به نام خدا

دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

درس تحلیل و طراحی شبکه های عصبی عمیق

تمرین شماره ۲

نام و نام خانوادگی : کیانا هوشانفر - علیرضا حسینی

شماره دانشجویی : ۸۱۰۱۰۱۳۶۱ - ۸۱۰۱۰۱۱۴۲

آذر ماه ۱۴۰۲

فهرست

مقدمه

هدف از انجام این تمرین آشنایی با دو مبحث Model Ranking و Layer-wise Evaluation است. در این تمرین با رتبه بندی مدلهای مختلف توانایی آنها را در استخراج ویژگیها ارزیابی میشود. همچنین تاثیر لایه های مختلف را در شبکه عصبی بررسی میشود.

در این تمرین مجموعه داده A مورد استفاده همان مجموعه داده Cifar 100 خواهد بود.

پیاده سازی کلاس *SEPARATION INDEX*

در این بخش هدف پیاده سازی الگوریتم SI جهت استفاده در سوال های ۱ و ۲ میباشد.

لازم به ذکر است تمامی کد های با توجه به الگوریتم ارایه شده در کلاس توسط دکتر کلهر و کلاس Kalhor_SeparationIndex نوشته شده است و تلاش شده با تغییراتی مشکلات موجود اعم از زمان اجرا و عدم پردازش برخی محاسبات بر روی cuda و .. میباشد.

در ادامه به کلاس نوشته شده و توضیحات هر بخش آن و کد های آن بخش پرداخته شده است.

بخش ۱ : Init کلاس

در این بخش که کد آن در ادامه آمده است ، دستگاه ، CPU یا GPU (CUDA) را بر اساس در دسترس بودن تنظیم می کند. سپس داده های ورودی (داده ها) و برچسب ها (برچسب) به دستگاه انتخابی منتقل می شوند. اگر flag نرمال سازی فعال باشد، داده ها با کم کردن میانگین و تقسیم بر انحراف استاندارد برای هر ویژگی نرمال می شوند، که با روش normalize_data انجام می شود. برچسب‌ها طوری تنظیم می‌شوند که از ۰ شروع شوند و نمایه‌سازی و مقایسه‌های بعدی را تسهیل می‌کنند. این روش همچنین یک ماتریس فاصله (dis_matrix) را با استفاده از torch.cdist محاسبه می‌کند، که فاصله‌های اقلیدسی را به صورت زوجی بین نقاط داده محاسبه می‌کند. مورب این ماتریس با یک عدد بزرگ پر شده است تا هنگام مقایسه نقاط داده با خود، از فاصله صفر جلوگیری شود. همچنین در ادامه سایر موارد مورد نیاز نیز تعریف میشود.

```
class ARH_SeparationIndex:
    def __init__(self, data, label, normalize=False):
        """
        Initialize the ARH_SeparationIndex class.

        Args:
            data (Tensor): The input features, a tensor of shape (n_data,
n_feature).
            label (Tensor): The labels for the data, a tensor of shape (n_data,).
            normalize (bool, optional): Whether to normalize the data. Defaults
to False.
        """
        # Set up the device for CUDA support
        self.device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
        # Initialize class attributes
        self.normalize = normalize
        self.data = data.to(self.device)
        self.label = label.to(self.device)

        # Normalize data if required
        if normalize:
            self.normalize_data()
            print('Data has been normalized')

        # Adjust labels and calculate necessary statistics
        self.label_min = round(torch.min(self.label).detach().item())
        self.label = (self.label - self.label_min).long()
        self.big_number = 1e10
        # Compute distance matrix
        self.dis_matrix = torch.cdist(self.data, self.data,
p=2).fill_diagonal_(self.big_number)
        # Calculate the number of classes, data points, and features
        self.n_class = round(torch.max(self.label).detach().item()) + 1
        self.n_data = self.data.shape[0]
        self.n_feature = self.data.shape[1]
```

بخش دوم: تابع نرمال سازی

متد normalize_data وظیفه نرمال سازی داده های ویژگی را بر عهده دارد. نرمال سازی یک مرحله پیش پردازش بسیار مهم در بسیاری از الگوریتم های یادگیری ماشین است. این شامل تنظیم مقادیر مقیاس های مختلف داده ها در یک مقیاس مشترک است. این روش میانگین و انحراف استاندارد مجموعه داده را محاسبه می کند و سپس از این مقادیر برای استانداردسازی داده ها استفاده می کند. یک عدد

کوچک (1e-10) به انحراف استاندارد اضافه می شود تا از تقسیم بر صفر جلوگیری شود. (محاسبات این بخش تقریبا همانند کد پیاده سازی دکتر کلهر میباشد و تنها محاسبات بر روی GPU انجام میشود تا instance از کلاس گرفتن مدت زیادی طول نکشد.

```
def normalize_data(self):
  """
  Normalize the data by subtracting the mean and dividing by the standard
deviation.
  """
  small_number = 1e-10
  mean_data = torch.mean(self.data, dim=0)
  std_data = torch.std(self.data, dim=0) + small_number
  self.data = (self.data - mean_data) / std_data
```

بخش سوم : SI

روش Si (شاخص جداسازی) نسبت نقاط داده ای را در مجموعه داده اندازه گیری می کند که برچسب یکسانی با نزدیک ترین همسایه خود دارند. ابتدا، نزدیکترین همسایه برای هر نقطه داده با استفاده از ماتریس فاصله از پیش محاسبه شده (dis_matrix) شناسایی می شود. این از طریق تابع torch.min به دست می آید که حداقل مقدار (نزدیک ترین فاصله) و شاخص مربوطه (نزدیک ترین همسایه) را برای هر ردیف (نقطه داده) در ماتریس فاصله پیدا می کند. سپس یک حلقه روی هر نقطه داده تکرار می‌شود و برای هر یک، بررسی می‌کند که آیا برچسب آن با برچسب نزدیک‌ترین همسایه‌اش مطابقت دارد یا خیر. کتابخانه tqdm برای ارائه یک نوار پیشرفت برای این حلقه استفاده می شود و تجربه کاربر را در طول محاسبات افزایش می دهد. امتیاز نهایی SI مجموع این مقایسه ها تقسیم بر تعداد کل نقاط داده است که یک امتیاز نرمال شده بین ۰ و ۱ به دست می دهد.

برخلاف کد قبلی بر روی گیت دکتر کلهر که برای داده های Cifar میتوانستیم فقط از ۶۰ درصد داده های ترین برای محاسبه SI استفاده کنیم و حدود ۱ دقیقه هم طول میکشد به کمک انجام محاسبات بر روی GPU و بهینه سازی های انجام شده این کد بر روی تمام داده های Cifar100 بدون مشکل اجرا شده و در حدود ۱ ثانیه محاسبات را انجام میدهد.

```
def si(self):
  """
```

```
  Calculate the separation index (SI) for the dataset.

  SI measures the proportion of data points having the same label as their
nearest neighbor.

  Returns:
      float: The calculated Separation Index (SI).
  """
  _, nearest_neighbors_indices = torch.min(self.dis_matrix, dim=1)
  si_sum = 0
  for i in tqdm(range(self.n_data), desc="Calculating SI"):
      si_sum += (self.label[i] ==
self.label[nearest_neighbors_indices[i]]).float().item()
  si = si_sum / self.n_data
  return si
```

بخش چهارم : پیاده سازی High order SI

روش high_order_si مفهوم SI را گسترش می دهد تا به جای نزدیکترین همسایه، مرتبه متفاوت نزدیکترین همسایه را در نظر بگیرد. این روش دقیق تر است و می تواند بینش عمیق تری از ساختار مجموعه داده ارائه دهد. پس از مرتب‌سازی ماتریس فاصله برای یافتن نزدیک‌ترین همسایگان برای هر نقطه داده، روش بر روی هر نقطه داده تکرار می‌شود. برای هر نقطه، برچسب را با برچسب های مرتبه نزدیکترین همسایگان خود مقایسه می کند. مقایسه برای کارایی بردار شده است. حاصلضرب این مقایسه‌ها (همه باید مطابقت داشته باشند تا به امتیاز کمک کنند) برای هر نقطه داده محاسبه می‌شود و مجموع در حال اجرا نگهداری می‌شود. امتیاز نهایی با تعداد نقاط داده نرمال می شود و معیاری برای جداسازی مرتبه بالا ارائه می دهد.

در اینجا نیز از tqdm برای نمایش نوار پیشرفت استفاده شده است.

```
def high_order_si(self, order):
  """
  Calculate the high order separation index for the dataset.

  This index is a stricter version of SI, considering the first 'order' nearest
neighbors.

  Args:
      order (int): The order of separation to consider.
```

```
  Returns:
      float: The calculated high order separation index.
  """
  try:
      sorted_distances, sorted_indices = torch.sort(self.dis_matrix, 1)
      repeated_labels = self.label.expand(self.n_data, order)
      sorted_neighbor_labels = self.label[sorted_indices[:,
:order]].view(self.n_data, order)
      total_high_order_si = 0
      for idx in tqdm(range(self.n_data), desc="Computing High Order SI"):
          match_labels = (repeated_labels[idx] ==
sorted_neighbor_labels[idx]).float()
          total_high_order_si += torch.prod(match_labels)
      final_high_si = total_high_order_si / self.n_data
      return final_high_si.item()
  except RuntimeError as e:
      if "out of memory" in str(e):
          print("Insufficient CUDA memory. Consider lowering 'order' or using a
device with more GPU memory.")
      else:
          raise e
```

با توجه به کد فوق اگر cuda out of memory داشته باشیم کد error میدهد تا با دیتای کمتری تست شود و رم خالی میشود تا برای اجرای دوباره نیاز به kernel restart و اجرای مجدد تمام کد نباشد.

بخش پنجم : پیاده سازی order SI

روش soft_order_si نسخه ای کمتر دقیق از شاخص جداسازی را محاسبه می کند. بر خلاف high_order_si، که برای کمک به امتیاز، تمام همسایگان نزدیک‌ترین مرتبه را ملزم می‌کند که برچسب یکسانی برای یک امتیاز داشته باشند، soft_order_si رویکرد نرم‌تری دارد. تعداد برچسب‌های منطبق را در میان ترتیب نزدیک‌ترین همسایگان برای هر نقطه داده می‌شمارد و این تعداد را بر ترتیب تقسیم می‌کند تا برای هر امتیاز امتیازی به دست آید. سپس این نمرات فردی با تعداد کل نقاط داده خلاصه و نرمال می شوند. این روش نمای دقیق تری از جداسازی را ارائه می دهد، که مطابقت های جزئی را تأیید می کند و ارزیابی درجه بندی شده تری از ساختار مجموعه داده ارائه می دهد.

```
def soft_order_si(self, order):
      """
```

```
        Calculate the soft order separation index (Soft-SI) for the dataset.

        This index provides a less strict measure of separation, considering
matching labels among 'order' nearest neighbors.

        Args:
            order (int): The order of separation to consider.

        Returns:
            float: The calculated soft order separation index.
        """
        try:
            sorted_distances, neighbor_indices = torch.sort(self.dis_matrix,
dim=1)
            if self.label.dim() == 1:
                labels_reshaped = self.label.unsqueeze(1)
            else:
                labels_reshaped = self.label
            expanded_labels = labels_reshaped.expand(self.n_data, order)
            neighbor_labels = labels_reshaped[neighbor_indices[:,
:order]].view(self.n_data, order)
            total_soft_si = 0
            for i in tqdm(range(self.n_data), desc="Calculating Soft Order SI"):
                matching_labels_count = (expanded_labels[i] ==
neighbor_labels[i]).sum()
                total_soft_si += matching_labels_count.float() / order
            final_soft_si = total_soft_si / self.n_data
            return final_soft_si.item()
        except RuntimeError as e:
            if "out of memory" in str(e):
                print("CUDA out of memory. Try reducing 'order' or using a device
with more memory.")
            else:
                raise e
```

بخش ششم : Anti SI

روش anti_si یک شاخص جداسازی «Anti» را محاسبه می‌کند و بر عدم تشابه به جای شباهت تمرکز می‌کند. نسبت نقاط داده‌ای را ارزیابی می‌کند که برچسب‌های آنها با ترتیب نزدیک‌ترین همسایگانشان متفاوت است. پیاده سازی تا حدودی شبیه high_order_si است، اما مقایسه برچسب را معکوس می کند (به دنبال عدم تطابق به جای مطابقت). این روش بینشی را در مورد مجموعه داده از منظری متفاوت ارائه

می‌کند، با تمرکز بر اینکه چگونه کلاس‌های مختلف در فضای ویژگی به خوبی از هم جدا شده‌اند. به ویژه در مواردی که همپوشانی کلاس ها نگران کننده است، می تواند آموزنده باشد.

```
def anti si(self, order):
  """
  Calculate the anti-separation index (Anti-SI) for the dataset.

  This index measures the proportion of data points having different labels from their
'order' nearest neighbors.

  Args:
      order (int): The order of separation to consider.

  Returns:
      float: The calculated anti-separation index.
  """
  try:
      sorted_dist, sorted_indices = torch.sort(self.dis_matrix, dim=1)
      if self.label.dim() == 1:
          labels = self.label.unsqueeze(1)
      else:
          labels = self.label
      expanded_labels = labels.expand(self.n_data, order)
      nearest neighbor labels = labels[sorted indices[:, :order]].view(self.n data, order)
      total_anti_si = 0
      for i in tqdm(range(self.n_data), desc="Calculating Anti-SI"):
          label_difference = 1 - (expanded_labels[i] == nearest_neighbor_labels[i]).float()
          total anti si += torch.prod(label difference)
      final_anti_si = total_anti_si / self.n_data
      return final_anti_si.item()
  except RuntimeError as e:
      if "out of memory" in str(e):
          print("CUDA out of memory. Try reducing 'order' or using a device with more
memory.")
      else:
          raise e
```

بخش هفتم : Center SI

روش center_si (شاخص جداسازی مبتنی بر مرکز) روشی کارآمد برای اندازه‌گیری جداسازی کلاس در مجموعه‌های داده است که در آن هر کلاس یک خوشه مجزا را تشکیل می‌دهد. این روش ابتدا میانگین (مرکز) نقاط داده برای هر کلاس را محاسبه می کند. سپس فاصله هر نقطه داده تا این مراکز کلاس را محاسبه کرده و نزدیکترین مرکز کلاس را برای هر نقطه تعیین می کند. امتیاز CSI نسبت نقاط داده ای

است که نزدیکترین مرکز کلاس با برچسب کلاس واقعی آنها مطابقت دارد. این روش از نظر محاسباتی کارآمد است، به ویژه برای مجموعه داده هایی که کلاس ها به خوبی از هم جدا شده اند و به طور معمول توزیع می شوند. این یک راه سریع و موثر برای ارزیابی تفکیک پذیری کلاس های مختلف در مجموعه داده ارائه می دهد.

```
def center_si(self):
  """
  Calculates the center-based Separation Index (CSI) for the dataset.

  CSI measures the proportion of data points closest to the mean of their respective
classes.
  It's a faster computation method, especially suitable for datasets where each class forms
a unique and normal distribution.

  Returns:
      float: The calculated Center-based Separation Index (CSI).
  """
  try:
      class_centers = torch.stack([
          self.data[self.label.squeeze() == cls].mean(dim=0)
          for cls in tqdm(range(self.n_class), desc="Calculating Class Centers")
      ])
      distances_to_centers = torch.cdist(self.data, class_centers, p=2)
      nearest_center_labels = torch.argmin(distances_to_centers, dim=1)
      csi = torch.sum(nearest_center_labels == self.label.squeeze()).float() / self.n_data
      return csi.item()
  except RuntimeError as e:
      if "out of memory" in str(e):
          print("CUDA out of memory. Try reducing the dataset size or using a device with
more GPU memory.")
          return None
      else:
          raise e
```

بخش هشتم : هندل کردن داده های زیاد

در این بخش تمرکز بر روی هندل کردن یکی از مشکلات کد SI که تعداد داده های زیاد و گرفتن cuda out of memory میباشد.

