FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Influence of Pruning on Polysemanticity in Artificial Neural Networks

Bachelor's Thesis

in partial fulfillment of the requirements for
the degree of Bachelor of Science (B.Sc.)
in Informatik

submitted by
Joris Plettscher

First examiner: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Advisor: Asst. Prof. Dr. David Klindt
NeuroAI, Cold Spring Harbor Laboratory

Abstract

Features are measurable properties of the data that are relevant to solving a certain task. When multiple unrelated features significantly impact the activation of an individual neuron in a neural network, it is called polysemanticity. This thesis aims to investigate the influence of pruning, meaning the removal of parameters from a model, on the degree of polysemanticity in neural networks. To solve this question, a method to generate toy data with known ground truth features is presented and used to create a controlled environment in which polysemanticity can be measured. Starting with a large model that sufficiently represents the ground truth features, different pruning ratios are applied and the pruned model retrained. The polysemanticity in the retrained model is then measured. Assuming that the number of represented features is preserved through pruning, the degree of polysemanticity could be expected to increase since there are fewer neurons to represent the same amount of features. In addition to that, the Dropout technique will be examined in the context of pruning, as it encourages neurons to adapt to the potential absence of other neurons and promotes redundancy in feature representations. The experiments of this thesis could show that the training process includes an adjustment of polysemanticity, often reducing it until it converges to a certain level. The impact of pruning on polysemanticity was not definitive, though a tendency for polysemanticity to decrease after pruning was observed in some cases.

Contents

1. Introduction

As described by Haykin [Hay98], a neural network is a mathematical function that maps input data to output data through a series of interconnected layers. Formally, a k-layer neural network can be defined as a composition of functions $f = g \circ f_k \circ \ldots \circ f_1$, where $f : \mathbb{R}^n \mapsto \mathbb{R}^p$ with input dimension n and output dimension p. Each intermediate function f_i with $0 \leq i \leq k$ corresponds to a hidden layer in the network and can be defined as $f_i(x) = a(W_i x + b_i)$. Here, x is an input vector, W_i is a weight matrix, b_i is a bias vector and a is an activation function. The activation functions within these layers can introduce non-linearity, enabling the network to model complex relationships in the data. The final function g of the model serves as the output layer and varies depending on the task, such as classification or regression. The training process of a neural network involves optimizing the network's weights and biases, referred to as the parameters of the model, to minimize a specific loss function indicating the model's performance on training data.

For each problem, certain features define the ground truth. For now, features can be understood as measurable properties of the data that are relevant to solving the task [EHO+22]. Neurons within a network respond to these features, and when a single neuron is influenced by multiple unrelated features, it exhibits polysemanticity. This phenomenon complicates neural network interpretability, as such neurons cannot be directly linked to single features or tasks [MK24]. Zhang et al. provide a detailed discussion of neural network interpretability [ZTLT21], which can be roughly defined as the capacity to provide human-understandable explanations. Neural network interpretability has become increasingly important for identifying the causes of misbehavior in neural networks, understanding how they solve tasks, and explaining the factors that define better performing networks. Understanding and evaluating factors that influence polysemanticity is therefore essential for improving interpretability. Additionally, research in both neuroscience and machine learning has shown that polysemanticity can enhance performance in certain scenarios [LM20a, FMR16], further motivating its study.

An important aspect of a neural network model is its structure, defined by the number of neurons on each layer. Pruning, the process of removing parameters such as weights and biases, alters the structure of the network. A comprehensive overview over pruning is given by Blalock et al. [BOFG20]. When pruning removes entire neurons, the corresponding feature representations of these neurons are naturally also lost. Retraining the model after pruning then allows it to adapt to the modified structure. This process can potentially influence the degree of polysemanticity, measured as the average number of features represented per neuron, by redistributing or altering feature representations. This thesis investigates whether pruning consistently impacts polysemanticity and whether trends can be observed as the pruning ratio increases. The initial expectation was that polysemanticity would increase as a result of pruning, since a reduced number of neurons would need to compensate for the loss by representing the features of the pruned neurons too.

To allow for precise measurements of polysemanticity, a method for generating synthetic toy data with known ground truth features will be presented. This toy data is a set of samples consisting of an input vector and an expected output vector for the input. Using this method, controlled experimental scenarios will be created. Each experiment begins with a large neural network model designed to sufficiently represent the ground truth features. Pruning is then applied at varying ratios to reduce the network's size, and the pruned model is subsequently retrained to allow for adaptation. Neurons are removed by pruning those with the lowest mean magnitude of input weights across the entire network. Throughout this process, polysemanticity is measured both during training and after pruning to identify any changes. Additionally, the Dropout technique [HSK+12] was studied for its potential impact on polysemanticity in the context of pruning. By randomly omitting neurons during training, Dropout encourages redundancy in feature representations and helps the network adapt to the absence of individual components.

The key findings of this thesis are as follows:

- The training process adjusts polysemanticity, often reducing it until it converges to a stable level.
- The impact of pruning on polysemanticity was not definitive, though a tendency for polysemanticity to decrease after pruning was observed in some cases.

Section 2 introduces the key terminology and foundational literature relevant to this study. Section 3 then outlines the experimental design, including the creation of synthetic data and methodologies for measuring polysemanticity. Section 4 presents results from six different experiments, which are subsequently analyzed and discussed in Section 5. Finally, Section 6 summarizes the findings and suggests directions for future research.

2. Literature review

To understand how pruning of artificial neural networks affects the degree of polysemanticity, a clear understanding of the underlying terminology and existing literature is needed. For this, definitions related to features, their neural representations and the ground truth will first be established. Based on these definitions, the concept of polysemanticity can be introduced. Next, an overview of various pruning methods will be provided, followed by a description of the Dropout Technique as a potentially significant approach in the context pruning and polysemanticity.

2.1. Features and their Representations

Intuitively we can think of a feature as a property of an object or phenomenon. When using a neural network to solve a problem, we can see the observed situation

of that problem as the phenomenon. For instance, when classifying handwritten digits in images, a certain pixel being white could be a feature of this instance of the phenomenon, the observed image. The example of handwritten digits will be used throughout this section to explain the outlined concepts. A concrete example of such a problem is the *MNIST* dataset [Den12] containing images of handwritten digits with corresponding labels indicating the digit between 0 and 9. Features are organized across various levels of a hierarchy, such as a circle at a specific position being composed of multiple pixel-level features. We can therefore distinguish between features that depend on other features and those that are independent.

Definition 1 *As defined by Bishop [Bis06], a feature is a measurable property of a phenomenon.*

The concepts of ground truth, feature structure, and feature representations are still ambiguous, and a universal understanding of how to define these foundational concepts has not yet been established [EHO+22, SSJ+23]. The following definitions of these concepts have therefore been formulated specifically for this work based on the fundamental definition of a feature provided above (see Definition 1). Hence, they have been formulated solely to aid in understanding the design and rationale behind the experiments in this thesis, reflecting a possible interpretation of these concepts.

To describe the features that are present in an instance of a phenomenon, we can make use of the following definitions inspired by the work of Elhage et al. [EHO+22]:

Definition 2 *(cf. [EHO+22]) A feature coefficient vector $f \in [0,1]^n$ contains numerical values of how prominently each represented feature is present in a given phenomenon. A Feature Coefficient Space (FCS) $F = (F_1, \ldots, F_n)$ is the vector space of the feature coefficient vectors for a given set of represented features.*

So we can view a feature's value in a feature coefficient vector as the "activation" of that feature. When considering an individual pixel being white as a feature, the corresponding value in a feature coefficient vector could be 1 if the pixel is white and 0 if it is black.

For any neural network that is used to solve a specific problem, there is a range of possible desired outputs which we can call the output domain. This means that for all possible instances of a phenomenon, there exists an expected output that the model should ideally generate. As defined in Section 1, the output domain is generally $\mathcal{O} = \mathbb{R}^p$, though it may also be more constrained, such as $\mathcal{O} = \{0,1\}^p$. A feature is considered relevant for a given problem if it has any predictive power over the output, such as a relevant feature for image classification. An important requirement to construct a controlled environment in which we can evaluate the degree of polysemanticity is the knowledge of all features that play a role in the observed phenomenon. To construct a formal description of this requirement, we will need to define further concepts.

Definition 3 *A Complete Feature Coefficient Space (CFCS) $\hat{F} \neq \mathcal{O}$ is a FCS for which every instance of the given phenomenon must contain a unique combination of the represented features. This means that a unique feature coefficient vector $f \in \hat{F}$ corresponds to each instance of the phenomenon. Moreover, the represented features in $\hat{F}$ must be linearly independent of one another. This means that no feature $F_i \in \hat{F}$ may be a linear combination of the other features, defined as $F_i = \Sigma_{F_j \in \hat{F} \setminus \{F_i\}} c_j F_j$ with coefficients $c_j \in \mathbb{R}$.*

Definition 3 also entails that there exists a mapping $\omega : \hat{F} \mapsto \mathcal{O}$ that correctly maps each feature coefficient vector $f \in \hat{F}$ to the expected output of the phenomenon for which f indicates the prominence of the corresponding features in that phenomenon. This is because the feature coefficient vectors in $\hat{F}$ correspond to specific instances of the phenomenon for all of which there exists a desired output. An example of a CFCS would be the pixels' grayscale values in the images for the classification of handwritten digits. A feature coefficient vector within that space would correspond to a certain phenomenon, as it describes a specific image which again corresponds to a desired output. It is also possible that some pixels are not part of the CFCS since they are not relevant, meaning that all images would have the same value for that pixel. Further CFCS could also consist of higher-level features such as features that correspond to two or more pixels.

Definition 4 *A Perfect Feature Coefficient Space (PFCS) $\mathcal{F}$ is the FCS that contains the features of all possible CFCS of a problem. This means that the PFCS consists of all relevant unique features of a phenomenon, meaning that a feature $f \in \mathcal{F}$ may not be expressed as a linear transformation $f = a \cdot g + b$ for another feature $g \in \mathcal{F} \setminus \{f\}$ and $a, b \in \mathbb{R}$. Consequently, the possible CFCS of a problem are all distinct combinations of these features that fulfill the definition of a CFCS.*

Therefore, a PFCS is also a CFCS and we can see it as the space of all relevant features for a problem. So, overall each feature coefficient vector within the PFCS corresponds with an individual instance of the given phenomenon. Additionally, the instances of the phenomenon are already distinct based on their feature coefficient vector from any CFCS of the problem. In the context of the example of image classification, the PFCS contains all relevant features such as individual pixels and relevant shapes. In this case, the instance of the phenomenon is already distinct based on its values for each individual CFCS, such as the pixels.

Moreover, there are certain mappings that describe an observed phenomenon:

Definition 5 *An output mapping $\omega : \hat{F} \mapsto \mathcal{O}$ maps each feature coefficient vector $f \in \hat{F}$ for a CFCS $\hat{F}$ to the corresponding desired output $y \in \mathcal{O}$ of a given problem.*

The perfect output mapping $\Omega : \mathcal{F} \mapsto \mathcal{O}$ is the output mapping of the PFCS $\mathcal{F}$.

Definition 6 *An input mapping $\iota : \hat{F} \mapsto \mathcal{I}$ maps each feature coefficient vector $f \in \hat{F}$ for a CFCS $\hat{F}$ to a corresponding input $X \in \mathcal{I}$ for an arbitrary input domain $\mathcal{I}$.*

A perfect input mapping $I : \mathcal{F} \mapsto \mathcal{I}$ is an input mapping of the PFCS $\mathcal{F}$.

The output domain stays consistent for all output mappings of a given problem, regardless of the CFCS. The input domain, however, can vary between different CFCS and even for a single CFCS as it only describes the perception of the given features. Note that any input mapping can be converted into a perfect input mapping by extending the input domain to include an undefined value and by mapping features that are not represented in the original mapping to this undefined value. For instance, there is a clear mapping of which digit is written in a given image of handwritten digits. However, the input mapping depends on the "perception" of the image. It is possible that the neural network only gets a blurred version of the image or that certain pixels are left out for some reason.

In addition to that, there can be mappings between features to represent interdependencies:

Definition 7 *A feature mapping $\phi : F_1 \mapsto F_2$ maps each feature coefficient vector $f \in F_1$ to a corresponding feature coefficient vector $g \in F_2$.*

The perfect feature mapping $\Phi : F \mapsto \mathcal{F}$ maps the independent feature coefficient vectors $f \in F \subseteq \mathcal{F}$ to all feature coefficient vectors $g \in \mathcal{F}$ for the PFCS $\mathcal{F}$. The FCS of independent features F thereby is specific for each individual phenomenon.

Such a feature mapping means that the features from the FCS F_1 depend on the features from F_2. In the context of images, this can be the dependency of a shape as a feature on the individual pixels as features it consists of. Which features are independent depends on the specific phenomenon, such as the individual pixels in image classification.

If we construct a hierarchy of the dependencies between features, the desired output of the given problem can be seen as a special FCS with each output dimension as a feature that lies on the highest level of this hierarchy. Then again, the perfect output mapping can be seen as a feature mapping from the PFCS to the output domain.

Finally, we can define the requirements of a controlled environment to evaluate the degree of polysemanticity. The PFCS defines all relevant features for the observed phenomenon and therefore needs to be known. Additionally, we need to know the perfect input mapping that defines the input of the observed neural network model. To have a fully controlled environment, we will also need to know the perfect output mapping of the problem and the perfect feature mapping. The ground truth of a given problem can be defined as follows:

Definition 8 *The ground truth of a problem consists of:*

- *The ground truth features defined by the PFCS $\mathcal{F}$*
- *The perfect feature mapping Φ*
- *The output domain $\mathcal{O}$*
- *The perfect output mapping Ω*

Note that perfect input mappings are not part of the ground truth as they only represent a perception of the ground truth features. Figure 1 illustrates a simplified example of how the ground truth could be structured.

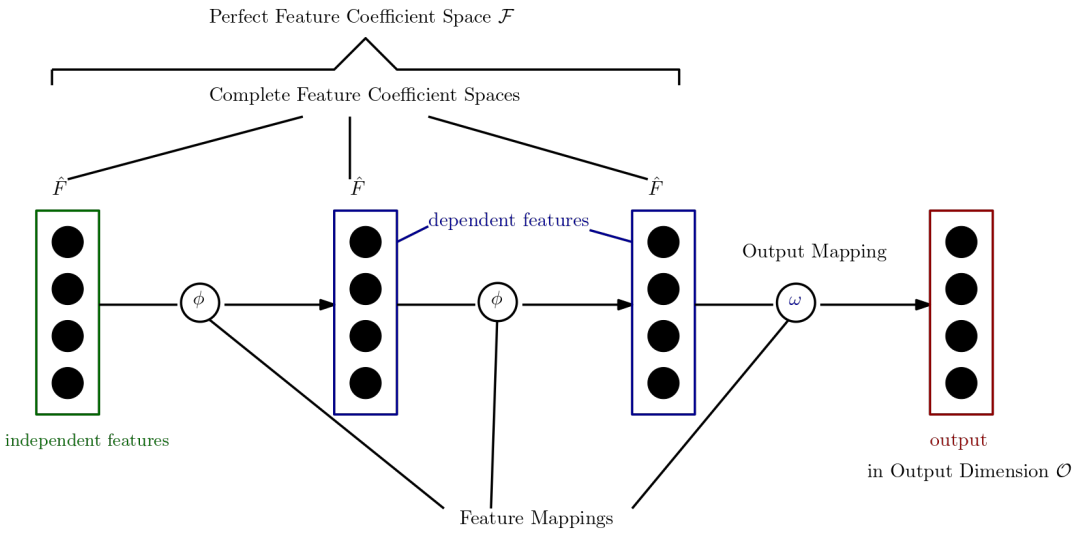


Figure 1: Illustration of the ground truth

Let us consider two possible definitions of when a neuron represent a feature:

- The neuron activates significantly if this feature is present, meaning that the output of the neuron has a significant magnitude.
- The presence of this feature significantly impacts the neuron's activation.

An example of a nonlinear activation function is a rectified linear unit (ReLU) function [NH10], which sets all negative values to zero. For the first definition, in case of a ReLU activation function, this would mean that if the gradient of the activation with respect to this feature is significantly negative, the neuron does not represent this feature. This is because the feature results in the neuron being less likely to activate. This can again be illustrated by the example of image classification. If a certain shape being present impacts a neuron's activation so that it is significantly higher than without the shape being present, this is a feature represented by the neuron. However, if a shape being present results in a much lower activation of the neuron, this would mean that the neuron does not significantly activate for the feature and would therefore not be seen as represented by the first definition.

For the purpose of this work, the second definition will be used as we want to consider a feature represented by a neuron if it has a significant impact on the neuron's response in any way:

Definition 9 *Let $n(x)$ be the activation function of the neuron n and $m(f)$ be the mapping from feature coefficient vector $f = (f_1, \cdots, f_n)$ to the input of the neuron. The neuron n*

represents feature f_i with $1 \leq i \leq n$ if the absolute partial derivative $|\frac{\partial}{\partial f_i} n(m(f))| > \theta$ for a certain threshold $\theta \in \mathbb{R}^+$ and some feature coefficient vector $f \in \hat{F}$ with a CFCS $\hat{F}$.

2.2. Polysemanticity in Neural Networks

The central concept of this thesis is polysemanticity. Before evaluating polysemanticity in neural networks, it is essential to understand what it is and why it occurs. Additionally, it is necessary to understand why pruning is considered a potential factor influencing polysemanticity and its significance in neural networks. To address these points, this section will first provide an overview of polysemanticity, followed by a discussion of its advantages in neural networks. The benefits of this concept are observed not only in artificial neural networks but also in biological ones, which will be briefly discussed as well. Finally, the foundational aspects of measuring polysemanticity will be presented.

2.2.1. Overview

The following outline of polysemanticity will be based on the work of Marshall and Kirchner [MK24]. They define polysemanticity as a neuron activating in response to multiple unrelated features. In contrast, monosemanticity refers to a neuron activating for just one feature. As discussed in Section 2.1, a neuron represents a feature if this feature significantly impacts the neuron's output. In this context, we can therefore see polysemanticity as a neuron representing multiple features. A term frequently used in neuroscience to describe polysemanticity is "mixed selectivity." From this, a similar definition of polysemanticity can be derived: polysemanticity refers to neurons responding to multiple task-relevant variables, also known as features [FMR16].

With more features than neurons per layer, the model is more likely to benefit from polysemanticity as some features would otherwise be lost from the neural representations. Polysemantic neurons allow models to make full use of the network's capacity by assigning multiple roles to individual neurons. Moreover, polysemanticity is encouraged in a model exposed to moderate noise. This is because the representations of features across the neurons can be expected to become more redundant. These beneficial roles of polysemanticity will be discussed in more detail in the following Section 2.2.2.

Marshall and Kirchner also highlight that polysemanticity is more likely to be found particularly in early and late layers, while less redundant codes are employed in intermediate layers.

2.2.2. The Importance of Polysemanticity

To illustrate the difference between monosemanticity and polysemanticity, we can look at a simplified example of a neural network distinguishing between shapes of different colors in images. Let there be two neurons, one of which activates for

squares (S) and the other for the color red (R). The information implied by these monosemantic neurons activating (1) or not (0) can be seen in the table of Figure 2.

Neuron 1 Activation	Neuron 2 Activation	Implied Information
0	0	$\neg S \land \neg R$
0	1	$\neg S \land R$
1	0	$S \land \neg R$
1	1	$S \land R$

Figure 2: Monosemantic Neurons

The monosemantic neurons can clearly distinguish between the two given features. However, for the two polysemantic neurons, as shown in the table in Figure 3, the information represented is more complex and higher-dimensional compared to the monosemantic neurons. Here, one neuron activates for triangles (T) and red, while the other activates for squares and red.

