Learning by Digging

A Differentiable Prediction Model for an Autonomous Wheel Loader

Arvid Fälldin

June 10, 2022

Master's Thesis in Engineering Physics
Umeå University
June, 2022

Author: Arvid Fälldin
Supervisor: Martin Servin
Examinor: Erik Wallin

Abstract

Wheel loaders are heavy duty machines that are ubiquitous on construction sites and in mines all over the world. Fully autonomous wheel loaders remains an open problem but the industry is hoping that increasing their level of autonomy will help to reduce costs and energy consumption while also increasing workplace safety. Operating a wheel loader efficiently requires dig plans that extend over multiple dig cycles and not just one at a time. This calls for a model that can predict both the performance of a dig action and the resulting shape of the pile. In this thesis project, we use simulations to develop a data-driven artificial neural network model that can predict the outcome of a dig action. The model is able to predict the wheel loader's productivity with an average error of 7.3%, and the altered shape of the pile with an average relative error of 4.5%. We also show that automatic differentiation techniques can be used to accurately differentiate the model with respect to input. This makes it possible to use gradient-based optimization methods to find the dig action that maximises the performance of the wheel loader.

Acknowledgements

I would like to extend my thanks to all members of the Digital physics group at UMIT Research Lab for their support during the project, especially Koji Aoshima for being my assistant supervisor in all but name.

Contents

Chapter 1

Introduction

1.1 Background

Operating a wheel loader is generally a very versatile task that can involve anything from driving on public roads to moving gravel on a construction site. There are however some subtasks which can be repetitive and it is tempting to automate them. There already exists semi-autonomous systems for wheel loaders, e.g. for assisted digging, but fully autonomous wheel loading remains an open problem. One subtask that is especially tempting to automate is the so-called *short loading cycle*. It consists of picking up material from a large pile and dumping it onto a truck positioned nearby. If this task could be successfully automated it is easy to see how it could be generalized to many other tasks as well. The short loading cycle itself is typically divided into separate stages. Their definition can vary between authors but we chose the following partitioning:

1. Approach the pile.
2. Fill the bucket.
3. Retract from the pile.
4. Approach the dumper.
5. Empty the bucket.
6. Retract from the dumper.

The cycle is illustrated in Figure 1.1. It is repeated until the dumper is fully loaded or all material has been removed. In this project we will focus mainly on stages 1 and 2.

Stage 1 is defined to not only include navigating from the turning point to the pile, but also the act of choosing the next position where to dig. This makes it a non-trivial problem even for experienced operators. When forming a dig plan, it is important to not only optimize one dig action at a time, as this may lead to a poor pile state in the long run. Instead, one should optimize a sequence of dig actions. Since each dig action alters the state of the pile, this would require an ability to approximately predict the new shape of a pile resulting from each dig action.

Stage 2, bucket filling, is perhaps the stage of most interest and possibly also the hardest to automate. The goal is to maximize the amount of material in the bucket while minimizing the time and energy consumed. Experiments show that the bucket filling typically accounts for around 35-40% of the fuel consumption but only 25% of the cycle time [2]. This has made it an important area of research for the industry

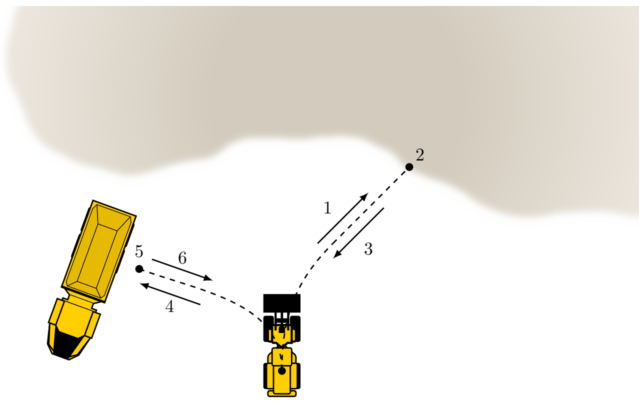

Figure 1.1: Illustration of the short loading cycle as described in [1].

and a lot of effort has been put into finding the most optimal bucket-filling strategy. Discrete element method (DEM) simulations suggest that a good strategy is to aim for a bucket tip trajectory known as the “Slicing cheese” strategy, which is illustrated in Figure 1.2. This result was shown for a planar, steep slope [3]. In general, the optimal bucket trajectory is likely heavily dependent on the local shape of the pile. However, even if one knows in advance which bucket trajectory is optimal, is it often ill-advised to try to make the bucket follow a pre-defined path (trajectory control) [4]. Material inhomogeneties alone can make the dynamics very unpredictable and can easily throw a bucket off its path. The Digital Physics group at UMIT Research Lab specializes in

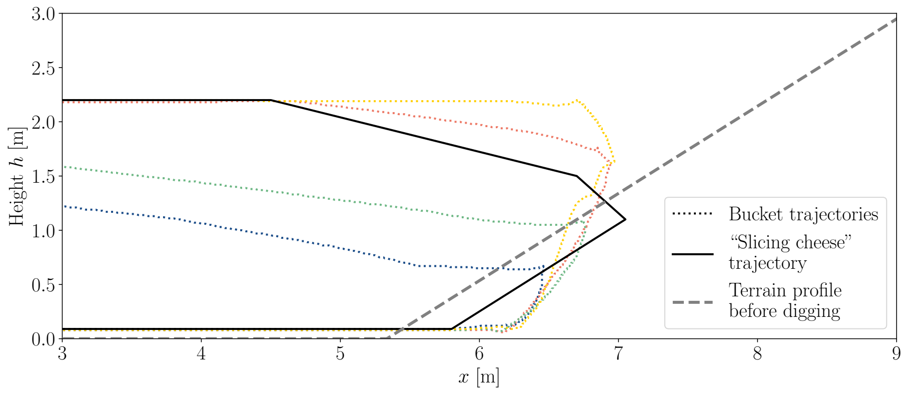

Figure 1.2: The “Slicing cheese” bucket trajectory along with four trajectories created by our autonomous loader.

how one can represent complex physical systems using virtual environments that stay true to the laws of physics. One of their current projects is to use a digital twin and simulations to develop an autonomous bucket controller for wheel loaders. The controller they are developing belongs to a category known as *compliance controllers*. By having the wheel loader’s boom and bucket actuators respond to the resistance forces met from the pile, they have created a controller that can adapt to the surrounding dynamics in a way that a trajectory controller cannot. Instead of feeding the controller a pre-defined bucket trajectory, it is given a set of *action variables* $\mathbf{a}$ that control *how* and *when* the wheel loader should react to the pile resistance. Much like the bucket trajectories, the optimal choice of $\mathbf{a}$ is highly dependent on the material and the local shape of the pile. It is possible to simulate the results of dig action $\mathbf{a}$ applied to the pile state $\mathbf{s}$ in real time.

However, even real time simulation is not fast enough to efficiently find the optimal dig plan for an autonomous wheel loader. This is where this thesis project comes in.

1.2 Aim

The aim of this master's thesis is to develop a data-driven model f that, given the current pile state $\mathbf{s}^{(i)}$ and a set of action variables $\mathbf{a}$, can accurately predict the simulated outcome $\mathbf{y}$ of the dig action

$$\mathbf{y} = f(\mathbf{a}, \mathbf{s}). \tag{1.1}$$

The outcome in this context is defined to include both a set of performance metrics $\mathcal{P}_j$, and the new, altered pile state $\mathbf{s}^{(i+1)}$. With such a model in place, one could use it to approximately solve the optimization problem

$$\max_{\mathbf{a}} \sum_{i=1}^{N} \mathbf{w}^\top \mathcal{P}^{(i)} \left(\mathbf{a}^{(i)}, \mathbf{s}^{(i)} \right), \tag{1.2}$$

i.e. maximizing the wheel loader's performance over a sequence of N dig actions. Here, $\mathbf{w}$ is a vector of weights that can be used to balance the importance of different performance metrics $\mathcal{P}_j$. The idea of optimizing sequences of dig actions rather than one by one is of especial importance for wheel loaders, as seemingly good dig actions could lead to a poor pile state that decreases the performance in the long run.

The model should be constructed to be fully differentiable with respect to the action variables $\mathbf{a}$, without having to resort to finite differences. Finite differences can quickly become impractical if the number of input and output variables grow large, and the quality of the derivatives could be subject to discretization errors.

An accurate, fully differentiable prediction model would allow us to solve the optimization problem (1.2) using gradient based methods. The model could then be used during dig planning in the short loading cycle to find the, statistically speaking, best sequence of actions given the current shape of the pile.

1.3 Delimitations

Solving the optimization problem in equation (1.2) is beyond the scope of this thesis project. Our focus will rather be on developing one of many tools needed for solving it in the future. We will also disregard the problem of navigating the wheel loader from the turning point to the pile. Lastly, we will only consider simulated data and will not make any attempts at verifying our models via field experiments.

1.4 Related work

Apart from the research group's previous work [5], [6], the main inspiration for this thesis project has been the work by Saku et al. [7]. They used a convolutional autoencoder paired with a Long short-term memory (LSTM) network as a time-stepper to predict soil deformations caused by an excavator bucket. On the topic of differentiable neural network models, the work by Montes et. al [8] provided a lot of insight regarding network architectures, especially the impact of using activation functions which are differentiable everywhere.

Chapter 2

Theory

2.1 Neural networks basics

Artificial neural networks or simply neural networks are computation systems paired with learning algorithms. They are loosely inspired by how neurons in animal brains process information. When employed correctly they can be a very powerful tool and over the past decade they have been used to advance the performance of machine learning in everything from medical image analysis to natural language processing. The theory behind neural networks is vast and in this section we will only cover some of the basic concepts of neural networks, especially the ones that explain how we will be able to treat them as differentiable functions later on. Unless otherwise stated, we have used the book *Deep Learning* [9] as reference for this section.

2.1.1 Artificial neurons and fully connected neural networks

The smallest building blocks in neural networks are knows as *artificial neurons* or *units*. An artificial neuron is a function $f : \mathbb{R}^n \to \mathbb{R}$, typically consisting of an affine transformation of the input $\mathbf{x}$, composed with an activation function ϕ,

$$\hat{y} = f(\mathbf{x}; b, \mathbf{w}) = \phi(\mathbf{w}^\top \mathbf{x} + b), \quad \mathbf{w} \in \mathbb{R}^n, \quad b \in \mathbb{R}. \tag{2.1}$$

The structure of such a neuron is illustrated in Figure 2.1. We say that a neuron is *active* or *firing* when it outputs a non-zero value.

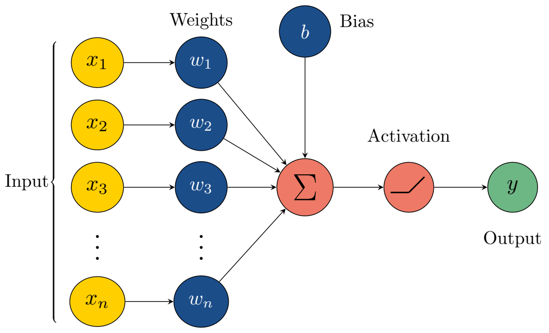

Figure 2.1: Typical structure of an artificial neuron. Each input is multiplied by a weight factor before being summed. An offset/bias term is usually added as well before the summed value is passed through an activation function.

Neurons are typically grouped together in a hierarchical structure to form layers, see Figure 2.2. The neurons in one layer receives data from the neurons in the preceding layer and output data to the succeeding one. The computations carried out by a layer of m neurons can collectively be seen as a single function, $f : \mathbb{R}^n \to \mathbb{R}^m$,

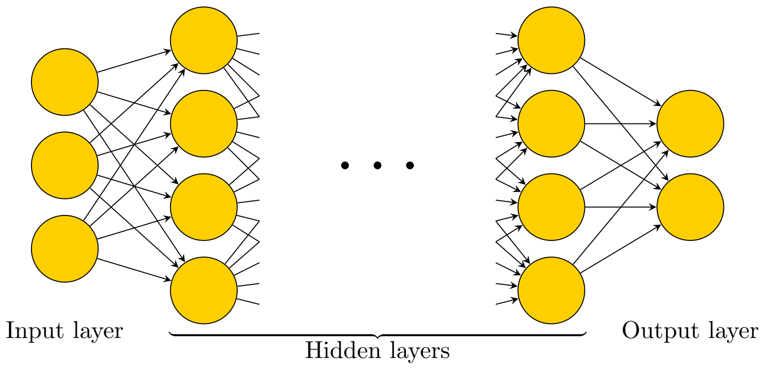

Figure 2.2: Typical structure of a fully connected neural network.

$$\hat{\mathbf{y}} = f(\mathbf{x}; b, \mathbf{W}) = \phi(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) \in \mathbb{R}^m. \tag{2.2}$$

Again, this is an affine transformation of the input composed with an activation function ϕ, which is now a vector-valued function. When using a network for model fitting, we want to find the set of weights and biases $\mathbf{W}, \mathbf{b}$ that gets our predicted responses $\hat{\mathbf{y}}$ as close to the observed responses $\mathbf{y}$ as possible. Using θ to refer to the weights and biases collectively, we can formulate this as an optimization problem

$$\min_\theta \ell(\hat{\mathbf{y}}, \mathbf{y})$$

where ℓ is the *loss function*, which we use to evaluate the model's goodness of fit. A low value of ℓ corresponds to a good fit, so we want to minimize it. If we choose ϕ in equation (2.2) to be the identity function and the mean squared error (MSE)

$$\ell(\hat{\mathbf{y}}, \mathbf{y}) = ||\hat{\mathbf{y}} - \mathbf{y}||_2^2,$$

as our loss, then it reduces to a linear regression model. In that case we can find a closed form solution for the optimal model parameters $\mathbf{W}, \mathbf{b}$. For most other choices of activation functions however, we do not have a closed form solution and instead we need to resort to gradient based methods during model fitting. There, the main idea is to make an initial guess $\theta^{(0)}$ of the model parameters and use it to make a prediction $\hat{\mathbf{y}}$. By taking the gradient of the loss function with respect to the model parameters, we can make a new update to the parameters $\theta^{(1)}$ as

$$\theta^{(i+1)} = \theta^{(i)} - \rho \frac{\partial \ell}{\partial \theta}. \tag{2.3}$$

Since the gradient always points in the direction of steepest ascent, a step in the opposite direction should, under some assumptions, take us closer to a local minima. ρ controls the length of such a step and in deep learning it is known as the *learning rate*. To train the network, we use equation (2.3) iteratively until the desired accuracy has been obtained or we stop improving. Computing the derivative of one layer is simple enough. However, as we will see in the next subsection, the true potential of neural networks is reached when one composes many layers with each other, forming a chain of dependencies

$$\hat{\mathbf{y}} = \phi_L \circ \phi_{L-1} \circ \ldots \circ \phi_0(\mathbf{x}). \tag{2.4}$$

As the depth L of the network increases, the dependencies get very complicated and computing them by hand and/or programming them explicitly quickly becomes unfeasible. For this reason, neural network implementations use automatic differentiation methods to compute derivatives during training.