کلاس ARH_SeparationIndex شامل چندین توابع تجدید نظر شده (si_batch، soft_order_si_batch و center_si_batch) است که برای مدیریت کارآمد مجموعه داده های بزرگ در حین مدیریت منابع حافظه CUDA طراحی شده اند. این توابع تجدید نظر شده از پردازش دسته‌ای (batch) برای محاسبه شاخص‌های جداسازی مختلف استفاده می‌کنند و اطمینان حاصل می‌کنند که محاسبات را می‌توان حتی روی مجموعه‌های داده‌ای که خیلی بزرگ هستند و به طور کامل در حافظه GPU قرار نمی‌گیرند، انجام داد.

مراحل پردازش به شرح زیر میباشد:

پردازش دسته ای

- **محاسبات دسته ای**: هر تابع اصلاح شده داده ها را به صورت دسته ای پردازش می کند. این رویکرد مجموعه داده را به قطعات کوچکتر و قابل مدیریت تقسیم می کند که به صورت جداگانه پردازش می شوند. با انجام این کار، نیاز به حافظه برای هر عملیات به میزان قابل توجهی کاهش می یابد و امکان پردازش مجموعه داده های بزرگ را فراهم می کند که در غیر این صورت منجر به خطاهای خارج از حافظه در GPU می شود.
- **دسته بندی پویا:** اندازه دسته پارامتری است که می تواند بر اساس حافظه GPU موجود تنظیم شود. این انعطاف پذیری را فراهم می کند، زیرا کاربران با پیکربندی های سخت افزاری مختلف می توانند اندازه دسته را متناسب با محدودیت های حافظه GPU خاص خود بهینه کنند.
- **پردازش مداوم در سرتاسر دسته ها:** علیرغم پردازش دسته ای، هر تابع ثبات را در محاسبات تضمین می کند. این با در نظر گرفتن تمام نقاط داده لازم در کل مجموعه داده برای هر دسته به دست می آید. به عنوان مثال، در محاسبه نزدیکترین همسایگان یا مراکز کلاس، توابع تضمین می کنند که این محاسبات نسبت به کل مجموعه داده انجام می شود، نه فقط در دسته های جداگانه.

مدیریت خطاهای CUDA خارج از حافظه

- **Try-Except Block**: هر تابع در یک بلوک try-except پیچیده شده است تا به خوبی خطاهای CUDA خارج از حافظه را مدیریت کند. اگر چنین خطایی در حین محاسبه رخ دهد،

این تابع پیام مفیدی را چاپ می کند که توصیه می کند اندازه دسته را کاهش دهید یا از دستگاهی با حافظه GPU بیشتر استفاده کنید.

- **مدیریت حافظه**: پس از پردازش هر دسته، توابع torch.cuda.empty_cache() را فراخوانی می کنند تا حافظه GPU استفاده نشده آزاد شود. این مرحله در مدیریت منابع GPU، به ویژه هنگام کار با مجموعه داده های بزرگ، بسیار مهم است. این حافظه کش که دیگر مورد نیاز نیست را پاک می کند و امکان استفاده کارآمدتر از حافظه GPU را فراهم می کند.

با توجه به توضیحات فوق توابع کد batch اضافه شده به آن ها به شرح زیر میباشد.

```
def si_batch(self, batch_size):
    """
    Calculate the separation index (SI) for the dataset in a batched manner.
    This measures the proportion of data points having the same label as their nearest neighbor.
    Args:
        batch_size (int): The size of each batch to process.
    Returns:
        float: The calculated Separation Index (SI).
    """
    total_matches = 0.0
    try:
        for batch_start in tqdm(range(0, self.n_data, batch_size), desc="Calculating SI"):
            batch_end = min(batch_start + batch_size, self.n_data)
            _, nearest_neighbors_indices = torch.min(self.dis_matrix[batch_start:batch_end], dim=1)
            batch_labels = self.label[batch_start:batch_end]
            total_matches += (batch_labels == self.label[nearest_neighbors_indices]).float().sum()

        si = total_matches / self.n_data
        return si.item()
    except RuntimeError as e:
        if "out of memory" in str(e):
            print("CUDA out of memory. Try reducing 'batch_size'.")
            return None
        else:
            raise e
    finally:
        torch.cuda.empty_cache()
```

```
def high_order_si_batch(self, order, batch_size):
  """
  Calculate the high order separation index for the dataset in a batched manner.

  This index is a stricter version of SI, considering the first 'order' nearest neighbors.
  It handles large datasets by processing in batches to avoid CUDA memory issues.

  Args:
      order (int): The order of separation to consider.
      batch_size (int): The size of each batch to process.

  Returns:
      float: The calculated high order separation index.
  """
  try:
      total_high_order_si = 0.0  # Initialize the high order SI accumulator

      for batch_start in tqdm(range(0, self.n_data, batch_size), desc="Computing High Order SI"):
          batch_end = min(batch_start + batch_size, self.n_data)  # Determine the batch end
          batch_distances = torch.cdist(self.data[batch_start:batch_end], self.data, p=2)
          _, sorted_indices = torch.sort(batch_distances, dim=1)
          batch_sorted_neighbor_labels = self.label[sorted_indices[:, 1:order+1]]
          batch_labels = self.label[batch_start:batch_end].unsqueeze(1)
          match_labels = batch_labels == batch_sorted_neighbor_labels
          match_score = match_labels.float().prod(dim=1)
          total_high_order_si += match_score.sum()

      # Normalize the high order SI by the number of data points
      final_high_si = total_high_order_si / self.n_data
      return final_high_si.item()

  except RuntimeError as e:
      if "out of memory" in str(e):
          print("CUDA out of memory. Try reducing 'batch_size' or using a device with more GPU memory.")
          return None
      else:
          raise e
  finally:
    # Release GPU memory cache to free unused memory
    torch.cuda.empty_cache()

def soft_order_si_batch(self, order, batch_size):
    """
    Calculate the soft order separation index (Soft-SI) for the dataset in a batched manner.
    This provides a less strict measure of separation, considering matching labels among 'order' nearest
neighbors.
    Args:
        order (int): The order of separation to consider.
```

```
        batch_size (int): The size of each batch to process.
    Returns:
        float: The calculated soft order separation index.
    """
    total_soft_scores = 0.0
    try:
        for batch_start in tqdm(range(0, self.n_data, batch_size), desc="Calculating Soft Order SI"):
            batch_end = min(batch_start + batch_size, self.n_data)
            batch_distances = torch.cdist(self.data[batch_start:batch_end], self.data, p=2)
            _, sorted_indices = torch.sort(batch_distances, dim=1)
            batch_labels = self.label[batch_start:batch_end].unsqueeze(1)
            nearest_neighbor_labels = self.label[sorted_indices[:, 1:order+1]]
            soft_scores = (batch_labels == nearest_neighbor_labels).float().sum(dim=1) / order
            total_soft_scores += soft_scores.sum()

        soft_si = total_soft_scores / self.n_data
        return soft_si.item()
    except RuntimeError as e:
        if "out of memory" in str(e):
            print("CUDA out of memory. Try reducing 'batch_size'.")
            return None
        else:
            raise e
    finally:
        torch.cuda.empty_cache()


def center_si_batch(self, batch_size):
  """
  Calculates the center-based Separation Index (CSI) for the dataset in a batched manner.
  CSI measures the proportion of data points closest to the mean of their respective classes.
  Args:
      batch_size (int): The size of each batch to process.
  Returns:
      float: The calculated Center-based Separation Index (CSI).
  """
  try:
      # Squeeze the label tensor to make it one-dimensional for indexing
      squeezed_labels = self.label.squeeze()

      # Calculate the class centers
      class_centers = torch.stack([self.data[squeezed_labels == cls].mean(dim=0) for cls in
range(self.n_class)])
      total_center_matches = 0.0

      # Batch processing for CSI calculation
      for batch_start in tqdm(range(0, self.n_data, batch_size), desc="Calculating CSI"):
          batch_end = min(batch_start + batch_size, self.n_data)
```

```
            batch_distances = torch.cdist(self.data[batch_start:batch_end], class_centers, p=2)
            nearest_center_labels = torch.argmin(batch_distances, dim=1)
            total_center_matches += (squeezed_labels[batch_start:batch_end] ==
nearest_center_labels).float().sum()

        # Normalize the CSI value
        csi = total_center_matches / self.n_data
        return csi.item()
    except RuntimeError as e:
        if "out of memory" in str(e):
            print("CUDA out of memory. Try reducing 'batch_size'.")
            return None
        else:
            raise e
    finally:
        torch.cuda.empty_cache()
```

این توابع تجدید نظر شده برای استفاده در سناریوهایی در نظر گرفته شده است که در آن مجموعه داده بسیار بزرگ است و نمی توان آن را به یکباره در GPU موجود پردازش کرد. آنها یک راه حل مقیاس پذیر برای محاسبه شاخص های جداسازی در مجموعه داده های بزرگ ارائه می دهند و در عین حال منابع GPU را به طور موثر مدیریت می کنند. بدین ترتیب دیگر به کمک این توابع میتوان پردازش را بر روی تمام داده های مجموعه داده های مختلف محاسبه کرد و مشکلات سخت افزاری ای برای محاسبه پیش نمیاید.

کد کلاس فوق در نسخه fork شده از گیت اصلی به ادرس زیر قابل دسترسی میباشد. لازم به ذکر است جهت ارزیابی خروجی های این کلاس با کلاس اصلی مقایسه شده و دقیقا به همان اعداد میرسد.

https://github.com/Arhosseini77/data_complexity_measures/blob/main/models/ARH_SeparationIndex.py

---

در تمرین ۱ نتوانسته بودیم روی کل داده ها متریک ها را بدست بیاوریم، با توجه به اینکه کد اصلاح شد، جدول پایین عدد متریک های جدید است:

جدول ۱ – عدد متریک های جدید

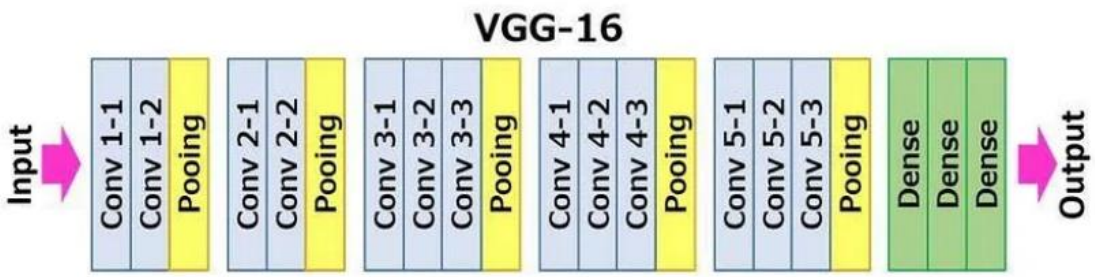
تمرین۱	SI	High order SI	High order Soft Si	Anti SI	Center SI
Cifar100 داده خام	0.17406	0.073199	0.1501	0.8363999	0.1065199971
در latent فضای Efficientnetv2-s	0.69742	0.5702399611	0.6812399626	0.254399985	0.7508999705

سوال اول : رتبه بندی معماری های مختلف شبکه عصبی

الف) آموزش شبکه

- VGG16

پیاده سازی مدل VGG16:



شکل ۱ – مدل vgg16

با توجه به معماری شبکه که در بالا نمایش داده شده است، کد آن را بصورت زیر پیاده سازی میکنیم:

```
import torch
import torch.nn as nn
cfg = {
    'A' : [64,     'M', 128,      'M', 256, 256,           'M', 512,
512,           'M', 512, 512,           'M'],
    'B' : [64, 64, 'M', 128, 128, 'M', 256, 256,           'M', 512,
512,           'M', 512, 512,           'M'],
    'D' : [64, 64, 'M', 128, 128, 'M', 256, 256, 256,      'M', 512, 512,
512,      'M', 512, 512, 512,      'M'],
    'E' : [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512,
512, 'M', 512, 512, 512, 512, 'M']
}

class VGG(nn.Module):
    def __init__(self, features, num_class=100):
        super().__init__()
        self.features = features
        self.classifier = nn.Sequential(
            nn.Linear(512, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_class)
```

```
        )
    def forward(self, x):
        output = self.features(x)
        output = output.view(output.size()[0], -1)
        output = self.classifier(output)
        return output

def make_layers(cfg, batch_norm=False):
    layers = []
    input_channel = 3
    for l in cfg:
        if l == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            continue
        layers += [nn.Conv2d(input_channel, l, kernel_size=3, padding=1)]
        if batch_norm:
            layers += [nn.BatchNorm2d(l)]
        layers += [nn.ReLU(inplace=True)]
        input_channel = l
    return nn.Sequential(*layers)

def vgg16_bn():
    return VGG(make_layers(cfg['D'], batch_norm=True))
```

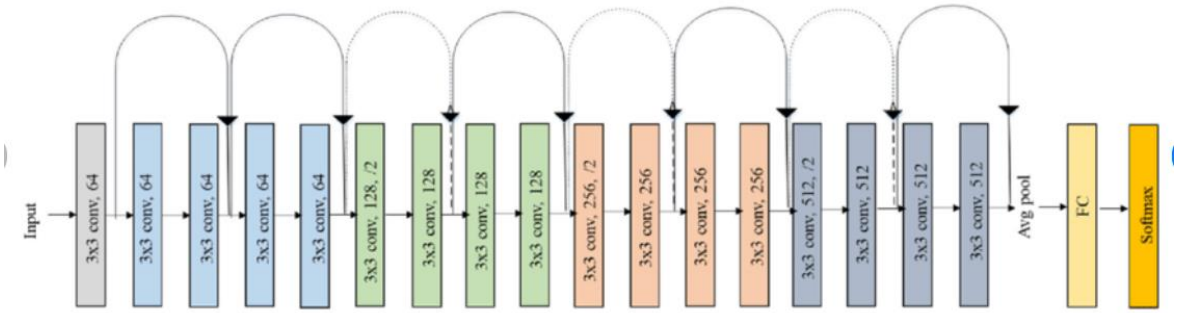
در ابتدا cfg را تعریف میکنیم، این تنظیمات VGG را برای اعماق مختلف مشخص می کند. هر پیکربندی لیستی از اعداد و M است، که در آن اعداد نشان دهنده تعداد فیلترها در لایه های کانولوشن هستند و M نشان دهنده حداکثر ادغام است.

این تابع یک پیکربندی cfg می‌گیرد و دنباله ای از لایه ها (پیچیدگی ها، نرمال سازی دسته ای و فعال سازی های ReLU) را ایجاد می کند. از پیکربندی مشخص شده برای ساخت بخش کانولوشنال شبکه VGG استفاده می کند.

class VGG(nn.Module) این کلاس مدل کلی VGG را تعریف می کند. بخش کانولوشن (ویژگی ها) را می گیرد و یک طبقه بندی کاملا متصل در بالا اضافه می کند. طبقه‌بندی‌کننده شامل سه لایه کاملاً متصل با فعال‌سازی ReLU و dropout در بین آن‌ها است.

