ExerciseSheet_08

June 21, 2021

1 Probabilistic Machine Learning

University of Tübingen, Summer Term 2021

1.1 Exercise Sheet 8

This sheet is **due on Tuesday 22 June 2021 at 10am sharp.**

1.1.1 1. The Laplace Approximation for Neural Networks

[1]:
```
import numpy as np
import torch
import torch.nn.functional as F
from torch.nn.utils import parameters_to_vector
from torch.utils.data import TensorDataset, DataLoader
from torch.distributions import MultivariateNormal
from torch.distributions.multivariate_normal import _precision_to_scale_tril
from torchvision import datasets, transforms

import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_moons

from backpack import backpack, extend, memory_cleanup
from backpack.extensions import BatchGrad
from backpack.context import CTX

from einops import rearrange

import warnings
warnings.filterwarnings("ignore")
%config InlineBackend.figure_formats = 'svg', 'pdf'
```

[3]:
```
sns.set_style('white')
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

In this exercise, we will learn about a possibility to gain a notion of uncertainty over the outputs of a neural network. First, we implement a Laplace approximation (LA) over all parameters θ of a neural network f_θ. We assume i.i.d. data $\mathcal{D} = \{x_n, y_n\}_{n=1}^N$, a categorical likelihood $p(\mathcal{D} \mid \theta) = \prod_{n=1}^N p(y_n \mid f_\theta(x_n))$ (which corresponds to the standard cross-entropy classification loss), and an isotropic Gaussian prior $p(\theta) = \mathcal{N}(\theta\,;\, 0, \delta^{-1}\mathbf{I})$ (corresponding to regular weight decay). Then, the posterior distribution

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \theta)\, p(\theta)}{p(\mathcal{D})} \tag{1}$$

over the network weights can be approximated by

$$p(\theta \mid \mathcal{D}) \approx \mathcal{N}(\theta\,;\, \theta_{MAP}, \Sigma), \tag{2}$$

where

$$\theta_{MAP} := \arg\max_\theta p(\theta \mid \mathcal{D}) \quad \text{and} \quad \Sigma := \big(- \nabla^2_\theta \log p(\theta \mid \mathcal{D})|_{\theta_{MAP}} \big)^{-1}. \tag{3}$$

The derivation of this approximation is based on a second-order expansion of $\log p(\theta, \mathcal{D})$.

Since the Hessian of the log-likelihood is not guaranteed to be positive definite, in practice we employ the Generalized Gauss Newton (GGN) matrix as an approximation of the Hessian, which as defined as

$$GGN := \sum_{n=1}^N J(x_n)^T \big(\nabla^2_f \log p(y_n | f_\theta(x_n))|_{f_{\theta_{MAP}}(x_n)} \big) J(x_n), \tag{4}$$

where $J(x_n)_{cp} := \frac{\partial f_\theta(x_n)_c}{\partial \theta_p}|_{\theta_{MAP}}$ is the neural networks Jacobian matrix.

To implement the LA in practice, we usually need 1. a set of pre-trained weights, i.e. the MAP estimate θ_{MAP}, via regular training of the neural network, and 2. the GGN matrix (or some factorization of it, like the diagonal).

The cost overhead of LAs compared to simply using the MAP estimate mostly comes from (2). Moreover, (2) requires the most implementation effort. We will use the PyTorch library BackPack to efficiently calculate the Jacobian matrices required for the GGN approximation of the Hesssian, using the BatchGrad extension.

To make predictions using the LA, we could simply repeatedly sample sets of parameters from our approximate posterior to the true posterior $p(\theta|\mathcal{D})$, predict using these weights, and then average the softmax of these predictions. However, if we linearize the neural network at the MAP estimate, the predictive distribution over the neural network outputs can be obtained in closed-form, since the approximate posterior over the weights constructed with the LA has Gaussian form and Gaussians are closed under linear transformations. In fact, the GGN implicitly assumes that the neural network is linearized which makes this the natural approach of making predictions with the LA when using the GGN approximation to the Hessian (see this paper for more details if you are interested). The linearized predictive distribution for the sample x_* is

$$p(f(x_*) \mid x_*, \mathcal{D}) = \mathcal{N}(f(x_*)\,;\, f_{\theta_{MAP}}(x_*), J(x_*) \Sigma J(x_*)^T). \tag{5}$$

With this predictive distribution, we can simply use Monte Carlo integration to get a categorical distribution over the classes, by sampling from the predictive distribution above, applying the softmax function, and averaging over all MC samples.

Task 1.1: Fill in the missing pieces of code in the `Laplace` class. Look at the comments for hints.

[4]:
```
class Laplace:
    """Implements a Laplace approximation over all weights of a neural network,
    using the full Generalized Gauss Newton (GGN) matrix as an approximation to␣
↪the Hessian.
    Moreover, it allows sampling from the predictive distribution over the␣
↪network outputs.
    """

    def __init__(self, model, prior_precision=1.):
        self.model = model
        self._device = next(model.parameters()).device
        self.mean = parameters_to_vector(self.model.parameters()).detach()
        self.n_params = len(self.mean)

        # transform scalar prior precision to a vector (the diagonal of the␣
↪prior precision matrix)
        self.prior_precision = prior_precision * torch.ones_like(self.mean,␣
↪device=self._device)
        self.H = None
        self.loss = 0.

