Advanced Compression Techniques

"The problem is that we attempt to solve the simplest questions cleverly,
thereby rendering them unusually complex.
One should seek the simple solution."
— Anton Pavlovich Chekhov

In this chapter, we will discuss two advanced compression techniques. By 'advanced' we mean that these techniques are slightly more involved than quantization (as discussed in the second chapter). But that doesn't mean they are harder to learn or implement. We provide a gentle introduction to both of them, with an eye towards conceptual understanding as well as practically using them in your deep learning models.

We start with sparsity. If your goal was to optimize your brain for storage, you can often trim a lot of useless trivia without it impacting your life materially. This is also how your brain naturally works. You can *prune* all this extra information and still drive well, eat, sleep etc. To memorize something for the long term, you need to improve recall by repetition (i.e., increase the weight of that connection). Can we do the same with neural networks? Can we optimally prune the network connections, remove extraneous nodes, etc. while retaining the model's performance? In this chapter we introduce the intuition behind sparsity, different possible methods of picking the connections and nodes to prune, and how to prune a given deep learning model to achieve storage and latency gains with a minimal performance tradeoff.

Next, the chapter goes over weight sharing using clustering. Weight sharing, and in particular clustering is a generalization of quantization. If you noticed, quantization ensures that any two weights that lie within the same quantization *bin*, are mapped to the same quantized weight value. That is an implicit form for weight sharing. However, quantization falls behind in case the data that we are quantizing is not uniformly distributed, i.e. the data is more likely to take values in a certain range than another equally sized range. It creates equal sized quantization ranges (bins), regardless of the frequency of data. Clustering helps solve that problem by adapting the allocation of precision to match the distribution of the data, which ensures the decoded value deviates less from the original value and can help improve the quality of our models.

Did we get you excited yet? Let's learn about these techniques together!

Model Compression Using Sparsity

Sparsity or Pruning refers to the technique of removing (pruning) weights during the model training to achieve smaller models. Such models are called sparse or pruned models. The simplest form of pruning is to zero out a certain, say *p*, percentage of the smallest absolute valued weights in each training epoch. The result of such a training process is *p%* weights with *zero* values. Sparse compressed models achieve higher compression ratio which results in lower transmission and storage costs. Figure 5-1 visually depicts two networks.

The one on the left is the original network and the one on the right is its pruned version. Note that the pruned network has fewer nodes and some retained nodes have fewer connections. Let's do an exercise to convince ourselves that setting parameter values to *zero* indeed results in a higher compression ratio.

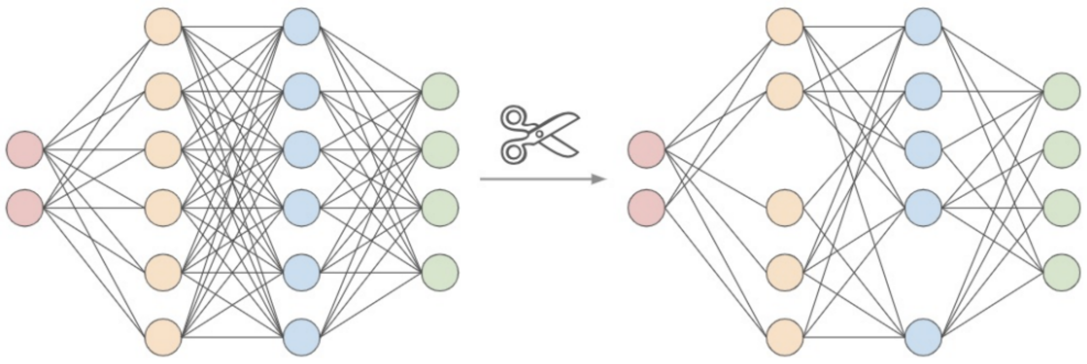


Figure 5-1: An illustration of pruning weights (connections) and neurons (nodes) in a neural network consisting of fully connected layers.

Exercise: Sparsity improves compression

Let's import the required libraries to start with. We will use the *gzip* python module for demonstrating compression. The code for this exercise is available as a Jupyter notebook **here**.

```
%%capture
import gzip
import operator, random
import numpy as np
import tensorflow as tf
from functools import reduce
from matplotlib import pyplot as plt
```

We define two functions *sparsify_smallest()* and *compress()*. The *sparsify_smallest()* sets the absolute smallest weights in the input weight matrix to *zero*. The number of the absolute smallest weights is computed based on the *sparsity_rate* parameter which denotes the fraction of total weights to be removed. The *compress()* function compresses the input array using gzip compression. It returns the compressed bytes.

```
# Sparsify the weights by setting a fraction of the weights to zero.
def sparsify_smallest(w, sparsity_rate):
   w = w.copy()
   w_1d = np.reshape(w, (-1))

   # Create a list of indices sorted by the absolute magnitude of the weights.
```

```
    w_1d_sorted_indices = np.argsort(np.abs(w_1d))

    # Compute the number of elements to zero.
    num_elements_to_zero = int(w_1d.shape[0] * sparsity_rate)

    # Set the respective indices to zero.
    w_1d[w_1d_sorted_indices[:num_elements_to_zero]] = 0.0

    w = np.reshape(w_1d, w.shape)
    return w

def compress(w):
    # Compress the weights matrix using gzip.
    compressed_w = gzip.compress(w.tobytes())
    return compressed_w
```

To demonstrate the effect of sparsity on compression, we create a sample 2D weight matrix with randomly initialized float values. We also define a *sparsity_rate* variable initialized with the value 0.4 to sparsify 40% of the total number of weights. Finally, we compute the original weight matrix size, compressed weight matrix size, and compressed and sparsified weight matrix size. As shown in the output below, the **sparsified compressed matrix is smaller than the regular compressed matrix by nearly 50%**.

```
weights = np.random.normal(size=(100, 100)).astype(np.float32)
sparsity_rate = 0.4 # The percentage of weights that are zeroed out.

sparse_weights = sparsify_smallest(weights, sparsity_rate)

print('Original Size:', reduce(operator.mul, weights.shape)*weights.itemsize)

weights_compressed = compress_and_save(weights)
print('Original Compressed Size:', len(weights_compressed))

weights_sparsified_compressed = compress_and_save(sparse_weights)
print('Sparsified Compressed Size:', len(weights_sparsified_compressed))
```

Output

```
Original Size: 40000
Original Compressed Size: 37103
Sparsified Compressed Size: 24844
```

We hope that you are convinced that sparsity helps with improving compression. Increasing the *sparsity_rate* variable's value will further reduce the size of the sparsified and compressed size.

To take a step back, in the above exercise, we pruned the weights with the smallest absolute values (magnitudes). The general form of this strategy is known as *Magnitude-Based*

Pruning. However, is this a good strategy to choose prunable weights? Are there other strategies that we can try? Let's introduce the concept of saliency scores to abstract the pruning strategies from the pruning process. The saliency scores are the scores assigned to the weights (edges / nodes to be removed) to facilitate the sparsity training algorithm to choose pruning candidates. Once the saliency scores are assigned, the weights are pruned in the ascending order of their saliency scores.

Figure 5-2 describes a sparsity training algorithm. It operates on a pre-trained dense network with weights W and input X. It receives the number of pruning rounds N and the per-round fraction of weights to prune (p). In each pruning round, the algorithm computes the saliency scores for all the weights and resets (set to *zero*) the p fraction of the weights with smallest saliency scores. Then, it proceeds to fine-tune the network. The outcome of this algorithm is a network where $Np|W|$ weights have been pruned. This style of pruning is called iterative pruning because we prune the model iteratively for N rounds. The fine-tuning phase after a pruning step gives a chance to the network to reconfigure itself after the selected weights have been pruned.

```
Algorithm 1: Standard Network Pruning with Fine-Tuning
Data: Pre-trained dense network with weights $W$, inputs $X$, number of pruning rounds $N$, fraction of
      weights to prune per round $p$.
Result: Pruned network with weights $W'$.
$W' \leftarrow W$;
for $i \leftarrow 1$ to $N$ do
    $S \leftarrow$ compute_saliency_scores($W'$);
    $W' \leftarrow W' -$ select_min_k($S, p|W|$);
    $W' \leftarrow$fine_tune($X, W'$)
end
return $W'$
```

Figure 5-2: Algorithm for pruning a given network. We start with a dense network, and compute the saliency scores for each of the candidates, followed by removing the smallest $p|W|$ number of weights, and fine-tuning the network in each round. At the end of N rounds, $Np|W|$ weights will have been removed.

Now that we have presented a general algorithm for pruning, we should go over some examples of different ways we implement them. Concretely, a practitioner might want to experiment with at least the following aspects:

1. Computing saliency scores.
2. Deciding on a pruning schedule.
3. Unstructured / Structured sparsity.

Seems daunting? Don’t fret! Let’s start with how to compute the saliency scores.

Saliency Scores

In the paper Optimal Brain Damage[1], LeCun et al. suggested that as much as 50% of the connections (weights) from a large network could be safely removed with minimal performance deterioration. A *random* removal could work for removing a few weights. However, when pruning a large number of weights, say 60%, we risk the removal of key weights. Hence, a more measured approach to select removal candidates is required. If we assign *saliency scores* to the model weights based on a certain criteria, those scores can be used to identify the candidates for removal. The criteria is picked such that lower the saliency score, the less important the weight is for the network. Therefore given a saliency score heuristic, in order to achieve, say, a 60% sparse model, the 60% of the weights with the lowest saliency scores can be pruned.

One such saliency score metric is using the magnitude of the weights, which has become a popular pruning technique because of its simplicity and effectiveness. Later on in this chapter, we have a project that relies on it for sparsifying a deep learning model.

The authors of the Optimal Brain Damage (OBD) paper approximate the saliency score using a second-derivative of the weights $\partial^2 L/\partial w_i^2$, where L is the loss function, and w_i is the candidate parameter for removal. Why do we want to compute the second-derivative? It is intuitive that the first derivative $\partial L/\partial w_i$ helps us compute the instantaneous slope of the loss function with respect to w_i. $\partial^2 L/\partial w_i \partial w_j$ gives us the rate at which the curvature of the loss function is changing with respect to w_i and w_j. The second derivative gives us a clearer insight into how important w_i might be to minimize the loss. Since computing pairwise second-derivatives for all i and j might be very expensive (even with just 10^4 weights, this quickly grows to 10^8 computations), OBD exclusively relies on second derivatives with respect to each weight ($\partial^2 L/\partial w_i^2$) and ignores cross interaction between the weights $\partial^2 L/\partial w_i \partial w_j$. The authors demonstrated that pruning by taking the second-derivative into consideration along with the weight magnitude could reduce the number of weights in a well-trained neural net by as much as 8x without a drop in classification accuracy.

