Darts详解

Darts目的

- 传统的NAS是基于**离散**空间上的黑盒优化过程
- 强化学习、进化算法、贝叶斯优化，**都不能**用Loss的梯度更新网络架构，只能间接优化生成子网络模型的控制器Controller RNN
- Darts把搜索空间弱化为连续的空间结构，网络模型以**可微分**参数化的形式实现，可用**梯度下降**进行性能优化

- 参考链接：

[1] 视频讲解

[2] 论文+代码(tensorflow)

[3] 论文讲解

Darts搜索基本思想

(a) 搜索问题

灰色小方块：cell中的node，也叫节点

方块间的边：可能的操作，例如池化、卷积等，图中共3种，这些操作本身也有参数，称为**模型参数w**

(b) 搜索空间连续松弛化

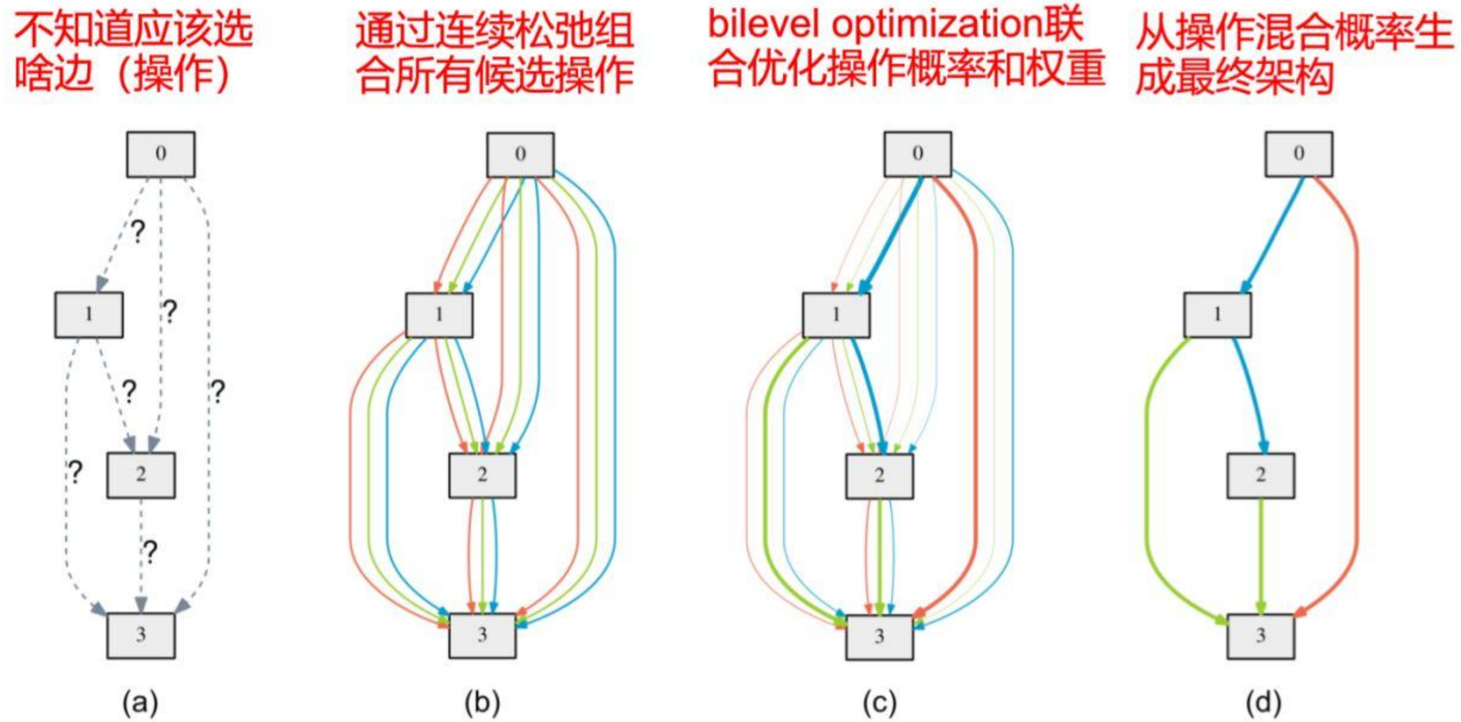
每个节点和**所有的**前驱节点相连，两个块之间所有可能的操作**都赋权重**，称为**架构参数α**，真实权重 softmax(α)

