به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

درس شبکه‌های عصبی و یادگیری عمیق

تمرین پنجم

نام و نام خانوادگی	محمدامین یوسفی
شماره دانشجویی	810100236
نام و نام خانوادگی	محمدرضا نعمتی
شماره دانشجویی	810100226

فهرست

شکل‌ها و جدول‌ها

پرسش ۱ – پیش‌بینی نیروی باد به کمک مبدل و تابع خطای Huber

۱-۱. مقدمه

روش های آماری سنتی مانند ARIMA و GARCH چه محدویت‌هایی دارند؟

- وابستگی به فرضیات توزیع خاص داده‌ها که ممکن است در بسیاری از شرایط واقعی نقض شوند
- نیاز به تست‌های همواری smoothness و پیش‌پردازش پیچیده داده.
- کاهش توانایی در مدل‌سازی پدیده‌های غیرخطی و شرایط ناپایدار.

برخی از مزایای مدل‌های یادگیری ماشین مانند SVM، Random Forest و XGboost نسبت به رویکردهای آماری سنتی در پیش‌بینی نیروی باد چیست؟

- توانایی پردازش حجم بالای داده‌ها و استخراج الگوهای پیچیده از داده‌های غیرخطی.
- دقت پیش‌بینی بالاتر به دلیل قابلیت یادگیری ویژگی‌های پنهان و وابستگی‌های طولانی‌مدت.
- تطبیق‌پذیری بهتر در شرایط مختلف و قابلیت کاربرد در زمینه‌های متنوع.

مدل‌های یادگیری عمیق مانند LSTM، GRU و CNN چگونه محدودیت‌های روش‌های یادگیری ماشین سنتی در پیش‌بینی نیروی باد برطرف می‌کنند؟

- یادگیری ویژگی‌های پیچیده: این مدل‌ها توانایی یادگیری ویژگی‌های غیرخطی و پیچیده داده‌ها را دارند که در مدل‌های سنتی مانند SVM و Random Forest کمتر قابل دستیابی است.
- حفظ وابستگی‌های طولانی‌مدت: مدل‌هایی مانند LSTM و GRU با طراحی حافظه داخلی، قادر به یادگیری وابستگی‌های بلندمدت در داده‌های زمانی هستند که در روش‌های سنتی مشکل است.
- مدیریت تغییرات غیرمنتظره و نویز: مدل‌های CNN با شناسایی الگوهای مکانی و زمانی در داده‌ها می‌توانند تاثیر نویز و نوسانات را کاهش دهند.

اهمیت مکانیسم خودتوجهی (self-attention) شبکه مبدل را در یادگیری وابستگی‌های بلندمدت و همبستگی‌های محلی بیان کنید.

- برقراری وابستگی‌های جهانی: مکانیسم خودتوجهی اطلاعات موجود در تمام نقاط دنباله داده را در نظر می‌گیرد و وابستگی‌های بلندمدت را با دقت بالا تشخیص می‌دهد.
- استخراج همبستگی‌های محلی: از طریق وزن‌دهی به نقاط داده با توجه به میزان اهمیت آن‌ها در زمینه فعلی، همبستگی‌های محلی و جزئی نیز به خوبی شناسایی می‌شوند.
- بهبود قابلیت پیش‌بینی: مکانیسم خودتوجهی باعث می‌شود مدل بتواند همزمان جزئیات محلی و وابستگی‌های بلندمدت را در داده‌ها لحاظ کند که این امر پیش‌بینی دقیق‌تری را فراهم می‌کند.

تابع خطای Huber چگونه پایداری (stability) و استحکام (robustness) مدل‌های پیش‌بینی را برای داده‌های باد فراساحلی افزایش می‌دهد؟

- کاهش حساسیت به داده‌های پرت: تابع خطای Huber با ترکیب ویژگی‌های squared loss و absolute loss، حساسیت مدل به داده‌های پرت را کاهش می‌دهد. در شرایطی که خطا کوچک باشد، مانند خطای مربعی عمل می‌کند و برای خطاهای بزرگ، به خطای مطلق تغییر حالت می‌دهد.
- حفظ پیوستگی و مشتق‌پذیری: تابع Huber به دلیل پیوستگی و مشتق‌پذیری کامل در کل دامنه خود، فرآیند بهینه‌سازی را پایدارتر می‌کند و اجازه می‌دهد از روش‌هایی مانند gradient descent برای تنظیم پارامترها استفاده شود.
- مدیریت نوسانات بالا: با مدیریت بهتر داده‌های دارای نوسانات زیاد، تابع Huber کمک می‌کند مدل‌های پیش‌بینی در مواجهه با تغییرات غیرمنتظره در داده‌های باد فراساحلی دقت و robustness بیشتری داشته باشند.

۱-۲. آماده‌سازی

ساختار Autoencoder برای چه هدفی مورد استفاده قرار گرفته است؟ این ساختار از چه بخش‌هایی تشکیل شده است؟ به کمک کتابخانه‌های مربوط، یک Autoencoder با ابرپارامترهای دلخواه طراحی کنید.

در این مقاله، ساختار Autoencoder برای بازسازی و کاهش نویز داده‌های نیروی باد فراساحلی استفاده شده است. هدف اصلی استفاده از این ساختار، حفظ ویژگی‌های اصلی داده‌ها و بهبود دقت پیش‌بینی از طریق حذف نویزهای اضافی است. این فرآیند باعث می‌شود داده‌های بازسازی‌شده در نواحی نوسانی صاف‌تر شوند و در نواحی ثابت دقت بیشتری داشته باشند. با بهینه‌سازی یک تابع هدف، که معمولا MSE، $L(x,\hat{x}) = (x-\hat{x})^2$ است، Autoencoder تلاش می‌کند داده‌های اصلی را بدون تاثیر نویز بازسازی کند. این روش به بهبود دقت مدل‌های پیش‌بینی نیروی باد کمک می‌کند و پیش‌نیازی برای ایجاد داده‌های تمیز و بدون خطا جهت استفاده در مراحل بعدی است. Autoencoder به عنوان یک روش یادگیری بدون نظارت، با فشرده‌سازی داده‌ها و بازسازی آن‌ها، بازنمایی‌های کم‌بعدی اما معناداری از داده‌ها ایجاد می‌کند.

ساختار Autoencoder شامل دو بخش اصلی است:

1. **Encoder**: این بخش وظیفه تبدیل داده‌های ورودی به یک بازنمایی پنهان را بر عهده دارد. این بازنمایی پنهان فضای کم‌بعدی‌تری است که ویژگی‌های کلیدی داده‌ها را استخراج می‌کند. در این مقاله، این فرآیند با استفاده از یک تبدیل ریاضیاتی $f: R^d \rightarrow R^h$ مدل‌سازی شده است. در این رابطه، d تعداد ویژگی‌های اصلی داده‌ها و h ابعاد فضای پنهان است.
2. **Decoder**: این بخش بازنمایی پنهان را به فضای اولیه بازمی‌گرداند و داده‌ها را بازسازی می‌کند. فرآیند بازسازی با یک تبدیل ریاضیاتی $g: R^h \rightarrow R^d$ انجام می‌شود، که هدف آن کمینه‌سازی اختلاف بین داده‌های اولیه و بازسازی‌شده است.

```
class DenoisingAutoencoder:
    def init(self, input_dim, encoding_dim, hidden_dims=None,
        activation="relu", learning_rate=0.001):
        hidden_dims = hidden_dims or []
        self.autoencoder = self._build_model(input_dim, encoding_dim,
        hidden_dims, activation, learning_rate)

    def _build_model(self, input_dim, encoding_dim, hidden_dims, activation,
        learning_rate):
        encoder_input = Input(shape=(input_dim,))
        x = encoder_input
        for hidden_dim in hidden_dims:
```

```
            x = layers.Dense(hidden_dim, activation=activation)(x)
        encoded = layers.Dense(encoding_dim, activation=activation)(x)

        x = encoded
        for hidden_dim in reversed(hidden_dims):
            x = layers.Dense(hidden_dim, activation=activation)(x)
        decoded = layers.Dense(input_dim, activation="sigmoid")(x)

        autoencoder = Model(encoder_input, decoded, name="AutoEncoder")
        autoencoder.compile(optimizer=Adam(learning_rate=learning_rate),
        loss="mse")
        return autoencoder

    def train(self, X_train, epochs=50, batch_size=64):
        return self.autoencoder.fit(X_train, X_train, epochs=epochs,
        batch_size=batch_size, verbose=1)

    def denoise(self, X):
        return self.autoencoder.predict(X)

autoencoder = DenoisingAutoencoder(input_dim=normalized_data.shape[1],
        encoding_dim=2, hidden_dims=[16, 8])
autoencoder.train(normalized_data)
```

پیاده سازی یک Denoising Autoencoder انجام شده است که هدف آن یادگیری نمایش‌های فشرده‌شده از داده‌ها و بازسازی آنها با حذف نویز است. کلاس دارای لایه‌های قابل تنظیم برای Encoding و Decoding است. از یک تابع فعال‌سازی قابل تغییر استفاده می‌کند و با استفاده از MSE بهینه‌سازی می‌شود. هایپرپارامترهای استفاده شده در جدول 1-1 قابل مشاهده است.

