pacbb: PAC-Bayes Bounds Evaluation Framework

Yauheni Mardan, Maksym Tretiakov, Alexander Immer, Vincent Fortuin

Abstract

Evaluating PAC-Bayes bounds often requires extensive boilerplate code. We present a modular framework that streamlines the probabilistic model training and bound evaluation, with key components such as **Variables**, **Distributions**, **Objectives**, **Bounds**, **Metrics**, and **Losses**. This design enables easy customization, allowing researchers to focus on new methods rather than routine implementation. We demonstrate its flexibility on a ConvNet trained on CIFAR10.

1 PAC-Bayes Theory and Foundations

1.1 Introduction

In a **supervised learning** setting, we have a dataset $\mathcal{S} = \{(X_i, Y_i)\}_{i=1}^n$ of input-output pairs drawn i.i.d. from an unknown distribution P. Our goal is to learn a parametrized function f_θ, $\theta \in \Theta$ (for instance, a neural network with its weights as parameters θ) that predicts Y given X, where Θ is a parameter set. To measure the prediction quality, a loss function $l(\hat{Y}, Y)$ determines how far a prediction $\hat{Y}$ is from the true label Y. Since the distribution P is unknown, we instead minimize the **empirical risk** $r(\theta)$ which is used as a practical proxy of the **generalization risk** $R(\theta)$:

$$r(\theta) \;=\; \frac{1}{n}\sum_{i=1}^{n} l\big(f_\theta(X_i),\, Y_i\big), \tag{1}$$

$$R(\theta) \;=\; \mathbb{E}_{(X,Y)\sim P}\big[l(f_\theta(X),\, Y)\big]. \tag{2}$$

In practice, it is not possible to calculate $R(\theta)$ because it requires an inaccessible distribution P. That is why $r(\theta)$ is usually minimized and then used to upper-bound $R(\theta)$.

1.2 Key Ideas

The PAC-Bayes framework provides a convenient way of upper-bounding the **generalization risk** of a randomized estimator defined by a parametrized function $f(\theta)$. Specifically, it relates the difference between the **generalization risk** of the randomized estimator $\mathbb{E}_{\theta\sim\rho}[R(\theta)]$ and its **empirical risk** $\mathbb{E}_{\theta\sim\rho}[r(\theta)]$ to a **divergence** term quantifying how much the **posterior** ρ departs from a **prior** distribution π. Formally, with high probability over the choice of the dataset $\mathcal{S}$, this difference remains bounded if ρ does not deviate much from π. Note that ρ and π are not Bayesian distributions, so they are not necessarily connected by the likelihood, and, in general, they can be any distributions, as long as π does not depend on the data.

In essence, the PAC-Bayes approach **penalizes** a **posterior** that is too far from the prior, thereby creating a balance between fitting the data well and adhering to the **prior** structure. This trade-off prevents overfitting.

A generic example of a PAC-Bayes bound can be written as:

$$\mathbb{E}_{\theta\sim\rho}[R(\theta)] \;\;\leq\;\; \mathbb{E}_{\theta\sim\rho}[r(\theta)] \;+\; \Phi\big(\mathrm{D}(\rho\|\pi),\, n,\, \epsilon\big), \tag{3}$$

where n is the dataset size, ϵ is a confidence parameter, and Φ is some function, which combines **divergence** D, dataset size, and confidence parameter. However, in general, this inequality is not explicit, and the generalization risk, empirical risk, and other components can be combined implicitly.

1.3 Classical PAC-Bayes Bounds

One of the classical PAC-Bayes bounds is due to McAllester (McAllester, 1999). It exemplifies the typical trade-off structure.

For any $\epsilon > 0$, with probability at least $1 - \epsilon$ over the choice of the dataset $\mathcal{S}$, we have for all ρ:

$$\mathbb{E}_{\theta\sim\rho}[R(\theta)] \leq \mathbb{E}_{\theta\sim\rho}[r(\theta)] + \sqrt{\frac{\mathrm{KL}(\rho\|\pi) + \log\left(\frac{1}{\epsilon}\right) + \frac{5}{2}\log(n) + 8}{2n-1}}.$$

In words, the difference $\mathbb{E}_{\theta\sim\rho}[R(\theta)] - \mathbb{E}_{\theta\sim\rho}[r(\theta)]$ is upper-bounded by a term involving the $\mathrm{KL}(\rho\|\pi)$ divergence, regularized by $\log(1/\epsilon)$ and the dataset size n. This expresses the usual trade-off: if ρ deviates substantially from the prior π, the bound penalizes that complexity.

Another important result for classification is the so-called **PAC-Bayes-kl** bound (Langford and Seeger, 2001), often called "KL bound":

$$\mathrm{kl}\big(\mathbb{E}_{\theta\sim\rho}[R(\theta)],\, \mathbb{E}_{\theta\sim\rho}[r(\theta)]\big) \;\leq\; \frac{\mathrm{KL}(\rho\|\pi) \;+\; \log\left(\frac{2\sqrt{n}}{\epsilon}\right)}{n}$$

where $\mathrm{kl}(\cdot,\cdot)$ is the KL divergence between two Bernoulli parameters. By inverting the kl, the **expected risk** $\mathbb{E}_{\theta\sim\rho}[R(\theta)]$ can be upper-bounded via an explicit function of $\mathrm{KL}(\rho\|\pi)$ and $\mathbb{E}_{\theta\sim\rho}[r(\theta)]$ (Pérez-Ortiz et al., 2021).

