Recunoașterea unei cifre scrise de mână folosind o rețea neuronală convoluțională și biblioteca TensorFlow

Paul TEODORESCU

Institutul Național de Cercetare-Dezvoltare în Informatică – ICI București

paul.teodorescu@ici.ro

Rezumat: În această lucrare se propune rezolvarea unei probleme vizuale de recunoaștere, cu ajutorul unui instrument software, a unei cifre scrise de mână. A fost utilizată o tehnică de învățare automată în care se produce un rezultat bazat pe o experiență anterioară. Se arată cum, având la început valori de intrare și valori de ieșire numite etichete sau labels, computerul începe să învețe să recunoască corect valoarea de ieșire (în acest caz o cifră), prin modelul construit în tehnica numită învățare supervizată. Așadar obiectivul este de a ghici valoarea de ieșire la o valoare nouă a intrării, odată ce a fost cunoscut modelul. Cheia alegerii unui algoritm corect în rezolvarea unei probleme prin tehnologia de învățare supervizată este identificarea corectă a metodologiei, adică răspunsul la întrebarea: „este o problemă de regresie sau de clasificare?”. În cazul prezentat, se dorește ghicirea categoriei sau a clasei (ce are un număr fix de posibile valori, numite și valori discrete) din care fac parte acele date de intrare (practic cifrele scrise de mână). O rețea neuronală convoluțională cu 4 straturi, împreună cu instrumentul numit *TensorFlow* ce aduce o întreagă bibliotecă de inteligență artificială a fost folosită de computer la rezolvarea problemei de clasificare a unor cifre scrise manual (a fost stabilit că se va lucra cu 10 clase, care reprezintă cifrele de la 0 la 9). Întrucât înțelegerea tehnologiei *TensorFlow* cere un extra efort pentru că are o logică puțin mai ciudată, lucrarea oferă explicații printr-un exemplu care are ca punct de plecare o bază de date numită MNIST (*Modified National Institute of Standards and Technology*) ce cuprinde o sumedenie de imagini reprezentând cifrele de la 0 la 9 scrise de mână, în caligrafie variată. Prin hrănirea rețelei neuronale cu aceste zeci de mii de imagini, modelul construit de *TensorFlow* reușește să ghicească în bună măsură numărul reprezentat în acea imagine.

Cuvinte cheie: bibliotecă, vector, tensor, variabile, matrice, optimizator, propagare (înainte și înapoi).

Recognizing handwritten digits using a convolutional neural network and TensorFlow library

Abstract: In this paper it is proposed to solve a visual problem of recognizing a handwritten figure. A machine learning technique will be used in which a result is produced based on previous experience. It will be seen how, starting with input values and output values (called *labels*), the computer begins to correctly recognize the output value (in this case a figure), through the model built in the technique called *supervised learning*. So the goal is to guess/predict the output value for a new input value (in other words to map each input image to the correct numeric digit), once the model is known. The key in choosing a correct algorithm for solving a problem through supervised learning technology is the correct identification of the methodology to be used, i.e. the answer to the question: „is it a regression or is it a classification problem?”. In the case presented here, it is desired to guess the category or class (which has a fixed number of possible values, also called discrete values) of which those input data (practically handwritten figures) are part of. In order for the computer to resolve the classification of manually written numbers (it has been established that it will work with 10 classes, which represent the numbers from 0 to 9), a 4-layer convolutional neural network will be used together with the instrument called *TensorFlow* which brings with it a whole artificial intelligence library. Since the understanding of TensorFlow technology requires an extra effort (because its strange logic), this article will attempt to explain it using an already classic example. At the base of this example is a database called MNIST (Modified National Institute of Standards and Technology) that contains a lot of pictures representing the numbers from 0 to 9 manually written in a very wide palette. By feeding the neural network with these tens of thousands of images, the model built by TensorFlow will learn to guess the number represented in that image.

Keywords: library, vector, tensor, variables, matrix, optimizer, backpropagation, forward propagation.

1. Introducere

Acest articol propune un excelent exercițiu-prototip pentru a învăța despre rețelele neuronale în general: acela în care computerul învață să recunoască scrisul de mână. Cum mașina învață să recunoască o cifră scrisă de mână dintr-o imagine, se spune că este un exercițiu de învățare automată (*Machine Learning* – ML). Dintre cele 2 tipuri de probleme care stau sub umbrela

tehnologiei de ML, (regresie și clasificare), acest exercițiu intră sub incidența tipului de clasificare: fiecare cifră de la 0 la 9 reprezintă o clasă, iar algoritmul construit va identifica clasa din care aparține cifra din imaginea de intrare. Ca în majoritatea problemelor de inteligență artificială, se va folosi o rețea neuronală artificială inspirată de procesele biologice și de modelul de conectivitate între neuronii din creierul uman. Pe lângă această rețea artificială, vom utiliza baza de date MNIST care este o bază mare de date ce cuprinde 60.000 imagini de training și 10.000 de test, toate fiind imagini cu cifre scrise de mână. Scopul acestui exercițiu este construcția unei reprezentări matematice numite *model* care să poată fi folosit în a face predicții. Modelul este hrănit cu date din MNIST și, după ce este instruit cu date de training, va putea ghici categoria din care face parte imaginea de intrare.

Așa cum s-a menționat, exercițiul este considerat ca fiind clasic întrucât a fost făcut de mii de cercetători și reprezintă un caz clasic de demonstrație al mecanismului de ML, al modului în care lucrează o rețea neuronală artificială și al forței instrumentului *TensorFlow*. Întrucât în construcția modelului sunt implicate zeci de variabile și hiperparametri, fiecare exercițiu prezentat în revistele de specialitate arată diferit și are rezultate diferite prin sutele de modificări și combinații de variabile. Însă, foarte rar, se explică procesele și fenomenele de profunzime ce au loc la nivelul rețelei și rar se explică rațiunile pentru care este nevoie de toate aceste ecuații și variabile. Astfel a apărut oportunitatea scrierii acestui articol, un material-ghid care să îi ajute pe tinerii informaticieni în clarificarea și înțelegerea întregului peisaj în care are loc acest minunat exemplu de ML. Nu se dorește aici obținerea unui rezultat de excepție (rezultatul fiind *acuratețea* cu care modelul face predicția) în competiție cu alte rezultate din spațiul deschis al inteligenței artificiale. A rezultat un material care prezintă toate elementele ce trebuie luate în considerare atunci când practic se dorește scrierea liniilor de cod: funcțiile de activare, optimizarea Gradientului, *batching*, rata de învățare, inițializarea ponderilor, acuratețea modelului, oprirea prematură etc. Acestea sunt aspectele deosebite prezente în universul inteligenței artificiale care trebuie înțelese de oricine face cercetare în domeniu și de aici se conturează importanța articolului. Se pleacă de la ipoteza că pentru a putea parcurge articolul cititorul posedă cunoștințe de Python, are un nivel ridicat în domeniile algebrei liniare și statisticii și este familiar cu rețelele neuronale artificiale. Pentru ușurință, setul de date oferit de MNIST este ideal pentru scopul exercițiului, lăsând la latitudinea cititorului alegerea tipului de rețea neuronală, a platformei și a setului de biblioteci cu care poate lucra. În cazul de față a fost aleasă o rețea neuronală convoluțională împreună cu *framework*-ul oferit de *TensorFlow*.