Neuron 1 Activation	Neuron 2 Activation	Implied Information
0	0	$\neg T \land \neg S \land \neg R$
0	1	$\neg T \land S \land \neg R$
1	0	$T \land \neg S \land \neg R$
1	1	$R \lor S \land T$

Figure 3: Polysemantic Neurons

This example highlights a key difference between neurons: monosemantic neurons are better at distinguishing a highly selective set of classes, whereas polysemantic neurons offer less selective, higher dimensional information. In this scenario with only two neurons, the monosemantic network provided two-dimensional information, whereas the polysemantic network was able to convey three-dimensional information. High-dimensionality is not inherently beneficial; however, as will now be discussed based on findings from the literature, it facilitates significant advantages in neural networks.

In most scenarios, the training and test data contains noise and various perturbations. As a result, the model is required to implement redundancies and error-correcting measures to ensure robustness. Polysemantic neurons enable the model to use additional dimensions to redundantly encode specific features across multiple neurons. The model thus avoids the need for entirely redundant, monosemantic neurons that would consume significant network capacity. This is why, as mentioned in the previous Section 2.2.1, neural networks tend to introduce polysemanticity when exposed to moderate noise. Redundancy is a way of handling noise and it is reflected in the distribution of specific features across multiple neurons. This redundancy, while preserving the overall number of represented features,

increases the number of features represented per neuron and therefore polysemanticity [MK24].

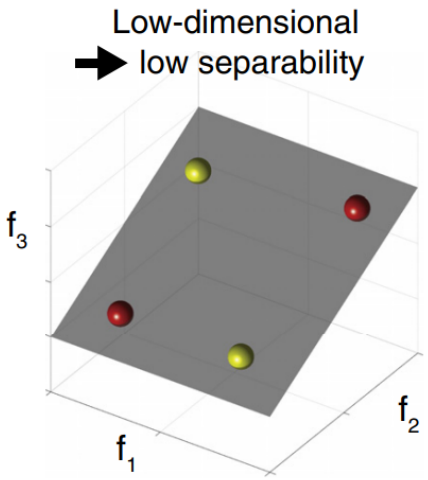
Leavitt and Morcos provide further findings on the effects of polysemanticity on robustness to perturbations [LM20b]. In this context, the term "class selectivity" is used, which is commonly found in neuroscience. Low class selectivity, or mixed selectivity, refer to the same concept as polysemanticity and are therefore used interchangeably [FMR16]. They found that mean class selectivity predicts vulnerability to naturalistic corruptions. Specifically, higher levels of polysemanticity decreased vulnerability to nearly all tested corruptions across multiple scenarios. However, the results also indicated that polysemanticity increases vulnerability to gradient-based adversarial attacks [LM20b].

Besides promoting robustness to perturbations, polysemanticity has also been suggested to have a positive impact on the overall performance of a neural network. Leavitt and Morcos could show that regularizing against class selectivity, thus increasing polysemanticity, improved accuracy by over 2% in one scenario and did not negatively affect performance when reducing class selectivity by significant factors in another scenario. Increasing class selectivity, however, has been shown to significantly decrease test accuracy across all tested scenarios [LM20a].

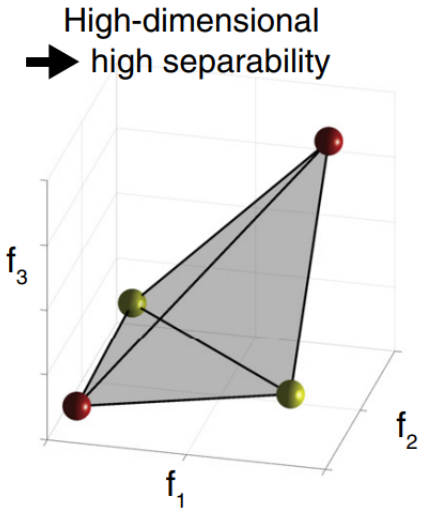
To further understand the benefits of mixed selectivity, we can turn to the neuroscience literature. Fusi et al. evaluate possible reasons behind mixed selectivity occurring in biological neural networks and the following outline of the biological nature of class selectivity will be based on their work [FMR16]. They highlight that many neurons, especially in higher-order cortex, seem to implement mixed selectivity. An example of this is the occurrence of mixed selectivity in the hippocampus, where single neurons represent multiple contextual and episodic features. They argue that mixed selectivity may be crucial for complex behavior and higher cognition. Experiments both in rodents as well as primates suggest that mixed selectivity plays an important role in representing information in a way that can be used to generate desired behavior. This is indicated by the fact that dimensionality collapses in the presence of errors, suggesting that high dimensionality may be essential for successfully completing a task.

As mixed selectivity neurons behave consistently in the same context, but their selectivity is highly context-dependent, their activations are not interpretable on their own. Rather, the meaning of their activations emerges from the context provided by other neuron's activations. As a measure of how useful these activations are for further processing structures, Fusi et al. propose determining whether a linear readout can extract the relevant information. The advantage of nonlinear mixed selectivity can then be illustrated by Figure 4 [FMR16]. Nonlinear mixed selectivity refers to a neuron's activation being related to a nonlinear combination of multiple features. The axes in Figure 4a each represent the activation for one of three different neurons and the spheres show the neuron's activations for a certain input. In this instance, the inputs consist of sound and visual contrast. Let the activation of Neuron 1 increase linearly with sound intensity, while the activation of Neuron 2 increases

linearly with visual contrast. Neuron 3 may correspond to either of these factors or represent a linear combination of both. As the neuron's activations have a linear relationship to the features, the spheres in Figure 4a are on a plane. The spheres belong to one of the two task-relevant classes, marked as yellow or red. As the yellow and red spheres cannot be separated by a plane in this case, a linear readout could not extract the relevant information. If Neuron 3, on the other hand, exhibits nonlinear mixed selectivity, the neural representations turn three-dimensional and could consequently create the vertices of a tetrahedron. The classes would then be separable.



(a) Neurons without nonlinear mixed selectivity: The spheres lie on a plane and the neural representations are thus 2D (low-dimensional). It is not possible to find a plane that separates yellow spheres from red ones (low separability).



(b) Neurons with nonlinear mixed selectivity (in Neuron 3): The spheres are the vertices of a tetrahedron and the neural representations are hence 3D (high-dimensional). It is therefore possible to find a plane that separates yellow spheres from red ones (high separability).

Figure 4: Each axis corresponds to the activation of a neuron and each sphere represents the activity of the three neurons within the space of activations. The colors red and yellow show the classes of the spheres that need to be distinguished [FMR16].

Fusi et al. highlight that the number of possible classifications by a linear readout grows exponentially with the dimensionality of the neural representations. To achieve high-dimensionality, nonlinear mixed selectivity and diversity is necessary, which means that the mixed selectivity neurons should each respond to different combinations of feature values. In contrast, it is also pointed out that high dimensionality is not always beneficial. Especially classification tasks often require selec-

tive representation of information that is relevant for the discrimination between distinct classes. This can be seen in the example described earlier in Figures 2 and 3. If the objective is to identify the presence of a square and the color red in an image, the monosemantic model performs the task most effectively.

2.2.3. Measuring Polysemanticity

The literature on polysemanticity offers several metrics for its measurement, each shaped by the adopted definitions and interpretations of underlying concepts such as features and ground truth. There are a number of approaches that focus on measuring polysemanticity when the features are known. In this section, some of these metrics will be reviewed. The described concepts will introduce some of the foundational ideas underlying the metrics used in the experiments of this thesis.

Feature Capacity

Scherlis et al. [SSJ+23] propose a method for measuring polysemanticity by examining the *capacity* allocated to each feature. Here, *capacity* refers to the fraction of an embedding dimension assigned to a feature, with values ranging from 0 to 1 as defined by the authors. A feature with a capacity of 1 occupies a full dimension in the embedding space, while a feature with fractional capacity is polysemantic, i.e. it shares its dimension with other features. A capacity of 0 indicates no representation within the embedding.

Described in more detail [SSJ+23], consider a model composed of linear layers followed by nonlinear activation functions. Each layer's linear transformation produces an embedding vector e from the input vector x, and applying the element-wise nonlinear function yields an activation vector h. For instance, the linear transformation $e = W \cdot x$ could be paired with a ReLU function $h = ReLU(e)$, where $W \in \mathbb{R}^{d \times p}$, $x \in \mathbb{R}^p$ and $e, h \in \mathbb{R}^d$. Each dimension in this nonlinear layer corresponds to a neuron. The authors assume that each dimension of the input vector x represents a distinct feature, such that each input feature is uniquely assigned to a specific dimension in the input space. The embedding vector for feature $1 \leq i \leq p$ is represented by the i-th column of W. Therefore, let $W_{\cdot,i} \in \mathbb{R}^d$ denote the embedding vector for feature i. The capacity for feature i can then be defined as

$$C_i = \frac{(W_{\cdot,i} \cdot W_{\cdot,i})^2}{\Sigma_{j=1}^{p} (W_{\cdot,i} \cdot W_{\cdot,j})^2}$$

Thus, C_i describes the fraction of a dimension allocated to feature i. C_i lies between 0 and 1 and in case all elements of $W_{\cdot,i}$ are 0, we can set $C_i = 0$. The nominator represents the size of the embedding of feature i, while the denominator captures the influence of other features on the embedding, tracking their interference [SSJ+23].

The key distinction between the metric proposed by Scherlis et al. [SSJ+23] and the context of this thesis lies in the assumption that each feature is allocated a specific dimension in the input space. However, in this thesis, we assume that there is some mapping between features and the input. If we see this input mapping as a part

of the model, functioning as the first layer, the input will consist of the features. While this would validate the assumption, it is also important to note that this metric measures capacity for a specific layer rather than for the model as a whole.

Monosemanticity

Jermyn et al. [JSH22] propose a measure of a neuron's monosemanticity based on its activations in relation to the features. Since monosemanticity represents a state opposite to polysemanticity, the reciprocal of this measure can serve as a foundation for a polysemanticity metric. To obtain an approximate measure of how monosemantic a neuron i is, the following formula is presented [JSH22]:

$$r_i = \frac{max_j(h_i(F_j))}{\delta + \Sigma_j max(0, h_i(F_j))}$$

Here, $F_j \in \hat{F}$ for a CFCS $\hat{F}$ represents the feature coefficient vector with feature j active at unit strength while all other features remain inactive. In this context, this would mean that the j-th value is set to 1 and all other values are 0. The term $h_i(F_j)$ then denotes the activation of neuron i in response to the feature coefficient vector F_j. Hence, the nominator is the activation of neuron i in response to the feature that activates it most strongly. The denominator represents the sum of the neuron's activations across all features, with a small constant $\delta = 10^{-10}$ added to prevent the measure from becoming undefined for a neuron that is inactive for all features. Moreover, only positive activation values are included in the sum [JSH22].

This metric considers a feature to be represented by a neuron if the neuron activates in the presence of that feature. However, as discussed in Section 2.1, within this thesis we see a feature as represented by a neuron if it significantly influences the neuron's activation. This requires examining the absolute gradient of the neuron with respect to the feature coefficient vectors, as the absolute gradient reflects the extent to which individual features impact the neuron's activation. The resulting metric to measure polysemanticity in the context of this thesis will be presented in more detail in Section 3.3.

Total Feature Dimensionality

When measuring polysemanticity in a model, it is essential to consider it in the context of how many ground truth features the model represents in total. For example, if a model represents only 3 features in total with an average polysemanticity of 2, meaning each neuron represents 2 features on average, this differs significantly from a model that represents 100 ground truth features with the same polysemanticity level. To reflect this distinction, the polysemanticity of the second model should be considered as representing a higher level in some sense.

As stated earlier in this section, Scherlis et al. [SSJ$^+$23] examine polysemanticity through the lens of capacity, defined as the fraction of an embedding dimension allocated to each feature. Originally, Elhage et al. [EHO$^+$22] introduced this concept under the term feature dimensionality. In this thesis, we can approach it from a different angle, adapting Elhage et al.'s concept. Aligned with the understanding of

the ground truth presented in this thesis, feature dimensionality can be defined as outlined in Definition 10.

Definition 10 *(cf. [EHO+22]) Let a neuron defined by the activation function $n : \mathcal{F} \mapsto \mathbb{R}$ and a feature $f_i \in \mathcal{F}$ with $i \in \mathbb{N}$ be given for the PFCS $\mathcal{F}$. The feature dimensionality $D_n(i) \in [0, 1]$ of feature f_i within the given neuron n is defined as $D_n(i) = \frac{|\frac{\partial}{\partial f_i} n|}{\Sigma_{f_j \in \mathcal{F}} |\frac{\partial}{\partial f_j} n|}$.*

The feature dimensionality can be understood as the fractional contribution of a specific feature to a neuron's output. In Definition 10, the numerator is the absolute partial derivative of the neuron's activation function concerning that feature. The denominator is the sum of all absolute partial derivatives for all features in that neuron.

The aim for the experiments of this thesis is to establish a metric that indicates the fraction of the ground truth features represented by the model. To quantify the extent to which a feature is represented by the model as a whole, we can sum the feature dimensionality across all neurons for that specific feature. To establish such a metric, the extent of representation for any feature should be capped at 1, disregarding any redundancies. The total feature dimensionality will reflect the proportion of ground truth features represented by the model. It is defined as the sum of the representation extent of each feature divided by the total number of features, as outlined in Definition 11.

Definition 11 *The total feature dimensionality of a model consisting of m neurons with the set of activation functions per neuron $N = \{n_i : \mathcal{F} \mapsto \mathbb{R} | i = 1, \ldots, m\}$ is defined as $\mathcal{D} = \frac{\Sigma_{f_i \in \mathcal{F}} min(\Sigma_{n \in N} D_n(i), 1)}{|\mathcal{F}|}$ with the PFCS $\mathcal{F}$.*

This can be illustrated with a simple example of a neural network with 5 neurons and a ground truth containing 5 features. If each neuron of the neural network represents one distinct features, all features are fully represented and the total feature dimensionality would be $\mathcal{D} = 1$. If, on the other hand, there are 6 features and 3 neurons each representing two distinct features, each feature would not be fully represented by a neuron and would share dimensions. This would result in a total feature dimensionality of $\mathcal{D} = 0.5$.

Integrated Gradients

The approximation of gradients plays an important role when measuring polysemanticity in neural networks because they can describe the contribution of an input dimension to the output. As described in Section 2.1, a feature is represented by a neuron if it significantly impacts the neuron's activation. Therefore, the gradient of the neuron's activation with respect to the features describes this underlying concept of polysemanticity. For this purpose, the Integrated Gradients method will now be introduced based on its description by Sundararajan et al. [STY17].

Integrated Gradients is an attribution method that is used to approximate the contribution of a dimension in the input to the output of a function $F(x)$. In our

context, the term 'integrated gradient' actually refers to the approximated integrated gradient with m interpolation steps. Integrated gradients require a baseline, which can be defined as an input without the presence of features. The gradients of $F(x)$ are summed along the path between this baseline and an input vector x where a feature is present for a specified number of interpolation steps.

Definition 12 *[STY17] The integrated gradient of a function $F : \mathbb{R}^n \mapsto \mathbb{R}^p$ along the i^{th} dimension between baseline $x' \in \mathbb{R}^n$ and input vector $x \in \mathbb{R}^n$ with $m > 0$ interpolation steps and $1 \leq i \leq n$ can be defined as follows:*

$$IG_i(x) = (x_i - x'_i) \cdot \Sigma_{k=1}^{m} \frac{\partial F(x' + \frac{k}{m} \cdot (x - x'))}{\partial x_i} \cdot \frac{1}{m}.$$

In the formula for integrated gradients, at each interpolation step k between baseline vector x' and input vector x, the partial derivative of the i-th dimension of F is computed. The mean of these partial derivatives is then calculated, which is multiplied by the distance between the baseline and the input vector in the i-th dimension. This yields the approximate contribution of the i-th dimension of the input vector x to the output in F.

Integrated gradients fulfill certain desirable properties that we require from an attribution method. A concise outline of how we can understand these properties, as described by Sundararajan et al. [STY17] in more detail, is as follows:

- **Sensitivity:** If the prediction varies for each baseline and input that differ by a specific feature, that feature should have a non-zero attribution. If, on the other hand, the function $F(x)$ does not depend on a specific feature, its attribution should be zero.

- **Implementation Invariance:** If two implementations of a network result in the same output for all inputs, the attribution values should be accordingly identical.

A simple example can be given with three input dimensions $x = (x_1, x_2, x_3)$ and $f(x) = 2 \cdot x_1 + x_3$ as the considered function of this example. The baseline is set to the vector $x' = (0, 0, 0)$ with all features being absent. To measure the attribution of x_1, this feature will be set to 1 with the remaining features remaining at 0. As a result, the input vector is $(1, 0, 0)$. Since the considered function is linear and the gradient does not change at any point, the formula of the integrated gradient in Definition 12 will result in the same value regardless of the number of interpolation steps. The integrated gradient of $f(x)$ along the first dimension between x' and x is 2. This fulfills the property of *sensitivity*. The feature x_1 has a non-zero attribution as the output changes between baseline x' and x. Considering x_2 in the same way would result in an attribution of 0, again according to the property of sensitivity.

To demonstrate *implementation invariance*, we can look at two functions $f_1(x) = h_1(g_1(x))$ and $f_2(x) = h_2(g_2(x))$ with the same input dimensions as above. Let

$g_1(x) = (2 \cdot x_1, x_3)$ and $h_1(y) = y_1 + y_2$. Moreover, let $g_2(x) = x$ and $h_2(y) = 2 \cdot y_1 + y_3$. Both functions will result in $f_1(x) = f_2(x) = 2 \cdot x_1 + x_3$ but have different implementations. Regardless of the difference in the implementation, both functions will also result in the same attributions for the features x_1, x_2 and x_3.

2.3. Pruning of Neural Networks

In this section, the current state of pruning will be presented to give an overview of this method based on the work of Blalockto et al. [BOFG20].

Pruning means removing parameters from a neural network by either setting them to zero or fully omitting them from the model. In both cases, these pruned parameters no longer contribute to the model's operations, including during training and predictions. Pruning typically starts with a large, high-accuracy network and can serve different purposes. It can reduce overfitting in an overly large network and create a more efficient model that requires fewer computational resources while retaining most of its original accuracy [BOFG20, LDS89, HS92].

Most pruning strategies start by training the initial model until the accuracy converges. Subsequently, parameters or structural elements are pruned based on a score. To reduce the loss of accuracy, The pruned model is then fine-tuned, meaning it is trained again. This process can optionally be repeated several times. As a result, the parameters of the new model may differ from those in the original, with a binary mask setting certain parameters to zero. Instead of applying a mask, parameters may also be removed entirely.

Pruning strategies can be distinguished based on several key characteristics:

- *Unstructured pruning:* Individual parameters are pruned, which results in a sparse neural network.

  Structured pruning: Certain groups of parameters, such as entire neurons, are removed.

- Options for scoring parameters include using their absolute values, trained importance coefficients, or contribution to the network's activations or gradients.

- The scores can be compared *locally*, resulting in independently pruning certain subcomponents of the network. When considering scores *globally*, all scores are compared to one another when pruning.

- The amount of parameters pruned per step can be a fixed fraction of the network or based on a more complex scheduling function. Otherwise, the desired amount of parameters can also be pruned in one step.

- Typically, the weights of the initial model will also be used as a starting point of the fine-tuning process. Moreover, it is possible for the weights to be completely reinitialized.

Blalockto et al. [BOFG20] note that global pruning tends to outperform local pruning. Moreover, they highlight that magnitude-based approaches are a common baseline in the literature and often shown to be competitive with more complex strategies. One example for this is *Global Magnitude Pruning* [FC19], where the weights with the lowest absolute value are pruned in the network. In the experiments of this thesis, this method will be applied as structured pruning by removing neurons with the lowest mean of absolute weight values.

