

Instituto de Física da USP

Desenvolvimento de Ferramentas de Machine Learning para Estudos de Reações Nucleares em Alvos Ativos

Relatório de atividades do projeto de pesquisa de mestrado referente ao período de fevereiro à dezembro de 2020

Guilherme Ferrari Fortino (Bolsista)

Valdir Guimarães

Professor Dr. no Instituto de Física da USP (Orientador)

Juan Carlos Zamora Cardona

Pós-doutorando no Instituto de Física da USP (Coorientador)

São Paulo

2020

Sumário

Lista de figuras

TabelasLista de algoritmos

Capítulo 1

Introdução

Um dos principais objetivos do estudo em física nuclear é entender a estrutura do núcleo. Apesar do sucesso do modelo de camadas em explicar as estruturas de núcleos estáveis, os núcleos instáveis ou exóticos, ricos ou pobres em nêutrons, continuam sendo um grande desafio para nossa compreensão.

Os núcleos leves radioativos são de grande interesse para a astrofísica nuclear.

Experimentos para o estudo desses núcleos

Capítulo 2

Uma breve introdução ao *Machine Learning*

Machine learning é a área de estudo que desenvolve algoritmos para que eles possam aprender com os dados, sem serem explicitamente programados para isso[mlbook]. Supõe-se que uma rede neural imite um sistema biológico, em que os neurônios interajam enviando sinais na forma de funções matemáticas entre as camadas. Isso inspirou um modelo matemático simples para um neurônio artificial:

$$y = f \left(\sum_{i=1}^n \omega_i x_i + b_i \right) = f(z), \quad (2.1)$$

onde y é a saída do neurônio, que corresponde à função de ativação f que depende da soma ponderada, onde o peso é ω_i , das entradas x_i dos outros n neurônios. O termo b_i corresponde ao parâmetro *bias*. A ideia é fazer um neurônio receber a informação de todos os outros neurônios da camada anterior, fazendo uma média ponderada (onde o peso que será estimado pelo algoritmo de *machine learning*) e somando com um termo independente (*bias*, que também é estimado). Os parâmetros ω_i e b_i serão estimados através de um determinado procedimento, chamado de treino.

2.1 Tipos de redes neurais

Uma rede neural artificial, *Artificial Neural Network* (ANN), é um modelo computacional que consiste de camadas de neurônios. Muitas ANN's foram desenvolvidas, mas a maioria consiste em uma camada de entrada (*input layer*), uma camada de saída (*output layer*) e eventuais camadas entre essas duas, chamadas de camadas ocultas (*hidden layers*). Os tipos mais comuns são:

Feed-Forward Neural Networks

A *Feed-forward neural networks* (FFNN) é a primeira e mais simples rede neural desenvolvida. Nessa rede a informação se move apenas para frente através das camadas (da camada de entrada até a camada de saída). A figura ?? mostra a representação da rede, onde os neurônios são representados por círculos, enquanto que as linhas mostram as conexões entre os neurônios. Cada neurônio recebe informação de todos os neurônios da camada anterior, portanto a rede é chada de totalmente conectada, *fully-connected* (FC), FFNN.

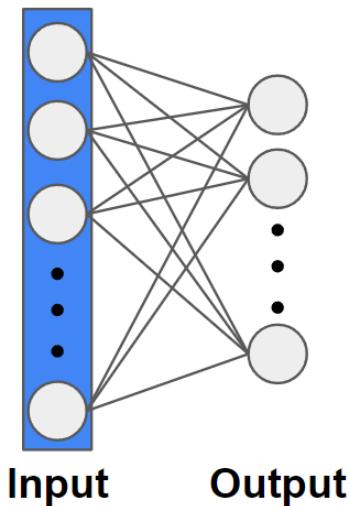


Figura 2.1: Exemplo de FFNN. A camada de entrada na esquerda propaga a informação para a direita (camada de saída). Todos os neurônios entre camadas estão conectados entre si.

Convolutional Neural Network

Uma variante da FFNN é a chamada de rede neural convolucional, *convolutional neural network* (CNN). Do ponto de vista matemático sobre convoluções, a convolução descrita como $(f * g)(t)$ de uma função $f(t)$ e outra $g(t)$ é definida como:

$$(f * g)(t) \equiv \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau. \quad (2.2)$$

Para o caso discreto, com g sendo uma função resposta finita de tamanho $2M$, temos

$$(f * g)[n] = \sum_{m=-M}^{M} f[n - m]g[m]. \quad (2.3)$$

Convoluções são invariantes sobre rotação e translação, portanto são muito utilizadas para processamento de sinais e imagens[[signal book](#)]. Para a convolução discreta se escolhe um filtro que irá atuar no vetor desejado. Para ilustrar o que significa isso, no caso discreto e unidimensional, a figura ?? mostra o processo de convolução de um vetor de tamanho 9 e um filtro de tamanho 3.

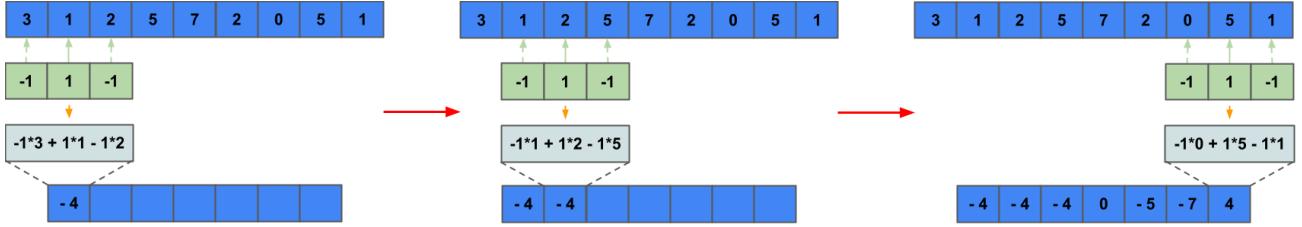


Figura 2.2: Processo de convolução entre sinal azul em cima e o filtro em verde, resultando no sinal azul embaixo.

Percebe-se que o sinal resultante tem dimensão menor que o sinal original. O filtro (também chamado de *kernel*) atua em um ponto que possua vizinhos o suficiente para o restante do filtro poder fazer a multiplicação ponto a ponto. Esse tipo de convolução tem o chamado emparelhamento válido (*valid padding*). O tamanho n_2 resultante do vetor de saída é

$$n_2 = n_1 - m + 1, \quad (2.4)$$

onde n_1 é o tamanho do vetor de entrada e m o tamanho do filtro (*kernel size*). Para que o vetor de saída tenha o mesmo tamanho do vetor de entrada, são acrescentados zeros em torno da entrada, de forma que a saída tenha o mesmo tamanho da entrada. Esse é o chamado emparelhamento igual (*same padding*). A figura ?? ilustra esse processo.

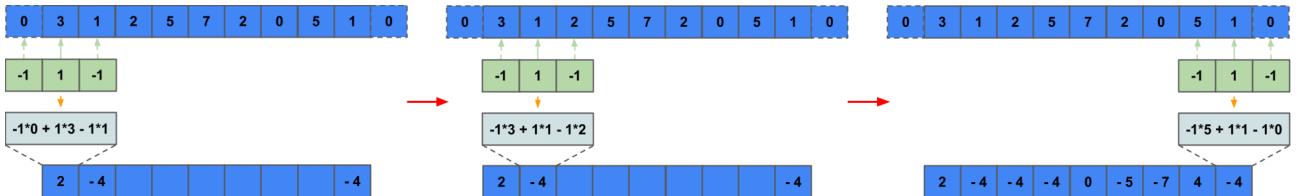


Figura 2.3: Processo de convolução entre o sinal azul em cima e o filtro em verde, resultando no sinal azul embaixo. Agora são acrescentados zeros no inicio e no final do vetor para que o vetor saída tenha o mesmo tamanho do vetor de entrada (nesse caso 9).

Uma CNN é capaz de fazer convoluções. Como estamos no contexto de inteligência artificial, *a priori* não sabemos quais os valores dos filtros que devem ser aplicados, apenas seus tamanhos e como agem. A ideia é estimar os valores do filtro, através do treino da rede neural, que deve ser aplicado para se obter o resultado desejado.

Cada filtro aplicado gera um mapa característico (*feature map*), que é o resultado da atuação do filtro em um vetor. Usualmente em uma CNN se escolhe o tamanho do filtro, *padding* (*valid* ou *same*) e quantos filtros serão aplicados (para saber quantos *feature maps* serão gerados). Os filtros têm seus valores estimados pelo treino da rede neural, gerando vários mapas (*feature maps*). Como temos vários mapas, isso acarreta em um aumento de dimensionalidade. Para filtrar/selecionar os mapas é usado um critério, como por exemplo selecionar valores máximos dos mapas gerados dada uma janela de atuação (quantos mapas serão comparados para selecionar o máximo valor). O *Max-Pooling* faz isso, selecionando valores máximos para uma determinada quantidade de mapas sendo comparados (*pool size*).

Existem outros tipos de redes neurais, porém não serão discutidas aqui.

2.2 Construindo uma rede neural

Para a construção de uma rede neural (nesse caso em específico de uma rede neural supervisionada, que será discutida mais para frente), precisamos primeiro entender sobre os dados que estamos trabalhando. A maioria das redes neurais possuem um *input* que deve ter dimensão fixa, ou seja, todos os dados devem ter o mesmo formato. O mesmo vale para o *output*.

Para cada camada da arquitetura devemos escolher sua função de ativação. Tanto FNN's quanto CNN's podem possuir funções de ativação. Dentre muitas funções de ativação podemos citar a *Rectified Linear Units* (ReLU)[**ReLU**], sigmoide, linear, tangente hiperbólica etc. A figura ?? mostra os gráficos dessas funções de ativação.

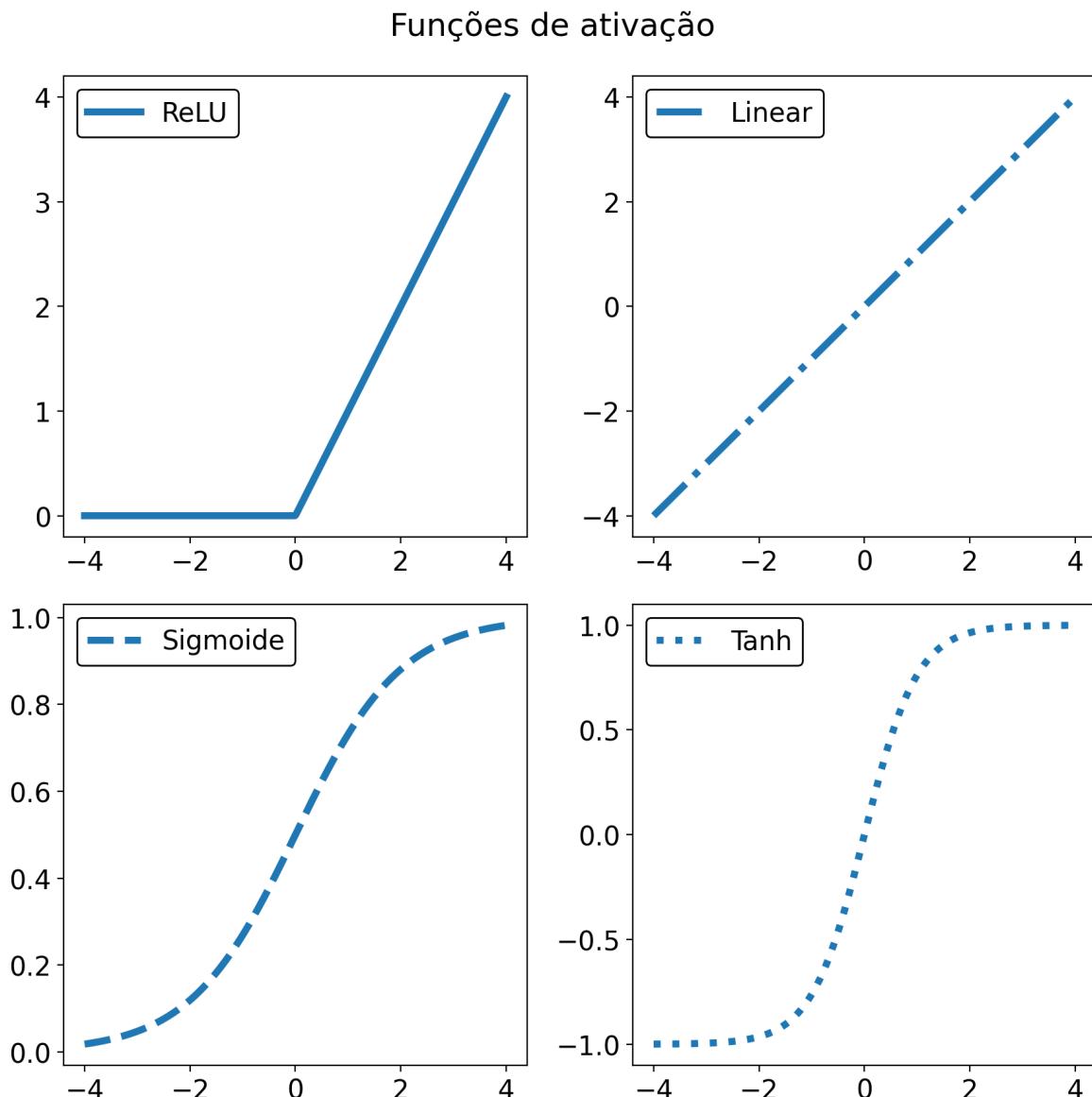


Figura 2.4: Funções de ativação e seus respectivos gráficos.

