Universitat de Girona
Escola Politècnica Superior

Treball Final de Màster

Estudi: Màster en Ciència de Dades

Títol: Modelització de les corbes d'aparició de glucosa en sang d'àpats reals per al seu ús en aplicacions biomèdiques

Document: Memòria

Alumne: Josep Noguer i Torres

Tutor: Dr. Iván Contreras Fernández-Dávila

Departament: Enginyeria elèctrica, electrònica i automàtica

Àrea: Enginyeria de sistemes i automàtica

Convocatòria: Setembre 2023

Universitat de Girona
Escola Politècnica Superior

TREBALL FINAL DE MÀSTER

Modelització de les corbes d'aparició de glucosa en sang d'àpats reals per al seu ús en aplicacions biomèdiques

Josep Noguer i Torres

TUTOR: Dr. Iván Contreras Fernández-Dávila

Màster en Ciència de Dades

UNIVERSITAT DE GIRONA

Setembre 2023

Agraeixo aquest treball als membres de MICELab per inculcar-me la passió per la recerca

Continguts

Figures

Taules

Acrònims

IoT *Internet de les coses*

T1DM diabetis mellitus de tipus 1

RA ràtio d'aparició de glucosa exògena

GAN xarxa generativa adversària

C-GAN xarxa generativa adversària condicionada

CHO hidrats de carboni

IOB *insulin on board*

COB *carbohidrates on board*

kcal quilocalories

F *Fast*

NF *Non-Fast*

AUC *àrea sota la corba*

LeakyReLU leaky rectified linear unit

FDA Food and Drug Administration

MDI injeccions múltiples diàries

IDF International Diabetes Federation

CGM *continuous glucose monitor*

ReLU unitat lineal rectificada

JS Jensen–Shannon

IDE Integrated Development Environment

VAE *variational autoencoder*

Resum

L'experimentació és essencial per tal que el coneixement sigui precís, confiable i perquè aquest pugui servir de suport a l'hora de prendre decisions en una àmplia varietat de camps, provar teories, determinar l'eficàcia de tractaments o la seguretat de nous dissenys. De tota manera l'experimentació és un procés complex i requereix dissenys precisos per a reduir problemes de cost econòmic, ètica, reproductibilitat i biaix. Amb aquesta finalitat les dades sintètiques permeten representar entorns reals de manera acurada, que poden ser utilitzats per a realitzar proves i investigació. Els progressos en computació i en les diverses tècniques de generació de dades han suposat un creixement en el potencial de les dades sintètiques, ja que cada cop és possible treballar amb conjunts més grans i personalitzats que els que poden obtenir-se únicament amb dades reals. Es poden construir escenaris que d'altra manera no podrien ser simulats. Un dels camps que més es beneficia d'aquest tipus de dades és la medicina, on les dades sintètiques són una eina per a poder dur a terme proves i assaigs de noves tecnologies, protegint alhora la privacitat i la seguretat del pacient.

L'objecte d'aquest treball és el disseny i implementació d'un sistema basat en tècniques d'aprenentatge automàtic generatiu, per tal de modelar de manera acurada les corbes de la ràtio d'aparició de glucosa exògena (RA) en sang. A partir d'un set de dades d'àpats reals es dissenya un model per a replicar las corbes que genera aquesta RA les hores postprandials. Les dades subministrades pel laboratori de recerca MICELab de la Universitat de Girona contenen l'especificació detallada dels diferents ingredients, informació nutricional precisa i la RA. Per tant, es busca obtenir un sistema basat en dades, capaç de millorar l'aproximació dels models metabòlics actuals.

La metodologia proposada es basa en arquitectures d'aprenentatge profund (xarxa generativa adversària condicionada (C-GAN), *autoencoders*, *transformers*, entre d'altres) per a millorar les aproximacions actuals a partir de l'aprenentatge de les distribucions de dades reals, condicionades als paràmetres que defineixen cada àpat (ingredients, valor nutricional, etc.). Els resultats son contrastats a través de la validació estadística pertinent i comparant corbes reals amb corbes generades a partir dades no utilitzades en l'entrenament del model. Finalment, es fa servir el model construït per a definir les corbes d'aparició de glucosa en sang d'un conjunt de dades sintètic a partir de valors simulats de les diferents entrades que condicionen al model.

1 Introducció

1.1 Antecedents

Per tal de garantir un coneixement verídic, actualitzat i ajustat a la realitat és innegable la necessitat de dades igualment realistes i de qualitat que permetin una experimentació fiable, ja que sense aquestes el progrés en tots els camps de la ciència no seria possible. En la última dècada i sobretot en els anys més recents, amb l'aparició de tecnologies *Big Data* i *Internet de les coses* (IoT) entre d'altres, s'ha aconseguit un accés pràcticament il·limitat i a temps real a la informació, permetent la realització d'experiments i el desenvolupament de noves eines a un ritme molt accelerat. L'enorme volum de dades al que donen accés aquestes tecnologies és crucial per a l'aplicació de models d'aprenentatge automàtic que tenen la capacitat de resoldre problemes complexos amb resultats i rendiment en alguns casos superiors als éssers humans. Això ha afavorit en gran mesura el desenvolupament en l'àmbit industrial i de l'enginyeria (1). En els camps de la medicina i les ciències biomèdiques l'adopció d'aquestes tecnologies es fa més lentament, ja que l'adquisició d'aquestes dades requereix de l'aprovació de comitès d'ètica, nombrosos tràmits burocràtics i per descomptat de la participació de pacients reals amb el cost humà i econòmic que això comporta. És per tant primordial disposar de mètodes alternatius per a obtenir dades clíniques realistes i utilitzables. Aquest problema es veu reflectit àmpliament en el cas que s'exposa en aquest projecte, on es planteja la generació de corbes de la ràtio d'aparició de glucosa exògena (RA) de glucosa en plasma per a usar-les en el desenvolupament de noves tecnologies de suport als pacients i per a l'experimentació en general. Aquestes corbes, que estan condicionades al metabolisme dels humans i a un seguit de variables externes molt diverses, actualment es modelen amb l'us de models matemàtics deterministes que exclouen l'estocàstica inherent a aquest tipus de variables fisiològiques.

1.2 Objectius i motivació

Aquest projecte neix de la necessitat de suplir les carències de les estimacions matemàtiques que es fan actualment de les corbes de la RA que malgrat ser estadísticament vàlides, aquestes no són prou realistes i els manca utilitat en casos on és necessària la dimensió estocàstica pròpia dels processos metabòlics, com són aplicacions de testeig de models, sistemes de control i experimentacions entre d'altres. L'àmbit d'ús d'aquest tipus de dades abraça un ampli ventall en camps sobretot

relacionats amb la nutrició, essent enfocades en el cas d'aquest projecte en pacients amb diabetis mellitus de tipus 1 (T1DM). Per tant, aquest treball es basa en l'us de dades obtingudes en el context de la T1DM que han estat mesurades en condicions de laboratori.

L'objectiu genèric i principal contribució d'aquest treball és el desenvolupament d'una arquitectura d'aprenentatge automàtic generatiu capaç de modelar corbes de la RA condicionades a les variables que tenim disponibles, de manera que la metodologia sigui capaç d'assolir nivells realistes en un conjunt de mètriques que puguin definir amb precisió les característiques de les series obtingudes. No només això sinó que també cal que les mostres generades i les variables d'entrada mantinguin la relació adient

Els objectius específics del projecte, enumerats, són els següents:

- Dissenyar un sistema capaç de generar corbes realistes de la RA basant-se en les mostres d'una llibreria d'àpats reals mesurats en pacients amb T1DM. Aquestes mostres generades no s'han de poder diferenciar de les mostres originals.

- Implementar una xarxa generativa adversària (GAN) capaç de treballar amb corbes de la RA, que pugui replicar la temporalitat d'aquest tipus de dades. Cal estudiar diferents metodologies i arquitectures, ja que en aprenentatge automàtic la majoria de tipus de neurona presenten dificultats al treballar amb series temporals.

- Implementar una xarxa generativa adversària condicionada (C-GAN) capaç de generar corbes de la RA condicionades a diferents paràmetres com poden ser el tipus d'àpat, la seva composició o els ingredients que el conformen. No només cal que les series generades siguin indistingibles de les originals sinó que és necessària una relació entre les mostres sintètiques i les variables que les condicionen. Aquesta relació ha de ser la mateixa que presenten les mostres reals.

- Donar validesa tan empírica com estadística als resultats d'ambdós tipus de models generatius. Caldrà dissenyar una serie d'experiments i buscar les mètriques d'avaluació que correspongui per tal de comprovar que els primers tres objectius s'estan complint correctament.

1.3 Abast

El projecte compren la generació de corbes de la RA amb models d'aprenentatge automàtic en base a un set de 54 àpats mesurats en condicions de laboratori dels quals coneixem els valors

nutricionals de fibra, hidrats de carboni (CHO), grassa, energia i proteïna així com una variable categòrica que agrupa cada àpat en una classe referent a la velocitat d'assimilació del cos humà, i que està associada al contingut de carbohidrats de cada aliment. Coneixem també d'aquestes dades la composició referent als ingredients i les pròpies corbes de la RA que s'han obtingut de la ingesta. Amb aquestes característiques es defineix com a l'abast del projecte la generació de corbes realistes, mantenint la relació original entre les variables característiques de l'àpat original i les sèries corresponents. Això ha de ser possible per a qualsevol set de dades que compleixi les mateixes característiques amb les que es condiciona. En el cas que alguna d'aquestes no estigués disponible, s'hauria de poder obtenir a partir de la reproductibilitat de l'assaig clínic original o bé altres tècniques com poden ser la classificació amb mètodes supervisats de *clustering* per a les variables categòriques. També s'han de poder generar corbes de la RA completament sintètiques, és a dir, utilitzant un conjunt de variables d'entrada que sigui coherent però sense que correspongui a àpats reals.

Malgrat que l'ús de les dades generades està dirigit al disseny d'experiments i la validació d'algoritmes d'aprenentatge automàtic en el camp de la T1DM per sintonia amb la font original de les dades, també serà possible el seu ús per a aplicacions nutricionals com el disseny de dietes especialitzades, l'àmbit esportiu, entre d'altres. El model només produeix corbes de la RA i és agnòstic de l'individu particular que fa la ingesta de l'àpat.

La principal limitació d'aquest treball és la seva naturalesa experimental ja que com a tal no es desenvoluparà un model destinat a funcionar en un entorn de producció sinó que funciona a mode de prototip, és pretén demostrar la validesa de la metodologia emparada i de les corbes generades. També es vol destacar que en aquest projecte no es pretén desenvolupar una nova arquitectura de models d'aprenentatge automàtic sinó que es busca aplicar models ja existents a problemes on encara no s'han aplicat aquest tipus de solucions.

2 | Marc conceptual

2.1 Diabetis tipus 1

La principal motivació d'aquest projecte és la investigació en pacients amb diabetis mellitus de tipus 1 (T1DM) per a tal de mitigar els efectes de la seva malaltia i proporcionar-los eines més realistes i eficaces per al seu control. La T1DM és una patologia crònica que involucra la destrucció de cèl·lules β del pàncrees provocant un dèficit d'insulina (veure Figura 2.1 (2)). La insulina és una hormona necessària per a la regulació dels nivells d'absorció de glucosa en sang, mantenint el metabolisme a nivells euglicèmics (3) i afavorint el transport de la glucosa als òrgans i teixits on s'utilitza.

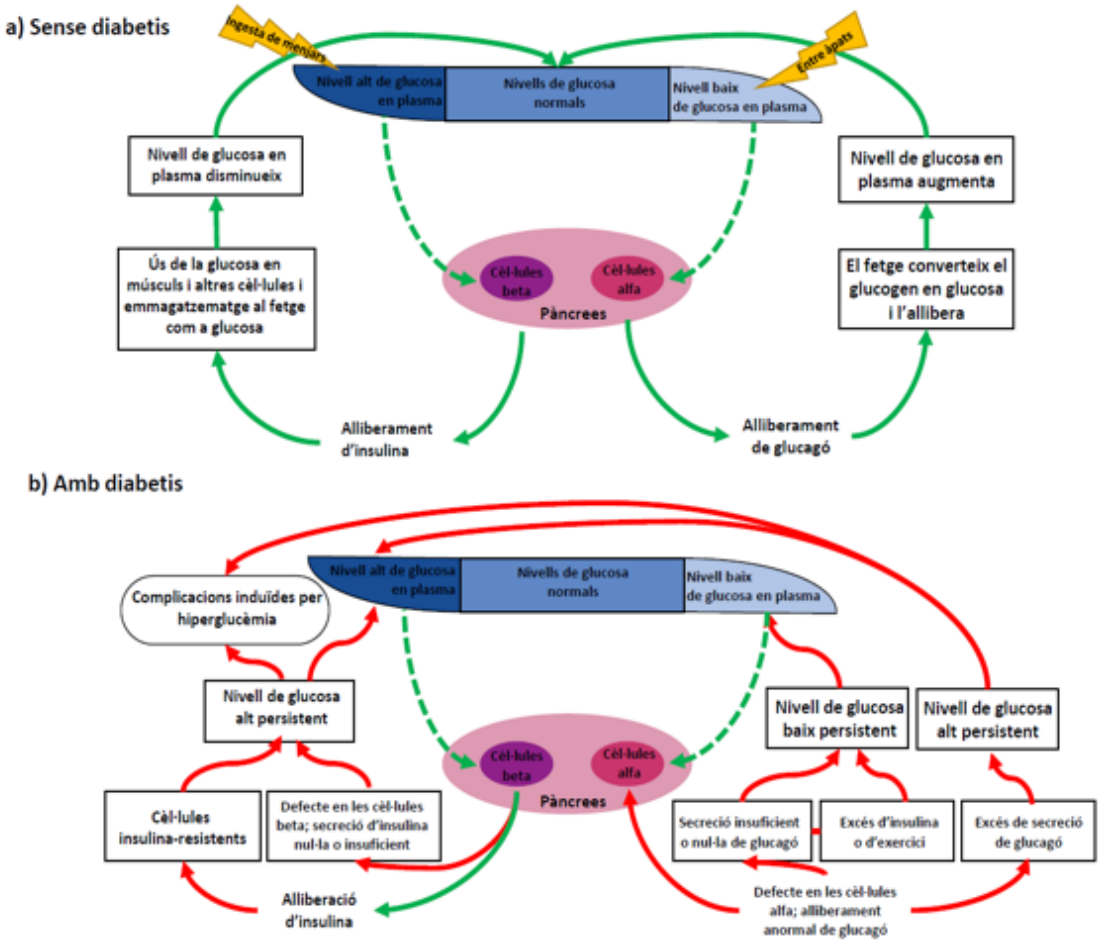


Figure 2.1: Mecanisme homeostàtic de la glucosa en condicions normal (a) i en condicions diabètiques (b).

Els pacients que pateixen aquesta patologia han de regular la glucosa en sang mitjançant les teràpies intensives d'insulina. Principalment les teràpies d'administració d'insulina són dues, la

basada en injeccions d'insulina i la basada en una bomba d'insulina. Les injeccions múltiples diàries (MDI) són fins ara la forma més estesa. Aquest tractament requereix l'administració diària d'insulina d'acció ràpida i de llarga durada. Les dosis d'insulina ràpida (bolus) permeten contrarestar pertorbacions brusques com els àpats i la insulina de llarga durada o basal permet compensar les necessitats fisiològiques d'insulina del pacient. La segona opció de tractament és mitjançant l'administració continuada d'insulina utilitzant bombes d'insulina (4) que injecten contínuament petites dosis d'insulina d'acció ràpida.

Els principals perills d'un descontrol del nivell glúcic es tradueixen en una manca (hipoglucèmia) o excés (hiperglucèmia) de glucosa en sang. Alguns dels perills de les hiperglucèmies són patir una cetoacidosis diabètica o un estat hiperosmolar hiperglicèmic, que pot evolucionar cap a la mort. Un pacient amb hipoglucèmia pot patir de deficiències neurològiques com somnolència, mareig o confusió i en casos més extrems coma i inclús la mort (3). Els efectes d'una hipoglucèmia varien segons la intensitat d'aquesta. Es defineix la hipoglucèmia de nivell 1 com un descens del nivell de glucosa en sang per sota de 70 mg/dl mentre que una hipoglucèmia de nivell 2 representa un descens per sota de 54 mg/dl (5).

Segons la International Diabetes Federation (IDF) al 2019 hi havia 463 milions de persones amb diabetis al món (6), dels quals aproximadament un 10% són diabètics tipus 1, el que converteix a la diabetis en una de les majors emergències sanitàries mundials de segle XXI. Per tant, s'ha esdevingut primordial que els pacients amb T1DM disposin de les eines adequades per al correcte control i desenvolupament de les seves activitats quotidianes. Exemples serien les ja esmentades bombes d'insulina, que regulen les dosis de manera automàtica, els *continuous glucose monitor* (CGM), que cada cop son més econòmics i discrets o els algoritmes d'intel·ligència artificial (7) que han suposat un gran increment del potencial que tenen les aplicacions en la diabetis. Aquests tres exemples suposen grans beneficis a la qualitat de vida dels pacients i poden funcionar de manera cohesionada, tenint totes un punt en comú: requereixen d'experimentació i dades per a funcionar de manera correcta i validada.

En concret hi ha 3 variables molt determinants que intervenen en els sistemes de control, gestió i predicció de la glucosa en sang i són:

- La glucosa en sang que té el pacient: Tradicionalment es mesura amb l'ús del glucòmetre tot i que els models actuals de CGM ja permeten mesures precises, constants i d'obtenció molt més còmode. Per als pacients la glucosa en sang és la variable que cal controlar i mantenir a nivells estables essent aquesta la que es monitoritza i es pren com a variable objectiu en la

majoria de models.

- La insulina subministrada: Aquesta variable és la que permet al pacient regular i absorbir els hidrats de carboni (CHO) que ha consumit en un àpat. Se sol treballar amb una mesura convertida de la insulina subministrada que és la *insulin on board* (IOB) i fa referència a la quantitat d'insulina de tipus ràpid que queda al cos. Tradicionalment aquest valor es calcula amb equacions de cinètica metabòlica.

- Els CHO ingerits en l'àpat: Són la font de glucosa necessària per a que el cos pugui fer les seves funcions i han de ser processats mitjançant la insulina. Aquests depenen del tipus d'aliment consumit i requereixen una atenció especial per part dels pacients, que han de mantenir una dieta estricta. De la mateixa manera que passa amb la insulina se sol mesurar de manera convertida, com pot ser el *carbohidrates on board* (COB) que és la quantitat de CHO que té disponible el cos i per tant és acumulativa entre àpats si el cos no ha tingut prou temps a absorbir els que s'han subministrat en l'àpat anterior. Una altra mesura convertida dels CHO és la ràtio d'aparició de glucosa exògena (RA) que indica com el cos processarà un àpat concret al llarg del temps. Aquests dos camps també es calculen actualment utilitzant equacions de cinètica metabòlica.

2.2 Model generatiu condicionat

Tot i que les aplicacions on els models generatius profunds destaquen estan sobretot relacionades amb la generació d'imatges, també han estat àmpliament utilitzats en els últims anys per a produir dades sintètiques en diversos àmbits, on la seva capacitat generativa pot ser aprofitada aportant en ampli un ventall de beneficis. L'objectiu d'un model generatiu profund és aproximar una distribució de probabilitat complexa i d'alta dimensionalitat mitjançant xarxes neuronals (8). Aquesta tasca comporta l'entrenament d'una xarxa neuronal generativa de manera que el model obtingut aprengui a transformar una distribució de probabilitat controlada i manejable Z en una distribució de probabilitat complexa X. La naturalesa de aquestes xarxes fan gairebé impossible identificar de manera única la distribució de probabilitat d'un conjunt finit de mostres, per tant, els models profunds generats depenen en gran mesura dels hiperparàmetres involucrats en el disseny del model. Es poden trobar diversos tipus de models generatius profunds a la literatura amb diferent característiques que els fan adequats per diferent escenaris. Els tipus més comuns de models profunds són les GAN, els *variational autoencoder* (VAE), la xarxes bayesianes i els fluxos normalitzadors.

Degut al petit volum de dades del que es disposa i malgrat els altres models disponibles són més potents, el model utilitzat en aquest projecte ha estat una xarxa generativa adversària condicionada (C-GAN). Això implica que a partir d'unes variables d'entrada determinades es poden obtenir corbes de la RA realistes que s'hi ajusten, essent crucial la decisió de quines s'utilitzen i les transformacions que se'ls apliquen prèviament a la ingesta d'aquestes a la xarxa. En un model d'aquest tipus les corbes resultats es corresponen a les variables d'entrada però, i això és el que els diferencia d'un model determinista, no per a unes mateixes variables s'obtenen sempre corbes iguals. Això és degut a que també s'alimenta el model amb soroll aleatori per tal d'emular la variabilitat pròpia de l'entorn i l'individu. Addicionalment l'estocàstica característica d'aquest tipus d'entrenament també afegeix soroll en el model resultant.

En els models tradicionals d'aprenentatge automàtic supervisat una xarxa disposa d'un set de dades i es pretén assolir una relació entre una mesura objectiu i les variables que la condicionen a partir d'una funció d'error que el model utilitza de manera iterativa per a comparar els resultats esperats i els que obté en una instància concreta. Posteriorment propaga aquest error per a modificar els pesos de les seves neurones i reduir-lo.

Pel contrari les xarxes del tipus GAN (9) es fonamenten en l'aprenentatge adversari que consisteix en dos models competidors que tractant d'optimitzar el seus objectius, un model discriminador i un model generador. La funció del discriminador és distingir dades reals de les dades produïdes pel generador, mentre que aquest continua millorant les dades que genera per a enganyar al discriminador. Els dos models són xarxes neuronals profundes entrenades per superar-se mútuament. El discriminador s'entrena amb dades (imatges, series temporals, etc.) reals i, per tant, pot classificar correctament entre dades reals i sintètiques. D'altra banda, el generador s'entrena utilitzant la retroalimentació del discriminador. Una vegada que el generador ha après prou bé la distribució de probabilitat del conjunt de dades reals, pot crear mostres infinites que siguin estadísticament similars a les dades reals i enganyar el discriminador. Es mostra quina és l'arquitectura del model GAN a la imatge 2.2.

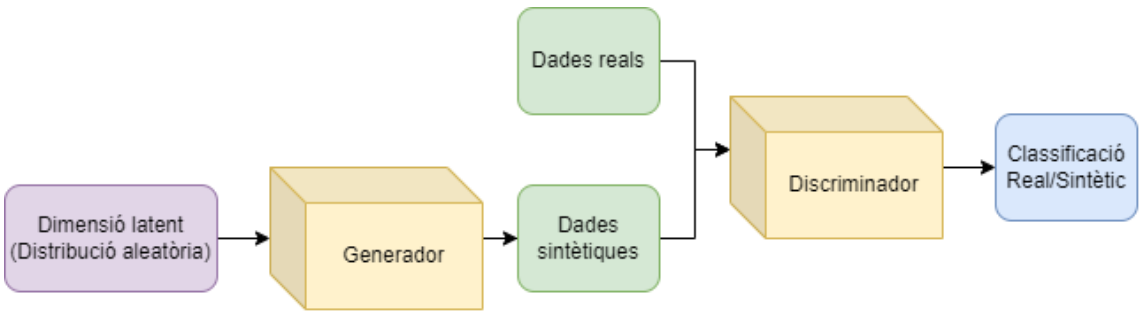


Figure 2.2: Arquitectura estàndard de la xarxa xarxa generativa adversària (GAN)

El model C-GAN (10) és una variant de les GAN. Conceptualment són també xarxes adversàries en les que dos models entrenen de manera oposada un per augmentar la funció d'error i l'altre per a disminuir-la. Es diferencien de les GAN senzilles ja que en aquestes s'introdueix un entrenament addicional al generador. Ara aquest té la tasca de generar distribucions realistes basant-se en una dimensió latent però també en una serie de variables d'entrada que mantenen relació amb la sortida. El discriminador també pateix modificacions, ja que ara no només entrena per a reduir l'error en la classificació entre mostres reals i sintètiques, sinó que ha d'aprendre si les mostres i variables d'entrada que rep es corresponen entre elles o no. En aquest cas, l'arquitectura estàndard és la que es mostra en la figura 2.3 i s'acompanya de l'exemple de la figura 2.4 on es veu més clar com és el funcionament d'aquestes xarxes. S'alimenta al generador d'una conjunt de variables, en aquest cas un dibuix lineal d'una bota (que pertany a una de les instàncies de dades reals) i aquest produeix una imatge foto-realista de la mateixa bota. Llavors s'alimenta el discriminador amb les imatges sintètica i real per tal que aquest les classifiqui segons la seva veracitat i segons si l'imatge correspon a la pròpia etiqueta d'entrada.

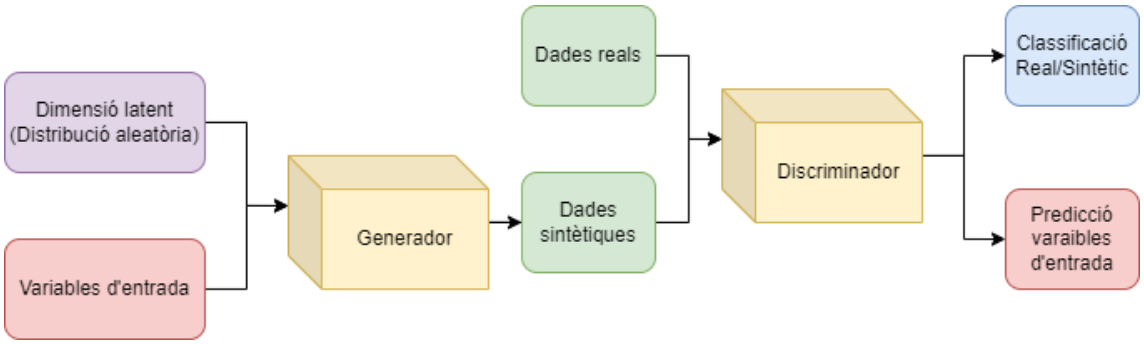


Figure 2.3: Arquitectura estàndard de la xarxa xarxa generativa adversària condicionada (C-GAN)

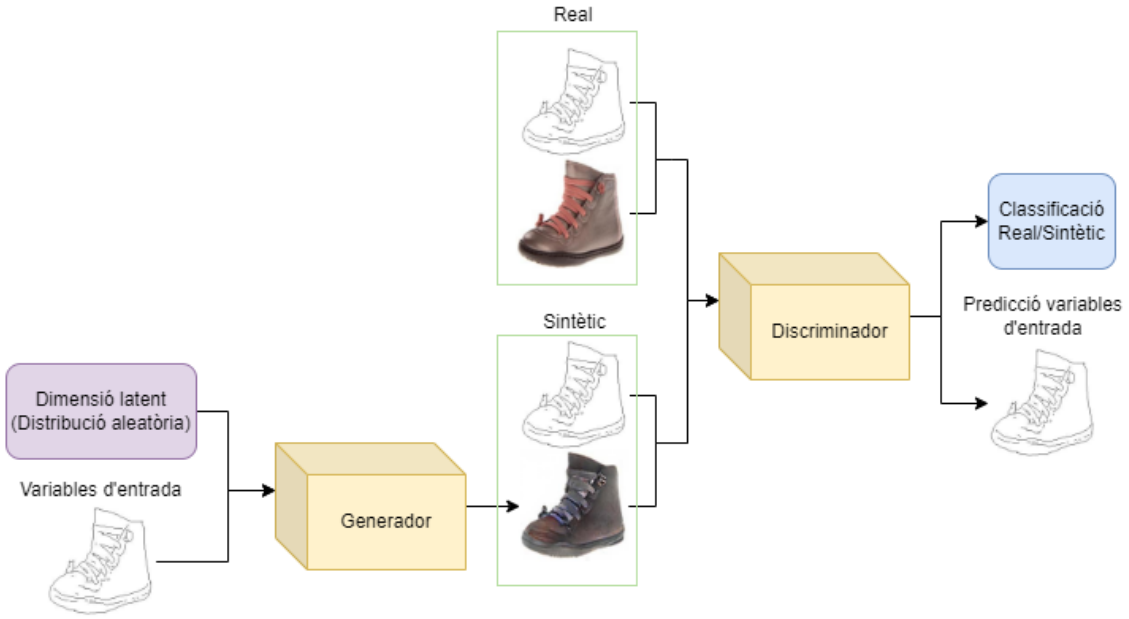


Figure 2.4: Exemple d'implementació d'una xarxa xarxa generativa adversària condicionada (C-GAN)

Tant el model GAN com el C-GAN presenta alguns problemes que li son propis, els més comuns són la no convergència, la desaparició de gradients i el col·lapse de modes, inconvenients que s'han de tenir en compte alhora de plantejar el disseny de la xarxa

- **No convergència:** Durant l'aprenentatge el generador i del discriminador no assoleixen un equilibri sinó que apareixen oscil·lacions aleatòries de resultats. Per evitar aquest problema es recomana afegir soroll a les entrades del discriminador, tant a les reals com les sintètiques o bé penalitzar-ne els pesos.

- **Desaparició de gradients:** El model del discriminador és superior al generador i per tant aquest segon no és capaç d'extreure informació de l'entrenament del primer, no podent produir mostres realistes. Aquest problema es combat amb l'ús de funcions d'error que permeten treballar en escales diferents a la sigmoide o la tangent hiperbòlica, que només permeten treballar entre 0 i 1 la primera i entre -1 i 1 la segona. També es recomana utilitzar funcions unitat lineal rectificada (ReLU) o leaky rectified linear unit (LeakyReLU) per tal de fer els valors propers negatius propers a 0 i l'ús d'optimitzadors diferents, com per exemple el *MinMax*.

- **Col·lapse de modes** En aquest cas el generador és més potent que el discriminador i per tant assoleix una configuració de pesos que genera una distribució que sempre enganya al discriminador. Això és similar al sobre-entrenament típic de les xarxes neuronals convencionals. Per atacar el problema es recomana utilitzar un optimitzador com pot ser el *Nudged-Adam*

(11) amb la intenció d'eliminar els valors propis més òptims, fent que el model aprengui més lentament però sense col·lapsar modes. Per evitar aquest fenomen també es pot utilitzar una tècnica d'estimació, coneguda en anglès com *Implicit maximum likelihood estimation* (12), en la que addicionalment a buscar un generador que només permeti produir mostres realistes, es fa que aquestes sempre siguin diferents entre elles. Amb això s'aconsegueix que tots els modes es vegin representats a la sortida del generador.

Tots aquest problemes es poden atacar de manera conjunta amb una versió modificada de la GAN que rep el nom de model de *Wasserstein* (13). Proposada al 2017, aquesta modifica el discriminador, rebent en aquest model el nom de crític, perquè ja no avalua si una mostra és o no realista mitjançant la funció sigmoide sinó que dona una sortida de nivell de veracitat. Per a disposar d'un model de *Wasserstein* s'han d'aplicar un seguit de canvis que són:

- Funció d'activació lineal a la capa de sortida del crític.
- Modificació de les classes en les mostres d'entrada per -1 a les generades i +1 a les reals.
- Funció d'error específica que és $W_{loss} = y_{cert} \cdot y_{predit}$
- Retallat de pesos del crític.
- Actualització desequilibrada del generador i el crític, essent aquest últim més entrenat.
- Ús de l'optimitzador *RMSprop*.
- Ús de ràtios d'aprenentatge molt petits.

