KAN Kolmogorov Arnold Network Vision Transformer

Kolmogorov Arnold Network

相關參考資料

- 原始論文：KAN: Kolmogorov-Arnold Network
- 原始論文網址：https://arxiv.org/abs/2404.19756

BIBTEX 供論文引用

```
@article{liu2024kan,
  title={KAN: Kolmogorov-Arnold Networks},
  author={Liu, Ziming and Wang, Yixuan and Vaidya, Sachin and Ruehle, Fabian and Halverson, James and
Solja{\v{c}}i{\'c}, Marin and Hou, Thomas Y and Tegmark, Max},
  journal={arXiv preprint arXiv:2404.19756},
  year={2024}
}
```

論文內容

Summary

KAN（Kolmogorov-Arnold Network）論文基本概念。

1. Research Background
   - Kolmogorov-Arnold Representation Theorem（KART）：此定理說明任何多變量的連續函數都可以表示為單變量連續函數和加法操作的有限組合。
   - Limitations of Multi-Layer Perceptron（MLP）：傳統由 MLP 形成的神經網路雖然具有強大的表達能力，但在某些應用中存在固定的 Activation Function，使得其解釋性較差且參數效率低下。
2. Kolmogorov-Arnold Network（KAN）
   - Network Structure：與 MLP 不同，KAN 在 Edge（即權重）上使用可學習的 Activation Function，而不是在 Node（即神經元）上使用固定的 Activation Function。
   - Activation Function：KAN 中的每個權重參數被替換為一個參數化為樣條函數的單變量函數。節點只進行簡單的信號相加操作，不應用任何非線性操作。
3. Advantages of KAN
   - Higher Precision：KAN 在數據擬合和偏微分方程求解方面比 MLP 更準確。例如，在偏微分方程求解中，一個兩層且寬度為十的 KAN 比一個四層且寬度為一百的 MLP 準確度高百倍。
   - Explainability：KAN 可以直觀地可視化，並能與人類用戶進行互動，有助於科學家重新發現數學和物理定律。

Summary

KAN（Kolmogorov-Arnold Network）和 MLP（Multi-Layer Perceptron）之間的主要差異與比較。

1. 激活函數的位置和特性
   - MLP：激活函數固定，位於節點（神經元）上；且激活函數一般是非線性函數，例如 ReLU、Sigmoid 等。
   - KAN：激活函數是可學習的，位於邊（權重）上；且每個權重參數被替換為一個參數化為樣條函數的單變量函數。
2. 網路結構和權重表示
   - MLP：使用線性權重矩陣進行計算，然後應用固定的非線性激活函數；節點進行非線性變換。
   - 結構公式：

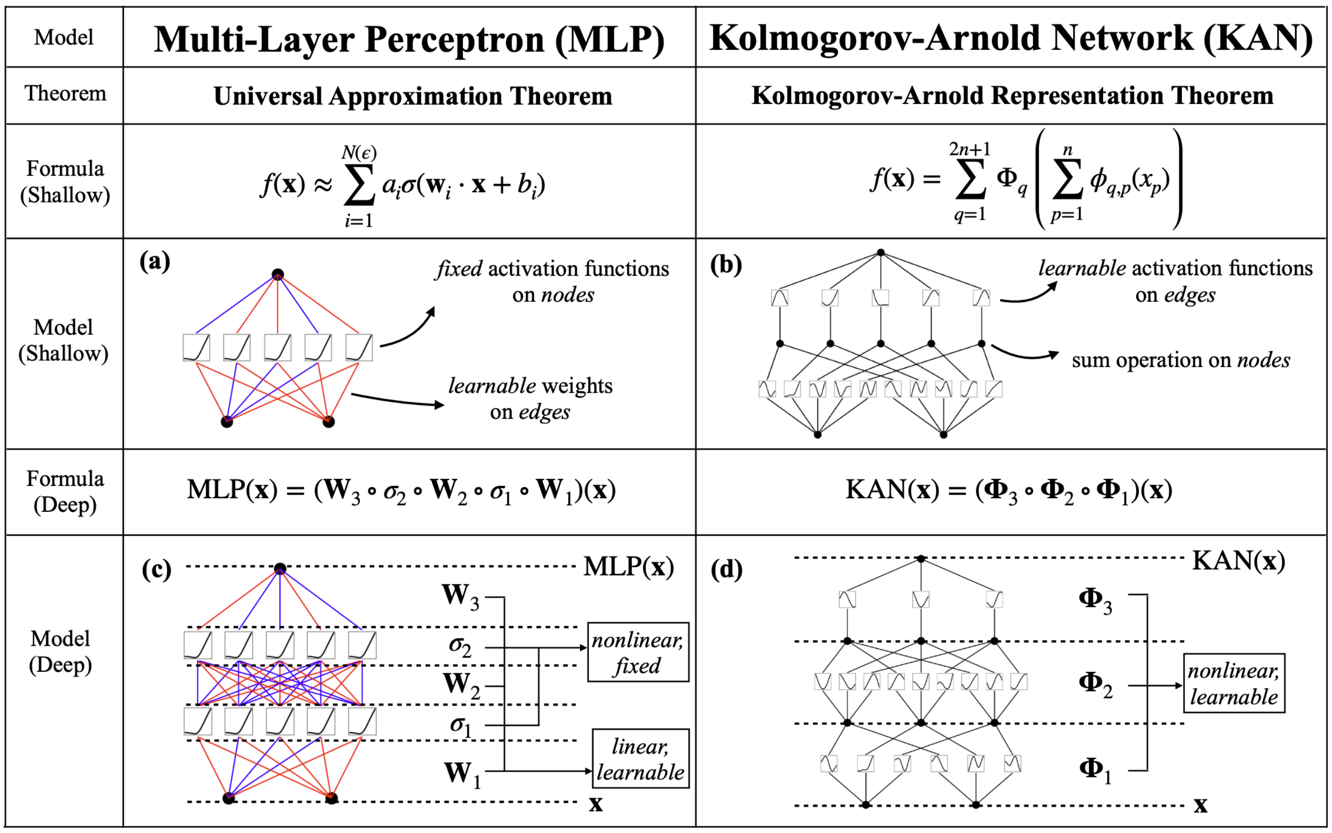
$$MLP(x) = ((W3 \circ \sigma 2 \circ W2 \circ \sigma 1 \circ W1)(x))$$

