Exercise 3: Normalizing Flow

Erik Stolt

What you did and how

We were tasked to implement three different types normalizing flow (NF); diagonal Gaussian, full 3D-Gaussian and full flow. Diagonal meaning that there is no off-diagonal elements in the covariance matrix when we assume Gaussian distributions of the variables. In full Gaussian we have off-diagonal elements and hence correlation. In full flow we have non-linear transformations g plus some affine (linear) transformation t in the end.

These NF:s were implemented using `Jammy Flow` which provides the skeleton for the NF. Then I implemented the CNN architecture to train the parameters of the NF to predict the parameters of the probability density function (pdf) of the labels $T_{\text{eff}}, \log g$ and $[Fe/h]$. In the end, we have a pdf of the labels (for a given input) which tells us "how certain" we are about the true value of the labels. Additionally, I presented the results in 5 main plots: Loss vs epoch, pdf visualization at different epochs, final distribution for the uncertainty, a joint pdf for two labels and histogram plots for each label also showing the true value for the input.

In order to obtain the target conditional distribution did I first create a grid and evaluated the grid points for the taget pdf by calling `model.pdf(...)`. This then evaluated the target distribution on the grid, giving me the predicted pdf model.

To quantify my results, I plotted the coverage for the fraction of true values inside the model's 68th and 95th predicted intervals. For a properly trained model, the predicted percentiles should agree with the 68th and 95th percentile. This works even for non-Gaussian shapes, since the pdf should agree with the observations if trained properly. Moreover, this does not assume any specific shape for the pdf or the data.

What results you obtained

Overall, I obtained good results. Down below, I present some plots from the exercise. I trained all models for 100 with early stopping (patience = 10) and reduced learning rate on plateau (patience = 4). Additionally, I used batch size = 64. An example output during training looked like this:

```
Epoch 88/100 - Train Loss: 0.745930
          - Val Loss:   0.938252 and lr: 3.91e-07
          - No improvement for 3 epoch(s).
```

And some plots...

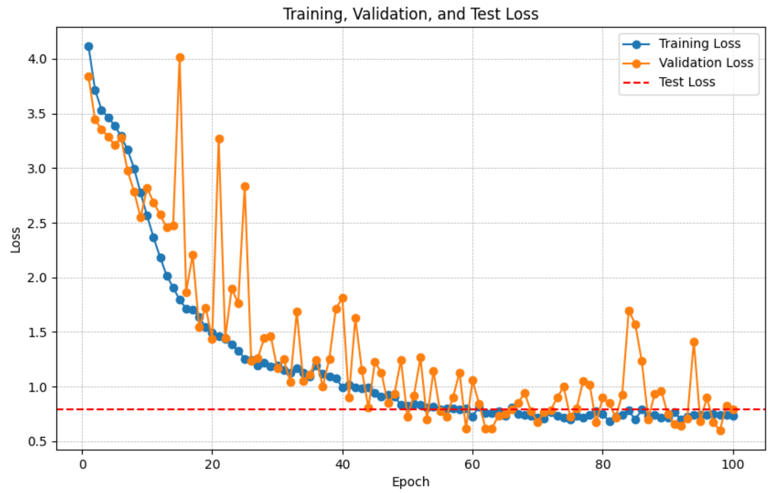

Figure 1: Loss for the training data, validation data and the test data over 100 epochs using diagonal Gaussian.

Figure 2: Loss for the training data, validation data and the test data over 100 epochs using full Gaussian.

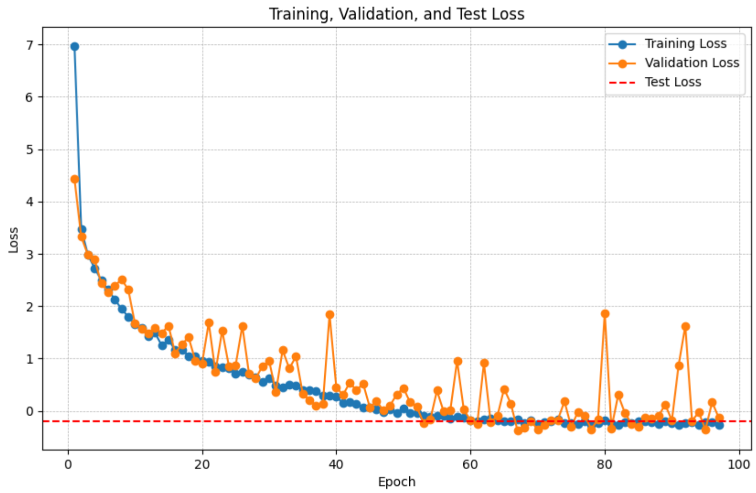

Figure 3: Loss for the training data, validation data and the test data over 100 epochs using full flow.

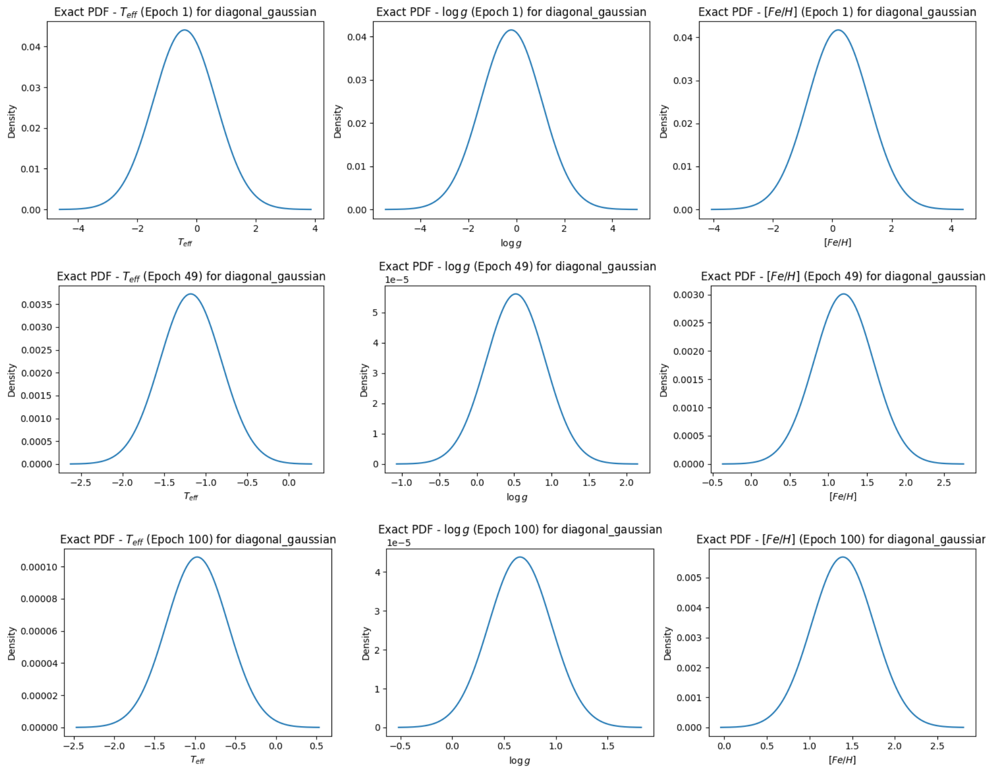

Figure 4: Evolution of the pdf for the three labels using diagonal Gaussian.