2.3 Estat de l'art

2.3.1 Ràtio d'aparició de glucosa exògena

La principal aportació d'aquest treball es centra en obtenir una representació veraç i realista de les corbes de la RA, que són un element clau per al desenvolupament de simuladors de pacients virtuals i per dur a terme estudis que incloguin la validació i avaluació de tractament a la T1DM (pàncrees artificial, sistemes de suport a la presa de decisions, calculadors de bolus, etc.), però també suposant d'interès en altres malalties i camps com són l'esport professional o la nutrició en general. A la Figura 2.5 podem veure diversos exemples d'aquestes corbes que presenten l'evolució de la RA en un lapse de temps de 6 hores després d'haver ingerit un àpat específic. Malgrat si que existeixen dades reals de concentració de glucosa en plasma, no hi ha pràcticament

disponibilitat d'informació sobre la RA, essent els models matemàtics la millor estimació de la qual es disposa a dia d'avui. (14)

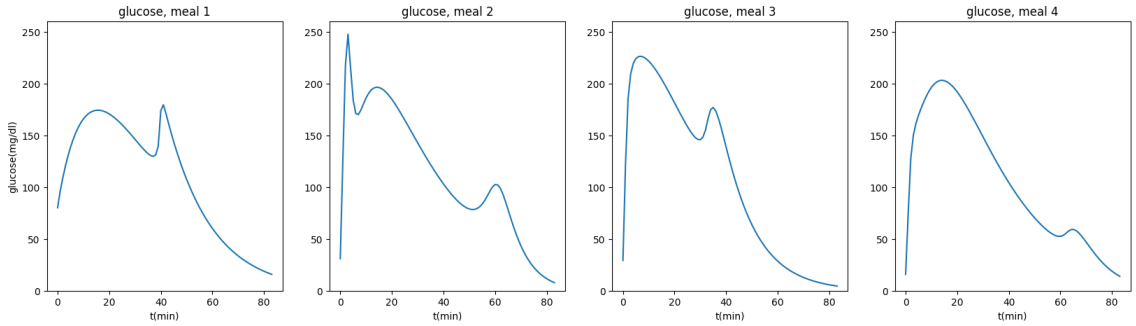


Figure 2.5: Exemples de corbes de la ràtio d'aparició de glucosa exògena (RA)

El procés d'ingesta de glucosa a la sang és oral. La absorció es realitza al tub digestiu superior i es transporta principalment al fetge, on es transmet al torrent sanguini. Es considera que el millor mètode per obtenir corbes de la RA son els anomenats *multitracers* (15) però aquesta metodologia requereix de condicions de laboratori i té un cost elevat, per tant no és viable per a utilitzar-la en aplicacions que els pacients puguin utilitzar en el dia a dia, sinó que se sol fer servir en assaigs clínics i per a validar altres tècniques. Actualment existeixen models que representen aquesta absorció amb l´us de funcions lineals. El model *Lehmann-Deutsch* (16) n'és un exemple i descriu l'absorció de glucosa al tracte intestinal segons les cinètiques lineals de primer ordre següents:

$$\begin{cases} \dot{q}_{\text{gut}}(t) = -k_{\text{abrs}} \cdot q_{\text{gut}}(t) + C_{\text{emuIt}}(t) \\ \text{Rat}(t) = f \cdot h_{\text{abs}} \cdot q_{\text{sul}}(t) \end{cases} \tag{2.1}$$

També els tres models que es presenten a (17), essent aquests més complexes que l'anterior:

- Funcions definides a trams, segmentant l'espai temporal en els punts 0, 10, 30, 60, 90, 120, 180 i 420 minuts

$$\text{R}_\text{a}(t) = \begin{cases} \kappa_{i-1} + \frac{\kappa_i - \kappa_{i-1}}{t_i - t_{i-1}} \cdot (t - t_{i-1}) & \text{per } t_{i-1} \leq t \leq t_i i = 1 \ldots 7 \\ 0 & \text{altrament} \end{cases} \tag{2.2}$$

  On κ_i és el set de paràmetres desconeguts que representen els valors de la RA en els canvis de segments

- Funcions Spline,

$$\text{Ra} = \sum_{i=1}^{n} B(i,k) \cdot a_i \tag{2.3}$$

- Funcions dinàmiques

$$\mathrm{Ra} = \sum_{i=1}^{3} C_i \cdot \left[\mathrm{rect}_i \otimes \left(k_i \cdot t \cdot e^{-k_i \cdot t} \right) \right] \tag{2.4}$$

  On $\mathrm{rect}_1 = 1$ en $[0,30]$, $\mathrm{rec}_2 = 1$ en $[30,120]$, i rect $3 = 1$ in $[120,210]$. La quantitat de paràmetres desconeguts a estimar és de 6 i.e. tres amplades $\{c_i\}$ i tres constants de ràtio $\{k_i\}$.

Totes aquestes propostes aporten solucions estadísticament competitives per al modelat de la RA. De tota manera el principal problema al que s'enfronten és la seva naturalesa determinista ja que per un mateix àpat sempre es rebrà la mateixa sortida i per tant no son capaços de representar l'estocàstica pròpia de tots els processos del metabolisme humà i de l'entorn (18). Això és més clar quan analitzem la naturalesa de l'absorció de la glucosa on factors tan variables com són l'ordre en el que ingerim els aliments o si han estat refrigerats entre la cocció i la ingesta (19) fan que la corba resultant presenti característiques completament diferents. Addicionalment es conegut que solament la meitat dels pacients són capaços de calcular correctament la quantitat de CHO que rebran d'un àpat, cometent un error mitjà superior al 30%, sovint identificant una menor quantitat de la que realment hi ha i inclús amb una educació per part d'especialistes aquest error es manté sobre el 20 % (20). Tots aquests factors fan que sigui extremadament complicat i perillós disposar de models que depenen exclusivament d'aquesta informació.

2.3.2 Generació de dades sintètiques

Ja ha estat demostrat que les xarxes generatives són una alternativa vàlida per a la síntesis de dades tant per a mostres reals no condicionades (21) com per a les que depenen de variables d'entrada (22). En aquest exemple una xarxa C-GAN rep com a dades d'entrada ingestes d'insulina i de CHO, aquestes es transformen a dades IOB i COB i és capaç de generar corbes de resposta de nivells de glucosa en plasma, imitant amb precisió la resposta de pacients reals.

Els avantatges de l'aplicació d'aquestes tècniques són notoris en el desenvolupament de nous models d'aprenentatge automàtic on, amb volums inicials de dades molt reduïts, permeten assolir nivells de precisió i exactitud equivalents i en ocasions superiors, als models de l'estat de l'art. Aquests models als que se sol anomenar simuladors virtuals se solen usar també per a la validació *in silico* sense necessitat d'intervenir en pacients reals i ometent tràmits burocràtics i reduint l'elevat el cost econòmic i humà. En particular a l'àrea de la T1DM aquests tipus de simuladors suposen un progrés respecte als models que trobem més establerts en l'estat de l'art i que es basen en funcions matemàtiques, que ja van suposar un avanç molt important al 2008 quan la Food and

Drug Administration (FDA) en va permetre l'ús en assaigs pre-clínics de diversos tractaments d'insulina (23) on un simulador era capaç d'estimar quina serà la resposta de la glucosa en sang en base a dades de CHO i bolus d'insulina, el seu principal problema és que degut a la seva naturalesa determinista, malgrat les corbes resultants siguin vàlides, no tenen la capacitat de representar un ventall ampli del metabolisme real.

El problema del determinisme en les mostres simulades se sol anomenar l'escletxa de la realitat (24) i complica enormement el desenvolupament d'algoritmes d'aprenentatge automàtic que siguin capaços de resoldre problemes reals. Per a evitar-lo i suplir la manca d'aquestes dades realistes s'han proposat nombrosos models i arquitectures generatives en camps com poden ser la dermatologia (25) (26), cardiologia (27), o l'endocrinologia (28) on estan suposant una millora molt rellevant en l'ampliació de sets de dades, modelat de pacients virtuals i fins i tot l'anonimització de conjunts de dades ja existents (29) (30)

3 | Materials i Mètodes

3.1 Dades i processament

Malgrat que el problema de la representació realista de les corbes de la ràtio d'aparició de glucosa exògena (RA) pot arribar a tenir un gran impacte per a un control acurat de malalties com la diabetis mellitus de tipus 1 (T1DM) i altres que en depenen directament, no hi ha pràcticament disponibilitat de sets de dades complerts i validats en estudis en laboratori. Per aquesta raó, la quantitat d'informació disponible és escassa i de difícil accés, dificultant l'anàlisi d'aquest problema i per tant també el desenvolupament de models d'aprenentatge basat en dades. Malgrat aquesta escassetat d'informació disponible, estudis com (15) generen mesures de corbes de RA de manera precisa i inclouen la quantitat de la hidrats de carboni (CHO) de cada àpat, però no sense considerar els paràmetres específics o nutrients dels menjars que són ingerits pels pacients (tipus de carbohidrats ingerits, quantitat de proteïnes, sucres, greixos, etc.) i per tant, obtenint un nombre de condicionants insuficients per poder caracteritzar els diferents àpats, que al seu torn, condicionen la generació de les corbes de la RA. D'altra banda, tenim estudis que se centren en les característiques dels menjars de forma molt precisa, detallant tots els tipus d'ingredients i nutrients, però sense incidir en les implicacions de l'aliment en aparició de glucosa, com és el cas de la base de dades de *5k-Nutrition* (31). En concret en aquest projecte s'utilitzen les dades d'un estudi realitzat al laboratori de recerca MICELab, que ha estat el set de dades més complet que al que hem pogut accedir per poder completar els objectius d'aquest treball. Aquesta llibreria d'àpats mixtes es correspon concretament a la publicació (32) on es va presentar una metodologia robusta i senzilla per a estimar les corbes de la RA i que no requereix de fer assumpcions de paràmetres. En aquest estudi es va dur a terme un assaig clínic amb el mètode *multitracer* i va servir també com a fonament per a la tesis doctoral (18) entre altres projectes de recerca. Corresponen a pacients amb T1DM i van ser recollides en condicions estables amb un control precís de la ingesta d'aliments i en condicions estandarditzades per a evitar introduir soroll de condicionants externs a les corbes resultants.

En aquesta llibreria es comprenen 54 àpats diferents mesurats amb detall en un entorn de laboratori, d'aquests tenim primerament la llista d'ingredients que els conformen i també dades de la quantitat de CHO, energia, mesurada en quilocalories (kcal), fibra, grassa, i proteïna, en grams. Totes aquestes variables són numèriques i en alguns dels àpats tenim valors buits, excepte en el cas

dels CHO. També tenim una variable categòrica que basat en la distància Kolmogorov-Smirnov classifica el àpats en:

- *Fast*: Aliments que alliberen molt ràpid el seu contingut de CHO, que a més és elevat, al torrent sanguini. Són aliments generalment amb baix contingut en proteïna i no disposem dades de la quantitat de fibra de cap d'aquests àpats, que són els que tenen un nombre més reduït, essent un total de 6. Un exemple d'àpat de tipus *fast* són els cereals de blat de moro amb llet.

- *Large*: Aquest grup el conformen àpats amb un contingut de CHO i energia molt elevat i la glucosa s'aporta al torrent sanguini en un període de temps ampli. Està conformat per àpats com pot ser un gelat amb nata i caramel.

- *Medium*: Els àpats que entren en aquesta classificació són, tal i com indica el seu nom, àpats equilibrats en quan a CHO, energia, fibra, etc. També es traslladen al torrent sanguini de manera moderada i en un període de temps relativament curt. Un exemple d'àpat *medium* són mongetes vermelles, pa, salami i formatge.

- *Small*: Aquests són àpats on els CHO, malgrat en tinguin una petita quantitat, es desprenen molt ràpidament al torrent sanguini. En són exemples una barreta de xocolata i gerds o truita amb pa, espinacs i tomàquet.

Tots aquests grups es descriuen a la taula 3.1, on es pot comprovar la corresponent informació nutricional que fa referència a cada un dels grups. Es pot observar que la quantitat de fibra és una dada de la que disposem a molts pocs àpats i que per tant no es podrà utilitzar per a entrenar el model de manera coherent, també es pot apreciar que els CHO, que s'espera que siguin el factor més condicionant en les corbes de la RA, es troben ben definits en cadascuna de les 54 instàncies.

Classificació	*Fast*	*Large*	*Medium*	*Small*
Nombre de mostres	6.00	10.00	26.00	12.00
CHO (g) max	104.00	126.00	75.00	38.00
CHO (g) mitja	68.83	94.80	54.53	22.58
CHO (g) min	42.00	75.00	27.00	10.00
CHO NaN	0.00	0.00	0.00	0.00
Energia (kCa) max	609.06	2913.92	2842.27	526.66
Energia (kCal) mitja	413.80	920.32	506.40	252.34
Energia (kCal) min	234.07	331.00	240.00	150.47
Energia NaN	0.00	0.00	2.00	0.00
Fibra (g) max	NaN	22.00	18.00	13.70
Fibra (g) mitja	NaN	14.20	5.96	3.56
Fibra (g) min	NaN	6.40	0.00	0.00
Fibra NaN	6.00	8.00	10.00	4.00
Grassa (g) max	17.00	80.00	80.00	24.00
Grassa (g) mitja	8.68	28.11	16.43	7.48
Grassa (g) min	2.50	0.80	0.90	1.10
Grassa NaN	0.00	0.00	6.00	1.00
Proteina (g) max	25.00	146.00	135.00	45.00
Proteina (g) mitja	14.70	33.93	17.02	17.42
Proteina (g) min	10.20	12.80	4.60	4.50

Table 3.1: Resum amb els valors que defineixen cada grup d'àpats

Cal fer especial ressò en que aquests quatre grups d'àpats poden ser combinats en només dos grups, podent agrupar-se els àpats *fast* i *small* en la categoria *Fast* (F) i els grups *large* i *medium* en la categoria *Non-Fast* (NF). Degut al reduït nombre d'instàncies disponibles a cadascuna de les quatre categories, aquesta última és de fet la classificació triada per l'entrenament del model en combinació amb el valor dels CHO que és el principal factor diferenciant entre els dos subgrups de cada una de les noves categories. De tota manera s'han fet experiments tant amb diferents agrupacions com amb combinacions de diferents variables de les que disposem durant la fase de disseny amb l'objectiu d'aconseguir un model tant condicionat com sigui possible sense assolir un sobre-entrenament. Per tant, tal i com ja s'ha especificat, s'utilitzen les categories F i NF amb una codificació tipus *one-hot* i els CHO normalitzats a valors entre 0 i 1.

3.2 Model

El model que s'utilitza per a la producció de corbes de la RA segueix la mateixa estructura d'un model generador d'imatges, però amb les modificacions pertinents per a ajustar-se a la generació de series temporals i per a evitar els inconvenients propis de les GAN que han aparegut en el seu desenvolupament, el principal dels quals ha estat la no convergència. Per a afrontar-lo s'ha implementat una modificació de la xarxa *Wasserstein* (13), això implica que tant el discriminador

com el conjunt de l'arquitectura en si, que és a la combinació de generador i discriminador, s'han entrenat utilitzant un optimitzador *RMSprop* amb una ràtio d'aprenentatge de 0.00001. De tota manera s'ha mantingut l'estructura de classificador binari al discriminador, tenint una funció sigmoide a la capa de sortida d'aquest i utilitzant la funció d'error *binary crossentropy* en tot el model. Per a l'entrenament s'ha utilitzat la funció *train on batch* de la llibreria *Keras* fent que per a cada època es faci un entrenament del generador i dos del discriminador, que ha entrenat amb un paquet de dades reals i un de generades. Es considera per tant que el model que s'ha fet servir és un model *Wasserstein* hibrid, ja que no s'han implementat tots els aspectes d'aquest.

L'arquitectura exacta de cada component del model és la que es presenta en els apartats 3.2.1 i 3.2.2.

3.2.1 Generador

Pel generador es fa servir una estructura similar a la que s'empra en un model generador d'imatges condicionat, adaptant el model a les series unidimensionals que pretenem generar. Per a mantenir la temporalitat en les dades s'utilitzen capes convolucionals en les que els filtres donen pes als valors consecutius. El model rep dos tipus d'entrades diferents, per un costat la dimensió latent i per altre les variables. Cadascuna de les capes d'aquest model, representat a la figura 3.1 és la següent:

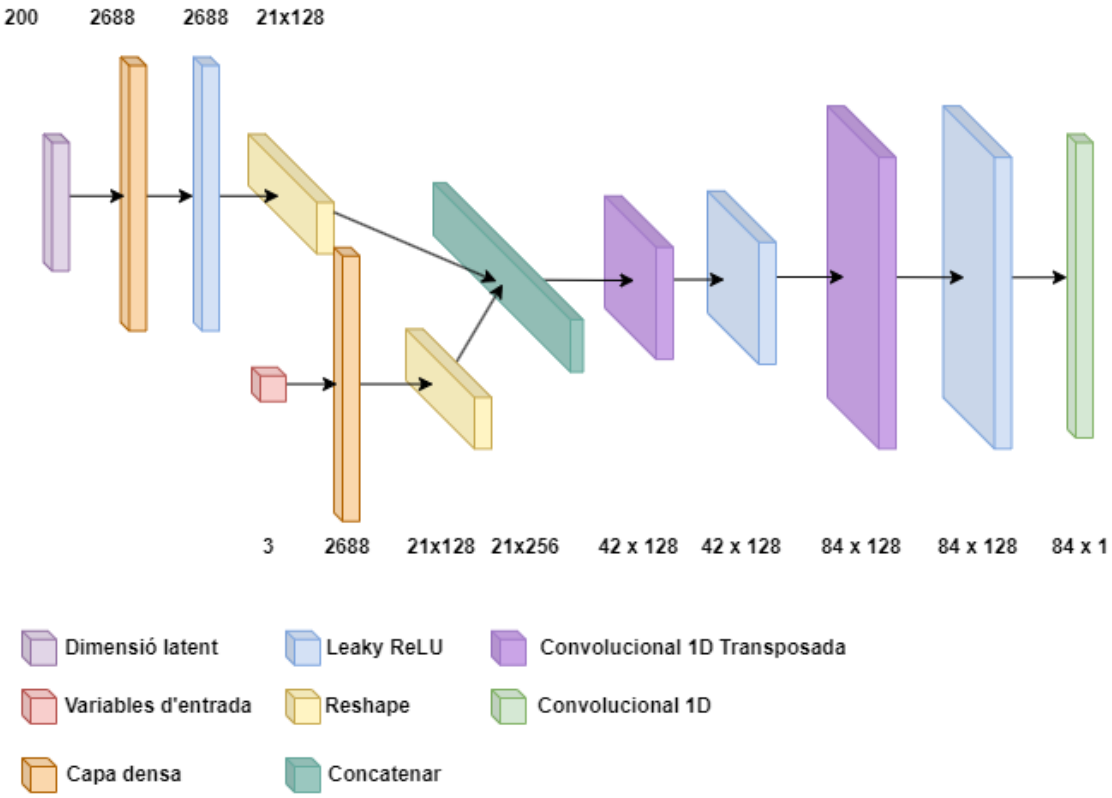


Figure 3.1: Arquitectura del generador implementat al projecte

- Una capa d'entrada de dimensió 200, que equival a la dimensió latent que es passa al model.
- Una capa densa de dimensió 2688 i funció d'activació lineal que augmenta la primera capa d'entrada.
- Una capa leaky rectified linear unit (LeakyReLU) amb $\alpha = 0.2$ que homogeneïtza la capa anterior.
- Una capa de *Reshape* que afegeix una segona dimensió a la capa anterior passant a ser de 28 amb una profunditat de 128.
- Una capa d'entrada que en aquest cas té una dimensió 3 però que pot ser ampliada o reduïda depenent de la quantitat de variables amb la que es vulgui condicionar el model. En aquest cas es fan servir dues variables categòriques codificades mitjançant el mètode *one-hot* i una variable numèrica.
- Una capa densa de dimensió 2688 i funció d'activació lineal que augmenta aquesta capa anterior i la iguala a la primera entrada al model.
- Igualment que es fa amb la primera branca d'entrada al model s'afegeix una capa *Reshape*

per tal d'afegir la segona dimensió i tenir forma $21x128$.

- En la següent capa és concatenen les dues branques d'entrada al model en una sola capa amb un tensor unidimensional de mida 21 i profunditat 256, ja que aquest només ens ha de donar un tensor de sortida.

- Una capa de tipus convolucional transposada unidimensional que augmenta la mida del tensor, passant a 42 disminuint i la profunditat a 128.

- Una capa LeakyReLU amb $\alpha = 0.2$ que homogeneïtza la capa anterior.

- Una capa de tipus convolucional transposada unidimensional fa que augmenta la mida del tensor, passant a 84 i conservant la profunditat en 128.

- Novament una capa LeakyReLU amb $\alpha = 0.2$.

- Una capa convolucional unidimensional que agrega les profunditats de la capa anterior en un sol tensor de sortida de mida 84, que equival a les series temporals a generar i per tant fent alhora de capa de sortida del model.

3.2.2 Discriminador

De la mateixa manera que passa amb el generador, el discriminador és un model per si mateix i té una arquitectura que recorda als models convolucionals de classificació d'imatges, tot i que amb tensors unidimensionals. Degut a que aquest ha de rebre les variables que condicionen a cada una de les series temporals, tant les sintètiques com les originals, el discriminador també tindrà dues capes d'entrada però solament una de sortida, essent la seva arquitectura de capes la que es veu representada a la figura 3.2 i explicades a continuació:

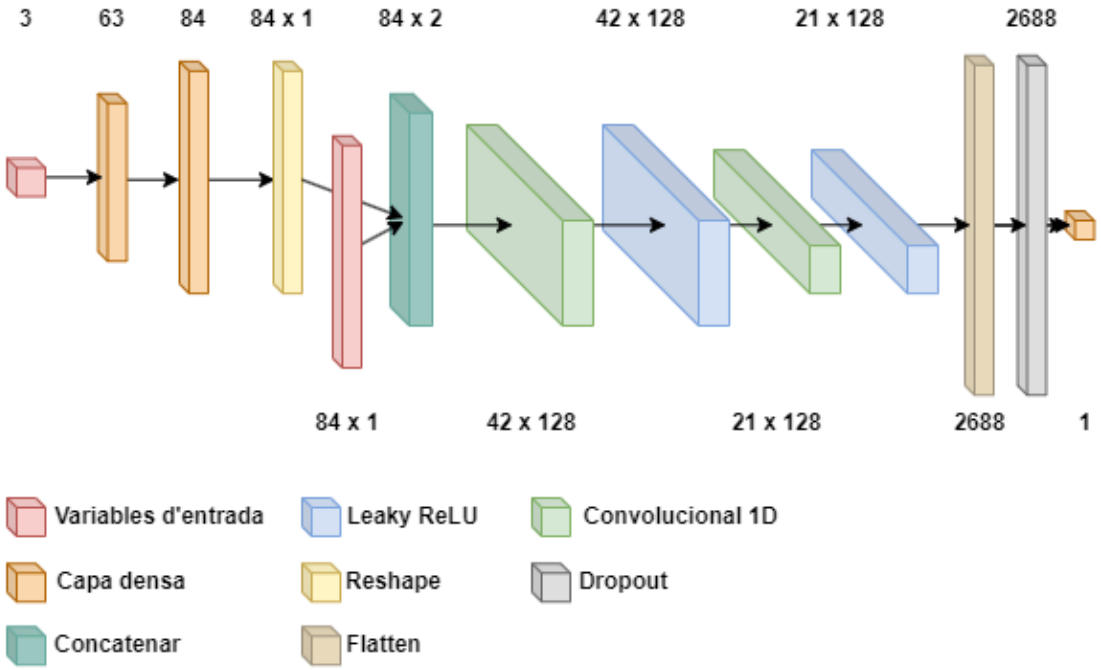


Figure 3.2: Arquitectura del discriminador implementat al projecte

- Primera capa d'entrada que, a igual que amb el generador té la dimensió 3, que es correspon al nombre de variables que condicionen el model.
- Capa densa de dimensió 63 que amplia les variables d'entrada
- Capa densa que igual que l'anterior l'amplia i li dona una dimensió 84, equivalent a la mida de les sèries temporals a discriminar.
- Capa de *Reshape* que afegeix profunditat al tensor tot i que el manté a $1x84$.
- Segona capa d'entrada que en aquest cas fa referencia a les series temporals que rep el model, en aquest cas ja té la dimensió $1x84$.
- Capa que concatena les dues branques d'entrada en una sola amb mida 84 i profunditat 2.
- Capa convolucional unidimensional que aplica un filtre per passar a una mida de 42 amb una profunditat de 128.
- Una capa LeakyReLU amb $\alpha = 0.2$ que homogeneïtza la capa anterior.
- Capa convolucional unidimensional que redueix la mida del tensor a 42.
- Novament una capa LeakyReLU amb $\alpha = 0.2$.

- Capa *Flatten* que elimina la profunditat passant a una mida senzilla de 2688.
- Capa de *Dropout* que afavoreix a reduir el sobre-entrenament del model.
- Capa densa de sortida amb funció d'activació sigmoïdal per tal d'obtenir una predicció de falsedat o veracitat de la serie temporal equivalents a 0 o 1.

3.3 Mecanismes d'Avaluació

Per tal d'avaluar la validesa de les mostres generades i del model en si, s'han dissenyat un seguit d'experiments i juntament amb un conjunt de mètriques seleccionades específicament per a comparar seqüencies de la RA, es busquen uns resultats amb prou rellevància estadística. Com que s'utilitzen dues classes d'àpat per a simplificar el model (*Fast* (F) i *Non-Fast* (NF)) en combinació amb la quantitat de CHO, es fan servir mètriques que comparen entre aquests dos grups les seves principals característiques descriptives. Per a fer aquests anàlisis es generen conjunts de mostres sintètiques en els que es pretén que els grams de CHO, que representen l'altra variable que condiciona el model, repliquin la mateixa distribució de valors de les mostres originals. Aquesta és la que es mostra a la imatge 3.3. Això fa que malgrat en aquest set de dades d'avaluació es corri el risc de perdre representació de valors de CHO d'àpats que no tenim al set original, es pugui fer una comparació més ajustada i la validació dels resultats sigui més correcta.

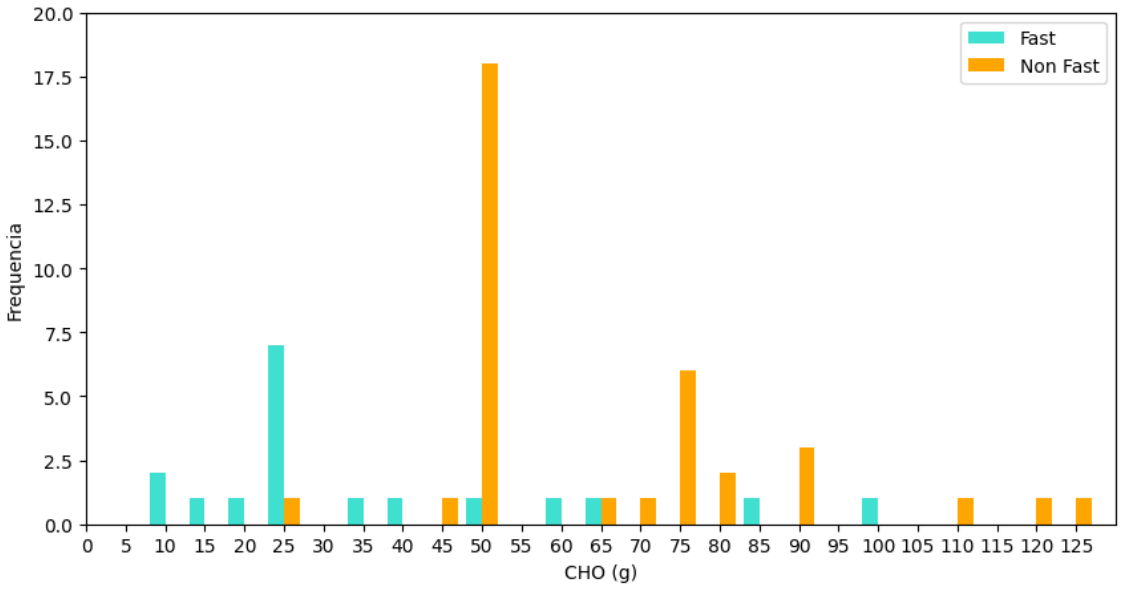


Figure 3.3: Distribució de valors de hidrats de carboni (CHO) de les mostres originals

Les mètriques a avaluar són les següents:

- **Temps fins al primer pic:** El primer pic de glucosa en la corba de la RA, que normalment també és el punt màxim, és un indicador de si un àpat ha de ser considerat de tipus F (que tenen el pic abans dels primers 10 minuts després de la ingesta) o NF (on el pic ocorre un cop transcorreguts uns 10 minuts).

- **AUC:** Essent aquesta una representació clara del total de CHO que proporciona un àpat al llarg del temps, els àpats de tipus F tenen valors d'*àrea sota la corba* (AUC) molt reduïts en comparació a la contrapart NF.

- **Pendent mitjà:** Malgrat no és excessivament notable, hi ha una diferència entre el pendent a la corba de la RA després del pic entre cadascuna de les classes, tenint un pendent més pronunciat les classes F.

També per a validar el model s'han dissenyat un seguit de proves per a comprovar la relació entre les dades d'entrada i les de sortida així com la similitud entre els tipus de series tant reals com generades.

- Es generen corbes de la RA fixant la dimensió latent i la quantitat de CHO de l'àpat, variant les etiquetes categòriques, primer per a generar corbes F i després corbes NF.

- Es fixen totes les variables que condicionen al generador i fer petites modificacions a la dimensió latent, la qual cosa hauria de resultar en corbes similars.

- Es fixen la dimensió latent i la classe d'àpat generat, fent modificacions en la quantitat de CHO dins el rang de mínim i màxim que tenim a les dades originals. S'espera que a mesura que s'augmenta la quantitat de CHO en un àpat el pic de la corba de la RA sigui també més alt.

- Es calculen les distàncies Jensen–Shannon (JS), que donen una estimació de la similitud o diferencia entre dues distribucions de dades amb la mateixa mida, generant així mapes de calor que comparen les mostres entre elles i en blocs.

4 | Planificació i metodologia

4.1 Especificacions tècniques

La programació de la totalitat del treball s'ha dut a terme en llenguatge Python i a través de l'Integrated Development Environment (IDE) Visual Studio Code. Concretament les versions dels diferents programes i llibreries que s'han utilitzat són les mostrades a la taula 4.1. No s'especifiquen les llibreries pròpies de la versió concreta de Python, malgrat s'han utilitzat, ja que se les considera implícites a la versió d'aquest. Pel que fa al maquinari, s'ha fet servir un ordinador amb les especificacions tècniques presentades a la taula 4.2.

Programari	**Versió**
Sistema Operatiu	Windows 10
Python	3.10.11
Visual Studio Code	1.81.1
Numpy	1.22
Tensorflow	2.12.0
Keras	2.3.1
Scipy	1.10.0
Pandas	1.5.3

Table 4.1: Versions del programari utilitzat

Component	**Model**
Processador	Intel Core i7-4770 CPU 3.40GHz
Font d'alimentació	Corsair TX 750M 750 Watt
Placa mare	Asus Z87-A
Memòria RAM	DDR3 16GB

Table 4.2: Especificacions i models del maquinari utilitzat

4.2 Planificacció

Aquest projecte s'ha desenvolupat seguint una metodologia per iteracions que s'ha triat degut a la complexitat del model a desenvolupar i disponibilitat de temps i recursos. El procés d'entrenament ha estat relativament lent i els mètodes d'afinat de les xarxes tipus GAN no són senzills d'implementar de manera programàtica al no disposar d'una mètrica senzilla d'avaluació de les series resultants i per tant la metodologia escollida a resultat la més optima.