   - KAN：沒有線性權重矩陣，所有權重都被樣條函數替代；節點僅進行簡單的信號相加操作，不進行非線性變換。

- 結構公式：

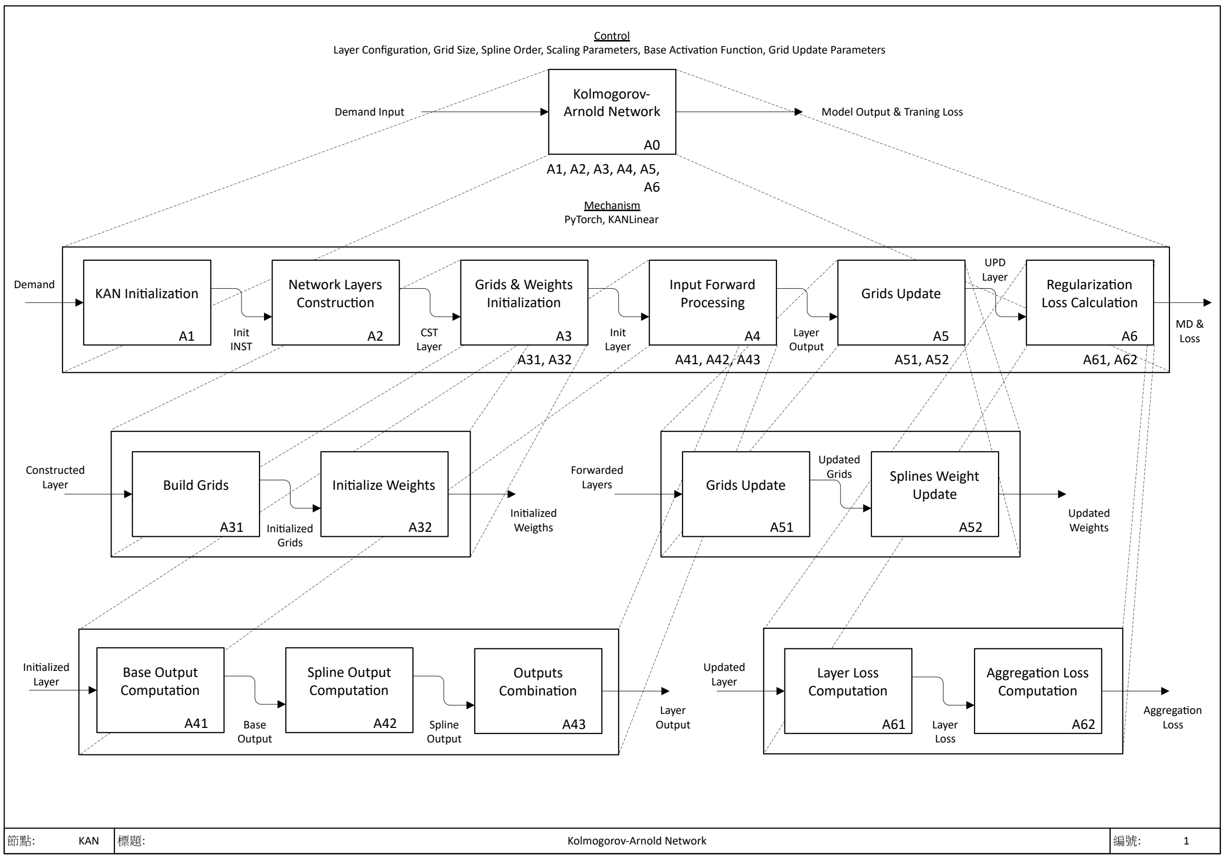
$$KAN(x) = ((\Phi 3 \circ \Phi 2 \circ \Phi 1)(x))$$

3. 訓練方法和參數優化
   - MLP：權重矩陣通過梯度下降法進行訓練；訓練過程需要調整大量的線性權重參數。
   - KAN：核心在於樣條函數的參數化和學習；樣條函數通過調整其參數進行優化。
4. 表達能力和適用範圍
   - MLP：基於普適近似定理，MLP 能夠逼近任意連續函數；常用於各種回歸和分類問題，但在高維數據下可能效率低下。
   - KAN：基於 Kolmogorov-Arnold 表示定理，能夠表達高維數據的組合結構和單變量函數；對於需要高準確度和解釋性的應用，如數學模型和物理模型，有顯著優勢。
5. 可解釋性和直觀性
   - MLP：由於固定的激活函數和複雜的權重矩陣，MLP 在解釋性方面較為薄弱；解釋模型需要額外的工具和方法，如 SHAP、LIME 等。
   - KAN：由於激活函數是可學習的單變量函數，KAN 的結構更易於直觀理解；KAN 的節點僅進行信號相加，使得整體網路更易於可視化和解釋。
6. 計算和資源需求
   - MLP：訓練和推理過程中，計算資源需求較大，特別是在高維數據和大模型情況下。
   - KAN：由於樣條函數的引入，KAN 在同樣準確度下所需的參數和計算資源相對較少；能夠在較小的計算圖上達到與大型 MLP 相同甚至更好的準確度。

