Jai Pise

An Investigation of the Compression of Neural Networks for Tiny Machine Learning

Research Question: How much can a traditional Deep Learning model be compressed to be more memory-efficient for the field of Tiny Machine Learning?

EE Subject: Computer Science

3954 words*

*Excluding in-text citations, tables, and diagrams

Table Of Contents

1. Introduction and Background Information

Artificial Intelligence (AI), the field of intelligent machines, was first introduced as a concept in the 1950s (Smith et al.). However, it is only since 2010 that this field in Computer Science has seen a rise in popularity ("History of Artificial"). Specifically, attention has turned towards a subset of AI - Machine Learning - where statistical methods are used to find underlying patterns in data (Tamir). These patterns are contained in a model, which can then be used to make inferences, or predictions, about previously-unseen data. Hence, a Machine Learning model learns to "predict things based on past observations," rather than having a programmer explicitly code the rules, as is the case in traditional algorithmic approaches (Warden and Situnayake 12).

The most widely-used method within Machine Learning is Deep Learning, where Artificial Neural Networks, inspired by the human brain, are trained to "model the relationships between … inputs and outputs" (Warden and Situnayake 13). The neurons that form these Neural Networks (NN) can identify patterns in the data presented to them. Neurons store these patterns in parameters, contained in arrays of numbers, which make up the trained model. The NN can then make inferences on new input data by referencing these trained parameters. Different architectures, or arrangements, of these neurons excel at different tasks. For example, the optimal architecture of neurons for developing an Image Classification model is different from the architecture used with audio. NNs have become popular recently and are used in various applications because they can harness modern computers' ever-increasing memory and computational speed to perform several statistical calculations. This allows the models to find patterns in vast datasets (Warden and Situnayake 14).

However, powerful NNs require high amounts of memory and computational power because of the high number of neurons and parameters used to learn from data. Once the NN has learned from the data, it stores all the patterns in the trained parameters. Generally, more parameters result in higher model accuracy, but also require more memory and consume more power (Warden and Situnayake 15). As a result, a traditional NN model - one that has not been compressed by one of the techniques described in this paper - cannot directly be used for inference on edge devices. Edge devices are devices used for real-world inference, such as smartphones or microcontrollers, which are computer systems with limited memory capabilities ("Types and Applications"). Instead, the traditional NN model's inferences must be made in data centers - buildings containing several computers (with immense memory capabilities) built specifically for Machine Learning.

Consequently, if developers want to implement NNs on resource-constrained edge devices with low memory capabilities and limited power sources, they must reference the trained model from a data center, which can have several drawbacks. For instance, there will be a high latency in model predictions, meaning that there will be a time lag for the inference to arrive to the user as data is transferred between the device and the data center. There are also security risks associated with this form of inference because malicious actors could intercept the data sent between the edge device and the data center.

Tiny Machine Learning (TinyML) is a burgeoning field that addresses these problems of memory, computational power, latency, and security by altering traditional NN architectures to allow resource-constrained edge devices, specifically microcontrollers, to perform inference at the edge, without access to models stored in data centers, and with minimal memory usage and power consumption ("tinyTalks ANZ").

The difference between traditional Machine Learning practices and TinyML becomes clear when considering the memory and power available in data centers compared to microcontrollers. Microcontrollers have memory on the order of kilobytes and consume milliwatts of energy ("tinyTalks ANZ"). In contrast, an average Google data center handles petabytes of data and consumes terawatts of energy each day (Bawden; Canini).

By evaluating different techniques to compress traditional NN architectures and decrease the amount of memory such models are using, this paper will explore the Research Question: **"How much can a traditional Deep Learning model be compressed to be more memory-efficient for the field of Tiny Machine Learning?"** To answer this question, two different methods of decreasing the memory usage of NN models will be implemented on a Convolutional Neural Network, a specialized NN architecture used for computer vision, where a model is trained to classify images (Valueva et al. 1). These two methods have been selected after careful consideration of their validity by examining their acceptance by Machine Learning experts. After implementing these compression methods, the reduction in memory usage and the change in the model's accuracy will be analyzed to better understand the effectiveness of these techniques in compressing the model, and also the tradeoff between memory and accuracy.

Although this paper focuses on optimizing the memory usage of a NN, its energy consumption is highly reliant on its memory usage. Hence, decreasing the model size likely corresponds to lower power consumption and higher energy-efficiency because fewer computational operations are performed with a smaller model. This topic and research question are worthy choices for an Extended Essay in Computer Science because compressing NNs enables a diverse range of applications of TinyML, such as Animal Conservation, Disaster Prevention, and Industrial Monitoring ("tinyTalks ANZ"). This means

that it is essential to understand the basis of TinyML as it grows more popular in real-world applications and its influence on our daily lives grows. **Hence by examining and quantifying different methods of compressing Neural Networks, traditional Deep Learning models can be compressed by anywhere from a factor of 10 to a factor of 45 to be more memory-efficient for the field of Tiny Machine Learning.**

2. Deep Compression

The first method of compressing Neural Networks that will be analyzed is Deep Compression (Han et al. 1). This method was developed by Han et al. - experts in Deep Learning - and was published at the 2016 International Conference on Learning Representations, where its validity was peer-reviewed by Machine Learning experts. Hence, Deep Compression will be analyzed in the context of the most widely-used Neural Network architecture, a Deep Neural Network (DNN), described below.