O próximo passo é definir a função custo (também chamada de *loss*) e o otimizador. A função custo tem o papel de retornar valores altos para previsões erradas e valores baixos para previsões corretas. Por exemplo, se queremos treinar uma rede neural para classificação binária (que prevê, devemos usar a função custo chamada de *binary cross-entropy* dada por

$$C(p(y_i)) = -\frac{1}{N} \sum_{i=1}^N y_i \log(p(y_i)) + (1 - y_i) \log(1 - p(y_i)), \quad (2.5)$$

onde y_i é o rótulo (*label*), $p(y_i)$ é a probabilidade do ponto y_i ser 1 e N é o número de pontos. O objetivo da rede neural é achar o mínimo da função $C(p(y_i))$, o que implica diretamente na melhor solução para o conjunto de dados. Isso é feito pelo método de retropropagação do erro (*backpropagation*) por um otimizador. Outros exemplos de *loss* são: erro quadrático médio, *categorical cross-entropy* etc.

O otimizador tem o objetivo de otimizar os parâmetros presentes na rede neural, buscando o mínimo global da função custo, o que nem sempre acontece, pois a minimização pode parar em um mínimo local da função. Existem diversos otimizadores, como por exemplo o *Stochastic Gradient Descent* (SGD), ADAM, ADAMAX etc. Para o SGD temos que a atualização de parâmetros é dada por

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} C(\theta), \quad (2.6)$$

onde θ_j é o parâmetro a ser atualizado, α é a *learning rate* e $C(\theta)$ é a *loss* que depende dos parâmetros.

Para enfim treinar a rede neural, se escolhe o *batch size*, que é o tamanho de amostras que irá ser usada para o treino, por iteração em cada rodada de treino (*epoch*). Por exemplo, se usamos 1000 dados para o treino, e o *batch size* é 500, cada *epoch* terá duas iterações. No geral se usam *batch sizes* pequenos, pois o consumo de memória é mais eficiente.

Para avaliação do modelo se usam dados de validação, que servem para verificar o comportamento da rede neural que está sendo treinada em dados que não são usados para treino, a fim de verificar problemas, como por exemplo o *overfit*, que ocorre quando a rede neural começa a se adequar perfeitamente aos dados de treino, perdendo a capacidade de previsão em dados que não estão sendo vistos pela rede neural.

Além dos dados de validação podemos escolher métricas que auxiliam a visualização do treino e nos retornam informações importantes sobre sua qualidade. Exemplos importantes de métricas são: acurácia binária, erro médio absoluto, acurácia categórica, falsos positivos etc. Tudo depende do objetivo da rede neural.

2.3 Sistemas de *machine learning*

Podemos dividir os tipos de sistemas de *machine learning* em quatro tipos:

Aprendizado Supervisionado

Aprendizado supervisionado é quando fornecemos para a rede neural um conjunto de dados para o treino com a solução desejada (chamados de *labels*). Um uso típico é para problemas de classificação. Por exemplo, classificação de imagens (identificação de figuras), previsão de valores numéricos etc. Exemplos de algoritmos supervisionados são:

- *k-Nearest Neighbors*[knn]
- Regressão linear
- *Support Vector Machines* (SVMs)
- *Decision Trees and Random Forests*
- Redes neurais

Aprendizado não supervisionado

Aprendizado não supervisionado é quando fornecemos o conjunto de dados para o treino, porém sem solução. A ideia é aprender sem supervisão. Um problema comum, por exemplo, é quando queremos identificar *clusters* em um conjunto de dados (*clustering*).

Aprendizado semi supervisionado

Aprendizado supervisionado é quando apenas parte do conjunto de dados para o treino possui *labels*. Isso é comum quando se obtém conjuntos de dados diferentes e apenas parte deles foi classificado.

Aprendizado por reforço

Aprendizado por reforço é quando um sistema, chamado de *agente* nesse contexto, aprende através do ambiente, realizando ações que maximizam sua recompensa. Por exemplo, caso o sistema realize uma ação incorreta, ele recebe uma penalidade, fazendo com que procure outra maneira de realizar a ação, dessa vez de maneira correta, para poder ganhar uma recompensa. Esse tipo de sistema é muito usado, por exemplo, em automatização robótica, como carros que pilotam sozinhos, robôs que aprendem a andar etc.

2.4 Aplicações de *machine learning* na física nuclear

Em física nuclear, o uso de técnicas de *machine learning* tem se mostrado cada vez mais importante. Podemos citar alguns exemplos de uso em:

Estudo de propriedades de núcleos

É possível estimar propriedades de núcleos usando modelos e dados experimentais já existentes. Dada a capacidade de redes neurais de aprenderem padrões não lineares nos dados, isso tem se mostrado uma alternativa para estimar propriedades dos núcleos como: raio de carga nuclear [**raio·carga**], massa de núcleos [**nuclear·mass**] etc.

Decaimento beta e processo-*r*

O decaimento β é fundamental para entender a origem dos elementos pesados. Prever o tempo de meia vida do decaimento β é de grande importância para simulações do processo-*r* (caputra rápida de nêutrons). Com redes neurais é possível fazer previsões que levam em conta a vida do problema, como visto na Ref. [**mlbetadecay**]. A figura [xx] mostra a previsão do tempo de meia vida ($T_{1/2}$) do decaimento β (em segundos) para isótonos com número de nêutrons $N = 126$.

Alvos ativos

Experimentos com alvos ativos (que será discutido nos próximos capítulos) geram enormes quantidades de dados, o que gera a necessidade do uso de algoritmos de *machine learning* para diminuir o consumo em tempo. Além disso são geradas nuvens de pontos tridimensionais, o que demanda um grande trabalho do ponto de vista de visão computacional, com algoritmos capazes de identificar estruturas em três dimensões. A figura [xx] mostra exe

Com essa breve descrição sobre *machine learning* podemos seguir adiante e entender seu uso dentro deste trabalho, que será feito nos próximos capítulos.

Capítulo 3

O experimento

O prototype Active Target - Time Projection Chamber (pAT-TPC) é um detector de alvo ativo que usa o volume de gás tanto quanto alvo quanto como detector. Seu grande volume ativo e capacidade de *tracking* fornecem boa resolução energética e angular, o que torna o pAT-TPC adequado para trabalhar com feixes exóticos de baixa intensidade [pattpc, pattpc2]. O feixe secundário de ^{17}F usado pelo pAT-TPC é produzido pelo *Twin Solenoids* (TWINSOL) [twinsol]. A figura ?? mostra os dois sistemas acoplados.

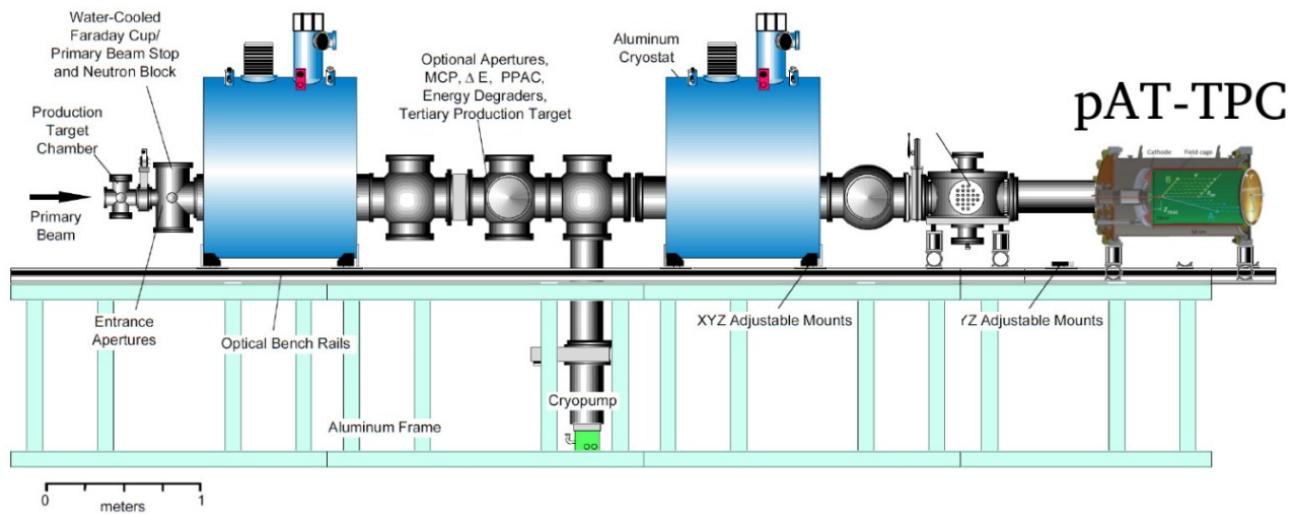


Figura 3.1: Sistema TWINSOL à esquerda e pAT-TPC à direita. Todo o sistema está localizado na University of Notre Dame.

Essa seção irá descrever brevemente o pAT-TPC, bem como o TWINSOL. Descrições mais detalhadas podem ser encontradas nas referências [pattpc, twinsol].

3.1 O sistema TWINSOL

O TWINSOL é um sistema de produção de feixes radioativos de baixa intensidade que possui dois solenoides supercondutores alinhados que são usados para produzir, coletar, transportar, focar e analisar feixes estáveis e radioativos.

Cada solenoide possui 30 cm de raio interno e 1 m de comprimento. O fato de ser um solenoide finito faz com que surjam efeitos de borda na componente radial do campo magnético do solenoide, cujo efeito faz com que o solenoide seja capaz de focalizar partículas. A figura ?? mostra um exemplo de cálculo do valor do campo magnético, usando valores do sistema “irmão” do TWINSOL, o Radioactive Ion Beams in Brasil (RIBRAS) [ribbras], para as componentes x , y e z . É nítido que o campo em z permanece praticamente constante e próximo das extremidades da bobina (linha vertical tracejada preta), B_x e B_y crescem em módulo, fazendo com que o solenoide funcione como uma lente delgada.

