Vi phân kiến trúc mạng

Tìm kiếm kiến trúc mạng khả vi trên không gian

Nhóm 8 - Neural Architecture Search

Ngày 8 tháng 11 năm 2023

Mục lục

Bài toán tối ưu hai cấp

Bài toán tối ưu hai cấp (Bilevel Optimization Problem) tổng quát có dạng:

$$\min_{x \in X \subseteq \mathbb{R}^{d_1}} \ell(x) \coloneqq f(x, y^*(x))$$
$$\text{với } y^*(x) \in \arg\min_{y \in \mathbb{R}^{d_2}} g(x, y)$$

Trong đó $d_1, d_2 \in \mathbb{N}$, X là tập đóng, lồi và $f, g : X \times \mathbb{R}^{d_2} \to \mathbb{R}$ là các hàm số khả vi liên tục theo từng biến x, y.

- $\min_y g(x, y)$ là bài toán trong (cấp dưới)
- $\min_x \ell(x)$ là bài toán ngoài (cấp trên)

Ngay cả khi hàm số $\ell(x)$ và $g(x, y)$ lồi chặt theo các biến thì bài toán cũng rất khó giải quyết.

Lời giải bài toán

Để giải bài toán theo phương pháp gradient, với mỗi $x^{\text{cur}} \in \mathbb{R}^{d_1}$, ta thực hiện 2 vòng lặp để:

1. Giải bài toán trong $y^*(x^{\text{cur}}) = \arg\min g(x^{\text{cur}}, y)$ và sau đó
2. Tính toán gradient $\nabla \ell(x^{\text{cur}})$ dựa trên lời giải $y^*(x^{\text{cur}})$

Giải bài toán BiO, ta tập trung tìm cặp nghiệm (x^*, y^*) thỏa mãn điều kiện dừng bậc một:

$$\nabla_y g(x^*, y^*) = 0 \,(\text{i}) \quad \text{và} \quad \langle \nabla \ell(x^*), x - x^* \rangle \geq 0, \forall x \in X \,(\text{ii})$$

- Nếu cho trước x^*, nghiệm y^* thỏa (i) có thể tìm được bằng cách cập nhật:

$$y \leftarrow y - \beta \nabla_y g(x^*, y)$$

- Mặt khác, nếu cho trước $y^*(x)$, nghiệm x^* thỏa (ii) cập nhật theo cách sau:

$$x \leftarrow x - \alpha \nabla \ell(x)$$

Theo quy tắc vi phân toàn phần hàm nhiều biến, ta có:

$$\nabla \ell(x) = \nabla_x f(x, y^*(x)) + \nabla_x y^*(x)^\top \nabla_y f(x, y^*(x))$$

Mặt khác, $y^*(x)$ thỏa mãn $\nabla_y g(x, y^*(x)) = 0$. Lấy vi phân hàm ẩn:

$$\nabla^2_{xy} g(x, y^*(x)) + \nabla_x y^*(x)^\top \nabla^2_{yy} g(x, y^*(x)) = 0$$

Do đó:

$$\nabla \ell(x) = \nabla_x f(x, y^*(x)) - \nabla^2_{xy} g(x, y^*(x)) \left[\nabla^2_{yy} g(x, y^*(x))\right]^{-1} \nabla_y f(x, y^*(x))$$

Thay $y^*(x)$ bởi $y \in \mathbb{R}^{d_2}$. Ta định nghĩa:

$$\overline{\nabla}_x f(x, y) \coloneqq \nabla_x f(x, y) - \nabla^2_{xy} g(x, y) \left[\nabla^2_{yy} g(x, y)\right]^{-1} \nabla_y f(x, y)$$

Giả mã giải thuật

Thuật toán 1: TTSA[1]

1. Bắt đầu với $(x_0, y_0) \in X \times \mathbb{R}^{d_2}$ và dãy bước nhảy $\{\alpha_k, \beta_k\}_{k \geq 0}$.
2. Trong vòng lặp thứ k, ta cập nhật:

$$y_{k+1} = y_k - \beta_k \nabla_y g(x_k, y_k)$$
$$x_{k+1} = x_k - \alpha_k \overline{\nabla}_x f(x_k, y_{k+1})$$

[1] A Two-Timescale Stochastic Algorithm Framework for Bilevel Optimization: Complexity Analysis and Application to Actor-Critic

Biểu diễn kiến trúc mạng

Biểu diễn mạng bởi các khối tính toán (computation cell), sắp xếp các khối tạo nên kiến trúc mạng cuối cùng.