Les iteracions s'han fet de manera setmanal des del mes de febrer de 2023 fins al mes de setembre

de 2023, amb reunions d'entre 30 i 60 minuts amb el tutor per avaluar el progrés del projecte. El transcurs de les diferents etapes del projecte es veu representat al diagrama de la figura 4.1 on es pot apreciar que s'ha mantingut un procediment orgànic, lògic i coherent amb la metodologia iterativa, realitzant tasques primer de prototipat d'un model no condicionat, després afegint variables que el condicionin i progressivament realitzant-li millores continues, implementant les modificacions de *Wasserstein*, provant combinacions de variables diferents i definint altres mètriques d'avaluació entre d'altres. En l'etapa final del projecte la tasca principal ha estat la redacció del document que correspon a la memòria i l'execució d'experiments i proves per a l'obtenció dels resultats definitius.

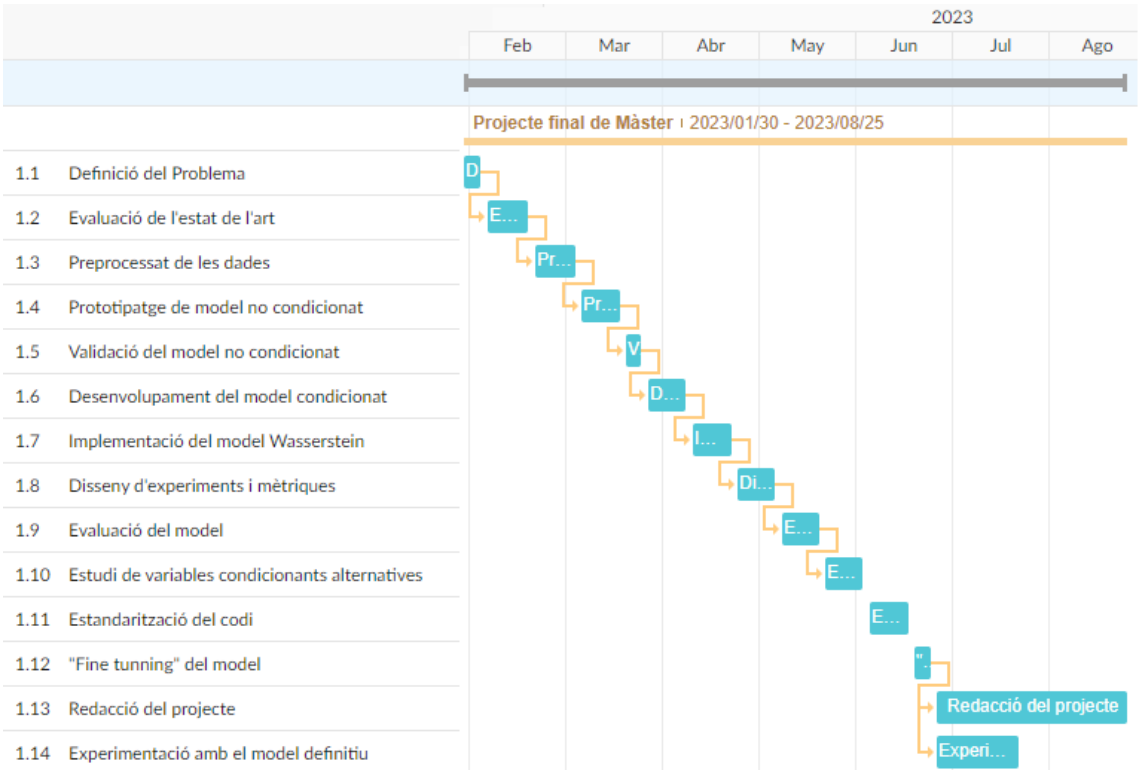


Figure 4.1: Diagrama de Gantt de la distribució de les tasques al llarg del temps

5 | Resultats i discussió

En aquest capítol es presenten els resultats obtinguts en aquest projecte segons la metodologia explicada a la secció 3.3. Per a generar les mostres sintètiques s'han utilitzat diferents instàncies del model entrenat amb les característiques definides en 3.2, la qual cosa fa que els resultats siguin més contrastats. Les corbes reals són la totalitat de les mostres originals després del pre-processament corresponent.

Es presentaran en aquest capítol els diferents estadístics i mètriques que ja s'han definit amb la comparació entre les mostres i els resultats dels experiments realitzats per a comprovar la validesa del model i influencia de les variables d'entrada en les corbes resultants, amb les pertinents visualitzacions de dades.

5.1 Estadístics d'avaluació

En aquesta secció analitzem les dades amb l'objectiu de revelar patrons, tendències i conclusions significatives. Aquestes mètriques aquí ens permeten comprendre millor les relacions entre les mostres reals i les generades per oferir una visió objectiva dels resultats obtinguts en els nostres experiments posteriors. Tant a la taula 5.1 com a la taula 5.2 es presenten respectivament les mètriques per a cada una de les dues classes i per a les dades generades essent cada columna una de les mètriques. S'han utilitzat un nombre de mostres de 500 corbes sintètiques i el total de les 54 corbes reals disponibles.

	Pics	*àrea sota la corba* (AUC)	Pendent
Fast (F) sintètic	5.96(3.11)	52.22(23.51)	4.88(5.02)
Fast (F) real	7.39(5.81)	48.89(41.48)	4.09(5.34)
Non-Fast (NF) sintètic	12.94(4.46)	2246.36(402.86)	3.01(2.9)
Non-Fast (NF) real	15.03(9.54)	2509.47(1526.31)	2.4(3.25)

Table 5.1: Comparació dels valors de *mitja (desviació estàndard)* entre els dos tipus de mostres generades i reals.

	Pics	*àrea sota la corba* (AUC)	Pendent
Fast (F) sintètic	6.0(4.0 – 7.0)	48.37(35.57 – 66.28)	3.4(0.48 – 7.7)
Fast (F) real	5.5(3.0 – 8.75)	32.02(18.16 – 75.12)	2.84(−0.0 – 4.4)
Non-Fast (NF) sintètic	13.0(9.0 – 15.0)	2199.63(1951.41 – 2495.81)	2.2(0.7 – 4.46)
Non-Fast (NF) real	14.0(8.75 – 18.25)	2194.31(1457.66 – 3134.72)	0.96(−0.08 – 4.18)

Table 5.2: Comparació de la distribució *mediana (q1-q3)* entre els dos tipus de mostres generades i reals.

Tal i com es mostra a les dues taules els valors per a les mostres F són similars entre si i d'igual manera passa amb les mostres NF. Això ens indica que el model ha aprés correctament a representar corbes segons la categoria de cada àpat però no assegura que es vegi representada la quantitat de CHO amb la que condicionem el model, ja que aquesta s'ha introduït a les dades sintètiques seguint la mateixa distribució que a les originals.

Es notable destacar que les desviacions estàndard que es presenten a la taula 5.1 són sempre inferiors en les mostres sintètiques que en les mostres reals. Això és degut a que la variabilitat aportada per un model generatiu, malgrat sigui adversari i de complexitat i nombre de paràmetres elevat, generalment és menor que la pròpia dels processos naturals, com en aquest cas ho és el metabolisme. Caldria per tant, per a augmentar la desviació estàndard, incrementar el nombre de paràmetres que es fan servir per a entrenar el model, la qual cosa també dificultaria el propi entrenament de la xarxa, fent-la més propensa a errors ja resolts amb la configuració actual com és la no convergència o el sobre-entrenament. S'ha considerat per tant que la disminució en la desviació estàndard és acceptable.

5.2 Resultats experimentals

En aquesta secció, presentarem els resultats dels nostres experiments empírics, els quals s'han dut a terme amb l'objectiu d'analitzar i validar les hipòtesis plantejades a la recerca així com les estadístiques de la secció anterior. Aquests resultats ofereixen una visió concreta i quantitativa de com es comporten les variables d'interès en el context de les nostres proves. Al llarg d'aquesta secció, es detallen els procediments experimentals juntament amb les dades específiques relacionades i es discuteixen les implicacions directes d'aquestes troballes en el context de la nostra investigació.

5.2.1 Dimensió latent i carbohidrats fixats

Amb aquest experiment es determina quin és l'efecte de la classe referent al tipus d'àpat. Malgrat en la representació de la imatge 5.1 només es mostra aquest experiment per a un valor de CHO fixat i la mateixa dimensió latent, en totes les repeticions efectuades s'ha obtingut el mateix resultat: Per a les 500 series generades de cada classe sempre s'obté la mateixa seqüencia, presentant a més les mètriques que li son característiques de pics, AUC i pendent, tal i com es mostra a la taula 5.3.

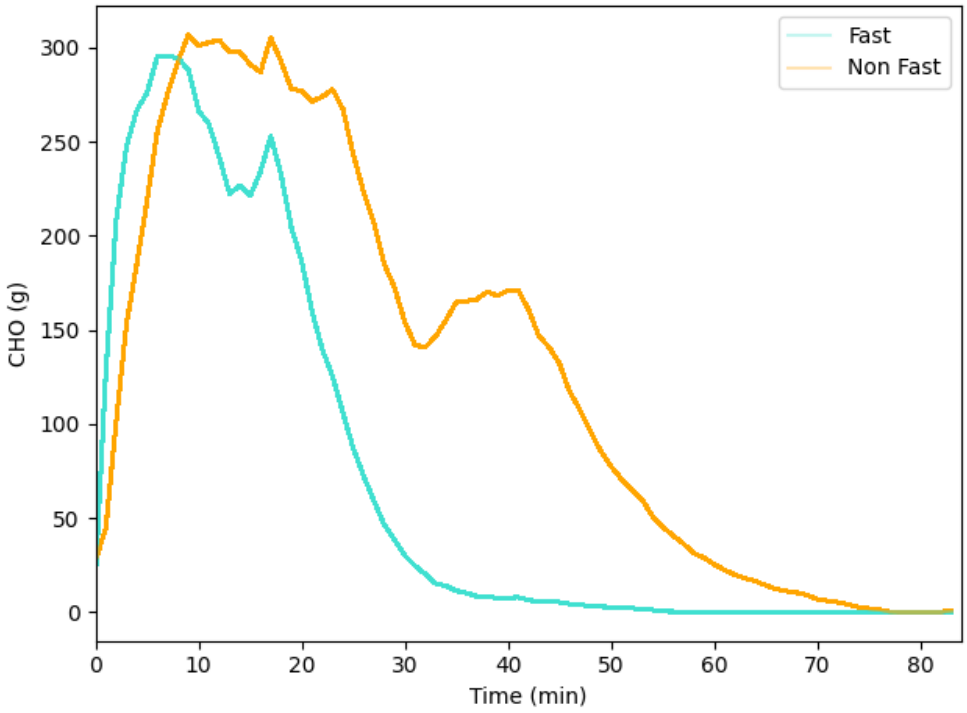


Figure 5.1: Representació de mostres de la ràtio d'aparició de glucosa exògena (RA) fixant la dimensió latent i els hidrats de carboni (CHO)

	Pics	AUC	Pendent
Fast (F)	6.00	30.65	0.10
Fast (F) real	7.39	48.89	4.09
Non-Fast (NF)	13.00	2439.09	0.96
Non-Fast (NF) real	15.03	2509.47	2.4

Table 5.3: Avaluació del pics, *àrea sota la corba* (AUC) i pendent de les mostres de ràtio d'aparició de glucosa exògena (RA) amb dimensió latent i hidrats de carboni (CHO) fixats.

5.2.2 Modificacions a la dimensió latent amb les classes i carbohidrats fixats.

Es fa aquesta prova per a comprovar quin és l'efecte de la dimensió latent en les mostres generades i considerar si aquestes modifiquen directament la sortida del model. Es veu com a mesura que es fan més variacions a l'espai latent el soroll aleatori en les mostres augmenta i per tant es veu representada l'estocàstica dels individus i el metabolisme a través d'aquest paràmetre. Es pot apreciar que en la imatge 5.2 que referencia a aquest experiment, com malgrat s'està partint de la mateixa distribució i solament s'hi apliquen canvis, i per tant la variació en els valors originals no és excessiva, aquesta ja és suficient per a ser representativa. En la visualització s'està mostrant cada una de les seqüencies amb una transparència constant, és per això que es mostren amb més intensitat les series que tenen una variació respecte la mediana més petita.

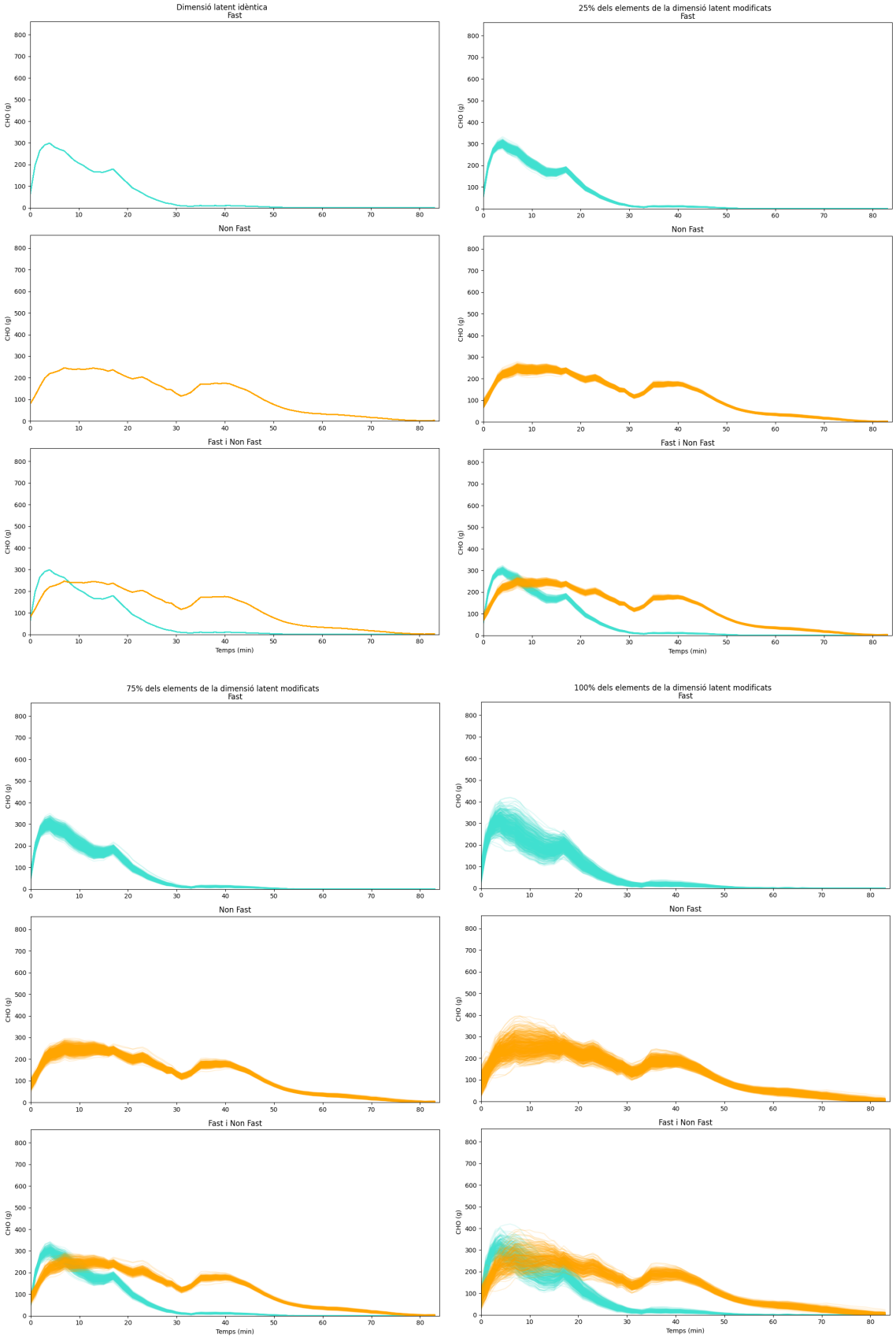


Figure 5.2: Representació de mostres modificant la dimensió latent i fixant els hidrats de carboni (CHO).

5.2.3 Modificacions a la dimensió latent i als carbohidrats

Aquí es pretén demostrar quin és l'efecte dels CHO en el model. S'han generat una sèrie de 500 etiquetes per a cada classe amb el corresponent valor de CHO, seguint la distribució original. Amb aquestes seqüencies s'ha produït la visualització 5.3, on s'assigna un color a cada valor de CHO en una escala cromàtica proporcional i ascendent. Es pot apreciar com a mesura que augmenta aquest valor també ho fa l'altura dels pics en les corbes generades, essent així que els tons més blavosos de la visualització pertanyen sempre a sèries amb els pics més baixos. De tota manera aquest efecte dels CHO és menor del que s'esperaria. Caldria per tant fer un nou entrenament on els pesos de cada variable condicional fossin més balancejats, donant menys rellevància a la classe i més als CHO.

5.2.4 Distàncies Jensen–Shannon

Tal i com es mostra a la figura 5.4 les distàncies Jensen–Shannon (JS) demostren la similitud entre les seqüencies generades per a cadascuna de les classes. S'han produït un total de 18 mostres tant F com NF i també s'han seleccionat a l'atzar 18 de les seqüencies originals de tipus NF. Això s'ha fet per tal de garantir que es fan les comparacions de distàncies sempre amb la mateixa quantitat de mostres, essent les més restrictives les mostres originals de tipus F.

Prenent com a valors de similitud objectiva la distància mitjana de les comparacions de mostres real-real, es pot apreciar com malgrat els resultats sintètic-sintètic són en general més baixos, els valors es mantenen molt propers als objectius en la comparativa real-sintètica. Aquest fenomen es deu a que, com ja s'ha explicat, les dades generades per models d'aprenentatge automàtic tendeixen a tenir una variància menor entre si, sobretot al estar generades amb sets de dades tant reduïts.

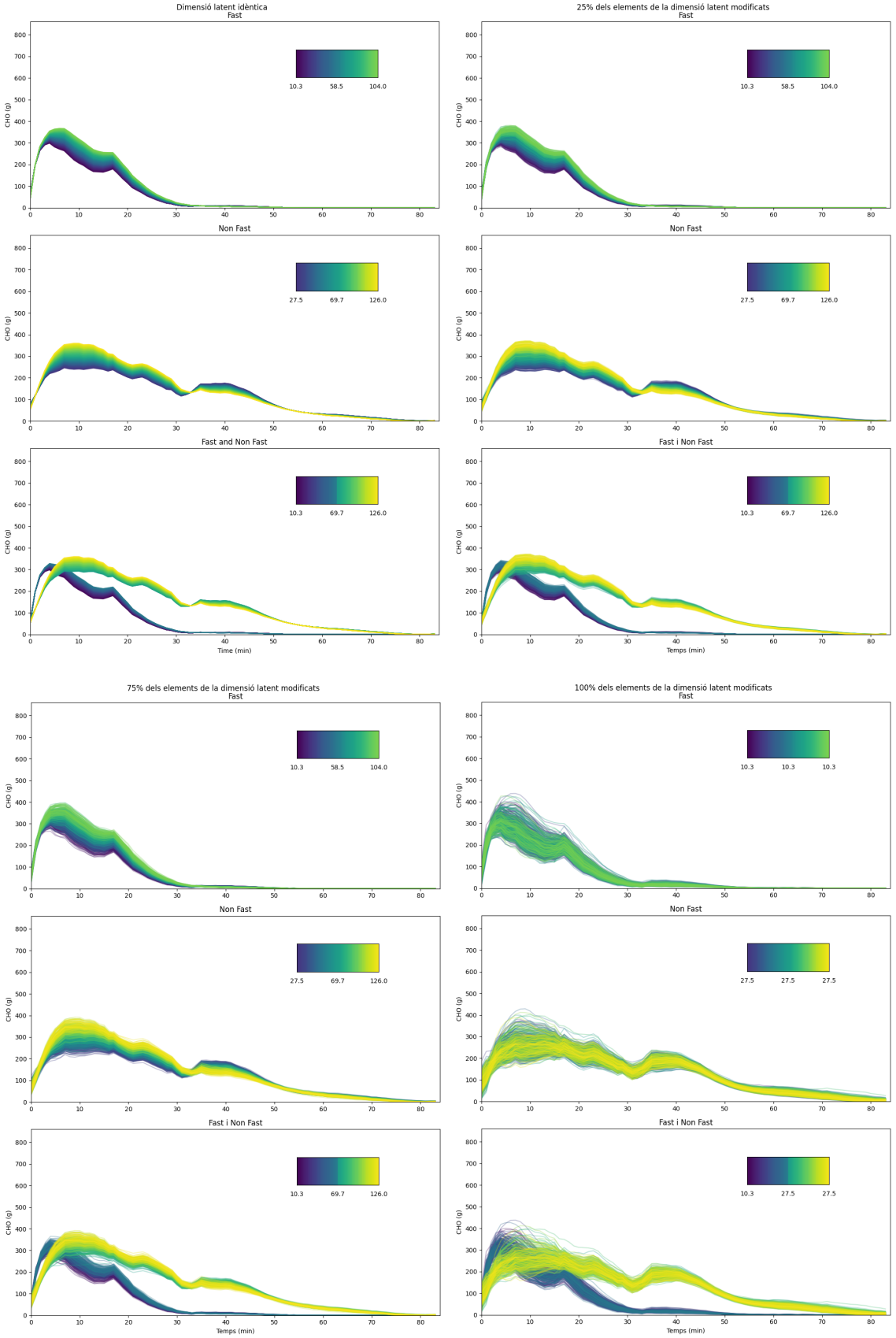


Figure 5.3: Representació de mostres modificant la dimensió latent i els hidrats de carboni (CHO)

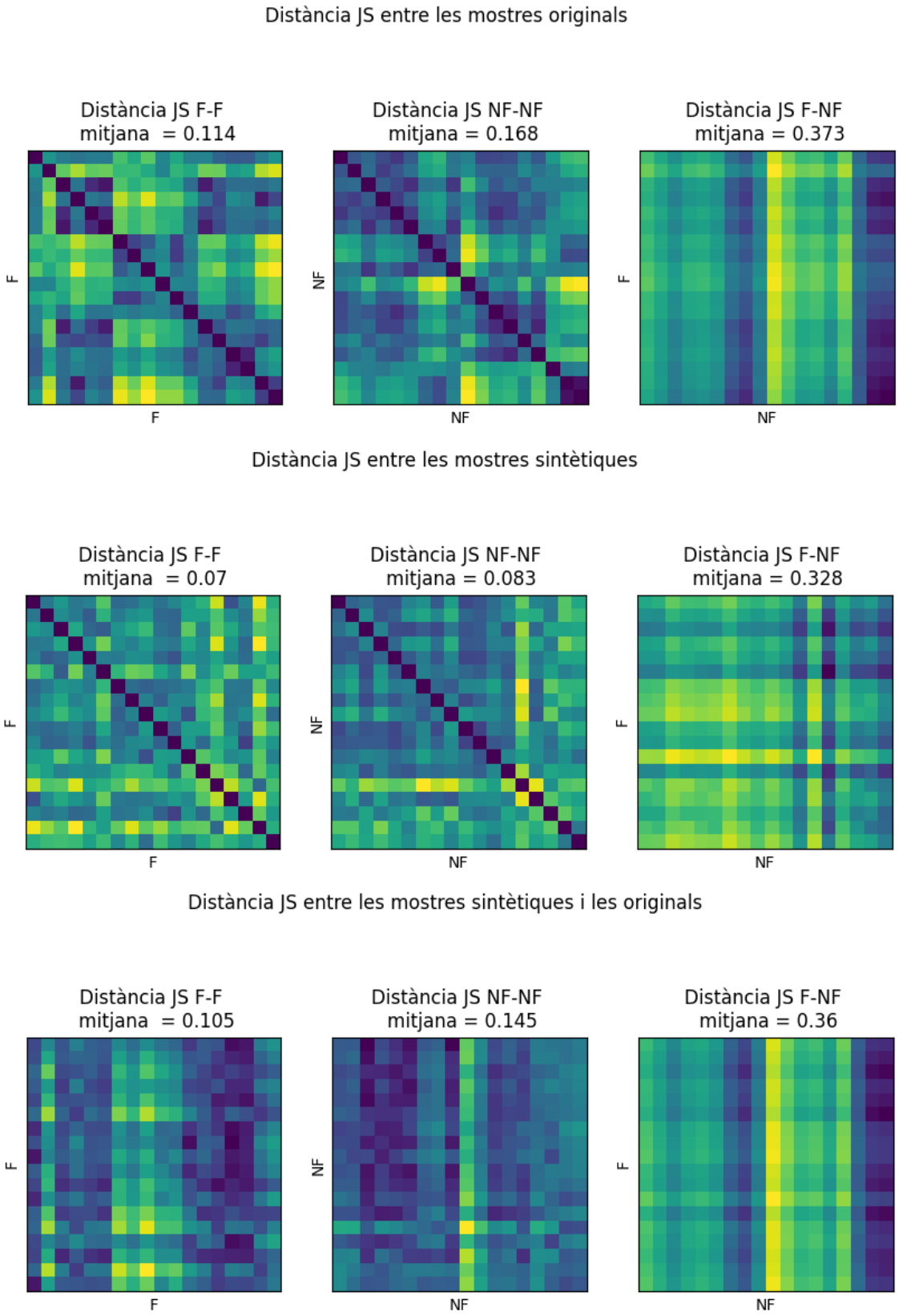


Figure 5.4: Representació dels mapes de calor de la distància Jensen–Shannon (JS) entre els tipus de mostres i per classes

6 | Discussió i conclusió

6.1 Discussió

L'ús de models generatius profunds condicionats per generar corbes de la ràtio d'aparició de glucosa exògena (RA) representa un avanç significatiu en la recerca de la monitorització i el control de la glucosa en persones amb diabetis mellitus de tipus 1 (T1DM). En aquesta secció de discussió, analitzarem els resultats i les implicacions del nostre treball. En primer lloc, i tal i com es presenta en els resultats obtinguts de l'entrenament i corresponent avaluació del model desenvolupat en aquest projecte, els models de tipus xarxa generativa adversària condicionada (C-GAN) demostren poder representar amb alts nivells de veracitat les tendències reals de corbes de la RA. A través d'un estudi a fons de la naturalesa d'aquestes dades i el seu correcte processament, s'han aconseguit generar de manera indefinida series temporals condicionades al tipus d'àpat i a la quantitat de hidrats de carboni (CHO) que aquest conté. Aquests models no només poden ajudar a comprendre millor la resposta de l'organisme a l'administració d'insulina i d'altres tractaments, sinó que també poden facilitar la presa de decisions clíniques més informades.

Un dels aspectes més destacats del nostre treball és la capacitat dels models C-GAN per a capturar la complexitat de les respostes individuals. Cada persona pot tenir una resposta única a la ingesta d'un àpat i els nostres models són capaços d'adaptar-se a aquestes diferències. Això té el potencial de personalitzar encara més els tractaments, la qual cosa pot portar a un millor control glúcic i una millora en la qualitat de vida dels pacients amb diabetis. Malgrat que l'us d'altres variables com pot ser la quantitat de fibra o matèria grassa, de les que també disposem, ha donat resultats igualment vàlids, la informació que aquests han aportat al model ha estat mínima i ha dificultat el seu entrenament, aportant una millora reduïda en comparació als inconvenients que suposen (per exemple la necessitat d'eliminar àpats on aquesta informació no és disponible). S'ha optat per tant per un model més senzill i amb el nombre de condicionants el més petit possible.

La generació de corbes de la RA a partir d'aquests models pot ser útil en la simulació de diferents escenaris clínics. Això permetria als metges i pacients que pateixen T1DM explorar com certs canvis en el tractament o l'estil de vida poden afectar els nivells de glucosa en sang, la qual cosa pot ser una eina valuosa per a l'educació i la presa de decisions compartides.

En el model presentat s'ha demostrat el clar efecte del tipus d'àpat en la corba generada, variable

que ha estat clarament assimilada en l'entrenament del model i que es considera la més influent alhora de produir noves sèries. Pel que fa a la representació que té la quantitat de CHO, es veu com la seva rellevància en les corbes resultants és baixa (s'aprecia al produir series de dimensió latent i tipus d'àpat fixats, modificant solament els CHO). Això es pot justificar gràcies al fet que el cos humà no en pot absorbir quantitats molt grans i de fet que es veu reflectit en la pròpia condició real de les dades originals.

En aquest projecte s'han assolit la totalitat dels objectius proposats podent afirmar que el model desenvolupat compleix amb aquestes especificacions. El model no condicionat dissenyat inicialment a mode de validació del procés ha estat capaç de reproduir la temporalitat de les dades en l'origen i les sèries de sortida de la xarxa C-GAN són clarament condicionades a les variables d'entrada i malgrat la limitació en el pes dels CHO s'han obtingut uns valors en les mètriques comparatives vàlids. S'han pogut generar corbes de manera indefinida produint de forma algorítmica tant el tipus d'àpat com els CHO dels àpats. Els diferents experiments empírics han demostrat també com el model és capaç d'introduir l'estocàstica pròpia de la cinètica real del metabolisme.

Cal ser conscients però de les limitacions que presenta aquest projecte, la primera de les quals és la reduïda quantitat de dades de les que es disposa. L'entrenament d'un model d'aquestes característiques requereix de volums molt majors per a poder fer una representació acurada i precisa de la realitat i per tant és provable que hi hagi modes que no s'hagin pogut representar degut a la seva omissió a les dades disponibles. Addicionalment a aquesta limitació es presenta el problema dels valors mancants als àpats, fent que no es puguin utilitzar algunes de les variables que es coneix que serien d'utilitat com pot ser la quantitat de fibra. La combinació d'aquests dos factors fa que les entrades que condicionen al model hagin de ser reduïdes, específiques i patir transformacions prèvies a l'entrenament.

La segona limitació del model és la seva validació experimental, ja que malgrat si es pot utilitzar en simuladors de pacients virtuals entre altres, per a la pertinent posada en producció són necessaris tant assaigs clínics com la validació feta per un comitè d'ètica.

Aquests dos aspectes fan que la projecció del treball al futur sigui el desenvolupament de noves validacions basades en la demostració empírica, sintetitzant corbes de la RA amb un set de dades no utilitzat en l'entrenament del model de les que coneixem les corbes reals i comparant els valors obtinguts. Degut a l'àmbit d'ús principal on s'ha enfocat el treball també cal provar que en un simulador de pacient virtual amb T1DM que utilitzi models matemàtics per a la simulació de corbes de la RA es produeix una millora en la resposta dels valors de glucosa en sang.

6.2 Conclusió

En aquest treball, hem aplicat models generatius profunds per generar corbes de la RA. Els resultats obtinguts a partir d'aquests models demostren la seva eficàcia en la generació de corbes precises i coherents que capturen el comportament de la glucosa exògena en l'organisme. Mitjançant una àmplia varietat d'experiments i anàlisis estadístics, hem validat la capacitat d'aquests models de produir corbes que s'assemblen a les observacions reals, la qual cosa suggereix la seva utilitat en la simulació i predicció de respostes glúciques en diferents escenaris. En conclusió es pot afirmar que el model dissenyat es capaç de generar corbes de la RA realistes i veraces, consistents amb els valors d'entrada que les condicionen i amb un potencial molt elevat en la millora de l'experimentació, la virtualització del metabolisme dels éssers humans i el desenvolupament tant d'algoritmes com de models d'aprenentatge automàtic, essent així el camp de la generació de dades sintètiques un camp molt alentidor a explorar en el futur recent.