(c) 联合优化

通过梯度下降对 α 和 w 进行优化

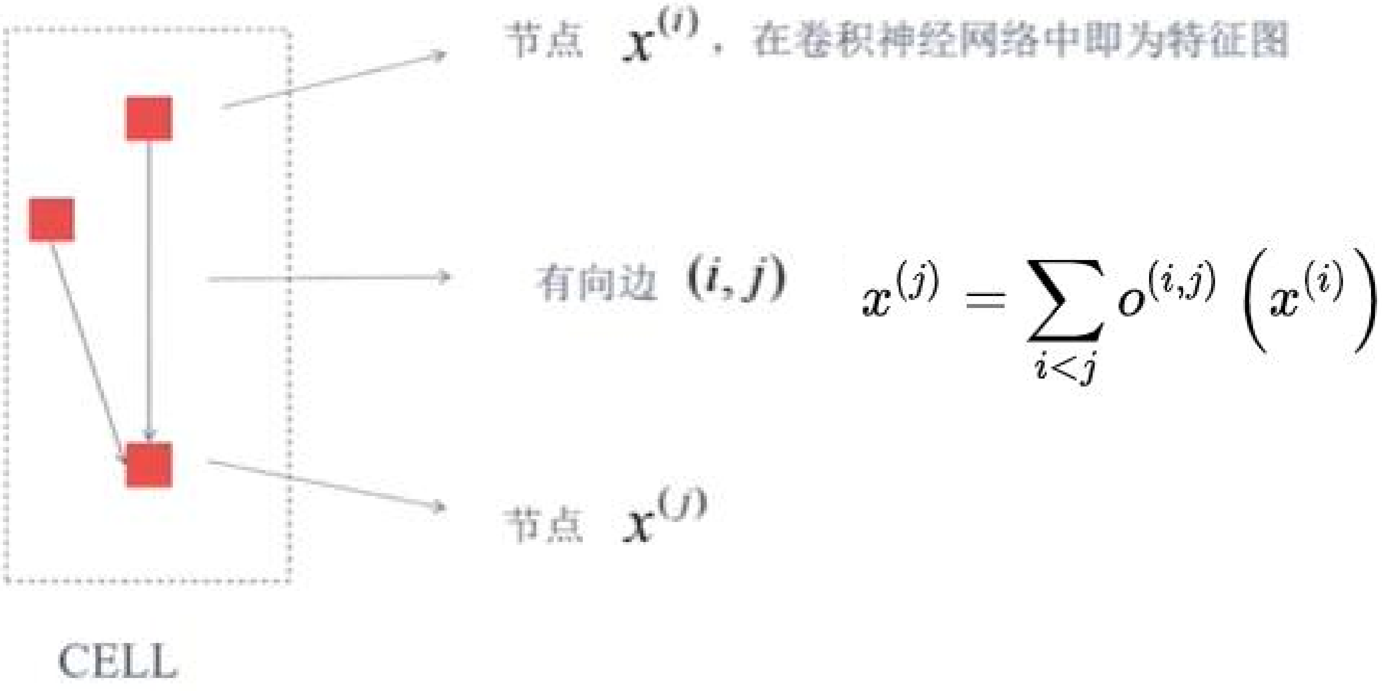
(d) 选择架构

每个节点取argmax 即**权重最大**的操作



Darts的搜索空间-cell定义

- 目标：**搜索cell的结构**，然后用cell构建CNN或者RNN。
- Cell：由N个节点的有序序列组成的有向无环图，下图是N=3的例子
- 每个中间节点（特征图）都是由有向无环图中**所有**的前继节点计算



- ✓ 有向边$o^{(i,j)}$表示节点i与节点j之间进行转换的相互关联的操作（卷积，池化，正则化等）
- ✓ 为了表示某些节点之间是没有任何联系的，因此此处引入了Zero-Operation
- ✓ Node i是由所有小于它的Node j经过操作而得到

$$x^{(j)}=\sum_{i<j}o^{(i,j)}\left(x^{(i)}\right)$$

Darts的搜索空间-CNN network定义

- CIFAR-10定义的**CNN网络结构**
- 1个Network包括8个cell，cell分为reduction/normal cell，分别共享架构参数α-reduction和α-normal
- network的1/3和2/3处是reduction cell，即第3和第6个cell。其他为normal cell
- 1个cell包括7个nodes

  2个input node: 前2个cell的输出节点

  4个intermediate node: 与所有前驱相连的节点

  1个output node: 对4个intermediate node进行concat，原来输入的通道是C，输出之后变成4C

$$x^{(j)} = \sum_{i<j} o^{(i,j)}\left(x^{(i)}\right)$$

- **代码讲解见：pt.darts**
- **search_cell.py**：定义了cell的结构和前向传播操作
- **search_cnn.py**：定义了network的结构和操作

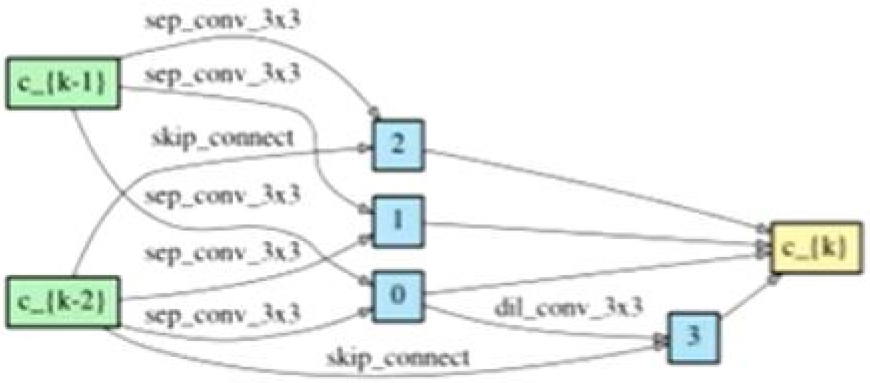


Figure 4: Normal cell learned on CIFAR-10.

