Analyzing the Effect of Quantization on the Neural Network Loss Landscape with the Hessian

Ashwin Adulla, aadulla
GitHub: https://github.com/aadulla/21344_NN_Quantization_Hessian_Analysis.git

1. Introduction

As deep learning grows in its applicability to various problems from image classification to speech synthesis, new research in the field has shifted towards making neural networks more memory/energy efficient so that they can be deployed in edge devices such as microcontrollers, smartphones, and cameras. The common approach used nowadays is quantization where the weights of the matrices in a neural network are quantized from 32-bit float or 64-bit doubles to 8-bit integer. Quantization not only compresses the model to a smaller size, making it more memory efficient, but also allows the underlying hardware's integer functional units to be used which are much faster and less power-hungry than floating-point functional units.

In the process of quantization, information is inevitably lost and the network accuracy tends to decrease. To address these issues, various quantization schemes have been developed, each with their advantages and disadvantages. This paper investigates these quantization schemes by analyzing their loss landscapes with second-order information without explicitly forming the Hessian because that would be computationally infeasible as the network size grows.

2. Background

This section will present a brief survey of a) the various quantization schemes used and b) the power iteration algorithm used to compute second order information.

2.1 Quantization

Quantization is the process of mapping a range of higher-precision numbers to a range of lower-precision numbers. Most of the ideas explained below are adapted from [1] and [2].

Let $[\alpha, \beta]$ be the range of real numbers to be quantized. Let *b* be the number of bins of the integer format we want to quantize to. Quantization would then map the range $[\alpha, \beta]$ to the range $[-2^{b-1}, 2^{b-1} - 1]$ by defining some scale *s* and zero-point *z*. Given a real number $x \in [\alpha, \beta]$ the quantized representation of *x* is given by: $f(x) = s * x + z$.

This project investigates the quantization of 32-bit floating-point to 8 bit integers by 3 popular schemes: Affine, Scale, and KL Divergence. The only difference between these 3 schemes is in how the scale *s* and the zero point *z* are chosen.

2.1.1 Affine Quantization

Affine quantization determines *s, z* by equations in Figure 1.

$$s = \frac{2^b - 1}{\alpha - \beta}$$

$$z = -\operatorname{round}(\beta \cdot s) - 2^{b-1}$$

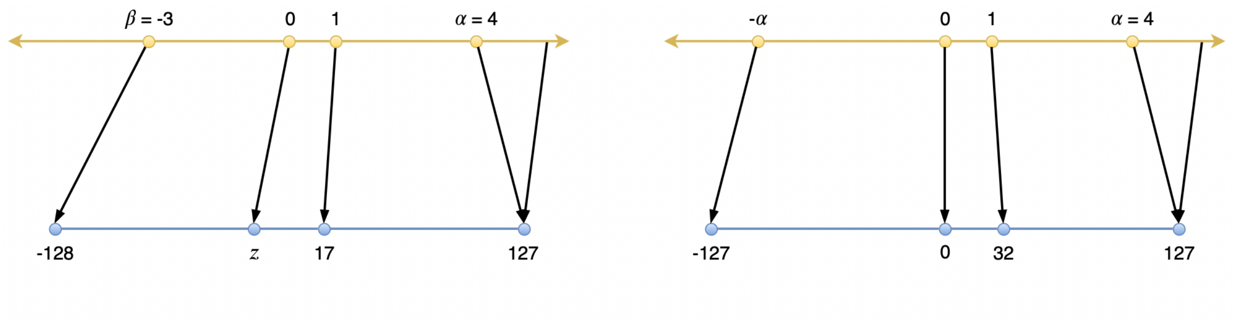
Figure 1: Affine Quantization Equations

2.1.2 Scale Quantization

Scale quantization hardcodes *z*=0 and determines *s* the equation in Figure 2.

$$s = \frac{2^{b-1} - 1}{\alpha}$$

Figure 2: Scale Quantization Equation



(a) Affine quantization (b) Scale quantization

Figure 3: Affine and Scale Quantization Mapping

2.1.3 KL Divergence Quantization

KL Divergence Quantization attempts to minimize the KL divergence between the distribution of numbers after quantization (*Q*) to the distribution of numbers before quantization (*P*), where the KL divergence is given by the following formula:

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

Figure 4: KL Divergence Formula

Intuitively, given the real numbers we wish to quantize, we create a pre-quantized distribution *P* (shown as the red histogram in Figure 5). We then iteratively change the scale and zero point so that our post-quantized distribution *Q* (shown as the green histogram in Figure 5) will shift towards the left so that it more closely matches *P*.

