# A APPENDIX

## A.1 EXPERIMENTAL SETUP

**NAS-Bench-201.** We ran the DARTS and GDAS search for 50 epochs on CIFAR-10 with batch size 64, using NAS-Bench-201's standard split of data points into 50% test and 50% training/validation for the search. Following Dong & Yang (2020), as weight optimizer we use SGD with momentum 0.9, weight decay of $3 \times 10^{-4}$ and initial learning rate 0.025 with cosine annealing (Loshchilov & Hutter, 2017) to 0.001. As optimizer for the architectural weights, we use Adam Kingma & Ba (2014) with learning rate $3 \times 10^{-4}$ and weight decay of $0.001$. For GDAS we set the initial $\tau = 10$ and reduce linearly to 0.1 and trained it for 250 epochs. For regularized evolution (Real et al., 2019) and Random Search, we set the number of function evaluations to 1000, population size to 100 and sample size 10.

**DARTS search space.** We also run DARTS and GDAS on this search space using the same hyperparameters as for NAS-Bench-201 which are also the same as used in (Liu et al., 2019b). For training the final model found by the optimizer we used the pipeline from Liu et al. (2019b) with their hyperparameters (Cutout, drop path, auxiliary towers) using the full 50000 images training set. The final architecture found by GDAS includes more parameters which is why we set the batch size for this to 72 instead of 96. A single search took 7.5 hours, evaluation 12 to 51 hours on a single GPU. Following (Liu et al., 2019b; Dong & Yang, 2019) we pick the best out of four searches after low-fidelity evaluation on the validation set and train it from scratch for longer and with more initial channels and stacked cells.

**Hierarchical search space.** We ran DARTS and GDAS for 10 epochs with batch size 10 on CIFAR-10 using the search space as Liu et al. (2018). Apart from this we used the same hyperparameters as for DARTS. The resulting search took 66 and 7 hours on a single GPU for DARTS and GDAS, respectively, in contrast to the dramatically longer runs for 1.5 days on 200 GPUs in Liu et al. (2018). Different to Liu et al. (2018), we trained the found cells and motifs on 50000 images for 600 epochs using batch size of 32 and SGD with initial learning rate of 0.025 with cosine annealing to 0.001. We use cutout and drop path with probability of 0.2. The evaluation took 23 hours on a single GPU.

## A.2 NASLIB CODE SNIPPETS

Here we show more code snippets from NASLib.

```
class HierarchicalSearchSpace(Graph):
  OPTIMIZER_SCOPE = [
    "stage_1",
    "stage_2",
    "stage_3"]

  def __init__(self):
    super().__init__()
    level2_motifs = []  # 6 level-2 motifs
    for j in range(6):
      motif = Graph()
      motif.name = "motif{}".format(j)
      motif.add_nodes_from([1, 2, 3, 4, 5])
      motif.add_edges_densly()
      level2_motifs.append(motif)

    cell = Graph()       # 1 level-3 motif
    cell.name = "cell"
    cell.add_nodes_from([1, 2, 3, 4, 5, 6])
    cell.add_edges_densly()

    cells = []
    channels = [16, 32, 64]
    for c, s in zip(channels,
                    self.OPTIMIZER_SCOPE):
      cell_i = cell.copy()
      # place level 2 motifs as op
      cell_i.update_edges(
        update_func=lambda current_edge_data:
            _set_motifs(current_edge_data,
                motifs=level2_motifs, c=c),
        private_edge_data=True
      )
      cell_i.set_scope(s, recursively=True)
      # place level 1 motifs, i.e. primitives
      cell_i.update_edges(
        update_func=lambda current_edge_data:
            _set_cell_ops(current_edge_data,
                c, stride=1),
        scope=s,
        private_edge_data=True
      )
      cells.append(cell_i)

    # Macro graph omitted for brevity
```

Snippet 5: The hierarchical search space written using the language in NASLib. The macro graph definition is omitted for brevity. The search space is entirely defined as instances of graphs.

```python
import torch.nn as nn
from naslib.search_spaces.core import primitives as ops
from naslib.search_spaces.core.graph import Graph, EdgeData
from naslib.search_spaces.core.primitives import AbstractPrimitive
from .primitives import ResNetBasicblock

class NasBench201SeachSpace(Graph):

    OPTIMIZER_SCOPE = [
        "stage_1",
        "stage_2",
        "stage_3",
    ]

    QUERYABLE = True

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

        # Cell definition
        cell = Graph()
        cell.name = "cell"
        cell.add_node(1)             # Input node
        cell.add_node_from([2, 3])   # Intermediate nodes
        cell.add_node(4)             # Output node
        cell.add_edges_densly()      # Edges

        # Macro graph definition
        self.name = "makrograph"

        total_num_nodes = 20
        self.add_nodes_from(range(1, total_num_nodes+1))
        self.add_edges_from([(i, i+1) for i in range(1, total_num_nodes)])

        # operations at the edges
        channels = [16, 32, 64]

        self.edges[1, 2].set('op', ops.Stem(channels[0]))    # preprocessing
        # stage 1
        for i in range(2, 7):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_1'))
        # stage 2
        self.edges[7, 8].set('op', ResNetBasicblock(channels[0], channels[1], stride=2))
        for i in range(8, 13):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_2'))
        # stage 3
        self.edges[13, 14].set('op', ResNetBasicblock(channels[1], channels[2], stride=2))
        for i in range(14, 19):
            self.edges[i, i+1].set('op', cell.copy().set_scope('stage_3'))
        # post-processing
        self.edges[19, 20].set('op', ops.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Flatten(),
            nn.Linear(channels[-1], self.num_classes)
        ))

        # set the ops at the cells (channel dependent)
        for c, scope in zip(channels, self.OPTIMIZER_SCOPE):
            self.update_edges(
                update_func=lambda current_edge_data: _set_cell_ops(current_edge_data, C=c),
                scope=scope,
                private_edge_data=True
            )


    def query(self, metric=None, dataset=None, path=None):
        # query logic here


def _set_cell_ops(current_edge_data, C):
    current_edge_data.set('op', [
        ops.Identity(),
        ops.Zero(stride=1),
        ops.ReLUConvBN(C, C, kernel_size=3),
        ops.ReLUConvBN(C, C, kernel_size=1),
        ops.AvgPool1x1(kernel_size=3, stride=1),
    ])
```

