Surprise-based online learning for sensor systems

Maxim Bonnaerens

Supervisor: Prof. dr. ir. Joni Dambre
Counsellors: Prof. dr. ir. Joni Dambre, Ir. Len Vande Veire

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Koen De Bosschere
Faculty of Engineering and Architecture
Academic year 2017-2018

GHENT
UNIVERSITY

Permission of use of loan

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

Maxim Bonnaerens, August 2018

Preface

I would like to thank my supervisor prof. dr. ir. Joni Dambre for providing me with the opportunity to do research on this topic. This dissertation would not have been possible without her wise guidance and support. I would also like to thank my counsellor ir. Len Vande Veire for his input and for being always available for my questions.

Next, I would like to express my gratitude towards my fellow students and friends David Vercauteren, for the countless hours we spent together at the student lab working each on our own master's dissertation and for all the breaks in between, and Henri Verroken, for joining us during these breaks and proofreading this work.

I also deeply appreciate my parents and my brother, for their unconditional love and support. Without them I would not be the person I am today.

Surprise-based online learning for sensor systems

by

Maxim Bonnaerens

Supervisor: Prof. dr. ir. Joni Dambre
Counsellors: Prof. dr. ir. Joni Dambre, Ir. Len Vande Veire

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Koen De Bosschere
Faculty of Engineering and Architecture
Academic year 2017–2018
Ghent University

Abstract

Surprise is an emotion emerging from the mismatch between expectations and what is actually experienced or observed. In this master's dissertation our concern is not with the emotion of surprise but rather with the quantification of surprise. We look at surprise in the context of deep learning, where being surprised means that the neural network is processing data that it did not expect as it is too dissimilar to the data on which the network was trained. When a neural network is surprised, it makes inaccurate predictions, which can only be measured when the true labels of the inputs are known. In this work we define three surprise metrics that make it possible to measure surprise in the context of deep learning, without the need of having the true labels. These surprise metrics are reconstruction accuracy surprise, hidden activations surprise and reconstruction loss surprise. The proposed metrics make it possible to adapt deep learning models to unexpected changes to the input. This can be used to train sensor systems in an online fashion so that they can adapt to surprising situations.

Keywords: *surprise, deep learning*

Surprise-based online learning for sensor systems

Maxim Bonnaerens
Supervisor: Prof. dr. ir. Joni Dambre
Counsellors: Prof. dr. ir. Joni Dambre, Ir. Len Vande Veire

***Abstract*—Surprise is an emotion emerging from the mismatch between expectations and what is actually experienced or observed. In this work our concern is not with the emotion of surprise but rather with the quantification of surprise. We look at surprise in the context of deep learning, where being surprised means that the neural network is processing data that it did not expect as it is too dissimilar to the data on which the network was trained. When a neural network is surprised, it makes inaccurate predictions, which can only be measured when the true labels of the inputs are known. In this work we define three surprise metrics that make it possible to measure surprise in the context of deep learning, without the need of having the true labels. These surprise metrics are reconstruction accuracy surprise, hidden activations surprise and reconstruction loss surprise. The proposed metrics make it possible to adapt deep learning models to unexpected changes to the input. This can be used to train sensor systems in an online fashion so that they can adapt to surprising situations.**

***Index Terms*—surprise, deep learning**

I. Introduction

Surprise is something everybody knows what it feels like. It is a feeling caused by something that is unexpected or unusual. Surprise can occur in a variety of intensities, certain events can be perceived as very surprising while other events may only be observed as a little surprising. When surprising events start reoccurring more regularly, over time they will be observed as less and less surprising.

Surprise is often used interchangeably with novelty in many studies, and while closely related, they are very different from each other [1]. Novelty refers to the property of being not previously encountered or experienced while surprise refers to the result of experiencing something suddenly or unexpected.

When using machine learning the assumption is made that the samples used during training are similar to the ones seen during inference. As a result, machine learning cannot handle surprising events properly. For example, when all of a sudden something in the setup produces rotated images during inference, the network will continue making predictions about the labels of these rotated images. When the network has not encountered rotated images before, it will most likely be inaccurate. However, it is impossible to detect this change in accuracy because it happens during inference where the true labels are no longer available. Therefore the network is no longer reliable. Adding a notion of surprise to neural networks allows recovering from an unexpected situation. Being able to use this notion of surprise can then be used to retrain the network on rotated images to adapt to the new samples seen during inference. Retraining the network to adapt to the surprising situation is a way of doing online learning, as opposed to the standard way of training once and then doing inference.

Currently, the most commonly used machine learning technique to prevent surprising events is to anticipate all possible surprising events and already include them during training using data augmentation. When using data augmentation, all augmentations of the original image are also processed during training. This makes the network robust to any anticipated surprising situation (which are as a result no longer surprising). However, creating such a robust network results in a larger network than strictly necessary. This has as a downside that such a network requires a more extensive training process which needs more processing power, more memory and consumes more energy. A smaller network that is tailored to a specific task, on the other hand, has as advantages that it can do faster inference, needs less resources and consumes less energy. It has as downside that it is not robust to surprising situations. Adding a notion of surprise to such a smaller network can enable it to adapt to surprising situations through online learning, while keeping the advantages of being a smaller network.

While surprise is an intuitive concept and could prove to be useful in machine learning, one of the key problems is that it is hard to quantify. The goal of this work is to define and validate surprise measurements that can be used in the context of deep learning for image classification tasks.

II. Technical Background

In neuroscience it has been recognised that the amplitude of the P300 component of event-related brain potentials reflects the degree of surprise [2]. However, outside the context of neuroscience, surprise has no standard metric to be quantified with. In what follows, several alternative definitions are introduced

A. Kullback-Leibler divergence

The Kullback-Leibler divergence (D_{KL}) plays a role in both the Bayesian surprise formula (4) and the confidence-corrected surprise (6), which are discussed later. This is not unexpected as the Kullback-Leibler divergence on its own can also be seen as a measure of surprise. The D_{KL} has its origin in information theory. The Kullback-Leibler divergence [3] is defined as:

$$D_{KL}(P||Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)}. \quad (1)$$

It measures how one probability distribution differs from the expected probability distribution. $D_{KL}(P||Q)$ is the divergence from Q to P, and it measures the information gained when changing the prior distribution Q to the posterior distribution P. If applicable, P represents the true distribution of the data while Q represents a model or approximation of P. The D_{KL} is defined only if for all i, $Q(i) = 0$ implies $P(i) = 0$. An important property is that the Kullback-Leibler divergence is additive for independent distributions. If P_1 and P_2 are independent distributions with joint distribution P and likewise for Q_1, Q_2 and Q, then $D_{KL}(P||Q) = D_{KL}(P_1||Q_1) + D_{KL}(P_2||Q_2)$.

B. Shannon Surprise

In information theory, Shannon's information content [4] or self-information

$$\mathrm{I}(X) = -\ln p(X), \tag{2}$$

is often associated with surprise, and therefore also sometimes called surprisal. The information content of a sample X measures the amount of information that is revealed when observing X. This metric of surprise will further be referenced as Shannon surprise. Unlikely events have a high amount of information. Conversely, when a common event is observed, a small amount of information is associated with this event.

C. Bayesian Surprise

Another way to look at surprise comes from Bayesian theory. In the Bayesian framework models or hypotheses are associated with confidence or belief. Bayesian surprise is then defined as the changes for a given observer that take place when going from prior to posterior distributions of the data [5]. When considering a set of possible models $\mathcal{M}$, an observer has a prior distribution $P(M)$. Observing new data D leads to reevaluating the beliefs and changes the prior probability into a posterior probability. Bayes theorem states that

$$P(M|D) = \frac{P(D|M)}{P(D)} P(M). \tag{3}$$

From this formula it can be seen that the effect of D is that it can be used to change $P(M)$ to $P(M|D)$. The surprise of data D is then defined as the Kullback-Leibler divergence between the prior and the posterior

$$S(D, \mathcal{M}) = D_{KL}(P(M)||P(M|D)). \tag{4}$$

The alternative version $S(D, \mathcal{M}) = D_{KL}(P(M|D)||P(M))$ is also used and may be even preferred if the "best" distribution is used as first argument.

D. Confidence-corrected surprise

The Shannon surprise and Bayesian surprise formulas are distinct formulas to capture surprise yet they can be seen as complementary. Confidence-corrected surprise [6] is a formula that tries to combine both surprise measures by using their complementary benefits and overcoming their individual shortcomings. First it replaces Shannon surprise by a weighted average over all possible model parameters θ. The weight is determined by the current belief $\pi_0(\theta)$. This gives the raw surprise

$$S_{raw}(X; \pi_0) = -\int_\Theta \pi_0(\theta) \ln p(X|\theta) d\theta. \tag{5}$$

This is also equal to the sum of the Shannon and Bayesian surprise: $S_{raw}(X; \pi_0) = Surprise_{Shannon}(X; \pi_0) + Surprise_{Bayes}(X; \pi_0)$. Confidence corrected surprise extends this by also including the confidence in his belief, in the formula. This confidence is represented by the entropy of the current belief about the model parameters $\mathcal{H}(\pi_0)$. Confidence-corrected surprise is then derived from subtracting the entropy from the raw surprise, where $p(X|\theta)$ is normalised to the scaled likelihood $\hat{p}_x(\theta)$.

The confidence-corrected surprise then becomes:

$$S_{corr}(X; \pi_0) = \int_\Theta \pi_0(\theta) \ln \frac{\pi_0(\theta)}{\hat{p}_x(\theta)} d\theta = D_{KL}(\pi_0(\theta)||\hat{p}_X(\theta)). \tag{6}$$

III. Setup

In the context of deep learning, surprise is a metric that captures a persistent mismatch between the expected characteristics of the input data and the characteristics of the observed data. In this work we define a neural network as being surprised when an unexpected event occurs that affects the stream of input data. This means that a surprising event does not just influence a single data sample but rather a sequence of samples following the event. In this work we define unexpected and thus surprising events as events that are unlikely to happen according to the characteristics of the data that the model learned during training. As a consequence, surprising events often result in inaccurate predictions of the model. Sometimes it can be useful to measure the mismatch between a single input sample and the expected characteristics of the input data. Capturing such 'surprise' of a single sample is covered by the field of novelty and outlier or anomaly detection [7], [8], but this is not what is regarded as surprise in this work.

Concretely we will work with image classification on the MNIST dataset [9] where the unexpected event is rotating all inputs with a fixed angle between 0° and 90°, and where the model has not seen any rotated inputs during training.

A. Reconstruction accuracy surprise

The main idea behind reconstruction accuracy surprise is that autoencoders are only able to reconstruct images that are similar to the ones seen during the training phase of the autoencoder, which is a characteristic that can be used to measure surprise. The typical deep learning architectures for image classification are convolutional neural networks, which consists of several convolutional layers, optionally followed by a few fully-connected layers. This neural network can be trained using the standard techniques such as gradient descent and regularisation.

In order to capture surprise, the classifier is expanded into an autoencoder architecture by adding decoder layers directly

to the output layer of the classifier, or to one of the last hidden layers of the classifier. The actual configuration is task dependent. For example, when only a few classes appear in the output layer, it is recommended to use the second to last or an even earlier layer, so that a latent space with a higher dimension can be used. An important requirement however is that this latent space is small enough so that nearly lossless reconstruction cannot happen, as a lossless autoencoder will be able to reconstruct any input, regardless whether it is similar to input data seen during training or not.

After the network is extended, it is retrained as an autoencoder. However, it is not trained in the default way autoencoders are trained. The layers that are originally part of the classifier keep their weights fixed and only the weights of the decoder layers are trained. The autoencoder is trained on the same data that has been used to train the classifier part.

To measure the reconstruction accuracy surprise, each data sample $\boldsymbol{x}$ is first passed through the classifier part of the network, resulting in output $\boldsymbol{p} = [p_1, ..., p_k]$ where p_i is the probability that the input belongs to class C_i of the k possible classes. Next, the data sample $\boldsymbol{x}$ is fed through the entire architecture including the decoder to reconstruct image $\boldsymbol{x}^*$, this second pass can continue from the activations of the last common part of the full classifier and the autoencoder. The reconstructed data sample $\boldsymbol{x}^*$, is then again passed through the classifier network, resulting in output $\boldsymbol{p}^*$. The reconstruction accuracy divergence (D_{RA}) of a single data sample is then calculated as

$$D_{RA} = D_{KL}(\boldsymbol{p}||\boldsymbol{p}^*). \tag{7}$$

As surprise in the context of this work is not applied to a single sample, this metric is averaged over a sequence or batch of samples in order to come to the reconstruction accuracy surprise of the batch. The reconstruction accuracy surprise is thus defined as,

$$Surprise_{RA} = \sum_i D_{KL}(\boldsymbol{p_i}||\boldsymbol{p_i}^*). \tag{8}$$

B. Hidden activations surprise

An alternative method for surprise in the context of deep learning is the hidden activations surprise ($Surprise_{HA}$). This formula does not use an autoencoder to capture surprise but is based on the activations of the last hidden layer. As in the previous formula, the classifier is first trained in the standard way. However, part of the training set is not used during training and kept separately. This unused part is then fed through the network and the outputs of the last hidden layer are recorded. These outputs are then used to estimate the probability density function of the output values for each node in the last hidden layer. Let $P(h_1, ..., h_n)$ be the joint probability of the outputs of the last hidden layer, with h_i the output value of node i. After training, during inference, the same methodology can be applied to a set of new samples, to again estimate the joint probability of the outputs of the last hidden layer, let this new estimate be $P^*(h_1, ...h_n)$. This estimate will be very similar to P if the samples of this new set are similar to the ones used for estimating P, but will likely be different if the samples in the new set capture a surprising event. The hidden activations surprise formula measures surprise as

$$Surprise_{HA} = D_{KL}(P(h_1, ..., h_n)||P^*(h_1, ..., h_n)). \tag{9}$$

However, as the last hidden layer can have hundreds to thousands of nodes, approximating the joint probability becomes infeasible due to the curse of dimensionality. This problem is solved by making the naive Bayes assumption that all probabilities $P_1(h_1), ..., P_2(h_2)$ are independent. Using the property that the Kullback-Leibler is additive for independent distributions then gives

$$Surprise_{HA} = \sum_i D_{KL}(P_i(h_i)||P_i^*(h_i)). \tag{10}$$

The observation that not every hidden node influences the final output equally allows reducing the complexity even more. Therefore only a subset of the most activated nodes is used. In this context, a node is activated when it outputs a value greater than 0 when using a ReLU activation function.

C. Reconstruction loss surprise

The third and last surprise formula tested in this work is the reconstruction loss surprise ($Surprise_{RL}$). It combines elements from both the reconstruction accuracy surprise method and the hidden activations surprise method. The model architecture is the same as used for $Surprise_{RA}$. Training happens in a similar fashion as well. First, the classifier is trained, followed by the autoencoder. The weights of the classifier stay fixed during this last training stage. However training is not done on the full training set, a subset is kept separately just as with hidden activations surprise method. After this two-stage training process is done, the part of the training set that was kept separately is fed through the entire architecture and the reconstruction loss of the autoencoder is recorded. The reconstruction losses of the samples are then used to estimate the probability distribution of the reconstruction loss. Let P_l be the probability density function of the reconstruction loss, approximated during training. After training, the same probability density function can be estimated using new sequences of data samples. Let this new estimate be P_l^*. This estimate will again be very similar to P_l if the set used for the new estimate is similar to the subset used during training. The reconstruction loss surprise is calculated as

$$Surprise_{RL} = D_{KL}(P_l||P_l^*). \tag{11}$$

D. Model architecture

The baseline architecture used for the experiments, including the decoder part, is illustrated in figure 1. The blue part shows the classifier, while the green part shows the autoencoder extension that is used in reconstruction accuracy surprise and reconstruction loss surprise.

The classifier consists of three convolutional layers with respectively 32, 64 and 128 filters. The last convolutional layer is then flattened and used as input of a fully connected

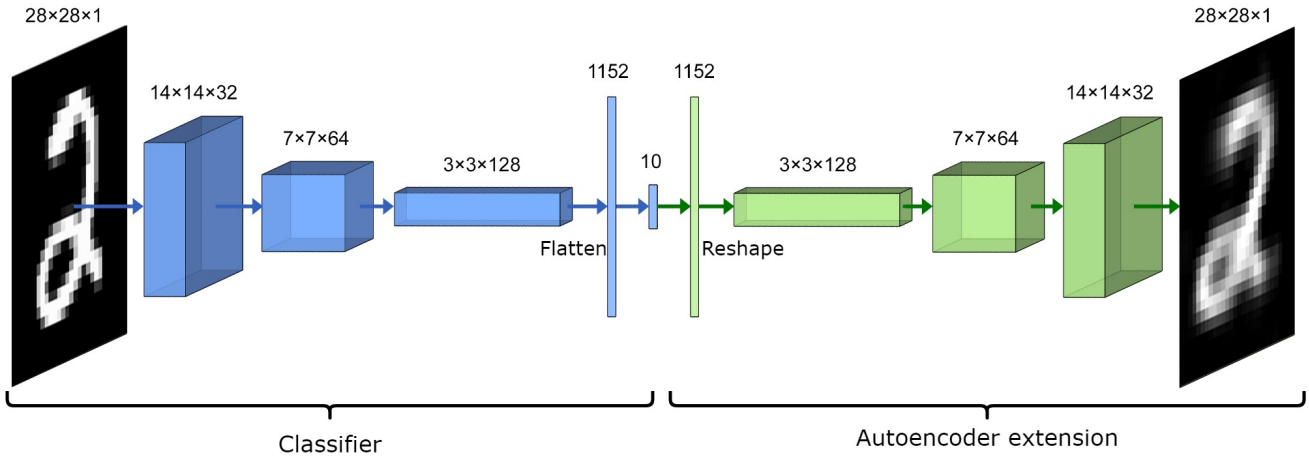


Fig. 1. Architecture used in the experiments. The blue part shows the classifier, while the green part shows the autoencoder extension that is used in reconstruction accuracy surprise and in reconstruction loss surprise.

layer on which the output appears. A softmax function is then applied to the output, in order to obtain probabilities for each output class. The autoencoder extension used in reconstruction accuracy surprise and reconstruction loss surprise is applied directly to the output layer of the classifier, but skipping the softmax function. The 10 outputs thus become the inputs of a fully-connected layer with 1152 outputs. These outputs are reshaped into 128 patches of 3×3 to which several transpose convolutional layers are applied in order obtain to an output image of 28×28.

IV. Results

The classifier achieves an accuracy of 99.11% on the test set after 10 epochs. The autoencoder achieves reconstruction error of 0.3299, which is a good performance.

What is important for both the reconstruction accuracy surprise and the reconstruction loss surprise is that the autoencoder is only able to reconstruct images that are similar to the ones seen during training and that it cannot reconstruct surprising images with the same precision. This is tested by rotating the test set over different angles and measuring the reconstruction loss. The loss increases as the angle of the rotation goes from 0° to 90° anticlockwise, which means that the autoencoder is performing worse on 'surprising' samples.

A. Rotation invariant network

The current way of dealing with possible surprising events such as rotated images, is to apply data augmentation during training. This means that during training, rotations are also added to the inputs. This way, whenever the surprising event of rotated input images happens, the network can still correctly classify them as it has seen these kinds of images during training. However, training a network that can accurately classify the MNIST dataset with random rotations between 0 and 90 degrees requires a bigger network architecture. Achieving a 99% accuracy on this augmented dataset is no longer simple. There are networks that achieve 99% accuracy on rotated MNIST by using rotation-invariant convolutional filters [10] but such architectures are not relevant to try in this context as rotation is just one example of surprise. Data augmentation on the other hand, is a technique that can adapt the network to any kind of surprising event, as long as this surprising event can be anticipated during training. We adapted our baseline model to achieve the highest possible accuracy on MNIST with random rotations between 0 and 90 degrees. The resulting network has a total of 1,689,594 parameters. This is more than ten times the amount of parameters than the classifier part of the baseline and more than five times compared to the extended baseline. In addition, it is also trained for 40 epochs instead of just 10 as more data needs to be processed and more parameters need to be learned.

B. Reconstruction accuracy surprise

Reconstruction accuracy surprise depends on the autoencoder extension and the ability of the network to make predictions on the reconstructed inputs. As mentioned in the previous section, the average reconstruction loss is 0.3299. The classification accuracy of the reconstructed inputs of the test set is 94.22%. This means that the network does a reasonable job at making accurate predictions of the reconstructed images. Calculating the reconstruction accuracy surprise on the test set gives $Surprise_{RA} = 0.1307$. This value is the surprise for the regular, non-rotated test set, which should not be surprising to the network. The surprise is then calculated over rotated variants of the test set. Starting from a rotation of 5° which should be only a little surprising up to a rotation of 90°, which should be perceived as very surprising as the accuracy on this rotation is only 18%. The $Surprise_{RA}$ values range from 0.2 for a rotation of 5° to 1.9458 for a rotation of 90°. Figure 2 shows the development of the $Surprise_{RA}$ value compared to

the classification error. The surprise metric shows that for the least surprising event, i.e. a rotation of 5°, the surprise goes up by 53% while the error goes up 32%. Over the course of all rotations, the surprise metric seems to have similar characteristics compared to the classification error. However, measuring surprise does not require labelled samples while measuring the classification error does.

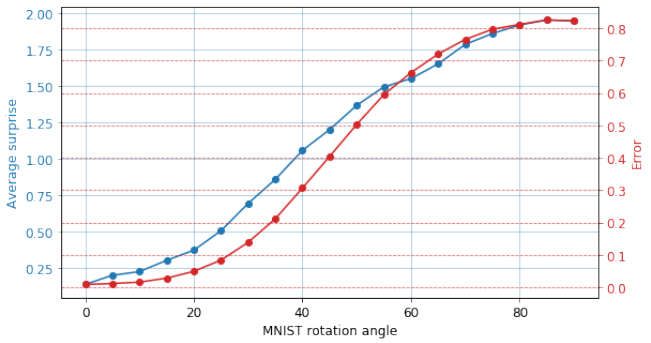


Fig. 2. $Surprise_{RA}$ and the classification error curves.

C. Hidden activations surprise

Hidden activations surprise captures surprise without using the autoencoder extension, $Surprise_{HA}$ is measured by estimating $P(h_1, ..., h_{1152})$, the joint probability of the outputs of the last hidden layer. Estimating this probability distribution function is done by observing the outputs at this hidden layer on a set of data points.

