ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA VẬT LÝ

NGUYỄN TRƯỜNG DANH

NEURAL STRUCTURED LEARNING WITH PYTHON/TENSORFLOW

TIỂU LUẬN
NGÀNH KỸ THUẬT ĐIỆN TỬ VÀ TIN HỌC
(CHƯƠNG TRÌNH ĐÀO TẠO CHUẨN)

Hà Nội - 2023

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA VẬT LÝ

NGUYỄN TRƯỜNG DANH

NEURAL STRUCTURED LEARNING WITH PYTHON/TENSORFLOW

TIỂU LUẬN
NGÀNH KỸ THUẬT ĐIỆN TỬ VÀ TIN HỌC
(CHƯƠNG TRÌNH ĐÀO TẠO CHUẨN)

GIẢNG VIÊN HƯỚNG DẪN: TS. NGUYỄN TIẾN CƯỜNG

Hà Nội - 2023

"*Cái tôi và sự hiểu biết tỷ lệ nghịch với nhau. Hiểu biết càng nhiều cái tôi càng bé. Hiểu biết càng ít, cái tôi càng to.*"

Albert Einstein

Lời cảm ơn

Để hoàn thành học phần cũng như báo cáo này đã có sự hướng dẫn và giúp đỡ tận tình của thầy TS. Nguyễn Tiến Cường. Vì vậy em muốn gửi lời cảm ơn sâu sắc nhất tới thầy đã hướng dẫn và chỉ dạy em trong học phần này cũng như các học phần khác. Em hi vọng sẽ được tiếp tục làm việc, học hỏi cùng với thầy và các thầy cô khác trong bộ môn ở những học phần tiếp theo, đặc biệt là khóa luận tốt nghiệp!

Mục lục

Danh sách hình vẽ

Danh sách tên viết tắt

NSL	**N**eural **S**truct **L**earning
CPU	**C**enter **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
API	**A**pplication **P**rogramming **I**nterface

Danh sách ký hiệu

v_t	tổng bình phương của gradien trong quá khứ
β	tham số trung bình
α	tốc độ học
δL	đạo hàm của hàm mất mát
w_t	trọng số tại thời điểm t
w_{t-1}	trọng số tại thời điểm t-1
ε	một hằng số dương rất nhỏ để tránh chia cho 0
δw_t	đạo hàm của trọng số tại thời điểm t
m_t	tổng các gradient tại thời điểm t.

Chương 1 Tổng Quan

1.1 TensorFlow

1.1.1 TensorFlow là gì?

Với sự bùng nổ của lĩnh vực Trí Tuệ Nhân Tạo – A.I. trong thập kỷ vừa qua, machine learning và deep learning rõ ràng cũng phát triển theo cùng. Và ở thời điểm hiện tại, TensorFlow chính là thư viện mã nguồn mở cho machine learning nổi tiếng nhất thế giới, được phát triển bởi các nhà nghiên cứu từ Google. Việc hỗ trợ mạnh mẽ các phép toán học để tính toán trong machine learning và deep learning đã giúp việc tiếp cận các bài toán trở nên đơn giản, nhanh chóng và tiện lợi hơn nhiều.

Hiện nay, TensorFlow cơ bản được xem như một trong những phương tiện trung gian giúp tính toán cho các số lượng có trong sản xuất và đồng thời trở thành một công cụ không thể thiếu trong Machine Learning, cùng với sự tương thích với nhiều ngôn ngữ lập trình như C++, Python, Java... Từ đó, phục vụ cho nhu cầu học tập cũng như nghiên cứu một cách dễ dàng hơn.

Tensor được hiểu là một loại cấu trúc dữ liệu được tập hợp trong một thư viện mà ở đó chính là TensorFlow. Trong số đó, cấu trúc của dữ liệu sẽ được miêu tả rồi điều chỉnh theo nhiều cách sao cho phù hợp nhất với các kiểu dữ liệu này. Và, cấu trúc dữ liệu này sẽ bao gồm 3 thuộc tính chính là: Bậc, chiều và loại dữ liệu.

Các hàm được dựng sẵn trong thư viện cho từng bài toán cho phép TensorFlow xây dựng được nhiều neural network. Nó còn cho phép tính toán song song trên nhiều máy tính khác nhau, thậm chí trên nhiều CPU, GPU trong cùng 1 máy hay tạo ra các dataflow graph – đồ thị luồng dữ liệu để dựng nên các model.

1.1.2 Cách TensorFlow hoạt động

Kiến trúc của Tensorflow cơ bản bao gồm 3 phần chính là:

- **Tiền xử lý dữ liệu**

- **Dựng Model**
- **Train và ước tính Model**

Khi TensorFlow hoạt động sẽ cho phép các lập trình viên có thể tạo ra dataflow graph, cũng như cấu trúc mô tả làm sao để cho dữ liệu có thể di chuyển qua 1 biểu đồ; hoặc di chuyển qua 1 seri mà các node đang xử lý. Mỗi một node có trong đồ thị thường đại diện cho 1 operation toán hoặc và mỗi kết nối thường hay edge giữa các node với nhau.

Từ đó, mỗi kết nối hoặc edge giữa các node được xem là mảng dữ liệu đa chiều. TensorFlow sẽ cung cấp tất cả mọi điều đến cho lập trình viên dựa theo phương thức của ngôn ngữ Python. Ngôn ngữ này sẽ cung cấp nhiều cách tiện lợi để ta có thể hiểu được nên làm thế nào cho các high-level abstractions có thể kết hợp được với nhau. Node cũng như tensor có trong TensorFlow chính là đối tượng của Python. Và, mọi ứng dụng Tensorflow bản thân chúng chính là một ứng dụng Python.

Các operation toán học thực sự thì thường không được thi hành bằng Python. Những thư viện biến đổi thường không có sẵn thông qua TensorFlow được viết bằng các binary C++ có hiệu suất cao. Ngoài ra, Python chỉ điều hướng cho các lưu lượng giữa các phần cũng như cung cấp các high-level abstraction lập trình để có thể nối chúng lại với nhau. Train thường phân tán dễ chạy hơn nhờ vào API mới và sự hỗ trợ cho TensorFlow Lite để cho phép việc triển khai các mô hình trên với nhiều nền tảng khác nhau.

1.1.3 Adam Optimizer

Adam là viết tắt của Adaptive Moment Estimation là một thuật toán cho kỹ thuật tối ưu hóa để giảm dần độ dốc. Adam có hiệu quả cao và được sử dụng rộng rãi trong các bài toán học máy. Phương pháp này thực sự hiệu quả khi làm việc với bài toán lớn liên quan đến nhiều dữ liệu hoặc tham số. Nó đòi hỏi ít bộ nhớ hơn và hiệu quả. Nó là sự kết hợp của thuật toán "gradient descent" và "RMSProp".

Gradient descent with momentum

Thuật toán gradient descent with momentum được sử dụng để tăng tốc thuật toán giảm độ dốc bằng cách xem xét "trung bình trọng số theo cấp số nhân" của độ dốc. Sử dụng

mức trung bình làm cho thuật toán hội tụ về phía cực tiểu với tốc độ nhanh hơn.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\frac{\delta L}{\delta w_t} \\
w_t &= w_{t-1} - \alpha m_t
\end{aligned}
\tag{1.1}
$$

trong đó :

- m_t là tổng các gradient tại thời điểm t. (với $m_0 = 0$)
- v_t là tổng các gradient tại thời điểm t-1.
- β là tham số trung bình.
- α là tốc độ học.
- δL là đạo hàm của hàm mất mát.
- w_t là trọng số tại thời điểm t.
- w_{t-1} là trọng số tại thời điểm t-1.

Root Mean Square Propagation (RMSP)

Root mean square prop hay RMSprop là một thuật toán học thích ứng cố gắng cải thiện AdaGrad. Thay vì lấy tổng tích lũy của bình phương độ dốc như trong AdaGrad, nó lấy "trung bình động hàm mũ".

$$
\begin{aligned}
v_t &= \beta v_{t-1} + (1-\beta)(\frac{\delta L}{\delta w_t})^2 \\
w_t &= w_{t-1} - \alpha_t \frac{\delta L}{\delta w_t}\frac{1}{\sqrt{v_t + \varepsilon}}
\end{aligned}
\tag{1.2}
$$

trong đó:

- v_t tổng bình phương của gradient trong quá khứ.
- β là tham số trung bình.

- α là tốc độ học.
- δL là đạo hàm của hàm mất mát.
- w_t là trọng số tại thời điểm t.
- w_{t-1} là trọng số tại thời điểm t-1.
- ε là một hằng số dương rất nhỏ để tránh chia cho 0.
- δw_t là đạo hàm của trọng số tại thời điểm t .

Adam Optimizer kế thừa các điểm mạnh hoặc thuộc tính tích cực của hai phương pháp trên và dựa trên chúng để tạo ra độ dốc giảm dần được tối ưu hóa hơn.

