به نام خدا

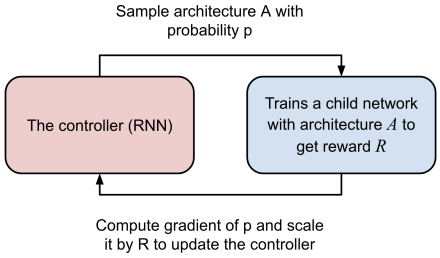
درس یادگیری عمیق

تمرین هفتم

غزل زمانی نژاد

401722244

1. الف) <u>یادگیری تقویتی:</u> اجزای اصلی آن یک کنترلر و یک trainer هستند. در مقاله NAS مطابق شکل زیر، از یک شبکه بازگشتی به عنوان کنترلر استفاده شده که از فضای اکشن های موجود، خروجی (با طول های متغیر) به عنوان ابرپارامترهای شبکه تولید می کند. نام آن را اکشن $a_{1:T}$ می گذاریم (T تعداد کل توکن هاست). سپس trainer با ابرپارامترهایی که کنترلر به عنوان خروجی تولید کرده، آموزش می بیند و پاداشی دریافت می کند. در اینجا منظور از پاداش، دقت شبکه فرزند که می تواند در همگرایی به دست بیاید است. کنترلر بر اساس پاداش دریافت شده، از فضای جستجو نمونه برداری می کند تا در مرحله بعد بتواند خروجی های بهتری تولید کند. برای این کار از تابع ضرر یادگیری تقویتی استفاده می شود که در آن پاداش مورد انتظار بیشینه می شود. نکته خوب این تابع ضرر آن است که حتی اگر پاداش مشتق پذیر نباشد می توان از آن استفاده کرد.



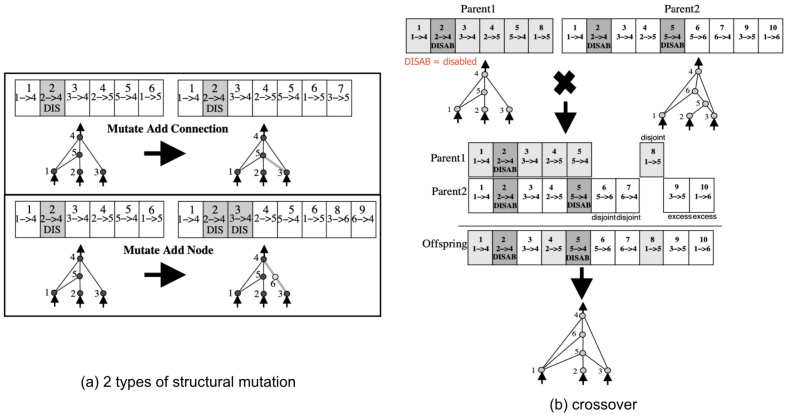
$$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \mathbb{E}[\nabla_\theta \log P(a_t | a_{1:(t-1)}; \theta) R]$$

در مقاله MetaQNN، یک عامل را آموزش می دهد تا به طور متوالی لایه های CNN را با استفاده از Q-learning و با روش ε-greedy (اکتشاف و بهره برداری طی چند مرحله) انتخاب کند.

$$Q^{(t+1)}(s_t, a_t) = (1 - \alpha) Q^{(t)}(s_t, a_t) + \alpha (R_t + \gamma \max_{a \in \mathcal{A}} Q^{(t)}(s_{t+1}, a'))$$

در اینجا s_t یک تاپل از عملیات لایه و پارامترهای مرتبط است. اکشن a ارتباط بین عملیات ها را تعیین می کند. Q-value متناسب با میزان اطمینان ما در دو عملیات متصل است که منجر به دقت بالا می شود.

یادگیری تکاملی: در مقاله NEAT از الگوریتم ژنتیک برای پیدا کردن ابرپارامترها استفاده شد. در این روش وزن اتصال و توپولوژی شبکه با هم تغییر می کنند. هر ژن اطلاعات کامل تنظیمات یک شبکه، از جمله وزن گره ها و لبه ها، را رمزگذاری می کند. رشد جمعیت با اعمال جهش وزن و اتصالات و همچنین crossover بین دو ژن والد انجام می شود. در شکل زیر به خوبی جهش ها دیده می شود:



(a) 2 types of structural mutation

(b) crossover

در مقاله AmoebaNet از روش انتخاب tournament استفاده شده است که در هر تکرار، بهترین کاندید را از میان مجموعه‌ای تصادفی از نمونه‌ها انتخاب می‌کند و فرزندان جهش‌یافته را دوباره در جمعیت قرار می‌دهد. AmoebaNet روش tournament را به گونه ای تغییر داد تا به نفع ژن های جوان‌تر باشد و در هر چرخه، همیشه قدیمی‌ترین مدل‌ها را کنار بگذارد. به این رویکرد aging evolution گفته می شود و موجب می شود فضای جستجوی بیشتری مورد پوشش و اکتشاف قرار گیرد، نه اینکه مدل های با عملکرد خوب را خیلی زود انتخاب کند. مراحل کلی این الگوریتم به شرح زیر است:

- تعدادی مدل از جمعیت نمونه برداری می شود و مدل با بالاترین دقت به عنوان والد انتخاب می شود.
- مدل فرزند از طریق جهش والدین تولید می شود.
- سپس مدل فرزند آموزش و ارزیابی می شود و دوباره به جمعیت اضافه می شود.
- قدیمی ترین مدل از جمعیت حذف می شود.

در این الگوریتم دو نوع جهش نیز وجود دارد: جهش hidden state و جهش عملیات.

یادگیری مبتنی بر گرادیان: جستجو و ارزیابی به صورت مستقل برای تعداد زیادی مدل هزینه بر است. در روش های مبتنی بر گرادیان که به صورت one-shot انجام می شوند، معمولاً یادگیری ابرپارامترها و وزن های شبکه را با هم در یک مدل ترکیب می کنند. برای استفاده از این روش در ابتدا لازم است فرآیند انتخاب عملیات های گسسته، مشتق پذیر باشد. در این روش، همه معماری‌های فرزند را به عنوان زیرگراف‌های مختلف یک ابرگراف با وزن‌های مشترک بین یال‌های مشترک در ابرگراف در نظر می‌گیرند. در مقاله one-shot model، یک شبکه بزرگ با تعداد پارامترهای بسیار زیاد ایجاد می شود به گونه ای که هر عملیات ممکن در فضای جستجو را در بر داشته باشد. پس از آموزش چنین مدل عظیمی، می توان از آن برای ارزیابی هر مدل فرزند نمونه برداری شده از ابرگراف استفاده کرد. برای این کار، بعضی از عملیات ها حذف می شوند و بدین ترتیب مدل های فرزند ایجاد و ارزیابی می شوند. این نوع نمونه برداری می تواند جایگزین روش هایی مثل RL و یادگیری تکاملی شود.