2.1.2 The universal function approximation theorem

The expressive capabilities of neural networks are given solid footing by the *universal approximation theorem*. The theorem states that a neural network with a linear output layer and at least one hidden layer with a "squashing" type activation function e.g. tanh, can be used to approximate any function that is continuous on a closed and bounded subset of $\mathbb{R}^n$ with arbitrarily small error, given that enough hidden units are used. The result also applies to the function's first derivative [10]. Since the theorem was published in 1989, it has further been shown that any non-polynomial activation function would work [11]. Note that this result only tells us that it is possible to approximate any continuous function with a neural network, it does not tell us anything about how to achieve the desired accuracy. The theorem proves that a single hidden layer is enough, but to achieve the desired performance with a one-layer network one would usually need a huge number of neurons. It has been shown that one can get away with a more lightweight network by increasing the model depth instead [12].

2.1.3 Activation functions

The universal approximation theorem states that almost any activation function would work. In practice some choices are more appropriate than others. Typically, one uses the same activation for all hidden layers. In recent years the rectified linear unit, ReLU, have been the go-to choice for hidden layers as it does not suffer from the same vanishing gradient problems that sigmoid-shaped activation functions do. The success with ReLU has given rise to a whole family of similarly-shaped activations, sometimes collectively referred to as the ReLU family. We will now have a closer look at four members of the ReLU family. Graphs of all four functions and their respective first derivatives are shown in Figure 2.3.

ReLU & LeakyReLU

The ReLU function has been very popular in use over the past decade and has been used as the hidden layer activation in many state-of-the-art architectures. It is defined as

$$\mathrm{ReLU} = \max(0, x),$$

and has first derivative

$$\frac{d}{dx}\mathrm{ReLU}(x) = \begin{cases} 0, & x < 0 \\ \text{Undefined}, & x = 0 \\ 1, & x > 0. \end{cases}$$

Many neural network implementations ignore the singularity at $x = 0$ and assigns the derivative the value 0 or 1 at this point instead of raising a NaN error [13]. PyTorch

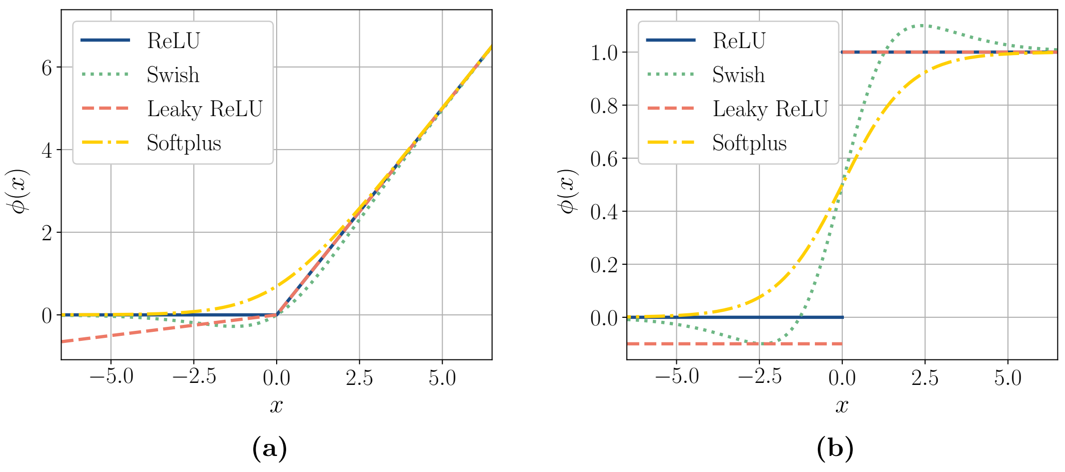

Figure 2.3: (a) Activation functions from the ReLU family and (b) their first derivatives. The LeakyReLU shown uses $\alpha = 0.1$ and both the Softplus and the Swish uses $\beta = 1$. Note that the derivatives of LeakyReLU and ReLU both have a jump discontinuity at $x = 0$.

for example uses $\mathrm{ReLU}'(0) = 0$. We will do the same hereafter and ask the reader to excuse this slight abuse of the derivative notation. With the extension $\mathrm{ReLU}'(0) = 0$, the derivative of the ReLU is the well-known Heaviside step function

$$\theta(x) = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0. \end{cases} \tag{2.5}$$

ReLU has the advantage of being very cheap computationally while also allowing gradients to flow past it whenever the neuron is active. This property solves the vanishing gradient problem in the case when the input is large. A very close relative to ReLU is the LeakyReLU,

$$\mathrm{LeakyReLU} = \max(x, \alpha x)$$

for some value of $\alpha > 0$. α is usually taken to be very small ~ 0.01. This gives the LeakyReLU a very small slope when $x < 0$, allowing gradients to flow in this interval as well.

SoftPlus

The Softplus function is defined as

$$\mathrm{Softplus}(x) = \frac{1}{\beta}\ln(1 + e^{\beta x})$$

and was introduced to deep learning as a smooth version of ReLU. $\beta > 0$ is known as the *sharpness* parameter, as increasing it will bring the Softplus closer to the shape of a ReLU. Unlike ReLU it is infinitely differentiable everywhere and the derivative is given by

$$\frac{d}{dx}\mathrm{Softplus}(x) = \frac{1}{1 + e^{-\beta x}}, \tag{2.6}$$

also known as the logistic sigmoid function $\sigma(\beta x)$. At the time of its introduction, it was hypothesized that its smoothness would be beneficial during training, but empirical studies carried out since suggest otherwise, and it is generally advised against using it as an activation in favor of ReLU. The softplus function is also much more computationally expensive than ReLU.

Swish

Our final activation function is the swish function, introduced to deep learning in 2017 [14], defined as

$$\text{Swish}(x) = x\sigma(\beta x) = \frac{x}{1+e^{-\beta x}} \tag{2.7}$$

and has derivative

$$\frac{d}{dx}\text{Swish}(x) = \sigma(\beta x)\left\{1 + \beta x[1-\sigma(\beta x)]\right\},$$

where β again can be interpreted as a sharpness parameter and can either be a constant or a trainable parameter. When $\beta = 1$ it is sometimes called the *sigmoid weighted linear unit* or SiLU. It approaches ReLU in the limit $\beta \to \infty$. Just like Softplus it sacrifices computational efficiency in favor of smoothness. It can produce a small negative output and allows gradients to flow even if the neuron itself is close to being inactive. It has been claimed to be a better choice than ReLU for deep architectures as it does not suffer as much from the dying ReLU problem. Montes et al. used Swish to create a differentiable neural network prediction model for the potential energy in ionic liquids [8].

2.2 Variational Autoencoders

Predicting the altered pile shape resulting from a dig action is more complicated than simply subtracting height values from one part of the heightfield. The prediction is less about highlighting a feature that is present in the initial pile and more about generating a whole new pile shape. Because of this we will use the generative machine learning method of Variational Autoencoders (VAEs). VAEs can be seen as a probabilistic extension of traditional autoencoders and unlike their deterministic predecessors, they try to approximate distributions rather than functions. The theory behind them is very general and we will only consider a very special case. For more details about VAEs, the original paper by Kingma and Welling is a great source of insight [15]. For this theory section we have also used [9], [16], [17] as reference.

2.2.1 The Kullback-Leibler divergence

As we will see later on when we define our loss function, a central concept in the theory behind VAEs is the *Kullback-Leibler divergence.* Let $p(\mathbf{x}), q(\mathbf{x})$ be two distributions of the same continuous variable $\mathbf{x}$. Then the Kullback-Leibler divergence of $p(\mathbf{x})$ from $q(\mathbf{x})$ is given by

$$D_{\text{KL}}(p(\mathbf{x})||q(\mathbf{x})) = \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[\ln(p(\mathbf{x})) - \ln(q(\mathbf{x}))]$$

where $\mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[f(\mathbf{x})]$ is used to denote the expectation value of function $f(\mathbf{x})$, given that $\mathbf{x} \sim p(\mathbf{x})$. To avoid having to go into details, we can think of the divergence as a measure in how different $q(\mathbf{x})$ is from $p(\mathbf{x})$. A value of 0 means that the two distributions are identical, and a large value means that they are very different from each other. The Kullback-Leibler divergence is always non-zero.

2.2.2 Model formulation and assumptions

Suppose we have a collected a large dataset $\mathcal{D} = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$, where $\mathbf{x}^{(i)}$ are independent and identically distributed samples of a random variable $\mathbf{x} \sim p(\mathbf{x})$. In our model, the local pile shape variable $\mathbf{s}$ will play the part of $\mathbf{x}$. Now assume that the variable $\mathbf{x}$ is generated via a two-step process involving an unobserved variable $\mathbf{z}$:

1. A value of $\mathbf{z} \in \mathbb{R}^k$ is sampled from a prior distribution $p_\theta(\mathbf{z})$.

2. A value of $\mathbf{x}$ is then sampled from the conditional distribution $p_\theta(\mathbf{x}|\mathbf{z})$.[1]

We also assume that $p_\theta(\mathbf{z})$ and $p_\theta(\mathbf{x}|\mathbf{z})$ belong to known families of distributions parameterized by neural networks, which in turn are parameterized by θ. For all intents and purposes, we can think of $\mathbf{z}$ as a low-dimensional latent representation of the high-dimensional observed variable $\mathbf{x}$. Now consider the special case where

$$\begin{aligned}
\mathbf{z} &\sim \mathcal{N}(0, \mathbf{I}) \\
\boldsymbol{\mu}_\mathbf{x}(\mathbf{z}) &= \text{Decoder}_\theta(\mathbf{z}) \\
\mathbf{x} &\sim \mathcal{N}(\boldsymbol{\mu}_\mathbf{x}(\mathbf{z}), \sigma^2 \boldsymbol{I})
\end{aligned}$$

i.e. we choose $p_\theta(\mathbf{z})$ to be a standard multivariate isotropic Gaussian distribution, and $p_\theta(\mathbf{x}|\mathbf{z})$ to be a multivariate Gaussian whose mean value $\boldsymbol{\mu}_\mathbf{x}(\mathbf{z})$ is computed by the *decoder* neural network, as a deterministic function of $\mathbf{z}$. If the distribution parameters θ are known, one could in principle also find the marginal distribution using the chain rule of statistics,

$$p_\theta(\mathbf{x}) = \int_\mathcal{Z} p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int_\mathcal{Z} p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z} \tag{2.8}$$

Additionally, if either of the marginal distribution $p_\theta(\mathbf{x})$ or the posterior distribution $p_\theta(\mathbf{z}|\mathbf{x})$ is known, we could compute the other via Bayes' rule

$$p_\theta(\mathbf{z}|\mathbf{x}) p_\theta(\mathbf{x}) = p_\theta(\mathbf{x}|\mathbf{z}) p_\theta(\mathbf{z}).$$

In our setting neither is known, and the integral in equation (2.8) is intractable. We will now describe a method that allows us to

1. Make efficient maximum likelihood estimates of parameters θ. This would allow us to efficiently generate new samples of $\mathbf{x}$ simply by sampling a random vector from a multivariate standard Gaussian and passing them through the decoder network.

2. Learn parameters ϕ of an approximate posterior distribution $q_\phi(\mathbf{z}|\mathbf{x}) \approx p_\theta(\mathbf{z}|\mathbf{x})$, which allows us to encode a sample of $\mathbf{x}$ to find the approximate latent representation $\mathbf{z}$ that is most likely to have generated it.

It might not come as a surprise to the reader that we will use yet another neural network to represent the approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$. To be able to use gradient descent methods to update the posterior parameters ϕ, we need that network to be completely deterministic. To represent a probability distribution using a deterministic neural network, we use the following reparametrization trick:

$$\mathbf{z} = \boldsymbol{\mu}_\mathbf{z} + \boldsymbol{\sigma}_\mathbf{z} \odot \boldsymbol{\epsilon}, \tag{2.9}$$

[1] As we will see in section 3.3.1 this two-step process happens to be exactly how we create our artificial piles. We do not use normal distributions however.

where

$$(\boldsymbol{\mu}_{\mathbf{z}}, \boldsymbol{\sigma}_{\mathbf{z}}) = \text{Encoder}_\phi(\mathbf{x}),$$
$$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}),$$

and $\odot$ is used to denote an element-wise multiplication. If we let equation (2.9) sink in for a while, we see that $\mathbf{z} \sim q_\phi(\mathbf{z}|\mathbf{x})$ is a multivariate Gaussian distribution whose mean $\boldsymbol{\mu}_{\mathbf{z}}$ and (diagonal) covariance matrix $\text{diag}(\boldsymbol{\sigma}_{\mathbf{z}})^2$ is determined by the *encoder* network. The structure of the VAE setup is illustrated in Figure 2.4. Using our data samples $\mathcal{D}$, we want to find the parameters ϕ, θ that maximize the log probability of our observed samples.