2.1 Pruning

A DNN consists of a set of distinct layers, each containing neurons. All these neurons are interconnected, as depicted in Figure 1.

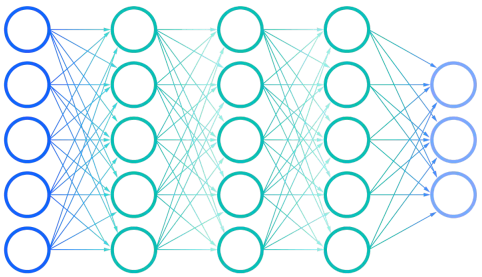

Figure 1: A densely connected Deep Neural Network (IBM Cloud Education)

These inter-neuron connections represent the model's weights, which are simply parameters, in the case of a DNN. Even relatively small DNNs contain several weights. However, some

weights have a negligible effect on the final prediction of the model. In essence, DNNs are "over-parameterized," leading to a "significant redundancy [in] Deep Learning models" and a "waste of both computation and memory" (Denil et al. 1). Hence, removing the redundant connections between neurons can significantly decrease the model's memory usage. There are several methods for eliminating redundant weights; however, most cause a significant loss in the model's accuracy. Network Pruning minimizes the accuracy loss while eliminating the maximum number of connections.

2.2 Quantization

The weights in a DNN are generally represented in the float32 variable type (uses 32 bits) - the default in most systems. The storage requirements of the model can be decreased, along with its inference time, by changing the variable type of the weights in a DNN to int8 (uses 8 bits). Quantization reduces the model size "by lowering the number of bits [that] represent [the weights]" (Paupamah et al. 1). This technique results in the weights being less precise because storing decimal values with 32 bits is naturally more precise than 8 bits, possibly leading to a lower accuracy. Nevertheless, the decrease in the model size and latency greatly outweighs the relatively small impact on the model's accuracy.

There are two types of Quantization, each of which manipulates the variable type to make the model more memory-efficient. Post Training Quantization (PTQ) is performed after training and conditions the model to perform inference in the int8 format, rather than in the default float32 format (Warden and Situnayake 405). On the other hand, Quantization Aware Training (QAT) is implemented while the DNN is being trained. It builds upon PTQ by mimicking "inference-time" calculations with quantized data and parameters while the DNN is being trained (TensorFlow Developers, "Quantization"). The reason why Deep

Compression is so effective in improving the memory-efficiency of a DNN will be explored in the following experimental analysis.

2.3 Experimental Overview

Deep Compression will be implemented on a Convolutional Neural Network (CNN), a type of DNN for computer vision - "[deriving] meaningful information from digital images" ("Computer Vision"). It is a traditional Deep Learning model and a popular application of Machine Learning.

To generate primary experimental data relevant to this paper's research question, Google's TensorFlow Machine Learning software library will be used to implement Deep Compression with the Python programming language. Additionally, the CNN will be trained using the MNIST dataset, which contains 60,000 images of various handwritten digits, making it a valid source of data (LeCun et al.). Finally, after being compressed into a model suitable for TinyML, the memory requirements and accuracy of the CNN will be recorded to quantify how well the compression algorithms have performed and will be analyzed and evaluated in the context of this paper's research question.

2.4 Deep Compression Experimental Analysis

The baseline CNN architecture is initially highly memory-inefficient because it contains several parameters and is densely connected, using 78,211 bytes of memory. The code for this implementation of Deep Compression (see Appendix 5.1) is heavily adapted from the TensorFlow documentation (TensorFlow Developers, "Pruning").

The data generated from this experiment is summarized in Table 1.

Table 1: Effect of Deep Compression on the accuracy and memory usage of a CNN

Metric		**Pruning**	**Quantization (QAT + PTQ)**
Memory required to store the model	**Before Compression**	78,211 bytes	25,797 bytes
	After Compression	25,797 bytes	**8,039 bytes**
Compression rate (compared to the baseline model)		3.03x	**9.73x**
Accuracy	**Before Compression**	97.74%	97.22%
	After Compression	97.22%	**97.21%**

From Table 1, Deep Compression compresses the traditional Deep Learning model by a factor of 9.73, from an initial memory usage of 78,211 bytes to a final memory usage of 8,039 bytes, which equates to a 90% decrease in the model's memory usage. Also, Table 1's data shows the CNN's accuracy to be minimally affected, decreasing by only 0.53%, even after a significant improvement in the model's memory-efficiency. The reasons behind this decrease in memory usage are analyzed below.

Network Pruning highlights the most important connections, rather than "learning the final values of the parameters," as is the case with traditional training, described in the Introduction (Han et al. 3). Thus, Network Pruning involves identifying and eliminating redundant connections. The CNN used in this experiment was implemented such that its weights had values between -1 and 1. The closer the value of a weight is to 0, the less influence the connection, represented by the weight, has on the final model output. To prune the least important connections, a minimum threshold value is chosen manually. If a weight is lower than this fixed threshold, Network Pruning eliminates the connection that the weight

represents. As a result, the dense network is converted into a sparse network, as diagrammed in Figure 2 (Han et al. 3).

