Visualisierungsinstitut der Universität Stuttgart (VISUS)

Masterarbeit

Visual Exploration for Deep Learning Models and Trainings for Microstructure Data

Hai Dang Nguyen

Course of Study:	Informatik
Examiner:	Prof. Dr. Daniel Weiskopf
Supervisor:	David Hägele, M.Sc., Julian Lißner, M.Sc., Tanja Munz, Dipl.-Inf., M.Sc.
Commenced:	April, 2022
Completed:	November, 2022

Abstract

Artificial neural networks have become a staple in machine learning research and are employed in many interdisciplinary domains. Thus, it is paramount to understand their inner workings when designing and developing new models. One field of research that is particular useful is called visual analytics, which combines interactive visual representations and data analysis algorithms to obtain knowledge. The aim of this thesis is the development of a visual analytics system to analyze the training behavior of a machine learning model predicting microstructure material responses. The goal is to enable the user to explore how different training configurations influence the training process and the model's performance. In addition, a novel regularization technique and a novel optimization improvement, greedy stochastic permutations, are proposed.

Contents

1 Introduction

In recent years, neural networks have seen a surge in popularity in research. With the risen popularity, not only is more effort invested in improving neural networks, but also in understanding neural networks to facilitate that improvement. Neural networks are often seen as a black-box that is fed with data and outputs predictions with its inner working unbeknownst due to its complexity. The source of this complexity stems from the fact that in most non-trivial neural networks, a multitude of neurons are necessary to achieve adequate results. In addition to the sheer amount of neurons in a deep neural network, the interactions between different optimizers and their learning process is difficult to grasp without tools. Neural networks can come in different architectures and varying levels of deepness and wideness, and it is nontrivial to figure out which architecture works best for a specific problem. Further, there are many different hyperparameters that require expert tuning, for example, the learning rate or regularization options. Understanding how different training data and different hyperparameter settings affect prediction accuracy is difficult even for experts with years of experience. However, by using certain tools and techniques to solve these issues, researchers can obtain improved insight and enhanced interpretability for their neural networks to locate and ameliorate flaws and to ultimately create better networks.

One effective way to gain more insight is to use a visual analytics system. During the training of neural networks, a huge amount of data is generated. Combing through this data manually to find information on the effects of each hyperparameter is far from practical. To handle huge amounts of data, it is possible to use data analysis algorithms that can find correlations and structures in the data or use visualizations that enable the user to understand vast amounts of data at once. In general, visual analytics combines automated data analysis with interactive visualization tools. It allows the user to visually explore the data with interactive visual representations of the data by employing the user's experience and intuition. In the tool developed in this work, the exploration process starts with the user setting the training configuration for a neural network. Afterwards, the training commences and data is gathered during the training. This data can then be visualized interactivly by the user. To this end, several visualization tools provide support in analyzing and comparing the training history. With the insights gained from the analysis, the user then adjusts the training configuration and the process starts anew. Practically, this can be used to gain knowledge on how different optimimzers compare to each other or which hyperparameter configurations work well with the training data.

This work will showcase one such visual analytics system that was build specifically to allow explorations for neural network models predicting microstructure material responses. The use of machine learning techniques for predicting material properties, like heat or electric conductivity, given the microstructure is reasoned in [LF19]. Given the microstructure, it is possible to compute the properties through virtual testing, but this approach is costly. Instead, the more efficient approach is to use a trained neural network to predict the material properties. Training the neural network requires a dataset of microstructure images with their properties annotated. To be used in a neural network, a microstructure image needs to be converted into a feature vector. This is done by

multiplying its two-point correlation function with a precomputed reduced basis. The resulting feature vector can then be fed into the network, and a prediction can be obtained. By comparing the prediction with the annotation in the dataset, the neural network will then move towards the correct parameters to minimize its prediction error. Once the network is sufficiently trained, it should be able to accurately predict the right material properties, even for unseen microstructure images. This can be used to discover materials with desired material properties, by randomly generating different microstructures until the network predicts fitting properties.

The goal of this visual analytics system is to facilitate the exploration of different neural network models predicting the material properties, and discover the optimal network architecture and hyperparameters that yield the best prediction. This system is build as a web-based system with a web application as the front end. Users are able to interact with the web page to build, configure, and train neural networks. The resulting training history is then available for the user, and can be interactively visualized using various tools.

In addition to that, experiments on weight regularization methods are performed to empirically test hypothesis on how smaller network weights affect a model's performance. Further, a novel optimization approach is proposed that can significantly improve the performance of common optimizers like SGD or Adam. All experiments are done and supported by the visual analytics systsem, showcasing its usefulness in exploring deep learning models and trainings to obtain more knowledge and insights.

This thesis is structured as follows: After this introduction, the related works are listed and described in Chapter 2. In Chapter 3, the methods and the data that are used in this work are presented by laying the theoretical groundwork. Next, in Chapter 4, a more in-depth description of this visual analytics tool is given and the practical implementations of the methods in Chapter 3 are described. Afterwards, in Chapter 5, the insights that were gained during the visual exploration of deep learning models and trainings for microstructure data are presented together with visualizations that illustrate the results. Lastly, a conclusion is provided and an outlook given in Chapter 6.

2 Related Work

In this chapter, an overview of the works that are related to this thesis is given. Visual Analytics is an active research field in visualization and many visual analytics systems have already been developed. Additionally, multiple visualization tools that are used in this work stem from research in the field of visualtization and the field of deep neural networks. This thesis also builds on the many works of the very active machine learning research on deep learning.

A crucial work for this thesis is [LF19]. In that paper, Lißner motivates the use of deep learning neural networks for predicting the resulting material properties of experimental materials given the underlying microstuctures. The two-dimensional microstructure data set and the code to extract the representative feature vectors necessary for neural networks were provided by them.

In regards to visual analytics systems, this work takes inspiration from other such systems like ClaVis [HMN+20]. Many of the functionalities and tools can also be found in other papers. PathExplorer [HSS+21] showcases a visualization of decision-making paths using dimensionality reduction techniques in order to find patterns. Similarly in this work, dimensionality reduction techniques are employed to create visualizations of the training path of neural network models. One popular dimensionality reduction technique is principle component analysis (PCA), which is explained in [AW10] but is originally described by Pearson in [Pea01]. A newer dimensionality reduction technique that is also used here is t-SNE, first described by Van der Maaten and Hinton in [VH08], and builds upon the Stochastic Neighbor Embedding (SNE) described in [HR02]. Lastly, UMAP is a more modern dimensionality reduction algorithm. It was proposed in [MHM18] in 2018 and provides a similar visualization as t-SNE. These techniques allow the visualization of multivariate data by reducing high dimensional data points to a lower dimensional space. In most cases, this means a projection onto two dimensions which can be plotted on a flat screen. The implementation details depend on the respective algorithms, but in general, the information loss that inevitably occurs when projecting to a lower amount of dimensions should be minimized, i.e. nearby points in high dimensional space should also be nearby in the low dimensional projection space.
Besides dimensionality reduction, another visualization process called loss landscape is part of this visual analytics system. [LXT+18] presents a way to explore the loss function of a neural network by visualizing its high-dimensional optimization space using methods from topographic maps in dependence to the network parameters. Alternatively, this can also be used to visualize the optimizer path in robot motion planning as [HAO+22] shows.

On the machine learning side of things, this thesis is build on top of works pertaining to training techniques for neural networks. One focal training configuration that can be set in this tool are the regularization options. Regularization strategies play an important role in the training of deep learning models. They strive to prevent overfitting and improve the ability of the model to generalize from the training data. There is however no strategy that works for every problem, and for each model configuration a different approach might lead to the best result. Thus, over the years many diverse regularization techniques were developed and tested. An overview on the various options

can be found in [MBM20]. Here, two strategies are considered: The first one is called weight decay. As the name suggests, the main idea here is to reduce the value of the weights of the neural network. This is accomplished by adding a regularization term in the objective function that punishes high value weights. The other regularization strategy is dropout, and is described in [SHK+14]. The general idea of dropout is that each neuron in a neural network has a certain chance to drop out of a learning epoch, meaning that the input to that neuron is set to zero. This way, during each training epoch, one of all possible thinned subnetworks is trained instead of the whole network, effectively training multiple thinned models whose weights are highly shared. This results in significant reduction in overfitting, making this a popular choice for regularization since it also works on almost all neural network architectures.

Apart from regularization techniques, this work will also look at some other machine learning mechanisms like early stopping, ensemble learning, and simulated annealing. Early stopping as a concept existed as early as 1998 by Prechelt in [Pre98]. The question behind that concept is to find a way to choose the optimal number of training epochs to balance the bias-variance trade-off. If the training is too short, the model is underfitted, leading to high bias, whereas when the training is too long, the model is overfitted and as a result has a high variance. The best training duration lies somwhere inbetween and early stopping can be used to find the optimal balance. Basically, during the training the network is tested on validation data, which represents unseen data that was not in the training. As soon as the measured validation loss, which does not contribute to optimization, worsens, the training is stopped. In practice, however, a less strict stopping criterion is employed: the training stops after n epochs of no improvements where n is a hyperparameter to be tuned. The reason for this is the fact that during the optimization process, the validation loss can temporarily raise, only to drop lower than before the increase.

This work also includes a short dive into ensemble learning. The core idea of ensemble learning is to train multiple networks on the same task and using a combination of their predictions to come to a solution. Here, instead of using multiple models, the neurons on the output layer are duplicated, which results in a multi-headed network where each head represents an ensemble learner similar to [CS20] where one single ConvNet was used instead of a group of ConvNets.

Lastly, a big part of this thesis is the exploration of the effect of simulated annealing techniques on the training of neural networks. Simulated annealing originally stems from optimization research like in [KGV83], and describes how it is possible to reach global optimas by using stochastic samples. Here, this concept is carried over to find the global minima in the loss function of the models. After every nth step, random perturbations are created and sampled to find a potentially better starting point for the next step.

3 Methods and Data

In this chapter, several relevant concepts and techniques are explained to lay the groundwork for future chapters. The main task of this thesis is to build a visual analytisc system that can assist in developing neural networks and gaining insights about them. In order to understand this work, a certain level of knowledge of neural networks is required. Thus, in the first section of this chapter a brief explanation of the basics is given. The next section of this chapter focuses on the different regularization methods that are used in this work. Afterwards, the different visualization techniques that are used in this tool are described. Finally, the training data that all models in the experiments train on is introduced.

3.1 Artificial Neural Networks

Artificial Neural Networks are frequently used in supervised machine learning due to their scalability with big amounts of data. When doing supervised machine learning, annotated training data is required, and in the age of big data they have become more and more popular. Once sufficiently trained, they can swiftly process unseen data. The processing speed can be further increased by using less accurate networks. Vice versa, the accuracy can be improved at the cost of their speed. In any case, their use cases are plentiful, they are used in natural language processing, computer vision, robotics and many more.

A neural network consists of multiple neurons. They were inspired by their biological counterpart and they receive and forward signals. Usually, these neurons are organized into layers, with each layer connecting to the previous and the next layer. Each individual neuron can receive inputes from neurons of the previous layer and its output is fed to neurons in the next layer. Of course, there are exceptions and different network architectures exists where neuron connections are less restrictive.

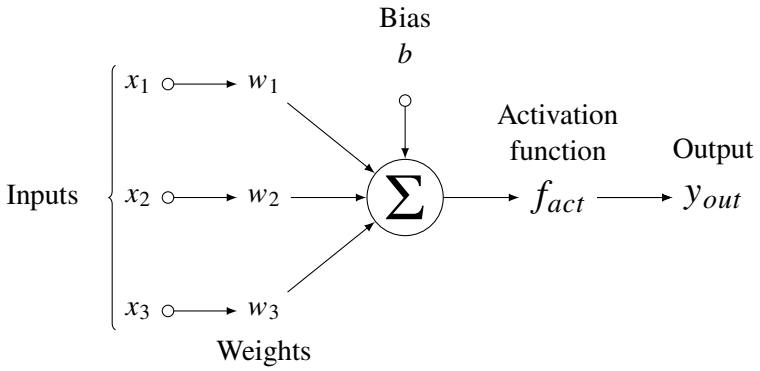


Figure 3.1: Operation principle of single neuron

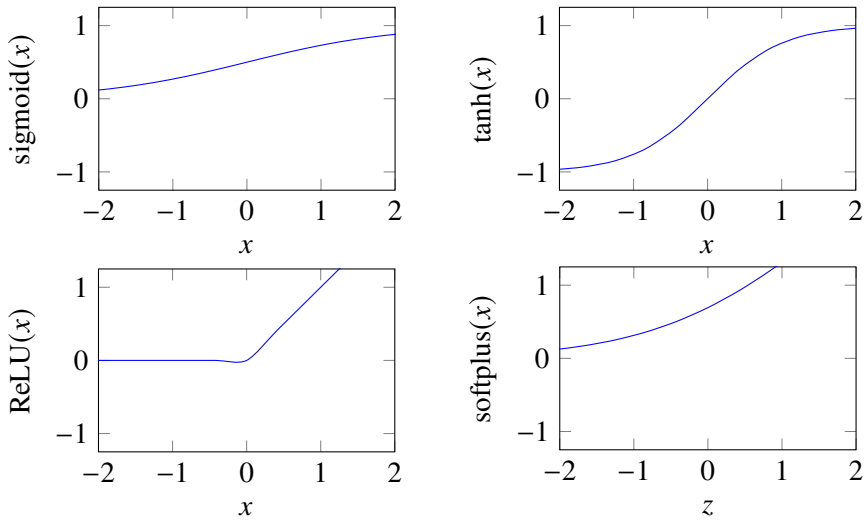


Figure 3.2: Examples of commonly used activation functions: logistic sigmoid, hyperbolic tangent, rectified linear unit, and softplus

In this work, however, the neurons are only connected to neurons in neighboring layers. Figure 3.1 depicts the typical neuron: the inputs from the preceding neurons are weighted with individual weight values. In addition, each neuron has a bias value. All weighted inputs and the bias are then summed up and put in an activation function. The output is then forwarded to following neurons. In equation form that is:

$$y_{out} = f_{act}(b + x_1 \cdot w_1 + x_2 \cdot w_2 + ... + x_n \cdot w_n)$$

The activation function is often a nonlinear function since without any nonlinear activation functions the whole network will be linear and thus unable to solve non-trivial problems. Examples of commonly used activation functions are depicted in Figure 3.2.

The first layer of an artificial neural network is called input layer and is filled with the values of the feature vector. A feature vector contains the features of a data instance of a training data set. This can be, for example, the age, the height, and the sex of a patient. Numerical or boolean features can be put directly in the feature vector, while categorical features are usually one-hot encoded. Input images usually have their individual RGB values of each pixel as feature vectors. The last layer is called the output layer. The amount of output neurons will depend on the problem, for example, predicting tomorrow's temperature is a one-dimensional regression task and thus only has one output neuron that predicts the temperature, while image classification tasks have an output neuron for each class that outputs the probability of the input image being in that class.

