UNIVERSITY
OF OSLO

Master's thesis

# -sometitle-

Classifying N-body simulations with and without relativistic corrections using machine learning techniques

**Johan Mylius Kroken**

Computational Science: Astrophysics
60 ECTS study points

Institute of Theoretical Astrophysics
Faculty of Mathematics and Natural Sciences

Spring 2024

**Johan Mylius Kroken**

# -sometitle-

Classifying N-body simulations with and without relativistic corrections using machine learning techniques

Supervisors:
A David Fonseca Mota
B Julian Adamek
C Francisco Antonio Villaescusa Navarro

## Abstract

On large scales, comparable to the horizon, relativistic effects will affect the cosmological observables. In order to solve for these effects, one need to consistently solve for the metric, velocities and densities in a particular gauge. When simulating large-scale structures we use N-body simulations, which are usually performed in the Newtonian limit. However, it is not obvious that Newtonian gravity yield a good global description of an inhomogeneous cosmology across all scales (Jeong, Schmidt and Hirata 2012). However, literature suggest that Newtonian simulations are still solving the dynamics correctly, even on large scales close to the horizon where relativistic effects are important but may be corrected for (Chisari and Zaldarriaga 2011) (Green and Wald 2012).

Recently, Adamek et al. 2016 developed a relativistic N-body code, `gevolution`, which evolves large scales structures based on the weak field expansion in GR. I investigate the differences in the gravitational dynamics between structures evolved with and without relativistic effects, with focus on the gravitational potential $\Phi$. This is a good choice for comparison as $\Phi$ is gauge invariant and the Newtonian and relativistic simulations are performed in different gauges.

The investigation is done by running 2000 simulations using identical $\Lambda$CDM cosmologies for the two gravity theories. The simulations are run using $64^4$ particles on a $256^3$ grid each with dimension 5120 Mpc/h which is a compromise in order to include both large and nonlinear scales. The data analysis consist of a preliminary analysis using conventional summary statistics, with focus on the bispectrum of $\Phi$. There is a difference in the two cases for low redshifts in the equilateral and squeezed configurations. However, the main idea is to train a Convolutional Neural Network (CNN) to classify the two cases, given snapshots of $\Phi$. The main analysis then involves interpretability of the CNN, which may be done by considering for instance saliency maps (Alqaraawi et al. 2020) or Grad-CAM (Selvaraju et al. 2020). In either case, revealing the features separating the two cases may help us understand the differences in the gravitational dynamics between the two theories. I expect that such a network is able to find relativistic corrections to the Newtonian snapshots that are of higher order than those obtained from power spectra and bispectra analysis. Further, it may also reveal which configurations of Fourier modes $\mathbf{k}$ yield the highest bispectral power, which for now is mainly trial and error.

# Contents

# List of Figures

# List of Tables

# Preface

Here comes your preface, including acknowledgments and thanks.

# Part II

# Machine Learning

# Chapter 6

# Fundamental Elements of Machine Learning

## 6.1 Introduction

In this chapter I will give a brief introduction into machine learning. This includes a mathematical description of some fundamental concepts common across numerous machine learning models. The more advanced models will be dealt with at a later stage. If not otherwise stated, the following chapter is based on Goodfellow, Bengio and Courville 2016 and Hastie, Tibshirani and Friedman 2009.

## 6.2 Linear Algebra

maybe

## 6.3 Probability and Information Theory

maybe

## 6.4 Basic Machine Learning

TODO: Fill more here

### 6.4.1 Optimisation and Generalisation

**Optimisation** Optimsation problems are problems in which we want to minimise some error, given some data. In other words, we want to optimise an algorithm or model given a specific dataset. We care about the error of our model for that specific dataset only.

**Generalisation** The concept of *generalisation* is what makes a machine learning model different from an optimisation model. We still *train* the machine learning model on some specific data, but we measure how good the model is based on how it performs on a different set of data, which it was not trained on. I.e. we need a generalised model, which is not restricted to the data it was trained on, and has the ability to perform well on unobserved data.

### 6.4.2 Data and Fitting

**Data** The key ingredient to any machine learning algorithm is the data fed into it. In an optimisation problem we deal with one single set of data on which we train the model. In a machine learning scenario we want to quantify how well the model is generalised. This is done by training the model on part of the data only, called the *training data*. The model is then assessed on the remaining data, on which it was not trained, in order to determine how general it is. I will come back to how the models are trained, but in essence they are learning features of the data they are trained on. Thus, in order to have a general model we need the data to have some inherent properties. The most important property is that the data is *independent and identically distribution* (i.i.d.). In other words, both the training and testing data are drawn from the same probability distribution.

