296 | 2107123 | 99.949 | 2.000 |

Table 5: Performance results of longest-pack-first histogram-packing for Wikipedia BERT pre-training with maximum sequence length $512$.

We can see that longest-pack-first histogram-packing (LPFHP) uses a much higher packing depth when no limit is set ($29$ instead of $16$). Splitting the histogram counts results in slightly higher numbers of used strategies compared to SPFHP where the number of used strategies is limited by the maximum sequence length. The best efficiency of LPFHP is $99.949\%$ with packing factor of $2$ which is slightly higher than the $99.75\%$ ($1.996$ packing factor) for NNLSHP and $99.6\%$ for SPFHP ($1.993$ packing factor). All algorithms are very close to the upper limit.

Note that for NNLSHP, we only fill up the unpacked samples with padding. Applying best-fit on the remains, similar results can be expected. Although the benefits of the improved algorithm are negligible, we share the concept and code below in case packing is applied to other data with a different distribution that would benefit more from it, or for applications where only perfectly packed sequences without padding are of interest.

### G.1 Extended NNLS with padding token weighting

In Section D.4.4, we defined the residual as

$$r = b - A \cdot \mathit{round}(x) \tag{21}$$

and discovered that a positive residual corresponds to sequences that we did not pack at all and should be avoided. Negative residuals correspond to padding and should be minimized. Due to this discrepancy, we decided to set small weights for very short sequences (that don't occur in the data). However, it was not possible to directly optimize the amount of padding. A negative residual component for length $i$, $r_i$, results in $|r_i| \cdot i$ padding tokens, however a positive residual actually results into $(512 - r_i) \cdot i$ padding tokens. This cannot be addressed by our weighting approach in

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \|(wA) \cdot x - (wb)\|^2 \\ \text{s.t.} \quad & x \geq 0 \end{aligned} \tag{22}$$

Working within the NNLS approach, we can strictly enforce a non-positive residual $r$ (before rounding to integer). To that end, we define a new auxiliary variable $\overline{r} \approx -(b - Ax)$ which is the negative of the residual, $r$. This will allow us to reformulate the objective $r \leq 0$ to the non-negative constraint: $\overline{r} \geq 0$.

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \|(wA) \cdot x - (wb)\|^2 + \|\overline{w} \cdot A \cdot x - \overline{w} \cdot b - \overline{w} \cdot \overline{r}\|^2 \\ \text{s.t.} \quad & x \geq 0 \\ & \overline{r} \geq 0 \end{aligned} \tag{23}$$

This will enforce $\overline{r} = Ax - b \geq 0$ due to the large weight, $\overline{w} := 10^6$, and no upper limits on $\overline{r}$. Now, we can set $w_i := i$ to optimize for the padding tokens. Due to the use of the squared error, we would however optimize the squared sum of padding tokens instead of the preferred sum of padding tokens. To accomplish the latter, we would have to replace the L2-norm problem by an L1-norm problem which would be too complex to solve. Note that due to rounding, the unwanted positive residuals $r$ ($\overline{r} < 0$) might still occur. This could be avoided by rounding up $x$ instead of normal rounding of $x$. To put the new formulation into a solver, we replace

$$b \text{ by } \begin{pmatrix} b \\ b \end{pmatrix}, \; x \text{ by } \begin{pmatrix} x \\ \overline{r} \end{pmatrix}, \; w \text{ by } \begin{pmatrix} w \\ \overline{w} \end{pmatrix}, \text{ and } A \text{ by } \begin{pmatrix} A & 0_m \\ A & -D_m \end{pmatrix}, \tag{24}$$

where $0_m$ is an $m \times m$ matrix with $m$ being the maximum sequence length, 512, and $D_m$ is a unit matrix of the same dimensions as $0_m$. Since, we are already close to optimum especially on the Wikipedia dataset, the results are only a little bit better. The processing time however increases from 30 to 415 seconds without considering the increased time for constructing the processing matrix. Since the slightly improved algorithm might be nevertheless relevant for other applications, we share it in Listing 9.

# H Packing source code

File links are currently non-functional for anonymous review but will be replaced in final version.

