Escuela Politécnica Superior

Reconeixement Automàtic de Notació Musical

Grau en Enginyeria Informàtica

Treball Fi de Grau

Autor:
Jaume Zaragoza Bernabeu

Tutor/es:
Jorge Calvo Zaragoza

Juan Ramón Rico

Universitat d'Alacant
Universidad de Alicante

Setembre 2017

Reconeixement Automàtic de Notació Musical

Autor
Jaume Zaragoza Bernabeu

Directors
Jorge Calvo Zaragoza
Departament de Llenguatges i Sistemes Informàtics
Juan Ramón Rico
Departament de Llenguatges i Sistemes Informàtics

Grau en Enginyeria Informática

Escuela
Politécnica
Superior

Alacant, 4 de setembre de 2017

La ciència de les operacions,
derivada més especialment de les matemàtiques,
és una ciència en sí mateix,
i té la seua pròpia veritat i valor abstracte

Ada Lovelace.

Índex

Índex de figures

1 Introducció

1.1 Justificació

Actualment, en el món de la música trobem documents amb partitures impreses o escrites a mà. Per poder beneficiar-nos dels avantatges que ens proporcionen les tecnologies de la informació (tals com reproducció, edició i compartició) en la música, cal que aquests documents siguen digitalitzats. Per a aconseguir-ho, es pot recórrer a ferramentes d'edició de partitures per tal de digitalitzar-les manualment, o bé compondre les cançons directament amb aquestes ferramentes. No obstant això, per a l'usuari seria molt més còmode que la digitalització siga automàtica. A més a més, molts compositors prefereixen compondre directament en paper també per raons de comoditat. Altre problema que trobem són la gran quantitat de manuscrits de música creats durant segles, abans de les noves tecnologies, i que ara es troben en perill de desaparéixer a causa del deteriorament del paper o altres mitjans físics pel pas del temps. És necessari beneficiar-se'n també de la digitalització, compartició i sobretot la conservació d'aquestes peces musicals.

En aquest treball es proposa crear un sistema que permeta, mitjançant una imatge de la partitura, detectar i classificar les seqüències de símbols que es troben dins d'aquesta. Amb l'objectiu d'avançar cap a la creació, o millora dels sistemes actuals que pretenen **automatitzar del procés de digitalització de partitures**. Estalviant molta de la feina que suposa fer aquest procés de forma manual. Així com la utilització de les tècniques més actuals i innovadores actualment utilitzades en aquest camp.

Dins del camp de la Visió per Computador trobem el Reconeixement Òptic de Caràcters o en les seues sigles en anglés **OCR**(14). Aquest és el procés de transformació d'imatges amb símbols o caràcters, impresos o manuscrits, a alguna mena de dades amb les quals puguem manipular digitalment. Generalment es busca una forma d'automatitzar aquest procés i es fa servir sovint la intel·ligència artificial junt amb el reconeixement de patrons. On més s'aplica l'OCR és al reconeixement de textos, encara que també s'aplica a la música. Aquest és conegut com a Reconeixement Òptic de Música o les seues sigles en anglés **OMR**(15).

1.2 Plantejament del problema

La realització d'aquest treball té com a objectiu la investigació de les tècniques més actuals de l'àmbit de la intel·ligència artificial que s'estan aplicant a OCR i OMR. Fent servir algunes de les ferramentes més utilitzades d'aprenentatge profund que els últims anys han suposat un canvi prou radical en matèria de visió per computador i reconeixement de patrons. En concret es faran servir xarxes neuronals profundes, que són capaces

d'entrenar, d'una manera més autònoma que altres sistemes anteriorment utilitzats, a partir de grans conjunts de dades. I així aprendre les diferents característiques o propietats d'aquests conjunts de dades per tal de distingir els actuals i els nous, i poder classificar-los correctament. El motiu pel qual s'utilitzaran les xarxes neuronals profundes és la gran millora que han suposat en els últims 5 anys, en la majoria de sistemes d'aprenentatge automàtic, en termes de precisió.

L'enfocament del problema serà, aconseguir grans *dataset*s d'imatges de pentagrames (de l'ordre de 20000 a 100000 imatges). Aquestes imatges seran generades de manera automàtica, contindran seqüències de símbols musicals i les seues corresponents etiquetes. Es generaran diversos *dataset*s de distints tipus, uns que contindran símbols a manera de tipografia i altres amb símbols manuscrits que han sigut prèviament digitalitzats en forma d'imatge de manera individual.

Les diferents parts del sistema que farà servir aquestes imatges són diverses. En primer lloc, la xarxa neuronal que conformarà el cor del sistema. Estarà formada per dues parts ben diferenciades. La primera part que rebrà l'entrada (les imatges), que serà la part convolucional. Aquesta entrenarà per extreure les característiques més rellevants de les imatges (fent servir filtres convolucionals). Aquelles que aporten major informació per a poder identificar els elements que componen la imatge. La segona part serà la xarxa recurrent. Encarregada de rebre la informació extreta de la primera part, la xarxa recurrent a diferència de les xarxes neuronals clàssiques rep com a entrada, també la predicció en l'instant de temps anterior. Amb açò s'aconsegueix un classificador que pot emetre com a eixida per a un exemple, una etiqueta formada per distintes subetiquetes. Ja que el que estem intentant classificar és una seqüència de símbols que tenen relació entre ells. Per tant els símbols que s'han identificat amb anterioritat dins d'una mateixa imatge, tenen a veure amb els següents símbols, i aporten informació sobre quins possibles símbols poden anar després. Per últim, un component que completa la xarxa neuronal i que dóna molt de sentit al treball perquè facilita molt la forma d'entrenar el sistema. Es farà servir una tècnica anomenada Connectionist Temporal Classification(3) o CTC. Aquesta permet que, donat un *dataset* d'imatges amb les seues etiquetes. On tan sols tenim la informació de quins símbols i en quin ordre apareixen però no en quin lloc de la imatge. Entrenar la xarxa neuronal com un tot, on no és necessari segmentar la imatge de manera manual prèviament, ni fer un postprocessament de l'eixida de la xarxa recurrent per determinar l'exactitud de la predicció respecte a l'etiqueta real. Açò dóna la capacitat a la xarxa neuronal per a calcular l'error que comet durant l'entrenament, d'una manera més autònoma i amb menys treball per als desenvolupadors.

Totes aquestes parts produeixen un sol paquet amb la capacitat de ser entrenat de principi a fi, les diferents feines que comporta, automatitzades. Proporcionant l'entrada de les imatges sense cap preprocessament i l'eixida de les etiquetes, sense cap postprocessament. I a banda els híperparametres que calen ser configurats de la xarxa neuronal.

1.3 Fases del Treball

1.3.1 Estudi de la metodologia

Atés que un dels objectius del treball és construir una xarxa neuronal maquetada amb el framework de Deep Learning Keras, escrit en llenguatge Python. El primer pas ha sigut l'estudi de l'exemple que es troba al seu repositori(4). És un exemple que realitza OCR a partir d'imatges amb paraules en anglés. Aquest estudi ha servit per a entendre com enfocar el problema de l'OCR o OMR amb xarxes neuronals. A més de comprendre l'ús de la funció CTC conjuntament amb la xarxa neuronal i dins del framework Keras.

1.3.2 Construcció del sistema

La segona fase del projecte inclou la construcció del sistema base. L'objectiu d'aquesta fase ha sigut agafar el model de xarxa neuronal de l'exemple, amb les modificacions sobre el model estàndard de Keras. Les quals permeten l'entrenament de la xarxa neuronal conjuntament amb la funció CTC. I adaptar-ho a les necessitats d'aquest projecte, concretament als *dataset*s que s'utilitzaran en la fase d'experimentació. Les passes per tal d'adaptar-ho han sigut la creació d'una funció de càrrega en memòria dels *dataset*s i un *callback* que mostre la distància d'edició en cada època de l'entrenament.

1.3.3 Creació del dataset semisintètic

Aquesta fase busca satisfer l'objectiu de tindre un *dataset* d'imatges amb seqüències de símbols musicals manuscrits. Fent servir el conjunt de dades HOMUS(1) el qual conté un símbol manuscrit per cada imatge, recollits de diferents músics. Crear imatges amb el fons d'un pentagrama i introduint seqüències aleatòries d'aquests símbols.

1.3.4 Experimentació

La fase d'experimentació del projecte es divideix en dues parts. En primer lloc la recerca d'un mètode senzill i eficaç de detecció del cap de les notes musicals. Per a poder realitzar la construcció d'un *dataset* semisintètic que tingués en compte l'altura de les notes (tonalitat). En segon lloc totes les proves relatives a la millora de la xarxa neuronal. Les diferents arquitectures i híperparametres provats per adaptar i crear un model de xarxa que maximitze la precisió en el reconeixement de seqüències de símbols musicals.

1.4 Organització de la memòria

A continuació s'explicarà de manera breu els diferents apartats que conté aquesta memòria i el seu contingut.

- Capítol introductori: aquest és el capítol actual. Explica la justificació i motivació del treball així com la perspectiva amb que s'enfocarà i les fases per les quals ha anat passant tota la feina realitzada.

- Capítol de l'estat de l'art: ací s'explicarà tota la base teòrica que hi ha darrere d'aquesta investigació, a més del treball previ formatiu o experimental.
- Capítol de metodologia: el contingut serà l'explicació de les diferents ferramentes de programació utilitzades durant el projecte, com s'ha elaborat el codi font i els detalls sobre el procés de generació dels *dataset*s.
- Capítol d'experimentació: un repàs de totes les proves que s'han dut a terme per determinar quins tipus d'arquitectura i quins híperparametres són els adients per a millorar el rendiment del sistema.
- Capítol de conclusió: un recull de totes les idees extretes de la realització d'aquest treball. Així com quins plantejaments de futur pot tindre aquest problema.

2 Estat de l'art

2.1 Marc teòric

2.1.1 Xarxes neuronals

En l'àmbit de la intel·ligència artificial i l'aprenentatge automàtic trobem les xarxes neuronals artificials. Són un sistema d'interconnexió de neurones (agrupades en capes), inspirat en el funcionament de les xarxes de neurones biològiques del sistema nerviós dels animals. Cada neurona té una sèrie d'entrades amb el seu pes corresponent 2.1.1, les quals se sumen per passar el valor a una funció d'activació que dóna el valor d'eixida. Per a la funció d'activació s'utilitzen funcions que donen valors entre 0 o -1, i 1. La funció utilitzada als seus orígens fou la funció sigmoide: $f(x) = \frac{1}{1+e^{-x}}$. Però en l'actualitat s'ha anat desplaçant per altres funcions que han demostrat major rendiment i\o eficiència, com la tangent hiperbòlica o la ReLU:

$$f(x) = tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.1}$$

$$f(x) = max(0, x) \tag{2.2}$$

L'arquitectura clàssica d'una xarxa neuronal artificial sol ser la de tres capes 2.1.1.

- Capa d'entrada: les neurones reben com a entrada les dades del *dataset*.
- Capa oculta: rep com a entrada l'eixida d'altra capa (la capa d'entrada o altra capa oculta en cas que hi haja més d'una).
- Capa d'eixida: rep com a entrada l'eixida de l'última capa oculta i la seua eixida és la que s'utilitza com a eixida del sistema.
  - Normalment aquesta capa, en problemes de classificació, té tantes neurones com classes es volen classificar. On cadascuna dona com a eixida la probabilitat que té cada classe.

Per tal d'entrenar la xarxa es fa servir l'algorisme *backpropagation* amb descens per gradient. L'objectiu és que, a partir d'un conjunt de parells de dades, per a cada entrada el sistema produïsca l'eixida corresponent. Llavors per ajustar els valors cal calcular la funció de pèrdua de cada exemple, amb aquest valor calcular el gradient i propagar aquest des de l'eixida cap a l'entrada. Ajustant el valor dels pesos per minimitzar l'error, i així amb cada exemple d'entrenament.

El resultat d'aquests entrenaments sol ser un sistema capaç de generalitzar a partir dels exemples d'entrenament, per tal d'emetre l'eixida correcta per a exemples nous desconeguts.

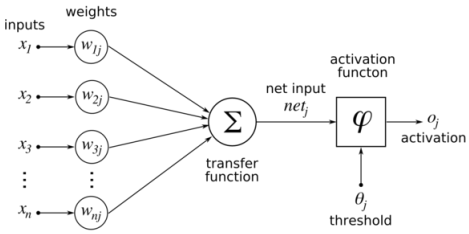


Figura 2.1: Neurona artificial.

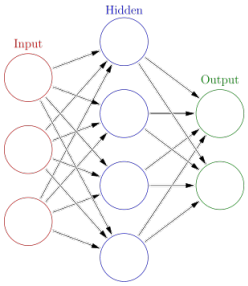


Figura 2.2: Arquitectura clàssica de 3 capes.