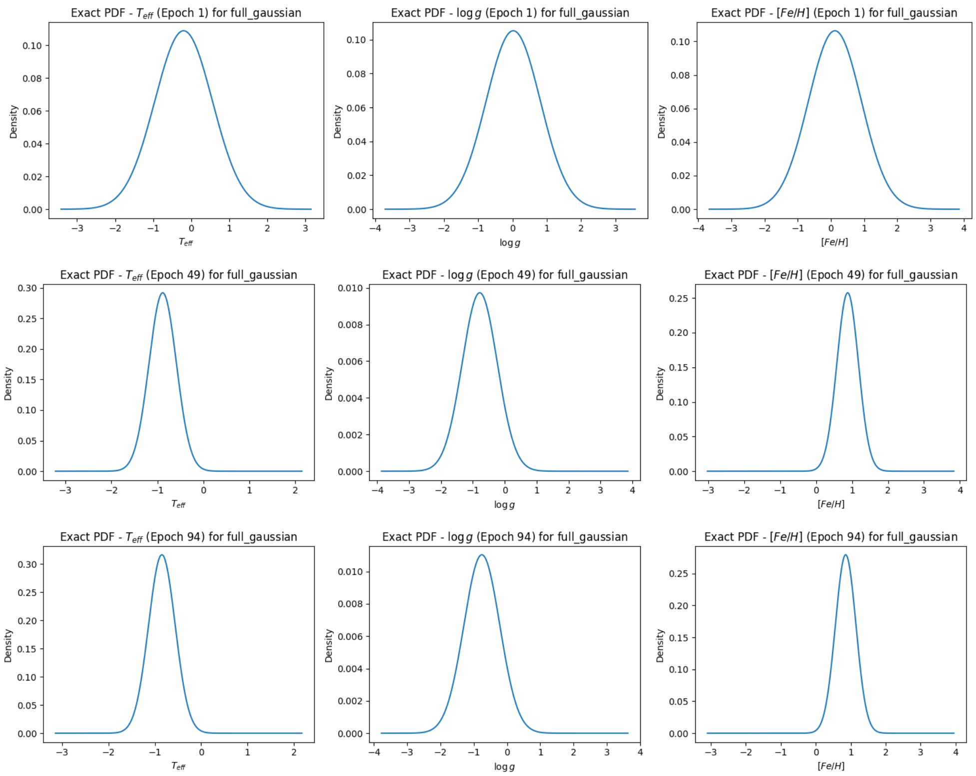

Figure 5: Evolution of the pdf for the three labels using full Gaussian.

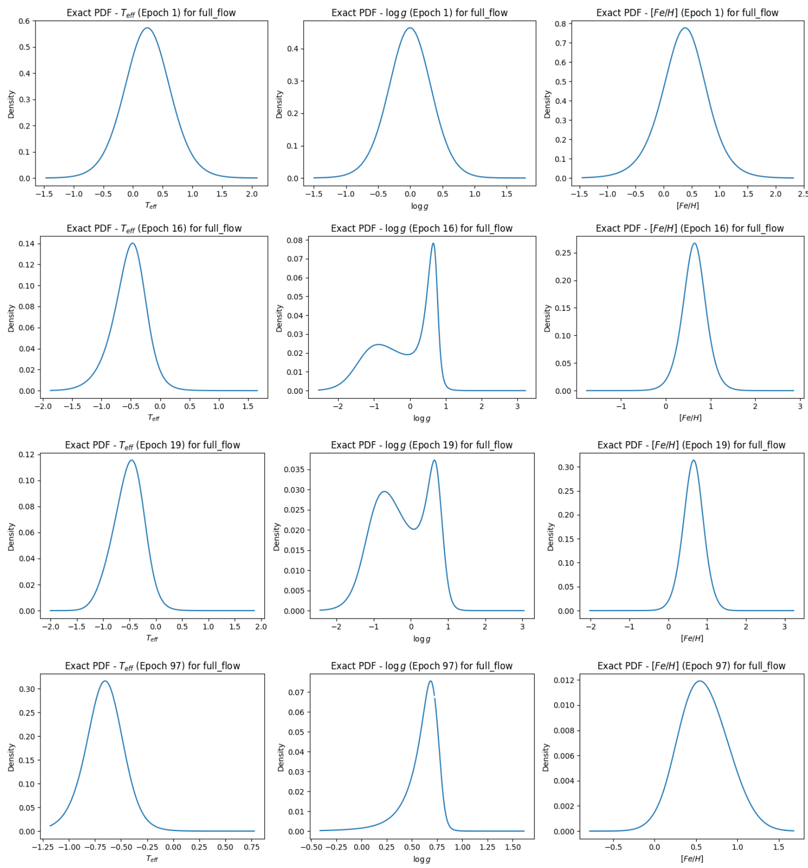

Figure 6: Evolution of the pdf for the three labels using full flow. We clearly see the non-linearity of the NF, especially in epochs 16 and 19. In the end the pdf's converge to somewhat of a Gaussian (though not quite).

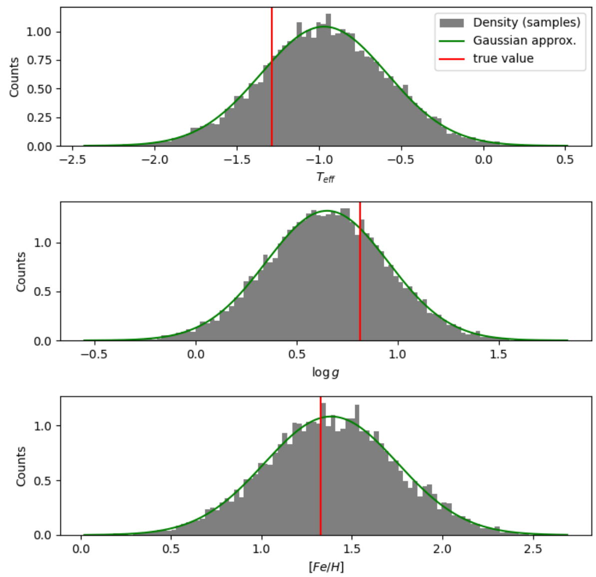

Figure 7: Plot over the sampled target distribution for the three labels with a fitted Gaussian and the true value for the label. Here using diagonal Gaussian.

