Adaptive Pooling Padding

David Gormley, Veer Bathwal

Review: Why do we have to pad?

- PyTorch data loaders feed images to our network in batches. This allows us to benefit from parallel processing (based on num_workers input)
- Create groups of size = batch_size, that are all fed into the model at once, before each new backpropagation update
- Images in each batch have to be the same size, however different batches can have different sizes
- This is a problem if we're using adaptive pooling instead of resizing all images
- As a result, we have to pad images in each batch up to the width of the largest image in that respective batch

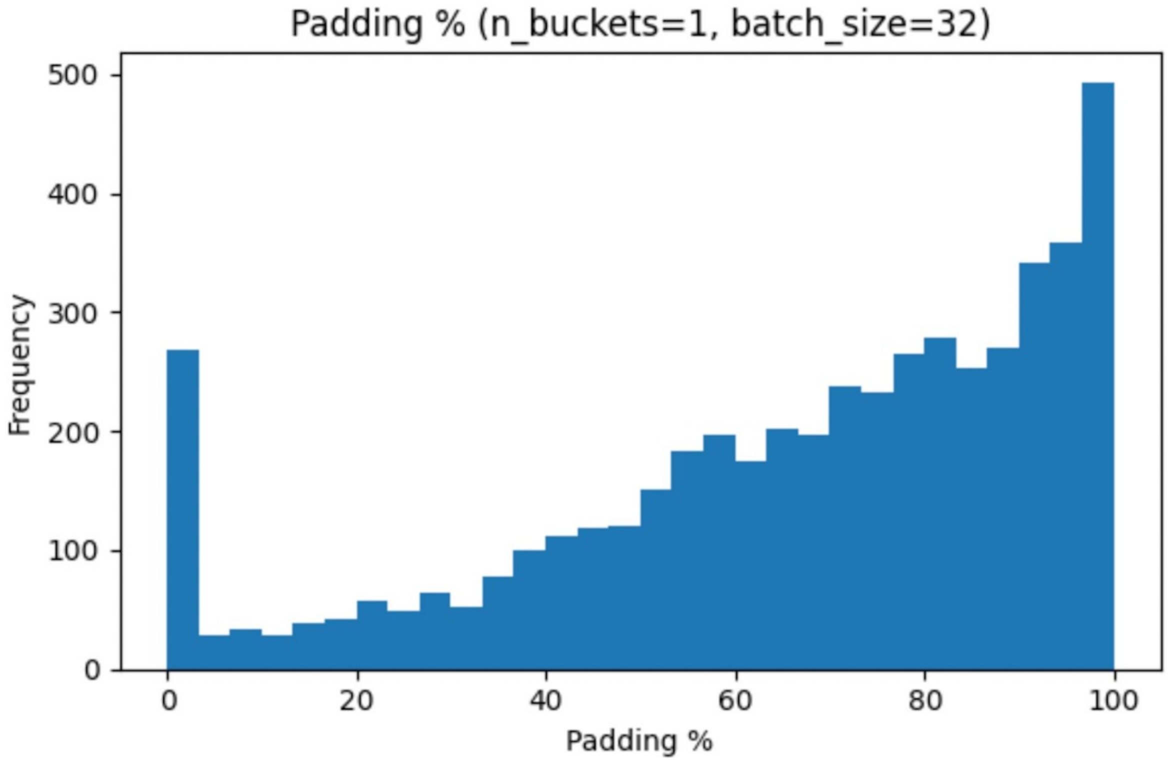
Review: Bucketing logic

- Since we are padding images up to max width within each batch, we want to group together images of similar size (into “buckets”) in order to minimize the amount of padding that is done
- This is what the BucketSampler and make_boundaries_for_buckets functions in the skeleton code do
- So what is the optimal number of buckets in order to minimize padding?