Layers that are neither the input layer nor the output layer are called hidden. Depending on the type of neural network, there are different types of hidden layers, like convolutional layers, encoder layers, embedding layers and many more. In this work, the hidden layers are all fully connected layers, which means that a neuron from a fully connected layer receives inputs from all neurons from the preceding layer and forwards its output to all neurons in the following layer. Figure 3.3 shows an example neural network with all connections drawn.

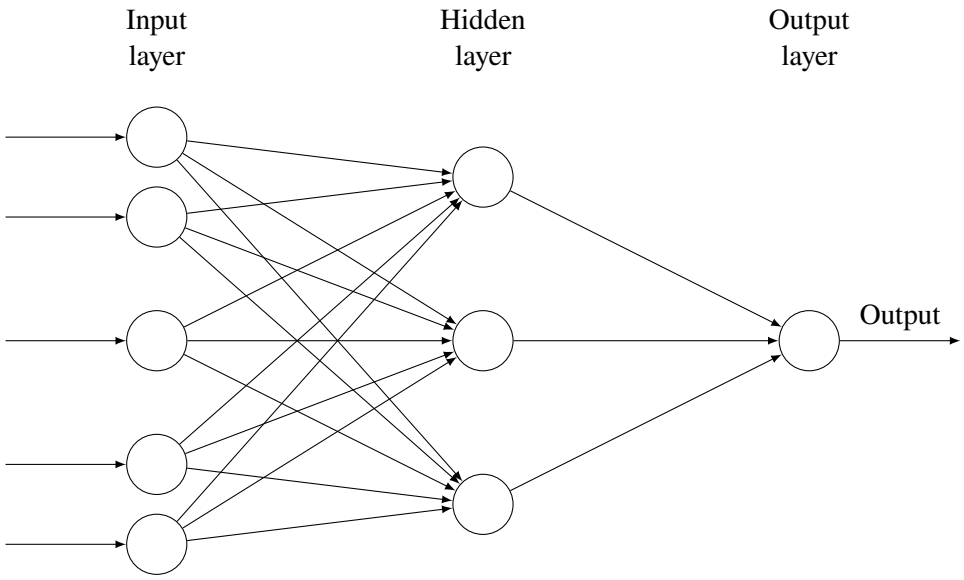


Figure 3.3: Fully connected neural network with one hidden layer and one output neuron

To learn from the training data, backpropagation is used. This algorithm computes the gradient of the loss function with respect to the neurons' weights. The loss is a measure of how far off the predictions of the network is to the actual annotation in the training data. The gradient is calculated first for the last layer with respect to the neurons' weight. Changing these weights according to the gradient would lead to a better loss. By iterating backwards, layer for layer, the gradients for every neurons' weight can be obtained efficiently.

Training an artificial neural network consists of iterative improvements of the weights. A training epoch is over when the whole training data set is put through the network and all the gradients are calculated. Afterwards, an optimizer returns improved weights using the gradients. Usually, a training consists of hundreds or even thousands of epochs, until the weights converge, i.e., when the early stopping condition is met. At that point a loss minima is reached.

3.2 Regularization Methods

To motivate the use of regularization methods, it is important to learn about a common concept in machine learning which is the variance-bias tradeoff. Bias describes the distance between a model's average prediction and average true values, while variance describes the spread of a model's prediction. High bias occurs if a model cannot learn the latent data representation of the training data, which is called underfitting. In contrast, high variance occurs if the model learned too much of the training data and does not generalize well on unseen data, which is called overfitting. Ideally, the optimal network has low variance and low bias. In reality, there is a tradeoff between both, and balancing the model's complexity is key to reduce the total error. Figure 3.4 visualizes this tradeoff: a model with low complexity cannot learn the latent data representation and thus has high bias.

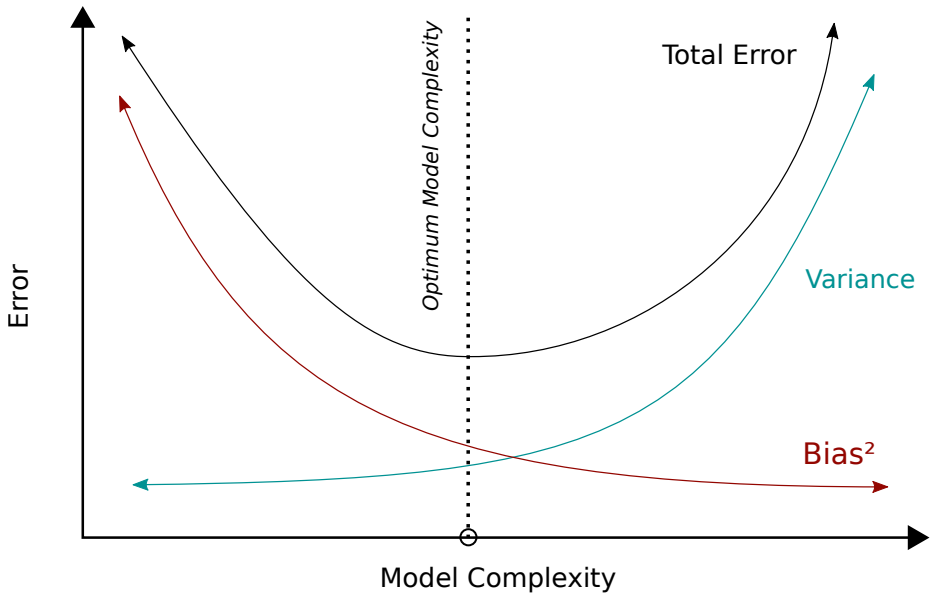


Figure 3.4: Bias and variance as function of model complexity

With high complexity, the model gains more and more degrees of freedom in its parameters and can minimize the training loss by learning the data by heart. When confronted with unseen data the model fails due to the high variance. Thus, the optimal spot is somewhere in the middle.

Overfitting is more often a concern than underfitting and over the years several techniques were developed for neural networks that aim at preventing overfitting. They are often called regularizers. In this work, two state of the art regularization methods are used: weight decay and dropout. Additionally, novel regularization techniques are proposed and their effectiveness empirically tested.

The first method assumes that overfitting occurs due to bigger weight values, and aims to keep these weights small. As a result, the effective output range of each neuron and the spread of the model's predictions is reduced and thus the variance is lowered. Weight regularization punishes high values in the weights by adding a regularization term in the loss function. Usually, the loss functions measures the model's prediction error and its gradient tells the model how to change the weights to achieve a better prediction. By adding a term to the loss that scales with a network's weights, its gradient will now also try to improve the predictions while keeping the weights as small as possible. The regularizing term can theoretically be anything that increases monotonically, in practice the L_2-Norm is used. With that, the loss Φ looks like this:

$$\Phi = L(Y_{\text{true}}, Y_{\text{pred}}) + \alpha \parallel W \parallel_{L_2}$$

where L is the loss function, α the weight decay factor, and W the weight tensor of the network.

The latter regularization method is called dropout because there each neuron has a certain probability to drop out during training, meaning that the input to that neuron is set to zero, whereas for all other neurons in that layer which do not drop out, the input is scaled such that the sum over all inputs remains unchanged. This way, during each training epoch, one of all possible thinned subnetworks is trained instead of the whole network, effectively training multiple thinned models whose weights are highly shared.

3.3 Other Machine Learning Methods

Apart from regularization, there are other machine learning schemes available that improve a model's performance. One example includes using cyclical learning rate as proposed in [Smi17]. In this work, two well known methods, early stopping and a variant of ensemble learning, and a novel optimization improvement, which was developed in cooperation with a supervisor, are tested.

3.3.1 Greedy Stochastic Perturbations

Greedy Stochastic Perturbations (GSP) is the name of the novel optimization improvement proposed in this work. During the training of a neural network an optimizer is employed to perform a gradient descent step with the goal of arriving at the global loss minima. During that process, the optimizer might get stuck at some local minima. To prevent that, the weight tensor of the network is perturbed randomly, and after each perturbation the loss is measured, it jumps to the best performing perturbation.

The random perturbations are obtained as follows: let W_{curr} be the current weight tensor of the network and W_{last} be the weight tensor of the network before the last optimization step. The perturbation tensor P is then:

$$P = W_{curr} - W_{last}$$

Next, a sign tensor S of the same dimension is generated. Each element in S is either -1 or 1, which is decided with equal probability during generation. To obtain a perturbed weight tensor W_P, the pertubation tensor is element-wise multiplied with S and scaled with an perturbation factor pf:

$$W_P = W_{curr} + pf \cdot (P \odot S)$$

The network is then fitted with the new perturbed weight tensor, and the validation loss is measured. This is repeated for multiple different perturbed weight tensors. If no perturbation has a lower loss than the validation loss with W_{curr}, then the network continues with that weight tensor. Otherwise, it takes the best performing perturbed weight tensor.

This optimization improvement aims to prevent the optimizer from getting stuck at local minimas and to accelerate the optimization by taking bigger steps due to the perturbation factor. This will be seen in the results of the experiments ran in Section 5.3. This improvement also works for any optimizer and is not too costly because measuring the validation losses is rather cheap since no gradients have to be calculated.

3.3.2 Ensemble Learning

Another way to receive better results is to use multiple models and average their predictions. This is called ensemble learning. The models used are usually of lower complexity but, as an ensemble, they can achieve better results. In this work, a variant of that is used. Here, instead of building multiple smaller models, the last layer of the neural network is repeated multiple times. The resulting architecture is a multi-headed neural network. Each head represents one instance of the ensemble. During training, the training set is modified accordingly to accommodate multiple predictions.

That way, the weights of each head receives individual corrections during the optimization steps. However, when measuring the validation loss, all outputs of each head are averaged to form a prediction.
The success of ensemble learning is well documented, but it remains to be known, if that multi-headed approach delivers the desired improvements here.

3.3.3 Early Stopping

In the previous section, the variance-bias tradeoff and the need to find the balance between them were explained. Too many epochs will result in overfitting, while too few epochs will underfit the model. Thus, finding the amount of training epochs that trains the model just enough to not overfit is necessary. This can be done by using the validation loss as a metric for the model's performance. With increasing training, that metric should decrease until a point where the model is starting to overfit. By monitoring the validation loss, it is possible to pinpoint that moment and immediately stop the training. In general, at the epoch where the validation loss stops decreasing, a counter is started and the validation loss is recorded. With each following epoch where the validation loss is not below the recorded loss, the counter increases. Otherwise it is reset to zero. If the counter surpasses a defined threshhold, the training is stopped. If the threshhold is set to low, then the training might stop too soon since during the optimization process, phases of temporary loss increase could occur, only to decrease further down the line. If the threshhold is set too high, too much unneccessary training might occur. Finding the right threshhold is one of the experiments that will be presented later in Section 5.5.

3.4 Visualization Techniques

Creating and training neural networks is only the first half of this work. What really enables exploration and gaining knowledge are visualizations. A network's training history contains way to many values to directly detect any patterns or outliers. To this end, visualization can help by exploiting a human's ability to quickly and effectivly recognize patterns in visual representations.

3.4.1 Dimensionality Reduction

For one-dimensional data, conventional plots suffice. For example, it is very trivial to plot a model's validation loss over all epochs. However, this is not possible for high-dimensional data like a network's weights. Yet, not visualizing it is also not an option, e.g., consider a very small neural network as depicted in Figure 3.3. Each of the three neurons in the hidden layers have five trainable weights with three more trainable weights in the output layer. This sums up to eighteen weights which seems manageable for very few epochs but in practice, the number of epochs are in the thousands. To solve that problem, multiple dimensionality reduction algorithms exist that aim to project high-dimensional data to low dimensional space while minimizing the inevitable information loss. Here, three are presented:
The first dimensionality reduction algorithm is Principal Component Analysis (PCA). First, a data matrix X is assembled. Each row of that data matrix contains a data point. Depending on the filters, this can be the weight tensor of a model at an epoch or just a part of it. In any case, the tensor

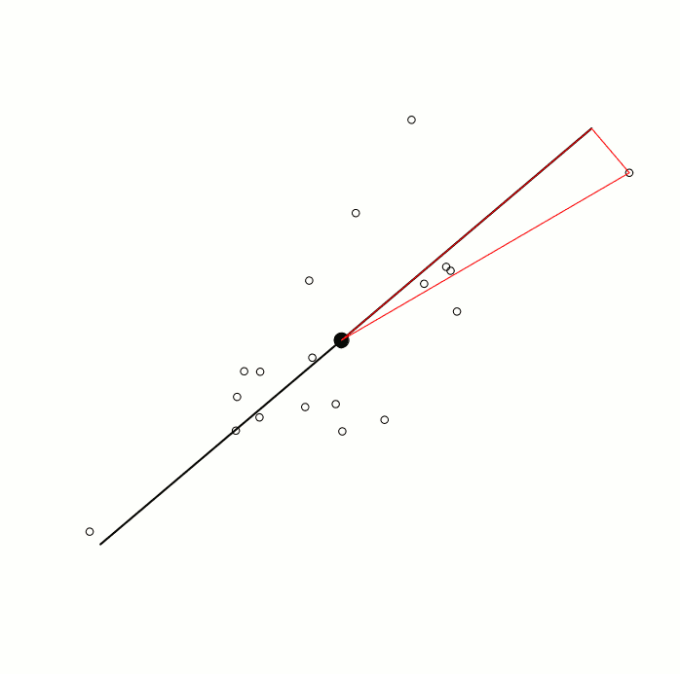


Figure 3.5: One-dimensional PCA on two-dimensional data points. Image from [Com09]

is flattened and added as a row to the data matrix. Each row has m components, and for n data points, the data matrix has size $n \times m$. Next, that matrix is corrected such that its mean is the origin. Afterwards, the covariance matrix is calculated:

$$C = \frac{X^T X}{n - 1}$$

which can be diagonalized:

$$C = VDV^T$$

V is the matrix of eigenvectors and D is the diagonal matrix with the eigenvalues on the diagonal in decreasing order. The projection of the data matrix is simply XV. For two dimensions, it is sufficient to only use the first two columns of V, i.e., only the two eigenvectors with the highest eigenvalue. In practice, running PCA is computational intensive. Even for small networks, the number of component can easily surpass one thousand and the number of data points is also in the thousands, making the calculation of the covariance matrix and its diagonalization costly. Luckily, a cheaper method exists: Singular Value Decomposition (SVD). With SVD, the data matrix can be decomposed into three different matrices:

$$X = USV^T$$

When projecting the data matrix, the projection is simply

$$XV = USV^T V = US$$

Figure 3.5 illustrates a one-dimensional PCA used on two-dimensional data. All data points are orthogonally projected onto the black line which was chosen such that the information loss, i.e., the sum of the squared distances of the points to the line, is minimalized. In two-dimensional PCA

the points are orthogonally projected onto the plane spanned by the first two eigenvectors of the covariance matrix C. That way, the variance of the data points is maximized, thus minimizing the squared distances.