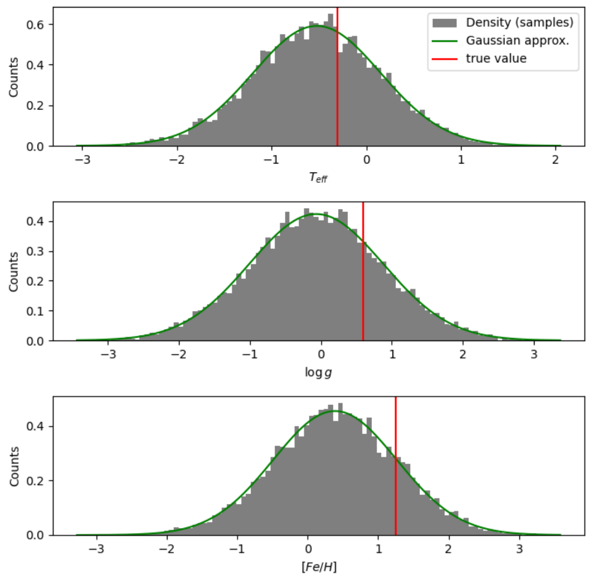

Figure 8: Plot over the sampled target distribution for the three labels with a fitted Gaussian and the true value for the label. Here using full Gaussian.

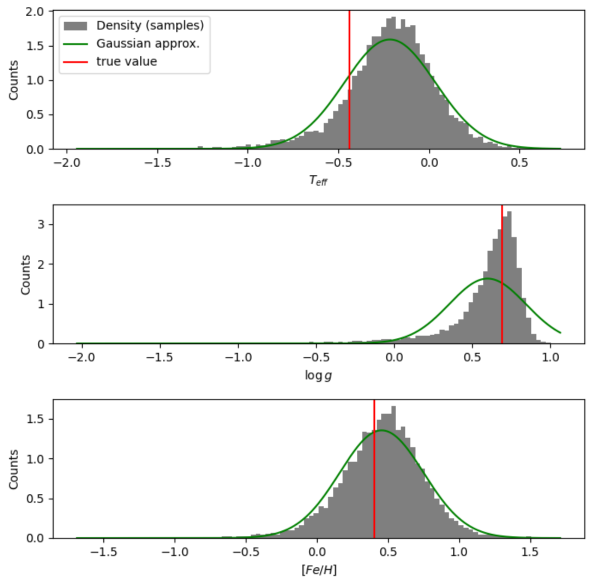

Figure 9: Plot over the sampled target distribution for the three labels with a fitted Gaussian and the true value for the label. Here using full flow.

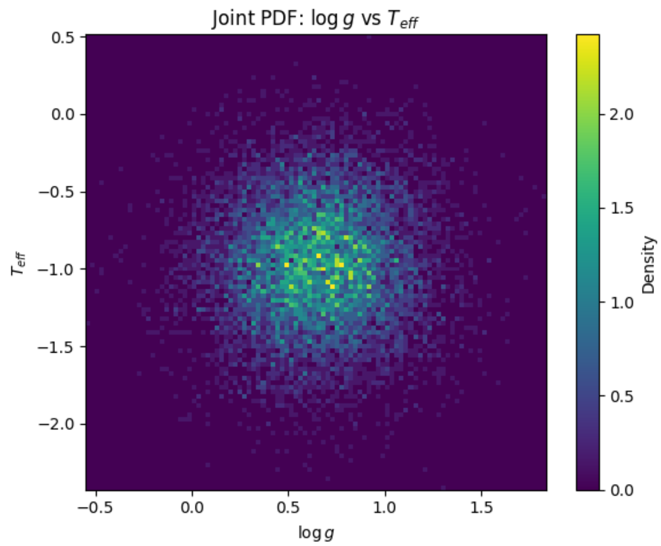

Figure 10: Heat map showing the joint pdf of $\log g$ *and* T_{eff} *for diagonal Gaussian. Notice that there is no covariance.*

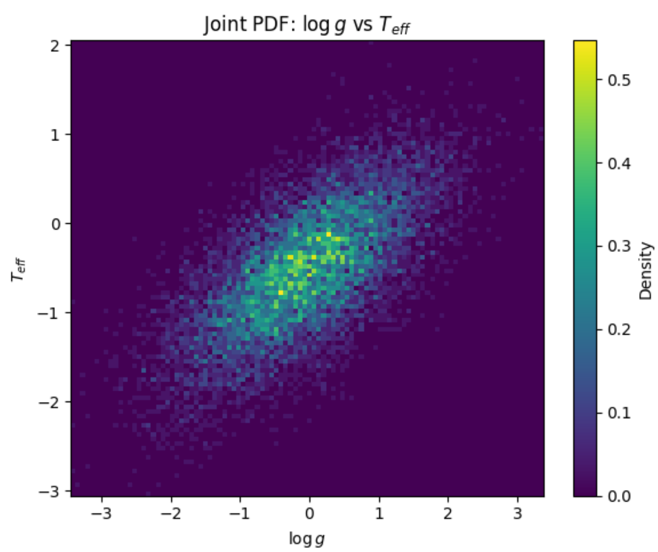

Figure 11: Heat map showing the joint pdf of $\log g$ *and* T_{eff} *for full Gaussian. Notice that there is a covariacne.*

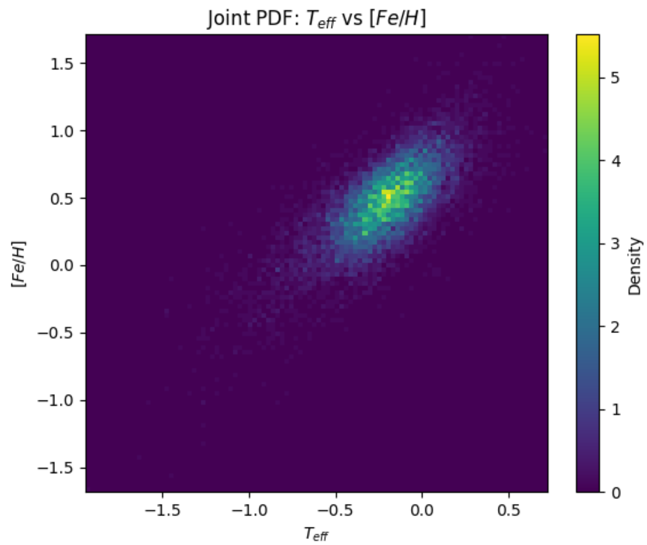

Figure 12: Heat map showing the joint pdf of $[Fe/H]$ and T_{eff} for full flow.

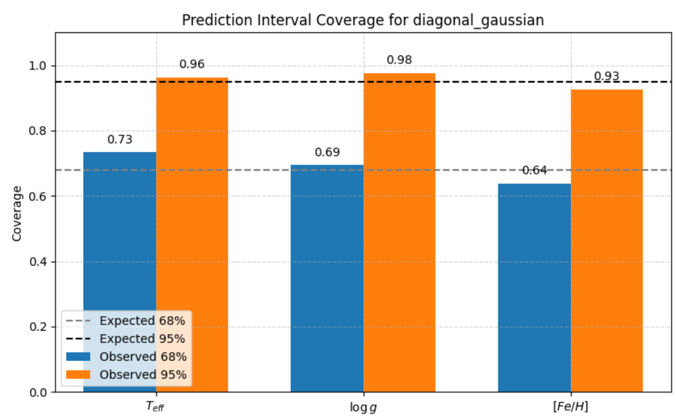

Figure 13: Convergence plot for each label for the 68th and 95th percentile for diagonal Gaussian.

