A Survey and Compilation of Natural Language Processing Model Compression Techniques

Jorge Murillo and Lawrence Su

Mathematics for College of Creative Studies, University of California, Santa Barbara

Abstract

Recent advances in Deep Neural Networks (DNN's) over the last decade have allowed modern neural networks to be reliably deployed ”on the edge” in countless applications ranging from computer vision to natural language processing. Existing hardware is capable of running complex models with low latency, but a problem occurs when applications are scaled to require cheaper hardware with shallower memory resources or minimal latency. The goal of model compression is to take pre-trained, deep neural networks and reduce their size to allow them to be readily deployed in areas requiring ”on-device” inference such as self-driving vehicles and A.I. assistants. This paper covers recent advances in the field of model compression that allowed us to create a 100x smaller model in terms of memory storage while maintaining relatively stable accuracy.

1. Introduction

To motivate the need for model compression, consider self-driving cars. Self-driving cars have complex models taking large input vectors from hundreds of sensors detecting and analyzing moving objects, potential obstacles, and other vehicles, as well as being able to navigate the roads. Large deep neural networks are capable of performing these tasks but must be able to be computed in real time using the vehicle's processing power. It would not suffice to use the cloud to perform these inferences, since a vehicle could lose connection to the cloud depending on the terrain or region and cause an accident. Making these models smaller would allow for hardware to perform inference more quickly, and reduce space and energy requirements. With these benefits in mind, fields outside of vision techniques have demonstrated benefits from compressing neural network structures, such as the NLP domain. Consider GPT-3, OpenAI's newest revolutionary NLP model. This is an A.I. program that can write poetry and perform interactive storytelling. Given its revolutionary performance, the size of the model is huge, containing over 175 billion parameters. Consuming 5 million USD of compute resources, such large models are difficult to practically deploy. In order to combat its difficulty in deployment, we can explore model compression techniques

There are currently four main known ways of compressing neural networks. The first method is to train a smaller model using the primary larger model. This is known as Knowledge Distillation (KD). The second method, Quantization, involves reducing the number of bits representing each weight or activation outputs for each model. The third method is known as Pruning, and involves removing unnecessary weights or neurons in the model by converting the weights to zero or deleting a neuron. The final method is known as Low-rank approximation and is used to learn the ranks of each layer. The primary techniques covered in this paper are knowledge distillation and quantization for Feed Forward Neural Networks and Recurrent Neural Networks on Google's Go-Emotions dataset. For some context, the Go-Emotions dataset consists of 58,000 Reddit comments with 27 class labels, which we reduce to 7 class labels using Paul Ekman's grouping of emotions: anger, disgust, fear, joy, sadness, and surprise, and adding a final ”neutral” emotion whenever none of these emotions are expressed. In the following sections, I introduce some explanations when necessary.

2. A Simple Neural Network

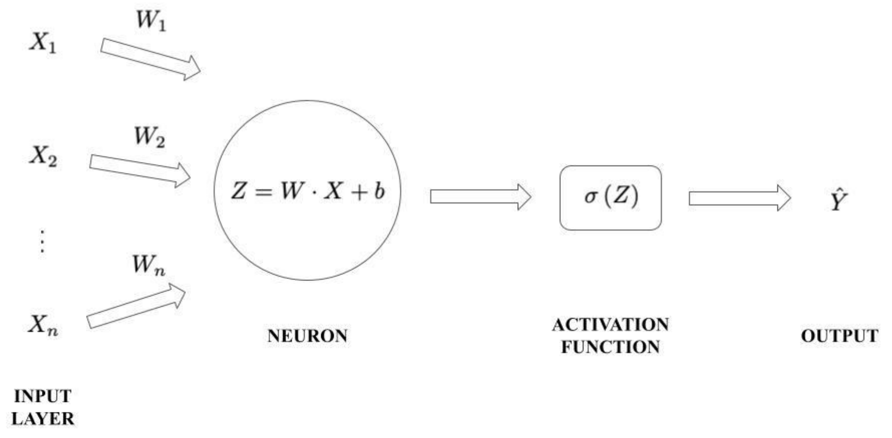

Figure 1: A neuron. Photo Credit to Yan Laschev.

If you are familiar with a neural network structure, then you can skip this section.

Compressing a neural network requires modifying its structure. In this section we give a brief overview of a simple neural network. A neural network consists of a set of neurons that are connected through multiple layers. The goal of a neural network is to learn the mapping of input data to output data using a given training dataset. Let's consider the simplest neural network structure, a perceptron, which is shown above. For simplicity, suppose we are trying to predict a reddit comment as having a neutral emotion as either 0 (False) or 1 (True) based off our input data. There are two broad steps in training a neural network, and in our case a perceptron: the forward-pass and backwards pass.

Step 1: Forward Pass

A forward pass involves passing the input data through the neuron, applying a non-linear activation function, and returning a class label. A perceptron consists of N real-valued inputs, one neuron, and one output. We define our weights $w = (w_1, w_2, \ldots, w_n) \in \mathbb{R}^n$, inputs $(x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ from our training dataset, and a bias $b \in \mathbb{R}$. Then a neuron is defined as

$$z = w \cdot x + b.$$

We apply a non-linear activation function *f*(*z*) onto *z*. Since oftentimes data is non-linear, applying this non-linear transformation allows a neural network to find non-linear relationships between random variables. It is able to better predict non-linear processes compared to models that apply linear transformations onto the data, such as linear regression. In the case of a perceptron, our non-linear activation function is a sigmoid function, i.e.

$$f(z) = \sigma(z) = \frac{1}{1+e^{-z}}.$$

One useful property of $\sigma(z)$ is the property to return a binary output. This allows us to interpret $\sigma(z)$ as a binary probability of our output being either 0 or 1. Usually we round $f(z)$ to get predicted class label 0 or 1. In addition, $\sigma(z)$ is a differentiable function with derivative $\sigma'(z) = \sigma(z)(1-\sigma(z))$, which makes it useful for the backwards pass.