2. Datele de lucru și rețeaua neuronală aleasă

Așa cum a fost precizat, se va lucra cu 60.000 imagini pentru instruirea rețelei și 10.000 de date de test, diferite de datele de instruire. Folosind datele de instruire, modelul construit învată să recunoască cifra din imagine.

Datele de intrare reprezintă imagini, iar ca tip de rețea a fost aleasă o rețea neuronală convoluțională (CNN, Figura 1), deoarece este tipul de rețea folosită cu precădere în analiza de imagini vizuale, în recunoașterea de imagini (*Image Recognition*) sau în clasificarea obiectelor (*Object Classification*), adică exact ceea ce este nevoie pentru tipul problemei vizuale propuse.

Fiecare imagine (având 28 x 28 =784 de pixeli) poate fi considerată ca o matrice având valori între 0 și 1 și va fi transformată într-un vector de lungime 784. Așadar fiecare imagine reprezintă un set de 784 de intrări. Ieșirile vor fi tot vectori: de exemplu cifra 0 va fi reprezentată de vectorul [1,0,0,0,0,0,0,0,0,0], iar cifra 5 de vectorul [0,0,0,0,0,1,0,0,0,0]. Pentru ultimul strat al rețelei propuse (cu 2 straturi ascunse), funcția de activare folosită este Softmax.

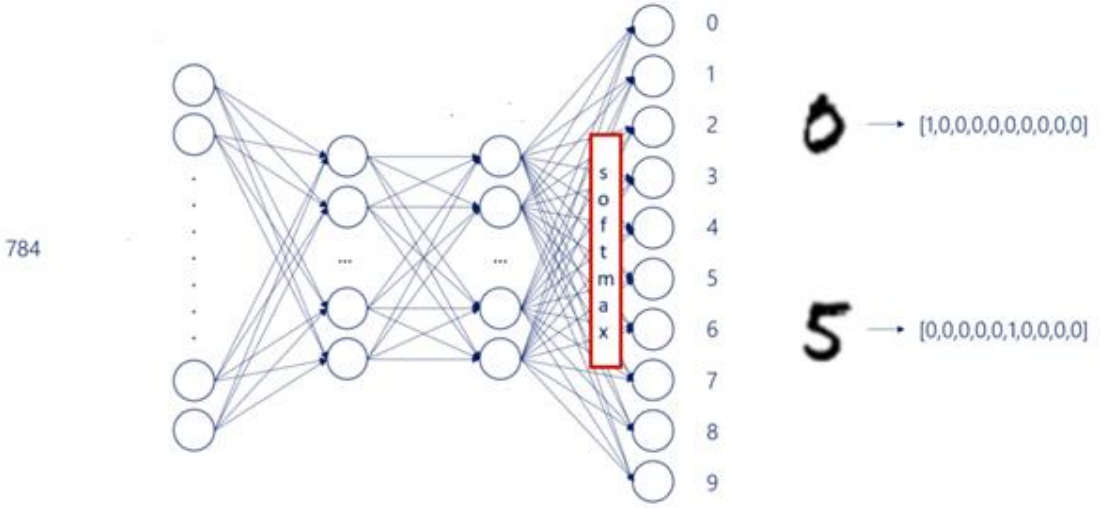


Figura 1. Rețeaua aleasă având 2 straturi ascunse

Etapele parcurse sunt:

- schițarea modelului și alegerea funcției de activare;
- descrierea punctelor de intrare a datelor, variabilelor; precizarea operațiilor specifice folosind sintaxa TensorFlow;
- alegerea optimizărilor corespunzătoare (*optimizers*);
- împărțirea setului de date în sub-seturi (*batch)* pentru o instruire mai rapidă;
- inițializarea variabilelor în vederea începerii operației de învățare (*learning*);
- procesul de învățare.

3. Algoritmul

Pentru acuratețea instruirii rețelei, algoritmul folosit este "propagarea înapoi" (*backpropagation*) în efortul de a învăta rețeaua în modul supervizat (*supervized learning*): se calculează gradientul funcției cost (*loss function*) iterând înapoi în rețea, strat după strat, începând cu ultimul. (Rezultatul acestei funcții este penalizarea/costul pentru o clasificare incorectă; în efortul de a optimiza procesul de predicție al rețelei, se urmărește micșorarea acestei penalizări. Cu alte cuvinte costul este valoarea care indică cât de greșită este predicția modelului, iar zero reprezintă predicția perfectă.)

Întregul set de date de instruire va trece de 15 ori (valoare aleasă la întâmplare de utilizator) prin rețea, adică vor exista 15 cicluri numite epoci (epoch). La fiecare epocă se validează datele și se pune (câteodată) stop la etapa de instruire. Este ceea ce se numește oprire prematură (*early stopping*): se oprește mai devreme instruirea rețelei înainte de apariția fenomenului de 'modelare peste așteptări' sau *overfitting*: modelul construit a învățat datele de instruire mult prea bine pentru a putea să producă rezultate noi cu date suplimentare noi (cu alte cuvinte nu mai poate să prezică observații viitoare). Datele de validare sunt cele care ajută la detectarea și prevenirea fenomenului de *overfitting*. Se actualizează ponderile **w** și bias-urile doar pentru setul de date de antrenare. Datele au fost împărțite în date de instruire, validare și testare. Când se observă că modelul este cumva gata instruit, se va aplica setul de date de validare însă execuția se face doar în direcția înainte (*forward propagation*). În felul acesta se calculează costul de validare (*validation loss*), care trebuie să fie aproximativ egale cu cele din operația anterioară de instruire. S-au folosit funcțiile *training_loss* și *validation_loss* din biblioteca *TensoFlow*. Datele sunt pregătite/antrenate cu ajutorul metodei Gradient Descent: tentativa de găsire a minimului funcției *loss* prin deplasarea pe porțiunea descendentă a curbei funcției. Aplicând această metodă, se obțin descreșterile funcției *loss* din etapa de instruire a rețelei. Însă dacă se va observa că ele cresc (în loc să descrească),

atunci, în acel punct, se pune o atenționare (*red-flag*) pentru că în acel punct se oprește procesul de instruire. Acel punct este avertizarea unui *overfit*. În final se testează acuratețea modelului folosind datele de test.

Observație: Întregul set de date de instruire (exemple) a fost împărțit în sub-seturi de date (*batches*), fiecare *batch* având 100 de imagini. A nu se confunda termenul de iterație cu cel de epoci: o iterație este o singură procesare a unui *batch*. Numărul de iterații este de fapt numărul de exemple din întregul set de date împărțit la numărul de exemple dintr-un *batch*. Există 2 bucle de procesare a datelor: o buclă mare pentru procesarea pe toate cele 15 epoci, buclă care înglobează o buclă mică pentru procesarea tuturor sub-seturilor de date.

