Machine Learning
Few-Shot Learning with Prototypical Networks

Edoardo Re
7042226
edoardo.re@stud.unifi.it

Abstract

Nella seguente relazione si descrivono la teoria, l'implementazione ed infine i risultati di esecuzione di un algoritmo di few-shot learning, nello specifico di un metodo di meta-learning che si avvale di prototypical networks. Si mostra che feature backbone profonde riescono a ridurre significativamente la differenza di performance tra metodi allo stato dell'arte su datasets con limitate differenze di dominio. Infine si analizza la capacità di generalizzazione tra cross domain per l'algoritmo di few-shot classification implementato e si propongono degli ulteriori adattamenti.

1. Introduzione: Meta Learning

Il meta learning, spesso definito come come learn to learn ha come scopo quello di migliorare un algoritmo di apprendimento effettuando multipli episodi di learning [4]. Solitamente vi è un algoritmo esterno (upper/meta) che ne aggiorna uno interno in modo tale che il modello che apprende migliori l'obiettivo esterno. Nello specifico della prototypical network il livello esterno apprende uno spazio metrico, ovvero addestra un estrattore di features in modo tale da rendere i dati comparabili. Possiamo ricondurre infine l'obiettivo a quello di studiare un algoritmo general purpose che generalizzi sui tasks e idealmente permetta ad ogni nuovo task di essere studiato meglio del precedente. Nel machine learning supervisionato convenzionale si ha che dato un dataset di training $D = \{(x_1, y_1)...(x_N, y_N)\}$ si addestra un modello predittivo $\hat{y} = f_\theta(x)$ parametrizzato da θ risolvendo: $\theta^\star = \text{argmin}_\theta \, L(D; \theta, \omega)$ con L la loss function che misura l'errore tra true labels e quelle predette da $f_\theta(.)$. Il condizionamento ω denota la dipendenza della soluzione dalle assunzioni di "how to learn". Il meta learning invece valuta le performance di ω su di una distribuzione di tasks $p(\tau)$ dove $\tau = \{D, L\}$. Dunque "learning how to learn" diventa $\min_\omega \mathbb{E}_{\tau \sim p(\tau)} L(D; \omega)$. Nei prossimi paragrafi scenderemo nel dettaglio di come l'algoritmo di few-shot meta learning sia stato progettato ed infine implementato.

2. Architettura

L'architettura proposta e implementata (Fig.1)[1] si articola in due parti: Meta-training e Meta-testing. Nella prima fase l'algoritmo seleziona N classi e preleva piccoli S_b (base support set) e Q_b (base query set) dai dati delle classi selezionate. L'obiettivo è quello di addestrare un modello di classificazione M che minimizzi la loss N-way di predizione: L_{N-way} dei campioni nel query set Q_b. Il classificatore M è condizionato dal support set S_b fornito, dunque un metodo di meta learning può apprendere come studiare da dati e label limitate tramite il training di una collezione di tasks (episodi). Nella fase di Meta-Testing invece, tutti i novel class data X_n sono considerati come il support set per classi novel S_n e sempre lo stesso classificatore M viene adottato per predire le novel classes con il nuovo support set S_n. Il modello M condizionato dal support set adottato nei successivi test fa uso delle prototypical network, ovvero un tipo di rete che compara la distanza euclidea tra query features e la media di classe delle support features (Fig. 2).

