Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Continual Learning : Visual Recognition per Problemi con Task Incrementali

Candidato
Lorenzo Gianassi

Relatore
Andrew D. Bagdanov

Anno Accademico 2019/2020

Indice

Ringraziamenti

Il primo ringraziamento va al relatore di questa tesi, il professor Andrew D. Bagdanov, che mi ha seguito e corretto durante il percorso che ha portato a questo elaborato.
Vorrei ringraziare la mia famiglia che mi è sempre stata dietro senza mai farmi mancare niente, permettendomi di completare questo percorso.
Vorrei dire grazie a tutti i miei amici, in particolare modo al mio gruppo storico e ai miei compagni di squadra.
Grazie ai tutti miei i amici e i componenti della *"Gang"* dell'università .
Infine un grazie speciale va alla mia ragazza che mi ha supportato e sopportato durante tutti questi anni.

Introduzione

Motivazioni

Gli esseri umani e gli animali hanno la capacità di acquisire, perfezionare e modificare continuamente le proprie conoscenze e abilità per tutta la durata della loro vita. Questa capacità, chiamata *Continual Learning*, è composta da una ricca serie di meccanismi *neurocognitivi* che insieme contribuiscono allo sviluppo e alla specializzazione delle nostre abilità sensomotorie, nonché al rafforzamento e al recupero della memoria a lungo termine. Di conseguenza, l'applicazione del *Continual Learning* a *sistemi computazionali* e *agenti autonomi* che interagiscono nel mondo reale ed elaborano flussi continui di informazioni può essere di fondamentale importanza. Tuttavia, il *Continual Learning* rimane una sfida di lunga data per il *Machine Learning* e i modelli di *Rete Neurale* poiché l'acquisizione continua di informazioni disponibili in modo incrementale da distribuzioni di dati non stabili generalmente porta ad ottenere il *Catastrophic Forgetting*, concetto che andremo a definire successivamente.

Questo rappresenta un grave problema per i modelli di *Rete Neurale* moderni che in genere apprendono rappresentazioni da *batch* stazionari di dati di *Training*, quindi senza tenere conto delle situazioni in cui le informazioni diventano disponibili. Di conseguenza, la capacità di apprendere da un flusso

continuo senza avere una perdita di memoria sui dati precedenti potrebbe avvicinare l'apprendimento delle *reti neurali* a quello dell'essere umano con sviluppi interessanti per la scienza.

Presentazione del Lavoro

In questo elaborato di tesi, partendo dal lavoro di [1], verrà descritto il concetto *Continual Learning* mostrandone i pregi e le difficoltà(*Catastrophic Forgetting*) tramite un problema di *Visual Recognition*. Per fare ciò è stato implementato un *framework* che fosse in grado di eseguire una *Pipeline*, che avrà il compito di eseguire l'apprendimento svolto da un modello per il *Continual Learning* mostrando l'occorrere del *forgetting*. Prima di mostrare i risultati ottenuti verranno descritti i concetti a livello teorico e presentati gli strumenti utilizzati durante la *Pipeline*.
Il lavoro descritto è diviso in tre parti: la prima relativa al *Joint-Training* e le ultime due alle configurazioni *Task Agnostic/Task Aware*. Successivamente, sarà presentata una soluzione molto *naïve* ispirata dal *paper* [1].

Capitolo 1

Definizioni

In questo capitolo andiamo a introdurre i concetti principali su cui si basa l'elaborato, che ci serviranno successivamente per apprezzarne l'utilità.

1.1 Continual Learning

Negli ultimi anni i modelli del **Machine Learning** sono stati addirittura in grado di sorpassare l'intelletto umano per svariati problemi, come ad esempio il **Visual Recognition**. Sebbene questi risultati siano sorprendenti, sono stati ottenuti con modelli statici non in grado di espandere il loro comportamento o adattarlo nel tempo. Si introduce quindi il concetto del **Continual Learning** su cui si basa questo elaborato.
Continual Learning mira a creare algoritmi di*Machine Learning* in grado di accumulare un insieme di conoscenze apprese sequenzialmente. L'idea generale alla base di quest'ultimo è rendere gli algoritmi in grado di apprendere da una fonte di dati reale. In un ambiente naturale le opportunità di apprendimento non sono disponibili contemporaneamente e devono essere elaborate in sequenza.

Il **Continual Learning** studia il problema dell'apprendimento da uno *stream* infinito di dati, con l'obbiettivo di estendere gradualmente la conoscenza e di usarla per allenamenti successivi. La dimensione dello *stream* di dati ed il numero di *tasks* su cui si lavora non è necessariamente noto a priori. Il *Continual Learning* può essere anche definito come **Lifelong Learning**, **Sequential Learning** o **Incremental Learning**. Il concetto fondamentale su cui si basa è la natura sequenziale del processo di apprendimento, in cui solo una porzione dei dati di input di uno o più *tasks* è disponibile in quell'istante di tempo.

1.2 Catastrophic Forgetting

Come viene introdotto in [1], una rete neurale *dimentica* quando le sue prestazioni su una distribuzione dati vengono ridotte dall'apprendimento su una successiva. La sfida maggiore del *Continual Learning* consiste nell'apprendere evitando il **Catastrophic Forgetting**, cioè la performance di previsione su dati appartenenti a *tasks* precedentemente visionati nel training non dovrebbe calare nel tempo in seguito all'aggiunta di nuovi *tasks*. Definiamo adesso il **Catastriphic Forgetting**, noto anche come **Catastrophic Interference**, come la tendenza di una rete neurale artificiale a dimenticare completamente e in modo improvviso le informazioni apprese in precedenza dopo aver appreso nuove informazioni (come viene definito in [1]).

1.3 Stability-Plasticity Dilemma

Per sopperire al **Catastrophic Forgetting** i sistemi di apprendimento devono, da una parte , mostrare la capacità di acquisire nuova conoscenza

e affinare quella già esistente sulla base di un input continuativo, dall'altra, impedire alla nuova informazione di interferire con la conoscenza pregressa. Il concetto per il quale un sistema sia in grado di essere "*plastico*" per l'integrazione di nuove informazioni e "*stabile*" in modo da non interferire *catastroficamente* con la conoscenza precedentemente consolidata è definito con il nome **Stability-Plasticity Dilemma**.
Come viene descritto in [2] , troppa "plasticità" farà sì che i dati precedentemente codificati siano costantemente dimenticati, mentre troppa "stabilità" impedirà la codifica efficiente di questi dati a livello delle sinapsi.

