Uncertainty Estimation Practical Part I

There are many ways to estimate predictive uncertainty in deep learning. However, in this practical, we focus on simple yet effective methods. One such method that has achieved impressive performance is the Deep Ensembles method[1]. In this practical, we first study this method in Part I, then a recent extension in Part II.

Simple and Scalable Predictive Uncertainty Estimation Using Deep Ensembles -- Lakshminarayanan B, Pritzel A, Blundell C. [1]

Abstract:

Deep neural networks (NNs) are powerful black box predictors that have recently achieved impressive performance on a wide spectrum of tasks. Quantifying predictive uncertainty in NNs is a challenging and yet unsolved problem. Bayesian NNs, which learn a distribution over weights, are currently the state-of-the-art for estimating predictive uncertainty; however these require significant modifications to the training procedure and are computationally expensive compared to standard (non-Bayesian) NNs. We propose an alternative to Bayesian NNs that is simple to implement, readily parallelizable, requires very little hyperparameter tuning, and yields high quality predictive uncertainty estimates. Through a series of experiments on classification and regression benchmarks, we demonstrate that our method produces well-calibrated uncertainty estimates which are as good or better than approximate Bayesian NNs. To assess robustness to dataset shift, we evaluate the predictive uncertainty on test examples from known and unknown distributions, and show that our method is able to express higher uncertainty on out-of-distribution examples. We demonstrate the scalability of our method by evaluating predictive uncertainty estimates on ImageNet.

Notations

Assume we have i.i.d. training data points $(\boldsymbol{x}, y)$, where $\boldsymbol{x} \in \mathbb{R}^D$ is the D-dimensional input features. And for a classification problem with K classes, $y \in \{1, \ldots, K\}$; for a regression problem, $y \in \mathbb{R}$. We use a NN with parameters θ to model the probablistic predictive distribution $p_\theta(y|\boldsymbol{x})$ over the labels.

Proper scoring rules

Scoring rules measure the quality of predictive uncertainty [2]. Suppose we have $(y, \boldsymbol{x})$ following the true distribution $q(y, \boldsymbol{x})$, and we want to evaluate the quality of the predictive distribution, $p_\theta(y|\boldsymbol{x})$. The scoring rule, a function, $S(p_\theta, (y, \boldsymbol{x}))$, assigns a numerical score to the predictive distribution $p_\theta(y|\boldsymbol{x})$. Here we consider scoring rules where a high score means better quality. The expected scoring rule is then $S_{\mathbb{E}}(p_\theta, q) = \int q(y, \boldsymbol{x}) S(p_\theta, (y, \boldsymbol{x})) dy d\boldsymbol{x}$. A *proper scoring rule* is, for all p_θ and q, $S_{\mathbb{E}}(p_\theta, q) \leq S_{\mathbb{E}}(q, q)$ with equality if and only if $p_\theta(y|\boldsymbol{x}) = q(y|\boldsymbol{x})$. Since a larger value $S_{\mathbb{E}}(p_\theta, q)$ means a better quality of the predictive uncertainty, we could train NNs by minimizing the loss $\mathcal{L}(\theta) = -S_{\mathbb{E}}(p_\theta, q)$.

Log-likelihood, $\log p_\theta(y|\boldsymbol{x})$, turns out to be a proper scoring rule due to [Gibbs inequality]:

$$S_{\mathbb{E}}(p_\theta, q) = \mathbb{E}_{q(\boldsymbol{x})} q(y|\boldsymbol{x}) \log p_\theta(y|\boldsymbol{x}) \leq \mathbb{E}_{q(\boldsymbol{x})} q(y|\boldsymbol{x}) \log q(y|\boldsymbol{x}). \tag{1}$$

Thus, minimizing the negative log-likelihood (NLL), which is equivalent to the softmax cross entropy loss in classification, is a proper scoring rule. Interestingly minimizing the squared error between the predictive probability of a label and one-hot encoding of the correct label, is also a proper scoring rule known as the Brier score. In regression, if we assume p_θ to be Gaussian $\mathcal{N}(\mu_\theta(\boldsymbol{x}), \sigma_\theta^2(\boldsymbol{x}))$:

========================

TODO: Derive the NLL loss for Gaussian assumption.

========================

$$\begin{align} -\log p_\theta(y|\boldsymbol{x}) = \tfrac{1}{2}\log\big(2\pi\,\sigma_\theta^2(\boldsymbol{x})\big) \tag{2} \\ + \frac{\big(y - \mu_\theta(\boldsymbol{x})\big)^2}{2\sigma_\theta^2(\boldsymbol{x})} \tag{3} \end{align}$$

Derivation :

Derivation of NLL for Gaussian Assumption **Goal:** Derive the Negative Log-Likelihood (NLL) loss function starting from the Gaussian PDF.

We assume the predictive distribution $p_\theta(y|x)$ is a Gaussian distribution $\mathcal{N}(\mu_\theta(x), \sigma_\theta^2(x))$. The probability density function (PDF) is:

$$p(y|x) = \frac{1}{\sqrt{2\pi\sigma_\theta^2(x)}} \exp\left(-\frac{(y - \mu_\theta(x))^2}{2\sigma_\theta^2(x)}\right)$$

To obtain the NLL, we take the negative natural logarithm ($-\log$) of the PDF:

$$-\log p(y|x) = -\log\left(\frac{1}{\sqrt{2\pi\sigma_\theta^2(x)}}\exp\left(-\frac{(y-\mu_\theta(x))^2}{2\sigma_\theta^2(x)}\right)\right)$$

Using the logarithm rules $\log(ab) = \log(a) + \log(b)$ and $\log(a^k) = k\log(a)$:

$$-\log p(y|x) = -\left[\log((2\pi\sigma_\theta^2(x))^{-1/2}) + \log\left(\exp\left(-\frac{(y-\mu_\theta(x))^2}{2\sigma_\theta^2(x)}\right)\right)\right]$$

$$-\log p(y|x) = -\left[-\frac{1}{2}\log(2\pi\sigma_\theta^2(x)) - \frac{(y-\mu_\theta(x))^2}{2\sigma_\theta^2(x)}\right]$$

Distributing the negative sign yields the final loss function:

$$\mathcal{L}_{Gaussian} = \frac{1}{2}\log(2\pi\sigma_\theta^2(x)) + \frac{(y-\mu_\theta(x))^2}{2\sigma_\theta^2(x)}$$

Of course we can model the p_θ to be other distributions, e.g. [Laplacian distribution], which corresponds to L1 loss, MAE, rather than Gaussian distribution, which corresponds to L2 loss, MSE. If we assume p_θ to be Laplacian $\mathcal{L}(\mu_\theta(\boldsymbol{x}), \sigma_\theta(\boldsymbol{x}))$, with mean $\mu_\theta(\boldsymbol{x})$ and variance $\sigma_\theta(\boldsymbol{x})$:

========================

TODO: Derive the NLL loss for Laplacian assumption.