Darts的搜索空间-cell中边的候选操作

cell中边的**8种**可选操作： 3×3极大值池化，3×3均值池化，恒等， 3×3深度可分离卷积，5×5深度可分离卷积，3×3空洞深度可分离卷积，5×5空洞深度可分离卷积，0操作（两个节点无连接）

代码见 ops.py

```python
PRIMITIVES = [
    'max_pool_3x3',
    'avg_pool_3x3',
    'skip_connect', # identity
    'sep_conv_3x3',
    'sep_conv_5x5',
    'dil_conv_3x3',
    'dil_conv_5x5',
    'none'
]
```

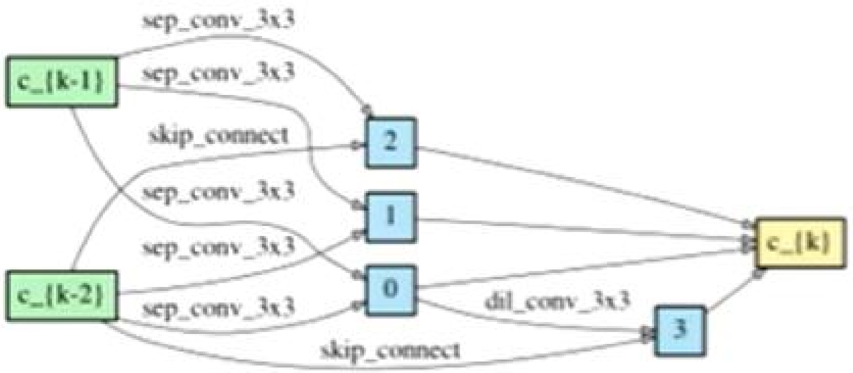
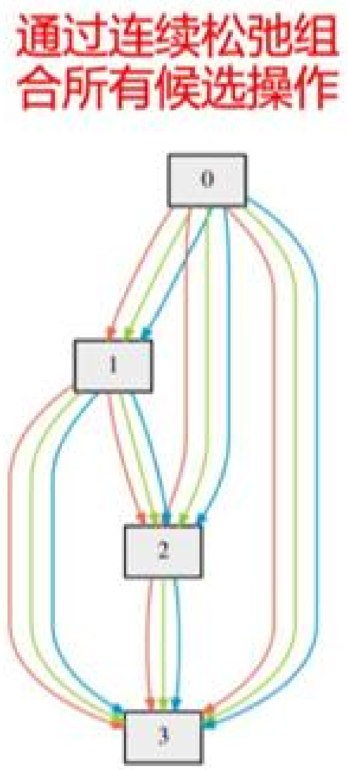


Figure 4: Normal cell learned on CIFAR-10.

如何把离散选择边的操作弱化为连续空间-softmax

- 把整个搜索空间看成supernet，学习最优的subnet。
- 传统的NAS，在候选操作中，**只能**选1个操作，这种选择**离散不可导**
- Darts把选择单一操作的步骤松弛化为**softmax的所有操作子权值叠加**
- **架构参数 α** 是第 i 个特征图到第 j 个特征图之间操作的权重。如果权重=0，表示不需要这个操作



Softmax操作，即 $$\frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o'\in\mathcal{O}}\exp\left(\alpha_{o'}^{(i,j)}\right)}$$

```
PRIMITIVES = [
    'max_pool_3x3',
    'avg_pool_3x3',
    'skip_connect',
    'sep_conv_3x3',
    'sep_conv_5x5',
    'dil_conv_3x3',
    'dil_conv_5x5',
    'none'
]
```

$$\bar{o}^{(i,j)}(x)=\sum_{o\in\mathcal{O}}\frac{\exp\left(\alpha_o^{(i,j)}\right)}{\sum_{o'\in\mathcal{O}}\exp\left(\alpha_{o'}^{(i,j)}\right)}o(x)$$

混合操作MixOp：操作集的每个操作都会处理每个节点的特征图，再对所有操作得到的结果加权求和

Darts的优化目标

优化目标：**验证集上的损失函数**
找到**最优的架构参数α**使Lval最小，即Lval的式子
找到**最优的模型参数w**使Ltrain最小，即w*的式子

$$\begin{array}{ll} \min_{\alpha} & \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right) \\ \text{s.t.} & w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha) \end{array}$$

