UNCERTANTY QUANTIFICATION IN DNNs

S. Giagu / A. Ciardiello - Sapienza Università di Roma, INFN, and ISS

AIPHY 2024 - Monopoli - 29.9-5.10.2024

SAPIENZA
UNIVERSITÀ DI ROMA

UNCERTAINTY QUANTIFICATION IN ANN

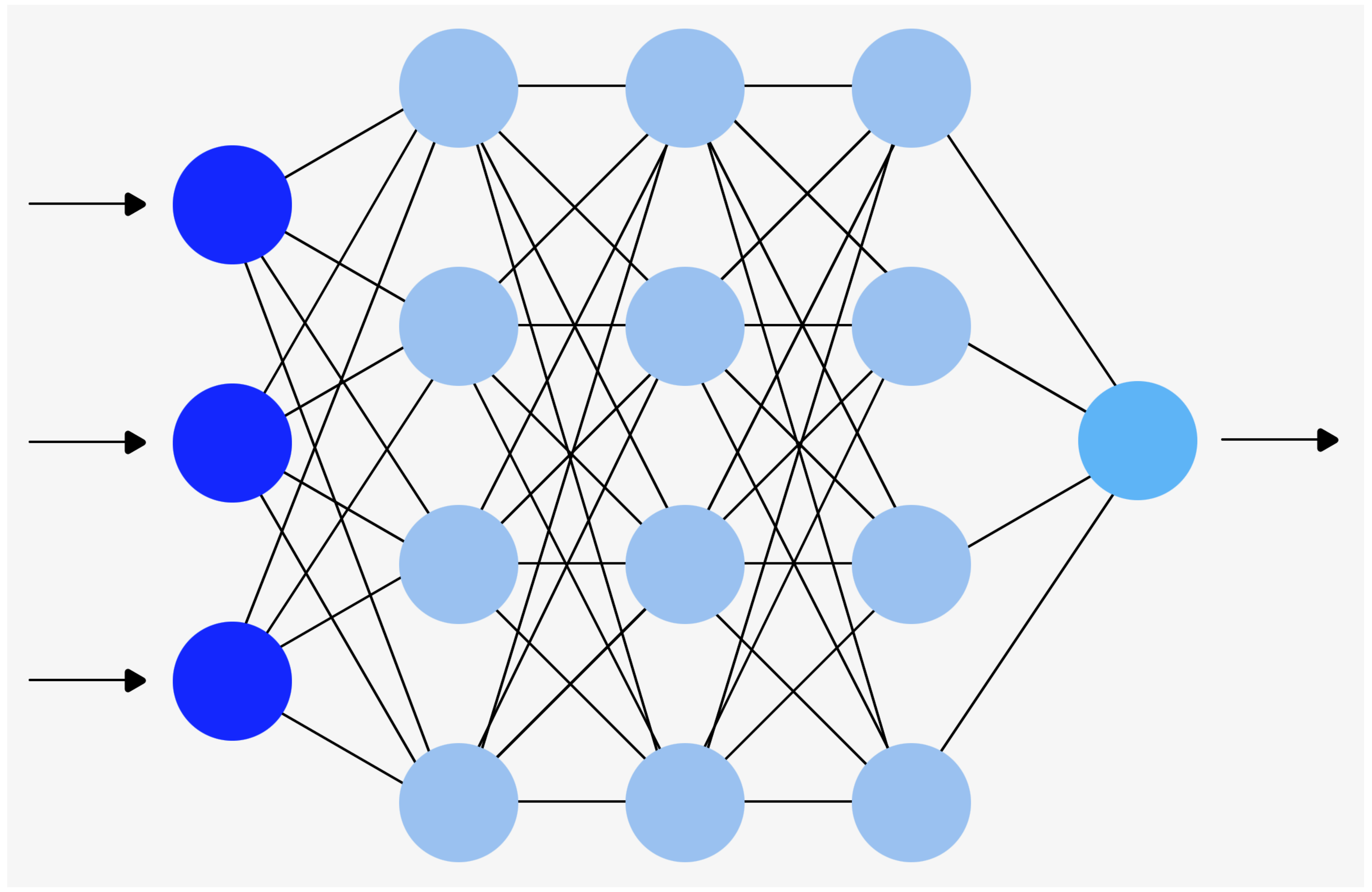
- develop reliable and robust models while retaining high performance is one of the critical aspects that bridges the gap between theoretical predictions and real-world applications in DL
- understanding and quantifying uncertainty is not just academic, but a mandatory requirement for deploying reliable and transparent AI systems in several fields of applied AI: natural sciences, healthcare, autonomous driving, financial forecasting, …

- Definition:
  - HQ in DNN: deploy DL systems that does not only make predictions but also provide measures of confidence in their predictions

- program of this lecture and associated hands-on:
  - discuss foundational understanding of the various types of uncertainties in DNN
  - brief introduction to methodologies for quantifying and managing these uncertainties in neural networks (ensemble methods, Bayesian-NN, conformal methods), and discuss the implications and applications of such practices in real-world scenarios
  - apply in a simple practical use-case the principal uncertainty methods discussed during the course

UNDERSTANDING UNCERTAINTY OF ANNs

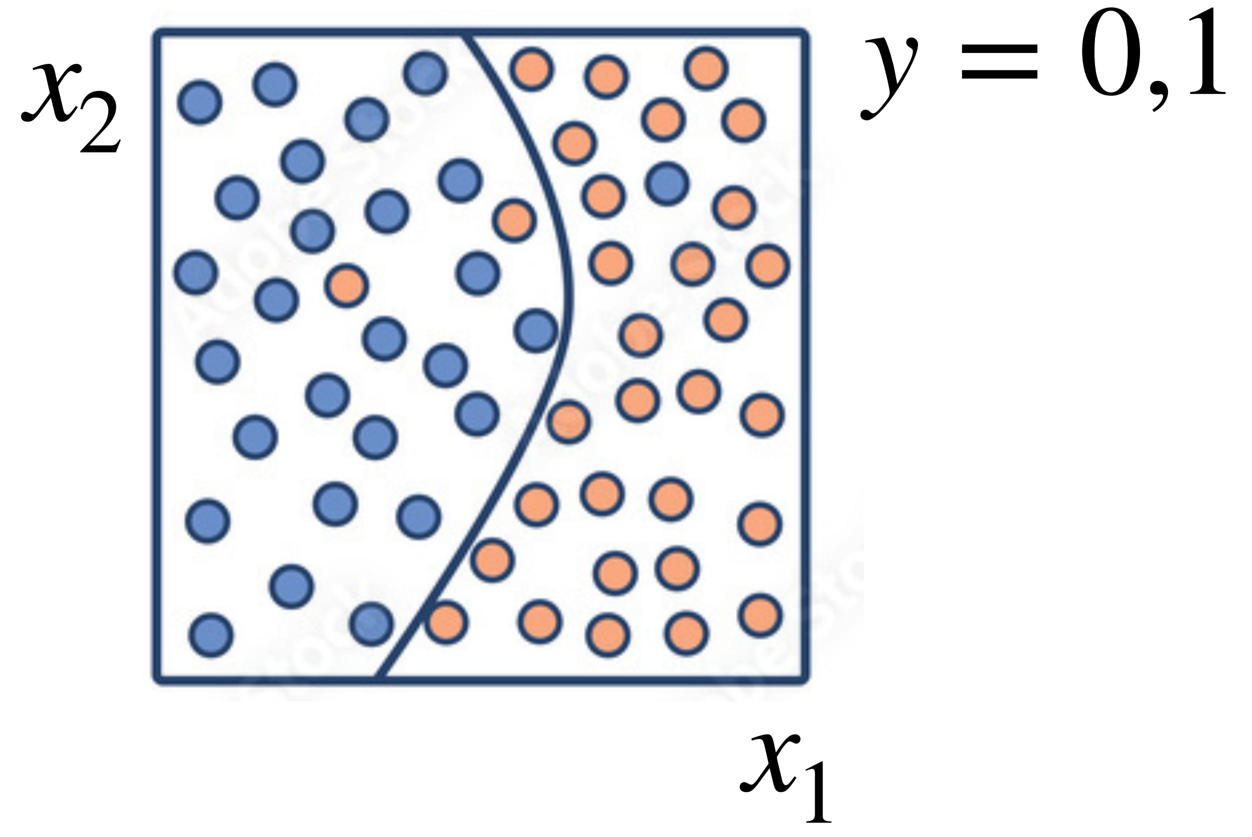
- in practice to provide an uncertainty assessment for an ANN means to return a distribution over a prediction instead of a point prediction



$p(y|x)$

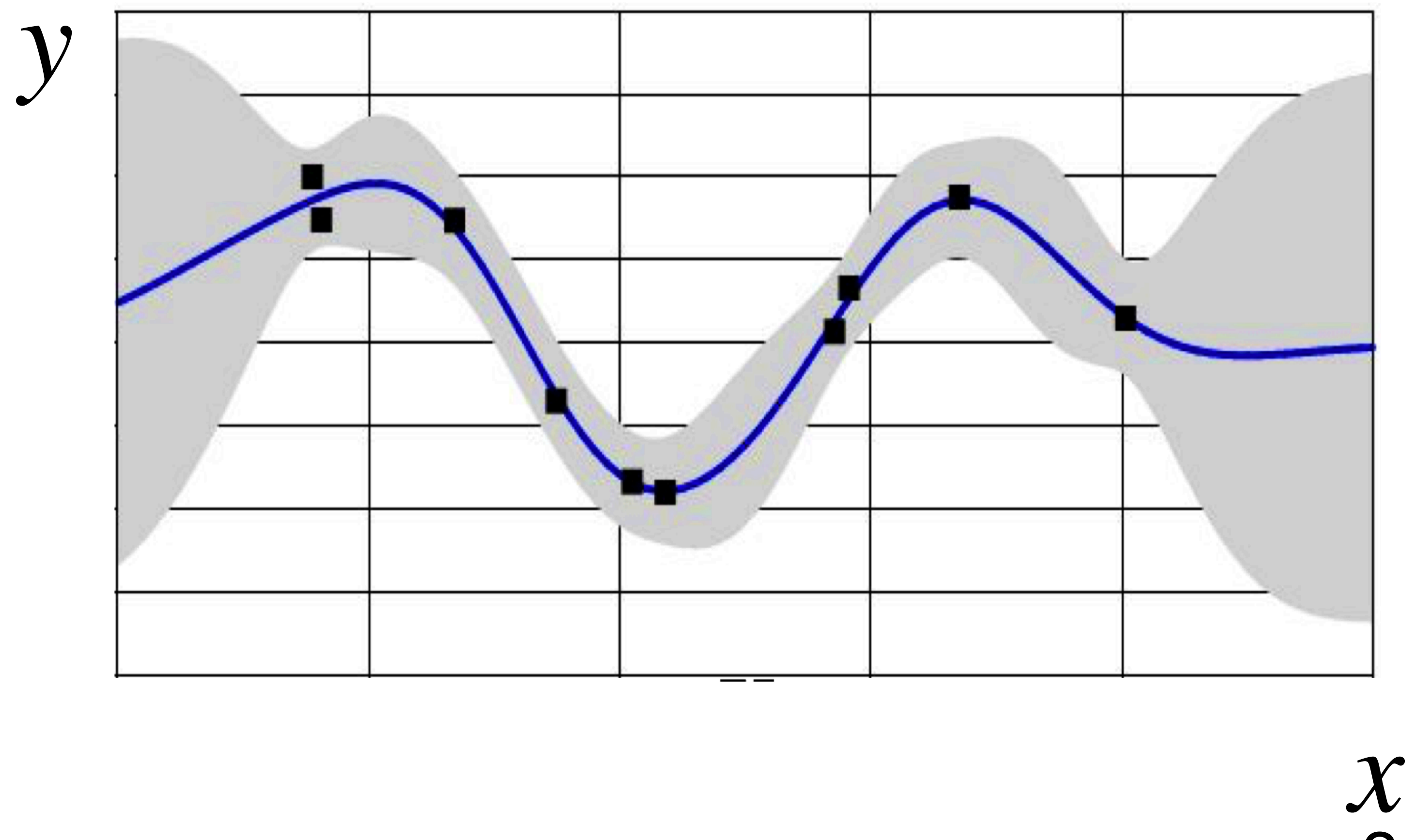
Classification:

output: label + confidence

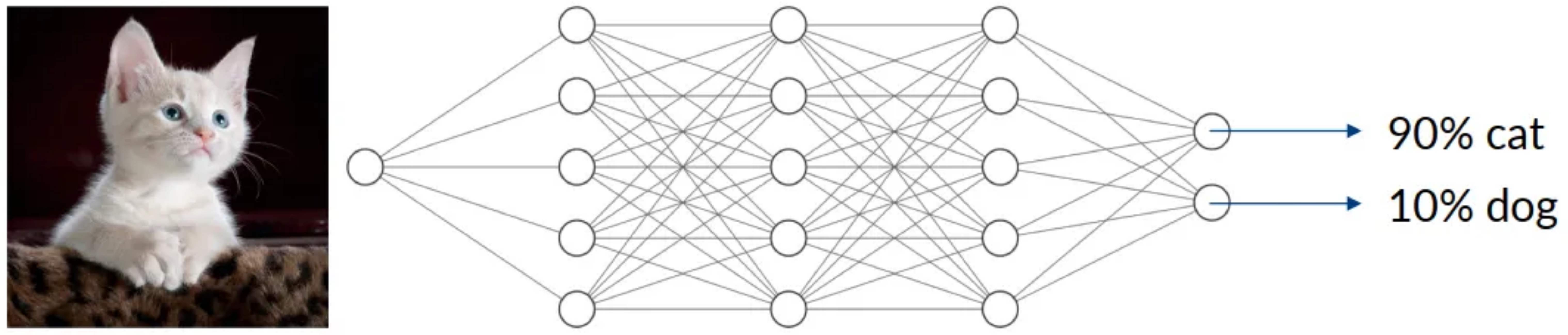


Regression:

output: mean + variance

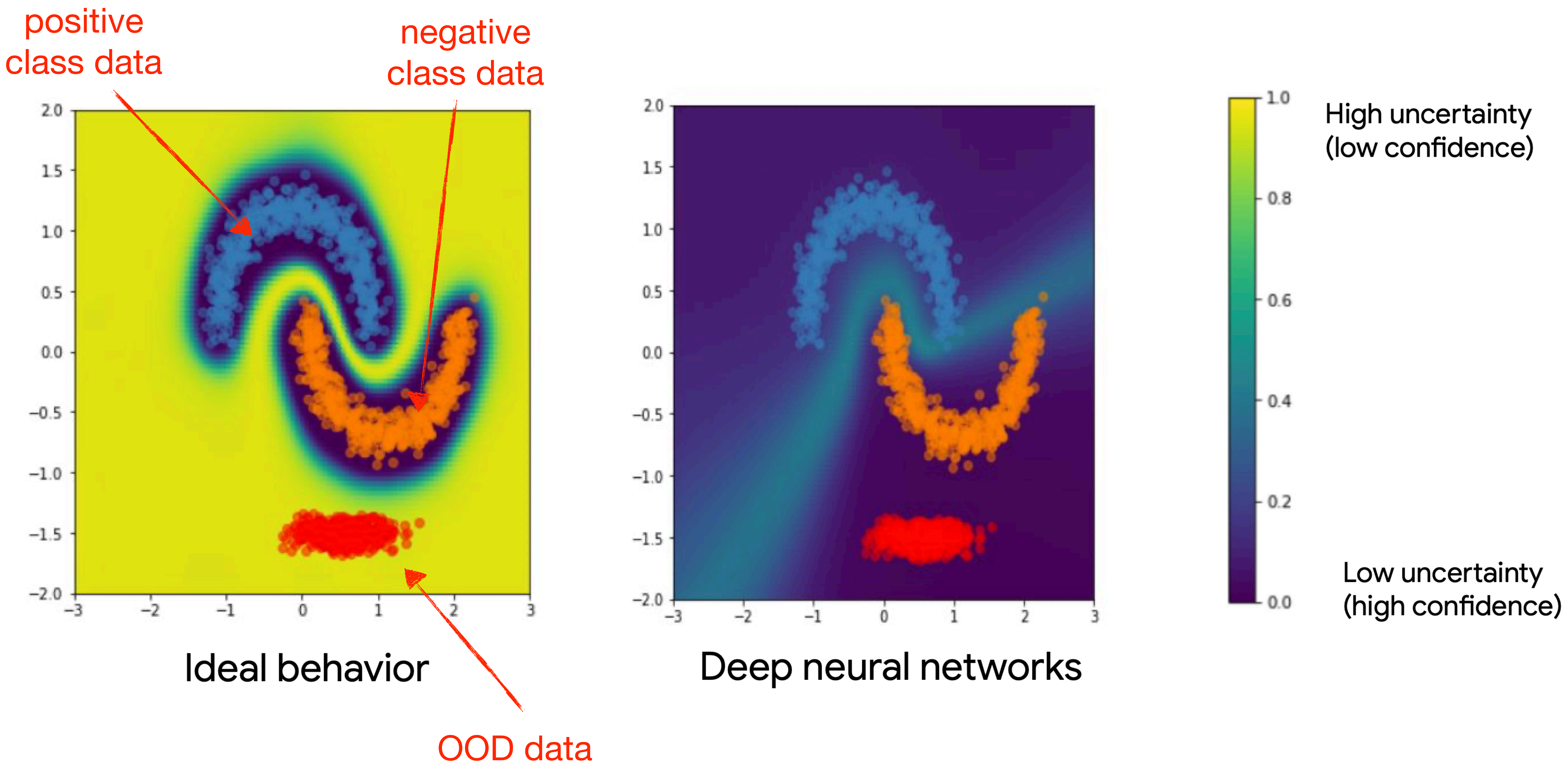


DNN UNCERTAINTY AND ROBUSTNESS OF AN ANN



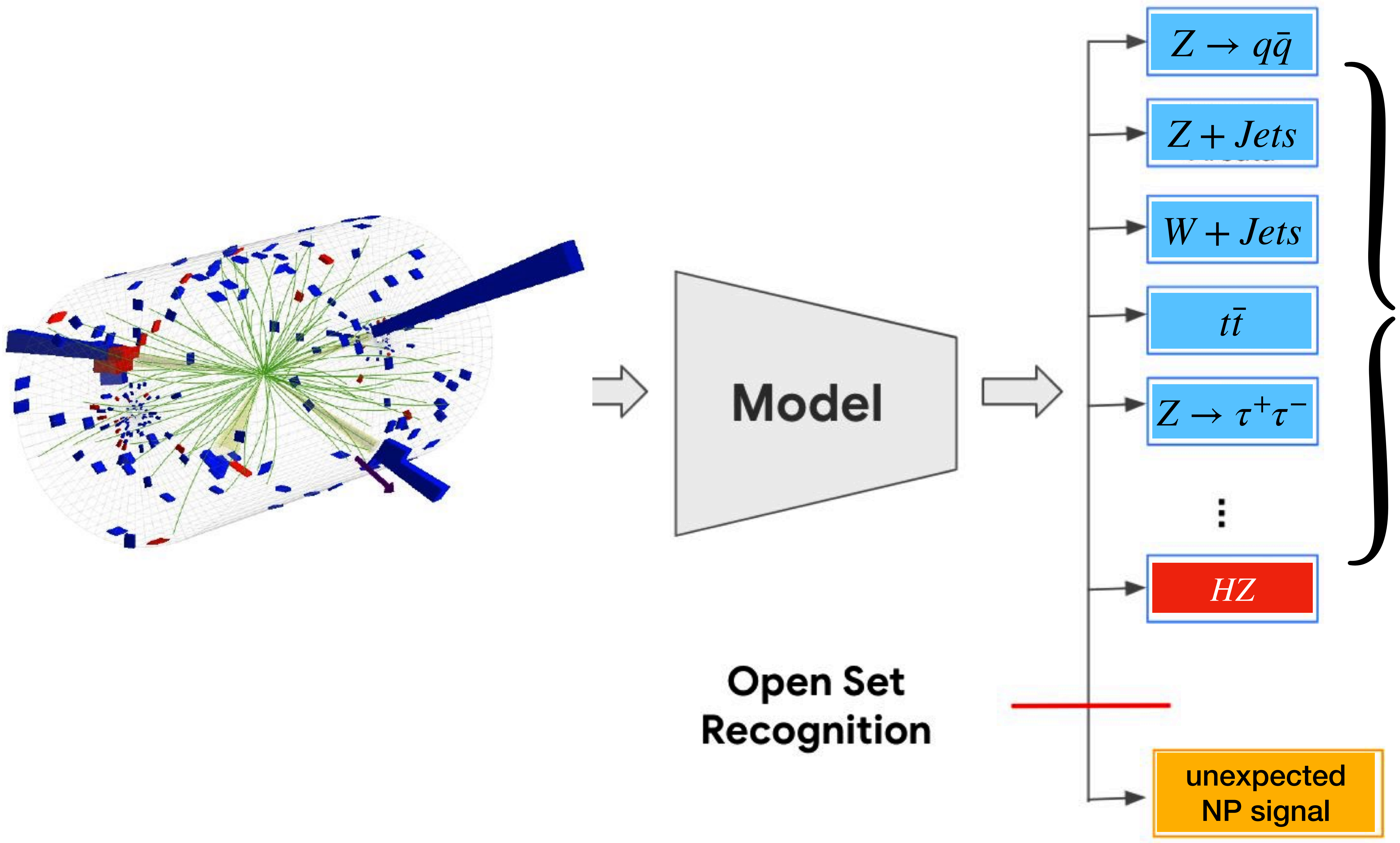
- immagine we train a CNN for the task of recognizing cat & dogs, we know that a state-of-the-art vision model can solve the task with high accuracy measured on I.I.D. (Independent and Identically Distributed) samples, eg samples for which $p_{test}(x, y) = p_{train}(x, y)$
- what happens if in prediction we feed the model with O.O.D. (Out-Of-Distribution) samples, eg samples for which $p_{test}(x, y) \neq p_{train}(x, y)$ (in physics we call these: sources of systematic uncertainty)? For example an image containing a horse …
  - in general a typical NN model trained with softmax output will still predicts with a high probability score the unseen image to belong to one of the two cat/dog classes
- model uncertainty can help deciding when to trust the model in such common situations

EXAMPLE OF ROBUSTNESS AGAINST OOD



J.Z.Liu et al., arXiv:2006.10108 [cs.LG]

OPEN-SET RECOGNITION



the model should be able to say that the specific input doesn't belong to any of these categories

EXAMPLE OF SOURCES OF O.O.D. SHIFTS IN DATA

- several sources: ex. temporal changes, geographical variations, sampling bias, label inconsistencies, sensor/detector changes, over-aggressive/wrong data-augmentation, novelties or rare events, adversarial attacks, data corruption/data loss during transmission/storage/handling of the data …

- different possible effects on the dataset:
  - covariate shift in a dataset: the distribution of the input variables (covariates) $p(x)$ in the training data is different from the distribution of the input variables in the test or real-world application data. However, the conditional distribution of the output variable given the input variables $p(y \mid x)$ remains the same across the training and test datasets → even if the inputs look different between training and testing, the way the output relates to the input doesn't change
  - open-set samples: new categories my appear at inference time
  - sub-population shift: frequencies of data of sub-populations changes during inference
  - label shift: distribution of the labels $p(y)$ changes while $p(y \mid x)$ is fixed, eg while the overall proportion of each class in the target variable changes, the way the features relate to the target within each class does not

TYPE OF UNCERTAINTIES IN ANNs

- ANN uncertainty is usually decomposed in three components:

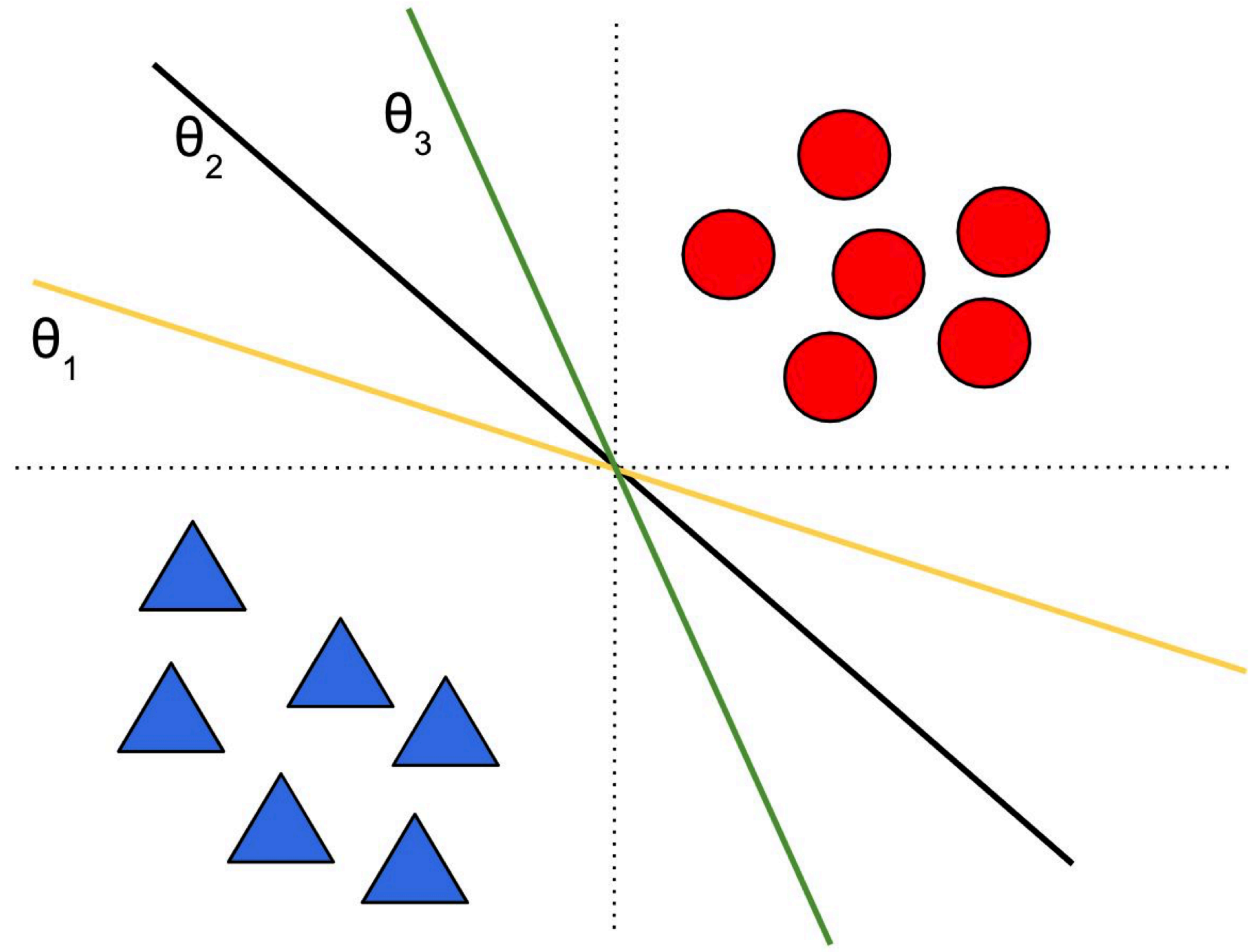
1. approximation uncertainty: related to the expressive power / assumptions of the model
    - the model is not sufficiently expressive to model the data-to-label association
    - the data are not aligned with the inductive biases of the model
        - example: the model assumes rotational invariance of the input while the input is not

- in general is not reducible (w/o changing the model)

TYPE OF UNCERTAINTIES IN ANNs

2. model uncertainty (epistemic uncertainty): accounts for the uncertainty in the model parameters, it arises when the model is not suitably trained due to the lack of training data

- with a finite training set many models can fit the training data well, epistemic uncertainty captures our ignorance about which model actually generated the training set data



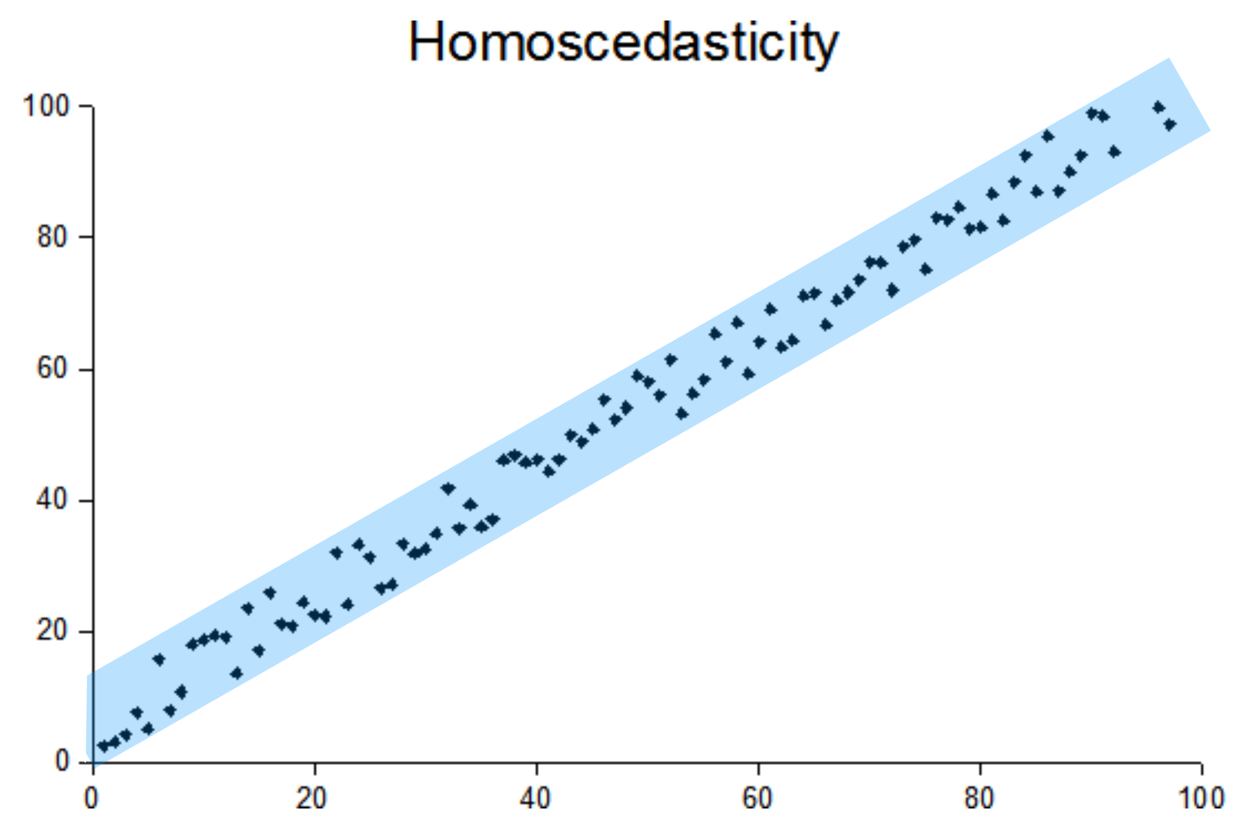
- in general is reducible as vanishes in the limit of infinite data (assuming model identifiability)

TYPE OF UNCERTAINTIES IN ANNs

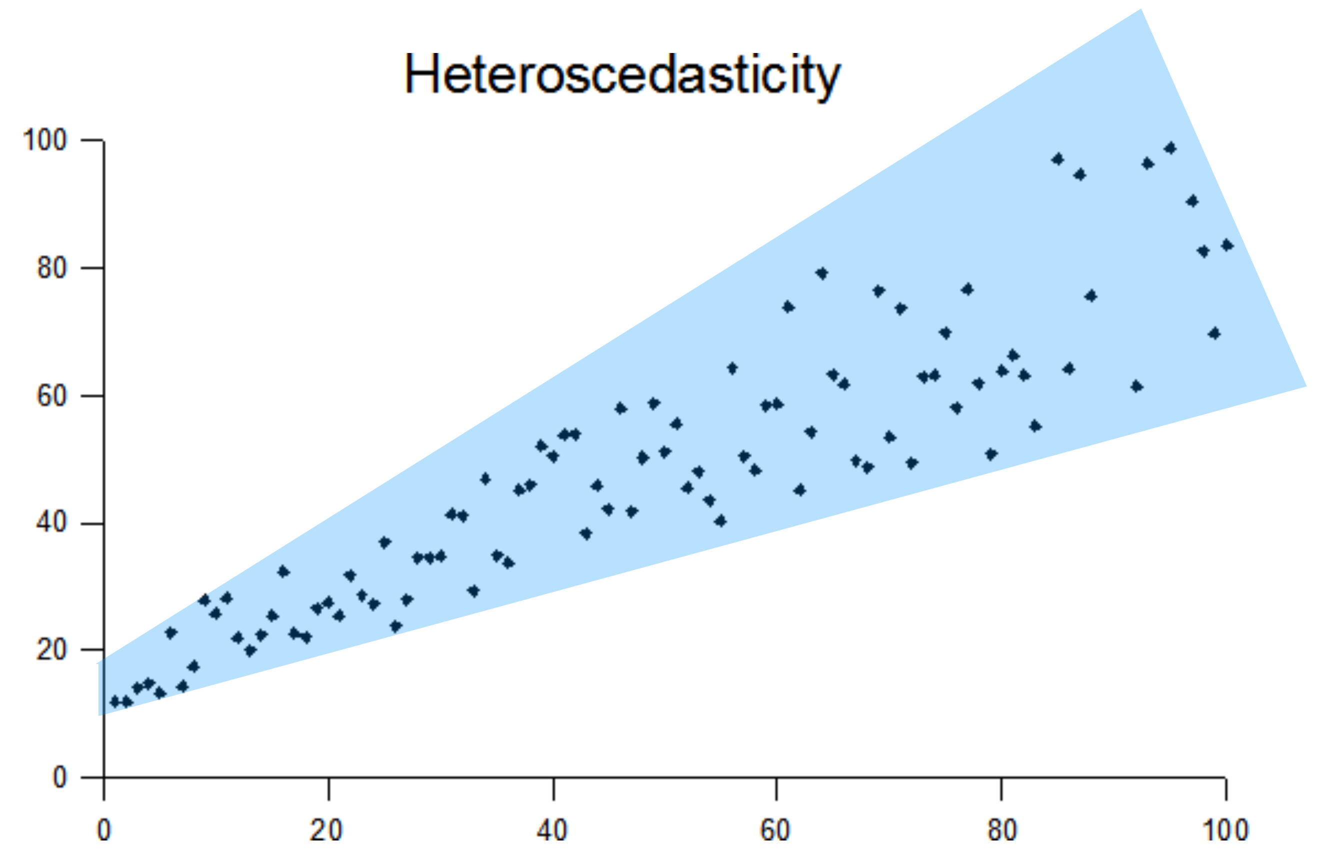
3. data-inherent uncertainty (aleatoric uncertainty): captures inherent noise in the observations (ex. sensor noise, motion noise, labelling noise, …)

   - further classified in:

     - homoscedastic uncertainty: uncertainty that stays constant across the input space



     - heteroscedastic uncertainty: uncertainty that varies with the input (often occurs when there is a large difference among the sizes of the observations)



- in general is not reducible adding additional data to the training set, but can be reduced with additional features/views describing the data

METRICS TO EVALUATE/CORRECT QUALITY OF AN UNCERTAINTY ESTIMATE

- several empirical measures proposed in literature:

- calibration error: $CE = |\underbrace{\text{confidence}}_{\text{predicted probability of correctness}} - \underbrace{\text{accuracy}}_{\text{observed rate of correctness}}|$

example: of the proton-proton collisions predicted to be higgs decay with 90% probability, what fraction did we observe an actual higgs decay?