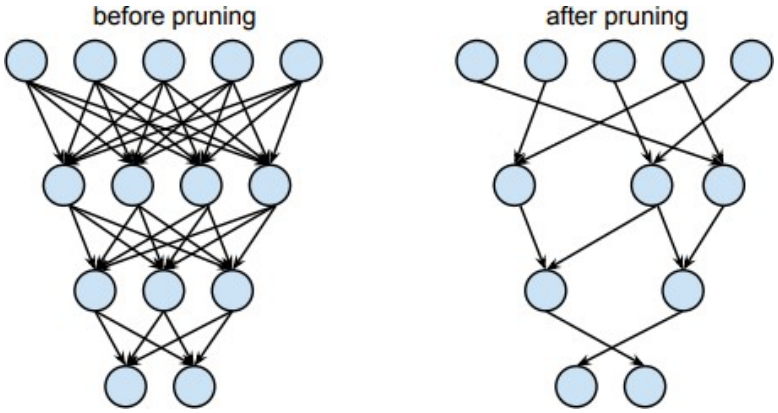

Figure 2: The effect of Pruning the weights of a DNN (Han et al. 3)

TensorFlow was used to determine the threshold value based on the desired sparsity and then set weights below this threshold to a value of zero, rather than actually deleting them. Thus, "a standard compression algorithm" was required to eliminate the redundancy and consequently improve the model's memory-efficiency (TensorFlow Developers, "Pruning").

However, this sparse network had a significantly lower accuracy than the original network. As a result, the model was retrained to adapt to the pruned connections. As the sparsely connected model was retrained, the CNN fine-tuned the weights for the remaining connections to adjust to Network Pruning. Thus, retraining the pruned network was critical to ensure the model maintained a high accuracy during inference while maximizing its memory-efficiency.

It is worth mentioning that although not implemented in this paper, Pruning can be repeated numerous times to extract only the crucial connections from the DNN. However, the model's accuracy can be significantly degraded after a large proportion of the original

parameters have been pruned, so the optimal number of iterations of Pruning, whereby model size is minimized, and accuracy maximized, is found through trial-and-error (Han et al. 7).

TensorFlow presents functionality to implement Quantization immediately after the original Deep Learning model has been pruned (TensorFlow Developers, "Pruning"). As was the case with Pruning, once Quantization was implemented, the model was retrained to ensure the compressed model adapted to the changes it had undergone.

While PTQ is mathematically technical, float32 was simply more precise than was "required for inference" (Warden and Situnayake 404). Thus, representing the model's parameters with the memory-efficient int8 format allowed the CNN to retain its high accuracy and decreased its memory requirements by about a factor of four (Paupamah et al. 3). Although PTQ is a lossy compression algorithm, the information lost upon compressing the model parameters was outweighed by the improved memory-efficiency. Hence, PTQ contributed considerably to the increased memory-efficiency seen in Table 1.

Next, QAT was implemented. The motivation for emulating inference during training was to make the final model more resilient to the errors due to quantized input data during inference. That is, in the process of being trained, the CNN was exposed to possible errors due to quantized data so that the model parameters were trained to learn values that somewhat accounted for the quantized input data. Thus, the QAT algorithm prepared the model for real-world inference and ensured a minimal decrease in its accuracy. Crucially, it helped maintain model performance. From Table 1, the accuracy drops by only 0.01% after Quantization because QAT offsets the decrease in accuracy due to PTQ.

Thus, PTQ and QAT worked together to achieve "lower bit-precision" of model parameters while finding a balance between the accuracy and memory usage of the CNN (Ghamari et al. 1). Furthermore, Quantization was beneficial for other metrics of the CNN.

For example, when the model weights were in int8, the inference-time calculations were faster, so the latency of the model was decreased, leading to a better user experience.

Thus, the findings of this experiment show that Deep Compression drastically improves the traditional Deep Learning model's memory-efficiency through the Pruning and PTQ algorithms, while the QAT algorithm and retraining ensure that the model's accuracy is not significantly degraded. Referring to the Research Question, Deep Compression can compress traditional Deep Learning models to be more memory-efficient for the field of Tiny Machine Learning by approximately a factor of 10, as evidenced by the decrease in the memory required by a model's parameters when the CNN, a traditional Deep Learning model, is compressed with Deep Compression.

By analyzing the experimental data, it can be understood that Deep Compression is effective at decreasing the model's memory footprint because it uses distinct algorithms in concert to reduce the model size significantly while minimizing the accuracy loss. The Deep Compression workflow is often leveraged to compress a DNN for use in smartphones or microcontrollers (Han et al. 12). However, the tradeoff between the model's accuracy and its memory requirements becomes more significant when compressing a model for deployment onto a microcontroller because its resources are more constrained than a smartphone's. Therefore, to perform inference with a DNN on a microcontroller, there will inevitably be an accuracy loss, albeit one that can be minimized by efficiently applying the Deep Compression workflow.