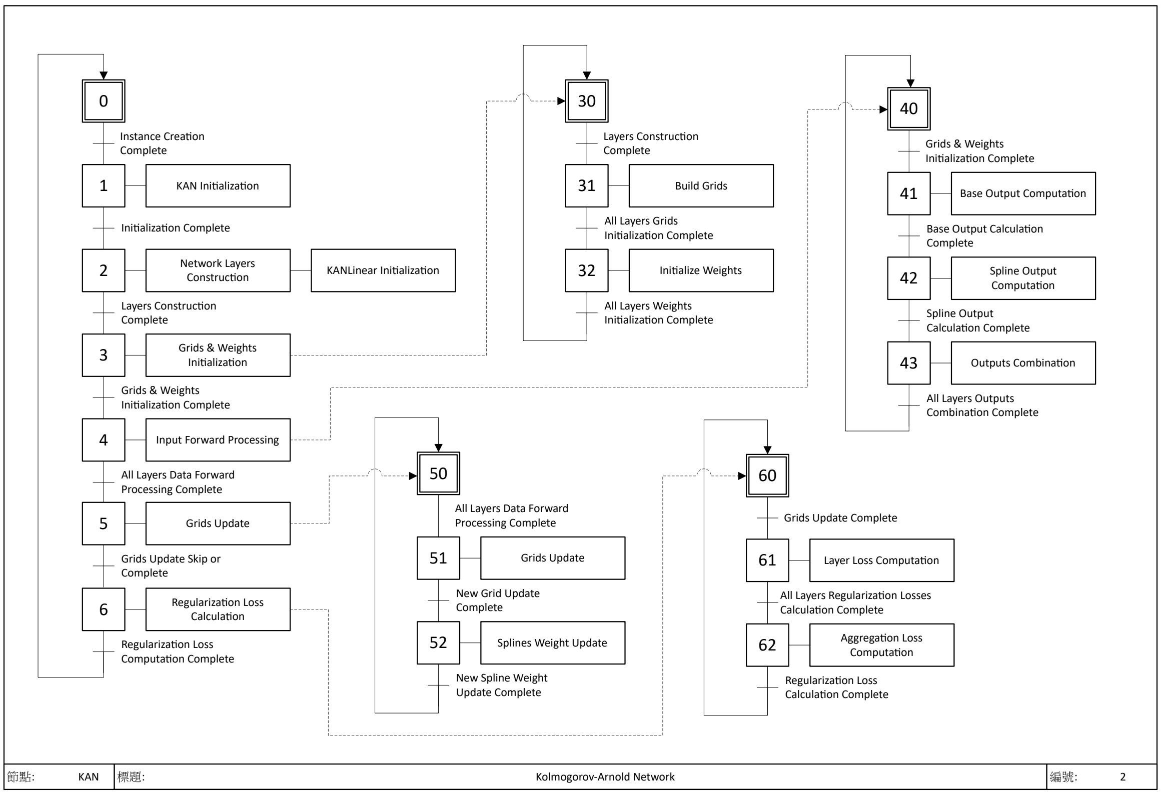


Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a)	(b)
Formula (Deep)	$\mathrm{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\mathrm{KAN}(\mathbf{x}) = (\mathbf{\Phi}_3 \circ \mathbf{\Phi}_2 \circ \mathbf{\Phi}_1)(\mathbf{x})$
Model (Deep)	(c)	(d)

IDEF0 設計



GRAFCET 設計


0
Instance Creation Complete
1 KAN Initialization
Initialization Complete
2 Network Layers Construction
KANLinear Initialization
Layers Construction Complete
3 Grids & Weights Initialization
Grids & Weights Initialization Complete
4 Input Forward Processing
All Layers Data Forward Processing Complete
5 Grids Update
Grids Update Skip or Complete
6 Regularization Loss Calculation
Regularization Loss Computation Complete
30
Layers Construction Complete
31 Build Grids
All Layers Grids Initialization Complete
32 Initialize Weights
All Layers Weights Initialization Complete
40
Grids & Weights Initialization Complete
41 Base Output Computation
Base Output Calculation Complete
42 Spline Output Computation
Spline Output Calculation Complete
43 Outputs Combination
All Layers Outputs Combination Complete
50
All Layers Data Forward Processing Complete
51 Grids Update
New Grid Update Complete
52 Splines Weight Update
New Spline Weight Update Complete
60
Grids Update Complete
61 Layer Loss Computation
All Layers Regularization Losses Calculation Complete
62 Aggregation Loss Computation
Regularization Loss Calculation Complete
節點: KAN
標題: Kolmogorov-Arnold Network
编號: 2