1.4 Task Agnostic-Task Aware

Un concetto importante su cui si basa l'esecuzione del programma che simula il **Continual Learning** è la dualità di approccio **Task-Agnostic/Task-Aware**. Definiamo un approccio **Task-Agnostic** quando la rete non conosce bene i limiti dei *tasks*, cioè non conosciamo a quale *task* appartenga la *label* del dato in input alla **Rete Neurale Convoluzionale**. Mentre, nell'altro caso, l'approccio **Task-Aware** è contraddistinto dalla nozione del *task* corrente a cui appartiene il dato in input.
Questi due approcci ci consentono di ottenere risultati diversi sotto l'aspetto dell'*accuracy* della rete, sia durante che al termine del ciclo previsto per vedere tutti i *tasks*. In particolar modo noi possiamo avere per entrambi sia il processo di **training** che quello di **testing**, ottenendo sostanzialmente quattro combinazioni di possibili processi.
Vedremo successivamente come sarà possibile simulare queste due tipologie di *training/testing* tramite il metodo *set_ tasks* della classe che rappresenta la **rete neurale convoluzionale**.

Capitolo 2

Componenti del Progetto

In questo capitolo verranno mostrati gli strumenti utilizzati per la creazione del *framework*, che riproduce l'apprendimento del *Continual Learning*.

2.1 PyTorch

PyTorch è un *framework* di *Machine Learning* open source basato sulla libreria **Torch**, utilizzato per applicazioni come la visione artificiale e l'elaborazione del linguaggio naturale, sviluppata principalmente dal laboratorio di ricerca AI di Facebook. È un software gratuito e open source nato per essere utilizzato con il linguaggio di programmazione Python (come avviene in questo elaborato), ma presenta anche un'interfaccia C ++. È stato utilizzata questo *framework* perchè fornisce due specifiche molto utili per lavorare con le Reti Neurali: *Pytorch Tensor* e i Moduli (AutoGrad ,Optim, nn). PyTorch definisce una classe chiamata *Tensor* (torch.Tensor) per memorizzare e operare su array di numeri rettangolari multidimensionali omogenei.

2.1.1 PyTorch Tensor

I tensori PyTorch sono simili agli array NumPy, ma possono essere utilizzati anche su una GPU Nvidia compatibile con CUDA. Per lavorare al **Visual Recognition** è molto utile l'utilizzo della GPU: i tempi e le perfomance ne risentono in positivo.

2.1.2 Moduli

PyTorch utilizza un metodo chiamato *automatic differentiation* (**AutoGrad Module**). Un *Recorder* registra le operazioni eseguite, quindi le riproduce all'indietro per calcolare i gradienti. Questo metodo è particolarmente potente quando si costruiscono reti neurali per risparmiare tempo in un'epoca calcolando la differenziazione dei parametri nel passo del *Forward*.
Il secondo Modulo che ci interessa è **Optim Module**(*torch.optim*). È un modulo che implementa vari algoritmi di ottimizzazione utilizzati per la creazione di reti neurali. La maggior parte dei metodi più usati sono già supportati, quindi non è necessario crearli da zero. In particolare l'algoritmo di *ottimizzazione* che è stato scelto in questo elaborato è *optim.SGD*, il quale implementa lo *stochastic gradient descent*. Durante il *training* si adotteranno i metodi della classe *optim* come *.zero_grad()* e successivamente *.step()* per fare l'update dei parametri della rete.
Infine abbiamo il modulo ***torch.nn*** che ci fornisce molte più classi per implementare e addestrare la *rete neurale*. In particolare ci aiuta nella definizione della *rete neurale*, e in quella dei *layers* che la formano.

I *Packages* utilizzati in questo elaborato sono:

- *torch.nn.Module*: è una classe base per tutti i moduli di *Rete Neurale*
- *torch.nn.Linear*: utilizzato per i *layer Fully Connected*
- *torch.nn.Conv2d*: utilizzato per i *Layer Convoluzionali*
- *torch.nn.MaxPool2d*: utilizzato per applicare un max pooling 2D
- *torch.nn.ReLU*: funzione di attivazione dei *layer*
- *torch.nn.CrossEntropyLoss*: funzione per il calcolo della *Loss* della rete

Per tutte le nozioni su *Pytorch* e sui suoi *Module* e sui *Tensori* sono stati presi come riferimento i *docs* di *PyTorch* [3], [4] e [5].