Step 2: Backwards Pass

In this step, we optimize the neural network to predict comments in a training dataset (which are given a vector representation) with high accuracy. To do this, we need to use a loss function to determine how far our model's predicted labels are from the true-labels, and then use gradient descent (or some other optimization method) to update the weights *w* and bias *b* to minimize the loss function, and lead to higher accuracy. In the case of classification we use a cross-entropy function as our loss function.

Suppose we are given a reddit comment and want to determine if it's expressing a neutral emotion or not. We want to compute the loss function and be able to update our model to better predict the label for this (and upcoming) comments. We can assign a probability function *P*(*x*) where the assigned neutral-label for our reddit comment has a probability 1.0 of occurring (since it's already been assigned), and probability 0.0 for the other label. Cross entropy is a function that can be used for calculating the difference between two probability distributions. Let P be the class distributions $x \in X$ for our neutral label, and Q is our approximation for $P(x)$, which is

$$Q(x) = \begin{cases} \sigma(z), & x = 1 \\ 1-\sigma(z), & x = 0 \end{cases}$$

The cross entropy computed using this value x is defined as

$$H(P,Q) = -\sum_{x\in\{0,1\}} P(X=x)\cdot\log(Q(X=x)).$$

where log is the natural log. So in our case, our loss function is

$$\mathcal{L} = H(P,Q)$$

Once we have computed the loss function, we compute the error-gradient and update each weight bias accordingly using chain rule. For example, if we want to update each weight w_i, we first want to compute w_i, by computing

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial Q}\frac{\partial Q}{\partial z}\frac{\partial z}{\partial w_i}.\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial Q}\frac{\partial Q}{\partial z}\frac{\partial z}{\partial w_i}.$$

Since $\frac{\partial L}{\partial w}$ points to the direction where the loss function has the steepest ascent with respect to w, we update w to go opposite of it. Then,

$$w_i^{(new)} = w_i^{(old)} - \alpha \frac{\partial L}{\partial w_i},$$

where α is a hyper-parameter chosen by the programmer. Note that each pass through the training data set is called an epoch. Usually we train the model for multiple epochs, and then check the accuracy of our model on a separate validation dataset.

In general, the process of training a Feed Forward Neural Network (FFNN) involves some input data (which we denote as vector $x \in \mathbb{R}^n$, neurons in the hidden layers $l_1, l_2, ..., l_k \in \mathbb{R}^m$, and then an output $o \in \mathbb{R}^p$, where the dimensions n, k, m, p are dependent on both the data and the programmer. Each neuron n, k, m, p are dependent on both the data and the programmer. Each neuron z in layer l_i takes in any input from the prior layer and applies a non-linear activation function to interpret the data non-linearly. The forward and backwards pass steps are similar to that of the perceptron.

3. Base Feed Forward Neural Network Architecture

We construct our base FFNN architecture as follows

Input	(300-d vector)
Layer 1	(256-ReLU)
Layer 2	(128-ReLU)
Layer 3	(128-ReLU)
Output	(8-Sigmoid)

This architecture was chosen through using grid search, where the notation is ”n-activation”, n the number of neurons, activation the activation function. We train the model for 20 epochs on the training dataset, based on seeing convergence of the models scores at around 20 epochs for that dataset.

4. Metrics: F1 Score, Precision, Recall

In our dataset, we have 7 emotions that we seek to classify: fear (F), anger (A), joy (J), sadness (S), disgust(D), and surprise (R) and neutral (N), and a comment can have multiple labels at once. We count the emotions expressed for each comment and see the following distribution. There are 58,000 comments, each reviewed by around 3-5 people, to reduce the chances of a comment being mislabeled.

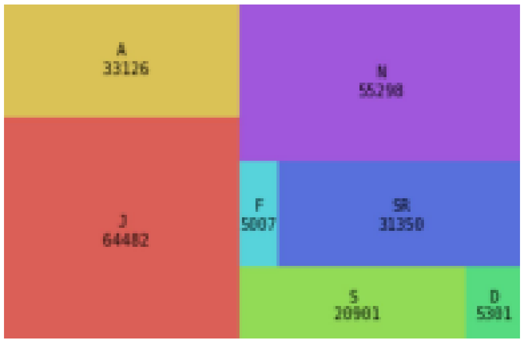

Figure 2: Emotion Distribution

As we can see, we have data imbalance. A few class labels are highly underrepresented (E.g. Disgust). Therefore, when training a model, viewing accuracy as a metric might be misleading, as the model may learn to not predict disgust or fear at all, and still attain a high accuracy over predicting all the labels. As such, we need to use metrics that can account for this imbalance. A good metric is the proportion of correct predictions over all predictions of a particular emotion. This is known as Precision, which is computed as

$$\text{Precision} = \frac{TP}{TP + FP},$$

where TP (True Positives) is the number of labels predicted to be True that were, FP (False Positives) are the number of labels predicted to be true that weren't false.