Estimating the joint probability $P(h_1, ..., h_{1152})$ is impractical due to the curse of dimensionality. To overcome this limitation, we assume that all probabilities $P_1(h_1), ..., P_{1152}(h_{1152})$ are independent, just as done in the naive Bayes algorithm. Additionally, to further simplify the surprise calculation, only a subset of the 1152 nodes is taken as not every node contributes equally to the final output predictions. The selection of which nodes to use, is done by looking at the nodes that are activated the most. As the activation function in the last hidden layer a ReLU is used, hence a node can be seen as activated when its value exceeds 0. The complete node selection procedure is as follows: first, for all the samples in the surprise training set, the values at the last hidden layer are stored together with the predicted label. Next, for each possible output class, the nodes that are activated the most are selected. Finally, for each class the most activated nodes are added to the final set of nodes until this set contains at least 50 nodes.

Estimating the probability density function for each of the selected nodes happens in two stages. As a ReLU activation function is used, most of the nodes have a high percentage of values that are 0. $P(h_i = 0)$ is simply estimated as the fraction of the samples in the surprise training set that have a 0 output for hidden node h_i. For the values greater than zero, the probability distribution is approximated using Gaussian kernel density estimation (KDE) [11].

To capture surprise, this estimation is also done over the test set. This estimation P^* is then used to calculate $Surprise_{HA}$ using equation (10). The $Surprise_{HA}$ of the non-rotated test set, using all samples in the test set to estimate P^*, is 0.0685. The surprise is then calculated over rotated versions of the test set, up to a rotation of 90°. Figure 3 shows the development of the surprise values and the classification error as the test set gets rotated from 0° to 90°. The least surprising test set, which has a rotation of 5°, has a surprise of 0.5476. This is a value that is 8 times larger than the non-surprising test set. The most surprising variant of the test set, the version rotated by 90°, has a surprise of 14.496.

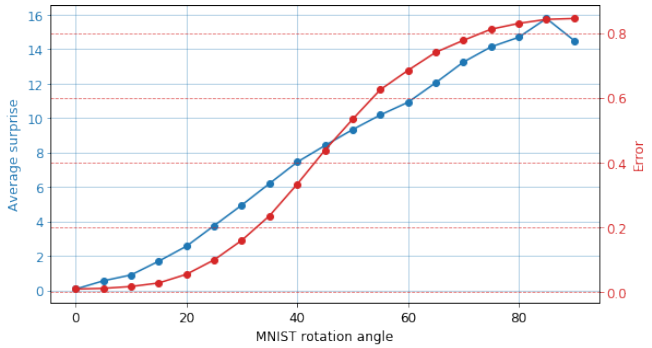


Fig. 3. $Surprise_{HA}$ and classification error curves.

D. Reconstruction loss surprise

The third and final surprise method tested is the reconstruction loss surprise, which is a combination of the previous formulas. The architecture of the full network is the same as used in reconstruction accuracy surprise, as both formulas use the autoencoder extension. However, as a probability distribution function needs to be estimated just as in hidden activations surprise, the training set is split in the same fashion as done for $Surprise_{HA}$. This has as a consequence that the classifier accuracy is also 99.07%. Training the autoencoder with this modified training set results in an average reconstruction error of 0.334, which is only slightly higher than the reconstruction loss when trained on the full training set.

The probability distribution function P_l of the reconstruction loss between $\boldsymbol{x}$ and $\boldsymbol{x}^*$ is estimated and used to measure surprise with. Estimating the probability distribution P_l of the reconstruction loss is achieved by storing the losses for all samples of the surprise training set and then applying Gaussian KDE to approximate a probability density function. Next, the probability distribution of the reconstruction loss is estimated over the data set of which we want to measure the surprise. This approximation P_l^* is then used in equation (11) to calculate the reconstruction loss surprise. The $Surprise_{RL}$ of the non-rotated test set is 0.00113 when using the test to approximate P_l^*. The test set is then rotated from 0° to 90°, and the surprise is calculated over these variants of the test set. The least surprising test set, which has a rotation of 5°, has a $Surprise_{RL}$ of 0.0258. This is more than 20 times higher than the surprise for the non rotated test set, while the classification error only slightly increases from 0.93% to 1.15%. The most

surprising version of the test set is the one with a rotation of 90°. The accuracy on this rotated version is as low as 15.5%, which corresponds to a $Surprise_{RL}$ of 2.185. Figure 4 shows the evolution of the surprise values compared to the classification error as the test set gets rotated to 90 degrees. Over the course of all rotations, the surprise metric seems to show similar behaviour compared to the classification error.

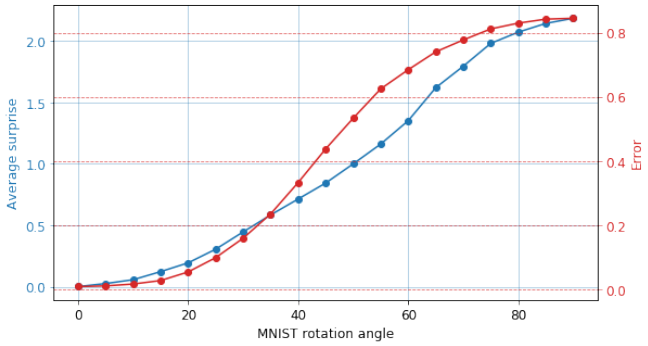


Fig. 4. $Surprise_{RL}$ and classification error curves.

V. Discussion

The presented surprise metrics prove being capable of capturing surprising situations without the need of having the true labels of the data. From the conducted experiments it cannot be concluded which formula is the best to use in order to capture surprise. All formulas show behaviour that is desired in a surprise metric. In order to have a clear view of which formula is best, more experiments need to be done on larger and more complex problems and using a wider range of surprising situations. However, reconstruction accuracy surprise is likely to be the least useful metric due to its sensitivity on the autoencoder performance.

The hardest part of the various presented methods is the interpretation of the surprise result. What does it mean to have a $Surprise_{RA}$ of 1.254, a $Surprise_{HA}$ of 5.89 or a $Surprise_{RL}$ of 0.67? The answer is that it is very much task and model dependent, a $Surprise_{RL}$ of 0.025 might be surprising for a certain dataset and deep learning model, while that value may indicate no surprise on another dataset and model. In order to determine which value is surprising for a given dataset and model, first the average surprise over the test set needs to be measured. Once this is known a threshold for surprise can be set by looking at the surprise values for all the different batches of the test set. A good threshold would be a slightly higher value than the maximum surprise value found across all the batches in the test set. For example, if for a given dataset and model the average $Surprise_{RL}$ is 0.016 with some batches having values close to 0.25, a $Surprise_{RL}$ threshold to flag a situation as surprising could be set at around 0.30 or higher.

VI. Conclusion

Every human knows surprise as the feeling one gets when something unexpected or unusual happens. It captures our attention and enables us to learn from a new experience. Deep learning models work differently, while their architecture is roughly based on the brain, they do not have a notion of surprise. Deep learning models apply what they have learned to new inputs. When something in the input images changes and creates inputs that are unexpected to what the model has learned, the network will try to keep applying its knowledge on it, but will fail to make accurate predictions.

By measuring the surprise of a deep learning model on input sequences, it becomes possible to track the network performance on new situations without having access to the true classes of the data. Adapting to surprising situations using online learning, is key for machine learning models to stay accurate when inputs suddenly behave unexpectedly.

This work defined three metrics to quantify surprise: reconstruction accuracy surprise, hidden activations surprise and reconstruction loss surprise. Reconstruction accuracy surprise and reconstruction loss surprise are metrics based on the idea that an autoencoder is only able to reconstruct images that are similar to the ones seen during training. Hidden activations surprise is based on the idea that the nodes in the last hidden layer of a neural network activate differently during surprising situations. The proposed surprise metrics offer measurements that can be used to capture the surprise of a deep learning model in image classification tasks. The surprise metrics prove being capable of capturing surprising situations. Retraining a model to surprising situations also correctly lowers the surprise again to a non-surprising value. While more experiments on more complex and industry relevant deep learning tasks should be conducted, surprise shows to be a promising tool for deep learning applications.

References

[1] A. Barto, M. Mirolli, and G. Baldassarre, "Novelty or surprise?" *Frontiers in psychology*, vol. 4, p. 907, 2013.

[2] E. Donchin, "Surprise! surprise?" *Psychophysiology*, vol. 18, no. 5, pp. 493–513, 1981.

[3] S. Kullback, *Information Theory and Statistics*. New York: Wiley, 1959.

[4] S. C. E., "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1948.tb01338.x

[5] P. Baldi and L. Itti, "Of bits and wows: a bayesian theory of surprise with applications to attention," *Neural Networks*, vol. 23, no. 5, pp. 649–666, 2010.

[6] M. Faraji, "Learning with surprise," 2016.

[7] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial intelligence review*, vol. 22, no. 2, pp. 85–126, 2004.

[8] D. Xu, E. Ricci, Y. Yan, J. Song, and N. Sebe, "Learning deep representations of appearance and motion for anomalous event detection," *arXiv preprint arXiv:1510.01553*, 2015.

[9] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[10] M. Weiler, F. A. Hamprecht, and M. Storath, "Learning steerable filters for rotation equivariant cnns," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[11] D. W. Scott, *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

Contents

Acronyms

AI Artificial Intelligence.

CAE Convolutional Autoencoder.

CNN Convolutional Neural Network.

KDE Kernel Density Estimation.

MLP Multilayer Perceptron.

MSE Mean Squared Error.

ReLU Rectified Linear Unit.

Chapter 1

Introduction

Surprise is something everybody knows what it feels like. It is a feeling caused by something that is unexpected or unusual. Surprise can occur in a variety of intensities, certain events can be perceived as very surprising while other events may only be observed as a little surprising. When surprising events start reoccurring more regularly, over time they will be observed as less and less surprising. Surprise also plays a vital role in learning as it raises our attention [16]. It is a concept that is also covered in neuroscience where it has been studied how it is triggered in the brain and what its effects are [10].

Surprise is often used interchangeably with novelty in many studies, and while closely related, they are very different from each other [5]. Novelty refers to the property of being not previously encountered or experienced while surprise refers to the result of experiencing something suddenly or unexpected. Surprise often, if not always, accompanies novelty. When something is encountered for the first time, it should not only trigger novelty but also surprise, because the novel item could not have been predicted or expected. On the other hand, it is obvious that surprise does not imply novelty. A familiar situation may be surprising in a context where something else is expected. This thesis focuses on surprise. While our research may also be useful in areas focusing on novelty such as novelty detection, the focus in this work is on situations where something unexpected happens that persistently changes the observations after the event.

The goal of this thesis is to use the concept of surprise in the context of deep learning and more specifically on tasks involving sensor systems. Concretely we will look into image classification which is a task performed on data from an image sensor. Image classification is the task of predicting the class of given input images, categorising the image as one class of a defined set of classes. Each class usually corresponds to the main object that can be identified in the image. A popular machine learning technique for classification is using neural networks, which are described in detail in section 2.1.1. Neural networks have two phases of operation. The first phase is the training phase. During training, the network processes labelled images, i.e. every image has a label indicating the class of the image. The goal of the training phase is to learn the network parameters so that the neural network can make accurate predictions of the labels of previously unseen images. The second phase starts after the training process is finished, and is called the inference phase. During this phase the network uses the knowledge it has acquired and utilises it to predict the labels of new images.

An important remark is that, when using machine learning the assumption is made that the samples used during training are similar to the ones seen during inference. A neural network cannot recognise an image of a horse if all images during training are of cats. As a result, machine learning cannot handle surprising events properly. For example, when all of a sudden something in the setup produces rotated images during inference, the network will continue making predictions about the labels of these rotated images. Either the network has seen rotated images during training and will continue making accurate predictions, or the network has not encountered rotated images and will most likely be inaccurate. In the latter case, it is impossible to detect this change in accuracy because it happens during inference where the true labels are no longer available. Therefore the network is no longer reliable. Adding a notion of surprise to neural networks would allow recovering from an unexpected situation. If rotated images were not present during training and suddenly appear during inference, this would be a surprising event. Being able to use this notion of surprise can then be used to retrain the network on rotated images to adapt to the new samples seen during inference. Retraining the network to adapt to the surprising situation is a way of doing online learning, as opposed to the

standard way of training once and then doing inference.

Currently, the most commonly used machine learning technique to prevent surprising events that will be classified inaccurately, is to anticipate all possible surprising events and already include them during training using data augmentation. When using data augmentation, all augmentations of the original image, such as a rotated, skewed, flipped image are also processed during training. This makes the network robust to any anticipated surprising situation (which are as a result no longer surprising). However, creating such a robust network results in a larger network than strictly necessary. This has as a downside that such a network requires a more extensive training process which needs more processing power, more memory and consumes more energy. A smaller network that is tailored to a specific task, on the other hand, has as advantages that it can do faster inference, needs less resources and consumes less energy. It has as downside that it is not robust to surprising situations. Adding a notion of surprise to such a smaller network can enable it to adapt to surprising situations through online learning, while keeping the advantages of being a smaller network.

While surprise is an intuitive concept and could prove to be useful in machine learning, one of the key problems is that it is hard to quantify. In information theory, Shannon's information content is sometimes described as surprise, and in Bayesian statistics, surprise is quantified by measuring the difference between posterior and prior beliefs. Yet neither of those metrics for surprise are suited to be used in the context of deep learning as they define surprise differently, as will be shown in section 2.2 . The goal of this work is to define and validate surprise measurements that can be used in the context of deep learning for image classification tasks.

This thesis starts with a technical background of deep learning and surprise in chapter 2. The proposed surprise formulas and the setup of the experiments are described in detail in chapter 3. Chapter 4 covers the result of the experiments. These results are evaluated and discussed further in chapter 5. Finally, this thesis is concluded in chapter 6.

Chapter 2

Technical background

2.1 Deep Learning

Artificial Intelligence (AI) is a field of computer science that aims to create machines that are capable of thinking and acting like humans. Today, AI is a thriving field and rapidly transforming our world with many practical applications such as autonomous vehicles, personal assistants on our smartphones, and so on. Many of the contributing factors to the remarkable progress made during the past decade can be attributed to advancements in machine learning [25].

Machine learning is a subfield of AI seeking to provide knowledge to computers not by explicitly programming them but through data. Machine learning enables computers to make decisions not by following fixed rules but through knowledge of the real world. The term machine learning was first used by Arthur Samuel back in 1959, he described the field as "Field of study that gives computers the ability to learn without being explicitly programmed" [39].

Machine learning algorithms can be divided into three categories: supervised learning, unsupervised learning and reinforcement learning [41]. In supervised learning, a rela-

tionship between input and output is modelled. Supervised learning also means that the desired outputs are provided to the machine during training. The aim is to learn a mapping from these inputs to the desired outputs so that the outputs of new unseen inputs can be predicted. Supervised learning can be further categorised into regression and classification. Regression tasks are those where the output is a real value such as price or distance, while classification tasks have a categorical output that can be put into classes, such as 'cat' or 'dog'. The machine learning problem tackled in this thesis is image classification which is a classification problem where the inputs are images.

In unsupervised learning, there is only the data, no supervisor is providing the desired output. Unsupervised learning tasks aim to find structure in the data. A significant portion of unsupervised algorithms are clustering algorithms; their goal is to find groupings or clusters in the input. Clustering has many use cases such as document clustering [7], data compression [40] and customer segmentation [47], which provides natural groups of customers that can be targeted differently.

The last category is reinforcement learning. The output in this category is a sequence of actions. Unlike supervised learning, there are no input-output pairs given during training. Instead, the machines are provided with a reward function to evaluate its performance. Reinforcement learning mimics how animals and humans learn, by trying out different things and rewarding what goes well and penalising what goes wrong.

A simple machine learning algorithm called linear regression is able to estimate housing prices [35]. Another simple algorithm is naive Bayes, which can filter spam emails from legitimate emails [38]. The performance of traditional techniques such as the ones just mentioned depends heavily on the representation of the data. For example, when linear regression is used to estimate housing prices, the house is not directly examined. Instead, certain pieces of information of the house are used, such as the surface of the living area, the neighbourhood, surface of the lot, etc. These make up the data representation or so-called features of the house. Linear regression learns how the housing prices relate to each of these features. The algorithm is heavily dependent on these chosen features, if only the number of bathrooms of a house were given, rather than all relevant measurable

features, linear regression would not be able to make accurate predictions.

For many years the main task in machine learning problems was to create the right features for a given problem. Many tasks can be solved by a relatively simple machine learning algorithm when given the right features. The difficulty lies however in knowing what features should be extracted. For example, suppose that we would like to do face recognition. We know how faces look like, they have amongst others, eyes, a nose and a mouth. We also know the geometric shape of the eyes, but sometimes people wear sunglasses or hair obscures part of the eyes, and so on. It is difficult to describe precisely what a face should look like in terms of raw pixel values. One approach to solve this problem is to use machine learning to discover the best representation of the data. These learned features often perform much better than manually designed representations. They also make it possible for AI systems to adapt to new tasks, without having to spend many hours manually engineering new features [17]. Learning algorithms are also able to learn these representations in minutes or up to days for complex tasks, where hand-designing representation for a complex task can require many efforts from an entire community of researchers.

Extracting high-level features from raw data can be very difficult, even for learning algorithms. Individual pixels in an image of an outdoor scene depend heavily on the shadows and light intensities. The viewing angle also changes the silhouette of objects. Therefore high-level representations need to have disentangled factors of variation. Many of those high-level features can only be extracted by having a sophisticated understanding of the data.

Deep learning does feature learning by creating features that are expressed in terms of other, more straightforward features. For example, the simplest representation of an image are the input pixels. It has been shown that the features in the first layer can represent the edges in the image and from those edges, the next features can represent things like corners and contours, and so on. Until finally the features can be used to identify entire objects [49].

Deep learning is a powerful tool for supervised learning. A deep neural network can represent increasingly complex functions by adding more layers and more units within a layer. Many tasks that are easy for a person to do but are hard to program explicitly and that consist of mapping an input vector to an output vector can be done with deep learning, given sufficiently large labelled datasets and sufficiently large models.

2.1.1 Neural Networks

Artificial neural networks are the basis for deep learning, and are inspired from the brain. But while they are inspired by neuroscience, the goal is not to understand the brain per se, but to create powerful machines that achieve good results in machine learning tasks. A biological brain consists of neurons and synapses. Neurons are the fundamental computational units of the brain. The human brain houses approximately 86 billion neurons, operating largely in parallel [3]. Those neurons are largely connected to other neurons, with each neuron having around 10^4 synapses, or connections to other neurons. In the brain, each neuron receives an input signal from its dendrites. All inputs from the dendrites are accumulated in the cell body and if they reach a certain threshold, a spike is sent through the axon where it outputs a signal. The axons are connected via synapses to dendrites of other neurons. This flow is something that is mimicked in artificial neural networks.

Multilayer perceptron

The perceptron, invented by Rosenblatt in 1958 [36], is the basic unit of a neural network. It has inputs $x_j \in \mathbb{R}, j = 1, ..., m$ and output y. Each input also has a corresponding weight, in the simplest case, the output is a weighted sum of the inputs,

$$y = \sum_{j=1}^{m} w_j x_j + w_0. \tag{2.1}$$

The extra w_0 weight is called the bias weight and is added to make the model more

general. It is modelled as the weight of an extra input x_0 which is always 1. The output y can also be written as a dot product,

$$y = \boldsymbol{w}^T \boldsymbol{x}, \tag{2.2}$$

with $\boldsymbol{w} = [w_0, w_1, ..., w_m]^T, \boldsymbol{x} = [1, x_1, ..., x_m]^T$. To solve a task with the perceptron, the weights w need to be adapted so that the correct outputs are calculated for the given inputs, how exactly this is done, is covered later.

The perceptron implements a linear discriminant function as equation (2.2) defines a hyperplane. As such it can separate two classes as a hyperplane divides the input space into two half-spaces. Class C_1 is chosen if $\boldsymbol{w}^T \boldsymbol{x} > 0$ and C_2 is chosen otherwise. It is important to remark that two classes can only be separated using a linear discriminant function if those classes are linearly separable.

When there are more than two classes, multiple perceptrons can be combined. For $K > 2$ outputs, there are K perceptrons with each of them having their own weight vector $\boldsymbol{w}_i$. The corresponding outputs y_i can be combined to an output vector $\boldsymbol{y}$ and calculated as,

$$\boldsymbol{y} = \boldsymbol{W} \boldsymbol{x}, \tag{2.3}$$

with $\boldsymbol{W}$ the $K \times (m+1)$ weight matrix where the rows correspond to the weight vectors $\boldsymbol{w}_i$ of the K perceptrons. To select a class in a classification task, the class C_i is chosen if $y_i = \max_k y_k$. Figure 2.1 shows K parallel perceptrons.

With a single layer of parallel perceptrons, only linear functions can be approximated. Other simple functions like for example, the XOR function, cannot be represented. One way to solve this is by using multilayer perceptrons (MLP) or feedforward neural networks. An MLP combines multiple layers of parallel perceptrons, where the inputs of intermediate layers, also known as hidden layers, are the outputs of the previous layer after a non-linear function is applied to them. The reason for this non-linearity is that linear combinations of linear combinations result in a linear combination, i.e. an MLP

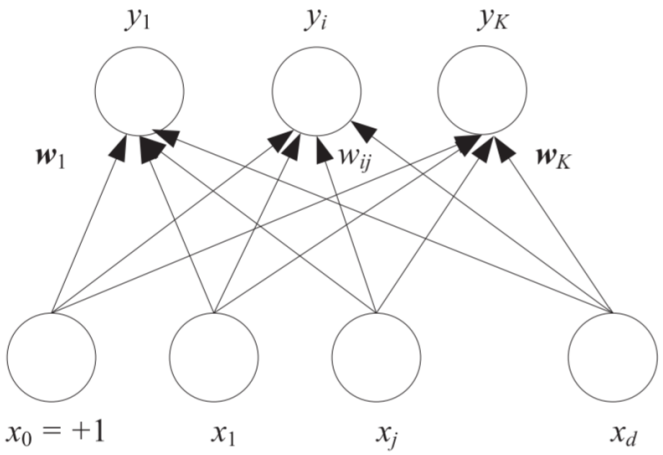


Figure 2.1: K parallel perceptrons with inputs x_i and outputs y_j. Each output is a weighted sum using weight w_ij to connect input x_i with output y_j. Figure taken from [2].

without a non-linearity can be simplified to a single layer perceptron and hence has the same expressive power. Including a non-linearity allows learning non-linear relationships between inputs and outputs. An often used non-linearity is the sigmoid function,

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{2.4}$$

and the hyperbolic tangent,

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.5}$$

These function are differentiable, continuous functions that mimic a threshold, e.g. the sigmoid function generates an output in the (0, 1) range. The non-linear function that is applied to the output of a perceptron is called the activation function.