() vgg16_bn این تابع یک مدل VGG-16 با نرمال سازی دسته ای ایجاد می کند. از تابع make_layers با پیکربندی D استفاده می کند و batch_norm را روی True تنظیم می کند.

ResNet18 -



شکل ۲ -ResNet18

پیاده سازی مدل:

```
import torch
import torch.nn as nn

class BasicBlock(nn.Module):

    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()

        #residual function
        self.residual_function = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels * BasicBlock.expansion,
kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels * BasicBlock.expansion)
        )

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != BasicBlock.expansion * out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels * BasicBlock.expansion,
kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * BasicBlock.expansion)
            )

    def forward(self, x):
        return nn.ReLU(inplace=True)(self.residual_function(x) +
self.shortcut(x))
```

```
class BottleNeck(nn.Module):
    """Residual block for resnet over 50 layers

    """
    expansion = 4
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.residual_function = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, stride=stride, kernel_size=3,
padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels * BottleNeck.expansion,
kernel_size=1, bias=False),
            nn.BatchNorm2d(out_channels * BottleNeck.expansion),
        )
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels * BottleNeck.expansion:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels * BottleNeck.expansion,
stride=stride, kernel_size=1, bias=False),
                nn.BatchNorm2d(out_channels * BottleNeck.expansion)
            )
    def forward(self, x):
        return nn.ReLU(inplace=True)(self.residual_function(x) +
self.shortcut(x))

class ResNet(nn.Module):
    def __init__(self, block, num_block, num_classes=100):
        super().__init__()
        self.in_channels = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True))
        #we use a different inputsize than the original paper
        #so conv2_x's stride is 1
        self.conv2_x = self._make_layer(block, 64, num_block[0], 1)
        self.conv3_x = self._make_layer(block, 128, num_block[1], 2)
        self.conv4_x = self._make_layer(block, 256, num_block[2], 2)
        self.conv5_x = self._make_layer(block, 512, num_block[3], 2)
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)
```

```
    def _make_layer(self, block, out_channels, num_blocks, stride):
        """make resnet layers(by layer i didnt mean this 'layer' was the
        same as a neuron netowork layer, ex. conv layer), one layer may
        contain more than one residual block

        Args:
            block: block type, basic block or bottle neck block
            out_channels: output depth channel number of this layer
            num_blocks: how many blocks per layer
            stride: the stride of the first block of this layer

        Return:
            return a resnet layer
        """

        # we have num_block blocks per layer, the first block
        # could be 1 or 2, other blocks would always be 1
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        output = self.conv1(x)
        output = self.conv2_x(output)
        output = self.conv3_x(output)
        output = self.conv4_x(output)
        output = self.conv5_x(output)
        output = self.avg_pool(output)
        output = output.view(output.size(0), -1)
        output = self.fc(output)
        return output

def resnet18():
    """ return a ResNet 18 object
    """
    return ResNet(BasicBlock, [2, 2, 2, 2])
```

این کد یک معماری ResNet را با استفاده از PyTorch تعریف می کند. ResNet یک معماری است که اتصالات یا skip connection را برای کمک به آموزش شبکه های عصبی بسیار عمیق معرفی می کند.

کلاس BasicBlock:

این کلاس بلوک residual اصلی را برای معماری ResNet18 تعریف می کند. این شامل دو لایه ۳ در ۳ کانولوشن با batch normalization و عملکردهای فعال سازی ReLU است. ویژگی کلاس expansion ر وی ۱ تنظیم شده است که نشان می دهد بلوک تعداد کانال ها را تغییر نمی دهد.

کلاس Bottleneck:

این کلاس بلوک گلوگاه مورد استفاده در معماری های ResNet را با بیش از ۵۰ لایه تعریف می کند. از سه لایه کانولوشن با کرنل های ۱ در ۱، ۳ در ۳ و ۱ در ۱ استفاده می کند. ویژگی کلاس توسعه روی ۴ تنظیم شده است که نشان می دهد بلوک تعداد کانال ها را ضریب ۴ افزایش می دهد.

کلاس ResNet:

این کلاس معماری کلی ResNet را تعریف می کند. یک نوع بلوک یا BasicBlock یا BottleNeck ، تعداد بلوک‌ها برای هر لایه num_block و تعداد کلاس‌های خروجی num_classes را به عنوان پارامتر در نظر می‌گیرد.

این معماری از یک لایه کانولوشنال اولیه conv1 و به دنبال آن چهار لایه conv2_x تا conv5_x تشکیل شده است که هر کدام شامل چندین بلوک از نوع مشخص شده است. تعداد کانال ها با استفاده از روش _make_layer تنظیم می شود که لایه را با تعداد بلوک و گام مشخص ایجاد می کند. لایه نهایی شامل یک لایه adaptive average pooling است که به دنبال آن یک لایه کاملاً متصل fc برای طبقه بندی می باشد.

روش _make_layer:

این روش یک لایه با تعداد مشخص شده بلوک، کانال های خروجی و گام ایجاد می کند. از نوع بلوک مشخص شده استفاده می کند و تعداد کانال های ورودی را متناسب با آن تنظیم می کند. بلوک اول در لایه ممکن است گام متفاوتی داشته باشد و بلوک های بعدی گامی ۱ داشته باشند.

روش forward:

عبور رو به جلو شبکه را تعریف می کند. ورودی را از لایه های کانولوشنال و بلوک های باقیمانده conv1 تا conv5_x) عبور می دهد. ادغام میانگین تطبیقی را اعمال می کند، خروجی را صاف می کند و برای طبقه بندی از لایه کاملا متصل عبور می دهد.

عملکرد resnet18:

یک نمونه مدل ResNet-18 را با استفاده از کلاس BasicBlock و پیکربندی [۲، ۲، ۲، ۲] برای تعداد بلوک‌ها در هر لایه برمی‌گرداند.

- Train مدل

ابتدا repository زیر را clone میکنیم:

```
! git clone https://github.com/K-Hooshanfar/pytorch-cifar100
```

با استفاده از دستورات زیر دو مدل بالا را ترین میکنیم:

```
!python train.py -net resnet18 -gpu
```

```
!python train.py -net vgg16 -gpu
```

کد train آن بصورت زیر است که در ادامه آن را توضیح میدهیم:

```
import os
import sys
import argparse
import time
from datetime import datetime
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt
from conf import settings
from utils import get_network, get_training_dataloader, get_test_dataloader,
WarmUpLR, \
    most_recent_folder, most_recent_weights, last_epoch, best_acc_weights

# Function to train the model for one epoch
def train(epoch):
    # Set the model to training mode
    net.train()
    correct_train = 0.0
    total_train = 0
```

```
    for batch_index, (images, labels) in enumerate(cifar100_training_loader):
        # Move data to GPU if available
        if args.gpu:
            labels = labels.cuda()
            images = images.cuda()

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = net(images)

        # Compute the loss
        loss = loss_function(outputs, labels)

        # Backward pass
        loss.backward()
        optimizer.step()

        # Compute and track accuracy
        _, predicted_train = outputs.max(1)
        correct_train += predicted_train.eq(labels).sum().item()
        total_train += labels.size(0)

        n_iter = (epoch - 1) * len(cifar100_training_loader) + batch_index + 1

        # Warm-up learning rate if applicable
        last_layer = list(net.children())[-1]
        if epoch <= args.warm:
            warmup_scheduler.step()

        # Compute training accuracy
        train_accuracy = correct_train / len(cifar100_training_loader.dataset)

    # Log parameters for each layer
    for name, param in net.named_parameters():
        layer, attr = os.path.splitext(name)
        attr = attr[1:]

    return loss.item() / len(cifar100_training_loader), train_accuracy

# Function to evaluate the model on the validation set
@torch.no_grad()
def eval_training(epoch=0, tb=True):
    # Set the model to evaluation mode
    net.eval()
    val_loss = 0.0
    correct = 0.0
```

```
    for (images, labels) in cifar100_val_loader:
        # Move data to GPU if available
        if args.gpu:
            images = images.cuda()
            labels = labels.cuda()

        # Forward pass
        outputs = net(images)

        # Compute the loss
        loss = loss_function(outputs, labels)
        val_loss += loss.item()

        # Compute and track accuracy
        _, preds = outputs.max(1)
        correct += preds.eq(labels).sum()

        val_losses = (val_loss / len(cifar100_val_loader.dataset))
        val_acc = correct.float() / len(cifar100_val_loader.dataset)

    # Print evaluation results
    finish = time.time()
    print('Evaluating Network.....')
    print('Val set: Epoch: {}, Average loss: {:.4f}, Accuracy: {:.4f}, Time
consumed:{:.2f}s'.format(
        epoch,
        val_loss / len(cifar100_val_loader.dataset),
        correct.float() / len(cifar100_val_loader.dataset),
        finish - start
    ))
    print()

    return val_losses, val_acc, correct.float() /
len(cifar100_val_loader.dataset)

if __name__ == '__main__':
    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument('-net', type=str, required=True, help='net type')
    parser.add_argument('-gpu', action='store_true', default=False, help='use gpu
or not')
    parser.add_argument('-b', type=int, default=128, help='batch size for
dataloader')
    parser.add_argument('-warm', type=int, default=1, help='warm up training
phase')
    parser.add_argument('-lr', type=float, default=0.1, help='initial learning
rate')
```

```
    parser.add_argument('-resume', action='store_true', default=False,
help='resume training')
    args = parser.parse_args()

    # Get the neural network model
    net = get_network(args)

    # Lists to store training and validation losses, accuracies
    train_loss_list = []
    val_loss_list = []
    train_accuracy_list = []
    val_accuracy_list = []

    # Data preprocessing
    cifar100_training_loader, cifar100_val_loader = get_training_dataloader(
        settings.CIFAR100_TRAIN_MEAN,
        settings.CIFAR100_TRAIN_STD,
        num_workers=4,
        batch_size=args.b,
        shuffle=True
    )

    # Loss function, optimizer, and learning rate scheduler setup
    loss_function = nn.CrossEntropyLoss()
    optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=0.9,
weight_decay=5e-4)
    train_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
milestones=settings.MILESTONES, gamma=0.2)
    iter_per_epoch = len(cifar100_training_loader)
    warmup_scheduler = WarmUpLR(optimizer, iter_per_epoch * args.warm)

    # Checkpointing and logging setup
    if args.resume:
        recent_folder = most_recent_folder(os.path.join(settings.CHECKPOINT_PATH,
args.net), fmt=settings.DATE_FORMAT)
        if not recent_folder:
            raise Exception('no recent folder were found')

        checkpoint_path = os.path.join(settings.CHECKPOINT_PATH, args.net,
recent_folder)

    else:
        checkpoint_path = os.path.join(settings.CHECKPOINT_PATH, args.net,
settings.TIME_NOW)

    if not os.path.exists(settings.LOG_DIR):
        os.mkdir(settings.LOG_DIR)
```

```
    input_tensor = torch.Tensor(1, 3, 32, 32)
    if args.gpu:
        input_tensor = input_tensor.cuda()

    # Create checkpoint folder to save model
    if not os.path.exists(checkpoint_path):
        os.makedirs(checkpoint_path)
    checkpoint_path = os.path.join(checkpoint_path, '{net}-{epoch}-{type}.pth')

    best_acc = 0.0
    if args.resume:
        # Load the best weights for evaluation
        best_weights = best_acc_weights(os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder))
        if best_weights:
            weights_path = os.path.join(settings.CHECKPOINT_PATH, args.net,
recent_folder, best_weights)
            print('found best acc weights file:{}'.format(weights_path))
            print('load best training file to test acc...')
            net.load_state_dict(torch.load(weights_path))
            best_acc = eval_training(tb=False)
            print('best acc is {:0.2f}'.format(best_acc))

        # Load the most recent weights for resuming training
        recent_weights_file =
most_recent_weights(os.path.join(settings.CHECKPOINT_PATH, args.net,
recent_folder))
        if not recent_weights_file:
            raise Exception('no recent weights file were found')
        weights_path = os.path.join(settings.CHECKPOINT_PATH, args.net,
recent_folder, recent_weights_file)
        print('loading weights file {} to resume
training.....'.format(weights_path))
        net.load_state_dict(torch.load(weights_path))

        # Get the last epoch for resuming training
        resume_epoch = last_epoch(os.path.join(settings.CHECKPOINT_PATH,
args.net, recent_folder))

    # Main training loop
    for epoch in range(1, settings.EPOCH + 1):
        if epoch > args.warm:
            train_scheduler.step(epoch)

        if args.resume:
            if epoch <= resume_epoch:
                continue
```

```
        train_losses, train_accs = train(epoch)
        val_losses, val_accs, acc = eval_training(epoch)

        # Append training and validation metrics to lists
        train_loss_list.append(train_losses)
        train_accuracy_list.append(train_accs)
        val_loss_list.append(val_losses)
        val_accuracy_list.append(val_accs)

        # Save the model with the best validation accuracy
        if epoch > settings.MILESTONES[1] and best_acc < acc:
            weights_path = checkpoint_path.format(net=args.net, epoch=epoch,
type='best')
            print('saving weights file to {}'.format(weights_path))
            torch.save(net.state_dict(), weights_path)
            best_acc = acc
            continue

        # Save the model at regular intervals
        if not epoch % settings.SAVE_EPOCH:
            weights_path = checkpoint_path.format(net=args.net, epoch=epoch,
type='regular')
            print('saving weights file to {}'.format(weights_path))
            torch.save(net.state_dict(), weights_path)

    # Plot training and validation metrics
    plt.figure(figsize=(10, 5))

    # Plot loss
    plt.subplot(1, 2, 1)
    plt.plot([float(x) for x in train_loss_list], label='Train')
    plt.plot([float(x) for x in val_loss_list], label='Val')
    plt.title('Train-Val Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    # Plot accuracy
    plt.subplot(1, 2, 2)
    plt.plot([float(x) for x in train_accuracy_list], label='Train')
    plt.plot([float(x) for x in val_accuracy_list], label='Val')
    plt.title('Train-Val Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.savefig('training_plots.png')
```

```
    plt.show()
```

این کد برای آموزش یک معماری شبکه عصبی مشخص بر روی مجموعه داده CIFAR-100 طراحی شده است. در ادامه به ترتیب کد را توضیح دادیم، همچنین توضیحات تکمیلی بصورت کامنت در کد قرار دارند.

اسکریپت با وارد کردن کتابخانه‌های لازم، از جمله PyTorch، NumPy، و برخی از ماژول‌های تعریف شده در سایر فایل‌های conf.py و utils.py شروع می‌شود. تابع get_network برای به دست آوردن معماری شبکه عصبی مشخص شده استفاده می شود.

توابع آموزش:

Train:

این تابع شبکه عصبی را برای یک دوره آموزش می دهد. این شامل پاس های forward and backward است، loss را محاسبه می کند، پارامترهای مدل را به روز می کند، و دقت آموزش را دنبال می کند. همچنین از نرخ یادگیری warm-upدر طول دوره های آموزشی اولیه پشتیبانی می کند.

eval_training(epoch=0, tb=True):

این تابع مدل آموزش دیده را در مجموعه اعتبارسنجی ارزیابی می کند. این loss و دقت اعتبارسنجی را محاسبه می کند، نتایج ارزیابی را چاپ می کند و معیارهای مربوطه را برمی گرداند.

اسکریپت اصلی:

سپس اسکریپت با استفاده از اگر __ :'__main__' == __name وارد بلوک اصلی می شود.