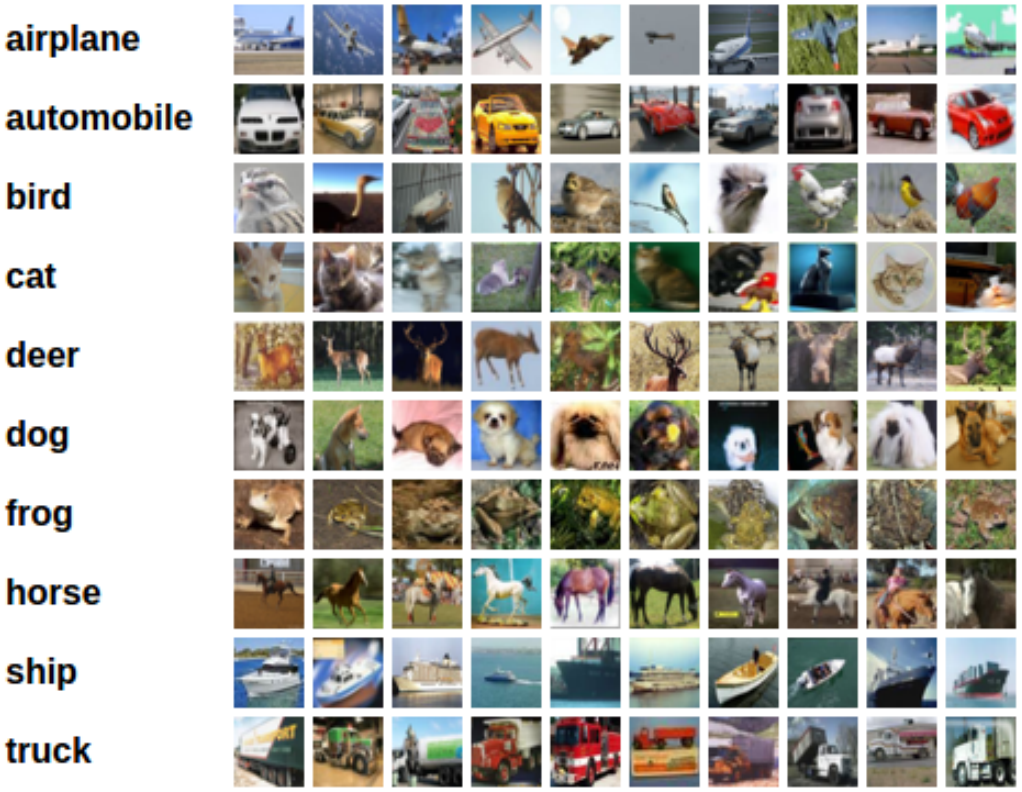
2.2 Dataset

2.2.1 CIFAR-10

Il dataset utilizzato in questo elaborato è **CIFAR-10**. Esso è costituito da 60000 immagini a colori (avranno 3 canali per ciascuna immagine per gestire l'RGB) 32x32 divise in 10 classi, con 6000 immagini per classe. Sono disponibili 50000 immagini di allenamento e 10000 immagini di prova, come viene descritto in [6].
Il dataset è suddiviso in cinque *batches* di *training* e un batch di *testing*, ciascuno con 10000 immagini. Il *batch* di prova contiene esattamente 1000 immagini selezionate casualmente da ciascuna classe. I *batches* di *training* contengono le immagini rimanenti in ordine casuale, ma alcuni di essi possono contenere più immagini di una classe rispetto a un'altra. In particolare, i *batches* di *training* contengono esattamente 5000 immagini di ciascuna classe.

In questa immagine possiamo notare le dieci **classi** di **CIFAR-10** oltre a dieci esempi presi in maniera casuale da ciascuna classe.

Figura 2.1: Classi di CIFAR-10



2.2.2 Gestione del Dataset

Alla base della gestione dei dataset vi è la classe *torch.utils.data.DataLoader*. In particolare tutti i dataset sono sottoclassi di torch.utils.data.Dataset, ovvero presentano i metodi *getitem()* e *len()* implementati.
Per gestire il dataset **CIFAR-10** è stato istanziato nel progetto una classe *Filtered_ dataset* che si occupasse di questa mansione. L'obbiettivo dell'elaborato era rappresentare il problema del **Continual Learning** e per fare ciò

è stato necessario dividere il dataset utilizzato in ***tasks***. Ciò significa che a seconda del numero di *tasks* che si vuole utilizzare per simulare il **Continual Learning** sarà diviso il dataset per *labels*. Ad esempio se si volesse utilizzare 5 *tasks*, ognuno di essi conterrebbe 2 *labels*.
La classe *Filtered_ dataset* si occupa di creare dei **subsets** con il metodo della libreria di **PyTorch** *torch.utils.data.Subset* che crea quest'ultimi dal dataset originario, supportato dal metodo *idx_ tasks* (usato per la divisione delle *Labels* nei *tasks*). Nel processo di suddivisione del dataset è stato necessario *"mappare"* le *labels* del dataset in modo tale da essere coerenti con l'output della rete. In particolare ciò è stato fatto tramite due attributi della classe *Filtered_ dataset*:*original2task* e *task2original*. Questi due attributi consistono in due dizionari con chiave-valore, le *labels* e la rispettiva mappatura. Si reso inoltre possibile *randomizzare* le *labels* all'interno di ciascun *task* per poter fare ulteriori *tests* con il metodo *idx_ tasks*.

2.3 Rete Neurale Convoluzionale

La **Rete Neurale Convoluzionale** (CNN o ConvNet) è una classe di *reti neurali* profonde, molto spesso applicata all'analisi ed al riconoscimento delle immagini. La **Rete Neurale Convoluzionale** utilizzata in questo progetto è formata da sei *Conv2D* layers separati da due *MaxPool2D* ed infine da due *Fully Connected Layer* sui cui ho applicato *Dropout* per limitare l'*overfitting* durante il *training*. Per la scelta della conformazione della rete e per le definizioni delle tipologie dei vari *layers* ho seguito il libro [7] e i docs di *PyTorch*. La particolarità di questa rete è che il *layer* dell'output è ***Dinamico***, cioè che cambia a seconda del numero di *tasks* su cui si vuole lavorare

e sulla tipologia di approccio scelto tra *Task-Agnostic* e *Task-Aware*. In particolare è possibile aggiungere nuovi *layers* di output con il metodo *add_task* che richiama semplicemente il metodo *add_module* della classe *nn.Module*. Un altro metodo importante della *rete neurale* sarà *set_tasks* che permette di settare il numero di *tasks* voluti come output. I due attributi fondamentali della classe *net* sono *task_fcs* e *current_tasks*, che sono sostanzialmente due *array* contenenti gli indici dei *tasks*. Il primo conterrà tutti i *layers* lineari per le varie *Classification Head*, mentre il secondo, seleziona quale(i) task(s) sono correntemente attivi. Qui di seguito si illustra la *rete neurale* per un singolo *task* con tutte le classi corrispondenti al *Joint Training*:

Net(
(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(pool): MaxPool2d(kernel_size=2,stride=2, padding=0, dilation=1,
ceil_mode=False)
(conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv5): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(fc1): Linear(in_features=16384, out_features=120, bias=True)
(fc2): Linear(in_features=120, out_features=84, bias=True)
(dropout): Dropout(p=0.5, inplace=False)
(task0_fc): Linear(in_features=84, out_features=10, bias=True)
)

Il *layer Task0_fc* è l'ultimo che è stato aggiunto con *add_task* e selezionato con *set_task*. Se l'esperimento fosse stato condotto su più *tasks* ci sarebbero stati altri *layers* oltre a *Task0_fc*. Nel caso in cui l'output voluto fosse stato su più tasks, nel metodo *Forward* della rete tramite *torch.cat*, sarebbero stati concatenati tra di loro i parametri corrispondenti selezionati da *current_tasks*.

La *Funzione di Attivazione* utilizzata nei *layers convoluzionali* è **ReLu**, come anticipato precedentemente. La **rectified linear activation function** o **ReLU** in breve è una funzione lineare a tratti che darà come output direttamente l'input se è positivo, altrimenti produrrà zero. L'utilizzo della **ReLu** consente di ottenere un *training* e una performance migliori.

