Gordon Research

10/30/24

```python
class KANLinear(torch.nn.Module):
    def __init__(
        self,
        in_features,
        out_features,
        grid_size=5,
        spline_order=3,
        scale_noise=0.1,
        scale_base=1.0,
        scale_spline=1.0,
        enable_standalone_scale_spline=True,
        base_activation=torch.nn.SiLU,
        grid_eps=0.02,
        grid_range=[-1, 1],
```

- KANLinear class is custom PyTorch layer designed to integrate linear transformations with kernel-based techniques
    - B-spline interpolation
- grid_size: Sets the resolution of the B-spline grid. Higher values allow finer detail in the spline representation.
- spline_order: Determines the order of the B-spline (how smooth the curves are). Common values are 2 or 3.
- scale_noise, scale_base, scale_spline: control scaling factors for noise, the base transformation, and spline transformation

- enable_standalone_scale_spline: A boolean flag indicating whether each spline connection has its unique scaling parameter
- base_activation: The activation function applied to the base layer, set to torch.nn.SiLU by default.
- grid_eps: An epsilon value used for fine-tuning the grid.
- grid_range: Specifies the range within which the grid operates, typically between -1 and 1.

Grid creation

- Computes the step size h to ensure the grid spans from grid_range[0] to grid_range[1].
- The grid tensor stores the grid points as a buffer (non-learnable parameter) for spline interpolation.

```
h = (grid_range[1] - grid_range[0]) / grid_size
grid = (
    (
        torch.arange(-spline_order, grid_size + spline_order + 1) * h
        + grid_range[0]
    )
    .expand(in_features, -1)
    .contiguous()
)
self.register_buffer("grid", grid)
```

```python
def reset_parameters(self):
    torch.nn.init.kaiming_uniform_(self.base_weight, a=math.sqrt(5) * self.scale_base)
    with torch.no_grad():
        noise = (
            (
                torch.rand(self.grid_size + 1, self.in_features, self.out_features)
                - 1 / 2
            )
            * self.scale_noise
            / self.grid_size
        )
        self.spline_weight.data.copy_(
            (self.scale_spline if not self.enable_standalone_scale_spline else 1.0)
            * self.curve2coeff(
                self.grid.T[self.spline_order : -self.spline_order],
                noise,
            )
        )
        if self.enable_standalone_scale_spline:
            # torch.nn.init.constant_(self.spline_scaler, self.scale_spline)
            torch.nn.init.kaiming_uniform_(self.spline_scaler, a=math.sqrt(5) * self.scale_spline)
```

```
torch.nn.init.kaiming_uniform_(self.base_weight, a=math.sqrt(5) * self.scale_base)
```

- Purpose: Initializes the base_weight matrix using Kaiming (He) uniform initialization, which is commonly used for layers with ReLU-based activations.
- Details: Kaiming initialization helps maintain stable variance throughout the network, which is crucial for effective learning. The a=math.sqrt(5) * self.scale_base scales the weight matrix giving control over the magnitude of the initialized weights.
-

```python
with torch.no_grad():
    noise = (
        (
            torch.rand(self.grid_size + 1, self.in_features, self.out_features) - 1 / 2
        )
        * self.scale_noise
        / self.grid_size
    )
```

- Purpose: Creates a tensor noise that will be added to the spline weights.
- Details: * self.scale_noise / self.grid_size: Scales the noise based on the scale_noise and grid_size parameters, controlling its magnitude

- Purpose: computes the B-spline basis functions for a given input tensor x. This function is essential for creating the spline transformation, allowing the layer to map inputs smoothly based on the spline basis.

```python
def b_splines(self, x: torch.Tensor):

    Args:
        x (torch.Tensor): Input tensor of shape (batch_size, in_features).

    Returns:
        torch.Tensor: B-spline bases tensor of shape (batch_size, in_features, grid_size + spline_order).
    """
    assert x.dim() == 2 and x.size(1) == self.in_features

    grid: torch.Tensor = (
        self.grid
    )  # (in_features, grid_size + 2 * spline_order + 1)
    x = x.unsqueeze(-1)
    bases = ((x >= grid[:, :-1]) & (x < grid[:, 1:])).to(x.dtype)
    for k in range(1, self.spline_order + 1):
        bases = (
            (x - grid[:, : -(k + 1)])
            / (grid[:, k:-1] - grid[:, : -(k + 1)])
            * bases[:, :, :-1]
        ) + (
            (grid[:, k + 1 :] - x)
            / (grid[:, k + 1 :] - grid[:, 1:(-k)])
            * bases[:, :, 1:]
        )

    assert bases.size() == (
        x.size(0),
        self.in_features,
        self.grid_size + self.spline_order,
    )
    return bases.contiguous()
```

