In the name of Allah

Physics-Informed Neural Network: Notes

By:

Ehsan Ghaderi

Adviser:

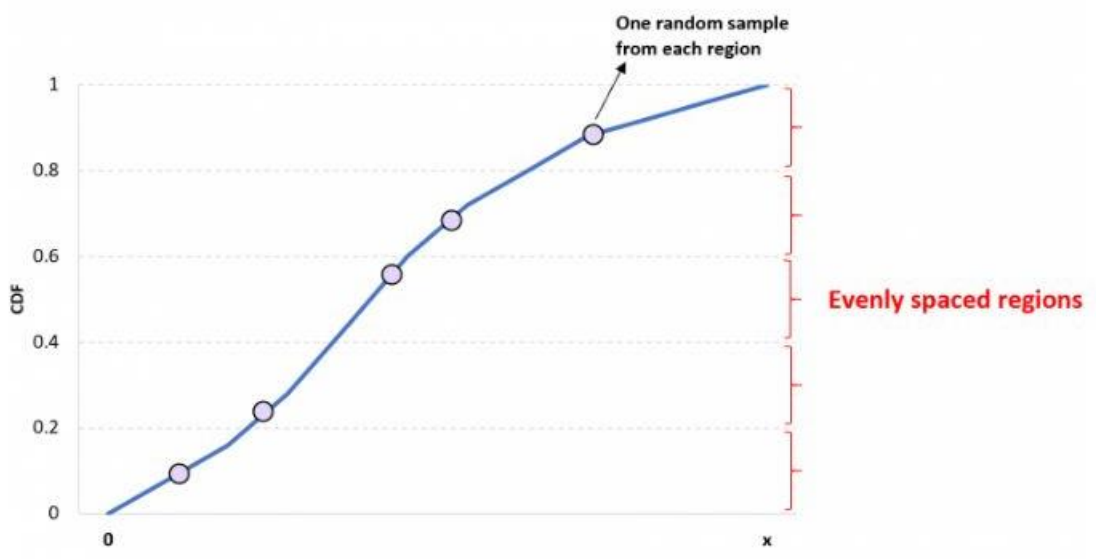
Dr. Bijarchi

October 18,2023

Latin Hypercube Sampling (LHS)

The idea behind one-dimensional latin hypercube sampling is simple:

Divide a given CDF into *n* different regions and randomly choose one value from each region to obtain a sample of size *n*.

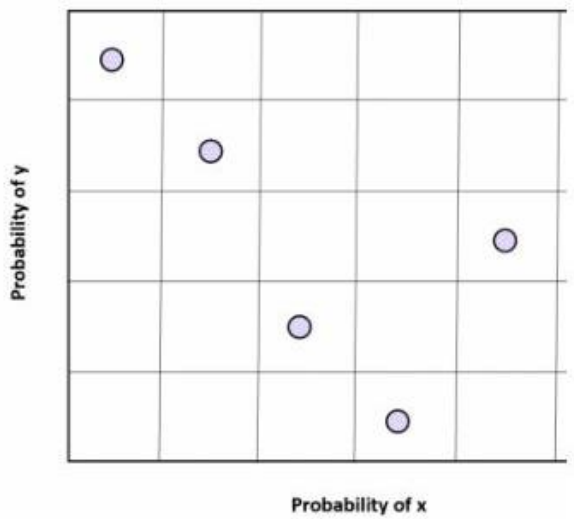


The benefit of this approach is that it ensures that at least one value from each region is included in the sample.

Latin Hypercube Sampling (LHS)

We can easily extend the idea of one-dimensional latin hypercube sampling into two dimensions as well.

For two variables, x and y, we can divide the sample space of each variable into *n* evenly spaced regions and pick a random sample from each sample space to obtain random values across two dimensions.



LHS

```
def lhs(n, samples=None, criterion=None, iterations=None):
    """
    Generate a latin-hypercube design

    Parameters
    ----------
    n : int
        The number of factors to generate samples for

    Optional
    --------
    samples : int
        The number of samples to generate for each factor (Default: n)
    criterion : str
        Allowable values are "center" or "c", "maximin" or "m",
        "centermaximin" or "cm", and "correlation" or "corr". If no value
        given, the design is simply randomized.
    iterations : int
        The number of iterations in the maximin and correlations algorithms
        (Default: 5).
```

```
>>> lhs(2, samples=5, criterion='center')
array([[ 0.3,  0.5],
       [ 0.7,  0.9],
       [ 0.1,  0.3],
       [ 0.9,  0.1],
       [ 0.5,  0.7]])
```