Deep Learning

En català aprenentatge profund, és una rama de l'aprenentatge automàtic que fa servir xarxes neuronals amb més de tres capes ocultes. Ha tingut el seu boom recentment en investigació en àmbits com la visió artificial, el processament del llenguatge, o reconeixement de la parla. Ja les bases teòriques d'aquest àmbit havien sigut exposades anys abans, però no fou fins al 2012, quan va explotar amb la participació de la xarxa neuronal AlexNet(7) a la competició ImageNet(12). Principalment aquesta crescuda ha estat donada per la gran millora de capacitat de còmput dels computadors actuals, com per exemple la utilització de GPU's per a còmput general[Sec. 3.1.3]. També la disponibilitat de grans quantitats de dades en l'era de la informació (ja que aquests sistemes necessiten grans volums de dades per a aprendre[Sec. 2.1.1]). Els últims anys la utilització d'aquestes tècniques ha suposat una millora important en la precisió dels sistemes d'aprenentatge automàtic. Inclús ha permés la superació d'estancaments en alguns camps de la visió artificial o en en el reconeixement de la parla.

Xarxes convolucionals

Les xarxes neuronals clàssiques presenten certes limitacions, una d'elles és a l'hora de treballar amb imatges. A partir d'una mida relativament gran d'imatge, és inviable utilitzar-la com a entrada directa a la xarxa neuronal. A més a més, les clàssiques s'han fet servir com a classificadors d'imatges prèviament processades o característiques extretes d'aquestes imatges. Amb mètodes a banda i relativament manuals (ja que han

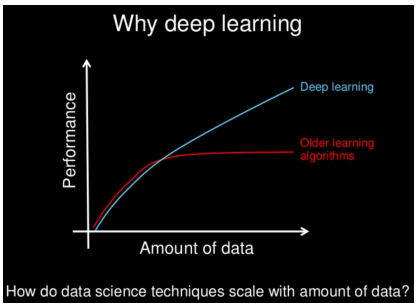


Figura 2.3: Imatge per Andrew Ng(10)

de ser ajustats de manera separada), tals com OpenCV. Donades aquestes limitacions, i inspirant-se biològicament en com funciona el còrtex visual dels animals, sorgeix una evolució anomenada xarxa neuronal convolucional.

El concepte consisteix principalment a substituir les neurones per filtres convolucionals amb els seus corresponents pesos que seran ajustats durant l'entrenament. Podem trobar tres tipus de capes de les quals conformen aquest tipus de xarxes: convolució, activació i votació (en anglés *pooling*).

- **Convolució**. Donada una mida de filtre, el producte escalar entre la finestra de la imatge i els pesos del filtre. A més en una mateixa capa s'entrenen diversos filtres al mateix temps, amb diferents pesos.
- **Activació**. La funció d'activació amb el mateix propòsit que les neurones clàssiques.
- **Votació**. També anomenat *pooling*, consisteix a combinar distintes activacions d'una capa anterior com a una sola entrada per a la capa següent. Pot variar la mida de finestra i el mètode de resum pot ser triant el valor màxim o el valor mitjà dels píxels que es troben en la finestra.

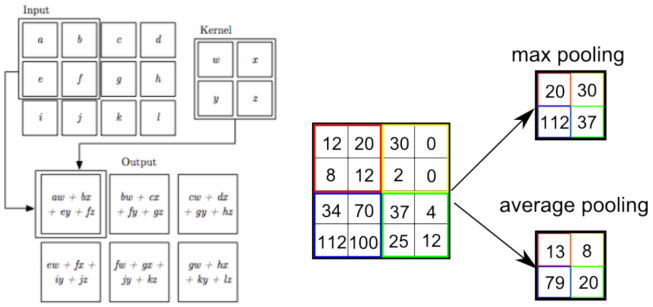


Figura 2.4: Capa de convolució i capa de votació.

Aquestes capes es combinen, cada convolució amb la seua activació corresponent seguida d'una capa de *pooling* per formar una arquitectura convolucional típica. On cada

capa de convolució serà capaç d'aprendre característiques amb un nivell d'abstracció diferent, segons una mida del filtre i de l'entrada. I finalment, després de les convolucions on es duu a terme l'extracció de característiques, es troben les capes de xarxa neuronal clàssica per tal d'aprendre sobre aquestes característiques i realitzar la corresponent classificació.

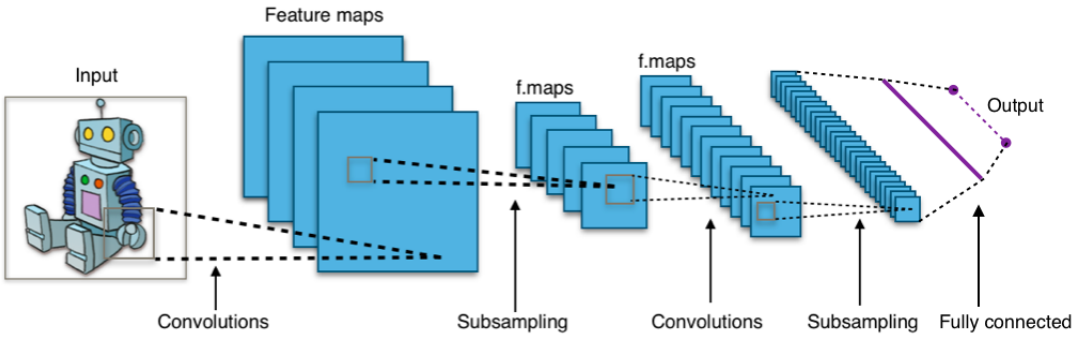


Figura 2.5: Model d'arquitectura convolucional típic.

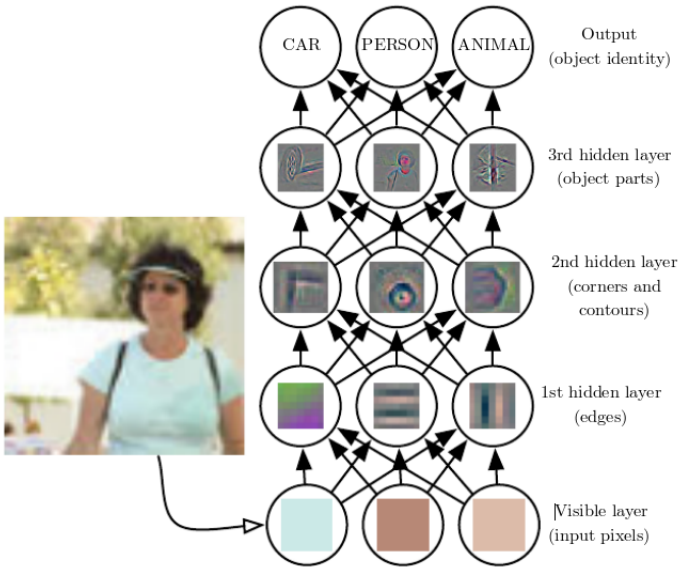


Figura 2.6: Característiques d'imatges amb diferents nivells d'abstracció. Font: Deep Learning Book (2)

Xarxes recurrents

La recurrència en les xarxes neuronals apareix per a classificar exemples d'entrada els quals la seua etiqueta està composta per una seqüència d'etiquetes. En el cas de la traducció de textos o de la música, hi ha escenaris on l'eixida és una seqüència temporal

de símbols o paraules, que estan relacionats entre ells. Perquè la xarxa neuronal siga capaç d'emetre aquesta eixida, quan les dades d'entrada es troben sense segmentar (p. ex. identificar els diferents símbols musicals en una sola imatge sense estar segmentats), és necessari que les neurones tinguen una connexió amb elles mateixes. Açò implica que en cada instant de temps, dins la predicció d'un sol exemple, la neurona tindrà com a entrada l'eixida que ha produït en l'instant anterior.

A més a més, existeixen altres modificacions més avançades com LSTM(6)[Fig:2.1.1] i GRU [Fig:2.1.1]. Aquestes són neurones recurrents amb una petita memòria, que els permet recordar certa informació durant un major període de temps que l'instant anterior. Aquestes neurones són entrenades per saber quant de temps mantenir aquesta informació, quan oblidar-la o quan utilitzar-la en la funció d'activació.

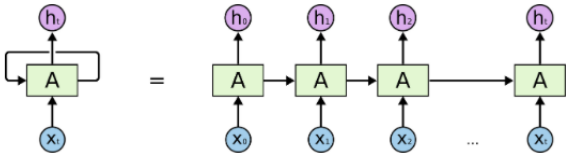
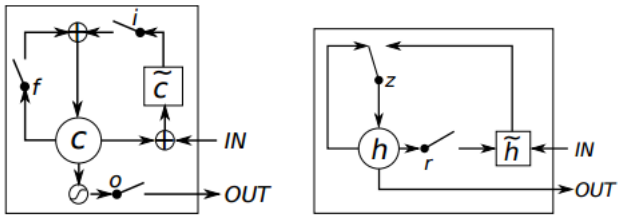


Figura 2.7: Neurona recurrent



(a) Long Short-Term Memory (b) Gated Recurrent Unit

Figure 1: Illustration of (a) LSTM and (b) gated recurrent units. (a) i, f and o are the input, forget and output gates, respectively. c and $\tilde{c}$ denote the memory cell and the new memory cell content. (b) r and z are the reset and update gates, and h and $\tilde{h}$ are the activation and the candidate activation.

El seu ús en conjunt

Per als problemes de visió per computador on s'ha que classificar seqüències, neix la combinació de les xarxes neuronals explicades anteriorment. La idea d'aquesta combinació és substituir les capes clàssiques al final de la xarxa convolucional per capes recurrents. Fent el corresponent mapeig de les característiques de l'última capa convolucional amb la primera recurrent. El que s'anomena camp receptiu. Les connexions de les capes convolucionals amb les clàssiques, on cada neurona de la primera capa clàssica està connectada amb totes les eixides de l'última convolucional, és diferent de les recurrents. Si tenim per exemple una eixida de l'última capa convolucional de 512 filtres amb una mida de 2 píxels d'alçada i 30 d'ample, la part recurrent rebrà com a entrada els 512 filtres

de 2 píxels d'alçada i 1 d'ample. I obtindrà un total de 30 instants de temps. És a dir en cada instant de temps classificarà amb una etiqueta, la informació de la concatenació de les característiques extretes al llarg de les capes de convolució, d'un tros de la imatge o *frame*. Un exemple concret seria el que podem veure a la figura 2.1.2 explicada en la següent secció. Amb aquesta combinació s'obtenen diversos avantatges sobre altres models com per exemple les xarxes convolucionals profundes:

1. Té les mateixes propietats que aporten les convolucions.
   - Aprendre informació representativa directament de les imatges.
   - No requerix cap preprocessament ni característiques extretes manualment.
   - Realitza per si mateix les tasques de binarització, segmentació i localització dels components.
2. Té les mateixes propietats que les recurrents, és capaç de produir seqüencies com a eixida. A més a més, la divisió de la imatge en *frames* ha de ser major a la longitud màxima de les seqüències. Per tant és molt probable que un mateix símbol es trobe dividit en diversos *frames*, i això requereix que la xarxa neuronal sàpiga quina informació ha emés en l'instant anterior.
3. No està subjecte a una longitud determinada de seqüència.
4. Produeix millors resultats o resultats molt competitius en els camps aplicats respecte a les tècniques anteriors.
5. Conté significativament menys paràmetres que les xarxes convolucionals profundes.

L'última peça: CTC

L'inconvenient que tenen les xarxes recurrents i per extensió el seu ús en conjunt amb les convolucionals és: la necessitat de pre-segmentar el conjunt d'entrenament per saber, en cada exemple, quina subetiqueta correspon a cadascun dels instants de temps de la xarxa recurrent. I en molts problemes com el reconeixement d'escriptura, reconeixement de la parla o dels gestos, les dades solen ser sense segmentar. Per exemple, en l'arquitectura que hem estat comentant [Fig: 2.1.2] al mapejar l'eixida de la convolució cap a la recurrent, es divideix la imatge en *frames*. Llavors per poder entrenar la xarxa, és necessari per a cada exemple saber quina subetiqueta correspon a cada *frame* ja que la xarxa recurrent produirà una eixida per a cadascun (cada *frame* suposa un instant de temps). Això pot conduir a errors, ja que en molts casos els *frames* poden tindre ambigüitat sobre quina etiqueta els correspon, o si realment els correspon una etiqueta o no conté cap element. A més a més, també cal un posprocessar l'eixida de la xarxa per transformar-la en la seqüència d'etiquetes.