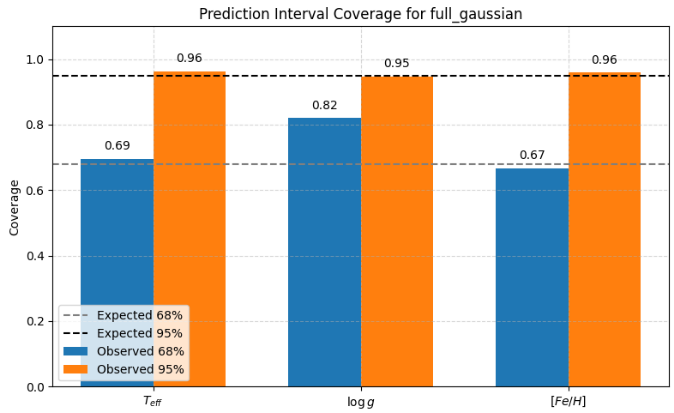

Figure 14: Convergence plot for each label for the 68th and 95th percentile for full Gaussian.

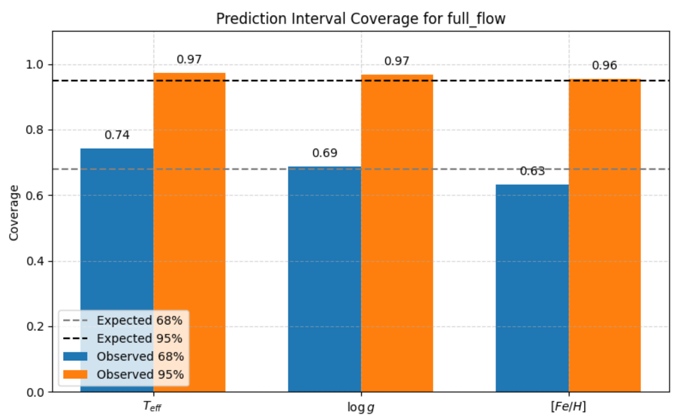

Figure 15: Convergence plot for each label for the 68th and 95th percentile for full flow.

Remark: The plots were plotted using the first element in the first batch in the test set with random seed = 42 (see more in discussion), and therefore not fully representative for the entire dataset. This however was not the exercise, only to plot some pdf's and to predict the pdf for a given input spectra.

Discussion

The first thing i noticed when training the network was that dropout really did not work. This indicates to me that the model is very sensitive to perturbing/dropping nodes which results in poor parameter predictions. Therefore, I trained the model without dropout.

We observe that the model's predicted uncertainties are not always centered on the true values. This is, however, not surprising when looking at only a single batch. The model predicts a full probability distribution conditioned on the input, and the mean of this distribution may differ from the true label for individual examples. Importantly, we do not expect each individual prediction to be unbiased. However, if the model is well-calibrated and unbiased overall, the average residual (i.e., prediction minus true value) across the full test set should be approximately zero. We strengthen this by looking in the convergence plot. There see see that percentiles, sampled from the conditional distribution, matches well with the observed percentiles. When a data point is far from the mean, the model's uncertainty (assuming it's well-trained) tends to predict a larger correction for that point. This means that the pdf may not be centered around the true mean of the distribution, as the model is adjusting the data point towards the mean. Essentially, the larger corrections for outlier points cause the pdf to shift, reflecting the model's attempts to correct those points.

The reason for full flow converging to a almost Gaussian shape means that the underlying data already is distributed similar to a Gaussian. We see that the model is indeed capable of predicting more complex patterns but since it converges and stays roughly the same for 30 epochs means that its confident in its shape. Which is further strengthen by the convergence plot.

The coverage plot for full Gaussian shows that the model is well-calibrated at the 95% level across all parameters but slightly underestimates uncertainty at the 68% level, particularly for T_{eff} and [Fe/H]. This behavior is expected for a *full Gaussian* model, which may struggle to capture asymmetries or heavier tails in the true posterior distribution. A more expressive model, such as a full flow, could better capture the distributional shape and improve coverage at narrower confidence levels, which we indeed see!

I also chose to keep the fitted gaussian in the visualization plots even for full flow since it was given in the template and one can see it does not fit very well compared to the full flow NF. Lastly, we see that full flow captures more complex patterns and that the diagonal Gaussian only has a simple 3D Gaussian with no correlation between the labels. Full Gaussian is also a 3D Gaussian but with correlation.

What challenges you encountered and what could be improved

The main challenge was that the data uncertainties matched very well with a simple Gaussian, hence, the full flow did not deviate a lot from the Gaussian. This meant that I had to train the network a lot in order to see a more complex pdf. However, this was resolved in the later stage of the exercise where I saw that I had incorrectly split the data. When this was resolved much better results and nicer pdf's were obtained!

Finding a good network architecture is always crucial in ML problems. I have tried a few networks, but there is probably a better one. If I had more time I would try some more networks. Moreover, It was also somewhat difficult to fully understand how `Jammy Flows` really works. This led me to dedicate a lot of time just to understand how the template code worked. In the end, I would say I have a good grasp of how it works!

Link to Github

Link: `https://github.com/erst6955/Advanced-Applied-Deep-Learning-in-Physics-And-Engineering`

Code (for backup)

Remark: The only thing I changed between the final runs was the pdf model. Everything else was unchanged.