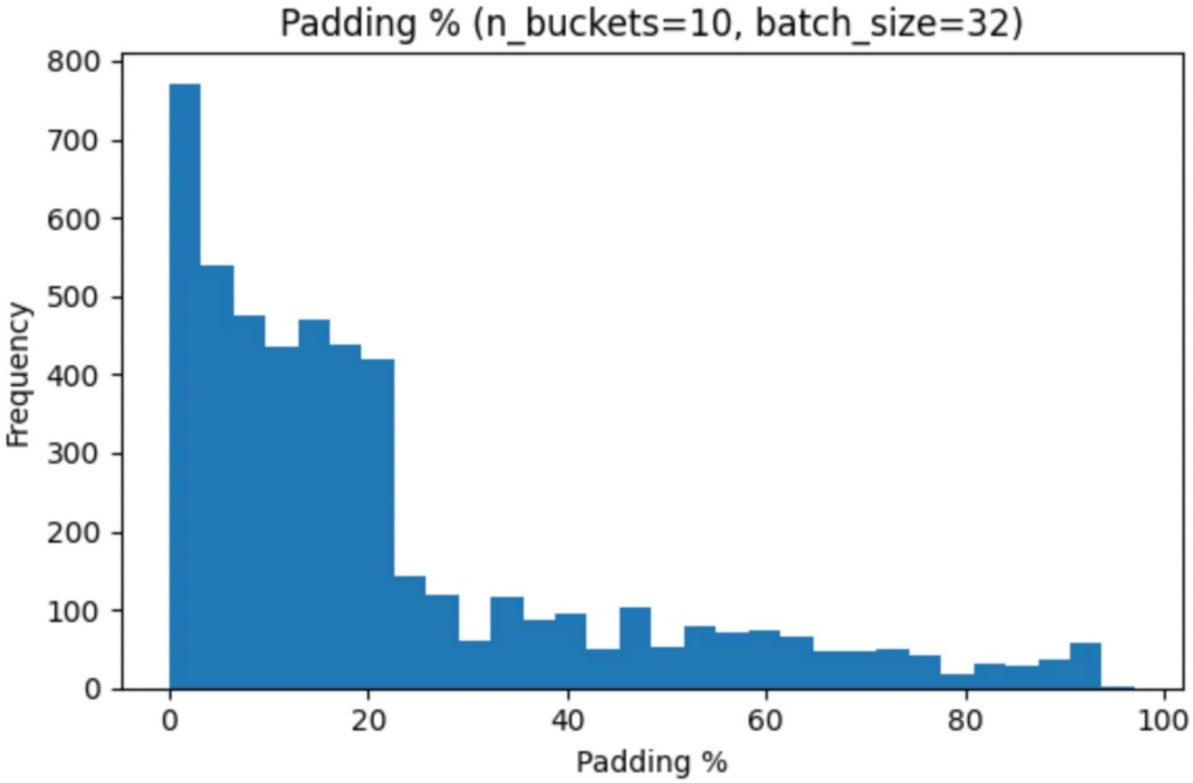
Bucketing vs Batching

- A batch is simply the group of images put together by the data loader, which is ultimately fed into the model
- A bucket is just a term we came up with, it is not defined terminology
- If we divide our images into 2 subgroups based on size, we say we have created 2 buckets
- Each batch is formed from one particular bucket
- Our hypothesis is that plugging number of buckets = number of batches is the optimal solution, since each bucket simply then becomes a batch
- We can calculate number of batches as (num of images/batch_size)
- Since number of images may differ across train, test, and validation sets, we may have to use different number of buckets across our 3 data loaders

