PRAGUE UNIVERSITY OF ECONOMICS AND BUSINESS

Faculty of finance and accounting

Department of Banking and Insurance

MASTER‘S THESIS

2024 Miroslav Holub

PRAGUE UNIVERSITY OF ECONOMICS AND BUSINESS

Faculty of Finance and Accounting

Department of Banking and Insurance

Field of study: Financial engineering

Application of machine learning for CVA calculation

Thesis author:	Miroslav Holub
Thesis supervisor:	prof. RNDr. Jiří Witzany, Ph.D.
Year of submission:	2024

Declaration of Authorship

I hereby declare that I have worked on my master's thesis titled " Application of machine learning for CVA calculation " independently, using only the sources and literature listed in the bibliography at the end of this thesis.

Prague, August 2024

....

Miroslav Holub

Ackowledgement

I would like to express my gratitude to prof. RNDr. Jiří Witzany, Ph.D. for his supervision of this thesis and also to Dr. ir. L.A. Lech Grzelak. for his valuable comments and feedback.

Abstract

This thesis investigates the application of neural networks, specifically Adam and AdamW networks, to accelerate the Credit Valuation Adjustment (CVA) calculation process, traditionally performed using Monte Carlo methods. The goal is to significantly reduce computational time while maintaining accurate results. By leveraging the predictive capabilities of neural networks, this research aims to optimize financial computations in CVA assessments. A dataset of synthetic Interest Rate Swaps (IRS) was generated, varying in selected parameters. The neural networks were trained to predict the Expected Positive Exposure (EPE). Results show that neural networks can achieve comparable accuracy to Monte Carlo methods, with significantly faster computation times. However, the process of generating artificial datasets and training the neural network is time-consuming. The findings contribute to the ongoing efforts in financial risk management using advanced machine learning techniques.

Key words

Counterparty Credit risk, Credit Valuation adjustment, xVA, Monte Carlo, Neural Networks, Deep learning, Adam, AdamW

Table of contents

Introduction

In the field of financial risk management, the calculation of Credit Valuation Adjustment (CVA) and other xVAs is a computationally intensive task, primarily due to the reliance on Monte Carlo methods. These methods require extensive simulations, making the process time-consuming. This thesis investigates the application of neural networks, specifically Adam and AdamW networks, to speed up the CVA calculation process. By leveraging the predictive capabilities of these neural networks, the goal of this thesis is to determine whether it is possible to significantly reduce computational time while maintaining accurate results. This thesis aims to contribute to the ongoing efforts to optimize financial computations using advanced machine learning techniques, potentially transforming the efficiency of CVA assessments in the financial industry. Through this exploration, we hope to find practical solutions that can streamline financial risk management practices and improve overall performance.

1 Counterparty Credit risk (CCR)

Counterparty credit risk is a form of risk inherent in financial transactions such as those involving financial derivatives in the OTC markets. It pertains to the possibility that our counterparties may not fulfill their obligation to provide the expected payoff due to either default or an unwillingness to meet their financial commitments.

1.1 Credit Valuation Adjustment (CVA)

An essential valuation adjustment associated with Counterparty Credit Risk (CCR) is known as Credit Valuation Adjustment (CVA). It refers to the difference between the market value of a transaction under the assumption of a theoretically risk-free counterparty and the market value of the identical transaction with respect to specific risky counterparty.(Witzany 2020, 203)

$$CVA = f_{nd} - f_{def}$$

Where:

CVA = Value of adjustment of counterparty credit risk under risk-neutral probability measure.

f_{nd} = Non-default valuation of derivative. Derivative value calculated by pricing method under risk-neutral world (e.g. Black-Scholes model)

f_{def} = Market value of derivative in real world.

Theoretical CVA value can also be expressed as expectation under risk-neutral measure of discounted CCR loss:

$$CVA = E_{\mathbb{Q}}[discounted\ CCR] = E_{\mathbb{Q}}[e^{-r\tau} \times \max(f_\tau, 0) \times LGD \times I(\tau \leq T)]$$

Where:

$E_{\mathbb{Q}}$ = Expectation under risk-neutral measure over τ

r = Risk-free rate in continuous compounding

τ = Stochastic time of default of counterparty (if counterparty can't default $\tau = +\infty$)

$\max(f_\tau, 0)$ = Non-negative market value of derivative contract at time of default

LGD = Stochastic fractional Loss Given Default coefficient

$I(\tau \leq T) = \begin{cases} 1\ if\ \tau \leq T \\ 0\ otherwise \end{cases}$

The previously mentioned definition of CVA is notably intricate as it accounts for the potential mutual correlation of variables and the stochastic nature of Loss Given Default (LGD). Typically, Wrong-way Risk or Right-way Risk is also considered. However, for the sake of simplicity, we can adopt an alternative discretized formula for CVA, which incorporates a deterministic LGD (Witzany 2020, 204):

$$CVA \doteq LGD \times \sum_{j=1}^{m} e^{-r(t_j)\times t_j} \times EE(t_j) \times q(t_j) \times \Delta t_j$$

Where:

$r(t_j)$ = Discounted rate at time t_j

$EE(t_j) = E\left[\max\left(f_{t_j}, 0\right)\right]$ = Expected exposere independent on other factors

$q(t_j)$ = Counterparty default intensity at time t_j

$\Delta t_j = t_j - t_{j-1}$

$q(t_j) \times \Delta t_j$ = Probability of counterparty's default during time $[t_{j-1}, t_j)$

Further simplification can be obtained by introducing the concept of Expected Positive Exposure (EPE), which is defined as the average of the Expected Exposure (EE) over time:

$$EPE \doteq \frac{1}{T} \times \sum_{j=1}^{m} EE(t_j) \times \Delta t_j$$

Additionally, with a constant default intensity $\bar{q}$, the CVA can be approximated as:

$$CVA \doteq \bar{q} \times \text{LGD} \times \text{EPE} \times e^{-r(t_j)\times t_j}$$

(Witzany 2020, 206)

Expected exposure at particular time t_j is in general estimated by single or double Monte Carlo simulation method. Single method is used if have an analytical or semi-analytical formula for value f_{t_j} (e.g. forwards, swap) and we're just using Monte Carlo simulation for generating different future non-negative exposures and than making average of them. In case of double Monte Carlo simulation we also have to calculate our derivative value with use of Monte Carlo for every future possible scenario.(Witzany 2020, 204)

1.2 Bilateral Credit Valuation Adjustment (BCVA)

When negotiating prices with a counterparty, it's important to consider that they may have a different Credit Valuation Adjustment (CVA) with us as well. Therefore, it becomes necessary to define Bilateral Credit Valuation Adjustment (BCVA) to appropriately account for this (Witzany 2020, 211):

$$BCVA = CVA_C - CVA_I$$

Where:

$CVA_C = E_{\mathbb{Q}}\left[e^{-r\tau} \times \max\left(f_{\tau_C}, 0\right) \times LGD_C \times I(\tau_C \leq T \, \&\tau_C \leq \tau_I)\right]$

$CVA_I = DVA = E_{\mathbb{Q}}\left[e^{-r\tau} \times \max\left(f_{\tau_I}, 0\right) \times LGD_C \times I(\tau_I \leq T \, \&\tau_I \leq \tau_C)\right]$

In formula above CVA_C is basic CVA from our point of view which is just covering counterparty's possibility of default. On the contrary DVA also known as Debt Valuation adjustment is CVA from the counterparty's perspective because counterparty also hase CCR with us. As we can see DVA is benefit for us because it is decreasing our CVA due to our possibility of default.(Witzany 2020, 211)

1.3 Other xVAs

There is many more so called xVAs mainly used for internal reporting and monitoring and because we will be interested mainly in CVA I'm going to slightly introduce following one:

Funding Valuation Adjustment (FVA) is adjustment to the value of a derivative (or portfolio of derivatives) designed to ensure that a institution will recover their average funding costs.(Forjan 2023)

Margin Valuation Adjustment (MVA) is adjustment which represents the cost of the posting intial margin to Central counterparty (CCP) over the lifetime of transaction.(Forjan 2023)

Capital Valuation Adjustment (KVA) is simply adjustment of costs of holding regulatory capital over the duration of the transaction.(Forjan 2023)

2 Neural nets and Deep learning

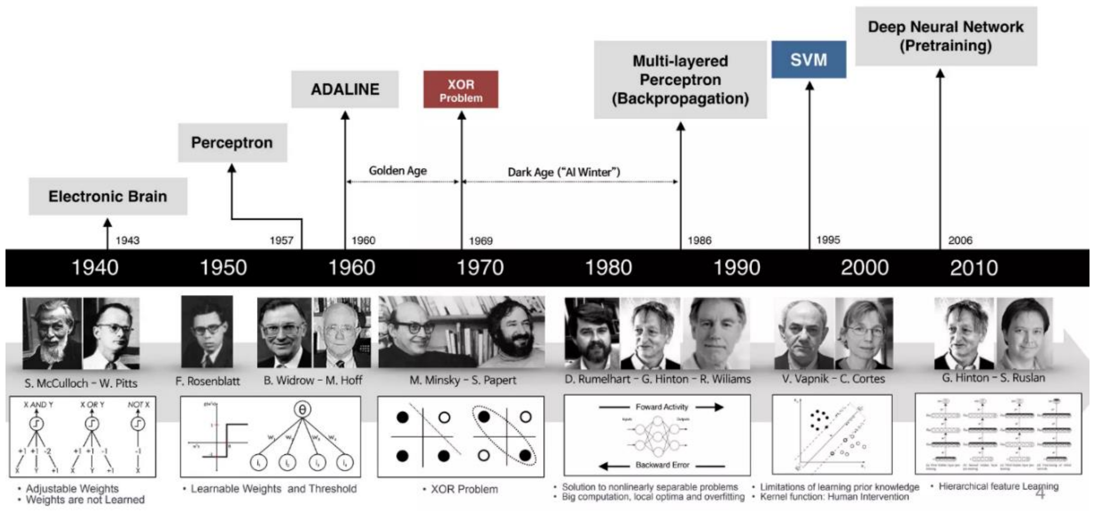
Neural networks are models inspired by the human brain, designed to learn patterns from data. They consist of interconnected layers of artificial neurons that process input data, make predictions, and adjust their parameters during training to minimize errors. Through this iterative process, neural networks can solve tasks such as classification, regression, and pattern recognition.

In this chapter, we are going to introduce the reader to neural networks and deep learning, which is essentially a multi-layered neural network, i.e., a neural network with more than one hidden layer. We will discuss the history of neural networks, how they work, the different types of architectures, which ones we will use in our project, how we can improve our neural network model, and much more.

History of Neural networks and Deep learning

Neural networks have evolved significantly since the 1940s. In 1943, McCulloch and Pitts introduced the first conceptual model of a neural network, considering only binary (0,1) inputs and outputs. Then, in 1957, Rosenblatt developed the perceptron, inspired by individual neurons in the visual cortex of cats. This was followed by Widrow and Hoff's ADALINE in 1960, which utilized error correction for learning. The field faced setbacks in 1969 when Minsky and Papert highlighted the perceptron's inability to solve the XOR problem in their book, leading to an "AI Winter." Interest was revived in 1986 with Rumelhart, Hinton, and Williams' backpropagation algorithm, enabling effective training of multi-layered perceptrons. The development of Support Vector Machines by Vapnik and Cortes in 1995 further advanced the field. Finally, in 2006, Hinton and Rulan's work on deep neural networks and pretraining techniques marked the beginning of modern deep learning. (Berka 2003, 163; Anon. 2015, 4)

Figure 1: Brief history of Neural network


Deep Neural Network (Pretraining)
SVM
Multi-layered Perceptron (Backpropagation)
XOR Problem
ADALINE
Perceptron
Electronic Brain
Golden Age
Dark Age ("AI Winter")
1943
1957
1960
1969
1986
1995
2006
1940
1950
1960
1970
1980
1990
2000
2010
S. McCulloch - W. Pitts
F. Rosenblatt
B. Widrow - M. Hoff
M. Minsky - S. Papert
D. Rumelhart - G. Hinton - R. Wiliams
V. Vapnik - C. Cortes
G. Hinton - S. Ruslan
• Adjustable Weights
• Weights are not Learned
• Learnable Weights and Threshold
• XOR Problem
• Solution to nonlinearly separable problems
• Big computation, local optima and overfitting
• Limitations of learning prior knowledge
• Kernel function: Human Intervention
• Hierarchical feature Learning

Source: (Anon. 2015, 4)

Universal Approximation Theorem

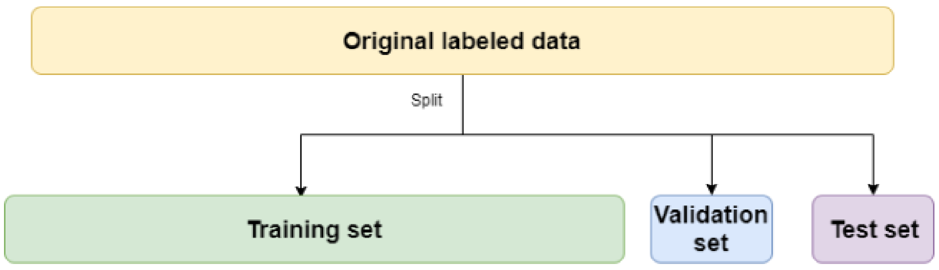
The Universal Approximation Theorem states that a neural network with at least three layers (input layer, hidden layer, and output layer) and non-linear activation functions can replicate any continuous function with the desired precision by increasing the number of hidden neurons. Therefore, it serves as an incredibly versatile tool capable of modeling almost any relationship between inputs and outputs, provided we allocate enough neurons to the hidden layer. (Berka 2003)

$$(\forall\, f(x), f(x) \text{ is continuous})(\forall \varepsilon > 0)(\exists g(x))(|g(x) - f(x)|)$$

Data Splitting

Data splitting plays a crucial role in all supervised machine learning, including neural networks. To effectively train our model without underfitting or overfitting, we follow these steps: First, we divide our dataset into training, validation, and test sets, initializing parameters (weights and biases) and hyperparameters (like learning rate, momentum coefficient, etc.). Second, we train our model by adjusting parameters using backpropagation algorithm on the training set, aiming to minimize our chosen loss function (e.g., MSE). Third, we evaluate our optimized model on the validation set and adjust hyperparameters if its performance significantly differs from the training set. Finally, we assess our model with the new hyperparameters on the test set.

Figure 2: Data Splitting



Source: (Juan 2020)

Activation Functions

The activation function, commonly differentiable if we are using gradient-based training algorithms, is a linear or non-linear transformation of affine inputs from the preceding layer. These functions are applied solely to the hidden and output layers. Non-linear transformations are employed in the hidden and output layers to enable neural networks to address problems that are not linearly separable and generally yield better results during NN training. Linear activation functions are not commonly used in practice, and if employed, they are typically used only on the output layer.