（星号上标代表最优的　　s.t. subject to　满足.. 条件，受..约束）

每次更新架构参数α都理应重新训练模型的权重w*，求出**最优w的训练代价高**
第一个式子要优化α，但要w*，想优化w又跟架构参数α有关，所以是**两级最优化问题**

如何求梯度？梯度近似

$$\min_\alpha \quad \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right)$$
$$\text{s.t.} \quad w^*(\alpha) = \text{argmin}_w \, \mathcal{L}_{train}(w, \alpha)$$

$$\boxed{\nabla_\alpha \mathcal{L}_{val}\left(w(\alpha), \alpha\right) \approx \nabla_\alpha \mathcal{L}_{val}\left(w - \xi \nabla w \mathcal{L}_{train}(w, \alpha), \alpha\right)}$$

ξ是模型参数w的学习率

这种近似在架构于训练集上达到局部极值点（ $\nabla_\omega \mathcal{L}_{train}(\omega, \alpha) = 0$ ）时， $\omega = \omega^*(\alpha)$

所以这样近似是有道理的

也就是说，这种近似实际上是用 $w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)$ （训练集上对权重执行一次梯度下降）来近似最优权重 $w^*(\alpha)$ 。

核心思想：每次更新α 让w在Ltrain上做一次single training step，进行**一步优化**，近似w*，不需要多次训练求出最优w* （这种方法**在元学习中用过**）

NAS训练过程：

1. 在验证集*Lval*损失上梯度下降更新架构参数*α*
2. 在训练集*Lval*损失上梯度下降更新模型参数*w*

近似后的梯度如何求解

具体公式推导参考知乎：https://zhuanlan.zhihu.com/p/73037439

$$\min_\alpha \quad \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right)$$
$$\text{s.t.} \quad w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha)$$

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega^*(\alpha), \alpha\right)$$
$$\approx \nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$

符号
替换

记为 $\nabla_\alpha f\left(g_1(\alpha), g_2(\alpha)\right)$

$$f(\cdot,\cdot) = \mathcal{L}_{val}(\cdot,\cdot)$$
$$g_1(\alpha) = \omega - \xi \nabla_\omega \mathcal{L}_{\text{train}}(\omega, \alpha)$$
$$g_2(\alpha) = \alpha$$

① 复合函数求导公式：

$$\nabla_\alpha f\left(g_1(\alpha), g_2(\alpha)\right)$$
$$= \nabla_\alpha g_1(\alpha) \cdot D_1 f\left(g_1(\alpha), g_2(\alpha)\right) + \nabla_\alpha g_2(\alpha) \cdot D_2 f\left(g_1(\alpha), g_2(\alpha)\right)$$

$$\nabla_\alpha g_1(\alpha) = -\xi \nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha)$$
$$\nabla_\alpha g_2(\alpha) = 1$$

$$D_1 f\left(g_1(\alpha), g_2(\alpha)\right) = \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)$$
$$D_2 f\left(g_1(\alpha), g_2(\alpha)\right) = \nabla_\alpha \mathcal{L}_{val}(\omega', \alpha)$$

①

$$w' = w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)$$

复合函数求导，兼顾2个α

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$
$$= \nabla_\alpha \mathcal{L}_{val}(\omega', \alpha) - \underline{\xi \nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha) \cdot \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)}$$

只对函数里，第2个α求导

w’变成了常数，而不是关于α的复合函数

②

有这个公式后，现在可以求出目标函数的梯度

② 泰勒公式：

$$f(x_0 + h) = f(x_0) + \frac{f'(x_0)}{1!} h + \ldots$$

相减：

$$f(x_0 + hA) = f(x_0) + \frac{f'(x_0)}{1!} hA + \ldots$$
$$f(x_0 - hA) = f(x_0) - \frac{f'(x_0)}{1!} hA + \ldots$$

$$f'(x_0) \cdot A \approx \frac{\underline{f(x_0 + hA)} - f(x_0 - hA)}{2h}$$

A 换成 $\nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)$

h 换成 ϵ

x_0 换成 w

f 换成 $\nabla_\alpha \mathcal{L}_{train}(\cdot,\cdot)$

$$\nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha) \cdot \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha) \approx \frac{\nabla_\alpha \mathcal{L}_{train}(\omega^+, \alpha) - \nabla_\alpha \mathcal{L}_{train}(\omega^-, \alpha)}{2\epsilon}$$

其中，$\omega^\pm = \underline{\omega \pm \epsilon \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)}$。 $\epsilon = 0.01 / \|\nabla_{w'} \mathcal{L}_{val}(w', \alpha)\|_2$ ）

