Comparison of architectures and parameters for artificial neural networks

Sebastian Jost

January 22, 2022

eingereicht von:	Sebastian Jost
geboren am:	04.09.2000
geboren in:	Berlin, Detuschland
Gutachter/innen:	Prof. Dr. Andrea Walther
	Dr. Benjamin Jurgelucks

Eingereicht beim Institut für Mathematik der Humboldt-Universität zu Berlin am:

Contents

1 Abstract

We investigate the importance of many different hyperparemters of artificial neural networks regarding training time and performance of the trained networks. At the same time we try to find decent starting values for later parameter optimization for these parameters.
We do all of this by performing a grid search with a few different values for every parameter. Therefore we train neural networks for all combinations of those values and two different datasets.
After training these networks we find that we also get good model performance with comparably very small networks.
Regarding parameter importance we mostly confirm expectations but also show that the parameters of the optimizer can have a very significant influence on the final performance.
We also tried out a small modification of the Adam optimizer, however the cost of that modification outweighs the very small benefit.

2 Introduction and motivation

Machine learning is a field of study that grows more important every year. The frequently used artificial neural networks are incredibly powerful but they are also very hard to tune to achieve good results. There are many hyperparameters that can be chosen and optimized, but testing a certain setting of parameters can be very time consuming.
Here we will examine the influence many of these hyperparameters have on the training time and performance of the neural networks to find out on which parameters the optimization efforts are most important and what are good starting points for their values.
As examples we will use two small datasets introduced later to keep the computation time reasonable.
We will also try out a modification of the well known optimization algorithm Adam.

3 Definitions

First we will define most of the parameters and settings used in the experiments.

3.1 Neural network architecture

(Artificial) neural network

Artificial neural network is a very broad and not very well defined term. Many different definitions exist, some are very specific, others are very broad. The idea is to mimic the biological neural networks found in brains.
In this thesis we will consider sequential, densely connected, artificial neural networks. That means, that they are functions built by composing layer functions.
A neural network with $s \in \mathbb{N}_{\geq 1}$ layers can be expressed as a function $f : \mathbb{R}^{k_0} \to \mathbb{R}^{k_s}$, where $k_0, ..., k_s \in \mathbb{N}_0$ are the number of neurons in each layer. Then f is defined by layer functions $g_0, ..., g_s$ (defined in Eq. (3)):

$$f\left(\vec{x}_{k_0}\right) = \left(g_s \circ ... \circ g_1 \circ g_0\right)\left(\vec{x}_{k_0}\right) \tag{1}$$

The first layer g_0 only consists of an activation function ψ_0 (more on those in section 3.3). In this thesis we will always assume the identity map as the activation function of the first layer $g_0(x) = x$.
Neural networks are often also referred to as *neural nets* or, if the context clarifies it, just *models*.

Neuron

A Neuron is the smallest unit of a neural network and can be expressed as a function $h_{n+1,l} : \mathbb{R}^{k_n} \to \mathbb{R}$ ($n \in \{0, ..., s-1\}$, $l \in \{1, ..., k_{n+1}\}$). Let $b_{n+1,l} \in \mathbb{R}$ be a so called bias summand, $x_{n,i} \in \mathbb{R}$ are the outputs of the previous layer (layer n) and $a_{n+1,l,i} \in \mathbb{R}$ are weights for those outputs. Then the neuron is defined by the function:

$$h_{n+1,l}(x_{n,1}, ..., x_{n,k_n}) = b_{n+1,l} + \sum_{i=1}^{k_n} a_{n+1,l,i} \cdot x_{n,i} \tag{2}$$

Layer

A layer of an artificial neural network is a function $g_n : \mathbb{R}^{k_{n-1}} \to \mathbb{R}^{k_n}$ $n \in \{1, ..., s\}$ built from multiple neurons.
Let $\vec{x}_{k_{n-1}} \in \mathbb{R}^{k_{n-1}}$ be the output of the previous layer as a vector, $\psi_n : \mathbb{R}^{k_n} \to \mathbb{R}^{k_n}$ is an activation function (more details in section 3.3) and $h_{n,l} : \mathbb{R}^{k_{n-1}} \to \mathbb{R}, l \in \{1, ..., k_n\}$ are the neurons of that layer. Then the layer function is defined as:

$$\begin{aligned} g_n\left(\vec{x}_{k_{n-1}}\right) &= \psi_n\left(h_{n,1}(\vec{x}_{k_{n-1}}), ..., h_{n,k_n}(\vec{x}_{k_{n-1}})\right) \\ &= \psi_n\left(A_n\vec{x}_{k_{n-1}} + B_n\right) \end{aligned} \tag{3}$$

Eq. (3) simplifies the definition by combining the parameters of the neurons into a weight matrix $A_n \in \mathbb{R}^{k_n \times k_{n-1}}$ and a bias vector $B_n \in \mathbb{R}^{k_n}$. Then the combination of a row of A_n and the corresponding entry of B_n defines the parameters of a single neuron in that layer.
A layer is called *dense*, if all elements of A_n (and B_n) are learnable parameters, that can be changed by the optimizer. Other layer types can fix some of those parameters, usually to zero.
The first layer g_0 is usually called *input layer* and only consists of an activation function ($\to A_0 = Id_{k_0 \times k_0}, B_0 = \vec{0}_{k_0}$).
The last layer g_s is called *output layer*.
All layers in between are called *hidden layers*.

3.2 Training parameters

Once we have decided on a network layout by choosing a number of layers and number of neurons for each layer, there are still many hyperparameters we have to set before actually training the network. Here we will define the ones we will use later.
During training we will evaluate the neural network on a labelled dataset and use the labels and the network output to calculate the weights in the network.
Let N be the number of labelled samples in the dataset.

Number of epochs $E \in \mathbb{N}$

The number of epochs is the number of times we use each labelled sample (input-output pair) in the dataset for training the network. The training will start in epoch one and progress to the next one once time all samples have been used.

Batch size $B \in \{1, ..., N\}$

During training on the dataset, we can compute updates to the weights and biases at any time. The batch size defines the number of samples after which such an update is calculated. If there are less than B samples left, an update can be calculated with the remaining samples.
So during each epoch, there will be $\lceil N/B \rceil$ updates to the weights. Consequently there will be $E \cdot \lceil N/B \rceil$ optimization steps during the entire training process.

3.3 Activation functions

Activation functions, as seen in Eq.(3), can be any function $\psi_n : \mathbb{R}^{k_n} \to \mathbb{R}^{k_n}$ ($n \in \{0, ..., s\}$. They are applied to the output of a layer and transform it to introduce nonlinearity to the neural network.
We will now list the different activation functions used in this thesis. Functions defined on $\mathbb{R} \to \mathbb{R}$ are applied to each neuron's output independently. The following definitions can be found in [Tfd, activations].

