Examining Theoretical Sparsity and Data Topology's Influence on Neural Network Architectures with Insights from Lasso

Jo Dang

Paper References:

- "Equivalences Between Sparse Models and Neural Networks" by Ryan J. Tibshirani.
- "Topology of Deep Neural Networks" by University of Chicago and Technion – Israel Institute of Technology.

Summary

This report explores the mathematical equivalence between the Lasso regression model and a two-layer neural network with specific configurations, focusing on the role of sparsity. By comparing both frameworks, we demonstrate how a neural network can replicate Lasso's regularization effects. In Lasso, the ℓ_1-norm penalty promotes sparsity by setting some weights to zero, facilitating feature selection and reducing overfitting. Similarly, neural networks achieve sparsity through regularization techniques like weight pruning and dropout, which help focus the model on essential features while minimizing computational load. These insights apply across various loss functions and fully connected networks with ReLU activations. The report emphasizes how integrating sparsity into neural network design leads to more interpretable, generalizable models that effectively capture important data patterns. Architectural considerations and data topology provide further context to refine the relationship between model structure and performance.

Key Findings:

- Lasso (Least Absolute Shrinkage and Selection Operator) is equivalent to a 2-layer network fit with a ridge penalty (sum of squares) placed on all parameters, linear activations, no bias term, and a very simple connectivity structure.
- A k-layer network is equivalent to an l_p-penalized ($|\beta|^p$) regression where $p = 2/k < 1$.
- These equivalences can also be extended to any loss functions, not just squared loss.
- Weight decay is sparsity inducing.

Introduction

In recent years, the field of machine learning has witnessed rapid advancements in the design and application of neural networks. As these models become increasingly complex, understanding their architecture and behavior is crucial for improving performance and interpretability. This project investigates the relationship between neural network structures and data topology, aiming to uncover how various architectural designs can influence model efficiency and generalization capabilities.

The central research questions guiding this investigation are:

- *How does sparsity represented in Lasso mathematically equivocate to sparsity in a neural network?*
- *What are the implications of sparsity in neural networks for model interpretability and generalization?*
- *How do different neural network architectures correlate with the underlying topology of the data they process, and what role does sparsity play in this relationship?*

A significant focus is placed on the concept of sparsity in both Lasso regression and neural networks, emphasizing its importance for model interpretability and generalization. In **Lasso regression**, the objective is to minimize the error while adding an ℓ_1-norm penalty that encourages some of the weights to be zero. The objective function can be represented as:

$$\min_{\beta} \sum_{i=1}^{n} \left(y_i - x_i^T \beta\right)^2 + \lambda \sum_{j=1}^{d} |\beta_j|$$

Lasso sets some coefficients β_j exactly to zero, effectively excluding the corresponding features x_j from the model and creating sparsity. This is useful for identifying only the most relevant features, allowing us to focus on a smaller, more meaningful subset of data. For instance, in a dataset with hundreds of features, Lasso may select just a few key predictors, clarifying which variables have the most significant impact. This sparsity also helps prevent overfitting, especially when the number of features is large relative to the number of observations, ultimately enhancing the model's generalizability to unseen data.

In this report, "equivalence" refers to the mathematical alignment between Lasso regression and neural networks with a ridge penalty (sum of squares) placed on all parameters, using linear activations, no bias term, and a simple connectivity structure. A k-layer network can be viewed as an l_p-penalized regression, where $p = 2/k < 1$. These mathematical equivalences suggest that similar principles of sparsity govern both regression and neural network architectures.

Introducing sparsity in neural networks involves setting some weights across different layers to 0, achieved through techniques such as regularization with penalties and adding an ℓ_1-penalty (or a similar sparsity-inducing penalty) to the objective function. Structured sparsity through depth helps networks to focus on significant weights, transforming dense, fully connected networks into sparse networks where only essential connections remain.

The benefits of sparsity in neural networks include:

- **Efficient Computation**: Sparse networks require fewer computations, significantly reducing memory and computational costs, especially for large models.
- **Improved Generalization**: Sparsity helps mitigate overfitting by reducing model complexity, which is particularly useful in neural networks with a high capacity for memorization.

Data Section

Boston Housing Dataset

The dataset used in this analysis is the Boston Housing dataset, which comprises housing information for various towns and suburbs in the Boston area. This dataset contains **506 rows** and **13 features**, with a target variable representing the median value of owner-occupied homes (MEDV) in thousands of dollars. The goal of our machine learning model is to predict house prices based on the provided features.

Variables Overview

1. **CRIM**: Per capita crime rate by town.
2. **ZN**: Proportion of residential land zoned for lots over 25,000 sq. ft.
3. **INDUS**: Proportion of non-retail business acres per town.
4. **CHAS**: Charles River dummy variable (1 if tract bounds river; 0 otherwise).
5. **NOX**: Nitric oxides concentration (parts per 10 million).
6. **RM**: Average number of rooms per dwelling.
7. **AGE**: Proportion of owner-occupied units built prior to 1940.
8. **DIS**: Weighted distances to five Boston employment centers.
9. **RAD**: Index of accessibility to radial highways.
10. **TAX**: Full-value property-tax rate per $10,000.
11. **PTRATIO**: Pupil-teacher ratio by town.
12. **B**: Calculated as $1000(Bk - 0.63)^2$, where Bk is the proportion of black residents by town.
13. **LSTAT**: Percentage of lower status of the population.
14. **MEDV**: Median value of owner-occupied homes in $1000's (target variable).

Because our focus in this report is not data analysis, no preliminary data exploration was included in this final draft. The data wrangling steps involved were:

- Normalizing and standardizing features for both Lasso regression and training neural networks.
- Splitting the dataset into training and testing sets to evaluate model performance.

Methods

1. Lasso Regression and 2-layer Neural Networks

$$\hat{y} = W_2\sigma(W_1X)$$

where

- W_1 and W_2: matrices of neural network weights that represent the strength of connections between inputs, hidden neurons, and the output.
    - W_1 determines how input features (X) influence the activations of the first layer, which are transformed by the activation function σ (e.g., ReLU).
    - W_2 then combines these activations to produce the final output ($\hat{y}$), with all weights being adjusted during training to minimize a loss function and capture complex data relationships.
- σ: activation function

In the context of neural networks, sparsity is achieved when:

- W1 involves regularizing/adding penalties to weights so that the model relies on fewer neuron in the hidden layers
- W2 considers only a subset of the "activated" neurons so that not all neurons participate in the calculation of the output

Standard Lasso

With $y_i \in \mathbb{R},\ i = 1, 2, \ldots, n$ responses, $x_i \in \mathbb{R}^d,\ i = 1, 2, \ldots, n$ feature vectors, and a $\lambda \geq 0$ tuning parameter. We choose the values of β_j by minimizing:

$$\min_{\beta} \left(\sum_{i=1}^{n} \left(y_i - x_i^T\beta\right)^2 + 2\lambda \sum_{j=1}^{d} |\beta_j| \right)$$

where:

- $\left(y_i - x_i^T\beta\right)^2$: represents the least squares error, which quantifies the difference between the predicted values and the actual responses. Minimizing this term ensures that the predictions closely approximate the observed data.

- $\sum_{j=1}^{d} |\beta_j|$: the Lasso penalty, which imposes an ℓ_1-norm constraint on the coefficients β_j, promoting sparsity by encouraging some coefficients to shrink to exactly zero.
- The parameter λ determines the trade-off between minimizing the prediction error and enforcing sparsity. Larger values of λ increase the Lasso penalty $\sum_{j=1}^{d} |\beta_j|$, which encourages sparsity by setting more coefficients β_j to zero. However, λ and the β_j values are not random variables. Instead, λ directly controls the extent to which sparsity is applied.

In this section, we begin by considering a key observation that will lead us to a reformulation of the Lasso objective function, which we will later show to be equivalent to a two-layer neural network architecture. This connection between Lasso and neural networks forms the core of our analysis.