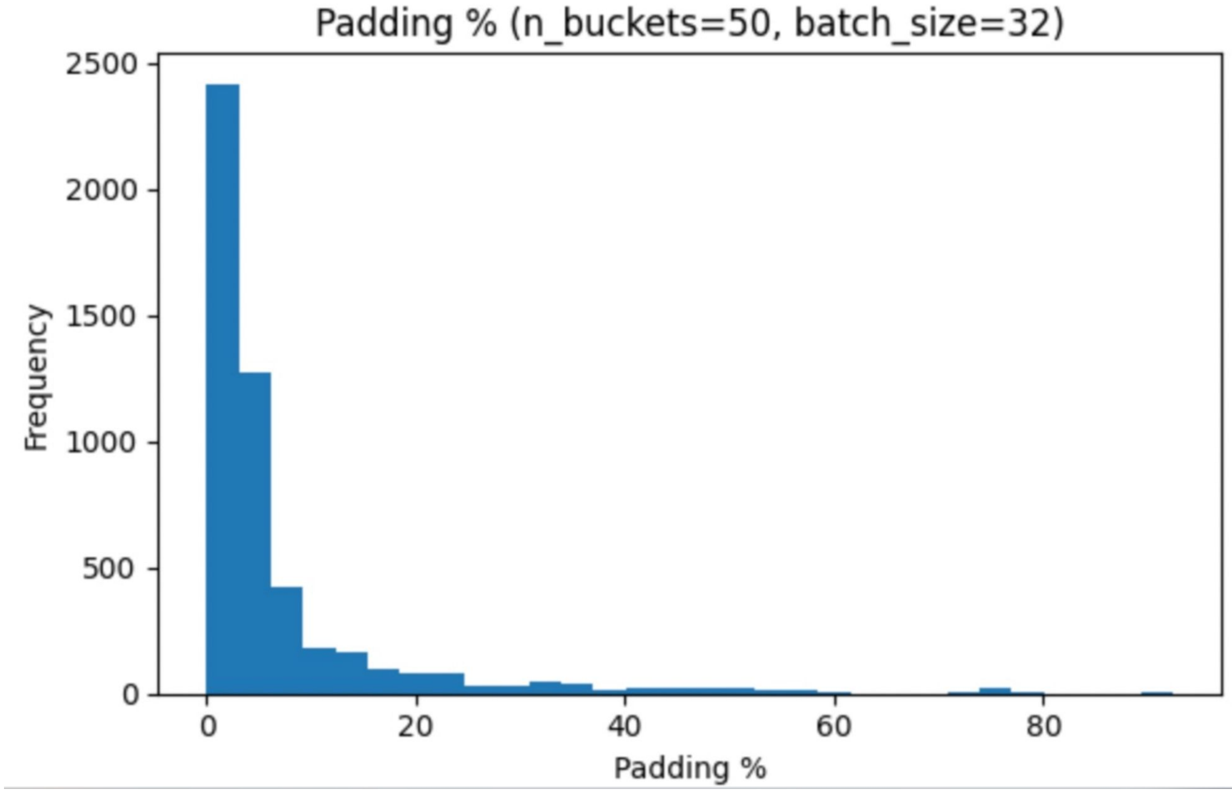
Not doing any grouping results in excessive padding