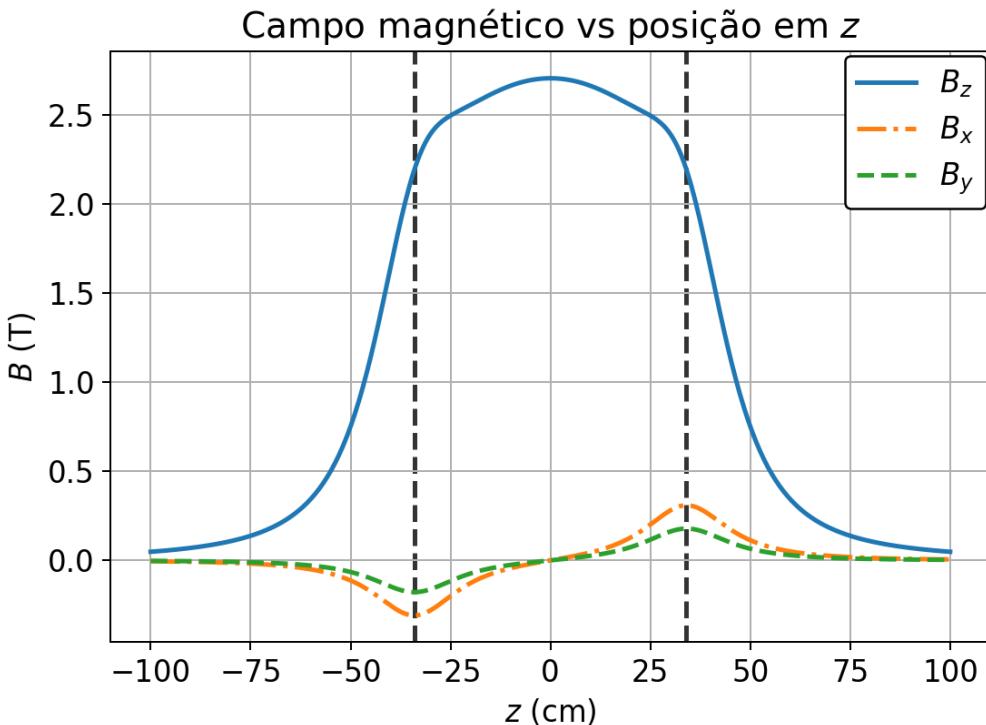


Figura 3.2: Valor do campo magnético B em T em função da posição em z em cm da bobina. A linha vertical tracejada preta indica o limite físico da bobina. O campo foi calculado à uma distância de 8 cm do eixo do solenoide. É possível ver claramente o efeito de borda que há em um solenoide finito.

A trajetória das partículas dentro do solenoide é helicoidal e, como os solenoides funcionam como uma lente, é possível calibrar o ponto focal. O foco depende da rigidez magnética da partícula através da relação:

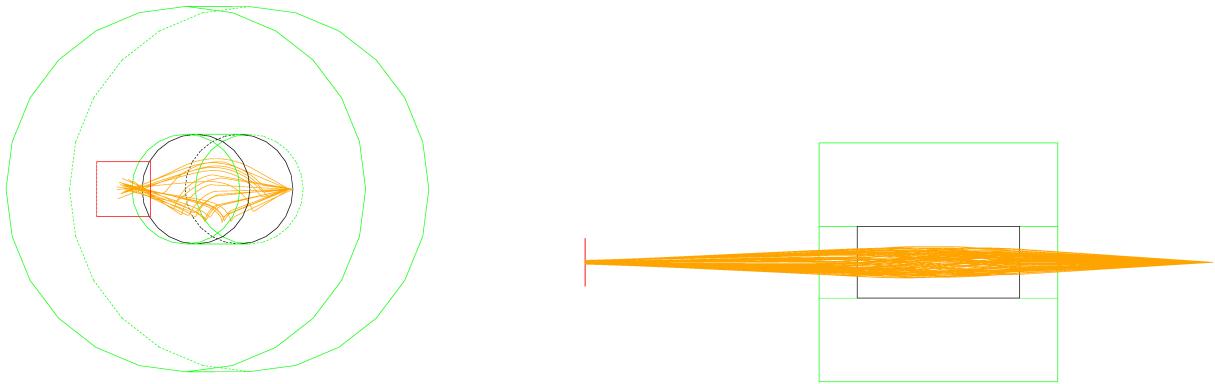
$$\frac{1}{f} = \frac{B_z^2}{(B\rho)^2}, \quad (3.1)$$

onde f é o ponto focal, B_z a componente z do campo magnético, e $B\rho$ é dado por:

$$B\rho = \frac{mv}{q} = \frac{\sqrt{2mE}}{q}, \quad (3.2)$$

onde E é a energia, m sua massa e q seu estado de carga. A figura ?? mostra a simulação,

usando o GEANT4[geant4], usando novamente valores do RIBRAS, da trajetória das partículas dentro do solenoide e em ?? pode-se ver que o feixe é focalizado em um ponto.



(a) Simulação que mostra as trajetórias helicoidais das partículas em laranja dentro do solenoide de cor verde até colidirem em um plano vermelho.

(b) Partículas em laranja passam pelo solenoide de cor verde que funciona como lente, sendo focalizadas em ponto da linha vermelha à esquerda da figura.

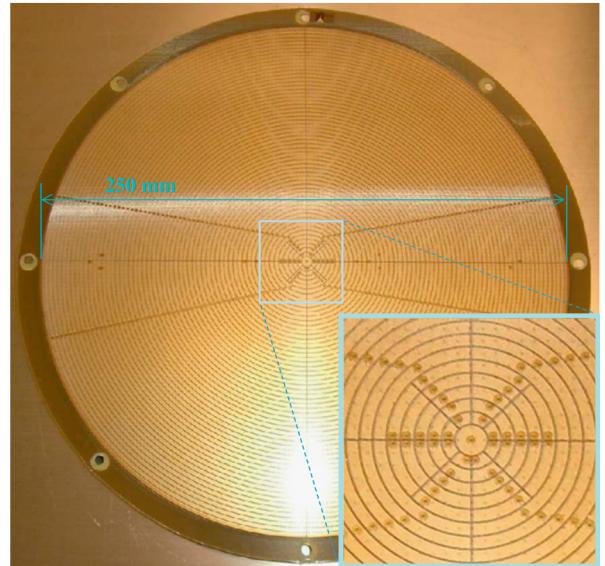
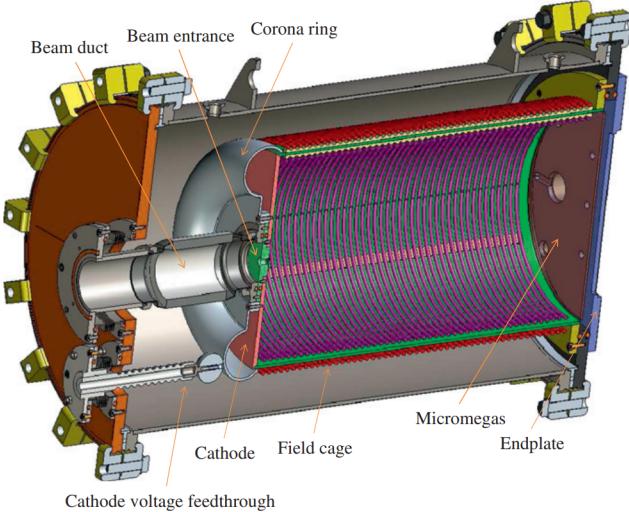
Figura 3.3: Simulações computacionais usando o GEANT4 para entender o comportamento de partículas carregadas que passam por um solenoide supercondutor.

Sabendo então das propriedades do sistema, podemos produzir então feixes radioativos. Para produzir ^{17}F uma célula gasosa preenchida com deutério é bombardeada por um feixe de ^{16}O . Não há só a reação que produz ^{17}F , temos as reações: $^{16}\text{O}(\text{d}, \text{n})^{17}\text{F}$, $^{16}\text{O}(\text{d}, \text{p})^{17}\text{O}$ e também o feixe de ^{16}O pode ser apenas espalhado. Outras reações, por exemplo, com a estrutura da célula gasosa (janela feita de 5 μm de titânio) podem ocorrer, mas o papel do TWINSOL é de selecionar e focalizar apenas as partículas desejadas.

Um problema comum é de que, mesmo para partículas diferentes, o B_ρ possa ser muito próximo ou igual. Isso faz com que não seja possível obter um feixe de ^{17}F com 100% de pureza. O feixe produzido possui 54 % de ^{17}F , 41 % de ^{17}O e cerca de 5 % de ^{16}O . Por fim, o feixe produzido pelo TWINSOL é conduzido até o pAT-TPC.

3.2 O alvo ativo pAT-TPC

A figura ?? mostra o desenho esquemático do pAT-TPC. O detector possui uma cela cilíndrica de 50 cm de comprimento e 28 cm de diâmetro, onde o seu eixo é alinhado com o eixo do feixe. A câmara é preenchida com o ^4He puro gasoso à uma pressão de 350 Torr que serve tanto quanto alvo de reação quanto meio detector. Tanto o feixe quanto partículas originadas da reação ionizam o gás e os elétrons que surgem dessa ionização são conduzidos por um campo elétrico de 1 kV/cm perpendicular ao eixo da câmara até o detector (*pad plane*), o *Micromegas*[micromegas], mostrado na figura ??.



(a) Visão transversal do pAT-TPC. O gás é preenchido dentro da cela que possui um campo elétrico perpendicular ao plano do *Micromegas*.

(b) Foto do *Micromegas*. O detector é multipixelado com uma maior densidade no centro, parte destacada na imagem, pois seu ganho é menor para não queimar os canais centrais. O *pad* central tem diâmetro de 5 mm enquanto que as faixas coaxiais possuem passo de 2mm. Os *pads* são separados por um intervalo de 0.25 mm.

Figura 3.4: Figura esquemática do pAT-TPC e o detector *Micromegas*[pattpc].

O *Micromegas* é um dispositivo de amplificação de elétrons, que consiste em um rede com 2048 canais triangulares feitos em ouro em que cada canal possui uma eletrônica independente. O formato triangular dos canais tem como objetivo maximizar a resolução espacial do detector. Cada canal possui uma posição (x, y) fixa e a terceira coordenada z será determinada a partir do tempo de voo da partícula. Isso só é possível pois a velocidade de drift dos elétrons é constante[**drift constant**], portanto a posição em z da partícula é diretamente proporcional ao tempo de voo. Esse princípio que deu origem ao nome de *Time Projection Chamber*, pois o evento é projetado no tempo. A equação ?? (equação de Langevin) descreve o movimento de um elétron com massa m e carga e é descrito por

$$m \frac{d\vec{v}}{dt} = e (\vec{E} + \vec{v} \times \vec{B}) - \frac{m}{\tau} \vec{v}, \quad (3.3)$$

onde \vec{E} é o vetor campo elétrico, \vec{B} o vetor campo magnético, \vec{v} é o vetor de velocidade do elétron e τ é o tempo de colisão médio, que depende das propriedades termodinâmicas do gás. No caso do pAT-TPC, \vec{B} é zero e a solução estacionária para a velocidade de drift do elétron é

$$\vec{v} = \frac{\tau}{m} e \vec{E}. \quad (3.4)$$

A velocidade de drift depende das propriedades termodinâmicas do gás (temperatura, pressão) e também de sua condutividade elétrica. Isso significa que é a calibração da velocidade envolve a calibração não só do campo elétrico, mas também do gás dentro do TPC.

A conversão para a coordenada z é linear, portanto no tempo $t = 0$ o elétron está no plano do detector.

O *Micromegas* amplifica o sinal usando *thick gems*. *Thick gems* usam do fato de que, no momento em que o elétron passa para uma região de campo elétrico ordens de grandeza maior que de sua origem, ocorre a ionização secundária (quando o elétron ioniza o gás). Isso provoca o que é chamado de avalanche de elétrons, amplificando a intensidade do sinal recebido. A figura ?? mostra a esquematização do *Micromegas*.

