Adaptive Data Batching

Introduction

Deep learning models are trained on large datasets, which can often include millions of instances, ranging from images to text, audio, or sensor signals. Training a model using the entire dataset at once is computationally expensive and inefficient, especially with modern large-scale datasets. As a solution, the concept of **data batching** has become central to deep learning workflows. This process involves dividing the dataset into smaller subsets called **batches**, which are processed individually during model training.

A **batch** is a small subset of the full dataset that is passed through the model during a single forward and backward pass. It is one unit of work done by the neural network during training or inference.

Important terms:

- **Batch size**: Number of instances in a single batch.
- **Mini-batch**: A commonly used term for a batch that contains more than one sample but less than the full dataset.
- **Epoch**: One complete pass through the entire training dataset.
- **Iteration**: One update of the model parameters, typically based on a single batch.

Training on batches, rather than the entire dataset at once (full-batch), offers several advantages:

- **Memory efficiency:** Modern deep learning models are typically trained on GPUs, which have limited memory. Loading the full dataset into memory is impractical. Batching allows the training process to operate within hardware limitations.
- **Computational efficiency:** Using batches allows for vectorized computation. Processing a batch is significantly faster than processing individual instances sequentially due to parallelism and GPU acceleration.
- **Generalization:** Mini-batch training introduces stochasticity into the optimization process. Unlike full-batch training (which has deterministic gradients), mini-batch gradients have noise, which can help escape local minima and improve generalization.
- **Stability and convergence:** Stochastic Gradient Descent (SGD) and its variants are typically used in mini-batch mode. The choice of batch size affects the stability and convergence of the training process. Small batches may cause noisy updates, while large batches may converge more slowly or cause overfitting.

Batching strategies

There are multiple strategies for batching depending on the dataset and task:

- **Fixed-size batching:** Each batch contains the same number of instances. This is the most common strategy and simplifies computation.
- **Dynamic batching:** Used in NLP and variable-length input scenarios. Sequences are grouped by similar lengths to minimize padding and maximize efficiency.
- **Bucketed Batching:** Data is grouped into "buckets" by characteristics (e.g., sequence length), and batches are drawn from these buckets to minimize padding and maintain uniformity.
- **Adaptive batching:** Batch size is dynamically adjusted during training based on model convergence, memory usage, or training progress.

Batch size Selection

The choice of **batch size** is an important hyperparameter that influences performance and training dynamics.

- Small batches (1-32 instances):
    - Noisier gradient estimates help generalization.
    - Less efficient on hardware, slow convergence.
- Medium batches (64-512 instances):
    - Often a good balance between generalization and efficiency.
    - Default choice in many practical scenarios.
- Large batches (1024+ instances):
    - Efficient on GPUs, stable convergence.
    - May require learning rate adjustments, risk of overfitting, reduced generalization.

Adaptive Batching

Adaptive batching refers to the dynamic adjustment of batch sizes during training or in deep learning. Unlike fixed batching, where a pre-defined and static batch size is used throughout the training process, adaptive batching involves strategies that adjust the batch size according to various runtime conditions, dataset properties, model behavior, or resource constraints.
Adaptive batching is especially valuable in scenarios involving:

- Hardware constraints: batch size is limited by memory availability (especially GPU memory). If memory consumption spikes (for example, when long sequences are processed), fixed batching may cause crashes or underutilization. Adaptive strategies help maintain stability and efficiency.
- Variable-length inputs: In NLP or speech tasks, input sequences often have different lengths. Padding them to the longest sequence in a fixed-size batch wastes memory. Adaptive batching groups sequences to maximize utilization with minimal padding.
- Dynamic computational load: Some instances or tasks take longer to process (longer sentences, larger images). Adjusting batch sizes in real time can ensure that hardware is neither overloaded nor idle.
- Optimization dynamics: Adaptive batch sizing can be used to guide the training process. For example, smaller batches in early epochs can introduce useful stochasticity, while larger batches in later epochs can stabilize convergence.

Categories of Adaptive Batching

- Sequence-Length-Based Adaptive Batching: often used in NLP or speech processing, this technique groups sequences by their length, using variable batch sizes to fit within a the available memory.
    - Use a bucketing mechanism to sort or group sequences by length.
    - Use dynamic padding within each batch.
    - Dynamically adjust batch size to stay within token or memory size.
- Memory-based adaptive batching: batch size is adjusted in real-time based on available memory or profiling statistics
    - Monitor memory usage using frameworks like PyTorch's `torch.cuda.max_memory_allocated()`.
    - Dynamically reduce batch size if memory exceeds a threshold.

    - Increase batch size when memory is underutilized.
- Time-constrained adaptive batching: Used in latency-sensitive applications (for example, online translation), where batch size is adjusted to meet real-time constraints.
    - Collect requests for a fixed time window (ex., 10 ms).
    - Aggregate as many requests as can be processed in that window.
    - Submit the batch for processing.
- Adaptive batching by training phase: the batch size changes as training progresses
    - Use small batches during early epochs to benefit from noise in gradients.
    - Gradually increase batch size to stabilize convergence.
    - Known as **progressive batching** or **batch size warm-up**.