Listing 3: Non-negative least squares histogram-packing

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Non-Negative least squares histogram-packing algorithm."""
import time
import numpy as np
from scipy import optimize, stats
from functools import lru_cache

def get_packing_matrix(strategy_set, max_sequence_length):
    num_strategies = len(strategy_set)
    A = np.zeros((max_sequence_length, num_strategies), dtype=np.int32)
    for i, strategy in enumerate(strategy_set):
        for seq_len in strategy:
            A[seq_len - 1, i] += 1
    return A

@lru_cache(maxsize=None)
def get_packing_strategies(start_length, minimum_increment, target_length, depth):
    gap = target_length - start_length
    strategies = []
    # Complete the packing with exactly 1 number
    if depth == 1:
        if gap >= minimum_increment:
            strategies.append([gap])
    # Complete the sample in "depth" steps, recursively
    else:
        for new in range(minimum_increment, gap + 1):
            new_gap = target_length - start_length - new
            if new_gap == 0:
                strategies.append([new])
            else:
                options = get_packing_strategies(start_length + new, new, target_length, depth - 1)
                for option in options:
                    if len(option) > 0:
                        strategies.append([new] + option)
    return strategies

def pack_using_nnlshp(histogram, max_sequence_length, max_sequences_per_pack):
    # List all unique ways of packing to the desired maximum sequence length
    strategy_set = get_packing_strategies(0, 1, max_sequence_length, max_sequences_per_pack)
    print(f"Packing will involve {len(strategy_set)} unique packing strategies.")
    # Get the packing matrix corresponding to this list of packing strategies
    A = get_packing_matrix(strategy_set, max_sequence_length)
    # Weights that penalize the residual on short sequences less.
    penalization_cutoff = 8
    w0 = np.ones([max_sequence_length])
    w0[:penalization_cutoff] = 0.09
    # Solve the packing problem
    print(f"Sequences to pack: ", histogram.sum())
    start = time.time()
    strategy_repeat_count, rnorm = optimize.nnls(np.expand_dims(w0, -1) * A, w0 * histogram)
    print(f"Solving non-negative least squares took {time.time() - start:3.2f} seconds.")
    # Round the floating point solution to nearest integer
    strategy_repeat_count = np.rint(strategy_repeat_count).astype(np.int64)
    # Compute the residuals, shape: [max_sequence_length]
    residual = histogram - A @ strategy_repeat_count
    # Handle the left-over sequences i.e. positive part of residual
    unpacked_seqlen = np.arange(1, max_sequence_length + 1)[residual > 0]
    for l in unpacked_seqlen:
        strategy = sorted([l, max_sequence_length - l])  # the depth 1 strategy
        strategy_index = strategy_set.index(strategy)
        strategy_repeat_count[strategy_index] += residual[l-1]
    # Re-compute the residual with the updated strategy_repeat_count
    # This should now be strictly < 0
    residual = histogram - A @ strategy_repeat_count
    # Add padding based on deficit (negative residual portion of residual)
    padding = np.where(residual < 0, -residual, 0)
    # Calculate some basic statistics
    sequence_lengths = np.arange(1, max_sequence_length + 1)
    old_number_of_samples = histogram.sum()
    new_number_of_samples = int(strategy_repeat_count.sum())
    speedup_upper_bound = 1.0/(1 - (histogram*(1 - sequence_lengths / max_sequence_length)).sum()/
     old_number_of_samples)
    num_padding_tokens_packed = (sequence_lengths * padding).sum()
    efficiency = 1 - num_padding_tokens_packed/(new_number_of_samples*max_sequence_length)
