Neural Splines for Resource–Constrained Deep Learning

Prepared for the Neural Splines Project

August 1, 2025

Abstract

This technical note accompanies the *Neural Splines* repository. It distils the key theoretical ideas behind spline–based weight compression and demonstrates how those ideas are realized in code. The goal is to provide researchers from diverse backgrounds with a concise yet comprehensive exposition. Readers are encouraged to explore the repository for full implementations and examples.

Contents

1 Introduction

Neural networks are powerful function approximators, but their dense parameterization can be a liability on devices with limited memory or compute. Splines offer an elegant solution: by representing the weight matrix with a grid of *control points* and reconstructing the full matrix via interpolation, we dramatically reduce the number of trainable parameters while maintaining expressivity. This technique is particularly appealing for scenarios where models must run on edge devices or in environments with intermittent connectivity.

The original paper (see the attached Markdown document) situates neural splines within a broad philosophical framework. Here we focus on the practical aspects: defining spline layers, training a simple classifier on the MNIST dataset, densifying the resulting network and running inference with standard PyTorch. All components in this repository are implemented using only the core PyTorch library.

2 Spline–Based Compression

Consider a fully–connected layer with input dimension d_{in} and output dimension d_{out}. A standard implementation stores a weight matrix $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ and a bias vector $b \in \mathbb{R}^{d_{\text{out}}}$, giving $d_{\text{out}} \times d_{\text{in}} + d_{\text{out}}$ parameters. When d_{in} and d_{out} are large, this representation becomes cumbersome.

A spline layer stores instead a much smaller grid of control points. For example, we can hold a $N \times N$ grid $\{c_{i,j}\}\_0 \leq i, j < N$ for the weight matrix and a vector of N control points for the bias. To compute the weight at arbitrary indices (u, v), we map the indices to a normalized coordinate in $[0, 1]$ and apply bicubic interpolation. Formally, for u in $\{0, \ldots, d_{\text{out}} - 1\}$ and v in $\{0, \ldots, d_{\text{in}} - 1\}$, let

$$t_u = \frac{u}{d_{\text{out}} - 1}, \qquad t_v = \frac{v}{d_{\text{in}} - 1}.$$

Mapping t_u and t_v into the control–point grid yields indices $i = t_u\,(N - 3)$, $j = t_v\,(N - 3)$, with fractional parts $\alpha, \beta \in [0, 1]$. The weight at (u, v) is then computed via a weighted sum of the sixteen neighbouring control points with cubic B–spline basis functions:

$$W_{u,v} = \sum_{p=0}^{3} \sum_{q=0}^{3} B_p(\alpha)\, B_q(\beta)\, c_{i+p,j+q},$$

where $B_0, \ldots, B_3$ are the cubic basis functions. The bias term is obtained similarly from a one–dimensional spline over its control points. This mechanism enforces smoothness across the virtual weight matrix and yields an enormous compression ratio: in our experiments we replace tens of thousands of parameters with fewer than 100 control values.

2.1 Control Point Optimisation

The control points themselves are learned via gradient descent. During training we compute virtual weights and biases on the fly, perform the forward and backward passes, and update the control points. The interpolation introduces a modest computational overhead but pays dividends in memory savings. Once training is complete, we can *densify* the network by explicitly evaluating the spline on every index and storing the resulting dense tensors.

3 Reference Implementation

The repository provides a reference implementation in `src/neural_spline.py`. Below we present the core of the spline–linear layer, focusing on clarity rather than performance.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SplineLinear(nn.Module):
    def __init__(self, in_features: int, out_features: int, num_control: int =
        6):
```

```
        super().__init__()
        # 2D control points for weights and 1D for bias
        self.weight_cp = nn.Parameter(torch.randn(num_control, num_control))
        self.bias_cp = nn.Parameter(torch.randn(num_control))
        self.in_features = in_features
        self.out_features = out_features
        self.num_control = num_control

    def interp2d(self, u_idx: torch.Tensor, v_idx: torch.Tensor) -> torch.
        Tensor:
        """Perform bicubic interpolation at discrete positions."""
        # Normalize to [0,1]
        t_u = u_idx.float() / (self.out_features - 1)
        t_v = v_idx.float() / (self.in_features - 1)
        # Map to control grid coordinates
        coords_u = t_u * (self.num_control - 3)
        coords_v = t_v * (self.num_control - 3)
        i = coords_u.floor().clamp(0, self.num_control - 4).long()
        j = coords_v.floor().clamp(0, self.num_control - 4).long()
        alpha = (coords_u - i).unsqueeze(-1)
        beta = (coords_v - j).unsqueeze(-1)
        # Precompute basis functions B0-B3 for alpha and beta
        def B0(x): return ((1 - x) ** 3) / 6
        def B1(x): return (3*x**3 - 6*x**2 + 4) / 6
        def B2(x): return (-3*x**3 + 3*x**2 + 3*x + 1) / 6
        def B3(x): return (x ** 3) / 6
        Bs = [B0, B1, B2, B3]
        weight = 0
        # Sum over neighbourhood
        for p in range(4):
            for q in range(4):
                b_u = Bs[p](alpha)
                b_v = Bs[q](beta)
                cp = self.weight_cp[i + p, j + q]
                weight += (b_u * b_v) * cp
        return weight