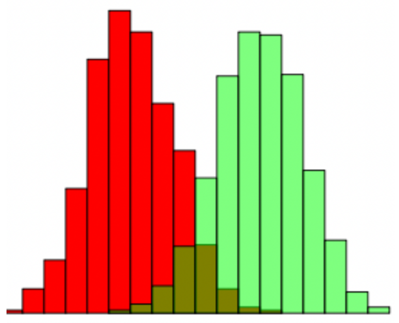


Figure 5: 2 Histogram Plot

2.2 Power Iteration for Second-Order Information

2.2.1 Neural Network Loss Function

A neural network can be compactly represented as a function $f: x \rightarrow \hat{y}$ where x is the input to the network, such as an image of a single digit, and $\hat{y}$ is the output of the network, such as a classification of the digit as either 0, 1, …, or 9. To train the network, we aim to minimize a differentiable loss function $L(y, \hat{y})$ where y is what we want the network to output and $\hat{y}$ is what the network actually outputs. This minimization procedure, known as back-propagation, is typically carried out through some variant of stochastic gradient descent. That is, we compute the derivative of the loss with respect to the individual parameters of the network, and then update the network's parameters in the negative direction of their gradients.

Just as we are able to take the first derivative of the network's parameters, we can also take the second derivative which yields the Hessian. The Hessian has form shown in Figure 6. In the case of a neural network, each x_i is a parameter of the network (i.e. a component of the weight matrices), and *f* is the loss function *L*.

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\[2.2ex] \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\[2.2ex] \vdots & \vdots & \ddots & \vdots \\[2.2ex] \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Figure 6: Hessian Matrix

2.2.2 Hessian Power Iteration

As neural networks typically contain on the order of millions, and sometimes even billions, of parameters, explicitly computing the Hessian is infeasible. However, if we are only concerned with analyzing the dominant eigenvalue and eigenvector of the Hessian, then we can use a slightly modified version of power iteration. Most of the ideas below are adapted from [3].

Let θ ∈ $\mathbb{R}^m$ be the parameters of a neural network (i.e. there are m parameters). The first derivative of the loss w.r.t the parameters is given by $\frac{\partial L}{\partial \theta} = g_\theta \in \mathbb{R}^m$. The second derivative of the loss w.r.t. the parameters is given by H = $\frac{\partial^2 L}{\partial \theta} = \frac{\partial g_\theta}{\partial \theta} \in \mathbb{R}^{mxm}$.

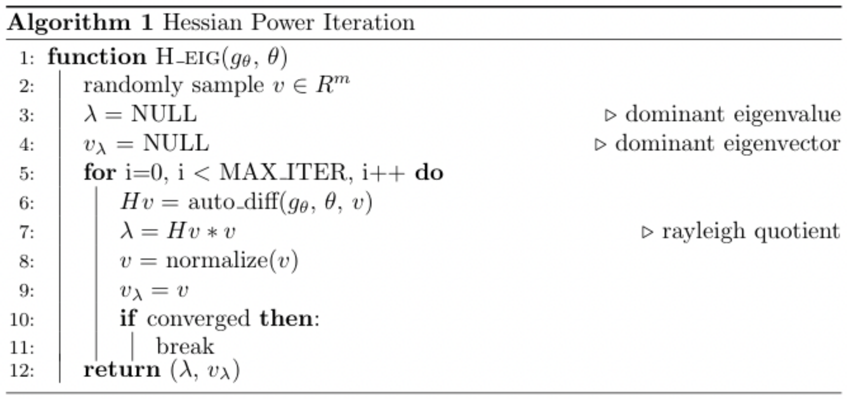
The to find the dominant eigenvalue of H, power iteration requires a matrix-vector product of H with some vector *v*. We can form this product H*v* with the following identity:

$$\frac{\partial g_\theta^T v}{\partial \theta} = \frac{\partial g_\theta^T}{\partial \theta} v + g_\theta^T \frac{\partial v}{\partial \theta} = \frac{\partial g_\theta^T}{\partial \theta} v = Hv$$