Observation:

$$a^2 + b^2 - 2ab = (a-b)^2 \geq 0 \implies (a^2 + b^2) \geq 2ab. \implies \min(a^2 + b^2) = 2ab$$

where $a = b$

Let $u, v \in \mathbb{R}^2$ such that $\beta_j = u_j v_j$.

$$2\sum_{j=1}^{d} |\beta_j| = 2\sum_{j=1}^{d} |u_j v_j| \leq \sum_{j=1}^{d} (u_j^2 + v_j^2)$$

With this, you can rewrite the Standard Lasso expression as:

$$\min_{u,v} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{d} x_{ij} u_j v_j \right)^2 + \lambda \sum_{j=1}^{d} (u_j^2 + v_j^2)$$

This is a *simply-connected network* (each input neuron is directly connected to each neuron in the hidden layer, with no additional complex connections or branching).

- When u_j and v_j are small, the product $\beta_j = u_j v_j$ will also be small, effectively driving β_j toward zero.
- The ℓ_2 penalty term $\sum_{j=1}^{d} (u_j^2 + v_j^2)$ ensures that both u_j and v_j remain as small as possible, thus minimizing β_j for each feature.

In this way, minimizing the ℓ_2 penalty $(u_j^2 + v_j^2)$ directly influences the sparsity of β_j because it limits the size of each β_j term. This is conceptually similar to the ℓ_1 penalty in Lasso regression, where large values of β_j are discouraged.

What's the point of factoring β_j into $u_j v_j$?

The math so far has been to convince you that a simple 2 layer-network is mathematically equivalent to the standard LASSO technique.

Furthermore, the interaction between regularization and the product parametrization (in the loss function, where W become $W1.W2$) create new critical points, or minima specifically, creating nonconvexity. This actually creates the ability to force some coefficients to be 0 instead of only minimizing them like in the case the standard Lasso. This creates sparsity. You can also take this expression as a more detailed version of the expression we previously saw involving W_1 and W_2

Consider: constrain $v \geq 0$, and take the example that variance terms must always be nonzero. This is fine because the signs can always be absorbed into the components of u. Now, we have

$$v_j = \sqrt{\alpha_j}, \quad u_j = \frac{\beta_j}{\sqrt{\alpha_j}}, \quad j = 1, \ldots, d$$

α_j, a new parameter, is a non-negative scalar associated with each β_j in the model:

- $\frac{\beta_j^2}{\alpha_j}$ ensures that the β_j coefficients are penalized by a factor inversely proportional to α_j.
- The term α_j in the regularization encourages sparsity in the optimization process, similar to the Lasso penalty.

In essence, α_j controls the balance between the penalty on β_j and the overall sparsity (thanks to the $\frac{\beta_j}{\sqrt{\alpha_j}}$), allowing for better control over the regularization strength.

Consequentially, we have:

$$\min_{\alpha \geq 0, \beta} \quad \sum_{i=1}^{n} (y_i - x_i^T \beta)^2 + \lambda \sum_{j=1}^{d} \left(\frac{\beta_j^2}{\alpha_j} + \alpha_j \right)$$

When we look at this expression, sparsity may become more apparent. Take into account that:

- Because $\alpha_j = (v_j)^2$, when α_j is minimized, $(v_j)^2$ is exponentially lower.
- The first derivative smoothes the gradient, helping optimization converge more effectively.

Some care needs to be taken to handle division by zero: we interpret the j-th summand to be 0 if $\alpha_j = \beta_j = 0$, and ∞ if $\alpha_j = 0$ but $\beta_j \neq 0$.

The function $g(x, y)$, as defined in the following Lemma, is a way of balancing two objectives: promoting sparsity in the model while allowing the network to maintain flexibility. The idea is that we want a regularization function that penalizes the model parameters in a way that encourages

sparsity (i.e., some parameters should be forced to zero) while still enabling the model to capture meaningful patterns in the data.

The function $g(x, y)$ is designed to achieve this balance by combining two components:

- **The $\frac{y^2}{x}$ term**: This term encourages sparsity by penalizing larger values of x when y is nonzero. As x gets smaller, this term has less of an effect, leading to fewer nonzero parameters in the model (sparsity).
- **The x term**: This term introduces flexibility by allowing the model to fit the data more effectively when x is larger, scaling directly with the magnitude of x.

Together, these terms strike a balance, and the design of $g(x, y)$ ensures that the sparsity is maintained at the minimum of the function.

The function $g(x, y)$ is carefully constructed to handle various cases that might arise during optimization:

- If both x and y are 0, the function is set to 0, as this corresponds to an optimal solution where no regularization is needed.
- If $x = 0$ but $y \neq 0$, the function returns infinity, reflecting an invalid or infeasible solution that would result in division by zero. This prevents the model from selecting parameters that could lead to undefined or unstable solutions.
- Otherwise, the function returns a value that combines both the sparsity-inducing and flexibility-promoting terms.
    - *Achieves sparsity at the minimum*: The form of $g(x, y)$ was chosen to ensure that at the function's minimum, the resulting values promote sparsity, meaning many values of x will be zero for appropriate choices of y.

Then, consider the following simple lemma.

Lemma 1. The function $g : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ defined by

$$g(x, y) = \begin{cases} \frac{1}{2}\left(\frac{y^2}{x} + x\right) & \text{if } x \neq 0 \\ 0 & \text{if } x = y = 0 \\ \infty & \text{if } x = 0, y \neq 0 \end{cases}$$

Suppose we have both $x \neq 0$, $y \neq 0$, and $x = |y|$. Then,

$$g(x, y) = \frac{1}{2}\left(\frac{y^2}{|y|} + |y|\right) = \frac{1}{2}(|y| + |y|) = |y|$$

Then, with the minimum achieved at $x = |y|$,

$$\min_{x \geq 0} g(x, y) = |y|$$

The last 2 cases of the lemme accounts for the aforementioned division by 0. The first case comes from the reparametrization of v_j and u_j.

If we apply this lemma to the expression with reparametrization, we get the initial expression of the standard lasso.

Consider, then:

$$g(\alpha, \beta) = \frac{1}{2}\left(\frac{\beta^2}{\alpha} + \alpha\right)$$

We know that the minimum of this is $|\beta|$. Then,

$$\min_{\alpha \geq 0, \beta} \quad \sum_{i=1}^{n}(y_i - x_i^T\beta)^2 + \lambda \sum_{j=1}^{d}\left(\frac{\beta_j^2}{\alpha_j} + \alpha_j\right) = \min_{\beta}\left(\sum_{i=1}^{n}\left(y_i - x_i^T\beta\right)^2 + 2\lambda\sum_{j=1}^{d}|\beta_j|\right)$$

which is the Standard Lasso expression. So, the reparametrization of β, which introduces higher dimensionality and further mathematically illustrates sparsity in the case of a simple neural networks, also shows the correlation between the two ideas.

Matrix Factorization

The concept of matrix factorization, particularly in the context of sparsity, is built on the work of Srebro et al. (2004). For any data matrix $Y \in \mathbb{R}^{n \times d}$, where n is the number of rows and d the number of columns, we aim to find a low-rank approximation by minimizing a general loss function L and incorporating regularization. This can be formulated as:

$$\min_{M} L(Y, M) + 2\lambda\|M\|_*$$

where:

- $\|\cdot\|_F$ is the Frobenius norm, defined as the square root of the sum of the absolute squares of a matrix's elements. Mathematically, for a matrix A, it's given by $\|A\|_F = \sqrt{\sum_{i,j}|a_{ij}|^2}$.
- $\|\cdot\|_*$ represents the trace norm, which is the sum of the singular values of the matrix M. In diagonal matrices, this norm is equivalent to the ℓ_1 norm of the diagonal elements.

Here, Y represents the data matrix we are analyzing, and M is the low-rank approximation of Y that minimizes the loss L with a trace norm penalty. This formulation encourages sparsity in the solution, meaning fewer significant singular values or connections are needed to represent the data.