    def _init_H(self):
        """Initialize self.H as zero matrix"""
        self.H = torch.zeros(self.n_params, self.n_params, device=self._device)

    def infer(self, train_loader):
        """Fit the Laplace approximation at the MAP estimate, i.e. calculate␣
↪the GGN approximation
        to the Hessian over the whole training dataset. Accumulate the loss in␣
↪self.loss and the
        GGN in self.H.
        Parameters
        ----------
        train_loader : torch.data.utils.DataLoader
            each iterate is a training batch (X, y)
        """
        self.model.eval()
        loss_fn = torch.nn.CrossEntropyLoss(reduction='sum')

        # perform a forward pass for a single data point to get the output␣
↪dimension
        X, _ = next(iter(train_loader))
        with torch.no_grad():
            self.n_outputs = self.model(X[:1].to(self._device)).shape[-1]
        setattr(self.model, 'output_size', self.n_outputs)
```

```
        # initailize H
        self._init_H()

        ␣
↪#########################################################################################
        # Loop through the DataLoader and use ``_get_full_ggn`` to calculate␣
↪the batch Hessian and loss;
        # accumulate in self.loss and self.H
        ␣
↪#########################################################################################
        for X, y in train_loader:
            Js, f = self._jacobians(X)
            loss, H_gnn = self._get_full_ggn(Js, f, y)
            self.loss += loss
            self.H += H_gnn

    def _get_full_ggn(self, Js, f, y):
        """Compute full GGN from Jacobians Js, the network outputs f, and the␣
↪labels y.
        Parameters
        ----------
        Js : torch.Tensor
            Jacobians `(batch, parameters, outputs)`
        f : torch.Tensor
            functions `(batch, outputs)`
        y : torch.Tensor
            labels compatible with loss
        Returns
        -------
        loss : torch.Tensor
        H_ggn : torch.Tensor
            full GGN approximation `(parameters, parameters)`
        """
        loss_fn = torch.nn.CrossEntropyLoss(reduction='sum')
        loss = loss_fn(f, y)
        # second derivative of log lik is diag(p) - pp^T
        ps = torch.softmax(f, dim=-1)
        H_lik = torch.diag_embed(ps) - torch.einsum('mk,mc->mck', ps, ps)

        ␣
↪#########################################################################################
        # Calculate the GGN matrix H_ggn using the Jacobians (Js) and the loss␣
↪Hessian (H_lik)
        ␣
↪#########################################################################################
```

```
        JsT = rearrange(Js, 'n a b -> n b a')
        H_ggn = (JsT @ H_lik @ Js).sum(0)

        return loss.detach(), H_ggn

    def _check_H(self):
        """Check that self.H has been created by self.infer()"""
        if self.H is None:
            raise AttributeError('Laplace approximation not fitted. Run infer()␣
↪first.')

    def __call__(self, X, n_samples=100):
        """Predict on input data X by sampling network outputs (the predictive␣
↪distribution),
        turning them into probabilities (by applying the softmax), and␣
↪averaging over all samples.
        Parameter
        ----------
        X : torch.Tensor
            `(batch_size, input_shape)`
        n_samples : int
            number of samples for the Monte Carlo integration.
        Returns
        -------
        predictive: torch.Tensor
            a torch.Tensor is returned with a distribution over classes␣
↪(similar to a Softmax).
        """
        f_mu, f_var = self.glm_predictive_distribution(X)

        ␣
↪#############################################################################################
        # Sample from the predictive distribution (hint: look at the imports to␣
↪see how you can easily
        # sample from a Gaussian distribution), take the softmax, and then␣
↪return the mean over all samples
        ␣
↪#############################################################################################
        return F.softmax(MultivariateNormal(f_mu, f_var).sample_n(n_samples),␣
↪dim=2).mean(0)

    @property
    def posterior_precision(self):
        self._check_H()
        return self.H + torch.diag(self.prior_precision)
```

```
    @property
    def posterior_scale(self):
        return _precision_to_scale_tril(self.posterior_precision)

    @property
    def posterior_covariance(self):
        return self.posterior_scale @ self.posterior_scale.T

    def functional_variance(self, Js):
        ␣
↪#########################################################################################
        # Return the variance of the outputs, given by J(x) \Sigma J(x)^T
        ␣
↪#########################################################################################
        JsT = rearrange(Js, 'n a b -> n b a')
        return Js @ self.posterior_covariance @ JsT

    @torch.enable_grad()
    def glm_predictive_distribution(self, X):
        """Compute the linearized GLM posterior predictive parameters.
        Parameter
        ----------
        X : torch.Tensor
            `(batch_size, input_shape)`
        Returns
        -------
        f_mu : torch.Tensor
            the mean of the predictive distribution `(batch_size, num_classes)`
        f_var : torch.Tensor
            the covariance of the predictive distribution `(batch_size,␣
↪num_classes, num_classes)`
        """
        Js, f_mu = self._jacobians(X)
        f_var = self.functional_variance(Js)
        return f_mu.detach(), f_var.detach()