    print(f"Packing efficiency (fraction of real tokens): {efficiency:3.4f}\n",
          f"Speed-up theoretical limit: {speedup_upper_bound:3.4f}\n",
          f"Achieved speed-up over un-packed dataset: {old_number_of_samples/new_number_of_samples:3.5f
     }")
    return strategy_set, strategy_repeat_count
```

Listing 4: Shortest-pack-first histogram-packing

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Shortest-pack-first histogram-packing."""
from collections import defaultdict
import numpy as np

def add_pack(pack, count, tmp, final, limit, offset):
    """Filter out packs that reached maximum length or number of sequences."""
    if len(pack) == limit or offset == 0:
        final[offset].append((count, pack))
    else:
        tmp[offset].append((count, pack))

def pack_using_spfhp(histogram, max_sequence_length, max_sequences_per_pack):
    """Shortest-pack-first histogram-packing algorithm."""
    reversed_histogram = np.flip(histogram)
    # Initialize main strategy data dictionary.
    # The key indicates how many tokens are left for full length.
    # The value is a list of tuples, consisting of counts and respective packs.
    # A pack is a (sorted) list of sequence length values that get concatenated.
    tmp_strategies_per_length = defaultdict(list)
    strategies_per_length = defaultdict(list)
    # Index i indicates here, how much space is left, due to reversed histogram
    for i in range(max_sequence_length):
        n_sequences_to_bin = reversed_histogram[i]
        length_to_bin = max_sequence_length - i
        offset = i + 1  # largest possible offset
        while n_sequences_to_bin > 0:
            if (length_to_bin + offset) in tmp_strategies_per_length:
                # extract shortest pack that will get modified
                n_sequences_to_pack, pack = tmp_strategies_per_length[
                    length_to_bin + offset].pop()
                new_pack = pack + [length_to_bin]
                count = min(n_sequences_to_pack, n_sequences_to_bin)
                if n_sequences_to_pack > n_sequences_to_bin:
                    # old pack gets reduced
                    n_sequences_to_pack -= n_sequences_to_bin
                    tmp_strategies_per_length[length_to_bin + offset].append(
                        (n_sequences_to_pack, pack))
                    n_sequences_to_bin = 0
                else:
                    n_sequences_to_bin -= n_sequences_to_pack
                add_pack(new_pack, count,
                         tmp_strategies_per_length, strategies_per_length,
                         max_sequences_per_pack, offset)
                # clean up to speed up main key search
                if not tmp_strategies_per_length[length_to_bin + offset]:
                    tmp_strategies_per_length.pop(length_to_bin + offset)
            else:
                offset -= 1
            # Does not fit anywhere. Create new pack.
            if offset < 0:
                add_pack([length_to_bin], n_sequences_to_bin,
                         tmp_strategies_per_length, strategies_per_length,
                         max_sequences_per_pack, i)
                n_sequences_to_bin = 0
    # merge all strategies
    for key in tmp_strategies_per_length:
        strategies_per_length[key].extend(tmp_strategies_per_length[key])
    # flatten strategies dictionary
    strategy_set = []
    strategy_repeat_count = []
    for key in strategies_per_length:
        for count, pack in strategies_per_length[key]:
            pack.reverse()
            strategy_set.append(pack)
            strategy_repeat_count.append(count)
    return strategy_set, np.array(strategy_repeat_count)
```