**Training and testing** When training the model, we want to minimise some error with respect to the training data, which is essentially an optimisation problem. The model is then tested by measuring the same error with respect to the testing date. It is the latter error we want to be low in order to call the model general. The discrepancy between the two are summaries by the concept of fitting.

**Overfitting** Overfitting is when we train the model in such a way that the training error becomes too low. This optimises the model too much with respect to the training data, effectively reconstructing the training data point by point and thereby loosing trends in the data features. The result of this is poor performance on the testing data, and as a result a non-general model.

**Underfitting** Underfitting is the opposite, when the model is unable to achieve a low error on the training data, not capturing the main features of the dataset. This is also bad, as the training error would also be large. This also results in a non-general model.

TODO: Figure explaining over- and underfitting

### 6.4.3 Estimators, Bias, Variance and Error

**Estimators** Based on the assumption that there exists some true parameter(s) $\boldsymbol{\theta}$ which remain unknown,[1] we are able to make predictions and estimations of such parameter(s). Let's say we have $m$ independent and identically distributed (i.i.d.) random variables $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$ drawn from the same probability distribution $p(\mathbf{x})$. An *estimator* of the true values $\boldsymbol{\theta}$ is any function of the data such that $\hat{\boldsymbol{\theta}}_m = g(\mathbf{x}_1, \ldots, \mathbf{x}_m)$, where $\hat{\boldsymbol{\theta}}$ is the estimate of $\boldsymbol{\theta}$. This is known as point estimation, as we are estimating a single value. This definition does not pose any restrictions on the function $g$. However, a good estimator would yield an estimate $\hat{\boldsymbol{\theta}}_m$ that is close to the true value $\boldsymbol{\theta}$.

**Function estimators** Say we want to predict a variable $\mathbf{y}$ given some vector $\mathbf{x}$. We assume the true variable $\mathbf{y}$ is given by some function approxiamation $f(\mathbf{x})$ plus some error $\boldsymbol{\epsilon}$: $\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\epsilon}$. The aim is then to estimate the function $f$ with the estimator $\hat{f}$. If we then realise that $\hat{f}$ is really just a point estimator in function space, the two above concepts are equivalent.

[1]This is the frequentist perspective of statistics

**Bias** The bias of the estimator $\hat{\boldsymbol{\theta}}_m$ is defined as the difference between the expected value of the estimator and the true value of the parameter: $\text{Bias}\left[\hat{\boldsymbol{\theta}}_m\right] = \mathbb{E}\left[\hat{\boldsymbol{\theta}}_m\right] - \boldsymbol{\theta}$. An unbiased estimator has zero bias, i.e. $\mathbb{E}\left[\hat{\boldsymbol{\theta}}_m\right] = \boldsymbol{\theta}$. An estimator is asymptotically unbiased if its bias approaches zero as the number of data points $m$ approaches infinity, i.e. $\lim_{m\to\infty} \mathbb{E}\left[\hat{\boldsymbol{\theta}}_m\right] = \boldsymbol{\theta}$.

**Variance** Variance serves as a crucial metric in assessing the variability of an estimator concerning changes in the input data. Denoted as $\text{Var}\left[\hat{\theta}\right]$, where $\hat{\theta}$ represents the training set, low variance is desirable in an estimator. In essence, variance quantifies the extent to which we anticipate the estimator to fluctuate based on different datasets.

**Standard Error** The standard error, denoted as $\text{SE}\left[\hat{\theta}\right]$, is a fundamental concept closely related to variance, being the square root of the latter. Specifically, for the estimator of the mean, $\hat{\mu}$, the standard error is defined by the formula:

$$\text{SE}\left[\hat{\mu}\right] = \sqrt{\text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \tag{6.1}$$

Here, $\sigma^2$ represents the true variance. The standard error of $\hat{\mu}$ is pivotal in gauging the precision of the estimator for the mean, especially when applied to testing data, aiding in the evaluation of a model's generalization performance.

**Mean Squared Error** The Mean Squared Error (MSE) provides a comprehensive measure for evaluating the performance of an estimator. Expressed as:

$$\text{MSE} = \mathbb{E}\left[(\hat{\theta} - \theta)^2\right] = \text{Bias}\left[\hat{\theta}\right]^2 + \text{Var}\left[\hat{\theta}\right], \tag{6.2}$$

MSE encompasses both the bias and variance components. The MSE offers a unified perspective, capturing the overall accuracy and variability of the estimator in estimating the true parameter $\theta$.