2.4. The Dropout Technique

The Dropout technique will now be presented based the corresponding work by Hinton et al. [HSK+12]. It aims to prevent overfitting in neural networks by randomly omitting a specific fraction of hidden units, meaning neurons on hidden layers, during each training case. The issue of overfitting occurs when neurons are tuned to make accurate predictions on the training data but fail to generalize to test cases. Dropout addresses this by randomly omitting each hidden unit with a probability of 0.5. To avoid too large weights, an upper limit is set on the L2 norm of the incoming weights for each hidden unit. The L2 norm of a vector $x \in \mathbb{R}^n$ is defined as $||x||_2 = \sqrt{x \cdot x}$. If this limit is exceeded, the weights are renormalized through division. When testing the network, all hidden units are used, but their weights are halved to reflect the increased number of active units compared to the training phase. Hinton et al. highlight that applying Dropout to all hidden layers in fully connected networks outperforms using Dropout in only one layer.

This technique has the effect that a hidden unit cannot rely on other hidden units being present. As a result, instead of forming complex co-adaptations based on the training data, hidden units learn to identify relevant features with better generalization performance [HSK+12]. This improvement in extracting relevant features makes the technique interesting when evaluating feature representations in neurons and assessing the impact of pruning on these representations. Therefore, the question arises as to how the Dropout technique affects the degree of polysemanticity in neural networks, both before and after pruning.

3. Experimental Setup and Methodology

The central research question of this thesis is whether a correlation exists between the pruning ratio applied to a model and the resulting degree of polysemanticity observed after retraining. To investigate this, experiments will be conducted across several scenarios. Here, models will be trained to solve a regression task, meaning to predict numeric values represented by labels. In each scenario, an initial baseline model will be constructed with a sufficient number of neurons to minimize polysemanticity (see Section 3.4 for further details). The baseline models will then undergo pruning and retraining according to a standardized procedure that is consistent across all experiments, as described in Section 3.5. The degree of polyse-

manticity will be measured for varying pruning ratios, allowing for an evaluation of any correlation based on the metrics described in Section 3.3.

The experimental scenarios can be categorized based on the extent of available knowledge about the ground truth. In a controlled environment, the ground truth of the problem, including the relationships between features and inputs, is known. This setup allows for accurate measurement of polysemanticity, as all relevant knowledge of the features is available. A controlled environment is ideal for establishing an initial understanding of the potential correlation between pruning and polysemanticity, minimizing the impact of unforeseen or random factors. The construction of such a controlled environment for the experiments is outlined in Section 3.2.

Within a controlled environment, scenarios can further be distinguished as either linear or nonlinear. In a linear scenario, all mappings between features, inputs and outputs are linear. Accordingly, the activation functions in all models are linear. In the nonlinear scenarios of the following experiments, the activation functions will be ReLU functions [NH10]. The nonlinear scenario more closely reflects most real-world applications of neural networks and is therefore more meaningful.

In contrast, an uncontrolled environment lacks prior knowledge of the ground truth, representing a more realistic scenario where such knowledge is typically unavailable. While this setup reflects real-world conditions more closely, it introduces challenges in measuring polysemanticity with precision. Since such experiments exceed the scope of this thesis, further research is needed in this area.

The source code for this thesis, including the generation of toy data and measurement of polysemanticity, can be found in *this repository*. Additionally, it contains the code of the experiments as examples.

3.1. Training Process

The training processes within the experiments use a standardized approach with the Adam (Adaptive Moment Estimation) optimizer [KB17] and Mean Squared Error (MSE) loss function. This process enables consistent comparisons across experiments and is outlined in this section.

First, the toy dataset is split into 70% training and 30% testing data. This ensures we have enough data to train the model and evaluate its performance on new data. The models are implemented in PyTorch, a popular deep learning framework in Python [PGM+19].

The training process is iterative, involving multiple epochs during which the model learns from the entire training dataset. The number of epochs is determined based on the context of each individual experimental scenario to ensure that the model's loss converges. The selected number of epochs will be mentioned for each experiment in the corresponding results section and will remain consistent throughout the entire scenario. Moreover, a batch size of 64 is used, meaning the model processes 64 samples at once before updating its parameters.

The Mean Squared Error serves as the loss function as it is well-suited for regres-

sion tasks. MSE computes the average squared difference between the predicted values and the actual values, which allows for a clear evaluation of the model's prediction accuracy of the numerical values.

To reduce this loss function, the Adam optimizer [KB17] will be used. It is a well-known iterative algorithm created by Diederik P. Kingma and Jimmy Ba. The algorithm combines the advantages of the two optimization techniques RMSprop and Stochastic Gradient Descent (SGD) with momentum. For a detailed overview of these techniques, see the work by Ruder [Rud17]. Adam adjusts the learning rate using the squared gradients, similar to RMSprop, while also applying momentum by using the moving average of the gradients instead of just the gradients themselves, as in SGD with momentum. In the following experiments, an initial learning rate of 0.001 will always be used.

Overall, the training consists of processing all batches in each epoch to gradually improve the model's performance. For each batch, the model generates predictions through a forward pass and computes the loss. Then, backpropagation determines the gradients of the loss, which the Adam optimizer uses to adjust the model's parameters and reduce the loss.

3.2. Generating Toy Data with known Ground Truth Features

To be able to measure the degree of polysemanticity in a neuronal network, we need to be able to define which features are present in an input vector. For the toy data experiments of this thesis, we construct a fully controlled setup where the whole ground truth and perfect input mapping, as defined in Section 2.1, are known. In this case, 'toy data experiment' means that the problem is not grounded in a real-world scenario and has no meaning. The setup will consist of two neural network models.

3.2.1. Feature Model

The first model, called Feature Model, will define the ground truth of our problem. The neurons on the hidden layers of this model can be seen as the features of the PFCS and their activations define the feature coefficient vector for any given sample. The output of this model also represents the desired output, or labels, of the toy data and the activation functions of the hidden layers are the feature mappings between all features of the problem. The formal definition of a Feature Model is as follows:

Definition 13 *For the ground truth $G = (\mathcal{F}, \Phi, \mathcal{O}, \Omega)$ of a given problem, the corresponding Feature Model $FM : I \mapsto \mathcal{O}$ is a neural network consisting of the activation functions per layer $(a_0, \cdots, a_l, \omega)$ with $FM = \omega \circ a_m \circ \cdots \circ a_0$. This Feature Model fulfills the following characteristics:*

- *With $\hat{F}_i$ denoting the codomain of layer $0 \leq i \leq l$, it is also a CFCS of the problem. The neuron activations $f_i \in \hat{F}_i$ are consequently feature coefficient vectors. In addition to that, the domain of the Feature Model I is also a CFCS (see Definition 3).*

- *Based on Definition 4, the PFCS consists of $\mathcal{F} = I \cup \hat{F}_0 \cup \cdots \cup \hat{F}_l$. Hence, the neurons of the Feature Model can be understood as the features of the ground truth.*
- *The perfect feature mapping is $\Phi : I \mapsto \mathcal{F}$, where the features from the input are independent. For this mapping, the feature coefficient vector of layer k can be defined as $f_k = (a_k \circ \cdots \circ a_1)(f_0)$ with $f_0 \in \hat{F}_0$.*
- *The output domain $\mathcal{O}$ of the ground truth is also the codomain of the Feature Model. So the output of the Feature Model defines the desired output corresponding to each instance of the phenomenon. Note that, as outlined in Section 2.1, the instances of a phenomenon consist of the feature coefficient vector $f \in \mathcal{F}$ defined by the activations of the Feature Model, excluding the output.*
- *The perfect output mapping Ω maps the feature coefficient vector $f_k \in \hat{F}_k$ of each layer $0 \leq k \leq l$ of the model to the corresponding output $\omega_k(f_k) = (\omega \circ \cdots \circ a_{k+1})(f_k)$.*

All in all, the Feature Model describes an ideal mathematical model of the ground truth which maps the features to their corresponding desired output and describes interdependencies between features through the hierarchical structure defined by the layers of the model. This can be illustrated based on the example of image classification of handwritten digits. The first layer being the lowest in the hierarchy could correspond to the grayscale levels between 0 (black) and 255 (white) of each pixel in an image. The neuron activations on higher layers of the Feature Model would then correspond to higher-order features, such as edges at certain positions in the image. Note that any given instance of the phenomenon has a unique combination of values on each layer of the Feature Model. In the context of this example, each instance consists of a unique image with a distinct combination of pixels. However, not only individual pixels but also the combination of relevant higher-order features on each layer are unique for each image. To put it simple, it can be described as follows. Assuming there are an even number of pixels on each horizontal layer of the image, higher-order features could describe the horizontally neighboring pairs of pixels. Starting with the grayscale value of the left pixel l and the right value r, the higher-order feature could be $f = 1000 \cdot l + r$. For instance, $l = 155$ and $r = 30$ would result in $f = 155030$. Each instance of the phenomenon would also have a distinct combination of such feature values. This, however, is oversimplified since higher-order features are likely much more complex and not all possible combinations of such values would be relevant for a given phenomenon. Since the Feature Models used in the following experiments are not meant to describe real problems, they fulfill the given requirements of a ground truth by definition, defining their own synthetic problem.

Furthermore, the interdependencies between the features are described by the activation functions of the model which define how the features on each layer can be obtained based on the features of the previous layer. Finally, the output of the Feature Model would be values between 0 and 9 classifying the handwritten digit as the desired output.

For a realistic Feature Model, the number of features per layer decreases, which can be illustrated by the example of image classification. To classify handwritten digits, there are more small features that are relevant to the problem compared to larger features. In this case, most pixels are significant for the classification, while only a few specific shapes need to be considered when identifying digits.

The hidden layer architecture of our Feature Models is generated by iteratively adding a layer with a random number of neurons within the range defined by the lower bound $lower\_bound$ and the upper bound $upper\_bound$. This process takes three parameters that define the minimum $min\_neurons$ and maximum size $max\_neurons$ of the hidden layers, and the number of features $num\_features$ for the Feature Model. The upper bound is initialized to $max\_neurons$, while the lower bound is set to $\lfloor \frac{max\_neurons+min\_neurons}{2} \rfloor$. With the number of remaining features that need to be placed as neurons in the hidden layers stored in $remaining\_features$, the process is described by Algorithm 1. The algorithm will return a tuple of hidden layer sizes for the Feature Model based on the given parameters. It is possible that a given layer size may fall below the minimum required number of neurons; however, the algorithm will largely generate hidden layer structures that fulfill the desired properties while allowing for some randomness.

After the architecture of the Feature Model has been defined, the weights and biases are set, which determine the feature and output mappings of the ground truth. This will be done based on an importance function P, which returns random values $x \in \mathbb{R}$. In addition to that, any value x generated by P with an absolute value $|x| < \rho$ for a threshold $\rho \in \mathbb{R}^+$ will be set to 0. The weights of the Feature Model will be assigned the values derived from the importance function P, taking the threshold ρ into account. This may lead to some neurons having either no non-zero input weights or just one, resulting in meaningless features or those that are merely transformations of others. To prevent this, neurons with fewer than two non-zero input weights will be assigned additional weights based on the importance function without applying the threshold ρ, ensuring at least two non-zero weights. The biases of the Feature Model can then be set based on the expected weighted input of the neurons.

For the experiments of this thesis, the weights will be set based on a uniformly distributed importance function with values between -1 and 1. This will be done with the built-in function $torch.rand$ of PyTorch [PGM+19]. In the context of this thesis, it is sufficient to know that this function returns random values between 0 and 1, where it is equally likely for any value within that range to occur. To get values between -1 and 1, the result will then be multiplied by 2 and 1 will be subtracted, resulting in the random value $x = 2 \cdot torch.rand() - 1$. Moreover, we aim for feature coefficient vectors with a mean value of 0.5, which means that the input and output values of the neurons in the Feature Model should also have a mean of 0.5. In this case, the expected input vector μ of the layer will consist entirely of 0.5 values. For a given layer with weights W, the biases are set to $b = \mathbf{0.5} - W\mu$. Here, $\mathbf{0.5}$ is a vector of the same dimensions as $W\mu$ with all entries equal to 0.5. This will

Algorithm 1 Feature Model: Hidden Layer Generation

Require: $min\_neurons, max\_neurons, num\_features \in \mathbb{N}$
Ensure: $hidden\_layers \in \mathbb{N}^d$ for some $d \in \mathbb{N}$
  $hidden\_layers \leftarrow \emptyset$
  $remaining\_features \leftarrow num\_features$
  $upper\_bound \leftarrow max\_neurons$
  $lower\_bound \leftarrow \lfloor \frac{max\_neurons+min\_neurons}{2} \rfloor$
  while $remaining\_features > 0$ **do**
    if $remaining\_features \geq min\_neurons$ **then**
      $neurons \leftarrow random(lower\_bound, min(remaining\_features, upper\_bound))$
      $hidden\_layers \leftarrow hidden\_layers \cup neurons$
      $remaining\_features \leftarrow remaining\_features - neurons$
    else
      for $layer\_size$ in $hidden\_layers$ **do**
        $added\_neurons \leftarrow min(max\_neurons - layer\_size, remaining\_features)$
        $layer\_size \leftarrow layer\_size + added\_neurons$
        $remaining\_features \leftarrow remaining\_features - added\_neurons$
        if $remaining\_features = 0$ **then**
          break
        end if
      end for
      if $remaining\_features > 0$ **then**
        $hidden\_layers \leftarrow hidden\_layers \cup remaining\_features$
      end if
    end if
    if $remaining\_features \leq upper\_bound$ **then**
      $upper\_bound \leftarrow lower\_bound$
      $lower\_bound \leftarrow \lfloor \frac{min\_neurons+upper\_bound}{2} \rfloor$
    end if
  end while
  return $hidden\_layers$

result in an expected output per neuron of 0.5.

The Feature Model also provides a function to calculate an approximate input vector for the Feature Model that will result in the neuron activations being close to a given target feature coefficient vector. This will be done by minimizing the MSE loss between the activations and the target vector, starting with a random input vector and using the Adam optimizer [KB17]. Because of the declining layer size in the Feature Model, this might cause the algorithm to only find the optimum that sets the input vector to the corresponding first feature coefficients in the target vector. To avoid this, we will not consider the input as relevant features and only look at the features corresponding to the hidden layers. Let $\hat{\mathcal{F}}$ denote this relevant FCS. While the activation functions of the hidden layers may be nonlinear, the input layer will retain a linear activation function. The input vector of a Feature Model that produces hidden layer activations matching a certain target vector is called a seed vector.

3.2.2. Autoencoder

The second model of the setup will define how the features are mapped to the actual input of the data. An assumption that can be made about the relationship between features and the input is that there are more features than input dimensions [EHO+22, SBM22]. Hence, a simple input mapping might compress these features to form the input. To achieve this, a straightforward approach can be used: an encoder as part of an autoencoder, which takes a feature coefficient vector f and maps it to the corresponding input within the input domain $\mathcal{I}$.

Based on the work by Bank et al. [BKG21], an autoencoder, in this context, can be described as a neural network that is trained to reconstruct its input. An autoencoder consists of two main components: the encoder and the decoder. The encoder first compresses the input into a lower-dimensional, meaningful representation. The decoder then attempts to reconstruct the original input from this compressed representation. Autoencoders are trained using unsupervised learning, so no labels are required and training is based solely on input samples. The network learns by minimizing the reconstruction loss, which measures the difference between the input and output, indicating how accurately the decoder has reconstructed the input. Once training is complete, the encoder serves as the component responsible for compressing new data.

The encoder will consist of an input layer, a hidden layer containing $\lfloor \frac{|f|}{2} \rfloor$ neurons, and an output layer. For training purposes, the model will have an attached decoder that mirrors the structure of the encoder. During training, the reconstruction loss, defined as the MSE between the encoder's input and the decoder's output, is minimized using the Adam optimizer [KB17], as described in Section 3.1. The key difference is, as described above, that the desired output for a given input sample is the input itself. The input samples in each experimental scenario will consist of all generated feature coefficient vectors of the dataset. Thus, the encoder ideally learns to compress the scenario-specific features into the input as meaningful as possible.

3.2.3. Toy Data Generation

To generate toy data, the algorithm uses a list of sparsity values, where each value $s \in [0, 1]$ defines the probability of the corresponding feature being present in a sample as $p = 1 - s$. Based on this list, random feature coefficient vectors are generated with coefficients between 0 and 1 for features that occur in a sample. Afterward, the corresponding seed vectors for the generated feature coefficient vectors will be calculated. With these seed vectors as input in the Feature Model, the desired output for the feature coefficient vectors will be returned. The feature coefficient vectors will then be passed to the autoencoder for training and subsequently to get the corresponding input vector of the toy data from the encoder.

The structure of the toy data generation is illustrated in Figure 5 through an example of a possible Feature Model and encoder. The toy data generation results in samples of input-output pairs. Beginning with a seed vector highlighted in blue, the Feature Model computes the activations of the hidden layers shown in red, which correspond to the feature coefficient vector of that sample. Subsequently, the output of the Feature Model, displayed in green, also represents the toy data output. The feature coefficient vector then serves as input for the encoder, which encodes it into the corresponding toy data input shown in orange.

To give a concise overview, the parameters used for the toy data generation of the experiments can be summed up as follows:

- The number of features, $num\_features$, defines the number of neurons in the hidden layers of the Feature Model, and thus determines the relevant features in the toy data.
- The parameters $min\_neurons$ and $max\_neurons$ determine the limits of the number of neurons per hidden layer in the Feature Model. Consequently, this also specifies the limits of features per CFCS.
- The input and output dimensions, $input\_dim$ and $output\_dim$, specify the dimensions of the toy data and therefore the corresponding dimensions in the observed model.
- A list of sparsity values $sparsity$ defines the probability per feature of being absent in a sample.
- An importance function P and a threshold ρ to initialize the weights of the Feature Model.
- There are several options to determine the number of samples for generating the toy data. To ensure consistency across experiments, the number of samples will be set to $num\_samples = \alpha \cdot num\_features \cdot output\_dim$, with α fixed at 50 for the following experiments.

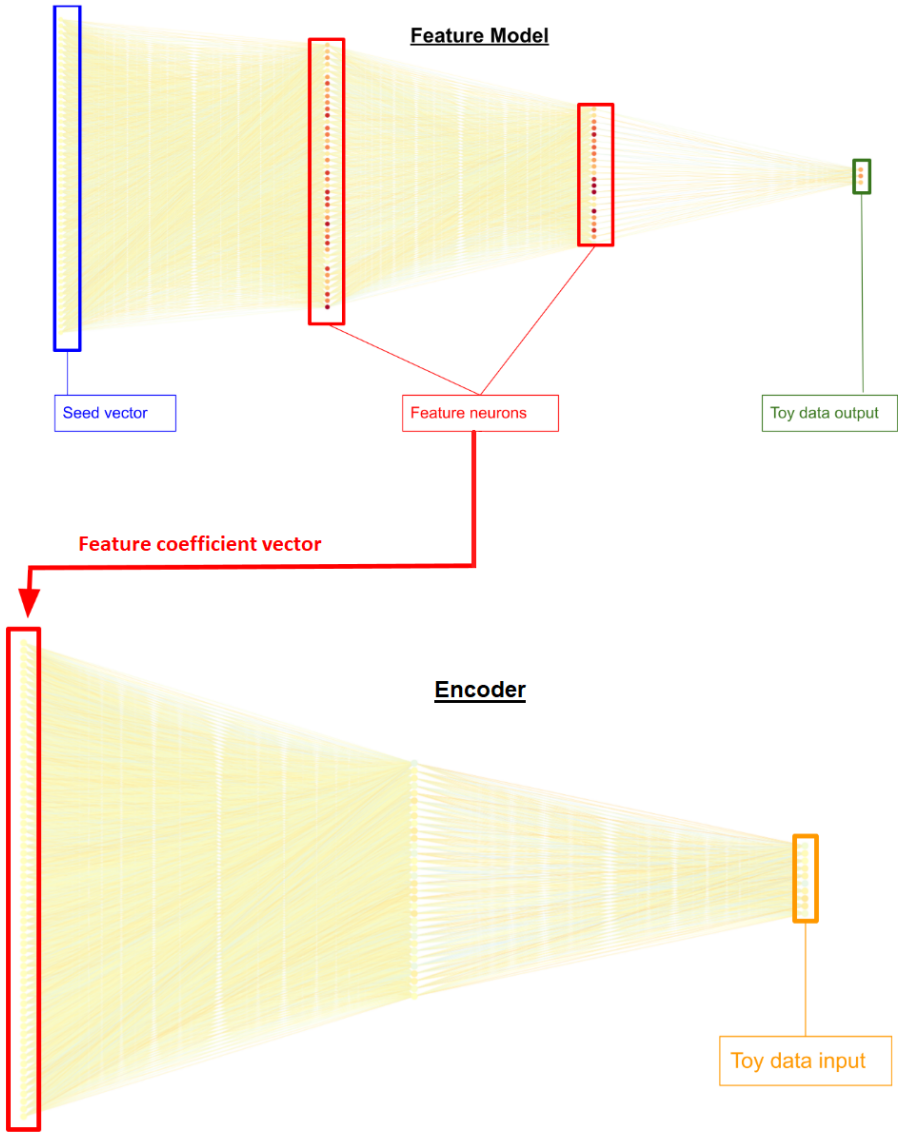


Figure 5: Toy Data Generation with a Feature Model and an Encoder

The resulting toy dataset will consist of $num\_samples$ pairs of input-output vectors consisting of numerical values, where each input vector has a dimension of $input\_dim$ and each output vector has a dimension of $output\_dim$.