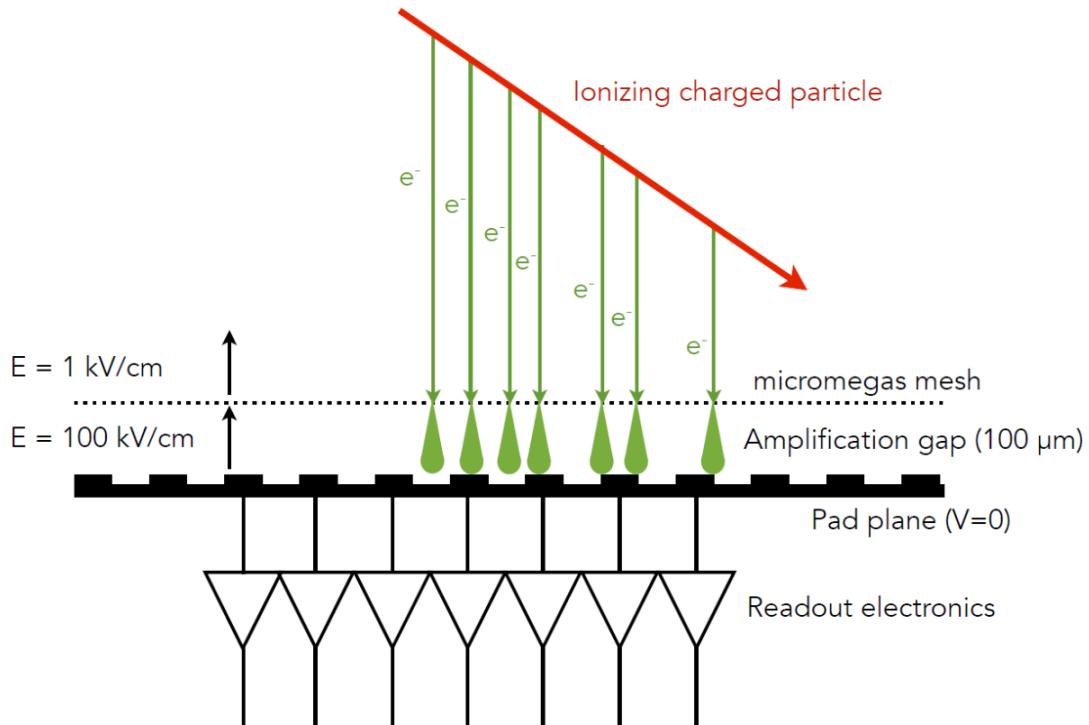


Figura 3.5: Plano do *Micromegas* com a esquematização das *thick gems*. Os elétrons quando passam para um campo elétrico mais intenso ionizam o gás, produzindo ainda mais elétrons (evento chamado de avalanche de elétrons).

Nem todos os *pads* do detector são ativados por evento. Existem canais auxiliares que servem para evitar armazenar canais sem detecção. Caso haja detecção além do centro do *Micromegas* então os sinais gerados pelo evento são armazenados. São gerados cerca de 300 sinais por evento, sendo que existem milhões de eventos, o que gera a necessidade de desenvolvimento de algoritmos extremamente eficientes em tempo para a análise. Para análise completa do experimento precisamos seguir as seguintes etapas:

- Reconstruir os eventos tridimensionais (nuvens de pontos) a partir dos sinais gerados pelo *micromegas*. Isso inclui remover o fundo, localizar todos os pontos de interação das partículas carregadas com o gás etc. Isso será mostrado em detalhes no capítulo ??;
- A partir das nuvens de pontos é necessário reconstruir a cinemática das reações, identificando trajetórias das partículas e o vértice de reação;

- Com a cinemática reconstruída podemos associar as partículas com as trajetórias e finalmente construir as seções de choque.

As análises dos pulsos, reconstituição de eventos e resultados serão mostradas nos próximos capítulos.

Capítulo 4

Reconstrução de nuvens de pontos

Esse capítulo irá mostrar como são recriadas as nuvens de pontos (*pointclouds*) a partir dos pulsos gerados por cada pixel do *micromegas*. Primeiro será mostrado como os sinais são analisados a partir de métodos mais tradicionais e depois usando algoritmos de *machine learning*.

4.1 Análise dos pulsos com algoritmos tradicionais

Para a reconstituição dos eventos devemos analisar os sinais dos canais do *micromegas*. Cada canal possui eletrônica independente e os sinais recebidos são como os mostrados na figura ??.

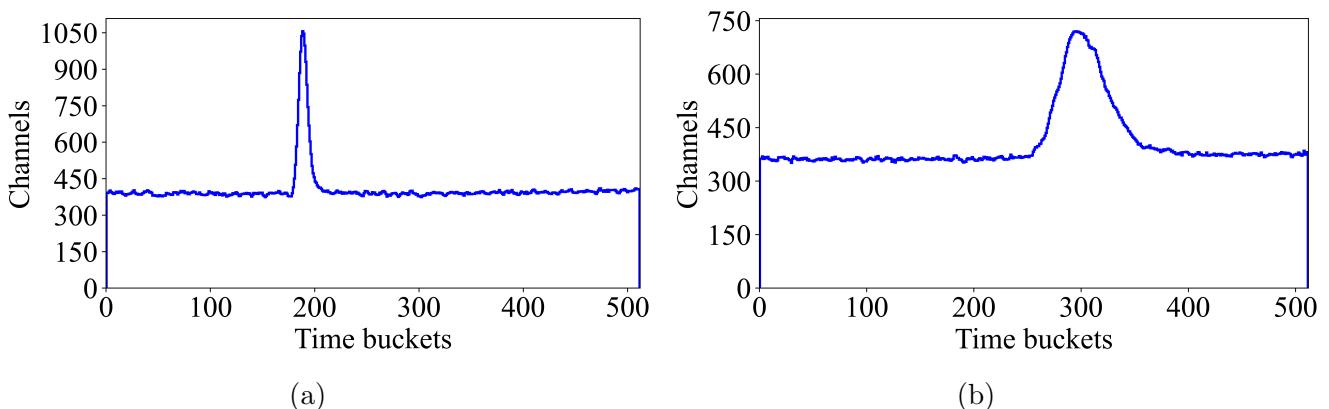


Figura 4.1: Exemplos de sinais produzidos pelos canais do detector. Em ?? o sinal possui apenas um pulso, enquanto em ?? há vários pulsos em sobreposição, formando um único pulso com largura maior que em ??.

No eixo x , cada um dos 512 *time buckets* possui largura de 195 ns. No eixo y temos a carga acumulada no detector para cada *time bucket*. Em ?? vemos um sinal com um pedestal (fundo ou *baseline*) com altura entre 300 e 450, e um pulso estreito em cima. Como dito na seção ??, os elétrons que surgem da ionização do gás são conduzidos perpendicularmente pelo campo elétrico até o detector. A interação da partícula com o gás é evidenciada justamente pelo pulso presente em ???. Cada pixel i do detector está em uma posição (x_i, y_i) , o centroide

de cada gaussiana fornece a coordenada em t (*bucket*) para então ser convertida na posição em z da partícula detectada, e a carga do ponto é a área do pulso (gaussiana com centroide t) sem a *baseline*.

Para ?? temos apenas um pulso, o que significa que há um feixe paralelo àquele canal do histograma, afinal, temos apenas um único ponto (x, y, z). Já para ?? temos o que é chamado de *gaussian mixture*, que é a presença de várias gaussianas sobrepostas. Esse tipo de sinal corresponde ao feixe indo perpendicularmente ao pixel do detector. A ilustração desse problema está na figura ??, que mostra o processo da passagem de uma partícula carregada e como o sinal é gerado a partir disso. As gaussianas presentes em ?? devem ter a mesma largura da gaussiana presente em ??.

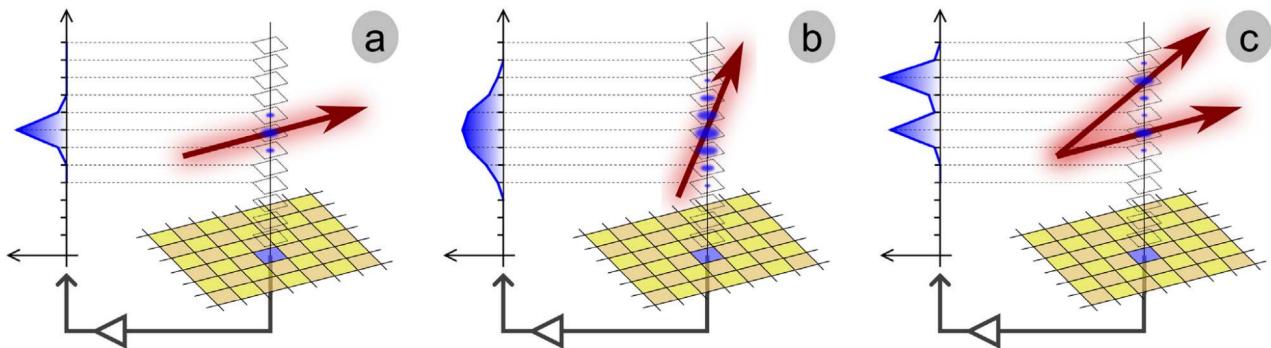


Figura 4.2: Ilustração que mostra a variação no formato da carga coletada a partir da passagem de uma partícula carregada dentro do TPC, onde o plano do detector está embaixo. No lado esquerdo de cada imagem, a distribuição do sinal coletado por um único pad (escuro) do plano de coleta é mostrado (o canal eletrônico de leitura é representado pela seta cinza em negrito). No caso de uma trajetória quase horizontal (a), o sinal é uma distribuição estreita, enquanto para uma trajetória próxima a uma direção vertical (b), a distribuição deve ser muito mais ampla (vários picos devem ser extraídos desse sinal). A última imagem ilustra o caso em mais de uma trajetória de partículas contribui para o sinal[GET].

Para analisar os pulsos devemos então remover a *baseline* dos sinais. O fundo não é trivial de se determinar, pois não é analítico, oscilando muito entre os canais.

4.1.1 Remoção do fundo

A primeira tentativa de estimar o sinal sem o fundo é usando transformada de Fourier e um filtro passa-baixa. Seja $f(t)$ uma função qualquer, sua transformada de Fourier é dada por

$$\hat{f}(v) = \mathcal{F}[f(t)] = \int_{-\infty}^{\infty} f(t) e^{-2\pi i vt} dt. \quad (4.1)$$

Primeiro calculamos a transformada de Fourier $\hat{f}(v)$ do sinal, em seguida por $\text{sinc}(\nu/a)$, onde

$$\text{sinc } x \equiv \frac{\sin(\pi x)}{\pi x}, \quad (4.2)$$

onde a é um fator de escala. A função sinc foi escolhida para tirar vantagem do Teorema da Convolução, dado por

$$\mathcal{F}^{-1}[\hat{f}(\nu)\hat{g}(\nu)] = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau, \quad (4.3)$$

onde $(f * g)(t)$ é a convolução entre $f(t)$ e $g(t)$. Multiplicar o sinal transformado por $\text{sinc}(\nu/a)$ e depois inverter inverte a transformação é o mesmo que convoluir o sinal original com a transformação inversa de $\text{sinc}(\nu/a)$, pois

$$\mathcal{F}^{-1}[\text{sinc}(\nu)] = \text{rect}(t) \equiv \begin{cases} 1, & -\frac{1}{2} < t < \frac{1}{2} \\ 0, & \text{qualquer outro } t \end{cases}, \quad (4.4)$$

é uma função que representa uma janela retangular. Resultados desse procedimento estão na figura ??.