Per solucionar aquest problema Graves u. a. (3) proposa una tècnica per evitar que aquesta feina siga feta de manera manual i per humans. *Connectionist Temporal Classification* és una forma d'aconseguir que les xarxes recurrents siguen entrenades de principi a fi. Complementa la xarxa aconseguint un mètode per calcular l'error en cada instant de

temps sense necessitat de segmentar les dades. L'únic necessari serà les dades d'entrada sense segmentar (imatges en aquest cas) i la seqüència d'etiquetes corresponent sense necessitat d'adaptar-les al nombre d'instants de temps o *frames* que tinga la xarxa. A més afegeix a la xarxa una classe més de totes les possibles, que simbolitza l'absència de classe o classe en blanc. Amb aquesta, la capa d'eixida de la xarxa tindria una neurona per a cada possible etiqueta més l'etiqueta en blanc. L'activació de les primeres etiquetes correspondrà a la probabilitat d'observar eixa etiqueta, i l'activació de l'etiqueta en blanc suposarà la probabilitat de no observar cap etiqueta. Com podem veure a la figura 2.1.2 la xarxa divideix la imatge en 10 *frames* i aquest és el nombre d'eixides. Per tant si la xarxa produeix l'eixida: `-s-t-aatte` (on els guions signifiquen que no hi ha cap caràcter), per a la paraula *state* en anglés, no cal fer una codificació d'aquesta eixa a l'etiqueta corresponent. I el que és més important, no cal segmentar els exemples del dataset prèviament per saber quin caràcter correspon a cada instant de temps per a calcular l'error. La funció CTC s'encarrega de fer-ho. Basant-se en la idea d'interpretar l'eixida de la xarxa com una distribució de probabilitats sobre totes les possibles seqüències, condicionada a l'etiqueta donada.

2.1.2 Reconeixement Òptic

Reconeixement Òptic de Caràcters

En anglés *Optical Caracter Recognition* i abreviat comunament com OCR, és un camp d'investigació de la visió per computador, el reconeixement de patrons i la intel·ligència artificial. Té com a objectiu identificar els diferents caràcters, impresos o manuscrits i que pertanyen a un determinat alfabet, que puguen aparéixer en una imatge. Per poder digitalitzar la informació de text que tenim en un format físic, i així beneficiar-se de tots els avantatges dins l'era de les tecnologies de la informació. Tals com l'edició, compartició, automatització del procés de digitalització i fins i tot còpies de seguretat.

Als anys 70 es van començar a posar les bases d'aquest camp, però cal remarcar l'important millora que ha sofert i està sofrint en l'actualitat gràcies a l'aprenentatge profund. Trobem que molts dels inconvenients a l'hora d'abordar el problema han sigut superats o molt reduïts a cause d'aquestes tècniques. L'ús de xarxes neuronals fa que els classificadors siguen molt menys sensibles a la presència de soroll, canvi dels colors de les lletres o el fons, existència de separacions variables entre caràcters, superposició dels caràcters o inclús estils diferents d'escriptura o tipografies distintes. A més a més, en l'àmbit de l'escriptura manual, els sistemes estan millorant molt gràcies a la capacitat de generalitzar d'aquestes tècniques, que poden fins i tot reconéixer escrits amb un estil que no han vist amb anterioritat.

Concretament l'any 2015 Shi u. a. (11) proposà un model de xarxa per a tractar aquest problema en el seu article, sobre el qual es basa aquest projecte. Aquest nou model de xarxa és capaç de realitzar les tasques d'extracció de característiques, modelació de seqüències i transcripció. Tot això suposa un seguit de millores respecte a altres sistemes més antics:

- S'entrena com un sol paquet, a diferència d'altres sistemes d'OCR on els diferents

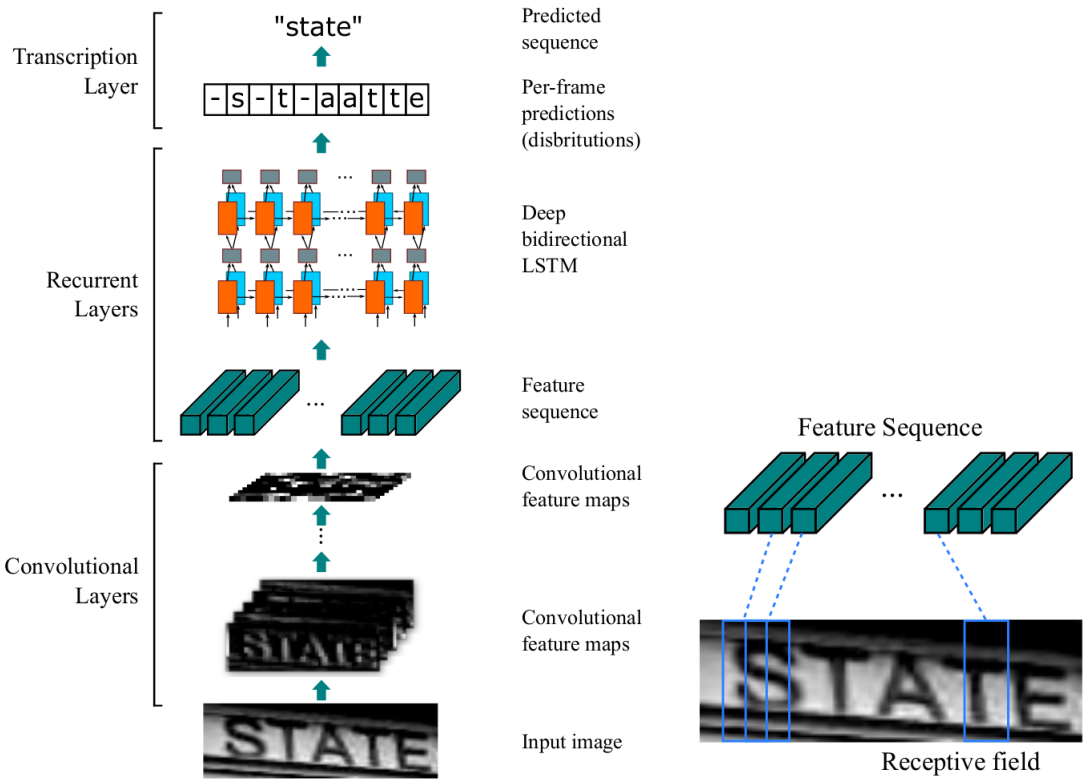


Figura 2.8: Arquitectura de la xarxa neuronal proposada per Shi u. a. (11), a la dreta el camp receptiu.

components s'han d'entrenar de forma separada.

- Pot gestionar exemples amb etiquetes de distinta longitud. El que implica que no és necessari segmentar els símbols ni normalitzar la seua amplària.
- No està subjecte a llenguatges amb lèxic, pot donar bons resultats en ambdós tipus. Una condició necessària per a poder utilitzar-ho en l'àmbit musical.
- Genera un model molt més eficient, ja que millora la precisió amb un nombre significativament menor de paràmetres.

Reconeixement Òptic de Música

El principal mitjà de transmissió de la música de forma no auditiva són les partitures musicals. Símbols musicals escrits o impresos damunt d'un pentagrama. Al llarg de la història hi ha hagut molts artistes que han escrit les seues obres en paper o semblants, i que encara es conserven. Però totes aquestes obres, tenen el perill de perdre's en l'oblit

amb el pas del temps. També a pesar que en l'actualitat existeixen moltes ferramentes d'edició digital de partitures, molts compositors (de tots els gèneres) continuen escrivint a mà les seues creacions. Per poder preservar totes aquelles obres antigues i poder facilitar als actuals autors un mitjà còmode de digitalització, cal un sistema per computador que puga descodificar imatges d'aquestes partitures i passar-les a un format digital. Encara que han estat en continu desenvolupament al llarg dels anys, actualment els sistemes anàlegs de l'OCR dins de l'àmbit de la música, estan encara lluny d'oferir un sistema ideal.

Digitalitzar partitures és una molt bona forma d'oferir una manera de conservar fàcilment les obres, copiar-les, distribuir-les o fins i tot processar-les i analitzar-les. Malauradament per poder dur a terme aquest procés cal que les partitures tinguen un format comprensible per al computador. I la transcripció manual de partitures a formats digitals és una feina que comporta molt de temps. Per això el desenvolupament d'algorismes de visió per computador i reconeixement d'objectes han servit de font d'inspiració per a la creació d'algorismes dins de l'OMR(*Optical Music Recognition*). Algorismes per tal de fer automàtic el procés de reconeixement de símbols musicals a partir d'imatges i codificar-los en un format comprensible per a la màquina.

Durant els anys 80 van sorgir molt programari comercial de reconeixement de notació musical, però cap d'ells amb un nivell de precisió suficient per a considerar-se satisfactori. Amb el temps alguns sí que han millorat el suficient per a poder produir resultats acceptables i que puguen ser utilitzats en la vida diària per a ajudar a la transcripció. Encara que en cap cas, els sistemes comercials suposen una ferramenta per a poder dur a terme la feina de transcripció autònomament, i sobretot si parlem de partitures manuscrites on el seu rendiment està molt lluny de l'ideal.

En aquest projecte i marc teòric ens enfocarem en un mètode fora de línia, on la informació s'extreu d'imatges de partitures ja escrites o impreses. No obstant cal recalcar que existeixen altres mètodes en línia. Els quals tracten de reconéixer els símbols amb la informació que obtenen a partir d'escriure una partitura en una tauleta gràfica, o en un paper amb un bolígraf intel·ligent que registra els moviments.

Aquesta tasca de reconeixement òptic de símbols és una tasca comuna però molt complexa. A la figura 2.1.2 podem trobar les diferents fases per les quals passa aquest procés:

- Preprocessament de la imatge.
- Reconeixement dels símbols
  - Aïllament, detecció i eliminació de les línies del pentagrama.
  - Aïllament dels símbols més primitius.
  - Extracció de característiques i classificació dels símbols.
- Reconeixement de la notació a partir de la combinació dels símbols primitius i l'anàlisi gràfica i sintàctica.
- Transformació en un format de simbologia comprensible per al computador.

Per tal de realitzar correctament aquestes fases, també cal primerament fer una anàlisi i una comprensió exhaustives pel part del programador, de les característiques i propietats dels símbols musicals. És necessari per a aquesta arquitectura, estudiar i conéixer el significat de la simbologia, quins tipus de símbols hi ha, quines característiques gràfiques els defineixen i els diferencien, quina repercussió sintàctica tenen i moltes altres coses.

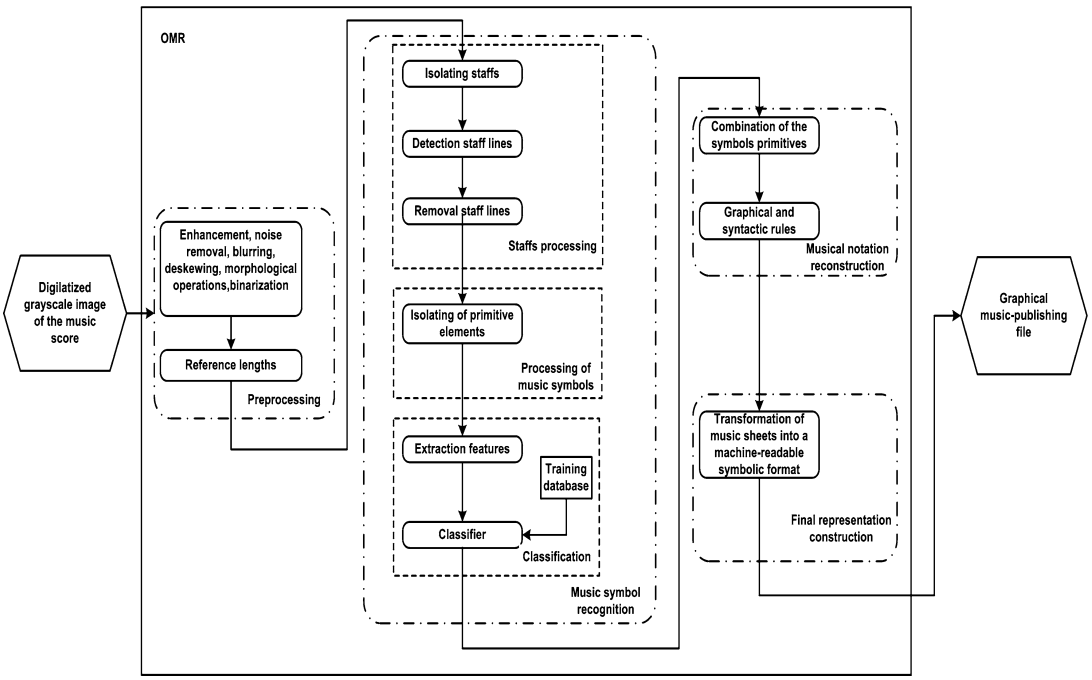


Figura 2.9: Arquitectura típica d'un sistema OMR.

