Storchastic

Release 0.3.5

Emile van Krieken

Apr 11, 2023

CONTENTS:

Storchastic is a PyTorch library for stochastic gradient estimation in Deep Learning. Stochastic deep learning models are becoming increasingly relevant. For example, they are commonly used in the fields of Variational Inference and Reinforcement Learning. We can formalize stochastic models using so-called **stochastic computation graphs**. While PyTorch computes gradients of deterministic computation graphs automatically, PyTorch will not automatically estimate gradients on such stochastic graphs. This is because they require marginalization over the stochastic nodes in the graph, which is usually intractable and needs to be estimated.

With Storchastic, you can easily define any stochastic deep learning model and let it estimate the gradients for you. Storchastic provides a large range of gradient estimation methods that you can plug and play, to figure out which one works best for your problem. Storchastic provides automatic broadcasting of sampled batch dimensions, which increases code readability and allows implementing complex models with ease.

When dealing with continuous random variables and differentiable functions, the popular reparameterization method is usually very effective. However, this method is not applicable when dealing with discrete random variables or non-differentiable functions. This is why Storchastic has a focus on gradient estimators for discrete random variables, non-differentiable functions and sequence models.

Mail e.van.krieken@vu.nl for any help or questions.

Storchastic requires Python 3.6+ and PyTorch 1.5+. Install using:

```
pip install storchastic
```

To install from source, use:

```
git clone https://github.com/HEmile/storchastic.git
cd storchastic
python setup.py install
```

CHAPTER

ONE

INTRODUCTION TO STORCHASTIC

The following pages introduce the essential concepts and API calls to get started with Storchastic.

1.1 What is Storchastic?

On this page we introduce the ideas behind Storchastic before diving into the code, and explain what kinds of problems it could be applied to. If you are already familiar with stochastic computation graphs [A12] and gradient estimation, you can safely skip this page and start at *Sampling, Inference and Variance Reduction*.

1.1.1 Stochastic computation graphs

PyTorch relies on computation graphs for its automatic differentiation algorithm. These graphs keeps track of all operations that happen while executing PyTorch code by recording the inputs and outputs to PyTorch functions. Each node represents the output of some function. Consider the differentiable function

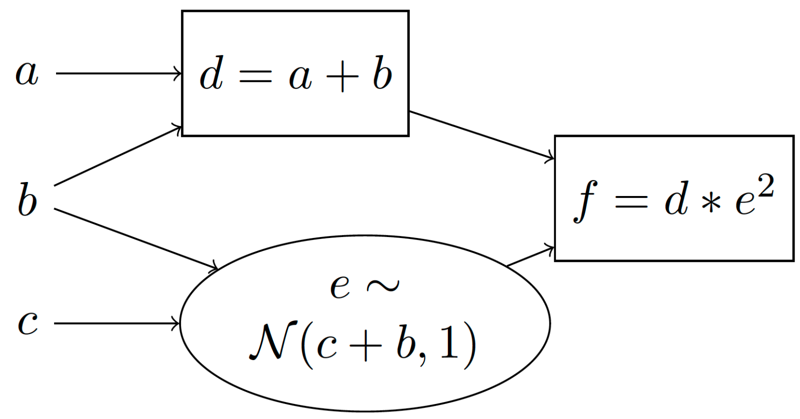
$$f = (a + b) \cdot (b + c)^2$$

This function can be represented using a (deterministic) computation graph as

$$\begin{tikzcd}
a \arrow[r] & \boxed{d = a + b} \arrow[dr] & \\
b \arrow[ur] \arrow[dr] & & \boxed{f = d * e^2} \\
c \arrow[r] & \boxed{e = c + b} \arrow[ur] &
\end{tikzcd}$$

By assigning to a, b, c a concrete value, we can deterministically compute the value of f. PyTorch then uses reverse-mode differentiation on such graphs to find the derivatives with respect to the parameters.

However, in many applications we are interested in computation graphs with *stochastic nodes*. Stochastic nodes are when we take a sample from a distribution, and use the resulting sample to compute the output. For example, suppose that we **sample** e from a normal distribution with mean *c+b* and standard deviation 1. We can represent this using a *stochastic* computation graph:



We use rectangles to denote deterministic computations and ellipses to denote stochastic computations. We can also equivalently represent this using a *generative story*:

1. Compute $d = a + b$
2. Sample $e \sim \mathcal{N}(c + b, 1)$[1]
3. Compute $f = d \cdot e^2$

A generative story is a nice and easy to understand way to show how outputs are generated step-by-step. The goal of Storchastic is to be able to write code that looks just like a generative story. Because of this, we will present both stochastic computation graphs and their associated generative stories in these tutorials.

A very common question in stochastic computation graphs is: What is the *expected* value of f? Mathematically, this is computed as:

$$\mathbb{E}_{e\sim\mathcal{N}(c+b,1)}[(a+b)\cdot e^2] = \int p(e|b,c)\cdot(a+b)\cdot e^2\, de$$

This expression requires computing the integral over all values of e, which is generally intractable.[2] A very common method to approximate expectations is to use *Monte Carlo* methods: Take some (say S) samples of e from the normal distribution and average out the resulting values of f:

$$\mathbb{E}_{e\sim\mathcal{N}(c+b,1)}[(a+b)\cdot e^2] \approx \frac{1}{S}\sum_{i=1}^{S}(a+b)\cdot e_i^2, \quad e_1, ..., e_S \sim \mathcal{N}(c+b,1) \tag{1.1}$$

1.1.2 Gradient estimation

We have shown a simple model with a stochastic node, and we have shown how to compute samples of the output. Next, assume we are interested in the derivative with respect to input c $\frac{\partial}{\partial c}\mathbb{E}_{e\sim\mathcal{N}(c+b,1)}[(a+b)\cdot e^2]$. For the same reason as before, we will use Monte Carlo estimation and sample some values from the distribution to estimate the derivatives.

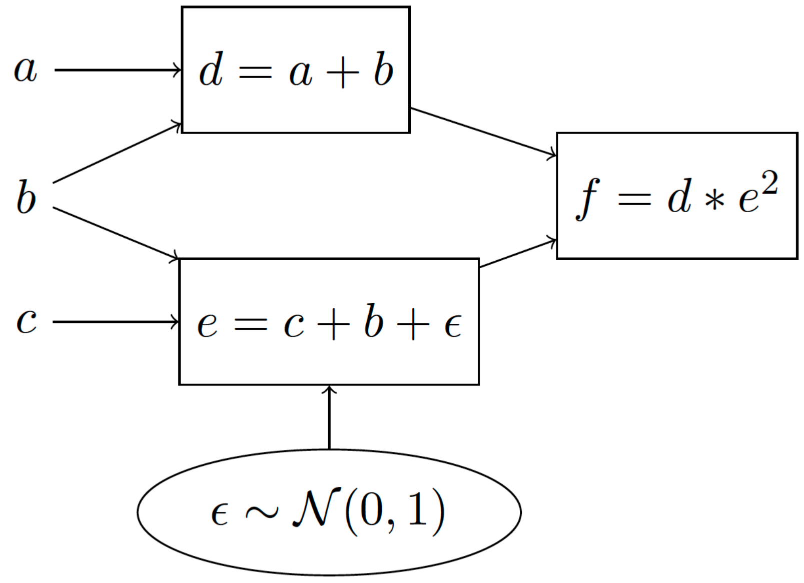
There is however a big issue here: Sampling is not a differentiable procedure! An easy way to see this is by looking at equation (1.1): c does not appear in the Monte Carlo estimation. This means we cannot use reverse-mode differentiation to compute the derivatives with respect to the inputs b, c. Luckily, we can use *gradient estimation methods* [A11].

[1] $\mathcal{N}(\mu, \sigma)$ is a normal distribution with mean μ and standard deviation σ.

[2] For a simple expression like this, a closed-form analytical form can pretty easily be found. However, usually our models are much more complex and non-linear.

The pathwise derivative

A well known gradient estimation method is the *pathwise derivative* [A3] which is commonly referred to in Machine Learning as *reparameterization* [A6]. We explain this estimation method by transforming the previous stochastic computation graph to one that is equivalent:



Which has the following generative story:

1. Compute $d = a + b$
2. Sample $\epsilon \sim \mathcal{N}(0, 1)$
3. Compute $e = c + b + \epsilon$
4. Compute $f = d * e^2$.

The idea behind the pathwise derivative is to move the sampling procedure outside of the computation path, so that the derivatives with respect to b, c can now readily be computed using automatic differentiation! It works because it shifts the mean of the 0-mean normal distribution by $c + b$.

Unfortunately, this does not end our story, because the pathwise derivative has two heavy assumptions that limit its applicability. The first is that a reparameterization must exist for the distribution to sample from. For the normal distribution, this reparameterization is very simple, and a reparameterization has been derived for many other useful continuous distributions. However, no (unbiased[3]) reparameterization exists for discrete distributions! Secondly, the pathwise derivative requires there to be a differentiable path from the sampling step to the output. In many applications, such as in Reinforcement Learning, this is not the case.

[3] There is a very popular *biased* and low-variance reparameterization called the Gumbel-softmax-trick [A5][A10], though!

The score function

The pathwise derivative is a great choice if it is applicable because it is unbiased and usually has low variance. When it is not applicable, we can turn to the *score function*, which is known in Reinforcement Learning as *REINFORCE*. Rewrite f as a function of e using $f(e) = (a + b) \cdot e^2$. Then

$$\begin{aligned}
\frac{\partial}{\partial c}\mathbb{E}_{e\sim\mathcal{N}(c+b,1)}[f(e)] &= \frac{\partial}{\partial c}\int p(e|b,c)f(e)de \\
&= \int \frac{\partial}{\partial c}p(e|b,c)\frac{p(e|b,c)}{p(e|b,c)}f(e)de \\
&= \int p(e|b,c)f(e)\frac{\partial}{\partial c}\log p(e|b,c)de \\
&= \mathbb{E}_{e\sim\mathcal{N}(c+b,1)}\left[f(e)\frac{\partial}{\partial c}\log p(e|b,c)\right]
\end{aligned}$$

By introducing the $\log p(e|b, c)$ term in the expectation, Monte Carlo samples now depend on c and so we can compute a derivative with respect to c! Additionally, the score function can be used for any probability distribution and also works for non-differentiable functions f: It is universally applicable!

That sounds too good to be true, and unfortunately, it is. The score function is notorious for having very high *variance*. The variance of an estimation method can be seen as the average difference between the samples. That means we will need to look at many more samples to get a good idea of what gradient direction to follow.

Luckily, there is a significant amount of literature on variance-reduction methods, that aim to reduce the variance of the score function. These greatly help to apply stochastic computation graphs in practice! Storchastic implements many of these variance-reduction methods, to allow using stochastic computation graphs with non-differentiable functions and discrete distributions.

1.1.3 Applications

Next, we show some common applications of gradient estimation to get an idea of what kind of problems Storchastic can be useful for.

Reinforcement Learning

In Reinforcement Learning (RL), gradient estimation is a central research topic. The popular policy gradient algorithm is the score function applied to the MDP model that is common in RL:

$$\nabla_\theta J(\theta) \propto \mathbb{E}_{s\sim p_\theta(s),a\sim p_\theta(a|s)}[Q_\pi(s,a)\nabla_\theta \log p_\theta(a|s)]$$

Decreasing the variance of this estimator is a very active research area, as lower-variance estimators generally require fewer samples to train the agent. This is often done using so-called "actor-critic" algorithms, that reduce the variance of the policy gradient estimator using a critic which predicts how good an action is relative to other possible actions. Other recent algorithms employ the pathwise derivative to make use of the gradient of the critic [A4][A9]. There is active work on generalizing these ideas to stochastic computation graphs [A13].

Variational Inference

Variational inference is a general method for Bayesian inference. It introduces an approximation to the posterior distribution, then minimizes the distance between this approximation and the actual posterior. In the deep learning era, so-called 'amortized inference' is used, where the approximation is a neural network that predicts the parameters of the approximate distribution. To train the parameters of this neural network, samples are taken from the approximate posterior, and gradient estimation is used. For continuous posteriors, the pathwise derivative is usually employed [A6], but for discrete posteriors, the choice of gradient estimator is an active area of research [A5].

Discrete Random Variables

Discrete random variables are challenging to deal with in practice, but have many promising applications. Deep learning usually acts in the continuous space and discrete random variables are a theoretically motivated way to do some computation in the discrete world. This allows deep learning methods to make clear cut decisions, instead of a continuous attention vector over all options which does not scale in practice.

For example, a variational autoencoder (VAE) with a discrete latent space could be useful to discern properties on the data. Other applications include querying Wikipedia within a language model [A7], learning how to generate computer programs [A1][A8] and hard attention layers [A2]. Additionally, sequence models such as neural machine translation can be trained directly on BLEU scores using gradient estimation.

Footnotes

References

1.2 Sampling, Inference and Variance Reduction

Storchastic allows you to define stochastic computation graphs using an API that resembles generative stories. It is designed with plug-and-play in mind: it is very easy to swap in different gradient estimation methods to compare their performance on your task. In this tutorial, we apply gradient estimation to a simple and small problem.

1.2.1 Converting generative stories

We return to the generative story from *Stochastic computation graphs*:

1. Compute $d = a + b$
2. Sample $e \sim \mathcal{N}(c + b, 1)$
3. Compute $f = d \cdot e^2$

This story is easily converted using the following code:

```python
import torch
import storch
from torch.distributions import Normal
from storch.method import Reparameterization, ScoreFunction

def compute_f(n):
    a = torch.tensor(5.0, requires_grad=True)
```

(continues on next page)

(continued from previous page)

```
8      b = torch.tensor(-3.0, requires_grad=True)
9      c = torch.tensor(0.23, requires_grad=True)
10     d = a + b
11
12     # Sample e from a normal distribution using reparameterization
13     normal_distribution = Normal(b + c, 1)
14     method = Reparameterization("e", n_samples=n)
15     e = method(normal_distribution)
16
17     f = d * e * e
18     return f, c
```

Lines 10 and 17 represent the deterministic nodes. Lines 13-15 represent the stochastic node: We sample a value from the normal distribution using reparameterization (or the pathwise derivative). The `storch.method.Reparameterization` class is a subclass of `storch.method.Method`. Subclasses implement functionality for sampling and gradient estimation, and you can subclass `storch.method.Method` to implement new methods gradient estimation methods. Furthermore, `storch.method.Method` subclasses `torch.nn.Module`, which makes it easy for them to become part of a PyTorch model.

`storch.method.Reparameterization` is initialized with the variable name "e". This is done to initialize the **plate** that corresponds to this sample. We will introduce plates later on. Furthermore, they have an optional *n_samples* option, which controls how many samples are taken from the normal distribution. Note that the method is called directly on the distribution (`torch.distributions.Distribution`) to sample from.

1.2.2 Gradient estimation

Great. Now how to get the derivative with respect to c? Storchastic requires you to register *cost nodes* using `storch.add_cost()`. These are leave nodes that will be minimized. When all cost nodes are registered, `storch.backward()` is used to estimate the gradients:

```
>>> f, c = compute_f(1)
>>> storch.add_cost(f, "f")
>>> storch.backward()
tensor(3.0209, grad_fn=<AddBackward0>)
>>> c.grad
tensor(-4.9160)
```

The second line registers the cost node with the name "f", and the third line computes the gradients, where PyTorch's automatic differentiation is used for deterministic nodes, and Storchastic's gradient estimation methods for stochastic nodes. `storch.backward()` returns the estimated value of the sum of cost nodes, which in this case is just f.

We also show the estimated gradient with respect to c (-4.9160). Note that this gradient is stochastic! Running the code another time, we get -12.2537.

1.2.3 Computing gradient statistics

We can estimate the mean and variance of the gradient as follows:

```
19 n = 1
20 gradient_samples = []
21 for i in range(1000):
22     f, c = compute_f(n)
23     storch.add_cost(f, "f")
24     storch.backward()
25     gradient_samples.append(c.grad)
26 gradients = storch.gather_samples(gradient_samples, "gradients")
```

```
>>> storch.variance(gradients, "gradients")
Deterministic tensor(16.7321) Batch links: []
>>> print(storch.reduce_plates(gradients, "gradients"))
Deterministic tensor(-11.0195) Batch links: []
```

Alright, a few things to note. `storch.gather_samples()` is a function that takes a list of tensors that are (conditionally) independent samples of some value, in this case the gradients. Like most other methods in Storchastic, it returns a `storch.Tensor`, in this case a `storch.IndependentTensor`:

```
>>> type(gradients)
<class 'storch.tensor.IndependentTensor'>
```

`storch.Tensor` is a special "tensor-like" object which wraps a `torch.Tensor` and includes extra metadata to help with estimating gradients and keeping track of the plate dimensions. Plate dimensions are dimensions of the tensor of which we know conditional independency properties. We can look at the plate dimensions of a `storch.Tensor` using

```
>>> gradients.plates
[('gradients', 1000, tensor(0.0010))]
```

The gradients tensor has one plate dimension with name "gradients" (as we defined using `storch.gather_samples()`). As we simulated the gradient 1000 times, the size of the plate dimension is 1000. The third value is the **weight** of the samples. In this case, samples are weighted identically (that is, the weight is 1/1000), which corresponds to a normal monte carlo sample.

Note that we used the plate dimension name "gradients" in `storch.variance(gradients, "gradients")`. With this we mean that we compute the variance over the gradient plate dimension, which represent the different independent samples of gradient estimates.

1.2.4 Reducing variance

Next, let us try to reduce the variance. A simple way to do this is to use more samples of e. In line 14 (`method = Reparameterization("e", n_samples=n)`, we pass the amount of samples to use for this method. Let's use 10 by setting line 19 to `n = 10`, and compute the variance again:

```
>>> storch.variance(gradients, "gradients")
Deterministic tensor(1.6388) Batch links: []
```

By using 10 times as many samples, we reduced the variance by (about) a factor 10. Note that we did not have to change any other code but changing the value of n. Storchastic is designed so that all (left-broadcastable!) code supports both using a single or multiple samples. Using more samples is an easy way to reduce variance. Storchastic automatically parallelizes the computation over the different samples, so that if your gpu has enough memory, there is (usually) almost no overhead to using more samples, yet we can get better estimates of the gradient!

1.2.5 Using different estimators

Storchastic is designed to make swapping in different gradient estimation as easy as possible. For instance, say we want to use the score function instead of reparameterization. This is done as follows:

```python
def compute_f(n):
    a = torch.tensor(5.0, requires_grad=True)
    b = torch.tensor(-3.0, requires_grad=True)
    c = torch.tensor(0.23, requires_grad=True)
    d = a + b

    # Sample e from a normal distribution using reparameterization
    normal_distribution = Normal(b + c, 1)
    method = ScoreFunction("e", n_samples=n, baseline_factory=None)
    e = method(normal_distribution)

    f = d * e * e
    return f, c
```

Note how we only changed the line (`method = Reparameterization("e", n_samples=n)`) where we defined the gradient estimation method to now create a `storch.method.ScoreFunction` instead of `storch.method.Reparameterization`. Let's see the variance of this method (using 1 sample):

```
>>> storch.variance(gradients, "gradients")
Deterministic tensor(748.1914) Batch links: []
```

Ouch, that really is much higher than using Reparameterization! While the score function is much more generally applicable than reparameterization (as it can be used for discrete distributions and non-differentiable functions), it clearly has a prohibitive large variance. Storchastic also has the `storch.method.Infer` gradient estimation method, which automatically applies reparameterization if possible and otherwise uses the score function.

Can we do something about the large variance? Using more samples is always an option. To get the variance in the same ballpark as a single-sample reparameterization, we would need to use about 748.2/16.7 samples, or about n=45!

```
>>> storch.variance(gradients, "gradients")
Deterministic tensor(17.0591) Batch links: []
```

Luckily, we can make efficient reuse of the multiple samples we take. Note how we set `baseline_factory=None` when defining the `storch.method.ScoreFunction`. A baseline is a very common variance reduction method that subtracts a value from the cost function to stabilize the gradient. A simple but effective one is the batch average baseline (`storch.method.baseline.BatchAverage`) that subtracts the average of the other samples. Simply change `ScoreFunction("e", n_samples=n, baseline_factory="batch_average")`. Let's use 20 samples:

```
>>> storch.variance(gradients, "gradients")
Deterministic tensor(16.8761) Batch links: []
```

Sweet! We used fewer than halve of the samples, yet get a lower variance than before. For complicated settings where reparameterization is not an option, strong variance reduction is unfortunately very important for efficient algorithms.

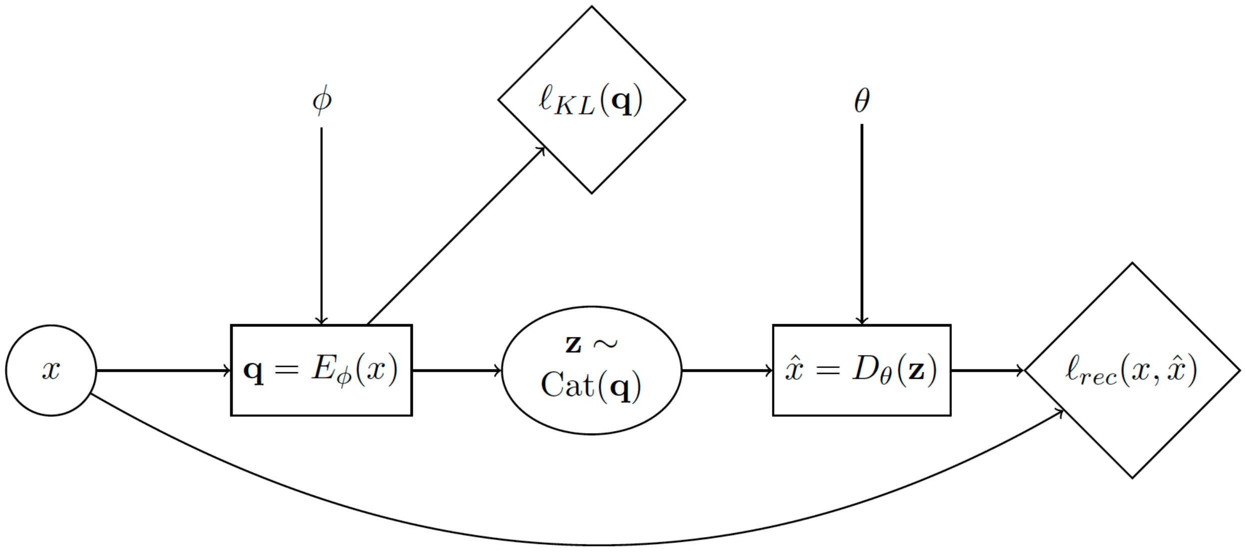
For full code of this example, go to *Introduction Example*.

1.3 Discrete Gradient Estimation

Next, we are going to dive deeper into Storchastic's API and discuss how to use gradient estimation to train a variational auto-encoder with a discrete latent space on MNIST.

1.3.1 Discrete Variational Auto-Encoder

A variational auto-encoder (VAE) is a popular family of generative deep learning models that use variational inference and gradient estimation [B3]. See [B1] for a thorough mathematical introduction. In this tutorial, we show how to train this model when using a discrete latent space using Storchastic. Let's first look at the stochastic computation graph of VAEs:



The diamond nodes are deterministic nodes that represent cost functions (or "losses"). The corresponding generative story is:

1. Sample a datapoint x from the dataset.
2. Compute $\mathbf{q} = E_\phi(x)$. This is the logits of the variational distribution encoded from x.
3. Sample $\mathbf{z} \sim \mathrm{Cat}(\mathbf{q})$ from categorical distribution with parameters $\mathbf{q}$
4. Compute cost node $\ell_{KL}(\mathbf{q})$. This is the KL-divergence of the prior with the variational posterior.
5. Compute $\hat{x} = D_\theta(\mathbf{z})$. This decodes from $\mathbf{z}$ the reconstructed input $\hat{x}$.
6. Compute cost node $\ell_{rec}(x, \hat{x})$. This is the 'reconstruction' loss.

Because of the sample of $\mathbf{z} \sim \mathrm{Cat}(\mathbf{q})$, we need to use gradient estimation. Otherwise, we cannot train the parameters ϕ! Reparameterization is not an option, as we are dealing with discrete random variables. Let's see how we can solve this using Storchastic.

First, we define the encoder E_ϕ and decoder D_θ networks. For both the encoder and decoder we use 2 fully connected hidden layers with 512 and 256 hidden units. For the latent space, we will use 2 conditionally independent categorical distributions of 10 choices. This means there are $10^2 = 100$ possible configurations to sum over.

```python
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```python
import storch
from torch.distributions import OneHotCategorical


class DiscreteVAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 2 * 10)
        self.fc4 = nn.Linear(2 * 10, 256)
        self.fc5 = nn.Linear(256, 512)
        self.fc6 = nn.Linear(512, 784)

    def encode(self, x):
        h1 = self.fc1(x).relu()
        h2 = self.fc2(h1).relu()
        return self.fc3(h2)

    def decode(self, z):
        h3 = self.fc4(z).relu()
        h4 = self.fc5(h3).relu()
        return self.fc6(h4).sigmoid()
```

In `DiscreteVAE.__init__()`, we pass the `storch.method.Method` that we will use to estimate gradients with respect to $\mathbf{q}$.

Time to translate our generative story!

```python
def generative_story(method: storch.method.Method, model: DiscreteVAE, data: torch.
↪Tensor):
    x = storch.denote_independent(data.view(-1, 784), 0, "data")
```

`data` is a tensor containing a minibatch of MNIST images of shape *(minibatch,28,28)*. As we mentioned in our generative story, we **sample** a datapoint x from the dataset. We thus have to tell Storchastic that the first minibatch dimension is an **independent*** dimension! We give this dimension the plate name “data”.

```python
    # Encode data. Shape: (data, 2 * 10)
    q_logits = model.encode(x)
    # Shape: (data, 2, 10)
    q_logits = q_logits.reshape(-1, 2, 10)
    # Define variational posterior
    q = OneHotCategorical(probs=q_logits.softmax(dim=-1))
    # Sample from variational posterior. Shape: (amt_samples, data, 2, 10)
    z = method(q)
```

Here, we define a one-hot categorical distribution based on the logits from the encoder. Using the passed `storch.method.Method`, we sample from this distribution to get `z`. We have to reshape the logits and sample to properly denote that we want 2 conditionally independent categorical latent variables with 10 choices, instead of 1 categorical latent variable with 20 choices.

The KL-divergence loss $\ell_{KL}(\mathbf{q})$ can be computed using

```
    prior = OneHotCategorical(probs=torch.ones_like(q.probs) / 10.0)
    # Shape: (data)
    KL_div = torch.distributions.kl_divergence(q, prior).sum(-1)
    storch.add_cost(KL_div, "kl-div")
```

We define an uniform prior over the categorical random variables, and then use `torch.distributions.kl_divergence()` to analytically compute the KL-divergence between this prior and the variational posterior we found. We want to minimize this KL-divergence, so we use `storch.add_cost()` to register this node.