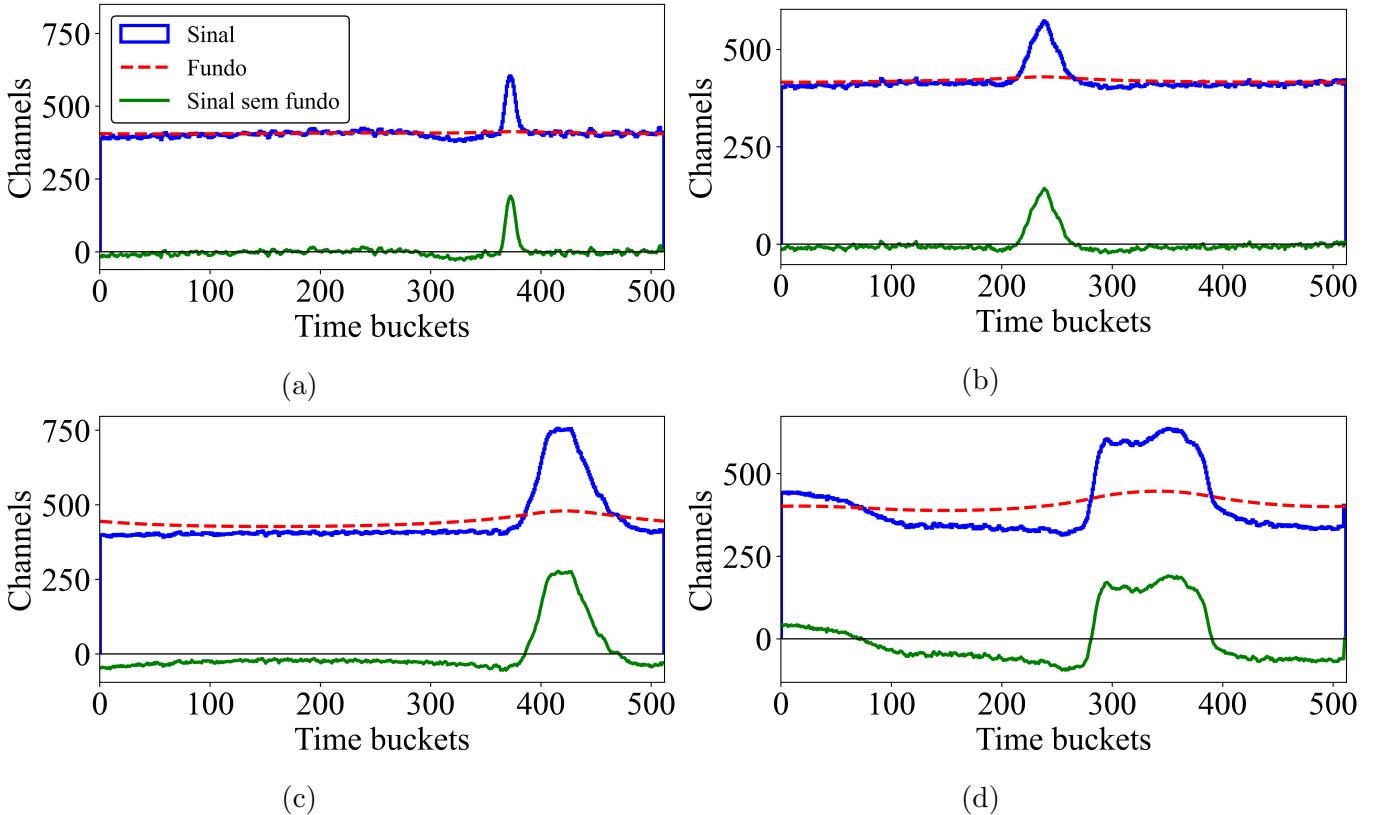


Figura 4.3: Histogramas com as respectivas *baselines* (linhas tracejadas) estimadas pelo método da convolução. O espectro resultante (sem o fundo) está em verde.

Esse não é o melhor método pois, em muitos casos, acaba estimando o sinal original, na região do pulso, menor do que deveria ser, fazendo com que o sinal tenha menos carga do que deveria. Isso ocorre pois o sinal de fundo não é analítico, dificultando muito o problema.

Para um resultado mais eficiente, o fundo será determinado usando o algoritmo *background removal* da biblioteca *TSpectrum* do *ROOT* [root]. A função tem a capacidade de separar o fundo dos picos presentes no espectro[BKG·1, BKG·2, BKG·3]. Exemplos de estimativa do

fundo estão na figura ??.

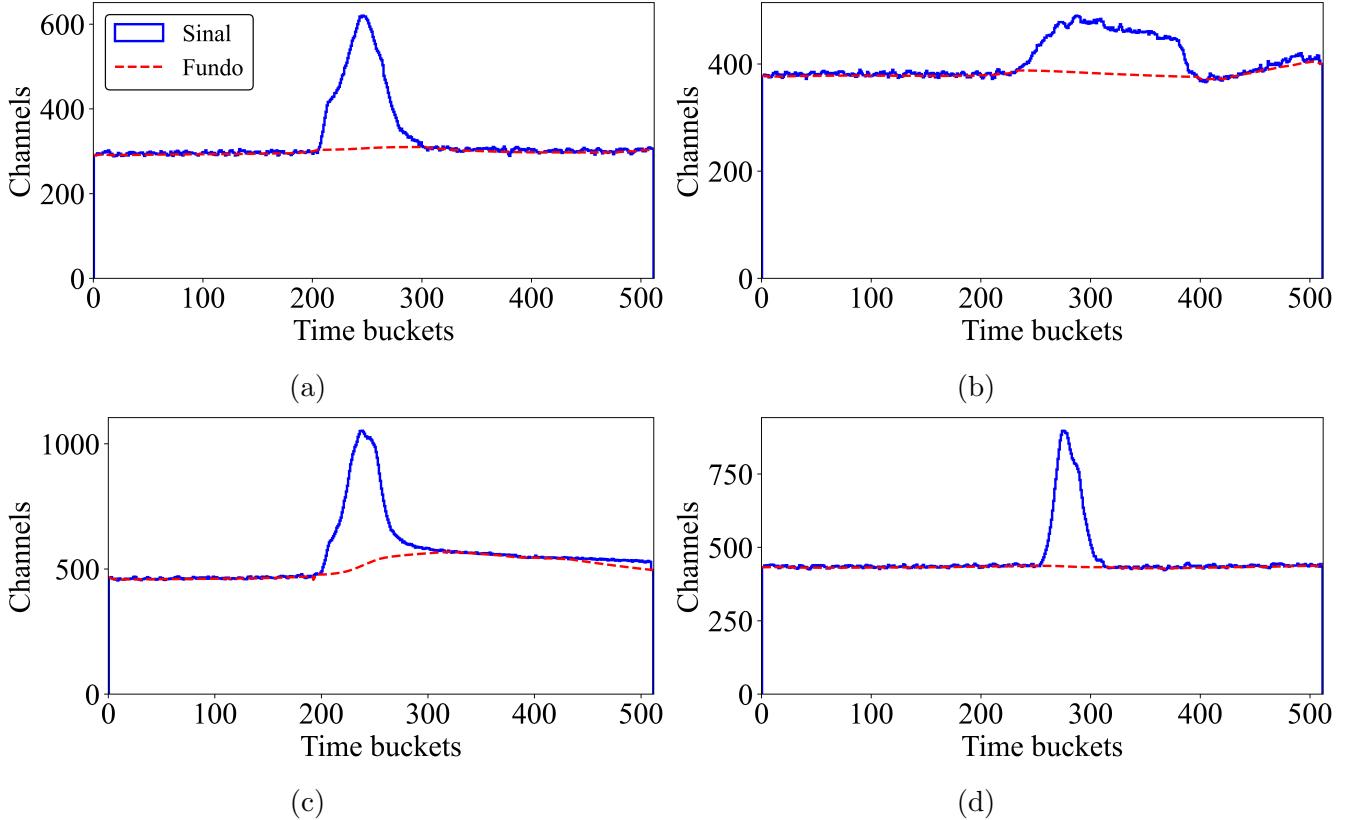


Figura 4.4: Histogramas com as respectivas *baselines* (linhas tracejadas) calculadas pelo *TSpec-trum*.

Para evitar valores negativos após a remoção do fundo, o valor mínimo do sinal sem o fundo é zero.

Sem o fundo podemos buscar por todos os picos e suas cargas correspondentes no sinal. Não podemos detectar diretamente todos os picos pois muitos deles estão em sobreposição. Para isso será feita a deconvolução do sinal.

4.1.2 Deconvolução do sinal

Para aumentar a resolução dos picos será usado o algoritmo *gold deconvolution* presente na biblioteca *TSpec-trum* do *ROOT*[[paper](#)·[gold](#)·[deconv](#)]. O algoritmo tem como objetivo fazer a deconvolução do espectro, gerando uma função resposta de acordo com o sigma esperado para os pulsos. Isso significa que devemos descobrir qual o valor de sigma dos pulsos para buscar a função resposta.

O sigma dos pulsos deve ser o mesmo de um sinal que possui apenas um pico. Ou seja, podemos determinar o sigma fazendo a análise de sinais que possuem apenas 1 pico, fazendo um ajuste pelo método dos mínimos quadrados (MMQ) de uma gaussiana. Para buscar espectros com apenas um pico foi usado o algoritmo de detecção de picos do *scipy*[[scipy](#)] e para o ajuste da gaussiana foi usada o pacote *lmfit* [[lmfit](#)]. O valor de sigma encontrado foi de 4.09 (17) *time*

buckets.

O sigma escolhido foi ligeiramente maior, pois em alguns casos a deconvolução separava um pico real em dois. O valor de sigma usado na deconvolução foi de 4.30 *time buckets*. Devemos também escolher o número de iterações do algoritmo de deconvolução. O número de iterações escolhido foi de 700, menos que isso o algoritmo não estava separando totalmente picos sobrepostos. O limiar para a escolha de um ponto como um pico é ter altura maior que 20% do valor máximo do sinal. Resultados da deconvolução estão na figura ??.

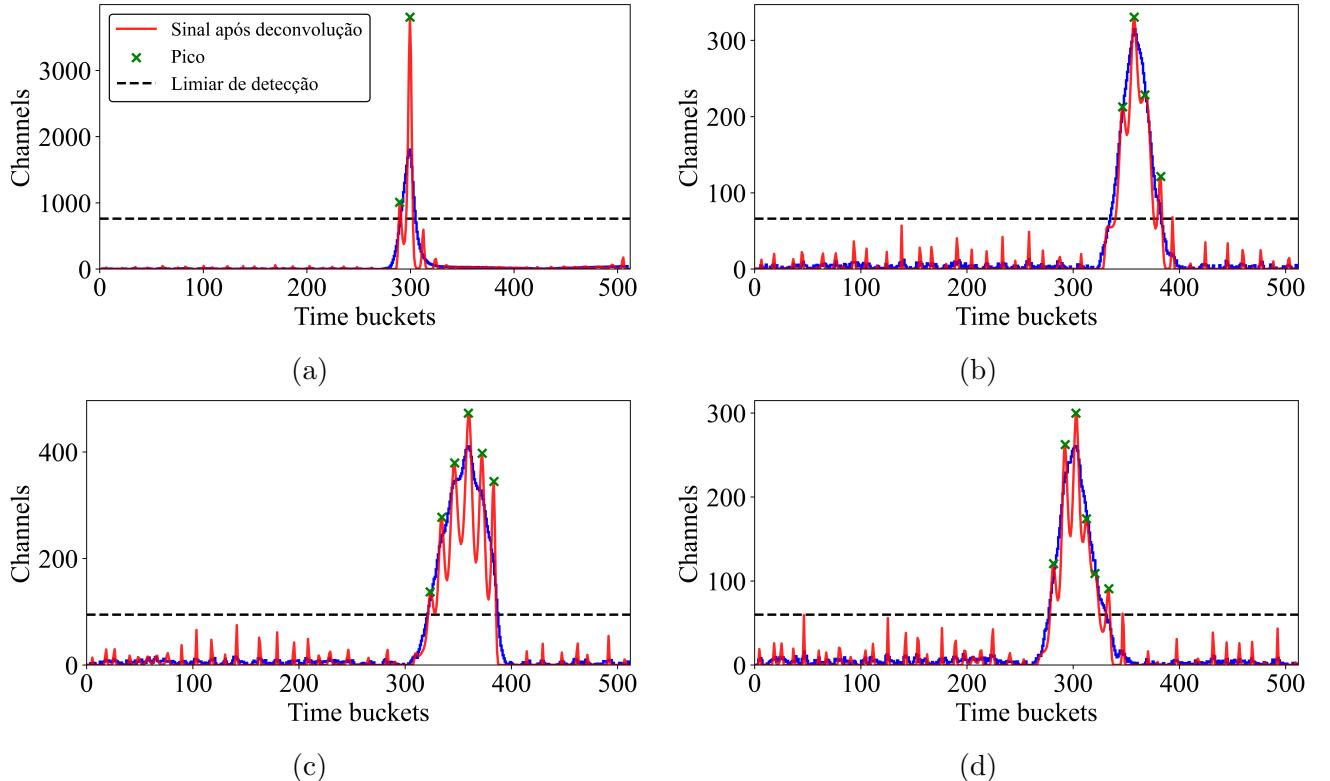


Figura 4.5: Histogramas sem as *baselines* antes (em azul) e depois da deconvolução (em vermelho). Os picos (em verde) e o limiar (linha tracejada preta) de detecção também estão indicados.

