Report I - SF2957 Statistical Machine Learning

Simon Carlsson 970404-8033 August Regnell 970712-9491

December 2, 2021

Project 1

Objectives

In part one of this project, the aim is to implement and train a support vector machine on the MNIST dataset of handwritten digits. Before implementation of the SVM, we show that the empirical loss function is convex, and derive a subgradient of it. Then, we implement a projected SGD algorithm and investigate the performance with various training set sizes as well as varying step sizes.

In the second part of the project, we use Gaussian processes to train two different types of models. First, we construct a GP multi-class classifier and investigate the performance with regards to different kernels. Second, we construct a GP regression model based with a 10 dimensional one-hot vector as the output to classify the images. Then we evaluate the performance with regards to different kernels.

In the third and final part of the project one is instead supposed to use convolutional neural networks to perform the classification. We were supposed to compare simple and more complicated networks, try to find a well-performing network without too many trainable parameters and see how the performance depends on the size of the training data.

Mathematical Background

Projected stochastic subgradient descent

Given a convex function G and a subgradient set $\partial_G(\theta)$ evaluated at θ, an iterative method to find the minimum value of G over a closed set C is given by the updates:

$$\theta_{n+1} = \pi_C(\theta_n - \epsilon_n g_n), \tag{1}$$

where g_n is a stochastic subgradient of G such that $\mathbb{E}[g_n] \in \partial_G(\theta)$, $\{\epsilon_n\}_{n\geq 0}$ are some step sizes, and $\pi_C(\cdot)$ is a projection of the argument onto the set C. The algorithm in (1) is called the projected stochastic subgradient descent algorithm.

The **subgradient set** of a function f at a point $x \in \text{dom} f$ is defined as

$$\partial f := \{g : f(y) \geq f(x) + \langle g, y - x\rangle, \ \forall y\}.$$

A **subgradient** of a function f is any $g \in \partial f$. Whenever f is a convex function, g generalizes the derivative of f in non-differentiable points. Where f is differentiable, the subgradient is the derivative (or gradient).

Support vector machines

Support vector machines aim to classify observations into various classes by separating the data belonging to different classes such as the the gap between classes and the separating hyperplane(s) is as large as possible. See figure 1 for a graphical illustration.

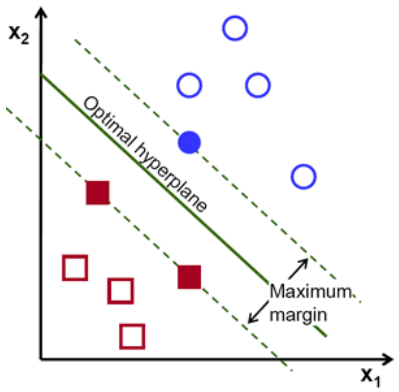

Figure 1: A support vector machine illustrated

One hot encoding

One hot encoding is a process by which (typically) categorical variables (data or labels) are encoded into 0/1-vectors. E.g. $C = 4$, $x_1 = 1$ and $x_2 = 3$, then $OneHot(x_1) = [1, 0, 0, 0]$ and $OneHot(x_2) = [0, 0, 1, 0]$.

Gaussian processes

A Gaussian process (GP) on X is a collection of random variables $\{f(x)\}_{x \in X}$ such that for any finite collection $(x_1, \ldots, x_d) \in X^d$ the random vector $(f(x_1), \ldots, f(x_d))^T$ has a multivariate Normal distribution.

In the regression problem, one can derive posterior of the Gaussian process quite easily without any approximations. However, in order to use GP for non-binary classification more work is needed, more specifically using Laplace approximation. To get around the complicated math for the multi-class classification problem one can instead use several binary one-versus-rest classifiers or one-hot-encode the labels and perform regression on them.

Kernels

Kernels, in the context of Gaussian processes, describe the covariance of GPs. For a given GP $Y \sim \mathcal{GP}(m(x), k(x, x'))$, the kernel function $k(x, x')$ models the covariance between each pair x, x'. In this project, we use the Constant, White noise, RBF, Matern, Rational Quadratic, and Dot product kernels. Their respective kernel function follows below.

The **Constant kernel** is constant for all pairs:

$$k(x, x') = c,$$

where $c \in \mathbb{R}$ is a constant and the hyperparameter of the model.

The **White noise kernel** represents iid noise:

$$k(x, x') = \sigma^2 I_n,$$

where $I_n \in \mathbb{R}^{n \times n}$ is the identity matrix and σ is the hyperparameter.

