**Resnet si cum sa creati alterari ale arhitecturii originale**

**Introducere**

In acest document o sa explic codul, dar si oleaca din arhitectura retelei resnet. Nu este important neaparat sa facem poze complicate ale arhitecturii aici, ci sa intelegem dimensiunile tensorilor care sunt in ea. Daca ne dam seama de dimensiunile tensorilor si de ideea generala a oamenilor de la Google care au creat aceasta arhitectura, putem creste aceasta arhitectura (putem adauga noi insine la ea bucatele similare cu ce au descoperit cei de la Google sa mearga, sau puteti sa incercati voi tot felul de idei nastrusnice). Eu am creat o alterare bazata pe ideile celor de la Google. Am marit creierul praactic adaugand la el bucatele similare cu ce Google a desoperit sa mearga. Am preferat aceasta abordare pentru ca sunt sigur ca cei de la Google au incercat tot felul de idei nastrusnice, dar au descoperit ca aceasta sa functioneze cel mai bine. Prin marirea creierului cu bucatele similare cu cele descoperite de Google, este foarte probabil ca noua retea sa aibe capacitate si mai amre de invatare (sunt si mai multe matrici de optimizat, antrenarea poate dura mai mult si poate avea nevoie de mai multe imagini, dar fit-ul final ar trebui sa fie si mai perfect). Acuratetea extrem de ridicata (peste 95% probabil) este extrem de importanta pentru acest sistem. Stim ca este posibil pentru ca China o are facuta (chiar si cand porti masca) si companii precum Google si Apple de asemenea.

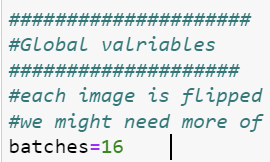
Mai concret, in aceasta alterare, am adaugat un block layer si un mixed layer (exact in tiparul celor declarate de Google, doar ca sunt oleaca mai mari). Codul o sa sufere modificari si deci si aceasta documentatie de asemenea. Nu am cum sa stiu de dinainte cat de bune sau proaste arhitecturile vor fi pana nu incep sa le antrenez pe fetele downloadate de pe Wikipedia si mai bine, pe fetele romanilor.

**Cod**

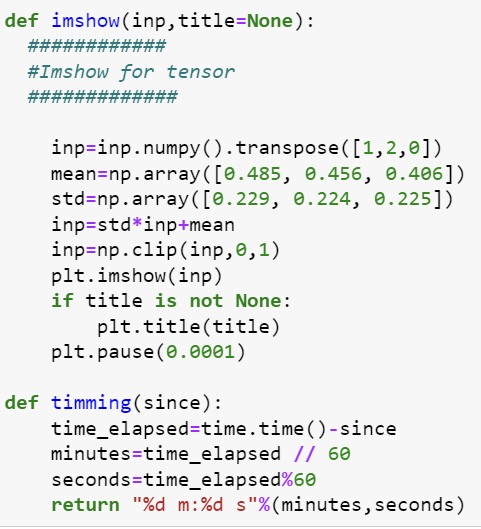
Primul pas este importarea tuturor librariilor de care avem nevoie.

****

Urmatoarea este declararea unei variabile (*batches*)care este folosita in *DataLoader*. Asta practic organizeaza fetele in grupuri si, la antrenare si testare vor fi introduce toate de o data in resnet. Se face acest lucru pentru a creste viteza de antrenare si testare. De obicei grupurile contin batches imagini, numai cand avem mai putine personae decat batches, atunci contin numarul de persoane.