- 90% → perfect calibration
- <90% → over-confidence
- >90% → under-confidence

- for regression task: calibration corresponds to the coverage of the confidence interval/credibility interval (eg the probability that the confidence interval will include the true value (parameter) of interest)

METRICS TO EVALUATE QUALITY OF AN UNCERTAINTY ESTIMATE

- Expected Calibration Error:

provides an aggregated measure across different probability intervals. It divides predictions into bins, calculates the difference between the average predicted probability and the actual accuracy in each bin, and then computes a weighted average of these differences

set of indices of samples whose prediction confidence falls into the interval *m*

$$ECE = \sum_{m=1}^{M} \frac{n_m}{N} \left| \text{confidence}(B_m) - \text{accuracy}(B_m) \right|$$

group probability predictions in M bins

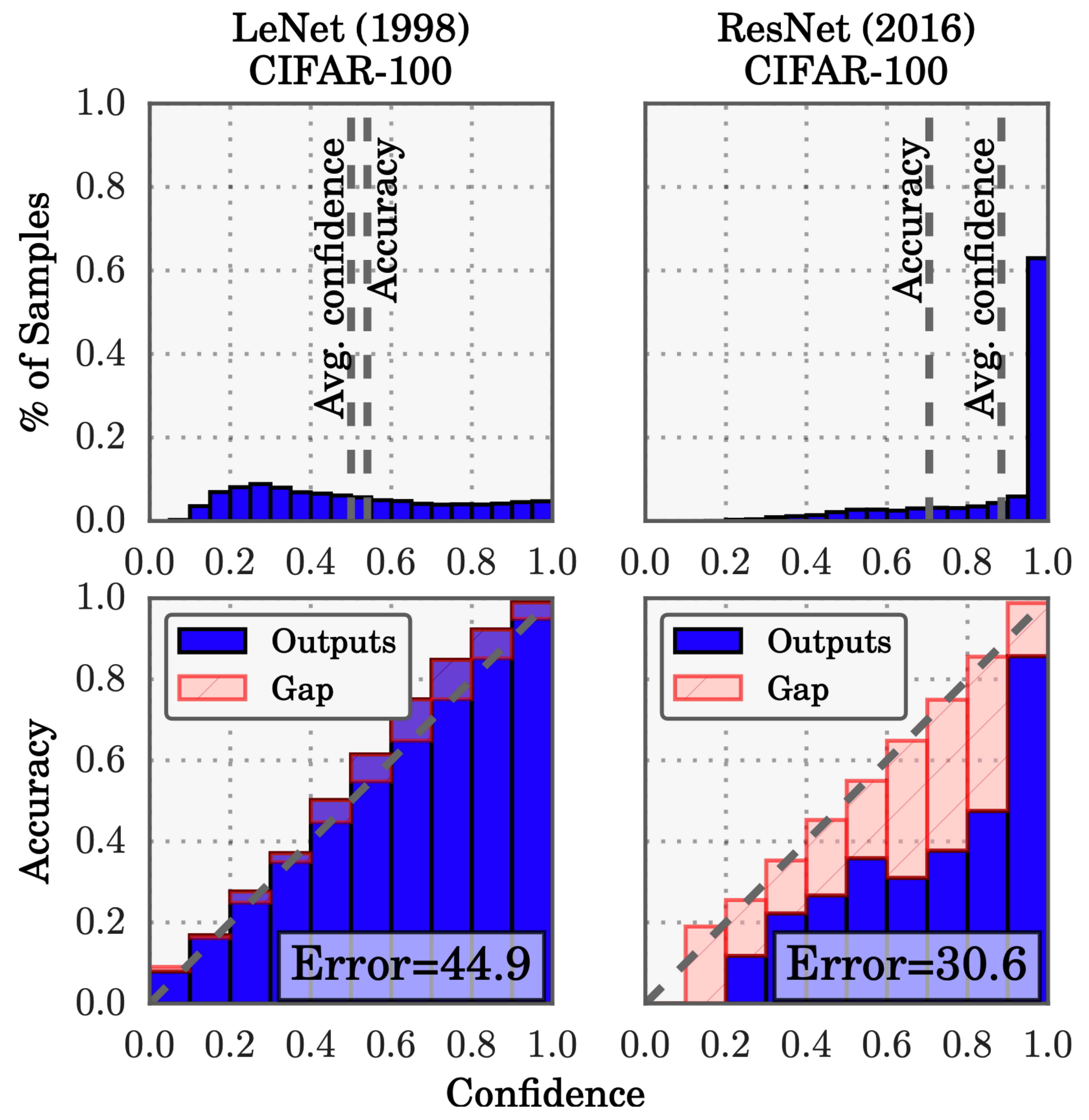
average the calibration error across bins (weighted by number of points in each bin)

within-bin predicted confidence

$$\frac{1}{n_m} \sum_{i \in B_m} \hat{p}_i$$

within-bin rate of correctness

$$\frac{1}{n_m} \sum_{i \in B_m} \mathbf{1}(y_i = \hat{y}_i)$$

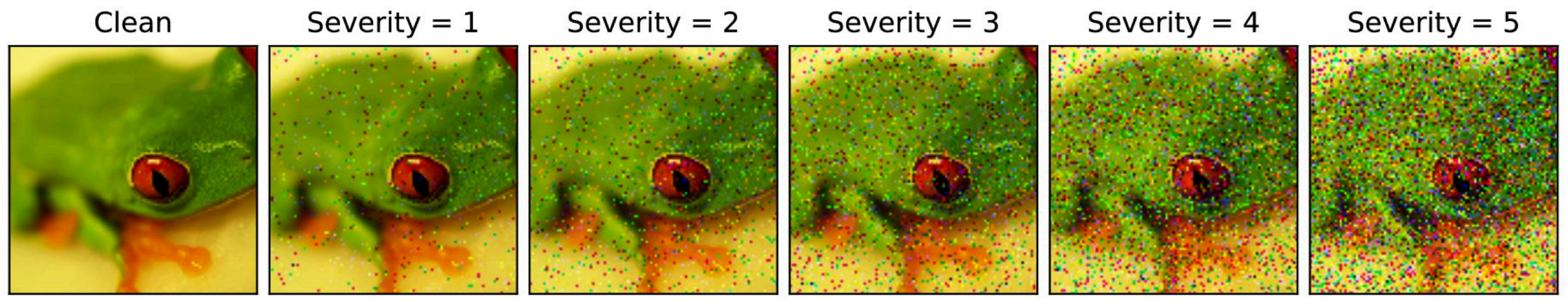


LeNet: slightly under-confident

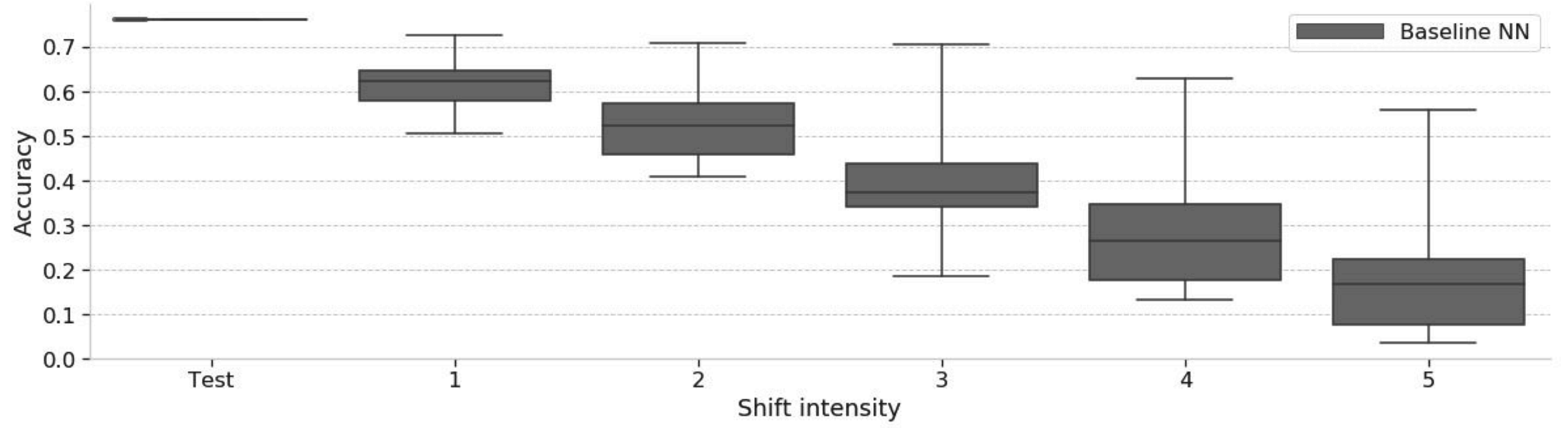
ResNet: over-confident

C.Guo et al., arXiv:1706.04599

EXAMPLE OF ROBUSTNESS AGAINST OOD

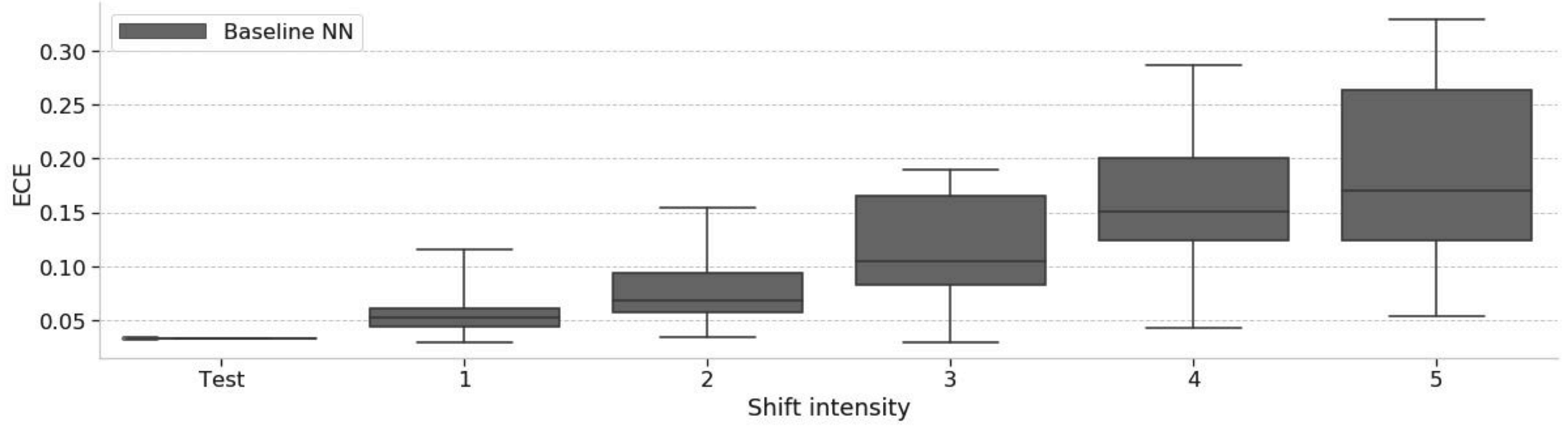


accuracy drop with increasing shift



quality of uncertainty degrade
with increasing shift

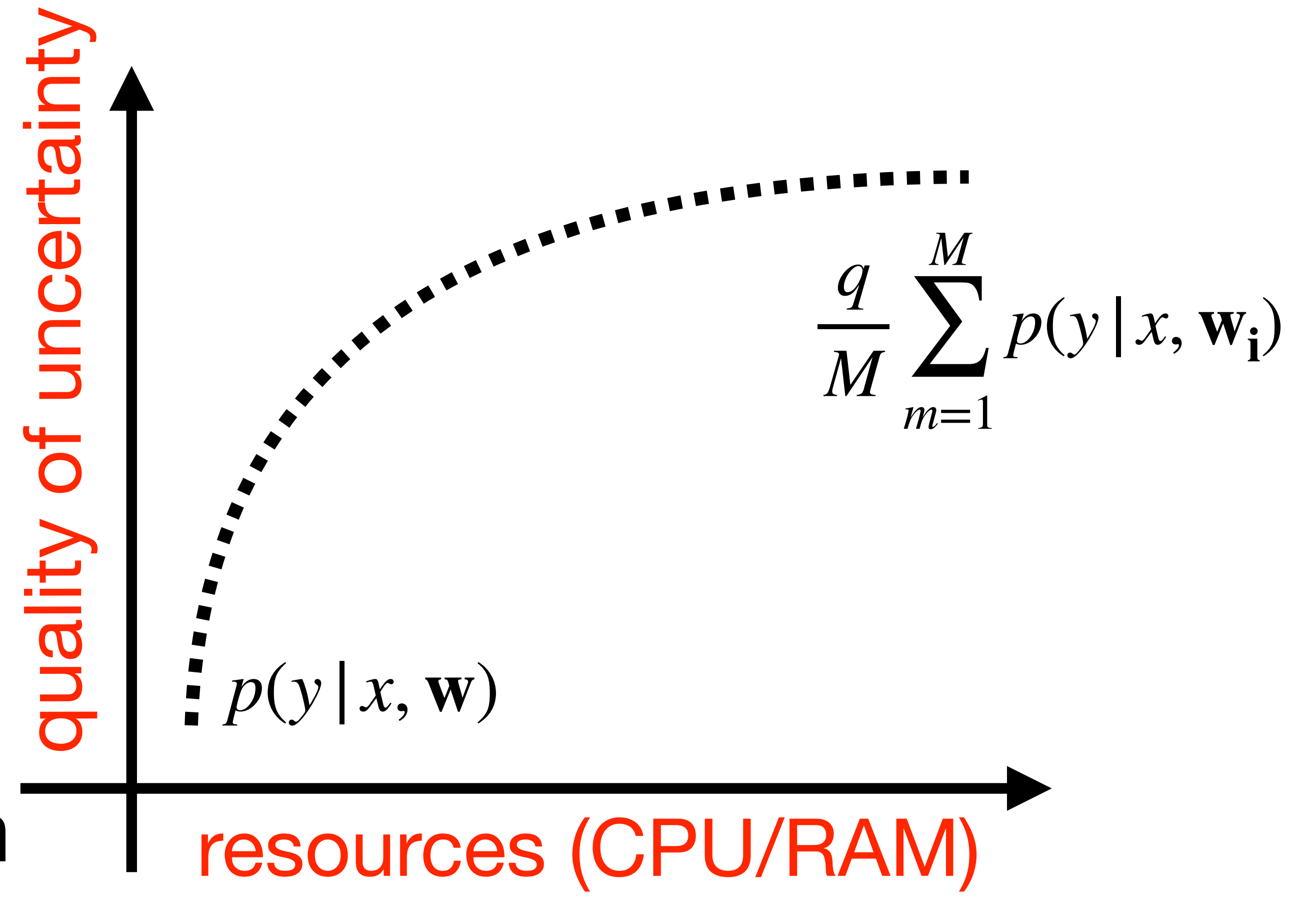
Expected Calibration Error
(measure correspondence between predicted probabilities and empirical accuracy)



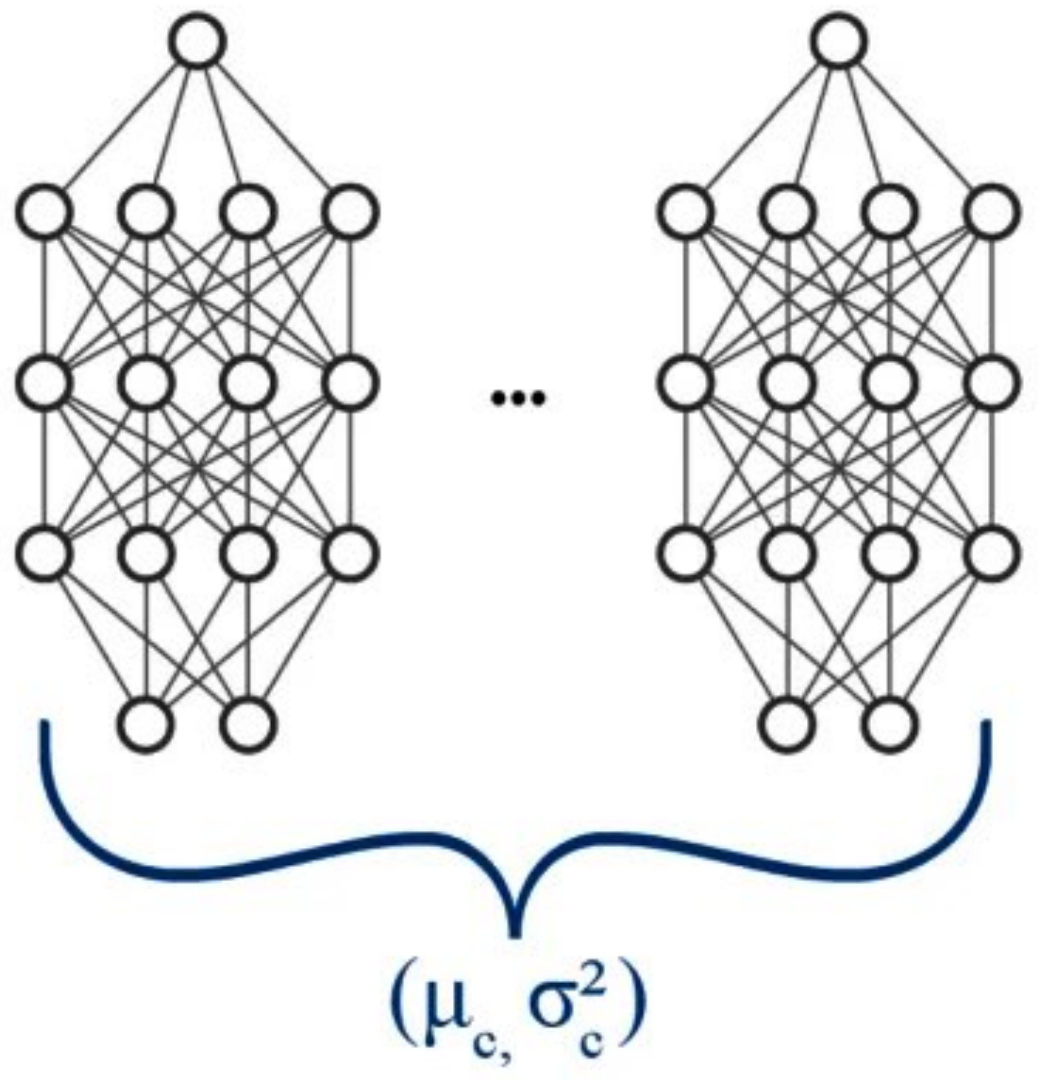
D.Hendrycks, T.Dietterich, arXiv:1903.12261 [cs.LG]
Y.Ovadia et al., arXiv:1906.02530 [stat.ML]

ENSAMBLE-BASED UNCERTAINTY ESTIMATION METHODS

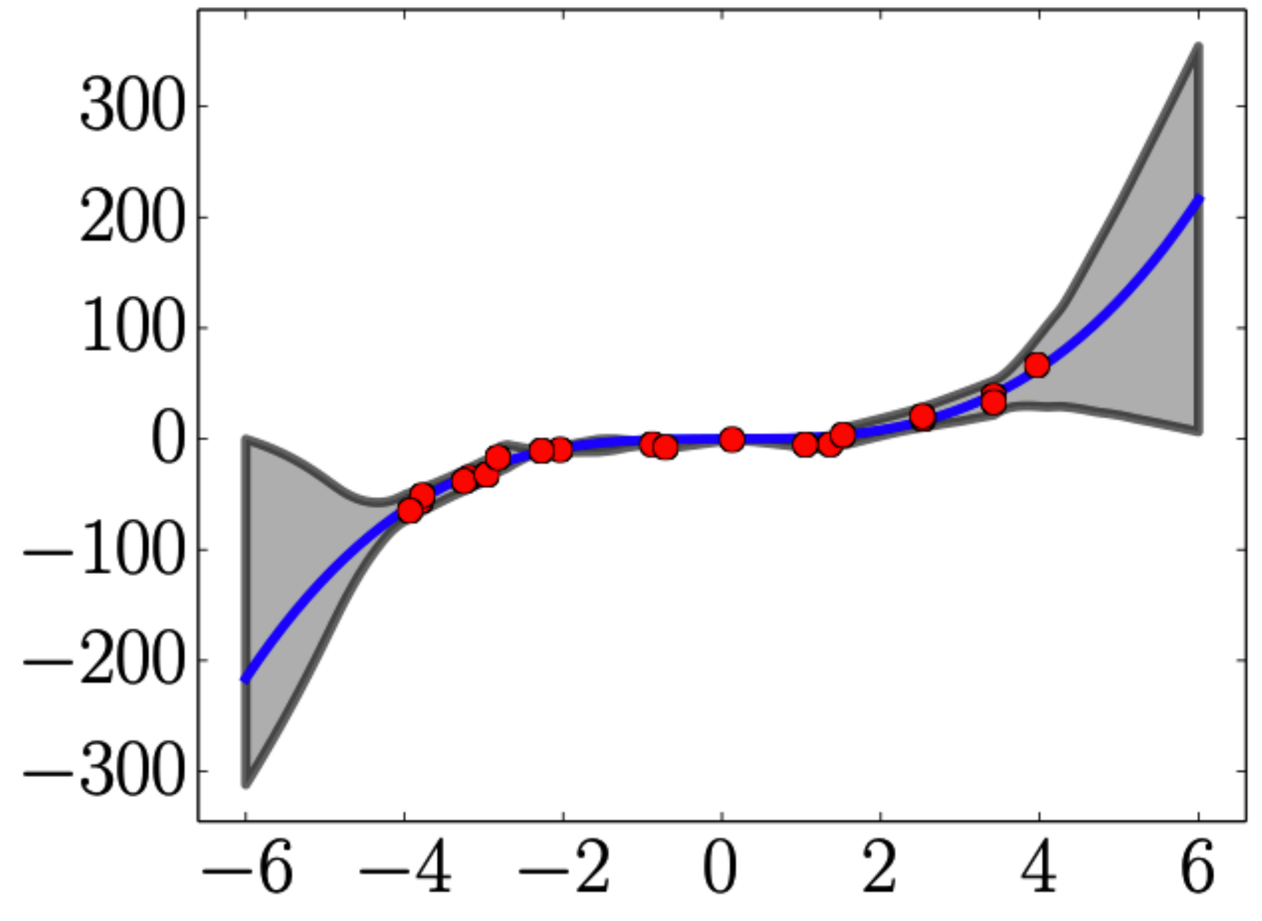
- an intuitive way to quantify epistemic uncertainty is to employ an ensemble of models, instead of relying on a single one
- simple to implement in practice and readily parallelizable
- several strategies related to the collection of models to ensemble, and the aggregation strategy, most popular approach average predictions of independently trained models (bagging, boosting, deep ensembles etc…), forming a mixture distribution