Listing 5: Evaluation function of shortest-pack-first histogram-packing

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Max depth analysis of shortest-pack-first histogram-packing."""
from collections import defaultdict
import tabulate
import time
import numpy as np

def evaluate_spfhp(histogram, max_sequence_length):
    """Evaluate shortest-pack-first histogram-packing algorithm."""
    stats_data = [["pack. depth", "# strat. used", "# packs", "# tokens",
                   "# padding tok.", "efficiency (%)", "pack.factor", "time"]]
    for max_sequences_per_pack in [1, 2, 3, 4, 8, 16, "max"]:
        start = time.time()
        strategy_set, strategy_repeat_count = pack_using_spfhp(
            histogram, max_sequence_length, max_sequences_per_pack)
        duration = time.time() - start

        # Performance Evaluation of packing approach
        n_strategies = int(len(strategy_set))
        packs = int(sum(strategy_repeat_count))
        sequences = sum([count*len(pack) for count, pack in
                         zip(strategy_repeat_count, strategy_set)])
        total_tokens = int(max_sequence_length * packs)
        empty_tokens = int(sum([
            count*(max_sequence_length-sum(pack)) for count, pack in
            zip(strategy_repeat_count, strategy_set)]))
        token_efficiency = 100 - empty_tokens / total_tokens * 100
        if max_sequences_per_pack == "max":
            m_length = max([len(pack) for pack in strategy_set])
            max_sequences_per_pack = "max ({})".format(m_length)
        stats_data.append([
            max_sequences_per_pack, n_strategies, packs, total_tokens,
            empty_tokens, token_efficiency, sequences / packs, duration])
    print(tabulate.tabulate(stats_data, headers="firstrow", floatfmt=".3f"))
```

Listing 6: Wikipedia and SQUaD 1.1 histograms

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Wikipedia and SQUaD 1.1 histograms."""
import numpy as np
wikipedia_histogram = np.array([
    0, 0, 0, 0, 1821, 1226, 1969, 1315, 1794, 1953, 3082, 3446, 4166, 5062,
    9554, 16475, 19173, 17589, 17957, 19060, 21555, 23524, 26954, 30661, 33470, 36614, 40134, 43256,
    46094, 49350, 52153, 55428, 58109, 60624, 63263, 64527, 65421, 66983, 68123, 68830, 70230, 70486,
    72467, 72954, 73955, 74311, 74836, 74489, 74990, 75377, 74954, 75096, 74784, 74698, 74337, 74638,
    74370, 73537, 73597, 73153, 72358, 71580, 71082, 70085, 69733, 69445, 67818, 67177, 66641, 65709,
    64698, 63841, 63218, 62799, 61458, 60848, 60148, 59858, 58809, 58023, 56920, 55999, 55245, 55051,
    53979, 53689, 52819, 52162, 51752, 51172, 50469, 49907, 49201, 49060, 47948, 47724, 46990, 46544,
    46011, 45269, 44792, 44332, 43878, 43984, 42968, 42365, 42391, 42219, 41668, 41072, 40616, 40587,
    39999, 40169, 39340, 38906, 38438, 38142, 37757, 37818, 37535, 37217, 36757, 36589, 36151, 35953,
    35531, 35496, 35089, 35053, 34567, 34789, 34009, 33952, 33753, 33656, 33227, 32954, 32686, 32880,
    32709, 31886, 32126, 31657, 31466, 31142, 31106, 30650, 30316, 30494, 30328, 30157, 29611, 29754,
    29445, 28921, 29271, 29078, 28934, 28764, 28445, 28319, 28141, 28282, 27779, 27522, 27333, 27470,
    27289, 27102, 27018, 27066, 26925, 26384, 26188, 26385, 26392, 26082, 26062, 25660, 25682, 25547,
    25425, 25072, 25079, 25346, 24659, 24702, 24862, 24479, 24288, 24127, 24268, 24097, 23798, 23878,
    23893, 23817, 23398, 23382, 23280, 22993, 23018, 23242, 22987, 22894, 22470, 22612, 22452, 21996,
    21843, 22094, 21916, 21756, 21955, 21444, 21436, 21484, 21528, 21597, 21301, 21197, 21281, 21066,
    20933, 21023, 20888, 20575, 20574, 20511, 20419, 20312, 20174, 20023, 20087, 19955, 19946, 19846,
    19562, 19710, 19556, 19477, 19487, 19387, 19225, 19069, 19360, 18655, 19034, 18763, 18800, 19012,