Next, we reconstruct $\hat{x}$ from z, and compute the reconstruction loss:

```
    z_in = z.reshape(z.shape[:-2] + (2 * 10,))
    # Shape: (amt_samples, data, 784)
    reconstruction = model.decode(z_in)
    bce = torch.nn.BCELoss(reduction="none")(reconstruction, x).sum(-1)
    storch.add_cost(bce, "reconstruction")

    return z
```

Here we use our model to decode z, then compute the binary cross entropy between the reconstruction and the original datapoint. The computation of the binary cross entropy is a bit subtle. We first pass "none" to `reduction` to denote that we do not want to sum over the result, yet then we still sum over the last dimension afterwards. Why not:

```
>>> torch.nn.BCELoss(reduction="sum")(reconstruction, x)
ValueError: Got an input tensor with too few dimensions. We expected 2 plate dimensions.␣
↪Instead, we found only 0 dimensions. Violated at dimension 0
```

This error means that we have removed a **plate dimension**. Setting :python:`reduction="sum" makes the loss function return only a single number. In Storchastic, it is not allowed to remove dimensions that are denoted as independent unless the user explicitly asks Storchastic to do so. We can also, for example, not do the following:

```
>>> torch.mean(bce)
ValueError: Got an input tensor with too few dimensions. We expected 2 plate dimensions.␣
↪Instead, we found only 0 dimensions. Violated at dimension 0
```

Why would it not be allowed here, as we are just computing our loss function? We would average over our samples anyways? Storchastic is no longer able to compute gradient estimates after one would take the mean. For example, if we use the score function and we take multiple samples, we would need to multiply the log probability of the samples with the corresponding computed loss. This happens during inference in `storch.backward()`. If we would have taken the mean, we could no longer recover the individual loss outputs!

To make life easier, Storchastic is designed with "fail-quick" in mind. Therefore, if code is written that is likely to result in such errors, it will crash!

Next, we load the MNIST dataset[1]:

```
from torchvision import datasets, transforms

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST(
        "./data", train=True, download=True, transform=transforms.ToTensor(),
    ),
    batch_size=64,
```

(continues on next page)

[1] Note that it is best practice to use the binarized MNIST dataset as proposed by http://proceedings.mlr.press/v15/larochelle11a/larochelle11a.pdf.

(continued from previous page)

```
    shuffle=True,
)
```

Finally, we put everything together in the training loop and add a training evaluation that also computes gradient variance:

```
def train(method: storch.method.Method, train_loader):
    model = DiscreteVAE()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    for epoch in range(5):
        print("Epoch:" + str(epoch + 1))
        for i, (data, _) in enumerate(train_loader):
            optimizer.zero_grad()

            generative_story(method, model, data)
            storch.backward()
            optimizer.step()
            if i % 300 == 0:
                evaluate(method, model, data, optimizer)


def evaluate(method: storch.method.Method, model: DiscreteVAE, data, optimizer):
    gradients = []
    for i in range(100):
        optimizer.zero_grad()

        z = generative_story(method, model, data)
        elbo = storch.backward()
        gradients.append(z.param_grads["probs"])
    gradients = storch.gather_samples(gradients, "gradients")

    print(
        "Training ELBO "
        + str(elbo.item())
        + ". Gradient variance "
        + str(storch.variance(gradients, "gradients")._tensor.item())
    )
```

We use an optimizer as normal, however, we call `storch.backward()` to compute the gradients. To get the gradient for the gradient variance computation, we use `storch.StochasticTensor.param_grads`. In this example, we will do 5 training epochs.

1.3.2 Experimenting with the Discrete VAE

Let us naively try with the score function, no baseline and a single sample:

```
>>> train(storch.method.ScoreFunction("z", n_samples=1, baseline_factory="None"))
Epoch:1
Training ELBO 543.1165771484375. Gradient variance 12860.05078125
Training ELBO 192.7163848876953. Gradient variance 1841.5672607421875
Training ELBO 196.0297088623047. Gradient variance 146922.4375
```

(continues on next page)

(continued from previous page)

```
Training ELBO 203.4992218017578. Gradient variance 277160.9375
Epoch:2
Training ELBO 191.17823791503906. Gradient variance 28171.1796875
Training ELBO 193.627685546875. Gradient variance 130113.6953125
Training ELBO 200.20506286621094. Gradient variance 210252.90625
Training ELBO 197.44796752929688. Gradient variance 523125.375
Epoch:3
Training ELBO 202.95068359375. Gradient variance 39035.52734375
Training ELBO 195.01597595214844. Gradient variance 104070.5078125
Training ELBO 198.34580993652344. Gradient variance 7388.16845703125
Training ELBO 192.8509979248047. Gradient variance 457.5860595703125
Epoch:4
Training ELBO 184.8743896484375. Gradient variance 1029.0767822265625
Training ELBO 201.11277770996094. Gradient variance 168409.390625
Training ELBO 199.73178100585938. Gradient variance 13173.759765625
Training ELBO 198.03358459472656. Gradient variance 4439.3017578125
Epoch:5
Training ELBO 194.9002685546875. Gradient variance 18869.560546875
Training ELBO 188.87930297851562. Gradient variance 45681.5546875
Training ELBO 209.60585021972656. Gradient variance 5402.30615234375
Training ELBO 190.37799072265625. Gradient variance 34421.51953125
```

It clearly is training, but the training ELBO seems rather unstable. Let's again use the batch average baseline and 10 samples:

```
>>> train(storch.method.ScoreFunction("z", n_samples=10, baseline_factory="batch_average
↪"), train_loader)
Epoch:1
Training ELBO 543.8447265625. Gradient variance 0.00031363777816295624
Training ELBO 195.42037963867188. Gradient variance 4.808237075805664
Training ELBO 176.24708557128906. Gradient variance 20.694387435913086
Training ELBO 175.12139892578125. Gradient variance 114.5234375
Epoch:2
Training ELBO 161.55543518066406. Gradient variance 117.06814575195312
Training ELBO 163.81021118164062. Gradient variance 764.1204223632812
Training ELBO 167.0965576171875. Gradient variance 0.7521735429763794
Training ELBO 163.2193145751953. Gradient variance 1854.9320068359375
Epoch:3
Training ELBO 165.54237365722656. Gradient variance 34.32332229614258
Training ELBO 159.99176025390625. Gradient variance 121.91394805908203
Training ELBO 149.61558532714844. Gradient variance 4.967251777648926
Training ELBO 165.6819305419922. Gradient variance 2.564244270324707
Epoch:4
Training ELBO 156.40789794921875. Gradient variance 215.02999877929688
Training ELBO 152.97520446777344. Gradient variance 258.04400634765625
Training ELBO 157.0828094482422. Gradient variance 13.990401268005371
Training ELBO 157.7599639892578. Gradient variance 1.4151099920272827
Epoch:5
Training ELBO 164.08978271484375. Gradient variance 391.89794921875
Training ELBO 156.1527862548828. Gradient variance 2.9808785915374756
Training ELBO 154.16932678222656. Gradient variance 10.244932174682617
Training ELBO 151.02488708496094. Gradient variance 4701.791015625
```

Much better! Our variance reduction techniques help reduce the gradient variance by several factors, which results in far lower training ELBO scores.

Another popular technique is the Gumbel-softmax-trick [B2][B4]. This trick uses a continuous that approximates the One Hot Categorical distribution. This distribution allows reparameterization. Because the decoder of the Discrete VAE does not require its inputs to be discrete, we can apply this trick here!

The Gumbel-softmax trick is a biased gradient estimation tool. This means that it is not an estimate of the correct gradient. Using `storch.method.Expect`, we can estimate just how biased it is. Let's edit our evaluation function:

```python
def evaluate(method: storch.method.Method, model: DiscreteVAE, data, optimizer):
    # Compute expected gradient
    optimizer.zero_grad()

    z = generative_story(storch.method.Expect("z"), model, data)
    storch.backward()
    expected_gradient = z.param_grads["probs"]

    # Collect gradient samples
    gradients = []
    for i in range(100):
        optimizer.zero_grad()

        z = generative_story(method, model, data)
        elbo = storch.backward()
        gradients.append(z.param_grads["probs"])

    gradients = storch.gather_samples(gradients, "gradients")
    mean_gradient = storch.reduce_plates(gradients, "gradients")
    bias_gradient = (
        storch.reduce_plates((mean_gradient - expected_gradient) ** 2)
    ).sum()
    print(
        "Training ELBO "
        + str(elbo.item())
        + " Gradient variance "
        + str(storch.variance(gradients, "gradients")._tensor.item())
        + " Gradient bias "
        + str(bias_gradient._tensor.item())
    )
```

`storch.method.Expect` is not a gradient estimation method, but computes the gradient analytically by summing over all options. Because we have a small latent space of $10^2 = 100$ options, this is viable.

```
>>> train(storch.method.GumbelSoftmax("z", n_samples=1), train_loader)
Epoch:1
Training ELBO 543.6360473632812 Gradient variance 0.00038375251460820436 Gradient bias 2.
↪135414251824841e-05
Training ELBO 204.85665893554688 Gradient variance 5.555451931815725e-15 Gradient bias 5.
↪555451931815725e-15
Training ELBO 205.2622528076172 Gradient variance 5.326468432986786e-15 Gradient bias 5.
↪326468432986786e-15
Training ELBO 212.6741485595703 Gradient variance 0.001139726140536368 Gradient bias 0.
↪5215573906898499
Epoch:2
```

(continues on next page)