Figure 7: Hessian Matrix-Vect Mult Identity

All that remains then is finding some way to compute $\frac{\partial {g_\theta}^T}{\partial \theta}$ which can easily be done through popular auto-differentiation libraries such as PyTorch and Tensorflow.

Thus, without explicitly forming H, we can compute H*v*, which then allows us to use the following power iteration algorithm to find the dominant eigenvalue and eigenvector:



Algorithm 1 Hessian Power Iteration

```
function H_EIG($g_\theta$, $\theta$)
    randomly sample $v \in R^m$
    $\lambda$ = NULL                                    ▷ dominant eigenvalue
    $v_\lambda$ = NULL                                  ▷ dominant eigenvector
    for i=0, i < MAX_ITER, i++ do
        $Hv$ = auto_diff($g_\theta$, $\theta$, $v$)
        $\lambda = Hv * v$                              ▷ rayleigh quotient
        $v$ = normalize($v$)
        $v_\lambda = v$
        if converged then:
            break
    return ($\lambda$, $v_\lambda$)
```

Figure 8: Hessian Power Iteration

3. Methodology

In this project, I trained and analyzed 3 different neural networks on the MNIST dataset. The MNIST dataset is a collection of 28x28 pixel images where each image contains a single handwritten digit. Each neural network was trained to predict the digit in the image. All networks were trained and tested on the same portions of the MNIST dataset to allow for equal comparisons across networks.

3.1 Experiment Setup

All networks were configured as follows:
28x28 image → flatten to 784 vec → 784xM layer → (1) MxM layer → (2) MxM layer → Mx10 layer

When passing an image through the network, it is first flattened, then it is passed through 3 layers with ReLU activation. The last layer (Mx10) outputs a probability distribution across the 10 possible classes for the digit in the image. M was the parameter I varied across the 3 different networks.

Network 32 had M=32.
Network 64 had M=64.
Network 128 had M=128.

This was the procedure followed for each network:

1. Train network for 20 epochs
2. For each of the MxM layers ((1) and (2))
    a. For each quantization scheme (affine, scale, kl divergence)
        i. Quantize the weight matrix of this MxM layer to int8
        ii. Dequantize from int8 to float (note that the weight matrix will not be the same as before quantization since information was lost in the quantization process. We want to determine how sensitive the weight matrix is now after having lost that information)
        iii. Get the dominant eigenvector and eigenvalue of the Hessian (H)
        iv. Perturb the weight matrix by H * some delta
        v. With this perturbed matrix, re-assess the accuracy and loss of the network
        vi. Restore the original weight matrix to this layer

It is important to note that each layer is individually analyzed while the rest of the network is held constant. That is, only a single layer is quantized→dequantized→perturbed at a time.

3.2 Experiment Code

Most of the programming was done in C++ using the LibTorch API for training and analyzing neural network parameters and the Eigen library for linear algebra algorithms. A Python script launches several runs with different neural network configurations. Each run calls into my C++ code to train, analyze, and log the results. The results are aggregated into JSON files, which I then extract and visualize in Jupyter Notebooks.