Công thức toán học của Adam Optimizer

Từ 2 công thức 1.1 và 1.2 ta có :

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\frac{\delta L}{\delta w_t} \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\frac{\delta L}{\delta w_t})^2 \\ \hat{m_t} &= \frac{m_t}{1-\beta_1^t}\hat{v_t} \\ w_t &= w_{t-1} - \alpha_t \frac{\hat{m_t}}{\sqrt{\hat{v_t}}+\varepsilon} \end{aligned} \tag{1.3}$$

trong đó:

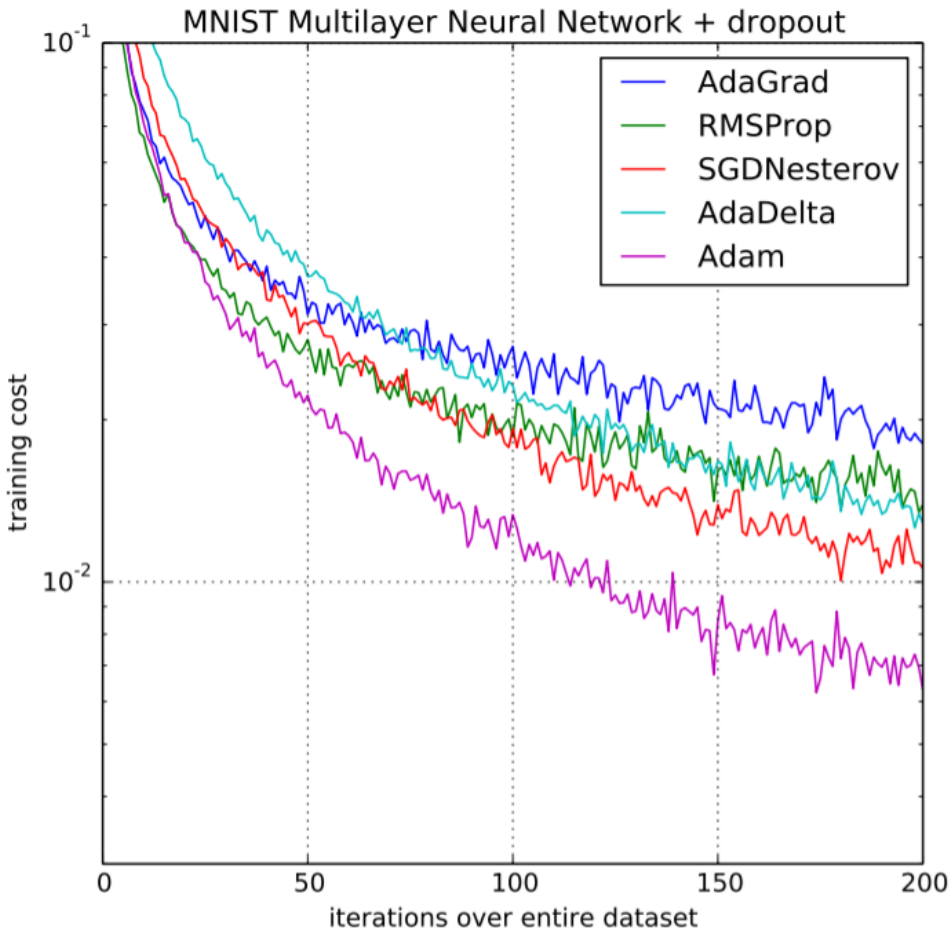
- β_1 và β_2 tốc độ phân rã trung bình của các gradient trong hai phương pháp trên.
- $\hat{m_t}$ và $\hat{v_t}$ là các tham số trọng số đã hiệu chỉnh sai lệch.

Từ đó ta thu được:

$$w_t = w_{t-1} - \alpha_t \frac{\hat{m_t}}{\sqrt{\hat{v_t}}+\varepsilon} \tag{1.4}$$

Dựa trên những điểm mạnh của các mô hình trước đó, trình tối ưu hóa Adam mang lại hiệu suất cao hơn nhiều so với mô hình được sử dụng trước đây và vượt trội hơn chúng

một cách đáng kể trong việc tạo ra độ dốc giảm dần được tối ưu hóa. Biểu đồ được hiển thị bên dưới mô tả rõ ràng cách trình tối ưu hóa của Adam vượt trội so với phần còn lại của trình tối ưu hóa bằng một biên độ đáng kể về chi phí đào tạo (thấp) và hiệu suất (cao).



HÌNH 1.1: So sánh các thuật toán tối ưu hóa
[1]

Trong TensorFlow thuật toán tối ưu hóa Adam được sử dụng với các thông số mặc định của các công thức trên là $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, và $\varepsilon =$ 1e-8. Các thông số này có thể dễ dàng tùy biến bằng cách gán các giá trị cho các thuộc tính tương ứng có sẵn trong Adam.

1.2 Neural Structured Learning

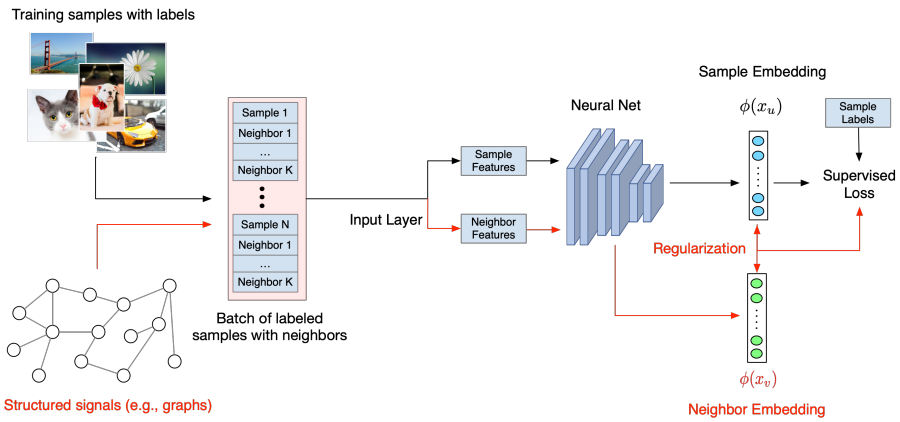
Neural Structured Learning (NSL) là một mô hình học tập mới để huấn luyện mạng thần kinh bằng cách tận dụng các tín hiệu có cấu trúc bên cạnh các đầu vào tính năng. Cấu

trúc có thể rõ ràng như được biểu thị bằng biểu đồ hoặc ẩn khi gây ra "nhiễu loạn đối nghich.

Các tín hiệu có cấu trúc thường được sử dụng để biểu thị các mối quan hệ hoặc sự giống nhau giữa các mẫu có thể được gắn nhãn hoặc không gắn nhãn. Do đó, việc tận dụng các tín hiệu này trong quá trình huấn luyện mạng thần kinh sẽ khai thác cả dữ liệu được gắn nhãn và không được gắn nhãn, điều này có thể cải thiện độ chính xác của mô hình, đặc biệt khi lượng dữ liệu được gắn nhãn tương đối nhỏ . Ngoài ra, các mô hình được đào tạo với các mẫu được tạo bằng cách thêm nhiễu loạn đối nghịch đã được chứng minh là hoạt động tốt đối với các dữ liệu không tốt được thiết kế để đánh lừa dự đoán hoặc phân loại của mô hình.

NSL khái quát hóa thành Neural Graph Learning cũng như Adversarial Learning. Khung NSL trong TensorFlow cung cấp các API và công cụ dễ sử dụng sau đây cho các nhà phát triển để đào tạo các mô hình với các tín hiệu có cấu trúc:

- **Keras APIs** để cho phép đào tạo với đồ thị (cấu trúc rõ ràng) và nhiễu đối phương (cấu trúc ngầm định).
- **TF ops and functions** cho phép đào tạo với cấu trúc khi sử dụng các API TensorFlow cấp thấp hơn.
- **Tools** để xây dựng đồ thị và xây dựng đầu vào đồ thị để đào tạo.

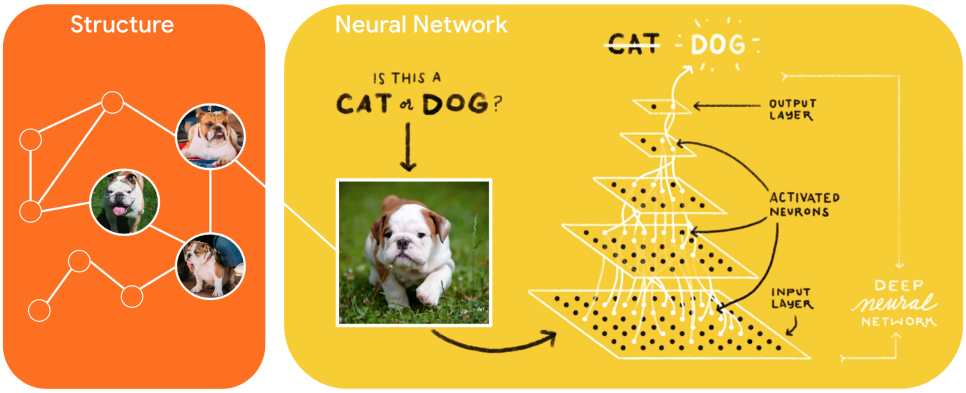


HÌNH 1.2: Neural Structured Learning [2]

Trong Neural Structured Learning (NSL), các tín hiệu có cấu trúc dù được xác định rõ ràng dưới dạng biểu đồ hay được học ngầm dưới dạng cách ví dụ đối nghịch được được sử dụng để thường xuyên đào tạo mạng neural, buộc mô hình phải học các dự đoán chính xác (bằng cách giảm thiểu mất mát có giám sát), đồng thời duy trì sự giống nhau giữa các đầu vào từ cùng một cấu trúc(bằng cách giảm thiểu tổn thất lân cận). Kỹ thuật này là chung và có thể được áp dụng trên các kiến trúc neural tùy ý, chẳng hạn như Feed-Forward NNs, CNNs, RNNs.