The **RBF kernel** (radial basis function) is given by:

$$k(x, x') = \exp\left(- \frac{| x - x' |^2}{2\ell^2} \right),$$

where l is the hyperparameter.

The **Matern kernel** is given by:

$$k(x, x') = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} \mid x - x' \mid}{\ell} \right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu} \mid x - x' \mid}{\ell} \right),$$

where l and ν are the hyperparameters.

The **Rational quadratic kernel** is given by:

$$k(x, x') = \sigma^2 \left(1 + \frac{| x - x' |^2}{2\alpha\ell^2} \right)^{-\alpha},$$

where l and α are the hyperparameters.

The **Dot product kernel** is given by:

$$k(x, x') = \sigma^2 + x \cdot x',$$

where σ is the hyperparameter.

Neural networks

Neural networks try to imitate human learning by using several layers of connected artificial neurons with activation funtions to extract features from the input.

In a mathematical sense given the training data $\{(x_i, y_i)\}_{i=1}^n$ the conditional probabilities are modeled as (for a binary classification problem)

$$Pr(Y_i = 1 \mid \Theta = \theta, X_i = x_i) = sigm\big(f_\theta(x_i)\big)$$

$$Pr(Y_i = -1 \mid \Theta = \theta, X_i = x_i) = sigm\big(f_\theta(-x_i)\big)$$

Here f_θ is a composition of several functions, where every component $f_{\theta_i}^{(i)}$ is considered a transition between hidden layers.

$$f_\theta(x) = f_{\theta_m}^{(m)} \circ \cdots \circ f_{\theta_1}^{(1)}(x)$$

The transition $f_{\theta_i}^{(i)}$ has the form

$$f_{\theta_i}^{(i)}(h^{(i-1)}) = g^{(i)}(W^{(i)}h^{(i-1)} + b^{(i)})$$

where $W^{(i)}$ is the weight matrix, $b^{(i)}$ the bias vector and $g^{(i)}$ is an activation function. Training the network is equivalent to optimizing the θ_i $\forall i$ where $\theta_i = (W^{(i)}, b^{(i)})$.

By setting up the weight matrices $W^{(i)}$ and changing the structure of the x_i:s and the $h^{(i)}$:s one get a so called **convolutional network**. Every layer consists of convolution filters that slides along the input features. These types of networks have been shown to perform very well on visual tasks. An illustration is shown in figure 2.

This structure drastically reduces the number of trainable parameters for the typical network since instead of every coordinate in the input image having its own parameter, all coordinates shares the common parameters in the filter.

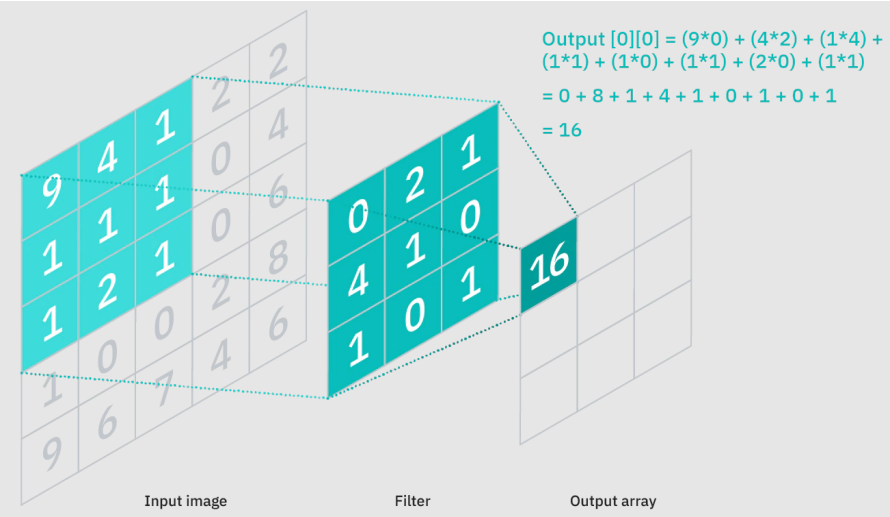

Figure 2: An example of a calculation in a convolutional neural network. Here one value of the output array is calculated using 3×3 values of the input array and the 3×3 filter.