Back-propagation

A multilayer perceptron can be used to approximate any function. The universal approximation theorem [9] states that any continuous function can be approximated by an MLP with at least one hidden layer. However, it does not state how the weights of such multilayer perceptrons are obtained. To go from a multilayer perceptron to a neural network and deep learning, the network must learn how to obtain the right weights for a task.

To learn the weights, a cost function is used, which is defined in terms of these weights. In supervised learning, cost functions are used to measure how good or bad the model is in terms of the model predictions $\hat{\boldsymbol{y}}$ compared to the true labels $\boldsymbol{y}$. Typically the cost function is expressed as the difference or distance between the predicted value and the actual value. The cost function is also often referred to as the loss or error function. The objective of machine learning is to find parameters that minimise this cost function. The exact cost function used is heavily problem dependent. Classification tasks use different cost functions than regression tasks, and for each of those category different cost functions can be used. For regression tasks, the most used cost function is the mean squared error (MSE)

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2. \tag{2.6}$$

While for classification the Cross-Entropy loss is commonly used

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right). \tag{2.7}$$

To minimise the cost functions in neural networks, an iterative process is used, called gradient descent. Gradient descent is an optimisation algorithm that finds the minimum in an iterative way. Minimising $f(\boldsymbol{x})$ can be done by taking a step in the direction in which f decreases the fastest. This can be done by using the gradient, as the negative gradient points directly downhill. As a result, f can be decreased by moving in the direction of the negative gradient. Gradient descent selects a new point by

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{2.8}$$

with ϵ the learning rate. The learning rate is a positive scalar that determines the size of the step taken. Choosing the right learning rate is an essential step in optimising neural networks. While a high learning rate can cover more distance in each step, it has a risk of overshooting the minimum as the gradient is constantly changing. A low learning rate is more precise but has as a downside that it is more compute intensive as it will take a longer time to get to the minimum.

In the standard gradient descent formula of equation 2.8, the cost function is evaluated over the entire training set. Having to compute the gradient over every training sample for each iteration in gradient descent is a rather compute-intensive task and not strictly necessary. A variant of standard gradient descent is stochastic gradient descent where the gradient is calculated over only one sample. This has the advantage that the gradient is much faster evaluated, but the downside is that the gradient can have a lot of noise as it is only computed over a single data sample. Therefore, the most common variant of gradient descent is a compromise of both. It uses minibatches to compute the gradient, this keeps the speed advantage over standard gradient descent while also being more stable than stochastic gradient descent.

If the cost function is $J(\boldsymbol{\theta})$, then we need to calculate the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. So the parameters can be updated using gradient descent by applying

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^{t} - \epsilon \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}. \tag{2.9}$$

In this formula θ denotes the parameters of the network. The primary parameters are w_{ij}^k, which is the weight between node i in layer $k-1$ and node j in layer k. In order to calculate the gradient of the weights with respect to the cost function, back-propagation is used. The back-propagation algorithm [37] is a way to compute the gradient by using the information from the cost function and flowing it backwards through the network. The backward part comes from the fact that the gradients of the weights of the final layer are being calculated first and the gradients of the first layer are calculated last. The back-propagation algorithm calculates the gradient by using the chain rule and the product rule. This also explains why the activation function had to be a continuous, differentiable function, as it will be used in the chain rule for derivatives.

The back-propagation algorithm is based on applying the chain rule to the error function partial derivative

$$\frac{\partial J}{\partial w_{ij}^k} = \frac{\partial J}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}, \tag{2.10}$$

where a_j^k is the activation of node j in the kth layer, before it is passed through the

non-linear activation function. The first term can also be written as

$$\delta_j^k \equiv \frac{\partial J}{\partial a_j^k}, \tag{2.11}$$

and is called the error term. The second term can be rewritten as the weighted sum of the outputs from the previous layer

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}\left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1}\right) = o_i^{k-1} \tag{2.12}$$

where o_i^k is the output of node i in layer k, i.e. after passing through the non-linear activation function and r_k is the number of nodes in layer k. Putting these expression together we can write equation (2.10) as

$$\frac{\partial J}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \tag{2.13}$$

This means that the partial derivative of a weight w_{ij}^k is the product of the error term of node j in layer k and the output of node i in layer $k-1$. The calculation of the error term δ_j^k is dependent on the error terms of the next layer and eventually on the error function J. Computing these error terms starts from the output layer down to the input layer.

Taking the MSE cost function as example, the cost function becomes $J = \frac{1}{2}(\hat{y} - y)^2$, where $\hat{y}$ is the output of the network and y is the true output. Without loss of generality a one output final layer is assumed ($j = 1$). J can then be written as $J = \frac{1}{2}\big(g_o(a_1^m) - y\big)^2$, where g_o is the activation function for the output layer. Through formula (2.11) and the product rule, the error term of the final layer becomes $\delta_1^m = (g_0(a_1^m) - y)\, g_o'(a_1^m) = (\hat{y} - y)\, g_o'(a_1^m)$. Plugging this into equation (2.13), the partial derivative with respect to the weights of the last layer becomes

$$\frac{\partial J}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = (\hat{y} - y)\, g_o'(a_1^m)\; o_i^{m-1}. \tag{2.14}$$

In the hidden layers, the partial derivative takes a bit more calculations. The error

term in the hidden layers depends on all the nodes in the next layer. The error term in these layers can be written as

$$\delta_j^k = \frac{\partial J}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}, \tag{2.15}$$

where r_{k+1} is the number of nodes in the next layer. The last term of this equation can be written as

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'\big(a_j^k\big), \tag{2.16}$$

where $g'\big(x\big)$, is the derivative of the activation function. Combining all of this together we get the back-propagation formula for the partial derivatives of the cost function J in terms of the weights

$$\frac{\partial J}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\big(a_j^k\big) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{2.17}$$

To summarise, during the forward phase of the network the output $\hat{y}$ is computed. In the backward phase the error between $\hat{y}$ and y is used to calculate the error terms and partial derivatives with respect to the weights in each layer in a backward flow. Once the backward phase is completed, the weights can be updated using gradient descent. This process iterates until a local minimum is found.

Rectified Linear Units

As described above, each layer adds a non-linearity through the activation function. The *sigmoid* and *tanh* functions have been the most used activation functions for many years but have lost popularity in recent years due to the vanishing gradient problem [20]. The sigmoid function outputs any value into the $(0, 1)$ range, this is useful as it can be used for representing probabilities. There is, however, an issue with the derivative. The derivative of the sigmoid function has a maximum of $1/4$ and a horizontal asymptote at 0. In back-propagation, gradients propagate through the network using the product rule, and since the derivatives of the activations are smaller than one, the final derivative becomes

smaller and smaller. In deep learning where there are many layers in the network, the first layers will have very small and inaccurate gradients. However, the first layers are the most important in the forward phase, as the following layers build on this. Inaccuracies in the first layer therefore corrupt the entire network. To solve this, an activation function with better derivative behaviour is required.

Rectified linear units (ReLUs) [32] use the activation function

$$g\big(z\big) = \max\big(0, z\big). \tag{2.18}$$

A ReLU is fast to evaluate and has a very simple derivative. The gradient is 0 across half of its domain and 1 in the linear part. This means that it does not suffer from the vanishing gradient problem. There is a drawback when using ReLUs though, they cannot learn from samples for which the activation is zero. One negative value will produce a 0 gradient in all following layers in back-propagation. There are variants of ReLUs that guarantee gradients everywhere such as leaky ReLUs [28] and Maxout units [18].

Overfitting and underfitting

The goal in machine learning is that the model must generalise well, i.e. perform well on previously unseen inputs, and not just on those on which the model was trained. Neural networks are optimised during training by using the cost function to minimise the training error. In machine learning we also want to minimise the test error. This is measured on a test set of samples that were collected separately from the training samples. To achieve a network that performs and generalises well it needs to have a small training error and a small gap between training and test errors. These two elements correspond to the concepts of underfitting and overfitting. When the model is not able to produce a sufficiently low training error, it is underfitting. Overfitting, on the other hand, happens when the test error is much larger than the training error.

Regularisation

The behaviour of neural networks is strongly affected by the number of nodes per layer and the number of layers. Neural networks can approximate increasingly complex functions by adding more nodes and more layers to the network. While more complex neural networks can approximate more complex functions, they are also very sensitive to overfitting. One way to overcome overfitting is to modify the cost function to include weight decay. Weight decay expresses a preference for weights with a small $L2$-norm. Taking the MSE cost function as an example, the new cost function with weight decay becomes

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^T \boldsymbol{w}, \tag{2.19}$$

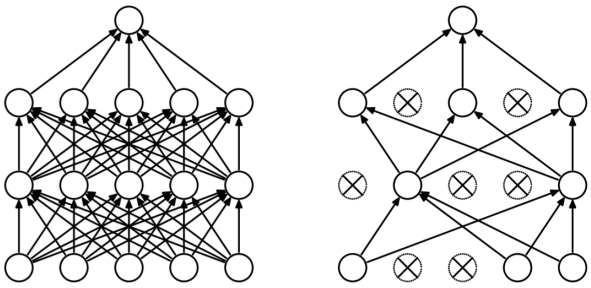
with λ is a hyperparameter that controls the strength of the weight decay.

More generally, any modification to the learning algorithm that aims to reduce overfitting is known as regularisation.

A recent powerful method of regularisation is dropout [44]. Dropout refers to dropping out nodes in a neural network. Dropping out a node means temporarily removing it from the network, including its incoming and outgoing connections. Which nodes to drop is chosen randomly. Each node is dropped with probability p, independent of the other nodes. Dropout can be seen as mimicking an ensemble of similar models. It is applied only during the training phase. After training, all nodes are used, but the activations are reduced by a factor p to account for the missing activations during training. Figure 2.2 illustrates dropout.

2.1.2 Convolutional Neural Networks

Convolutional neural networks [26], also known as CNNs, are a specific kind of neural network. CNNs work mainly on image data, where they exploit the two-dimensional structure of the data. This section describes the structure of a convolutional neural



(a) Standard Neural Net (b) After applying dropout.

Figure 2.2: Comparison of a standard neural network on the left and a neural network on which dropout is applied on the right. [44]

network and how it is used to do image classification.

Standard neural networks do not scale well when used on image data. Small images that are only of size $32 \times 32 \times 3$ (32 pixels wide, 32 pixels high, and 3 colour channels), require a large regular neural network. One neuron in the first hidden layer has $32 \times 32 \times 3 = 3072$ inputs and thus has 3073 learnable weights (this includes the bias weight). In deep learning where there are many layers and many neurons in each layer, the amount of learnable weights for image input data would quickly add up. This huge number of parameters would need a long training process and would quickly lead to overfitting. Convolutional neural networks exploit the fact that the inputs are images and thus have a certain 2D structure to reduce the number of parameters.

A convolutional neural network has three fundamental type of layers, namely convolutional layers, pooling layers, and fully-connected layers. The latter type is the layer described in regular neural networks. The full architecture of a CNN is shown in figure 2.3. The convolutional layer and pooling layer are described next.

Convolutional layer

In a convolutional layer the parameters consist of a set of filters. These filters are essentially small 2D patches, e.g. 3×3 or 5×5, that extend through the depth of the network. Every filter is convolved across the width and the height of the input. The computed value is the discrete convolution of the filter and the corresponding patch of the inputs. A

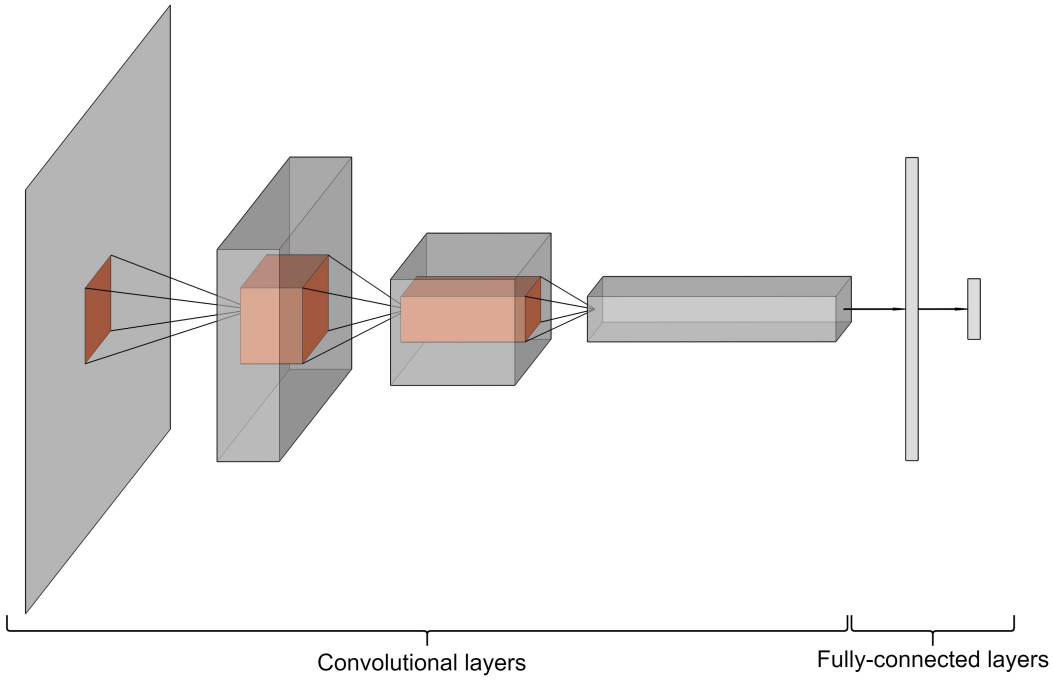


Figure 2.3: The architecture of a convolutional neural network.

discrete convolution is a linear transformation that preserves the 2D structure. Each filter slides over the input and calculates at every position the product between each element of the filter and the input element it overlaps with. The results at each position are then summed up to obtain the output value at that position. Sliding the filter across the entire image results in a 2D feature map. Repeating this process for multiple filters results in as many output channels as there are filters. If the input has multiple channels, the filter needs to be 3-dimensional and the output is obtained by element-wise summation of the resulting feature maps. An illustration of applying a convolutional filter over an input image is shown in figure 2.4.

It can be noticed that the output size of a convolutional layer is not only dependent on the filter size but also on the size of the inputs, this is different than in fully-connected layers where the amount of neurons determine the output size. Apart from the filter size and input size, three more hyperparameters control the size of the output layer: the depth, stride and zero-padding. The depth corresponds to the number of filters used in a convolutional layer as each filter corresponds with an output channel, the depth thus determines the amount of channels in the output. The stride in a convolutional layer

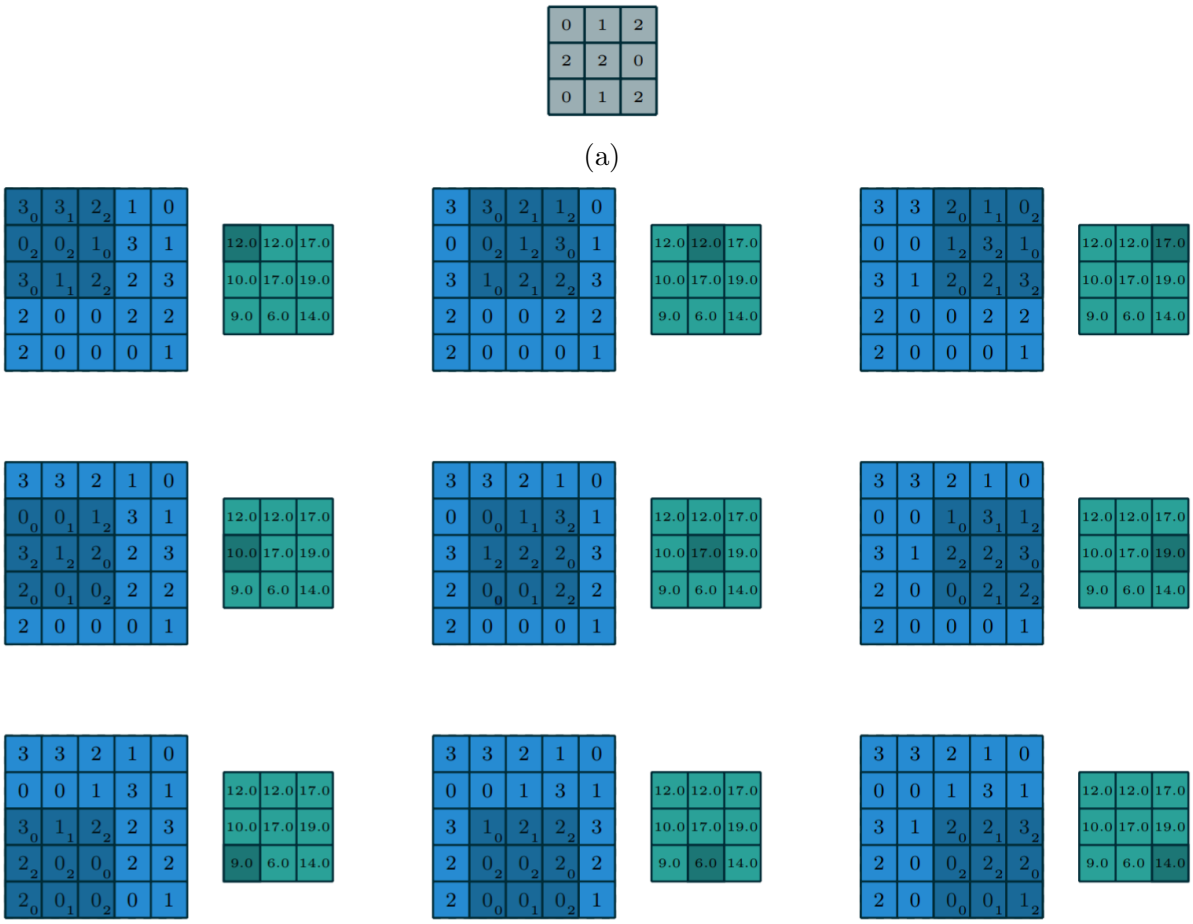


Figure 2.4: Illustration of the computations in a convolutional layer. (a) shows the filter used in (b) to compute the discrete convolution over [11].

controls how the filter slides over the input. With a stride of one, the filter is moved one pixel at a time. With a stride N the filter jumps N pixels at a time when sliding over the input. Higher strides result in a lower output size. Zero-padding is a feature that pads the input with zeros around the border. The size of the zero-padding is a hyperparameter that allows changing the output dimensions. This can be helpful when using strides higher than 1, to adjust the input so that the strides can fit in the input. The final output size of a convolutional layer with input of size $W_1 \times H_1 \times C_1$, K kernels of size $F \times F$, a stride S and zero-padding P is $W_2 \times H_2 \times C_2$. Where $W_2 = (W_1 - F + 2P)/S + 1$, $H_2 = (H_1 - F + 2P)/S + 1$ and $D_2 = K$.

The amount of learnable parameters is independent of the size of the image. A convolutional layer with K filters of size $F \times F$ has $K \times F \times F + 1$ weights, this includes a

bias weight that is added to the result of the convolution. Training convolutional layers happens in the same way as fully-connected layers, namely using back-propagation.

Pooling layer

Pooling layers are an optional but often encountered layer in convolutional neural networks. Pooling layers have as purpose to reduce the dimensions of the output of a convolutional layer. A pooling function operates on the output of a convolutional layer and creates new outputs by summarising the outputs of a small patch. The most used pooling function is max pooling [50], which takes the maximum output within a rectangular patch. The most common form of a max pooling layer uses a 2×2 filter, applied with a stride of 2. This downsamples the output of a convolutional layer by a factor of 2 in both the width and the height, the amount of channels remains untouched. Figure 2.5 illustrates max pooling with a 2×2 filter and stride 2. Pooling layers are not always used in CNNs, many researchers prefer using architectures that only consists of convolutional layers [43]. They use larger strides in the convolutional layer to reduce the output instead of an additional pooling layer. It has been shown that not having pooling layers is important in generative models, such as deep convolutional generative adversarial networks (DCGANs) [34].

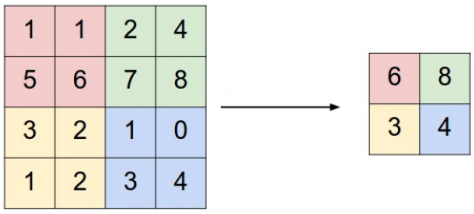


Figure 2.5: Illustration of max pooling. [22]

2.1.3 Autoencoders

An autoencoder [19] is a special kind of neural network that is trained to reconstruct a copy of its input to its output. An autoencoder takes an input $\boldsymbol{x}$ and internally maps it to the latent representation $\boldsymbol{h}$ that describes a 'code' used to represent the input. An autoencoder has two parts: an encoder that produces $\boldsymbol{h} = f_\theta(\boldsymbol{x})$ and a decoder that

produces a reconstruction $y = g_{\theta'}(\boldsymbol{h})$, with θ and θ' the parameters of the respective parts of the network. The architecture of an autoencoder is shown in figure 2.6. The latent space of the autoencoder is designed so that it is unable to learn to copy the inputs perfectly. This way the model is forced to select which aspects of the input should be copied to the latent representation, and as a result learns useful properties of the data. The parameters are optimised, by minimising the reconstruction loss between $\boldsymbol{x}$ and $\boldsymbol{y}$. This makes autoencoders an unsupervised learning technique as they do not need any additional labels for the data.