- Loop for Higher Orders: The loop iteratively computes higher-order B-spline bases up to spline_order. B-splines are calculated recursively, where each order k depends on the previously computed order k-1.

```
for k in range(1, self.spline_order + 1):
    bases = (
        (x - grid[:, : -(k + 1)])
        / (grid[:, k:-1] - grid[:, : -(k + 1)])
        * bases[:, :, :-1]
    ) + (
        (grid[:, k + 1 :] - x)
        / (grid[:, k + 1 :] - grid[:, 1:(-k)])
        * bases[:, :, 1:]
    )
```

- computes the coefficients for B-spline interpolation, allowing the KANLinear layer to fit a curve that passes through a set of given points.
- function takes input points x and target output points y and computes the spline coefficients needed to produce an interpolated curve. These coefficients are returned as a tensor and stored in spline_weight in the reset_parameters method.

```python
def curve2coeff(self, x: torch.Tensor, y: torch.Tensor):
    """
    Compute the coefficients of the curve that interpolates the given points.

    Args:
        x (torch.Tensor): Input tensor of shape (batch_size, in_features).
        y (torch.Tensor): Output tensor of shape (batch_size, in_features, out_features).

    Returns:
        torch.Tensor: Coefficients tensor of shape (out_features, in_features, grid_size + spline_order).
    """
    assert x.dim() == 2 and x.size(1) == self.in_features
    assert y.size() == (x.size(0), self.in_features, self.out_features)

    A = self.b_splines(x).transpose(
        0, 1
    )  # (in_features, batch_size, grid_size + spline_order)
    B = y.transpose(0, 1)  # (in_features, batch_size, out_features)
    solution = torch.linalg.lstsq(
        A, B
    ).solution  # (in_features, grid_size + spline_order, out_features)
    result = solution.permute(
        2, 0, 1
    )  # (out_features, in_features, grid_size + spline_order)

    assert result.size() == (
        self.out_features,
        self.in_features,
        self.grid_size + self.spline_order,
    )
    return result.contiguous()
```

- The forward function applies both a base linear transformation and a spline-based transformation to the input tensor x. It combines these outputs to produce a final result, which enhances the model’s ability to capture complex patterns in the data.
-

```
def forward(self, x: torch.Tensor):
    assert x.size(-1) == self.in_features
    original_shape = x.shape
    x = x.reshape(-1, self.in_features)

    base_output = F.linear(self.base_activation(x), self.base_weight)
    spline_output = F.linear(
        self.b_splines(x).view(x.size(0), -1),
        self.scaled_spline_weight.view(self.out_features, -1),
    )
    output = base_output + spline_output

    output = output.reshape(*original_shape[:-1], self.out_features)
    return output
```

- Purpose:
    - Computes a new adaptive grid based on the sorted values of the input x
    - Blends the adaptive grid with a uniform grid for flexibility
    - Updates the spline weights to fit the new grid, recalculating the coefficients for B-spline interpolation

```python
def update_grid(self, x: torch.Tensor, margin=0.01):
    assert x.dim() == 2 and x.size(1) == self.in_features
    batch = x.size(0)

    splines = self.b_splines(x)  # (batch, in, coeff)
    splines = splines.permute(1, 0, 2)  # (in, batch, coeff)
    orig_coeff = self.scaled_spline_weight  # (out, in, coeff)
    orig_coeff = orig_coeff.permute(1, 2, 0)  # (in, coeff, out)
    unreduced_spline_output = torch.bmm(splines, orig_coeff)  # (in, batch, out)
    unreduced_spline_output = unreduced_spline_output.permute(
        1, 0, 2
    )  # (batch, in, out)

    # sort each channel individually to collect data distribution
    x_sorted = torch.sort(x, dim=0)[0]
    grid_adaptive = x_sorted[
        torch.linspace(
            0, batch - 1, self.grid_size + 1, dtype=torch.int64, device=x.device
        )
    ]

    uniform_step = (x_sorted[-1] - x_sorted[0] + 2 * margin) / self.grid_size
    grid_uniform = (
        torch.arange(
            self.grid_size + 1, dtype=torch.float32, device=x.device
        ).unsqueeze(1)
        * uniform_step
        + x_sorted[0]
        - margin
    )
```

Comparison of CNN and KAN on mnist (10 epochs)

CNN

Loss: 0.0267

Accuracy: 0.99

Training time: 4 minutes 10 seconds

KAN

Loss: 0.0894

Accuracy: 0.97

Training time: 4 minutes 5 seconds