Amb la metodologia utilitzada en aquest treball el que es pretén és abordar el problema des d'una perspectiva diferent. El que permeten les tècniques explicades anteriorment a la secció 2.1.1 respecte a l'enfocament clàssic és construir un sistema que no necessitarà tot l'estudi previ de la simbologia i com diferenciar-la. La part convolucional de la xarxa ajustarà els seus pesos, per tal de realitzar totes les feines que comprenen el processament de les línies del pentagrama i l'extracció de característiques, que permetran a la part recurrent reconéixer la notació present en les imatges. Tot açò de manera automàtica, sense necessitat d'un preprocessament de la imatge, segmentació o aïllament dels símbols primitius i una combinació d'aquests per a una anàlisi sintàctica i gràfica. Tenint com a entrada les imatges directament sense processar (tan sols un possible redimensionament), i produint una eixida que serà directament la notació musical que apareix dins la imatge. En aquest projecte no es considerarà la part de la transformació en un format digital, ja que tan sols es pretén investigar i experimentar per millorar la precisió en el reconeixement.

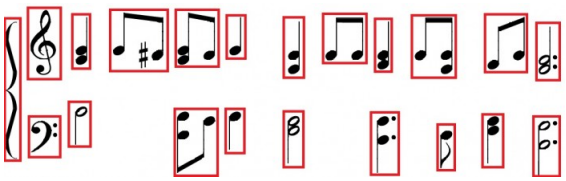


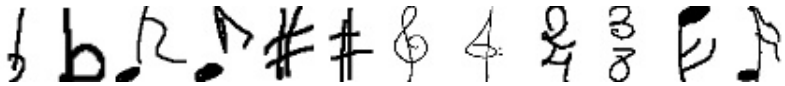
Figura 2.10: Exemple d'una imatge de partitura amb les línies del pentagrama esborrades i els símbols segmentats.

2.2 Treballs previs

2.2.1 Classificació de símbols musicals amb xarxes convolucionals

Amb anterioritat a aquest projecte s'ha realitzat una pràctica en l'assignatura Desafiaments de Programació, que ha servit de formació teòrica prèvia i presa de contacte amb el món de l'OMR. L'objectiu d'aquesta pràctica era crear un classificador de símbols musicals manuscrits, concretament el *dataset* HOMUS(1). Cada exemple del *dataset* estava format per una imatge amb la traça (recollida amb la pantalla tàctil d'una tauleta) d'un símbol musical aïllat, amb la seua corresponent etiqueta (un total de 32 classes de símbols). En aquest cas es va fer servir una xarxa neuronal convolucional per classificar els símbols, partint del model de la primera xarxa neuronal convolucional(8). Utilitzada per classificar dígits manuscrits.

Figura 2.11: Imatges de símbols musicals d'HOMUS.

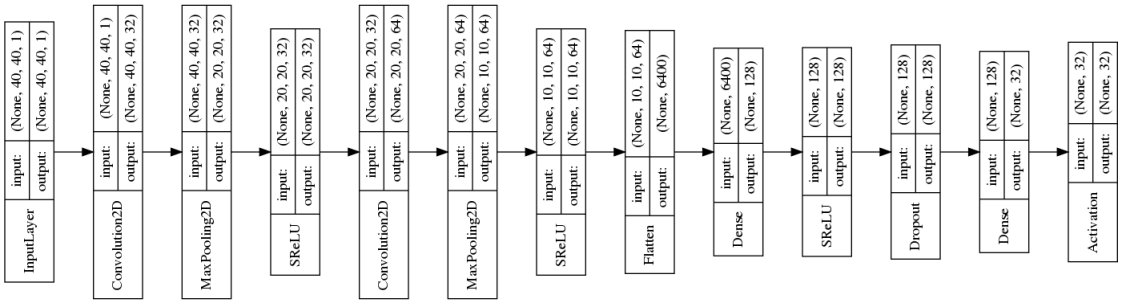


Es van realitzar distintes proves per ajustar els hiper-paràmetres de la xarxa, adaptar l'arquitectura i les funcions d'activació utilitzades. A més de fer estudis estadístics i comparatius de models de xarxa amb distints paràmetres, ús de tècniques d'augment de dades i comparació amb altres classificadors clàssics tals com *Support Vector Machines*. Finalment es va obtenir un classificador que rondava el 98% de precisió, una gran millora respecte a classificadors més clàssics que no fan ús de convolucions i que podien arribar fins a un 63% de precisió.

2.2.2 Proves preliminars

Abans de la realització de la proposta de treball es van realitzar algunes proves per tal d'assegurar que el problema era abordable. Es va procedir a descarregar i provar el codi de la xarxa neuronal de l'article de Shi u. a. (11), fent servir el *dataset* sintètic[Sec. 3.3.1], generat prèviament. Amb açò es va poder comprovar que era factible fer funcionar una xarxa neuronal amb aquestes dades, i amb la configuració proposada per l'article.

Figura 2.12: Arquitectura final de la xarxa.



3 Metodologia

3.1 Ferramentes

3.1.1 Llenguatge principal: Python

Un llenguatge de programació d'alt nivell, multiparadigma i interpretat, que permet desenvolupar ràpidament programes o scripts. Amb gran llegibilitat de codi i que permet expressar les accions amb un nombre menor de línies de codi. Tot açò amb una gran optimització al darrere i afegint molt pocs costos computacionals o de sobrecàrrega amb l'alt nivell d'abstracció. Abans del treball hi tenia pocs coneixements d'aquest llenguatge, però ha sigut molt senzill i ràpid aprendre coneixent altres llenguatges clàssics com C/C++ i Java. El fet de ser interpretat ha ajudat molt a depurar els errors amb facilitat i poder maquetar el codi ràpidament per als experiments.

Finalment altre aspecte a destacar és la facilitat d'ús de llibreria estàndard. S'han fet servir els mòduls:

- `os`: per a tractar amb els directoris i obtenir llistes d'arxius.
- `sys`: gestió dels paràmetres del programa i eixida del programa.
- `json`: per guardar els diccionaris.
- `re`: expressions regular per filtrar noms d'arxius.

3.1.2 API: Keras i Tensorflow

L'API triada per a Deep Learning ha sigut Keras, donat que era coneguda amb anterioritat. Atorga una gran capacitat de prototipatge respecte a altres API, gran modularitat i extensibilitat, i suport per a xarxes convolucionals, recurrents i el seu us en conjunt. A més de proporcionar l'exemple que ha servit d'inici per a l'experimentació d'aquest treball.

Keras permet la utilització de diversos *backends*, en aquest cas s'ha utilitzat Tensorflow, amb una comunitat i desenvolupament prou actius. A més de proporcionar facilitat i optimitzacions per a executar-se en targetes gràfiques Nvidia3.1.3.

Figura 3.1: Logos de Keras i Tensorflow.

3.1.3 GPGPU: CUDA

Un dels principals motius pels quals aquestes tècniques d'aprenentatge automàtic hagen ressorgit, i que han permés l'experimentació d'aquest treball en un temps assequible. La computació de propòsit general sobre processadors gràfics (amb les seues sigles GPGPU en anglés) dóna grans optimitzacions i reduccions de temps de còmput per a l'entrenament de xarxes neuronals d'aquestes dimensions (la utilitzada en el projecte està entre 5 i 10 milions de paràmetres). Concretament la plataforma de computació paral·lela CUDA és la que s'ha utilitzat en aquest projecte. Ja que és quasi l'única utilitzada en aquest àmbit i en frameworks de programació com Tensorflow. S'ha pogut executar en el servidor del Departament de Llenguatges i Sistemes Informàtics on es troba una targeta gràfica Nvidia model GTX980.

3.1.4 Tractament d'imatges: PIL i OpenCV

Per a manipular, ajuntar, dibuixar i crear imatges fàcilment en Python s'ha utilitzat la llibreria Pillow, un *fork* del projecte Python Image Library que es troba abandonat. Aquesta llibreria aporta els mateixos avantatges amb les imatges que Python aporta amb la programació. Operar de manera flexible i amb molt poques línies de codi amb les imatges, i una gran facilitat d'aprenentatge són els principals beneficis.

Per altre lloc per a la realització d'algunes operacions de visió artificial s'ha utilitzat la llibreria OpenCV. Principalment un detector de cercles i un detector de taques en imatges.

3.1.5 Estructures de dades: Numpy

Una de les llibreries d'estructures de dades més potents en Python és Numpy. Entre els seus avantatges es troben: la flexibilitat, l'optimització, la simplicitat i facilitat d'ús. Tot açò ha fet que siga l'estructura de dades per excel·lència utilitzada en la majoria de frameworks d'aprenentatge automàtic i per extensió en aquest projecte.

3.1.6 Control de versions: GitHub

Github és una plataforma que proporciona interfície web a tot tipus de repositoris Git. Actualment és la plataforma més popular i amb major nombre de repositoris públics i de programari lliure. Proporciona moltes ferramentes que complementen a Git per tal d'ajudar a la col·laboració de gran nombre d'usuaris. S'ha utilitzat per publicar el codi font d'aquest projecte(17), distribuint-lo sota Llicència Pública de GNU versió 3. També ha servit per a poder sincronitzar fàcilment el projecte entre distints dispositius i facilitar el desenvolupament del programari.

3.1.7 Notació musical: Lilypond

Un programa d'edició de partitures i tipografia musical que forma part del projecte GNU. Utilitza un llenguatge específic per descriure la música, que després és interpretat per crear les imatges amb les partitures. S'ha utilitzat per a la generació de *dataset*s.

3.1.8 Altres: Bash i SSH

Donat que el sistema operatiu triat per executar i desenvolupar el projecta és GNU/Linux el llenguatge d'*scripting* Bash ha sigut necessari en algunes ocasions. Principalment per llençar diversos tipus de proves a execució en el servidor o esperar la finalització d'altres processos per executar proves. Açò ens ha atorgat major automatització en les proves i experimentació. També ha sigut necessari l'ús de la ferramenta *Secure Shell* o SSH, per a l'accés al servidor remot on s'han dut a terme les proves amb la xarxa neuronal.

3.2 Elaboració del codi font

3.2.1 Punt de partida

Com s'ha comentat amb anterioritat, un dels motius que ha permés crear un sistema d'OMR sense la necessitat d'un pos/preprocessament de l'entrada i eixida del sistema, ha sigut la tècnica CTC. A més a més l'API triada per a aquest treball ha sigut Keras. Per això s'ha pres com a punt de partida del projecte l'exemple que aquesta llibreria exposa al seu repositori(4). Aquest exemple pretén fer una demostració del funcionament de la tècnica CTC (implementada com a funció als respectius *backends*, però no com a tal en l'API) conjuntament amb Keras. I ho demostra fent una xarxa neuronal que aborda el problema d'OCR, que com ja s'ha comentat abans, és molt similar al problema que ens ocupa. Aquestes raons fan l'exemple de Keras, un bon punt de partida com a base del codi font.

En primer lloc el que es va fer va ser provar el funcionament d'aquest codi, sense modificacions, al servidor del departament. Una vegada comprovat el seu funcionament, s'ha estudiat tot el funcionament del codi exemple, per comprendre com abordar el problema d'OCR amb aquesta API i quines parts d'aquest són interessants per al nostre sistema. Concretament les parts que s'han utilitzat ha sigut tota la part que defineix l'arquitectura de la xarxa neuronal i els afegits pertinents per entrenar-la amb CTC. Es va descartar gran part del codi, ja que s'utilitzava per generar el seu *dataset* d'imatges amb paraules en anglés, i poder visualitzar el progrés de l'entrenament amb aquest *dataset* en concret.

3.2.2 Diccionari d'etiquetes

A l'hora de la creació dels *dataset*s, les etiquetes assignades a cada imatge han de ser comprensibles per als humans. Per poder saber quins són els símbols de cada imatge o per realitzar tasques de depuració posteriors durant l'entrenament. Per altra banda, la xarxa neuronal necessita com a entrada les etiquetes codificades en números des de zero fins al nombre de classes total que volem classificar. Amb l'objectiu de codificar i no perdre la informació de les etiquetes s'ha creat un *script* en Python. En primer lloc crea un diccionari en format JSON que codifica les classes en parells `"music-symbol": "number"`.

```
f = open(path + "labels.txt")
    for line in f:
        for (i,token) in enumerate(line.split()):
            if token not in categories:
                categories.append(token)
            if i > max_length:
                max_length = i

words = dict()
for i,cat in enumerate(categories):
    words[cat] = i

with open(dict_path,'w') as fp:
    json.dump(words, fp, sort_keys=True, indent=4)
```

En segon lloc utilitza el diccionari per crear un nou fitxer amb les etiquetes de cada imatge codificades.

```
f = open(path + "labels.txt")
    aux = ""
    for line in f:
        tokens = line.split()
        for i in range(len(tokens)):
            aux = aux + str(words[tokens[i]]) + ' '
        aux = aux + '\n'

f = open(path + "labels_cod.txt", 'w')
f.write(aux)
```

Exemples de codificació de fitxers:

```
# Etiquetes originals
EIGHTH_REST SIXTEENTH_REST EIGHTH–E5b HALF_REST EIGHTH_REST SIXTEENTH–G4f
SIXTEENTH–B4b EIGHTH_REST HALF_REST EIGHTH_REST SIXTEENTH_REST EIGHTH_REST
EIGHTH–F5f HALF–C5b SIXTEENTH–F4b QUARTER–F5b SIXTEENTH–E5b
# Etiquetes codificades
0 1 2 3 0 4
5 0 3 0 1 0
6 7 8 9 10
# Diccionari
"EIGHTH-E5b": 2,
"HALF_REST": 3,
"SIXTEENTH_REST": 1,
```

3.2.3 Càrrega de dades

Com que en aquest sistema s'han utilitzat diversos *dataset*s, que no tenen res a veure amb els de l'exemple, s'ha escrit des de zero una funció que s'encarregue de carregar les dades. Aquesta funció llig les imatges del directori, d'una a una amb l'etiqueta corresponent, i les estructura en arrays de quatre dimensions. Una per al nombre d'exemples, altra per a l'alçada de la imatge, altra per a l'amplada de la imatge i altra amb el nombre de canals de color, que en aquest cas sols és un, ja que són en blanc i negre. També normalitza els valors de cada píxel a un interval [-1,1].

