Università degli Studi di Catania
Dipartimento di Matematica ed Informatica
Corso di Laurea Magistrale in Informatica

CLASSIFICAZIONE PRODOTTI BARILLA STUDIO TRAMITE CONVOLUTIONAL NEURAL NETWORK

Prof. Giovanni Maria Farinella

Alessandro Lauria
Francesco Anastasio

Sommario

Introduzione

1.1 Barilla Studio

Barilla Studio è un'iniziativa Barilla che mette a disposizione, gratuitamente, le immagini di tutti i prodotti forniti dall'azienda e dalle sub-aziende ad essa associate. Le immagini vengono date ad alta risoluzione e con sfondo trasparente, in formato png. Per ogni prodotto è presentata almeno un'immagine. Quelli che ne hanno più di una riportano la vista frontale, laterale sinistra e laterale destra, anche se in alcuni casi sono forniti solamente due esempi.

1.2 Convolutional Neural Network

Le Convolutional Neural Networks sono una categoria di reti neurali mostratesi molto efficaci per l'image recognition e l'image classification. Le reti neurali classiche non risultano adatte al processamento di immagini in quanto queste possono essere molto grandi, complesse e potrebbero essere necessari parecchi layer. Quando il numero di input e di layer cresce lo fa anche il numero di parametri. L'idea è, quindi, di ridurre il numero di parametri per ogni layer e creare modelli più profondi.

Nelle CNN la moltiplicazione matriciale viene sostituita con la convoluzione. Così facendo ogni nodo di un layer sarà totalmente connesso solo a una regione dell'immagine e i pesi da imparare saranno i valori dei kernel convolutivi.

1.3 PyTorch

PyTorch è una libreria di machine learning per il linguaggio di programmazione Python basata sulla libreria Torch. Inizialmente sviluppata dal gruppo di ricerca in Intelligenza Artificiale di Facebook, fornisce due features principali ad alto livello:

- Tensor computing con forte accelerazione tramite uso di GPU
- Deep Neural Network costruite su sistema di auto differenziazione

1.4 Scopo del progetto

Il progetto si propone lo scopo di sfruttare reti neurali convoluzionali note per implementare un algoritmo in grado di discernere, data una foto, che tipo di prodotto Barilla è stato immortalato. Per implementare l'algoritmo sarà utilizzata la libreria PyTorch e il linguaggio di programmazione Python. Verranno adottate diverse tecniche per limare e migliorare i risultati ottenuti. La principale difficoltà risiede nel dover distinguere un elevato numero di classi avendo pochi esempi per classe (a volte solamente uno).

Modelli

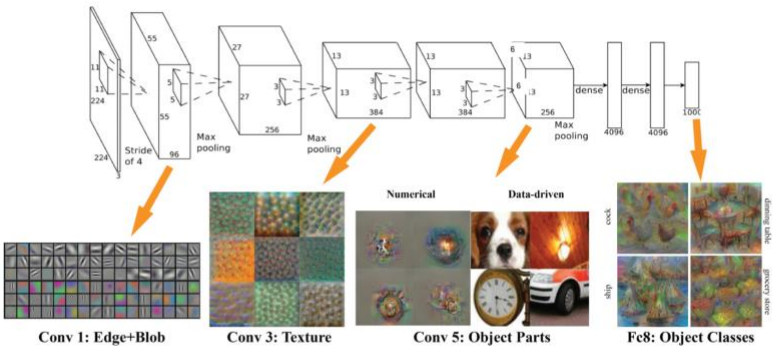
2.1 AlexNet

2.1.1 Introduzione

AlexNet è una Convolutional Neural Network progettata da Alex Krizhevsky. La rete ha vinto il ILSVRC nel 2012. Risolse il problema di classificazione dove l'input era una immagine di una certa classe su 1000 e l'output un vettore di 1000 elementi rappresentanti una distribuzione di probabilità sulle classi.
L'input atteso è una immagine di dimensioni 256x256 a 3 canali (RGB).

2.1.2 Architettura

AlexNet è molto più grande delle precedenti CNN usate per risolvere task di Computer Vision. Ha 60 milioni di parametri e 650.000 neuroni.

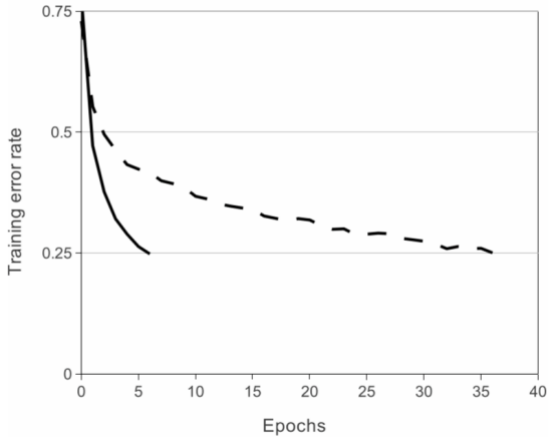



```
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3,   , kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96,256 , kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384,384 , kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 6 * 6)
        x = self.classifier(x)
        return x
```

La rete è composta da 5 layer convolutivi e 3 layer fully-connected. Il primo layer convolutivo contiene 96 kernel di dimensione 11x11x3, dove il 3 è dovuto al numero di canali dell'immagine.