- deep ensembles: multiple instances of a base architecture are obtained each initialised with different weights values, trained on the same dataset, and then averaging their output. Each model output can be considered a sample from the output distribution which expresses model uncertainty



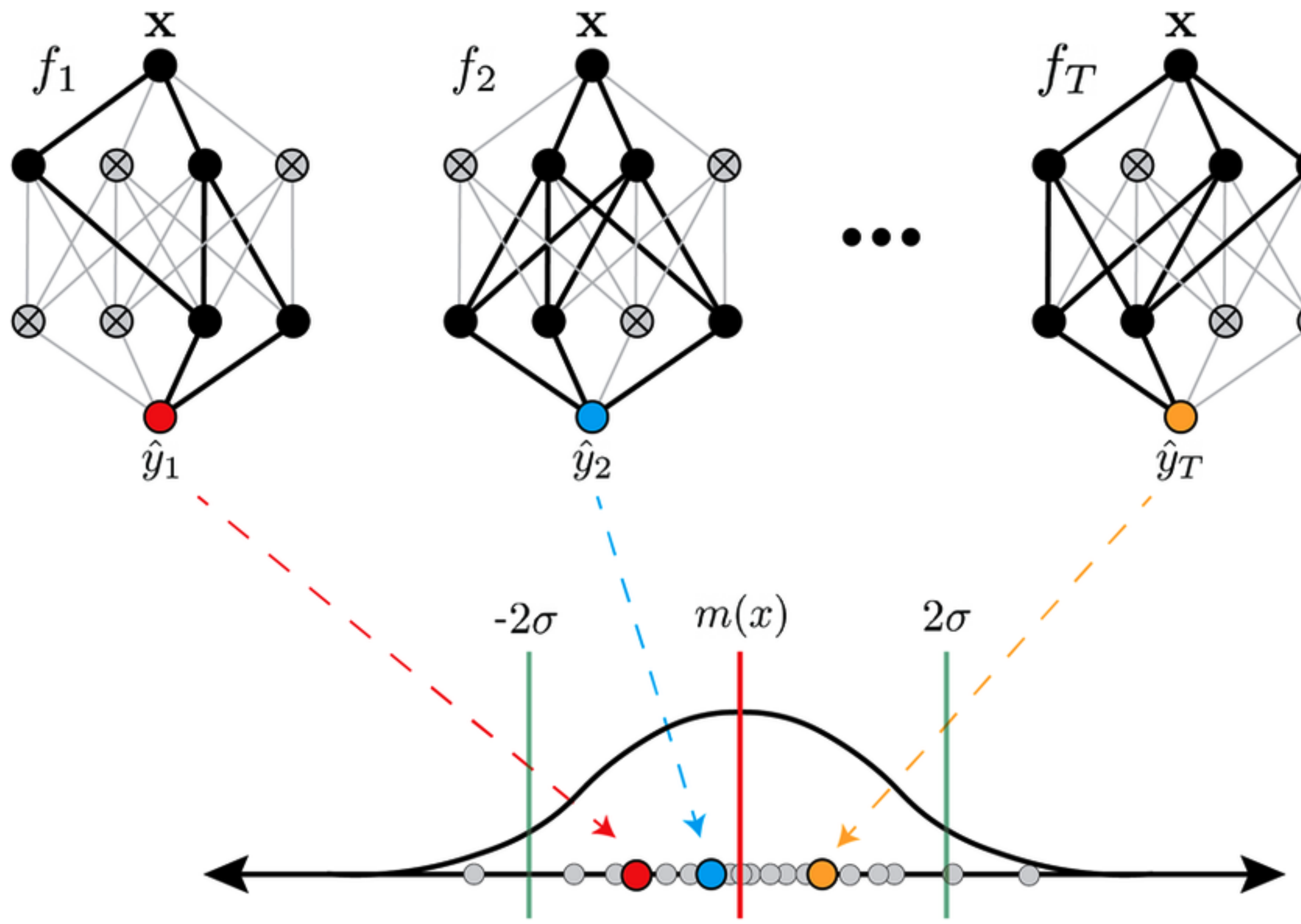
mean of the predictions and variance will be used as metric for the uncertainty



B. Lakshminarayanan et al., arXiv:1612.01474 [stat.ML]

ENSAMBLE-BASED UNCERTAINTY ESTIMATION METHODS

- Monte Carlo Dropout: a very simple and fast method to sample N independent models without requiring multiple and independent trainings
- at training time, connections between layers are randomly dropped with a probability p to avoid overfitting. By keeping dropout enabled also at test time, we can perform multiple forwards sampling of a different network every time



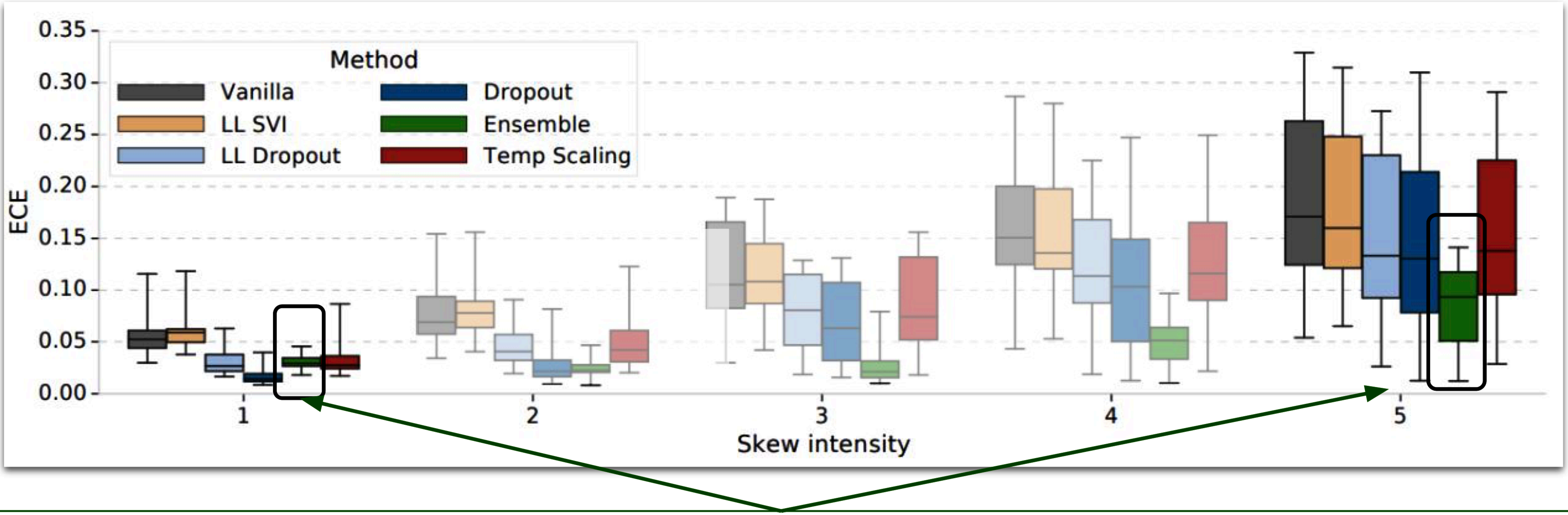
I. train model with dropout layers
II. activate dropout layers at test time
III. repeat T times

$$\mu(x) = \frac{1}{T}\sum_{i=1}^{T} p_i(y\,|\,x)$$

$$\sigma^2(x) = \frac{1}{T-1}\sum_{i=1}^{T} (p_i(y\,|\,x) - \mu(x))^2$$

Y.Gal et al., arXiv:1506.02142

DEEP ENSAMBLES WORKS VERY WELL IN PRACTICE

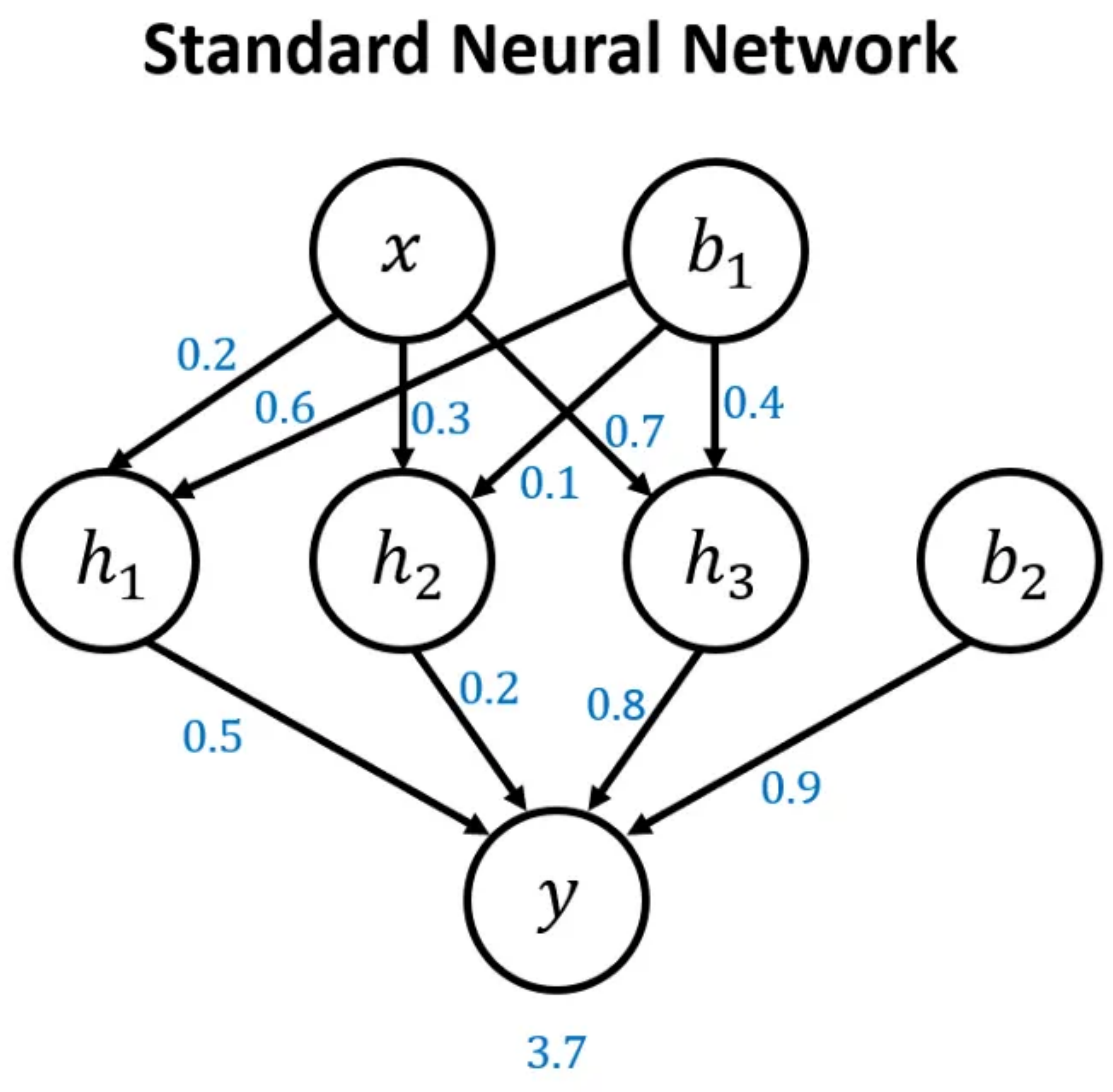


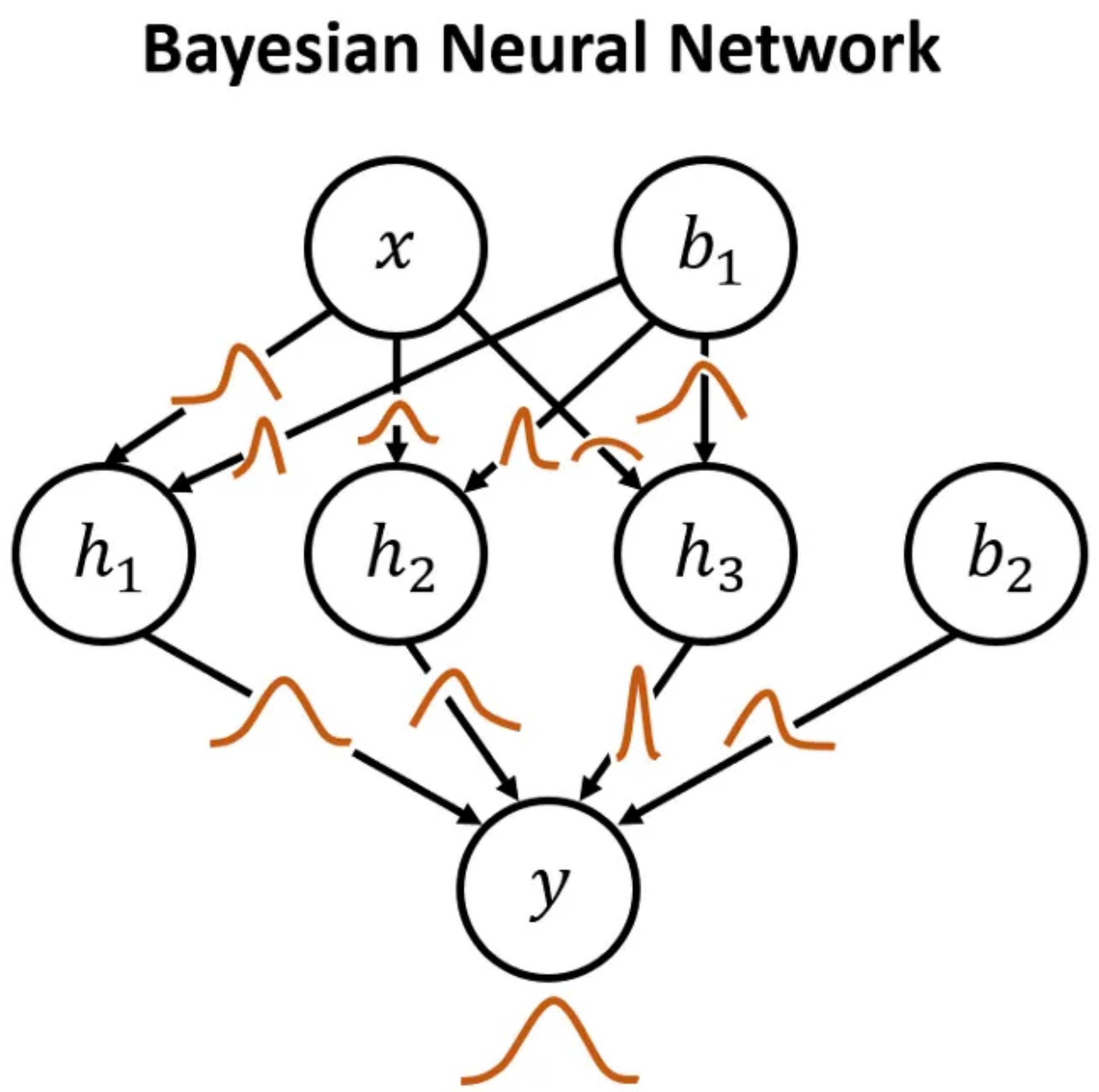
Deep Ensembles are consistently among the best performing methods, especially under dataset shift

Y.Ovadia et al., arXiv:1906.02530 [stat.ML] 16

BAYESIAN NEURAL NETWORKS

- A Bayesian neural network is a probabilistic model that allows to quantify uncertainty in predictions by representing the weights and biases of the network as probability distributions rather than fixed values
  - allows to incorporate prior knowledge about the weights and biases into the model, and update our beliefs about them as we observe data
  - it can simulate multiple possible models of parameters ***w*** with an associated probability distribution *p(**w**)*. By comparing these multiple predictions, it is possible to obtain an estimation of the model's prediction uncertainty. If the different models agree, then the uncertainty is low. If they disagree, then the uncertainty is high





BAYESIAN NEURAL NETWORKS

- the procedure to build and train a bayesian NN consists in two steps:
  - design of the neural network architecture: eg the functional model $y = f_w(x)$
  - choose the probabilistic distributions for the parameters of the model (prior): $p(w)$, and the model confidence: $p(y|x, w)$

$$T = \{(x_i, y_i)\} \; i = 1\cdots, N$$

$$p(w|T) = \frac{p(T|w)p(w)}{p(T)} = \frac{p(T|w)p(w)}{\int_w p(T|w)p(w)dw}$$

posterior distribution of the model parameters

intractable: approximated via Monte Carlo or Variational Inference

$$p(y|x, T) = \int_w p(y|x, w)p(w|T)dw$$

marginal distribution allows to quantify model's uncertainty

MC EXAMPLE PROCEDURE

- with the MC approach we use a finite set of random samples to approximate an expected value
- in the specific case: generate a set of neural networks asymptotically distributed according to $p(w \mid T)$ in order to approximate $p(\hat{y} \mid x, T)$ as the empirical expectation of the single-network predictions $p(\hat{y} \mid x, w)$ under the sampled networks

MC Algorithm

```
define $p(w|T) = \frac{p(T|w)p(w)}{\int_w p(T|w)p(w)dw}$
for $i = 0$ to $N$ do
    draw $w_i \sim p(w|T)$
    $y_i = f_{w_i}(x)$
end for
return $y = \{y_i | i \in [0, N(\}, w = \{w_i | i \in [0, N)\}$
```

set of samples of the marginal and set of samples from the posterior *p(w)*

estimator $\hat{y}$ of the output y

$$\hat{y} = \frac{1}{N} \sum_{w_i} f_{w_i}(x)$$

$$\hat{\Sigma} = \frac{1}{N-1} \sum_{w_i} (f_{w_i}(x) - \hat{y})(f_{w_i}(x) - \hat{y})^T$$