Table 1: Some of the most popular activation functions

Name	$f(x)$	$f'(x)$
Identity	x	1
Binary Step	$\begin{cases} 0\ if\ x < 0 \\ 1\ if\ x \geq 0 \end{cases}$	$\begin{cases} 0\ if\ x < 0 \\ 0\ if\ x \geq 0 \end{cases}$
Sigmoid	$\frac{1}{1+e^{-x}}$	$f(x) \times (1 - f(x))$
Tanh	$\frac{e^x - e^{-x}}{e^x - e^{-x}}$	$1 - f(x)^2$
ReLU	$\max(0, x)$	$\begin{cases} 0\ if\ x \leq 0 \\ 1\ if\ x > 0 \end{cases}$
Leaky ReLU	$\begin{cases} x\ if\ x > 0 \\ \alpha x\ if\ x \leq 0 \end{cases}$	$\begin{cases} 1\ if\ x > 0 \\ \alpha\ if\ x \leq 0 \end{cases}$
Swish	$\frac{x}{1+e^{-x}}$	$f(x) \times \sigma(x) \times (1 - f(x))$
Softmax	$\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$	$\begin{cases} f(x_i) \times \left(1 - f(x_i)\right)\ if\ i = j \\ -f(x_i) \times f\left(x_j\right)\ if\ i \neq j \end{cases}$

Source: Own work

In the past, the most commonly used activation function was the sigmoid, primarily because it was used to model a Bernoulli distribution, representing the probability of outcomes. However, it became evident that the derivative of the sigmoid, often utilized in the backward pass (except for the Cross Entropy loss function), proved impractical. The main issue is that the sigmoid's derivative is typically very small. Since we often multiply many derivatives during backpropagation, this leads to the vanishing gradient problem, which we will discuss later. (Goodfellow et al. 2016; Michael 2019) Additionally, even in the forward pass, the sigmoid is not ideal because it becomes too flat at the extremes. Consequently, in this case even a large change in weights doesn't have much effect, resulting in a long learning process.(Grant 2017)

Currently, the most popular activation function in modern NN architectures is ReLU. Despite its unconventional appearance, it can approximate any function with desired precision given a sufficient number of hidden nodes. Its main benefit lies in significantly reducing the probability of gradient vanishing and avoiding flatness in extremes. Moreover, due to its simplicity, it is computationally more efficient than sigmoid. (Jason 2020) In practice, architectures with ReLU as the activation function demonstrate better convergence performance than sigmoid. (Krizhevsky et al. 2017, 3)

Finally, we will also introduce Softmax, a popular activation function commonly used in the output layer of neural networks. The main benefit of Softmax is its ability to nicely represent the probability of each node, ensuring that the sum of probabilities adds up to 1. This property makes it particularly useful for multi-class classification tasks, where the output needs to be interpreted as probabilities across different classes. Unlike Softmax, which ensures that the sum of all function values in the output layer adds up to 1 to form a valid probability distribution, sigmoid activation functions do not enforce this requirement. Each node's output value independently ranges between 0 and 1 in sigmoid, without needing the sum of these values across all nodes to equal 1. (Michael 2019) Nevertheless, Softmax also faces a limitation in that it tends to require a lot of parameters, especially as the number of classes increases. This can cause overfitting issues and make it harder for the model to perform well on new, unseen data, particularly when dealing with datasets that have many classes.(Bismi 2023)

Loss Functions

Every neural network model is usually trained to minimize a so-called loss function, also known as a cost function. In practice, there are many loss functions that can be used for different tasks such as regression or classification. A common principle used in designing a loss function is the maximum likelihood principle. This principle states that if we independently draw training data from a true but unknown data-generating distribution $p_{data}(\mathbf{x})$ and define $p_{model}(\mathbf{x};\boldsymbol{\theta})$ as a parametric family of probability distributions over the same $\mathbf{x}$ with various $\boldsymbol{\theta}$, we can obtain the desired model by maximizing the likelihood function over many probabilities $p_{model}(\boldsymbol{x}^{(i)};\boldsymbol{\theta})$. This maximization causes the minimization of dissimilarities between the empirical distribution $\hat{p}_{data}(\mathbf{x})$ defined by training set and the model distribution $p_{data}(\mathbf{x})$. (Goodfellow et al. 2016, 128). Since there are many loss functions I'm going to mention just few of them which I consider as most used in practice. Also take into consideration that for sake of simplicity I'm going to assume that we have just one target variable for each training exaple (i.e. one output node).

First, I mention the old but still widely used, especially for regression problems, Mean Square Error (MSE), which assumes that the model computes $f(\mathbf{x};\boldsymbol{\theta})$ using parameters $\boldsymbol{\theta}$ and given by N examples $(\boldsymbol{x}^{(1)},t^{(1)}),(\boldsymbol{x}^{(2)},t^{(2)}),\ldots(\boldsymbol{x}^{(N)},t^{(N)})$. Loss function is than(Straka 2024, 10):

$$C(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{i=1}^{N}\left(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right)^2$$

Second, a commonly used cost function for classification is Cross-Entropy. It often results in better convergence during training due to its properties. Especially if we are using cross-entropy as the loss function in combination with the sigmoid activation function, we'll effectively get rid of explicit derivatives of the sigmoid function during backpropagation. Cross-Entropy loss also has a probabilistic interpretation, as it measures the difference between the true probability distribution (one-hot encoded labels) and the predicted probability distribution (output of the activation function). This makes it especially effective for training models to output accurate probabilities.(Michael 2019) We can define the Binary Cross-Entropy loss function as:

$$C(\boldsymbol{\theta}) = -\frac{1}{N}\sum_{i=1}^{N}\left[t^{(i)} \ln f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) + \left(1 - t^{(i)}\right) \ln(1 - f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))\right]$$

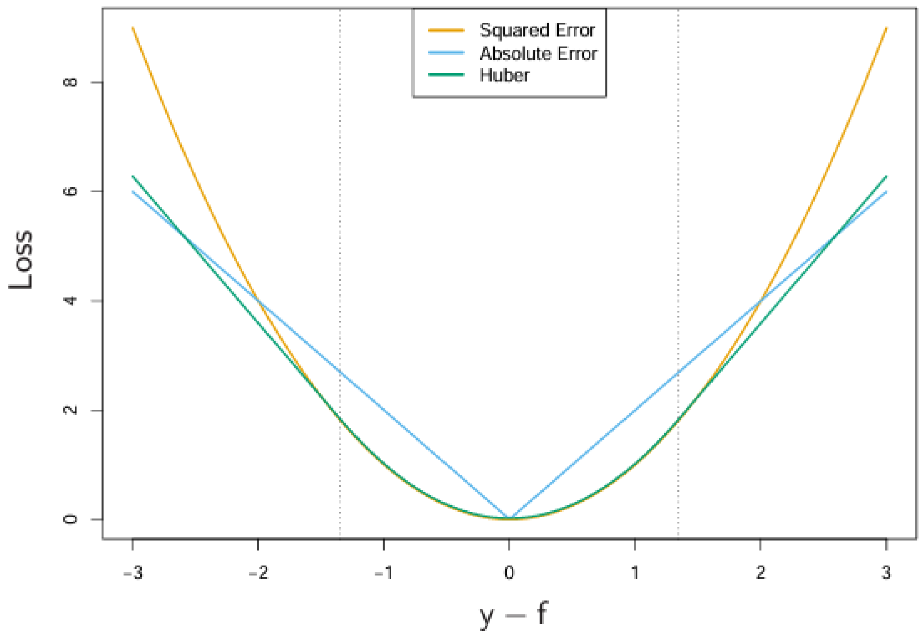
Finally, we will introduce another useful function for regression, which combines the strengths of Mean Square Error and Mean Absolute Error, $\frac{1}{N}\sum_{i=1}^{N}\left|f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right|$. This function is designed to be less sensitive to outliers than MSE but more sensitive to outliers than MAE, providing a more robust evaluation metric in the presence of anomalous data points. (Hastie et al. 2009) We can define the Huber loss function as:

$$C(\boldsymbol{\theta}) = \sum_{i=1}^{N}\begin{cases} \frac{1}{2}\left(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right)^2 & if\ \left|f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right| \leq \delta \\ 2\delta\left|f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right| - \delta^2 & if\ \left|f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - t^{(i)}\right| > \delta \end{cases}$$

Where:

δ = threshold hyperparameter determining the point where the loss function for i-th example transitions from quadratic to linear

Figure 3: A comparison of three loss functions for regression, plotted as a function of the margin y−f.



Source: (Hastie et al. 2009, 350)

2.1 Neural Networks Architectures

Before we discuss different neural network architectures, it is important to categorize neural networks into three main types: Feed-Forward Neural Networks (FFNNs), Recurrent Neural Networks (RNNs), and Unsupervised Neural Networks (UNNs). In an FFNN, information flows in one direction, from the input layer through one or more hidden layers to the output layer, without any feedback loops. This unidirectional flow allows FFNNs to serve as universal approximators of nonlinear functions. In contrast, RNNs are designed to process sequences of data by introducing connections that loop back from later nodes in the sequence to earlier ones, or within the same layer, creating a form of memory within the network. As a result, RNNs are effective universal approximators of dynamic systems. (Fičura 2023, 6) UNNs are networks primarily used for tasks such as clustering, dimensionality reduction, or generative modeling.

Now if you understood the key difference between FFNNs, RNNs and UNNs we can simply characterize main NN achitechtures as:

FFNN

- Multilayer Perceptron (MLP)
    - MLP is a general type of FFNN that contains multiple layers (usually one or more hidden layers) of neurons with non-linear activation functions.
- Convolutional Neural Network (CNN)
    - CNN is a type of deep FFNN typically used for analyzing images. It performs feature extraction through operations such as convolution and pooling, followed by classification.
- Radial Basis Function Network (RBFN)
    - RBFN is a type of FFNN that utilizes radial basis functions as activation functions $f(x) = e^{-x^2}$ to process input data and produce output. (Fičura 2023, 8)
- Transformer

    - Transformer is a type of deep FFNN with an attention mechanism, typically used in both small and large language models. (Fičura 2023, 23)

RNN

- Long Short-Term Memory Network (LSTM)
    - LSTM is a NN designed to better retain long-term and short-term dependencies in input data through special cells and gates that control the flow of information.
- Gated Recurrent Unit (GRU)
    - GRU is a similar type of NN to LSTM but simpler, with fewer parameters. It is mainly used to retain long-term dependencies in data.
- Elman Neural Network (ENN)
    - ENN is a type of RNN that has additional inputs from the hidden layer, forming a new context layer. (Ren et al. 2018, 2)
- Echo State Network (ESN)
    - ESN is a type of RNN that simplifies ENN by generating a large hidden layer randomly and training only the readout. (Fičura 2023, 19)

UNN

- Kohonen Self-Organizing Map (SOM)
    - SOM is a special type of NN that has only two layers: the input layer and the feature map (Kohonen grid). The task of the learning process is to assign all examples to individual neurons in the feature map. It is usually used for clustering and visualization. (Berka 2003, 169)
- General Adversarial Network (GAN)
    - GAN is a type of NN comprising two adversarial networks: the generator and the discriminator. These networks collaborate to generate data that looks realistic. (Fičura 2023,26)
- Autoencoder (AE)
    - AE is a NN usually used for non-linear dimensionality reduction. The NN is trained to recreate the initial inputs, and the hidden layer (encoded data) is finally used as our features in lower dimension. (Fičura 2023, 26)

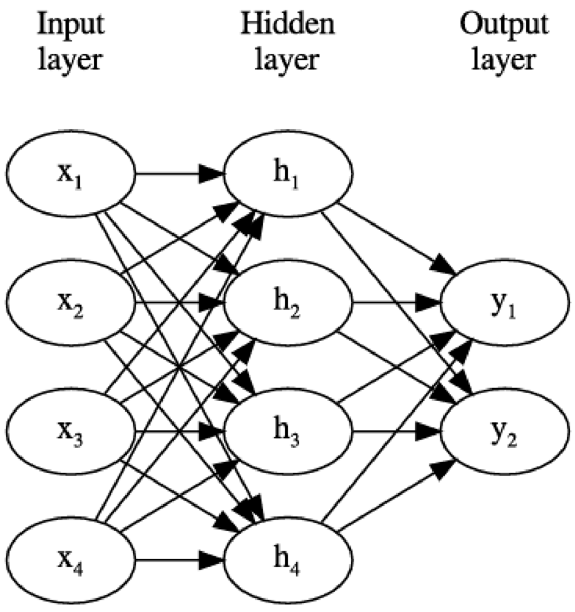
- Variational Autoencoder (VAE)
    - VAE is type of AE that generates data by learning the probabilistic distribution of latent variables allowing it to generate new samples from these distributions.(Singh Choudhary 2023)

As you can see, there are numerous types of NNs, each with various variants, and we didn't even mention all of them. We'll focus our exploration on FFNN, which we'll employ for calculating expected positive exposure. But don’t worry, we won’t just look at simple designs of architectures. Instead, we'll delve deeper, exploring various network variants, as well as different regularization and optimization techniques we can use.

2.1.1 Multilayer Perceptron (MLP)

MLP is the most well-known FFNN, which can be used in various domains and can be adjusted in many ways according to our needs. To start simply, we are going to introduce the basic MLP with a single hidden layer. The input layer serves just as a placeholder for our features, while the hidden and output layers have active nodes (i.e each node uses an activation function) that serve as non-linear transformations of the preceding nodes' values.

Figure 4: Single-Hidden-Layer MLP



Source: (Straka 2024a, 3)

Before training our neural network, we need to initialize parameters $\boldsymbol{\theta}$ (weights and biases) and perform one forward pass. During the forward pass, we gradually calculate the activation functions for each node from the hidden to the output layer. The activation function in the i-th neuron i in the hidden layer is used as follows if we process our data in batch b:

$$H_{b,i} = f^{(1)}\left(\sum_j X_{b,j} W_{i,j}^{(1)} + b_i^{(1)}\right)$$

This can be written in tensorized form as follows:

$$\boldsymbol{H} = f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)$$

Where:

$\boldsymbol{X} \epsilon\ \mathbb{R}^{BxNxJ}$ = Input tensor of B batch matrices $X_{b,j}$, where each batch matrix consists of N vectors of input examples with J input variables.

$\boldsymbol{W}^{(1)} \epsilon\ \mathbb{R}^{DxH}$ = Matrix of weights (Kernel) between input layer with D neurons and hidden layer with H neurons.

$\boldsymbol{b}^{(1)} \epsilon\ \mathbb{R}^{H}$ =Vector of biases on hidden layer $\left(b_i^{(1)} = -threshold_i^{(1)}\right)$.