Yet another technique could be momentum based pruning[2] which uses the magnitude of the *momentum* of the weights to evaluate their saliency. Instead of relying on the second-derivative to tell us what the curvature of the loss function with respect to w_i is, it relies on the momentum of the weights which is an exponentially *smoothed estimate* of $\partial L/\partial w_i$ over time. For instance, the momentum of weight w_i at training step $t = t + 1$ is given by:

$$M_i^{t+1} = \alpha M_i^t + (1-\alpha)\frac{\partial L}{\partial w_i^t}$$

[1] LeCun, Yann, John Denker, and Sara Solla. "Optimal brain damage." *Advances in neural information processing systems* 2 (1989).
[2] Dettmers, Tim, and Luke Zettlemoyer. "Sparse networks from scratch: Faster training without losing performance." *arXiv preprint arXiv:1907.04840* (2019).

As you can deduce, the parameter α changes the influence of the previous value of momentum computed at step t, which itself was a smooth estimate of the gradient at the previous step. This helps us even out temporary fluctuations in $\partial L/\partial w_i^t$, and gives us a better understanding of the curvature of the loss function. The authors prune weights which have low magnitude of momentum, follow it up with fine-tuning the network, and then *regrow* some weights (allow some weights to become non-zero again) in the network per layer, in proportion to the mean magnitude of momentum of weights in that layer.

There might be other ways of computing saliency scores, but they will all try to approximate the importance of a given weight at a certain point of time in the training process to minimize the loss function. The better we can estimate this importance, the more accurately we can prune weights in the network. Next, let's look at how many weights to prune in each round.

Pruning Schedules

The algorithm described in figure 5-2 uses a fixed pruning rate $$p$$. However, we could use variable pruning rates across the pruning rounds. The motivation behind using variable sparsity is that a pre-trained model's weights will get disrupted if we use a large pruning rate in the initial rounds. A gentle increase in the pruning rates can produce a more robust sparse model. For example, we could start with 10% sparsity in the first round, and increase this sparsity in the following rounds such that we reach a final sparsity of 80% in the last round. The pruning rates for each round can be fixed initially or we can use an algorithm such as polynomial decay which computes the rates for each step based on the initial sparsity, final sparsity and the number of pruning rounds.

This is also analogous to learning rate schedules. Whenever we fine-tune a pre-trained network, we want to gently warm up the learning rate, get it to an optimal value over some epochs, and then gradually decay it once again. The motivation is the same: to not disrupt the network too much at once. We would recommend the readers to try out various pruning schedules and empirically verify the schedule that works the best.

Finally, our approach towards sparsity so far has been to operate on a per-parameter basis. Hence, they are called unstructured pruning techniques. However, pruning can be extended to remove a group of weights as well. Such techniques are classified under structured pruning techniques. In the next section, we will discuss pruning at different granularities.

Sparsity Granularities

The examples of sparsity that we have considered so far are categorized under *unstructured* sparsity (also referred to as *irregular*, or *random* sparsity). Unstructured sparsity considers each weight independently for sparsity. The weights with the lowest saliency scores get axed.

This is great, but there is one tiny wrinkle here. To get improvement in latency (inference or training), we also need to make sure that the hardware can exploit the sparsity in our network and skip some computation. Most of our computation in deep learning models can be abstracted in the form of a matrix multiplication. If the sparsity is unstructured, CPUs, GPUs, TPUs, and other accelerators cannot assume a pattern, and thus have to do the full matrix multiplication anyway.

Structured sparsity as the name suggests, incorporates some sort of structure into the process of pruning. One way to do this is through pruning blocks of weights together (block sparsity). The blocks could be 1-D, 2-D or 3-D, and so on.

Let's start with a simple example of a dense fully connected layer. Instead of pruning arbitrary weights / connections, we can apply a 1-D granularity pruning on the weight matrix between two dense fully-connected layers to prune all the connections of a neuron in the first layer to the neurons in the second layer as shown in figure 5-3.

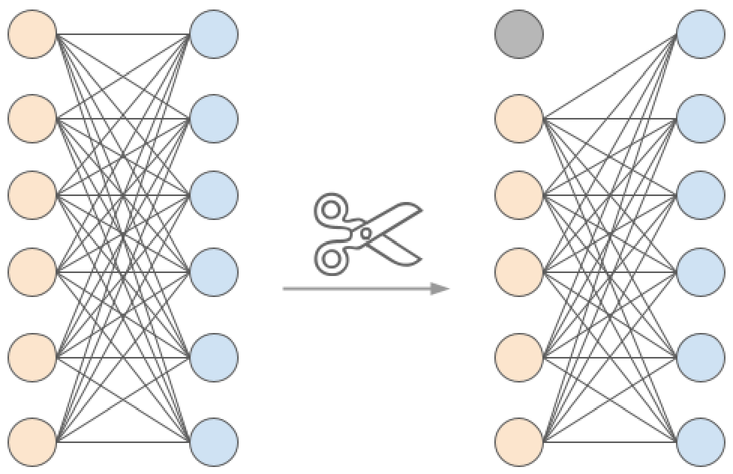


Figure 5-3: 1-D block pruning between two dense layers. The network on the right is the pruned version of the network on the left. The pruned network is a result of pruning the first row of the weight matrix.

By pruning all 6 weights from one neuron, instead of pruning 6 arbitrary weights, we can ignore the first row in the weight matrix. If the input was of shape [n, 6], where n is the batch size, and the weight matrix was of shape [6, 6], we can now treat this problem to be of input [n, 5] and weight matrix of size [5, 6]. This is because we have simply removed the first neuron.

Now, consider a convolution layer with 3x3 sized filters and 3 input channels. At 1-D granularity, a vector of weights is pruned. An entire kernel is pruned when the pruning is performed at 2-D granularity. We prune an entire channel for 3-D pruning. Figure 5-4 shows these granularities visually.

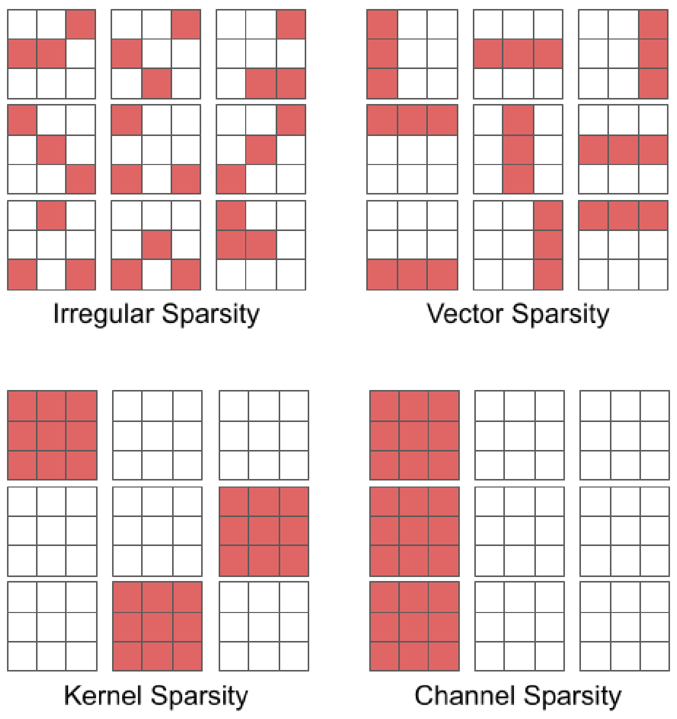


Figure 5-4: An example of sparsified weight matrices (zero-d weights are dark) each with 33% sparsity at various granularity levels. It shows the parameter layout for a convolutional layer which receives a 3-channel input. Each individual 3x3 matrix is a kernel. A column of 3 kernels represents a channel.

As you might notice, with such structured sparsity we can obtain efficient implementations which can drop unnecessary computation. In the case of this convolutional layer, we can drop rows, columns, kernels, and even whole channels. Libraries like XNNPACK[3,4] can help accelerate networks on a variety of web, mobile, and embedded devices, provided the user can design networks that match their constraints.

One might wonder what are the drawbacks of structured sparsity? It's easy: with unstructured sparsity we have the flexibility of pruning any weight that is suitable. However, with the additional constraint of structured sparsity, we lose that flexibility and might have to prune weights that are not necessarily the least important. In practice, unstructured sparsity achieves better model quality metrics than structured sparsity, for the same number of weights pruned.

Phew! It feels like we have gone through a lot of talk without much code! In chapter four, we trained a model to predict masks for pets to build snapchat like filters. Let's continue on the same project to demonstrate how we can create a pruned network without significant drop in accuracy in the next section.