Results

1. Stochastic subgradients for training support vector machines

(a) Show that $R^{\text{emp}}(\Theta) = \frac{1}{n}\sum_{k=1}^{n} L(\Theta, (x_k, y_k))$ is convex where

$$L(\Theta, (x, y)) = \max_{j \neq y} \left[1 + \sum_{i=1}^{d} x_i(\Theta_{ij} - \Theta_{iy})\right]_+$$

Denote $\Theta = \begin{bmatrix}\theta_1 \dots \theta_K\end{bmatrix}$. Then, note that

$$1 + \sum_{i=1}^{d} x_i(\Theta_{ij} - \Theta_{iy}) = 1 + x^\top(\theta_j - \theta_y) =: f(\Theta) \tag{2}$$

is convex, both in θ_j and θ_y, $j \neq y$, since $\nabla_{\theta_j} f(\Theta) = x$, $\nabla_{\theta_y} f(\Theta) = -x$, and $\nabla^\top_{\theta_j} x = \nabla^\top_{\theta_y}(-x) = \mathbf{0}$ (i.e., the function in (2) is linear in θ_j and θ_y and thus convex).

Additionally, the function $f(x) = \max(x_1, x_2, \dots, x_n)$ is convex since for any $\theta \in [0, 1]$ and a fixed $k \in \{1, \dots, n\}$ we have

$$\theta x_k + (1-\theta) y_k \leq \theta \max_i x_i + (1-\theta) \max_i y_i$$

because $x_k \leq \max_i x_i$, $y_k \leq \max_i y_i$ and $\theta \in [0, 1]$. Since this holds for any k we have:

$$\max_k [\theta x_k + (1-\theta) y_k] \leq \theta \max_i x_i + (1-\theta) \max_i y_i$$

For the functions $f : \mathbb{R}^n \to \mathbb{R}$ and $g_i : \mathbb{R}^k \to \mathbb{R}$ such that f is convex and nondecreasing, and each g_i is convex, $i = 1, \dots, n$, it holds that the composition $f \circ g$ is convex, see [1].

Thus, since $t_+ = \max(0, t)$ is convex and nondecreasing, and $1 + x^\top(\theta_j - \theta_y)$ is convex, it holds that $\left[1 + x^\top(\theta_j - \theta_y)\right]_+$ is convex.

Additionally, it holds by the very same argument that

$$\max_{j \neq y} \left[1 + x_k^\top(\theta_j - \theta_{y_k})\right]_+$$

is convex.

Lastly, a conical combination of convex functions is also convex. This implies that

$$\frac{1}{n}\sum_{k=1}^{n} \max_{j \neq y_k} \left[1 + x_k^\top(\theta_j - \theta_{y_k})\right]_+$$

is convex. This concludes the proof.

(b) For a given pair (x, y), representing (image, label), provide an expression for a subgradient G of $R^{\text{emp}}(\Theta)$.

We start by finding an expression for a subgradient of the multi-class hinge loss

$$L(\Theta, (x, y)) = \max_{j \neq y} \left[1 + x^\top(\theta_j - \theta_y)\right]_+.$$

A subgradient of this with regards to θ_j, $j \neq y$, is

$$\begin{aligned}\nabla_{\theta_j} L(\Theta, (x, y)) &= \nabla_{\theta_j} \max_{j \neq y} \left[1 + x^\top(\theta_j - \theta_y)\right]_+ \\ &= \begin{cases} x & \text{if } 1 + x^\top(\theta_j - \theta_y) > \max\limits_{\ell \neq y, j} \left[1 + x^\top(\theta_\ell - \theta_y)\right]_+, \\ \alpha x & \text{if } 1 + x^\top(\theta_j - \theta_y) = \max\limits_{\ell \neq y, j} \left[1 + x^\top(\theta_\ell - \theta_y)\right]_+, \\ 0 & \text{else,} \end{cases}\end{aligned}$$

where $\alpha \in [0, 1]$. For simplicity, we put $\alpha = 0$.

Additionally, a subgradient with regards to θ_y is

$$\begin{aligned}\nabla_{\theta_y} L(\Theta, (x, y)) &= \nabla_{\theta_y} \max_{j \neq y} \left[1 + x^\top(\theta_j - \theta_y)\right]_+ \\ &= \begin{cases} -x & \text{if } \max\limits_{j \neq y} 1 + x^\top(\theta_j - \theta_y) > 0, \\ -\alpha x & \text{if } \max\limits_{j \neq y} 1 + x^\top(\theta_j - \theta_y) = 0, \\ 0 & \text{else,} \end{cases}\end{aligned}$$

where $\alpha \in [0, 1]$. For simplicity, we put $\alpha = 0$.

Next, an expression for the subgradient of the empirical loss function $R^{\text{emp}}(\Theta)$ can be written as follows

$$\nabla_{\theta_\ell} R^{\text{emp}}(\Theta) = \frac{1}{n} \sum_{k=1}^{n} \nabla_{\theta_\ell} L(\Theta, (x_k, y_k)),$$

where each term $\nabla_{\theta_\ell} L(\Theta, (x_k, y_k))$ is computed as above.