- it is common to assume a normal distribution for the prior

generative process

$$w \sim p(w) = N(\mu, \Sigma)$$

$$y \sim p(y|x,w) = N(f_w(x), \Sigma)$$ regression

$$y \sim p(y|x,w) = Categorical(f_w(x))$$ classification

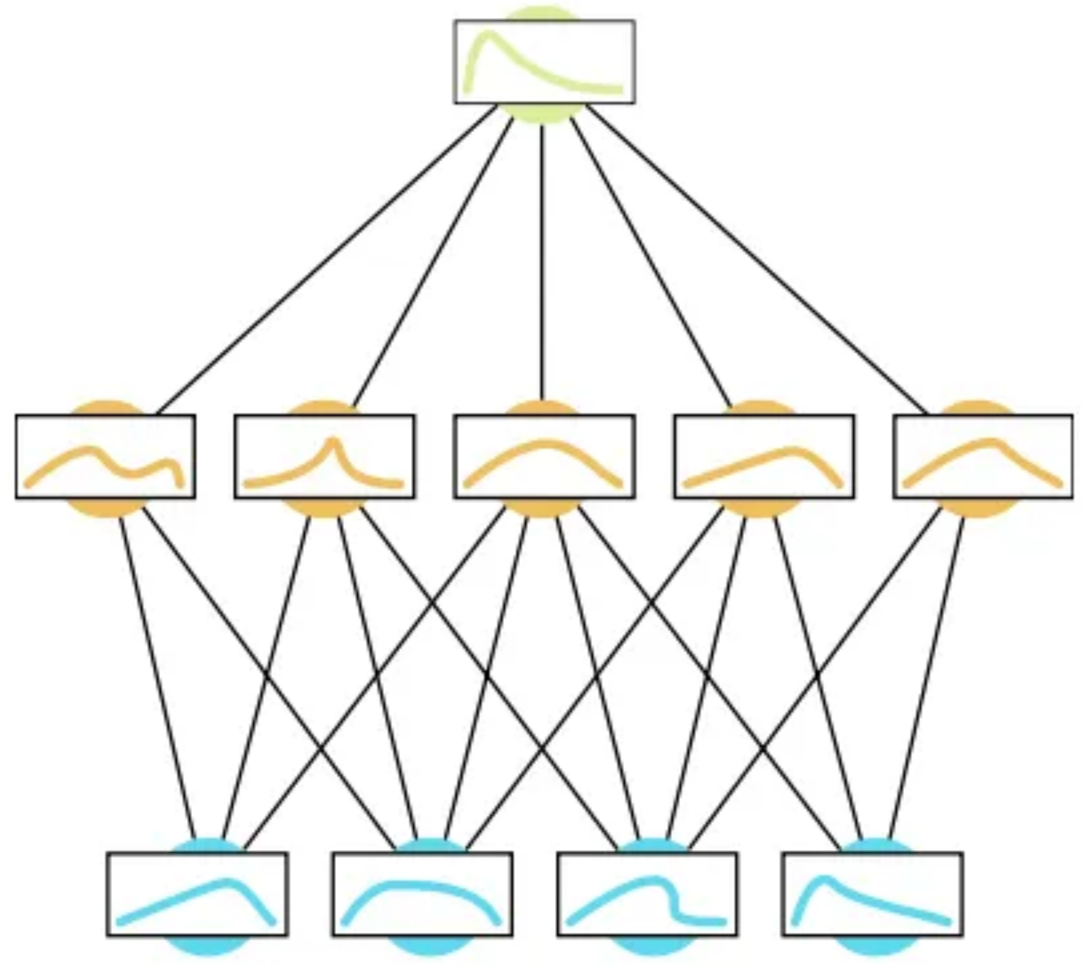
- NOTE: are also possible BayesianNN with stochastic activations instead of stochastic weights (similar to Kolmogorov-Arnold NN —KAN)

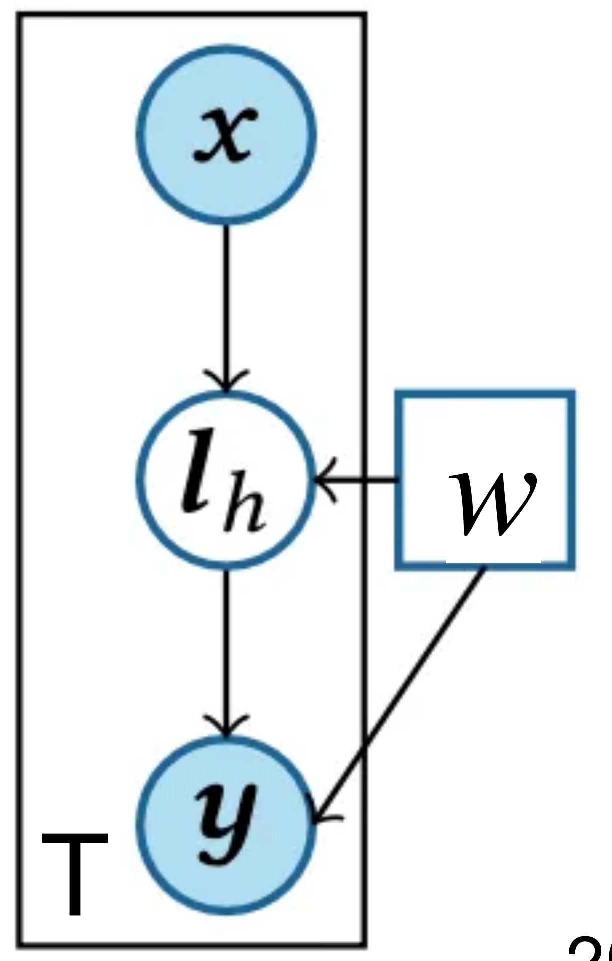
generative process

$$l_0 = x$$

$$l_i \sim a_i(l_i | l_{i-1}) = a_i(N(W_i l_{i-1} + b_i, \Sigma)) \; \forall i \in [1,L]$$

$$y = l_L$$





VARIATIONAL INFERENCE PROCEDURE

- with the MC approach there is no learning phase for the Bayesian-NN as it is sufficient to sample the posterior $p(w \mid T)$ to obtain its estimator
- the approach however suffers of the curse of dimensionality, as directly sampling the posterior becomes harder and harder with the increase of the dimensionality of the sampling space
- to cope with this issue several techniques of approximate inferences have been proposed, with the variational inference being the most popular

- same idea of the variational method used in VAE:
  - approximate the intractable density $p(w \mid T)$ with a tractable parametrised and amortized density (eg a NN): $q_\phi(w)$, (where ϕ represents the variational parameters), chosen among a family of probability densities Q
  - goal: find $q_\phi(w)$ that best approximate the posterior by minimising wrt ϕ the KL divergence: $KL(q_\phi(w) \| p(w \mid T))$
  - in practice as directly minimising the KL is also not tractable because $p(w \mid T)$ is not tractable, the ELBO (Evidence Lower Bound) is used as tractable surrogate of the $KL(q_\phi(w) \| p(w \mid T))$:

$$KL(q_\phi(w) \| p(w \mid T)) = \int q_\phi(w) \log \frac{q_\phi(w)}{p(w \mid T)} dw = E_q[\log \frac{q_\phi(w)}{p(w \mid T)}] =$$

$$= E_q[\log q_\phi(w)] - E_q[\log p(w \mid T)] =$$

$p(w, T) = p(w \mid T) p(T)$

$$= E_q[\log q_\phi(w)] - E_q[\log \frac{p(w, T)}{p(T)}] =$$

$$= E_q[\log q_\phi(w)] - E_q[\log p(w, T)] + E_q[\log p(T)] =$$

$$= E_q[\log q_\phi(w)] - E_q[\log p(w, T)] + \int q_\phi(w) \log p(T) dw =$$

$$= E_q[\log q_\phi(w)] - E_q[\log p(w, T)] + \log p(T) \int q_\phi(w) dw =$$

$$= \underbrace{E_q[\log q_\phi(w)] - E_q[\log p(w, T)]}_{\text{-ELBO}} + \log p(T)$$

constant wrt $q_\phi(w)$, so we can ignore it in the optimisation procedure

$$ELBO(q_\phi(w)) = -\left(KL(q_\phi(w) \| p(w \mid T)) - \log p(T)\right) =$$

$$= -\left(E_q[\log q_\phi(w)] - E_q[\log p(w, T)] + \underbrace{\cancel{\log p(T) - \log p(T)}}_{\text{cancel}} \right) =$$

$$= E_q[\log p(w, T)] - E_q[\log q_\phi(w)] =$$

$$\swarrow \; p(w, T) = p(T \mid w) p(w)$$

$$= E_q[\log p(T \mid w)] + E_q[\log p(w)] - E_q[\log q_\phi(w)] =$$

$$= E_q[\log p(T \mid w)] + E_q\left[\log \frac{p(w)}{q_\phi(w)}\right] =$$

$$= E_q[\log p(T \mid w)] - E_q\left[\log \frac{q_\phi(w)}{p(w)}\right] =$$

$$= E_q[\log p(T \mid w)] - KL(q_\phi(w) \| p(w)) \longleftarrow \text{tractable !}$$

by maximizing the ELBO wrt ϕ, we indirectly minimize $KL(q_\phi(w) \| p(w \mid T))$

OPTIMIZING WITH GRADIENTS

- variational inference re-frames the computation of an integral (the marginal likelihood from exact inference) as the optimization of its lower bound
- the most interesting point is that this implies that we can now use tools from the optimization literature to approximately solve our inference problem. This includes optimizing with the same stochastic gradients descent methods used in standard neural network training
- the simples and most popular choice for the variational density $q_\phi(w)$ is a diagonal-covariance gaussian distribution (note that it may not always the best choice in order to model the true posterior $p(w \mid T)$)
- with this assumption all the network weights are jointly distributed according to the multivariate Gaussian:

$$q_\phi(w) = q_{(\mu,\sigma)}(w) = N(\mu, \sigma^2\mathbf{I})$$

$$\Rightarrow \log q_{(\mu,\sigma)}(w) = -\frac{1}{2}(w-\mu)^T(\sigma^2\mathbf{I})(w-\mu)$$

loss $L(\mu,\sigma) = E_T[ELBO(q_\phi(w))] = E_T[\ E_{(\mu,\sigma)\sim q(w)}[\log p(T \mid w) + \log p(w) - \log q_{(\mu,\sigma)}(w)]\ ]$

we want to optimise *L* with gradients, so we need to compute $\nabla_\mu L(\mu,\sigma)$ and $\nabla_\sigma L(\mu,\sigma)$

- we can use SGD through the expectation E_T over data using the batches, but there is a problem with the second expectation value $E_{(\mu,\sigma)\sim q(w)}[]$ as the distribution $q(w)$ depends on the parameters μ, σ that we want to optimise

$$L(\mu,\sigma) = E_T[\ E_{(\mu,\sigma)\sim q(w)}[\log p(T\,|\,w) + \log p(w) - \log q_{(\mu,\sigma)}(w)]\ ] \Rightarrow$$

$$\nabla_{\mu,\sigma} L(\mu,\sigma) = \frac{1}{N}\sum_{i=1}^{N}[\,\nabla_{\mu,\sigma} E_{(\mu,\sigma)\sim q_{(\mu,\sigma)}(w)}[\log p(T\,|\,w) + \log p(w) - \log q_{(\mu,\sigma)}(w)]$$

we cannot proceed with the gradient due to this dependence

- solution: the reparameterization trick used also in VAE :

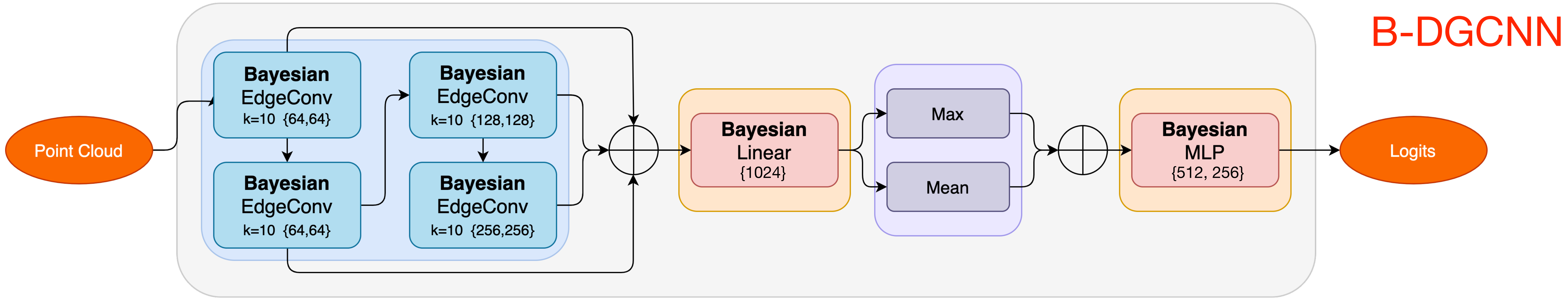
doesn't depend on μ, σ

$w \sim N(\mu, \sigma^2 \mathbf{I})$ reparameterized as: $w = \mu + \sigma\epsilon$ with: $\epsilon \sim N(0,1)$

$$\Rightarrow \nabla_{\mu,\sigma} L(\mu,\sigma) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{S}[\,\nabla_{\mu,\sigma}[\log p(T\,|\,w(\mu,\sigma,\epsilon_j)) + \log p(w(\mu,\sigma,\epsilon_j)]]$$

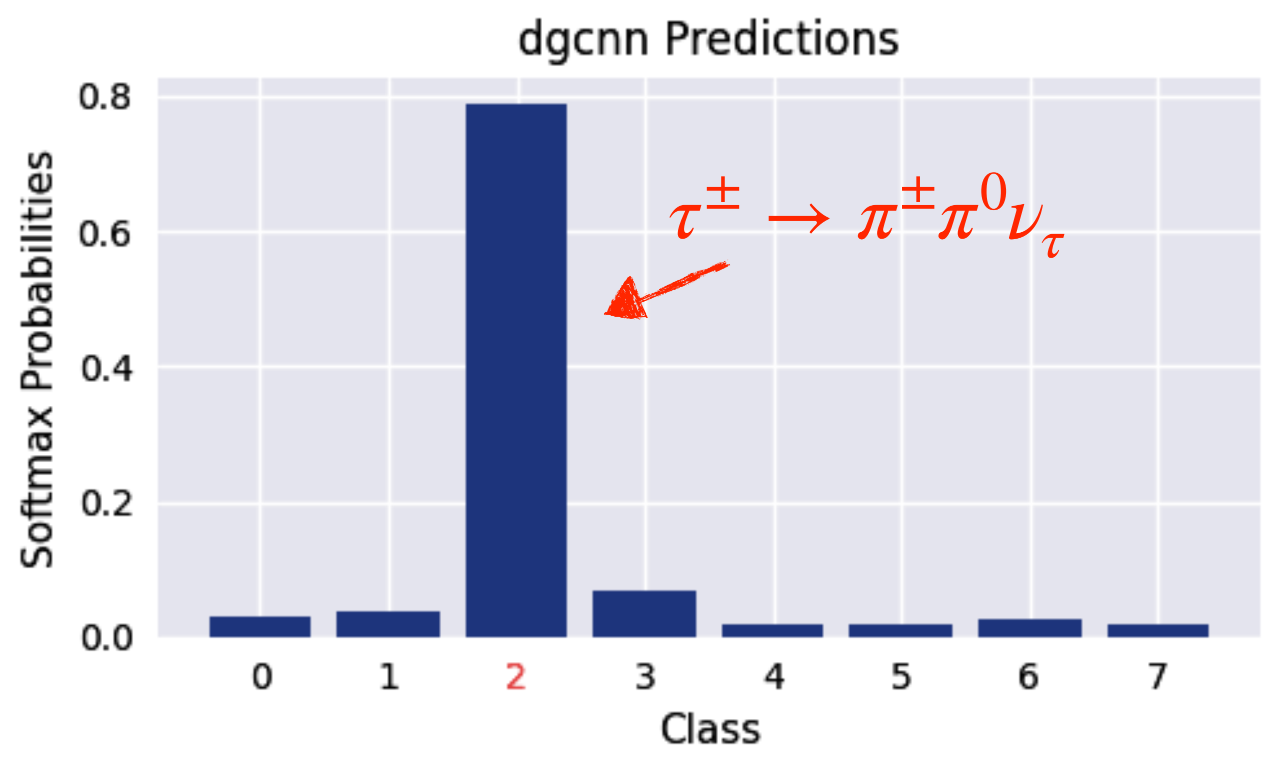
new hyperparameter S: sets the number of samples from $p(\epsilon)$ to estimate the expected log-likelihood (typically S=1 is used)

