Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN TECNOLOXÍAS DA INFORMACIÓN

Unha aproximación Deep Learning para render de nubes de puntos

Estudante: Martín Sánchez Fontao
Dirección: Emilio José Padrón González
Dirección: Francisco Javier Taibo Pena
Dirección: Luis Omar Álvarez Mures

A Coruña, xuño de 2021.

Dedícollo á miña moza por crer en min en todo momento.

Agradecementos

Aos meus directores de proxecto por teren paciencia suficiente para axudarme nesta aventura. A miña moza, nai e familia por repetirme tódolos días ”Martín, o proxecto...”. E ós meus amigos e compañeiros por darme ánimos e consellos ainda que sexa dende a distancia.

Resumo

Unha nube de puntos é un conxunto de datos asociados a posicións puntuais no espazo. As nubes de puntos xeralmente son producidas por escáneres 3D, que miden gran cantidade de puntos nas superficies externas dos obxectos que os rodean, xerando conxuntos de datos que poden conter unha alta densidade de puntos. As nubes de puntos utilízanse para moitos propósitos, como para crear modelos CAD en 3D, para pezas manufacturadas, para metroloxía e inspección de calidade, e para multitude de aplicacións de visualización, animación e renderizado. Para unha visualización de calidade dunha nube de puntos de densidade arbitraria precísase de técnicas de renderizado específicas, xa que tanto o hardware das tarxetas gráficas como o *pipeline* de render que expoñen as principais APIS gráficas están deseñados para a visualización de polígonos (triángulos normalmente). Estas técnicas coñécense polo nome de Renderizado Baseado en Puntos (*PBR, Point-based Rendering*). Unha nube de puntos non ten topoloxía, o que dificulta conseguir vectores normais para a obtención dunha representación visual que permita reconstruír superficies sen buratos (emprégase para isto, polo xeral, algunha variante da técnica denominada *"splatting"*). O feito de que as nubes de puntos procedentes dunha captura adoiten ter ademais bastante ruído dificulta o xa complexo proceso de estimación de normais.

As arquitecturas de Deep learning aplícanse na actualidade con éxito en campos como visión artificial, recoñecemento de voz, procesamento de linguaxe natural, recoñecemento de audio, filtrado de redes sociais, tradución entre máquinas, bioinformática, etc. Nalgunha destas disciplinas estas arquitecturas están obtendo resultados comparables ou nalgúns casos superiores aos expertos humanos. A estimación dos vectores normais das superficies representadas por unha nube de puntos encaixa perfectamente dentro do que este tipo de sistemas pode proporcionar.

O primeiro obxectivo deste proxecto é a implementación dunha visualización avanzada dunha nube de puntos seguindo **técnicas Point-Based Rendering (PBR)** do estado da arte empregando un **algoritmo de estimación de normais** baseado en técnicas clásicas.

O segundo obxectivo é a implementación dun **método para a estimación de normais empregando ténicas de Deep Learning** que permitan mellorar os resultados obtidos mediante a aplicación de técnicas tradicionais.

Para as tarefas de visualización farase uso da API gráfica multiplataforma **OpenGL**, mentres que para a parte de Deep Learning empregaremos **TensorFlow** e **Keras**.

Abstract

A point cloud is a set of data points in space. Point clouds are generally produced by 3D scanners, which measure a large number of points on the external surfaces of objects around them, creating sets of data with a severe amount of points. As the output of 3D scanning processes, point clouds are used for many purposes, including to create 3D CAD models for manufactured parts, for metrology and quality inspection, and for a multitude of visualization, animation and rendering applications. In order to achieve a quality visualization of a point cloud specific render techniques are required, graphic cards hardware and render pipeline are normally designed to process polygon meshes (mainly triangles). Those techniques are known as Point-Based Rendering (PBR). Since point clouds have no topology, obtaining proper normal vectors that make it possible to have a quality render of the surfaces with no holes (usually by applying any variant of a technique known as 'splatting') is not trivial. Additionally, the noise commonly present in point clouds obtained from a capture makes any normal estimation process even harder.

Deep learning architectures have been successfully applied to fields as computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases superior to human experts. Normal estimation of point cloud surfaces perfectly fits the kind of result these universal function approximators can provide

The first goal of this project is the implementation of an **interactive viewer** that provides a quality visualization of a point cloud dataset, applying some state-of-the-art Point-based Rendering technique. A classic normal estimation process based on a numerical method will be used firstly.

The second goal of this project will be the implementation of **normal estimation algorithms** based on a Deep Learning model. This model will be tested out against the classic numerical approach.

In order to achieve these results, **OpenGL** will be used to render primitives in real-time. On the Deep Learning side, **TensorFlow** and **Keras** will be used since it is the standard for Machine Learning and Deep Learning models.

Palabras chave:

- OpenGL
- Nube de puntos
- Estimación de normais
- Splatting
- Aprendizaxe profundo
- Keras
- Red neuronal profunda

Keywords:

- OpenGL
- Point clouds
- Normal estimation
- Splatting
- Deep learning
- Keras
- Deep neural network

Índice Xeral

Índice de Figuras

Capítulo 1

Introdución

PARA coñecer o contexto no que se realiza esta memoria falaremos de dous ámbitos distintos que rematan por confluir no mesmo fin. Por unha banda a visualización de gráficos 3D, e por outra banda, o Aprendizaxe Profundo (ou Deep Learning).

A temática principal serán as *nubes de puntos* (ou *Point Clouds*) que representaremos gráficamente en pantalla mediante o desenvolvemento dun visualizador baseado na API *OpenGL*. Unha visualización avanzada destes datos precisa dispoñer do vector normal á superficie ao que corresponde cada punto, polo que a segunda parte deste traballo céntrase na obtención dunha estimación destas normais mediante un modelo construído cunha *rede de neuronas artificiais* (ou *Deep Neural Networks*).

Neste capítulo contextualizamos o traballo desenvolvido neste TFG, introducindo os obxectivos perseguidos no mesmo e bosquexando o contido desta memoria

1.1 Motivación e contexto

Unha *nube de puntos* é un conxunto de puntos 3D no espazo, sen ninguha relación topolóxica, que, ademais das coordenadas xeométricas "xyz" coa posición 3D, pode incorporar distinta información a cada un dos puntos da nube: cor, normal, reflectividade, etc. Estes puntos representan a superficie dun ou de varios obxectos, e permiten recrear con precisión diferentes escenarios. As *nubes de puntos* xeralmente se obteñen a partir dun proceso de escaneo 3D, por exemplo cun láser LiDAR, mediante fotogrametría ou como froito dunha simulación numérica. Dúas son as principais problemáticas ás que nos enfrontamos á hora de representar estas *nubes de puntos*; primeiro, que as APIs gráficas xeralmente están deseñadas para a visualización de polígonos (típicamente, triángulos), e segundo, que as capturas de *nubes de puntos* por parte dos escáneres adoitan ter superficies ruidosas ou imprecisas. As nubes de puntos non teñen topoloxía, o que nos dificulta poder representalas sen que aparezan buratos de por medio. Para que a representación sexa o máis realista posible, pódese

facer uso de técnicas de *"splatting"* [10] e recrear unha superficie a partir dun punto e os seus veciños cercanos, ou facer un teselado [11] e reconstruir a superficie en forma de malla poligonal a partir dos puntos. Para sacar o máximo partido á representación mediante nubes de puntos, evitaremos a etapa intermedia de construcción de malla de polígonos que se utiliza no teselado, e utilizaremos por tanto técnicas de *"splatting"*. Os splats dan forma de disco aos puntos, o que nos permite evitar os buratos na superficie dos obxectos, e obter uns resultados de visualización máis realistas. Existen varias metodoloxías para poder aproximar o tamaño dos *splats*. Nós nos centraremos nalgunha técnica que explote o uso das normais para acadar os mellores resultados posibles. Xa hai métodos matemáticos que permiten estimar as *normais*, pero neste traballo imos intentar conseguir aproximar esas *normais* utilizando técnicas de *Deep Learning*.

O Machine Learning é unha disciplina dentro da Intelixencia Artificial que se utiliza para identificar patróns nun conxunto de datos e poder facer logo predicción sobre datos da mesma índole, pero completamente novos. Unha das vantaxes que ten o Machine Learning é que non tes que programar explícitamente ese recoñecemento de patróns, senón que ti marcas unhas pautas de aprendizaxe e uns datos a analizar e o algoritmo será capaz de aproximar novos resultados. Para que un algoritmo de Machine Learning sexa fiable e funcional, necesita un adestramento cun número finito (e grande) de datos, e unha vez acadados uns bos resultados do adestramento, o algoritmo poderá utilizarse para datos novos e descoñecidos. No noso caso, este tipo de algoritmos poderíamos utilizalo para estimar o valor das normais da nube de puntos. Para iso teríamos que alimenta coas súas normáis como datos coñecidos, e unha vez adestrado o algoritmo, estimar novas normáis descoñecidas previamente. O problema co Machine Learning é que está limitado pola cantidade de datos coa que adestras o algoritmo, xa que chega a un punto no que por máis datos que engadas ao adestramento, o algoritmo xa non terá máis capacidade de aprendizaxe. Para poder obter mellores resultados a cantos máis datos de adestramento utilices, existe unha rama do Machine Learning que pretende emular o funcionamento das redes neuranais humanas. Estas son as Redes de Neuronas Artificiais. Máis concretamente utilizaremos Deep Neural Networks, xa que a priori non sabemos cal será a complexidade do problema, e este tipo de Rede Neuronal nos permite probar diferentes combinacións de capas ocultas e número de neuronas a utilizar para resolver problemas de maior ou menor complexidade.

1.2 Obxectivos do proxecto

O primeiro obxectivo será obter unha visualización de calidade das nubes de puntos. Para esta finalidade necesitaremos dispoñer das *normais* dos puntos posto que é un requisito de calquera método de visualización avanzada de nubes de puntos. Vaise deselvolver un visualizador a medida en OpenGL que nos permita ver os obxectos e as súas *normais*, no que se poida facer comparativas dos resultados que obtemos con diferentes métodos de visualización e con diferentes datos para as *normais* dos puntos.

Unha vez conseguido isto, haberá que conseguir propoñer un modelo de rede neuronal profunda que consiga uns resultados aceptables á hora de estimar as *normais*. A finalidade dos resultados deste modelo, será intentar acadar con iguais ou mellores resultados ás técnicas habituais de estimación. E cando estes requisitos previos estean completos, como último obxectivo engadirase ao visualizador a posibilidade de cargar as *normais* adquiridas mediante métodos númericos habituais, e tamén as *normais* obtidas mediante o noso modelo de *Deep Learning*, para poder comparar ambos resultados.

Capítulo 2

Planificación e Metodoloxía

Neste capítulo veremos o método escollido para o desenvolvemento do proxecto. Este proxecto ten unha compoñente importante de investigación, sobre todo na parte do modelo de estimación de normais, o que unido ao completo descoñecemento das tecnoloxías empregadas para o seu desenvolvemento, tanto na parte de rendering como na parte de Deep Learning, complicaba a estimación e planificación do traballo. Por tanto, interésanos marcar pequenas metas, e ir engadindo novas funcionalidades a medida que avanza o deseño. Ademáis priorizamos a funcionalidade temperá, para poder probar os nosos avances o antes posible, polo que se determinou que a mellor aproximación sería utilizar unha metodoloxía áxil. Falaremos a continuación de como funcionan os principios áxiles e como levamos a cabo o proxecto mediante un diagrama de Gantt.

2.1 Desenvolvemento áxil en software

O desenvolvemento áxil de software [12] está baseado no desenvolvemento iterativo e incremental, deste xeito os requisitos e as solucións evolucionan segundo as necesidades do proxecto. Cada iteración do ciclo de vida inclúe planificación, análise de requisitos, deseño, codificación, probas e documentación. Ten moita importancia conseguir prototipos funcionais en cada iteración, para poder seguir avanzando e engadindo funcionalidades. Xeralmente os métodos áxiles enfatizan máis na comunicación e revisión directa do producto, en lugar de levar unha documentación.

2.2 Agile Manifesto

O Agile Manifesto [13] é un escrito que recolle moitas ideas e principios de como levar a cabo o desenvolvemento áxil. En 2001 xuntáronse un grupo de expertos desenvolvedores para debater sobre distintas metodoloxías lixeiras de desenvolvemento coa fin de redactar un

escrito e chegar a uns estándars básicos que se deben seguir. Culminando na publicación do *Manifesto for Agile Software Development.* Estes son algúns dos principios que definen:

- A maior prioridade é satisfacer ao cliente mediante a entrega temprana e contínua de software funcional.
- Aceptamos que os requisitos cambien, incluso en etapas tardías do desenvolvemento. Os procesos áxiles aproveitan o cambio para proporcionar vantaxa competitiva ao cliente.
- Entrégase software funcional frecuentemente, entre dúas semanas e dous meses, con preferencia ao tempo máis curto posible.
- Os responsables do negocio e os desenvolvedores traballamos xuntos de forma coordinada durante todo o proxecto.
- O método máis eficiente e efectivo de comunicar información ao equipo de desenvolvemento é a conversación cara a cara.

2.3 Scrum

O Scrum [14] é unha metodoloxía áxil que se basea nun desenvolvemento incremental. Isto quere dicir que partiremos dun sotware base e desenvolveremos novas funcionalidades a partir de el. Cada nova funcionalidade ou mellora se propón como un 'Sprint'. Calqueira ciclo de desenvolvemento ou Sprint do produto ou servizo se divide en pequenas partes que incluirán fases de análise, desenvolvemento e probas. Esta metodoloxía permite abordar modelos complexos que demandan flexibilidade e rapidez na obtención de resultados. A clave será xestionar e normalizar os erros que adoitan producirse en desenvolvementos largos a través de reunións frecuentes para asegurar o cumplimento dos obxectivos establecidos. Ao rematar un Sprint debe reflexionarse e propoñer melloras sobre o proxecto.

2.4 Aplicación

Durante a realización do proxecto foron utilizadas moitas das ideas e propostas de *Agile Manifesto.* Fíxose un bosquexo inicial de planificación a grandes rasgos, formando Sprints. Eses Sprints fóronse dividindo en pequenas tareas máis sinxelas que non levarían máis de 1 ou 2 semanas, para obter resultados con presteza. Utilízanse tamén algunhas ideas propostas en *Scrum*, como a xestión e o deseño do proxecto que se fixo de forma incremental, creando sempre un prototipo funcional básico, ao que se lle agregarían novos requerimentos e funcionalidades a posteriori. Así en cada reunión falaríase de engadir novas funcionalidades

completas ao prototipo inicial. Houbo que diferenciar a parte de deseño e implementación software, coa parte de investigación e implementación de métodos de Deep Learning. Aínda que tiveron que convivir en moitas ocasións durante a planificación.

As reunións cos 3 directores leváronse a cabo mediante videoconferencias para aclarar os pasos a seguir no desenvolvemento do proxecto. Nestas reunións expoñíanse os avances do proxecto, e se valoraba que obxectivos se cumplirían, e se marcaban os novos obxectivos para a seguinte reunión. Tamén houbo comunicación por correo para solucionar problemas puntuais. Non se puideron realizar reunións en persóa debido á situación de pandemia aínda vixente.

Creouse dende o principio un repositorio Git onde se almacenou o proxecto. Isto sirveu para que todos puidésemos ver de primeira man os avances que se producían. Git é un software libre utilizado para almacenar código e levar un control de versións do mesmo. Polo que foi moi útil para poder probar diferentes funcionalidades creando ramas paralelas á versión estable, e tamén sirveu como backup, e para revisar as variacións no código.

Debido a que estamos tratando unha investigación partindo de moi poucos coñecementos acerca da materia, isto fixo difícil ter unha estimación do que nos podía levar o proxecto, por iso nas figuras (2.2) e (2.1) podemos ver o diagrama de Gantt co resumo de como avanzou o proxecto. Os recadros de cor verde representan o avance das tareas, e os recadros en cor vermello marcan os hitos máis determinantes para o avance do proxecto.

Recurso	Unidades	Horas	Coste por hora	Total
Software				0 €
Ordenador	1			1.500 €
Analista-Programador	1	900	15 €	13.500 €
				15.000 €

Figura 2.3: Estimación de costes

Tamén se pode ver na figura (2.3) unha estimación final dos costes[15] que supondría o proxecto. Para realizalo soamente foi necesario un analista-programador e un ordenador con acceso a internet, o software que utilizamos foi software libre.

Figura 2.1: Diagrama de Gantt: segunda metade

Figura 2.2: Diagrama de Gantt: primeira metade

Capítulo 3

Fundamentos de Programación Gráfica Interactiva

Comezamos co apartado teórico que abarcará os coñecementos clave e necesarios para poder comprender os pasos posteriores no deseño do visualizador de nubes de puntos. Esta parte estará máis enfocada nos usos máis típicos das APIs gráficas, explicando o seu funcionamento básico, e despois centrándonos nas nubes de puntos que é o que máis nos interesa neste traballo. As APIs para render 3D interactivo (como OpenGL, Vulkan, DirectX...) utilízanse para poder programar aplicacións gráficas e conseguir renderizar os resultados por pantalla. Debido a que *OpenGL* foi ata fai pouco o API estándar de gráficos multiplataforma por excelencia, será o que usaremos no proxecto.

OpenGL é unha especificación estándar que define unha API multilinguaxe e multiplataforma coa que podemos xerar gráficos e imaxes 3D e 2D. Dise que *OpenGL* é unha especificación xa que non ten un código único de funcionamento, o único que indica é como debería ser o resultado de cada función e como debería funcionar, pero os detalles de implementación poden variar entre plataformas e distintos sistemas. Imos utili Na actualidade, esta e outras APIs e utilidades gráficas están mantidas polo grupo Khronos [16]. Tanto as APIs gráficas coma OpenGL, como o hardware gráfico, están enfocadas ao traballo con triángulos, o que dificulta obter representacións 3D de calidade en tempo real con nubes de puntos.