L'eixida d'aquesta funció és un diccionari amb distints paràmetres d'entrada i eixida que requereix la xarxa neuronal i la funció de CTC:

1. L'array d'entrada que conté els valors dels píxels de les imatges.

2. L'array amb un array que conté la seqüència de símbols, per a l'etiqueta de cada exemple.

3. Els valors de l'amplada de la imatge d'entrada. En aquest cas són tots iguals, ja que tots els exemples es redimensionen a una mida fixa per simplificar. Però el sistema permet com a entrada imatges de longitud distinta.

4. Els valors amb la longitud de cada seqüència de símbols per a l'etiqueta de cada imatge.

5. Un array amb zeros per a l'eixida de la funció d'error.

```
for filename, lb in zip(paths, labels):
    im=Image.open(filename).resize((img_w,img_h)).convert('L')
    im=ImageOps.invert(im)      # Meaning of grey level is 255 (black) and 0 (
        white)
    label = np.fromstring(lb, dtype=int, sep=' ')
    label_length.append(len(label))
    fill = np.full((lb_max_length - len(label),), -1, dtype=int)
    class_list.append(np.append(label,fill))
    image_list.append(np.asarray(im).astype('float32'))

n = len(image_list)     # Total examples
input_shape = (img_h, img_w, 1)
X = np.asarray(image_list).reshape(n, img_h, img_w, 1)
class_list = np.asarray(class_list)
label_length = np.asarray(label_length)

# Normalize
mean_image = np.mean(X,axis=0)
X -= mean_image
X /= 128

inputs= { 'the_input' : X,
          'the_labels' : class_list,
          'input_length' : np.full((n,), img_w // downsample_factor ** 2 - 2),
          'label_length' : label_length,
        }
outputs= {'ctc' : np.zeros((n,), dtype=int)}
```

3.2.4 Model de xarxa neuronal

Per a la creació de la xarxa neuronal s'han creat diverses funcions. En primer lloc una funció que crea blocs de convolució, segons els paràmetres donats. La configuració conté el nombre de capes de convolució, el nombre de filtres de cada capa, la mida dels filtres i la mida dels filtres de votació.

```
def conv_block(input, config, name='1'):
    for i in range(config[0]):
        input = Convolution2D(config[1], config[2], config[2], border_mode='same',
            activation='relu', name='conv'+ name + '_' + str(i+1))(input)
    input = MaxPooling2D(pool_size=config[3], name='max'+name)(input)
return input
```

En segon lloc la funció que crea els blocs recurrents. La configuració conté el nombre de capes recurrents i el nombre de neurones per capa. La funció d'activació per defecte per a les recurrents és tanh.

```
def rnn_block(input, config):
    for i in range(config[0]):
```

```
        input = Bidirectional(GRU(config[1], return_sequences=True, name='gru'+str(i)))(input)
    return input
```

En tercer lloc la funció encarregada d'invocar a la funció de CTC. Es descarten els dos primers valors de la xarxa recurrent perquè solen ser brossa.

```
def ctc_lambda_func(args):
    y_pred, labels, input_length, label_length = args
    y_pred = y_pred[:, 2:, :]
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)
```

Per últim la funció encarregada de construir el model de xarxa i compilar-lo. Establint les capes de convolució, el mapeig a la part recurrent, les capes recurrents, la capa d'eixida i els paràmetres necessaris perquè CTC calcule l'error. Com que Keras no inclou l'ús de CTC, aquesta funció s'introdueix amb una funció lambda que retorna el valor d'error que calcula CTC.

```
ef create_rnn(input_shape, lb_max_length, nb_classes, pool_size=2):
    img_h = input_shape[0]
    img_w = input_shape[1]

    # Network parameters
    nb_filters1 = 64
    nb_filters2 = 128
    nb_filters3 = 256
    nb_filters4 = 512
    filter_size1 = 3
    filter_size2 = 3
    filter_size3 = 3
    filter_size4 = 3
    pool1 = (pool_size, pool_size)
    pool2 = (pool_size, 1)
    rnn_size = 256

    input_data = Input(name='the_input', shape=input_shape, dtype='float32')
    inner = conv_block(input_data, (1, nb_filters1, filter_size1, pool1), name='1')
    inner = conv_block(inner, (1, nb_filters2, filter_size2, pool1), name='2')
    inner = conv_block(inner, (2, nb_filters3, filter_size3, pool2), name='3')
    inner = conv_block(inner, (2, nb_filters4, filter_size4, pool2), name='4')

    conv_to_rnn_dims = (img_w // (pool_size ** 2), (img_h // (pool_size ** 4)) * nb_filters4)
    inner = Permute((3,1,2))(inner)
    inner = Reshape(target_shape=conv_to_rnn_dims, name='reshape')(inner)
    gru = rnn_block(inner, (2,rnn_size))
    inner = Dense(nb_classes, init='he_normal',
                  name='dense2')(gru)
    y_pred = Activation('softmax', name='softmax')(inner)
    # Imprimir per pantalla la configuracio de la xarxa
    Model(input=[input_data], output=y_pred).summary()

    labels = Input(name='the_labels', shape=[lb_max_length], dtype='float32')
    input_length = Input(name='input_length', shape=[1], dtype='int64')
    label_length = Input(name='label_length', shape=[1], dtype='int64')
    loss_out = Lambda(ctc_lambda_func, output_shape=(1,), name='ctc')([y_pred, labels, input_length, label_length])
```

```
38
39      model = Model(input=[input_data, labels, input_length, label_length], output=[
            loss_out])
40      model.compile(loss={'ctc': lambda y_true, y_pred: y_pred}, optimizer='rmsprop'
            )
41      test_func = K.function([input_data],[y_pred])
42
43  return model, test_func
```

El model de xarxa final que s'ha obtingut és semblant al proposat per l'article de Shi u. a. (11) amb algunes modificacions. Seguint l'esquema de dalt a baix, els blocs convolucionals:

1. Una capa de 64 filtres convolucionals i una capa de votació de 2x2.
2. Una capa de 128 filtres convolucionals i una capa de votació de 2x2.
3. Dues capes de 256 filtres convolucionals i una capa de votació de 2x1.
4. Dues capes de 512 filtres convolucionals i una capa de votació de 2x1.

El motiu de l'ús de capes de votació de 2x1 és per evitar reduir més l'ample de la imatge i que hi haja el nombre de *frames* necessaris per a la classificació. Açò ens dona com a eixida final de les convolucions una imatge amb 30 *frames* de dos píxels d'altura, de cada un dels 512 filtres aplicats en l'última capa de convolució. La part recurrent rep per a cada neurona 1024 entrades en cada instant de temps, i en total 30 instants de temps. Les capes usades són dues GRU bidireccionals amb 256 neurones. L'ús de bidireccionals serveix per a que cada neurona tinga en cada instant, l'eixida emesa en l'instant anterior i en l'instant següent. Açò aporta major informació per a fer les prediccions. Finalment la capa d'eixida amb tantes neurones com classes possibles, la funció lambda que retorna el càlcul de l'error de CTC i l'optimitzador RMSProp.

La funció torna el model compilat i una funció que permet passar valors d'entrada manualment a la xarxa per obtenir la predicció d'eixos valors.

3.2.5 AccCallback

L'API Keras té algunes funcions per mostrar diferents mètriques durant l'entrenament(p. ex. la precisió o percentatge d'exemples encertats), que no són usades en el mateix entrenament però si serveixen per a observar el progrés d'aquest. Encara que Keras no té cap funció integrada de mètriques que calculen la precisió d'un sistema que té com a eixida seqüències de distintes longituds. El valor que sempre es mostra, a banda de les mètriques, és l'error comés pel sistema en cada època (utilitzat per actualitzar els pesos de les neurones). I aquest valor no proporciona gran informació per als humans, ja que segons el problema abordat o el tipus de dades que utilitzem, els valors poden ser molt distints. A més a més, que en aquest cas el valor de l'error el calcula la funció CTC, el que fa que siga més complex d'entendre.

En aquest projecte s'ha creat un *callback* que s'executa en acabar cada època i mostrar per pantalla la distància d'edició mitjana i la distància d'edició mitjana normalitzada.

Aquestes mètriques són comunament usades en OCR per a determinar quina és la precisió del model durant l'entrenament o en els tests. La distància d'edició entre dues cadenes mesura la diferència entre elles. El nombre mínim d'operacions d'inserció, substitució o supressió d'un caràcter, necessàries per a transformar una de les cadenes en l'altra. Aquesta mesura també es pot extrapolar a música, considerant cada símbol musical com si fos un caràcter. A més d'això les etiquetes amb les quals s'han treballat en la xarxa neuronal hi són codificades en números, per tant s'ha utilitzat una implementació de l'algorisme que funcione amb números i no caràcters. En cada test es calcularà la distància d'edició per a cada exemple, entre la cadena predita per la xarxa neuronal i el seu *ground truth*. El *callback* mostrarà finalment la distància d'edició mitjana de tots els exemples de test i la distància d'edició mitjana normalitzada a valors d'entre 0 i 1. Aquesta última mètrica té la intenció de mostrar una mesura similar al que seria la mitjana de símbols fallats.

Aquest *callback* guarda el conjunt de dades de test i una funció per avaluar exemples de test amb la xarxa neuronal i saber quina és la seua predicció. En finalitzar cada època avalua[1]cada exemple i calcula la distància d'edició respecte al seu *ground truth*.

Finalment es mostra per pantalla les mètriques per al conjunt de test. Funció que calcula la mínima distància d'edició(5) entre cadenes de números:

```
n, m = len(a), len(b)
if n > m:
    # Make sure n <= m, to use O(min(n,m)) space
    a,b = b,a
    n,m = m,n

current = range(n+1)
for i in range(1,m+1):
    previous, current = current, [i]+[0]*n
    for j in range(1,n+1):
        add, delete = previous[j]+1, current[j-1]+1
        change = previous[j-1]
        if a[j-1] != b[i-1]:
            change = change + 1
        current[j] = min(add, delete, change)

return current[n]
```

Bucle per avaluar les dades del conjunt de test per trossos, i acumular els valors de les eixides:

```
count = 0
while count <= self.inputs['the_input'].shape[0]:
    if count==0:
        func_out = self.test_func([self.inputs['the_input'][count:count + self.
            batch_size]])[0]
```

[1]La funció que avalua els exemples d'entrenament dins del *callback* rep per paràmetre un array amb tots els exemples a avaluar. Posteriorment carrega tots els exemples a avaluar en memòria principal. Els conjunts d'exemples de test solen estar entre 1000 i 20000 i açò provocava que en alguns casos l'entrenament s'interrompia, a causa que la targeta gràfica es quedava sense memòria per carregar totes les dades. Finalment es va solucionar fent diferents cridades a la funció dividint el conjunt de test en grups de 64 o 128.

```
5       elif count + self.batch_size <= self.inputs['the_input'].shape[0]:
6           func_out = np.append(func_out,
7                   self.test_func([self.inputs['the_input'][count:count + self.
                        batch_size]])[0],
8                   axis=0)
9       else:
10          func_out = np.append(func_out,
11                  self.test_func([self.inputs['the_input'][count:]])[0],
12                  axis=0)
13      count += self.batch_size
```

Bucle principal del *callback* que calcula les distàncies d'edició per al conjunt de dades de test ja avaluades:

```
1   ed = 0
2   mean_ed = 0.0
3   mean_norm_ed = 0.0
4   for i in range(func_out.shape[0]):
5       rnn_out = []
6       output = []
7       prev = -1
8       for j in range(func_out.shape[1]):
9           out = np.argmax(func_out[i][j])
10          rnn_out.append(out)
11
12          # Elimina espais i repetits
13          if out != prev and out != self.blank:
14              output.append(out)
15          prev = out
16
17      ed = self.levenshtein(self.inputs['the_labels'][i].tolist(),output)
18      mean_ed += float(ed)
19      mean_norm_ed += float(ed) / self.inputs['label_length'][i]
20  mean_ed = mean_ed / len(func_out)
21  mean_norm_ed = mean_norm_ed / len(func_out)
22  print("--Mean edit distance: %0.3f, mean normalized edit distance: %0.3f\n" % (
        mean_ed, mean_norm_ed))
```

3.2.6 Generador semisintètic

En aquesta secció es descriurà el funcionament del programa en Python per generar *datasets* semisintètics. L'explicació dels objectius d'aquest *datasets* estan explicats en la secció 3.3.2.