Per quanto riguarda la funzione che si occupa del calcolo della *Loss* è stata selezionata la **CrossEntropyLoss**. Questa funzione combina in una unica classe *nn.LogSoftmax()* e *nn.NLLLoss()*. Come anticipato nel paragrafo relativo a **PyTorch** è stato utilizzato come *optimizer* **SGD** con *Learning Rate* pari a 0.001.

Infine, è stato reputato necessario applicare **Dropout** ai due *Fully Connected Layer* che precedono il *layer* di output dinamico. Durante il *training* azzera in modo casuale alcuni degli elementi del tensore di input con probabilità p utilizzando campioni da una distribuzione di Bernoulli. In questo modo è stata ottenuta una diminuzione dell'*overfitting* riscontrato nella *rete*.

Capitolo 3

Esperimenti

3.1 Introduzione al Progetto

Prima di poter iniziare a mostrare il progetto è necessario porre delle basi e limiti per quest'ultimo. Per analizzare il concetto del **Continual Learning** ci concentreremo sul problema di *classificazione*, tipico del *Deep Learning*. La classificazione implica la previsione a quale classe appartenga un elemento. Alcuni classificatori sono binari, altri sono multi-classe, in grado di discernere un esempio in una delle diverse classi. Noi, quindi, ci andremo a concentrare sull'utilizzo di un classificatore *multi-classe*.
La seconda limitazione riguarda l'approccio ***Task Incremental***.
Il ***Task Incremental*** corrisponde ad un approccio in cui i dati arrivano in sequenza di *batches* e ognuno dei quali corrisponde ad un *task*. Ad ogni *task* corrisponderà un nuovo insieme di *labels* il cui numero dipenderà dalla quantità di quest'ultime nel *dataset* e dalla divisione scelta. In altre parole, assumiamo che per un dato *task*, tutti i dati diventino disponibili simultaneamente seguendo il concetto di *Training Offline*. Ciò consente un *training* per più epoche su tutti i suoi dati di addestramento, mescolati ripetutamente

per garantire delle condizioni di *i.i.d.*. È importante sottolineare che i dati appartenenti al precedente o al futuro *task* non saranno utilizzabili. Ottimizzare/Allenare per un nuovo *task* in questa configurazione si tradurrà nel ***Catastrophic Forgetting***, con significativi cali sulle prestazioni relative ai vecchi *tasks*, salvo siano adottate strategie specifiche.
A differenza della limitazione alla configurazione *Multi-Head* utilizzata nel paper [1], in questo elaborato proveremo ad analizzare entrambe le configurazioni *Multi-Head/Single-Head*. Ciò corrisponde ai due approcci che abbiamo già introdotto nel primo capitolo: **Task-Agnostic/Task-Aware**. Nel caso di **Task-Agnostic** avremo una *Single-Head* per tutti i *tasks* perchè non è noto su quale stiamo facendo *training/testing*, mentre per **Task-Aware** avremo una *Multi-Head* e selezioneremo l'*Output* corrispondente a quello corrente.

3.2 Pipeline

Per simulare il processo di **Continual Learning** è stato necessario stabilire una *Pipeline* che avrebbe dovuto seguire l'algoritmo. A seconda delle tipologie di approccio **Task-Agnostic/Task-Aware** avremo delle differenze all'interno della *Pipeline* che verranno analizzate successivamente.
Per descrivere al meglio il problema del *Catastrophic Forgetting* è stato valutato di dividere il dataset in 5 *tasks* ciascuno con i dati relativi a due *labels*, visto che *CIFAR-10* ha 10 classi. Se fossero stati utilizzati solamente due *tasks* si sarebbero potuti ottenere dei risultati poco significativi per il progetto.

La ***Pipeline*** del processo è la seguente:

1. Creare ***Rete Neurale Convoluzionale*** che farà da *Backbone*;
2. Per ogni t in *Tasks*:
   (a) Aggiungere un nuovo *Classification Module* per il *task* corrente;
   (b) *SetTask* per selezionare l'*output* corretto della rete a seconda di *Aware/Agnostic Training*;
   (c) Fare il *Training* per il *Task* t;
   (d) *SetTask* per selezionare l'*output* corretto della rete a seconda di *Aware/Agnostic Testing*;
   (e) Fare *Test* per il *Task* t;
3. Fare *Test* per ogni *task* dopo l'ultimo *Training*, selezionando *output* giusto per la tipologia di *testing*.

2.b/2.d/3 sono le fasi che vengono influenzate dalla scelta della tipologia di *Agnostic/Aware*. Ciò consiste nel fatto che l'**output** della rete verrà modificato seguendo il paradigma *Task-Aware/Task-Agnostic*, diventando unico per più *tasks* nel caso *Agnostic* e singolo per il *task* specifico per *Aware*.
Inoltre, il processo della ***pipeline*** sarà il medesimo sia al variare del numero di *tasks* che della formazione di quest'ultimi (caso di *labels randomiche*).
Avremo, quindi, 5 configurazioni diverse di Processi di cui 4 andranno a combinare *Aware/Agnostic outputs* e una sarà relativa al *Joint-Train*.
Verranno mostrati i risultati ottenuti nelle quattro configurazioni tenendo presente come ***upperbound*** il valore della ***accuracy*** ottenuta dal ***Joint-Train***.

3.3 Esperimenti

Quindi le configuazioni di nostro interesse saranno:

1. ***Joint-Training/Testing***: Corrisponde sostanzialmente ad allenare e testare la rete su tutte le *labels* contemporaneamente. Sarà equivalente ad un unico *Task* con tutti gli *examples* del dataset.

2. ***Task-Agnostic Training / Task-Agnostic Testing***: *training* con *output* per ogni *task*, *testing* con *output* per ogni *task*.

3. ***Task-Agnostic Training / Task-Aware Testing*** : *training* con *output* per ogni *task*, *testing* con *output* per *task* specifico.

4. ***Task-Aware Training / Task-Aware Testing*** : *training* con *output* per *task* specifico, *testing* con *output* per *task* specifico.