All'output di ogni layer convolutivo è applicata la ReLU. I primi due layer convolutivi sono seguiti da layer di Max Pooling, mentre gli altri 3 sono direttamente connessi tra loro. L'ultimo layer convolutivo è pure seguito da un Max Pooling, il quale output viene dato a una serie di due layer fully-connected. Per classificare, in fine, l'output dell'ultimo layer viene passato ad un Softmax. L'overfitting viene ridotto con due fasi di DropOut, che permettono di spegnere e accendere i neuroni con probabilità del 50%.

È stato dimostrato che l'utilizzo della ReLU in AlexNet, come funzione di attivazione, permette di raggiungere il 25% di train error rate 6 volte più velocemente rispetto le più tradizionali TanH e Sigmoid.



2.2 SqueezeNet

2.2.1 Introduzione

SqueezeNet è una Neural Network rilasciata nel 2016. È stata sviluppata dai ricercatori della DeepScale, University of California, Berkeley e Stanford University. Il goal degli autori è stato creare una rete neurale piccola e con pochi parametri mantenendo la qualità dei risultati.

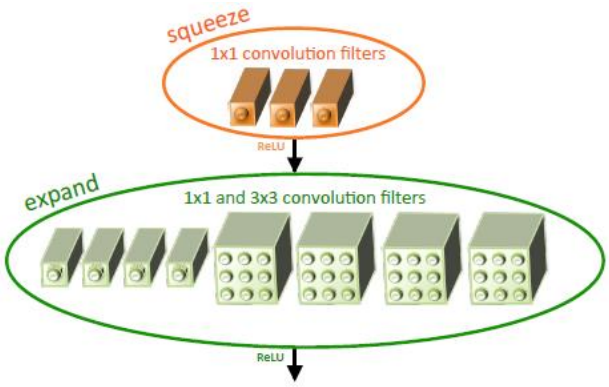
La rete venne presentata come "in grado di ottenere accuracy ai livelli di AlexNet ma con molti meno parametri". È importante notare che nonostante

questa premessa, si tratta di una rete totalmente diversa da AlexNet, avente in comune con essa solo le percentuali di accuracy ottenute sul dataset ImageNet.

2.2.2 Architettura

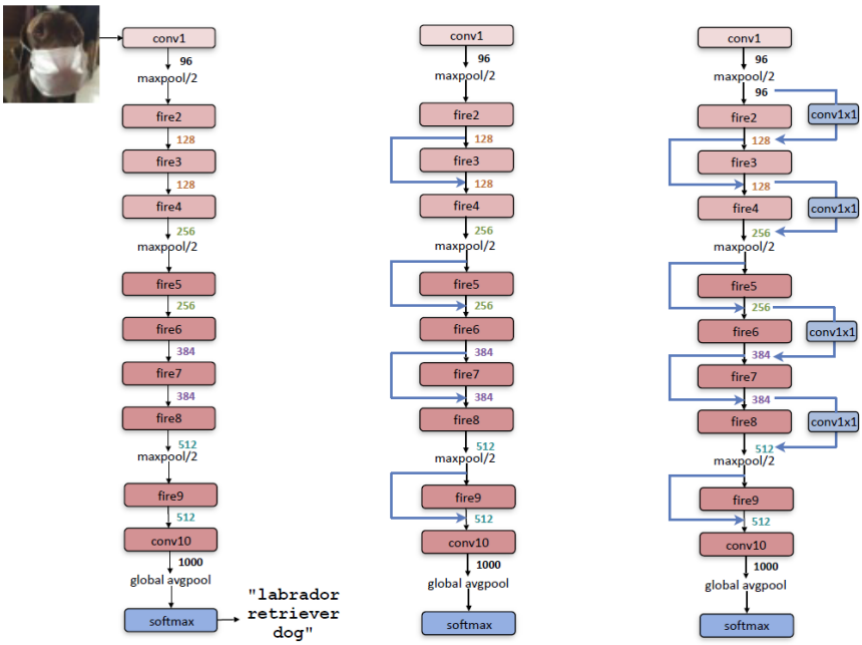
Fire module:

Un modulo che verrà utilizzato nella rete finale. È composto da uno strato di convoluzione che ha solamente kernel 1x1, il cui output viene passato a uno strato convolutivo di espansione formato da un mix di kernel 1x1 e 3x3.