- Mỗi khối là một đồ thị có hướng không chu trình, gồm dãy N nút có thứ tự.
- Nối giữa hai nút x_i và x_j $(i < j)$ bởi một cạnh có hướng (i, j) thể hiện toán tử $o_{ij}(\cdot)$.
- Một nút sẽ được tính toán bằng tổng tất cả các nút trước nó sau khi tác động toán tử.

$$x_j = \sum_{i<j} o_{ij}(x_i)$$

- Có những *toán tử không*, thể hiện không kết nối.
- Đầu ra của cả khối do sự xếp chồng tất cả các nút, hay là concat$(x_1, x_2, \ldots, x_N)$.

Nới lỏng ràng buộc liên tục

Xét một nút x_i nào đó. Gọi $\mathcal{O}$ là tập các toán tử ứng viên, chẳng hạn: tích chập, gộp tối đại, và cả *toán tử không*.

Để không gian tìm kiếm trở nên liên tục, ta giảm sự lựa chọn theo loại của một toán tử cụ thể thành một softmax trên tất cả các toán tử có thể:

$$\overline{o}_{ij}(x_i) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_{ij}(o))}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{ij}(o'))} o(x_i)$$

Trong đó, mỗi toán tử ứng với cạnh (i, j) được tham số hóa bởi vector $\alpha_{ij} \in \mathbb{R}^{|\mathcal{O}|}$.

Việc tìm kiến kiến trúc mạng trở thành việc học các biến mã hóa liên tục $\alpha = \{\alpha_{ij}\}$. Kết thúc tìm kiếm, ta thay toán tử hỗn hợp $\overline{o}_{ij}$ bởi toán tử nào tựa nó nhất hay là:

$$o_{ij} = \arg\max_{o \in \mathcal{O}} \alpha_{ij}(o)$$

Mục tiêu tối ưu

Ta sẽ học đồng thời kiến trúc α và trọng số w trong mọi toán tử và cực tiểu hóa mất mát trên tập đánh giá (validation set) bằng gradient descent.

Gọi $\mathcal{L}_{\text{train}}$ và $\mathcal{L}_{\text{validation}}$ tương ứng là mất mát trên tập huấn luyện và tập đánh giá. Cả hai đều phụ thuộc kiến trúc α và trọng số w.

Nhiệm vụ trở thành Bài toán tối ưu hai cấp độ:

$$\min_{\alpha} \mathcal{L}_{\text{validation}}(w^*(\alpha), \alpha)$$
$$\text{với } w^*(\alpha) = \arg\min_{w} \mathcal{L}_{\text{train}}(w, \alpha)$$

Trong đó α là biến cấp trên và w là biến cấp dưới.

Xấp xỉ gradient kiến trúc

Theo Thuật toán TTSA, ta thực hiện cập nhật theo dạng:

$$w_{k+1} = w_k - B_k \nabla \mathcal{L}_{\text{train}}(w_k, \alpha_k)$$
$$\alpha_{k+1} = \alpha_k - A_k \overline{\nabla}_\alpha \mathcal{L}_{\text{validation}}(w_{k+1}, \alpha_k)$$

Tính toán chính xác gradient kiến trúc có thể trở nên không khả thi do bài toán tối ưu cấp dưới tốn kém. Với $w' = w - \xi \nabla_w \mathcal{L}_{\text{train}}(w, \alpha)$, lấy gradient toàn phần ta có:

$$\overline{\nabla}_\alpha \mathcal{L}_{\text{validation}}(w', \alpha) = \underbrace{\nabla_\alpha \mathcal{L}_{\text{validation}}(w', \alpha)}_{\text{thành phần cấp 1}} - \xi \underbrace{\nabla^2_{\alpha w} \mathcal{L}_{\text{train}}(w, \alpha)^\top \nabla_{w'} \mathcal{L}_{\text{validation}}(w', \alpha)}_{\text{thành phần cấp 2}} \quad \text{(i)}$$

Trong đó ta sử dụng tốc độ học $\xi \geq 0$ chung cho cả hai bài toán tối ưu. Không cần giải chính xác bài toán cấp dưới hay huấn luyện đến khi nó hội tụ. Nếu $\xi = 0$, ta có thể xấp xỉ (i) bởi thành phần gradient cấp 1 $\nabla_\alpha \mathcal{L}_{\text{validation}}(w, \alpha)$, tuy nhanh hơn nhưng kém hiệu quả hơn.

Xấp xỉ gradient kiến trúc

Thành phần gradient cấp 2 chứa phép toán nhân ma trận có khối lượng tính toán lớn. Do đó, ta sẽ tìm cách xấp xỉ đại lượng này.