    def _jacobians(self, X):
        """Compute Jacobians of the network output with respect to the␣
↪parameters using BackPack's
        BatchGrad per output dimension. This corresponds to J(x) in the theory␣
↪section above.
        Parameters
        ----------
        X : torch.Tensor
            input data `(batch, input_shape)` on compatible device with model.
        Returns
        -------
```

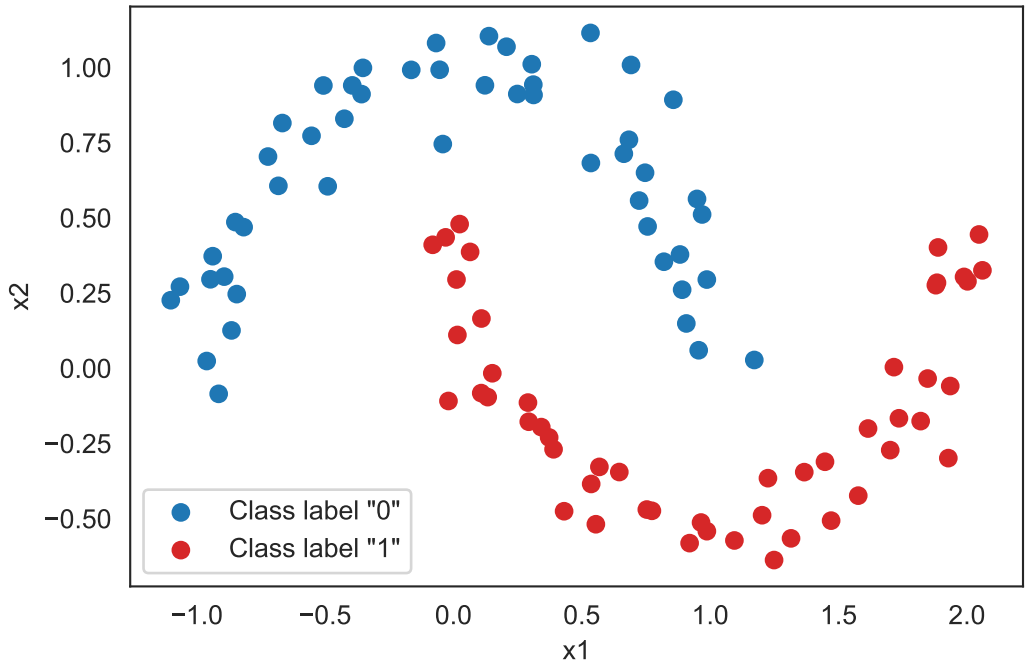
```
        Js : torch.Tensor
            Jacobians `(batch, parameters, outputs)`
        f : torch.Tensor
            output function `(batch, outputs)`
        """
        # Jacobians are (batch size x output dimension x number of network␣
↪parameters)
        model = extend(self.model)
        to_stack = []
        for i in range(model.output_size):
            model.zero_grad()
            out = model(X)

            # backward pass with Backpack
            with backpack(BatchGrad()):
                if model.output_size > 1:
                    out[:, i].sum().backward()
                else:
                    out.sum().backward()
                to_cat = []
                for param in model.parameters():
                    to_cat.append(param.grad_batch.detach().reshape(X.shape[0],␣
↪-1))
                    delattr(param, 'grad_batch')
                Jk = torch.cat(to_cat, dim=1)
            to_stack.append(Jk)
            if i == 0:
                f = out.detach()

        # cleanup
        model.zero_grad()
        CTX.remove_hooks()
        cleanup(model)
        if model.output_size > 1:
            return torch.stack(to_stack, dim=2).transpose(1, 2), f
        else:
            return Jk.unsqueeze(-1).transpose(1, 2), f


def cleanup(module):
    for child in module.children():
        cleanup(child)

    setattr(module, "_backpack_extend", False)
    memory_cleanup(module)
```

1.1.2 2. Toy Example

To get some intuition on how the LA behaves, we first consider a simple binary classification toy example.

[5]:
```python
def plot_data(X):
    labels = {
        'Class label "0"': 0,
        'Class label "1"': 1,
    }
    colors = ['tab:blue', 'tab:red']
    for key, color in zip(labels.keys(), colors):
        selection = (y == labels[key])
        plt.scatter(X[selection, 0], X[selection, 1], label=key, c=color)
    plt.legend(loc='lower left')
    plt.xlabel('x1')
    plt.ylabel('x2')
```

[6]:
```python
# create and plot moon dataset with two classes
X, y = make_moons(noise=0.1, random_state=0)
plot_data(X)
```



We first need to train a neural network, in this case a simple MLP, on the data to obtain the MAP estimate.

Task 2.1: Add the training loop. Look at the comments for hints.

[7]:
```
def train(model, X, y, epochs):
    model.train()
    loss_fn = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=5e-4)

    ␣
↪#############################################################################################
    # Write a simple full batch training loop and print the loss every 100␣
↪epochs
    ␣
↪#############################################################################################
    for i in range(epochs):
        optimizer.zero_grad()
        out = model(X)
        loss = loss_fn(out, y.flatten())
        loss.backward()
        optimizer.step()
        if (i + 1) % 100 == 0:
            print(f"Epoch {i + 1:4d}. Loss = {loss:.3f}")
```

[8]:
```
# set up a simple MLP
model = torch.nn.Sequential(
    torch.nn.Linear(X.shape[1], 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 2)
).to(device)

# train on data
X_train = torch.tensor(X, device=device, dtype=torch.float)
y_train = torch.tensor(y, device=device, dtype=torch.long).view(-1, 1)
train(model, X_train, y_train, 1000)
```