5. ***Task-Aware Training / Task-Agnostic Testing***: *training* con *output* per *task* specifico, *testing* con *output* per ogni *task*

Per comprendere nel miglior modo i risultati saranno rappresentate le *accuracies* delle configurazioni in due grafici. Il primo sarà relativo al ***Task-Agnostic Training*** riportando le *accuracies* relative ai casi *Aware/Agnostic* così da visualizzare le differenze, mentre il secondo avrà ***Task-Aware Training***. Entrambi i grafici saranno confrontati al valore *Joint-Training* che rappresenterà l'*upperbound* per qualsiasi configurazione.
Un altro valore importante da analizzare per comprendere al massimo il valore del *Catastrophic Forgetting* è la differenza tra la media delle *accuracies* dopo l'allenamento relativo a ciascun *task* e quella dopo l'ultimo *task*. Questo valore ci fornisce il ***Catastrophic Forgetting*** a cui siamo andati incontro grazie al *Continual Learning*.

3.3.1 Joint-Training

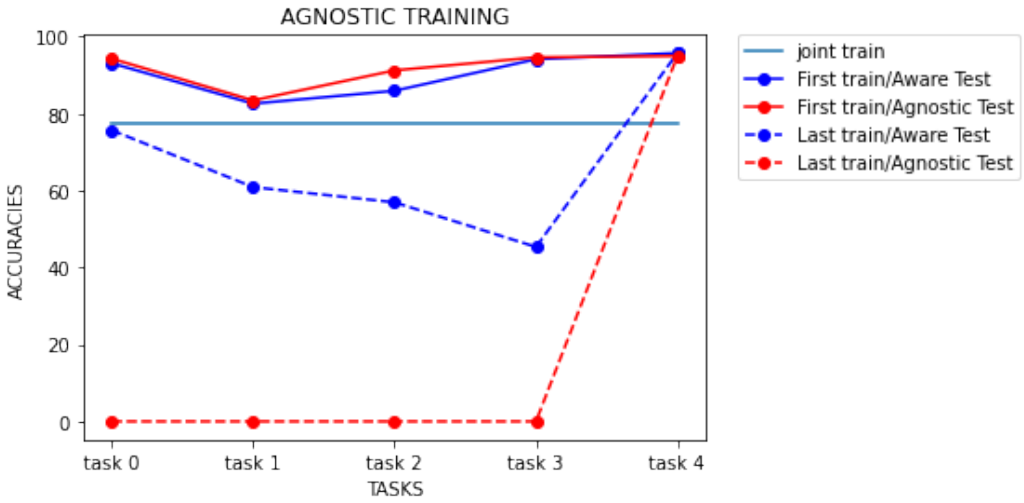
Prima di andare ad analizzare le varie configurazioni è necessario concentrarci sul ***Joint-Training***.

Il ***Joint-Training*** corrisponde al generico procedimento di *Training/Testing* che viene eseguito nel *Visual Recognition*. Ciò consiste in un addestramento e *testing* fatto sulla totalità degli esempi appartenenti ai *batches* che compongono il dataset senza considerare la divisione in *Tasks* ignorando, quindi, il paradigma del *Continual Learning*. Questa *baseline* ci fornisce un *upper-Bound* per le *accuracies* rilevate. Il valore di *accuracy* che abbiamo ottenuto dal nostro *Joint-Training* è di ***77.6%*** e lo utilizzeremo come riferimento nei nostri grafici.

3.3.2 Agnostic-Training

Qui di seguito viene riportato il grafico con i risultati ottenuti sia per *Agnostic* che *Aware Testing*:

Figura 3.1: Agnostic Training



Prima di analizzare il grafico andiamo a considerare il paradigma l'*Agnostic-Training* che consiste nel *training* senza sapere di quale *task* ci stiamo occupando. Ciò comporta che ad ogni *task* l'*output* della rete consisterà in tutti i moduli dei *tasks* fino a quello corrente concatenati nel metodo *Forward*, questo perchè non possiamo selezionare l'*output* relativo al *Task* in esecuzione. Dal grafico 3.1, che si trova a pagina precendete, possiamo notare vari aspetti interessanti sui risultati ottenuti. Prima di tutto, notiamo che i valori delle *accuarcies* sui vari *tasks*, calcolate dopo i rispettivi *Training*, ottengono valori molto elevati che superano persino il valore del *Joint-Training*. Questo è dovuto al numero minore di dati utilizzati rispetto al dataset completo e dal fatto che il *test* è effetuato subito dopo il *training* del relativo *task*. Inoltre, da notare come i due approcci di *testing* ottengano *accuracies* quasi identiche: la rete viene *testata* subito dopo il *training* relativo di conseguenza i pesi associati alla clasificazione sono molto precisi per entrambe le configurazioni, ottenendo risultati per ciascun *task* migliori anche del *Joint-Training* . L'*accuracy* calcolata sull'ultimo *task*, naturalmente, avrà valore uguale per tutte e quattro le configurazioni, come si può notare nel grafico 3.1 nel punto relativo a quest'ultimo.

Per quanto riguarda le *accuracies* calcolate successivamente all'ultimo *training*, vediamo che si ottiene un calo drastico di precisione su ciascun *Task* escluso l'ultimo come abbiamo precedentemente affermato. Questo è il fenomeno del **Catastrophic Forgetting**, introdotto nel Capitolo 1, che porta il *modello* a dimenticare tutti i *weight* relativi ai *tasks* precedenti. In particolare nella configurazione *Aware* si ottengono dei valori migliori dati dall'output più preciso e specifico per il relativo *task*, dati dalla selezione della *Classification Head* relativa a quest'ultimo. Per quanto riguarda la configurazione di *Agnostic Testing*, otteniamo un valore molto alto per l'ultimo *Task* mentre

per i 4 precedenti l'*accuracy* cala a 0. Questo fenomeno prende il nome di ***Task Recency Bias***. Si tratta di un fenomeno che consiste nel fatto che la *rete* abbia la tendenza a ricordare e a prevedere meglio dati su cui è stato fatto per ultimo il *training* portando a dimenticare totalmente i **parametri** appartenenti ai *tasks* precedenti, come viene descritto in [8]. Questo risultato è ricondotto all'utilizzo di *CrossEntropyLoss* che aumenta il valore della probabilità relativa alla classe corretta e diminuisce quella relativa alla classe non corretta all'interno della distribuzione di probabilità, dato che grazie al *SoftMax* la somma dei valori deve essere uguale a uno. Ad ogni esempio del *batch* del *task* corrente riduce la probabilità delle classi appartenenti a quelli precedenti, andando incontro al **Catastrophic Forgetting**.
Per questo motivo è interessante valutare le medie delle *accuracies* per analizzare l'occorrere del **Catastrophic Forgetting** nelle varie configurazioni. Qui di seguito i valori delle *accuracies* sono riportate in due tabelle, una per tipologia di *testing*.