Most classical PAC-Bayes bounds (e.g., McAllester, 1999; Langford and Seeger, 2001; Pérez-Ortiz et al., 2021) employ $\mathrm{KL}(\rho\|\pi)$ to measure the discrepancy between **posterior** and **prior**. The KL divergence (Kullback and Leibler, 1951) is a convenient choice because it often admits closed-form solutions for common parametric distributions (e.g., Gaussians). However, there are PAC-Bayes bounds with alternative divergences like χ^2 (Rodríguez-Gálvez et al., 2024a,b) or other f-divergences (Kuzborskij et al., 2024). While these non-KL bounds can be tighter in theory, they are often more cumbersome to optimize in practice.

1.4 Theoretical requirements for practical applications

Although the PAC-Bayes framework is highly flexible—for example, one is free to use almost any divergence D—some theoretical limitations are important to consider.

1.4.1 Bounded Losses and Extensions

A standard requirement in classical PAC-Bayes is that the loss $l(\cdot,\cdot)$ is bounded in $[0, C]$. An essential example here is the 0-1 loss for **classification**. However, as the 0-1 loss is not differentiable, it is usually replaced with a differentiable, bounded surrogate when one wants to optimize a PAC-Bayes bound. In classification, Dziugaite and Roy (2017) suggest bounding the negative log-likelihood by a constant $\log(1/p_{\min})$, thereby ensuring a $[0, 1]$ range after rescaling. On the other hand, if the bound is not supposed to be optimized, but only should be calculated, the 0-1 loss is sufficient.

For a **regression** task, losses are usually unbounded. Here sub-Gaussian or sub-Gamma assumptions (Germain et al., 2017), or more general bounded-second-moment assumptions (Rodríguez-Gálvez et al., 2024a), are often used for PAC-Bayes bounds calculations.

1.4.2 Prior Choices and Data-Dependence

Selecting a **prior** distribution π is another crucial design choice. While the theory is simplest if π is chosen independently of the dataset $\mathcal{S}$, one typically seeks a 'good' **prior** that is closer to the eventual **posterior**, thus reducing divergence costs. Various **data-dependent** priors have been explored to enhance tightness:

- **Differentially private priors** (Dziugaite and Roy, 2019) aim to keep data-dependence controlled, so the **prior** can be effectively seen as 'almost' independent.
- **Splitting the dataset** (Perez-Ortiz et al., 2021; Dziugaite et al., 2020): one part of the data is used to inform the **prior**; the remaining part is used for the bound. This avoids direct correlation with the test portion.

- **Sequential prior updates** (Wu et al., 2024) allow re-using data repeatedly, though the complexity of these methods can grow.

1.5 PAC-Bayes for Probabilistic Neural Networks

This work focuses on developing a framework for efficiently evaluating PAC-Bayes bounds for probabilistic neural networks, a special class of randomized estimators. Early attempts at applying PAC-Bayes bounds to large neural networks often resulted in vacuous bounds. A breakthrough was the work by Dziugaite and Roy (2017), where they optimized a relaxed version of the PAC-Bayes-kl bound and achieved improved (non-vacuous) certificates.

In standard (deterministic) deep models, the parameter vector θ is fixed after training. PAC-Bayes uses a **distribution** $\rho(\theta)$ over the parameters. A typical parameterization is Gaussian with diagonal covariance:

$$\rho(\theta) \;=\; \mathcal{N}(w,\, \mathrm{diag}(s)),$$

where w and s are learned via gradient-based methods (Blundell et al., 2015; Pérez-Ortiz et al., 2021).

A common simple choice is a Gaussian prior (e.g., zero mean, fixed covariance). Despite its simplicity, it already enables competitive non-vacuous bounds (Dziugaite and Roy, 2017).

2 Introducing the PAC-Bayes Evaluation Framework

Our PAC-Bayes bound framework, `pacbb`, offers a powerful and flexible methodology for risk certification, enabling quantifiable generalization bounds for complex models such as deep networks. Many works on PAC-Bayes provide not only theoretical but also practical results. Nevertheless, to replicate these results in practice—for instance, to evaluate a newly developed bound—one often needs to write a substantial amount of repetitive or "boilerplate" code. This includes training a probabilistic network, performing Monte Carlo sampling from the posterior distribution, computing bounds, and more.

The main goal of the `pacbb` framework we developed is to mitigate the redundant effort associated with repeatedly writing the same code. During the development of the framework, we made it modular and extensible (Fig. 1). At the same time, we tried to keep it simple, so that one would need minimal effort to implement and test a new bound or training process.

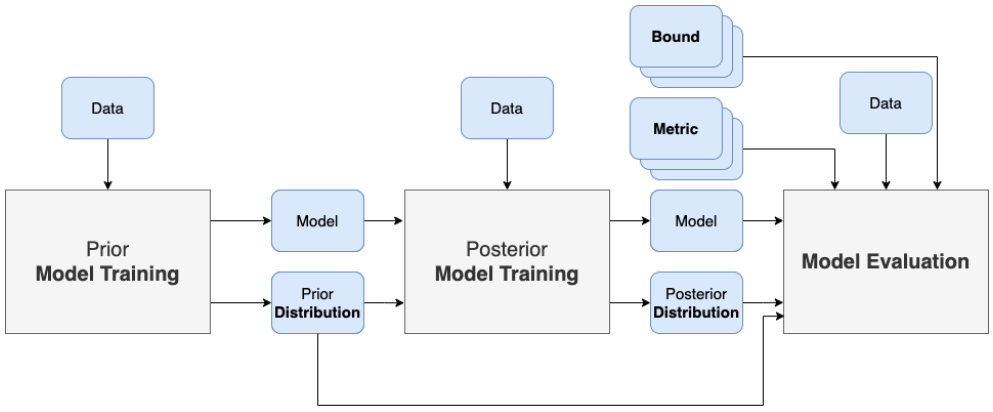

Figure 1: Generic pipeline for probabilistic model training and subsequent bounds/metrics evaluation. Gray boxes represent the main steps of the pipeline; blue boxes represent intermediate artifacts, data inputs, and final outputs. Each main step of the pipeline is detailed in the following sections: Prior **Model Training** in subsection 2.2, Posterior **Model Training** in subsection 2.3, and **Model Evaluation** in subsection 2.4.