$\boldsymbol{Z}^{(1)} = \boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}$ = Input to the activation function in the hidden layer, which is an affine transformation.

$f^{(1)}$ = Activation function in hidden layer.

Similarly, we can proceed for the output layer:

$$Y_{b,i} = f^{(2)}\left(\sum_j H_{b,j} W_{i,j}^{(2)} + b_i^{(2)}\right)$$

In tensorized form, we get:

$$\boldsymbol{Y} = f^{(2)}\left(\boldsymbol{H}\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right) = f^{(2)}\left(f^{(1)}\left(\boldsymbol{X}\boldsymbol{W}^{(1)} + \boldsymbol{b}^{(1)}\right)\boldsymbol{W}^{(2)} + \boldsymbol{b}^{(2)}\right)$$

Where:

$\boldsymbol{H} \epsilon\ \mathbb{R}^{BxNxJ}$ = Hidden tensor of B batch matrices $H_{b,j}$, where each batch matrix consist of N vectors of input examples with J input variables.

$\boldsymbol{W}^{(2)} \epsilon\ \mathbb{R}^{HxO}$ = Matrix of weights (Kernel) between hidden layer with H neurons and output layer with O neurons.

$\boldsymbol{b}^{(2)} \epsilon\ \mathbb{R}^{H}$ = Vector of biases on output layer $\left(b_i^{(2)} = -\,threshold_i^{(2)}\right)$.

$\boldsymbol{Z}^{(2)} = \boldsymbol{HW}^{(2)} + \boldsymbol{b}^{(2)}$ = Input in activation function in the output layer, which is in affine transformation.

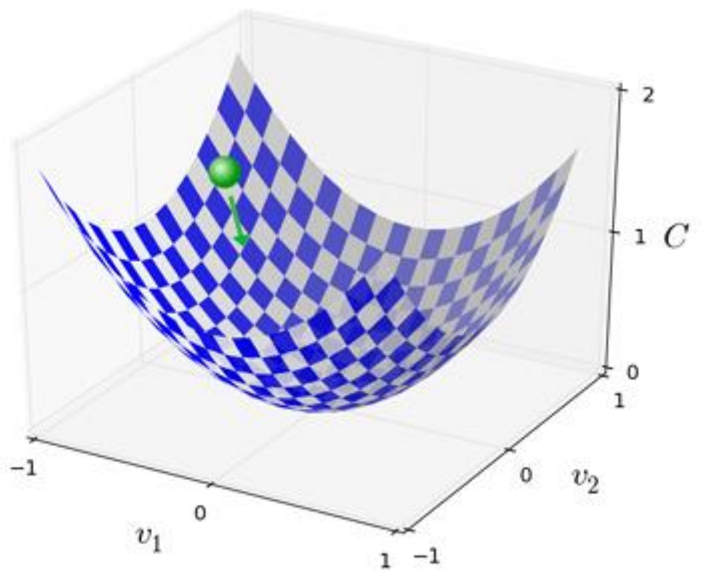
$f^{(2)}$ = Activation in output layer.

This knowledge is all we need to understand how to perform forward propagation in a NN, i.e., the way to move from the input to the output layer. This is crucial for comparing our calculated outputs with true outputs via a loss function.

2.1.1.1 Backpropagation of error

If our NN uses differentiable activation functions, we can train it using a algorithm called backpropagation, which relies on the popular gradient descent (GD) method. The goal of gradient descent is to find the local minimum of a function (if function is convex, it is global minimum). This is done by calculating the gradient of the function with respect to its input variables and then adjusting these variables proportionally over many iterations.

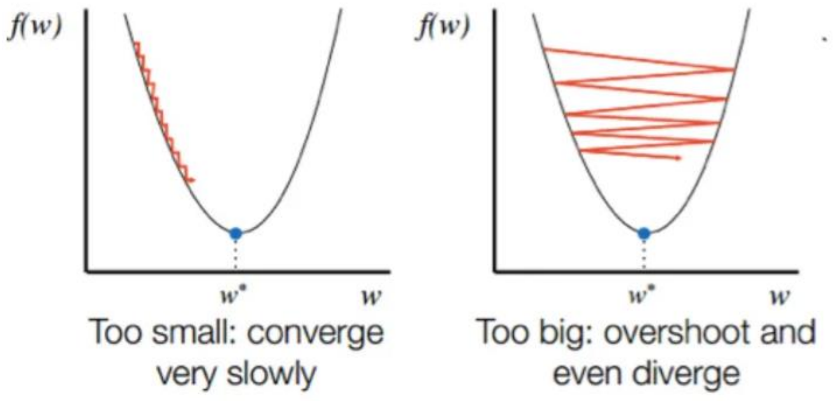
Figure 5: Visualisation of GD method applied on function $C(v_1, v_2)$



Source: (Nielsen 2019b)

An important hyperparameter of gradient descent is the step size, which determines how quickly we approach the local minimum and whether we can find it. If the step size is too large, we might overshoot the local minimum and never find it. Conversely, if the step size is too small, the learning process will be slow.

Figure 6: Implication of small and high step size



Source: (Kansal 2020)

The main goal of the backpropagation algorithm is to find parameters $\boldsymbol{\theta}$ that minimize the cost function C($\boldsymbol{\theta}$). For now, I will present only the Stochastic Gradient Descent version with specified mini-batch, which is a basic and most used form of backpropagation algorithm. There are many other variations of this algorithm, and I will introduce some of them later.

Mini-batch SGD Backpropagation

Input:

- Set training examples $(\boldsymbol{x}, \boldsymbol{t})$ from training dataset
- Set number of batches B and corresponding number of training examples in each bach N
- Specify differentiable cost function C($\boldsymbol{\theta}$)
- Set number of epochs
- Set number of input nodes
- Set number of hidden layers and nodes in them
- Set number of output nodes
- Specify Learning rate α (similar to step size in GD) which can be later changed after assesing NN on validation dataset

Initialization:

- Initialize ale weights and biases (usually sampled from specified distribution)

Output: After Minibatch SGD Backpropagation algorithm run time we'll get final updated weights and biases

Training:

- Repeat epoch until stopping criterion is met:

Propagate the input forward throught the network:

1. Sample N training examples ($\boldsymbol{x}^{(i)}$, $\boldsymbol{t}^{(i)}$) forming the mini-batch b. In theory, we could sample each mini-batch independently. However, it is generally preferred to process all training instances before repeating them. This can be implemented by generating a random permutation of the training dataset and then splitting it into mini-batch-sized chunks.
2. Each instance $\boldsymbol{x}^{(i)}$ in batch b input to the NN and compute the output $\boldsymbol{y}^{(i)}$ of every unit node in the output layer
   - For each layer $l = 2, 3, \dots, L$ compute vectors of inputs $\boldsymbol{z}^{(i,l)}$ and vectors of activation function values $\boldsymbol{a}^{(i,l)}$:
     $\boldsymbol{z}^{(i,l)} = \boldsymbol{W}^{(l)} f\left(\boldsymbol{z}^{(l-1)}\right) + \boldsymbol{b}^{(l)}$;
     $\boldsymbol{a}^{(i,l)} = f(\boldsymbol{z}^{(i,l)}) = f\left[\boldsymbol{W}^{(l)} f\left(\boldsymbol{z}^{(l-1)}\right) + \boldsymbol{b}^{(l)}\right]$

Propagate the errors backward throught the network:

3. Compute the output error vector for batch b: $\boldsymbol{\delta}^{(i,L)} = \nabla_{\boldsymbol{a}} C_{\boldsymbol{b}} \odot f'(\boldsymbol{z}^{(i,l)})$, where $\nabla_A C_b$ is average gradient of the batch b with respect to vector of activation function values and $\odot$ is Hadamard product
4. Backpropagate the error for each $l = L-1, L-2, \dots, 2$ by computing:
   $\boldsymbol{\delta}^{(i,l)} = \left(\left(\boldsymbol{W}^{(l+1)}\right)\boldsymbol{\delta}^{(i,l+1)}\right) \odot f'(\boldsymbol{z}^{(i,l)})$
5. Update the network weights and biases for each $l = L, L-1, \dots, 2$:
   $\boldsymbol{W}^{(l)} \leftarrow \boldsymbol{W}^{(l)} - \frac{\alpha}{N}\sum_{i=1}^{\mathrm{N}} \boldsymbol{\delta}^{(i,l)}\left(\boldsymbol{a}^{(i,l-1)}\right)^{T}$;
   $\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \frac{\alpha}{N}\sum_{i=1}^{N} \boldsymbol{\delta}^{(i,l)}$

(Nielsen 2019a; Mitchell 1997, 98; Straka 2024b, 28)

If I were to change the mini-batch size to one, meaning I sample just one training example at a time, I would then be using the well-known Online backpropagation algorithm. This method is unbiased but tends to be very noisy. Mini-batch SGD strikes a balance between Online SGD and Standard Gradient Descent backpropagation. In Standard Gradient Descent, we use all training data to compute the gradient of the cost function, which makes learning quite slow. Mini-batch SGD, on the other hand, offers a trade-off by providing faster learning than Standard Gradient Descent while being less noisy than Online SGD. It's also worth mentioning that, in practice, we do not usually represent the learning process in this way but rather as a collection of tensor functions. However the backpropagation algorithm is analogous. (Straka 2024b, 19, 26)

Since I will discuss other adjustments to backpropagation in the upcoming chapters and it could be confusing to modify the entire algorithm, I will simplify the aforementioned explanation for now.

Simplified Mini-batch SGD Backpropagation

Input:

- NN computing function $f(\boldsymbol{x};\, \boldsymbol{\theta})$ with initial parameters $\boldsymbol{\theta}$ (sampled from specified distribution) during first forward pass.
- Learning rate α

Output: Updated parameters $\boldsymbol{\theta}$

Training:

- Repeat epoch until stopping criterion is met:
    - Sample N training examples forming the mini-batch b.
    - Compute gradient estimate of loss function with respect to parameters $\boldsymbol{\theta}$ for mini-batch b:
      $\hat{\boldsymbol{g}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\, C(f\big(\boldsymbol{x}^{(i)};\, \boldsymbol{\theta}\big),\, \boldsymbol{t}^{(i)})$
    - Update parameters:
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\hat{\boldsymbol{g}}$

(Goodfellow et al. 2016, 286)

Finally, I want to mention that we can also adjust the learning rate α_i during the learning process to accelerate convergence to the minimum. By adjusting the learning rate dynamically during training, such as decreasing it when progress slows down or increasing it when progress is rapid, we can potentially speed up convergence. It can also be proven that if the loss function is convex and continuous, then the Mini-batch SGD backpropagation algorithm will almost surely converge to the global optimum if the sequence of learning rates α_i fullfills the following conditions:

$$\sum_{k=1}^{\infty} \alpha_i = \infty; \sum_{k=1}^{\infty} {\alpha_i}^2 \ \leq \infty$$

For nonconvex loss functions, we can only guarantee convergence to a local minimum.

(Goodfellow et al. 2016, 287; Straka 2024b, 20)

2.1.1.2 Other optimization techniques

Beyond backpropagation, there exist other adjusted versions of the algorithm and entirely different optimization methods that can be employed when training our NN. Some of these techniques are tailored to converge faster to the minimum of the loss function, while others are designed to handle non-differentiable loss functions or to search for the global minimum. This diversity of optimization methods allows practitioners to choose the most suitable approach based on the specific characteristics of their NN and the requirements of the problem at hand.

2.1.1.2.1 Hessian technique

To improve the GD or SGD backpropagation algorithm by incorporating second-order derivative information of the cost function during backpropagation, we can use the Hessian matrix H, provided it is positive definite. Parameter updates are then:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha H^{-1} \hat{\boldsymbol{g}}$$

In theory, this should lead to fewer steps due to the incorporation of second-order derivatives. However, in deep learning, this approach is quite impractical because NNs have many parameters, and computing the large Hessian matrix in every epoch is quite time-consuming.(Michael 2019)

2.1.1.2.2 Momentum-based method

Based on similar intuition as the Hessian technique but avoiding the computation of the Hessian matrix in each epoch is the so-called Momentum-based GD or Momentum-based SGD backpropagation. Here I'll show just Momentum-based SGD, but the same adjustments also apply to GD. In this adjusted backpropagation algorithm, we use another hyperparameter called the momentum coefficient β and a velocity vector $\boldsymbol{v}$. The momentum coefficient controls the amount of damping or friction in the system by $(1-\beta)$. If $\beta = 1$ there is no friction, and we increase the speed of convergence by repeatedly adding gradient estimates. If $\beta = 0$, there is no increasing velocity, and the algorithm is the same as basic SGD backpropagation. (Michael 2019)

Mini-batch SGD Backpropagation with Momentum

Input:

- NN computing function $f(\boldsymbol{x};\, \boldsymbol{\theta})$ with initial parameters $\boldsymbol{\theta}$ (sampled from specified distribution) during first forward pass.
- Learning rate α, Momentum β

Output: Updated parameters $\boldsymbol{\theta}$

Training:

- Repeat epoch until stopping criterion is met:
    - Sample N training examples forming the mini-batch b.
    - Compute gradient estimate of loss function with respect to parameters $\boldsymbol{\theta}$ for mini-batch b:
      $\hat{\boldsymbol{g}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\, C(f(\boldsymbol{x}^{(i)};\, \boldsymbol{\theta}),\, \boldsymbol{t}^{(i)})$
    - Compute current velocity vector:
      $\boldsymbol{v} \leftarrow \beta\boldsymbol{v} - \alpha\hat{\boldsymbol{g}}$
    - Update parameters:
    - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$

(Goodfellow et al. 2016, 289)

2.1.1.2.3 Adaptive learning rate optimization algorithms

There are several adaptive learning rate optimization algorithms, including AdaGrad, RMSProp, and Adam. In this explanation, I will focus on Adam, which I consider as one of the best and most popular due to its advantageous characteristics. Adam, short for Adaptive Moment Estimation, is a widely-used optimization algorithm in deep learning. It combines momentum, a technique that accelerates optimization, with adaptive learning rates. These adaptive rates adjust how much the model learns from each parameter update, particularly useful for complex optimization tasks. Adam computes unique learning rates for each parameter based on gradient history, enhancing its ability to find optimal solution. This adaptability improves stability and convergence speed during training. Additionally, Adam includes bias corrections to enhance performance, especially at the start of training. Due to its effectiveness in handling various optimization challenges, Adam is widely regarded as one of the most effective and popular optimization algorithms in deep learning. (Goodfellow et al. 2016, 301)

Adam

Input:

- NN computing function $f(\boldsymbol{x}; \boldsymbol{\theta})$ with initial parameters $\boldsymbol{\theta}$ (sampled from specified distribution) during first forward pass.
- Learning rate α (default 0.001), Momentum β_1 (default 0.9), Momentum β_2 (default 0.999), Numerical stabilization constant ε (usually 10^{-8})

Output: Updated parameters $\boldsymbol{\theta}$

Training:

- Initialize first and second moment variables (adaptive learning rates):
  $\boldsymbol{s} = 0, \boldsymbol{r} = 0$
- Initialze time step $t = 0$
- Repeat epoch until stopping criterion is met:
  - Sample N training examples forming the mini-batch b.
  - Compute gradient estimate of loss function with respect to parameters $\boldsymbol{\theta}$ for mini-batch b:

$\hat{\boldsymbol{g}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} C(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{t}^{(i)})$

- Compute current time step:

  $t \leftarrow t + 1$

- Update biased first moment estimate error (mean):

  $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} - (1 - \beta_1)\hat{\boldsymbol{g}}$

- Update biased second moment estimate error (variance):

  $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} - (1 - \beta_2)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}}$

- Compute unbiased estimates of the moments, i.e. correct bias in the first and second moments:

  $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_1^t}$

  $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\beta_2^t}$

- Update parameters:

  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \varepsilon}$

(Goodfellow et al. 2016, 301; Straka 2024b, 34)

2.1.1.2.4 Other optimization methods

Given that we will not be using non-differentiable loss function in the practical part of this project, and considering that advanced techniques for finding the global minimum are not necessary for our purposes, I will just focus on discussing some fundamental aspects of most well-known alternative algorithms.