$\underline{f(x_0 \pm hA)}$

根据经验取值

实际上这种有限差分近似只需要**对梯度进行两次前向传播**，以及**对架构进行两次反向传播**。其计算复杂度也会从 $O(|\alpha||w|)$ 降至 $O(|\alpha| + |w|)$

Darts训练算法

- 混合操作mixOp表示所有侯选边的混合计算结果
  即softmax(权重 α) * 操作结果，再求和

整体的训练算法在**代码search.py**中
- 先在验证集上更新**架构参数α**，需要在训练集上模拟一步优化计算w'，见architect.py
- 在更新后的α基础上，在训练集上更新**模型参数 w**，见search.py

Algorithm 1: DARTS – Differentiable Architecture Search

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i,j)
while *not converged* **do**
1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w,\alpha),\alpha)$
   ($\xi = 0$ if using first-order approximation)
2. Update weights w by descending $\nabla_w \mathcal{L}_{train}(w,\alpha)$

Derive the final architecture based on the learned α.

第一步：更新架构参数α

采用梯度下降来更新α，代码见 architect.py

$$
\begin{aligned}
\min_\alpha \quad & \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right) \\
\text{s.t.} \quad & w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha)
\end{aligned}
$$

$$
\boxed{w' = w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)}
$$

① 计算w' 见右边函数

```python
def virtual_step(self, trn_X, trn_y, xi, w_optim):
    """
    Compute unrolled weight w' (virtual step)

    根据公式计算 w' = w - ξ * dw Ltrain(w, α)
    Monmentum公式:  dw Ltrain -> v * w_momentum + dw Ltrain + w_weight_decay * w
    -> m + g + 正则项

    Step process:
    1) forward
    2) calc loss
    3) compute gradient (by backprop)
    4) update gradient

    Args:
        xi: learning rate for virtual gradient step (same as weights lr)  即公式中的 ξ
        w_optim: weights optimizer 用来更新 w 的优化器
    """
    # forward & calc loss
    loss = self.net.loss(trn_X, trn_y) # L_trn(w)

    # compute gradient 计算  dw L_trn(w) = g
    gradients = torch.autograd.grad(loss, self.net.weights())

    # do virtual step (update gradient)
    # below operations do not need gradient tracking
    with torch.no_grad():
        # dict key is not the value, but the pointer. So original network weight have to
        # be iterated also.
        for w, vw, g in zip(self.net.weights(), self.v_net.weights(), gradients):
            # m = v * w_momentum  用的就是Network进行w更新的momentum
            m = w_optim.state[w].get('momentum_buffer', 0.) * self.w_momentum

            # 做一步momentum梯度下降后更新得到 w' = w - ξ * (m + dw Ltrain(w, α) + 正则项 )
            vw.copy_(w - xi * (m + g + self.w_weight_decay*w))

        # synchronize alphas
        for a, va in zip(self.net.alphas(), self.v_net.alphas()):
            va.copy_(a)
```

第一步：更新架构参数α

$$\min_\alpha \quad \mathcal{L}_{val}\left(w^*(\alpha), \alpha\right)$$
$$\text{s.t.} \quad w^*(\alpha) = \operatorname{argmin}_w \mathcal{L}_{train}(w, \alpha)$$

$$w' = w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)$$

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega^*(\alpha), \alpha\right)$$
$$\approx \nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$
$$= \boxed{\nabla_\alpha \mathcal{L}_{val}(\omega', \alpha) - \xi \nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha) \cdot \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)}$$

② 计算目标函数关于α的近似梯度
见右边函数

```
def unrolled_backward(self, trn_X, trn_y, val_X, val_y, xi, w_optim):
    """ Compute unrolled loss and backward its gradients
    Args:
        xi: learning rate for virtual gradient step (same as net lr)
        w_optim: weights optimizer - for virtual step
    """
    # do virtual step (calc w`)
    self.virtual_step(trn_X, trn_y, xi, w_optim)

    # calc unrolled loss
    loss = self.v_net.loss(val_X, val_y) # L_val(w', α)  在使用w', 新alpha的net上计算损失值

    # compute gradient
    v_alphas = tuple(self.v_net.alphas())
    v_weights = tuple(self.v_net.weights())
    v_grads = torch.autograd.grad(loss, v_alphas + v_weights)
    dalpha = v_grads[:len(v_alphas)]    # dα L_val(w', α)   梯度近似后公式第一项
    dw = v_grads[len(v_alphas):]        # dw' L_val(w', α)  梯度近似后公式第二项的第二个乘数

    hessian = self.compute_hessian(dw, trn_X, trn_y)        # 梯度近似后公式第二项

    # update final gradient = dalpha - xi*hessian
    with torch.no_grad():
        for alpha, da, h in zip(self.net.alphas(), dalpha, hessian):
            alpha.grad = da - xi*h    # 求出了目标函数的近似梯度值
```