Urmeaza 2 functii: *timming()* care o sa tina cont de cat timp a trecut de cand am inceput sa antenam reteaua si *imshow()* care o sa ia ca input un grup de persoane din *DataLoader* si o sa printeze imaginile din grup intr-un rand impreuna cu numele persoanelor:

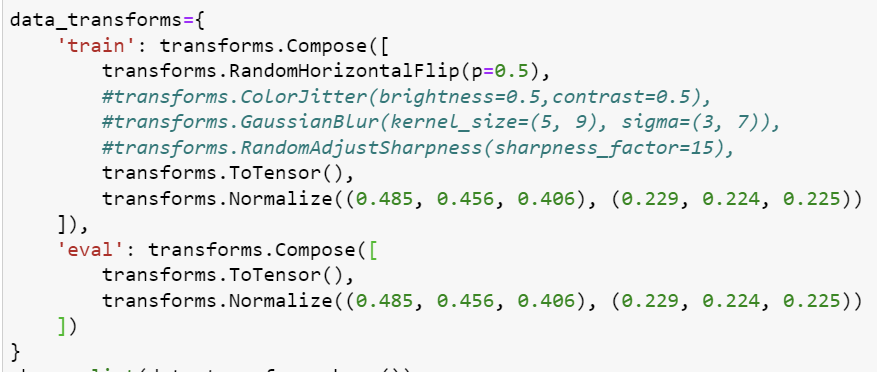


In urmatoarea parte in variabila *data\_transforms* se declara niste transformari care se aplica imaginilor. Aceasta variabila este un dictionar de fapt de transformari ale imaginilor. Dictionarul are 2 chei: “train” si “eval”. Transformarile listate sub “train” vor fi applicate numai fetelor care merg spre antrenare, iar cele sub “eval” numai fetelor care merg la evaluarea (sau testarea) retelei neuronale.

Din moment ce, in general, guvernul romaniei nu are decat o singura poza pentru fiecare cetatean cu buletin, am aplicat mai multe transformari la antrenare. Asta este si deoarece la logarea pe platforma [www.romania.gov.ro](http://www.romania.gov.ro), resnet-ul trebuie sa recunoasca fata in diverse conditii de luminozitate, unghi (dintr-o parte, din fata, etc.). Astfel, am aplicat urmatoarele transformari pentru fiecare fata din antrenare:

* *transforms.RandomHorizontalFlip* care ajusteaza unghiul din care este facuta poza in mod aleator
* *transforms.ColorJitter* care ajusteaza luminozitatea fetei
* *transforms.GaussianBlur* – ajusteaza rezolutia
* *transforms.RandomAdjustSharpness –* ajusteaza claritatea
* *transforms.ToTensor()* – transforma imaginea in tensor
* *transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))* – normalizeaza valorile din tensor (numerele din paranteza sunt deja foarte bine cunoscute online si reprezinta media si deviatia standard, iar normalizarea este similara cu ).

Fetele care merg la evaluarea retelei neuronale trebuie sa nu fie modifcate pentru ca asa o sa functioneze reteaua in realitate. Ce primeste de pe camera aia este fata, fara nici o alterare. Acele alterari au fost facute numai la antrenare (“train”) pentru ca, in general nu avem decat o poza a fiecarui cetatean si trebuie sa acomodam diverse luminozitati, rezolutii, etc.



Variabila *phases* contine cele doua faze de antrenare (sau cele 2 parti in care fetele vor fi impartite): antrenare si evaluare:



Folderul de unde se vor lua fetele este:



Dupa aceea se creeaza un dictionar cu doua chei corespunzand cu cele 2 faze (antrenare si evaluare). Sub o cheie creez *DataLoader*-ul despre care am vorbit la inceput:



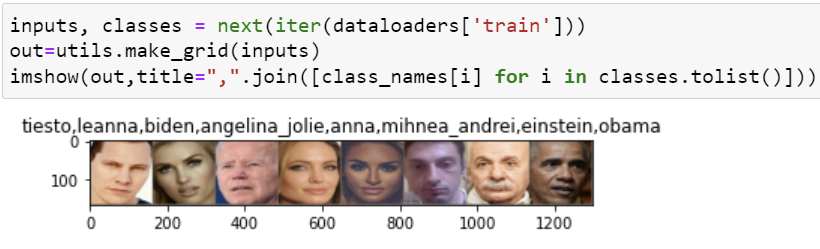
Un dictionar care tine minte cate grupuri de fete am:



Si numele persoanelor care o sa mearga la antrenarea retelei:

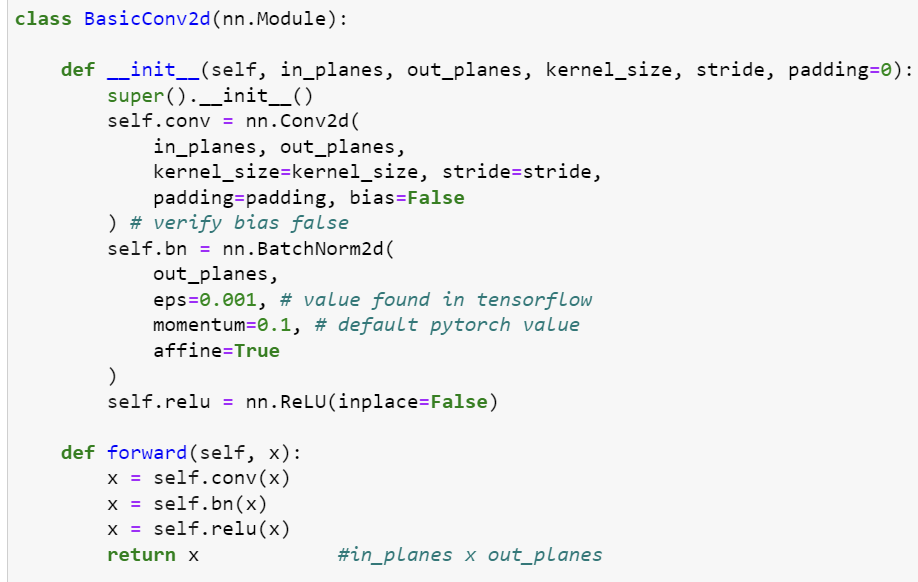


O data creat *DataLoader*-ul,putem vedea exact ce printeaza acea functie *imshow()*:

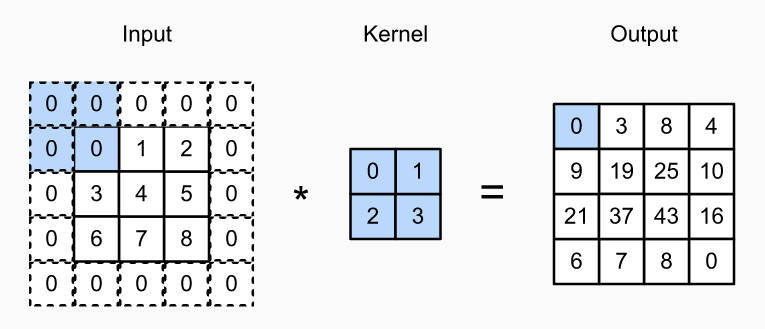


Urmeaza implemetarea arhitecturii neuronale a resnetului, unde o sa explic dimensiunile (am pus comentarii si in cod pentru asta). Este important de invatat aceste dimensiuni pentru a putea crea alternative la retelele neuronale deoarece, ca si in algebra liniara clasica unde nu poti inmulti orice matrice cu oricare alta, nici aici nu poti inmulti orice tensor cu orice alt tensor, deci nici nu poti combina oricum vrei aceste functii (layere din libraria pytorch).

In aceasta arhitectura neuronala, cea mai mica component (neuronii daca vreti) este un Convolutional 2d layer, cu un Batch normalization si un ReLU, in aceasta ordine:



Un convolutional 2d layer este creat in pytorch pur si simplu prin functia *nn.Conv2d().* In general, este important sa intelegem si ce sunt parametrii din functia aceasta. Pentru asta sa luam exemplul foarte bun de la link-ul <https://d2l.ai/chapter_convolutional-neural-networks/padding-and-strides.html> :



Acum o sa incerc sa explic parametrii din functia *nn.Conv2d()* folosind acest exemplu:

* *padding*=1 in aceasta imagine pentru ca, puteti observa ca matricea originala de 4x4 este inconjurata de un zid de 0 cu latimea 1.
* *Kernel\_size*=2 in aceasta imagine pentru ca matricea kernel este de 2x2.
* *In\_planes*=4 in aceasta imagine pentru ca matricea originala este de 3x3 (am adaugat noi un zid de latime 1 in jurul ei care a facut-o de 5x5)
* *Out\_planes*=4 in aceasta imagine, care este fortata de celelalte inputuri de fapt. Pentru asta, trebuie sa intelegem cum este calculat output-ul:
  + Se incepe cu submatricea de 2x2 din coltul stanga sus si se inmulteste element cu element cu matricea kernel pentru a se obtine primul element din stanga sus al matricei output:
  + Cand *stride=1*, mutam submatricea de 2x2 din matricea input cu o coloana mai la dreapta (“slide it with stride 1” in engleza) si repetam calculul pentru a obtine elemental din pozitia (1,2) din output: . Daca *stride=2* am muta submatricea 2x2 cu 2 coloane la dreapta, etc.
  + Continuam cu *stride=1* si repetam procesul de la punctul anterior. Mutam submatrices 2x2 cu o coloana mai la dreapta si inmultim element cu element cu matricea kernel, Astfel, obtinem ca rezultatul in pozitia (1,3) din output este: .
  + Se tot repeat acest process pana se completeaza tot outputul care are de altfel dimensiuni fixe o data ce stim ceilalti parametrii, dupa cum observati. Se poate scrie o formula pentru asta, dar nu este asa de importanta pentru aplicatii. De asemenea, nu are rost sa pierdem timpul si sa ne dam smecheri cu de ce se numeste convolutie si istorie (vine de la discretizarea integralei de convolutie si aplicarea discretizarii pe tensori).

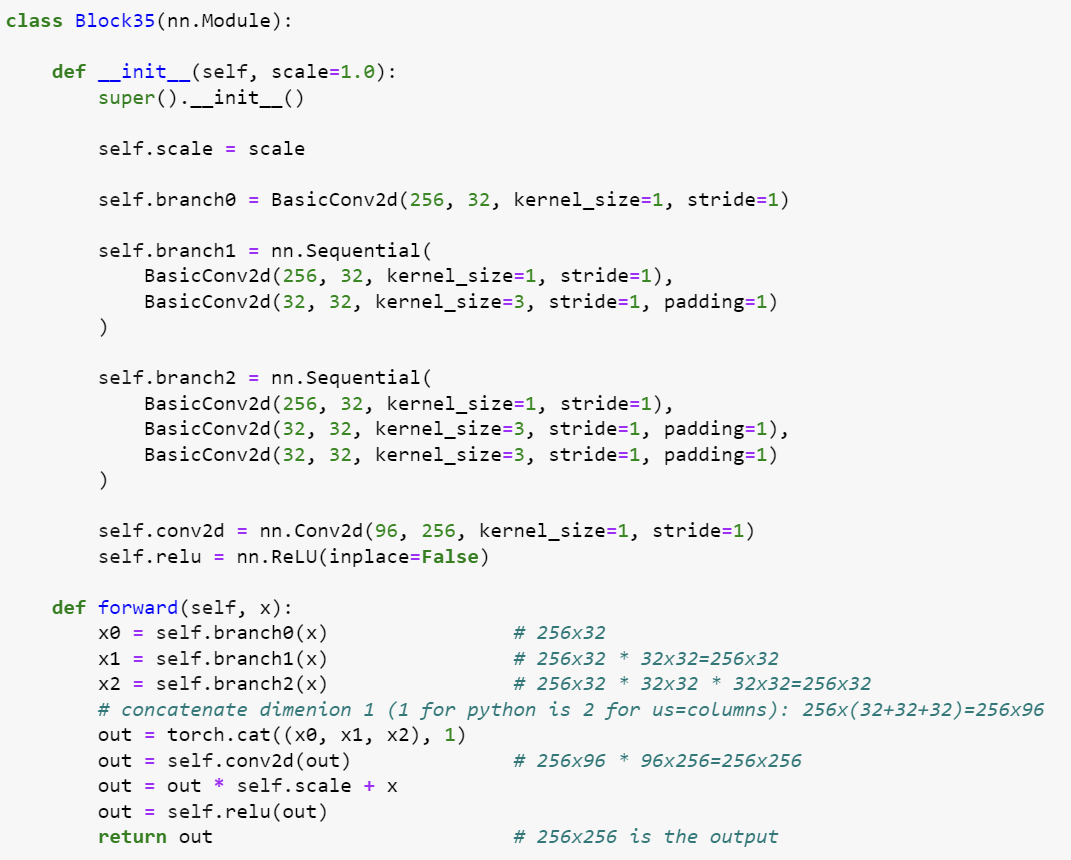
Acum, revenim la reteaua neuronala resnet, intelegerea ei si prezentarea alternativei cu care am venit. Google a combinat acesti neuroni in parti de creierasi de 2 tipuri:

* *Block*- care are un output patratic (256x256 de exemplu)
* *Mixed*- care are un output dreptunghiular (256x896 de exemplu)

Ei au creat mai multe parti de creierasi de tip *block* si *mixed* de diverse dimensiuni, pe care la final de tot le-au combinat intr-un creier.

*Block*

Acum haideti sa ne uitam exact cum arata un creierasi de tip *block*. Felul in care toate aceste retele neuronale sunt construite in pytorch este prin definirea unei clase. In prima parte dupa *super().\_\_init\_\_()* se defines functiile care o sa fie folosite, iar in a doua parte numita *forward()* se defineste cum trece informatia prin aceste functii. In *Block35* (asa cum a fost denumit de ei) am lasat si niste comentarii cand vine vorba de dimensiunile output-ului din fiecare functie:



Ideile sunt destul de simple de fapt:

* *self.branch0* este de 256x32 pentru ca primul argument este 256, iar al doilea este 32 (stim si ca matricea de kernel este de fapt un numar in acest caz si ca ne miscam cu cate o coloana la dreapta din moment ce *stride=1*). Acum ne uitam in *forward()* si vedem o informatia *x* trece prin acest neuron mai intai, deci output-ul *self.bracnh0(x)* este de 256x32. Asadar, *x0* este de 256x32.
* *self.brach1* pare mai urat, dar este de fapt la fel de simplu. Functia nn.Sequential practic spune ca informatia o sa treaca mai intai prin *BasicConv2d(256,32,….)* si dupa aceea prin *BasicConv2d(32,32,….)*. Este ca si cum as scrie *x=self.basic1(x)* urmat de *x=self.basic2(x)*, dar este mai elegant folosind *nn.Sequential*. Astfel, o data ce informatia iese din acesti 2 neuroni convolutionali o sa aibe dimensiuni: . Asta este exact ca la matricile de 2 dimensiuni. Deci *x1* o sa aiba dimensiuni 256x32.
* *self.branch2* este la fel ca *self.branch1*, numai ca sunt 3 neuroni acolo. Pe aceeasi idee, dimensiunea lui *x2* este .
* Continuam sa ne concentram atentia pe *forward()* pentru ca ea dicteaza in ce ordine si cum trece informatia prin creieras. Urmatorul pas este un *torch.cat* al celor 3 puncte de mai sus pe coloane (in python 0 este 1 al nostrum si 1 este 2 al nostrum, deci 0 ar fi fost pe randuri, iar 1 este pe coloane). Dupa cum sugereaza numele *torch.cat* face o concatenare a rezultatelor de la cele 3 puncte anterioare pe coloane, astfel ca output-ul dupa acest pas are dimensiunea: .
* Urmatorul pas in *forward()* este introducerea rezultatului de la punctul anterior in al neuron convolutional 2d de dimensiuni 96x256. Acest ultim pas este exact pentru a face acest creieras *block* sa scoata ceva patratic deoarece: .
* Ultimul pas este un ReLU, ceva comun tuturor retelelor neuronale.