The next dimensionality reduction algorithm is t-distributed stochastic neighbor embedding, or t-SNE in short. This algorithm was first presented in [VH08] and builds upon the Stochastic Neighbor Embedding (SNE) described in [HR02]. There, conditional probability distributions are established between each pair of datapoints in the high dimensional space. These are obtained by converting the euclidean distance between two points x_i and x_j to a conditional probability $p_{j|i}$ which is proportional to the probability that point x_i picks the point x_j as its neighbor under a Gaussian distribution centered at point x_i with variance σ_i. Similarily, this can be done in the lower dimensional projection with probabilities $q_{j|i}$. The goal now is to minimize the difference between the distributions P and Q to correctly model the similarities. This is done by minimizing the Kullback-Leibler divergence of both distributions via gradient descent. The result is that similar datapoints are kept nearby and different datapoints are further apart. Now, SNE has several problems upon which t-SNE improves. The main improvement is that instead of also using a Gaussian distribution in the lower dimensional space, t-SNE uses the Student's t-distribution with one degree of freedom. This solves the crowding problem that SNE visualizations run into, where the datapoints are crowded around the center, since the heavier tails of the t-distribution allow moderately similar datapoints to be further apart. The t-distribution is also faster to evaluate and needs less steps in the gradient descent.

Lastly, UMAP is a more modern dimensionality reduction algorithm. It was proposed in [MHM18] in 2018 and provides a similar visualization as t-SNE. It uses theory from Riemannian geometry and algebraic topology and claims to be an improvement over t-SNE. Going into the details would be outside the scope of this thesis, as such it is advised to refer to the original paper.

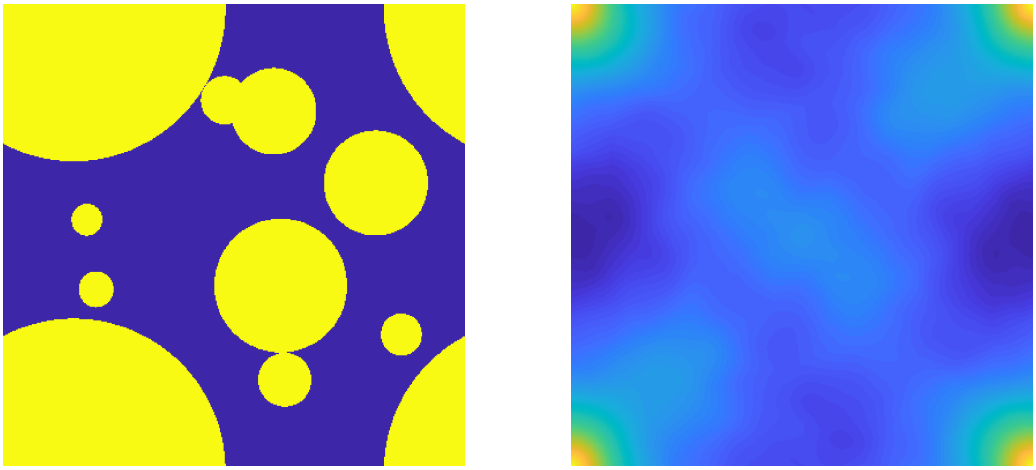
3.4.2 Loss Landscape

Loss landscapes are visualizations that show the loss values across the weight space. They are useful to show in which areas the loss functions have minimas and where different models converge. The visualized loss function is dependent on the network's weight tensor W and its value is computed by measuring the loss on a sample data set that remains consant for all weight tensors. Due to the nature of neural networks, the weight tensors are high-dimensional and thus, the loss function exists in a high dimensional space. Visualizations, however, only really make sense in one or two dimensions. To overcome this problem, interpolations between interesting weight tensors are used. In the one dimensional case this means that two weight tensors W_0 and W_n are chosen from training histories. One represents the left end of the x-axis and the other the right end of the x-axis. Now, to obtain the weight tensors inbetween both points, linear interpolation can be used. For a finite amount of interpolation steps n this yields:

$$W_i = (1 - \frac{i}{n})W_0 + \frac{i}{n}W_n$$

for $i \in \{1, \dots, n\}$. In the two-dimenionsal case, four weight tensors are chosen, one for each corner: top right W_{tr}, top left W_{tl}, bottom right W_{br}, and bottom left W_{bl}. This time, the weight tensors are interpolated bilinearly:

$$W_{ij} = \frac{i}{n} \cdot \frac{j}{n}W_{tr} + (1 - \frac{i}{n}) \cdot \frac{j}{n}W_{tl} + \frac{i}{n} \cdot (1 - \frac{j}{n})W_{br} + (1 - \frac{i}{n}) \cdot (1 - \frac{j}{n})W_{bl}$$



(a) Exemplary microstructure image **(b)** 2-point spatial correlation function

Figure 3.6: Left, an exemplary microstructure image; right, its 2-point spatial correlation function

with i and j ranging from 0 to n, indicating the amount of steps in x and y direction from the bottom left corner. Alternatively, only one weight tensor W_c is picked and put in the center. Using the first two eigenvectors, scaled to match the norm of the last optimization step of that weight tensor, one can then take a step in x or y direction. With this, one can again obtain the grid of weight tensors necessary for the loss landscape:

$$W_{ij} = W_c + (i - \frac{n}{2}) \cdot U + (j - \frac{n}{2}) \cdot V$$

with i and j ranging from 0 to n, indicating the amount of steps in x and y direction from the bottom left corner and U, V being the scaled eigenvectors.

Figure 5.2 shows two loss landscapes generated with this tool. For the one-dimensional loss landscape, the measured loss is plotted over all W_i while for the two-dimensional loss landscape, each W_{ij} is represented by a cell colored according to their loss. In addition, ten isolines are added with their equidistant values ranging from the lowest measured loss to the highest measured loss.

3.5 Training Data

In the last section, it is explained what training data was used and how the feature vector was extracted from the training data. As mentioned in the introduction, the problem domain is about microstructure data and this work is built upon the foundations of [LF19]. Their work also included an annotated training set which will be used extensively here. The data set consists of up to 15,000 randomly generated microstructure images. Those microstructure images are labelled with their manually computed heat conductivity. To convert these microstructure images to feature vectors, a precomputed reduced basis is matrix-multiplied with the two-point correlation function obtained from the images. There are three precomputed reduced bases obtained through training excusively with each inclusion type. The three inclusion types are: circle, rectangle, and mixed. The inclusion

type signifies the shapes used to randomly generate the microstructure image. It was shown that it was best to choose the same basis training type as the inclusion type. More details about the training data and the feature extraction process can be found in their paper.

4 Visual Analytics System

Now that the theoretical foundations are laid, the visual analytics system can be introduced. It aims to facilitate the developement of neural networks by providing the possibility to train and obtain the training history from a configurable training setup and providing interactive visualizations of the training history. This chapter aims to give a quick overview over the visual analytics system and explains how the methods mentioned in Chapter 3 were implemented.

This visual analytics system is split into a front end and a back end. The front end consists of a standard web page. Core to the visualizations is the D3.js library [Bos12], which allows the plotting of data points onto an SVG format. On the back end, there is a web server that listens for JSON requests sent from the web application that executes the corresponding python code. The result is returned as JSON back to the front end for display. In addition, it has access to a computer cluster for computing intensive training and visualizations. In case the computer cluster is needed, the web server uploads the JSON request to the computer cluster instead and runs a batch file on the cluster. Upon completion of the task the cluster copies the result to the web server which consecutively forwards them to the front end.

A screenshot of the user interface can be seen in Figure 4.1. On the left panel, the user can choose the training configuration and see an overview of already loaded training histories where the models can be inspected by selecting a model and clicking on the “View configuration” button. The training configurations of that model will then overwrite the current parameters in the training configuration

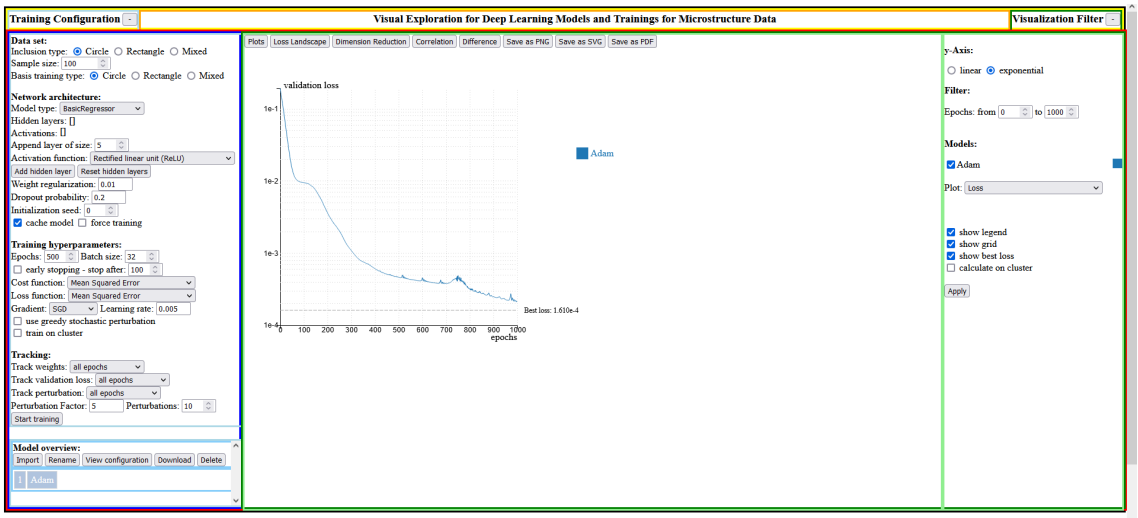


Figure 4.1: Screenshot of the user interface, left is the configuration panel, in the middle is the visualization panel, and on the right is the filter panel

panel. This is useful when the user wants to keep most of the hyperparameters constant and only change a few settings. Apart from viewing the configuration, the user can also import a training history, download a training history, rename a model or delete a model.

Concrete details on how to configure the training is described in the next section. The following section dives into the implementation details of the trainings and how the training history is saved. Afterwards, there is a section on the visualizations that are possible and how they are created. As can be seen in the screenshot, the middle panel is where the visualizations are presented, and the right panel is where the user can interact with them by adding filters and selecting further visualization options.

4.1 Training Configuration

As mentioned in the introduction, the visual analytics system presented here, allows the user to choose the training configuration. This encompasses a plethora of hyperparameters and training options. In the following, each option is briefly explained.

First, the user can select the training data for this training instance. The domain of this visual analytic sytem is microstructure data. The data set from [LF19] contains three types of microstructures: one generated with circle inclusions, one generated with rectangle inclusions, and one generated with mixed inclusions, i.e., circle and rectangle inclusions. For the feature extraction, there are three precomputed reduced bases and they are choosing by selecting the basis training type. Usually this type should be the same type as the inclusions, but other basis training types are possible.

Secondly, the user decides on the network architecture. Two model types are offered: a standard fully connected regressor or one with multiple heads, used for ensemble learning. In both cases, the user proceeds by adding the hidden layers of the network. The input and output are added by default and are not shown in the interface. Depending on the choice of the training data, the input features have 80 (circle), 282 (rectangle), or 260 (mixed) dimensions. The to be predicted heat tensor is a two by two matrix, and thus, there are always four output neurons. For each hidden layer, the amount of neurons and the activation function can be chosen.
Additionally, two hyperparameters for regularization must be picked. The weight decay factor impacts the weight of the regularization term during the optimization process. A weight decay of zero negates that term. Usually this value ranges between 0.001 and 0.01 but technically there is no limit set. The other hyperparameter is the dropout probability. This parameter sets the probability of all neurons in all hidden layers to drop out during a training epoch. Since it is a probability the value must be in $[0, 1]$.
Further, it is possible to set an initialization seed. The initial weights of the model are then seeded based on it. This is very important to compare the effect of different hyperparameters since it is desirable to remove all sources of randomness to achieve reliable experiment results. As such, setting the same initialization seed for the experiments is a necessary feature.
Lastly, there are two caching options for the sake of efficiency. By ticking the cache model checkbox, the training configuration is mapped to the resulting training history. If in the future the same training configuration is detected, the backend can simply return the cached training history instead of retraining a model. If this behavior is not desired on subsequent requests, it is possible to tick the force training checkbox, which forces the backend server to retrain the model even if a cached training history already exists.

Thirdly, the user has to choose the training hyperparameters. There are the basic hyperparameters like the amount of training epochs, the learning rate, and the optimizer to use. Moreover, advanced hyperparameters can be chosen. While in regression, mean squared error is used for both the cost and the loss function other functions can be selected. Also, the user can activate early stopping and specify the threshhold for the amount of allowed consecutive epochs with no improvements in the validation loss.
Furthermore, by selecting to apply greedy stochastic perturbations during the training process, every nth epochs, the weights of the networks are perturbed randomly. This way, stochastic samples in the vicinity are taken and the perturbation that yields the best loss is taken as the starting point for the next optimization step. This newly proposed optimization scheme will be discussed in detail below in Section 5.3.
Finally, the user has the option to train the model on the cluster. Since this visual analytics system is built with domain experts as end users in mind, it is left up to the user to make that call. Training on the cluster can lead to a decrease in training time, but additional overhead is introduced, which makes it slower for smaller trainings. However, if the model is too complex, training on the cluster should also be considered for other reasons, like storage limitations for example.

Lastly, the user can configure the tracking options. During the training process of the model, the training history is recorded and sent to the front end for visualization. The training history consists of several metadata, but most importantly of the neuron weights in the network. The user can select the frequency with which these weights are selected. For example, by choosing to track all epochs, after each training epoch, the weights and biases of every neuron are tracked and saved in the training history, which is a HDF5-file. The advantages of this file format are the efficient writing and reading of multidimensional data in addition to its directory structure which allows storing metadata in the same file. Since that file can get rather big for longer trained and complex networks (up to one gigabyte), it might be preferable to only track every nth epoch. Here, n is calculated as $n = \max(1, \lfloor \frac{\text{epochs}}{500} \rfloor)$ meaning that the amount of epochs tracked is always lower than 1000. The training history of a small network with around a thousand epochs tracked takes about 25 MB of storage. If the user is only interested in the weights of the best epoch, i.e., the epoch with the lowest validation loss, then that can also be selected.
Another metric that is recorded in the training history is the validation loss. Before the training starts, the training data is split into a training set and a validation set. The validation set is used after each training epoch to evaluate the model's performance on unseen data. Unlike the weight data, storing the validation loss is very cheap since it only consists of one float per epoch. Still, if the user prefers to only track the same nth epochs as the weight data, then they can select that option here as well.
The last metric that is stored in the data tracker is the perturbation data. When enabled, after every or after every nth epochs of the training, stochastic samples in the vicinity will be taken. To do that, the tensor of the last optimizer step is taken, scalar-multiplied by the perturbation factor, and the sign in each dimension is selected randomly. This results in a perturbation tensor which is added to the current weight tensor of the network. The amount of perturbations and the perturbation factor can be set by the user.

