Efficient Inference in Deep Learning

1st Sebastien Perre
Under the supervision of George Tzanetakis
UVic Department of Computer Science
University of Victoria
Victoria, Canada
sebastienperre@uvic.ca

***Abstract*—This project looks at how we can make deep learning models run faster and more efficiently, especially when resources are limited. I explore methods such as quantization and pruning, which reduce model size and speed up inference. Along the way, I build an oracle model that represents the best possible performance for adaptive inference and compare it to a real classifier that learns to choose the right model for each input.**

***Index Terms*—Deep learning, efficient inference, quantization, pruning, model compression, neural networks, PyTorch, computer vision, image classification, adaptive inference, model selection, oracle model, structured pruning, unstructured pruning, static quantization, dynamic quantization, model evaluation, training optimization, VGG, CIFAR 10, ImageNette, ImageWoof, hardware-aware optimization**

Part I: Learning Overview

I. Introduction

I have only used PyTorch a couple of times, so plenty of learning had to be done before working on the research. The learning consisted of reading a book, articles, and documentation, as well as watching lecture series and various tutorials. The learning included PyTorch for the implementation, general deep learning topics, and specific content related to the research (ex. convolutional layers used for image classification).

II. Deep Learning Framework

For the deep learning framework, I decided to go with the popular Python package PyTorch. It has the ability to quickly create deep learning models using the nn module. This is due to the well-abstracted functions. Figure 2 illustrates how quickly you can create a neural network.

Fig. 1: PyTorch Logo. Taken from [1]

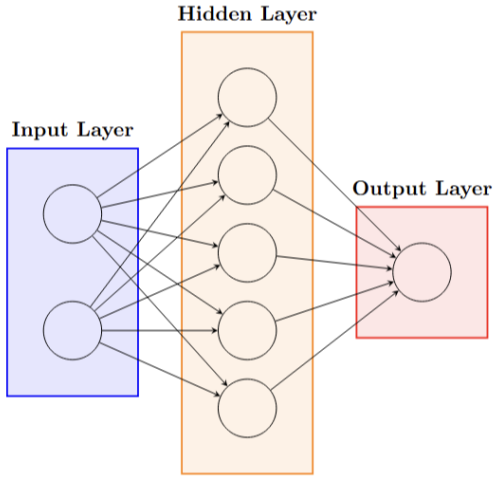
.

Not only known for its well-abstracted neural network classes, but its also known forvits breadth of activation functions, layers, training code, testing code, integration with NumPy, and community support. The biggest reason for utilizing PyTorch is its support with Nvidia's GPUs, which accelerates training significantly as we will see in Part II: Section IV.

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = nn.Linear(in_features = 2, out_features = 5)
        self.layer_2 = nn.Linear(in_features = 5, out_features = 1)

    def forward(self, x):
        return self.layer_2(self.layer_1(x))
```

(a) Example PyTorch code showcasing how easy it is to create deep neural networks. This one is a two layer linear neural network with no activation functions.



(b) Drawing of the neural network described in (a).

Fig. 2: Code and drawing showing how you can easily create neural networks.

III. Youtube Course: PyTorch 101 Crash Course For Beginners in 2025!

To learn how to use PyTorch, I used an introductory video from Zero to Mastery to learn PyTorch [2]. The video covered a lot of topics, starting with foundational concepts of neural networks and deep learning, then guides through practical implementations, including building, training, and evaluating models for classification and computer vision tasks. The last section were PyTorch tools to work with custom datasets.

A. Ch. 0 - Fundamentals

The first chapter starts by discussing the fundamentals and reasons for using PyTorch, followed by an in-depth tutorial on tensors (a PyTorch datatype). The tutorial consisted of various tensor functions and instructions on how to put tensors onto a GPU to benefit from the acceleration. To give a brief overview of what a tensor is, it can be thought of as a matrix that can take on dimension. Check Figure 3 for the tensor representation of an image.

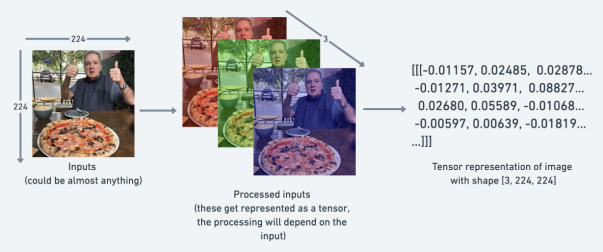


Fig. 3: Example of an image represented as a tensor. Taken from [3]

.

B. Ch. 1 - Workflow

The next chapter goes through creating a linear regression model. The chapter outlines a workflow that explains how to create models by training them, testing them, and checking how well they do. Furthermore, it goes through various loss functions and optimizers (of course, doing a straight linear regression on the model will give a better result, but the purpose is to learn how to train and test models). Check Figure 4 for the PyTorch workflow.

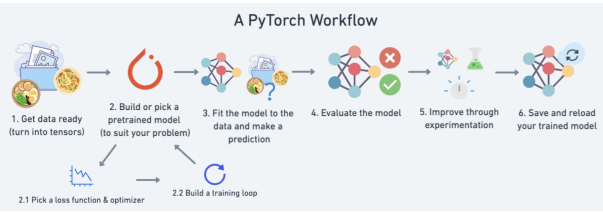


Fig. 4: A Pytorch Workflow. Taken from [3]

.

C. Ch. 2 - Neural Network Classification

In Chapter 2, it explains how to train models in a classification setting. The tutorial goes through how to create a PyTorch model that classifies in both binary and multiclass settings. The primary problem I looked at in this section is binary classifying the points in Figure 5. To deal with the non-linear structure, the tutorial guided us to add ReLU (an activation function). The chapter went through various activation functions such as ReLU, SoftMax, and Sigmoid. Finally, the chapter concludes by extending classification to the multiclass setting and introducing various metrics to test model performance (Check Table **??** for metrics and their descriptions).

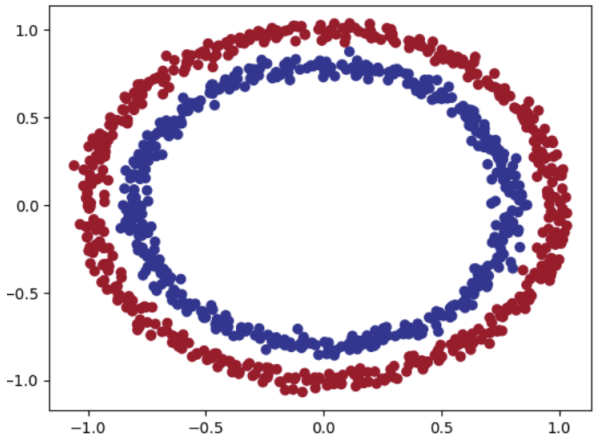


Fig. 5: The Classification Problem I Solve in Ch. 2.

.

Metric Name / Method	Definition
Accuracy	Proportion of total correct predictions over all predictions: Accuracy $= \frac{TP+TN}{TP+TN+FP+FN}$
Precision	Proportion of correctly predicted positive observations to the total predicted positives: Precision $= \frac{TP}{TP+FP}$
Recall	Proportion of correctly predicted positives to all actual positives: Recall $= \frac{TP}{TP+FN}$
F1-Score	Harmonic mean of precision and recall: F1 $= 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$
Confusion Matrix	A table showing true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), used to describe the performance of a classifier.
Classification Report	A summary of precision, recall, F1-score, and support for each class.

TABLE I: Classification Evaluation Metrics

D. Ch. 3 - Computer Vision

Chapter 3 was crucial as my research pertains primarily to computer vision. This chapter focused on training a model to classify different items of clothing in the Fashion MNIST dataset (Check Figure 6 for samples of the dataset). The chapter introduced a Dataloader to load images in batches (Loading images in batches is necessary as the image datasets take a lot of space and cannot fit into a GPU). The chapter also introduced more loss functions such as cross-entropy loss, convolutional neural networks, pooling layers, and saving and loading models.

E. Ch. 4 - Custom Datasets

The final chapter I watched, I learned how to use custom datasets and a variety of PyTorch libraries' for audio, vision, text, and recommendation. They also explained how to visualize loss curves, compare models and do data augmentation.

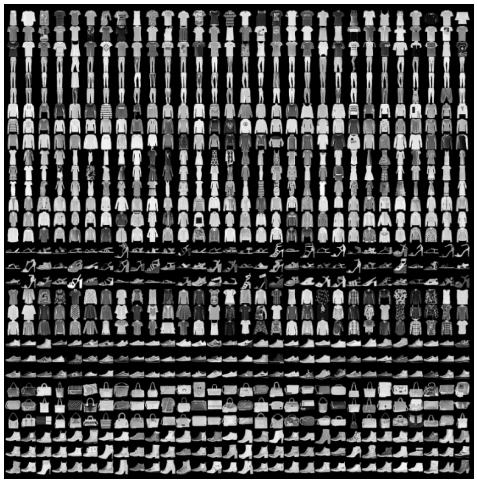


Fig. 6: Samples from the Fashion MNIST dataset. Taken from [3]

.

IV. Book: Deep Learning

To get more of a theoretical understanding ofdeep learning, I went through the first twelve chapters of "Deep Learning" by Ian Goodfellow, Yoshua Bengio and Aaron Courville [4]. The first twelve chapters included the math behind machine learning, deep feedforward networks, regularization, optimization, convolutional networks, recursive and recurrent nets, practical methodology, and applications.

A. Mathematics Behind Machine Learning

Chapters 1-5 covered the introduction, mathematical concepts required for deep learning, and machine learning basics.

TABLE II: Chapter Overview

Chapter #	Subject
1	Introduction
2	Linear Algebra
3	Probability and Information Theory
4	Numerical Conditioning
5	Machine Learning Basics

TABLE III: First five chapters of Deep Learning [4]

1) Introduction: The introduction covered the motivation for deep learning, its place in relation to Artificial Intelligence (Check Figure 7), and the history of deep learning.