4. Scrierea programului în limbajul Python

Mai întâi se importă librăriile *Numpy* și *TensorFlow*. Apoi se încarcă automat datele MNIST folosind linia de cod:

from tensorflow.examples.tutorials.mnist ***import*** *input_data*

Prin această linie de cod se încarcă setul de date în folderul ales care este și directorul lui Jupiter notebook. Dataset-ul a fost fragmentat în *training*, *validation* și *test*. Mai mult, a fost și *preprocesat* într-un format simplu și util. Se va folosi o functie a lui *TensorFlow* specială pentru datele MNIST. Fiecare imagine este convertită într-un vector de lungime 28x28=784 unde fiecare valoare corespunde cu intensitatea culorii. Scala culorilor - standartizată - este între 0 și 1 (ne referim la scala nuanțelor de gri), unde 0 înseamnă alb și 1 înseamnă negru. Fiecare imagine va fi transformată rând cu rând într-un vector. A fost precizat că se tratează o problemă de *clasificare*, iar țintele (*targets*) sunt date discrete (*categorical data*). Pentru ca datele/variabilele categorice să poată fi folosite de algoritmul de învățare a mașinii, ele sunt convertite de o funcție de codificare numită *one-hot encoding*. Țintele pentru fiecare exemplu/imagine reprezintă tot un vector de lungime 10 având 9 zerouri și un singur "1" (de exemplu [0,0,0,1,0,0,0,0,0,0]). La prima rulare a acestei comenzi, se așteaptă un timp mai îndelungat întrucât se încărcă întregul set de date.

mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

Așadar se folosesc capabilitățile *built-in* ale lui *TensorFlow*.

5. Etapa de schițare a modelului

Primul strat al rețelei conține 784 de neuroni, pentru că sunt 784 de intrări (ce reprezintă o imagine), iar ultimul strat are 10 neuroni pentru cele 10 ieșiri ce reprezintă un număr. Între aceste straturi, se poate alege așezarea a 2 straturi ascunse fiecare cu câte 50 de neuroni. Alegerea lățimii (*width*) și adâncimii (*depth*) rețelei (numiți hiperparametrii) este pur aleatorie. Cercetătorul jonglează cu ei pentru a obține cel mai bun rezultat de predicție. La fel, se va alege ca dimensiunea straturilor ascunse (*size* = 50) este aceeași pentru cele 2 straturi.

input_size = 784

output_size = 10

hidden_layer_size = 50

Pentru noi valori ale variabilelor se fac rulări noi și se resetează graful computațional cu ștergerea din memorie a tuturor variabilelor (rămase în urma ultimei rulări):

tf.reset_default_graph()

Această resetare trebuie făcută, deoarece algoritmul construit pentru prima oară este sau nu cel mai bun. Resetarea variabilelor are loc pentru antrenarea sau instruirea rețelei de la zero.

Primele obiecte ce trebuie create pentru *TensorFlow* sunt numite *placeholders* și pentru intrări, dar și pentru ținte (*targets*), și reprezintă locurile de hrănire cu date:

inputs = tf.placeholder(tf.float32, [None, input_size])

targets = tf.placeholder(tf.float32, [None, output_size])

În continuare se definesc ponderile și *bias*-urile care intră în ecuația liniară (f = wx + b) aplicată între stratul de intrare și primul strat ascuns. *TensorFlow* ajută la inițializarea lor printr-un inițializator propriu folosind formulele de calcul Xavier și prin funcția proprie *tf.get_variable*. În felul acesta se setează valorile inițiale ale ponderilor și ale bias-urilor.

Observație: se poate folosi și o altă metodă de setare a lor, de exemplu cea a bibliotecii NumPy numită *random uniform* care alege ponderile și bias-urile la întâmplare, dar într-o manieră uniformă în care fiecare valoare are aceeași probabilitate de a fi aleasă. Însă cercetătorul Xavier Glorot a venit cu 2 formule de inițializare mai bune decât orice alte formule:

- **Uniform Xavier initialization**, unde se alege fiecare pondere dintr-o distribuție uniformă:

$$\text{in [-x,x] for } x = \sqrt{\frac{6}{inputs + outputs}}$$

- **Normal Xavier initialization**, unde se alege fiecare pondere dintr-o distribuție normală cu un *mean* = 0 și deviația standard de:

$$\sigma = \sqrt{\frac{2}{inputs + outputs}}$$

Nu este atât de importantă metoda de inițializare aleasă cât numărul de intrări în stratul următor. Dacă nu se specifică o anume inițializare, implicit se face inițializarea folosind metoda Xavier.

Așa cum se poate vedea în Figura 2, există 3 perechi de (ponderi, bias): (w_1, b_1), (w_2, b_2) și (w_3, b_3) folosite la ecuațiile dintre straturi.

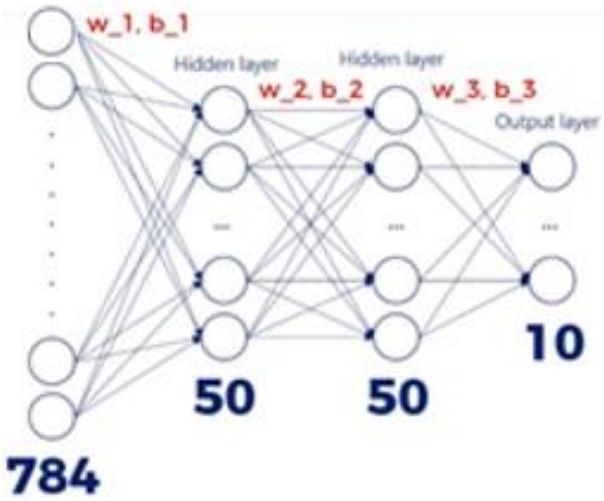


Figura 2. Modelul ales cu identificarea ponderilor și a bias-urilor

A fost așadar stabilită folosirea funcției ***tf.get_variable*** care, implicit, vine cu metoda de inițializare Xavier (Glorot) pentru ponderi și bias-uri și care are nevoie de 2 parametri: dimensiunea stratului de intrare și dimensiunea stratului ascuns (*shape*), respectiv 784 și 50 (Figura 3).

weights_1 = tf.get_variable("weights_1", [input_size, hidden_layer_size])

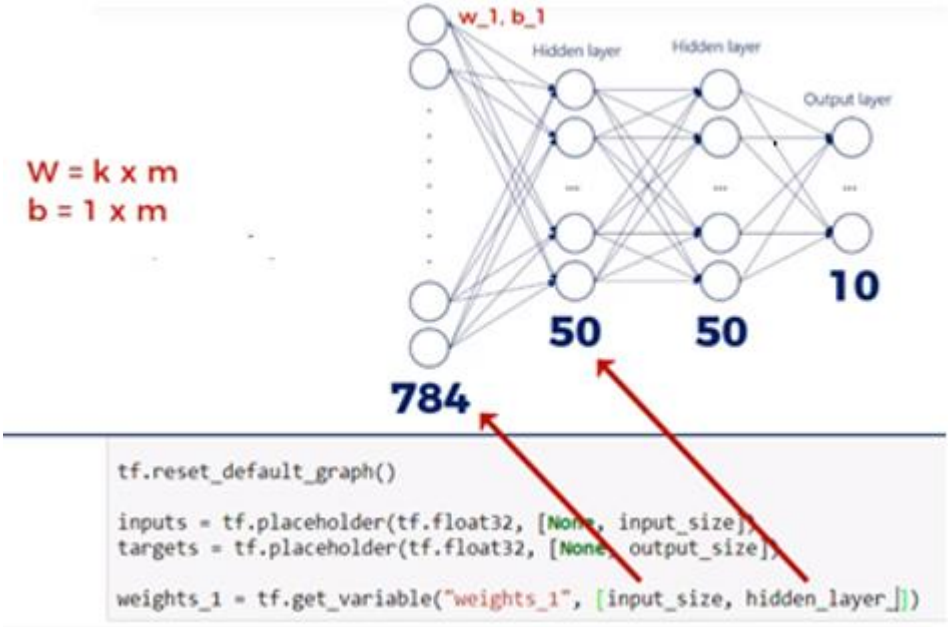


Figura 3. Explicație linie de cod pentru ponderi

La fel se procedează și pentru bias-uri.

biases_1 = tf.get_variable("biases_1", [hidden_layer_size])

Dacă ieșirile din neuronii primului strat ascuns se vor numi *output_1*, și dacă se va folosi funcția ReLu (*rectified linear unit*) ca funcție de activare (Figura 4), atunci:

outputs_1 = tf.nn.relu(tf.matmul(inputs, weights_1) + biases_1)

unde *tf.nn* este un modul a lui *TensorFlow* ce conține cele mai uzitate funcții de activare. Se poate observa că practic se face o înmulțire de matrici (*matmul*) între matricea intrărilor x și matricea ponderilor w, la care se adaugă bias-ul.