Another metric that we might want to consider is a metric that allows us to see the proportion of true positives among all true examples of a particular emotion, known as Recall.

$$\text{Recall} = \frac{TP}{FN + TP}.$$

If we want to balance the two, we use the F1-score, which is a type of harmonic mean, that promotes Precision and Recall being having similar scores/values. This is usually the preferred metric since it captures information about both precision and recall.

5. Binary Cross Entropy Loss

Traditional classification oftentimes performs multi-class classification, in which we predict a particular input has 1 of n classes. To do this, a real-value z_i(oftentimes called a logit) is formed from applying linear combinations to activations for the prior label considering each i in our class $1, 2, ..., n.$ To predict the probability of a particular class i, we compute the soft-max probability, which is defined as

$$Q(i) = \sum_{i=1}^{n} \frac{e^{z_i}}{\sum_{i=1}^{n} e^{z_i}}.$$

Then, the loss function is usually a cross-entropy loss function, where the cross entropy loss computed is defined as

$$\mathcal{L}_{CE} = - \sum_{x \in \{0,1,...,n\}} P(x) \cdot \log(Q(x)).$$

However, a sentence oftentimes evokes multiple emotions, so we cannot assume every input has exactly 1 of n-classes. Therefore our problem becomes a multi-label classification problem, where we assume the probability of each class occurring is independent of the others. We need to have a loss function that computes the loss for each 1 of n class label predictions.

To do this, we define

$$Q(i) = \sigma(z_i) = \frac{1}{1+e^{-z_i}},$$

and use the Binary Cross Entropy loss function, which is defined as

$$\mathcal{L}_{BCE} = -\frac{1}{n}\sum_{x\in\{0,1,\ldots,n\}} P(x)\cdot\log(Q(x)) + (1-P(x))\log(1-Q(x)).$$

This is equivalent to computing the average cross entropy for each particular label as a binary classification problem.

6. Word Embeddings

Word embeddings are vector representations of words and sentences. Similar words have similar representation and can be compared using cosine-similarity, you can restrict the dimension of the embedding space (which in our case is 300 dimensions), and capture the context of the word by how they've been trained. In particular we use the publicly available GloVe (Global Vectors for Word Representation), which uses global word-word co-occurrence statistics to construct a vector-representation of words [11]. To construct a vector-representation of sentences, we can take the average of the word vectors in every sentence. If any words are missing from our mapping, we ignore them . This method tends to capture the general meaning behind every sentence relatively well, but has the limitation that sentences such as "The cat bit the dog" have the same sentence representation as "The dog bit the cat". We end up using both word-vector representations and sentence-vector representations for our dataset.

7. Quantization

The most widespread method of model compression is quantization, the method of using reduced float point integers from the original model to represent its neurons. Neurons can be defined as

$$w\cdot x + b,$$

where w is the weight vector, x is the activation of neurons from the prior layer, and b is the bias. We change the notation to be $\vec{w}\cdot\vec{x}+b$

When we quantize a network, we are using a function Q to modify the network and use a reduced precision integer representation for the weights and/or activation. We consider how we can construct this representation for both the weights and activations for n bits.

Idea 1. Defining a quantization function Q for weights w.

Suppose we are trying to quantize weights in a given layer. One of the simplest functions used to quantize a network is known as uniform quantization. The quantization function is defined as

$$Q(r) = \mathrm{Int}(\frac{r}{\beta - \alpha} \cdot (2^n - 1)) + Z, r \in \mathbb{R}.$$

Here $[\alpha, \beta]$ is a bounded range that our weights $w \in \mathbb{R}$ are clipped into, and

$$Z = -\mathrm{round}(\frac{2^n - 1}{\beta - \alpha} \cdot \alpha) - 2^{n-1}$$

is an integer zero point number that can be used to center our quantized weight distribution around zero [13].

Note that a clipping function with min α and max β is defined as

$$\mathrm{clip}(\alpha, \beta, x) = \begin{cases} \alpha & x \leq \alpha \\ x & \alpha < x < \beta \\ \beta & x \geq \beta \end{cases}$$

If we assume the weights w are symmetrically distributed around zero, for computational efficiency we can let

Then our quantization function becomes

$$Q(r) = \mathrm{Int}(\frac{r}{2\beta_w} \cdot (2^n - 1)),$$

Which results in $Q(r)$ mapping each weight w to integers in the interval $[-2^{n-1}, 2^{n-1} - 1]$, and allows us to store the weights w efficiently as $Q(w)$. If we define $\vec{w} = (w_1, w_2, ...w_m)$, then $q_{\vec{w}} = (Q(w_1), Q(w_2), ..., Q(w_m)$. If we want to compute $\vec{w} \cdot \vec{x} + b$, we can dequantize $q_{\vec{w}}$ by performing the quantization functions operations in reverse, i.e.

$$\hat{w} = q_{\vec{w}} \cdot (\frac{\beta_w - \alpha}{2^n - 1}) = q_{\vec{w}} \cdot (\frac{2\beta_w}{2^n - 1}) \approx \vec{w},$$

and compute $\hat{w} \cdot \vec{x} + b$. This is the same method implemented by TensorFlow.