~\Desktop\Advanced Applied Deep Learning In Physics And Engineering\Exercises\Exercise 3 - Normalizing Flows\stellar_prediction.py

```python
import time
import sys
import os
import argparse
import glob
import subprocess
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset, random_split
from matplotlib import pyplot as plt
import jammy_flows
from scipy.stats import norm
from sklearn.preprocessing import MinMaxScaler
from torch.utils.data import Dataset, DataLoader, random_split
import pylab
from sklearn.preprocessing import StandardScaler
import random


# Set random seed for reproducibility
seed = 42
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(seed)


DATA_PATH = r"C:\Users\Erik\Desktop\Advanced Applied Deep Learning In Physics And Engineering\Datasets\Astronomy"
fp64_on_cpu = False

# Hyperparameters
#learning_rate = 1e-4

names = ["$T_{eff}$", "$\log g$", "$[Fe/H]$"]



# ============================ Jammy Flow functions ============================

# Defining the normalizng flow model is a bit more involved and requires knowledge of the jammy_flows library.
# Therefore, we provide the relevant code here.
```

```python
class CombinedModel(nn.Module):
    """
    A combined model that integrates a normalizing flow with a CNN encoder.
    """

    def __init__(self, encoder, nf_type="diagonal_gaussian"):
        """
        Initializes the normalizing flow model.

        Parameters
        ----------
        encoder : callable
            A function or callable object that returns an encoder model. The encoder model
            should take the number of flow parameters as input and output the latent dimension.
        nf_type : str, optional
            The type of normalizing flow to use. Options are "diagonal_gaussian", "full_gaussian",
            and "full_flow". Default is "diagonal_gaussian".
        Raises
        ------
        Exception
            If an unknown `nf_type` is provided.
        Notes
        -----
        This method sets up a 3-dimensional probability density function (PDF) over Euclidean space (e3)
        using the specified normalizing flow type. The flow structure and options are configured based on
        the provided `nf_type`. The PDF is created using the `jammy_flows` library, and the number of flow
        parameters is determined and printed. The encoder is initialized with the number of flow parameters.
        """

        super().__init__()

        # we define a 3-d PDF over Euclidean spae (e3)
        # using recommended settings (https://github.com/thoglu/jammy_flows/issues/5 scroll down)
        opt_dict = {}
        opt_dict["t"] = {}
        if (nf_type == "diagonal_gaussian"):
            opt_dict["t"]["cov_type"] = "diagonal"
            flow_defs = "t"
        elif (nf_type == "full_gaussian"):
            opt_dict["t"]["cov_type"] = "full"
            flow_defs = "t"
        elif (nf_type == "full_flow"):
            opt_dict["t"]["cov_type"] = "full"
            flow_defs = "gggt"
```

```
        else:
            raise Exception("Unknown nf type ", nf_type)

        opt_dict["g"] = dict()
        opt_dict["g"]["fit_normalization"] = 1
        opt_dict["g"]["upper_bound_for_widths"] = 1.0
        opt_dict["g"]["lower_bound_for_widths"] = 0.01

        self.nf_type = nf_type

        # 3d PDF (e3) with ggggt flow structure. Four Gaussianation-flow (https://arxiv.org/abs/2003.01941) layers ("g") and an affine flow ("t")
        self.pdf = jammy_flows.pdf("e3", flow_defs, options_overwrite=opt_dict,
                                   amortize_everything=True, amortization_mlp_use_custom_mode=True)

        # get the number of flow parameters based on the selected model
        num_flow_parameters = self.pdf.total_number_amortizable_params

        print("The normalizing flow has ", num_flow_parameters, " parameters...")

        # latent dimension (output of the CNN encoder) is set to 128
        self.encoder = encoder(num_flow_parameters)

    def log_pdf_evaluation(self, target_labels, input_data):
        """
        Evaluate the log probability density function (PDF) for the given target labels and input data.

        The normalizing flow parameters are predicted by the encoder network based on the input data.
        Then, the log PDF is evaluated at the position of the label.

        Parameters:
        -----------
        target_labels : torch.Tensor
            The target labels for which the log PDF is to be evaluated.
        input_data : torch.Tensor
            The input data to be encoded and used for evaluating the log PDF.
        Returns:
        --------
        log_pdf : torch.Tensor
            The evaluated log PDF for the given target labels and input data.
        """
        latent_intermediate = self.encoder(input_data)  # get trained flow parameters from the CNN encoder

        if (self.nf_type == "full_flow"):
            # convert to double. Double precision is needed for the Gaussianization flow. This is for numerical stability.
```

```python
            if fp64_on_cpu:  # MPS does not support double precision, therefore we need to run the flow on the CPU
                latent_intermediate = latent_intermediate.cpu().to(torch.float64)
                target_labels = target_labels.cpu().to(torch.float64)
            else:
                latent_intermediate = latent_intermediate.to(torch.float64)
                target_labels = target_labels.to(torch.float64)

        # evaluate the log PDF at the target labels. We use log pdf for numerical stability.
        log_pdf, _, _ = self.pdf(target_labels, amortization_parameters=latent_intermediate)
        return log_pdf

    def sample(self, flow_params, samplesize_per_batchitem=1000):
        """
        Sample new points from the PDF given input data.

        Parameters
        ----------
        flow_params : tensor
            Parameters for the normalizing flow, must be of shape (B, L) where B is the batch size and L is the latent dimension.
        samplesize_per_batchitem : int, optional
            Number of samples to draw per batch item. Defaults to 1000.

        Returns
        -------
        tensor
            A tensor of shape (B, S, D) where B is the batch dimension, S is the number of samples,
            and D is the dimension of the target space for the samples.
        """
        # for full flow we need to convert to double precision for the normalizing flow
        # for numerical stability
        if (self.nf_type == "full_flow"):
            # convert to double
            if fp64_on_cpu: # MPS does not support double precision, therefore we need to run the flow on the CPU
                flow_params = flow_params.cpu().to(torch.float64)
            else:
                flow_params = flow_params.to(torch.float64)

        batch_size = flow_params.shape[0] # get the batch size
        # sample from the normalizing flow
        repeated_samples, _, _, _ = self.pdf.sample(amortization_parameters=flow_params.repeat_interleave(
            samplesize_per_batchitem, dim=0), allow_gradients=False)

        # reshape the samples to be grouped by batch item
        reshaped_samples = repeated_samples[:, None, :].view(
            batch_size, samplesize_per_batchitem, -1)