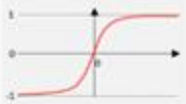
Nume	Formula	Derivata	Graficul	Interval
sigmoid	$\sigma(a) = \frac{1}{1+e^{-a}}$	$\frac{\partial\sigma(a)}{\partial a} = \sigma(a)(1-\sigma(a))$		(0,1)
TanH	$\tanh(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}$	$\frac{\partial \tanh(a)}{\partial a} = \frac{4}{(e^{a}+e^{-a})^2}$		(-1,1)
ReLu	relu(a) = max(0,a)	$\frac{\partial \text{relu}(a)}{\partial a} = \begin{cases} 0, if\ a \le 0 \\ 1, if\ a > 0 \end{cases}$		(0,∞)
softmax	$\sigma_i(a) = \frac{e^{a_i}}{\sum_j e^{a_j}}$	$\frac{\partial \sigma_i(a)}{\partial a_j} = \sigma_i(a)\left(\delta_{ij} - \sigma_j(a)\right)$	diferit de fiecare data	(0,1)

Figura 4. Funcții de activare

Observație: dacă funcția ReLu se folosește la activarea neuronilor primelor 3 straturi, în schimb la ultimul strat (cel de ieșire din rețea) se folosește funcția Sigmoid, fiind cea mai bună în probleme de clasificare. Indiferent de ce s-a întâmplat înainte în rețea, ieșirea finală a algoritmului este o valoare între 0 și 1, reprezentând probabilitatea unui potențial rezultat. Așa cum se vede în Figura 5, aplicarea funcției Softmax transformă elementele vectorului **a** în probabilități, cu suma lor 1.

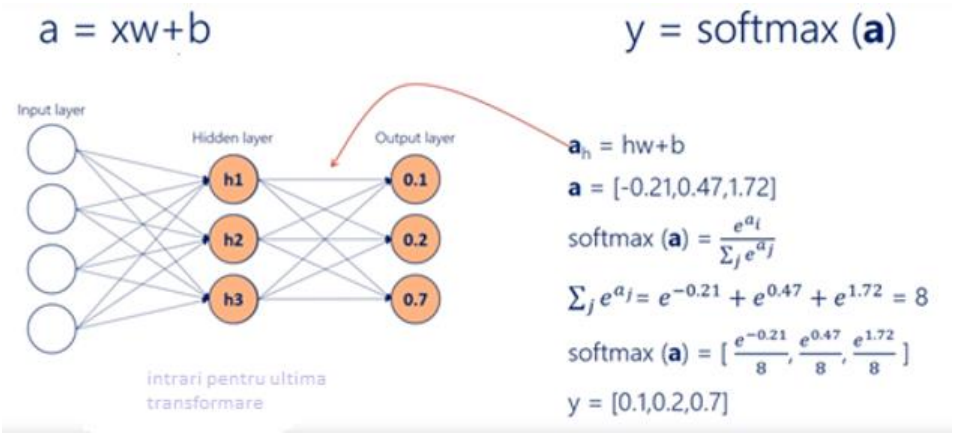


Figura 5. Funcția Softmax

Se procedează similar cu stratul 2 și cu stratul 3, cu referire la ponderile dintre primul și al doilea strat ascuns, respectiv dintre al doilea strat ascuns și ultimul strat. Între primul și al doilea strat avem:

weights_2 = tf.get_variable("weights_2", [hidden_layer_size, hidden_layer_size])

biases_2 = tf.get_variable("biases_2", [hidden_layer_size])

Operația de înmulțire a matricilor *output_1* (rezultată la ieșirea din stratul 1) cu matricea *weights_2* (ponderile stratului 2) și aplicarea funcției ReLu:

outputs_2 = tf.nn.relu(tf.matmul(outputs_1, weights_2) + biases_2)

Între al doilea strat ascuns și stratul de ieșire avem:

weights_3 = tf.get_variable("weights_3", [hidden_layer_size, output_size])

biases_3 = tf.get_variable("biases_3", [output_size])

Ieșirile se calculează tot cu înmulțirea de matrici, dar se renunță la funcția de activare Relu și utilizarea funcției Softmax direct în ecuația de cost (*loss*):

outputs = tf.matmul(outputs_2, weights_3) + biases_3

Aceasta este o practică comună: nu se folosește nici o funcție de activare pentru ieșiri (*outputs*) dar în schimb se încorporează ultima funcție de activare în funcția de cost (*loss*). Se utilizează funcția *TensorFlow* (*tf.nn.softmax_cross_entropy_with_logits)* care este o funcție specială (explicată mai jos) pentru calcularea costului:

loss = tf.nn.softmax_cross_entropy_with_logits(logits=outputs, labels=targets)

Explicația este următoarea: întrucât în calculele noastre se pot întâlni numere foarte mici, iar această instabilitate numerică ar putea pune în pericol modelul, este nevoie de această funcție specială *softmax_cross_entropy_with_logits*. (Nu este întotdeauna necesară, dar este mai sigură folosirea ei). Aceste probabilități nescalate (*logits*) sunt de fapt valori înainte de aplicarea funcției *Softmax*. În Figura 5, **a**-urile sunt *logits*. Funcția așează **a**-urile într-o distribuție probabilistică a cărei sumă este 1. Pentru că aceste *logits* sunt ieșiri (*outputs*) înainte de a fi scalate de softmax, se scrie *logits=outputs.*

În continuare se calculează media costurilor introdusă în variabila *mean_loss* prin funcția *reduce_mean* a lui *TensorFlow*:

mean_loss = tf.reduce_mean(loss)

De menționat că funcția *reduce_mean* se aplică și elementelor unui tensor de n-dimensiuni.

Până aici a fost schițat modelul și au fost calculate costurile (*loss*).

6. Optimizarea și momentum

În etapa a 3-a avem cel mai important ingredient din rețeta algoritmului de învățare a mașinii: optimizarea. Așa cum creierul nostru face o optimizare a tuturor deciziilor noastre de zi cu zi, așa optimizăm tehnica prin care un algoritm învață să prezică, să ghicească ceva. Se ridică problema alegerii metodei de optimizare. În trecut era utilizat **Gradient Descent Optimizer** cu o simplă linie de cod:

optimize = tf.train.GradientDescentOptimizer(learning_rate=0.05).minimize(mean_loss)

În timp, lucrurile au evoluat și a fost demonstrat matematic că metoda ADAM (*Adaptive Moment Estimation*) corelată cu împărțirea setului de date în mini-seturi pentru un proces mai rapid cu menajarea memoriei calculatorului (*batching)* este o alternativă mult mai bună. Deci se poate folosi un modul ce conține optimizarea ADAM:

optimize = tf.train.AdamOptimizer(learning_rate=0.001).minimize(mean_loss)

adică se alege o rată de învățare (*learning rate*) de 0.001 (fiind tot un hiperparametru) cu care se poate jongla pentru găsirea unui rezultat mai bun.

Chiar dacă este asigurată o viteză mai mare de lucru al algoritmului prin folosirea de sub-seturi de date (*batch*), totuși rezultatul este încă aproximativ. Mai mult de atât, în viața reală, funcția cost (*loss*) nu este simplă, ci poate conține mai multe puncte de minim (numite *impostori locali*). Se caută *minimul global* și de aceea este nevoie de așa numitul 'momentum' adică de o anumită viteză care ajută ieșirea din 'gropile' impostorilor de minime. Desigur, asta depinde și de rata de învățare întrucât o rată de învățare mare poate ajuta saltul peste prima 'groapă', dar e posibil să oscileze și să nu atingă niciodată minimul global (Figura 6).

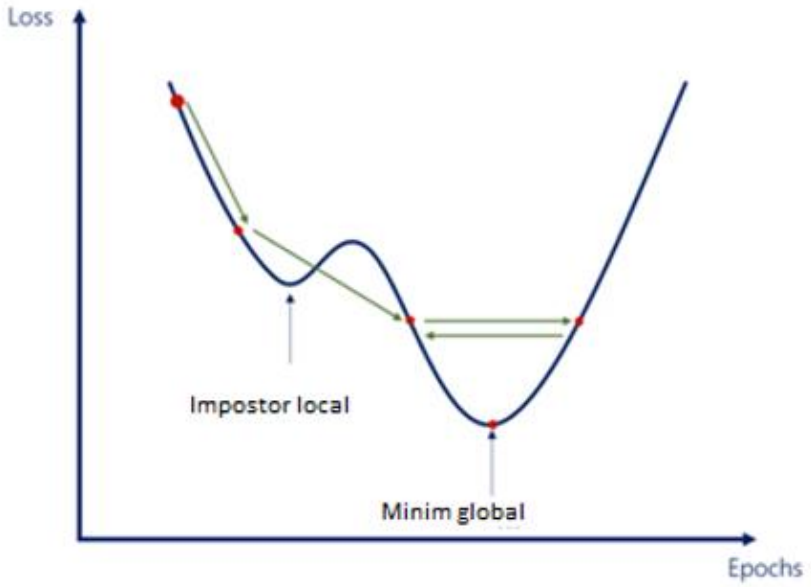


Figura 6. Saltul peste gropile de minime

Așadar, extensia ce se aplică în algoritmul propus (adică acela numit Gradient Descent) este *momentum*. Fără acest inițial *momentum*, nu se poate ajunge la minimum global căci algoritmul se va opri la falsul minim, adică la impostorul minim. Formula de modificare a ponderilor, folosind momentum este dată în Figura 7:

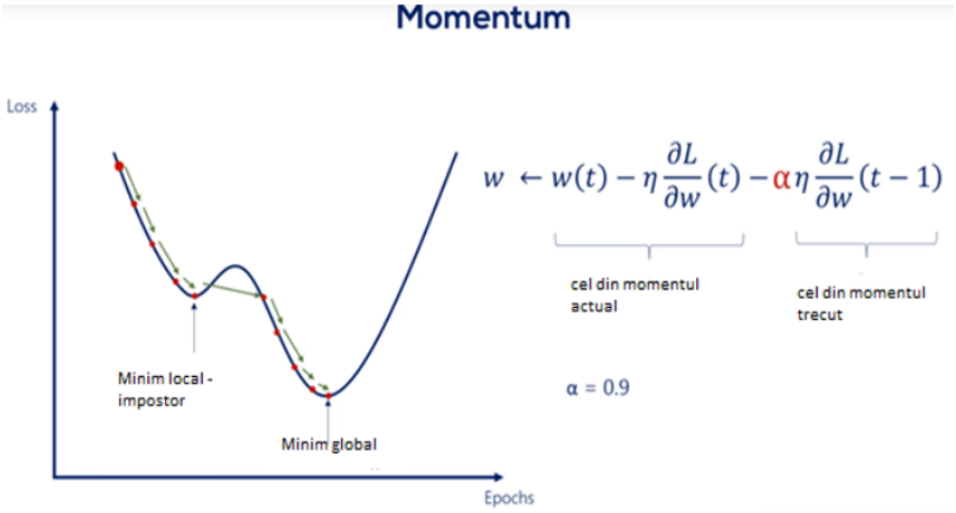


Figura 7. Variația costului (*loss*) și momentum

Referitor la rata de învățare, se poate observa din Figura 8, că o rată mică va atinge minimul dar încet (curba albastră), o rată prea mare va minimiza funcția cost (*loss*) rapid dar până la un punct, după care va oscila și se va opri să scadă (curba bleu), iar o rată și mai mare va face ca valorile funcției să explodeze în direcție de creștere (curba verde).

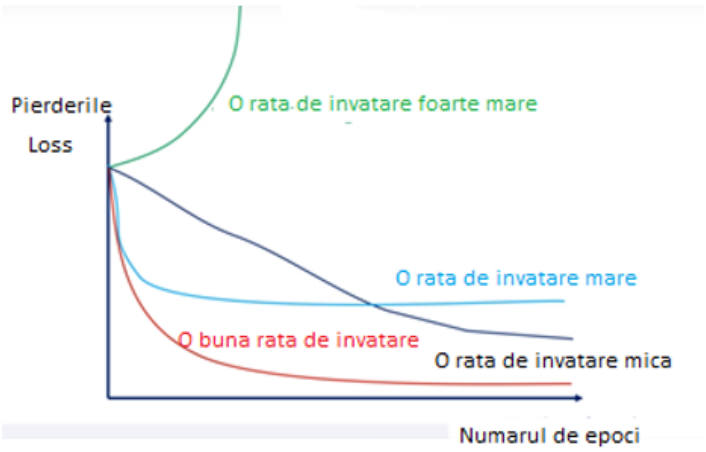


Figura 8. Curba costului în funcție de rata de învățare

Pentru alegerea unei rate optime, au fost inventate două metode de calcul, *Learning Rate Schedules* care sunt *AdaGrad* și *RMSProp* și care vin la pachet cu *TensorFlow*. Acești doi algoritmi sunt relativ noi, (de exemplu *Adaptive Gradient Algorithm* sau *AdaGrad* a apărut în 2011) și, în prezent, sunt considerate tehnologii de ultimă oră în tehnologia de *Machine Learning*. *AdaGrad* variază dinamic această rată de învățare (*learning rate*) la fiecare modificare a ponderilor și individual pentru fiecare pondere (*weight*). *RMSProp* (*Root mean square propagation*) este ușor similar cu AdaGrad având o formulă puțin diferită.

Fără a intra în formulele lor, trebuie spus că există o a treia formulă care le combină și care este superioară. Acest algoritm combină cele două concepte cu *momentum* dând naștere la **ADAM** – *Adaptive Moment Estimation*. Acesta este cel mai modern optimizator folosit în prezent (apărut în 2015). De-abia acum se poate scrie linia de cod:

optimize = tf.train.AdamOptimizer(learning_rate=0.001).minimize(mean_loss)

Pentru a putea continua, trebuie înțeles modul de implementare al operației de divizare a setului masiv de date (*batching*) și de măsurare a acurateții predicției pentru modelul construit. Se dorește procentul de cazuri în care output-ul este egal cu ținta. Sau, cel în care algoritmul atribuie cea mai mare probabilitate a ieșirii care se potrivește cu ținta.