4.2 Training and Training History

Upon pressing the “Start training” button, the training configuration is sent to the backend where the training is done. Any incoming training request must come with the training configuration attached. Depending on the training configuration the feature vectors from the correct training data must be extracted. The extracted feature vectors are cached in a .npz file for future requests. The training begins by loading the feature vectors and setting up the model according to the provided training configuration. The models are built and trained using the keras [Cho+15] library.
The training of every model creates a training history, i.e., a HDF5-file, which contains every relevant information of that model and its training. While this visual analytics system caches the training history by maintaining a training configuration to training history file map, it is also possible to upload a training history file to the system. This has the same effect as training a model with the specified training configuration of the training history file. In addition to the training configuration, the training history file contains for each tracked epoch:

- the weights and biases collected during the training
- the validation loss
- the percentage of perturbations that resulted in a lower validation loss
- the degree of how the best perturbation improved the loss.

The training history also contains metadata which indicates the training time, the amount of epoch trained or which epoch performed best.

4.3 Visualization

After obtaining the training history of a model, multiple visualizations are offered. To start a visualization, the user can select a visualization mode in the visualization panel. Each mode will be explained in this section. In general, any number of training histories can be visualized at the same time, although there are a few exceptions.

4.3.1 Plots

This mode of visualization is the simplest mode and is preferred to obtain an overall understanding of a model. In the filter panel the user can select a metric which is then plotted over the epochs. These metrics are:

- the validation loss
- the weight values
- the average weight values
- the average weight change
- the percentage of perturbations that yield a better loss
- the best perturbations loss improvement.

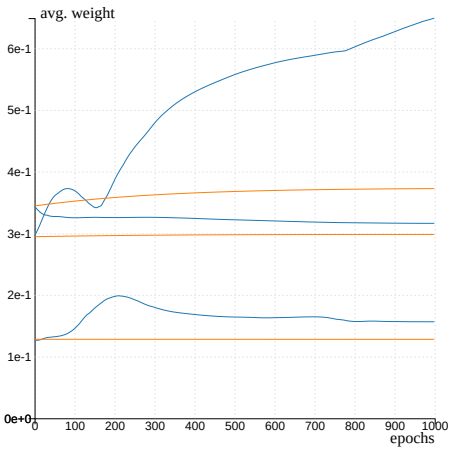


Figure 4.2: An example visualization. The average weight of two networks is plotted over every epoch. For each network, the weights are grouped per layer, thus multiple lines of the same color exist.

The epoch range is costumizable in the filter panel as well as the model selection for this visualization and the color of each model can be freely set. In this visualization, the epochs are plotted over the x-axis while the chosen metric is plotted on the y-axis. There is an option to make the y-axis scale logarithmically, which is practical for discriminating small values. Depending on the selected metrics, more filter options are available. Every weight metric can further be filtered by layer and neuron. For the average weight and the average weight change they can also be grouped, which results in multiple lines per model which can be seen in Figure 4.2. Hovering over a line, will highlight it and a tooltip will appear with more details. Selecting the perturbations also unlocks a smoothing option, since that metric contains high frequency fluctuations. Here, exponential smoothing and a moving average are offered. Lastly, there are a few options to show or hide a legend, a grid or the best loss, as well as an option to calculate the visualization data on the cluster.

4.3.2 Loss Landscape

Three different methods of showing the loss landscape are possible with this tool. The first method shows the loss landscape between two points by plotting the loss over one dimension. To do that, two weight tensors are selected by specifying a model and an epoch. The weights in the epoch of the model's training history are taken for the leftmost point and for the rightmost point, respectively. Note, that both tensors must match in all dimensions, i.e., both models must have the same architecture. Both tensors are then sent to the web server together with the training configuration. There, the model architecture is extracted from the training configuration and a new model is built using that training configuration. To obtain the weight tensors for the loss landscape inbetween both points, each weight is linearly interpolated. The amount of interpolation steps is determined by the user. For each step, the model's weights are set with the weight tensor that are linearly interpolated and then evaluated on the validation set. This results in a sequence of validation losses, that is sent to the front end and visualized on the screen. It is also possible to add more models, which will result in multiple loss landscapes on the same plot. The leftmost weight

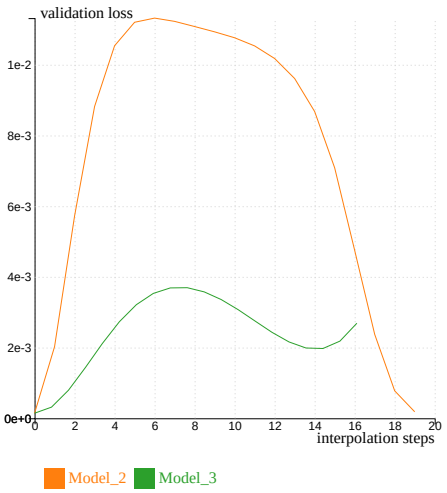
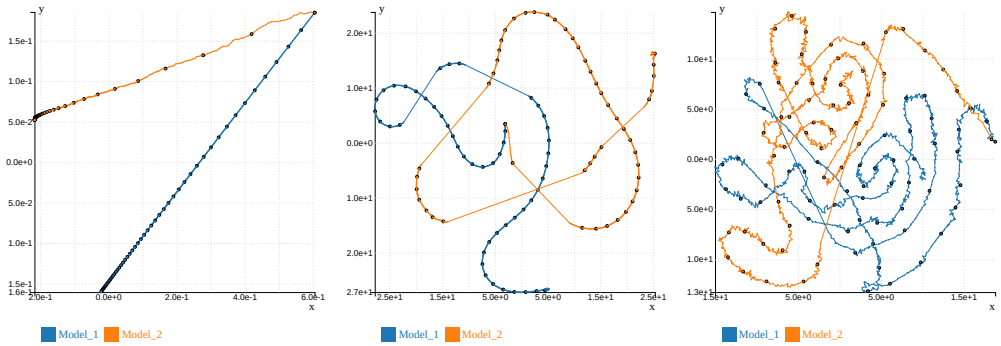


Figure 4.3: An example visualization of multiple one-dimensional loss landscapes. The left most points represents the loss of the weight tensor of model 1 in the last epoch. The rightmost orange point represents the weight tensor of model 2 in the last epoch, while the rightmost green point represents model 3.

tensor is the same for all loss landscapes, and every addition model adds a new rightmost point. That rightmost point is then scaled to represent the distance. An example visualization can be seen in Figure 4.3. Since the weight tensor of the last epoch of model 3 is closer from the weight tensor of the leftmost point than the weight tensor of model 2, they are scaled accordingly, i.e., the end point is closer to the left.

The second method is a two-dimensional loss landscape. Here, four weight tensors are chosen. Each corner is occupied by one weight tensor and the weight tensors in the square are interpolated bilinearly. Thus, the amount of necessary weight configurations that are evaluated have to be squared. The loss value is represented by a color, and isolines can be added for more clarity. The isolines are drawn using the marching squares algorithm described in [Map03]. A recommended way to use the two-dimensional loss landscape is to choose the same epoch for the left corners and the same epoch for the right corners while choosing the same model for the top corners and the same model for the bottom corners. This will show the loss history of the two models on the top border and the bottom border, while the rest of the square visualizes the loss landscape inbetween both paths.

The third option also shows a two-dimensional loss landscape but this time, one weight tensor is chosen to be in the middle of the loss landscape. This visualization aims to show the loss landscape of the surroundings of that particular weight configuration. The surrounding weight tensors are calculated using two base vectors. The first base vector is the last optimization step, and the other is a randomly generated vector that is orthogonal to the first base vector with the same norm. In the two-dimensional projection, the first base vector represents the x-axis and the second base vector represents the y-axis. The step size is chosen such that the base vectors cover half the square, i.e., the leftmost center point represents the network's weights before the last optimizations step, the center point is the chosen weight tensor, and the rightmost center point are the network's weights after adding the optimization step twice.



(a) PCA **(b)** t-SNE **(c)** UMAP

Figure 4.4: Three different visualizations after applying different dimensionality techniques on the same models. Both networks start with the same initial weights but diverge from each other during the training process.

4.3.3 Dimensionality Reduction

In this visualization mode, the user can select different dimensionality reduction algorithms to visualize the weights of the chosen model(s). Just like in the plots mode, it is possible to filter certain epochs or layers and neurons or to group the results by model or neuron. The goal is to reduce the weight tensor of each model for every epoch to just two dimensions. That way each epoch of each model can be drawn as a datapoint on a coordinate system. Close points in the projection mean that the euclidean distance between the high dimensional weight tensors that they represent is also small. If the distance between the datapoints of a model keeps diminishing with advancing epochs, then this can be a sign that the model converges. If different models converge to the same point in the projection, then their weight tensor also converges, suggesting that these networks arrived in the same loss minima. If they, however, converge in different points, that can be interpreted as getting stuck in different local loss minimas.

Figure 4.4 shows three different visualizations that resulted from different dimensionality reduction techniques. The two depicted networks have the same weight tensor in the beginning of the training. During training, the weights diverge from each other. In the visualization, each weight tensor during an epoch is represented by a point and is connected to the point of the preceding and succeeding epoch. Every nth epoch is represented by a larger point and a black outline. This makes it possible to quickly grasp the point density of each line. In all three visualization, the starting point of both models overlap before quickly diverging. This is seen most cleanly with PCA, where the lines do not curve much.

All dimensionality reduction algorithms were implemented in JavaScript libraries. The SVD decomposition was done with a library [Sal] which implemented the algorithm described in [GR71]. t-SNE is implemented using the Druid-JS library [Cut] and can be configurated with two parameters, the perplexity parameter which determines the amount of neighbors used to learn the probability distribution P and the learning parameter epsilon of the gradient descent. UMAP was also in the DruidJS library.

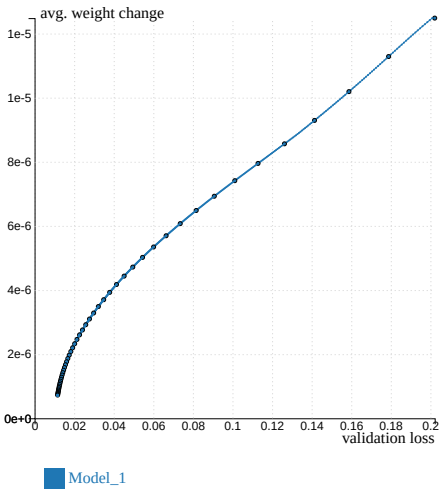
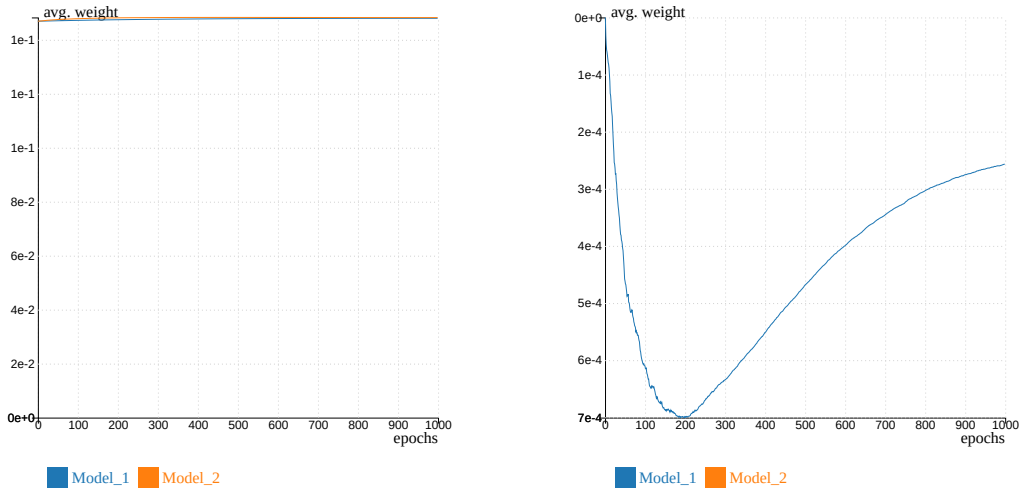


Figure 4.5: Using the correlation mode, two different metrics can be plotted against each other. Here, the average weight change and the validation loss are shown to be correlated.



(a) Using plot mode

(b) Using difference mode

Figure 4.6: It can be hard to discern the difference of the average weight of both models in the plot mode. Instead, using the difference mode, the exact difference can be plotted.

4.3.4 Correlation and Difference

The next visualization mode offers a method to find correlations between different metrics. Here, the user can choose which metric to display on the x-axis and y-axis. The chosen datapoints are then plotted accordingly. The resulting visualizations allow the user to easily spot negative or positve correlations. As an example, Figure 4.5 depicts a visualization that showcased a positive correlation between the average weight change and the validation loss of the model.

Sometimes it is helpful to look at the difference between two models and compare them closely. In this visualization mode exactly two different models need to be chosen and the absolute difference between their values in the selected metric are plotted over all chosen epochs. This can be helpful to inspect minor differences between two models that would not be apparent in the standard plot as can be seen in Figure 4.6.

5 Experiments and Results

In this chapter, the results and the knowledge gained from various experiments are presented. The experiments in this work are mostly explorative in nature since their goal is to gain additional insight into the used models rather than to prove a theorem. The visual analytics system will execute the experiments and visualize the results. Often, the results of an experiment lead to the design of the next experiment. Each experiment will increase the understanding of how to ameliorate the training configurations in order to improve the model's performance. In addition, there are also experiments that test whether known techniques that improve a model's performance would also work in this domain.

To ensure reproducibility, all outside factors that can influence the results must be eliminated. In particular, sources of randomness will distort the experiments, making it hard to say whether a model performed better because the independant variables were changed or because of stochastic variance. As such, the training process should be as deterministic as possible, meaning all sources of randomness should be removed or at least seeded with the same seed for all experiments. The main offender for this are the initial weights. When a model is first initialized every weight is set to random weight using the Glorot uniform initializer. The initializer can take an integer as seed and depending on the seed the result can be very different, as Figure 5.1 shows. The models have very different performances even after a thousand training epochs. The reason for this is that depending on the seed each model converges into a different loss minima.