جدول 1-1. هایپرپارامترهای استفاده شده برای **Autoencoder**

Encoding Dimension	2
Hidden Dimensions	[16, 8]
Activaion Function	ReLU
Last Layer Activation	Sigmoid
Epochs	50
Batch Size	64
Optimizer	Adam
Learning Rate	0.001
Loss Function	Mean Squared Error

مکانیزم توجه و Positional Encoding در شبکه مبدل برای چه هدفی مورد استفاده قرار می‌گیرند؟

مکانیزم توجه:

مکانیزم توجه یکی از اجزای اصلی شبکه Transformer است که به مدل اجازه می‌دهد تا اهمیت نسبی نقاط مختلف داده را در یک دنباله زمانی تعیین کند. اهداف و ویژگی‌های این مکانیزم به شرح زیر است:

- یادگیری وابستگی‌های بلندمدت: با محاسبه وزن‌های توجه برای تمام نقاط داده در یک دنباله، مدل می‌تواند وابستگی‌های بلندمدت میان نقاط دور از هم را شناسایی کند. این ویژگی به خصوص برای داده‌های سری زمانی، مانند پیش‌بینی نیروی باد، بسیار مفید است.
- استخراج همبستگی‌های محلی: مکانیزم توجه قادر است همبستگی‌های محلی میان نقاط نزدیک در دنباله را نیز شناسایی کند و وزن بیشتری به نقاط مرتبط‌تر اختصاص دهد.
- انعطاف‌پذیری و تطبیق‌پذیری: برخلاف مدل‌های سنتی مبتنی برRNN، مکانیزم توجه نیازی به پردازش ترتیبی ندارد و می‌تواند به طور همزمان تمام نقاط دنباله را بررسی کند، که منجر به افزایش سرعت و دقت می‌شود.

Positional Encoding:

شبکه Transformer فاقد ساختار داخلی برای درک ترتیب داده‌ها است. برای حل این مشکل از Positonal Encoding استفاده شده است تا اطلاعات مربوط به ترتیب و موقعیت نقاط داده در دنباله زمانی را ارائه دهد. اهداف این مکانیزم عبارت‌اند از:

- ارائه اطلاعات ترتیبی: Positonal Encoding به مدل کمک می‌کند تا موقعیت نسبی هرنقطه داده را در دنباله زمانی درک کند، زیرا مدل‌های مبدل به طور پیش‌فرض ترتیب نقاط را نمی‌شناسند.
- حفظ همبستگی داده‌ها: با اضافه کردن بردارهای سینوسی و کسینوسی به داده‌ها، اطلاعات موقعیتی به گونه‌ای به داده‌ها اضافه می‌شود که مدل بتواند همبستگی میان نقاط مجاور و دورتر را بهتر یاد بگیرد.
- تعمیم بهتر در دنباله‌های بلند: Positional Encoding با مقیاس‌بندی سینوسی و کسینوسی می‌تواند اطلاعات موقعیتی را برای دنباله‌های بلند حفظ کند و به مدل امکان یادگیری وابستگی‌های پیچیده را می‌دهد.

با مطابعه رابطه ریاضی تابع خطای Huber، رفتار این تابع را شرح دهید. توضیح دهید هدف از استفاده از این تابع در این آزمایش چیست. این تابع خطا را پیاده‌سازی کنید.

شرح رفتار تابع خطای Huber و هدف استفاده در این آزمایش:

تابع خطای Huber ترکیبی از خطای مربعی و خطای مطلق است و به گونه‌ای طراحی شده که رفتار متفاوتی در مواجهه با خطاهای کوچک و بزرگ نشان می‌دهد. این تابع به صورت زیر تعریف می‌شود:

$$L_\delta(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 |y-\hat{y}| \le \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 \text{otherwise} \end{cases}$$

در این رابطه، y مقدار واقعی، $\hat{y}$ مقدار پیش‌بینی شده و δ پارامتر آستانه که تعیین می‌کند خطا به صورت مربعی یا مطلق محاسبه شود.

تحلیل رفتار تابع Huber:

1. **خطاهای کوچک $|y-\hat{y}| \le \delta$**
   - در این محدوده، تابع Huber مانند Squared Loss رفتار می‌کند.
   - این رفتار باعث می‌شود مدل به حداقل‌سازی خطاهای کوچک حساس‌تر باشد و دقت بالایی برای پیش‌بینی داده‌های معمولی ارائه دهد.
2. **خطاهای بزرگ $|y-\hat{y}| > \delta$**
   - برای خطاهای بزرگ، تابع به یک فرم خطی تبدیل می‌شود.
   - این تغییر رفتار باعث کاهش حساسیت تابع به داده‌های پرت می‌شود و از اثر مخرب آن‌ها بر فرآیند بهینه‌سازی جلوگیری می‌کند.
3. **پیوستگی و مشتق‌پذیری**
   - تابع Huber در نقطه آستانه δ پیوسته و مشتق‌پذیر است. این ویژگی باعث می‌شود فرآیند بهینه‌سازی به طور پیوسته و پایدار انجام شود و امکان استفاده از الگوریتم‌هایی مانند gradient descent فراهم باشد.

هدف استفاده از تابع Huber در این آزمایش:

- مدیریت نویز و نوسانات زیاد: داده‌های نیروی باد فراساحلی دارای نوسانات شدید و مقادیر پرت هستند. استفاده از تابع Huber به مدل کمک می‌کند تا اثر این نویزها را کاهش داده و دقت پیش‌بینی را افزایش دهد.
- بهبود پایداری مدل: با توجه به ماهیت پیوسته و مشتق‌پذیر بودن تابع Huber، فرآیند تنظیم پارامترها در مدل به طور پایدار و هموار انجام می‌شود و از همگرایی نامناسب جلوگیری می‌کند.
- افزایش استحکام در برابر داده‌های پرت: ویژگی خطی تابع Huber در مواجهه با خطاهای بزرگ باعث می‌شود داده‌های پرت تأثیر زیادی بر مدل نداشته باشند. این ویژگی برای داده‌های پراکنده و متغیر باد بسیار ارزشمند است.

```
def huber_loss(y_true, y_pred, delta=1.0):
    error = y_true - y_pred
    is_small_error = tf.abs(error) <= delta
    squared_loss = 0.5 * tf.square(error)
    linear_loss = delta * (tf.abs(error) - 0.5 * delta)

    return tf.reduce_mean(tf.where(is_small_error, squared_loss, linear_loss),
       axis=-1)
```

روش کار الگوریتم Slime Mould را بیان کنید و آن را پیاده‌سازی کنید. آرگومان‌های ورودی این الگوریتم را نام ببرید و هر یک را توضیح دهید.

روش کار

الگوریتم Slime Mould (SMA) یک روش بهینه‌سازی مبتنی بر رفتار طبیعی کپک‌های مخاطی است که از طریق مکانیزم‌هایی مانند حرکت سلولی و انتشار شیمیایی، منبع غذایی را جستجو و شناسایی می‌کنند. این الگوریتم از سه مرحله اصلی برای بهینه‌سازی استفاده می‌کند:

1. **مرحله جذب منابع غذایی:**

عامل‌های الگوریتم (agents) به سمت بهترین موقعیت (بالاترین مقدار fitness) در جمعیت حرکت می‌کنند. وزن‌دهی بر اساس مقدار fitness هر agent تعیین می‌شود تا میزان جذب هر عامل متناسب با کیفیت محل آن تنظیم شود.

2. **محاصره منابع:**

در این مرحله، agentها با استفاده از وزن‌های محاسبه‌شده به موقعیت‌های دیگر نزدیک می‌شوند یا موقعیت جدیدی در فضای جستجو انتخاب می‌کنند. این مکانیزم به الگوریتم اجازه می‌دهد که هم از جستجوی local و هم از جستجوی global بهره ببرد.

3. **مرحله جستجوی جهانی:**

در صورتی که agentها در نزدیکی یک منبع غذایی قرار بگیرند، حرکت به سمت موقعیت‌های تصادفی در فضای جستجو انجام می‌شود تا تنوع جمعیت حفظ شود و از گیر افتادن در local optimum جلوگیری شود.

در کل الگوریتم Slime Mould با الهام از رفتار طبیعی کپک‌های مخاطی، از وزن‌دهی مبتنی بر fitness، مکانیزم جستجوی local و global برای یافتن بهترین راه‌حل در مسائل بهینه‌سازی استفاده می‌کند.