========================

$$-\log p_\theta(y|\boldsymbol{x}) = \log(2\sigma_\theta(\boldsymbol{x})) + \frac{|y-\mu_\theta(\boldsymbol{x})|}{\sigma_\theta(\boldsymbol{x})} \tag{4}$$

Derivation of NLL for Laplacian Assumption

Goal: Derive the Negative Log-Likelihood (NLL) loss function starting from the Laplacian PDF.

We assume the predictive distribution $p_\theta(y|x)$ is a Laplacian distribution $\mathcal{L}(\mu_\theta(x), \sigma_\theta(x))$. The probability density function (PDF) is:

$$p(y|x) = \frac{1}{2\sigma_\theta(x)}\exp\left(-\frac{|y-\mu_\theta(x)|}{\sigma_\theta(x)}\right)$$

We take the negative natural logarithm:

$$-\log p(y|x) = -\log\left(\frac{1}{2\sigma_\theta(x)}\exp\left(-\frac{|y-\mu_\theta(x)|}{\sigma_\theta(x)}\right)\right)$$

Using logarithm rules:

$$-\log p(y|x) = -\left[\log((2\sigma_\theta(x))^{-1}) + \log\left(\exp\left(-\frac{|y-\mu_\theta(x)|}{\sigma_\theta(x)}\right)\right)\right]$$

$$-\log p(y|x) = -\left[-\log(2\sigma_\theta(x)) - \frac{|y-\mu_\theta(x)|}{\sigma_\theta(x)}\right]$$

Adversarial training

Adversarial examples (see [3] and [4]) are those which are very close to the original training examples but are misclassified by the NN. In [4], the authors proposed the *fast gradient sign method* as a fast solution to generate adversarial examples. Given $(\boldsymbol{x}, y)$ and the loss $\ell(\theta, \boldsymbol{x}, y)$, we can use the fast gradient sign method to generate an adversarial example as $\boldsymbol{x}' = \boldsymbol{x} + \epsilon\text{sign}\big(\nabla_{\boldsymbol{x}}\ell(\theta, \boldsymbol{x}, y)\big)$, where ϵ is a small value to bound the perturbation. The procedure to use this adversarial example as an additional training sample to augument the training set, referred to as *adversarial training* was found to improve the classifier's robustness [4].

There are more than one way to generate adversarial examples, for example, *virtual adversarial training* (VAT)[5], which does not require to know the true target y, could be used in *semi-supervised learning* (SSL).

Ensembles

In general there are two classes of ensembles: *randomization*-based approaches and *boosting*-based approaches. Here we only focus on randomization-based approaches since it is more suitable for parallel computation. In practice, random initialization of the NN parameters θ and with random shuffling of the data points during training were found to be sufficient to obtain good performance. During inference, the ensemble is treated as a uniformly-weighted mixture model and the predictions are combined as $p(y|\boldsymbol{x}) = M^{-1}\sum_{m=1}^{M} p_{\theta_m}(y|\boldsymbol{x}, \theta_m)$. For classification, this corresponds to averaging the predicted probabilities. For regression, if we assume a Gaussian distribution over label y given point $\boldsymbol{x}$, i.e. $p(y|\boldsymbol{x}) = \mathcal{N}(\mu, \sigma)$ for some μ and σ, then the prediction's distribution is a mixture of Gaussian distributions (as we have one Gaussian output for each model in the ensemble). We then approximate the ensemble prediction as a Gaussian, whose mean and variance are respectively the mean and variance of the mixture:

$$\mu_*(\boldsymbol{x}) = M^{-1}\sum_{m=1}^{M}\mu_{\theta_m}(\boldsymbol{x}) \tag{5}$$

$$\sigma_*^2(\boldsymbol{x}) = M^{-1}\sum_{m=1}^{M}(\sigma_{\theta_m}^2(\boldsymbol{x}) + \mu_{\theta_m}^2(\boldsymbol{x})) - \mu_*^2(\boldsymbol{x}). \tag{6}$$

And if we assume a Laplacian distribution over label y given input $\boldsymbol{x}$, then the mean and variance of the mixture is:

$$\mu_*(\boldsymbol{x}) = M^{-1}\sum_{m=1}^{M}\mu_{\theta_m}(\boldsymbol{x}) \tag{7}$$

$$\sigma_*^2(\boldsymbol{x}) = M^{-1}\sum_{m=1}^{M}(2\sigma_{\theta_m}^2(\boldsymbol{x}) + \mu_{\theta_m}^2(\boldsymbol{x})) - \mu_*^2(\boldsymbol{x}). \tag{8}$$

Regression on toy dataset

In [1]:

```
!pip install -U matplotlib
```

```
Requirement already satisfied: matplotlib in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (3.10.0)
Collecting matplotlib
  Downloading matplotlib-3.10.7-cp310-cp310-macosx_11_0_arm64.whl.metadata (11 kB)
Requirement already satisfied: contourpy>=1.0.1 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matp
lotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matplotl
ib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from mat
plotlib) (4.55.3)
Requirement already satisfied: kiwisolver>=1.3.1 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from mat
plotlib) (1.4.7)
Requirement already satisfied: numpy>=1.23 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matplotli
b) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matpl
otlib) (24.2)
Requirement already satisfied: pillow>=8 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matplotlib)
(10.4.0)
Requirement already satisfied: pyparsing>=3 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from matplotl
ib) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from
matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/envs/torch/lib/python3.10/site-packages (from python-dateu
til>=2.7->matplotlib) (1.17.0)
Downloading matplotlib-3.10.7-cp310-cp310-macosx_11_0_arm64.whl (8.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 8.1/8.1 MB 14.5 MB/s eta 0:00:00a 0:00:01
Installing collected packages: matplotlib
  Attempting uninstall: matplotlib
    Found existing installation: matplotlib 3.10.0
    Uninstalling matplotlib-3.10.0:
      Successfully uninstalled matplotlib-3.10.0
Successfully installed matplotlib-3.10.7
```

In [5]:

```
import jax
import jax.numpy as jnp
from jax import grad, jit, vmap, value_and_grad
from jax import random
import matplotlib.pyplot as plt
from jax.scipy.special import logsumexp
import numpy as np
from jax.example_libraries import optimizers
import time
import sys
```

Note that for the sake of numerical stability, we assume that the prediction variance is modeled as follows:

$$\sigma = \log(1 + \exp(\rho)) + \epsilon \tag{9}$$

$$= \mathrm{softplus}(\rho) + \epsilon \tag{10}$$

where ρ is the model output in $\mathbb{R}$, and σ is in $(\epsilon, +\infty)$. Usually, ϵ, a very small number, is used to avoid division by zero.