3. Knowledge Distillation

Knowledge Distillation is a model compression technique that uses a specialized Teacher-Student framework to "distill" the "knowledge" of a traditional Deep Learning model

into a more memory-efficient model (Hinton et al. 1). This technique was developed by Hinton et al., who all worked at Google AI, and it was one of the first model compression techniques. Consequently, this technique has been examined extensively by several experts in AI, making Hinton et al.'s overview of Knowledge Distillation a valid source for this paper. Hence, Knowledge Distillation will be analyzed by examining how a Teacher-Student framework is applied to a DNN to improve its memory-efficiency.

3.1 Teacher-Student Framework

The underlying idea of the Teacher-Student framework used in Knowledge Distillation is simple - a large DNN is trained using a dataset of interest, most likely in a data center, and then the "knowledge" contained in this cumbersome DNN's parameters is "distilled," or transferred into a smaller model (Hinton et al. 1). The original cumbersome DNN is the "Teacher Network" which transfers its "knowledge" to the smaller "Student Network" (Wang et al. 3).

Naturally, the smaller Student Network, with significantly fewer neurons, layers, and parameters, has lower storage requirements. Therefore, inference with the Student Network is computationally cheaper, meaning that it can be implemented on memory-constrained edge devices for TinyML. Also, the process of Knowledge Distillation does not significantly diminish the accuracy of the original model because the Student Network's parameters learn the crucial information of the model from the Teacher Network.

3.2 Knowledge Distillation Experimental Analysis

Similar to Deep Compression, Knowledge Distillation was implemented on a CNN (a subset of DNNs and a traditional Deep Learning Model) with the Keras library and was also

trained on the MNIST dataset (Borup). The code for implementing Knowledge Distillation (see Appendix 5.2) was heavily adapted from the Keras documentation because the implementation of the 'distiller' algorithm, which distills the Teacher Network's knowledge into the memory-efficient Student Network, is very technical (Borup). First, the traditional Deep Learning model, or the Teacher CNN, was trained as a normal CNN. However, after training, the distiller algorithm was applied to the Teacher CNN to transform it into a memory-efficient Student CNN. The newly distilled Student Network was then retrained to ensure its accuracy was not degraded.

The data generated from this experiment is summarized in Table 2.

Table 2: Effect of Knowledge Distillation on the accuracy and memory usage of a CNN

Metric	Teacher CNN	Student CNN (after retraining)
Memory required to store the model	6,215,760 bytes	**142,800 bytes**
Compression rate (compared to the Teacher CNN)	N/A	**43.53x**
Accuracy	96.21%	**95.71%**

From Table 2, it is clear that Knowledge Distillation leads to a drastic decrease in memory usage when the Teacher's Knowledge is distilled into the Student CNN. A compression rate of 43.53x means that the Student CNN's memory requirements are a mere 2.3% of the Teacher CNN's memory usage. Additionally, the accuracy of the Student model is only 0.5% less than that of the Teacher, meaning that the distiller algorithm was able to effectively pass on the Teacher's 'knowledge' to the Student.

In Knowledge Distillation, the main hurdle to implementation was to actually "transfer the knowledge from [the] large Teacher model to [the] small Student model," as shown in Figure 3 (Gou et al. 2).

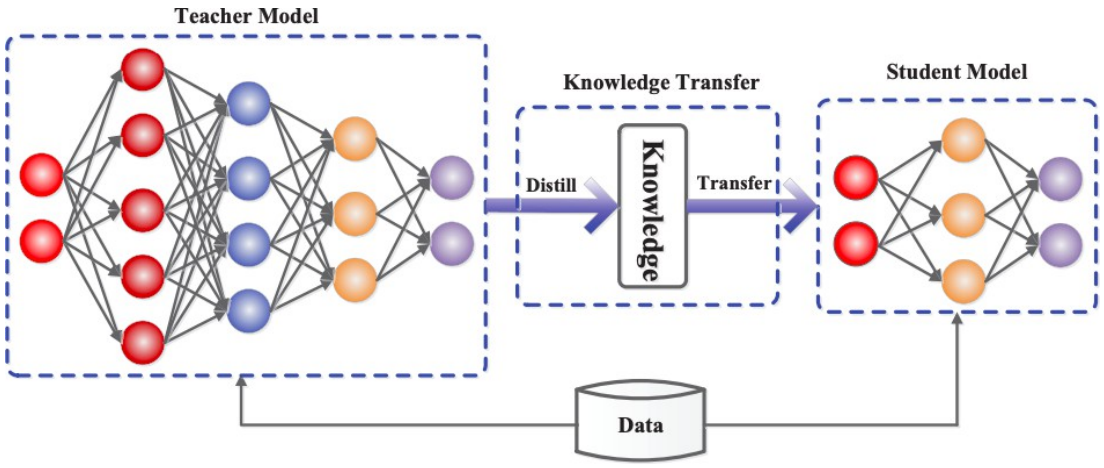

Figure 3: Knowledge transfer from Teacher to Student (Gou et al. 2)

In order to complete this transfer of knowledge, the Student CNN, with a significantly lower number of layers and neurons, had to learn to mimic the Teacher Network rather than learn from the dataset itself (Wang et al. 2). At this point, it is essential to note that the Student model was not trained on the raw dataset because it, by definition, had fewer neurons and layers than the large Teacher model, so if the Student were trained with the original dataset, it's accuracy would be sub-optimal. Instead, the Student model attempted to mimic the Teacher by tuning its weights such that given a specific input, the Student could match the Teacher's output reasonably well.