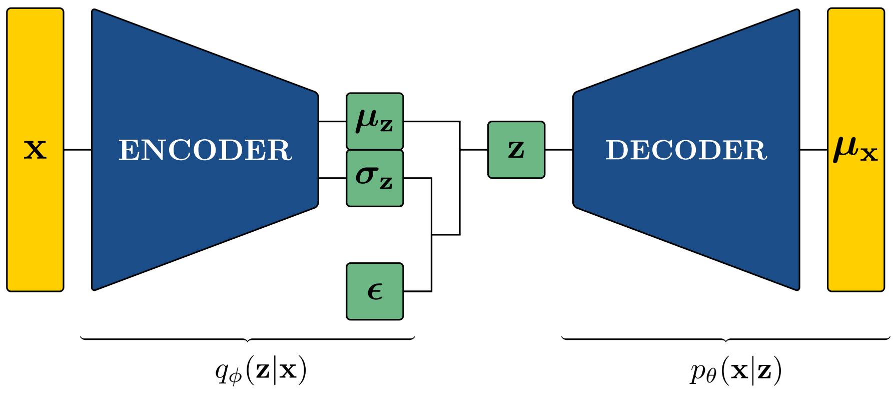
$\mathbf{x}$
ENCODER
$\boldsymbol{\mu}_{\mathbf{z}}$
$\boldsymbol{\sigma}_{\mathbf{z}}$
$\boldsymbol{\epsilon}$
$\mathbf{z}$
DECODER
$\boldsymbol{\mu}_{\mathbf{x}}$
$q_\phi(\mathbf{z}|\mathbf{x})$
$p_\theta(\mathbf{x}|\mathbf{z})$

Figure 2.4: The overall structure of a Variational Autoencoder. During training, a high-dimensional $\mathbf{x}$ is encoded into a low-dimensional representation $\mathbf{z}$ and then decoded back into a high-dimensional $\boldsymbol{\mu}_{\mathbf{x}}$. Note that, without the reparametrization step involving $\boldsymbol{\mu}_{\mathbf{z}}, \boldsymbol{\sigma}_{\mathbf{z}}, \boldsymbol{\epsilon}$, the structure is identical to a regular autoencoder.

2.2.3 The loss function

Suppose we use a regular reconstruction loss when training our VAE. In that case we could imagine that the encoder network would learn $\boldsymbol{\sigma}_{\mathbf{z}} = 0$ and just function as a regular autoencoder from there on. Hence, variational autoencoders require a special kind of loss function to work as intended. That special loss function is usually chosen to be the sum of a reconstruction error and a regularisation error. We will use MSE as our reconstruction loss. The regularization term is to the best of our knowledge always taken to be some form of Kullback-Leibler loss. Let $\ell^{(i)}$ be the loss with respect to the ith sample. Then the loss is given by

$$\ell^{(i)} = \ell^{(i)}_{\text{MSE}} + \ell^{(i)}_{\text{KL}} \tag{2.10}$$

The MSE loss is the same as in an ordinary autoencoder:

$$\ell^{(i)}_{\text{MSE}} = ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_{\mathbf{x}}(\mathbf{z}^{(i)})||_2^2. \tag{2.11}$$

The $\ell^{(i)}_{\text{KL}}$ is taken to be the Kullback-Leibler divergence between the prior $p_\theta(\mathbf{z})$ and the approximate posterior $q_\phi(\mathbf{z}|\mathbf{x})$

$$\ell^{(i)}_{\text{KL}} = D_{\text{KL}}(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})). \tag{2.12}$$

The loss will be large if $q_\phi(\mathbf{z}|\mathbf{x}^{(i)})$ drift away too much from $p_\theta(\mathbf{z})$. Since both distributions are multivariate normal distributions, there exists a closed form solution to equation (2.11)

[18]:

$$D_{\mathrm{KL}}(q(\mathbf{z}|\mathbf{x})||p_\theta(\mathbf{z})) = \frac{1}{2}\sum_{j=1}^{k}\left(\boldsymbol{\sigma}_{\mathbf{z},j}^2 + \boldsymbol{\mu}_{\mathbf{z},j}^2 + \ln(\boldsymbol{\sigma}_{\mathbf{z},j}^2)\right) - \frac{k}{2} \tag{2.13}$$

where k is the dimension of the latent space. Here, we have suppressed the (i) superscripts on $\mathbf{x}$, $\boldsymbol{\mu}_\mathbf{z}$ and $\boldsymbol{\sigma}_\mathbf{z}$ to keep thing more tidy, but note that the mean value and the variance are different for different samples $\mathbf{x}^{(i)}$. The total loss of an entire batch is simply the average of all $\ell^{(i)}$. Sometimes one weights one of the two terms in equation (2.10) by a factor β to emphasise the importance of a structured latent space or accurate reconstructions. In words, the reconstruction loss will promote reconstructions to be as faithful to the input as possible. The regularization term will promote a nicely structured latent space. By optimizing them simultaneously we can make both our generative model and the encoder better at the same time.

2.3 Computational graphs & automatic differentiation

Automatic differentiation (AD) is a family of methods that describe how to compute the derivatives of numerical functions through accumulation of values during code execution. They are methods that generate derivative evaluations rather than derivative expressions. The backpropagation algorithms that are used during training of neural networks are examples of automatic differentiation [19]. We will illustrate the main principles of AD via an example. Consider the following function,

$$f(a, b, x) = \mathrm{ReLU}(ax + b). \tag{2.14}$$

We can visualize the evaluation of this function using a computational graph, see Figure 2.5. This is known as the forward graph. Given a, x and b, we can evaluate the function f by traversing the graph from left to right. Now, suppose we are not only interested in

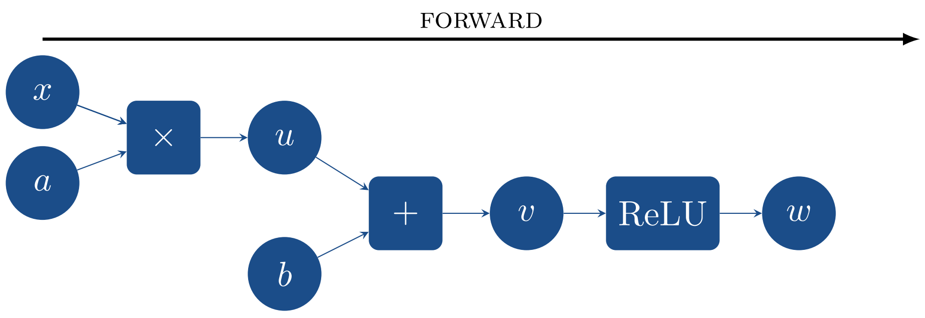

Figure 2.5: Computational graph illustrating the evaluation of the function f in equation (2.14). Computations are carried out from left to right. Circular and rectangular nodes represent variables and operations respectively.

the function value for some triplet a, b, x, but also the derivative evaluated at the same point. Herein lies one of the main strengths of computational graphs. By considering the function to be the combined result of many smaller computations, we can apply the chain rule of calculus to obtain the analytical derivatives at a low computational cost. Let

$$u(s, t) = st, \quad v(s, t) = s + t, \quad w(s) = \mathrm{ReLU}(s). \tag{2.15}$$

Using this notation we can express f as the composite function

$$f(a, b, x) = w(v(u(a, x), b))$$

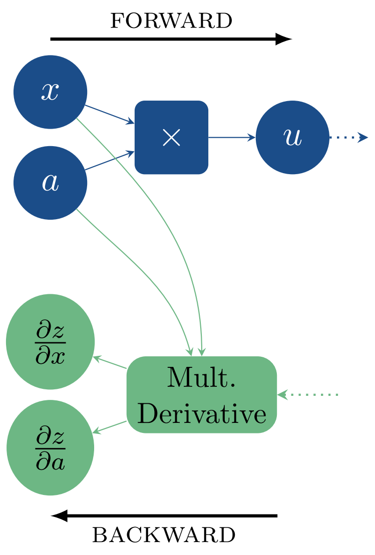
FORWARD
x
a
$\times$
u
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial a}$
Mult. Derivative
BACKWARD

Figure 2.6: Illustration of the first step of the forward pass. When the product $a \cdot x$ is computed, we append a "Mult. Derivative" node to the backward graph and make references to variables x and a.

and the derivative with respect to x will be

$$\frac{\partial f}{\partial x} = \frac{\partial w}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial x}. \tag{2.16}$$

In automatic differentiation methods one sequentially builds a second computational graph in parallel with the function evaluation graph, which describes the partial derivative of each contributing computation. We refer to this graph as the backward graph. Again, the process is most easily explained using an illustration, see Figure 2.6. When the multiplication is carried out we add a "MultDerivative" node to the backward graph and give the node the variables x and a as input. We do not yet know what will precede the MultDerivative, but whatever it is we know that $\frac{\partial u}{\partial x} = a$ and $\frac{\partial u}{\partial a} = x$. For this to work automatically, we must of course have an implemented node type that knows how to compute the derivative of a product. Any AD implementation worth its salt will include derivative operations for at least all elementary functions and matrix operations. To complete the graphs, we then continue to move on down the forward graph, and each time we perform a computation we add a node to the right end of the backward graph. In the end it will look something like the graphs in Figure 2.7. The backward graph is not necessarily evaluated during the forward pass, in fact, in most AD implementations the backward pass is only carried out once the full backward graph is complete. Since $w = z$ we will always use $\frac{\partial z}{\partial w} = 1$ as our starting point. One can then traverse the backward graph and for each derivative node, multiply the current derivative value with the local contribution. To compute the

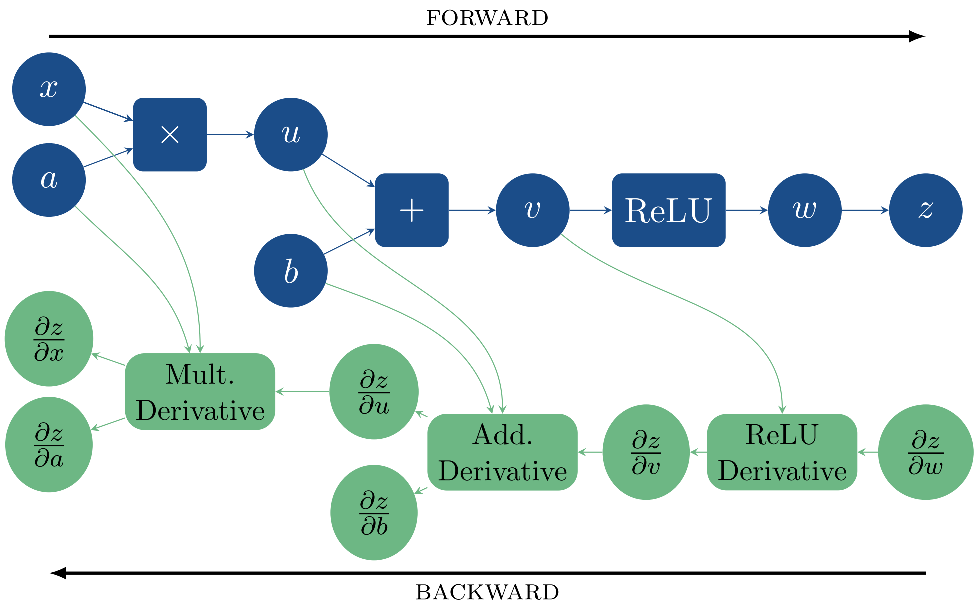
FORWARD
x
a
$\times$
u
$+$
b
v
ReLU
w
z
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial a}$
Mult. Derivative
$\frac{\partial z}{\partial u}$
$\frac{\partial z}{\partial b}$
Add. Derivative
$\frac{\partial z}{\partial v}$
ReLU Derivative
$\frac{\partial z}{\partial w}$
BACKWARD

Figure 2.7: The complete forward and backward graphs after the forward pass is complete.

derivative with respect to x, the computation would look something like this:

$$\begin{aligned}
t_1 &= \frac{\partial z}{\partial w} = 1 \\
t_2 &= \frac{\partial z}{\partial v} = t_1 \frac{\partial w}{\partial v} = \theta(v) \\
t_3 &= \frac{\partial z}{\partial u} = t_2 \frac{\partial v}{\partial u} = \theta(v) \\
\frac{\partial z}{\partial x} &= t_3 \frac{\partial u}{\partial x} = a\theta(v)
\end{aligned}$$

where we have used the Heaviside step function θ in place of the ReLU derivative as described in section 2.1.3. Now suppose we want the derivative with respect to a instead. A benefit with AD is that we do not need to traverse the graph all the way from the right end. Instead, we can use the intermediate value of t_3 and compute

$$\frac{\partial z}{\partial a} = t_3 \frac{\partial u}{\partial a} = x\theta(v).$$

The example shown in Figures 2.5-2.7 was heavily inspired by an example in PyTorch's documentation [20]. It shows a very simplified case where all forward operations have scalar outputs. If one instead considers vector-valued functions, the principle would still be the same but instead of sequentially multiplying scalar derivatives one instead computes Jacobian-vector products.

Chapter 3

Method

3.1 Virtual environment

All experiments are carried out inside a virtual environment (VE) created with AGX Dynamics [21]. VEs and simulations are in many ways a perfect match for deep learning, as they allow us to easily generate large amounts of data at a relatively low cost. In a VE we can also measure essentially any physical quantity that we are interested in. In the following subsections we will describe our digital wheel loader model and how AGX Dynamics can be used to simulate its interactions with the terrain.

3.1.1 AGX Dynamics & AGX Terrain

AGX Dynamics is a multipurpose physics engine that can model complex mechanical systems and interactions between bodies on many different scales. It is centered around rigid multibody systems with joints and contacts at realtime and high accuracy.

AGX Terrain is a library within AGX Dynamics that supports real-time simulation of deformable soil, including interactions between soil and earthmoving equipment such as blades or buckets [22], [23]. Soil consist of small heterogeneous particles of minerals surrounded by water and gas. In bulk, soils have a number of characteristic features that make their dynamics very different from other media. They can transition between behaving like solids and fluids depending on their water contents and what stresses they are subjected to. This transition between solid and fluid is well-described by the Mohr-Coulomb criterion. It predicts that a solid continuum of soil will fail along any plane where the tangent stresses τ and the normal stresses σ satisfy the following equality

$$\tau = \sigma \tan(\phi) + c, \tag{3.1}$$

where ϕ and c are the material's *angle of internal friction* and *cohesion* respectively. The interpretation of the criterion is that as long as the shearing stresses are low, the soil will not display any noticeable deformation. But if the shear stresses reach this critical value the soil will fail and start to flow much like a fluid. Gravel and sand, which are the type of materials our work focuses on, are often considered cohesionless. For such materials, the angle of internal friction corresponds well to the *angle of repose*. Anyone who has ever built a sand castle is likely to have faced problems with the angle of repose, also known as the critical angle of a soil. It describes the steepest possible slope relative to the ground a given granular material can reach before collapsing. Theoretically it can range from 0° to 90° but is typically between 30-45° [24].

AGX Terrain is a hybrid model where resting soil is modeled as a rigid solid, discretized using a voxel representation. A voxel can be completely or partially filled with soil material.

The boundary between the soil and its environment is discretized as a 2D heightfield, see Figure 3.1.