SqueezeNet Architecture:

- Inizia con un layer di convoluzione seguito da 8 Fire Module e finisce con un layer di convoluzione finale.
- Il numero di kernel per Fire Module viene incrementato gradualmente man mano che si procede nella rete.
- Viene applicato il Max Pooling, con stride 2, dopo il layer conv1, fire4, fire8, conv10.
- Può essere applicato un bypass semplice o un bypass complesso, ispirato a ResNet.


conv1
96
maxpool/2
fire2
128
fire3
128
fire4
256
maxpool/2
fire5
256
fire6
384
fire7
384
fire8
512
maxpool/2
fire9
512
conv10
1000
global avgpool
softmax
"labrador retriever dog"
conv1
96
maxpool/2
fire2
128
fire3
128
fire4
256
maxpool/2
fire5
256
fire6
384
fire7
384
fire8
512
maxpool/2
fire9
512
conv10
1000
global avgpool
softmax
conv1
96
maxpool/2
96
fire2
conv1x1
128
fire3
128
fire4
conv1x1
256
maxpool/2
fire5
256
fire6
conv1x1
384
fire7
384
fire8
conv1x1
512
maxpool/2
fire9
512
conv10
1000
global avgpool
softmax

Implementazione

3.1 Creazione Dataset

Tramite l'implementazione di un algoritmo di scraping sono state scaricate le immagini presenti nel sito www.barillastudio.it mantenendo la struttura gerarchica originale (collocando le foto in cartelle con lo stesso nome della directory del sito).

Di seguito si è proceduto alla scrittura di codice che permettesse di rinominare le foto col path assoluto (a partire dalla cartella principale), passo necessario per riuscire a classificare correttamente in seguito. Un ulteriore script si è occupato di suddividere le foto in Train e Test, cercando di mantenere una divisione 70% - 30%, per quanto possibile. È stato inoltre necessario generare file .txt che contenessero in ogni riga una coppia (nome_file, classe), dove "classe" rappresenta il numero corrispondente alla classe.

- train.txt contiene le coppie per le immagini di Train
- test.txt le coppie per il Test
- classes.txt presenta invece le coppie (nome_classe, classe), utilizzato per mappare i valori predetti dalla rete con il nome della classe predetta.

Nella parte finale dell'esperimento è stato implementato anche un algoritmo che realizzasse data augmentation sul dataset ruotando le immagini a 45° alla volta e salvandole in un nuovo dataset.

3.2 K-NN

Per tutti i modelli, oltre a valutare l'accuracy della rete allenata, si è scelto di sfruttare un k-nn, fittato con le features estratte dal modello, per capire se questo permettesse di ottenere risultati migliori rispetto la classificazione tramite Softmax. Sono stati utilizzati 1-nn, 3-nn e 5-nn, scelti anche in base al

numero minimo di esempi per classe disponibili. A volte è stato effettivamente utile per guadagnare alcuni punti percentuali di accuracy, mentre in altre ha peggiorato la situazione.

3.3 Allenamento senza modello precaricato

L'approccio iniziale, per prendere confidenza con il task e capire che risultati fossero ottenibili, è stato quello di utilizzare AlexNet senza sfruttare il modello pre-allenato fornito da PyTorch. La rete è stata quindi allenata da zero sul dataset di Train per 150 epoche. L'esperimento è stato ripetuto sia per il dataset classico che per quello con data augmentation. Ovviamente è stato necessario ridefinire il numero di neuroni dell'ultimo layer in base al numero di classi dell'attuale task.

```
net = models.alexnet()
net.classifier[6] = nn.Linear(4096, 153)
name_save = "retraining_alexnet_153classes_fulldataset"
```

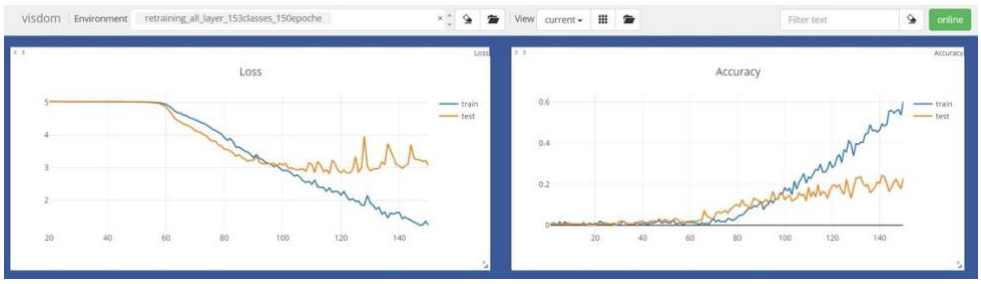
I risultati ottenuti dopo l'allenamento su **dataset standard** sono i seguenti:

Train:

Accuracy: 0.73
Precision_score: 0.662844611528822
Recall_score: 0.7236842105263158
F1_Score: 0.6662242728032202

Test:

Accuracy: 0.23
Precision_score: 0.14742324561403508
Recall_score: 0.23026315789473684
F1_Score: 0.1681286549707602



Quello che i risultati mostrano è un ampio overfitting sul Train con scarsi valori in Test. Ciò è probabilmente dovuto al ridotto numero di esempi.