In this paper, we present and explain the key concepts of the developed framework. It is worth noting that while other code repositories do exist for implementing and evaluating PAC-Bayes bounds, many of these are designed solely for specific experiments and offer limited extensibility. For example,

they often require specialized or redundant code for each new bound, are not modular in structure, or do not integrate well with standard deep-learning libraries. Consequently, they are difficult to adapt to novel methods, larger datasets, or alternative neural architectures. We build our framework on the code from Pérez-Ortiz et al. (2021), which already has a clear separation of key components, and extend it to address these limitations more systematically.

Capabilities and Limitations. This framework was developed for the PAC-Bayes bounds evaluation. Hence, it is suitable for the probabilistic model training, evaluation, and comparison. We do not consider other models beyond neural networks (NN). However, it is possible to extend the framework to other parametric models if they also use **PyTorch** (Paszke et al., 2019).

2.1 Key components and Framework design

Our framework operates with the six main components present in nearly every PAC-Bayes setting. These components are:

- **Variables** and **Distributions** for handling layer distributions,
- **NN model to ProbNN model conversion** for transforming standard neural networks into probabilistic models,
- **Objectives** for training probabilistic models,
- **Bounds** for representing PAC-Bayes generalization guarantees,
- **Metrics** for evaluating model performance,
- **Losses** for training and bound estimation.

In Fig. 1, the generic training and evaluation pipeline is depicted. This pipeline first trains a probabilistic NN (ProbNN) and then evaluates its PAC bounds. The pipeline can be modified based on user needs, and the user is free to replace any of the steps with their implementation.

As shown in Figure 1, the generic training and evaluation pipeline consists of two main stages: training a probabilistic neural network (ProbNN) and evaluating its PAC-Bayes bounds. The framework allows flexibility, enabling users to modify the pipeline as needed and replace any step with their implementation.

The framework is built on **PyTorch** and mainly follows an object-oriented approach. Key components such as **Variables**, **Distributions**, **Bounds**, and **Metrics** are implemented as objects, which are then managed by generic training and bound evaluation functions. Together, these components form an end-to-end pipeline for model training and bounds evaluation. Alternatively, individual components of the framework can be used independently, for instance, to convert a neural network into a probabilistic model without running the full pipeline. It is also possible to completely replace a part of the pipeline with another implementation, such as replacing learned priors with the initialization of a non-informative prior.

To simplify the conversion process from a standard neural network to a probabilistic one, the framework replaces deterministic **PyTorch** layers with probabilistic layers from our implementation. These probabilistic layers are still based on **PyTorch** but incorporate distributions for weights and biases, along with reference distributions. As a result, each probabilistic layer encapsulates all necessary information for weight and bias sampling, inference, and divergence calculation.

The weight and bias distributions are represented as **Variables**, which are grouped into **Distributions**. This design makes KL divergence computation straightforward—simply iterating over the model's layers and invoking the divergence calculation on the corresponding **Variables** and reference **Variables**.

This approach ensures flexibility, allowing users to select different **Variable** types, such as **GaussianVariable** or **LaplaceVariable**, for weight and bias representation. At the same time, it maintains a structured design to minimize boilerplate code and reduce potential bugs.

One trade-off of this design is that users need to implement their own probabilistic layers if they are not yet included in the framework. However, this process is streamlined—users need only inherit from **AbstractProbLayer** and implement the forward pass.

Other parts of the framework are designed to combine the advantages of both object-oriented and functional programming. A prime example is the set of utility functions that simplify the casting of standard PyTorch tensors into the Variables and Distributions notation used in the framework.

2.2 Prior Model Training

The pipeline begins by choosing a prior distribution in a Prior **Model Training** step. In the case of a data-dependent prior, we need input data for its training. To keep the explanation straightforward, we describe data handling separately in Section 3.5. Fig. 2 shows the Prior **Model Training** step for a data-dependent prior.

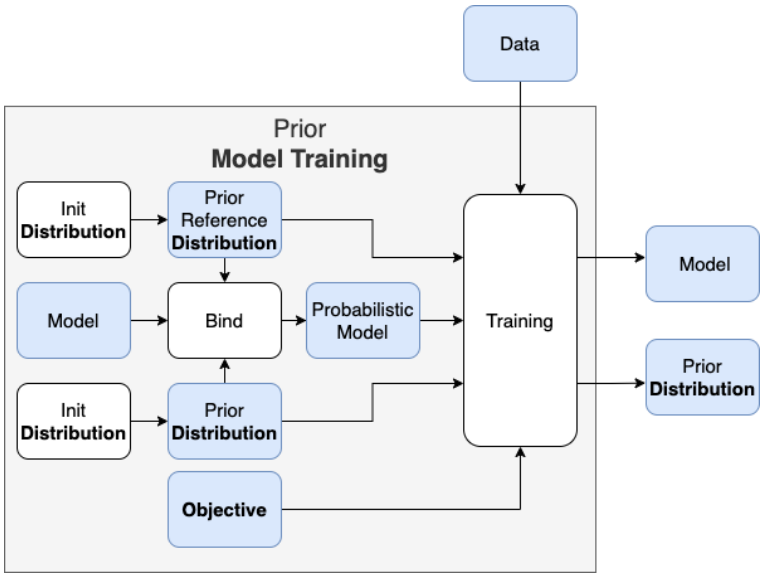

Figure 2: Prior **Model Training** step of a data-dependent prior.