در مقاله DARTS، حذف مداوم بعضی یال ها در هر مسیر در ابرگراف جستجو معرفی می شود. به بررسی بیشتر این الگوریتم می پردازیم:

هر سلول یک گراف جهت دار بدون دور (DAG) است که شامل N نود با topological order است. هر نود یک نمایش x_i دارد که باید آموخته شود. هر یال (i,j) یک عملیات $o^{(i,j)}$ است که x_j را به x_i تبدیل می کند.

$$x_i = \sum_{j<i} o^{(i,j)}(x_j)$$

به جهت اینکه فضای جستجو گسسته باشد، این الگوریتم انتخاب های عملیات های مختلف را به عنوان یک softmax بر روی عملیات ها در نظر می گیرد و بدین ترتیب مسئله به آموختن مجموعه ای از احتمال ها کاهش می یابد.

$$\overline{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_{ij}^{o})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{ij}^{o'})} o(x)$$

در مرحله بعد هدف بهینه کردن وزن ها و ابرپارامترها به صورت همزمان است:

$$\min_{\alpha} \mathcal{L}_{\text{validate}}(w^*(\alpha), \alpha)$$
$$\text{s.t.} w^*(\alpha) = \arg\min_{w} \mathcal{L}_{\text{train}}(w, \alpha)$$

در قدم k، با داشتن ابرپارامترها α_{k-1}، ابتدا وزن ها w_k را بهینه سازی می کنیم. برای این کار w_{k-1} را در جهتی تغییر می دهیم که منجر به کاهش ضرر روی داده آموزشی شود. سپس در حالیکه وزن ها را ثابت نگه داشته ایم، احتمال ها را آپدیت می کنیم به طوری که ضرر ارزیابی کم شود.

$$J_\alpha = \mathcal{L}_{\text{val}}(w_k - \xi \nabla_w \mathcal{L}_{\text{train}}(w_k, \alpha_{k-1}), \alpha_{k-1})$$

ب) اندازه ورودی:

- یادگیری تقویتی: قابل انجام است. با یک اندازه مشخص شروع می کنیم و reward را بر حسب نیاز (مثلا دقت مدل در تشخیص اشیا) تعریف می کنیم. سپس بر اساس پاداش کنترلر را آموزش می دهیم و کنترلر در تکرار بعدی، بر اساس فضای اکشن اندازه ورودی بهتری را تولید می کند.
- یادگیری تکاملی: قابل انجام است. اندازه ورودی را به عنوان ژن در نظر می گیریم. در این الگوریتم طی گام های متفاوت از جمعیت موجود، فرزند (اندازه های ورودی متفاوت) تولید می کنیم و با تابع fitness عملکرد آن را می سنجیم و بدین ترتیب پس از تعدادی تکرار اندازه مناسب بدست می آید.
- یادگیری مبتنی بر گرادیان: این روش نیازمند طراحی دقیق مکانیزم تغییر اندازه ورودی به گونه ای که مشتق پذیر باشد و گنجاندن آن در فرآیند آموزش است. در این صورت می توان از آن استفاده کرد.

تعداد لایه های کانولوشنی:

- یادگیری تقویتی: قابل انجام است. مشابه قسمت قبل، فضای حالت (تعداد لایه ها) و فضای اکشن ها را تعریف می کنیم و در هر گام کنترلر بر اساس پاداش آموزش می بیند تا بتواند در چندین تکرار تعداد لایه ها را نزدیک به حالت بهینه کند.
- یادگیری تکاملی: قابل انجام است. مثل قسمت قبل، تعداد لایه های کانولوشنی را به عنوان ژن در نظر می گیریم و جمعیت را بر این اساس تولید می کنیم. با ارزیابی نمونه های موجود در جمعیت، به حالت بهینه تعداد لایه ها، نزدیک می شویم.
- یادگیری مبتنی بر گرادیان: در این قسمت نیز باید مکانیزم را به گونه ای تغییر دهیم که مشتق پذیر باشد. بهینه سازی تعداد لایه ها در مقاله DARTS مورد بررسی قرار گرفته است.

2. استفاده از early stopping: بجای آموزش تا رسیدن به همگرایی، زمانی که در آموزش مدل کاندید، معیار مورد نظر پس از چندین تکرار بهبود نیافت می توانیم فرایند آموزش را متوقف کنیم و بدین صورت ارزیابی سریعتر انجام می شود.

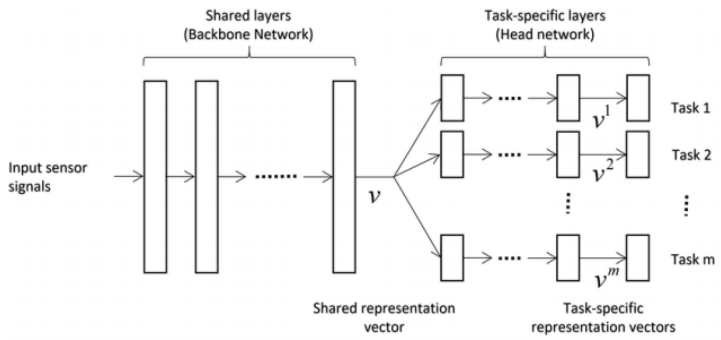
استفاده از warmup برای نرخ یادگیری: در ابتدای آموزش مدل کاندید، نرخ یادگیری را زیاد کنیم تا فضای جستجو بیشتری را در برگیرد و سرعت همگرایی آن بیشتر شود و کم کم بر اساس الگویی (مثلا نمایی) آن را کاهش دهیم. بدین صورت مدل سریعتر همگرا خواهد شد.

به اشتراک گذاری پارامترها: برخی یا همه پارامترها در چندین مدل کاندید مشترک باشند و آموزش آنها به طور همزمان یا متوالی انجام شود. با استفاده مجدد از پارامترها، هزینه محاسباتی را می توان به میزان قابل توجهی کاهش داد و امکان ارزیابی سریعتر فراهم می شود.

مدل surrogate: مدل‌هایی را برای پیش بینی عملکرد تقریبی معماری‌های کاندید حتی بدون آموزش کامل، آموزش دهیم. این مدل ها یاد می گیرند که معیار عملکرد را بر اساس ویژگی های معماری پیش بینی کنند. با استفاده از این مدل‌ها، فضای جستجو را می‌توان به سرعت جستجو کرد و تنها کاندیدها با عملکرد بهتر را برای ارزیابی بیشتر انتخاب کرد و هزینه محاسباتی را کاهش داد.