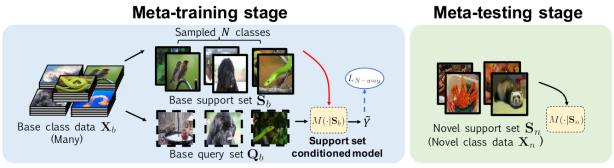

Figura 1. Meta-learning few-shot classification

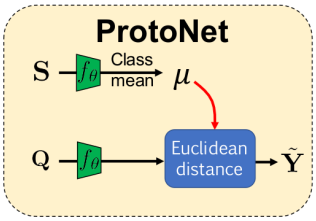

Figura 2. Modello Prototypical Network

3. Prototypical Networks

La Prototypical Networks descritta da Snell et al. [2] è realizzata in modo tale che il classificatore generalizzi nuove classi mai viste nel training set, dando soltanto un piccolo numero di esempi di ogni nuova classe. Tale rete apprende uno spazio metrico in cui la classificazione è attuata calcolando la distanza dal prototipo che rappresenta ciascuna classe. Più nel dettaglio la rete risulta così modellata: si consideri un support set $S = \{(x_1, y_1)...(x_n, y_n)\}, x_i \in \mathbb{R}^D$, con D dimensione del feature vector di esempio e $y_i \in \{1...k\}$ la corrispondente label. Indichiamo con S_k il set di esempi con label k. Si calcola per ogni classe il prototipo $c_k \in \mathbb{R}^M$ con una funzione di embedding $f_\phi : \mathbb{R}^D \to \mathbb{R}^M$, dove ϕ è un parametro learnable. Ogni prototipo è dato dalla media dei punti di supporto embedded appartenenti alla classe, ovvero: $c_k = \frac{1}{|S_k|} \sum_{(x_i,y_i)\in S_k} f_\phi(x_i)$. Si considera infine una funzione distanza $d : \mathbb{R}^M \times \mathbb{R}^M \to [0, +\infty)$ La rete dunque produce una distribuzione su classi per un punto query x basata sulla softmax delle distanze dei prototipi nello spazio di embedding nella seguente maniera: $P_\phi(y = k|x) = \frac{exp(-d(f_\phi(x),c_k))}{\sum_{k'} exp(-d(f_\phi(x),c'_k))}$ Il procedimento di learning si ha minimizzando $J(\phi) = -logP_\phi(y = k|x)$ della vera classe. Si espone successivamente lo pseudocodice per il calcolo della loss J adottato da Snell et al.

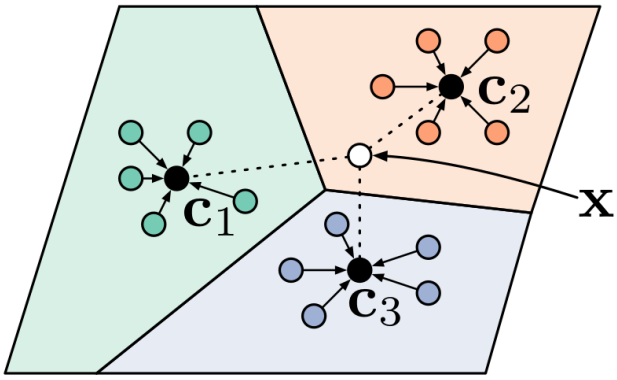

Figura 3. Prototypical Networks Few Shot

Input:
Training set $D = \{(x_1, y_1)...(x_N, y_N)\}, y_i \in \{1...K\}$
Indico con D_k il sottoinsieme di D con (x_i, y_i) t.c. $y_i = k$
Output:
J Loss per un episodio di training random generato

- N esempi training set
- K classi nel training set
- $N_C \leq K$ classi per episodio
- N_S esempi di supporto per classe
- N_Q esempi query per classe
- $RandomSample(S, N)$ insieme di N elementi scelti uniformemente da S senza essere reimmessi

Algorithm 1 Training episode loss computation for prototypical networks

$V \leftarrow RandomSample(\{1...K\}, N_C)$
for k in $\{1, ..., N_C\}$ **do**
  $S_k \leftarrow RandomSample(D_{Vk}, N_S)$
  $Q_k \leftarrow RandomSample(D_{Vk} \setminus S_k, N_Q)$
  $c_k \leftarrow \frac{1}{N_C} \sum_{(x_i,y_i)\in S_k} f_\phi(x_i)$
end for
$J \leftarrow 0$
for k in $\{1, ..., N_C\}$ **do**
  for (x, y) in Q_k **do**
    $J \leftarrow J + \frac{1}{N_C N_Q}[d(f_\phi(x), c_k) + log \sum_{k'} exp(-d(f_\phi(x), c_k))]$
  end for
end for

4. Backbones

I training e successivamente i test condotti sul dataset CUB prevedono la variazione della feature backbone di riferimento. La prima ad essere implementata è una backbone convoluzionale a 4 livelli con input size di 84×84 (Conv4), successivamente si utilizza anche una backbone convoluzionale con due blocchi addizionali senza pooling dopo la Conv4 (nominata Conv6) ed infine altre tre implementazioni di Residual Network: ResNet10, ResNet18 e ResNet34 [3] con input size di 224×224. L'algoritmo di few-shot con prototypical network migliora notevolmente in prestazioni quando la backbone diventa profonda. Nel dataset CUB il gap tra metodi allo stato dell'arte viene ridotto se la variazione intra class è ridotta da una backbone profonda [1]. Nel file *backbone.py* si ha l'implementazione delle diverse backbone realizzate, nel dettaglio:

- La Conv4 e la Conv6 sono realizzate grazie alla classe ConvNet che usa 4 blocchi convoluzionali con pooling e nel caso della Conv6 due blocchi convoluzionali aggiuntivi senza pooling. Ogni blocco convoluzionale è realizzato grazie alla libreria *torch.nn* ed ha come attributi una dimensione in ingresso, una di uscita, una convoluzione 2d, una batch norm 2d ed infine una funzione di attivazione ReLU. Nel caso di blocchi con pooling si ha l'aggiunta di un max pooling 2d sempre dalla libreria *torch.nn*.
- Le ResNet10, ResNet18 e ResNet34 sono composte da blocchi semplici caratterizzati da: dimensione di ingresso e uscita, due strati di convoluzione 2d e batch

norm ed infine due ReLU. Quando vengono istanziate essendo più complesse rispetto alle due reti viste precedentemente si ha che vengono specificati il numero di layer ed il numero di canali di uscita per ogni stadio.

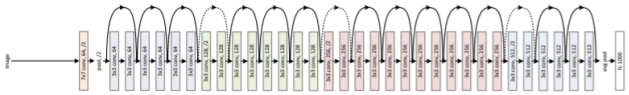

Figura 4. Residual Network con 34 parameter layers

L'istanziazione delle cinque differenti backbones si effettua chiamando i cinque relativi metodi che le inizializzano. La complessità delle ResNet è visibile (Fig. 5) anche osservando i parametri che si riferiscono al numero di layer e numero di canali di uscita degli stadi.

```
def Conv4():
    return ConvNet(4)

def Conv6():
    return ConvNet(6)

def ResNet10( flatten = True):
    return ResNet(SimpleBlock, [1,1,1,1],[64,128,256,512], flatten)

def ResNet18( flatten = True):
    return ResNet(SimpleBlock, [2,2,2,2],[64,128,256,512], flatten)

def ResNet34( flatten = True):
    return ResNet(SimpleBlock, [3,4,6,3],[64,128,256,512], flatten)
```

Figura 5. File backbone.py

5. Datasets e dettagli di esecuzione

I risultati sperimentali ottenuti si avvalgono di dataset specifici per realizzare fine-grained image classification, character recognition e cross domain character recognition. Per la prima categoria si è scelto il dataset CUB-200-2011 che è composto da 11.788 immagini divise in 200 classi differenti. Come in letteratura si ha la separazione delle classi in 100 per il base set, 50 per il validation e 50 per il novel. Per gli esperimenti di character recognition si è utilizzato il dataset Omniglot composto da 1623 caratteri provenienti da 50 lingue e modificato aumentando le classi inserendo rotazioni di 90, 180 e 270 gradi facendo risultare così 1692 classi novel. Le classi validation in questo esperimento sono usate per valutare le performance nel meta-training. Infine nel caso di cross domain character recognition sono stati utilizzati i dataset Omniglot senza caratteri latini e senza aggiunta di rotazioni nelle base classes (1597 classi base) e il dataset EMNIST che contiene un alfabeto a 10-digits upper e lower case in inglese (62 classi: 31 validation e 31 novel). Come evidenziato in Fig.6 EMNIST ha caratteri bianchi su sfondo nero, si è dunque invertito il colore a nero su bianco come in Omniglot. A supporto dei file immagine dei datasets sono stati realizzati dei file .json con nomi: base, val e novel tutti nello stesso formato: *"label_names": ["class0","class1",...], "image_names": ["filepath1","filepath2",...],"image_labels":[l1,l2,l3,...].*

Figura 6. Omniglot vs EMNIST, Cross domain character recognition

Più nel dettaglio si usa il validation set per selezionare gli episodi di training con migliore accuratezza con il dataset CUB. In ogni episodio vengono prelevate N classi per la $N - Way$ classification ($N = 5$ in tutti gli esperimenti). Per ogni classe vengono scelte k istanze etichettate come support set e 16 istanze per il query set per i $k - shot$ task. Nel meta training si effettuano 60.000 episodi (600 epoche x 100 iterazioni) nei casi di 1-shot, mentre 40.000 episodi (400 epoche x 100 iterazioni) nel caso di 5-shot. Nel meta testing si effettua infine una media su 600 esperimenti in ognuno dei quali si prelevano 5 classi da novel e in ogni classe si usa l'intero support set per addestrare un nuovo classificatore per 100 iterazioni con un batch size di 4. Il metodo implementato è ottimizzato e addestrato da zero con Adam con learning rate iniziale di 0.001 ($10^{-}3$).

