BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

WEIGHT INITIALIZATION IN NEURAL NETWORKS

NEURONOVÉ SÍTĚ, PŘEDTRÉNOVÁNÍ KOMPLEXITY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **RICHARD KOCIÁN**
AUTOR PRÁCE

SUPERVISOR **Ing. KAREL FRITZ**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

162825

Institut:	Department of Computer Systems (DCSY)
Student:	**Kocián Richard**
Programme:	Information Technology
Title:	**Weight Initialization in Neural Networks**
Category:	Artificial Intelligence
Academic year:	2024/25

Assignment:

1. Study the fundamentals of neural networks, focusing on their architecture and learning principles.
2. Focus on the importance of weight initialization and how different initialization methods can affect training efficiency.
3. Propose various weight initialization strategies based on task type (e.g., classification, regression) and network complexity.
4. Implement the proposed strategies and test them on selected tasks of varying complexity.
5. Conduct experiments to observe the impact of initialization on convergence speed and neural network stability.
6. Analyze the results and suggest possible extensions of the project.

Literature:

- According to the instructions of the project supervisor.

Requirements for the semestral defence:

- Completion of items 1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Fritz Karel, Ing.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	31.10.2024

Abstract

This thesis investigates reversed model distillation for neural networks, where a smaller teacher model guides the early training of a larger student model. The goal is to evaluate whether this approach can serve as an effective weight initialization method and improve training dynamics. Experiments on classification tasks (CIFAR-10 and Fashion-MNIST) show that applying distillation throughout the entire training improves accuracy and robustness to FGSM attacks. For the regression task (California Housing), while final accuracy did not improve, the method led to lower test loss under adversarial conditions. These results suggest that reversed distillation can support more stable and robust neural network training.

Abstrakt

Tato práce se zabývá technikou obrácené destilace modelu neuronových sítí, při které menší model (učitel) vede počáteční fázi tréninku většího modelu (studenta). Cílem je ověřit, zda tento přístup může sloužit jako efektivní způsob inicializace vah a zlepšit průběh tréninku. Experimenty na klasifikačních úlohách (CIFAR-10 a Fashion-MNIST) ukazují, že destilace po celou dobu tréninku zvyšuje přesnost a odolnost vůči FGSM útokům. U regresní úlohy (California Housing) sice nedošlo ke zlepšení výsledné přesnosti, ale metoda vedla ke snížení testovací ztráty při adversariálním útoku. Výsledky naznačují, že obrácená destilace může podpořit stabilnější a odolnější trénink neuronových sítí.

Keywords

Neural Networks, Weight Initialization, Task Complexity, Pre-training, Convergence, Local Minima, Adaptive Initialization, Spectral Properties, Model Performance, Network Stability, Reversed Model Distillation, FGSM Attack

Klíčová slova

Neuronové sítě, Inicializace vah, Komplexita úlohy, Předtrénování, Konvergence, Lokální minima, Adaptivní inicializace, Spektrální vlastnosti, Výkon modelu, Stabilita sítě, Obrácená destilace modelu, FGSM útok

Reference

KOCIÁN, Richard. *Weight Initialization in Neural Networks.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Fritz

Rozšířený abstrakt

Tato práce se zabývá technikou obrácené destilace modelu (reversed model distillation) u neuronových sítí, která představuje méně tradiční směr v oblasti přenosu znalostí mezi modely. Zatímco klasická destilace modelů spočívá v tom, že větší a výkonnější model (učitel) pomáhá zjednodušenému modelu (student) dosáhnout podobné výkonnosti s menšími nároky na výpočetní zdroje, v této práci je tento přístup obrácen. Cílem je prozkoumat, zda může menší model (učitel) sloužit jako užitečný průvodce při trénování většího modelu (studenta). Tento přístup může být chápán jako alternativní způsob inicializace vah, kdy se namísto pouze náhodného startu použije vedení učitelem na začátku (nebo po celou dobu) tréninku studenta.

Motivací pro tuto práci je snaha o efektivnější a robustnější trénování hlubokých neuronových sítí, které jsou stále častěji nasazovány v reálných aplikacích. Komplexnější modely sice obecně dosahují vyšší přesnosti, ale jejich trénování je náročné jak z hlediska výpočetních zdrojů, tak z hlediska stability průběhu učení. Motivací pro tento přístup je myšlenka, že malý model může efektivně zachytit základní struktury v datech, které pak může větší model dále rozvíjet. Hypotézou této práce je, že vedení menším, předtrénovaným modelem může pomoci většímu modelu zvýšit jeho přesnost nebo zlepšit jeho generalizaci a být robustnější vůči různým typům narušení, například adversariálním útokům.

Experimenty byly provedeny na třech rozdílných datasetech reprezentujících jak klasifikační, tak regresní úlohy – CIFAR-10 (klasifikace barevných obrázků do 10 tříd), Fashion-MNIST (klasifikace módních obrázků ve stupních šedi) a California Housing (regrese cen nemovitostí na základě socioekonomických údajů). Pro každý dataset byly navrženy sítě ve čtyřech velikostních variantách – Small, Medium, Large a Student, které se využily pro různé učitele a pro studentský model. Reversed distillation byla implementována tak, že student během části tréninku minimalizoval kombinaci dvou ztrát: klasické (tvrdé) loss funkce vůči správným štítkům a měkké loss funkce vůči predikcím učitele. Míra důležitosti těchto ztrát byla řízena parametrem α, zatímco délka destilační fáze byla určena parametrem `switchEpoch`.

Výsledky ukazují, že použití destilace po celou dobu tréninku (tedy bez přepnutí na standardní trénink) vede v klasifikačních úlohách k vyšší přesnosti než trénink s náhodnou inicializací, a zároveň zvyšuje odolnost modelu vůči adversariálním útokům. Tato zlepšení byla konzistentní napříč oběma klasifikačními datasety a více konfiguracemi modelů. U regresní úlohy (California Housing) sice nedošlo ke zlepšení celkové přesnosti modelu, avšak při aplikaci FGSM útoku se ukázalo, že modely trénované pomocí reversed distillation vykazují nižší testovací ztrátu, a tedy vyšší odolnost vůči narušení vstupních dat. Tento výsledek naznačuje, že reversed distillation může mít smysl i mimo oblast klasifikace, a to především jako nástroj pro zvýšení robustnosti modelů.

Práce tedy ukazuje, že reversed model distillation může sloužit jako užitečný nástroj pro stabilnější a odolnější trénink neuronových sítí. Ačkoli není univerzálně přínosná pro všechny typy úloh (například u regrese nepřinesla výrazné zvýšení přesnosti), přináší zajímavé výhody v oblasti bezpečnosti modelů a jejich chování při narušení vstupních dat. Výsledky také ukazují, že klíčovou roli hraje správné nastavení parametrů destilace, zejména délka trvání této fáze. Tento přístup může být dále rozvíjen například zařazením více učitelů, adaptivním řízením váhového parametru α, nebo aplikací na rozsáhlejší a realističtější datasety. Přínos práce spočívá nejen ve zmapování této relativně málo prozkoumané oblasti, ale také v praktickém ověření efektivity tohoto přístupu na konkrétních datech, což může být inspirací pro další výzkum v oblasti trénování hlubokých modelů a jejich bezpečnosti vůči útokům.

Weight Initialization in Neural Networks

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Karel Fritz. The supplementary information was also provided by Ing. Karel Fritz. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis. Generative AI tools were used to assist with language formulation and stylistic improvements of the text, which was written independently by the author.

. . . .
Richard Kocián
May 11, 2025

Acknowledgements

I would like to thank my supervisor, Ing. Karel Fritz, for his valuable guidance, support, and for always being willing to share his knowledge throughout the work on this thesis. I'm also grateful to my family for their continuous support during my studies.

Contents

List of Figures

Chapter 1

Introduction

In recent years, deep learning has achieved remarkable success across a wide range of applications, including image recognition and natural language processing. A major factor contributing to this progress has been the increasing size and complexity of neural networks, which, while delivering improved performance, often come with significantly higher computational and memory requirements. Training large models from scratch can be both time-consuming and expensive, especially when considering the need for hyperparameter tuning, extensive data preprocessing, and the sheer volume of training data required for effective learning.

To address these challenges, the machine learning community has explored various techniques aimed at making training more efficient. One such approach is model distillation, where a compact student model is trained to mimic the behavior of a larger, pre-trained teacher model. The goal is to transfer the knowledge from the teacher to the student in such a way that the student retains most of the teacher's performance while being computationally cheaper to run. Traditionally, this process involves transferring knowledge from a more complex model to a smaller one.

However, a less-explored but potentially valuable direction is to reverse this process – that is, to use a smaller teacher model to guide the training of a larger student model. This approach, known as reversed model distillation, raises an interesting research question: Can the knowledge captured by a lightweight, efficiently trained teacher be effectively used to initialize and guide the training of a more complex student model? Instead of relying solely on random weight initialization, the student receives early guidance based on the teacher's predictions, which could help it converge faster or reach better generalization.

This thesis investigates the potential of reversed model distillation as an alternative form of weight initialization for larger models. The primary objective is to determine whether transferring knowledge from a smaller model can lead to improved training dynamics for a larger one, such as faster convergence, reduced loss variability, or improved accuracy. In addition to evaluating standard training metrics, this work also examines the impact of reversed model distillation on adversarial robustness by analyzing how the models perform under Fast Gradient Sign Method (FGSM) attacks with varying levels of perturbation.

To this end, a series of experiments are conducted across three datasets of varying complexity and domain: CIFAR-10 and Fashion-MNIST for classification tasks, and California Housing for a regression task.

Several key aspects are examined throughout the experimental design:

- The impact of the alpha parameter, which balances the use of soft labels (from the teacher) and hard labels (from ground truth) during distillation.
- The role of the switch epoch, which defines when the model transitions from knowledge distillation to standard supervised training.
- The influence of the teacher model's size on the effectiveness of the distillation process.
- The adversarial robustness of the resulting models, measured by their performance under FGSM attacks.

Each student model is trained multiple times with different random seeds to ensure the robustness of observed patterns and to analyze the stability of the training process. By comparing the performance of models trained with and without reversed distillation, this thesis aims to provide insights into when and how reversed distillation may be beneficial, and what limitations or challenges it may present.

Ultimately, the findings contribute to the broader understanding of knowledge transfer in neural networks and open several avenues for future research in model initialization and efficient training strategies.

Chapter 2

Neural Networks

Neural networks, often called artificial neural networks, are computational models inspired by the structure and functioning of the human brain. A neural network can be considered a non-linear mathematical model that transforms a set of inputs into a set of outputs. This transformation is determined by adjustable parameters known as weights and biases, which play a crucial role in the network's learning process. [3]

2.1 Artificial Neurons

Artificial neurons are at the core of a neural network, interconnected units designed to mimic the behavior of biological neurons. Each neuron receives one or more inputs, processes them by calculating a weighted sum, adds a bias, and applies a non-linear activation function to produce an output. Mathematically, the output of a single neuron can be expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.1}$$

Where y represents the output, x_i represents the inputs, w_i represents the weights, b represents the bias, and f represents the activation function. This design allows neurons to model complex relationships in the data. [3]

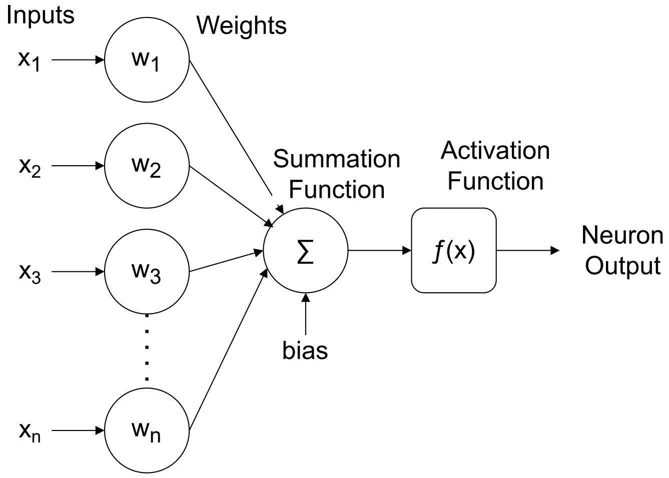


Figure 2.1: Artifical neuron

2.2 Network layers

Neural networks are organized in multiple layers. Generally, the neural network can be divided into three types of layers:

- **Input neuron layer** – takes raw data and feeds it into the network.
- **Hidden neuron layers** – lie between the input and output layers and perform the main computations, extracting features and patterns from the data.
- **Output neuron layer** – generates the network's final predictions.

Neural networks often contain multiple hidden layers to extract more complex patterns in the data. [40]

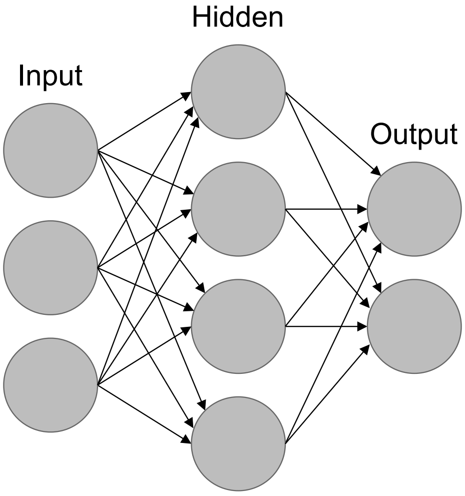


Figure 2.2: Neural Network Layers

2.3 Types of Neural Networks

There are several types of neural networks. They differ in their architecture and intended use.

1. **Feedforward neural networks** – A type of neural network in which the connections between individual neurons do not form cycles. Data flows in one direction – from the input layer through the hidden layers to the output layer. [10]
2. **Recurrent neural networks** – Neural networks with connections that form cycles, enabling them to maintain an internal state and process sequences of data, such as time series or language. [25]
3. **Transformers** – A neural network architecture introduced in the research paper called „Attention Is All You Need“, which replaces recurrence with a self-attention mechanism. This allows the model to process entire sequences in parallel and capture long-range dependencies efficiently. [39] Originally designed for machine translation, transformers have become the foundation for many state-of-the-art models in natural language processing.

4. **Convolutional neural networks** – A type of feedforward neural network primarily used for image data, inspired by the visual cortex in the brain. They use convolutional layers to extract spatial features using filters (kernels) that slide over the input and learn to detect patterns like edges or textures. Filters have dimensions corresponding to height, width, and number of channels, and their movement is controlled by a stride parameter. [23]

2.4 Learning Principles in Neural Networks

The neural network is trained using a training dataset. Training is usually done in epochs, where each epoch means one pass through the entire training dataset. The training data are typically divided into smaller batches, and after processing each batch, the weights of the neurons are updated. [13]

Thus, when a neural network is trained, learning occurs. Using backpropagation the gradient of weights is computed. Then, the optimization of weights is performed using an optimization algorithm, such as stochastic gradient descent (SGD) or more advanced methods like Adam or RMSprop. This optimization algorithm incrementally updates the weights of neurons to minimize a loss function, which measures how far the network's output is from the desired output. [13]

Neural network learning can be summarized in the following sections [13]:

1. **Forward Pass** – the input data passes through the entire neural network through all layers so that at the end of the neural network, the resulting prediction is computed.

2. **Loss Computation** – using a certain loss function, the difference between the prediction and the actual expected value is calculated.

3. **Backward Pass (Backpropagation)** – for each weight in the neural network, a gradient is calculated.

4. **Weight Update** – using a specific optimization algorithm, the weights are updated.

2.4.1 Loss Functions

Loss functions (also called cost functions or objective functions) play a crucial role in the training of neural networks. They measure how far the network's prediction is from the actual target value and serve as the objective that the optimization algorithm seeks to minimize. Choosing an appropriate loss function is essential, as it directly influences the convergence and performance of the model. A well-chosen loss function helps guide the learning process by providing meaningful gradient information for backpropagation. [13]

While there are several types of loss functions, this section focuses specifically on **Cross-Entropy Loss** and **Mean Squared Error (MSE) Loss**, as these were the primary loss functions used in this thesis.

Cross-Entropy Loss

Cross-entropy loss is commonly used in classification tasks, especially when the output of the neural network is a probability distribution. It quantifies the dissimilarity between the predicted probability distribution and the true distribution. Formally, for a target label y

and a predicted probability $\hat{y}$, the loss is computed as:

$$\mathcal{L}_{\text{CE}} = -\sum_{i}^{n} y_i \log(\hat{y}_i) \tag{2.2}$$

Cross-Entropy Loss penalizes confident but incorrect predictions more than less confident ones and encourages the model to output high probabilities for the correct class. [26]

Mean Squared Error (MSE) Loss

The Mean Squared Error (MSE) Loss is widely used in regression tasks. It measures the average squared difference between the predicted values and the actual values. Given predictions $\hat{y}$ and true values y, the MSE loss is defined as:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{2.3}$$

MSE loss is simple, convex, and provides smooth gradients, making it a popular choice for continuous output problems. However, it can be sensitive to outliers due to the squaring of errors. [26]

2.4.2 Backpropagation

Backpropagation is a method used to compute the gradients of the loss function with respect to the weights of a neural network. These gradients are then used by optimization algorithms, such as gradient descent, to update the weights in order to reduce the error between the predicted output and the true label. [4]

To calculate these gradients, backpropagation uses a mathematical rule called the *chain rule* from calculus. The chain rule helps us compute the derivative of a function that is made up of several nested functions. In a neural network, each layer applies a function to the output of the previous layer. The chain rule lets us „chain together“ the derivatives of each layer to find how changes in the weights affect the final output. [4]

For example, suppose the output of the network is a function $y = f(g(h(x)))$. The chain rule tells us that the derivative of y with respect to x is:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x} \tag{2.4}$$

This is exactly what backpropagation does, it moves backward through the network and applies the chain rule at each step to compute gradients efficiently. [4]

Once these gradients are computed, the network uses them to update the weights in the direction that reduces the loss. This is typically done using optimization methods like gradient descent [26].

2.4.3 Gradient Descent

Gradient descent is an optimization algorithm used to minimize the loss function of a neural network by updating its weights in the direction that reduces the error. [32] After computing the gradients using backpropagation, the weights are adjusted according to the formula:

$$w := w - \eta \cdot \frac{\partial L}{\partial w} \tag{2.5}$$

where w is the weight, η is the learning rate (a small positive number), and $\frac{\partial L}{\partial w}$ is the gradient of the loss function L with respect to the weight. [32]

There are several variants of gradient descent based on how much data is used to compute the gradients in each step. [32]

- **Batch Gradient Descent** – the weights in the neural network are updated only once per epoch, after computing the average gradient over the entire training dataset.
- **Stochastic Gradient Descent (SGD)** – the weights are updated after each individual training sample. This can introduce noise in the updates but may help the model escape local minima.
- **Mini-batch Gradient Descent** – a compromise between the two above, where the weights are updated after processing a small batch of training samples. This is the most commonly used variant in practice.

To improve the efficiency and stability of gradient descent, various optimization algorithms have been developed. These optimizers adjust the learning rate dynamically, incorporate momentum, or keep track of past gradients. [32] Some commonly used optimizers include:

- **Momentum** – accelerates gradient descent by taking into account the past direction of updates.
- **RMSprop** – adapts the learning rate for each parameter by dividing by a moving average of recent gradients.
- **Adam (Adaptive Moment Estimation)** – combines momentum and RMSprop, and is widely used due to its robustness and fast convergence.

For most modern neural networks, Adam is often the default choice due to its efficiency and adaptability across a wide range of tasks. [32]

2.4.4 Types Of Learning in Neural Networks

Neural networks can be trained using various learning paradigms, depending on the structure and availability of labeled data. The main types of learning are:

- **Supervised Learning** – In supervised learning, the neural network is trained on labeled data, meaning that each training sample is paired with a corresponding target or label. The goal is to learn a mapping from inputs to outputs, minimizing the error between predicted and true labels. This type of learning is commonly used for tasks such as image classification, speech recognition, or sentiment analysis [7].
- **Unsupervised Learning** – In unsupervised learning, the training data contains no labels—only input features. The neural network attempts to discover patterns, structures, or groupings in the data, such as clusters or low-dimensional representations. This approach is often used in anomaly detection, dimensionality reduction, or generative modeling. [11]

- **Reinforcement Learning** – In reinforcement learning, the neural network (agent) learns by interacting with an environment and receiving feedback in the form of rewards or penalties. The objective is to learn a policy that maximizes the cumulative reward over time. Reinforcement learning is commonly used in robotics, game playing, and decision-making problems. [22]

- **Semi-Supervised Learning** – Semi-supervised learning combines a small amount of labeled data with a large amount of unlabeled data during training. The model uses the labeled examples to guide learning and the unlabeled data to discover structure and improve generalization. This is useful when labeling data is expensive or time-consuming. [30]

- **Self-Supervised Learning** – Self-supervised learning is a special case of unsupervised learning where the system creates its own supervisory signals from the input data. It defines pretext tasks (e.g., predicting missing parts of an input) that help the model learn meaningful representations [24]. It can also serve as a form of regularization or pretraining to improve downstream supervised learning [38].