```
Epoch  100. Loss = 0.327
Epoch  200. Loss = 0.184
Epoch  300. Loss = 0.114
Epoch  400. Loss = 0.059
Epoch  500. Loss = 0.031
Epoch  600. Loss = 0.019
Epoch  700. Loss = 0.012
Epoch  800. Loss = 0.008
Epoch  900. Loss = 0.006
Epoch 1000. Loss = 0.005
```

To visualize the different behaviour of the LA compared to the MAP estimate, we want to plot the confidence of the predictions on a grid of test points, i.e., part of the plane where the data lives. The confidence is simply defined as the value of the highest softmax output. Hence, for

binary classification it is always in $[0.5, 1]$, where 0.5 indicates maximal uncertainty, i.e., uniform probability, and 1 maximal certainty. For test points around the training data, we want to have high certainty (similar to the accuracy on the training data) and for test points far away from the training data, we want to have high uncertainty.

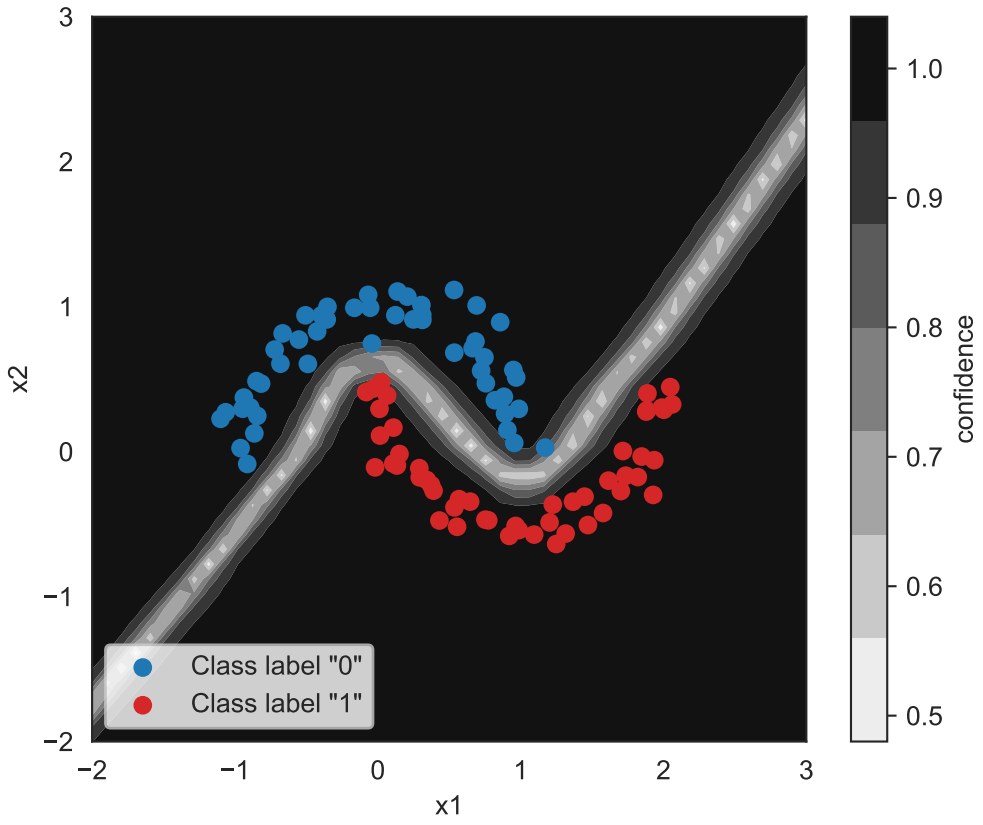
We start by looking at the confidence of the predictions with the MAP estimate.

Task 2.2: Predict on the test grid using the MAP estimate and calculate the confidence (name it `map_conf`).

[9]:
```
# create a grid of test points
test_rng = np.linspace(-2, 3, 50)
X1_test, X2_test = np.meshgrid(test_rng, test_rng)
X_test = torch.tensor(np.stack([X1_test.ravel(), X2_test.ravel()]).T,␣
↪device=device, dtype=torch.float)

##########################################################################################
# Test MAP estimate on grid
##########################################################################################
with torch.no_grad():
    map_conf = rearrange(F.softmax(model(X_test), 1).max(1).values, '(n m) -> n␣
↪m', n=50)
```

[10]:
```
def plot_confidence(conf, X, X1_test, X2_test):
    plt.figure(figsize=(6, 5))

    # plot confidence of the predictions
    plt.contourf(X1_test, X2_test, conf, cmap='binary')
    cbar = plt.colorbar(ticks=[0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
    cbar.set_label('confidence')

    plot_data(X)