To illustrate the process of Knowledge Distillation used in this experiment, consider a simple example where a Teacher DNN using nine layers, with five neurons in each layer, has been trained, and the knowledge of the trained Teacher model is to be distilled into a smaller Student DNN containing three layers of three neurons each. In this scenario, the neurons in the first layer of the Student would tune their parameters to emulate the first three layers of

the Teacher, the next layer of the Student DNN would mimic the next three layers of the Teacher, and so on until the Student DNN has distilled the Teacher into a more memory-efficient model with a similar accuracy.

In practice, Knowledge Distillation is performed almost entirely in data centers because this TinyML compression technique requires a computationally intensive 'helper' algorithm that helps the Student Network mimic the Teacher Network during Knowledge Distillation (Wang et al. 1). However, in the experiment described in Table 2, there was no need to use a data center's vast resources because the Teacher CNN was relatively small. From Table 2, distillation was highly effective, with the Student CNN's accuracy only half a percentage point below the Teacher CNN's accuracy while using 97.7% less memory.

Hence, it can be concluded from this experiment that Knowledge Distillation is a viable alternative to Deep Compression for compressing DNNs to be deployed on resource-constrained edge devices because it results in Student DNNs that have a similar accuracy to the original model but are also highly memory-efficient. With reference to the Research Question, the results of this experiment support the thesis of this paper as the experimental data displays that Knowledge Distillation improves the memory-efficiency of the traditional Deep Learning model by a factor of almost 45, without sacrificing the accuracy of the original model, making it a valuable tool for TinyML.

In addition, further insight can be drawn regarding the Research Question by critically analyzing Table 1 and Table 2 together. Although the goal of this paper is not to carry out a comparative study of Deep Compression and Knowledge Distillation, it is valuable to explore these two DNN memory-compression techniques. Specifically, Knowledge Distillation results in a considerably more memory-efficient model than Deep Compression because its compression rate is approximately 45x compared to Deep Compression's compression rate of

10x. However, implementing Knowledge Distillation is more mathematically intensive than Deep Compression, meaning that it requires more computational resources during implementation. Yet, both these techniques result in similar accuracies of the final compressed model. Hence, both techniques have their advantages and disadvantages, and which one is ultimately chosen to compress a traditional Deep Learning model depends on the resources available and the amount of memory compression desired.

4. Conclusion

In this paper, two techniques for compressing traditional Deep Learning Models for TinyML were analyzed. TinyML was born out of the need to perform inference at the edge; that is, to perform Machine Learning on resource-constrained edge devices rather than in remote data centers. The major challenge in accomplishing this is that traditional Deep Learning models require too much memory and energy to be used on edge devices like smartphones or microcontrollers. To address this, several methods like Deep Compression and Knowledge Distillation have been implemented to sufficiently decrease the memory and energy requirements of traditional Deep Learning models for TinyML use cases.

Deep Compression and Knowledge Distillation were discussed in the context of increasing the memory-efficiency of Deep Learning models like DNNs or CNNs. Although the memory-efficiencies of the original models were improved after applying both techniques, there was no evidence of the assumption that the model's energy-efficiency improved. To address this unanswered question, further research would need to be undertaken, and the original scope of this paper would have to be broadened. For instance, the power consumption of a traditional Deep Learning model performing inference in a data center could be compared with the energy-efficiency of a compressed model in a

microcontroller. This would result in valuable data regarding the feasibility of these compression techniques for TinyML applications. It would also be interesting to investigate the effect of these techniques in compressing specific layers or even using different techniques together to achieve greater memory-efficiency. Moreover, to expand the original scope of the paper, it would be valuable to explore compression techniques of other Deep Learning architectures like Recurrent Neural Networks and compare these results to those discovered for DNNs in this paper.

The data generated in this paper provides strong support for the thesis expressed in the introduction because the experimental data shows that through Deep Compression or Knowledge Distillation, the model's efficiency can be improved by an approximate factor of 10 or 45 times, respectively. This means that given a memory-inefficient traditional Deep Learning model, a TinyML compression algorithm can be applied upon the original model to yield a model that can perform inference in microcontrollers. For example, the Knowledge Distillation experiment displayed that the size of a CNN could be decreased from 6.2 megabytes to 142 kilobytes meaning that the model, which was previously incompatible with a microcontroller with kilobytes of memory, is now sufficiently memory-efficient to perform inference on the microcontroller, assuming that the model's energy-efficiency is also compatible with the microcontroller. In this manner, TinyML is shaping the future of on-device Machine Learning and enabling applications from digital voice assistants like Siri on iPhones to wildfire detection ("tinyTalks ANZ"). **In conclusion, the data presented in this paper demonstrates that traditional Deep Learning models can be compressed by up to a factor of 45 to be more memory-efficient for the field of Tiny Machine Learning.**

Works Cited

Bawden, Tom. "Global Warming: Data Centres to Consume Three Times as Much Energy in Next Decade, Experts Warn." The Independent, 23 Jan. 2016, www.independent.co.uk/climate-change/news/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html. Accessed 2 Aug. 2021.