1.2.1 Neural Graph Learning

Training with natural graphs



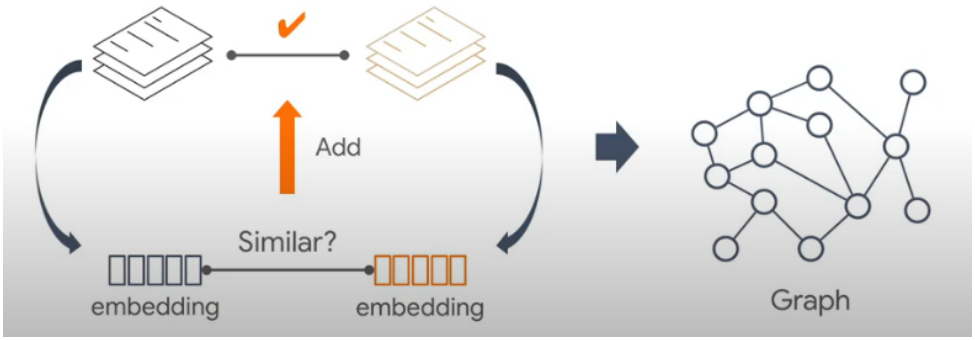
HÌNH 1.3: Training with natural graphs
[2]

Về cơ bản, đồ thị tự nhiên là một tập hợp các điểm dữ liệu có mối quan hệ mật thiết với nhau, bản chất của mối quan hệ này có thể thay đổi dựa trên bối cảnh. Mạng xã hội và Web là những ví dụ kinh điển mà chúng ta tương tác hàng ngày, ngoài những ví dụ này, chúng còn thường xuất hiện trong dữ liệu thường được sử dụng cho nhiều nhiệm vụ học máy. Ví dụ khi ta cố gắng nắm bắt hành vi của người dùng dựa trên tương tác của họ với dữ liệu, thì việc lập mô hình dữ liệu dưới dạng biểu đồ có thể hợp lý. Đối với xử lý ngôn ngữ tự nhiên, chúng ta có thể định nghĩa một biểu đồ văn bản trong đó các nút biểu thị các thực thể và các cạnh biểu thị mối quan hệ giữa các cặp thực thể.

Xem xét bài toán phân loại tài liệu. Ví dụ như những kỹ sư AI thường chỉ quan tâm đến các bài viết về học máy trên một chủ đề cụ thể như thị giác máy tính hoặc xử lý ngôn ngữ tự nhiên hoặc học tăng cường. Và thông thường, chúng ta có rất nhiều tài liệu hoặc

giấy tờ như thế để phân loại, nhưng rất ít trong số chúng có nhãn. Vì vậy cần phải làm cho dữ liệu được thiết lập thành một biểu đồ tự nhiên, điều này nghĩa là nếu một bài báo hoặc tài liệu được trích dẫn từ bài báo hoặc tài liệu khác thì chúng có thể có cùng nhãn. Việc sử dụng các thông tin quan hệ như vậy từ biểu đồ trích dẫn tận dụng được cả các mẫu được gắn nhãn cũng như không được gẫn nhãn. Điều này có thể bù đắp cho việc thiếu nhãn trong dữ liệu đào tạo. Vì vậy, việc xây dựng các đồ thị tự nhiên là rất cần thiết để giúp đào tạo các mô hình học máy một cách hiệu quả hơn.

Traning with synthesized graphs



HÌNH 1.4: Traning with synthesized graphs
[2]

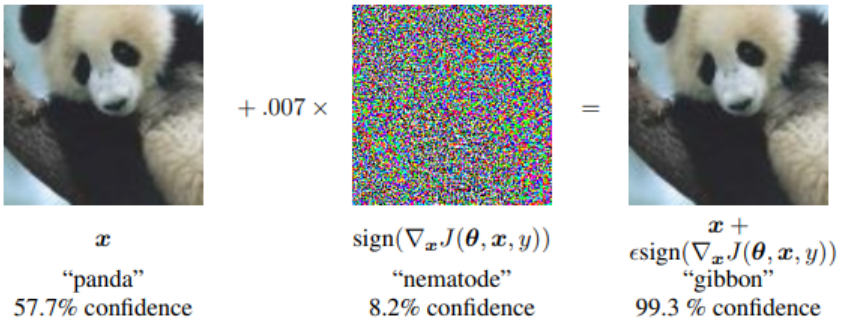
Mặc dù đồ thị tự nhiên là phổ biến, tuy nhiên có nhiều bài toán học máy với dữ liệu đầu vào không tạo thành đồ thị tự nhiên. Ví dụ như phân loại văn bản đơn giản hoặc phân loại hình ảnh thì dữ liệu đầu vào chỉ chứa hình ảnh hoặc văn bản thô, do đó ta không thể tạo ra biểu đồ tự nhiên. Vì vậy Training with synthesized graphs được sử dụng để giải quyết vấn đề này. Với ý tưởng chính là xây dựng hoặc tổng hợp một biểu đồ từ dữ liệu đầu vào. Trong Training with synthesized graphs chúng ta vẫn sự dụng sự giống nhau giữa các dữ liệu để xây dựng biểu đồ. Để xác định số liệu tương tự, thì cần phải chuyển đổi các văn bản thô hoặc hình ảnh thành các thành phần nhúng tương ứng hoặc các biểu diễn dày đặc. Khi chuyển đổi dữ liệu thành các thành phần nhúng tương ứng, thì ta có thể sử dụng các mô hình đào tạo trước đó hoặc một số hàm chẳng hạn như cos để so sánh mức độ tương ứng giữa của các cặp phần nhúng. Nếu điểm tương đồng lớn hơn một ngưỡng nhất định thì ta sẽ thêm vào một cạnh tương ứng vào biểu đồ kết quả. Việc lặp lại quy trình này sẽ bao phủ toàn bộ tập dữ liệu và sẽ tạo ra một biểu đồ. Và khi ta có biểu đồ thì việc sử dụng phương pháp học có cấu trúc trung tính rất đơn giản.

1.2.2 Adversarial Learning

Adversarial examples

Adversarial example là các mẫu được tạo ra với những thao tác tinh vi bằng cách thêm vào các nhiễu đối nghịch nhỏ mà mắt người không thể nào nhìn thấy được đã biến nó thành một hình ảnh hoàn toàn khác dưới con mắt kỹ thuật số của thuật toán machine learning.

Một số mô hình học máy bao gồm các mạng lưới thần kinh hiện đại nhất, dễ bị sai lệch trước những Adversarial examples. Những mô hình này cho ra kết quả sai các ví dụ chỉ khác một chút so với các ví dụ được phân loại chính xác từ trong tập dữ liệu. Ví dụ như khi chúng ta đưa ra đặc điểm của một con gấu trúc thì chúng ta sẽ tìm những đặc trưng của nó như mắt đen, đầu tròn, thân trắng... Nhưng đối với một mạng neural nhân tạo, miễn là khi dữ liệu được đưa vào chạy qua các layer đưa ra kết quả trả lời đúng thì nó sẽ tin hình ảnh của dữ liệu đó là con gấu trúc.



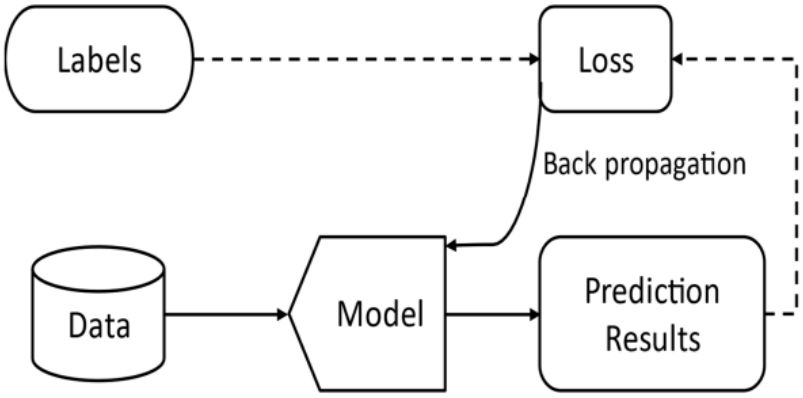
HÌNH 1.5: Adversarial examples
[3]

Từ hình 1.4 ta thấy, khi thêm vào 1 vector tín hiệu nhiễu rất rất bé mà mắt người không thể phân biệt được sự khác nhau giữa hai hình ảnh của con gấu trúc thì nó đã đánh lừa được mạng neural và khiến nó phán đoán sai và nó tin chắc rằng thứ mà nó đang nhìn thấy là con vượn (99.3%). Điều này cho thấy khi các tập dữ liệu không tốt(có nhiễu) thì có thể khiến mạng neural đưa ra phán đoán sai. Để giải quyết vấn đề này, Adversarial training được đề xuất để dạy các mạng neural không bị đánh lừa và phân loại sai.