آرگومان های Command Line:

این اسکریپت از ماژول argparse برای تجزیه آرگومان های Command Line مانند نوع شبکه عصبی، استفاده از GPU، اندازه batch، مرحله warm-up، سرعت یادگیری اولیه استفاده می کند.

راه اندازی شبکه عصبی:

سپس اسکریپت مدل شبکه عصبی را با استفاده از تابع get_network بر اساس نوع شبکه ارائه شده دریافت می کند.

پیش پردازش داده ها:

بارگذارهای داده برای مجموعه داده های آموزشی و اعتبار سنجی با استفاده از تابع get_training_dataloader ایجاد می شوند که شامل مشخص کردن تبدیل داده ها، میانگین و انحراف استاندارد است.

در سوال از ما خواسته شده که مجموعه داده را به train-test-val تقسیم کنیم که در این بخش این کار را انجام دادیم. داده train را برای ترین کردن مدل و از val برای ارزیابی استفاده کردیم. همچنین در انتهای train از داده تست فقط برای تست نهایی مدل استفاده کردیم.

عملکرد از دست دادن، بهینه ساز و scheduler:

این اسکریپت تابع ضرر (آنتروپی متقابل)، بهینه‌ساز (SGD با کاهش حرکت و وزن)، و زمان‌بندی نرخ یادگیری را تنظیم می‌کند.

```
   loss_function = nn.CrossEntropyLoss()
        optimizer   =   optim.SGD(net.parameters(),   lr=args.lr,   momentum=0.9,
weight_decay=5e-4)
```

```
    train_scheduler = optim.lr_scheduler.MultiStepLR(optimizer,
milestones=settings.MILESTONES, gamma=0.2)
    iter_per_epoch = len(cifar100_training_loader)
    warmup_scheduler = WarmUpLR(optimizer, iter_per_epoch * args.warm)
```

در قطعه کد ارائه شده، دو زمان‌بندی نرخ یادگیری برای مدیریت پویایی آموزشی شبکه عصبی نمونه‌سازی شده‌اند. اولین زمانبندی، train_scheduler، با استفاده از کلاس MultiStepLR PyTorch پیکربندی شده است. این زمان‌بند نرخ یادگیری بهینه‌ساز مشخص‌شده را در نقاط عطف از پیش تعیین‌شده در طول آموزش، همانطور که با لیست (MILESTONES = [60, 120, 160]) settings.MILESTONES مشخص شده است، تنظیم می‌کند. در هر نقطه عطف، نرخ یادگیری در ضریب ۰,۲ (گاما) ضرب می شود. این رویکرد معمولاً برای تنظیم دقیق نرخ یادگیری با پیشرفت فرآیند بهینه‌سازی استفاده می‌شود. دومین زمانبندی، warmup_scheduler، از یک پیاده سازی سفارشی به نام WarmUpLR استفاده می کند. این زمان‌بندی برای افزایش تدریجی میزان یادگیری به صورت خطی در طول مرحله گرم کردن طراحی شده است، که شامل تعدادی از epochهای تعیین شده توسط مقدار args.warm می‌شود. مرحله گرم کردن برای تسهیل همگرایی روان تر شبکه عصبی در طول epochهای آموزشی اولیه در نظر گرفته شده است. این استراتژی‌های زمان‌بندی نرخ یادگیری به پایداری و کارایی فرآیند آموزشی کمک می‌کنند و به طور بالقوه عملکرد مدل را در مجموعه داده CIFAR-100 افزایش می‌دهند.

چک پوینت و logging:

این اسکریپت چک پوینت را برای ذخیره وضعیت مدل در حین آموزش مدیریت می کند و پیشرفت آموزش را ثبت می کند.

حلقه آموزشی:

حلقه اصلی آموزشی در طول **epoch** ها تکرار می شود و توابع آموزش و ارزیابی را فراخوانی می کند. سرعت یادگیری را پس از مرحله **warm-up** تنظیم می کند و وزن مدل را بر اساس عملکرد اعتبارسنجی ذخیره می کند.

Plotting:

در نهایت، اسکریپت آموزش و از دست دادن اعتبار، و همچنین دقت آموزش و اعتبار سنجی را در طول دوره ها ترسیم می کند.

برای ترین کردن کد بالا به تابع های کمکی زیر نیاز داریم که در ادامه آن را توضیح میدهیم.

```
def get_network(args):
    """Return the specified neural network architecture."""
    if args.net == 'vgg16':
        from models.vgg import vgg16_bn
        net = vgg16_bn()
    elif args.net == 'resnet18':
        from models.resnet import resnet18
        net = resnet18()
    else:
        print('The specified network is not supported yet.')
        sys.exit()
    if args.gpu:
        net = net.cuda()  # Move the model to GPU if specified
    return net

def get_training_dataloader(mean, std, batch_size=16, num_workers=2,
shuffle=True):
    """Return training and validation data loaders for CIFAR-100 dataset."""
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])

    cifar100_training = torchvision.datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform_train)
```

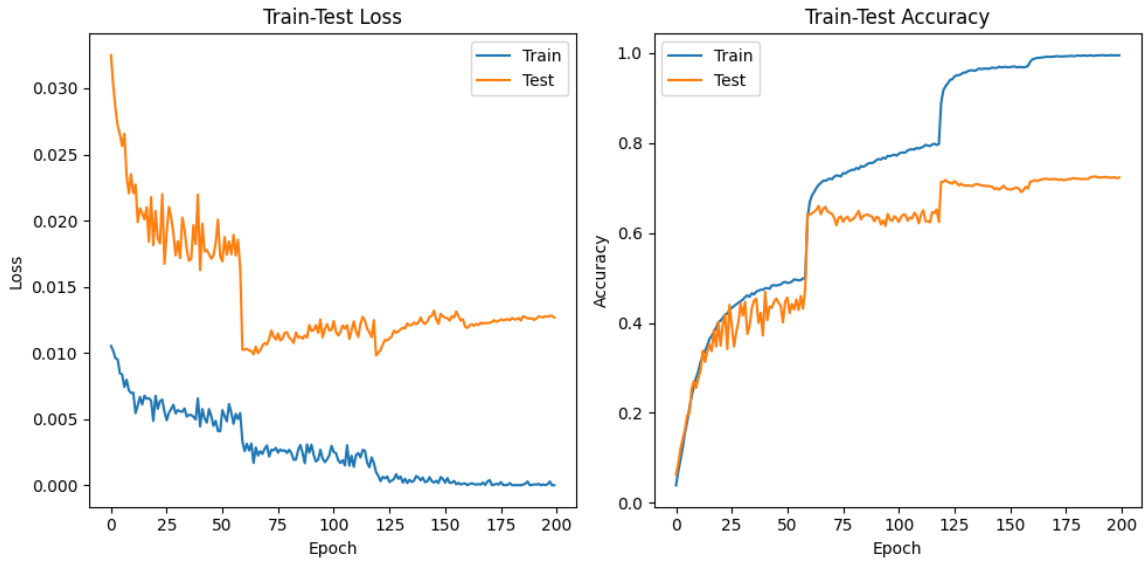
```
    # Split the dataset into training and validation sets
    train_size = int(0.8 * len(cifar100_training))
    val_size = len(cifar100_training) - train_size
    train_subset, val_subset = torch.utils.data.random_split(cifar100_training,
[train_size, val_size])
    trainloader = DataLoader(train_subset, batch_size=batch_size,
shuffle=shuffle, num_workers=2)
    valloader = DataLoader(val_subset, batch_size=batch_size, shuffle=False,
num_workers=2)
    return trainloader, valloader

def get_test_dataloader(mean, std, batch_size=16, num_workers=2, shuffle=True):
    """Return the test data loader for CIFAR-100 dataset."""
    transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])

    cifar100_test = torchvision.datasets.CIFAR100(root='./data', train=False,
download=True, transform=transform_test)
    cifar100_test_loader = DataLoader(
        cifar100_test, shuffle=shuffle, num_workers=num_workers,
batch_size=batch_size)
    return cifar100_test_loader

def compute_mean_std(cifar100_dataset):
    """Compute the mean and standard deviation of the CIFAR-100 dataset."""
    data_r = numpy.dstack([cifar100_dataset[i][1][:, :, 0] for i in
range(len(cifar100_dataset))])
    data_g = numpy.dstack([cifar100_dataset[i][1][:, :, 1] for i in
range(len(cifar100_dataset))])
    data_b = numpy.dstack([cifar100_dataset[i][1][:, :, 2] for i in
range(len(cifar100_dataset))])
    mean = numpy.mean(data_r), numpy.mean(data_g), numpy.mean(data_b)
    std = numpy.std(data_r), numpy.std(data_g), numpy.std(data_b)

    return mean, std

class WarmUpLR(_LRScheduler):
    """Learning rate scheduler for warm-up training."""
    def __init__(self, optimizer, total_iters, last_epoch=-1):
        self.total_iters = total_iters
        super().__init__(optimizer, last_epoch)

    def get_lr(self):
        """Linearly increase the learning rate during warm-up."""
        return [base_lr * self.last_epoch / (self.total_iters + 1e-8) for base_lr
in self.base_lrs]
```

کد ارائه شده شامل مجموعه ای از توابع و یک زمانبندی نرخ یادگیری است که برای آموزش شبکه های عصبی در مجموعه داده CIFAR-100 با استفاده از PyTorch طراحی شده است. تابع get_network به صورت پویا یک معماری شبکه عصبی را بر اساس نوع شبکه مشخص شده با گزینه هایی از جمله vgg16 و resnet18 انتخاب و برمی گرداند. اگر استفاده از GPU فعال باشد، شبکه به GPU منتقل می شود. تابع get_training_dataloader بعدی بارگذارهای داده را برای مجموعه‌های آموزشی و اعتبارسنجی ایجاد می‌کند و یک سری تبدیل‌ها مانند برش تصادفی و نرمال‌سازی را اعمال می‌کند. این امر پیش پردازش مناسب داده ها را برای آموزش کارآمد تضمین می کند. علاوه بر این، تابع get_test_dataloader یک بارگذار داده برای مجموعه داده آزمایشی CIFAR-100 ایجاد می‌کند و ارزیابی مدل را تسهیل می‌کند.

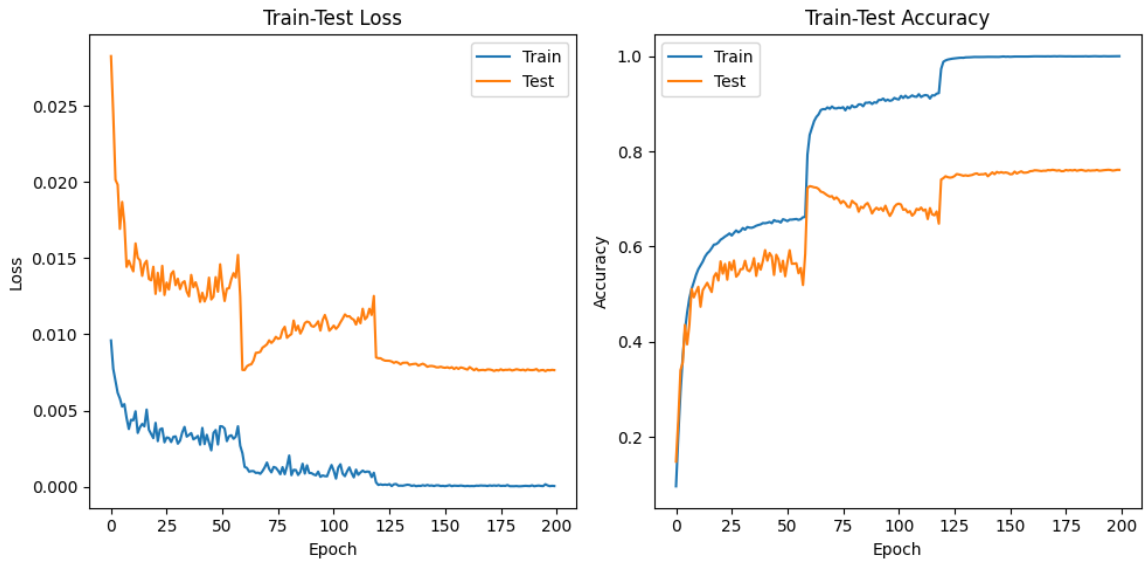
تابع compute_mean_std میانگین و انحراف استاندارد مجموعه داده CIFAR-100 را محاسبه می کند که برای عادی سازی داده ها در طول آموزش ضروری است. برای بدست آوردن آمار دقیق، کانال های رنگی هر تصویر را تجزیه و تحلیل می کند. در نهایت، کد یک زمان‌بندی نرخ یادگیری سفارشی، WarmUpLR را معرفی می‌کند که از کلاس _LRScheduler PyTorch به ارث رسیده است. این زمان‌بندی افزایش خطی در نرخ یادگیری را در طول مرحله تمرین گرم کردن پیاده‌سازی می‌کند و به مدل کمک می‌کند تا در دوره‌های اولیه به آرامی همگرا شود. زمان‌بندی نرخ یادگیری نقش مهمی در بهینه‌سازی فرآیند آموزش و افزایش عملکرد مدل دارد.

دو مدل بالا برا برای ۲۰۰ ایپاک ران کردیم و به نتایج زیر رسیدیم:

(منظور از test همان داده val است)



شکل ۳ – نمودار دقت و تابع هزینه داده های train و val برای vgg16



شکل ٤ - نمودار دقت و تابع هزینه داده های train و val برای resnet18

دقت دو مدل بالا بر روی داده های تست به شرح زیر است:

:Resnet18

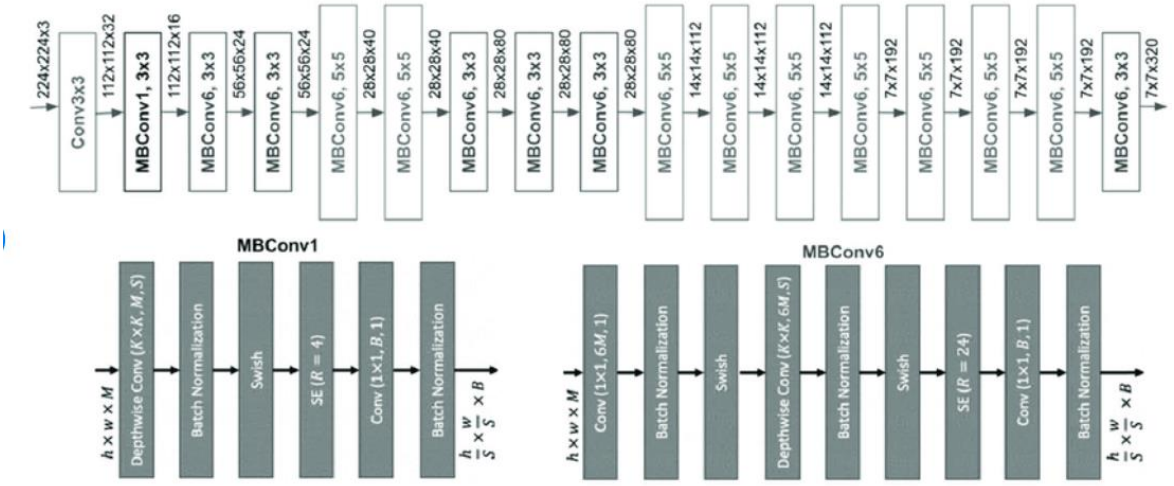
Final Test Accuracy: 76.28%

:VGG16

Final Test Accuracy: 72.29%

در مقالات این دو مدل درمورد دقت آن روی دیتاست cifar100 صحبتی نشده است، ولی دقت های بدست آمده نزدیک به اعدادی است که در اینترنت قابل پیدا شدن بود، همچنین برای بهبود دقت میتوانیم از augmentation و همچنین از lrscheduler دیگری استفاده کنیم.

