Gaussian process classifiers and CNN uncertainty

Sebastian Borgeaud dit Avocat
spb61@cam.ac.uk

LE49 - Probabilistic Machine Learning Project

Abstract

The abstract paragraph should be indented ½ inch (3 picas) on both the left- and right-hand margins. Use 10 point type, with a vertical spacing (leading) of 11 points. The word **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the abstract. The abstract must be limited to one paragraph.

1 Introduction

Model uncertainty is often of crucial importance.

Softmax output does not give model confidence, maybe show the example used by Yarin Gal.

With model uncertainty it is possible to treat uncertain inputs and special cases explicitly. Example if model uncertainty is high we might decide to pass the input to a human for classification.

Another example is in Reinforcement learning: with uncertainty a RL agent can decided when to explore and when to exploit in an environment. With principled uncertainty for the agent's Q-value function, it can learn much faster using techniques such as Thompson sampling.

Yarin Gal shows how dropout in neural networks can be interpreted as a Bayesian approximantion of a Gaussian Process.

1.1 Convolutional neural networks [1] (ADD MORE CITATIONS)

Convolutional neural networks (CNNs) were originally inspired by biological processes. They have become standard in many deep learning applications, especially in image processing or vision tasks. A convolutional neural network is a type of feedforward neural network, typically consisting of convolution layers, pooling layers and fully connected layers:

- **Convolution layers** are composed of several convolution kernels each computing a different feature map. The output feature maps are obtained by convolving the input with the convolution kernel and then applying an element-wise nonlinearity. Mathematically, the feature value $z^l_{i,j,k}$ at location (i, j) of the k^{th} feature map in the l^{th} layer is computed as:

$$z^l_{i,j,k} = {\mathbf{w}^l_k}^T \mathbf{x}^l_{i,j} + b^l_k$$

  where $\mathbf{w}^l_k$ and b^l_k are the weight and bias vectors for the k^{th} convolution kernel in the l^{th} layer and $\mathbf{x}^l_{i,j}$ is the input patch centered around (i, j) in the l^{th} layer. The ouput value is computed by apply a nonlinearity $a(\cdot)$ point-wise:

$$x^{(l+1)}_{i,j,k} = a(z^l_{i,j,k})$$

- **Pooling layers** aim to achieve shift-invariance and reduce the number of parameters in the network by reducing the resolution of the feature maps. The pooling layer operates on each

feature map independently. Mathematically, the output of a pooling layer with pooling operation pool$(\cdot)$ is given by

$$y^l_{i,j,k} = \text{pool}(x^l_{m,n,k}), \forall (m,n) \in \mathcal{R}_{i,j}$$

where $\mathcal{R}_{i,j}$ is a local neighbourhood around (i,j). Typically, the pooling operation computes the average or the maximum.

- **Fully connected layers** connect every neuron in the previous layer to single neuron in the current layer. Fully connected layers are the layers used in standard neural networks. Mathematically, the output of a fully connected layer is given by:

$$x^{(l+1)}_i = a\Big((\sum_j w^l_{i,j} x^l_j) + b^l_i \Big)$$

where $a(\cdot)$ is a nonlinearity, $w^l_{i,j}$ is the weight connecting neuron j in the l^{th} layer to neuron i in layer $(l+1)^{\text{th}}$, and b^l_i is the bias weight for neuron i.

Optimising a convolutional neural network is done in the same way as optimising standard neural networks. A differentiable loss function is computed for the training examples (often done in batches) and the gradients w.r.t. the weights of the network are computed. Using these gradients, the weights are updated in a gradient descent step. Typically, more complex update rule that take into account change momentum (e.g. Adam [2]) are used as they converge faster to a local minimum.

1.2 Gaussian processes

Formally, a Gaussian Process is defined as a collection of random variables, any finite number of which have (consistent) joint Gaussian distributions. A Gaussian process therefore defines a distribution over functions and is fully specified by a mean function $m(x)$ and a covariance function $k(x,x')$. Write $f \sim \mathcal{GP}(m,k)$ meaning f is distributed as a GP with mean m and covariance k.

Using the GP we can draw samples from the function for any finite number n of locations. Given locations $\mathbf{x} = [x_1, \ldots, x_n]$, first compute $\mu_i = m(x_i)$, $\Sigma_{i,j} = k(x_i, x_j)$. We can then sample a vector from this distribution: $\mathbf{f} \sim \mathcal{N}(\mu, \mathbf{\Sigma})$.