To conclude, the subgradient G can be written as

$$G = \nabla_\Theta R^{\text{emp}}(\Theta) = \begin{bmatrix} \nabla_{\theta_1} R^{\text{emp}}(\Theta) & \dots & \nabla_{\theta_K} R^{\text{emp}}(\Theta) \end{bmatrix},$$

where each column vector $\nabla_{\theta_\ell} R^{\text{emp}}(\Theta) \in \mathbb{R}^d$, $\ell = 1, \dots, K$, is computed as above.

Note that evaluating the subgradient G may be computationally costly if n is large. Therefore, we randomly select an index $i \in \{1, \dots, n\}$ and use the $g_i = \nabla_\Theta L(\Theta, (x_i, y_i))$ in the stochastic subgradient algorithm below. It's easily shown that $\mathbb{E}[g_i \mid \Theta_i] \in \partial_R(\Theta_i)$.

An explicit expression of g_i when we only consider a point (x_i, y_i) can easily be derived. Note that $\nabla_{\theta_j} L(\Theta, (x, y))$, $j \neq y_i$, may only be nonzero for one $j = 0, \dots, 9$,

namely the $\arg\max\ 1 + x_i^\top(\theta_j - \theta_{y_i})$, given that the expression is greater than zero. Also, $\nabla_{\theta_y} L(\Theta, (x, y))$ is only nonzero given that $1 + x_i^\top(\theta_j - \theta_{y_i})$ is larger than zero.

Thus, we may conclude that if $L(\Theta, (x_i, y_i)) = 0$, the g_i is the zero matrix. Also, if $L(\Theta, (x_i, y_i)) > 0$, there will only be two nonzero column vectors in g_i. In this case, g_i is given by

$$g_i = \begin{bmatrix} 0 & \dots & 0 & -x & 0 & \dots & x & 0 \end{bmatrix}$$

where the $-x$ term is in the y_ith column (if we start the indexing at zero), and the x term is in the ℓth column, where $\ell = \arg\max_j 1 + x_i^\top(\theta_j - \theta_{y_i})$.

(c) Python 3 code for training the classifier is provided in the file `svmclassifier.py`. Two functions are left un-implemented: `svmsubgradient` and `sgd`. The first function implements the calculation of the subgradient in (b) for a given data point. The function sgd implements a projected stochastic gradient descent method, where you should take the stepsize ϵ_k proportional to $1/\sqrt{k}$ and project Θ to the ball

$$B_r = \{\Theta \in \mathbb{R}^d k : \sum_{ij} \Theta_{ij}^2 \leq r^2\}.$$

Implement the functions `svmsubgradient` and `sgd`.

We set $\Theta_0 = \mathbf{0}$ and use projected SGD to iteratively update

$$\Theta_{n+1} = \pi_{B_r}(\Theta_n - \epsilon_n g_n)$$

where $r = 40$, B_r is defined as above, and $\pi_C(\cdot)$ denotes the projection of the argument onto C.

If $\Theta_n - \epsilon_n g_n \notin B_{40}$, i.e. $\sum_{i,j} \Theta_{i,j}^2 > 40^2$, the projection is given by

$$\pi_{B_{40}}(\Theta_n - \epsilon_n g_n) = (\Theta_n - \epsilon_n g_n)\frac{40}{\sqrt{\sum_{i,j} \Theta_{i,j}^2}}$$

(d) Evaluate the performance of the classifier, using training sets of size 20, 50, 100, 500, 1000, and 1500, and a test set of 100 images.

In general, the test accuracy increases significantly up until a training set size of 500, after which the increase in accuracy is more modest. The overfitting is very large for smaller training set sizes, for instance, for a training set size of 20, the train accuracy is 100%, but only 52% for the test set. The top accuracy achieved is 97% for a training set size of 1500 and 1697.

(e) Investigate other choices of the learning rate, say $\epsilon_k \propto k^{-\alpha}$ for $\alpha \in (0, 1]$, on the performance of the classifier.

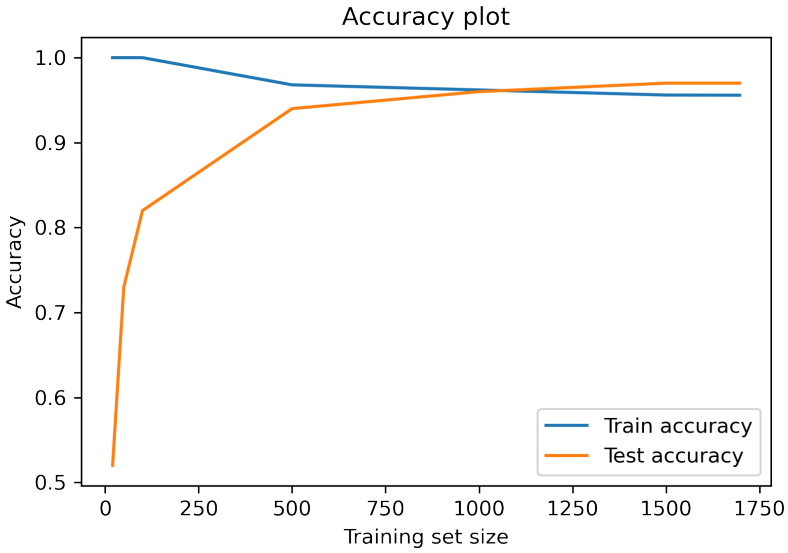

Figure 3: Accuracy plot for different training set sizes

In figure 4, the test and train accuracy is plotted for the learning rate $\epsilon_k \propto k^{-\alpha}$, where $\alpha \in \{0.1, 0.2, \ldots, 0.9, 1\}$. The training set size was 1697. As can be seen in the figure, the is higher for all values on α and peaks at 99%. For $\alpha > 0.8$, the accuracy drops significantly, and reaches a minimum of 91% at $\alpha = 1$.

A rather interesting observation is that the algorithm converges and produces stable and precise models for $\alpha \leq 1/2$. Robbins-Monro requires $\alpha > 1/2$ for convergence w.p.1. (recall that the general RM algorithm requires that $\sum \epsilon_k = \infty$ and $\sum \epsilon_k^2 < \infty$, which implies $\alpha > 1/2$, for almost sure convergence). It can be proven that square summability of the step sizes is required for a.s. convergence. However, for $\alpha \in (0, 1/2]$, the algorithm converges with rate $\mathcal{O}(k^{-\alpha})$.[1]

[1]Francis Bach (2016) `https://www.di.ens.fr/~fbach/orsay2016/lecture3.pdf`

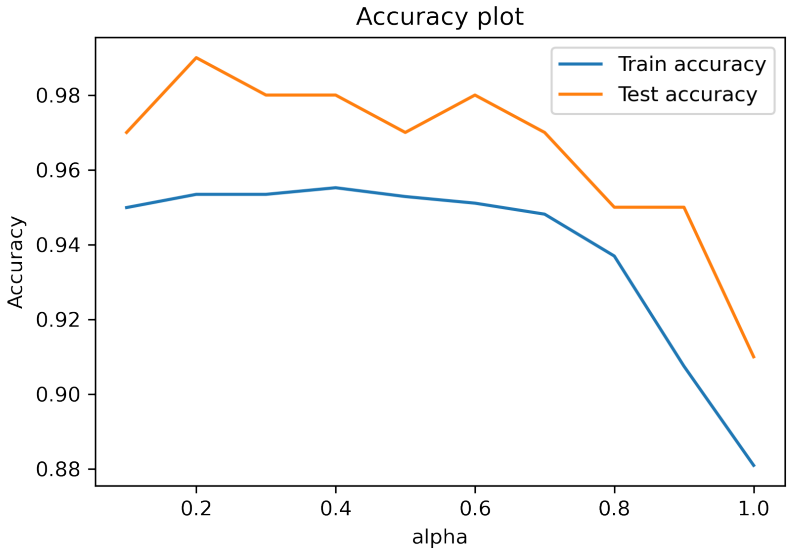

Figure 4: Accuracy plot for varying values on α with training set size 1697

2. Gaussian process classification

(a) In the file `gpclassify.py` a Gaussian process multi-class classifier is implemented for claissifying the digits data set. Note that in the scikit implementation several binary one-versus-rest classifiers are fitted, it does not implement a true multi-class Laplace approximation for GP multi-class classification. Investigate the classification performance of the GP classifier with respect to size of the training data and the choice of covariance function.

The kernels considered in this section are Constant kernal, White kernel, RBF kernel, Matern kernel, Rational Quadratic kernel, and Dot product kernel. For the Constant, White, and RQ kernel, the default hyperparameters are used. For the RBF and Matern kernels, the length scale parameter values of 3 and 2 are used, respectively, to avoid errors. In figure 5, the accuracy is plotted of the GP multi-class classifier for each respective kernel over different training set sizes. No hyperparameter tuning was performed. In figure 6, the hyperparameters have been tuned.