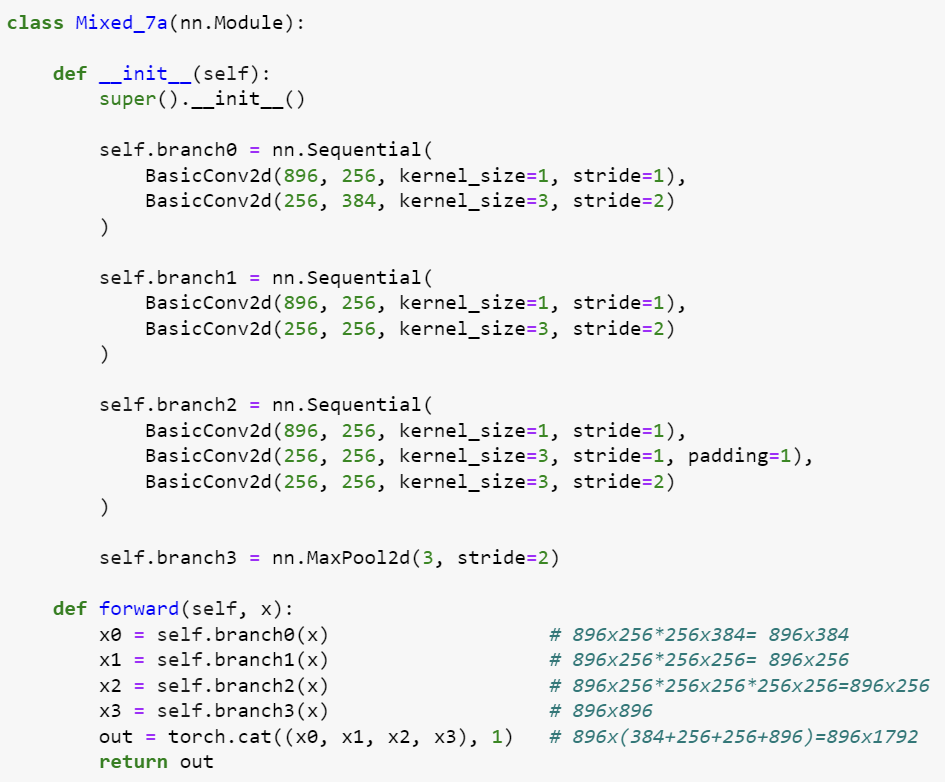
Ideile explicate mai sus sunt comune absolut tuturor creieraselor convolutionale, iar *Block35* de mai sus arata foarte similar cu toate celelate creierase numite *Block* in resnet. Resnet-ul original are *Block17, Block8.* Asa ca de ce sa nu creez si eu un *Block* al meu, pe care o sa il alipesc la celelalte. Metoda de alipire la celelalte *Block* si *Mixed* o sa fie prezentata la final, dar haideti sa vedem cum creez eu unul asemanator:



Probabil ca observati din prima ca este foarte asemanator:

* *self.branch0* are tot un neuron convolutional 2d, dar cu alte dimensiuni. Am fost fortat de fapt sa folosesc 3840 si o sa vedem de ce atunci cand o sa vedem cum sunt alipite toti creierasii *Block* si *Mixed* la final. Acel al doilea parametru de 256 a fost ales la intamplare de mine de fapt. As fi putut sa pun orice alt numar de fapt. Dar o data ce ageleti alt numar aici, trebuie sa aveti grija oleaca la pasii urmatori. Principiile de baza sunt aceleasi ca cele prezentate mai devreme in schimb.
* *self.branch1* are tot 2 neuroni convolutionali 2d, iar primul are prima dimensiune de 3840 deoarece trebuie sa fie aceeasi cu prima dimensiune din *self.branch0*. Asta este pentru ca acestea vor fi concatenate dupa coloane, deci trebuie sa aiba acelasi numar de randuri.
* *self.branch2* are tot 3 neuroni convolutionali 2d, si primul are tot 3840 la primul input din acelasi motiv ca la punctul anterior. Parametrul al doilea de 256 a fost ales la intamplare si puteti allege orice de fapt.
* In schimb, o data ce faceti o alegere pentru al doilea parametru, o sa va forteze ca ultimul neuron convolutional 2d sa fie de o anumita dimensiune pentru a avea un output patratic. In cazul meu, dupa concatenarea pe coloane a lui *x0,x1,x2* care toate au dimensiunile 3840x256, am obtinut . Asadar, pentru ca rezultatul sa fie patratic, ultimul neuron convolutional 2d trebuie sa aiba dimensiunea .