Linear $\mathbb{R} \to \mathbb{R}$

The *linear* activation function is the identity map:

$$\psi_{lin}(x) = x$$

ReLU $\mathbb{R} \to \mathbb{R}$

ReLU is a commonly used piecewise linear activation function.

$$\psi_{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x \geq 0 \end{cases}$$

Sigmoid $\mathbb{R} \to \mathbb{R}$

The *sigmoid* activation is defined by a continuous function that approaches 0 for $x \to -\infty$ and approaches 1 for $x \to \infty$.

$$\psi_\sigma(x) = \frac{1}{1 + e^{-x}}$$

Softmax $\mathbb{R}^{k_n} \to \mathbb{R}^{k_n}$

The *softmax* activation is different in the sense that it doesn't act on each Neuron in a layer independently but instead uses all values of a layer as inputs. $e^{\vec{x}}$ means the vector $(e^{x_1}, ..., e^{x_{k_n}})$.

$$\psi_{softmax}(\vec{x}) = \frac{e^{\vec{x}}}{\sum_{i=1}^{d} e^{x_i}}$$

3.4 Loss functions

Loss functions are used to evaluate the output $f(\vec{x}_0) =: \vec{y}$ of a neural network and compare it to a given target value (=label) $\vec{\hat{y}}$ for the input $\vec{x}_0$. As the name suggests, they are functions too. More specifically they can be defined as any function $L : \mathbb{R}^{k_s} \times \mathbb{R}^{k_s} \to \mathbb{R}$, that maps the output of a neural network and a target value to a real number. Here we will list the loss functions used in the experiments. These definitions can also be found in [Tfd, losses].

Mean absolute error $\mathbb{R}^{k_s} \times \mathbb{R}^{k_s} \to \mathbb{R}$

$$L_{MAE}\left(\vec{y}, \vec{\hat{y}}\right) := \frac{\sum_{i=1}^{k_s} |y_i - \hat{y}_i|}{k_s}$$

Mean squared error: $\mathbb{R}^{k_s} \times \mathbb{R}^{k_s} \to \mathbb{R}$

$$L_{MSE}\left(\vec{y}, \vec{\hat{y}}\right) := \frac{\sum_{i=1}^{k_s} |y_i - \hat{y}_i|^2}{k_s}$$

When dealing with real numbers, the absolute value does not need to be calculated. If $k_s = 1$, then $L_{MAE} = \sqrt{L_{MSE}}$.

log cosh: $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$

$$L_{logcosh}\left(y_i, \hat{y}_i\right) := \log\left(\cosh(y_i - \hat{y}_i)\right)$$

Categorical crossentropy: $\mathbb{R}^{k_s} \times \mathbb{R}^{k_s} \to \mathbb{R}$

Categorical crossentropy is a loss function meant for multi-class classification, where each input falls into exactly one category. $\hat{y}$ should be a label in $\left\{v \in \{0,1\}^{k_s}, |v| = 1\right\}$. This is also called one-hot encoding of $\hat{y}$. The following formula can be found in [Cro].

$$L_{crossentropy}\left(\vec{y}, \vec{\hat{y}}\right) := -\sum_{i=1}^{k_s} y_i \cdot \log(\hat{y}_i) \tag{4}$$

3.5 Optimizers

The optimizer is used to calculate updates to the weights and biases of the neural network based on the value and gradient of the loss function. These updates are calculated for every batch and can therefore be calculated many times in each epoch.

3.5.1 Adam

Adam is an optimization algorithm used for finding optimal parameters of artificial neural networks. It was first introduced in [KB17]. In Algorithm 1 we repeat the algotihm as it was defined in [KB17]. In that algorithm and in all following calculations, the vector operations are elementwise. When using Adam for neural network optimization, *maximum iterations* is $E \cdot \lceil N/B \rceil$ (with N the number of samples in the dataset, E the number of epochs and B the batch size).
Adam is a first order optimizer, meaning that it only requires the first derivative of the loss function. In addition Adam is momentum based, meaning that it uses the moments of the stochastic gradient to improve results. These moments simulate inertia, providing resistance to changes of the descent direction by averaging the gradients of the past steps. This can help to overcome local minima and find either lower local minima or even a global minimum of the loss function.
Here we collect all weights and biases into one vector $\theta \in \mathbb{R}^d$ and interpret $\nabla\tilde{L}(\theta)$ as the average loss of the neural network with parameters θ over all samples within the batch. The variable t counts how many updates have been computed and serves as an index for the other variables that change in each iteration.

Algorithm 1 original Adam algorithm as given in [KB17, Alg. 1] with very minor changes

Require: $\beta_1, \beta_2 \in [0, 1[, \quad \alpha, \varepsilon \in \mathbb{R}_{>0}$ ▷ hyperparameters
Require: $\nabla\tilde{L} : \mathbb{R}^d \to \mathbb{R}^d$ or $\tilde{L} : \mathbb{R}^d \to \mathbb{R}$ ▷ loss function $\tilde{L}$
Require: $\theta_0 \in \mathbb{R}^d$ ▷ parameter of loss function
▷ All vector operations are elementwise.

$m_0 := \vec{0} \in \mathbb{R}^d$
$v_0 := \vec{0} \in \mathbb{R}^d$
$t := 0 \in \mathbb{N}_0$
while $t \leq$ maximum iterations **or** sufficient convergence of θ_t **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla\tilde{L}(\theta_{t-1})$ ▷ calculate gradient
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ ▷ calculate first moment
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ ▷ calculate second moment
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ ▷ bias correction for first moment
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ ▷ bias correction for second moment
  $\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ ▷ update parameters θ
end while
return θ_t

[KB17, section 2] also introduces an alternative update rule that uses less operations and therefore runs faster and uses less memory. The variables $\hat{m}_t$ and $\hat{v}_t$ can be ommited. Instead the authors add a learning rate for each step that includes the bias correction factors:

$$\alpha_t = \alpha \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t} \tag{5}$$

Consequently the update of the parameters θ is changed to:

$$\theta_t \leftarrow \theta_{t-1} - \alpha_t m_t / (\sqrt{v_t} + \varepsilon) \tag{6}$$

This reduces the number of times a scalar and vector need to be multiplied with the slight change that now the bias correction for v_t is also applied to ε. A value of ε dependent on t could correct for this change, but this is usually not done.

3.5.2 Centered Adam

As a second optimizer I introduce a slight variation of Adam with hopes of improving Adam.
Adam uses $\hat{m}_t$ as an approximation of the expected value (= first moment) of g_t and $\hat{v}_t$ as an approximation of the second moment of g_t, also called "uncentered variance".

The update formula scales the steps using this second moment. The reason to do that is to make larger steps if there is low variance in the calculated gradients and to make smaller steps if the variance is large, corresponding to gradient entries changing quickly. Keep in mind, that every vector operation in Adam is componentwise. Therefore the expected value and variance are calculated for each component of θ individually.

However Adam doesn't actually calculate the variance ($\mathbb{E}[(g_t - \mathbb{E}[g_t])^2]$) but the second moment $\mathbb{E}[g_t^2]$ instead. If $\mathbb{E}[g_t] = 0$, both definitions are identical and the second moment requires less operations to calculate. But $\mathbb{E}[g_t] = 0$ means that θ_t is a critical point of the loss function and the steps taken don't actually decrease the loss.