5.1. Data Augmentation

Per migliorare la generalizzazione ed evitare overfitting si sintetizzano i dati attraverso trasformazioni che preservano le labels. Nel meta-training sul dataset CUB si applica una standard data augmentation che comprende: random crop, left-right flip, color jitter. Si è inoltre effettuato il meta training anche senza data augmentation per mostrare le differenze di prestazioni che ne risultano applicandola. Dato che i caratteri Omniglot sono bianchi e neri allineati al centro e sensitivi alla rotazione non si è usata la data augmentation nei relativi esperimenti.

6. Implementazione proposta

Il codice disponibile sul repository pubblico GitHub ha tre file principali: *train.py, save_features.py, test.py*. Esistono altri file python che supportano questi tre moduli nell'esecuzione in modo tale da separare i compiti di ogni file.

6.1. train.py

Il file *train.py* deve essere eseguito per primo ed è possibile intervenire e modificare alcune variabili per poter addestrare un differente esperimento. Si puo' modificare la variabile stringa del *model* della backbone per addestrare una *Conv4, Conv6...* come mostrato precedentemente. Vi è la variabile *dataset* che permette di selezionarne uno diverso tra CUB, Omniglot e cross_char, sono presenti anche variabili per l'*n-way* del train e del test, l'*n-shot* ed infine la frequenza di salvataggio del modello realizzato nelle 600 (per 1-shot) o 400 (5-shot) epoche. Dopo aver impostato queste variabili in base all'esperimento che si vuole condurre il codice seguente imposta e va a selezionare il base set, il validation ed il query set automaticamente in base al dataset selezionato ed il numero di *n-way*. Si istanzia infine il modello ProtoNet passando la backbone selezionata e si esegue la restante computazione su GPU grazie a *model.cuda()*. I risultati vengono successivamente inseriti dentro la cartella checkpoints e a sua volta dentro la cartella col nome del dataset di riferimento. Nei successivi esperimenti viene effettuato il salvataggio del modello ogni 50 epoche e si tiene sempre in memoria il miglior modello trovato nelle epoche. Se il training dovesse interrompersi per problemi si può impostare la variabile booleana *resume* a True in modo tale da ricominciare il training eseguendo nuovamente *train.py* dal più recente salvataggio effettuato in precedenza. Nella funzione *train* si istanzia l'ottimizzatore Adam dalla libreria *torch.optim*, successivamente si inizia ad addestrare il modello in un ciclo di epoche precedentemente impostate. La durata dell'addestramento effettuato nel *train.py* dipende dal dataset scelto, dal numero n-way e k-shot, dall'hardware e molteplici fattori. A fine esecuzione il miglior modello ed i vari modelli intermedi sono disponibili nella cartella dei checkpoints come file *.tar*. Importantissimo per il funzionamento del processo di training è il codice di *protonet.py* che realizza la prototypical network descritta da Senll et al. [2].

6.2. save_features.py

Il seguente file python permette di salvare le feature estratte precedentemente dal classification layer per incrementare la velocità di test successiva. Dopo aver condotto il training di un determinato esperimento con il file *train.py* è necessario eseguire il file per salvare le features all'interno dell'omonima cartella *features* in un formato *.hdf5*.