Simulated annealing

This algorithm, designed for differentiable and non-differentiable loss functions, aids us in reaching the global minimum of a multi-dimensional space, even in the presence of numerous hills and valleys (local maxima and minima). It achieves this by making slight adjustments to the cost function shape when our optimization algorithm (e.g., SGD backpropagation) has converged to a minimum. Then, after repeating the same optimization algorithm, there is some chance that we'll converge to a deeper minimum. This iterative process can be repeated multiple times to reach even deeper minima. (Berka 2003, 81)

Genetic algorithm

A genetic algorithm (GA) is another popular method inspired by the process of natural selection and evolution, particularly useful for optimizing non-differentiable cost functions. GAs operate by evolving a population of candidate solutions over successive generations to find optimal or near-optimal solutions. Instead of using a cost function, GAs utilize a fitness function to evaluate the quality of each candidate solution. Additionally, GAs introduce several unique hyperparameters, including population size, crossover rate, and mutation rate. Instead of referring to the number of iterations as epochs, GAs use the term generations. First, we initialize a population of neural networks (NNs) with random weights. Then we evaluate each NN's performance on a given task using a fitness function. Next, we select high-performing networks to create offspring with a combination of their weights using the crossover rate. We then mutate our offspring by making random changes in weights and form a new population by replacing some of the old networks with offspring. We repeat this process until a stopping criterion like the number of generations or a satisfactory fitness threshold is met. Finally, the best-performing NNs in the final population represent the solutions to the task.(Mitchell 1997, 250; Berka 2003, 177)

Partical Swarm optimization

Particle swarm optimization (PSO) for NNs is inspired by the behavior of bird swarming. In PSO, a population of potential solutions (particles) iteratively adjusts their positions in the search space based on their own experience and the experience of the best-performing particles within the swarm. The collective movement aims to find optimal or near-optimal solutions for training NNs by adjusting the network's parameters to minimize a defined cost function.

2.1.1.3 Regularization Techniques

The extensive number of parameters typically used in NNs can lead to overfitting, where the model excels on training data but fails to generalize well to new data. To address this issue, various regularization methods are employed. Regularization methods involve any modification to a machine learning algorithm aimed at reducing generalization error, even if it doesn't necessarily decrease training error. In this section, we will explore several approaches to regularization in NNs.

2.1.1.3.1 L2 and L1 regularization

Just as in Ridge and Lasso regression, NNs can employ L^2 or L^1 norms as regularization. These regularization techniques favor simpler models by penalizing large weights and preferring those with smaller weights.

L^2 regularization

This regularization technique, also called Tikhonov regularization or weight decay, is a popular method that adds an extra regularization term to the original cost function:

$$\tilde{C}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2$$

Where λ is the regularization parameter that determines the extent to which we penalize our weights. The greater λ is, the more emphasis is placed on small weights, implying a less complex model that is more robust to noise.. It is important to note that, in practice, the regularization term is usually applied only to the weights and not to the biases, as biases do not typically cause overfitting. (Straka 2024a, 28) When incorporating weight decay into Mini-batch SGD Backpropagation, the gradient estimate is modified as follows:

$$\hat{\boldsymbol{g}} = \frac{1}{N}\sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}}\left(C\left(f\left(\boldsymbol{x}^{(i)};\, \boldsymbol{\theta}\right),\, \boldsymbol{t}^{(i)}\right) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2\right)$$

And the parameter update equation becomes:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\theta}C(\boldsymbol{\theta}) - \alpha\lambda\boldsymbol{\theta} = \boldsymbol{\theta}(1 - \alpha\lambda) - \alpha\nabla_{\theta}C(\boldsymbol{\theta})$$

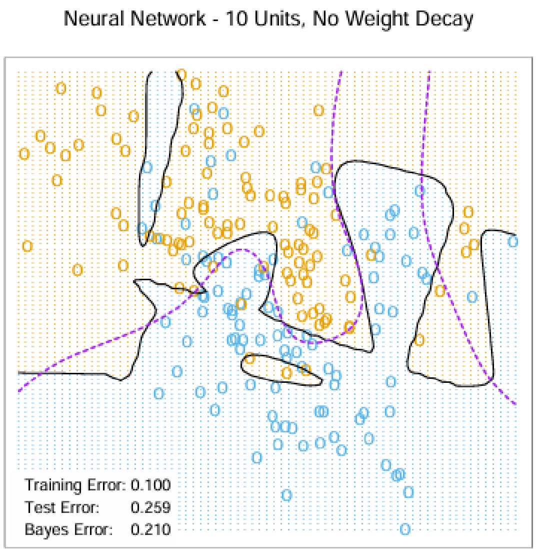
Alongside Adam's popularity in practical NN work, there's also AdamW, serving as its specialized counterpart with integrated weight decay. To avoid repetition and be straightforward, we will only mention the parameter update equation:

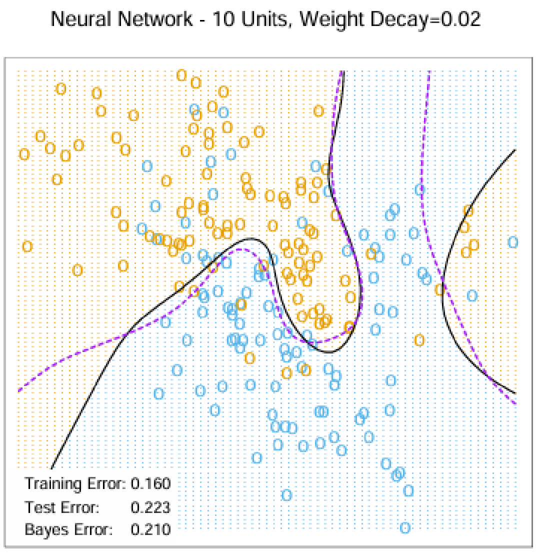
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \varepsilon} - \alpha\lambda\boldsymbol{\theta}$$

The change in the gradient estimate remains the same as for Mini-batch SGD Backpropagation, and everything else remains unchanged.

Finally, I'm going to demonstrate how L^2 regularization works in practice. In the following picture, we can see an example where a NN with and without weight decay is compared on classification problem. As shown, although the training error is slightly worse when using weight decay, the generalization error improves, which is our intended outcome.

Figure 7: Neural network with and without weight decay





Source: (Hastie et al. 2009, 399)

L^1 regularization

Just like with Tikhonov regularization, we can adjust the unregularized cost function by incorporating an additional regularization term:

$$\tilde{C}(\boldsymbol{\theta}) = C(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_1$$

This results in the following update equation:

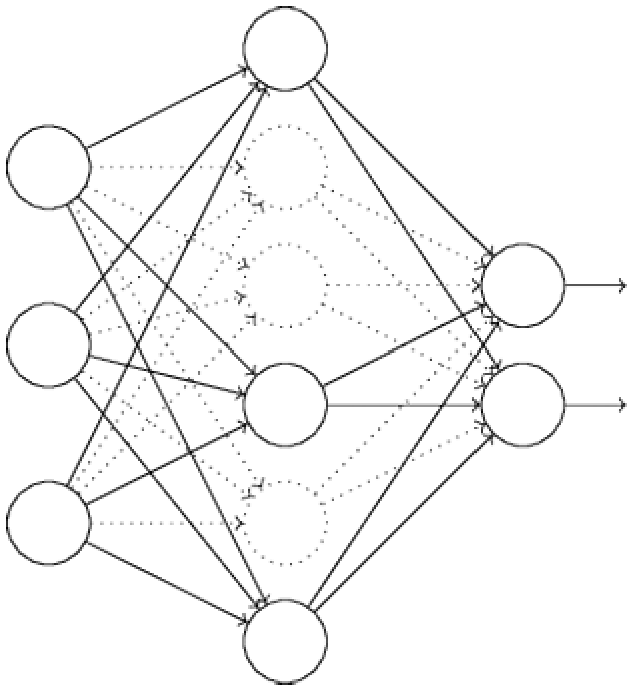
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_\theta C(\boldsymbol{\theta}) - \min(\alpha\lambda, \|\boldsymbol{\theta}\|_1)\, sign(\boldsymbol{\theta})$$

Unlike L^2 regularization, which proportionally decreases the value of the weights, L^1 regularization simply reduces weights by a constant. However, L^1 regularization typically doesn't perform well empirically with deep learning and is not commonly used in practice.(Michael 2019; Straka 2024a, 31)

2.1.1.3.2 Dropout

The core principle of dropout entails randomly deactivating specific neurons and their subsequent followers within the network during training. Each neuron is deactivated with a predetermined probability (e.g. 1/3), leading to the selection of a random subset from the entire network. Dropout is one of the most widely used method for enhancing a network's ability to generalize. For instance, in each batch, approximately 1/3 of neurons in the hidden layer are randomly deactivated. This approach resembles training multiple distinct neural networks and averaging their outputs.(Michael 2019)

Figure 8:Dropout employed in hidden layer



Source:(Michael 2019)

2.1.1.3.3 Early stopping

Because overfitting happens when the training error goes down but the validation error goes up, we aim to stop training when we observe the validation error rising.

However, pinpointing the exact moment to halt training is challenging because sometimes, with continued training, the validation error might start going down again. That's why using the Early stopping technique is smart. This technique involves saving the network's settings at different times during training and choosing the best ones later on.

2.1.1.3.4 Data augmentation

Another useful regularization technique typically used for classification is Data Augmentation, which involves artificially expanding the training data. This is done by taking some of our training data with known targets and slightly changing the input variables with noise or performing some transformations. For example, if we have an image where each input neuron represents one pixel, we can change a few of them or rotate the entire image.(Michael 2019)

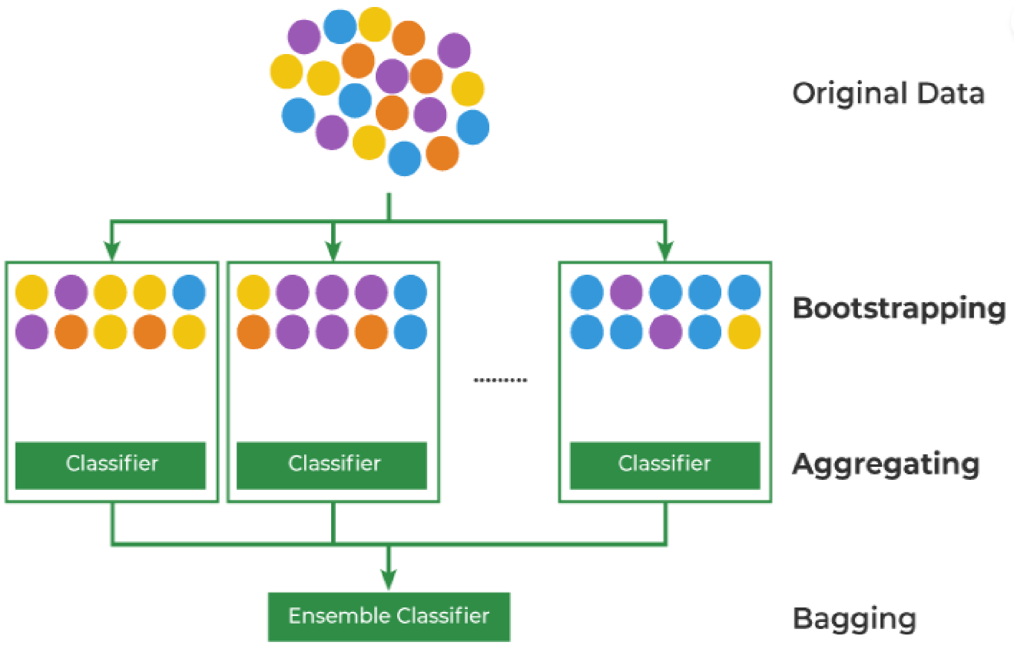
2.1.1.3.5 Ensembling

In general, in Machine Learning, a combination of models often performs better on a validation dataset than a single model. The same principle applies to Deep Learning, where we can combine entirely different architectures of NNs or variations of a single NN into a model ensemble. Some of the most popular approaches to model ensembling in Deep Learning include:

Bagging

Short for Bootstrap Aggregating, bagging involves aggregating NNs trained on multiple re-sampled (bootstrapped) datasets. By training several models on different subsets of the data and averaging their predictions (bagging), which in NNs means averaging the parameters $\boldsymbol{\theta}$, we can reduce variance and improve overall performance.(Anon. 2024)

Figure 9: Bagging Classifier

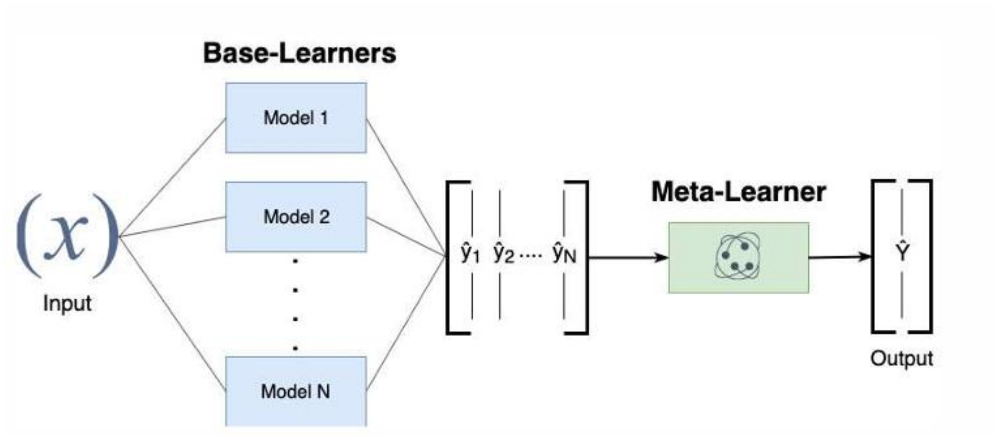


Source:(Anon. 2024)