2) Linear Algebra: The Linear Algebra chapter begins with scalars, vectors, matrices, and tensors. It covered matrix and vector multiplication, along with identity and inverse matrices, which are important to solving equations and simplifying computations.

It covered linear dependence, span, and norms, which helped me understand special matrix types such as symmetric and orthogonal matrices.

More advanced topics included eigendecomposition and singular value decomposition (SVD), both of which break down matrices into simpler components to reveal their structure. The

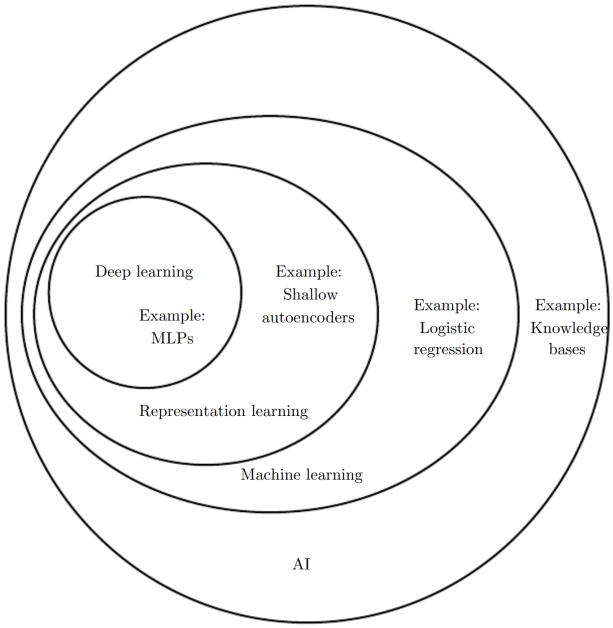


Fig. 7: AI Venn Diagram. Taken from [4]

.

Moore-Penrose pseudoinverse was introduced as a method to handle cases where a normal inverse does not exist, while the trace and determinant gave us ways to summarize important matrix properties.

The chapter concluded with Principal Component Analysis (PCA), which is a way to reduce data complexity while keeping the most important patterns in the data.

3) Probability and Information Theory: The Probability and Information Theory chapter explained how to use math to work with uncertainty, which is a large component of machine learning. It started by showing why probability is important, especially when the data is noisy or incomplete.

It introduced random variables and probability distributions, discussed marginal and conditional probability, along with the chain rule, which helps me break down complicated probabilities into smaller parts.

It introduced the ideas of independence and conditional independence, which are useful when we want to simplify how variables are connected. I also looked at expectation, variance, and covariance, which are ways to understand how variables behave and how they relate to each other.

It also introduced Bayes' Rule, which helps us update what we know when new information arises. Later, it discussed continuous variables, and then moved into information theory. This part showed how to measure uncertainty and compare different probability distributions using tools such as entropy and KL divergence.

The chapter ended by looking at structured probabilistic models, which are helpful for showing how different variables in a system are connected in a clear and organized way.

4) Numerical Conditioning: The Numerical Computation chapter explained how computers do math and why that is

important when training machine learning models. It started with overflow and underflow, which happens when numbers get too large or too small for a computer to manage, leading to strange results.

Next, I learned about poor conditioning, which means small changes in input can cause big changes in output. This makes some problems harder to solve accurately.

A big part of the chapter focused on gradient-based optimization, which is the main way we train models by adjusting parameters to reduce error. It explains how gradients work and how they help us move in the right direction during learning.

This chapter also introduced constrained optimization, which is when we want to find the best solution but have to follow certain rules or limits.

It ended with a practical example using linear least squares, which is a common method for fitting a line to data by minimizing the difference between predictions and actual values.

5) Machine Learning Basics: The Machine Learning Basics chapter introduced how machines learn from data. It starts by explaining what learning algorithms are and how they improve by seeing more examples.

It introduced capacity, overfitting, and underfitting. These ideas helped me understand when a model is too simple, too complex, or just right for the task. The chapter also covered the importance of hyperparameters and how to use validation sets to tune them and avoid overfitting.

Next, it explained estimators, along with bias and variance, which helps measure how good a learning method is. I also looked at maximum likelihood estimation, which is a common way to find the best parameters for a model by making the observed data as likely as possible.

The chapter introduced Bayesian statistics, which gives me a way to update beliefs as we get more data. Then we compared supervised learning algorithms, which learn from labeled data, with unsupervised learning algorithms, which find patterns in data without labels.

We also learned about stochastic gradient descent, which is a faster way to train models by updating parameters a little bit at a time using random samples.

Finally, the chapter goes through the steps on how to build a machine learning algorithm, and concluded by explaining some of the big challenges in machine learning.

B. Deep Feedforward Networks

The Deep Feedforward Networks chapter introduced one of the most important building blocks in deep learning: the feedforward neural network. It started with a classic example—learning the XOR function—to show how simple models like linear classifiers fall short, and how deeper networks can solve more complex problems.

The chapter then looked at gradient based learning. Gradient based learning is the main method used to train these networks by adjusting weights to reduce error. One core concept in deep networks is the use of hidden units, which allow the model to learn and represent complex patterns in the data.

Later in the chapter, it delves into architecture design, showing how the number of layers and units affects the network's ability to learn. One of the results in this section showed how deeper models perform better than shallower models even if they have the same number of parameters (Check Figure 8 for the result).

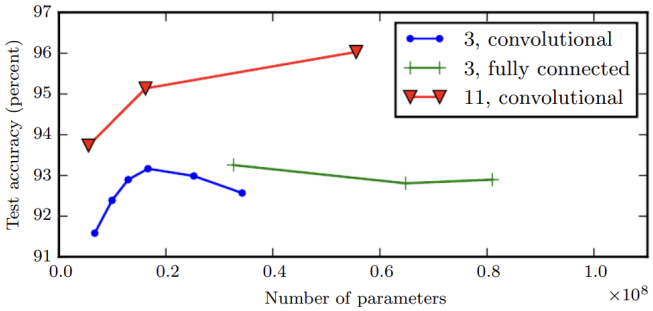


Fig. 8: Graph illustrating the benefit of having deeper models. Taken from [4]

.

Then came backpropagation, the algorithm that makes training deep networks efficient by computing gradients quickly and accurately.

It concludes with some historical notes that gave context on how these networks evolved and became a major part of modern machine learning.

C. Regularization

The Regularization chapter focused on how to help deep learning models perform well on new and unseen data—not just the training data. It started with parameter norm penalties, which add extra terms to the loss function to keep the model's weights small and prevent overfitting.

These penalties were also explained as a type of constrained optimization, where we limit how complex the model can be. Further on in the chapter it then discussed how regularization helps in under-constrained problems, where there are many possible solutions and you want to choose the simplest one that works.

We learned about techniques like dataset augmentation, where we created extra training data by slightly changing the original data, and noise robustness, which helps models handle noisy or imperfect inputs. The chapter also covered semi-supervised learning and multi-task learning, which let the model learn better by sharing information across tasks or using unlabeled data.

Other strategies included early stopping, where training is stopped before the model starts overfitting, and parameter tying and sharing, which reduce the number of unique weights in a model. Check Figure 9 to see early stopping in action.

The chapter also introduced sparse representations, which encourage the model to use fewer active features at once.

Ensemble methods such as bagging were discussed as ways to combine multiple models for better performance. A popular

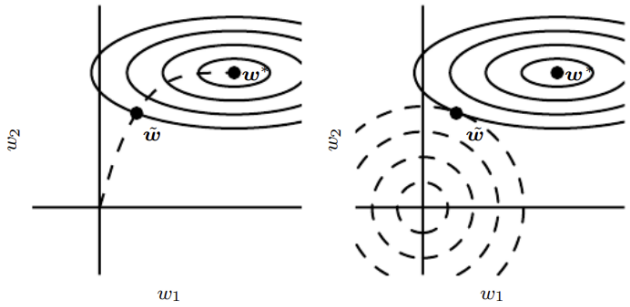


Fig. 9: Graph illustrating early stopping. You see the model stops at a contour and not at the optimal solution w^* to avoid overfitting the data. Taken from [4]

.

and powerful technique called dropout was also explained, it works by randomly turning off parts of the network during training to make the model more robust.

The chapter ended with advanced ideas like adversarial training, where the model learns to resist small but tricky changes to inputs, and methods like tangent distance, tangent prop, and the manifold tangent classifier. This helps the model focus on meaningful changes in the data.

D. Optimization

The chapter on Optimization for Training Deep Models looked at how to train neural networks by adjusting their parameters to improve performance. It starts by explaining how learning is different from regular optimization in deep learning. It explains how you are not just finding a single best solution, but trying to generalize well to new data.

The chapter then highlighted some of the challenges that come up when optimizing neural networks, for instance getting stuck in bad local minima, saddle points, or dealing with vanishing and exploding gradients.

The chapter proceeded with some basic algorithms including stochastic gradient descent, and then looked at parameter initialization strategies that help start training in a good place so the model can learn faster and more reliably. This section also introduced the idea of momentum which aims to solve two problems of stochastic gradient descent (ill-conditioning and variance). Check Figure 10 for momentum in action

Next, it introduced algorithms with adaptive learning rates, for example, AdaGrad, RMSProp, and Adam, which automatically adjust how quickly the model learns based on past updates. It also touched on approximate second-order methods, which try to use curvature information to guide optimization more effectively, though they are more complex and harder to scale.

Finally, the chapter highlights different optimization strategies and meta-algorithms. These are higher-level ideas for improving training, which include combining methods, using restarts, or switching strategies during training to get better results.