    plt.show()
```

[11]:
```
# plot confidence of MAP predictions
plot_confidence(map_conf, X, X1_test, X2_test)
```



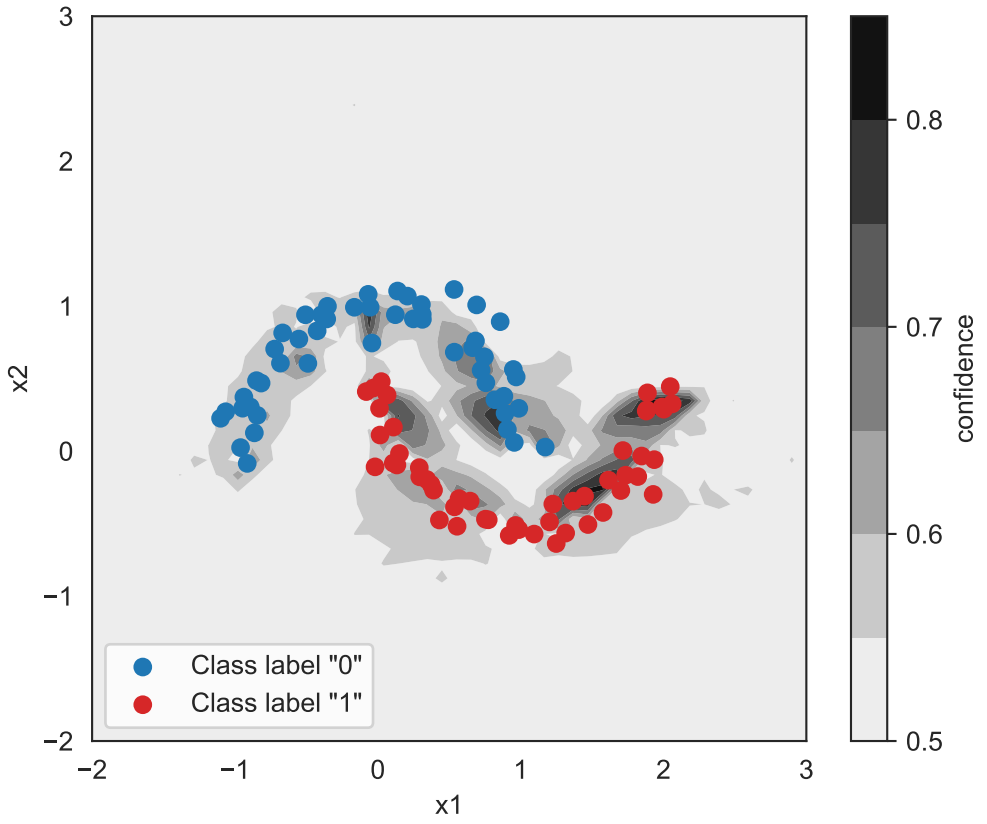
You should be able to clearly see, that the model is very confident everywhere but at the decision boundary. In fact, it is possible to show that neural networks with ReLU activation functions (like the one we are using here) are provably overconfident "far away" from the training data. You can have a look at the paper which first showed this if you are interested.

Task 2.3: Now, let's see how the confidence of predictions with the LA behaves. Initialize the LA with a prior precision of $5e{-}4$ (the same as the weight decay used for training). You can try different values and observe how the behaviour changes. Use 10000 MC samples to make predictions. Name the confidence of the predictions `lap_conf`. Look at the comments for hints.

[12]:
```
# create data loader from training data
train_loader = DataLoader(TensorDataset(X_train, y_train.view(-1)),␣
↪batch_size=64)

###############################################################################################
# Initialize and infer Laplace approximation at MAP estimate; use␣
↪prior_precision=5e-4
###############################################################################################
lap = Laplace(model, prior_precision=5e-4)
lap.infer(train_loader)
```

```
##########################################################################################
# Predict on the test grid (n_samples=10000) and get confidence of the␣
 ↪predictions
##########################################################################################
lap_conf = rearrange(lap(X_test, n_samples=10000).max(1).values, '(n m) -> n␣
 ↪m', n=50)
```

[13]:
```
# plot confidence of predictions with the Laplace approximation
plot_confidence(lap_conf, X, X1_test, X2_test)
```



Task 2.4: Describe the difference between the results with the MAP estimate and with the LA.

The MAP estimate has high certainty everywhere except on the decision boundary. The LA certainty is only high very hear the training data, but there are places that should have higher certainty that are very uncertain.

The difference to the MAP estimate should be clearly visible: predictions around the training data have relatively high confidence, whereas predictions further away from the training data have uniform confidence. This is much closer to the desired behaviour. This property of the Laplace approximation (and in fact all methods which use a Gaussian approximate posterior) can also be shown theoretically, at least asymptotically. If you are interested, take a look at the corresponding paper.

1.1.3 3. Last-Layer Laplace approximation

Since the GGN scales quadratically in the number of parameters, it is usually not feasible to use the full matrix over all parameters. Instead, one can either choose some factorization, like simply using the diagonal or a Kronecker-factored (K-FAC) approximation, or only treat the parameters of the last layer of the neural network probabilistically, which results in a so-called last-layer Laplace approximation (LLLA). We choose the second option for our next task.

In this case, we consider the network f as a linear function in the weight matrix $W \in \mathbb{R}^{C \times P}$ of the last layer, i.e. $f_W(x_*) = W\phi(x_*)$, with fixed features $\phi(x_*) =: \phi_* \in \mathbb{R}^P$, which are simply the output of the penultimate layer of the neural network given an input x_*. Note, that this formulation includes the case where we have a bias parameter, by simply employing the standard bias trick. In this setting, we therefore only need to do a Laplace approximation on a single weight matrix W: We obtain a Gaussian approximation $\mathcal{N}(\mathrm{vec}(W); \mu, \Sigma)$ with $\mu = \mathrm{vec}(W_{MAP}) \in \mathbb{R}^{CP}$ and $\Sigma = (H_W + \lambda \mathbf{I})^{-1} \in \mathbb{R}^{CP \times CP}$, where H_W is the Hessian of $-\log p(\mathcal{D} \mid W)$ w.r.t. $\mathrm{vec}(W)$ evaluated at $\mathrm{vec}(W_{MAP})$. For a single linear layer, the Hessian is equivalent to the GGN.

Let $x_* \in \mathbb{R}^D$ be an arbitrary test point. Since f_W is linear in W, we also have a Gaussian distribution $p(f_*|x_*, \mathcal{D}) = \mathcal{N}(f_*; m_*, C_*)$ over the marginal network outputs $f_* := f(x_*)$, where $m_* = W_{MAP}\phi_* \in \mathbb{R}^C$ and $C_* = (\phi_*^T \otimes \mathbf{I})\Sigma(\phi_* \otimes \mathbf{I}) \in \mathbb{R}^{C \times C}$. Again, we can simply use MC integration to make predictions.

Task 3.1: Fill in the missing pieces of code in the `LastLayerLaplace` class. Look at the comments for hints. Also, think about where the functions need to be different from the ones of the `Laplace` class; in most cases you can just copy your solution from above.

[20]:
```
class LastLayerLaplace:
    '''Implements a Laplace approximation over the weights of the last layer of a neural network,
    using the full Generalized Gauss Newton (GGN) matrix, which in this case is equivalent to the full Hessian.
    '''
    def __init__(self, model, prior_precision=1.):
        self.model = model
        if isinstance(self.model, torch.nn.Sequential):
            # Split model into Phi and the last layer
            self.feature_extractor = torch.nn.Sequential(*list(self.model.children())[:-1])
            self.last_layer = list(self.model.children())[-1]
        else:
            raise ValueError('only torch.nn.Sequential models are supported.')
        self._device = next(model.parameters()).device
        self.mean = parameters_to_vector(self.last_layer.parameters()).detach()
        self.n_params = len(self.mean)