From the two figures, we can conclude that the RBF, Matern, rational quadratic, and dot product kernel perform very well, at least as the training set size increases. The performance of the Constant and White kernel is very poor for all training set sizes. When the hyperparameters were tuned, the GP classifiers with RBF, Matern, and rational quadratic kernels performed excellently, and all got 100% test accuracy for the training set size 1697. The dot product kernel obtained a slightly lower accuracy at 95%.

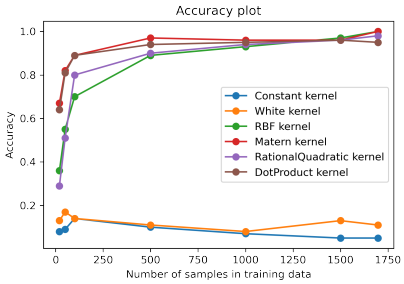

Figure 5: Accuracy plot for the GP multi-class classifier with different kernels

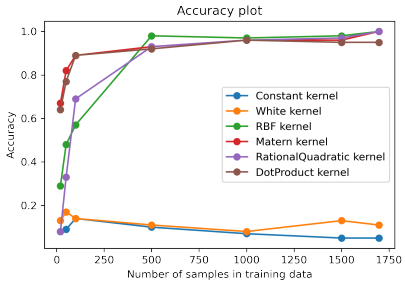

Figure 6: Accuracy plot for the GP multi-class classifier with different kernels with tuned kernel parameters

(b) A rather simple way to produce a multi-class classifier with Gaussian processes is to use Gaussian process regression, as a replacement of Gaussian process classification. Using a one-hot representation of the labels, one can treat the classification problem as a regression problem with a K-dimensional response variable. GP regression is used to predict numerical values of the K-dimensional one-hot-vector and the classification is done by predicting the class with the largest predicted value. Use the functionality in the scikit class `GaussianProcessRegressor` to construct a clas-

sifier based on GP regression and evaluate its performance in terms of size of the training data and choice of the covariance function.

The same kernels as in 2 (a) were also considered here. By using the scikit class `OneHotEncoder` the labels were transformed into one-hot-coded vectors. `GaussianProcessRegressor` was used to regress the images onto these vectors.

As a first test the default values for all kernels were used. The results of this can be seen in figure 7.

The hyperparameters of the kernels were then tuned based on using 1,697 samples as training data. The results can be seen in figure 8.

While the difference between hyperparameter tuning and not are barely distinguishable from the graphs alone, one can see a clear increase in accuracy on the test set when trained on 1,697 samples. I.e. RBF 99% $\rightarrow$ 100% and RationalQuadratic 95% $\rightarrow$ 99%. The tuning of the other kernels did not have an effect on the test accuracy.

Looking at the results one sees that the simple Constant and White kernels perform very poorly. The White kernel implies independence between the function values of the datapoints, which in this case doesn't seem like a valid assumption. Similarly the constant kernel assumes that all covariances are constant. While this allows more complicated functions, the results show that a more complicated covariance structure is needed in order to capture the information in the images.

The RBF, Matern and RationalQuadratic kernels all perform very well and their results are very similar. The latter is not very surprising, the Matern kernel is a generalization of the RBF kernel and the RationalQuadratic kernel can be seen as a scale mixture of RBF kernels.

The DotProduct kernel also performs well, but not as well as the RBF kernels, indicating it's not as suitable for this problem.

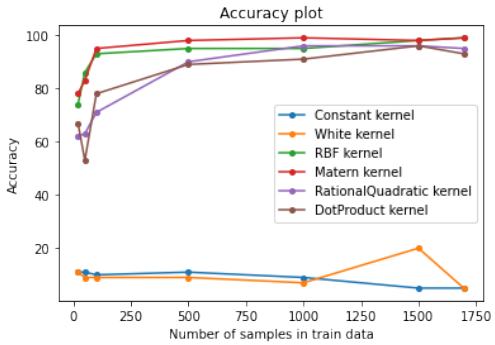

Figure 7: Accuracy plot for the GaussianProcessRegressor with different kernels.

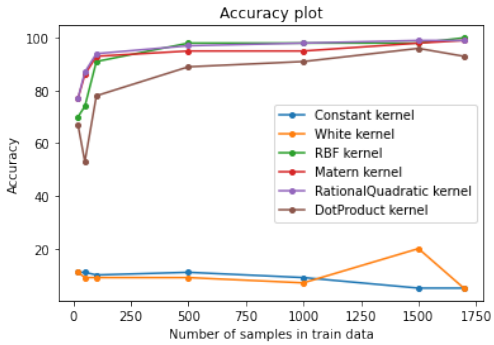

Figure 8: Accuracy plot for the GaussianProcessRegressor with tuned kernels.