1.2.1 Regression

We can now use this GP as a prior for Bayesian inference. Let $\mathbf{f}$ be the known function values for the training examples an let $\mathbf{f}_*$ be the set of function values corresponding to the set of test input X_*. The joint distribution is given by

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} = \mathcal{N}\Big(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} \mathbf{\Sigma} & \mathbf{\Sigma}_* \\ \mathbf{\Sigma}_*^T & \mathbf{\Sigma}_{**} \end{bmatrix} \Big)$$

where μ_* are the test means, $\mathbf{\Sigma}_*$ are the training-test covariances, and $\mathbf{\Sigma}_{**}$ are the test-test covariances. Since we know the training values $\mathbf{f}$, we are interested in the conditional distribution of $\mathbf{f}_*$ given $\mathbf{f}$:

$$\mathbf{f}_* \big| \mathbf{f} \sim \mathcal{N}\big(\mu_* + \mathbf{\Sigma}_*^T \mathbf{\Sigma}^{-1}(\mathbf{f} - \mu), \mathbf{\Sigma}_{**} - \mathbf{\Sigma}_*^T \mathbf{\Sigma}^{-1} \mathbf{\Sigma}_*\big)$$

This corresponds to a posterior Gaussian process, given by

$$\begin{aligned} f \big| \mathcal{D} &\sim \mathcal{GP}(m_\mathcal{D}, k_\mathcal{D}), \\ m_\mathcal{D}(x) &= m(x) + \Sigma(X,x)^T \Sigma^{-1}(\mathbf{f} - \mathbf{m}) \\ k_\mathcal{D}(x,x') &= k(x,x') - \Sigma(X,x)^T \Sigma^{-1} \Sigma(X,x') \end{aligned}$$

where $\Sigma(X,x)$ is a vector of covariances between every training case in X and x. Furthermore, it is easy to incorporate noise in the observations. Assuming i.i.d. additive Gaussian noise, every $f(x)$ now has extra covariance with itself with a magnitude equal to the noise variance σ_n^2:

$$f \big| \mathcal{D} \sim \mathcal{GP}(m_\mathcal{D}, k_\mathcal{D} + \delta_{ii} \sigma_n^2)$$

where $\delta_{ii'} = 1$ iff $i = i'$ is the Kronecker's delta.

The mean function $m(x)$ and the covariance function $k(x, x')$ are typically parametrised in terms of hyper-parameters $\boldsymbol{\theta}$. During training we find the values of the hyper-parameters which optimise the marginal likelihood:

$$L = \log p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\theta}) = -\frac{1}{2}|\Sigma| - \frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{y} - \boldsymbol{\mu}) - \frac{n}{2}\log(2\pi)$$

This optimisation can be done using standard gradient methods.

1.2.2 Classification

Binary classification using Gaussian processes can be done by setting a GP prior over a latent function $f(\mathbf{x})$ and then using squashing function such as the sigmoid to obtain a probability:

$$\pi(\mathbf{x}) = p(y = +1|\mathbf{x}) = \sigma(f(\mathbf{x}))$$

Inference is done in two steps. First, compute the distribution of the latent variable corresponding to a new test input $\mathbf{x}_*$

$$p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int p(f_*|\mathbf{X}, \mathbf{x}_*, \mathbf{f}) p(\mathbf{f}|\mathbf{X}, \mathbf{y}) d\mathbf{f}.$$

Second, use this distribution to compute a probabilistic prediction

$$\bar{\pi}_* = p(y_* = +1|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) = \int \sigma(f_*) p(f_*|\mathbf{X}, \mathbf{y}, \mathbf{x}_*) df_*$$

As the likelihood is non longer Gaussian, the first integral becomes analytically intractable. Similarly, depending on the sigmoid function, the second integral can also be intractable. Hence, we need to use approximations, either analytical or numerical, for example using Monte Carlo sampling, to solve the integrals.

Multi-class classification is typically [3] approached by assuming the following labelling rule for y_* given $\mathbf{x}_*$:

$$y_* = \arg\max_k f^k(\mathbf{x}_*), \text{ for } k = 1, \ldots, C$$

where each $f^k(\cdot)$ is a nonlinear latent function with a GP prior. The likelihood is again non-Gaussian meaning that approximation techniques have to be used to perform inference and to optimise the hyper-parameters.