Borup, Kenneth. Implementation of Classical Knowledge Distillation. Keras, 1 Sept. 2020, keras.io/examples/vision/knowledge_distillation/. Accessed 20 July 2021.

Canini, Marco. "Virtualization and Cloud Computing." King Abdullah University of Science and Technology, web.kaust.edu.sa/Faculty/MarcoCanini/classes/CS240/F17/slides/L3-cloud-VM.pdf. Accessed 20 July 2021.

"Computer Vision." International Business Machines, www.ibm.com/topics/computer-vision. Accessed 9 Aug. 2021.

Denil, Misha, et al. "Predicting Parameters in Deep Learning." International Conference on Neural Information Processing Systems, vol. 26, 5 Dec. 2013, arxiv.org/pdf/1306.0543.pdf. Accessed 6 Dec. 2021.

Ghamari, Sedigh, et al. "Quantization-Guided Training for Compact TinyML Models." TinyML 2021 Research Symposium, 5 Feb. 2021 Accessed 30 June 2021.

Gou, Jianping, et al. "Knowledge Distillation: A Survey." International Journal of Computer Vision, vol. 129, no. 6, 22 Mar. 2021, pp. 1789-819, https://doi.org/10.1007/s11263-021-01453-z. Accessed 1 Oct. 2021.

Han, Song, et al. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." International Conference on Learning Representations, 15 Feb. 2016, arxiv.org/pdf/1510.00149.pdf. Accessed 2 Dec. 2021.

Hinton, Geoffrey, et al. "Distilling the Knowledge in a Neural Network." Conference on Neural Information Processing Systems, 9 Mar. 2015, arxiv.org/pdf/1503.02531.pdf. Accessed 6 Dec. 2021.

"History of Artificial Intelligence." Council of Europe, www.coe.int/en/web/artificial-intelligence/history-of-ai. Accessed 16 June 2021.

IBM Cloud Education. "Neural Networks." International Business Machines, 17 Aug. 2020, www.ibm.com/cloud/learn/neural-networks. Accessed 5 July 2021.

LeCun, Yan, et al. "The MNIST Database of Handwritten Digits." Yan LeCun, 1999, yann.lecun.com/exdb/mnist/. Accessed 6 Dec. 2021.

Paupamah, Kimessha, et al. "Quantisation and Pruning for Neural Network Compression and Regularisation." Southern African Universities Power Engineering Conference 2020, 14 Jan. 2020, https://doi.org/10.1109/SAUPEC/RobMech/PRASA48453.2020.9041096. Accessed 2 June 2021.

Smith, Chris, et al. "The History of Artificial Intelligence." University of Washington Computer Science and Engineering, Dec. 2006, courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf. Accessed 30 June 2021.

Tamir, Michael. "What Is Machine Learning?" Berkeley School of Information, 26 June 2020, ischoolonline.berkeley.edu/blog/what-is-machine-learning/. Accessed 2 July 2021.

TensorFlow Developers. "Pruning in Keras Example." TensorFlow, www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras. Accessed 6 Dec. 2021.

TensorFlow Developers. "Quantization Aware Training." TensorFlow, www.tensorflow.org/model_optimization/guide/quantization/training. Accessed 6 Dec. 2021.

"tinyTalks ANZ: What, Why and How of TinyML." YouTube, uploaded by TinyML, 29 May 2021, www.youtube.com/watch?v=v1nMMnWU_dY&ab_channel=tinyML. Accessed 6 Dec. 2021.

"Types and Applications of Microcontrollers." Engineering Institute of Technology, www.eit.edu.au/resources/types-and-applications-of-microcontrollers/. Accessed 29 July 2021.

Valueva, M.V., et al. "Application of the Residue Number System to Reduce Hardware Costs of the Convolutional Neural Network Implementation." Mathematics and Computers in Simulation, vol. 177, Nov. 2020, pp. 232-43, https://doi.org/10.1016/j.matcom.2020.04.031. Accessed 6 Dec. 2021. Abstract.

Wang, Junpeng, et al. "DeepVID: Deep Visual Interpretation and Diagnosis for Image Classifiers via Knowledge Distillation." IEEE Transactions on Visualization and Computer Graphics, vol. 25, no. 6, 1 June 2019, pp. 2168-80, https://doi.org/10.1109/TVCG.2019.2903943. Accessed 6 Dec. 2021.

Warden, Pete, and Daniel Situnayake. TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers. O'Reilly Media, 2019.

5. Appendix

As mentioned in the body of this paper, the code used in this paper was heavily adapted from online sources of software documentation. Specifically, the code for Deep Compression (section 5.1) was adapted from the TensorFlow documentation (TensorFlow Developers, "Pruning"; TensorFlow Developers, "Quantization"). And the code for Knowledge Distillation (section 5.2) was adapted from the Keras documentation (Borup).