Với số ε đủ nhỏ, chọn $\varepsilon = \frac{0.01}{\|\nabla_{w'} \mathcal{L}_{\text{validation}}(w', \alpha)\|_2}$ và đặt $w^{\pm} = w \pm \varepsilon \mathcal{L}_{\text{validation}}(w', \alpha)$. Khi đó:

$$\nabla^2_{\alpha w} \mathcal{L}_{\text{train}}(w, \alpha)^\top \nabla_{w'} \mathcal{L}_{\text{validation}}(w', \alpha) \approx \frac{\nabla_\alpha \mathcal{L}_{\text{train}}(w^+, \alpha) - \nabla_\alpha \mathcal{L}_{\text{train}}(w^-, \alpha)}{2\varepsilon}$$

Tóm tắt các bước thuật toán

Thuật toán 2: DARTS[2]

1. Tạo toán tử hỗn hợp $\overline{o}_{ij}$ tham số hóa bởi α_{ij} cho các cạnh (i, j).
2. **while** *chưa đạt điều kiện dừng* **do**
   2.1 Cập nhật kiến trúc α:
   $$\alpha \coloneqq \alpha - \nabla_\alpha \mathcal{L}_{\text{validation}}\left(w - \xi \nabla_w \mathcal{L}_{\text{train}}(w, \alpha), \alpha\right)$$
   2.2 Cập nhật trọng số w:
   $$w \coloneqq w - \nabla_w \mathcal{L}_{\text{train}}(w, \alpha)$$
3. Huấn luyện kiến trúc α học được.

[2] DARTS: Differentiable Architecture Search

Định nghĩa các khối phép toán

1. Phép toán không

```
class Zero(nn.Module):

  def __init__(self, stride):
    super(Zero, self).__init__()
    self.stride = stride

  def forward(self, x):
    if self.stride == 1:
      return x.mul(0.)
    return x[:,:,::self.stride,::self.stride].mul(0.)
```

Mục đích: Trả về tensor toàn số 0 có cùng hoặc đã giảm kích thước dựa trên bước nhảy.

2. Phép toán đồng nhất

```
class Identity(nn.Module):

  def __init__(self):
    super(Identity, self).__init__()

  def forward(self, x):
    return x
```

Mục đích: Trả về chính đầu vào.

3. Phép toán điều chỉnh kích thước kênh

- Lấy cảm hứng từ kiến trúc NASNet và AmoebaNet.
- Sử dụng 2 phép toán tích chập, có bước nhảy 2, kích thước chỉ là 1×1.
- Mỗi phép tạo ra một tensor có độ sâu $C_{\text{out}}/2$, sau đó nối chúng lại.
- Kết quả là tạo ra tensor có độ sâu (số kênh) đúng bằng C_{out} (nếu C_{out} chẵn).

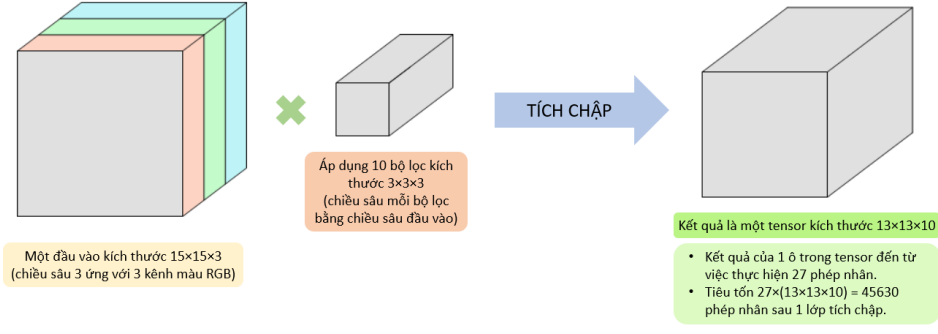
```
class FactorizedReduce(nn.Module):

  def __init__(self, C_in, C_out, affine=True):
    super(FactorizedReduce, self).__init__()
    assert C_out % 2 == 0
    self.relu = nn.ReLU(inplace=False)
    self.conv_1 = nn.Conv2d(C_in, C_out // 2, 1, stride=2, padding=0, bias=False)
    self.conv_2 = nn.Conv2d(C_in, C_out // 2, 1, stride=2, padding=0, bias=False)
    self.bn = nn.BatchNorm2d(C_out, affine=affine)

  def forward(self, x):
    x = self.relu(x)
    out = torch.cat([self.conv_1(x), self.conv_2(x[:,:,1:,1:])], dim=1)
    out = self.bn(out)
    return out
```