Prèviament a la creació del generador s'han processat les imatges d'HOMUS, ja que les imatges originals tenen el fons negre amb el traç en blanc i de la grossor d'un o dos píxels. Amb l'ajuda de la llibreria Pillow s'han processat totes les imatges per invertir els colors. Després s'ha aplicat un filtre d'Opencv per engrossir el traç de l'escriptura dels símbols.

```
1   dest = "data/HOMUS_filtered/"
2   source = "data/HOMUS_4Fold/"
3
4   # Aplicar el filtre d'OpenCV amb distints dipus de tras
5   def dilate(image, size, shape):
```

Figura 3.2: Imatge d'una semicorxera del dataset HOMUS original.

```
6       if shape==1 :
7           kernel = cv2.getStructuringElement(cv2.MORPH_RECT, size)
8       elif shape==2:
9           kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, size)
10      elif shape==3:
11          kernel = cv2.getStructuringElement(cv2.MORPH_CROSS, size)
12      return cv2.dilate(image, kernel)
13
14  # Obtenir el llistat d'arxius
15  imgs = []
16  for i in range(1,5):
17      imgs.extend(glob.glob('./data/HOMUS_4Fold/F{}/*'.format(i)))
18
19  # Per a cada imagte dilata el tras i guardar en el directori desti en format PNG
20  for img in imgs:
21      mat = dilate(cv2.imread(img,0), (3,3), 2)
22      name = re.sub(r'\.pbm','',re.sub(r'\./data/HOMUS_4Fold/','',img))
23      cv2.imwrite(dest + name + '.png',mat,[9])
```

La primera funció del generador és la creació de la imatge base amb el pentagrama, on les línies ocupen el 15% del pentagrama i els espais en blanc el 85%. Posteriorment es crea la imatge amb fons blanc i es dibuixen les línies, afegint un espai en blanc dalt i baix del pentagrama. Ja que hi ha notes que poden sobreeixir.

```
1   def gen_stave(width, heigth):
2       blank_size = int(heigth*0.85)//4
3       line_size = int(heigth*0.15)//5
4
5       im = Image.new('L',(width,heigth + border*2), color=white)
6       draw = ImageDraw.Draw(im)
7       x = 0 + border
8       num_lines = 0
9       while x <= heigth + border*2 and num_lines < 5:
10          draw.line([(0,x),(width,x)], fill=black, width=line_size)
11          x += line_size + blank_size
12          num_lines+=1
13      return im
```

La següent funció genera una seqüència aleatòria de símbols per a un exemple. Donada la llista conté les rutes dels fitxers d'imatges d'HOMUS. El bucle extreu un nombre donat de fitxers aleatòriament, obre les imatges i les fica a la llista. A banda també es crea altra llista on van ficant-se l'etiqueta de cada imatge. Com a resultat torna la llista amb les imatges, la llista paral·lela amb les etiquetes i la longitud de la seqüència.

```
1   def gen_sequence(files, length, dic):
2       symbols = []
3       label = []
4       size = sep    # Accumulate the width of every image
```

```
5       for i in range(length):
6           ran = random.randint(0,len(files)-1)
7           label_i = parse_label(files[ran])
8           if label_i not in dic:
9               print("ERROR",files[ran])
10              return  None, [files[ran]], None
11          label_i = dic[label_i]
12          symbols.append(ImageOps.invert(Image.open(files[ran])))
13          label.append(label_i)
14          size += symbols[-1].size[0] + sep
15      return symbols, label, size
```

Altra funció dissenyada és l'encarregada de ficar les imatges dels símbols dins la imatge del pentagrama. Amb el llistat de les imatges dels símbols, s'apeguen a la imatge base usant com a màscara la mateixa imatge del símbol amb els colors invertits. Açò dona com a resultat que la funció de Pillow només copia els píxels del símbol i no el fons blanc. Evitant que el fons blanc esborre les línies del pentagrama.

```
1   def put_symbols(stave, symbols, offset=0):
2       box = [sep,int(stave.size[1]*0.05) + border]
3       for img in symbols:
4           stave.paste(img,
5                   box=(box[0], box[1] ),
6                   mask=ImageOps.invert(img))
7           box[0] += img.size[0] + sep
```

Altres funcions auxiliars que s'han utilitzat són les expressions regulars. En primer lloc una expressió regular que extreu la part del nom de fitxer que conté l'etiqueta del símbol. En segon lloc l'expressió regular que s'encarrega de filtrar les classes que no s'usaran (o bé per motius de ser símbols poc freqüents o per llevar símbols difícils).

```
1   def parse_label(filename):
2       return re.sub(r'\.png','',re.sub(r'\./data/HOMUS_filtered/F./W.*_.*_','',
            filename))
3
4   def class_filter(label):
5       regex = r'((3-8)|(9-8)|(12-8)|(Sixty-Four)|(Thirty-Two)|(Natural)|(Double-
            Sharp)|(Sharp)|(Flat)|(Dot)|(Barline))+'
6       return re.match(regex,label)
```

Finalment tenim el cos del generador semisintètic. Recorre el llistat d'imatges per eliminar les classes que no es necessiten i crear el diccionari.

```
1   for img in imgs:
2       label = parse_label(img)
3       if not class_filter(label):
4           files.append(img)
5           if label not in dictionary:
6               dictionary[label] = len(dictionary)
```

S'exporta el diccionari en format JSON.

```
1   with open(dest + 'dictionary.json', 'w') as f:
2       json.dump(dictionary, f, sort_keys=True, indent=4)
```

Per a cada exemple que es genere: s'obté una longitud aleatòria de símbols, es genera la seqüència, es genera el pentagrama, s'apeguen les imatges i es guarda la imatge final. Per acabar s'escriu en un fitxer totes les seqüències d'etiquetes de cada imatge generada.

```
for i in range(nb_examples):
    length = random.randint(1,8) #Number of characters in the sequence
    symbols, label, size = gen_sequence(files, length, dictionary)
    if symbols is None:
        print(label in files)
        sys.exit(-1)
    stave = gen_stave(size,80)
    put_symbols(stave, symbols)
    for l in label:
        labels += str(l) + ' '
    labels += '\n'
    stave.save(dest + '{}.png'.format(i+1))
with open(dest + 'labels.txt','w') as fp:
    fp.write(labels)
```

3.3 Datasets utilitzats

3.3.1 Dataset sintètic

El primer *dataset* utilitzat, tant en les proves preliminars com en el sistema base, ha sigut sintètic. Generat per ordinador de forma automàtica. Amb l'ús de la ferramenta LilyPond que proporciona un llenguatge per compondre música, s'ha escrit un *script* que genera automàticament exemples d'entrenament. Cada exemple està compost per una imatge amb els diferents símbols musicals tipogràfics sobre un pentagrama. A més cada exemple està numerat i la seua corresponent etiqueta es guarda en un arxiu de text con el número de línia correspon amb el número d'exemple.

Amb aquest mètode s'han generat 20.000 exemples que contenen seqüències aleatòries d'entre 1 i 17 símbols. Cada símbol està etiquetat amb una de les 217 classes possibles. On el nom de cada classe inclou el tipus de símbol i un afegit que representa tonalitat en cas de tractar-se d'una nota musical. Les seqüències al ser aleatòries no tenen cap tipus d'harmonia musical ni representen cap cançó. No obstant al ser generades amb lilypond si que compleixen les regles del temps. Cada exemple representa un compàs 4/4, per tant les notes i silencis continguts en un exemple sempre sumen el temps corresponent. És a dir, a cada compàs només cap una redona o un silenci de redona, o per exemple dues negres i una blanca. (13)

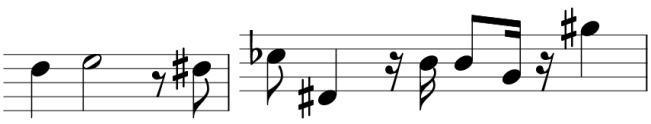


Figura 3.3: Exemples del dataset sintètic.

Les etiquetes de la figura 3.3, la primera correspon a l'exemple de l'esquerra i la segona al de la dreta:

```
QUARTER–D5 HALF–E5 EIGHTH_REST EIGHTH–D5f
EIGHTH–C5b QUARTER–D4f SIXTEENTH_REST SIXTEENTH–B4 EIGHTH–B4 SIXTEENTH–G4
    SIXTEENTH_REST QUARTER–G5f
```

3.3.2 Dataset semisintètic

Amb l'objectiu d'explorar un àmbit més complex, on els sistemes comercials tenen tasses de precisió molt baixes, s'ha elaborat un *dataset* que incorpora símbols manuscrits. I així poder provar com es comportaria un sistema on els símbols són molt més difícils d'identificar. La idea d'aquest ha sigut generar seqüències aleatòries d'imatges d'HOMUS [Fig. 2.11] i posar-les sobre un pentagrama generat automàticament. Estructurant de la mateixa forma que l'anterior, cada imatge numerada i un arxiu on el número de línia indica a quina imatge pertany cada seqüència d'etiquetes. En aquest cas els símbols es posicionen a diferents altures del pentagrama amb la única intenció d'afegir més dificultat al reconeixement. Com s'explicarà en la següent secció, no s'ha pogut identificar el cap de les notes musicals amb suficient precisió. Llavors a l'hora d'establir l'etiqueta dels símbols no és té en compte l'altura dins del pentagrama. Per acabar, s'ha generat un conjunt de 100.000 exemples amb les corresponents etiquetes, formades per un total de 18 classes distintes. Ja que ha hagut alguns símbols d'HOMUS que no han sigut incorporats (com les notes i silencis amb valors de temps d'1/32 i 1/64, o altres símbols poc freqüents), i no s'ha afegit a la classe la variable de la tonalitat.

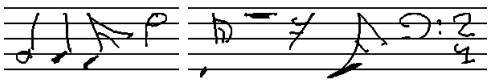


Figura 3.4: Exemples d'imatges del dataset semisintètic.

Les etiquetes de les imatges ja codificades per a usar-les amb la xarxa neuronal. Les etiquetes de la imatge de la dreta corresponen a la primera línia i la segona a la imatge de l'esquerra:

```
3 6 9 3
9 4 11 9 7 1

# Diccionari per codificar i descodificar les etiquetes
{
    "2-2-Time": 14,
    "2-4-Time": 1,
    "3-4-Time": 5,
    "4-4-Time": 0,
    "6-8-Time": 2,
    "C-Clef": 16,
    "Common-Time": 8,
    "Cut-Time": 15,
```

```
    "Eighth-Note": 10,
    "Eighth-Rest": 13,
    "F-Clef": 7,
    "G-Clef": 12,
    "Half-Note": 3,
    "Quarter-Note": 6,
    "Quarter-Rest": 18,
    "Sixteenth-Note": 9,
    "Sixteenth-Rest": 11,
    "Whole-Half-Rest": 4,
    "Whole-Note": 17
}
```

Aquest dataset serà usat com el dataset de referència a l'hora de realitzar les proves per optimitzar la xarxa neuronal. Ja que té major interés per al projecte desenvolupar un sistema que classifique seqüències de símbols que siguen manuscrits.

Detecció del cap de les notes musicals

Tindre en compte l'altura o tonalitat significa atribuir distintes classes a un mateix símbol de nota musical (blanca, negra, corxera etc.) segons a quina altura del pentagrama estiga situada. Aquesta opció augmenta la dificultat per al classificador. El nombre de classes augmenta considerablement: 19 classes per al *dataset* semisintètic sense l'altura i 217 classes per al *dataset* sintètic que si té l'altura. Implica un major nombre de neurones en la capa d'eixida, per tant molts més pesos a entrenar i major dificultat en el càlcul de l'error durant l'entrenament. A més cal remarcar que, en imatges d'entre 30 i 60 píxels d'altura, la diferència entre pertànyer a una classe o altra pot estar sols en uns 4 o 5 píxels de la ubicació del cap de la nota.

Malauradament el *dataset* HOMUS no incorpora la localització del cap de les notes musicals (la seua posició al pentagrama determina quina tonalitat representa) dins la imatge. Amb l'objectiu de poder incorporar la variable de l'altura de les notes musicals a les classes del *dataset* semisintètic (i així aconseguir major similitud amb el *dataset* sintètic), s'ha experimentat per buscar una forma senzilla de detectar el cap de la nota musical dels símbols d'HOMUS. Els mètodes provats per dur a terme aquesta detecció, junt amb els diferents resultats es troben a la secció 4.2

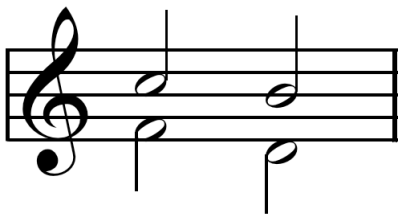


Figura 3.5: Exemple de diverses blanques a un pentagrama representant distintes tonalitats.