This formulation is equivalent to:

$$\min_{U,V} L(Y, UV^T) + \lambda(\|U\|_F^2 + \|V\|_F^2)$$

where U and V are matrices that, when multiplied, approximate M. In this setup:

- For diagonal matrices, the trace norm $\|M\|_*$ corresponds to the ℓ_1 norm of the diagonal elements, while the Frobenius norm $\|M\|_F$ corresponds to the ℓ_2 norm of the diagonal elements.

Through this equivalence, we observe that regardless of the loss function L, the regularization framework still encourages sparsity. Specifically, we can express the optimization problem in a form that highlights this sparsity as follows:

$$\min_{u,v} \sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{d} x_{ij} u_j v_j\right) + \lambda \sum_{j=1}^{d} (u_j^2 + v_j^2)$$

This is mathematically equivalent to:

$$\min_{\beta} \sum_{i=1}^{n} L\left(y_i, x_i^T \beta\right) + 2\lambda \sum_{j=1}^{d} |\beta_j|$$

This formulation underscores that, regardless of the chosen loss function L, the mathematics consistently demonstrate the presence of sparsity in the optimization process. This framework allows us to draw parallels between matrix factorization and the sparsity-inducing mechanisms seen in models such as Lasso regression and neural networks.

---

2. k-layer Neural Network Case

Let weights across layers be denoted as $w_j^{(\ell)},\ \ell = 1, \ldots, k$.

We aim to minimize the following expression:

$$\min_{w^{(\ell)},\, \ell=1,\ldots,k} \sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{d} x_{ij} \prod_{\ell=1}^{k} w_j^{(\ell)}\right) + \lambda \sum_{j=1}^{d} \sum_{\ell=1}^{k} \left(w_j^{(\ell)}\right)^2$$

Weights Across Layers: $w_j^{(\ell)}$ are the weights for feature j in layer ℓ. The term $\prod_{\ell=1}^{k} w_j^{(\ell)}$ captures the cumulative effect of all layers on feature j.

Regularization: The term $\lambda \sum_{j=1}^{d} \sum_{\ell=1}^{k} \left(w_j^{(\ell)}\right)^2$ is a summation over all weights across all layers, penalizing their magnitudes.

Using the AM-GM Inequality for Sparsity

To facilitate our sparsity approach, we can employ the AM-GM inequality, which states:

For any non-negative real numbers $a_1, a_2, \ldots, a_k$,

$$\frac{1}{k}\sum_{\ell=1}^{k} a_\ell \geq \left(\prod_{\ell=1}^{k} a_\ell\right)^{\frac{1}{k}}$$

This inequality tells us that the arithmetic mean of these terms is always at least as large as their geometric mean. Equality holds if all a_ℓ are the same.

In our case, we are not just dealing with a_ℓ but with a_ℓ^2, so the inequality applies to their squares:

$$\frac{1}{k}\sum_{\ell=1}^{k} a_\ell^2 \geq \left(\prod_{\ell=1}^{k} a_\ell^2\right)^{\frac{1}{k}}$$

This simplifies to:

$$\sum_{\ell=1}^{k} a_\ell^2 \geq k\left(\prod_{\ell=1}^{k} a_\ell\right)^{2/k}$$

Explanations:

- Each a_ℓ can be thought of as $|w_j^{(\ell)}|$.
- The AM-GM inequality then yields $\left(\prod_{\ell=1}^{k} a_\ell\right)^{\frac{2}{k}}$, representing the geometric mean of squared terms across the k layers.
- The inequality indicates that if all weights $|w_j^{(\ell)}|$ are equal, the average of their squares is minimized.
- When we sum the weights' contributions across layers, the AM-GM inequality helps us rewrite the regularization term, emphasizing that to minimize the average weight's magnitude, we should keep them similar.

Why This Matters for Our Weights In our neural network context, each a_ℓ corresponds to the magnitude $|w_j^{(\ell)}|$ of a weight for a specific feature j in layer ℓ. Applying the AM-GM inequality to the squares of these weights helps us rewrite the regularization term by combining the influence of weights across layers.

Encouraging Uniformity in Weights The AM-GM inequality is "tightest" (i.e., it reaches equality) when all a_ℓ are equal, meaning the weights $|w_j^{(\ell)}|$ across layers would ideally be the

same for a given feature j. This uniformity across layers simplifies the model and leads to a more straightforward sparsity structure in the network.

$\prod_{\ell=1}^{k} w_j^{(\ell)}$ represents the product of the weights $w_j^{(\ell)}$ across k layers for a specific feature j in a neural network. This product is often used to express how the feature contributes to the final output of the network through all layers.

The output for a single feature j after k layers can be expressed as a cumulative product of the weights across those layers. If we denote the weights connecting the input feature x_j to the k layers as $w_j^{(1)}, w_j^{(2)}, \ldots, w_j^{(k)}$, the contribution of the feature j to the final prediction is expressed as:

$$\text{Output} = \prod_{\ell=1}^{k} w_j^{(\ell)} \cdot x_j.$$

β represents the combined effect of all weights on the input features. It encapsulates the contributions from all layers.

The product $\prod_{\ell=1}^{k} w_j^{(\ell)}$ can be seen as a way of expressing β_j because both represent the cumulative transformation of the feature x_j through the weights from the input layer to the final output. Thus, we have:

$$\beta_j = \prod_{\ell=1}^{k} w_j^{(\ell)}.$$

This equivalence allows us to simplify the notation, making it easier to analyze the effects of weights across layers using β.

Rewriting the Objective with ℓ_p-Regularization

By applying the AM-GM inequality and combining weights across layers, we can express our objective function in terms of a single parameter β:

$$\min_{\beta} \sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{d} x_{ij}\beta_j\right) + k\lambda \sum_{j=1}^{d} |\beta_j|^{2/k}$$

Here:

- **Loss Function**: The term $\sum_{i=1}^{n} L\left(y_i, \sum_{j=1}^{d} x_{ij}\beta_j\right)$ measures how well the model predicts the outputs y_i based on the inputs x_{ij} weighted by β_j. This part remains consistent with previous formulations, capturing the prediction error.

- **Regularization Term**: The term $k\lambda \sum_{j=1}^{d} |\beta_j|^{2/k}$ arises from the application of the AM-GM inequality. Here, k represents the depth of the network, and λ is a regularization parameter. The exponent $\frac{2}{k}$ indicates that as the number of layers (depth) increases, the penalty on the weights changes.

Effects of p:

- **For small** k: The regularization behaves like ℓ_2-norm, which shrinks weights but allows them to remain non-zero. This encourages smaller weight values without pushing many to zero.
- **For larger** k: The regularization shifts towards an ℓ_0-norm-like behavior, which promotes sparsity by pushing many weights exactly to zero. This means the model will focus only on the most critical connections.

We notice that this is a more generalized version of the standard Lasso expression. This time, it has an $\ell_{2/k}$ penalty term.

The relationship between the depth of neural networks and the concept of sparsity can be summarized as follows:

- **Shallow Networks**: In shallow networks (small k), regularization resembles Ridge regression (ℓ_2 penalty). In this context, small weights are favored, but they are not necessarily pushed to zero. This allows for greater model flexibility, enabling the network to adapt to the data without overly restricting its capacity.
- **Deeper Networks**: As the depth k of the network increases, the regularization effect becomes more aggressive, promoting the zeroing out of weights. When $p < 1$ for larger k, we begin to observe a stronger selection of significant features, leading to a sparser model. This means that as we go deeper, the model tends to focus more on the most important features while discarding less relevant ones.

Understanding the Complexity of ℓ_p-Penalized Problems with $p < 1$

NP (Nondeterministic Polynomial time)-hardness describes problems that are extremely complex computationally. Specifically, NP-hard problems cannot be solved efficiently (in "polynomial time"), meaning the time needed to solve them grows very quickly as the problem size increases.

- **When $p < 1$, ℓ_p-norms are non-convex**, which complicates the search for a global optimum. This non-convexity makes finding an optimal solution to these penalized problems **NP-hard**.