    18893, 18714, 18645, 18577, 18317, 18458, 18374, 18152, 17822, 18102, 17735, 17940, 17805, 17711,
    17690, 17703, 17669, 17410, 17583, 17331, 17313, 16892, 16967, 16870, 16926, 17233, 16845, 16861,
    16576, 16685, 16455, 16687, 16747, 16524, 16473, 16349, 16273, 16255, 16228, 16219, 16021, 16111,
    15867, 15751, 16081, 15703, 15751, 15854, 15665, 15469, 15431, 15428, 15464, 15517, 15335, 15461,
    15237, 15292, 15305, 15351, 15078, 14810, 15119, 14780, 14664, 14869, 14722, 14890, 14672, 14439,
    14685, 14706, 14840, 14373, 14286, 14596, 14615, 14168, 14299, 13987, 14167, 14107, 14096, 14202,
    13985, 14118, 14094, 14127, 13896, 13864, 13597, 13572, 13717, 13669, 13782, 13617, 13284, 13333,
    13425, 13457, 13256, 13404, 13318, 13425, 13317, 13179, 13193, 13257, 13160, 12813, 13149, 13010,
    12867, 12958, 12818, 12801, 12749, 12810, 12575, 12673, 12514, 12735, 12523, 12677, 12298, 12469,
    12341, 12445, 12477, 12326, 12110, 12087, 12305, 12156, 12032, 12190, 12150, 11980, 12022, 11825,
    11969, 11831, 11997, 11924, 11739, 11685, 11702, 11783, 11783, 11659, 11647, 11610, 11526, 11577,
    11538, 11536, 11497, 11480, 11374, 11234, 11433, 11466, 11475, 11147, 11376, 11217, 11002, 11245,
    11124, 11000, 11129, 10923, 10966, 11071, 11029, 11036, 10972, 11012, 10800, 10936, 10904, 10750,
    10669, 10766, 10780, 10675, 10905, 10511, 10598, 10583, 10658, 10471, 10667, 10601, 10430, 10440,
    10510, 10148, 10468, 10346, 10257, 10286, 10235, 10351, 10182, 10182, 10095, 10192, 9866, 10070,
    10148, 9956, 10132, 10043, 9741, 10003, 10056, 9920, 10021, 9838, 9854, 9740, 9782, 9799,
    9798, 9788, 9840, 9747, 9797, 9893, 9593, 9535, 9658, 9554, 9593, 9530, 9523, 9488,
    9548, 9418, 9418, 9508, 9638, 9521, 9277, 9289, 9255, 9322, 9281, 9351, 9259, 9255,
    9225, 9098, 9268, 9227, 9224, 9106, 9239, 3815044], dtype=np.int64)

wikipedia_max_sequence_length = 512

squad_1.1_histogram = np.array([
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 3, 2, 0, 9, 10, 16, 22, 24, 36, 35, 46, 42, 48, 57, 86, 83, 86, 87, 86, 97, 90, 99, 85, 94,
    105, 114, 110, 93, 116, 118, 114, 116, 117, 127, 115, 155, 137, 145, 157, 151, 153, 149, 163, 157,
    134, 150, 144, 132, 166, 162, 177, 160, 149, 151, 138, 156, 148, 176, 163, 182, 188, 182, 177, 199,
    182, 203, 201, 264, 250, 244, 289, 346, 327, 298, 377, 386, 444, 431, 503, 553, 532, 570, 611, 677,
    648, 673, 712, 722, 745, 692, 697, 747, 754, 741, 777, 781, 825, 813, 836, 777, 776, 756, 789, 790,
    765, 753, 729, 748, 772, 766, 760, 741, 725, 729, 759, 732, 730, 730, 741, 705, 708, 725, 656, 688,
    688, 677, 662, 628, 635, 618, 586, 527, 562, 619, 562, 578, 538, 558, 582, 541, 575, 526, 556, 498,
    529, 486, 528, 541, 482, 521, 483, 466, 514, 459, 447, 436, 383, 401, 408, 381, 369, 364, 381, 420,
    391, 388, 358, 365, 357, 358, 355, 297, 290, 267, 308, 329, 304, 332, 289, 282, 304, 242, 263, 288,
    238, 257, 271, 288, 277, 264, 253, 239, 217, 260, 214, 247, 237, 212, 205, 193, 200, 208, 195, 193,
    201, 187, 170, 176, 195, 156, 201, 179, 159, 183, 169, 178, 163, 153, 171, 144, 138, 181, 165, 171,
    161, 159, 166, 142, 138, 151, 155, 134, 141, 132, 123, 119, 109, 125, 123, 131, 135, 115, 108, 102,
    117, 105, 99, 84, 100, 85, 85, 85, 95, 122, 105, 114, 113, 100, 80, 96, 86, 79, 80, 87, 92, 73, 73,
    64, 76, 72, 77, 67, 60, 71, 77, 79, 72, 55, 67, 42, 59, 65, 72, 49, 43, 62, 48, 50, 54, 45, 42, 53,
    56, 45, 43, 32, 30, 36, 42, 37, 45, 28, 41, 31, 44, 35, 36, 47, 47, 48, 65, 32, 23, 35, 38, 20, 23,
    22, 21, 27, 20, 26, 18, 18, 22, 17, 17, 14, 26, 15, 20, 22, 19, 24, 17, 15, 20, 20, 22, 22, 17, 20,
    16, 21, 16, 23, 12, 14, 1054], dtype=np.int64)

squad_1.1_max_sequence_length = 384
```