Stacking

Short for Stacked Generalization, Stacking involves combining the predictions of multiple different NNs (Base-Learners) on a validation set using a Meta-Learner, which is often a weighted average of the predictions of various NNs or another machine learning algorithm. This approach leverages the strengths of different models to achieve better performance than any single model could alone.(Plašil 2023,9)

Figure 10: Process of Stacking



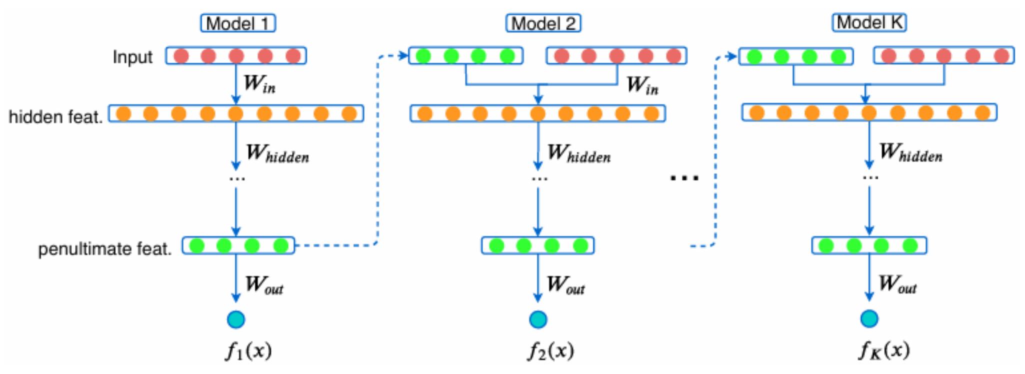
Source: (Plašil 2023, 10)

Boosting

Boosting for NNs is a method designed to create ensemble that significantly enhance the predictive power beyond what individual NNs can achieve. Unlike other regularization techniques, boosting increases the ensemble's effectiveness by gradually incorporating additional NNs. This involves adding simple NNs to the ensemble one at a time. Furthermore, boosting can be used to interpret a single neural network as an ensemble by incrementally adding hidden neurons to the network, resulting in an ensemble with greater complexity and predictive capabilities.(Goodfellow et al. 2016, 251)

One of the well-known Gradient Boosting NN algorithms is called GrowNet. In GrowNet, shallow NNs (one or two hidden layers) are used as weak learners. During each boosting step, the vectors of the original input features are combined with the output from the penultimate layer of the previous network. This new feature set is then used as input to train the next weak learner using the boosting algorithm with the current residuals. The final output of the model is a weighted combination of the output values from all these sequentially trained models.(Badirli et al. 2020, 3)

Figure 11: GrowNet architecture



Source:(Badirli et al. 2020, 3)

2.1.1.4 Unstable Gradient Problem

The Unstable Gradient Problem is a common issue encountered by NNs during the learning process, often resulting in a slowdown in learning or divergence from the minimum in backpropagation. The deeper the NN, the more likely we will encounter this problem. This phenomenon arises from the use of numerous derivatives of activation functions in backpropagation, where each derivative may be excessively small or large. As these derivatives are multiplied during the backward pass, the resulting values can become even smaller or larger. The Unstable Gradient problem is an umbrella term encompassing both the Vanishing Gradient Problem (VGP) and the Exploding Gradient Problem (EGP).

Vanishing Gradient Problem (VGP)

VGP refers to the challenge of gradients diminishing as they are propagated backward through the layers during the training of deep NNs. This phenomenon occurs when these gradients become exceedingly small, slowing the process of weight updating within the network.

Exploding Gradient Problem (EGP)

On the other hand, EGP presents the opposite scenario to VGP. Here, gradients tend to grow larger, resulting in large weight updates that can cause the Gradient Descent algorithm to diverge.

2.1.1.4.1 Handling Unstable Gradient Problem

Since there are many ways to solve the VGP or EGP, I'm going to describe the most well-known and utilized ones in practice.

Better weight initialization

In standard practice, weights are often initialized from a random distribution. While a normal distribution with a mean of 0 and a variance of 1 was traditionally used, modern techniques like Xavier (Glorot) initialization are preferred to ensure better performance. This method aims to achieve similar variance in each layer's output as in the input of that layer (previous layer’s output). Additionally, it aims to maintain similar variance in gradients before and after backpropagation for stable training and to prevent issues such as VGP and EGP.(Bohra 2024) One of the best-known ways to initialize weights is the normalized initialization proposed by Xavier (Glorot). In this method, the weight matrix between layer j with n_j neurons and layer $j+1$ with n_{j+1} neurons is initialize as follows:

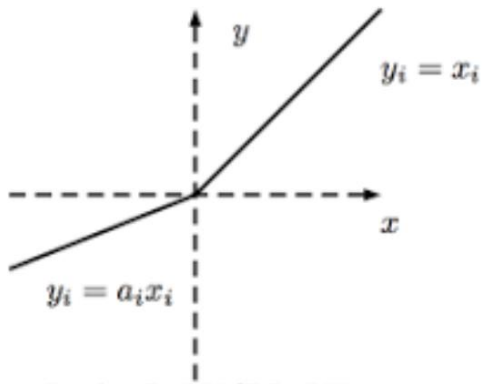
$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

(Glorot a Bengio [b.r.], 253)

Using non-saturating activation functions

Another simple way to handle the VGP and the EGP is to use non-saturating functions like ReLU. To be clear, a saturating function is function where large positive or negative input values tend to result in output values that are close to their maximum or minimum. This is problematic because it can lead to a loss of information; large positive or negative inputs do not have a significant impact on weight updates. For example, the sigmoid activation function outputs values between 0 and 1 and has almost flat regions at both ends. This also implies small derivatives of sigmoid, and multiplying these small derivatives during backpropagation can lead to the VGP. Unfortunately, ReLU is also not a perfect activation function because it can suffer from a problem known as "dying ReLU," where neurons output many zeros due to lots of negative inputs. Some popular alternatives that mitigate this problem include Leaky ReLU (LReLU), Parametric ReLU (PReLU), Exponential Linear Unit (ELU), and Scaled Exponential Linear Unit (SELU). (Bohra 2024) For example, Leaky ReLU (LReLU) looks like this:

Figure 12: Leaky ReLU



Source: (Bohra 2024)

Batch normalization

Using normalized initialization together with LReLU (or other ReLU alternatives) can significantly reduce the chances of experiencing the VGP and the EGP. However, it does not guarantee that these issues will not occur during training. Another useful method for handling VGP and EGP is called batch normalization, where we normalize each input variable of the mini-batch in SGD backpropagation using estimates of the mean and variance obtained from mini-batch examples. We then transform them using scale γ and shift β parameter (Batch Normalizing Transform $BN_{\gamma.\beta}: x_{(1\ldots J)} \rightarrow y_{(1\ldots.J)}$). (Ioffe a Szegedy 2015, 3) Finally, we input these normalized inputs to the activation function $f\big(BN(\boldsymbol{Wx})\big)$ without considering bias, since its effect would be canceled by mean subtraction. Nevertheless, the scale β parameter serves the role of bias.(Ioffe a Szegedy 2015, 4) The batch normalization algorithm for a mini-batch consisting of N training examples is as follows:

Input:

- Sampled mini-batch of N training examples $\left(\boldsymbol{x}^{(i)},\ \boldsymbol{t}^{(i)}\right)$
- Numerical stabilization constant ε with a default value 0.001
- Vector of scale parameters $\boldsymbol{\gamma}$ initialized to **1** (trained by optimizer)
- Vector of shift parameters $\boldsymbol{\beta}$ initialized to **0** (trained by optimizer)

Output: Normalized and transformed batch vectors $\left(\boldsymbol{t}^{(1)},\ \boldsymbol{t}^{(2)}, \ldots, \boldsymbol{t}^{(N)}\right)$

Algorithm:

- $\boldsymbol{\mu} \leftarrow \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}^{(i)}$
- $\boldsymbol{\sigma}^2 \leftarrow \frac{1}{N}\sum_{i=1}^{N}\left(\boldsymbol{x}^{(i)} - \boldsymbol{\mu}\right)^2$
- $\widehat{\boldsymbol{x}}^{(i)} \leftarrow \frac{(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})}{\sqrt{\boldsymbol{\sigma}^2 + \varepsilon}}$
- $\widehat{\boldsymbol{t}}^{(i)} \leftarrow \boldsymbol{\gamma} \odot \widehat{\boldsymbol{x}}^{(i)} + \boldsymbol{\beta} \equiv BN_{\boldsymbol{\gamma}.\boldsymbol{\beta}}\left(\boldsymbol{x}^{(i)}\right)$

(Ioffe a Szegedy 2015, 3; Straka [b.r.], 37)

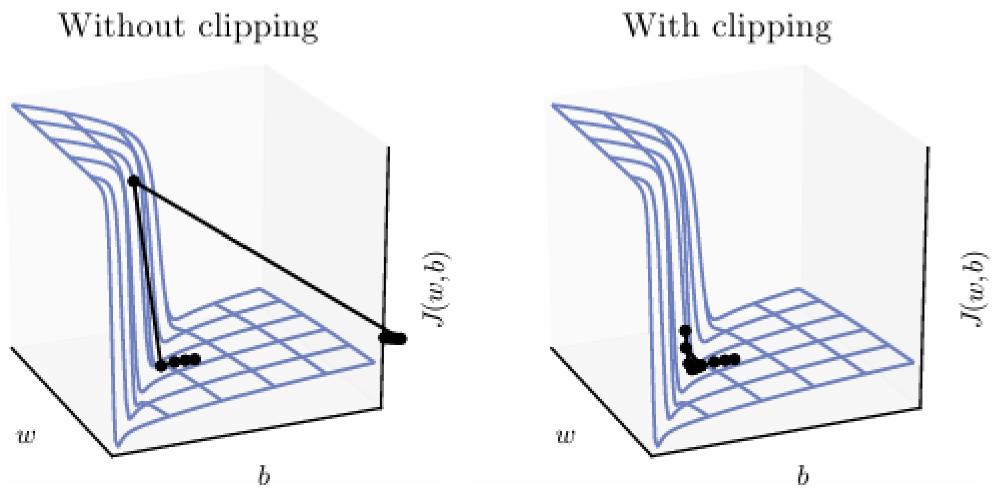
Skip connections

Skip connection, primarily used in CNNs, is a technique where the output of an earlier (non-skipped) layer is added to the output of a later (non-skipped) layer, bypassing one or more intermediate (skipped) layers. This addition occurs before the activation function of the later (non-skipped) layer. Skip connections help maintain the gradient flow during backpropagation, which enables the training of deeper networks and helps mitigate issues such as the VGP and EGP.(Adaloglou 2023)

Gradient clipping

During training, we sometimes face a problem where some gradients become so large that, after an update, they can trow our parameters into a region where the loss function is significantly higher, making all our previous training efforts useless. This issue is known as the exploding gradient problem (EGP). To prevent this, we can simply clip large gradients before updating the parameters.(Goodfellow et al. 2016, 402)

Figure 13: Effect of Gradient clipping with two parameters w and b



Source:(Goodfellow et al. 2016, 402)

One possible way to implement this is to set a norm threshold v. Before each parameter update, the gradient $\boldsymbol{g}$ is adjusted as follows:

$$\boldsymbol{g} \leftarrow \begin{cases} \boldsymbol{g} & if\ \|\boldsymbol{g}\| \le v \\ v\dfrac{\boldsymbol{g}}{\|\boldsymbol{g}\|} & if\ \|\boldsymbol{g}\| > v \end{cases}$$

(Goodfellow et al. 2016, 403)

Different architectures

Other solutions for handling VGP or EGP include using various types of architectures, such as specialized RNNs like LSTM, GRU, and Echo State Networks, or even gradient-free architectures like EVOLINO, where weights are updated through evolutionary strategies rather than a gradient descent approach.

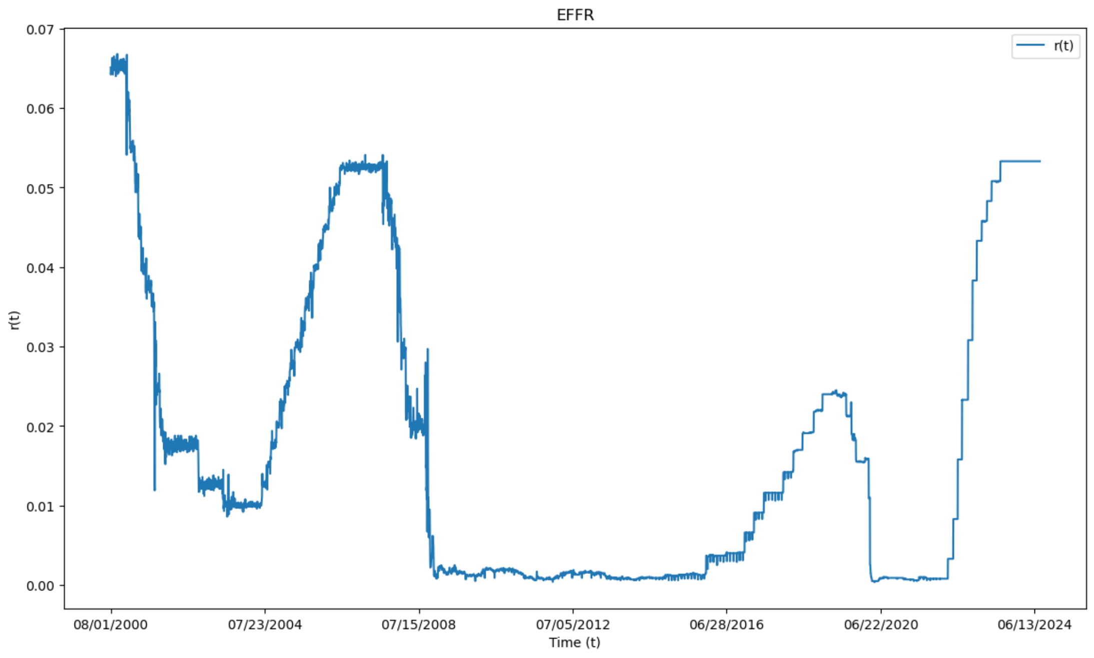
3 Calculation of Credit Valuation Adjustment

Finally, I’m going to present the practical part of this thesis, where I will demonstrate the standard Monte Carlo approach for calculating the CVA of a basic Interest Rate Swap (IRS). In this swap, one counterparty pays a floating interest rate and another counterparty pays a fixed interest rate in same time. Additionally, I will propose an alternative method, wherein a NN is trained on numerous generated examples with the EPE as the target variable.

3.1 Calculation with Monte Carlo method

To generate a reasonable expected exposure (EE) of IRS based on real data, I realized that it's rational to calibrate the expected interest rate processes to market compound interest rates known from the current market term structure via a 1-factor Vasicek model. To do that, I have chosen US Government Treasury Yields with maturities from 3 months up to 1 year and the current effective Fed Funds rate as the current ON rate. Additionally, I utilized EFFR for the calculation of the diffusion parameter σ of Vasicek dynamics as the annualized standard deviation of differences of these O/N rates over the last 23 years.

Figure 14: Historical data of EFFR



Source: (Anon. [b.r.])

To calculate the diffusion parameter σ of the Vasicek‘s model, I simply use the annualized standard deviation of the changes in the O/N rates:

$$\sigma = \frac{\sigma_{dr}}{\sqrt{dt}}$$

Which in our case is 1.48 %.

Now, after calculating the diffusion parameter σ from historical EFFR rates, we can find the optimal parameters a (speed of reversion) and b (long-term mean) as the values where the sum of the squared differences between observed term-structure rates and Vasicek's affine rates over all observed maturities is minimal. To do that, we need to:

1. Set the bounds for parameters a and b by specifying a reasonable range for optimization. I used the range $a \in \langle 0{,}01;\ 0{,}5 \rangle$ and $b \in \langle 0{,}01;\ 0{,}1 \rangle$.
2. Define Vasicek's affine rate formula for current timestamp:

$$R(0,T) = \alpha(0,T) + \beta(0,T) r(0)$$

   Where:

$$\alpha(0,T) = -\frac{\ln\bigl(A(0,T)\bigr)}{T-0}$$

$$\beta(0,T) = \frac{B(0,T)}{T-0}$$

$$A(0,T) = e^{\left(\frac{(B(0,T)\,T-0)(a^2 b-\sigma^2/2)}{a^2} - \frac{\sigma^2 B(0,T)^2}{4a}\right)}$$

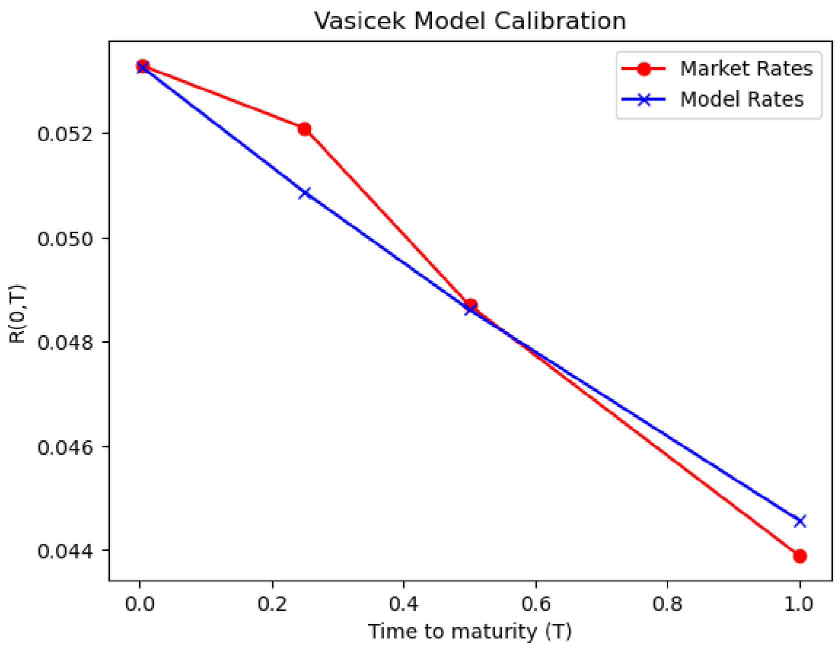
$$B(0,T) = \frac{\left(1 - e^{-a(T-0)}\right)}{a}$$

3. Define the optimization problem minimizing the sum of squared differences and employ the L-BFGS-B optimization method, or an alternative method, to determine the optimal parameters a and b.

$$SSE(a,b) = \sum_{i=1}^{N} \left(R(0,T_i)_{Market} - R(0,T_i)_{Vasicek(a,b)}\right)^2$$

After completing all the steps, I determined the optimal parameters to be $a = 0{,}46$ and $b = 1{,}0$ %. When comparing the calibrated Vasicek model rates with the market rates, there are small discrepancies. These differences could be addressed by using more sophisticated interest rate models such as the Hull-White model. However, for the purposes of this thesis, the Vasicek model is sufficient.

Figure 15:Comparision of calibrated Vasicek's model rates with market rates



Source: Own work

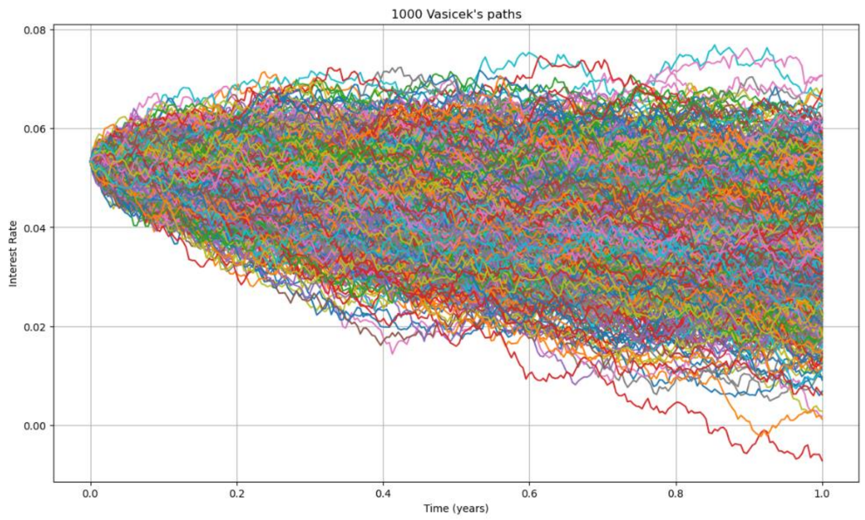
To better interpret mean reversion value, we will use the concept of the half-life, defined as:

$$HL = \frac{\ln(2)}{a}$$

This formula tells us how long it takes for our rate $r(t)$ at time t to reach the midpoint between its initial value $r(0)$ and the long-term mean b. Because we are only considering expected values of process differences, the half-life approach assumes the absence of volatility. For our process, the initial rate, according to the half-life approach, should on average reach the midpoint between the initial rate and the long-term mean in approximately 534 days.

Now, for illustration, I will present some processes generated using Vasicek's model through the stochastic differential equation $dr(t) = a\big(b - r(t)\big)dt + \sigma dW(t)$. These processes will be generated similarly during the calculation of the IRS value over time.

Figure 16: Generated 1000 simulations of possible Vasicek's process with calibrated parameters



Source: Own work

As mentioned earlier, for generating possible future exposure, I chose a basic Interest Rate Swap that exchanges floating and fixed rates. Where:

- Floating rate = Compounded interest rate derived from the ON rate
- Fixed rate = Chosen fixed interest rate

To generate the value paths of the IRS, we will use the following formula for the value of the swap at time t:

$$f_t = L\sum_{i=1}^{N} P(t,T_i)(r(t,T_{i-1},T_i)_{float,Vasicek} - r_{fix})\delta_i$$

Where:

L = Notional Amount

δ_i = $T_i - T_{i-1}$ = Time period distance

$T_1, T_2, \dots, T_N$ = Payment dates

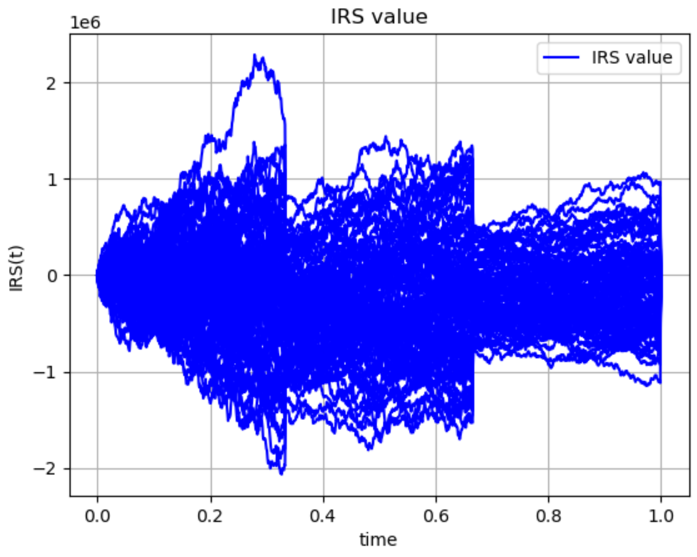
T_0 = Beginning of IRS (there is no payment, we just know first floating rate which wil be changed at T_1)

And because we know that $r(t,T_{i-1},T_i) = \frac{P(t,T_{i-1})-P(t,T_i)}{P(t,T_i)\delta_i}$, we can simplify the formula for value of the swap at time t as:

$$f_t = L(P(t,T_0) - P(t,T_N)) - L \times r_{fix}\sum_{i=1}^{N} P(t,T_i)\delta_i$$

For example, consider a notional amount of 100 000 000 USD with four exchanged payments over one year. The fixed rate is set at 4,5 %, and the floating rate is the Overnight Effective Federal Funds Rate. The value of the IRS over time for payer counterparty can be represented as follows:

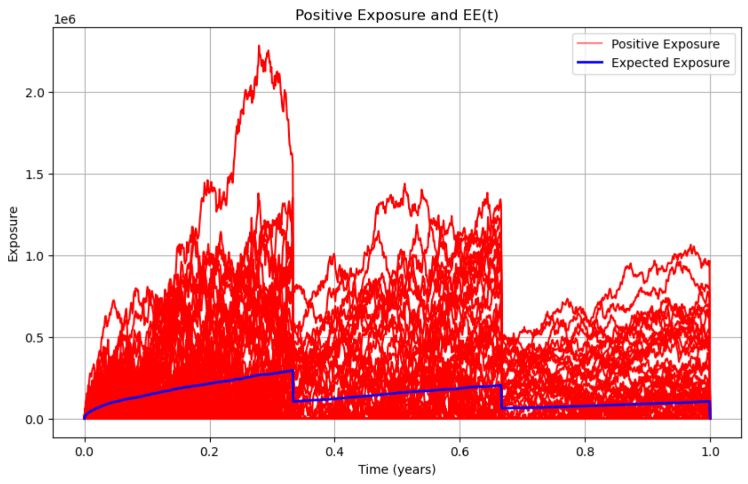
Figure 17: IRS value over time



Source: Own work

To calculate the Expected Exposure, we need to generate many processes. For instance, if I generate 2000 possible swap values and consider only the positive values, the exposures and the Expected Exposure can be represented as follows:

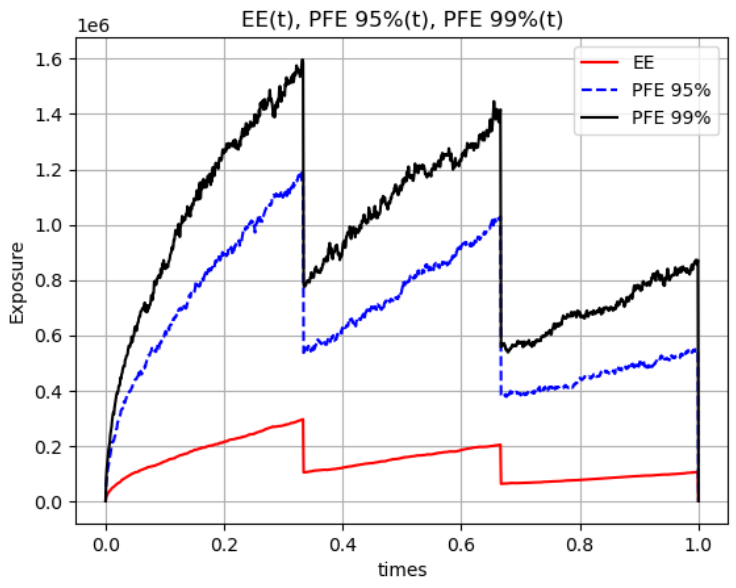
Figure 18: Exposures and Expected Exposure for IRS



Source: Own work

For those interested in worst-case scenarios, it can be valuable to visualize the Potential Future Exposure (PFE), which represents the expected exposure under worst-case conditions, i.e., specified quantiles. The following chart compares EE with PFE at the 99 % and 95 % quantiles.

Figure 19: EE and PFE for IRS



Source: Own work

Finally, we will calculate the simplified CVA and EPE value for this specific derivative. By setting the Loss Given Default (LGD) to 60 %, the risk-free rate to 4,33 %, and the average default intensity to 4 %, we obtain the following results:

- Expected Positive Exposure (EPE): 137 758 USD
- Credit Valuation Adjustment (CVA): 3 238 USD

3.2 Calculation with Neural Networks

Given the slow process of calculating EPE, I propose an alternative approach that leverages a NN trained on a synthetic dataset of IRS contracts with varying parameters such as current O/N rate, mean reversion rate , long-term mean rate, volatility of O/N rate changes, number of payments, counterparty type, and fixed rate. The target variable for this dataset will be the EPE. I also assume a notional value of 1 for all observations in this dataset, as L is simply a multiplicative constant and varying L values can complicate the training of neural networks.

To train the model, I generated a dataset comprising of 100,000 distinct standard IRS products. The variations are created as follows:

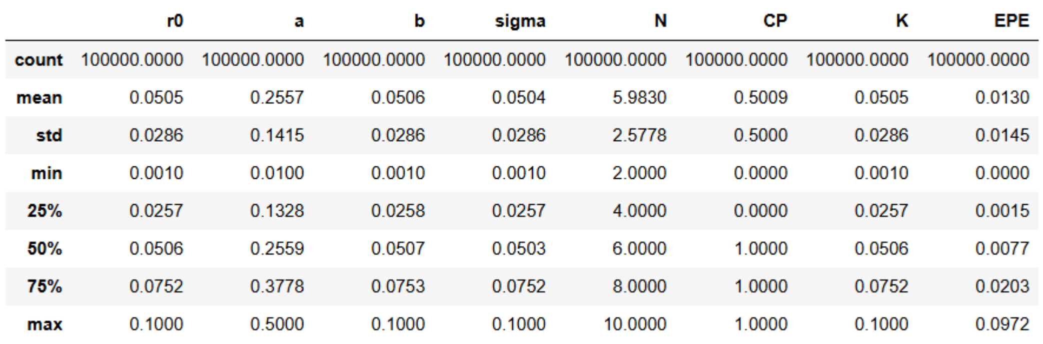
- Current O/N rate ($r(0)$) values are sampled from a continuous uniform distribution ranging between 0,1 % and 10 %.
- Mean reversion rate (a) values are sampled from a continuous uniform distribution ranging between 0,01 and 0,5.
- Long-term mean rate (b) values are sampled from a continuous uniform distribution ranging between 0,1 % and 10 %.
- Volatility of O/N rate differences (σ) values are sampled from a continuous uniform distribution ranging between 0,1 % and 10 %.
- Number of payments (N) values are sampled from a discrete uniform distribution with values between 2 and 10.
- Counterparty type (CP) values are sampled from a discrete uniform distribution with two possible outcomes: RECEIVER and PAYER.
- Fixed rate values (K) are sampled from a continuous uniform distribution ranging between 0,1 % and 10 %.

For each generated sample, I calculated the EPE using the Monte Carlo method, based on 2,000 different swap values each from the initial date to the maturity date.

3.2.1 Exploratory Data Analysis

First, as part of the exploratory data analysis, we will examine some basic statistics for each explanatory variable and the response variable. From the following table, we can see that the data appears to be well-behaved without any unusual values. The means and medians of the variables are close to each other, indicating a relatively normal distribution. There is no significant variance in the variables that would cause issues during the training of the NN. Overall, the dataset seems appropriate for further analysis.

Table 2: Statistics of explanatory and target features

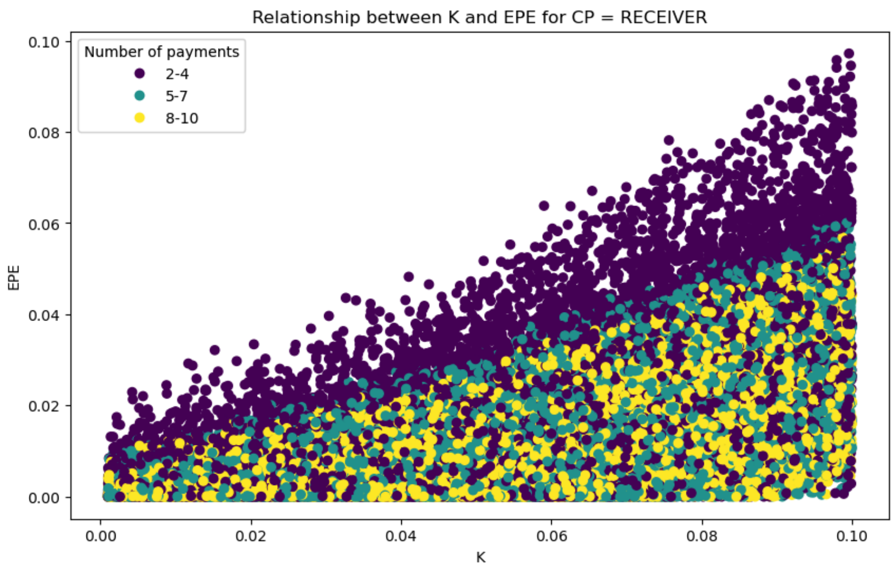


	r0	a	b	sigma	N	CP	K	EPE
count	100000.0000	100000.0000	100000.0000	100000.0000	100000.0000	100000.0000	100000.0000	100000.0000
mean	0.0505	0.2557	0.0506	0.0504	5.9830	0.5009	0.0505	0.0130
std	0.0286	0.1415	0.0286	0.0286	2.5778	0.5000	0.0286	0.0145
min	0.0010	0.0100	0.0010	0.0010	2.0000	0.0000	0.0010	0.0000
25%	0.0257	0.1328	0.0258	0.0257	4.0000	0.0000	0.0257	0.0015
50%	0.0506	0.2559	0.0507	0.0503	6.0000	1.0000	0.0506	0.0077
75%	0.0752	0.3778	0.0753	0.0752	8.0000	1.0000	0.0752	0.0203
max	0.1000	0.5000	0.1000	0.1000	10.0000	1.0000	0.1000	0.0972

Source: Own work

When visualizing the relationship between fixed rate and EPE for the counterparty type receiver, we observe that EPE values tend to increase with higher Fixed rate. Higher fixed rate is beneficial for the receiver but results in greater exposure to counterparty risk. Additionally, swaps with fewer payments generally show higher EPE.

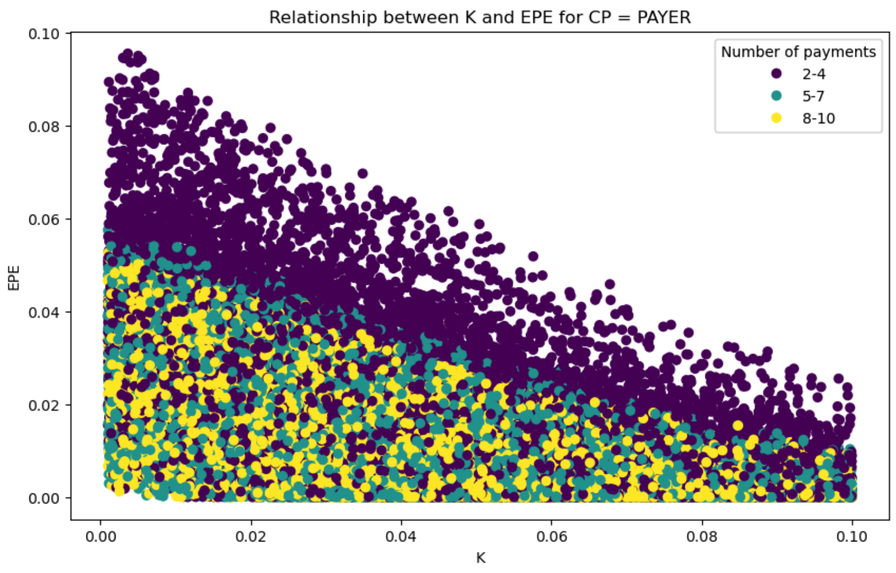
Figure 20:Relationship between fixed rate and EPE for receiver



Source: Own work

For the counterparty type payer, we observe the relationship between fixed rate and EPE in the opposite direction. Higher fixed rates result in lower EPE values, showing an inverse relationship compared to the receiver. Swaps with fewer payments still generally show higher EPE.

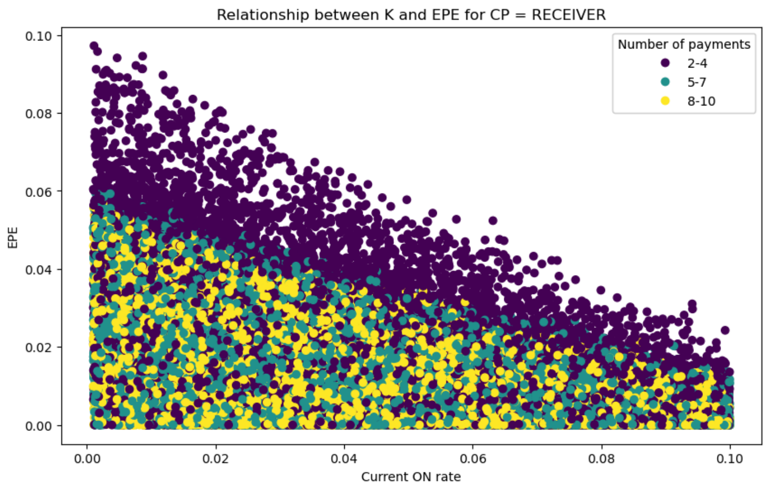
Figure 21: Relationship between fixed rate and EPE for payer

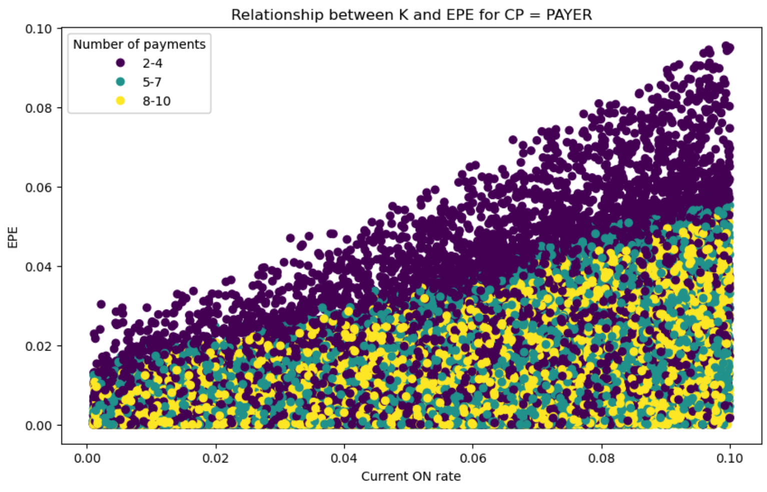


Source: Own work

When we replace the fixed rate with the current O/N rate, the trends also reverse. For receiver, higher current O/N rate values result in lower EPE, while for payer, higher current ON rate values lead to increased EPE.

Figure 22: Relationship between current O/N rate and EPE – receiver and payer





Source: Own work

3.2.2 Feature engineering and training NNs

No feature engineering was required for this approach. However, if we had not assumed all Lvalues to be equal to 1, it would have been necessary to normalize the data. This would involve scaling L and EPE, to fall within the range of 0 to 1 using min-max normalization. Normalization would be essential due to the large values and high variance associated with EPE and L when L values are not constant. This process helps manage high variance, facilitating the training of NNs and improving their ability to detect patterns.

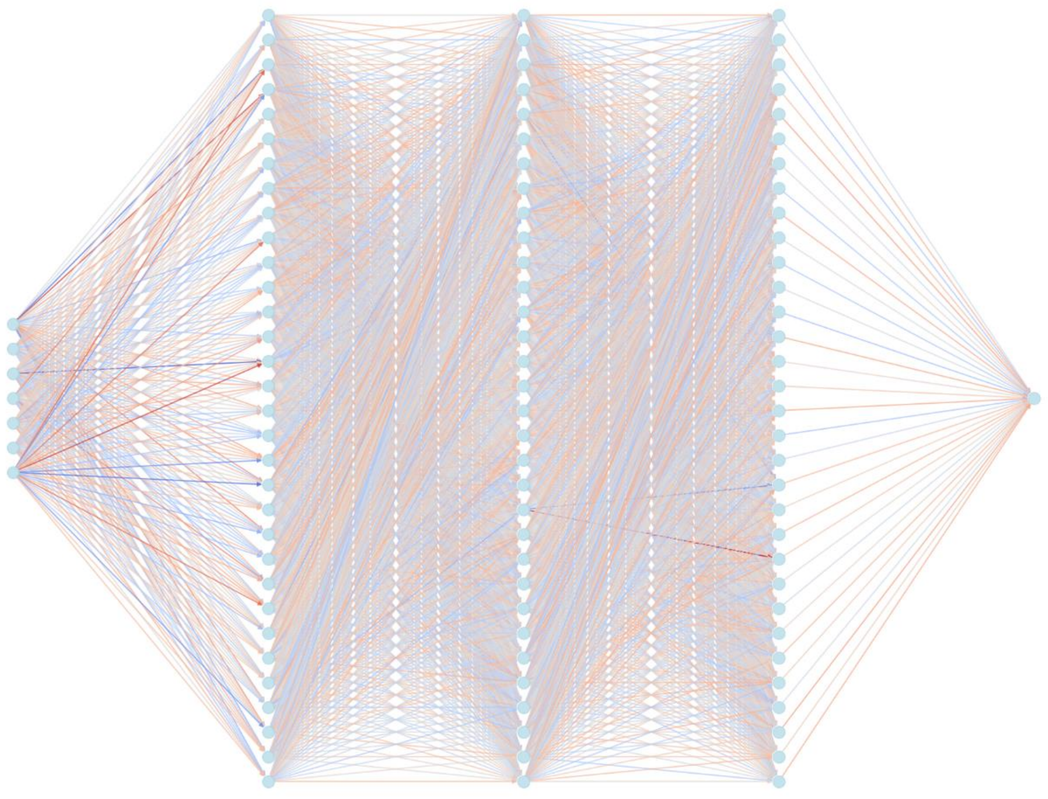
3.2.2.1 Training NNs

This section describes the implementation and training of neural network architectures using the Adam and AdamW optimizers. Both architectures are designed to predict EPE and share several common parameters and configurations.

Shared Architecture Details

Both the Adam and AdamW models were trained on a artificial dataset with a learning rate of $\alpha = 0{,}00001$. The neural network architecture for both models comprised an input layer, three hidden layers with 32 nodes each, all using the ReLU activation function, and an output layer also utilizing the ReLU activation function.The weights were initialized using the Glorot Uniform initializer, as discussed in the theoretical section.

Figure 23: Adam and AdamW architectures used for prediction of EPE



Source: Own work

The dataset was split into training, validation, and test sets, with 80 % of the data allocated for training, 10 % for validation, and 10 % for testing. Early stopping was employed during the training process to halt learning if the validation error did not improve after 40 epochs, with a maximum of 1000 epochs set as an additional stopping criterion. The batch size for training was set to 16. Mean Squared Error was used as the loss function for both models. Both optimizers incorporated momentum with $\beta_1 = 0{,}9$ and $\beta_1 = 0{,}999$.

Adam

As the initial model, I employed a NN optimized with the Adam algorithm, without applying any regularization. The Adam optimizer enhances the convergence speed of the training process by adaptive learning rates based on the first and second moments of the gradients. This approach is superior to traditional MLP models, as it offers more efficient weight adjustments, resulting in faster and more stable training. Moreover, by monitoring the validation loss and incorporating early stopping, the model effectively mitigates the risk of overfitting, ensuring better generalization performance.

AdamW

The second model utilized the AdamW optimizer, which adds weight decay regularization with a value of 0.004. This regularization technique helps prevent overfitting by penalizing large weights, thereby promoting a simpler and more generalizable model. While the network structure, learning rate, data splitting strategy, early stopping criteria, batch size, and loss function remained identical to those used with the Adam optimizer, the key difference was the inclusion of weight decay in the AdamW optimizer. This addition helps mitigate overfitting by introducing a penalty on the size of the weights, encouraging the model to maintain smaller weights during training.