2.4.5 Activation functions

It is necessary to apply the activation function to the output of the neuron because if the activation function is not used, then the output signal would be a simple linear function, which is just a polynomial of degree one. A neural network without an activation function acts as a linear regression model, which has a limited ability to extract patterns in the data. [33]

Widely used and commonly known activation functions:

- **Rectified Linear Unit (ReLU)** – If the output is greater than 0, then the value is not changed (linear behavior); if the value is lower than 0, then the value is changed to 0. ReLU is defined as:

  $$f(x) = \max(0, x) \tag{2.6}$$

  Thanks to its simplicity and computational efficiency, ReLU has become one of the most widely used activation functions in deep learning models. [1]

- **Sigmoid** – Transforms the input into the range (0, 1). It is defined as:

  $$f(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

  Making it suitable for binary classification. [28]

- **Leaky ReLU** – A modification of ReLU that allows a small, non-zero gradient when the input is negative. is defined as:

  $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{2.8}$$

  where α is a small constant (e.g., 0.01). [28]

- **SoftMax** – Used in the output layer for multi-class classification problems. It converts raw scores (logits) into probabilities, ensuring that their sum equals 1. For an input vector $\mathbf{z}$ with components z_i, the SoftMax function is defined as:

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{2.9}$$

  where K is the total number of classes, the denominator sums the exponentials of all class scores z_j. This formulation ensures that each output value lies in the range (0, 1) and the entire output vector forms a valid probability distribution over the K classes. [4]

2.5 Training, Validation, and Test Datasets

When training and evaluating neural networks, the original dataset is typically split into three distinct subsets, each serving a specific purpose. [5]

1. **Training Dataset** – used directly during model training. The neural network learns patterns from this data and updates its internal parameters (weights and biases) to minimize the loss function.

2. **Validation Dataset** – used for continuous validation of the model during its training. For example, it can be used to detect overfitting and thus terminate training earlier.

3. **Testing Dataset** – used only after training is complete, providing an unbiased evaluation of how well the model generalizes to entirely new data. To ensure a fair assessment, the test set must remain isolated from both the training and validation phases.

2.6 Challenges with Training

Training neural networks is not always easy. There are several problems that can make learning slow or unstable. These issues often appear when the network is deep or when the training setup is not ideal. Common challenges include vanishing and exploding gradients, slow convergence, and getting stuck in bad local minima. It is important to understand these problems in order to train models more effectively.

2.6.1 Vanishing Gradient

When training deep neural networks, the problem of *vanishing gradients* can sometimes occur. This happens when the values of the gradients become very small as they are passed backward through the network during training. As a result, the early layers of the network (closer to the input) receive only a very weak signal for learning, and their weights are updated very slowly. This makes it hard for the network to learn useful features in those layers. [37]

The problem is caused by the way backpropagation works. During training, gradients are multiplied layer by layer from the output back to the input. If these values are small, the final result can become close to zero. This slows down or even stops learning in the first layers. [37]

2.6.2 Exploding gradient

The *exploding gradient* problem is the opposite of the vanishing gradient issue. It occurs when gradients become very large during backpropagation. This typically happens when gradients greater than one are repeatedly multiplied as they are propagated backward through the layers, leading to a rapid increase in their values. [9]

As a result, the weights in the network can be updated by excessively large amounts, causing the training process to become unstable. The model may start to oscillate or diverge, preventing it from converging to a good solution. [9]

2.6.3 Slow Convergence

Another issue that may occur during training is *slow convergence*, which refers to a slow decrease in the training error over time. This phenomenon prolongs the time needed to reach the desired performance and can result in unnecessarily high computational costs. Slow convergence may be caused by inappropriate weight initialization, poorly chosen learning rate, or suboptimal model architecture. [2]

2.6.4 Local Minima and Optimization Landscape

Neural networks are trained by minimizing a loss function using optimization algorithms such as gradient descent. The loss surface is usually non-convex, meaning it may contain many local minima, saddle points, and flat regions. A local minimum is a point where the loss is lower than its nearby values, but not necessarily the global minimum.

Earlier research considered local minima a major obstacle because the model could get stuck and stop improving. However, later studies showed that in high-dimensional settings, many local minima perform similarly in terms of generalization. [17] In fact, flat local minima are often associated with better generalization performance. More problematic are saddle points, where gradients are close to zero in some directions, potentially causing the optimization process to become inefficient and preventing the model from making meaningful progress. [8]

2.6.5 Overfitting and Underfitting

During the training of neural networks, two common problems can occur: overfitting and underfitting. These issues significantly affect the model's ability to generalize to unseen data and thus impact its overall performance and usability.

Overfitting happens when a model learns the training data too well, including its noise, outliers, or random fluctuations, instead of learning general patterns. It typically results in high accuracy on the training set but poor accuracy on the validation or test set. A large gap between training and validation loss is also a common sign of overfitting. Various *regularization techniques* can be used to mitigate this issue (see Section 2.7).

Underfitting, on the other hand, occurs when the model is not able to learn the training data well enough, not even the basic patterns and relationships. Common signs include low accuracy on both the training and validation sets, a small gap between training and validation error, and a training loss that stagnates even after many epochs.

Underfitting can be caused by factors such as an overly simple model or too few training epochs. It can often be mitigated by using a more complex architecture, training for a longer time, or adjusting the learning rate and other hyperparameters. [18]

2.7 Regularization Techniques

Regularization techniques are strategies used to improve the generalization ability of machine learning models. They act as constraints or modifications during the learning process to reduce model complexity or introduce useful biases that guide learning. Regularization is essential for building robust models that perform well on unseen data. [13]

2.7.1 Dropout

Dropout is one of the regularization techniques that is used to solve the overfitting problem. The main idea is to randomly drop the results of neurons in the hidden layers of the neural network. [35]

This technique is used in the training phase. Each neuron in the network is switched off with a certain probability each time a single batch is passed. This means that each time a single batch is passed, a smaller subnetwork of the entire trained neural network is trained. This makes the neural network more resilient to overfitting. During testing, dropout is no longer used, and all neurons are always active. [35]

2.7.2 L1 Regularization

L1 regularization adds a penalty to the loss function based on the absolute values of the weights. This means that the penalty for a weight w_i is proportional to its absolute value, i.e.,

$$\lambda \sum |w_i|, \tag{2.10}$$

where w_i is the weight of the model, and λ is the regularization strength. This penalty is added to the original loss function during training, effectively increasing the total loss that the model tries to minimize. As a result, the model will try to reduce not only the error between the predicted and actual values but also the sum of the absolute values of the weights. The effect of this penalty is that some weights are pushed to zero. This means L1 regularization can „select“ only the most important features (because some weights become exactly zero), simplifying the model and potentially improving interpretability. It's like a way of saying „keep only the important weights, and ignore the rest“. This technique is often used when you want to perform feature selection, i.e., identify which input features are most important. [13]

2.7.3 L2 Regularization

L2 regularization, on the other hand, adds a penalty to the loss function based on the squared values of the weights, i.e.,

$$\lambda \sum w_i^2, \tag{2.11}$$

the penalty is again added to the original loss function, so the model tries to minimize both the prediction error and the sum of the squared weights. The effect of L2 regularization is that it shrinks the weights towards zero, but unlike L1, it doesn't set them to exactly zero. It simply makes the weights smaller and prevents any single weight from becoming too large. This encourages the model to distribute the „importance“ across all features more evenly, leading to a more stable and generalizable model. L2 regularization is commonly used when you want to ensure all features are considered but prevent any weight from dominating the learning process. [13]

2.7.4 Early Stopping

This technique involves monitoring the model's performance on a validation set during training and stopping the training process as soon as the performance stops improving. This helps prevent the model from overfitting by stopping training before it begins to memorize the training data. [29]

2.7.5 Data Augmentation

Data augmentation generates new training samples by applying transformations such as rotations, translations, and flips to the original data. This increases the diversity of the training set and helps the model generalize better, reducing the likelihood of overfitting. [34]

2.7.6 Model Size

Reducing the complexity of a model, such as decreasing the number of layers or units in a neural network, can help prevent overfitting. A smaller model has fewer parameters and is less likely to memorize the training data, leading to better generalization. [13]

2.7.7 Self-Supervised Learning as Implicit Regularization

Self-supervised learning (SSL) is a technique where the model learns useful feature representations without requiring manual labels. It generates pseudo-labels from the data itself using pretext tasks, such as predicting missing parts of the input, solving jigsaw puzzles, or contrasting similar and dissimilar samples. Although SSL is primarily a representation learning method, it also acts as a form of implicit regularization. [19]

By forcing the model to solve auxiliary tasks before or alongside supervised learning, SSL guides the model to focus on generalizable patterns in the data rather than memorizing specific labels. This additional training signal introduces inductive biases and often leads to better generalization performances. [19]

SSL can also be combined with traditional supervised learning as a form of multi-task learning or pretraining. In such cases, the representations learned in the self-supervised phase regularize the downstream supervised task by initializing the model with weights that already capture meaningful structure in the input space. [19]

2.8 Weight Initialization in Neural Networks

Weight initialization refers to the strategy used to set the initial values of weights before training a neural network. Choosing an appropriate initialization method is critical, as it influences the speed of convergence, stability of training, and final model performance. Poor initialization can cause slow learning, exploding or vanishing gradients, and poor generalization. [27]

One key requirement is to avoid initializing all weights to the same value, which would cause all neurons in a layer to learn the same features, rendering the network ineffective. Instead, weights are typically initialized using random distributions that account for the number of input and output connections in each layer. [27]

This section presents several commonly used initialization methods and explains how they help maintain the flow of gradients during training.

2.8.1 Random Initialization

Random initialization assigns weights using a normal or uniform distribution, typically centered around zero. A simple choice is to sample from a standard normal distribution:

$$W \sim \mathcal{N}(0, 1) \tag{2.12}$$

Although easy to implement, this method can cause problems in deep networks, such as exploding or vanishing activations and gradients.

2.8.2 Xavier (Glorot) Initialization

Proposed by Glorot and Bengio in 2010, Xavier initialization aims to maintain a consistent variance of activations and gradients throughout the network layers. This helps prevent the vanishing or exploding gradient problems, particularly in deep networks. It is especially suited for activation functions that are symmetric around zero, such as sigmoid or tanh. [12]

The weights are initialized using a normal distribution with zero mean and variance depending on the number of input and output units:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right), \tag{2.13}$$

where n_{in} and n_{out} represent the number of input and output units in the layer.

The core idea of this initialization is to keep the scale of the weights balanced with respect to the layer size, ensuring stable forward and backward propagation. Bias parameters are typically initialized to zero, as they do not contribute to signal variance and this simplification does not negatively affect the learning process. [12]

2.8.3 He Initialization

He initialization, proposed by He et al. in 2015 , is specifically designed for neural networks using ReLU or ReLU-like activation functions (e.g., Leaky ReLU, PReLU). Unlike Xavier initialization, which assumes symmetric activations like tanh or sigmoid, He initialization accounts for the fact that ReLU-type functions output zero for half of the inputs, and therefore compensates by scaling the weights differently. [16]

Weights are initialized from a normal distribution with zero mean and variance:

$$W \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right), \tag{2.14}$$

where n_{in} is the number of input units in the layer.

This approach helps maintain stable variances of activations and gradients across layers, which is crucial for deep networks to avoid vanishing or exploding gradients. As with Xavier initialization, the bias terms are typically initialized to zero, as they do not affect variance propagation and zero initialization is generally sufficient for effective learning. [16]

2.8.4 LeCun Initialization

LeCun initialization, introduced in the context of efficient backpropagation, is particularly well-suited for activation functions such as sigmoid or SELU. It aims to preserve the variance of activations throughout the network without the need for explicit normalization layers. [21]

In this method, weights are initialized from a normal distribution with zero mean and a variance that depends on the number of input units:

$$W \sim \mathcal{N}\left(0, \frac{1}{n_{\text{in}}}\right), \tag{2.15}$$

where n_{in} denotes the number of input neurons in the layer.

LeCun initialization helps keep activation values within a range that avoids saturation, particularly when used with self-normalizing activation functions like SELU [20]. As with other initialization strategies, biases are commonly initialized to zero, which is sufficient for enabling effective learning and does not interfere with variance control.

2.8.5 Custom and Adaptive Initialization

Adaptive methods use data-driven strategies, such as pretraining or statistics of the training set, to initialize weights. These approaches can improve convergence and are sometimes combined with random methods to balance generalization and training efficiency [27].

2.9 Model Distillation and the Proposed Reversed Version

Model distillation is a technique used for compressing neural networks. Its core idea is to transfer knowledge from a large, high-performing neural network model (referred to as the *teacher*) to a smaller and simpler model (referred to as the *student*), which can then achieve comparable performance with significantly lower computational cost. Such compressed models are particularly useful in environments with limited computational resources, such as mobile devices. [15]

In general, there are three main approaches to performing model distillation:

1. **Response-based** – The student is trained to match the teacher's output probabilities as closely as possible.

2. **Feature-based** – The student learns to align its internal representations (e.g., hidden layer outputs) with those of the teacher.

3. **Relation-based** – The student is trained to capture the structural relationships between individual data samples as represented by the teacher.

Proposed Reversed Distillation Approach

This thesis proposes a reversed variant of model distillation, in which the traditional roles of teacher and student are swapped. In this approach, the larger model takes the role of the student, while the smaller model acts as the teacher. The goal is to explore whether a small model can still provide useful knowledge to a larger one, and whether this knowledge transfer can accelerate training or improve the generalization and final accuracy of the resulting model.

The motivation for this reversed setup is the idea that a small model may be capable of efficiently capturing the fundamental structures in the data, which a larger model can then refine and build upon. This could be especially beneficial in scenarios with limited computational resources or when aiming to warm-start the training of large models in a more informed way.

More generally, this method is investigated as an alternative approach to weight initialization, offering a potential substitute for random or heuristic-based initialization strategies. Instead of starting from random weights, the student model is guided in its training phases by a pretrained smaller model, possibly leading to a more stable and efficient learning process from the start.

In the experiments, the technique is evaluated across different types of data, including image and tabular datasets, as well as across multiple teacher–student architectural pairings. The focus is on comparing the performance of large models trained from scratch with those trained using the proposed reversed distillation technique.

2.10 Adversarial Examples

Adversarial examples are specially crafted inputs that are intentionally perturbed to fool machine learning models after their training, while remaining almost indistinguishable from the original inputs to a human observer. Typically, these perturbations are small in magnitude but cause the model to make incorrect predictions with high confidence. [36]

The phenomenon of adversarial examples was first described by Szegedy et al., who showed that even state-of-the-art neural networks could be manipulated with minimal input changes. These findings revealed that many models rely on fragile patterns in the training data rather than learning robust, human-like abstractions. [36]

2.10.1 Fast Gradient Sign Method (FGSM)

The *Fast Gradient Sign Method (FGSM)* is one of the most widely used techniques for generating adversarial examples. It was introduced by Goodfellow et al. and is based on linearizing the loss function around the current input. [14]

FGSM creates an adversarial example $\tilde{x}$ by perturbing the input x in the direction of the gradient of the loss function $\mathcal{L}$ with respect to the input:

$$\tilde{x} = x + \epsilon \cdot \text{sign}\left(\nabla_x \mathcal{L}(x, y)\right) \tag{2.16}$$

where:

- x is the original input,
- y is the true label,
- ϵ is a small constant that controls the strength of the perturbation,
- $\nabla_x \mathcal{L}(x, y)$ is the gradient of the loss with respect to the input,
- sign() is a function that returns the sign of each component in the gradient vector. This way, the perturbation is applied in the direction that most increases the loss. [14]

FGSM is computationally efficient because it requires only a single gradient computation, making it suitable for large-scale adversarial evaluations. [14]

Chapter 3

Reversed Model Distillation Experiments

This chapter aims to describe the design of experiments focused on testing the technique of reversed model distillation – that is, whether knowledge transfer from a smaller teacher model can help a larger student model learn more effectively. While traditional distillation typically relies on a larger and more accurate teacher model, this approach takes the opposite direction – reversed. The main question is whether the distillation phase can serve as an effective mechanism for initializing weights by leveraging the teacher's outputs during the early stages of student training.

The goal of this section is to design experiments whose results will allow for evaluating the benefits of the proposed approach and comparing them with models trained without using distillation.

The experiments were performed on three datasets: – CIFAR-10, Fashion-MNIST, and California Housing – covering both classification tasks and a regression task. For each dataset, a set of models of varying sizes (Small, Medium, Large) was defined and used as teacher models, while the largest model (size Student) was used as the student. Common evaluation metrics were chosen to compare performance – accuracy for classification tasks and mean squared error for the regression task.

3.1 Experiments Design and Objectives

Since all experiments require a trained teacher model, the first step involves independently training this model using only its own outputs and a selected loss function.

In the main phase of the experiments, focused on the technique of *reversed model distillation*, a larger student model is trained with guidance from a smaller teacher model. Rather than relying solely on its own predictions (known as the *hard loss*), the student also incorporates the outputs produced by the teacher (*soft loss*). The duration of this guidance is determined by the `switchEpoch` parameter, which specifies the number of epochs during which the student learns from the teacher. Once this epoch is reached, distillation is disabled and the student continues training only on the hard loss. Another important parameter is `alpha`, which determines the relative contribution of each loss component during the distillation phase. Specifically, `alpha` defines the proportion of the hard loss, while (1 − `alpha`) defines the proportion of the soft loss. For example, an `alpha` value of 0.6 combined with a `switchEpoch` of 10 means that that for the first 10 epochs, the student learns from a weighted combination of 60% hard loss and 40% soft loss. After epoch 10, the model is trained solely on the hard loss, relying entirely on its own predictions. The overall setup is illustrated in Figure 3.1.

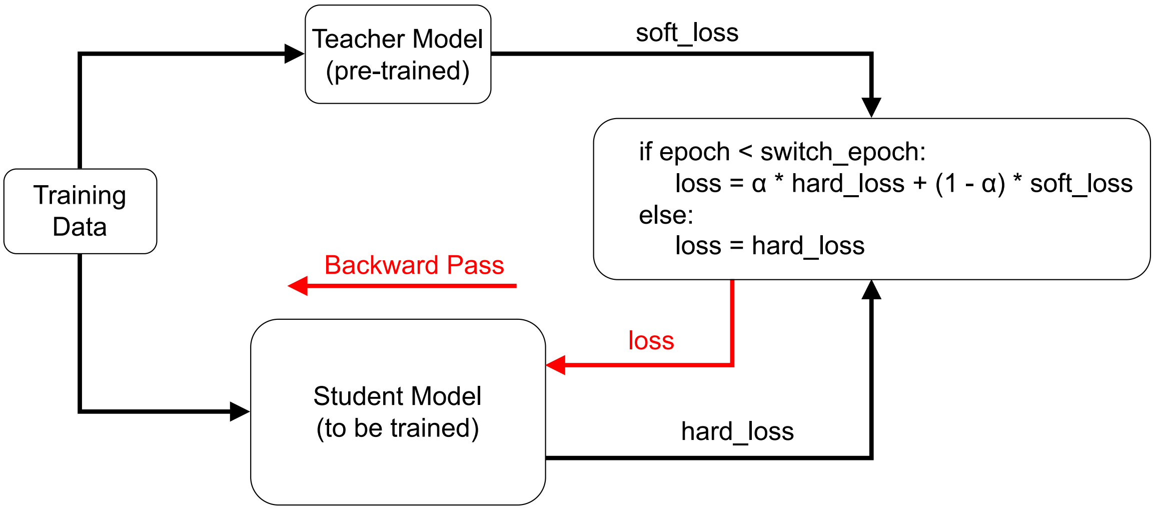


Figure 3.1: Design of Reversed Model Distillation Approach

The core objective of these experiments is to examine whether knowledge transferred from a smaller teacher can help the larger student model to learn smoother and better-generalizing representations. By imitating the teacher's soft outputs, which typically carry more nuanced information than hard labels, the student can be guided towards creating more robust decision boundaries and avoiding overfitting. The `switchEpoch` parameter thus directly affects how long the student benefits from the teacher's guidance.

In this thesis, the distillation phase is viewed as an alternative to standard weight initialization. It represents a way to provide the model with an initial sense of direction and guide them towards more effective learning. This strategy has the potential to influence not only the speed of convergence, but also the final generalization and overall training stability.

3.1.1 Experimental Design

As previously mentioned, this method was applied to three different datasets. For each dataset, initialization strategies were designed with respect to its complexity. In addition, three different teacher model sizes were selected (Small, Medium, Large).

For example, in the case of the simpler Fashion-MNIST dataset, it was expected that a smaller value of the parameter α and a shorter distillation phase would lead to better results. Conversely, for more complex datasets such as CIFAR-10, it was assumed that a larger teacher model and a longer distillation period would be more beneficial. For the regression task California Housing, an even longer distillation period was expected due to the more complex nature of the task.

A summary of these expected configurations for each dataset is presented in Table 3.1.

Table 3.1: Expected optimal distillation parameters for different datasets

Dataset	**Optimal α**	**Optimal switch epoch**	**Teacher model size**
Fashion-MNIST (simple classification)	0.1 or 0.6	Low (1–5) first 30% of training	Small / Medium
CIFAR-10 (more complex classification)	0.6 or 0.9	Medium (5–10) first 30-60% of training	Medium / Large
California Housing (regression)	0.6 or 0.9	High (800–1600) 60-100% of training	Medium / Large

It is assumed that more complex datasets require a longer distillation period and a higher value of the α parameter in order for the student model to effectively benefit from the teacher model's knowledge.

To objectively evaluate the impact of individual factors, all combinations of parameters (the value of α, teacher model size, and switch epoch) were tested across all three datasets.

3.1.2 Experimental Objectives

The goal of the experiments is to answer the following research questions:

- What is the impact of transferring knowledge from a smaller, simpler teacher model on the training of a larger student model?
- How does switching from distillation to standard training (controlled by the `switchEpoch` parameter) affect performance and convergence speed?
- Does the combination of soft and hard targets improve final accuracy and training stability compared to training without a teacher?
- How does the method perform across different task types – classification vs. regression?
- What is the impact of the teacher model's complexity (size)?
- How does varying the `alpha` parameter, which controls the balance between soft and hard loss, influence the student model's performance?

3.2 Datasets and Experimental Setup

This section describes used datasets and fixed settings of all conducted experiments.