[4] **Adversarial training**

Khái niệm về Adversarial training liên quan đến việc đào tạo bộ phân loại để khái quát hóa Adversarial examples cũng như mẫu sạch. Trong lược đồ đào tạo thông thường, được hiển thị trong Hình 1.5, dữ liệu đào tạo chuyển tiếp thông qua mô hình và tổn thất dự đoán được lan truyền ngược để cải thiện phân loại kết quả. Kết quả là, mô hình sẽ khái quát hóa việc phân phối dữ liệu huấn luyện để đưa ra một dự đoán chính xác về nhãn.

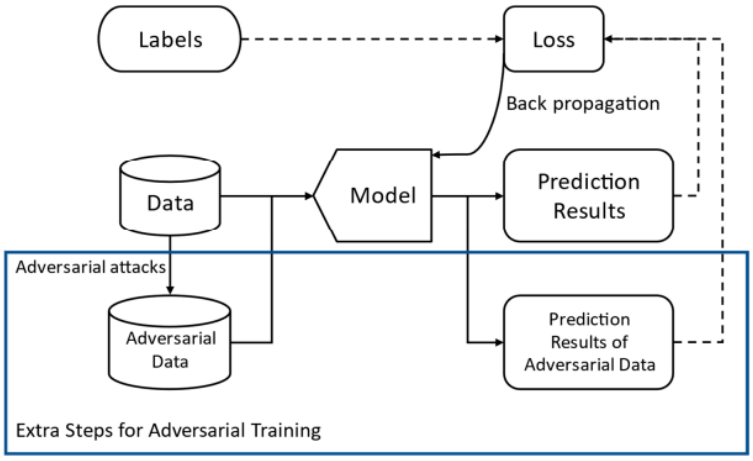
Để đào tạo mô hình tránh bị nhầm lẫn khi tập dữ liệu có nhiễu thì ta sẽ tạo ra các dữ liệu có nhiễu nhỏ từ tập dữ liệu đào tạo sau đó thêm các cạnh để kết nối dữ liệu vừa tạo ra với các mẫu của nó để xây dựng một cấu trúc linh hoạt sau đó, cấu trúc này có thể được sử dụng trong khung học tập của cấu trúc thần kinh. Trong khung học cấu trúc thần kinh, mạng thần kinh cố gắng học cách duy trì cấu trúc bằng cách giữ sự giống nhau giữa một mẫu và hàng xóm của nó. Vì vậy về cơ bản việc sử dụng tập dữ liệu sạch và dữ liệu đối nghịch của nó sẽ nói với mạng thần kinh rằng mẫu và mẫu đối nghịch của nó thực sự giống nhau. Vì vậy, hãy giữ sự giống nhau giữa chúng và đừng bị nhầm lẫn bởi các nhiễu.



HÌNH 1.6: Basic training
[5]

Adversarial training mở rộng các phương pháp đào tạo thông thường bằng cách bổ sung thêm bước vào quy trình đào tạo, như được minh họa trong Hình 1.6. Bằng cách này, mô hình có thể khái quát hóa cả dữ liệu sạch và dữ liệu đối nghịch được tạo ra bởi các phương thức được sử dụng trong Adversarial training để chống lại sự đánh lừa của mẫu đối nghịch.

Trong thư viện TensorFlow có sẵn các hàm chức năng để tạo ra các mẫu đối ngịch. Tương



HÌNH 1.7: Adversarial training
[5]

tự thì Keras API cũng có thể sử dụng để cho phép đào tạo từ đầu đến cuối một cách dễ dàng với Adversarial training như: AdversarialRegularization, AdvNeighborConfig, AdvRegConfig...

Chương 2 Adversarial regularization for image classification

2.1 Đặt vấn đề

Ở chương một chúng ta đã đi qua tổng quan về TensorFLow và Neural Structured Learning. Tuy nhiên để làm rõ hơn về hiệu quả của Neural Structured Learning, chúng ta cần thực nghiệm trên một bài toán thực tế. Vì vậy ta sẽ ứng dụng Neural Structured Learning và chính xác hơn là Adversarial Regularization for Image Classification.

Dựa trên tập dữ liệu có sẵn của thư viện TensorFlow MNIST, chúng ta sẽ thực hiện một bài toán phân loại ảnh chữ số viết tay. Tập dữ liệu bao gồm 70.000 ảnh chữ số viết tay được chia thành 60.000 ảnh để huấn luyện và 10.000 ảnh để kiểm tra. Mỗi ảnh có kích thước 28x28 pixel và được biểu diễn dưới dạng một mảng 2 chiều 28x28. Mỗi phần tử trong mảng này là một giá trị từ 0 đến 255 biểu diễn độ sáng của một pixel. Để thuận tiện cho việc huấn luyện, chúng ta sẽ chuyển đổi mỗi ảnh thành một mảng 1 chiều 784 phần tử. Để đơn giản hơn, chúng ta sẽ chia tập dữ liệu thành 2 tập huấn luyện và kiểm tra. Tập huấn luyện sẽ bao gồm 60.000 ảnh và tập kiểm tra sẽ bao gồm 10.000 ảnh. Mỗi ảnh sẽ được gán nhãn là một số từ 0 đến 9 tương ứng với chữ số viết tay.

Chúng ta có thể sử dụng các mạng neural thông thường để xây dựng mô hình phán đoán chữ số. Ý tưởng là sử dụng một vài mạng neural để tính toán các trọng số của mô hình.

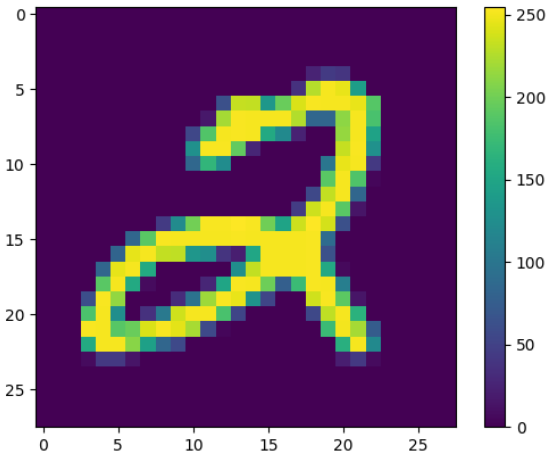
$$w_1X_1 + w_2X_2 + ... + w_nX_n = y \tag{2.1}$$

trong đó:

- $w_1, w_2, ..., w_n$ là các trọng số của mô hình.
- $X_1, X_2, ..., X_n$ là giá trị của các pixel.
- y là giá trị dự đoán của mô hình.

Mô hình sẽ tìm kiếm các trọng số phù hợp để đưa ra được đầu ra y chính xác đối với các hình ảnh đầu vào. Tuy nhiên nếu thực hiện như vậy, chúng ta sẽ không thể đạt được kết

quả tốt. Vì vậy chúng ta cần phải sử dụng một số kỹ thuật để cải thiện độ chính xác của mô hình. Bằng cách thêm các convolutional layer và pooling layer vào mô hình, chúng ta sẽ có thể giảm số lượng tham số của mô hình và cải thiện độ chính xác của mô hình.



HÌNH 2.1: Ảnh chữ số viết tay được lấy từ tập dữ liệu MNIST

Chúng ta sẽ thực hiện xây dựng và đào tạo một mạng neural đơn giản và một mạng neural sử dụng Adversarial Regularization. Sau đó chúng ta sẽ so sánh kết quả dự đoán của 2 mạng neural này để thấy được hiệu quả của Neural Structured Learning. Sau khi đào tạo và dự đoán thử trên tập dữ liệu có sẵn của TensorFlow, thì ta sẽ thử viết tay một vài chữ số và để cho mạng neural dự đoán xem nó có thể dự đoán chính xác không.

Với những thư viện đồ sộ và khả năng làm việc với các ma trận tốt, Python là ngôn ngữ được lựa chọn để thực hiện ý tưởng trên. Cùng với đó là các thư viện như numpy, matplotlib và đặc biệt là TensorFlow với API keras và Neural structed learning.

2.2 Thực nghiệm

2.2.1 Cài đặt môi trường

Bước đầu tiên cần cài đặt biến môi trường và các thư viện cần thiết. Cài đặt phiên bản python3 và editor hoặc IDE để viết mã như Pycharm hoặc VScode. Sau đó, trên command line chạy dòng lệnh ’pip install -r Setup.txt’ để tiến hành cài đặt các thư viện cần thiết. File "Setup.txt" là file chứa các thư viện cần cài đặt.