پیاده‌سازی

```
class SlimeMouldAlgorithm:
    def init(self, objective_function, num_agents, max_iterations, bounds):
        self.objective_function = objective_function
        self.num_agents = num_agents
        self.max_iterations = max_iterations
        self.bounds = bounds

        self.dimensions = len(bounds)
        self.population = self.initialize_population()
        self.best_position = None
        self.best_fitness = float("inf")

    def initialize_population(self):
        return np.random.uniform(self.bounds[:, 0], self.bounds[:, 1],
     (self.num_agents, self.dimensions))

    def fitness(self, position):
        return self.objective_function(position)

    def update_population(self, fitness_values):
        sorted_indices = np.argsort(fitness_values)
        sorted_population = self.population[sorted_indices]

        best_agent = sorted_population[0]
        best_fitness = fitness_values[sorted_indices[0]]

        if best_fitness < self.best_fitness:
```

```
            self.best_position = best_agent
            self.best_fitness = best_fitness

        weight = np.zeros(self.num_agents)
        for i in range(self.num_agents):
            if i < self.num_agents // 2:
                weight[i] = 1 + np.log(1 + (best_fitness -
    fitness_values[sorted_indices[i]]) /
                                       (best_fitness -
    fitness_values[sorted_indices[-1]] + 1e-8))
            else:
                weight[i] = 1 - np.log(1 + (fitness_values[sorted_indices[i]] -
    fitness_values[sorted_indices[-1]]) /
                                       (best_fitness -
    fitness_values[sorted_indices[-1]] + 1e-8))

        for i in range(self.num_agents):
            if np.random.rand() < 0.8:
                rand_index = np.random.randint(0, self.num_agents)
                self.population[i] += weight[i] * (self.population[rand_index]
    - self.population[i])
            else:
                rand_position = np.random.uniform(self.bounds[:, 0],
    self.bounds[:, 1], self.dimensions)
                self.population[i] += weight[i] * (rand_position -
    self.population[i])

        self.population = np.clip(self.population, self.bounds[:, 0],
    self.bounds[:, 1])

    def optimize(self):
        for iteration in range(self.max_iterations):
            fitness_values = np.array([self.fitness(agent) for agent in
    self.population])
            self.update_population(fitness_values)

        return self.best_position, self.best_fitness
```

آرگومان‌های ورودی الگوریتم:

1. **objective_function**

تابع هدفی که باید بهینه‌سازی شود. این تابع معیار عملکرد هر موقعیت (یا راه‌حل) را ارائه می‌دهد. در مسائل عددی می‌تواند یک تابع ریاضی مانند $sin(x) + x^2$ باشد یا در مسائل کاربردی، تابعی که خطا یا هزینه را محاسبه کند.

2. **num_agents**

تعداد agentها یا اعضای جمعیت در الگوریتم. تعداد بیشتر agent معمولا باعث افزایش دقت جستجو می‌شود اما هزینه محاسباتی را نیز بالا می‌برد.

3. **max_iterations**

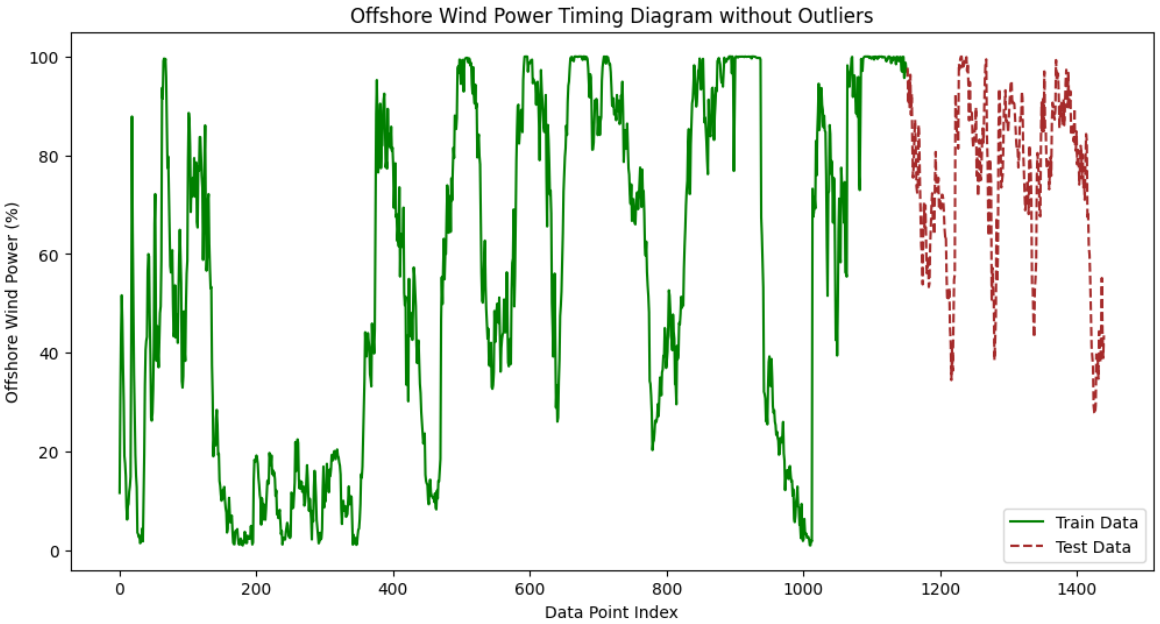
حداکثر تعداد تکرار الگوریتم برای یافتن بهینه. نقش تعیین‌کننده مدت زمان اجرای الگوریتم و عمق جستجو در فضای مسئله.

4. **bounds**

محدوده مجاز برای مقادیر متغیرهای مسئله را مشخص می‌کند. فرمت آن یک آرایه دوبعدی است که حداقل و حداکثر مقدار هر متغیر را مشخص می‌کند و تضمین می‌کند که هر agent در فضای معتبر مسئله جستجو کند.

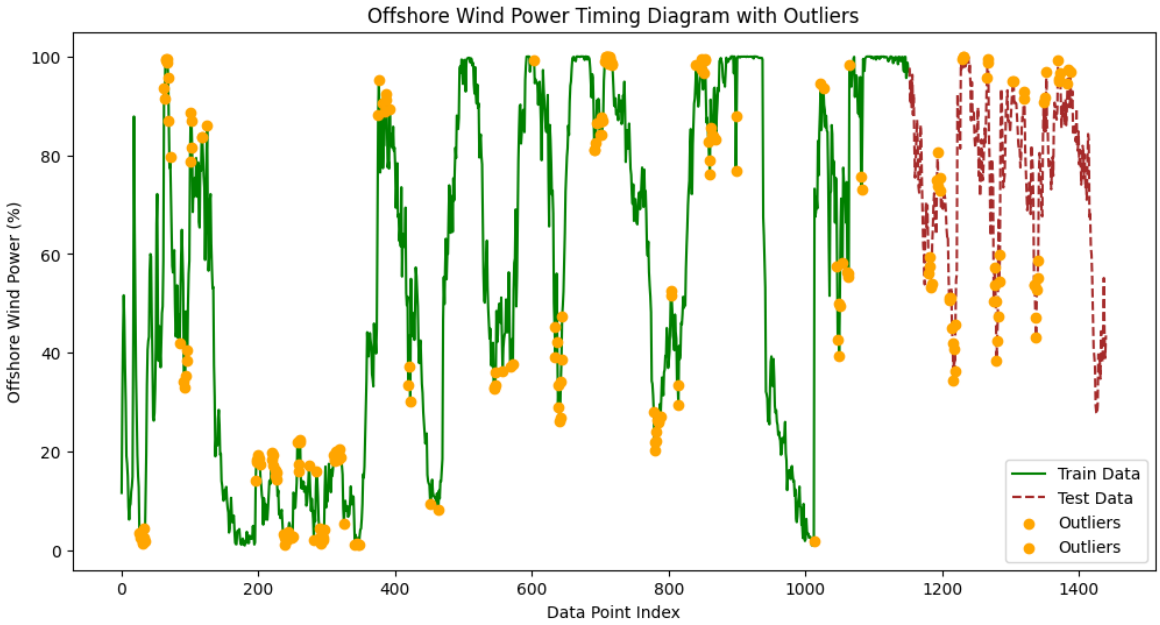
3-1. روش‌شناسی و نتایج

در ابتدا شکلی همانند شکل 6 مقاله تولید می‌کنیم.



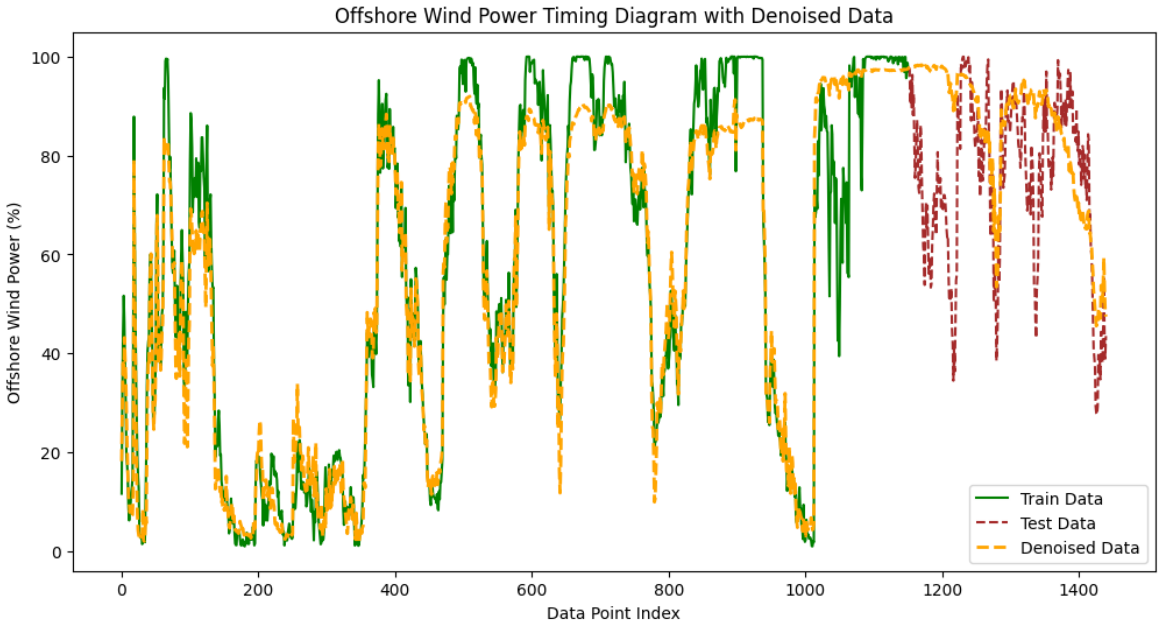
شکل 1-1. قدرت باد در گذشت زمان

سپس outlierها را در داده آموزش و تست می‌یابیم و نمایش می‌دهیم. در اینجا برای پیروی از مقاله، مقدار threshold برابر 1 انتخاب شده است اما 2 به نظر انتخاب منطقی‌تری می‌باشد.