(continued from previous page)

```
Training ELBO 213.31932067871094 Gradient variance 6.1461252753858275e-15 Gradient bias␣
↪6.1461252753858275e-15
Training ELBO 202.0615234375 Gradient variance 5.118301615869569e-15 Gradient bias 5.
↪118301615869569e-15
Training ELBO 211.42044067382812 Gradient variance 0.0004477511683944613 Gradient bias 1.
↪2102100849151611
Training ELBO 215.71697998046875 Gradient variance 2.4727771913424235e-12 Gradient bias␣
↪0.6834085583686829
Epoch:3
Training ELBO 221.59030151367188 Gradient variance 1.2252018451690674 Gradient bias 15.
↪654888153076172
Training ELBO 211.0780487060547 Gradient variance 2.080887545607979e-14 Gradient bias 2.
↪080887545607979e-14
Training ELBO 219.01422119140625 Gradient variance 2.0171364578658313e-14 Gradient bias␣
↪2.0171364578658313e-14
Training ELBO 210.15830993652344 Gradient variance 2.049922731561793e-14 Gradient bias 2.
↪049922731561793e-14
Epoch:4
Training ELBO 219.97352600097656 Gradient variance 2.148281552649678e-14 Gradient bias 2.
↪148281552649678e-14
Training ELBO 215.22779846191406 Gradient variance 2.0663158684097738e-14 Gradient bias␣
↪2.0663158684097738e-14
Training ELBO 208.27081298828125 Gradient variance 2.0371725140133634e-14 Gradient bias␣
↪2.0371725140133634e-14
Training ELBO 213.13644409179688 Gradient variance 2.049922731561793e-14 Gradient bias 2.
↪049922731561793e-14
Epoch:5
Training ELBO 202.03463745117188 Gradient variance 1.931527854326376e-14 Gradient bias 1.
↪931527854326376e-14
Training ELBO 209.62664794921875 Gradient variance 2.0262437561147095e-14 Gradient bias␣
↪2.0262437561147095e-14
Training ELBO 212.3344268798828 Gradient variance 1.951563910473908e-14 Gradient bias 1.
↪951563910473908e-14
Training ELBO 209.84085083007812 Gradient variance 1.993457482418748e-14 Gradient bias 1.
↪993457482418748e-14
```

Oof, that is not great! The gumbel softmax does even worse than the score function without variance reduction. Theoretically, using stochastic optimization with gradient estimation only for unbiased gradient estimation methods. We should note that the gumbel-softmax performs much better for larger latent spaces, for example when using 20 categorical latent variables of 10 options.

We could also just use `storch.method.Expect` to train the model:

```
Epoch:1
Training ELBO 543.6659545898438
Training ELBO 175.1640625
Training ELBO 163.3818359375
Training ELBO 158.3362274169922
Epoch:2
Training ELBO 159.03167724609375
Training ELBO 158.54054260253906
Training ELBO 151.9814453125
Training ELBO 162.34519958496094
```

(continues on next page)

(continued from previous page)

```
Epoch:3
Training ELBO 154.2731475830078
Training ELBO 159.92709350585938
Training ELBO 157.92642211914062
Training ELBO 147.97755432128906
Epoch:4
Training ELBO 151.23654174804688
Training ELBO 155.57571411132812
Training ELBO 142.53665161132812
Training ELBO 141.1732635498047
Epoch:5
Training ELBO 152.55979919433594
Training ELBO 154.68777465820312
Training ELBO 151.78952026367188
Training ELBO 156.02206420898438
```

Footnotes

References

CHAPTER TWO

LICENSE

TODO!

CHAPTER

THREE

FAQ

3.1 A method I'm using doesn't play well with the required independent dimensions in Storchastic

An example of this is `torch.nn.Conv2d()`, which expects exactly an input of (N, C, H, W) and cannot have any more independent dimensions to the left of N. However, when sampling using Storchastic, we use the dimensions on the left to keep track of independent samples from different proposal distributions, meaning we might have an input of size (Z, N, C, H, W), which will not fit `torch.nn.Conv2d()`. Can we fix this? Yes!

The function `storch.wrappers.make_left_broadcastable()` helps us out there. It makes sure to flatten all independent dimensions into a single dimension before calling the function, and after calling the function it will restore them. You can call it using *make_left_broadcastable(Conv2d(16, 33, 3))*.

CHAPTER FOUR

NEED HELP?

Send a mail to e.van.krieken@vu.nl or add an issue to https://github.com/HEmile/storchastic.

CHAPTER FIVE

EXAMPLES

Lists several examples of Storchastic to help you get started. Also check out https://github.com/HEmile/storchastic/tree/master/examples.

5.1 Introduction Example

```
import torch
from torch.distributions import Normal
from storch.method import Reparameterization, ScoreFunction
import storch

torch.manual_seed(0)


def compute_f(method):
    a = torch.tensor(5.0, requires_grad=True)
    b = torch.tensor(-3.0, requires_grad=True)
    c = torch.tensor(0.23, requires_grad=True)
    d = a + b

    # Sample e from a normal distribution using reparameterization
    normal_distribution = Normal(b + c, 1)
    e = method(normal_distribution)

    f = d * e * e
    return f, c


# e*e follows a noncentral chi-squared distribution https://en.wikipedia.org/
↪wiki/Noncentral_chi-squared_distribution
# exp_f = d * (1 + mu * mu)
repar = Reparameterization("e", n_samples=1)
f, c = compute_f(repar)
storch.add_cost(f, "f")
print(storch.backward())

print("first derivative estimate", c.grad)

f, c = compute_f(repar)
```

(continues on next page)

(continued from previous page)

```
storch.add_cost(f, "f")
print(storch.backward())

print("second derivative estimate", c.grad)


def estimate_variance(method):
    gradient_samples = []
    for i in range(1000):
        f, c = compute_f(method)
        storch.add_cost(f, "f")
        storch.backward()
        gradient_samples.append(c.grad)
    gradients = storch.gather_samples(gradient_samples, "gradients")
    # print(gradients)
    print("variance", storch.variance(gradients, "gradients"))
    print("mean", storch.reduce_plates(gradients, "gradients"))
    print("st dev", torch.sqrt(storch.variance(gradients, "gradients")))

    print(type(gradients))
    print(gradients.shape)
    print(gradients.plates)


print("Reparameterization n=1")
estimate_variance(Reparameterization("e", n_samples=1))

print("Reparameterization n=10")
estimate_variance(Reparameterization("e", n_samples=10))

print("Score function n=1")
estimate_variance(ScoreFunction("e", n_samples=1))

print("Score function n=45")
estimate_variance(ScoreFunction("e", n_samples=45))

print("Score function with baseline n=20")
estimate_variance(ScoreFunction("e", n_samples=20, baseline_factory="batch_
↪average"))
```

5.2 Discrete Variational Autoencoder

```
import torch
import torch.nn as nn
import storch
from storch.method import ScoreFunction

class DiscreteVAE(nn.Module):
    def __init__(self):
        self.method = ScoreFunction("z", 8, baseline_factory="batch_average")
```

(continues on next page)

(continued from previous page)

```
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 20 * 10)
        self.fc4 = nn.Linear(20 * 10, 256)
        self.fc5 = nn.Linear(256, 512)
        self.fc6 = nn.Linear(512, 784)

    def encode(self, x):
        h1 = self.fc1(x).relu()
        h2 = self.fc2(h1).relu()
        return self.fc3(h2).reshape(logits.shape[:-1] + (20, 10))

    def decode(self, z):
        z = z.reshape(z.shape[:-2] + (20 * 10,))
        h3 = self.fc4(z).relu()
        h4 = self.fc5(h3).relu()
        return self.fc6(h4).sigmoid()

    def KLD(self, q):
        p = torch.distributions.OneHotCategorical(probs=torch.ones_like(q.
→logits) / (1.0 / 10.0))
        return torch.distributions.kl_divergence(p, q).sum(-1)

    def forward(self, x):
        q = torch.distributions.OneHotCategorical(logits=self.encode(x))
        KLD = self.KLD(q)
        z = self.method("z", q, n=8)
        return self.decode(z), KLD

model = DiscreteVAE()
for data in minibatches():
    optimizer.zero_grad()
    # Denote the minibatch dimension as being independent
    data = storch.denote_independent(data.view(-1, 784), 0, "data")

    # Compute the output of the model
    recon_batch, KLD = model(data)

    # Register the two cost functions
    storch.add_cost(KLD)
    storch.add_cost(storch.nn.b_binary_cross_entropy(recon_batch, data,␣
→reduction="sum"))

    # Go backward through both deterministic and stochastic nodes
    average_ELBO, _ = storch.backward()
    print(average_ELBO)

    optimizer.step()
```

```
import torch
import storch
from vae import minibatches, encode, decode, KLD
```

(continues on next page)

(continued from previous page)

```
method = storch.method.ScoreFunction("z", 8, baseline_factory="batch_average")
for data in minibatches():
    optimizer.zero_grad()
    # Denote the minibatch dimension as being independent
    data = storch.denote_independent(data.view(-1, 784), 0, "data")

    # Define the variational distribution given the data, and sample latent␣
↪variables
    q = torch.distributions.OneHotCategorical(logits=encode(data))
    z = method(q)

    # Compute and register the KL divergence and reconstruction losses to form␣
↪the ELBO
    reconstruction = decode(z)
    storch.add_cost(KLD(q))
    storch.add_cost(storch.nn.b_binary_cross_entropy(reconstruction, data,␣
↪reduction="sum"))

    # Go backward through both deterministic and stochastic nodes, and optimize
    average_ELBO, _ = storch.backward()
    optimizer.step()
```

```
import torch
import storch
from vae import minibatches, encode, decode, KLD

method = ScoreFunctionLOO("z", 8)
for data in minibatches():
    optimizer.zero_grad()
    # Denote the minibatch dimension as being independent
    data = storch.denote_independent(data.view(-1, 784), 0, "data")

    # Define variational distribution given data, and sample latent variables
    q = torch.distributions.OneHotCategorical(logits=encode(data))
    z = method(q)

    # Compute and register the KL divergence and reconstruction losses to form␣
↪the ELBO
    reconstruction = decode(z)
    storch.add_cost(KLD(q))
    storch.add_cost(storch.nn.b_binary_cross_entropy(reconstruction, data))

    # Backward pass through deterministic and stochastic nodes, and optimize
    ELBO = storch.backward()
    optimizer.step()
```

```