Snippet 6: The complete Nasbench 201 cell search space written using the language in NASLib. The query implementation is omitted for brevity. The search space is entirely defined as graph object.

```
class DARTSOptimizer(MetaOptimizer):

    @staticmethod
    def add_alphas(current_edge_data):
        len_primitives = len(current_edge_data.op)
        alpha = Parameter(1e-3 * torch.randn(size=[len_primitives], requires_grad=True))
        current_edge_data.set('alpha', alpha, shared=True)

    @staticmethod
    def update_ops(current_edge_data):
        primitives = current_edge_data.op
        current_edge_data.set('op', MixedOp(primitives))

    def adapt_search_space(self, search_space, scope=None):
        graph = search_space.clone()    # We are going to modify the search space

        if not scope:
            scope = graph.OPTIMIZER_SCOPE   # use the search space default one

        # 1. add alphas
        graph.update_edges(self.add_alphas, scope, private_edge_data=False)
        # 2. replace primitives with mixed_op
        graph.update_edges(self.update_ops, scope, private_edge_data=True)

        for alpha in graph.get_all_edge_data('alpha'):
            self.architectural_weights.append(alpha)

        graph.parse()

        # Init optimizers
        self.arch_optimizer = self.arch_optimizer(self.architectural_weights.parameters(),
            lr=self.config.arch_learning_rate, betas=(0.5, 0.999),
            weight_decay=self.config.arch_weight_decay)
        self.op_optimizer = self.op_optimizer(graph.parameters(),
            lr=self.config.learning_rate, momentum=self.config.momentum,
            weight_decay=self.config.weight_decay)

        graph.train()
        self.graph = graph
        self.scope = scope

    def step(self, data_train, data_val):
        input_train, target_train = data_train
        input_val, target_val = data_val

        # Update architecture weights
        self.arch_optimizer.zero_grad()
        logits_val = self.graph(input_val)
        val_loss = self.loss(logits_val, target_val)
        val_loss.backward()
        clip_grad_norm_(self.architectural_weights.parameters(), self.grad_clip)
        self.arch_optimizer.step()

        # Update op weights
        self.op_optimizer.zero_grad()
        logits_train = self.graph(input_train)
        train_loss = self.loss(logits_train, target_train)
        train_loss.backward()
        clip_grad_norm_(self.graph.parameters(), self.grad_clip)
        self.op_optimizer.step()

        return logits_train, logits_val, train_loss, val_loss

    def get_final_architecture(self):
        graph = self.graph.clone().unparse()
        graph.prepare_discretization()     # e.g. darts sspace: only 2 in-edges with max alpha

        def discretize_ops(current_edge_data):
            if current_edge_data.has('alpha'):
                primitives = current_edge_data.op.get_embedded_ops()
                alpha = current_edge_data.alpha.detach().cpu()
                current_edge_data.set('op', primitives[np.argmax(alpha)])

        graph.update_edges(discretize_ops, self.scope, private_edge_data=True)
        graph.prepare_evaluation()
        graph.parse()
        return graph
```

Snippet 7: The DARTS optimizer as implemented in NASLib. Non-important aspects are ommited for brevity.

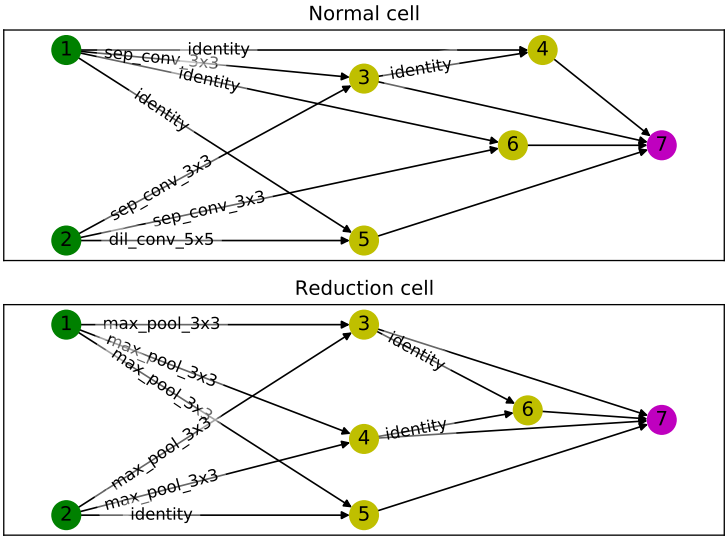


Figure 5: Normal and reduction cell found by DARTS on CIFAR-10.