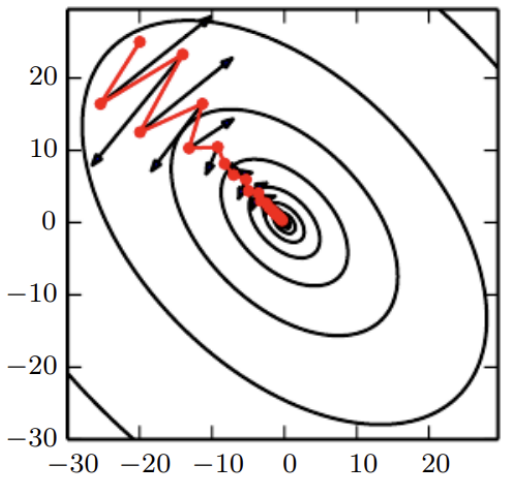


Fig. 10: Graph illustrating momentum. The red line indicates the time steps using momentum while the black arrows represent if the model took steps from the original stochastic gradient descent. Taken from [4]

.

E. Convolutional Networks

The chapter on Convolutional Networks introduced a powerful type of neural network that works well with images. It starts by explaining the convolution operation, which allows the network to scan across input data (like an image) to detect patterns including edges, textures, or shapes (Check Figure 11 for an example of convolution applied to a 2D tensor).

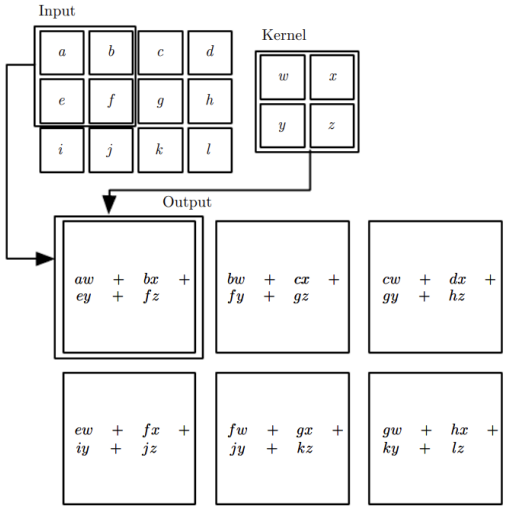


Fig. 11: Example of a convolution applied on a 2D tensor. Taken from [4]

.

Then I looked at the motivation behind using convolutional networks. They reduce the number of parameters and take advantage of patterns that appear in different parts of an image. Later on, the chapter introduced pooling which is

a technique that helps simplify the information in feature maps by summarizing nearby values, making the model more efficient and robust.

It explained how convolution and pooling act like strong built-in assumptions, helping the model focus on local patterns that are useful in many tasks. The chapter also described different variants of the basic convolution function, including tweaks that can improve flexibility or performance.

Later sections in the chapter covered advanced topics which included generating structured outputs, handling different data types, and using efficient algorithms to speed up convolution operations. It also highlighted the use of random or unsupervised features to improve performance when labeled data is limited.

The chapter touched on the neuroscientific inspiration behind convolutional networks, explaining how ideas from the human visual system helped shape their design. Finally, it placed convolutional networks in the context of the history of deep learning, showing how they helped drive major breakthroughs in the field.

F. Recursive and Recurrent Nets

The Sequence Modeling chapter delved into how to work with data that comes in a sequence, like text, speech, or time series. It started by explaining unfolding computational graphs, which helps visualize how a model handles sequences step by step over time.

Then I learned about recurrent neural networks, or RNNs, which are designed to process sequences by passing information from one step to another. The chapter introduced bidirectional RNNs, which look at a sequence from both directions to get a better understanding of the context.

Next is encoder-decoder architectures. These are used in tasks such as translation where I need to turn one sequence into another. I also looked at deep recurrent networks, which stack multiple RNN layers to learn more complex patterns, and recursive neural networks, which work on structured data like trees instead of flat sequences.

One big challenge discussed was long-term dependencies. This is where the model needs to remember data from far back in the sequence. To help with this, the chapter explores ideas with echo state networks and leaky units, which are designed to track information over different time scales.

A major solution to this problem is the long short-term memory network, or LSTM, along with other gated RNNs. These models use special gates to control the flow of information and help keep important details alive over time.

The chapter finishes with techniques for optimizing long-term dependencies, and introduced the idea of explicit memory, where models are given separate memory components to store and retrieve information more effectively.

G. Practical Methodology

The Practical Methodology chapter focused on how to apply deep learning in real-world projects. It started with performance metrics, showing how to choose the right way to measure how well a model is doing based on the task.

Next, it discussed the importance of using default baseline models—simple models that give us a quick sense of how hard the problem is and whether a more complex model is worth the effort. The chapter also advised on deciding whether to collect more data, which can often be more helpful than changing the model itself.

Then I learned about hyperparameter selection, including tips on how to choose learning rates, model sizes, and other settings that affect performance. The book provided a list of hyperparameter examples along with how it affects model capacity in Figure 12 (Model capacity in layman's terms is how well the model can learn complex patterns) The chapter also covered debugging strategies to help figure out why a model is not working as expected and how to fix common issues.

Hyperparameter	Increases capacity when...	Reason	Caveats
Number of hidden units	increased	Increasing the number of hidden units increases the representational capacity of the model.	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model.
Learning rate	tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure	
Convolution kernel width	increased	Increasing the kernel width increases the number of parameters in the model	A wider kernel results in a narrower output dimension, reducing model capacity unless you use implicit zero padding to reduce this effect. Wider kernels require more memory for parameter storage and increase runtime, but a narrower output reduces memory cost.
Implicit zero padding	increased	Adding implicit zeros before convolution keeps the representation size large	Increased time and memory cost of most operations.
Weight decay coefficient	decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

Fig. 12: Example hyperparameters and how they affect model capacity. Taken from [4]

.

It ended with a real-world example on multi-digit number recognition, showing how all these strategies come together in practice—from evaluating performance and tuning hyperparameters to solving problems during development.

H. Applications

The Applications chapter showed how deep learning is used in real-world problems across many different fields. It starts with large-scale deep learning, explaining how companies and

researchers train huge models using powerful hardware and lots of data.

Then the chapter looked at computer vision, where deep learning is used to recognize images, detect objects, and understand scenes. This has led to big improvements in areas such as facial recognition, self-driving cars, and medical imaging.

Later in the chapter, it covered speech recognition, showing how deep models can turn spoken words into text with impressive accuracy, even in noisy environments.

In natural language processing, deep learning helps computers understand and generate human language. This includes applications which include translation, chatbots, and text summarization.

At the end of the chapter, it highlights other applications, such as healthcare, robotics, finance, and game playing. These examples show how far deep learning has reached and how flexible it is across different types of problems.

V. Other

I used a variety of other resources to get a better understanding of PyTorch and deep learning. These included various articles, videos, and PyTorch documentation.

Part II: Experimental Outcomes[1]

I. Introduction

Can I make inference faster in deep neural networks (DNNs)? It is a fair question, especially as models like large language models continue to grow in size and require more power to run. For example, OpenAI reportedly spent over 2.5 million dollars just to showcase the capabilities of GPT 4 on various benchmarks. At that scale, power efficiency is not just a bonus, it becomes a necessity. Even a 10 percent improvement in efficiency could save a company hundreds of thousands of dollars. When Deepseek entered the scene claiming major gains in inference speed and open sourcing their work, it sent a clear message to the industry. The markets responded, showing how critical optimization has become. Deepseek clearly pushed the limits of what is possible. So the question of how to improve inference performance is not just interesting, it is necessary. I explored multiple methods for the sake of efficiency.

II. Datasets Used

I used a variety of datasets to test the efficiency of my methods which are CIFAR-10, ImageNet, ImageNette, and ImageWoof.

[1]All code can be found at https://github.com/sebperre/efficient-inference-in-dl

A. *CIFAR-10*

CIFAR-10 is a dataset of 60000 32 by 32 color images in 10 classes (Check Figure 37 for some sample images). The 60000 images are split into 50000 training images and 10000 testing images. Furthermore, there are 6000 images for each class. Finally, there is no overlap between the classes (this is mentioned due to the classes of automobiles and trucks [5]).

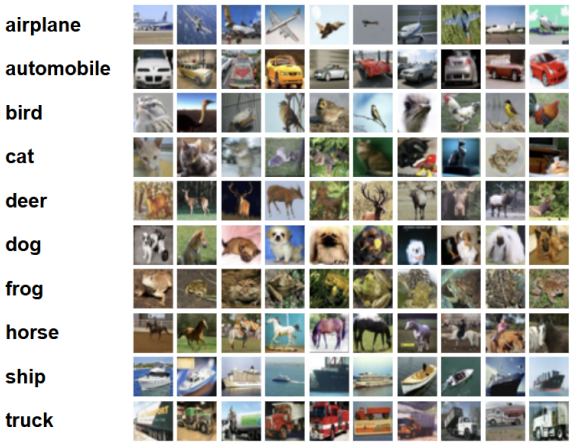


Fig. 13: Sample images from the CIFAR-10 dataset across 10 object categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each row shows 10 random examples from one class. Taken from [5]

.

An up close image is presented in Figure 14.

Fig. 14: Random Image taken from CIFAR-10 Dataset

.

B. *ImageNet*

I used the ImageNet-1K dataset for my experiments instead of the more expressive ImageNet-21K. The original ImageNet dataset consists of 14,197,122 images. As I do not have access to major computing power, I decided to go with the scaled down Kaggle version with 38700 images. Note: These images come in various different sizes so they have to be transformed before being put into the model [6]. Check Figure 15 for samples of ImageNet

Fig. 15: Random Image taken from CIFAR-10 Dataset

.

C. ImageNette

The third dataset I used is ImageNette. This is an even smaller subset of ImageNet with 13394 total images across 10 classes. Check Table IV for the classes and their distribution.

Class Name	Sample Count
Tench	1350
English Springer	1350
Cassette Player	1350
Chain Saw	1244
Church	1350
French Horn	1350
Garbage Truck	1350
Gas Pump	1350
Golf Ball	1350
Parachute	1350

TABLE IV: Sample Counts for ImageNette Classes

Check Figure 16 for an example image from the ImageNette dataset.

Fig. 16: An example of a French Horn in the ImageNette dataset.

.