In the Prior **Model Training** step, we initialize both a Prior distribution and a Prior reference distribution. The Prior reference distribution serves as a fixed baseline (sometimes called the *hyperprior*) and is not updated during training. Only the trainable Prior distribution is updated by minimizing an objective that typically combines the empirical loss and a divergence from the fixed reference distribution. By keeping the reference distribution fixed, we ensure that the training focuses on learning the Prior distribution itself, while preserving a consistent "belief baseline".

As a result, we obtain a trained probabilistic NN and a final Prior distribution. Section 3.1 explains how distributions are represented and how KL divergences are calculated.

2.3 Posterior Model Training

After the Prior **Model Training** step, we proceed to the Posterior **Model Training** step (Fig. 3). It mirrors the Prior **Model Training** step except that the Posterior and Posterior reference **Distributions** are initialized as copies of the trained Prior **Distribution**. The Posterior reference **Distribution** is fixed during training, just as the Prior reference **Distribution** was in the Prior **Model Training** step.

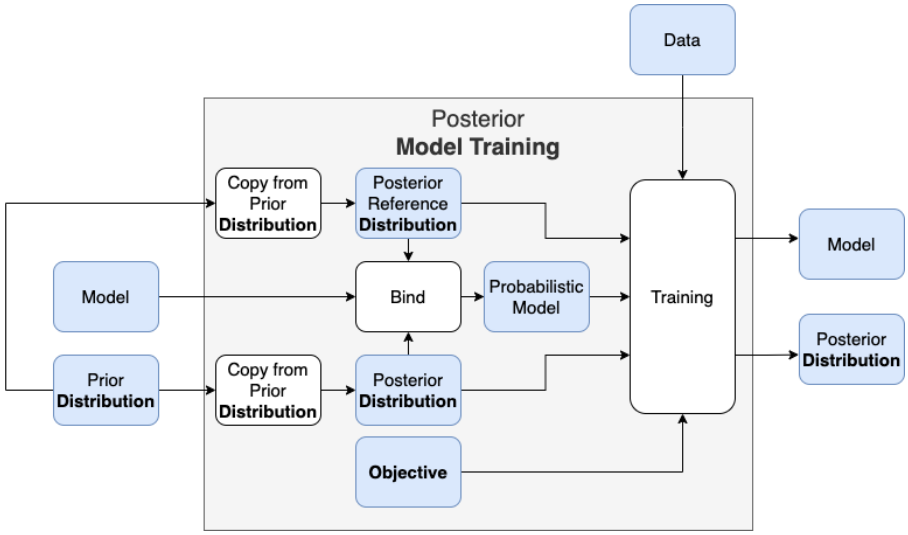

Figure 3: Posterior **Model Training** step.

The Prior reference and Posterior reference **Distribution** are used in both the Prior and Posterior **Model Training** steps. During the Prior **Model Training** step, the reference **Distribution** encodes our initial "prior beliefs" about the parameters of the prior distribution itself. We include a KL term in the training objective that measures how far the trainable Prior **Distribution** deviates from this fixed reference. Likewise, during Posterior **Model Training**, we copy the learned Prior **Distribution** into the Posterior reference **Distribution**, thereby capturing all the knowledge accumulated thus far. The training objective for the Posterior then adds a KL term that compares the updated Posterior **Distribution** to this fixed Posterior reference **Distribution**. In both cases, these reference distributions serve as baselines for regularization and ensure that the learned distributions do not stray arbitrarily far from either our initial or previously acquired knowledge.

The Posterior reference **Distribution** remains fixed throughout the posterior training phase, just as the Prior reference **Distribution** was fixed in the prior training phase. Practically, this reference **Distribution** captures the "prior knowledge" we gained during the Prior **Model Training** step. The trainable Posterior **Distribution** is allowed to update its parameters by balancing empirical performance (i.e., loss on the new data) with this divergence from the fixed Posterior reference **Distribution**. By doing so, the model can refine or specialize its parameters based on new objectives or additional data, yet remain close to what was learned earlier. At the end of the Posterior **Model Training** step, we obtain a final probabilistic neural network accompanied by a trained Posterior **Distribution** that ideally integrates both prior insights and new information.

2.4 Model Evaluation

In the final step, **Model Evaluation**, we assess the trained model's empirical performance and the theoretical guarantees using **Metrics, Losses, and Bounds**. **Metrics** help quantify prediction performance (e.g., accuracy, F1-score), **Losses** measure how well the model fits the data (e.g., negative log-likelihood), and **Bounds** provide theoretical guarantees on generalization (e.g., KL-bound, McAllester bound). A detailed discussion of these evaluation methods is provided in Section 3.4. Fig. 4 illustrates the internal steps of **Model Evaluation**, where Monte Carlo sampling from the trained posterior distribution is used for **Metrics** and **Losses** calculations.

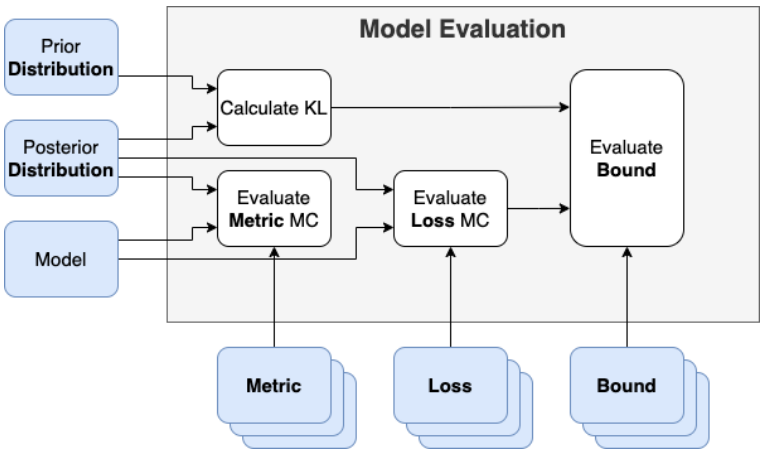

Figure 4: **Model Evaluation** step for a data-dependent prior. Prior and Posterior **Distributions** are used to calculate a KL divergence for bound evaluation, while the NN and Posterior **Distribution** are used with Monte Carlo sampling for **Metrics** and **Losses** calculation.

3 Technical Details of the framework

In the following subsections, we provide an in-depth look at the framework's key components, showing how they work together to serve as the foundational building blocks of our pipeline.

3.1 Variables and Distributions

Everything starts with **Variables** and **Distributions**. Following Blundell et al. (2015), we represent each model weight by a Gaussian distribution $\mathcal{N}(\mu, \sigma)$. From the code perspective, a **Distribution** is simply a collection of **Variables**. Each **Variable** implements an **AbstractVariable** interface, as shown in Listing 1.

Listing 1: Each of the **Variables** should implement the **AbstractVariable**.

```
class AbstractVariable(nn.Module, KLDivergenceInterface, ABC):
    ...
    @abstractmethod
    def sample(self) -> torch.Tensor:
        pass

    @abstractmethod
    def compute_kl(self, other: 'AbstractVariable') -> torch.Tensor:
        pass
```

We use the **GaussianVariable**, which corresponds to a Gaussian distribution $\mathcal{N}(\mu, \text{diag}(\sigma))$ with a diagonal covariance structure. This choice simplifies computation by assuming independent weights, allowing us to use a closed-form solution for the KL divergence between two Gaussian distributions. For an entire **Distribution**, the total KL is the sum of the KL values per **Variable**.

Although Gaussians are our main focus, the framework can also be extended to other distributions, such as Laplace (Pérez-Ortiz et al., 2021), or even more advanced covariance structures. However, more complex structures require more efficient KL-divergence calculations.

3.2 NN to ProbNN Conversion

Given a deterministic NN model and a chosen or trained **Distribution**, we can convert it to a probabilistic NN (ProbNN) via the function in Listing 2:

Listing 2: Conversion of an NN model to a probabilistic NN.

```
from core.model import dnn_to_probnn

model = ... # deterministic model
reference_distribution = ... # e.g. prior reference distribution
distribution = ... # e.g. prior/posterior distribution

dnn_to_probnn(model, reference_distribution, distribution)
# modifies the model in place
```

During training, only the **Distribution** parameters are updated, while the reference **Distribution** remains fixed. In our experiments, we employ the reparameterization approach of Blundell et al. (2015).

To implement **dnn_to_probnn**, we iterate over the **Distribution** (its **Variables**) and attach each variable to the corresponding NN layer (weights, bias, reference weights, etc.). We also redefine each layer's **forward** method to sample from the distribution. An example is shown in Listing 3, where the standard **Linear** layer is wrapped in a **ProbLinear**.

Listing 3: Each layer should implement AbstractProbLayer.

```
from core.layer import AbstractProbLayer

class ProbLinear(nn.Linear, AbstractProbLayer):
    def forward(self, input: Tensor) -> Tensor:
        sampled_weight, sampled_bias = self.sample_from_distribution()
        return F.linear(input, sampled_weight, sampled_bias)
```

Hence, in Fig. 5, one can see how the **ProbLinear** layer differs from the standard **Linear** layer: it has **Variables** for the weights/biases and performs sampling in the **forward** pass.

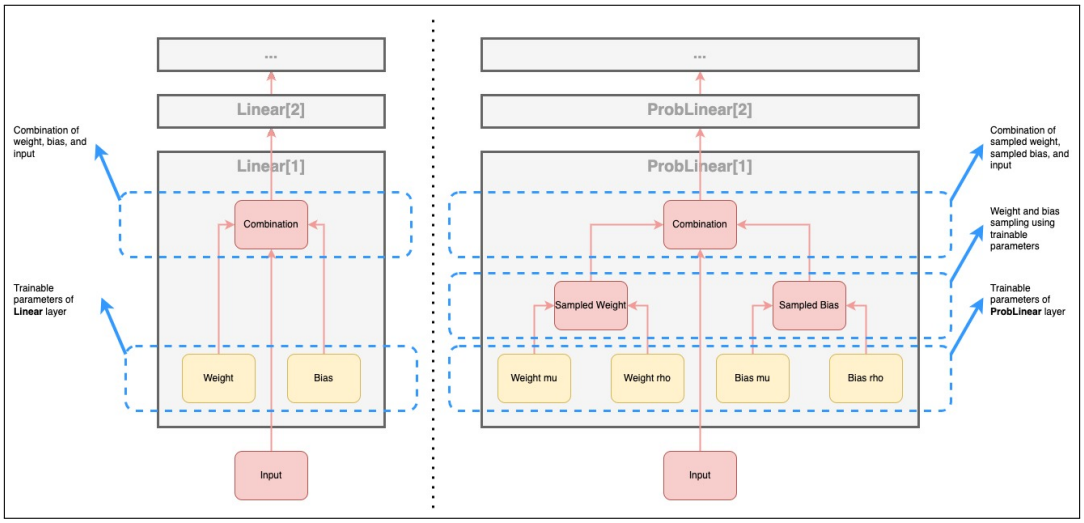

Figure 5: Computation graph difference between an NN with Linear layers (left) and an NN with ProbLinear layers (right). The ProbLinear layer uses attached **Variables** and sampling in its **forward** pass.