3.3. Measuring Polysemanticity in a controlled Environment

Measuring polysemanticity in a controlled environment means that we have knowledge of the Feature Model representing the ground truth and the input mapping, which in this case takes the form of an encoder as part of an autoencoder. The model trained on the toy data in which we aim to measure the degree of polysemanticity will simply be referred to as the observed model. When measuring polysemanticity in a controlled environment, we focus on two relevant cases. The first case is when the Feature Model, encoder, and the observed model are all linear. In the second case, at least one of these components is nonlinear, resulting in a nonlinear system overall.

3.3.1. Measuring Polysemanticity in a linear Scenario

In a linear scenario, we can measure a neuron's degree of polysemanticity with exact partial derivatives per feature. As defined in Definition 9, the absolute partial derivative is $|\frac{\partial}{\partial f_i} n(m(f))|$, where $n(x)$ represents the activation function of the neuron and $m(f)$ denotes the function of the encoder. Due to the linear nature of the scenario, the gradient containing the partial derivatives can be evaluated at any feature coefficient vector $f \in \hat{\mathcal{F}}$. To measure polysemanticity, we aim to assess how many absolute partial derivatives of that neuron per feature are relatively high. This will show which features are represented and determine the degree of polysemanticity. Three viable options for this are:

- Setting a threshold to determine when a feature is represented and counting how many absolute partial derivatives exceed this threshold.
- The partial derivatives can also be interpreted as a probability distribution by normalizing each absolute partial derivative by the sum of all absolute partial derivatives of a neuron. Neurons representing fewer features will have lower entropy, while those representing many features will display higher entropy.
- Inspired by the metric for measuring monosemanticity outlined in Section 2.2.3, another option to measure polysemanticity is: Dividing the sum of all absolute partial derivatives by the maximal absolute partial derivative as a measurement of how significant the partial derivatives are in relation to the strongest one. A higher value indicates that many partial derivatives are relatively large, suggesting that the neuron represents multiple features. For a neuron with all gradients equal to zero, the result will also be set to 0.

The first option has the drawback of needing a manually chosen threshold, which can be difficult to set correctly. Additionally, it does not differentiate between particularly large partial derivatives and smaller ones that merely exceed the threshold. Using entropy as a measure of polysemanticity better aligns with the expectations for this type of measurement, as it effectively captures whether there are many or few significant feature partial derivatives. Since entropy doesn't provide an interpretable count of how many features each neuron represents, we can normalize it between the network's minimum and maximum entropy values for better comparison. However, a significant drawback is that neurons representing no features and those representing a single feature will have the same entropy, making monosemantic neurons indistinguishable from those representing no features.

The final option provides a measurement that does not require any parameters to be set manually and a result interpretable as the number of features represented by the neuron. Consequently, the degree of polysemanticity will be evaluated using the formula from Definition 14 during the experiments.

Definition 14 *Let $n(x)$ be the activation function of the neuron n and $m(f)$ be the mapping from feature coefficient vector $f = (f_1, \cdots, f_n)$ to the input of the neuron. For both a linear $n(x)$ and linear $m(f)$, the degree of polysemanticity of n can be defined as follows:*

$$\mathcal{P}(n) = \frac{\Sigma_{i=1}^{n} |\frac{\partial}{\partial f_i} n(m(f))|}{max_{1 \leq j \leq n} |\frac{\partial}{\partial f_j} n(m(f))|}.$$

A straightforward example of this is a neuron with two equal partial derivatives and all other partial derivatives being zero, indicating that it represents these two features. Consequently, the formula will yield a result of two as well.

3.3.2. Measuring Polysemanticity in a nonlinear Scenario

In a nonlinear scenario, the approach to measuring polysemanticity can remain unchanged. However, determining the partial derivatives of the neurons' activations per feature becomes more complex since they may not be consistent across all feature coefficient vectors. Representative values for the partial derivatives can therefore only be approximated by sampling feature coefficient vectors. Again, we can consider multiple options for this approximation:

- Random sampling of feature coefficient vectors and calculating the local gradient. The resulting approximation for each partial derivative could then be the mean or maximum of this derivative over the gradients.
- We can see integrated gradients [STY17] (see Section 2.2.3) as representative values of the partial derivatives per feature. By definition, the baseline corresponds to the feature coefficient vector with all features set to zero. For the integrated gradient of each feature, the input vector x must be defined. The

value corresponding to the considered feature will be set to 1 and all other values will be 0 in the input x. The function to which the Integrated Gradients method will be applied is the activation function of the encoded features $n(m(f))$.

In contrast to the first option, the second option utilizing the Integrated Gradients method offers a clearer approach. Due to this and the desired properties outlined in Section 2.2.3, it will be used to measure polysemanticity in the nonlinear scenarios in the following experiments.

Let p represent the number of neurons and q the number of features in a given scenario. To implement this Integrated Gradients method, the baseline vector $x' \in \mathbb{R}^q$ is initialized as the feature coefficient vector with all elements set to 0. Then, each neuron in the neural network is examined iteratively to construct its gradient with respect to the features. For each feature, the corresponding input vector $x \in \mathbb{R}^q$ is created as the feature coefficient vector with all elements being 0 except the considered feature set to 1. Using the Python library Captum by Kokhlikyan et al. [KMM+20], the attribution of each feature for the considered neuron's output is calculated via Integrated Gradients. This is done based on the baseline x' and feature-specific input vector x, using a Riemann approximation, as outlined in Definition 12, with 50 interpolation steps. In this case, the considered function for each neuron is $n(m(f))$ as described above. This process results in a matrix $M \in \mathbb{R}^{p \times q}$ where rows represent neurons and columns represent features. Each entry provides the approximated partial derivative of a neuron with respect to a feature, as computed by Integrated Gradients.

3.3.3. Measuring Total Feature Dimensionality

Once the partial derivatives of the neurons' activation functions have been calculated per feature, the model's *total feature dimensionality* as defined in Section 2.2.3 needs to be measured. Depending on the linearity of the scenario, the partial derivatives have either been calculated as outlined in Section 3.3.1 or Section 3.3.2.

The resulting matrix will have rows representing p neurons and columns representing q features, with each entry containing the partial derivative of a neuron with respect to a specific feature. These entries will be converted to absolute values to get the matrix $M \in \mathbb{R}^{p \times q}$ of absolute partial derivatives. Subsequently, these absolute values will be summed for each row to calculate $v \in \mathbb{R}^p$ representing the total of all absolute partial derivatives per neuron. By dividing each element of the matrix M in a row by the corresponding entry of the row in v, we obtain a matrix $D \in \mathbb{R}^{p \times q}$ that reflects the dimensionality of each feature per neuron, as described by Definition 10. Next, the columns of D will be summed to determine the total dimensionality of each feature across all neurons, capping these sums at 1 (so that any value greater than 1 is set to 1) to get $u \in [0, 1]^q$. Finally, by dividing the sum of all elements in u by the number of features q, we derive the total feature dimensionality $\mathcal{D}$ as described in Section 2.2.3.

3.4. Baseline Model Construction

To evaluate the impact of pruning on polysemanticity, a baseline model is required for each experiment. This baseline model must meet three key criteria. First, it should accurately predict the output to ensure that we have a model capable of meaningfully addressing the given problem. Second, it is preferable for the neurons to be monosemantic, allowing us to establish a baseline for how monosemantic the model can become when there are sufficient neurons to represent the features. Finally, it is desirable that most features are represented by the initial model. This section outlines the general process of establishing a baseline model. The specific parameters for the baseline model used in each experiment will be detailed in the respective sections of the results.

First, the model's architecture must be defined to ensure it has enough neurons to represent all features. Here, we assume the most demanding case, where all ground truth features are represented in each hidden layer. The straightforward approach used here is to allocate each hidden layer at least as many neurons as there are ground truth features, ensuring sufficient hidden layer sizes.

Moreover, the total feature dimensionality, as outlined in Section 2.2.3, will also be measured for the baseline model. A value close to 1 reflects that most features are represented by the model and is therefore an important criterion for the baseline model construction. The following experiments take an intuitive approach to increase total feature dimensionality by allocating more neurons to the hidden layers, enabling the model to represent more features.

Afterward, the model must be trained to achieve a good performance, which will be done, like all training in the experiments, using the Adam optimizer to minimize the MSE loss as described in Section 3.1. The critical factor is that the MSE loss converges at the end of training, indicating a minimum of the loss. This will be achieved with a sufficient number of epochs during the training process.

To conclude, a well-defined baseline model will serve as a reference point against which the effects of pruning on polysemanticity can be evaluated. This allows for clear comparisons across different scenarios.

3.5. Pruning and Retraining

As described in Section 2.3, the baseline model will be pruned with *Structured Global Magnitude Pruning*. To implement this pruning technique, the Python library Torch-Pruning, based on the work of Fang et al. [FMS+23], will be used. Structured pruning removes groups of parameters rather than individual ones. Due to dependencies within each group, all parameters in a group must be removed together to preserve the model's integrity during the pruning process. Torch-Pruning uses a tool called *DependencyGraph* to identify these dependencies and handle them correctly, which is described in detail by Fang et al. [FMS+23]. The library supports the implementation of custom importance criteria, which rank parameter groups to determine the order of their removal, as well as implementing a custom pruning method.

To implement Structured Global Magnitude Pruning, an importance criterion is defined to assign each neuron an importance score. Based on this, a certain fraction of neurons having the lowest scores will be removed according to a specified pruning ratio. The importance score is calculated as the mean of the absolute values of its input weights. As a result, the neurons with the lowest mean of absolute input weights will be pruned. To preserve the output dimension, neurons in the output layer are excluded from pruning. The importance scores of the remaining neurons are then compared across the entire network, creating a global ranking of neurons for pruning.

Torch-Pruning prunes parameter groups by physically removing them, unlike multiple other libraries that apply a mask to zero out parameters. This approach fully removes the parameters of a neuron from the neural network's structure, resulting in a newly structured network [FMS+23]. The retraining process that follows pruning will be adjusted to fit the neural network's new structure, while still following the same steps outlined in Section 3.1.

A pruning ratio will be given as a parameter to determine how many neurons will be removed from the network. However, this ratio does not directly reflect the exact fraction of parameters eliminated by the pruning algorithm; instead, it provides an approximate number of neurons to be removed. The actual pruning ratio pr must subsequently be computed based on the number of parameters in the new model $nparams$ and in the baseline model $bparams$:

$$pr = 1 - \frac{nparams}{bparams}.$$

The exact pruning ratios used will therefore be specified for each experiment in the corresponding results section.

4. Experimental Results

The experimental results in this section are divided into linear and nonlinear scenarios. The first linear scenario is described in more detail to provide a clear overview of the experimental setup. In general, the details of the individual experiments can be found in Appendices A to F. The subsections of the linear and nonlinear experiments both start with the scenarios involving the fewest features and progress in complexity. Key results are highlighted in this section, with further details outlined in the appendices. Finally, an overall conclusion from these experiments will be drawn in Section 6.

The parameters of the experiments were selected to create a range of scenarios with varying complexities in order to observe polysemanticity in different contexts. In particular, the number of features and the number of layers in the Feature Model, which represent interdependencies between features, were key factors for the complexity of the scenarios. Here, it might appear that the second and third nonlinear scenarios respectively resemble the first and second linear scenarios. However,

these experiments cannot be directly compared, as they are based on fundamentally different Feature Models.

According to the approach described in Section 3, each experiment begins with a neural network that serves as the baseline model. The degree of polysemanticity is then assessed during training and for different pruning ratios after retraining. Next, a baseline model with the same structure is introduced, but with the Dropout technique applied to each hidden layer, using a dropout rate of 0.5. At last, this model is then evaluated in the same manner as the first one. Additional evaluations, which did not yield informative results but were included in the experiments, are also provided in the appendices.

4.1. Pruning in linear Scenarios

To gain a clear understanding of the overall design and approach of the experiments, the first linear experiment is conducted in a simple scenario as outlined in Figure 6. This figure shows the parameter configurations used for the first experiment. In this setup, $num\_features = 32$ features represent the ground truth and flow into an input with $input\_dim = 10$ dimensions. A sparsity list was created using a function $random(a, b)$ that generates random values uniformly between $a = 0.6$ and $b = 0.8$, resulting in an average sparsity of 0.7. Consequently, the average probability of a feature being present is $p = 1 - 0.7 = 0.3$, which averages to $0.3 \cdot 32 = 9.6$ features per input sample with 10 dimensions. The output of the toy data samples consists of 2 dimensions. The number of samples of the toy data can then be calculated as $num\_samples = 50 \cdot num\_features \cdot output\_dim = 3200$.

The Feature Model was designed with a single hidden layer of 32 neurons, each corresponding to a specific feature. As a result, no interdependencies between features are present. The input layers of the Feature Models in all experiments were simply set to have the same number of neurons as there are features. So overall, the model architecture includes an input layer of 32 neurons, a hidden layer of 32 neurons, and an output layer of 2 neurons. As $linear = True$, all activation functions in neural networks within this experiment are linear. To generate the weights of the Feature Model, the importance function $P = random(-1, 1)$ generates random values uniformly between -1 and 1, with a threshold of $\rho = 0.05$.

With the experimental scenario defined, the next step is to specify the baseline model that will be trained on the data. This observed model is a neural network with layer dimensions $observed\_model = [10, 128, 32, 2]$, referring to an input layer with 10 neurons, two hidden layers with respectively 128 and 32 neurons, and an output layer with 2 neurons. After training for $num\_epochs = 50$ epochs, the MSE loss converges to approximately 0.6, and the model achieves a total feature dimensionality close to 0.95, indicating that the features are well represented.

The development of the degree of polysemanticity during the training process of the baseline model can be seen in Figure 7. Each point represents the degree of polysemanticity as the y-coordinate for the epoch given by the x-coordinate. This graph

Parameter	Configuration
$linear$	$True$
$num\_features$	32
$hidden\_layers$	[32]
$input\_dim$	10
$output\_dim$	2
$sparsity$	$\{s_i = random(0.6, 0.8) \mid 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	3200
num_epochs	50
$observed\_model$	$[10, 128, 32, 2]$

Figure 6: Linear Experiment 1 (Parameter Configurations)

shows a downward trend in polysemanticity, with an overall decrease of around 2%.

After having trained the baseline model, it can be pruned and retrained for $num\_epochs = 50$ epochs to evaluate the development of polysemanticity with respect to different pruning ratios. Figure 8a shows the resulting graph, where each point represents a pruning ratio as the x-coordinate and the corresponding polysemanticity after retraining as the y-coordinate. Although an upward trend appears after pruning around 1.5% of the neural network's parameters (a pruning ratio of 0.015), there is no significant difference in the degree of polysemanticity when comparing the baseline model to the model after pruning approximately 70%. Moreover, introducing the Dropout technique by omitting half of the neurons on each hidden layer results in a small decrease of polysemanticity after pruning 70% of the model. However, a clear trend is still not visible. This is shown in Figure 8b.

To see if the degree of polysemanticity develops differently for the individual layers of the model, the layer-wise degree of polysemanticity is plotted in Figure 9. However, both layers seem to follow a similar development, suggesting that no distinction can be made.

Additional details of this experiment are provided in Appendix A. When considering the development of polysemanticity in the context of total feature dimensionality, it can be seen that it does not contribute additional insight to this trend.

The second linear experiment introduces more features and implements interdependencies between those features through multiple hidden layers in the Feature Model. The setup and results of this experiment are described in more detail in Appendix B.

As can be seen in Figure 10a, the degree of polysemanticity increases overall by over 5% after pruning around 70% of the model. However, polysemanticity fluctuates strongly for different pruning ratios, such as the decline by around 7% between

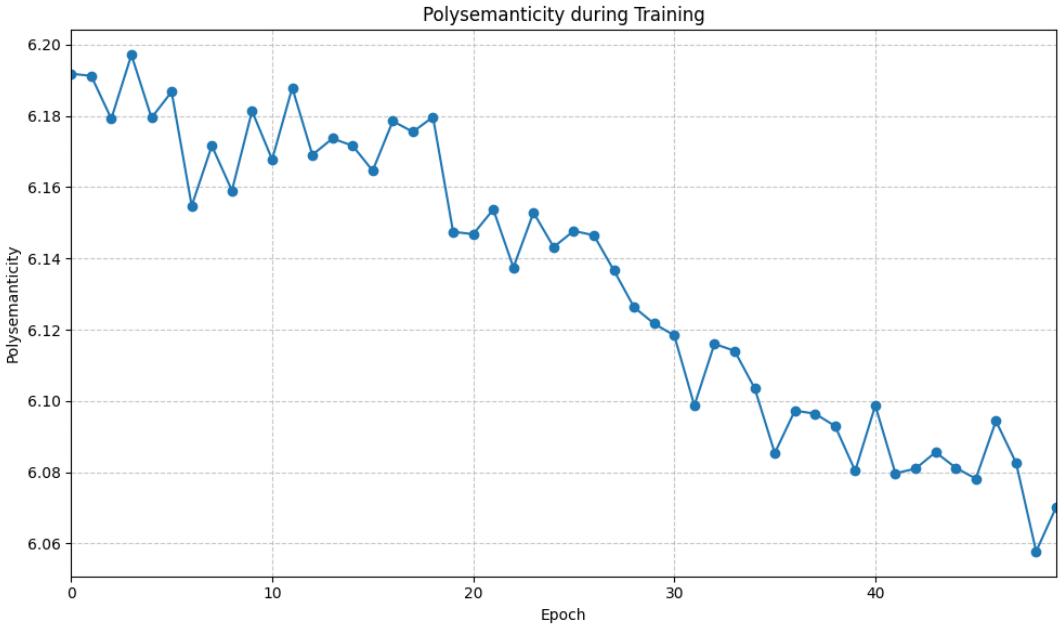


Figure 7: Linear Experiment 1 | Polysemanticity during Training

the ratios of 0.16 and 0.18. As a result, no definite trend can be concluded from this experiment. In contrast, polysemanticity in the model that implements the Dropout technique slightly decreases for higher pruning ratios, as shown in Figure 10b.

Like the first experiment, this experiment also suggests that the individual layers of a neural network seem to follow a similar development of polysemanticity after pruning, as described in Appendix B.

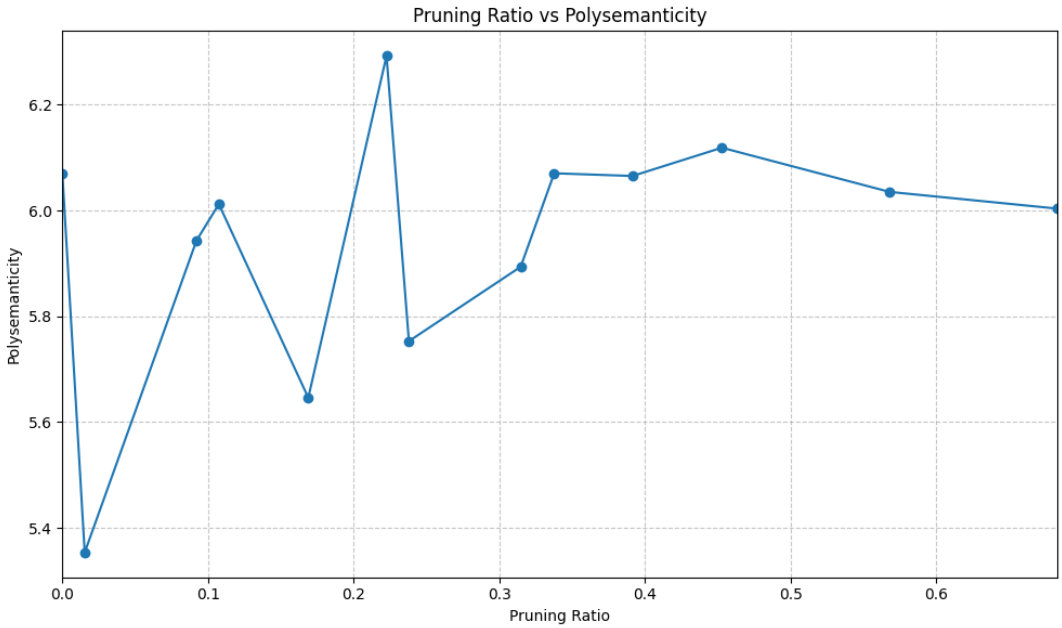
4.2. Pruning in nonlinear Scenarios

Since the nonlinear experiments more closely reflect realistic scenarios and must rely on mere approximations of gradients, four nonlinear experiments were conducted, compared to just the two linear ones.

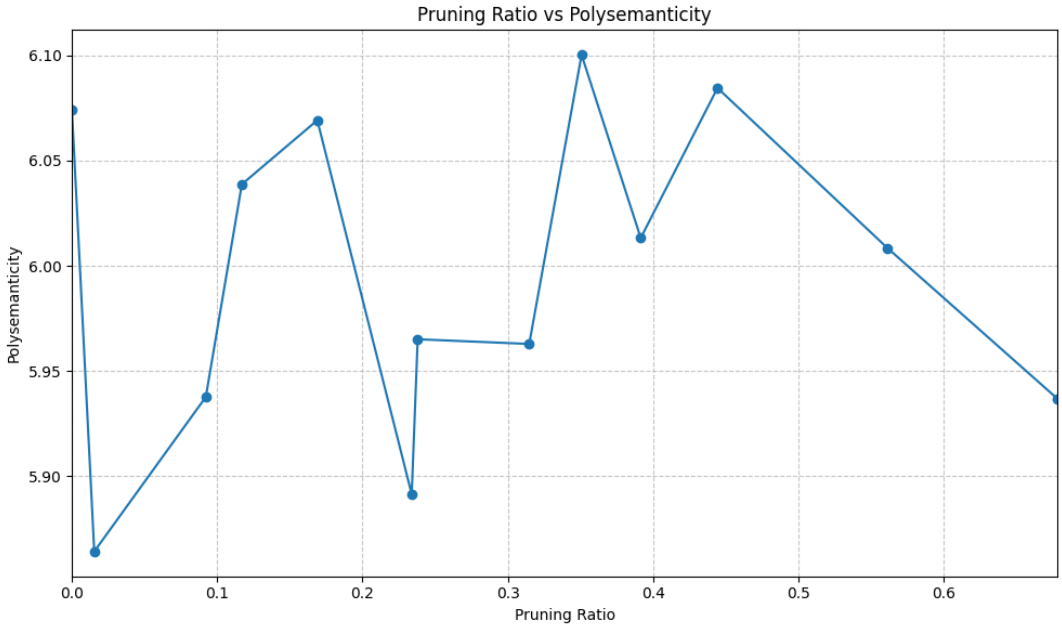
The parameter configurations of the first nonlinear experiment are outlined in Figure 11. As specified by the parameter $linear = False$, the nonlinear scenarios differ from the linear ones in that all activation functions in the hidden layers of neural networks are ReLU. The resulting baseline model converges to a MSE loss of around 0.027 and achieves a total feature dimensionality of 0.99.

This experiment demonstrates again, both with and without the Dropout technique, that polysemanticity appears to decrease throughout the training process, as shown in Figure 12.

The details of the second nonlinear experiment are explained in Appendix D. However, two results in particular stand out for this scenario. In contrast to the previous experiments, polysemanticity seems to increase during training, both with



(a) Model without Dropout



(b) Model with Dropout

Figure 8: Linear Experiment 1 | Polysemanticity after Pruning and Retraining

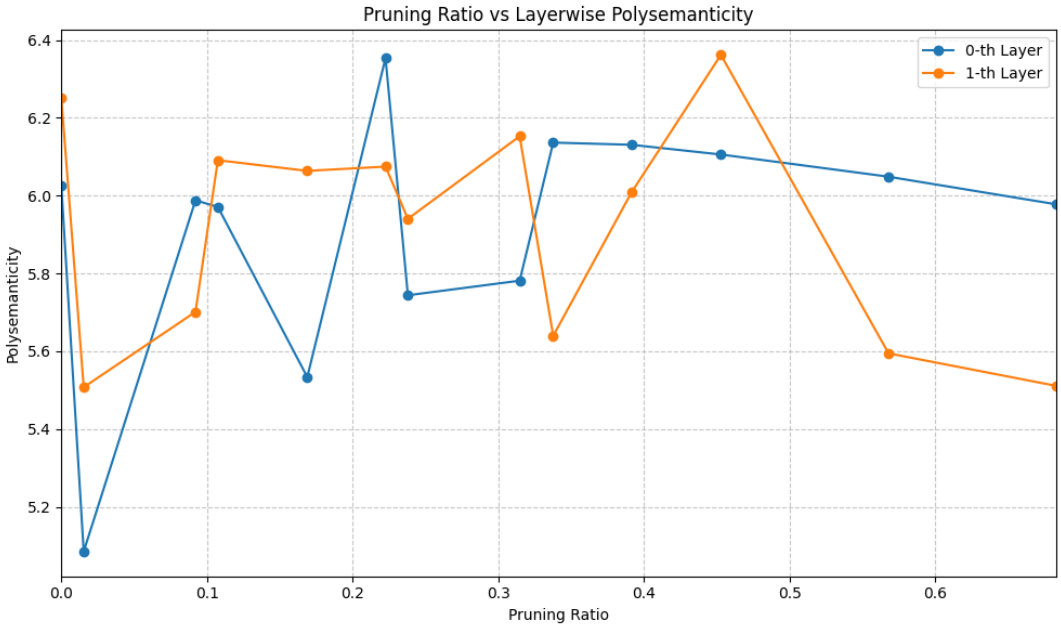


Figure 9: Linear Experiment 1 | Layer-wise Polysemanticity for different Pruning Ratios

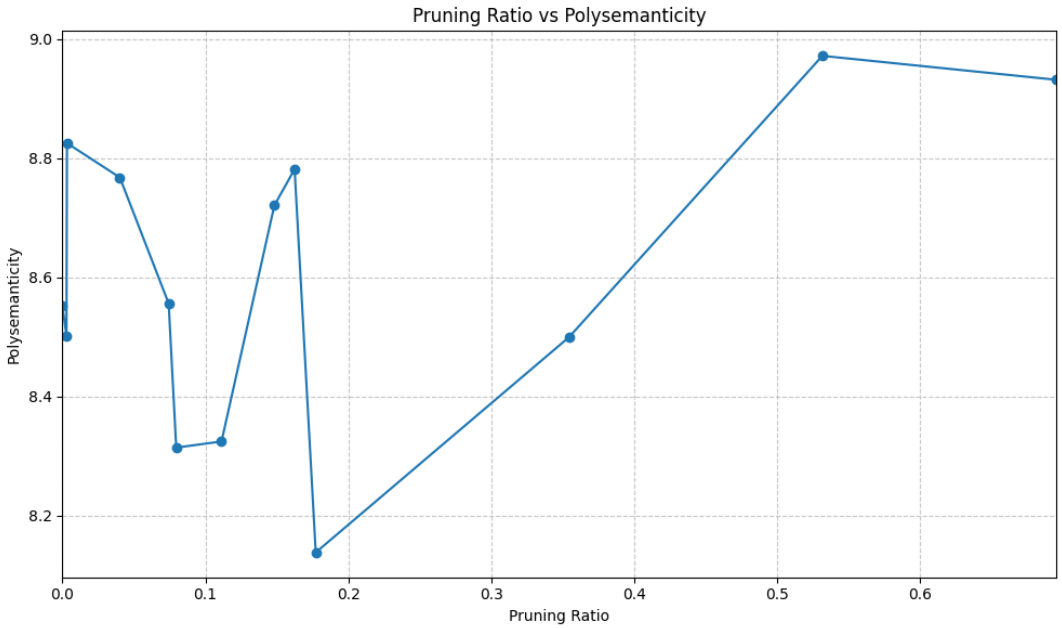
and without Dropout, as shown in Figure 13. Additionally, a clearer trend of polysemanticity after pruning is visible for both models, as can be seen in Figure 14. Without Dropout, polysemanticity decreases by about 5%, while with Dropout, it drops by 11% after pruning more than 70% of the baseline model.

In the third nonlinear experiment, as described in Appendix E, polysemanticity appears to develop in opposite directions during training for the models with and without Dropout. This is shown in Figure 15, where polysemanticity increases by approximately 6% without Dropout and decreases by around 19% with Dropout. The results in Appendix E also show that polysemanticity tends to be lower for higher pruning ratios without Dropout. With Dropout, however, no trend is apparent.

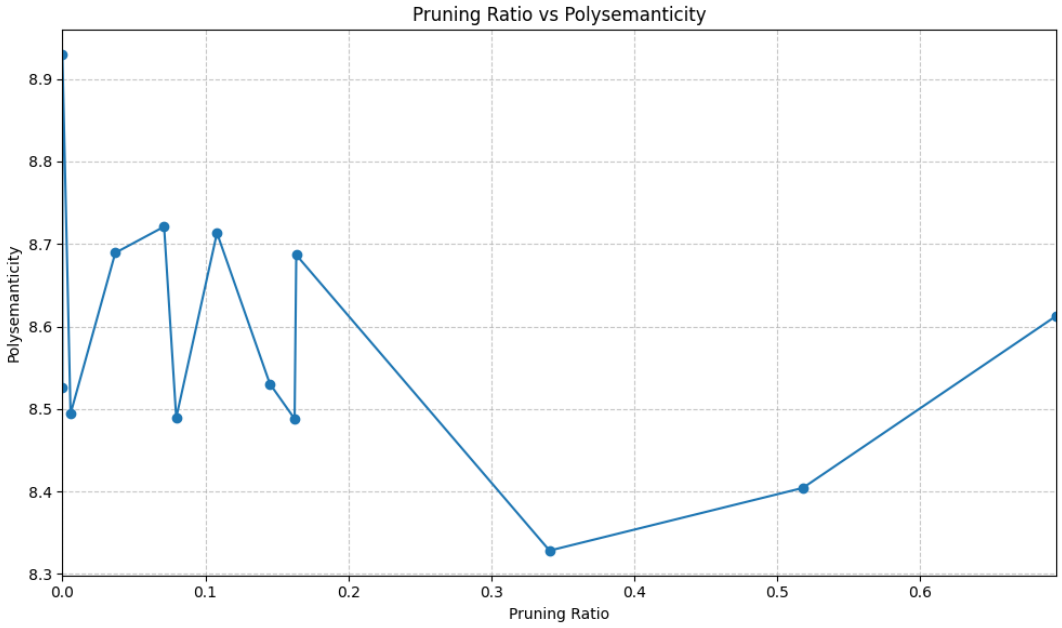
Figure 16 shows that in the final nonlinear experiment, polysemanticity declines at higher pruning ratios both with and without Dropout. Additional details on this experiment are provided in Appendix F.

5. Discussion

The experiments demonstrated that the training process adjusts the degree of polysemanticity within the models, often revealing a clear trend toward decreasing polysemanticity, though increases can be observed in some cases too. Figures 12, 13 and 15 highlight that models incorporating the Dropout technique exhibit steeper



(a) Model without Dropout



(b) Model with Dropout

Figure 10: Linear Experiment 2 | Polysemanticity after Pruning and Retraining

Parameter	Configuration
$linear$	$False$
$num\_features$	16
$hidden\_layers$	$[12, 4]$
$input\_dim$	5
$output\_dim$	2
$sparsity$	$\{s_i = random(0.65, 0.75) \mid 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	1600
num_epochs	50
$observed\_model$	$[5, 64, 16, 16, 2]$

Figure 11: Nonlinear Experiment 1 (Parameter Configurations)

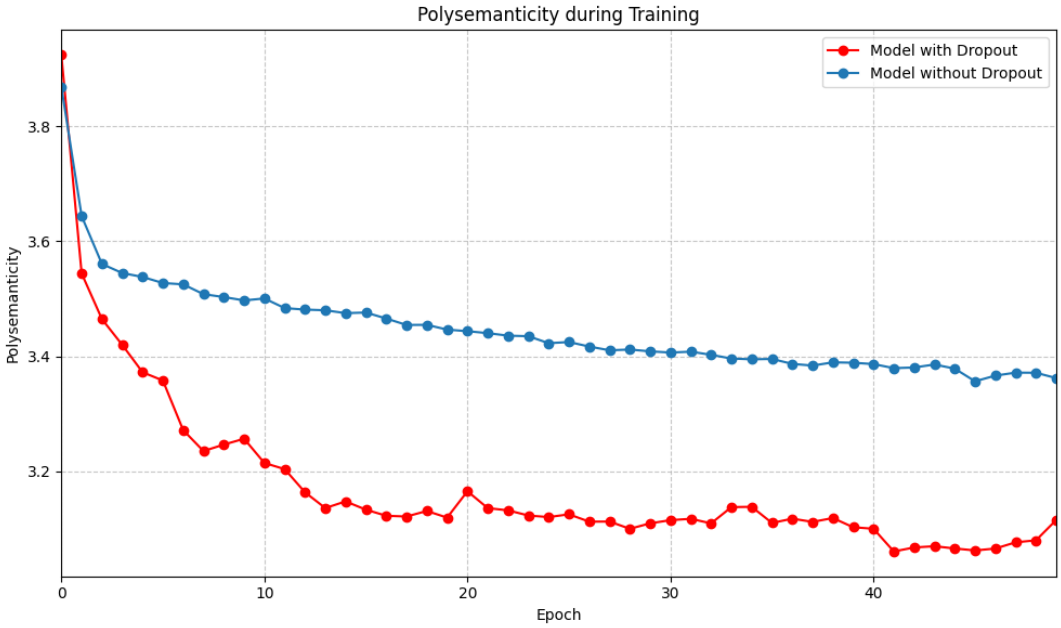


Figure 12: Nonlinear Experiment 1 | Polysemanticity during Training with and without Dropout

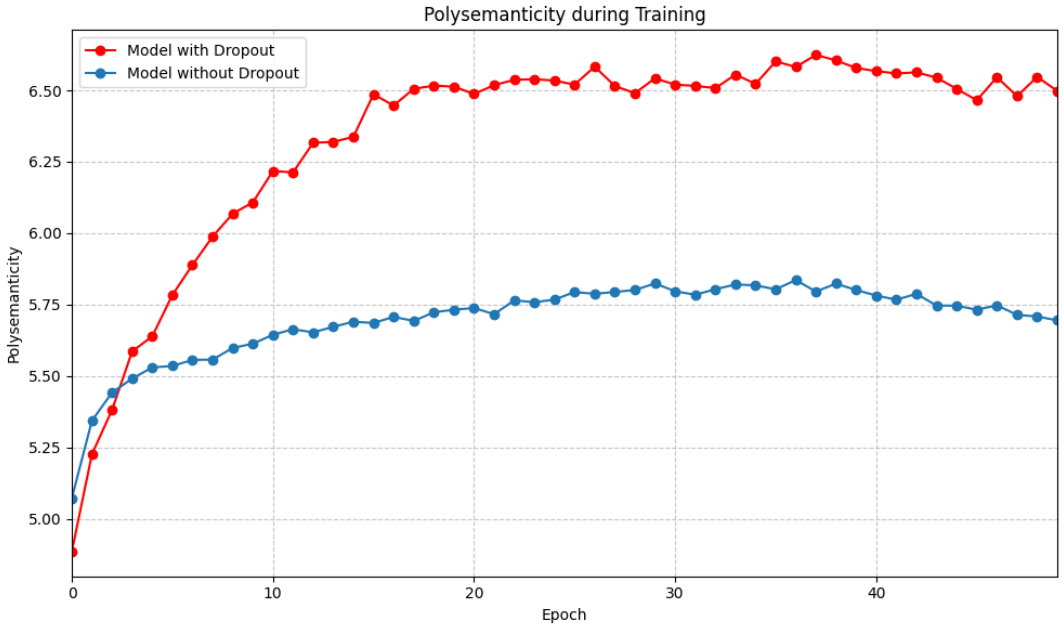


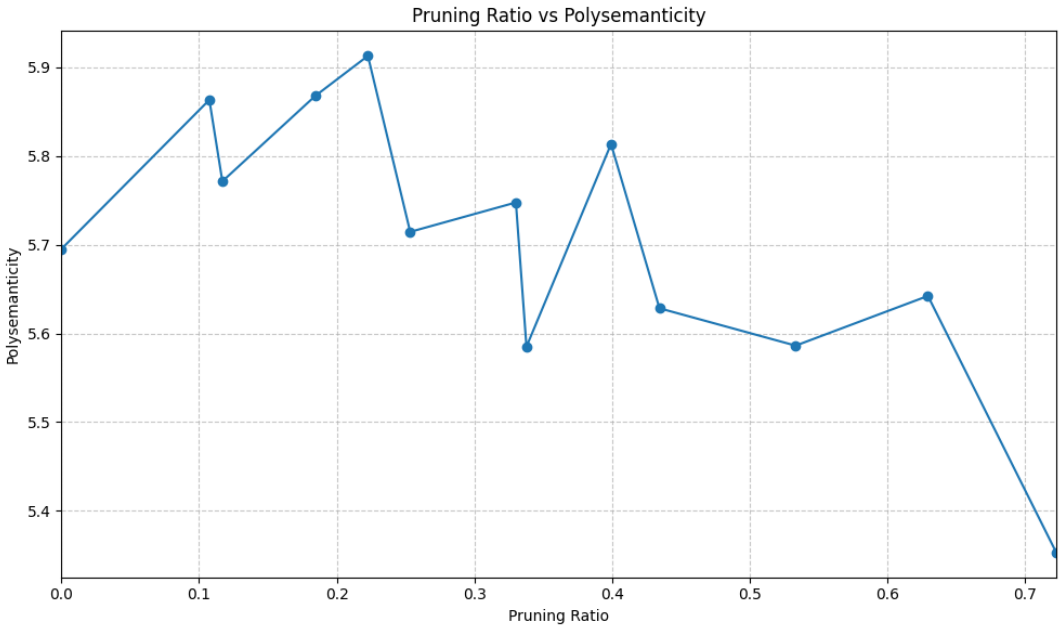
Figure 13: Nonlinear Experiment 2 | Polysemanticity during Training with and without Dropout

changes in polysemanticity, with more pronounced rates of change compared to models without Dropout. Additionally, models with and without Dropout tended to follow similar trends of polysemanticity during training and after pruning, both increasing or decreasing in polysemanticity.

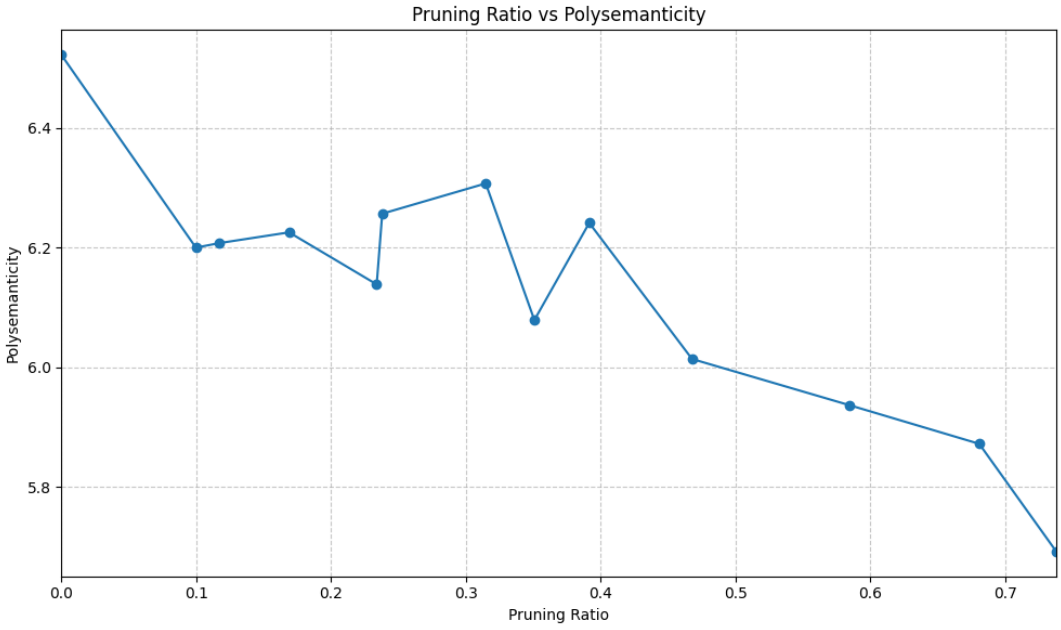
When examining polysemanticity after pruning and retraining, the layers of neural networks appeared to exhibit uniform patterns of change. As a result, polysemanticity could be evaluated globally during the experiments instead of taking the individual layers into account. In addition to that, total feature dimensionality did not seem to play a significant role in evaluating polysemanticity after pruning.

This brings us to the central question of whether pruning affects polysemanticity in neural networks or not. While no definitive trend could be seen in most cases, pruning led to a reduction of polysemanticity in the cases for which a clear trend could be concluded. This can be seen in Figures 10b, 14 and 16. These findings suggest that pruning may reduce polysemanticity under certain conditions, though the results remain inconclusive. The experiments were all conducted in a controlled environment, leaving the impact of pruning on polysemanticity in more realistic scenarios, such as one involving the MNIST dataset of handwritten digits [Den12], for future research.

An intriguing approach proposed by Sharkey et al. [SBM22] suggests that ground truth features of a problem can be extracted using a sparse autoencoder applied to the activations of a neural network. Here, sparse autoencoders were used to encode



(a) Model without Dropout



(b) Model with Dropout

Figure 14: Nonlinear Experiment 2 | Polysemanticity after Pruning and Retraining

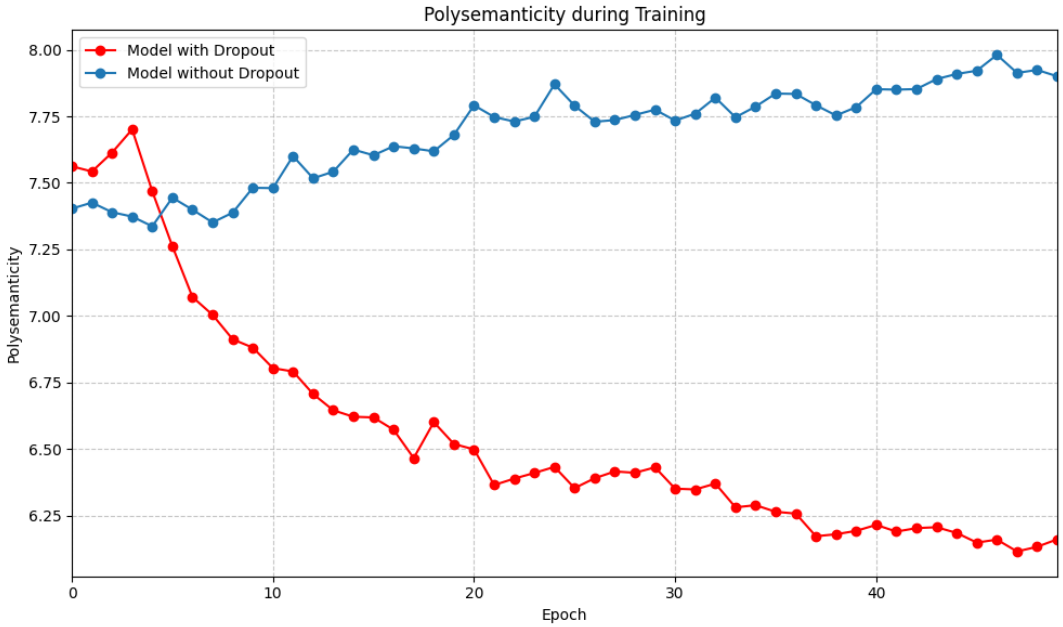


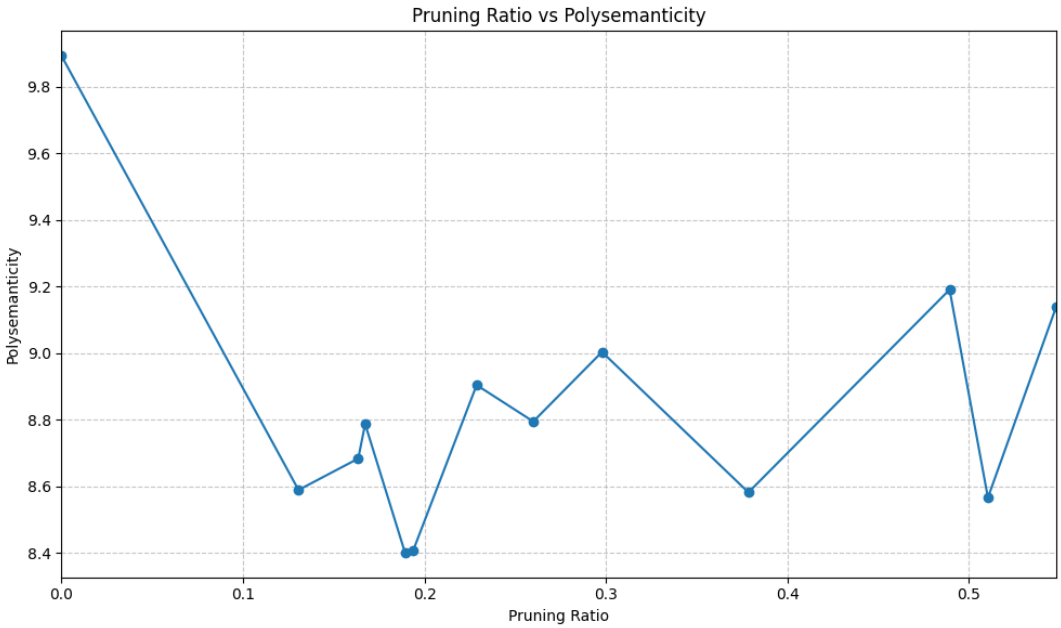
Figure 15: Nonlinear Experiment 3 | Polysemanticity during Training with and without Dropout

and decode the neuron activations of a model. The learned weights of the decoder then correspond to the features that should be extracted. This process is described in more detail by Sharkey et al. [SBM22]. If this method enables identification of features present in an input sample, it could provide a means to analyze the influence of specific features on individual neurons, offering a way to measure polysemanticity in scenarios without knowledge of the underlying ground truth.

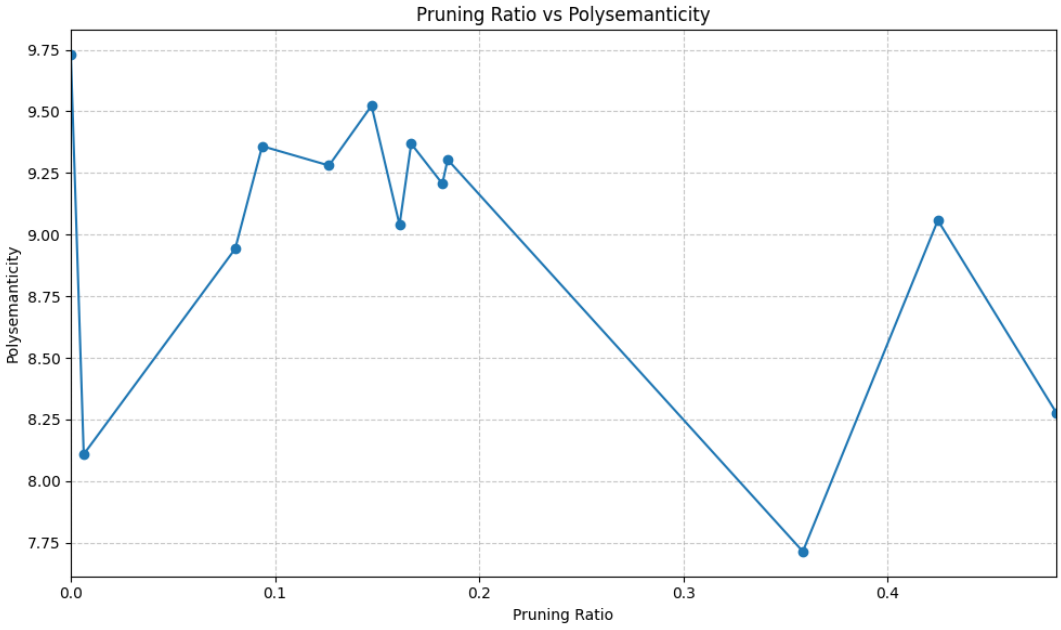
Section 2.2.2 explored the potential benefits of polysemanticity to highlight its relevance for neural networks and the need to evaluate factors that influence it. However, the experimental setup within this thesis did not align with scenarios where polysemanticity is thought to be advantageous, leaving this aspect for future studies. That is why corresponding metrics to evaluate advantages of polysemanticity were left out in the experiments of this thesis.

6. Conclusion

Polysemanticity refers to a neuron's activation being affected by multiple unrelated features. This thesis aimed to examine how pruning, meaning the removal of parameters from a model, impacts polysemanticity in neural networks. To create controlled experimental scenarios and enable precise measurements of polysemanticity, a method for generating synthetic toy data with known ground truth features was developed. The conducted experiments can be divided into linear and nonlinear



(a) Model without Dropout



(b) Model with Dropout

Figure 16: Nonlinear Experiment 4 | Polysemanticity after Pruning and Retraining

scenarios, depending on the linearity of the models' activation functions.

Each experiment begins with a large model designed to sufficiently represent the ground truth features. Various pruning ratios are then applied to reduce the model's size, followed by retraining to adapt the pruned network to its reduced structure. In this process, entire neurons are pruned based on the mean magnitude of their input weights. Neurons with the lowest mean magnitudes across the entire network are removed first. Polysemanticity is then evaluated throughout training and after the pruning process to observe any changes. Additionally, the Dropout technique was examined for its role in the context of pruning and polysemanticity, as it encourages redundancy in feature representations by enabling neurons to adapt to the absence of others.

Key findings from the experiments include the observation that the training process adjusts polysemanticity until it converges at a certain level, often showing a decrease during training. Dropout appeared to have minimal effect on the direction of how polysemanticity changes, with both the model with and without Dropout tending to change polysemanticity in the same direction either decreasing or increasing.

The core research question of whether pruning consistently affects polysemanticity could not be definitively answered. However, there was a tendency for polysemanticity to decrease in some cases after pruning. This challenges the initial expectation that polysemanticity would increase due to fewer neurons being available to represent the same number of features. A reduction in polysemanticity might occur because neurons with less relevant feature representations are pruned, leaving out the need for other neurons to compensate by representing additional features. Furthermore pruning can help prevent overfitting, as described in Section 2.3. As a result, the models may be able to shift their focus more on representing only the most relevant features, leading to reduced polysemanticity.

The evaluation of factors influencing polysemanticity is relevant due to its role in neural network interpretability [SSJ+23] and its connection to better cognitive performance, as outlined in Section 2.2.2. For this, pruning could play a central role, as the removal of neurons also removes the corresponding feature representations, while retraining can allow for an adjustment in polysemanticity to compensate for this loss.

The work of Sharkey et al. [SBM22], as already discussed in Section 5, is particularly interesting for the evaluation of polysemanticity in scenarios without knowledge of the ground truth features. They propose a method for extracting these features from neuron activations, which could enable measuring polysemanticity by analyzing a neuron's activation patterns for specific features. However, implementing this method would greatly increase the complexity of experiments, requiring a vast number of samples to assess feature impacts on neural activations. Thus, evaluating polysemanticity in uncontrolled environments without predefined ground truth features remains a task for future research.

Lastly, the observed trend of decreasing polysemanticity during training warrants

further investigation to determine whether it is a generalizable phenomenon. Expanding on these experiments could help clarify the nature of polysemanticity and its changes during training in neural networks.

References

[Bis06] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, 2006.

[BKG21] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, 2021.

[BOFG20] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[Den12] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[EHO+22] Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, Roger Grosse, Sam McCandlish, Jared Kaplan, Dario Amodei, Martin Wattenberg, and Christopher Olah. Toy models of superposition. *Transformer Circuits Thread*, 2022.

[FC19] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[FMR16] Stefano Fusi, Earl K Miller, and Mattia Rigotti. Why neurons mix: high dimensionality for higher cognition. *Current Opinion in Neurobiology*, 37:66–74, 2016. Neurobiology of cognitive behavior.

[FMS+23] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16091–16101, 2023.

[Hay98] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, USA, 2nd edition, 1998.

[HS92] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1992.

[HSK+12] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[JSH22] Adam S. Jermyn, Nicholas Schiefer, and Evan Hubinger. Engineering monosemanticity in toy models, 2022.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KMM+20] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi Yan, and Orion Reblitz-Richardson. Captum: A unified and generic model interpretability library for pytorch, 2020.

[LDS89] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989.

[LM20a] Matthew L. Leavitt and Ari Morcos. Selectivity considered harmful: evaluating the causal impact of class selectivity in dnns, 2020.

[LM20b] Matthew L. Leavitt and Ari S. Morcos. On the relationship between class selectivity, dimensionality, and robustness, 2020.

[MK24] Simon C Marshall and Jan H Kirchner. Understanding polysemanticity in neural networks through coding theory. *arXiv preprint arXiv:2401.17975*, 2024.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[Rud17] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[SBM22] Lee Sharkey, Dan Braun, and Beren Millidge. [interim research report] taking features out of superposition with sparse autoencoders, 2022.

[SSJ+23] Adam Scherlis, Kshitij Sachan, Adam S. Jermyn, Joe Benton, and Buck Shlegeris. Polysemanticity and capacity in neural networks, 2023.

[STY17] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks, 2017.

[ZTLT21] Yu Zhang, Peter Tino, Ales Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(5):726–742, October 2021.

Appendices

Appendix A Linear Experiment 1

The parameter configurations for this experiment are illustrated in Figure 6. While most results have already been presented in Section 4.1, Figure 17 includes the polysemanticity during training in the model with Dropout, where a decrease in polysemanticity can again be observed. Additionally, Figure 18 shows the progression of polysemanticity after pruning in relation to total feature dimensionality, as described in Section 3.3.3. In this figure, the polysemanticity values are multiplied by the total feature dimensionality, but this adjustment does not appear to add further meaning to the trend in polysemanticity, as can also be seen in the results of the other experiments.

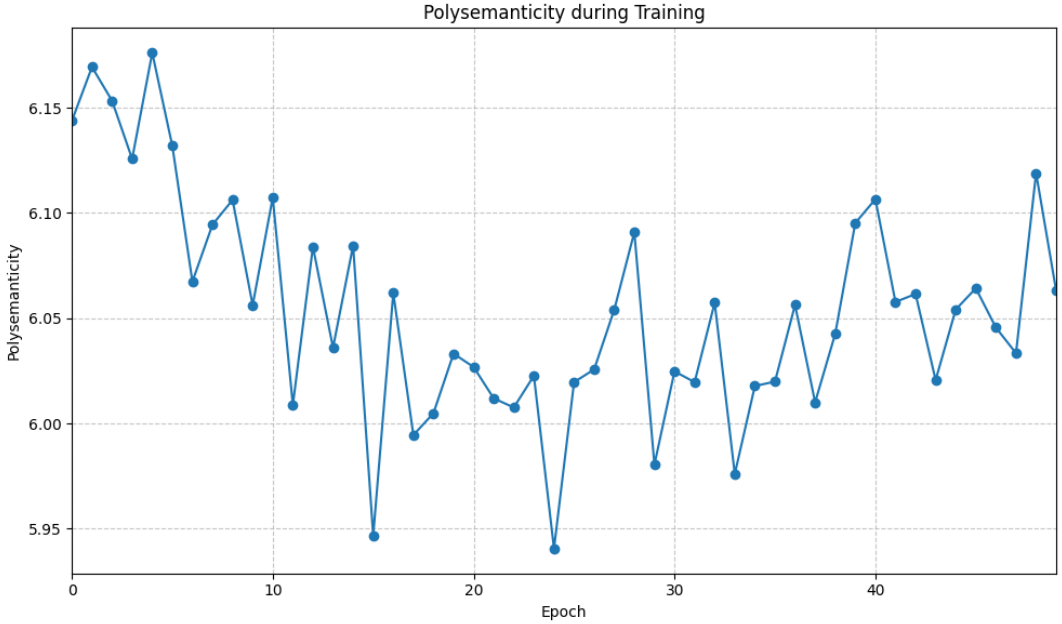


Figure 17: Linear Experiment 1 | Polysemanticity during Training with Dropout

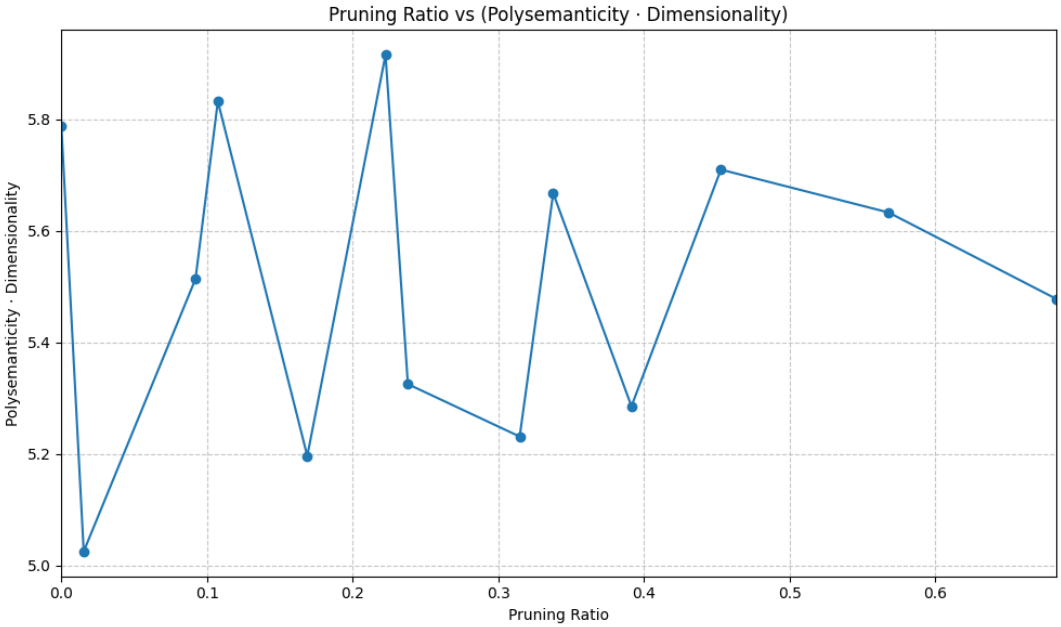


Figure 18: Linear Experiment 1 | Polysemanticity after Pruning and Retraining in the Context of Total Feature Dimensionality

Appendix B Linear Experiment 2

The parameter configurations of this experiment are defined in detail in Figure 19.