- For instance, even training a **simple neural network** with linear activations and multiple layers ($k > 2$) becomes NP-hard to optimize globally. This theoretical difficulty increases as the complexity or depth of the model grows.

Practical Implications for Deep Learning

- Although finding a global optimum is computationally hard, **modern optimization tools** in deep learning (numpy) often reach **local optima** that work well in practice. These local solutions may not be perfect but are still **effective at scale**, making neural networks highly useful for complex, real-world problems.

    - **TensorFlow**: Developed by Google, TensorFlow provides tools for distributed computing, making it suitable for large datasets and extensive models. It includes optimizers like **Adam**, **RMSprop**, and **SGD** (Stochastic Gradient Descent) with momentum, which are crucial for efficiently navigating complex, non-convex landscapes.

    - **PyTorch**: Developed by Facebook's AI Research lab, it provides dynamic computation graphs, PyTorch includes a range of powerful optimizers, and has built-in support for GPU acceleration, enabling large-scale deep learning with relatively lower computational time.

    - **Keras**: Part of TensorFlow, it provides access to various optimization algorithms, loss functions, and regularization techniques, making it easier to experiment with different architectures and tune hyperparameters.

    - **JAX**: Developed by Google, JAX is designed for high-performance numerical computing and automatic differentiation. It can support high-performance, highly parallelized computations, including advanced optimization tasks.

    - **MXNet**: Backed by Apache, MXNet is optimized for high efficiency in both training and inference, especially for distributed computing across multiple GPUs and devices. It also supports a variety of languages.

    - **Fastai**: Built on top of PyTorch, Fastai provides pre-built models and easy-to-use components for tasks like image classification, NLP, and tabular data analysis Its emphasis on "progressive resizing," learning rate schedules, and data augmentation techniques helps achieve efficient training, even with limited computational resources.

    - **Horovod**: Horovod, initially developed by Uber, is not a deep learning library itself but rather a tool for distributing deep learning training across multiple GPUs or machines. Compatible with TensorFlow, PyTorch, and Keras, it's especially useful for large-scale model training in industry settings.

    - **Optimizers and Techniques Across Frameworks**: Many frameworks implement **AdamW**, **AdaGrad**, **AdaDelta**, **Nesterov Accelerated Gradient (NAG)**, and **L-BFGS**, which are advanced optimization methods. These tools are designed to handle complex, non-convex landscapes.
- Thus, while theoretical global solutions are difficult, **practical solutions** remain achievable, enabling neural networks to handle non-convex problems successfully in many applications.

Results

explain equivalence and edit the topology part. make conclusion about categorical variables

LASSO and a 2-layer NN

Applying ℓ_1 Regularization in a Two-Layer Neural Network

In a two-layer neural network, ℓ_1 regularization is applied by adding a penalty term to the loss function, which encourages the weights in the network to take on smaller or zero values, effectively promoting sparsity. In the previous section, we saw this in the mathematical from of a 2-layer network. Here's how it works in more detail:

The key part of how the ℓ_1 penalty term affects the gradients and weights lies in the **gradient calculation during backpropagation.**

To adjust the weights during training, we need to compute the **gradient** (derivative) of the loss function with respect to each weight. The gradient tells us how to adjust the weight in order to minimize the loss.

For the ℓ_1 penalty, the derivative of the absolute value term $|w_j|$ is **piecewise**:

$$\frac{\partial}{\partial w_j}|w_j| = \begin{cases} \lambda & \text{if } w_j > 0 \\ -\lambda & \text{if } w_j < 0 \\ 0 & \text{if } w_j = 0 \end{cases}$$

This is where the **sin function** comes in, as $\sin(w_j) = 1$ when $w_j > 0$, -1 when $w_j < 0$, and 0 when $w_j = 0$. So, the gradient of the ℓ_1 regularization term is:

$$\lambda \cdot \sin(w_j)$$

Effect of the Gradient on Weight Update

Now, this gradient is combined with the regular gradient of the main loss function (e.g., Mean Squared Error for regression or cross-entropy for classification). The total gradient for weight update becomes:

$$\text{Total Gradient} = \text{Gradient from Main Loss} + \lambda \cdot \sin(w_j)$$

The Effect on the Weights

- **Large weights ($|w_j| \gg 0$)**: Since the penalty term is proportional to the absolute value of the weights, it **shrinks** large weights over time, causing them to become smaller. This

process is often referred to as **weight shrinkage**. The ℓ_1 penalty contributes a relatively larger update (compared to smaller weights), pushing these weights towards 0.

- **Small weights ($|w_j| \ll 0$)**: Over time, weights that are less important to the model's performance will be **pushed to 0** by the ℓ_1 penalty. If a weight's update due to the loss function is smaller than the penalty, the weight will gradually decrease until it reaches 0. The ℓ_1 penalty contributes a smaller update, but still encourages them to shrink further towards zero.

- **Weights close to zero ($w_j \approx 0$)**: Eventually, some weights will be **exactly 0**, meaning that those connections will no longer contribute to the network's output. This is how **sparsity** is achieved—the network learns to ignore certain weights, simplifying the model.

Summary

- The ℓ_1 regularization term adds a gradient component that **pushes weights toward zero** by penalizing large values.

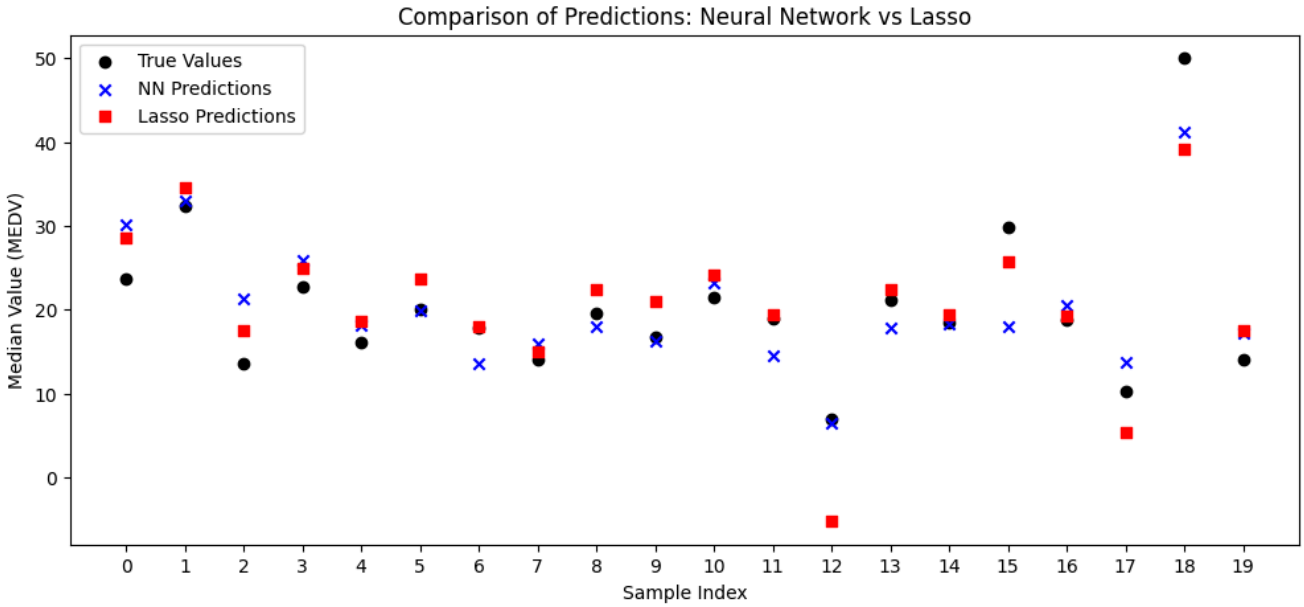
- This penalty term encourages the model to shrink unimportant weights, effectively setting them to 0 over time.

- The result is **sparsity** in the network, where only the most relevant weights remain non-zero.

Models' Results

The dataset is split into training and testing sets, and the features are standardized using StandardScaler

We can take a closer look at how our models specifically made predictions compared to each other and the real response value from our test set.




```
Neural Network Performance:
MSE: 24.7924, R²: 0.6619

Lasso Regression Performance:
MSE: 25.6567, R²: 0.6501
```

In these results, we're comparing the performance of two models—Neural Network and Lasso Regression—on the Boston Housing dataset, using two key metrics:

- MSE (Mean Squared Error): The squared differences between the predicted and actual housing prices, on average. Lower MSE values generally indicate better model performance.
- The R^2 value (or the "coefficient of determination") shows how well the model's predictions match the actual data. An R^2 of x means that approximately x of the variance in housing prices is explained by the possible relationships found by model.

The very similar results is consistent with the mathematical proof that we explored in the Methods section.

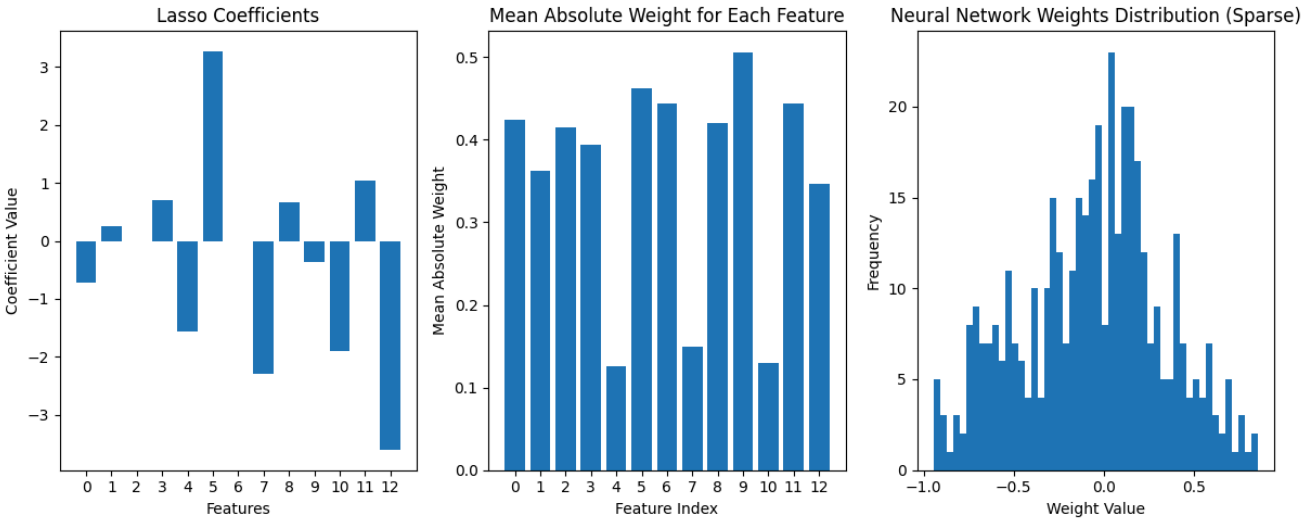
We can compare the coefficients learned by Lasso regression with the weights of the first layer of the 2-layer neural network. However, keep in mind that the neural network weights (or coefficients) are more complex to interpret compared to linear models like Lasso. In Lasso regression, the coefficients directly correspond to the features, while in neural networks, weights are distributed across layers and neurons, making direct comparison a bit more abstract.

- **In Lasso regression**:
  - **"Pattern Recognition and Machine Learning" by Christopher Bishop**: Bishop explains that sparsity arises when a model assigns zero values to certain parameters,

effectively selecting a subset of features and reducing model complexity. Sparse models capture only the most relevant features, improving interpretability and often aiding in generalization by minimizing overfitting.

- **In neural networks**:
  - Particularly in models with sparsity-promoting regularization (such as ℓ_1 or similar penalties), the absolute weights provides a rough idea of how much each feature is influencing the hidden layer's neurons. While the relationship is not as direct as in Lasso, larger absolute weights generally signify a stronger impact on the network's output, especially when sparsity constraints are applied.
  - Average Absolute Weight Calculation:
    - **Take Absolute Values**: Convert each weight in the layer's weight matrix to its absolute value to capture the strength of influence (ignoring direction).
    - **Average Across Neurons**: For each feature, average its absolute weights across all neurons in the layer. This gives a single value per feature, representing its average influence on the layer.

This comparison is not perfect, as the neural network's weights represent a more complex, distributed pattern of relationships, but it gives us a general sense of how feature importance looks like between the two models.



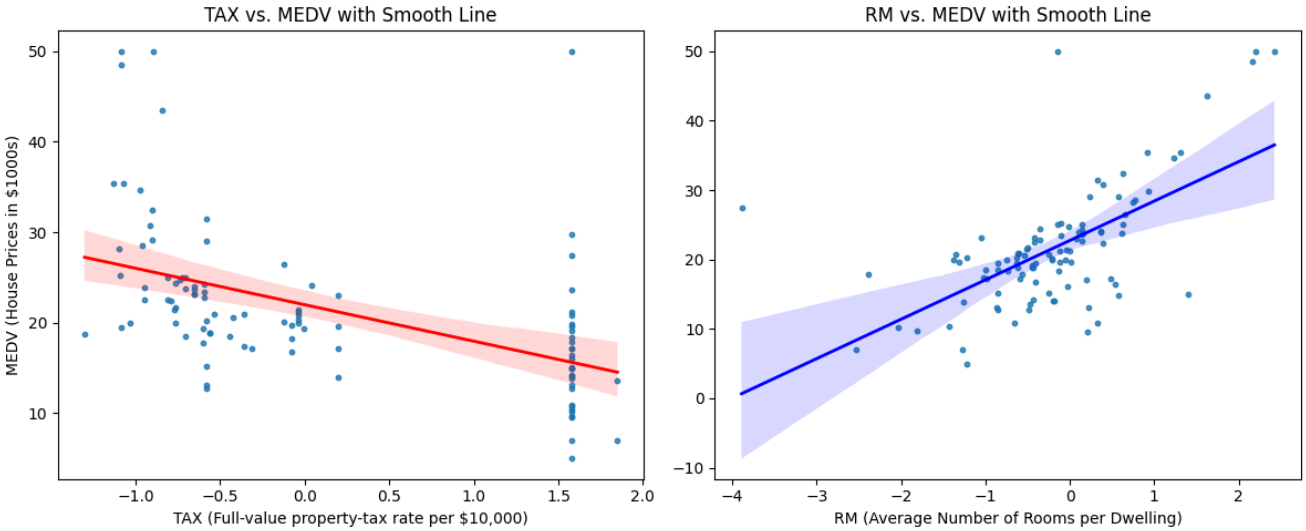
Despite the similar MSE and R^2 results, the weight distributions across variables differ between the two models. We can take a closer look into one of those apparent occurrences:

Feature 9, TAX: The full-value property-tax rate per $10,000 for a given town or suburb. In the limited exploratory data analysis scope of this report, we will not investigate how RAD specifically affects MEDV (the house prices) in this data set. However, we can easily look at a

scatterplot with a linear regression line to get a slight idea of whether there exists a linear relationship between these variables.

As a comparison, a similar graph will illustrate the relationship between feature 5 and MEDV. Feature 5 is **RM**, average number of rooms per dwelling. This seems like a variable that would have a fairly direct effect on MEDV. More importantly, both our Lasso model and neural network found quite similar weight magnitudes for this variable.

the different techniques respond differently to categorical variables



It seems that a somewhat linear relationship is more apparent in the graph with RM and MEDV, which correlates to similar weight distributions from both Lasso and our 2-layer neural network. In contrast, TAX, a variable whose effect on our response were measured differently by our models, does not seem to have a strongly linear relationship with MEDV. This does make sense, considering that tax brackets and policies from county to county are not quite linear (one might say, categorical).

The differences in how the Lasso regression model and the neural network treat variables can arise from several key factors:

Modeling Flexibility and Non-Linearity:

- **Lasso**: It assumes a linear relationship between the input variables and the target. This means that if a variable doesn't provide a strong linear relationship, its coefficient will be shrunk.
- **Neural Networks**: Even if a variable appears less significant in a linear context, the networks may keep the feature in the model and assign it importance, especially if the feature has non-linear predictive power.