I propose changing the update for v_t in Adam, such that v_t approximates the variance instead of the second moment. The new update formula would be:

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)(g_t - m_t)^2 \tag{7}$$

Note that we use m_t as the approximation of g_t rather than the bias corrected $\hat{m}_t$. This is because preliminary tests have shown that using $\hat{m}_t$ yields significantly worse results (frequently more than 30% lower accuracy on MNIST dataset with parameters that worked well for Adam and cAdam with update rule Eq. (7)). In those tests I used the following update rule:

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)(g_t - \hat{m}_t)^2 \tag{8}$$

Using m_t instead of $\hat{m}_t$ also has the advantage of still allowing the variation of using Eq.(5) to improve runtime. Eq. (8) requires computing $\hat{m}_t$ explicitly and therefore more operations. How much time the optimization Eq. (5) saves, can be seen in appendix 11.2.

3.5.3 Bias correction for centered Adam

Now we will calculate the bias correction term required for this new update formula for v_t. The bias correction for m_t remains unchanged.
First of all we use an explicit formula for v_t instead of the recursive one used in implementations of the algorithm:

$$v_t = (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot (g_i - m_i)^2$$

v_t is supposed to be an approximation of $\mathrm{Var}(g_t) := \mathbb{E}[(g_t - \mathbb{E}[g_t])^2]$. Therefor we want $\mathbb{E}[v_t] = \mathrm{Var}(g_t)$. To check, whether that is the case, we calculate $\mathbb{E}[v_t]$:

$$\begin{aligned}
\mathbb{E}[v_t] &= \mathbb{E}\left[(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot (g_i - m_i)^2\right] \\
&= (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot \mathbb{E}\left[(g_i - m_i)^2\right] && \text{linearity of } \mathbb{E} \\
&= (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} \cdot \left(\mathbb{E}\left[(g_t - m_t)^2\right] - k_i\right) && k_i := \mathbb{E}\left[(g_t - m_t)^2\right] - \mathbb{E}\left[(g_i - m_i)^2\right] \\
&= (1-\beta_2)\sum_{i=1}^{t} \left(\beta_2^{t-i} \cdot \mathbb{E}\left[(g_t - m_t)^2\right] - \beta_2^{t-i} \cdot k_i\right) \\
&= \mathbb{E}\left[(g_t - m_t)^2\right](1-\beta_2)\sum_{i=1}^{t}\left(\beta_2^{t-i}\right) \underbrace{-(1-\beta_2)\sum_{i=1}^{t}\left(\beta_2^{t-i} \cdot k_i\right)}_{=:\delta_t} \\
&= \mathbb{E}\left[(g_t - m_t)^2\right]\sum_{i=1}^{t}\left(\beta_2^{t-i} - \beta_2^{t-i+1}\right) + \delta_t \\
&= \mathbb{E}\left[(g_t - m_t)^2\right](1-\beta_2^t) + \delta_t && \text{teslescopic sum}
\end{aligned}$$

This proof is inspired by the bias correction proof in [KB17, Section 3] with added details from [Jos21],
The original derivation (in [KB17, Section 3]) for the bias correction factor in Adam assumes that $\beta_2^{t-i}\left(\mathbb{E}[g_t^2] - \mathbb{E}[g_i^2]\right)$ is small. Here we assume $\beta_2^{t-i}\left(\mathbb{E}\left[(g_t - m_t)^2\right] - \mathbb{E}\left[(g_i - m_i)^2\right]\right)$ is small instead. This is even more likely to be fulfilled since $g_t - m_t$ should already be small because m_t is supposed to be an approximation of $\mathbb{E}[g_t]$.
Therefore the bias correction for Adam can also be used for centered Adam.

3.5.4 Number of operations per iteration

We see that centered Adam only requires one more vector addition per iteration than Adam and is otherwise identical. Let there be d parameters ($\theta_t \in \mathbb{R}^d$), then Adam (with modification (5) and some other simple optimizations) requires approximately $3d$ additions, $7d$ multiplications and d square root operations per iteration. Centered Adam requires $4d$ additions, $7d$ multiplications and d square root operations per iteration.
Based on this, we expect centered Adam to yield slightly longer training times.

4 Experiment description

4.1 Experiment goals

There are several goals for the experiments:

1. Find out which parameters have the biggest impact on network performance and training time.
2. Find out whether centered Adam is actually an improvement over Adam.
3. Find good parameter settings for the neural networks.

The focus is on the first two goals, but we want to ensure that our results are still comparable with those of other publications. Therefor we need to find sufficiently good parameter settings.

4.2 Experiment setup

4.2.1 Datasets

Now we will introduce the two datasets that are used in the experiments. To avoid the common problem of differently scaled variables in optimization algorithms, we use some preprocessing on the given data.

MNIST dataset

MNIST is a dataset commonly used for comparing neural networks. It contains greyscale images of handwritten digits as well as corresponding integer labels 0 to 9. Each input is a 28x28 matrix of integer values in the interval $[0, 255]$. The dataset contains 60000 training images and 10000 dedicated test images.

Preprocessing for MNIST

For training, the integer values of the inputs get scaled to the interval $[0, 1] \subset \mathbb{R}$ by dividing by 255.
The labels of the dataset are integers in $[0, 9]$. Those get one-hot encoded, such that each label is a vector $v \in \{0, 1\}^{10}$ with $|v| = 1$. Every integer in the original labels corresponds to one position in these vectors.
Examples:
$2 \mapsto (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$
$5 \mapsto (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$

ChemEx dataset

The second dataset contains information about ternary chemical compounds as inputs and their formation energy as labels. We will call this dataset ChemEx.
More Specifically inputs are 36 real numbers describing the following properties in the given units:

- average electronegativity of compound
- average group of compound
- average row of compound
- average atomic mass of compound (in u)
- average ionic radius of compound (in Å)
- average electron affinity of compound (in eV)
- average first ionization energy of compound (in eV)
- average van der waals radius of compound (in Å)
- ratio of electronegativity of each element to average of compound (3 values)
- ratio of group of each element to average of compound (3 values)
- ratio of row of each element to average of compound (3 values)
- ratio of atomic mass of each element to average of compound (3 values)
- ratio of ionic radius of each element to average of compound (3 values)
- ratio of electron affinity of each element to average of compound (3 values)
- ratio first ionization energy of each element to average of compound (3 values)
- ratio of van der waals radius of each element to average of compound (3 values)
- orbital fraction of valence electrons s
- orbital fraction of valence electrons p
- orbital fraction of valence electrons d
- orbital fraction of valence electrons f

The label is a real number representing the formation energy of the compound in eV/atom.

Conversion to SI-units:

- $1\text{Å} = 10^{-10}\text{m}$
- $1\text{eV} \approx 1.602 \times 10^{-19}\text{kg} \cdot \text{m}^2 \cdot \text{s}^{-2}$
- $1\text{u} \approx 1.661 \times 10^{-27}\text{kg}$

This dataset contains 49345 labeled input vectors and no dedicated datapoints for testing. Therefore the program will randomly choose 10% of the samples (4934 samples)as a test dataset.

There is a known error in the dataset that was used for the experiments below: The ratio of electronegativity of element C to the average of the compound is the same as the ratio of element B. Therefore this column is redundant. However this value can also be learned from the combination of row and group for the elements, since those also uniquely define the element and it's electronegativity.
We still use this incorrect dataset to be able to compare the results to those in [Ren19], since the same error was included there.

Preprocessing for ChemEx

Before splitting the dataset into training and test sets and therefore before training, the inputs are rescaled and normalized such that they have expected value 0 and standard deviation 1. The code for loading and rescaling the data was provided by Franz Bethke. Note that the scaling factors used depend on the given data and they must be applied to any new data before that data is given to the trained neural network. Otherwise the prediction will likely be much further from the truth.