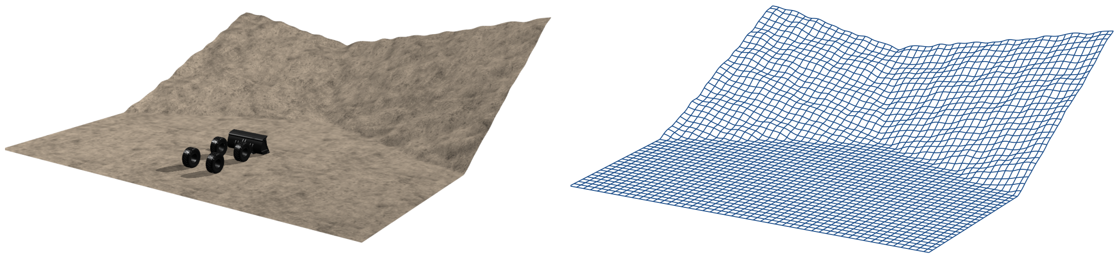
Figure 3.1: (Left) rendered image of the terrain surface in AGX Dynamics. (Right) 2D heightfield representation of the terrain's free surface. The flat parts of the terrain are modelled as solid ground.

When earth-moving equipment such as a bucket comes in contact with the soil, the model predicts the *active zone*, and resolves the 3D grid into particles which are simulated using DEM. Note that these *active particles* are typically still much larger than the real soil grains — the model uses these large particles to represent the macroscopic flow of the finer grains. Collectively, the active particles for an *aggregate body* which is used to model the terrain forces acting on the wheel loader. Whenever active particles outside the bucket stop moving they "blend" into the surrounding terrain and return to being part of the voxel-represented terrain. An illustration of the voxel terrain and its interactions with a bucket is shown in Figure 3.2.

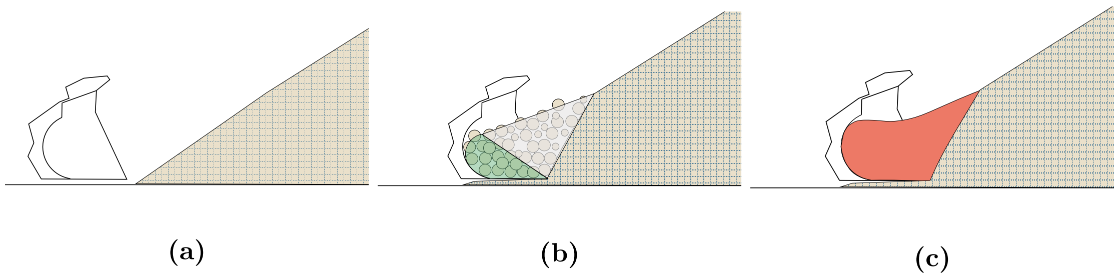

Figure 3.2: Three different representations of soil in AGX Terrain. (a) Resting soil is discretized using a voxel representation. (b) When the bucket comes in contact with the soil, the active zone (grey triangle) is predicted. The green area inside the bucket shows the dead load, which has already been filled with soil. The soil inside the bucket and active zone are represented by pseudo particles (grey circles). (c) Collectively, the pseudo particles form an aggregate body which is used to model forces from the terrain acting on the bucket.

In our work we have set the ground at zero-level to be completely rigid, meaning it cannot be dug into or compressed by the weight of the machine. Everything above the zero level is governed by the AGX Terrain model. This makes it easier to identify which parts of the terrain that are part of the pile and which are not. It also makes it easier to compare different runs from each other when the surrounding conditions are always the same. Currently, we are also only considering one type of soil, the AGX Terrain library material "Gravel 1" which is supposed to emulate dry gravel.

3.1.2 A digital wheel loader

The wheel loader model is the same as the one used in [5] and consists of 27 rigid bodies held together by 23 joints, 5 of which are actuated. It has articulated steering and a driveline provides rotational power to the two wheel pairs via differentials. An illustration of the wheel loader model is shown in Figure 3.3.

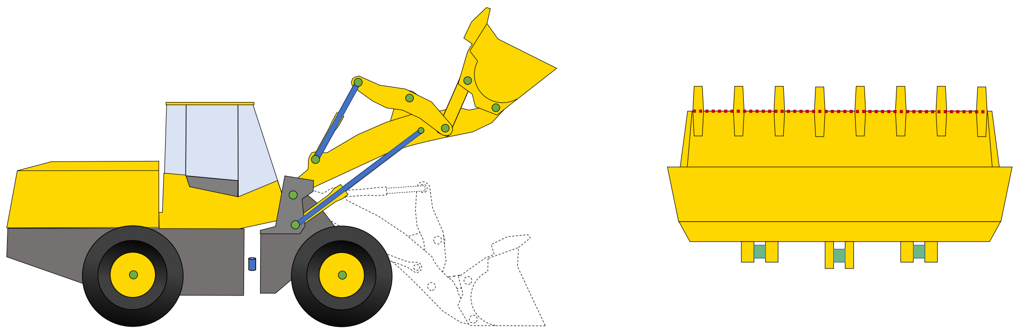

Figure 3.3: Illustration of the modelled wheel loader. Active and passive joints are shown in blue and green respectively. The red dotted line on the right indicates the cutting edge of the bucket.

To fill the bucket the wheel loader has three mechanisms at its disposal: Thrusting the vehicle into the pile using the forward drive, lifting the boom arm, and tilting the bucket. Boom arm lift and bucket tilt is achieved by extending the hydraulic cylinder joints. We model the boom and bucket actuators using prismatic joints and we control their extension $\ell_{\text{bm}}, \ell_{\text{bt}}$ indirectly by giving the actuators target velocities $v_{\text{bm}}, v_{\text{bt}}$. The actuator velocities are then gradually increased or decreased until the target velocities have been reached. Positive velocities correspond to cylinder extensions and a resulting lifting motion of the bucket, see Figure 3.4. It should be noted that assigning a prismatic joint with a velocity does not guarantee that it will actually attain said velocity. This is because the joint's motion could be constrained by forces from other objects in the simulation e.g. the terrain.

3.2 Controlling the wheel loader

The wheel loader is not fully autonomous but can perform autonomous bucket filling. During the loading cycle, the bucket is controlled using a compliance-type controller algorithm developed by Koji Aoshima, PhD student at the research group, as a part of the their current project *Computational Earthmoving*.

The compliance controller provides fully autonomous bucket filling by splitting the loading cycle into five phases: "Forward" → "Dig" → "Post Dig" → "Reverse" → "End". For each phase the controller has a set of rules defined that specify how the wheel loader should act in the current phase and when it should move on to the next one.

3.2.1 Governing equations

The phase of most interest is of course the "Dig" phase. Here the controller uses four action variables $\mathbf{a} = [k_1, k_2, t_1, t_2]^\top$ to control when and how the wheel loader should lift its boom and tilt its bucket.

Let v_{bm} denote the velocity of the boom cylinders' linear actuators, and let v_{bt} be the corresponding velocity for the bucket cylinder. After the bucket's cutting edge has penetrated the terrain surface we set the actuator target velocities to be

$$v_{\mathrm{bm}} = a_1 v_{\mathrm{bm}}^{\mathrm{max}}, \quad v_{\mathrm{bt}} = a_2 v_{\mathrm{bt}}^{\mathrm{max}}$$

where $v_{\mathrm{bm}}^{\mathrm{max}}$ and $v_{\mathrm{bt}}^{\mathrm{max}}$ are the respective maximum velocities of the boom and bucket actuators. The a_is are scaling factors, defined as

$$a_i = \begin{cases} 0, & f_{\mathrm{bm}} < f_1 \\ k_i \left(\frac{f_{\mathrm{bm}}}{f_{\mathrm{ref}}} - t_i \right), & f_1 \leq f_{\mathrm{bm}} < f_2 \\ 1, & f_{\mathrm{bm}} \geq f_2 \end{cases} \qquad i = 1, 2 \tag{3.2}$$

where

$$f_1 = t_i f_{\mathrm{ref}}, \quad f_2 = f_{\mathrm{ref}} \left(\frac{1}{k_i} + t_i \right)$$

and $k_i, t_i, f_{\mathrm{ref}}$ are free parameters. f_{bm} is a moving average of the boom actuator force taken over 10 time steps of size 0.01 s. The boom actuator force is closely related to the resistance forces from the terrain met by the bucket. k_i is the gain coefficient and controls how fast the wheel loader control responds to a change in boom actuator force. t_i is the threshold constant and controls at which point the wheel loader starts responding to the sensed forces. These two parameters are essentially everything we need to fully describe the shape of the response curve, see Figure 3.5. We do however use a third parameter f_{ref} to scale the equation. This helps to keep the other parameter values on the order ~ 1 and could potentially make it easier to transfer the results for our particular machine to other models. In our work, we have chosen $f_{\mathrm{ref}} = 396\,\mathrm{kN}$ and this roughly corresponds to the maximum working hydraulic pressure for a boom actuator in a Komatsu WA320-7 wheel loader [25].

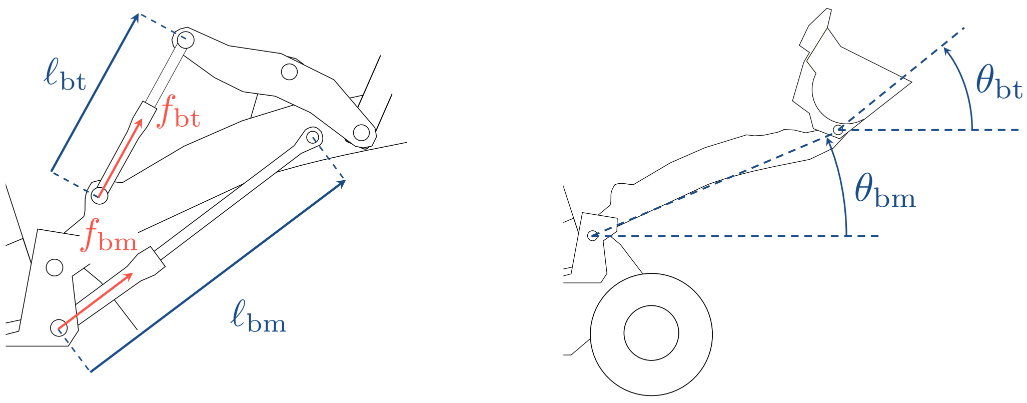

Figure 3.4: Definition of the actuator lengths $\ell_{\mathrm{bm}}, \ell_{\mathrm{bt}}$, forces $f_{\mathrm{bm}}, f_{\mathrm{bt}}$ and the boom and bucket angles $\theta_{\mathrm{bm}}, \theta_{\mathrm{bt}}$.

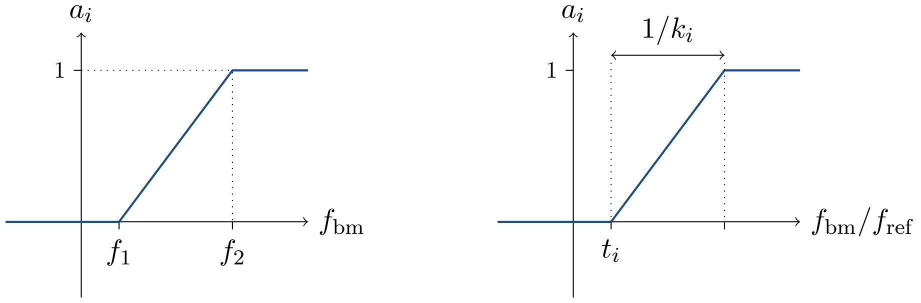

Figure 3.5: The compliance control's force response curves and the interpretation of the tunable parameters t_i and k_i.

3.2.2 Breakout conditions

The part of the dig cycle where the wheel loader bucket emerges from the pile is known as the *breakout*. For any controller to work efficiently it is important that it knows when it is time to stop digging and start retracting from the pile. We considered breakout to have occurred if more than half of the bucket's cutting edge has emerged from the pile surface. The pile surface is not easily defined when the terrain is resolved into fluidized soil and pseudo particles, so we use the initial pile shape surface as reference instead. When the breakout condition has been met, the wheel loader is instructed to tilt its bucket backward at full speed and then start retracting from the pile. If the wheel loader fails to reach breakout within 15 seconds from the time of pile entry, we trigger a timeout condition and tell it to abort the digging. Similarly, if the bucket tilt actuators are fully extended, we do not expect the wheel loader to make much more progress and instruct it to abort the digging.

3.2.3 Control algorithm

The following algorithm describes how we control the wheel loader during a full dig cycle:

```
 1: status ← "Forward"
 2: gear ← "D"
 3: Set wheel loader target speed $v_{\text{wl}} \leftarrow 0.8v_{\text{wl}}^{\max}$
 4: repeat
 5:     if status = "Forward" then
 6:         Throttle as much as traction permits to reach/maintain target speed
 7:         if break_in = True then
 8:             x_poe ← current position
 9:             status ← "Dig"
10:         end if
11:     else if status = "Dig" then
12:         Throttle as much as traction permits to reach/maintain target speed
13:         Measure the boom actuator force $f_{\text{bm}}$
14:         Compute $a_1$ and $a_2$ using equations (3.2)
15:         Set boom target speed $v_{\text{bm}} \leftarrow v_{\text{bm}}^{\max} a_1$
16:         Set bucket target speed $v_{\text{bk}} \leftarrow v_{\text{bk}}^{\max} a_2$
17:         if break_out or is_bucket_tilt_max or time_out then
```

```
            status ← "Post dig"
            t_pd ← 0
        end if
    else if status = "Post-Dig" then
        t_pd ← t_pd + dt
        Set wheel loader target speed $v_{\mathrm{wl}} \leftarrow 0$
        Set boom target speed $v_{\mathrm{bm}} \leftarrow 0$
        if is_bucket_tilt_max = False then
            Set bucket target speed $v_{\mathrm{bt}} \leftarrow v_{\mathrm{bt}}^{\max}$
        else
            Set bucket target speed $v_{\mathrm{bt}} \leftarrow 0$
        end if
        if is_bucket_tilt_max = True and t_pd $\geq 0.5$ then
            status ← "Reverse"
            gear ← "R"
        end if
    else if status = "Reverse" then
        Set wheel loader target speed $v_{\mathrm{wl}} \leftarrow 0.8 v_{\mathrm{wl}}^{\max}$
        Throttle as much as traction permits to reach/maintain target speed.
        if distance from current position to x_poe $\geq 8$ m then
            status ← "End"
        end if
    end if
until status = "End"
```

3.3 Sampling data

The created model is supposed to be data-driven, which means we need to simulate many different dig actions applied to many different pile shapes. For each simulated dig action we store the resulting performance metrics. We later feed these examples to a neural network model to let it learn from experience. In the following two subsections we will describe how we generate our piles and how we sample our action variables.