4. Results

4.1 Loss and Accuracy during Training

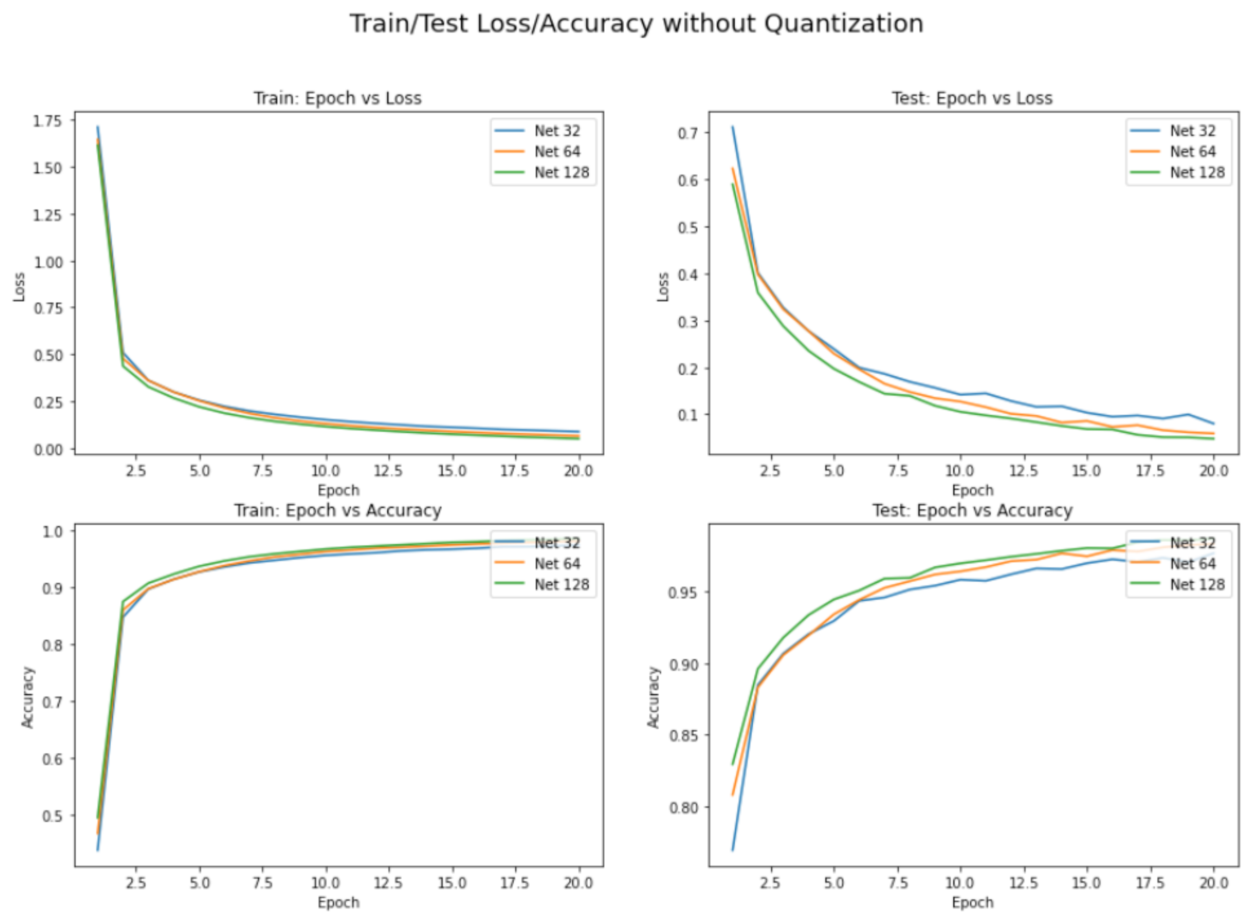


Figure 9: Train/Test Loss/Accuracy without Quantization

All networks were trained via Stochastic Gradient Descent with the same hyperparameters for 20 epochs. All networks converged in training loss and accuracy, but not in test loss and accuracy. Net 128 performed the best, followed by Net 64, and then by Net 32.

4.2 Loss Landscape

After training for 20 epochs, Layers 1 and 2 of each of the neural networks were individually quantized, dequantized, and perturbed by the dominant eigenvector of their Hessian with respect to their loss on the test portion of the MNIST dataset.

The convexity/concavity of the loss/accuracy curve over the perturb deltas highlights how sensitive this layer to slight changes in its weights. Specifically, the sharper the curve, the more sensitive the layer is.