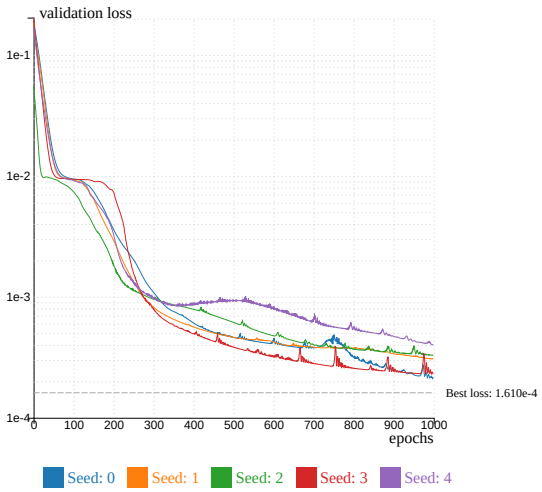
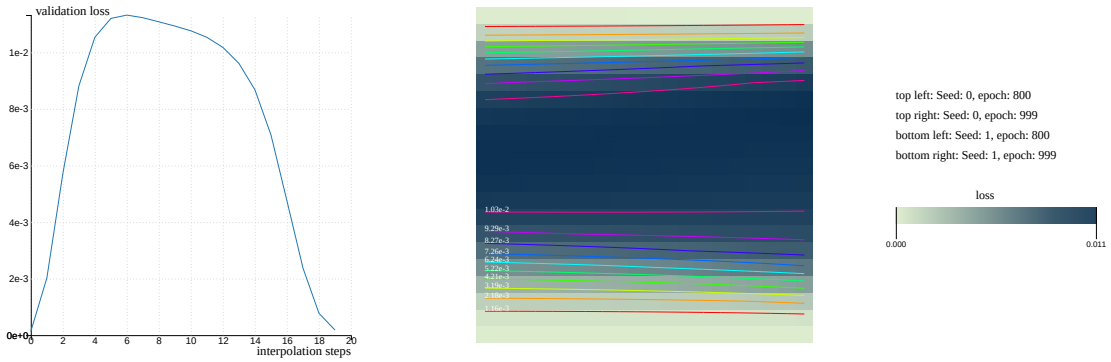


Figure 5.1: The validation loss per each training epoch for two networks with the same architecture ((10, 6), softmax, Adam) but different seeds for the initial weights. Depending on the initial weights the networks can result in different performance.



(a) One-dimensional loss landscape **(b)** Two-dimensional loss landscape

Figure 5.2: The loss landscape between two networks with the same architecture ((10, 6), softmax, Adam) but different seeds for the initial weights. Left, a one-dimensional interpolation between the neurons of both networks in the last epoch. Right, a two-dimensional interpolation using four different network's states as corners. The loss landscape between both networks shows a loss ridge inbetween.

Figure 5.2 visualizes this by showing the loss landscape between two models with different initializations. If both models converge to the same loss minima, the interpolated weight configurations would also lead to a similar loss value. However, as the figure shows, the loss landscape inbetween both models shows a loss hill, indicating that each model converges to a local loss minima. The two-dimensional loss landscape illustrates that this divergence is consistent in the last 200 epochs. To keep the results comparable, a default set of training configurations is used for every experiment. If not mentioned explicitely otherwise, there is no weight regularization nor early stopping and the models are trained on the first thousand entries of the circle dataset. The loss metric will always be mean squared error (MSE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

with n being the size of the validation set, Y_i the true value, and $\hat{Y}_i$ the predicted value.

Now, that each training can be run deterministically, it is possible to compare different network setups. The experiments in the first section are about the architecture of the models. Their results provide the default network architecture for the following experiments. With this, the experiments in the next section explore how different weight regularization techniques affect the training process, i.e., how they influence the loss and the weights of the network. Then, in the third section, the greedy stochastic perturbation technique is presented and its effectiveness is shown in several experiments. Next, a brief overview of the ensemble learning experiments is given and lastly some more insights are unveiled.

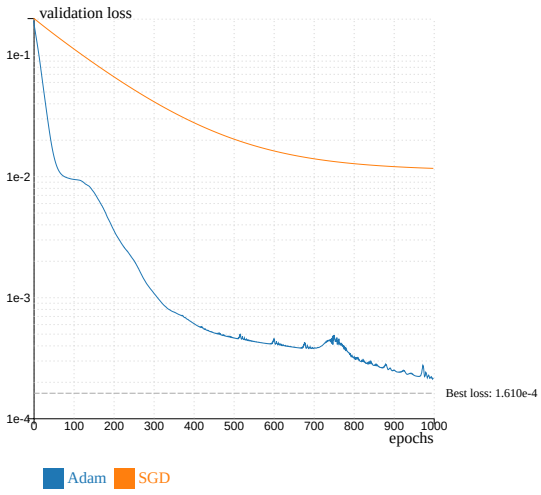


Figure 5.3: The validation loss per each training epoch for two networks with the same architecture ((10, 6), softmax,) but different optimizers. Adam greatly outperforms SGD in this visualization.

5.1 Architecture and Initialization

When designing a neural network, one of the unavoidable issues is to decide on a network architecture. A neural network can take on many different forms. It can differ in the number of neurons, the number of layers, how the neurons are connected to each other or which activation functions are used. In the field of deep learning, there are a plethora of different possible architectures from simple multilayer perceptrons (MLP) to complex Long short-term memory (LSTM) networks, each better or worse depending on the problem. Bigger and more complex architectures can learn more complex structures but also require more training while smaller and simple networks train faster and may perform well enough. To find the most fitting architecture, an expert might rely on their experience, or intuition or educated guesses. However, ultimately it often comes down to trying out different architectures and sizes, or using some algorithms for hyperparameter optimization, such as grid search, and picking the best one. In this work, the problem can be solved by simple regression. Thus, only MLPs are considered and because the regression target is the heat conduction tensor this already dictates the number of output neurons and since the feature vector is also given, the number of input neurons is also prescribed. Ergo, the *only* parameters left to decide is the number of hidden layers, and the amount of neurons and the type of activation function of each layer. The experiments in this sections will explore the different possibilities and reveal how they perform against each other.

All hyperparameters will be the same, except for the network architecture, which is changed between training. The optimizer was chosen to be Adam, since it is the optimizer that consistently performed the best. In the beginning SGD was used, but it was soon discovered that Adam greatly outperforms SGD in all tested architectures. An example of that can be seen in Figure 5.3 where after a thousand epochs the model with SGD has a validation loss of almost a factor of one hundred higher.

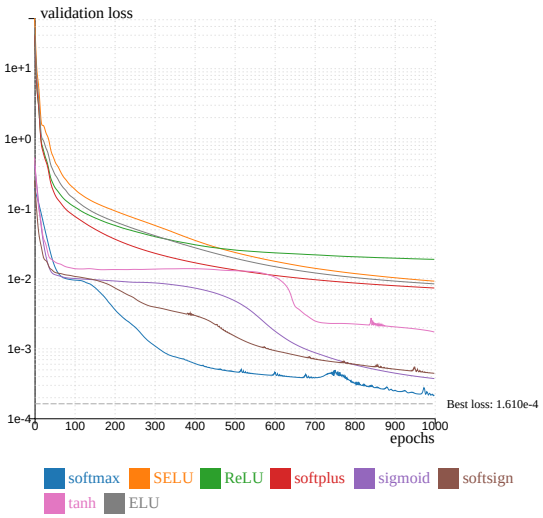
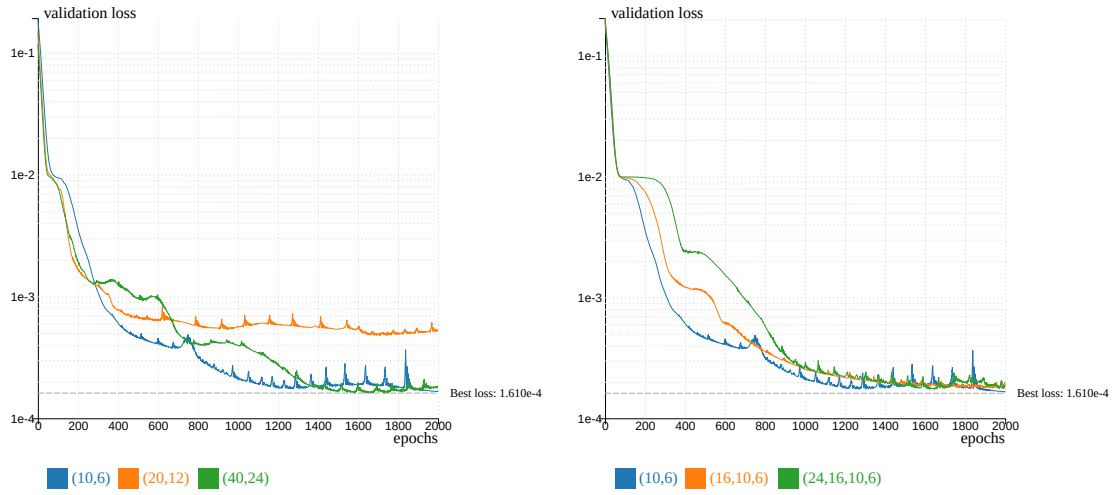


Figure 5.4: The validation loss per each training epoch for two networks with the same architecture ((10, 6), Adam) but with different activation functions.

With that out of the way, there are still a multitude of hyperparameters to experiment on. The amount of hidden layers, the number of neurons per layer and the activation function of each layer will influence the model's performance. Testing the many different possible permutations seem unattainable, so in this work a few simplifications are assumed. First, the activation function is the same for all hidden layers. It is possible to choose different activation functions for each individual layer, but for the experiments here they will remain identical for all layers. Secondly, the amount of neurons per layer is decreasing with progressing layers. Recall, that the output layer only has four neurons and the input feature vectors have at least eighty dimensions. Thus, it makes sense to gradually decrease the number of neurons in each layer. The third and last simplification is that each hyperparameter is independant from the others. This is obviously not the case in reality, an activation function might outperform in deeper networks and underperform in simplier network. But looking for the best hyperparameters is outside the scope of this work. This is merely a demonstration of what this tool is capable of. In the following, a network configuration is shortened to a tuple, an activation function, and an optimizer. For example, ((10, 6), softmax, Adam) denotes a neural network with two hidden layer, the first hidden layer with ten neurons and the second hidden layer with six neurons, with softmax as activation function in both layers and Adam as optimizer for the gradient.

First, an experiment on the different possible activation functions is undertaken. Figure 5.4 shows the validation loss during training of different activation functions. All models use the ((10, 6), Adam) architecture. The results are quite spread out, ranging from $2 \cdot 10^{-4}$ to 10^{-2} after a thousand epochs. The best performer is the softmax functions with a loss under $2 \cdot 10^{-4}$ after the same amount of training, a loss more than fifty times smaller than the worst performer. Interestingly, ReLU and similar activation functions like SELU, ELU, and softplus perform the worst, while softmax, softsign, and sigmoid perform the best. The main difference between both groups is that the first group has an infinite endpoint in their range, while the range of the second group does not exceed 1.



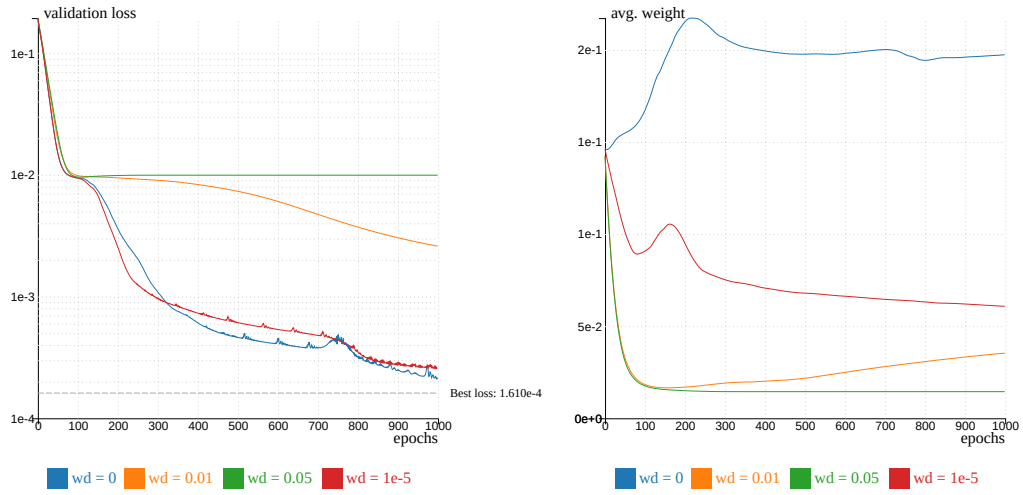
(a) Different amounts of neurons per layer

(b) Different amounts of layers

Figure 5.5: The validation loss per each training epoch for networks with different network complexity.

Using the assumption that softmax also works best for models of different sizes, it is now possible to experiment with different amount of neurons. Considering that bigger networks might require more time to converge, the amount of epochs trained is also increased. Figure 5.5a depicts the performance of different networks that vary in the amounts of neurons per layer. The biggest and smallest network perform about equal, while the middle network has worse performance. The smallest network also converges faster, i.e., less training epochs are needed and is thus preferred. In addition, due to the small amount of weights, its training is also faster and it is less prone to overfitting.
It seems that smaller networks perform as good or even better than their larger counterparts. To confirm that this also applies to the depth of a neural network, different models with varying amounts of hidden layers are compared. The results of this comparison can be seen in Figure 5.5b. Each model achieve a similar validation loss after 2000 epochs with the smallest one having a small lead. This confirms the hypothesis, that smaller networks can perform equally good or better than bigger networks on this dataset.

In the following experiments, if not mentioned otherwise, the ((10,6), softmax, Adam) architecture is chosen, since it is the best performing architecture and due to its small size it reduces the time needed for training. Of course, it is a naive assumption to say that this particular architecture will still work best with other hyperparameters but the alternative would be to test all kinds of architectures every time a hyperparameter is changed, which would be very time consuming and outside the scope of this work.



(a) The validation loss per each training epoch

(b) The average weight per each training epoch

Figure 5.6: Comparison of four networks with the same architecture ((10, 6), softmax, Adam) but different weight decay (wd). Adding weight decay seems to worsen this network's performance.

5.2 Weight Regularization

Now that the optimal network architecture was found, it is possible to look into other ways to improve a model's predictions. A common method is to introduce some kind of regularization to avoid overfitting, reduces the variance of the model and ultimately improve the prediction of the regressor. The experiments in this section aim to show how these effects manifest during the training. The challenge here is that there is no singular metric that describes whether a model is underfitted or overfitted and thus it is hard to measure the effects of weight regularizers. In general, if the validation loss stops decreasing and starts increasing, this can be a sign for overfitting. In addition, the weights themselves can be an indicator. Growing weights tend to be a sign of overfitting, so observing the average weights can be used to detect that.
This tool offers two levers for weight regularization. One can define a factor that will directly impact the weight regularization term that is added to the loss function. That term is the sum of the L_2-Norm of all weights in the network. The other option is to set a dropout probability, which will dictate the frequency of which a neuron in the networks drops out during an epoch during the training.