Listing 7: Histogram creation for GLUE training datasets

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
# Copyright 2020 The HuggingFace Inc. team. All rights reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
"""GLUE data loading and histogram creation.

Some code snippets were taken from
https://github.com/huggingface/transformers/blob/master/examples/text-classification/run_glue.py
Most is original code.
"""
from transformers import AutoTokenizer
import datasets
import numpy as np

# constants
max_sequence_length = 128
task_to_keys = {
    "cola": ("sentence", None),
    "mnli": ("premise", "hypothesis"),
    "mrpc": ("sentence1", "sentence2"),
    "qnli": ("question", "sentence"),
    "qqp": ("question1", "question2"),
    "rte": ("sentence1", "sentence2"),
    "sst2": ("sentence", None),
    "stsb": ("sentence1", "sentence2"),
    "wnli": ("sentence1", "sentence2"),
}
glue_keys = ['cola', 'sst2', 'mrpc', 'qqp', 'stsb', 'mnli', 'rte', 'wnli']
# unused datasets due to missing training data
unglue_keys = ['mnli_matched', 'mnli_mismatched', 'qnli', 'ax']

# load data
dataset_loads = {}
for key in glue_keys:
    dataset_loads[key] = datasets.load_dataset("glue", key, split='train')

# tokenize data
tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
tokenized_data = {}
for key in dataset_loads:
    sentence1_key, sentence2_key = task_to_keys[key]

    def preprocess_function(examples):
        """Tokenize the texts"""
        args = (
            (examples[sentence1_key],) if sentence2_key is None
            else (examples[sentence1_key], examples[sentence2_key])
        )
        result = tokenizer(*args, padding=False, max_length=max_sequence_length, truncation=True)
        return result

    tokenized_data[key] = dataset_loads[key].map(preprocess_function, batched=True)

# extract length information (for histogram plots)
histogram_length = {}
for key in tokenized_data:
    histogram_length[key] = []
for number, key in enumerate(tokenized_data.keys()):
    for raw_record in tokenized_data[key]["input_ids"]:
        histogram_length[key].append(len([x for x in raw_record if x!=0]))

# create histogram for packing
glue_histogram = {}
for data_key in histogram_length:
    glue_histogram[data_key] = np.array([0] * max_sequence_length, dtype=np.int64)
    for entry in histogram_length[data_key]:
        glue_histogram[data_key][entry-1] += 1
```

Listing 8: Longest-pack-first histogram-packing

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Longest-pack-first histogram-packing."""
from collections import defaultdict
import numpy as np

def add_pack(pack, count, tmp, final, limit, offset, max_sequence_length=512):
    """Filter out packs that reached maximum length or number of components."""
    # sanity checks
    assert(max_sequence_length-sum(pack) == offset), "Incorrect offset."
    assert(offset >= 0), "Too small offset."
    assert(offset < max_sequence_length), "Too large offset."
    if len(pack) == limit or offset == 0:
        final[offset].append((count, pack))
    else:
        tmp[offset].append((count, pack))

def pack_using_lpfhp(histogram, max_sequence_length, max_sequences_per_pack, distribute=True):
    """Longest-pack-first histogram-packing algorithm."""
    reversed_histogram = np.flip(histogram)
    # Initialize main strategy data dictionary.
    # The key indicates how many tokens are left for full length.
    # The value is a list of tuples, consisting of counts and respective packs.
    # A pack is a (sorted) list of sequence length values that get concatenated.
    tmp_strategies_per_length = defaultdict(list)
    strategies_per_length = defaultdict(list)
    if max_sequences_per_pack is "max":
        max_sequences_per_pack = max_sequence_length
    # Index i indicates here, how much space is left, due to reversed histogram
    for i in range(max_sequence_length):
            if (length_to_bin + offset) in tmp_strategies_per_length:
                # extract worst pack that will get modified
                n_sequences_to_pack, pack = tmp_strategies_per_length[
                    length_to_bin + offset].pop()
                # calculate how often the current sequence maximally fits in
                repeat = min(1 + offset // length_to_bin, max_sequences_per_pack-len(pack))
                # correct dependent on count
                while n_sequences_to_bin//repeat == 0:
                    repeat -= 1
                if not distribute:
                    repeat = 1
                new_pack = pack + [length_to_bin]*repeat
                count = min(n_sequences_to_pack, n_sequences_to_bin//repeat)
                if n_sequences_to_pack > count:
                    # old pack gets reduced
                    n_sequences_to_pack -= count
                    tmp_strategies_per_length[length_to_bin + offset].append(
                        (n_sequences_to_pack, pack))
                    n_sequences_to_bin -= count * repeat
                else:
                    n_sequences_to_bin -= n_sequences_to_pack * repeat
                add_pack(new_pack, count,
                         tmp_strategies_per_length, strategies_per_length,
                         max_sequences_per_pack, offset -(repeat-1)*length_to_bin,
                         max_sequence_length)
                # clean up to speed up main key search
                if not tmp_strategies_per_length[length_to_bin + offset]:
                    tmp_strategies_per_length.pop(length_to_bin + offset)
                # reset offset in case best fit changed
                offset = 0
            else:
                offset += 1
            # Does not fit anywhere. Create new pack.
            if offset >= max_sequence_length - length_to_bin + 1:
                # similar repetition but no dependence on pack.
                repeat = min(max_sequence_length//length_to_bin, max_sequences_per_pack)
                while n_sequences_to_bin//repeat == 0:
                    repeat -= 1
                if not distribute:
                    repeat = 1
                add_pack([length_to_bin]*repeat, n_sequences_to_bin//repeat,
                         tmp_strategies_per_length, strategies_per_length,
                         max_sequences_per_pack, max_sequence_length-length_to_bin*repeat,
                         max_sequence_length)
                n_sequences_to_bin -= n_sequences_to_bin//repeat * repeat
    # merge all strategies
    for key in tmp_strategies_per_length:
        strategies_per_length[key].extend(tmp_strategies_per_length[key])
    # flatten strategies dictionary
    strategy_set = []
    strategy_repeat_count = []
    for key in strategies_per_length:
        for count, pack in strategies_per_length[key]:
            pack.reverse()
            strategy_set.append(pack)
            strategy_repeat_count.append(count)
    return strategy_set, np.array(strategy_repeat_count)
```