3.2.1 Used Datasets

CIFAR-10 – An image classification dataset consisting of 60,000 RGB images (32×32 pixels) across 10 classes. 50,000 images are used for training and 10,000 for testing. The dataset represents a moderately challenging classification task due to its relatively high visual variability.

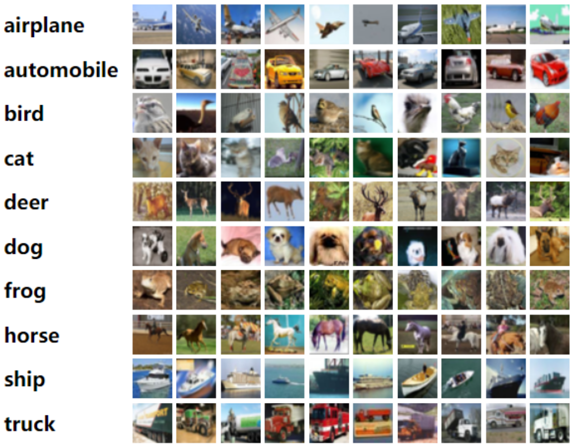
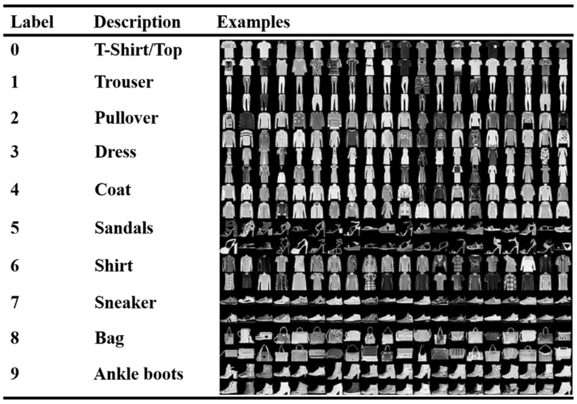


Figure 3.2: Cifar-10 Dataset [6]

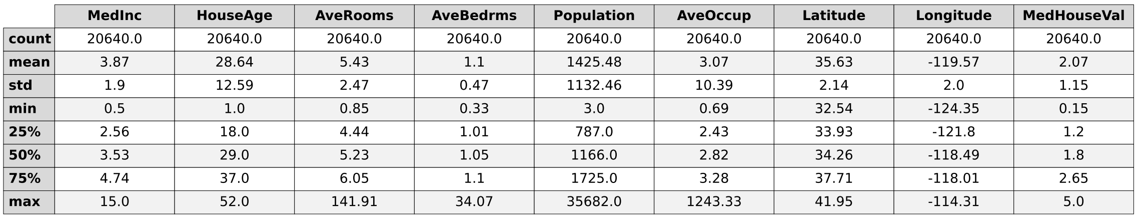
Fashion-MNIST – A simpler image classification dataset consisting of 80,000 grayscale images (28×28 pixels) of clothing items, categorized into 10 classes. 60,000 images are used for training and 10,000 for testing.



Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Figure 3.3: Fashion-MNIST Dataset [31]

California Housing – A real-world regression dataset containing socio-economic and geographic features used to predict the median house value in various California districts.



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
count	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0	20640.0
mean	3.87	28.64	5.43	1.1	1425.48	3.07	35.63	-119.57	2.07
std	1.9	12.59	2.47	0.47	1132.46	10.39	2.14	2.0	1.15
min	0.5	1.0	0.85	0.33	3.0	0.69	32.54	-124.35	0.15
25%	2.56	18.0	4.44	1.01	787.0	2.43	33.93	-121.8	1.2
50%	3.53	29.0	5.23	1.05	1166.0	2.82	34.26	-118.49	1.8
75%	4.74	37.0	6.05	1.1	1725.0	3.28	37.71	-118.01	2.65
max	15.0	52.0	141.91	34.07	35682.0	1243.33	41.95	-114.31	5.0

Figure 3.4: Summary Statistics of California Housing Dataset

3.2.2 Experimental Setup

In all experiments, models were trained with a fixed **batch size of 64**, which represents a commonly used compromise between computational efficiency and convergence speed. The **Adam** optimizer with a learning rate of **0.001** was used for weight optimization, as it is known for its stability and effectiveness even when training deeper neural networks.

During student model training, three values of the parameter **alpha** were explored: **0.1**, **0.6**, and **0.9**. This parameter determines the weighting of the final loss function – specifically, how much emphasis is placed on the *soft loss* (based on the teacher's outputs) versus the *hard loss* (based on the ground truth labels).

The parameter **switchEpoch**, as previously described, defines the number of epochs during which the student is influenced by the teacher. After this point, the distillation phase ends and the model continues training solely based on its own predictions, without any further guidance from the teacher. For classification tasks (**CIFAR-10** and **Fashion-MNIST**), training was conducted over **15 epochs**, and the tested values of `switchEpoch` ranged from **1 to 16**. The value of 16 indicates that distillation was used throughout the entire training process.

For the regression task (**California Housing**), which involved significantly longer training (**1500 epochs**), `switchEpoch` values were sampled from a broader range – specifically from **100 to 1600** in steps of 100. This broader scope enables a more detailed investigation into how the duration of the distillation phase affects final model performance.

To ensure fair comparison of different configurations (varying `alpha`, `switchEpoch`, and teacher model size), the same set of **random seed** values was used in all experiments. This ensured consistency in weight initialization, input batch ordering, and other stochastic processes across all experiment variants.

3.2.3 Hardware Environment:

All experiments were conducted on the server `sc-gpu2.fit.vutbr.cz`, which is part of the computing cluster of the Faculty of Information Technology at Brno University of Technology. The server is equipped with four powerful NVIDIA RTX A5000 graphics cards, each with 24 GB of VRAM. Typically, three of these GPUs were used for running the experiments, depending on resource availability. The server also features an Intel Xeon Silver 4314 processor with 32 cores, providing sufficient computational power for parallel processing. With 256 GB of RAM, the server can comfortably handle memory-intensive tasks. The system runs on the Linux distribution Ubuntu 22.04 with kernel version 6.8.0-57-generic. Thanks to fixed random seeds, all experiment results are fully reproducible on this server.

3.3 Neural Network Architectures

For each dataset, three teacher models and one student model were designed, differing in size and complexity. The goal was to create models representing various levels of computational demand and network capacity. The following descriptions provide an overview of the architectures to illustrate the extent of their differences.

3.3.1 CIFAR-10

This is an image classification task, so convolutional neural networks were used in all models. Each network contains a fully connected layer at the end with 10 neurons (10 classes). Additionally, dropout with a rate of 0.3 is used in all models.

- **Teacher Small** – It consists of one convolutional layer with 4 filters of size 3×3, followed by a max-pooling operation (2×2) and a fully connected layer with 16 neurons. The network is very simple and represents a low-capacity model.
- **Teacher Medium** – It also includes one convolutional layer, but with 16 filters, followed by max-pooling (2×2) and a fully connected layer with 64 neurons. Compared to the Small version, it offers significantly higher learning capacity.
- **Teacher Large** – It includes two convolutional layers (16 and 32 filters) followed by max-pooling (2×2) and a fully connected layer with 128 neurons.
- **Student** – The model is deeper than all teacher variants – it has three convolutional layers (64, 128, and 256 filters) with max-pooling (2×2) after the second and third convolutional layers) and a fully connected layer with 512 neurons.

3.3.2 Fashion-MNIST

Similar to CIFAR-10, this is an image classification task, so convolutional neural networks were also used here. However, they are adapted for single-channel input (only black-and-white images, not RGB as in CIFAR-10) and are significantly simpler since the complexity of this dataset is much lower. Again, all models have a fully connected layer at the end with 10 neurons (10 classes), and dropout with a rate of 0.3 is used.

- **Teacher Small** – Only one convolutional layer with 2 filters, followed by max-pooling (2×2) and a fully connected layer with 16 neurons.
- **Teacher Medium** – One convolutional layer with 4 filters, followed by max-pooling (2×2) and a fully connected layer with 32 neurons.
- **Teacher Large** – It consists of two convolutional layers (4 and 16 filters), followed by pooling and a fully connected layer with 64 neurons.
- **Student** – The largest model – 3 convolutional layers (32, 64, and 128 filters), with max-pooling (2×2) after the second and third convolutional layers, followed by a fully connected layer with 256 neurons.

3.3.3 California Housing

The California Housing dataset is a regression task with 8 input attributes and one output. The inputs are numerical, and the models here use fully connected layers exclusively.

- **Teacher Small** – A simple network with two linear layers ($8 \to 16 \to 1$). No dropout regularization.
- **Teacher Medium** – It includes three layers ($8 \to 16 \to 32 \to 1$) with dropout (0.3) between the layers.

- **Teacher Large** – A deeper network with four layers (8 → 32 → 64 → 32 → 1), also with dropout.
- **Student** – The deepest model, with five layers (8 → 64 → 128 → 64 → 32 → 1). It also uses dropout (0.3) between all hidden layers.

3.4 Training Procedure

This section describes the procedure used for training the models. It also covers the normalization of the input data.

3.4.1 Dataset Splitting

For the CIFAR-10 and Fashion-MNIST datasets, the standard 10,000 samples were reserved for testing. For the California Housing dataset, 20% was randomly selected as the test set, and the remaining 80% was used for training.

3.4.2 Preprocessing the Input Data from the Datasets

Each dataset was preprocessed in a specific way to make the training as efficient as possible. All preprocessing was performed separately for the training and test datasets to prevent information leakage from the test data into the training set, which could affect the results of the model.

- **CIFAR-10** – It contains RGB images, which were converted into tensors and normalized using the values: mean = (0.5, 0.5, 0.5), std = (0.5, 0.5, 0.5). As a result of this transformation, all pixel values in the entire dataset will be in the range <-1;1>.
- **Fashion-MNIST** – Grayscale images were converted into tensors and normalized using the values: mean = (0.5,), std = (0.5,). Again, due to this transformation, all pixel values in the entire dataset will be in the range <-1;1>.
- **California Housing** – The dataset contains only numerical values, and both the input attributes and the target variable, which the model will try to predict, were standardized. The standardization was performed using the StandardScaler from the sklearn library, which transforms each variable to have a mean of 0 and a standard deviation of 1.

3.4.3 Initial Weight Initialization

Before the start of training, all weights were initialized using the Kaiming He initialization method, which is well-suited for neural networks utilizing the ReLU activation function. This initialization technique was applied to both convolutional layers and fully connected layers.

3.4.4 Teacher models training

The teacher models were trained independently of the student models using standard supervised learning. In each iteration, the loss was computed as the difference between the model's predictions and the true targets using an appropriate loss function, CrossEntropyLoss was used for classification tasks (CIFAR-10 and Fashion-MNIST), and MeanSquaredErrorLoss was used for the regression task (California Housing). The training was carried out for a predefined number of epochs: 15 for classification tasks and 1500 for the regression task.

Depending on the nature of the task, two slightly different training procedures were used. The general structure of the training loops is described below.

For classification tasks, the teacher model was trained using minibatches, with the loss being accumulated and recorded every 100 minibatches to monitor convergence:

```
def train_model(train_loader, model, optimizer, criterion):
    epochs = config.EPOCHS
    model.train()
    training_losses = []
    for epoch in range(epochs):
        running_loss = 0.0
        for i, (inputs, targets) in enumerate(train_loader):
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if i % 100 == 99: # Save every 100 mini-batches
                training_losses.append(running_loss / 100)
                running_loss = 0.0
    return training_losses
```

Listing 3.1: Training loop for classification teacher models

In contrast, for the regression task, minibatches were not used. Instead, the entire training dataset was processed in each epoch as a single batch. The training loss was recorded once per epoch:

```
def train_model_reggresion(train_loader, model, optimizer, criterion):
    epochs = config.EPOCHS_REGRESSION
    X_train, y_train = train_loader
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32).to(device)
    y_train_tensor = torch.tensor(y_train, dtype=torch.float32).to(device)
    model.train()
    training_losses = []
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(X_train_tensor)
        loss = criterion(outputs, y_train_tensor)
        loss.backward()
        optimizer.step()
        training_losses.append(loss.item())
    return training_losses
```

Listing 3.2: Training loop for regression teacher models

This difference in training setup reflects the nature of the datasets: while classification tasks involve large amounts of small images requiring minibatch processing, the California Housing dataset is small enough to be processed entirely within a single batch during each epoch.

For each dataset and each teacher model size (Small, Medium, Large), 15 teacher models were trained using different random seeds. The model that achieved the highest test accuracy was selected and used in all subsequent distillation experiments for all configurations of a given dataset.

3.4.5 Student Models Training

For the training of student models, the reversed model distillation technique was applied. The training loss was not solely based on the difference between the student's outputs and the ground truth labels (hard loss), but also included a component measuring the difference between the student's outputs and the teacher's outputs (soft loss).

The type of soft loss depended on the nature of the task:

- For **classification tasks** (CIFAR-10 and Fashion-MNIST), soft loss was computed using the Kullback-Leibler (KL) divergence between the softened outputs (after applying `softmax`) of the teacher and the student.
- For the **regression task** (California Housing), soft loss was computed using the standard Mean Squared Error (MSE) between the raw outputs of the student and the teacher.

Training was carried out in two distinct phases:

- **Distillation phase:** From the beginning of training up to the epoch defined by `switchEpoch`, the total loss was computed as a weighted combination of hard and soft loss.

  The final loss function was defined as:

  $$\mathcal{L} = \alpha \cdot \mathcal{L}_{\text{soft}} + (1 - \alpha) \cdot \mathcal{L}_{\text{hard}} \tag{3.1}$$

  where α is a weighting parameter that controls the influence of the teacher during training. In the experiments, values of $\alpha \in \{0.1,\ 0.6,\ 0.9\}$ were evaluated.

- **Independent learning phase:** After reaching the specified `switchEpoch`, the student was trained exclusively using the `hard loss`, with no further guidance from the teacher. This phase was intended to assess whether the initial distillation phase serves as an effective weight initialization strategy.

The teacher model remained in the `eval()` state throughout the training process, and its weights were frozen (no backpropagation was performed) to ensure consistent outputs.

The following code snippets illustrate the training process for student models using reversed distillation for both classification and regression tasks:

For **classification tasks**, training was conducted in minibatches with training loss recorded every 100 minibatches, similarly to the teacher models:

```
def train_student_distill(train_loader, student_model, teacher_model,
              optimizer, criterion, switch_epoch, alpha):
```

```
    epochs = config.EPOCHS
    student_model.train()
    teacher_model.eval()
    training_losses = []

    for epoch in range(epochs):
        running_loss = 0.0
        for i, (inputs, targets) in enumerate(train_loader):
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()

            student_outputs = student_model(inputs)

            if epoch < switch_epoch:
                # Until switch_epoch use also teacher model
                # for computing loss
                with torch.no_grad():
                    teacher_outputs = teacher_model(inputs)

                loss_soft = F.kl_div(F.log_softmax(student_outputs, dim=1),
                F.softmax(teacher_outputs, dim=1), reduction="batchmean")
                loss_hard = criterion(student_outputs, targets)
                loss = alpha * loss_hard + (1 - alpha) * loss_soft
            else:
                loss = criterion(student_outputs, targets)

            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if i % 100 == 99: # Save every 100 mini-batches
                training_losses.append(running_loss / 100)
                running_loss = 0.0
    return training_losses
```

Listing 3.3: Training loop for student model with distillation (classification task)

For **regression tasks**, the entire dataset was processed once per epoch, with loss recorded once per epoch, also following the similar setup as for teacher model training:

```
def train_student_distill_regression(train_loader, student_model,
        teacher_model, optimizer, criterion, switch_epoch, alpha):
    epochs = config.EPOCHS_REGRESSION
    X_train, y_train = train_loader
    X_train_tensor = torch.tensor(X_train, dtype=torch.float32).to(device)
    y_train_tensor = torch.tensor(y_train, dtype=torch.float32).to(device)
    student_model.train()
    teacher_model.eval()
    training_losses = []
    for epoch in range(epochs):
        optimizer.zero_grad()
        student_outputs = student_model(X_train_tensor)

        if epoch < switch_epoch:
            with torch.no_grad():
                teacher_outputs = teacher_model(X_train_tensor)

            loss_soft = criterion(student_outputs, teacher_outputs)
            loss_hard = criterion(student_outputs, y_train_tensor)
            loss = alpha * loss_hard + (1 - alpha) * loss_soft
        else:
            loss = criterion(student_outputs, y_train_tensor)

        loss.backward()
        optimizer.step()
        training_losses.append(loss.item())
    return training_losses
```

Listing 3.4: Training loop for student model with distillation (regression task)

3.5 Evaluation Metrics

The results of the experiments were evaluated using several metrics that reflect both the performance of the model and the dynamics of its learning. The type of metric depended on the nature of the task (classification vs. regression), and all experiments were repeated with 15 different random seed values to ensure the robustness of the conclusions.

3.5.1 Classification Tasks

For the CIFAR-10 and Fashion-MNIST datasets, which represent multi-class classification tasks, the following metrics were used:

- **Accuracy** – The primary evaluation metric, calculated as the ratio of correctly classified samples to the total number of samples in the test dataset. The final accuracy was recorded after the completion of the training for each model variant.
- **Training Loss (Cross-Entropy Loss)** – The loss function used during the training of classification models. This metric allows monitoring of the convergence speed and enables the comparison of different configurations, such as the effect of the `switchEpoch` parameter. The running loss was computed by accumulating the loss over minibatches. Every 100 minibatches, the accumulated loss was averaged, saved, and the counter was reset. This approach gives a more detailed view of the training dynamics compared to recording only once per epoch, allowing the better detection of sudden changes in convergence speed or training instability.

3.5.2 Regression Task – California Housing

- **Testing Loss** – After the training of each model was completed, the model was evaluated on the entire test dataset. For each test example, the model's predictions were compared with the ground truth labels, and the total test loss was calculated as the Mean Squared Error (MSE) over all test samples

- **Training Loss (Mean Square Error)** – The loss function used during the training of regression models. Its progression was recorded after each training epoch, which allows for the analysis of the convergence speed and stability at different values of the `switch_epoch` parameter.

3.5.3 Other Metrics

- **Stability of Results Across Random Initializations (Seed)** – Each model was trained on the same 15 seeds. This allows for tracking the stability of each model across the given seeds.

- **Adversarial Robustness** – In addition to standard accuracy and loss-based metrics, adversarial robustness is evaluated to quantify each model's resistance to adversarial attacks. The evaluation is performed using the Fast Gradient Sign Method (FGSM) with varying levels of perturbation ϵ. For classification models, the drop in accuracy compared to clean data is reported; for regression models, the increase in loss (e.g., mean squared error) is measured. The robustness evaluation is applied uniformly across all models to enable comparison of the effects of different teacher models and values of α on adversarial robustness.

Chapter 4

Results and Analysis

This chapter presents an analysis of the experimental results aimed at evaluating the proposed *reversed model distillation* method. The goal is to compare the performance of models trained with this technique against those trained without distillation, across different datasets, model architectures, and hyperparameter settings. Special attention is given to the influence of key parameters such as `switchEpoch` and `alpha` on the training behavior and final performance of the student model, as well as its robustness to adversarial attacks (FGSM). The results are presented in the form of graphs with a focus on interpretation and comparative evaluation of the obtained metrics.

4.1 Training of Models without Distillation

Before running the experiments with proposed reversed model distillation method, it was necessary to train a set of teacher models, which would serve as sources of knowledge for the students. For each dataset (CIFAR-10, Fashion-MNIST, and California Housing), three teacher models of different sizes – *Small*, *Medium*, and *Large* – were defined. Each model was trained independently 15 times using different random seeds. For the reversed distillation experiments, the teacher model achieving the best test performance for each dataset was selected. The goal was to ensure that the student models learn from the highest-quality teacher available. Additionally, baseline student models for each dataset were trained without any distillation, providing a reference for later comparison with students trained using reversed model distillation.

4.1.1 Selection of the Best Teacher Models

For the classification tasks (CIFAR-10 and Fashion-MNIST), the primary selection criterion was the test accuracy achieved after 15 epochs of training. For the regression task (California Housing), the corresponding metric was the test Mean Squared Error (MSE). For each model type (*Small*, *Medium*, *Large*), the model with the best performance on the test set was selected as the final teacher model. The remaining models were retained only for the purpose of analyzing result variability.

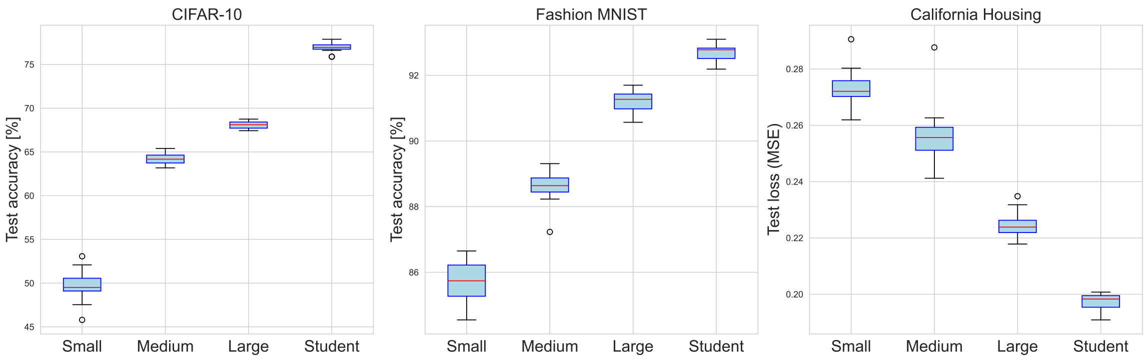


Figure 4.1: Performence variability across seeds by dataset and model size

As shown in Figure 4.1, the results of individual training runs varied depending on the selected random seed. Across all tasks (especially CIFAR-10), it can be observed that smaller models tend to be more unstable. As previously mentioned, for the distillation experiments, the best-performing teacher model from each group was selected (even if it was an outlier). Also, it is evident that the models differ sufficiently in performance from the smallest to the largest variant.