```

class ScoreFunctionLOO(Method):

def proposal_dist(self, distr: Distribution, amt_samples: int,) -> torch.Tensor:

return distr.sample((amt_samples,))

def weighting_function(self, distr: Distribution, amt_samples: int,) -> torch.Tensor:

  return torch.full(amt_samples, 1/amt_samples)

def estimator(self, tensor: StochasticTensor, cost: CostTensor) -> Tuple[Optional[storch.Tensor], Optional[storch.Tensor]]:

  # Compute gradient function (log-probability) log_prob = tensor.distribution.log_prob(tensor) sum_costs = storch.sum(costs.detach(), tensor.name) # Compute control variate baseline = (sum_costs - costs) / (tensor.n - 1) return log_prob, (1.0 - magic_box(log_prob)) * baseline

CHAPTER

SIX

STORCH TENSORS

To keep track of the stochastic computation graph, Storchastic returns wrapped `torch.Tensor` that are subclasses of `storch.Tensor`. This wrapper contains information that allows Storchastic to analyse the computation graph during inference to properly estimate gradients. Furthermore, `storch.Tensor` contains plate information that allows for automatic broadcasting with other `storch.Tensor` objects with different plate information.

class storch.tensor.**IndependentTensor**(*tensor: torch.Tensor*, *parents: [Tensor]*, *plates: [Plate]*, *tensor_name: str*, *plate_name: str*, *weight: Optional[storch.Tensor]*)

Bases: `Tensor`

Used to denote independencies on a Tensor. This could for example be the minibatch dimension. The first dimension of the input tensor is taken to be independent and added as a batch dimension to the storch system.

stochastic() → bool

Returns
True if this is a stochastic node in the stochastic computation graph, False otherwise.

Return type
bool

CHAPTER

SEVEN

PLATES

CHAPTER EIGHT

GRADIENT ESTIMATION METHODS

8.1 Baselines

class storch.method.baseline.**Baseline**

Bases: ABC, Module

abstract compute_baseline(*tensor: StochasticTensor*, *cost_node: CostTensor*) → Tensor

training: bool

class storch.method.baseline.**BatchAverageBaseline**

Bases: *Baseline*

Uses the average over the other samples as baseline. Introduced by https://arxiv.org/abs/1602.06725

compute_baseline(*tensor: StochasticTensor*, *costs: CostTensor*) → Tensor

training: bool

class storch.method.baseline.**MovingAverageBaseline**(*exponential_decay=0.95*)

Bases: *Baseline*

Takes the (unconditional) average over the different costs.

compute_baseline(*tensor: StochasticTensor*, *cost_node: CostTensor*) → Tensor

training: bool

8.2 Multi-sample estimators

8.3 RELAX

8.4 Unordered set estimator

CHAPTER

NINE

SAMPLING METHODS

class storch.sampling.method.**MonteCarlo**(*plate_name: str*, *n_samples: int = 1*)

Bases: *SamplingMethod*

Monte Carlo sampling methods use simple sampling methods that take n independent samples. Unlike complex ancestral sampling methods such as SampleWithoutReplacementMethod, the sampling behaviour is not dependent on earlier samples in the stochastic computation graph (but the distributions are!).

sample(*distr: Distribution*, *parents: [storch.Tensor]*, *plates: [Plate]*, *requires_grad: bool*)

training: bool

class storch.sampling.method.**SamplingMethod**(*plate_name: str*)

Bases: ABC, Module

forward(*distr: Distribution*, *parents: [storch.Tensor]*, *plates: [Plate]*, *requires_grad: bool*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

mc_sample(*distr: Distribution*, *parents: [storch.Tensor]*, *plates: [Plate]*, *amt_samples: int*) → torch.Tensor

mc_weighting_function(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

on_plate_already_present(*plate: Plate*)

reset() → None

abstract sample(*distr: Distribution*, *parents: [storch.Tensor]*, *plates: [Plate]*, *requires_grad: bool*)

set_mc_sample(*new_sample_func: Callable[[Distribution, [storch.Tensor], [Plate], int], torch.Tensor]*) → *SamplingMethod*

Override storch.Method specific sampling functions. This is called when initializing a storch.Method that has slightly different MC sampling semantics (for example, reparameterization instead of normal sampling). This allows for compatibility of different *storch.Method*'s with different *storch.sampling.Method*'s.

set_mc_weighting_function(*new_weighting_func: Callable[[StochasticTensor, Plate], Optional[Tensor]]*) → *SamplingMethod*

Override storch.Method specific weighting functions. This is called when initializing a storch.Method that has slightly different MC weighting semantics (for example, REBAR that weights some samples differently). This allows for compatibility of different *storch.Method*'s with different *storch.sampling.Method*'s.

training: bool

update_parameters(*result_triples: [storch.StochasticTensor, storch.CostTensor, torch.Tensor]*)

weighting_function(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

Weight by the size of the sample. Overload this if your sampling method uses some kind of weighting of the different events, like importance sampling or computing the expectation. If None is returned, it is assumed the samples are iid monte carlo samples.

This method is called from storch.method.Method.sample, and it is not needed to manually call this on created plates

9.1 Expectation

class storch.sampling.expect.**Enumerate**(*plate_name: str*, *budget=10000*)

Bases: `SamplingMethod`

sample(*distr: ~torch.distributions.distribution.Distribution, parents: [<class 'storch.tensor.Tensor'>], plates: [<class 'storch.tensor.Plate'>], requires_grad: bool) -> (<class 'storch.tensor.StochasticTensor'>, <class 'storch.tensor.Plate'>*)

training: bool

weighting_function(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

Weight by the size of the sample. Overload this if your sampling method uses some kind of weighting of the different events, like importance sampling or computing the expectation. If None is returned, it is assumed the samples are iid monte carlo samples.

This method is called from storch.method.Method.sample, and it is not needed to manually call this on created plates

9.2 Sequence decoding

class storch.sampling.seq.**AncestralPlate**(*name: str*, *n: int*, *parents: List[Plate]*, *variable_index: int*, *parent_plate:* AncestralPlate, *selected_samples: Optional[Tensor]*, *log_probs: Optional[Tensor]*, *weight: Optional[Tensor] = None*)

Bases: `Plate`

index_in(*plates: List[Plate]*) → int

is_in(*plates: Iterable[Plate]*) → bool

on_collecting_args(*plates: [storch.Plate]*) → bool

Filter the collected plates to only keep the AncestralPlates (with the same name) that has the highest variable index. :param plates: :return:

on_duplicate_plate(*plate: Plate*) → bool

on_unwrap_tensor(*tensor: Tensor*) → Tensor

Gets called whenever the given tensor is being unwrapped and unsqueezed for batch use. This method should not be called on tensors whose variable index is higher than this plates.

selected_samples is used to choose from the parent plates what is the previous element in the sequence. This is for example used in sampling without replacement. If set to None, it is assumed the different sequences are indexed by the plate dimension.

Parameters
`tensor` – The input tensor that is being unwrapped

Returns
The tensor that will be unwrapped and unsqueezed in the future. Can be a modification of the input tensor.

class storch.sampling.seq.**IterDecoding**(*plate_name*, *k*, *eos*)

Bases: *SequenceDecoding*

decode(*distr: Distribution*, *joint_log_probs: Optional[storch.Tensor]*, *parents: [storch.Tensor]*, *orig_distr_plates: [storch.Plate]*)

Decode given the input arguments :param distribution: The distribution to decode :param joint_log_probs: The log probabilities of the samples so far. prev_plates x amt_samples :param parents: List of parents of this tensor :param orig_distr_plates: List of plates from the distribution. Can include the self plate k. :return: 3-tuple of *storch.Tensor*. 1: The sampled value. 2: The new joint log probabilities of the samples. 3: How the samples index the parent samples. Can just be None if there is no choosing happening.

abstract decode_step(*indices: Tuple[int]*, *yv_log_probs: storch.Tensor*, *joint_log_probs: Optional[storch.Tensor]*, *sampled_support_indices: Optional[storch.Tensor]*, *parent_indexing: Optional[storch.Tensor]*, *is_conditional_sample: bool*, *amt_plates: int*, *amt_samples: int*)

Decode given the input arguments for a specific event :param indices: Tuple of integers indexing the current event to sample. :param yv_log_probs: Log probabilities of the different options for this event. distr_plates x k? x **|D_yv|** :param joint_log_probs: The log probabilities of the samples so far. None if *not is_conditional_sample*. prev_plates x amt_samples :param sampled_support_indices: Tensor of samples so far. None if this is the first set of indices. plates x k x events :param parent_indexing: Tensor indexing the parent sample. None if *not is_conditional_sample*. :param is_conditional_sample: True if a parent has already been sampled. This means the plates are more complex! :param amt_plates: The total amount of plates in both the distribution and the previously sampled variables :param amt_samples: The amount of active samples. :return: 3-tuple of *storch.Tensor*. 1: sampled_support_indices, with *:, indices* referring to the indices for the support. 2: The updated *joint_log_probs* of the samples. 3: The updated *parent_indexing*. How the samples index the parent samples. Can just return parent_indexing if nothing happens. 4: The amount of active samples after this step.

training: bool

class storch.sampling.seq.**MCDecoder**(*plate_name: str*, *k: int*, *eos: None*)