2.2.2 Viết mã

Import thư viện

```
import tensorflow as tf
import numpy as np
import neural_structured_learning as nsl
import matplotlib.pyplot as plt
import tensorflow_datasets as tfds
```

Khai báo các tham số

```
input_shape = [28, 28, 1]
num_classes = 10
conv_filters = [32, 64, 64]
kernel_size = (3, 3)
pool_size = (2, 2)
num_fc_units = [64]
batch_size = 32
epochs = 5
adv_multiplier = 0.1
adv_step_size = 0.2
adv_grad_norm = 'infinity'
```

Khai báo các tham số cần thiết cho việc xây dựng mạng neural và đào tạo. Các tham số được khai báo như sau:

Đầu vào và đầu ra:

- **input_shape**: Hình dạng của tensor đầu vào. Mỗi hình ảnh có kích thước 28x28pixel và có 1 kênh màu.
- **num_classes**: Số lượng lớp đầu ra. Trong bài toán này là 10 lớp từ 0 đến 9.

Các tham số để xây dựng mô hình mạng neural:

- **conv_filters**: Số lượng lọc cho mỗi lớp tích chập. Mỗi lớp tích chập có số bộ lọc bao gồm 32,64,64.
- **kernel_size**: Kích thước của kernel tích chập 2D. Mỗi kernel có kích thước 3x3.
- **pool_size**: Các yếu tố để thu nhỏ hình ảnh trong mỗi lớp tổng tối đa. Mỗi pooling có kích thước 2x2.
- **num_fc_units**: Số lượng đơn vị ẩn của mỗi lớp fully connected. Mỗi lớp fully connected có 64 đơn vị ẩn. Nghĩa là chiều rộng của mỗi lớp được kết nối đầy đủ.

Các tham số để đào tạo mô hình:

- **batch_size**: Kích thước batch. Mỗi batch có 32 hình ảnh. Là số lượng bức ảnh được đưa vào mạng neural mỗi lần.
- **epochs**: Số lần huấn luyện. Mỗi lần sẽ duyệt qua tất cả các hình ảnh trong tập dữ liệu.
- **adv_multiplier**: Trọng số tổn thất do "nhiễu loạn đối nghịch" gây ra trong mục tiêu huấn luyện, so với tổn thất của mô hình gốc(tổn thất được gắn nhãn).
- **adv_step_size**: Độ lớn của "nhiễu loạn đối nghịch".
- **adv_grad_norm**: Định mức để đo mức độ nhiễu loạn của đối nghịch.

Tải dữ liệu MNIST

```
    data_train, data_test = tfds.load('mnist', split=['train', 'test'])
```

Bộ dữ liệu MNIST chứa hình ảnh thang độ xám của các chữ số viết tay từ 0 dến 9. Mỗi hình ảnh hình ảnh hiển thị một chữ số ở độ phân giải thấp 28x28 pixel. Nhiệm vụ liên quan là phân loại hình ảnh thành 10 loại, mỗi loại tương ứng với một chữ số từ 0 đến 9.

Bộ dữ liệu đã có sẵn trong thư viện TensorFlow Datasets (TFDS) và có thể được tải xuống và xây dựng tệp tf.data.Dataset. Tập dữ liệu đã tải bao gồm hai tập con :

- **train**: Tập dữ liệu huấn luyện với 60000 ảnh.
- **test**: Tập dữ liệu kiểm tra với 10000 ảnh.

Các dữ liệu trong cả hai tập đều được lưu trữ dưới dạng dictionary với hai khóa:

- **image**: Là các mảng pixel 28x28 chứa các giá trị từ 0 đến 255.
- **label**: Nhãn thật của hình ảnh, là một số nguyên từ 0 đến 9.

```
    def normalize(features):
        features['image'] = tf.cast(features['image'], dtype=tf.float32) / 255.0
        return features

    def convert_to_tuples(features):
        return features['image'], features['label']
```

```
    def convert_to_dictionaries(image, label):
        return {'image': image, 'label': label}

    data_train = data_train.map(normalize).shuffle(10000).batch(batch_size).map(
        convert_to_tuples)
    data_test = data_test.map(normalize).batch(batch_size).map(convert_to_tuples)
```

Để làm cho mô hình ổn định về số lượng, chúng ta chuẩn hóa các giá trị pixel từ [0,255] thành [0, 1] bằng cách ánh xạ tập dữ liệu qua hàm normalize(). Sau khi xáo trộn tập huấn luyện và chia theo nhóm, chúng ta chuyển đổi các ví dụ thành các bộ dữ liệu đặc trưng (image, label) để huấn luyện mô hình cơ sở. Chúng ta cũng cung cấp một hàm chức năng để chuyển đổi từ bộ sang từ điển để sử dụng sau này.

Xây dựng và đào tạo mô hình cơ bản

Mô hình cơ bản sẽ là một mạng thần kinh bao gồm 3 lớp tích chập, theo sau là 2 lớp được kết nối đầy đủ (với các tham số như được định nghĩa trong phần khai báo tham số). Trong đó các lớp tích chập sẽ có hàm kích hoạt activation là 'relu' còn hai lớp theo sau thì một lớp có hàm kích hoạt là 'relu'và lớp output sẽ là 'softmax'. Lớp output sử dụng activation 'softmax' để đánh giá xác suất phân loại dữ liệu đầu vào và tính toán trọng số cho dữ liệu. Chúng ta tạo hàm build_model() để thuận tiện cho việc tái sử dụng khi khởi tạo mô hình khác. Ở đây chúng ta xác định nó bằng các hàm API của Keras.

```
    def build_model():
        inputs = tf.keras.Input(shape=input_shape, dtype=tf.float32, name='image')
        x = inputs
        for i, num_filters in enumerate(conv_filters):
            x = tf.keras.layers.Conv2D(num_filters, kernel_size, activation='relu')(x)
            if i < len(conv_filters) - 1:
                x = tf.keras.layers.MaxPooling2D(pool_size)(x)
        x = tf.keras.layers.Flatten()(x)
        for num_units in num_fc_units:
            x = tf.keras.layers.Dense(num_units, activation='relu')(x)
        outputs = tf.keras.layers.Dense(num_classes, activation='softmax')(x)
        return tf.keras.Model(inputs=inputs, outputs=outputs)

    model_base = build_model()
```

Sau khi xây dựng model thì chúng ta sẽ tiến hành đào tạo và đánh giá model. Với hàm tối ưu hóa 'Adam' như đã giới thiệu ở phần trước, hàm mất mát 'SparseCategoricalCrossentropy' và đánh giá 'SparseCategoricalAccuracy'. Sau khi đào tạo xong chúng ta sẽ đánh giá model bằng hàm evaluate() và in ra độ chính xác của model.

```
model_base.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])
```

```
model_base.summary()

model_base_history = model_base.fit(data_train, epochs=epochs)

results = model_base.evaluate(data_test)
named_results = dict(zip(model_base.metrics_names, results))
print('\naccuracy:', named_results['sparse_categorical_accuracy'])
```

```
_________________________________________________________________
Epoch 1/5
2022-12-16 16:50:01.458506: I tensorflow/stream_executor/cuda/cuda_dnn.cc:384] Loaded cuDNN version 8700
1875/1875 [==============================] - 17s 4ms/step - loss: 0.1500 - sparse_categorical_accuracy: 0.9530
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0461 - sparse_categorical_accuracy: 0.9858
Epoch 3/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0332 - sparse_categorical_accuracy: 0.9900
Epoch 4/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0270 - sparse_categorical_accuracy: 0.9917
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0201 - sparse_categorical_accuracy: 0.9936
313/313 [==============================] - 1s 3ms/step - loss: 0.0289 - sparse_categorical_accuracy: 0.9915

accuracy: 0.9915000200271606
```

HÌNH 2.2: Accuracy

Chúng ta có thể thấy rằng mô hình cơ bản đạt được độ chính xác lên đến 99,15% trên bộ thử nghiệm. Từ đó cho thấy dữ liệu đào tạo và mô hình đào tạo là khá tốt. Tuy nhiên cần phải sử dụng các tập dữ liệu khác để đánh giá mô hình một cách chính xác hơn.

Xây dựng và đào tạo mô hình Adversarial-regularized

Chúng ta sẽ xây dựng mô hình mạng neural đối nghịch bằng cách kết hợp adversarial training vào mô hình keras và sử dụng khung Neural Structured Learning (NSL). Mô hình cơ sở được bao bọc để tạo một mô hình mới **tf.Keras.Model**, có mục tiêu đào tạo bao gồm adversarial regularization.

Đầu tiên, chúng tôi tạo một đối tượng cấu hình với tất cả cáctham số có liên quan bằng cách sử dụng hàm chức năng **nsl.configs.make_adv_reg_config** có sẵn trong thư viện NSL.

```
    adv_config = nsl.configs.make_adv_reg_config(multiplier=adv_multiplier,
        adv_step_size=adv_step_size,
        adv_grad_norm=adv_grad_norm)
```

Ở đây chúng ta sẽ tạo một mô hình cơ bản mới **base_adv_model** để mô hình hiện tại (**model_base**) dùng để so sánh sau này.