3.1 Pipeline

Cando usamos unha API de gráficos como pode ser OpenGL, necesitamos cargar os datos de entrada na GPU para poder visualizalos por pantalla. Estos datos de entrada indican as pocisións e outras características das primitivas, como pode ser cor, textura, valor das normais, etc. Unha vez cargados esos datos en GPU, para poder obter un resultado final, no que se visualizan obxectos completos acotando a súa forma e mostrando súas cores, texturas, etc;

será necesario que esos datos pasen por varias etapas e sufran determinadas transformacións. Todas esas etapas nas que os obxectos pasan dunha representación básica de primitivas ata conseguir formar obxectos máis complexos, forman o **pipeline**3.1.1. Nalgunhas destas etapas permítense realizar modificacións por parte do usuario, estas etapas que son programables reciben o nome de **shaders**. Están escritos nunha linguaxe creada para OpenGL similar a C, que se chama OpenGL Shading Language (GLSL) [17]. Unha vez temos os datos dun obxecto en memoria da GPU, con GLSL defínese o comportamento dos obxectos. Poden cambiarse as cores, engadir unha textura, dar movemento aos obxectos ou ao escenario, modificar as formas das primitivas, engadir transparencia, etc.

3.1.1 Etapas do pipeline

Na figura 3.1 vemos un exemplo básico dún triángulo a través das etapas do pipeline. O pipeline recibe como entrada as coordenadas 3D dos vértices e outra información como pode ser a cor que queremos utilizar. Imos explicar paso a paso que transformacións se dan en cada etapa:

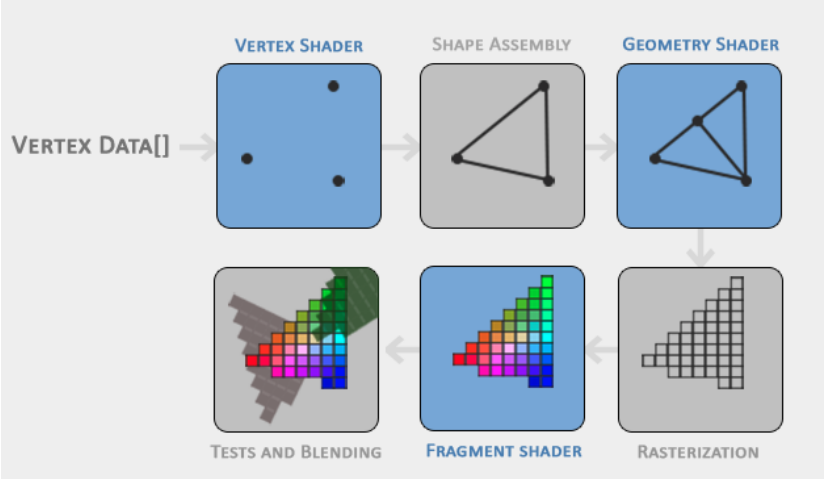


Figura 3.1: Aquí pódense ver representadas as estapas que forman o pipeline [1]

A primeira etapa do pipeline é o **vertex shader**. Aquí recolleremos os datos de cada un dos vértices da primitiva, que xa temos previamente cargados na GPU. Eses datos son xeralmente os de posición, de cor e das normais, ainda que se poden engadir os datos que necesitemos. Para poder representar a posición dos puntos, estes deben ter valores entre -1 e 1 para que poidan verse por pantalla. Nesta etapa decídese onde se colocarán as coordenadas das primitivas que darán forma aos obxectos finais. Unha vez temos o vertex shader definido, o pipeline pasa á seguinte etapa, o **shape assembly**. Nesta etapa, tamén chamada ensamblaxe de primitivas (primitive assembly), se collen todos os puntos que xa temos localizados da etapa anterior e se lles da a forma da primitiva seleccionada. En este caso son as coordenadas

dun triángulo polo que uniría eses puntos para obter unha forma triángular. A terceira fase correspondese co **geometry shader**, este ten a capacidade de dar formas distintas aos vértices que xa tiñamos da anterior fase. Por exemplo, poderíamos pasar uns puntos como primitiva, e darlle a cada punto unha lonxitude determinada, transformando os puntos en liñas. Se non se queren modificar as primitivas de entrada, o geometry shader pode deixarse sen especificar en OpenGL, e queda configurado coa forma das primitivas orixinais. A saída da fase anterior achéganos á etapa de 'rasterización' (**rasterization stage**), onde se mapean as primitivas que temos cos seus píxeles correspondentes na pantalla. De este xeito obtemos fragmentos dos que poderemos determinar a forma e a cor. Antes de pasar á seguinte fase, descártanse os vertices que quedan fóra da representación na pantalla (ver 3.2), e se formarán novos vértices que se adecúen ao espazo visible; isto chámase "clipping".

Antes de obter o resultado final, poderemos configurar o **fragment shader** [18]. Aquí calcúlase a cor e as características finais de cada fragmento.

Despois de configurar e pasar tódalas fases temos que rematar co **alpha test e 'blending'**. Nesta última etapa explótase o depth value[19] para saber que partes do render van por diante ou por atrás. Isto utilízase para renderizar sóamente os fragmentos que son visibles en cada momento, e tamén comproba os alpha values [20] que nos axudan a coñecer a opacidade dos obxectos. Finalmente toda a información recollida se mestura ou se fai un "blend" e se pode representar por pantalla.

3.2 Sistema de coordenadas

Tras a sección das etapas do pipeline, xa sabemos un pouco como se representan os nosos obxectos por pantalla. Para conseguir ver un obxecto dende diferentes perspectivas e poder movernos polo entorno, necesitamos coñecer algúns conceptos sobre o sistema de coordenadas. Unha vez entendamos estes conceptos poderemos crear a nosa propia cámara [21] [1], e visualizar os obxectos dende todos os ángulos.

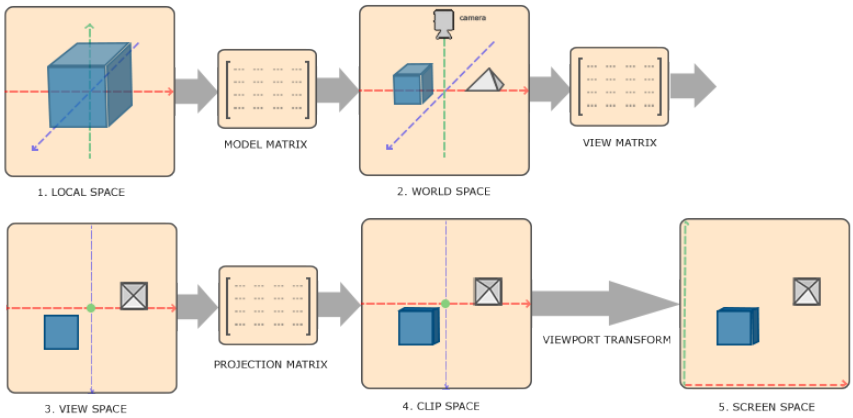
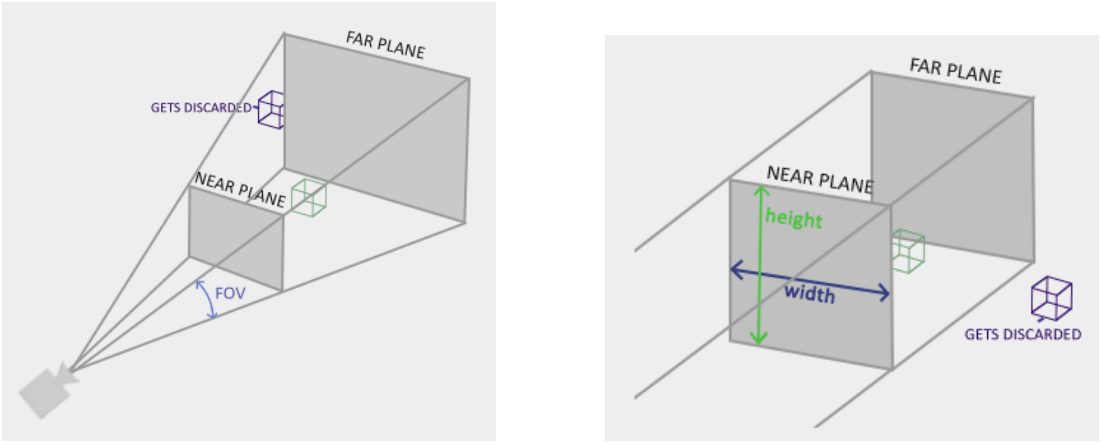


Figura 3.2: Aquí vense as transformacións que se levan a cabo para visualizar os obxectos [1]

Para obter por pantalla a visualización que nos gustaría dos obxectos, temos que levar a cabo varias transformacións. Podemos seguir a figura 3.2 para guiarnos no proceso. Cando definimos inicialmente un obxecto, damos as súas coordenadas no espazo local, isto quere dicir que esas coordenadas son soamente relativas ao obxecto a representar. Se cargásemos dous obxectos, só se terían en conta a si mesmas, polo que superpoñeríanse e non acadaríamos o resultado esperado. Normalmente nos interesa ubicar un obxecto nun espazo relativo ao mundo, onde poder colocar varios obxectos en distintos lugares. Podemos escalar, rotar e trasladar os obxectos. Para realizar esta primeira transformación utilízase a **matriz modelo**. Usando a **matriz vista** podemos moldear a escena, conseguindo movernos como se tiveramos unha cámara observando. No espazo vista estarán incluídas todas os obxectos que indicamos ao vertex shader, pero a nosa pantalla ten un tamaño limitado. Se os obxectos están moi cerca e algún fragmento non puidera verse por pantalla, eses vértices estarían ”clipped” (descartados ou fóra do plano). A **matriz de proxección** encárgase en determinar que partes dos obxectos entran nos límites establecidos para representar por pantalla, e que partes quedan fóra e por tanto non chegan a renderizarse. Existen dúas aproximacións para realizar a matriz de proxección. Unha máis fiel ao mundo real e como vemos as cousas, **proxección perspectiva** 3.3a. E outra máis fiel ao que se esta representando, que non varía os tamaños dos obxectos ainda que esten a distancias distintas, a **proxección ortográfica** 3.3b. Ambas marcan os seus limites de renderizado dentro dun “frustum”, que é a parte visible da escena. O frustum está contido entre o ‘far plane’, ‘near plane’ e o ‘field of view’, todo o que rebase esos límites será descartado.



(a) Perspectiva

(b) Ortográfica

Figura 3.3: Representación dos tipos de proxección [1]

Xuntando as matrices vistas anteriormente coas coordenadas das primitivas que recibi-

mos no vertex shader, obtemos a matriz MVP (Modelo Vista Proxección) que nos axudará a determinar a colocación dos obxectos ao salir por pantalla:

*MVP = Proxección * Vista * Modelo * Vértices De Entrada*

3.3 Nubes de puntos

De entre todas as primitivas, neste traballo nos centraremos nos puntos. Unha nube de puntos [22] é un conxunto de vértices no espazo que representa unha forma ou obxecto en 3D. Cada punto ten definido como mínimo as coordenadas X, Y e Z indicando a súa posición. Os puntos estan localizados nas superficies dos obxectos visibles. As nubes de puntos representan con moita precisión obxectos da vida real, polo que se poden aforrar por exemplo visitas a un edificio para saber como está organizado, isto reduciría moitos costes. Además os puntos son máis fáciles de manexar en grandes cantidades que os triángulos, xa que o ordenador non ten que preocuparse do escalado, a rotación ou a relación entre obxectos; só ten que preocuparse da color e posición dos puntos. Para xerar arquivos de nubes de puntos utilízanse escáneres láser [23] ou fotogrametría [24].

3.3.1 Tipos de arquivos de nubes de puntos

Os tipos de arquivos de nubes de puntos poden ser ASCII ou binarios. Os sistemas binarios almacenan os datos directamente en código binario, o que facilita a súa lectura e escritura, pos iso son máis usados para un acceso máis eficiente dos datos; algúns tipos son FLS, PCD e LAS. Os sistemas ASCII representan a información en arquivos de texto, polo tanto é máis pesado nos procesos de escritura e lectura, por iso son máis usados para ter almacenados os datos durante un periodo largo de tempo; algúns tipos son XYZ, ASC. Algúns tipos poden ser representados en sistemas ASCII ou binarios, como PLY e OBJ.

Algunhas ferramentas que se utilizan para traballar con nubes de puntos son **Euclideon** [25], **Meshlab** [26], **CloudCompare** [27], **Point Cloud Library (PCL)** [28].

3.3.2 Splats

Sabemos que os puntos dunha nube representan a superficie do obxecto escaneada, pero os puntos non teñen dimensionalidade. Isto quere dicir que non teñen volume, área ou tamaño. Por isto, á hora de poder representar fielmente superficies necesitaríamos darlle un tamaño e unha forma a estes puntos. Para poñer solución a isto, existen dúas formas de representar estes puntos, unha variando o tamaño dos mesmos ata conseguir visualizalo como unha superficie homoxénea, e outra creando "splats" [29] na ubicación dos puntos. Os splats son representacións en forma de disco, dos que se determina xeralmente o seu radio, a súa

posición, súa cor e a normal. Estímase o tamaño que debería ter cada splat para que as superficies se vexan o máis realistas posibles e sen buracos. Ao dispoñer das normais podemos identificar a orientación das superficies e obter representacións moito máis realistas e fieis ao obxecto real.

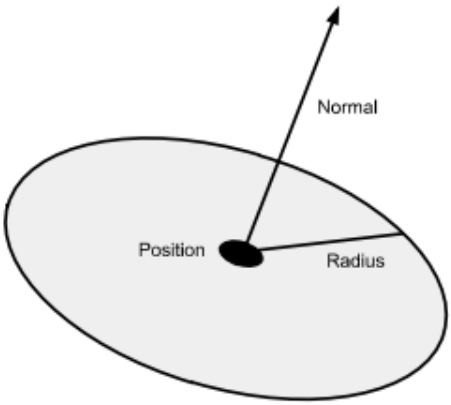


Figura 3.4: Representación básica dun splat [2]

3.3.3 Técnicas de rasterización de puntos

A rasterización é o proceso no que unha imaxe descrita nun formato vectorial, se convirte nun conxunto de píxeles desplegados na pantalla do ordenador, como xa falamos no 3.1.1. Como os puntos dunha nube de puntos teñen dimensión cero, non poden ser representados directamente. OpenGL non ten as ferramentas necesarias para resolver este inconveniente de forma predeterminada, polo que haberá que implementar algunhas alternativas . Estas son algunhas delas [2] [30]:

Sized-Fixed Points

Sen dúbida é a opción máis sinxela á hora de representar puntos. A finalidade deste método é establecer un tamaño fixo a todos os puntos da nosa nube para poder visualizalo por pantalla. Pero escoller o tamaño ideal pode ser tedioso, e ademáis, se queres cambiar a perspectiva dende a que ves o obxecto, todo o traballo que ocupaches buscando o tamaño ideal se perdería...Os puntos que representemos, terán un aspecto cadrado e todos terán o mesmo tamaño.

Image-aligned Squares

Para poder solventar algunhas das problemáticas coas que nos atopábamos cos puntos fixos, podemos utilizar "Image-aligned Squares". Con este método buscamos adaptar o tamaño

dos puntos según a perspectiva dende onde miramos un obxecto. Se queremos vela dende máis lonxe, os puntos crecen, e se nos acercamos, diminúen; dando a impresión de que en realidade é unha superficie completa, e non formada por puntos. Necesitaremos estimar o tamaño dos splats e máis o seu radio para poder representalos.

O tamaño se aproxima facendo o escorzo (*perspective foreshort*) [29] do radio *r* do splat usando o *depth value* do centro p_z que proporcionan as coordenadas da cámara. Para isto, engadiremos a seguinte fórmula no noso vertex shader:

$$PointSize = 2r \cdot \frac{n}{p_z} \cdot \frac{h}{t - b'}$$

Os valores *n*, *t* e *b* son valores recollidos da cámara, e son near, top e bottom. Finalmente o valor *h* representa á altura que indica o noso viewport.

Os calculos que se fan para establecer o tamaño dos puntos son bastante lixeiros, polo que é o método máis rápido e aproximado a un resultado real. Aínda así non consigue resultados óptimos en renders de alta calidade, xa que cerca do contorno, onde os calculos son menos precisos, pode verse a forma dos splats. Todos os pixels xerados dun punto, teñen o mesmo *depth value*, o que imposibilita facer técnicas de blending en zonas que visualmente se están superpoñendo.

Affinely Projected Point Sprites

Por agora só conseguimos representar os splats con forma cadrada, e isto era potencialmente problemático nos visualización dos obxectos, posto que se podían apreciar os bordes dentados cerca dos límites do obxecto. Para obter unha mellor aproximación, imos intentar recrear a forma redonda que ten un splat.

Para conseguir obter un círculo, imos utilizar a mesma metodoloxía de aproximación que anteriormente, usando a perspectiva da cámara para adecuar o tamaño dos splats. O seguinte paso será identificar os fragmentos que queden fóra do radio do splat e descartalos, así obtemos a forma circular. Para levalo a cabo, temos que parametrizar o espacio que ocupan os cadrados nun rango de $[-r, r]^2$, sendo *r* o radio do splat. Para cada fragmento $(x, y) \in [-r, r]^2$, o valor de desprazamento de profundidade δz (*depth offset*) do punto **p**, que é o centro do splat, pode computarse como unha función lineal que depende do vector normal da cámara **n** = (n_x, n_y, n_z):

$$\delta z = -\frac{n_x}{n_z} \cdot x - \frac{n_y}{n_z} \cdot y$$

Unha vez calculado isto, o valor de desprazamento de profundidade utilizarase para determinar que queda dentro do radio do splat, e que queda fóra con:

$$\|(x, y, \delta z)\| \leq r$$

Un problema de "Image-aligned Squares" era que o valor de profundidade (*depth value*) era fixo, isto podemos correxilo co valor δz que acabamos de calcular. Partindo do valor de profundidade en espacio da cámara $z' = p_z + \delta z$ e máis o frustrum, pode modificarse o valor de profundidade do $zbuffer(x, y)$.

$$zbuffer(x, y) = \frac{1}{z'} \cdot \frac{fn}{f - n} + \frac{f}{f - n}$$

Esta aproximación mellora bastante as propostas anteriores, sobre todo nos bordes redondedos onde xa non se aprecia o pixelado formado polos cadrados. De calqueira xeito o valor de desprazamento de profundidade é unha estimación, polo que pode ocorrer que se aprecien pequenos ocos nas zonas cercanas aos bordes.

3.3.4 Estimación Normais

Antes de mostrar os resultados do deseño do visualizador, xa que traballaremos con métodos de rasterización que utilizarán as normais, aclararemos este concepto.

Unha **normal** é un vector que é perpendicular a un plano tanxente. Vai a ser moi útil para saber a que dirección apunta a superficie dos *splats* (3.3.2) do que están formados os obxectos, e así conseguir representacións o máis realistas e detalladas posibles.

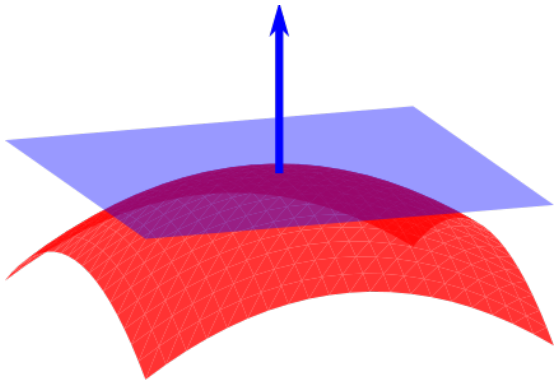


Figura 3.5: En azul o plano tanxente a un punto do obxecto en vermello, e a flecha azul indicando a dirección da normal a ese plano. [3]

Cando temos unha superficie xeométrica, como pode ser o obxecto vermello da figura (3.5), calcular a normal en certo punto da mesma é xeralmente trivial. Pero cando temos unha nube de puntos que representa unha superficie real, non é tan doado. Para poder obter a normal necesitaremos un plano tanxente ao punto, pero con un punto como única referencia non podemos obtelo. Entón, unha maneira común de abarcalo é tomando de referencia

outros puntos cercanos ou veciños do "dataset", e estimar ese plano tanxente con eles para poder calcular a normal. Máis concretamente, a solución para estimar a normal da superficie [31] baséase no analisis dos vectores propios e os valores propios [32] de unha matriz de covarianza[33] creada polos veciños do punto en concreto. Para cada punto p_i formamos a matriz de covarianza C do seguinte xeito:

$$C = \frac{1}{k}\sum_{i=1}^{k} \cdot (p_i - \overline{p}) \cdot (p_i - \overline{p})^T, \quad C \cdot \vec{v_j} = \lambda_j \cdot \vec{v_j}, \quad j \in 0, 1, 2$$

Onde k é o número de puntos veciños do punto de referencia p_i, $\overline{p}$ representa o centroide 3D dos veciños cercanos (o centroide é o punto que se estima que está máis cercano ao centro real da superficie que se forma entre os k veciños). λ_j é o j valor propio da matriz de covarianza e $\vec{v_j}$ é o j vector propio.

Para estimar unha superficie a partir dun punto, sobre a que poidamos calcular a súa normal, debemos realizar unha búsqueda de puntos veciños máis cercanos. Dependendo do número de veciños que escollamos como superficie de referencia, conseguiremos obxectos máis ou menos detallados. Se escollemos un número menor de veciños, o coste computacional será maior, posto que teremos que repetir este cálculo máis veces, pero os resultados serán máis detallados. En caso contrario, sendo o número de veciños maior, o coste computacional é menor pero o resultado das visualizacións perderá calidade tamén.

Capítulo 4

Deseño Visualizador de Nubes de Puntos

Existen algunhas ferramentas como CloudCompare [27] e Meshlab [26] que xa teñen o seu propio visualizador, pero estas ferramentas non dispoñian de todas as funcionalidades que íamos necesitar neste proxecto. Principalmente necesitabamos representar as normais das nubes de puntos, e poder comparar co visualizador as estimacións de normais convencionais coa estimación que achegamos nós con Deep Learning. Tamén buscábamos visualizar como se representan os obxectos cun método de rasterización no que as normais tomen parte, como é "Affinely Projected Point Sprites" (3.3.3). Deste xeito podemos comprobar con que normais estimadas se representarán mellor os splats, e consecuentemente, os obxectos se representarán máis fielmente ao orixinal. Aínda que en versións previas do visualizador se utilizasen outros métodos de rasterización máis sinxelos, decideuse implementar final e únicamente o método "Affinely Projected Point Sprites", pois é o único dos implementados que está directamente influenciado polas normais. Imos indagar un pouco máis en que nos ofrece o noso visualizador ao que bautizamos como **iPoint** a continuación.

4.1 Estructura e deseño

O noso visualizador está programado íntegramente en C++ (GLSL [17] nos shaders) e empregramos OpenGL 4.6 xunto coas librerías GLFW [34] e GLEW [35]. Escollemos estas tecnoloxías debido a súa compatibilidade en Linux, Windows e MacOS. GLFW proporciona a capacidade de crear e dirixir ventás e aplicacións OpenGL, e tamén permite recibir as peticións e os eventos que chegan do teclado e do rato. Con OpenGL é posible utilizar diferentes versións de drivers e de modelos, ainda que as funcións de cada driver esten escritas nunha linguaxe ou dunha maneira distinta. Isto é porque esta librería ten definido un comportamento estándar para as súas funcións, e despois os fabricantes terán que adaptar os seus drivers ao

comportamente estipulado de OpenGL. Para que isto funcione, a librería chama ás funcións que necesite dando por feito que existen, e en tempo de execución lee o código especifico de cada versión. Para que esta xestión sexa máis doada, utilízase a librería GLEW. Tamén usamos GLM [36] e PCL [28]. GLM útilizase concretamente para facilitar o uso estructuras de datos como matrices e vectores, aínda que a librería inclúe máis operacións. E a librería PCL será moi utilizada no noso visualizador, xa que nos permite cargar e xestionar as nosas nubes de puntos, e tamén dispón de métodos para a estimación de normais entre outras moitas utilidades con nubes de puntos.

Para cargar todas estas librerías e xerar os arquivos de compilación do noso proxecto, estaremos usando a ferramente CMAKE 3.15 [37]. Esta é unha ferramenta multiplataforma co propósito de xeración e automatización de código. Utilízase con entornos nativos como Make, QtCreator, Ninja, Xcode e Visual Studio.

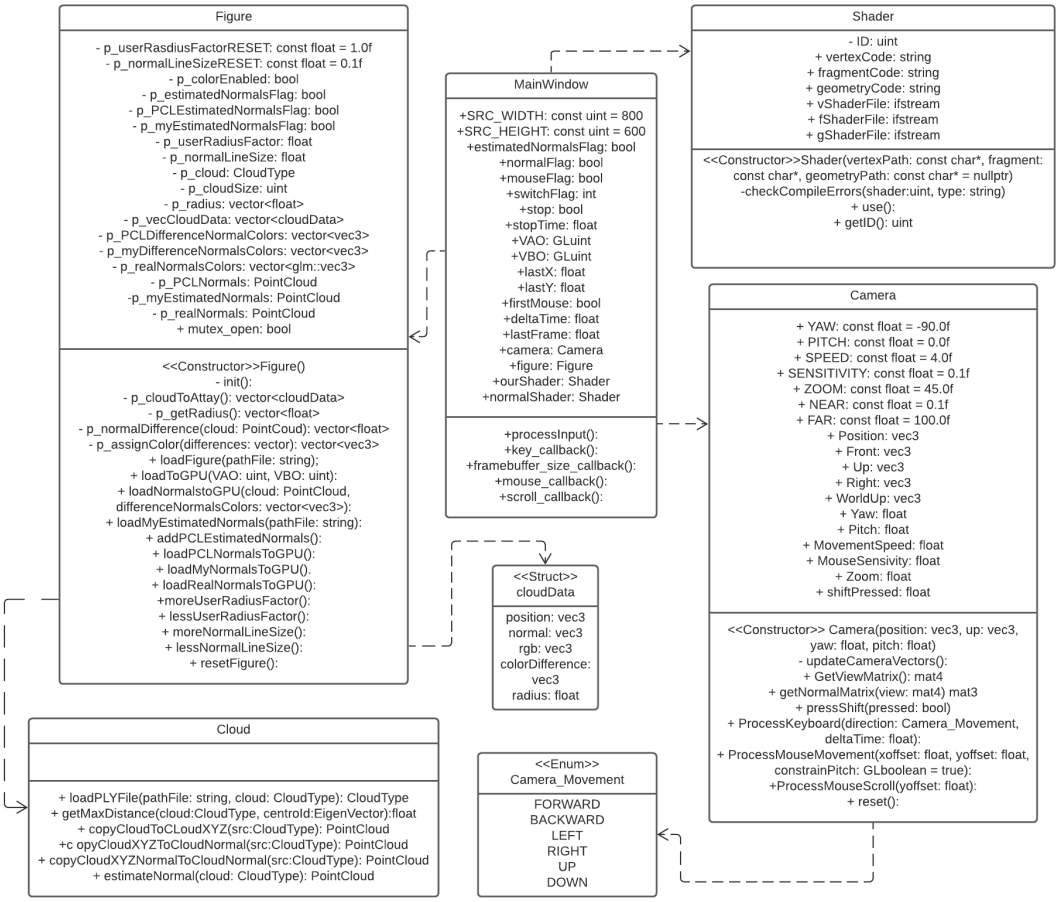


Figura 4.1: Diagrama de clases da aplicación do visualizador IPoint.

Pódemos ver na figura (4.1) o diagrama de clases de toda a aplicación do visualizador. Para que se vexa correctamente e sexa lexible houbo que simplificar algúns apartados. Por

exemplo, os tipos de obxecto que proveñen da librería GLM, que se utilizan para matrices e vectores, obviouse o nome da librería e púxose só o tipo: vec3, vec4, mat3, etc. Para as nubes de puntos, non se especificou a librería, nin os tipos de nubes de puntos exactos. Obviáronse os datos de entrada ás funcións da ventá principal, pero todos son chamadas que involucran á ventá principal e que interactuan co teclado e rato. E por último non se engadiron os "getters" e "setters", posto que todas as variables están xa representadas en cada clase. Estes datos, unha vez explicados, non deberían influír na comprensión do esquema xeral.

Agora imos investigar cada unha das clases do visualizador. A ventá principal (**Main Window**) será a encargada de dar acceso ao usuario de tódalas funcionalidades e será a que nos mostre a interfaz visual da aplicación. O "feedback" durante o uso do visualizador se verá en modo texto por terminal, será de moita axuda para saber que está a ocorrer en cada momento. Debido a este uso da ventá de visualización, xunto co texto no terminal, decidiuse utilizar unha resolución de 800x600 para que se poida adaptar a todo tipo de pantallas, e poder leer o terminal sen problemas.

Os datos dos obxectos que queremos sacar por pantalla teñen que ser cargados primeiramente na GPU mediante buffers. Unha vez cargados, coa clase **Shaders** poderáse compilar, xestionar e organizar eses datos do obxecto, e poderán engadírselle novas transformacións modificando o código nas diferentes etapas do pipeline. O código GLSL compílase en tempo de execución, polo que esta clase ten a necesidade de definir unha funcionalidade que devolva os erros de compilación e permitir así un mínimo de "debugging". Coa clase **Camera** pódese controlar a perspectiva e o movemento ao redor do obxecto. A clase **Figure** é claramente a máis completa de todas, é a encargada de traballar directamente coas nubes de puntos. Aquí podemos cargar nubes de puntos, e almacenar tódolos datos necesarios para estimar ou comparar as normais. A clase Figure conta co apoio da clase **Cloud**, que se utiliza para adaptar algunhas funcionalidades das librerías que usamos habitualmente e facer así máis fácil e concreto o uso que lles damos. Por exemplo, se para cargar unha nube de puntos temos que facer 5 chamadas distintas, e modificar certos datos, todo eso simplifícase a un só método na clase Cloud.

4.2 Funcionalidades

Xa vista a estructura do visualizador, imos falar das funcionalidades principais. Primeiro de todo aclarar que se despregará unha ventá onde se mostrarán os obxectos; e o terminal dende o que executamos, será o que nos proporcione información do que está a ocorrer mentres usamos o programa. Cando arrancas o visualizador, xa temos cargado un obxecto por defecto, polo que aínda que non teñas acceso a outras nubes de puntos, poderás probar todas as opcións dispoñibles. Pódese apreciar a interfaz inicial predeterminada na figura (4.2). Está

despregado tamén o menú de axuda no terminal, o que nos guiará entre tódalas funcionalidades posibles.

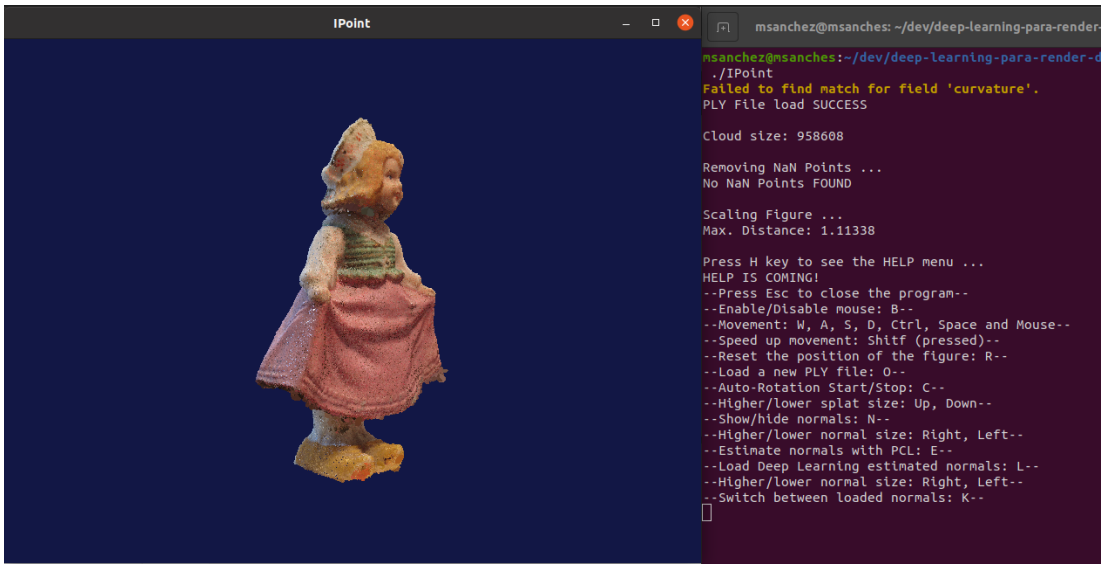


Figura 4.2: Interfaz principal de iPoint.

Se dispós de nubes de puntos, tes dispoñible unha opción para **cargar arquivos** no formato PLY. Terás que acceder ao terminal e escribir a ruta completa do arquivo para poder abrilo. No caso de que éste non exista, se notificará no terminal. Hai que comentar que a aplicación soamente acepta arquivos no formato PLY, en este formato, o arquivo é binario e ocupará menos espazo que os formatos baseados en ASCII. Se o arquivo cargado non ten cores, se lle asignará unha cor plana a todo o obxecto. E no caso de que o arquivo non dispoña de normais, terás que cargar as normais estimadas ou usar o método clásico de estimación de PCL na aplicación. Cando xa temos cargado o obxecto, podemos desprazarnos libremente ao redor del e incluso atravesalo. Podemos facer zoom no obxecto e aumentar a velocidade de desprazamento da cámara en calqueira dirección. Se perdemos o obxecto de vista en algún momento, temos a opción de resetealo e devolvelo ao centro do noso visualizador na posición inicial onde o cargamos. Se o que queremos é admirar o obxecto, podemos facer que rote indefinidamente ata que o paremos. Todas estas opcións están dispoñibles usando o noso teclado, non sendo o desprazamento xeral, que usa o teclado e máis o rato, facendo un pequeno guiño ao desprazamento normalmente usado en videoxogos. Desde que se carga un obxecto no visualizador, pódese activar a opción de mostrar os fotogramas por segundo, para coñecer a duración dos ciclos dependendo das funcións que utilicemos.

Neste traballo as funcións máis específicas e importantes para poder testear o algoritmo de estimación, son as funcións relacionadas coas normais. A propia representación do obxecto está influenciado polas normais que temos cargadas, posto que o método de rasterización

que utilizamos (3.3.3) basease no seu uso. Pero tamén temos dispoñible outra funcionalidade coas normais, podemos representalas sobre o obxecto e mirar hacia onde apuntan. As normais reais, cárganse por defecto xunto coa nube de puntos, polo que xa estarán dispoñibles para visualizar. Pero poderemos cargar tamén as normais estimadas polo noso algoritmo de Deep Learning e máis as normais estimadas mediante un método clásico de estimación. Utilizaremos o método clásico de estimación do que nos provén PCL (3.3.4). Este método xa está incluído no código da aplicación, polo que só haberá que pulsar unha tecla para realizalo. A estimación que fai PCL podería realizarse no momento que se abre unha nube de puntos, pero decideuse facer a posteriori debido a que a carga computacional pode ser moi alta para nubes moi densas, e pode semellar que o visualizador non está funcionando correctamente. Por outra banda, para incluír os datos da estimación que realiza o modelo de Deep Learning, teremos que cargar un arquivo PLY aparte. Primeiro, debemos precedir as normais dun obxecto usando o código do estimador que creamos 6 escrito en Python, e ese programa nos devolverá un arquivo coas normais estimadas polo algoritmo de Deep Learning listo para cargar no visualizador. Unha vez temos as normais cargadas no programa, poderemos comparar fácilmente os resultados que nos ofrecen as distintas estimacións. Estará dispoñible a posibilidade de cambiar rápidamente á vista das normais coa estimación que se desexe. O título do visualizador nos dará "feedback" sobre as normais que están cargadas en cada momento.