TODO: Explain how classification is done with GPs, explain the difficulties. Say something about approximation techniques?

1.3 Uncertainty in Deep Learning

Obtaining uncertainty bounds in deep learning would be without doubt of great benefit to many fields. An autonomous car could make much safer decisions knowing how much uncertainty there is in the consequence of taking different actions. When using current deep learning methods we only have point estimates of parameters and predictions (CITE DL BOOK). Despite the fact the output of a softmax layer in a classification task can be interpreted as a probability distribution $p(y_i = k) = f(x_i)_k$, it doesn't say anything about how certain the model is: The model could assign a high probability to a class but can still be highly uncertain about it (CITE GAL).

One way to get uncertainty from deep learning models was discovered by Yarin Gal (GAL THESIS). In fact, there is a rather subtle but deep connection between Gaussian Process and neural networks. In particular, he showed that using Dropout not only at training time but also at test time, and obtaining multiple estimates for a single test example allows us to compute the uncertainty in a principled manner by taking the sample variance of those predictions (CITATION).

In this project, I focus on a different way to obtain those uncertainty bounds: Using a Gaussian process trained on the features extracted by a CNN that was trained in a previous step. It should be noted that this is a less theoretically grounded approach than the one taking by Gal (CITE), but nonetheless provides a practical way of getting uncertainty whilst keeping the power of neural networks.

2 Method

2.1 Convolutional Neural Network architecture

The first step consists in training a convolutional neural network on the MNIST dataset, which can be done using one of the many deep learning libraries. For example, I use Keras (CITATION) which defines an extra abstraction layer above TensorFlow. The network architecture is provided in the Keras tutorial for image classifcation on MNIST. The first two layers are convolutional layers 3×3 kernels and ReLU activations, where $\text{ReLU}(x) = \max(x, 0)$. The first layer has 32 feature maps and the second layer has 64. A max-pooling layer with kernel size 2×2 is then applied to the output of the convolutional layer. The final 2 layers are fully connected layers, with hidden sizes of 128 and 10 respectively. The first fully connected layer has a ReLU activation. The last fully connected layer uses a softmax activation, which outputs a probability distribution over the 10 classes representing the 10 digits. Furthermore, Dropout (CITATION) is applied after the max-pooling layer with $p = 0.25$ and after the first fully connected layer with $p = 0.5$, where p is the probability of dropping a neuron. The network is trained using an Adadelta optimiser over 10 epochs with batches of size 128.

2.2 Gaussian process

Using the trained CNN model, we can extract the activations in the last hidden layer. These correspond to a vector of size 128 for each input image, and can be thought of as the features extracted by the CNN for classification. I then train a Gaussian Process using these features. This has the advantage of being a much lower dimensional classification task compared to classifying the entire input image directly, which has dimension $28 \times 28 = 784$. However, the MNIST dataset, with 60,000 training images, is still considered a large dataset for GPs because inference takes $\mathcal{O}(n^3)$ time where n is the number of training instances [4]. To make the training possible, I use GPFlow [5], which implements various approximation algorithms for Gaussian Processes. Furthermore, the library is built on top of TensorFlow which has the further advantage of being usable on a GPU out-of-the-box, which provides a further speed-up. More precisely, I use the Sparse Variational Gaussian Process Classifier presented by Hensman et al. [4]. The model learns a set of inducing points of size m which are then used instead of the training points. Typically, $m < n$, which makes the task tractable as its complexity is $\mathcal{O}(nm^2)$. For the experiments done in this project I chose to use 600 inducing points, which allowed the model to be trained in about 15 minutes on a NVidia Titan Xp GPU.

I train the Gaussian process with a simple kernel as it is sufficient to obtain good accuracy. The kernel consists of a Matern kernel with $\nu = \frac{3}{2}$ (CITATION) and a white noise function to account for the noise in the input data.

TODO: Add figure of the model...

2.3 Evaluation

2.3.1 Accuracy

The standard metric for evaluating a model f on a classification task is the accuracy, which can be computed as follows:

$$\text{accuracy} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(f(\mathbf{x}^{(i)}) = y^{(i)})$$

where N is the number of test data points and $(\mathbf{x}^{(i)}, y^{(i)})$ is the i^{th} test point. Note that the accuracy does not take into account the uncertainty with which a model makes the prediction and only rewards correctly classified examples.