Bây giờ chúng ta có thể bọc một mô hình cơ sở bằng **AdversarialRegularization** bằng cách sử dụng hàm chức năng có sẵn **nsl.keras.AdversarialRegularization**.

```
base_adv_model = build_model()
model_adv = nsl.keras.AdversarialRegularization(
    base_adv_model,
    label_keys=['label'],
    adv_config=adv_config)

data_train_adv = data_train.map(convert_to_dictionaries)
data_test_adv = data_test.map(convert_to_dictionaries)
```

Trả về **adv_model** là một đối tượng có kiểu **tf.keras.Model**, có mục tiêu đào tạo bao gồm phần regularization và mất mát của adveresarial. Để tính toán tổn thất đó, mô hình phải có quyền truy cập vào thông tin nhãn (feature label), ngoài đầu vào thông thường (feature image). Vì lý do này, chúng ta sẽ chuyển đổi các mẫu trong tập dữ liệu trở lại kiểu dictionary. Và chúng ta cho mô hình biết tính năng nào chứa thông tin nhãn thông qua tham số **label_keys**.

Tiếp theo, chúng ta sẽ biên dịch, đào tạo và đánh giá mô hình chính quy hóa đối thủ. Với các hàm tối ưu, mất mát, và đánh giá độ chính xác tương tự như khi đào tạo mô hình cơ bản. Có thể có các cảnh báo như "Output missing from loss dictionary", điều này không sao cả vì **adv_model** không dựa vào triển khai cơ sở để tính tổng tổn thất.

```
model_adv.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

model_adv_history = model_adv.fit(data_train_adv, epochs=epochs)

adv_results = model_adv.evaluate(data_test_adv)
named_adv_results = dict(zip(model_adv.metrics_names, adv_results))
print('\naccuracy:', named_adv_results['sparse_categorical_accuracy'])
```

```
305/313 [============================>.] - ETA: 0s - loss: 0.0590 - sparse_categorical_accuracy: 0.9933 - sparse_categorical_crossentropy: 0.0204 - scaled_adv
313/313 [==============================] - ETA: 0s - loss: 0.0595 - sparse_categorical_accuracy: 0.9931 - sparse_categorical_crossentropy: 0.0207 - scaled_adv
313/313 [==============================] - 2s 5ms/step - loss: 0.0595 - sparse_categorical_accuracy: 0.9931 - sparse_categorical_crossentropy: 0.0207 - scaled
_adversarial_loss: 0.0388

accuracy: 0.9930999875068665
```

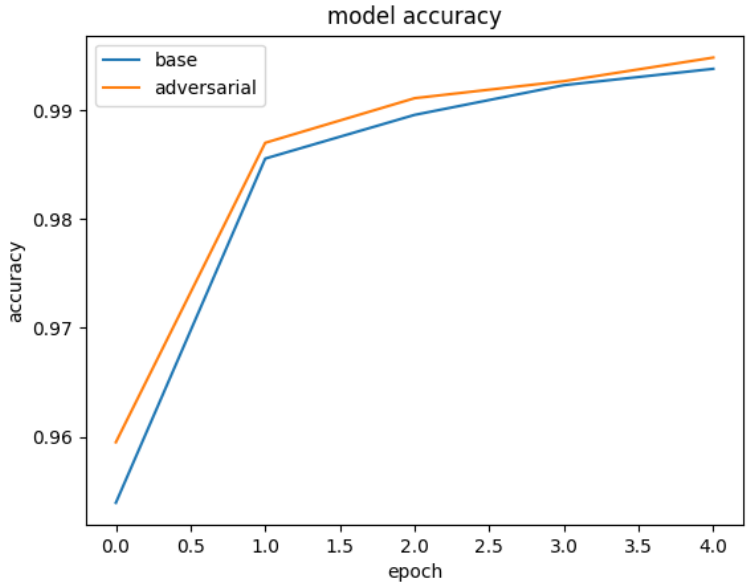
HÌNH 2.3: Độ chính xác của mô hình Adversarial-regularized.

Chúng ta có thể thấy rằng mô hình Adversarial-regularized cũng hoạt động rất tốt (độ chính xác 99,31%) trên tập thử nghiệm. Tuy độ chính xác có nhỉnh hơn một xíu so với mô hình cơ bản nhưng không quá nhiều.

Chúng ta sẽ đưa ra đồ thị sự thay đổi của độ chính xác qua mỗi lần học tập trên tập dữ liệu thử nghiệm của cả hai mô hình để dễ dàng quan sát hơn.

```
plt.plot(model_base_history.history['sparse_categorical_accuracy'])
plt.plot(model_adv_history.history['sparse_categorical_accuracy'])
plt.title('model accuracy')
```

```
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['base', 'adversarial'], loc='upper left')
    plt.show()
```



HÌNH 2.4: Đồ thị độ chính xác của hai mô hình qua từng lần học.

Từ đồ thị, ta thấy đường cong độ chính xác của mô hình Adversarial-regularized và mô hình cơ bản có xu hướng tương tự nhau. Mô hình adversarial-regularized có nhỉnh hơn một chút nhưng chênh lệch là không quá nhiều. Điều này là do cả hai mô hình đều sử dụng các hàm tối ưu, mất mát và đánh giá tương tự nhau.

So sánh độ chính xác của hai mô hình dưới các mẫu "nhiễu loạn đối nghịch"

Bây giờ chúng ta so sánh mô hình cơ sở và mô hình Adversarial-regularized về khả năng phán đoán chính xác dưới sự nhiễu loạn của các mẫu đối nghịch. Chúng ta sẽ sử dụng hàm chức năng **AdversarialRegularization.perturb_on_batch()** để tạo các mẫu "nhiễu loạn đối nghịch". Để có thể sánh được độ chính xác của hai mô hình, ta sẽ phải bọc mô hình cơ bản bằng **AdversarialRegularization**. Các biến mà mô hình cơ bản đã học trước đó sẽ không bị thay đổi miễn là chúng ta không gọi hàm đào tạo **fit()**.

```
    ref_model = nsl.keras.AdversarialRegularization(
        model_base,
        label_keys=['label'],
        adv_config=adv_config)


    ref_model.compile(
        optimizer=tf.keras.optimizers.Adam(),
```

```
        loss=tf.keras.losses.SparseCategoricalCrossentropy(),
        metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])
```

Tiếp theo chúng ta sẽ đánh giá và so sánh khả năng phán đoán của hai mô hình trên các mẫu đối nghịch. Ở đây chúng ta lấy **adv_model.base_model** để có cùng định dạng đầu vào (không yêu cầu thông tin nhãn) làm mô hình cơ sở. Các biến đã học trong **adv_model.base_model** giống như các biến trong **adv_model**.

```
models_to_evaluate = {
    'base': model_base,
    'adversarial': model_adv.base_model,
}

metrics = {
    name: tf.keras.metrics.SparseCategoricalAccuracy()
    for name in models_to_evaluate.keys()
}
```

Tiếp theo chúng ta sẽ tạo các "ví dụ nhiễu loạn" và đánh giá các mô hình với chúng. Chúng ta lưu các hình ảnh, nhãn và dự đoán bị nhiễu vào 3 mảng perturbed_imgs,labels,predictions để trực quan hóa trong phần sau để có cách nhìn rõ hơn về khả năng phán đoán của hai mô hình. Các mẫu nhiễu loạn được tạo ra từ tập dữ liệu test và được chuẩn hóa để chúng có cung kích thước với các mẫu thông thường bằng hàm chức năng **tf.clip_by_value**.

```
perturbed_imgs,labels,predictions = [],[],[]

for batch in data_test_adv:
    perturbed_batch = ref_model.perturb_on_batch(batch)
    perturbed_batch['image'] = tf.clip_by_value(perturbed_batch['image'], 0, 1)

    y_true = perturbed_batch.pop('label')
    perturbed_imgs.append(perturbed_batch['image'].numpy())
    labels.append(y_true.numpy())
    predictions.append({})

    for name, model in models_to_evaluate.items():
        y_pred = model(perturbed_batch)
        metrics[name].update_state(y_true, y_pred)
        predictions[-1][name] = tf.argmax(y_pred, axis=-1).numpy()

for name, metric in metrics.items():
    print(f'{name} accuracy: {metric.result().numpy()}')
```

```
base accuracy: 0.5770000219345093
adversarial accuracy: 0.9620000123977661
```

HÌNH 2.5: Độ chính xác của hai mô hình trên mẫu "nhiễu loạn đối nghịch".

Từ hình 2.5 Chúng ta có thể thấy rằng độ chính xác của mô hình cơ bản giảm đáng kể (từ 99,15% xuống còn khoảng 57,7%) khi đầu vào bị nhiễu. Mặt khác, độ chính xác của

mô hình Adversarial-regularized chỉ giảm một chút (từ 99,31% xuống 96,2%). Điều này chứng tỏ tính hiệu quả của việc học đối thủ trong việc cải thiện độ chính của mô hình và khả năng hoạt động tốt của mô hình trên những tập dữ liệu nhiều nhiễu.