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Create index grids and interpolate
        idx_out = torch.arange(self.out_features, device=x.device)
        idx_in = torch.arange(self.in_features, device=x.device)
        W = self.interp2d(idx_out[:, None], idx_in[None, :])
        # Bias via 1D cubic interpolation (similar logic)
        t = idx_out.float() / (self.out_features - 1)
        coords = t * (self.num_control - 3)
        k = coords.floor().clamp(0, self.num_control - 4).long()
        u = coords - k
        b = (
            ((1 - u)**3) / 6 * self.bias_cp[k] +
            ( (3*u**3 - 6*u**2 + 4) / 6) * self.bias_cp[k+1] +
            ((-3*u**3 + 3*u**2 + 3*u + 1) / 6) * self.bias_cp[k+2] +
```

```
            (u**3 / 6) * self.bias_cp[k+3]
        )
        return x @ W.t() + b

    def to_dense(self) -> nn.Linear:
        """Convert control points to a standard dense linear layer."""
        with torch.no_grad():
            idx_out = torch.arange(self.out_features)
            idx_in = torch.arange(self.in_features)
            W = self.interp2d(idx_out[:, None], idx_in[None, :])
            t = idx_out.float() / (self.out_features - 1)
            coords = t * (self.num_control - 3)
            k = coords.floor().clamp(0, self.num_control - 4).long()
            u = coords - k
            b = (
                ((1 - u)**3) / 6 * self.bias_cp[k] +
                ( (3*u**3 - 6*u**2 + 4) / 6) * self.bias_cp[k+1] +
                ((-3*u**3 + 3*u**2 + 3*u + 1) / 6) * self.bias_cp[k+2] +
                (u**3 / 6) * self.bias_cp[k+3]
            )
        dense = nn.Linear(self.in_features, self.out_features)
        dense.weight.copy_(W)
        dense.bias.copy_(b)
        return dense
```

Listing 1: Simplified implementation of a spline-based linear layer.

The ‘SplineLinear‘ class defines a weight grid and a bias spline. The `forward` method synthesizes dense weights and bias on demand, performs a matrix multiplication, and adds the bias. The ‘to_dense‘ function evaluates the interpolants once and packs the results into a conventional ‘nn.Linear‘, which can then be used in place of the spline layer during inference.

4 Training on MNIST

The script ‘src/train.py‘ trains a simple two–layer network on the MNIST digit dataset. The first layer uses spline compression; the second is a standard linear classifier. During training we report both the spline model's accuracy and the accuracy of the densified model obtained via `to_dense`. A typical invocation is

```
python3 src/train.py \
  --epochs 5 --batch-size 128 --cp 6 --hidden-size 256
```

Listing 2: Training the spline network.

On a modern CPU the training completes in a few minutes and achieves around 96 accuracy. The script writes checkpoint files for both the spline and densified models.

5 Densification and Inference

After training we often wish to deploy the network on systems where on-the-fly interpolation is undesirable. To this end we convert each spline layer into a fixed dense layer

using the ‘to_dense‘ method and assemble a dense multilayer perceptron. This densified model no longer references control points; its weights and biases are plain tensors and can be executed with vanilla PyTorch. The repository contains ‘src/inference.py‘, which reconstructs a dense network, loads its state dictionary and computes the classification accuracy on the MNIST test set. A typical command line invocation is

```
python3 src/inference.py \
  --model-path checkpoints/dense_model.pth \
  --input-size 784 --hidden-size 256 --output-size 10
```

Listing 3: Running inference with the densified model.

If you trained with different hidden sizes or control point counts you should adjust the arguments accordingly.

6 Conclusion

Neural splines provide an intuitive yet powerful framework for compressing neural networks without sacrificing performance. By replacing thousands of individual weights with a small set of interpolated control points, we obtain models that are compact, smooth, and efficient. The accompanying code and scripts in this project demonstrate how to implement, train and deploy such networks using PyTorch. We hope this concise exposition and the well–commented code will serve as a useful starting point for researchers exploring resource–constrained deep learning.