2.3.2 Reject option

One way to incorporate the model uncertainty into an evaluation metric is by allowing the model to say "I don't know". For example, if the model was presented with an image of the letter 'a', it would still classify the image into one of the 10 digits, which might not be a desired outcome. Instead, the model should be able to reject classification for the inputs in which the uncertainty is too high. This is known as **classification with a reject option** [6]. An appropriate metric assigns a cost α to

misclassification and a cost β to the reject option ®:

$$\text{®-accuracy}_{\alpha,\beta} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(f(\mathbf{x}^{(i)}) = y^{(i)}) - \alpha\mathbb{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)}) - \beta\mathbb{1}(f(\mathbf{x}^{(i)}) = ®)$$

Here, α and β are parameters that are set by the user. A large value for α corresponds to cases where misclassification is expensive, for example in medical diagnosis or when a self-driving car has to make a steering decision. The β parameter intuitively indicates how much value the user places on obtaining a prediction, as a higher β will increase the cost of the reject option.

Now, given uncertainty bounds with each class probability, we can design the model to reject classification according to certain rejection rules. For example, we could decided to reject if the standard deviation of the highest probability is above a pre-defined threshold. Another rejection rule could decide to reject if the second highest class probability lies within a certain factor ϵ of the standard deviations of the highest probability, i.e. the model would reject if

$$|f_p(\mathbf{x}^{(i)})_1 - f_p(\mathbf{x}^{(i)})_2| < \epsilon f_{\text{std}}(\mathbf{x}^{(i)})_1$$

where $f_p(\mathbf{x}^{(i)})$ is the classification probability vector for input $\mathbf{x}^{(i)}$ with $f_p(\mathbf{x}^{(i)})_j$ being defined as the probability of the j^{th} most likely class, and $f_{\text{std}}(\mathbf{x}^{(i)})_1$ is the standard deviation of the most likely class returned by the model for input $\mathbf{x}^{(i)}$. Note that the ϵ parameter defines a trade-off between the number of misclassified examples at the number of examples that would have been correctly classified but were rejected instead. For the results presented next, I focus on this decision rule with different values for ϵ.

3 Results

3.1 MNIST dataset

The MNIST dataset is consider a toy dataset in deep learning and is generally seen as a "unit test" for models. In fact, using the model provided in the Keras tutorial and presented above, I obtain an accuracy of 99.12% on the held-out test images. This means that only 88 of the $10,000$ test images are misclassified.

Using the Gaussian process with the CNN features, the accuracy of the model on the test images improves to 99.28%, reducing the number of misclassified images to 72. Although this increase in performance might seem surprising at first, it can be explained by the fact that the Gaussian process is a more complex model than the last layer of the CNN, which just computes a linear combination of the features and then applies the softmax activation.

Model	**Accuracy**	**Misclassified images**
CNN	99.12%	88
GP	99.28%	72

Table 1: MNIST test images accuracy for the CNN and the GP.

Furthermore, we can inspect the uncertainty the Gaussian process assigns to different test images. The model gives the correct class probability with a standard deviation of less than 0.1 for 97.3% of the correctly classified images. For the incorrectly classified images, the most standard deviation of the most likely class is below 0.1 for only 19.4% of the images. This suggests that the model is generally less certain of its prediction in the cases it got the classification wrong. The cumulative distribution of standard deviations is shown in figure 1. The plot supports the claim that the model is less certain for the misclassified images as the standard deviation for the most likely class is higher for a larger proportion of those images that were misclassified.

3.2 Noisy MNIST (n-MNIST)

Next, I evaluate both the CNN and the Gaussian process on the n-MNIST dataset [7]. The dataset consists of the MNIST images, but modified in 3 different ways so as to make the classification task

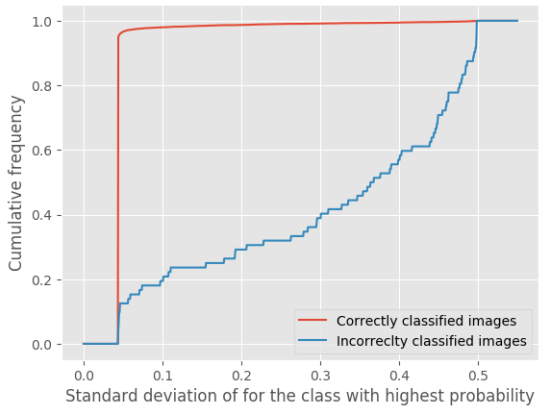

Figure 1: Cumulative distribution of standard deviations for correctly and incorrectly classified test images. The standard deviation of the most likely class is taken for both correctly and incorrectly classified images.

harder. The first set of images is created by adding white Guassian noise with a signal-to-noise ratio of 9.5 to the images. The second set of images is obtained by emulating a linear motion blur. The last set of images is the hardest to classify as it is created by reducing the contrast and also adding white Guassian noise to the original images. Figure 2 shows an example image taken from the 3 datasets.