        # transform scalar prior precision to a vector (the diagonal of the prior precision matrix)
        self.prior_precision = prior_precision * torch.ones_like(self.mean, device=self._device)
```

```
        self.H = None
        self.loss = 0.

    def _init_H(self):
        """Initialize self.H as zero matrix"""
        self.H = torch.zeros(self.n_params, self.n_params, device=self._device)

    def infer(self, train_loader):
        """Fit the Laplace approximation at the MAP estimate, i.e. calculate␣
↪the GGN/Hessian over
        the whole training dataset. Accumulate the loss in self.loss and the␣
↪GGN in self.H.
        Parameters
        ----------
        train_loader : torch.data.utils.DataLoader
            each iterate is a training batch (X, y)
        """
        self.model.eval()
        self._init_H()

        ␣
↪#########################################################################################
        # Loop through the DataLoader and use ``_get_full_ggn`` to calculate␣
↪the batch Hessian and loss;
        # accumulate in self.loss and self.H
        ␣
↪#########################################################################################
        for X, y in train_loader:
            Js, f = self._last_layer_jacobians(X)
            loss, H_gnn = self._get_full_ggn(Js, f, y)
            self.loss += loss
            self.H += H_gnn

    def _get_full_ggn(self, Js, f, y):
        """Compute full GGN from Jacobians Js, the network outputs f, and the␣
↪labels y.
        Parameters
        ----------
        Js : torch.Tensor
            Jacobians `(batch, parameters, outputs)`
        f : torch.Tensor
            functions `(batch, outputs)`
        y : torch.Tensor
            labels compatible with loss
        Returns
        -------
```

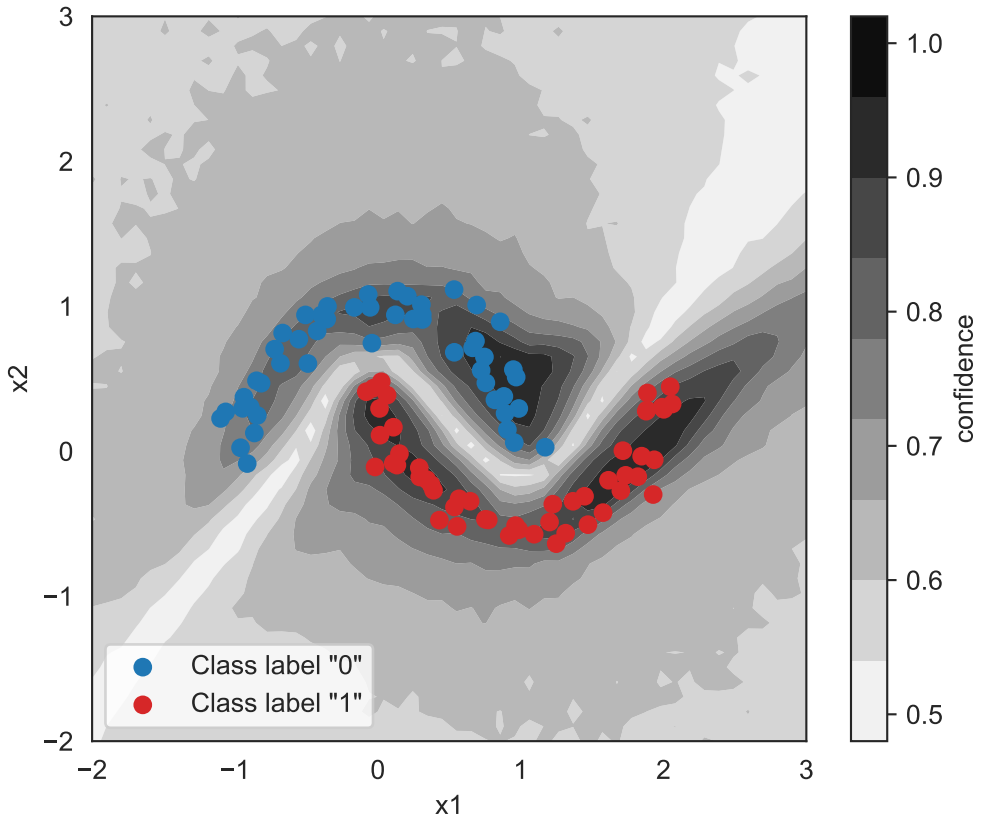
```
        loss : torch.Tensor
        H_ggn : torch.Tensor
            full GGN approximation `(parameters, parameters)`
        """
        loss_fn = torch.nn.CrossEntropyLoss(reduction='sum')
        loss = loss_fn(f, y)