The horizontal/vertical shift of the local minimas/maximas between any of the quantization curves (Affine, KL Div, Scale) and the non-quantization curve (No Q) highlights how the loss landscape generated by the layer has changed. Specifically, any shift implies that quantization has adjusted the basin around the local optimum in which the parameters currently sit. If the loss/accuracy at Perturb Delta=0.0 is not near the local optimum, then this could signal that we should resume training with the altered weights so that it converges to the local optimum.

4.2.1 Net 32 Loss Landscape

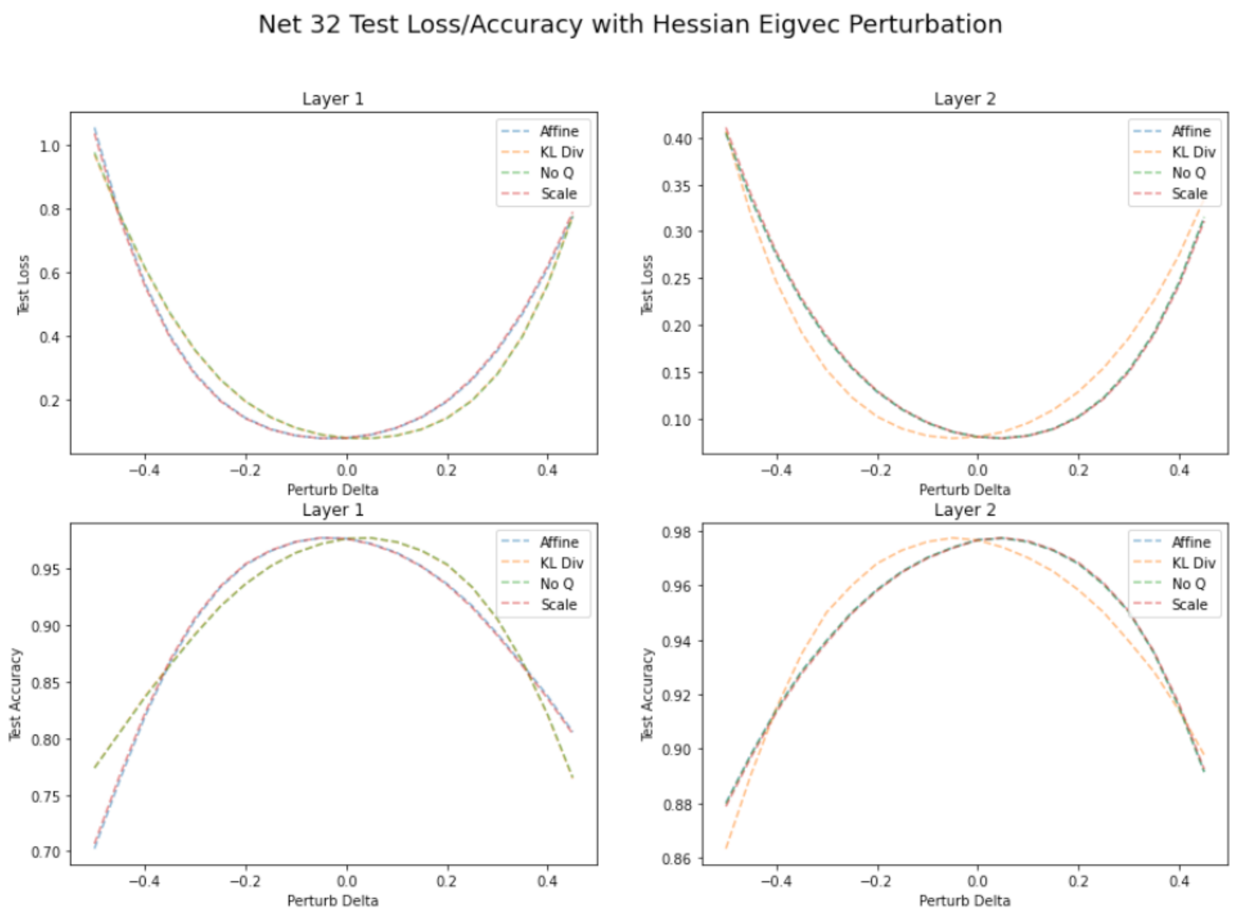


Figure 10: Net 32 Loss Landscape

In Layer 1, the Affine and Scale curves line up together while the KL Div and No Q curves line up together. This means that Affine and Scale quantization generate a similar loss landscape for this weight matrix, while KL Divergence quantization generates a similar loss landscape compared to that of the weight matrix before quantization. As the optimum of the Affine and Scale curves are slightly shifted away from Perturb Delta=0.0, we could resume training of the network to converge.