3.3.1 Artificial piles

To make the predictor model versatile enough we need to expose it to many different pile shapes. We create these artificial piles by using three different basic shapes, later throughout this report referred to as *Simple slopes*, *Eliptic piles*, and *Wedge slopes*. Each shape category has a set of parameters that control steepness, rotation etc. Sampling these parameters randomly allows us to create new piles automatically. Illustrations of these shapes are shown in Figure 3.6. To further vary the pile shapes and make them look more realistic, we add two layers of *simplex noise* on top of the piles. Simplex noise is a gradient based and aperiodic noise and is often used within the computer-graphics field to create realistic-looking terrains at many different scales [26]. Since the model is aimed to be used in sequence during dig planning it is also important that it is not only exposed to smooth, undisturbed piles but also to piles that have been subject to previous dig actions. To accomplish this we use the following algorithm:

```
Require: $G, M \in \mathbb{N}$.
1: Generate $M$ random piles $\{P_i^1\}_{i=1}^M$ by sampling from the three pile shape categories.
2: for $g = 1, \dots, G$ do
3:     for $i = 1, \dots, M$ do
4:         Apply a random dig action to pile $P_i^g$ a from a random direction.
5:         Remove any spillage from the resulting pile state.
6:         Save the new pile state as pile $P_i^{g+1}$
7:     end for
8: end for
```

This will give us a set of $M \times (G+1)$ piles. In our study we have used $G = 4$ The set of piles $\{P_i^g\}_{g=}^{(G+1)}$ will have the same overall shape but will successively be in a more deteriorated state.

3.3.2 Sampling action parameters

The research group have previously made rough estimates of suitable ranges for the four action parameters via grid searches. After some further testing, we decided on using the following ranges

$$0.1 \leq k_1 \leq 3.0, \qquad 0.1 \leq k_2 \leq 3.0, \qquad 0 \leq t_1 \leq 0.5, \qquad 0 \leq t_2 \leq 0.5, \tag{3.3}$$

subject to

$$t_1 + t_2 \leq 0.5. \tag{3.4}$$

Simply put, the t_i parameters tell the wheel loader when to do "something". If they are both set high, we increase the risk of getting stuck during digging because the wheel loader is never given instructions to lift its boom arm or tilt its bucket. The condition in equation (3.4) allows both t_1 and t_2 to attain high values but not at the same time.

Action parameter quadruples were sampled from a uniform random distribution on the ranges described in equations (3.3) and (3.4) and scatter plots of their distribution are shown in Figure 3.7.

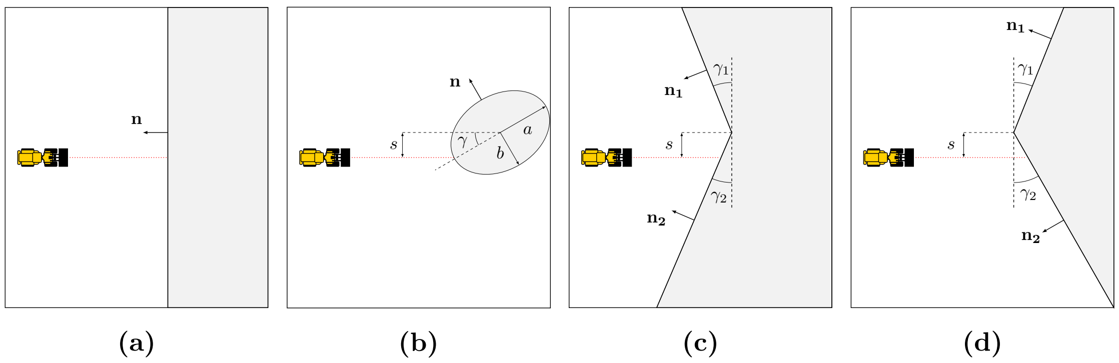

Figure 3.6: The three basic shapes used to generate piles. (a) Simple slope, (b) Elliptic pile, (c) and (d) Wedge slopes. For all shapes, we make sure the slope angle does not exceed the material's angle of repose in the normal directions $\mathbf{n}_i$. New pile shapes can be created automatically by changing the values of parameters a, b, s, γ_i.

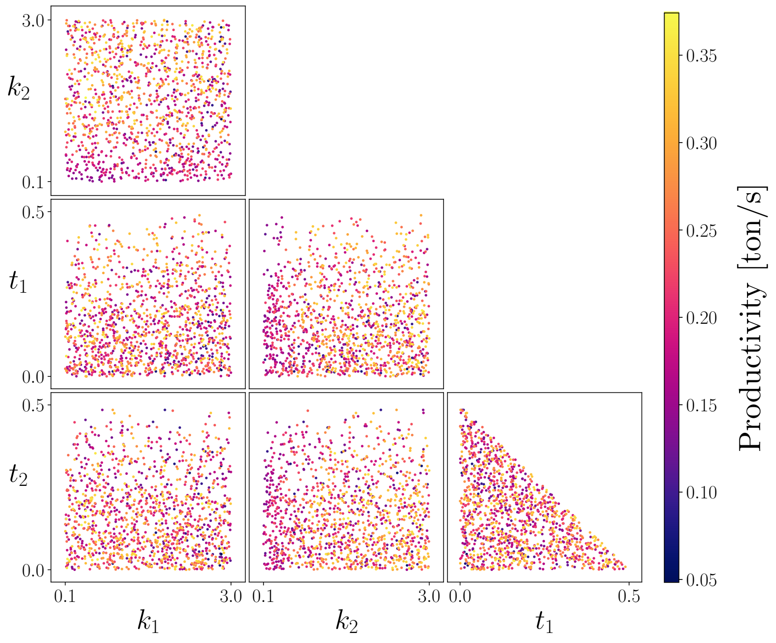

Figure 3.7: Pair-wise scatter plots showing the distribution of 1300 sampled action parameter combinations in the Scalar network dataset. Note that the simulations use different slope angles (not shown here). Note that there seems to be a correlation between productivity and k_2, seen as a higher density of darker dots in the lower end of the k_2 range.

3.3.3 Performance metrics

To evaluate a dig action we need to establish a set of performance metrics. We use the following three fundamental metrics: The loaded mass M, the loading time, T and the work done W. The loaded mass is the total mass M of the material inside the bucket at the end of the dig cycle. This quantity can be obtained directly from AGX Terrain. The distance from the wheel loader's starting point to the edge of the pile is not always the same so to make the different piles comparable to each other we count the loading time from the instance the bucket enters the pile until the wheel loader has completed the digging and backed away 8 meters from the point of entry. The reason we include the reverse phase is again to make comparisons more fair. During the reverse phase, the wheel loader has time to lift the bucket to its final height if it did not already reach it during digging. Since the wheel loader will have to raise its bucket eventually during dumping, this extra work it has done is not for nothing. If we did not include the reverse phase we would penalize dig actions with a high bucket position at breakout. Finally, we also compute the total work W

$$W = W_{\text{engine}} + W_{\text{bm},L} + W_{\text{bm},R} + W_{\text{bt}} \tag{3.5}$$

where the terms on the right hand side represent the work done by the engine, the boom arm actuators and the bucket actuator, respectively.

The engine generates a torque τ. The work done by the engine as it revolves from angle θ_0 to θ_T in time T is given by

$$W_{\text{engine}} = \int_{\theta_0}^{\theta_T} \tau d\theta = \int_0^T \tau\omega dt \approx \sum_{i=1}^{N} \tau_i \omega_i \Delta t, \quad N = \frac{T}{\Delta t} \tag{3.6}$$

where ω is the angular velocity of the engine and Δt is the simulation time step. Both the engine torque and angular velocity can be extracted from AGX Dynamics directly. The expressions for the other three terms in (3.5) are similar:

$$W_{\mathrm{bm},j} = \int_0^T f_{\mathrm{bm},j} v_{\mathrm{bm},j} dt \approx \sum_{i=1}^N f_{\mathrm{bm},j} v_{\mathrm{bm},j} \Delta t, \quad j \in \{L, R\} \tag{3.7}$$

$$W_{\mathrm{bt}} = \int_0^T f_{\mathrm{bt}} v_{\mathrm{bt}} dt \approx \sum_{i=1}^N f_{\mathrm{bt}} v_{\mathrm{bt}} \Delta t. \tag{3.8}$$

From the three fundamental metrics we can additionally obtain the derived metrics *productivity* $\mathcal{P}_{\mathrm{p}}$ and *efficiency* $\mathcal{P}_{\mathrm{e}}$ defined as

$$\mathcal{P}_{\mathrm{p}} = \frac{m}{T}, \quad \mathcal{P}_{\mathrm{e}} = \frac{m}{W}. \tag{3.9}$$

These two metrics are commonly used to evaluate digging and excavation and are motivated by the fact that it can sometimes be worthwhile to spend more time and/or energy as long as the gain in loaded mass is sufficiently large.

3.4 Handling non-determinism and control sensitivity

The fundamental assumption behind our work is that the outcome is determined by the four action parameters and the shape of the terrain. We do however observe that repeating the same simulation can sometimes produce very different results. AGX Dynamics is supposed to be deterministic but small variations can nevertheless occur. It is therefore relevant to investigate how robust the controller is with respect to variations in terrain dynamics. Figure 3.8 shows typical variations in performance and bucket trajectories after repeating the same simulation 200 times. The standard deviation is around 4% of the mean in all performance metrics except for the loading time T, where it is even smaller, around 1%. This motivates making the following amendment to our original model in equation (1.1):

$$\mathbf{y} = f(\mathbf{a}, \mathbf{s}) + \boldsymbol{\varepsilon}$$

where $\boldsymbol{\varepsilon}$ is a random variable following some unknown zero-centered distribution. Instead of aiming for a model that can predict the outcome accurately every single time we will rather aim for a model that can predict the mean value of $\mathcal{P}_j$ if a simulation is repeated many times. This does not affect our approach noticeably. There are however examples that are more troublesome and one such example is shown in Figure 3.9, where we again have repeated the same simulation many times. We observe a bifurcation around $x = 7$ m where about half of the trajectories retract from the pile and the other half keep pushing in deeper. The two groups have wildly different performance, as seen in the scatter plot in Figure 3.9. In this case one could question whether it is meaningful to predict the sample mean value. After running tests on simple slope piles we find that we can filter out most of the scenarios where these huge variations are present by using suitable ranges for the action parameters. It is however difficult to confirm that these suitable ranges are the same for a general pile shape.

3.5 The models

We use three different models to predict the outcome of dig actions and in the following subsections we describe their architectures and how they are trained. The *Scalar regression*

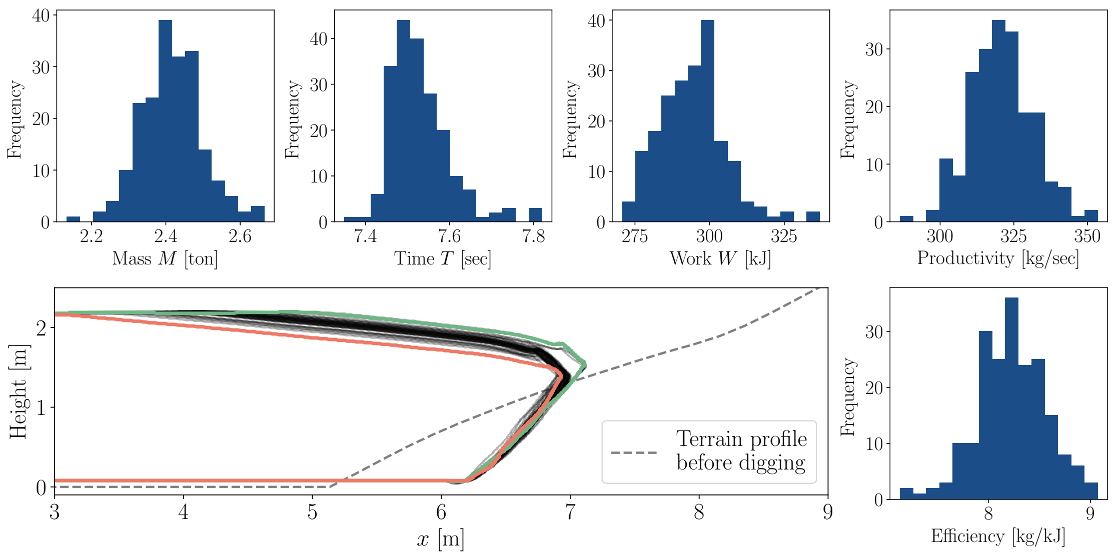

Figure 3.8: Typical variations observed after repeating a simulation many times using the same action variables a and the same pile shape. Histograms show the distribution in performance. The bottom left plot shows a side-view of the bucket tip trajectories during digging. Two of the most extreme cases are highlighted in red and green.

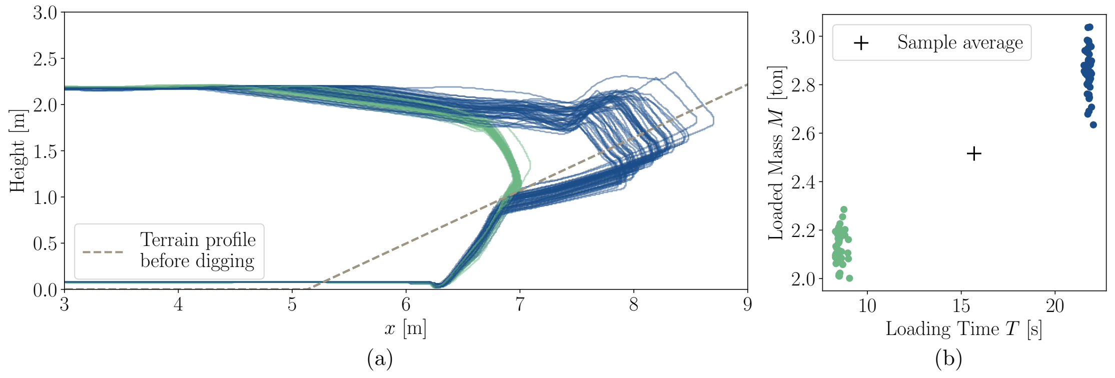

Figure 3.9: An example of a simulation setting with extremely large variations. (a) The samples split up into two distinct groups just below the terrain surface around $x = 7$m. (b) Scatter plot of the the loaded mass M versus the loading time t resulting from the two trajectory groups.

model and the *Convolutional neural network regression model* (CNNR) are used to predict the wheel loader's performance and the *pile state predictor model* is used to predict the altered pile shape following a dig action. All models are implemented using PyTorch [27] and the performance derivatives with respect to actions were computed using PyTorch's Autograd engine [28]. For each model a separate dataset is created and for each dataset we use roughly a 75-10-15 percent split between training, validation, and test data. All model training is done using the Adam optimizer. During training we use *early stopping* and the model with the best performance on the respective validation set is saved for the model testing stage. Unless otherwise stated, we apply min-max scaling to all variables.