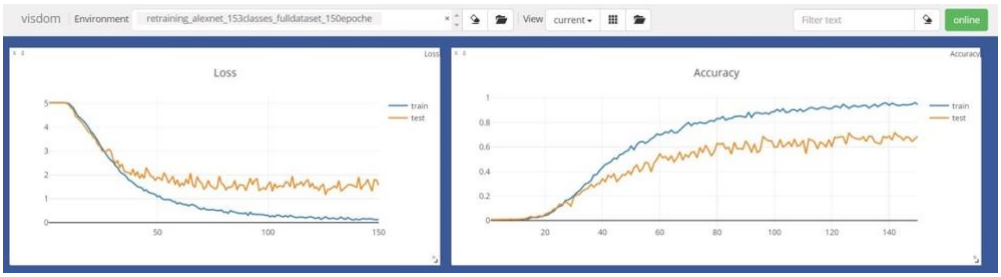
I risultati ottenuti dopo l'allenamento su **dataset aumentato** sono, invece, i seguenti:

Train:

Accuracy: 0.98
Precision_score: 0.9782688179864785
Recall_score: 0.9769736842105263
F1_Score: 0.9739270228685475

Test:

Accuracy: 0.69
Precision_score: 0.7186494595768009
Recall_score: 0.6875
F1_Score: 0.6624812623629627



Anche in questo caso si è presentato dell'overfitting sul Train, anche se in maniera decisamente più lieve, visto che è stato ottenuto un buon risultato in Test.

Sono state sperimentate altre tecniche per vedere se si potessero migliorare i risultati e ridurre l'overfitting.

3.4 Fine tuning sull'ultimo layer

La tecnica del fine tuning consiste nell'utilizzare un modello allenato per un certo tipo di task e riallenare solamente gli ultimi layer per adattare la rete ad un nuovo task simile al precedente.

Per poter allenare solo l'ultimo layer è stato scaricato il modello pre-allenato fornito da PyTorch ed è stato settato requires_grad di ogni parametro a False, riassegnando l'ultimo layer (i cui parametri di default avranno valore True).

```
net = models.alexnet(pretrained=True)
for param in net.parameters():
    param.requires_grad = False

net.classifier[6] = nn.Linear(4096, num_classes)
sum([p.numel() for p in net.parameters()])
name_save = "last_layer_training_153classes"
```

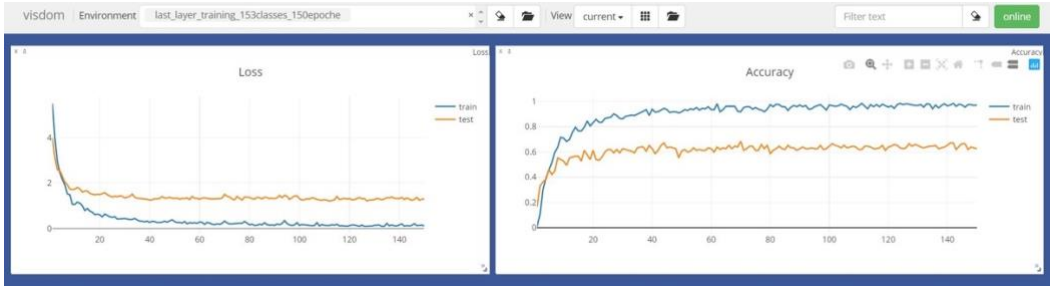
I risultati ottenuti dopo l'allenamento su **dataset standard** sono i seguenti:

Train:

Accuracy: 0.99
Precision_score: 0.9901315789473685
Recall_score: 0.993421052631579
F1_Score: 0.9912280701754386

Test:

Accuracy: 0.62
Precision_score: 0.5679824561403508
Recall_score: 0.625
F1_Score: 0.5853070175438596



Come si può vedere non è stato eliminato l'overfitting nel Train ma si è raggiunto un risultato di Test decisamente migliore rispetto lo 0.23 dell'approccio precedente.

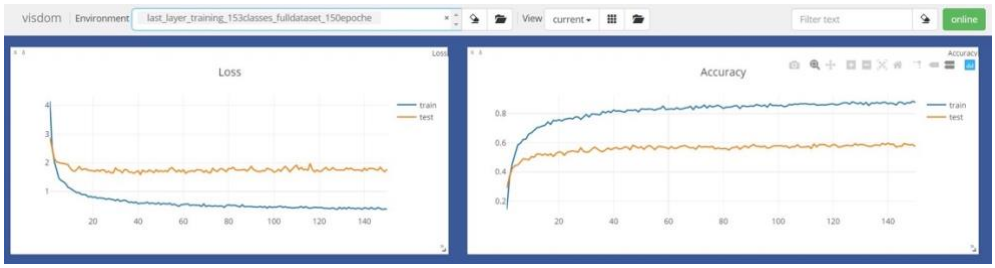
I risultati ottenuti dopo l'allenamento su **dataset aumentato**, invece, sono i seguenti:

Train:

Accuracy: 0.99
Precision_score: 0.9891750790805703
Recall_score: 0.9876644736842105
F1_Score: 0.9876259287459211

Test:

Accuracy: 0.57
Precision_score: 0.6328272312447757
Recall_score: 0.5740131578947368
F1_Score: 0.5702898998208803



I valori ottenuti non sono migliorati e l'overfitting è leggermente aumentato. Confrontando col risultato dell'approccio precedente sul dataset aumentato è chiaro che le performance sono peggiorate.