Un exemplu cu imagini poate fi cea mai bună explicație: modelul din figura 9 a ghicit 3 fotografii din 4 și deci acuratețea predicției este de 75%.

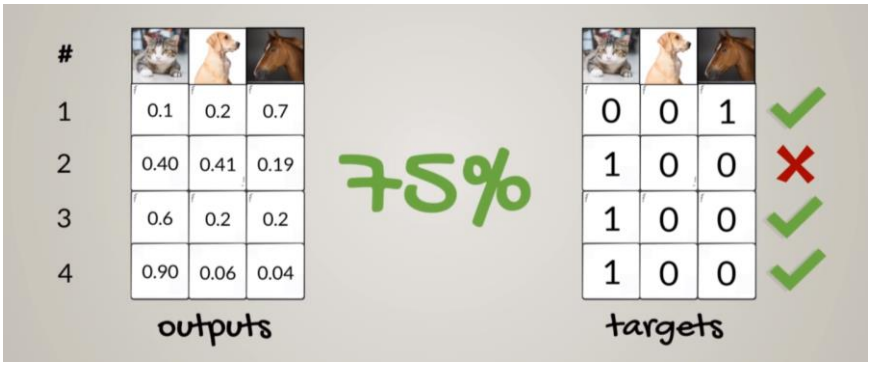


Figura 9. Acuratețea predicției

Cum se exprimă acest lucru în ecuație? Prin funcția **argmax** care returnează indexul celei mai mari valori din ieșire, deci ceea ce se dorește. Așadar **tf.argmax** (cu care vine *TensorFlow*) returnează indexul coloanei la care se găsește cea mai mare valoare. În exemplul din Figura 10, pentru ieșiri returnează vectorul de 4 valori: (2,1,0,0). Pentru ținte returnează (2,0,0,0). Acolo unde indecșii au valori identice, modelul a făcut o predicție bună.

Figura 10. Funcția argmax

Practic este o situație de Adevărat sau Fals. Variabila care conține rezultatul de 0 sau 1 (rezultatul funcției *equal*) se numește *out_equals_target*. Funcția *tf.equal* nu face decât să verifice egalitatea a 2 valori și să returneze o valoare booleană (1 pentru *True* și 0 pentru *False*). Funcția *tf.argmax* are drept al doilea argument cifra 1 pentru că se face referire doar la axa 1, a coloanelor (axis=1). Deci, dacă din cele 10 răspunsuri posibile, se ghicește cifra corectă, atunci variabila este 1.

out_equals_target = tf.equal(tf.argmax(outputs, 1), tf.argmax(targets, 1))

Variabila *out_equals_target* este vectorul (1,0,1,1). Se calculează acuratețea ca fiind media vectorului:

Mean=(1+0+1+1)/4=0.75

Cum *out_equals_target* este boolean, se convertește în tipul *float* cu funcția *cast* care convertește un obiect într-un alt tip de date (*data type*):

accuracy = tf.reduce_mean(tf.cast(out_equals_target, tf.float32))

Așadar, cele 2 linii de cod calculează media acurateții procesului de instruire al modelului, care va fi un număr între 0 și 1.

De aici încolo se poate începe procesul de pregătire al etapei de învățare. Se folosesc obiectele: *sesiuni* și un *inițializator* specifice *TensorFlow*:

sess = tf.InteractiveSession()

initializer = tf.global_variables_initializer()

sess.run(initializer)

În acest momentul are loc divizarea întregului set de date în mini-seturi. Știind că un mini-batch conține 100 de imagini, se cunoaște numărul de mini-seturi (*batch*). Este bine de reamintit că, pentru a mări viteza de procesare, nu numai că se lucrează cu sub-seturi de date, dar se folosește algoritmul *Stochastic Gradient Descent* (SGD) în contrast cu *Gradient Descent* (GD). Cu această alegere se pot calcula gradienții (practic actualizarea ponderilor și bias-urilor) la fiecare mini-procesare de mini-set (procesare mini-set după mini-set), până la terminarea întregului set de date când se poate spune că s-a terminat o epocă. Apoi se reia pentru a doua epocă, a treia, … până la a 15-a epocă (Figura 11).

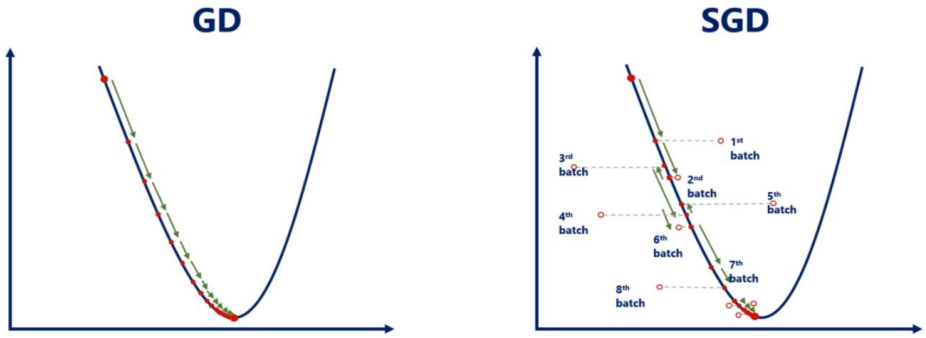


Figura 11. GD *versus* SGD la operația de *batching*

La alegerea dimensiunii sub-seturilor de date, este necesară o dimensiune suficient de mică ca modelul să învețe repede dar suficient de mare ca să păstreze dependențele. Se va alege 100.

batch_size = 100

7. Hiperparametri

Până acum au fost implicați 5 hiperparametri:

- lățimea rețelei (*Width*),
- adâncimea rețelei (*Depth*),
- funcția de activare (*Activations*),
- rata de învățare (*Learning rate*),
- dimensiunea sub-seturilor de date (*Batch size*).

Cu toți acești parametri se jonglează pentru a face un *fine-tuning* al algoritmului.

Numărul de sub-seturi este numărul de exemple de imagini împărțit la dimensiunea (*size*-ul) sub-setului (se ia partea întreagă a diviziunii):

batches_number = mnist.train._num_examples //batch_size

Pentru numărul de epoci alegem 15:

max_epochs = 15

Dacă valoarea costurilor (*loss*) la validare (*validation loss*) începe să crească, se va stopa procesul (*early stopping*). De aceea creăm o variabilă care înregistrează *validation loss* din pasul anterior.

La început, aceasta se va alege suficient de mare pentru siguranța că nu se va declanșa procesul de întrerupere a algoritmului după prima epocă:

prev_validation_loss = 9999999

Acum totul e pregătit pentru a face modelul să învețe.

8. Etapa de Învățare - Learning

Se repetă execuția procesului pentru toate sub-seturile și se reia întregul process de la capăt pentru o altă epocă. Practic există o buclă de execuție (*loop*) pentru epoci care va conține o altă buclă de execuție pentru sub-seturile de date. Ponderile se actualizează la fiecare rulare din bucla de sub-seturi. Numărul care indică cât de greșită este predicția modelului (cost) pe fiecare epocă va fi media pierderilor obținute pe fiecare sub-set. Cu cât acest număr este mai mare, cu atât predicția este mai proastă. Dacă el devine zero înseamnă predicție perfectă. Se setează acest număr la zero la începutul fiecărei bucle. Nu se va pierde nicio informație întrucât totul se va propaga înapoi (*backpropagation*) la sfârșitul fiecărei epoci.

for epoch_counter in range(max_epochs)

În această variabilă se adună pierderile fiecărui sub-set:

curr_epoch_loss = 0.