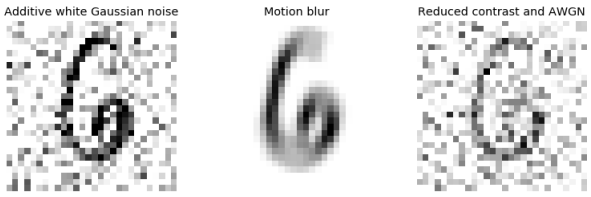

Figure 2: Example digits from the noisy-MNIST dataset. From left to right: Additive white Guassian noise, Motion blur, and Reduced contrast with AWGN

The task of classifying examples whose underlying distribution is different from the underlying distribution of the training images is called **out-of-distribution** classification (CITE GAL DISSERTATION). The performance of the two models is shown in table 2. Although the GP performs better on the NMIST test images, the CNN performs better on all 3 noisy datasets. This could be explained by the fact that the GP is a more complex model, and therefore is overfitting more on the NMIST data. Especially noticeable is the difference in performance for the reduced contrast dataset, where the CNN obtains an accuracy of 77.71% compared to 70.13% for the GP. As the 3$^{\text{rd}}$ dataset is the least similar the original one, this higher discrepancy supports the claim that the GP is overfitting more on the MNIST dataset.

Model	**Accuracy**		
	AWGN	Motion blur	Reduced contrast
CNN	94.35%	93.27%	77.71%
GP	93.34%	92.09%	70.13%

Table 2: MNIST test images accuracy for the CNN and the GP.

We can again inspect how uncertain the model is when classifying the noisy dataset, by plotting the cumulative distributions of the standard deviations of the most likely digit for the 3 datasets. Figure 3 shows the resulting plots, which indeed suggest that the uncertainty increases as the model is asked to classify images that are increasingly different images from those it was trained on. The uncertainty for the additive white Guassian noise (AWGN) images is higher than the uncertainty for the standard

MNIST test set. This is shown by the fact that the cumulative distribution for the correctly classified images in the AWGN images is lower. Similarly, the cumulative distribution further decreases for the motion blur and reduced contrast images, with a lower classification accuracy corresponding to a higher uncertainty.

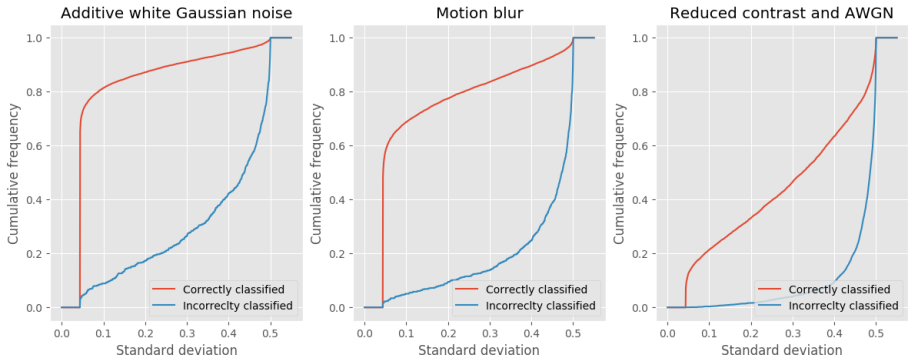

Figure 3: Cumulative distribution of standard deviations for correctly and incorrectly classified test images. The standard deviation of the most likely class is taken for both correctly and incorrectly classified images.