3.5 Fine tuning sugli ultimi due layer

Si è voluto quindi procedere a riallenare gli ultimi due layer, anziché solamente uno, per vedere se, aumentando il numero di parametri da allenare, si potesse guadagnare qualche punto percentuale di accuracy in più.

```
net = models.alexnet(pretrained=True)
for param in net.parameters():
    param.requires_grad = False

net.classifier[4] = nn.Linear(4096, 4096)
net.classifier[6] = nn.Linear(4096, 153)
sum([p.numel() for p in net.parameters()])
name_save = "last_two_layers_training_153classes"
```

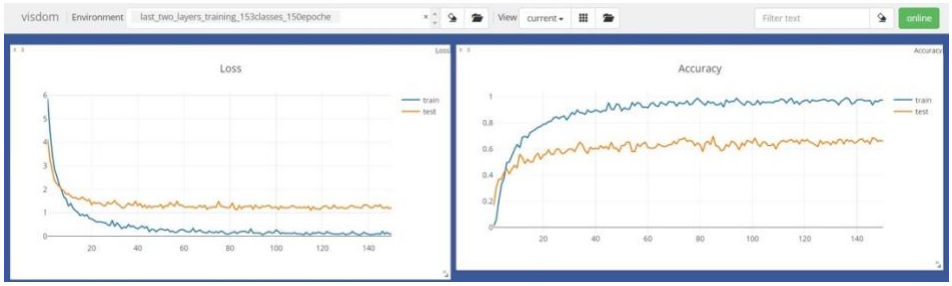
I risultati ottenuti dopo l'allenamento su **dataset standard** sono i seguenti:

Train:

Accuracy: 1.00
Precision_score: 0.9978070175438598
Recall_score: 0.996710526315789
F1_Score: 0.9964912280701754

Test:

Accuracy: 0.66
Precision_score: 0.6120614035087719
Recall_score: 0.6578947368421053
F1_Score: 0.62468671679198



Come prima, l'overfitting rimane presente e, a parte qualche lieve miglioramento in Test, non si è raggiunto un risultato molto differente all'allenamento di un solo layer.

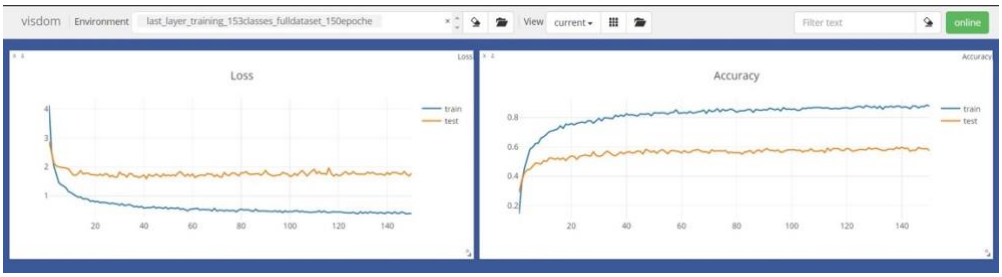
I risultati ottenuti dopo l'allenamento su **dataset aumentato**, invece, sono i seguenti:

Train:

Accuracy: 1.00
Precision_score: 0.9976780185758513
Recall_score: 0.9975328947368421
F1_Score: 0.9975163348253332

Test:

Accuracy: 0.60
Precision_score: 0.6332078694303722
Recall_score: 0.5970394736842105
F1_Score: 0.5956866784689481



Stessa osservazione precedente. Si hanno avuto anche dei peggioramenti in Test.

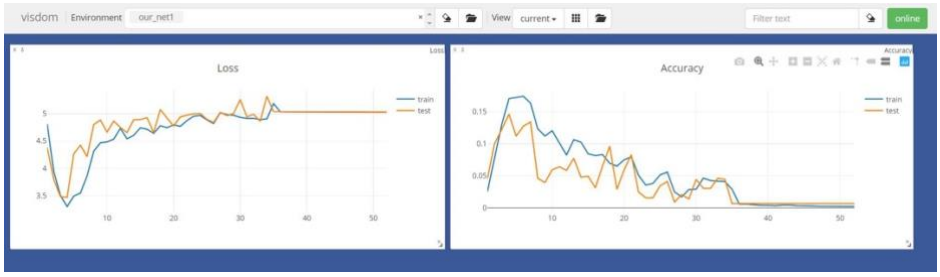
3.6 Aggiunta Multi Layer Perceptron ad AlexNet

Rendendosi conto di non star ottenendo risultati soddisfacenti è stato necessario provare a cambiare approccio. L'idea perseguita consiste nello sfruttare AlexNet, con modello pre-allenato da PyTorch, per estrarre le features ed agganciare un Multi Layer Perceptron che prenda queste features in input e si occupi di fare la classificazione. In tal modo si vuole sfruttare la robustezza di AlexNet, già allenata, come punto di partenza per allenare una nuova rete che si occupi della classificazione partendo da features già consolidate.
Il Multi layer Perceptron è stato così strutturato:

```
class Multilayer_percetron(nn.Module):
    def _init_(self):
        super()._init_()
        self.features = nn.Sequential(
            nn.Linear(1000, 256, bias=True),
            nn.Linear(256, 184, bias=True),
            nn.ReLU(inplace = True),
            nn.Linear(184, 153, bias=True)
        )
        self.classifier =  nn.Sequential(
            nn.Softmax(dim=1)
        )
    def forward(self, xb, train = True):
        x = xb.view(-1,1000)
        x = self.features(x)
        if train == False:
            x = self.classifier(x)
            pred = x.data.cpu().numpy().copy()
            print(pred)
        return x
```