Batch Adaptation During Training

Research and practice have shown that dynamically adjusting the batch size during training can provide benefits in terms of optimization dynamics, training speed, and model generalization. This technique is referred to as **adaptive batching during training** or **progressive batching**. Unlike adaptive batching that reacts to input shape or memory constraints, this form adapts batch size as training progresses, often in tandem with learning rate values or model convergence behavior.

In early training, models benefit from noisy gradients (small batch sizes), which help escape suboptimal local minima. As training proceeds, and the model approaches a minimum, larger batch sizes produce more stable gradients for fine-tuning.

Smaller batches reduce computational requirements per update and introduce more frequent weight updates in the early stages. Later, larger batches benefit from GPU efficiency and allow for more stable convergence.

Small batches may allow convergence to flatter minima**,** which are associated with better generalization, while large batches may converge to sharper minima that generalize poorly. Adaptive strategies attempt to harness the best of both worlds.

There are multiple ways to implement adaptive batch sizing during training:

- Epoch-based adaptive batching (aka progressive batching): Gradually increase the batch size after a certain number of epochs or training milestones.
- Loss-based adaptive batching: Increase the batch size when the training loss converges or falls below a certain threshold.
- Gradient Noise Scale (GNS) – based adaptive batching: use gradient noise information to guide batch size.
- Learning rate coupled adaptive batching: vary batch size with the learning rate. For example, as the batch size increases, so does the learning rate.
- Adaptive batching with curriculum learning: introduce simpler examples first and harder ones later. Start with smaller batches of instances which are easier to process and gradually include more difficult instances while increasing batch size.

Tasks

Inplement various adaptive batch strategies during training. In each case, establish and maintain a minimum/maximum batch size.
Use the various strategies with multiple data sets and neural network models. Note whether there is an improvement in convergence behavior / loss minimization and which batch adjustment methods provides the best result depending on the dimensionality, type and properties of the data sets. Write down your results/observations.

1. **Linear adjustment:** gradually update batch size over epochs, using a linear dependency:

   $B_i = B_{init} + ai$

   where:

   B_i is the batch size at epoch i

   B_{init} is the initial batch size

   a is an adjustable coefficient

2. **Exponential adjustment:** increase batch size over epochs, exponentially:

   $B_i = B_{init} a^i$

3. **Cyclical batch size:** vary batch size in cycles – start with small batches, increase for several epochs, then restart with the initial small batches.

4. **Loss-based adjustment:** adjust batch size based on the loss:
    - If loss decreases, increase batch size (attempting to improve stability)
    - If loss increases or fluctuates, decrease batch size (update parameters for frequently)

   The increase/decrease can be linear or non-linear (exponential) with the variation of the loss across epochs or batches.

   The loss variation should be computed in the following ways:

- Simple before-after difference

$$\Delta_{loss} = loss_{currrent} - loss_{previous}$$

- Using a moving average ($loss_{ma}$), to smooth the loss values:

$$loss_{ma} = \alpha loss_{current} + (1-\alpha) loss_{ma\_previous}$$
$$\Delta_{loss} = loss_{ma} - loss_{ma\_previous}$$

  where $\alpha \in (0, 1)$ is an adjustable parameter

- Determining loss variance over *K* epochs: instead of computing two values, we measure the variance over a *K*-epoch window:

  Assuming the window contains epochs *i* … *i*+*K*:

$$\text{var}(loss) = \sum_{k=i}^{i+K} (loss_k - loss_m)^2$$

  Where $loss_m$ is the mean loss of $loss_{i+k}$, *k = 1..K*

5. **Confidence-based – change batch size based on model output entropy**
    - Increase batch size when entropy is low (high confidence)
    - Decrease batch size when entropy is high (uncertain predictions)

- Entropy is computed based on the softmax outputs of the model (obtained by applying the softmax function to the model output)

$$H(p) = -\sum_{i=1}^{C} p_i \log(p_i)$$

where:

- C is the number of classes
- p_i is the predicted probability for class i

  The entropy is:
  - zero, when the model is 100% confident (it predicts a class with maximum certainty)
  - log(C), when the model is maximally uncertain (the same probability for each class)

6. **Gradient noise scale (GNS):** compute the ratio of gradient variance to squared gradient norm (the noise scale) and increase batch size when gradient noise is high.

GNS is defined as:

$$GNS = \frac{tr(\Sigma)}{\|G\|^2}$$

Where:

Σ = covariance matrix of the gradient

$tr(\Sigma)$ = trace of the covariance matrix (sum of main diagonal elements), signifying total gradient variance

$||G||^2$ = squared gradient norm

How to compute in practice:

- For a given batch:
  - split the batch into k microbatches
  - compute gradient for each microbatch: $g_1, g_2, ..., g_k$
  - compute mean gradient $g_m = 1/k * sum(g_i)$
  - compute gradient variance $var(g) = 1/k * sum((g_i - g_m)^2)$
  - estimate $GNS = var(g) / ||g_m||^2$

Hint: for a batch or microbatch, a flattened vector with all gradients can be obtained as follows:

```
grad_vec = torch.cat([param.grad.view(-1)
                      for param in model.parameters()
                      if param.grad is not None])
```