Idea 2 : Defining a quantization for the activations $\vec{x}$:$\vec{x}$:

Sigmoid and Relu functions are some of the most commonly used activation functions in a DNN,

where the sigmoid function is defined as $\sigma(r) = \frac{1}{1 + e^{-r}}$, and the relu function is defined as

$$\mathrm{relu}(r) = \begin{cases} r, & r \geq 0 \\ 0, & r \leq 0 \end{cases}.$$

We can see for both activation functions, for any r, the resulting activation $f(r) = x_i \in R$ will be greater than zero, hence the activation's distribution will not be symmetrically distributed around the origin. So defining $\beta_x = \max |x_i|, \alpha = -\beta_x$, will result in information loss as we constrict activations to the interval

$[0, 2^{n-1}-1] \subset [-2^{n-1}, 2^{n-1}-1]$, forcing us to use asymmetric quantization.

If we let $\alpha = 0, \beta = \max(x), Z = -2^{n-1}$, the uniform quantization function can be defined as

$$Q(x) = \text{Int}(\frac{x}{\beta_x} \cdot (2^n - 1)) + Z, x \in \mathbb{R},$$

and the resulting outputs for $Q(x)$ will be mapped to integers in the interval

$$[-2^{n-1}, 2^{n-1}-1].$$

Suppose q_x is our quantized vector representation of $\vec{x} = (x_1, x_2, .., x_n)$, and we want to approximate

$$\vec{w} \cdot \vec{x} + b.$$

We can dequantize the activation by using

$$\hat{x} = (q_x - z_x) \cdot (\frac{\beta_x}{2^n - 1}) \approx \vec{x},$$

to then compute

$$w \cdot \hat{x} + b.$$

Idea 3: Combining both methods.

However, usually if we're quantizing the activations, we are also quantizing the weights, meaning we are computing $\hat{w} \cdot \hat{x} + b$. We can then write $\hat{w} \cdot \hat{x} + b$ as

$$(q_w \cdot (q_x - Z_x)(\frac{2\beta_w}{2^n - 1})((\frac{\beta_x}{2^n - 1})) + b = c(q_w \cdot q_x - q_w \cdot Z_x) + b,$$

$$c = (\frac{2\beta_w}{2^n - 1})((\frac{\beta_x}{2^n - 1})).$$

We can store $c, q_w \cdot Z_x$ offline and just apply it after every matrix multiplication step for this layer. This implies that once we've quantized, $\vec{w}, \vec{x}$, we can perform integer multiplication [6]. (which is more efficient for hardware to compute than floating point multiplication), and then re-scale the final output to a floating point number.

Now that we've discussed the above functions, we can discuss their implementations. There are three quantization methods : Dynamic Quantization, Post-Training Static Quantization, and Quantization Aware Training [3] .

Dynamic quantization is simply when we quantize every weight $\vec{w}$. If we want to quantize $\vec{w}$ and $\vec{x}$ for each layer without retraining the model, this is called Post-Training Static Quantization. This usually requires some calibration data that passes through the network in order to understand the distributions of the activations. In Post Training Static Quantization, we are quantizing $\vec{w}, \vec{x}$ and not necessarily optimizing $\hat{w}, \hat{x}$, to best fit the data.

If we want to quantize $\vec{w}, \vec{x}$ optimally into $\hat{w}, \hat{x}$,the method we use is called Quantization Aware Training

(QAT).

In QAT, we tend to simulate quantized weights, quantized activations (which will be stored in fp-32) in the forward pass, and then perform backpropagation on the values for the full precision model. After training for a few epochs, we reduce the model to quantized weights, and activations to the desired bit format.

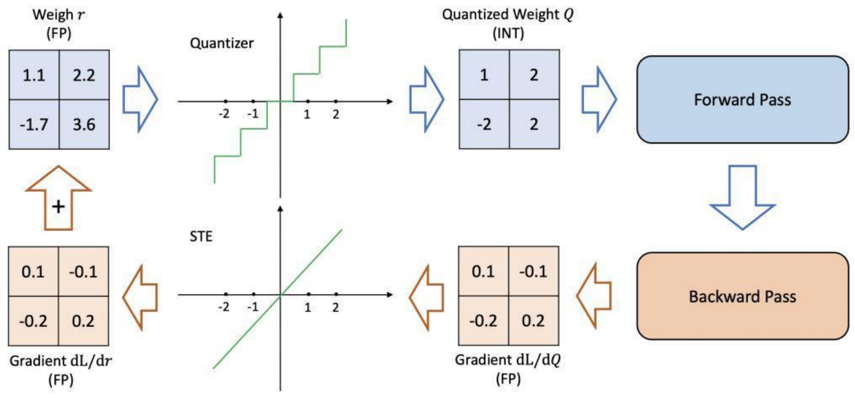

Figure 3: QAT, taken from [3]

In order to update the original models weights, using the simulated model's outputs, we have to compute

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial \hat{w}} \frac{\partial \hat{w}}{w}.$$

But applying the Int() function in the quantization function *Q*(*x*) (which converts a number from floating point to an integer), causes *Q*(*x*) having a derivative of 0 almost everywhere, meaning $\hat{w}_2$has a derivative of zero almost everywhere. To overcome this, we can use the straight through estimator (*ST E*) , which defines

$$\frac{\partial \hat{w}}{\partial w} = 1.$$

For us to then use

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\partial \mathcal{L}}{\partial \hat{w}}.$$

STE has been shown to empirically work, and has foundations from being an unbiased estimator of stochastic-binary neurons [1].

8. Clustering Quantization Aware Training (CQAT)

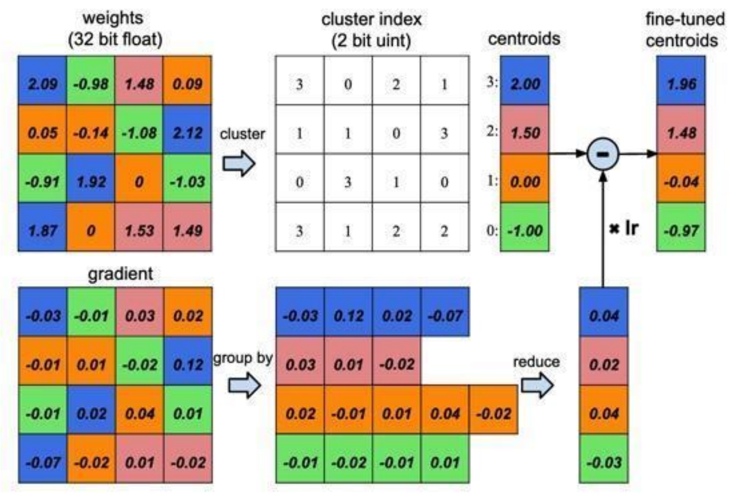

Figure 4: CQAT implementation on the weights, figure taken from [5].

One idea to compress the model further is to take the model and cluster its weights $w_1, w_2, ..., w_n$ in a given layer into k-groups, and then map each weight to their centroid $c_1, c_2, ..., c_n$, or the center of each approximate group [5]. We do this so as to minimize the within-cluster sum of squares (*W CCS*):

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

Each c_i is initialized with equal distance between the minimum and maximum weights in a layer, as this initialization has been shown to be highly effective (linear initialization). Afterwards, We perform QAT in similar steps, except we preserve the weight clusters.To store the weights, we store their integer index as well as their centroid, and then apply a lookup table for matrix multiplication. To update the weights, we compute the loss for each centroid, and then update the centroids (and therefore the associated weights for each centroid). We compute

$$\frac{\partial \mathcal{L}}{\partial c_k} = \sum_i \frac{\partial L}{\partial w_i} \frac{\partial w_i}{\partial c_k} = \sum_i \frac{\partial L}{\partial w_{i,k}} 1(I_i = k),$$

where 1 is the indicator function of whether or not index I_i of weight w_i is equal to k. Note that the activations are quantized in the standard manner. This method is immediately available on TensorFlow, and we implement it using a feed forward neural network with the following specifications, comparing each clustered model with the number of clusters.

Num. Clusters	F1	Precision	Recall
16	0.4753	0.5913	0.3965
32	0.477	0.5853	0.401
64*	0.486	0.583	0.417
128	0.486	0.581	0.418
Base	0.463	0.630	0.366

8.1 Comparing CQAT vs base FFNN vs QAT

Model	F1	Precision	Recall	Model size
CQAT FFNN, k = 64	0.486	0.583	0.41	91kb
Uniform QAT	0.478	0.613	0.391	49 kb
Base FFNN	0.463	0.630	0.366	589 kb

We can see that CQAT outperforms the base RNN-Model in terms of the f1 score, while still being smaller than the student model, and has more predictive power than the QAT model. Although this behavior is not usually the case, it might be the case that by simplifying the model, we make it more robust against noise, since parallels in the human brain have suggests that the human brain stores information in a quantized form, with a rational being that discrete signal representations can be more robust to low-level noise, as well allowing for higher efficiency under limited resources [3] . One con of CQAT and other non-uniform quantization methods is that they require a Lookup Table (LUT) in order to implement consistent, where we reference each weight index to it's dequantized value, in order to perform multiplication with each weight and activation, then sum them all up. Compare that to QAT methods, where we can multiply each integer quantized weight and quantized activations together in a neuron, sum them all up, then multiply the two sums by a scalar. Note that we were not able to compare memory speedups or reductions in memory usage from quantization using Pytorch or Tensorflow, so we only compare storage size. If we wanted to see these gains, we would have used edge devices, which we did not buy.

9 Additive Powers of Two (APOT) Quantization.

9.1 Formula

Most weights distributions in a neural network are normally distributed, with a mean centered near zero [8]. Now, uniform quantization distributes quantized values that when de-quantized, are spread uniformly across an interval. While efficient, this method means less common weights are roughly the same distance from their quantized counterparts as weights that are more common. To remedy that, non-uniform quantization methods have been explored that assign more quantization values near the mean, and hence closer approximation to the dequantized values on average, than other methods.

In particular, we explored additive powers of two (APOT)[8], which is defined as

$$\lambda \sum_{i=1}^{n} p_i, \text{ where } p_i \in \{0, \frac{1}{2^i}, \frac{1}{2^{i+n}}, \ldots, \frac{1}{2^{i+(2^k-2)n}}\},$$

where λ is a scaling factor to ensure the maximal sum is of length 1, k is the base bin-width, and

$$n = \frac{b}{k},$$

where b is the number of bits. We compute computing all possible

$$\lambda \sum_{i=1}^{n} p_i,$$

of which there are

$$(2^k)^n = 2^b$$

total possible numbers, which are the desired number of bits. The definition of n restricts k to be a factor of b, yet allows it to be a hyper-parameter that can be tuned to vary the degree of precision.

For post training quantization and quantization aware training, in the case of Sigmoid and Relu functions used to activate neurons, their outputs are non-negative so we can use standard additive powers of two. For the weights however (which are negative), We would first detach the sign, quantize the weights using $\hat{b} = b - 1$, then re-attach the sign to get a b total bit representation of the weights. You can assign an integer representation of the number of bits by sorting the powers of two from smallest to largest, then you take the index of the particular sum of powers of two which weight is closest to.

9.2 Approximating $\hat{w}$ using APOT .

In order to use powers of two, we need to re-scale our weights, activation outputs to be within the interval $[0, 1]$. Let's consider weights w in the interval $[a, b]$. (Note that in my project we applied this method when we moved on to explore compression methods on RNN's and not FFNN's). We want to apply APOT quantization, using function q with parameters k, b. We may also need to clip weights to be within a desired interval $(-\alpha, \alpha)$. Then we map the weights into the interval $(0, 1)$ in order to use APOT quantization. This is equivalent to dividing the weights by alpha, then clipping the weights into the interval

$(0,1)$. So to implement APOT quantization, we perform the following steps:

Step 1: divide w by α,and store the absolute value of

$$w_1 = \frac{w}{\alpha}.$$

Step 2: clip $\hat{w}_1$ to be in the interval $(-1,1)$

$$w_2 = \text{clip}(-1,1,w_1)$$

Now w has been mapped to the interval $(-1,1)$.

Step 3: apply APOT quantization onto *w*$_2$.

Here we map *w* to a sum of powers of two.

$$\hat{w}_2 = \text{APOT}(|w_2|, k, b-1) \cdot \text{sign}(w),$$

where $APOT(|w_2|, k, b-1)$ maps $|w_2|$, to the nearest sum of powers of two. Now w has been mapped to the interval $(-1,1)$. If we wanted to store the weights for later use efficiently, we would store the index of the power of two that w has been mapped to, multiplied by the sign of w.

Step 4: Get the quantized approximation of *w.*

We do this multiplying $\hat{w}_2$ by α:

$$\hat{w} = \alpha \cdot \hat{w}_2.$$

We are now mapping w into the interval $(-\alpha, \alpha)$.

9.3 Optimizing *α*:

Rather than letting $\alpha = \max|w|$, for our weights w in a layer, we may want to optimize our clipping value α [10] (Note that we did not implement this method in the final model due to some difficulties in implementing it, but cover it for the sake of completeness, since it's something the authors did in the paper). In order to do so, we will need to compute $\frac{\partial L}{\partial \alpha}$. We only need to compute $\frac{\partial \hat{w}}{\partial \alpha}$, since chain rule tells us that

$$\frac{\partial L}{\partial \alpha} = \frac{\partial L}{\partial \hat{w}} \frac{\partial \hat{w}}{\partial \alpha}.$$

Note that

$$\frac{\partial \hat{w}}{\partial \alpha} = \frac{\partial}{\partial \alpha}(a \cdot \hat{w}_2) = \hat{w}_2 + a\frac{\partial}{\partial \alpha}\hat{w}_2,$$

where

$$\frac{\partial}{\partial \alpha}\hat{w}_2 = \frac{\partial \hat{w}_2}{\partial w_2}\frac{\partial w_2}{\partial \alpha} \approx \frac{\partial w_2}{\partial \alpha}$$

by using STE, and

$$\frac{\partial w_2}{\partial \alpha} = \begin{cases} 0 & \text{if } |\frac{w}{\alpha}| > 1 \\ -\frac{w}{\alpha^2} & \text{if } |\frac{w}{\alpha}| \leq 1 \end{cases}$$

$$= \begin{cases} 0 & \text{if } |w| > \alpha \\ -\frac{w_2}{\alpha} & \text{if } |w| \leq \alpha \end{cases}$$

Since

$$w_2 = \text{clip}(-1, 1, \frac{w}{\alpha}).$$

Therefore

$$\frac{\partial \hat{w}}{\partial \alpha} = \hat{w}_2 + a\frac{\partial}{\partial \alpha}\hat{w}_2$$

$$= \begin{cases} \text{sign}(w) + 0 & \text{if } |w| > \alpha, \\ \hat{w}_2 - \alpha\frac{w_2}{\alpha} & \text{if } |w| \leq \alpha \end{cases},$$

Where we used the fact that

$$\hat{w}_2 = \text{sign}(w) \text{ if } \left|\frac{w}{\alpha}\right| > 1.$$

10. Knowledge Distillation (KD)

The core idea of this method is to take a large model, and then use information about the model's outputs/layers to train a smaller model.

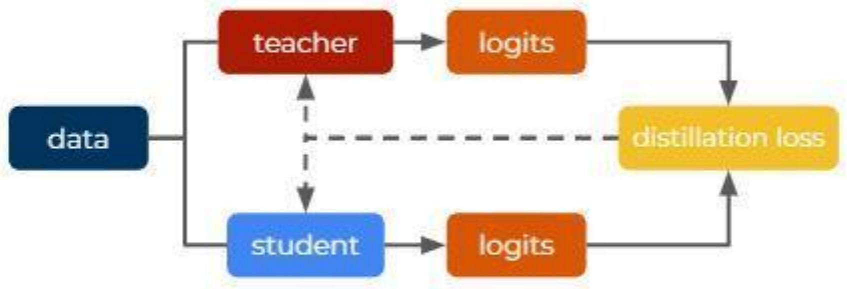

Figure 5: A KD process where the student and teacher model co-teach one another.

One of the most common methods of knowledge distillation is known as Response Based Knowledge Distillation, where the goal is to mimic the last layer (i.e. logits/predictions) of the neural network model [4]. KL-Divergence is a commonly used function that's applied onto the loss function to measure and minimize the difference in distribution.

10.1 A standard response based knowledge distillation:

Step 1: Perform a forward pass on both the student and teacher model

Step 2: Compute loss comparing the difference in teachers predictions to the students predictions: We can use KL divergence, where KL-Divergence between two distributions A, B is defined as

$$KL(A||B) = \sum_i p_A(v_i)\log(p_A(v_i)) - p_A(v_i)\log(p_B(v_i)).$$

KL-divergence measures the relative divergence in distributions between A,B, unlike cross entropy, which compares the total divergence in distribution. $p_A(v_i)$, $p_b(v_i)$ can be defined as the final sigmoid activations [9], reflecting the probability of each emotion for a given reddit comment. In standard classification, we are given hard-class labels of the form 0*,* 1. However, in response-based knowledge distillation, the smaller model will try to mimic the teacher's probability outputs for each emotion. and since the student model is given more information by the teacher model when it sees the probabilities versus hard-class predictions, we can train smaller models to converge towards an optimal solution more efficiently using a larger teacher model, than simply viewing the hard class labels. Furthermore, we can allow the student model to learn to both mimic the teacher and learning the task by combining KL-Divergence with BCE, meaning the loss of the student model becomes
$\mathcal{L}_{\text{student}} = \mathcal{L}^S_{BCE} + KL(z^S||z^T).$

Step 3: Perform back propagation on the student distribution.

11. Recurrent Neural Networks (RNN)

One limitation that Feed Forward Neural Networks have is their inability to record sequential information. In order to predict emotions using a Feed Forward Neural Network, we needed to extract each word vector from a comment, then construct a sentence vector by averaging these vectors as an input. Our model may have better predictive power if we gave the model the word-vectors from a sentence in their sequential order. We then could capture the sentence semantics, and the order of a word to reduce ambiguity. We can do this by using a Recurrent Neural Network (RNN), and in particular, a Gated Recurrent Unit RNN (GRU RNN).

RNN’s in general take in a variable sequence of data, say

$$x_1, x_2, ..., x_t, ...x_n$$

and for each *t,* compute a hidden state (h_t) that captures information from x_t, x_{t-1}, ...x_1, by using non-linear transformations. This allows for the model to have some ”memory” about prior x_t, and use that to predict $\hat{y}$

11.1 GRU-RNN

GRU-RNNs resemble short-term and long term memory. To accomplish this task, GRU RNNs have a reset gate, and an update gate. GRU-RNNs use the following four variables (Note that for $t = 0$, $h_0 = 0$.):

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r),$$

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z),$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h),$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t.$$

W_s, U_s are matrices for s in $\{x, r, h\}$, $\odot$ represents the hadamard product, and σ, tanh are implemented element wise on their vector inputs. r_t is known as the reset gate. Interpreted as the component responsible for the model's short term memory, it is a non-linear function used to determine how much of h_{t-1} to consider in the new candidate state $\tilde{h}_t$. z_t is known as the update gate, where it constructs the new hidden state h_t by computing proportions of which to consider the prior hidden state h_{t-1} and the new candidate hidden state $\hat{h}_t$. We can interpret this step as storing information or memory as long term. We would compute each h_t and return h_n as an input for the remaining layers in our RNN.

11.2 Base RNN Model Architecture

We construct a large RNN model with a high F1 score. We choose the following architecture (found through grid search).

Input	Sequence of 300-d vectors
Layer 1	(256-GRU)
Layer 2	(256-GRU)
Layer 3	(128-Linear)
Output	(8-Sigmoid)

Indeed, this model has a far higher F1 score than FFNN:

Model	F1	Precision	Recall
Base RNN	0.566	0.574	0.559
Base FFNN	0.463	0.630	0.366

Note: We chose to use a GRU model over the more popular LSTM model because it converges more quickly. We train it for 10 epochs, seeing convergence at around 10 epochs.

11.3 Limitations in quantizing the GRU-RNN.

As we can see, Recurrent Neural Networks can take in variable length inputs, and apply activation functions for each input. This makes applying QAT relatively difficult, since to quantize the activations using APOT or symmetric uniform quantization , we would at minimum have to approximate

$$\hat{r}_t, \hat{z}_t, \hat{\tilde{h}}_t, \hat{h}_t,$$

for each *t* in our dataset, where the *t* is variable.

In applications, computing these approximations may not be feasible for our t. For example, in our model and go emotions dataset, if we were to implement symmetric uniform QAT, we would need to compute α_t, β_t for each of the four variables , and we'd only be able to do so for up to $t = 32$, since the maximum length sentence in our dataset had 32 words. If we returned a sentence with word count larger than this one, we would not be able to accurately approximate α_t, β_t for activations in the remaining characters effectively by simply computing

$$||r_t||_\infty, ||z_t||_\infty, ||\tilde{h}_t||_\infty, ||h_t||_\infty \text{ for } t \geq 32,$$

due to a small/non-existent sample size. So we limit ourselves to dynamic quantization, where we quantize W_s, U_s.

11.4 Applying Knowledge Distillation + APOT quantization onto the GRU-RNN.

Applying KD onto the GRU-RNN requires us to only modify the student and teacher loss functions. Inspired by the QAT-KD method [7], We do this in two steps

Step 1: In order to teach the student model, we use the loss function

$$\mathcal{L}_{KD}^S = \mathcal{L}_{BCE}^S + KL(z^T || z^S),$$

where z^T, z^S are logits, and which we apply sigmoid functions to compute $p(z^T)$, $p(z^S)$. For the teacher to better adapt to the student distribution, we use the loss function

$$\mathcal{L}_{KD}^T = \mathcal{L}_{BCE}^T + KL(z^S || z^T)$$

until the teacher model saturates. We perform the forward pass and backwards pass accordingly for both the student and teacher model for a few epochs.