3. الف) می توانیم با روش های داده افزایی تعداد داده های اقلیت را بیشتر کنیم. مثلا از back translation (ترجمه داده ها به زبان موقت دیگر و ترجمه از زبان موقت به زبان اصلی) استفاده کنیم. همچنین می توانیم به کلاس ها وزن های متفاوت اختصاص دهیم و به کلاس اقلیت اهمیت بیشتری دهیم. علاوه بر این فرمول تابع ضرر را به گونه ای تغییر دهیم که اگر مدل در پیش بینی کلاس های اقلیت اشتباه کرد، بیشتر جریمه شود.

ب) برای اینکه بازنمایی های مشترک آموخته شود، نیاز است تا backbone خوبی داشته باشیم. این قسمت که بین هر دو تسک مشترک است بازنمایی ها را می آموزد و سپس لایه های task-specific وزن ها را به گونه ای تنظیم می کنند که برای هر تسک مناسب باشد. سپس باید توابع ضرر مناسب هم برای تحلیل احساسات و هم برای طبقه بندی موضوع تعریف کنیم (مثلا cross-entropy برای هر تسک). برای اینکه اطمینان حاصل کنیم هر دو بخش به خوبی آموزش می بینند، می‌توانیم ترکیبی از ضررهای ناشی از هر دو تسک را تعریف کنیم که لایه های مشترک را تشویق می کند تا بازنمایی های مفید برای هر دو کار را بیاموزند و بدین شکل بین عملکرد خاص هر تسک و استفاده از اطلاعات مشترک تعادل برقرار کند.



علاوه بر این، برای اینکه در طول آموزش مدل نسبت به یکی از تسک ها دچار بایاس نشود، باید از batchهایی که ترکیبی از داده های هر دو تسک است استفاده شود. می‌توانیم از batchها به‌گونه‌ای نمونه برداری کنیم که بازنمایی یکسانی از هر دو وظیفه را تضمین کند یا از تکنیک‌هایی مانند نمونه برداری dynamic بر اساس عملکرد مدل فعلی استفاده کنیم.

در آخر در صورت استفاده از روش های regularization، می توانیم از overfit شدن مدل بر روی یکی از تسک ها پیشگیری کنیم و تعمیم دهی مدل را بهبود بخشیم.

ج) استفاده از یادگیری انتقالی در قسمت backbone می تواند در بهبود عملکرد مدل کمک کند. باید مدل پیش آموخته ای را انتخاب کنیم که با نیازهای ما و همچنین اندازه و پیچیدگی دیتاست مطابقت داشته باشد. Shared backbone را با وزن های مدل پیش آموخته و لایه های task-specific را به صورت تصادفی مقداردهی اولیه می کنیم. سپس باید مدل را fine-tune کنیم. تنظیم دقیق شامل آپدیت پارامترهای لایه‌های مشترک و همچنین لایه‌های مخصوص هر تسک است. در طول تنظیم دقیق، مدل یاد می‌گیرد که در عین حال ویژگی های عمومی را -که توسط مدل پیش آموخته یاد گرفته- حفظ کند و ویژگی‌های خاص تسک را استخراج کند. بهتر است از نرخ یادگیری کوچک استفاده کنیم تا اطلاعات مدل پیش آموخته به سرعت از بین نرود. همچنین می توانیم روش های regularization را به کار ببریم تا مدل overfit نشود. تنظیم درست ابرپارامترها (نرخ dropout، پارامتر λ در L2، اندازه batch و ...) می تواند در بهبود عملکرد مدل نقش بسزایی داشته باشد.

د) یک معیار ارزیابی برای هرکدام از تسک ها تعریف می کنیم:

برای تسک تحلیل احساسات، از آنجایی که دیتاست متعادل نیست و تعداد زیادی داده با برچسب مثبت دارد، accuracy نمی تواند معیار خوبی باشد. بهتر است از precision و F1-score استفاده کنیم. همچنین با استفاده از confusion matrix می توانیم تعداد پیش بینی های درست و نادرست مدل را برای هر کلاس به دقت بررسی کنیم. منحنی AUC-ROC نیز می تواند عملکرد مدل را در تمایز بین احساسات مثبت و منفی در آستانه های طبقه بندی مختلف ارزیابی کند. و در آخر معیار specificity -که نسبت احساسات منفی پیش‌بینی‌شده درست را از همه احساسات منفی واقعی اندازه‌گیری می‌کند- به ویژه در هنگام برخورد با دیتاست نامتعادل که به سمت احساسات مثبت بایاس شده، مفید است.

برای تسک طبقه بندی موضوع، می توانیم از معیارهای مختلفی از جمله accuracy، precision، recall، F1-score بهره ببریم. همچنین در سناریوهای طبقه‌بندی موضوع چند کلاسه، macro-average F1 امتیاز F1 را برای هر موضوع جداگانه محاسبه می‌کند و سپس میانگین را می‌گیرد. micro-average F1، امتیاز F1 را با در نظر گرفتن تعداد کل موارد مثبت واقعی، مثبت کاذب و منفی کاذب در همه موضوعات محاسبه می کند. این دو معیار هم می توانند مناسب باشند.

4. الف) معماری این بخش، یک MLP دارای ۸ لایه است. در روش های دیگر GAN، ورودی latent از طریق یک لایه ورودی به مدل داده می شد اما در این روش یک شبکه مپ غیرخطی بر روی ورودی اعمال می شود و intermediate latent به نام W را تولید می کند. با اعمال این شبکه بر روی ورودی، یک بازنمایی معنادار از ورودی تولید می شود که به تفکیک ویژگی های تصویر تولید شده کمک می کند. همچنین کنترل بر روی style تصاویر تولیدی بیشتر می شود.

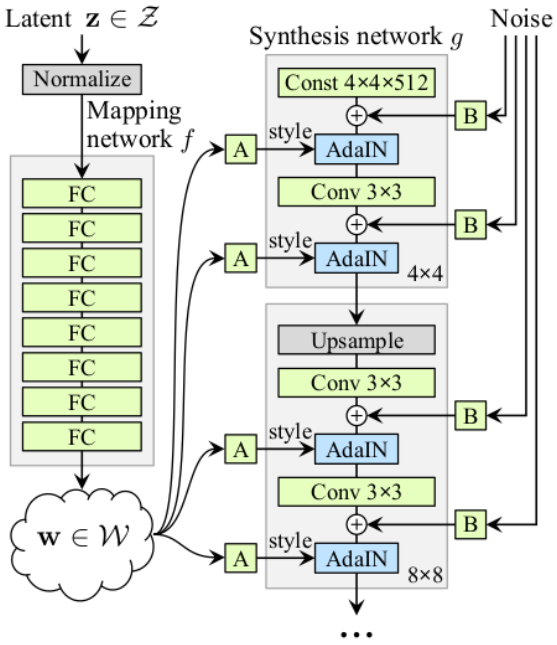
ب) در روش سنتی از یک بردار رندوم (معمولا با توزیع گاوسی) به عنوان latent استفاده می شود و مستقیما به عنوان ورودی مولد داده می شود که کنترلی بر روی ویژگی های تصویر تولید شده ندارد اما در style-GAN بردارد latent از یک شبکه ۸ لایه عبور می کند و W تولید می شود. سپس AdaIN بر روی بردار W حاصل به جهت تعدیل سبک هر لایه در شبکه مولد اعمال می شود. با تعدیل سبک های لایه های مختلف، StyleGAN کنترل دقیق تری بر فرآیند تولید به دست می آورد.