```

```
        return reshaped_samples

    def forward(self, input_data, samplesize_per_batchitem=1000):
        """
        Perform a forward pass through the model, predicting the mean and standard deviation of the samples.

        Normalizing flows do not directly predict the target labels. Instead, they predict the parameters of the flow that
        transforms the base distribution to the target distribution. Often, we still want to predict the target labels.
        Then, we can sample from the distribution and form the mean of the samples and their standard deviations.
        This is what this function does.

        Parameters
        ----------
        input_data : torch.Tensor
            The input data tensor.
        Returns
        -------
        torch.Tensor
            A tensor of size (B, D*2) where the first half (size D) are the means,
            the second half (another D) are the standard deviations.
        """
        flow_params=self.encoder(input_data)
        samples=self.sample(flow_params, samplesize_per_batchitem=samplesize_per_batchitem)

        # form mean along dim 1 (samples)
        means=samples.mean(dim=1)
        # form std along dim 1 (samples)
        std_deviations=samples.std(dim=1)

        # return means and std deviations as a concatenated tensor along dim 1
        return torch.cat([means, std_deviations], dim=1)

    def visualize_pdf(self, input_data, filename, pdf_model, samplesize=10000, batch_index=0, truth=None):
        """
        Visualizes the probability density function (PDF) of the given input data using a normalizing flow model.

        The function generates samples from the normalizing flow (using the sample() function)
        and plots the histogram of the samples together with a Gaussian approximation.

        Parameters
        ----------
        input_data : torch.Tensor
            The input data tensor from which to pick one batch item for visualization.
        filename : str
```

```
            The filename where the resulting plot will be saved.
        samplesize : int, optional
            The number of samples to generate for the PDF visualization (default is 10000).
        batch_index : int, optional
            The index of the batch item to visualize (default is 0).
        truth : torch.Tensor, optional
            The true values of the labels, used for comparison in the plot (default is None).

        Returns
        -------
        None
        """
        # pick out one input from batch
        input_bitem = input_data[batch_index:batch_index+1]

        # get the flow parameters (by passing the input data through the CNN encoder
network). This is basicallly evaluation of the encoder network.
        flow_params = self.encoder(input_bitem)

        # sample from the normalizing flow (i.e. samples are drawn from the base distribution
and transformed by the flow
        # using the change-of-variable formula)
        samples = self.sample(flow_params, samplesize_per_batchitem=samplesize)
        # the rest of the code is just plotting.

        # we only have 1 batch item
        samples = samples.squeeze(0)

        # plot three 1-dimensional distributions together with normal approximation,
        # so we calculate the mean and std of the samples
        mean = samples.mean(dim=0).cpu().numpy()
        std = samples.std(dim=0).cpu().numpy()
        samples = samples.cpu().numpy()

        fig, axdict = plt.subplots(3, 1, figsize=(8, 8))

        fig.subplots_adjust(hspace=0.4)  # Adjust the space between subplots
        for dim_ind in range(3):
            ax = axdict[dim_ind]
            ax.hist(samples[:, dim_ind], color="k", density=True, bins=100, alpha=0.5,
label="Density (samples)")

            # Gaussian overlay
            xvals = np.linspace(samples[:, dim_ind].min(), samples[:, dim_ind].max(), 1000)
            yvals = norm.pdf(xvals, loc=mean[dim_ind], scale=std[dim_ind])
            ax.plot(xvals, yvals, color="green", label="Gaussian approx.")

            # plot the true value if it is given
            if (truth is not None):
                true_value = truth[dim_ind].cpu().item()
                axdict[dim_ind].axvline(
```

```
                true_value, color="red", label="true value")

            # Add axis labels
            axdict[dim_ind].set_xlabel(names[dim_ind])  # x-axis label
            axdict[dim_ind].set_ylabel("Counts")  # y-axis label

            # plot the legend only for the first panel
            if (dim_ind == 0):
                axdict[dim_ind].legend()

        plt.savefig(f"{filename}_{pdf_model}")  # Save the plot to a file
        plt.show()  # Display the plot
        plt.close(fig)


        # === Add 2D heatmap plot for joint distribution ===
        dim1, dim2 = 1, 0  # Example: Teff vs log g
        fig2, ax2 = plt.subplots(figsize=(6, 5))
        h = ax2.hist2d(samples[:, dim1], samples[:, dim2], bins=100, cmap='viridis', density=True, edgecolors='none')
        plt.colorbar(h[3], ax=ax2, label="Density")

        ax2.set_xlabel(names[dim1])
        ax2.set_ylabel(names[dim2])
        ax2.set_title(f"Joint PDF: {names[dim1]} vs {names[dim2]}")

        # if truth is not None:
        #     ax2.plot(truth[dim1].cpu().item(), truth[dim2].cpu().item(), "rx", label="True value")
        #     ax2.legend()

        plt.tight_layout()
        fig2.savefig(f"joint_pdf_heatmap_{pdf_model}")
        plt.show()
        plt.close(fig2)




# ============================= Data related functions ==============================
def get_normalized_data(data_path):
    spectra = np.load(f"{data_path}\spectra.npy")
    spectra_length = spectra.shape[1]
    # labels: mass, age, l_bol, dist, t_eff, log_g, fe_h, SNR
    labelNames = ["mass", "age", "l_bol", "dist", "t_eff", "log_g", "fe_h", "SNR"]
    labels = np.load(f"{data_path}\labels.npy")

    # We only use the three labels: t_eff, log_g, fe_h
    labelNames = labelNames[-4:-1]
    labels = labels[:, -4:-1]
```

```python
    n_labels = labels.shape[1]

   # normalize the spectra and labels via log
    spectra = np.log(np.maximum(spectra, 0.2))

    # scale all labels with minmaxscaler independently and keep the parameters for unscaling.

    scaler = StandardScaler()
    labels = scaler.fit_transform(labels)

    print("Spectra shape: ", spectra.shape)

    return spectra, labels, spectra_length, n_labels, labelNames, scaler


def get_datasets(spectra, labels, split_ratio=0.1, batch_size=64):
    """
    Create datasets for training, validation, and testing.

    Returns
    -------
    tuple
        A tuple containing the training, validation, and test datasets.
    """
    # Create a TensorDataset from the data
    spectra_tensor = spectra.clone().detach()  # Use clone().detach() to avoid warnings. Detach is needed to avoid gradient tracking.
    labels_tensor = labels.clone().detach()
    dataset = TensorDataset(spectra_tensor, labels_tensor)

    n = len(dataset)
    val_size = int(split_ratio * n)
    test_size = int(split_ratio * n)
    train_size = n - val_size - test_size  # Ensure full coverage

    train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])

    print(f"Train size: {len(train_dataset)}, Validation size: {len(val_dataset)}, Test size: {len(test_dataset)}")


    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True) # create
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

    return train_loader, val_loader, test_loader


# ============================= Model related functions =============================
def plot_loss(train_losses, val_losses, test_loss, pdf_model, filename="loss_plot.png"):
```

```
    """
    Plots the training, validation, and test loss.

    Parameters
    ----------
    train_losses : list
        List of training loss values for each epoch.
    val_losses : list
        List of validation loss values for each epoch.
    test_loss : float
        The test loss value.
    filename : str, optional
        The filename where the plot will be saved (default is "loss_plot.png").
    """
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, len(train_losses) + 1), train_losses, label="Training Loss",
marker="o")
    plt.plot(range(1, len(val_losses) + 1), val_losses, label="Validation Loss", marker="o")
    plt.axhline(y=test_loss, color="red", linestyle="--", label="Test Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training, Validation, and Test Loss")
    plt.legend()
    plt.grid(True, linestyle="--", linewidth=0.5)
    plt.savefig(f"{filename}_{pdf_model}")
    plt.show()
    plt.close()


def train_nf_model(model, train_loader, val_loader, device, pdf_model, num_epochs=50,
learning_rate=1e-4, num_grid=1000):
    """
    Trains the normalizing flow model.

    Parameters
    ----------
    model : nn.Module
        The combined normalizing flow and encoder model.
    train_loader : DataLoader
        DataLoader for the training data.
    val_loader : DataLoader
        DataLoader for the validation data.
    device : torch.device
        The device to run training on (CPU, CUDA, or MPS).
    num_epochs : int
        Number of epochs to train.
    learning_rate : float
        Learning rate for the optimizer.

    Returns
```

```
    -------
    model : nn.Module
        The trained model.
    train_losses : list
        List of training loss values for each epoch.
    val_losses : list
        List of validation loss values for each epoch.
    """
    best_val_loss = float("inf")
    epochs_without_improvement = 0
    early_stopping_patience = 10
    reduce_lr_patience = 4

    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=reduce_lr_patience, min_lr=1e-8, verbose=True)

    train_losses = []
    val_losses = []


    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        for batch_inputs, batch_labels in train_loader:
            batch_inputs, batch_labels = batch_inputs.to(device), batch_labels.to(device) # move data to device

            optimizer.zero_grad() # zero the gradients
            loss = nf_loss(batch_inputs, batch_labels, model) # compute the loss
            loss.backward() # update the gradients
            optimizer.step() # update the weights

            running_loss += loss.item() * batch_inputs.size(0) # accumulate the loss wighted by the batch size

        avg_train_loss = running_loss / len(train_loader.dataset) # average the loss over the dataset
        train_losses.append(avg_train_loss)
        print(f"Epoch {epoch+1}/{num_epochs} - Train Loss: {avg_train_loss:.6f}")

        # Optional: Validation loop
        if val_loader is not None:
            model.eval()
            val_loss = 0.0
            with torch.no_grad():
                for val_inputs, val_labels in val_loader:
                    val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)
                    val_loss += nf_loss(val_inputs, val_labels, model).item() * val_inputs.size(0)
```

```

            avg_val_loss = val_loss / len(val_loader.dataset)
            val_losses.append(avg_val_loss)

            for param_group in optimizer.param_groups:
                print(f"              - Val Loss:   {avg_val_loss:.6f} and lr: {param_group['lr']:.2e}")


            scheduler.step(avg_val_loss) # Reduce learning rate on plateau



        # ======== Plot the PDF for a test example every 3 epochs ====== Here we create a automatic range for the grid and evaluate the target pdf
        if epoch % 3 == 0: # # Plot the PDF for a test example every 3 epochs
            model.eval() # set the model to evaluation mode
            with torch.no_grad(): # no gradients needed
                # Grab one test example
                test_input, test_label = next(iter(test_loader)) # get the first batch of the test set
                x_example = test_input[0].unsqueeze(0).to(device) # add batch dimension (s.t. [B, 1, 16 384]) and move to device
                truth = test_label[0].cpu().numpy() # get the true labels for the first batch item

                fig, axes = plt.subplots(1, 3, figsize=(15, 4)) # create a figure with 3 subplots, one for each label
                for i, label_name in enumerate(["$T_{eff}$", "$\log g$", "$[Fe/H]$"]):
                    # Sample from the flow to estimate mean and std
                    samples = model.sample(model.encoder(x_example), samplesize_per_batch-item=1000)[0].cpu().numpy() # pass the input through the encoder (i.e. the CNN) to get the flow parameters
#   and then sample from the target distribution.
                    mean = samples[:, i].mean() # get the mean of the samples for each label
                    std = samples[:, i].std() # get the std of the samples for each label

                    # Create grid: mean ± 6 stds
                    xvals = np.linspace(mean - 4 * std, mean + 4 * std, num_grid) # create a grid of x values for the i-th label
                    grid = np.zeros((num_grid, 3)) # create a grid of 500 points for each label with mean at 0.5 (doesnt really matter, we will overwrite it)
                    grid[:, i] = xvals # set the i-th label to the xvals

                    grid_tensor = torch.tensor(grid, dtype=torch.float32, device=x_example.device) #¤ create a tensor from the grid and move to device
                    flow_params = model.encoder(x_example) # get the flow parameters from the encoder
                    log_pdf, _, _ = model.pdf(grid_tensor, amortization_paramet-ers=flow_params.expand(num_grid, -1)) # evaluate the target distribution log pdf at the grid points
```

```
                    pdf = torch.exp(log_pdf).cpu().numpy() # get the pdf by exponentiating the log pdf

                    # Plotting
                    axes[i].plot(xvals, pdf)
                    #axes[i].axvline(truth[i], color='red', linestyle='--', label='True')
                    axes[i].set_xlabel(label_name)
                    axes[i].set_ylabel("Density")
                    axes[i].set_title(f"Exact PDF - {label_name} (Epoch {epoch+1}) for {pdf_model}")
                    #axes[i].legend()

                plt.tight_layout()
                plt.savefig(f"exact_pdf_epoch_{epoch+1}_{pdf_model}.png")
                plt.close()




            # ===== Early stopping based on validation loss ======
            if avg_val_loss < best_val_loss - 1e-5:
                best_val_loss = avg_val_loss
                epochs_without_improvement = 0
            else:
                epochs_without_improvement += 1
                print(f"              - No improvement for {epochs_without_improvement} epoch(s).")

            if epochs_without_improvement >= early_stopping_patience:
                print("Early stopping triggered.")
                break

    return model, train_losses, val_losses



def evaluate_test_loss(model, test_loader, device):
    """
    Evaluates the model on the test dataset and computes the test loss.

    Parameters
    ----------
    model : nn.Module
        The trained model.
    test_loader : DataLoader
        DataLoader for the test data.
    device : torch.device
        The device to run evaluation on (CPU, CUDA, or MPS).

    Returns
    -------
```

```
    float
        The computed test loss.
    """
    model.eval()
    test_loss = 0.0
    with torch.no_grad():
        for test_inputs, test_labels in test_loader:
            test_inputs, test_labels = test_inputs.to(device), test_labels.to(device)
            test_loss += nf_loss(test_inputs, test_labels, model).item() * test_inputs.size(0)
    test_loss /= len(test_loader.dataset)

    return test_loss



# Define the CNN encoder model. The output of the model is the input to the normalizing flow.
# The latent dimension is the number of parameters in the normalizing flow.
class TinyCNNEncoder(nn.Module):
    def __init__(self, latent_dimension):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Conv1d(1, 8, kernel_size=7, stride=1, padding=3),    # [B, 8, 16384]
            nn.BatchNorm1d(8),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4),                             # [B, 8, 4096]

            nn.Conv1d(8, 16, kernel_size=5, stride=1, padding=2),    # [B, 16, 4096]
            nn.BatchNorm1d(16),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4),                             # [B, 16, 1024]

            nn.Conv1d(16, 32, kernel_size=5, stride=1, padding=2),   # [B, 32, 1024]
            nn.BatchNorm1d(32),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=4),                             # [B, 32, 256]

            nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1),   # [B, 64, 256]
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.AdaptiveAvgPool1d(1),                                 # [B, 64, 1]
        )

        self.project = nn.Sequential(
            nn.Flatten(),                   # [B, 64]
            nn.Linear(64, 256),
            nn.ReLU(),
            nn.Linear(256, latent_dimension)
        )