[3] https://github.com/google/XNNPACK
[4] Elsen, E., Dukhan, M., Gale, T., & Simonyan, K. (2019). Fast Sparse ConvNets. arXiv, 1911.09723. Retrieved from https://arxiv.org/abs/1911.09723v1

Project: Lightweight model for pet filters application

Recall that our regular CNN model in the pet filters project consisted of thirteen convolution blocks and five deconvolution blocks. Our model achieved an accuracy of 85.11%. Here, we will prune the convolution blocks from block *two* (*zero* indexed) onwards. We will leave the deconvolution blocks untouched. We define a *create_model_for_pruning()* function which takes a pre-trained model and the names of the prunable blocks as inputs. It returns a model that is capable of sparse training. It clones the input model and wraps the prunable blocks for sparse training using TFMOT (Tensorflow Model Optimization) library. In this case, we prune the 50% of the weights in each prunable block using magnitude-based pruning. Note that the below code is in addition to the original segmentation project in chapter four. The code for this project is available as a Jupyter notebook **here**.

```
def create_model_for_pruning(m, prunables, info=True):
   def apply_pruning_to_conv_blocks(block):
       clone_block = models.clone_model(block)
       clone_block.set_weights(block.get_weights())

       if block.name in prunables:
           clone_block = tfmot.sparsity.keras.prune_low_magnitude(clone_block)

       return clone_block

   model_for_pruning = models.clone_model(
       m,
       clone_function=apply_pruning_to_conv_blocks,
   )

   optimizer = optimizers.Adam(learning_rate=LEARNING_RATE)
   loss = losses.SparseCategoricalCrossentropy(from_logits=True)

   model_for_pruning.compile(
       optimizer=optimizer,
       loss=loss,
       metrics='accuracy'
   )

   return model_for_pruning
```

The below code prepares the input arguments to create a model for pruning. The *prunable_blocks* variable is the list of names of prunable convolution blocks. We prune all convolution blocks from second (zero indexed) onwards. The *model* variable refers to the pet segmentation model from chapter four.

```
# Pruning start and end blocks
prunable_blocks = list(map(lambda l: l.name, model.layers[2:13]))
model_for_pruning = create_model_for_pruning(model, prunable_blocks)
prunable_layers = filter(lambda l: l.name in prunable_blocks,
model_for_pruning.layers)
```

```
prunable_layers[0].summary()
```

Note that TFMOT's *prune_low_magnitude* method creates wrapped prunable blocks that have additional non-trainable weights as shown in the summary of the second convolution block below. It uses these weights to apply sparsity to the layers and the blocks.

```
Model: "conv_block_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
prune_low_magnitude_conv2d_2 (None, 32, 32, 128)       147586
_________________________________________________________________
prune_low_magnitude_batch_no (None, 32, 32, 128)       513
_________________________________________________________________
prune_low_magnitude_re_lu_2  (None, 32, 32, 128)       1
=================================================================
Total params: 148,100
Trainable params: 74,112
Non-trainable params: 73,988
```

Let's train our pruning enabled model and evaluate its performance.

```
EPOCHS = 50
BATCH_SIZE = 16

# UpdatePruningStep() works in conjunction with the TFMOT pruning wrappers to update the wrappers after each optimization step.
update_pruning = tfmot.sparsity.keras.UpdatePruningStep()
callbacks = [update_pruning]
tds = train_prep_ds.cache().shuffle(1000,
reshuffle_each_iteration=True).batch(BATCH_SIZE)
vds = val_prep_ds.batch(256).cache()

hist = train(model_for_pruning, tds, vds, epochs=EPOCHS, callbacks=callbacks)
```

```
Epoch 1/50
184/184 [==============================] - 34s 90ms/step - loss: 0.8422 - accuracy: 0.6502
- val_loss: 0.8573 - val_accuracy: 0.6633
Epoch 2/50
184/184 [==============================] - 10s 56ms/step - loss: 0.7534 - accuracy: 0.6896
- val_loss: 0.8549 - val_accuracy: 0.6442
Epoch 3/50
184/184 [==============================] - 10s 57ms/step - loss: 0.6946 - accuracy: 0.7139
- val_loss: 0.7629 - val_accuracy: 0.6743
                            xxxxxxxxx Skip to 48th epoch xxxxxxxxx
Epoch 48/50
184/184 [==============================] - 11s 59ms/step - loss: 0.1204 - accuracy: 0.9442
- val_loss: 0.4923 - val_accuracy: 0.8451
Epoch 49/50
```

```
184/184 [==============================] - 10s 56ms/step - loss: 0.0910 - accuracy: 0.9561
- val_loss: 0.5369 - val_accuracy: 0.8473
Epoch 50/50
184/184 [==============================] - 11s 58ms/step - loss: 0.0819 - accuracy: 0.9596
- val_loss: 0.5619 - val_accuracy: 0.8460
```

```
# Evaluate the pruned model on the test set.
model_for_pruning_acc = model_for_pruning.evaluate(test_prep_ds.batch(256))[1]
print('Accuracy: ', model_for_pruning_acc)
```

```
Accuracy: 0.8471
```

Recall that the regular model performed with a 85.11% accuracy on the test set. Our pruned model performed with an accuracy of 84.71%. It's a slight drop in performance. Let's go ahead and strip the pruning weights from the model that were added by the TFMOT library as shown below.

```
# Strip the pruning wrappers from the model.
stripped_model = tfmot.sparsity.keras.strip_pruning(model_for_pruning)
```

The *stripped_model* is a sparse model after cleaning up the pruning artifacts. Let's print the convolution layer sparsity in each convolution block to ensure we are getting the configured 50% sparsity. The *get_conv_block_sparsity()* takes a convolution block as input and computes the percentage of *zeros* in its parameter matrix. We can that the convolution blocks from *two* to *twelve* (inclusive) have 50% sparsity.

```
def get_conv_block_sparsity(block):
   # Get block conv layer weights
   conv_weights = block.weights[0].numpy()
   non_zeros = np.count_nonzero(conv_weights)
   total = conv_weights.size
   zeros = total - non_zeros
   sparsity = (zeros/total)*100

   return (sparsity, total)

for block in stripped_model.layers:
   sparsity, total = get_conv_block_sparsity(block)
   print('Block: {} Sparsity: {}% Total Weights: {}'.format(block.name, sparsity,
total))
```

Output

```
Block: conv_block_0 Sparsity: 0.0% Total Weights: 864
Block: conv_block_1 Sparsity: 0.0% Total Weights: 18432
Block: conv_block_2 Sparsity: 50.0% Total Weights: 73728
Block: conv_block_3 Sparsity: 50.0% Total Weights: 147456
Block: conv_block_4 Sparsity: 50.0% Total Weights: 294912
Block: conv_block_5 Sparsity: 50.0% Total Weights: 589824
Block: conv_block_6 Sparsity: 50.0% Total Weights: 1179648
Block: conv_block_7 Sparsity: 50.0% Total Weights: 2359296
```

```
Block: conv_block_8 Sparsity: 50.0% Total Weights: 2359296
Block: conv_block_9 Sparsity: 50.0% Total Weights: 2359296
Block: conv_block_10 Sparsity: 50.0% Total Weights: 2359296
Block: conv_block_11 Sparsity: 50.0% Total Weights: 2359296
Block: conv_block_12 Sparsity: 50.0% Total Weights: 4718592
Block: conv_transpose_block_13 Sparsity: 0.0% Total Weights: 4718592
Block: conv_transpose_block_14 Sparsity: 0.0% Total Weights: 1179648
Block: conv_transpose_block_15 Sparsity: 0.0% Total Weights: 294912
Block: conv_transpose_block_16 Sparsity: 0.0% Total Weights: 73728
Block: conv_transpose_block_17 Sparsity: 0.0% Total Weights: 1728
```

Figure 5-5 shows the comparison of compressed sizes of our regular model and its 50% sparse version. We used Tensorflow's *save_model()* API and zipped the model files using *gzip*. In addition to the usual models, the figure also shows compressed size comparisons for their TFLite variants. The sparse models are 27% smaller than the regular models. We would recommend the readers to try this experiment with even higher sparsity values and observe the resulting accuracies.

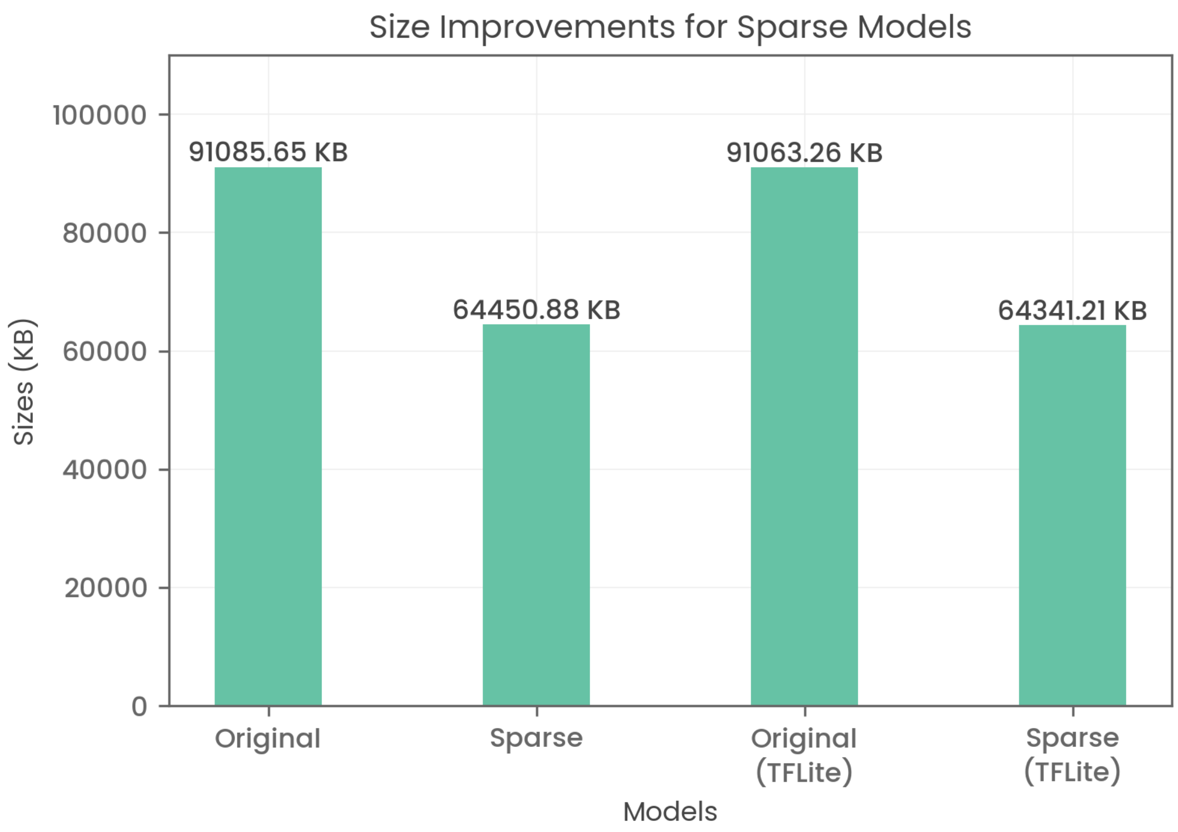


Figure 5-5: Compressed size comparison of a regular model with its 50% sparse version.