A més, aquest estudi ha destacat la importància de la utilització de models generatius profunds en la recerca biomèdica, ja que permeten explorar i comprendre millor les dinàmiques de la glucosa exògena en sang, la qual cosa pot tenir implicacions significatives en el tractament i control de la diabetis i altres malalties relacionades.

En última instància, els models generatius profunds es perfilen com una eina prometedora en la generació de corbes de glucosa exògena, la qual cosa podria contribuir a millorar l'atenció mèdica i el disseny de teràpies personalitzades en el futur. Tanmateix, és important destacar que aquest camp de recerca encara presenta desafiaments i àrees de millora, com la necessitat de dades d'alta qualitat i l'optimització de la precisió dels models. En resum, aquest treball estableix les bases per a futures investigacions en l'ús de models generatius profunds en el camp de la glucosa exògena en sang, amb l'objectiu de millorar la qualitat de vida de les persones afectades per trastorns metabòlics.

Referències

[1] C. G. Machado, M. Winroth, D. Carlsson, P. Almström, V. Centerholt, and M. Hallin, "Industry 4.0 readiness in manufacturing companies: challenges and enablers towards increased digitalization." *Procedia CIRP*, vol. 81, pp. 1113–1118, Jan 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827119305670

[2] Melmed, Solmo and S. Polonsky, Kenneth and Larsen, P. Reed and Kronenberg, Henrry M., *Williams. Tratado de endocrinología*, 13th ed. Elsevier Health Sciences Spain, 2017.

[3] Jameson, J. Larry, *Harrison's Endocrinology*, 3rd ed. Mc Graw Hill Education, 2013. [Online]. Available: https://books.google.es/books?id=yFzFQMCMOGYC

[4] J. Walsh and R. Roberts, *Pumping insulin: everything you need for success on a smart insulin pump*. Torrey Pines Press San Diego, CA, 2006, vol. 4.

[5] American Diabetes Association, "Glycemic targets: Standards of medical care in diabetes—2020," *Diabetes Care*, vol. 43, no. Supplement 1, pp. S66–S76, 2020. [Online]. Available: https://care.diabetesjournals.org/content/43/Supplement_1/S66

[6] International Diabetes Federation, *IDF DIABETES ATLAS*, 9th ed. International Diabetes Federation, 2019. [Online]. Available: www.diabetesatlas.org

[7] I. Contreras and J. Vehi, "Artificial intelligence for diabetes management and decision support: Literature review," *J Med Internet Res*, vol. 20, no. 5, p. e10775, May 2018. [Online]. Available: http://www.jmir.org/2018/5/e10775/

[8] L. Ruthotto and E. Haber, "An introduction to deep generative modeling," *CoRR*, vol. abs/2103.05180, 2021. [Online]. Available: https://arxiv.org/abs/2103.05180

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014. [Online]. Available: https://arxiv.org/abs/1406.2661

[10] M. Mirza and S. Osindero, "Conditional generative adversarial nets," 2014. [Online]. Available: https://arxiv.org/abs/1411.1784

[11] R. Durall, A. Chatzimichailidis, P. Labus, and J. Keuper, "Combating mode collapse in

gan training: An empirical analysis using hessian eigenvalues," 2020. [Online]. Available: https://arxiv.org/abs/2012.09673

[12] K. Li and J. Malik, "Implicit maximum likelihood estimation," *CoRR*, vol. abs/1809.09087, 2018. [Online]. Available: http://arxiv.org/abs/1809.09087

[13] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," 2017. [Online]. Available: https://arxiv.org/abs/1701.07875

[14] C. Dalla Man, M. Camilleri, and C. Cobelli, "A system model of oral glucose absorption: Validation on gold standard data," *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 12, pp. 2472–2478, 2006.

[15] R. Basu, B. Di Camillo, G. Toffolo, A. Basu, P. Shah, A. Vella, R. Rizza, and C. Cobelli, "Use of a novel triple-tracer approach to assess postprandial glucose metabolism," *American Journal of Physiology-Endocrinology and Metabolism*, vol. 284, no. 1, pp. E55–E69, 2003, publisher: American Physiological Society. [Online]. Available: https://journals.physiology.org/doi/full/10.1152/ajpendo.00190.2001

[16] E. Lehmann and T. Deutsch, "A physiological model of glucose-insulin interaction in type 1 diabetes mellitus," *Journal of Biomedical Engineering*, vol. 14, no. 3, pp. 235–242, 1992, annual Scientific Meeting. [Online]. Available: https://www.sciencedirect.com/science/article/pii/014154259290058S

[17] C. Della Man, A. Caumo, and C. Cobelli, "The oral glucose minimal model: Estimation of insulin sensitivity from a meal test," *IEEE Transactions on Biomedical Engineering*, vol. 49, no. 5, pp. 419–429, 2002.

[18] F. M. Leon Vargas, "Design and implementation of a closed-loop blood glucose control system in patients with type 1 diabetes," http://purl.org/dc/dcmitype/Text, Universitat de Girona, 2013. [Online]. Available: https://dialnet.unirioja.es/servlet/tesis?codigo=84737

[19] S. Strozyk, A. Rogowicz-Frontczak, S. Pilacinski, J. LeThanh-Blicharz, A. Koperska, and D. Zozulinska-Ziolkiewicz, "Influence of resistant starch resulting from the cooling of rice on postprandial glycemia in type 1 diabetes," *Nutrition & Diabetes*, vol. 12, no. 1, p. 21, 2022. [Online]. Available: https://doi.org/10.1038/s41387-022-00196-1

[20] S. Buck, C. Krauss, D. Waldenmaier, C. Liebing, N. Jendrike, J. Högel, B. M. Pfeiffer,

C. Haug, and G. Freckmann, "Evaluation of meal carbohydrate counting errors in patients with type 1 diabetes," *Experimental and Clinical Endocrinology & Diabetes*, vol. 130, no. 7, pp. 475–483, 2022, publisher: Georg Thieme Verlag KG. [Online]. Available: http://www.thieme-connect.de/DOI/DOI?10.1055/a-1493-2324

[21] J. Noguer, I. Contreras, O. Mujahid, A. Beneyto, and J. Vehi, "Generation of individualized synthetic data for augmentation of the type 1 diabetes data sets using deep learning models," *Sensors*, vol. 22, no. 13, 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/13/4944

[22] O. Mujahid, I. Contreras, A. Beneyto, I. Conget, M. Giménez, and J. Vehi, "Conditional synthesis of blood glucose profiles for t1d patients using deep generative models," *Mathematics*, vol. 10, no. 20, 2022. [Online]. Available: https://www.mdpi.com/2227-7390/10/20/3741

[23] C. Dalla Man, F. Micheletto, D. Lv, M. Breton, B. Kovatchev, and C. Cobelli, "The uva/padova type 1 diabetes simulator: New features," *Journal of diabetes science and technology*, vol. 8, pp. 26–34, 05 2014.

[24] T. Alkhalifah, H. Wang, and O. Ovcharenko, "Mlreal: Bridging the gap between training on synthetic data and real data applications in machine learning," in *82nd EAGE Annual Conference & Exhibition*, vol. 2021. European Association of Geoscientists & Engineers, 2021, pp. 1–5.

[25] Z. Qin, Z. Liu, P. Zhu, and Y. Xue, "A gan-based image synthesis method for skin lesion classification," *Computer Methods and Programs in Biomedicine*, vol. 195, p. 105568, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169260720302418

[26] H. Rashid, M. A. Tanveer, and H. Aqeel Khan, "Skin lesion classification using gan based data augmentation," in *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2019, pp. 916–919.

[27] F. Zhu, F. Ye, Y. Fu, Q. Liu, and B. Shen, "Electrocardiogram generation with a bidirectional LSTM-CNN generative adversarial network," *Scientific Reports*, vol. 9, no. 6734, 05 2019.

[28] Y. Deng, L. Lu, L. Aponte, A. M. Angelidi, V. Novak, G. E. Karniadakis, and C. S. Mantzoros, "Deep transfer learning and data augmentation improve glucose levels prediction in type 2 diabetes patients," *npj Digital Medicine*, vol. 4, no. 1, 2021.

[29] E. Piacentino, A. Guarner, and C. Angulo, "Generating synthetic ecgs using gans for anonymizing healthcare data," *Electronics*, vol. 10, no. 4, 2021.

[30] J. Yoon, L. N. Drumright, and M. van der Schaar, "Anonymization through data synthesis using generative adversarial networks (ads-gan)," *IEEE Journal of Biomedical and Health Informatics*, vol. 24, no. 8, pp. 2378–2388, 2020.

[31] Q. Thames, A. Karpur, W. Norris, F. Xia, L. Panait, T. Weyand, and J. Sim, "Nutrition5k: Towards automatic nutritional understanding of generic food," *CoRR*, vol. abs/2103.03375, 2021.

[32] P. Herrero, J. Bondia, C. C. Palerm, J. Vehí, P. Georgiou, N. Oliver, and C. Toumazou, "A simple robust method for estimating the glucose rate of appearance from mixed meals," *Journal of Diabetes Science and Technology*, vol. 6, no. 1, pp. 153–162, 2012, publisher: SAGE Publications Inc. [Online]. Available: https://doi.org/10.1177/193229681200600119

A | Anexes

A.1 Package Loading

[137]:
```
# Import necessary libraries
import os
import warnings
import imageio
from IPython.display import clear_output
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import patches as mpatches
import scipy.io as sio
from scipy.spatial.distance import jensenshannon
from scipy.stats import entropy
from scipy.signal import savgol_filter, find_peaks

from numpy.random import randn, randint
from numpy import expand_dims, zeros, ones, vstack

import keras
from keras.optimizers import Adam, RMSprop
from keras.models import Model, Sequential, load_model
from keras.layers import Input, Dense, Reshape, Flatten, Conv1D,␣
↪Conv1DTranspose, LeakyReLU, Dropout, Embedding, Concatenate
from keras.utils.vis_utils import plot_model
from keras import backend

# Ignore warnings
warnings.filterwarnings("ignore")
```

```
# Set path for Graphviz if needed
os.environ["PATH"] += os.pathsep + 'C:/Program Files/Graphviz/bin/'
```

A.2 Data Preparation

A.2.1 Data Loading

[138]:
```
# Get the Current Working Directory as CWD
CWD = os.getcwd()

# Load the mat file in CWD\data\raw\Meals_Library_Original.mat
mat = sio.loadmat(os.path.join(CWD, 'data', 'raw',
                  'Meals_Library_Original.mat'))

# Use the elements in mat['Library_Meal'][0][0][0][0][0][0][0] as the␣
 ↪column names
colnames = [x[0] for x in mat['Library_Meal'][0][0][0][0][0][0][0]]

df_all_summary = pd.DataFrame()
df_all_ra = pd.DataFrame()
for i in range(0, len(mat['Library_Meal'][0][0])-2):
    # Read mat['Library_Meal'][0][0][0][0][0][0].shape as a dataframe
    df_group = pd.DataFrame(mat['Library_Meal'][0][0][i][0][0][0])
    df_group_ra = pd.DataFrame(mat['Library_Meal'][0][0][i][0][0][1])
    # Use the first row as the column names
    df_group.columns = colnames
    # Drop the first row
    df_group.drop(df_group.index[0], inplace=True)
    # Flatten every column in df2 except the column 'Nom'ArithmeticError
    for col in df_group.columns:
        if col != 'Nom':
            df_group[col] = df_group[col].apply(lambda x: x[0][0])
    df_group['Grup'] = i+1
    # Drop the row with R in the column 'Nº aliment'
```

```
    df_group.drop(df_group[df_group['Nº aliment'] == 'R'].index,␣
 ↪inplace=True)
    df_all_summary = pd.concat([df_all_summary, df_group],␣
 ↪ignore_index=True)
    df_all_ra = pd.concat([df_all_ra, df_group_ra], ignore_index=True)

# Transform the dataframe df_all_ra into a column of arrays
df_all_ra['RA'] = df_all_ra.apply(lambda x: np.array(x), axis=1)
df_all_ra.reset_index(inplace=True)
# Rename the column 'index' to 'Id'
df_all_ra.rename(columns={'index': 'Id'}, inplace=True)
# Keep the columns 'Nº aliment' and 'RA'
df_all_ra = df_all_ra[['Id', 'RA']]

# Reset index of df_all_summary
df_all_summary.reset_index(inplace=True)
# Rename the column 'index' to 'Id'
df_all_summary.rename(columns={'index': 'Id'}, inplace=True)
# Merge df_all_summary and df_all_ra
df_all_summary = pd.merge(df_all_summary, df_all_ra, on='Id', how='left')
df_all_summary.drop('Id', axis=1, inplace=True)

# Transform the column Nom into a string
df_all_summary['Nom'] = df_all_summary['Nom'].astype(str)
df_all_summary['Nom'] = df_all_summary['Nom'].str.replace('[', '',␣
 ↪regex=False)
df_all_summary['Nom'] = df_all_summary['Nom'].str.replace(']', '',␣
 ↪regex=False)
df_all_summary['Nom'] = df_all_summary['Nom'].str.lower()
```

A.3 Ingredient names normalization

[139]:
```
df_all_summary['Nom'] = df_all_summary['Nom'].str.replace(
    'standard', 'std', regex=False)\
    .str.replace("+", ",", regex=False)\
    .str.replace('and', ',', regex=False)\
    .str.replace('c,y', 'candy', regex=False)\
    .str.replace('with', ',', regex=False)\
    .str.replace('white', '', regex=False)\
    .str.replace('benas', 'beans', regex=False)\
    .str.replace('low gi (250ml of water): boiled ', '', regex=False)\
    .str.replace('high gi (285ml of water): ', '', regex=False)\
    .str.replace('low content of ', '', regex=False)\
    .str.replace('medium content of ', '', regex=False)\
    .str.replace('lightly salted ', '', regex=False)\
    .str.replace('fat-free ice-cream', 'light ice cream', regex=False)\
    .str.replace('^ | $', '', regex=True)\
    .str.replace(' , ', ', ', regex=False)
```

[140]:
```
df_all_summary.head()
```

[140]:
```
   Nº aliment                                                Nom␣
↪Classificació  \
0           1  'milk,  rice, pear, bran-cookies, low-fat chee...          ␣
↪ M
1           3        'milk,  bread, low-fat cheese, butter, oil'          ␣
↪ M
2          18                                     'pearl barley'          ␣
↪ M
3          22            ' bread, eggs, margarine, orange juice'          ␣
↪ M
4          23                  'powdered nutritional supplement'          ␣
↪ M
```

```
   CHO (g) Grassa (g) Proteïna (g) Fibra (g) Energia (kCal) Temps (min) ␣
↪Grup  \
0       52       10.50         14.50       2.90          362.09          300   ␣
↪ 1
1    52.50       10.50         14.50       2.60          362.33          300   ␣
↪ 1
2       50        1.30          9.20       5.10          243.15          240   ␣
↪ 1
3       50          13            13          2          360.90          300   ␣
↪ 1
4       50          12            14          0          366.87          300   ␣
↪ 1

                                                    RA
0  [80.06129765, 95.85627093, 108.7114773, 119.94...
1  [30.84068552, 126.1904248, 217.6727333, 247.42...
2  [29.18326202, 121.9148437, 185.8368828, 209.21...
3  [15.79361365, 74.10652237, 127.6169281, 149.95...
4  [69.4217951, 89.28182292, 106.727891, 122.2116...
```

A.3.1 Transform data to the specifications of the model

Here we select which numerical variables we are going to train the model with, normalize them if corresponds and determine the categorical variables to work. In this section is where the Groups of the column Classificació are maped to fit other categories

[142]:
```
# Define the numerical and categorical variable lists
numerical_variable_list = ['CHO (g)']
categorical_variable_list = ['Classificació']


# Replace all '-' with np.nan in the numerical variables
df_all_summary[numerical_variable_list] =␣
↪df_all_summary[numerical_variable_list].replace(
    '-', np.nan)
```

```
# Map the groups in the 'Classificació' column to fit other categories
df_all_summary['Classificació'] = df_all_summary['Classificació'].map(
    {'S': 'Fast', 'F': 'Fast', 'M': 'Non Fast', 'L': 'Non Fast'})

# Get the minimum and maximum of each class
min_max = df_all_summary.groupby(
    'Classificació').agg({'CHO (g)': ['min', 'max']})

# Add a column with CHO (g) original
df_all_summary['CHO (g) original'] = df_all_summary['CHO (g)']

# Apply a min-max normalization to the numerical variables
df_all_summary[numerical_variable_list] =␣
 ↪(df_all_summary[numerical_variable_list] -␣
 ↪df_all_summary[numerical_variable_list].min()) / (
    df_all_summary[numerical_variable_list].max() -␣
 ↪df_all_summary[numerical_variable_list].min())

# Replace the '-' with np.nan in the real data
real_data = df_all_summary[numerical_variable_list +
                           categorical_variable_list].replace('-', np.nan)

# Turn the columns from categorical_variable_list into categorical
real_data[categorical_variable_list] =␣
 ↪real_data[categorical_variable_list].astype(
    'category')
```

[148]:
```
df_all_summary[['CHO (g) original'] +
               categorical_variable_list].head()
```

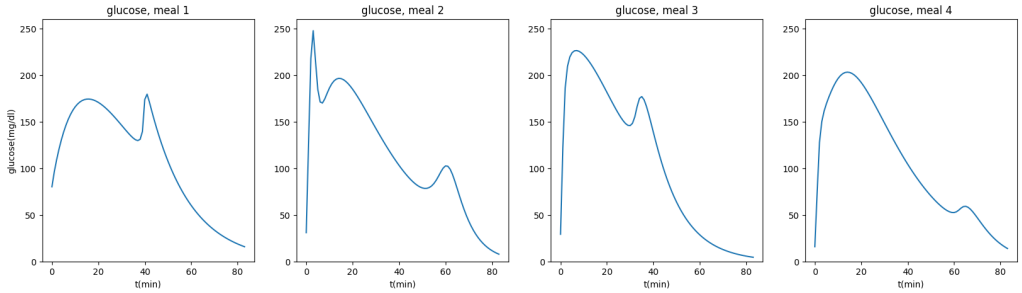
[148]:

	CHO (g) original	Classificació
0	52	Non Fast
1	52.50	Non Fast
2	50	Non Fast

```
3                50        Non Fast
4                50        Non Fast
```

Sample plot of four RA curves

[111]:
```
# Plot the first 4 elements of df_all_summary['RA'] in a row of 4 plots␣
 ↪with RA(mg/dl) and t(min)
# make the y axis always from 0 to 260

plt.figure(figsize=(20, 5))
plt.subplot(1, 4, 1)
plt.plot(df_all_summary['RA'][0])
plt.ylim(0, 260)
plt.title('glucose, meal 1')
plt.xlabel('t(min)')
plt.ylabel('glucose(mg/dl)')
plt.subplot(1, 4, 2)
plt.plot(df_all_summary['RA'][1])
plt.ylim(0, 260)
plt.title('glucose, meal 2')
plt.xlabel('t(min)')
plt.subplot(1, 4, 3)
plt.plot(df_all_summary['RA'][2])
plt.ylim(0, 260)
plt.title('glucose, meal 3')
plt.xlabel('t(min)')
plt.subplot(1, 4, 4)
plt.plot(df_all_summary['RA'][3])
plt.ylim(0, 260)
plt.title('glucose, meal 4')
plt.xlabel('t(min)')
plt.show()
```



A.3.2 Apply the proper transformations to summarize the data

[151]:
```
# Select the columns to keep
columns_to_keep = ['Classificació',
                   'CHO (g) original', 'Proteïna (g)', 'Grassa (g)',␣
 ↪'Fibra (g)', 'Energia (kCal)']

# Create a new dataframe with the selected columns
df_reduced = df_all_summary[columns_to_keep].copy()

# Replace '-' with np.nan in the dataframe
df_reduced.replace('-', np.nan, inplace=True)

# Convert all columns except 'Classificació' to numerical
df_reduced[columns_to_keep[1:]
           ] = df_reduced[columns_to_keep[1:]].apply(pd.to_numeric)

# Group the dataframe by 'Classificació' and aggregate the data
df_reduced_summary = df_reduced.groupby('Classificació').agg(
    ['size', 'count', 'mean', 'min', 'max'])

# Rename the columns
df_reduced_summary.columns = [
    '_'.join(col).strip() for col in df_reduced_summary.columns.values]
```

```
# Apply size-count
df_reduced_summary['CHO (g) original NaN'] = df_reduced_summary['CHO (g)␣
 ↪original_size'] - \
    df_reduced_summary['CHO (g) original_count']
df_reduced_summary['Proteïna NaN'] = df_reduced_summary[
    'Proteïna (g)_size'] - df_reduced_summary['Proteïna (g)_count']
df_reduced_summary['Grassa NaN'] = df_reduced_summary['Grassa (g)_size']␣
 ↪- \
    df_reduced_summary['Grassa (g)_count']
df_reduced_summary['Fibra NaN'] = df_reduced_summary['Fibra (g)_size'] - \
    df_reduced_summary['Fibra (g)_count']
df_reduced_summary['Energia NaN'] = df_reduced_summary['Energia␣
 ↪(kCal)_size'] - \
    df_reduced_summary['Energia (kCal)_count']

df_reduced_summary['Number_of_samples'] = df_reduced_summary['CHO (g)␣
 ↪original_size']

# Remove the columns with the suffix '_size' and '_count'
df_reduced_summary.drop(df_reduced_summary.filter(
    regex='_size|_count').columns, axis=1, inplace=True)

# Show with 2 decimals
pd.options.display.float_format = '{:.2f}'.format
df_transposed = df_reduced_summary.transpose()

# Sort by the index
df_transposed.sort_index(inplace=True)

# Place 'Number_of_samples' as the first row
df_transposed = df_transposed.reindex(
    ['Number_of_samples'] + list(df_transposed.index[:-1]))
```

[152]:
```
df_transposed
```

[152]:

Classificació	Fast	Non Fast
Number_of_samples	18.00	36.00
CHO (g) original NaN	0.00	0.00
CHO (g) original_max	104.00	126.00
CHO (g) original_mean	38.00	65.72
CHO (g) original_min	10.00	27.00
Energia (kCal)_max	609.06	2913.92
Energia (kCal)_mean	306.16	628.14
Energia (kCal)_min	150.47	240.00
Energia NaN	0.00	2.00
Fibra (g)_max	13.70	22.00
Fibra (g)_mean	3.56	6.87
Fibra (g)_min	0.00	0.00
Fibra NaN	10.00	18.00
Grassa (g)_max	24.00	80.00
Grassa (g)_mean	7.91	20.32
Grassa (g)_min	1.10	0.80
Grassa NaN	1.00	6.00
Number_of_samples	18.00	36.00
Proteïna (g)_max	45.00	146.00
Proteïna (g)_mean	16.51	22.65
Proteïna (g)_min	4.50	4.60

A.4 Model building

A.4.1 Data arrangements

The length of the numerical variable list greater than 0 implies that the model will be a conditional GAN, if the length is 0, it will be a vanilla GAN This list is selected at the data transformation section

[11]:
```
if len(categorical_variable_list) > 0:


    trainX = np.array(df_all_summary['RA'].tolist())
    trainy_df = pd.get_dummies(real_data, columns=[
```

```
        'Classificació'], prefix=['Classificació'])


    # Sort the columns with the prefix 'Classificació' in alphabetical␣
↪order
    trainy_df = trainy_df.reindex(sorted(trainy_df.columns), axis=1)
    # Put the columns in numerical_variable_list at the end of the␣
↪dataframe
    trainy_df = trainy_df[[c for c in trainy_df if c not in␣
↪numerical_variable_list] +
                          [c for c in numerical_variable_list if c in␣
↪trainy_df]]


    trainy = np.array(trainy_df)


    trainX = np.asarray(trainX).astype('float32')
    trainy = np.asarray(trainy).astype('float32')
else:
    trainX = np.array(df_all_summary['RA'].tolist())
    trainX = np.asarray(trainX).astype('float32')
```

A.4.2 Model development and functions for the Vanilla GAN (Unconditioned)

[12]:
```
# define the standalone generator model
if len(numerical_variable_list) == 0 and len(categorical_variable_list)␣
↪== 0:
    def define_generator(latent_dim):
        model = Sequential()
        model.add(Dense(84, activation='relu', input_dim=latent_dim))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Reshape((1, 84)))
        model.add(Conv1D(84, 1, padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv1D(84, 1, padding='same'))
        model.add(LeakyReLU(alpha=0.2))
```

```
        model.add(Flatten())
        model.add(Dense(84, activation='linear'))
        model.add(LeakyReLU(alpha=0.01))

    # Define the discriminator model
    def define_discriminator(opt, loss):
        model = Sequential()
        # Add a dense layer with 200 units and ReLU activation function␣
↪with input shape of 84
        model.add(Dense(200, activation='relu', input_dim=84))
        # Add a leaky ReLU activation function with alpha=0.2
        model.add(LeakyReLU(alpha=0.2))
        # Reshape the output to have shape (1, 200)
        model.add(Reshape((1, 200)))
        # Add a 1D convolutional layer with 200 filters, kernel size of 1␣
↪and ReLU activation function
        model.add(Conv1D(200, 1, activation='relu'))
        # Add a leaky ReLU activation function with alpha=0.2
        model.add(LeakyReLU(alpha=0.2))
        # Add a 1D convolutional layer with 128 filters, kernel size of␣
↪1, ReLU activation function and same padding
        model.add(Conv1D(128, 1, activation='relu', padding="same"))
        # Add a leaky ReLU activation function with alpha=0.2
        model.add(LeakyReLU(alpha=0.2))
        # Flatten the output
        model.add(Flatten())
        # Add a dense layer with 1 unit and linear activation function
        model.add(Dense(1, activation='linear'))
        # Compile the model with the given optimizer and loss function
        model.compile(loss=loss, optimizer=opt, metrics=['accuracy'])
        # Return the discriminator model
        return model

    # Generate real samples from the dataset
```

```
    def generate_real_samples(dataset, n_samples):
        # Choose n_samples random instances from the dataset
        ix = np.random.randint(0, dataset.shape[0], size=n_samples)
        # Retrieve the selected images
        X = dataset[ix]
        # Generate 'real' class labels (1)
        y = ones((n_samples, 1))
        # Return the real samples and their labels
        return X, y

    # Generate points in latent space as input for the generator
    def generate_latent_points(latent_dim, n_samples):
        # Generate n_samples points in the latent space
        x_input = randn(latent_dim * n_samples)
        # Reshape the points into a batch of inputs for the network
        x_input = x_input.reshape(n_samples, latent_dim)
        # Return the generated points
        return x_input

    # Use the generator to generate n fake examples, with class labels
    def generate_fake_samples(g_model, latent_dim, n_samples):
        # Generate points in latent space
        x_input = generate_latent_points(latent_dim, n_samples)
        # Predict outputs using the generator model
        X = g_model.predict(x_input, verbose=0)
        # Create 'fake' class labels (0)
        y = zeros((n_samples, 1))
        # Return the Synth samples and their labels
        return X, y

    # Define the combined generator and discriminator model, for updating␣
↪the generator
    def define_gan(generator, discriminator, opt, loss):
        # Make weights in the discriminator not trainable
```

```
        discriminator.trainable = False
        # Connect the generator and discriminator models
        model = Sequential()
        # Add the generator model
        model.add(generator)
        # Add the discriminator model
        model.add(discriminator)
        # Compile the model with the given optimizer and loss function
        model.compile(loss=loss, optimizer=opt, metrics=['accuracy'])
        # Return the combined model
        return model

    # Train the generator and discriminator
    def train(g_model, d_model, gan_model, latent_dim, n_epochs, n_batch,␣
↪dataset):
        d_loss_array = []
        g_loss_array = []

        # Determine half the size of one batch, for updating the␣
↪discriminator
        half_batch = int(n_batch / 2)
        # Manually enumerate epochs
        for i in range(n_epochs):
            # Clear the output of the cell
            clear_output(wait=True)
            # Prepare real samples
            x_real, y_real = generate_real_samples(dataset, half_batch)
            # Prepare fake examples
            x_fake, y_fake = generate_fake_samples(
                g_model, latent_dim, half_batch)

            # Update the discriminator with the real and fake samples
            d_model.train_on_batch(x_real, y_real)
            d_model.train_on_batch(x_fake, y_fake)
```

```
            # Prepare points in latent space as input for the generator
            x_gan = generate_latent_points(latent_dim, n_batch)
            # Create inverted labels for the fake samples
            y_gan = ones((n_batch, 1))
            # Update the generator via the discriminator's error
            gan_model.train_on_batch(x_gan, y_gan)
            # Combine the real and fake samples and their labels
            x, y = vstack((x_real, x_fake)), vstack((y_real, y_fake))
            # Update the discriminator with the combined samples
            d_loss, _ = d_model.train_on_batch(x, y)
            # Update the generator via the combined model
            g_loss = gan_model.train_on_batch(x_gan, y_gan)
            # Append the losses to their respective arrays
            d_loss_array.append(d_loss)
            g_loss_array.append(g_loss)
            # Plot the losses
            plt.plot(range(0, len(d_loss_array)), d_loss_array,␣
↪label='d_loss')
            plt.plot(range(0, len(g_loss_array)), g_loss_array,␣
↪label='g_loss')
            plt.legend()
            plt.title('Losses, epoch: ' + str(i))
            plt.show()
            plt.close()
            clear_output(wait=True)
            # Save generated samples every 5 epochs
            if i % 5 == 0:
                fake_samples = g_model.predict(
                    generate_latent_points(latent_dim, 10), verbose=0)
                for a in range(0, 10):
                    plt.plot(range(0, len(fake_samples[0])),␣
↪fake_samples[a])
                plt.title('Generated Samples, epoch: ' + str(i))
```

```
                # Save the plot to the CWD\old_model folder
                plt.savefig(os.path.join(CWD, 'old_model',
                            'epoch_' + str(i) + '.png'))
                plt.close()

    def wasserstein_loss(y_true, y_pred):
        return backend.mean(y_true * y_pred)
```

[13]:
```
# Check if there are no numerical or categorical variables
if len(numerical_variable_list) == 0 and len(categorical_variable_list)
 ↪== 0:
    # Define the dimensionality of the latent space
    latent_dim = 100
    # Define the number of training epochs
    n_epochs = 1000
    # Define the size of the batch of real and fake samples
    n_batch = 8
    # Define the optimizer for the models
    opt = RMSprop(lr=0.0005)
    # Define the loss function for the models
    loss = wasserstein_loss
    # Define the dataset to use for training
    dataset = trainX
    # Define the generator model
    g_model = define_generator(latent_dim)
    # Define the discriminator model
    d_model = define_discriminator(opt, loss)
    # Define the combined generator and discriminator model
    gan_model = define_gan(g_model, d_model, opt, loss)
    # Train the generator and discriminator models
    train(g_model, d_model, gan_model, latent_dim, n_epochs, n_batch)
```