3.3 Objectives and Model Training

Both Prior and Posterior **Model Training** steps utilize **Objectives** that combine the empirical loss and a regularization term (often the KL divergence). The training loop is much like standard **PyTorch**, except we add the KL to the **Loss** (Listing 4):

Listing 4: Training a probabilistic NN with an **Objective** that includes KL.

```
for epoch in range(epochs):
    for data, target in loader:
        optimizer.zero_grad()
        output = model(data)
        kl = compute_kl(posterior, prior)
        loss = criterion(output, target)
        objective_value = objective.calculate(loss, kl, num_samples)
        objective_value.backward()
        optimizer.step()
```

Every objective implements an **AbstractObjective** (Listing 5), where **Loss** and KL are combined into a single differentiable scalar. For instance, we have:

- **BBBObjective** from Blundell et al. (2015),
- **FquadObjective** and **FClassicObjective** from Pérez-Ortiz et al. (2021),
- **McAllesterObjective** from McAllester (1999),
- **TolstikhinObjective** from Tolstikhin and Seldin (2013).

Listing 5: Each **Objective** merges **Loss** and KL into a differentiable scalar.

```
class AbstractObjective(ABC):
    ...
    @abstractmethod
    def calculate(self, loss: Tensor, kl: Tensor, num_samples: float) -> Tensor:
        pass
```

3.4 Bounds, Metrics, and Losses

After training, one can evaluate any chosen **Metrics** (e.g., accuracy, F1-score) and **Losses** (e.g., negative log-likelihood) on the dataset. **Metrics** typically summarize empirical performance (e.g., classification accuracy), while **Losses** often refer to the training objective function itself (e.g., NLL in classification). However, to obtain theoretical guarantees on generalization, one computes the **PAC-Bayes bounds**.

Our framework provides two widely used bounds out of the box:

- **KLBound** (Langford and Seeger, 2001),
- **McAllesterBound** (McAllester, 1999),

both of which inherit from **AbstractBound** (Listing 6). Implementing other bounds (e.g., the one proposed in Tolstikhin and Seldin (2013), or non-KL bounds such as Kuzborskij et al. (2024)) is straightforward: the user can simply create a new class that inherits from **AbstractBound** and overrides the `calculate()` method to perform the desired bound calculation.

Listing 6: Each **Bound** implements a calculate method returning the bound value.

```
class AbstractBound(ABC):
    ...
    @abstractmethod
    def calculate(
        self, *args, **kwargs
    ) -> Tuple[Union[Tensor, float], Union[Tensor, float]]:
        pass
```

3.4.1 Bounds Relation to Metrics and Losses

In many PAC-Bayes derivations, the empirical performance is captured by a chosen loss function (e.g., bounded negative log-likelihood), and the bound itself combines this empirical loss with a divergence measure (e.g., KL) between the prior and posterior distributions. A concrete example of how this is computed in practice can be found in Section **??**. When we compute a PAC-Bayes bound via our framework, we typically pass:

- The empirical loss (computed on a designated subset of data specifically used for bounding, often called a **bound set**),
- the divergence between the trainable distribution and the reference distribution,
- and relevant hyperparameters such as the confidence level (ϵ) or the dataset size (n).

The `calculate()` function then outputs a numeric bound on the generalization risk.

3.5 Data

In the previous sections, we focused on the framework components but not on handling the data. Here, we introduce **PBPSplitStrategy** for convenient data management. It ensures that the data used for the bound evaluation does not overlap with the data used for the prior training. **PBPSplitStrategy** consumes a **Data Loader** (inheriting from **AbstractLoader**) and splits its data into subsets for posterior, prior, bound, validation, and test (see Fig. 6).

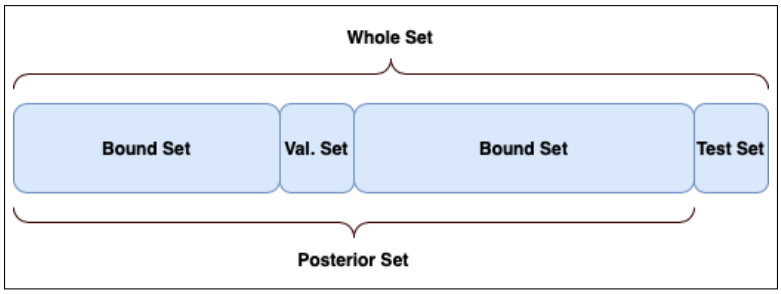

Figure 6: Split strategy used in **PBPSplitStrategy**.

Any class implementing **AbstractSplitStrategy** (Listing 7) can define how the dataset is partitioned. We primarily use **PBPSplitStrategy** along with loaders such as **MNISTLoader** and **CIFAR10Loader**.

Listing 7: **SplitStrategy** offers convenient data manipulation.

```
class AbstractSplitStrategy(ABC):
    @abstractmethod
    def split(self, dataset_loader: AbstractLoader, split_config: Dict) -> None:
        pass
```

4 Example and Extendability

A complete end-to-end example is available in our `pacbb` framework's GitHub repository at `generic_train.py`. It implements the pipeline from Figure 1 and trains both Prior and Posterior with subsequent **Bound** or **Metric** evaluation.

The framework is designed to be **easily extendable**:

- Swapping models (e.g., from **Conv** to **ResNet**) requires changing only a small code block for the model definition.
- Adding a new dataset can be done by writing a new **AbstractLoader**-based class.

- Implementing a new bound is as simple as creating a subclass of **AbstractBound**.

Hence, our framework simplifies and standardizes the PAC-Bayes bound evaluation for neural networks, enabling researchers to focus on innovative methods rather than repeatedly implementing the same processes.

5 Experimental results

To demonstrate the functionality and suitability of our framework, we have conducted experiments comparing the IVON objective (Shen et al., 2024), **ivon**, with the existing objectives **fquad**, **fclassic**, and **bbb**. This allows us to evaluate **ivon**'s performance while also showcasing how our framework enables objective comparisons within a unified setup. The results highlight the flexibility and applicability of our approach to different PAC-Bayes formulations.