EfficientNetB0 -



شکل ۵ – معماری EfficientNetB0

این مدل را به شکل زیر پیاده سازی میکنیم.

این کد یک مدل EfficientNet را با استفاده از PyTorch، پیاده سازی معماری EfficientNet، یک معماری شبکه عصبی کانولوشنال مقیاس پذیر و کارآمد تعریف می کند. این مدل از بلوک‌های ساختمانی به نام MBConv تشکیل شده است که از پیچیدگی‌های قابل تفکیک در عمق، بلوک‌های فشار و تحریک SE و عملیات drop-connect تشکیل شده است. این معماری با width_multipler, depth_multipler, do_ratio (dropout ratio), min_width, width_divisor, se_ratio (squeeze-and-excitation ratio) و dc_ratio (drop-connect ratio) پارامتر شده است. EfficientNet از چندین مرحله تشکیل شده است که با یک پایه اولیه شروع می شود و به دنبال آن یک توالی از بلوک های MBConv در مراحل مختلف سازماندهی شده اند (مرحله ۱، مرحله ۲، مرحله ۳). مرحله نهایی شامل global average pooling، dropout و یک لایه کاملاً متصل برای طبقه بندی است. تابع Efficientnet به عنوان یک روش راحت برای نمونه سازی یک مدل EfficientNet با پارامترهای مشخص ارائه شده است.

معماری EfficientNet استراتژی های مختلفی مانند مقیاس بندی ترکیبی و طراحی بلوک کارآمد را برای متعادل کردن اندازه مدل و کارایی محاسباتی معرفی کرد. کد هر بلوک ساختمانی را تعریف می کند، مدل را با پارامترهای مشخص شده مقداردهی اولیه می کند و شامل یک تابع مقداردهی اولیه وزن است. این پیاده سازی به کاربران اجازه می دهد تا با تنظیم پارامترهای مدل، مدل های EfficientNet را با پیچیدگی های مختلف و متناسب با وظایف خاص ایجاد کنند.

```
import torch.nn as nn
import torch
import torch.nn.functional as F

class conv_bn_act(nn.Module):
    def __init__(self, inchannels, outchannels, kernelsize, stride=1, dilation=1,
groups=1, bias=False, bn_momentum=0.99):
        super().__init__()
        self.block = nn.Sequential(
            SameConv(inchannels, outchannels, kernelsize, stride, dilation,
groups, bias=bias),
            nn.BatchNorm2d(outchannels, momentum=1-bn_momentum),
            swish()
        )

    def forward(self, x):
        return self.block(x)

class SameConv(nn.Conv2d):
    def __init__(self, inchannels, outchannels, kernelsize, stride=1, dilation=1,
groups=1, bias=False):
        super().__init__(inchannels, outchannels, kernelsize, stride,
                         padding=0, dilation=dilation, groups=groups, bias=bias)

    def how_padding(self, n, kernel, stride, dilation):
        out_size = (n + stride - 1) // stride
        real_kernel = (kernel - 1) * dilation + 1
        padding_needed = max(0, (out_size - 1) * stride + real_kernel - n)
        is_odd = padding_needed % 2
        return padding_needed, is_odd

    def forward(self, x):
        row_padding_needed, row_is_odd = self.how_padding(x.size(2),
self.weight.size(2), self.stride[0], self.dilation[0])
        col_padding_needed, col_is_odd = self.how_padding(x.size(3),
self.weight.size(3), self.stride[1], self.dilation[1])
        if row_is_odd or col_is_odd:
            x = F.pad(x, [0, col_is_odd, 0, row_is_odd])

        return F.conv2d(x, self.weight, self.bias, self.stride,
                        (row_padding_needed//2, col_padding_needed//2),
self.dilation, self.groups)

class swish(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
```

```
        return x * torch.sigmoid(x)


class SE(nn.Module):
    def __init__(self, inchannels, mid):
        super().__init__()
        self.AvgPool = nn.AdaptiveAvgPool2d(1)
        self.SEblock = nn.Sequential(
            nn.Linear(inchannels, mid),
            swish(),
            nn.Linear(mid, inchannels)
        )

    def forward(self, x):
        out = self.AvgPool(x)
        out = out.view(x.size(0), -1)
        out = self.SEblock(out)
        out = out.view(x.size(0), x.size(1), 1, 1)
        return x * torch.sigmoid(out)


class drop_connect(nn.Module):
    def __init__(self, survival=0.8):
        super().__init__()
        self.survival = survival

    def forward(self, x):
        if not self.training:
            return x

        random = torch.rand((x.size(0), 1, 1, 1), device=x.device)
        random += self.survival
        random.requires_grad = False
        return x / self.survival * torch.floor(random)

import torch.nn as nn
import math

class MBConv(nn.Module):
    def __init__(self, inchannels, outchannels, expan, kernelsize, stride,
se_ratio=4,
                 is_skip=True, dc_ratio=(1-0.8), bn_momentum=0.90):
        super().__init__()
        mid = expan * inchannels
        self.pointwise1 = conv_bn_act(inchannels, mid, 1) if expan != 1 else
nn.Identity()
        self.depthwise = conv_bn_act(mid, mid, kernelsize, stride=stride,
groups=mid)
```

```
        self.se = SE(mid, int(inchannels/se_ratio))
        self.pointwise2 = nn.Sequential(
            SameConv(mid, outchannels, 1),
            nn.BatchNorm2d(outchannels, 1-bn_momentum)
        )
        self.skip = is_skip and inchannels == outchannels and stride == 1
        self.dc = nn.Identity()

    def forward(self, x):
        residual = self.pointwise1(x)
        residual = self.depthwise(residual)
        residual = self.se(residual)
        residual = self.pointwise2(residual)
        if self.skip:
            residual = self.dc(residual)
            out = residual + x
        else:
            out = residual

        return out


class MBblock(nn.Module):
    def __init__(self, inchannels, outchannels, expan, kernelsize, stride,
se_ratio, repeat,
                 is_skip, dc_ratio=(1-0.8), bn_momentum=0.90):
        super().__init__()

        layers = []
        layers.append(MBConv(inchannels, outchannels, expan, kernelsize, stride,
                             se_ratio, is_skip, dc_ratio, bn_momentum))
        while repeat-1:
            layers.append(MBConv(outchannels, outchannels, expan, kernelsize, 1,
                                 se_ratio, is_skip, dc_ratio, bn_momentum))
            repeat = repeat - 1

        self.block = nn.Sequential(*layers)

    def forward(self, x):
        return self.block(x)


class EfficientNet(nn.Module):
    def __init__(self, width_multipler, depth_multipler, do_ratio, min_width=0,
width_divisor=8,
                 se_ratio=4, dc_ratio=(1-0.8), bn_momentum=0.90, num_class=100):
        super().__init__()
```

```
        def renew_width(x):
            min = max(min_width, width_divisor)
            x *= width_multipler
            new_x = max(min, int((x + width_divisor/2) // width_divisor *
width_divisor))

            if new_x < 0.9 * x:
                new_x += width_divisor
            return int(new_x)

        def renew_depth(x):
            return int(math.ceil(x * depth_multipler))

        self.stage1 = nn.Sequential(
            SameConv(3, renew_width(32), 3),
            nn.BatchNorm2d(renew_width(32), momentum=bn_momentum),
            swish()
        )
        self.stage2 = nn.Sequential(
                    # inchannels     outchannels  expand
k  s(mobilenetv2)  repeat      is_skip
            MBblock(renew_width(32), renew_width(16), 1, 3, 1, se_ratio,
renew_depth(1), True, dc_ratio, bn_momentum),
            MBblock(renew_width(16), renew_width(24), 6, 3, 2, se_ratio,
renew_depth(2), True, dc_ratio, bn_momentum),
            MBblock(renew_width(24), renew_width(40), 6, 5, 2, se_ratio,
renew_depth(2), True, dc_ratio, bn_momentum),
            MBblock(renew_width(40), renew_width(80), 6, 3, 2, se_ratio,
renew_depth(3), True, dc_ratio, bn_momentum),
            MBblock(renew_width(80), renew_width(112), 6, 5, 1, se_ratio,
renew_depth(3), True, dc_ratio, bn_momentum),
            MBblock(renew_width(112), renew_width(192), 6, 5, 1, se_ratio,
renew_depth(4), True, dc_ratio, bn_momentum),
            MBblock(renew_width(192), renew_width(320), 6, 3, 1, se_ratio,
renew_depth(1), True, dc_ratio, bn_momentum)
        )
        self.stage3 = nn.Sequential(
            SameConv(renew_width(320), renew_width(1280), 1, stride=1),
            nn.BatchNorm2d(renew_width(1280), bn_momentum),
            swish(),
            nn.AdaptiveAvgPool2d(1),
            nn.Dropout(do_ratio)
        )
        self.FC = nn.Linear(renew_width(1280), num_class)
        self.init_weights()

    def init_weights(self):
        for m in self.modules():
```

```
            if isinstance(m, SameConv):
                nn.init.kaiming_normal_(m.weight, mode='fan_out')
            elif isinstance(m, nn.Linear):
                bound = 1/int(math.sqrt(m.weight.size(1)))
                nn.init.uniform(m.weight, -bound, bound)

    def forward(self, x):
        out = self.stage1(x)
        out = self.stage2(out)
        out = self.stage3(out)
        out = out.view(out.size(0), -1)
        out = self.FC(out)
        return out


def efficientnet(width_multipler, depth_multipler, num_class=100,
bn_momentum=0.90, do_ratio=0.2):
    return EfficientNet(width_multipler, depth_multipler,
                        num_class=num_class, bn_momentum=bn_momentum,
do_ratio=do_ratio)
```

۱. بلوک ها

conv_bn_act:

این کلاس یک بلوک کانولوشنال پایه با نرمال سازی دسته ای و یک تابع فعال سازی چرخشی را نشان می دهد. از کلاس SameConv استفاده می‌کند که استاندارد nn.Conv2d را گسترش می‌دهد تا لایه صفر را به گونه‌ای مدیریت کند که ابعاد فضایی را حفظ کند. کلاس swish تابع فعال سازی swish را تعریف می کند.

SameConv:

پسوند nn.Conv2d که padding را برای اطمینان از یکسان بودن ابعاد فضایی خروجی با ورودی معرفی می کند. این برای حفظ اطلاعات در سراسر لایه ها بسیار مهم است.

swish:

تابع فعال‌سازی swish را پیاده‌سازی می‌کند، یک تابع فعال‌سازی صاف و قابل تمایز که مشخص شده است در معماری شبکه‌های عصبی به خوبی کار می‌کند.

:SE

بلوک Squeeze-and-Excitation را تعریف می کند، که به طور تطبیقی پاسخ های ویژگی کانال را مجددا کالیبره می کند. این شامل global average pooling، چند لایه خطی، و یک فعال سازی swish است.

:drop_connect

عملیات drop-connect را نشان می‌دهد، شکلی از tochastic depth regularization. در طول آموزش، واحدهای تصادفی حذف می‌شوند و شبکه را مجبور می‌کند ویژگی‌های قوی‌تری را بیاموزد.

:MBConv

بلوک گلوگاه معکوس موبایل، یک جزء کلیدی .EfficientNetاین شامل پیچش های نقطه ای (۱x1) و عمقی ۳ در ۳، فشرده سازی و تحریک، و اتصال skip connection با اتصال دراپ است.

:MBblock

تجمعی از چندین بلوک MBConv که یک بلوک با ساختارهای تکراری را تشکیل می دهد. این امکان ایجاد شبکه های عمیق تر و گویاتر را فراهم می کند.

۲. مدل EfficientNe::

:EfficientNet

این کلاس ساختار کلی مدل EfficientNet را تعریف می کند. این شامل سه مرحله است - پایه اولیه، دنباله ای از بلوک های MBConv مرحله ۱، مرحله)، و مرحله نهایی با ,global average pooling، dropout، و یک لایه کاملاً متصل برای طبقه بندی. تابع init_weights وزن اجزای مدل را مقدار دهی اولیه می کند. یک تابع راحت برای ایجاد نمونه‌ای از مدل EfficientNet با ضرب‌کننده‌های عرض و عمق مشخص، نسبت خروج، مومنتوم برای عادی‌سازی دسته‌ای و تعداد کلاس‌های خروجی.

۳. مقداردهی اولیه مدل:

وزن‌های مدل با استفاده از تکنیک‌هایی مانند kaiming_normal برای لایه‌های کانولوشنال و مقداردهی اولیه یکنواخت برای لایه‌های خطی مقداردهی اولیه می‌شوند و از پویایی یادگیری مناسب در طول آموزش اطمینان می‌دهند.

۴. مقیاس بندی مرکب Compound Scaling :

معماری مدل از اصول مقیاس بندی ترکیبی پیروی می کند و هم عرض و هم عمق را تنظیم می کند تا تعادل خوبی بین اندازه مدل و کارایی محاسباتی حاصل شود.

- Train مدل

ابتدا repository زیر را clone میکنیم:

```
! git clone https://github.com/K-Hooshanfar/small-net-cifar100
```

با استفاده از دستورات زیر مدل بالا را ترین میکنیم:

```
!python train.py -net efficientnetb0
```