O algoritmo de deconvolução também retorna a posição dos centroides encontrados. A execução de 200.000 sinais, desde a estimativa e remoção do fundo, até a detecção dos centroides, demora cerca de 23.25 minutos, usando o processador Ryzen 5 3600X.

Para determinar a carga de cada ponto temos que calcular a área do centroide do pico detectado. A área do sinal antes e depois da deconvolução é a mesma, mesmo para a região dos pulsos, portanto podemos olhar diretamente para o sinal após a deconvolução. Para achar a área podemos calcular o sigma dos pulsos após a deconvolução, para determinar a área como uma simples integral gaussiana. O sigma dos pulsos após a deconvolução é $\sigma_{dd} = 1.1543$ (44) *time buckets*. Com isso podemos calcular a carga acumulada para cada ponto descoberto do evento. A carga acumulada Q para cada ponto i é dada por:

$$Q = \int_{-\infty}^{\infty} Ae^{-(t'-t_i)^2/2\sigma_{dd}^2} dt' = A |\sigma_{dd}| \sqrt{2\pi}, \quad (4.5)$$

onde A é a amplitude do ponto com centroide t_i e desvio padrão após a deconvolução σ_{dd} . Com esse procedimento podemos então reconstituir os eventos (nuvens de pontos), obtendo, para cada evento, todas as coordenadas x, y, t e Q de cada ponto. A figura ?? mostra exemplos de eventos reconstruídos.

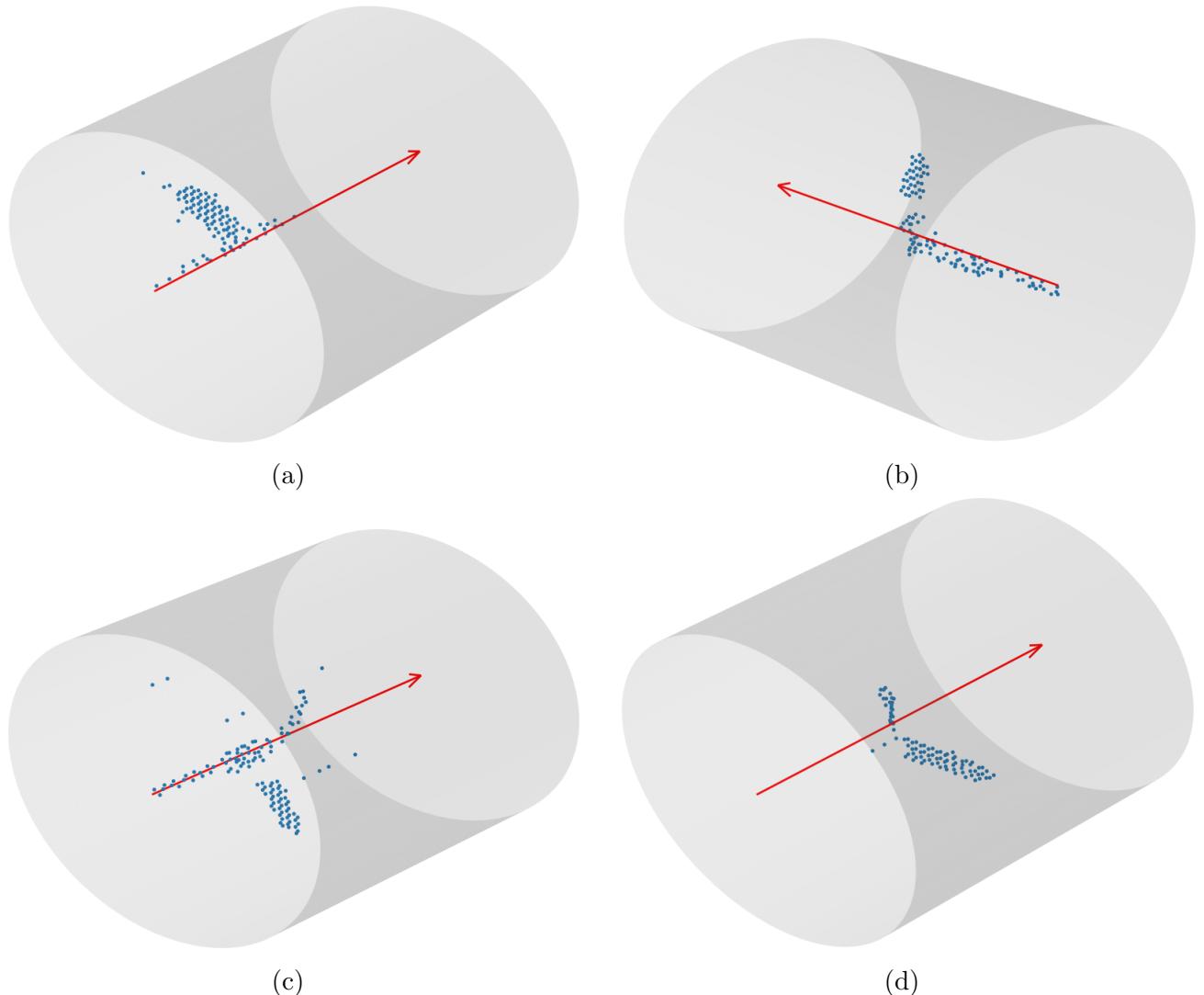


Figura 4.6: Exemplos de eventos reconstruídos através da análise dos sinais. A seta vermelha indica o sentido do feixe.

4.2 Análise dos pulsos com *machine learning*

Com *machine learning* temos a possibilidade de criar algoritmos extremamente complexos sem definir operações explícitas. Podemos nos basear na metodologia da seção anterior que é dividir o problema em três etapas: remover o fundo, fazer a deconvolução e por fim detectar os picos. As subseções seguintes irão descrever uma rede neural para cada etapa.

4.2.1 Rede neural para o fundo

O objetivo é criar uma rede neural que reproduza o comportamento do algoritmo *background removal* que estima o fundo do sinal, tentando reproduzir resultados muito similares. A rede neural é supervisionada, onde os dados de entrada são os sinais brutos e as saídas devem ser os fundos de cada sinal. A arquitetura está na figura ??.

Estimar o fundo é uma tarefa muito complexa pois ele não é analítico, podendo muitas vezes fazer com que o fundo tenha saltos em diferentes *time buckets*.

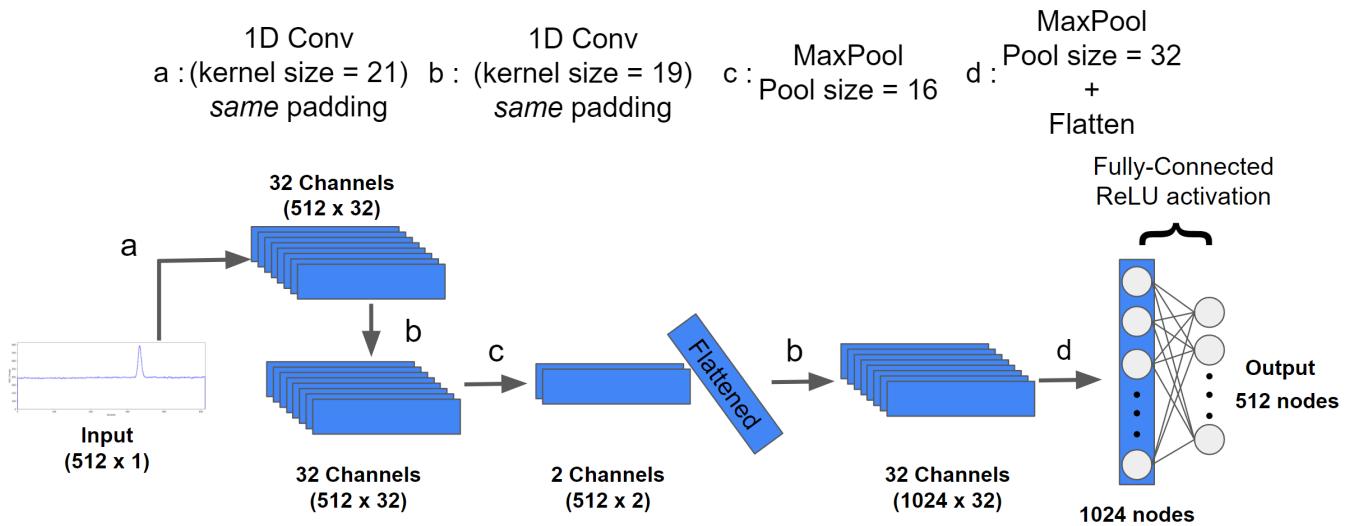


Figura 4.7: Arquitetura da rede neural que faz a inferência do fundo. O vetor de entrada deve ter dimensionalidade 512×1 . Todas as partes com convolução não possuem o parâmetro *bias*.

A entrada da rede é o sinal com dimensionalidade 512×1 . Há duas convoluções seguidas (passagens *a* e *b*) com o *padding same*, seguida de um *Max pooling*. Os dois canais restantes sofrem um *flat* para então passar mais uma convolução com *padding same* e filtros de tamanho 19 seguido de *Max pooling* e *Fully Connected* com ativação *ReLU*. Toda a rede foi construída usando o TensorFlow 2 e possui um total de 545.536 parâmetros, todos treináveis. O tamanho dos filtros das convoluções devem levar em conta a largura do pulso, devendo ser no mínimo maior que a largura, a fim de cada *kernel* visualizar um pulso completo na convolução.

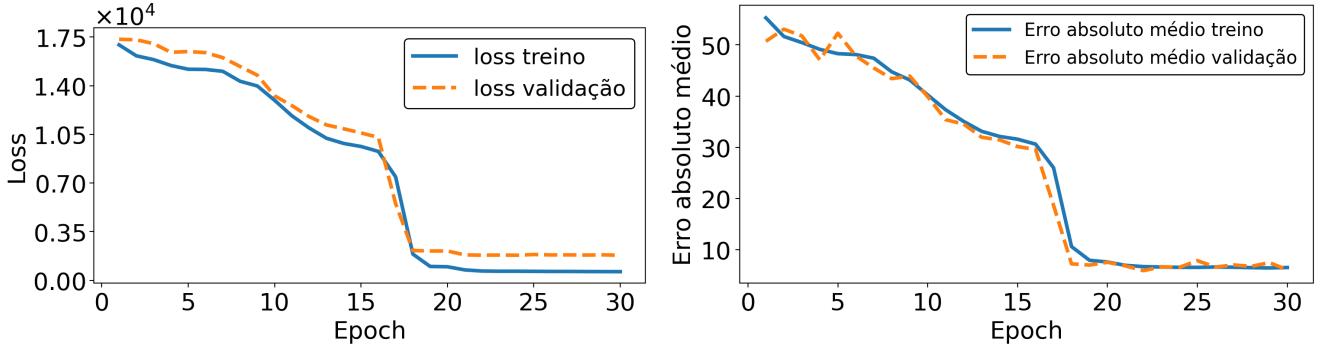
A camada final com ativação *ReLU* garante o valor mínimo de saída em 0 e, principalmente, pelo fato de não causar problemas à minimização do gradiente [VGP]. Foram testadas diversas combinações e a mostrada na figura ?? é a que obteve os melhores resultados.

Para o treino foram usados 160.000 sinais para treino e 40.000 para validação. O *loss* foi escolhido como sendo o erro quadrático médio (equação ??, o otimizador foi o *ADAMAX* [ADAMAX], com *learning rate* de 0.0005, e métrica para avaliação foi o erro médio absoluto, dado por.

$$E = \frac{1}{N} \sum_{i=1}^N |x_i - \hat{x}_i|, \quad (4.6)$$

onde E é o erro absoluto médio, N é o número de pontos e x_i o ponto da saída da rede

para ser comparado com o ponto original \hat{x}_i . Foram 30 *epochs* e o *batch-size* foi 8. O treino foi realizado no Google Colaboratory [google colab] usando a GPU (graphics processing unit) NVIDIA Tesla P100 e durou cerca de 34 minutos. Os resultados do treino estão na figura ??.



(a) *Loss* dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch* no treino da rede dada por ??.

(b) Erro absoluto médio dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch* no treino da rede dada por ??.

Figura 4.8: Resultados do treino da rede neural dada por ???. A rede atingiu seu melhor resultado a partir da *epoch* 20 aproximadamente, quando começa um platô no *loss*.

Exemplos de resultados da rede podem ser vistos na figura ???. A previsão do fundo possui um erro absoluto nos dados de treino de 6.5315 Channels e nos dados de validação de 6.0783 Channels. Podemos subtrair o fundo do sinal original (colocando o valor minimo da subtração em 0). Comparando o erro médio absoluto de 200.000 sinais sem o respectivo fundo (resultante do algoritmo do TSpectrum) com o resultado da rede neural obtemos 4.5 Channels.

Rede neurais convolucionais têm a vantagem de usarem poucas variáveis e serem facilmente paralelizadas em sua execução. Uma vantagem de redes neurais é o seu tempo de execução. Empiricamente a rede neural pode executar 200.000 sinais em apenas 8s (ou 25.000 sinais por segundo), sendo extremamente eficiente em tempo.

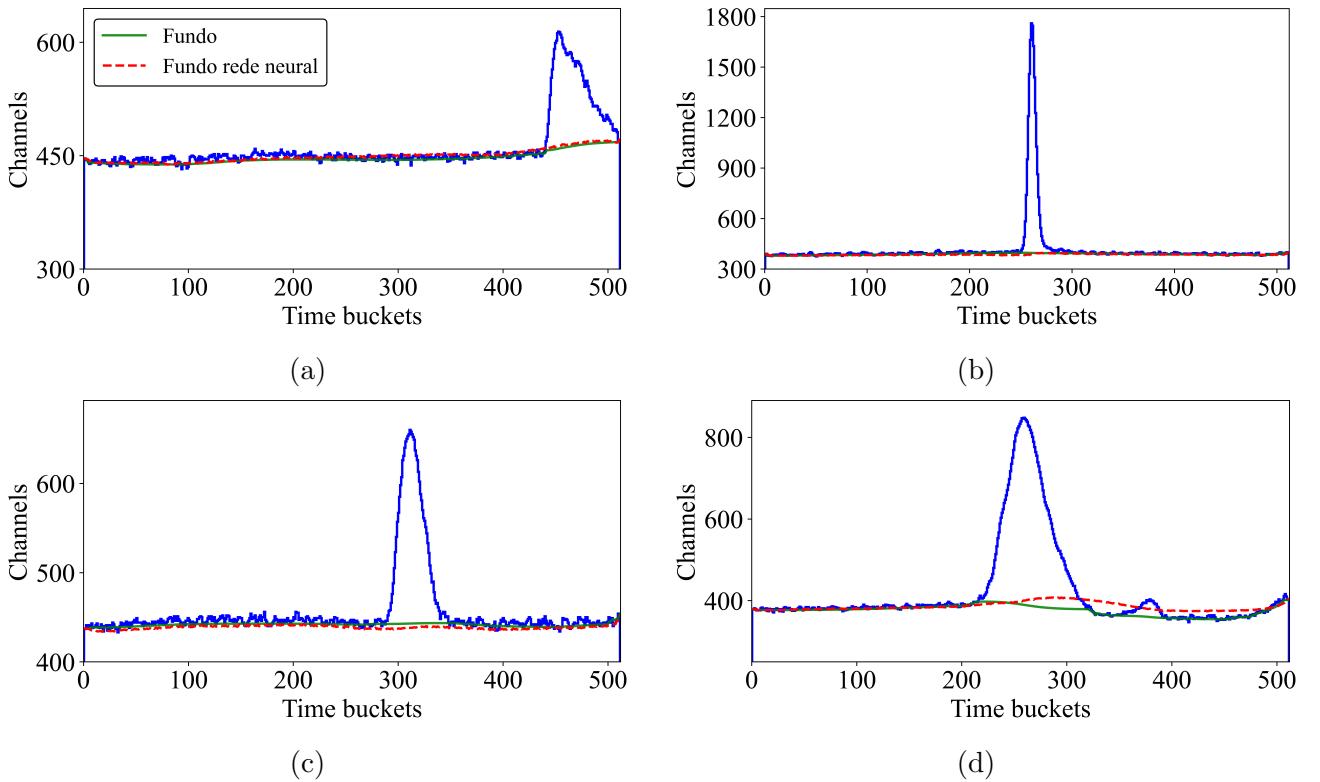


Figura 4.9: Exemplos da rede neural dada por ?? em comparação com a saída do *TSpectrum*.

Nos exemplos mostrados nas figuras ??, ?? e ?? os fundos dos sinais possuem grande flutuação e a rede neural se mostrou eficaz na previsão. No exemplo ?? a *baseline* do sinal é extremamente complexo de se determinar pois o sinal, do canal 300 a 500 aproximadamente, varia em cerca de 50 unidades em y . Apesar da rede neural determinar o fundo acima do fundo original, ao subtrair o espectro do fundo e colocar o valor mínimo em 0, o pulso presente entre os canais 200 e 300 é praticamente inalterado.

Com os resultados da rede podemos então seguir a diante para criar a rede neural que faz a deconvolução do espectro sem a *baseline*.

4.2.2 Rede neural para a deconvolução

Podemos usar a mesma abordagem da rede neural anterior, fazer uma sequencia de convoluções e por fim uma *fully connected* com ativação *ReLU*, pois precisamos ter o valor mínimo do espectro em 0. Os filtros das convoluções precisam ter tamanho mínimo de duas vezes o sigma das gaussianas para atuarem sobre cada pulso do espectro. A entrada da rede é o sinal com o fundo subtraído e com mínimo em 0. A saída é a deconvolução dada pelo algoritmo *gold deconvolution* na biblioteca *TSpectrum* do ROOT. A figura ?? mostra a arquitetura da rede de deconvolução.

A rede é a sequência de duas convoluções com 32 filtros, *valid padding* e *kernels* de tamanho 19 e 17, respectivamente, seguida de *Max pooling* com *pool size* igual à 16. No final há o *flat* na camada para seguir com uma *fully connected* com ativação *ReLU*. O *valid padding* se mostrou

mais eficiente para a convergência da rede. Toda a rede foi construída usando o TensorFlow 2, possuindo 508.000 parâmetros treináveis.

Assim como na rede anterior, foram usados 160.000 sinais para treino e 40.000 para validação. O *loss* foi escolhido como sendo o erro quadrático médio, o otimizador foi o *ADAM*, com *learning rate* de 0.0005 porém com o parâmetro *clipnorm* igual a 0.45. A métrica para avaliação foi o erro médio absoluto. Foram 75 *epochs* e o *batch-size* foi 8. Os resultados do treino estão na figura ??.

Alterar a norma do gradiente (usar o parâmetro *clipnorm* = 0.45) significa que, caso a norma do vetor do gradiente exceda 0.45, então o valor da norma é reajustado para o *threshold* escolhido (0.45). Isso faz com que não ocorra problemas comuns como o gradiente sumir [VGP, ADAMAX], o que estava acontecendo especificamente nesse caso.

O treino foi realizado no Google Colaboratory [google.colab] usando a GPU NVIDIA Tesla P100 e demorou cerca de 54 minutos. Os resultados do treino estão na figura ???. Empiricamente a rede é capaz de executar 200.000 sinais em 5.4 segundos (ou 37.000 sinais por segundo).

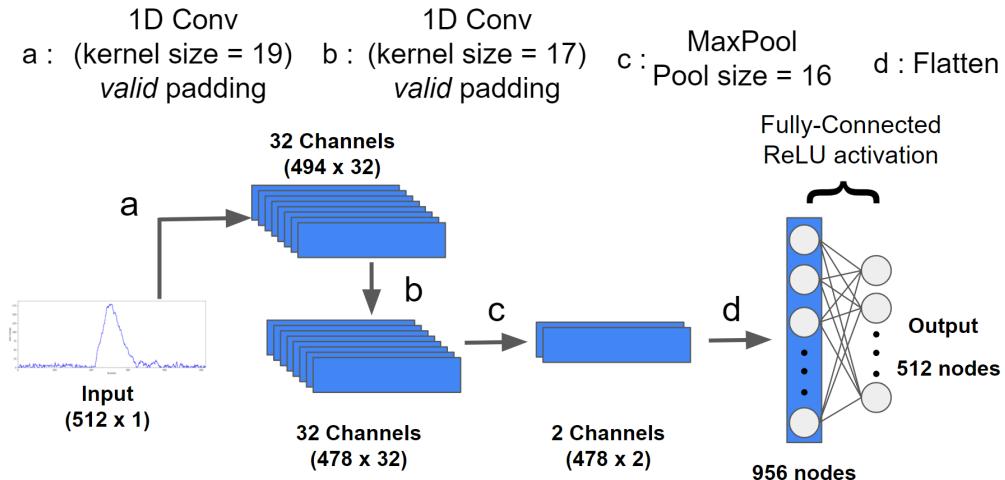
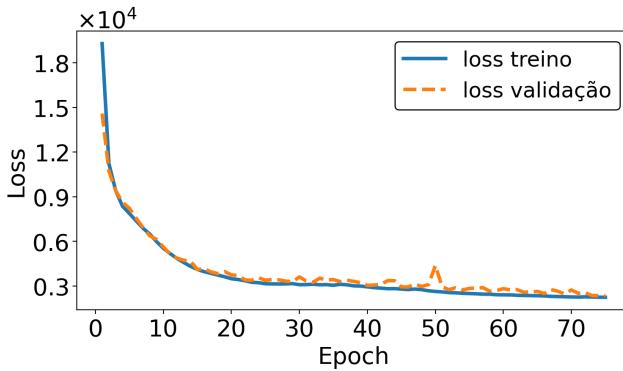
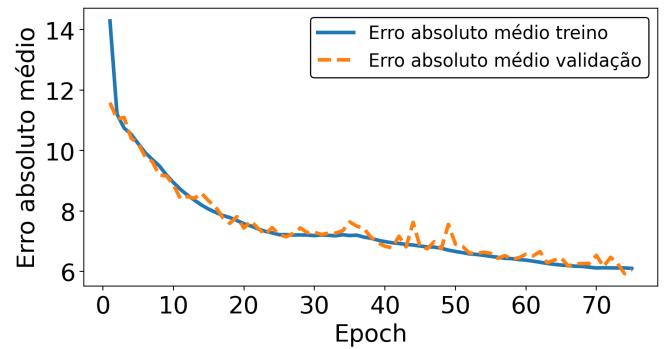


Figura 4.10: Arquitetura da rede neural que faz a inferência da deconvolução do espectro. O vetor de entrada deve ter dimensionalidade 512 x 1. Todas as partes com convolução não possuem o parâmetro *bias*.

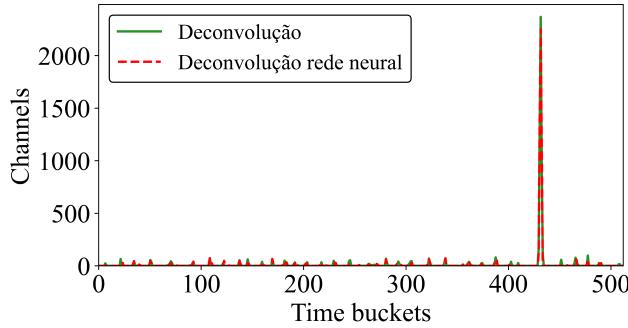


(a) *Loss* dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch* no treino da rede dada por ??.