To evaluate these objectives, we compute bounds on NLL and 0-1 losses, referred to as NLL risk and 0-1 risk, respectively.

For all objectives, we use a convolutional model with 12 convolutional layers and 3 fully connected layers. However, for **ivon**, we use GoogLeNet instead. We found that GoogLeNet performed worse on **fquad**, **fclassic**, and **bbb**, which is why we used the convolutional model for those objectives.

Each objective undergoes prior and posterior training, where key hyperparameters such as KL penalty, learning rate, and momentum are tuned. Prior training runs for 100 epochs to ensure the model learns a stable prior distribution. Posterior training is performed for a single epoch.

For training, 70% of the data is used for prior training, while the remaining 30% is used for bound evaluation. We use Monte Carlo sampling during bound computation to estimate posterior uncertainty, ensuring a reliable evaluation of each model.

An extensive set of hyperparameter configurations used in our experiments is available in the code repository for further reference.

In the following sections, we present our experiments' results, comparing each objective's performance based on the computed bounds.

The Table 1 presents the best results per objective after hyperparameter tuning on CIFAR10.

Objective	0-1 Loss	0-1 Risk	NLL Loss	NLL Risk
ivon	0.226	**0.246**	0.104	**0.118**
bbb	**0.199**	0.328	**0.077**	0.178
fclassic	0.229	0.324	0.106	0.142
fquad	0.231	0.294	0.09	0.136

Table 1: Best results per objective in hyperparameter tuning, **CIFAR10**.

The **ivon** objective achieves the lowest 0-1 Risk (0.246). More results per objective and their detailed hyperparameter settings can be found in Appendix B.

The Figure 7 visualizes a comparison of 0-1 Loss (blue bars) against 0-1 Risk (red bars). Notably, **bbb** shows the lowest raw loss among these runs, but its risk is higher than **ivon**'s. This gap highlights how focusing purely on empirical performance (0-1 Loss) can differ from the theoretical guarantee (0-1 Risk), emphasizing the importance of a PAC-Bayes viewpoint.

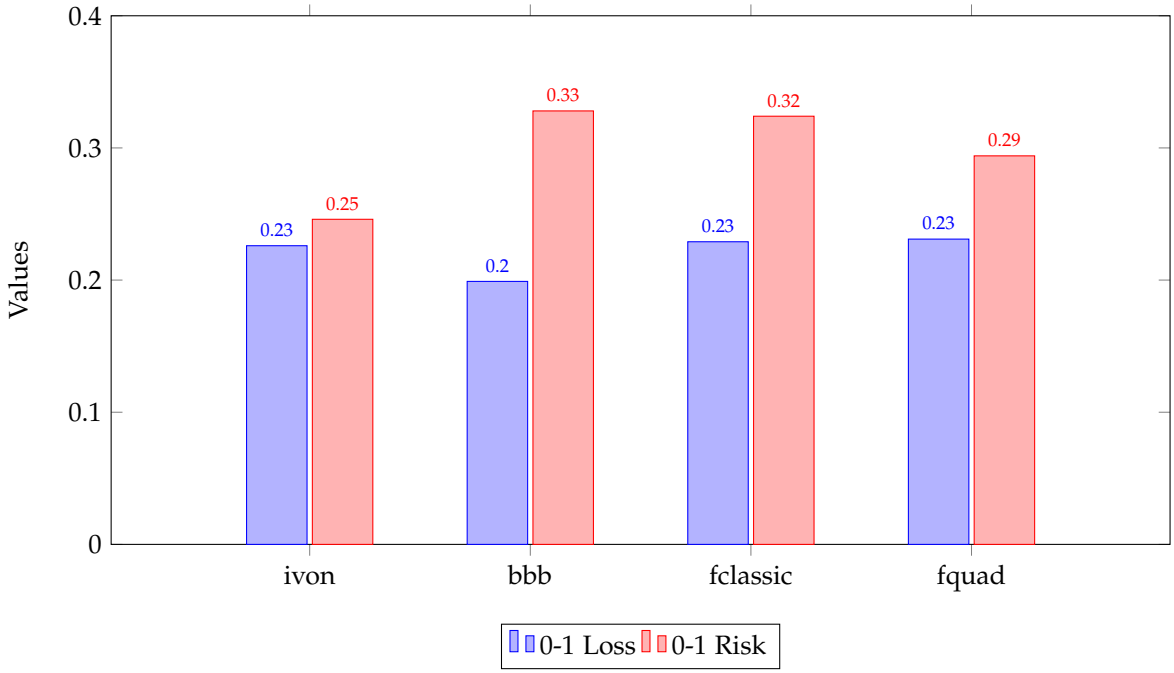

Figure 7: Comparison of **0-1 Loss** and **0-1 Risk** for various objectives on **CIFAR10**.

From the results in Table 1 and Figure 7, we see that **ivon** offers the best balance between empirical loss and PAC-Bayes guarantees, with a 0-1 Risk of 0.246. In contrast, **bbb** achieves a lower empirical loss but ends up with a noticeably higher risk. This underlines how the choice of objective can shape not only model accuracy but also the tightness of the final PAC-Bayes bounds.

6 Conclusion

In this paper, we introduced `pacbb`, a modular and easily extensible PAC-Bayes bounds evaluation framework for neural networks. The framework has the following abstract core components: **Variables** and **Distributions**, **NN to ProbNN Conversion**, **Objectives**, **Bounds**, **Metrics**, and **Losses**. Its modular design simplifies the workflow required to train, evaluate, and compare probabilistic models under PAC-Bayes theories. We provided code examples demonstrating how researchers can integrate their bounds, objectives, and data loaders into the pipeline.