4 Experimentació

4.1 Xarxa neuronal

Les primeres proves per comprovar el funcionament de la xarxa neuronal s'han fet amb el *dataset* sintètic. S'ha utilitzat principalment com dataset de prova durant l'elaboració del codi per comprovar que les funcions creades funcionaven correctament. Després de l'elaboració del codi, durant les proves, es van trobar i corregir alguns errors en la càrrega de dades i en el mapeig de la part convolucional a la part recurrent de la xarxa. Després de les correccions es van realitzar alguns entrenaments amb aquest conjunt de dades i la primera arquitectura de xarxa descrita en la següent secció, que van donar bons resultats. Distàncies d'edició mitjana entre 1,7 i 2, que són bons resultats tenint en compte que les seqüències poden arribar fins a 15 símbols. Encara que no són comparables amb els resultats del següent *dataset*, ja que les mides són molt distintes (20.000 el sintètic i 100.000 el semisintètic). Sabem que són bons i que les imatges amb tipografies són més fàcils de classificar, ja que en aquest cas només s'han necessitat 20.000 exemples per a aprendre d'un conjunt que té un total de 217 classes possibles per a cada símbol. I per altra banda, s'han necessitat 100.000 exemples per a aprendre d'un conjunt que té tan sols 19 classes distintes per a cada símbol, i una longitud màxima de 8 símbols per seqüència.

Els resultats mostrats en aquesta secció fan referència a l'experimentació duta a terme amb el *dataset* semisintètic. Amb l'objectiu de maximitzar la precisió de la xarxa neuronal sobre aquest conjunt de dades, o de donar unes pinzellades sobre quins paràmetres poden millorar el rendiment i quins no.

4.1.1 Arquitectura de les capes

La primera arquitectura provada per a aquest sistema ha sigut la mateixa que trobem a l'exemple base(4). Aquest exemple només busca demostrar el funcionament de CTC i una xarxa neuronal per a fer OCR en Keras. Llavors l'arquitectura proposada és prou bàsica i suficient per a la demostració. Però per al problema que ens ocupa, prou limitada, com s'ha comprovat a les proves. La part convolucional està composta per dos blocs convolucionals. Ambdós amb una capa de convolució de 16 filtres de mida 3x3 i una capa de votació de mida 2x2. La part recurrent està formada per dues capes GRU bidireccionals amb 128 neurones cadascuna. Després de la construcció del sistema i comprovar que la xarxa neuronal era capaç d'aprendre amb les dades del *dataset* sintètic, es va procedir a fer un canvi de tota l'arquitectura. Ja que era necessària una amb major nombre de convolucions i filtres i un major nombre de neurones recurrents, a causa de la dificultat del problema.

Crear una nova arquitectura resulta una feina molt complexa i un coneixement de les xarxes neuronals molt profund. Llavors el que sol ser comú en aquest camp, és basar-se en arquitectures ja existents de problemes semblants i que han estat provades amb anterioritat per altres investigadors. I posteriorment anar ajustant algunes capes o paràmetres per al problema propi. L'arquitectura triada per al canvi, com a base, ha sigut la que es proposa a l'article Shi u. a. (11) i el seu funcionament s'ha explicat a la secció 2.1.2. L'estructuració de la part convolucional ha sigut de 4 blocs convolucionals:

1. Una capa de convolució amb 64 filtres de 3x3 i una capa de votació de 2x2.
2. Una capa de convolució amb 128 filtres de 3x3 i una capa de votació de 2x2.
3. Dues capes de convolució amb 256 filtres de 3x3 cadascuna i una capa de votació de 2x1.
4. Dues capes de convolució amb 512 filtres de 3x3 cadascuna i una capa de votació de 2x1.

L'estructura recurrent continuen sent dues capes GRU bidireccionals, aquesta vegada amb 256 neurones cadascuna. Encara que la xarxa de l'article fa ús de LSTM com a capes recurrents, s'ha triat GRU a causa d'un millor rendiment. El millor valor de distància d'edició normalitzada amb LSTM ha sigut de 0,072, en canvi amb GRU s'ha obtingut 0,024. Una millora considerable si tenim en compte com evoluciona l'entrenament, i que a partir de valors menors a 0,1 obtenir una millora suposa un major nombre d'èpoques. Amb l'ús de GRU també s'aconsegueix una xarxa més òptima, ja que aquestes neurones són més simples, i el nombre de paràmetres de la xarxa es veu reduït de 8.300 milions a 7.600 milions.

4.1.2 Híperparametres

Mida de la part recurrent

S'han provat distintes mides de la xarxa recurrent, variant el nombre de capes i de neurones. Totes les combinacions possibles de 128, 256 i 512 neurones en una, dues o tres capes. Les combinacions amb una capa no són capaces d'aprendre, posen patent la necessitat d'una capa oculta, no sols una capa d'entrada. El nombre de neurones més òptim trobat per a dues capes ha sigut 256 neurones, amb els resultats mencionats a la secció 4.1.1. Les combinacions amb 3 capes si han tret resultats positius però no han millorat l'anterior. El millor resultat de 3 capes ha sigut amb 256 neurones arribant a valors entre 0,03 i 0,04 de distància d'edició normalitzada.

Mides de filtres

Seguint plantejaments similars a altres problemes (com el reconeixement d'objectes), s'han provat filtres convolucionals de diferents mides de manera escalada. S'han testejat per als quatre blocs respectius de convolució, en ordre des de la capa d'entrada cap a la d'eixida, les mides: 9,6,3,3 - 6,6,3,3 - 5,5,3,3 - 3,3,2,2. En cap d'aquestes configuracions

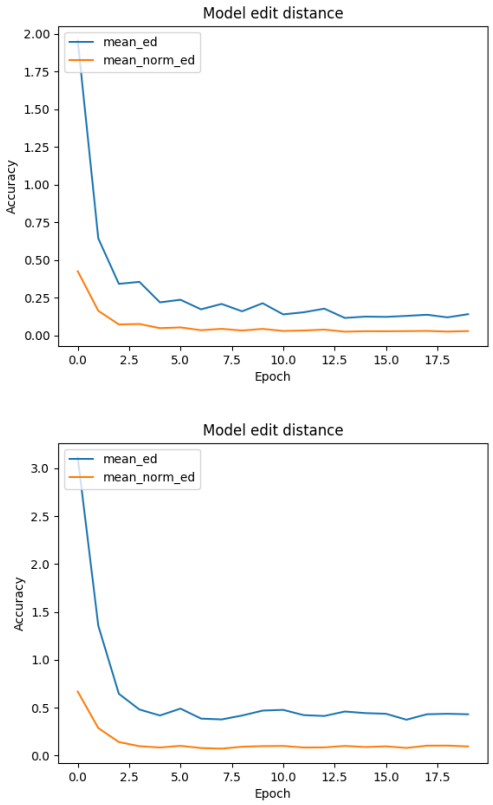


Figura 4.1: Evolució dels valors de distància d'edició mitjana i distància d'edició mitjana normalitzada del conjunt de test, per a cada època de l'entrenament. Amunt GRU, avall LSTM.

s'ha obtingut un model capaç d'aprendre per baixar d'una distància d'edició mitjana de 2. Resultats molt pobres, tenint en compte que amb filtres de 3x3 s'ha arribat fins a unes distàncies d'edició mitjana menors a 0,2.

4.1.3 Funcions d'activació

La funció d'activació utilitzada a totes les capes, després del canvi d'arquitectura, ha sigut tanh. Posteriorment s'han fet proves amb ReLU per tal de buscar millores. L'ús de ReLU en les capes convolucionals sí que ha aportat millores en la velocitat d'aprenentatge. En canvi si, s'usa en les recurrents, la xarxa neuronal tan sols divergeix. Açò és degut al fet que aquesta activació va sorgir, entre altres coses, perquè aportava una major resistència al gradient evanescent(16). Les capes LSTM ja estan dissenyades per evitar açò, llavors l'ús de ReLU no és necessari.

També s'han fet proves amb la funció sigmoide, i s'ha pogut comprovar que l'entrenament és molt més lent i s'aconsegueixen resultats molt pobres en aquest problema.

4.1.4 Optimitzador

Les funcions d'optimització són les diferents implementacions de l'algorisme de descens per gradient. Amb la diferència de què alguns aporten maneres de calcular-ho més eficients, o que van canviant els paràmetres de l'algorisme automàticament durant l'entrenament. Els que s'han provat han sigut quasi tots els disponibles en Keras, concretament els que ajusten el rang d'aprenentatge automàticament en cada moment de l'entrenament, que és el recomanable quan fem servir xarxes recurrents. També s'ha provat el descens per gradient estocàstic amb els paràmetres usats en l'exemple d'OCR en Keras. Cap d'ells ha sigut competitiu amb RMSProp, ja que alguns no eren capaços de fer aprendre a la xarxa o tardaven moltes més èpoques a aprendre. Per això s'ha utilitzat aquest en la resta de proves del sistema.

4.1.5 Exemple d'un entrenament

En cada execució es mostra primerament els paràmetres de l'entrenament i la configuració de la xarxa:

```
Number of classes: 19
Max length label: 8
90000 training examples
10000 test examples
30 epochs
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, 32, 120, 1)        0
_________________________________________________________________
conv1_1 (Conv2D)             (None, 32, 120, 64)       640
_________________________________________________________________
max1 (MaxPooling2D)          (None, 16, 60, 64)        0
_________________________________________________________________
conv2_1 (Conv2D)             (None, 16, 60, 128)       73856
_________________________________________________________________
max2 (MaxPooling2D)          (None, 8, 30, 128)        0
_________________________________________________________________
conv3_1 (Conv2D)             (None, 8, 30, 256)        295168
_________________________________________________________________
conv3_2 (Conv2D)             (None, 8, 30, 256)        590080
_________________________________________________________________
max3 (MaxPooling2D)          (None, 4, 30, 256)        0
_________________________________________________________________
conv4_1 (Conv2D)             (None, 4, 30, 512)        1180160
_________________________________________________________________
conv4_2 (Conv2D)             (None, 4, 30, 512)        2359808
_________________________________________________________________
max4 (MaxPooling2D)          (None, 2, 30, 512)        0
_________________________________________________________________
permute_1 (Permute)          (None, 512, 2, 30)        0
_________________________________________________________________
reshape (Reshape)            (None, 30, 1024)          0
_________________________________________________________________
bidirectional_1 (Bidirection (None, 30, 512)           1967616
_________________________________________________________________
bidirectional_2 (Bidirection (None, 30, 512)           1181184
_________________________________________________________________
```

```
dense2 (Dense)               (None, 30, 20)            10260
_________________________________________________________________
softmax (Activation)         (None, 30, 20)            0
=================================================================
Total params: 7,658,772
Trainable params: 7,658,772
Non-trainable params: 0
```

Posteriorment es mostra per a cada època el valor d'error que comet la xarxa en els conjunt d'entrenament i al final de cada època les mètriques de distància d'edició en el conjunt de dades de test.

```
Super epoch 1/30
    -Trainning from 0 to 10000
      Loss: 24.472
    -Trainning from 10000 to 20000
      Loss: 14.434
.......
    -Trainning from 80000 to 90000
      Loss: 6.665
    ——— MED: 2.249, MED_norm: 0.464 ———
.......
Super epoch 10/30
    -Trainning from 0 to 10000
      Loss: 0.206
.......
    -Trainning from 80000 to 90000
      Loss: 0.179
——— MED: 0.138, MED_norm: 0.030 ———
```

Altrament el *callback* guarda per a cada època un fitxer `.log` amb els resultats de tots els exemples de test. En primer lloc l'etiqueta que té assignada l'exemple, en segon lloc l'eixida que dóna la xarxa tenint com a entrada l'exemple, en tercer lloc la seqüència que seria la hipòtesi i finalment la distància d'edició i la distància normalitzada respecte a la longitud de la cadena. En aquest cas la classe 19 seria el blanc, que s'elimina per a establir la hipòtesi. En l'etiqueta assignada a l'exemple els -1 són els valors afegits per completar fins a la longitud màxima de seqüència, no es tenen en compte com a símbols. També es poden observar que les eixides de la xarxa són els 30 *frames* en els que s'ha dividit la imatge, i quina etiqueta veu en cada instant. Ací podem veure dos exemples de l'època número deu de l'entrenament mostrat anteriorment, on la precisió de la xarxa ja és prou alta.

```
Test:        [18, 14, 6, 1, 7, 3, 7, -1]
RNN output: [19, 19, 18, 19, 19, 19, 19, 14, 19, 19, 19, 19, 6, 19, 19, 19, 1,
     19, 19, 19, 19, 19, 19, 19, 7, 7, 19, 3, 7]
Hypothesis: [18, 14, 6, 1, 7, 3, 7]
    ED: 0 | ED_norm: 0.0

Test:        [14, 5, 5, 3, 10, 3, 13, 16]
RNN output: [19, 19, 14, 19, 19, 19, 19, 19, 5, 19, 19, 19, 19, 5, 19, 19, 19, 10,
     10, 19, 19, 19, 19, 19, 19, 19, 3, 19, 13, 16]
Hypothesis: [14, 5, 5, 10, 3, 13, 16]
    ED: 1 | ED_norm: 0.125

Test:        [14, 3, 9, 18, -1, -1, -1, -1]
```

```
RNN output: [19, 19, 14, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 3, 19, 19, 19,
    19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 9, 19, 18]
Hypothesis: [14, 3, 9, 18]
    ED: 0 | ED_norm: 0.0
```

4.2 Detecció del cap de les notes musicals

El cap de les notes musicals sol ser la zona de la imatge amb més concentració de píxels negres. El primer mètode utilitzat ha sigut obtindre l'histograma de les columnes i les files de la imatge, fent una suma dels valors dels píxels. Un mètode utilitzat freqüentment en OMR i bastant efectiu si parlem d'identificar notes tipogràfiques. Com que les imatges són en escala de grisos i no binaritzades (blanc o negre), s'ha triat sumar els valors dels pixels per no perdre tots els que tenen valors molt propers a negre. Com que en aquest cas el valor per al negre és 0 i per al blanc és 255, per saber el valor on més negres hi ha s'agafa el mínim.