A.4.3 Model development and functions for the Conditional GAN

[14]:
```
if len(categorical_variable_list) > 0:


   def generate_latent_points(latent_dim, n_samples, real_data,␣
↪numerical_variable_list, categorical_variable_list):
       input_data = real_data.copy()
       # Calculate the max and min of each numerical variable per␣
↪Classificació and add them as new columns
       for var in numerical_variable_list:
           input_data[var+'_max'] = input_data.groupby(
               categorical_variable_list)[var].transform(max)
           input_data[var+'_min'] = input_data.groupby(
               categorical_variable_list)[var].transform(min)
       # Remove the columns from the numerical_variable_list
       input_data.drop(numerical_variable_list, axis=1, inplace=True)
       # Drop duplicates
       input_data.drop_duplicates(inplace=True)
       # Create a list named expected_columns with the values from␣
↪input_data['Classificació'].unique() with the prefix 'Classificació_'
       expected_columns = ['Classificació_' +
                           str(x) for x in input_data['Classificació'].
↪unique()]
       random_strings = np.random.choice(
           input_data['Classificació'], size=n_samples)
       one_hot_arr = pd.DataFrame({"Classificació": random_strings})


       # join one_hot_arr and input_data on the column Classificació
       input_data_aux = pd.merge(one_hot_arr, input_data,
                                 on='Classificació', how='left')


       # For each variable in numerical_variable_list create a random␣
↪value between the max and min of each Classificació
       for var in numerical_variable_list:
```

```
            # Given the columns var+'_min' and var+'_max' generate a␣
↪random value between them
            input_data_aux[var] = np.random.uniform(
                input_data_aux[var+'_min'], input_data_aux[var+'_max'],␣
↪n_samples)

        # One hot encode the Classificació column
        one_hot = pd.get_dummies(input_data_aux, columns=[
            'Classificació'], prefix=['Classificació'])
        for column in expected_columns:
            if column not in one_hot.columns:
                one_hot[column] = 0

        # Sort the columns with the prefix 'Classificació_'␣
↪alphabetically
        one_hot_aux = one_hot[expected_columns].reindex(
            sorted(expected_columns), axis=1)
        # Add the numerical variables to the one_hot_aux dataframe
        for var in numerical_variable_list:
            one_hot_aux[var] = one_hot[var]

        x_input = randn(latent_dim * n_samples)
        # reshape into a batch of inputs for the network
        z_input = x_input.reshape(n_samples, latent_dim)
        # generate labels
        labels = np.array(one_hot_aux)
        return [z_input, labels]

    def generate_fake_samples(generator, latent_dim, n_samples,␣
↪input_data):
        # generate points in latent space
        z_input, labels_input = generate_latent_points(
```

```
        latent_dim, n_samples, input_data, numerical_variable_list,␣
↪categorical_variable_list)
    # predict outputs
    z_input = z_input.astype('float32')
    labels_input = labels_input.astype('float32')
    images = generator.predict([z_input, labels_input], verbose=0)
    # create class labels
    y = zeros((n_samples, 1))
    return [images, labels_input], y

def generate_real_samples(dataset, n_samples):
    # split into images and labels
    images, labels = dataset
    # choose random instances
    ix = randint(0, images.shape[0], n_samples)
    # select images and labels
    X, labels = images[ix], labels[ix]
    # generate class labels
    y = ones((n_samples, 1))
    return [X, labels], y

def load_real_samples(trainX, trainy):
    # load dataset
    # expand to 3d, e.g. add channels
    X = expand_dims(trainX, axis=-1)
    # convert from ints to floats
    X = X.astype('float32')
    # scale from [0,255] to [-1,1]
    return [X, trainy]

# define the standalone generator model
def define_generator(latent_dim, n_classes, n_num, output_shape):
    cat_inputs = Input(shape=(n_classes,))
    num_input = Input(shape=(n_num,))
```

```
        concatenated = Concatenate()([cat_inputs, num_input])
        n_nodes = 128 * int(output_shape/4)
        li = Dense(n_nodes)(concatenated)
        # reshape to additional channel
        li = Reshape((int(output_shape/4), 128))(li)
        # image generator input
        in_lat = Input(shape=(latent_dim,))
        gen = Dense(n_nodes)(in_lat)
        gen = LeakyReLU(alpha=0.2)(gen)
        gen = Reshape((int(output_shape/4), 128))(gen)
        # merge RA gen and label input
        merge = Concatenate()([gen, li])
        gen = Conv1DTranspose(128, (4), strides=(2),␣
↪padding='same')(merge)
        gen = LeakyReLU(alpha=0.2)(gen)
        gen = Conv1DTranspose(128, (4), strides=(2), padding='same')(gen)
        gen = LeakyReLU(alpha=0.2)(gen)
        # output
        out_layer = Conv1D(1, (int(output_shape/4)),
                           activation='relu', padding='same')(gen)
        # define model
        model = Model([in_lat, concatenated], out_layer)
        return model

    def wasserstein_loss(y_true, y_pred):
        return backend.mean(y_true * y_pred)

    # define the standalone discriminator model
    def define_discriminator(output_shape, n_classes, n_num, opt, loss):
        in_shape = (output_shape, 1)
        cat_inputs = Input(shape=(n_classes,))
        num_input = Input(shape=(n_num,))
        in_label = Concatenate()([cat_inputs, num_input])
```

```
        n_features = (n_classes + n_num)
        n_nodes = int(output_shape/4) * n_features
        li = Dense(n_nodes)(in_label)
        li = Dense(in_shape[0])(li)
        # reshape to additional channel
        li = Reshape((in_shape[0], 1))(li)

        # image input
        in_image = Input(shape=in_shape)
        # concat label as a channel
        merge = Concatenate()([in_image, li])
        # downsample
        fe = Conv1D(128, (3), strides=(2), padding='same')(merge)
        fe = LeakyReLU(alpha=0.2)(fe)
        # downsample
        fe = Conv1D(128, (3), strides=(2), padding='same')(fe)
        fe = LeakyReLU(alpha=0.2)(fe)
        # flatten feature maps
        fe = Flatten()(fe)
        # dropout
        fe = Dropout(0.4)(fe)
        # output
        out_layer = Dense(1, activation='sigmoid')(fe)
        # define model
        model = Model([in_image, in_label], out_layer)
        # compile model
        model.compile(loss=loss,
                      optimizer=opt, metrics=['accuracy'])
        return model

    # define the combined generator and discriminator model, for updating␣
↪the generator

    def define_gan(g_model, d_model, opt, loss):
```

```
        # make weights in the discriminator not trainable
        d_model.trainable = False
        # get noise and label inputs from generator model
        gen_noise, gen_label = g_model.input
        # get image output from the generator model
        gen_output = g_model.output
        # connect image output and label input from generator as inputs␣
↪to discriminator
        gan_output = d_model([gen_output, gen_label])
        # define gan model as taking noise and label and outputting a␣
↪classification
        model = Model([gen_noise, gen_label], gan_output)
        # compile model
        model.compile(loss=loss, optimizer=opt)
        return model
```

A.4.4 Functions used for experimental evaluation

[15]:
```
def real_value(value, min_max):
    return value * (min_max['max'] - min_max['min']) + min_max['min']



def generate_latent_points_based_on_real(n_samples, dataset):
    n_group = int(n_samples/4)
    # Pick 10 elements of dataset with the first element of [1,:] as 1␣
↪and 10 with the first element as 0
    data_1 = []
    data_0 = []
    labels_1 = []
    labels_0 = []
    for i in range(0, len(dataset[0])):
        if dataset[1][i][0] == 1:
            data_1.append(dataset[0][i].tolist())
            labels_1.append(dataset[1][i].tolist())
```

```
        else:
            data_0.append(dataset[0][i].tolist())
            labels_0.append(dataset[1][i].tolist())

    # Calculate the mean using numpy
    mean_labels_0 = np.mean([arr[-1] for arr in labels_0])
    mean_labels_1 = np.mean([arr[-1] for arr in labels_1])

    data_1_0 = []
    labels_1_0 = []
    data_1_1 = []
    labels_1_1 = []
    data_0_0 = []
    labels_0_0 = []
    data_0_1 = []
    labels_0_1 = []

    for i in range(0, len(labels_1)):
        if labels_1[i][2] > mean_labels_1:
            data_1_1.append(data_1[i])
            labels_1_1.append(labels_1[i])
        else:
            data_1_0.append(data_1[i])
            labels_1_0.append(labels_1[i])

    for i in range(0, len(labels_0)):
        if labels_0[i][2] > mean_labels_0:
            data_0_1.append(data_0[i])
            labels_0_1.append(labels_0[i])
        else:
            data_0_0.append(data_0[i])
            labels_0_0.append(labels_0[i])

    # Get the n_samples first elements of data_1_0, data_0_0
```

```
    data_1_0 = data_1_0[:n_group]
    data_0_0 = data_0_0[:n_group]
    # Get the n_samples last elements of data_1_1, data_0_1
    data_1_1 = data_1_1[-n_group:]
    data_0_1 = data_0_1[-n_group:]
    # Get the n_samples first elements of labels_1_0, labels_0_0
    labels_1_0 = labels_1_0[:n_group]
    labels_0_0 = labels_0_0[:n_group]
    # Get the n_samples last elements of labels_1_1, labels_0_1
    labels_1_1 = labels_1_1[-n_group:]
    labels_0_1 = labels_0_1[-n_group:]

    # Append the four data_1_0, data_1_1, data_0_0 and data_0_1 to a␣
↪single list
    data = data_1_0 + data_1_1 + data_0_0 + data_0_1
    labels = labels_1_0 + labels_1_1 + labels_0_0 + labels_0_1

    x_input = randn(latent_dim * n_samples)
    # reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)

    return [z_input, np.array(labels)], data


def generate_random_numbers(n_randoms, df_all_summary):
    random_numbers_class = {}
    for class_classificacio in df_all_summary['Classificació'].unique():
        values = df_all_summary[df_all_summary['Classificació']
                                == class_classificacio]['CHO (g)'].
↪tolist()
        # Round to 2 decimals
        values = [round(num, 2) for num in values]

        # Value count of values_F
```

```
        values_count = {}
        for i in values:
            if i in values_count:
                values_count[i] += 1
            else:
                values_count[i] = 1

        # Get the total frequency
        total_frequency = sum(values_count.values())
        # Normalize the frequencies to get probabilities
        probabilities = [v / total_frequency for v in values_count.
↪values()]
        # Generate random numbers following the distribution
        random_numbers_class[class_classificacio] = np.random.choice(
            list(values_count.keys()), size=n_randoms, p=probabilities)
    return random_numbers_class
```

A.4.5 Definition of the training process for the C-GAN

This process executes the training of the C-GAN architecture and saves on model each "output_image_step" epocs and an example of generated sequences with the model at that epoch

[16]:
```
if len(categorical_variable_list) > 0:


    # train the generator and discriminator
    def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs,␣
↪n_batch, output_image_step):
        bat_per_epo = int(dataset[0].shape[0] / n_batch)
        half_batch = int(n_batch / 2)
        loss_d_1 = []
        loss_d_2 = []
        loss_g = []
        g_loss = 0
```

```
        [z_input_samples, labels_samples], sample_images =␣
↪generate_latent_points_based_on_real(
            16, dataset)
        labels_samples_scaled = labels_samples.copy()
        # labels_samples_scaled[:, -1:] = np.round(real_value(
        #     labels_samples_scaled[:, -1:], min_max_grouped.
↪transpose()[0]), 1)

        # manually enumerate epochs
        for i in range(n_epochs):
            # enumerate batches over the training set
            for j in range(bat_per_epo):
                # get randomly selected 'real' samples
                [X_real, labels_real], y_real = generate_real_samples(
                    dataset, half_batch)
                # update discriminator model weights
                d_loss1, _ = d_model.train_on_batch(
                    [X_real, labels_real], y_real)
                # generate 'fake' examples
                [X_fake, labels], y_fake = generate_fake_samples(
                    g_model, latent_dim, half_batch, real_data)
                # update discriminator model weights
                d_loss2, _ = d_model.train_on_batch([X_fake, labels],␣
↪y_fake)
                # prepare points in latent space as input for the␣
↪generator
                [z_input, labels_input] = generate_latent_points(
                    latent_dim, n_batch, real_data,␣
↪numerical_variable_list, categorical_variable_list)
                # create inverted labels for the fake samples
                y_gan = ones((n_batch, 1))
                # update the generator via the discriminator's error
                z_input = z_input.astype('float32')
```

```
                labels_input = labels_input.astype('float32')
                y_gan = y_gan.astype('float32')
                if i % 1 == 0:
                    g_loss = gan_model.train_on_batch(
                        [z_input, labels_input], y_gan)
                # summarize loss on this batch
            print('>%d, %d/%d, d1=%.3f, d2=%.3f g=%.3f' %
                  (i+1, j+1, bat_per_epo, d_loss1, d_loss2, g_loss))

            loss_d_1.append(d_loss1)
            loss_d_2.append(d_loss2)
            loss_g.append(g_loss)

            # save the generator model
            plt.plot(loss_d_1)
            plt.plot(loss_d_2)
            plt.plot(loss_g)
            plt.legend(['Discriminator loss 1',
                        'Discriminator loss 2',  'Generator loss'])
            plt.show()
            plt.close()
            if (i) % output_image_step == 0:
                # predict outputs
                X_fake_samples = g_model.predict(
                    [z_input_samples, labels_samples], verbose=0)

                unique_labels = np.unique(labels_samples_scaled[:, :4],␣
↪axis=0)
                colors_dark = ['darkred', 'darkblue', 'darkgreen',␣
↪'saddlebrown', 'darkcyan',
                               'darkmagenta', 'darkgoldenrod',␣
↪'darkslategray', 'darkturquoise', 'darkviolet']
                colors = ['lightcoral', 'cornflowerblue', 'limegreen',␣
↪'sandybrown',
```

```
                        'paleturquoise', 'violet', 'gold',␣
↪'lightsteelblue', 'mediumturquoise', 'plum']
                colors_dark = colors_dark[:len(unique_labels)]
                colors = colors[:len(unique_labels)]

                # Plot the 16 images in a 4*5 grid with the trainy[i] as␣
↪the subtitle and i as the title of the figure
                fig, axs = plt.subplots(4, 4, figsize=(15, 15))
                for a in range(4):
                    for b in range(4):
                        # Find the index of unique_labels that matches␣
↪labels_samples[a*4+b]
                        index = np.where(
                            (unique_labels[:, :-1] ==␣
↪labels_samples_scaled[a*4+b, :-1]).all(axis=1))[0][0]
                        axs[a, b].plot(
                            X_fake_samples[a*4+b],␣
↪color=colors_dark[index], label='Generated')
                        axs[a, b].plot(sample_images[a*4+b],
                                       color=colors[index], label='Real')
                        axs[a, b].legend()
                        axs[a, b].
↪set_title(str(labels_samples_scaled[a*4+b]))
                        # meke the y scale go from 0 to 200 for all plots
                        axs[a, b].set_ylim([0, 860])
                # Set the title of the figure as the epoch number
                fig.suptitle('Epoch ' + str(i))
                fig.tight_layout()
                # Save at the folder in the CWD, models, subfolder str(i)
                path = os.path.join(CWD, 'models', str(i))
                # Create the folder if it doesn't exist
                if not os.path.exists(path):
                    os.makedirs(path)
```

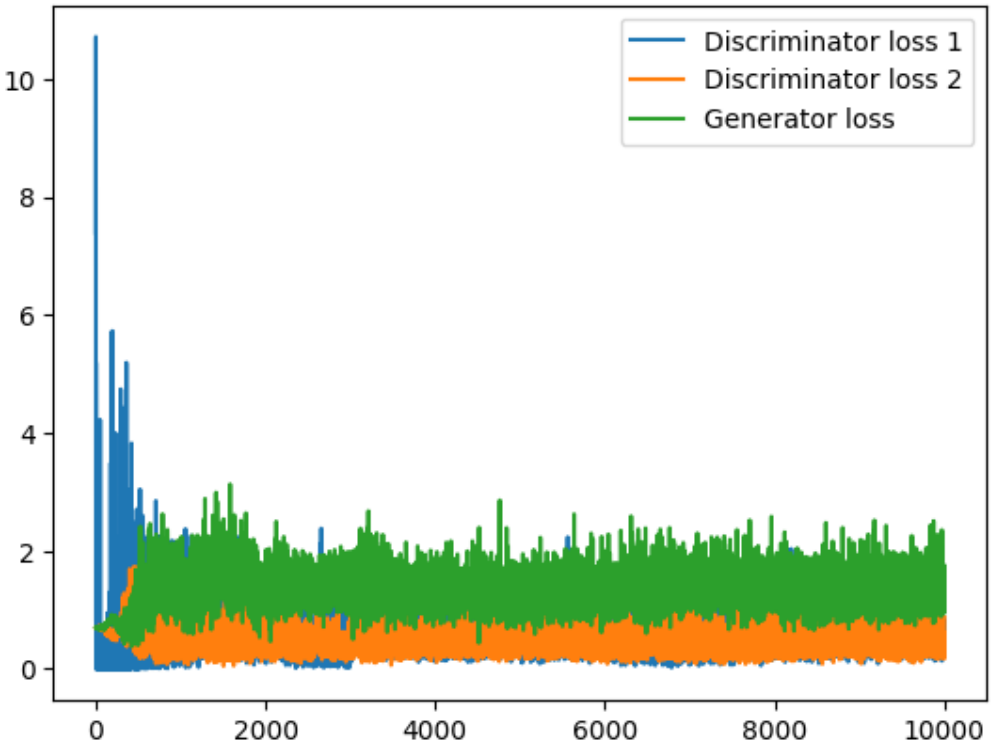
```
                plt.savefig(os.path.join(path, 'generated.png'))
                g_model.save(os.path.join(path, 'cgan_generator.h5'))
                plt.close()
            # Clear cell output
            clear_output(wait=True)
```

A.4.6 Run the C-GAN training with the desired parameters

[38]:
```
if len(categorical_variable_list) > 0:


    n_epochs = 10000
    output_image_step = 50
    n_batch = 8


    n_num = len(numerical_variable_list)
    n_classes = len(real_data['Classificació'].unique())


    # Definde optimizer
    opt = RMSprop(lr=0.00001)
    loss = 'binary_crossentropy'


    output_shape = 84
    # size of the latent space
    latent_dim = 200
    # create the discriminator
    d_model = define_discriminator(output_shape, n_classes, n_num, opt,␣
↪loss)
    # create the generator
    g_model = define_generator(latent_dim, n_classes, n_num, output_shape)
    # create the gan
    gan_model = define_gan(g_model, d_model, opt, loss)
    # load image data
    dataset = load_real_samples(trainX, trainy)
    # train model
```

```
train(g_model, d_model, gan_model, dataset,
      latent_dim, n_epochs, n_batch, output_image_step)
```

```
>10000, 6/6, d1=0.723, d2=0.877 g=1.108
```



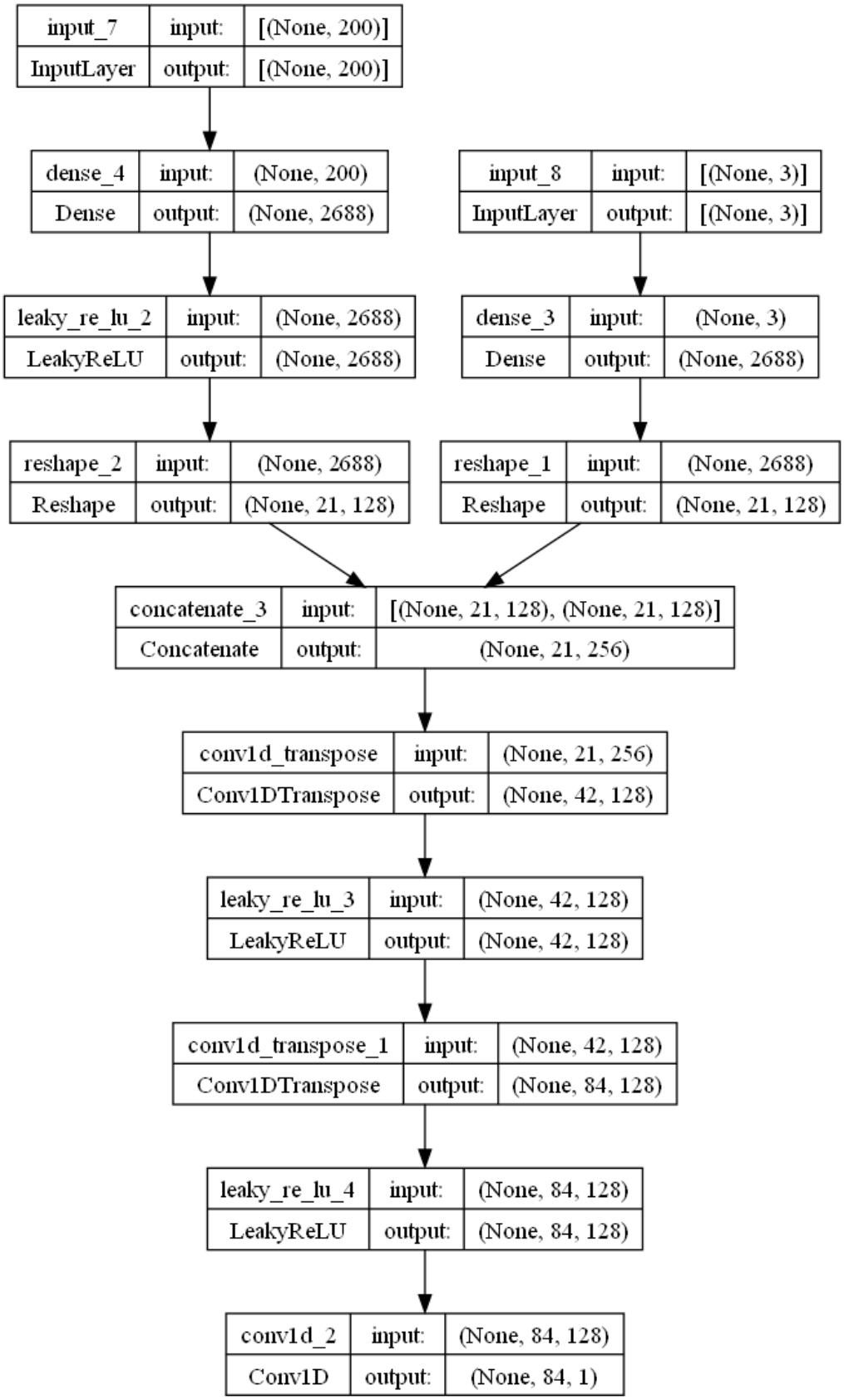
A.5 Representation and experimentation

A.5.1 Visualization

Structure of the diferent architectures involved in the C-GAN

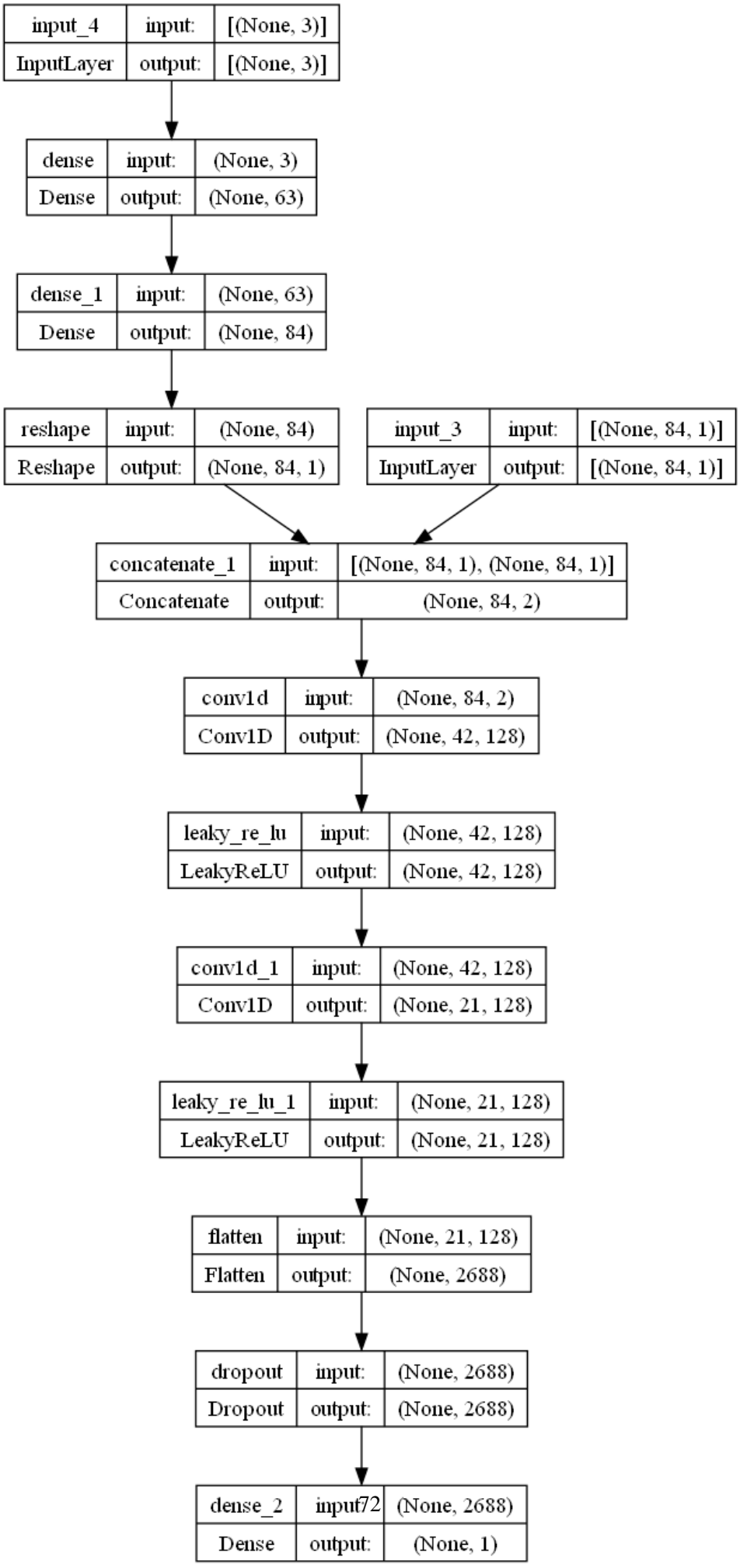
[18]:
```
# print the structure of g_model using graphviz
plot_model(g_model, to_file='g_model_plot.png',
           show_shapes=True, show_layer_names=True)
```

[18]:


input_7 | input: | [(None, 200)]
InputLayer | output: | [(None, 200)]
dense_4 | input: | (None, 200)
Dense | output: | (None, 2688)
input_8 | input: | [(None, 3)]
InputLayer | output: | [(None, 3)]
leaky_re_lu_2 | input: | (None, 2688)
LeakyReLU | output: | (None, 2688)
dense_3 | input: | (None, 3)
Dense | output: | (None, 2688)
reshape_2 | input: | (None, 2688)
Reshape | output: | (None, 21, 128)
reshape_1 | input: | (None, 2688)
Reshape | output: | (None, 21, 128)
concatenate_3 | input: | [(None, 21, 128), (None, 21, 128)]
Concatenate | output: | (None, 21, 256)
conv1d_transpose | input: | (None, 21, 256)
Conv1DTranspose | output: | (None, 42, 128)
leaky_re_lu_3 | input: | (None, 42, 128)
LeakyReLU | output: | (None, 42, 128)
conv1d_transpose_1 | input: | (None, 42, 128)
Conv1DTranspose | output: | (None, 84, 128)
leaky_re_lu_4 | input: | (None, 84, 128)
LeakyReLU | output: | (None, 84, 128)
conv1d_2 | input: | (None, 84, 128)
Conv1D | output: | (None, 84, 1)

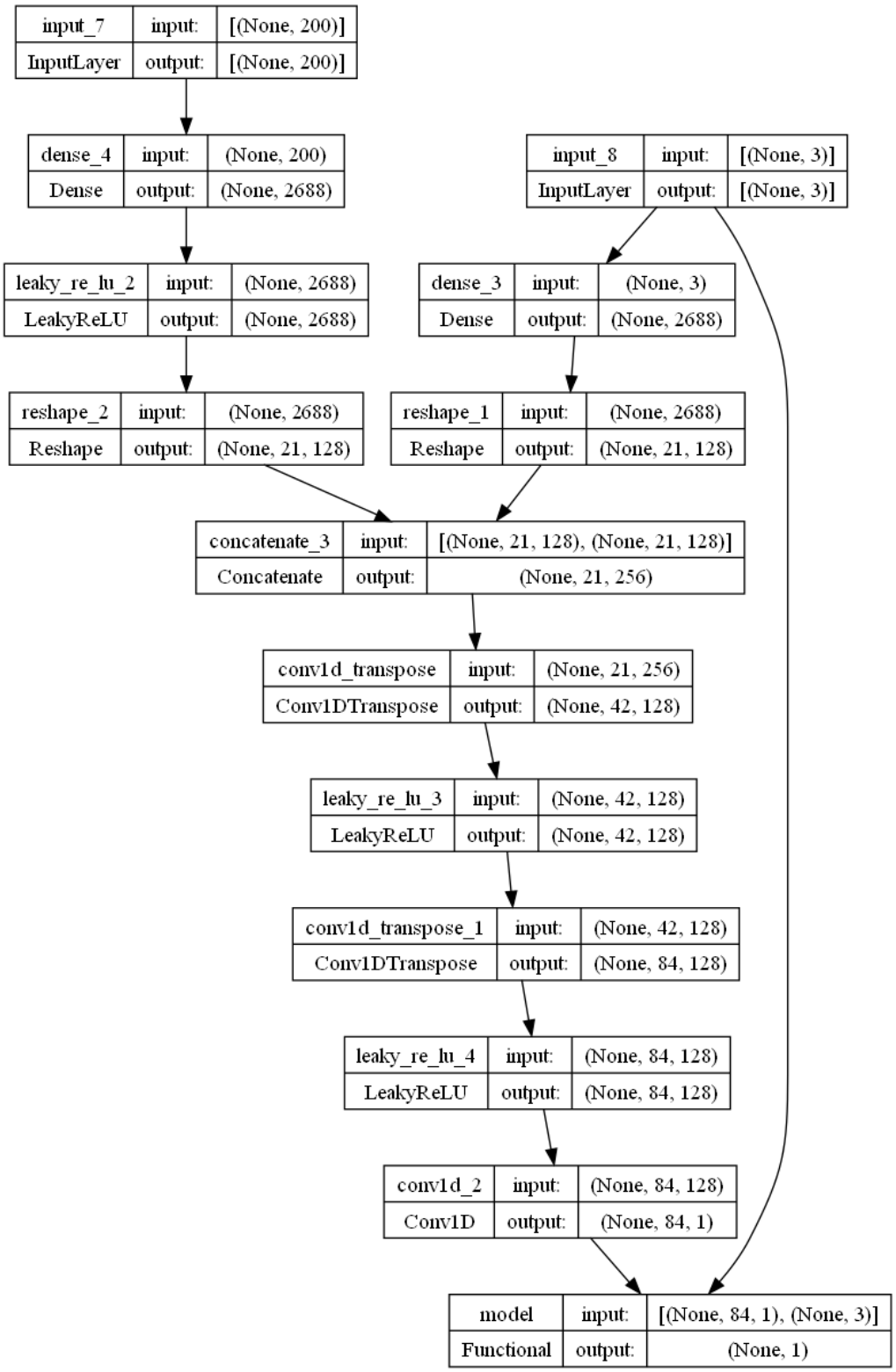
[19]:
```
# print the structure of d_model using graphviz
plot_model(d_model, to_file='g_model_plot.png',
           show_shapes=True, show_layer_names=True)
```