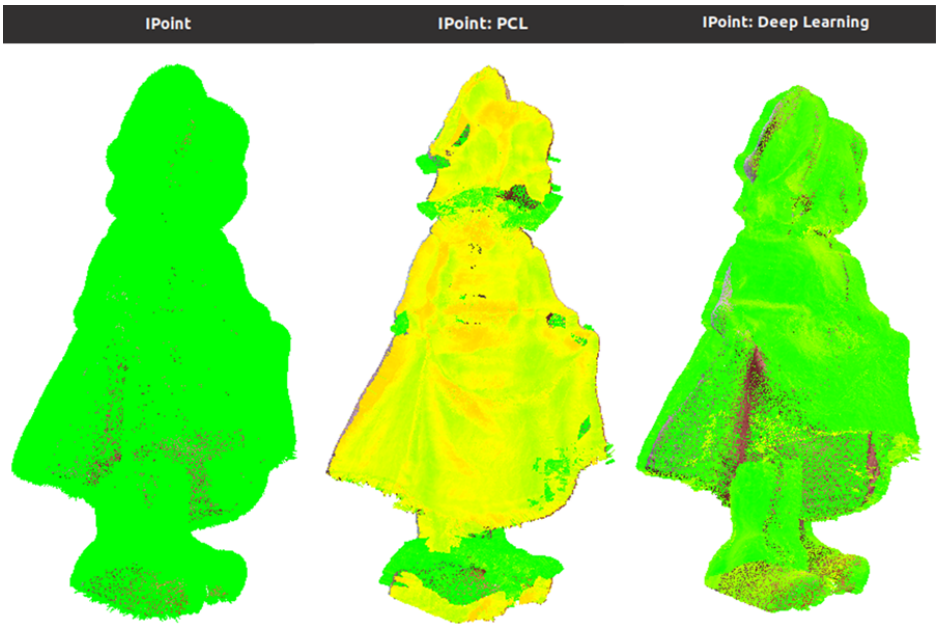


Figura 4.3: Visualización de normais reais, normais estimadas por PCL, e normais estimadas por Deep Learning.

Por último, a funcionalidade que máis nos vai a aportar se complementa coa visualización das normais. Como podemos ver na figura (4.3), as normais varian de cor en distintos pun-

tos do obxecto. Estas cores varían dende unha cor verde intensa a unha cor completamente vermella, pasando polo amarelo para as medicións intermedias. O que indican estas cores é a diferencia que existe entre as normais reais e as normais estimadas, canto máis cerca máis verde, e canto máis lonxe amarelo ou incluso vermello nos peores casos. Podemos adaptar o tamaño das normais e dos splats do obxecto en todo momento para conseguir a visualización que mellor se adapte en cada caso. Esta opción incluíuse debido a que as figuras poden proceder de fontes completamente distintas e ter métricas moi dispares.

O cálculo das diferencias entre dúas nubes de puntos realízase comparando os valores dos puntos que están na mesma posición en nubes distintas. Recollemos as coordenadas de posición de cada un dos puntos e as restamos coa súa homónima na nube de puntos coa que queremos comparar. É dicir, a coordenada X dun punto nunha nube de puntos coa coordenada X dese mesmo punto na outra, e se repite o mesmo proceso para Y e máis Z. E tamén para cada un dos puntos. Para seguir explicando o proceso, imos reducir o exemplo á comparativa dun só punto en distintas nubes de puntos. Unha vez obtemos os valores de restar as coordenadas X, Y e Z por separado, aplicamos neles a función de valor absoluto, e despois as sumamos. Obtendo un só valor para cada punto. Podemos ver os cálculos no seguinte pseudocódigo:

```
vector normalDifference(PointCloud cloud_A, PointCloud cloud_B)
{
  vector diferencias;

  for (size_t i=0; i<cloud.size();i++)
  {
    diferencias.push_back(
      abs(cloud_A[i].normal_x - cloud_B.normal_x) +
      abs(cloud_A[i].normal_y - cloud_B.normal_y) +
      abs(cloud_A[i].normal_z - cloud_B.normal_z));
  }

    return diferencias;
}
```

Con estes cálculos, obteremos un valor máximo de 6, e mínimo de 0. Escollemos este formato porque nos facilitaría os cálculos á hora de cambiar a cor, pero se adaptará cando necesitemos expresar os datos no capítulo de Resultados (7). A idea é que os valores coas diferencias dos puntos influyan directamente na función de selección de cor. Podemos velo en pseudocódigo a continuación:

```
vector assignColor(vector diferencias)
  {
    vector<vec3> colors;

    for (size_t i=0; i<diferencias.size();i++)
    {
      vec3 aux;
      if (diferencias[i] <= 3)
      {
        aux =  vec3(diferencias[i]/3, 1, 0);
      }else
        aux = vec3(1, 2-(diferencias[i]/3), 0);
      colors.push_back(aux);
    }

    return colors;
  }
```

A priori coñecemos que para obter cor verde co modelo RGB, necesitamos que o valor sexa (0 Red, 1 Green, 0 Blue). E se imos engadimos vermello gradualmente, a cor irase transformando en cor amarela. Entón a primeira condición é que se o erro é menor ou igual a 3 (a metade), a cor pode variar de verde a amarelo tanto como o porcentaxe de erro. E despois para conseguir cor vermella completamente, necesitamos que o valor sexa (1 Red, 0 Green, 0 Blue). Entón, a medida que aumenta o erro, por encima da metade o valor do verde irá rebaixandose e conseguindo unha cor vermella. Finalmente, os datos de cor cargaránse na visualización das normais do obxecto que esteamos a usar, permitíndonos observar a cantidade de erro que se produciu na estimación das normais dos puntos.

Capítulo 5

Fundamentos de Deep Learning

Rematado o bloque de gráficos, imos adentrarnos no mundo da Intelixencia Artificial, concretamente no Deep Learning. Primeiro de todo imos expoñer unhas bases teóricas sobre que é o Deep Learning, como funciona, e que características poderemos utilizar para acadar os mellores resultados posibles. A intención é que cos coñecementos que se adquiren en este capítulo, xunto coas referencias, deberían entenderse os conceptos que se tratarán máis adiante.

5.1 Intelixencia Artificial, Machine Learning e Deep Learning

A Intelixencia Artificial pode considerarse todo un conxunto (moi amplo) de disciplinas que teñen como obxectivo dotar aos ordenadores da capacidade de resolver problemas sen ter que deseñar algoritmos *ad hoc* específicos para eles, combinando as ciencias da computación coa análitica de datos e a estatística. Dentro deste campo atópase a Aprendizaxe Máquina (ou *Machine Learning*), que ten como obxectivo desenvolver técnicas que permitan que as computadoras aprendan e sucesivamente vaian mellorando na súa resolución dos problemas. Para conseguir resultados en Aprendizaxe Máquina hai que definir un modelo cunhas características de aprendizaxe determinadas, e ese modelo se alimentará de datos de entrada dos que se procura estimar un resultado, ou facer una predicción, ou atopar algunha característica común nos datos de entrada. O proceso de aprendizaxe chámase adestramento, consiste en fixar un número de iteracións nas que o modelo vai aprendendo a asimilar os datos de entrada cos resultados esperados, e cando rematan as iteracións, o modelo queda adestrado. Nese momento poderase alimentar o modelo con novos datos descoñecidos e predecir novos resultados, o que se coñece como etapa de inferencia. Un exemplo básico sería, se usamos de datos de entrada cadrados de cores, e como resultado esperado propoñemos as cores deses cadrados; unha vez temos adestrado o modelo, podemos alimentalo con cadrados dos que descoñecemos a cor e o modelo predecirá as cores dos cadrados. Este exemplo do que falamos trátase concretamente

dun modelo de *aprendizaxe supervisada*, xa que determinamos claramente cal é a saída que esperamos. Tamén poden plantexarse algorimos de *aprendizaxe non supervisada*, onde non se marca claramente a saída esperada. A *aprendizaxe non supervisada* funciona xeralmente con clústers, o que fai é tratar de recadar características comúns dos datos de entrada, e reflexar os datos por separado como saída. É útil por exemplo para cantidades de datos inmensas que están completamente desorganizadas, e para as que queremos atopar similitudes.

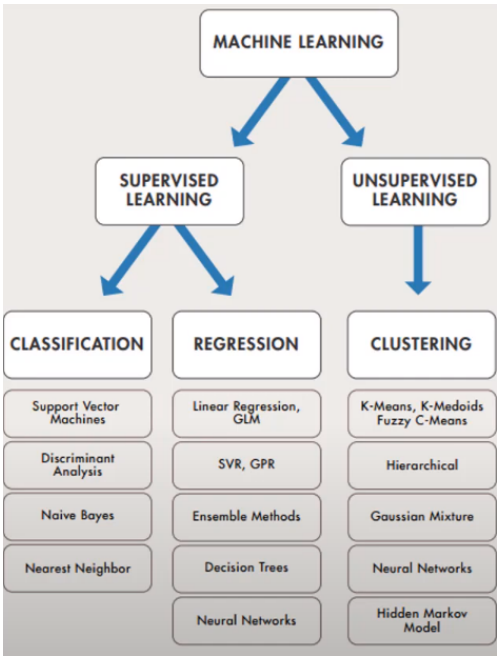


Figura 5.1: Clasificación dentro do Machine learning. [4]

Imos centrarnos neste proxecto na *aprendizaxe supervisada*, como podemos ver na figura (5.1) existen dous tipos, *clasificación* e *regresión*. Un modelo de *clasificación* é o exemplo que expuxemos previamente, no que alimentamos ao modelo con cadrados de cores, e o modelo predicirá de que cor son os cadrados. Trátase de alimentar o modelo e que aprenda a seleccionar entre unha das opcións de saída. Mentres que o modelo de regresión busca aproximarse o máximo posible a un resultado buscado. Por exemplo, se crearamos un modelo para estimar con que probabilidade se terá una boa colleita este ano, podemos alimentar o modelo cos datos de tódalas colleitas anteriores e os seus resultados. E poderemos coñecer a probabilidade de éxito deste ano.

Dentro do Machine Learning, foron aparecendo novas formas de plantexar os modelos. Utilizaremos neste traballo as redes de neuronas artificiais, que son modelos computacionais inspirados no sistema nervioso animal. Na práctica funcionan como aproximadores universais de funcións, e o xeito en que poden ir mellorando iterativamente a aproximación que ofrecen dunha función fanos moi utilizados en Machine Learning.

Co crecemento deste campo, a explosión de datos cos que traballar supuxo toda unha revolución no uso e explotación de redes de neuronas para a confección de modelos de Machine Learning, redes que pasaron a ser moi profundas (con moitas capas e de distintos tipos/funcións), dando lugar ao que se coñece como Deep Learning, subdisciplina hoxe en día dentro do Machine Learning.

5.2 Redes neuronais e redes neuronais profundas

As redes *neuronais artificiais* (ANN) serven para adestrar modelos, e predecir resultados con datos novos, igual que o Machine Learning. Pero o funcionamento interno destas varía completamente. As ANN están formadas de capas (ou "layers") que conteñen un número determinado de *neuronas*, e esas *neuronas* estan interconectadas entre si, de tal xeito que os datos de entrada teñen que pasar por elas ata dar un resultado. Existen varios tipos de ANN como Perceptrón Multicapa (MLP), Feedforward neural network (FFNN), Redes Neuronais Convolucionais (CNN), Redes Neuronais Recurrentes (RNN) e Redes Neuronais Profundas (DNN); estas últimas serán as que utilizaremos neste traballo. As DNN son redes neuronais artificiais con 2 ou máis capas ocultas (habitualmente decenas ou centenas destas capas intermedias)

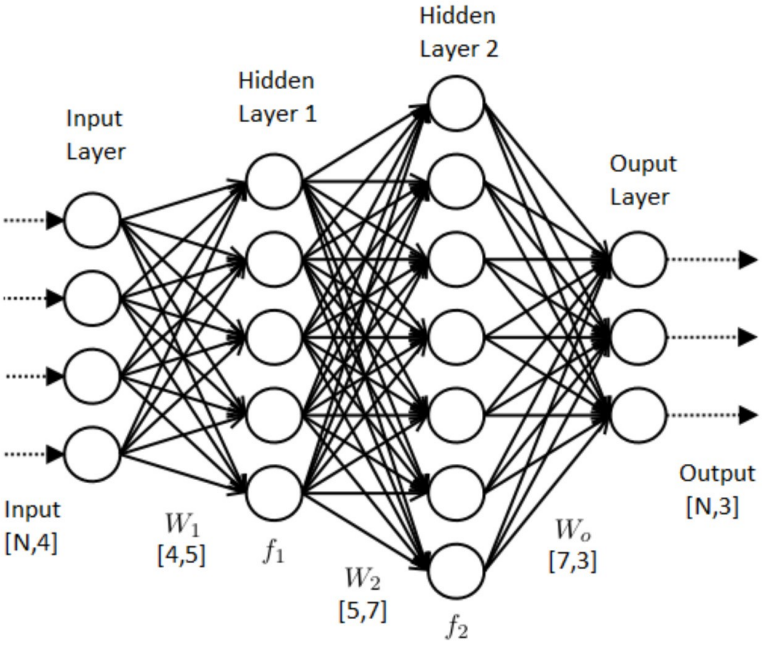


Figura 5.2: Representación dunha rede neuronal artificial. [5]

5.2.1 Funcionamento redes neuronais

Como podemos ver na imaxe (5.2) as redes neuronais reciben un input e poden pasar por distintas neuronas de cada capa ata chegar ao resultado final. Na *capa de entrada* (ou "input layer") definirase a cantidade de datos que van a entrar no modelo, e na *capa de saída* (ou "output layer") o número de saídas que esperamos. As *capas ocultas* (ou "hidden layers") sirven para recoñecer patróns nos datos que inserimos, e eses patróns serán clave para a predicción de resultados unha vez adestrado o noso modelo. Por iso, dependendo da complexidade do problema que queiramos resolver, pode variar o número de *capas ocultas*. Cada unha das neuronas que forman as capas ten un valor asociado chamado *'bias'* e un conxunto de *'pesos'* (ou "weights") que proveñen da capa de entrada ou da neurona inmediatamente anterior. Para que os datos de entrada pasen dunha neurona a outra, deben sumarse os datos do *bias* mais o resultado de multiplicar os *pesos* anteriores en cada neurona, e o resultado desa suma deberá superar os criterios da *función de activación* [38] para pasar á seguinte neurona. Se non supera eses criterios, se pasará o valor 0 ou nulo. Na seguinte figura [39] está descrito este proceso de forma gráfica:

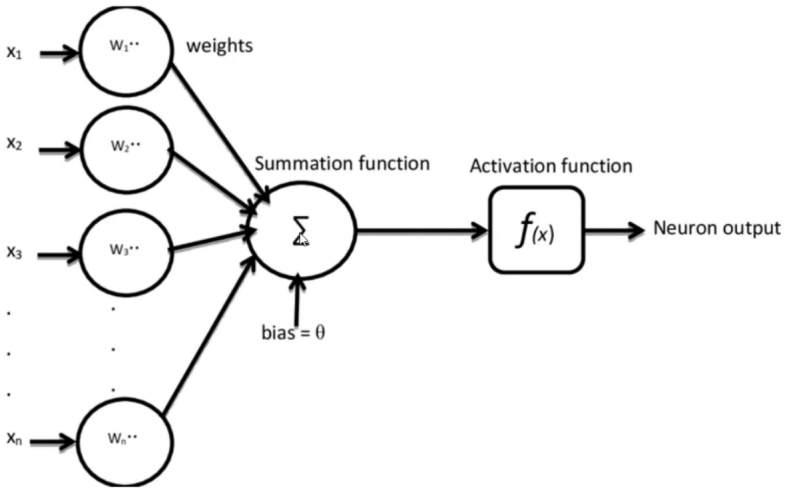


Figura 5.3: Debuxo representativo do funcionamento dunha soa neurona artificial [4]

Cando alimentas unha rede neuronal con datos, e adéstrala, na súa primeira iteración a capa de saída obterá uns resultados tendo en conta os datos de entrada, os *bias* e os *pesos* polos que pasou. Probablemente os resultados desa primeira iteración estean lonxe de conformar un modelo que cumpra co que se pretendíame dij, posto que o modelo aínda non coñece nada dos datos que acabamos de pasarlle. O modelo recollerá esa saída de datos e calculará o erro que cometemos conforme ao resultado esperado. A encargada de determinar ese erro é a *función de erro* [40], hay moitas opcións, entre elas destácanse Mean Absolute Error (MAE) e Mean Square Error (MSE), e debe escollerse unha destas funcións cando describimos o noso

modelo. Esa información de erro utilízase para recalcular os pesos nas conexións das neuronas, de tal xeito que en cada iteración vaia afinando máis e máis o resultado. Os algoritmos de optimización [41] son os encargados de reducir ao máximo ese erro producido e o proceso de retroalimentación que se dá para levar a cabo esta actualización de *pesos* chámase "backpropagation" [42].

Unha vez completadas todas as iteracións, se o noso modelo ten unha taxa de erro razoable, estará listo para predecir resultados con datos de entrada descoñecidos. Pero non é doado conseguir bos resultados a primeira vez que configuramos e adestramos o noso modelo. Temos que graficar os resultados obtidos e utilizar a información que nos aportan esas gráficas, máis a experiencia, para sacar conclusións e mellorar o noso modelo.

5.3 Funcións de activación

As funcións de activación son ecuacións matemáticas que determinan o valor de saída de cada neurona dunha rede neuronal. Recolle a información de saída da función de activación da neurona anterior (a suma do bias e dos valores de entrada multiplicados polos pesos son os que inflúen no resultado da función de activación) ou dos datos de entrada que temos introducidos no caso da primeira capa de neuronas; e ofrece unha saída, normalmente entre 0 e 1 ou -1 e 1. Estas funcións marcarán o percorrido que vai seguir un valor dende a entrada da rede neuronal ata acadar o resultado na saída. Existe unha gran variedade de funcións de activación, ainda que a máis utilizada no caso dos modelos de regresión actualmente é ReLU, e nos modelos de clasificación tende a usarse SoftMax [43]. Podemos ver algúns tipos de función de activación na figura seguinte:

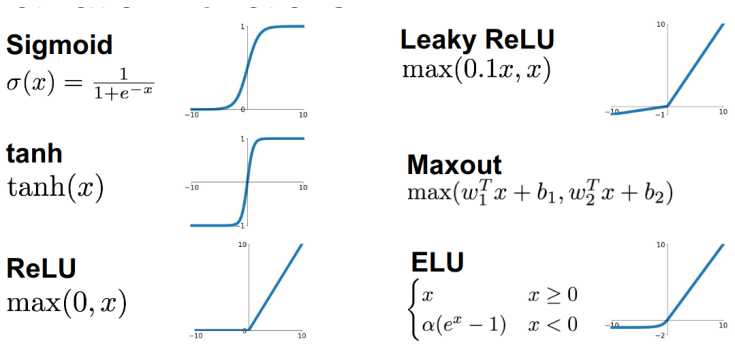


Figura 5.4: Exemplos de funcións de activación [6]

5.4 Algoritmos de optimización

Durante o adestramento dun modelo, recóllense os valores de erro comparando os resultados do modelo cos resultados esperados. Iteración a iteración realizase un estudo deses valores de erro, formando unha función que indica en que iteracións o erro foi máis pequeno ou máis grande, e que valores ten asociados ese erro. Entón, os algoritmos de optimización intentan minimizar o erro na función o máximo posible para obter os mellores resultados. O algoritmo de optimización irá variando os pesos e a taxa de aprendizaxe para esta fin.