3.5.1 The scalar regression model

The scalar regression model is used to predict the performance of a dig action in the simplified case when the pile is a simple slope without any simplex noise added to it. This simplification lets us encode the shape of the pile using only a single parameter θ, the slope angle. Because of the low-dimensional input, this model is very fast to train and does not require as much data as the other two models. This allows us to do a lot of brute force testing before moving on to more complicated setups. The model architecture is shown in Figure 3.10. We train it using the results of 1300 dig actions applied to simple slopes of 13 different angles, using an MSE loss. We train three different versions of this model, each using a different activation function for all hidden layers. This is to investigate what impact the activation function has on the model differentiability. During training, all input and output variables are scaled to range from 0 to 1.

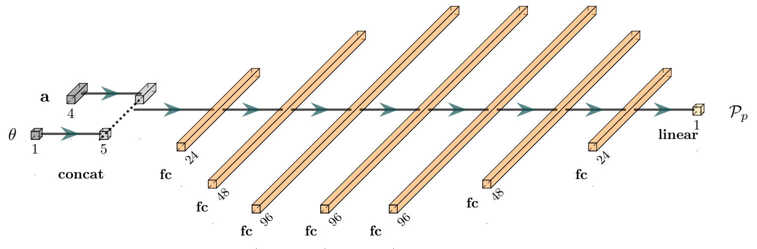

Figure 3.10: The Scalar regression model architecture. Fully connected layers all have a dropout rate of 0.1 during training. Three separate models were trained, each with a different activation function in its fully connected layers. The three activation functions considered were ReLU, Softplus and Swish.

3.5.2 The CNN regression model

The convolutional neural network regression model (CNNR) is meant to take a heightfield representing the local shape of the pile, and use it together with the action variables to predict the resulting performance. The heightfield is fundamentally a 2-dimensional object, and the action variables are fundamentally scalars. Hence, before we let these variables interact we first encode the heightfield into a low dimensional vector $\mathbf{z}$ using three convolutional layers. We then concatenate $\mathbf{z}$ with a vector containing the action variables, and thereafter use a similar architecture as the scalar regression model to compute the resulting performance. The overall network architecture was inspired by [29] and an illustration of it is shown in Figure 3.11. We use ReLU as the activation function in the convolutional layers and softplus in the fully connected layers.

We sample the local heightfields automatically by looking at the terrain directly ahead of the wheel loader prior to digging. We use an algorithm to make an estimate at which point the wheel loader bucket will penetrate the pile surface and sample the heightfield beyond that point as a 3.4 m $\times$ 3.4 m square grid. The side lengths of this grid are not much larger than the bucket width of 2.9 m. Our hypothesis is that only the terrain in the immediate surroundings of the bucket will have a noticeable effect on the dig performance.

The dataset used for this model consists of 2000 unique piles, each subjected to 4 unique dig actions. During training we apply mirroring and small translations to the local

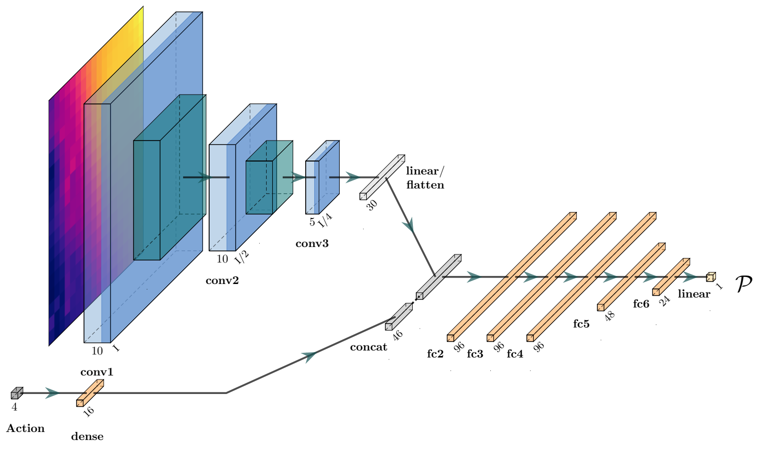

Figure 3.11: The CNN regression model's network architecture. The heightfield is encoded into a lower-dimensional representation before it is combined with the action data. All convolution blocks use ReLU as activation and batch normalization before the nonlinearity. Green blocks represent max pooling layers with a 2-by-2 window. All fully connected layers use Softplus and **fc2-fc6** have a dropout rate of 0.05. The image dimension I is 18 pixels.

heightfields to augment the data and decrease the generalization error. Just as with the scalar model, we use an MSE loss.

3.5.3 The pile state prediction model

The pile state predictor is perhaps the most complicated of our models. It consists of two parts that are trained separately. Our solution is mainly inspired by the work by Saku et al. who used an autoencoder architecture to make short-term predictions of soil deformation due to a bucket [7]. Let $\mathbf{s}_{\text{ini}}$ denote a heightfield representing the initial shape of the pile around the bucket's point of entry. Our goal is to build a model that can predict $\mathbf{s}_{\text{fin}}$, the altered heightfield resulting from a dig action $\mathbf{a}$ applied to $\mathbf{s}_{\text{ini}}$. The overall idea of our solution is to

1. Encode the high-dimensional heightfield $\mathbf{s}_{\text{ini}}$ into a low-dimensional latent representation $\mathbf{z}_{\text{ini}} \in \mathbb{R}^k$.

2. Use the action $\mathbf{a}$ to transform $\mathbf{z}_{\text{ini}}$ into another latent representation $\mathbf{z}_{\text{fin}} \in \mathbb{R}^k$.

3. Decode $\mathbf{z}_{\text{fin}}$ back into the heightfield $\mathbf{s}_{\text{fin}}$.

The first step is to learn to encode and decode our heightfields. For this task we use a VAE. Unlike Saku et al. we want to make long-term predictions of the soil state. This means $\mathbf{s}_{\text{ini}}$ and $\mathbf{s}_{\text{fin}}$ can look very different from each other, meaning $\mathbf{z}_{\text{ini}}$ and $\mathbf{z}_{\text{fin}}$ are also likely to be very different. VAEs are known for having continuous and structured latent spaces, which could make it easier to learn the latent space transformation $\mathbf{z}_{\text{ini}} \to \mathbf{z}_{\text{fin}}$ later on. Our VAE architecture is inspired by the one used by [30] and is shown in Figure 3.12. We train it using 24 000 unique heightfields depicting everything from an undisturbed simple slope to a heavily deteriorated elliptic pile. All heightfields are scaled *individually* to range from 0 to 1. That is, a pile of height 1.5m and a pile of height 2.5m will both be scaled to a maximum height of 1. We find the VAE to be easier to train when it can focus only

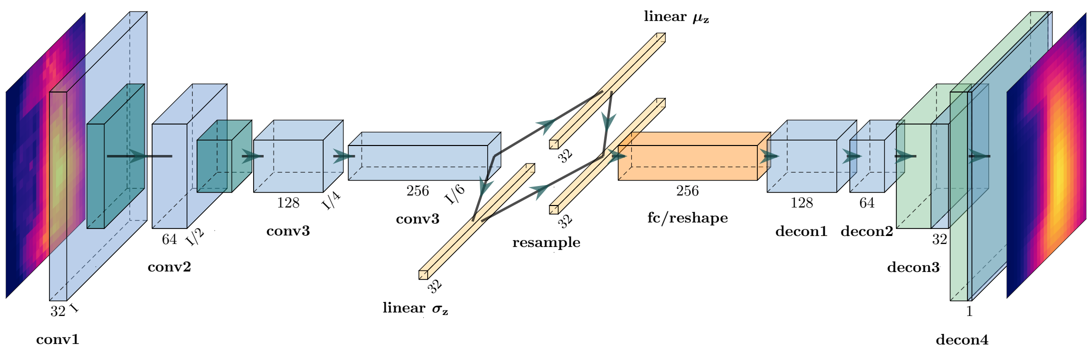

Figure 3.12: Network architecture of the VAE. All convolutional layers except for the last use LeakyReLU as activation function. The last layer uses a sigmoid activation to force the output to be between 0 and 1. Green blocks represent max pooling layers in the encoder half and upsampling layers in the decoder half. The fully connected layer uses LeakyReLU as well. $I = 24$ pixels in this case.

on learning the shape of the piles and without having to care about the scale. Another difference from the CNN model is that we use a larger local heightfield representation, $\mathbf{s}_{\text{ini}}, \mathbf{s}_{\text{fin}} \in \mathbb{R}^{24\times 24}$.

We train the VAE using the loss $\ell = \ell_{\text{MSE}} + 0.1\ell_{\text{KL}}$. For the details of this loss, see section 2.2. The resampling step of the VAE is only used during training. During inference, we use the mean value vector $\mu_{\mathbf{z}}$ directly, as it is the maximum likelihood estimate of $\mathbf{z}$.

Figure 3.13 shows a heatmap of a heightfield and its reconstructed counterpart that has been passed through the VAE. The overall shape is well-preserved but the output is blurry. It is an inherent downside of the "vanilla" VAE as the one we are using that it put high probability on blurry images. To the best of our knowledge it is currently not fully understood why this is. Note that at this point, the VAE can also be used as a generative model to draw new feasible pile shapes, just by drawing samples from a multivariate standard normal distribution and passing them through the decoder. With

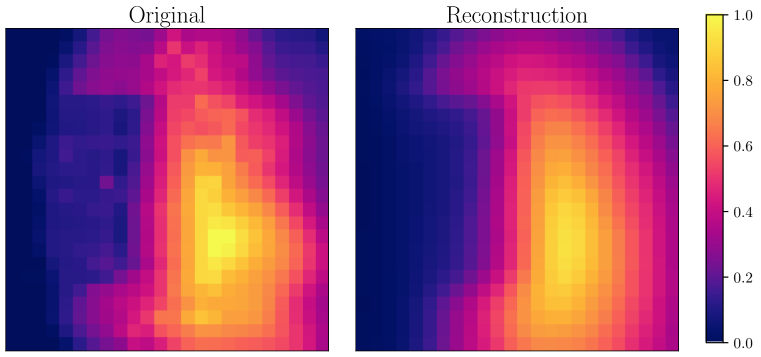

Figure 3.13: Example of a heightfield heat map and its reconstructed counterpart produced by the VAE. The images show a top-down view of the pile. This particular heightfield is an elliptic pile that the wheel loader has dug into (driving from left to right in the image). The VAE's output images tend to be blurry, which corresponds to an overly smooth pile shape. The pixel values have been normalized to range from 0 to 1.

the VAE in place we take 6000 pairs $\{\mathbf{s}_{\text{ini}}^{(i)}, \mathbf{s}_{\text{fin}}^{(i)}\}$ and pass them through the encoder to

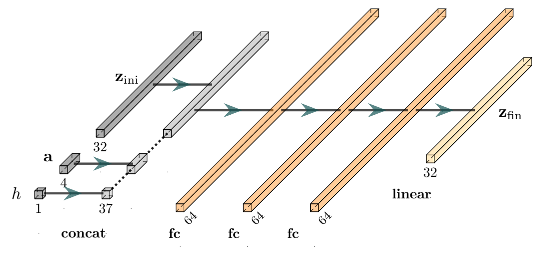

Figure 3.14: The LCTN's architecture. All fully connected layers use a ReLU activation and each has a dropout rate of 0.1.

produce 6000 latent vector pairs $\{\mathbf{z}_{\text{ini}}^{(i)}, \mathbf{z}_{\text{fin}}^{(i)}\}$. We then train a very simple, fully connected neural network to learn the mapping $\mathbf{z}_{\text{ini}} \to \mathbf{z}_{\text{fin}}$. The altered pile shape does of course depend on what the wheel loader does and how high the pile is. We therefore use the action vector **a** and the height h of the pile, as model input. We denote this network as the latent code transformation network (LCTN). The architecture of the model is shown in Figure 3.14.

Finally, we are ready to assemble our pile state predictor model by inserting the LCTN in-between the VAE's encoder and decoder. The full process of predicting a heightfield can be described by the following steps:

$$\begin{aligned}(\mu_{\mathbf{z}}, \sigma_{\mathbf{z}}) &\leftarrow \text{Encoder}(\mathbf{s}_{\text{ini}}),\\ \mathbf{z}_{\text{ini}} &\leftarrow \mu_{\mathbf{z}},\\ \hat{\mathbf{z}}_{\text{fin}} &\leftarrow \text{LCTN}(\mathbf{z}_{\text{ini}}, \mathbf{a}, h),\\ \hat{\mathbf{s}}_{\text{fin}} &\leftarrow \text{Decoder}(\hat{\mathbf{z}}_{\text{fin}}),\end{aligned}$$

where we have used a circumflex to emphasize that the outputs from the LCTN and the Decoder are only estimates.

As previously mentioned, the VAE's decoder tends to produce blurry images. We also note that the pile state predictor model rarely ever makes any useful predictions of the altered pile shape close to the edge of the heightfield. Because of this, we find that we can improve the model further by using some information from the initial pile state to denoise the predicted pile state $\hat{\mathbf{s}}_{\text{fin}}$. To be more precise, we interpolate between $\hat{\mathbf{s}}_{\text{fin}}$ and $\mathbf{s}_{\text{ini}}$ along the heightfield edges and use the shape of the pile-ground boundary in $\mathbf{s}_{\text{ini}}$ to make the prediction more accurate and sharp. The post-processing is illustrated in Figure 3.15 and a comparison of a prediction with the ground truth heightfield is shown in Figure 3.16.