3.3 Classifying with a reject option

3.4 Uncertainty with additive white noise

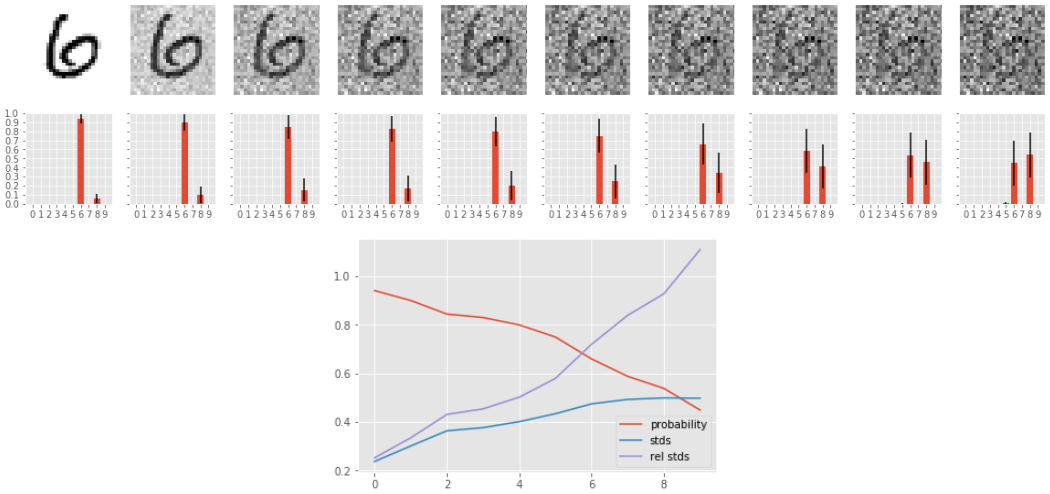

Figure 4: Cumulative distribution of standard deviations for correctly and incorrectly classified test images. The standard deviation of the most likely class is taken for both correctly and incorrectly classified images.

3.5 Uncertainty with rotated images

TODO: Use rotation of an image, e.g. Yarin Gal

4 Discussion

More quantitave evalution of the results.

Show examples of images that are missclassified.

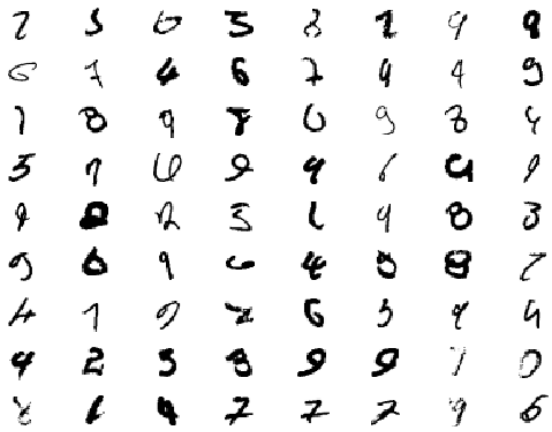

Figure 5: Test images misclassified images by the GP model.

Discuss definition of the metrics.

TODO: ADD VISUALLISATION of n-MNIST features

4.1 CNN features visualisation

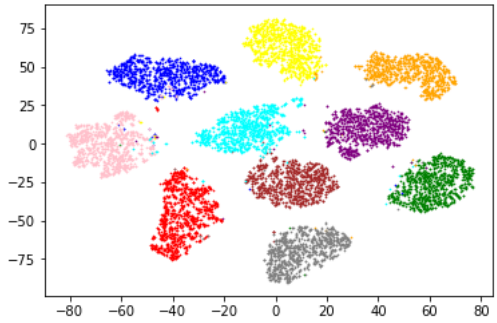

Figure 6: Test point embedding using t-SNE

References

[1] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, and Gang Wang. Recent advances in convolutional neural networks. *CoRR*, abs/1512.07108, 2015.

[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[3] Carlos Villacampa-Calvo and Daniel Hernández-Lobato. Scalable multi-class gaussian process classification using expectation propagation. *arXiv preprint arXiv:1706.07258*, 2017.

[4] James Hensman, Alexander G de G Matthews, and Zoubin Ghahramani. Scalable variational gaussian process classification. 2015.

[5] Alexander G. de G. Matthews, Mark van der Wilk, Tom Nickson, Keisuke. Fujii, Alexis Boukouvalas, Pablo Le‘on-Villagr‘a, Zoubin Ghahramani, and James Hensman.

[6] C Chow. On optimum recognition error and reject tradeoff. *IEEE Transactions on information theory*, 16(1):41–46, 1970.

[7] Saikat Basu, Manohar Karki, Sangram Ganguly, Robert DiBiano, Supratik Mukhopadhyay, and Ramakrishna R. Nemani. Learning sparse feature representations using probabilistic quadtrees and deep belief nets. *CoRR*, abs/1509.03413, 2015.