Tasks	First Train	Last Train
Task 0	94.30	0.0
Task 1	83.45	0.0
Task 2	91.25	0.0
Task 3	94.65	0.0
Task 4	95.00	95.00

Tabella 3.1: Agnostic-Agnostic

Tasks	First Train	Last Train
Task 0	93.10	75.70
Task 1	82.60	60.90
Task 2	85.95	56.95
Task 3	94.15	45.35
Task 4	95.75	95.75

Tabella 3.2: Agnostic-Aware

Nelle tabelle 3.1 e 3.2 notiamo, come avevamo già fatto nel grafico in figura 3.1 a pagina 15, che l'*accuracy* della configurazione *Agnostic-Agnostic* è peggiore rispetto a quella di *Agnostic-Aware*, ma per valutare la differenza

di valori, ma soprattutto il *forgetting*, calcoliamo la media di quest'ultimi. Facendo le medie otteniamo i seguenti valori:

- Tabella 3.1: Abbiamo una *accuracy* iniziale di 91.73% e finale di 19.0%, quindi otteniamo un decremento del 72.71%.
- Tabella 3.2: Abbiamo una *accuracy* iniziale di 90.30% e finale di 66.92%, quindi otteniamo un decremento del 23.38%.

Notiamo che la media dell'*accuracies* del *Task Agnostic* e *Task Aware Testing* sono entrambe inferiori dell'*upperbound* rappresentato dal *Joint-Train*, confermando ciò che si poteva notare già a livello grafico nell'immagine 3.1. Il decremento della *accuracy* ci serve a valutare l'entità del *Catastrophic Forgetting* a cui siamo andati incontro.
Questo valore inoltre ci conferma il *forgetting* molto elevato che abbiamo ottenuto sui *tasks* precedenti all'ultimo utilizzando la configurazione *Agnostic-Agnostic*. Vedremo successivamente una soluzione *naïve* al problema del *Task Recency Bias*, basato sull'approccio *Replay Based Methods*.
In generale, il decremento di *accuracy* per entrambe le configurazioni utilizzate in questo paragrafo è elevato, anche se con entità diverse. Inoltre la distanza di *accuracy* in media dal valore ottenuto con il *Joint-Train* assume un valore del 10,68%, per la configurazione *Agnostic-Aware*, 58,60% per *Agnostic-Agnostic*. Questo ci fa comprendere l'entità del *drop* di *accuracy* a cui si può andare incontro adottando una divisione in *tasks* dovuto al *Catastrophic Forgetting*.

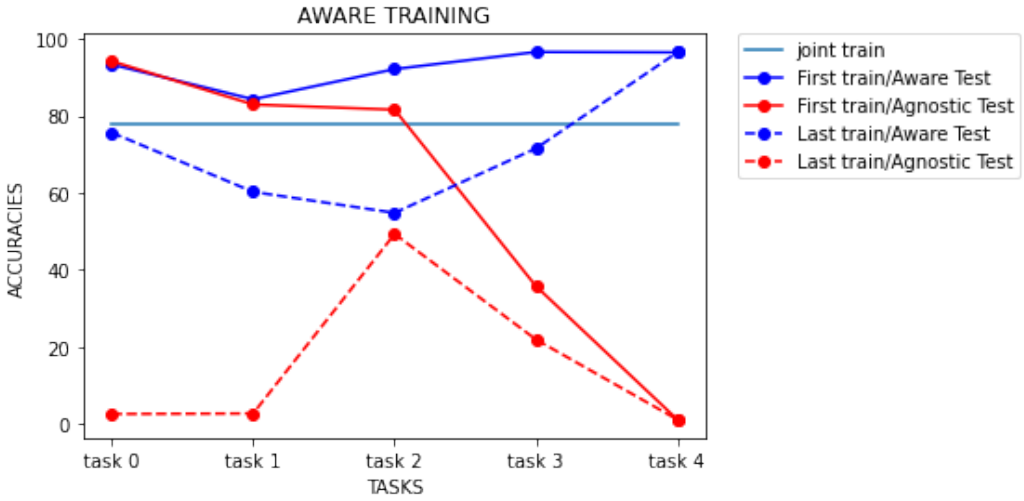
3.3.3 Aware-Training

In questa sezione ci concentriamo su altre due configurazioni *Task-Aware* per il *training* introdotte già a *pag.*14. A differenza della configurazione

Task-Agnostic possiamo selezionare la *Classification-Head* relativa al *Task* corrente: di conseguenza il *Training* per ognuno di essi sarà eseguito con un *output* di soli due valori modificando, quindi, l'aggiornamento dei parametri della *Rete*.

Riportiamo qui di seguito il grafico che rappresenta le *accuracies*:

Figura 3.2: Agnostic Training



Dalla figura 3.2 possiamo notare che l'andamento generale delle *accuracies* segue, in linea generale, quello della figura 3.1 a pagina 15. Di conseguenza rileviamo che l'*accuracy* su un *task* calcolata subito dopo il corrispondente addestramento ha un valore molto buono, che in media è addirittura superiore al *Joint-Training* per *Aware-Test*. Per *Agnostic-Test*, invece, tende a calare all'aumentare del *task*. Inoltre abbiamo che l'*accuracy* sull'ultimo *task* ha lo stesso valore per tutti i casi riportati, tranne uno: *Aware-Agnostic*. Si può notare che l'*accuracy* calcolata sull'ultimo *task* a seconda della tipologia assume dei valori diversi. In particolare per la configurazione *Agnostic Test* notiamo che l'*accuracy* subito dopo il *Training* assume un valore sempre

minore fino all'ultimo *task* in cui è quasi nulla. Questo risultato è ottenuto perchè durante la fase *training* utilizziamo la *Classification-Head* specifica del *task* mentre nella fase di *Testing* utilizziamo un *output* unico per tutti *tasks* fino a quello corrente. Adesso mostriamo di seguito le tabelle con le *accuracies* ottenute per poter capire il *forgetting* ottenuto e comparare i risultati con l' *Agnostic-Training*:

Tasks	First Train	Last Train
Task 0	94.15	2.50
Task 1	82.90	2.65
Task 2	81.55	49.25
Task 3	35.70	21.85
Task 4	1.05	1.05

Tabella 3.3: Aware-Agnostic

Tasks	First Train	Last Train
Task 0	93.30	75.60
Task 1	84.25	60.20
Task 2	92.10	54.75
Task 3	96.55	71.50
Task 4	96.45	96.45

Tabella 3.4: Aware-Aware

Nelle tabelle 3.3 e 3.4 notiamo subito che la *accuracy* rilevata nel caso di *Aware-Testing* è migliore, ma consideriamo adesso le medie delle *accuracies* e il *forgetting* ottenuto.