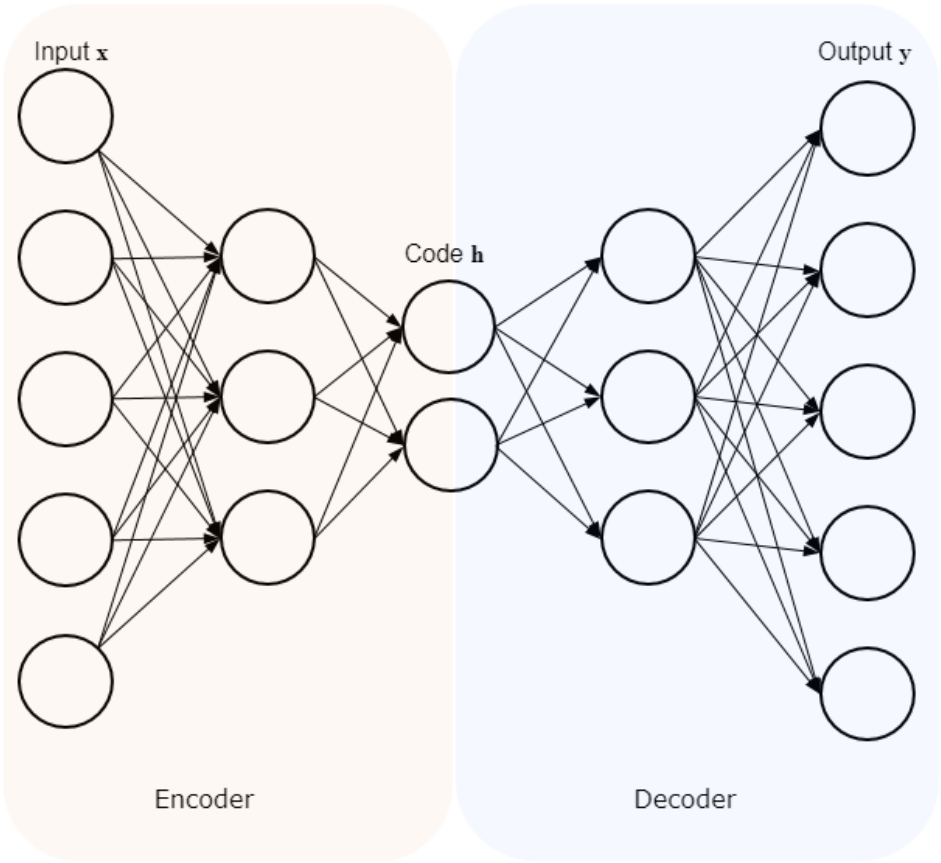


Figure 2.6: The general structure of an autoencoder, mapping an input $\boldsymbol{x}$ to the output $\boldsymbol{h}$ through the latent space representation $\boldsymbol{h}$.

Denoising autoencoders

One way to force an autoencoder to learn a useful representation in the latent space is by changing the reconstruction term of the cost function. A traditional autoencoder

minimises a cost function $J(\boldsymbol{x}, g(f(\boldsymbol{x})))$, penalising high reconstruction losses. A result of this cost function is that the network will try to learn the identity function.

A denoising autoencoder instead minimises $J(\boldsymbol{x}, g(f(\hat{\boldsymbol{x}})))$, where $\hat{\boldsymbol{x}}$ is a corrupted version of $\boldsymbol{x}$ to which some form of noise is added. The autoencoder now has to reconstruct the clean input from a partially distorted one. As a result denoising training forces f and g to learn implicitly the structure of $p_{data}(\boldsymbol{x})$ [1, 6].

Convolutional autoencoder

Standard autoencoders and denoising autoencoders use fully-connected layers, and thus ignore the 2D structure of images. The convolutional autoencoder (CAE) [29] uses convolutional layers that can exploit the 2D structure of the data. The CAE architecture is intuitively similar to the one of a standard autoencoder. In the encoder part of the network, it replaces the fully-connected layers with convolutional layers. For the decoder part, the network needs to go from the smaller latent space back to the original input dimensions. The decoder requires an operation that also respects the 2D structure but can expand in dimension. Transpose convolutions, also known as fractionally strided convolutions or deconvolutions (although it is not an actual deconvolution operation), can be seen as the gradient of a convolution with respect to its input. The simplest way to reason about a transposed convolution is to view the input of a transpose convolution as the output of a regular convolution on some initial feature map. The transposed convolution can then be seen as the operation that reverses this back to the same shape as the initial feature map. However, it does not recover the original inputs as a transpose convolution is not the reverse of a convolution. A transpose convolution uses the normal convolution operation but uses zero-padding and strides to obtain the desired output shape.

2.2 Surprise

In psychology, it is widely agreed that surprise is an emotion emerging from the mismatch between expectations and what is actually experienced or observed [14]. In this thesis our concern is not with the emotion of surprise but rather with the quantification of surprise. In neuroscience it has been recognised that the amplitude of the P300 component of event-related brain potentials reflects the degree of surprise [10]. P300 is an event-related potential (ERP), which is measured by electroencephalography (EEG). Studies have linked P300 with the processing of unexpected events [33], P300 has also been used as neural surprise marker in odd-ball experiments [12]. However, outside the context of neuroscience, surprise has no standard metric to be quantified with. In what follows, several alternative definitions are introduced.

Kullback-Leibler divergence

The Kullback-Leibler divergence (D_{KL}) plays a role in both the Bayesian surprise formula (2.23) and the confidence-corrected surprise (2.25), which are discussed later. This is not unexpected as the Kullback-Leibler divergence on its own can also be seen as a measure of surprise. The D_{KL} has its origin in information theory. The Kullback-Leibler divergence [24] is defined as:

$$D_{KL}(P||Q) = -\sum_i P(i) \log \frac{Q(i)}{P(i)}. \tag{2.20}$$

It measures how one probability distribution differs from the expected probability distribution. $D_{KL}(P||Q)$ is the divergence from Q to P, and it measures the information gained when changing the prior distribution Q to the posterior distribution P. If applicable, P represents the true distribution of the data while Q represents a model or approximation of P. The D_{KL} is defined only if for all i, $Q(i) = 0$ implies $P(i) = 0$. An important property is that the Kullback-Leibler divergence is additive for independent distributions. If P_1 and P_2 are independent distributions with joint distribution P and likewise for Q_1, Q_2 and Q, then $D_{KL}(P||Q) = D_{KL}(P_1||Q_1) + D_{KL}(P_2||Q_2)$.

Shannon Surprise

In information theory, Shannon's information content [13] or self-information

$$\mathrm{I}(X) = -\ln p(X), \tag{2.21}$$

is often associated with surprise, and therefore also sometimes called surprisal. The information content of a sample X measures the amount of information that is revealed when observing X. This metric of surprise will further be referenced as Shannon surprise. Unlikely events have a high amount of information. Conversely, when a common event is observed, a small amount of information is associated with this event. A simple example is the case of a coin flip. If it is a fair coin, both heads and tails will have a probability of 0.5. This gives a self-information of 1 bit for both events if we use log_2. If the coin is biased and has a probability of 0.75 to be heads, the self-information of a sample coin toss that lands heads would be 0.415 bits while the self-information of the less common event of landing tails would be 2.

Bayesian Surprise

Another way to look at surprise comes from Bayesian theory. In the Bayesian framework models or hypotheses are associated with confidence or belief. Bayesian surprise is then defined as the changes for a given observer that take place when going from prior to posterior distributions of the data [4]. When considering a set of possible models $\mathcal{M}$, an observer has a prior distribution $P(M)$. Observing new data D leads to reevaluating the beliefs and changes the prior probability into a posterior probability. Bayes theorem states that

$$P(M|D) = \frac{P(D|M)}{P(D)}P(M). \tag{2.22}$$

From this formula it can be seen that the effect of D is that it can be used to change $P(M)$ to $P(M|D)$. The surprise of data D is then defined as the Kullback-Leibler divergence

between the prior and the posterior

$$S(D, \mathcal{M}) = D_{KL}(P(M)||P(M|D)). \tag{2.23}$$

The alternative version $S(D, \mathcal{M}) = D_{KL}(P(M|D)||P(M))$ is also used and may be even preferred if the "best" distribution is used as first argument. This metric gives a higher surprise for data samples that cause a larger change in belief.

Confidence-corrected surprise

The Shannon surprise and Bayesian surprise formulas are distinct formulas to capture surprise yet they can be seen as complementary. Shannon surprise captures the unexpectedness in sample X given a model, while Bayesian surprise is about the model as its belief changes. Another difference is that Bayesian surprise is calculated only after the belief is changed while Shannon surprise is immediately available upon observation of the sample.

Confidence-corrected surprise [15] is a formula that tries to combine both surprise measures by using their complementary benefits and overcoming their individual shortcomings. First, it replaces Shannon surprise by a weighted average over all possible model parameters θ. The weight is determined by the current belief $\pi_0(\theta)$. This gives the raw surprise

$$S_{raw}(X; \pi_0) = -\int_{\Theta} \pi_0(\theta) \ln p(X|\theta) d\theta. \tag{2.24}$$

This is also equal to the sum of the Shannon and Bayesian surprise: $S_{raw}(X; \pi_0) = Surprise_{Shannon}(X; \pi_0) + Surprise_{Bayes}(X; \pi_0)$. Confidence corrected surprise extends this by also including the confidence in his belief, in the formula. This ensures that data samples that occur with a low probability when there is uncertainty about the world, because not enough is learned about the world yet, are less surprising than samples that have a low probability when there is more certainty about the world model. This confidence is represented by the entropy of the current belief about the model parameters $\mathcal{H}(\pi_0)$. Confidence corrected surprise is then derived from subtracting the entropy from

the raw surprise, where $p(X|\theta)$ is normalised to the scaled likelihood $\hat{p}_x(\theta)$.

The confidence-corrected surprise then becomes:

$$S_{corr}(X;\pi_0) = \int_\Theta \pi_0(\theta) \ln \frac{\pi_0(\theta)}{\hat{p}_x(\theta)} d\theta = D_{KL}(\pi_0(\theta)||\hat{p}_X(\theta)). \tag{2.25}$$

Chapter 3

Setup

3.1 Defining and quantifying surprise in deep learning

In the previous chapter it was mentioned that in psychology, surprise is defined as an emotion emerging from the mismatch between expectations and what is actually experienced or observed. In the context of deep learning, surprise is a metric that captures a persistent mismatch between the expected characteristics of the input data and the characteristics of the observed data. In this work we define a neural network as being surprised when an unexpected event occurs that affects the stream of input data. This means that a surprising event does not just influence a single data sample but rather a sequence of samples following the event. In this work we define unexpected and thus surprising events as events that are unlikely to happen according to the characteristics of the data that the model learned during training. As a consequence, surprising events often result in inaccurate predictions of the model. Concretely we will work with image classification tasks where the unexpected event is rotating all inputs with a fixed angle between $0°$ and $90°$, and where the model has not seen any rotated inputs during training.

As this definition of surprise captures the impact of a persistent event such as suddenly rotating all the input images, surprise is a metric that is measured over a sequence of samples. However, sometimes it can be useful to measure the mismatch between a single input sample and the expected characteristics of the input data. Capturing such 'surprise' of a single sample is covered by the field of novelty and outlier or anomaly detection, but this is not what is regarded as surprise in this work. As mentioned in the introduction, surprise and novelty are often interchanged concepts but are in fact very different [5]. Outlier and novelty detection are widely studied topics in machine learning [21, 48].

Section 2.2 has already covered different formulas of surprise. To recapitulate, Shannon surprise captures the self-information of an observed data sample, Bayesian surprise captures the change in belief of an observer about the distribution of the data after observing a data sample, and confidence-corrected surprise is a combination of both. While those formulas have proven useful in some cases, they are not suited for surprise as it is defined here, as they work on single data samples or assume a set of models of the world. A new formula is needed to capture the surprise in the context of deep learning.

3.1.1 Reconstruction accuracy surprise

The first proposed method to measure the surprise of a neural network is reconstruction accuracy surprise ($Surprise_{RA}$). The main idea behind this method is that autoencoders are only able to reconstruct images that are similar to the ones seen during the training phase of the autoencoder, which is a characteristic that can be used to measure surprise. Consider a typical autoencoder trained on input images of dogs, the autoencoder will easily reconstruct images of other dogs but will fail to reconstruct the input image when it is given an image of a car, assuming that the latent space of the autoencoder is small enough, and lossless reconstruction is impossible.

The typical deep learning architecture of an image classifier has been shown in figure 2.3. This architecture consists of several convolutional layers, optionally followed by a few fully-connected layers. This neural network can be trained using the standard techniques,

such as gradient descent and regularisation, see section 2.1.1.

In this method, after the training phase of the classifier is done, this architecture is expanded into an autoencoder architecture by adding decoder layers directly to the output layer of the classifier, or to one of the last hidden layers of the classifier. This creates an autoencoder where the encoder part consists of the trained classifier. Whether the decoder is appended to the last layer or a hidden layer is task dependent. For example, when only a few classes appear in the output layer, it is recommended to use the second to last or an even earlier layer, so that a latent space with a higher dimension can be used. An important requirement however is that this latent space is small enough so that nearly lossless reconstruction cannot happen, as a lossless autoencoder will be able to reconstruct any input, regardless whether it is similar to input data seen during training or not. Figure 3.1 shows the architecture used to measure $Surprise_{RA}$.

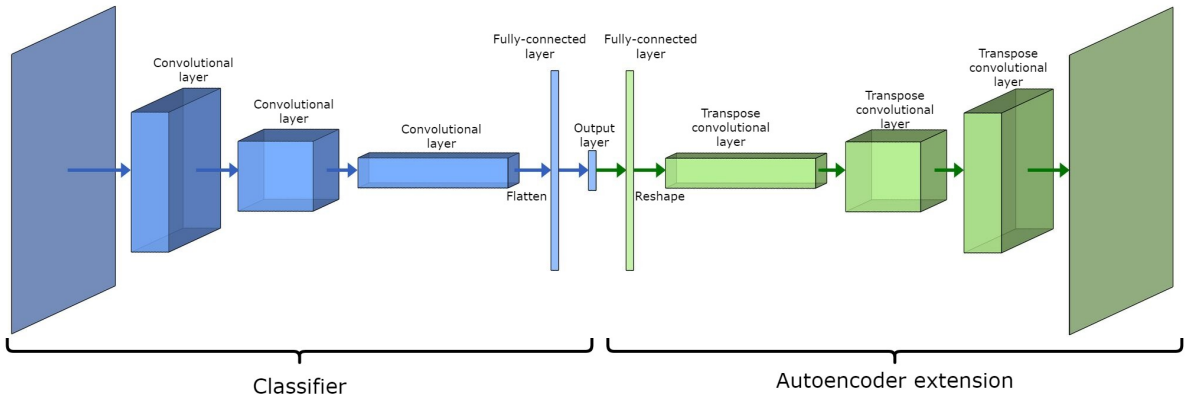


Figure 3.1: The architecture to capture the $Surprise_{RA}$ of an image classifier. The left part of the architecture corresponds to the original classifier, and the right part is the extension that turns the classifier into an autoencoder.

After the network is extended, it is retrained as an autoencoder. However, it is not trained in the default way autoencoders are trained. The layers that are originally part of the classifier keep their weights fixed and only the weights of the decoder layers are trained. This is done because the primary goal is to classify images accurately, adding mechanisms to capture surprise should in no way limit this performance. It is important that the training of the decoder part happens on the same data that has been used to train the classifier part. When the layers of the decoder part closely resemble the reverse

of the classifier part, i.e. a similar amount of layers, a similar amount of nodes in the fully-connected layers and a similar amount of filters in the transpose convolutional layers, we advise to train the autoencoder for a similar number of epochs so that the autoencoder training phase closely mirrors the classifier training phase. Adding this decoder part to the full network setup means that training time and learnable parameters approximately double.

To measure the reconstruction accuracy surprise, each data sample $\boldsymbol{x}$ is first passed through the classifier part of the network, resulting in output $\boldsymbol{p} = [p_1, ..., p_k]$ where p_i is the probability that the input belongs to class C_i of the k possible classes. Next, the data sample $\boldsymbol{x}$ is fed through the entire architecture including the decoder to reconstruct image $\boldsymbol{x}^*$, this second pass can continue from the activations of the last common part of the full classifier and the autoencoder. The reconstructed data sample $\boldsymbol{x}^*$, is then again passed through the classifier network, resulting in output $\boldsymbol{p}^*$. The reconstruction accuracy divergence (D_{RA}) of a single data sample is then calculated as

$$D_{RA} = D_{KL}(\boldsymbol{p}||\boldsymbol{p}^*). \tag{3.1}$$

As surprise in the context of this work is not applied to a single sample, this metric is averaged over a sequence or batch of samples in order to come to the reconstruction accuracy surprise of the batch. The reconstruction accuracy surprise is thus defined as,

$$Surprise_{RA} = \sum_i D_{KL}(\boldsymbol{p_i}||\boldsymbol{p_i}^*). \tag{3.2}$$

3.1.2 Hidden activations surprise

An alternative method for surprise in the context of deep learning is the hidden activations surprise ($Surprise_{HA}$). This formula does not use an autoencoder to capture surprise but is based on the activations of the last hidden layer. As in the previous formula, the classifier is first trained in the standard way. However, part of the training set is not used during training and kept separately. This unused part is then fed through the network and

the outputs of the last hidden layer are recorded. These outputs are then used to estimate the probability density function of the output values for each node in the last hidden layer. Let $P(h_1, ..., h_n)$ be the joint probability of the outputs of the last hidden layer, with h_i the output value of node i. After training, during inference, the same methodology can be applied to a set of new samples, to again estimate the joint probability of the outputs of the last hidden layer, let this new estimate be $P^*(h_1, ...h_n)$. This estimate will be very similar to P if the samples of this new set are similar to the ones used for estimating P, but will likely be different if the samples in the new set capture a surprising event. The hidden activations surprise formula measures surprise as

$$Surprise_{HA} = D_{KL}(P(h_1, ..., h_n)||P^*(h_1, ..., h_n)). \tag{3.3}$$

However, as the last hidden layer can have hundreds to thousands of nodes, approximating the joint probability becomes infeasible due to the curse of dimensionality. This problem is solved by making the naive Bayes assumption that all probabilities $P_1(h_1), ..., P_2(h_2)$ are independent. Using the property that the Kullback-Leibler is additive for independent distributions then gives

$$Surprise_{HA} = \sum_i D_{KL}(P_i(h_i)||P_i^*(h_i)). \tag{3.4}$$

The observation that not every hidden node influences the final output equally allows reducing the complexity even more. Therefore only a subset of the most activated nodes is used. In this context, a node is activated when it outputs a value greater than 0 when using a ReLU activation function.

3.1.3 Reconstruction loss surprise

The third and last surprise formula tested in this thesis is the reconstruction loss surprise ($Surprise_{RL}$). It combines elements from both the reconstruction accuracy surprise method and the hidden activations surprise method. The model architecture is the same

as used for $Surprise_{RA}$, as was shown in figure 3.1. Training happens in a similar fashion as well. First, the classifier is trained, followed by the autoencoder. The weights of the classifier stay fixed during this last training stage. However training is not done on the full training set, a subset is kept separately just as with hidden activations surprise method. After this two-stage training process is done, the part of the training set that was kept separately is fed through the entire architecture and the reconstruction loss of the autoencoder is recorded. The reconstruction losses of the samples are then used to estimate the probability distribution of the reconstruction loss. Let P_l be the probability density function of the reconstruction loss, approximated during training. After training, the same probability density function can be estimated using new sequences of data samples. Let this new estimate be P_l^*. This estimate will again be very similar to P_l if the set used for the new estimate is similar to the subset used during training. The reconstruction loss surprise is calculated as

$$Surprise_{RL} = D_{KL}(P_l||P_l^*). \tag{3.5}$$

3.2 MNIST dataset: handwritten digit classification

The dataset used in all experiments is the MNIST (Modified National Institute of Standards and Technology database) dataset [27]. This dataset contains samples of handwritten digits. Each sample is a greyscale image of 28 by 28 pixels. The handwritten digits are centred in the samples. The MNIST dataset contains 60,000 training samples and 10,000 test samples.

Figure 3.2: Samples from the MNIST dataset.

MNIST is a widely used dataset to test new image classification techniques, it was first used by LeCun to demonstrate convolutional neural networks [26]. Image classification on MNIST is considered to be a relatively simple task to solve and has been widely studied. It is relatively straightforward to gain an accuracy above 99% with convolutional neural networks, with some models going up to 99.77% accuracy [8].

This dataset was chosen for this thesis as surprise in the context of deep learning is a new concept for image classification. Using the MNIST dataset allows us to evaluate surprise on a well-understood dataset, so that the influence of surprise on the results can be well evaluated.

3.3 Model architecture

The architecture used for the experiments has to meet several requirements. The main goal of the network is to perform well on image classification tasks, therefore it should achieve at least 99% accuracy on MNIST. Furthermore, the architecture should be suitable to be extended into an autoencoder. Many architectures have been tested, the final architecture, including the autoencoder extension, is illustrated in figure 3.3.

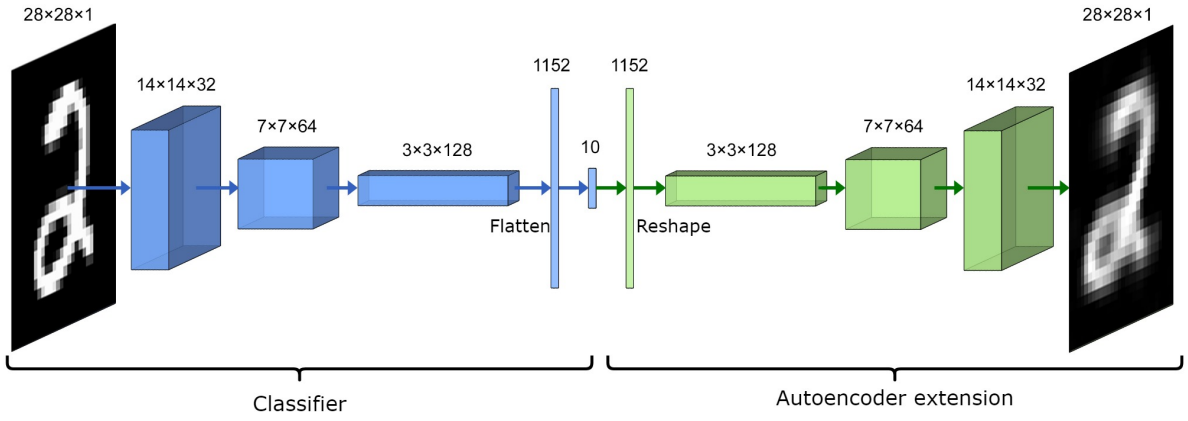


Figure 3.3: Architecture used in the experiments. The blue part shows the classifier, while the green part shows the autoencoder extension that is used in reconstruction accuracy surprise and in reconstruction loss surprise.