شکل 1-2. قدرت باد در گذشت زمان با مشخص شدن outlierها

سپس داده‌ها را نرمال‌سازی کرده و به کمک آموزش اتوانکودر طراحی شده، ذاذه‌ها را دینویز می‌کنیم.



شکل 1-3. قدرت باد در گذشت زمان در دو حالت عادی و دینویز شده

در ابتدای این بخش، برای همگام شدن با مقاله، 1440 داده اول انتخاب شد .

سپس داده‌ها به دو بخش مجزا تقسیم شدند: داده‌های آموزش و داده‌های تست. هدف از این کار، ایجاد مرز مشخصی بین داده‌هایی است که مدل بر اساس آن‌ها آموزش می‌بیند و داده‌هایی که برای ارزیابی

عملکرد مدل استفاده می‌شوند. انجام این مرحله قبل از هرگونه تغییر یا نرمال‌سازی در داده‌ها ضروری است؛ زیرا در غیر این صورت، ممکن است اطلاعات مجموعه تست به‌صورت ناخواسته در فرایند آموزش مدل تأثیر بگذارد(Data Leakage).

پس از تقسیم داده‌ها، نرمال‌سازی به‌طور جداگانه بر روی مجموعه‌های آموزش و تست انجام شد. این فرایند به منظور استانداردسازی مقیاس متغیرها و جلوگیری از غلبه ویژگی‌هایی با مقیاس بزرگ‌تر بر ویژگی‌های با مقیاس کوچک‌تر انجام می‌شود. نرمال‌سازی پس از تقسیم داده‌ها اجرا شد تا اطمینان حاصل شود که مقیاس‌گذاری داده‌های تست به هیچ وجه تحت تأثیر داده‌های آموزش قرار نمی‌گیرد.

در گام بعدی، متغیرهای مستقل (ویژگی‌ها) و متغیر وابسته (هدف) از داده‌ها استخراج شدند و عمل denoising روی آنها اعمال شد.

در نهایت، داده‌ها با استفاده از روش پنجره متحرک آماده‌سازی شدند. این روش به منظور تبدیل داده‌های زمانی به فرم قابل استفاده برای مدل‌های یادگیری ماشین مورد استفاده قرار گرفت. این گام آخرین مرحله پیش‌پردازش است، زیرا باید روی داده‌های نرمال‌سازی شده و به‌درستی تقسیم‌شده اعمال شود.

ترتیب مراحل ذکرشده به‌طور مستقیم بر کیفیت آموزش مدل و ارزیابی آن تأثیر می‌گذارد. برای توضیح اهمیت این ترتیب، به یک مثال نقض اشاره می‌کنم: فرض کنید نرمال‌سازی قبل از تقسیم داده‌ها به مجموعه‌های آموزش و تست انجام شود. در این حالت، میانگین و انحراف معیار کل داده‌ها (شامل داده‌های تست) برای نرمال‌سازی استفاده می‌شود. این امر منجر به نشت اطلاعات از داده‌های تست به داده‌های آموزش خواهد شد. در نتیجه، مدل در حین آموزش به اطلاعاتی دسترسی خواهد داشت که قرار است تنها در مرحله ارزیابی با آن مواجه شود. این موضوع باعث ایجاد دقت کاذب در مدل و کاهش کارایی آن در مواجهه با داده‌های جدید می‌شود.

در شکل 1-4 می‌توانید نتایج آموزش و ارزیابی مدل‌ها با توابع هزینه مختلف به روش single step را مشاهده کنید.

Model Evaluation Results

Model	Loss Function	MAE	RMSE	MAPE	R^2
MLP	MSE	0.017650259658694267	0.025354143232107162	0.10759906470775604	0.9347947910428047
MLP	Huber	0.02563931792974472	0.033005014061927795	0.17724701762199402	0.8895045891404152
RNN	MSE	0.015690630301833153	0.02184993587434292	0.09131639450788498	0.9515733122825623
RNN	Huber	0.012819786556065083	0.02036539651453495	0.07451143860816956	0.9579302296042442
Transformer	MSE	0.044301558285951614	0.05439966917037964	0.3093396723270416	0.6998233795166016
Transformer	Huber	0.03801577910780907	0.050759170204401016	0.24906106293201447	0.7386554777622223

شکل 1-4. نتایج ارزیابی مدل‌ها روی داده تست

نتایج نشان می‌دهد که مدل RNN با هر دو تابع زیان MSE و Huber عملکرد بهتری نسبت به سایر مدل‌ها داشته است. مقدار MAE و RMSE پایین‌تر در کنار R2 بالای 0.94 بیانگر دقت بالای این مدل در پیش‌بینی داده‌ها است. تفاوت بین عملکرد دو تابع زیان در این مدل ناچیز بوده و نشان می‌دهد که RNN به خوبی با داده‌های مسئله سازگار است.

مدل MLP نیز عملکرد نسبتاً خوبی داشته است، اما دقت آن کمتر از RNN است. استفاده از تابع زیان Huber در این مدل باعث کاهش خطا و افزایش دقت شده است، که نشان می‌دهد Huber می‌تواند در کاهش اثر داده‌های پرت موثر باشد. با این حال، همچنان R2 پایین‌تر از RNN باقی می‌ماند.

از سوی دیگر، مدل Transformerعملکرد ضعیف‌تری نسبت به RNN و MLP نشان داده است. مقدار MAE و RMSE بالاتر و R2 پایین‌تر از سایر مدل‌ها، به‌ویژه با تابع زیان Huber، نشان می‌دهد که این مدل به تنظیمات بیشتری نیاز دارد. این مسئله احتمالاً به دلیل حساسیت این مدل به داده‌های آموزشی یا هایپرمترهای ناهماهنگ است.

در اشکال زیر، می‌توانید نتایج ارزیابی مدل مبدل را در حالت multi step را مشاهده نمایید.

Model Evaluation Results

Model	Loss Function	MAE	RMSE	MAPE	R^2
Transformer	MSE	0.041572585701942444	0.054429687559604645	0.28244125843048096	0.726879321038723
Transformer	Huber	0.039798837155103683	0.055060096085071564	0.24722126126289368	0.7205512002110481

شکل 1-5. نتایج ارزیابی مدل مبدل در t+4

Model Evaluation Results

Model	Loss Function	MAE	RMSE	MAPE	R^2
Transformer	MSE	0.038951992988586426	0.04877437651157379	0.28055641055107117	0.7579071093350649
Transformer	Huber	0.037145569920539856	0.04869341105222702	0.25584518909454346	0.7586621269583702

شکل 1-**6**. نتایج ارزیابی مدل مبدل در t+8

Model Evaluation Results

Model	Loss Function	MAE	RMSE	MAPE	R^2
Transformer	MSE	0.03622713312506676	0.04872289299964905	0.23750367760658264	0.7652277406305075
Transformer	Huber	0.038068436086177826	0.04897396266460419	0.27842995524406433	0.7627686485648155

شکل 1-7. نتایج ارزیابی مدل مبدل در t+16

همانطور که مشاهده می‌کنید، در t + n ، هرچه مقدار n بیشتر می‌شود، عملکرد مبدل نیز بهتر می‌شود اما همچنان نیز عملکرد خوبی ندارد که این نشان‌دهنده این است که ابرپارامترهای این مدل نیاز به تغییرات دارند. اینکه با افزایش n نتایج بهتر می‌شوند، دور از انتظار است چرا که هرچه زمان‌های دورتر را پیشبینی کنیم، به صورت طبیعی باید عملکرد ما ضعیف تر باشد. این رفتار دور از انتظار مدل به علت کم بودن تعداد داده‌ها است و در صورت استفاده از تعداد بیشتری داده این مشکل رفع می‌شود (البته این مقدار تفاوت بسیار کم و در حد 2 درصد است). همچنین این مدل به علت پیچیدگی برای آموزش بهتر نیاز به داده بیشتری دارد و 1440 داده برای آن کم است.

در t + n، مقدار n بیان می‌کند که چند دقیقه پس از داده فعلی مدنظر است. برای مثال، t + 4 ، 40 دقیقه پس از زمان فعلی را نشان می‌دهد.