To evaluate the accuracy of our predicted heightfields we use the *volume of the error*, V_{error}. The element-wise absolute difference between the ground truth heightfield $\mathbf{s}_{\text{fin}}$ and our prediction $\hat{\mathbf{s}}_{\text{fin}}$ will form an error surface. If we integrate the volume below this surface using the trapezoidal rule we obtain V_{error}. We can also form a relative error metric by dividing V_{error} by V_{fin}, the total volume of the ground truth final pile shape.

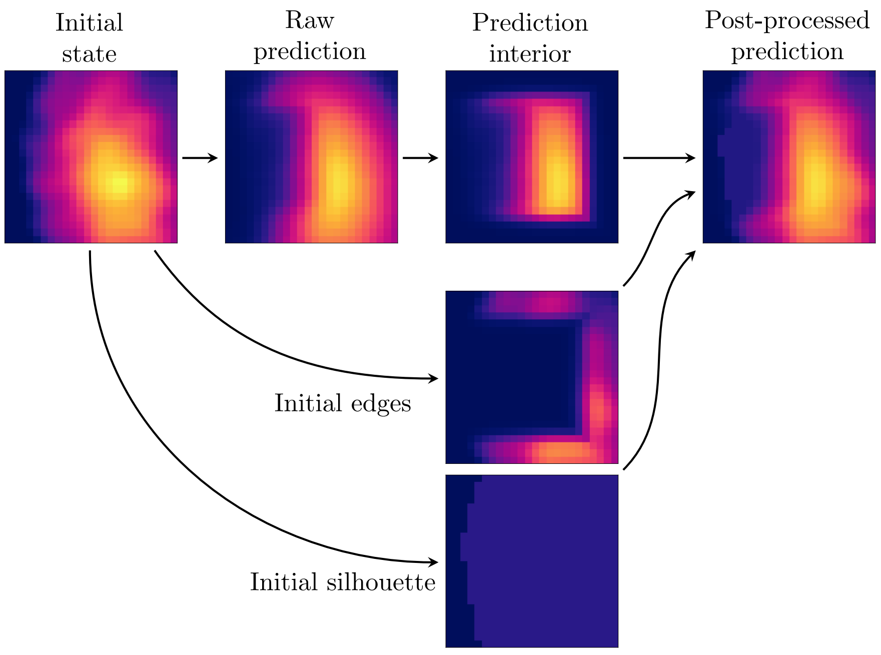

Figure 3.15: Illustration of the heightfield post-processing following the model's prediction.

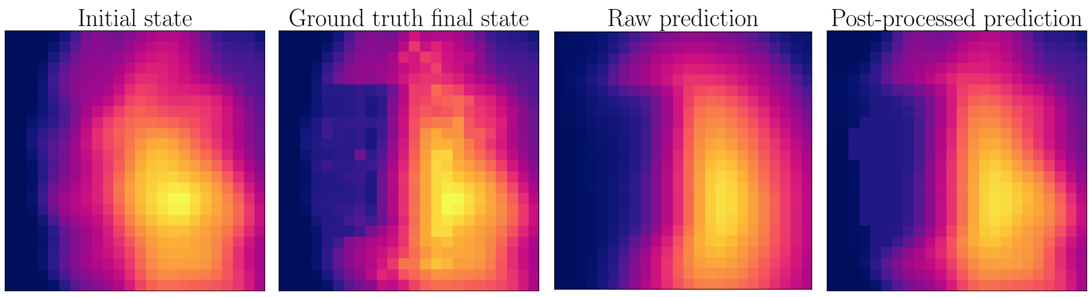

Figure 3.16: Example of a pile state prediction. The two leftmost images show ground truth data of the pile shape before and after a dig action has been applied to it. The “Raw prediction” is the pile shape predicted by the VAE+LCTN model. The rightmost image shows the final prediction after the post-processing step.

Chapter 4

Results

This sections presents the prediction models' performance on test data i.e. data that was neither part of the training or validation datasets during model training. The test datasets were created using the same methods as the training and validation sets.

4.1 Performance prediction models

4.1.1 The scalar model

All three Scalar models reached a similar accuracy on the validation set but the softplus model proved to be the best one and is the one that we have used during testing unless otherwise stated. We tested it on a dataset of 300 unique dig actions applied to 15 perfectly smooth simple slopes with a slope angle ranging from 11° to 39°. The RMSE in predicted productivity compared to ground truth values was 0.014 ton/sec and the average error was 4.5% of the ground truth productivity. Distributions of the errors are shown in Figure 4.1.

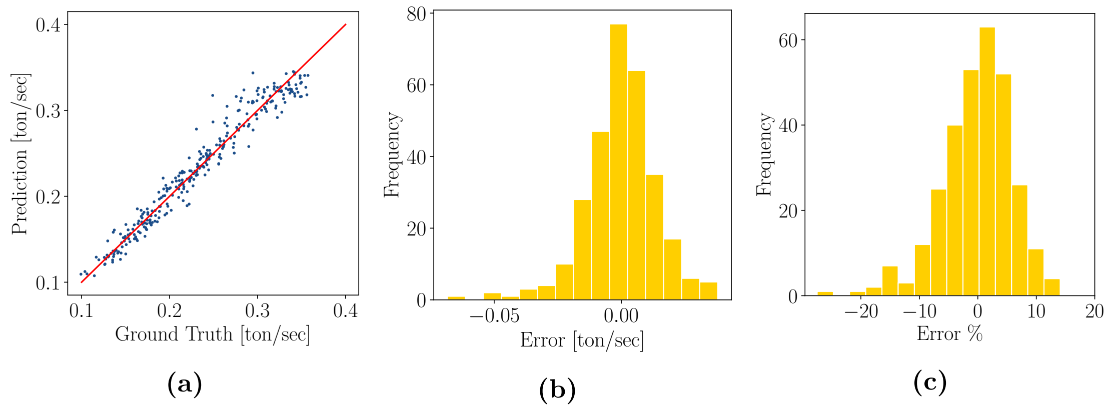

Figure 4.1: Scalar prediction model performance on test data. **(a)** Scatter plot of the predicted productivity vs. the ground truth productivity. **(b)** Distribution of prediction errors measured in [ton/sec]. The RMSE 0.014 ton/sec. **(c)** Distribution of prediction error relative to ground truth value measured in %. Average relative error is 4.5% of ground truth value.

4.1.2 Model derivatives

We use the three scalar models to test if our networks can be treated as differentiable functions of **a** and if so, what impact different activation functions have on the quality of the derivatives. To make it easier to visualize the results, we make our comparison along a straight line running from one edge of the action space to another:

$$\mathbf{a}_\alpha = \mathbf{a}_0 + \alpha(\mathbf{a}_1 - \mathbf{a}_0), \quad \mathbf{a}_0 = \begin{bmatrix} 0.1 \\ 3.0 \\ 0.0 \\ 0.3 \end{bmatrix}, \quad \mathbf{a}_1 = \begin{bmatrix} 3.0 \\ 0.1 \\ 0.3 \\ 0.0 \end{bmatrix}, \quad \alpha \in [0, 1]. \tag{4.1}$$

While keeping the input slope angle fixed at $\theta = 22°$, we evaluate both the function values and the Autograd gradients with respect to the action $\mathbf{a}_\alpha$ along this line. We can obtain the derivative with respect to α using the chain rule,

$$\frac{\partial \mathcal{P}_p}{\partial \alpha} = \frac{\partial \mathcal{P}_p}{\partial \mathbf{a}_\alpha} \frac{\partial \mathbf{a}_\alpha}{\partial \alpha}.$$

We obviously do not have access to ground truth derivatives so as a sanity check we also use the function evaluations to compute finite difference approximations of the derivatives. To verify that the models' predictions are meaningful we also choose 20 equally spaced values of α and for each value, we repeat the same simulation 10 times. The average performance of each group of 10 is computed and we use these averages to benchmark our models. The results are shown in Figure 4.2. We see that all three models are good at predicting the mean productivity and that the models using smooth activation functions produce much smoother derivatives than the ReLU model.

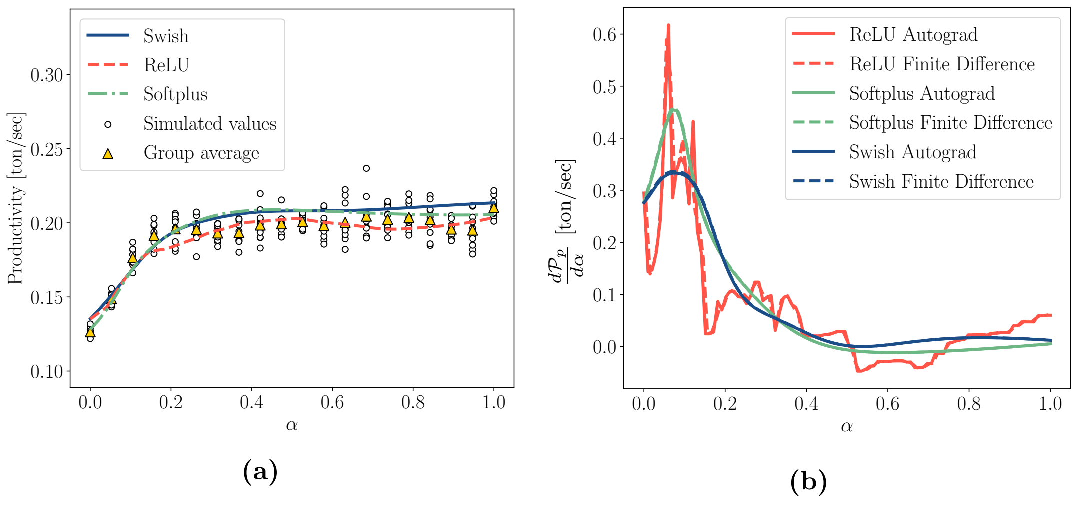

Figure 4.2: Comparison of three models of identical architecture using different hidden layer activation functions. (a) The models are tested by comparing them to the averages of repeated simulations. (b) The derivative of the prediction models with respect to the interpolation parameter. For each of the three models, the Autograd derivatives are compared with a finite difference approximation for reference. The finite differences are computed using a resolution of $\Delta\alpha = 0.01$.

Recall the forward and backward graphs from section 2.3. During model training, PyTorch will always construct both the forward graph to evaluate the target function, and the backward graph to use for parameter updates. During inference one usually do not need the backward graph and the computational load can be reduced by telling PyTorch to ignore it. Since we *do* intend to use the derivatives it is relevant to ask exactly how expensive it is to compute them. We test this by making many function calls to our softplus scalar model with, and without computing derivatives, and measuring the resulting execution time. The tests were carried out on an Intel® Core™ i7-6700 CPU @ 3.40GHz×8 and an NVIDIA GeForce GTX 980 GPU. The results are summarized in Table 4.1.

Table 4.1: Total execution time needed to evaluate the scalar model 100 000 times. To evaluate the network without computing gradients we use `with torch.no_grad()`. When we do want to compute gradients, we use the keyword argument `requires_grad=True` when declaring our tensor of actions, and then use the `torch.autograd.grad()` function to evaluate the gradients after the forward pass. Reported values are sample averages from 100 repeated trials.

	W/ gradients	W/o gradients
CPU	1.5×10^{-1} s	6.5×10^{-2} s
CUDA	1.1×10^{-2} s	3.4×10^{-3} s

4.1.3 The CNNR model

The CNNR productivity prediction model was tested on 100 different pile shapes, each subjected to 10 unique dig actions. The model predicts the productivity with an RMSE of 0.021 ton/sec. The average magnitude of the error relative to ground truth is 7.3%. Plots showing the distribution of error are shown in Figure 4.3.

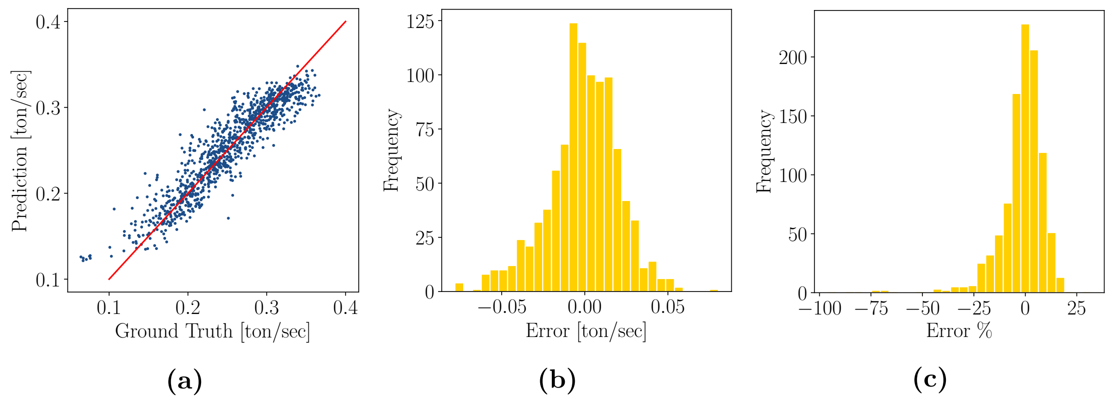

Figure 4.3: CNNR prediction model performance on test data. **(a)** Scatter plot of the predicted productivity vs the ground truth productivity. **(b)** Distribution of prediction errors measured in [ton/sec]. The RMSE error is 0.021 ton/sec. **(c)** Distribution of prediction error relative to ground truth value measured in %. The average magnitude of the relative error is 7.3%.

4.1.4 Model comparison

We compare the softplus scalar model with the CNNR model by having them both make productivity predictions for a perfectly smooth simple slope. Figure 4.4 shows the result of one such comparison along the straight line given by equation (4.1). The results are shown in Figure 4.4. Similar tests were carried out along other lines cutting through action space and for other slope angles and the comparative results were very similar in all cases.