### 6.4.4 Maximum Likelihood Estimation

### 6.4.5 Bayesian Statistics

### 6.4.6 Supervised Learning

### 6.4.7 Unsupervised Learning

# Chapter 7

# Neural Networks

## 7.1 Forward pass - Prediction

### 7.1.1 Computational Graphs

In order to understand how information flows through a neural network we must understand computational graphs. These are collection of *nodes* and *edged*, where each node represents an *operation* and each edge represents the numerical values as they flow through the network. Nodes may be simple functions such as addition or multiplication but also take more complex. In the latter case, they can be further divided into their respective component, so that the most fundamental computational graph only consists fundamental operations.

For example, we consider the functions $\mathbf{y} = g(\mathbf{x})$ and $\mathbf{z} = f(\mathbf{y})$ such that $\mathbf{z} = f(g(\mathbf{x}))$. This can be represented as a computational graph as shown in Fig. 7.1. The computational graph is read from left to right, where the input $\mathbf{x}$ is fed into the function $g$ which outputs $\mathbf{y}$. This $\mathbf{y}$ is then fed into the function $f$ which outputs $\mathbf{z}$. The computational graph is a visual representation of the function $f(g(\mathbf{x}))$.

$$\begin{tikzcd} {} \arrow[r, "\mathbf{x}"] & g \arrow[r, "\mathbf{y}"] & f \arrow[r, "\mathbf{z}"] & {} \end{tikzcd}$$

Figure 7.1: Computational graph of a function $\mathbf{z} = f(g(\mathbf{x}))$.

### 7.1.2 Architecture

The simplest architecture of a neural network is the fully connected feed forward (FCNN), which consist of $L$ layers in total, the first being an input layer and the remaining $L-1$ layers are called *hidden layers*, $\mathbf{h}$. Each hidden layer has an *activation* $\mathbf{a}$, used as an input to an *activation function*, $g$. Mathematically we may write this architecture as:

$$\begin{aligned} \mathbf{h}^0 &= \mathbf{x}^{(i)} \\ \mathbf{h}^1 &= g_1(\mathbf{a}^1) \\ \mathbf{h}^2 &= g_2(\mathbf{a}^2) \\ &\vdots \\ \mathbf{h}^L &= g_L(\mathbf{a}^L), \end{aligned} \tag{7.1}$$

where $\mathbf{h}^L$ and $g_L$ are the output layer and output function respectively. The parameter $L$ governs the depth of the neural network. The result of the output layer is simply called the *output* or *predictor*, and typically denoted as $\hat{\mathbf{y}} = \mathbf{h}^L$. When training, we will have a true value, or label, denoted $\mathbf{y}$. The difference between the true value and the predicted value is called the *residual* or *error*, denoted $\tilde{\mathbf{y}} = \mathbf{y} - \hat{\mathbf{y}}$. The goal of training is to minimise the error, or equivalently, maximise the likelihood of the true value.

### 7.1.3 Activation

The activation $\mathbf{a}^l$ of a layer $l$ is an affine transformation of the output of the previous layer, $\mathbf{h}^{l-1}$. The intercept of this affine transformation is known as the bias $\mathbf{b}^l$, [1] typically used to ensure that no activation becomes zero. The activation takes the form:

$$\mathbf{a}^l = (\mathbf{W}^{l-1\to l})^{\mathrm{T}}\mathbf{h}^{l-1} + \mathbf{b}^l, \tag{7.2}$$

where $\mathbf{W}^{l-1\to l} \in \mathbb{R}^{\dim_{l-1}\times\dim_l}$ is the matrix of weights describing the mapping from layer $l-1$ to layer $l$. Each layer $l$ has dimension (or neurons) $\dim_l$ which governs the width of each layer.

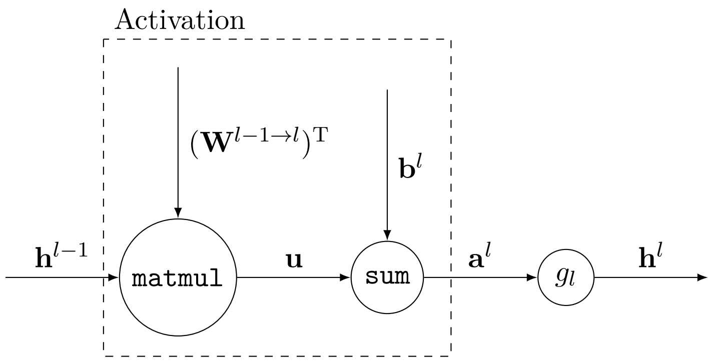

Figure 7.2: Computational graph of a hidden layer of a neural network.