Parameter	Configuration
$linear$	$True$
$num\_features$	64
$hidden\_layers$	$[32, 24, 8]$
$input\_dim$	10
$output\_dim$	2
$sparsity$	$\{s_i = random(0.8, 0.9) \mid 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	6400
num_epochs	50
$observed\_model$	$[10, 256, 64, 64, 2]$

Figure 19: Linear Experiment 2 (Parameter Configurations)

The resulting baseline model of this experiment converges to a MSE loss of around 0.15 and represents the features sufficiently with a total feature dimensionality of

0.99.

Similar to the first experiment, there first seems to be a downward trend of polysemanticity during the training process, as shown in Figure 20. In contrast, polysemanticity increases slightly during the final half of the training epochs.

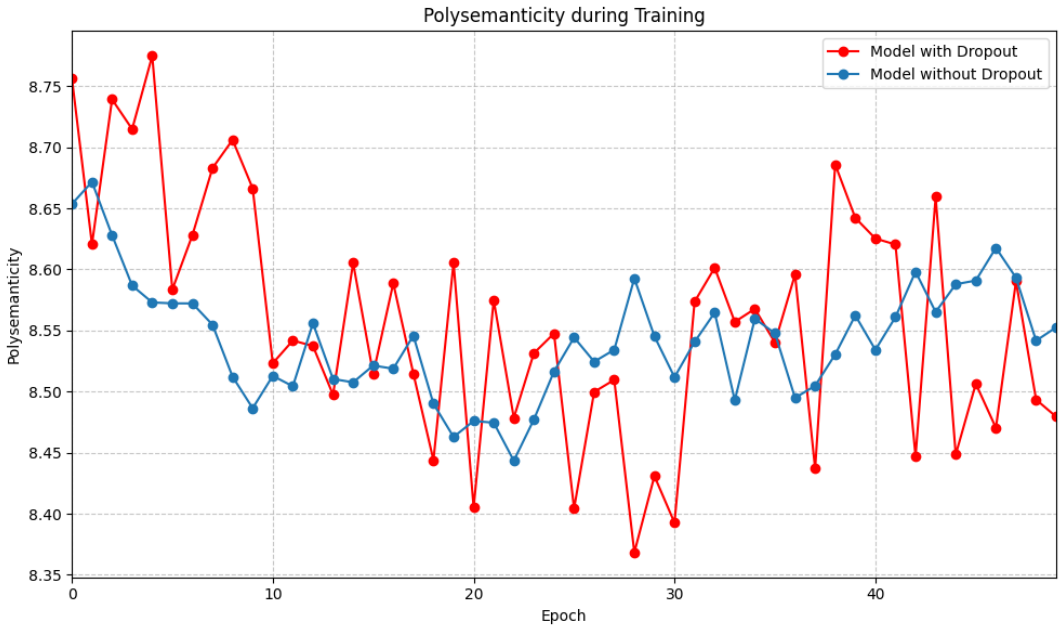


Figure 20: Linear Experiment 2 | Polysemanticity during Training

Figure 21 shows the layer-wise polysemanticity for different pruning ratios for this experiment. As can also be seen in the first experiment, the layers seem to develop in a similar way.

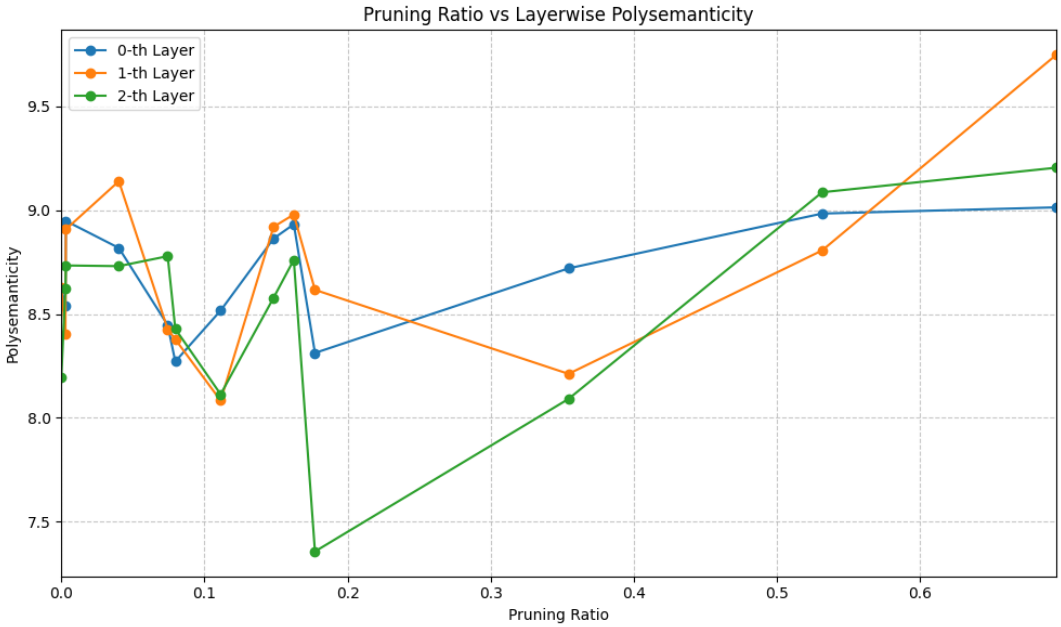


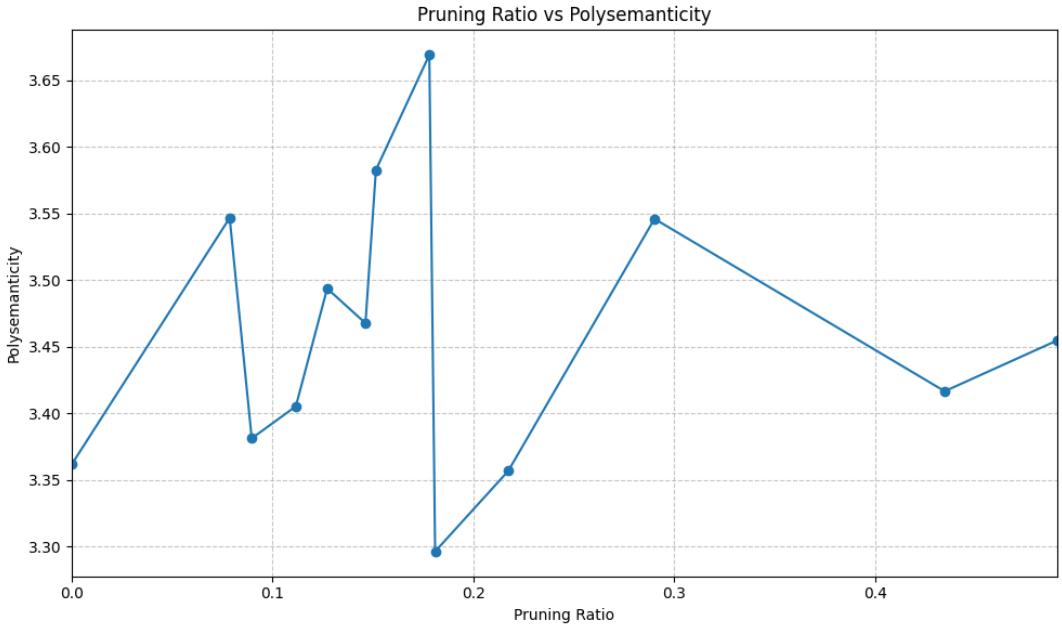
Figure 21: Linear Experiment 2 | Layer-wise Polysemanticity for different Pruning Ratios

Appendix C Nonlinear Experiment 1

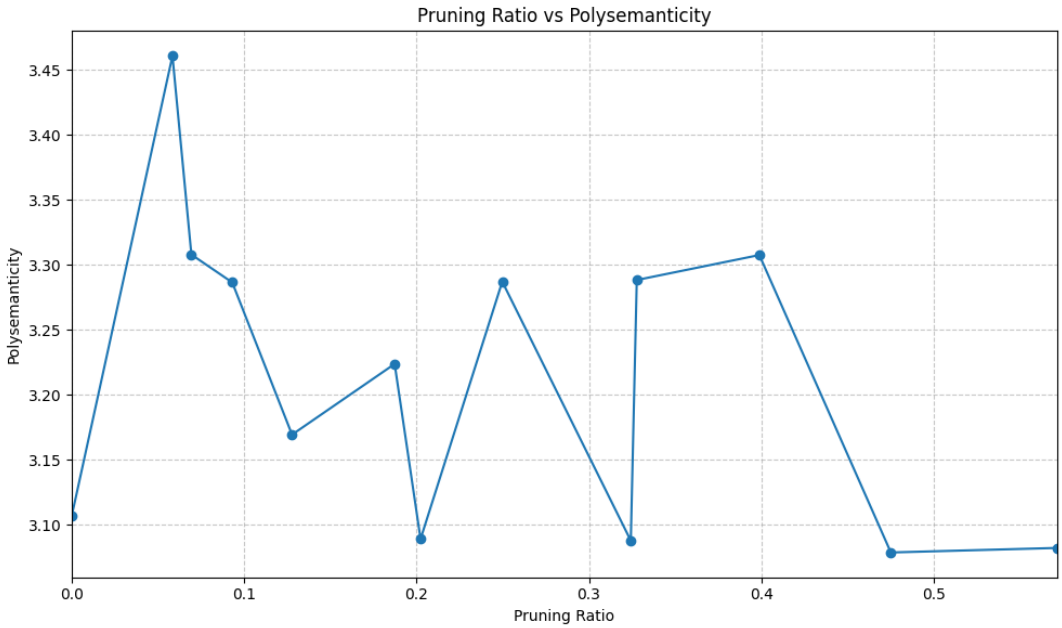
The parameter configurations of this experiment are already described in Section 4.2.

Figure 22 shows the degree of polysemanticity after pruning and retraining in the model with and without Dropout. It can be seen that polysemanticity does not change significantly for the highest pruning ratios. There are, however, fluctuations of polysemanticity which tend to be higher than the initial value of polysemanticity.

Additionally, Figure 23 illustrates polysemanticity after pruning multiplied by the total feature dimensionality. The trends of polysemanticity for the models with and without Dropout appear similar to those observed without accounting for total feature dimensionality. This further indicates that incorporating total feature dimensionality does not significantly impact the trends.

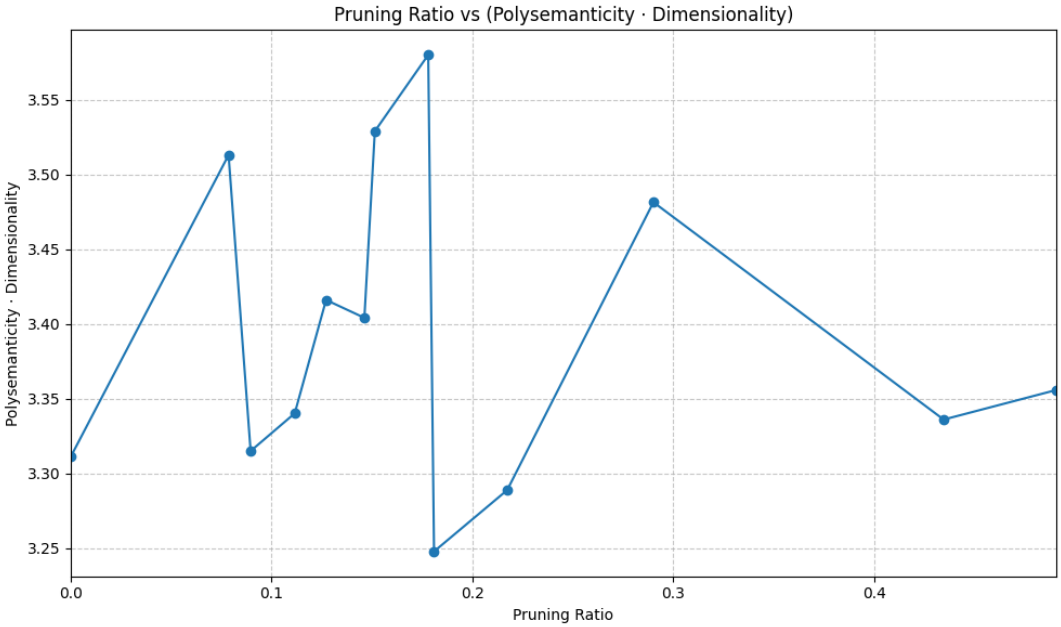


(a) Model without Dropout

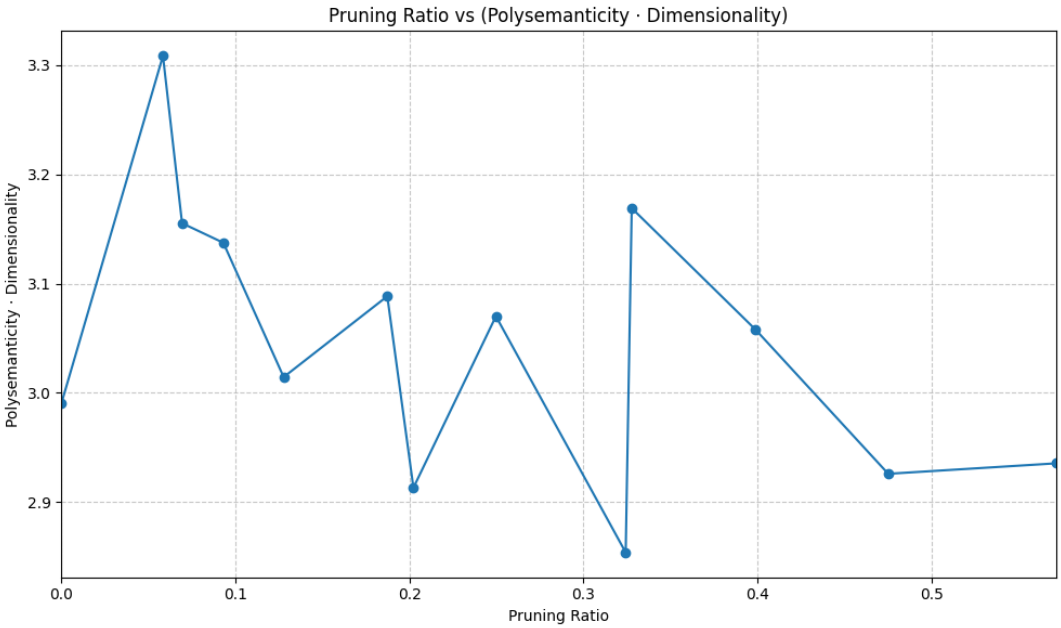


(b) Model with Dropout

Figure 22: Nonlinear Experiment 1 | Polysemanticity after Pruning and Retraining



(a) Model without Dropout



(b) Model with Dropout

Figure 23: Nonlinear Experiment 1 | Polysemanticity after Pruning and Retraining in the Context of Total Feature Dimensionality

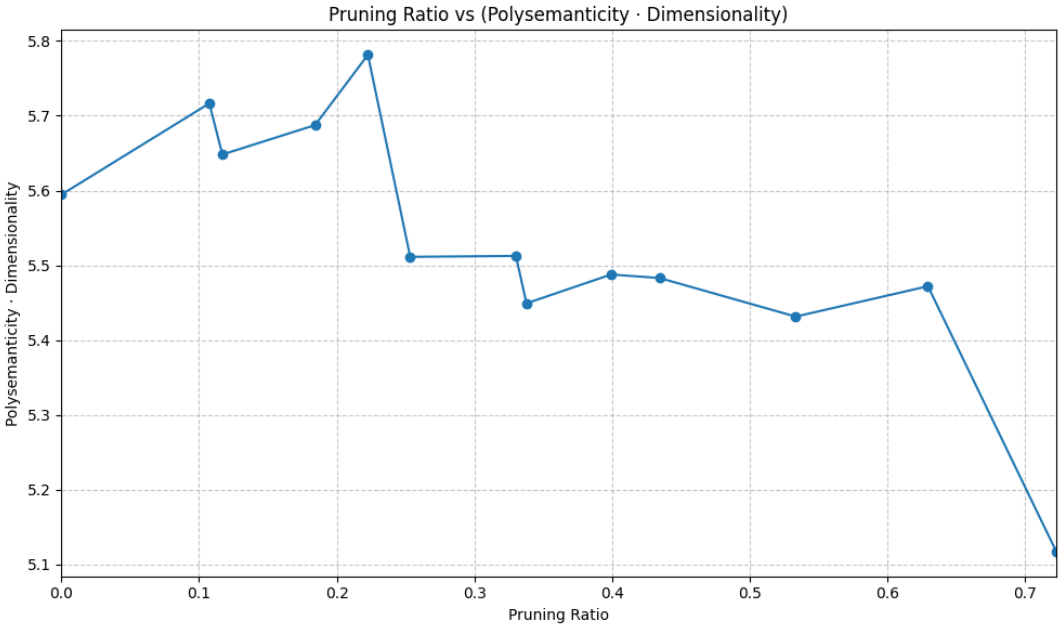
Appendix D Nonlinear Experiment 2

Figure 24 outlines the parameter configurations of this experiment. The baseline model achieves a MSE loss of 0.448 with a total feature dimensionality of 0.98.

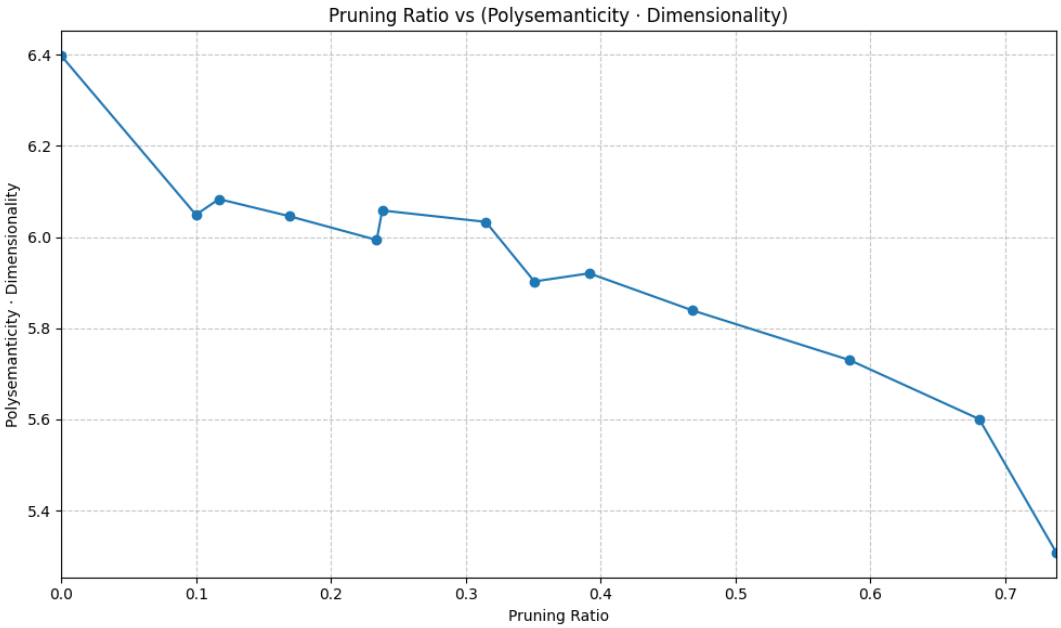
Parameter	Configuration
$linear$	$False$
$num\_features$	32
$hidden\_layers$	[32]
$input\_dim$	10
$output\_dim$	2
$sparsity$	$\{s_i = random(0.6, 0.8) \vert 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	3200
num_epochs	50
$observed\_model$	$[10, 128, 32, 2]$

Figure 24: Nonlinear Experiment 2 (Parameter Configurations)

Figure 25 shows polysemanticity after pruning scaled by total feature dimensionality. In comparison to Figure 14, this suggests again that total feature dimensionality adds no significant meaning for the development of polysemanticity.