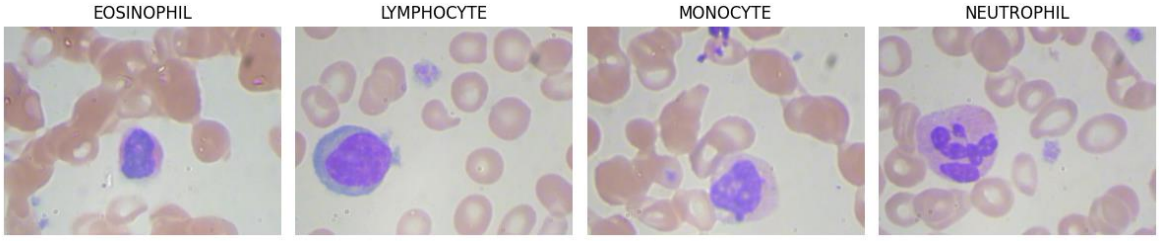
در نهایت نیز به کمک الگوریتم Slime mould، تلاش شد تا بهترین پارامترها برای مدل مبدل یافت شود. **کد این بخش در انتهای نوتبوک موجود است اما خروجی آن پاک شده و به علت طولانی بودن زمان اجرای آن و نبود وقت، فرصت برای اجرای مجدد وجود نداشت.**

- میزان drop_out: 0.3
- تعداد heads: 8
- میزان key_dim: 32

پرسش ۲ – استفاده از ViT برای طبقه‌بندی تصاویر گلبول‌های سفید

۲-۲. آماده‌سازی داده‌ها

نمایش نمونه تصاویر



شکل 2-1. نمونه‌هایی از هر دسته دیتا

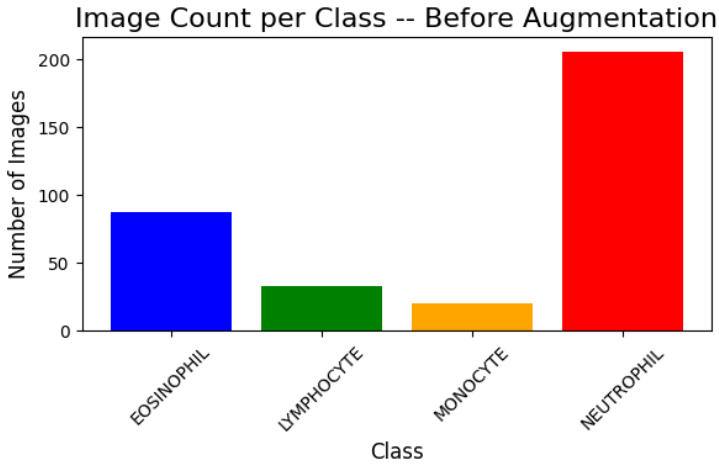
بررسی تعداد داده‌ها در هر کلاس

برای این بررسی، نمودار تعداد داده‌های هر کلاس را رسم می‌کنیم که در شکل 2-2 آورده شده است. مشاهده می‌شود که imbalance زیادی بین داده‌ها وجود دارد که در صورت رفع نکردن آن، میتواند مدل نهایی را دچار ضعف کند. به همین دلیل با استفاده از Data Augmentation های موجود در جدول 2-1، کلاس‌هایی که تعداد کمتری از بیشترین کلاس دارند را با ایجاد نمونه‌های جدید، به آن تعداد می‌رسانیم. در شکل 2-3، تعداد هر کلاس پس از Data Augmentation آورده شده است.

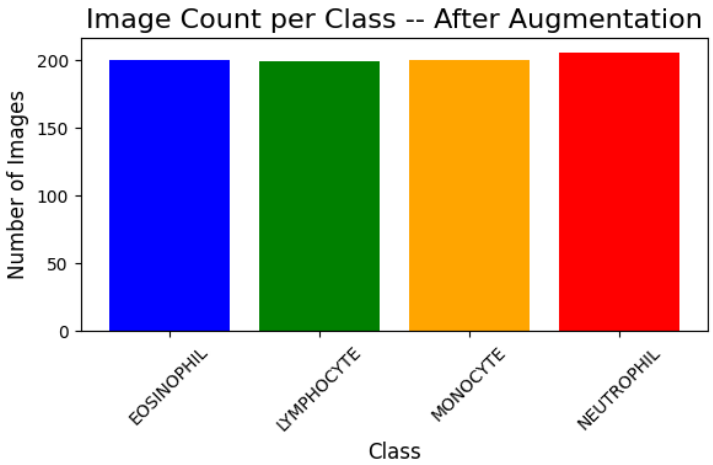
جدول 2-1. روش‌های انتخاب شده برای تقویت داده و توضیحات

Augmentation	Value	Description
rotation_range	25	تصاویر را در محدوده مثبت منفی می‌چرخاند. این عملیات تغییرات جزئی زاویه‌ای را شبیه‌سازی کرده و تعمیم‌پذیری مدل را بهبود می‌بخشد.
width_shift_range	0.2	این پارامتر تصاویری تولید می‌کنند که به صورت **افقی** جابه‌جا شده‌اند. مقدار داده‌شده، درصدی از اندازه تصویر اصلی است. این تکنیک برای مقاوم‌سازی مدل در برابر تغییرات موقعیتی سوژه در تصویر مفید است.
height_shift_range	0.2	این پارامتر تصاویری تولید می‌کنند که به صورت **عمودی** جابه‌جا شده‌اند. مقدار داده‌شده، درصدی از اندازه تصویر اصلی است. این تکنیک برای مقاوم‌سازی مدل در برابر تغییرات موقعیتی سوژه در تصویر مفید است.
shear_range	0.2	این پارامتر تصاویر را با اعمال یک تغییر شکل (shear) هندسی تغییر می‌دهد، به طوری که خطوط موازی تصویر، زاویه‌دار می‌شوند. این تکنیک

		برای تولید داده‌هایی که شکل سوژه تغییر کرده است، استفاده می‌شود و مدل را به چنین تغییراتی مقاوم می‌کند.
zoom_range	0.2	این تکنیک بخش‌هایی از تصویر را بزرگ یا کوچک‌نمایی می‌کند که باعث می‌شود مدل بتواند ویژگی‌ها را در مقیاس‌های مختلف یاد بگیرد. این کار کمک می‌کند تا مدل در برابر تغییرات مقیاس مقاوم‌تر شود و ویژگی‌های مهم را در سطوح مختلف تصویر شناسایی کند.
horizontal_flip	True	این روش تصاویر را به صورت تصادفی افقی برمی‌گرداند. این کار برای داده‌هایی که ویژگی‌هایشان در دو طرف تصویر یکسان است (مانند شکل گلبول سفید) مفید است و باعث می‌شود مدل از یادگیری الگوهای جهت‌دار غیرعمومی جلوگیری کند.



شکل 2-2. تعداد نمونه‌های هر کلاس قبل از **Data Augmentation**



شکل 2-3. تعداد نمونه‌های هر کلاس بعد از **Data Augmentation**

همچنین با توجه به محدودت منابع، 70 درصد تصاویر انتخاب شدند تا در فرایند آموزش و ارزیابی شرکت کنند. در کل 805 داده وجود داشت که زیرمجموعه 563 تایی از آن انتخاب شد.

ایجاد مجموعه نهایی

داده‌ها را به نسبت 90-10 به آموزش و ارزیابی تقسیم می‌کنیم. سپس DataLoader های آنها را ایجاد می‌کنیم تا به استفاده از آنها به آموزش مدل بپردازیم. توجه کنید که برای مدل ViT نیاز است که تصاویر normalized و به سایز مناسب باشند که با استفاده از `ViTImageProcessor`، این را انجام می‌دهیم.

```
subset_paths, _, subset_labels, _ = train_test_split(
    image_paths, labels,
    test_size=0.3,  # Keep only 70% of the data
    stratify=labels,
    random_state=42
)

train_paths, val_paths, train_labels, val_labels =
train_test_split(
    subset_paths,
    subset_labels,
    test_size=0.1,
    stratify=subset_labels,
    random_state=42
)

processor = ViTImageProcessor.from_pretrained(MODEL_NAME)

train_dataset = WBCDataset(train_paths, train_labels, processor)
val_dataset = WBCDataset(val_paths, val_labels, processor)
train_loader = DataLoader(train_dataset, Patch_size=16,
shuffle=True)
val_loader = DataLoader(val_dataset, Patch_size=16, shuffle=False)
```

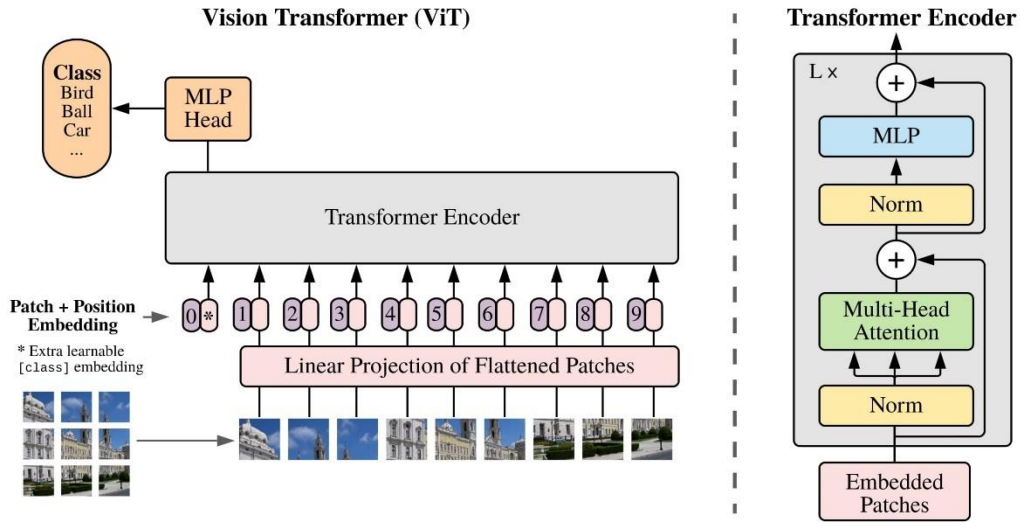
۲-۳: آموزش مدل‌ها

توضیح مدل ViT

مدل ViT (Vision Transformer) یک معماری مبتنی بر Transformer است که برای پردازش تصاویر طراحی شده است. برخلاف مدل‌های متداول برپایه CNN، ViT تصاویر را به صورت Patch های کوچک تقسیم می‌کند، سپس این Patchها را به بردارهای مسطح تبدیل کرده و به مدل Transformer می‌دهد.