        # second derivative of log lik is diag(p) - pp^T
        ps = torch.softmax(f, dim=-1)
        H_lik = torch.diag_embed(ps) - torch.einsum('mk,mc->mck', ps, ps)

        ␣
↪#############################################################################################
        # Calculate the GGN matrix using the Jacobians (Js) and the loss␣
↪Hessian (H_lik)
        ␣
↪#############################################################################################
        JsT = rearrange(Js, 'n a b -> n b a')
        H_ggn = (JsT @ H_lik @ Js).sum(0)

        return loss.detach(), H_ggn

    def _check_H(self):
        """Check that self.H has been created by self.infer()"""
        if self.H is None:
            raise AttributeError('Laplace approximation not fitted. Run infer()␣
↪first.')


    def __call__(self, X, n_samples=100):
        """Predict on input data X by sampling network outputs (the predictive␣
↪distribution),
        turning them into probabilities (by applying the softmax), and␣
↪averaging over all samples.
        Parameter
        ----------
        X : torch.Tensor
            `(batch_size, input_shape)`
        n_samples : int
            number of samples for the Monte Carlo integration.
        Returns
        -------
        predictive: torch.Tensor
            a torch.Tensor is returned with a distribution over classes␣
↪(similar to a Softmax).
        """
```

```
        f_mu, f_var = self.glm_predictive_distribution(X)

        ␣
↪#########################################################################################
        # Sample from the predictive distribution (hint: look at the imports to␣
↪see how you can easily
        # sample from a Gaussian distribution), take the softmax, and then␣
↪return the mean over all samples
        ␣
↪#########################################################################################
        return F.softmax(MultivariateNormal(f_mu, f_var).sample_n(n_samples),␣
↪dim=2).mean(0)

    @property
    def posterior_precision(self):
        self._check_H()
        return self.H + torch.diag(self.prior_precision)

    @property
    def posterior_scale(self):
        return _precision_to_scale_tril(self.posterior_precision)

    @property
    def posterior_covariance(self):
        return self.posterior_scale @ self.posterior_scale.T

    def functional_variance(self, Js):
        ␣
↪#########################################################################################
        # Return the variance of the outputs, given by J(x) \Sigma J(x)^T
        ␣
↪#########################################################################################
        JsT = rearrange(Js, 'n a b -> n b a')
        return Js @ self.posterior_covariance @ JsT

    def glm_predictive_distribution(self, X):
        """Compute the posterior predictive parameters.
        Parameter
        ----------
        X : torch.Tensor
            `(batch_size, input_shape)`
        Returns
        -------
        f_mu : torch.Tensor
            the mean of the predictive distribution `(batch_size, num_classes)`
        f_var : torch.Tensor
```

```
            the covariance of the predictive distribution `(batch_size,␣
↪num_classes, num_classes)`
        """
        Js, f_mu = self._last_layer_jacobians(X)
        f_var = self.functional_variance(Js)
        return f_mu.detach(), f_var.detach()

    def _last_layer_jacobians(self, X):
        """Compute Jacobians of the network output with respect to the␣
↪parameters of the last layer
        (no automatic differentiation needed). Js = I \otimes \phi.T
        Parameters
        ----------
        X : torch.Tensor
            input data `(batch, input_shape)` on compatible device with model.
        Returns
        -------
        Js : torch.Tensor
            Jacobians `(batch, parameters, outputs)`
        f : torch.Tensor
            output function `(batch, outputs)`
        """
        phi = self.feature_extractor(X)
        f = self.last_layer(phi)
        bsize = len(X)
        output_size = f.shape[-1]

        # calculate Jacobians using the feature vector 'phi'
        identity = torch.eye(output_size, device=X.device).unsqueeze(0).
↪tile(bsize, 1, 1)


        ␣
↪#########################################################################################
        # Use the feature vector phi and the identity to compute the Jacobians;␣
↪Jacobians are batch size
        # x output size x number of parameters
        # Js = I \otimes \phi.T
        # (hint: you can use torch.kron and a for-loop over each sample in the␣
↪batch, or torch.einsum)
        ␣
↪#########################################################################################
        Js = torch.empty(bsize, output_size, output_size*phi.shape[1])
        for i in range(bsize):
            Js[i] = torch.kron(identity[i], phi[i])