La funció utilitzada per calcular el centroide del cercle que forma el cap de les notes:

```
def centroid(img):
    data = np.array(img.getdata(0)).reshape(img.size[0],img.size[1])
    cols = np.sum(data,axis=0)
    cols = outliers_filter(cols)
    rows = np.sum(data,axis=1)
    centroid = [np.argmin(cols),np.argmin(rows)]

    # Pinta el centroide
    img = img.convert('RGB')
    draw = ImageDraw.Draw(img)
    draw.point(centroid, fill=ImageColor.getrgb('red'))
    img.show()

    # Dibuixa la grafica
    plt.plot(cols)
    plt.plot(rows)
    plt.ylabel("sum")
    plt.xlabel("pixels")
    plt.legend(['cols','rows'])
    plt.show()
```

Amb les proves realitzades es van trobar un gran nombre de símbols amb els quals funciona aquesta tècnica. Però també alguns altres casos on la detecció fallava (imatge de la dreta en la figura 4.2). El principal problema sol ser amb les corxeres, que tenen cua, quan el cap dibuixat és molt petit i no suposa un gran volum de píxels en la imatge com en altres situacions. Per pal·liar el problema de les cues o algunes deformacions que causaven una gràfica molt irregular es va crear una funció de filtratge. Aquesta funció tracta d'eliminar els valors que despunten i que no pertanyen a la zona del cap. Els resultats milloren amb aquesta funció, però en molts casos no és possible eliminar els valors fora de lloc i diferenciar-los amb la zona del cap. El que condueix algunes voltes a situar el centroide molt lluny, ja que s'obtenen valors tan sols un poc majors i fa fallar el sistema.

```
def outliers_filter(array, m=1.75):
    std = np.std(array)
    mean = np.mean(array)
    for i in range(array.shape[0]):
        if not abs(array[i] - mean) < m * std:
            array[i] = mean
    return array
```

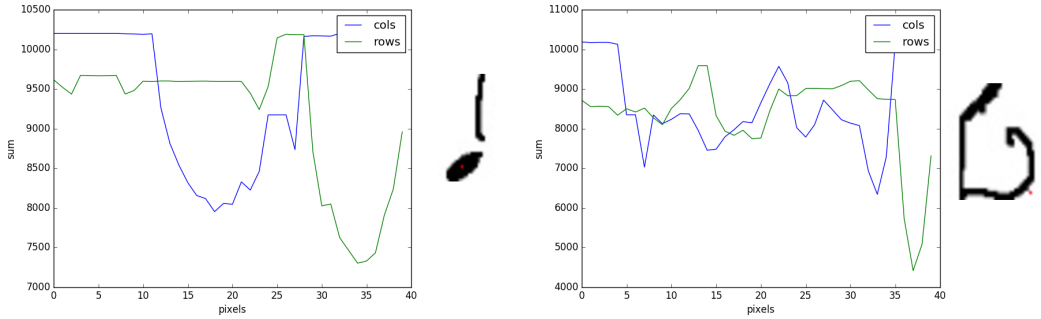


Figura 4.2: Resultats de detecció del cap de les notes. El punt roig és el centroide detectat.

El segon mètode utilitza va ser un detector de taques circulars en imatges. La llibreria OpenCV té alguns detectors bàsics, un d'ells per detectar taques en imatges (9). S'han fet successives proves amb el detector i les imatges d'HOMUS, però en cap d'elles s'ha aconseguit detectar el cap. Ni tan sols amb les imatges més senzilles de detectar. Provant diversos paràmetres del detector i tractant de posar uns valors molt relaxats. El més segur és que la baixa resolució de les imatges impedisca el bon funcionament de l'algorisme en aquests casos.

```
import cv2
import numpy as np
import sys

# Read image
im = cv2.imread(sys.argv[1], cv2.IMREAD_GRAYSCALE)
im = cv2.resize(im, (0,0), fx=4, fy=4, interpolation=cv2.INTER_NEAREST)

# Setup SimpleBlobDetector parameters.
params = cv2.SimpleBlobDetector_Params()

# Change thresholds
params.minThreshold = 0
params.maxThreshold = 100

# Filter by Area.
params.filterByArea = True
params.minArea = 3

# Filter by Circularity
params.filterByCircularity = True
```

```
22  params.minCircularity = 0.1
23
24  # Filter by Convexity
25  params.filterByConvexity = True
26  params.minConvexity = 0.1
27
28  # Filter by Inertia
29  params.filterByInertia = True
30  params.minInertiaRatio = 0.01
31
32  # Create a detector with the parameters
33  detector = cv2.SimpleBlobDetector(params)
34
35  # Detect blobs.
36  keypoints = detector.detect(im)
37  print(len(keypoints))
38
39  # Draw detected blobs as red circles.
40  # cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures
41  # the size of the circle corresponds to the size of blob
42  im_with_keypoints = cv2.drawKeypoints(im, keypoints, np.array([]), (0,0,255), cv2.
        DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
43
44  # Show blobs
45  cv2.imshow("Keypoints", im_with_keypoints)
```

5 Conclusions

La principal conclusió extreta de la realització d'aquest treball ha sigut que s'han obtingut grans avanços en el camp de l'OMR introduint aquestes noves metodologies. I que no obstant això, queda molta feina al davant per a aconseguir sistemes que siguen usables en el món real per músics i que quasi no requerisquen intervenció humana en el procés de digitalització. A més ha quedat patent que l'existència d'aquests sistemes pot ser real dins d'un temps. Per això és necessari que s'invertisca més temps i diners en el desenvolupament d'aquest camp. Ja que resulta molt important el desenvolupament d'aquest camp pel bé de la distribució, conservació i estudi tant de la música actual com la música antiga.

L'avanç aconseguit amb l'aplicació d'aprenentatge profund al reconeixement automàtic de notació musical s'ha pogut visualitzar amb aquest treball. També després d'haver estudiat amb deteniment diverses seccions de l'aprenentatge profund, s'ha pogut observar que queden moltes combinacions per explorar. I noves arquitectures més complexes i més completes per provar. Tampoc cal obviar l'aspecte més important per a que aquestes tècniques funcionen: les dades. Són molt necessaris més *dataset*s, amb un major nombre d'exemples (ja hem pogut veure com el nombre d'exemples és molt important), amb major quantitat d'informació. I el més important, conjunts de dades que siguen el més representatius possible del món real, o que provinguen de partitures reals manuscrites.

Altra demostració de l'avanç d'aquestes tècniques és la possibilitat d'haver realitzat aquest projecte com a Treball de Fi de Grau. Ja que l'alt grau de complexitat que suposa comprendre les xarxes neuronals profundes i el seu entrenament, donaria a entendre anys enrere, que un treball d'aquest tipus sols seria realitzable amb el temps que dura una tesi doctoral o algun treball de major nivell acadèmic. Principalment les ferramentes que han permés açò han sigut, el llenguatge, l'API i la funció d'error utilitzades. Amb un llenguatge com Python i la llibreria Keras el prototipat de models de xarxa i la manipulació de les dades ha sigut tremendament àgil i ràpid, en comparació d'altres ferramentes. I el complement de les xarxes neuronals que ha permés entrenar les xarxes neuronals de principi a fi, *Connectionist Temporal Classification*.

Per acabar, les idees que s'han extret per a treballs futurs que amplien aquest són diverses. Per una banda es podrien realitzar proves amb diversos *dataset*s semisintètics que representen major informació. Conjunts de dades que complisquen el criteri del tempo com és el cas del *dataset* sintètic (el que podria ajudar a comprovar si les xarxes neuronals són capaces d'aprendre la informació del temps dels símbols sense estar explicitada), o que tinguen en compte la tonalitat de les notes musicals. O també l'ús i creació de conjunts de dades que facen servir imatges de partitures manuscrites, fotografiades o escanejades, que siguen reals. Açò últim podria servir per a crear sistemes vertaderament robustos de cara a l'ús diari, o observar com es comporten en fer servir

dades amb seqüències de símbols que tinguen un sentit musical.

Bibliografia

[1] Calvo-Zaragoza, Jorge ; Oncina, Jose: Recognition of pen-based music notation: the HOMUS dataset. In: *Pattern Recognition (ICPR), 2014 22nd International Conference on* IEEE (Veranst.), 2014, S. 3038–3043

[2] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – `http://www.deeplearningbook.org`

[3] Graves, Alex ; Fernández, Santiago ; Gomez, Faustino: Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In: *In Proceedings of the International Conference on Machine Learning, ICML 2006*, 2006, S. 369–376

[4] Henry, Mike: *Example of an image OCR in Keras*. – URL `https://github.com/fchollet/keras/blob/master/examples/image_ocr.py`

[5] Hetland, Magnus L.: *Levenshtein Distance in Python*. – URL `http://hetland.org/coding/python/levenshtein.py`

[6] Hochreiter, Sepp ; Schmidhuber, Jürgen: Long Short-Term Memory. In: *Neural Comput.* 9 (1997), November, Nr. 8, S. 1735–1780. – URL `http://dx.doi.org/10.1162/neco.1997.9.8.1735`. – ISSN 0899-7667

[7] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F. (Hrsg.) ; Burges, C. J. C. (Hrsg.) ; Bottou, L. (Hrsg.) ; Weinberger, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, S. 1097–1105. – URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`

[8] LeCunn, Yann: *LeNet-5, convolutional neural networks*. – URL `http://yann.lecun.com/exdb/lenet/`

[9] Mallick, Satya: *Blob Detection Using OpenCV (Python, C++)*. – URL `https://www.learnopencv.com/blob-detection-using-opencv-python-c/`

[10] Ng, Andrew: *What data sciencists should know about deep learning*. – URL `https://www.slideshare.net/ExtractConf`

[11] Shi, Baoguang ; Bai, Xiang ; Yao, Cong: An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition. In: *CoRR* abs/1507.05717 (2015). – URL `http://arxiv.org/abs/1507.05717`

[12] Wikipedia: *ImageNet Challenge*. – URL `https://en.wikipedia.org/wiki/ImageNet#ImageNet_Challenge`

[13] Wikipedia: *Note value*. – URL `https://en.wikipedia.org/wiki/Note_value#List`

[14] Wikipedia: *Optical Character Recognition*. – URL `https://en.wikipedia.org/wiki/Optical_character_recognition`

[15] Wikipedia: *Optical Music Recognition*. – URL `https://en.wikipedia.org/wiki/Optical_music_recognition`

[16] Wikipedia: *Vanishing gradient problem*. – URL `https://en.wikipedia.org/wiki/Vanishing_gradient_problem`

[17] Zaragoza-Bernabeu, Jaume: *Image OMR*. – URL `https://github.com/ZJaume/image_omr`

Glossari

callback Objecte que rep per paràmetre l'API Keras per ser executat en moments concrets de l'entrenament. 3, 24–26, 37

dataset Conjunt de dades, estructurats en forma de parella (valor, etiqueta) o (entrada, eixida). vii, 2–5, 15, 19–21, 26, 29–31, 33, 41

ground truth Valor de l'etiqueta d'un exemple en el món real. 25

Acronyms

CTC *Connectionist Temporal Classification*. 10, 11, 23, 24, 33, 41

GPU *Graphics Processing Unit*. 6

GRU *Gated Recurrent Unit*. 9, 24

LSTM *Long Short Term Memory*. 9

OCR *Optical Caracter Recognition*. 1, 11, 13, 25, 33

OMR *Optical Music Recognition*. 1, 13, 15, 38, 41