Existen varios algorimos de optimización, o máis clásico é o *Descenso de Gradente* (ou "Gradient Descent" [8]) do cal existen moitas variables diferentes como "Stochastic Gradient Descent", "Mini-Batch Gradient Descent", "Momentum", etc. Unha característica deste algoritmo é que ten unha taxa de aprendizaxe fixa. A taxa de aprendizaxe é o un valor que indica a distancia á que pode moverse o optimizador cada vez que intenta aproximarse ao mínimo local. Irá así dando pequenos saltos ata acadar o mínimo global. Para levalo a cabo necesitamos calcular dende un punto, súa derivada, para saber en que dirección está a pendente da función, e dar un salto hacia o valor máis baixo. Repítese este proceso ata acercanos o máximo posible ao mínimo global.

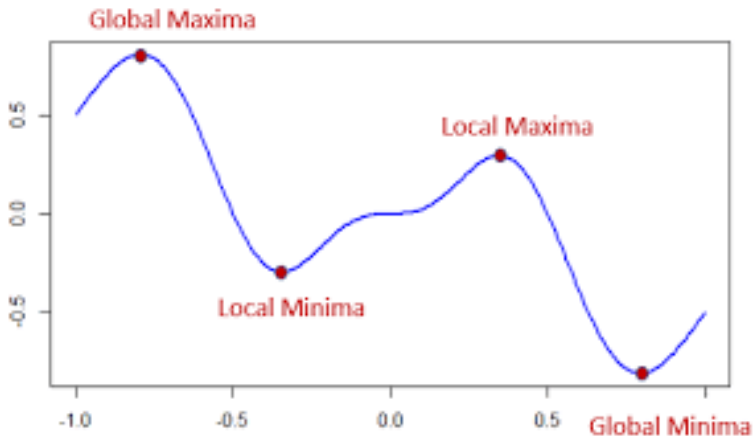


Figura 5.5: Máximos e mínimos nunha función [7]

O problema chega cando chegamos a un mínimo local (5.5), e non ao global. O algoritmo *Descenso de Gradente* intentará saltar ese mínimo local, movéndose dunha dirección á outra do mesmo, pero a taxa de aprendizaxe é o valor que determina o tamaño do salto que pode dar o algoritmo para buscar ese mínimo global. Sendo esta taxa fixa, pode ocurrir que non consiga saltar ese mínimo, e por tanto que nunca alcance o mínimo global, polo que non obteñeríamos o resultado que buscamos.

Para solventar esta problemática surxiron algoritmos novos, entre eles *Adam* [44]. Este é

un algoritmo popular en Deep Learning, porque ainda que sexa lento, consegue normalmente resultados robustos. E además, ten a taxa de aprendizaxe variable, que se adapta en cada neurona independientemente, polo que gracias ao seu diseño poderá saltar eses mínimos locais para alcanzar finalmente o mínimo global. Existen algunhas variables do algoritmo como "Nada" e "AdaMax". Na figura a continuación (5.6) podemos ver as diferencias entre algoritmos con taxa de aprendizaxe fixa e variable.

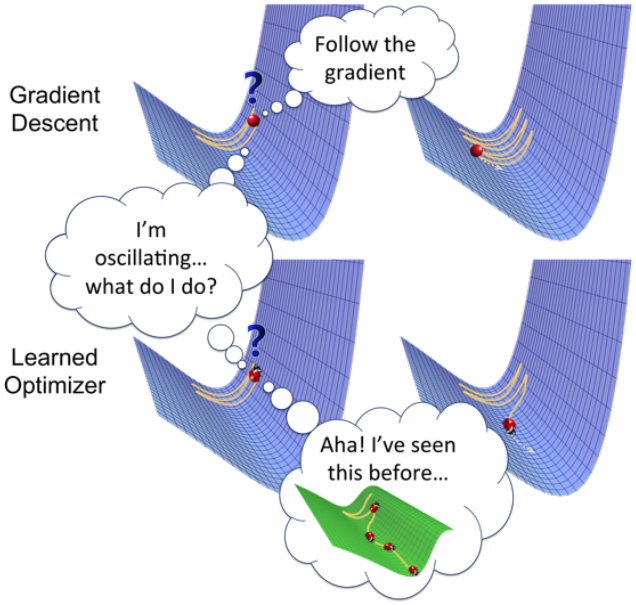


Figura 5.6: Comparación entre descenso de gradente e outros algoritmos con taxa de aprendizaxe variable [8]

5.5 Funcións de erro e de custo

Outra ferramenta que nos axuda a atopar o noso modelo ideal é a función de erro [40]. Cando obtemos o valor de erro ao final de cada iteración o encargado de recollelo será a función de erro. Mentres imos avanzando nas iteracións, utilízase a función de erro para calcular típicamente unha media de todos os datos de erro que se van engadindo en cada iteración. O resultado tras obter a media de todas as funcións de erro chámase función de custo. Aínda que existe unha diferencia clara entre función de erro e función de custo, nos textos adóitase referir sempre como función de erro referíndose á función de custo [45]. Hai moitas formas de achar a media, dependendo da función de custo que escollamos obteremos erros máis altos ou máis baixos. Por exemplo, para modelos de regresión utilízanse as funcións "Mean Square Error" (MSE) e "Mean Absolute Error" (MAE). Mentras MSE devolverá uns datos bastante baixos, posto que para calcular o erro utiliza as raíces cadradas, o mesmo problema pode devolver un erro maior se utilizamos MAE, onde se usan os valores absolutos do erro para atopar a me-

dia. En problemas de clasificación se utilizan outras funcións de erro, como "Cross Entropy Loss" e "Multi-class SVM Loss".

5.6 Underfitting e Overfitting

Depende moito do problema e da cantidade de datos de entrada no modelo, pero normalmente os adestramentos en Deep Learning levan unha cantidade de tempo longa, dende días a meses. A cantidade de datos que recibiremos será moi grande tamén e necesitaremos de moitas iteracións para conseguir bos resultados. Entón, unha vez rematado o adestramento, para poder visualizar tódolos datos rápidamente o máis cómodo é construír gráficas con eles, e sacar conclusións usando esas gráficas. Cabe dicir tamén que, cando adestras un modelo, un porcentaxe dos datos que envias a adestrar terán que ser datos de validación. Estes datos son exemplos coa mesma estructura que os datos de adestramento e se utilizan para constrastar a mellora do modelo no tempo. É dicir, intenta facerse unha predicción cos datos de validación en cada iteración do modelo, para ver como de fiable é o noso modelo iteración a iteración. Xeralmente nos interesa consultar unha gráfica na que poidamos ver reflexados os valores de taxa de erro e de taxa de erro de validación por iteración, para sacar conclusións futuras sobre a evolución do noso modelo. Coa axuda destas gráficas, e cos resultados obtidos de predicións do modelo, poderemos tomar decisións sobre como mellorar o noso modelo para obter mellores estimacións.

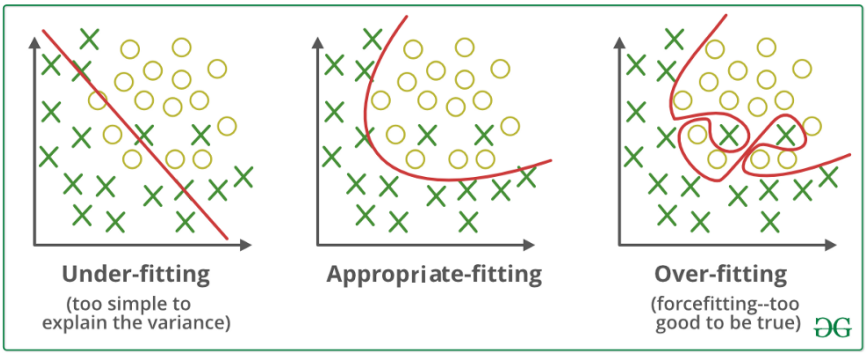


Figura 5.7: Exemplos de underfitting, overfitting e o adestramento apropiado [9]

Unha das problemáticas observables gráficamente son as de *subaxuste* (ou "underfitting") e *sobreaxuste* (ou "overfitting"). O *subaxuste* dase cando a rede non é capaz de aproximar os detalles finos na estructura dos datos de entrada, isto pode ser por non ter suficientes datos para poder achegar o resultado correcto. Polo que o modelo devolverá resultados aleatorios e non será capaz de recoñecer patróns de predicción. O *sobreaxuste* pola contra, adáptase demais aos datos de adestramento, e despois non é capaz de xeralizar. Esto quere dicir que o modelo ofrecerá boas prediccións cos datos de adestramento, pero cando se alimente con

datos novos, o modelo non será quen de identificalos. So recoñocerá o conxunto de datos de adestramento. Pódense ver as diferencias e cal sería o punto intermedio entre *subaxuste* e *sobreaxuste* na figura (5.7).

Capítulo 6

Deseño Estimador de Normais

Cando traballamos con modelos de Machine Learning ou Deep Learning, conseguimos evitar ter que programar exactamente como vai funcionar o algoritmo, o que facemos é propoñer un modelo, baseado nunha rede de neuronas artificiais no noso caso, e axustar a configuración dos parámetros que inflúen no modelo. A priori non podemos coñecer a configuración óptima de parámetros para o noso modelo, xa que cada problema pode solucionarse de un xeito distinto, e incluso o mesmo problema pode solucionarse de maneiras distintas. Por tanto, teremos que utilizar unha metodoloxía de proba-erro, na que iremos variando varias características e parámetros ata conseguir modelos de aprendizaxe robustos. Este proceso, que se coñece como axuste de hiperparámetros, e a súa optimización é obxecto de numerosas liñas de investigación hoxe en día.

6.1 O modelo de aprendizaxe

O problema a tratar neste traballo será unha estimación de valores, e non dunha clasificacións dos mesmos en distintas categorías, polo que podemos concluír que traballaremos cun modelo de regresión (5.1). Cando se crea un modelo de regresión con Deep Learning, haberá características comúns que se definen en todos os modelos, como pode ser que terá uns datos de entrada, e uns datos de saída. Trátase de alimentar o modelo con datos dos que se vai a predecir un resultado, e tamén cos resultados esperados para os datos de entrada. Deste xeito o modelo, unha vez adestrado, cando reciba datos novos e descoñecidos como entrada, poderá predecir os resultados dos mesmos. Outra cousa que teñen en común os modelos é que están compostos por un número determinado de capas, que pode incluír capas ocultas ou non. Estas capas están formadas á súa vez por un número determinado de neuronas. Utilizando distintas variacións do número de capas e de neuronas que conforma a estructura do modelo, poderemos dar solución á predicción de modelos diferentes. Tamén dependerá das funcións de activación que se apliquen en cada capa para obter uns resultados ou outros. Sempre está

presente un método que permite adestrar o modelo, no que se poderá variar o número de iteracións que necesitará o modelo para ofrecer resultados robustos, máis o tamaño dos datos que poden recollerse en cada iteración. E finalmente, unha vez o modelo está adestrado, necesítase un método para predecir resultados dos datos de entrada novos e descoñecidos.

Concretamente, o que buscamos co noso modelo é estimar as normais dunha nube de puntos, e para conseguilo utilizaremos todas as características comúns dun modelo de regresión. Pero teremos que ir modificando as características e variables do mesmo para conseguir resultados óptimos e robustos na estimación. A medida que vaiamos axustando os parámetro de aprendizaxe, o algoritmo irá conseguindo mellores estimacións.

Un modelo de Deep Learning aseméllase ao do pseudocódigo seguinte:

```
1
2  ## Modelo de regresión.
3
4  # Cargamos os datos de entrada.
5  def load_inputs():
6      # Datos dos que predeciremos as normais.
7      X=load(arquivo)
8      # Resultados esperados das normais que estimaremos.
9      Y=load(arquivo)
10
11     # Separamos os datos de entrada que usaremos para adestrar, dos
       que usaremos para comprobar se o adestramento está dando bos
       resultados.
12     X_train, X_test, y_train, y_test = train_test_split(X,Y)
13
14     return X_train, X_test, y_train, y_test
15
16 def create_model()
17     model=Modelo de regresión
18     model.add(capa cos datos de entrada)
19     model.add(capa oculta)
20     model.add(capa oculta)
21             ...
22     model.add(capa oculta)
23     model.add(capa de saída)
24     model.compile()
25
26     return model
27
28 def fit_model(X_train, X_test, y_train, y_test)
29     history=model.fit(X_train, y_train, batch_size, n_epochs,
       validation_data=(X_test, y_test))
```

```
30
31      return history
32
33  def plot(history, epochs)
34      loss = history.history['loss']
35      val_loss = history.history['val_loss']
36      plot(loss, val_loss, epochs)
```

Neste pseudocódigo do modelo podemos ver como primeiro cargamos os datos de entrada para adestrar o modelo, que serán as posicións dos puntos da nube de puntos e as súas normais reais. Parte deses datos terán que utilizarse para adestrar o modelo, e outra parte para validar o adestramento do mesmo. Seguidamente definiremos as características do propio modelo. Crearemos un modelo de regresión no que poderemos ir modificando o número de capas ocultas, e o número de neuronas que as compoñen. Á hora de crear o modelo tamén debemos definir cal será a función de activación de cada capa, o algoritmo de optimización e máis o calculo do erro. Probando distintas combinacións destas variables, obteremos resultados diferentes que nos permitirán perfilar o modelo ata que consigamos unha aproximación que se adecue ao que buscamos. Unha vez creado o modelo, temos que adestralo, neste punto decídese o número de iteracións (`n_epochs`) e o tamaño de datos máximo que pode leer en cada iteración (`batch_size`). Finalmente necesitaremos unha función na que se debuxen os resultados nunha gráfica para poder contrastar as variacións dos parámetros do modelo.

Concretamente, a metodoloxía de traballo que se utilizou para conseguir elaborar o modelo definitivo de estimación de normais, foi a de establecer unhas características para conformar un modelo base, e despois ir axustando estas características tomando como referencia os resultados e a experiencia. Tras unhas probas preliminares estableceuse que se utilizaría una estructura de 3 capas ocultas, de 128, 64 e 32 neuronas, debido a que a complexidade deste algoritmo aportaba resultados axeitados con esa estructura de capas. O número de capas e neuronas está relacionado directamente coa complexidade do problema, por iso se o problema a solventar é moi sinxelo, con menos capas e neuronas pódense obter mellores resultados. E en caso contrario será necesario ter unha estructura de capas máis complexa. Tamén escolleuse como función de activación ReLU, por ser a utilizada máis habitualmente. Inicialmente fixéronse probas cos algoritmos de optimización RMSProp e ADAM. RMSProp conseguía reducir o erro moi rápidamente, pero a medida que avanzaban as iteracións da aprendizaxe, a gráfica de erro desestabilizábase por completo. Mentres que con ADAM os resultados non melloraban tan rapidamente, pero a mellora do adestramento tiña unha tendencia de crecemento. Por isto, determinouse que ADAM sería o que utilizáramos. Con este algoritmo tardaríamos máis iteracións en conseguir os resultados, pero estes serían máis robustos. Ademáis o algoritmo dispón dunha taxa de aprendizaxe adaptativo que se adecúa ás necesidades do modelo en cada momento, e a maiores, esa taxa se pode configurar e personalizar en cada instante

do aprendizaxe baseándose en parámetros do modelo ou da experiencia. Deste xeito pódese programar a taxa de aprendizaxe adaptándose a todas as necesidades do modelo que esteamos a adestrar.

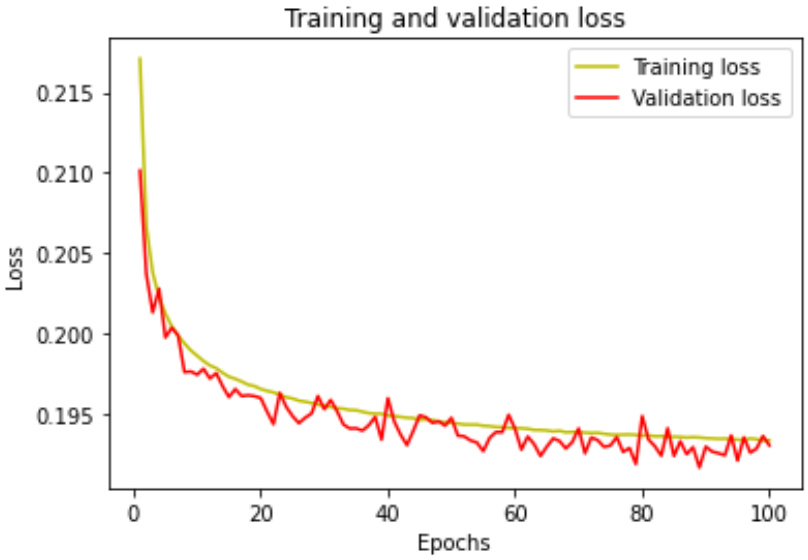


Figura 6.1: 3 capas (128,64,32), MSE, ADAM, 100 epochs.

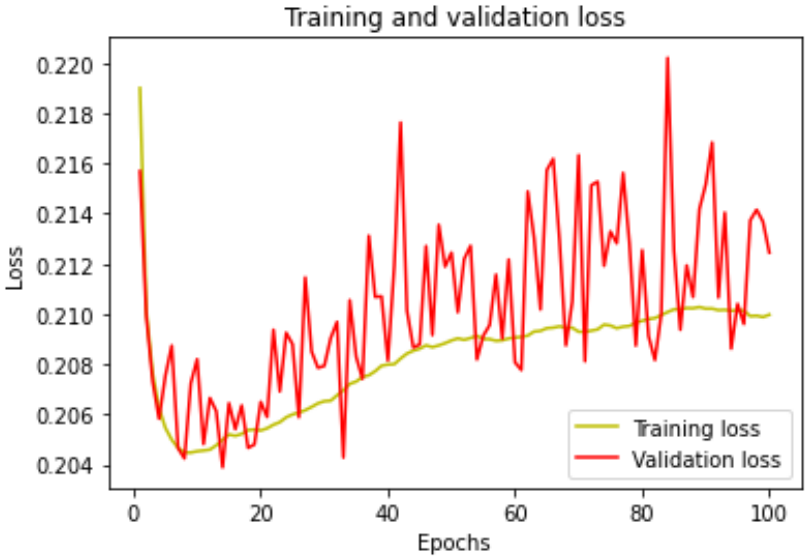


Figura 6.2: 3 capas (128,64,32), MSE, RMSProp, 100 epochs.

Nas figuras (6.1) e (6.2) pode verse como os resultados evolucionan favorablemente co algoritmo ADAM a medida que se realizan máis iteracións, mentres que con RMSProp á gráfica descontrólase. Á hora de escoller unha función de erro, interesaranos escoller a función MAE sobre MSE (5.5). Isto é debido a que a función MSE aplica a raíz cadrada no seu cálculo, e

consegue un erro menor que MAE, pero o que pretendemos é obter un dato do erro o máis cercano da realidade posible, sen calculos que reduzan o valor real do erro. Desta forma poderemos saber con máis precisión se a mellora do modelo que se está adestrando é máis notable.

As probas e variacións no modelo fanse cun número inferior de iteracións, xa que o adestramento pode levar desde horas, ata días ou semanas. E cunha porción de adestramento suficientemente grande podemos extrapolar se os resultados mellorarán ao aumentar o número de iteraccións e estamos configurando correctamente o modelo. Unha vez atopado un modelo prometedor, o ideal é engadir un número de iteracións máis notorio, como pode ser de 5.000 iteracións, e así é como obtivemos o modelo definitivo.

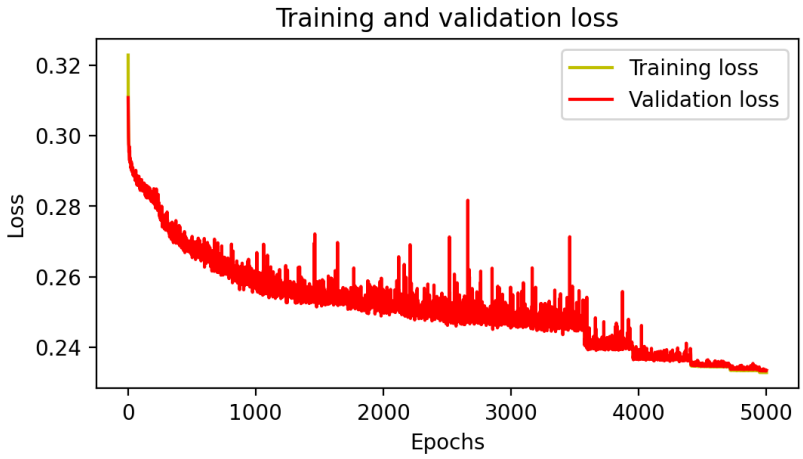


Figura 6.3: 3 capas (128,64,32), MAE, ADAM, 5000 epochs.

Pódese apreciar na figura 6.3 como a aprendizaxe foi mellorando de forma lineal ata pasadas as 3.500 itercións onde comezan a marcarse pequenos saltos de mellora por intervalos. Isto foi debido a que se utilizou unha taxa de aprendizaxe adaptativo, que no momento no que a melloría do modelo estivera estancada un determinado número de iteracións, este reduciría o valor desa taxa. Deste modo, o modelo non se estanca e consigue converxer e rematar o adestramento con valores de erro máis baixos.

6.1.1 Conxunto de datos de adestramento e formato dos datos de entrada

Cando se fixeron as primeiras probas para adestrar o modelo de Deep Learning, os datos se pasaban tal e como chegaban das nubes de puntos, e os resultados no adestramento non eran moi alentadores. Decidiuse entón presentar os datos dunha forma na que o modelo puidera asimilar mellor que era o que se lle estaba pedindo que estimara. Para isto, os datos están formatados de tal xeito que cada entrada de datos está composta polo punto do que queremos estimar a normal, e os n veciños [46] máis cercanos a el a continuación, como información

adicional. Estes datos se proporcionaran ao modelo como "predictors" ou datos dos que se espera predecir un resultado. E como datos de "target", ou resultados esperados, pasáranse directamente os valores das normais reais de cada punto da nube de puntos, unha normal por punto, e por tanto por fila. A idea de presentar asi os datos provén dos algoritmos de estimación de normais tradicionais, onde, para calcular a normal dun punto, tómanse como referencia n veciños desde punto, formando así unha superficie da que será posible calcular a dirección da normal. Os datos das normais xa están "normalizados" de por si, pero os datos de posición de cada punto poden ter valores completamente dispares, polo que haberá que normalizalos. Isto débese a que cada nube de puntos pode ter puntos en posicións en rangos de valores máis amplos e pouco acotados. E se os datos non se normalizan e limitan nun rango idéntico para todos os puntos, costará moito máis traballo ao modelo xeneralizar as prediccióǹs. Na figura 6.4 está representado o formato que tería cada unha das liñas dos datos de entrada do modelo, partindo dun arquivo cunha nube de puntos.

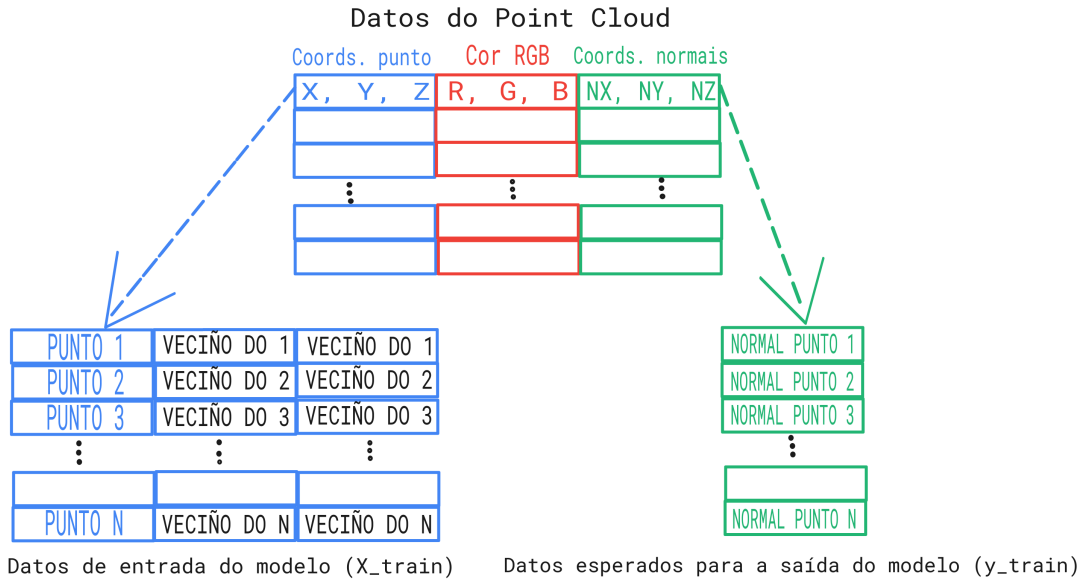


Figura 6.4: Esquema do formatado X_train ("predictors") e y_train ("target") para alimentar ao modelo.

Para formar o dataset co que alimentamos o modelo, debemos ter en conta que a variedade nos axudará a recoñecer un maior espectro de datos de entrada, conseguindo que o modelo se adapte máis facilmente a calquera nube de puntos. E debido a que os modelos con redes neuronais melloran indefinidamente cantos máis datos procese no entrenamento, tamén nos beneficiará que o noso dataset sexa o máis grande posible. Por isto, o dataset que utilizamos co modelo incluía nubes de puntos cunha variabilidade de entre uns poucos miles de puntos ata case 2 milleiros, facendo unha suma final de puntos de **4.822.996**. E o arquivo final con todas as entradas procesadas según comentabamos anteriormente, tiña un peso de **2.52 GB**

6.2 Arquitectura do sistema

Ata agora falamos do modelo que usamos para adestrar o noso algoritmo de estimación de normais. Unha vez temos o modelo adestrado, o que nos vai interesar é probar a súa eficacia. Para programar o estimador de normais utilizaremos a mesma linguaxe que utilizamos para adestrar o noso modelo, Python 3.8. Faise uso doutras ferramentas como Tensorflow 2.4.1 [47], que ten incrustada a libraría Keras[48]. Esta libraría simplifica a programación e o manexo dos modelos de Deep Learning, sendo unha linguaxe de máis alto nivel que Tensorflow, e permitindo conseguir funcionalidade con moitas menos liñas de código. Usaremos o paquete scikit-learn 0.23.2 [49], que aporta varias funcionalidades interesantes para traballar con modelos, como pode ser normalizar os datos ou transformar os datos de entrada de tal forma que se adapten ao modelo de aprendizaxe. Tamén se fará uso de numpy 1.19 [50] e pandas 1.1.0 [51], para traballar comodamente con grandes cantidades de datos e poder realizar transformacións e formatado dos mesmos de forma doada. Para poder ver representadas as gráficas cos resultados do modelo ao largo do tempo, utilízase a libraría Matplotlib 3.3 [52]. Finalmente, tras obter a estimación de normais por parte do algoritmo de Deep Learning que estamos a adestrar, necesitamos convertir eses datos novamente nun arquivo PLY válido, e isto faise usando a libraría PyntCloud 0.1.4 [53].

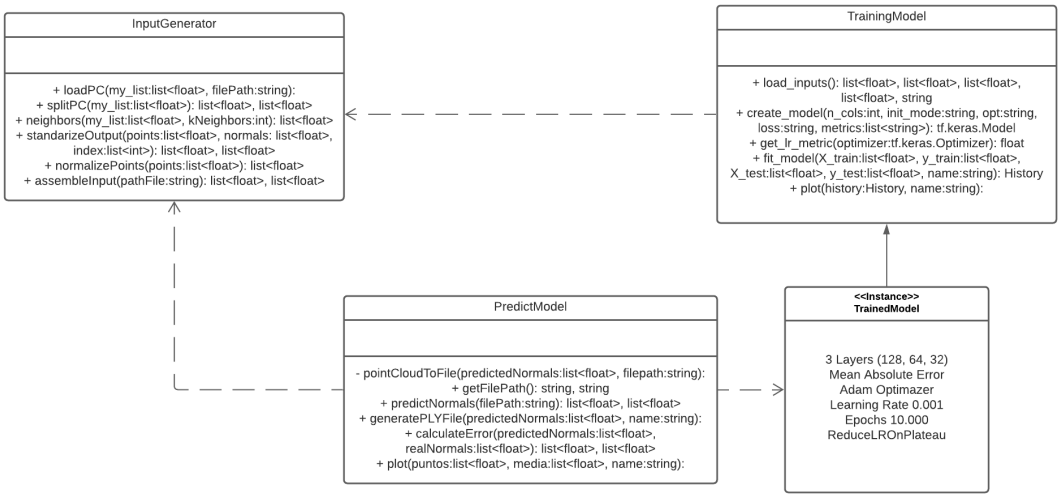


Figura 6.5: Diagrama de clases da aplicación encargada de xerar as diferencias entre as normais reais e as normais estimadas polo noso algoritmo de Deep Learning.

A clase **TrainingModel** utilízase para darlle forma ao modelo, permite cargar os datos de entrada, modificar varias características e configuracións do modelo para obter distintos resultados e tamén nos permite graficar os resultados tras ter o modelo adestrado. Esta clase non se utiliza directamente na utilidade final que desenvolvemos, pero si que foi a encargada

de xerar o noso modelo adestrado final co que estimaremos as normais das nubes de puntos. A clase **InputGenerator** utilízase para "dixerir" os datos de entrada das nubes de puntos. Isto debeuse ao que comentabamos no apartado 6.1.1, onde se veia que o formato de entrada típico dunha nube de puntos non daba bos resultados para adestrar o modelo, pero organizando á entrada dun xeito no que o modelo recoñeza mellor os datos podían conseguirse mellores resultados. Os métodos desta clase utilizaranse para transformar os datos das nubes de puntos das que se quere estimar as súas normais, polo que calquera nube de puntos nova que chegue como dato de entrada sufrirá varios cambios ata adaptarse a un formato que o modelo adestrado poida recoñecer. Finalmente na clase **PredictModel** primeiro recíbese un aquivo PLY cunha nube de puntos, e utilízase a clase **InputGenerator** para convertir e transformar eses datos, de tal xeito que se poidan utilizar na clase. Cárgase tamén o modelo adestrado que se vai utizar para estimar as normais da nube de puntos, e unha vez obtidos eses datos, necesítase volver a transformar eses datos coa estructura dunha nube de puntos no formato PLY. Esta clase tamén ten métodos de calculo do erro entre as normais reais e as normais estimadas, e permite ver gráficas coa diferencia media destes datos en varios rangos de puntos dunha nube de puntos.

6.3 Funcionalidades

O estimador de normais que creamos recibe o nome de **NormalGenerator**, e a súa principal función será calcular as normais estimadas dunha nube de puntos. Interesa que esas estimacións poidan cargarse no noso visualizador de nubes de puntos, e poder así comparar os resultados entre as normais reais, as estimacións tradicionais e as novas estimacións realizadas polo noso modelo adestrado. Entón, as funcionalidades principais serán, primeiro cargar os datos de posición e normais dos puntos dunha nube de puntos dende un arquivo PLY. O nome de arquivo de entrada será determinante para atopar o nome do arquivo de saída coas normais estimadas, dado que se chamará do mesmo xeito pero co texto *"predicted" + nome do arquivo.ply*. Unha vez ingresada a ruta dos datos de entrada, será o propio programa o encargado de transformar a entrada de datos, de tal xeito que o modelo de Deep Learning que temos adestrado poida estimar as normais da nube de puntos. Unha vez se realizou a estimación de normais, utilizando a libraría PyntCloud, devolveránse os datos das normais estimadas a un arquivo co formato PLY. Isto é debido á integración co visualizador de nubes de puntos no que haberá que cargar un arquivo PLY para ver os resultados á hora de debuxar obxectos coas normais estimadas; e máis a comparar estas normais coas orixinais do obxecto, e máis as estimadas mediante PCL. Outra utilidade da que se dispón é a de representar en gráficas a diferencia media das normais nos puntos da nube de puntos. Podendo consultar as diferencias entre diferentes conxuntos de puntos con un só vistazo.

Capítulo 7

Resultados e rendemento

Tras ter entendidas as funcionalidades do visualizador **IPoint** e do estimador de normais **NormalEstimator**, é interesante ver os resultados que acadamos facendo comparativas. Para ver si sería viable utilizar un modelo de Deep Learning para resolver o dilema da estimación de normais, analizaremos tanto en imaxes co visualizador IPoint, como con datos que obtemos ao realizar esas predicciónes.

As características do equipo no que realizaremos as probas son as seguintes:

CPU: Intel Core i5-9400F 2.90 GHz (6 CPUs)
RAM: Corsair Vengeance Pro 16 GB DDR4 3200 MHz CL16
Tarxeta gráfica: Nvidia GeForce GTX 1060 3GB
Sistema Operativo: Ubuntu 20.04 LTS (64 bits)
Versión kernel: 5.8.0-50-generic
Versión TensorFlow: 2.4.1

7.1 Cálculo de fotogramas por segundo

Aproveitando a utilidade onde se mostran os fotogramas por segundo do visualizador, obtouse por gardar un rexistro de todos os valores aos que se mantiveron os fotogramas por segundo, desde que se arranca o programa ata que se cerra. Ao observar esta métrica durante o uso do visualizar, o valor medio que atopamos ronda os 48-50 FPS.

7.2 Comparativa de tempos de execución

Na seguinte comparativa entre varios arquivos PLY con nubes de puntos, analízanse nubes cun abanico amplo de densidades para estudar o seu comportamento mediante o método

de estimación clásico de PCL e o noso método de estimación de Deep Learning. Os datos empregados para a análise son o tamaño do arquivo, o número de puntos que posúe a nube e as medicións en milisegundos da execución de cada un dos métodos de estimación empregados.

	Archivos de nubes de puntos (.ply)							
	suzanne30k	suzanne100k	hand	lucy	suzanne500k	blade	suzanne1M	head_ten24
Tamaño arquivo	0,6419 MB	2.1 MB	16,4 MB	9.5 MB	11,4 MB	44.1 MB	23,2 MB	33 MB
Número de puntos	26.734	87.972	327.323	397.664	473.675	882.954	968.643	1.223.693
Estimación PCL	60 ms	364 ms	6543 ms	8147 ms	5734 ms	13348 ms	22960 ms	15356 ms
Estimación Deep Learning	226 ms	2068 ms	2525 ms	3378 ms	3683 ms	6888 ms	7682 ms	9444 ms

Figura 7.1: Táboa cos datos de execución dos métodos de estimación de normais de PCL e do modelo Deep Learning.

Pódese apreciar na táboa da figura 7.1 que, a medida que unha nube ten máis puntos, os tempos de execución van crecendo en ambos casos. Tamén vemos que para nubes moi pouco densas a estimación de PCL ofrece un maior rendemento, mais a medida que aumentamos o tamaño da nube o tempo de execución crece rápidamente.

Obsérvanse algúns valores atípicos, no caso de *"suzanne500k.ply"* e *"head_ten24.ply"*, que non seguen a progresión de crecemento que segue o número de puntos. Isto pode ser debido a que os puntos están a menor distancia, e polo tanto é menos costoso realizar o cálculo de veciños, ou tamén pode que esté influenciado polo tamaño dos arquivos.

A execución do estimador do modelo adestrado ofrece uns valores maiores en nubes de puntos pouco densas, pero máis estables a medida que crece. O tempo de execución aumenta lixeiramente dependendo do número de puntos que teña que estimar. En nubes de puntos moi densas o estimador de Deep Learning ofrece uns valores considerablemente mellores.

7.3 Comparativa da calidade de visualización

O seguinte paso é ver os resultados no visualizador, para que os resultados sexan fiables e ver que o noso modelo está facendo boas estimacións, usaremos nubes de puntos que non se utilizaron no adestramento. Estas nubes se corresponden cos arquivos *"hand.ply"* e *"blade.ply"*. Como xa sabemos, o método de rasterización que estamos utilizando no visualizador (Affinely Projected Point Sprites 3.3.3), explota o uso das normais para representar os obxectos definidos nas nubes de puntos.