In this project, we used a consistent sparsity value of 50% across the epochs. However, we could also use a variable sparsity value across epochs as explained in the earlier section.

Active Research

Some recent works like Sparse Evolutionary Training[5] (SET), Dynamic Sparse Reparametrization[6] (DSR) and Sparse Networks from Scratch[7] (SNFS) have introduced an additional step to regrow pruned weights after fine tuning the pruned model in each pruning round. We briefly introduced the concept of parameter regrowth in the Salient Scores sections. The motivation behind parameter regrowth is to minimize the impact of the pruned weights on the model loss. By adding some of the pruned weights again, the regrowth step attempts to align the loss value to the pre-pruning levels. The regrowth step, in some cases, also redistributes the lost weight across layers such that the growth is higher in layers which have a higher impact on the loss value.

Han et al. in their seminal paper titled "Learning both Weights and Connections for Efficient Neural Networks[8]" proposed a three step approach for pruning. The three steps are: Train Connectivity, Prune Connections, and Train Weights. The algorithm in figure 5-2 is based on these three steps. Their approach is called iterative pruning since it continuously prunes connections and fine-tunes weights (the Train Weights step) until the model achieves its best performance.

Frankle et al.'s work[9] on the Lottery Ticket Hypothesis took a different look at pruning. They postulated that within every large network lies a smaller network which can be extracted without using the fine-tuned weights and retrained to match or exceed the performance of the larger network. Liu et al. in their work titled "Rethinking the Value of Network Pruning[10]" replicated Frankle et al.'s work using structured pruning and demonstrated that the pruned architecture with random initialization is no worse than the pruned architecture with the trained weights. In essence, the structural aspect of pruning helps the network achieve a structure which could be trained to achieve a better performance than the trained dense network even without using its trained weights. Lottery Ticket Hypothesis shifted the focus from training weights towards the hidden structure. The lottery based pruning techniques strive to discover this structure. Zhou et al. in their work[11] highlighted the importance of the signs on the weights to the learning process. They demonstrated that the networks with randomly initialized weights combined with a signed constant mask performs as well as the trained network. The signs of such a mask are chosen based on the trained weights. These experiments are yet to be replicated with the large networks.

---

[5] Mocanu, Decebal Constantin, et al. "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science." *Nature communications* 9.1 (2018): 1-12.
[6] Mostafa, Hesham, and Xin Wang. "Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization." *International Conference on Machine Learning*. PMLR, 2019.
[7] Dettmers, Tim, and Luke Zettlemoyer. "Sparse networks from scratch: Faster training without losing performance." *arXiv preprint arXiv:1907.04840* (2019).
[8] Han, Song, et al. "Learning both weights and connections for efficient neural network." *Advances in neural information processing systems* 28 (2015).
[9] Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." *arXiv preprint arXiv:1803.03635* (2018).
[10] Liu, Zhuang, et al. "Rethinking the value of network pruning." *arXiv preprint arXiv:1810.05270* (2018).
[11] Zhou, Hattie, et al. "Deconstructing lottery tickets: Zeros, signs, and the supermask." *Advances in neural information processing systems* 32 (2019).

Weight sparsity has typically been the primary focus of sparsity research. However, in the last few years, some researchers have started to explore activation sparsity as well. Activation sparsity involves sparsifying activation maps to produce robust models. Rhu et al., through their work on Compression DMA Engine[12], observed that a non-trivial fraction of activation values for ReLU activation function are naturally sparse. Kurtz et al. leveraged this idea in their work[13] to achieve 1.8x (approx.) inference performance gains for ResNets and MobileNets. More recently, the researchers have started to combine these two forms to achieve both accuracy and latency gains.

Weight Sharing using Clustering

Recall that in quantization, we divided the original floating point domain between $x_{\mathbf{min}}$ and $x_{\mathbf{min}}$ into 2^b non-overlapping bins that collectively span the entire range, where b is the number of bits allocated for quantization. All values within the same bin share the same weight. Hence, we can view *weight sharing* as the general concept behind quantization.

However, what happens if our $x_{\mathbf{min}}$ and $x_{\mathbf{max}}$ were outliers, and the real data was clustered in some smaller concentrated ranges? Quantization will still assign an equal number of precision bits to all subranges of the same length. In this scenario, the dequantization error would be large for ranges where the data is densely distributed. Quantization-aware training can mitigate some of the losses by making the network resilient to the errors, but if we want to minimize the dequantization error, this is not an ideal situation. Refer to figure 5-6 for an illustration of this problem.

[12] Rhu, Minsoo, et al. "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks." *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.
[13] Kurtz, Mark, et al. "Inducing and exploiting activation sparsity for fast inference on deep neural networks." *International Conference on Machine Learning*. PMLR, 2020.

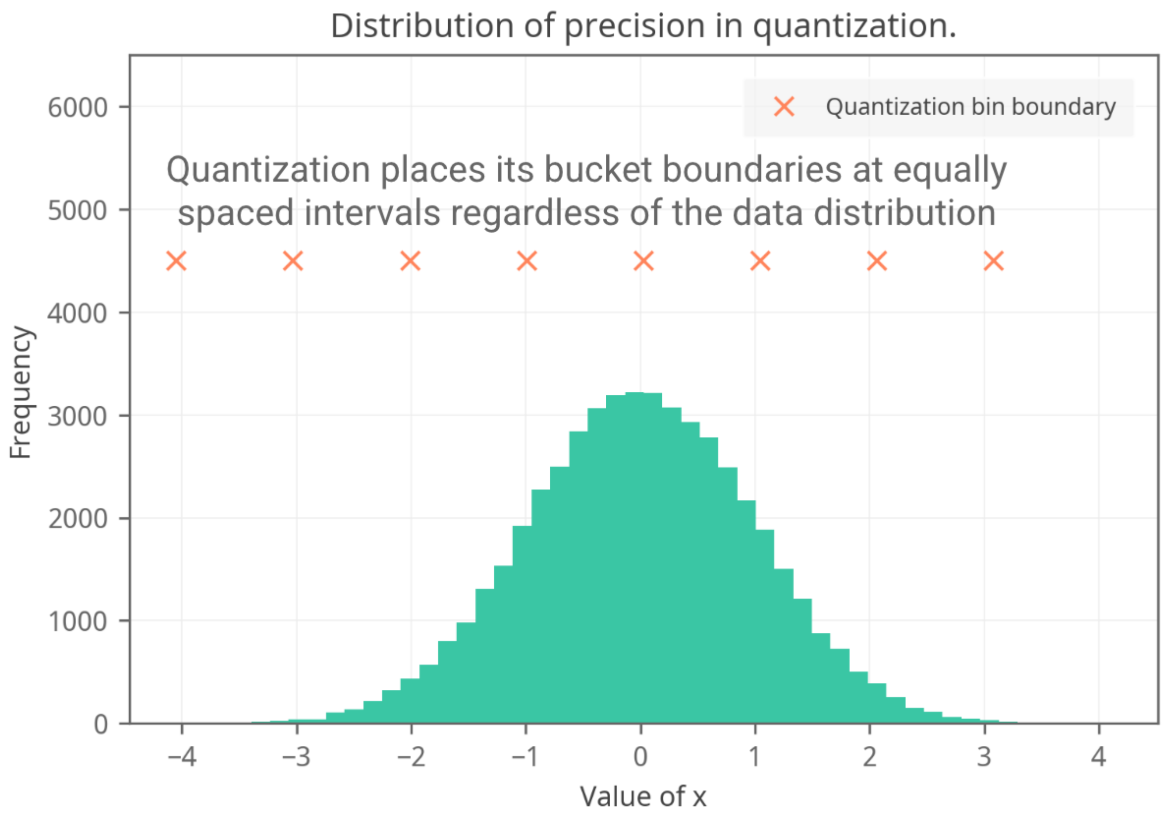


Figure 5-6: In the above figure we work with a hypothetical distribution of a variable x. Notice that x is likely to take a value in [-4, 4]. However quantization linearly assigns precision to all ranges equally based on the minimum and maximum values it observes for x. Each quantization bin boundary is denoted by a cross. This is not ideal because the precision allocated to the range [-4.0, -2.0] or [2.0, 4.0] (spanning two quantization bins) is the same as the precision allocated to [-1, 1]. [-1, 1] has a much higher likelihood of x.

Can we do better such that we assign more bits to regions where more of our data is distributed, and fewer bits to the sparser regions? Recall that huffman encoding does this by trying to create a huffman tree based on symbol frequency. As a result it comes up with a variable-length code, where a smaller length code is assigned to frequently occurring symbols such that the total number of bits required to encode a typical message can be minimized.

Let's try a non-linear mode of quantization where we can set the range of the buckets based on distribution of the data. One way to do it is via K-Means Clustering.

K-means clustering is a variation on the simpler quantization method we saw earlier. Assuming we had k bins earlier, where $k = 2^b$ in the case of quantization, in the case of k-means clustering we will have k centroids. The aim is to learn these k centroids as well as we can, so that they closely mimic the original distribution of the tensor's elements.

For a moment, let's assume that the centroids we obtain are optimal, i.e. the reconstruction error when we decode the encoded representation is minimal[14]. In such a scenario, we can list all the centroids in a *codebook* and replace each element in our tensor with the index of the centroid in the codebook closest to that element. The decoding process simply requires replacing the centroid indices in the encoded tensor by looking up the value of the respective centroid at that index in the codebook.

If we had k centroids that took w bytes each (typically $w = 4$ for floating point values), the codebook will cost us wk bytes to store. For each tensor element we will now store only the index of its centroid in the codebook, which will only take up $\log_2(k)$ bits. For a tensor with n elements, the cost would be $n\log_2(k)/8$ bytes. Originally, storing all the elements would have cost wn bytes.

Therefore, the compression ratio turns out to be:

$$\textbf{Compression Ratio} = \frac{\textbf{Original Size}}{\textbf{Size of codebook} + \textbf{Size of encoded representation}}$$

$$= \frac{wn}{wk + n\log_2(k)/8}$$

Substituting $w = 4$ we get,

$$\textbf{Compression Ratio} = \frac{32n}{32k + n\log_2(k)}$$

Say we are compressing a small dense layer with a weight matrix of size 64x64, the number of elements n is $64x64 = 4096$. Assuming $k = 16$, and $w = 4$ as usual, the compression ratio using the above formula computes to: 7.75.