5.1 Code for Deep Compression

```
pip install -q tensorflow-model-optimization
import tempfile
import os
import tensorflow as tf
import numpy as np
from tensorflow import keras

%load_ext tensorboard
# Load MNIST dataset
mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

# Normalize the input image so that each pixel value is between 0 and
1.
train_images = train_images / 255.0
test_images = test_images / 255.0

# Define the model architecture.
model = keras.Sequential([
 keras.layers.InputLayer(input_shape=(28, 28)),
 keras.layers.Reshape(target_shape=(28, 28, 1)),
 keras.layers.Conv2D(filters=12, kernel_size=(3, 3),
activation='relu'),
 keras.layers.MaxPooling2D(pool_size=(2, 2)),
 keras.layers.Flatten(),
 keras.layers.Dense(10)
])

# Train the digit classification model
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model.fit(
 train_images,
```

```
 train_labels,
 epochs=4,
 validation_split=0.1,
)
# Check the baseline model accuracy
_, baseline_model_accuracy = model.evaluate(
    test_images, test_labels, verbose=0)
print('Baseline test accuracy:', baseline_model_accuracy)
_, keras_file = tempfile.mkstemp('.h5')
tf.keras.models.save_model(model, keras_file, include_optimizer=False)
print('Saved baseline model to:', keras_file)
# Pruning
import tensorflow_model_optimization as tfmot
prune_low_magnitude = tfmot.sparsity.keras.prune_low_magnitude
# Compute end step to finish pruning after 2 epochs.
batch_size = 128
epochs = 2
validation_split = 0.1 # 10% of training set will be used for
validation set.
num_images = train_images.shape[0] * (1 - validation_split)
end_step = np.ceil(num_images / batch_size).astype(np.int32) * epochs
# Define model for pruning.
pruning_params = {
      'pruning_schedule':
tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.50,
final_sparsity=0.80,
begin_step=0,
end_step=end_step)
}
model_for_pruning = prune_low_magnitude(model, **pruning_params)
# `prune_low_magnitude` requires a recompile.
model_for_pruning.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
model_for_pruning.summary()
callbacks = [
 tfmot.sparsity.keras.UpdatePruningStep(),
 tfmot.sparsity.keras.PruningSummaries(log_dir=logdir),
]
# retrain the pruned model
model_for_pruning.fit(train_images, train_labels,
                  batch_size=batch_size, epochs=epochs,
validation_split=validation_split,
                  callbacks=callbacks)
_, model_for_pruning_accuracy = model_for_pruning.evaluate(
  test_images, test_labels, verbose=0)
print('Baseline test accuracy:', baseline_model_accuracy)
print('Pruned test accuracy:', model_for_pruning_accuracy)
model_for_export =
tfmot.sparsity.keras.strip_pruning(model_for_pruning)
_, pruned_keras_file = tempfile.mkstemp('.h5')
tf.keras.models.save_model(model_for_export, pruned_keras_file,
include_optimizer=False)
print('Saved pruned Keras model to:', pruned_keras_file)
```

```
# TFLite Converter
converter = tf.lite.TFLiteConverter.from_keras_model(model_for_export)
pruned_tflite_model = converter.convert()
_, pruned_tflite_file = tempfile.mkstemp('.tflite')
with open(pruned_tflite_file, 'wb') as f:
 f.write(pruned_tflite_model)
print('Saved pruned TFLite model to:', pruned_tflite_file)
# helper function to get size of compressed models
# adapted from the TensorFlow documentation
(https://www.tensorflow.org/model_optimization/guide/pruning/pruning_wi
th_keras)
def get_gzipped_model_size(file):
 # Returns size of gzipped model, in bytes.
 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)
print("Size of gzipped baseline Keras model: %.2f bytes" %
(get_gzipped_model_size(keras_file)))
print("Size of gzipped pruned Keras model: %.2f bytes" %
(get_gzipped_model_size(pruned_keras_file)))
print("Size of gzipped pruned TFlite model: %.2f bytes" %
(get_gzipped_model_size(pruned_tflite_file)))
# Quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model_for_export)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
quantized_and_pruned_tflite_model = converter.convert()
_, quantized_and_pruned_tflite_file = tempfile.mkstemp('.tflite')
with open(quantized_and_pruned_tflite_file, 'wb') as f:
 f.write(quantized_and_pruned_tflite_model)
print('Saved quantized and pruned TFLite model to:',
quantized_and_pruned_tflite_file)
print("Size of gzipped baseline Keras model: %.2f bytes" %
(get_gzipped_model_size(keras_file)))
print("Size of gzipped pruned and quantized TFlite model: %.2f bytes" %
(get_gzipped_model_size(quantized_and_pruned_tflite_file)))
# helper function to actually evaluate the model
# adapted from the TensorFlow documentation
(https://www.tensorflow.org/model_optimization/guide/pruning/pruning_wi
th_keras)
def evaluate_model(interpreter):
 input_index = interpreter.get_input_details()[0]["index"]
 output_index = interpreter.get_output_details()[0]["index"]
 # Run predictions on every image in the "test" dataset.
 prediction_digits = []
 for i, test_image in enumerate(test_images):
   if i%1000 == 0:
     print('Evaluated on {n} results so far.'.format(n=i))
   # Pre-processing: add batch dimension and convert to float32 to
match with
   # the model's input data format.
```

```
    test_image = np.expand_dims(test_image, axis=0).astype(np.float32)
    interpreter.set_tensor(input_index, test_image)

    # Run inference.
    interpreter.invoke()
    # Post-processing: remove batch dimension and find the digit with
highest
    # probability.
    output = interpreter.tensor(output_index)
    digit = np.argmax(output()[0])
    prediction_digits.append(digit)
 print('\n')
 # Compare prediction results with ground truth labels to calculate
accuracy.
 prediction_digits = np.array(prediction_digits)
 accuracy = (prediction_digits == test_labels).mean()
 return accuracy
interpreter =
tf.lite.Interpreter(model_content=quantized_and_pruned_tflite_model)
interpreter.allocate_tensors()
test_accuracy = evaluate_model(interpreter)
print('Pruned and quantized TFLite test_accuracy:', test_accuracy)
print('Pruned TF test accuracy:', model_for_pruning_accuracy)
```