Listing 9: Extended non-negative least squares histogram-packing

```python
# Copyright (c) 2021 Graphcore Ltd. All rights reserved.
"""Extended Non-Negative least squares histogram-packing algorithm."""
import time
import numpy as np
from scipy import optimize, stats
from functools import lru_cache

def get_packing_matrix(strategy_set, max_sequence_length):
    num_strategies = len(strategy_set)
    A = np.zeros((max_sequence_length, num_strategies), dtype=np.int32)
    for i, strategy in enumerate(strategy_set):
        for seq_len in strategy:
            A[seq_len - 1, i] += 1
    return A

@lru_cache(maxsize=None)
def get_packing_strategies(start_length, minimum_increment, target_length, depth):
    gap = target_length - start_length
    strategies = []
    # Complete the packing with exactly 1 number
    if depth == 1:
        if gap >= minimum_increment:
            strategies.append([gap])
    # Complete the sample in "depth" steps, recursively
    else:
        for new in range(minimum_increment, gap + 1):
            new_gap = target_length - start_length - new
            if new_gap == 0:
                strategies.append([new])
            else:
                options = get_packing_strategies(start_length + new, new, target_length, depth - 1)
                for option in options:
                    if len(option) > 0:
                        strategies.append([new] + option)
    return strategies

def pack_using_ennlshp(histogram, max_sequence_length, max_sequences_per_pack):
    # List all unique ways of packing to the desired maximum sequence length
    strategy_set = get_packing_strategies(0, 1, max_sequence_length, max_sequences_per_pack)
    print(f"Packing will involve {len(strategy_set)} unique packing strategies.")
    # Get the packing matrix corresponding to this list of packing strategies
    A = get_packing_matrix(strategy_set, max_sequence_length)
    # Weights that penalize the residual by the number of resulting padding tokens.
    w0 = np.array([x+1 for x in range(max_sequence_length)])
    # construct the packing matrix
    A_bar = np.zeros((2*max_sequence_length, len(strategy_set) + max_sequence_length), 'd')
    # Base weighted matrix
    A_bar[:max_sequence_length, :len(strategy_set)] = np.expand_dims(w0, -1) * A
    # Higher weight to avoid positive residual
    A_bar[max_sequence_length:, :len(strategy_set)] = np.expand_dims(
        10**6*np.ones([max_sequence_length]), -1) * A
    # negative diagonal unity matrix for mapping to residual
    A_bar[max_sequence_length:, len(strategy_set):] = np.expand_dims(
        10**6*np.ones([max_sequence_length]), -1)*np.ones((max_sequence_length,max_sequence_length))
    b_bar = np.zeros(2*max_sequence_length)
    # Apply weighting to histogram vector
    b_bar[:max_sequence_length] = w0 * histogram
    b_bar[max_sequence_length:] = 10**6*np.ones([max_sequence_length]) * histogram
    # Solve the packing problem
    print(f"Sequences to pack: ", histogram.sum())
    start = time.time()
    strategy_residual, rnorm = optimize.nnls(A_bar, b_bar)
    strategy_repeat_count = strategy_residual[:len(strategy_set)]
    print(f"Solving non-negative least squares took {time.time() - start:3.2f} seconds.")
    # Round the floating point solution to nearest integer
    strategy_repeat_count = np.rint(strategy_repeat_count).astype(np.int64)
    # Compute the residuals, shape: [max_sequence_length]
    residual = histogram - A @ strategy_repeat_count
    # Handle the left-over sequences i.e. positive part of residual
    unpacked_seqlen = np.arange(1, max_sequence_length + 1)[residual > 0]
    for l in unpacked_seqlen:
        strategy = sorted([l, max_sequence_length - l])  # the depth 1 strategy
        strategy_index = strategy_set.index(strategy)
        strategy_repeat_count[strategy_index] += residual[l-1]
    # Re-compute the residual with the updated strategy_repeat_count
    # This should now be strictly < 0
    residual = histogram - A @ strategy_repeat_count
    # Add padding based on deficit (negative residual portion of residual)
    padding = np.where(residual < 0, -residual, 0)
    # Calculate some basic statistics
    sequence_lengths = np.arange(1, max_sequence_length + 1)
    old_number_of_samples = histogram.sum()
    new_number_of_samples = int(strategy_repeat_count.sum())
    speedup_upper_bound = 1.0/(1 - (histogram*(1 - sequence_lengths / max_sequence_length)).sum()/
     old_number_of_samples)
    num_padding_tokens_packed = (sequence_lengths * padding).sum()
    efficiency = 1 - num_padding_tokens_packed/(new_number_of_samples*max_sequence_length)
    print(f"Packing efficiency (fraction of real tokens): {efficiency:3.4f}\n",
          f"Speed-up theoretical limit: {speedup_upper_bound:3.4f}\n",
          f"Achieved speed-up over un-packed dataset: {old_number_of_samples/new_number_of_samples:3.5f
     }")
    return strategy_set, strategy_repeat_count
```