Starting with the first regularization method, Figure 5.6a shows a visualization of the validation loss. The three depicted models show networks with varying weight regularization factors, namely 0, 0.01, and 0.05. A weight decay of zero effectively ignores the weight regularization term. As the visualization shows, adding a factor significantly worsens the network's performance in these cases. While the model with the lower factor slows down the validation loss decrease, the model with the higher factor seemingly stops all improvements.
Further inspection of the average model weights resulted in the visualization in Figure 5.6b. The plot clearly shows that the models with a weight regularizing factor greater than zero immediately

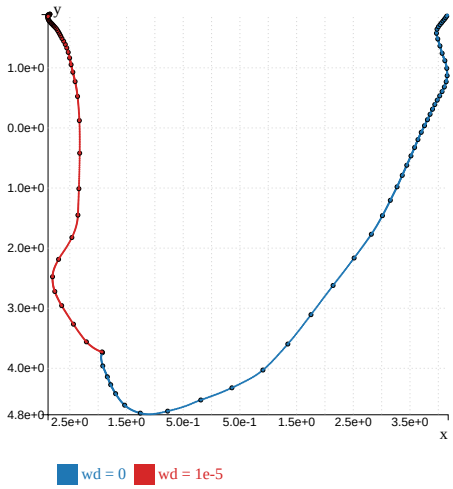
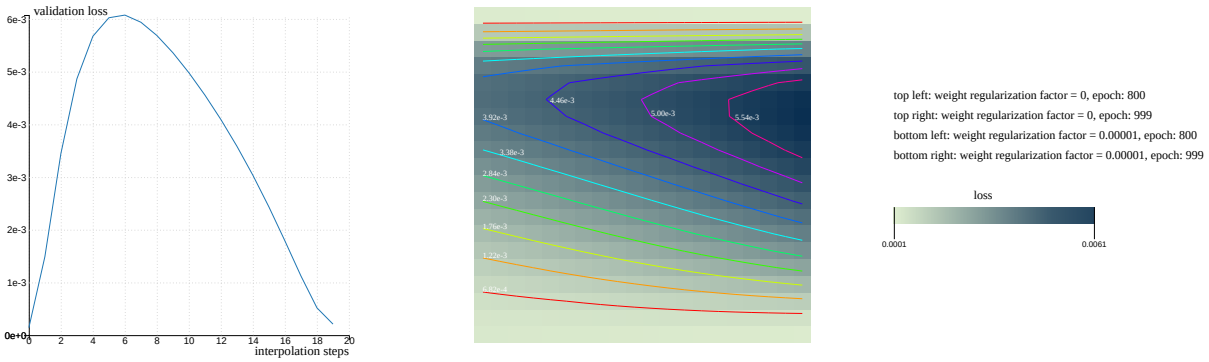


Figure 5.7: The weights of two networks with the same architecture ((10, 6), softmax, Adam) but different weight decay. The weights were reduced to two dimensions using PCA. In the starting epochs their weights are the same and they diverge during the training process.

push the average weights down. While this direction was expected, the speed of the decrease is somewhat unexpected. The reason for that is that the regularizing term outweights the loss term in the calculation of the loss function. The training losses are in the realm of 10^{-3} while the average weights start at around 10^{-1}. Even with factors in the hundredths, this will significantly weigh on the loss gradient. Additional experiments show that choosing a factor of 10^{-5} results in a comparatively similar loss as using no weight regularization at all. Again, the average weight of the model with no weight regularization starts growing immediately while the average weight if the other model decreases, albeit not at the extreme rate as before. This makes itself noticeable in the validation loss, keeping its performance competitive.

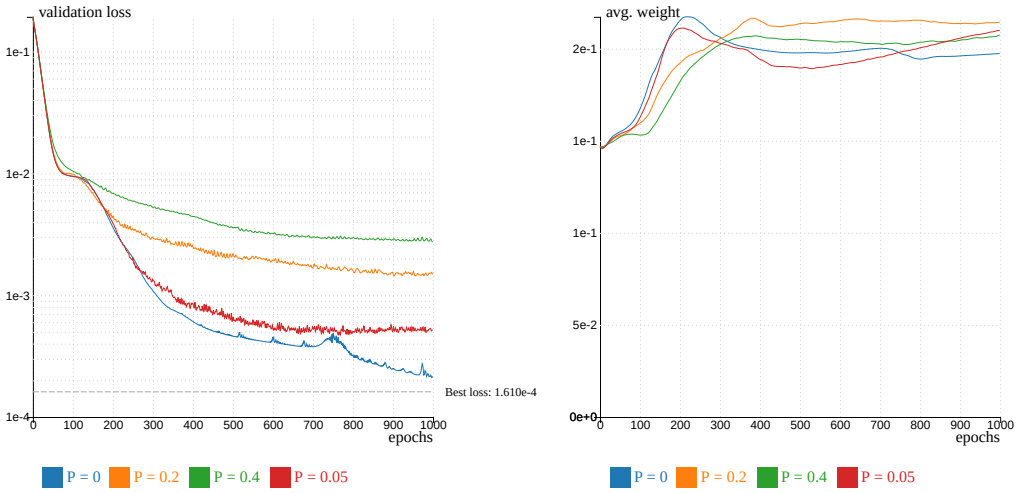
One more way of showing the divergence in the weights is by using the dimensionality reduction tools. Figure 5.7 shows the resulting projection after using PCA on all the model weights during all epochs. Each circle represents all weights of a model in one epoch, the circles are so close to each other, that they look like a solid line in the visualization. For clarity, every nth epoch the circle is increased in size to better show the weight convergence. With growing time, each circle is closer to the previous circle. In the beginning, the datapoints of both model start at the same place, since the initialization seed is identical. Immediately after, however, they quickly diverge. Another interesting result is, that both models converge to different local minima. This can be proven by using the same visualization techniques as used for the initialization seed. Figure 5.8 shows the loss landscapes between both models and since a loss ridge can be seen inbetween the path of both models, one can say that each model converges in their own local minima.

For dropout the same experiments were performed and the results are similar. Figure 5.9a shows the validation losses for four different models. The model with no dropout has the lowest loss and with increasing dropout probability the validation losses also increase.



(a) One-dimensional loss landscape **(b)** Two-dimensional loss landscape

Figure 5.8: The loss landscape between two networks with the same architecture ((10, 6), softmax, Adam) but different weight decay. Left, a one-dimensional interpolation between the neurons of both networks in the last epoch. Right, a two-dimensional interpolation using four different network's states as corners.



(a) The validation loss per each training epoch **(b)** The average weight per each training epoch

Figure 5.9: Comparison of four networks with the same architecture ((10, 6), softmax, Adam) but different dropout probability. Increasing the probability seems to worsen this network's performance.

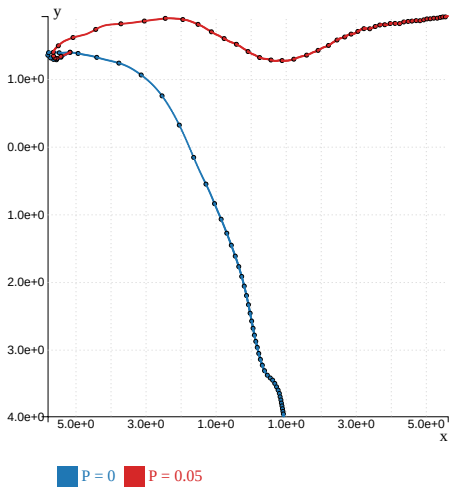
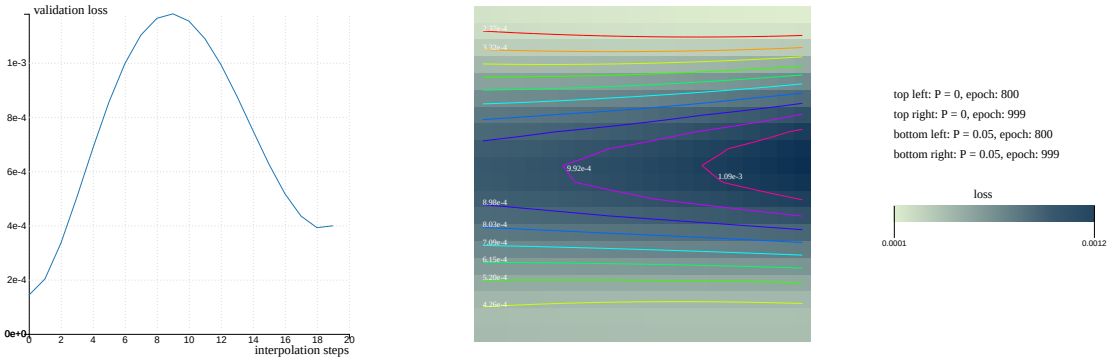


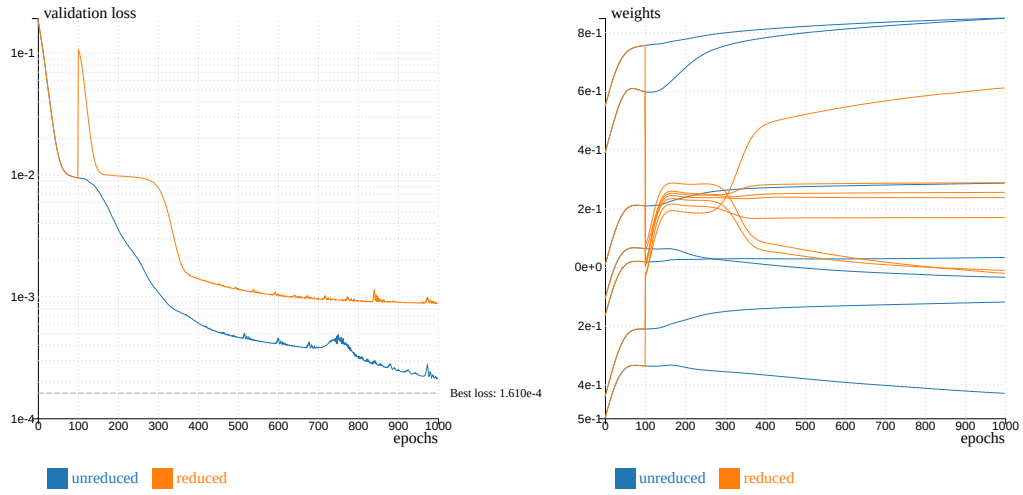
Figure 5.10: The weights of two networks with the same architecture ((10, 6), softmax, Adam) but different dropout probability. The weights were reduced to two dimensions using PCA. In the starting epochs their weights are the same and they diverge during the training process.



(a) One-dimensional loss landscape

(b) Two-dimensional loss landscape

Figure 5.11: The loss landscape between two networks with the same architecture ((10, 6), softmax, Adam) but different dropout probability. Left, a one-dimensional interpolation between the neurons of both networks in the last epoch. Right, a two-dimensional interpolation using four different network's states as corners.

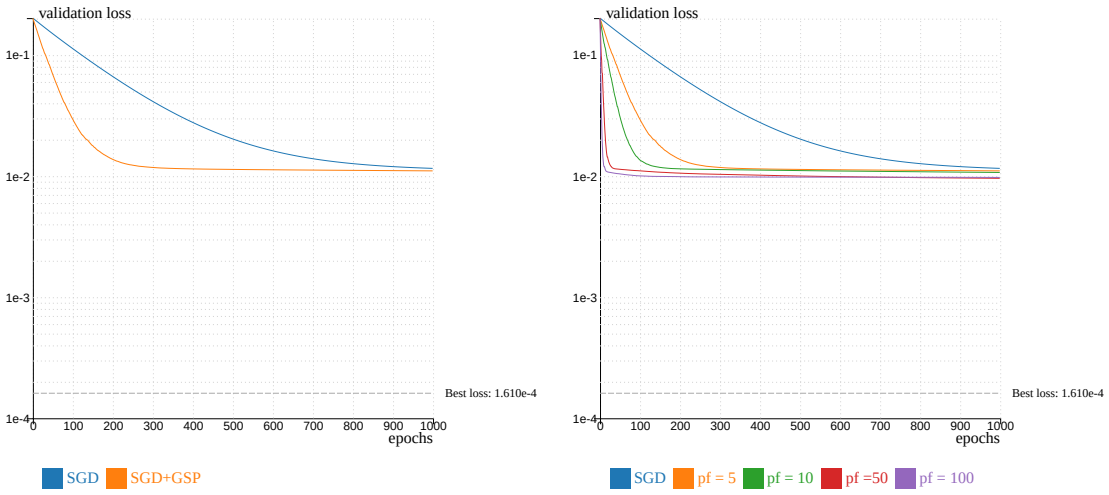


(a) The validation loss per each training epoch **(b)** An output neuron's weight per each training epoch

Figure 5.12: Comparison between two networks ((10, 6), softmax, Adam). The weights of the second network were manually reduced by multiplying them by 0.1 at the 100th epoch. Instead of returning to their original values, most weight stay around their new values.

Looking at the average weights of each model in Figure 5.9b it can also be seen that the weight reducing effects of weight regularization do not occur with dropout. Instead, each model's average weights fluctuate around the same values. Using a combination of dimensionality reduction and loss landscape visualizations, Figure 5.10 and Figure 5.11, respectively, it is possible to see that dropout influences the network in such a way that it pushes the weights and the path in the loss landscape into a different local minima that is worse than the original one. In conclusion, dropout is not particular helpful for the models in this problem space, even though it usually adds improvements.

After confirming that neither weight regularization nor dropout improved the model's performance, a more radical experiment was proposed. In essence, regularization punishes large values in weights and thus prevents overfitting. That weight reduction could be obeserved with weight regularization and during all training epochs there was incentive to keep the weights small. In the next experiment no regularization method was used, however the weights were manually reduced exactly once at the hundredth epoch by multiplying them by 0.1. Figure 5.12a shows the effect on the validation loss of this intrusion. There is a spike right after the reduction but the loss quickly decreases again. It does however never reach the same level as the unreduced model instance. The more interesting results can be seen in Figure 5.12b, where the individual weights are shown. For clarity, only the weights of the output layer are shown, but all other weights show a similar result. In the first hundred epochs both instances weights develop equally. After the reduction however, the individual weights do not climb back to their previous values, instead they settle for totally different values.