5.2 Code for Knowledge Distillation

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
class Distiller(keras.Model):
    def __init__(self, student, teacher):
        super(Distiller, self).__init__()
        self.teacher = teacher
        self.student = student
    def compile(
        self,
        optimizer,
        metrics,
        student_loss_fn,
        distillation_loss_fn,
        alpha=0.1,
        temperature=3,
    ):
        super(Distiller, self).compile(optimizer=optimizer,
metrics=metrics)
        self.student_loss_fn = student_loss_fn
        self.distillation_loss_fn = distillation_loss_fn
        self.alpha = alpha
        self.temperature = temperature
    def train_step(self, data):
        # Unpack data
        x, y = data
```

```
        # Forward pass of teacher
        teacher_predictions = self.teacher(x, training=False)

        with tf.GradientTape() as tape:
            # Forward pass of student
            student_predictions = self.student(x, training=True)
            # Compute losses
            student_loss = self.student_loss_fn(y, student_predictions)
            distillation_loss = self.distillation_loss_fn(
                tf.nn.softmax(teacher_predictions / self.temperature,
axis=1),
                tf.nn.softmax(student_predictions / self.temperature,
axis=1),
            )
            loss = self.alpha * student_loss + (1 - self.alpha) *
distillation_loss
        # Compute gradients
        trainable_vars = self.student.trainable_variables
        gradients = tape.gradient(loss, trainable_vars)
        # Update weights
        self.optimizer.apply_gradients(zip(gradients, trainable_vars))
        # Update the metrics configured in `compile()`.
        self.compiled_metrics.update_state(y, student_predictions)
        # Return a dict of performance
        results = {m.name: m.result() for m in self.metrics}
        results.update(
            {"student_loss": student_loss, "distillation_loss":
distillation_loss}
        )
        return results
    def test_step(self, data):
        # Unpack the data
        x, y = data
        # Compute predictions
        y_prediction = self.student(x, training=False)
        # Calculate the loss
        student_loss = self.student_loss_fn(y, y_prediction)
        # Update the metrics.
        self.compiled_metrics.update_state(y, y_prediction)
        # Return a dict of performance
        results = {m.name: m.result() for m in self.metrics}
        results.update({"student_loss": student_loss})
        return results
# Create the teacher:
teacher = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(256, (2, 2), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1),
padding="same"),
        layers.Conv2D(512, (2, 2), strides=(2, 2), padding="same"),
        layers.Flatten(),
        layers.Dense(10),
```

```
    ],
    name="teacher",
)
# Create the student
student = keras.Sequential(
    [
        keras.Input(shape=(28, 28, 1)),
        layers.Conv2D(16, (2, 2), strides=(2, 2), padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.MaxPooling2D(pool_size=(2, 2), strides=(1, 1),
padding="same"),
        layers.Conv2D(32, (2, 2), strides=(2, 2), padding="same"),
        layers.Flatten(),
        layers.Dense(10),
    ],
    name="student",
)
# Prepare the train and test dataset.
batch_size = 64
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
# Normalize data
x_train = x_train.astype("float32") / 255.0
x_train = np.reshape(x_train, (-1, 28, 28, 1))
x_test = x_test.astype("float32") / 255.0
x_test = np.reshape(x_test, (-1, 28, 28, 1))
# Train teacher as usual
teacher.compile(
    optimizer=keras.optimizers.Adam(),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
)
# Train and evaluate teacher on data.
teacher.fit(x_train, y_train, epochs=5)
teacher.evaluate(x_test, y_test)
# Begin Knowledge Distillation
distiller = Distiller(student=student, teacher=teacher)
distiller.compile(
    optimizer=keras.optimizers.Adam(),
    metrics=[keras.metrics.SparseCategoricalAccuracy()],
student_loss_fn=keras.losses.SparseCategoricalCrossentropy(from_logits=
True),
    distillation_loss_fn=keras.losses.KLDivergence(),
    alpha=0.1,
    temperature=10,
)
# Distill teacher to student
distiller.fit(x_train, y_train, epochs=3)
# Evaluate student on test dataset
distiller.evaluate(x_test, y_test)
print("Teacher Summary")
teacher.summary()
print("Distilled Student Summary")
student.summary()
```