Se iterează peste toate batch-urile din această epocă:

for batch_counter in range(batches_number)

MNIST Data Provider vine cu această funcție care încarcă batch-urile una după alta în variabilele *input-batch* și *target_batch*:

input_batch, target_batch = mnist.train.next_batch(batch_size)

Se execută pasul de optimizare și se obține *mean_loss* pentru acel sub-set de date.

Apoi se hrănesc intrările și țintele (*inputs* și *targets*) cu datele obținute în *batch*-uri. Se folosește simbolul *under_score* (_) la începutul liniei de cod pentru a ignora o valoare returnată de o funcție, pentru că funcția *optimize* nu returnează nimic:

_, batch_loss = sess.run([optimize, mean_loss],

feed_dict={inputs: input_batch, targets: target_batch})

La sfârșitul fiecărui batch se adaugă *loss*-ul de-abia calculat în variabila care adună toate costurile:

curr_epoch_loss += batch_loss

Până acum variabila ***curr_epoch_loss*** conține suma tuturor *pierderilor* sub-seturilor dintr-o epocă.

Însă se dorește media pentru întreaga epocă (linie de cod care este în afara buclei interioare). Acesta este de fapt costul *(loss)* pe epoca curentă iar '/=' înseamnă operație de împărțire urmată de atribuirea rezultatului variabilei *curr_epoch_loss*:

curr_epoch_loss /= batches_number

La sfârșitul fiecărei epoci se calculează costul de validare. Validarea se face prin propagarea înainte (*forward propagation*) a datelor de validare fără operația de optimizare. Cum pentru instruire s-a folosit:

input_batch, target_batch = mnist.train.next_batch(batch_size)

similar se face pentru setul de validare, știind că numărul de observații este egal cu numărul de eșantioane de validare (*validation samples*):

input_batch, target_batch = mnist.validation.next_batch(mnist.validation._num_examples)

Când se propagă înainte, se înregistrează costul la etapa de validare *(validation loss)* și acuratețea validării. Se vor utiliza două variabile:

validation_loss, validation_accuracy = sess.run([mean_loss, accuracy],

feed_dict={inputs: input_batch, targets: target_batch})

Așadar, a fost calculat *mean loss* și acuratețea pe datele de validare înregistrate în două variabile.

Printăm aceste rezultate:

```
print('Epoch '+str(epoch_counter+1)+
 '. Mean loss: '+'{0:.3f}'.format(curr_epoch_loss)+
 '. Validation loss: '+'{0:.3f}'.format(validation_loss)+
 '. Validation accuracy: '+'{0:.2f}'.format(validation_accuracy * 100.)+'%')
```

Mai este însă ceva de făcut: s-a calculat *validation loss*, dar nu s-a setat un mecanism de stopare premature (*early stopping).*

Acest cod va rupe bucla (*loop*) când *validation loss* este mai mare decât cel anterior:

```
if validation_loss > prev_validation_loss:
break
```

Se va înregistra acest *validation loss* al acestei epoci pentru a fi folosit ca un *loss* anterior în următoarea iterație.

```
prev_validation_loss = validation_loss
print('End of training.')
```

9. Rezultatele

```
print ('End of training.')
```

```
Epoch 1. Training loss: 0.403. Validation loss: 0.197. Validation accuracy: 94.32%
Epoch 2. Training loss: 0.182. Validation loss: 0.149. Validation accuracy: 95.52%
Epoch 3. Training loss: 0.137. Validation loss: 0.130. Validation accuracy: 96.14%
Epoch 4. Training loss: 0.111. Validation loss: 0.123. Validation accuracy: 96.52%
Epoch 5. Training loss: 0.094. Validation loss: 0.119. Validation accuracy: 96.54%
Epoch 6. Training loss: 0.082. Validation loss: 0.100. Validation accuracy: 97.02%
Epoch 7. Training loss: 0.071. Validation loss: 0.091. Validation accuracy: 97.08%
Epoch 8. Training loss: 0.063. Validation loss: 0.093. Validation accuracy: 97.22%
End of training.
```

Instruirea a fost terminată în 8 epoci. Aceasta întrucât costul din faza de validare (*validation loss*) a început să crească și de aceea s-a declanșat mecanismul de stopare. Cu alte cuvinte, modelul a început să …*overfit*. La prima epocă se observă că rezultatul este foarte bun, adică modelul s-a comportat cu acurateţe chiar de la prima epocă pentru că am avut câteva sute de sub-seturi de date care au optimizat algoritmul înainte de sfârșitul acelei epoci. Aceasta este puterea metodei de *batching*! În plus este foarte rapid și dă rezultate impresionante. La final se poate vedea o acurateţe de 97.22 % în epoca 8.

Se propune acum jonglarea cu hiperparametrul numit *lățimea straturilor ascunse* modificându-l din 50 în 100 (*hidden_layer_size=100*), și repetarea întregului process. Normal că toți vechii parametri se vor șterge prin linia de cod *tf.reset_default_graph()*.

Iată noile rezultate:

```
print ('End of training.')
```

```
Epoch 1. Training loss: 0.321. Validation loss: 0.147. Validation accuracy: 95.84%
Epoch 2. Training loss: 0.132. Validation loss: 0.113. Validation accuracy: 96.72%
Epoch 3. Training loss: 0.093. Validation loss: 0.097. Validation accuracy: 97.20%
Epoch 4. Training loss: 0.071. Validation loss: 0.088. Validation accuracy: 97.64%
Epoch 5. Training loss: 0.059. Validation loss: 0.081. Validation accuracy: 97.68%
Epoch 6. Training loss: 0.046. Validation loss: 0.084. Validation accuracy: 97.74%
End of training.
```

S-a reușit o creștere a acurateței modelului, de la 97.22 la 97.74% în 6 epoci.

Dar nu trebuie uitat că această cifră nu este acuratețea modelului, ci doar acuratețea validării (*validation accuracy*). De-abia acum se poate testa modelul cu setul de date de test. Acuratețea finală a modelului vine doar din propagarea înainte (*forward propagation*) prin datele de test și nu din datele de validare.

Recapitulare a celor întâmplate până acum:

- fiecare iterație din marea buclă reprezintă o epocă. Se setează *training loss* la zero la începutul fiecărei epoci;
- pentru fiecare *batch* există o altă buclă (mai mică). S-a iterat peste un număr presetat de loturi/mini-seturi de date care formează setul de date de instruire;
- modificarea (*update*) ponderilor și bias-urilor a fost făcută de atâtea ori de câte sub-seturi există;
- la fiecare iterație în bucla mică, s-a calculat suma tuturor costurilor și la final, când această buclă se încheie, se calculează suma pe toate sub-seturile adică *current_epoch_loss* pentru acea epocă;
- a fost făcută propagarea înainte pe datele de validare și se calculează acuratețea;
- se verifică dacă *validation loss* crește. Dacă nu, se continuă cu următoarea epocă. Când se ajunge la numărul maxim de epoci sau când *validation loss* începe să crească, instruirea se oprește.