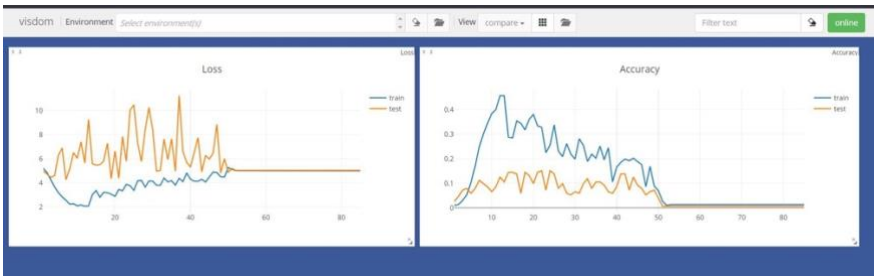
I risultati ottenuti dopo l'allenamento su **dataset standard** sono i seguenti:

Accuracy Train: 0.01
Accuracy Test: 0.00



I risultati ottenuti dopo l'allenamento su **dataset aumentato**, invece, sono i seguenti:

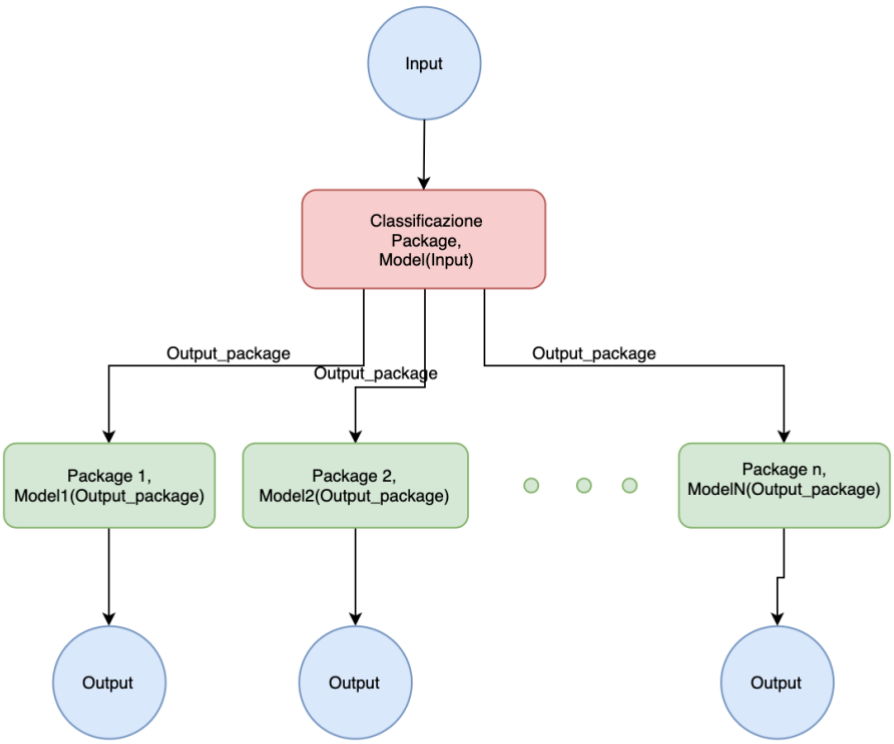
Accuracy train: 0.00
Accuracy test : 0.00



Sono state provate diverse configurazioni ma nessuna ha portato a risultati soddisfacenti. Ciò è probabilmente dovuto all'attuale poca esperienza degli sviluppatori nel creare reti neurali da zero. Dati gli esiti si è quindi deciso di percorrere altre strade.

3.7 TreeNet

L'approccio finale, che ha permesso di ottenere i risultati migliori, sfrutta un piccolo trick. Risulta evidente che le reti precedentemente testate faticassero a superare una certa soglia di accuracy in Test, nell'intorno del 60%-70%. Guardando bene il dataset si è notato che i prodotti aventi lo stesso tipo di package presentano differenze minimali, caratterizzate per lo più dal nome scritto sulla confezione e dal piccolo disegno del prodotto. Da tale osservazione si è dedotto che la rete imparasse maggiormente a distinguere la confezione rispetto il tipo di prodotto e perdesse di conseguenza i dettagli semantici utili a discernere, dato lo stesso package, le differenze tra i prodotti.
Data questa intuizione si è scelto di suddividere il task in sotto-task. Non si avrà più un unico modello allenato per classificare le 153 classi, ma più modelli, organizzati ad albero.
Il nodo radice sarà un modello allenato per classificare il tipo di package, il suo output determinerà quale nodo foglia richiamare. Il numero di nodi foglia corrisponde al numero di package e ognuno di questi è caratterizzato da un modello allenato per classificare i prodotti aventi lo stesso package.