D. ImageWoof

The final dataset I decided to use is ImageWoof, which is also a subset of ImageNet, consists of 13394 images across 10 classes. Check Table V for the classes and their distribution and Figure 17 for an example of the class. Compared to ImageNette, this will be harder for a deep learning model to learn as ImageNette has classes which have no overlap while ImageWoof has overlap. This is due to the fact that the purpose of ImageWoof is to classify dog breeds.

Class Name	Sample Count
Shih	1350
Rhodesian Ridgeback	1350
Beagle	1350
English Foxhound	804
Border Terrier	1350
Australian Terrier	1350
Golden Retriever	1350
Old English Sheepdog	1350
Samoyed	1350
Dingo	1350

TABLE V: Sample Counts for ImageWoof Classes

Fig. 17: Sample image from the ImageWoof dataset. This is a Golden Retriever.

.

III. Setting Up a Cloud GPU

After doing a couple of rounds of training and testing with the GPU on my computer, I realized my computer lagged whenever I tried to run any sort of code. This was an issue as I wanted the ability to train my models for a long period of time without it affecting the performance of my computer, or training multiple models one after another. To resolve this issue, I decided to setup my own cloud computing by using my desktop computer with a Nvidia GTX 1060 6GB graphics card. To check the performance, please refer to Part II: Section IV.

A. Arch Linux

To ensure ease of use and flexibility for my system, I decided to go with Linux and the Arch distro. The choice for Arch is simple, I wanted to have a distro which was bare bones so I can build my system from the ground up.

I did the setup by booting Arch, installing necessary packages, Gnome (a UI interface), Nvidia drivers, etc. Once I had the system in place, I moved on to security measures before proceeding with setting up my system to be able to be accessed from anywhere.

Fig. 18: Arch Linux Logo. Taken from [7]

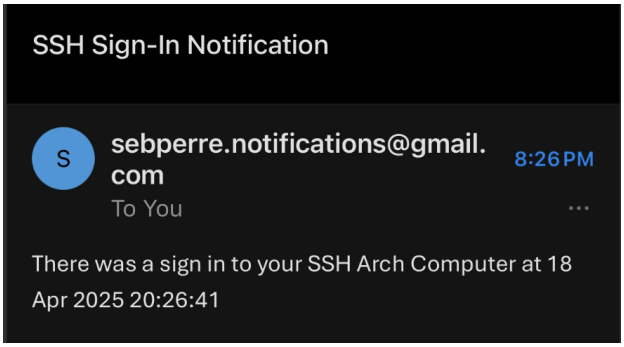
.

B. Security

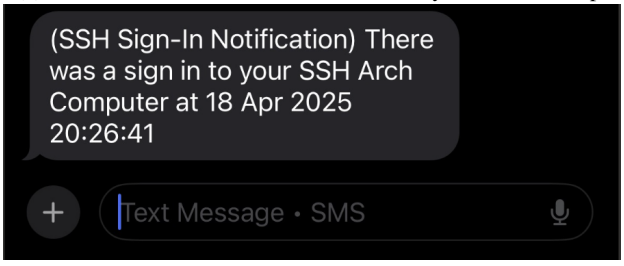
As I wanted to setup the SSH protocol to access my remote Desktop from anywhere, I needed security measures so only myself can access the device. To do this, I implemented two extra security measures in addition to my password, multi-factor authentication and notifications.

1) Multi-Factor Authentication: I setup Google Authenticator through my phone, which was a hassle as Arch did not have a streamlined process to do this. Furthermore, I had to link Google Authenticator to SSH. This required me to go into multiple configuration files and tinker around.

2) Notifications: If someone does manage to access my system, I will be notified immediately thanks to a custom alert system I set up. A Python script, triggered at the end of the SSH process, sends both a text message and an email to alert me. Figure 19 shows examples of these notifications in action.



(a) Email Notification when I SSH into my remote desktop.



(b) Text Notification when I SSH into my remote desktop.

Fig. 19: Notifications when I SSH into my remote desktop.

C. Port Forwarding

To access my remote desktop from anywhere, I configured port forwarding on my router to allow external SSH connections. I also assigned a static IP to the desktop to ensure the port forwarding settings remain consistent and do not need to be updated each time the device reconnects.

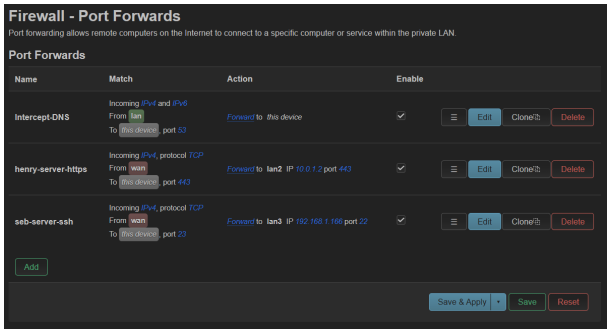


Fig. 20: Port forwarding settings on my router.

.

D. Run Background Shell Script

When you run a Python script on a remote machine, the process is tied to the user session, meaning it will stop if the user disconnects. To prevent this, I prefixed the command with *nohup*, which tells the system to ignore the *SIGHUP* signal that is sent when a user logs out. Additionally, disowning the process ensures it continues running independently of the user session. As I wanted to run my scripts when I am offline, I implemented this using a bash script.

IV. Testing Time Differential between CPU and GPU

To start off, I did tests on the performance on both my computer and my remote desktop. Check Table VI for the results.

TABLE VI: Training Time (in seconds) for 5 Epochs. Laptop: AMD Ryzen 7 5800HS (CPU), Nvidia GeForce RTX 3050 (GPU); Remote Desktop: Intel Core i5-4460 (CPU), Nvidia GeForce GTX 1060 6GB (GPU). ImageNet, ImageNette, and ImageWoof were trained on 1000 images.

Device	Dataset (Model)	CPU (s)	GPU (s)
Laptop	CIFAR-10 (Simple CNN)	176.00	61.78
	ImageNet (ResNet-18)	288.30	38.44
	ImageNette (ResNet-18)	304.47	38.12
	ImageWoof (ResNet-18)	287.09	35.65
Remote Desktop	CIFAR-10 (Simple CNN)	176.00	61.78
	ImageNet (ResNet-18)	288.30	38.44
	ImageNette (ResNet-18)	336.92	45.95
	ImageWoof (ResNet-18)	333.77	44.92

As we can see, the table shows my laptop outperforms my remote desktop. Although, this comes at a cost as when I train my models, it significantly slows down my laptop. Moreover, I can't run it all night as my laptop automatically sleeps after a certain period of time. Therefore, even though it is slower, my desktop serves as a good machine to train my models.

V. Very Deep Convolutional Networks (VGG)

Before getting into the experiments, it's worth taking a moment to talk about VGG networks, since they show up a lot

in the sections that follow. VGGs are a type of convolutional neural network known for their depth and simplicity. They stack many convolutional layers using small 3×3 filters, with a consistent structure that makes them easy to work with. VGG models helped show that simply going deeper with more layers can significantly improve performance on image classification tasks. Check Figure 21 for a visualization of a VGG.

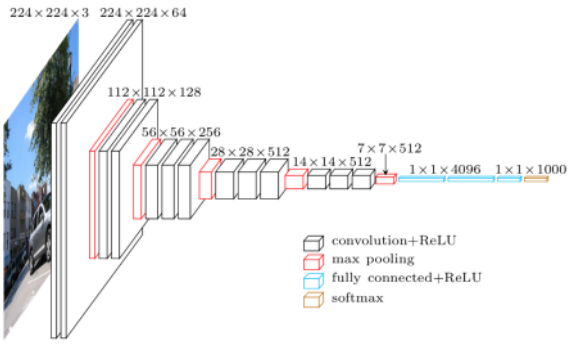


Fig. 21: VGG Visualization. Taken from [8]

.

A. *Kaiming Initialization*

As networks like VGG get deeper, they can run into problems during training, like gradients vanishing or blowing up; Kaiming initialization was designed to help with this. It sets the initial weights based on the number of inputs to a layer, which helps keep the scale of activations steady as they move through the network. This makes training more stable, especially when using ReLU activations.

VI. Testing Model Performance

A. *CIFAR-10*

I use two models, Simple CNN and VGG-16, on 20 epochs

1) Simple CNN: The simple CNN is a convolutional neural network consisting of two convolutional layers with ReLU and max pooling, followed by a fully connected layer and an output layer for classification. Check Table VII for the metrics after training for 20 epochs and Figure 22 for the training loss per epoch.

Metric	Value
Training Time	4m 23s
Accuracy	0.7220
Precision	0.7261
Recall	0.7220
F1 Score	0.7218

TABLE VII: Training Results for CIFAR-10 using Simple CNN (20 Epochs)

In the training loss figure, the training loss significantly reduces until epoch 9 and minimal improvements are made from there. Furthermore, our metrics are in the 72% range.

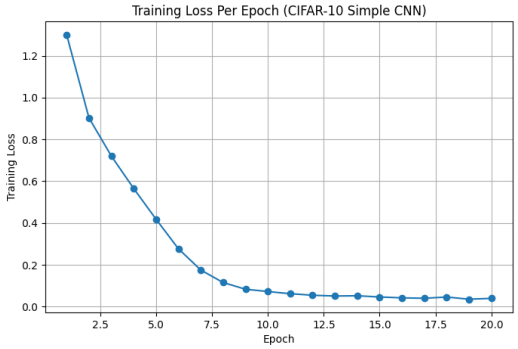


Fig. 22: Training loss per epoch on CIFAR-10 with a simple CNN.

.

Metric	Value
Training Time	22m 4s
Accuracy	0.7791
Precision	0.7989
Recall	0.7791
F1 Score	0.7837

TABLE VIII: Training Results for CIFAR-10 using VGG-16 (20 Epochs)

2) VGG-16: The VGG-16 is a very deep convolutional neural network consisting of 16 layers. Check Table VII for the metrics after training for 20 epochs and Figure 23 for the training loss per epoch.

From the graph, we see the training loss keeps going lower and lower with each epoch getting a lower training loss differential. We see the metrics are in the 78% range which is better than our ResNet model.