Table 5-1 lists the compression ratios for different values of k, using the above values of n and w.

Number of Centroids (k)	Compression Ratio
2	31.51
4	15.75
8	10.44
16	7.75
32	6.09
64	4.92

[14] The reconstruction error can simply be the mean squared error between the original tensor and the decoded tensor, as we did when working with quantization.

128	4.0

Table 5-1: Number of centroids v/s the compression ratio, assuming $n = 4096$, and $w = 4$.

As we increase the value of k, the compression ratio goes down since the codebook size will increase, hence the denominator in the compression ratio expression increases. However, with the increase in k, the quality of the representation typically improves as each centroid has a smaller range to cover, therefore the reconstruction error would be smaller. At some point, k

Now coming back to the hard problem that we punted earlier: how do we compute the centroids? The objective function to optimize k-means clustering problems is the within-cluster-sum-of-squares (WCSS) metric.

$$\mathbf{WCSS} = \arg\min_{C} \sum_{i=1}^{k} \sum_{x \in c_i} |x - c_i|^2 .$$

Here, we are trying to find a set C which has k centroids $(C = c_0, c_1, \ldots, c_{k-1})$, such that for each element x that is closest to the centroid c_i in C, the sum of the squared distances between the every such weight and c_i pair is minimized.

In our case, since we are trying to find the optimal distribution of the centroids in a single dimension, the centroids will also be 1-dimensional. In other scenarios, we can also apply clustering over blocks of elements of size d. In this case the centroids would be d -dimensional.

With all that being said, how do we compute C? Turns out that the problem of computing the exact optimal C is NP-Hard. However, we can approximate this computation. The general algorithm is as follows:

```
Algorithm: Clustering a given set of elements in a tensor $\mathbf{X}$.

  1. Initialization: Select an initial set of $k$ centroids.
  2. Assignment step: Assign each element in $\mathbf{X}$ to the closest
     centroid. At the end of this step, there should be $k$ clusters.
  3. Update step: Compute the new centroids by computing the mean of
     all the points assigned to each cluster.
  4. Run steps (2) & (3) until convergence.
```

Notice that this algorithm's runtime is not deterministic and depends on the initial seed centroids, which can be selected in many ways. For example they can be chosen randomly amongst the provided list of data points. Another scheme is to select centroids that are linearly spaced amongst these data-points. Yet another scheme is to incrementally and probabilistically select centroids from points based on the candidate centroid's distances from all other centroids[15]. We can also solve the above problem using gradient descent[16,17].

Enough talk, let's try to use k-means clustering with a real example. The code for the next few exercises is available **here** as a Jupyter notebook.

Using clustering to compress a 1-D tensor.

Let us first implement the Within-Cluster-Sum-of-Squares (WCSS) loss as shown above.

```
def get_clustering_loss(x_var, centroids_var):
 """Computing the loss to optimize."""
 # Compute the pairwise squared distance between the input (x) and the
 # centroids.
 distances = tf.subtract(tf.reshape(x_var, (-1, 1)), centroids_var) ** 2
 best_distances = tf.math.reduce_min(distances, axis=1)
 return tf.reduce_mean(best_distances)
```

Now we can implement the method that learns the centroids. We will provide it `x`, which has `N` scalar values, the number of clusters we want to create (`num_clusters`), the number of steps we want to run this algorithm for, and finally the learning rate.

```
def get_centroids(x, num_clusters, num_steps=10, learning_rate=1e-3,
                  verbose=1):
 """Get num_clusters centroids for the given value of x."""
 # Pick initial centroids that are evenly spaced.
 x_sorted = np.sort(x.flatten())
 centroids_init = np.linspace(x_sorted[0], x_sorted[-1], num_clusters)

 # Construct the variables in this optimization problem.
 # We will not update 'x', and hence it is not trainable.
 x_var = tf.Variable(initial_value=x_sorted, trainable=False)

 # The centroids are going to be updated, thus they will be trainable.
 centroids_var = tf.Variable(initial_value=centroids_init, trainable=True)

 opt = tf.keras.optimizers.SGD(learning_rate=learning_rate)

 for step_idx in range(num_steps):
```

[15] David Arthur and Sergei Vassilvitskii. 2007. K-means++: the advantages of careful seeding. In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '07). Society for Industrial and Applied Mathematics, USA, 1027–1035.
[16] Bottou, Leon, and Yoshua Bengio. "Convergence properties of the k-means algorithms." *Advances in neural information processing systems* 7 (1994).
[17] Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149* (2015).

```
    with tf.GradientTape() as tape:
      loss = get_clustering_loss(x_var, centroids_var)
      if verbose == 2 or (
          verbose == 1 and
          ((step_idx + 1) % ((int)(num_steps / 5)) == 0)):
        print(f'Step: {step_idx + 1}, Loss: {loss:.5f}.')

    # Compute the gradients w.r.t. only the centroids_var.
    gradients = tape.gradient(loss, [centroids_var])

    # Update the centroids_var.
    opt.apply_gradients(zip(gradients, [centroids_var]))
  return centroids_var
```

Since we have learnt the best possible centroids, we can also encode x using it.

```
def encode_x(x, centroids):
  """Encode the given x using the given centroids."""
  x_flattened = tf.reshape(x, [-1])
  # Compute pairwise squared distances between all the elements of x and the
  # centroids.
  distances = tf.subtract(
      tf.reshape(x_flattened, (-1, 1)),
      centroids) ** 2
   # Now compute the distance to the closest centroid.
  best_distances = tf.math.reduce_min(distances, axis=1)
   # Create an indicator variable matrix, where 1.0 at index [i][j] denotes that
  # for the i-th element, the j-th centroid is the closest.
  is_closest = tf.cast(
    tf.equal(distances, tf.reshape(best_distances, (-1, 1))),
    dtype=tf.float64)
   # Now lookup the centroid indices.
  encoded_x_flattened = tf.math.argmax(is_closest, axis=1)

  # Finally, reshape the array to the original shape.
  return tf.reshape(encoded_x_flattened, tf.shape(x))
```

Decoding x is also fairly simple as we discussed earlier.

```
def decode_x(encoded_x, centroids):
  """Decode the x back from the centroids."""
  encoded_x_flattened = tf.reshape(encoded_x, [-1])
  # Lookup the centroids for their values, and then reshape it back to the
  # original shape.
  return tf.reshape(
      tf.gather(centroids, encoded_x),
      tf.shape(encoded_x))
```

The reconstruction loss is the regular mean squared error.

```
def compute_reconstruction_loss(x, decoded_x):
 """Compute a simple mean squared error (MSE) loss."""
 loss = tf.math.reduce_mean((x - decoded_x)**2)
 return loss
```

Finally the below snippet connects everything together, encodes x using num_clusters, and then tries to decode it and measure the reconstruction error.

```
def simulate_clustering(x, num_clusters, num_steps=20, learning_rate=5e-3,
                       verbose=1):
 def vprint(v, str):
   if v:
     print(str)

 vprint(verbose, 'Computing the centroids.')
 computed_centroids = get_centroids(
     x, num_clusters, num_steps=num_steps, learning_rate=learning_rate,
     verbose=verbose)
  vprint(verbose, 'Encoding x using the computed centroids.')
 encoded_x = encode_x(x, computed_centroids)

 # Now finally decode the encoded x, and compute the reconstruction error.
 vprint(verbose, 'Decoding x using the computed centroids.')
 decoded_x = decode_x(encoded_x, computed_centroids)
 reconstruction_error = compute_reconstruction_error(x, decoded_x)
 vprint(verbose, f'Final reconstruction error: {reconstruction_error:.4f}.')
 return decoded_x, computed_centroids, reconstruction_error
```

In order to test our algorithm, let us create a random x like we saw in figure 5-6, and try to run our clustering algorithm on it.

```
# Setting a seed here helps us reproduce the same output over multiple runs.
np.random.seed(1337)

# Let's create a tensor with a normal (gaussian) distribution.
x = np.random.normal(size=50000, loc=0.0, scale=1.0)

num_clusters = 8
x_decoded, centroids, reconstruction_error = simulate_clustering(
   x, num_clusters, num_steps=5000, learning_rate=2e-1)
```

The following is the log of the above training.

```
Computing the centroids.
Step: 1000, Loss: 0.04999.
```

```
Step: 2000, Loss: 0.03865.
Step: 3000, Loss: 0.03458.
Step: 4000, Loss: 0.03455.
Step: 5000, Loss: 0.03455.
Encoding x using the computed centroids.
Decoding x using the computed centroids.
Final reconstruction error: 0.0345.
```

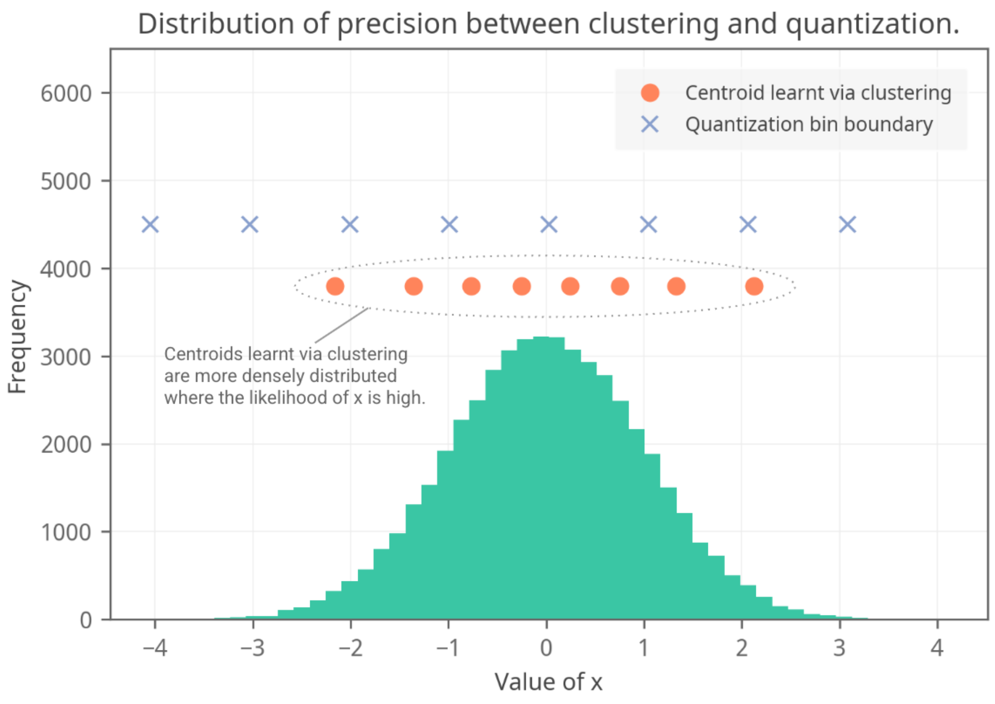
As we see, the reconstruction error matches the final loss. To be thorough, we should ideally test the reconstruction error using a different x, drawn from the true distribution, otherwise the centroids will be overfitting to the given x. If the centroids are truly good, we should see a similar reconstruction error with this new x too. We leave this as an exercise for the reader.