6.3. test.py

Dopo l'addestramento ed il salvataggio delle features del dataset CUB con tutte le backbone proposte, in configurazione 1 e 5-shot, con e senza data augmentation e del dataset Omniglot e cross_char (Omniglot→ EMNIST) con la backbone Conv4 e configurazione 1 e 5 shot è possibile eseguire il file *test.py*. Si ha dunque il caricamento con torch dei modelli precedentemente addestrati presenti nella directory checkpoints in ordine in base al test da condurre. Il caricamento è scritto all'interno di un ciclo for che permette di eseguire in una singola esecuzione del *test.py* il test di tutti i modelli addestrati precedentemente. Nel file si valutano delle classi estratte randomicamente da novel ed infine si attribuisce uno score di media di predizione su 600 iterazioni, valore fissato nei successivi test ma modificabile nella variabile *iter_num*. Per ogni test condotto vengono riportati i valori percentuali di predizione corretta mediata sulle 600 iterazioni ed il relativo intervallo di confidenza al 95% su tale media. I risultati vengono organizzati in tabelle nella finestra di output e su grafici che mostrano e comparano le prestazioni. Il codice non ha bisogno di modifiche da parte dell'utilizzatore poichè esegue tutti i test dei modelli addestrati precedentemente e fornisce una visuale unificata dei risultati grazie alle tabelle e ai grafici 7, 8, 9, 10, 11, 12, 13.

Backbone:	Conv4	Conv6	ResNet10	ResNet18	ResNet34
CUB 5W-1S NoAug	52.16 +- 0.92	53.2 +- 0.95	59.97 +- 0.93	61.52 +- 0.97	59.69 +- 0.98
CUB 5W-5S NoAug	67.39 +- 0.72	67.71 +- 0.74	72.46 +- 0.73	73.66 +- 0.69	74.43 +- 0.72
CUB 5W-1S Aug	50.86 +- 0.92	65.67 +- 1.01	73.16 +- 0.87	74.18 +- 0.9	74.56 +- 0.92
CUB 5W-5S Aug	76.37 +- 0.69	81.74 +- 0.61	85.83 +- 0.49	86.51 +-0.51	87.91 +- 0.46

Figura 7. Output dei risultati CUB del file test.py

Omni 5W-1S	Omni 5W-5S	Omni->Emnist 5W-1S	Omni->Emnist 5W-5S
97.92 +- 0.28	99.37 +- 0.12	73.29 +- 0.78	86.95 +- 0.58

Figura 8. Output dei risultati Omniglot e cross_char del file test.py

I tre differenti esperimenti implementati in *test.py* sono inseriti all'interno di tre funzioni differenti: *testCUB(), testOmniglotAndCross(), testFurtherAdaptation()*

6.4. protonet.py

Nel file è presente la classe che realizza la prototypical network caratterizzata da: una cross entropy loss, i parametri di $n-way, n-support, n-query$ e da feature con la relativa dimensione. Sono stati implementati i metodi di: forward per andare in un layer successivo, parse delle feature, correttezza: utilizzata dal loop di test per ritornare un valore di accurazezza, train loop che ottimizza con Adam e calcola la nuova loss, setter per il forward e la forward loss. Infine è presente una funzione che calcola la distanza euclidea tra x ed y.

7. Esecuzione del codice

Il codice deve seguire la pipeline di train→salvataggio delle features per i dataset CUB, Omniglot, e cross_char (Omniglot→EMNIST) solamente alla fine è possibile eseguire il test che valuta e riassume tutti i risultati ottenuti.
L'esecuzione del file di train risulta la più lunga per attesa temporale, gli addestramenti impiegano anche svariate ore per essere completati nelle loro 400 o 600 epoche. Il salvataggio delle features così come il test risultano invece immediati e nell'arco di qualche minuto terminano la loro esecuzione. Gli output dei tre file sono: per il train il modello addestrato per l'esperimento richiesto. Per il salvataggio delle features un file *.hdf5* con le features del modello addestrato. Infine per il test una tabella riassuntiva per CUB al variare delle 5 backbone, nelle configurazioni 1-5 shot (5 way fisso) con e senza data augmentation, una tabella per Omniglot e cross_char al variare di 1-5 shot (5 way fisso) ed infine una tabella con i risultati di ulteriori adattamenti della protonet in configurazione 5 shot ResNet18 e dataset CUB. Per il dataset CUB l'output di test dà anche i grafici riportati nel capitolo successivo analoghi a quelli riportati in letteratura grazie alla libreria *matplotlib*.

8. Risultati e prestazioni

I risultati di seguito riportati sono stati estratti a seguito dell'addestramento della prototypical network negli esperimenti: CUB 1 shot Conv4 no aug, CUB 5 shot Conv4 no aug, CUB 1 shot Conv4 aug, CUB 5 shot Conv4 aug, ... ,CUB 5 shot ResNet34 aug (20 training in totale) e Omniglot 1 shot Conv4, Omniglot 5 shot Conv4, CrossChar 1 shot Conv4, CrossChar 5 shot Conv4. I valori di seguito riportati sono coerenti con quelli mostrati nell'articolo di Chen et al. [1] e possono essere usati per trarre importanti considerazioni.

8.1. Aumento della profondità di rete

- Gli esperimenti sul test CUB, destinato a fine-grained image classification hanno permesso di confrontare le performance dell'algoritmo di few-shot learning implementato al variare della profondità delle cinque backbone. L'esecuzione del file *test.py* grazie alla funzione testCUB valuta le performance ed i risultati di questo esperimento. I primi training condotti sono stati effettuati in una configurazione 5-way 1-shot con data augmentation al variare delle backbone, i risultati evidenziati dal grafico in Fig.9 mostrano un notevole miglioramento all'aumentare della profondità della backbone. La capacità della rete di attribuire una label corretta ad una nuova classe migliora sempre di più a discapito però del tempo di training che anch'esso aumenta con una backbone più profonda. I risultati dai test condotti sono stati mediati su 600 iterazioni e nel train l'addestramento è stato effettuato per 600 epoche.

- Il training nella configurazione 5-way 5-shot nelle 400 epoche si dimostra più robusto nella fase di testing, i risultati visibili nel grafico in Fig.10 mostrano sempre una crescita di performance anche in questo caso. I risultati appena mostrati risultano fedeli a quelli riportati da Chen et al. e sono competitivi con i metodi di meta learning allo stato dell'arte, in queste configurazioni la differenza tra metodi attuali decresce sempre di più all'aumentare della profondità della backbone utilizzata [1]. Per mostrare infine l'efficacia della data augmentation sono stati effettuati inoltre dei training e test in esperimenti analoghi ai precedenti che non ne prevedessero l'utilizzo.

- Confrontando i risultati nel grafico in Fig.9 e Fig.11 si ha che il primo test con backbone Conv4 risulta anche migliore nell'esperimento senza data augmentation anche se con risultati relativamente bassi. La crescita di performance al variare delle backbone però risulta maggiore nel caso con data augmentation che diventa dunque la scelta vincente.

- Analogamente considerando l'esperimento in Fig.10 e Fig.12 si ha che le performance nel caso di data augmentation sono migliori, i risultati ottenuti senza con 5-way 5-shot sono paragonabili ai risultati con data augmentation nel caso 5-way 1-shot che penalizza notevolmente il training a causa di un solo esempio per cinque classi.

I risultati sono infine riassunti nella successiva Tab.1 analoga a quella di output del file *train.py* in Fig.7.

8.2. Differenze di dominio

Per sperimentare le performance dell'algoritmo di few-shot learning con prototypical network infine sono stati realizzati 4 scenari differenti due dei quali in una configurazione cross domain ovvero il set di novel è stato realizzato con un dataset differente in modo tale da rendere più complicata per l'algoritmo la classificazione. I risultati con il singolo dataset Omiglot risultano elevati e si classifica correttamente con un ottimo punteggio, invece i risultati con i due dataset Omniglot ed EMNIST sono competitivi e coerenti con quelli riportati da Chen et al. [1]. Tutti i risultati riassunti nella Tab.2 sono stati ottenuti con la backbone fissa più semplice Conv4 e in configurazione senza data augmentation con input size di 28×28.