B. *ImageNet*

I use three models, pretrained VGG-11, VGG-16 and ResNet-18 on 20 epochs.

1) Pretrained VGG Model: The pretrained VGG-11 has the highest metrics of around 82% with a training loss graph significantly decreasing and sometimes even gaining loss on

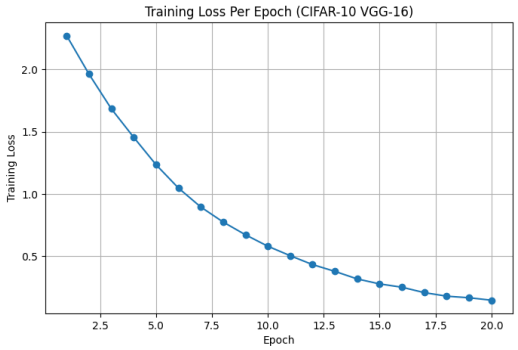


Fig. 23: Training loss per epoch on CIFAR-10 with VGG-16.

.

some epochs (epoch 13). Check Figure 24 for the training loss and Table IX for the metrics.

Metric	Value
Training Time	30m 1s
Accuracy	0.8254
Precision	0.8349
Recall	0.8254
F1 Score	0.8271

TABLE IX: Training Results for CIFAR-10 using Pretrained VGG-11 (20 Epochs)

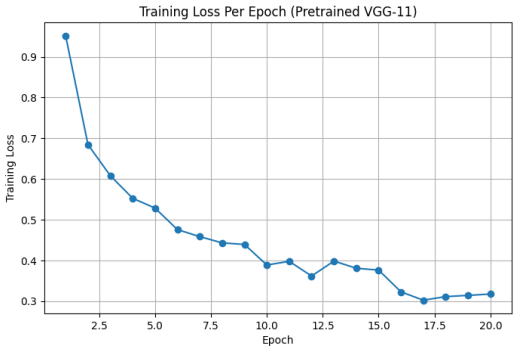


Fig. 24: Training loss per epoch on ImageNet with a pretrained VGG-11

.

2) VGG-16: I trained the VGG-16 model on 3000 images of the ImageNet dataset for the speed. We can see the model got the training loss down from 7 to 4.5, but the metrics indicated the model poorly generalized to the data. Check Figure 25 and Table X

Metric	Value
Training Time	35m 15s
Accuracy	0.0003
Precision	0.0003
Recall	0.0003
F1 Score	0.0003

TABLE X: Training Results for ImageNet using VGG-16 (20 Epochs, 3,000-Image Subset)

3) ResNet-18: The final model I trained for ImageNet is ResNet-18 and I trained it on all the data for 20 epochs. Just like the VGG model, it failed to generalize evident by XI. Although, it was to get the training loss down to practically zero (and it was stagnent). The training loss being almost 0 and the model failing to generalize is a clear sign of overfitting.

Metric	Value
Training Time	1h 50m 30s
Accuracy	0.0719
Precision	0.0826
Recall	0.0719
F1 Score	0.0648

TABLE XI: Training Results for ImageNet using Simple ResNet (20 Epochs, Full Dataset)

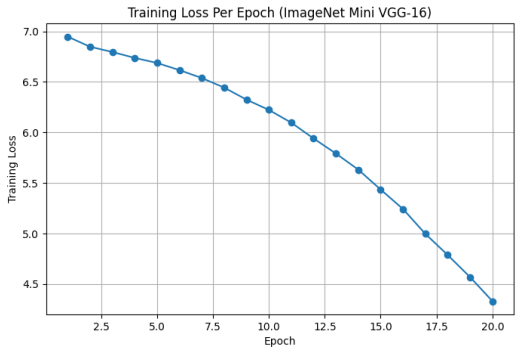


Fig. 25: Training loss per epoch on ImageNet with VGG-16.

.

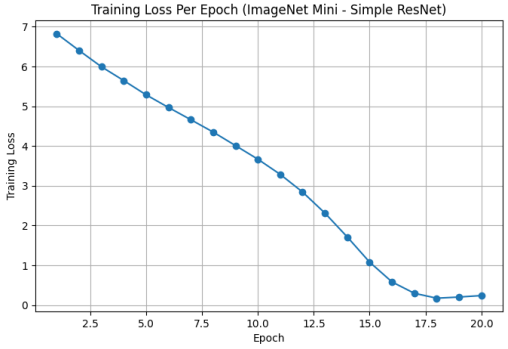


Fig. 26: Training loss per epoch on ImageNet with ResNet-18.

.

With ImageNet being hard to train on, I decided not to use ImageNet anymore.

C. ImageNette

The only model I used to test the ImageNette dataset was VGG-16.

1) VGG-16: From Table XII, it is apparent that our model generalized to ImageNette better than it did to ImageNet. Figure 27 trend indicates we can train the ImageNette model even more and get a lower training loss which could give us better metrics on the testing data.

Metric	Value
Training Time	35m 19s
Accuracy	0.3767
Precision	0.4124
Recall	0.3767
F1 Score	0.3551

TABLE XII: Training Results for ImageNette using VGG-16 (20 Epochs, 3,000-Image Subset)

D. ImageWoof

1) VGG-16: ImageNette consists of images which don't overlap in ImageNet which gives motivation to the ImageWoof dataset which is a dataset of 10 dog breeds and is harder

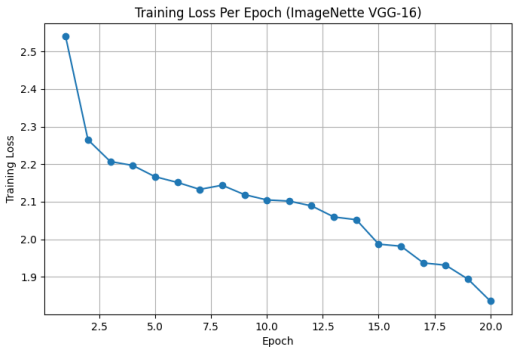


Fig. 27: Training loss per epoch on ImageNette with VGG-16.

.

to classify. We see this hypothesis holds true as Table XIII shows worse metrics than our ImageNette results. Just like ImageNette, Figure 28 indicates we can train ImageWoof for longer and get better training loss.

Metric	Value
Training Time	34m 58s
Accuracy	0.2130
Precision	0.1926
Recall	0.2130
F1 Score	0.1504

TABLE XIII: Training Results for ImageWoof using VGG-16 (20 Epochs, 3,000-Image Subset)

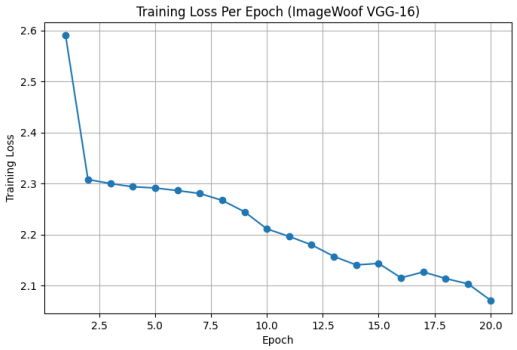


Fig. 28: Training loss per epoch on ImageWoof with VGG-16.

.

VII. Oracle Model

For efficient inference we want to be able to speed up the inference time with sacrificing as little accuracy as possible. One idea is to train multiple models with each successive model being stronger than the previous (stronger is meant as more parameters and layers), calculate the maximum accuracy loss which is acceptable and figure out a way to decide which sample should be sent to which model. Before proceeding with the second part, it is important to figure out what the best case scenario is. To figure out the best case scenario, we can utilize an oracle which is the best case scenario (aka. we are able to successfully determine which sample should go to which model. This can be achieved by creating a dictionary of labels to models).

A. Implementation

The implementation is rather simple; I trained five tiny VGGs to use as the models to choose from. Check Table XIV for an in-depth view of the models.

Depth	Layer Configuration	# Conv	Channels per Block	Flattened
1	[64, Pool]	1	64	16,384
2	[64, 64, Pool]	2	64	16,384
3	[64, 64, Pool, 128, 128, Pool]	4	64, 128	8,192
4	[64, 64, 64, Pool, 128, 128, 128, Pool]	6	64, 128	8,192
5	[64, 64, 64, Pool, 128, 128, 128, Pool, 256, 256, 256, Pool]	9	64, 128, 256	4,096

TABLE XIV: Summary of Custom VGG Architectures for CIFAR-10. Each configuration uses 3×3 convolutions with padding and ReLU activations. All models end with a classifier consisting of two hidden layers of size 512 and a final output layer for classification. Flattened feature size is taken before the first linear layer.

There is also an accuracy sacrifice variable that determines how much accuracy we are willing to sacrifice for efficiency.

B. Results

1) CIFAR-10: Check Figure 29 for the total prediction overlap. This shows how many samples a model predicts correctly the same as another model (The label specific heatmaps located in the Appendix are used for determining what model to use). Looking at the figure closely, it shows every model is around the same in terms of predictions.

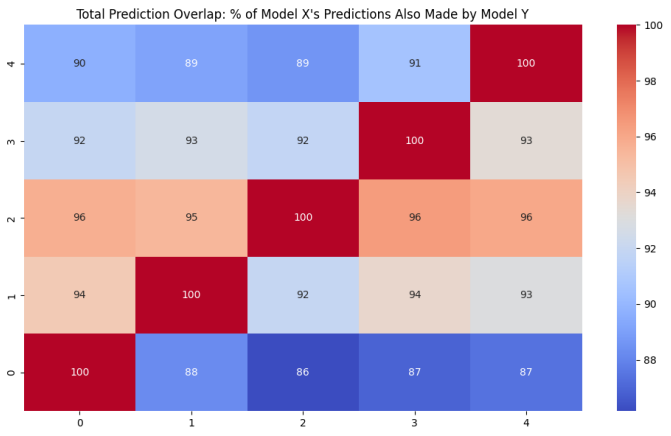


Fig. 29: The total prediction overlap between models.