Regardless, what we have achieved with this exercise is that we have essentially encoded x using just eight centroids each of which is a floating point value. Then for each of the 50000 scalars can just be encoded in 3 bits each theoretically. Using the above compression ratio formula, plugging $n = 50000$ and $k = 8$, the compression ratio is 10.65, which is quite impressive.

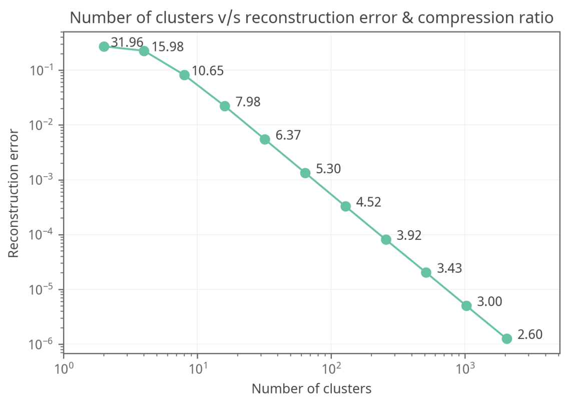
This is all good. Does clustering solve the problem that tripped quantization in figure 5-6? If you see figure 5-7 (a), it demonstrates the eight centroids that clustering picked (orange dots). Notice that the centroids are densely distributed around the ranges where the frequency of x is high. How satisfying is that? You can rely on clustering to put its centroids where the data is.

Next, we ran some calculations to verify how the reconstruction error changes as we increase the number of clusters (k). Figure 5-7 (b) shows the plot. Note that both the x and y axes are in log-scale.

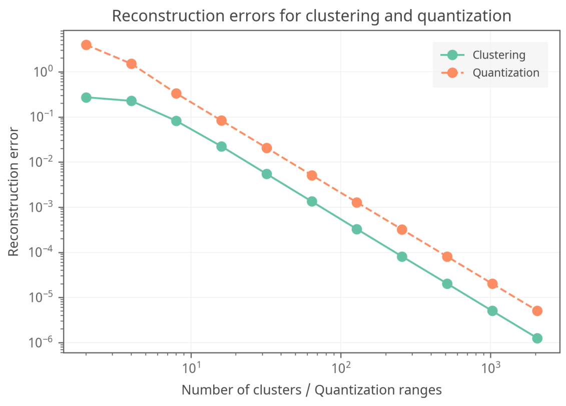
Finally, figure 5-7 (c) compares the reconstruction errors between quantization and clustering on the same x that we used. We picked the number of quantization bits (b) such that $2^b = k$, i.e. the number of quantization bins (2^b) and the number of clusters / centroids is the same. This makes them roughly comparable compression ratio wise, although as k increases, the cost of keeping a codebook starts becoming a significant cost too. But we will ignore that for now. Do notice how the almost perfect trend in the reconstruction errors of the two methods. Quantization predictably does worse, and this is an empirical verification of what we saw earlier in figure 5-6 and also reinforced by how clustering does a better job in figure 5-7 (a).



(a)



(b)



(c)

Figure 5-7: (a) Distribution of centroids (and hence precision) in the case of clustering. The centroids mimic the distribution of x. (b) Change in reconstruction loss as the number of clusters (both x and y axes are in log scale). (c) Comparison of reconstruction errors for both clustering and quantization with the same x. Again, both axes are in log scale.

Now that we have verified how clustering works on a hypothetical tensor. We should get working on some more real life examples.

Mars Rover beckons again! Can we do better with clustering?

Remember the Mars Rover transmission project from Chapter 2? It seems like the scientists back at the Jet Propulsion Laboratory in Pasadena, need your help again. They would appreciate it if you can help compress the transmissions some more, so that they can use the saved bandwidth for other projects.

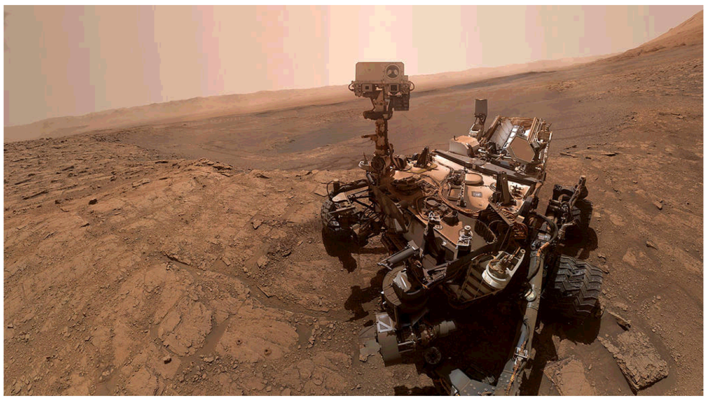
The goal in this scenario is to use clustering to outperform our quantization-based solution. Let us implement a method that simulates the transmission using clustering and measures the reconstruction error.

```
def simulate_transmission_clustering(img, num_clusters):
 decoded_img, _, reconstruction_error = simulate_clustering(
     img, num_clusters, num_steps=10, learning_rate=2e-1, verbose=2)
 print(f'Reconstruction error: {reconstruction_error:.4f}.')
 plt.axis('off')
 plt.imshow(decoded_img)
```

Let us try clustering with 22 clusters and centroids.

```
num_clusters = 22
simulate_transmission_clustering(img, num_clusters)
```

```
Computing the centroids.
Step: 1, Loss: 0.00019.
Step: 2, Loss: 0.00019.
Step: 3, Loss: 0.00019.
Step: 4, Loss: 0.00019.
Step: 5, Loss: 0.00019.
Step: 6, Loss: 0.00019.
Step: 7, Loss: 0.00019.
Step: 8, Loss: 0.00019.
Step: 9, Loss: 0.00019.
Step: 10, Loss: 0.00019.
Encoding x using the computed centroids.
Decoding x using the computed centroids.
Final reconstruction error: 0.0002.
Reconstruction error: 0.0002.
```



Notice that the reconstruction error using clustering with 22 clusters (0.0002) is lower than that using quantization with 5 bits (0.0003).

```python
# Compute various statistics related to size when using quantization /
# clustering.

def get_quantized_size_bytes(num_elements, num_quantization_bits):
 # num_elements = img.size
 return (num_elements * num_quantization_bits) / 8.0

def get_clustered_size_bytes(num_elements, num_clusters,
                             floating_point_word_size=4):
 codebook_size_bytes = num_clusters * floating_point_word_size
 encoded_img_size_bytes = (num_elements * np.math.log2(num_clusters)) / 8.0
 return codebook_size_bytes + encoded_img_size_bytes


def compute_compression_ratio(num_elements, num_quantization_bits, num_clusters,
                              floating_point_word_size=4):
 original_size_bytes = num_elements * floating_point_word_size
 quantized_size_bytes = get_quantized_size_bytes(num_elements, num_bits)
 clustered_size_bytes = get_clustered_size_bytes(num_elements,
                                                num_clusters)
 quant_vs_clustering_compression_ratio = (quantized_size_bytes * 1.0 /
                                         clustered_size_bytes)
 original_vs_clustering_compression_ratio = (original_size_bytes * 1.0 /
                                            clustered_size_bytes)
 print(f'Original input size: {original_size_bytes} bytes.')
 print(f'Quantized input size: {quantized_size_bytes} bytes.')
 print(f'Clustered input size: {clustered_size_bytes} bytes.')
 print('Compression Ratio (Quant Size / Clustered Size): '
       f'{quant_vs_clustering_compression_ratio:.3f}.')
```

We also have a method that computes the compression ratios for both clustering and quantization. There is nothing novel going on there, so we are skipping listing the code here, but feel free to go through it in the Jupyter notebook. Regardless, this is the output we get from it.

```python
stats = compute_compression_ratio(img.size, num_bits, num_clusters)
```

```
Original input size: 7320312 bytes.
Quantized input size: 1143798.75 bytes.
Clustered input size: 1020226.46 bytes.
Compression Ratio (Quant Size / Clustered Size): 1.121.
```

Clustering thus leads to a **12.1%** reduction in size compared to quantization while achieving a similar reconstruction loss.

Simulating clustering on a dummy dense fully-connected layer

Now that we have looked at how to compress a given tensor, applied it to the Mars Rover problem, wouldn't it be great if we can also use clustering to compress a dense layer as used in a model? Let's do that then!