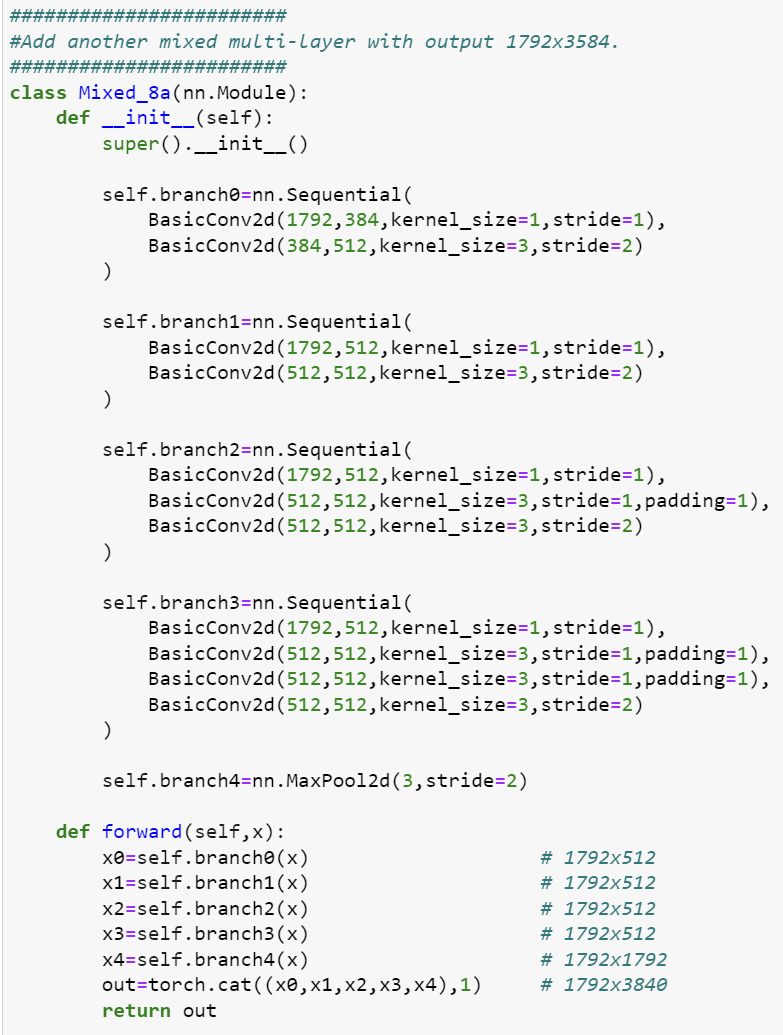
*Mixed*



Nu o sa mai explic ideile pas cu pas, pentru ca sunt foarte similare:

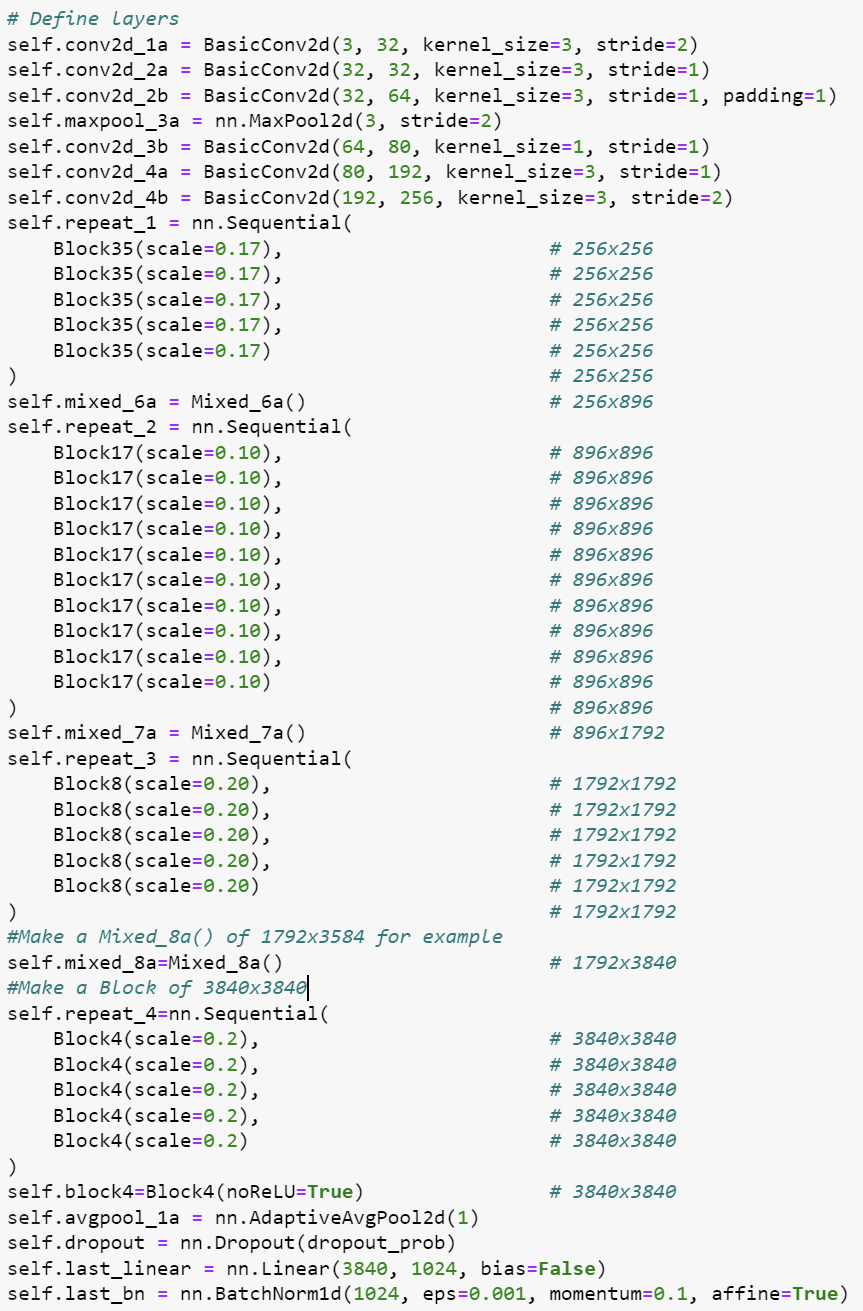
* *x0* are dimensiuni
* *x1* are dimensiuni
* *x2* are dimensiuni
* *x3* este un MaxPool care este o matrice patratica cu dimensiunile egale cu numarul de randuri al inputului x ().
* *out* este concatenarea celor 4 puncte de mai sus, cu dimensiunea din comentarii.

Reteaua originala *resnet* are *Mixed\_7a* si *Mixed\_6a*. Pe aceste idei am creat si eu urmatorul creieras:



Ca si inainte, prima dimensiune de 1792 a fost fortata de felul in care sunt alipite aceste creierase, dar celelalte pot fi alese sa fie orice.

Toate aceste creierase sunt alipite in creierul final. Ideea este ca informatia este trimisa prin layere convolutionale 2d simple successive pana cand se ajunge la dimensiunea primului *block*. Dupa aceea, mai multe creierase *block* de acelasi fel sunt alipite intr-un *block* mai mare. Acest *block* mai mare este urmat de unul *mixed*, urmat de un *block* mai mare si tot asa. Ultimul layer, este unul linear care reduce dimensiunea de la output-ul ultimului block la 1024 ini acest caz iar ultimul ia de la dimensiunea 1024 la numarul de nume din dataset. Ideea este ca outputul final o sa fie un vector de probabilitati pentru fiecare nume. Cand creierul primeste o imagine, outputul final o sa fie un vector cu probabilitatea ca acea imagine sa fie fata fiecarei persoane pe care creierul a vazut-o vreodata.



Regulile pentru dimensiunile din creier sunt aceleasi ca cele pe care le-am vazut pana acum la creierasele *block* si *mixed*:

