Machine learning and computer vision for urban drainage inspections
Meijer, D.W.J.

Citation
Meijer, D. W. J. (2023, November 7). *Machine learning and computer vision for urban drainage inspections*. Retrieved from https://hdl.handle.net/1887/3656056

Version:	Publisher's Version
License:	Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden
Downloaded from:	https://hdl.handle.net/1887/3656056

Note: To cite this publication please use the final published version (if applicable).

2 Preliminaries

This chapter covers preliminary knowledge that is required to understand the main content of the thesis. Depending on the reader's proclivities, this chapter may be skipped in its entirety, or referred back to when needed.

2.1 Machine Learning

Machine learning can be broadly summarised as using statistical methods to extract *knowledge* from *data*. The general case considers data which is comprised of *instances*, each containing the same *attributes*, one of which may be the *target* attribute. The goal is then to find some pattern in the non-target attributes that can predict the target attribute as best as possible, meaning that the prediction error across all instances is limited in some way. Machine learning can be supervised, unsupervised, or semi-supervised. This distinction indicates whether the value of the target attribute is known for all, none, or some of the instances, respectively.

This section will go over several use cases, methods, and concepts relevant to the remainder of the thesis and may be skipped or referred back to at the reader's discretion.

2.1.1 Classification

Supervised classification (henceforth simply classification) is a machine learning task where the goal is to infer a relationship between instances and target 'labels' by generalizing from a training set, a collection of such instances for which the label is known. This generalization can then be used to classify objects for which the label is not known.

We consider a dataset $\mathbf{X}$ consisting of N real-valued vec-

tor instances of equal length:

$$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \tag{2.1}$$

$$\mathbf{x}_i \in \mathbb{R}^d \qquad \forall i \in [1, N] \tag{2.2}$$

$$\mathbf{x}_i = \left\{x_{i,1}, x_{i,2}, \ldots, x_{i,d}\right\} \qquad \forall i \in [1, N] \tag{2.3}$$

Each of these vectors also has an associated target, y_i, belonging to one of m distinct classes:

$$\mathbf{Y} = \left\{y_1, y_2, \ldots, y_N\right\} \tag{2.4}$$

$$y_i \in \{c_1, c_2, \ldots, c_m\} \qquad \forall i \in [1, N] \tag{2.5}$$

The goal of classification is then to find the function F that relates the vectors to their label:

$$F(\mathbf{x}_i) = y_i \qquad \forall i \in [1, N] \tag{2.6}$$

We can state that $F(\cdot)$ should be a function between the two domains:

$$F : \mathbb{R}^d \mapsto \{c_1, c_2, \ldots, c_m\} \tag{2.7}$$

In reality, we will have a *model*, $f(\cdot)$, which does not return the real label, but rather an estimation, $f(\mathbf{x}) = \hat{y}$. Finding and improving this $f(\cdot)$ is done through a *loss function L*, a function that is minimised when $y = f(\mathbf{x})$.

The way in which L is minimised depends on the choice of model. For some combinations of loss function and model there may exist an analytical solution that minimises L. For most combinations however, this will be a heuristic process without any analytical solution.

2.1.2 Regression

Regression is a learning problem very similar to classification, but one where the target attribute is on a spectrum. We are still dealing with a dataset $\mathbf{X}$ consisting of N real-valued

vector instances of equal length and associated targets $\mathbf{Y}$. However, instead of the target being a discrete label, y_i is now a real-valued scalar:

$$y_i \in \mathbb{R} \qquad \forall i \in [1, N] \tag{2.8}$$

$$F : \mathbb{R}^d \mapsto \mathbb{R} \tag{2.9}$$

The most common form that the model $f(\cdot)$ will take is a *linear regression model*:

$$f(\mathbf{x}_i) = \beta_0 + \sum_{k=1}^{d} \beta_k x_{i,k} \tag{2.10}$$

$$\beta = \{\beta_0, \ldots, \beta_d\} \tag{2.11}$$

Where the collection of parameters β define the model.

The most common choice of loss function for a linear regression model is the Euclidean distance, known also as the squared error:

$$L = \left\| y - f(\mathbf{x}) \right\|^2 \tag{2.12}$$

With this combination of model and loss function, the values of β that minimise the squared error as defined in equation (2.12) for given values of $\mathbf{X}$ and $\mathbf{Y}$ can be found analytically through the *ordinary least squares* method.[1]

[1] GOLDBERGER, A. S. 1964. *Classical Linear Regression, Econometric Theory*. New York: John Wiley & Sons

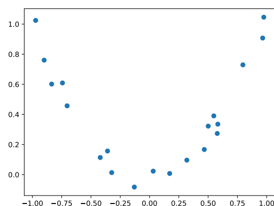
2.1.3 Overfitting, Regularization, Cross Validation

When performing classification or regression, we assume that the N vectors in $\mathbf{X}$ are sufficient for a model $f(\cdot)$ to estimate the targets $\mathbf{Y}$. Often, this is not the case, and the model $f(\cdot)$ may perform well on the training set, without generalizing well to new data from the same distribution. This effect is called *overfitting*, as the model is fitted to the training data so well that it hinders generalization performance.

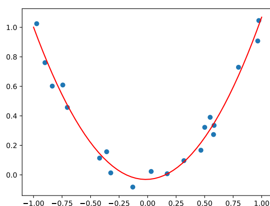
To further appreciate the risks of overfitting, consider a polynomial regression model, fit onto one-dimensional data $\mathbf{X}$ with target $\mathbf{Y}$. We define the model as:

$$f(x_i) = \sum_{k=0}^{\kappa} \beta_k \cdot (x_i)^k \qquad (2.13)$$

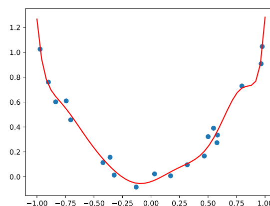
The value of κ is called the order of the polynomial, and is a direct measure of the complexity of the model, as the model has $\kappa + 1$ parameters. As an example, we create an artificial dataset by sampling $N = 20$ datapoints on a parabola and adding uniform noise to the target attribute, as shown in figure 2.1(a). Then compare the difference between a fit of a polynomial of order $\kappa = 2$ in figure 2.1(b), and a fit of a polynomial of order $\kappa = 10$ in figure 2.1(c).



(a) Scatterplot of artificial data



(b) Polynomial fit of order $\kappa = 2$



(c) Polynomial fit of order $\kappa = 10$

Figure 2.1: Example of overfitting in one-dimensional regression

The more complex model with $\kappa = 10$ has a smaller error than the model with $\kappa = 2$, but an important distinction is that the smaller error was achieved on the data that the model was fit on, but not necessarily on a future sample from the same distribution (a parabola with uniform noise). The model with $\kappa = 10$ has in fact *overfitted* on the training data: the fit is so precisely tailored to this sample, that it will generalise less well than the $\kappa = 2$ model to other samples from the same distribution. One approach to prevent overfitting is to simply limit the complexity of the model we use. A good rule of thumb is that the number of model parame-

ters should be at least one order of magnitude smaller than the sample size, in this case e.g. $\kappa = 2$.

REGULARIZATION is a different approach to prevent overfitting: instead of limiting our model complexity, we solve a proxy problem that forces constraints on the solution. A regularization term is added to the loss function, which grows larger in size for more complex models. The new loss function is defined as:

$$L = L_p(\mathbf{x}, y) + \lambda L_r(f) \tag{2.14}$$

Where L_p is the *problem loss*, L_r is the *regularization loss*, and λ is a scaling factor.

The problem loss is the loss function that was described in section 2.1.1, which when minimised gives the optimal fit. The regularization loss is a value that scales with the complexity of the model. The scaling factor λ is used to weight the importance of the regularization.

The rationale is that a compromise has to be made between a model that fits the training data perfectly, and the complexity of that model. In our one-dimensional regression example, we could determine the complexity of the model by taking $L_r = \sum_{k=0}^{\kappa} ||\beta_k||$, known as L_1 regularization, which prefers that only some β_k have non-zero values.

CROSS VALIDATION does not prevent overfitting, but does give us an indication of whether overfitting is happening with our model. To validate a model, it is important to assess the performance on data that was not part of the training set. A common option is to split the available data into a training set and a testing set, using the first part to fit the model and the second to test it.

This method of splitting data becomes problematic when the amount of available data is small, as any data point not in the training set will make the fit worse, and any data point

not in the test set will make the estimated performance less reliable.

Cross validation offers a middle ground by having a small test set and a large training set, but repeating the process to better estimate the performance. The data set is divided into k non-overlapping *folds*. One fold is used as the test set and the remaining folds are concatenated into the training set. This process is repeated k times, each time using a different fold as the test set, such that every item in the data set has been tested once. $k = 10$ is a customary value, which we call '10-fold cross validation'.

2.1.4 Model Validation

Perhaps just as important as obtaining a trained model, is knowing the model's boundaries. In the context of urban drainage inspection, if a classifier predicts some defect, it is important to know how trustworthy this result is to put it into context before acting on it. To properly assess the quality of our models, we examine several different performance metrics to discuss their value.

Commonly, the classification *accuracy* is used as a performance metric, which is the ratio of correctly classified samples out of all samples. It can be observed that this is not a very useful measure for extremely imbalanced datasets, or use cases where different types of errors do not carry the same cost. For an extremely imbalanced dataset, we could create a classifier that classifies every datum as the majority class, and it would have a high accuracy. When different types of errors carry different costs, we may want to use a metric that is aware of these costs, which accuracy is not.

Table 2.1 shows a *confusion matrix* for a binary classification problem, which illustrates the types of errors we can make when misclassifying instances. True positives (TP) and true negatives (TN) are correct classifications, false negatives (FN, sometimes called Type II Errors) and false pos-

Table 2.1: Confusion matrix for a binary classification scenario

Actual \ Predicted	Defect	No Defect
Defect	True Positive	False Negative (Type II error)
No Defect	False Positive (Type I error)	True Negative

itives (FP, sometimes called Type I Errors) are misclassifications. We might not always want these types of errors to count equally in our performance assessment.

We define the false positive rate (FPR), true negative rate (TPR or *specificity*), false negative rate (FNR), and true positive rate (TPR, or *recall*) by dividing a quadrant in the confusion matrix with the row total:

$$FPR = \frac{FP}{FP + TN} = 1 - TNR \tag{2.15}$$

$$TNR = \frac{TN}{FP + TN} = 1 - FPR \tag{2.16}$$

$$FNR = \frac{FN}{FN + TP} = 1 - TPR \tag{2.17}$$

$$TPR = \frac{TP}{FN + TP} = 1 - FNR \tag{2.18}$$

These rates hold some significance as they are equivalent to the chances that a classifier will make a certain error. For example, if $FNR = 0.2$, this means in practice that 20% of actual defects are not recognised as defects by the classifier.

A binary probabilistic classifier may output real-valued predictions in the interval $[0, 1]$, while the actual labels are always either 0 or 1. This means we have some freedom in choosing a threshold τ, that separates a predicted 0 from a predicted 1. We write this as:

$$\hat{y} = \begin{cases} 0 & \text{if} f(x) < \tau \\ 1 & \text{if} f(x) \geq \tau \end{cases} \tag{2.19}$$

where $\hat{y}$ is the predicted label and $f(x)$ is the classifier's real-valued output for instance x. Setting a specific threshold τ for the classifier's output results in each classified instance being either a true positive ($y = \hat{y} = 1$), a true negative ($y = \hat{y} = 0$), a false positive ($y = 0; \hat{y} = 1$), or a false negative ($y = 1; \hat{y} = 0$).

We have some freedom on how to choose τ, which gives us a way to balance the false negatives and false positives. Any increase in τ leads to an increase in *FNR* and a decrease in *FPR* (and vice versa). If we decide that a false negative is 20 times as costly as a false positive, we could set τ at an optimum such that $\frac{FPR}{FNR} = 20$.

The trade-off leads to the construction of the *receiver operating characteristic* (ROC) curve [2], as shown in figure 2.2, showing values of the *TPR* and *FPR* on the vertical and horizontal axes respectively. Every value of τ corresponds to a point in the ROC curve, and it is common to use the area under the ROC curve (AUROC) as a measure of classifier performance that is independent of the particular setting of threshold τ.

[2] BISHOP, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics).* Springer-Verlag, Berlin, Heidelberg

The reason accuracy is not a very useful measure for urban drainage inspections specifically, is that it is a particularly unbalanced problem: defects are very uncommon [3]. The *FPR* (and the equivalent *TNR*) are dominated by the *TN* term in this unbalanced classification scenario, which isn't very interesting, as it is easy to achieve a very high *TN* by classifying everything as negative. In practice, this would mean that the detection system would never detect a defect, and in fact would be correct in that detection for 99% of cases. As the *FPR* is one of the dimensions of the ROC, this leads to the (AU)ROC only having limited usefulness. Instead of the *FPR*, we use *precision*, defined as:

[3] In the real-world datasets used in this thesis, we found defects to appear in approximately 1% of images.

$$Pr = \frac{TP}{TP + FP} \tag{2.20}$$

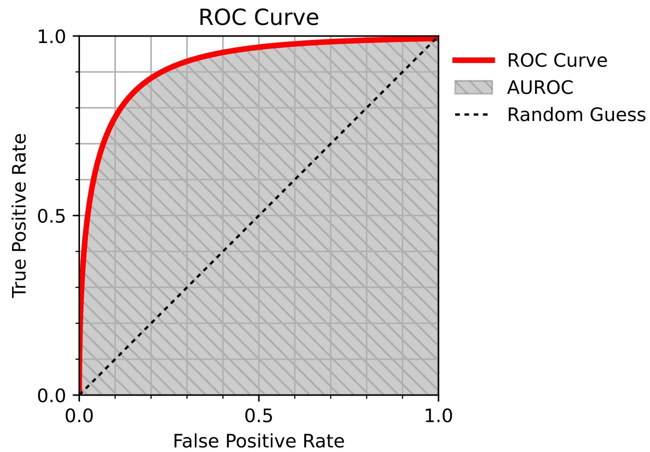


Figure 2.2: Example of a Receiver-Operating Characteristic (ROC) curve. Every point on the red line corresponds to a possible threshold τ, that defines the TPR and FPR. The shaded area shows the area under the ROC curve (AUROC), and the dashed line is the ROC curve one would obtain by randomly guessing the label for each instance in a binary classification scenario.

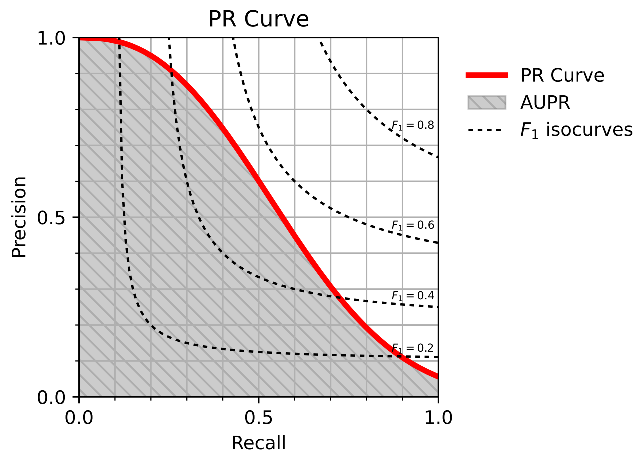


Figure 2.3: Example of a Precision-Recall (PR) curve. Every point on the red line corresponds to a possible threshold τ, that defines the precision and recall. The shaded area shows the area under the PR curve (AUPR), and the dashed lines are curves with a constant F_1-score.

Both the *FPR* and the precision are measures of the number of type I errors (detecting a defect when there is none) a classifier makes. The difference is that the *FPR* compares this to the total negative cases (*"How many of the sewer pipes without defects did we label as defective?"*), whereas the precision compares this to the total cases that were classified as positive (*"How many of the sewer pipes labeled as defective are false alarms?"*). The former is heavily skewed towards the appearance of a good performance, because of the prevalence of negative cases, but the latter does not have this issue.

If we now combine precision with *TPR*, we can construct a curve analogous to the ROC curve, called the Precision Recall [4] curve, or PR curve, as shown in figure 2.3. The area under the PR curve is more meaningful than the AUROC and also independent of τ. Figure 2.3 also displays F_1-score isocurves, curves where the harmonic mean of the precision and recall are constant. Being a function of precision and recall, the F_1-score is also a metric of performance that is not influenced by the overwhelming amount of false positives that rule the accuracy metric.

[4] Recall is a synonym for TPR.

2.1.5 Anomaly Detection

Anomaly detection, sometimes referred to as outlier detection, is a machine learning problem aimed at finding instances in a dataset that deviate from the majority [5]. It has many applications, from fraud detection to noise removal.

[5] Zimek, A. and Schubert, E. 2017. *Outlier Detection*. Springer New York, New York, NY, 1–5

Unsupervised anomaly detection relies in most cases on robust [6] regression. This means that we look for some model that explains the behaviour of *most* instances in our dataset. Any instances not explained by this model are considered to be anomalies or outliers.

[6] *Robust* in this context refers to a reduced sensitivity to noise or outliers.

An important quality of the model we fit on our data is that it has limited complexity. If the model's complexity is too high, it may fit the anomalies that we are trying to detect as well, meaning they become inliers and are no longer detected as anomalies. Still, a certain degree of complexity may be required in order to account for variation in the data. This implies a trade-off between how complex we allow patterns to be, and when complexities become anomalies. This is similar in concept to regularization as described in section 2.1.3: a regularised model may be well suited to detect anomalies by finding data that it does not generalise well to.

2.1.6 Principal Component Analysis

Principal Component Analysis (PCA) [7] is a popular tool in statistics, data science and many other scientific fields, used to reduce the dimensionality of data to facilitate data exploration and the use of algorithms that are sensitive to high dimensionality. It may be thought of as a form of unsupervised learning.

[7] Pearson, K. 1901. LIII. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11, 559–572

Given a dataset $\mathbf{X}$, consisting of N instances with d real-valued attributes each, we express this as an $[N \times d]$ matrix. PCA is performed by calculating the covariance matrix $\mathbf{C}$ of this matrix and performing eigenvalue analysis on $\mathbf{C}$. This results in d eigenvalues and d eigenvectors (or 'principal components') of length d. These eigenvectors form an orthonormal basis for the space in which our dataset $\mathbf{X}$ exists and the corresponding eigenvalues are proportional in magnitude to the variance of dataset $\mathbf{X}$ explained by each eigenvector (their sum being equal to the total variance present in $\mathbf{X}$). This establishes a transformation from the original space of d dimensions to a new space that also consists of d dimensions.

When using PCA for dimensionality reduction, we choose a dimensionality $\theta \leq d$ and project $\mathbf{X}$ [8] on the first θ eigenvectors (in order of descending eigenvalues). The projected matrix $\mathbf{P}$ retains as much variance as is possible [9] in θ dimensions. This allows researchers to view high-dimensional data in two or three-dimensional visualisations, or employ algorithms that are not designed for high-dimensional data.

[8] The covariance matrix and the newly-found orthonormal basis do not contain the mean values of the original dataset $\mathbf{X}$, so the dataset should be centred around zero before projecting onto the basis.

[9] Barring non-linear embeddings

When $\theta = d$, we can return to the original space by inverting the projection matrix and adding the mean values of each feature after transformation, without any loss of information. In chapter 3 we will utilise a partial projection as an unsupervised anomaly detection method.

2.2 Digital Image Processing

When using visual data for statistical learning, we will often convert this visual data to digital images, suited for computer processing. Digital images are represented as two-dimensional matrices of pixels. The pixels themselves may be scalar or vector values, for greyscale or colour images respectively.

In the case of scalar pixels that take binary values, we speak of a binary image or a *mask*. In the case of colour images, the most common representation is RGB [10], corresponding to digital screens, although HSL/HSV [11] and CMYK [12] are also common, depending on the application. By convention, for greyscale (and binary) images, higher values correspond to lighter tones, the maximum value corresponding to white and the minimum value corresponding to black. For convenience, the colourspace is often scaled to a range of $[0, 1]$.

[10] Red, Green, Blue

[11] Hue, Saturation, Lightness/Value

[12] Cyan, Magenta, Yellow, blacK

Sometimes the individual values of vector pixels may be called *channels*. We might speak of the 'hue' channel of an HSV image for example, which is itself an image with scalar values.

We write an image as $I(x, y)$, where x and y are the horizontal and vertical locations of the pixel in the image, and the value of I is the pixel value at that location.

2.2.1 Convolution

In signal processing, it is common to apply filters to signals through the use of a convolution operation. In the case of images, these filters can be used to smooth or sharpen certain patterns in images, like edges, corners, or textures. In convolution, a filter is moved across the image, and at

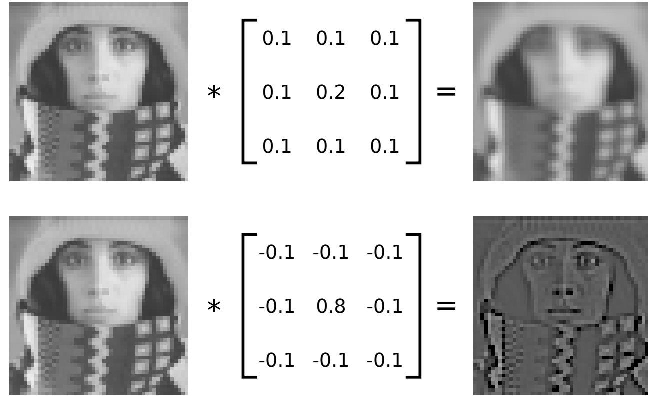


Figure 2.4: Example of how convolution with different kernels can be used to smooth an image or emphasise its edges.

each position the 'overlap' between the image and the filter is calculated.

Discrete [13] convolution in two dimensions is a mathematical operation defined as:

[13] As opposed to continuous convolution, which is defined for continuous signals. Our signals are images, consisting of pixels at discrete locations, and as such continuous convolution is beyond the scope of this thesis.

$$I(x, y) * k(x, y) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} I(u, v) k(x - u, y - v) \tag{2.21}$$

In practice we might smooth image $I(x, y)$ by convolving it with a Gaussian *kernel* $k(x, y)$. It should be noted that while convolution is defined on an infinite domain, in practice both the image and the kernel will be non-zero only within limited domains and the convolution can be performed by summing only over those domains. The kernel itself is commonly a scalar image, and as such the output of the convolution has the same number of channels as the image $I(x, y)$. Figure 2.4 shows two examples of how convolution can be used to smooth an image or to detect edges in an image.

2.3 Convolutional Neural Networks

Neural networks are a type of machine learning algorithm modelled after the way synapses in the human brain activate and pass information along [14]. Convolutional neural networks are a type of neural network that has been shown to work particularly well with image data [15]. This section will explain them into as much detail as is required for the context of this thesis.

[14] Goodfellow, I., Bengio, Y., and Courville, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

[15] Szeliski, R. 2010. *Computer Vision: Algorithms and Applications*, 1st ed. Springer-Verlag, Berlin, Heidelberg

2.3.1 The Perceptron

Neural networks are composed of neurons, smaller elements that perform a simple function. The network itself performs functions much more complicated than the neurons are capable of. The simplest neuron is Rosenblatt's perceptron [16]. A perceptron has inputs, weights, a bias, and an activation function. Each input is multiplied with its assigned weight, and all are added together with the bias. The single scalar resulting from this operation is passed through the activation function, which can be any function, but is usually a non-linear non-decreasing function [17]. (The reason it is non-linear is that if it were linear, the network itself would learn a linear function in the inputs, which means the network *would* be limited to a function no more complex than the neurons themselves.) Traditionally, the step function was used, but sigmoidal functions or piecewise linear functions are also common [18].

[16] Goodfellow, I., Bengio, Y., and Courville, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

[17] Bishop, C. M. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg

[18] Goodfellow, I., Bengio, Y., and Courville, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

The perceptron itself is a neural network, and can be employed as a classifier as defined in section 2.1.1, by using it as a function to describe a relationship between data and

labels:

$$f(\mathbf{x}) = \hat{y} = \phi\left(w_0 + \sum_j x_j w_j\right) \tag{2.22}$$

where x_j are the inputs, w_j are the weights, w_0 is the bias, $\phi(\cdot)$ is the activation function, and $\hat{y}$ is the output.

The weights and bias are initialised to random values, which means its output $\hat{y}$ is initially unlikely to resemble its target y much at all. The perceptron is trained through an iterative process called *backpropagation*, which brings the output $\hat{y}$ closer to the target y with every iteration, by changing the weights and bias by slight increments.

Backpropagation works by feeding a single instance x into the perceptron, which returns output $\hat{y}$. We then compare $\hat{y}$ to the expected output y with the loss function, and determine the derivative of the loss with regards to $\hat{y}$. We use the differentiation chain rule [19] to calculate the derivative of the loss with respect to each weight. A gradient step towards minimizing the loss is calculated by multiplying each weight's derivative by a (typically small) learning rate α and the weights are adjusted.

[19] Differentiation chain rule:

$$\frac{dL}{dw_j} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dw_j}$$

After sufficiently many iterations of the backpropagation process, the perceptron will have converged to a state where the backpropagation process cannot further reduce the loss function for the data it was trained on. This process often requires multiple *epochs*, the amount of iterations it takes for the entire dataset to be processed.

Multi-Layer Perceptron

To make a slightly more complex neural network, capable of learning more complex relations between input and output, we can chain multiple perceptrons together to create the multi-layer perceptron [20]. The perceptrons are ordered in several *layers*. Only the first layer has the dataset as its input, and each perceptron in the layer processes this data in parallel, each outputting a single scalar. The perceptrons in the

[20] Goodfellow, I., Bengio, Y., and Courville, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

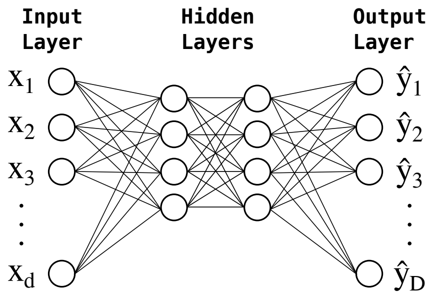


Figure 2.5: An illustration of a multilayer perceptron with 2 hidden layers.

next layer then each take these outputs from the previous layer as inputs, applying their weights, bias, and activation function as if these were input data, et cetera for consecutive layers. The final layer has as many neurons as the dimensionality of the output y.

Such a network in which data flows from input to output without any loops is called a *feed-forward* neural network. The backpropagation process still works the same way, except that the gradients require more complex calculations with each layer added. This way, adding additional layers allows us to model more complex functions by adding the same elementary neurons. An illustration is shown in figure 2.5.

The layers of neurons described in this section are called *dense* or *fully-connected* layers, as the output of each neuron in a layer is connected to the input of each neuron in the next layer. To construct a convolutional neural network, two additional layer types are required: the *convolutional* layer, a layer in which perceptrons have limited connections with the previous and next layers, and the *pooling* layer, a layer in which the neurons also have limited connections, but more importantly, the neurons perform a significantly different function from perceptrons.

2.3.2 Convolutional layers

Convolutional layers are generally used for image processing and perform the convolution operation as described in section 2.2.1.

We can simplify the perceptron somewhat to simulate this function. The advantage here is that we can use the backpropagation process to learn image filters, instead of having to design the filters based on what structure we expect the images to contain.

A simple convolutional layer consists of as many neurons as the previous layer in the network, but neurons are connected only to neurons in the previous layer that *surround* the neuron in the same location. In a one-dimensional setting, this would mean that neuron n_i in the convolutional layer receives as input the output from neurons n_{i-s} to n_{i+s} in the previous layer. We call $2s+1$ the *filter size*, as each neuron in the convolutional layer has $2s + 1$ inputs. In a higher-dimensional setting the neighbourhood around neuron n_i that is connected to neuron n_i in the next layer extends in all dimensions.

Additionally, each neuron in the convolutional layer has the same weights, just different connections. This is called *weight sharing*, and it results in an equivalent of the convolution operation performed by the network, as each neuron applies the same filter, but at a different location in the image.

Three additional hyperparameters are used to define a convolutional layer, the *stride*, the *depth*, and the *edge condition*.

The stride, or step size, incorporates a subsampling mechanism into the layer. If a stride of x is chosen in a one-dimensional setting, the convolutional layer contains N/x neurons, where N is the number of neurons in the previous layer. In other words, $N \cdot (x-1)/x$ neurons in the convolutional layer are discarded. The reasons one might do this,

is that if neither the kernel nor the image consist entirely of white noise, adjacent samples in the result will be highly correlated, and we can reduce the neurons in the layer (and with it the computational requirements of backpropagation) without losing much of the information present in the input data. In higher-dimensional settings, the stride has as many dimensions as the data because it can be defined for every dimension individually.

The depth of a convolutional layer determines how many filters are applied in parallel. With a depth of 1, every neuron is just a perceptron with limited connections, as mentioned earlier in this section. However, when the depth increases, each neuron becomes a collection of these limited perceptrons, and the output of the neuron is simply a vector of these perceptrons' outputs.

Finally, the edge condition determines what happens at the edges of an image, when the input does not fully overlap with the filter size. For example, with a filter size of $2s + 1$, the behavior for the first and last s samples of the input is poorly defined. It is an option to simply discard these cases, but this results in a layer size that is dependent on the filter size. Another option is to pad the input by half the filter size (rounded down) and then discard the violating cases, which does ensure the convolutional layer to be of the same size as the input layer. In this thesis we have chosen to apply zero-padding, black pixels are added to the edges of the images.

2.3.3 Pooling layers

A pooling layer is similar in structure to a convolutional layer, but instead of consisting of perceptrons, it consists of neurons that perform a non-linear function. They are often placed immediately after a convolutional layer, to introduce a non-linearity that allows the network to learn more complex structures than it would with just convolutional and

dense layers [21].

The most common example is the max-pooling layer. It has the same structure as the convolutional layer, taking as its input only the surrounding neurons from the previous layer, and has a filter size and stride as well. However, instead of multiplying the input by weights and summing, it simply outputs the maximum value of each depth slice from the inputs, as if we were performing a morphological dilation.

By taking the maximum value over a spatial window, we can perform a dimensionality reduction when the stride is larger than 1. The reason the maximum value is used is that this works well with the convolution operation: convolution overlaps two signals (image and kernel in this case) and returns the inner product, so a high response at a location means that the image resembles the kernel at that location. By taking the maximum, we achieve a sense of how much that portion of the image resembles each kernel (each depth slice).

[21] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

2.3.4 CONVOLUTIONAL NEURAL NETWORK DESIGN

Conventional wisdom in recent image processing techniques dictates a set of design patterns for convolutional neural network architectures that have been shown to work well. Specifically, the most common pattern is to interlace convolutional layers with max-pooling layers. The input is fed through some number of these interlaced layers successively before being passed into some number of dense layers, before being passed to the output.

The issue with designing and optimizing these networks is that the search space is infinite, and while there exists a lot of knowledge on what does and does not work for common datasets and tasks, still much research has to be done on *why* these are good design patterns [22].

[22] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. 2016. *Deep learning*. Vol. 1. MIT press Cambridge

Commonly, networks are used that have been shown to work well on difficult datasets (such as ImageNet), often pre-trained to reduce the time it takes to train on the new dataset, but the "no free lunch" theorem [23] dictates that there cannot be a single architecture that works best on all different tasks and datasets.

[23] Wolpert, D. H. and Macready, W. G. 1997. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation 1*, 1, 67–82

2.4 Computer Stereovision

Computer stereovision, or simply 'stereovision', is a computer vision technique in which two side-by-side cameras simultaneously record an image. The correspondence between points that appear in both images give us information on the distance from the cameras to that point, similar to how the correspondence between the left and right eyes allows humans to perceive depth [24].

[24] Howard, I. P. and Rogers, B. J. 2012. *Perceiving in depth, Volume 2: Stereoscopic vision*. Oxford University Press

To illustrate the principle, we examine the *epipolar plane* [25] of two horizontally aligned cameras and an object that is visible to both cameras, as shown in figure 2.6. The problem is significantly simplified by the camera axes being parallel, which is an achievable situation for the application of urban drainage inspections. $\mathbf{C}_1$ and $\mathbf{C}_2$ are the two cameras, and $\mathbf{P}$ the point of interest. Both cameras have identical physical properties, and we consider $\mathbf{C}_1$ to be the *reference camera*. f is the focal distance of the cameras, b is the baseline distance between the cameras, two physical distances that we know precisely. $\mathbf{I}_1$ and $\mathbf{I}_2$ are the *virtual* image planes, one focal length distance in front of the cameras.

[25] Szeliski, R. 2010. *Computer Vision: Algorithms and Applications*, 1st ed. Springer-Verlag, Berlin, Heidelberg

We wish to calculate X and Z, the physical location of $\mathbf{P}$ in the epipolar plane, from the perspective of our reference camera $\mathbf{C}_1$. Consider d_1 and d_2, the projected locations of $\mathbf{P}$ onto $\mathbf{I}_1$ and $\mathbf{I}_2$, relative to the centres of the image planes [26].

[26] Note that we take d_2 to be a negative value in this case, as it is to the left of the centre of $\mathbf{I}_2$. If $\mathbf{P}$ would be on the same side of both camera axes, d_1 and d_2 would have the same sign.

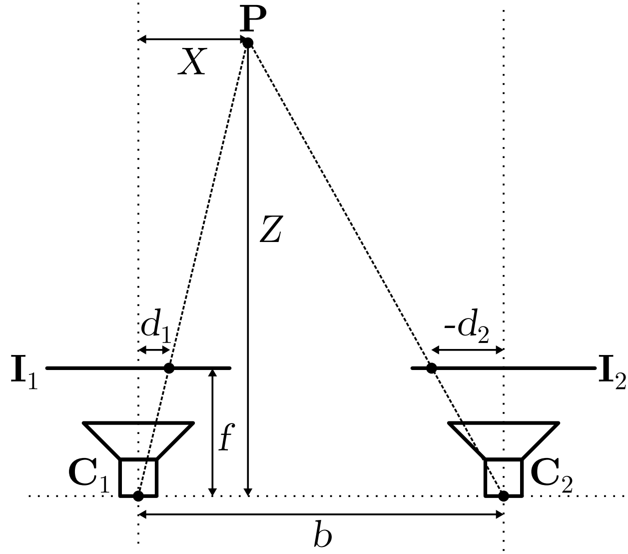


Figure 2.6: Epipolar geometry with parallel camera axes

Similar triangle geometry allows us to solve for Z and X:

$$d_1/f = X/Z \tag{2.23}$$
$$-d_2/f = (b - X)/Z \tag{2.24}$$
$$(d_1 - d_2)/f = b/Z \tag{2.25}$$
$$Z = b \cdot f/(d_1 - d_2) \tag{2.26}$$
$$X = d_1 \cdot b/(d_1 - d_2) \tag{2.27}$$

Two important things should be noted at this point:

◇ The Y coordinate of **P**, which we neglected in this planar example for simplicity, also has to be computed.

$$Y = d_y \cdot b/(d_1 - d_2) \tag{2.28}$$

where d_y is the vertical position relative to the centre of the projection of **P** on $\mathbf{I}_1$. Since the cameras are aligned in the horizontal plane, there is no need to take a vertical shift into consideration, as any point

will be projected on both virtual image planes at equal height.

◇ For the calculated coordinates to be represented in physical units, we can either express d_1 and d_2 in physical units, or we can express f in pixels instead of physical units. Either conversion is done by finding the physical size of a pixel on the camera's sensor array. In this thesis, we will assume the focal length f is expressed in pixels.

Stereovision algorithms apply this principle to all pixels in an image: each pixel is considered to be a projection of some point with physical coordinates that we try to find. Each pixel in the image produced by the reference camera is matched to a pixel in the second image, and the difference in horizontal positions of these pixels produces the *disparity* $(d_1 - d_2)$. More specifically, for each pixel in the reference image, a local neighbourhood around the pixel is compared to a patch of the same size as this neighbourhood, in the same position in the other image, but shifted horizontally. The horizontal shift that minimises the difference between the two image patches is considered the best match.

This introduces multiple difficulties, as finding the correspondence between pixels is a heuristic search process with multiple local optima, as exhaustive search is often infeasible. Images that have some periodicity in the horizontal direction may result in the correspondence being off by a multiple of the period. An even bigger challenge arises when the selected neighbourhood patch is entirely smooth: matching the exact location will become difficult, as small shifts lead to little difference in matching quality. Practically, an exact alignment of the cameras is also difficult to achieve, and any physical camera and lens are going to introduce distortion to the recorded images [27], both of which will have to be corrected before the matching process commences.

[27] Hecht, E. et al. 2002. *Optics*. Vol. 5. Addison Wesley San Francisco