ج) Adaptive Instance Normalization برای تعدیل سبک هر لایه در شبکه مولد استفاده می شود. AdaIN از ایده نرمال سازی هر نمونه همراه با scale و شیفت به صورت adaptive استفاده می کند. به جای استفاده از پارامترهای نرمال سازی ثابت، AdaIN این پارامترها را به صورت پویا بر اساس اطلاعات استایل انکود شده در فضای latent میانی محاسبه می کند. با گرفتن نقشه ویژگی میانی و یک بردار استایل از فضای W، مراحل زیر را طی می کند:

1. میانگین و انحراف معیار نقشه ویژگی را در ابعاد spatial محاسبه می کند.
2. با کم کردن میانگین و تقسیم بر انحراف معیار، نقشه ویژگی را نرمالایز می کند.
3. نقشه ویژگی نرمال شده را در انحراف معیار استایل ضرب می کند و میانگین استایل را اضافه می کند.
4. مقیاس و بایاس قابل یادگیری را بر روی نقشه ویژگی بدست آمده اعمال می کند.

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i},$$

د) در تصویر زیر، A نشان دهنده affine transform های آموخته شده است. کار این ماژول تولید استایل از روی W (خروجی شبکه مپینگ) است. affine transform به عنوان یکی از ورودی های AdaIN، برای تعدیل سبک هر لایه در شبکه مولد به کار می رود.

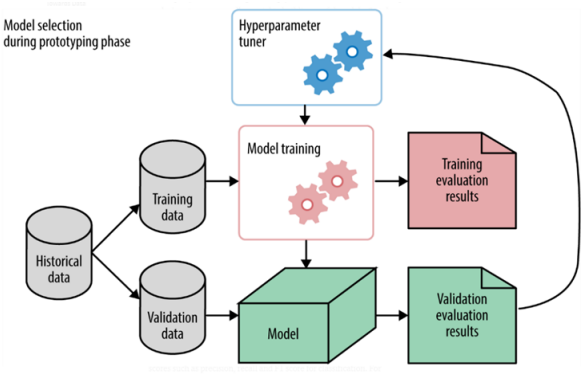


ه) style mixing شامل ترکیب اطلاعات استایل از لایه های مختلف شبکه مولد برای ایجاد تغییرات جدید در تصاویر تولید شده است. در طول فرآیند تولید، به جای استفاده از یک بردار latent ثابت به عنوان ورودی، از چندین بردار latent نمونه برداری می شود. هر لایه می تواند بردار latent خود را انتخاب کند یا استایل لایه قبلی را به ارث برد. بدین ترتیب، استایل های لایه های مختلف می توانند باهم ترکیب شوند. برای این کار به طور تصادفی یک نقطه crossover در معماری شبکه، معمولاً در لایه‌های میانی، انتخاب می شود. بردارهای latent از این نقطه جدا می شوند و استایل ها با ترکیب بردارهای latent از یک طرف با استایل های طرف دیگر ترکیب می شوند. این کار منجر به تولید تصاویر متنوع با ترکیبی از سبک‌ها از لایه‌های مختلف می‌شود که امکان ایجاد طیف وسیع‌تری از تغییرات و افزایش کیفیت تصاویر تولیدی را فراهم می‌کند.

. .5

```
fake_images_2 = genAB(realA)

def adversarial_loss(discriminator, generated_images):
   # discriminator's output for the generated images
   discriminator_output = discriminator(generated_images)
   return -log(discriminator_output)

def cycle_consistency_loss(generator1, generator2, real_images):
   # Generate images from the real images using both generators
   reconstructed_images = generator2(generator1(real_images))
   cycle_loss = np.mean(math.abs(real_images - reconstructed_images))
   return cycle_loss

adv_loss_1 = adversarial_loss(discB, fake_images_2)
cycle_loss_1 = cycle_consistency_loss(genAB, genBA, realAB)

# Total generator loss
genAB_loss = adv_loss_1 + lambda_cycle * cycle_loss_1
```

6. الف) تیونر ابرپارامتر جدا از مدل است و تنظیم قبل از آموزش مدل انجام می شود. نتیجه فرآیند تنظیم مقادیر بهینه ابرپارامترها است که سپس به مرحله آموزش مدل داده می شود. Optuna یک فریم ورک برای اتوماتیک کردن فرآیند بهینه سازی ابرپارامترهاست. این فریم ورک به طور خودکار مقادیر ابرپارامتر بهینه را با استفاده از روش های مختلف مانند grid search، تصادفی، Bayesian و الگوریتم‌های تکاملی پیدا می‌کند.



توضیح مختصر هر یک از sampler ها:

Grid Search: فضای جستجوی هر ابرپارامتر گسسته شده است. بهینه ساز، یادگیری را بر روی هر یک از تنظیمات ابرپارامترها انجام می دهد و در پایان بهترین ها را انتخاب می کند.

تصادفی: به صورت تصادفی از فضای جستجو نمونه برداری می کند و این کار تا زمان دستیابی به معیارهای توقف ادامه می یابد.

Bayesian: رویکرد مبتنی بر مدل احتمالی برای یافتن ابرپارامترهای بهینه

الگوریتم های تکاملی: از مقدار تابع fitness برای یافتن ابرپارامترهای بهینه استفاده می کنند.

دلایل استفاده از این فریم ورک:

- فضاهای جستجوی dynamic
- الگوریتم های نمونه برداری و pruning کارآمد
- integration آسان
- تصویرسازی خوب
- بهینه سازی توزیع شده

Optuna وابسته به فریم ورک نیست و می تواند با اکثر فریم ورک های پایتون، از جمله keras Pytorch، Scikit-learn و غیره استفاده شود. این فریم ورک عمدتاً برای یادگیری ماشین طراحی شده است و تا زمانی که بتوانیم تابع هدف را تعریف کنیم، می توان از آن در کارهای غیر ML استفاده کرد.

ب) در قسمت prepare data، دیتاست CIFAR100 را دانلود می کنیم. سپس مدل را تعریف می کنیم. برای این کار یک ModuleList می سازیم و بر اساس تعداد لایه های کانولوشنی، داخل یک حلقه لایه conv2d می سازیم و به لیست اضافه می کنیم. تعداد کرنل هر لایه نیز از ابرپارامترهاست اما سایز هر کرنل ۳*۳ است. با اضافه کردن هر لایه کانولوشنی، اندازه خروجی را هم محاسبه می کنیم زیرا برای لایه fully connected به آن نیاز داریم. سپس در یک حلقه به تعداد num_fc_layers لایه Linear می سازیم و به لیست مربوطه اضافه می کنیم.