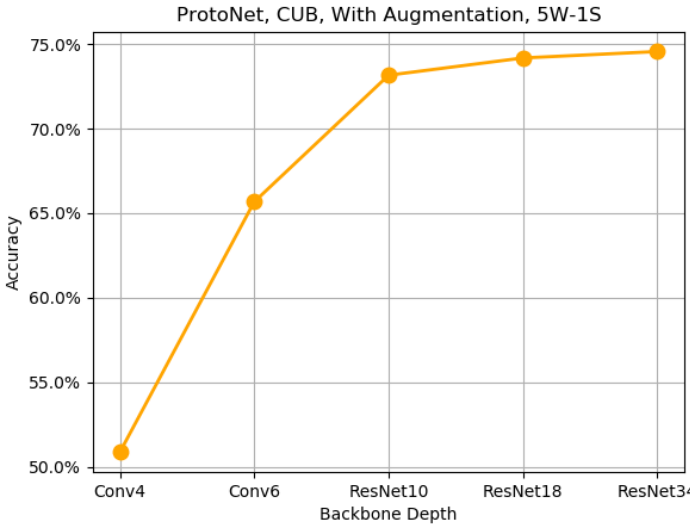

Figura 9. Test al variare della backbone, 5-way 1-shot

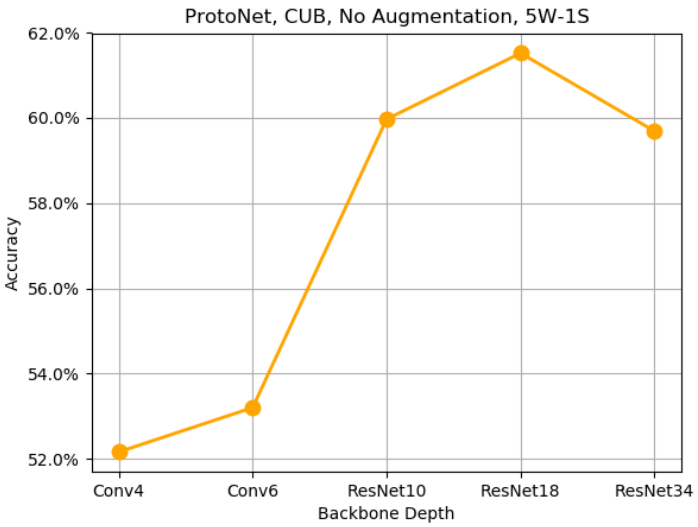

Figura 11. Test al variare della backbone, no augmentation

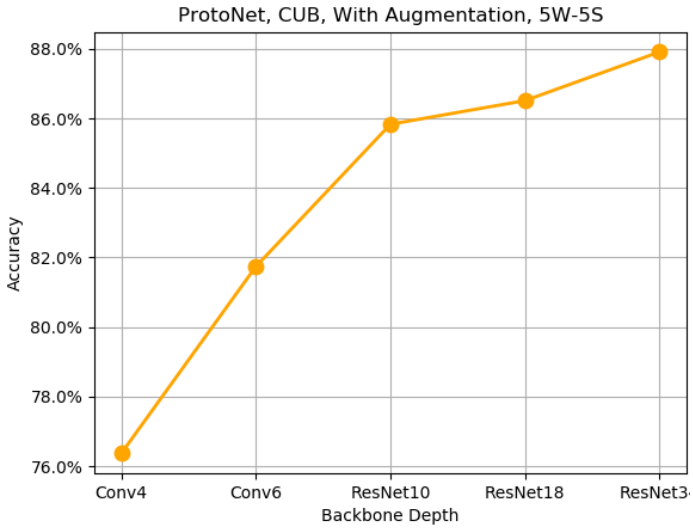

Figura 10. Test al variare della backbone, 5-way 5-shot

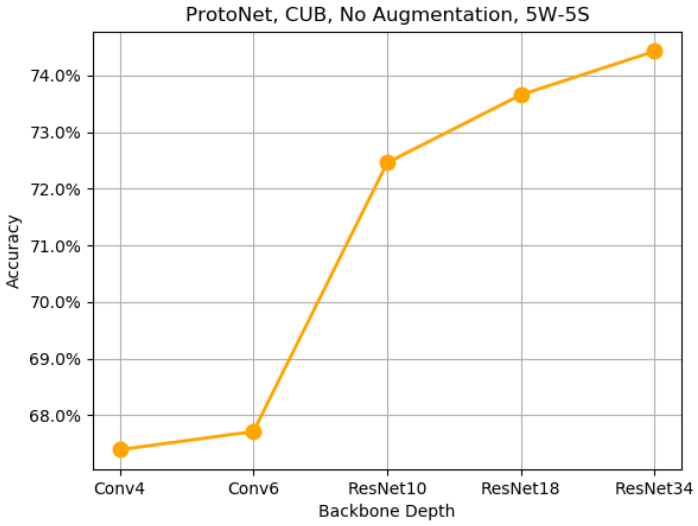

Figura 12. Test al variare della backbone, no augmentation

Backbone:	Conv4	Conv6	ResNet10	ResNet18	ResNet34
CUB 5W-1S NoAug	52.16 +- 0.92	53.2 +- 0.95	59.97 +- 0.93	61.52 +- 0.97	59.69 +- 0.98
CUB 5W-5S NoAug	67.39 +- 0.72	67.71 +- 0.74	72.46 +- 0.73	73.66 +- 0.69	74.43 +- 0.72
CUB 5W-1S Aug	50.86 +- 0.92	65.67 +- 1.01	73.16 +- 0.87	74.18 +- 0.9	74.56 +- 0.92
CUB 5W-5S Aug	76.37 +- 0.69	81.74 +- 0.61	85.83 +- 0.49	86.77 +- 0.49	87.91 +- 0.46

Tabella 1. Perofrmance di ProtoNet al variare della della backbone

Omni 5W-1S	Omni 5W-5S	Omni → Emnist 5W-1S	Omni → Emnist 5W-5S
97.92 +- 0.28	99.37 +- 0.12	73.29 +- 0.78	86.95 +- 0.58

Tabella 2. Character recognition vs cross domain

CUB ResNet18 5W-5S NoAdaptation	CUB ResNet18 5W-5S Adaptation
86.77 +- 0.49	86.51 +- 0.5

Tabella 3. Confronto con adattamenti aggiuntivi

8.3. Ulteriori adattamenti

Si riportano infine i risultati sul datset CUB 5 shot ResNet18 con Adaptation, ovvero fissando le features e addestrando un nuovo classificatore softmax in questo caso utilizzando come ottimizzatore SGD con momentum 0.9 e weight decay 0.001. La prototypical net in letteratura [1] dimostra che le performance degradano in scenari con poche differenze di dominio, il risultato visibile nell'analogo grafico di Fig.13 ed in Tab.3 mostra che le performance sono equivalenti nei due casi.