Trực quan hóa các mẫu "nhiễu loạn đối nghịch"

Chúng ta sẽ chọn ra một batch ngẫu nhiên trong mảng perturbed_imgs đã lưu ở trên để trực quan hóa, ở đây batch được chọn là batch thứ 5. Sử dụng thư viện **matplotlib** để hiển thị các hình ảnh và hiển thị các dự đoán của hai mô hình và kiểm tra xem mô hình đoán có đúng không so với nhãn thật.

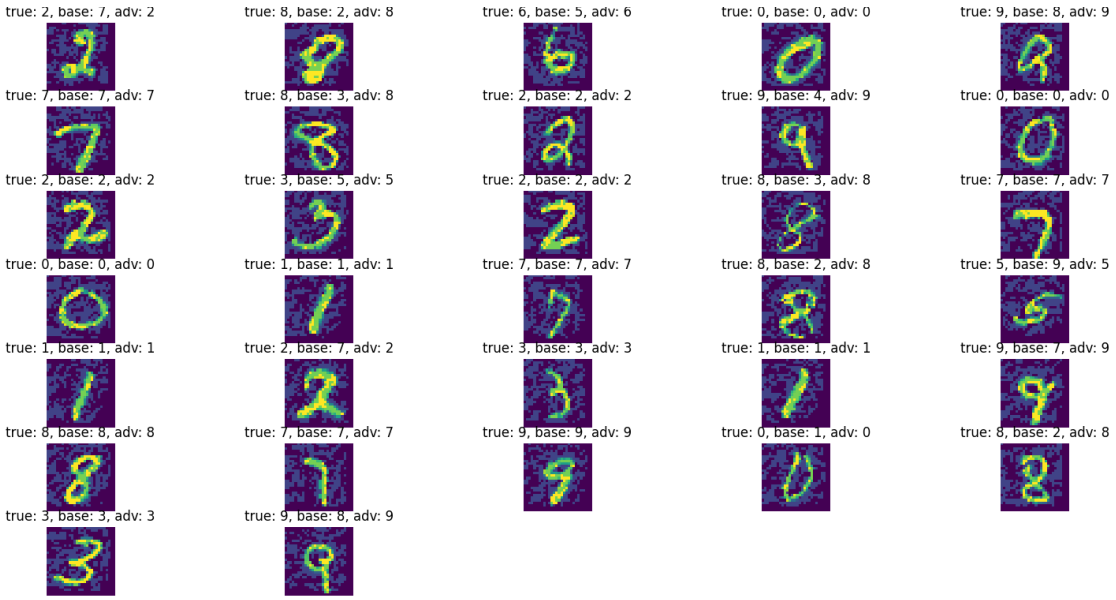
```
    batch_index = 5

    batch_images = perturbed_imgs[batch_index]
    batch_labels = labels[batch_index]
    batch_predictions = predictions[batch_index]

    n_columns = 5
    n_rows = (batch_size + n_columns - 1) // n_columns

    print('acc in batch %d: ' % batch_index, end='')
    for name,pred in batch_predictions.items():
        print('%s model: %d / %d' % (name, np.sum(batch_labels == pred), batch_size))

    plt.figure(figsize=(n_columns * 2, n_rows * 2))
    for i,(img,y) in enumerate(zip(batch_images, batch_labels)):
        y_base = batch_predictions['base'][i]
        y_adv = batch_predictions['adversarial'][i]
        plt.subplot(n_rows, n_columns, i + 1)
        plt.title('true: %d, base: %d, adv: %d' % (y, y_base, y_adv))
        plt.imshow(img)
        plt.axis('off')

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

Từ hình 2.6 ta thấy, các nhiễu loạn nhỏ được thêm vào những bức ảnh chữ số viết tay dưới mắt người vẫn có thể nhận biết được tuy nhiên nó đã đánh lừa được mô hình cơ bản phán đoán sai.

Từ hình 2.7 ta thấy rằng, khả năng phán đoán của mô hình cơ bản(base model) khá thấp(21/32 mẫu). Trong khi đó mô hình Adversarial-regularized phán đoán rất tốt (32/32 mẫu).

Lưu mô hình vừa đào tạo

Sau khi đào tạo chúng ta sẽ lưu lại các mô hình để thử cho các mô hình phán đoán một số hình ảnh viết tay không phải thuộc tập dữ liệu của tensorflow_datasets. Mô hình được lưu lại có thể dễ dàng mang ra sử dụng hoặc tiếp tục đào tạo tùy mục đích và no được



HÌNH 2.6: Một số mẫu "nhiễu loạn đối nghịch" và các phán đoán của hai mô hình.

```
acc in batch 5: base model: 16 / 32
adversarial model: 32 / 32
```

HÌNH 2.7: Số dự đoán chính xác của hai mô hình trên mẫu "nhiễu loạn đối nghịch" trong batch 5.

lưu dưới định dạng '.h5' với sự hỗ trợ của thư viện **h5py**. Lưu ý rằng, sau khi lưu mô hình thì các biến mà mô hình đã học tập được sẽ được giữ nguyên.

```
1    model_adv.save('Model/advs_model.h5')
2    model_base.save('Model/base_model.h5')
```

[6]

2.2.3 Kiểm tra mô hình với một số ảnh viết tay

Ở phần trước, chúng ta đã xây dựng, đào tạo và kiểm tra mô hình trên tập dữ liệu của tensorflow_datasets. Tuy nhiên, chúng ta để biết được thực sự mô hình có hoạt động tốt với các bức ảnh ngẫu nhiên hay không, ta sẽ viết tay một số chữ số và cho mô hình dự đoán.

Ở phần này, chúng ta sẽ sử dụng thư viện xử lý ảnh OpenCV để đọc và tiền xử lý các bức ảnh chụp. Do đầu vào của mô hình là một bức ảnh 28x28 nên chúng ta cần phải resize

ảnh về kích thước này. Sau đó, chúng ta cũng cần chuyển ảnh về dạng mảng numpy với giá trị trong khoảng 0-1 để mô hình ổn định hơn và tiến hành cho mô hình dự đoán.

Import thư viện

Các thư viện này đã được yêu cầu cài đặt trong file Setup.txt nên chỉ cần import vào là có thể sử dụng.

```
import cv2
import numpy as np
import tensorflow as tf
```

Đọc ảnh và tiền xử lý ảnh

Các bức ảnh chụp chữ số viết tay sẽ được lưu lại trong thư mục Image/. Chúng ta sẽ đọc các bức ảnh này và tiền xử lý để cho mô hình dự đoán.

```
image = cv2.imread("Image/test2.jpg")
copy = image.copy()

im_gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
im_blur = cv2.GaussianBlur(im_gray,(5,5),0)
im,thre = cv2.threshold(im_blur,90,255,cv2.THRESH_BINARY_INV)
contours,hierachy = cv2.findContours(thre,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE
    )
rects = [cv2.boundingRect(cnt) for cnt in contours]
```

Chúng ta sẽ đọc các bức ảnh này bằng hàm **imread()** có sẵn trong opencv sau đó sẽ copy ảnh vừa đọc và gán vào biến copy để sử dụng cho sau này. Tiếp theo ta cần chuyển ảnh về dưới dạng thang xám để dễ dàng xử lý hơn bằng hàm **cvtColor()**. Sau đó, chúng ta sẽ làm mờ ảnh bằng hàm **GaussianBlur()** để loại bỏ các nhiễu trong ảnh và tăng khả năng phán đoán chính xác của các mô hình. Cuối cùng ta sẽ chuyển ảnh về dưới dạng nhị phân để có thể đưa vào mô hình dự đoán.

Do ảnh chụp có thể có nhiều chữ số viết tay nên chúng ta cần phải tách các chữ số ra. Để làm được điều này, chúng ta sẽ sử dụng hàm **findContours()** để tìm vị trí chuỗi số và các đường viền của các chữ số trong ảnh. Sau đó, chúng ta sẽ sử dụng hàm **boundingRect()** để tao các hình chữ nhật bao quanh các chữ số trong ảnh.

Gọi mô hình và dự đoán

Sau khi đã chuẩn bị các dữ liệu ảnh cần thiết, chúng ta sẽ gọi mô hình đã được train trước đó và dự đoán kết quả.