3. Convolutional neural network classification

In this part the python package `pytorch` was used to create and train the convolutional networks. The networks were trained using the minibatch gradient descent (MGD) algorithm with a batchsize of 16. The optimizer used was *Adam*.

(a) Build and train a simple convolutional neural network model (of one or two convolutional layers) to classify the images in the digits data set. Evaluate its performance.

The simple networks consists of one convolutional layer (as defined in table 1) and one fully connected layer with size 1024×10. This architecture results in a total of 10,410 parameters.

Layer	Convolutional layer	Max-pooling layer
1 (input)	c:16, k:(3,3), s:(1,1), p:(1,1)	-

Table 1: Specification of the simple CNN. c: number of filters, k: filter size, s: stride size, p:padding size, po: pooling size.

The network was trained for 30 epochs with a learning rate of 0.005. The results can be seen in figure 11 and 12. The final accuracy was 97% on the test set and 100% on the training set. This accuracy is already very high, but adding more power to the network should yield even higher performance.

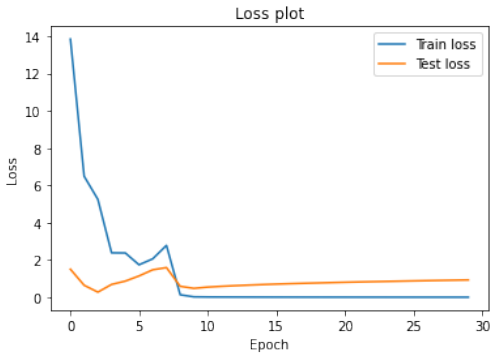

Figure 9: Loss plot for the simple CNN.

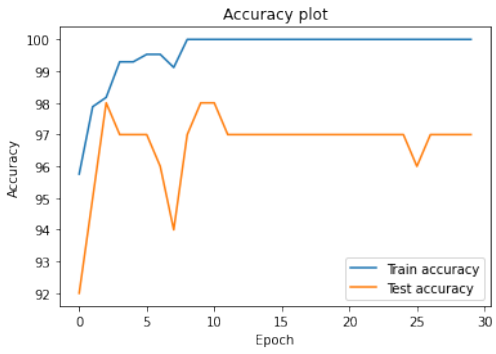

Figure 10: Accuracy plot for the simple CNN.

(b) Investigate the performance of the convolutional neural network in terms of the size of the training data and the design of the layers. Try to find a network with as few parameters as possible, without sacrificing performance.

First we will explore the network's architecture. By adding three more layers the network will get more predictive power. To combat the overfitting potential and the increase in number of trainable parameters, we also add dropout and max-pooling layers. The full network is defined in table 2. The final fully connected layer has the size 784×10.

Layer	**Convolutional layer**	**Max-pooling layer**	**Dropout prob.**
4	c:16, k:(2,2), s:(1,1), p:(1,1)	-	0.2
3	c:16, k:(2,2), s:(1,1), p:(1,1)	-	0.2
2	c:32, k:(2,2), s:(1,1), p:(1,1)	po:(2,2), s:(2,2), p:(0,0)	0.2
1 (input)	c:64, k:(2,2), s:(1,1), p:(1,1)	-	0.2

Table 2: Specification of the CNN encoder. c: number of filters, k: filter size, s: stride size, p:padding size, po: pooling size.

The network was trained for 100 epochs with a learning rate of 0.003. The results can be seen in figure 11 and 12. The final accuracy was 99% on the test set and 100% on the training set. We notice that we get higher accuracy on the test set with this model.

The model now has 19,498 trainable parameters, almost double compared to the simple model. However, the increase in number of parameters significantly increases the performance on the test set. We also explored networks with even more parameters, specifically with an increase number of filters. Even though these networks achieved a smaller loss, the performance did not increase on the test set due to overfitting.

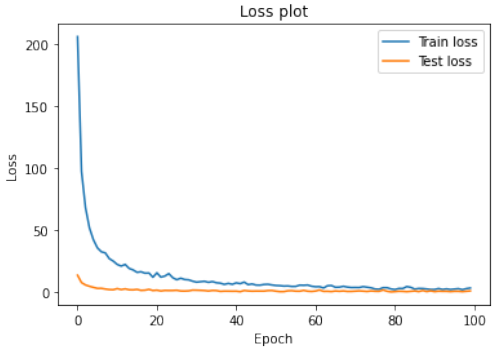

Figure 11: Loss plot for the complicated CNN.