The classifier consists of three convolutional layers with respectively 32, 64 and 128

filters. The last convolutional layer is then flattened and used as input of a fully connected layer on which the output appears. A softmax function is then applied to the output, in order to obtain probabilities for each output class. The autoencoder extension used in reconstruction accuracy surprise and reconstruction loss surprise is applied directly to the output layer of the classifier, but skipping the softmax function. The 10 outputs thus become the inputs of a fully-connected layer with 1152 outputs. These outputs are reshaped into 128 patches of 3×3 to which several transpose convolutional layers are applied in order obtain to an output image of 28×28. The full architecture is summarised in table 3.1.

Table 3.1: Architecture summary. The layers marked in grey are only used during the classifier part, and are disabled when the model is extended to an autoencoder.

Layer	Output shape	Parameters
Input	$28 \times 28 \times 1$	
Conv2d kernel size 5, stride 2	$14 \times 14 \times 32$	832
ReLU	$14 \times 14 \times 32$	
BatchNorm	$14 \times 14 \times 32$	
Conv2d kernel size 5, stride 2	$7 \times 7 \times 64$	51,264
ReLU	$7 \times 7 \times 64$	
Conv2d kernel size 3, stride 2	$3 \times 3 \times 128$	73,856
ReLU	$3 \times 3 \times 128$	
Flatten	1152	
Dropout, p=0.5	1152	
Linear	10	11,530
Softmax	10	
TOTAL classifier		**137,482**
Linear	1152	12,672
Reshape	$3 \times 3 \times 128$	
ConvTranspose2d kernel size 3, stride 2	$7 \times 7 \times 64$	73,856
ReLU	$7 \times 7 \times 64$	
ConvTranspose2d kernel size 5, stride 2	$14 \times 14 \times 32$	51,264
ReLU	$14 \times 14 \times 32$	
ConvTranspose2d kernel size 3, stride 2	$28 \times 28 \times 1$	832
TOTAL		**276,106**

Chapter 4

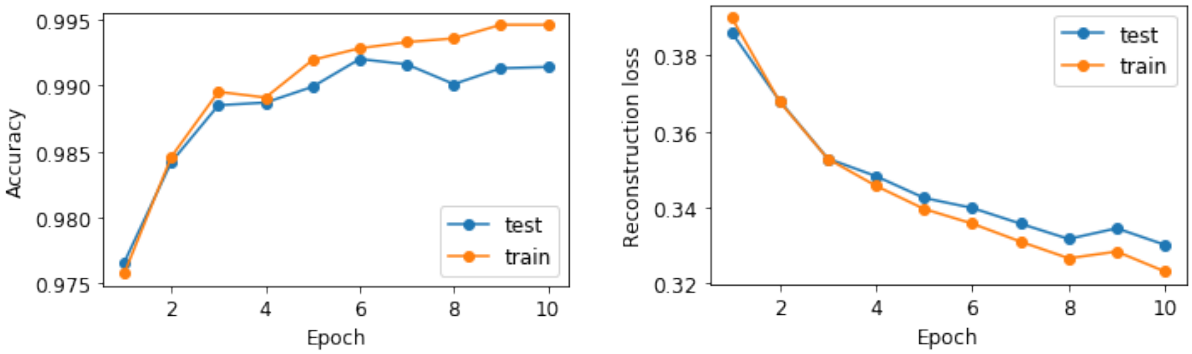
Results

4.1 Baseline

This section describes the classification performance as well as the behaviour after inducing an unexpected event of the architecture as described in section 3.3. For the training phase of the classifier, the negative log-likelihood loss is minimised, using stochastic gradient descent with hyperparameters, 0.01 for the learning rate, 0.9 for momentum and 10^{-3} for weight decay. Training the autoencoder part of the network happens by minimising the MSE loss using the Adam optimiser [23] with a learning rate of 0.001, betas (β_1, β_2) of (0.9, 0.999) and a weight decay of 10^{-5}.

The classifier achieves an accuracy of 99.11% on the test set after 10 epochs. This puts it well above the 99% accuracy goal which was set in the previous chapter. By training for more than 10 epochs, it is possible to get the accuracy slightly higher. But as the goal is not to achieve the highest possible accuracy on MNIST, 10 epochs is a good trade-off between training time and accuracy. Next, the autoencoder is also trained for 10 epochs, while keeping the learned weights of the classifier layers fixed. 10 epochs are chosen so that the decoder part is trained similarly to the classifier part, as the decoder has a similar amount of learnable parameters compared to the classifier. The network

achieves a reconstruction error of 0.3299, which is a good performance as can be seen in figure 4.3a. The validation curves of the classifier and autoencoder are shown in figure 4.1



(a) (b)

Figure 4.1: The validation curves of the training phase of the network. On the left the validation curve of the classifier part, on the right the validation curve of the autoencoder part.

What is important for both the reconstruction accuracy surprise and the reconstruction loss surprise is that the autoencoder is only able to reconstruct images that are similar to the ones seen during training and that it cannot reconstruct surprising images with the same precision. This is tested by rotating the test set over different angles and measuring the reconstruction loss. The loss increases as the angle of the rotation goes from 0° to 90° anticlockwise, which means that the autoencoder is performing worse on 'surprising' samples. Figure 4.2 shows the average reconstruction loss as the MNIST test set gets rotated from 0 to 90 degrees, it also shows that the classification error goes up to 0.82 for a rotation of 90°.

Visually inspecting the reconstructed images makes it clear that the handwritten digits are still recognisable in the unaltered test set, but no longer in the rotated variant as is shown in figure 4.3.

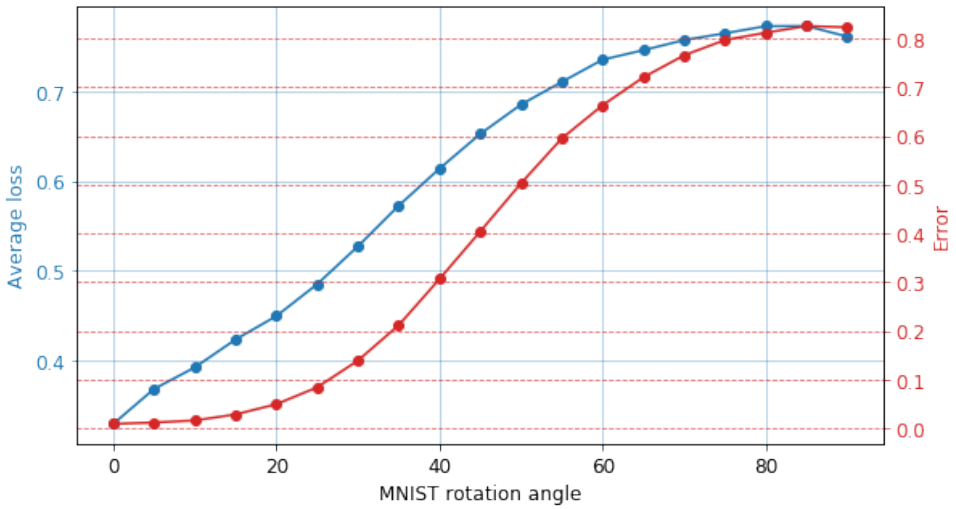
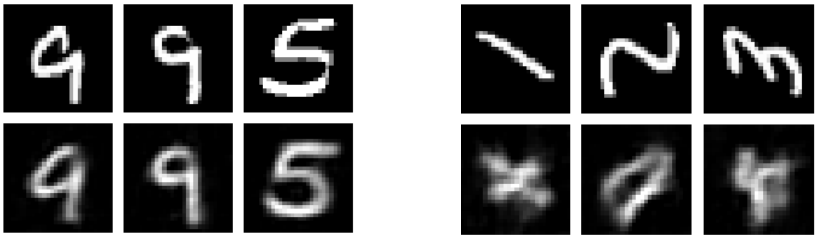


Figure 4.2: Average reconstruction loss and classification error for the test set as it gets rotated from 0 to 90 degrees.



(a) (b)

Figure 4.3: Impact of rotation on the autoencoder. On the left, the unaltered test set, on the right the rotated test set. The input images are shown on the top row, the reconstructions on the bottom row.

4.1.1 Rotation invariant network

The current way of dealing with possible surprising events such as rotated images, is to apply data augmentation during training. This means that during training, rotations are also added to the inputs. This way, whenever the surprising event of rotated input images happens, the network can still correctly classify them as it has seen these kinds of images during training. However, training a network that can accurately classify the MNIST dataset with random rotations between 0 and 90 degrees requires a bigger network architecture. Achieving a 99% accuracy on this augmented dataset is no longer simple. There are networks that achieve 99% accuracy on rotated MNIST by using rotation-

invariant convolutional filters [46] but such architectures are not relevant to try in this context as rotation is just one example of surprise. Data augmentation on the other hand, is a technique that can adapt the network to any kind of surprising event, as long as this surprising event can be anticipated during training. We adapted our baseline model to achieve the highest possible accuracy on MNIST with random rotations between 0 and 90 degrees. The resulting network is summarised in table 4.1, it has a total of 1,689,594 parameters. This is more than ten times the amount of parameters than the classifier part of the baseline and more than five times compared to the extended baseline. In addition, it is also trained for 40 epochs instead of just 10 as more data needs to be processed and more parameters need to be learned.

Table 4.1: Robust architecture summary. Architecture is similar to the baseline, only the relevant layers and the amount of learnable parameters are shown.

Layer	Output shape	Parameters
Input	$28 \times 28 \times 1$	
Conv2d	$14 \times 14 \times 56$	1,456
Conv2d	$7 \times 7 \times 128$	179,328
Conv2d	$3 \times 3 \times 256$	295,168
Flatten	2304	
Linear	512	1,180,160
Linear	64	32,832
Linear	10	650
TOTAL		**1,689,594**

The resulting average accuracy over all rotations is 98.95%, i.e. just below the envisaged goal of 99% accuracy. The accuracy for fixed rotations ranges from 98.13% to 99.21%. Figure 4.4 shows the accuracy for fixed rotations of the test set as well as the average over all rotations between 0° and 90°.

4.2 Reconstruction accuracy surprise

The reconstruction accuracy divergence compares the output class probabilities $\boldsymbol{p}$ of sample image $\boldsymbol{x}$ to the output probabilities $\boldsymbol{p}^*$ of the reconstructed image $\boldsymbol{x}^*$. The reconstruction accuracy surprise is measured by taking the average reconstruction accuracy

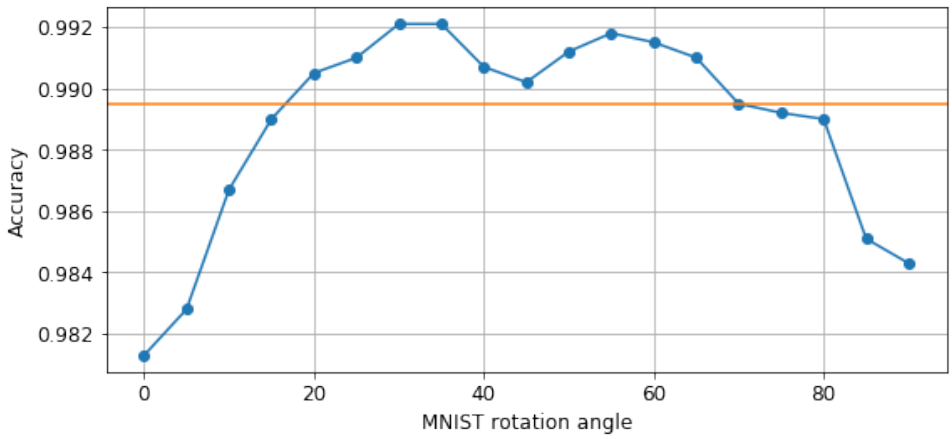


Figure 4.4: Accuracy for the MNIST test set as it is rotated. The orange line shows the average over all rotations.

divergence over a sequence of samples during inference. $Surprise_{RA}$ thus depends on the autoencoder extension and the ability of the network to make predictions on the reconstructed inputs. As mentioned in section 4.1, the average reconstruction loss is 0.3299. The classification accuracy of the reconstructed inputs of the test set is 94.22%. This means that the network does a reasonable job at making accurate predictions of the reconstructed images. Calculating the reconstruction accuracy surprise on the test set gives $Surprise_{RA} = 0.1307$. This value is the surprise for the regular, non-rotated test set, which should not be surprising to the network. The surprise is then calculated over rotated variants of the test set. Starting from a rotation of 5° which should be only a little surprising up to a rotation of 90°, which should be perceived as very surprising as the accuracy on this rotation is only 18%. The $Surprise_{RA}$ values range from 0.2 for a rotation of 5° to 1.9458 for a rotation of 90°. Figure 4.5 shows the development of the $Surprise_{RA}$ value compared to the classification error. The surprise metric shows that for the least surprising event, i.e. a rotation of 5°, the surprise goes up by 53% while the error goes up 32%. Over the course of all rotations, the surprise metric seems to have similar characteristics compared to the classification error. However, measuring surprise does not require labelled samples while measuring the classification error does.

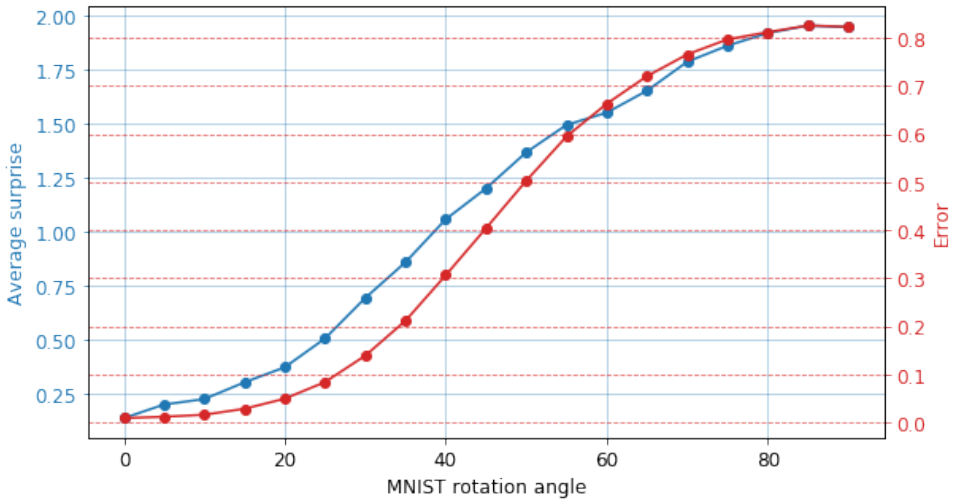
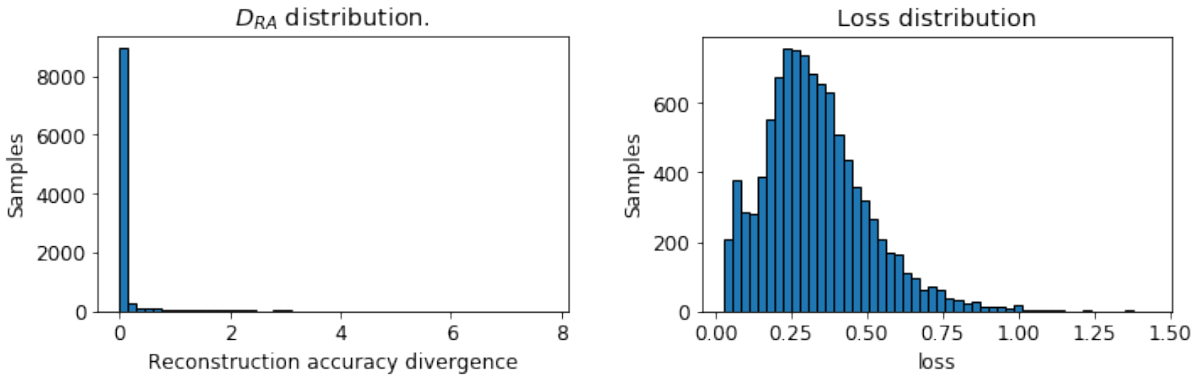


Figure 4.5: $Surprise_{RA}$ and the classification error curves.

4.2.1 Surprise batch size

In this work we mean by a batch, a sequence of samples over which surprise is calculated. This is not to be confused with the batches used to train a neural network. For the remainder of this work, the batch size will refer to the number of samples in a batch used to measure surprise. The $Surprise_{RA}$ values of the previous section are computed over the entire MNIST test set by averaging the individual D_{RA} values of each sample. In order to be a useful metric for surprise, the number of samples to calculate the $Surprise_{RA}$ should be relatively small. This can enable quick interventions when the surprise value exceeds a certain threshold. On the other hand, the batch size should be large enough so that the surprise of a certain batch does not differ too much from the surprise over all batches. If the batch size is chosen too small, non-surprising batches might be wrongly regarded as surprising. Figure 4.6 shows the reconstruction accuracy divergence and loss distributions for individual samples on the non-rotated test set. It can be noticed that the D_{RA} values are generally very close to 0, but a few outliers exist with values above 5, while the loss distribution does not have large outliers. As the $Surprise_{RA}$ of the test set rotated by 90° is only around 2, these outlier values are extremely high to be present in a non-rotated test set where no surprise should be present.

A good batch size should result in as few as possible batches that are falsely seen as



(a) (b)

Figure 4.6: Left: Distribution of the D_{RA} for individual samples. Right: Distribution of the reconstruction loss. Both histograms contain 10,000 samples.

surprising in the non-rotated and thus non-surprising test set. As the least surprising variant of the test set, which is the variant that is rotated by 5°, has a $Surprise_{RA}$ of 0.2 and the non-surprising, non-rotated test set has a surprise of 0.13, the first threshold for slightly surprising events could be set at 0.19. As the second threshold, representing a higher surprise, a value of 0.5 could be taken, which corresponds to a classification error of about 10%. In order to determine the ideal batch size, different sizes are tested on the non-rotated test set and the amount of false positives, i.e. batches that exceed a surprise threshold, are registered. Figure 4.7 shows different batch sizes and their corresponding percentage of false positives. Using batch sizes of 250 results in about 10% false positives, going up to 500 results in even less false positives. Choosing the batch size is a trade-off between the number of false positives and how quickly surprise can be detected.

4.2.2 Adapting the network after a surprising event

When a surprising event has occurred, the network has to adapt in such a way that the 'surprising' inputs are no longer seen as surprising. In this work, we adapt the network to the surprising event of rotated inputs by retraining the network on a rotated training set.

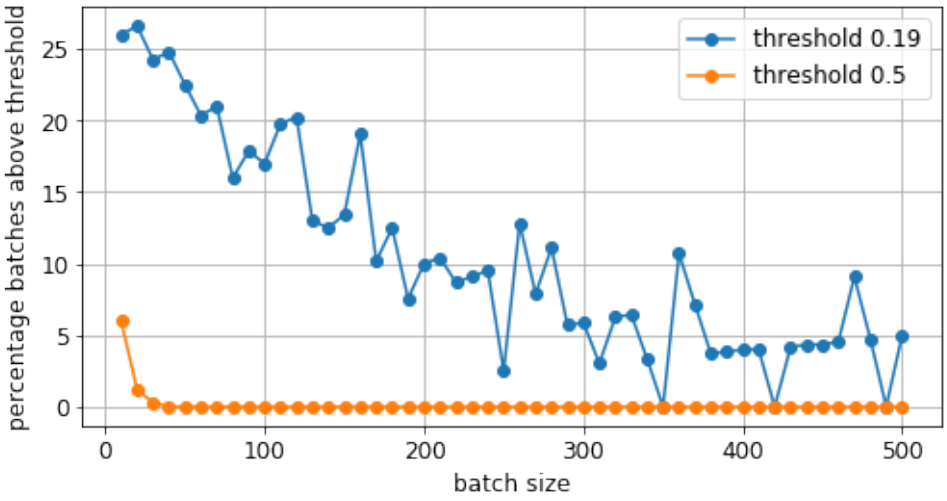


Figure 4.7: Percentage of batches that exceed certain surprise thresholds for different batch sizes.

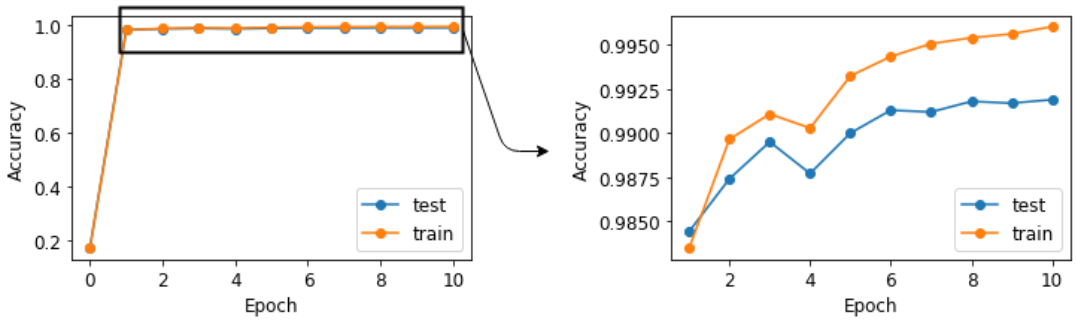


Figure 4.8: Validation curves of the retraining process of the classifier. The network is trained to classify the MNIST dataset rotated by 90°.

Adapting to a 90° rotation

When the inputs of the network suddenly get rotated by 90°, the $Surprise_{RA}$ of the network goes up to nearly 2.0, which is more than ten times higher than the surprise of the non-rotated test set, and the classification accuracy goes down to 17.6%. Adapting to the new situation means retraining the network. Therefore the MNIST training set is also rotated by 90° and then used to retrain the network. After 1 epoch the accuracy jumps back to 98.19%, and after 5 epochs the network is again above 99% with an accuracy of 99.06%. When giving the network as much training time as when the network was trained originally, which is 10 epochs, the model reaches an accuracy of 99.22% on the rotated test set. The validation curves are shown in figure 4.8.