.

For the key performance metrics in Figure 30, the combined model performs better than any singular model. This is great as the model got better metrics with less inference time (a double win!).

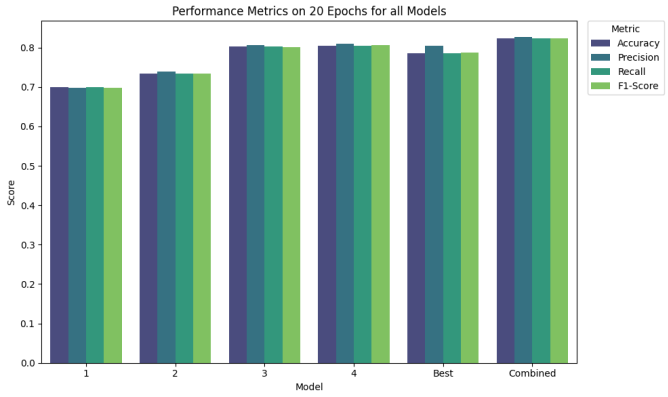


Fig. 30: Key Performance Metrics of the tiny VGGs as well as the combined model.

.

In Figure 31, the inference time on the testing set takes around the average of model 3's and 4's inference time. Thus, we were able to achieve some inference speed gains.

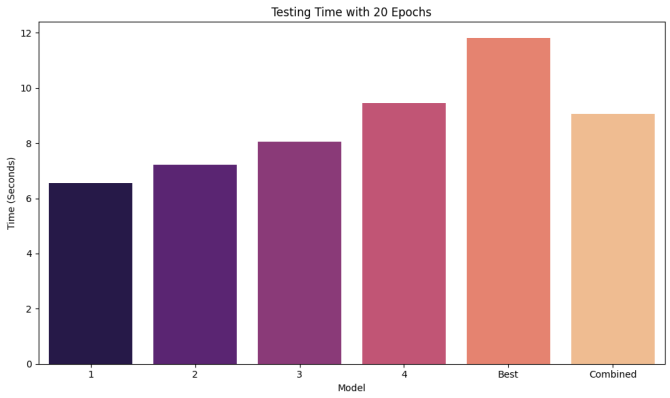


Fig. 31: The time it took for the models to go through the testing sample.

.

In Figure 32, it was expected the training time for each of the models increased as the models got more complex (layers, size, etc).

Finally, Figures 33 and 34 show that as inference time decreases, key performance metrics also decline. While sacrificing more accuracy does lead to faster inference, the improvements taper off, showing clear diminishing returns.

2) ImageNette and ImageWoof: Running the oracle model requires loading multiple models and images onto the GPU, which causes an Out of Memory error. Figure 35 shows the error message that appears when this happens. Thus, we can't proceed with using our oracle or efficient classification model.

VIII. Classification Model

A. Implementation

The implementation is exactly the same as Section VII, but the oracle is replaced with a CNN model

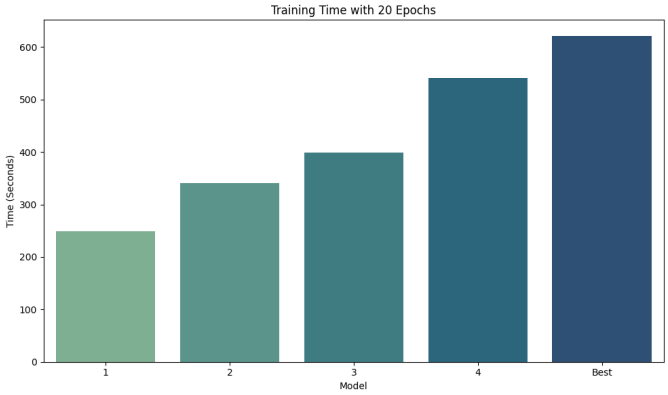


Fig. 32: The time it took to train the models on 20 epochs.

.

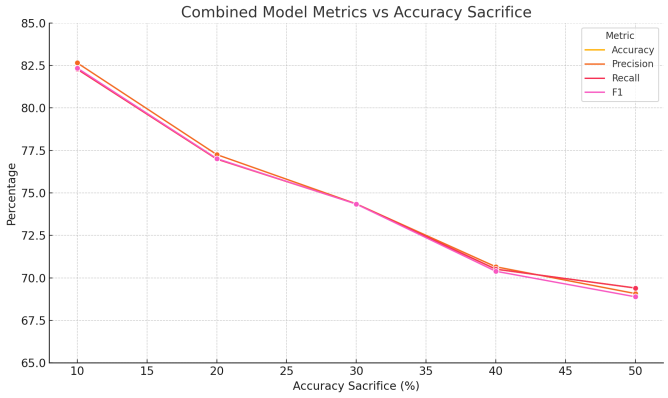


Fig. 33: A multi-line chart showing how much the key performance metrics suffer with each percent in accuracy sacrifice.

.

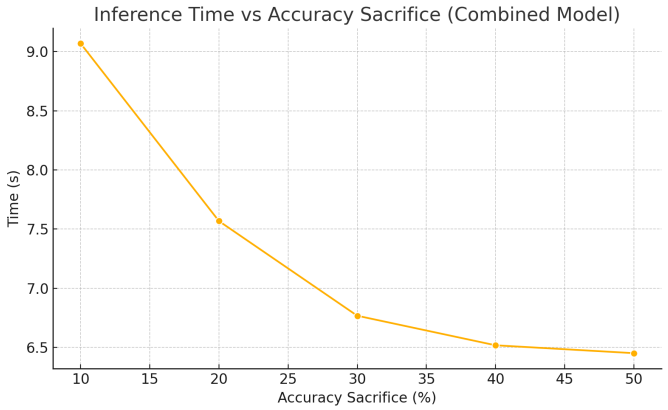


Fig. 34: How much better the inference time takes on the testing sample for each percent sacrifice in accuracy.

.

```
torch.OutOfMemoryError: CUDA out of memory. Tried to all
ocate 400.00 MiB. GPU 0 has a total capacity of 5.92 GiB
 of which 264.25 MiB is free. Process 2158 has 37.08 MiB
 memory in use. Including non-PyTorch memory, this proce
ss has 5.62 GiB memory in use. Of the allocate~/prog~/pr
```

Fig. 35: Out of Memory Error message whenever I try running the Oracle Model on ImageNette and ImageWoof

.

B. Results

1) CIFAR-10: From Table XV we see that the introduction of a tiny CNN drastically affects the time for the combined model. The reason this occurs is because our models are so small that finding a method which predicts what model the sample should be sent to leads to worse inference time. One can imagine if the inference task takes multiple minutes such as a large language model then it will be able to reap the benefits.

Statistic	Best	Combined	Difference
Accuracy (%)	82.38	81.45	0.93
Precision (%)	82.4832	81.3891	1.0941
Recall (%)	82.38	81.45	0.93
F1 (%)	82.37	81.3779	0.9921
Time (s)	11.359	58.8768	-47.5178

TABLE XV: Comparison of Best vs Combined Results Across Multiple Metrics

2) ImageNette and ImageWoof: The same issue occurs for Section VII-B2.

IX. Quantization

Quantization is a way to make deep learning models smaller and faster by using lower-precision values. Instead of storing and processing everything with high-precision 32-bit floats, we use lower-precision values like 8-bit integers. This means the model takes up less space and runs more efficiently. The trade-off is that you might lose a bit of accuracy, but in many cases, the speed savings are well worth it.

A. Post-Training Quantization

1) Static Quantization: Static quantization converts model weights and activations to lower precision ahead of time using calibration data. The model first runs on a sample dataset to measure the range of values, and then those ranges are used to quantize everything. It's accurate and efficient, but needs access to representative data.

2) Dynamic Quantization: Dynamic quantization only converts the weights ahead of time. Activations are quantized on-the-fly during inference. It's easier to apply since it doesn't need calibration data, and works well for models with lots of linear layers like transformers.

3) Weight-Only Quantization: Weight-only quantization reduces the precision of just the model's weights, leaving activations in full precision. It's simple and still gives some memory and bandwidth savings, making it a lightweight option when full quantization isn't needed.

B. Other Types of Quantization

1) Quantization-Aware Training
   Simulates quantization during training so the model learns to adapt to the reduced precision.
2) Very-Small Quantization
   Uses extremely low bit-widths (like 4-bit or even binary) for weights and activations. It can dramatically shrink models, but often at the cost of accuracy.
3) Mixed-Precision Quantization
   Different parts of the model use different bit-widths depending on sensitivity to quantization. This balances efficiency and performance by assigning lower precision where it's safe and higher precision where needed.

X. Pruning

Pruning is a technique to remove unnecessary weights or structures from a neural network in order to reduce its size and improve computational efficiency, often with minimal impact on accuracy. We start with the lottery ticket hypothesis which was a big inspiration for pruning.

A. Lottery Ticket Hypothesis

The Lottery Ticket Hypothesis proposes that inside a large randomly initialized neural network, there exists a smaller subnetwork that can be trained to reach similar accuracy as the full model. This smaller network, often called a winning ticket, is found by training the full model, pruning the least important weights, resetting the remaining weights to their original values, and retraining. Surprisingly, these sparse subnetworks can perform just as well, even with far fewer parameters. The idea shifted how researchers think about pruning, showing that overparameterized models may not be necessary if the right sparse structure is found early. While the original method is not practical for large models due to the need for full training, the concept has inspired new strategies like pruning at initialization and dynamic sparse training, pushing pruning research in new and more efficient directions.

B. Unstructured Pruning

Unstructured pruning removes individual weight connections in the network without regard to any structural grouping. The goal is to create a sparse weight matrix by deleting the least important weights. The key challenge is determining which weights are unimportant.

1) Magnitude-Based Pruning (L1 Norm, L2 Norm): Magnitude-based pruning is a simple and commonly used method to reduce the size of a neural network. It removes the weights with the smallest absolute values, based on the idea that smaller weights contribute less to the model's predictions. By setting a threshold or pruning a fixed percentage, any weight below that value is set to zero. This method is easy to apply since it only looks at the weight values and does not require complex calculations. It often maintains model accuracy well when combined with retraining, especially if done gradually. However, it creates an unstructured pattern of zeros, which can be hard to accelerate on standard hardware.