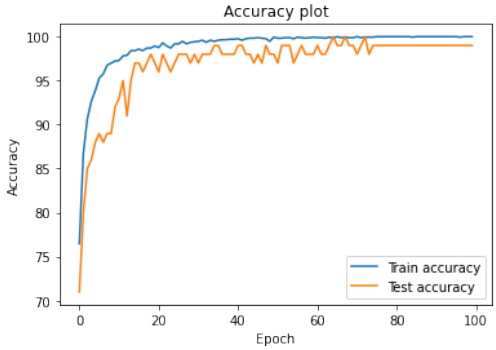

Figure 12: Accuracy plot for the complicated CNN.

We will now also take a look a how the size of the training data affects the performance. We now train the complicated CNN with different sizes of the training set ($n \in \{100, 200, 350, 500, 750, 1000, 1250, 1500\}$) and observe how that affects the perfor-

mance. Notice that we now look at the average loss in order to be able to compare the different results.

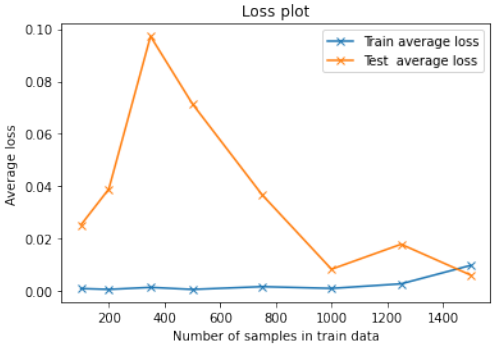

Figure 13: Loss plot for different sizes of the training data with the complicated CNN.

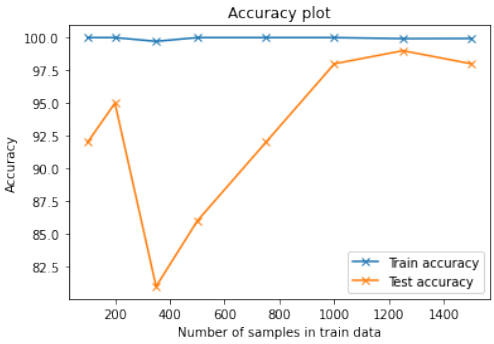

Figure 14: Accuracy plot for different sizes of the training data with the complicated CNN.

The results can be seen in figure 13 and 14. All networks were trained for 50 epochs with a learning rate of 0.002.

Apart from training sizes < 400 and $= 1500$ we notice that the average test loss decreases and the test accuracy increases with the training size. This is due to model being able to learn and generalize better with more data. However, the opposite seems to be true for the training set. This can be explained by the increased training size acting as a regularizer, the network now has to take more images into account and generalize over them. This should in turn reduce overfitting, which is what we observe on both the loss and accuracy plot where the test and train plots get closer.

Summary

All four methods used to classify handwritten digits from the MNIST dataset performed well given that suitable hyperparameters, kernels, and training set sizes were used. The results of the tested models are summarized in table 3.

Looking at the model types one by one, we firstly notice that the SVM performs very well and achieves a top accuracy of 97% for $\alpha = 1/2$ and 99% for $\alpha = 0.2$.

Secondly, the two types of Gaussian processes both perform well when using the RBF or RBF-like kernels. These models performed the best overall, achieving an accuracy of 100% on the test set.

Lastly, CNNs also perform well on this task. Without using too many trainable parameters very high accuracies were achieved.

However, one should note that most models (of those included in the table) achieve a very high test accuracy (99%-100%) which makes them hard to compare. In order to be able to distinguish the performance of the models better we could have done two things. The first was to use a larger test set, only having 100 samples easily leads to

Model type	**Model specifics**	**Test accuracy**
SVM	Projected SGD	97%
SVM	Projected SGD + adjusted learning rate	98%
GP Multi-Classifier	RBF kernel	100%
GP Multi-Classifier	Matern kernel	100%
GP Multi-Classifier	Rational Quadratic kernel	100%
GPRegressor	RBF kernel	100%
CNN	Complicated 4-layer CNN (see table 2)	99%

Table 3: The test accuracy of the best models in this project. These are the results for when the models where trained on 1,697 samples and tested on 100.

identical accuracies. The second was to run several tests per model and record both average accuracy and standard deviation.

References

[1] S. Boyd, S. P. Boyd, and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.