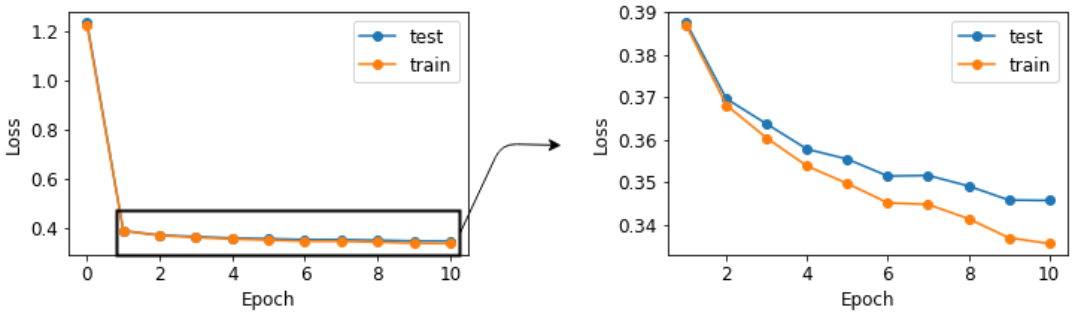
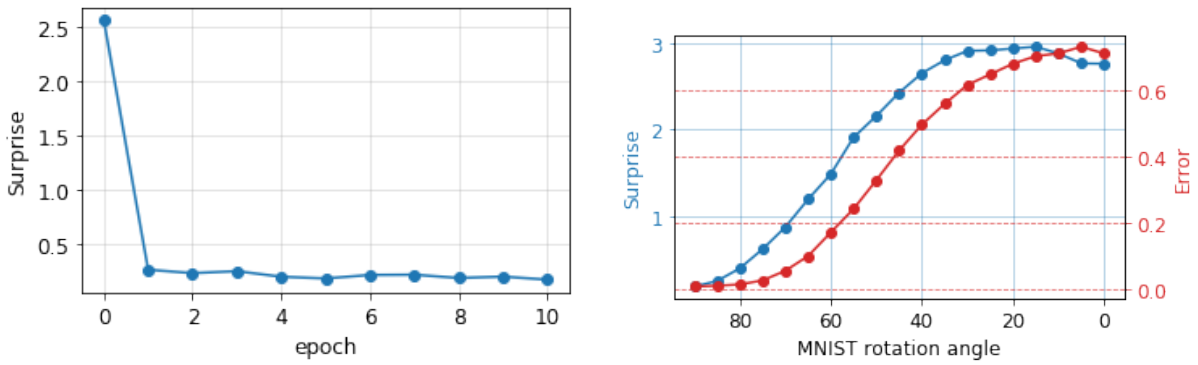


Figure 4.9: Validation curves of the retraining process of the autoencoder.

As the reconstruction accuracy surprise makes use of the autoencoder extension of the network, that part also needs to be retrained. Not retraining the decoder would result in an autoencoder of which only half the weights are retrained to do a new task, which would make the autoencoder useless. The average reconstruction loss is 1.2 before retraining. Just like the classifier, the decoder adapts quickly to retraining. After 1 epoch the reconstruction loss is down to 0.38, which lowers further to 0.34 after 10 epochs. This is shown in figure 4.9.

The surprise follows a similar trend, before retraining the $Surprise_{RA}$ for the rotated test set was 1.94, after only retraining the classifier part this further increased to 2.68. After 1 epoch of retraining the decoder, this dropped back to 0.33, which is no longer surprising. After 10 epochs the surprise becomes 0.18. When retraining both parts of the network is done, $Surprise_{RA}$ can again be used to measure how surprising events are to the new situation. Figure 4.10 shows the evolution of the average surprise of the rotated test set during the retraining phase of the autoencoder and how surprise can afterwards again be used to capture events that are unexpected to the new situation. This figure shows that adapting the network to be accurate on the rotated samples has as a consequence that it is no longer accurate on the original non-rotated samples. This is a tendency of neural networks that is known as catastrophic interference, which shows that previously learned information gets forgotten upon learning new information [30].



(a) (b)

Figure 4.10: Left: Evolution of $Surprise_{RA}$ for the test set rotated by 90° as the autoencoder part of the network gets retrained. Right: After retraining the network, the network can again capture the surprise of situations that are unexpected to the retrained network.

Adapting to a 10° rotation

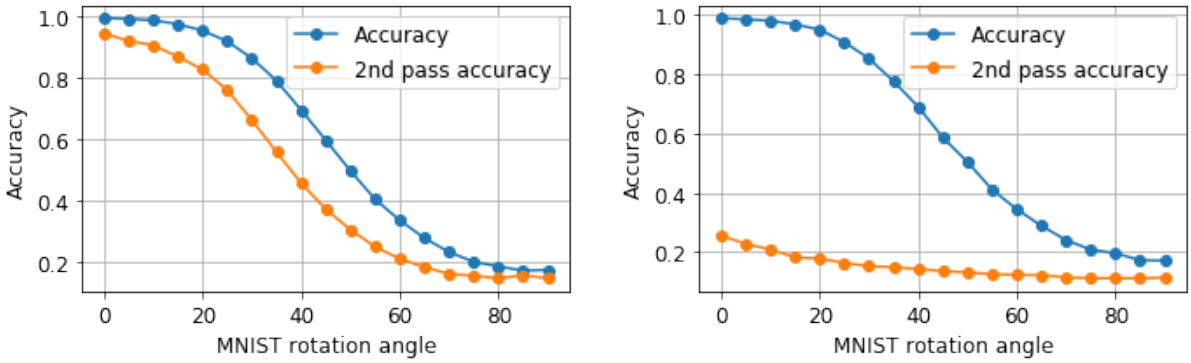
Feeding the network input images that are rotated by 10° is, with a $Surprise_{RA}$ of 0.22, perceived as less surprising than the previous situation where the rotation was 90°. This also has an effect of the speed at which the network can adapt to the new situation. After only 1 epoch of retraining, the classifier goes from an accuracy of 98.4% back to 99.09%. The training loss recovers from 0.51 after retraining the classifier to 0.35 after 1 epoch, which also brings the surprise back down to 0.13. As in this situation the surprising inputs differ only slightly from the non-rotated inputs, the accuracy on the non-rotated test set stays reasonably high, achieving an accuracy of 98.65%.

4.2.3 Influence of the autoencoder part

The decoder part of the network is the most crucial part for the $Surprise_{RA}$ calculation. As established earlier, the autoencoder needs to be able to reconstruct images similar to the ones seen during training reasonably well while not being able to reconstruct images of surprising situations. The baseline architecture fulfils these requirements as can be seen on the reconstructed images shown in figure 4.3, it achieves a reconstruction error of 0.32 on the non-rotated test set which becomes twice as bad on the test set rotated

by 90 degrees with a reconstruction error of 0.76. Besides those characteristics, it is also essential that the network can still recognise the reconstructed images when they are fed as new inputs to the classifier. In the baseline model, the accuracy of the reconstructed images for all rotated variants of the test set is between 3% and 8% lower than the original input image.

This surprise metric was also tested on a slightly modified version of the baseline. This version had a slightly lower accuracy, 99.03% instead of 99.14%, but a much worse performing autoencoder. The modified model has a reconstruction loss of 0.46 for the non-rotated test set. This is substantially higher than the baseline model. It however still satisfies the requirement that the original digits shown in the input images are recognisable in the reconstructions at the output of the autoencoder. Similarly to the baseline model, the reconstruction loss on the test set rotated by 90° was 0.78, which makes the input digits no longer recognisable. But while the digits in the original inputs can still be recognised in the reconstructed outputs when visually inspected, the network has difficulties with accurately classifying them. The accuracy of the network on the reconstructed images is only 25% on the non-rotated test set. Figure 4.11 shows the accuracy of the classifier for both the original inputs and the reconstructed inputs for both the baseline model and the modified model.

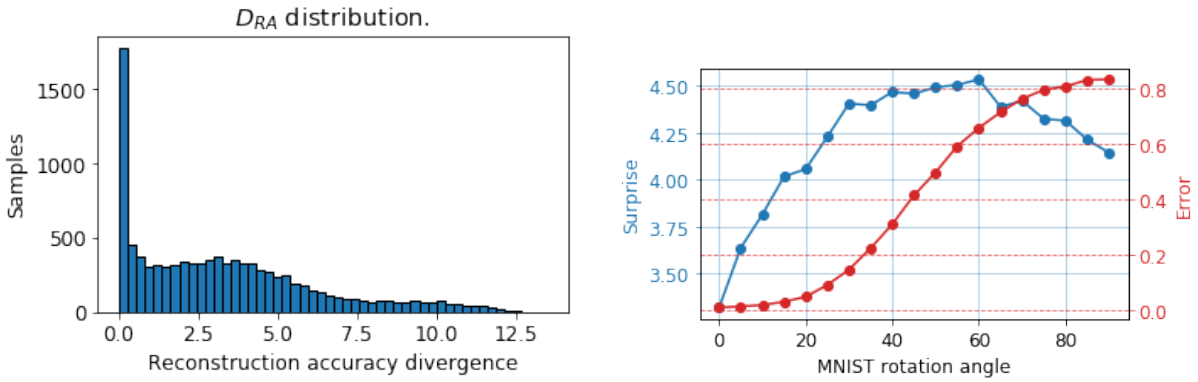


(a) (b)

Figure 4.11: Accuracy of the classifier on the input images versus the accuracy of the reconstructed images, as the test set gets rotated from 0° to 90 °, for the baseline model (left) and the modified model (right).

Because of the bad classification accuracy of the reconstructed images in the modified

model, the surprise can no longer be reliably measured. The distribution of the D_{RA} of the individual samples shows that there are much more outliers. The evolution of $Surprise_{RA}$ on the test set as it gets rotated from 0 to 90 degrees also has much worse characteristics than the baseline model. Figure 4.12 illustrates both of these properties of the modified network.



(a) (b)

Figure 4.12: Left: Surprise distribution of the modified model. Right: $Surprise_{RA}$ of the modified model as the test set gets rotated.

4.2.4 Reconstruction accuracy divergence of individual samples

This section looks at the reconstruction accuracy divergence of individual samples to get insight into what exactly makes a sample 'surprising' and what the network is doing. Insight can be gained by looking at individual samples of the non-rotated test set with an extremely high D_{RA}. Figure 4.13 shows an example of a sample with a high reconstruction accuracy divergence. The sample has a D_{RA} of 6.08. This can be explained by the fact that the classifier is quite confident that the image contains a 7, with only a small probability that it could be a 4 when given the original input. However the opposite happens with the reconstructed image, where relatively sure but wrongly is stated that the image contains a 4, with a small probability that it might be a 7 instead. When looking at the image, it can be understood that the network classifies the reconstructed image as a 4. This shows how dependent the metric is on the autoencoder performance, the network has no issues correctly classifying the original image as a 7, but after going through the entire network,

it makes a wrong prediction on the reconstructed input.

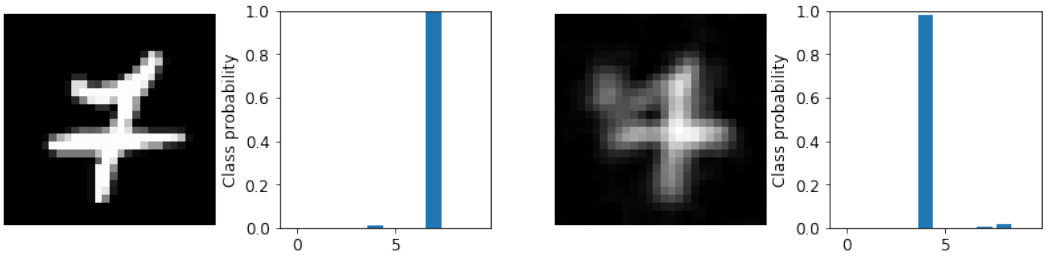
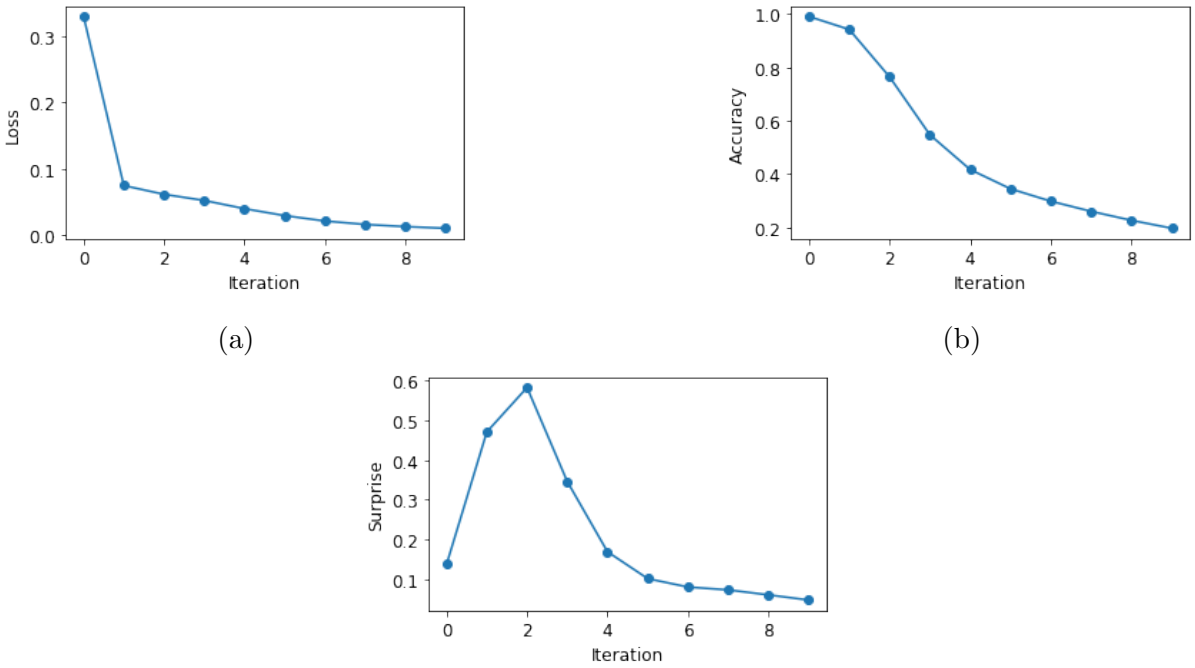


Figure 4.13: Sample with a high D_{RA}.

This also raises the question of what would happen when the reconstructed images are fed through the entire network as new inputs. How would the reconstructed image and the surprise evolve as the image gets put through the network again and again, i.e the reconstructed image is treated as original input in the following iteration? As an experiment, all input images of the test set are looped ten times through the network, calculating the accuracy, reconstruction accuracy divergence and reconstruction loss on every iteration. The results are shown in figure 4.14. The reconstruction loss lowers on each iteration, indicating that the image converges to a certain final image after a while. However, as the accuracy also drastically lowers each iteration, the image to which it converges does not seem to represent the same digit as the original input image. Lastly, the average D_{RA} and thus the $Surprise_{RA}$ metric goes up for a few iterations to finally also go to 0 as the image converges. This surprise behaviour indicates that for a few iterations the classifier predicts different output probabilities for the input than for the reconstructed image.

Figure 4.15 shows the reconstructions of 4 samples as the output of the autoencoder is fed through the network again as new input in the following iteration. It can be noticed that the majority of inputs converge to a certain image resembling an 8. This explains the behaviour for the above metrics. The loss decreased as the reconstruction converges to a certain image, the accuracy decreases as this certain image resembles an 8, regardless of the original input. And finally the surprise goes up for a few iterations as the network starts changing the reconstruction slowly from the original input to an 8, after which it goes down again once this reconstruction starts clearly looking like the final image.



(a) (b)

(c)

Figure 4.14: Loss, accuracy and surprise metrics as the images of the test set loop multiple times through the network.

While it would be better if the reconstructed image converged to an image similar to the original input, the convergence to this wrong image only happens after a few iterations. This phenomenon does not influence the D_{RA} and $Surprise_{RA}$ as only the original input and the first reconstruction are used in the surprise calculation.

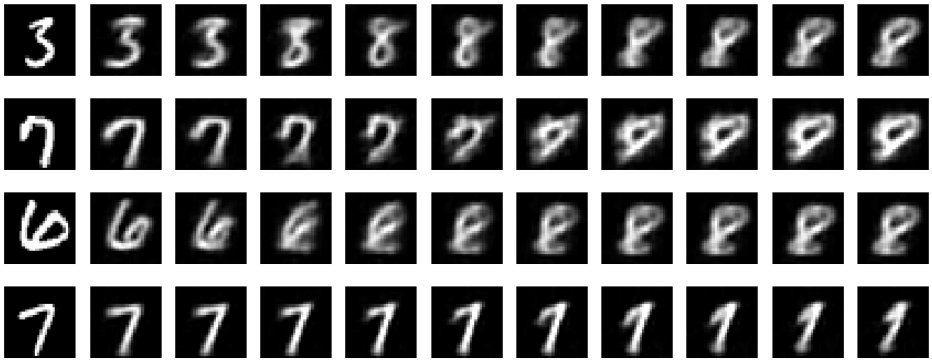


Figure 4.15: Convergence of some input images of the test set when fed multiple times though the network.

One last experiment that gives insight into the reconstruction accuracy divergence of samples is looking at the D_{RA} for a single sample as it gets rotated from 0° to 90°.

Figure 4.16 shows the evolution of two individual samples. It can be noticed how the prediction of the samples stays accurate for a reasonably long time, until 40° for the first sample and 60° for the second sample. As the reconstructed image starts looking less like the input image, the D_{RA} increases. The reconstruction accuracy divergence drops again once the network makes the same wrong prediction on the rotated input image as on the reconstructed image.

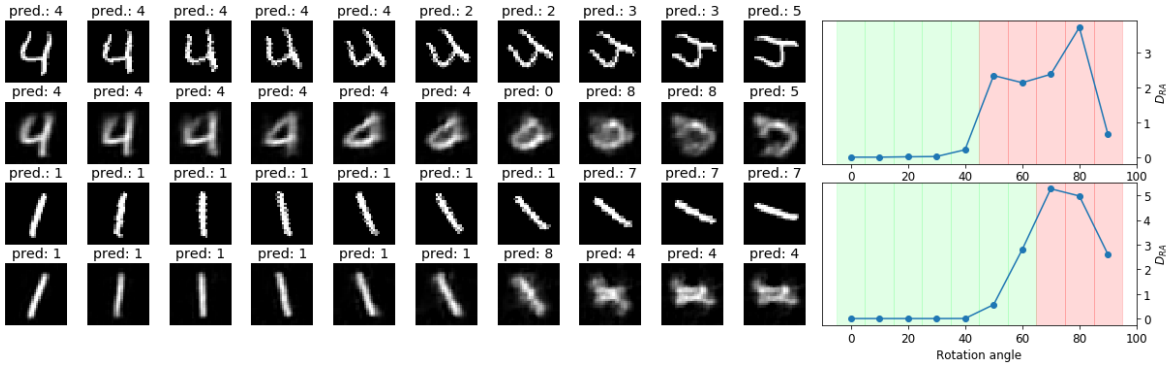


Figure 4.16: Surprise of two sample image as they get rotated to 90 degrees. Top row is the original input, bottom row the reconstructed version. The prediction of the network is shown above the images. The figure on the right shows the surprise for each rotation, a green background means the correct class was predicted.

4.3 Hidden activations surprise

Hidden activations surprise captures surprise without using the autoencoder extension, instead it uses the activations of the last hidden layer. The baseline architecture, as described in section 3.3, consists of several convolutional layers followed by one fully connected layer which goes from 1152 nodes on the last hidden layer to the 10 output nodes. Capturing surprise in this architecture is done by looking at the activations in the last hidden layer. As explained in section 3.1.2, $Surprise_{HA}$ is measured by estimating $P(h_1, ..., h_{1152})$, the joint probability of the outputs of the last hidden layer. Estimating this probability distribution function is done by observing the outputs at this hidden layer on a set of data points. The inputs used for this estimation should not be used during the training phase of the classifier. The training set is therefore split in two, one part

containing 50,000 samples is used to train the classifier, and another part containing 10,000 samples is used to estimate P. From now on this first part will be called the classifier training set and the latter part the surprise training set. The baseline classifier achieves an accuracy of 99.07% after training on the classifier training set. This is slightly lower than the accuracy reported in section 4.1 where the full training set of 60,000 samples was used.

Estimating the joint probability $P(h_1, ..., h_{1152})$ with 10,000 samples is however impractical due to the curse of dimensionality. To overcome this limitation, we assume that all probabilities $P_1(h_1), ..., P_{1152}(h_{1152})$ are independent, just as is done in the naive Bayes algorithm. Estimating each individual probability density function is considered to be feasible with 10,000 samples. Additionally, to further simplify the surprise calculation, only a subset of the 1152 nodes is taken as not every node contributes equally to the final output predictions. Only observing the hidden nodes that contribute the most to the final prediction should be sufficient to capture surprise. The selection of which nodes to use is done by looking at the nodes that are activated the most. As the activation function in the last hidden layer a ReLU is used, hence a node can be seen as activated when its value exceeds 0. The selection of nodes to consider happens per class. This is done to avoid including nodes that are active for all inputs, no matter which class is represented in the input, and therefore carry little information about the input. The complete node selection procedure is as follows: first, for all the samples in the surprise training set, the values at the last hidden layer are stored together with the predicted label. Next, for each possible output class, the nodes that are activated the most are selected. This selection is based on the predicted labels and not on the true labels, see chapter 5 for a more elaborate discussion on this. Finally, for each class the most activated nodes are added to the final set of nodes until this set contains at least 50 nodes. As an example: when looking at the most active node when label 2 is predicted, node 182 comes out as the most activated node. Figure 4.17 shows the histogram of the activations when only considering samples that are classified as an 2 and the histogram of the node when looking at all input samples. It can be seen that the distribution has a peak at 0 when the histogram contains the values of all inputs regardless of the predicted class. This shows

that this node activates more times when the input resembles an 2 than when it resembles any other digit.

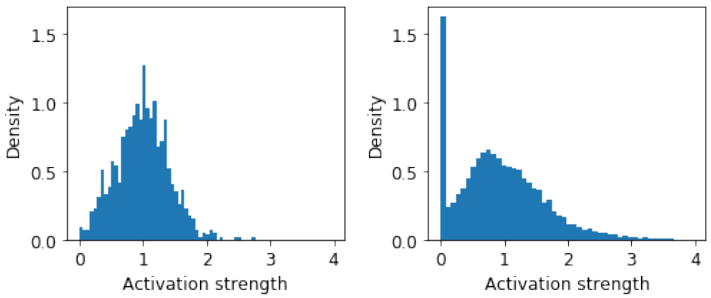


Figure 4.17: Histogram of the output values of node 182 of the 1152 nodes of the last hidden layer. Figure on the left is the histogram when only input samples that are predicted to be an 2 are used, while the figure of the right is the histogram over all inputs of the surprise training set.

Estimating the probability density function for each of the selected nodes happens in two stages. As a ReLU activation function is used, most of the nodes have a high percentage of values that are 0. $P(h_i = 0)$ is simply estimated as the fraction of the samples in the surprise training set that have a 0 output for hidden node h_i. For the values greater than zero, the probability distribution is approximated using Gaussian kernel density estimation (KDE) [42].