(a) Comparison of standard SGD and SGD with GSP **(b)** Comparison of different perutbation factors

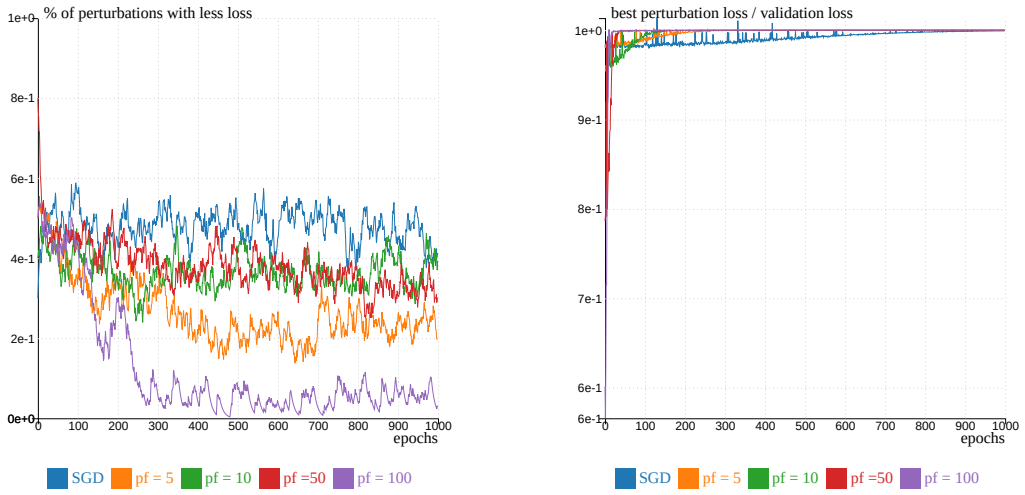
Figure 5.13: The validation loss per each training epoch for two networks with the same architecture ((10, 6), softmax) but different optimization processes. The novel greedy stochastic perturbations approach results in faster improvements and an overall better loss.

5.3 Greedy Stochastic Perturbation

This section is all about greedy stochastic perturbation (GSP) and how this technique can improve the training process of the models. Recall that this technique collects samples around the current weight tensor after each training epoch using the last optimization step. It then picks the sample that best improves the validation loss as the new weight tensor, or keeps the current weights if no sample was better. Each sample is obtained by adding a perturbation to the current weight tensor. The perturbation is calculated by using the last optimization step, randomizing the sign in each dimension and multiplying it with a perturbation factor which is a hyperparameter selected in the training configuration. The other relevant hyperparameter is the number of samples to take in each training epoch. The effect of this technique is best demonstrated with the SGD optimizer.

Figure 5.13a shows the validation losses for the stand-alone SGD optimizer and for the SGD optimizer improved with GSP. Not only does the latter achieve a better performance, it also reduces the validation loss alot earlier. This effect can be boosted by increasing the perturbation factor as Figure 5.13b shows. With an increasing factor, the accelerated validation loss decrease is more pronounced. After that phase, however, no real improvements are made and the validation loss quickly converges.

To inspect why that is the case, one can take a look at the percentage of perturbations that result in a better validation loss. Since the weights of the network converge to their optimal values, finding better samples in the vicinity might get harder, especially if the perturbation factor is too high. Figure 5.14a depicts exactly that. For the model with no GSP the metrics can still be measured, the samples are also taken in every epoch but never overtaken. In the early epochs about half of the samples lead to a better result but after that depending on the perturbation factor that percentage drops down. An obvious outlier is the model with a perturbation factor of one hundred. The



(a) The percentage of perturbations with less loss per each training epoch

(b) The ratio of the best perturbation loss to the validation loss of the SGD step per each training epoch

Figure 5.14: Comparison of for multiple networks with the same architecture ((10, 6), softmax) optimized using SGD assisted with the novel greedy stochastic perturbations approach but with different perturbation factors. For reference a network with the same architecture optimized with standard SGD is depicted. The results are smoothed using simple exponential smoothing ($\alpha = 0.1$). Networks with higher a perturbation factor have perturbations with much lower losses in the early epochs but struggle to find significant improvements in the later epochs.

samples are way to far from the current weight tensor, which is located very near to the optima, to find any good perturbations. Interestingly, the next lowest percentage belongs to the model with a perturbation factor of only five. This can mean that samples that are too close also do not have a high chance of being lower in the lost landscape. Still, for all variants, a handful of samples can be found in each epoch. In general, this particular experiment has many random elements, so much that the results had to be smoothed in the visualization, lest the high-frequency-like results do not obstruct each other.

To explain why the validation losses stop decreasing after the initial accelerated decrease albeit better samples are found frequently in even the later epochs the improvement of each sample is shown relative to the validation loss of the unperturbed setting. To this end Figure 5.14b, shows the ratio of the loss of the best sample divided by the loss of the current weight tensor. For the majority, this value is smaller than one, which means that the sampled loss was smaller. Only on afew occasions does this value exceed one, exactly when no better sample was found. In the very early epochs, the best samples validation losses are about 5-20% lower than the unperturbed validation losses. Over time, the best perturbation loss gets relatively less and less, resulting in only miniscule improvements in the later epochs. The reason for that is probably that the models quickly arrive at their optima and no improvemtents are possible anymore. Interestingly though, unlike the other models, the model using the standard SGD optimizer can still find better samples even in advanced epochs. Perhaps, by using GSP these possible improvements are frontloaded in the beginning of the training process, which makes GSP an interesting tool to speed up trainings.

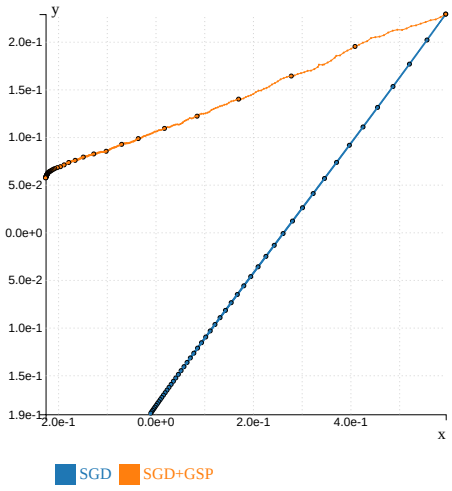
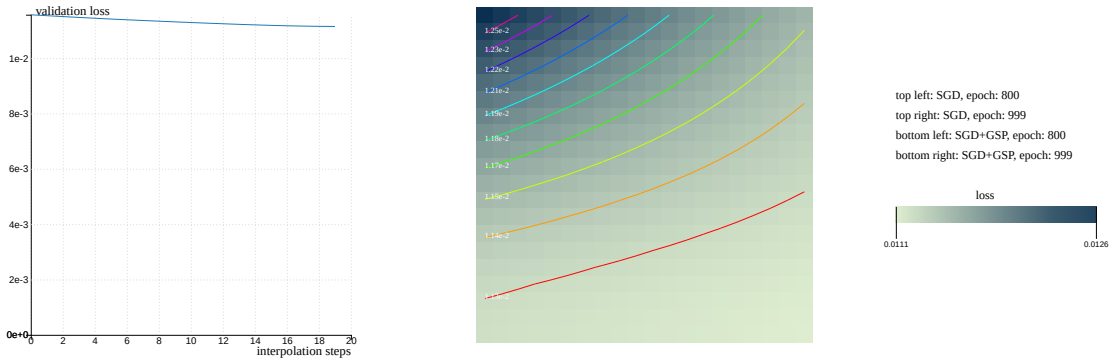


Figure 5.15: The weights of two networks with the same architecture ((10, 6), softmax) but different optimization processes. The first network uses the standard SGD, while the second network uses SGD assisted with the novel greedy stochastic perturbations approach. The weights were reduced to two dimensions using PCA. In the starting epochs their weights are the same and they diverge during the training process.



(a) One-dimensional loss landscape **(b)** Two-dimensional loss landscape

Figure 5.16: The loss landscape between two networks with the same architecture ((10, 6), softmax) but different optimization processes. Left, a one-dimensional interpolation between the neurons of both networks in the last epoch. Right, a two-dimensional interpolation using four different network's states as corners. Both networks converge into the same loss trench.

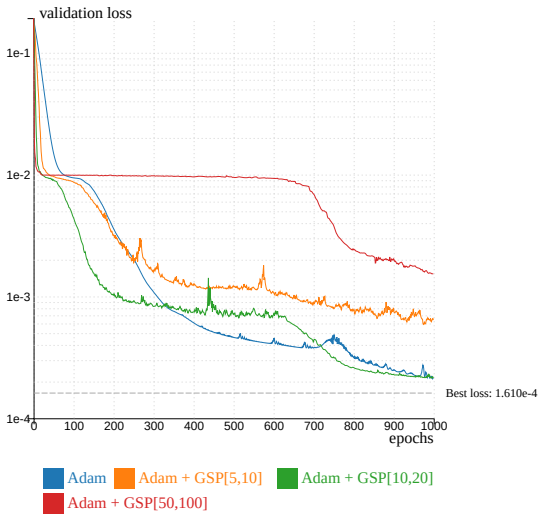


Figure 5.17: The validation loss per each training epoch for four networks with the same architecture ((10, 6), softmax) but different optimization processes. The first network uses the standard Adam, while the other networks use Adam assisted with the novel greedy stochastic perturbations approach. The numbers in the squared brackets [n,m] indicate the perturbation factor n and the number of samples taken m. Depending on the combination, the results may vary vastly.

The question remains, whether that speedup leads to the same weights as without GSP, or whether it leads to entirely different weights. Figure 5.15 visualizes the weights of both networks in two dimensions. Starting at the same point, the weight tensors of each model move to a similar direction, but ultimately still end at different places. Looking at the loss landscapes in Figure 5.16 shows that interpolating between both weight tensors lead to similar losses, suggesting that both models converge in the same local minima. Since the weight are quite different however, it may be more accurate to say that they are in the same loss ditch.

Taking these insights into consideration, it is now possible to experiment using the Adam optimizer. Unlike with SGD, adding GSP does not always seem to improve the network. By experimenting with different perturbation factors and increasing the number of samples accordingly, a competetive setting can be found. Figure 5.17 compares the stand-alone Adam optimizer with various combinations of GSP. Using samples that are too close or too far will worsen the performance, while choosing the samples that are just the right distance can show an improvement. Again, it can be seen that using GSP accelerates the validation loss decreases, but Adam is able to match the validation loss in the later stages. Still, even though in the end no real improvement is made, GSP remains an interesting option for potentially speeding up trainings.

Lastly, a brief look at the loss landscape in Figure 5.19a and the dimensionality reduction visualizations in Figure 5.19b reveal that unlike SGD, Adam and its GSP boosted variant do not converge in the same loss ditch, suggesting that GSP affects each optimizer differently.

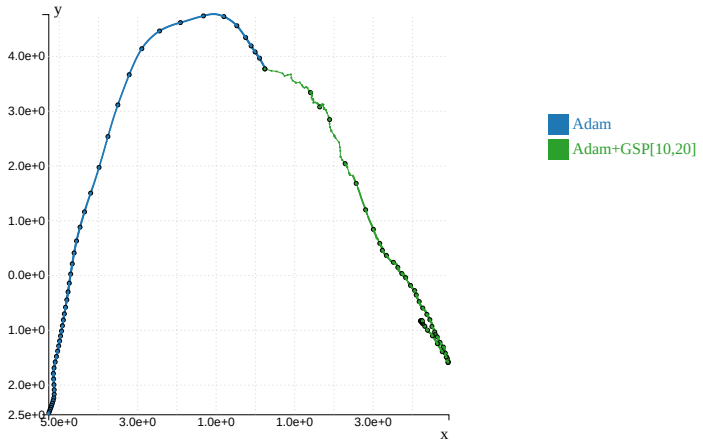
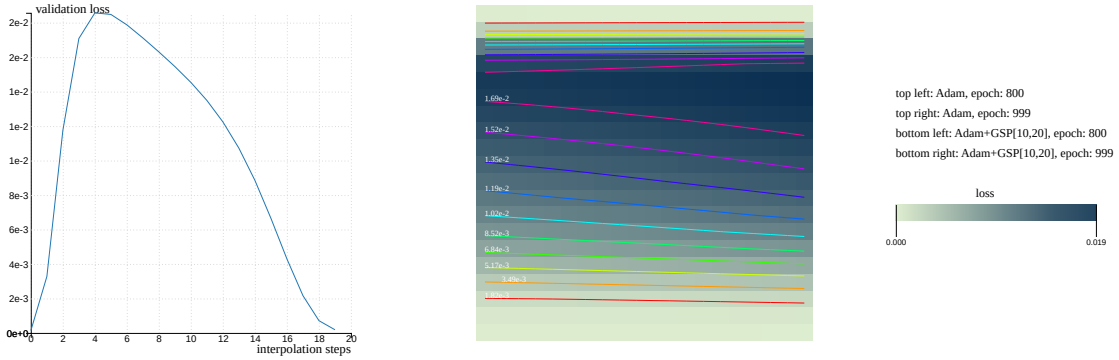
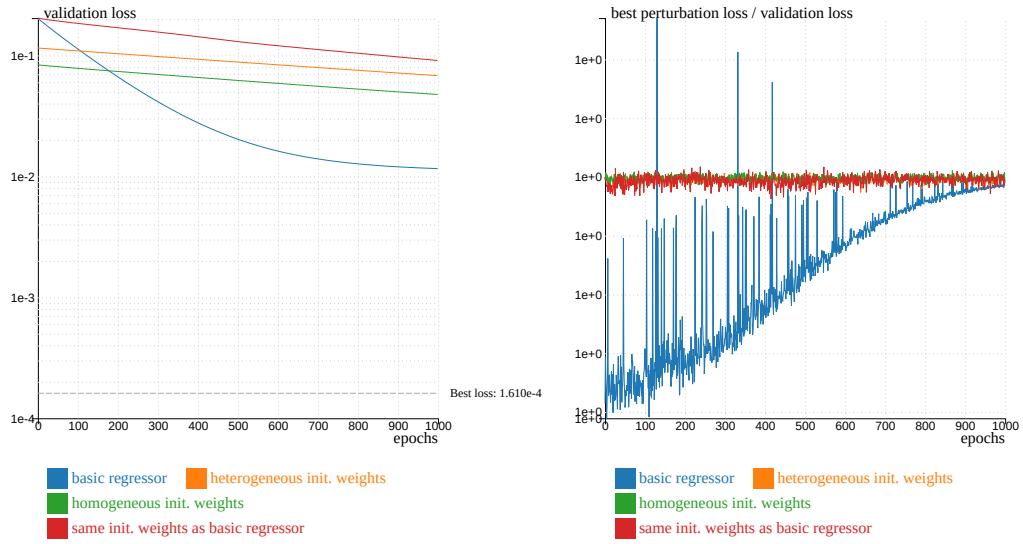


Figure 5.18: The weights of two networks with the same architecture ((10, 6), softmax) but different optimization processes. The first network uses the standard Adam, while the second network uses Adam assisted with the novel greedy stochastic perturbations approach. The weights were reduced to two dimensions using PCA. In the starting epochs their weights are the same and they diverge during the training process.