Interaction Effects:

- **Lasso**: Lasso treats each feature independently. It doesn't model interactions between variables, unless specifically instructed to.
- **Neural Networks**: They can automatically learn interactions between features in their hidden layers.

Optimization Process:

- **Lasso**: Lasso minimizes the error while enforcing sparsity, which may lead to different trade-offs in terms of which variables are selected. The optimization process is constrained to find the most parsimonious model.
- **Neural Networks**: Neural networks rely on gradient-based optimization (such as stochastic gradient descent), which does not have the same sparsity constraints as Lasso. This means the network can adjust the importance of each feature in a more flexible way, depending on how they influence the network's predictions.

Overfitting vs. Underfitting:

- **Lasso**: If the regularization parameter (lambda) in Lasso is too large, it might shrink the coefficients too much, causing underfitting. Some variables might be neglected, even if they hold value in predicting.
- **Neural Networks**: Neural networks can sometimes overfit. The neural network might more importance to a variable that may not necessarily contribute as much to predictive power in a simpler model like Lasso.

Graph 3 is the visualization of the network's assignments of weights to each feature, in each neuron, of each layer. We can see that many weights were zero and near-zero, which we can interpret as sparsity in the context of a neural network.

Conclusion

In summary, **Lasso** works well when the relationship between the features and target is linear, or when only a few features are truly relevant. It simplifies the model and helps prevent overfitting. **Neural Networks** are more effective in modeling non-linear relationships and capturing complex feature interactions. While the neural network might treat certain variables as important (even if they seem less so in a linear model like Lasso), it can achieve higher accuracy when the data is complex. However, without regularization, neural networks may overfit the data, reducing their predictive power on unseen examples.

The best part of discovering that Lasso can be equivalent to a simple neural network is realizing that we can apply regularization to achieve sparsity—helping with overfitting and interpretability—while also capturing complex, non-linear relationships that Lasso alone might miss. Neural networks have the advantage of identifying these relationships automatically, without requiring

manual specification. By uncovering these relationships through neural networks, we may then highlight them when fitting a Lasso model to improve interpretability.

Neural Networks Architectures Comparison and Data Topology

Moving from examining individual variables in a Lasso framework, neural networks can leverage data topology—allowing them to capture not only the influence of distinct variables but also intricate patterns and dependencies within the data itself. This capacity to dynamically respond to data topology marks a key advantage in applying neural networks beyond traditional linear methods.

In the next section, we will be testing different network structures on our data set. Taking what we have learned so far, I added an ℓ_2 regularization to every layer of every network to encourage sparsity in our models. By assessing the performances of different network architectures other than the simple 2-layer or k-layer structures that we discussed previously, we may find how sparsity well works with these structures as well as test a couple of existing hypotheses regarding possible correlations between data topology and certain network designs.

Early stopping was applied to all models to ensure that the network does not overfit. By monitoring the validation loss during training, the network halts when it no longer improves. The max number of neurons applied to a layer is 128 to ensure a "fair competition" for the models.

Wide Network

The wide model consists of a single hidden layer with 128 neurons. It captures information in a broad, non-hierarchical way, processing all features simultaneously without gradually narrowing the data representation. For comparison, the simple 2-layer network that we used in comparison to Lasso also contains only 1 hidden layer (the other layer is for outputs), but it had only 32 neurons.

A wide network may be sufficient in learning the complex relationships in a topologically convoluted data set, but it might underperform when compared to other networks that have been researched to potentially do well (Naitzat, G., Zhitnikov, A., & Lim, L.-H.), which we will examine in a moment.

- **Expected performance**: Might balance between overfitting and underfitting but could still miss hierarchical patterns captured by deeper models.
- **Sparsity and Weight Decay**:
    - Due to the large number of neurons in the single layer, weight decay (L2 regularization) is critical. It penalizes large weights, forcing the model to focus on the most important relationships and prevent overfitting. This is only my hypothesis, which we can test by simply examining the performances of the model with and without L2 regularization.

Deep Network (4 Hidden Layers)

Model 1 is a simple neural network with decreasing neurons in each layer. The architecture starts with 128 neurons in the first layer and gradually reduces (in halves) to 16 neurons in the last hidden layer, followed by a single output neuron for regression.

Architecture:

- This is a relatively straightforward design, where the number of neurons decreases progressively. This type of architecture can learn complex patterns but might not capture deeper hierarchical relationships as effectively as more complex networks. The simpler design helps reduce overfitting, and the gradual reduction in neurons helps the network focus on the most important features.

Deep Network (Bottleneck)

Model 2 is a "bottleneck" architecture, which consists of a series of layers that have 128, 64, 32, and 128 neurons in this order. This structure is designed to create a bottleneck, forcing the model to learn compact, abstract representations. We are exploring the bottleneck structure (wide layers at the ends and narrower layers in the middle) because, as highlighted in the

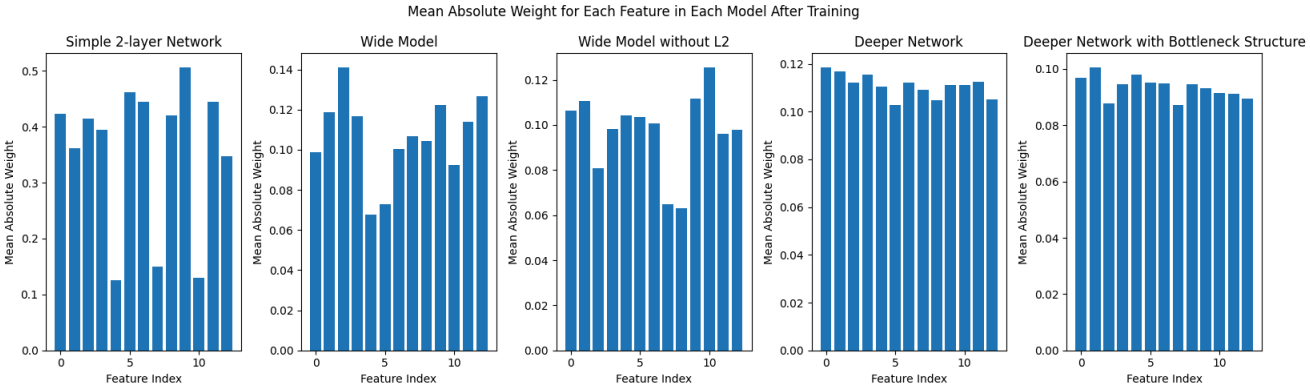
topology paper [insert citation], this architecture has been shown to be effective. While I've chosen not to delve into the details of the paper in this project to avoid adding extra time to the explanation, the bottleneck structure is believed to improve the model's ability to better capture data topology. This could be related to concepts like homeomorphism and bijection, where the model's ability to map inputs to outputs more efficiently helps it "unravel" the data, capturing essential features while simplifying the representation in the middle layers. This structure might enable the model to maintain useful relationships between variables while reducing unnecessary complexity in the hidden layers, leading to improved generalization.

Expected performance: Likely better at capturing complex patterns but more prone to overfitting.

Architecture:

- **Layer 1 (128 neurons)**: This receives the input from the feature space (e.g., the scaled training data). This initial layer captures general patterns and relationships within the data. With 128 neurons, it has sufficient capacity to learn many features, but it's not yet focused on abstracting the data.
- **Layer 2 (64 neurons)**: This reduction forces the model to start condensing the information and focusing on the most important features. The representation here becomes more abstract compared to the first layer, but it's still relatively general.
- **Layer 3 (64 neurons)**: This layer further makes the data representations even more compressed and abstract. At this point, the network has to distill the most important patterns and relationships from the input features and previous layers. The 32 neurons in this layer act as a true bottleneck, limiting the model's capacity and encouraging it to focus on the most essential information.
- **Layer 4 (128 neurons)**: This expansion allows the model to use the abstract, compressed features learned in the bottleneck layer to produce more detailed and refined representations before making a prediction. In essence, the model is trying to reconstruct the data's complexity with the insights it gained from the compressed middle layers.
- **Output Layer (1 neuron)**: Finally, the model outputs a single neuron, representing the predicted value for regression. The predictions here are based on the abstracted features that the network learned throughout its layers.