To capture surprise, this estimation is also done over the test set. This estimation P^* is then used to calculate $Surprise_{HA}$ using equation (3.4). The $Surprise_{HA}$ of the non-rotated test set, using all samples in the test set to estimate P^*, is 0.0685. The surprise is then calculated over rotated versions of the test set, up to a rotation of 90°. Figure 4.18 shows the development of the surprise values and the classification error as the test set gets rotated from 0° to 90°. The least surprising test set, which has a rotation of 5°, has a surprise of 0.5476. This is a value that is 8 times larger than the non-surprising test set. The most surprising variant of the test set, the version rotated by 90°, has a surprise of 14.496.

Looking at the probability distribution estimations for the selected nodes of the hidden layer, it can indeed be seen that the estimations done on the surprise training set and the non-rotated test set are nearly identical. However, the estimation done on the rotated test

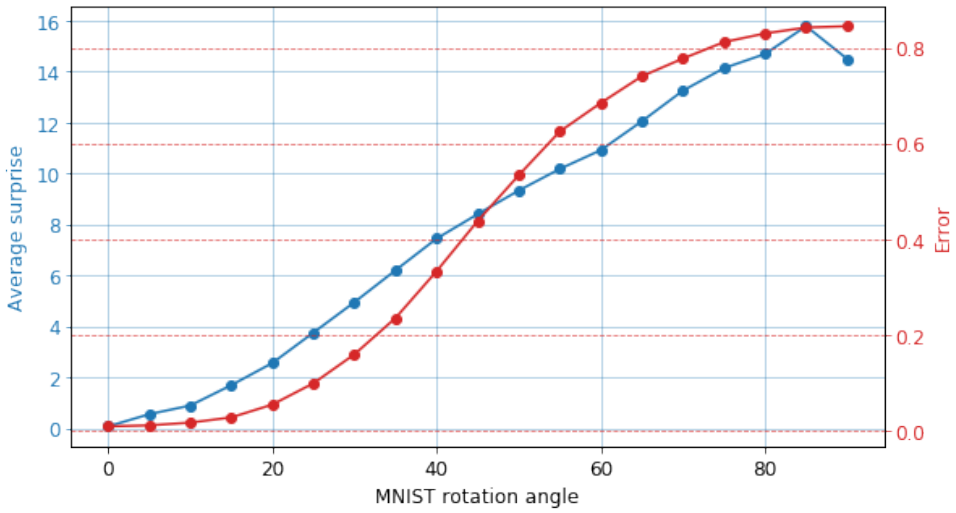


Figure 4.18: $Surprise_{HA}$ and classification error curves.

set shows different approximations for the activation distribution of some hidden nodes. This is shown in figure 4.19. The blue curves show the probability density function, estimated on the surprise training set. The red curve shows the estimation based on the rotated test set. The numbers above the figures indicate which one of the 1152 hidden nodes is estimated.

4.3.1 Surprise batch size

The previous results used the entire test set to calculate the $Surprise_{HA}$. However, to become a practically useful metric, it should be possible to calculate this over smaller sets of inputs. Similarly as to the batch size in $Surprise_{RA}$, the size of the batch is a trade-off between the amount of batches that are wrongly seen as surprising and the speed at which surprise can be detected. The batch size in reconstruction accuracy surprise determined the amount of samples that should be used to average the surprise over, in this surprise metric however, the batch size determines the amount of samples used to approximate the probability distribution of the hidden nodes. Larger batches mean more data points that can be used in the Gaussian KDE, especially considering that a substantial part of the activations for a node will be 0.

Evaluating different batch sizes is done two folded: by looking at the percentage of

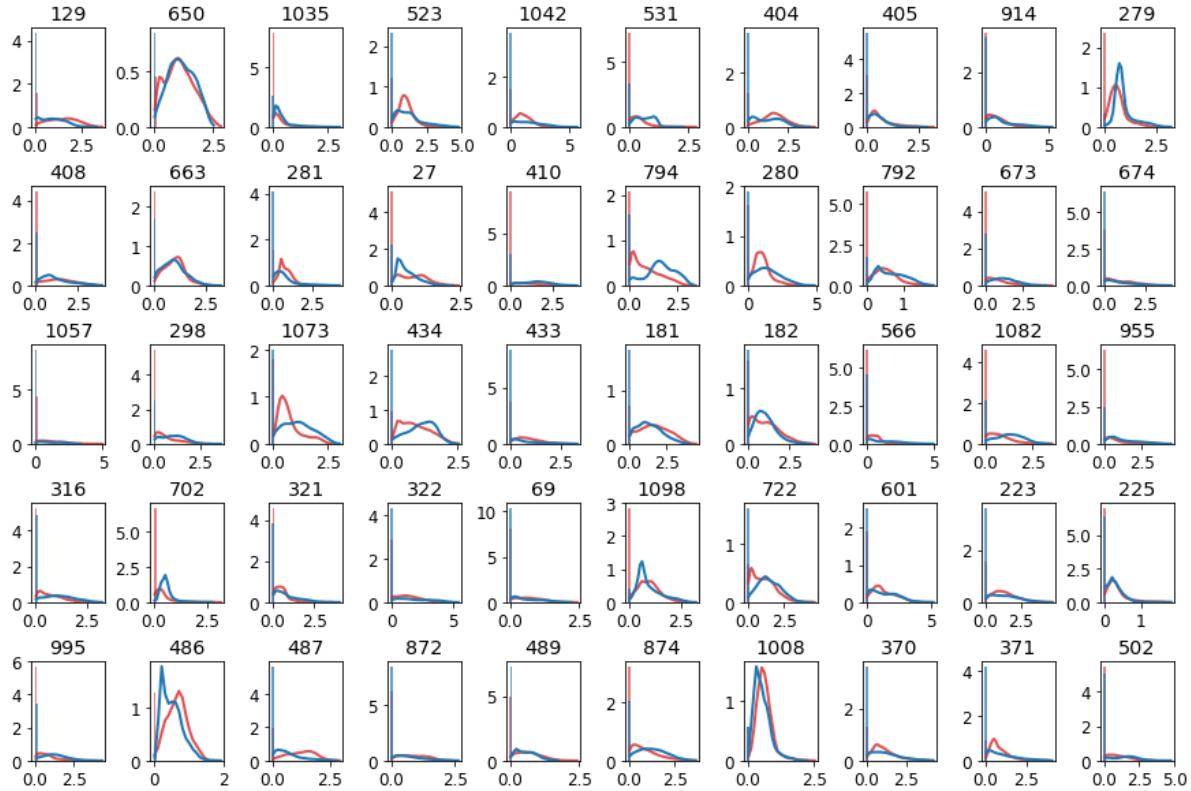
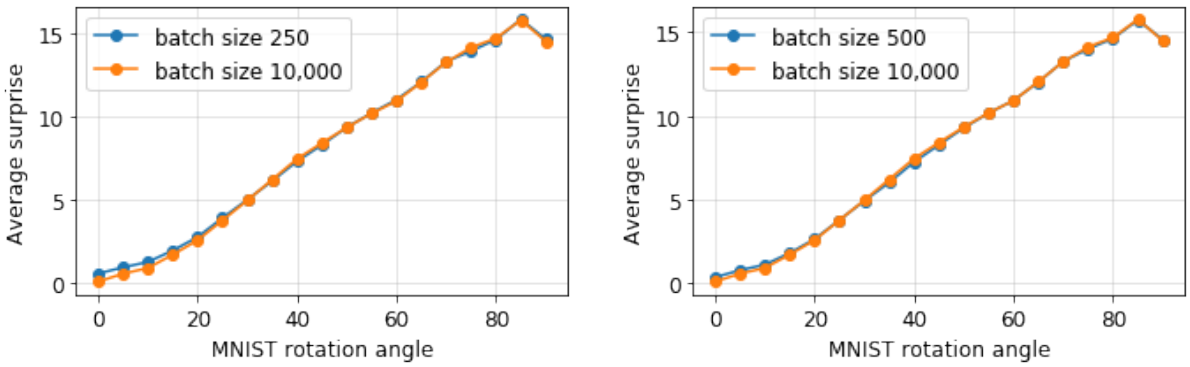


Figure 4.19: Approximations of the probability density functions of the most active hidden nodes in the last layer. The blue curves are estimated on the surprise training set and the red curves are estimated on the rotated test set. The number above each plot indicates which from the 1152 nodes is estimated.

batches of the non-rotated test set that are mistakenly regarded as surprising. And, by looking at the difference between the average surprise calculated over the batches of a certain size and the surprise over the entire test set. As the non-rotated, non-surprising test set has a $Surprise_{HA}$ of 0.068 and the least surprising variant of the test set a $Surprise_{HA}$ of 0.54, a first threshold used to define light surprise could be set at around 0.35, this value is chosen arbitrarily. Testing different batch sizes shows that a batch size of about 500 has a good trade-off between false positive batches and speed. The surprise metric gets more inaccurate for slightly surprising rotations when the batch size goes down. This can be seen in figure 4.20, which shows the average surprise over a batch size of 250 and 500 respectively, compared to the surprise calculated over the entire test set, as the inputs get rotated to 90 degrees. Both batch sizes yield an average surprise that comes close to the surprise value that is calculated over the entire test set. However, as the batch size gets smaller, $Surprise_{HA}$ becomes more inaccurate for the least surprising

rotations. As a result, when using a batch size of 250, all batches have a $Surprise_{HA}$ value higher than the threshold of 0.35 while the surprise of the batches with batch size 500 never goes below this threshold. This also becomes clear when calculating the mean square error between the average surprise of a certain batch size and the surprise over all samples in the test set, for different rotations of the test set. The MSE when using batch size 500 is 0.0155, which increases to 0.0399 when the batch size is reduced to 250.



(a) (b)

Figure 4.20: The average surprise when using batch size 250 and 500 compared to the surprise when calculated over the entire test set, as the samples get rotated from 0° to 90°.

4.3.2 Adapting the network after a surprising event

When the inputs of the network are suddenly rotated by 90°, the network measures a $Surprise_{HA}$ around 14.5, and the accuracy drops to 15.5%. As discussed in the corresponding section about the reconstruction accuracy surprise, the network can adapt to this situation by retraining. As the classifier used in this section is the same as used for evaluating $Surprise_{RA}$, it takes the same time for the classifier to adapt to the 90° rotation. However, there is no autoencoder needed in the setup for hidden activations surprise, which reduces the total time the network needs to adapt to the new situation. The only remaining thing to do after the classifier is retrained, is to make a new estimate P of the probability distribution of the hidden nodes, which includes selecting the new most activated nodes. Figure 4.21 shows that the hidden activations surprise can be used again to capture events that are unexpected to the retrained network after adapting to

the rotated test set.

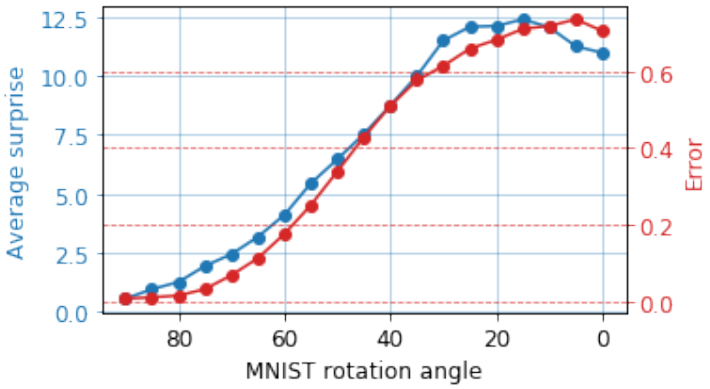


Figure 4.21: The retrained network can again capture the surprise of situations that are unexpected to the retrained network.

4.4 Reconstruction loss surprise

The third and final surprise method tested is the reconstruction loss surprise, which is a combination of the previous formulas. The architecture of the full network is the same as used in reconstruction accuracy surprise, as both formulas use the autoencoder extension. However, as a probability distribution function needs to be estimated just as in hidden activations surprise, the training set is split in the same fashion as done for $Surprise_{HA}$. This has as a consequence that the classifier accuracy is also 99.07%. Training the autoencoder with this modified training set results in an average reconstruction error of 0.334, which is only slightly higher than the reconstruction loss when trained on the full training set.

The probability distribution that is estimated is that of the loss of the autoencoder. In reconstruction accuracy surprise, the reconstruction $\boldsymbol{x}^*$ of the input sample $\boldsymbol{x}$, created by the autoencoder, was fed through the classifier a second time. In this metric, the probability distribution function P_l of the reconstruction loss between $\boldsymbol{x}$ and $\boldsymbol{x}^*$ is estimated and then used to measure surprise. Estimating the probability distribution P_l of the reconstruction loss is achieved by storing the losses for all samples of the surprise training set and then applying Gaussian KDE to approximate a probability density func-

tion. Figure 4.22 shows the histogram of the reconstruction losses and the corresponding probability density function estimated using Gaussian KDE.

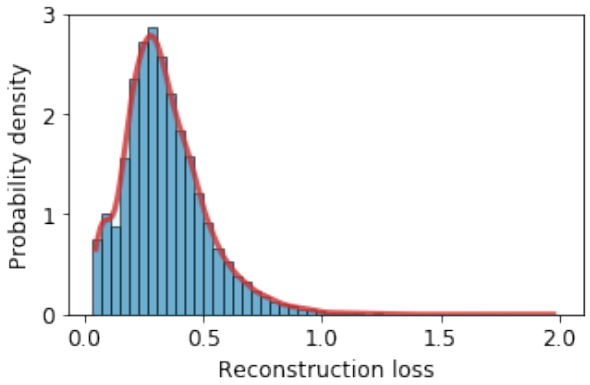


Figure 4.22: Histogram and approximated probability density function of the reconstruction loss on the surprise training set.

Next, the probability distribution of the reconstruction loss is estimated over the data set of which we want to measure the surprise. This approximation P_l^* is then used in equation (3.5) to calculate the reconstruction loss surprise. The $Surprise_{RL}$ of the non-rotated test set is 0.00113 when using all 10,000 test samples to approximate P_l^*. The test set is then rotated from 0° to 90°, and the surprise is calculated over these variants of the test set. The least surprising test set, which has a rotation of 5°, has a $Surprise_{RL}$ of 0.0258. This is more than 20 times higher than the surprise for the non rotated test set, while the classification error only slightly increases from 0.93% to 1.15%. The most surprising version of the test set is the one with a rotation of 90°. The accuracy on this rotated version is as low as 15.5%, which corresponds to a $Surprise_{RL}$ of 2.185. Figure 4.23 shows the evolution of the surprise values compared to the classification error as the test set gets rotated to 90 degrees. Over the course of all rotations, the surprise metric seems to show similar behaviour compared to the classification error.

Looking at the probability distribution of the reconstruction loss estimated on the surprise training set, non-rotated test set and the test set rotated by 90°, gives more insight into the surprise values. These probability density functions are shown in figure 4.24. It can be seen that the distribution P_l estimated on the surprise training set and the distribution P_l^* estimated on the non-rotated test set are nearly identical. However, when

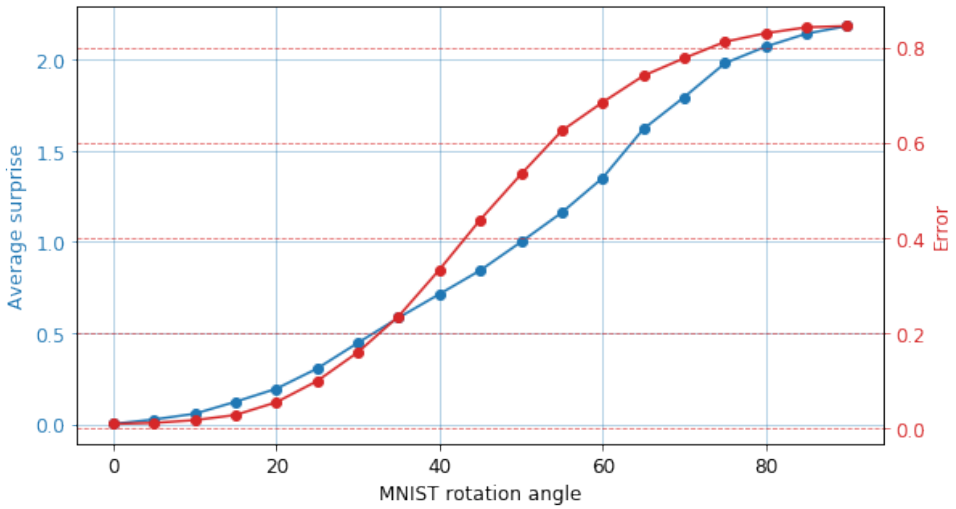


Figure 4.23: $Surprise_{RL}$ and classification error curves.

the distribution P_l^* is estimated on the rotated test set, the probability density function differs a lot from P_l.

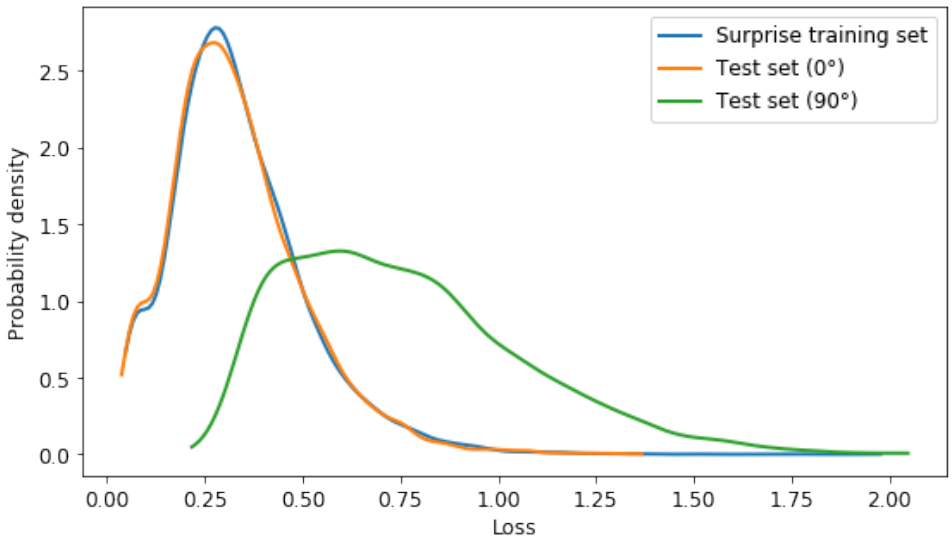


Figure 4.24: Approximated probability density functions of the reconstruction loss. Estimated on the surprise training set, the non-rotated test set and the test set rotated by 90°.

4.4.1 Surprise batch size

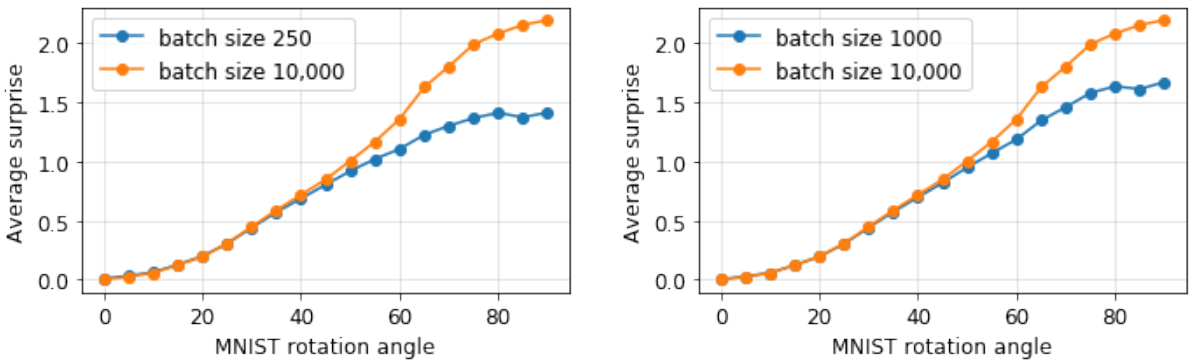
Similar as with hidden activations surprise, to be a practically useful metric, it should be possible to calculate the surprise over relatively small batches. The batch size determines

how many samples are used in the estimation of P_l^*. Smaller batch sizes have fewer samples to make an accurate estimation with but allow the surprise to be calculated faster as fewer inputs are needed.

The evaluation of different possible batch sizes is done by looking at the difference between the average surprise calculated over the batches of a certain batch size and the surprise calculated over the entire test set. The differences are compared for different rotations. Another evaluation metric is the percentage of batches of the non-rotated test set that are mistakenly seen as surprising. To determine this percentage, a threshold on the surprise value needs to be set that marks batches as surprising. As the $Surprise_{RL}$ for the non-surprising test set is 0.0011 and the $Surprise_{RL}$ for the least surprising test set, the variant that is rotated by 5°, is 0.025, a suitable threshold to capture light surprise with could be set at 0.02. Testing different batch sizes shows that for larger rotations, the surprise estimations for different batch sizes diverge from each other; larger batch sizes typically lead to larger surprise values. This is illustrated in figure 4.25, for batch sizes 250 and 1000. The large difference between the average surprise over the batches and the surprise over the entire test set is not present for slightly surprising situations where the test set is only rotated over a few degrees. Therefore this is not a real issue as this still allows surprise to be captured well, just less accurate for very surprising situations. Using a batch size of 250 provides a good trade-off. This results in 2.5% of the batches being wrongly seen as surprising, this starts going up when using smaller batch sizes.

4.4.2 Adapting the network after a surprising event

An important experiment that is used to evaluate the usefulness of the surprise formula is to check how the $Surprise_{RL}$ values evolve as the network adapts to the surprising situation. Applying a rotation by 90° to the inputs results in an accuracy that drops to 15.5%. The $Surprise_{RL}$ also increases from 0.0125 for the non-rotated test set to 1.403 for the rotated test set. Note that these values slightly differ from the ones reported earlier, as this experiment used the recommended batch size of 250 and not the entire test set to calculate the surprise over. After detecting this surprising event, the network can be



(a) (b)

Figure 4.25: The average surprise when using batch size 250 and 1000 compared to the surprise when calculated over the entire test set, as the samples get rotated from 0° to 90°.

retrained to adapt to the new situation. As the architecture of the network is identical to the one used in reconstruction accuracy surprise, similar results are obtained. After 1 epoch of retraining the classifier, the accuracy increases from 15.5% to 98.2%. Continuing the training process for 10 epochs results in an accuracy of 99.15%.