4.1.2 Training Progress

In addition to selecting the best-performing model, the convergence of the training process was also monitored. Figure 4.2 shows the training losses across all 15 initializations for each model size on the CIFAR-10 dataset. The graph also highlights the average training loss for each model size and, in the case of teacher models, additionally marks the training loss curve of the model selected based on the highest test accuracy.

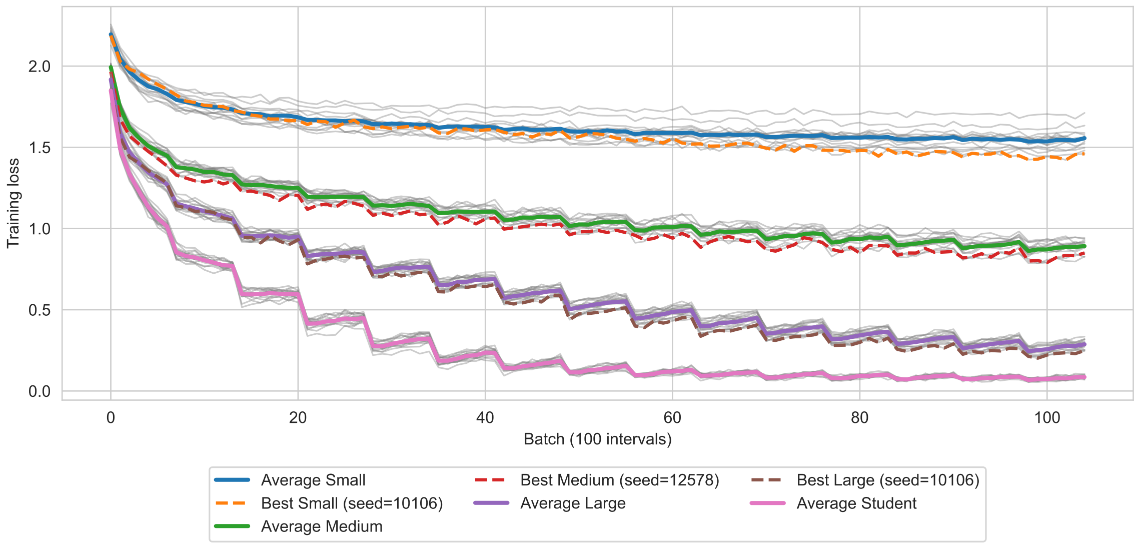


Figure 4.2: Training loss across CIFAR-10 models

The graph shows that the training curves were relatively stable. For the medium and large teacher models, as well as the student models, an interesting phenomenon can be observed in the form of „staircase“ in the training curves. Each such „step“ corresponds to a transition between epochs. After each epoch, there is a noticeable drop in the training loss value, followed by a gradual increase.

This pattern is related to how the model encounters training data during each epoch. At the beginning of a new epoch, the model experiences a sharp decrease in training loss because it initially processes data that it has already seen and adapted to during the previous epoch. As the epoch progresses, the model begins encountering new minibatches that it has not yet seen within the current epoch, causing a slight increase in training loss. This cycle repeats with each new epoch, resulting in the observed step-like shape of the training loss curves.

The training losses for the Fashion-MNIST dataset are similarly shown in Figure 4.3.

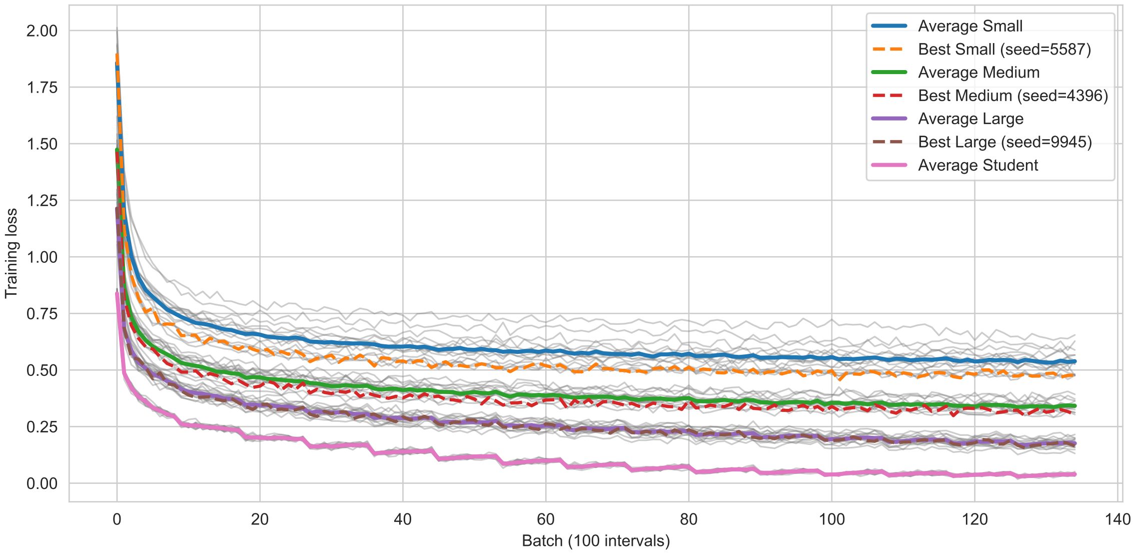


Figure 4.3: Training loss across Fashion MNIST models

The training curves again show that the training process was relatively stable, with the best-performing Small and Medium teacher models showing faster convergence or consistently lower training loss throughout the process. However, for the Large teacher models, the best model was not the one with the lowest training loss (i.e. the lowest training loss did not necessarily lead to the best test accuracy). This observation raises questions about the informativeness of the training loss function. In the case of the student model, the staircase effect in the training loss, previously observed with some CIFAR-10 models, appears again.

Graph 4.4 shows the training losses of the models on the California Housing dataset.

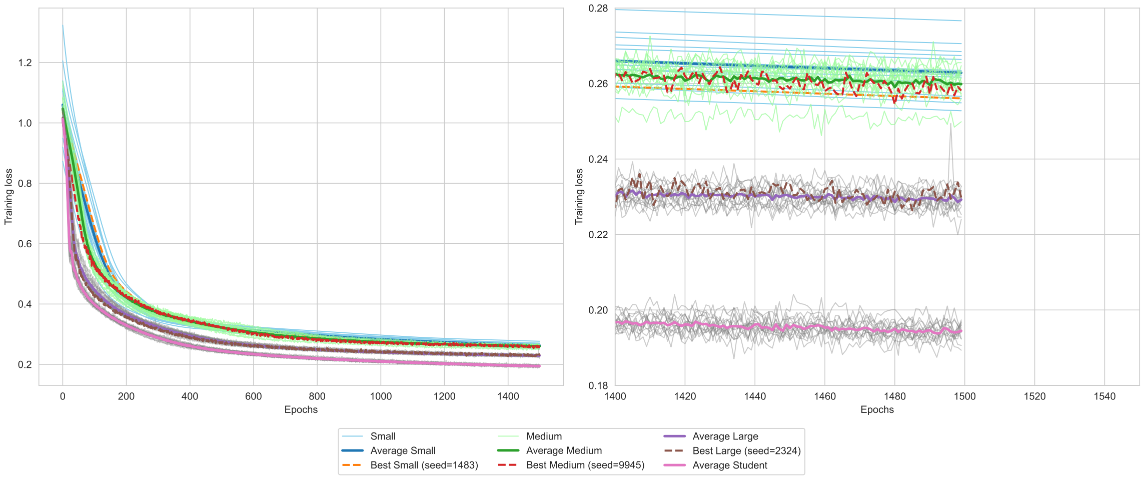


Figure 4.4: Training loss across California Housing models

The graph in Figure 4.4 shows that for the Medium and Large teacher models, the model achieving the best testing loss is not necessarily the one with the lowest training loss – similarly to what was observed with the Large teacher model on the Fashion-MNIST dataset 4.3.

Additionally, it can be observed that the training loss curves for the Small and Medium models largely overlap, despite their significantly different testing losses (as previously seen in Figure 4.1). This phenomenon can be explained by the fact that Dropout was not applied to the Small model, as its low capacity would make the use of Dropout too disruptive to the learning process and likely lead to worse performance on the test set. However, the absence of Dropout also means that the Small model has a training loss curve similar to that of the more complex Medium model. Nevertheless, the final testing loss of the Medium model is considerably lower, indicating that the Small model likely suffers from overfitting. Although overfitting is typically more problematic in larger models, in this case, the very limited capacity of the Small model without any regularization likely caused it to overfit to the training data.

Additionally, the lack of Dropout is reflected in the shape of the training loss curve itself, which is noticeably smoother for the Small model compared to the Medium and Large models.

4.1.3 Summary

The training of models demonstrated a certain degree of variability depending on the random seed used. It was also shown that the model sizes across all datasets were sufficiently distinct, with larger models generally achieving better results compared to smaller ones.

4.2 Training of Models with Distillation

This section presents the results of experiments aimed at evaluating the effectiveness of the *reversed model distillation* technique – that is, the transfer of knowledge from a smaller teacher model to a larger student model. The goal was to determine whether this technique could be used as an alternative strategy for weight initialization, potentially helping the student model to learn more efficiently, converge faster, and improve overall generalization. Across all experiments, three key parameters were systematically examined:

- The value of `alpha`, determining the balance between soft loss and hard loss during the distillation phase.
- The `switchEpoch` parameter, defining the epoch up to which distillation remains active.
- The size of the teacher model (*Small*, *Medium*, *Large*).

Experiments were conducted on three datasets: CIFAR-10 and Fashion-MNIST for classification tasks, and California Housing for a regression task. For each combination of parameters, 15 independent runs with different `seed` values were performed to evaluate not only the average performance but also the stability of the results. The following sections progressively analyze the impact of each parameter on model accuracy, convergence behavior, and training robustness.

4.2.1 Impact of the `switchEpoch` and `alpha` Parameters on the CIFAR-10 Dataset Accuracy

This section focuses on analyzing the influence of the reversed distillation phase length, represented by the `switchEpoch` parameter, on the final performance of the student model when trained on the CIFAR-10 dataset.

Detailed Analysis: Small Teacher, $\alpha = 0.9$

The first step of the analysis is a detailed examination of a specific configuration. This configuration represents a Small-sized teacher model that was used and the weighting parameter `alpha` that was set to 0.9.

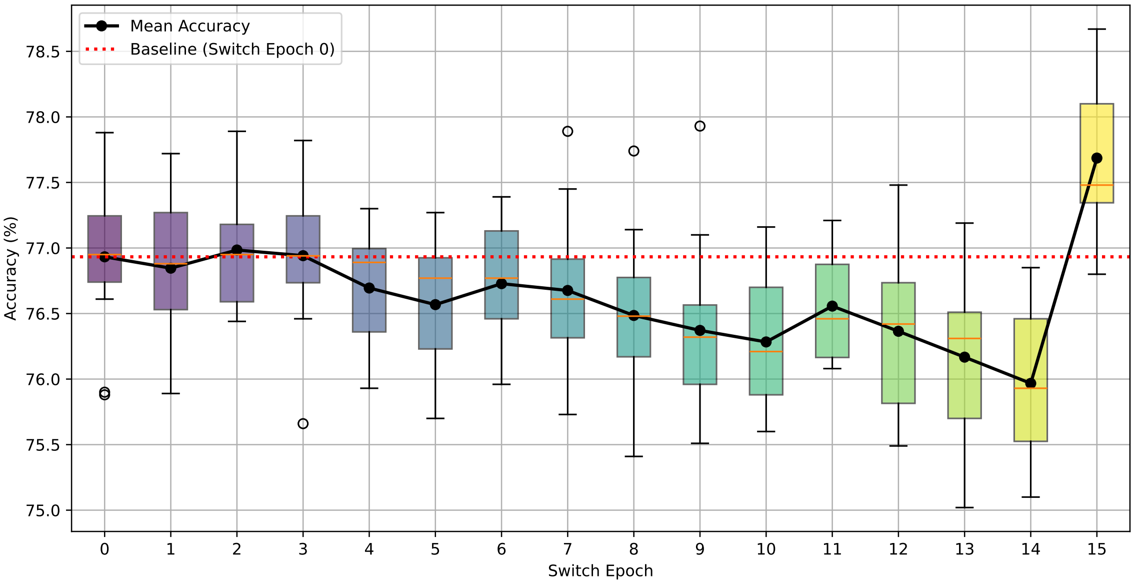


Figure 4.5: Accuracy boxplot of the CIFAR-10 student models trained with a Small teacher and $\alpha = 0.9$, depending on the `switchEpoch` value.

Figure 4.5 shows a boxplot of the student model accuracy across different values of `switchEpoch`. Each box represents the results of 15 training runs using the same set of random seeds for consistency. The point `switchEpoch` = 0 serves as a reference baseline, representing the student model trained without any distillation.

The results indicate that distillation improved accuracy only in the configuration with `switchEpoch` = 15, where the average accuracy exceeded the baseline. In most other cases, the use of distillation actually led to a slight degradation in performance. The lowest average accuracy was recorded between `switchEpoch` = 8 and 14, where the performance curve showed a noticeable decline.

This pattern suggests that the effectiveness of knowledge transfer through reversed distillation heavily depends on the duration of the distillation phase. Maintaining distillation throughout the entire training process can be beneficial, as seen with `switchEpoch` = 15. An inappropriate choice of distillation duration (for example, between the 8th and 14th epoch) can lead to a noticeable degradation in performance.

It is also noticeable that there is considerable variability in results across different random seeds.

Overview Across All Configurations

For a more comprehensive perspective, an overview visualization was created showing the average accuracy across all parameter combinations (teacher model size and `alpha` value). The resulting plot is presented in Figure 4.6. Each curve represents one of the nine configurations (3 different `alpha` values × 3 teacher model sizes). The X-axis corresponds to the various `switchEpoch` values, while the Y-axis shows the average student model accuracy computed across 15 training runs with different seed values.

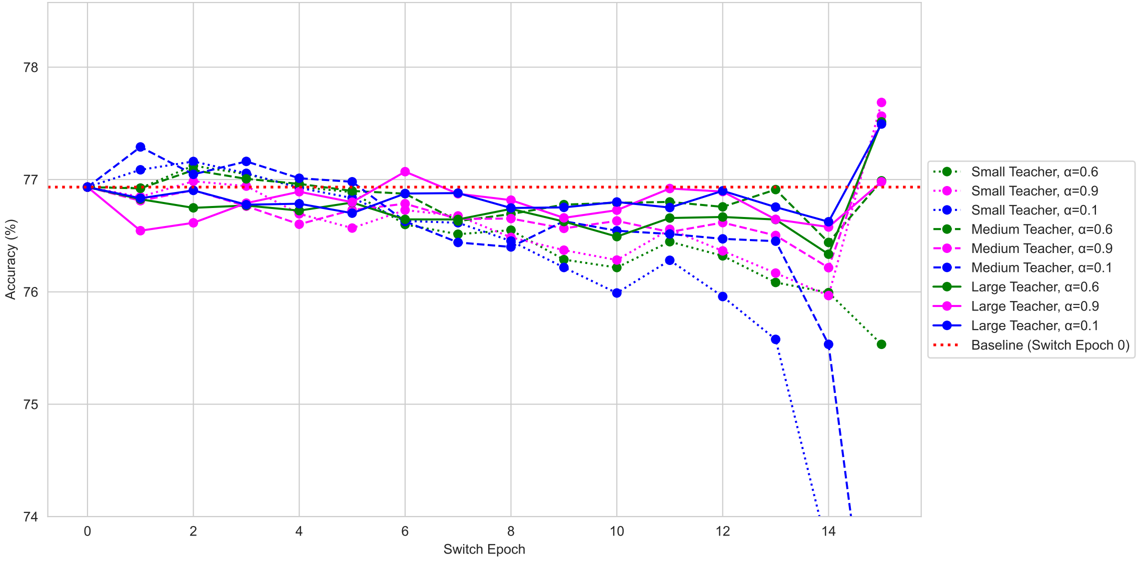


Figure 4.6: Summary of CIFAR-10 Student Models Training with Distillation

From this graph, it is evident that the effectiveness of distillation depends on multiple factors, such as the size of the teacher and the choice of the parameter α. Some configurations demonstrate either stable or slightly improved accuracy compared to the baseline, particularly for lower `switchEpoch` values. Several combinations, for example, a *Small* and *Medium* teacher model with $\alpha = 0.9$ or *Large* teacher model with $\alpha = 0.6$ and 0.1, achieve their best performance at `switchEpoch` = 15, again suggesting the potential benefits of longer distillation. In contrast, some configurations, especially those with $\alpha = 0.1$ trained with *Small* and *Medium* teacher, experienced a significant drop in accuracy when the switch from distillation to standard training was made at later epochs, likely due to excessive reliance on the teacher and insufficient adaptation to the student's own optimization objective.

Overall, the benefits of distillation are not strictly tied to a specific epoch but rather emerge from the interplay of several factors.

For better interpretation of the results, heatmaps were also created (Figure 4.7), showing the average accuracy achieved by the student across all combinations of `alpha` and `switchEpoch` parameters. Each heatmap corresponds to one teacher model size, – *Small*, *Medium*, and *Large* – and enables direct comparison of individual configurations.

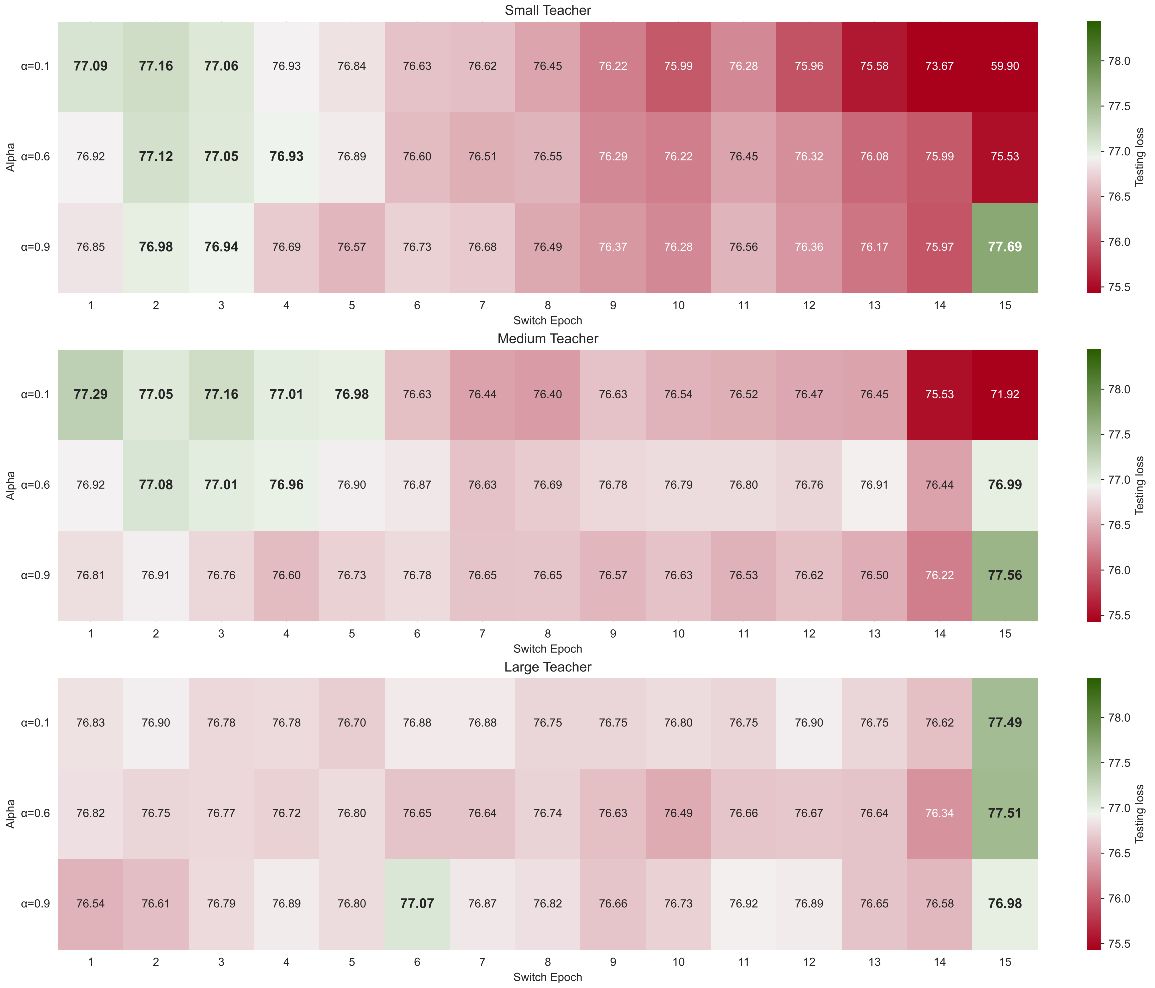


Figure 4.7: Average Accuracy Heatmap of CIFAR-10 Student Models Across Configurations

The X-axis represents the values of the `switchEpoch` parameter, ranging from 1 to 15, while the Y-axis shows the three selected `alpha` values (0.1, 0.6, 0.9). Each cell displays the average test accuracy computed over 15 independent runs (with different random seeds) for the corresponding configuration. To visually express the difference relative to the non-distilled baseline model, each value is color-coded as follows:

- Green indicates an improvement compared to the baseline (i.e., the average accuracy across all 15 runs of the student model trained without distillation).
- Red indicates a degradation compared to the baseline.
- Light or neutral colors (close to white) correspond to accuracy values similar to the baseline.