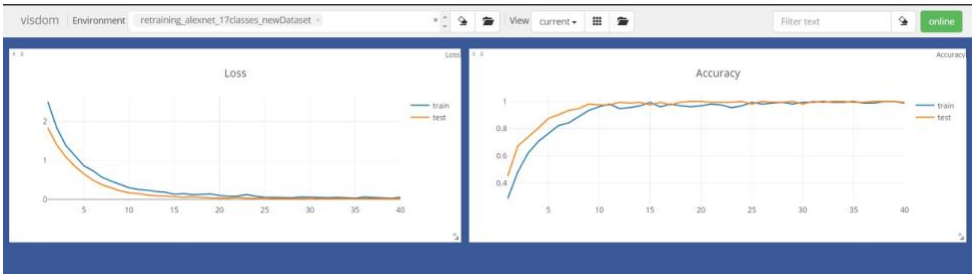


Data la struttura ad albero si è scelto di chiamare l'algoritmo TreeNet.

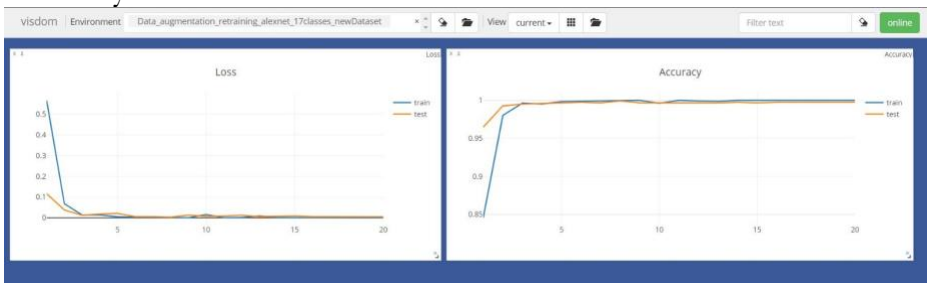
Durante l'allenamento dei modelli dei nodi foglia si è notato che AlexNet, per alcuni tipi di package, non riesce ad ottenere buoni risultati, per questo si è scelto di utilizzare SqueezeNet, che si è mostrata essere decisamente più adatta restituendo elevati valori di accuracy.

Il nodo padre, che si occupa di classificare il package, ha ottenuto i risultati in test riportati sotto:

- Accuracy su dataset standard: 0.997



- Accuracy su dataset aumentato: 0.999



I risultati di accuracy in test, ottenuti sul **dataset standard** per i nodi figli, sono i seguenti:

- Bio: 1.00
- Cereali: 1.00
- Farina semola pizza: 1.00
- Legumi: 1.00
- Pasta 5 cereali: 1.00
- Pasta all'uovo ripiena: 1.00
- Pasta base: 0.94
- Pasta box: 0.89
- Pasta emiliane chef: 1.00
- Pasta integrale: 0.75
- Pasta integrale voiello: 1.00
- Pasta piccolini: 1.00
- Pasta special pack: 1.00
- Pastas specialita: 0.93
- Pasta uovo 5 cereali: 0.50
- Senza glutine: 1.00
- Sughi: 1.00

Le accuracy totali sull'intera **TreeNet**:

- Train: 0.9039735099337748
- Test: 0.9276315789473685

I risultati di accuracy in test, ottenuti sul **dataset aumentato** per i nodi figli, sono i seguenti:

- Bio: 1.00
- Cereali: 1.00
- Farina semola pizza: 0.92
- Legumi: 1.00
- Pasta 5 cereali: 1.00

- Pasta all'uovo ripiena: 0.98
- Pasta base: 0.94
- Pasta box: 0.90
- Pasta emiliane chef: 1.00
- Pasta integrale: 0.88
- Pasta integrale voiello: 1.00
- Pasta piccolini: 0.98
- Pasta special pack: 0.94
- Pastas specialita: 1.00
- Pasta uovo 5 cereali: 0.50
- Senza glutine: 0.99
- Sughi: 0.98

Le accuracy totali sull'intera **TreeNet**:

- Train: 0.9449503311258278
- Test: 0.8980263157894737

È stato ottenuto un leggero overfitting sul dataset aumentato (accettabile se si considera che la rete così facendo diventa meno sensibile alle rotazioni), ma i risultati raggiungi sono decisamente migliori dei precedenti approcci.

```
class TreeNet:
    def __init__(self, dictionary, data_augmentation=""):
        self.package_net = models.alexnet(pretrained=True)
        self.package_net.classifier[6] = nn.Linear(4096, 17)  # Numero esatto di classi nel nostro dataset.
        self.softmax = nn.Softmax(dim=1)
        self.load_dict(self.package_net, data_augmentation + "retraining_alexnet_17classes_newDataset")
        self.dictionary = dictionary

    def load_dict(self, net, name):
        net.load_state_dict(torch.load("./pth/" + name + ".pth", map_location='cpu'))

    def get_class(self, net, img, n_classes, key="", data_augmentation=""):
        net.eval()
        net.cpu()

        if key == "":
            mean_pre_trained = [0.485, 0.456, 0.406]
            std_pre_trained = [0.229, 0.224, 0.225]
        else:
            path = data_augmentation + 'single_dataset/' + key
            txt_file = path + '/train.txt'

        mean_pre_trained = [0.485, 0.456, 0.406]
        std_pre_trained = [0.229, 0.224, 0.225]

        img = img.resize((256, 256))
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean_pre_trained, std_pre_trained)])
        img = transform(img)
        x = Variable(img).cpu()
        x = x.unsqueeze(0)

        if key != "":
            net = models.squeezenet1_0(pretrained=True)
            net.classifier[1] = nn.Conv2d(512, n_classes, kernel_size=(1, 1), stride=(1, 1))
            name_save = data_augmentation + "retraining_squeezenet_" + str(n_classes) + "classes_dataset_" + key
            path = "./" + name_save + ".pth"
            net.load_state_dict(torch.load('./pth/'+path, map_location='cpu'))

        pred = self.softmax(net(x)).data.cpu().numpy().copy()

        pred = pred.argmax(1)

        if key == "":
            filepath = "./classes_first_level.txt"
        else:
            filepath = data_augmentation + 'single_dataset/' + key + '/classes.txt'
        with open(filepath, "r+") as fp:
            for i in range(0, n_classes):
                line = fp.readline()
                n, c = line.strip().split(",")

                if int(c) == int(pred):
                    name = n

        return name, pred
```

Il metodo "get_class" di **TreeNet** si occupa di estrapolare nome e numero della classe predetta dell'immagine data come input. Pertanto, questa funzione verrà chiamata due volte. La prima sul modello nel nodo padre per predire il package, la seconda per caricare dinamicamente il modello allenato del nodo figlio corrispondente a quel package ed ottenere così la predizione finale.

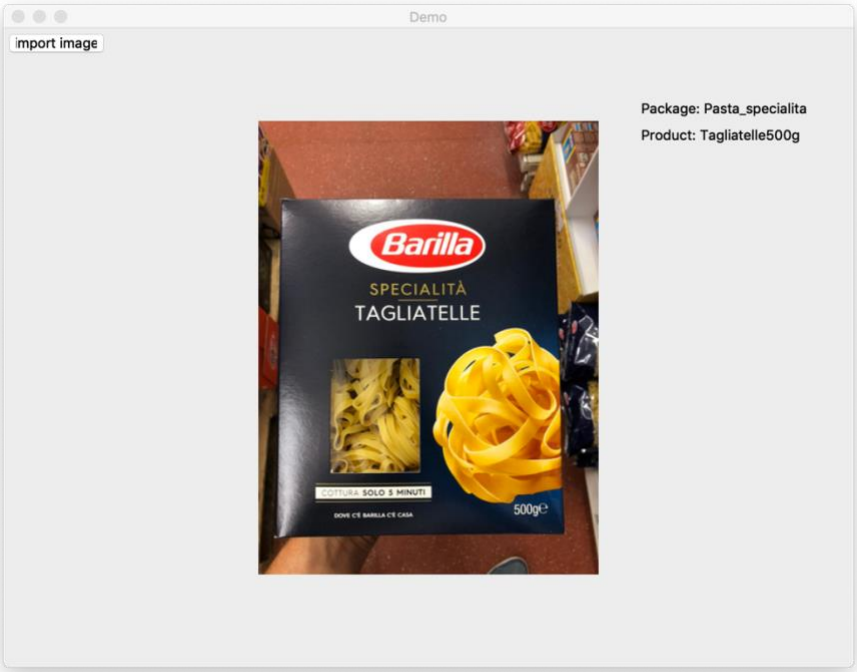
Demo

È possibile testare la TreeNet allenata sul dataset aumentato tramite una demo. Si è scelto di utilizzare il dataset con data augmentation così da avere dei modelli non vincolati alla vista verticale del prodotto.
La demo si presenta come un'applicazione desktop cross platform, sviluppata con PyQt5, in cui, alla pressione del tasto "import image" si apre una finestra di navigazione che permette di selezionare graficamente l'immagine da testare. Una volta confermata, l'immagine scelta sarà mostrata al centro della finestra principale e a destra verranno riportati il package e la classe finale predetta.
Alcune immagini di test sono state inserite nella cartella "Immagini Test" e, per mostrare il comportamento dell'algoritmo di fronte a situazioni reali e non ideali, nella cartella "DatasetSupermercato" sono state inserite foto fatte col cellulare ad alcuni tipi di pasta.

Per avviare la demo basta aprire un terminale nella cartella "Demo" e lanciare il comando:

- python3 Demo.py

Scaricando le dovute dipendenze qualora non lo si fosse già fatto.



Conclusioni

La prima fase del progetto è stata caratterizzata dallo scraping del sito www.barillastudio.it per la creazione di un dataset strutturato in maniera tale da poter facilmente riconoscere la classe di appartenenza dei prodotti.

Successivamente sono stati provati diversi approcci per poter ottenere i risultati migliori possibili, dato il task. I valori di accuracy su Test per ogni approccio sono riportati sotto:

- Allenamento da zero di AlexNet: 0.69
- Fine Tuning sull'ultimo layer: 0.57
- Fine Tuning sugli ultimi due layer: 0.60
- Aggiunta Multi Layer Perceptron ad AlexNet: 0.00
- TreeNet: 0.89

La strategia vincente è stata l'ultima, con una divisione del task in sotto-task e l'allenamento di più modelli per imparare le differenze tra prodotti con lo stesso package anziché imparare le differenze globali.