Comparing the Models



Weight Distribution Analysis

In the simpler two-layer network and the wide model without regularization, the uneven distribution implies that these models rely more on a subset of features while largely disregarding others, which is the concept of sparsity in the linear context of Lasso, potentially leading to a less comprehensive understanding of the data.

The deeper networks demonstrate a more balanced allocation of weight, reflecting the models' ability to learn more nuanced representations of the data and capture a variety of patterns and non-linear interactions across the input features, increasing their capacity to model intricate relationships. The combination of sparsity-inducing regularization and a deep network structure does a better job at striking the balance between overfitting and underfitting than either technique individually.

```
Simple 2-layer Network with 32 Neurons MSE: 24.7924
Wide Model MSE: 16.528930051673456
Wide Model without L2 Regularization MSE: 16.651329908586902
Deep Model MSE: 13.504876511513258
Deep bottleneck Model MSE: 10.669684182028318
```

- **Simple 2-Layer Network**: Highest error. Its limited depth and neuron count mean it lacks the capacity to capture complex data relationships. The simplicity does make it faster and less prone to overfitting, but at the cost of model performance.
- **Wide Model**: Better than the simple network. Its greater neuron count allows for richer data representation, enabling the network to capture more intricate patterns in the data. This improvement suggests that width can contribute positively to model accuracy, though too many neurons may introduce overfitting. The L2 regularization helps prevent overfitting, which helps the model generalize better on unseen data.
- **Wide Model without L2 Regularization**: Slightly higher MSE, indicating that the absence of regularization causes the model to slightly overfit the training data. This demonstrates the effectiveness of regularization in improving model generalization.

- **Deep Model**: Considerably lower error than what we have seen so far. This configuration supports hierarchical feature learning. Adding layers (depth) appears to significantly boost performance for complex datasets (as we already know).
- **Deep Bottleneck Model**: The reduced number of neurons in the middle seems to improve performance by reducing the number of parameters that the model needs to learn. This typically helps reducing the risk of overfitting. This is also consistent with what the researchers presented in "Topology of Deep Neural Networks" by University of Chicago and Technion – Israel Institute of Technology, where they discussed that a bottleneck structure performs well on a data set with convoluted topology.

Observed Correlations Between Network Architecture and Data Topology

- **Deeper networks tend to perform better on more complex data**. This is because additional layers enable the network to learn hierarchical representations of the data, allowing it to capture intricate relationships and abstract features.
- **Wider models, with more neurons in each layer, can capture more detailed patterns within the data**, improving performance compared to simpler networks.
- **Regularization, particularly L2 regularization, can play a role in improving the generalization of the model**, especially for wider networks. Although sparsity may not always lead to better predicting power, because it can also cause a good network to underfit, it is worth investigating regularization on a data set when we are striving for better accuracy.
- **When there is redundancy in the data, a bottleneck architecture can help isolate the most essential features**. The bottleneck forces the model to focus on more meaningful representations, reducing overfitting and improving generalization, especially for complex datasets.

Conclusion

This project explored the impact of sparsity in both Lasso regression and neural network architectures, emphasizing how sparse structures can enhance interpretability, efficiency, and generalization. Our analysis showed that sparsity in neural networks, encouraged by regularization techniques and deep architectures, mirrors the effects seen in Lasso regression.

Prior research, such as the work of Tibshirani on Lasso regression, underscores the value of sparsity for feature selection in statistical models (Tibshirani, 1996). Similarly, the insights from Srebro et al. (2004) on matrix factorization provided a mathematical basis for our findings, demonstrating that sparsity can be applied broadly, regardless of the specific loss function. Our findings align with these prior works but extend them by discussing how the performances of deep neural networks are directly affected by deliberate regularization.

Limitations and Future Research

A significant limitation of this work is that while we establish theoretical insights, the empirical performance and specific conditions under which these sparse structures optimize generalization in real-world data are beyond the scope of this study. Additionally, while we showed that deeper networks inherently tend to favor sparse representations, this sparsity might not always yield optimal results depending on the dataset's complexity and noise levels.

Future research could focus on empirical studies testing these theoretical insights across a variety of datasets and network architectures. There is also potential to explore adaptive sparsity techniques that dynamically adjust to the data's characteristics and develop algorithms that balance sparsity and model expressiveness. Further investigations into the efficiency of training sparse deep networks, especially in the context of NP-hardness in optimization, could also provide valuable insights into scalable implementations in large-scale applications.

References

- Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- Srebro, N., Rennie, J., & Jaakkola, T. (2004). Maximum-margin matrix factorization. *Advances in Neural Information Processing Systems*, *17*, 1329-1336.
- Tibshirani, R. J. (2021). Equivalences Between Sparse Models and Neural Networks. *Journal of Machine Learning Research*, *22*, 1-20.
- Chen, X., Xu, M., Caramanis, C., & Mannor, S. (2015). *The hardness of sparse approximations*.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. MIT Press.
- Naitzat, G., Zhitnikov, A., & Lim, L.-H. (2020). Topology of Deep Neural Networks. University of Chicago and Technion – Israel Institute of Technology.
- Stack Overflow. (2017). Keras: How to get the output of each layer? Retrieved from https://stackoverflow.com/questions/41711190/keras-how-to-get-the-output-of-each-layer
- Stack Overflow. (2019). Lasso regression with Python, simple question. Retrieved from https://stackoverflow.com/questions/54333467/lasso-regression-with-python-simple-question
- Stack Overflow. (2019). Regularization in neural networks. Retrieved from https://stackoverflow.com/questions/56251748/regularization-in-neural-networks
- Stack Overflow. (2022). Extracting first layer weights from a multi-layer Keras NN and transferring them to another model. Retrieved from https://stackoverflow.com/questions/71114994/extracting-first-layer-weights-from-a-multi-layer-keras-nn-and-transferring-them

---

Appendix

All of the code used in this project can be found at this github repository

We run this command in a terminal to have the file exported to HTML with select code blocks displayed and hidden as designed.

jupyter nbconvert sparse.ipynb --to html --TagRemovePreprocessor.enabled=True --TagRemovePreprocessor.remove_input_tags="{'hide_input'}" --TagRemovePreprocessor.remove_all_outputs_tags="{'hide_output'}"

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Lasso
import tensorflow as tf
from tensorflow import keras
from keras import layers
from keras import models
from keras import optimizers
from keras import regularizers
from keras import callbacks
%matplotlib inline
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2
```

In [ ]:
```
column_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', '
data = pd.read_csv('housing.csv', header=None, delimiter=r"\s+", names=column_name
data.head()
X = data.drop(['MEDV'],axis=1).to_numpy()
y = data['MEDV'].to_numpy()
train_data, test_data, train_targets, test_targets = train_test_split(X, y, test_s
```

In [ ]:
```
# A couple of functions to be used several times in this project

# Get the weights of all layers (excluding the output layer)
def calculate_mean_absolute_weights(model, input_shape):
    """
    Calculate mean absolute weights for each feature across all layers in a neural

    Parameters:
    - model: Trained Keras model (Sequential or Functional).
    - input_shape: Shape of the input data (number of features).