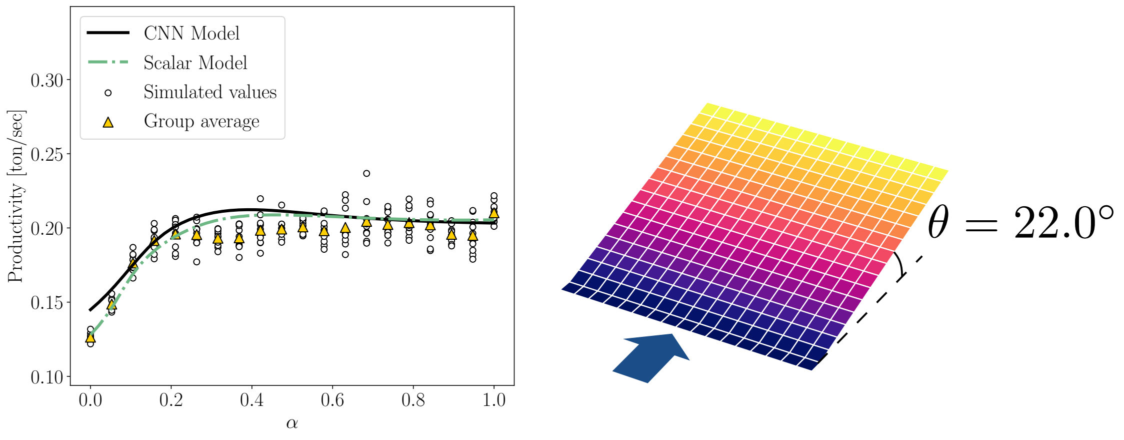

Figure 4.4: Comparison of the softplus scalar model and the CNN model. The two models are given the same pile shape, one in the form of the slope angle and the other in the form of a heightfield. The models are evaluated along the straight line given by equation (4.1). Scattered data shows simulated results along the same line. For each value of α the same simulation is repeated 10 times and the group averages are computed.

4.2 Pile state prediction model

We used the pile state prediction model to predict the altered pile shape of 1000 unique actions applied to 100 unique pile shapes. After computing the errors as described in section 3.5.3 we found the average volume of the error to be $0.79\,\mathrm{m}^3$ and the average relative error to be 4.2%. Examples of predicted pile shapes are shown in Figure 4.5.

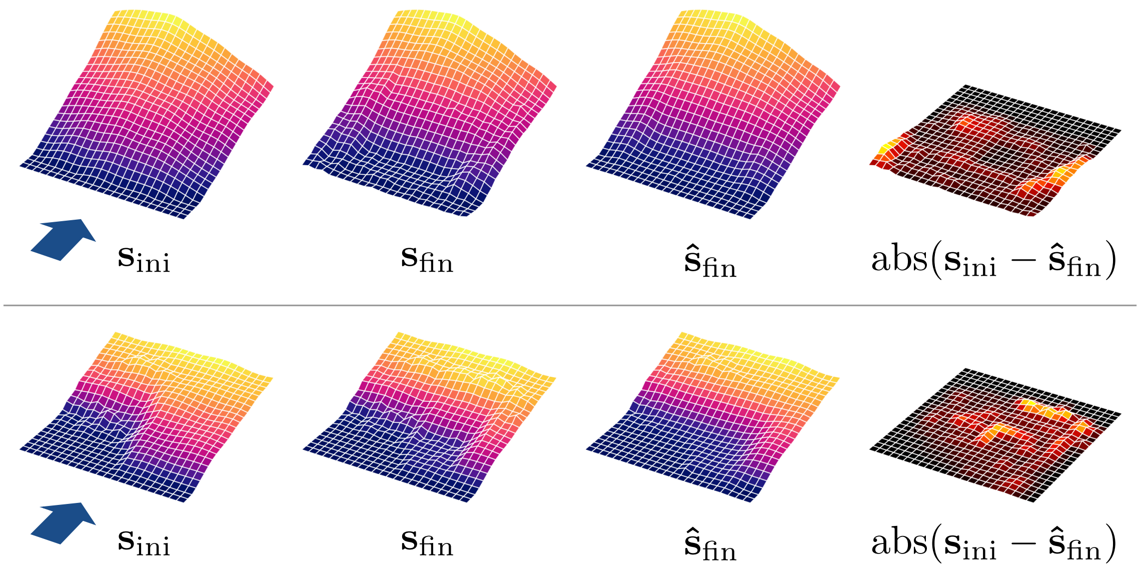
$\mathbf{s}_{\text{ini}}$
$\mathbf{s}_{\text{fin}}$
$\hat{\mathbf{s}}_{\text{fin}}$
$\text{abs}(\mathbf{s}_{\text{ini}} - \hat{\mathbf{s}}_{\text{fin}})$
$\mathbf{s}_{\text{ini}}$
$\mathbf{s}_{\text{fin}}$
$\hat{\mathbf{s}}_{\text{fin}}$
$\text{abs}(\mathbf{s}_{\text{ini}} - \hat{\mathbf{s}}_{\text{fin}})$

Figure 4.5: Two examples of pile state predictions made by our model. The blue arrows show the direction at which the wheel loader drives into the pile. The example in the top row has a volume error of $V_{\text{error}} = 0.96\,\mathrm{m}^3$ and a relative error of 3.8%. The corresponding values for the bottom row example is $0.61\,\mathrm{m}^3$ and 4.7%.

Chapter 5

Discussion

5.1 Performance prediction models

The observant reader may have noticed that none of the figures in section 4.1 showed the models' ability to predict the wheel loader's efficiency. Data was collected for constructing efficiency models as well but due to time limitations we had to prioritize and decided to consider only one of the two metrics. The loading time T was strongly correlated to the bucket tilt speed k_2 so predicting the productivity $\mathcal{P}_\mathrm{p} = m/T$ seemed like the simpler task of the two.

The scalar prediction model may appear to have very limited use and that is also the case. We decided to include it in our results anyway as it serves as good reference for what we can hope to expect from the full model if we could encode the shape of the pile perfectly and had access to an order of magnitude more data per unique encoding. Speaking of more data, we do *not* believe that is the main reason that the CNNR is struggling compared to the scalar model. We increased the size of the CNNR training set by 25% at one point during training but it only gave us a slight improvement in performance. The model architecture is far from optimized and improvements could surely be made on that front. When studying the cases where the CNNR model misses by a lot, it is often in scenarios where the wheel loader gets slightly stuck or aborts its digging too early. We believe that much could be gained by improving the breakout condition in the wheel loader's control.

As long as smooth activation functions were used, the predicted productivity surface was smooth along all straight lines we followed through the action space. The same thing was true for the models' gradients with respect to action. The ReLU gradients looked noisy in comparison but after all, they are obviously of high enough quality to optimize the model weights so maybe one should not reject them too quickly. The reported execution time obviously depend on the model architecture but it appears that computing the gradients of the network increases the execution times by a factor of 2-3. To compute finite differences of a function $f : \mathbb{R}^4 \to \mathbb{R}$ one would need at least five function evaluations so in comparison the Autograd derivatives might be the more efficient choice.

5.2 Pile state predictions

We believe that the pile state prediction model shows a lot of promise. When going through the test dataset we see that it has learned to make predictions for a variety of different initial pile shapes, and that it definitely uses the height input h to make its predictions. More specifically, it predicts that the wheel loader will dig deeper into a shallow pile than it would in a steep slope. It should be noted that the average volume error of $0.79\,\mathrm{m}^3$ does *not* mean that we lose or add almost a cubic meter of gravel with each prediction. One should rather see it as there being, on average, $0.79\,\mathrm{m}^3$ void or gravel that is misplaced on the predicted heightfield. This is still not a good thing of course, but it is an important distinction to make.

When we look at the entire set of predicted heightfields, we see that our model on average predicts a pile that is $0.15\,\mathrm{m}^3$ smaller than the ground truth. This means that the model over-estimates the amount of gravel that is removed by the wheel loader. This could be related to another issue we observe: the model rarely ever predicts a noticeable *gain* in height anywhere on the pile. In simulation, we often observe a rim of excess material around the excavated parts of the pile. Apart from this, the model appears to struggle the most in three cases: *i*) In the very steepest piles, *ii*) When the wheel loader attacks a pile at an angle instead of head on, and finally *iii*) When the controller has aborted the dig action unusually early. The first two of these issues should be possible to solve by simply increasing the frequency of these specific cases in the training data. The third problem could potentially be solved by, again, improving the controller's breakout condition.

Putting these worst-case scenarios aside, what is the source of error in the pile state predictions? The LCTN network was very easy to train to reach high accuracy on the validation data. This leads us to believe that the typical errors in pile shape predictions are mostly due to the VAE's blurry decodings. As mentioned in section 2.2, vanilla VAEs tend to produce blurry output, but the amount of blur can of course be reduced. In recent years, more sophisticated VAE architectures have been developed that can generate photo realistic images by using higher level features to supplement the low-level latent representation [31].

This study has only considered one type of soil. In future work, it would be interesting to see if we could get the models to take the material properties into account and adjust their predictions accordingly. It would also be interesting to see if we could use the loaded mass predicted by a CNNR model to improve the conservation of mass issue we have with the pile state prediction model.

Chapter 6

Conclusion

In this master thesis we have shown that convolutional neural networks can be used to predict the productivity of an autonomous dig action applied to an arbitrarily shaped pile of gravel with an average error of 7.3%. We have also shown how a variational autoencoder can be used to make predictions about the altered shape of a pile following such a dig action. Finally, we have also shown that the automatic differentiation engine that is used during network training can be used to produce high-quality gradients of the network with respect to the model input. More work is needed before we can say for certain that models such as ours can be used to optimize dig plans over many actions in sequence.

Bibliography

[1] C. Borngrund, F. Sandin, and U. Bodin, "Deep-learning-based vision for earth-moving automation," *Automation in Construction*, vol. 133, p. 104 013, 2022, ISSN: 0926-5805. DOI: `https://doi.org/10.1016/j.autcon.2021.104013`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0926580521004647`.

[2] R. Filla, M. Obermayr, and B. Frank, "A study to compare trajectory generation algorithms for automatic bucket filling in wheel loaders," in *3rd Commercial Vehicle Technology Symposium*, 2014, pp. 588–605.

[3] R. Filla and B. Frank, "Towards finding the optimal bucket filling strategy through simulation," in *Proceedings of 15: th Scandinavian International Conference on Fluid Power, June 7-9, 2017, Linköping, Sweden*, Linköping University Electronic Press, 2017, pp. 402–417.

[4] S. Dadhich, U. Bodin, and U. Andersson, "Key challenges in automation of earth-moving machines," *Automation in Construction*, vol. 68, pp. 212–222, 2016, ISSN: 0926-5805. DOI: `https://doi.org/10.1016/j.autcon.2016.05.009`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0926580516300899`.

[5] K. Aoshima, M. Servin, and E. Wadbro, "Simulation-based optimization of high-performance wheel loading," *arXiv preprint arXiv:2107.14615*, 2021.

[6] D. M. Lindmark and M. Servin, "Computational exploration of robotic rock loading," *Robotics and Autonomous Systems*, vol. 106, pp. 117–129, 2018.

[7] Y. Saku, M. Aizawa, T. Ooi, and G. Ishigami, "Spatio-temporal prediction of soil deformation in bucket excavation using machine learning," *Advanced Robotics*, pp. 1–14, 2021.

[8] H. Montes-Campos, J. Carrete, S. Bichelmaier, L. M. Varela, and G. K. Madsen, "A differentiable neural-network force field for ionic liquids," *Journal of chemical information and modeling*, 2021.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[10] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989, ISSN: 0893-6080. DOI: `https://doi.org/10.1016/0893-6080(89)90020-8`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

[11] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Networks*, vol. 6, no. 6, pp. 861–867, 1993, ISSN: 0893-6080. DOI: `https://doi.org/10.1016/S0893-6080(05)80131-5`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0893608005801315`.

[12] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, *On the number of linear regions of deep neural networks*, 2014. DOI: `10.48550/ARXIV.1402.1869`. [Online]. Available: `https://arxiv.org/abs/1402.1869`.

[13] D. Bertoin, J. Bolte, S. Gerchinovitz, and E. Pauwels, "Numerical influence of relu'(0) on backpropagation," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[14] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[15] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2013. DOI: `10.48550/ARXIV.1312.6114`. [Online]. Available: `https://arxiv.org/abs/1312.6114`.

[16] ——, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. DOI: `10.1561/2200000056`. [Online]. Available: `https://doi.org/10.1561%2F2200000056`.

[17] F.-F. Li, J. Johnson, and S. Yeung, Lecture notes, May 2019. [Online]. Available: `http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture11.pdf`.

[18] J. Duchi, *Derivations for linear algebra and optimization.* [Online]. Available: `https://web.stanford.edu/~jduchi/projects/general_notes.pdf`.

[19] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of machine learning research*, vol. 18, 2018.

[20] *Overview of pytorch autograd engine*, Jun. 2021. [Online]. Available: `https://pytorch.org/blog/overview-of-pytorch-autograd-engine/`.

[21] A. Simulations, *Agx dynamics*, `https://www.algoryx.se/agx-dynamics/`, May 2022.

[22] M. Servin, T. Berglund, and S. Nystedt, "A multiscale model of terrain dynamics for real-time earthmoving simulation," *Advanced Modeling and Simulation in Engineering Sciences*, vol. 8, no. 1, p. 11, 2021. [Online]. Available: `https://doi.org/10.1186/s40323-021-00196-3`.

[23] T. Berglund and M. Servin, "Agxterrain," Algoryx Simulation AB, Tech. Rep., Mar. 2020, `https://www.algoryx.se/download/agxTerrain_tech_report.pdf`.

[24] A. Verruijt, *An Introduction to Soil Mechanics.* Springer International Publishing AG, 2018.

[25] K. Europe, *Wa320-7 product brochure*, `https://www.komatsu.eu/Assets/GetBrochureByProductName.aspx?id=WA320-7&langID=en.`, May 2022.

[26] K. Perlin, *Noise hardware*, Lecture notes. [Online]. Available: `https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf`.

[27] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[28] A. Paszke, S. Gross, S. Chintala, *et al.*, "Automatic differentiation in pytorch," 2017.

[29] T. Osa and M. Aizawa, "Deep reinforcement learning with adversarial training for automated excavation using depth images," *IEEE Access*, vol. 10, pp. 4523–4535, 2022. DOI: `10.1109/ACCESS.2022.3140781`.

[30] N. Kuchur, *Landscape generator (github repository)*, Oct. 2021. [Online]. Available: `https://github.com/nikitakuchur/landscape-generator/blob/main/README.md`.

[31] A. Razavi, A. v. d. Oord, and O. Vinyals, *Generating diverse high-fidelity images with vq-vae-2*, 2019. DOI: `10.48550/ARXIV.1906.00446`. [Online]. Available: `https://arxiv.org/abs/1906.00446`.