The average performance of the student model without distillation on the CIFAR-10 dataset is **76.93%**. Therefore, all results exceeding this value represent an improvement achieved through reversed distillation.

Several important trends can be observed from the heatmaps:

- Short distillation phases (`switchEpoch` = 1 to 4) generally led to slight improvements when using a Small or Medium teacher with `alpha` values of 0.1 or 0.6.
- Longer distillation phases (particularly at `switchEpoch` = 15) were successful for all teacher sizes when `alpha` was set to 0.9. For example, with the `Small Teacher` and $\alpha = 0.9$, the model achieved a significantly better accuracy than the baseline (an improvement of 0.76%).
- In contrast, intermediate `switchEpoch` values between 7 and 11 consistently led to lower accuracy across all combinations of teacher sizes and `alpha` values. This suggests that having distillation for too long, but not for the entire training, can make learning less effective.
- Some configurations showed significant drops in accuracy, particularly those with `alpha` = 0.1 trained with *Small* or *Medium* teachers at higher `switchEpoch` values. The worst recorded accuracy was 59.90%, observed for the Small teacher with $\alpha = 0.1$.

Overall, the heatmaps complement the previous line plot visualizations by enabling precise identification of the optimal distillation parameter combinations, while also highlighting configurations where knowledge transfer through reversed model distillation from a smaller teacher may have a negative impact.

4.2.2 Impact of `switchEpoch` and α on the Fashion-MNIST Dataset Final Accuracy

Detailed Analysis: Large Teacher, $\alpha = 0.6$

This configuration achieved very strong results across the Fashion-MNIST experiments, which is why I focused on a more detailed analysis of the influence of the `switchEpoch` parameter. Figure 4.8 shows the distribution of accuracy across 15 different random initializations for each value of `switchEpoch`.

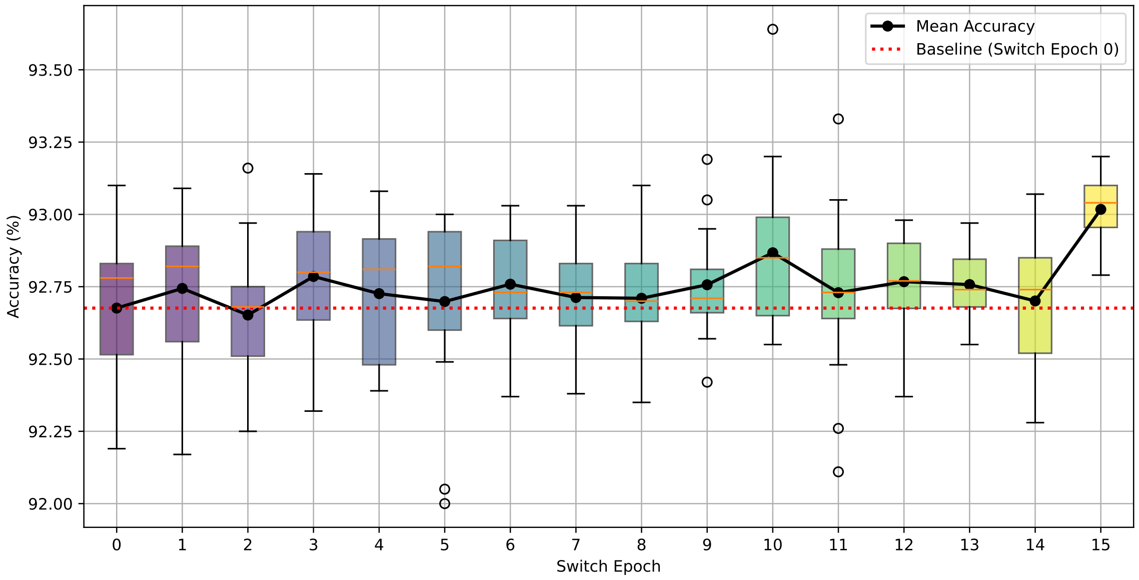


Figure 4.8: Accuracy boxplot of the Fashion-MNIST student models trained with a Large teacher and $\alpha = 0.6$, depending on the `switchEpoch` value.

From the graph, it is evident that apart from distillation with a `switchEpoch` value of 2, the final average accuracy consistently improved. The lowest mean accuracy was observed at `switchEpoch` $= 2$. It is also interesting that relatively good performance was achieved at `switchEpoch` $= 10$, which was not observed in the CIFAR-10 experiments. The highest and most consistent results were achieved at `switchEpoch` $= 15$, where the student model learned from the teacher throughout the entire training process.

However, the spread of the results is relatively high, indicating that the model's behavior is not very stable across different random initializations. This trend is very similar to what was observed on the CIFAR-10 dataset.

Overview Across All Configurations

Figure 4.9 shows the average achieved accuracy across different seeds for each tested value of α and each teacher model size, plotted against various `switchEpoch` values.

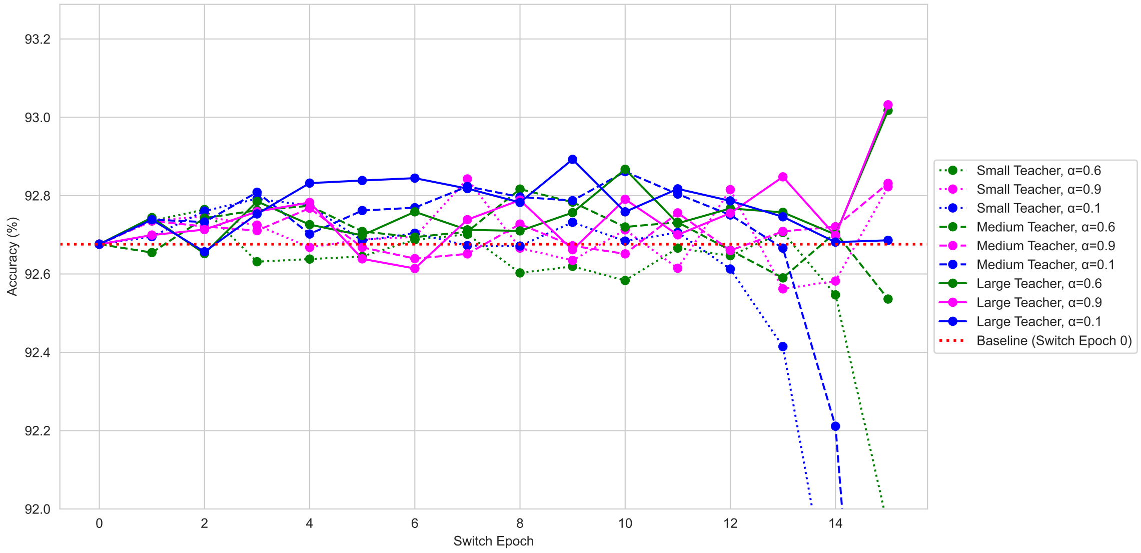


Figure 4.9: Summary of Fashion-MNIST Student Models Training with Distillation

The results, similarly to the CIFAR-10 dataset, indicate that using a Small or Medium teacher model in combination with a very low α (0.1) at higher `switchEpoch` values significantly reduces the final accuracy, making this configuration not a good choice for achieving better model performance. A similar trend can also be observed with the Small teacher model and $\alpha = 0.6$ at later `switchEpoch` values (14 to 15), consistent with the CIFAR-10 experiments. On the other hand, as with CIFAR-10, good results were achieved when distillation was applied throughout the entire training process with $\alpha = 0.9$.

Accuracy Heatmaps

A complete overview of the average results across seeds for all combinations of `switchEpoch` and α values is presented in the heatmap in Figure 4.10. The average accuracy of the student model trained on the Fashion-MNIST dataset without distillation across 15 seeds was **92.67%**. Values shaded in green indicate improved accuracy, while values shaded in red indicate decreased accuracy compared to the baseline.

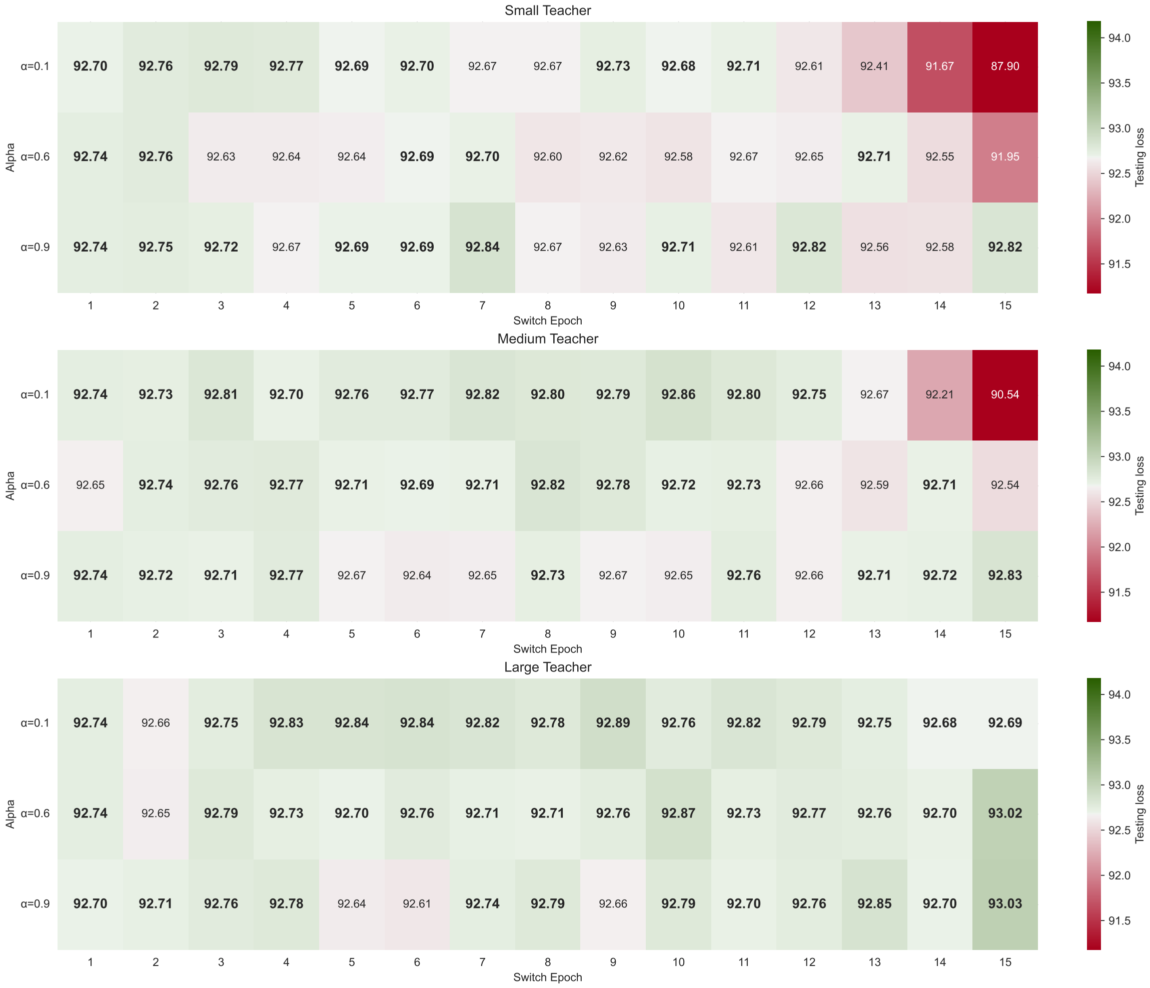


Figure 4.10: Average Accuracy Heatmap of Fashion-MNIST Student Models Across Configurations

Similar to the CIFAR-10 dataset, the highest average accuracies were achieved in the bottom-right corner of the matrices, corresponding to higher values of α and higher `switchEpoch` values, which was already observed in the previous analysis. However, compared to CIFAR-10, it can be seen that distillation generally benefited most configurations on Fashion-MNIST, although the improvements in accuracy were often only within a few hundredths of a percent.

On the other hand, unsuitable configurations again included those with a low α combined with distillation applied throughout the entire training, such as $\alpha = 0.1$ and `switchEpoch` $= 15$, which led to significantly lower accuracy. This further confirms that overly extensive distillation from a smaller teacher model combined with lonver distillation throughout training can be counterproductive to the student model's performance.

4.2.3 Impact of the `switchEpoch` and α Parameters on the Final Testing Loss of California Housing Dataset Models

This section focuses on analyzing the impact of the `switchEpoch` and α parameters on the performance of the student model in the regression task on the California Housing dataset.

The results are expressed using the testing loss metric (Mean Squared Error, MSE), where lower values indicate better performance.

Figure 4.11 presents an overview visualization, illustrating the evolution of the test loss across different `switchEpoch` values for all combinations of teacher model sizes and α values, similarly to the analyses for the CIFAR-10 and Fashion-MNIST datasets. The graph also includes a reference line (baseline) representing the performance of the student model trained without any distillation.

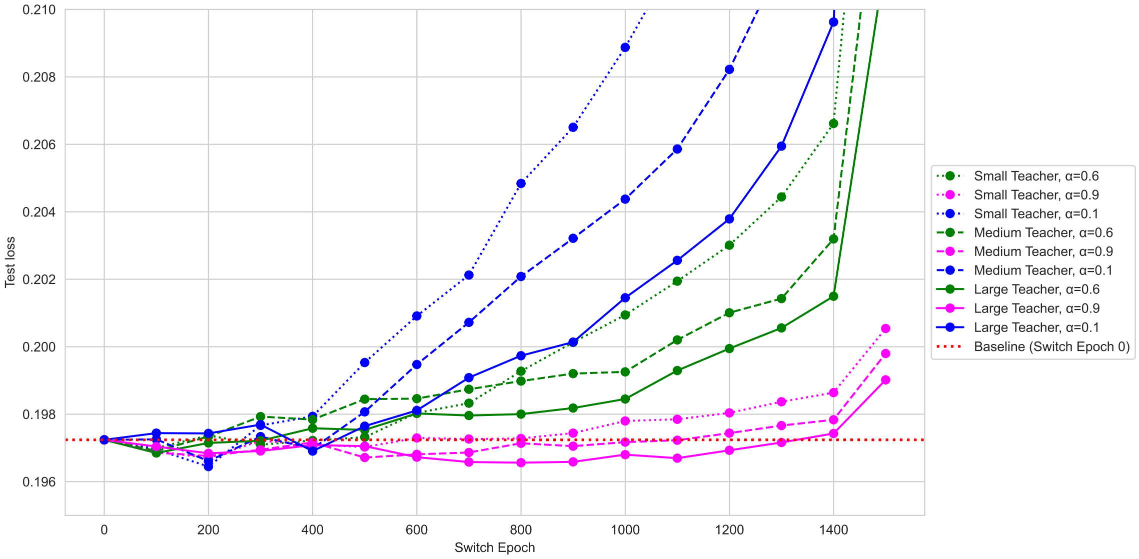


Figure 4.11: Summary of California Housing Student Models Training with Distillation

The graph reveals that the combination of a smaller teacher model and a higher α value generally led to worse results. This trend is particularly noticeable for small and medium teachers during longer distillation phases, where the testing loss increased significantly. In contrast, for some configurations involving a large teacher and lower α values, the performance remained closer to the baseline.

Overall, reversed distillation on the California Housing dataset had a predominantly negative impact, often resulting in worse performance compared to standard training without a teacher. One possible explanation lies in the nature of the regression task itself: distillation here was performed as a direct transfer of output values using MSE loss, unlike in classification tasks where KL divergence is used to capture more subtle differences between probability distributions. This distinction suggests that, in regression, the student may not benefit from as rich information through the teacher's outputs, thus limiting the effectiveness of the distillation phase.

For a more detailed overview, the results were also visualized using heatmaps (see Figure 4.12), which show the average testing loss (MSE) across all combinations of `switchEpoch` and `alpha` values. Each cell represents the mean result from 15 runs for the given configuration. The color scale is centered around the baseline model, which was trained without distillation (average loss of **0.1972**) – values better than the baseline are shaded green, while worse values are shaded red.

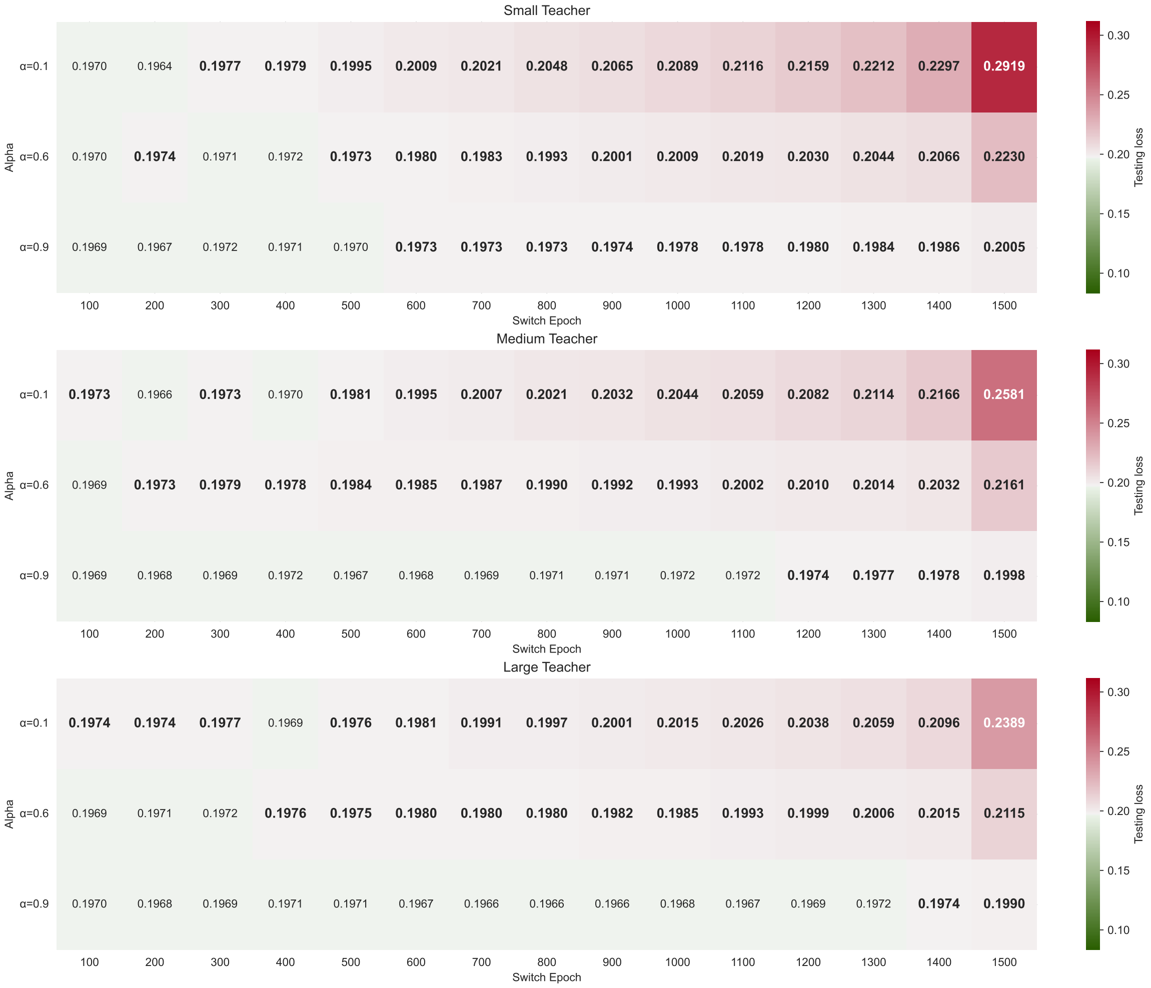


Figure 4.12: Average Accuracy Heatmap of California Housing Student Models Across Configurations

From the heatmaps, it is evident that only a very limited number of configurations resulted in a slight improvement over the baseline, typically when using a large teacher model combined with a low α value. Most other combinations, particularly those involving smaller teachers and higher α values, led to an increase in testing loss. This trend is especially pronounced for `switchEpoch` $= 1500$, where all configurations exhibited significantly worse results.

4.2.4 Impact of `switchEpoch` and α Parameters on Training Convergence for CIFAR-10

To gain a deeper understanding of how the individual distillation parameters affect the learning process of the student model, the development of training loss throughout the training phase is analyzed.

In all experiments, the training loss was recorded every 100 batches. For each value of the `switchEpoch` parameter (ranging from 1 to 15), 15 runs were performed using predefined random seeds, and the results were subsequently averaged. Each curve in the graph corresponds to the averaged loss curve for a given `switchEpoch` setting. For comparison, a reference curve is added to each graph in red, representing the average training loss progression of a model trained without any distillation.

Detailed View: Medium Teacher, $\alpha = 0.6$

As a representative case, the configuration with a medium-sized teacher and $\alpha = 0.6$ was selected for a more detailed analysis.