[19]:


input_4 | input: | [(None, 3)]
InputLayer | output: | [(None, 3)]
dense | input: | (None, 3)
Dense | output: | (None, 63)
dense_1 | input: | (None, 63)
Dense | output: | (None, 84)
reshape | input: | (None, 84)
Reshape | output: | (None, 84, 1)
input_3 | input: | [(None, 84, 1)]
InputLayer | output: | [(None, 84, 1)]
concatenate_1 | input: | [(None, 84, 1), (None, 84, 1)]
Concatenate | output: | (None, 84, 2)
conv1d | input: | (None, 84, 2)
Conv1D | output: | (None, 42, 128)
leaky_re_lu | input: | (None, 42, 128)
LeakyReLU | output: | (None, 42, 128)
conv1d_1 | input: | (None, 42, 128)
Conv1D | output: | (None, 21, 128)
leaky_re_lu_1 | input: | (None, 21, 128)
LeakyReLU | output: | (None, 21, 128)
flatten | input: | (None, 21, 128)
Flatten | output: | (None, 2688)
dropout | input: | (None, 2688)
Dropout | output: | (None, 2688)
dense_2 | input72 | (None, 2688)
Dense | output: | (None, 1)

[20]:
```
# print the structure of c-gan using graphviz
plot_model(gan_model, to_file='g_model_plot.png',
           show_shapes=True, show_layer_names=True)
```

[20]:


input_7 | input: | [(None, 200)]
InputLayer | output: | [(None, 200)]
dense_4 | input: | (None, 200)
Dense | output: | (None, 2688)
input_8 | input: | [(None, 3)]
InputLayer | output: | [(None, 3)]
leaky_re_lu_2 | input: | (None, 2688)
LeakyReLU | output: | (None, 2688)
dense_3 | input: | (None, 3)
Dense | output: | (None, 2688)
reshape_2 | input: | (None, 2688)
Reshape | output: | (None, 21, 128)
reshape_1 | input: | (None, 2688)
Reshape | output: | (None, 21, 128)
concatenate_3 | input: | [(None, 21, 128), (None, 21, 128)]
Concatenate | output: | (None, 21, 256)
conv1d_transpose | input: | (None, 21, 256)
Conv1DTranspose | output: | (None, 42, 128)
leaky_re_lu_3 | input: | (None, 42, 128)
LeakyReLU | output: | (None, 42, 128)
conv1d_transpose_1 | input: | (None, 42, 128)
Conv1DTranspose | output: | (None, 84, 128)
leaky_re_lu_4 | input: | (None, 84, 128)
LeakyReLU | output: | (None, 84, 128)
conv1d_2 | input: | (None, 84, 128)
Conv1D | output: | (None, 84, 1)
model | input: | [(None, 84, 1), (None, 3)]
Functional | output: | (None, 1)

GIF to show the training process and how the RA curves are generated

[21]:
```
n_epochs = 10000
output_image_step = 50
n_batch = 8


n_num = len(numerical_variable_list)
n_classes = len(real_data['Classificació'].unique())


# Definde optimizer
opt = RMSprop(lr=0.00001)
output_shape = 84
# size of the latent space
latent_dim = 200


# Create a gif named gan.gif for all the images in the models folder
# Use the folder of each frame as the
images = []
for epoch in range(0, n_epochs, output_image_step):
    images.append(imageio.imread(os.path.join(
        CWD, 'models', str(epoch), 'generated.png')))
imageio.mimsave(os.path.join(CWD, 'gan.gif'), images, duration=0.5)
```

Comparison of the real and generated samples with the generator from epoch 9950 and the same input labels

Loading the model and generating synthetic samples

[91]:
```
latent_dim = 200
dataset = load_real_samples(trainX, trainy)


g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
 ↪h5'))
[z_input_samples, labels_samples], sample_images =␣
 ↪generate_latent_points_based_on_real(
    16, dataset)
```

```
# Combine rows of min_max to keep the minor and maximum of each variable
min_max_grouped = pd.DataFrame(columns=['min', 'max'])
min_max_grouped['min'] = [min_max['CHO (g)']['min'].min()]
min_max_grouped['max'] = [min_max['CHO (g)']['max'].max()]
min_max_grouped = min_max_grouped.transpose()

labels_samples_scaled = labels_samples.copy()
labels_samples[:, 2] = np.round(real_value(
    labels_samples_scaled[:, 2], min_max_grouped[0]), 1)

fake_samples = g_model.predict(
    [z_input_samples, labels_samples_scaled], verbose=0)

# Plot the 16 images in a 4*5 grid with the trainy[i] as the subtitle and␣
 ↪i as the title of the figure
fig, axs = plt.subplots(4, 4, figsize=(15, 15))
for a in range(4):
    for b in range(4):
        if labels_samples[a*4+b][1] == 0.0:
            axs[a, b].plot(fake_samples[a*4+b],
                           color='darkred', label='Generated')
            axs[a, b].plot(sample_images[a*4+b],
                           color='lightcoral', label='Real')
            axs[a, b].legend()
            axs[a, b].set_title('Fast, ' + str(labels_samples[a*4+b][2]))
        else:
            axs[a, b].plot(fake_samples[a*4+b],
                           color='darkblue', label='Generated')
            axs[a, b].plot(sample_images[a*4+b],
                           color='cornflowerblue', label='Real')
            axs[a, b].legend()
            axs[a, b].set_title('Non Fast, ' +␣
 ↪str(labels_samples[a*4+b][2]))
```

```
        # meke the y scale go from 0 to 200 for all plots
        axs[a, b].set_ylim([0, 860])
```

WARNING:tensorflow:No training configuration found in the save file, so the model was *not* compiled. Compile it manually.

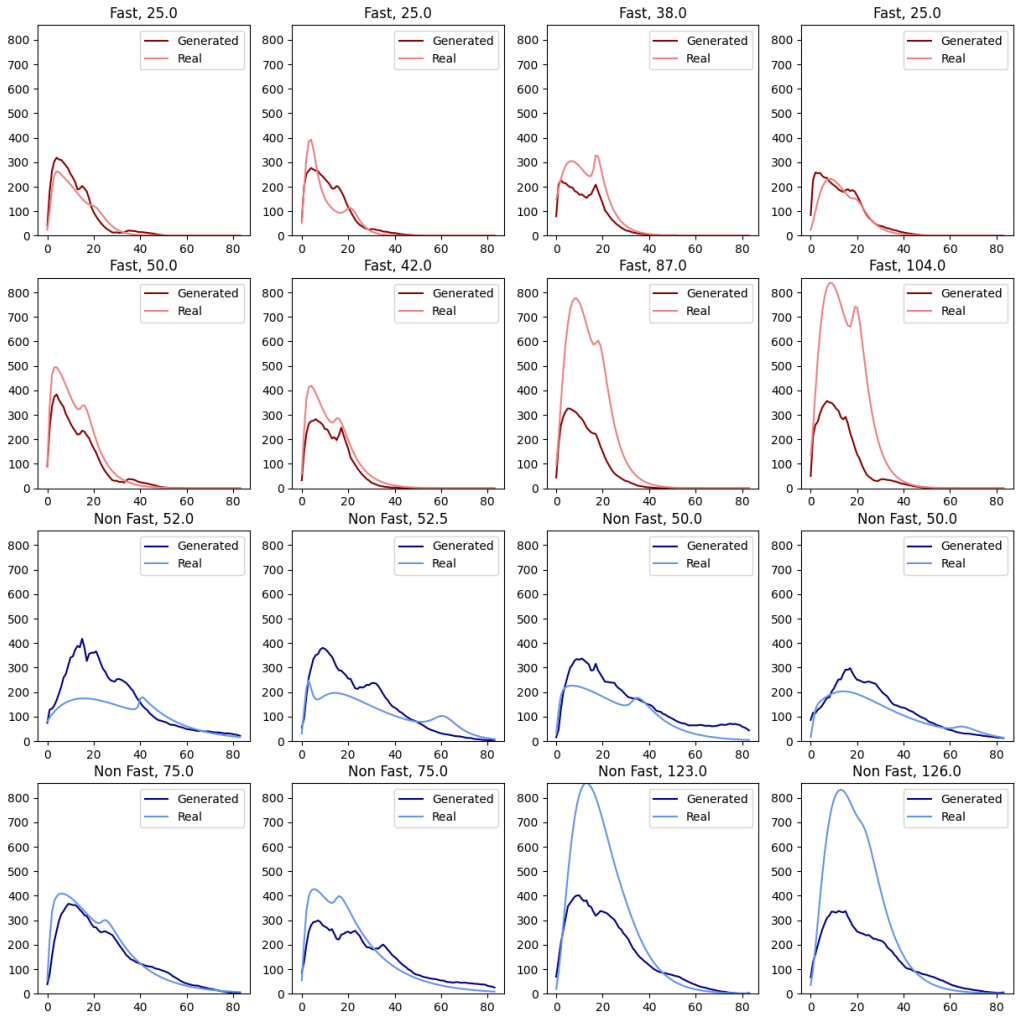


Image showing the generation of synthetic RA curves using the CHO variable as a category and fixed latent dimension

Here the process is:

- Generate a series of 5 groups of carbohydrate (CHO) values between the minimum (10) and the maximum (126). For example, group 1 includes

values approximately between 10 and 30, and so on for the successive groups. The legend displays the values that CHO adopts for each subgroup.

- Create an array of 5 labels like [0, 1, CHO_value_nth_group] and [1, 0, CHO_value_nth_group], both concatenated into a single array.
- Build a ‘‘z_input,’’ which represents the latent dimension of the model.

I have generated predictions of 5 samples per class (F, NF) for each CHO subgroup.

The plot illustrates that if the z_input remains the same, the generated curve remains nearly identical. The only noticeable difference is that the generated curves have slightly higher values as the CHO input increases.

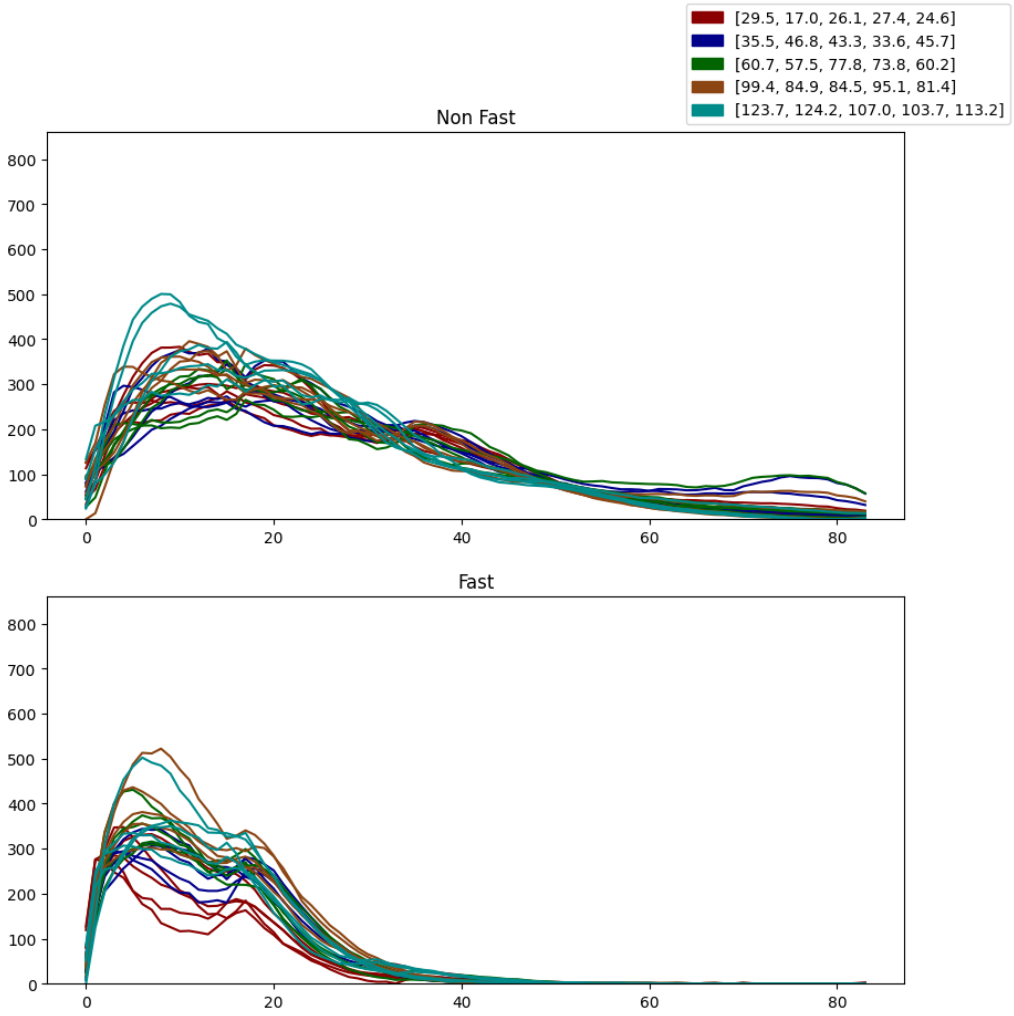
I’ve included the plot details using inputs only from the last CHO subgroup.

[92]:
```
# Generate 10 random numbers between min_max['CHO␣
↪(g)'][1]+i*(min_max['CHO (g)'][1]-min_max['CHO (g)'][0])/min_max['CHO␣
↪(g)'][0]
n_groups = 5
random_dict = {}
n_samples = 5

g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
↪h5'))

for i in range(n_groups):
    random_dict[i] = np.random.uniform(
        0+i*(1-0)/n_groups, 0+(i+1)*(1-0)/n_groups, n_samples).tolist()

# Scale the random numbers
random_dict_scaled = {}
for i in range(n_groups):
    random_dict_scaled[i] = np.round(real_value(
```

```
        np.array(random_dict[i]), min_max_grouped[0]), 1).tolist()


colors_dark = ['darkred', 'darkblue', 'darkgreen', 'saddlebrown',␣
 ↪'darkcyan',
              'darkmagenta', 'darkgoldenrod', 'darkslategray',␣
 ↪'darkturquoise', 'darkviolet']
colors = ['lightcoral', 'cornflowerblue', 'limegreen', 'sandybrown',
          'paleturquoise', 'violet', 'gold', 'lightsteelblue',␣
 ↪'mediumturquoise', 'plum']



fig, axs = plt.subplots(2, figsize=(10, 10))

for i in range(n_groups):
    x_input = randn(latent_dim * n_samples*2)
# reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples*2, latent_dim)
    labels_samples = np.array([[0, 1, i] for i in random_dict[i]])
    labels_samples = np.concatenate(
        (labels_samples, np.array([[1, 0, i] for i in random_dict[i]])),␣
 ↪axis=0)
    PREDICTED = g_model.predict([z_input, labels_samples], verbose=0)
    # Plot the first 10 images in a plot and the last 10 in another plot
    for j in range(n_samples):
        axs[0].set_title('Non Fast')
        axs[0].plot(PREDICTED[j], color=colors_dark[i])
        axs[1].set_title('Fast')
        axs[1].plot(PREDICTED[j+n_samples], color=colors_dark[i])

        # axs[1].set_title(str(random_dict_scaled[i][j]))
axs[0].set_ylim([0, 860])
axs[1].set_ylim([0, 860])
```

```
# range(0, 10) to list of strings
labels = [str(i) for i in range(0, 10)]
# Add a regend with the colors and the i values
fig.legend(handles=[mpatches.Patch(color=colors_dark[i],
           label=random_dict_scaled[i]) for i in range(n_groups)])


fig.show()
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```



Generation of an animated image that represents the same problem than the one in the previous cell but using the CHO variable as a numeric.

[93]:
```
n_samples = 500
latent_dim = 200


random_numbers_class = generate_random_numbers(n_samples, df_all_summary)
min_max_classes = df_all_summary.groupby(
    'Classificació').agg({'CHO (g)': ['min', 'max']})


# Create an array of len n_samples with [1, 0, random number between min␣
 ↪and max of CHO (g) F]
# Create an array of len n_samples with [0, 1, random number between min␣
 ↪and max of CHO (g) NF]
labels_F = np.zeros((n_samples, 3))
labels_F[:, 0] = 1
# labels_F[:, 2] = random_numbers_class['F']
labels_F[:, 2] = np.random.uniform(min_max_classes.loc['Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Fast', ('CHO (g)', 'max')],␣
 ↪n_samples)


labels_NF = np.zeros((n_samples, 3))
labels_NF[:, 1] = 1
# labels_NF[:, 2] = random_numbers_class['NF']
labels_NF[:, 2] = np.random.uniform(min_max_classes.loc['Non Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Non Fast', ('CHO (g)',␣
 ↪'max')], n_samples)


# Sort labels_F[:,-1]
labels_F = labels_F[labels_F[:, -1].argsort()]
# Sort labels_NF[:,-1]
labels_NF = labels_NF[labels_NF[:, -1].argsort()]


x_input = randn(latent_dim * n_samples)
```

```
# reshape into a batch of inputs for the network
z_input = x_input.reshape(n_samples, latent_dim)

# Generate fake samples
g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
 ↪h5'))
fake_samples_F = g_model.predict([z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([z_input, labels_NF], verbose=0)

labels_samples_scaled[:, -1:] = np.round(real_value(
    labels_samples_scaled[:, -1:], min_max_grouped[0]), 1)

labels_F_scaled = labels_F.copy()
labels_F_scaled[:, 2] = np.round(real_value(
    labels_F_scaled[:, 2], min_max_grouped[0]), 1)
labels_NF_scaled = labels_NF.copy()
labels_NF_scaled[:, 2] = np.round(real_value(
    labels_NF_scaled[:, 2], min_max_grouped[0]), 1)

if not os.path.exists(os.path.join(CWD, 'gif')):
    os.makedirs(os.path.join(CWD, 'gif'))
# Plot each fake sample in an axis

colors_F = plt.cm.viridis(labels_F[:, -1])
colors_NF = plt.cm.viridis(labels_NF[:, -1])
scalar_map_F = plt.cm.ScalarMappable(cmap='viridis')
scalar_map_F.set_array(labels_F[:, -1])

scalar_map_NF = plt.cm.ScalarMappable(cmap='turbo')
scalar_map_NF.set_array(labels_NF[:, -1])
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

[95]:
```
df_all_summary['CHO (g)'].min()
```

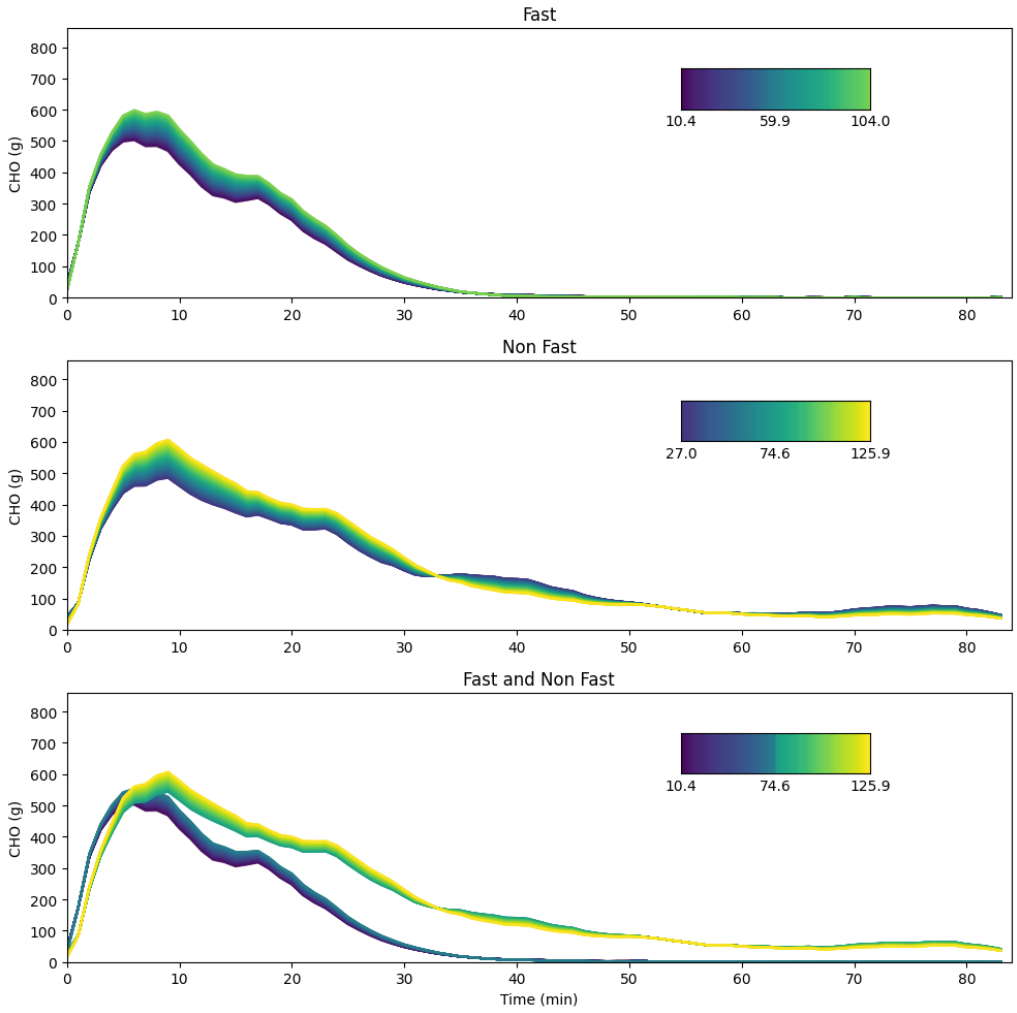
[95]:
```
0.0
```

By fixing the latent dimension and using half labels fast and half labels Non Fast we see the influence of the categories and that the amount of CHO produces a small efect in the actual output of the model

[116]:
```
z_input[:] = z_input[0]
# Drop half of the values in labels_F and append half of the values of␣
 ↪labels_NF
labels_meged = np.concatenate((labels_F[:250], labels_NF[250:]))
colors_merged = np.concatenate((colors_F[:250], colors_NF[250:]))

fake_samples_merged = g_model.predict([z_input, labels_meged], verbose=0)
fake_samples_F = g_model.predict([z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([z_input, labels_NF], verbose=0)

labels_F_scaled = labels_F.copy()
labels_F_scaled[:, 2] = np.round(real_value(
    labels_F_scaled[:, 2], min_max_grouped[0]), 1)
labels_NF_scaled = labels_NF.copy()
labels_NF_scaled[:, 2] = np.round(real_value(
    labels_NF_scaled[:, 2], min_max_grouped[0]), 1)
labels_merged_scaled = labels_meged.copy()
labels_merged_scaled[:, 2] = np.round(real_value(
    labels_merged_scaled[:, 2], min_max_grouped[0]), 1)

selected_indices = [0, len(colors_F) // 2, -1]
selected_numbers_F = [labels_F_scaled[:, 2][i] for i in selected_indices]
selected_numbers_NF = [labels_NF_scaled[:, 2][i] for i in␣
 ↪selected_indices]
selected_numbers_merged = [labels_merged_scaled[:, 2][i]
                           for i in selected_indices]
fig, axs = plt.subplots(3, figsize=(10, 10))
```

```
# Create a gradient using the RGB values
gradient_F = np.linspace(0, 1, len(colors_F)).reshape(-1, 1)
colors_F_array = np.array(colors_F).reshape(1, -1, 4)
legend_ax_F = axs[0].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_F.imshow(colors_F_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_F.set_xticks([])
legend_ax_F.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_F):
    legend_ax_F.text(i / 2, -0.4, str(num),
                     transform=legend_ax_F.transAxes, ha='center')

gradient_NF = np.linspace(0, 1, len(colors_NF)).reshape(-1, 1)
colors_NF_array = np.array(colors_NF).reshape(1, -1, 4)
legend_ax_NF = axs[1].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_NF.imshow(colors_NF_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_NF.set_xticks([])
legend_ax_NF.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_NF):
    legend_ax_NF.text(i / 2, -0.40, str(num),
                      transform=legend_ax_NF.transAxes, ha='center')
gradient_merged = np.linspace(0, 1, len(colors_merged)).reshape(-1, 1)
colors_merged_array = np.array(colors_merged).reshape(1, -1, 4)
legend_ax_merged = axs[2].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_merged.imshow(colors_merged_array,
                        aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
```

```
legend_ax_merged.set_xticks([])
legend_ax_merged.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_merged):
    legend_ax_merged.text(i / 2, -0.4, str(num),
                          transform=legend_ax_merged.transAxes,␣
 ↪ha='center')
for i in range(len(labels_F)):
    axs[0].plot(fake_samples_F[i], color=colors_F[i], alpha=0.30)
    axs[1].plot(fake_samples_NF[i], color=colors_NF[i], alpha=0.30)
    axs[2].plot(fake_samples_merged[i], color=colors_merged[i], alpha=0.
 ↪30)

axs[0].set_title('Fast')
axs[1].set_title('Non Fast')
axs[2].set_title('Fast and Non Fast')
axs[0].set_ylim([0, 860])
axs[1].set_ylim([0, 860])
axs[2].set_ylim([0, 860])
axs[0].set_xlim([0, 84])
axs[1].set_xlim([0, 84])
axs[2].set_xlim([0, 84])
axs[0].set_ylabel('CHO (g)')
axs[1].set_ylabel('CHO (g)')
axs[2].set_ylabel('CHO (g)')
axs[2].set_xlabel('Time (min)')

# Show the plot
plt.tight_layout()
plt.show()
```



[117]:
```
import numpy as np



def modify_array(input_array, modification_percentage=0.20,␣
↪value_range=(0, 0.5)):
    modified_array = input_array.copy()
    num_elements_to_modify = int(modification_percentage *␣
↪len(input_array))

    for _ in range(num_elements_to_modify):
        index = np.random.randint(0, len(input_array))
        modification_value = np.random.uniform(
```

```
            value_range[0], value_range[1]) * input_array[index]
        modification_direction = np.random.choice(
            [-1, 1])  # -1 for subtraction, 1 for addition

        modified_array[index] += modification_direction *␣
↪modification_value

    return modified_array
```

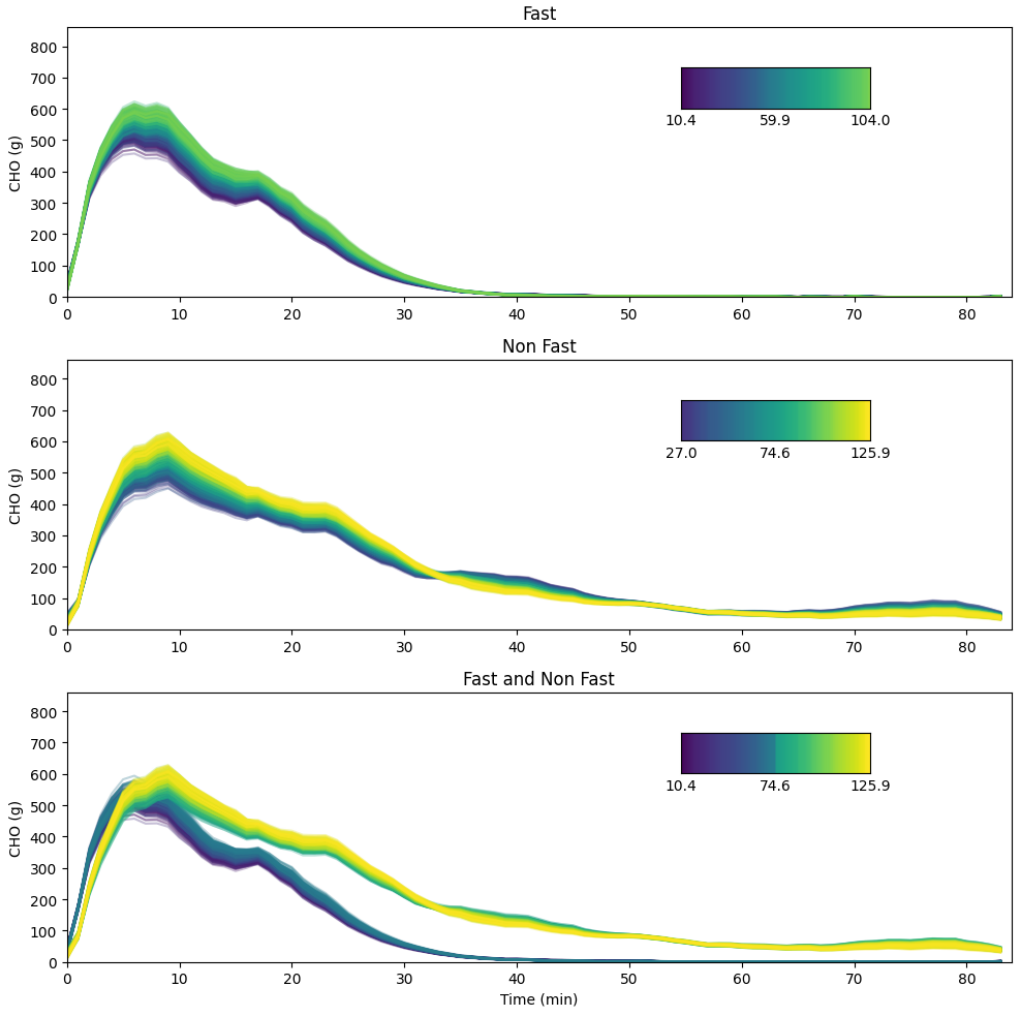
Modifying a 25% of the values in z_input between a 0 and 50% of the value itself

[119]:
```
# Modifying the array as described
modified_z_input = [modify_array(input_array, modification_percentage=0.
↪25, value_range=(
    0, 0.5)) for input_array in z_input]
modified_z_input = np.concatenate(modified_z_input).reshape(z_input.shape)

# Drop half of the values in labels_F and append half of the values of␣
↪labels_NF
labels_meged = np.concatenate((labels_F[:250], labels_NF[250:]))
colors_merged = np.concatenate((colors_F[:250], colors_NF[250:]))

fake_samples_merged = g_model.predict(
    [modified_z_input, labels_meged], verbose=0)
fake_samples_F = g_model.predict([modified_z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([modified_z_input, labels_NF],␣
↪verbose=0)

labels_F_scaled = labels_F.copy()
labels_F_scaled[:, 2] = np.round(real_value(
    labels_F_scaled[:, 2], min_max_grouped[0]), 1)
labels_NF_scaled = labels_NF.copy()
labels_NF_scaled[:, 2] = np.round(real_value(
```

```
    labels_NF_scaled[:, 2], min_max_grouped[0]), 1)
labels_merged_scaled = labels_meged.copy()
labels_merged_scaled[:, 2] = np.round(real_value(
    labels_merged_scaled[:, 2], min_max_grouped[0]), 1)

selected_indices = [0, len(colors_F) // 2, -1]
selected_numbers_F = [labels_F_scaled[:, 2][i] for i in selected_indices]
selected_numbers_NF = [labels_NF_scaled[:, 2][i] for i in␣
 ↪selected_indices]
selected_numbers_merged = [labels_merged_scaled[:, 2][i]
                           for i in selected_indices]
fig, axs = plt.subplots(3, figsize=(10, 10))

# Create a gradient using the RGB values
gradient_F = np.linspace(0, 1, len(colors_F)).reshape(-1, 1)
colors_F_array = np.array(colors_F).reshape(1, -1, 4)
legend_ax_F = axs[0].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_F.imshow(colors_F_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_F.set_xticks([])
legend_ax_F.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_F):
    legend_ax_F.text(i / 2, -0.4, str(num),
                     transform=legend_ax_F.transAxes, ha='center')

gradient_NF = np.linspace(0, 1, len(colors_NF)).reshape(-1, 1)
colors_NF_array = np.array(colors_NF).reshape(1, -1, 4)
legend_ax_NF = axs[1].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_NF.imshow(colors_NF_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_NF.set_xticks([])
```

```
legend_ax_NF.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_NF):
    legend_ax_NF.text(i / 2, -0.40, str(num),
                      transform=legend_ax_NF.transAxes, ha='center')
gradient_merged = np.linspace(0, 1, len(colors_merged)).reshape(-1, 1)
colors_merged_array = np.array(colors_merged).reshape(1, -1, 4)
legend_ax_merged = axs[2].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_merged.imshow(colors_merged_array,
                        aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_merged.set_xticks([])
legend_ax_merged.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_merged):
    legend_ax_merged.text(i / 2, -0.4, str(num),
                          transform=legend_ax_merged.transAxes,␣
 ↪ha='center')
for i in range(len(labels_F)):
    axs[0].plot(fake_samples_F[i], color=colors_F[i], alpha=0.30)
    axs[1].plot(fake_samples_NF[i], color=colors_NF[i], alpha=0.30)
    axs[2].plot(fake_samples_merged[i], color=colors_merged[i], alpha=0.
 ↪30)


axs[0].set_title('Fast')
axs[1].set_title('Non Fast')
axs[2].set_title('Fast and Non Fast')
axs[0].set_ylim([0, 860])
axs[1].set_ylim([0, 860])
axs[2].set_ylim([0, 860])
axs[0].set_xlim([0, 84])
axs[1].set_xlim([0, 84])
axs[2].set_xlim([0, 84])
```

```
axs[0].set_ylabel('CHO (g)')
axs[1].set_ylabel('CHO (g)')
axs[2].set_ylabel('CHO (g)')
axs[2].set_xlabel('Time (min)')

# Show the plot
plt.tight_layout()
plt.show()
```