کد train آن بصورت زیر است که در ادامه آن را توضیح میدهیم:

```
# Import necessary libraries
import torch.nn as nn
import torch
import os
import sys
import argparse
import torchvision.datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.optim as optim
import datetime
import torch.cuda
import matplotlib.pyplot as plt

# CIFAR-100 dataset mean and standard deviation for normalization
CIFAR100_TRAIN_MEAN = (0.5070751592371323, 0.48654887331495095,
0.4409178433670343)
CIFAR100_TRAIN_STD = (0.2673342858792401, 0.2564384629170883,
0.27615047132568404)

# Checkpoint path for saving model and logs
CHECK_POINT_PATH = "./checkpoint"

# Learning rate milestones for scheduler
MILESTONES = [60, 120, 160]


def training():
    # Set the model to train mode
    net.train()
    length = len(trainloader)
```

```
    total_sample = len(trainloader.dataset)
    total_loss = 0
    correct = 0

    # Iterate through batches in the training data
    for step, (x, y) in enumerate(trainloader):
        x = x.cuda()
        y = y.cuda()

        # Zero gradients, perform forward and backward passes, and update weights
        optimizer.zero_grad()
        output = net(x)
        loss = loss_function(output, y)
        loss.backward()
        optimizer.step()

        # Update training statistics
        total_loss += loss.item()
        _, predict = torch.max(output, 1)
        correct += (predict == y).sum()

        # Write step information to the step log file
        fstep.write("Epoch:{}\t Step:{}\t TrainedSample:{}\t TotalSample:{}\t
Loss:{:.3f}\n".format(
                epoch+1, step+1, step*args.b + len(y), total_sample, loss.item()
            ))
        fstep.flush()

        # Print training progress every 10 steps
        if step % 10 == 0:
            print("Epoch:{}\t Step:{}\t TrainedSample:{}\t TotalSample:{}\t
Loss:{:.3f}".format(
                epoch+1, step+1, step*args.b + len(y), total_sample, loss.item()
            ))

    # Write epoch information to the epoch log file
    fepoch.write("Epoch:{}\t Loss:{:.3f}\t lr:{:.5f}\t acc:{:.3%}\n".format(
        epoch + 1, total_loss/length, optimizer.param_groups[0]['lr'],
float(correct)/ total_sample
    ))
    fepoch.flush()
    return correct, total_sample, total_loss/length


def evaluating():
    # Set the model to evaluation mode
    net.eval()
    length = len(valloader)
```

```
    total_sample = len(valloader.dataset)
    total_loss = 0
    correct = 0

    # Iterate through batches in the validation data
    for step, (x, y) in enumerate(valloader):
        x = x.cuda()
        y = y.cuda()

        # Perform forward pass without gradient computation
        output = net(x)
        _, predict = torch.max(output, 1)
        torch.cuda.synchronize()
        loss = loss_function(output, y)
        total_loss += loss.item()
        correct += (predict == y).sum()

    # Calculate accuracy and write evaluation information to the eval log file
    acc = float(correct) / total_sample
    feval.write("Epoch:{}\t Loss:{:.3f}\t lr:{:.5f}\t acc:{:.3%}\n".format(
        epoch + 1, total_loss / length, optimizer.param_groups[0]['lr'], acc
    ))
    feval.flush()
    return acc, total_loss/length, total_loss


if __name__ == '__main__':
    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument("-net", default='efficientnetb0', help='net type')
    parser.add_argument("-b", default=128, type=int, help='batch size')
    parser.add_argument("-lr", default=0.1, help='initial learning rate',
type=int)
    parser.add_argument("-e", default=200, help='EPOCH', type=int)
    parser.add_argument("-optim", default="SGD", help='optimizer')
    args = parser.parse_args()

    # Data preprocessing
    transform_train = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(mean=CIFAR100_TRAIN_MEAN, std=CIFAR100_TRAIN_STD)
    ])

    # Load CIFAR-100 dataset and split into train and validation subsets
    traindata = torchvision.datasets.CIFAR100(root='./data', train=True,
download=True, transform=transform_train)
```

```
    train_size = int(0.8 * len(traindata))
    val_size = len(traindata) - train_size
    train_subset, val_subset = torch.utils.data.random_split(traindata,
[train_size, val_size])
    trainloader = DataLoader(train_subset, batch_size=args.b, shuffle=True,
num_workers=2)
    valloader = DataLoader(val_subset, batch_size=args.b, shuffle=False,
num_workers=2)

    # Define neural network architecture
    if args.net == 'efficientnetb0':
        from models.efficientnet import efficientnet
        print("loading net")
        net = efficientnet(1, 1, 100, bn_momentum=0.9).cuda()
        print("loading finish")
    else:
        print('We don\'t support this net.')
        sys.exit()

    # Define loss, optimizer, learning rate scheduler, and checkpoint path
    print("defining training")
    loss_function = nn.CrossEntropyLoss()
    if args.optim == "SGD":
        optimizer = optim.SGD(net.parameters(), lr=args.lr, momentum=0.9,
weight_decay=5e-4)
    else:
        optimizer = optim.RMSprop(net.parameters(), lr=args.lr, momentum=0.9,
weight_decay=5e-4)
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,
milestones=MILESTONES, gamma=0.2, last_epoch=-1)
    time = str(datetime.date.today() + datetime.timedelta(days=1))
    checkpoint_path = os.path.join(CHECK_POINT_PATH, args.net, time)
    if not os.path.exists(checkpoint_path):
        os.makedirs(checkpoint_path)
    print("defining finish")

    # Train and evaluate the model
    best_acc = 0
    total_time = 0

    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    with open(os.path.join(checkpoint_path, 'EpochLog.txt'), 'w') as fepoch:
        with open(os.path.join(checkpoint_path, 'StepLog.txt'), 'w') as fstep:
```

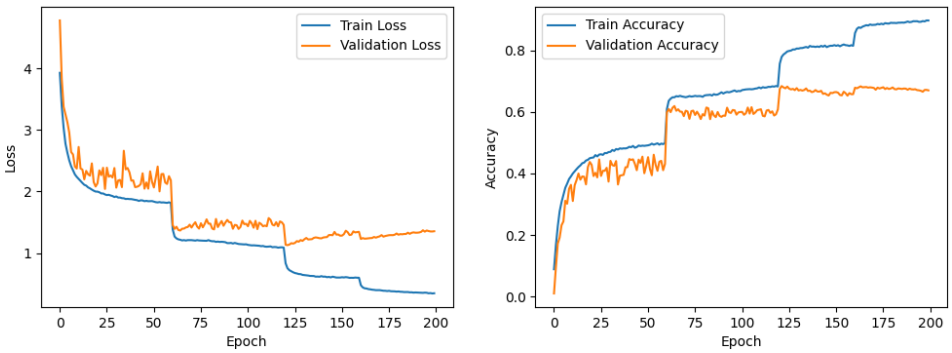
```
            with open(os.path.join(checkpoint_path, 'EvalLog.txt'), 'w') as
feval:
                with open(os.path.join(checkpoint_path, 'Best.txt'), 'w') as
fbest:
                    print("start training")
                    for epoch in range(args.e):
                        correct, total_sample, averagelosstrain = training()
                        print("evaluating")
                        accuracy, averageloss, total_loss = evaluating()

                        # Append values for plotting
                        train_losses.append(averagelosstrain)
                        val_losses.append(averageloss)
                        train_accuracies.append(float(correct) / total_sample)
                        val_accuracies.append(accuracy)
                        scheduler.step()
                        print("saving regular")
                        torch.save(net.state_dict(),
os.path.join(checkpoint_path, 'regularParam.pth'))
                        # if accuracy > best_acc:
                        print("saving best")
                        torch.save(net.state_dict(),
os.path.join(checkpoint_path, 'bestParam.pth'))

                        fbest.write("Epoch:{}\t Loss:{:.3f}\t lr:{:.5f}\t
acc:{:.3%}\n".format(
                                    epoch + 1, averageloss,
optimizer.param_groups[0]['lr'], accuracy
                                ))
                        fbest.flush()
                        best_acc = accuracy

    # Plotting
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.savefig(os.path.join(checkpoint_path, 'loss_plot.png'))
    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, label='Train Accuracy')
    plt.plot(val_accuracies, label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.savefig(os.path.join(checkpoint_path, 'accuracy_plot.png'))
```

```
    plt.tight_layout()
    plt.show()
```

این کد آموزش و ارزیابی را برای یک شبکه عصبی، به ویژه معماری EfficientNet، بر روی مجموعه داده CIFAR-100 پیاده سازی می کند. با وارد کردن کتابخانه های لازم و تعریف ثابت هایی مانند میانگین و انحراف استاندارد CIFAR-100 برای نرمال سازی داده ها، checkpoint paths و نقاط عطف نرخ یادگیری شروع می شود. تابع آموزش از طریق دسته‌ای از مجموعه داده آموزشی تکرار می‌شود، پاس‌های رو به جلو و عقب را انجام می‌دهد، وزن‌ها را به‌روزرسانی می‌کند و اطلاعات آموزشی را ثبت می‌کند. تابع ارزیابی عملکرد مدل را در مجموعه اعتبارسنجی بدون به‌روزرسانی وزن ارزیابی می‌کند. سپس اسکریپت اصلی آرگومان های خط فرمان را تجزیه می کند، بارگذارهای داده را مقداردهی اولیه می کند، شبکه عصبی، تابع هزینه، بهینه ساز و زمانبندی نرخ یادگیری را تعریف می کند. این مدل را در چندین دوره آموزش می دهد و ارزیابی می کند و checkpoints و logs مربوط را ذخیره می کند. در نهایت، منحنی‌های loss/دقت آموزش و اعتبارسنجی را ترسیم و ذخیره می‌کند.

مدل EfficientNet بر اساس آرگومان ها انتخاب می شود و حلقه آموزشی فرآیندهای بهینه سازی و ارزیابی را مدیریت می کند. کد اطلاعات دقیق را در هر دوره و مرحله ثبت می کند و پارامترهای مدل را در طول آموزش منظم و هر زمان که بهترین دقت جدید به دست می آید ذخیره می کند. علاوه بر این، اسکریپت نمودارهایی را ایجاد می کند که منحنی های یادگیری را برای از دست دادن و دقت نشان می دهد.

مدل بالا برا برای ۲۰۰ ایپاک ران کردیم و به نتایج زیر رسیدیم:



شکل ٦ – نمودار دقت و loss مدل EfficientNetB0

دقتدو مدل بالا بر روی داده های تست به شرح زیر است:

:EfficientNetB0

Final Test Accuracy: 69.78%

در مقاله این مدل درمورد دقت آن روی دیتاست cifar100 صحبتی شده است، و دقت آن را در پایین مشاهده میکنیم، همچنین برای بهبود دقت میتوانیم از augmentation و همچنین از lrscheduler دیگری استفاده کنیم.

Table 5. **EfficientNet Performance Results on Transfer Learning Datasets**. Our scaled EfficientNet models achieve new state-of-the-art accuracy for 5 out of 8 datasets, with 9.6x fewer parameters on average.

	Comparison to best public-available results						Comparison to best reported results					
	Model	Acc.	#Param	Our Model	Acc.	#Param(ratio)	Model	Acc.	#Param	Our Model	Acc.	#Param(ratio)
CIFAR-10	NASNet-A	98.0%	85M	EfficientNet-B0	98.1%	4M (21x)	†Gpipe	**99.0%**	556M	EfficientNet-B7	98.9%	64M (8.7x)
CIFAR-100	NASNet-A	87.5%	85M	EfficientNet-B0	88.1%	4M (21x)	Gpipe	91.3%	556M	EfficientNet-B7	**91.7%**	64M (8.7x)
Birdsnap	Inception-v4	81.8%	41M	EfficientNet-B5	82.0%	28M (1.5x)	GPipe	83.6%	556M	EfficientNet-B7	**84.3%**	64M (8.7x)
Stanford Cars	Inception-v4	93.4%	41M	EfficientNet-B3	93.6%	10M (4.1x)	‡DAT	**94.8%**	-	EfficientNet-B7	94.7%	-
Flowers	Inception-v4	98.5%	41M	EfficientNet-B5	98.5%	28M (1.5x)	DAT	97.7%	-	EfficientNet-B7	**98.8%**	-
FGVC Aircraft	Inception-v4	90.9%	41M	EfficientNet-B3	90.7%	10M (4.1x)	DAT	92.9%	-	EfficientNet-B7	**92.9%**	-
Oxford-IIIT Pets	ResNet-152	94.5%	58M	EfficientNet-B4	94.8%	17M (5.6x)	GPipe	**95.9%**	556M	EfficientNet-B6	95.4%	41M (14x)
Food-101	Inception-v4	90.8%	41M	EfficientNet-B4	91.5%	17M (2.4x)	GPipe	93.0%	556M	EfficientNet-B7	**93.0%**	64M (8.7x)
Geo-Mean						**(4.7x)**						**(9.6x)**

†GPipe (Huang et al., 2018) trains giant models with specialized pipeline parallelism library.
‡DAT denotes domain adaptive transfer learning (Ngiam et al., 2018). Here we only compare ImageNet-based transfer learning results.
Transfer accuracy and #params for NASNet (Zoph et al., 2018), Inception-v4 (Szegedy et al., 2017), ResNet-152 (He et al., 2016) are from (Kornblith et al., 2019).

شکل ۷ ــ دقت مقاله

ب)

برای بدست آوردن ۵ شاخص هندسی ابتدا بصورت متوازن داده ها را به ۳ دسته تقسیم میکنیم:

```
# Set random seeds for reproducibility
random_seed = 42
torch.manual_seed(random_seed)
np.random.seed(random_seed)
random.seed(random_seed)

# Load the CIFAR-100 dataset and create a balanced subset
transform = transforms.Compose([transforms.ToTensor()])
# Load CIFAR-100 dataset
cifar100_dataset = datasets.CIFAR100(root='./data', train=True, download=True,
transform=transform)

# Split the dataset into training and validation sets
train_size = int(0.8 * len(cifar100_dataset))
val_size = len(cifar100_dataset) - train_size
```

```
cifar100_traindataset, cifar100_valdataset =
torch.utils.data.random_split(cifar100_dataset, [train_size, val_size])

# Define the subset size
subset_fraction = 0.9
subset_size_train = int(subset_fraction * len(cifar100_traindataset))

class_indices = list(range(len(cifar100_traindataset.dataset.classes)))
class_subset_size = int(subset_size_train /
len(cifar100_traindataset.dataset.classes))

class_sampler_indices_train = []

for class_index in class_indices:
    class_indices_list_train = [i for i, label in
enumerate(cifar100_traindataset.dataset.targets) if label == class_index]
    class_sampler_indices_train.extend(class_indices_list_train[:class_subset_siz
e])

train_sampler = SubsetRandomSampler(class_sampler_indices_train)

batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_traindataset,
batch_size=batch_size, sampler=train_sampler)

# Check the number of samples in the balanced train set
print(f"Balanced Train set size: {len(train_loader.sampler)}")


# Define data transformations
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5))])

# Load CIFAR-100 dataset
cifar100_test = datasets.CIFAR100(root='./data', train=False, download=True,
transform=transform)

# Create PyTorch data loaders
batch_size = 128
test_loader = torch.utils.data.DataLoader(cifar100_test, batch_size=batch_size,
shuffle=False)
```

این کد یک زیر مجموعه متعادل از مجموعه داده CIFAR-100 را برای آموزش یک شبکه عصبی آماده می کند. ابتدا مجموعه داده CIFAR-100 را بارگیری می کند و آن را به مجموعه های آموزشی و اعتبار سنجی تقسیم می کند. برای ایجاد یک زیرمجموعه متعادل، کسری از مجموعه آموزشی در نظر گرفته

می‌شود و برای هر کلاس، تعداد مشخصی از نمونه‌ها به‌طور تصادفی انتخاب می‌شود و از نمایش متعادلی اطمینان می‌یابد. سپس بارگذار داده آموزشی با استفاده از نمونه‌بردار تصادفی زیر مجموعه پیکربندی می‌شود تا این زیر مجموعه متعادل را منعکس کند. بخش دوم کد بر روی مجموعه آزمایشی تمرکز می کند، مجموعه داده آزمایشی CIFAR-100 را بارگذاری می کند و یک بارگذار داده PyTorch ایجاد می کند.