(a) Model without Dropout



(b) Model with Dropout

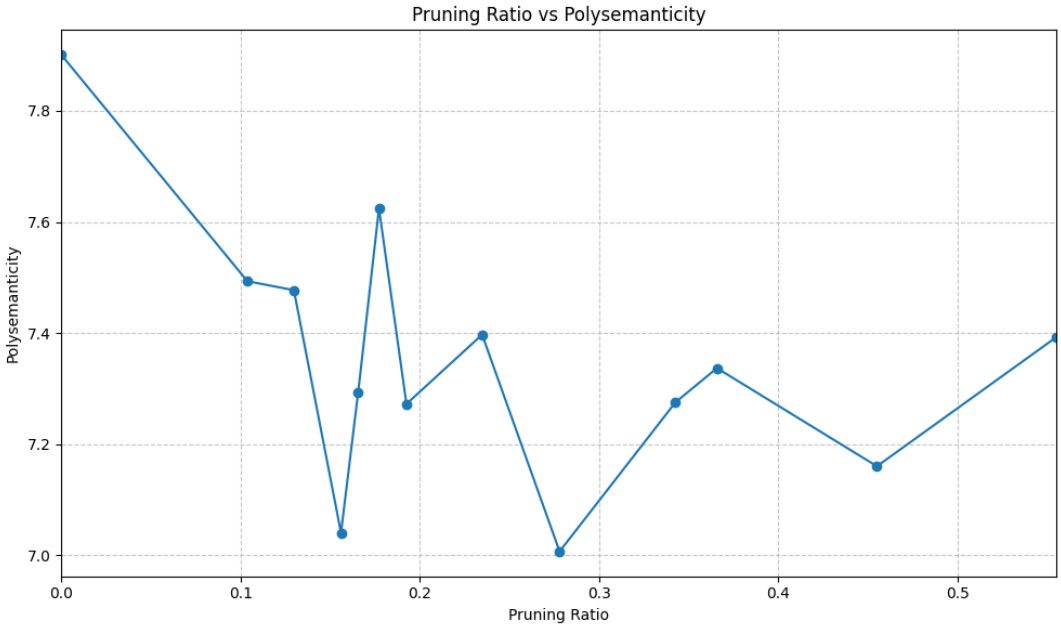
Figure 25: Nonlinear Experiment 2 | Polysemanticity after Pruning and Retraining in the Context of Total Feature Dimensionality

Appendix E Nonlinear Experiment 3

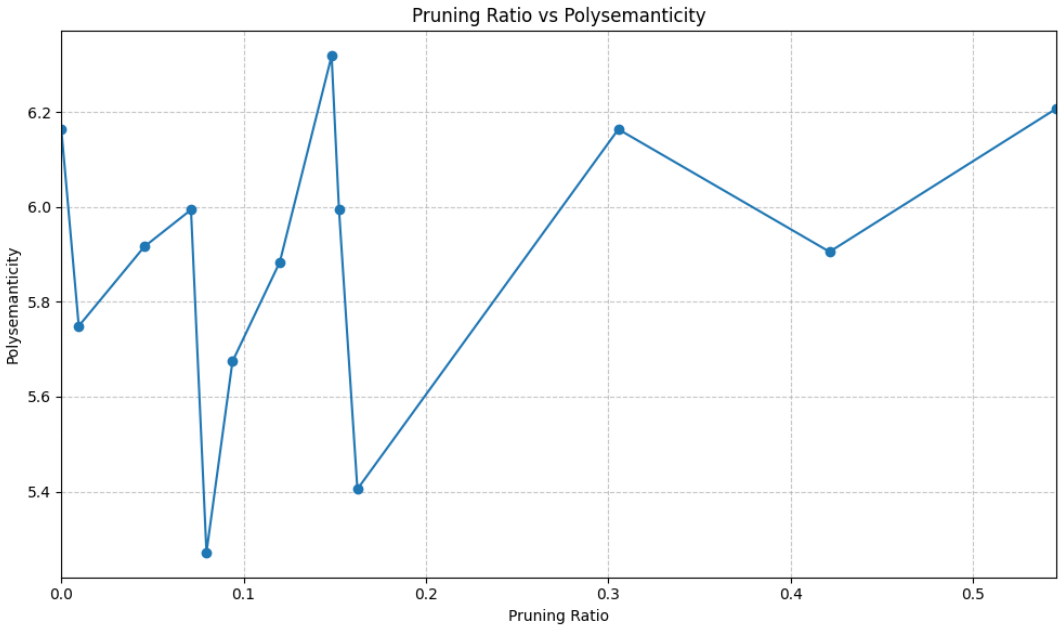
The parameter configurations described in Figure 26 result in a baseline model with a MSE loss of 0.0949 and a total feature dimensionality of 0.98. The development of polysemanticity in the models with and without Dropout is shown in Figure 27, where the model without Dropout exhibits a reduction in polysemanticity for higher pruning ratios. The corresponding trends in the context of total feature dimensionality are illustrated in Figure 28.

Parameter	Configuration
$linear$	$False$
$num\_features$	64
$hidden\_layers$	$[32, 24, 8]$
$input\_dim$	10
$output\_dim$	2
$sparsity$	$\{s_i = random(0.8, 0.9) \mid 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	6400
num_epochs	50
$observed\_model$	$[10, 256, 64, 64, 2]$

Figure 26: Nonlinear Experiment 3 (Parameter Configurations)

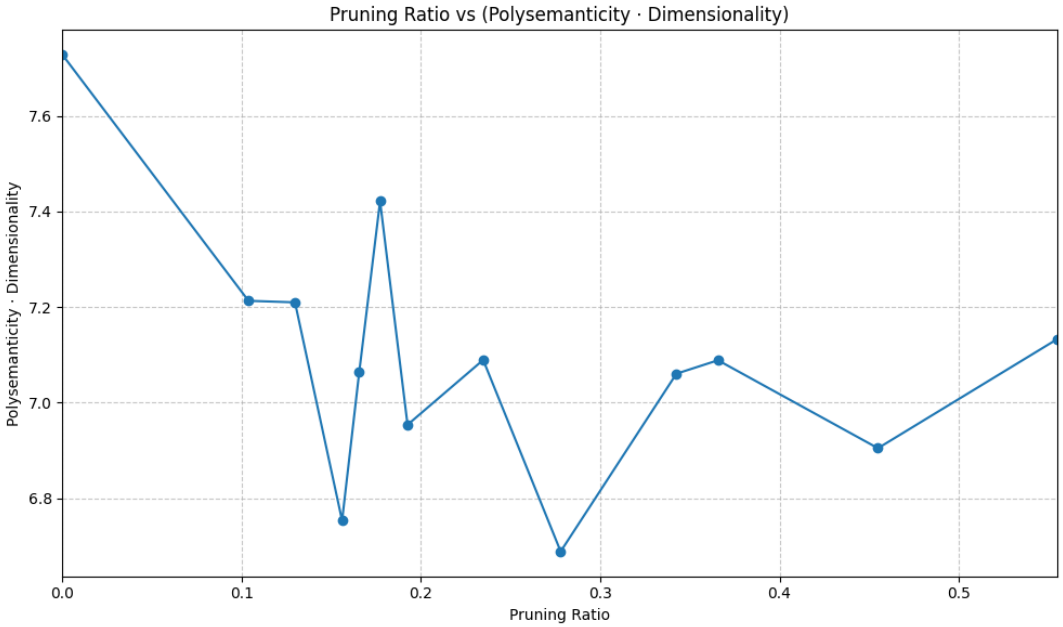


(a) Model without Dropout

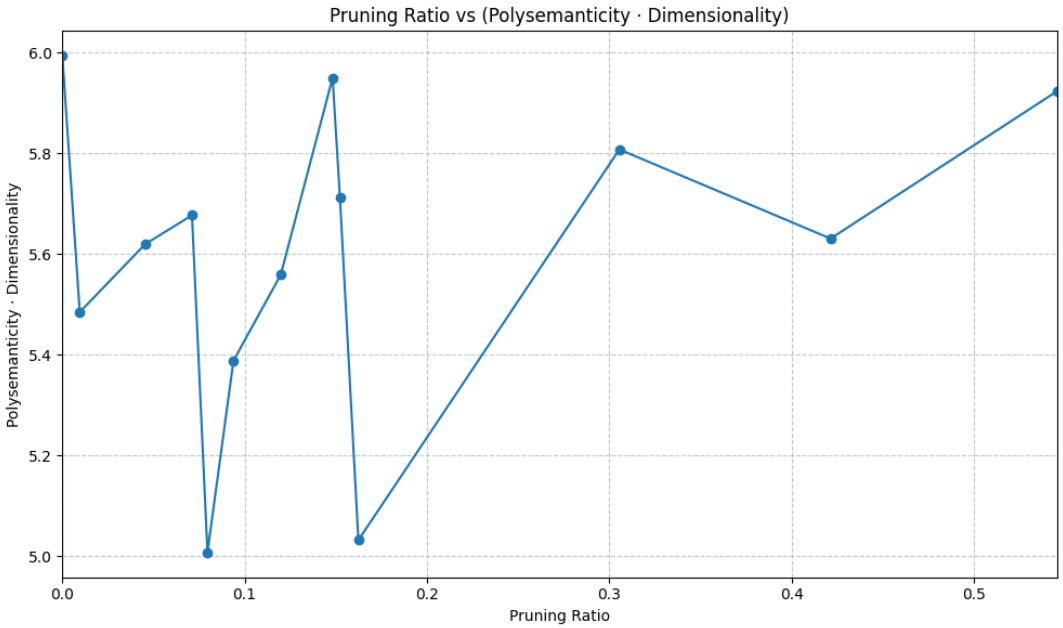


(b) Model with Dropout

Figure 27: Nonlinear Experiment 3 | Polysemanticity after Pruning and Retraining



(a) Model without Dropout



(b) Model with Dropout

Figure 28: Nonlinear Experiment 3 | Polysemanticity after Pruning and Retraining in the Context of Total Feature Dimensionality

Appendix F Nonlinear Experiment 4

The parameters for the final nonlinear experiment are detailed in Figure 29. The baseline model achieves an MSE loss of 0.5408 and a total feature dimensionality of 0.98. Figure 30 illustrates the polysemanticity during training for the model without Dropout, while Figure 31 shows the same for the model with Dropout. Here, polysemanticity seems to increase in the first half of training and decrease afterward for both models. Finally, Figure 32 presents the development of polysemanticity after pruning scaled by total feature dimensionality.

Parameter	Configuration
$linear$	$False$
$num\_features$	128
$hidden\_layers$	$[64, 48, 16]$
$input\_dim$	10
$output\_dim$	2
$sparsity$	$\{s_i = random(0.85, 0.95) \mid 1 \leq i \leq num\_features\}$
P	$random(-1, 1)$
ρ	0.05
$num\_samples$	12800
num_epochs	50
$observed\_model$	$[10, 512, 128, 128, 2]$

Figure 29: Nonlinear Experiment 4 (Parameter Configurations)

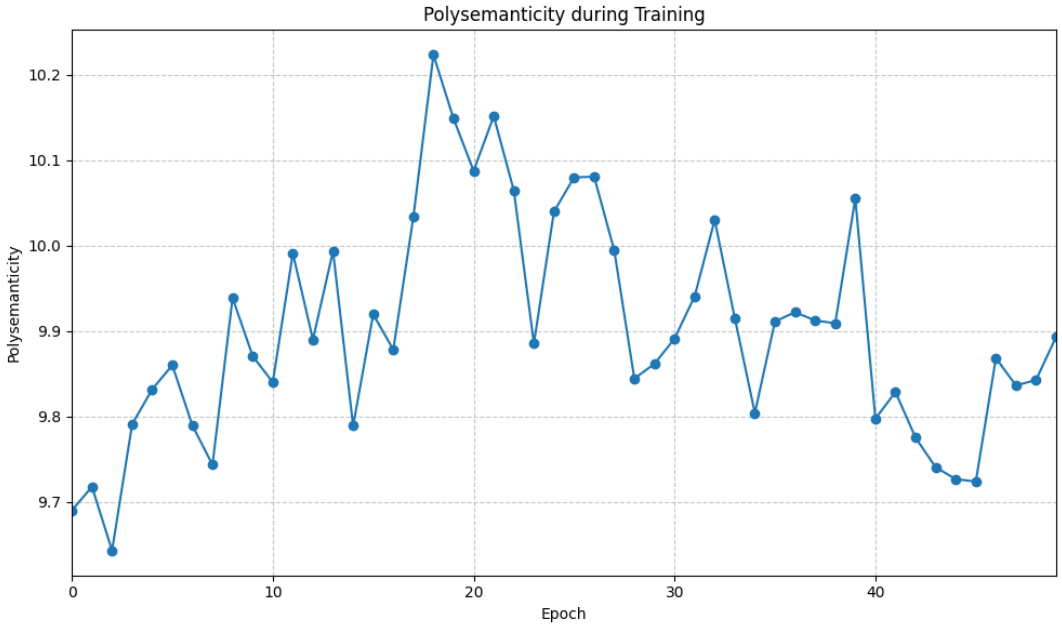


Figure 30: Nonlinear Experiment 4 | Polysemanticity during Training without Dropout

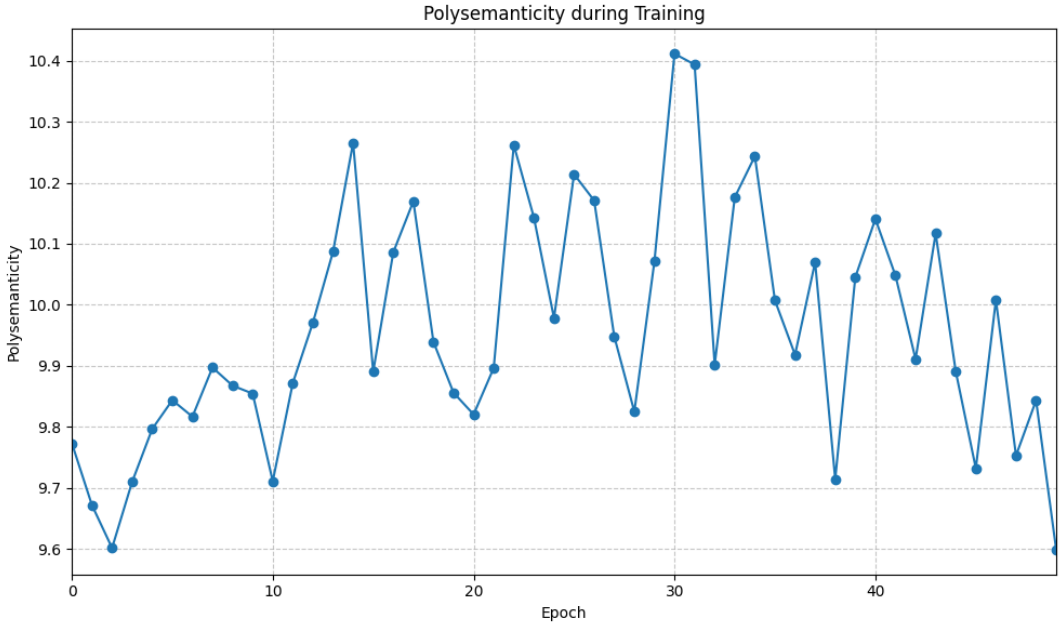
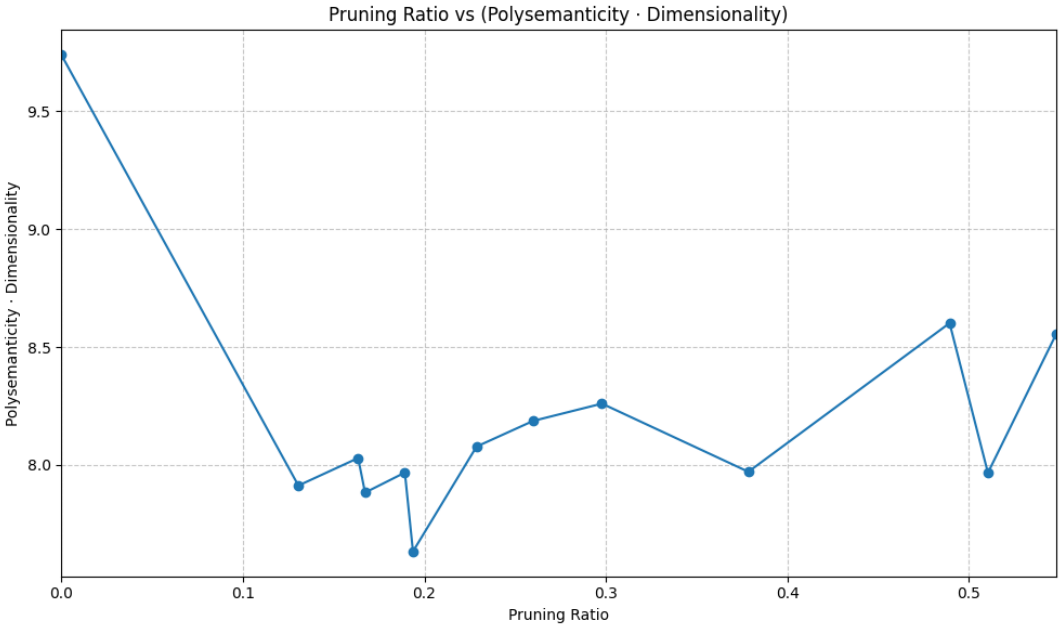
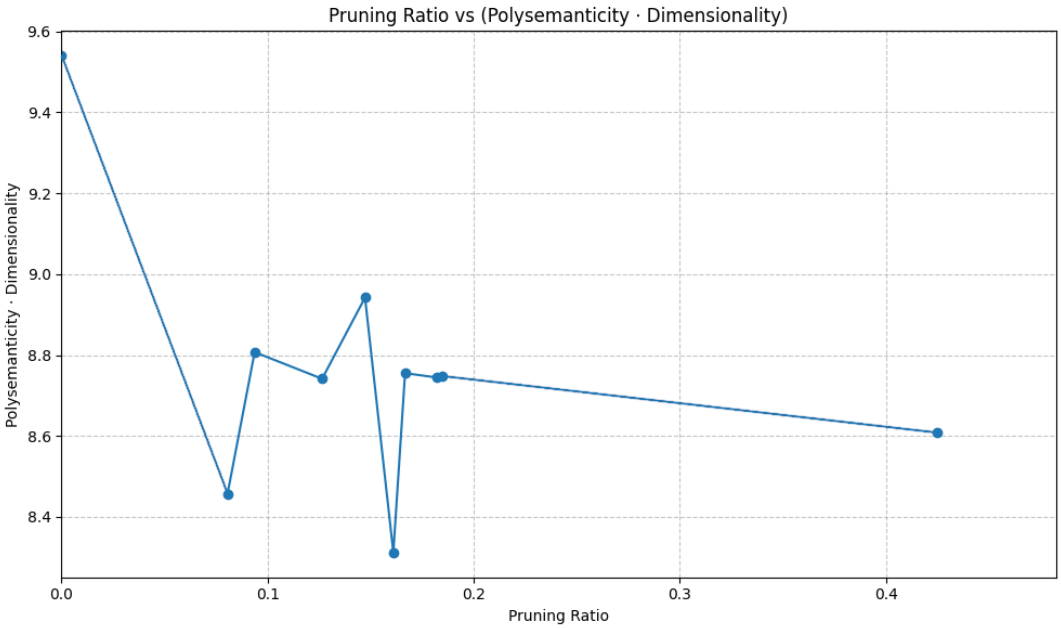


Figure 31: Nonlinear Experiment 4 | Polysemanticity during Training with Dropout



(a) Model without Dropout



(b) Model with Dropout

Figure 32: Nonlinear Experiment 4 | Polysemanticity after Pruning and Retraining in the Context of Total Feature Dimensionality