4. Phép toán tích chập phân tách (Separable Convolution)

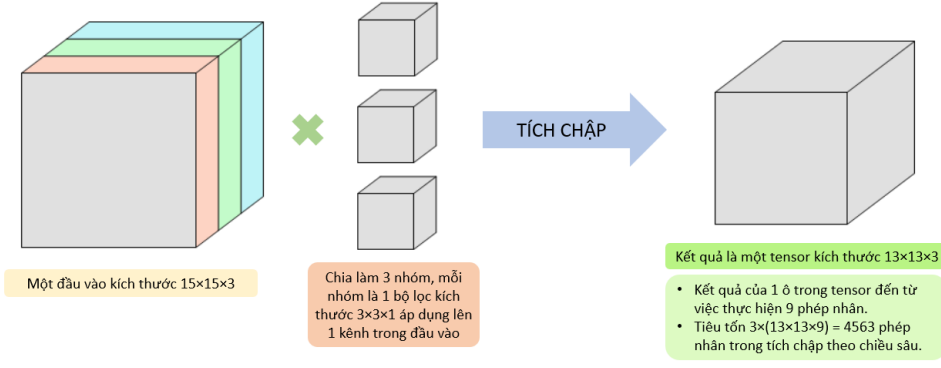
- Phiên bản tối ưu của tích chập truyền thống.
- Giúp giảm đáng kể số lượng phép nhân và tham số.
- Gồm 2 phép tích chập liên tiếp theo thứ tự: Depthwise, Pointwise.



Hình 1: Minh họa khối lượng tính toán của tích chập thông thường

4.1. Phép toán tích chập theo chiều sâu (Depthwise Convolution)

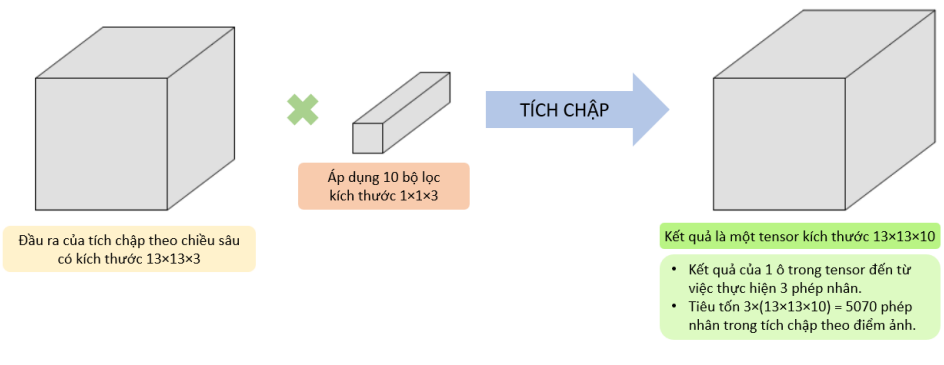
- Chia các bộ lọc làm C_{in} nhóm.
- Mỗi nhóm có một bộ lọc và thực hiện tích chập tương ứng lên một kênh.
- Đầu ra là một tensor bảo toàn số kênh.



Hình 2: Minh họa khối lượng tính toán của tích chập theo chiều sâu

4.2. Phép toán tích chập theo điểm ảnh (Pointwise Convolution)

- Áp dụng các bộ lọc kích thước 1×1, chiều sâu đúng bằng tensor áp dụng và số lượng đúng bằng chiều sâu muốn đạt được.
- Hai giai đoạn tích chập này có tổng khối lượng tính toán ít hơn rất nhiều so với tích chập truyền thống.



Hình 3: Minh họa khối lượng tính toán của tích chập theo điểm ảnh

```
class SepConv(nn.Module):

  def __init__(self, C_in, C_out, kernel_size, stride, padding, affine=True):
    super(SepConv, self).__init__()
    self.op = nn.Sequential(
      nn.ReLU(inplace=False),
      nn.Conv2d(C_in, C_in, kernel_size=kernel_size, stride=stride, padding=padding,
          groups=C_in, bias=False),
      nn.Conv2d(C_in, C_in, kernel_size=1, padding=0, bias=False),
      nn.BatchNorm2d(C_in, affine=affine),
      nn.ReLU(inplace=False),
      nn.Conv2d(C_in, C_in, kernel_size=kernel_size, stride=1, padding=padding, groups=
          C_in, bias=False),
      nn.Conv2d(C_in, C_out, kernel_size=1, padding=0, bias=False),
      nn.BatchNorm2d(C_out, affine=affine),
      )

  def forward(self, x):
    return self.op(x)
```

5. Phép toán tích chập mở rộng[3]

Với $F : \mathbb{Z}^2 \to \mathbb{R}$ là tensor và $k : [-r, r]^2 \cap \mathbb{Z}^2 \to \mathbb{R}$ là kernel kích thước $(2r + 1) \times (2r + 1)$. Phép toán tích chập $*$ giữa F và k được định nghĩa là:

$$(F * k)(\mathbf{p}) = \sum_{\mathbf{s}+\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t})$$

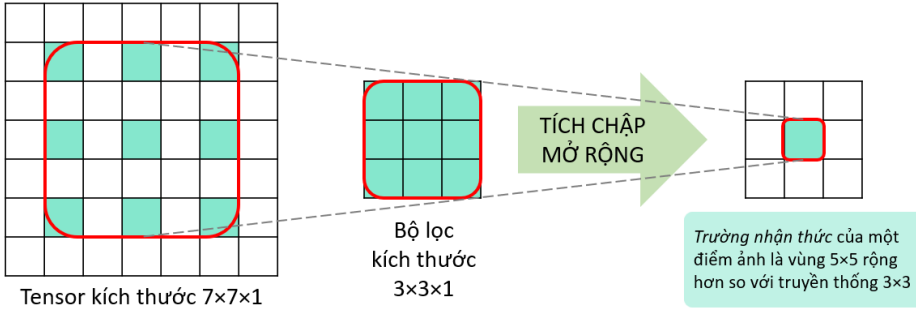
Ta mở rộng khái niệm này, phép toán tích chập với độ mở rộng $\ell \in \mathbb{N}$, kí hiệu $*_\ell$ cho bởi:

$$(F *_\ell k)(\mathbf{p}) = \sum_{\mathbf{s}+\ell\mathbf{t}=\mathbf{p}} F(\mathbf{s})k(\mathbf{t})$$

[3] Multi-Scale Context Aggregation by Dilated Convolutions

So với tích chập truyền thông, tích chập mở rộng có những ưu điểm hơn hẳn:

- Tăng trường nhận thức mà không tăng số lượng tham số, tăng hiệu suất tính toán.
- Giảm mất mát thông tin theo không gian so với phép gộp, duy trì trật tự dữ liệu.
- Không làm mất độ phân giải của hình ảnh đầu ra.



Hình 4: Minh họa tích chập mở rộng có độ mở rộng $\ell = 2$

```python
class DilConv(nn.Module):

  def __init__(self, C_in, C_out, kernel_size, stride, padding, dilation, affine=True):
    super(DilConv, self).__init__()
    self.op = nn.Sequential(
      nn.ReLU(inplace=False),
      nn.Conv2d(C_in, C_in, kernel_size=kernel_size, stride=stride, padding=padding,
          dilation=dilation, groups=C_in, bias=False),
      nn.Conv2d(C_in, C_out, kernel_size=1, padding=0, bias=False),
      nn.BatchNorm2d(C_out, affine=affine),
      )

  def forward(self, x):
    return self.op(x)
```

Định nghĩa toán tử ứng viên

Tập các toán tử ứng viên bao gồm:

- `none`: Phép toán không.
- `avg_pool_3x3`: Gộp trung bình.
- `max_pool_3x3`: Gộp tối đại.
- `skip_connection`: Đồng nhất hoặc Điều chỉnh kênh để phù hợp với đầu vào.
- `sep_conv_3x3, sep_conv_5x5, sep_conv_7x7`
- `dil_conv_3x3, dil_conv_5x5`
- `conv_7x1_1x7`: Tương đương áp dụng bộ lọc 7×7.

Hạn chế của DARTS

Với thuật toán DARTS trước đó, nhận thấy:

- Một khối (cell) được tạo nên từ các nút và các kết nối, mỗi kết nối là toán tử. Các toán tử gán trọng số kiến trúc, được học trong quá trình tìm kiếm.
- DARTS tìm kiếm kiến trúc trên mạng nông sau đó đánh giá trên mạng sâu hơn.

Như vậy, DARTS dẫn đến vấn đề *chênh lệch độ sâu*, nghĩa là giai đoạn tìm kiếm tìm ra một số toán tử hoạt động tốt trong kiến trúc nông, nhưng giai đoạn đánh giá thực sự cần những toán tử khác phù hợp hơn với một kiến trúc sâu.

Đề xuất của P-DARTS[4]

Khắc phục các hạn chế của DARTS, giải thuật tiến bộ hơn được đề xuất:

- **Tăng dần độ sâu:** Chia quá trình tìm kiếm thành nhiều giai đoạn và tăng dần độ sâu của mạng ở cuối mỗi giai đoạn.
- **Không gian tìm kiếm xấp xỉ** (*search space approximation*): Khi độ sâu tăng lên, P-DARTS giảm số lượng ứng viên (toán tử) dựa trên điểm số của họ trong các giai đoạn tìm kiếm trước đó. Mục tiêu là giảm thiểu chi phí tính toán và tăng tốc độ tìm kiếm.
- **Không gian tìm kiếm hiệu chỉnh** (*search space regularization*): Một vấn đề khác, thiếu ổn định, xuất hiện khi tìm kiếm trên một kiến trúc sâu. Thuật toán có thể bị lệch nặng về kết nối tắt (skip connect) vì nó thường dẫn đến sự giảm lỗi nhanh chóng trong quá trình tối ưu hóa.
  i. Sử dụng dropout cấp độ toán tử để giảm chi phối của kết nối tắt.
  ii. Kiểm soát sự xuất hiện của kết nối tắt trong quá trình đánh giá.

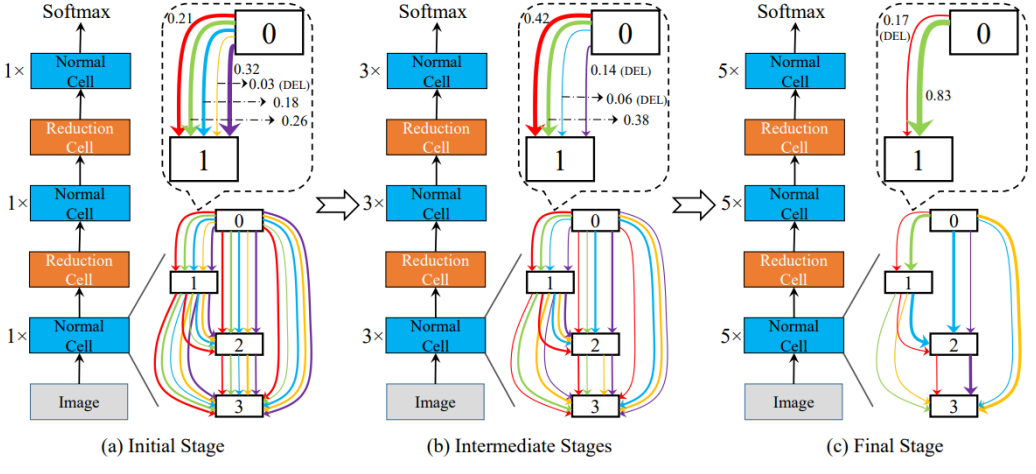
[4] Progressive Differentiable Architecture Search: Bridging the Depth Gap between Search and Evaluation

Không gian tìm kiếm xấp xỉ

Quá trình tìm kiếm được chia thành nhiều giai đoạn, bao gồm một giai đoạn ban đầu, một hoặc vài giai đoạn trung gian và một giai đoạn cuối.

Đối với mỗi giai đoạn, $\mathcal{S}_k$, mạng bao gồm $\mathcal{L}_k$ khối (cells) và tập toán tử có kích thước $\mathcal{O}_k$ hay là $\left|\mathcal{O}_{ij}^k\right| = \mathcal{O}_k$. Tức mọi cạnh đều có tập toán tử gồm $\mathcal{O}_k$ ứng viên.

- Ở giai đoạn đầu tiên, mạng tìm kiếm nông nhưng không gian toán tử lớn.
- Sau giai đoạn $\mathcal{S}_{k-1}$, tham số kiến trúc α_{k-1} được học và thứ hạng các toán tử trên mỗi cạnh được xếp hạng theo α_{k-1}. Khi đó, ta tăng số khối hay $\mathcal{L}_k > \mathcal{L}_{k-1}$ và giảm số toán tử hay $\mathcal{O}_k < \mathcal{O}_{k-1}$.
- Quá trình tăng chiều sâu kiến trúc tiếp tục cho đến khi nó đủ gần với chiều sâu được sử dụng để đánh giá. Sau giai đoạn tìm kiếm cuối cùng, xác định các khối theo các tham số kiến trúc đã học.



Hình 5: Minh họa các giai đoạn tìm kiếm: tăng khối và giảm toán tử

Không gian tìm kiếm hiệu chỉnh

Ở bắt đầu mỗi giai đoạn, $\mathcal{S}_k$, huấn luyện lại kiến trúc (lúc này đã sửa đổi) từ đầu, nghĩa là tất cả trọng số được khởi tạo, vì một vài toán tử ứng viên đã bị loại bỏ.

- Tuy nhiên, việc huấn luyện một mạng sâu hơn khó khăn hơn so với việc huấn luyện một mạng nông.
- Bằng thực nghiệm, nhận thấy thông tin thích chảy qua các kết nối tắt thay vì tích chập hoặc gộp, có lẽ do kết nối tắt thường dẫn đến sự giảm gradient nhanh chóng.

Do đó, quá trình tìm kiếm có xu hướng tạo ra các kiến trúc với nhiều toán tử kết nối tắt, giới hạn số lượng tham số có thể học và do đó tạo ra hiệu suất không thỏa đáng ở giai đoạn đánh giá. Đây là một dạng quá khớp (overfitting).

Giải quyết hạn chế bằng hiệu chỉnh không gian tìm kiếm, gồm hai phần:

i Chèn dropout ở cấp độ toán tử sau mỗi kết nối tắt nhằm cắt đứt một phần con đường truyền trực tiếp qua kết nối tắt, tạo điều kiện cho giải thuật khám phá các toán tử khác. Tuy nhiên, nếu liên tục chặn các kết nối tắt thì thuật toán sẽ loại bỏ bằng cách gán trọng số thấp cho chúng, điều này gây hại cho hiệu suất cuối cùng. Do đó, tỉ lệ dropout phải giảm dần theo thời gian.

ii Tinh chỉnh kiến trúc, đơn giản là kiểm soát số lượng kết nối tắt được giữ lại, sau giai đoạn tìm kiếm cuối cùng, để trở thành một hằng số M.

Quá trình này lặp đi lặp lại:

i Bắt đầu với việc xây dựng một cấu trúc khối (cell topology) bằng cách sử dụng thuật toán tiêu chuẩn mô tả bởi DARTS.

ii Kiểm tra xem số lượng kết nối tắt có đúng bằng M hay không.

iii Nếu số lượng kết nối tắt không chính xác, tìm M kết nối tắt có trọng số lớn nhất và đặt trọng số của những kết nối còn lại về 0.

iv Sau đó, xây dựng lại cấu trúc khối với các tham số kiến trúc đã được sửa đổi.

Việc điều chỉnh trọng số và xây dựng lại cấu trúc khối có thể tạo ra các kết nối tắt mới không mong muốn. Do đó, phải lặp lại quá trình này cho đến khi đạt được số lượng kết nối tắt mong muốn.

Chi tiết triển khai

Kỹ thuật điều chỉnh thứ hai (điều chỉnh số lượng kết nối tắt) cần được áp dụng sau khi đã áp dụng kỹ thuật điều chỉnh đầu tiên (dropout sau mỗi kết nối tắt). Nếu không áp dụng kỹ thuật đầu tiên, quá trình tìm kiếm sẽ tạo ra các trọng số kiến trúc kém chất lượng. Với những trọng số này, không thể xây dựng một kiến trúc mạnh mẽ ngay cả khi có một số lượng cố định của kết nối tắt.

Toàn bộ quá trình tìm kiếm bao gồm 3 giai đoạn. Trong đó, sử dụng DARTS làm khung sườn cơ bản.

- Giai đoạn đầu có 5 khối.
- Giai đoạn trung gian có 11 khối.
- Giai đoạn cuối có 17 khối.
- Kích thước không gian toán tử ứng với từng giai đoạn là $8, 5, 3$.

Đề xuất của PC-DARTS[5]

DARTS cần rất nhiều bộ nhớ và tài nguyên tính toán vì cố gắng huấn luyện một mạng lớn (gọi là siêu mạng) và đồng thời tìm kiếm kiến trúc tối ưu.

Để đạt được hiệu quả tìm kiếm nhanh chóng, giải thuật PC-DARTS được đề xuất:

- Thay vì tìm kiếm toàn bộ siêu mạng, ta chỉ lấy mẫu và tìm kiếm trên một phần nhỏ của nó. Điều này giúp giảm bớt sự dư thừa khi khám phá và tiết kiệm tài nguyên.
- Chọn ra một tập con các kênh trong siêu mạng để tìm kiếm, trong khi bỏ qua những kênh khác. Tuy nhiên, việc lựa chọn một phần của siêu mạng có thể tạo ra một số vấn đề. Để giải quyết vấn đề này, sử dụng *chuẩn hóa cạnh* (edge normalization).

[5]PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search

Kết nối một phần kênh

Với DARTS, trong mỗi khối có $|\mathcal{O}|$ toán tử và đầu ra tương ứng được lưu tại mỗi nút, khiến cho việc sử dụng bộ nhớ tăng lên theo tỷ lệ của số lượng toán tử $|\mathcal{O}|$. Để phù hợp với GPU, người ta phải giảm kích thước lô (batch) trong quá trình tìm kiếm, làm chậm tốc độ và có thể làm giảm ổn định cũng như tính chính xác của việc tìm kiếm.

Xét kết nối từ x_i đến x_j. Ta xác định một lớp phủ S_{ij} gán 1 cho các kênh được chọn và 0 cho các kênh còn lại. Các kênh được chọn được gửi vào tính toán hỗn hợp của $|\mathcal{O}|$ toán tử, trong khi các kênh còn lại bỏ qua các toán tử này, tức là:

$$\overline{o}_{ij}^{\mathrm{PC}} = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_{ij}(o))}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{ij}(o'))} o(S_{ij} x_i) + (1 - S_{ij}) x_i$$

Tỷ lệ của các kênh được chọn là $\frac{1}{K}$. Bằng cách thay đổi K, chúng ta có thể đánh đổi giữa độ chính xác của tìm kiếm kiến trúc (K nhỏ) và hiệu quả (K lớn).