It also assumes that small weights are unimportant, which is not always true, and selecting the right threshold can take some tuning. Despite these challenges, it remains a strong and effective pruning technique.

2) Gradient-Based Pruning: Gradient-based pruning removes weights based on how important they are to the model's performance, measured by their gradients. The idea is that if changing a weight has little effect on the loss, it can be safely removed. During training, the gradients of the weights are tracked, and those with the smallest impact on the loss are pruned. This method takes into account not just the size of the weight but how sensitive the model is to it. It can be more accurate than simple magnitude-based approaches, but it is also more complex and requires more computation.

C. Other Types of Pruning

1) Structured Pruning
   This method removes entire structures like filters or layers, making the resulting model more efficient and easier to run on standard hardware.
   a) Filter Pruning
      Removes full convolutional filters, reducing the number of output channels and cutting down on computation.
   b) Channel Pruning
      Eliminates specific input channels to a layer, decreasing the input size and simplifying computations in later layers.
   c) Layer Pruning
      Removes entire layers from the model, reducing its depth and overall complexity.
   d) Block Pruning
      Deletes larger functional blocks like residual modules, offering significant savings while preserving structural clarity.
2) Neural Architecture Search (NAS)-based Pruning
   NAS-based pruning uses automated search algorithms to explore different sparse model architectures. It finds efficient subnetworks tailored to performance and resource constraints.
3) Hardware-Aware Pruning
   This method considers the underlying hardware (like GPUs, CPUs, or edge devices) during pruning. The goal is to maximize real-world speedup, not just reduce parameter count.
4) Dynamic/Online Pruning During Training
   Weights are pruned as the model trains, adapting to the learning process rather than pruning after training ends. This can lead to better sparse models without retraining from scratch.
   a) Gradual Pruning
      Weights are slowly removed over time during training, allowing the model to adjust and maintain accuracy.
   b) Dynamic Sparse Training (DST)
      The model maintains a fixed sparsity level but dynamically rewires its sparse connections throughout training to keep learning capacity high.
5) Learning-Based Pruning (Trainable or Meta-Learned)
   These methods learn which weights to prune through optimization or training, often outperforming hand-designed rules.
   a) L0 Regularization
      This adds a term to the loss function that encourages sparsity by penalizing the number of active weights, leading to learned pruning decisions.
   b) AutoML/Reinforcement Learning-Based Pruning
      An automated agent learns pruning strategies by exploring the trade-off between model size and performance, often using reward signals.
   c) Bayesian Pruning
      Uses probabilistic models to estimate the importance of each weight or unit, pruning those with low uncertainty-based relevance.

D. Implementation

For my implementation of pruning, I decided to go with L1 pruning. Although, I should have started with the research above first as L1 pruning does not work well for convolutional networks as I use standard hardware to do the computation. This evident by Figure 36. Interestingly enough, we don't see a major decrease in accuracy until after we prune 60% of the weights indicating they were not necessary for the model's performance.

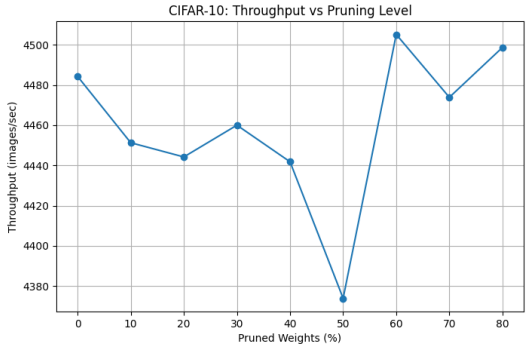


Fig. 36: Throughput vs. Pruning Level on CIFAR-10.

.

XI. Extending the Research

A. More Implementations

One way to extend this research is to introduce more implementations in the context of quantization and pruning. It would be interesting to quantitatively determine how effective each of the quantization and pruning techniques are.

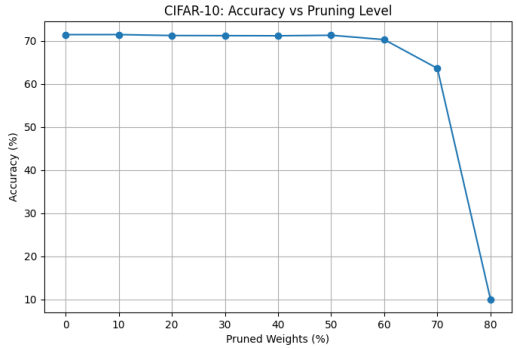


Fig. 37: Accuracy vs. Pruning Level on CIFAR-10.

.

B. Fixing the Oracle and Efficient Inference Models for ImageNette and ImageWoof

As my devices aren't powerful enough to handle creating big models for ImageNette and ImageWoof, a worthwhile endeavor would be to invest in better hardware to see how well the oracle and classification model do.

C. Combining Methods

Another way to extend the research is to combine multiple of the methods mentioned above and see how the accuracy and inference time is affected.

D. Research Efficient Inference for Different Types of Problems and Models

I only looked at these methods in the case of Image Classification. It would be interesting to see these methods and results in the context of regression or other sort of problems.

XII. Acknowledgments

I would like to thank Professor George Tzanetakis for his patience, time and supervision.

Appendix

References

[1] D. Wei. (2024, Feb.) Learning pytorch: The basic program structure. [Online]. Available: https://medium.com/@weidagang/learning-pytorch-the-basic-program-structure-ed5723118b67

[2] Zero To Mastery. (2024, Dec.) Pytorch 101 crash course for beginners in 2025! [Online]. Available: https://www.youtube.com/watch?v=LyJtbe__2i0

[3] D. Bourke. (2025) Zero to mastery learn pytorch for deep learning. [Online]. Available: https://www.learnpytorch.io/

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016. [Online]. Available: https://www.deeplearningbook.org

[5] A. Krizhevsky, "The cifar-10 dataset," https://www.cs.toronto.edu/~kriz/cifar.html.

[6] I. Figotin, "Imagenet 1000 (mini)," https://www.kaggle.com/datasets/ifigotin/imagenetmini-1000, n.d., kaggle dataset.

[7] (2025) Arch linux logos and artwork. [Online]. Available: https://archlinux.org/art/

[8] G. Boesch. (2021) Very deep convolutional networks (vgg) essential guide. [Online]. Available: https://viso.ai/deep-learning/vgg-very-deep-convolutional-networks/

[9] Jeremy and the fastai community, "Imagenette: A subset of 10 easily classified classes from imagenet," https://github.com/fastai/imagenette, 2020.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[11] Wikipedia contributors. (2025) Pytorch. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/PyTorch

[12] ——. (2025) Imagenet. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/ImageNet

[13] Stanford Vision Lab and Princeton University. (2020) Imagenet. ImageNet. [Online]. Available: https://www.image-net.org/

[14] Zalando Research and . collaborator, "Fashion mnist," https://www.kaggle.com/datasets/zalando-research/fashionmnist, 2017.

[15] H. Shen, N. Mellempudi, X. He, Q. Gao, C. Wang, and M. Wang, "Efficient post-training quantization with fp8 formats," 2024. [Online]. Available: https://arxiv.org/abs/2309.14592

[16] PyTorch Team. (2024) Quantization — pytorch 2.2 documentation. [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[17] (2025) Pytorch. [Online]. Available: https://pytorch.org/

[18] ArchWiki contributors. (2025) Google authenticator. [Online]. Available: https://wiki.archlinux.org/title/Google_Authenticator

[19] Arch Linux Developers. (2025) Arch linux. [Online]. Available: https://archlinux.org/

[20] P. Team. (2023) Post training quantization with torch-tensorrt. [Online]. Available: https://pytorch.org/TensorRT/tutorials/ptq.html

[21] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations," 2024. [Online]. Available: https://arxiv.org/abs/2308.06767

[22] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1803.03635

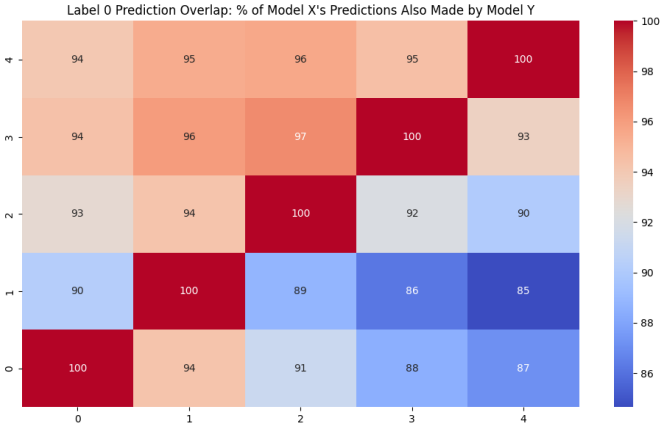
[23] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 5, p. 2900–2919, May 2024. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2023.3334614

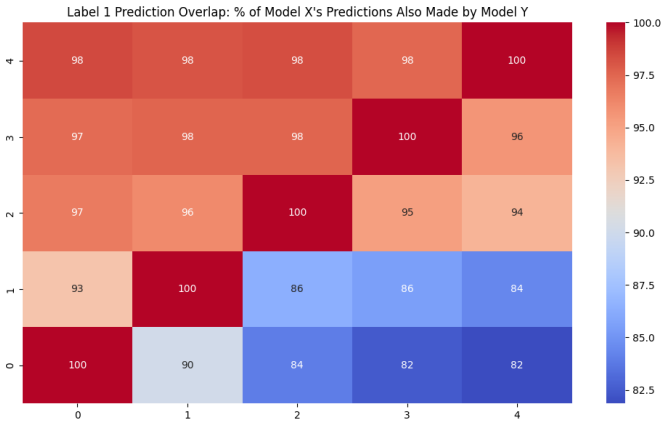
[24] S. Vadera and S. Ameen, "Methods for pruning deep neural networks," *IEEE Access*, vol. 10, pp. 1–1, 01 2022.