Initialization

- Xavier Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

- He Normal initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in}}}$$

- Xavier Uniform initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

- He Uniform initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}})$$

Xavier initialization

```
def xavier_normal_(tensor: Tensor, gain: float = 1.) -> Tensor:
    r"""Fills the input `Tensor` with values according to the method
    described in `Understanding the difficulty of training deep feedforward
    neural networks` - Glorot, X. & Bengio, Y. (2010), using a normal
    distribution. The resulting tensor will have values sampled from
    :math:`\mathcal{N}(0, \text{std}^2)` where

    .. math::
        \text{std} = \text{gain} \times \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}

    Also known as Glorot initialization.

    Args:
        tensor: an n-dimensional `torch.Tensor`
        gain: an optional scaling factor

    Examples:
        >>> w = torch.empty(3, 5)
        >>> nn.init.xavier_normal_(w)
    """
    fan_in, fan_out = _calculate_fan_in_and_fan_out(tensor)
    std = gain * math.sqrt(2.0 / float(fan_in + fan_out))

    return _no_grad_normal_(tensor, 0., std)
```

Weights are taken from the normal distribution whose mean is equal to zero and standard deviation equal to **sqrt(2/fan_in+ fan_out);** fan_out is the number of outputs from the neuron.

Optimization algorithms

L-BFGS	ADAM
$x_{k+1} = x_k + \alpha_k d_k$	$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k$
$s_k = x_{k+1} - x_k$	$v_k = \beta_2 v_{k-1} + (1 - \beta_2) g_k^2$
$y_k = g_{k+1} - g_k$	$m'_k = \frac{m_k}{1-\beta_1^k}$
$\alpha_k = \frac{s_k^T y_k}{y_k^T H_k y_k}$	$v'_k = \frac{v_k}{1-\beta_2^k}$
$x_{k+1} = x_k + \alpha_k d_k$	$x_{k+1} = x_k - \frac{\eta}{\sqrt{v'_k}+\epsilon} m'_k$

Adam

$$m_w^{t+1} = \beta_1 m_w^t + (1 - \beta_1)\nabla_w L^t$$
$$v_w^{t+1} = \beta_2 v_w^t + (1 - \beta_2)(\nabla_w L^t)^2$$

$$\hat{m}_w = \frac{m_w^{t+1}}{1 - \beta_1^t}$$
$$\hat{v}_w = \frac{v_w^{t+1}}{1 - \beta_2^t}$$
$$w^{t+1} = w^t - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} - \epsilon}$$

where $w(t)$ are the model parameters, L^t is the loss function, t is the current training iteration, β_1 and β_2 are the forgetting factors for gradients and second-order gradient moments, respectively.

Adam

```
def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
             weight_decay=0, amsgrad=False):

    Args:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-3)
        betas (Tuple[float, float], optional): coefficients used for computing
            running averages of gradient and its square (default: (0.9, 0.999))
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        amsgrad (boolean, optional): whether to use the AMSGrad variant of this
            algorithm from the paper `On the Convergence of Adam and Beyond`_
            (default: False)
```

Algorithm 1 L-BFGS

for $i = 1, 2, \ldots, n$ **do**

Obtain a direction P_k by solving $B_k P_k = -\nabla f(X_k)$

Perform a one-dimensional optimization to find an acceptable stepsize α_k in the direction found in the first step. In an exact line search, $\alpha_k = \operatorname{argmin} f(X_k + \alpha P_k)$.
In practice, an inexact line search usually suffices, with acceptable α_k satisfying the Wolfe conditions.

Set $S_k = \alpha_k P_k$ and update $X_{k+1} = X_k + S_k$

$y_k = \nabla f(X_{k+1}) - \nabla f(X_k)$

$B_{k+1} = B_k + \frac{Y_k Y_k^T}{s_k Y_k^T} + \frac{B_k s_k s_k^T B_k^T}{B_k s_k s_k^T}$

end for

where X_k are the model parameters, f is the loss function, B_k is an approximation of the Hessian matrix, and S_k and Y_k are the differences between the current and previous values of X and $\nabla f(X)$, respectively. The L-BFGS and ADAM optimization algorithms are both commonly

L-BFGS

```
def __init__(self,
             params,
             lr=1,
             max_iter=20,
             max_eval=None,
             tolerance_grad=1e-7,
             tolerance_change=1e-9,
             history_size=100,
             line_search_fn=None):
```

```
    Args:
        lr (float): learning rate (default: 1)
        max_iter (int): maximal number of iterations per optimization step
            (default: 20)
        max_eval (int): maximal number of function evaluations per optimization
            step (default: max_iter * 1.25).
        tolerance_grad (float): termination tolerance on first order optimality
            (default: 1e-5).
        tolerance_change (float): termination tolerance on function
            value/parameter changes (default: 1e-9).
        history_size (int): update history size (default: 100).
        line_search_fn (str): either 'strong_wolfe' or None (default: None).
    """
```