Bases: *SequenceDecoding*

decode(*distribution: Distribution*, *joint_log_probs: Optional[storch.Tensor]*, *parents: [storch.Tensor]*, *orig_distr_plates: [storch.Plate]*)

Decode given the input arguments :param distribution: The distribution to decode :param joint_log_probs: The log probabilities of the samples so far. prev_plates x amt_samples :param parents: List of parents of this tensor :param orig_distr_plates: List of plates from the distribution. Can include the self plate k. :return: 3-tuple of *storch.Tensor*. 1: The sampled value. 2: The new joint log probabilities of the samples. 3: How the samples index the parent samples. Can just be a range if there is no choosing happening. For all of these, the last plate index should be the plate index, with the other plates like *all_plates*

training: bool

class storch.sampling.seq.**SequenceDecoding**(*plate_name: str*, *k: int*, *eos: None*)

Bases: `SamplingMethod`

Methods for generating sequences of discrete random variables. Examples: Simple ancestral sampling with replacement, beam search, Stochastic beam search (sampling without replacement)

EPS = 1e-08

all_finished() → bool

create_plate(*plate_size: int*, *plates: [storch.Plate]*) → AncestralPlate

abstract decode(*distribution: Distribution*, *joint_log_probs: Optional[storch.Tensor]*, *parents: [storch.Tensor]*, *orig_distr_plates: [storch.Plate]*)

Decode given the input arguments :param distribution: The distribution to decode :param joint_log_probs: The log probabilities of the samples so far. prev_plates x amt_samples :param parents: List of parents of this tensor :param orig_distr_plates: List of plates from the distribution. Can include the self plate k. :return: 3-tuple of *storch.Tensor*. 1: The sampled value. 2: The new joint log probabilities of the samples. 3: How the samples index the parent samples. Can just be a range if there is no choosing happening. For all of these, the last plate index should be the plate index, with the other plates like *all_plates*

get_amt_finished() → Union[Tensor, Tensor]

get_sampled_seq(*finished: bool = False*) → [storch.StochasticTensor]

get_unique_seqs()

reset()

sample(*distr: Distribution*, *parents: [storch.Tensor]*, *orig_distr_plates: [storch.Plate]*, *requires_grad: bool*)

Sample from the distribution given the sequence so far. :param distribution: The distribution to sample from :return:

training: bool

weighting_function(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

Weight by the size of the sample. Overload this if your sampling method uses some kind of weighting of the different events, like importance sampling or computing the expectation. If None is returned, it is assumed the samples are iid monte carlo samples.

This method is called from storch.method.Method.sample, and it is not needed to manually call this on created plates

storch.sampling.seq.**expand_with_ignore_as**(*tensor*, *expand_as*, *ignore_dim: Union[str, int]*) → Tensor

Expands the tensor like expand_as, but ignores a single dimension. Ie, if tensor is of size a x b, expand_as of size d x a x c and dim=-1, then the return will be of size d x a x b. It also automatically expands all plate dimensions correctly. :param ignore_dim: Can be a string referring to the plate dimension

storch.sampling.seq.**left_expand_as**(*tensor*, *expand_as*)

storch.sampling.seq.**right_expand_as**(*tensor*, *expand_as*)

9.3 Sampling without replacement

class `storch.sampling.swor.`**`SampleWithoutReplacement`**(*plate_name: str*, *k: int*, *biased_iw: bool = False*, *eos=None*)

Bases: *IterDecoding*

Sampling method for sampling without replacement from (sequences of) discrete distributions. Implements Stochastic Beam Search https://arxiv.org/abs/1903.06059 with the weighting as defined by REINFORCE without replacement https://openreview.net/forum?id=r1lgTGL5DE

`EPS = 1e-08`

`compute_iw`(*plate: AncestralPlate*, *biased: bool*)

`create_plate`(*plate_size: int*, *plates: [storch.Plate]*) → *AncestralPlate*

`decode_step`(*indices: Tuple[int]*, *yv_log_probs: storch.Tensor*, *joint_log_probs: Optional[storch.Tensor]*, *sampled_support_indices: Optional[storch.Tensor]*, *parent_indexing: Optional[storch.Tensor]*, *is_conditional_sample: bool*, *amt_plates: int*, *amt_samples: int*)

Decode given the input arguments for a specific event using stochastic beam search. :param indices: Tuple of integers indexing the current event to sample. :param yv_log_probs: Log probabilities of the different options for this event. distr_plates x k? x **|D_yv|** :param joint_log_probs: The log probabilities of the samples so far. None if *not is_conditional_sample*. prev_plates x amt_samples :param sampled_support_indices: Tensor of samples so far. None if this is the first set of indices. plates x k x events :param parent_indexing: Tensor indexing the parent sample. None if *not is_conditional_sample*. :param is_conditional_sample: True if a parent has already been sampled. This means the plates are more complex! :param amt_plates: The total amount of plates in both the distribution and the previously sampled variables :param amt_samples: The amount of active samples. :return: 3-tuple of *storch.Tensor*. 1: sampled_support_indices, with *:, indices* referring to the indices for the support. 2: The updated *joint_log_probs* of the samples. 3: The updated *parent_indexing*. How the samples index the parent samples. Can just return parent_indexing if nothing happens. 4: The amount of active samples after this step.

`on_plate_already_present`(*plate: Plate*)

`perturbed_log_probs: Optional[storch.Tensor] = None`

`reset`()

`select_samples`(*perturbed_log_probs: storch.Tensor*, *joint_log_probs: storch.Tensor*)

Given the perturbed log probabilities and the joint log probabilities of the new options, select which one to use for the sample. :param perturbed_log_probs: plates x (k? * **|D_yv|**). Perturbed log-probabilities. k is present if first_sample. :param joint_log_probs: plates x (k? * **|D_yv|**). Joint log probabilities of the options. k is present if first_sample. :param first_sample: :return: perturbed log probs of chosen samples, joint log probs of chosen samples, index of chosen samples

`set_mc_sample`(*new_sample_func: Callable[[Distribution, [storch.Tensor], [storch.Plate], int], torch.Tensor]*) → *SamplingMethod*

Override storch.Method specific sampling functions. This is called when initializing a storch.Method that has slightly different MC sampling semantics (for example, reparameterization instead of normal sampling). This allows for compatibility of different *storch.Method*'s with different *storch.sampling.Method*'s.

`weighting_function`(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

Weight by the size of the sample. Overload this if your sampling method uses some kind of weighting of the different events, like importance sampling or computing the expectation. If None is returned, it is assumed the samples are iid monte carlo samples.

This method is called from storch.method.Method.sample, and it is not needed to manually call this on created plates

class storch.sampling.swor.**SumAndSample**(*plate_name: str*, *sum_size: int*, *sample_size: int = 1*, *without_replacement: bool = False*, *eos=None*)

Bases: `SampleWithoutReplacement`

Sums over S probable samples according to beam search and K sampled values that are not in the probable samples, then normalizes them accordingly.

select_samples(*perturbed_log_probs: storch.Tensor*, *joint_log_probs: storch.Tensor*)

Given the perturbed log probabilities and the joint log probabilities of the new options, select which one to use for the sample. :param perturbed_log_probs: plates x (k? * |D_yv|). Perturbed log-probabilities. k is present if first_sample. :param joint_log_probs: plates x (k? * |D_yv|). Joint log probabilities of the options. k is present if first_sample. :param first_sample: :return: perturbed log probs of chosen samples, joint log probs of chosen samples, index of chosen samples

training: bool

storch.sampling.swor.**cond_gumbel_sample**(*all_joint_log_probs*, *perturbed_log_probs*) → Tensor

storch.sampling.swor.**log1mexp**(*a: Tensor*) → Tensor

See appendix A of http://jmlr.org/papers/v21/19-985.html. Numerically stable implementation of log(1-exp(a))

9.4 Unordered set sampling

class storch.sampling.unordered_set.**GumbelSoftmaxWOR**(*plate_name: str*, *k: int*, *initial_temperature=1.0*, *min_temperature=0.0001*, *annealing_rate=1e-05*, *eos=None*)

Bases: `UnorderedSet`

sample(*distr: ~torch.distributions.distribution.Distribution, parents: [<class 'storch.tensor.Tensor'>], orig_distr_plates: [<class 'storch.tensor.Plate'>], requires_grad: bool) -> (<class 'torch.Tensor'>, <class 'storch.tensor.Plate'>*)

Sample from the distribution given the sequence so far. :param distribution: The distribution to sample from :return:

training: bool

class storch.sampling.unordered_set.**UnorderedSet**(*plate_name: str*, *k: int*, *comp_leave_two_out: bool = False*, *exact_integration: bool = False*, *num_int_points: int = 1000*, *a: float = 5.0*, *eos=None*)

Bases: `SampleWithoutReplacement`

training: bool

weighting_function(*tensor: StochasticTensor*, *plate: Plate*) → Optional[Tensor]

Weight by the size of the sample. Overload this if your sampling method uses some kind of weighting of the different events, like importance sampling or computing the expectation. If None is returned, it is assumed the samples are iid monte carlo samples.

This method is called from storch.method.Method.sample, and it is not needed to manually call this on created plates

CHAPTER

TEN

STORCH PACKAGE

10.1 Inference

10.2 Wrappers

`storch.wrappers.`**`deterministic`**(*fn: Optional[Callable] = None*, ***kwargs*)

Wraps the input function around a deterministic storch wrapper. This wrapper unwraps `Tensor` objects to `Tensor` objects, aligning the tensors according to the plates, then runs *fn* on the unwrapped Tensors.