## Appendix References


[27] BELOV, G., AND SCHEITHAUER, G. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research 171*, 1 (may 2006), 85–106.

[28] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[29] KOTZ, S., AND NADARAJAH, S. *Extreme Value Distributions*. World Scientific Publishing Company, 2000.

[30] LAWSON, C. L., AND HANSON, R. J. *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics, jan 1995.

[31] LEE, C. C., AND LEE, D. T. A Simple On-Line Bin-Packing Algorithm. *Journal of the ACM (JACM) 32*, 3 (jul 1985), 562–572.

[32] LUO, Y., AND DURAISWAMI, R. Efficient parallel non-negative least squares on multi-core architectures. *SIAM Journal on Scientific Computing 33* (2011), 2848 – 2863.

[33] NVIDIA. Performance catalogue for BERT on Pytorch. `https://ngc.nvidia.com/catalog/resources/nvidia:bert_for_pytorch/performance`, 2021.

[34] VIRTANEN, P., GOMMERS, R., OLIPHANT, T. E., HABERLAND, M., REDDY, T., COURNAPEAU, D., BUROVSKI, E., PETERSON, P., WECKESSER, W., BRIGHT, J., VAN DER WALT, S. J., BRETT, M., WILSON, J., MILLMAN, K. J., MAYOROV, N., NELSON, A. R. J., JONES, E., KERN, R., LARSON, E., CAREY, C. J., POLAT, İ., FENG, Y., MOORE, E. W., VANDERPLAS, J., LAXALDE, D., PERKTOLD, J., CIMRMAN, R., HENRIKSEN, I., QUINTERO, E. A., HARRIS, C. R., ARCHIBALD, A. M., RIBEIRO, A. H., PEDREGOSA, F., VAN MULBREGT, P., AND SCIPY 1.0 CONTRIBUTORS. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods 17* (2020), 261–272.

[35] WOLF, T., DEBUT, L., SANH, V., CHAUMOND, J., DELANGUE, C., MOI, A., CISTAC, P., RAULT, T., LOUF, R., FUNTOWICZ, M., DAVISON, J., SHLEIFER, S., VON PLATEN, P., MA, C., JERNITE, Y., PLU, J., XU, C., SCAO, T. L., GUGGER, S., DRAME, M., LHOEST, Q., AND RUSH, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Online, Oct. 2020), Association for Computational Linguistics, pp. 38–45.

[36] WOLF, T., LHOEST, Q., VON PLATEN, P., JERNITE, Y., DRAME, M., PLU, J., CHAUMOND, J., DELANGUE, C., MA, C., THAKUR, A., PATIL, S., DAVISON, J., SCAO, T. L., SANH, V., XU, C., PATRY, N., MCMILLAN-MAJOR, A., BRANDEIS, S., GUGGER, S., LAGUNAS, F., DEBUT, L., FUNTOWICZ, M., MOI, A., RUSH, S., SCHMIDD, P., CISTAC, P., MUŠTAR, V., BOUDIER, J., AND TORDJMANN, A. Datasets. *GitHub. Note: https://github.com/huggingface/datasets 1* (2020).

[37] WOLFRAM RESEARCH INC. Mathematica, Version 12.2. Champaign, IL, 2020.