### 7.1.4 Full graph and its output

The full computational graph of a fully connected neural network is shown in Fig. 7.3. The output of the network is the output of the output layer, $\mathbf{h}^L$. The output function is typically a probability distribution, such as the softmax function, or a scalar value, such as the sigmoid function. Either way, the output is the prediction of the network, also noted $\hat{\mathbf{y}} = \mathbf{h}^L$. In order to determine the goodness of this output we must compare it with the target value $\mathbf{y}$. This is done using a *loss function*, $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$, which is a function of the prediction and the target. The loss function is a measure of how good the prediction is. The goal of training is to minimise the loss function, or equivalently, maximise the likelihood of the target value. We define the loss $\delta = \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ which is the error of the prediction. The loss is then used to update the weights and biases of the network, which is done using the *backpropagation* algorithm explained in section TODO: ref this

[1]Must not be confused with the statistical bias of an estimator.

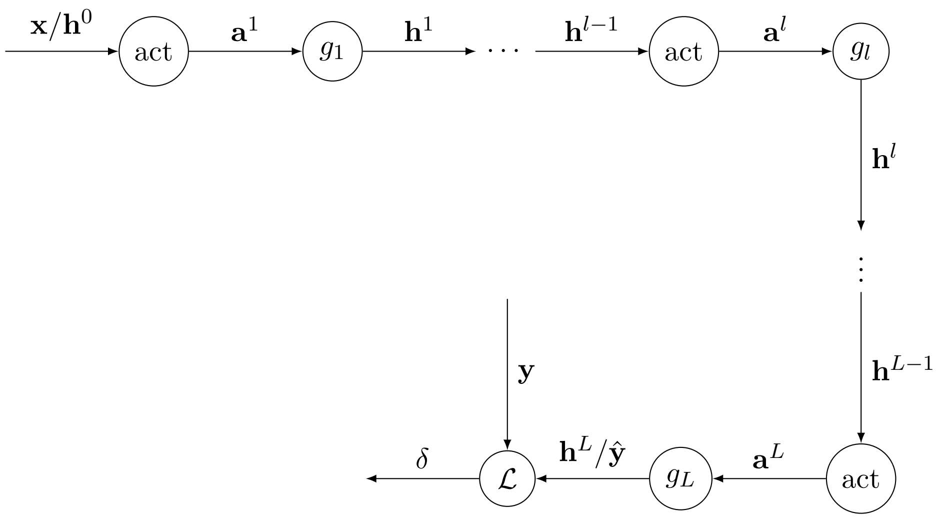

Figure 7.3: Computational graph of a fully connected feed forward neural network.

### 7.1.5 Activation functions

The activation functions, $g_l$, are what determines the output of each layer $l$. They can be broadly classifies into two types: *saturating activation functions* and *non-saturating activation function.* For saturating activations, $\lim_{|\mathbf{x}|\to\infty} |\nabla g(\mathbf{x})| = 0$.

**Linear** The linear activation is as simple as it gets:

$$g(\mathbf{x}) = a\mathbf{x} + b \quad \text{and} \quad g'(\mathbf{x}) = a. \tag{7.3}$$

This activation is non-saturating. There is however a problem with using linear activations, TODO: Write about UNIVERSAL APPROXIMATION THEOREM, here or elsewhere

**Sigmoid** The sigmoid function, or the logistic function is defined as:

$$g(\mathbf{x}) = \sigma(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}}} \quad \text{and} \quad g'(\mathbf{x}) = g(\mathbf{x})(1-g(\mathbf{x})) \tag{7.4}$$

**Hyperbolic tangent** The hyperbolic tangent is defined as:

$$g(\mathbf{x}) = \frac{e^{\mathbf{x}} - e^{-\mathbf{x}}}{e^{\mathbf{x}} + e^{-\mathbf{x}}} \quad \text{and} \quad g'(\mathbf{x}) = 1 - g(\mathbf{x})^2 \tag{7.5}$$

**Rectified linear unit** The rectified linear unit, ReLU, is defined as:

$$g(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} < 0 \\ \mathbf{x}, & \mathbf{x} \geq 0 \end{cases} \quad \text{and} \quad g'(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} < 0 \\ 1, & \mathbf{x} \geq 0. \end{cases} \tag{7.6}$$

**(Scaled) Exponential linear unit** The exponential linear unit, (S)ELU, is defined as:

$$g(\mathbf{x}) = \lambda \begin{cases} \alpha(e^{\mathbf{x}} - 1), & \mathbf{x} < 0 \\ \mathbf{x}, & \mathbf{x} \geq 0 \end{cases} \quad \text{and} \quad g'(\mathbf{x}) = \lambda \begin{cases} \alpha e^{\mathbf{x}}, & \mathbf{x} < 0 \\ 1, & \mathbf{x} \geq 0. \end{cases} \tag{7.7}$$

This function requires two hyperparameters $\alpha$ and $\lambda$.