در تابع forward، ابتدا ورودی را از تمامی لایه های کانولوشنی عبور می دهیم، آن را flat می کنیم و بعد از لایه های خطی عبور می دهیم.

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

    def __init__(self, trial, num_conv_layers, num_fc_layers, num_filters, num_neurons):

        super(Net, self).__init__()
        input_size = 32
        kernel_size = 3

        # define the convolutional layers
        self.convs = nn.ModuleList([nn.Conv2d(3, num_filters[0], kernel_size=kernel_size)])
        # output size of cnn is needed for fc
        out_size = input_size - kernel_size + 1

        for i in range(1, num_conv_layers):
            self.convs.append(nn.Conv2d(in_channels=num_filters[i-1], out_channels=num_filters[i], kernel_size=kernel_size))
            out_size = out_size - kernel_size + 1

        # define fully connected layers
        self.fcs = nn.ModuleList([])
        self.out_feature = num_filters[-1] * out_size * out_size
        num_neurons = [self.out_feature] + num_neurons

        for i in range(1, num_fc_layers):
            self.fcs.append(nn.Linear(num_neurons[i-1], num_neurons[i]))

        self.fcs.append(nn.Linear(num_neurons[-1], classes))


    def forward(self, x):

        for i, conv_i in enumerate(self.convs):
            x = F.relu(conv_i(x))

        x = x.view(-1, self.out_feature)
        for fc in self.fcs:
            x = F.relu(fc(x))

        return x
```

برای بهینه کردن ابرپارامترها با کمک optuna، باید یک تابع objective تعریف کنیم و ورودی آن را trial قرار دهیم. تعداد لایه های کانولوشنی و خطی یک عدد صحیح است پس از suggest_int استفاده می

کنیم. برای تعداد فیلترها / نورون ها از suggest_float استفاده می کنیم و به تعداد لایه های کانولوشنی / خطی این مقادیر را تولید می کنیم. سپس مدل را با ابرپارامترهای پیشنهادی تولید می کنیم. بهینه ساز و همچنین نرخ یادگیری نیز تولید می کنیم. برای مسئله چندکلاسه، از تابع ضرر کراس آنتروپی استفاده می کنیم.

سپس در یک حلقه به تعداد ایپاک مدل را آموزش می دهیم و بر روی داده تست عملکرد آن را بررسی می کنیم. در اینجا معیار دقت تست است.به عنوان خروجی تابع objective، دقت تست را برمیگردانیم.

```python
def objective(trial):
    """
    Hyperparameters:
    number of convolutional layers --> MAX: 4
    number of dense layers --> MAX: 3
    number of filters of convolutional layers --> MAX: 64
    number of neurons of fully connected layers --> MAX: 64
    learning rate --> MAX: 0.01
    optimizer
    """

    # Define range of values
    num_conv_layers = trial.suggest_int("num_conv_layers", 1, 4)
    num_fc_layers = trial.suggest_int("num_fc_layers", 1, 3)
    num_filters = [int(trial.suggest_float("num_filter_"+str(i), 40, 64, step=8))
                   for i in range(num_conv_layers)]              # 40, 48, 56, 64

    # number of out features of last fc layer is fix:100
    num_neurons = [int(trial.suggest_float("num_neurons_"+str(i), 32, 64, step=16))
                   for i in range(num_fc_layers-1)]

    # Generate the model
    model = Net(trial, num_conv_layers, num_fc_layers, num_filters, num_neurons).to(device)
    print(model)

    # Generate the optimizers
    optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "RMSprop", "SGD"])
    lr = trial.suggest_float("lr", 1e-3, 1e-2, log=True)
    optimizer = getattr(optim, optimizer_name)(model.parameters(), lr=lr)
    # loss function
    criterion = nn.CrossEntropyLoss()
```

```
    # Training loop
    for epoch in range(epochs):

        model.train()
        for batch_i, (x, y) in enumerate(train_loader):
            optimizer.zero_grad()
            output = model(x.to(device))
            loss = criterion(output, y.to(device))
            loss.backward()
            optimizer.step()

        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for batch_i, (x, y) in enumerate(test_loader):
                x = x.to(device)
                y = y.to(device)

                output = model(x)
                _, predicted = torch.max(output.data, 1)
                total += y.size(0)
                correct += (predicted == y).sum().item()

        accuracy_test = correct / total

    return accuracy_test
```

در قسمت بعد یک شی study می سازیم و به مدت ۳۰ دقیقه ابرپارامترها را بهینه می کنیم. در آخر مقادیر بدست آمده در بهترین trial را چاپ می کنیم.

```
# Create an Optuna study to maximize test accuracy
study = optuna.create_study(direction="maximize")
study.optimize(objective, timeout=1800) #30-minute timout

trial = study.best_trial
print("Best trial:")
print("  Test accuracy: ", trial.value)
print("  Params: ")
for key, value in trial.params.items():
    print(f"{key}: {value}")
```

در این مدت ۱۰ trial اجرا شدند که بهترین مقدار دقت بدست آمده 0.2362 است. بهترین مدل دارای ۴ لایه کانولوشنی و ۱ لایه خطی است و با SGD و نرخ یادگیری 0.008 بهینه شده است.

```
Best trial:
  Test accuracy:  0.2362
  Params:
num_conv_layers: 4
num_fc_layers: 1
num_filter_0: 40.0
num_filter_1: 56.0
num_filter_2: 40.0
num_filter_3: 48.0
optimizer: SGD
lr: 0.008092387757307865
```

ج) pruning فرآیند حذف اتصالات در یک شبکه برای افزایش سرعت و کاهش اندازه مدل است. به طور کلی، تعداد پارامترهای شبکه های عصبی بسیار زیاد است. هرس یک شبکه را می توان به عنوان حذف پارامترهای استفاده نشده تعریف کرد. می توان گفت pruning به عنوان جستجوی معماری شبکه عمل می کند. دو نوع pruning داریم: structured و unstructured. در structured دسته ای از اتصالات وزن ها همگی با هم حذف می شوند (مثلا تمامی اتصالات یک نود) در حالیکه در unstructured بعضی از اتصالات وزن ها به صورت فردی حذف می شوند.

برای استفاده از هرس کردن شبکه، کافیست در training loop تغییری اعمال کنیم که هرجا trial نیاز به هرس کردن داشت، آموزش آن شبکه متوقف شود.