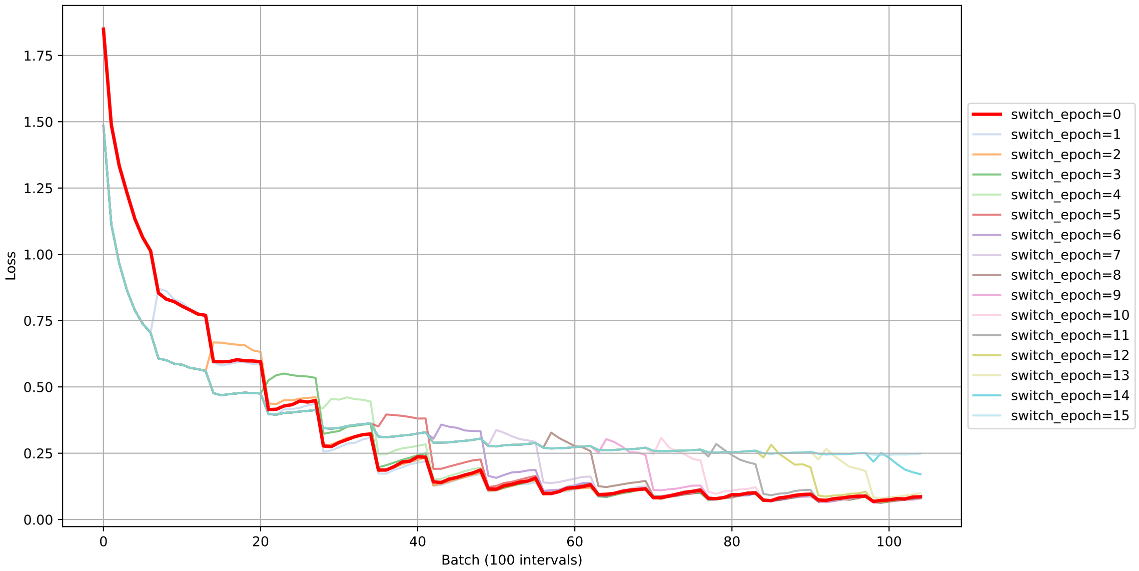


Figure 4.13: Average Running Training Loss Depending on the `switchEpoch` Parameter with a Medium Teacher and $\alpha = 0.6$

In the graph shown in Figure 4.13, typical staircase-like patterns in the training loss progression can be observed, similar to those described during the training of teacher models (models trained without distillation) 4.2. However, additional significant changes are also present, almost every curve (with the exception of the baseline and the configurations with `switchEpoch` = 14 and 15) exhibits a sharp drop in training loss, corresponding to the moment when the model transitions from learning with teacher guidance to relying solely on its own outputs. This transition leads to a steeper decrease in loss, as the student model, having greater capacity, is no longer „constrained“ by the smaller teacher.

The reference baseline student model does not display such a drop, as it is trained without any distillation from start to finish. Similarly, the curve for `switchEpoch` = 15 also lacks this drop, because distillation is active throughout the entire training process. In the case of `switchEpoch` = 14, only a slight hint of this decline appears near the end of training, but the remaining single epoch is insufficient to produce a more significant change.

It is also possible to notice a slight „spike“ at the point where distillation is turned off – the training loss temporarily increases before resuming its downward trend. This phenomenon occurs consistently across different configurations and is more clearly illustrated in Figure 4.14, where it is particularly noticeable.

Detailed View: Small Teacher, $\alpha = 0{,}1$

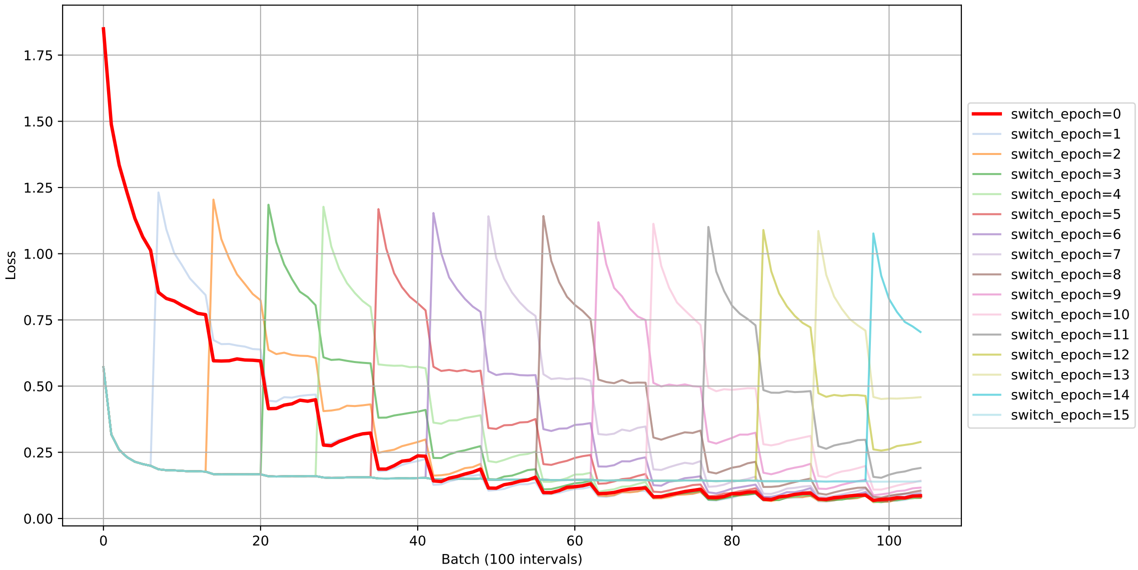


Figure 4.14: Average Running Training Loss Depending on the `switchEpoch` Parameter with a Small Teacher and $\alpha = 0.1$

In this graph, a sharp increase in training loss is clearly visible at the moment when the training switches from the distillation phase to conventional learning – that is, after reaching the specified `switchEpoch`. This phenomenon is caused by a change in the nature of the loss function: during distillation, the loss is predominantly (90%) influenced by the soft loss from the teacher, who in this case is significantly smaller than the student and produces more stable, less variable outputs. Once the distillation phase ends and the student begins learning solely based on its own predictions, the model's higher capacity leads to a larger error and, as a result, a higher training loss. The following decrease in loss reflects the model's adaptation to learning exclusively from its own outputs.

Summary Overview

In the appendices (see A.1), all average training loss curves across individual configurations are provided, evaluated as the mean over 15 independent runs for each configuration. It can be observed that the larger the model, the closer the training losses are to each other. Additionally, the smaller the value of α, the more noticeable the jumps in training loss at the point of switching from distillation to regular training.

4.2.5 Impact of `switchEpoch` and α Parameters on Training Convergence on the Fashion-MNIST Dataset

For the Fashion-MNIST dataset, similar trends to those observed on CIFAR-10 can be seen. Lower values of the `alpha` parameter often lead to more noticeable jumps in training loss when switching from distillation to standard learning. However, unlike the more complex CIFAR-10 dataset, there is no significant difference between the various teacher model sizes. The training loss trajectories are very similar across all three variants. This phenomenon likely results from the lower complexity of the Fashion-MNIST dataset, which makes the model less sensitive to the size of the teacher during training. In the appendices (see A.2), all average training loss curves across the different configurations on this dataset are again provided, evaluated as the mean over 15 independent runs for each configuration.

4.2.6 Impact of `switchEpoch` and α Parameters on Training Convergence on the California Housing Dataset

To gain deeper insights into the training dynamics under reversed model distillation in a regression setting, the evolution of training loss throughout the training phase was analyzed.

Unlike in the CIFAR-10 and Fashion-MNIST experiments, where the training loss was recorded every 100 minibatches, in the case of the California Housing dataset, the loss was recorded only once per epoch. This difference is due to the nature of the dataset, the entire training set could be processed within a single epoch without the need for minibatching. As a result, the recorded curves are smoother, as they reflect the average loss over the entire epoch rather than detailed variations within individual minibatches. Nevertheless, characteristic changes in training behavior at the `switchEpoch` point are still clearly observable.

Detailed View: Medium Teacher, $\alpha = 0.6$

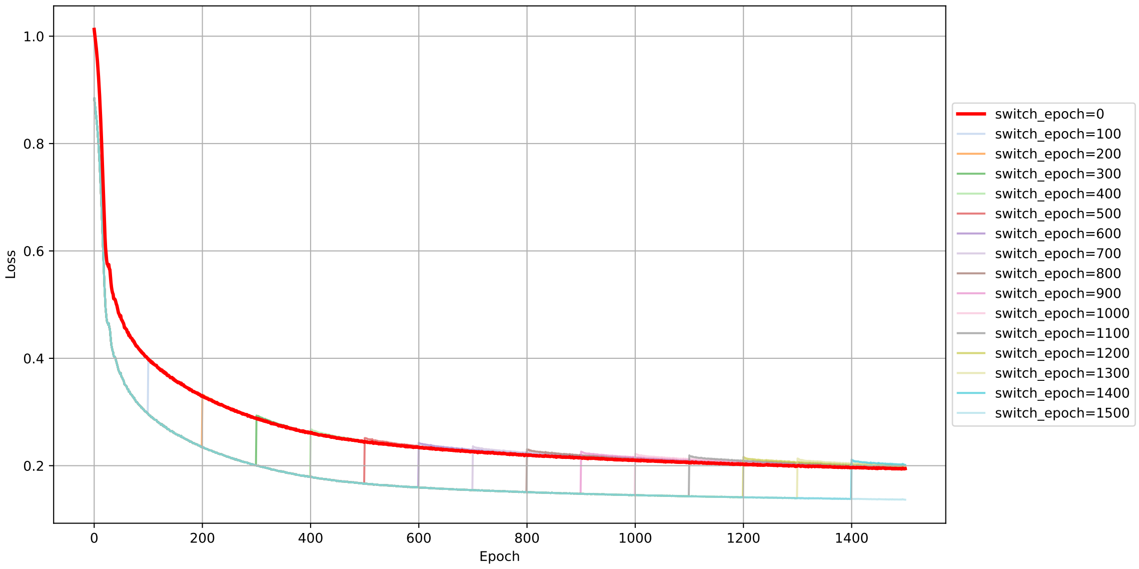


Figure 4.15: Average Training Loss Depending on the `switchEpoch` Parameter with a Medium Teacher and $\alpha = 0.6$

The graph in Figure 4.15 shows that, despite the smoother appearance, a noticeable change still occurs at the `switchEpoch` point. When distillation is turned off, the training loss experiences a temporary increase before resuming its downward trend. This pattern is consistent across different settings, similarly to what was observed in classification tasks.

Summary Overview

All average training loss curves across the various configurations for the California Housing dataset are provided in the appendices (see A.3), evaluated as the mean over 15 independent training runs for each configuration. Similarly to the Fashion-MNIST dataset, the size of the teacher model has only a minor impact on the overall training dynamics. Instead, the value of α remains the main factor influencing the shape of the training loss curves. Lower α values generally produce more noticeable shifts at the transition point.

4.2.7 Impact of the `switchEpoch` and α Parameters on the CIFAR-10 Dataset Accuracy under FGSM Attack

This section analyzes how different values of the `switchEpoch` and `alpha` parameters affect model robustness on the CIFAR-10 dataset under FGSM adversarial attacks. Accuracy is evaluated for varying attack strengths ($\epsilon \in \{0.005, 0.01, 0.05\}$), with comparisons made between models trained with and without distillation. For the evaluation, only those samples that were correctly classified under clean (non-adversarial) conditions are considered. This approach focuses the assessment on the model's robustness rather than its overall accuracy. The results provide insight into how early or late application of distillation and the weight given to the teacher's predictions influence the model's resistance to adversarial perturbations.

Detailed View: Medium Teacher, $\alpha = 0.6$, switch_epoch = 15

This configuration was selected because, under clean conditions (i.e., $\epsilon = 0$), the model with distillation achieves almost identical accuracy to the baseline model without distillation. This makes it a fair point of comparison when evaluating adversarial robustness.

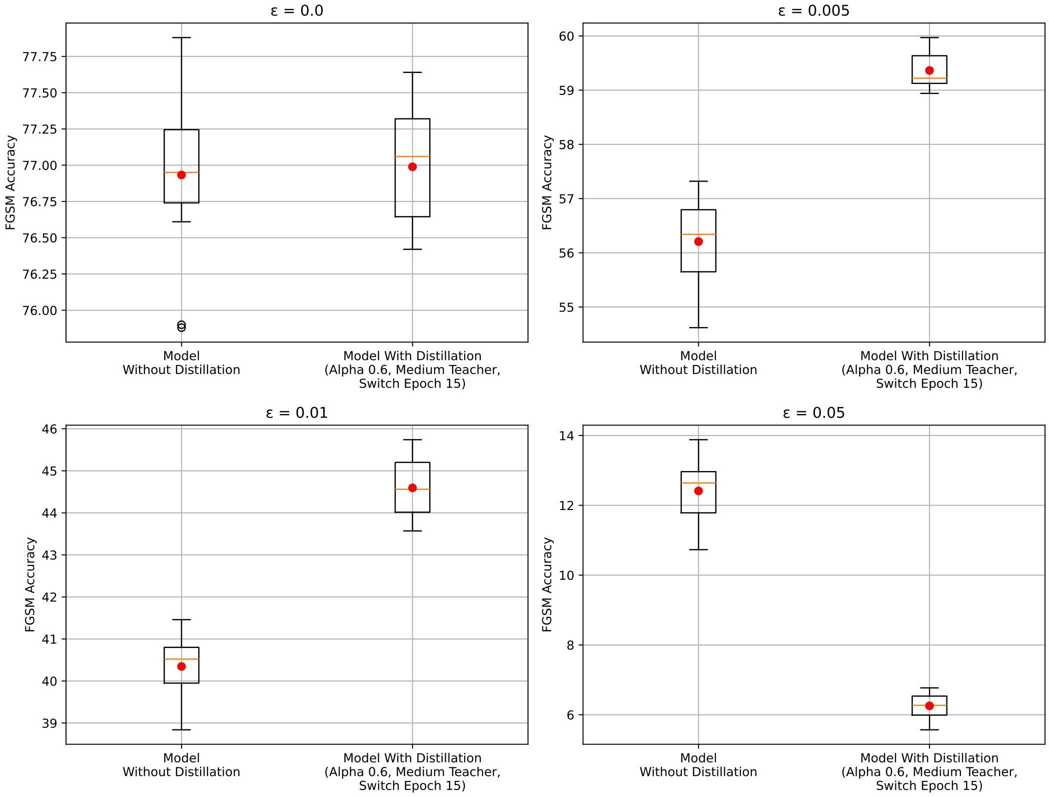


Figure 4.16: CIFAR-10 Medium Teacher with Alpha 0.6 with Switch Epoch 15 FGSM Accuracy

As shown in Figure 4.16, for both $\epsilon = 0.005$ and $\epsilon = 0.01$, the distilled model outperforms the non-distilled one under FGSM attack, suggesting improved robustness to small perturbations. Red dots represent the average across all 15 seeds, and the orange line represents the median. Interestingly, at a higher perturbation level ($\epsilon = 0.05$), the distilled model performs significantly worse than the baseline. The cause of this degradation remains unclear, but it suggests that distillation might introduce vulnerabilities under stronger adversarial conditions. However, the accuracy of both models drops close to 10%, which is equivalent to random guessing in a 10-class classification task.

Summary Overview

Figure 4.17 presents a summary overview of the average FGSM accuracy across all CIFAR-10 configurations and teacher models. The red dotted horizontal line represents avarage FGSM accuracy of the model trained without distillation.

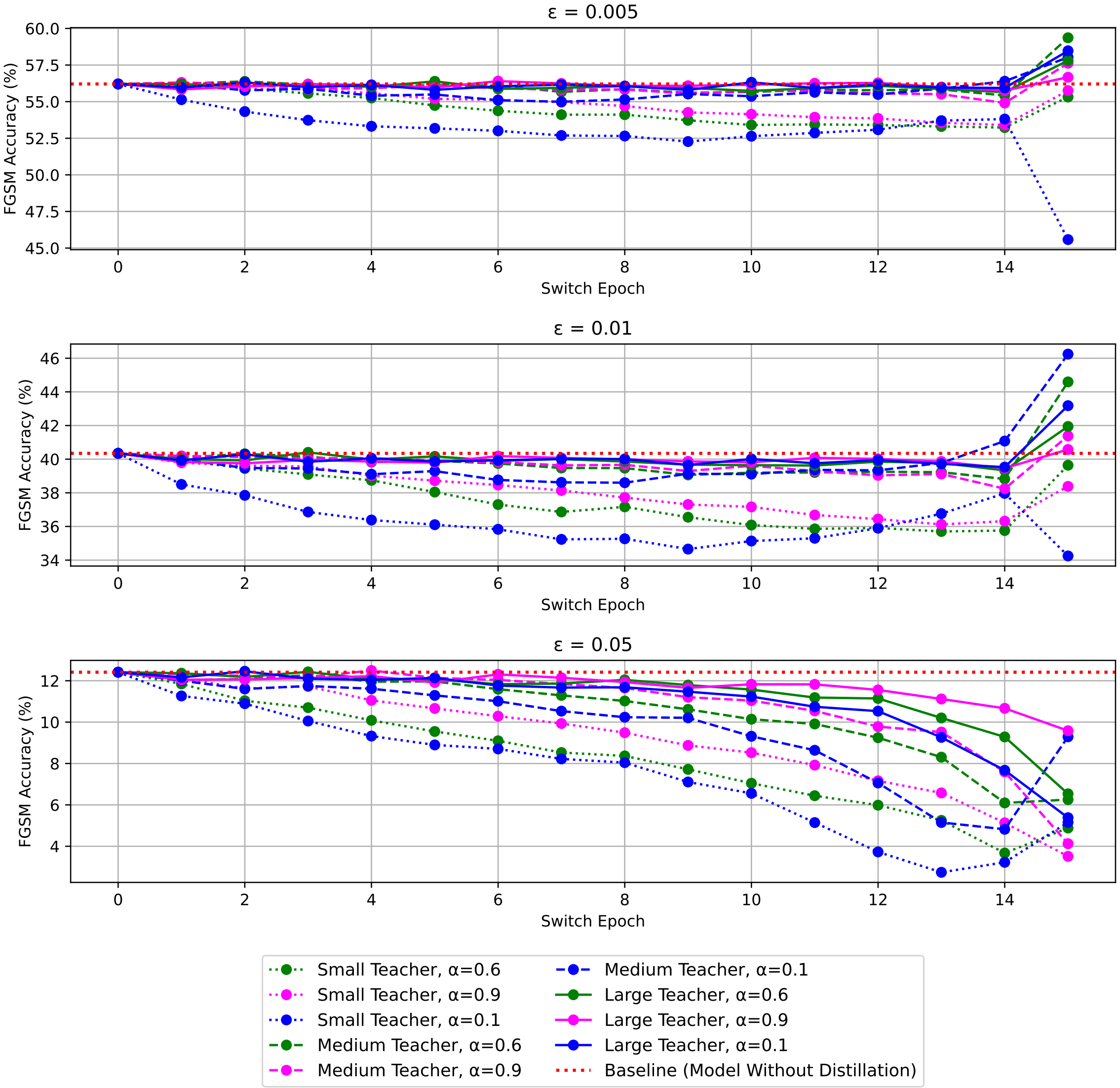


Figure 4.17: Average Accuracy under FGSM attack across all CIFAR-10 configurations

The summary reveals several key observations. Models trained with distillation with medium and large teacher models and a `switchEpoch` of 15 tend to achieve higher FGSM accuracy than those with earlier switching. Models trained with Small teacher models generally fail to overcome the baseline FGSM accuracy, this trend holds even at `switchEpoch` 15. These findings suggest that reversed distillation can improve adversarial robustness, particularly when sufficient model capacity and training time are provided.

4.2.8 Impact of the `switchEpoch` and α Parameters on the Fashion-MNIST Dataset Accuracy under FGSM Attack

Figure 4.18 summarizes the model performance under the FGSM attack for the Fashion-MNIST dataset, also with three different values of the perturbation strength ϵ.

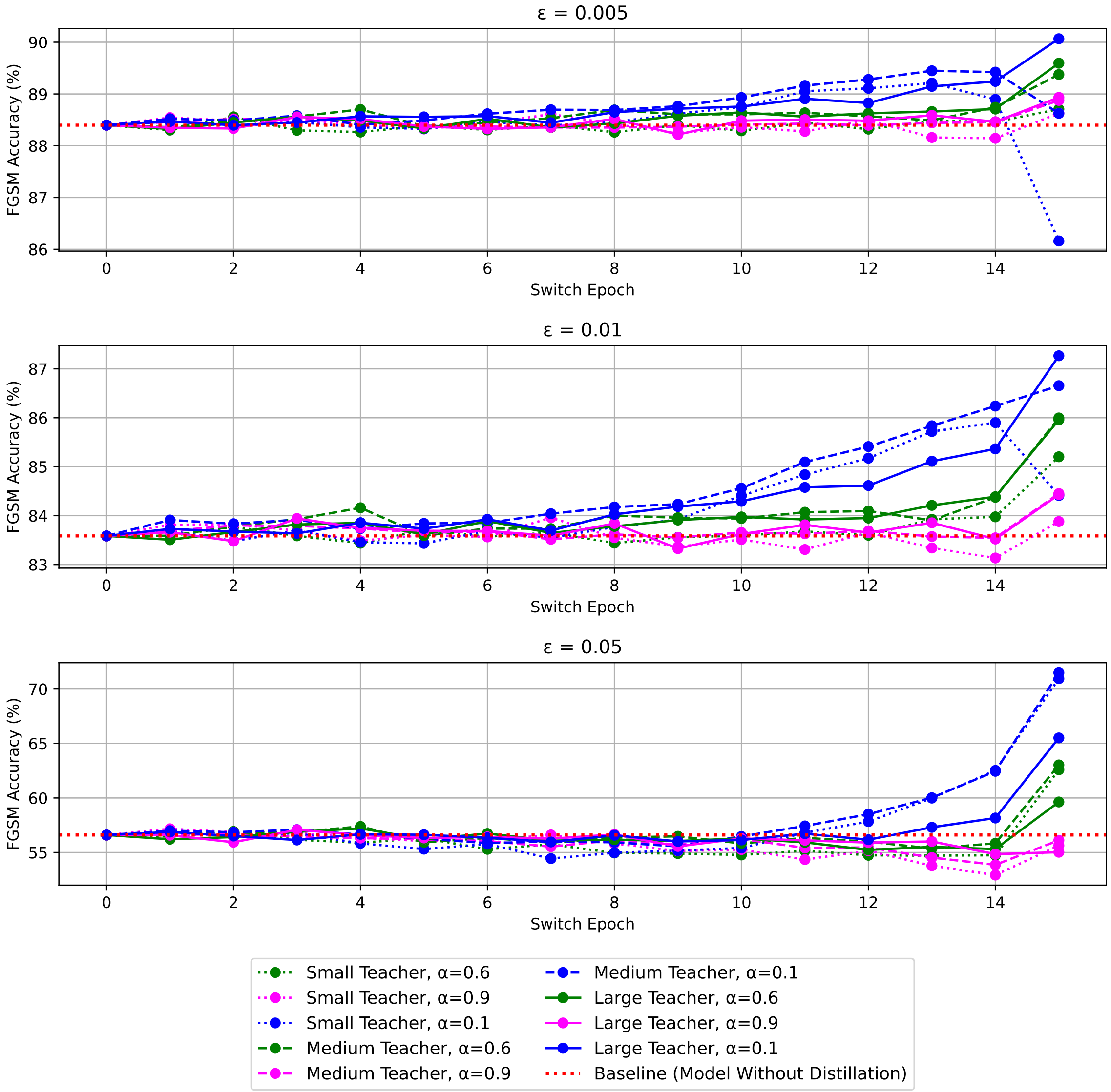


Figure 4.18: Average Accuracy under FGSM attack across all Fashion-MNIST configurations

Compared to CIFAR-10, models trained on Fashion-MNIST shows higher resilience to stronger perturbations. This is likely due to the lower complexity of the dataset, which makes it harder for small perturbations to cause a misclassification.