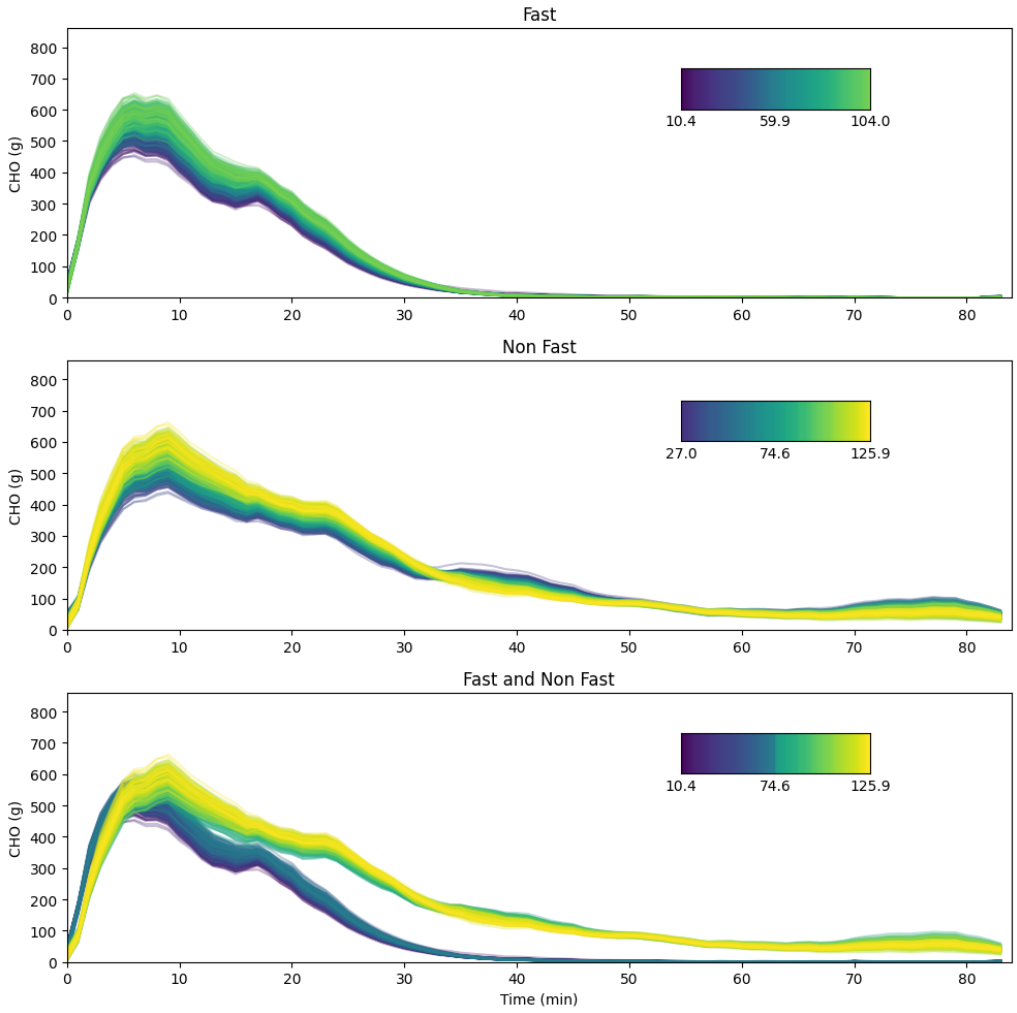


Modifying a 75% of the values in z_input between a 0 and 50% of the value itself

[120]:
```
# Modifying the array as described
modified_z_input = [modify_array(input_array, modification_percentage=0.
 ↪75, value_range=(
    0, 0.5)) for input_array in z_input]
modified_z_input = np.concatenate(modified_z_input).reshape(z_input.shape)

# Drop half of the values in labels_F and append half of the values of␣
 ↪labels_NF
labels_meged = np.concatenate((labels_F[:250], labels_NF[250:]))
colors_merged = np.concatenate((colors_F[:250], colors_NF[250:]))

fake_samples_merged = g_model.predict(
    [modified_z_input, labels_meged], verbose=0)
fake_samples_F = g_model.predict([modified_z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([modified_z_input, labels_NF],␣
 ↪verbose=0)

labels_F_scaled = labels_F.copy()
labels_F_scaled[:, 2] = np.round(real_value(
    labels_F_scaled[:, 2], min_max_grouped[0]), 1)
labels_NF_scaled = labels_NF.copy()
labels_NF_scaled[:, 2] = np.round(real_value(
    labels_NF_scaled[:, 2], min_max_grouped[0]), 1)
labels_merged_scaled = labels_meged.copy()
labels_merged_scaled[:, 2] = np.round(real_value(
    labels_merged_scaled[:, 2], min_max_grouped[0]), 1)

selected_indices = [0, len(colors_F) // 2, -1]
selected_numbers_F = [labels_F_scaled[:, 2][i] for i in selected_indices]
selected_numbers_NF = [labels_NF_scaled[:, 2][i] for i in␣
 ↪selected_indices]
selected_numbers_merged = [labels_merged_scaled[:, 2][i]
                           for i in selected_indices]
fig, axs = plt.subplots(3, figsize=(10, 10))
```

```
# Create a gradient using the RGB values
gradient_F = np.linspace(0, 1, len(colors_F)).reshape(-1, 1)
colors_F_array = np.array(colors_F).reshape(1, -1, 4)
legend_ax_F = axs[0].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_F.imshow(colors_F_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_F.set_xticks([])
legend_ax_F.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_F):
    legend_ax_F.text(i / 2, -0.4, str(num),
                     transform=legend_ax_F.transAxes, ha='center')

gradient_NF = np.linspace(0, 1, len(colors_NF)).reshape(-1, 1)
colors_NF_array = np.array(colors_NF).reshape(1, -1, 4)
legend_ax_NF = axs[1].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_NF.imshow(colors_NF_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_NF.set_xticks([])
legend_ax_NF.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_NF):
    legend_ax_NF.text(i / 2, -0.40, str(num),
                      transform=legend_ax_NF.transAxes, ha='center')
gradient_merged = np.linspace(0, 1, len(colors_merged)).reshape(-1, 1)
colors_merged_array = np.array(colors_merged).reshape(1, -1, 4)
legend_ax_merged = axs[2].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_merged.imshow(colors_merged_array,
                        aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
```

```
legend_ax_merged.set_xticks([])
legend_ax_merged.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_merged):
    legend_ax_merged.text(i / 2, -0.4, str(num),
                          transform=legend_ax_merged.transAxes,␣
 ↪ha='center')
for i in range(len(labels_F)):
    axs[0].plot(fake_samples_F[i], color=colors_F[i], alpha=0.30)
    axs[1].plot(fake_samples_NF[i], color=colors_NF[i], alpha=0.30)
    axs[2].plot(fake_samples_merged[i], color=colors_merged[i], alpha=0.
 ↪30)

axs[0].set_title('Fast')
axs[1].set_title('Non Fast')
axs[2].set_title('Fast and Non Fast')
axs[0].set_ylim([0, 860])
axs[1].set_ylim([0, 860])
axs[2].set_ylim([0, 860])
axs[0].set_xlim([0, 84])
axs[1].set_xlim([0, 84])
axs[2].set_xlim([0, 84])
axs[0].set_ylabel('CHO (g)')
axs[1].set_ylabel('CHO (g)')
axs[2].set_ylabel('CHO (g)')
axs[2].set_xlabel('Time (min)')

# Show the plot
plt.tight_layout()
plt.show()
```



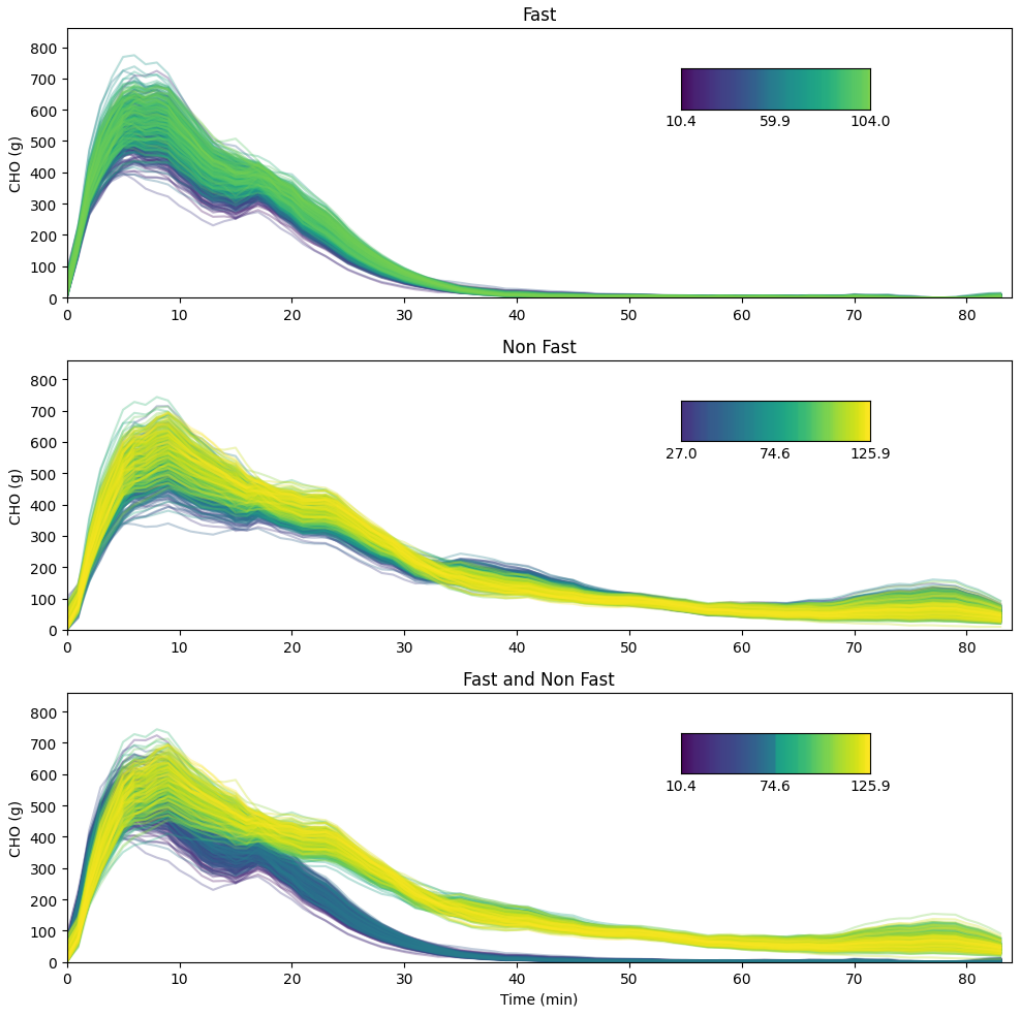
Modifying a 100% of the values in z_input between a 0 and 100% of the value itself

[123]:
```
# Modifying the array as described
modified_z_input = [modify_array(
    input_array, modification_percentage=1, value_range=(0, 1)) for␣
 ↪input_array in z_input]
modified_z_input = np.concatenate(modified_z_input).reshape(z_input.shape)

# Drop half of the values in labels_F and append half of the values of␣
 ↪labels_NF
labels_meged = np.concatenate((labels_F[:250], labels_NF[250:]))
```

```
colors_merged = np.concatenate((colors_F[:250], colors_NF[250:]))

fake_samples_merged = g_model.predict(
    [modified_z_input, labels_meged], verbose=0)
fake_samples_F = g_model.predict([modified_z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([modified_z_input, labels_NF],␣
 ↪verbose=0)

labels_F_scaled = labels_F.copy()
labels_F_scaled[:, 2] = np.round(real_value(
    labels_F_scaled[:, 2], min_max_grouped[0]), 1)
labels_NF_scaled = labels_NF.copy()
labels_NF_scaled[:, 2] = np.round(real_value(
    labels_NF_scaled[:, 2], min_max_grouped[0]), 1)
labels_merged_scaled = labels_meged.copy()
labels_merged_scaled[:, 2] = np.round(real_value(
    labels_merged_scaled[:, 2], min_max_grouped[0]), 1)

selected_indices = [0, len(colors_F) // 2, -1]
selected_numbers_F = [labels_F_scaled[:, 2][i] for i in selected_indices]
selected_numbers_NF = [labels_NF_scaled[:, 2][i] for i in␣
 ↪selected_indices]
selected_numbers_merged = [labels_merged_scaled[:, 2][i]
                           for i in selected_indices]
fig, axs = plt.subplots(3, figsize=(10, 10))

# Create a gradient using the RGB values
gradient_F = np.linspace(0, 1, len(colors_F)).reshape(-1, 1)
colors_F_array = np.array(colors_F).reshape(1, -1, 4)
legend_ax_F = axs[0].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_F.imshow(colors_F_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_F.set_xticks([])
```

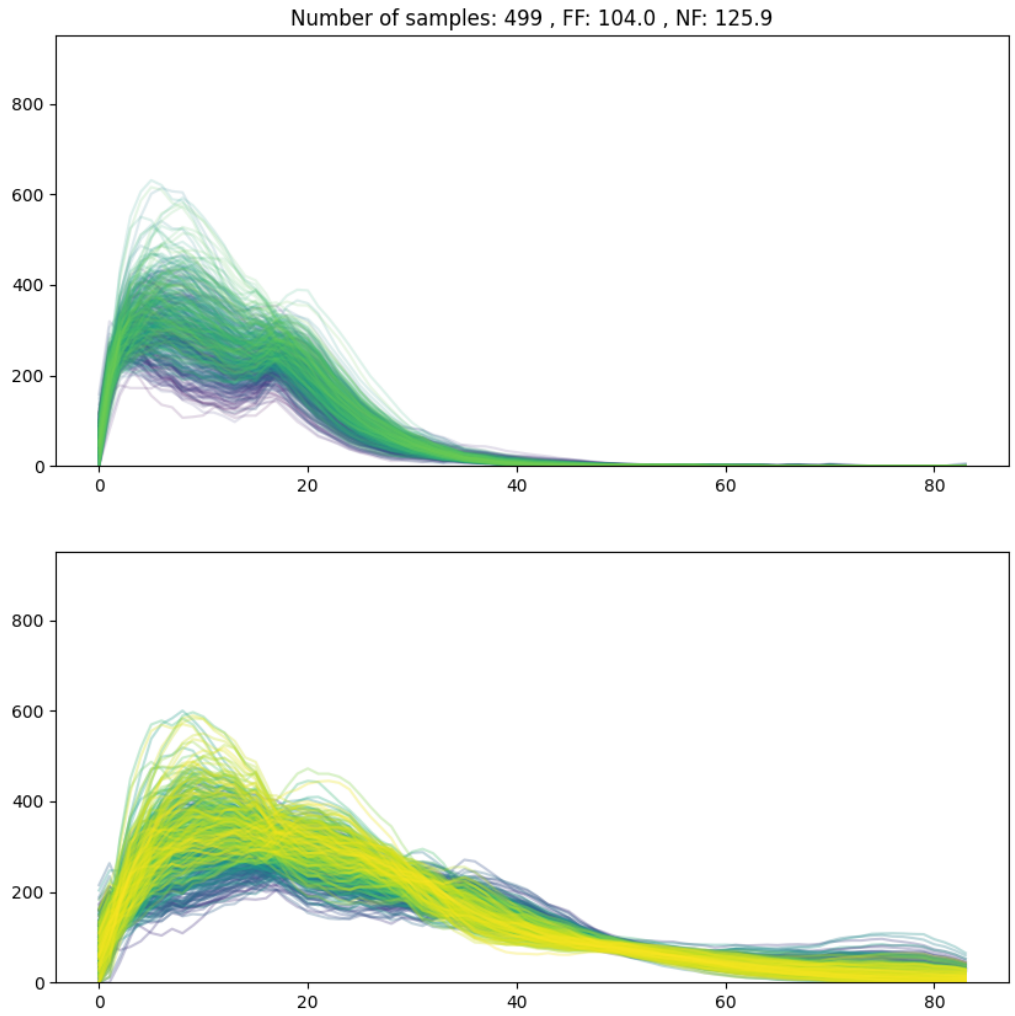
```
legend_ax_F.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_F):
    legend_ax_F.text(i / 2, -0.4, str(num),
                     transform=legend_ax_F.transAxes, ha='center')

gradient_NF = np.linspace(0, 1, len(colors_NF)).reshape(-1, 1)
colors_NF_array = np.array(colors_NF).reshape(1, -1, 4)
legend_ax_NF = axs[1].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_NF.imshow(colors_NF_array, aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_NF.set_xticks([])
legend_ax_NF.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_NF):
    legend_ax_NF.text(i / 2, -0.40, str(num),
                      transform=legend_ax_NF.transAxes, ha='center')
gradient_merged = np.linspace(0, 1, len(colors_merged)).reshape(-1, 1)
colors_merged_array = np.array(colors_merged).reshape(1, -1, 4)
legend_ax_merged = axs[2].inset_axes([0.65, 0.7, 0.2, 0.15])
# Display the gradient as a color scale legend
legend_ax_merged.imshow(colors_merged_array,
                        aspect='auto', extent=[0, 1, 0, 1])
# Display the gradient as a color scale legend
legend_ax_merged.set_xticks([])
legend_ax_merged.set_yticks([])
# Display the numbers associated with the legend
for i, num in enumerate(selected_numbers_merged):
    legend_ax_merged.text(i / 2, -0.4, str(num),
                          transform=legend_ax_merged.transAxes,␣
 ↪ha='center')
for i in range(len(labels_F)):
    axs[0].plot(fake_samples_F[i], color=colors_F[i], alpha=0.30)
```

```
    axs[1].plot(fake_samples_NF[i], color=colors_NF[i], alpha=0.30)
    axs[2].plot(fake_samples_merged[i], color=colors_merged[i], alpha=0.
↪30)

axs[0].set_title('Fast')
axs[1].set_title('Non Fast')
axs[2].set_title('Fast and Non Fast')
axs[0].set_ylim([0, 860])
axs[1].set_ylim([0, 860])
axs[2].set_ylim([0, 860])
axs[0].set_xlim([0, 84])
axs[1].set_xlim([0, 84])
axs[2].set_xlim([0, 84])
axs[0].set_ylabel('CHO (g)')
axs[1].set_ylabel('CHO (g)')
axs[2].set_ylabel('CHO (g)')
axs[2].set_xlabel('Time (min)')

# Show the plot
plt.tight_layout()
plt.show()
```



[131]:
```
plt.figure(figsize=(10, 10))
for i in range(0, n_samples):
    plt.subplot(2, 1, 1)
    plt.plot(fake_samples_F[i], color=colors_F[i], alpha=0.15)
    # Add labels_F[i] to the top right corner
    # plt.text(0.8, 0.9, str(round(labels_F[i, -1], 2)), fontsize=12,␣
↪transform=plt.gca().transAxes)
    plt.ylim(0, 950)
    plt.title('Number of samples: ' + str(i) + ' , FF: ' + str(round(
        labels_F_scaled[i, -1], 2)) + ' , NF: ' +␣
↪str(round(labels_NF_scaled[i, -1], 2)))
    plt.subplot(2, 1, 2)
```

```
    plt.plot(fake_samples_NF[i], color=colors_NF[i], alpha=0.3)
    # Add labels_NF[i] to the top right corner
    # plt.text(0.8, 0.9, str(round(labels_NF[i, -1], 2)), fontsize=12,␣
 ↪transform=plt.gca().transAxes)
    plt.ylim(0, 950)
    plt.savefig(os.path.join(CWD, 'gif', str(i) + '.png'))
images = []
for i in range(0, n_samples):
    images.append(imageio.imread(os.path.join(CWD, 'gif', str(i) + '.
 ↪png')))
imageio.mimsave(os.path.join(CWD, 'model_colorscale.gif'),
                images, duration=0.05)
```

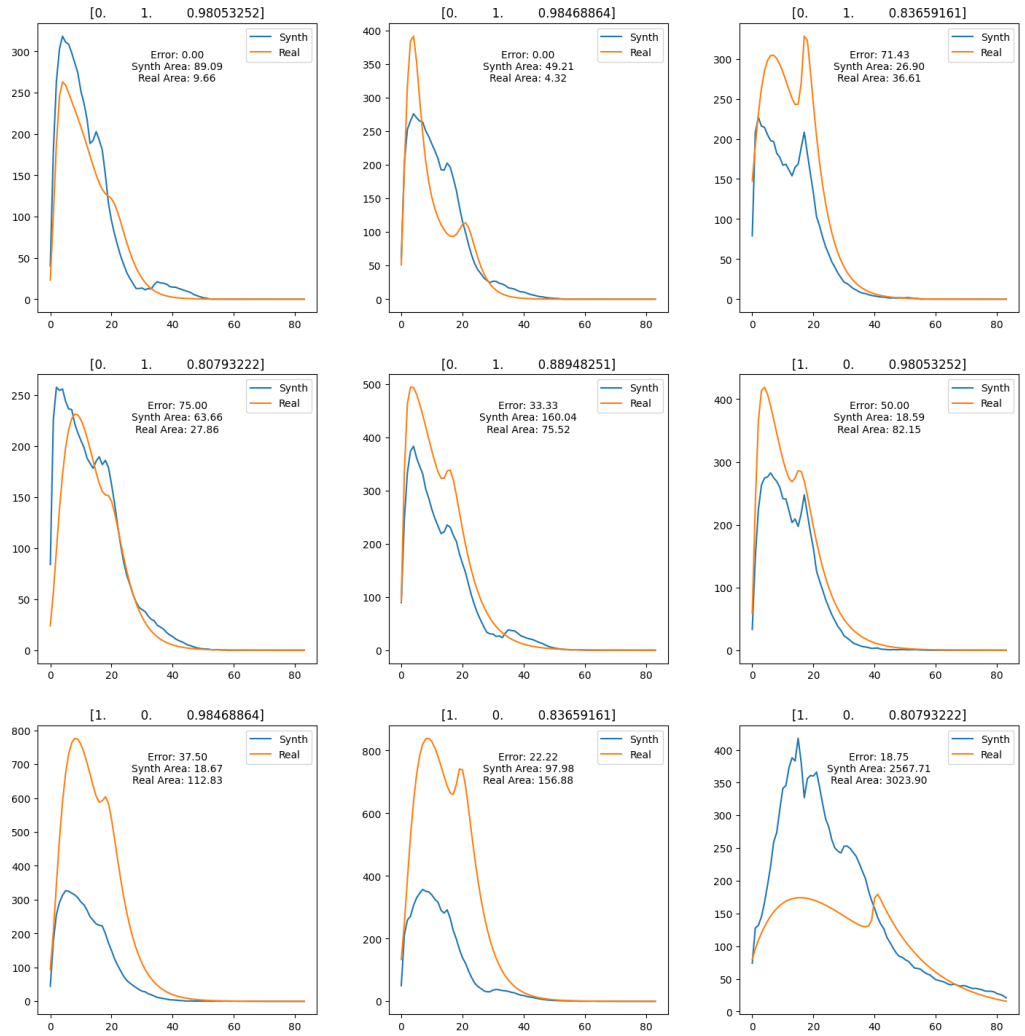


A.5.2 Calculation of the comparison of the results for the diferent metrics

Example of the error in the area between synthetic and real samples

[124]:
```
# Assuming you have the required data like fake_samples, sample_images,␣
 ↪labels_samples
fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.tight_layout(pad=4.0)

for A in range(9):
    fake_sample = fake_samples[A]
    fake_sample = fake_sample.reshape(84)
    real_sample = sample_images[A]
    real_sample = np.concatenate(real_sample).ravel()
    fake_mean = np.mean(fake_sample)
    fake_peaks, _ = find_peaks(fake_sample, height=0)
    fake_peaks = [x for x in fake_peaks if fake_sample[x] > fake_mean]
    real_mean = np.mean(real_sample)
    real_peaks, _ = find_peaks(real_sample, height=0)
    real_peaks = [x for x in real_peaks if real_sample[x] > real_mean]

    error = 100 * abs(fake_peaks[0] - real_peaks[0]) / real_peaks[0]
    fake_area = np.trapz(fake_sample[40:])
    real_area = np.trapz(real_sample[40:])
    error_area = 100 * abs(fake_area - real_area) / real_area

    row = A // 3
    col = A % 3

    axs[row, col].plot(fake_sample)
    axs[row, col].plot(real_sample)
    axs[row, col].set_title(labels_samples[A])
    axs[row, col].legend(['Synth', 'Real'])
```

```
    axs[row, col].text(0.5, 0.85, f'Error: {error:.2f}\nSynth Area:␣
↪{fake_area:.2f}\nReal Area: {real_area:.2f}',
                       horizontalalignment='center',␣
↪verticalalignment='center', transform=axs[row, col].transAxes)

plt.show()
```



Calculation of the JS and KL metrics comparing the samples between them for classes

[231]:
```
list_original_F = df_all_summary[df_all_summary['Classificació']
                                 == 'Fast']['RA'].to_list()
list_original_NF = df_all_summary[df_all_summary['Classificació']
                                  == 'Non Fast']['RA'].to_list()
list_original_F = list_original_F[:len(list_original_NF)]
list_original_NF = list_original_NF[:len(list_original_F)]
list_original_F = [x for x in list_original_F if len(x) == 84]
list_original_NF = [x for x in list_original_NF if len(x) == 84]


n_samples = min(len(list_original_F), len(list_original_NF))
# Create an array of len n_samples with [1, 0, random number between min␣
 ↪and max of CHO (g) F]
# Create an array of len n_samples with [0, 1, random number between min␣
 ↪and max of CHO (g) NF]
labels_F = np.zeros((n_samples, 3))
labels_F[:, 0] = 1
# labels_F[:, 2] = random_numbers_class['F']
labels_F[:, 2] = np.random.uniform(min_max_classes.loc['Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Fast', ('CHO (g)', 'max')],␣
 ↪n_samples)

labels_NF = np.zeros((n_samples, 3))
labels_NF[:, 1] = 1
# labels_NF[:, 2] = random_numbers_class['NF']
labels_NF[:, 2] = np.random.uniform(min_max_classes.loc['Non Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Non Fast', ('CHO (g)',␣
 ↪'max')], n_samples)


# Sort labels_F[:,-1]
labels_F = labels_F[labels_F[:, -1].argsort()]
# Sort labels_NF[:,-1]
labels_NF = labels_NF[labels_NF[:, -1].argsort()]
```

```
x_input = randn(latent_dim * n_samples)
# reshape into a batch of inputs for the network
z_input = x_input.reshape(n_samples, latent_dim)

# Generate fake samples
g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
 ↪h5'))
fake_samples_F = g_model.predict([z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([z_input, labels_NF], verbose=0)
# Reshape fake_samples_F to be a list of arrays of shape (84,)
fake_samples_F = [x.reshape(84) for x in fake_samples_F]
# Reshape fake_samples_NF to be a list of arrays of shape (84,)
fake_samples_NF = [x.reshape(84) for x in fake_samples_NF]
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

[232]:
```
# Calculate the JS Distance between each element of list_original_F and␣
 ↪the other elements of list_original_F
js_F_jeat_map_original = np.zeros((len(list_original_F),␣
 ↪len(list_original_F)))
for i in range(len(list_original_F)):
    for j in range(len(list_original_F)):
        js_F_jeat_map_original[i, j] = jensenshannon(
            list_original_F[i], list_original_F[j])

# Calculate the JS Distance between each element of list_original_NF and␣
 ↪the other elements of list_original_NF
js_NF_jeat_map_original = np.zeros(
    (len(list_original_NF), len(list_original_NF)))
for i in range(len(list_original_NF)):
    for j in range(len(list_original_NF)):
        js_NF_jeat_map_original[i, j] = jensenshannon(
```

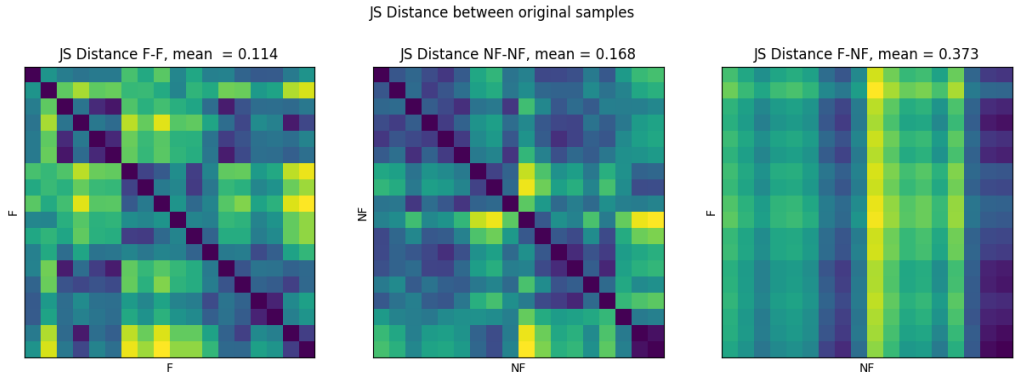
```
            list_original_NF[i], list_original_NF[j])