第一步：更新架构参数α

③ 计算近似梯度中第二项
采用泰勒展开后的近似公式

$$\nabla^2_{\alpha,\omega}\mathcal{L}_{train}(\omega,\alpha)\cdot\nabla_{\omega'}\mathcal{L}_{val}(\omega',\alpha)\approx\boxed{\frac{\nabla_\alpha\mathcal{L}_{train}(\omega^+,\alpha)-\nabla_\alpha\mathcal{L}_{train}(\omega^-,\alpha)}{2\epsilon}}$$

其中， $\omega^{\pm}=\omega\pm\epsilon\nabla_{\omega'}\mathcal{L}_{val}(\omega',\alpha)$ 。 $\epsilon=0.01/\|\nabla_{w'}\mathcal{L}_{val}(w',\alpha)\|_2$ ）

```python
def compute_hessian(self, dw, trn_X, trn_y):
    """
    求经过泰勒展开后的第二项的近似值
    dw = dw` { L_val(w`, alpha) }  输入里已经给了所有预测数据的dw
    w+ = w + eps * dw
    w- = w - eps * dw
    hessian = (dalpha { L_trn(w+, alpha) } - dalpha { L_trn(w-, alpha) }) / (2*eps)    [1]
    eps = 0.01 / ||dw||
    """
    norm = torch.cat([w.view(-1) for w in dw]).norm()   # 把每个 w 先拉成一行，然后把所有的 w 摞起来，变成 n 行，然后求L2值
    eps = 0.01 / norm

    # w+ = w + eps * dw`
    with torch.no_grad():
        for p, d in zip(self.net.weights(), dw):
            p += eps * d        # 将model中所有的w'更新成 w+
    loss = self.net.loss(trn_X, trn_y)      # L_trn(w+)
    dalpha_pos = torch.autograd.grad(loss, self.net.alphas()) # dalpha { L_trn(w+) }

    # w- = w - eps * dw`
    with torch.no_grad():
        for p, d in zip(self.net.weights(), dw):
            p -= 2. * eps * d   # 将model中所有的w'更新成 w-,   w- = w - eps * dw = w+ - eps * dw * 2, 现在的 p 是 w+
    loss = self.net.loss(trn_X, trn_y)      # L_trn(w-)
    dalpha_neg = torch.autograd.grad(loss, self.net.alphas()) # dalpha { L_trn(w-) }

    # recover w
    with torch.no_grad():
        for p, d in zip(self.net.weights(), dw):
            p += eps * d        # 将模型的参数从 w- 恢复成 w,  w = w- + eps * dw

    hessian = [(p-n) / 2.*eps for p, n in zip(dalpha_pos, dalpha_neg)]  # 利用公式 [1] 计算泰勒展开后第二项的近似值返回
    return hessian
```

Darts训练算法

- 混合操作mixOp表示所有侯选边的混合计算结果
  即softmax(权重 α) * 操作结果，再求和

整体的训练算法在**代码search.py**中

- 先在验证集上更新**架构参数α**，需要在训练集上模拟一步优化计算w'，见architect.py
- 在更新后的α基础上，在训练集上更新**模型参数 w**，见search.py

Algorithm 1: DARTS – Differentiable Architecture Search

Create a mixed operation $\bar{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i,j)
while *not converged* **do**
  1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi\nabla_w \mathcal{L}_{train}(w,\alpha),\alpha)$
     ($\xi = 0$ if using first-order approximation)
  2. Update weights w by descending $\nabla_w \mathcal{L}_{train}(w,\alpha)$

Derive the final architecture based on the learned α.

第二步：更新模型参数w

在第一步更新后的 α 的基础上，
在训练集上梯度下降更新w

```
def train(train_loader, valid_loader, model, architect, w_optim, alpha_optim, lr, epoch):
    top1 = utils.AverageMeter()      # 保存前 1 预测正确的概率
    top5 = utils.AverageMeter()      # 保存前 5 预测正确的概率
    losses = utils.AverageMeter()    # 保存loss值

    cur_step = epoch * len(train_loader)
    writer.add_scalar('train/lr', lr, cur_step)

    model.train()

    # 每个step取出一个batch, batchsize是64（256个数据对）
    for step, ((trn_X, trn_y), (val_X, val_y)) in enumerate(zip(train_loader, valid_loader)):
        trn_X, trn_y = trn_X.to(device, non_blocking=True), trn_y.to(device, non_blocking=True)
        # 用于架构参数alpha 更新的一个batch, 使用iter(dataloader)返回的是一个迭代器, 然后可以使用next访问
        val_X, val_y = val_X.to(device, non_blocking=True), val_y.to(device, non_blocking=True)
        N = trn_X.size(0)

        # phase 2. architect step (alpha) 对应伪代码的第 1 步, 结构参数梯度下降
        alpha_optim.zero_grad() # 清除之前学到的梯度的参数
        architect.unrolled_backward(trn_X, trn_y, val_X, val_y, lr, w_optim)
        alpha_optim.step()

        # phase 1. child network step (w) 对应伪代码的第 2 步, 网络参数梯度下降
        w_optim.zero_grad()      # 清除之前学到的梯度的参数
        logits = model(trn_X)
        loss = model.criterion(logits, trn_y)   # 预测值 logits 和真实值 target 的loss
        loss.backward()          # 反向传播, 计算梯度

        # gradient clipping  梯度裁剪
        nn.utils.clip_grad_norm_(model.weights(), config.w_grad_clip)
        w_optim.step()       # 应用梯度

        prec1, prec5 = utils.accuracy(logits, trn_y, topk=(1, 5))
        losses.update(loss.item(), N)
        top1.update(prec1.item(), N)
        top5.update(prec5.item(), N)
```