این مدل از مکانیزم Attentionاستفاده می‌کند تا روابط بین Patchها را یاد بگیرد و ویژگی‌های مهم تصویر را استخراج کند.

از نقاط قوت ViT می‌توان به توانایی آن در یادگیری روابط بلند مدت بین بخش‌های مختلف تصویر و کاهش وابستگی به طراحی دستی فیلترها اشاره کرد. این مدل در داده‌های بسیار بزرگ و متنوع عملکرد فوق‌العاده‌ای دارد و اغلب می‌تواند نتایج بهتری نسبت بهCNN ها به دست آورد. با این حال، برای دستیابی به عملکرد مطلوب، نیازمند داده‌های آموزشی زیاد و قدرت محاسباتی بالا است. این مدل به دلیل ساختار ماژولار و انعطاف‌پذیرش در حوزه‌های مختلف پردازش تصویر، از جمله تشخیص اشیا و طبقه‌بندی، به‌سرعت محبوب شده است. شکل 2-4، نمایی از مدل ViT گوگل است که بخش‌های توضیح داده شده در آن دیده می‌شوند.



شکل 2-4. معماری مدل **ViT** گوگل

این مدل ابتدا تصویر ورودی را به Patchهای کوچک تقسیم می‌کند. این Patchها پس از مسطح‌سازی به Vectorهایی تبدیل شده و همراه با Position Embedding وارد مدل می‌شوند. یک Vector قابل یادگیری به نام [class] نیز اضافه می‌شود که نماینده خروجی نهایی است. سپس این ورودی‌ها به Transformer Encoder ارسال می‌شوند که شامل مکانیزم Multi-Head Attention برای یادگیری روابط بین Patchها و MLP برای پردازش ویژگی‌ها است. خروجی نهایی [class] برای طبقه‌بندی تصویر استفاده می‌شود. در پایین، مدل بارگذاری شده و پس از تنظیم تعداد کلاس خروجی به 4، معماری آن خروجی گرفته شده است.

```python
model = ViTForImageClassification.from_pretrained(
    google/vit-base-patch16-224-in21k,
    num_labels=len(classes),
    id2label={i: c for i, c in enumerate(classes)},
    label2id={c: i for i, c in enumerate(classes)}
)
model.classifier = torch.nn.Linear(model.config.hidden_size, 4)
print(model)
```

```
ViTForImageClassification(
  (vit): ViTModel(
    (embeddings): ViTEmbeddings(
      (patch_embeddings): ViTPatchEmbeddings(
        (projection): Conv2d(3, 768, kernel_size=(16, 16), stride=(16, 16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): ViTEncoder(
      (layer): ModuleList(
        (0-11): 12 x ViTLayer(
          (attention): ViTSdpaAttention(
            (attention): ViTSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
            (output): ViTSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): ViTIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): ViTOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (layernorm_after): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
  )
  (classifier): Linear(in_features=768, out_features=4, bias=True)
)
```

پیاده‌سازی و آموزش مدل‌ها

برای پیاده‌سازی و آموزش مدل‌ها، از هایپرپارامترهای زیر طبق مقاله استفاده شده است. دقت و خطای مدل‌ها در آموزش و ارزیابی برای هر بخش در شکل‌های 2-5 تا 2-10 آورده شده است. همچنین برای تعداد کل پارامترها و تعداد پارامتر های قابل آموزش در جدول 2-3 آورده شده است.

جدول 2-2. هایپرپارامترهای استفاده شده

Epochs	17
Batch Size	16
Optimizer	Adam
Learning Rate	0.002
Loss Function	Cross-Entropy Loss

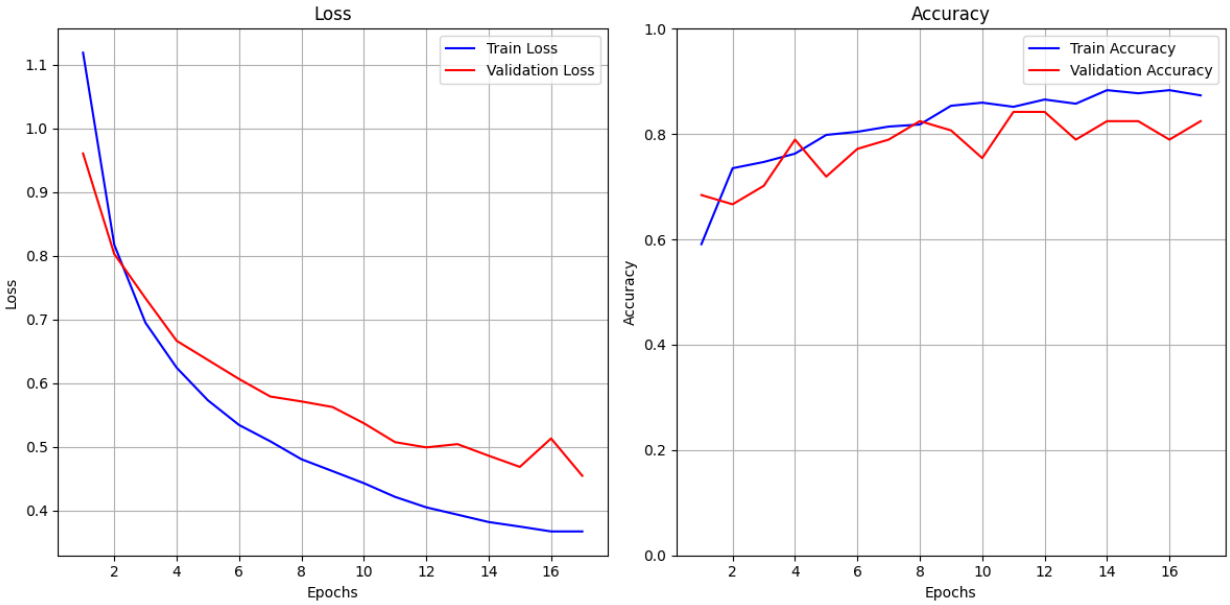
جدول 2-3. تعداد کل پارامترها و تعداد پارامترهای قابل آموزش در هر حالت

Model	Case	Total Parameters	Trainable Parameters
Google ViT	Fine-Tune **Classifier**	85,801,732	3,076
	Fine-Tune **Classifier** and **First 2** Layers of Encoder		14,178,820
	Fine-Tune **Classifier** and **Last 2** Layers of Encoder		14,178,820
	Full Fine-Tune		85,801,732
DenseNet-121	**Full** Fine-Tune	6,957,956	6,957,956
	Fine-Tune **Classifier**		4,100

مدل ViT

1. فقط Classifier قابل آموزش باشد

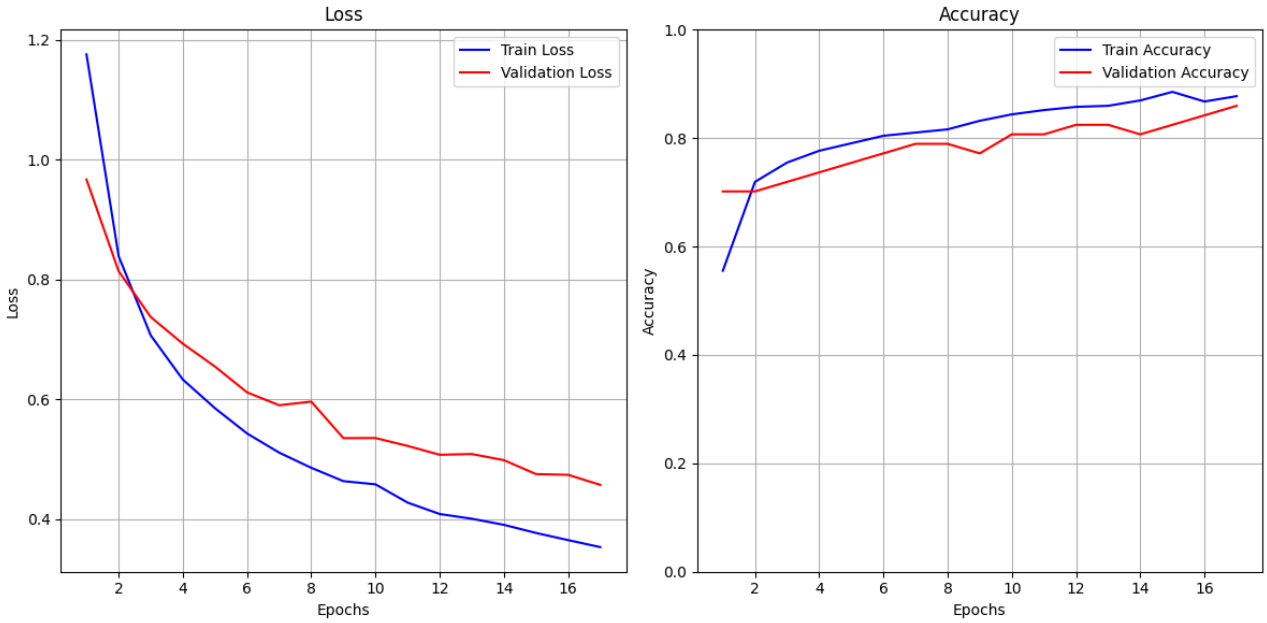
```
model = ViTForImageClassification.from_pretrained(MODEL_NAME)
model.classifier = torch.nn.Linear(model.config.hidden_size, len(classes))
model = model.to(device)

model = freeze(model)

for param in model.classifier.parameters():
    param.requires_grad = True

model_1 = ModelTrainer(model, train_loader, val_loader, loss_fn, optimizer)
model_1.fit(epochs=17)
```



شکل 2-5. دقت و خطای مدل **ViT** در حالت اول

2. Classifier و دو لایه اول Encoder قابل آموزش باشند

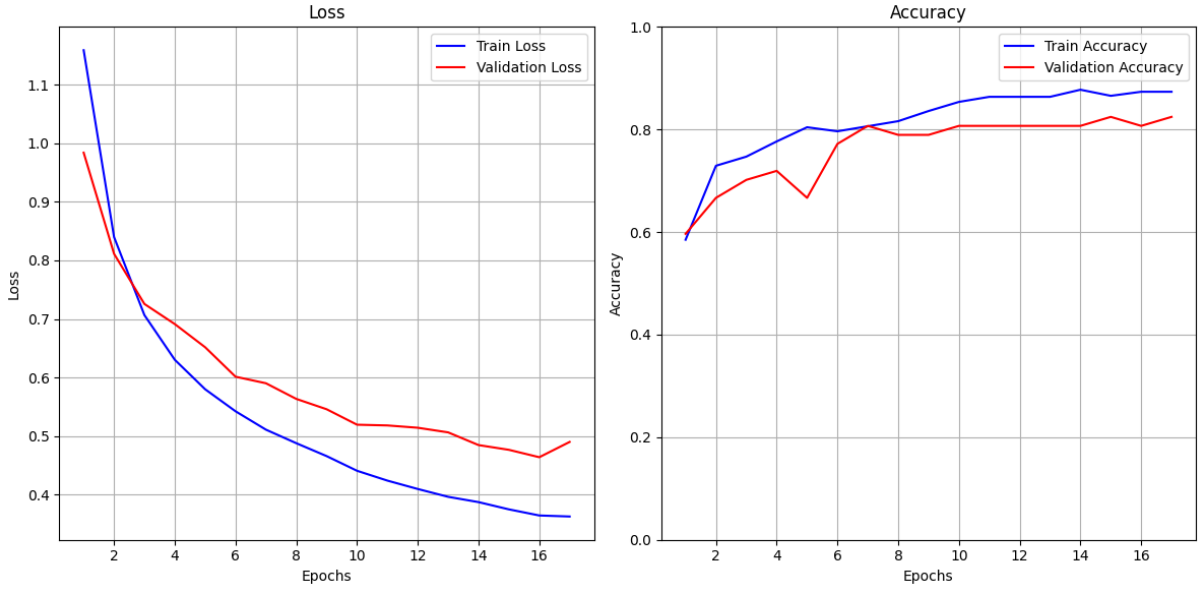
```
model = freeze(model)
for layer in model.vit.encoder.layer[:2]:
    for param in layer.parameters():
        param.requires_grad = True
for param in model.classifier.parameters():
    param.requires_grad = True
```



شکل 2-6. دقت و خطای مدل **ViT** در حالت دوم

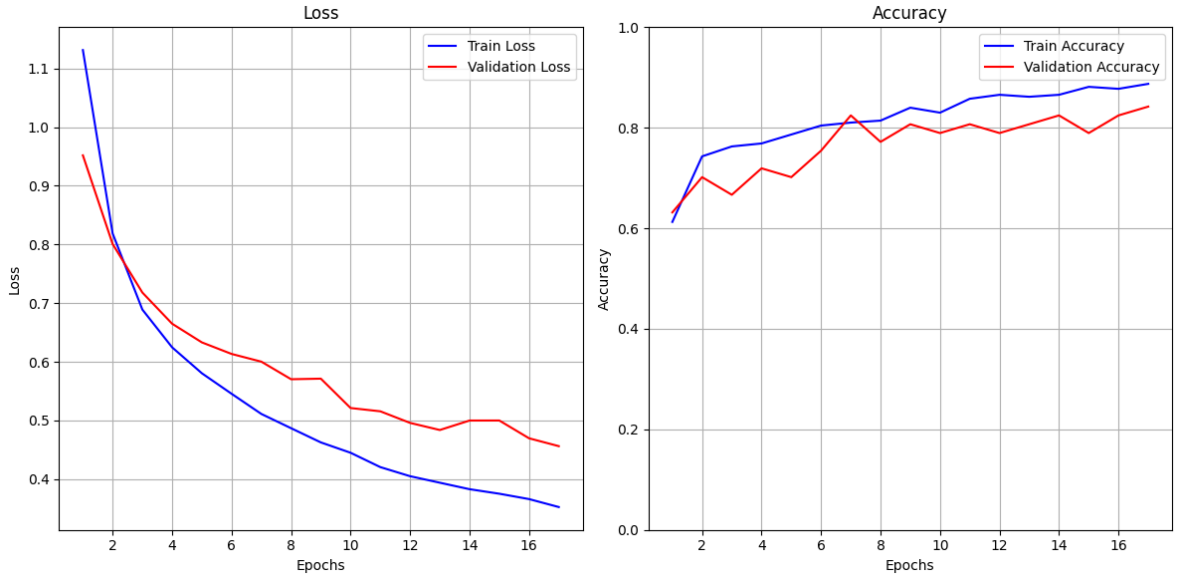
3. Classifier و دو لایه آخر Encoder قابل آموزش باشند

```
model = freeze(model)
for layer in model.vit.encoder.layer[-2:]:
    for param in layer.parameters():
        param.requires_grad = True
for param in model.classifier.parameters():
    param.requires_grad = True
```



شکل 2-7. دقت و خطای مدل **ViT** در حالت سوم

4. همه لایه‌ها قابل آموزش باشند



شکل 2-8. دقت و خطای مدل **ViT** در حالت چهارم

تحلیل

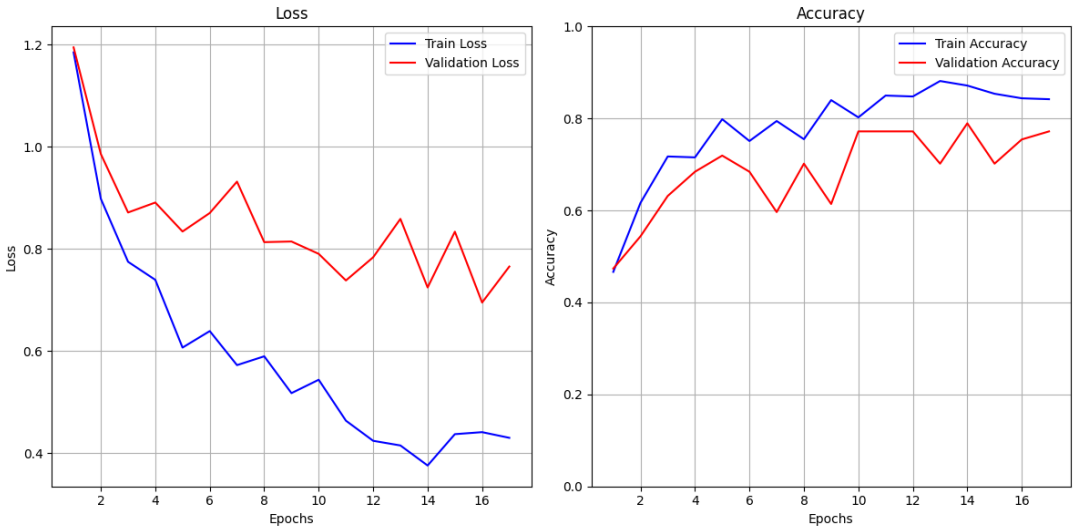
مشاهده می‌شود که همه حالت‌ها به دقت تقریبا یکسان و نزدیک به هم رسیدند. همچنین به نظر نمی‌رسد که هیچکدام از آنها دچار Overfit شده باشد، چون دقت آموزش و ارزیابی نزدیک به همدیگر حرکت می‌کنند. مدل در حالت اول که فقط Classifier آموزش داده می‌شود، در حین آموزش نوسان بیشتری نسبت به بقیه حالات دارد که میتوان دلیل آن را کم بودن پارامترهایی که در حال آموزش می‌باشند و تغییرات زیاد برای رسیدن به جواب مطلوب نسبت داد. *(البته این دلیل کمی شهودی است و باید بیشتر بررسی شود.)*

در کل مدل در همه حالت‌ها از همان ابتدا دقت نسبتا خوبی دارد و این را میتوان به دلیل دیتاست بزرگی که مدل بر روی آن Pre Train شده است، (ImageNet-21k) توجیه کرد.

