A. Experimental Details

This section contains all info to reproduce our experiments. In all experiments, we initialise the centroids using draws from $N(0, 0.05I_n)$. All convolutional layers are initialized using the default in PyTorch 0.4.2 based on (He et al., 2015).

A.1. Two Moons

We set the noise level to 0.1 and generate 1000 points for our training set. Our model consists of three layers with 20 hidden units each, the embedding size is 10. We use the relu activation function and standard SGD optimiser with learning rate 0.01, momentum 0.9 and L2 regularisation with weight 10^{-4}. Our batch size is 64 and we train for a set 30 epochs. We set the length scale to 0.3, γ to 0.99 and λ to 1.0.

A.2. FashionMNIST

We use a model consisting of three convolutional layers of 64, 128 and 128 3x3 filters, with a fully connected layer of 256 hidden units on top. The embedding size is 256. After every convolutional layer, we perform batch normalization and a 2x2 max pooling operation.

We use the SGD optimizer with learning rate 0.05 (decayed by a factor of 5 every 10 epochs), momentum 0.9, weight decay 10^{-4} and train for a set 30 epochs. The centroid updates are done with $\gamma = 0.999$. The output dimension of the model, d is 256, and we use the same value for the size of the centroids, n.

We normalise our data using per channel mean and standard deviation, as computed on the training set. The validation set contains 5000 elements, removed at random from the full 60,000 elements in the training set. For the final results, we rerun on the full training set with the final set of hyper parameters.

A.3. CIFAR-10

We use a ResNet-18, as implement in torchvision version 0.4.2[4]. We make the following modifications: the first convolutional layer is changed to have 64 3x3 filters with stride 1, the first pooling layer is skipped and the last linear layer is changed to be 512 by 512.

We use the SGD optimizer with learning rate of 0.05, decayed by a factor 10 every 25 epochs, momentum of 0.9, weight decay 10^{-4} and we train for a set 75 epochs. The centroid updates are done with $\gamma = 0.999$. The output dimension of the model, d is 512, and we use the same value for the size of the centroids n.

[4]Available online at: https://github.com/pytorch/vision/tree/v0.4.2

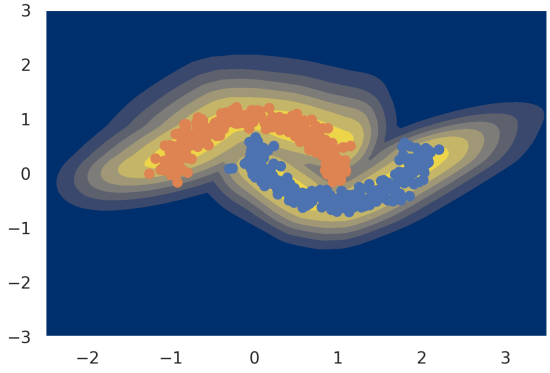

Figure 9. Uncertainty results on two moons data set. Yellow means certain, while blue indicates uncertainty. The model is a reversible feature extractor in combination with the kernel based output as in DUQ.

We normalise our data using per channel mean and standard deviation, as computed on the training set. We augment the data at training time using random horizontal flips (with probability 0.5) and random crops after padding 4 zero pixels on all sides. The validation set contains 10,000 elements, removed at random from the full 50,000 elements in the training set. For the final results, we rerun on the full training set with the final set of hyper parameters.

B. Deep Ensemble Uncertainty

The uncertainty in Deep Ensembles is measured as the entropy of the average predictive distribution:

$$\hat{p}(y|x) = \frac{1}{N}\sum_{i=1}^{N} p_{\theta_i}(y|x)$$

$$H(\hat{p}(y|x)) = -\sum_{i=0}^{C} \hat{p}(y_i|x)\log \hat{p}(y_i|x),$$

with θ_i the set of parameters for ensemble element i.

C. Gradient Penalty Alternatives

C.1. Reversible Models

An alternative method of enforcing sensitivity is by using an invertible feature extractor. A simple and effective method of doing so is by using invertible layers originally introduced in (Dinh et al., 2014). Using these type of layers leads to strong results on the two moons dataset as seen in Figure 9. Unfortunately, it is difficult to train reversible models on higher dimensional data sets. Without dimensionality reduction, such as max pooling, the memory usage of these type of networks is unreasonably high (Jacobsen et al.,

2018) . We found it impossible to obtain strong accuracy and uncertainty using these type of models, indicating that dimensionality reduction is an important component of why these models work.

C.2. Gradient Penalty

Empirically, we found that computing the penalty on $\nabla_x \sum_c K_c$ works well. However there are two other candidates to enforce the penalty on: $\nabla_x K_c$, the vector of kernel distances and $\nabla_x f_\theta(x)$, the feature vector output of the feature extractor. At first sight, these targets might actually be preferential. Sensitivity of $f_\theta(x)$ ought to be sufficient to obtain the out of distribution sensitivity properties we desire.

Computing the Jacobian of a vector valued output is expensive using automatic differentiation. To evaluate the two alternative candidates we turn to the Hutchinson's Estimator (Hutchinson, 1990), which allows us to estimate the trace of the Jacobian by computing the derivative of random projections of the output. This approach was previously discussed in the context of making neural networks more robust by Hoffman et al. (2019).

While we were able to get good uncertainty on the two moons data set using both alternative targets, the results were not consistent. We attempted to reduce the variance of the Hutchinson's estimator by using the same random projection for each element in the batch, which worked well on two moons, but lead to unsatisfactory results on larger scale data sets. In conclusion, we found that while $\nabla_x \sum_c K_c$ is not a priori the best place to compute the gradient penalty, it is still preferable over the noise that comes from applying Hutchinson's estimator on any of the alternatives, at least in our experiments.