در قدم بعدی برای بدست آوردن features و labels کد زیر را اجرا میکنیم: (برای هر ۳ مدل روند ماند همدیگر است، فقط اسم مدل را باید تغییر دهیم)

```
model = vgg16_bn()   # change it to resnet 18 and efficientnet
model.to(device)

# Load pre-trained weights - according to the model
pretrained_weights_path = '/content/vgg16-200-regular.pth'
model.load_state_dict(torch.load(pretrained_weights_path))
model = nn.Sequential(*list(model.children())[:-1])

# Define your CIFAR-100 dataloader
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

features = []
labels = []

# Set the model to evaluation mode
model.eval()

with torch.no_grad():
    for inputs, targets in tqdm(train_loader):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda')

        # Forward pass through the model.features
        features_batch = model(inputs)

        # Append the extracted features and labels
        features.append(features_batch)
        labels.append(targets)

        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()
```

این کد ویژگی ها را از یک مدل از پیش آموزش دیده در مجموعه داده CIFAR-100 استخراج می کند و آن را به دستگاه موجود CPU یا GPU منتقل می کند. وزن های از قبل آموزش دیده از یک فایل مشخص بارگذاری می شوند. آخرین لایه fully connected مدل برداشته می شود و فقط لایه های کانولوشن باقی می ماند. سپس کد یک دیتالودر CIFAR-100 را تعریف می کند و تغییرات لازم را اعمال می کند. در طول یک حلقه ارزیابی بر روی داده های آموزشی، ویژگی های مدل با استفاده از مدل استخراج می شوند. این ویژگی‌ها، همراه با برچسب‌های مربوطه، در لیست‌های «ویژگی‌ها» و «برچسب‌ها» جمع‌آوری و ذخیره می‌شوند. برای مدیریت حافظه GPU، ورودی ها حذف می شوند و حافظه نهان GPU پس از هر بار تکرار پاک می شود. ویژگی‌های استخراج‌شده روی هم چیده شده و به یک تانسور دوبعدی برای استفاده بعدی در آموزش یا ارزیابی مدل دیگری تغییر شکل می‌دهند و به عنوان استخراج‌کننده ویژگی برای تصاویر CIFAR-100 عمل می‌کنند. (کد بالا رو یکبار هم برای داده های تست نیز تکرار میکنیم)

در ادامه برای بدست آوردن شاخص ها همانند توضیحاتی که در ابتدای تمرین آوردیم عمل میکنیم. (کد کامل آن در فایل ضمیمه شده قرار دارد) در ادامه به نتایجی که بدست آوردیم می پردازیم.

```
# Create Instance of class
si_calculator = ARH_SeparationIndex(features, labels, normalize=True)
```

جدول ۲ – شاخص های بدست آمده برای ترین و تست قسمت الف

Model/Metric	SI	High order SI	High order Soft Si	Anti SI	Center SI
Cifar100 داده خام	0.1741	0.0732	0.1501	0.8364	0.1065
VGG16_train	0.339	0.2188	0.3128	0.593	0.3669
VGG16_test	0.5724	0.481	0.568	0.3441	0.6365
Resnet18_train	0.4858	0.33875	0.46242	0.4139	0.55965
Resnet18_test	0.599	0.4853	0.58985	0.3056	0.7026
EfficientNetb0_train	0.4625	0.2525	0.43875	0.375	0.681
EfficientNetb0_test	0.365	0.18	0.3195	0.541	0.927

مشاهده میشود که داده های خام که شاخص SI مناسبی نداشت پس از آموزش شبکه در آخرین لایه شبکه به مقادیر به مراتب بهتر و بالاتری میرسد و نشان میدهد شبکه به خوبی توانسته داده ها را در فضای فیچر ها جدا کند. همچنین مشاهده میکنیم که متریک ها تا حدی بهتر شده است. در high و Soft عملکرد resnet بهتر از بقیه مدل ها استکه احتمالا بدلیل وجود skip connection است. در

معماری vgg و Resnet18 میبینیم که عملکرد بهتری روی داده های تست داشته اند چون SI آن بیشتر شده است که این نشاندهنده ی این است که شبکه به خوبی توانسته داده ها را جدا کند.

ج) pre-trained

- VGG16

این کد ویژگی ها را از لایه های کانولوشنال یک مدل VGG16 از پیش آموزش دیده با استفاده از PyTorch استخراج می کند. مدل VGG16 را که از قبل آموزش داده شده در ImageNet بارگیری می کند، لایه های کاملا متصل آن را حذف می کند و فقط لایه های کانولوشن را در متغیر vgg16_features حفظ می کند. استخراج ویژگی ها با استفاده از دیتالودر ارائه شده train_loader انجام می شود. تابع extract_features از طریق دیتالودر تکرار می شود و خروجی های مدل را برای هر تصویر محاسبه می کند. سپس ویژگی‌های استخراج‌شده و برچسب‌های مربوطه جمع‌آوری شده و به تنسورها متصل می‌شوند.

```
import torch
import torchvision.models as models
from torchvision import transforms
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR100

# Load pre-trained models
vgg16_model = models.vgg16(pretrained=True)
# Remove fully connected layers
vgg16_features = torch.nn.Sequential(*(list(vgg16_model.features.children())))

import torch
from tqdm import tqdm

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def extract_features(model, dataloader):
    model.eval()
    model.to(device)
    features = []
    labels = []

    with torch.no_grad():
        for images, targets in tqdm(dataloader):
            images, targets = images.to(device), targets.to(device)
            outputs = model(images)
            features.append(outputs.squeeze())
            labels.append(targets)
```

```
    features = torch.cat(features)
    labels = torch.cat(labels)

    return features, labels

# Move models to GPU
vgg16_features.to(device)

vgg16_features, vgg16_labels = extract_features(vgg16_features, train_loader)
```

همانند توضیحاتی که برای vgg16 دادیم، دو مدل دیگر رو هم به همون شیوه feature و label ها را بدست می آوریم.

```
import torch
import torchvision.models as models
from torchvision import transforms
from torch.utils.data import DataLoader
from torchvision.datasets import CIFAR100

# Load pre-trained models
resnet18_model = models.resnet18(pretrained=True)
# Remove fully connected layers
resnet18_features = torch.nn.Sequential(*(list(resnet18_model.children())[:-1]))
import torch
from tqdm import tqdm

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def extract_features(model, dataloader):
    model.eval()
    model.to(device)
    features = []
    labels = []
    with torch.no_grad():
        for images, targets in tqdm(dataloader):
            images, targets = images.to(device), targets.to(device)
            outputs = model(images)
            features.append(outputs.squeeze())
            labels.append(targets)

    features = torch.cat(features)
    labels = torch.cat(labels)
    return features, labels

# Move models to GPU
```

```
resnet18_features.to(device)

resnet18_features, resnet18_labels = extract_features(resnet18_features,
train_loader)
```

برای efficientnetb0 از مدل در این لینک استفاده کردیم چون مدلی که در pytorch بود ارور میداد و نمی توانستیم از آن استفاده کنیم.

```
!pip install -qq efficientnet_pytorch
from efficientnet_pytorch import EfficientNet
model = EfficientNet.from_pretrained('efficientnet-b0')
import torch
from tqdm import tqdm
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
def extract_features(model, dataloader):
    model.eval()
    model.to(device)
    # Extract features up to _avg_pooling
    features = []
    labels = []

    # Switch off gradient computation
    with torch.no_grad():
        for images, targets in tqdm(dataloader):
            images, targets = images.to(device), targets.to(device)

            # Forward pass up to _avg_pooling
            x = model._conv_stem(images)
            x = model._bn0(x)
            for block in model._blocks:
                x = block(x)
                if isinstance(block, models.efficientnet.SqueezeExcitation):
                    # SqueezeExcitation block, apply Swish activation
                    x = block._swish(x)
            x = model._conv_head(x)
            x = model._bn1(x)
            x = model._swish(x)
            x = model._avg_pooling(x)

            # Flatten the output before appending to features
            features.append(x.view(x.size(0), -1))
            labels.append(targets)

    features = torch.cat(features)
    labels = torch.cat(labels)

    return features, labels
```

```
# Move models to GPU
model.to(device)
efficientnet_features, efficientnet_labels = extract_features(model,
train_loader)
```

```
      )
      (_conv_head): Conv2dStaticSamePadding(
        320, 1280, kernel_size=(1, 1), stride=(1, 1), bias=False
        (static_padding): Identity()
      )
      (_bn1): BatchNorm2d(1280, eps=0.001, momentum=0.010000000000000009, affine=True, track_running_stats=True)
      (_avg_pooling): AdaptiveAvgPool2d(output_size=1)
      (_dropout): Dropout(p=0.2, inplace=False)
      (_fc): Linear(in_features=1280, out_features=1000, bias=True)
      (_swish): MemoryEfficientSwish()
    )
```

شکل ۸ – آخرین لایه های مدل

متریک ها را نیز همانند قسمت های قبل بدست می آوریم و به نتایج زیر میرسیم (دیتا ها همان دیتاهای قسمت ب هستند که بصورت متوازن هم انتخاب شده اند.)

جدول ۳ - شاخص های بدست آمده برای ترین و تست قسمت pretrained

Model/Metric	SI	High order SI	High order Soft Si	Anti SI	Center SI
Cifar100 داده خام	0.1741	0.0732	0.1501	0.8364	0.1065
VGG16_train	0.23	0.1	0.211	0.739	0.277
VGG16_test	0.22	0.095	0.205	0.686	0.35
Resnet18_train	0.229	0.097	0.2048	0.74	0.234
Resnet18_test	0.2171	0.0892	0.1959	0.6976	0.291
EfficientNetb0_train	0.1278	0.0398	0.1117	0.85	0.1034
EfficientNetb0_test	0.1023	0.0284	0.089	0.8506	0.1173

در جدول بالا مشاهده می کنیم که وقتی بدون ترین کردن مدل ها متریک ها را بدست می آوریم، متریک ها کمتر از حالتی می شود که مدل ها ترین شده اند. و اعداد نزدیک حالتی است که داده خام را متریک ها داده ایم. دلیل آن این است که مدل هنوز داده ها را یاد نگرفته، در نتیجه متریک ها با حالت قبل تفاوت دارند. همچنین مشاهده می کنیم که متریک ها در ترین و تست تفاوت آنچنانی ندارند.

سوال دوم : ارزیابی لایه های شبکه عصبی

الف) منحنی SI بر حسب لایه ها

در این بخش از شبکه از قبل آموزش داده شده که در تمرین اول کد ها و توضیحات آن آمده است استفاده میشود.

شبکه : EfficientNetV2-S

در ابتدا کتاب خانه های مورد نیاز **import** میشود.

```
import torch
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from tqdm import tqdm
import random
from torch.utils.data import Subset, DataLoader
import numpy as np
from torch.cuda.amp import autocast, GradScaler
from torch.utils.data.sampler import SubsetRandomSampler

from data_complexity_measures.models.ARH_SeparationIndex import
ARH_SeparationIndex
```

در ادامه وزن های شبکه دانلود شده و مدل اصلی لود میشود.

```
# download Pretrained Model
!gdown 1yWb1OXm7LslxkRx3zHyen-PBZ12-W2sA


# Load EfficientNetV2-S model
model = models.efficientnet_v2_s(weights='IMAGENET1K_V1')

# 100 classes
model.classifier = nn.Sequential(
    nn.Dropout(p=0.2, inplace=True),
    nn.Linear(1280, 100)
)
```

```
transform = transforms.Compose([
    transforms.Resize(300),
    transforms.CenterCrop(260),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Load your pre-trained model weights
model.load_state_dict(torch.load('efficientnetv2_s_cifar100_finetuned.pth'))


model.eval()
model.cuda()

print("Model Loaded")
```

حال دیتاست Cifar100 را دانلود و لود کرده و یک subset نرمال شامل ۷۰ درصد داده های آموزشی و تمام داده های تست را جدا میکنیم.

```
# Load the CIFAR-100 dataset and create a balanced subset
transform = transforms.Compose([transforms.ToTensor()])

cifar100_train = datasets.CIFAR100(root='./data', train=True, download=True,
transform=transform)
cifar100_test = datasets.CIFAR100(root='./data', train=False, download=True,
transform=transform)

# Combine original and augmented datasets
cifar100_train_combined = torch.utils.data.ConcatDataset([cifar100_train])
cifar100_test_combined = torch.utils.data.ConcatDataset([cifar100_test])

# Define the size of the balanced subset for both train and test sets
subset_fraction_train = 0.7
subset_fraction_test = 1

# Calculate the number of samples needed for the balanced subset for both train
and test sets
subset_size_train = int(subset_fraction_train * len(cifar100_train_combined))
subset_size_test = int(subset_fraction_test * len(cifar100_test_combined))

# Create a balanced subset for both train and test sets using SubsetRandomSampler
class_indices_train = list(range(len(cifar100_train.classes)))
class_indices_test = list(range(len(cifar100_test.classes)))

class_subset_size_train = int(subset_size_train / len(cifar100_train.classes))
class_subset_size_test = int(subset_size_test / len(cifar100_test.classes))
```

```
class_sampler_indices_train = []
class_sampler_indices_test = []

for class_index in class_indices_train:
    class_indices_list_train = [i for i, label in
enumerate(cifar100_train.targets) if label == class_index]
    class_sampler_indices_train.extend(class_indices_list_train[:class_subset_siz
e_train])

for class_index in class_indices_test:
    class_indices_list_test = [i for i, label in enumerate(cifar100_test.targets)
if label == class_index]
    class_sampler_indices_test.extend(class_indices_list_test[:class_subset_size_
test])

train_sampler = SubsetRandomSampler(class_sampler_indices_train)
test_sampler = SubsetRandomSampler(class_sampler_indices_test)

# Create PyTorch data loaders using the balanced subset for both train and test
sets
batch_size = 256
train_loader = torch.utils.data.DataLoader(cifar100_train_combined,
batch_size=batch_size, sampler=train_sampler)
test_loader = torch.utils.data.DataLoader(cifar100_test_combined,
batch_size=batch_size, sampler=test_sampler)

# Check the number of samples in each set
print(f"Balanced Train set size: {len(train_loader.sampler)}")
print(f"Balanced Test set size: {len(test_loader.sampler)}")
```

قبل از هر چیزی به کمک مدل load شده تعداد لایه ها را بررسی میکنیم.

```
# Counter Num Layer
counter = 0
for idx, layer in enumerate(model.features):
  counter +=1
print(f"Number of layer is : {counter}")

Number of layer is : 8
```

همانطور که مشاهده میشود کد شبکه به گونه ۸ لایه ای نوشته شده است که شرح لایه ها در تصویر زیر آمده است.[1]

Table 4. EfficientNetV2-S architecture – MBConv and Fused-MBConv blocks are described in Figure 2.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

شکل ۹ : لایه های شبکه EfficientNetV2-S

پس از مشخص شدن تعداد لایه ها به کمک کد زیر به هر لایه hook اضافه میشود تا بتوانیم در هر لایه خروجی فیچر ها را بگیریم.

```
# Prepare storage for outputs and labels
features_per_layer = [[] for _ in range(len(model.features))]
labels_list = []

# Function to attach hooks
def get_layer_outputs(layer_idx):
    def hook(module, input, output):
        features_per_layer[layer_idx].append(output.detach())
    return hook

# Attach hooks to each layer
for idx, layer in enumerate(model.features):
    layer.register_forward_hook(get_layer_outputs(idx))
```

[1] https://arxiv.org/pdf/2104.00298.pdf

پس از انجام این کار اکنون میتوانیم مدل را بر روی داده های ترین ویا تست ران کرده و خروجی های فیچر هر لایه و همچنین لیبل ها را در لیست هایی ذخیره کنیم.