Python PyTorch 模擬驗證

- 基於 IDEF0 和 Grafcet 重構後之 Kolmogorov-Arnold Network

Info

基於 Grafcet 設計後的一些調整

- Original Implementation 的 Performance 問題主要在於需要展開所有中間變數來執行不同的 Activation Function。對於具有 `in_features` 輸入和 `out_features` 輸出的層，原始實現需要將輸入展開為形狀為 `(batch_size, out_features, in_features)` 的 Tensor 來執行 Activation Function。然而，所有的 Activation Function 都是一組固定 Base Function（B-Spline）的線性組合，也就可以將計算重新整理為使用不同的 Base Function 激活輸入，然後將它們線性組合。

```python
import math

import torch
import torch.nn.functional as F


class KANLinear(torch.nn.Module):
    def __init__(
            self,
            in_features,
            out_features,
            grid_size=5,
            spline_order=3,
            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],
    ):
        super(KANLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.grid_size = grid_size
        self.spline_order = spline_order

        # 構建網格點
        self.grid = self.build_grid(grid_range, grid_size, spline_order)

        # 初始化基礎權重和樣條權重
        self.base_weight, self.spline_weight, self.spline_scaler = self.initialize_weights(
            out_features, in_features, grid_size, spline_order, scale_base, scale_spline,
enable_standalone_scale_spline
        )

        self.scale_base = scale_base
        self.scale_spline = scale_spline
        self.enable_standalone_scale_spline = enable_standalone_scale_spline
        self.base_activation = base_activation()
        self.grid_eps = grid_eps

    def build_grid(self, grid_range, grid_size, spline_order):
        h = (grid_range[1] - grid_range[0]) / grid_size
        grid = (
            (
                    torch.arange(-spline_order, grid_size + spline_order + 1) * h
                    + grid_range[0]
            )
            .expand(self.in_features, -1)
            .contiguous()
        )
        return grid

    def initialize_weights(self, out_features, in_features, grid_size, spline_order, scale_base, scale_spline,
                           enable_standalone_scale_spline):
        base_weight = torch.nn.Parameter(torch.Tensor(out_features, in_features))
        spline_weight = torch.nn.Parameter(
            torch.Tensor(out_features, in_features, grid_size + spline_order)
        )
        if enable_standalone_scale_spline:
            spline_scaler = torch.nn.Parameter(
                torch.Tensor(out_features, in_features)
            )
        else:
            spline_scaler = None
        torch.nn.init.kaiming_uniform_(base_weight, a=math.sqrt(5) * scale_base)
        torch.nn.init.kaiming_uniform_(spline_weight, a=math.sqrt(5) * scale_spline)
        if enable_standalone_scale_spline:
            torch.nn.init.kaiming_uniform_(spline_scaler, a=math.sqrt(5) * scale_spline)
        return base_weight, spline_weight, spline_scaler

    def b_splines(self, x: torch.Tensor):
        bases = self.calculate_b_spline_bases(x)
        return bases.contiguous()

    def calculate_b_spline_bases(self, x: torch.Tensor):
        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:]
                    )
        return bases

    def curve2coeff(self, x: torch.Tensor, y: torch.Tensor):
        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)
        return result.contiguous()

    @property
    def scaled_spline_weight(self):
        if self.enable_standalone_scale_spline:
            return self.spline_weight * self.spline_scaler.unsqueeze(-1)
        else:
            return self.spline_weight

    def forward(self, x: torch.Tensor):
        base_output = self.compute_base_output(x)
        spline_output = self.compute_spline_output(x)
        return base_output + spline_output

    def compute_base_output(self, x: torch.Tensor):
        return F.linear(self.base_activation(x), self.base_weight)

    def compute_spline_output(self, x: torch.Tensor):