```python
    # Training loop
    for epoch in range(epochs):

        model.train()
        for batch_i, (x, y) in enumerate(train_loader):
            optimizer.zero_grad()
            output = model(x.to(device))
            loss = criterion(output, y.to(device))
            loss.backward()
            optimizer.step()

        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for batch_i, (x, y) in enumerate(test_loader):
                x = x.to(device)
                y = y.to(device)

                output = model(x)
                _, predicted = torch.max(output.data, 1)
                total += y.size(0)
                correct += (predicted == y).sum().item()

        accuracy_test = correct / total

        # pruning
        trial.report(accuracy_test, epoch)
        if trial.should_prune():
            raise optuna.exceptions.TrialPruned()

    return accuracy_test
```

نتیجه اجرا به شرح زیر است:

```
[I 2023-07-02 08:13:03,611] Trial 0 finished with value: 0.01 and parameters: {'
[I 2023-07-02 08:16:25,564] Trial 1 finished with value: 0.1211 and parameters:
[I 2023-07-02 08:19:33,456] Trial 2 finished with value: 0.2171 and parameters:
[I 2023-07-02 08:22:54,138] Trial 3 finished with value: 0.01 and parameters: {'
[I 2023-07-02 08:26:14,610] Trial 4 finished with value: 0.01 and parameters: {'
[I 2023-07-02 08:29:24,780] Trial 5 finished with value: 0.1779 and parameters:
[I 2023-07-02 08:29:44,550] Trial 6 pruned.
[I 2023-07-02 08:30:04,613] Trial 7 pruned.
[I 2023-07-02 08:30:23,341] Trial 8 pruned.
[I 2023-07-02 08:33:26,899] Trial 9 finished with value: 0.2167 and parameters:
[I 2023-07-02 08:33:47,151] Trial 10 pruned.
[I 2023-07-02 08:36:49,637] Trial 11 finished with value: 0.239 and parameters:
[I 2023-07-02 08:39:52,414] Trial 12 finished with value: 0.2342 and parameters:
Best trial:
  Test accuracy:  0.239
  Params:
num_conv_layers: 1
num_fc_layers: 1
num_filter_0: 56.0
optimizer: SGD
lr: 0.004400992296891414
```

با اعمال هرس، بهترین نتیجه بدست آمده 0.239 است که اندکی از حالت قبل بهتر است. به طور کلی pruning به افزایش سرعت آموزش و همچنین کاهش پیچیدگی مدل کمک می کند. در این مثال با کمک هرس کردن توانستیم در مدت زمان مشابه، تعداد trial 13 را بررسی کنیم.

البته باید توجه داشت که استفاده از pruning نمی تواند در همه موارد منجر به بهبود عملکرد شود. مثلا اگر مدل از ابتدا کوچک است، هرس کردن برخی اتصالات ممکن است باعث کاهش ظرفیت یادگیری و underfit شدن شبکه شود.

7. کلاس Vocabulary را برای ذخیره کردن کلمات منحصر بفرد موجود در داده آموزشی تعریف می کنیم.

```python
# Define the Word2Vec model class

class Word2Vec(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(Word2Vec, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.in_embed = nn.Embedding(vocab_size, embedding_dim)
        self.out_embed = nn.Embedding(vocab_size, embedding_dim)

    def forward(self, target_word, context_word):
        target_embed = self.in_embed(target_word)
        context_embed = self.out_embed(context_word)
        return target_embed, context_embed
```

```python
class Vocabulary:
    def __init__(self, words):
        self.vocab = list(set(words))

        self.stoi = {v:k for k, v in enumerate(self.vocab)}
        self.itos = {k:v for k, v in enumerate(self.vocab)}

    def __len__(self):
        return len(self.stoi)
```

در تابع train_word2vec، ابتدا دیتاست را با word_tokenize موجود در کتابخانه nltk توکنایز می کنیم و یک نمونه از کلاس Vocabulary می سازیم. سپس در یک حلقه باید جفت target, context ها را تشکیل دهیم و آنها را به یک لیست اضافه کنیم. سپس مدل را می سازیم. از تابع ضرر کراس آنتروپی و بهینه ساز AdamW برای آموزش مدل استفاده می کنیم.

```python
def train_word2vec(corpus, window_size, embedding_dim, num_epochs, learning_rate):
    # Preprocess the corpus and build the vocabulary
    tokens = word_tokenize(corpus)
    v = Vocabulary(tokens)

    training_pairs = []
    # Create the target-context word pairs
    for t in range(len(tokens)):
        if tokens[t] == '.' or tokens[t] == '!':
            continue

        for c in range(t-window_size//2, t+1+window_size//2):
            if c == t or c < 0 or c >= len(tokens) or tokens[c] == '.' or tokens[c] == '!':
                continue

            target = tokens[t]
            context = tokens[c]
            training_pairs.append((torch.tensor(v.stoi[target]), torch.tensor(v.stoi[context])))

    # Initialize the Word2Vec model
    model = Word2Vec(len(v), embedding_dim)

    # Define the loss function and optimizer
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters(), lr=learning_rate)
```

در یک حلقه به تعداد ایپاک مدل را آموزش می دهیم. هدف از آموزش این است که بردار جانمایی کلمه target و context به یکدیگر نزدیک شوند. (البته در word2vec مفهوم negative sampling نیز داریم. بدین معنی که بردار جانمایی کلماتی که به هم مربوط نیستند از هم دور شوند. اما در اینجا خواسته نشده)

پیش بینی مدل را بدست می آوریم و میزان ضرر را حساب می کنیم و با backpropagation پارامترها را آپدیت می کنیم.

```
    for epoch in range(num_epochs):
        total_loss = 0.0
        for target_word, context_word in training_pairs:
            # Zero the gradients
            optimizer.zero_grad()
            # Forward pass
            t, c = model(target_word, context_word)
            # Compute the loss
            loss = loss_fn(t, c)
            # Backward pass
            loss.backward()
            # Update the model parameters
            optimizer.step()
            # Accumulate the loss
            total_loss += loss.item()

        # Print the average loss for the epoch
        print(f"Epoch {epoch+1} Loss: {total_loss/len(training_pairs):.3f}")

    # Return the trained Word2Vec model and vocab
    return (v, model)
```

تابع k_most_similar برای پیدا کردن k تا امبدینگ مشابه به کلمه goal پیاده سازی شده است. روی تمامی کلمات موجود در vocab لوپ می زنیم و مقدار cosine similarity آن دو را محاسبه می کنیم. این مقادیر را در لیست scores ذخیره می کنیم. در آخر لیست را سورت می کنیم و k المان اول را برمی گردانیم.

```
def k_most_similar(goal, embeddings, stoi, k):

    scores = []
    goal_embed = embeddings[stoi[goal]]
    for (w, idx) in stoi.items():
        if w == goal:
            continue

        s = np.dot(goal_embed, embeddings[idx]) / (np.linalg.norm(goal_embed, 2) * np.linalg.norm(embeddings[idx], 2))
        scores.append((w, s))

    sort_scores = sorted(scores, key=lambda i: i[1], reverse=True)
    return sort_scores[:k]
```

در تابع main، مدل را برای ۵۰ ایپاک آموزش می دهیم و وزن های امبدینگی که مدل آموخته را در یک ماتریس ذخیره می کنیم. برای ارزیابی مدل، ۳ کلمه با بیشترین شباهت به learn و همچنین deep را چاپ می کنیم. در آخر وزن تمامی امبدینگ هایی که مدل آموخته را چاپ می کنیم.

```python
# Define the main function

def main():
    # Set hyperparameters
    corpus = "I love to learn deep learning. It is fascinating!"
    window_size = 3
    embedding_dim = 10

    LR = 1e-2
    EPOCHS = 50

    # Train the Word2Vec model
    vocab, model = train_word2vec(corpus, window_size, embedding_dim, EPOCHS, LR)
    embeddings = model.in_embed.weight.detach().numpy()

    # Evaluate the trained model using word similarity or analogy tasks
    x = k_most_similar("learn", embeddings, vocab.stoi, 3)
    print("3 most similar words to 'learn':", x)

    x = k_most_similar("deep", embeddings, vocab.stoi, 3)
    print("3 most similar words to 'deep':", x)

    print()
    # Print the learned word embeddings
    for (w, idx) in vocab.stoi.items():
        print(w, embeddings[idx])

    # Save the trained model
    torch.save(model.state_dict(), './word2vec')

# Run the main function
if __name__ == "__main__":
    main()

```

خروجی به شرح زیر است:

```
Epoch 35 Loss: -95.328
Epoch 36 Loss: -100.134
Epoch 37 Loss: -105.089
Epoch 38 Loss: -110.194
Epoch 39 Loss: -115.448
Epoch 40 Loss: -120.851
Epoch 41 Loss: -126.403
Epoch 42 Loss: -132.104
Epoch 43 Loss: -137.955
Epoch 44 Loss: -143.954
Epoch 45 Loss: -150.104
Epoch 46 Loss: -156.403
Epoch 47 Loss: -162.852
Epoch 48 Loss: -169.451
Epoch 49 Loss: -176.200
Epoch 50 Loss: -183.100
3 most similar words to 'learn': [('learning', 0.43414298), ('!', 0.3971264), ('fascinating', 0.3562515)]
3 most similar words to 'deep': [('fascinating', 0.39868173), ('love', 0.34144998), ('to', 0.15409891)]

love [ 1.4775798   0.8041824  -2.614521   -3.7062607  -0.62723225 -4.1373773
 -3.8834667  -2.1617348   1.737296    5.312615  ]
fascinating [-1.7544217 -1.167995  -2.144297  -0.7099107 -0.78177   -3.8719144
  4.384316  -2.8021953  4.912155  -1.6097721]
learning [-2.9400122  -2.2859077   2.8555007   0.25367412  2.9188175  -0.43283987
 -1.824673   -2.2402902   3.4572022   3.9518719 ]
! [-1.1135978  -0.48049936  1.5579455   0.62664634  1.1660824   0.01318888
 -1.6666498   0.65174854  0.8014188  -0.08017857]
It [-2.059316    2.5467362  -3.0649755   3.4732928  -2.2734075   0.79049593
  3.7148614  -3.3957176  -2.6177394  -2.345723  ]
I [ 2.8204997  -3.687538    3.3412094  -2.1550894  -2.4314075   0.6657372
 -0.33957657  2.8705516  -3.573452   -4.524783  ]
to [-2.6473744 -3.762436  -2.5661087 -3.0196512  5.7184396  5.0815096
  4.1962566 -4.4413004 -1.165969  -1.4747025]
. [ 0.6558491   0.3500321  -1.1028589   0.710806    0.40526348  0.47484872
 -0.58851224  1.0177428   2.113282    0.21641515]
learn [-2.010258  -1.9150654  5.976407  -1.8308094 -2.7581506 -2.6137593
 -2.635566  -4.566281   4.4235086 -2.999052 ]
is [-2.3327806  -2.249495   -4.3764625  -1.5924634   6.6428776   2.460241
 -2.4343288   0.14639644 -3.527408   -3.5387223 ]
deep [-1.004316   -3.4670644  -0.37852404 -2.6947887  -3.2636898  -3.622468
```

در طول آموزش مقدار loss مطابق انتظار کاهش یافته است. طبق نتایج بدست آمده با وجود اینکه امبدینگ learn به learning نزدیک است، اما به جز این سایر شباهت ها چندان مطلوب نیست. زیرا مدل بر روی داده بسیار کوچک (تنها ۲ جمله) آموزش دیده و به همین علت نمی توان انتظار داشت معنای بین کلمات را درک کند.

8. در قسمت اول در تابع make_gen_block، باید یک بلوک از مولد را پیاده سازی کنیم. اگر لایه نهایی نباشد از ConvTranspose2d، BatchNorm2d و تابع فعال سازی Relu استفاده می کنیم و در غیر این صورت از Tanh بدون BatchNorm2d.

```python
def make_gen_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
    '''
    Function to return a sequence of operations corresponding to a generator block of DCGAN;
    a transposed convolution, a batchnorm (except in the final layer), and an activation.
    Parameters:
        input_channels: how many channels the input feature representation has
        output_channels: how many channels the output feature representation should have
        kernel_size: the size of each convolutional filter, equivalent to (kernel_size, kernel_size)
        stride: the stride of the convolution
        final_layer: a boolean, true if it is the final layer and false otherwise
                    (affects activation and batchnorm)
    '''
    #     Steps:
    #       1) Do a transposed convolution using the given parameters.
    #       2) Do a batchnorm, except for the last layer.
    #       3) Follow each batchnorm with a ReLU activation.
    #       4) If its the final layer, use a Tanh activation after the deconvolution.

    # Build the neural block
    if not final_layer:
        return nn.Sequential(
            #### START CODE HERE ####
            nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
            nn.BatchNorm2d(output_channels),
            nn.ReLU()
            #### END CODE HERE ####
        )
    else: # Final Layer
        return nn.Sequential(
            #### START CODE HERE ####
            nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
            nn.Tanh()
            #### END CODE HERE ####
        )
```

برای ممیز هم به همان ترتیب خواسته شده بلوک را پیاده سازی می کنیم:

```python
def make_disc_block(self, input_channels, output_channels, kernel_size=4, stride=2, final_layer=False):
    '''
    Function to return a sequence of operations corresponding to a discriminator block of the DCGAN;
    a convolution, a batchnorm (except in the final layer), and an activation (except in the final laye
    Parameters:
        input_channels: how many channels the input feature representation has
        output_channels: how many channels the output feature representation should have
        kernel_size: the size of each convolutional filter, equivalent to (kernel_size, kernel_size)
        stride: the stride of the convolution
        final_layer: a boolean, true if it is the final layer and false otherwise
                    (affects activation and batchnorm)
    '''
    #     Steps:
    #       1) Add a convolutional layer using the given parameters.
    #       2) Do a batchnorm, except for the last layer.
    #       3) Follow each batchnorm with a LeakyReLU activation with slope 0.2.