Veremos representados os obxectos completos e despois achegarémonos aos bordes para apreciar como se comportan os splats. Compararemos a visualización de cada obxecto coas

normais reais, coas normais estimadas por métodos clásicos (PCL). E finalmente coas normais estimadas polo modelo de Deep Learning que desenvolvemos neste proxecto.

Figura 7.2: Plano xeral do obxecto "blade.ply". Comparativa de métodos de estimación.

Na figura 7.2 vemos como a representación é moi similar en tódolos casos. Neste caso o modelo de Deep Learning non calculou correctamente nalgunhas zonas e poden observarse ocos e espazos baleiros. De calqueira maneira, estes erros non son esaxeradamente significativos. Todas as opcións mostran o obxecto cunhas carasterísticas moi similares.

Figura 7.3: Plano cercano do obxecto "blade.ply". Comparativa de métodos de estimación.

Na figura 7.3 facemos zoom a unha das zonas máis accidentadas do borde da figura para poder apreciar ben a súa formación. Observando o obxecto de cerca, a forma é practicamente igual nos 3 casos. É certo que no método de Deep Learning hai menos superficies aparentemente rectas e asemella que todo o borde estivera debuxado a pincel, con rugosidades. Pero estas non son moi esaxeradas.

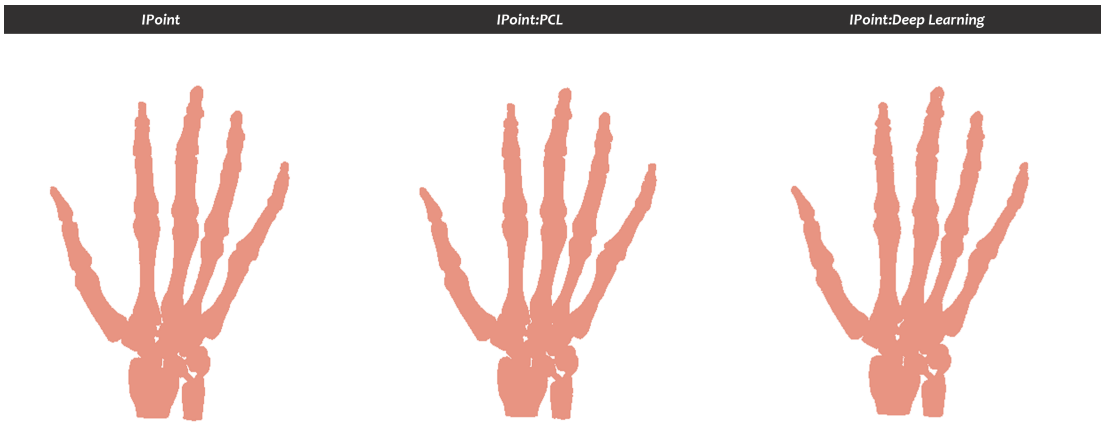


Figura 7.4: Plano xeral do obxecto "hand.ply". Comparativa de métodos de estimación.

Na figura 7.4 os 3 exemplos parecen completamente idénticos. Incluso a nivel de densidade de puntos e nos bordes, semellan estar representados polo mesmo modelo.

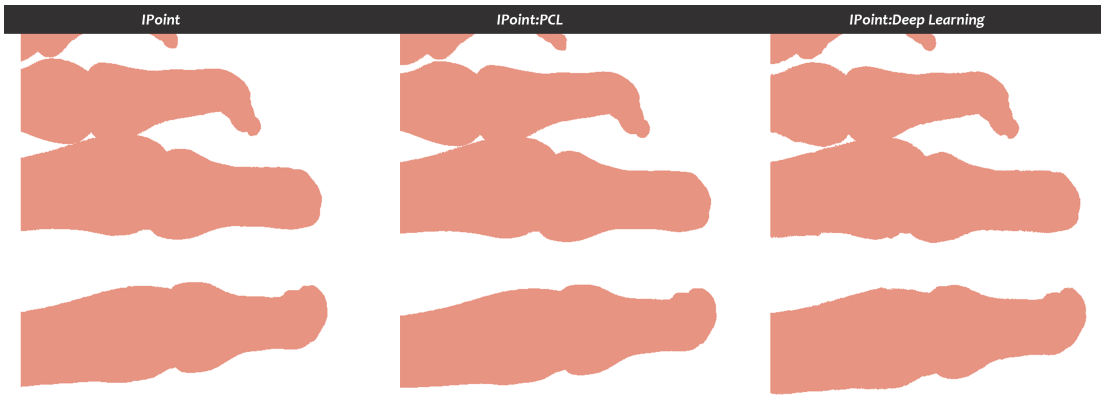


Figura 7.5: Plano cercano do obxecto "hand.ply". Comparativa de métodos de estimación.

Na figura 7.5 apenas se aprecian diferencias, pero no caso da estimación de Deep Learning, existe un lixeiro ruído que rodea as superficies do obxecto, facendo que os bordes non sexan completamente rectos. Isto é un ruido moi pouco pronunciado, que non inflúe apenas na representación dos obxectos.

Figura 7.6: Plano cercano do obxecto "head_ten24.ply". Comparativa de métodos de estimación.

Engadimos a figura 7.6, ainda que esta formara parte do conxunto de datos do modelo, debido a que a visualización asemella máis realista que nos anteriores casos, e se poden apreciar outros detalles. Xa se ve no obxecto orixinal existen algunhas zonas pequenas con erros, que pode ser debido a que o escaneado non foi perfecto. Estes erros vanse transladar aos outros métodos de estimación, debido a que forman parte do modelo. Pero podemos ver que varian dependendo da estimación de normais escollida, isto débese aos distintos ángulos que adoita a normal. Neste exemplo, tanto o obxecto orixinal coma o estimado por PCL son practicamente idénticos. Aprecianse uns puntiños negros do erro de escaneado no pescozo, e tamén se ve o nariz, os ollos e os beizos un tanto difuminados. Polo resto os bordes están ben definidos en xeral. En cambio, na estimación de Deep Learning, modifícase o patrón de erros no pescozo, debido a que as normais deben apuntar a unha dirección distinta, e cubrir esos erros. E a zona do nariz, ollos e beizos vese máis nítida e realista. Neste caso parece que o modelo de Deep Learning conseguiu suavizar lixeiramente os pequenos erros que se presentaban no obxecto orixinal.

Figura 7.7: Plano cercano do obxecto "head_ten24.ply". Comparativa de métodos de estimación.

Na figura 7.7 os 3 obxectos vense moi similares. Dende esta perspectiva os detalles notables son que ambos métodos de aproximación conseguiros suavizar os erros e a forma da orella lixeiramente, obtendo peores resultados no obxecto aproximado por Deep Learning. Os bordes inferiores quedaron ben definidos, pero hai unha superficie ruidosa na parte superior da orella. Tamén se suavizaron os erros do nariz e das cellas no obxecto da dereita, pero se crearon uns fallos en cor branca que non existían no orixinal na zona do lagrimal.

7.4 Comparativa de normais

Entendendo o comportamento das visualizacións dos obxectos, dependendo das normais engadidas, representamos esas normais para coñecer canto se acercan ás normais reais. Obsércanse as direccións das normais estimadas ao redor da figura e a cor nos indicará como de cerca ou lonxe están esas normais das reais. As cores serán coma un semáforo, se houbo pouco erro na normal, a súa representación verase verde, e canto maior sexa o erro tornará amarela e finalmente vermella.

Nótese que o obxecto representado máis a esquerda das seguintes figuras aparecerán en cor verde sempre, xa que son as normais reais do obxecto.

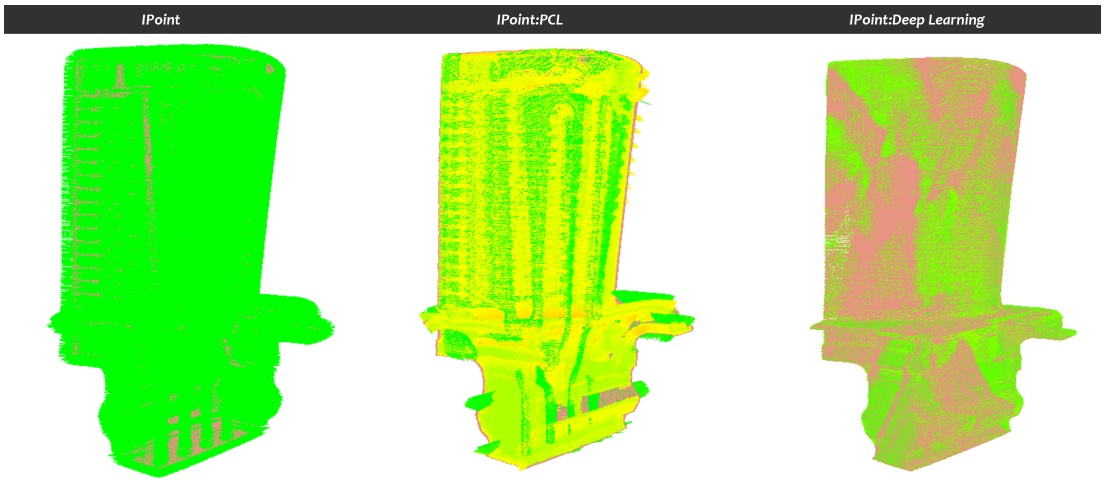


Figura 7.8: Plano xeral das normais do obxecto "blade.ply". Comparativa de métodos de estimación.

Na figura 7.8 resalta a cor amarela no obxecto estimado mediante PCL. Esta cor indica que as estimacións non son completamente erróneas, polo que poderían conseguirse resultados bos todavía. Se nos fixamos neste mesmo obxecto nalgúns dos bordes pódese apreciar que hay algunhas normais que apuntan en direccións atípicas, e que estes patróns repítense nas zonas cercanas ás esquinas, pero pasa menos nas superficies máis rectas. Con respecto ao obxecto estimado por Deep Learning podemos apreciar teñen unha cor verde, quizá non é o

verde máis intenso posible, pero en xeral asemella mellor que a estimación de PCL. Cabe dicir que se poden ver uns valores anómalos na zona na que se aprecian ocos, preto da metade do obxecto ao lado esquerdo.

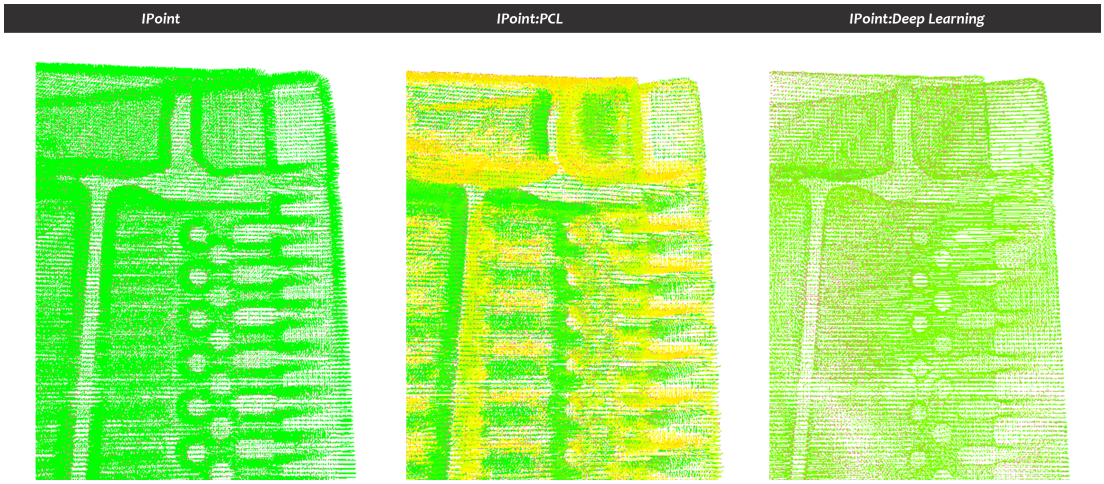


Figura 7.9: Plano cercano das normais do obxecto "blade.ply". Comparativa de métodos de estimación.

Na figura 7.9 decidíuse reducir o tamaño dos splats prácticamente a cero para poder ver ben a forma que teñen as normais. Apréciase que en todos os casos as estimacións recoñeceron os furados redondos que ten o obxecto, máis as formas que non son completamente planas como podería semellar na visualización do apartado anterior 7.2. De calquera xeito, aparenta que a estimación de PCL obtivo peores resultados identicando os pequenos detalles que se forman no obxecto, xa que non se identifican tan claramente. Aquí pódese ver máis claramente que na estimación de Deep Learning, o verde que obtemos non é un resultado exacto ou perfecto, pero o erro é menor que no caso de PCL de forma xeral. Vemos que no obxecto central, hay normais moi ben estimadas, e otras tantas cun erro suficientemente pronunciado para que se mostre en amarelo.

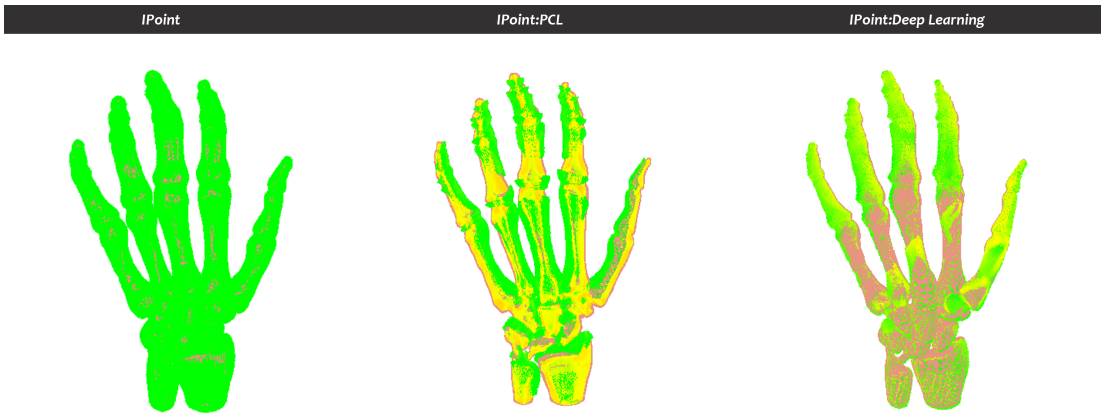


Figura 7.10: Plano xeral das normais do obxecto "hand.ply". Comparativa de métodos de estimación.

Na figura 7.10 vese claramente o contraste do que falabamos na figura 7.9. Ao ser un obxecto con máis formas e partes máis separadas, o contraste de amarelo e verde intenso é o que define a estimación mediante PCL. En cambio na estimación de Deep Learning, si que vemos que hay erros máis pronunciados que no obxecto *"blade.ply"*, nas zonas con esquinas máis pronunciadas. Pero o nivel de erro segue aparentando máis controlado e con menos contraste.

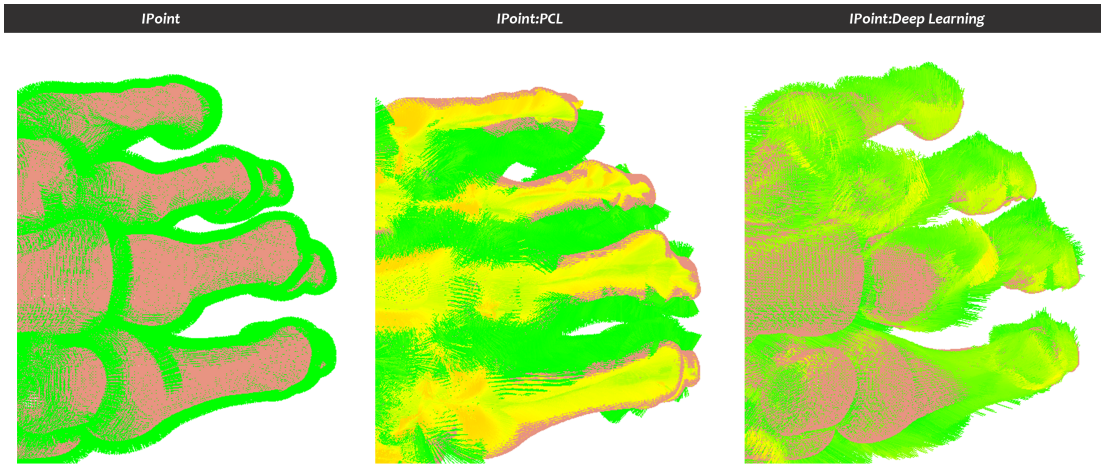


Figura 7.11: Plano cercano das normais do obxecto "hand.ply". Comparativa de métodos de estimación.

Ao ver a escena máis de cerca, na figura 7.11 seguimos atopando un contraste claramente marcado na estimación de PCL, con un alto porcentaxe de normais cun erro considerable. Ainda que como vimos na figura 7.4 a representación segue a ser moi boa. E no obxecto coas

normais estimadas por Deep Learning, vemos que a cor verde é predominante, e as normais parecen concordar coa superficie do obxecto.

7.5 Comparativa numérica de erro

Para ter uns datos máis concretos que a visualización, imos investigar os valores numéricos asociados ao erro de estimación de normais en cada caso. Para facer isto faremos uso dalgunhas ferramentas matemáticas como a media de erro que obtivemos en total, a desviación típica dese erro, e tamén veremos a cantidade de erros maior a un 10%, 25%, 50% e 75%.

	Arquivos de nubes de puntos (.ply)							
	suzanne30k	suzanne100k	hand	lucy	suzanne500k	blade	suzanne1M	head_ten24
Error medio	0,4588	0,4620	0,3101	0,3600	0,4636	0,2848	0,4626	0,4183
Desviación estandar	0,1293	0,1268	0,2267	0,1874	0,1251	0,2053	0,1261	0,1268
Nº errores > 10%	71	277	10.494	26.987	921	44.621	2.114	5.889
Nº errores > 25%	13.275	42.748	100.879	187.538	231.285	441.147	471.935	781.118
Nº errores > 50%	11.622	39.526	94.568	103.886	213.322	84.653	435.878	364.197
Nº errores > 75%	0	0	0	0	0	0	0	0

Figura 7.12: Táboa cos datos de erro do método de estimación de normais de PCL.