```

```python


    def forward(self, x):
        x = self.encoder(x)       # Expect input shape: [B, 1, 16384]
        x = self.project(x)       # Output shape: [B, latent_dim]
        return x


def nf_loss(inputs, batch_labels, model):
    """
    Computes the loss for a normalizing flow model, according to maximum likelihood estimation.
    The loss is defined as the negative log probability of the labels given the input data. This loss
    allows the model to learn the true value of the normalizing flow parameters.


    Parameters
    ----------
    inputs : torch.Tensor
        The input data to the model.
    batch_labels : torch.Tensor
        The labels corresponding to the input data.
    model : torch.nn.Module
        The normalizing flow model used for evaluation.
    Returns
    -------
    torch.Tensor
        The computed loss value.
    """
    log_pdfs = model.log_pdf_evaluation(batch_labels, inputs) # get the probability of the labels given the input data
    loss = -log_pdfs.mean() # take the negative mean of the log probabilities
    return loss



# ============================== Quantifying Model Functions =============================
def compute_and_plot_coverage(model, test_loader, device, scaler, pdf_model, n_samples=1000):
    """
    Computes and plots the coverage of the predicted intervals for the test dataset.
    """

    model.eval()
    all_true = []
    all_samples = []

    with torch.no_grad():
        for batch_inputs, batch_labels in test_loader: # get the test data
            batch_inputs = batch_inputs.to(device) # move to device
            batch_labels = batch_labels.to(device)
```

```
            flow_params = model.encoder(batch_inputs) # get the flow parameters from the encoder (i.e. the CNN)
            samples = model.sample(flow_params, samplesize_per_batchitem=n_samples) # sample from the normalizing flow from each pdf using the flow parameters

            all_true.append(batch_labels.cpu().numpy()) # [B, n_labels]
            all_samples.append(samples.cpu().numpy())  # [B, n_samples, 3]

    y_true = np.concatenate(all_true, axis=0) # true labels
    y_samples = np.concatenate(all_samples, axis=0)  # sampled labels

    # === Denormalize using StandardScaler ===
    y_true = scaler.inverse_transform(y_true)
    y_samples = y_samples * scaler.scale_[None, None, :] + scaler.mean_[None, None, :]

    coverage_68 = []
    coverage_95 = []

    # Calculate coverage for each label
    for i in range(y_true.shape[1]):
        lower_68 = np.percentile(y_samples[:, :, i], 16, axis=1)
        upper_68 = np.percentile(y_samples[:, :, i], 84, axis=1)
        coverage_68_i = ((y_true[:, i] >= lower_68) & (y_true[:, i] <= upper_68)).mean() # Fraction of true values inside the model's 68% predicted interval
        coverage_68.append(coverage_68_i)

        lower_95 = np.percentile(y_samples[:, :, i], 2.5, axis=1)
        upper_95 = np.percentile(y_samples[:, :, i], 97.5, axis=1)
        coverage_95_i = ((y_true[:, i] >= lower_95) & (y_true[:, i] <= upper_95)).mean() # Fraction of true values inside the model's 95% predicted interval
        coverage_95.append(coverage_95_i)

    # === Plot ===
    x = np.arange(len(names))
    width = 0.35

    fig, ax = plt.subplots(figsize=(8, 5))
    bars1 = ax.bar(x - width/2, coverage_68, width, label="Observed 68%")
    bars2 = ax.bar(x + width/2, coverage_95, width, label="Observed 95%")

    ax.axhline(0.68, color='gray', linestyle='--', label="Expected 68%")
    ax.axhline(0.95, color='black', linestyle='--', label="Expected 95%")


    # Add values on top of bars
    for bar in bars1:
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2., height + 0.02,
                f'{height:.2f}', ha='center', va='bottom', fontsize=10)