In [6]:

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1e-2):
    w_key, b_key = random.split(key)
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))

# Initialize all layers for a fully-connected neural network with sizes "sizes"
def init_network_params(sizes, key, scale):
    keys = random.split(key, len(sizes))
    return [random_layer_params(m, n, k, scale) for m, n, k in zip(sizes[:-1], sizes[1:], keys)]

def predict(params, inputs):
    # per-example predictions
    activations = inputs
    for w, b in params[:-1]:
        outputs = jnp.dot(w, activations) + b
        activations = jax.nn.relu(outputs)

    final_w, final_b = params[-1]
    scores = jnp.dot(final_w, activations) + final_b

    mean, rho = jnp.split(scores, 2)

    variance = jax.nn.softplus(rho) + 1e-6
    return mean, variance # for NLL loss
    # return scores # for MSE loss

batched_predict = vmap(predict, in_axes=(None, 0))

def loss(params, inputs, targets):
    preds = batched_predict(params, inputs)
    return jnp.mean(jnp.square(preds - targets))

def loss_l1(params, inputs, targets):
    preds = batched_predict(params, inputs)
    return jnp.mean(jnp.absolute(preds - targets))

def NLLloss_gaussian(params, inputs, targets):
    """
    Negative log-likelihood loss function.
    ==============================
    The ``inputs`` argument and ``targets`` of this function are both of shape (N,1)=(#examples, 1).
    1. Feed forward the data to NN to obtain mean and variance (use batched_predict function)
    2. Calculate the negative log-likelihood using the outputs of the NN, based on Gaussian assumption (the loss sh
    """
    mean, sigma = batched_predict(params, inputs)
    var = jnp.square(sigma)
    nll = 0.5 * jnp.log(2 * jnp.pi * var) + jnp.square(targets - mean) / (2 * var)
    return jnp.mean(nll)

def NLLloss_laplacian(params, inputs, targets):
    """
    Negative log-likelihood loss function.
    ==============================
    The ``inputs`` argument and ``targets`` of this function are both of shape (N,1)=(#examples, 1).
    1. Feed forward the data to NN to obtain mean and variance (use batched_predict function)
    2. Calculate the negative log-likelihood using the outputs of the NN, based on Laplacian assumption (the loss s
    """
    mean, sigma = batched_predict(params, inputs)
    nll_loss = jnp.log(2 * sigma) + jnp.abs(targets - mean) / sigma
    return jnp.mean(nll_loss)

def NLLlossAT_gaussian(params, inputs, targets, eps=0.08):
    """
    Negative log-likelihood loss function with adversarial training
    ==============================
    The ``inputs`` argument and ``targets`` of this function are both of shape (N,1)=(#examples, 1).
    1. Use the ``value_and_grad`` function together with the
       ``NLLloss_gaussian`` function you implemented to get
       the loss value and the gradients with respect to the ``inputs``.
       Note that you also need to pass ``argnums`` to ``value_and_grad``.
    2. Generate adversarial examples. Note that you need to call jnp.array
       on gradients to be able to compute the sign.
        So use jnp.sign(jnp.array(grads)) instead of jnp.sign(grads).
    3. Calculate the NLL loss based on the adversarial examples and average the two losses
    """
    nll, grads = value_and_grad(NLLloss_gaussian, argnums=1)(params, inputs, targets)
    inputs_ad = inputs + eps * jnp.sign(jnp.array(grads))
    nll_ad = NLLloss_gaussian(params, inputs_ad, targets)
    return (nll + nll_ad)/2

def NLLlossAT_laplacian(params, inputs, targets, eps=0.08):
    """
    Negative log-likelihood loss function with adversarial training
    ==============================
    The ``inputs`` argument and ``targets`` of this function are both of shape (N,1)=(#examples, 1).
    1. Use the ``value_and_grad`` function together with the
       ``NLLloss_laplacian`` function you implemented to get
       the loss value and the gradients with respect to the ``inputs``.
       Note that you also need to pass ``argnums`` to ``value_and_grad``.
    2. Generate adversarial examples. Note that you need to call jnp.array
       on gradients to be able to compute the sign.
        So use jnp.sign(jnp.array(grads)) instead of jnp.sign(grads).
    3. Calculate the NLL loss based on the adversarial examples and average the two losses
    """
    nll, grads = value_and_grad(NLLloss_laplacian, argnums=1)(params, inputs, targets)
    inputs_ad = inputs + eps * jnp.sign(jnp.array(grads))
    nll_ad = NLLloss_laplacian(params, inputs_ad, targets)
    return (nll + nll_ad)/2

@jit
def update(params, x, y, opt_state):
    """ Compute the gradient for a batch and update the parameters """
    # value, grads = value_and_grad(loss)(params, x, y)

    value, grads = value_and_grad(NLLloss_gaussian)(params, x, y)
    # value, grads = value_and_grad(NLLlossAT_gaussian)(params, x, y)

    # value, grads = value_and_grad(NLLloss_laplacian)(params, x, y)
    # value, grads = value_and_grad(NLLlossAT_laplacian)(params, x, y)

    opt_state = opt_update(0, grads, opt_state)
    return get_params(opt_state), opt_state, value

def data_set(points=20, xrange=(-4, 4), std=3.):
    xx = jnp.array([[np.random.uniform(*xrange)] for i in range(points)])
    yy = jnp.array([[x**3 + np.random.normal(0, std)] for x in xx])
    return xx.reshape(-1,1), yy.reshape(-1,1)
```

In [7]:

```
# network parameters

layer_sizes = [1, 10, 6, 30, 2]

# param_scale = 0.02
num_epochs = 600
step_size = 8e-2

# to ensure reproducibility
np.random.seed(173)
xx, yy = data_set(points=40, xrange=(-4, 4), std=3.) # generate data set of 20 samples
x = np.linspace(-6, 6, 100).reshape(-1, 1)
y = x**3
x = jnp.array(x)
y = jnp.array(y)