10. Etapa de TEST

Acum, după procesul de instruire cu datele de instruire și testul de validare, se testează adevărata putere de predicție pe care o are modelul construit, printr-o execuție cu datele de test, date pe care algoritmul/modelul nu le-a văzut niciodată. Este important de precizat că acest test este instanța absolută și finală. Nu se testează până nu se face jonglarea cu hiperparametri pentru ajustarea modelului.

input_batch, target_batch = mnist.test.next_batch(mnist.test._num_examples)

test_accuracy = sess.run([accuracy],

print (test_accuracy)

Test_accuracy este o listă cu o singură valoare și, pentru a extrage valoarea din ea, se folosește x[0]

*test_accuracy_percent = test_accuracy[0] * 100.*

Formatat pentru procente:

print('Test accuracy: '+'{0:.2f}'.format(test_accuracy_percent)+'%')

Rezultatul este:

```
print('Test accuracy: '+'{0:.2f}'.format(test_accuracy_percent)+'%')
Test accuracy: 97.29%
```

Se observă valori diferite pentru acurateț la validare și la test. Poate fi unul mai bun sau poate fi unul mai slab. Dar asta nu mai are importanță întrucât aceasta este acuratețea finală a algoritmului. Odată ce a fost testat, nu mai este permisă schimbarea modelului.

11. Codul exercițiului în întregime

```
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
input_size = 784
```

```
output_size = 10
hidden_layer_size = 50
tf.reset_default_graph()
inputs = tf.placeholder(tf.float32, [None, input_size])
targets = tf.placeholder(tf.float32, [None, output_size])
weights_1 = tf.get_variable("weights_1", [input_size, hidden_layer_size])
biases_1 = tf.get_variable("biases_1", [hidden_layer_size])
outputs_1 = tf.nn.relu(tf.matmul(inputs, weights_1) + biases_1)
weights_2 = tf.get_variable("weights_2", [hidden_layer_size, hidden_layer_size])
biases_2 = tf.get_variable("biases_2", [hidden_layer_size])
outputs_2 = tf.nn.relu(tf.matmul(outputs_1, weights_2) + biases_2)
weights_3 = tf.get_variable("weights_3", [hidden_layer_size, output_size])
biases_3 = tf.get_variable("biases_3", [output_size])
outputs = tf.matmul(outputs_2, weights_3) + biases_3
loss = tf.nn.softmax_cross_entropy_with_logits(logits=outputs, labels=targets)
mean_loss = tf.reduce_mean(loss)
optimize = tf.train.AdamOptimizer(learning_rate=0.001).minimize(mean_loss)
out_equals_target = tf.equal(tf.argmax(outputs, 1), tf.argmax(targets, 1))
accuracy = tf.reduce_mean(tf.cast(out_equals_target, tf.float32))
sess = tf.InteractiveSession()
initializer = tf.global_variables_initializer()
sess.run(initializer)
batch_size = 100
batches_number = mnist.train._num_examples //batch_size
max_epochs = 15
prev_validation_loss = 9999999.
for epoch_counter in range(max_epochs):
    curr_epoch_loss = 0.
    for batch_counter in range(batches_number):
        input_batch, target_batch = mnist.train.next_batch(batch_size)
        _, batch_loss = sess.run([optimize, mean_loss],
                feed_dict={inputs: input_batch, targets: target_batch})
        curr_epoch_loss += batch_loss
    curr_epoch_loss /= batches_number
    input_batch, target_batch = mnist.validation.next_batch(mnist.validation._num_examples)
    validation_loss, validation_accuracy = sess.run([mean_loss, accuracy],
    feed_dict={inputs: input_batch, targets: target_batch})

    print('Epoch '+str(epoch_counter+1)+
      '. Mean loss: '+'{0:.3f}'.format(curr_epoch_loss)+
      '. Validation loss: '+'{0:.3f}'.format(validation_loss)+
      '. Validation accuracy: '+'{0:.2f}'.format(validation_accuracy * 100.)+'%')
        -   if validation_loss > prev_validation_loss:
    break
    prev_validation_loss = validation_loss
print('End of training.')
```

12. Concluzii

În acest exercițiu au fost explicate cele 4 ingrediente ale unui algoritm ML: *Data*, *Model*, *Objective Function* (cea care duce la rezultatul propus), *Optimization Algoritm* (găsirea soluției optime). Au fost folosite rețele neuronale, funcții de activare și "propagarea înapoi" – *backpropagation* care este esențială în procesul de optimizare. A fost explicat că nu numai aranjarea datelor în mici sub-seturi de date (*batching*) este bună, dar și tehnica de stopare prematură - *early stopping* - este necesară. Au fost folosite inițializări de date prin metoda Xavier, dar a fost observată și cât de importantă este rata de învățare și aplicarea tehnicii ADAM ce include și momentum. În pre-procesarea datelor, având în vedere că s-a avut o problemă de clasificare, a fost necesară codarea datelor folosind *one-hot encoding*.

În exercițiul MNIST, a fost construit un algoritm care recunoaște ce cifră a fost scrisă pe 70.000 de imagini scrise în tot felul de scrieri manuale. S-a reușit ghicirea acestei cifre cu o acuratețe remarcabilă! Întrebarea este: se poate totuși și mai bine? Desigur! Se poate obține o acuratețe și mai bună prin joaca cu hiperparametri și cu parametri numiți ponderi și bias. De exemplu:

- se poate încerca cu un strat ascuns de 200 de noduri;
- se poate încerca o rețea cu mai multe straturi ascunse, de exemplu cu 3 în loc de 2;
- se poate încerca aplicarea diverselor funcții de activare (poate ReLu la primul strat ascuns și Tanh la al 2-lea);
- se pot încerca mini-seturi de diverse mărimi.

În acest exercițiu am asistat la recunoașterea unei cifre dintr-o imagine. De aici încolo, a fost ușor să se meargă mai departe în Inteligența Artificială: recunoașterea unui obiect, al unei figuri umane, al unui text sau chiar al unei emoții.

BIBLIOGRAFIE

1. Chollet, Francois (2018). *Deep Learning with Python*, 2018 by Manning Publications Co. http://bioserver.cpgei.ct.utfpr.edu.br/disciplinas/eeica/papers/Livros/%5BChollet%5D-Deep_Learning_with_Python.pdf.
2. Géron Aurélien (2017). *Hands-On machine Learning with TensorFlow*, published by O'Reilly Media, Inc.
3. http://indexof.es/Varios2/Hands%20on%20Machine%20Learning%20with%20Scikit%20Learn%20and%20Tensorflow.pdf.
4. Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*, MIT Press Cambridge.
5. Nielsen, Michael (2015). *Neural Network and Machine Learning*. Determination Press 2015.
6. *Udemy courses: Python for Data Science and Machine Learning Bootcamp*, https://www.udemy.com/.
7. *Udemy courses: Complete Data Science Bootcamp*, https://www.udemy.com/.

Paul TEODORESCU este român repatriat din Canada unde a lucrat 11 ani ca Software Developer și System Analyst. Acolo s-a specializat în baze de date Oracle și în tehnologiile de Data Warehousing. De profesie inginer, actualmente este asistent în cercetare pe domeniul Inteligenței Artificiale și Machine Learning.

Paul TEODORESCU repatriated after 11 years living in Canada. Over there he worked as Software Developer and System Analyst. He was specialized in Oracle databases and Data Warehousing. As an Engineer, he is currently a research assistant in the field of Artificial Intelligence and Machine Learning.