        # to account for the bias trick
```

```
        if self.last_layer.bias is not None:
            Js = torch.cat([Js, identity], dim=2)

        return Js, f.detach()
```

[21]:
```
lap = LastLayerLaplace(model, prior_precision=5e-4)
lap.infer(train_loader)
lap_conf = rearrange(lap(X_test, n_samples=10000).max(1).values, '(n m) -> n␣
↪m', n=50)
plot_confidence(lap_conf, X, X1_test, X2_test)
```



1.1.4 4. MNIST → FMNIST: out-of-distribution (OOD) detection

As a more "realistic" application of the LA, we consider a simple out-of-distribution (OOD) detection task. We first train on one dataset (here MNIST) and then also predict on a differently distributed dataset (here Fashion-MNIST (FMNIST)). Since the accuracy on the OOD dataset will be close to uniform, we want our model to reflect this by a matching confidence estimate.

We skip the training part and provide pre-trained weights. **Set the `model_path` variable appropriately.**

[22]:
```
# load simple CNN with pre-trained weights (on MNIST)
model = torch.nn.Sequential(
    torch.nn.Conv2d(1, 6, 5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(2),
    torch.nn.Conv2d(6, 16, 5),
    torch.nn.ReLU(),
    torch.nn.MaxPool2d(2),
    torch.nn.Flatten(),
    torch.nn.Linear(16 * 4 * 4, 120),
    torch.nn.ReLU(),
    torch.nn.Linear(120, 84),
    torch.nn.ReLU(),
    torch.nn.Linear(84, 10)  # <-- This layer is treated probabilistically
).to(device)
model_path = './lenet_mnist'  # adjust appropriately
model.load_state_dict(torch.load(model_path, map_location=device))
```

[22]:
```
<All keys matched successfully>
```

We need to create test data loaders for MNIST and FMNIST. In addition, we also need the training data loader to fit the LLLA.

Set the `data_path` and `download` variables appropriately.

[23]:
```
data_path = './data/'  # adjust appropriately
download = True  # set to True, if you haven't downloaded the data yet
tforms = transforms.ToTensor()

# create MNIST DataLoaders
mnist_train_loader = DataLoader(
    datasets.MNIST(data_path, train=True, transform=tforms, download=download),
    batch_size=128,
    shuffle=False,
)
mnist_test_loader = DataLoader(
    datasets.MNIST(data_path, train=False, transform=tforms, download=download),
    batch_size=512,
    shuffle=False,
)
# create F-MNIST DataLoader
fmnist_loader = DataLoader(
    datasets.FashionMNIST(data_path, train=False, transform=tforms,␣
 ↪download=download),
    batch_size=512,
    shuffle=False,
)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw


Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw


Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw


Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw


Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/26421880 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw


Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/29515 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw


Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
```

```
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/4422102 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/5148 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw
```

We want to track accuracy and confidence for predictions on both test sets.

[24]:
```python
@torch.no_grad()
def test(model, test_loader, device):
    out_dist = list()
    conf = 0.
    correct = 0
    for X, y in test_loader:
        if isinstance(model, LastLayerLaplace):
            out = model(X.to(device), n_samples=10000)
        else:
            out = torch.softmax(model(X.to(device)), 1)
        out_dist.append(out)

        # For each data point, preds contains the most likely class and
        # c the corresponding probability
        c, preds = torch.max(out, 1)
        conf += c.sum()
        correct += (y == preds).sum()
    out_dist = torch.cat(out_dist).cpu().numpy()

    # proportion of correctly classified data points
    accuracy = correct.item() / len(test_loader.dataset)

    # average confidence over all classifications
    conf = conf.item() / len(test_loader.dataset)
    return out_dist, accuracy, conf
```

Task 4.1: Use the MAP estimate (just the model with the pre-trained weights) to predict on both

test sets and print the accuracy and confidence.

[27]:
```
_, m_acc, m_conf = test(model, mnist_test_loader, 'cpu')
_, f_acc, f_conf = test(model, fmnist_loader, 'cpu')
print(f"Accuracy of model on MNIST: {m_acc:.4f}, Confidence: {m_conf:.4f}")
print(f"Accuracy of model on Fashion-MNIST: {f_acc:.4f}, Confidence: {f_conf:.
↪4f}")
```

```
Accuracy of model on MNIST: 0.9924, Confidence: 0.9936
Accuracy of model on Fashion-MNIST: 0.1224, Confidence: 0.6256
```

Task 4.2: Now, initialize and infer the LLLA. Use a prior precision of 0.1. As before, you can try different values and see how the results change.

[28]:
```
lap = LastLayerLaplace(model, 0.1)
lap.infer(mnist_train_loader)
```

Task 4.3: Predict on both test sets using LLLA and print the accuracy and confidence.

[29]:
```
_, m_acc, m_conf = test(lap, mnist_test_loader, 'cpu')
_, f_acc, f_conf = test(lap, fmnist_loader, 'cpu')
print(f"Accuracy of LLLA on MNIST: {m_acc:.4f}, Confidence: {m_conf:.4f}")
print(f"Accuracy of LLLA on Fashion-MNIST: {f_acc:.4f}, Confidence: {f_conf:.
↪4f}")
```

```
Accuracy of LLLA on MNIST: 0.9924, Confidence: 0.9862
Accuracy of LLLA on Fashion-MNIST: 0.1214, Confidence: 0.5267
```

Task 4.4: Describe how the results with the MAP estimate and the LLLA differ. Two or three sentences are sufficient.

The LLLA model is less certain on the out of distribution dataset than the MAP estimate. However, it is still much too confident. The confidence on the MNIST data has decreased only very slightly, which is good.