训练完成后，挑选每个node最大的2个α
操作方法在 **genotypes.py/parse**函数中

NAS过程结束后，需要对构建的CNN训练
代码在**augment_cells.py和augment_cnn.py**中

CNN中每个intermediate node有2条边
MixOp操作是两条边的计算结果求和

```
def forward(self, s0, s1):
    s0 = self.preproc0(s0)
    s1 = self.preproc1(s1)

    states = [s0, s1]
    for edges in self.dag:
        s_cur = sum(op(states[op.s_idx]) for op in edges)
        states.append(s_cur)

    s_out = torch.cat([states[i] for i in self.concat], dim=1)

    return s_out
```

```
def parse(alpha, k):
    """
    根据alpha权重挑选top k条边
    parse continuous alpha to discrete gene.
    alpha is ParameterList:
    ParameterList [
        Parameter(n_edges1, n_ops),
        Parameter(n_edges2, n_ops),
        ...
    ]

    gene is list:
    [
        [('node1_ops_1', node_idx), ..., ('node1_ops_k', node_idx)],
        [('node2_ops_1', node_idx), ..., ('node2_ops_k', node_idx)],
        ...
    ]
    each node has two edges (k=2) in CNN.
    """

    gene = []
    assert PRIMITIVES[-1] == 'none' # assume last PRIMITIVE is 'none'

    # 1) Convert the mixed op to discrete edge (single op) by choosing top-1 weight edge
    # 2) Choose top-k edges per node by edge score (top-1 weight in edge)
    for edges in alpha:
        # edges: Tensor(n_edges, n_ops)
        edge_max, primitive_indices = torch.topk(edges[:, :-1], 1) # ignore 'none'
        topk_edge_values, topk_edge_indices = torch.topk(edge_max.view(-1), k)
        node_gene = []
        for edge_idx in topk_edge_indices:
            prim_idx = primitive_indices[edge_idx]
            prim = PRIMITIVES[prim_idx]
            node_gene.append((prim, edge_idx.item()))

        gene.append(node_gene)

    return gene
```

总结整体代码流程：

1. search.py：主函数入口

    构建CNN network (search_cnn.py)，包括8个cell (search_cells.py)

    用前一半data做训练集data_train，后一半data做验证集data_val，

    初始化w的优化器**SGD (momentum)**和α的优化器**Adam**，多次搜索

    每次搜索都是分batch迭代完所有data

    每个batch：先更新架构参数 α（调architect.py）

    再用data_train更新模型参数 w

    训练完成得到最优的α，通过前向传播看下data_val上效果

    每一次搜索都把最优的结构保存下来

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega^*(\alpha), \alpha\right)$$ net

$$\approx \nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$ v_net

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$

$$= \nabla_\alpha \mathcal{L}_{val}(\omega', \alpha) - \xi \nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha) \cdot \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)$$

2. architect.py：利用梯度近似、复合函数求导、泰勒展开来更新α

    先计算 w'，使用data_train来训练一步，用得到的梯度通过momentum梯度下降计算

    ```
    w' = w - ξ * (m + dw Ltrain(w, α) + 正则项 )
    ```

    （在v_net上做一步优化，计算出w'，net上w暂时不变）

    再计算 L_val(w', α)，算出 dα L_val(w', α) 和 dw' L_val(w', α)

    然后根据泰勒展开求出公式第二项的近似值

    最终求出目标函数关于α的梯度，更新到net上

简述：先固定住w，用所有data_train做一步优化，计算w'，在w'基础上更新α

在更新后α的基础上，真实地训练一步w，然后再回到上面

$$\nabla_\alpha \mathcal{L}_{val}\left(\omega^*(\alpha), \alpha\right)$$
$$\approx \nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$
$$w' = w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha)$$
$$\nabla_\alpha \mathcal{L}_{val}\left(\omega - \xi \nabla_\omega \mathcal{L}_{train}(\omega, \alpha), \alpha\right)$$
$$= \nabla_\alpha \mathcal{L}_{val}(\omega', \alpha) - \xi \nabla^2_{\alpha,\omega} \mathcal{L}_{train}(\omega, \alpha) \cdot \nabla_{\omega'} \mathcal{L}_{val}(\omega', \alpha)$$