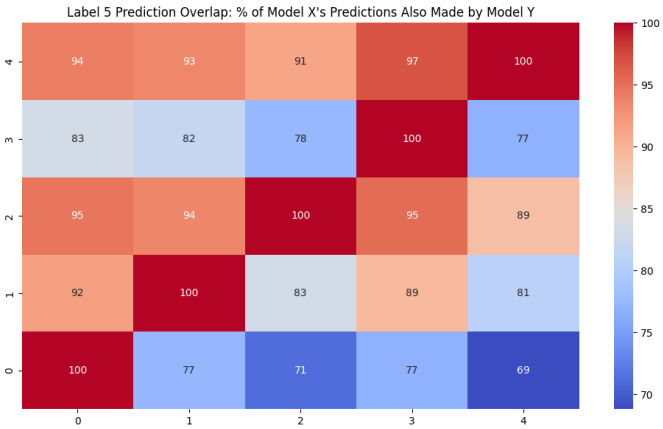
[25] GeeksforGeeks. (2024, Jun.) Vgg-net architecture explained. [Online]. Available: https://www.geeksforgeeks.org/vgg-net-architecture-explained/

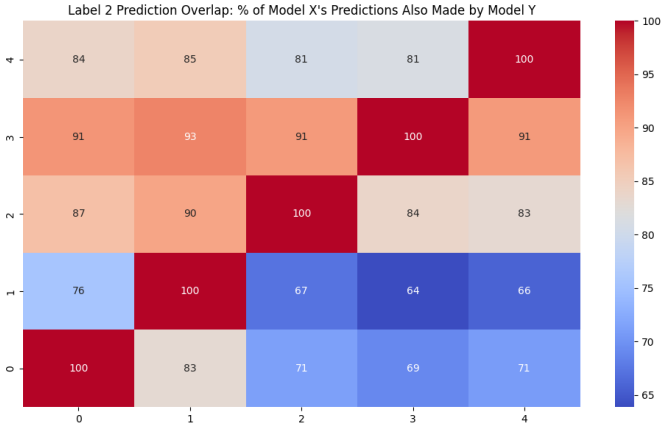
Extra Figures

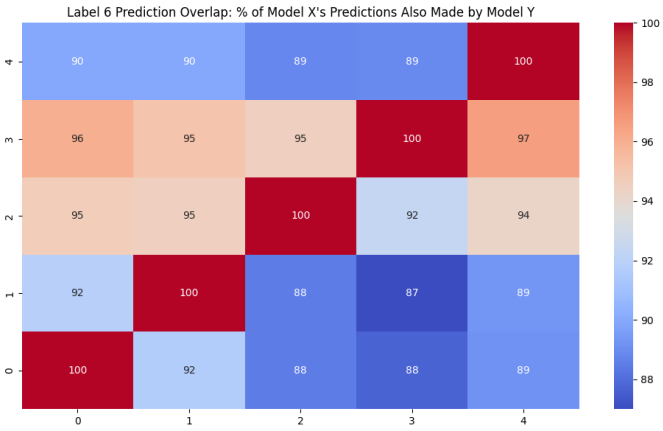
These extra figures are for Section VII. They show the overlap in correct predictions for specific labels.

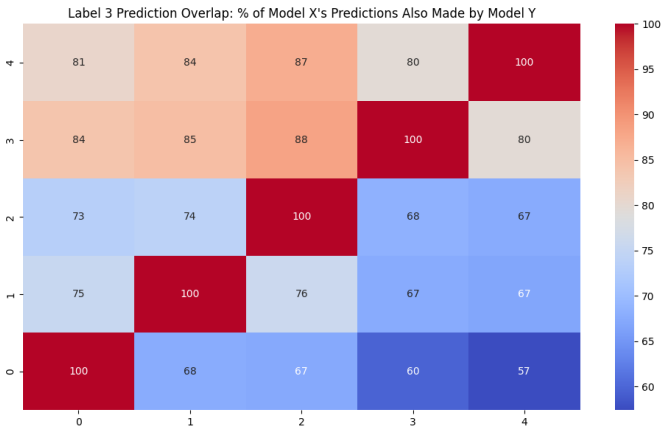


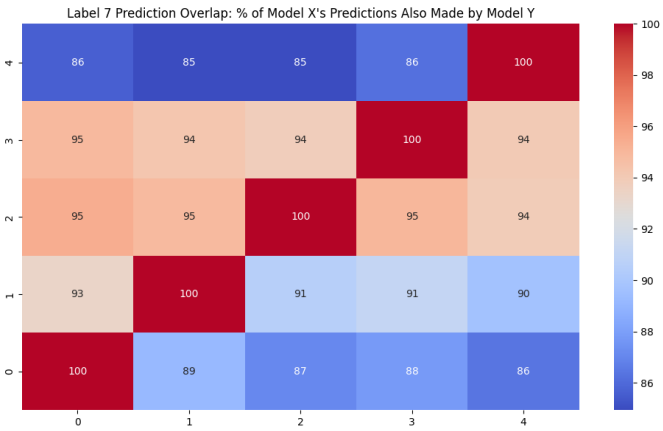

Label 1 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
98 98 98 98 100
97 98 98 100 96
97 96 100 95 94
93 100 86 86 84
100 90 84 82 82
0 1 2 3 4
100.0 97.5 95.0 92.5 90.0 87.5 85.0 82.5

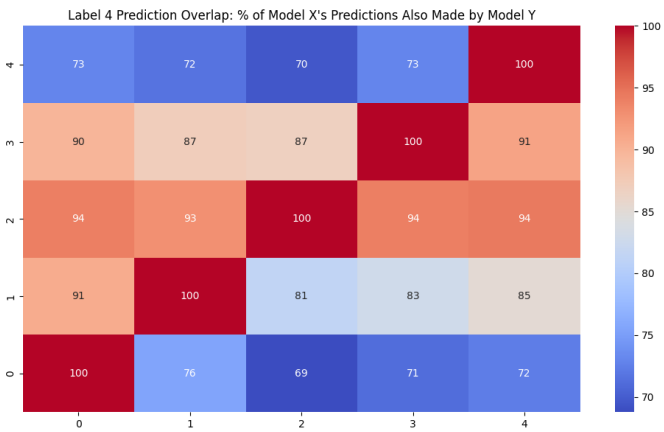

Label 5 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
94 93 91 97 100
83 82 78 100 77
95 94 100 95 89
92 100 83 89 81
100 77 71 77 69
0 1 2 3 4
100 95 90 85 80 75 70

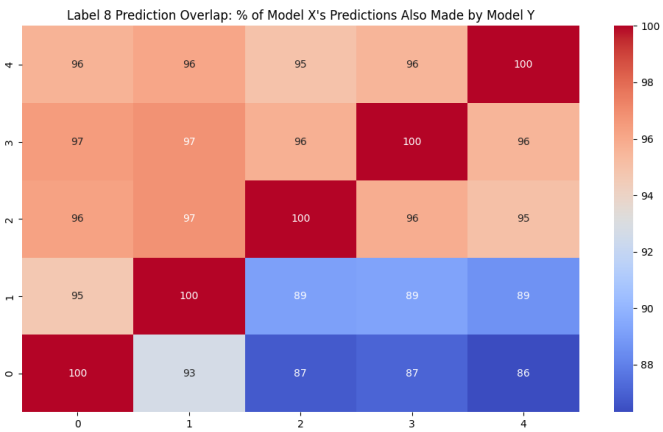

Label 2 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
84 85 81 81 100
91 93 91 100 91
87 90 100 84 83
76 100 67 64 66
100 83 71 69 71
0 1 2 3 4
100 95 90 85 80 75 70 65

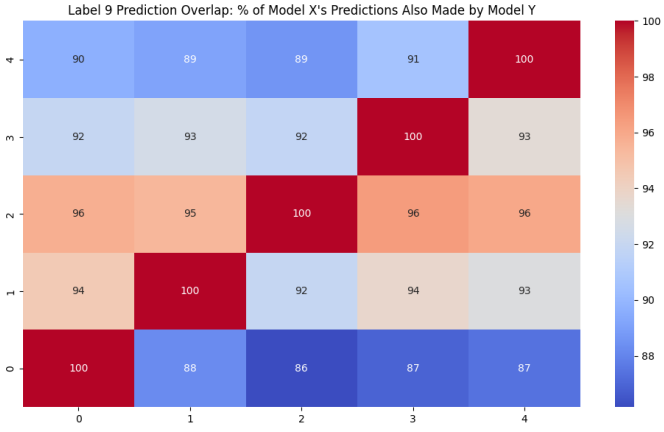

Label 6 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
90 90 89 89 100
96 95 95 100 97
95 95 100 92 94
92 100 88 87 89
100 92 88 88 89
0 1 2 3 4
100 98 96 94 92 90 88


Label 3 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
81 84 87 80 100
84 85 88 100 80
73 74 100 68 67
75 100 76 67 67
100 68 67 60 57
0 1 2 3 4
100 95 90 85 80 75 70 65 60


Label 7 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
86 85 85 86 100
95 94 94 100 94
95 95 100 95 94
93 100 91 91 90
100 89 87 88 86
0 1 2 3 4
100 98 96 94 92 90 88 86


Label 4 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
73 72 70 73 100
90 87 87 100 91
94 93 100 94 94
91 100 81 83 85
100 76 69 71 72
0 1 2 3 4
100 95 90 85 80 75 70


Label 8 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
96 96 95 96 100
97 97 96 100 96
96 97 100 96 95
95 100 89 89 89
100 93 87 87 86
0 1 2 3 4
100 98 96 94 92 90 88


Label 9 Prediction Overlap: % of Model X's Predictions Also Made by Model Y
4
3
2
1
0
90
89
89
91
100
92
93
92
100
93
96
95
100
96
96
94
100
92
94
93
100
88
86
87
87
0
1
2
3
4
100
98
96
94
92
90
88