    # Build the neural block
    if not final_layer:
        return nn.Sequential(
            #### START CODE HERE #### #
            nn.Conv2d(input_channels, output_channels, kernel_size, stride),
            nn.BatchNorm2d(output_channels),
            nn.LeakyReLU(0.2)
            #### END CODE HERE ####
        )
    else: # Final Layer
        return nn.Sequential(
            #### START CODE HERE #### #
            nn.Conv2d(input_channels, output_channels, kernel_size, stride),
            #### END CODE HERE ####
        )
```

با استفاده از تابع one_hot بردار برچسب ها را بر اساس تعداد کلاس ها به one hot تبدیل می کنیم:

```python
import torch.nn.functional as F
def get_one_hot_labels(labels, n_classes):
    '''
    Function for creating one-hot vectors for
    Parameters:
        labels: tensor of labels from the data
        n_classes: the total number of classes
    '''
    #### START CODE HERE ####
    return F.one_hot(labels, n_classes)
    #### END CODE HERE ####
```

در تابع combine_vectors، دو بردار را از روی بعد دوم با هم concatenate می کنیم.

```python
def combine_vectors(x, y):
    '''
    Function for combining two vectors with shapes (n_samples, ?) and (n_samples, ?).
    Parameters:
      x: (n_samples, ?) the first vector.
        In this assignment, this will be the noise vector of shape (n_samples, z_dim)
        but you shouldn't need to know the second dimension's size.
      y: (n_samples, ?) the second vector.
        Once again, in this assignment this will be the one-hot class vector
        with the shape (n_samples, n_classes), but you shouldn't assume this in your
    '''
    # Note: Make sure this function outputs a float no matter what inputs it receives
    #### START CODE HERE ####
    combined = torch.cat((x.float(), y.float()), dim=1)
    #### END CODE HERE ####
    return combined
```

از تابع ضرر BCE استفاده می کنیم:

```python
#### START CODE HERE ####
criterion = nn.BCEWithLogitsLoss()
#### END CODE HERE ####
```

در تابع get_input_dimensions باید اندازه ورودی مولد و تعداد کانال های ورودی ممیز را پیاده سازی کنیم. ورودی مولد حاصل concate بردار نویز و بردار کلاس است. برای ممیز باید برای هر کلاس یک کانال اضافه کنیم.

```python
def get_input_dimensions(z_dim, mnist_shape, n_classes):
    '''
    Function for getting the size of the conditional input dimensions
    from z_dim, the image shape, and number of classes.
    Parameters:
        z_dim: the dimension of the noise vector, a scalar
        mnist_shape: the shape of each MNIST image as (C, W, H), which is (1, 28, 28)
        n_classes: the total number of classes in the dataset, an integer scalar
                (10 for MNIST)
    Returns:
        generator_input_dim: the input dimensionality of the conditional generator,
                             which takes the noise and class vectors
        discriminator_im_chan: the number of input channels to the discriminator
                               (e.g. C x 28 x 28 for MNIST)
    '''
    #### START CODE HERE ####
    generator_input_dim = z_dim + n_classes
    discriminator_im_chan = mnist_shape[0] + n_classes
    #### END CODE HERE ####
    return generator_input_dim, discriminator_im_chan
```

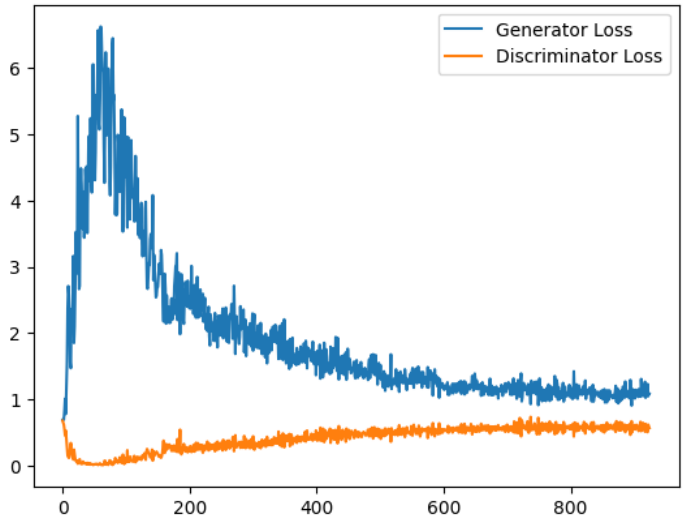
در قسمت آخر، باید در حلقه training تغییراتی اعمال کنیم. ورودی مولد حاصل کنار هم قرار دادن نویز و بردار کلاس است به همین دلیل از combine_vecotrs استفاده می کنیم و خروجی آن را به عنوان ورودی مولد می دهیم.

```python
#### START CODE HERE ####
noise_and_labels = combine_vectors(fake_noise, one_hot_labels)
fake = gen(noise_and_labels)
#### END CODE HERE ####
```

اکنون باید ورودی ممیز را تولید کنیم که شامل ترکیب تصویر غیراصل با بردار image_one_hot_labels و ترکیب تصویر اصل با image_one_hot_labels است. سپس خروجی ممیز را برای این دو بردار محاسبه می کنیم.

```python
#### START CODE HERE ####
fake_image_and_labels = combine_vectors(fake.detach(), image_one_hot_labels)
real_image_and_labels = combine_vectors(real, image_one_hot_labels)
disc_fake_pred = disc(fake_image_and_labels)
disc_real_pred = disc(real_image_and_labels)
#### END CODE HERE ####
```

در حین آموزش هرچه جلوتر می رویم، تصاویر fake تولید شده توسط مولد به تصاویر واقعی نزدیکتر می شوند. نمودار ضرر بدست آمده در گام های پایانی به صورت زیر است:



مولد تلاش می کند میزان ضرر را کاهش دهد در حالیکه ممیز سعی می کند ضرر را افزایش دهد.

منابع

https://www.v7labs.com/blog/multi-task-learning-guide

https://www.analyticsvidhya.com/blog/2020/11/hyperparameter-tuning-using-optuna/

https://medium.com/pytorch/using-optuna-to-optimize-pytorch-hyperparameters-990607385e36

https://perlitz.github.io/hyperparameter-optimization-with-optuna/

https://towardsdatascience.com/implementing-word2vec-in-pytorch-from-the-ground-up-c7fe5bf99889

https://www.analyticsvidhya.com/blog/2021/07/word2vec-for-word-embeddings-a-beginners-guide/