We can start with getting the inputs, weights, and bias initialized randomly using the normal (gaussian) distribution with mean = 0.0, and standard deviation = 1.0. We will also simulate the forward-pass behavior.

```
np.random.seed(10007)

def get_random_matrix(shape):
 return np.random.normal(0.0, 1.0, size=shape)

# Populate the inputs, weights and bias.
inputs = get_random_matrix([100, 300])
weights = get_random_matrix([300, 500])
bias = get_random_matrix([500])

y = np.maximum(np.matmul(inputs, weights) + bias, 0)
```

In the snippet below, we will quantize the given weights matrix with 8 bits of precision. We will leave the biases untouched since they do not contribute significantly to the layer's footprint.

```
num_bits = 8
weights_dequantized, weights_reconstruction_error_quant = simulate_quantization(
   weights, num_bits)
print(f'Weights reconstruction error: {weights_reconstruction_error_quant:.5f}')

y_via_quant = np.maximum(np.matmul(inputs, weights_dequantized) + bias, 0)
output_reconstruction_error_quant = compute_reconstruction_error(
   y, y_via_quant).numpy()
print(f'Output reconstruction error: {output_reconstruction_error_quant:.4f}')
```

The weight reconstruction loss and the output reconstruction errors are as follows.

```
Weights reconstruction error: 0.00036
Output reconstruction error: 0.0581
```

Now that we have a baseline with quantization, can we do better in terms of the weight matrix size without regressing on the reconstruction error?

```
num_clusters = 128
weights_decoded, _, weights_reconstruction_error = simulate_clustering(
   weights, num_clusters)
print('Weights reconstruction error: '
     f'{weights_reconstruction_error:.4f}')
```

```
y_via_clustering = np.maximum(np.matmul(inputs, weights_decoded) + bias, 0)
output_reconstruction_error_clustering = compute_reconstruction_error(
   y, y_via_clustering).numpy()
print(f'Output reconstruction error: {output_reconstruction_error_clustering:.4f}')
```

Let’s see what the reconstruction losses for clustering are.

```
Computing the centroids.
Step: 4, Loss: 0.00036.
Step: 8, Loss: 0.00036.
Step: 12, Loss: 0.00036.
Step: 16, Loss: 0.00036.
Step: 20, Loss: 0.00036.
Encoding x using the computed centroids.
Decoding x using the computed centroids.
Final reconstruction error: 0.0004.
Weights reconstruction error: 0.0004
Output reconstruction error: 0.0543
```

As we see, the final output reconstruction error using clustering with 128 clusters is better than the reconstruction error achieved using quantization. Does it achieve a better compression though?

```
Original input size: 600000 bytes.
Quantized input size: 150000.00 bytes.
Clustered input size: 131762.00 bytes.
Compression Ratio (Quant Size / Clustered Size): 1.138.
```

Clustering achieves a very similar output reconstruction loss as quantization, and leads to a weights encoding that is **13.8% smaller** than quantization. Again, these are on top of the gains provided by quantization (4x smaller). However, note that if the size of the weight matrix was very small, the size of the codebook would have dominated the total size of the encoded representation. Since our weight matrix was big enough (100x300), it allowed us to demonstrate saving space when compared to quantization.

Now that we are familiar with how k-means clustering works on a single layer, let’s apply it to an end-to-end deep learning model.

Using Clustering to compress a deep learning model

In this section, we will continue to work with the Speech Commands dataset which we used in chapter 3. To recap, it is a dataset of spoken words designed to help train and evaluate keyword spotting systems used in common home automation devices like the Google Home, Amazon Alexa, etc. For each possible input in the dataset, it can be classified into one of the 12 classes (each representing either a target word, with one class for ‘unknown’). The code for this project is available **here** as a Jupyter notebook.

Train the baseline model

Let’s go ahead and train the same convolutional network we used in chapter 3.

```
width_multiplier = 1.0
params = {
    'learning_rate': 1e-3,
    'dropout_rate': 0.5,
}
model_wm_10, _ = train_model(
    width_multiplier, params, epochs=5)
```

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_3 (InputLayer)        [(None, 124, 129)]        0         
                                                                 
 conv1d_12 (Conv1D)          (None, 124, 32)           37184     
                                                                 
 batch_normalization_12 (Bat  (None, 124, 32)          128       
 chNormalization)                                                
                                                                 
 conv1d_13 (Conv1D)          (None, 124, 32)           9248      
                                                                 
 batch_normalization_13 (Bat  (None, 124, 32)          128       
 chNormalization)                                                
                                                                 
 max_pooling1d_6 (MaxPooling  (None, 31, 32)           0         
 1D)                                                             
                                                                 
 dropout_10 (Dropout)        (None, 31, 32)            0         
                                                                 
 conv1d_14 (Conv1D)          (None, 31, 64)            18496     
                                                                 
 batch_normalization_14 (Bat  (None, 31, 64)           256       
 chNormalization)                                                
                                                                 
 conv1d_15 (Conv1D)          (None, 31, 64)            36928     
                                                                 
 batch_normalization_15 (Bat  (None, 31, 64)           256       
 chNormalization)                                                
                                                                 
 max_pooling1d_7 (MaxPooling  (None, 7, 64)            0         
 1D)                                                             
                                                                 
 dropout_11 (Dropout)        (None, 7, 64)             0         
                                                                 
 conv1d_16 (Conv1D)          (None, 7, 128)            73856     
                                                                 
 batch_normalization_16 (Bat  (None, 7, 128)           512       
 chNormalization)                                                
                                                                 
 conv1d_17 (Conv1D)          (None, 7, 128)            147584    
                                                                 
 batch_normalization_17 (Bat  (None, 7, 128)           512       
 chNormalization)                                                
 max_pooling1d_8 (MaxPooling  (None, 1, 128)           0
 1D)

 dropout_12 (Dropout)        (None, 1, 128)            0

 flatten_2 (Flatten)         (None, 128)               0

 dropout_13 (Dropout)        (None, 128)               0

 dense_4 (Dense)             (None, 512)               66048

 dropout_14 (Dropout)        (None, 512)               0

 dense_5 (Dense)             (None, 12)                6156

 activation_2 (Activation)   (None, 12)                0

=================================================================
Total params: 397,292
Trainable params: 396,396
Non-trainable params: 896
_________________________________________________________________
Epoch 1/50
125/125 [==============================] - 7s 27ms/step - loss: 1.9458 -
categorical_accuracy: 0.6004 - val_loss: 2.7220 - val_categorical_accuracy: 0.1080
Epoch 2/50
125/125 [==============================] - 3s 26ms/step - loss: 1.5526 -
categorical_accuracy: 0.6337 - val_loss: 2.3224 - val_categorical_accuracy: 0.2292
Epoch 3/50
125/125 [==============================] - 4s 32ms/step - loss: 1.3961 -
categorical_accuracy: 0.6434 - val_loss: 2.0764 - val_categorical_accuracy: 0.2779
Epoch 4/50
125/125 [==============================] - 3s 23ms/step - loss: 1.2690 -
categorical_accuracy: 0.6570 - val_loss: 1.6459 - val_categorical_accuracy: 0.4417
Epoch 5/50
125/125 [==============================] - 3s 25ms/step - loss: 1.1453 -
categorical_accuracy: 0.6769 - val_loss: 1.4320 - val_categorical_accuracy: 0.5270
. . .
Epoch 46/50
125/125 [==============================] - 2s 15ms/step - loss: 0.4101 -
categorical_accuracy: 0.9208 - val_loss: 0.5628 - val_categorical_accuracy: 0.8726
Epoch 47/50
125/125 [==============================] - 2s 14ms/step - loss: 0.3944 -
categorical_accuracy: 0.9246 - val_loss: 0.6327 - val_categorical_accuracy: 0.8605
Epoch 48/50
125/125 [==============================] - 2s 15ms/step - loss: 0.3945 -
categorical_accuracy: 0.9234 - val_loss: 0.6118 - val_categorical_accuracy: 0.8675
Epoch 49/50
125/125 [==============================] - 2s 15ms/step - loss: 0.3947 -
categorical_accuracy: 0.9253 - val_loss: 0.5722 - val_categorical_accuracy: 0.8800
Epoch 50/50
125/125 [==============================] - 2s 14ms/step - loss: 0.3912 -
categorical_accuracy: 0.9274 - val_loss: 0.6354 - val_categorical_accuracy: 0.8562
```

The model seems to have trained to a reasonable accuracy. We can now persist it to disk in the SavedModel format.

```
import tempfile
```

```
_, keras_file = tempfile.mkstemp('.h5')
print('Saving model to: ', keras_file)
tf.keras.models.save_model(model_wm_10, keras_file, include_optimizer=False)
```

```
Saving model to:  /tmp/tmp130dorox.h5
```

Clustering the model weights

Similar to what we did in sparsity, we will use the Tensorflow Model Optimization Toolkit (TFMOT) library to cluster our model's weights. Installing the library is fairly simple as we saw earlier.

```
! pip install -q tensorflow-model-optimization

import tensorflow_model_optimization as tfmot
```

You can now invoke the model clustering using the `cluster_weights` API by providing the model to be clustered and two important parameters: (1) the number of clusters, and (2) how to get the initial centroids. In this setup we use 16 centroids that are initially linearly spaced, similar to what we did in our previous examples.

```
cluster_weights = tfmot.clustering.keras.cluster_weights
CentroidInitialization = tfmot.clustering.keras.CentroidInitialization

clustering_params = {
 'number_of_clusters': 16,
 'cluster_centroids_init': CentroidInitialization.LINEAR
}

# Cluster a whole model
clustered_model = cluster_weights(model_wm_10, **clustering_params)
```

The cluster_weights API adds wrappers around our existing layers that help with learning the appropriate centroids, as seen below. These wrappers abstract away all the code that we wrote in the first exercise (and subsequently used in the following exercises so far) for computing the gradients, losses, and updating the centroids.

```
# Use smaller learning rate for fine-tuning clustered model
opt = tf.keras.optimizers.Adam(learning_rate=1e-5)

clustered_model.compile(
 loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
 optimizer=opt,
 metrics=['accuracy'])

clustered_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 input_1 (InputLayer)        [(None, 124, 129)]        0

 cluster_conv1d (ClusterWeig  (None, 124, 32)          74352
 hts)

 cluster_batch_normalization  (None, 124, 32)          128
  (ClusterWeights)

 cluster_conv1d_1 (ClusterWe  (None, 124, 32)          18480
 ights)

 cluster_batch_normalization  (None, 124, 32)          128
 _1 (ClusterWeights)

 cluster_max_pooling1d (Clus  (None, 31, 32)           0
 terWeights)

 . . .
 cluster_dropout_3 (ClusterW  (None, 128)              0
 eights)

 cluster_dense (ClusterWeigh  (None, 512)              131600
 ts)

 cluster_dropout_4 (ClusterW  (None, 512)              0
 eights)

 cluster_dense_1 (ClusterWei  (None, 12)               12316
 ghts)

 cluster_activation (Cluster  (None, 12)               0
 Weights)

=================================================================
Total params: 791,948
Trainable params: 396,524
Non-trainable params: 395,424
_________________________________________________________________
```

The clustered model needs to be fine-tuned for an epoch so that the centroids can be computed.