Similarly to previous observations, configurations where distillation was active throughout the entire training process (`switchEpoch` = 15) tend to outperform other setups. This effect is especially pronounced for lower values of ϵ, where the distillation appears to effectively improve robustness. However, for higher values of α, the accuracy tends to drop across higher levels of `switchEpoch`. This suggests that while distillation can improve robustness, overly relying on the hard loss (i.e., using high α) might lead to degraded performance under adversarial conditions.

Overall, the results show that for Fashion-MNIST, using low to moderate α values and applying distillation throughout the entire training tends to produce the best adversarial robustness.

4.2.9 Impact of the `switchEpoch` and α Parameters on the California Housing Dataset Testing Loss under FGSM Attack

Figure 4.19 shows the average testing loss under FGSM attack, again with perturbation strengths $\epsilon \in \{0.005, 0.01, 0.05\}$ for all evaluated configurations. For regression tasks such as California Housing, adversarial robustness is measured by the increase in testing loss.

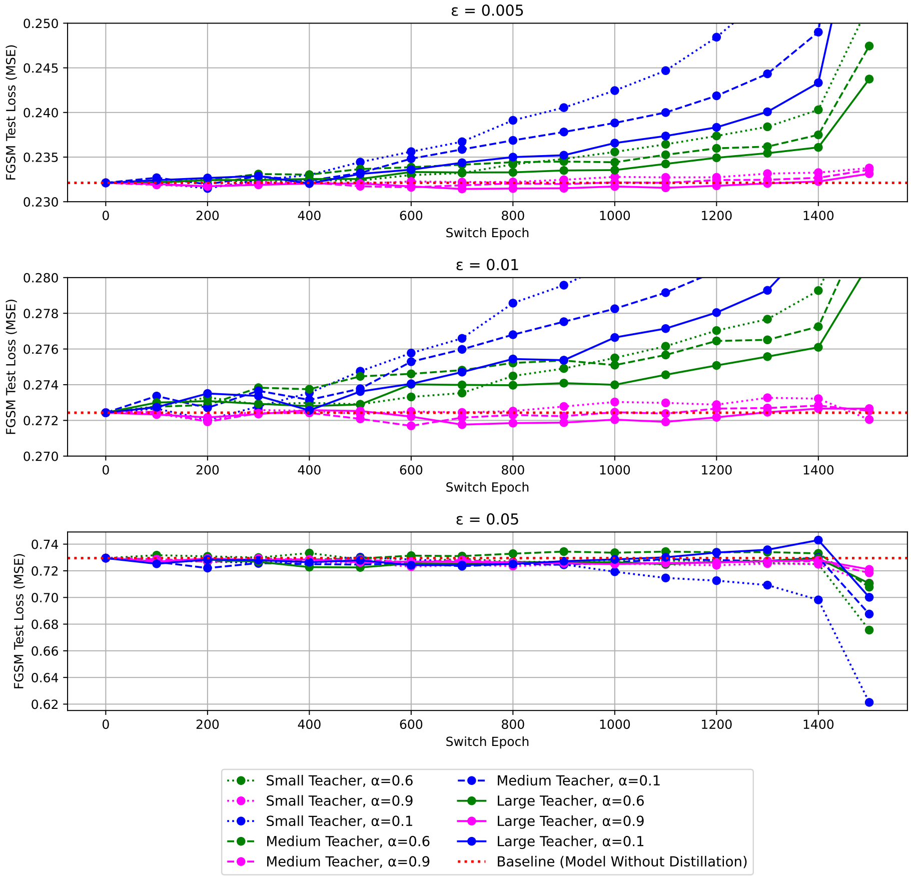


Figure 4.19: Average Testing Loss under FGSM attack across all California Housing configurations

The results suggest that, for this dataset, distillation does not significantly improve robustness against adversarial attacks at lower perturbation levels. For both $\epsilon = 0.005$ and $\epsilon = 0.01$, the testing loss increases similarly across models, with no consistent advantage for distilled configurations. These outcomes closely mirror the results observed without any adversarial attack (see Section 4.2.3).

Interestingly, at the highest perturbation level $\epsilon = 0.05$, a noticeable reduction in testing loss compared to the baseline can be observed in several distilled models. The most noticeable improvement appears for configurations with `switchEpoch` set to 1500, meaning the distillation was applied throughout the entire training process. In particular, models distilled from a small teacher with $\alpha = 0.1$ show a clear advantage. This again suggests that full-time distillation may help the student model learn smoother representations that generalize better even under stronger adversarial conditions.

4.3 Directions for Future Research

The results of the experiments demonstrated that the *reversed model distillation* technique can, under certain conditions, positively influence the training of a larger student model, particularly during the early stages of learning or when distillation is applied throughout the entire training process. Several avenues and potential improvements arise that could serve as subjects for future research:

- **Extending the Number of Training Epochs:** In this thesis, classification models were trained for only 15 epochs. The results suggest that distillation tends to be most beneficial either during the very early stages of training or when it is maintained throughout the entire training process. In future work, it would be interesting to verify whether this trend persists when training is extended to a larger number of epochs (e.g., 45 epochs). In particular, it would be worth investigating whether distillation becomes effective when applied only during the middle portion of training (e.g., between 20–80% of the total duration), or whether it continues to perform poorly in such scenarios.

- **Dynamic Adjustment of the `alpha` Parameter:** In the current experiments, the value of the `alpha` parameter was kept constant throughout the entire distillation phase. A potential improvement for future work could involve gradually adjusting `alpha` during training, for example, progressively decreasing its value, allowing the model to transition smoothly from soft loss to hard loss. This approach could naturally reduce the teacher's influence without the need for a sharp switch at a specific epoch.

- **Validation During Training:** In the current experiments, no separate validation set was used, and the models were evaluated only on the test data after training. Incorporating a validation set during training would enable, for example, automatic monitoring of potential overfitting. It would also allow better observation of how distillation affects generalization performance throughout the training process.

- **Distillation from Multiple Teachers Simultaneously:** The current approach relies on a single teacher model. In the future, it would be interesting to experiment with combining outputs from multiple teachers (e.g., of different sizes or architectures), which could provide the student with richer and more diversified training signals. These outputs could, for example, be averaged with appropriate weighting.

Chapter 5

Conclusion

This thesis explored the concept of reversed model distillation, in which knowledge is transferred from a smaller teacher model to a larger student model. The primary objective was to investigate whether this technique could serve as an alternative method of weight initialization that helps the student model learn more efficiently, converge faster, or achieve better generalization.

The experiments were conducted across three datasets – CIFAR-10, Fashion-MNIST, and California Housing – using student and teacher models of various sizes. The central idea was to guide the larger student during the initial training phase using the predictions of a smaller, pre-trained teacher model, and then continue training the student independently.

The results showed that the effectiveness of reversed distillation depends heavily on the length of the distillation phase (controlled by the `switchEpoch` parameter), the size of the teacher model, and the value of `alpha`, which balances soft and hard loss. In classification tasks (CIFAR-10 and Fashion-MNIST), distillation often led to improved performance, especially when applied throughout the entire training process. On the other hand, using distillation only in the middle portion of training often degraded performance. For the regression task (California Housing), reversed distillation generally proved less effective, and in many cases, it negatively affected the final results.

In addition to standard performance metrics, this thesis also investigated the adversarial robustness of trained models by subjecting them to Fast Gradient Sign Method (FGSM) attacks with varying levels of perturbation. The results consistently demonstrated that models trained using reversed model distillation over the entire training duration exhibited improved robustness against FGSM attacks across all datasets. This suggests that the guidance provided by the teacher model throughout training may promote learning of more stable, generalizable representations.

In conclusion, reversed model distillation can serve as a promising approach for initializing and guiding larger models, but its success is highly dependent on the specific configuration and task. The findings of this thesis also open up several directions for future research, including the use of multiple teacher models, adaptive control of the `alpha` parameter, or evaluation on more complex datasets and larger architectures.

Bibliography

[1] Agarap, A. F. Deep learning using rectified linear units (relu). *ArXiv preprint arXiv:1803.08375*, 2018.

[2] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. In: *Neural networks: Tricks of the trade: Second edition.* Springer, 2012, p. 437–478.

[3] Bishop, C. M. Neural networks and their applications. *Review of scientific instruments.* American Institute of Physics, 1994, vol. 65, no. 6, p. 1803–1832.

[4] Bishop, C. M. and Nasrabadi, N. M. *Pattern recognition and machine learning.* Springer, 2006.

[5] Brownlee, J. *Difference Between a Test Set and a Validation Set* https://machinelearningmastery.com/difference-test-validation-datasets/. 2020. [cit. 02.05.2025].

[6] Code, P. with. *CIFAR-10 (Canadian Institute for Advanced Research, 10 classes)* https://paperswithcode.com/dataset/cifar-10. [cit. 05.05.2025].

[7] Cunningham, P.; Cord, M. and Delany, S. J. Supervised learning. In: *Machine learning techniques for multimedia: case studies on organization and retrieval.* Springer, 2008, p. 21–49.

[8] Dauphin, Y. N.; Pascanu, R.; Gulcehre, C.; Cho, K.; Ganguli, S. et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in neural information processing systems*, 2014, vol. 27.

[9] GeeksforGeeks. *Vanishing and Exploding Gradients Problems in Deep Learning* https://www.geeksforgeeks.org/vanishing-and-exploding-gradients-problems-in-deep-learning/. 2025. [cit. 02.05.2025].

[10] GeeksforGeeks. *Feedforward Neural Network - GeeksforGeeks.* N.d. Available at: https://www.geeksforgeeks.org/feedforward-neural-network/.

[11] Ghahramani, Z. Unsupervised learning. In: *Summer school on machine learning.* Springer, 2003, p. 72–112.

[12] Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In: JMLR Workshop and Conference Proceedings. *Proceedings of the thirteenth international conference on artificial intelligence and statistics.* 2010, p. 249–256.

[13] Goodfellow, I.; Bengio, Y. and Courville, A. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[14] Goodfellow, I. J.; Shlens, J. and Szegedy, C. Explaining and harnessing adversarial examples. *ArXiv preprint arXiv:1412.6572*, 2014.

[15] Gou, J.; Yu, B.; Maybank, S. J. and Tao, D. Knowledge distillation: A survey. *International Journal of Computer Vision.* Springer, 2021, vol. 129, no. 6, p. 1789–1819.

[16] He, K.; Zhang, X.; Ren, S. and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: *Proceedings of the IEEE international conference on computer vision.* 2015, p. 1026–1034.

[17] Hochreiter, S. and Schmidhuber, J. Flat minima. *Neural computation.* MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 1997, vol. 9, no. 1, p. 1–42.

[18] Jabbar, H. and Khan, R. Z. Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study). *Computer science, communication and instrumentation devices.* Res. Publ Singapore, 2015, vol. 70, 10.3850, p. 978–981.

[19] Jaiswal, A.; Babu, A. R.; Zadeh, M. Z.; Banerjee, D. and Makedon, F. A survey on contrastive self-supervised learning. *Technologies.* MDPI, 2020, vol. 9, no. 1, p. 2.

[20] Klambauer, G.; Unterthiner, T.; Mayr, A. and Hochreiter, S. Self-normalizing neural networks. *Advances in neural information processing systems*, 2017, vol. 30.

[21] LeCun, Y.; Bottou, L.; Orr, G. B. and Müller, K.-R. Efficient backprop. In: *Neural networks: Tricks of the trade.* Springer, 2002, p. 9–50.

[22] Li, Y. Deep reinforcement learning: An overview. *ArXiv preprint arXiv:1701.07274*, 2017.

[23] Li, Z.; Liu, F.; Yang, W.; Peng, S. and Zhou, J. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems.* IEEE, 2021, vol. 33, no. 12, p. 6999–7019.

[24] Liu, X.; Zhang, F.; Hou, Z.; Mian, L.; Wang, Z. et al. Self-supervised learning: Generative or contrastive. *IEEE transactions on knowledge and data engineering.* IEEE, 2021, vol. 35, no. 1, p. 857–876.

[25] Medsker, L. R.; Jain, L. et al. Recurrent neural networks. *Design and Applications*, 2001, vol. 5, 64-67, p. 2.

[26] Murphy, K. P. *Machine Learning: A Probabilistic Perspective.* Cambridge, MA: MIT Press, 2012. ISBN 978-0262018029.

[27] Narkhede, M. V.; Bartakke, P. P. and Sutaone, M. S. A review on weight initialization strategies for neural networks. *Artificial intelligence review.* Springer, 2022, vol. 55, no. 1, p. 291–322.

[28] Nwankpa, C.; Ijomah, W.; Gachagan, A. and Marshall, S. Activation functions: Comparison of trends in practice and research for deep learning. *ArXiv preprint arXiv:1811.03378*, 2018.

[29] Prechelt, L. Early stopping-but when? In: *Neural Networks: Tricks of the trade.* Springer, 2002, p. 55–69.

[30] Reddy, Y.; Viswanath, P. and Reddy, B. E. Semi-supervised learning: A brief review. *Int. J. Eng. Technol*, 2018, vol. 7, 1.8, p. 81.

[31] ResearchGate. *Sparse Deep Tensor Extreme Learning Machine for Pattern Classification - Scientific Figure on ResearchGate* https://www.researchgate.net/figure/Examples-from-the-Fashion-MNIST-dataset_fig6_333997546. [cit. 05.05.2025].

[32] Ruder, S. An overview of gradient descent optimization algorithms. *ArXiv preprint arXiv:1609.04747*, 2016.

[33] Sharma, S.; Sharma, S. and Athaiya, A. Activation functions in neural networks. *Towards Data Sci*, 2017, vol. 6, no. 12, p. 310–316.

[34] Shorten, C. and Khoshgoftaar, T. M. A survey on image data augmentation for deep learning. *Journal of big data.* Springer, 2019, vol. 6, no. 1, p. 1–48.

[35] Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I. and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research.* JMLR. org, 2014, vol. 15, no. 1, p. 1929–1958.

[36] Szegedy, C.; Zaremba, W.; Sutskever, I.; Bruna, J.; Erhan, D. et al. Intriguing properties of neural networks. *ArXiv preprint arXiv:1312.6199*, 2013.

[37] Team, E. *Vanishing Gradient Problem* https://www.engati.com/glossary/vanishing-gradient-problem. 2023. [cit. 02.05.2025].

[38] Tran, P. V. Exploring self-supervised regularization for supervised and semi-supervised learning. *ArXiv preprint arXiv:1906.10343*, 2019.

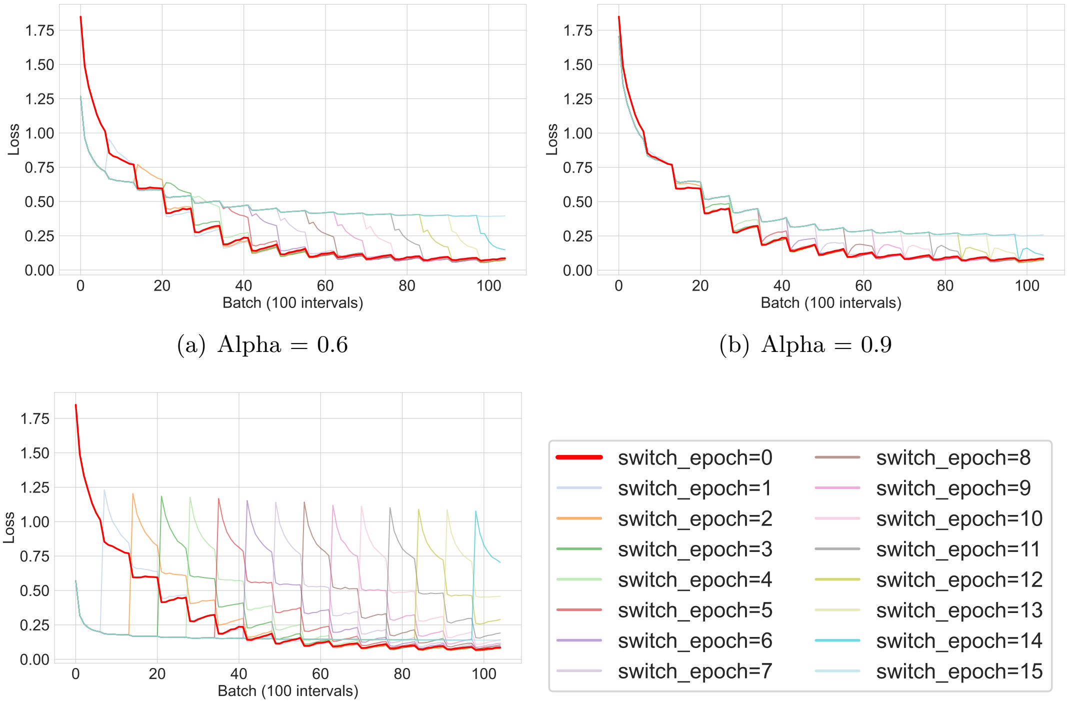
[39] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L. et al. Attention is all you need. *Advances in neural information processing systems*, 2017, vol. 30.

[40] Wu, Y.-c. and Feng, J.-w. Development and application of artificial neural network. *Wireless Personal Communications.* Springer, 2018, vol. 102, p. 1645–1656.