	Arquivos de nubes de puntos (.ply)							
	suzanne30k	suzanne100k	hand	lucy	suzanne500k	blade	suzanne1M	head_ten24
Error medio	0,1855	0,1819	0,2697	0,2031	0,1516	0,2290	0,1577	0,1265
Desviación estandar	0,0438	0,0438	0,1008	0,0733	0,0445	0,0304	0,0369	0,0439
Nº errores > 10%	24.241	80.740	132.048	266.769	427.985	675.879	926.524	804.071
Nº errores > 25%	2.318	6.804	181.383	97.949	19.266	207.052	28.935	26.848
Nº errores > 50%	0	0	2.879	0	0	22	0	0
Nº errores > 75%	0	0	0	0	0	0	0	0

Figura 7.13: Táboa cos datos de erro do método de estimación de normais creado mediante un modelo de Deep Learning.

Tras analizar varios obxectos podemos observar algunhas tendencias nos valores das táboas 7.12 e 7.13. Os valores de error medio son claramente maiores no caso de estimación con métodos tradicionais de PCL. Obsérvase que isto vai acompañado duns valores máis altos tamén na desviación estándar ou típica, como é de esperar ao ter unha incidencia de erro maior. Se analizamos o número de erros segundo se fallou por pouco (10%) ou por moito (75%), no caso da estimación de PCL apréciase que obtén mellores resultados que o modelo de estimación de Deep Learning para o número de erros superiores ao 10%. Pero despois o modelo de Deep Learning consegue resultados notoriamente mellores en erros superiores ao 25% e valores de 0 ou case cero en número de erros por encima do 50%. Nótese tamén que ambos teñen valores de 0 no erro superior ao 75%, o que é unha boa sinal, conforme ambos modelos están ofrecendo resultados prometedores.

Capítulo 8

Conclusións

Con respecto ao visualizador **IPoint** que creamos, é unha ferramenta moi útil para facer as comparativas que buscábamos dende o comezo do desenrolo deste proxecto. Foi o primeiro gran reto á hora de realizar este traballo, e foi evolucionando a medida que eu adquiría novos coñecementos. Ofrece as funcionalidades que se lle pediron nas especificacións iniciais e outras a maiores que foron surxindo a medida que crecían as necesidades. Este é un proxecto de investigación, e non de deseño de software, polo que ten marxen de mellora neste aspecto. Pero cumple coas espectativas de uso e incluso aporta novas utilidades.

Con respecto ao apartado de Deep Learning, que abarca o modelo adestrado e máis o pequeno software que permite estimar as normais dunha nube de puntos. O modelo adestrado é funcional e aporta uns resultados satisfactorios á hora de estimar as normais. O modelo ten moitas posibilidades de configuración, e son moitos os parámetros a ter en conta para conseguir inferir uns resultados cada vez mellores. Neste aspecto, este foi un dos meus primeiros modelos de aprendizaxe máquina, polo que ten unha gran capacidade de mellora, xa que a experiencia é un apartado moi importante á hora de conseguir melloras nos resultados. O pequeno software escrito en Python (**NormalGenerator**) que creamos para conseguir recoller esas normais estimadas polo modelo, ten un comportamento moi sinxelo e consigue cumplir coas expectativas marcadas inicialmente. É certo que podería mellorarse a integración de seu funcionamento co visualizador **IPoint**, facendo máis cómodo o uso continuado do software, pero o que buscábamos era utilidade, e foi o que se conseguimos.

Con respecto aos resultados que obtivemos facendo uso en conxunto da ferramenta **IPoint** e do estimador **NormalGenerator**. Así como visualmente, no apartado de resultados podía haber dúbidas con respecto a que modelo de estimación estaba ofrecendo mellor rendemento. Mirando as táboas 7.12 e 7.13 podemos dicir que o modelo de Deep Learning que temos adestrado obtén un erro máis reducido que nas estimacións clásicas. Ademáis vemos como a desviación estándar se mantén moi baixa e moi estable. Grazas a non ter apenas erros maiores ao 25% conséguese evitar que aparecen valores atípicos das normais en zonas aleatorias, como

veíamos que estaba pansando nas figuras 7.8 e 7.11 con PCL. Aínda que poden surxir zonas con ruido ou ocos como pasou na figura 7.2.

A ventaxa que teñen os modelos de Deep Learning sobre os métodos de estimación clásicos, é que estes teñen a capacidade de seguir aprendendo. Estes continuan mellorando o seu rendemento a medida que alimentas o modelo con máis datos, e aumentas o número de iteracións.

Os resultados parecen indicar que poderia seguir mellorándose a calidade da solución, de dispoñer de máis tempo e recursos para adicar ao modelo de Deep Learning.

Os vindeiros pasos a seguir para mellorar o proxecto aquí presentado serían por unha banda, facer a integración completa da ferramenta creada en Python no visualizador **IPoint** unificándose nun só producto software e por outra banda, axustar o modelo de Deep Learning coa fin de acadar unha cota de erro inferior ao actual.

Glosario

Sprint é o nome que recibe cada un dos ciclos ou operacións que se leva en un proxecto, utilízase na metodoloxía áxil Scrum.

Primitivas Para representar os nosos primeiros modelos 3D, utilízanse estructuras básicas que son as primitivas. A primitiva máis usada en desenvolvemento gráfico é o triángulo, aínda que temos unha gran variedade de elas como os puntos, as liñas e os quads (primitiva que forma un polígono de catro lados). Xuntando primitivas pódense formar obxectos máis complexos.

Quads Un quad é unha primitiva con 4 vertices en forma cadrada, os 4 vertices deben ser coplanares.

FoV Field of view. Campo de visionado. É a limitación do rango que se visualizar das figuras cando vamos a reprensentalas por pantalla.

Frustum é a delimitación do espazo visible por pantalla á hora de representar figuras. Só se verá o que está dentro do frustum

Viewport O viewport defínese para poder visualizar os gráficos que podemos representar. O que nos indica é o alto e ancho do espazo que queremos que sexa representable dentro dunha ventá da que teremos fixado una tamaño previamente.

Near Indica a limitación máis cercana da proxección que escollamos, que pode ser ortográfica ou perspectiva. Limitando o frustrum de visión que teremos por pantalla. A figura **??** pode axudar a entendelo visualmente.

Bottom e Top Son características da cámara, sirven para indicar os vectores que apuntan cara abaixo e cara arriba respectivamente.

Blending Despois de determinar os píxeles que son cubertos pola proxección dos splats no paso de rasterización, as técnica de blending utilízanse para variar a iluminación e o acabado estético da superficie das figuras.

Rasterización es el proceso en que cada primitiva se rompe en elementos discretos llamados Fragmentos, para determinar que espacio ocupan en pantalla.

Fragmento cada fragmento representa un segmento de una primitiva rasterizada. Los fragmentos tienen asociado un fragmento de la superficie de píxeles de la pantalla para representar la primitiva que tiene asociada.

Clipping as partes da figura que quedan fóra do frustum, e non se van a representar, son descartadas ou "clippeadas".

Underfitting [54] Cando os datos de entrenamento dun modelo de Machine ou Deep learning son poucos, o algoritmo non é capar de recoñecer o que se lle está pedindo. Por exemplo, non vai poder recoñecer 10 razas de can, ou non vai poder asegurar si o que sae nunha imaxe é un can ou un coche.

Overfitting [54] Cando o entrenamento do algoritmo se acostuma moito aos datos de entrada, e non é capaz de xeralizar. Por exemplo, si só pasamos fotos de cans brancos, e logo metemos un can negro, o algoritmo non o vai recoñecer. Esto solucionase conseguindo uns datos suficientemente distintos que permitan xeralizar.

Dropout O dropout "mata" de forma aleatoria un porcentaxe de neuronas en cada iteración do entrenamento. Esto axuda a mellorar o Overfitting.

Mínimo local Cando estamos intentando atopar o erro mínimo co noso optimizador, existen mínimos locais, pero o que nos interesa realmente é o mínimo global.

Mínimo global Cando estamos intentando atopar o erro mínimo, o mínimo global é o que nos interesa atopar para obter uns mellores resultados.

Momentum [55] É un termino añadido ás funcións de optimización, que se utiliza para variar o tamaño dos pasos que se dan ata atopar o mínimo global. Utilízase para saltar os mínimos locais e achar o global. Momentum tamén pode ser en si mesmo, un algoritmo de optimización.

Learning Rate [56] É un hiperparámetro que nos permite axustar o ratio de aprendizaxe da función de optimización que definimos no noso modelo. Unha boa configuración de este parámetro é clave para que o noso entrenamento chegue ao valor de erro máis baixo posible.

Capa de entrada ou Input layer É a capa de entrada de datos ao modelo. Estos datos alimentan a rede neuronal coa finalidade de facer una predicción final.

Capa oculta ou Hidden layer Son las capas que se atopan entre a capa de entrada e de saída. Pódense estructura de moitas formas variando o número de capas ocultas, e o número de neuronas que ten cada unha, dependendo do problema que estamos a resolver.

Capa de saída ou Output layer É a capa de saída onde se reflexan as predicciónns do modelo.

Set de entramento ou Training set Un conxunto de datos, que inclúe os datos de entrada do modelo, e os datos de saída esperados. É dicir, o resultado real.

Set de validación ou Validation set Pode ser un subconxunto dos datos do training set, que particionas para probar a evolución do modelo. Ou si tes datos de sobra, podes recoller datos a maiores co mesmo formato que o training set, para a mesme finalidade.

Taxa de erro ou Error rate Este erro define o lonxe que está o resultado do modelo do resultado real buscado por este. O obxetivo é minimizar o erro e acercarse o máximo posible ao resultado real.

Training loss É a información do erro que se está obtendo cos datos de entrenamento. Normalmente referenciada como 'Loss'.

Validation loss É a información do erro que se está obtendo dos valores que se están usando para probar o modelo que estamos a entrenar.

Neurona É a unidade básica das redes neuronais. Recolle unha entrada e xenera unha predicción según a función de activación que teña asignada.

Bias Os Bias utilizanse para equilibrar o modelo. Gracias a esta carácteristica influíndo nas neuronas, podemos facer que o Training Loss e Validation Loss chegen a un compromiso.

Pesos ou Weights Cada neurona ten un valor númerico como peso ou weight. Estos valores iran variando a medida que aprende o modelo, e serán os principales encargados de conseguir un erro no modelo o máis baixo posible.

Batch É cando se traballa por lotes, e se colle de entrada un pequeno trozo dos datos de entrada, e se alimenta o modelo con esos poucos datos. Este proceso se repite con cada Batch, e ao final se sacan conclusións totais do modelo para estimar o erro. Organizar en batches axuda a que o proceso de entrenamento sexa máis rápido, e tamén a evitar mínimos locais, e chegar aos mínimos absolutos.

Hiperparametros Os hiperparámetros son as características que podemos variar nunha rede neuronal para mellorar o seu funcionamento, xa seña variar o número de capas, o número de epochs, etc.

Bibliografía

[1] J. de Vries, "Learn opengl." [Online]. Available: https://learnopengl.com

[2] L. O. Álvarez Mures, "Real-time management tool for massive 3d point clouds," 2014. [Online]. Available: http://kmelot.biblioteca.udc.es/record=b1521392~S1*gag

[3] V. Autores, "Normal(geometry)." [Online]. Available: https://en.wikipedia.org/wiki/Normal_(geometry)

[4] DigitalSreeni, "Python for microscopists," 2019. [Online]. Available: https://www.youtube.com/channel/UC34rW-HtPJulxr5wp2Xa04w

[5] J. B. Ahire, "The artificial neural networks handbook: Part 1." [Online]. Available: https://medium.com/coinmonks/the-artificial-neural-networks-handbook-part-1-f9ceb0e376b4

[6] S. Jadon, "Introduction to different activation functions for deep learning." [Online]. Available: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

[7] i2tutorials, "What are local minima and global minima in gradient descent?" [Online]. Available: https://www.i2tutorials.com/what-are-local-minima-and-global-minima-in-gradient-descent/

[8] S. Doshi, "Various optimization algorithms for training neural network." [Online]. Available: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6

[9] R. Alberto, "¿qué es underfitting y overfitting?" [Online]. Available: https://rubialesalberto.medium.com/qu%C3%A9-es-underfitting-y-overfitting-c73d51ffd3f9

[10] M. Z. . H. P. . J. van Baar & Markus Gross, "Surface splatting." [En liña]. Dispoñible en: https://vcg.seas.harvard.edu/publications/surface-splatting/paper

[11] L. Reid, "Simple mesh tessellation & triangulation tutorial." [En liña]. Dispoñible en: https://lindenreid.wordpress.com/2017/12/03/simple-mesh-tessellation-triangulation-tutorial/

[12] EvaluandoSoftware, "¿qué es desarrollo de software ágil?" 2018. [En liña]. Dispoñible en: https://www.evaluandosoftware.com/desarrollo-software-agil/

[13] V. autores, "Agile manifesto," 2001. [En liña]. Dispoñible en: https://agilemanifesto.org/

[14] V. Autores, "Scrum." [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)

[15] Jobted, "Sueldo del ingeniero informático en españa." [En liña]. Dispoñible en: https://www.jobted.es/salario/ingeniero-inform%C3%A1tico

[16] Khronos, "Khronos group." [En liña]. Dispoñible en: https://www.khronos.org

[17] Varios, "Opengl shading language," 2020. [En liña]. Dispoñible en: https://en.wikipedia.org/wiki/OpenGL_Shading_Language

[18] Khronos, "Fragment shader." [En liña]. Dispoñible en: https://www.khronos.org/opengl/wiki/Fragment_Shader

[19] ——, "Depth test." [En liña]. Dispoñible en: https://www.khronos.org/opengl/wiki/Depth_Test

[20] ——, "glalphafunc." [En liña]. Dispoñible en: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glAlphaFunc.xml

[21] S. H. Ahn, "Opengl camera," 2016. [En liña]. Dispoñible en: http://www.songho.ca/opengl/gl_camera.html

[22] ScanPhase, "Generation and use of point clouds." [En liña]. Dispoñible en: https://www.scanphase.com/point-clouds

[23] captae, "Productos escáner laser 3d nube de puntos." [En liña]. Dispoñible en: https://captae.com/captae/productos-escaner-laser-3d-nube-de-puntos/

[24] globalmediterranea, "Fotogrametría: Qué es, ventajas y metodología." [En liña]. Dispoñible en: https://www.globalmediterranea.es/fotogrametria-que-es/

[25] Euclideon, "Euclideon." [En liña]. Dispoñible en: https://www.euclideon.com/

[26] MeshLab, "Meshlab." [En liña]. Dispoñible en: https://www.meshlab.net/

[27] Danielgm, "Cloudcompare." [En liña]. Dispoñible en: https://www.danielgm.net/cc/

[28] PCL, "Pcl." [En liña]. Dispoñible en: https://pointclouds.org/

[29] M. G. . H. Pfister, *Point-Based Graphics*, 1st ed. Morgan Kaufmann, 2007.

[30] D. A. González, "Visualización avanzada de nubes de puntos con opengl," 2015. [En liña]. Dispoñible en: http://kmelot.biblioteca.udc.es/record=b1535210~S1*gag

[31] V. Autores, "Estimating surface normals in a pointcloud." [En liña]. Dispoñible en: https://pcl.readthedocs.io/projects/tutorials/en/latest/normal_estimation.html?highlight=normal%20estimation#estimating-the-normals

[32] ——, "Vector propio y valor propio." [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/Vector_propio_y_valor_propio

[33] ——, "Covarianza." [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/Covarianza

[34] GLFW, "Glfw." [En liña]. Dispoñible en: https://www.glfw.org/

[35] GLEW, "The opengl extension wrangler library." [En liña]. Dispoñible en: http://glew.sourceforge.net/

[36] christophe lunarg, "Glm." [En liña]. Dispoñible en: https://github.com/g-truc/glm

[37] CMake, "Cmake." [En liña]. Dispoñible en: https://cmake.org/

[38] D. Calvo, "Función de activación – redes neuronales." [En liña]. Dispoñible en: https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/

[39] S. SHARMA, "What the hell is perceptron?" [En liña]. Dispoñible en: https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

[40] R. Parmar, "Common loss functions in machine learning." [En liña]. Dispoñible en: https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23

[41] S. Ruder, "An overview of gradient descent optimization algorithms." [En liña]. Dispoñible en: https://ruder.io/optimizing-gradient-descent/

[42] E. Blanco, "¿cómo funciona el algoritmo backpropagation en una red neuronal?" [En liña]. Dispoñible en: https://empresas.blogthinkbig.com/como-funciona-el-algoritmo-backpropagation-en-una-red-neuronal/

[43] V. Autores, "Función softmax." [En liña]. Dispoñible en: https://es.wikipedia.org/wiki/Funci%C3%B3n_SoftMax

[44] V. Bushaev, "Adam — latest trends in deep learning optimization." [En liña]. Dispoñible en: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

[45] K. MAHENDRU, "A detailed guide to 7 loss functions for machine learning algorithms with python code." [En liña]. Dispoñible en: https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/3

[46] scikit learn, "Nearest neighbors." [En liña]. Dispoñible en: https://scikit-learn.org/stable/modules/neighbors.html

[47] TensorFlow, "An end-to-end open source machine learning platform." [En liña]. Dispoñible en: https://www.tensorflow.org/

[48] KerasAPI, "Model training apis." [En liña]. Dispoñible en: https://keras.io/api/models/model_training_apis/

[49] scikit learn, "scikit-learn." [En liña]. Dispoñible en: https://scikit-learn.org/stable/

[50] NumPy, "Numpy." [En liña]. Dispoñible en: https://numpy.org/

[51] Pandas, "Pandas." [En liña]. Dispoñible en: https://pandas.pydata.org/

[52] matplotlib, "matplotlib." [En liña]. Dispoñible en: https://matplotlib.org/

[53] PyntCloud, "Pyntcloud." [En liña]. Dispoñible en: https://pyntcloud.readthedocs.io/en/latest/

[54] Na8, "Qué es overfitting y underfitting y como solucionarlo," 2017. [En liña]. Dispoñible en: https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo

[55] M. Stewart, "Neural network optimization," 2019. [En liña]. Dispoñible en: https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0

[56] ——, "Simple guide to hyperparameter tuning in neural networks," 2019. [En liña]. Dispoñible en: https://towardsdatascience.com/simple-guide-to-hyperparameter-tuning-in-neural-networks-3fe03dad8594