However, Layer 2 presents a different story. The KL Div curve is distinct from the other 3 and is shifted away from Perturb Delta = 0.0. Thus, KL Divergence quantization leads to worse overall performance on the loss/accuracy when used on Layer 2, compared to using Affine and Scale quantization.

4.2.2 Net 64 Loss Landscape

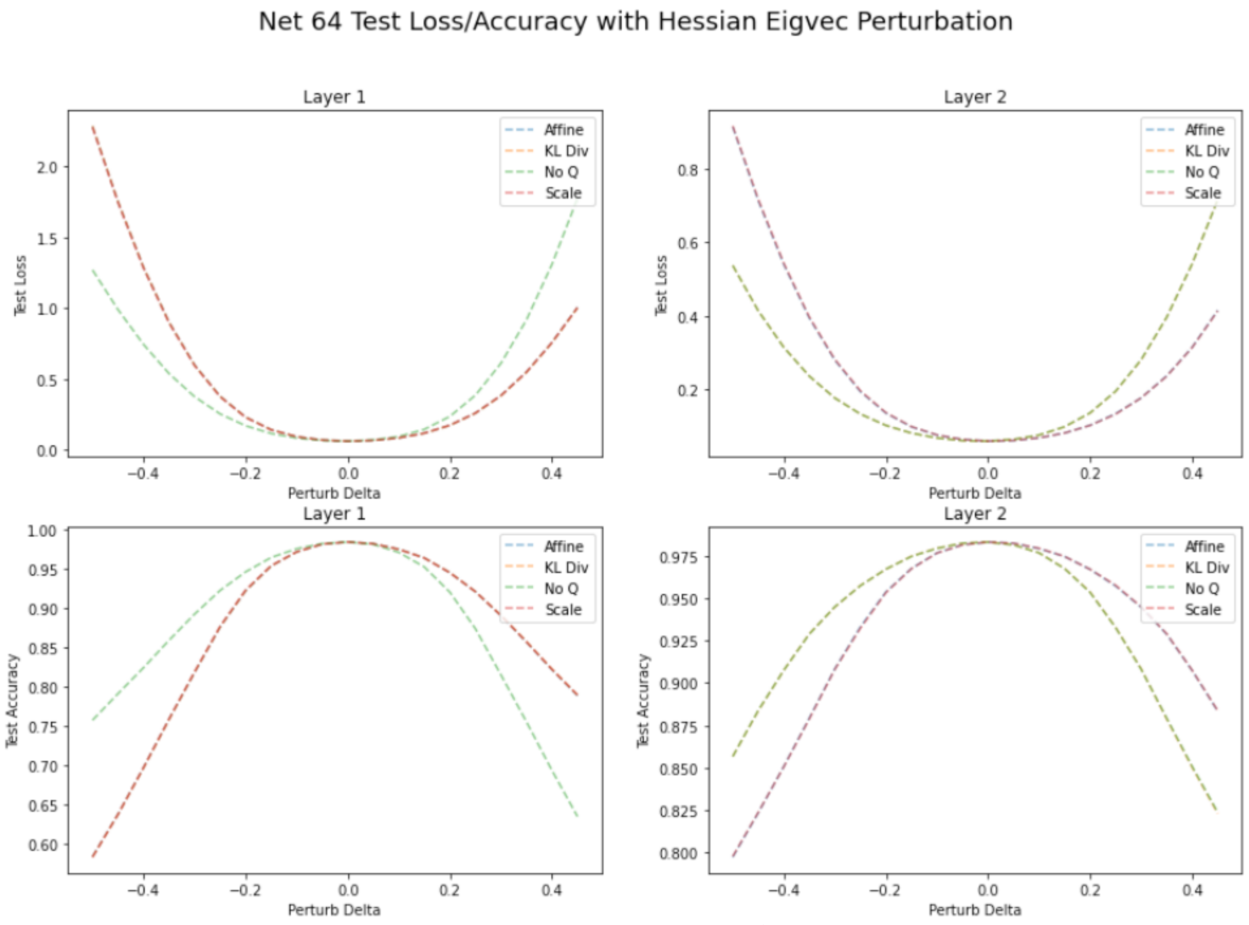


Figure 11: Net 64 Loss Landscape