Appendix A

Experiments Graphs

A.1 CIFAR-10 Average Training Losses Accross Configurations



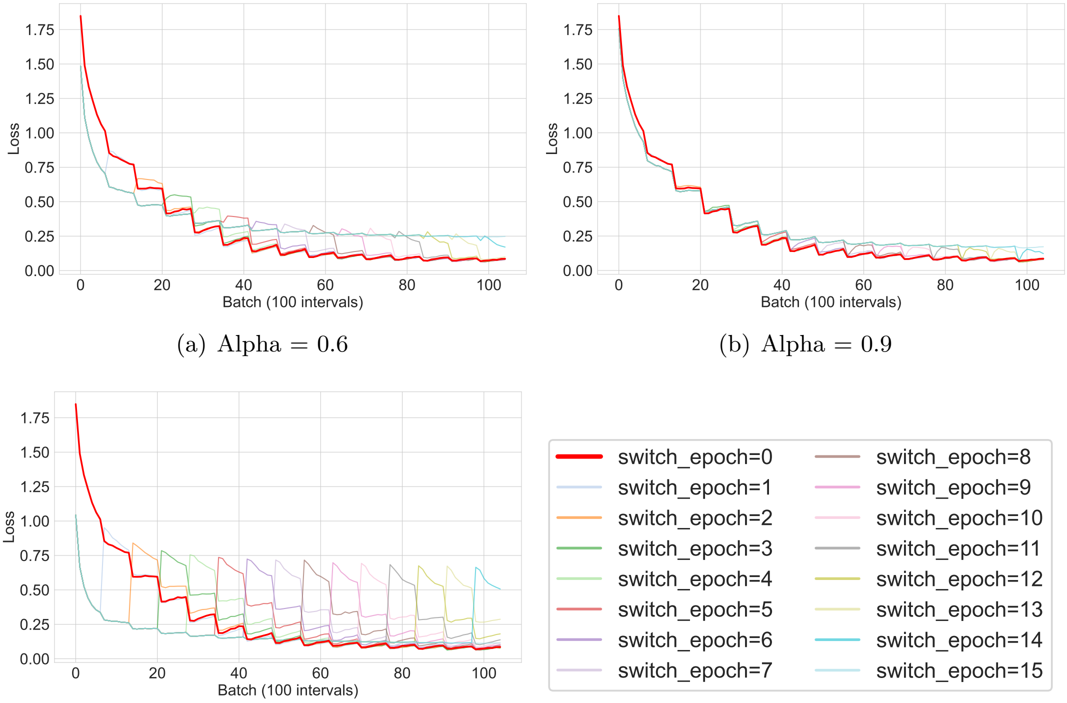
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.1: CIFAR-10 **Small Teacher Model** Training Loss Convergence



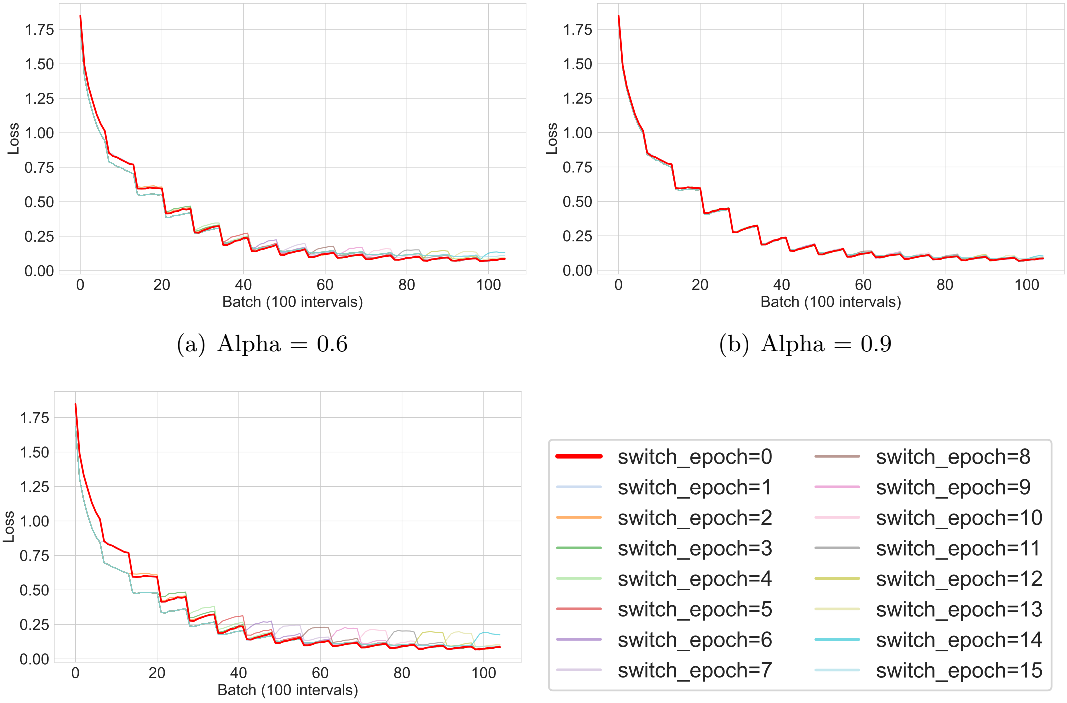
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.2: CIFAR-10 **Medium Teacher Model** Training Loss Convergence



(a) Alpha = 0.6

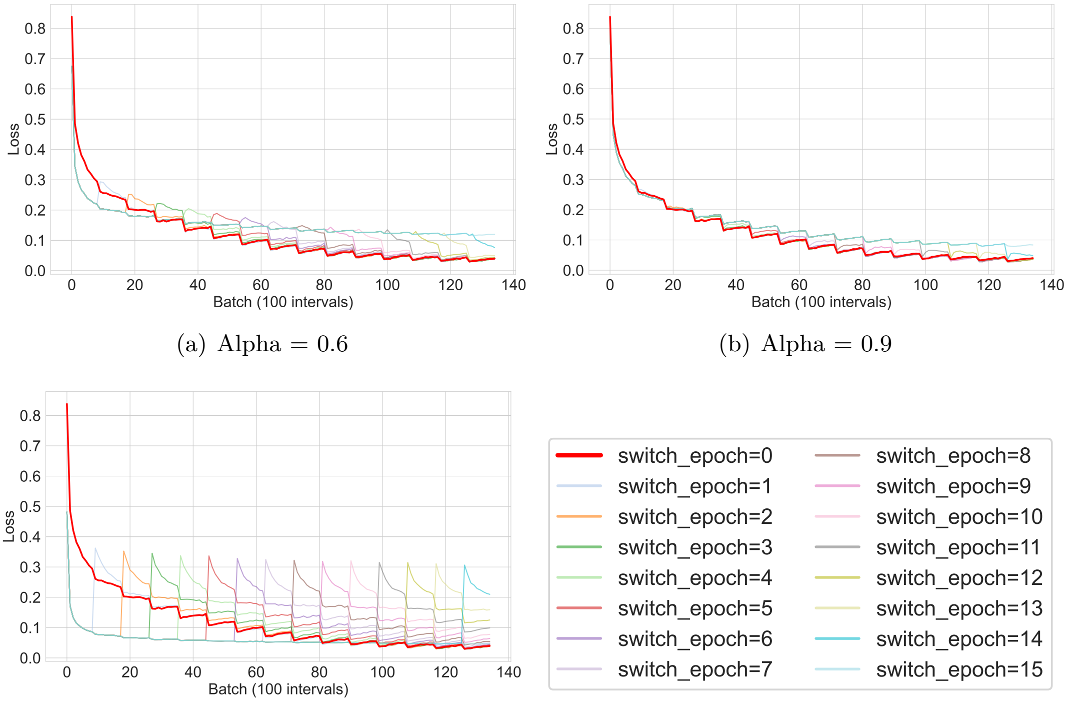
(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.3: CIFAR-10 **Large Teacher Model** Training Loss Convergence

A.2 Fashion-MNIST Average Training Losses Accross Configurations



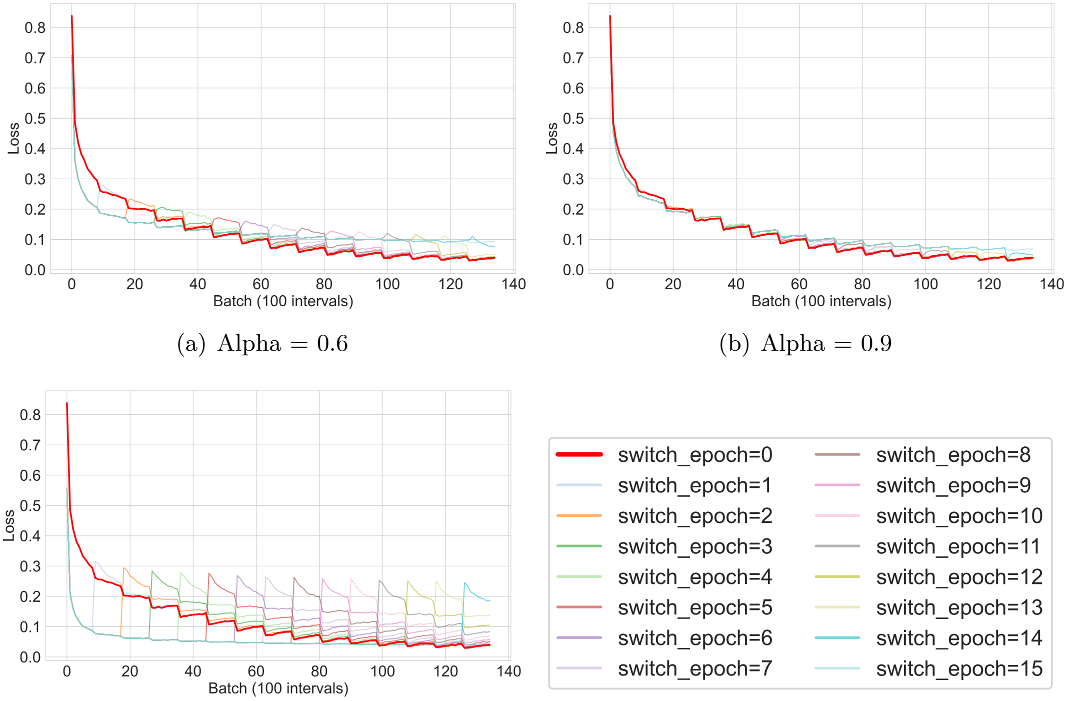
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.4: Fashion-MNIST **Small Teacher Model** Training Loss Convergence



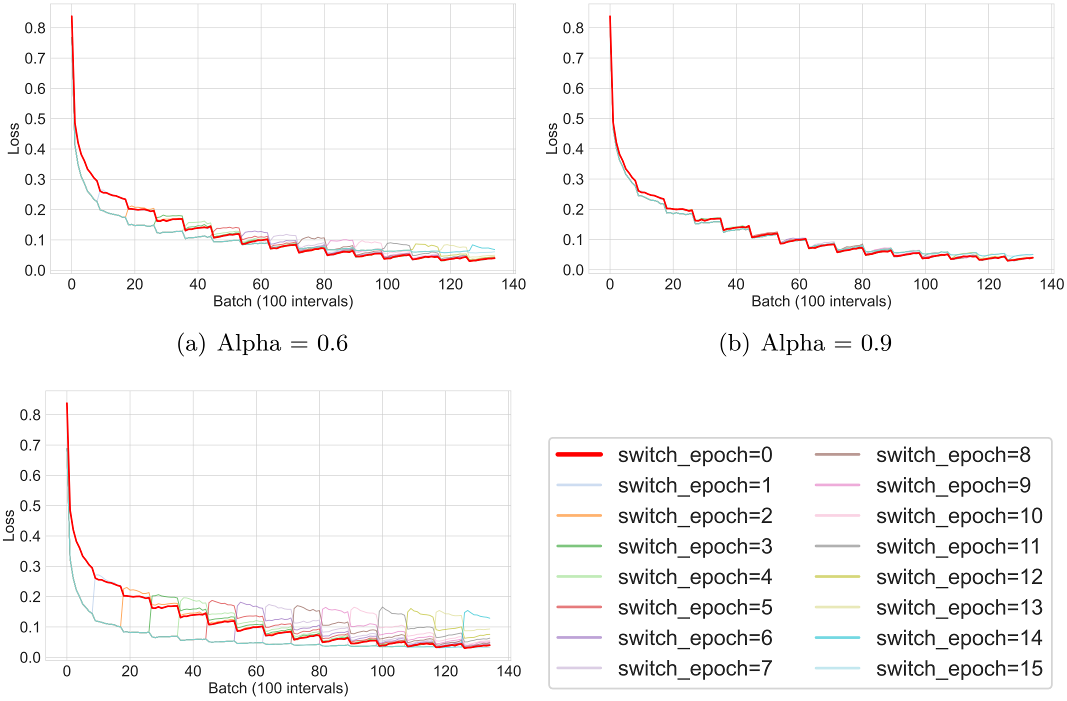
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.5: Fashion-MNIST **Medium Teacher Model** Training Loss Convergence



(a) Alpha = 0.6

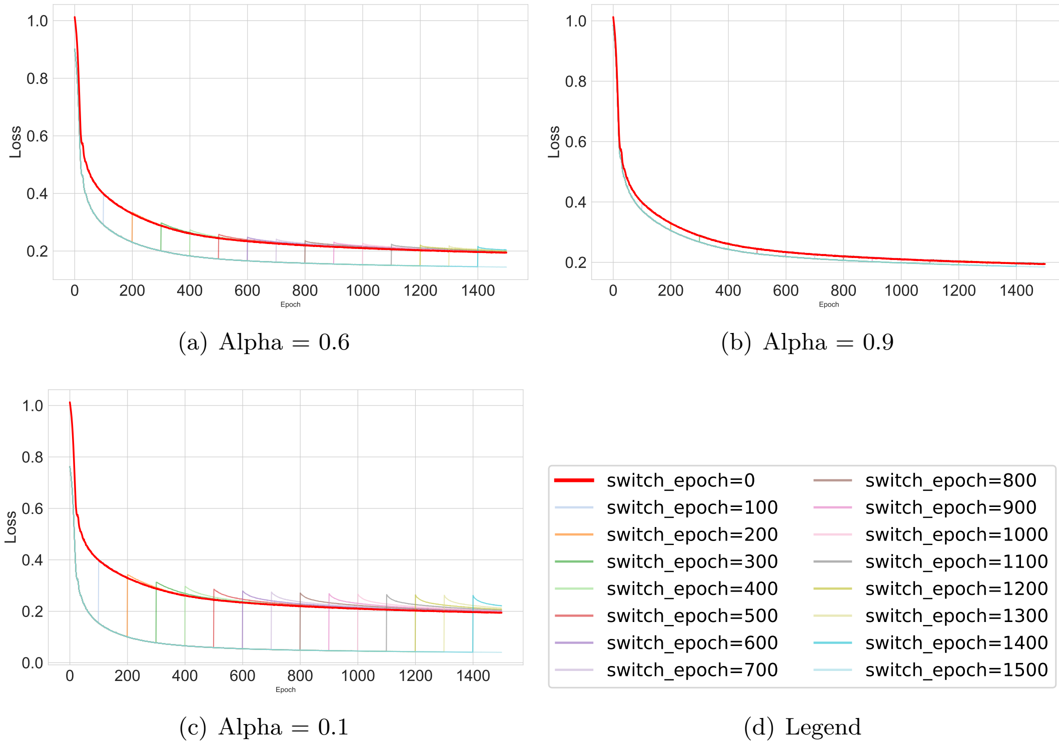
(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.6: Fashion-MNIST **Large Teacher Model** Training Loss Convergence

A.3 California Housing Average Training Losses Accross Configurations



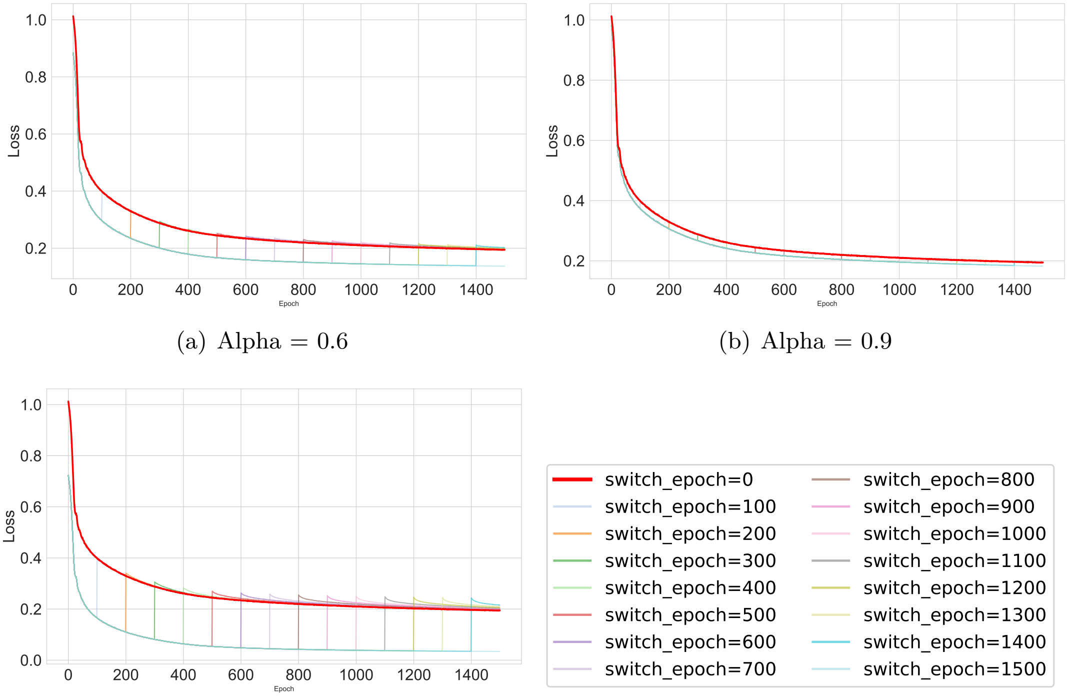
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.7: California Housing **Small Teacher Model** Training Loss Convergence



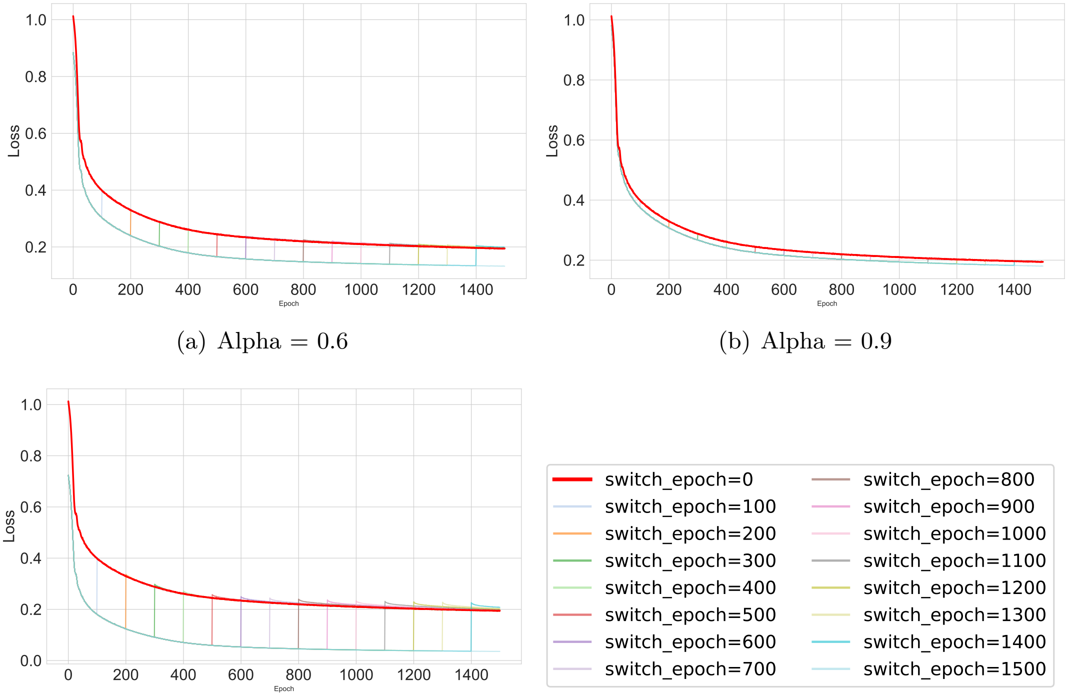
(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.8: California Housing **Medium Teacher Model** Training Loss Convergence



(a) Alpha = 0.6

(b) Alpha = 0.9

(c) Alpha = 0.1

(d) Legend

Figure A.9: California Housing **Large Teacher Model** Training Loss Convergence

Appendix B

Attached Files

B.1 Directory Structure

```
xkocia19_bachelor_thesis/
|-- scripts/      # Python scripts related to reversed model distillation
|   |-- configs/  # Configuration files for experiments
|   |-- data/     # Dataset definitions and data loading utilities
|   |-- models/   # PyTorch model definitions
|   |-- utils/    # Utility functions used across scripts
|   |-- train_model.py  # Script for standard model training
|   |                   # (without distillation)
|   |-- train_student_distil.py # Script for training models
|   |                           # using reversed model distillation
|   ‘-- fgsm_attack.py  # Script for performing FGSM adversarial attacks
|-- outputs.zip      # Output files and logs from experiments runs
|-- teacher_models/ # Best-performing teacher models
|                    (based on accuracy, trained on GPU with CUDA 12.8)
|-- visualize_outputs/   # Jupyter notebooks for visualizing results
|-- graphs/       # Plots and graphs generated from experiment outputs
|-- requirements.txt  # Python dependencies for running the project
|-- text/         # LaTeX source files of the thesis
|-- xkocia19_Weight_Initialization_in_Neural_Networks.pdf # Thesis document
‘-- README.md     # Project overview and setup instructions
```

B.2 Installation

B.2.1 Recommended environment

- Python version: 3.12.3
- Operating system: Linux (tested on Ubuntu 22.04)

First, create a virtual environment and install the required dependencies:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

B.3 Usage

Before running any script, navigate to the `scripts/` directory and set the `PYTHONPATH` environment variable:

```
cd scripts
export PYTHONPATH=.
```

Another important step is to set up script parameters inside `scripts/configs/config.py`.

B.3.1 Running the Scripts

Training a Teacher Model

To view the help message for training a teacher model, run:

```
python3 train_model.py -h
```

Example usage:

```
python3 train_model.py \
  --output ../outputs/medium_teacher_cifar \
  --batch-size 64 \
  --num-workers 6 \
  --seeds-file configs/seeds.txt \
  --model TeacherModelMedium \
  --dataset cifar10 \
  --datasets-path ../datasets
```

This command trains medium-sized teacher models on the CIFAR-10 dataset using all seeds defined in `configs/seeds.txt`.

For each seed, the following files will be saved:

- `training_loss.txt` — training loss over epochs
- `accuracy.txt` — final test accuracy
- `teacher_model.pth` — saved model weights

Training a Student Model with Distillation

To view the help message for training a student model with distillation, run:

```
python3 train_student_distil.py -h
```

It is necessary to run this script on the same device (CPU/GPU) as the teacher model, which is specified via `-teacher-path`, was trained on. The device can be selected in `scripts/configs/config.py`.

Example usage:

```
python3 train_student_distil.py \
  --output ../outputs/experiment1 \
  --batch-size 64 \
  --num-workers 6 \
  --seeds-file configs/seeds.txt \
  --dataset cifar10 \
  --datasets-path ../datasets \
  --alpha 0.6 \
  --teacher-path ../teacher_models/teacher_model_small_best_cifar_10106.pth
```

This command trains student models on CIFAR-10 using reversed model distillation from the specified teacher model with alpha parameter set to 0.6 (60% hard loss, 40% soft loss).

For each seed in `configs/seeds.txt` and for each switch epoch from 1 to `epochs + 1` (as defined in `configs/config.py`), one student model is trained. This simulates distillation from the teacher for varying durations — from only the first epoch to the entire training.

For each configuration, the following will be saved to the output directory:

- `training_loss.txt`
- `accuracy.txt`
- `fgsm_results.csv` — accuracy under FGSM adversarial attacks with different epsilon values