**Parametric rectified linear unit** The parametric rectified linear unit, PReLU, is defined as:

$$g(\mathbf{x}) = \begin{cases} \alpha\mathbf{x}, & \mathbf{x} < 0 \\ \mathbf{x}, & \mathbf{x} \geq 0 \end{cases} \quad \text{and} \quad g'(\mathbf{x}) = \begin{cases} \alpha, & \mathbf{x} < 0 \\ 1, & \mathbf{x} \geq 0. \end{cases} \tag{7.8}$$

If $\alpha = 0.01$ this function is also known as the leaky rectified unit, or Leaky ReLU.

**Softmax** The softmax activation function is defined as:

$$g(\mathbf{x}) = \frac{e^{\mathbf{x}}}{\sum_{i=1}^{n} e^{x_i}} \quad \text{and} \quad g'(\mathbf{x}) = g(\mathbf{x})(1 - g(\mathbf{x})). \tag{7.9}$$

This activation function is typically used for the output layer of a neural network, where the output is a probability distribution over $n$ classes, i.e. multiclass classification problems.

### 7.1.6 Loss functions

TODO: write about loss functions

**MSE**

$$\mathcal{L}_{\mathrm{MSE}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\tilde{\mathbf{y}}^{\mathsf{T}}\tilde{\mathbf{y}}, \tag{7.10}$$

where $\tilde{\mathbf{y}} \equiv \mathbf{y} - \hat{\mathbf{y}}$

**Cross entropy**

$$\mathcal{L}_{\mathrm{CE}}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \tag{7.11}$$

where $\hat{y}_i$ is the predicted probability of class $i$ and $y_i$ is the true probability of class $i$, i.e. the target probability for that class. $n$ is the number of classes. In the binary case, $n = 2$, this is equivalent to the binary cross entropy:

$$\mathcal{L}_{\mathrm{BCE}}(\mathbf{y}, \hat{\mathbf{y}}) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y}). \tag{7.12}$$

### 7.1.7 Regularization

## 7.2 Backpropagation - Training

### 7.2.1 Chain rule

**Basics** Lets say we have a function $f(x)$ that is composed of two functions $g(x)$ and $h(x)$, such that $f(x) = g(h(x))$. The chain rule states that the derivative of $f(x)$ with respect to $x$ is the product of the derivative of $g(x)$ with respect to $h(x)$ and the derivative of $h(x)$ with respect to $x$:

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx}. \tag{7.13}$$

This can be generalised into the vector case where $f(\mathbf{x}) = g(h(\mathbf{x}))$ and $\mathbf{x} \in \mathbb{R}^n$ and TODO: write more ahaha

### 7.2.2 Backpropagation algorithm

The learnable parameters of a neural network are the weights and biases of each layer. In order to update these in an educated[2] way, we need to know how the loss, $\delta$, changes with respect to each parameter, $\nabla_{\mathbf{b}^l}\delta$, $\nabla_{\mathbf{W}^{l-1\to l}}\delta \quad \forall\ l \in [1, L]$ . This is done using the chain rule, starting from the output layers and working our way backwards through the network. Recalling that the activation is written as Eq. (7.2) it makes sense to find the derivative of the loss with respect to the output activation:

$$\frac{\mathrm{d}\delta}{\mathrm{d}a_i^L} = \sum_j \frac{\partial\delta}{\partial h_j^L}\frac{\mathrm{d}h_j^L}{\mathrm{d}a_i^L} = \sum_j \frac{\partial\delta}{\partial\hat{y}_j}\frac{\mathrm{d}g_L}{\mathrm{d}a_i^L}\bigg|_{a_j^L} \leftrightarrow \nabla_{\mathbf{a}^L}\delta = \nabla_{\hat{\mathbf{y}}}\delta \odot \frac{\mathrm{d}g_L}{\mathrm{d}\mathbf{a}^L}\bigg|_{\mathbf{a}^L}, \tag{7.14}$$

where $\odot$ is the *element-wise Hadamard product*. The quantity $\nabla_{\mathbf{a}^L}\delta$ is the gradient of the loss with respect to the output activation. This is propagated backwards through the network as follows:

$$\nabla_{\mathbf{h}^{l-1}}\delta = \mathbf{W}^{l-1\to l}\nabla_{\mathbf{a}^l}\delta \quad \text{and} \quad \nabla_{\mathbf{a}^{l-1}}\delta = \nabla_{\mathbf{h}^{l-1}}\delta \odot \frac{\mathrm{d}g_{l-1}}{\mathrm{d}\mathbf{a}^{l-1}}\bigg|_{\mathbf{a}^{l-1}}. \tag{7.15}$$

When we know these loss gradients we can compute the gradient of the weight and biases:

$$\nabla_{\mathbf{W}^{l-1\to l}}\delta = \mathbf{h}^{l-1}(\nabla_{\mathbf{a}^l}\delta)^T \quad \text{and} \quad \nabla_{\mathbf{b}^l}\delta = \nabla_{\mathbf{a}^l}\delta. \tag{7.16}$$

We further define a parameter tuple $\Theta^l \equiv (\mathbf{W}^{l-1\to l}, \mathbf{b}^l)$ and a gradient tuple $\nabla_{\Theta^l}\delta \equiv (\nabla_{\mathbf{W}^{l-1\to l}}\delta, \nabla_{\mathbf{b}^l}\delta)$. The weight and biases of each layer is then updated as $\Theta^l \leftarrow \mathcal{U}(\Theta^l, \nabla_{\Theta^l}\delta)$, where $\mathcal{U}$ is an update function. This is done for each layer $l$ in the network. The backpropagation algorithm is summarised in Algorithm 1. TODO: Revise this

**Algorithm 1** Backpropagation algorithm

**Require:** Loss $\delta$, output activation $\mathbf{a}^L$, output layer $\mathbf{h}^L$, weight and bias tuples $\Theta^l \equiv (\mathbf{W}^{l-1\to l}, \mathbf{b}^l)$, activation functions $g_l$ and update function $\mathcal{U}$.
**Ensure:** Updated weight and bias tuples $\Theta^l \equiv (\mathbf{W}^{l-1\to l}, \mathbf{b}^l)$.

```
1: $\nabla_{\mathbf{a}^L}\delta \leftarrow \nabla_{\hat{\mathbf{y}}}\delta \odot \frac{\mathrm{d}g_L}{\mathrm{d}\mathbf{a}^L}\Big|_{\mathbf{a}^L}$
2: for $l = L, L-1, \ldots, 1$ do
3:     $\nabla_{\mathbf{h}^{l-1}}\delta \leftarrow \mathbf{W}^{l-1\to l}\nabla_{\mathbf{a}^l}\delta$
4:     $\nabla_{\mathbf{a}^{l-1}}\delta \leftarrow \nabla_{\mathbf{h}^{l-1}}\delta \odot \frac{\mathrm{d}g_{l-1}}{\mathrm{d}\mathbf{a}^{l-1}}\Big|_{\mathbf{a}^{l-1}}$
5:     $\nabla_{\mathbf{W}^{l-1\to l}}\delta \leftarrow \mathbf{h}^{l-1}(\nabla_{\mathbf{a}^l}\delta)^T$
6:     $\nabla_{\mathbf{b}^l}\delta \leftarrow \nabla_{\mathbf{a}^l}\delta$
7:     $\Theta^l \leftarrow \mathcal{U}(\Theta^l, \nabla_{\Theta^l}\delta)$
8: end for
```

algorithm

### 7.2.3 Optimization of parameters

**Gradient descent** The ultimate goal is to arrive at some parameters $\Theta$ that minimize the output loss $\delta$. In order to do so, we must update the parameters until this goal is reached. The question is then *how* we should update these parameters. In the most

[2]Instead of just wildly guessing

general case, we need to find some $\Delta\Theta$ such that $\Theta \leftarrow \Theta + \Delta\Theta$, where the only condition is that $\Delta\Theta$ yield a new $\Theta$ whose loss decrease. The backbone of this thinking is the *gradient descent* algorithm, which is a first order optimization algorithm. The idea is that since we want to minimize the loss, which is a function of $\Theta$, we should move in the direction of the negative gradient of the loss with respect to $\Theta$. the most naïve implementation of this is to simply advance in the direction of the negative gradient by a step given by the learning rate[3] $\gamma$: $\Delta\Theta = -\gamma\nabla_\Theta\delta$. This is known as *batch gradient descent* and is the most basic form of gradient descent. This would leave us with the following update function:

$$\mathcal{U} = \Theta - \gamma\nabla_\Theta\delta. \tag{7.17}$$

There are several problems with this approach, the most important of them being:

- The learning rate $\gamma$ is constant, which means that the step size is constant. This can lead to the algorithm getting stuck in local minima.
- The gradient is computed using the entire dataset, which can be very computationally expensive.