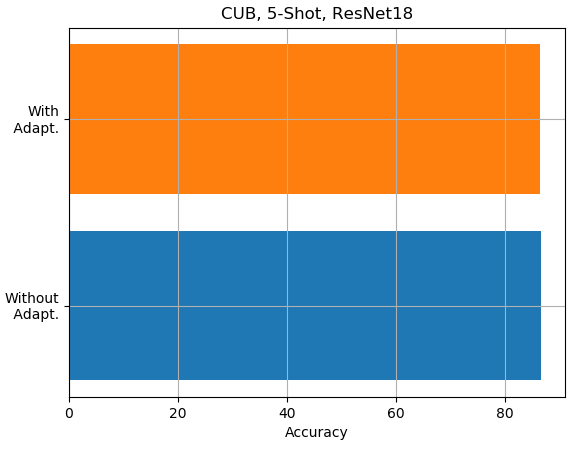

Figura 13. Ulteriori adattamenti

9. Considerazioni finali

I risultati riportati ed i test condotti mostrano il potenziale delle prototypical network in uno scenario con data limitati. I valori visti mostrano che si riescono ad ottenere performance soddisfacenti in condizioni complesse e soprattutto anche tra cross dataset quando le immagini proposte da novel sono di classi mai viste, completamente differenti da tutte le altre e di un dataset estraneo a quello usato per addestrare. Il metodo implementato raggiunge performance competitive paragonato agli altri algoritmi di meta learning di few shot classification allo stato dell'arte con profonde backbone. Il codice reso pubblico su GitHub permette di eseguire e confermare i risultati ottenuti negli esperimenti condotti e in futuro ampliare i training e i test con l'aggiunta di nuovi dataset per ulteriori esperimenti. I modelli addestrati per gli esperimenti sono resi invece disponibili su Google Drive.

Riferimenti bibliografici

[1] W.-Y. Chen, Y.-C. Liu, Z. Kira, Y.-C. Wang, and J.-B. Huang. A closer look at few-shot classification, 2019.

[2] R. S. Z. Jake Snell, Kevin Swersky. Prototypical networks for few-shot learning, 2017. https://arxiv.org/abs/1703.05175.

[3] S. R. Kaiming He, Xiangyu Zhang and J. Sun. Deep residual learning for image recognition, 2016.

[4] P. M. A. S. Timothy Hospedales, Antreas Antoniou. Meta-learning in neural networks: A survey, 2020. https://arxiv.org/abs/2004.05439.