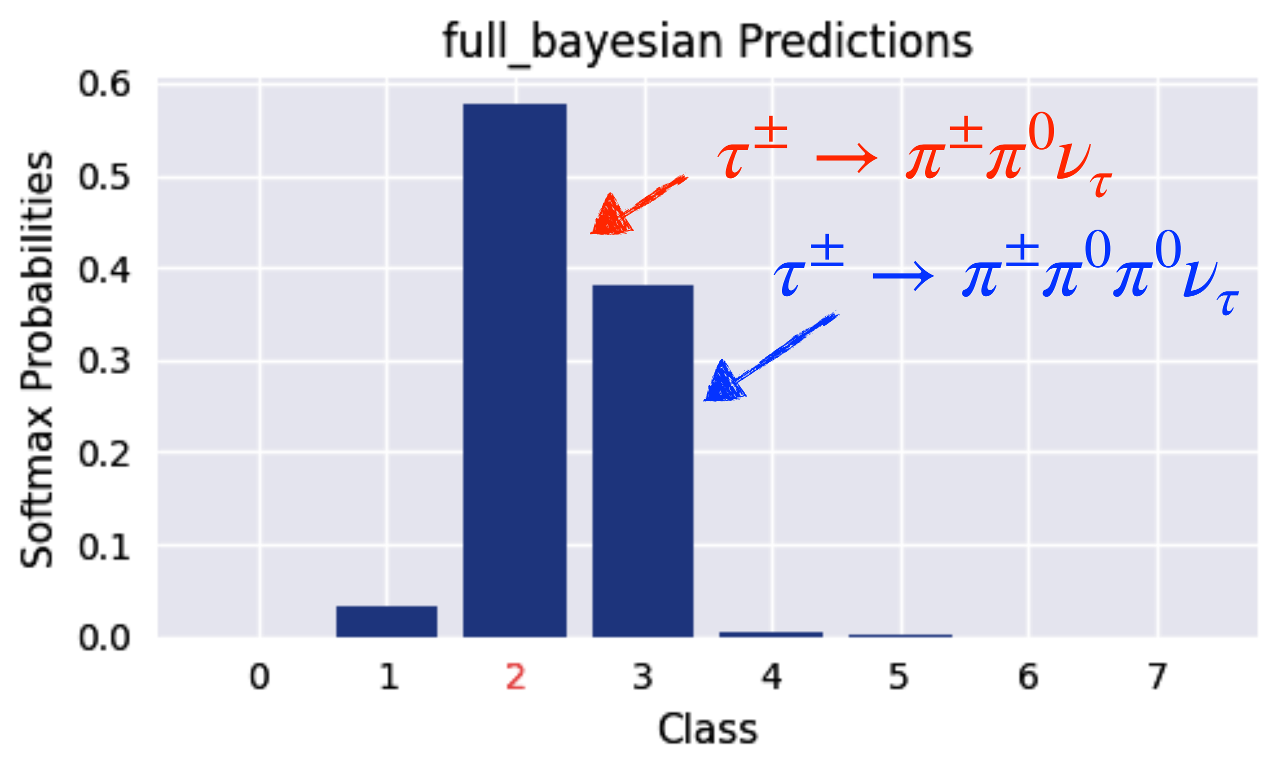
EXAMPLE BAYESIAN-DGNN

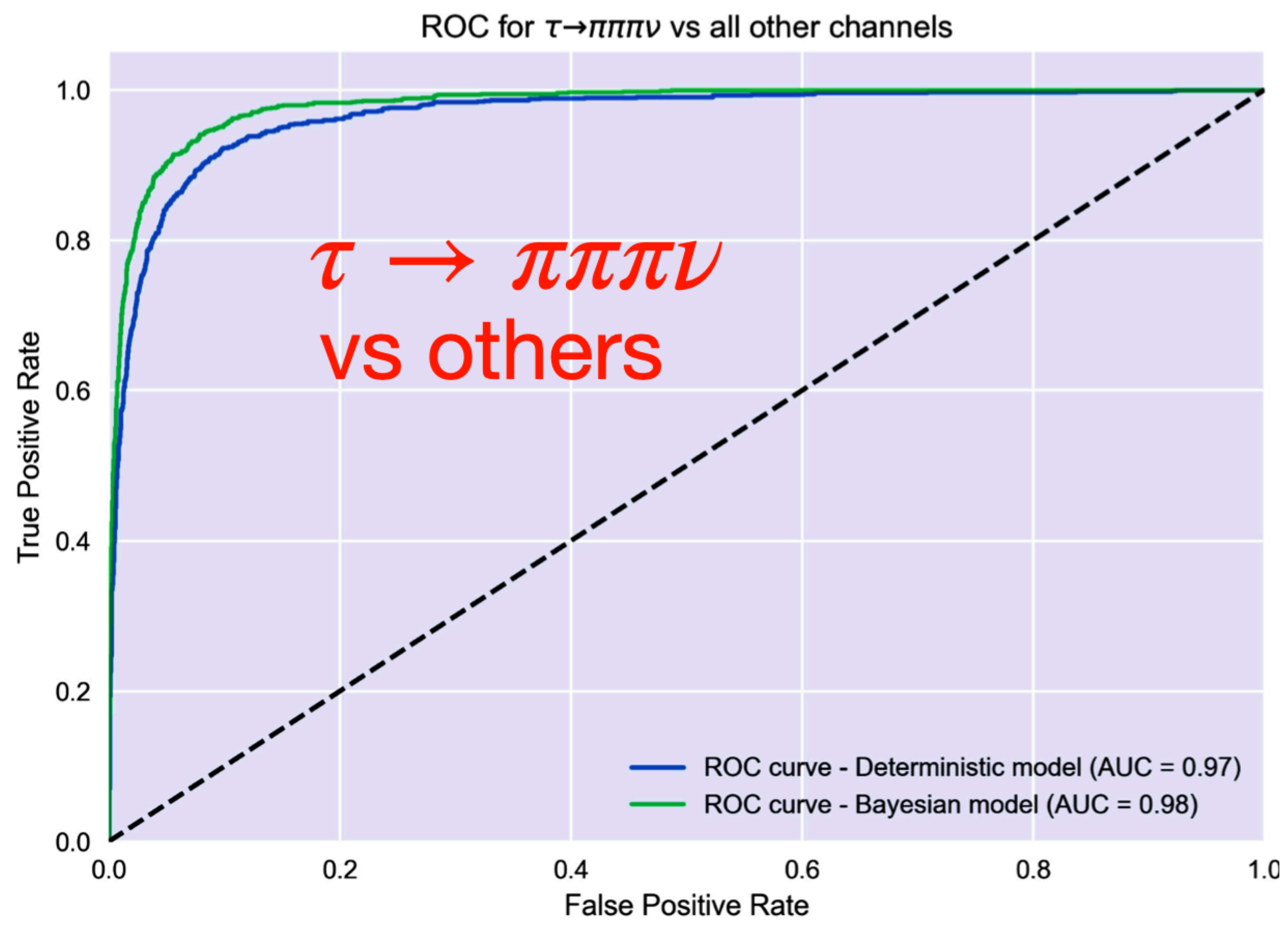


B-DGCNN
Point Cloud
Bayesian EdgeConv k=10 {64,64}
Bayesian EdgeConv k=10 {64,64}
Bayesian EdgeConv k=10 {128,128}
Bayesian EdgeConv k=10 {256,256}
Bayesian Linear {1024}
Max
Mean
Bayesian MLP {512, 256}
Logits

- all bayesian layers (EdgeConv, MLP, etc.), w/ gaussian priors (uncorrelated between layers and neurons)
- better classification performance wrt the point DGCNN
- class probabilities better aligned with physics expectations





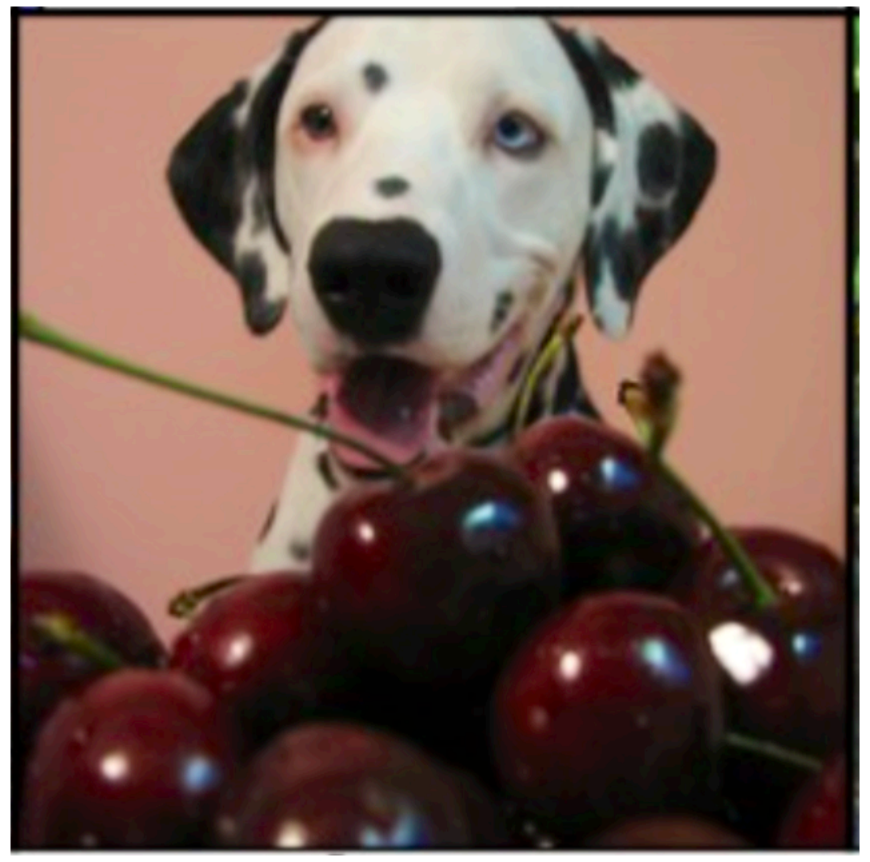


CONFORMAL PREDICTIONS

- conformal prediction is a framework in ML that provides a way to quantify and control the confidence or reliability of predictions made by a model

- in particular, given an input, conformal prediction instead of outputting a single prediction, return a set (a prediction interval in regression problems and a set of classes in classification problems), using a frequentist statistical setting

[leopard]



[dalmatian, grape, elderberry, staffordshire bullterrier, currant]

- the size of the prediction set quantifies the uncertainty of the model, and provides alternatives to the point prediction

A.N.Angelopoulos and S, Bates, arXiv:2107.07511 [cs.LG]

CONFORMAL PREDICTIONS

- advantages of conformal predictions:
  - both the prediction interval and classification sets in conformal prediction are guaranteed to cover the true value with a given confidence level:

$$p(y \in Set(x)) \geq (1 - \alpha)$$

  - very simple to apply, you don't need to re-train the model multiple times (like in ensemble methods) and the model itself doesn't need to have robust uncertainty incorporated (like in bayesian networks). The model is a black box in the conformal prediction approach

CONFORMAL PREDICTIONS

- how it works:

we need: $y = f_w(x)$ a trained model and: $T_C\{(x_i, y_i)\}\ i = 1, \cdots, N$ a calibration set (never seen by the model)

I. define a disagreement score function: $S(x, y)$

ex.
- classification: 1-f(x)$_y$
- regression: MAE

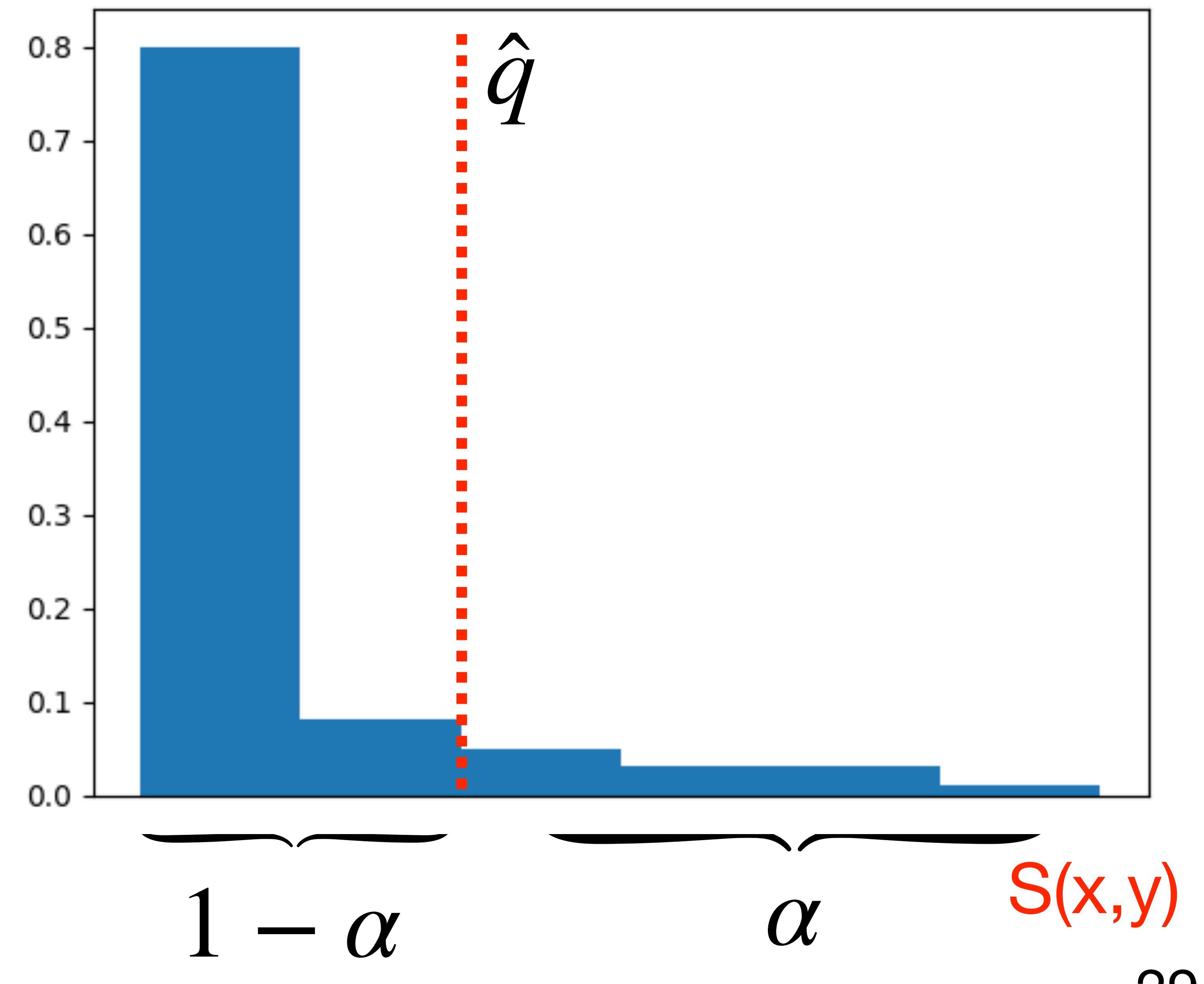
  - eg something for which larger scores means large errors

II. compute the disagreement score on T_C: $\{S(x_1, y_1), \cdots, S(x_N, y_N)\}$

III. compute $\sim (1 - \alpha)$ quantile of scores of the *N* events:

$\hat{q}$: quantile of the scores $\{S(x_1, y_1), \cdots, S(x_N, y_N)\}$

eg: for a new datapoint there would be a $\sim (1 - \alpha)$ chance that it's disagreement score would be less than the threshold $\hat{q}$



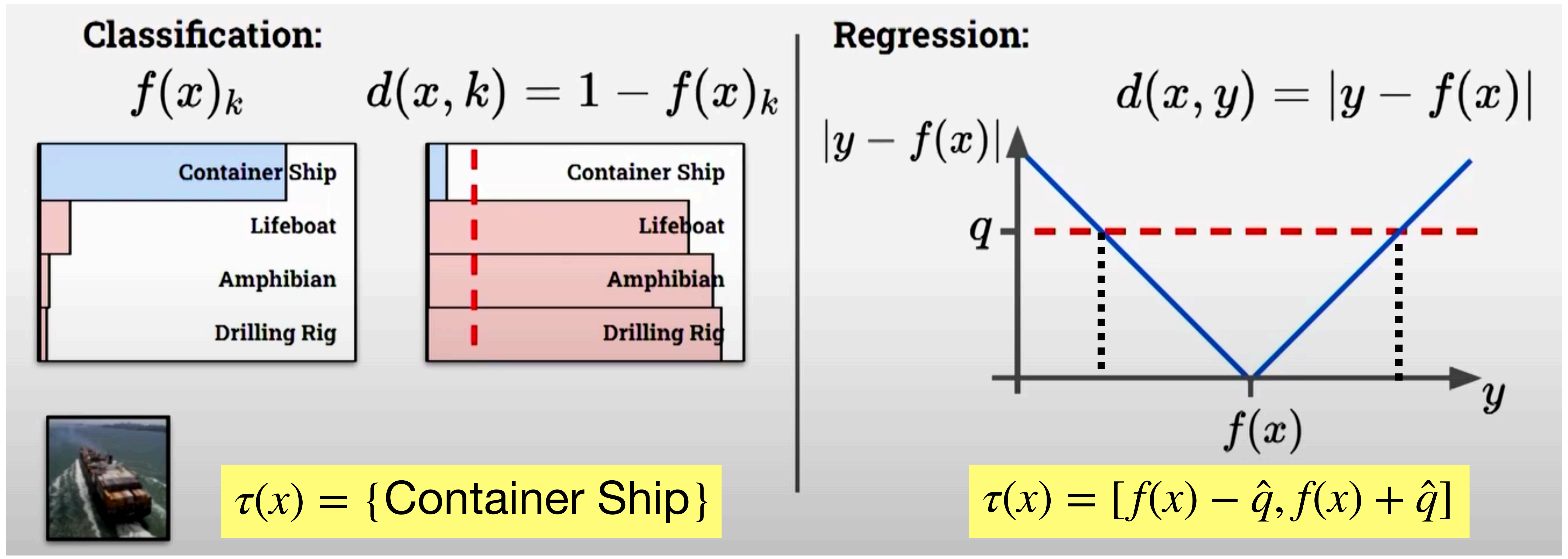
CONFORMAL PREDICTIONS

III. construct prediction sets using model predictions and quantile

- the quantile are used to predict a sets for a new example *x*:

$$\tau(x) = \{y : S(x,y) \leq \hat{q}\}$$

τ: function that return a set which contains a subset of all the possible classes or a confidence interval for a regression task



- possible to demonstrate:

$$(1-\alpha) \leq p(y \in \tau(x)) \leq (1-\alpha) + \frac{1}{N+1}$$

the size of the calibration set has an impact, that can be controlled by choosing N

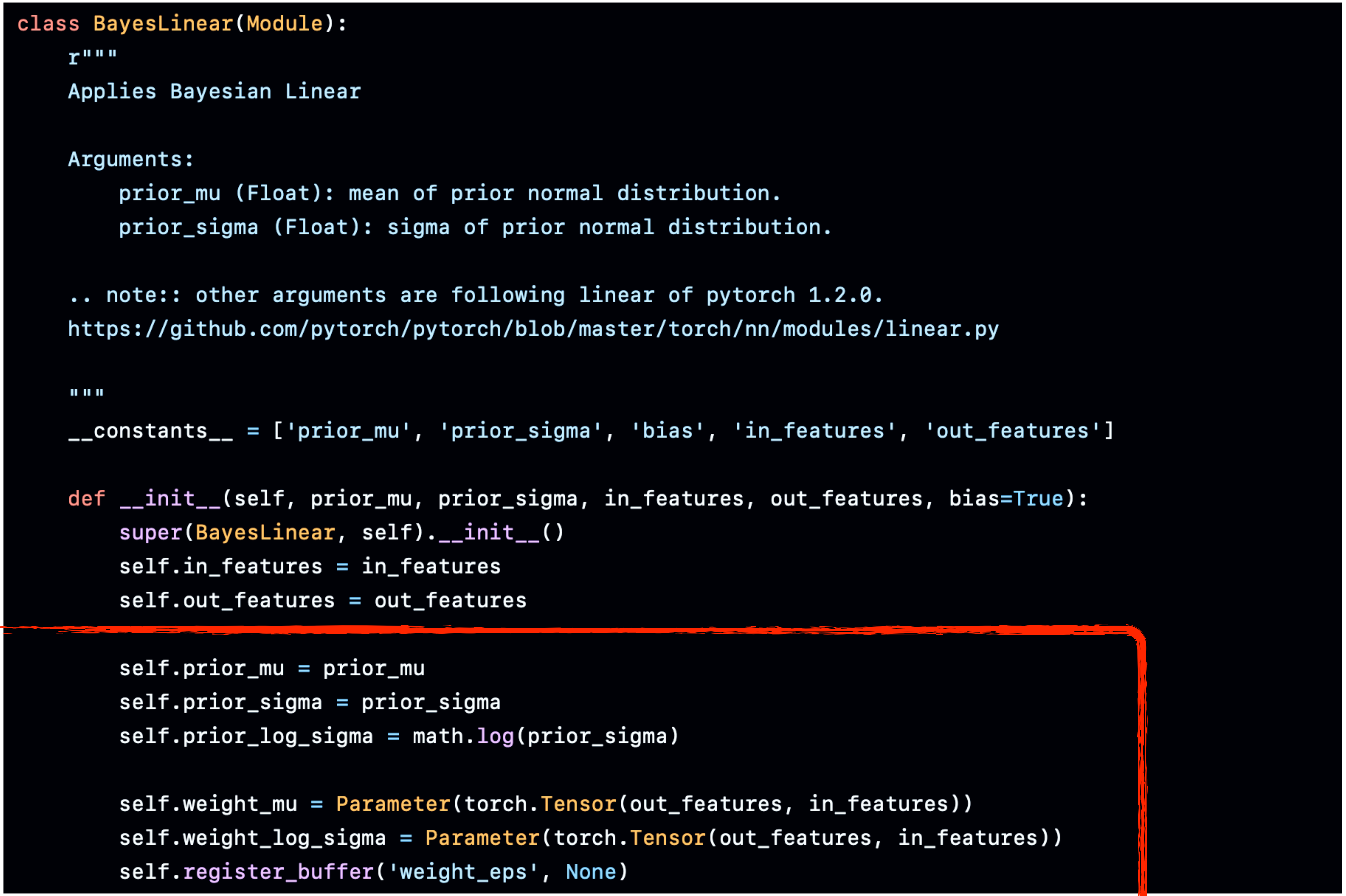
ex: $\alpha = 0.1, N = 100 \rightarrow 90\,\% \leq p \leq 91\,\%$