**Stochastic gradient descent** To our rescue comes stochastic gradient descent (SGD), which solves both of the above issues, by estimating the gradient using only a subset of the data, called a *mini-batch*. This means that the gradient is computed as an average over the mini-batch, which is a good approximation of the true gradient since the gradient of a dataset is statistically an expectation value. This allows us to increase the dataset without increasing the cost of computing the gradients. We may use several such minibatches chosen at random, say $m$ batches. This would yield the following estimate for the gradient:

$$\nabla_\Theta\delta \approx \frac{1}{m}\sum_{i=1}^{m}\nabla_\Theta\delta_i, \tag{7.18}$$

where $\delta_i$ is the loss from minibatch $i$. This is known as *mini-batch gradient descent*. The update function for SGD is then identical to Eq. (7.17) but with the gradient estimate from Eq. (7.18). Choosing minibatches at random also greatly reduce the chances of being stuck in a local minimum.

**Adding momentum** In order to speed up convergence we may add a momentum term, which takes into account the previously found gradients. This is done by adding a momentum term, called the velocity $\mathbf{v}$, to the update function. We start with some initial velocity, and it is updated as $\mathbf{v} \leftarrow \alpha\mathbf{v} - \gamma\nabla_\Theta\delta$, where $\alpha$ is the momentum hyperparameter, which controls the weight of the previous velocity (and thus the previous gradients). This yields simply $\Delta\Theta = \mathbf{v}$ and the following update function:

$$\mathcal{U} = \Theta + \mathbf{v}. \tag{7.19}$$

TODO: Nesterov momentum?

**Adaptive momentum** One algorithm that adapts the momentum term is the *Adam* algorithm TODO: ref Kinga and Ba which uses the first and second moments of the gradient to adapt the momentum term. The first moment is the mean of the gradient and the second moment is the variance of the gradient.

**Algorithm 2** Adam

```
Require: Learning rate $\gamma$, decay rates $\beta_1$, $\beta_2 \in [0, 1)$, first moment $\mathbf{m} = \mathbf{0}$, second
    moment $\mathbf{v} = \mathbf{0}$, weight decay rate $\epsilon$ (for numerical stability), initial parameters $\Theta_0$,
    and timestep $t = 0$.
 1: while $\Theta_t$ not converged do
 2:     $t \leftarrow t + 1$ (Update timestep)
 3:     $\mathbf{g}_t \leftarrow \nabla_{\Theta_{t-1}} \delta$ (Compute gradient at timestep $t$)
 4:     $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$ (Update biased first moment estimate)
 5:     $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t \odot \mathbf{g}$ (Update biased second moment estimate)
 6:     $\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\beta_1^t}$ (Bias-corrected first moment estimate)
 7:     $\hat{\mathbf{v}}_t \leftarrow \frac{\mathbf{v}_t}{1-\beta_2^t}$ (Bias-corrected second moment estimate)
 8:     $\Delta\Theta_t = -\gamma \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t + \epsilon}}$ (Compute update)
 9:     $\Theta_t \leftarrow \Theta_{t-1} + \Delta\Theta_t$ (Update parameters)
10: end while
        return $\Theta_t$ (Resulting parameters)
```

The adam algorithm is given as: The hyperparameters $\beta_1$ and $\beta_2$ are typically set to 0.9 and 0.999 respectively, and $\epsilon$ is typically set to $10^{-8}$. The learning rate $\gamma$ is typically set to 0.001.