# Calculate the JS Distance between each element of list_original_F and␣
 ↪the elements of list_original_NF
js_F_NF_jeat_map_original = np.zeros(
    (len(list_original_F), len(list_original_NF)))
for i in range(len(list_original_F)):
    for j in range(len(list_original_NF)):
        js_F_NF_jeat_map_original[i, j] = jensenshannon(
            list_original_F[i], list_original_NF[j])



fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_jeat_map_original)
axs[0].set_title('JS Distance F-F, mean  = ' +
                 str(np.round(np.mean(js_F_jeat_map_original), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_jeat_map_original)
axs[1].set_title('JS Distance NF-NF, mean = ' +
                 str(np.round(np.mean(js_NF_jeat_map_original), 3)))
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_jeat_map_original)
axs[2].set_title('JS Distance F-NF, mean = ' +
                 str(np.round(np.mean(js_F_NF_jeat_map_original), 3)))
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
```

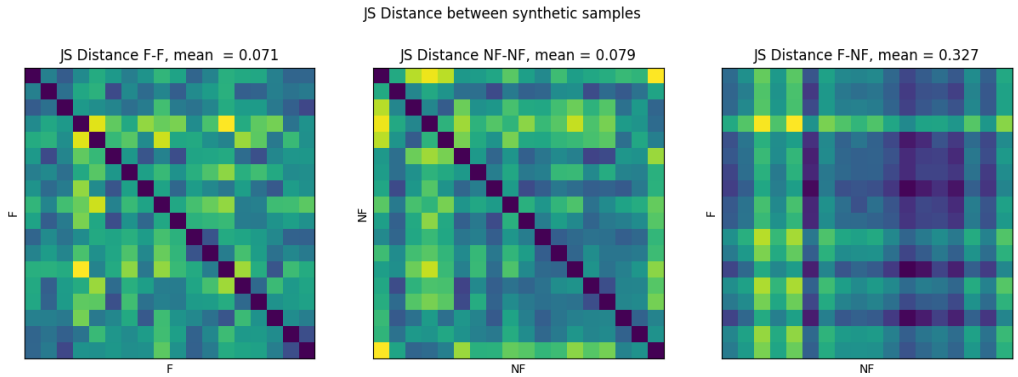
```
fig.suptitle('JS Distance between original samples')
fig.show()

# Calculate the JS Distance between each element of fake_samples_F and␣
 ↪the other elements of fake_samples_F
js_F_heat_map_syn = np.zeros((len(fake_samples_F), len(fake_samples_F)))
for i in range(len(fake_samples_F)):
    for j in range(len(fake_samples_F)):
        js_F_heat_map_syn[i, j] = jensenshannon(
            fake_samples_F[i], fake_samples_F[j])

# Calculate the JS Distance between each element of fake_samples_NF and␣
 ↪the other elements of fake_samples_NF
js_NF_heat_map_syn = np.zeros((len(fake_samples_NF),␣
 ↪len(fake_samples_NF)))
for i in range(len(fake_samples_NF)):
    for j in range(len(fake_samples_NF)):
        js_NF_heat_map_syn[i, j] = jensenshannon(
            fake_samples_NF[i], fake_samples_NF[j])

# Calculate the JS Distance between each element of fake_samples_F and␣
 ↪the elements of fake_samples_NF
js_F_NF_heat_map_syn = np.zeros((len(fake_samples_F),␣
 ↪len(fake_samples_NF)))
for i in range(len(fake_samples_F)):
    for j in range(len(fake_samples_NF)):
        js_F_NF_heat_map_syn[i, j] = jensenshannon(
            fake_samples_F[i], fake_samples_NF[j])



fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_heat_map_syn)
axs[0].set_title('JS Distance F-F, mean  = ' +
```

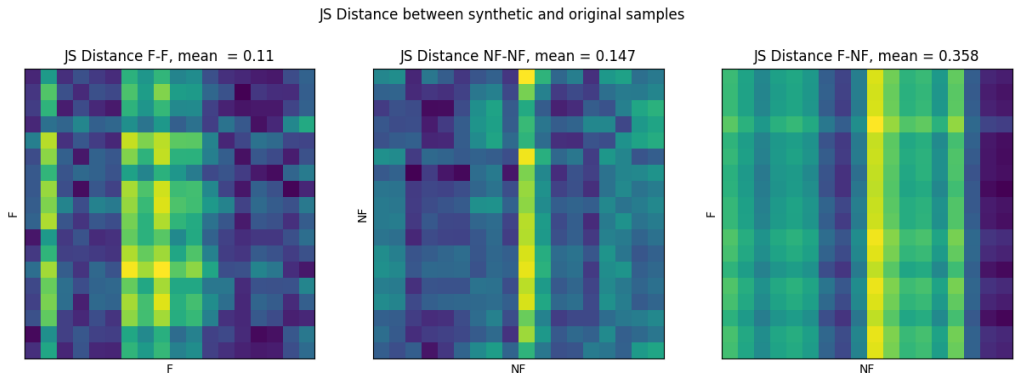
```
                str(np.round(np.mean(js_F_heat_map_syn), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_heat_map_syn)
axs[1].set_title('JS Distance NF-NF, mean = ' +
                str(np.round(np.mean(js_NF_heat_map_syn), 3)))
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_heat_map_syn)
axs[2].set_title('JS Distance F-NF, mean = ' +
                str(np.round(np.mean(js_F_NF_heat_map_syn), 3)))
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
fig.suptitle('JS Distance between synthetic samples')
fig.show()

# Calculate the JS Distance between each element of fake_samples_F and␣
 ↪the other elements of list_original_F
js_F_heat_map_syn_original = np.zeros(
    (len(fake_samples_F), len(list_original_F)))
for i in range(len(fake_samples_F)):
    for j in range(len(list_original_F)):
        js_F_heat_map_syn_original[i, j] = jensenshannon(
            fake_samples_F[i], list_original_F[j])

# Calculate the JS Distance between each element of fake_samples_NF and␣
 ↪the other elements of list_original_NF
js_NF_heat_map_syn_original = np.zeros(
```

```
    (len(fake_samples_NF), len(list_original_NF)))
for i in range(len(fake_samples_NF)):
    for j in range(len(list_original_NF)):
        js_NF_heat_map_syn_original[i, j] = jensenshannon(
            fake_samples_NF[i], list_original_NF[j])

# Calculate the JS Distance between each element of fake_samples_F and␣
 ↪the elements of list_original_NF
js_F_NF_heat_map_syn_original = np.zeros(
    (len(fake_samples_F), len(list_original_NF)))
for i in range(len(fake_samples_F)):
    for j in range(len(list_original_NF)):
        js_F_NF_heat_map_syn_original[i, j] = jensenshannon(
            fake_samples_F[i], list_original_NF[j])



fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_heat_map_syn_original)
axs[0].set_title('JS Distance F-F, mean  = ' +
                 str(np.round(np.mean(js_F_heat_map_syn_original), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_heat_map_syn_original)
axs[1].set_title('JS Distance NF-NF, mean = ' +
                 str(np.round(np.mean(js_NF_heat_map_syn_original), 3)))
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_heat_map_syn_original)
axs[2].set_title('JS Distance F-NF, mean = ' +
                 str(np.round(np.mean(js_F_NF_heat_map_syn_original), 3)))
```

```
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
fig.suptitle('JS Distance between synthetic and original samples')
fig.show()
```







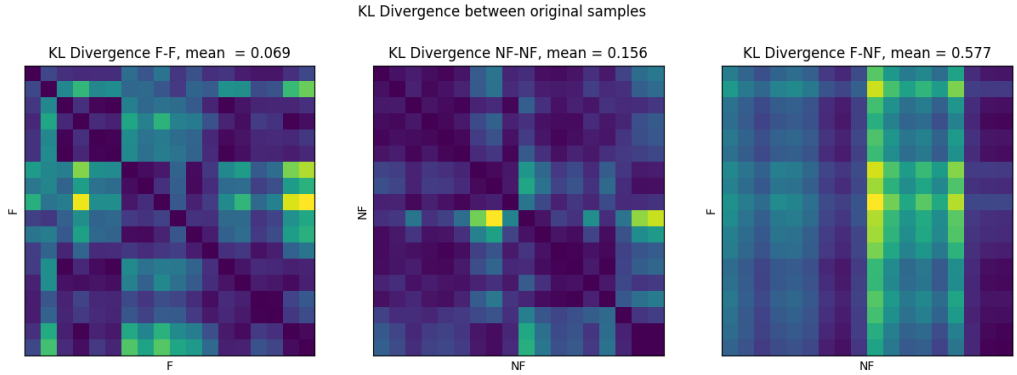
[233]:
```
# Calculate the KL divergence between each element of list_original_F and␣
 ↪the other elements of list_original_F
js_F_jeat_map_original = np.zeros((len(list_original_F),␣
 ↪len(list_original_F)))
for i in range(len(list_original_F)):
    for j in range(len(list_original_F)):
        js_F_jeat_map_original[i, j] = entropy(
            list_original_F[i], qk=list_original_F[j])


# Calculate the KL divergence between each element of list_original_NF␣
 ↪and the other elements of list_original_NF
js_NF_jeat_map_original = np.zeros(
    (len(list_original_NF), len(list_original_NF)))
for i in range(len(list_original_NF)):
    for j in range(len(list_original_NF)):
        js_NF_jeat_map_original[i, j] = entropy(
            list_original_NF[i], qk=list_original_NF[j])


# Calculate the KL divergence between each element of list_original_F and␣
 ↪the elements of list_original_NF
js_F_NF_jeat_map_original = np.zeros(
    (len(list_original_F), len(list_original_NF)))
for i in range(len(list_original_F)):
```

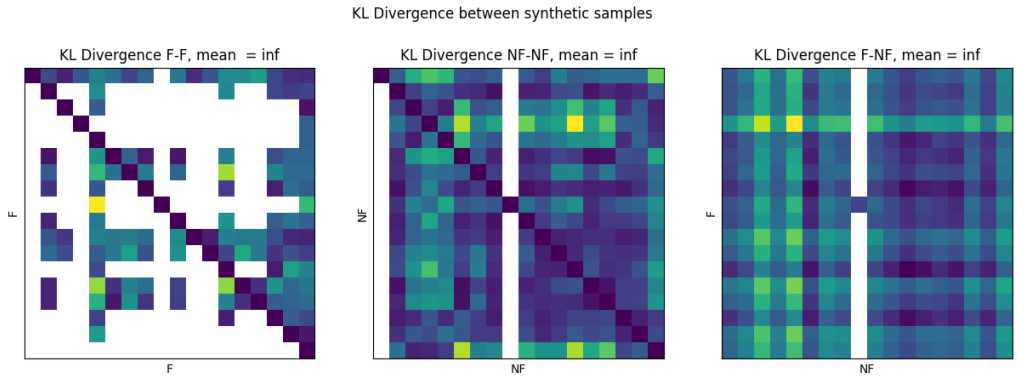
```
    for j in range(len(list_original_NF)):
        js_F_NF_jeat_map_original[i, j] = entropy(
            list_original_F[i], qk=list_original_NF[j])

fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_jeat_map_original)
axs[0].set_title('KL Divergence F-F, mean  = ' +
                 str(np.round(np.mean(js_F_jeat_map_original), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_jeat_map_original)
axs[1].set_title('KL Divergence NF-NF, mean = ' +
                 str(np.round(np.mean(js_NF_jeat_map_original), 3)))
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_jeat_map_original)
axs[2].set_title('KL Divergence F-NF, mean = ' +
                 str(np.round(np.mean(js_F_NF_jeat_map_original), 3)))
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
fig.suptitle('KL Divergence between original samples')
fig.show()

# Calculate the KL divergence between each element of fake_samples_F and␣
 ↪the other elements of fake_samples_F
js_F_heat_map_syn = np.zeros((len(fake_samples_F), len(fake_samples_F)))
for i in range(len(fake_samples_F)):
    for j in range(len(fake_samples_F)):
```

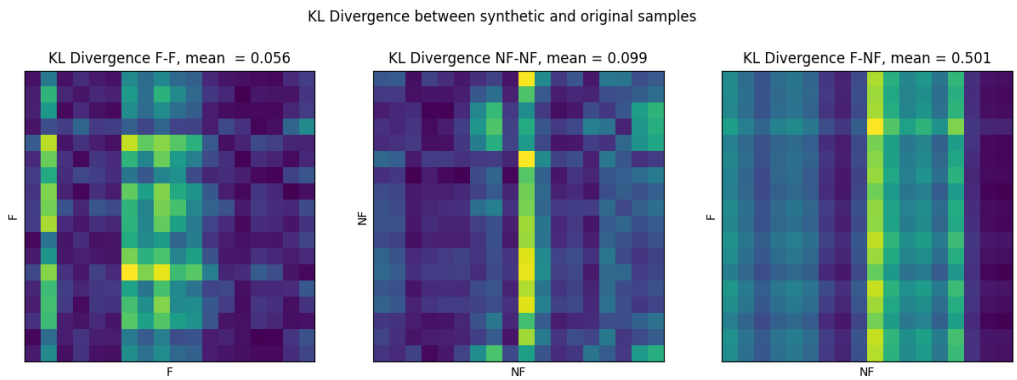
```
        js_F_heat_map_syn[i, j] = entropy(
            fake_samples_F[i], qk=fake_samples_F[j])

# Calculate the KL divergence between each element of fake_samples_NF and␣
 ↪the other elements of fake_samples_NF
js_NF_heat_map_syn = np.zeros((len(fake_samples_NF),␣
 ↪len(fake_samples_NF)))
for i in range(len(fake_samples_NF)):
    for j in range(len(fake_samples_NF)):
        js_NF_heat_map_syn[i, j] = entropy(
            fake_samples_NF[i], qk=fake_samples_NF[j])

# Calculate the KL divergence between each element of fake_samples_F and␣
 ↪the elements of fake_samples_NF
js_F_NF_heat_map_syn = np.zeros((len(fake_samples_F),␣
 ↪len(fake_samples_NF)))
for i in range(len(fake_samples_F)):
    for j in range(len(fake_samples_NF)):
        js_F_NF_heat_map_syn[i, j] = entropy(
            fake_samples_F[i], qk=fake_samples_NF[j])



fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_heat_map_syn)
axs[0].set_title('KL Divergence F-F, mean  = ' +
                 str(np.round(np.mean(js_F_heat_map_syn), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_heat_map_syn)
axs[1].set_title('KL Divergence NF-NF, mean = ' +
                 str(np.round(np.mean(js_NF_heat_map_syn), 3)))
```

```
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_heat_map_syn)
axs[2].set_title('KL Divergence F-NF, mean = ' +
                 str(np.round(np.mean(js_F_NF_heat_map_syn), 3)))
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
fig.suptitle('KL Divergence between synthetic samples')
fig.show()

# Calculate the KL divergence between each element of fake_samples_F and␣
 ↪the other elements of list_original_F
js_F_heat_map_syn_original = np.zeros(
    (len(fake_samples_F), len(list_original_F)))
for i in range(len(fake_samples_F)):
    for j in range(len(list_original_F)):
        js_F_heat_map_syn_original[i, j] = entropy(
            fake_samples_F[i], qk=list_original_F[j])

# Calculate the KL divergence between each element of fake_samples_NF and␣
 ↪the other elements of list_original_NF
js_NF_heat_map_syn_original = np.zeros(
    (len(fake_samples_NF), len(list_original_NF)))
for i in range(len(fake_samples_NF)):
    for j in range(len(list_original_NF)):
        js_NF_heat_map_syn_original[i, j] = entropy(
            fake_samples_NF[i], qk=list_original_NF[j])

# Calculate the KL divergence between each element of fake_samples_F and␣
 ↪the elements of list_original_NF
```

```
js_F_NF_heat_map_syn_original = np.zeros(
    (len(fake_samples_F), len(list_original_NF)))
for i in range(len(fake_samples_F)):
    for j in range(len(list_original_NF)):
        js_F_NF_heat_map_syn_original[i, j] = entropy(
            fake_samples_F[i], qk=list_original_NF[j])



fig, axs = plt.subplots(1, 3, figsize=(15, 5))
axs[0].imshow(js_F_heat_map_syn_original)
axs[0].set_title('KL Divergence F-F, mean  = ' +
                 str(np.round(np.mean(js_F_heat_map_syn_original), 3)))
axs[0].set_xlabel('F')
axs[0].set_ylabel('F')
axs[0].set_xticks([])
axs[0].set_yticks([])
axs[1].imshow(js_NF_heat_map_syn_original)
axs[1].set_title('KL Divergence NF-NF, mean = ' +
                 str(np.round(np.mean(js_NF_heat_map_syn_original), 3)))
axs[1].set_xlabel('NF')
axs[1].set_ylabel('NF')
axs[1].set_xticks([])
axs[1].set_yticks([])
axs[2].imshow(js_F_NF_heat_map_syn_original)
axs[2].set_title('KL Divergence F-NF, mean = ' +
                 str(np.round(np.mean(js_F_NF_heat_map_syn_original), 3)))
axs[2].set_xlabel('NF')
axs[2].set_ylabel('F')
axs[2].set_xticks([])
axs[2].set_yticks([])
fig.suptitle('KL Divergence between synthetic and original samples')
fig.show()
```







[227]:
```
n_samples = 500

latent_dim = 200
```

```
def generate_random_numbers(n_randoms, df_all_summary):
    random_numbers_class = {}
    for class_classificacio in df_all_summary['Classificació'].unique():
        values = df_all_summary[df_all_summary['Classificació']
                                == class_classificacio]['CHO (g)'].
 ↪tolist()
        # Round to 2 decimals
        values = [round(num, 2) for num in values]

        # Value count of values_F
        values_count = {}
        for i in values:
            if i in values_count:
                values_count[i] += 1
            else:
                values_count[i] = 1

        # Get the total frequency
        total_frequency = sum(values_count.values())
        # Normalize the frequencies to get probabilities
        probabilities = [v / total_frequency for v in values_count.
 ↪values()]
        # Generate random numbers following the distribution
        random_numbers_class[class_classificacio] = np.random.choice(
            list(values_count.keys()), size=n_randoms, p=probabilities)
    return random_numbers_class


random_numbers_class = generate_random_numbers(n_samples, df_all_summary)
min_max_classes = df_all_summary.groupby(
    'Classificació').agg({'CHO (g)': ['min', 'max']})
```

```
# Create an array of len n_samples with [1, 0, random number between min␣
 ↪and max of CHO (g) F]
# Create an array of len n_samples with [0, 1, random number between min␣
 ↪and max of CHO (g) NF]
labels_F = np.zeros((n_samples, 3))
labels_F[:, 0] = 1
# labels_F[:, 2] = random_numbers_class['F']
labels_F[:, 2] = np.random.uniform(min_max_classes.loc['Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Fast', ('CHO (g)', 'max')],␣
 ↪n_samples)

labels_NF = np.zeros((n_samples, 3))
labels_NF[:, 1] = 1
# labels_NF[:, 2] = random_numbers_class['NF']
labels_NF[:, 2] = np.random.uniform(min_max_classes.loc['Non Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Non Fast', ('CHO (g)',␣
 ↪'max')], n_samples)

# Sort labels_F[:,-1]
labels_F = labels_F[labels_F[:, -1].argsort()]
# Sort labels_NF[:,-1]
labels_NF = labels_NF[labels_NF[:, -1].argsort()]

x_input = randn(latent_dim * n_samples)
# reshape into a batch of inputs for the network
z_input = x_input.reshape(n_samples, latent_dim)

# Generate fake samples
g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
 ↪h5'))
fake_samples_F = g_model.predict([z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([z_input, labels_NF], verbose=0)
```

```
peaks_F = []
peaks_NF = []
auc_F = []
auc_NF = []
mean_slope_F = []
mean_slope_NF = []

for i in range(0, n_samples):
    fake_sample_F = fake_samples_F[i]
    fake_sample_F = fake_sample_F.reshape(84)
    fake_peaks_F, _F = find_peaks(fake_sample_F, height=0)
    # Find peak in fake_peaks_NF with the highest value
    peaks_F.append(fake_peaks_F[_F['peak_heights'].argmax()])
    auc_F.append(np.trapz(fake_sample_F[40:]))
    mean_slope_F.append(max(np.gradient(
        fake_sample_F[fake_peaks_F[_F['peak_heights'].argmax()]:], 2)))

    fake_sample_NF = fake_samples_NF[i]
    fake_sample_NF = fake_sample_NF.reshape(84)
    fake_peaks_NF, _NF = find_peaks(fake_sample_NF, height=0)
    peaks_NF.append(fake_peaks_NF[_NF['peak_heights'].argmax()])

    auc_NF.append(np.trapz(fake_sample_NF[40:]))
    mean_slope_NF.append(max(np.gradient(
        fake_sample_NF[fake_peaks_NF[_NF['peak_heights'].argmax()]:], 2)))

peaks_F_real = []
peaks_NF_real = []
auc_F_real = []
auc_NF_real = []
mean_slope_F_real = []
mean_slope_NF_real = []

for i in range(len(df_all_summary)):
```

```
    if df_all_summary['Classificació'][i] == 'Fast':
        real_sample_F = df_all_summary['RA'][i]
        real_sample_F = real_sample_F.reshape(84)
        auc_F_real.append(np.trapz(real_sample_F[40:]))
        real_F_peaks, _FR = find_peaks(real_sample_F, height=0)
        # Find peak in fake_peaks_NF with the highest value
        peaks_F_real.append(real_F_peaks[_FR['peak_heights'].argmax()])
        mean_slope_F_real.append(max(np.gradient(
            real_sample_F[real_F_peaks[_FR['peak_heights'].argmax()]:],␣
↪2)))

    else:
        real_sample_NF = df_all_summary['RA'][i]
        real_sample_NF = real_sample_NF.reshape(84)
        auc_NF_real.append(np.trapz(real_sample_NF[40:]))
        real_NF_peaks, _NFR = find_peaks(real_sample_NF, height=0)
        # Find peak in fake_peaks_NF with the highest value
        peaks_NF_real.append(real_NF_peaks[_NFR['peak_heights'].argmax()])
        mean_slope_NF_real.append(max(np.gradient(
            real_sample_NF[real_NF_peaks[_NFR['peak_heights'].argmax()]:
↪], 2)))


mean_peaks = [np.mean(peaks_F), np.mean(peaks_F_real),
              np.mean(peaks_NF), np.mean(peaks_NF_real)]
std_peaks = [np.std(peaks_F), np.std(peaks_F_real),
             np.std(peaks_NF), np.std(peaks_NF_real)]
mean_auc = [np.mean(auc_F), np.mean(auc_F_real),
            np.mean(auc_NF), np.mean(auc_NF_real)]
std_auc = [np.std(auc_F), np.std(auc_F_real),
           np.std(auc_NF), np.std(auc_NF_real)]
mean_slope = [np.mean(mean_slope_F), np.mean(mean_slope_F_real), np.mean(
    mean_slope_NF), np.mean(mean_slope_NF_real)]
std_slope = [np.std(mean_slope_F), np.std(mean_slope_F_real), np.std(
```

```
    mean_slope_NF), np.std(mean_slope_NF_real)]

median_peaks = [np.median(peaks_F), np.median(peaks_F_real),
                np.median(peaks_NF), np.median(peaks_NF_real)]
median_auc = [np.median(auc_F), np.median(auc_F_real),
              np.median(auc_NF), np.median(auc_NF_real)]
median_slope = [np.median(mean_slope_F), np.median(mean_slope_F_real), np.
 ↪median(
    mean_slope_NF), np.median(mean_slope_NF_real)]

q1_peaks = [np.quantile(peaks_F, 0.25), np.quantile(peaks_F_real, 0.25),
            np.quantile(peaks_NF, 0.25), np.quantile(peaks_NF_real, 0.25)]
q1_auc = [np.quantile(auc_F, 0.25), np.quantile(auc_F_real, 0.25),
          np.quantile(auc_NF, 0.25), np.quantile(auc_NF_real, 0.25)]
q1_slope = [np.quantile(mean_slope_F, 0.25), np.
 ↪quantile(mean_slope_F_real, 0.25), np.quantile(
    mean_slope_NF, 0.25), np.quantile(mean_slope_NF_real, 0.25)]
q3_peaks = [np.quantile(peaks_F, 0.75), np.quantile(peaks_F_real, 0.75),
            np.quantile(peaks_NF, 0.75), np.quantile(peaks_NF_real, 0.75)]
q3_auc = [np.quantile(auc_F, 0.75), np.quantile(auc_F_real, 0.75),
          np.quantile(auc_NF, 0.75), np.quantile(auc_NF_real, 0.75)]
q3_slope = [np.quantile(mean_slope_F, 0.75), np.
 ↪quantile(mean_slope_F_real, 0.75), np.quantile(
    mean_slope_NF, 0.75), np.quantile(mean_slope_NF_real, 0.75)]

# Round values
mean_peaks = [round(num, 2) for num in mean_peaks]
std_peaks = [round(num, 2) for num in std_peaks]
mean_auc = [round(num, 2) for num in mean_auc]
std_auc = [round(num, 2) for num in std_auc]
mean_slope = [round(num, 2) for num in mean_slope]
median_peaks = [round(num, 2) for num in median_peaks]
median_auc = [round(num, 2) for num in median_auc]
median_slope = [round(num, 2) for num in median_slope]
```

```
std_slope = [round(num, 2) for num in std_slope]
q1_peaks = [round(num, 2) for num in q1_peaks]
q1_auc = [round(num, 2) for num in q1_auc]
q1_slope = [round(num, 2) for num in q1_slope]
q3_peaks = [round(num, 2) for num in q3_peaks]
q3_auc = [round(num, 2) for num in q3_auc]
q3_slope = [round(num, 2) for num in q3_slope]

# dist_peaks = str(median_peaks (q1_peaks-q3_peaks))
# dist_auc = str(median_auc (q1_auc-q3_auc))
# dist_slope = str(median_slope (q1_slope-q3_slope))
dist_peaks = []
dist_auc = []
dist_slope = []
summary_peaks = []
summary_auc = []
summary_slope = []


for a in range(len(median_peaks)):
    dist_peaks.append(
        str(median_peaks[a]) + ' (' + str(q1_peaks[a]) + '-' +␣
↪str(q3_peaks[a]) + ')')
    dist_auc.append(
        str(median_auc[a]) + ' (' + str(q1_auc[a]) + '-' + str(q3_auc[a])␣
↪+ ')')
    dist_slope.append(
        str(median_slope[a]) + ' (' + str(q1_slope[a]) + '-' +␣
↪str(q3_slope[a]) + ')')

    summary_peaks.append(str(mean_peaks[a]) + ' (' + str(std_peaks[a]) +␣
↪')')
    summary_auc.append(str(mean_auc[a]) + ' (' + str(std_auc[a]) + ')')
```

```
    summary_slope.append(str(mean_slope[a]) + ' (' + str(std_slope[a]) +␣
 ↪')')



results = pd.DataFrame({'Summary peaks (mean(std))': summary_peaks,␣
 ↪'Summary AUC (mean(std))': summary_auc, 'Summary slope (mean(std))':␣
 ↪summary_slope,
                         'Distribution peaks': dist_peaks, 'Distribution␣
 ↪AUC': dist_auc, 'Distribution slope': dist_slope},
                        index=['F', 'F real', 'NF', 'NF real'])
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

[228]:
```
# Create an array of len n_samples with [1, 0, random number between min␣
 ↪and max of CHO (g) F]
# Create an array of len n_samples with [0, 1, random number between min␣
 ↪and max of CHO (g) NF]
labels_F = np.zeros((n_samples, 3))
labels_F[:, 0] = 1
# labels_F[:, 2] = random_numbers_class['F']
labels_F[:, 2] = np.random.uniform(min_max_classes.loc['Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Fast', ('CHO (g)', 'max')],␣
 ↪n_samples)


labels_NF = np.zeros((n_samples, 3))
labels_NF[:, 1] = 1
# labels_NF[:, 2] = random_numbers_class['NF']
labels_NF[:, 2] = np.random.uniform(min_max_classes.loc['Non Fast', (
    'CHO (g)', 'min')], min_max_classes.loc['Non Fast', ('CHO (g)',␣
 ↪'max')], n_samples)


# Sort labels_F[:,-1]
labels_F = labels_F[labels_F[:, -1].argsort()]
```

```
# Sort labels_NF[:,-1]
labels_NF = labels_NF[labels_NF[:, -1].argsort()]

x_input = randn(latent_dim * n_samples)
# reshape into a batch of inputs for the network
z_input = x_input.reshape(n_samples, latent_dim)

# Generate fake samples
g_model = load_model(os.path.join(CWD, 'models', '9950', 'cgan_generator.
 ↪h5'))
fake_samples_F = g_model.predict([z_input, labels_F], verbose=0)
fake_samples_NF = g_model.predict([z_input, labels_NF], verbose=0)



peaks_F = []
peaks_NF = []
auc_F = []
auc_NF = []
mean_slope_F = []
mean_slope_NF = []

for i in range(0, n_samples):
    fake_sample_F = fake_samples_F[i]
    fake_sample_F = fake_sample_F.reshape(84)
    fake_peaks_F, _F = find_peaks(fake_sample_F, height=0)
    # Find peak in fake_peaks_NF with the highest value
    peaks_F.append(fake_peaks_F[_F['peak_heights'].argmax()])
    auc_F.append(np.trapz(fake_sample_F[40:]))
    mean_slope_F.append(max(np.gradient(
        fake_sample_F[fake_peaks_F[_F['peak_heights'].argmax()]:], 2)))

    fake_sample_NF = fake_samples_NF[i]
    fake_sample_NF = fake_sample_NF.reshape(84)
    fake_peaks_NF, _NF = find_peaks(fake_sample_NF, height=0)
```

```
    peaks_NF.append(fake_peaks_NF[_NF['peak_heights'].argmax()])

    auc_NF.append(np.trapz(fake_sample_NF[40:]))
    mean_slope_NF.append(max(np.gradient(
        fake_sample_NF[fake_peaks_NF[_NF['peak_heights'].argmax()]:], 2)))

peaks_F_real = []
peaks_NF_real = []
auc_F_real = []
auc_NF_real = []
mean_slope_F_real = []
mean_slope_NF_real = []

for i in range(len(df_all_summary)):
    if df_all_summary['Classificació'][i] == 'Fast':
        real_sample_F = df_all_summary['RA'][i]
        real_sample_F = real_sample_F.reshape(84)
        auc_F_real.append(np.trapz(real_sample_F[40:]))
        real_F_peaks, _FR = find_peaks(real_sample_F, height=0)
        # Find peak in fake_peaks_NF with the highest value
        peaks_F_real.append(real_F_peaks[_FR['peak_heights'].argmax()])
        mean_slope_F_real.append(max(np.gradient(
            real_sample_F[real_F_peaks[_FR['peak_heights'].argmax()]:],␣
↪2)))

    else:
        real_sample_NF = df_all_summary['RA'][i]
        real_sample_NF = real_sample_NF.reshape(84)
        auc_NF_real.append(np.trapz(real_sample_NF[40:]))
        real_NF_peaks, _NFR = find_peaks(real_sample_NF, height=0)
        # Find peak in fake_peaks_NF with the highest value
        peaks_NF_real.append(real_NF_peaks[_NFR['peak_heights'].argmax()])
        mean_slope_NF_real.append(max(np.gradient(
```

```
            real_sample_NF[real_NF_peaks[_NFR['peak_heights'].argmax()]:
↪], 2)))
```

```
WARNING:tensorflow:No training configuration found in the save file, so the
model was *not* compiled. Compile it manually.
```

Presentation of the results for the measured metrics

[229]:
```
results
```

[229]:

	Summary peaks (mean(std))	Summary AUC (mean(std))	\
F	5.96 (3.11)	52.22 (23.51)	
F real	7.39 (5.81)	48.89 (41.48)	
NF	12.94 (4.46)	2246.36 (402.86)	
NF real	15.03 (9.54)	2509.47 (1526.31)	

	Summary slope (mean(std))	Distribution peaks	\
F	4.88 (5.02)	6.0 (4.0-7.0)	
F real	4.09 (5.34)	5.5 (3.0-8.75)	
NF	3.01 (2.9)	13.0 (9.0-15.0)	
NF real	2.4 (3.25)	14.0 (8.75-18.25)	

	Distribution AUC	Distribution slope
F	48.37 (35.57-66.28)	3.4 (0.48-7.7)
F real	32.02 (18.16-75.12)	2.84 (-0.0-4.4)
NF	2199.63 (1951.41-2495.81)	2.2 (0.7-4.46)
NF real	2194.31 (1457.66-3134.72)	0.96 (-0.08-4.18)


Universitat
de Girona