As reconstruction loss surprise uses the autoencoder extension of the network, that component also needs to be retrained. Just like the classifier, this part of the network adapts quickly to retraining. After 1 epoch, the reconstruction loss reduces from 1.27 to 0.39, which drops further to 0.344 after 10 epochs.

The $Surprise_{RL}$ values follow a similar trend, and after both parts of the network are trained for 10 epochs, the $Surprise_{RL}$ for the rotated test set is only 0.022, and thus no longer surprising. Figure 4.26 shows how the adapted network can again be used to capture events that are surprising to the new situation. Note that all surprise values are slightly higher compared to the original network. This is however not a problem as surprising events also report higher $Surprise_{RL}$ values, it does mean though that new thresholds need to be defined to detect surprise.

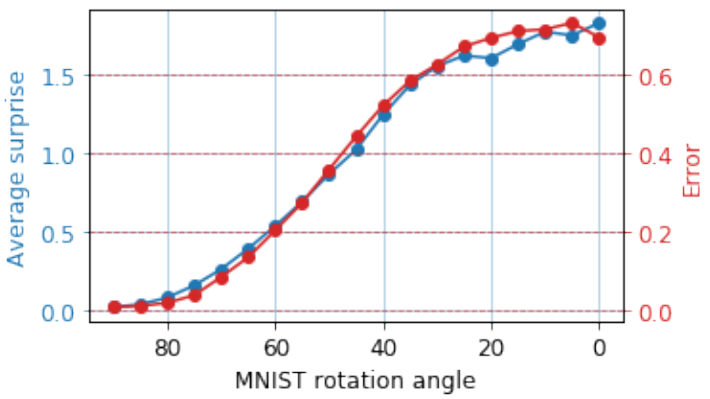


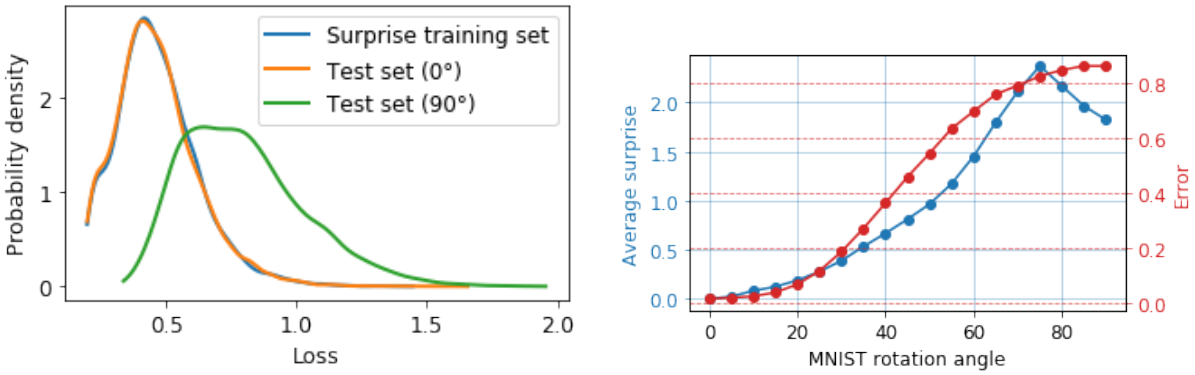
Figure 4.26: After retraining the network, it can again capture the surprise of situations that are unexpected to the retrained network.

4.4.3 Influence of the autoencoder part

During the evaluation of reconstruction accuracy surprise, it was shown that the metric was sensitive to the autoencoder performance. Considering that the reconstruction loss surprise uses the same autoencoder architecture, it is also checked whether this formula is equally influenced by the autoencoder performance. As reported in section 4.12, the modified network with the worse autoencoder has an average reconstruction loss of 0.46 compared to 0.32 of the baseline model. It, however, satisfies the requirement that the input digits are still visually recognisable for the non-rotated test set and no longer recognisable in the rotated test set. As reconstruction loss surprise relies heavily on the reconstruction loss, the question remains how the autoencoder performance influences the reliability of the surprise metric.

The results show that $Surprise_{RL}$ is less dependent on the autoencoder performance as $Surprise_{RA}$. Looking at the estimated probability density functions, it can be seen that the reconstruction loss of the modified model shows similar behaviour compared to the baseline model. It has the same characteristics but shifted towards a higher reconstruction loss. The estimated probability density function P_l, approximated on the surprise training set is still very close to P_l^* approximated on the test set, and much different from P_l^* when the latter is approximated using the rotated test set. The surprise values for rotated versions of the test set also still increase significantly as the rotation that is applied to

the test set increases, with the exception of the larger rotations. Both of these findings are shown in figure 4.27.



(a) (b)

Figure 4.27: Left: approximated probability density functions of the reconstruction loss of the modified model. Right: $Surprise_{RL}$ of the modified model as the test set gets rotated.

Chapter 5

Discussion

The previous chapter discussed the results of the experiments on the different proposed surprise metrics. In this chapter the proposed formulas are compared to each other and the general use of surprise is discussed.

5.1 Benefits of surprise

Surprise in the context of deep learning has many use cases. The biggest advantage of measuring the surprise of a neural network is that it enables capturing situations that are unexpected to the neural network without needing the true labels of the data. One of the characteristics of an unexpected or surprising event in the context of deep learning is that such a situation results in lower accuracy compared to the average accuracy over the training and test set. However, the accuracy of the predictions can only be observed during training and testing as it requires the true labels of the data samples. The test set and training set are usually assumed to have similar data, and as a result the network should not be surprised by any sequence of data samples from either set. Truly surprising situations happen during inference where the labels of the inputs are being predicted and the true labels are no longer available. A surprise metric with similar characteristics

compared to the classification accuracy enables deep learning models to evaluate their performance during inference. When suddenly the surprise values for batches of input samples increase, these batches can be examined by an expert to determine what causes the network to be surprised. The next step is then to adapt to this unexpected situation so that the network can again make accurate predictions and is no longer surprised by the situation. Failing to detect surprising events renders the model useless as it no longer makes accurate predictions. Surprise has enormous potential in the industry where the deep learning models spend the majority of their time in the inference phase. In a lot of situations, noise or bugs may introduce surprising events in existing working models.

In this thesis, we used the MNIST dataset and the 'surprising' event of rotated inputs. Other possible and perhaps more likely examples of surprising events are added noise due to a bad communication channel, having an extra output class that was not present in the training set, etc. The cause of the surprise determines the steps that need to be taken to adapt the network. Adapting to a missing output class requires a different strategy then adapting to rotations. In our experiments we retrained the network on rotated samples, this was done to demonstrate that retraining can be used to adapt the network in an online learning way. If however in a practical application the inputs suddenly get rotated due to accidental misalignment of the input sensor, the best way to adapt to this situation is to rotate the sensor back to its original position and not by retraining the entire network.

5.2 Comparison of the surprise metrics

This section compares the different proposed surprise formulas and discusses their results.

5.2.1 Reconstruction accuracy surprise

Reconstruction accuracy surprise is the only metric of the three that starts by calculating a value on individual samples and then combines them to a surprise metric over a sequence by averaging over the individual reconstruction accuracy divergence values. Although we

have defined surprise as a metric that is taken over a sequence of samples rather than a single sample, being able to obtain a value for individual samples has some advantages. The main advantage of this metric compared to hidden activations surprise and reconstruction loss surprise is that the distribution of the true classes of the inputs does not influence the surprise metric. In $Surprise_{HA}$, the distributions of the activations of the nodes in the last hidden layer are different for each predicted label. The nodes in hidden activations surprise that are selected, are activated the most for a specific label, and as a result show different behaviour when a batch only contains samples of another class. To compare the estimation over a batch to the estimation over the surprise training set, the distribution of the predicted classes of the inputs in each batch should be similar to the surprise training set. Similar behaviour happens for reconstruction loss surprise, the distribution of the reconstruction loss is different for each class. As $Surprise_{RL}$ uses the distribution of the reconstruction loss over samples of all classes, the formula is sensitive to the class distribution in the batches. Reconstruction accuracy surprise is the only formula of the three that is insensitive to the distribution of the classes as the D_{RA} values for the individual samples are independent of the true class. Although for $Surprise_{HA}$ and $Surprise_{RL}$, this could be solved by selecting the samples over a longer period and balancing the class distribution artificially, by subsampling the desired predicted classes distribution.

In order to calculate $Surprise_{RA}$ the classifier needs to be extended to an autoencoder. This effectively doubles the required training time of the network. This is a large increase, however, it may become negligible over time when the inference phase is much longer than the training time. The time required during inference to measure $Surprise_{RA}$ is triple the time it takes to do just the classification. This is because a sample $\boldsymbol{x}$ needs to go through the classifier, the decoder and then through the classifier again. Therefore it is advised to only periodically capture surprise during inference. The frequency at which surprise should be measured is a trade-off between the overhead introduced and the speed at which surprise can be detected.

Reconstruction accuracy surprise has one major downside and that is its sensitivity to

the autoencoder as shown in section 4.2.3. The surprise metric captured by $Surprise_{RA}$ is sensitive to the accuracy of the classifier on the reconstructed image $\boldsymbol{x}^*$. For the MNIST dataset it is relatively straightforward for an autoencoder to reconstruct the original input well enough so that the reconstructed image can still be accurately predicted during the second pass through the classifier. For more complex tasks however, it becomes much more difficult and often impractical to create an autoencoder that can reconstruct the images well enough to be still recognisable for the classifier. Therefore the sensitivity to the autoencoder performance in reconstruction accuracy surprise may be a limiting factor for use in practical applications.

5.2.2 Hidden activations surprise

In hidden activations surprise, the classifier is not extended with a decoder. The only modification to the training process is that the training set needs to be split in a part that is used to train the classifier and a part that is used to estimate $P(h_1, ..., h_n)$. This means either losing accuracy due to having less training samples for the classifier or collecting more labelled data. In the MNIST dataset we split the original 60,000 samples of the training set into 50,000 for the classifier training phase and 10,000 for the surprise training phase. This split was chosen to have the same amount of samples in the surprise training set as in the test set. The exact ratio of this split is not important, the only requirement is that the surprise training set has enough samples to estimate $P(h_1, ..., h_n)$. For large datasets this part may be relatively small compared to the classifier training part.

Not having the autoencoder extension means that the training phase takes only half the time than required for reconstruction accuracy surprise and reconstruction loss surprise. The only overhead during training is the extra computations needed to approximate the probability density function of the hidden nodes. Gaussian kernel density estimation is a very compute intensive task. There have been several studies to increase the performance on this computation-intensive task, including GPU acceleration [31]. While the exact overhead for the autoencoder extension and the Gaussian KDE are model and

task dependent, there is no clear advantage for either when it comes to overhead during inference.

Another advantage of the setup for hidden activations surprise is that it can be used in neural network architectures that do not have an obvious autoencoder extension. Any architecture of which the activations of a hidden layer can be captured, can be used to calculate $Surprise_{HA}$. Therefore this formula might have more potential than the others.

As mentioned in section 4.3, the node selection procedure happens per class. For each predicted class the most activated nodes are added to the final set of nodes. The reason that the predicted class is used rather than the true class is because we are interested in the node activations that lead to a certain prediction. Using the true classes could more make sense if the interest is in how the node activations correspond to the true labels. For the MNIST dataset where the accuracy is over 99%, the impact of choosing the predicted labels instead of the true labels is rather small. However, for more complex tasks with lower accuracy, this choice could have a larger impact.

5.2.3 Reconstruction loss surprise

As mentioned before, reconstruction loss surprise is a combination of the methods used in reconstruction accuracy surprise and hidden activations surprise. As a result it inherits some advantages and disadvantages of both the previous metrics. The advantage of $Surprise_{RL}$ is that it uses an autoencoder, which intuitively captures well what a network finds surprising and what not. But unlike reconstruction accuracy surprise, it is not sensitive to the absolute performance of the autoencoder. As shown in section 4.4.3, as long as the autoencoder satisfies the requirements of being worse at reconstructing surprising samples than at reconstructing non-surprising samples, it results in a good surprise metric. So it uses the idea of $Surprise_{RA}$, but improves it using the probability distribution estimating idea of $SurpriseHA$. Reconstruction loss surprise is therefore a better option than reconstruction accuracy surprise for situations where an autoencoder can not achieve well enough performance for reconstruction accuracy surprise to work.

However, it also inherits the downsides of both previous metrics. As was the case for $Surprise_{RA}$, the autoencoder effectively doubles the training time of the deep learning model. It also needs a split in the training set so that the reconstruction loss can be estimated using Gaussian KDE on the surprise training set. For inference, the autoencoder doubles the time as the sample needs to go through both classifier and the decoder part. This is not as slow as in reconstruction accuracy surprise where the samples have to pass through the classifier part twice. But after the reconstruction losses are calculated for all samples in a batch, the probability distribution still has to be estimated through Gaussian KDE, which is a compute intensive task, as shown in the previous section. This has as a consequence that reconstruction loss surprise has the most overhead during inference of all the proposed metrics. As mentioned earlier, this overhead can be reduced by measuring surprise periodically.

While the Gaussian KDE process is the same as the one used for $Surprise_{HA}$, the calculation for $Surprise_{RL}$ has as advantage that it needs less samples. In hidden activations surprise, 50 different probability distributions of hidden nodes needed to be estimated, and only samples with an activation above 0 contribute to the approximation. $Surprise_{RL}$ on the other hand only needs one function to be approximated and to which all samples in the batch contribute.

5.2.4 Best metric

The experiments conducted in the previous chapter show no clear best metric to use in order to capture surprise. All metrics have advantages and disadvantages and they all show behaviour that is desired in a surprise metric. In order to have a clear view of which formula is the best, more experiments need to be done on larger and more complex problems and using a wider range of surprising situations. However, reconstruction accuracy surprise is likely to be the least useful metric due to its sensitivity on the autoencoder performance.

5.3 Interpreting the surprise values

The hardest part of the various presented methods is the interpretation of the surprise result. What does it mean to have a $Surprise_{RA}$ of 1.254, a $Surprise_{HA}$ of 5.89 or a $Surprise_{RL}$ of 0.67? The answer is that it is very much task and model dependent, a $Surprise_{RL}$ of 0.025 might be surprising for a certain dataset and deep learning model, while that value may indicate no surprise on another dataset and model. In order to determine which value is surprising for a given dataset and model, first the average surprise over the test set needs to be measured. Once this is known a threshold for surprise can be set by looking at the surprise values for all the different batches of the test set. The previous chapter looked into the ideal batch size for every formula. A good threshold would be a slightly higher value than the maximum surprise value found across all the batches in the test set. For example, if for a given dataset and model the average $Surprise_{RL}$ is 0.016 with some batches having values close to 0.25, a $Surprise_{RL}$ threshold to flag a situation as surprising could be set at around 0.30 or higher.

5.4 Future work

This work presents three possible surprise metrics in the context of deep learning. While we obtained promising results for all three metrics, more research needs to be done in order to determine the true added value of measuring surprise during inference. In order to have a clearer view of which surprise metric is the best and the limiting factors of each formula, more experiments need to be done on tasks that are more challenging than image classification on MNIST as well as on a wider range of surprising situations. Another topic for future research is how these formulas can be used in other network architectures such as residual neural networks [45], as both the autoencoder and the activations of the hidden layer strategies are not limited to vanilla convolutional neural network.

As mentioned in the previous section, it is hard to interpret the values of any of the

surprise formulas without context of the dataset and model. Possible topics of future work include more in-depth analysis of the metrics so that their values can be understood more easily.

Finally, the biggest challenge for future work is evaluating the usefulness of it in applications. Surprise shows tremendous opportunities for the industry as a tool to monitor deep learning models that are constantly doing inference. Testing surprise in a practical application can gain valuable insights in how quickly surprise can help the model improve to unanticipated situations. Having a smaller model that is able to quickly adapt to new situations might be both faster and cheaper as a network that is trained to be robust to a large category of possible surprising situations.

Chapter 6

Conclusion

Every human knows surprise as the feeling one gets when something unexpected or unusual happens. It captures our attention and enables us to learn from a new experience. Deep learning models work differently, while their architecture is roughly based on the brain, they do not have a notion of surprise. Deep learning models apply what they have learned to new inputs. When something in the input images changes and creates inputs that are unexpected to what the model has learned, the network will try to keep applying its knowledge on it, but will fail to make accurate predictions.

By measuring the surprise of a deep learning model on input sequences, it becomes possible to track the network performance on new situations without having access to the true classes of the data. Adapting to surprising situations using online learning, is key for machine learning models to stay accurate when inputs suddenly behave unexpectedly.

This thesis defined three metrics to quantify surprise: reconstruction accuracy surprise, hidden activations surprise and reconstruction loss surprise. Reconstruction accuracy surprise and reconstruction loss surprise are metrics based on the idea that an autoencoder is only able to reconstruct images that are similar to the ones seen during training. Hidden activations surprise is based on the idea that the nodes in the last hidden layer of a neural network activate differently during surprising situations. The proposed surprise metrics

in this thesis offer measurements that can be used to capture the surprise of a deep learning model in image classification tasks. The surprise metrics prove being capable of capturing surprising situations. Retraining a model to surprising situations also correctly lowers the surprise again to a non-surprising value. While more experiments on more complex and industry relevant deep learning tasks should be conducted, surprise shows to be a promising tool for deep learning applications.

Bibliography

[1] Alain, G., and Bengio, Y. What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research 15*, 1 (2014), 3563–3593.

[2] Alpaydin, E. *Introduction to Machine Learning*. The MIT Press, 2014.

[3] Azevedo, F. A., Carvalho, L. R., Grinberg, L. T., Farfel, J. M., Ferretti, R. E., Leite, R. E., Filho, W. J., Lent, R., and Herculano-Houzel, S. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology 513*, 5 (2009), 532–541.

[4] Baldi, P., and Itti, L. Of bits and wows: a bayesian theory of surprise with applications to attention. *Neural Networks 23*, 5 (2010), 649–666.

[5] Barto, A., Mirolli, M., and Baldassarre, G. Novelty or surprise? *Frontiers in psychology 4* (2013), 907.

[6] Bengio, Y., Yao, L., Alain, G., and Vincent, P. Generalized denoising auto-encoders as generative models. In *Advances in Neural Information Processing Systems* (2013), pp. 899–907.

[7] Cai, D., He, X., and Han, J. Document clustering using locality preserving indexing. *IEEE Transactions on Knowledge and Data Engineering 17*, 12 (2005), 1624–1637.

[8] Ciregan, D., Meier, U., and Schmidhuber, J. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition* (June 2012), pp. 3642–3649.

[9] Csáji, B. C. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary 24* (2001), 48.

[10] Donchin, E. Surprise!... surprise? *Psychophysiology 18*, 5 (1981), 493–513.

[11] Dumoulin, V., and Visin, F. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

[12] Duncan-Johnson, C. C., and Donchin, E. On quantifying surprise: The variation of event-related potentials with subjective probability. *Psychophysiology 14*, 5 (1977), 456–467.

[13] E., S. C. A mathematical theory of communication. *Bell System Technical Journal 27*, 3, 379–423.

[14] Ekman, P. E., and Davidson, R. J. *The nature of emotion: Fundamental questions.* Oxford University Press, 1994.

[15] Faraji, M. Learning with surprise.

[16] Fletcher, P. C., Anderson, J., Shanks, D., Honey, R., Carpenter, T. A., Donovan, T., Papadakis, N., and Bullmore, E. T. Responses of human frontal cortex to surprising events are predicted by formal associative learning theory. *Nature neuroscience 4*, 10 (2001), 1043.

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

[18] Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. Maxout networks. *arXiv preprint arXiv:1302.4389* (2013).

[19] Hinton, G. E., and Zemel, R. S. Autoencoders, minimum description length and helmholtz free energy. In *Advances in neural information processing systems* (1994), pp. 3–10.

[20] Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[21] Hodge, V., and Austin, J. A survey of outlier detection methodologies. *Artificial intelligence review 22*, 2 (2004), 85–126.

[22] Karpathy, A. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`. Accessed: 2018-07-30.

[23] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] Kullback, S. *Information Theory and Statistics*. Wiley, New York, 1959.

[25] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436.

[26] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[27] LeCun, Y., Cortes, C., and Burges, C. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist 2* (2010).

[28] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml* (2013), vol. 30, p. 3.

[29] Masci, J., Meier, U., Cireşan, D., and Schmidhuber, J. Stacked convolutional auto-encoders for hierarchical feature extraction. In *International Conference on Artificial Neural Networks* (2011), Springer, pp. 52–59.

[30] McCloskey, M., and Cohen, N. J. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, vol. 24. Elsevier, 1989, pp. 109–165.

[31] Michailidis, P. D., and Margaritis, K. G. Accelerating kernel density estimation on the gpu using the cuda framework. *Applied Mathematical Sciences 7*, 30 (2013), 1447–1476.

[32] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)* (2010), pp. 807–814.

[33] Picton, T. W. The p300 wave of the human event-related potential. *Journal of clinical neurophysiology 9*, 4 (1992), 456–479.

[34] Radford, A., Metz, L., and Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[35] Raschka, S. *Python Machine Learning*. Packt Publishing, Birmingham, UK, 2015.

[36] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review 65*, 6 (1958), 386.

[37] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533.

[38] Sahami, M., Dumais, S., Heckerman, D., and Horvitz, E. A bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop* (1998), vol. 62, Madison, Wisconsin, pp. 98–105.

[39] Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development 3*, 3 (July 1959), 210–229.

[40] Saul, L. K., and Roweis, S. T. Think globally, fit locally: unsupervised learning of low dimensional manifolds. *Journal of machine learning research 4*, Jun (2003), 119–155.

[41] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural networks 61* (2015), 85–117.

[42] Scott, D. W. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 2015.

[43] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[44] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[45] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI* (2017), vol. 4, p. 12.

[46] Weiler, M., Hamprecht, F. A., and Storath, M. Learning steerable filters for rotation equivariant cnns. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018).

[47] Wu, R.-S., and Chou, P.-H. Customer segmentation of multiple category data in e-commerce using a soft-clustering approach. *Electronic Commerce Research and Applications 10*, 3 (2011), 331–341.

[48] Xu, D., Ricci, E., Yan, Y., Song, J., and Sebe, N. Learning deep representations of appearance and motion for anomalous event detection. *arXiv preprint arXiv:1510.01553* (2015).

[49] Zeiler, M. D., and Fergus, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.

[50] Zhou, Y., and Chellappa, R. Computation of optical flow using a neural network. In *IEEE International Conference on Neural Networks* (1988), vol. 1998, pp. 71–78.