Step 2: Now that the teacher model has saturated, we stop updating the loss function for the teacher, and only update the student, meaning we are only computing

$$\mathcal{L}_{KD}^S = \mathcal{L}_{BCE}^S + KL(z^T || z^S).$$

We perform forward pass and backwards pass for the student model for a few epochs until the models converge. Step 3 Finally, we apply APOT quantization to make the model smaller.

11.5 Student Model Architecture

We choose for our student model neural network the following architecture.

Input	Sequence of 300-d vectors
Layer 1	(8-GRU)
Layer 2	(8-Linear)
Output	(8-Sigmoid)

This architecture allows us to show the benefits of knowledge distillation. We select another larger GRU-RNN Model with a decent F1 score to highlight the power of knowledge distillation, knowledge distillation + APOT . It's architecture is below

Input	Sequence of 300-d vectors
Layer 1	(128-GRU)
Layer 2	(128-GRU)
Layer 3	(128-Linear)
Output	(8-Sigmoid)

We apply 5-bit APOT quantization on the weights, detaching the sign of every weight to apply 4 bit quantization with $k = 2$, then reattaching the sign to have a 5-bit representation. We train the student model, and a model with the same parameters for 30 epochs on training data (though the student model starts seeing convergence in 10 or so epochs), then check their scores on a separate validation set-data.

11.6 Model scores

Model	F1	Precision	Recall	Model size
Teacher Model (Base RNN)	0.566	0.574	0.559	3.2 mb
Other GRU Model	0.54	0.63	0.469	1.2 mb
Student Model, KD + 5 bit APOT*	0.543	0.579	0.511	7 kb
Student Model, KD	0.551	0.583	0.522	32 kb
Model with Student param., no KD	0.474	0.703	0.357	32 kb

As we can see, for our particular dataset, using KD and APOT quantization, we were able to construct models that can attain an F1 score just as high (if not higher) than a larger GRU models while being 37-171x smaller than that model, and in our case, 100-450x smaller than the teacher model.

12. Conclusion, Next Steps

We can see that model compression methods can shrink the size of neural networks tremendously. In particular, we took a look at how non-uniform quantization methods such as clustering, APOT quantization worked, and explored their impact the scores of our model. In addition, we took a look at how knowledge distillation can be used to train our model. The ability for a student model to preserve much of the predictive power of it's teacher model has a basis in the lottery ticket hypothesis [12].The lottery ticket hypothesis states that ”A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations” [2]. We saw something similar for f1-scores when training the student model and teacher model, in that they converge at around 10 epochs. In addition, [2] suggests that we need large, over parameterized neural network models in order to find the optimal sub-architecture, since they are easier to train than smaller neural networks.

In terms of next steps, it's important to compare the benefits of APOT quantization with respect to uniform quantization on an RNN, so it would be worth our time to construct a dynamically quantized model using symmetric uniform quantization or some other quantization method. In addition, it would be worth exploring later on if time permits, quantizing the activations for sequences of fixed length on a GRU-RNN, and then using QAT-KD to quantize the student model. Furthermore, one limitation of APOT quantization is that it needs for k to be a factor of its bits, meaning for smaller weight quantization whose bits are assigned $b = 4, 3, 2, 1$, k is equal to either 1 or b–1. APOT quantization on our RNN will be more optimized when optimizing α, where we can perhaps modify our approximation $\frac{\partial \hat{w}_2}{\partial w_2}$ by using a Generalized STE computed in [10].

To view the python notebook where we applied the APOT quantization and knowledge distillation onto the GRU-RNN, click here. To view the python notebook where we applied the CQAT and other quantization methods onto the FFNN, click here.

References

[1] Yoshua Bengio, Nicholas L´eonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[2] Jonathan Frankle and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635* (2018).

[3] Amir Gholami et al. "A survey of quantization methods for efficient neural network inference". In: *arXiv preprint arXiv:2103.13630* (2021).

[4] Jianping Gou et al. "Knowledge distillation: A survey". In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819.

[5] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[6] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[7] Jangho Kim et al. "Qkd: Quantization-aware knowledge distillation". In: *arXiv preprint arXiv:1911.12491* (2019).

[8] Yuhang Li, Xin Dong, and Wei Wang. "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks". In: *arXiv preprint arXiv:1909.13144* (2019).

[9] Yongcheng Liu et al. "Multi-label image classification via knowledge distillation from weakly-supervised detection". In: *Proceedings of the 26th ACM international conference on Multimedia*. 2018, pp. 700–708.

[10] Zechun Liu et al. "Nonuniform-to-Uniform Quantization: Towards Accurate Quantization via Generalized Straight Through Estimation". In: *ArXiv* abs/2111.14826 (2021).

[11] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *EMNLP*. 2014.

[12] Arman Rahbar et al. "On the unreasonable effectiveness of knowledge distillation: Analysis in the kernel regime". In: *arXiv preprint arXiv:2003.13438* (2020).

[13] Hao Wu et al. “Integer quantization for deep learning inference: Principles and empirical evaluation”. In: *arXiv preprint arXiv:2004.09602* (2020).