```
adv_model = tf.keras.models.load_model("Model/advs_model.h5")
base_model = tf.keras.models.load_model("Model/base_model.h5")
```

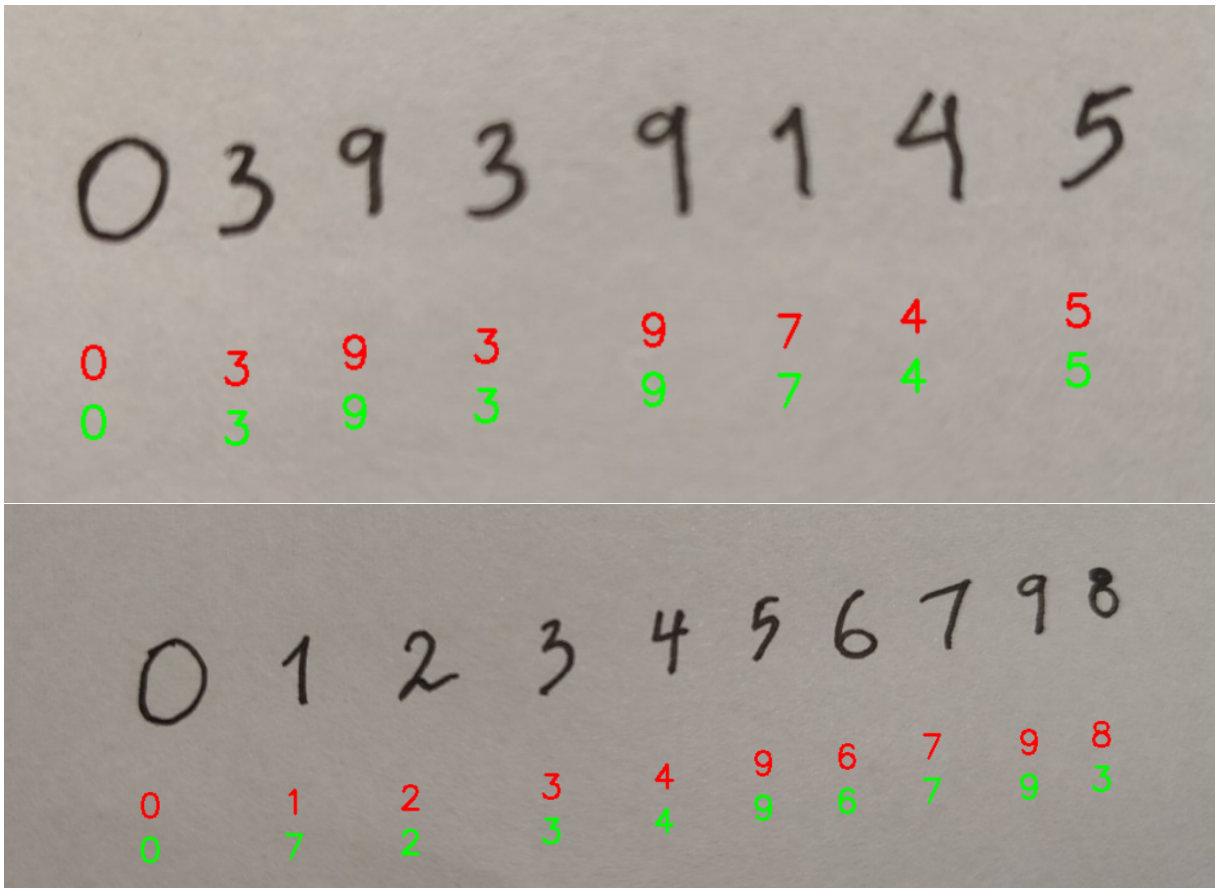
```
    for i in contours:
        (x,y,w,h) = cv2.boundingRect(i)
        cv2.rectangle(image,(x,y),(x+w,y+h),(0,255,0),3)
        subImage = thre[y:y+h,x:x+w]
        subImage = np.pad(subImage,(20,20),'constant',constant_values=(0,0))
        subImage = cv2.resize(subImage, (28, 28), interpolation=cv2.INTER_AREA)
        subImage = cv2.dilate(subImage, (3, 3))

        img = subImage.reshape(1,28,28,1)
        img = img/255.0
        img = img.astype(np.float32)

        adv_pred = adv_model.predict(img)
        adv_pred = np.argmax(adv_pred)
        cv2.putText(copy,str(int(adv_pred)),(x,y+160),0,1,(0,0,255),2)

        base_pred = base_model.predict(img)
        base_pred = np.argmax(base_pred)
        cv2.putText(copy,str(int(base_pred)),(x,y+200),0,1,(0,255,0),2)

    cv2.imshow("image",copy)
    cv2.waitKey(0)
```

Sử dụng hàm **load_model()** để gọi mô hình đã được train trước đó. Sau đó, chúng ta sẽ sử dụng vòng lặp để duyệt qua các chữ số trong ảnh. Lúc này chúng ta vẫn chỉ có các ảnh thô dạng nhị phân của từng chữ số nên ta cần xử lý chúng, sử dụng hàm **resize** để chuyển ảnh về kích thước 28x28. Tuy nhiên, đầu vào của các mô hình là ảnh có kích thước 28x28x1 nên ta cần chuyển ảnh về dạng một chiều. Để làm được điều này, chúng ta sẽ sử dụng hàm **reshape()**. Sau đó, chúng ta sẽ chuẩn hóa ảnh về dạng 0-1 bằng cách chia cho 255.0. Cuối cùng, chúng ta sẽ sử dụng hàm **predict()** để dự đoán chữ số, và dự đoán đó sẽ được hiển thị lên bức ảnh copy từ trước. Dự đoán của mô hình cơ bản sẽ có màu xanh lá cây, còn dự đoán của mô hình Adversarial-regularized sẽ có màu đỏ.

Kết quả dự đoán

Từ hình 2.8 ta thấy. Ở bức ảnh phía trên, khi các chữ số được viết khá rõ ràng thì cả hai mô hình đều đưa ra dự đoán sai 1 chữ số. Tuy nhiên ở hình dưới, khi các chữ số bị viết xấu hơn, các nét viết nguệch ngoạc thì mô hình Adversarial-regularized chỉ dự đoán sai 1/10 chữ số, trong khi mô hình cơ bản dự đoán sai 3/10 chữ số. Từ đó ta thấy rằng, mô hình Adversarial-regularized có độ chính xác cao hơn và hoạt động tốt hơn mô hình cơ bản khi dữ liệu bị nhiễu. Và đây là một trong những ưu điểm của mô hình Adversarial-regularized.

Link source code: https://github.com/Danhnt0/Tieu_Luan



HÌNH 2.8: Kết quả dự đoán

Chương 3 Kết luận

Qua các chương trên chúng ta đã có cái nhìn tổng quan về TensorFlow, Neural Structured Learning. Chúng ta đã cài đặt và sử dụng Neural Structured Learning để xử lý bài toán phân loại chữ số viết tay. Chúng ta biết rằng Neural Structured Learning được khái quát hóa bằng hai phần chính là Neural Graph Learning và Adversarial Learning tuy nhiên ở chương 2 chúng ta mới chỉ sử dụng Adversarial Learning để thử nghiệm, quan sát và đánh giá nó. Và ở mục 2.2 chúng ta đã minh chứng rằng Neural Structured Learning có thể giúp cải thiện độ chính xác của mô hình và giúp các mô hình hoạt động tốt hơn trên những tập dữ liệu kém.

Ở mục 1.1.2 chúng ta thấy rằng chỉ cần thêm một nhiễu rất nhỏ mà mắt thường không thể phát hiện được, ngay lập tức đã đánh lừa được mô hình cơ bản phán đoán sai. Điều này mang lại nhiều nguy cơ cho các hệ thống sử dụng mạng neural để phát hiện, phân loại hình ảnh như camera an ninh hay xe tự lái vì chỉ cần các bức ảnh mà camera thu thập được không đủ tốt hoặc có nhiễu thì sẽ gây ra sai lầm cho các phán đoán cho hệ thống. Đối với các ứng dụng như nhận dạng hay phân loại đồ vật thì các sai sót xảy ra ít gây nghiêm trọng hơn, nhưng những ứng dụng như xe tự lái, thu thập và phân tích hình ảnh theo thời gian thực, một phán đoán sai như không phát hiện người đi đường thì có thể gây ra hậu quả lớn. Vì vậy việc tìm ra các cách để cải thiện độ chính xác của mô hình trên các tập dữ liệu kém tương tự như Adversarial Learning là rất cần thiết.

https://github.com/Danhnt0/Latex_Tieu_Luan

Tài liệu tham khảo

[1] *Adam algorithm*. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning. Accessed on 2022-12-9. 2021.

[2] *Neural Structured Learning: Training with Structured Signals*. https://www.tensorflow.org/neural_structured_learning. Accessed on 2022-12-13.

[3] Christian Szegedy Ian J. Goodfellow Jonathon Shlens. "EXPLAINING AND HARNESSING ADVERSARIAL EXAMPLES". In: 62 (Feb. 2015). URL: https://arxiv.org/pdf/1412.6572.pdf.

[4] Vivek Ramavajjala Thang D. Bui Sujith Ravi. "Neural Graph Learning: Training Neural Networks Using Graphs". In: 62 (Feb. 2018). URL: https://storage.googleapis.com/pub-tools-public-publication-data/pdf/bbd774a3c6f13f05bf754e09aa45e7aa6faa08a8.pdf.

[5] Sanaa Alwidian Weimin Zhao and Qusay H. Mahmoud. "Adversarial Training Methods for Deep Learning". In: 62 (June 2022). URL: https://www.mdpi.com/1999-4893/15/8/283/pdf.

[6] *Adversarial regularization for image classification*. https://www.tensorflow.org/neural_structured_learning/tutorials/adversarial_keras_cnn_mnist. Accessed on 2022-12-9.