#### Learning rate schedules?

#### Batch normalization

### 7.2.4 Challenges in optimization

#### Ill-conditioning

#### Vanishing gradient

#### Exploding gradient

#### Local minima

[3] $\gamma$ is also often referred to as the step size

Chapter 8

# Convolutional Neural Networks

## 8.1 Convolution

### 8.1.1 Basic definitions

Mathematically, convolution is given as:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)\mathrm{d}a. \tag{8.1}$$

This definition is purely mathematical. For machine learning purposes we refer to the function $x(a)$ as the *input* and $w(t-a)$ as the *kernel.* The output of the convolution is often called the *feature map.* This is easily generalized to higher dimensions. For example, in two dimensions the discrete convolution between an input image $I$ and a kernel $K$ is given as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{8.2}$$

If we choose to *flip the kernel* relative to the input $I$, the convolution becomes commutative:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{8.3}$$

The latter is preferred in a machine learning context as there is less variation in the values of $m$ and $n$ because the size of the kernel is often much smaller than the size of the input. However, there is yet another quantity that is often used in machine learning, namely the *cross-correlation*, which is more intuitive in some sense, because the kernel is not flipped. The cross-correlation is given as:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{8.4}$$

### 8.1.2 Properties

## 8.2 New Layers

### 8.2.1 Convolutional layers

### 8.2.2 Pooling layers

## 8.3 Implementation

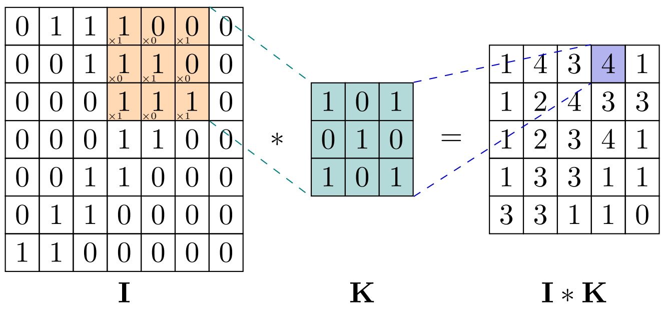

Figure 8.1: Example of a convolution operation. The input image is given by the matrix $\mathbf{I}$ and the kernel is given by the matrix $\mathbf{K}$. The output is given by the matrix $\mathbf{I} * \mathbf{K}$. The dashed lines indicate which elements of the input and kernel are multiplied together to produce the output.

# Bibliography

Adamek, Julian et al. (29th July 2016). 'gevolution: a cosmological N-body code based on General Relativity'. In: *Journal of Cosmology and Astroparticle Physics* 2016.7, pp. 053–053. ISSN: 1475-7516. DOI: 10.1088/1475-7516/2016/07/053. arXiv: 1604.06065[astro-ph,physics:gr-qc,physics:physics]. URL: http://arxiv.org/abs/1604.06065 (visited on 23/08/2023).

Alqaraawi, Ahmed et al. (3rd Feb. 2020). *Evaluating Saliency Map Explanations for Convolutional Neural Networks: A User Study.* arXiv: 2002.00772[cs]. URL: http://arxiv.org/abs/2002.00772 (visited on 23/10/2023).

Chisari, Nora Elisa and Matias Zaldarriaga (2nd June 2011). 'Connection between Newtonian simulations and general relativity'. In: *Physical Review D* 83.12, p. 123505. ISSN: 1550-7998, 1550-2368. DOI: 10.1103/PhysRevD.83.123505. arXiv: 1101.3555[astro-ph,physics:gr-qc]. URL: http://arxiv.org/abs/1101.3555 (visited on 23/08/2023).

Goodfellow, Ian, Yoshua Bengio and Aaron Courville (2016). *Deep Learning.* http://www.deeplearningbook.org. MIT Press.

Green, Stephen R. and Robert M. Wald (15th Mar. 2012). 'Newtonian and Relativistic Cosmologies'. In: *Physical Review D* 85.6, p. 063512. ISSN: 1550-7998, 1550-2368. DOI: 10.1103/PhysRevD.85.063512. arXiv: 1111.2997[astro-ph,physics:gr-qc]. URL: http://arxiv.org/abs/1111.2997 (visited on 23/08/2023).

Hastie, T., R. Tibshirani and J.H. Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer series in statistics. Springer. ISBN: 9780387848846. URL: https://books.google.no/books?id=eBSgoAEACAAJ.

Jeong, Donghui, Fabian Schmidt and Christopher M. Hirata (4th Jan. 2012). 'Large-scale clustering of galaxies in general relativity'. In: *Physical Review D* 85.2, p. 023504. ISSN: 1550-7998, 1550-2368. DOI: 10.1103/PhysRevD.85.023504. arXiv: 1107.5427[astro-ph]. URL: http://arxiv.org/abs/1107.5427 (visited on 23/08/2023).

Selvaraju, Ramprasaath R. et al. (Feb. 2020). 'Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization'. In: *International Journal of Computer Vision* 128.2, pp. 336–359. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-019-01228-7. arXiv: 1610.02391[cs]. URL: http://arxiv.org/abs/1610.02391 (visited on 23/10/2023).