Compared to Net 32's curves, Net 64's curves have a larger change in Test Loss/Accuracy over the same Perturb Delta domain. This suggests that Net 64's curves with and without quantization are sharper and thus are more sensitive small fluctuations in their weight parameters. Also, in both Layers 1 and 2, Affine and Scale share a similar landscape while KL Div and No Q share a similar landscape.

4.2.3 Net 128 Loss Landscape

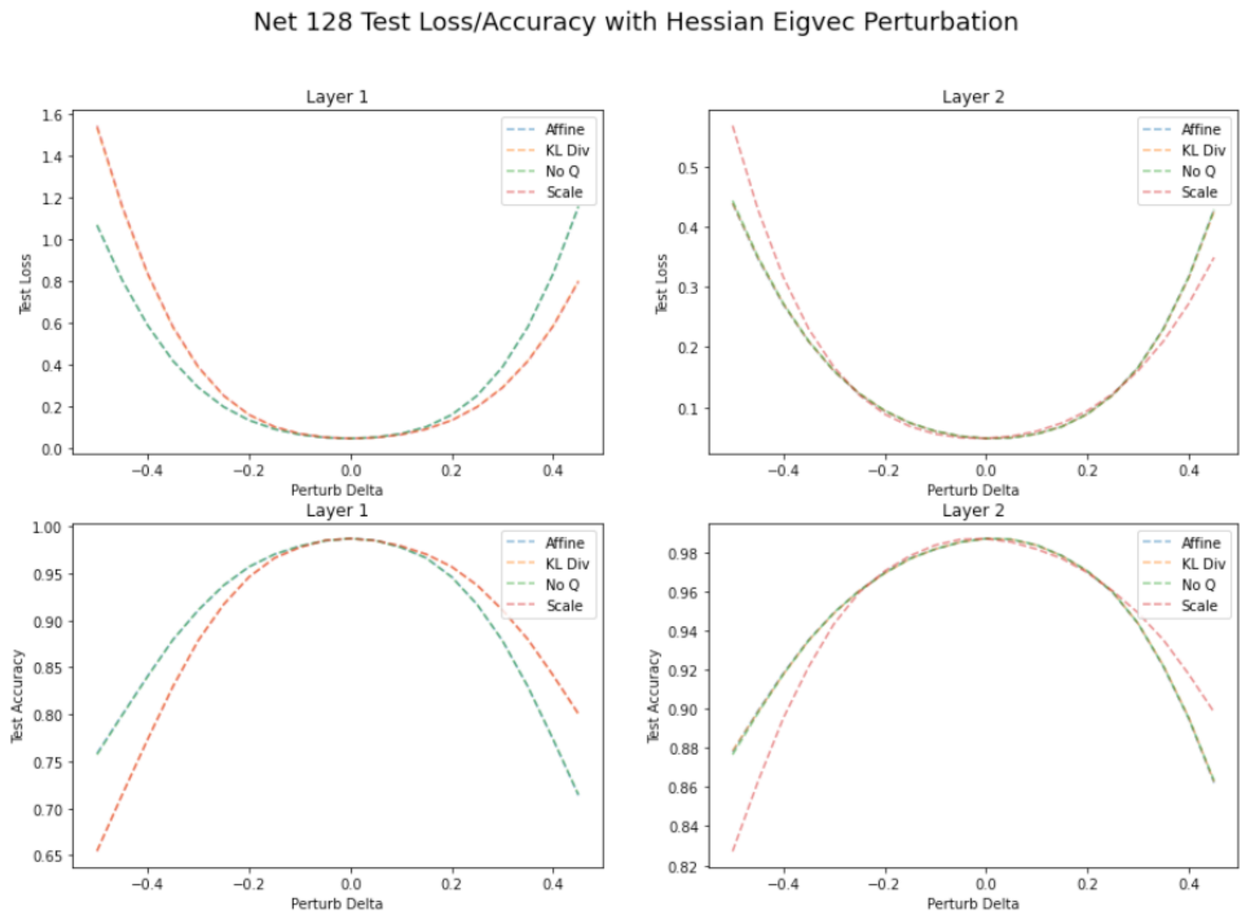
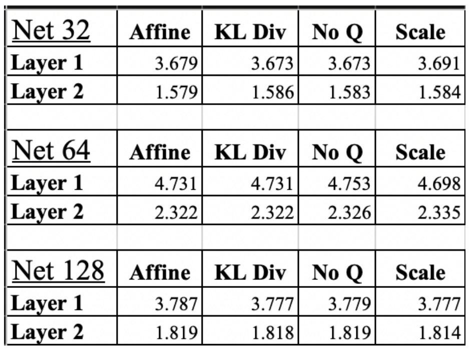