# number of ensembles
M = 6
params_all = [None]*M

opt_state_all = [None]*M
mean_all, var_all = [], []
opt_init, opt_update, get_params = optimizers.adam(step_size)

for i in range(M):
    # use random seed to randomly initialize the NN
    SEED = 100+abs(int(np.random.normal(0,1)*10000))

    np.random.seed(SEED)
    param_scale = abs(np.random.normal(0,0.1))
    print("----------training network "+str(i)+"------------param_scale "+str(param_scale))
    params_all[i] = init_network_params(layer_sizes, random.PRNGKey(i*10), param_scale)

    opt_state_all[i] = opt_init(params_all[i])

    log_acc_train, log_train_loss = [], []

    for epoch in range(num_epochs):
        start_time = time.time()

        params_all[i] = get_params(opt_state_all[i])
        params_all[i], opt_state_all[i], train_loss = update(params_all[i], xx, yy, opt_state_all[i])

        if epoch == 0:
            print("initial loss: ",train_loss)
        epoch_time = time.time() - start_time

        log_train_loss.append(train_loss)

    print("final loss: ",train_loss)

    mean, var = batched_predict(params_all[i], x)

    mean_all.append(mean)
    var_all.append(var)

mean_all = np.array(mean_all)
var_all = np.array(var_all)

mean_ = mean_all.mean(axis=0)
# Variance of the Gaussian mixture model (see derivation above)
component_second_moment = var_all ** 2 + mean_all ** 2
var_ = component_second_moment.mean(axis=0) - mean_ ** 2
std_ = np.sqrt(var_)
```

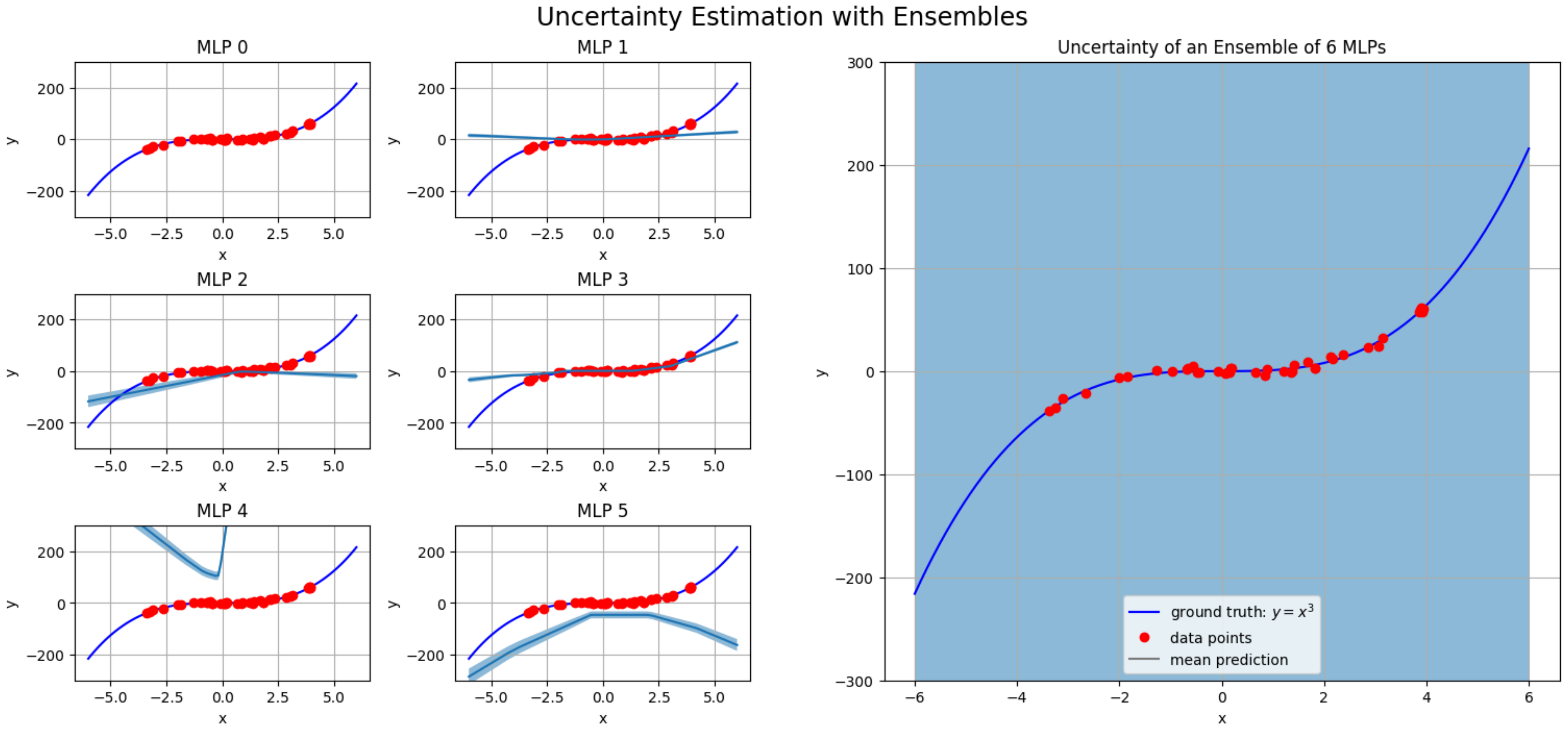
```
----------training network 0------------param_scale 0.05341477341995916
initial loss:  624.8802
final loss:  9.232912
----------training network 1------------param_scale 0.12899350046504218
initial loss:  864.7967
final loss:  3.532995
----------training network 2------------param_scale 0.06450740460784611
initial loss:  544.54504
final loss:  5.225137
----------training network 3------------param_scale 0.1859958185502153
initial loss:  532.7735
final loss:  3.0285718
----------training network 4------------param_scale 0.021813948079358225
initial loss:  617.4408
final loss:  8.246596
----------training network 5------------param_scale 0.17893712367557665
initial loss:  685.33264
final loss:  6.4121294
```

In [8]:

```python
# visualize ensemble output
fig = plt.figure(constrained_layout=True,
                 figsize=(15, 7),
                 dpi=100)
fig.suptitle('Uncertainty Estimation with Ensembles',
             fontsize='xx-large')
subfigs = fig.subfigures(1, 2, wspace=0.07)
axs_left = subfigs[0].subplots(3, 2)
for i,ax in zip(range(M),axs_left.flat):
    ax.plot(x, y, "b-")
    ax.plot(xx,yy,"or")
    ax.plot(x, mean_all[i])
    ax.fill_between(x.reshape(-1,), (mean_all[i]-np.sqrt(var_all[i])).reshape(-1,), (mean_all[i]+np.sqrt(var_all[i]
    ax.grid()
    ax.set_title("MLP " + str(i))
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_ylim((-300,300))

axs_right = subfigs[1].subplots(1, 1)
axs_right.plot(x, y, "b-", label="ground truth: $y=x^3$")
axs_right.plot(xx,yy,"or", label="data points")
axs_right.plot(x, mean_, label="mean prediction", color="grey")
axs_right.fill_between(x.reshape(-1,), (mean_-std_).reshape(-1,), (mean_+std_).reshape(-1,),alpha=0.5)
axs_right.grid()
axs_right.set_title("Uncertainty of an Ensemble of "+str(M)+" MLPs")
axs_right.set_xlabel("x")
axs_right.set_ylabel("y")
axs_right.set_ylim((-300,300))
axs_right.legend()
```

Out[8]:

```
<matplotlib.legend.Legend at 0x168d453c0>
```



Discussion points:

- In terms of regression, compared to only use NLL loss with a single network, what is the effect of adding Adversarial Training? How about adding ensembles of M networks?
- How is the number of networks, M, affecting the prediction and uncertainty estimation?

###########

1. In terms of regression, compared to only use NLL loss with a single network, what is the effect of adding Adversarial Training?

Effect : It significantly improves robustness and prevents overfitting to the noise in the training data.

- **Smoothing:** A single network trained with standard NLL often produces "overconfident" uncertainty estimates (very low variance) in regions slightly off the data manifold.
- **Robustness:** AT generates perturbed examples (x') that maximize loss and trains on them. This forces the network to smooth its predictive distribution, resulting in higher, more realistic uncertainty estimates in the immediate vicinity of the data points[cite: 53, 54].

2. How about adding ensembles of M networks?

Effect: Ensembles allow us to estimate **Epistemic Uncertainty** (model uncertainty).

- **Capturing Uncertainty:** A single network can only reliably capture aleatoric uncertainty (noise in the data). An ensemble captures the uncertainty of the model itself.
- **OOD Detection:** In regions where training data is sparse (Out-of-Distribution), the random initialization of each network m causes them to converge to different functions. While they may agree on the training data, they will disagree wildly in OOD regions.
- **Result:** This disagreement leads to a high variance in the mixture distribution, effectively flagging unknown regions [cite: 60-64].

3. How is the number of networks, M, affecting the prediction and uncertainty estimation?

Effect: Increasing M improves approximation quality but with diminishing returns.

- **Accuracy:** Generally, a larger M leads to a more accurate mean prediction as it averages out errors from individual models.
- **Uncertainty Quality:** As M increases, the ensemble distribution $p(y|x)$ better approximates the true posterior predictive distribution. However, empirical studies (including the Deep Ensembles paper) suggest that a small number (e.g., $M = 5$) is often sufficient to obtain well-calibrated uncertainty estimates.

Classification on toy dataset :

The NLL loss for categorical assumption is:

$$\begin{aligned}
-\log p_\theta(y|\boldsymbol{x}) &= -\log(\prod_{i=1}^{k} p^{y_i}) && (11)\\
&= -\sum_{i=1}^{k} \log(p_i^{y_i}) && (12)\\
&= -\sum_{i=1}^{k} y_i \log(p_i) && (13)\\
&= -\log(p_i) && (14)
\end{aligned}$$

where, for notational simplicity we assume $y = [y_1, y_2, \ldots, y_k]$ is the one-hot encoded representation of the random variable y.

In [9]:

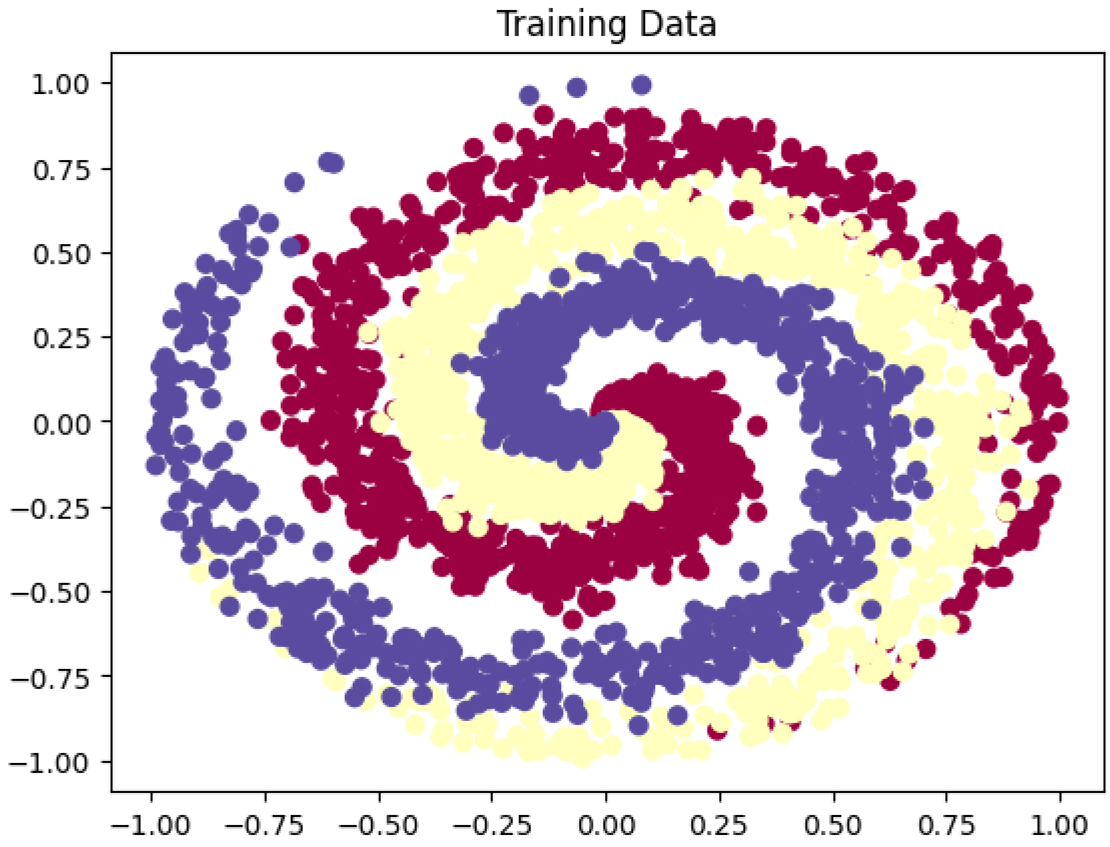
```
def one_hot(x, k, dtype=jnp.float32):
    """Create a one-hot encoding of x of size k."""
    return jnp.array(x[:, None] == jnp.arange(k), dtype)

def generate_datasets(N,K,noise):
    X = np.zeros((N*K,D)) # data matrix (each row = single example)
    y = np.zeros(N*K, dtype='uint8') # class labels

    for j in range(K):
        ix = range(N*j,N*(j+1))
        r = np.linspace(0.0,1.,N) # radius
        t = np.linspace(j*8,(j+1)*8,N) + np.random.randn(N)*noise * (r+1.0) # theta
        print(j, np.amin(t), np.amax(t))
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        y[ix] = j
    X = jnp.array(X)
    y = jnp.array(y)
    y_onehot = one_hot(y, K)
    return X, y, y_onehot

def shuffle_dataset(X, y, y_onehot, seed):
    indices = jax.random.permutation(seed, jnp.arange(X.shape[0]))
    X_shuffled = jnp.take(X, indices, axis=0)
    y_oh_shuffled = jnp.take(y_onehot, indices, axis=0)
    y_shuffled = y[indices]
    return X_shuffled, y_shuffled, y_oh_shuffled
```

In [10]:

```
N = 1000 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes
```

```
np.random.seed(0)
X, y, y_onehot = generate_datasets(N, K, noise=0.3)

# lets visualize the data:
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.title("Training Data")
print("Training X:", X.shape)
print("Training y:", y.shape)
```

```
0 -0.6210700567572351 9.160761050479454
1 7.506810982132371 16.88345105834359
2 15.192239293196302 25.21297645733773
Training X: (3000, 2)
Training y: (3000,)
```



In [11]:

```
# A helper function to randomly initialize weights and biases
# for a dense neural network layer
def random_layer_params(m, n, key, scale=1e-1):
    w_key, b_key = random.split(key)
    return scale * random.normal(w_key, (n, m)), scale * random.normal(b_key, (n,))

# Initialize all layers for a fully-connected neural network with sizes "sizes"
def init_network_params(sizes, key):
    keys = random.split(key, len(sizes))
    return [random_layer_params(m, n, k) for m, n, k in zip(sizes[:-1], sizes[1:], keys)]

@jit
def predict(params, image):
    # per-example predictions
    activations = image
    for w, b in params[:-1]:
        outputs = jnp.dot(w, activations) + b
        activations = jax.nn.relu(outputs)

    final_w, final_b = params[-1]
    scores = jnp.dot(final_w, activations) + final_b
    probs = jax.nn.softmax(scores)
    return scores, probs

batched_predict = vmap(predict, in_axes=(None, 0))

def accuracy(params, inputs, targets):
    target_class = jnp.argmax(targets, axis=1)
    scores, _ = batched_predict(params, inputs)
    predicted_class = jnp.argmax(scores, axis=1)
    return jnp.mean(predicted_class == target_class)

def NLLloss_categorical(params, inputs, targets):
    scores, preds = batched_predict(params, inputs)
    log_prob = scores - logsumexp(scores, axis=1)[:,None]
    return -jnp.mean(log_prob * targets)

def weight_decay_regularization(params):
    return jax.example_libraries.optimizers.l2_norm(params)

def loss(params, inputs, targets,wd):
    return NLLloss_categorical(params,
                               inputs,
                               targets) +            wd * weight_decay_regularization(params)

def NLLlossAT_categorical(params, inputs, targets, eps=0.02):
    """
    Negative log-likelihood loss function with adversarial training
    ==============================
    The ``inputs`` argument and ``targets`` of this function are both of shape (N,1)=(#examples, 1).
    1. Use the ``value_and_grad`` function together with the
       ``NLLloss_categorical`` function you implemented to get
       the loss value and the gradients with respect to the ``inputs``.
       Note that you also need to pass ``argnums`` to ``value_and_grad``.
    2. Generate adversarial examples. Note that you need to call jnp.array
       on gradients to be able to compute the sign.
       So use jnp.sign(jnp.array(grads)) instead of jnp.sign(grads).
    3. Calculate the NLL loss based on the adversarial examples and average the two losses
    """
    loss_ori, grads = value_and_grad(NLLloss_categorical, argnums=1)(params, inputs, targets)
    inputs_ad = inputs + eps * jnp.sign(jnp.array(grads))
    loss_ad = NLLloss_categorical(params, inputs_ad, targets)
    return (loss_ori + loss_ad)/2

def lossAT(params, inputs, targets,wd):
    return NLLlossAT_categorical(params,
                                 inputs,
                                 targets) +            wd * weight_decay_regularization(params)

@jit
def update(params, x, y, opt_state, wd):
    """ Compute the gradient for a batch and update the parameters """
    value, grads = value_and_grad(loss)(params, x, y, wd)
    # value, grads = value_and_grad(lossAT)(params, x, y, wd)

    opt_state = opt_update(0, grads, opt_state)
    return get_params(opt_state), opt_state, value

def logging(params_all, X, y, K, M, log_acc_train, log_nll_train):

    probs = jnp.mean( jnp.array([batched_predict(params_all[i], X)[1] for i in range(M)]), axis=0)

    y_pred = jnp.argmax(probs,axis=1)
    train_acc = jnp.mean(y_pred == y)

    y_conf = probs[np.arange(y.shape[0]),y]
    train_nll = -jnp.mean(jnp.log(y_conf))