4.2.2 Adjustable parameters

When training a neural network there are many parameters that can be adjusted. Some of those are:

parameter name	explanation
Neural network parameters:	
- neurons per layer	number of neurons in each layer
- input shape	number (and shape) of input neurons
- output shape	number (and shape) of output neurons
- activation functions	function applied to the output of each neuron
- last activation function	activation function in the output layer
- layer types	descriptor specifying how layers are connected
- loss function	function evaluating model performance
training parameters:	
- number of epochs	number of training iterations
- batch size	number of samples for calculating gradients
- validation split	proportion of training data used for validation
- training data percentage	proportion of non-test data used for experiment
- number of repetitions	how often each network is trained
optimizer parameters:	
- optimizer	optimization algorithm
- learning rate α	scaling factor for optimization steps (step size)
- ε	number to prevent division by 0
- β_1	decay factor for first moment (expected value) m_t
- β_2	decay factor for second moment or variance v_t

Table 4.1: parameter explanations

The optimizer parameters depend on the optimizer being used. The ones listed here are the hyperparameters of Adam and centered Adam (cAdam).
For the experiments we will choose some values for most of these parameters. Models will then be trained for every possible combinations of those parameters. The shape of inputs and outputs can make implementations easier, it does not influence the results.

4.2.3 Parameter values used

Here we list all the values used for the experiments. To keep the total computation time reasonable, we only choose up to three values for each parameter. While this does not provide very detailed results, for most parameters it is sufficient. "neurons per layer" describes the number of neurons in the hidden layers.

parameter name	values for MNIST	values for ChemEx
Model parameters:		
- neurons per layer	$(32,)$ / $(50, 10)$ / $(20, 15, 10)$	$(32,)$ / $(50, 10)$ / $(20, 15, 10)$
- input shape	$(28, 28)$	$(36,)$
- output shape	10	1
- activation functions	ReLU / sigmoid	ReLU / sigmoid
- last activation function	softmax / sigmoid	linear
- layer types	dense (=fully connected)	dense (=fully connected)
- loss function	categorical crossentropy / mean squared error	log cosh / mean squared error
training parameters:		
- number of epochs	5 / 25 / 50	50 / 150 / 500
- batch size	100 / 1000 / 10000	100 / 1000 / 10000
- validation split	0.2	0.2
- training data percentage	1	1
- number of repetitions	5	5
optimizer parameters:		
- optimizer	Adam / centered Adam	Adam
- learning rate α	0.1 / 0.01 / 0.001	0.1 / 0.01 / 0.001
- ε	1 / 10^{-7}	1 / 10^{-7}
- β_1	0.9	0.9
- β_2	0.999	0.999

Table 4.2: Parameter settings in experiments: values are separated by '/'. These choices result in 2592 different parameter combinations for MNIST and 648 combinations for ChemEx. Every model is trained 5 times.

The chosen hidden layers result in the following number of learnable parameters for the neural networks:

neurons in hidden layers	number of parameters	
	for MNIST	for ChemEx
$(32,)$	25450	1217
$(50, 10)$	39870	2371
$(20, 25, 10)$	16285	1226

Table 4.3: number of learnable parameters in trained networks

4.2.4 Measurements during training and dataset split

While training the networks we measure the time it takes to train them, the loss and validation loss in each epoch and in the case of MNIST also the accuracy and validation accuracy in each epoch.
Before training, some samples of the available data are withheld as test data. This test dataset remains the same for every parameter setting and every repetition of model training. Using these test datasets every trained model is tested and the test loss (and test accuracy for MNIST) is saved.
By separating test- and validation dataset and keeping the test data identical for all parameter settings, we ensure a fair comparison between all parameter settings.
Here, we only use the validation data to get an estimate of how the randomness of choosing it affects the result. However in future experiments the code provided in [Sou] could also be used to compare techniques that use the validation data for hyperparameter optimization.
In Section 11.2, we also include a different training data percentage to see how the amount of available data affects the result. For this we choose a quite extreme case of only having 30% of the data.

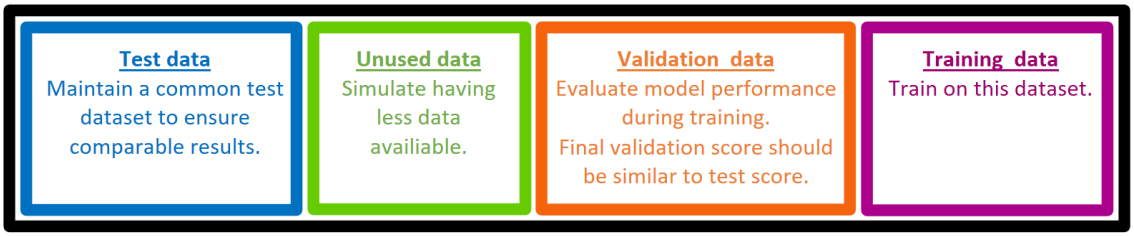

Figure 1: Dataset split visualization; The black box represents all available labeled datapoints.

4.3 Optimizer details

I used the python library tensorflow to implement centered Adam (=cAdam) to keep the computation time comparable to Adam. This was made much easier by looking at another example of a custom tensorflow optimizer that is available in [Wal]. I also used this to re-implement Adam to confirm that this custom implementation is comparable to the original tensorflow implementation [Tfa].
The implementation of cAdam used in the experiments uses the update rules similar to the one in Algorithm 1, without the faster modifications (5) and (6). This was a slight oversight but didn't change the results too much. Tensorflow 2.x on the other hand uses the more efficient update rule using (5) and (6).
A test using cAdam with the optimizations (5) and (6) can be found in Appendix 11.2.

5 Experiment results

Now that we planned what to calculate in the experiments, we need to plan what to do with the results.
Every model is trained five times with the same parameters but random initial values for all weights in the neural network. For the resulting training times and other tracked metrics we calculate the mean of the five corresponding values for each parameter setting. All further calculations will be based on these mean values.
To achieve goal 1 of estimating the influence each parameter has on the performance of the trained model, we will use two measurements: **win ratios** and **average differences**. Both are calculated by first choosing one parameter p with multiple different values. Then the program will loop through all combinations of all remaining parameter values and for each combination perform the appropriate action:

1. **win ratios:** Keep a counter for each possible value of the parameter p and for each category (training time, loss, ...). Determine which value for the parameter p has the best score in each category. Then add one to the corresponding counters.

   The win ratio shows in how many parameter settings a given value performs better than the other tested values (max = 100%, min = 0%).

2. **average differences:** Keep a list for each possible value of the parameter p and for each category. For each category find the value of p with the best score in that category. Then calculate the absolute difference of the score of each value of the parameter p to this best score and save these differences in the corresponding lists.

   Once the lists contain difference values for every examined parameter combination, the average values are calculated and later displayed.

   The average differences show the magnitude of the expected effects from changing each parameter's value. A small average difference means that the value works quite well in many different scenarios.

These two measurements will also be used to achieve goal 2 of evaluating the performance of centered Adam.