Figure 12: Net 128 Loss Landscape

The sharpness of Net 128's curves are between those of Net 32's and Net 64's. However, the overall shape of Net 128's curves almost exactly resemble those of Net 64. In both layers, the Affine and Scale are aligned, and KL Div and No Q are aligned. There also isn't much change in the shape of the loss landscapes between layers.

4.3 Dominant Hessian Eigenvalues



Net 32	**Affine**	**KL Div**	**No Q**	**Scale**
Layer 1	3.679	3.673	3.673	3.691
Layer 2	1.579	1.586	1.583	1.584
Net 64	**Affine**	**KL Div**	**No Q**	**Scale**
Layer 1	4.731	4.731	4.753	4.698
Layer 2	2.322	2.322	2.326	2.335
Net 128	**Affine**	**KL Div**	**No Q**	**Scale**
Layer 1	3.787	3.777	3.779	3.777
Layer 2	1.819	1.818	1.819	1.814

Figure 13: Dominant Hessian Eigenvalues

Across the 3 Nets, Layer 2 always has a smaller dominant eigenvalue than Layer 1. Given that all eigenvalues are positive, this suggests that we are looking at a locally convex landscape, which reinforces the loss landscape plots generated above. Between the quantization schemes and No Q, there does not seem to be any noticeable variation in the eigenvalues up to 2 significant figures.

5. Conclusion

This paper investigated how loss landscape of a neural network changes from before to after quantization through using information from the Hessian. Among the 3 quantization schemes experimented with (Affine, Scale, KL Divergence), there does not seem to be any clear, “better” scheme that will always work. Rather, the results suggest that quantization schemes may have to be tried out on a per-layer basis, and different schemes may have to be chosen for different layers. Among the 3 shallow neural networks (32x32, 64x64, 128x128) experimented with, the larger ones tended to originally converge to a lower loss/higher accuracy, but also had a sharper loss landscape. This suggests that full convergence during training may not be optimal as it can cause sensitive parameters that can lead to network instability after quantization.

6. Further Work

One possible extension to this project could be to consider total Eigen Spectral Density of the weight matrices before and after quantization to see if there is any significant shift in distributions. The condition number could also be taken into account as it might suggest greater sensitivity to quantization.

Another possible extension would be to investigate a 3D loss landscape similar to [3] and [4] where the weight matrix is perturbed across the 2 dominant eigenvectors of the Hessian. Such information would better illustrate the convexity/concavity of the landscape and may lead to more illustrative results on how quantization does fundamentally alter the landscape.

7. References

[1] Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. https://arxiv.org/pdf/1712.05877.pdf

[2] Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. https://arxiv.org/pdf/2004.09602.pdf

[3] PyHessian: Neural Networks Through the Lens of the Hessian. https://arxiv.org/pdf/1912.07145.pdf

[4] Visualizing the Loss Landscape of Neural Nets. https://arxiv.org/pdf/1712.09913.pdf