- 一阶近似：$\xi = 0$，梯度等价于 $\nabla_\alpha \mathcal{L}_{val}(w, \alpha)$， $w = w^*(\alpha)$, 即α与w相互独立，实验证明效果不好
- 二阶近似：$\xi > 0$，效果较好

学习率对于网络收敛性的影响

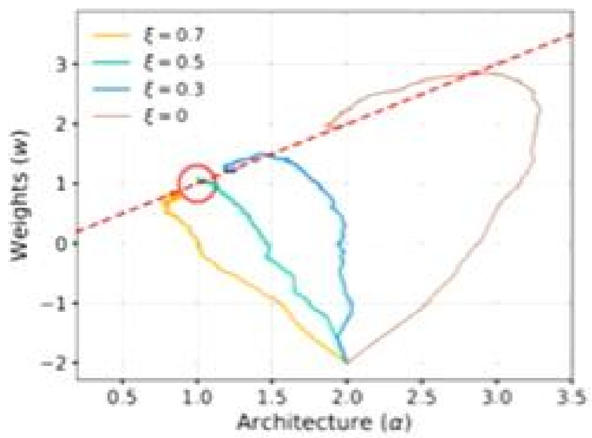


Figure 2: Learning dynamics of our iterative algorithm when $\mathcal{L}_{val}(w, \alpha) = \alpha w - 2\alpha + 1$ and $\mathcal{L}_{train}(w, \alpha) = w^2 - 2\alpha w + \alpha^2$, starting from $(\alpha^{(0)}, w^{(0)}) = (2, -2)$. The analytical solution for the corresponding bilevel optimization problem is $(\alpha^*, w^*) = (1, 1)$, which is highlighted in the red circle. The dashed red line indicates the feasible set where constraint equation 4 is satisfied exactly (namely, weights in w are optimal for the given architecture α). The example shows that a suitable choice of ξ helps to converge to a better local optimum.

separable convolutions	dilated convolutions	max pooling	average pooling	identity	Zero
3×3、5×5	3×3、5×5	3×3	3×3		

在CIFAR-10上搜索到的Cell为：

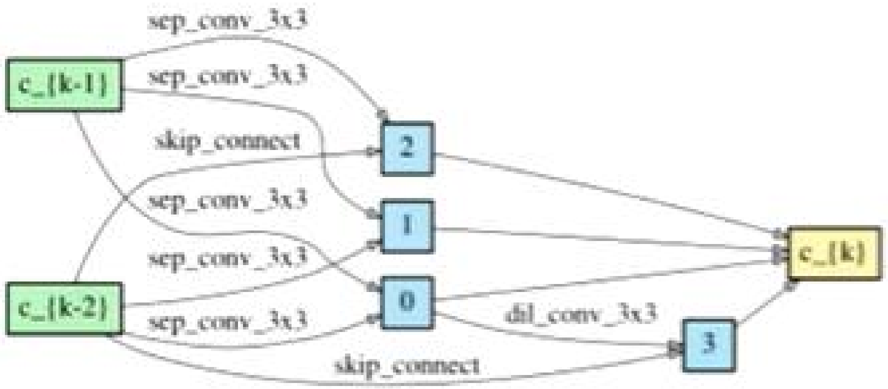


Figure 4: Normal cell learned on CIFAR-10.

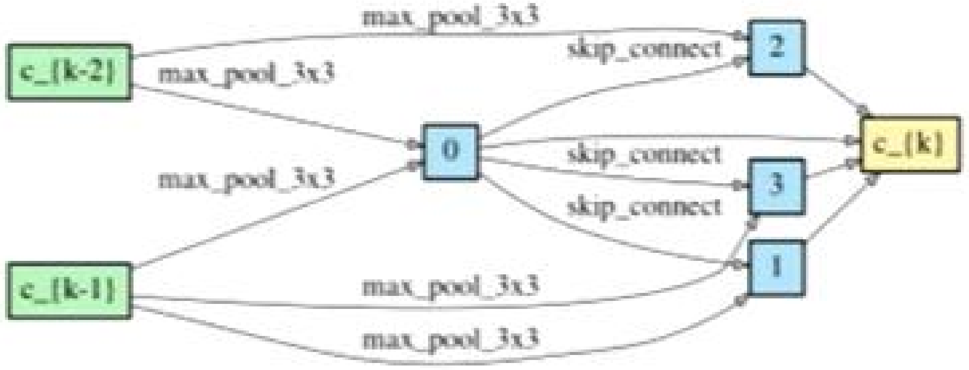


Figure 5: Reduction cell learned on CIFAR-10.

由于要同时训练所有的架构，所以Cell叠加的个数不能太大，也不能在大的数据集上进行搜索。

图中也可以看出CNN中network的每个intermediate node前驱有2条边