- Tabella 3.3: Abbiamo una *accuracy* iniziale di 59.07% e finale di 15.45%, quindi otteniamo un decremento del 43.61%.
- Tabella 3.4: Abbiamo una *accuracy* iniziale di 92.53% e finale di 71.7%, quindi otteniamo un decremento del 20.83%.

La prima cosa che rileviamo è che l'*accuracy* iniziale della configurazione *Agnostic-Agnostic* ha ottenuto un valore in media molto minore rispetto alle altre, dovuto all'*Agnostic-Testing*. L'*accuracy* ottenuta nel caso *Aware-Aware* è il miglior risultato e si avvicina a quella del *Joint-Training* con uno

scarto del 5,9%. Mentre nel caso della Tabella 3.3 l'*accuracy* ottenuta rappresenta il "*Lower-Bound*" dei risultati con uno scarto dal *Joint-Training* del 62,15%.

3.4 Soluzione Naïve

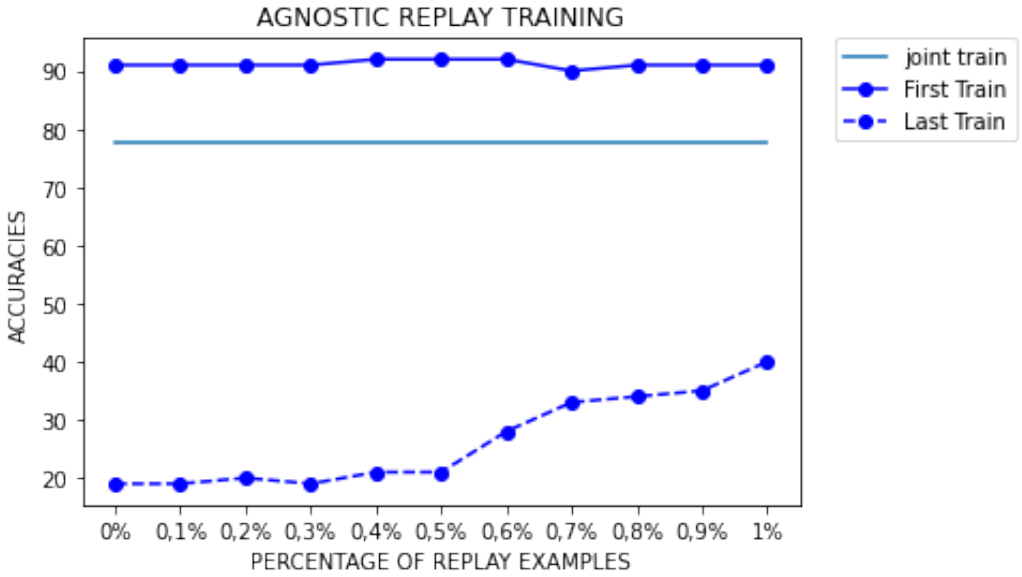
In questa sezione andiamo a proporre una soluzione con un *naïve* al problema del *Catastrophic Forgetting*. Esistono tre famiglie di soluzioni al problema del *Continual Learning*, che vengono descritte in [1]:

- *Replay-based methods*
- *Regularization-based methods*
- *Parameter isolation methods*

In questa sezione ci soffermeremo su una soluzione *Naïve* della famiglia dei ***replay-based methods***. Questo approccio consiste nel memorizzare i campioni o generare *pseudo-campioni*, con un modello generativo, appartenenti ai *tasks* precedenti. Questi esempi vengono riutilizzati durante l'apprendimento di un nuovo *task* per alleviare il *forgetting*.
Il problema principale dei *replay-based methods* risiede nella memoria: salvando esempi dai *tasks* precedenti la memoria necessaria a ciascuna fase di *training* sarà sempre maggiore. Ciò può esser ovviato utilizzando un limite di esempi possibili dai precedenti *tasks*, ottenendo però una perdita di generalizzazione del rispettivo *task*. In particolare questi due metodi utilizzano una specifica tecnica di scelta degli esempi da ciascun insieme di esempi appartenenti ai *tasks* precedenti.
In questo elaborato, però, adotteremo una soluzione *naïve* e semplificata, scegliendo gli esempi in modo *randomico* senza basarci su nessuna metrica o

bias. In particolare ci soffermeremo sul caso *Agnostic-Training/Agnostic-Test* in modo tale da poter apprezzare l'aumento di *accuracies* media e confrontarlo con il *Joint-Training* (essendo essenzialmente della stessa tipologia di configurazione). Inoltre, tale configurazione rappresenta il caso più vicino al superamento dell'approccio incrementale, non facendo affidamento sulla conoscenza del *task* corrente. Mostriamo come cambia il valore delle *accuracies* al variare del numero di esempi utilizzati appartenenti ai *tasks* precedenti per ogni processo di *training*. Partiamo dall'utilizzo dello 0.1% degli esempi precedenti, fino ad arrivare all'1%, andando a confrontare con il valore dei *Joint-Training* che rappresenta l'*UpperBound*. Come in precedenza, andiamo a riportare di seguito il grafico:

Figura 3.3: Agnostic Replay Training



Come si può notare dal grafico 3.3 all'aumentare della percentuale del numero di esempi di *replay* migliora l'*accuracy* in media, diminuendo il *forgetting*. Tutto ciò è stato ottenuto senza l'utilizzo di nessuna strategia specifica per

selezionare gli esempi ad ogni iterazione del *training*. Avremmo potuto, quindi, ottenere risultati ancora migliori rispetto a quelli presentati in 3.3.
Inoltre aumentando il numero degli esempi utilizzati nel *replay-training* dei *tasks* precedenti sarebbero stati ottenuti risultati migliori.
La scelta di questa percentuale è, però, in parte vincolata. Sarebbe stato poco rappresentativo utilizzare un numero elevato e poco intelligente da un punto di vista delle risorse di memorizzazione. Nel caso di *CIFAR-10* il problema della memoria utilizzata non sussiste, ma se avessimo utilizzato un dataset dal numero di esempi maggiore lo spazio di archiviazione assegnato agli esempi dei *tasks* precedenti sarebbe stato un punto focale, essendo un punto debole dei *replay-based methods*. Ciò che si nota dal grafico 3.3 è che l'*accuracy* ottenuta dal valore 50 dell' ascisse in poi tende a salire, mentre il valore rimane stabile intorno al 20% nei valori delle ascisse precedenti. Possiamo, quindi, affermare che utilizzando un numero superiore a 50 esempi per applicare il *Replay Training* otterremo un miglioramento di prestazioni crescente all'aumentare di tale valore.
La crescita comunque non è elevata e rimane un grosso divario tra i risultati ottenuti e il valore del *Joint-Training*, rimarcando nuovamente il problema del ***Catastrophic Forgetting*** che affligge le rappresentazioni pratiche del *Continual Learning*.

Capitolo 4

Conclusioni

4.1 Risultati

Il lavoro che è stato descritto in questo elaborato di tesi può essere ritenuto soddisfacente. Il nostro intento era quello di rappresentare al meglio il **Continual Learning** e il suo problema del **Catastrophic Forgetting**, perciò nella nostra *pipeline* non è stato inserito nessun metodo che potesse alleviare o bloccare tale problema. Abbiamo potuto osservare come il *training* effettuato in fasi diverse per ciascun *task* abbia portato al *forgetting*, valutandolo sia nel caso *Task-Aware* che *Task-Agnostic*.
È stato riscontrato un *forgetting* minore nelle configurazioni in cui è stato applicato il *Task-Aware Test*, peggiore con *Task-Agnostic Test*. Per questo motivo è stato scelto come caso interessante, su cui applicare una soluzione basata *Replay-Based Methods*, *Task-Agnostic Training/Task-Agnostic Test*. Con tale soluzione è stato possibile ottenere dei risultati migliori rispetto ai precedenti. Infatti, questa configurazione soffriva del *Task Rececncy Bias* che portava a dimenticare completamente i parametri dei *tasks* precedenti all'ultimo visionato. Usando un numero di esempi limitato per fare il *replay* siamo

riusciti ad avvicinarci al *Joint-Training*, mantenendo comunque un *forgetting* elevato. La rete neurale sviluppata non è ovviamente **immune** da **errori**, potrebbero essere apportate delle migliorie in modo tale da adattarsi meglio al problema in esame, ottenendo risultati migliori.

4.2 Sviluppi Futuri

Negli esperimenti eseguiti in questo elaborato, abbiamo considerato un ambiente di apprendimento basato sul concetto *Task Incremental*. In questo *setting* i *tasks* vengono ricevuti sequenzialmente e il *Training* viene eseguito sui dati di addestramento associati. È, quindi, richiesta la conoscenza dei limiti dei *tasks* (ovvero quando i *tasks* cambiano), consentendo più passaggi su grandi *batch* di dati di *training*. Può essere, quindi, un rilassamento del sistema di *Continual Learning* desiderato che è più probabile incontrare nella pratica. Una evoluzione potrebbe essere quella di rendere il modello capace di processare dati di *tasks* diversi senza considerarne i limiti, al fine di riconoscere se l'*input* appartiene a un *task* già osservato. Questa modifica potrebbe conferire grande flessibilità al metodo del *Continual Learning* rendendolo applicabile a qualsiasi scenario in cui i dati arrivano con uno *stream* infinito.

Un altro sviluppo possibile, potrebbe essere quello di utilizzare un dataset diverso. Sarebbe appunto interessante osservare i risultati all'aumentare del numero di classi presenti nel dataset, o altrimenti, all'aumentare del numero di esempi presenti nel *trainset* e *testset*. Un dataset che viene spontaneamente in mente dopo la lettura di questo elaborato è *CIFAR-100*. Quest'ultimo non è altro che una estensione di *CIFAR-10* (utilizzato in questo elaborato), composto da 100 *classi* differenti. Tale modifica ci consentirebbe di visualiz-

zare più *tasks* e con un numero di classi associato maggiore.

Infine, nella soluzione che abbiamo esposto per alleviare il *forgetting* non è stata attuata nessuna tecnica per sceglie gli esempi su cui fare il *replay*, quindi una direzione di sviluppo potrebbe essere questa. Come viene descritto in [1], esistono dei metodi specifici del **Reaplay Based Methods**: *iCaRL* e *GEM*. In particolare, questi due metodi attuano delle politiche per la scelta degli esempi utilizzati nel *replay*. *iCaRL* si basa sulla stima della *Loss*, mentre *GEM* si concentra sul *gradiente*.

Bibliografia

[1] Marc Masana Sarah Parisot Xu Jia Ales Leonardis Gregory Slabaugh Tinne Tuytelaars Matthias De Lange, Rahaf Aljundi. A continual learning survey: Defying forgetting in classification tasks. *arXiv preprint arXiv:1909.08383*, 2020.

[2] Jose L. Part Christopher Kanan Stefan Wermter German I. Parisi, Ronald Kemker. Continual lifelong learning with neural networks: A review. *Journal of the International Neural Network Society*, 2019.

[3] PyTorch. Torchvision. `https://pytorch.org/docs/stable/torchvision/index.html`, 2019.

[4] PyTorch. Torchvision.datasets. `https://pytorch.org/docs/stable/torchvision/datasets.html`, 2019.

[5] PyTorch. Torch.optim. `https://pytorch.org/docs/stable/torchvision/datasets.html`, 2019.

[6] Alex Krizhevsky. The cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`, 2009.

[7] Francois Chollet. *Deep Learning with Python*. Manning Publications, 2017.

[8] Bartłomiej Twardowski Mikel Menta Andrew D. Bagdanov Joost van de Weijer Marc Masana, Xialei Liu. Class-incremental learning: survey and performance evaluation. *arXiv preprint arXiv:2010.15277*, 2020.