(a) One-dimensional loss landscape **(b)** Two-dimensional loss landscape

Figure 5.19: The loss landscape between two networks with the same architecture ((10, 6), softmax) but different optimization processes. Left, a one-dimensional interpolation between the neurons of both networks in the last epoch. Right, a two-dimensional interpolation using four different network's states as corners. Both network converge into different loss trenches.



(a) The validation loss per each training epoch

(b) The ratio of the best perturbation loss to the validation loss of the SGD step per each training epoch

Figure 5.20: Comparison multiple networks with the same architecture (10, 6, tenfold repeated output) with different weight initializations. A basic regressor of the same architecture is included for reference.

5.4 Ensemble Learning

Yet another way to improve a model's performance is to employ multiple instances of models and using the results of each member of the ensemble to form a prediction. This apporach is also called Ensemble Learning. Unlike classical Ensemble Learning practices, a different process is employed. Instead of duplicating the whole network, only the heads, i.e., the output neurons are duplicated. This results in a multi-headed architecture, with each head acting as a member of the ensemble. In all the experiments, each output neuron is duplicated ten times. The final prediction is then made by averaging the predictions of each of the ten heads.

In the first experiment, different procedures for weight initialization are compared. Each head can be initialized with different weights or the same weights. Figure 5.20a depicts the results. For reference, a basic regressor, i.e., a single-headed network, is shown. The network with heterogeneous initial weights is outperformed by the network with homogeneous initial weights. However, when the homogeneous initial weights are set to be equal to the initial weights of the basic regressor the performance decreases. This again shows that the network's performance is very sensitive to its initial weights. It is important to notice, that when the initial weights for all heads are identical, their weights stay identical during the whole training process, due to the removel of all stochastic elements during the training process. As such only by initializing each head heterogeneously can ensmeble learning be simulated. In any case, all ensemble networks perform significantly worse then the basic regressor.

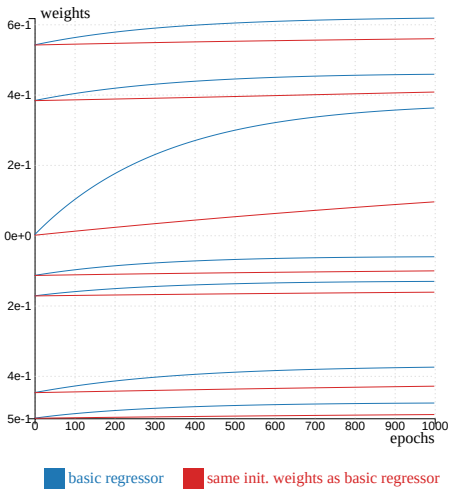


Figure 5.21: The weights per each training epoch for the basic regressor network and the ensemble network that was initiated with the same seed. Only the weights of neuron 0 in the output layer ist shown. The weights are identical in the first epoch but diverge during the training.

A reason for that can be seen in Figure 5.20b. There, a visualization of the quality of nearby perturbations can be seen. In case of the basic regressor, it shows that in the early epochs one can achieve better losses by stochastically perturbing the weight tensor of the network. With increasing epochs the improvements that could be made with GSP decrease as expected. For the ensemble networks however, no real improvements would be possible by using GSP even in the early epochs. This suggests that these networks have trouble finding a way to improve.

5.5 Early Stopping

Up until now, most trainings go up to a thousand epochs. There are many reasons for that: most interesting training developements occur in the early epochs; the models would overfit if the training would go longer and most models converge around that epoch and do not improve significantly afterwards. To empiracally prove that this is the case, a technique called early stopping is deployed here. The gist of that approach is to stop the training as soon as the validation loss does not decrease, indicating that the model is about to get worse due to overfitting. In reality, the training is noisy and after some epochs of increasing losses they can decrease again to an even lower value. Thus, a grace period is introduced where if in the next n epochs no improvements are found, then the training stops. Figure 5.22 shows the results of an experiment where different n where tested for ((10, 6), softmax, Adam) the best performing model. As reference, the training history of a model with no early stopping is shown. Choosing n=10 stops the training too early, since after the bump between epochs 700 to epoch 800 the model significantly improves again. Choosing n=100 stops the model after epoch 1353 with its best loss being measured at epoch 1253. Letting the model train up to epoch 3000 shows that there is actually an even better loss at epoch 2113. To reach that place with early stopping, it would be necessary to set n to a thousand, which seems unpractical. Further,

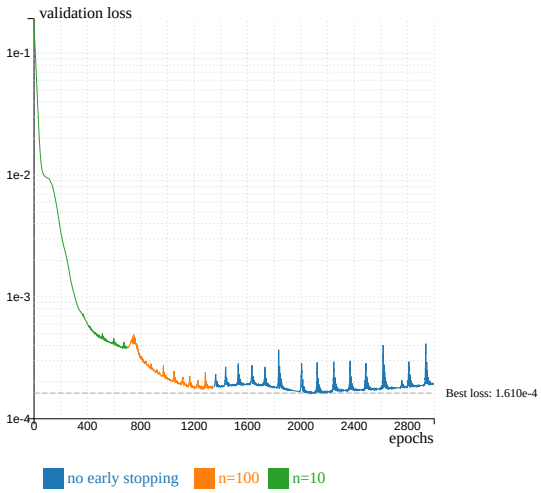


Figure 5.22: The validation loss per each training epoch for networks with the same architecture ((10, 6), softmax, Adam) but different stopping points in training. The best loss is achieved in epoch 2113.

that best loss is not even significantly smaller than the previous one ($1.610 \cdot 10^{-4}$ and $1.765 \cdot 10^{-4}$, respectively). In conclusion, it is not really worth it to train for more than a thousand epochs and it is best to stop there or use early stopping with $n = 100$.

6 Conclusion and Outlook

In summary, a visual analytisc system was presented that allows its users to train various neural networks for their experiments by selecting different training configurations and receiving their training history. Using the visualization tools that this visual analytisc system offers, these training histories can be visualized in numerous ways to gain more insight on the models. Several experiments showcased the usefulness of this tool. For example, using the validation loss plots allows the user to quickly compare the performance of different parameter settings. Inspecting the networks (average) weights enables them to see the impact of different regularization techniques. Two different weight loadouts can be compared by looking at the loss landscape inbetween them, or by looking at their distance after using dimensionality reduction algorithms. It was even possible to show the effects of the novel optimization technique GSP. All these insights can then be used to ameliorate a model's parameters or to design new experiments to gain even deeper knowledge.

In this work, several insights over the neural networks for microstructure data could be gained. Firstly, after testing several different architectures, the ones that performed the best were smaller networks. While some bigger networks achieved similar performances, the superior training speed of the smaller networks made them the obvious choice. In addition, several different activation functions and optimizers were compared with softmax and Adam coming out on top. Adam makes sense, since it is the state-of-the-art optimizer and also outperforms other optimizers in other domains. With the activation functions two groups could be observed. The main difference between both groups is that the worse performing group has an infinite endpoint in their range while the range of the better performing group does not exceed 1. It seems that limiting the neuron output, which in turn keeps the weights small, is beneficial to the network's performance.
Secondly, after experimenting with different known and one novel weight regularization techniques, the models with no or very light regularizations were significantly better than those with the regularization. This seems puzzling and contrary to common results. Perhaps, regularization is more effective in larger, more complex networks and can be detrimental in small networks like those used here.
Thirdly, the effects of the novel greedy stochastic perturbation improvement was demonstrated. GSP has the potential to significantly boost a model's performance. Adding GSP to optimizers like SGD will yield lower losses much quicker. Even with state-of-the-art optimizers, the improvements, albeit less significant, can be demonstrated. The accelerated loss decrease with GSP can be explained with the perturbation factor, which effectively makes the optimization take bigger steps.
Moving on to the ensemble learning experiments, where the results were less satisfactory. The ensemble learning variant used there could not show any improvement for the performance and even worsens it. Perhaps, the multiheaded approch does not simulated the usual ensemble learning effect and sharing the same body diminishes each heads prediction.
Lastly, the experiments with early stopping demonstrated that the ideal epoch length can be found

right after a thousand epochs. Training for longer periods still arrived at an even better epoch, however, that improvement is not much better and not worth the extra investment and risk of overfitting.

There is one intriguing observation that can be noticed in almost all the experiments, which is the lack of a global loss minima. Often a small change in the training configuration leads the network into a new different local loss minima. These local minimas can be better or worse or equally good. This suggests a non-convex structure of the loss function.

Apart from insights over the neural networks, there are also a couple of lessons learned on the used techniques. For instance, while being merely a linear projection method, PCA proved to be the dimensionality reduction technique that was the most suitable to represent the network's weights in two dimensions. The projected optimization trajectories in UMAP and t-SNE often overlap with each other, resulting in misleading gaps between those jumps. Additionaly, minor differences between the weight tensors resulted in a big divergence, while with PCA the lines remain close to each other.

Outlook

This visual analytisc system is still a prototype and can easily be extended in different facets. There is the possibility of adding more different visualizations, like a three-dimensional loss landscape or projecting the loss landscape over the dimensionality reduction plots. The different metrics of the training history can also be extended by tracking things like the neuron output. There are also many different machine learning techniques that could be implemented, like cyclic learning rates proposed in [Smi17]. This system also has some limitations that could be ameliorated, such as the fact, that each training configuration has to be done manually by hand. It would be desirable to have the options to automate this, which would be extremely useful during hyperparameter optimization. A feature that could start grid search or another algorithm described in [LL19] would go a long way. Another problem is that the loss landscape becomes fuzzy if the difference between the highest measured loss and the lowest measured loss is too big. Details and contours get lost because the isolines and the coloring have to accomodate for the places with extreme losses. One possible solution is to allow the user to manually set the values of isolines and the color scale. Of course there is always room for more features and more interactions, but the scope of this work only allowed for this much. Still, alot of insights have been gained using this visual analytics system and this tool has the potential to be upgraded in future works to facilitate even more learnings.

Bibliography

[AW10] H. Abdi, L. J. Williams. “Principal component analysis”. In: *Wiley interdisciplinary reviews: computational statistics* 2.4 (2010), pp. 433–459 (cit. on p. 6).

[Bos12] M. Bostock. *D3.js - Data-Driven Documents*. 2012. URL: http://d3js.org/ (cit. on p. 18).

[Cho+15] F. Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras (cit. on p. 21).

[Com09] W. Commons. *Visualize the idea of principal component analysis*. 2009. URL: https://commons.wikimedia.org/wiki/File:PcaIdea.png (cit. on p. 14).

[CS20] H. Chen, A. Shrivastava. “Group ensemble: Learning an ensemble of convnets in a single convnet”. In: *arXiv preprint arXiv:2007.00649* (2020) (cit. on p. 7).

[Cut] R. Cutura. *DruidJS*. https://github.com/saehm/DruidJS. Accessed: 2022-09-27 (cit. on p. 24).

[GR71] G. H. Golub, C. Reinsch. “Singular value decomposition and least squares solutions”. In: *Linear algebra*. Springer, 1971, pp. 134–151 (cit. on p. 24).

[HAO+22] D. Hägele, M. Abdelaal, O. S. Oguz, M. Toussaint, D. Weiskopf. “Visual analytics for nonlinear programming in robot motion planning”. In: *Journal of Visualization* 25.1 (2022), pp. 127–141 (cit. on p. 6).

[HMN+20] F. Heyen, T. Munz, M. Neumann, D. Ortega, N. T. Vu, D. Weiskopf, M. Sedlmair. “ClaVis: An interactive visual comparison system for classifiers”. In: *Proceedings of the International Conference on Advanced Visual Interfaces*. 2020, pp. 1–9 (cit. on p. 6).

[HR02] G. E. Hinton, S. Roweis. “Stochastic neighbor embedding”. In: *Advances in neural information processing systems* 15 (2002) (cit. on pp. 6, 15).

[HSS+21] A. Hinterreiter, C. Steinparz, M. SchÖfl, H. Stitz, M. Streit. “Projection Path Explorer: Exploring visual patterns in projected decision-making paths”. In: *ACM Transactions on Interactive Intelligent Systems (TiiS)* 11.3-4 (2021), pp. 1–29 (cit. on p. 6).

[KGV83] S. Kirkpatrick, C. D. Gelatt Jr, M. P. Vecchi. “Optimization by simulated annealing”. In: *science* 220.4598 (1983), pp. 671–680 (cit. on p. 7).

[LF19] J. Lißner, F. Fritzen. “Data-driven microstructure property relations”. In: *Mathematical and Computational Applications* 24.2 (2019), p. 57 (cit. on pp. 4, 6, 16, 19).

[LL19] P. Liashchynskyi, P. Liashchynskyi. “Grid search, random search, genetic algorithm: a big comparison for NAS”. In: *arXiv preprint arXiv:1912.06059* (2019) (cit. on p. 46).

[LXT+18] H. Li, Z. Xu, G. Taylor, C. Studer, T. Goldstein. “Visualizing the loss landscape of neural nets”. In: *Advances in neural information processing systems* 31 (2018) (cit. on p. 6).

[Map03] C. Maple. “Geometric design and space planning using the marching squares and marching cube algorithms”. In: *2003 international conference on geometric modeling and graphics, 2003. Proceedings*. IEEE. 2003, pp. 90–95 (cit. on p. 23).

[MBM20] R. Moradi, R. Berangi, B. Minaei. “A survey of regularization strategies for deep models”. In: *Artificial Intelligence Review* 53.6 (2020), pp. 3947–3986 (cit. on p. 7).

[MHM18] L. McInnes, J. Healy, J. Melville. “Umap: Uniform manifold approximation and projection for dimension reduction”. In: *arXiv preprint arXiv:1802.03426* (2018) (cit. on pp. 6, 15).

[Pea01] K. Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572 (cit. on p. 6).

[Pre98] L. Prechelt. “Early stopping-but when?” In: *Neural Networks: Tricks of the trade*. Springer, 1998, pp. 55–69 (cit. on p. 7).

[Sal] D. Salvati. *SVD-JS*. https://www.npmjs.com/package/svd-js. Accessed: 2022-09-26 (cit. on p. 24).

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 7).

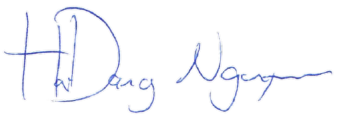
[Smi17] L. N. Smith. “Cyclical learning rates for training neural networks”. In: *2017 IEEE winter conference on applications of computer vision (WACV)*. IEEE. 2017, pp. 464–472 (cit. on pp. 12, 46).

[VH08] L. Van der Maaten, G. Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008) (cit. on pp. 6, 15).

All links were last followed on November 04, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 05.11.2022 

place, date, signature