    Returns:
    - mean_absolute_weights: A numpy array containing mean absolute weights for ea
    """
    # Initialize a list to store mean absolute weights per variable (feature)
    mean_absolute_weights = []

    # Get the weights of each layer in the model
    layer_weights = [layer.get_weights()[0] for layer in model.layers if len(layer

    # Loop through each variable (feature)
    for feature_index in range(input_shape[1]):
        abs_weights = []

        # For each layer's weights, find the associated weight for the current fea
        for weight_matrix in layer_weights:
            # The feature corresponds to the columns of the weight matrix
            if weight_matrix.shape[1] > feature_index:
                # Get the absolute weight for the current feature across all neuro
                abs_weights.extend(np.abs(weight_matrix[:, feature_index]))  # For

        # Calculate the mean of the absolute weights for this feature
        mean_abs_weight = np.mean(abs_weights)
        mean_absolute_weights.append(mean_abs_weight)

    # Convert to a numpy array for easier analysis
    mean_absolute_weights = np.array(mean_absolute_weights)

    return mean_absolute_weights


def train_and_stop_model(model, X_train, y_train):

    # Define the early stopping callback
```

```
    early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_w

    # Train the model with early stopping
    history = model.fit(X_train, y_train, epochs=100, validation_split=0.2, callba

    # Determine the stopped epoch based on the history
    stopped_epoch = len(history.history['loss'])

    # Return the history and stopped epoch
    return stopped_epoch

def plot_mean_absolute_weights_bar(model):
    mean_abs_weights = calculate_mean_absolute_weights(model, X_train_scaled.shape

    plt.bar(range(X_train_scaled.shape[1]), mean_abs_weights)
    plt.xlabel('Feature Index')
    plt.ylabel('Mean Absolute Weight')
```

In [ ]:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(train_data)
X_test_scaled = scaler.transform(test_data)

# Model 1: Simple Feedforward Network
model = Sequential([
    Dense(32, activation='relu', input_shape=(X_train_scaled.shape[1],), kernel_re
    Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mse')


# Train the model
model.fit(X_train_scaled, train_targets, epochs=100, validation_split=0.2, verbose

# Evaluate the model
y_pred_nn_1 = model.predict(X_test_scaled)
mse_nn_1 = mean_squared_error(test_targets, y_pred_nn_1)

lasso = Lasso(alpha=0.1)
lasso.fit(X_train_scaled, train_targets)

# Neural Network Predictions
y_pred_nn = model.predict(X_test_scaled)

# Lasso Predictions
y_pred_lasso = lasso.predict(X_test_scaled)

# Calculate performance metrics
mse_nn = mean_squared_error(test_targets, y_pred_nn)
r2_nn = r2_score(test_targets, y_pred_nn)

mse_lasso = mean_squared_error(test_targets, y_pred_lasso)
r2_lasso = r2_score(test_targets, y_pred_lasso)
```

In [ ]:
```
print("Neural Network Performance:")
print(f"MSE: {mse_nn:.4f}, R²: {r2_nn:.4f}")

print("\nLasso Regression Performance:")
print(f"MSE: {mse_lasso:.4f}, R²: {r2_lasso:.4f}")
```

In [ ]:
```
# Lasso regression coefficients
lasso_coefficients = lasso.coef_
mean_abs_weights = calculate_mean_absolute_weights(model, X_train_scaled.shape)

# Plot Lasso coefficients
plt.figure(figsize=(12,5))
plt.subplot(1, 3, 1)
plt.bar(range(len(lasso_coefficients)), lasso_coefficients)
plt.title('Lasso Coefficients')
plt.xlabel('Features')
plt.ylabel('Coefficient Value')

# Plot the average magnitude of neural network weights
plt.subplot(1, 3, 2)
plt.bar(range(X_train_scaled.shape[1]), mean_abs_weights)
plt.xlabel('Feature Index')
plt.ylabel('Mean Absolute Weight')
plt.title('Mean Absolute Weight for Each Feature')

nn_weights_layer_1 = model.layers[0].get_weights()[0]  # Weights for first layer

# Visualizing sparsity in the neural network weights
plt.subplot(1, 3, 3)
plt.hist(nn_weights_layer_1.flatten(), bins=50)
plt.title('Neural Network Weights Distribution (Sparse)')
plt.xlabel('Weight Value')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

In [ ]:
```
# Assuming X_test_scaled contains the scaled features, and test_targets is the tar
# Extract TAX (Feature 10) and TAX (Feature 6) for plotting
tax = X_test_scaled[:, 9]  # TAX is feature 10
rm = X_test_scaled[:, 5]  # RM is feature 6
medv = test_targets  # Target variable (MEDV)

# Plot 1: TAX vs. MEDV with smooth line
plt.figure(figsize=(12, 5))

# Scatterplot for TAX vs. MEDV
plt.subplot(1, 2, 1)
sns.regplot(x=tax, y=medv, scatter_kws={'s': 10}, line_kws={'color': 'red', 'lw':
plt.title("TAX vs. MEDV with Smooth Line")
plt.xlabel("TAX (Full-value property-tax rate per $10,000)")
plt.ylabel("MEDV (House Prices in $1000s)")
```

```
# Plot 2: RM vs. MEDV with smooth line
plt.subplot(1, 2, 2)
sns.regplot(x=rm, y=medv, scatter_kws={'s': 10}, line_kws={'color': 'blue', 'lw':
plt.title("RM vs. MEDV with Smooth Line")
plt.xlabel("RM (Average Number of Rooms per Dwelling)")

# Show plots
plt.tight_layout()
plt.show()
```

In [ ]:
```
wide_model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_scaled.shape[1],), kernel_r
    Dense(1)
])

wide_model.compile(optimizer='adam', loss='mse')

no_l2_wide_model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dense(1)
])

no_l2_wide_model.compile(optimizer='adam', loss='mse')
```

In [ ]:
```
model_1 = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_scaled.shape[1],), kernel_r
    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(32, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(16, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(1)
])

model_1.compile(optimizer='adam', loss='mse')
```

In [ ]:
```
model_2 = Sequential([
    Dense(128, activation='relu', input_shape=(X_train_scaled.shape[1],), kernel_r
    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    Dense(1)
])

model_2.compile(optimizer='adam', loss='mse')
```

In [ ]:
```
plt.figure(figsize=(16,5))
plt.suptitle('Mean Absolute Weight for Each Feature in Each Model After Training')

plt.subplot(1, 5, 1)
plot_mean_absolute_weights_bar(model)
plt.title('Simple 2-layer Network')
plt.subplot(1, 5, 2)
plot_mean_absolute_weights_bar(wide_model)
plt.title('Wide Model')
```

```
plt.subplot(1, 5, 3)
plot_mean_absolute_weights_bar(no_l2_wide_model)
plt.title('Wide Model without L2')
plt.subplot(1, 5, 4)
plot_mean_absolute_weights_bar(model_1)
plt.title('Deeper Network')
plt.subplot(1, 5, 5)
plot_mean_absolute_weights_bar(model_2)
plt.title('Deeper Network with Bottleneck Structure')

plt.tight_layout()
plt.show()
```

In [ ]:
```
wide_SE = train_and_stop_model(wide_model, X_train_scaled, train_targets) #SE = st
wide_nol2_SE = train_and_stop_model(no_l2_wide_model, X_train_scaled, train_target
m1_SE = train_and_stop_model(model_1, X_train_scaled, train_targets)
m2_SE = train_and_stop_model(model_2, X_train_scaled, train_targets)

y_pred_nn_1 = model_1.predict(X_test_scaled)
simple_mse = mean_squared_error(test_targets, y_pred_nn_1)

y_pred_nn_2 = model_2.predict(X_test_scaled)
deep_mse = mean_squared_error(test_targets, y_pred_nn_2)

y_pred_nn_3 = wide_model.predict(X_test_scaled)
wide_mse = mean_squared_error(test_targets, y_pred_nn_3)

y_pred_nn_4 = no_l2_wide_model.predict(X_test_scaled)
no_l2_wide_mse = mean_squared_error(test_targets, y_pred_nn_4)
```

In [ ]:
```
print(f"Simple 2-layer Network with 32 Neurons MSE: {mse_nn:.4f}")
print(f"Wide Model MSE: {wide_mse}")
print(f"Wide Model without L2 Regularization MSE: {no_l2_wide_mse}")
print(f"Deep Model MSE: {simple_mse}")
print(f"Deep bottleneck Model MSE: {deep_mse}")
```