3.2.3 Model evaluation

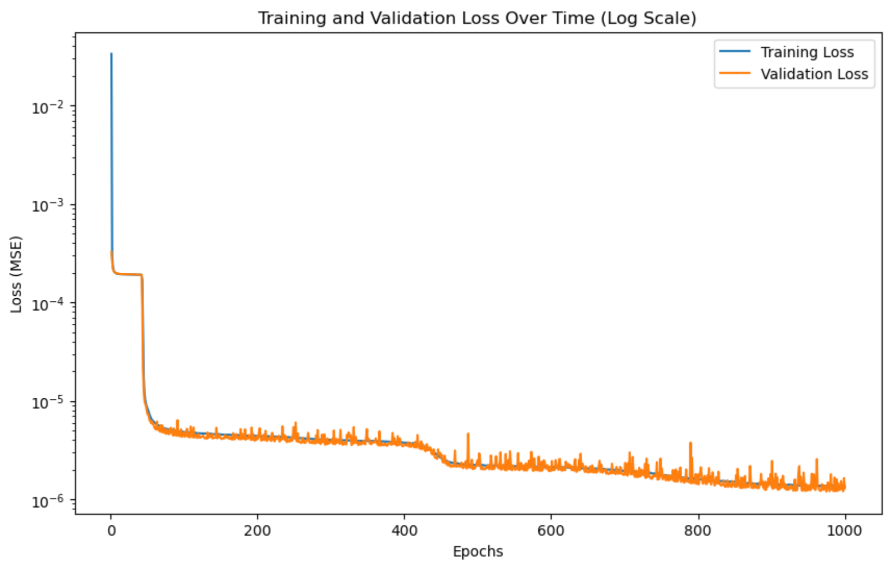
In this section, we evaluate the performance of two NN architectures trained with different optimizers: Adam and AdamW. By employing these optimizers, both architectures were effectively trained to predict EPE. The AdamW optimizer, in particular, provided an additional mechanism to prevent overfitting through weight decay regularization. The consistent use of early stopping and appropriate loss functions ensured robust training and reliable performance across both models.

Training and Validation Loss

The following graphs illustrate the training and validation loss over time for models optimized with Adam and AdamW respectively. These graphs are plotted on a logarithmic scale to better visualize the loss reduction.

The graph for the Adam optimizer demonstrates a steep initial decline in both training and validation loss, followed by a gradual decrease over the course of 1000 epochs. This indicates effective learning and convergence of the model. The validation loss closely follows the training loss, suggesting that the model generalizes well to unseen data. The early stopping mechanism was not employed, instead, a stopping criterion of 1000 epochs was used, ensuring the model did not train beyond the point of optimal performance on the validation set.

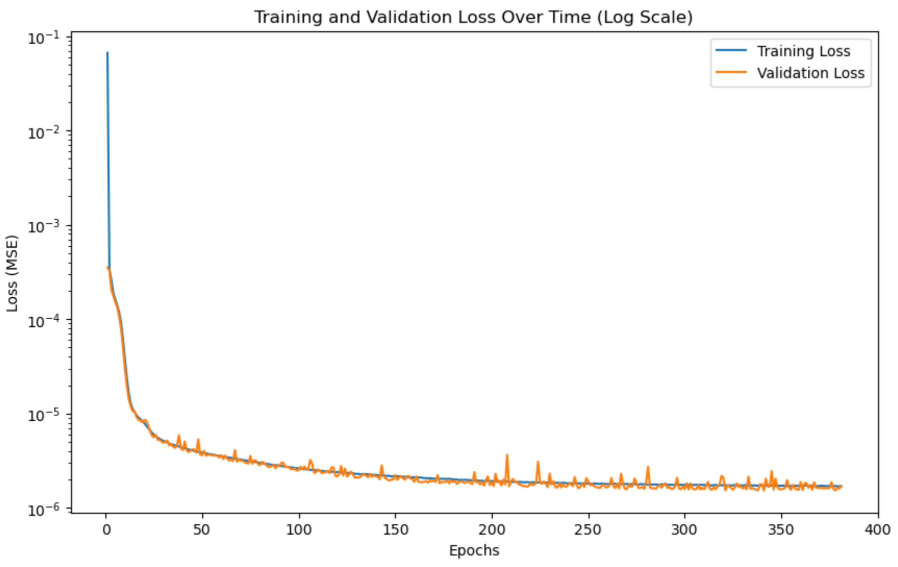
Figure 24: Comparision of training and validation error - Adam



Source: Own work

Similarly, the AdamW model shows a sharp initial reduction in both training and validation loss. However, it achieves convergence in significantly fewer epochs, 381 due to early stopping, compared to 1000 epochs with the Adam optimizer. This faster convergence can be attributed to the weight decay regularization mechanism of AdamW, which helps in preventing overfitting and improving generalization.

Figure 25: Comparision of training and validation error - AdamW



Source: Own work

Performance Metrics

The overall performance of the models is summarized in the following table. The table compares the number of epochs needed to train the NN, along with the Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Coefficient of Variation of the Root Mean Squared Error (CVRMSE), and the coefficient of determination (R^2) for both the training and test sets.

Table 3: Performance Metrics - Adam, AdamW

	Adam		**AdamW**	
	3 hidden layers		3 hidden layers	
	train	test	train	test
# of epochs	1000		381	
MSE	0,0000012	0,0000012	0,0000016	0,0000015
RMSE	0,0010880	0,0010855	0,0012481	0,0012351
CVRMSE	8,385 %	8,396 %	9,605 %	9,580 %
R^2	99,435 %	99,420 %	99,257 %	99,256 %

Source: Own work

The Adam model achieved slightly better performance in terms of MSE and RMSE, with lower values for both training and test sets compared to the AdamW model. This indicates a marginally better fit to the data, resulting in smaller prediction errors. Additionally, the Adam model's CVRMSE values were lower, signifying a more accurate relative error measurement.

However, the AdamW model demonstrated significant efficiency in training, converging in only 381 epochs as opposed to the Adam model's 1000 epochs. This efficiency is largely due to the weight decay regularization mechanism inherent in the AdamW optimizer, which helps to mitigate overfitting and enhances generalization capabilities.

Both models achieved very high R^2 values, indicating strong predictive power and an excellent fit to the data. The Adam model had slightly higher R^2 values for both training and test sets, but the difference is minimal, showing that both models are highly effective in explaining the variance in the data.

In summary, while the Adam optimizer exhibits slightly superior performance metrics, the AdamW optimizer offers notable advantages in training efficiency and overfitting prevention, making it a compelling choice for practical applications.

3.3 Comparison of approaches

Similarly to the standard Monte Carlo method, NNs can achieve comparable results with sufficient precision. While an NN calculates only the EPE value, which is the most time-consuming task, once we have this value, we proceed similarly to the Monte Carlo approach using the simplified formula for CVA. However, before using this simplified formula, the EPE must be multiplied by the principal amount.

The main disadvantage of the NN method is the time required to generate artificial dataset and train NN. This phase can be quite extensive and resource-intensive. Additionally, this approach would be more practical if a better interest rate model were used, enhancing the accuracy and reliability of the predictions.

However, the primary advantage of using NNs lies in the speed of obtaining sufficiently precise results once the NN has been trained. After the dataset generation and the initial training phase, the process of calculating EPE and subsequently CVA using the NN is significantly faster compared to the Monte Carlo method. This efficiency gain makes the NN approach particularly appealing for applications where rapid computations are critical.

In summary, while the initial setup of the NN approach requires considerable effort and computational resources, its speed and efficiency in producing precise results post-training offer a compelling alternative to the traditional Monte Carlo method for CVA calculation.

Conclusion

This thesis demonstrates that NNs can significantly expedite CVA calculation process traditionally reliant on Monte Carlo simulations. By employing NN models, specifically those optimized with Adam and AdamW, we achieved sufficiently accurate predictions of the EPE with substantially reduced computational time.

Despite these advantages, several challenges were identified. The generation of training datasets and the initial training of NNs are time-consuming processes. Additionally, the Vasicek model used in this thesis is not ideal, indicating a need for more sophisticated interest rate models in future work.

One promising alternative is the Hull-White model, which allows for a perfect fit to the current term structure of interest rates, contrasting with the Vasicek model. However, the Hull-White model is more complex to parameterize. This issue can be addressed by parameterizing the theta function of the Hull-White model and using these parameters as additional explanatory variables in the generated dataset.

In conclusion, while NNs offer a promising method to enhance the efficiency of CVA calculations, future research should focus on refining interest rate models and optimizing NN training processes. Incorporating more sophisticated models like Hull-White will likely improve accuracy and applicability in practical financial scenarios.

Bibliography

ADALOGLOU, Nikolas, 2023. *Intuitive Explanation of Skip Connections in Deep Learning* [online]. Dostupné z: https://theaisummer.com/skip-connections/

Anon., 2015. Implementing Deep Learning using cuDNN. In: [online]. B.m. Dostupné z: https://www.slideshare.net/slideshow/251-implementing-deep-learning-using-cu-dnn/52784152

Anon., 2024. *ML | Bagging classifier* [online]. Dostupné z: https://www.geeksforgeeks.org/ml-bagging-classifier/

Anon., [b.r.]. *Effective Federal Funds Rate* [online]. Dostupné z: https://www.newyorkfed.org/markets/reference-rates/effr

BADIRLI, Sarkhan, Xuanqing LIU, Zhengming XING, Avradeep BHOWMIK, Khoa DOAN a Sathiya S. KEERTHI, 2020. *Gradient Boosting Neural Networks: GrowNet* [online]. 14. červen 2020. B.m.: arXiv. [vid. 2024-05-26]. Dostupné z: http://arxiv.org/abs/2002.07971

BERKA, Petr, 2003. *Dobývání znalostí z databází*. Vyd. 1. Praha: Academia. ISBN 978-80-200-1062-9.

BISMI, Iqra, 2023. *Equiangular Basis Vectors: A Better Alternative to Softmax for Classification Tasks* [online]. Dostupné z: https://medium.com/@iqra.bismi/equiangular-basis-vectors-a-better-alternative-to-softmax-for-classification-tasks-2a2e7fa7b1fc#:~:text=One%20of%20the%20main%20drawbacks,a%20large%20number%20of%20classes.

BOHRA, Yash, 2024. *The Challenge of Vanishing/Exploding Gradients in Deep Neural Networks* [online]. Dostupné z: https://www.analyticsvidhya.com/blog/2021/06/the-challenge-of-vanishing-exploding-gradients-in-deep-neural-networks/

FIČURA, Milan, 2023. Quantitative Portfolio Management - Lecture 6. In: . B.m.

FORJAN, James, 2023. *Counterparty Risk | AnalystPrep - FRM Part 2 Study Notes* [online]. Dostupné z: https://analystprep.com/study-notes/frm/counterparty-risk/

GLOROT, Xavier a Yoshua BENGIO, 2010. Understanding the difficulty of training deep feedforward neural networks.

GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE, 2016. *Deep learning*. Cambridge, Massachusetts: The MIT Press. Adaptive computation and machine learning. ISBN 978-0-262-03561-3.

GRANT, Sanderson, 2017. *But what is a Neural Network?* [online]. Dostupné z: https://www.3blue1brown.com/lessons/neural-networks

HASTIE, Trevor, Robert TIBSHIRANI a J. H. FRIEDMAN, 2009. *The elements of statistical learning: data mining, inference, and prediction*. 2nd ed. New York, NY: Springer. Springer series in statistics. ISBN 978-0-387-84857-0.

IOFFE, Sergey a Christian SZEGEDY, 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* [online]. 2. březen 2015. B.m.: arXiv. [vid. 2024-05-27]. Dostupné z: http://arxiv.org/abs/1502.03167

JASON, Brownlee, 2020. *A Gentle Introduction to the Rectified Linear Unit (ReLU)* [online]. Dostupné z: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

JUAN, Luis Orozco Villalobos, 2020. *Test, training and validation sets* [online]. Dostupné z: https://www.brainstobytes.com/test-training-and-validation-sets/

KANSAL, Sahdev, 2020. *Quick Guide to Gradient Descent and Its Variants* [online]. Dostupné z: https://towardsdatascience.com/quick-guide-to-gradient-descent-and-its-variants-97a7afb33add

KRIZHEVSKY, Alex, Ilya SUTSKEVER a Geoffrey E. HINTON, 2017. ImageNet classification with deep convolutional neural networks. *Communications of the ACM* [online]. **60**(6), 84–90. ISSN 0001-0782, 1557-7317. Dostupné z: doi:10.1145/3065386

MICHAEL, Nielsen, 2019. *Improving the way neural networks learn* [online]. Dostupné z: http://neuralnetworksanddeeplearning.com/chap3.html

MITCHELL, Tom M., 1997. *Machine Learning*. New York: McGraw-Hill. McGraw-Hill series in computer science. ISBN 978-0-07-042807-2.

NIELSEN, Michael, 2019a. *How the backpropagation algorithm works* [online]. Dostupné z: http://neuralnetworksanddeeplearning.com/chap2.html

NIELSEN, Michael, 2019b. *Using neural nets to recognize handwritten digits* [online]. Dostupné z: http://neuralnetworksanddeeplearning.com/chap1.html

PLAŠIL, Miroslav, 2023. Úvod do ensemble methods. In: . Praha.

REN, Guanghua, Yuting CAO, Shiping WEN, Tingwen HUANG a Zhigang ZENG, 2018. A modified Elman neural network with a new learning rate scheme. *Neurocomputing* [online]. **286**, 11–18. ISSN 09252312. Dostupné z: doi:10.1016/j.neucom.2018.01.046

SINGH CHOUDHARY, Anurag, 2023. *An Overview of Variational Autoencoders (VAEs)* [online]. Dostupné z: https://www.analyticsvidhya.com/blog/2023/07/an-overview-of-variational-autoencoders/#:~:text=Variational%20Autoencoders%20(VAEs)%20are%20generative,dataset%20and%20generate%20novel%20samples.

STRAKA, Milan, 2024a. Convolutional Neural Networks. In: [online]. Praha. Dostupné z: https://ufal.mff.cuni.cz/~straka/courses/npfl138/2324/slides.pdf/npfl138-2324-04.pdf

STRAKA, Milan, 2024b. Training Neural Networks II. In: [online]. B.m. Dostupné z: https://ufal.mff.cuni.cz/~straka/courses/npfl138/2324/slides.pdf/npfl138-2324-03.pdf

STRAKA, Milan, 2024c. Training Neural Networks. In: [online]. Praha. Dostupné z: https://ufal.mff.cuni.cz/~straka/courses/npfl138/2324/slides.pdf/npfl138-2324-02.pdf

WITZANY, Jiří, 2020. *Derivatives: theory and practice of trading, valuation, and risk management* [online]. Cham: Springer. Springer texts in business and economics. ISBN 978-3-030-51750-2. Dostupné z: doi:10.1007/978-3-030-51751-9

List of figures and tables

List of figures

List of tables