مدل CNN: DenseNet-121

1. همه لایه‌ها قابل آموزش باشند

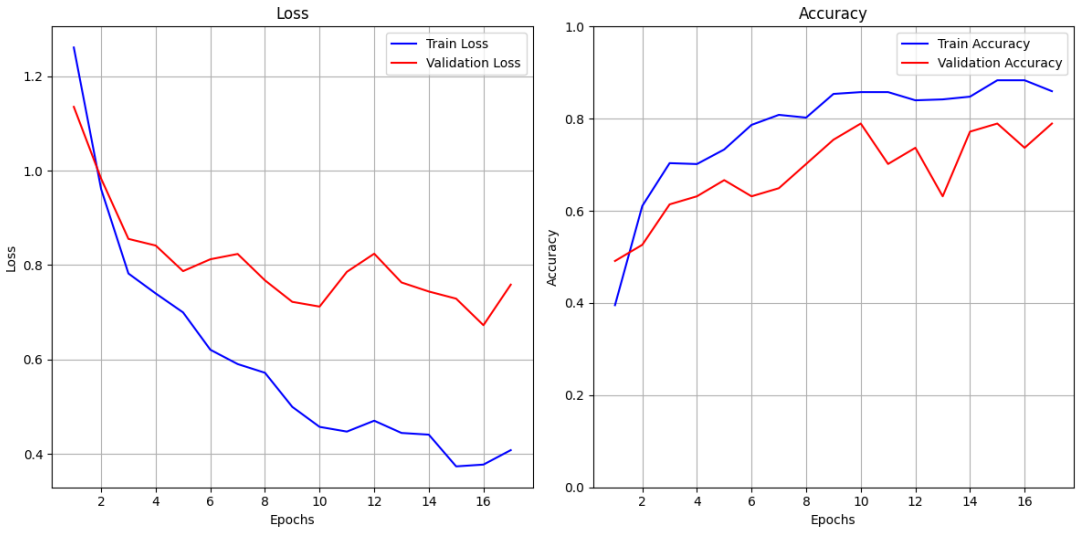
```
model = densenet121(pretrained=True)
model.classifier = nn.Linear(model.classifier.in_features, len(classes))
model = model.to(device)

# model = freeze(model)

model_5 = ModelTrainer(model, train_loader, val_loader, loss_fn, optimizer)
model_5.fit(epochs=17)
```



شکل 2-9. دقت و خطای مدل **DenseNet-121** در حالت اول

2. فقط Classifier قابل آموزش باشد

```
model = freeze(model)

for param in model.classifier.parameters():
    param.requires_grad = True
```



شکل 2-10. دقت و خطای مدل **DenseNet-121** در حالت دوم

مقایسه بین DenseNet-121 و ViT بر اساس نتایج آموزش و ارزیابی در حالتی که فقط Classifier آموزش داده شود، تفاوت‌های قابل توجهی در عملکرد و تعمیم‌پذیری این دو مدل را نشان می‌دهد. در مدل DenseNet-121، خطای آموزش به طور پیوسته کاهش می‌یابد، اما خطای ارزیابی پس از چند epoch به کمی افزایش می‌یابد. این مسئله نشان‌دهنده Overfit شدن مدل است. همچنین دقت ارزیابی در DenseNet-121 روند ثابتی ندارد و نشان می‌دهد که تعمیم‌پذیری این مدل به داده‌های جدید ضعیف‌تر است. برای رفع این مشکل باید روش‌هایی چون Early-Stopping (برای جلوگیری از بدتر شدن دقت) و استفاده از Data Augmentation بیشتر (برای بهتر شدن دقت) انجام داد.

در مقابل، مدل ViT عملکرد پایدارتر و بهتری از خود نشان می‌دهد. هر دو خطای آموزش و ارزیابی به‌طور پیوسته کاهش می‌یابند و در واقع خطای ارزیابی به دقت خطای آموزش را دنبال می‌کند که نشان‌دهنده تعمیم‌پذیری بهتر است. همچنین دقت ارزیابی در ViT روند صعودی و باثباتی دارد و بسیار نزدیک به دقت آموزش است، که نشان‌دهنده عملکرد قوی‌تر روی داده‌های دیده‌نشده است.

این تفاوت‌ها تا حد زیادی به معماری این دو مدل مربوط می‌شود. DenseNet-121 یک مدل بر پایه CNN است و در استخراج ویژگی‌های hierarchical و local در تصاویر بسیار خوب عمل می‌کند. با این

حال، این مدل ممکن است در درک وابستگی‌های کلی و global محدودیت داشته باشد. ViT که یک معماری برپایه Transformer دارد، از مکانیزم Self-Attention برای مدل‌سازی روابط global در کل تصویر استفاده می‌کند. این قابلیت به ViT اجازه می‌دهد تا الگوهای پیچیده‌تر و کلی‌تری را یاد بگیرد، که در نتایج بهتر آن در اعتبارسنجی و ثبات عملکرد منعکس شده است.

به طور کلی، مدل ViT در این شرایط عملکرد بهتری دارد. توانایی این مدل در حفظ خطای ارزیابی پایین و ارائه دقت پایدار، نشان‌دهنده تعمیم‌پذیری و استحکام بیشتر آن است. در صورتی که منابع محاسباتی کافی در اختیار باشد، Fine-Tuning بیشتر مدل ViT و استفاده از داده‌های بیشتر می‌تواند عملکرد آن را حتی بهتر کند.

۲-۴: تحلیل و نتیجه‌گیری

در جدول 2-4، همه نتایج مدل‌های آموزش داده شده جمع‌آوری شده است.

جدول 2-4. دقت و خطای آموزش و ارزیابی هر دو مدل در تمام حالت‌ها

Model	Case	Train Loss	Validation Loss	Train Accuracy	Validaton Accuracy
Google ViT	Fine-Tune **Classifier**	0.3668	0.4542	87.35 %	82.45 %
	Fine-Tune **Classifier** and **First 2** Layers of Encoder	0.3535	0.4574	87.74 %	85.96 %
	Fine-Tune **Classifier** and **Last 2** Layers of Encoder	0.3627	0.4900	87.35 %	82.45 %
	Full Fine-Tune	0.3524	0.4562	88.73 %	84.21 %
DenseNet-121	**Full** Fine-Tune	0.4298	0.7652	84.18 %	77.19 %
	Fine-Tune **Classifier**	0.4077	0.7585	85.96 %	78.94 %

حالتی که فقط Classifier و دو لایه اول Encoder در مدل ViT آموزش دیدند، **بهترین** دقت ارزیابی (85.96 %) را بدست آورده است. **کمترین** خطای ارزیابی (0.4552) نیز حالتی است که فقط Classifier مدل ViT آموزش دیده است. **بدترین** مدل از نظر دقت (77.19 %) و خطای ارزیابی (0.7652) هم حالتی که DenseNet-121 به طور کامل آموزش دیده است، می‌باشد که تفاوت چشمگیری با ViT دارد.

با توجه به جدول، ViT به طور کلی عملکرد بهتری نسبت به DenseNet-121 نشان می‌دهد. در تمامی حالت‌های Fine-Tuning، ViT توانسته دقت و خطای ارزیابی بهتری نسبت به DenseNet-121 داشته

باشد این نتایج برتری معماری برپایه Transformer در یادگیری الگوهای کلی و پیچیده‌تر نسبت به معماری برپایه CNN را تایید می‌کند.

نقش تعداد لایه‌های آموزش‌دیده واضح است؛ در ViT، Fine-Tuning محدود لایه‌ها (مثل دو لایه اول یا آخر Encoder) نتایج خوبی در دقت و خطای ارزیابی ارائه می‌دهد، زیرا این Fine-Tuning باعث حفظ ویژگی‌های از پیش آموزش‌دیده می‌شود. در مقابل، Full Fine-Tuning عملکرد مشابهی دارد اما به منابع بیشتری نیاز دارد. در DenseNet-121، حتی با Full Fine-Tuning، بهبود قابل‌توجهی نسبت به Fine-Tuning محدود مشاهده نمی‌شود، که نشان‌دهنده حساسیت کمتر این مدل به تغییرات در تعداد لایه‌های آموزش‌دیده است.

آیا ViT در شرایط موجود (مثلا داده‌های نویزدار یا کم‌حجم) توانسته جایگزین مناسبی برای CNN باشد؟

بله. ViT توانسته در شرایط موجود جایگزین مناسبی برایCNN باشد. زیرا با بهره‌گیری از مکانیزم Self-Attention، توانایی یادگیری روابط کلی و الگوهای پیچیده‌تر را دارد که در داده‌های کم‌حجم یا نویزدار به تعمیم‌پذیری بهتر کمک می‌کند. در نتایج مشاهده‌شده،ViT حتی با Fine-Tuning محدود نیز عملکرد بهتری در مقایسه باDenseNet-121 نشان داده است. با این حال، این موفقیت به منابع محاسباتی بالا و تنظیمات دقیق هایپرپارامترها وابسته است. ممکن است با تنظیم متفاوت هایپرپارامترها برای مدل DenseNet-121، به نتایج بهتری می‌رسیدیم. البته اگر منابع کافی در دسترس باشد، ViT گزینه‌ای قدرتمند برای جایگزینی CNN محسوب می‌شود؛ اما در شرایط محدودیت منابع یا نیاز به پردازش سریع‌تر، CNN همچنان کاربردی‌تر است.