PYTORCH AWARE BNN LIBS

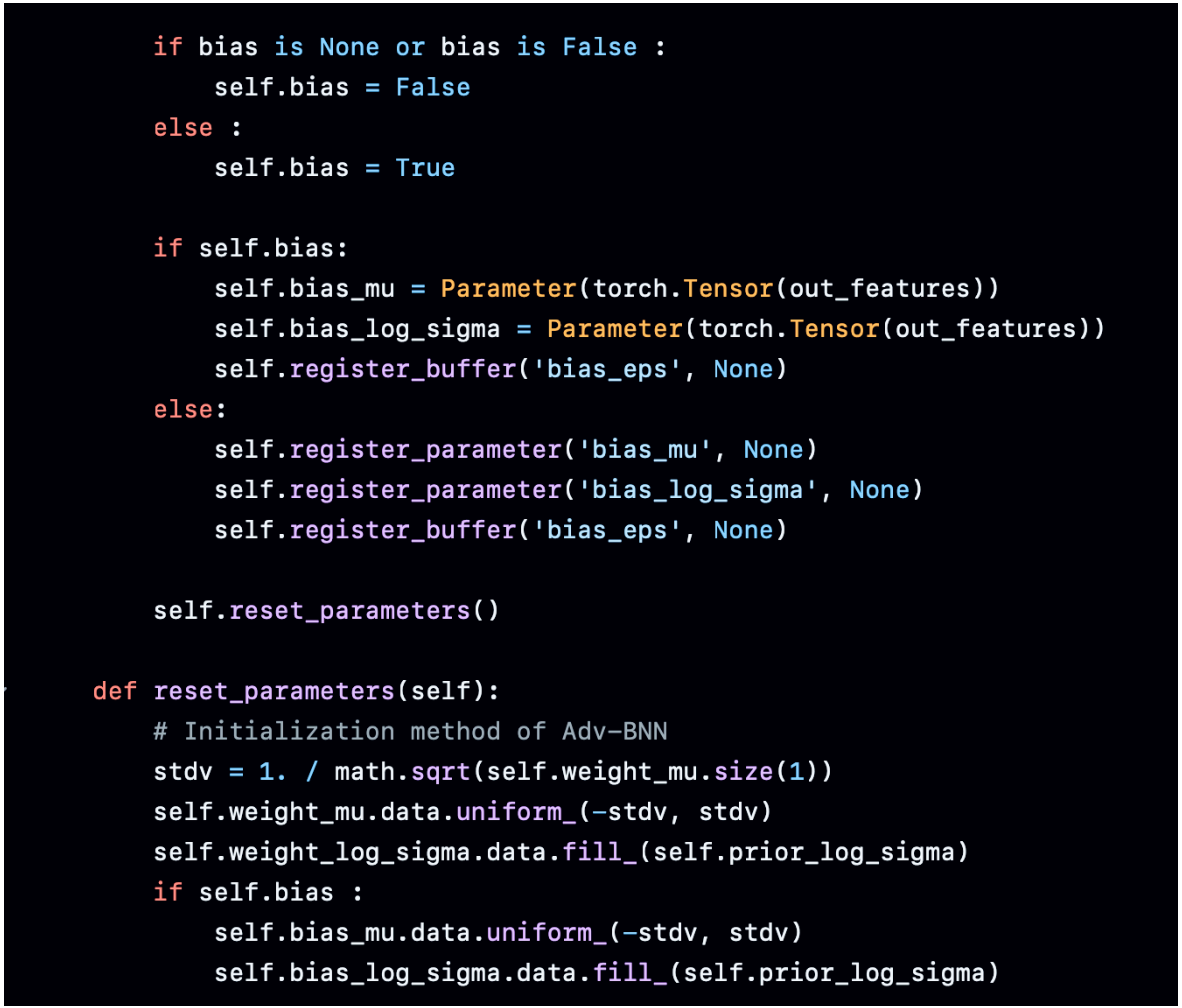
PYTORCH BNN LIBRARIES

- several python libraries to implement BayesianNN with pytorch available on web (with variable support from developers)
- Two extreme examples:
  - **Pyro**: https://github.com/pyro-ppl/pyro: flexible, scalable deep probabilistic programming library built on pytorch
    - 🙂 complete and with a large contributor base
    - 😕 a lot of functionalities means less intuitive use
  - **Torchbnn**: a simpler and limited implementation available in github (old but still functioning): https://github.com/Harry24k/bayesian-neural-network-pytorch/tree/master
    - works in google colab
    - to install it: !pip install torchbnn

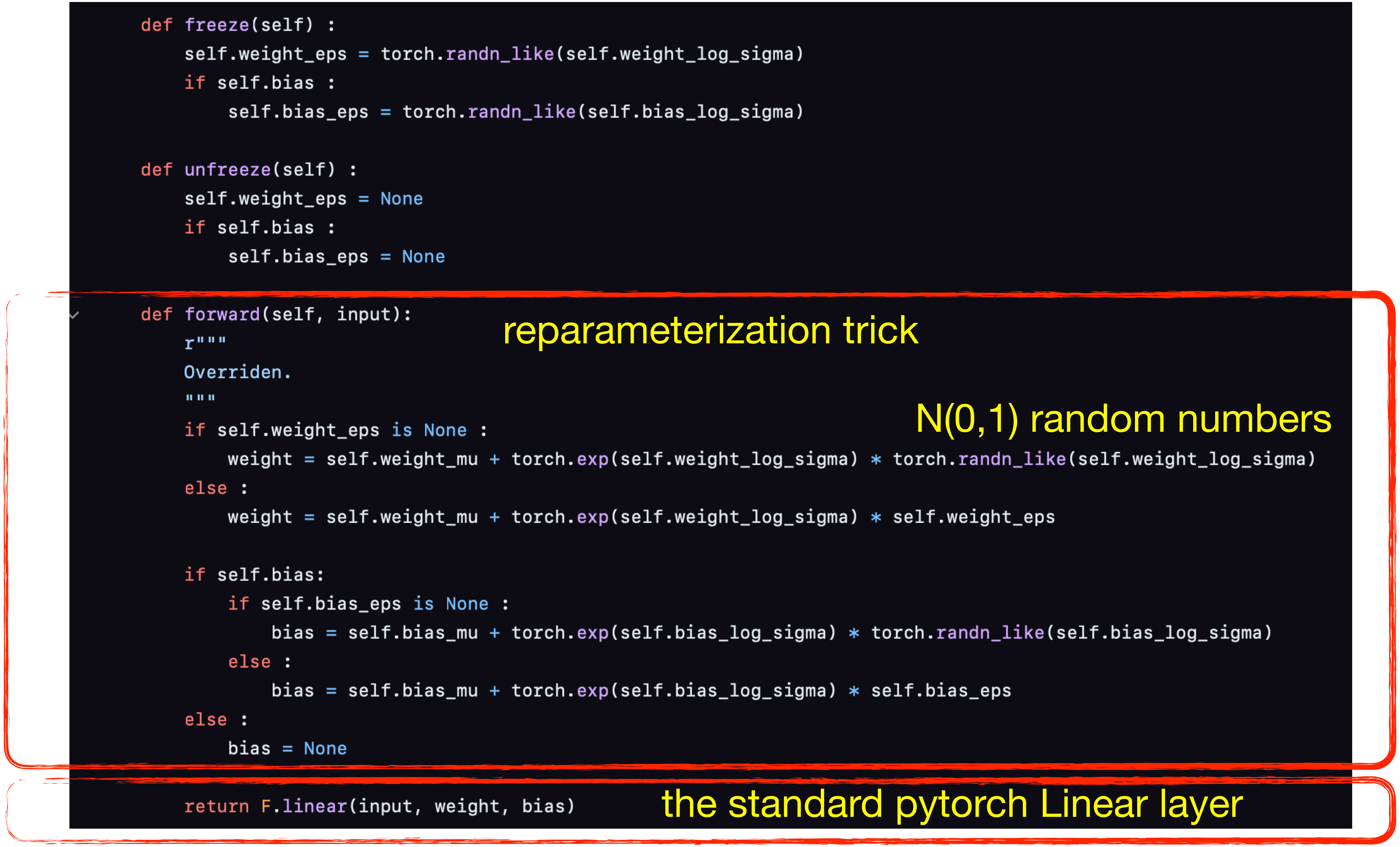
BAYESIAN LINEAR LAYER IN TORCHBNN



```python
class BayesLinear(Module):
    r"""
    Applies Bayesian Linear

    Arguments:
        prior_mu (Float): mean of prior normal distribution.
        prior_sigma (Float): sigma of prior normal distribution.

    .. note:: other arguments are following linear of pytorch 1.2.0.
    https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/linear.py

    """
    __constants__ = ['prior_mu', 'prior_sigma', 'bias', 'in_features', 'out_features']

    def __init__(self, prior_mu, prior_sigma, in_features, out_features, bias=True):
        super(BayesLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features

        self.prior_mu = prior_mu
        self.prior_sigma = prior_sigma
        self.prior_log_sigma = math.log(prior_sigma)

        self.weight_mu = Parameter(torch.Tensor(out_features, in_features))
        self.weight_log_sigma = Parameter(torch.Tensor(out_features, in_features))
        self.register_buffer('weight_eps', None)
```



```
        if bias is None or bias is False :
            self.bias = False
        else :
            self.bias = True

        if self.bias:
            self.bias_mu = Parameter(torch.Tensor(out_features))
            self.bias_log_sigma = Parameter(torch.Tensor(out_features))
            self.register_buffer('bias_eps', None)
        else:
            self.register_parameter('bias_mu', None)
            self.register_parameter('bias_log_sigma', None)
            self.register_buffer('bias_eps', None)

        self.reset_parameters()

    def reset_parameters(self):
        # Initialization method of Adv-BNN
        stdv = 1. / math.sqrt(self.weight_mu.size(1))
        self.weight_mu.data.uniform_(-stdv, stdv)
        self.weight_log_sigma.data.fill_(self.prior_log_sigma)
        if self.bias :
            self.bias_mu.data.uniform_(-stdv, stdv)
            self.bias_log_sigma.data.fill_(self.prior_log_sigma)
```



```python
    def freeze(self) :
        self.weight_eps = torch.randn_like(self.weight_log_sigma)
        if self.bias :
            self.bias_eps = torch.randn_like(self.bias_log_sigma)

    def unfreeze(self) :
        self.weight_eps = None
        if self.bias :
            self.bias_eps = None

    def forward(self, input):
        r"""
        Overriden.
        """
        if self.weight_eps is None :
            weight = self.weight_mu + torch.exp(self.weight_log_sigma) * torch.randn_like(self.weight_log_sigma)
        else :
            weight = self.weight_mu + torch.exp(self.weight_log_sigma) * self.weight_eps

        if self.bias:
            if self.bias_eps is None :
                bias = self.bias_mu + torch.exp(self.bias_log_sigma) * torch.randn_like(self.bias_log_sigma)
            else :
                bias = self.bias_mu + torch.exp(self.bias_log_sigma) * self.bias_eps
        else :
            bias = None

        return F.linear(input, weight, bias)
```

reparameterization trick

N(0,1) random numbers

the standard pytorch Linear layer

EXAMPLE BAYESIAN MLP

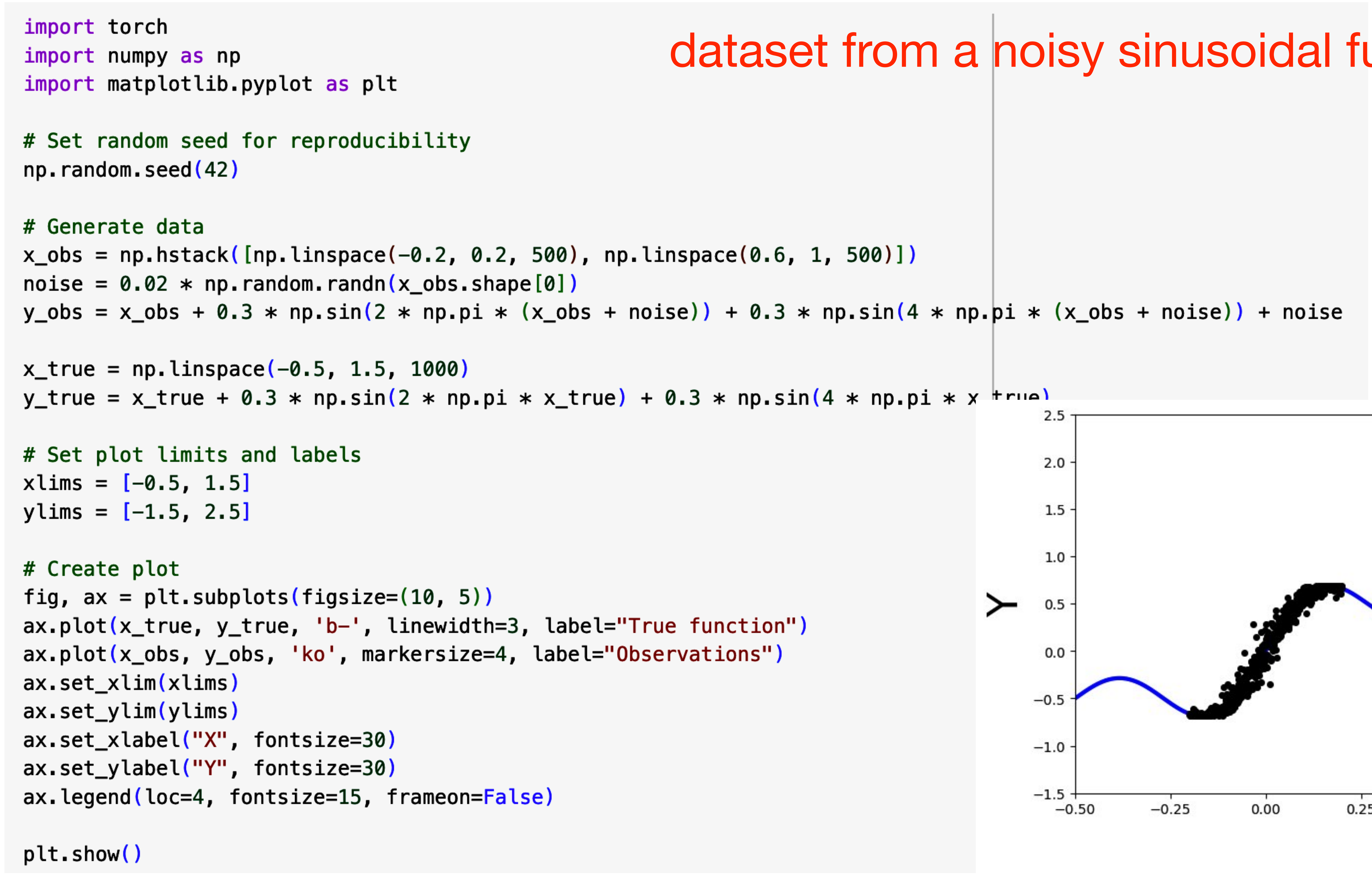
```python
model = nn.Sequential(
    bnn.BayesLinear(prior_mu=0, prior_sigma=1.5, in_features=1, out_features=20),
    nn.ReLU(),
    bnn.BayesLinear(prior_mu=0, prior_sigma=0.1, in_features=20, out_features=20),
    nn.ReLU(),
    bnn.BayesLinear(prior_mu=0, prior_sigma=0.5, in_features=20, out_features=1),
)
```

```python
# loss is the sum of a recontruction loss (MSE) + kl_weight*KL divergence

mse_loss = nn.MSELoss()
kl_loss = bnn.BKLLoss(reduction='mean', last_layer_only=False)

# weight of the KL term in the total loss
kl_weight = 1.0

# adam optimiser
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

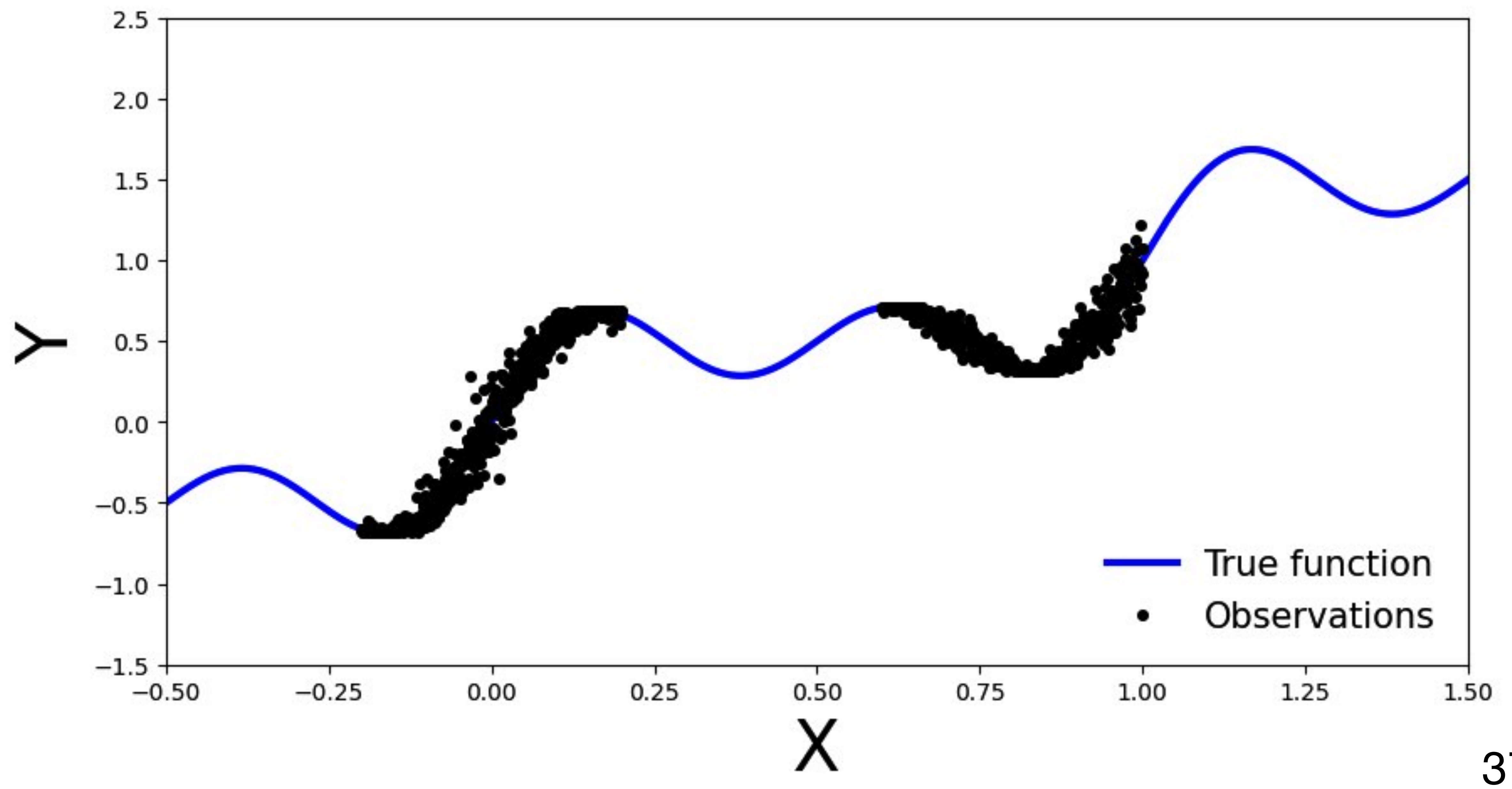
```python
# training loop

training_steps = 1500

for step in range(training_steps):
    pre = model(x)
    mse = mse_loss(pre, y)
    kl = kl_loss(model)
    cost = mse + kl_weight*kl