کد زیر فرایند فوق را انجام میدهد. (برای داده های ترین)

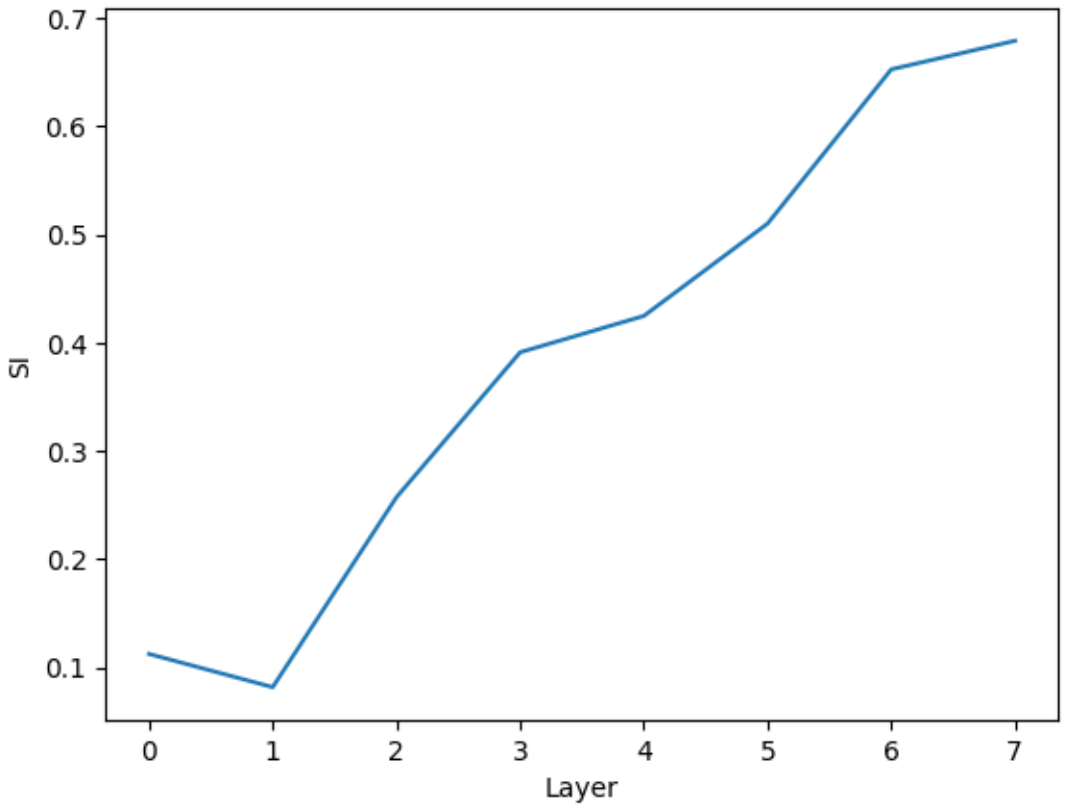
```
# Pass data through the model and collect layer outputs
with torch.no_grad():
    for inputs, targets in tqdm(train_loader):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda')

        # Trigger the hooks and collect layer outputs
        model(inputs)
        labels_list.append(targets)

        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()

# Post-process the data: Flatten and concatenate
for idx, layer_features in enumerate(features_per_layer):
    layer_features = torch.cat([f.view(f.size(0), -1) for f in
layer_features])
    features_per_layer[idx] = layer_features

labels = torch.cat(labels_list)
```

حال که فیچر های تمام لایه ها به دست آورده شد وقت آن است تا از الگوریتم SI پیاده سازی شده استفاده کرده و SI را به ازای خروجی هر لایه بدست بیاوریم. برای انجام این کار از کد زیر استفاده میکنیم.

```
si_layer_train =[]
for features in features_per_layer:
  instance_disturbance = ARH_SeparationIndex(features, labels,
normalize=True)
  si = instance_disturbance.si()
  si_layer_train.append(si)
```

مقادیر SI هر لایه به شرح زیر میباشد و منحنی آن نیز در ادامه کشیده شده است.

```
[0.1122, 0.08148, 0.257, 0.391, 0.4246, 0.50996, 0.65252, 0.67892]
```



شکل ۱۰ : منحنی SI بر حسب هر لایه برای داده های train

همانطور که مشاهده میشود مطابق انتظار خروجی هر لایه در هر لایه مقدار SI را بهتر میکند و در کل منحنی سیر صعودی ای دارد و جایی نیست که بگوییم با تعداد لایه های کمتر هم میتوانستیم به همان عملکرد قبلی برسیم.

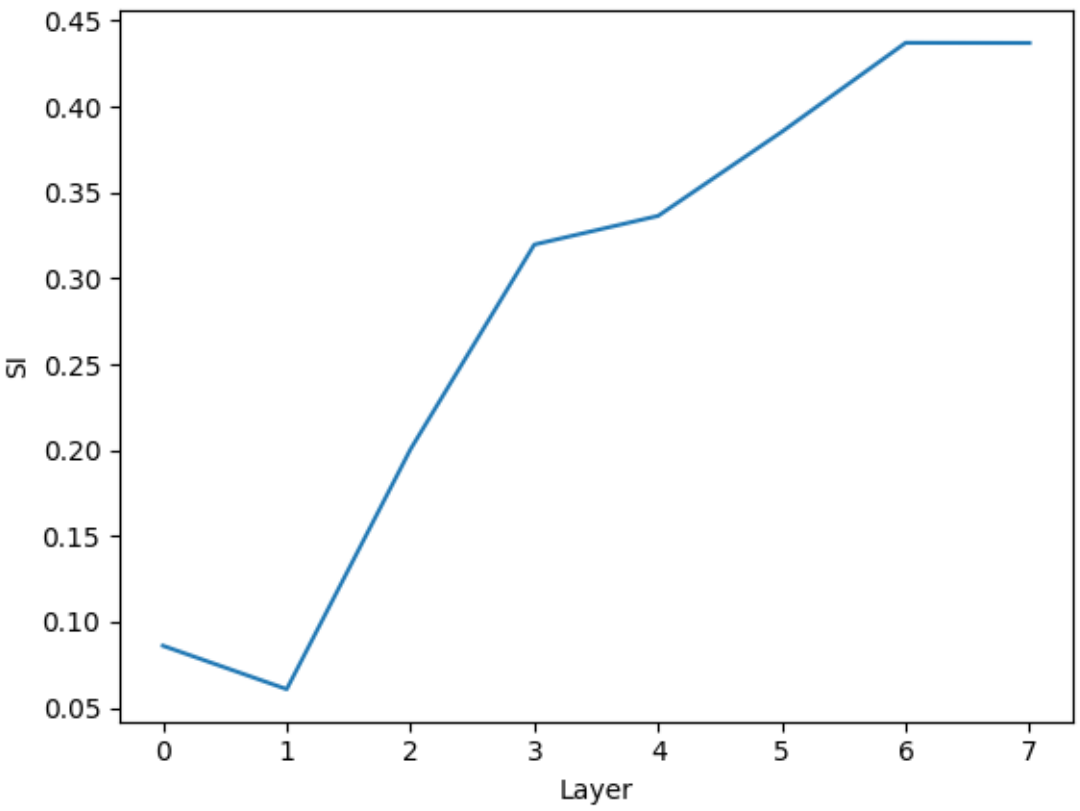
مشابه این کار را برای داده های تست به کمک کد زیر انجام میدهیم.

```
with torch.no_grad():
    for inputs, targets in tqdm(test_loader):
        if torch.cuda.is_available():
            inputs = inputs.to('cuda')

        # Trigger the hooks and collect layer outputs
        model(inputs)
        labels_list.append(targets)

        # Release GPU memory
        del inputs
        torch.cuda.empty_cache()

# Post-process the data: Flatten and concatenate
for idx, layer_features in enumerate(features_per_layer):
```

```
    layer_features = torch.cat([f.view(f.size(0), -1) for f in
layer_features])
    features_per_layer[idx] = layer_features

labels = torch.cat(labels_list)

si_layer_test =[]
for features in features_per_layer:
  instance_disturbance = ARH_SeparationIndex(features, labels,
normalize=True)
  si = instance_disturbance.si()
  si_layer_test.append(si)
```

برای داده های تست نیز مقادیر SI و منحنی آن به شرح زیر میباشد.

```
[0.0861, 0.0608, 0.2005, 0.3197, 0.3364, 0.3854, 0.4372, 0.4371]
```



شکل ۱۱ : مقادیر SI به ازای هر لایه برای داده های تست

همانطور که مشاهده میشود در این منحنی هم مطابق انتظار شبکه در هر لایه مقدار SI و میزان جدا شدن فیچر ها را بهتر میکند.

ب) ۱- بررسی روند کاهش یا افزایش SI در لایه ها

شاخص جدایی (SI) معیاری است که مشخص می کند تا چه اندازه ویژگی ها در فضای ویژگی از هم جدا شده اند. در زمینه شبکه‌های عصبی، SI بالاتر در لایه‌های عمیق‌تر معمولاً نشان می‌دهد که شبکه به طور مؤثری ویژگی‌های متمایز بیشتری را برای وظایف طبقه‌بندی می‌آموزد.

برای مدل EfficientNetV2-S که بر روی CIFAR-100 آموزش داده شده است، و بر اساس امتیازات SI که برای هر لایه در بخش قبل ارایه شده است، روند کلی افزایش SI وجود دارد که نشان دهنده بهبود جداسازی ویژگی ها با افزایش عمق شبکه است. با این حال، یک افت در SI از لایه اولیه (مرحله ۰) به لایه بعدی (مرحله ۱) مشاهده شده است. در اینجا چند توضیح احتمالی برای این مشاهده وجود دارد:

- پیچیدگی ویژگی‌ها: لایه‌های اولیه یک شبکه عصبی کانولوشن معمولاً ویژگی‌های اساسی مانند لبه‌ها، بافت‌ها و الگوهای ساده را به تصویر می‌کشند. همانطور که عمیق تر می شوید، لایه ها شروع به ترکیب این الگوهای ساده می کنند تا بازنمایی های پیچیده تر و انتزاعی را تشکیل دهند. این امکان وجود دارد که انتقال از مرحله ۰ به مرحله ۱ شامل انتقال از ویژگی های بسیار ساده به ویژگی های کمی پیچیده تر باشد، که ممکن است در ابتدا به خوبی از هم جدا نباشند.

- دینامیک یادگیری: در اوایل شبکه، فیلترها ممکن است هنوز یاد نگرفته باشند که متمایزترین ویژگی ها را استخراج کنند. لایه های اولیه شبکه ممکن است هنوز در حال یادگیری تمایز بین طبقات مختلف باشند که منجر به کاهش موقت جدایی می شود.

- تغییرات معماری شبکه: طبق تصویر ارایه شده از لایه ها ، مرحله ۱ از نوع متفاوتی از بلوک (Fused-MBConv1) در مقایسه با مرحله ۰ (Conv3x3) استفاده می کند. تفاوت‌های معماری بین این بلوک‌ها ممکن است در ابتدا منجر به جداسازی ویژگی‌های کمتر مؤثر شود، به‌ویژه از آنجایی که بلوک‌های Fused-MBConv طوری طراحی شده‌اند که از نظر محاسباتی کارآمد باشند و نه حداکثر.

- تغییرپذیری در داده ها: CIFAR-100 یک مجموعه داده پیچیده با ۱۰۰ کلاس است و لایه های اولیه ممکن است برای جدا کردن ویژگی هایی که ریز دانه هستند مشکل داشته باشند. کاهش SI می تواند نتیجه این پیچیدگی باشد و ممکن است لزوماً منعکس کننده نقص در شبکه نباشد، بلکه چالش کار در این مرحله از پردازش است.

مهم است که به یاد داشته باشیم که SI تنها یک معیار است و تصویر کاملی از عملکرد شبکه را به تصویر نمی‌کشد. عوامل دیگری مانند توانایی شبکه برای ترکیب ویژگی‌ها در لایه‌های بالاتر و دقت طبقه‌بندی کلی نیز برای ارزیابی اثربخشی یک شبکه عصبی بسیار مهم هستند. افت اولیه SI ممکن است دلیلی برای نگرانی نباشد اگر روند کلی بهبود پیدا کند و شبکه در کاری که برای آن آموزش دیده است عملکرد خوبی داشته باشد.

ب) ۲- تاثیر تفاوت هر لایه در آموزش شبکه

نوع لایه های مورد استفاده در یک شبکه عصبی به طور قابل توجهی بر پویایی آموزش و عملکرد نهایی شبکه تأثیر می گذارد. هر نوع لایه دارای ویژگی ها و جنبه های محاسباتی منحصر به فرد خود است که به طور متفاوتی به فرآیند یادگیری کمک می کند. در زمینه معماری EfficientNetV2-S و انتقال از یک لایه کانولوشن معمولی (Conv3x3) در مرحله ۰ به یک لایه Fused-MBConv1 در مرحله ۱، چندین اثر قابل بررسی است:

- کارایی پارامتر: لایه‌های Fused-MBConv گونه‌ای از لایه‌های MBConv هستند که برای کاهش هزینه‌های محاسباتی، مراحل بسط و پیچیدگی عمقی را در یک کانولوشن معمولی ترکیب می‌کنند. این تغییر می‌تواند بر سرعت آموزش شبکه تأثیر بگذارد، زیرا پارامترهای کمتر ممکن است منجر به محاسبات سریع‌تر شود، اما به طور بالقوه می‌تواند ویژگی‌های پیچیده کمتری را نیز ثبت کند.
- استخراج ویژگی: لایه‌های کانولوشن معمولی (Conv3x3) یک فیلتر را در تمام عمق حجم ورودی اعمال می‌کنند و الگوها و ویژگی‌ها را مستقیماً از ورودی خام می‌گیرند. در مقابل، لایه‌های MBConv و انواع آن‌ها از پیچیدگی‌های جداشدنی در عمق استفاده می‌کنند که ابتدا یک پیچیدگی فضایی عمیق و سپس یک پیچش نقطه‌ای را اعمال می‌کنند، که می‌تواند منجر به پویایی استخراج ویژگی‌های مختلف شود. لایه Fused-MBConv این فرآیند را تغییر می

دهد، که می تواند بر انواع ویژگی های آموخته شده و جداسازی بعدی آنها در فضای ویژگی تأثیر بگذارد.

- قدرت نمایش: لایه‌های Conv3x3 معمولاً به دلیل اعمال مستقیم فیلترها در کانال‌های ورودی، قدرت نمایش بیشتری دارند. این مشخصه در ابتدا می تواند منجر به گرفتن اطلاعات بیشتر شود که به طور بالقوه منجر به SI بالاتر می شود. لایه‌های Fused-MBConv، اگرچه کارآمد هستند، ممکن است در ابتدا تعاملات کمتر پیچیده‌تری را ثبت کنند، که می‌تواند دلیل کاهش مشاهده‌شده در SI بین مرحله ۰ و مرحله ۱ باشد.
- ظرفیت یادگیری: ظرفیت یادگیری یک لایه به تعداد پارامترها و طراحی معماری آن بستگی دارد. تغییر از Conv3x3 به Fused-MBConv1 ممکن است شامل کاهش ظرفیت یادگیری باشد، که می‌تواند به طور موقت منجر به جداسازی کمتر موثر ویژگی‌ها شود تا زمانی که شبکه ویژگی‌های قوی‌تری را در لایه‌های بعدی یاد بگیرد.
- وضوح و Receptive Field: stride و اندازه kernel لایه ها بر وضوح نقشه های ویژگی خروجی و Receptive هر نورون تأثیر می گذارد. Conv3x3 با گام ۲ وضوح فضایی را کاهش می دهد اما میدان دریافتی را افزایش می دهد، که اگر ویژگی های اولیه به خوبی با وضوح درشت از هم جدا شوند، می تواند منجر به SI بالاتر شود. هنگام انتقال به لایه های Fused-MBConv با گام ۱، وضوح حفظ می شود و میدان پذیرنده تغییر می کند، که می تواند بر جداسازی ویژگی ها تأثیر بگذارد.

به طور خلاصه، تفاوت‌ها در انواع لایه‌ها منجر به قابلیت‌های یادگیری ویژگی‌های متمایز، کارایی آموزشی و قدرت‌های نمایشی می‌شود که همه اینها بر نحوه یادگیری شبکه برای جدا کردن ویژگی‌ها در کلاس‌ها تأثیر می‌گذارد. درک این اثرات هنگام طراحی معماری های متناسب با مجموعه داده های خاص و محدودیت های محاسباتی بسیار مهم است.