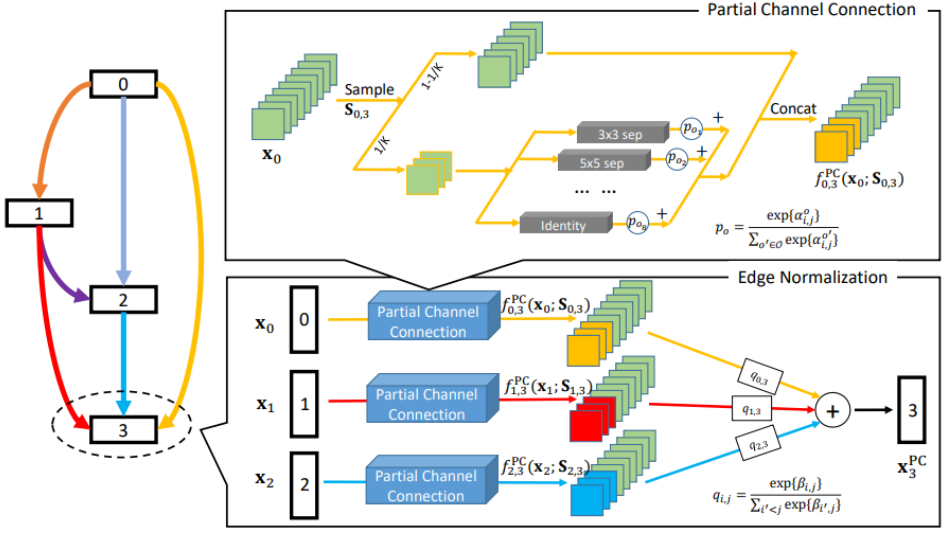
Hiệu quả tìm kiếm

Bộ nhớ phụ của việc tính toán $\overline{o}_{ij}^{\text{PC}}$ giảm đi K lần, do đó có thể sử dụng kích thước lô (batch) lớn hơn, tăng tính ổn định cho việc tìm kiếm.

Xem xét tác động của việc lấy mẫu kênh trong tìm kiếm:

- **Tích cực:** Ít thiên vị hơn trong việc chọn toán tử. Trong thực tế, các toán tử không sử dụng trọng số thường được ưu tiên ở giai đoạn đầu của quá trình tìm kiếm vì chúng không cần phải học trọng số và cung cấp đầu ra ổn định hơn. Ngược lại, các toán tử sử dụng trọng số cần thời gian để tối ưu hóa trọng số của chúng, làm cho đầu ra trở nên không ổn định.
- **Tiêu cực:** Trong một khối (cell), mỗi nút đầu ra cần phải chọn hai nút đầu vào từ tiền nhiệm của nó, điều này là dựa trên DARTS gốc. Những tham số kiến trúc này được tối ưu hóa thông qua việc lấy mẫu ngẫu nhiên các kênh. Nhưng vì sự ngẫu nhiên này, kết nối tối ưu có thể không ổn định theo thời gian.

Khắc phục hạn chế đó, ta sử dụng *chuẩn hóa cạnh*.



Hình 6: Minh họa giải thuật PC-DARTS

Chuẩn hóa cạnh

Theo DARTS, xét một khối, mỗi đầu ra x_j cần lấy đầu vào từ 2 trong số các nút trước nó $\{x_0, x_1, \ldots, x_{j-1}\}$ được đánh trọng số bởi $\max_{o \in \mathcal{O}} \alpha_{1,j}(o), \max_{o \in \mathcal{O}} \alpha_{2,j}(o), \ldots, \max_{o \in \mathcal{O}} \alpha_{j-1,j}(o)$.

Để giảm sự không ổn định khi lấy mẫu kênh ngẫu nhiên, ta chuẩn hóa cạnh bằng cách đánh trọng số β_{ij}. Khi đó:

$$x_j^{\mathsf{PC}} = \sum_{i<j} \frac{\beta_{ij}}{\sum_{i'<j} \beta_{i'j}} \overline{o}_{ij}(x_i)$$

Kết thúc quá trình tìm kiếm, kết nối trên cạnh (i, j) được xác định bởi cả $\{\alpha_{ij}(o)\}$ và $\{\beta_{ij}\}$. Bằng cách nhân hệ số:

$$\frac{\beta_{ij}}{\sum_{i'<j} \beta_{i'j}} \times \frac{\exp(\alpha_{ij}(o))}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{ij}(o'))}$$