Parameters

- **`fn`** – Optional function to wrap. If None, this returns another wrapper that accepts a function that will be instantiated
- **`kwargs.`** (*`by the given`*) –
- **`unwrap`** – Set to False to prevent unwrapping `Tensor` objects.
- **`fn_args`** – List of non-keyword arguments to the wrapped function
- **`fn_kwargs`** – Dictionary of keyword arguments to the wrapped function
- **`unwrap`** – Whether to unwrap the arguments to their torch.Tensor counterpart (default: True)
- **`align_tensors`** – Whether to automatically align the input arguments (default: True)
- **`l_broadcast`** – Whether to automatically left-broadcast (default: True)
- **`expand_plates`** – Instead of adding singleton dimensions on non-existent plates, this will
- **`(default`** (*`Note that outputs are unflattened automatically.`*) – False) flatten_plates sets this to True automatically.
- **`flatten_plates`** – Flattens the plate dimensions into a single batch dimension if set to true.
- **`dimension.`** (*`This can be useful for functions that are written to only work for tensors with a single batch`*) –
- **`(default`** – False)
- **`dim`** – Replaces the dim input in fn_kwargs by the plate dimension corresponding to the given string (optional)
- **`dims`** – Replaces the dims input in fn_kwargs by the plate dimensions corresponding to the given strings (optional)
- **`self_wrapper`** – storch.Tensor that wraps a

Returns
The wrapped function *fn*.

Return type
Callable

`storch.wrappers.`**`ignore_wrapping`**`()`

`storch.wrappers.`**`is_iterable`**`(`*a: Any*`)`

`storch.wrappers.`**`make_left_broadcastable`**`(`*fn: Optional[Callable]*`)`

Deterministic wrapper that is compatible with functions that are not by themselves left-broadcastable, such as `torch.nn.Conv2d()`. This function is on (N, C, H, W) and cannot deal with additional 'independent' dimensions on the left. To fix this, use *make_left_broadcastable(Conv2d(16, 33, 3))*

`storch.wrappers.`**`reduce`**`(`*fn*, *plates: Union[str, List[str]]*`)`

Wraps the input function around a deterministic storch wrapper. This wrapper unwraps `Tensor` objects to `Tensor` objects, aligning the tensors according to the plates, then runs *fn* on the unwrapped Tensors. It will reduce the plates given by *plates*.

Parameters
fn (*Callable*) – Function to wrap.

Returns
The wrapped function *fn*.

Return type
Callable

`storch.wrappers.`**`stochastic`**`(`*fn*`)`

Applies *fn* to the *inputs*. *fn* should return one or multiple *storch.Tensor`s. `fn* should not call *storch.stochastic* or *storch.deterministic*. *inputs* can include `storch.Tensor`s.

Parameters
fn –

Returns

10.3 Exceptions

exception `storch.exceptions.`**`IllegalStorchExposeError`**`(`*message*`)`

Bases: `Exception`

10.4 Unique

This module is highly experimental

10.5 Utilities

`storch.storch.cat(*args, **kwargs)`

Version of `torch.cat()` that is compatible with `storch.Tensor`. Required because `torch.Tensor.__torch_function__()` is not properly implemented for `torch.cat()`: https://github.com/pytorch/pytorch/issues/34294

`storch.storch.conditional_gumbel_rsample(hard_sample: Tensor, probs: Tensor, bernoulli: bool, temperature) → Tensor`

Conditionally re-samples from the distribution given the hard sample. This samples z sim p(z|b), where b is the hard sample and p(z) is a gumbel distribution.

`storch.storch.expand_as(tensor: Union[Tensor, Tensor], expand_as: Union[Tensor, Tensor]) → Union[Tensor, Tensor]`

`storch.storch.gather(input: Tensor, dim: str, index: Tensor)`

`storch.storch.grad(outputs, inputs, grad_outputs=None, retain_graph: Optional[bool] = None, create_graph: bool = False, only_inputs: bool = True, allow_unused: bool = False) → Tuple[Tensor, ...]`

Helper method for computing torch.autograd.grad on storch tensors. Returns storch Tensors as well.

`storch.storch.logsumexp(tensor: Tensor, dims: Union[List[Union[str, int, Plate]], str, int, Plate]) → Tensor`

`storch.storch.mean(tensor: Tensor, dims: Union[List[Union[str, int, Plate]], str, int, Plate]) → Tensor`

Simply takes the mean of the tensor over the dimensions given. WARNING: This does NOT weight the different elements according to the plates. You will very likely want to call the reduce_plates method instead.

`storch.storch.order_plates(plates: [<class 'storch.tensor.Plate'>], reverse=False)`

Topologically order the given plates. Uses Kahn's algorithm.

`storch.storch.reduce_plates(tensor: Union[Tensor, Tensor], plates: Optional[Union[List[Union[Plate, str]], Plate, str]] = None, detach_weights=True) → Tensor`

Reduce the tensor along the given plates. This takes into account how different samples are weighted, and should nearly always be used instead of reducing plate dimensions using the mean or the sum. By default, this reduces all plates.

Parameters

- **`tensor`** – Tensor to reduce
- **`plates`** – Plates to reduce. If None, this reduces all plates (default). Can be a string, Plate, or list of string
- **`Plates.`** (*and*) –
- **`detach_weights`** – Whether to detach the weighting of the samples from the graph

Returns

The reduced tensor

`storch.storch.sum(tensor: Tensor, dims: Union[List[Union[str, int, Plate]], str, int, Plate]) → Tensor`

Simply sums the tensor over the dimensions given. WARNING: This does NOT weight the different elements according to the plates. You will very likely want to call the reduce_plates method instead.

`storch.storch.variance(tensor: Union[Tensor, Tensor], variance_plate: Union[Plate, str], plates: Optional[Union[List[Union[Plate, str]], Plate, str]] = None, detach_weights=True) → Tensor`

Compute the variance of the tensor along the plate dimensions. This takes into account how different samples are weighted.

Parameters

- **tensor** – Tensor to compute variance over
- **plates** – Plates to reduce.
- **detach_weights** – Whether to detach the weighting of the samples from the graph

Returns
The variance of the tensor.

storch.util.**get_distr_parameters**(*d: Distribution*, *filter_requires_grad=True*) → Dict[str, Tensor]

storch.util.**has_backwards_path**(*output: ~storch.tensor.Tensor, inputs: [<class 'storch.tensor.Tensor'>]*) → [<class 'bool'>]

Returns true for each individual input if the gradient functions of the torch.Tensor underlying output is connected to the input tensor. This is only run once to compute the possibility of links between two storch.Tensor's. The result is saved into the parent links on storch.Tensor's. :param output: :param input: :param depth_first: Initialized to False as we are usually doing this only for small distances between tensors. :return:

storch.util.**has_differentiable_path**(*output: Tensor*, *input: Tensor*)

storch.util.**magic_box**(*l: Tensor*)

Implements the MagicBox operator from DiCE: The Infinitely Differentiable Monte-Carlo Estimator https://arxiv.org/abs/1802.05098 It returns 1 in the forward pass, but returns magic_box(l) cdot r in the backwards pass. This allows for any-order gradient estimation.

storch.util.**print_graph**(*costs: [<class 'storch.tensor.CostTensor'>]*)

storch.util.**reduce_mean**(*tensor: ~torch.Tensor, keep_dims: [<class 'int'>]*)

storch.util.**rsample_gumbel**(*distr: Distribution*, *n: int*) → Tensor

storch.util.**rsample_gumbel_softmax**(*distr: Distribution*, *n: int*, *temperature: Tensor*, *straight_through: bool = False*) → Tensor

storch.util.**split**(*tensor: Tensor*, *plate: Plate*, ***, *amt_slices: Optional[int] = None*, *slices: Optional[List[slice]] = None*, *create_plates=True*) → Tuple[Tensor, ...]

Splits the plate dimension on the tensor into several tensors and returns those tensors. Note: It removes the tensors from the computation graph and therefore should only be used when creating estimators, when logging or debugging, or if you know what you're doing.

storch.util.**tensor_stats**(*tensor: Tensor*)

storch.util.**topological_sort**(*costs: [<class 'storch.tensor.CostTensor'>]*) → [<class 'storch.tensor.Tensor'>]

Implements reverse kahn's algorithm :param costs: :return:

storch.util.**walk_backward_graph**(*tensor: Tensor*) → Iterable[Tensor]

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[A1] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[A2] Yuntian Deng, Yoon Kim, Justin Chiu, Demi Guo, and Alexander Rush. Latent alignment and variational attention. In *Advances in Neural Information Processing Systems*, 9712–9724. 2018.

[A3] Paul Glasserman and Yu-Chi Ho. *Gradient estimation via perturbation analysis*. Volume 116. Springer Science & Business Media, 1991.

[A4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[A5] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[A6] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[A7] Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. 2020. arXiv:2005.11401.

[A8] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, 9994–10006. 2018.

[A9] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[A10] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: a continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

[A11] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *arXiv preprint arXiv:1906.10652*, 2019.

[A12] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*. 2015. arXiv:1506.05254.

[A13] Théophane Weber, Nicolas Heess, Lars Buesing, and David Silver. Credit assignment techniques in stochastic computation graphs. *arXiv preprint arXiv:1901.01761*, 2019.

[B1] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[B2] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[B3] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[B4] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: a continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

PYTHON MODULE INDEX

INDEX