```
# Fine-tune model
clustered_model.fit(
 x_train,
 y_train,
 batch_size=128,
 epochs=1,
 validation_data=(x_test, y_test),
 # callbacks=[best_checkpoint_callback(model_name)],
 shuffle=True)
```

```
125/125 [==============================] - 7s 39ms/step - loss: 0.2502 - accuracy:
0.9685 - val_loss: 0.6145 - val_accuracy: 0.8636
<keras.callbacks.History at 0x7f3b23dae5d0>
```

Isn't this neat? We didn't have to do much here in terms of changing our model. The clustered model gets an accuracy similar to the baseline model (and we'll shortly verify the compression gains). However, given that we computed the clusters earlier from first principles, we hope that you also got an understanding of what is happening under the covers (or wrappers in this case).

We can now remove clustering variables and wrappers from our clustered model to generate the final model. We will also generate a TFLite model that we can use for inference on smartphones and other devices with lesser compute and memory resources.

```
import tempfile

final_model = tfmot.clustering.keras.strip_clustering(clustered_model)

_, clustered_keras_file = tempfile.mkstemp('.h5')
print('Saving clustered model to: ', clustered_keras_file)
tf.keras.models.save_model(final_model, clustered_keras_file,
                           include_optimizer=False)

clustered_tflite_file = '/tmp/clustered_speech.tflite'
converter = tf.lite.TFLiteConverter.from_keras_model(final_model)
tflite_clustered_model = converter.convert()
with open(clustered_tflite_file, 'wb') as f:
 f.write(tflite_clustered_model)
print('Saved clustered TFLite model to:', clustered_tflite_file)
```

Comparing the sizes of the exported models

Finally let's compare the sizes of the three models: the baseline Keras model, the clustered model, and the TFLite model. Unfortunately we do not (as of the time of writing this chapter) have the support to natively realize the size and latency gains from the clustering. However, we can use gzip to exploit the compressed representation generated by clustering.

```
def get_gzipped_model_size_kb(file):
 # It returns the size of the gzipped model in kb.
 import os
 import zipfile

 _, zipped_file = tempfile.mkstemp('.zip')
 with zipfile.ZipFile(zipped_file, 'w', compression=zipfile.ZIP_DEFLATED) as f:
   f.write(file)

 return os.path.getsize(zipped_file) / (1024.0)

original_model_size_kb = get_gzipped_model_size_kb(keras_file)
clustered_model_size_kb = get_gzipped_model_size_kb(clustered_keras_file)
clustered_tflite_model_size_kb = get_gzipped_model_size_kb(clustered_tflite_file)

print("Size of gzipped baseline Keras model: %.2f kb" % (original_model_size_kb))
print("Size of gzipped clustered Keras model: %.2f kb" % (clustered_model_size_kb))
print("Size of gzipped clustered TFlite model: %.2f kb" %
(clustered_tflite_model_size_kb))

Size of gzipped baseline Keras model: 1442.90 kb
Size of gzipped clustered Keras model: 263.02 kb
Size of gzipped clustered TFlite model: 256.71 kb
```

As mentioned, the sizes reported in the above snippet are computed after running gzip on the generated model files. The original model's size after gzip was 1442.9 KB. Applying clustering on the original model, followed by gzip the size of the final compressed model file was 263.02 KB. Converting to TFLite before gzip helped further reduce the size to 256.71 KB. All in all, we achieved a size reduction of **5.62x**. Refer to figure 5-8 for a comparison of the different model sizes.

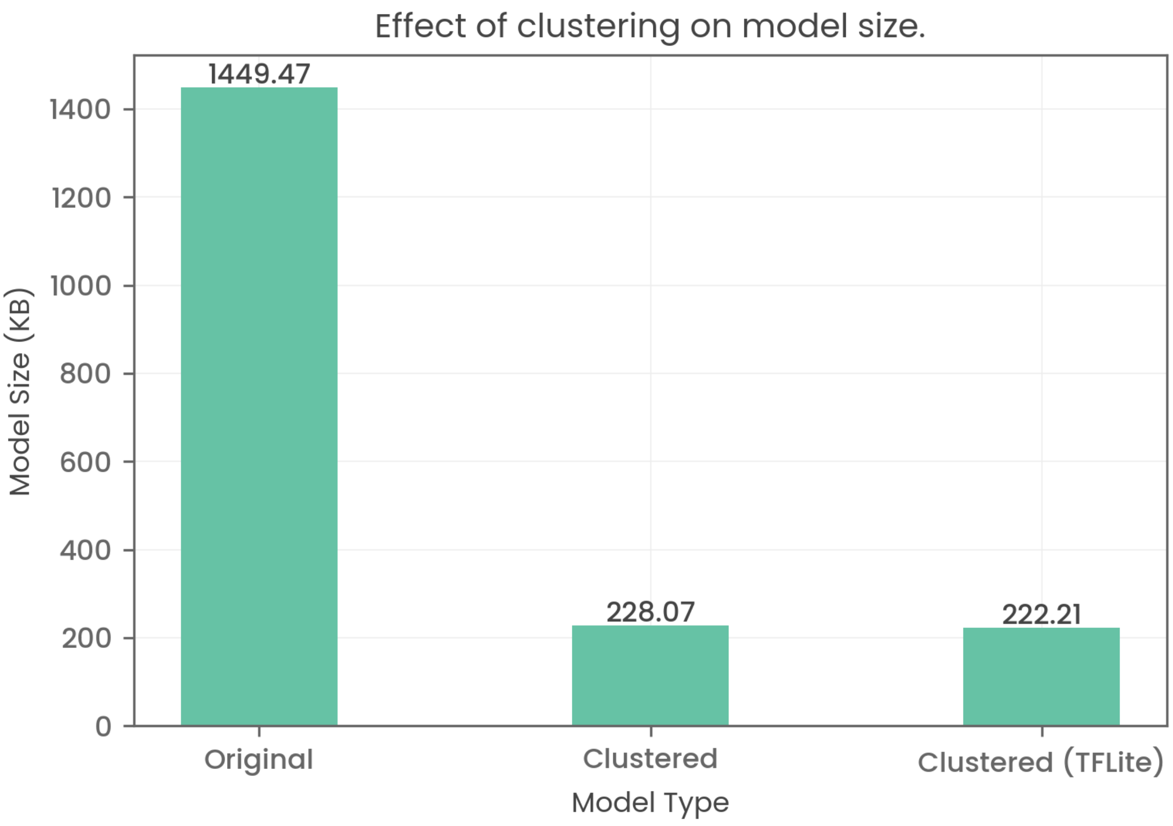


Figure 5-8: Comparison between the original unclustered model and the compressed clustered models.

Summary

Deep Learning models are often overfitted. For instance, there might be many connections between neurons where the weights might be infinitesimally small. Removing (pruning) them will likely not impact the overall model performance while sparsifying the given model. Training with some sparsity might actually improve model performance due to a regularizing effect[18] due to dropping spurious connections.

[18] https://en.wikipedia.org/wiki/Minimum_description_length

Apart from the most commonly used magnitude-based pruning, there are other heuristics for selecting the connections to prune based on their contribution to the loss / error. Similarly, it might be better to gradually increase the sparsity rate in the network over many epochs, while fine tuning the network to let it adapt to the sparsity. Unlike quantization, there isn't a general optimal algorithm for all scenarios, practitioners will have to empirically verify what works best for their specific model training setup.

Sparsity by itself helps with compressing the model size (footprint metric) since many connections can be removed without a noticeable impact on quality metrics. However, it is also possible to achieve latency improvements by pruning connections such that there is a certain structure to the sparsity. This helps hardware implementations leverage that structure for faster inference. For instance, NVIDIA GPUs rely on 2:4 sparsity, where exactly 2 out of 4 contiguous values in a matrix are 0 (effectively 50% sparsity). The intermediate model compiler rewrites a standard matrix multiplication operation to be performed using a compressed representation of the matrix with the non-zero values and their indices in the original matrix. Since the matrix is 50% sparse, the matrix multiplication can be performed in half the time.

With the advent of hardware support for sparsity and many industrial and academic use cases reporting significant improvements, we feel that sparsity will be one of the leading compression techniques used for model efficiency in the coming time.

Clustering is also a very powerful compression technique, yet implementing it is quite straightforward. We can achieve quality and footprint gains on top of quantization because clustering is a much more generic approach of allocating precision bits to the input range. In addition to that, the number of centroids allow us to scale the codebook size linearly and hence fine-tune the precision-size tradeoff that we need, whereas with quantization the precision and size changes by a factor of 2 between consecutive values of b (the number of bits allocated per value).

With that being said, first-class support for clustering in machine learning frameworks like Tensorflow and PyTorch is pending as of the time of writing this book. Mainly what is lacking is kernels that can efficiently leverage the compressed weight matrices on hardware so that you can actually see latency benefits, apart from the size benefits we demonstrated.

Another useful application for clustering (or any other compression technique for which there isn't native support) is embedding tables. Embedding tables are unique because the forward pass for them is a simple lookup in the table using the provided index. The embeddings themselves can be saved in an optimized format. Either the table in its entirety can be converted back to the original floating point tensor format either when loading the table into memory, or only the individual embeddings that need to be looked up can be converted during inference. Because this lookup operation is very simple, it is easy to create custom kernels for them, such as demonstrated here for Tensorflow and here for TFLite.

We would also encourage you to try other compression techniques such as Low-Rank Decomposition[19,20], which can be useful for compressing expensive dense fully-connected, convolutional layers and so on.

[19] X. Yu, T. Liu, X. Wang and D. Tao, "On Compressing Deep Models by Low Rank and Sparse Decomposition," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 67-76, doi: 10.1109/CVPR.2017.15.

[20] "Matrix Compression Operator." 17 July 2022, blog.tensorflow.org/2020/02/matrix-compression-operator-tensorflow.html.