    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

print('- MSE : %2.2f, KL : %2.2f' % (mse.item(), kl.item()))
```

EXAMPLE OF A BNN WITH MCMC IN PYRO

dataset from a noisy sinusoidal function



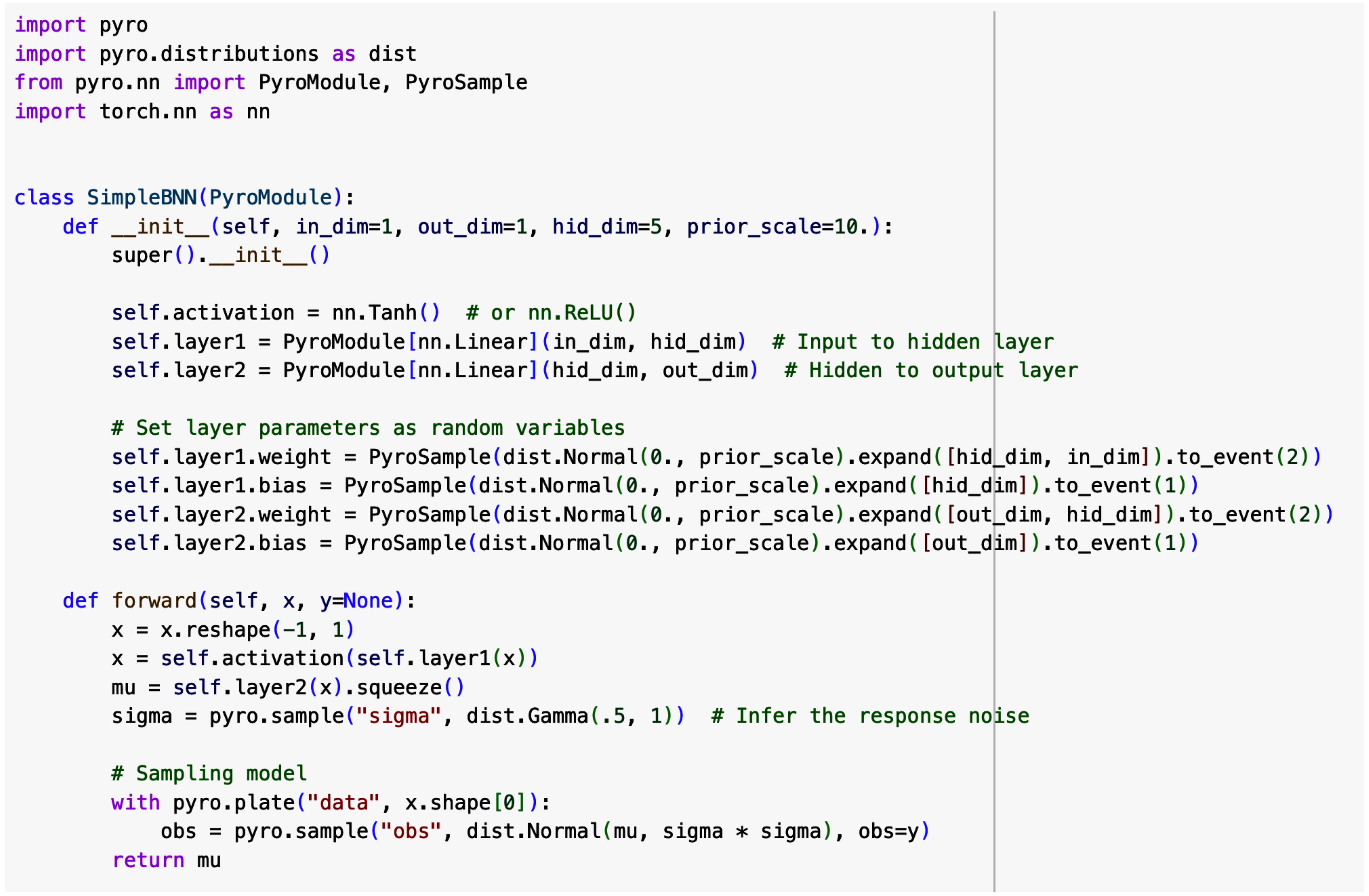
```
import torch
import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# Generate data
x_obs = np.hstack([np.linspace(-0.2, 0.2, 500), np.linspace(0.6, 1, 500)])
noise = 0.02 * np.random.randn(x_obs.shape[0])
y_obs = x_obs + 0.3 * np.sin(2 * np.pi * (x_obs + noise)) + 0.3 * np.sin(4 * np.pi * (x_obs + noise)) + noise

x_true = np.linspace(-0.5, 1.5, 1000)
y_true = x_true + 0.3 * np.sin(2 * np.pi * x_true) + 0.3 * np.sin(4 * np.pi * x_true)

# Set plot limits and labels
xlims = [-0.5, 1.5]
ylims = [-1.5, 2.5]

# Create plot
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(x_true, y_true, 'b-', linewidth=3, label="True function")
ax.plot(x_obs, y_obs, 'ko', markersize=4, label="Observations")
ax.set_xlim(xlims)
ax.set_ylim(ylims)
ax.set_xlabel("X", fontsize=30)
ax.set_ylabel("Y", fontsize=30)
ax.legend(loc=4, fontsize=15, frameon=False)

plt.show()
```



shallow BNN with gaussian priors on the weights

$$p(w) = N(0,10 \cdot \mathbf{1})$$

$$p(y_i \,|\, x_i, w) = N(ANN_w(x_i), \sigma^2)$$ with prior for sigma: $\sigma \sim \Gamma(0.5,1)$



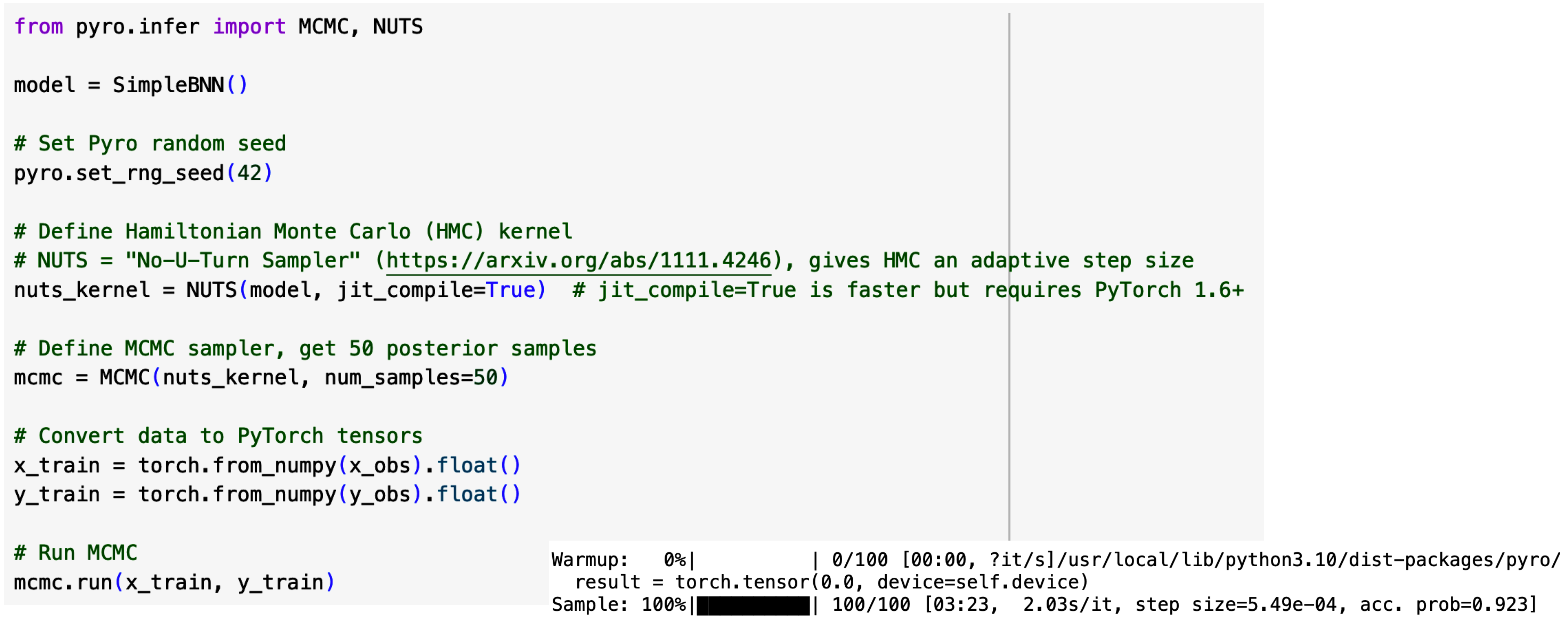
```python
import pyro
import pyro.distributions as dist
from pyro.nn import PyroModule, PyroSample
import torch.nn as nn


class SimpleBNN(PyroModule):
    def __init__(self, in_dim=1, out_dim=1, hid_dim=5, prior_scale=10.):
        super().__init__()

        self.activation = nn.Tanh()  # or nn.ReLU()
        self.layer1 = PyroModule[nn.Linear](in_dim, hid_dim)  # Input to hidden layer
        self.layer2 = PyroModule[nn.Linear](hid_dim, out_dim)  # Hidden to output layer

        # Set layer parameters as random variables
        self.layer1.weight = PyroSample(dist.Normal(0., prior_scale).expand([hid_dim, in_dim]).to_event(2))
        self.layer1.bias = PyroSample(dist.Normal(0., prior_scale).expand([hid_dim]).to_event(1))
        self.layer2.weight = PyroSample(dist.Normal(0., prior_scale).expand([out_dim, hid_dim]).to_event(2))
        self.layer2.bias = PyroSample(dist.Normal(0., prior_scale).expand([out_dim]).to_event(1))

    def forward(self, x, y=None):
        x = x.reshape(-1, 1)
        x = self.activation(self.layer1(x))
        mu = self.layer2(x).squeeze()
        sigma = pyro.sample("sigma", dist.Gamma(.5, 1))  # Infer the response noise

        # Sampling model
        with pyro.plate("data", x.shape[0]):
            obs = pyro.sample("obs", dist.Normal(mu, sigma * sigma), obs=y)
        return mu
```

define and run Markov Chain Monte Carlo sampler

e.g. $E_{w\sim p(w|T))}[p(y\,|\,x,w)]$ is approximated via: $E_{w\sim p(w|T))}[p(y\,|\,x,w)] \approx \frac{1}{N}\sum_{i=1}^{N} p(y\,|\,x,w_i)$

$w_i \sim p(w_i\,|\,T) \propto p(T\,|\,w)p(w)$

as the normalisation (evidence $p(T)$) is intractable MCMC methods (like Hamiltonian MC) are used to draw sample from the non-normalized posterior (M.D.Offman, A.Gelman, arXiv:1111.4246)



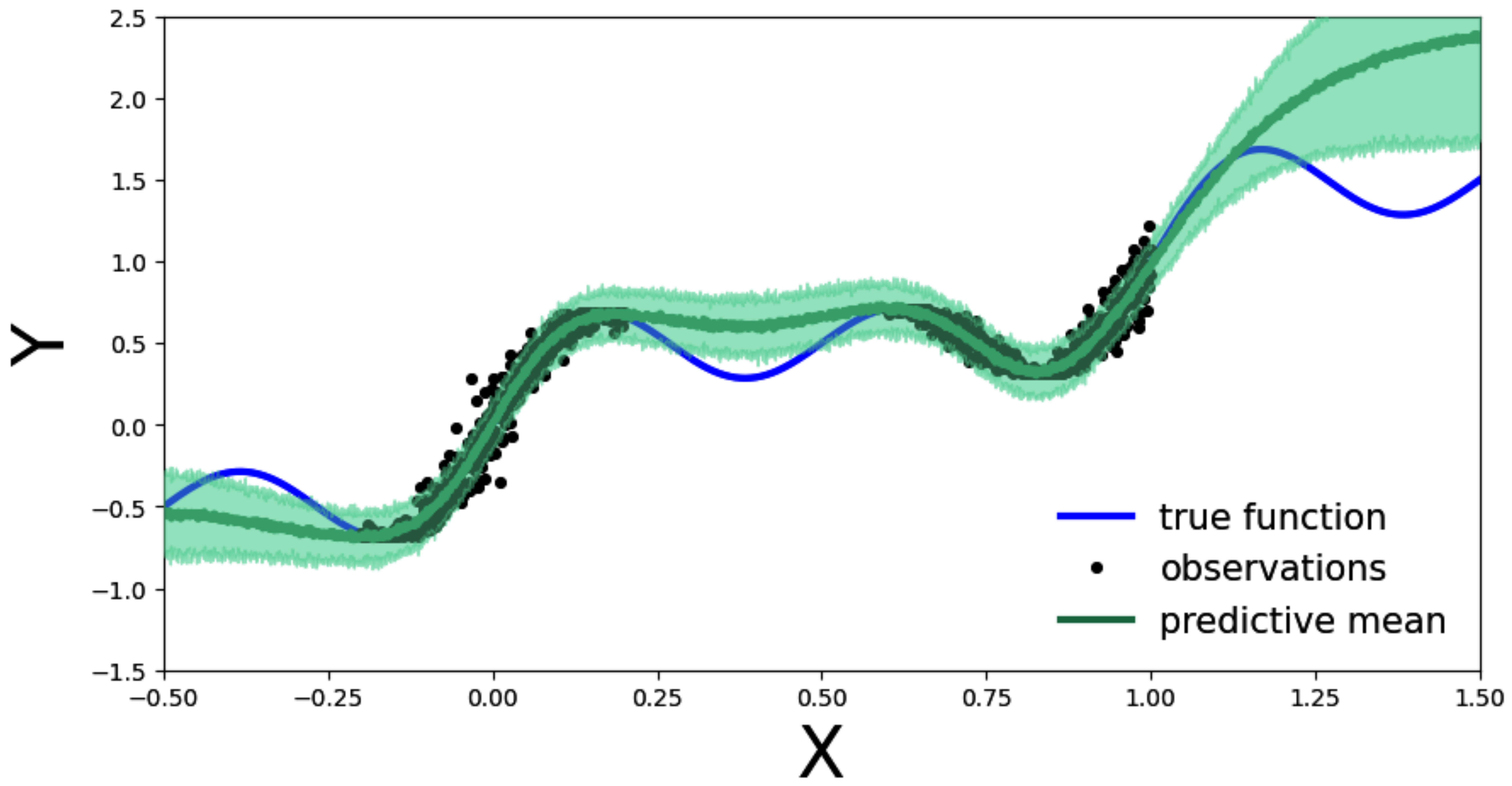
```
from pyro.infer import MCMC, NUTS

model = SimpleBNN()

# Set Pyro random seed
pyro.set_rng_seed(42)

# Define Hamiltonian Monte Carlo (HMC) kernel
# NUTS = "No-U-Turn Sampler" (https://arxiv.org/abs/1111.4246), gives HMC an adaptive step size
nuts_kernel = NUTS(model, jit_compile=True)  # jit_compile=True is faster but requires PyTorch 1.6+

# Define MCMC sampler, get 50 posterior samples
mcmc = MCMC(nuts_kernel, num_samples=50)

# Convert data to PyTorch tensors
x_train = torch.from_numpy(x_obs).float()
y_train = torch.from_numpy(y_obs).float()

# Run MCMC
mcmc.run(x_train, y_train)
```

```
Warmup:   0%|          | 0/100 [00:00, ?it/s]/usr/local/lib/python3.10/dist-packages/pyro/
  result = torch.tensor(0.0, device=self.device)
Sample: 100%|██████████| 100/100 [03:23,  2.03s/it, step size=5.49e-04, acc. prob=0.923]
```

calculate and plot the predictive distribution

```
from pyro.infer import Predictive

predictive = Predictive(model=model, posterior_samples=mcmc.get_samples())
x_test = torch.linspace(xlims[0], xlims[1], 3000)
preds = predictive(x_test)
```

```
def plot_predictions(preds):
    y_pred = preds['obs'].T.detach().numpy().mean(axis=1)
    y_std = preds['obs'].T.detach().numpy().std(axis=1)

    fig, ax = plt.subplots(figsize=(10, 5))
    xlims = [-0.5, 1.5]
    ylims = [-1.5, 2.5]
    plt.xlim(xlims)
    plt.ylim(ylims)
    plt.xlabel("X", fontsize=30)
    plt.ylabel("Y", fontsize=30)

    ax.plot(x_true, y_true, 'b-', linewidth=3, label="true function")
    ax.plot(x_obs, y_obs, 'ko', markersize=4, label="observations")
    ax.plot(x_obs, y_obs, 'ko', markersize=3)
    ax.plot(x_test, y_pred, '-', linewidth=3, color="#408765", label="predictive mean")
    ax.fill_between(x_test, y_pred - 2 * y_std, y_pred + 2 * y_std, alpha=0.6, color='#86cfac', zorder=5)

    plt.legend(loc=4, fontsize=15, frameon=False)


plot_predictions(preds)
```


Y
X
true function
observations
predictive mean