Notes for interpreting the following tables

There are two different kinds of tables listed:

- **best/worst tables** show the best and worst parameter settings for the given metric. Rows, where all values are equal have colored text, the others have black text. Columns are always sorted by the value in the first such that the best value is to the left and the worst value to the right. Additionally the columns with absolute best and worst results are highlighted too.
- **parameter-influence tables** show the impact each parameter setting had on the training results using win ratios and average differences as defined above. The column *best value* displays which parameter setting had the best average performance if both measurements agree on that. Otherwise the best value is *unclear*.

5.1 Training time results

Here I will list the measurements regarding training time of the neural networks obtained with the methods described above. We abbreviate categorical crossentropy as "cat-cross".

MNIST best and worst values for training time

parameter name	best values				
training time	1.0347	1.0368	1.0398	1.0412	1.0446
neurons per layer	(32,)	(32,)	(32,)	(32,)	(32,)
activation functions	sigmoid	sigmoid	ReLU	ReLU	sigmoid
last activation function	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid
loss function	MSE	MSE	MSE	MSE	MSE
training data percentage	1	1	1	1	1
number of epochs	5	5	5	5	5
batch size	10000	10000	10000	10000	10000
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.001	0.01	0.001	0.1	0.1
ε	10^{-7}	10^{-7}	10^{-7}	1.0	1.0

Table 5.1: best settings regarding *training time* for the MNIST dataset

parameter name	worst values				
training time	106.16	106.61	106.64	106.92	106.93
neurons per layer	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)
activation functions	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid
last activation function	softmax	softmax	softmax	softmax	softmax
loss function	cat-cross	cat-cross	cat-cross	cat-cross	cat-cross
training data percentage	1	1	1	1	1
number of epochs	50	50	50	50	50
batch size	100	100	100	100	100
optimizer	cAdam	cAdam	cAdam	cAdam	cAdam
learning rate	0.001	0.01	0.1	0.01	0.1
ε	1.0	1.0	1.0	10^{-7}	10^{-7}

Table 5.2: worst settings regarding *training time* for the MNIST dataset

MNIST parameter influence on training time

parameter name	parameter values			win ratios in %			avg. differences in s			best value
neurons per layer	(32,)	(50, 10)	(20, 15, 10)	98.6	1.2	0.2	0.0	2.24	4.60	(32,)
activation functions	ReLU		sigmoid	49.9		50.1	0.081		0.087	*unclear*
last activation func.	softmax		sigmoid	24.8		75.2	0.252		0.023	sigmoid
loss function	cat-cross		MSE	52.6		47.4	0.171		0.831	cat-cross
number of epochs	5	25	50	100	0	0	0	11.18	25.21	5
batch size	100	1000	10000	0	0	100	33.03	2.75	0	10000
optimizer	Adam		cAdam	100		0	0		7.187	Adam
learning rate	0.1	0.01	0.001	34.7	30.9	34.4	0.075	0.155	0.157	0.1
ε	1.0		10^{-7}	54.9		45.1	0.058		0.068	1.0

Table 5.3: parameter influence regarding *training time* for the MNIST dataset

ChemEx best and worst values for training time

parameter name	best values				
training time	1.9308	1.932	1.9436	1.9454	1.9488
neurons per layer	(32,)	(32,)	(32,)	(32,)	(32,)
activation functions	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid
last activation function	linear	linear	linear	linear	linear
loss function	MSE	MSE	MSE	MSE	MSE
training data percentage	1	1	1	1	1
number of epochs	50	50	50	50	50
batch size	10000	10000	10000	10000	10000
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.01	0.1	0.001	0.01	0.1
ε	10^{-7}	1	10^{-7}	1	10^{-7}

Table 5.4: best settings regarding *training time* for the ChemEx dataset

parameter name	worst values				
training time	629.59	634.79	640.87	643.69	652.02
neurons per layer	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)	(20, 15, 10)
activation functions	sigmoid	sigmoid	sigmoid	sigmoid	sigmoid
last activation function	linear	linear	linear	linear	linear
loss function	log cosh	log cosh	log cosh	log cosh	log cosh
training data percentage	1	1	1	1	1
number of epochs	500	500	500	500	500
batch size	100	100	100	100	100
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.1	0.001	0.1	0.01	0.01
ε	1	10^{-7}	10^{-7}	1	10^{-7}

Table 5.5: worst settings regarding *training time* for the ChemEx dataset

ChemEx parameter influence on training time

parameter name	parameter values			win ratios in %			avg. differences in s			best value
neurons per layer	(32,)	(50, 10)	(20, 15, 10)	99.5	0.5	0	0.0	6.562	19.63	(32,)
activation functions	ReLU	sigmoid		54.9	45.1		0.692	1.424		ReLU
loss function	MSE	log cosh		94.4	5.6		0.011	4.591		MSE
number of epochs	50	150	500	100	0	0	0	41.49	186.6	50
batch size	100	1000	10000	0	0	100	244.2	21.87	0	10000
learning rate	0.1	0.01	0.001	36.1	27.8	36.1	0.748	0.915	0.701	*unclear*
ε	1	10^{-7}		56.2	43.8		0.362	0.688		1

Table 5.6: parameter influence regarding *training time* for the ChemEx dataset

5.2 Loss results

With the tables in this section, we compare how good the neural networks were at predicting the test labels from the inputs.
Note that different loss functions may be scaled very differently. Therefore comparing results of two different loss functions is not very useful. For the MNIST dataset we can compare loss functions by measuring accuracy as that metric is independent of the loss function used. For the regression network required for the ChemEx dataset, this is

not possible. Instead we exploit that the regression network only has a single output node. The loss functions used can be inverted to yield the mean absolute error (MAE).
We can then compare the MAE values (in eV/atom) to compare the loss function's performance.
Since the loss functions for more than one output neuron are not invertible, we cannot use this method to compare loss values for the MNIST dataset. Therefore we will only show the results for accuracy for the MNIST dataset.
In the tables below, "final validation loss" refers to the validation loss in the last epoch of training.

ChemEx best and worst values for loss

parameter name	**best values**				
MAE	0.11408	0.11452	0.11653	0.11709	0.12137
test loss	0.013018	0.0065444	0.013588	0.0068427	0.0073477
final validation loss	0.01339	0.0066957	0.01449	0.0070507	0.0072831
training time	515.53	540.87	61.619	63.538	536.95
neurons per layer	(50, 10)	(50, 10)	(50, 10)	(50, 10)	(50, 10)
activation functions	sigmoid	sigmoid	sigmoid	sigmoid	ReLU
last activation function	linear	linear	linear	linear	linear
loss function	MSE	log cosh	MSE	log cosh	log cosh
training data percentage	1	1	1	1	1
number of epochs	500	500	500	500	500
batch size	100	100	1000	1000	100
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.001	0.001	0.01	0.01	0.001
ε	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}

Table 5.7: best settings regarding *test loss* for the ChemEx dataset

parameter name	**worst values**				
MAE	1.3636	1.38	1.4869	1.531	1.5561
test loss	0.75386	1.915	2.4475	0.90854	0.90961
final validation loss	0.78188	1.9706	2.5777	0.93786	0.94109
training time	1.9624	1.9914	2.0139	2.0641	2.0032
neurons per layer	(32,)	(32,)	(50, 10)	(50, 10)	(32,)
activation functions	sigmoid	ReLU	sigmoid	sigmoid	ReLU
last activation function	linear	linear	linear	linear	linear
loss function	log cosh	MSE	MSE	log cosh	log cosh
training data percentage	1	1	1	1	1
number of epochs	50	50	50	50	50
batch size	10000	10000	10000	10000	10000
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.001	0.001	0.001	0.001	0.001
ε	1	1	1	1	1

Table 5.8: worst settings regarding *test loss* for the ChemEx dataset

ChemEx parameter influence on loss

Note that in the following table the average differences are computed for the test loss using MAE as described above. These differences also have the unit eV/atom.

parameter name	parameter values			win ratios in %			avg. differences			best value
neurons per layer	(32,)	(50, 10)	(20, 15, 10)	28.7	68.1	3.2	0.014	0.033	0.056	*unclear*
activation functions	ReLU	sigmoid		67.6	32.4		0.019	0.063		ReLU
loss function	MSE	log cosh		56.8	43.2		0.012	0.025		MSE
number of epochs	50	150	500	2.8	4.2	93.1	0.135	0.058	0.006	500
batch size	100	1000	10000	72.7	18.1	9.3	0.016	0.065	0.221	100
learning rate	0.1	0.01	0.001	57.4	30.6	12.0	0.04	0.056	0.189	0.1
ε	1	10^{-7}		15.4	84.6		0.204	0.02		10^{-7}

Table 5.9: parameter influence regarding *test loss* for the ChemEx dataset

5.3 Accuracy results

Accuracy is only measured for MNIST since ChemEx is a regression network and accuracy is not defined for continuous outputs. In these tables we abbreviate categorical crossentropy loss as "cat-cross".

MNIST best and worst values for accuracy

parameter name	best values				
test accuracy	0.97118	0.97084	0.97048	0.97006	0.96986
final validation accuracy	0.96725	0.9688	0.96885	0.97	0.96845
training time	87.036	23.122	86.874	46.666	44.874
neurons per layer	(50, 10)	(50, 10)	(50, 10)	(50, 10)	(50, 10)
activation functions	ReLU	ReLU	ReLU	ReLU	ReLU
last activation function	softmax	softmax	softmax	sigmoid	sigmoid
loss function	cat-cross	cat-cross	cat-cross	cat-cross	cat-cross
training data percentage	1	1	1	1	1
number of epochs	50	25	50	50	50
batch size	100	100	100	100	100
optimizer	cAdam	Adam	cAdam	Adam	Adam
learning rate	0.001	0.1	0.1	0.001	0.1
ε	10^{-7}	1.0	1.0	10^{-7}	1.0

Table 5.10: best settings regarding *test accuracy* for the MNIST dataset

parameter name	worst values				
test accuracy	0.08544	0.08516	0.0845	0.0836	0.0821
final validation accuracy	0.08655	0.0833	0.085333	0.086567	0.0852
training time	1.2915	1.3563	1.4319	5.6985	6.3722
neurons per layer	(20, 15, 10)	(50, 10)	(32,)	(32,)	(20, 15, 10)
activation functions	ReLU	ReLU	ReLU	ReLU	ReLU
last activation function	sigmoid	sigmoid	softmax	sigmoid	sigmoid
loss function	cat-cross	MSE	MSE	MSE	MSE
training data percentage	1	1	1	1	1
number of epochs	5	5	5	50	5
batch size	10000	10000	1000	10000	100
optimizer	Adam	cAdam	Adam	Adam	Adam
learning rate	0.001	0.001	0.001	0.001	0.001
ε	1.0	1.0	1.0	1.0	1.0

Table 5.11: worst settings regarding *test accuracy* for the MNIST dataset

MNIST parameter influence on accuracy

parameter name	parameter values			win ratios in %			avg. differences			best value
neurons per layer	(32,)	(50, 10)	(20, 15, 10)	64.6	30.2	5.2	0.003	0.119	0.172	(32,)
activation functions	ReLU		sigmoid	73.8	26.2		0.062	0.114		ReLU
last activation function	softmax		sigmoid	63.1	36.9		0.007	0.07		softmax
loss function	cat-cross		MSE	87.9	12.1		0.001	0.222		cat-cross
number of epochs	5	25	50	7.6	19.0	73.4	0.159	0.047	0.006	50
batch size	100	1000	10000	75.5	19.2	5.3	0.023	0.131	0.284	100
optimizer	Adam		cAdam	33.7	66.3		0.031	0.01		cAdam
learning rate	0.1	0.01	0.001	47.6	35.1	17.4	0.175	0.096	0.203	*unclear*
ε	1.0		10^{-7}	16.3	83.7		0.438	0.058		10^{-7}

Table 5.12: parameter influence regarding *test accuracy* for the MNIST dataset

6 Observations and interpretation of experiments

6.1 Best performing networks

We start by examining the best performing network to see whether our networks are comparable to other networks trained to solve the same problems.

Best performing network for ChemEx dataset

In [Ren19], the best mean absolute error (MAE) achieved with the regressor was 0.129, whereas here we achieved a MAE of 0.114 (see table 5.7). Especially noteworthy is the size of the neural networks used to achieve these results. In [Ren19, table 5.14], several network layouts were tested and a layout of $(36 - 512 - 256 - 128 - 64 - 1)$ yielded the best results (MAE of 0.129), which has almost $200,000$ learnable parameters. The network that got the best results here only has 2371 learnable parameters (see table 4.3) using a layout of $(36 - 50 - 10 - 1)$. Comparing the training parameters to [Ren19, table 4.2], our best network uses different activation and loss function and a constant learning rate.

Best performing network MNIST dataset

Similar to the ChemEx dataset, our best performing network on the MNIST dataset has a layout of $(784-50-10-10)$, which has about $40,000$ learnable parameters (see table 4.3). Networks listed in [LCB, section: neural Nets] with similar setup and performance have at least 300 hidden neurons, resulting in $200,000$ to $500,000$ learnable parameters.

This significant difference in the number of learnable parameters can be explained by the findings in [Li+18]. They introduce the *intrinsic dimension* of a model for a given problem, which describes the minimum number of parameters needed to reach a given accuracy. That number of parameters required to reach an accuracy of 90% for the MNIST dataset is given in [Li+18, table 1] as about 750 parameters. All networks discussed before significantly surpass this number, so we can expect good results even with much smaller networks.
A similar behaviour can be expected for the ChemEx dataset, although the required number of parameters is likely different. And our tested networks are already a lot smaller.
While [Li+18] and our findings (see table 4.3) also shows that more parameters do tend to allow for better results, the gained performance becomes very small quite quickly. The extra computation time required for larger networks might be better spent on hyperparemeter optimization or training for more epochs.

6.2 Parameter influence

We will now analyse, which parameters have the greatest effect on different measured metrics and which parameters are less important.

6.2.1 Parameter influence on training time

First we will look at training time (see section 5.1). Judging from the result tables for training time (Tables 5.1 to 5.6), the optimizer parameters *learning rate* and ε have no significant influence on training time, as one would expect.
Similarly there is no significant difference in training time between choosing *ReLU* or *sigmoid* as activation functions.

Looking at loss functions we see that while *log cosh* is consistently slower than *mean squared error* (MSE), the average extra time is relatively low. On the other hand Categorical crossentropy seems to be similar to MSE in terms of runtime.
The two last activation functions used in the experiments on the MNIST dataset show a significant difference: *softmax* is quite consistently slower than *sigmoid*. However the average difference in training time is still quite small. But this shows that some activation functions can be significantly slower than others and when using them often enough, the time difference can add up.

Yet there are also parameters, that have a much larger influence on training time. Those are *network layout (=neurons per layer)*, *number of epochs* and *batch size*. This is to be expected, as these parameters significantly affect the number of mathematical operations that need to be executed to train the model. Here the average time differences between values are much larger than with the previously mentioned parameters.
Interestingly the training time seems to increase more with the number of layers in a network than it does with the number of parameters during training. Looking at tables 4.3, 5.3 and 5.6 we see that the network with hidden layers $(20, 15, 10)$ has about half as many parameters as the one with hidden layers $(50, 10)$, yet for both datasets the former network is much slower to train.

very important	medium influence	small influence
network layout	optimizer	learning rate
number of epochs	loss function	Adam parameter: ε
batch size	activation functions	activation function ReLU vs. sigmoid

Table 6.1: summary of parameter influence on *training time*

All results regarding training time are consistent between the two tested datasets.

6.2.2 Parameter influence on loss and accuracy

Here we will mainly consider tables 5.9 and 5.12. Looking at the average difference values we see that the effects different parameters have on the training results have different magnitudes.
The number of epochs and batch size directly affect the number of optimization steps that are computed. As expected the tables 5.9 and 5.12 clearly show, that these two parameters also have a significant effect on the final performance of the trained model.
Interestingly the Adam parameter ε can also drastically affect the performance by a similar amount.
While there is little difference in performance between the MSE and log cosh loss functions on the ChemEx dataset, categorical crossentropy very clearly outperformed MSE on the MNIST dataset. This shows that while some loss functions can be similar, using a suitable one for the problem can make a major difference.
The remaining investigated parameters seem to have a lower impact on the results. Although learning rates seem to have relatively high average-difference values, they are mostly quite similar between the different settings.
In other publications like [KB17] or [Dub+21] we see that the optimizer can also affect the final model performance. We did not see such a large difference here because the compared optimizers were so similar.

very important	medium influence	small influence
number of epochs	network layout	optimizer Adam vs. cAdam
batch size	learning rate	
loss function	activation functions	
Adam parameter ε	optimizer	

Table 6.2: summary of parameter influence on *loss* and *accuracy*

Most results regarding loss and accuracy are consistent between the two tested datasets.

6.3 Trends for parameter settings

6.3.1 Trends for settings regarding training time

From tables 5.3 and 5.6 we can suggest rules, how the most important parameters affect the training time.
It's very clear that increasing the number of epochs also increases training time. Similarly lowering the batch size increases training time as expected. In both cases the most extreme values tested delivered the best and worst results respectively.
Regarding the network layout, the experiments suggest that deeper networks are slower in training than shallow networks, even if the number of learnable parameters is similar or even smaller in the deeper networks.
The learning rate does not affect the training time.

6.3.2 Trends for settings regarding loss and accuracy

Tables 5.9 and 5.12 suggest that smaller batch sizes and larger number of epochs yield better loss and accuracy values. We expect to see improvements getting smaller for more extreme values. While three datapoints are not enough to draw definitive conclusions, they do support this expectation.
For both datasets we obtained the best loss/ accuracy from the network with the largest number of learnable parameters. But a large number of parameters also comes with the risk of overfitting and longer training times. [Li+18] shows that too many parameters can indeed lead to the networks memorizing the dataset rather than learning any structure that is contained.
While table 5.10 shows that a large value for the Adam parameter ε can work well, tables 5.12 and 5.9 suggest that it may be advisable to use a value close to the one recommended in [KB17] ($\varepsilon \approx 10^{-8}$) since that seems to work well much more consistently. $\varepsilon = 1$ on the other hand can often cause much worse final performance.
In table 11.4 we also see that more training data can definitely improve the results, as expected. Though the amount of data necessary to solve a task likely depends on the complexity of the task. But this one experiment is not enough to provide any definitive guidelines.
Good values for learning rate seem to be hard to predict. Since learning rate has basically no cost in terms of training time but can significantly affect the final performance, it is beneficial to spend some effort on finding a good learning rate.

The non-numerical parameter's effect on training time or final loss/ accuracy cannot be predicted easily and we don't have enough data here to provide general guidelines.

6.4 Adam vs. centered Adam

In table 5.10 we see that Adam and cAdam deliver very comparable best results. Additionally table 5.12 suggests, that cAdam achieves good results slightly more consistently, though the average difference in final accuracy is quite small.
However in table 5.10 and 5.3 we clearly see that training with cAdam takes much longer than training with Adam. From the trend that larger number of epochs and smaller batch sizes yield better results we can assume that given a fixed amount of time available for training, Adam gets better results because it can train for more epochs in the same time. Even with the runtime optimization described in Eq. (5) and Eq. (6) tested in Appendix 11.2 cAdam is still significantly slower than Adam.
Therefore the modification cAdam is not recommended for use.

6.5 Known problems

Having now finished the analysis of the experiments, there are some problems with the analysis that have not been discussed so far. These problems will be addressed in this subsection.

The analysis method using win ratios and average differences as defined in Section 5 favors parameter settings that work well for a wide variety of settings of the other parameters. Values that perform much better but only with specific settings of the remaining parameters will show up as being worse. This is not necessarily bad, but important to be aware of.
Since we keep a fixed test dataset for the experiments and report the loss/ accuracy values on that, one might expect the results to be slightly biased. We reduce this effect by training the networks multiple times with different, randomly chosen training and validation datasets and averaging the test scores for each one.
Looking at the measurements for test and validation data, we see that the values are slightly different. This suggests that the datapoints chosen for these sets and the size of the sets can affect the measured metrics. This can increase the uncertainty of the reported values. While this issue could be resolved using k-fold cross-validation, that would also require training even more models, which was not possible here.

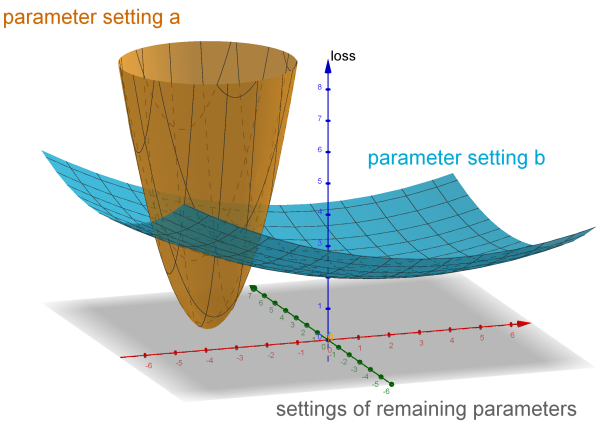

Figure 2: Visualization of analysis favoring globally good parameters over locally very good ones. Setting *a* yields the best result but setting *b* is less sensitive to other parameters.

Finally, the most important caveat: When discussing the results in section 6, we did not perform a thorough statistical analysis to quantify what "a significant difference" means. This would be useful to get more reliable results. A future improvement of the code written for these experiments could implement better statistical tests. However due to the number of trained models and often clear differences in win ratios and avg. differences, we can expect most differences to be significant according to most tests.

7 Further questions

During the experiments, we fixed the parameters and measured training time. In practice a more realistic scenario is, that the time available for training is fixed (or at least bounded from above) and the parameters should be chosen to achieve the best result in that time. Doing the experiments that way would provide slightly more actionable insights but is also more complicated to implement.
It would also be interesting to investigate the correlation of the parameters. Here we investigated how each parameter influences a given metric on it's own, but table 5.10 suggests that $\varepsilon = 1$ can also work well, but requires a larger learning rate than smaller ε. This and other potential correlations cannot be seen in tables like 5.12.
There are also some parameters we did not investigate here, some of them are the Adam decay parameters β_1 and β_2, but we could also investigate how the number of available training samples and layer connectivity affect the results. Lastly it could be useful to check how well the findings from the two examples here generalize to other datasets.

8 Conclusion

In summary, we confirmed that the number of training epochs and batch size have a significant effect on the model's training time and final performance, but also found that loss functions and the Adam parameter ε can significantly affect the results too. Most other parameters also have a noticeable, but slightly smaller, effect on the final performance and varying effects on training time. In tables 6.1 and 6.2 we also summarized how the other parameters affect the measured metrics. It is recommended to focus optimization efforts on parameters with low impact on training time but high impact on final performance.
In section 6.4 we concluded that the cost of the proposed modification of Adam outweighs it's small benefits and is not recommended for use.
As a somewhat unexpected result we found that we could match the neural network performance reported in other sources ([LCB], [Ren19]) with much smaller networks. This shows how important good hyperparameters can be.

9 Acknowledgements

I thank my brother for providing a more capable laptop to run the experiments on. Franz Bethke sent me the ChemEx data as well as a piece of code to load that data and Thomas Kühne helped me to report the correct units used in the dataset to provide more complete information. Testing my variation of Adam would likely not have been possible without the example of a custom optimizer in Tensorflow by Evan Walters on GitHub.

10 Sources and references

References

[Cro] *Categorical crossentropy*. accessed: 2022-01-04 on peltarion.com. URL: https://bit.ly/3tNOttb.

[Dub+21] Shiv Ram Dubey et al. *diffGrad: An Optimization Method for Convolutional Neural Networks*. 2021. arXiv: 1909.11015 [cs.LG].

[Jos21] Sebastian Jost. "Einführung in den Optimierungsalgorithmus Adam". In: (2021). geschrieben im Kurs: Seminar: Mathematische Optimierung SoSe 2021.

[KB17] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (2017). arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/pdf/1412.6980.pdf.

[LCB] Yann LeCunn, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. last accessed: 2022-01-17. URL: http://yann.lecun.com/exdb/mnist/.

[Li+18] Chunyuan Li et al. "Measuring the Intrinsic Dimension of Objective Landscapes". In: (2018). arXiv: 1804.08838 [cs.LG].

[Ren19] Varadarajan Rengaraj. "A two-step machine learning for predicting the stability of chemical compositions". In: (2019).

[Sou] source code for this thesis can be found on GitHub:

[Tfa] *Tensorflow Adam implementation*. last accessed: 2021-11-05. URL: https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.

[Tfd] *Tensorflow documentation*. last accessed: 2022-01-19. URL: https://www.tensorflow.org/api_docs/python/tf/keras.

[Wal] Evan Walters. *custom DiffGrad implementation*. last accessed: 2021-11-05. URL: https://github.com/evanatyourservice/diffGrad-tf/blob/master/diffgrad.py.

11 Appendix

11.1 Details for experiments

The experiments were performed on a laptop with Intel i7-8750h processor and a single Nvidia GTX 1050-Ti max-Q GPU running Python 3.7 and Tensorflow version 2.6.
The data analysis and generation of the tables was performed with Python 3.8.
The code used for all experiments can be found in [Sou].

Total computation time

Training all 2592 models for the MNIST dataset five times took about two days.
Training the 648 models for the ChemEx dataset five times took about 3.5 days.
The summary of the results and tables shown in sections 5 and 11.2 can be computed in just a few minutes.

11.2 Optimized cAdam comparison

Here we list the experiment results for comparing Adam, the cAdam variant used in the previous experiments and "fast cAdam" (abbreviated as "f cAdam") - the cAdam variant with the optimizations described in Eq. (5) and Eq. (6).

Tables regarding training time

parameter name	best values				
training time	13.848	13.918	13.99	14.016	14.231
neurons per layer	(50, 10)	(50, 10)	(50, 10)	(50, 10)	(50, 10)
activation functions	ReLU	ReLU	ReLU	ReLU	ReLU
last activation function	sigmoid	sigmoid	sigmoid	sigmoid	softmax
loss function	cat-cross	cat-cross	cat-cross	cat-cross	cat-cross
training data percentage	0.3	0.3	0.3	0.3	0.3
number of epochs	50	50	50	50	50
batch size	100	100	100	100	100
optimizer	Adam	Adam	Adam	Adam	Adam
learning rate	0.1	0.1	0.001	0.001	0.001
ε	1.0	10^{-7}	1.0	10^{-7}	1.0

Table 11.1: best settings regarding *training time* for the MNIST dataset

parameter name	worst values				
training time	83.745	83.803	84.063	84.15	84.494
neurons per layer	(50, 10)	(50, 10)	(50, 10)	(50, 10)	(50, 10)
activation functions	ReLU	ReLU	ReLU	ReLU	ReLU
last activation function	softmax	sigmoid	sigmoid	softmax	softmax
loss function	cat-cross	cat-cross	cat-cross	cat-cross	cat-cross
training data percentage	1.0	1.0	1.0	1.0	1.0
number of epochs	50	50	50	50	50
batch size	100	100	100	100	100
optimizer	cAdam	cAdam	cAdam	cAdam	cAdam
learning rate	0.1	0.001	0.1	0.001	0.1
ε	1.0	1.0	10^{-7}	10^{-7}	10^{-7}

Table 11.2: worst settings regarding *training time* for the MNIST dataset

parameter name	parameter values			win ratios in %			avg. differences in s			best value
last activation function	softmax	sigmoid		8.3	91.7		0.476	0.106		sigmoid
training data percentage	0.3	1.0		100.0	0		0	46.589		0.3
optimizer	Adam	cAdam	f cAdam	100.0	0	0	0	25.801	20.858	Adam
learning rate	0.1	0.001		54.2	45.8		0.137	0.273		0.1
ε	1.0	10^{-7}		62.5	37.5		0.085	0.344		1.0

Table 11.3: parameter influence regarding *training time* for the MNIST dataset

Table regarding accuracy

parameter name	parameter values			win ratios in %			avg. differences			best value
last activation function	softmax	sigmoid		54.2	45.8		0.006	0.009		softmax
training data percentage	0.3	1.0		12.5	87.5		0.036	0.004		1.0
optimizer	Adam	cAdam	f cAdam	31.2	43.8	25.0	0.009	0.008	0.018	cAdam
learning rate	0.1	0.001		50.0	50.0		0.405	0.057		0.001
ε	1.0	10^{-7}		50.0	50.0		0.057	0.406		1.0

Table 11.4: parameter influence regarding *test accuracy* for the MNIST dataset

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig erstellt und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen, einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.