    log_acc_train.append(train_acc)
    log_nll_train.append(train_nll)
```

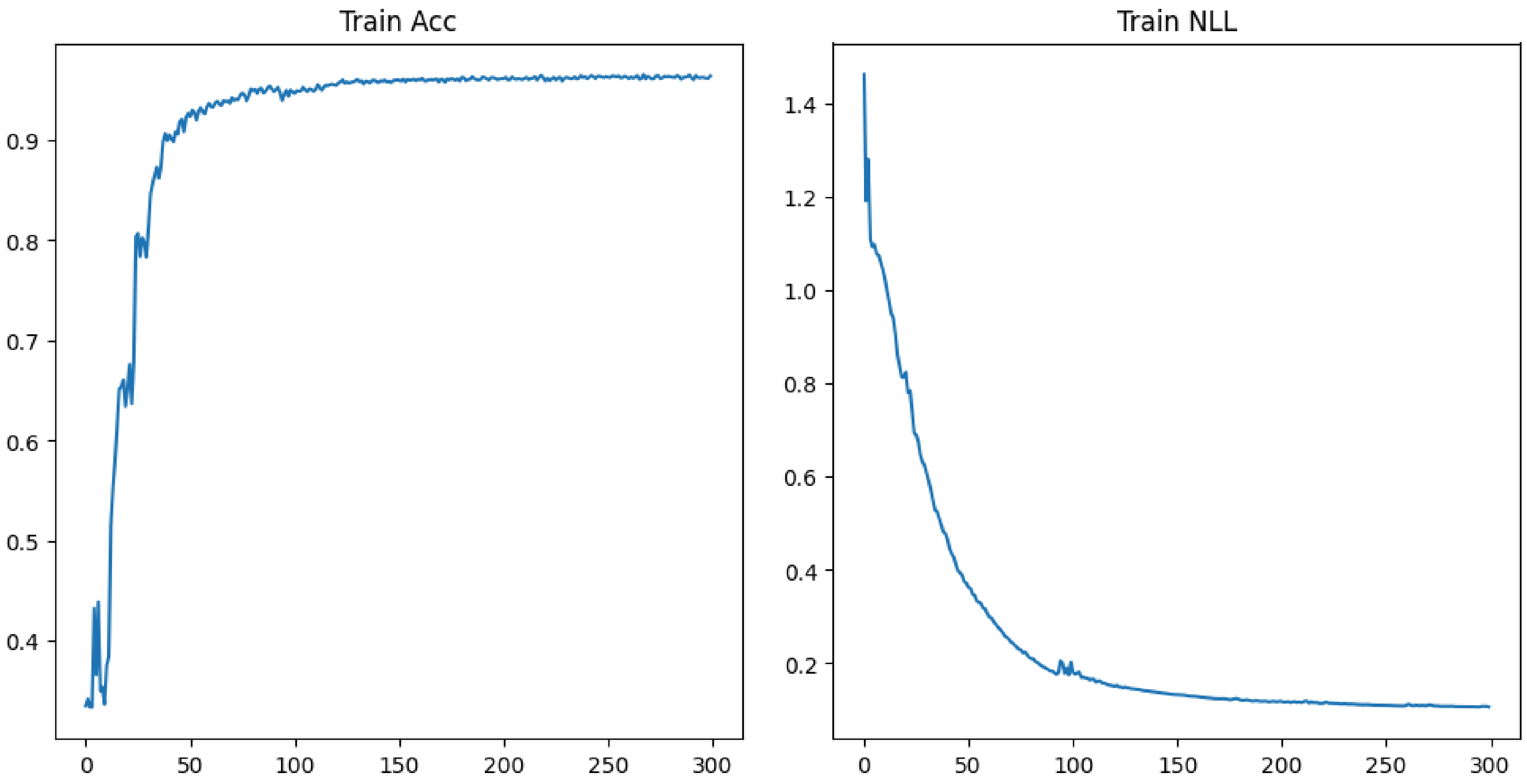
In [12]:

```
layer_sizes = [2, 150, 150, 3]
num_epochs = 300
step_size = 0.1
wd=1e-3

M = 4
K = layer_sizes[-1]
params_all = [None]*M
opt_state_all = [None]*M

log_acc, log_nll = [], []

opt_init, opt_update, get_params = optimizers.adam(step_size)

for i in range(M):
    # use random seed to randomly initialize the NN
    params_all[i] = init_network_params(layer_sizes, random.PRNGKey(i))
    opt_state_all[i] = opt_init(params_all[i])

key = random.PRNGKey(0)
for epoch in range(num_epochs):
    for i in range(M):
        key, seed = random.split(key)

        X_shuffled, y_shuffled, y_oh_shuffled = shuffle_dataset(X, y, y_onehot, seed)

        params_all[i] = get_params(opt_state_all[i])
        params_all[i], opt_state_all[i], train_loss = update(params_all[i], X_shuffled, y_oh_shuffled, opt_state_al

    logging(params_all, X, y, K, M, log_acc, log_nll)

    print('\r', f'[Epoch {epoch+1}]: Train Acc: {log_acc[-1]:.3f} | Train NLL: {log_nll[-1]:0.3f}', end='')
```

```
[Epoch 300]: Train Acc: 0.964 | Train NLL: 0.106
```

In [13]:

```
fig, axes = plt.subplots(1,2, figsize=(10,5))
fig.tight_layout()
axes[0].plot(log_acc)
axes[1].plot(log_nll)
axes[0].title.set_text('Train Acc')
axes[1].title.set_text('Train NLL')
```



Measuring Uncertainty - Total, Knowledge and Data Uncertainty

The Shannon entropy measures the information/uncertainty in a categorical distribution $\boldsymbol{p} = [p_1, p_2, \ldots, p_K]$

$$H(p) = -\sum_{i=1}^{K} p_i \log p_i$$

The entropy is high when the probability is spread out over the classes and low when most of the probability is in a single class.

We can use entropy to measure the uncertainty in a prediction. Using the same notation as Malinin et al. [1], we call the average entropy of each prediction of the ensemble the **data uncertainty**. Let $\boldsymbol{p}^{(1)}, \ldots, \boldsymbol{p}^{(M)}$ be the M predictive distributions of the ensemble, then the data uncertainty is

$$\frac{1}{M} \sum_{i=1}^{M} H\left(\boldsymbol{p}^{(i)}\right)$$

The entropy of the mean prediction is called the **total uncertainty**, which is high if the data uncertainty is high or if the predictions are diverse.

Knowledge uncertainty (think: uncertainty in parameters) is related to the diversity of the predictions, which we measure with the multi-distribution Jensen-Shannon divergence

$$H\left(\frac{1}{M} \sum_{i=1}^{M} \boldsymbol{p}^{(i)}\right) - \frac{1}{M} \sum_{i=1}^{M} H\left(\boldsymbol{p}^{(i)}\right)$$

Indeed, the divergence measures the difference between total and data uncertainty. It is low when the predictions are similar and high when the predictions are different but confident.

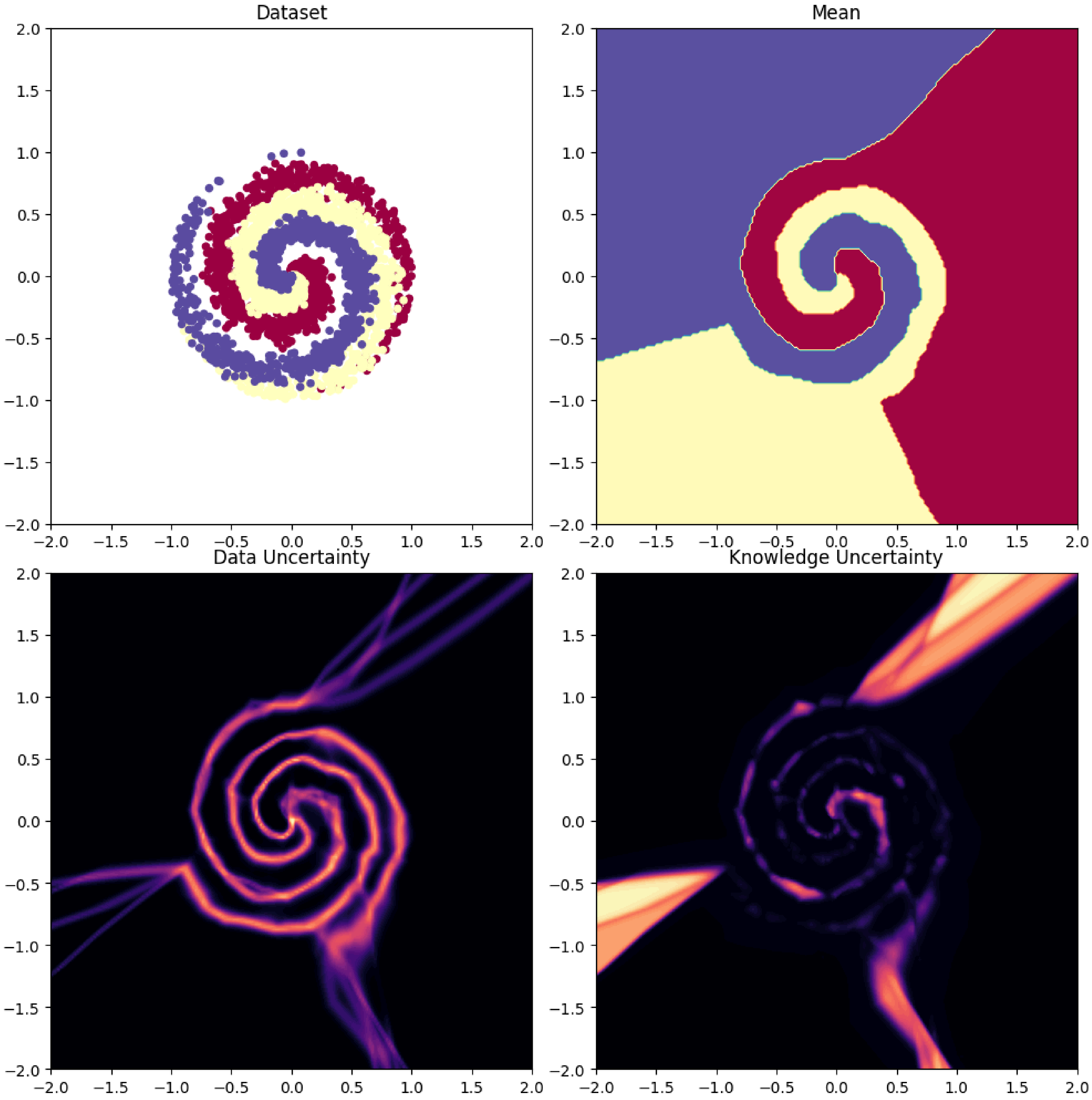
Below, we visualize the data and knowledge uncertainty for our deep ensemble.

In [14]:

```
def entropy(preds):
    preds = np.clip(preds, 1e-7, 1.0)
    return -np.sum( preds * np.log(preds), axis=1)

def js_terms(distributions):
    return entropy(np.mean(distributions, axis=0)), np.mean([entropy(p) for p in distributions], axis=0)

def visualize_predictions(X, y, params_list, min=-2.0, max=2.0, res=200, num_nets=1):
    xs = np.linspace(min, max, res)
    ys = np.linspace(min, max, res)
    N, M = len(xs), len(ys)
    xy = np.asarray([(_x,_y) for _x in xs for _y in ys])
    num_samples = xy.shape[0]

    predictions = [batched_predict(params, xy)[1] for params in params_list]
    predictions_ensemble = np.mean(predictions, axis=0)
    total, data = js_terms(predictions)

    Z, Z2, Z3 = np.zeros((N,M)), np.zeros((N,M)), np.zeros((N,M))
    indices = np.unravel_index(np.arange(num_samples), (N,M))
    Z[indices] = jnp.argmax(predictions_ensemble, axis=1)
    Z2[indices] = total - data
    Z3[indices] = data

    fig, axes = plt.subplots(2,2, figsize=(10,10))
    axes = axes.flatten()
    fig.tight_layout()

    axes[0].scatter(X[:, 0], X[:, 1], c=y, s=20, cmap=plt.cm.Spectral)
    axes[1].contourf(xs, ys, Z.T, cmap=plt.cm.Spectral, levels=50)
    axes[2].contourf(xs, ys, Z3.T, cmap='magma',levels=50)
    axes[3].contourf(xs, ys, Z2.T, cmap='magma', levels=50)

    axes[0].set_xlim([min, max]); axes[0].set_ylim([min, max]);

    axes[0].title.set_text('Dataset')
    axes[1].title.set_text('Mean')
    axes[2].title.set_text('Data Uncertainty')
    axes[3].title.set_text('Knowledge Uncertainty')
```

In [15]:

```
visualize_predictions(X, y, params_all, num_nets=M)
```



where dark to light increases in value.

Discussion points:

- Are there any regions where the network makes overconfident predictions?
- Where is the ensemble most uncertain (high entropy)? Why?
- Where are the members of the ensemble the most diverse? Why?
- How does adversarial training affect the results? Why?
- Finally, it is valuable for us to know how long did it take you to complete this practical?

1. Are there any regions where the network makes overconfident predictions?

Answer: Yes.

- Standard neural networks (without ensembles) tend to be overconfident (assigning close to 1.0 probability to a single class) in regions **far away from the training data** (the "background" regions in the plots).
- This happens because the decision boundaries typically extend linearly into infinity. If a point lies far away but on the "Class A" side of the boundary, the model is confident it is Class A, even though it has never seen data there.

2. Where is the ensemble most uncertain (high entropy)? Why?

Answer: The ensemble is most uncertain at the **decision boundaries** (e.g., the space directly between the spiral arms).

- [cite_start]**Metric:** This corresponds to high **Data Uncertainty** [cite: 594-596].
- **Why:** In these overlapping regions, the data naturally supports multiple classes (noise/ambiguity). Consequently, the predicted probabilities are split (e.g., 0.5/0.5), resulting in maximum Shannon entropy.

3. Where are the members of the ensemble the most diverse? Why?

Answer: The members are most diverse in **Out-of-Distribution (OOD)** regions (e.g., the corners of the plot or large gaps between spirals).

- [cite_start]**Metric:** This corresponds to high **Knowledge Uncertainty** [cite: 598-599].
- **Why:** Since there is no training data in these regions to constrain the parameters, the random initialization causes each ensemble member to draw its decision boundary differently. One model might classify a far-away point as Red, while another classifies it as Blue. This disagreement maximizes the Jensen-Shannon divergence.

4. How does adversarial training affect the results? Why?

Answer: Adversarial training smoothes the decision boundaries and expands the regions of uncertainty.

- **Effect:** It effectively "thickens" the decision boundary.
- **Why:** By training on perturbed examples, the model is forced to be uncertain not just *on* the mathematical line separating classes, but also in the ϵ-neighborhood around it. This prevents the model from making high-confidence predictions on points that are only marginally different from the training data.