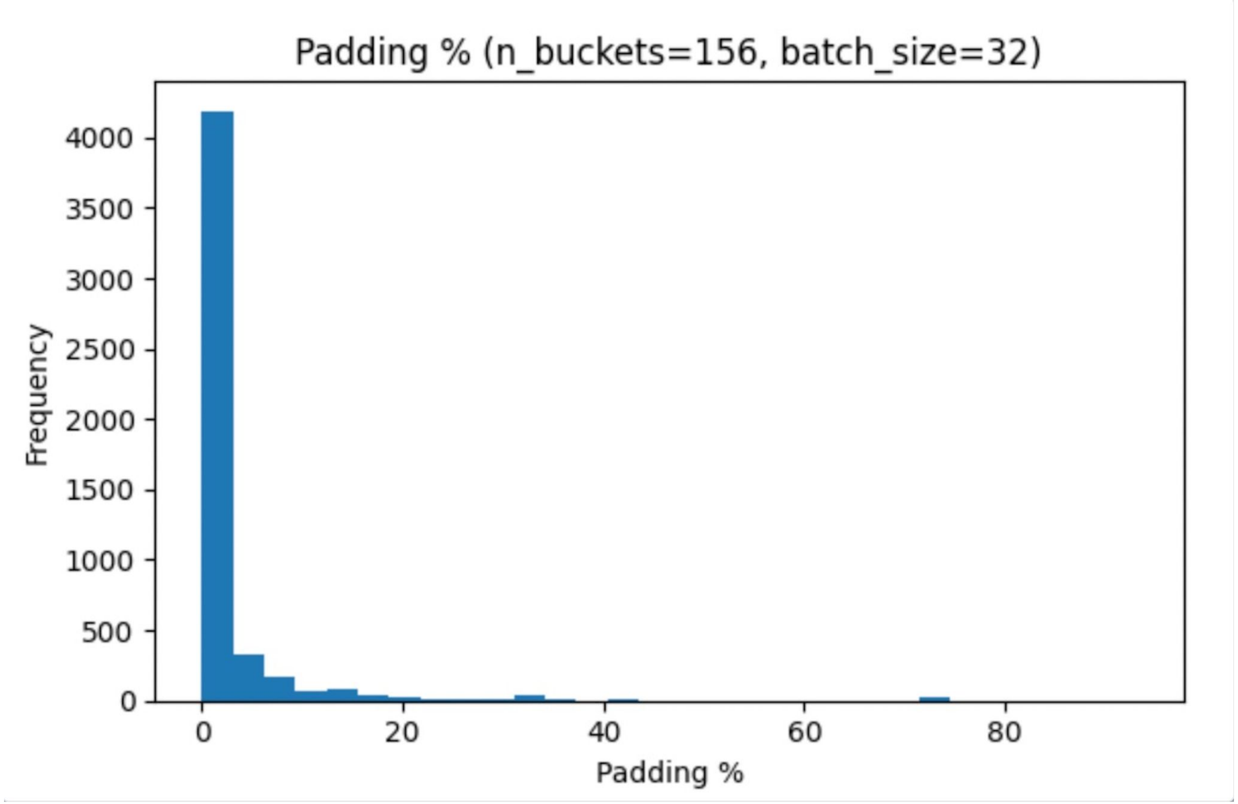
Increasing number of buckets gives better results



Increasing number of buckets gives better results



Increasing number of buckets gives better results



Summary stats across batch size and number of buckets

	n_buckets	batch_size	mean_%	median_%	q1_%	q3_%	variance_%
0	1	16	56.603700	61.154514	35.574375	82.050463	909.939213
1	10	16	19.759552	13.700930	5.332359	24.085874	423.993532
2	50	16	5.945354	3.059103	0.520833	6.007538	100.008579
3	313	16	1.454200	0.454545	0.000000	0.914175	29.173008
4	1	32	66.506714	72.926097	51.478099	89.240405	755.044351
5	10	32	21.898662	15.147929	6.059172	27.968410	490.637508
6	50	32	6.605304	3.287117	0.581395	6.633222	124.150940
7	156	32	2.684186	0.595238	0.000000	1.470588	60.453073
8	1	64	74.694937	82.700893	62.921142	94.266082	639.232260
9	10	64	23.321671	15.703977	6.792059	31.574394	548.225400
10	50	64	7.085192	3.425884	0.641026	7.080316	134.171295
11	78	64	5.250007	0.961538	0.409836	4.719133	124.296560

Impact of Batch Size as a Hyperparam (from perplexity)

Batch Size	Pros	Cons
Small (e.g., 32)	Better generalization, noisier gradients (regularization effect)	Slower training, less efficient hardware use
Large (e.g., 256+)	Faster training, stable gradients, efficient hardware use	May overfit, worse generalization, high memory use

Takeaways

- Lower batch size and higher number of buckets will reduce padding
- However, this comes at the trade-off of higher computational cost. A smaller batch_size means the data loaders have to create more batches, which means a longer runtime for our training loop
- Pros and cons of low vs high batch_size
- Furthermore, num_buckets cannot be larger than number of batches, since each batch is taken from a single bucket
- Our suggestion: keep number of buckets = number of batches and tune the batch_size as a hyperparameter
    - Where number of batches = number of images/batch_size
    - Since number of images differs across train, test, and validation sets, we will have to use a different number of buckets for each
    - This can be setup in the skeleton code's config, as shown in the updated code for this week