        return F.linear(
            self.b_splines(x).view(x.size(0), -1),
            self.scaled_spline_weight.view(self.out_features, -1),
        )

    @torch.no_grad()
    def update_grid(self, x: torch.Tensor, margin=0.01):
        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)

        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
        )

        grid = self.grid_eps * grid_uniform + (1 - self.grid_eps) * grid_adaptive
        grid = torch.concatenate(
            [
                grid[:1]
                - uniform_step
                * torch.arange(self.spline_order, 0, -1, device=x.device).unsqueeze(1),
                grid,
                grid[-1:]
                + uniform_step
                * torch.arange(1, self.spline_order + 1, device=x.device).unsqueeze(1),
            ],
            dim=0,
        )

        self.grid.copy_(grid.T)
        self.spline_weight.data.copy_(self.curve2coeff(x, unreduced_spline_output))

    def regularization_loss(self, regularize_activation=1.0, regularize_entropy=1.0):
        l1_fake = self.spline_weight.abs().mean(-1)
        regularization_loss_activation = l1_fake.sum()
        p = l1_fake / regularization_loss_activation
        regularization_loss_entropy = -torch.sum(p * p.log())
        return (
                regularize_activation * regularization_loss_activation
                + regularize_entropy * regularization_loss_entropy
        )


class KAN(torch.nn.Module):
    def __init__(
            self,
            layers_hidden,
            grid_size=5,
            spline_order=3,
            scale_base=1.0,
            scale_spline=1.0,
            base_activation=torch.nn.SiLU,
            grid_eps=0.02,
            grid_range=[-1, 1],
    ):
        super(KAN, self).__init__()
        self.grid_size = grid_size
        self.spline_order = spline_order
        # 構建 KAN 的層
        self.layers = self.build_layers(
            layers_hidden, grid_size, spline_order, scale_base, scale_spline, base_activation, grid_eps,
grid_range
        )

    def build_layers(self, layers_hidden, grid_size, spline_order, scale_base, scale_spline, base_activation,
grid_eps,
                     grid_range):
        layers = torch.nn.ModuleList()
        for in_features, out_features in zip(layers_hidden, layers_hidden[1:]):
            layers.append(
                KANLinear(
                    in_features,
                    out_features,
                    grid_size=grid_size,
                    spline_order=spline_order,
                    scale_base=scale_base,
                    scale_spline=scale_spline,
                    base_activation=base_activation,
                    grid_eps=grid_eps,
                    grid_range=grid_range,
                )
            )
        return layers

    def forward(self, x: torch.Tensor, update_grid=False):
        for layer in self.layers:
            if update_grid:
                layer.update_grid(x)
            x = layer(x)
        return x

    def regularization_loss(self, regularize_activation=1.0, regularize_entropy=1.0):
        return sum(
            layer.regularization_loss(regularize_activation, regularize_entropy)
            for layer in self.layers
        )
```

Vision Transformer (ViT)

相關參考資料

- 原始論文：An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale
- 原始論文網址：https://arxiv.org/abs/2010.11929

BIBTEX 供論文引用

```
@article{dosovitskiy2020vit,
  title={An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale},
  author={Dosovitskiy, Alexey and Beyer, Lucas and Kolesnikov, Alexander and Weissenborn, Dirk and Zhai, Xiaohua
and Unterthiner, Thomas and  Dehghani, Mostafa and Minderer, Matthias and Heigold, Georg and Gelly, Sylvain and
Uszkoreit, Jakob and Houlsby, Neil},
  journal={ICLR},
  year={2021}
}
```

論文內容

ViT PyTorch 重構