```

```python
    for bar in bars2:
        height = bar.get_height()
        ax.text(bar.get_x() + bar.get_width()/2., height + 0.02,
                f'{height:.2f}', ha='center', va='bottom', fontsize=10)


    ax.set_ylabel("Coverage")
    ax.set_ylim(0, 1.1)
    ax.set_xticks(x)
    ax.set_xticklabels(names)
    ax.set_title(f"Prediction Interval Coverage for {pdf_model}")
    ax.legend()
    ax.grid(True, linestyle="--", alpha=0.5)

    plt.tight_layout()
    plt.savefig(f"coverage_plot_percentiles_{pdf_model}")
    plt.show()



# ============================== Run Code =============================

# Parameters
epochs = 100
pdf_model = "diagonal_gaussian" # choose complexity of the resulting pdf --> more parameters


if __name__ == "__main__":

    # ============================ Setup ============================
    parser = argparse.ArgumentParser() ## Create an argument parser
    parser.add_argument("-normalizing_flow_type", default=pdf_model,
                        choices=["diagonal_gaussian", "full_gaussian", "full_flow"]) # Add an
argument for the normalizing flow type
    args = parser.parse_args() # Parse the arguments
    print("Using normalizing flow type ", args.normalizing_flow_type)

    model = CombinedModel(TinyCNNEncoder, nf_type=args.normalizing_flow_type) # Create the
model with the specified normalizing flow type


    # Detect and use Apple Silicon GPU (MPS) if available
    device = torch.device("cuda" if torch.cuda.is_available() else "mps" if
torch.backends.mps.is_available() else "cpu")
    if args.normalizing_flow_type == "full_flow" and device.type == "mps":
        # MPS does not support double precision, therefore we need to run the flow on the CPU
        fp64_on_cpu = True
    print(f"Using device: {device}, performing fp64 on CPU: {fp64_on_cpu}")
    model.to(device)


```

```
    # ============================ Load and train ============================
    spectra, labels, spectra_length, n_labels, labelNames, scaler = get_normalized_data(DATA_PATH) # Load normalized the data

    spectra_tensor = torch.tensor(spectra, dtype=torch.float32).unsqueeze(1)  # Add a channel dimension for the CNN
    labels_tensor = torch.tensor(labels, dtype=torch.float32)

    print(f"Spectra tensor shape: {spectra_tensor.shape}")  # Should be (batch_size, 1, 16384)


    train_loader, val_loader, test_loader = get_datasets(spectra_tensor, labels_tensor) # Split the data into train, val, and test sets


    model, train_losses, val_losses = train_nf_model(model, train_loader, val_loader, device, pdf_model, num_epochs=epochs)

    test_loss = evaluate_test_loss(model, test_loader, device)
    print(f"Test Loss: {test_loss:.6f}")
    plot_loss(train_losses, val_losses, test_loss, pdf_model, filename="loss_plot.png") # Plot the loss, including test loss


    # ============Visualize the PDF for a batch of input data ============
    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():
        for batch_inputs, batch_labels in test_loader:
            batch_inputs = batch_inputs.to(device)
            print(batch_inputs.shape)
            model.visualize_pdf(
                input_data=batch_inputs,
                filename="pdf_visualization.png",
                pdf_model=pdf_model,
                samplesize=10000,
                batch_index=0,
                truth=batch_labels[0] if batch_labels is not None else None
            )
            break  # Visualize only the first batch


    compute_and_plot_coverage(model, test_loader, device, scaler, pdf_model)







# ============================= End of Code =============================

```