Our experiments compared four different objectives (**ivon**, **bbb**, **fclassic**, and **fquad**) on a ConvNet trained with CIFAR10. The **ivon** objective achieved the lowest **0-1 risk (0.25)**, showing promising performance in this setting. These results show in action how the framework can be used to configure and test new ideas without repeatedly re-implementing foundational training and evaluation components.

References

David McAllester. Pac-bayesian model averaging. in colt '99: Proceedings of the twelfth annual conference on computational learning theory. New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131674.

John Langford and Matthias Seeger. Bounds for averaging classifiers. Technical Report CMU-CS-01-102, Carnegie Mellon University, 2001.

María Pérez-Ortiz, Omar Rivasplata, John Shawe-Taylor, and Csaba Szepesvári. Tighter risk certificates for neural networks, 2021.

S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79 – 86, 1951. doi: 10.1214/aoms/1177729694.

Borja Rodríguez-Gálvez, Omar Rivasplata, Ragnar Thobaben, and Mikael Skoglund. A note on generalization bounds for losses with finite moments, 2024a.

Borja Rodríguez-Gálvez, Ragnar Thobaben, and Mikael Skoglund. More pac-bayes bounds: From bounded losses, to losses with general tail behaviors, to anytime-validity, 2024b.

Ilja Kuzborskij, Kwang-Sung Jun, Yulian Wu, Kyoungseok Jang, and Francesco Orabona. Better-than-kl pac-bayes bounds, 2024.

Gintare Karolina Dziugaite and Daniel M. Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data, 2017.

Pascal Germain, Francis Bach, Alexandre Lacoste, and Simon Lacoste-Julien. Pac-bayesian theory meets bayesian inference, 2017.

Gintare Karolina Dziugaite and Daniel M. Roy. Data-dependent pac-bayes priors via differential privacy, 2019.

Maria Perez-Ortiz, Omar Rivasplata, Benjamin Guedj, Matthew Gleeson, Jingyu Zhang, John Shawe-Taylor, Miroslaw Bober, and Josef Kittler. Learning pac-bayes priors for probabilistic neural networks, 2021.

Gintare Karolina Dziugaite, Kyle Hsu, Waseem Gharbieh, Gabriel Arpino, and Daniel M. Roy. On the role of data in pac-bayes bounds, 2020.

Yi-Shan Wu, Yijie Zhang, Badr-Eddine Chérief-Abdellatif, and Yevgeny Seldin. Recursive pac-bayes: A frequentist approach to sequential prior updates with no information loss, 2024.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8024–8035, 2019.

Ilya O Tolstikhin and Yevgeny Seldin. Pac-bayes-empirical-bernstein inequality. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

Yuesong Shen, Nico Daheim, Bai Cong, Peter Nickl, Gian Maria Marconi, Bazan Clement Emile Marcel Raoul, Rio Yokota, Iryna Gurevych, Daniel Cremers, Mohammad Emtiyaz Khan, et al. Variational learning is effective for large deep networks. In *International Conference on Machine Learning*, pages 44665–44686. PMLR, 2024.

A Data

Each **Data Loader** which is passed in the **split** method of **AbstractSplitStrategy** should inherit from **AbstractLoader** and implement the **load** method (Listing 8).

Listing 8: **AbstracLoader** class: base class for **Data loaders**.

```
class AbstractLoader(ABC):
    def __init__(self, dataset_path):
        self._dataset_path = dataset_path

    @abstractmethod
    def load(self, seed: int) -> Tuple[data.Dataset, data.Dataset]:
        pass
```

Listing 9 shows the implementation of the **MNISTLoader**, which was used for experiments on the MNIST dataset.

Listing 9: **Data loader** for MNIST dataset.

```
class MNISTLoader(AbstractLoader):
    def __init__(self, dataset_path):
        super().__init__(dataset_path)

    def load(self, seed: int) -> Tuple[data.Dataset, data.Dataset]:
        torch.manual_seed(seed)
        # Define the transformation: convert images to tensors and
        # normalize using MNIST-specific mean and std
        transform = transforms.Compose(
            [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
        )
        # Split dataset to Train and Test
        train = datasets.MNIST(
            self._dataset_path, train=True, download=True, transform=transform
        )
        test = datasets.MNIST(
            self._dataset_path, train=False, download=True, transform=transform
        )
        return train, test
```

B Experiment details

Objective	Sigma	LR	KL Penalty	0-1 Loss	0-1 Risk	NLL Loss	NLL Risk	KL/n 10^{-3}
ivon	0.01	0.05	1000	0.226	**0.246**	0.104	0.118	0.55
ivon	0.01	0.1	1000	0.232	0.249	0.103	**0.115**	0.253
ivon	0.005	0.05	1000	0.238	0.257	0.104	0.117	0.480
bbb	0.01	0.01	0.01	**0.199**	0.328	**0.077**	0.178	41.009
bbb	0.001	0.05	0.0001	0.239	0.378	0.088	0.196	43.758
bbb	0.01	0.01	0.000001	0.219	0.408	0.092	0.249	80.005
fclassic	0.001	0.05	0.0001	0.229	0.324	0.106	0.142	5.448
fclassic	0.005	0.01	0.001	0.219	0.356	0.089	0.167	24.986
fclassic	0.005	0.05	0.01	0.23	0.382	0.084	0.175	33.602
fquad	0.001	0.01	0.0001	0.231	0.294	0.09	0.136	9.496
fquad	0.01	0.01	0.01	0.264	0.336	0.094	0.147	11.746
fquad	0.005	0.01	0.0001	0.22	0.34	0.109	0.209	34.222

Table 2: TOP-3 results for each objective in hyperparameter tuning, **CIFAR10**.