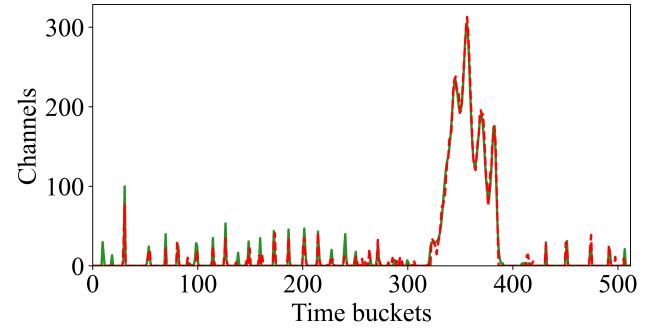


(b) Erro absoluto médio dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch* no treino da rede dada por ??.

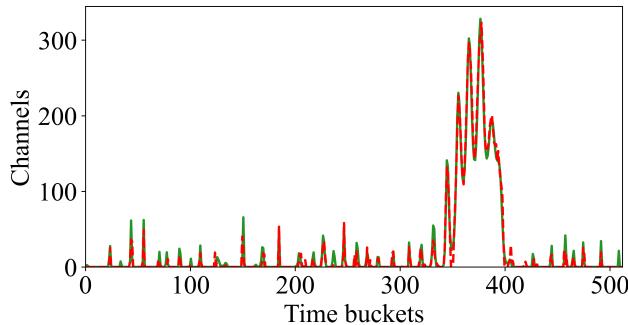
Figura 4.11: Resultados do treino da rede neural dada por ??.



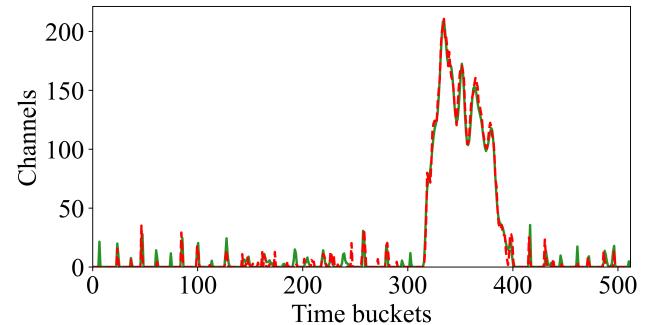
(a)



(b)



(c)



(d)

Figura 4.12: Exemplos de deconvolução da rede neural dada por ??.

4.2.3 Unificando as duas redes neurais

Com a rede que prevê o fundo e a que faz a deconvolução funcionando podemos agora testar se acopladas elas geram resultados satisfatórios, verificando a qualidade da detecção de picos. Usando novamente o TensorFlow 2 podemos carregar as redes neurais já treinadas e usar como se fosse uma única rede neural. A rede receberá de entrada o espectro original e devolverá tanto o fundo quanto o espectro deconvoluído. A arquitetura da rede está na figura ??.

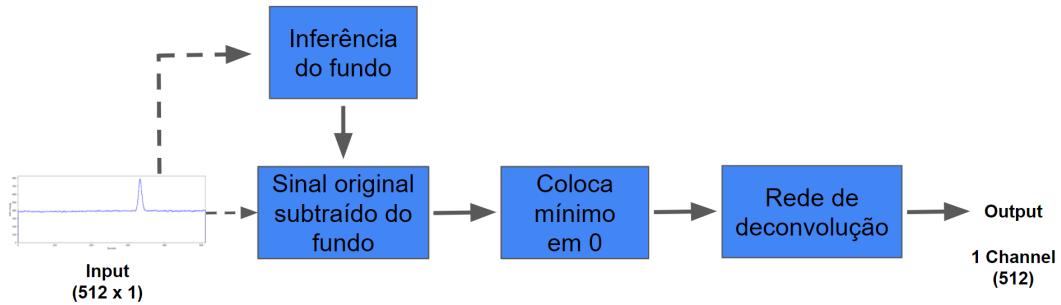


Figura 4.13: Arquitetura da rede neural que faz a inferência da *baseline* e depois faz a deconvolução do espectro. O vetor de entrada deve ter dimensionalidade 512×1 .

A rede é apenas a sequencia das redes anteriores, ou seja, o espectro passa pelo cálculo da *baseline*, então o espectro original é subtraído dessa *baseline* (colocando o valor mínimo em 0) e por fim ocorre a deconvolução. Pelo fato de cada parte ser treinada de modo separado não há necessidade de treinar a rede unificada, apenas carregar os valores das variáveis treinadas de cada parte. A rede unificada possui 1.053.536 de parâmetros.

Agora devemos comparar a saída dessa nova rede com a saída desejada. Podemos usar novamente o erro médio absoluto para fins de comparação. O erro médio absoluto, para os 200.000 sinais, é de 7.45 ADC. O valor é um pouco maior que o encontrado anteriormente com a rede de deconvolução, portanto podemos verificar a detecção de picos no espectro resultante.

Com o espectro deconvoluído devemos então detectar todos os picos em cada sinal. A detecção será feita com o *SciPy* que possui rotinas para a detecção. A função que usada é a “`find_peaks`”, onde é possível determinar altura mínima de detecção, distância mínima entre picos, dentre outros parâmetros, o que torna possível calibrar a detecção para o caso discutido. O objetivo será fazer a comparação entre os picos resultantes do algoritmo de deconvolução (na biblioteca *TSpectrum* do ROOT) e os resultantes do algoritmo do *SciPy*.

A comparação se limitou a detecção de 0 a 6 picos detectados pela deconvolução, que é o intervalo que faz sentido físico (mais que 6 picos detectados indica um sinal muito ruidoso). Caso o *SciPy* detecte mais que 6, então o espectro é descartado. Primeiro deve-se ver a comparação do número de espectros detectado por evento de cada algoritmo. O histograma da figura ?? mostra essa comparação.

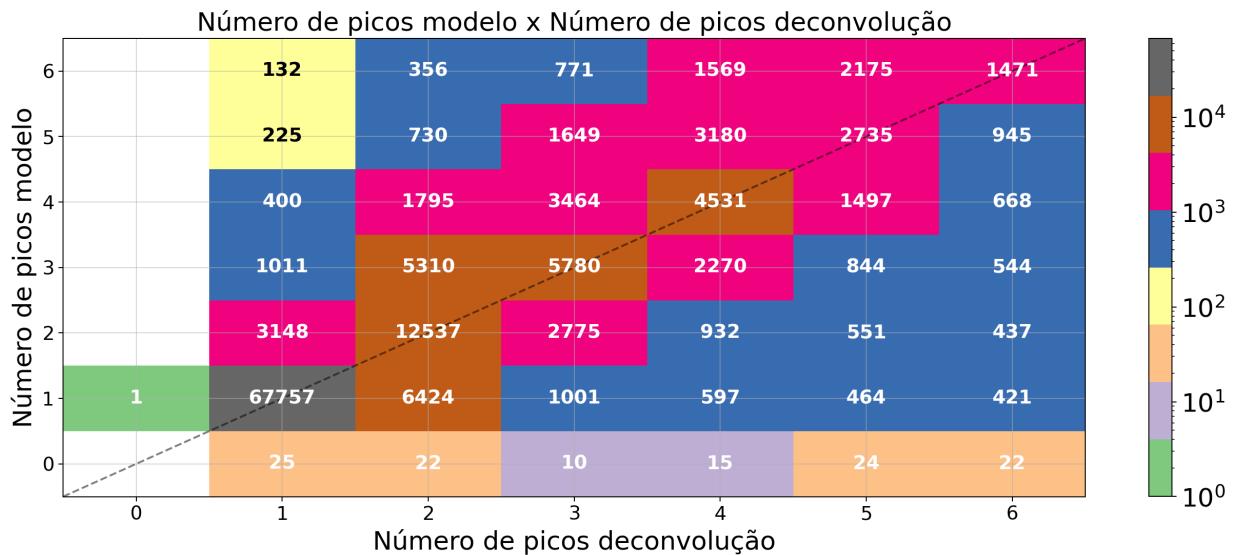


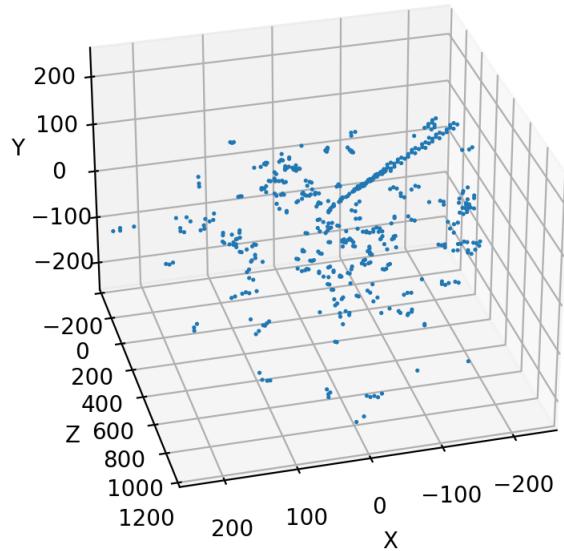
Figura 4.14: Histograma bidimensional que mostra em x a contagem do número de picos detectados por sinal (dado pelo *TSpectrum*) pela deconvolução e em y a contagem do número de picos detectados por sinal (resultante da rede neural) pelo *scipy*. O número de contagens está marcado em cima de cada *bin*.

A linha tracejada serve de referência para os caso em que o número de picos detectador por ambos algoritmos foi igual. A dispersão de detecção se dá em torno dessa e se mostrou com uma diferença muita baixa, indicando que o processamento de sinal pela rede neural seguido pela detecção de picos com o *scipy* é equivalente ao dado pelo uso dos algoritmos do *TSpectrum*. Para os casos em que o número detectado de picos foi o mesmo, a diferença da posição dos centroide é de 0.11 canais, mostrando que os métodos são praticamente equivalentes.

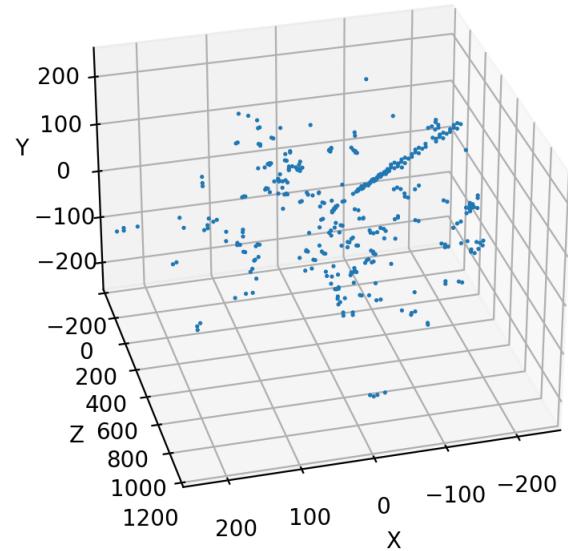
Com relação a eficiência, a rede neural da figura ?? processa 200.000 sinais em 8 segundos. Agora levando em conta a detecção de picos com o *SciPy*, o tempo total para processar 200 mil sinais é de cerca de 25 s. Nem foi preciso uma rede neural para os picos e a eficiência em tempo já é 30 vezes maior em comparação ao método analítico.

Exemplos da reconstrução das nuvens de pontos usando *machine learning* com detecção de picos pelo *SciPy* estão na figura ???. É possível perceber uma pequena melhora no ruído dos eventos, pois o com o *SciPy* podemos calibrar melhor a detecção. Além disso há uma correlação de amplitude do pico detectado entre espectro sem o fundo e o espectro sem o fundo após a deconvolução, como mostrado na figura ???. Colocando a condição de que, para cada pico detectado, a razão entre a amplitude do espectro após a deconvolução e do espectro original para o ponto seja maior que 3.8 faz com que a quantidade de pontos por evento caia em cerca de 35% em relação ao método analítico.

PointCloud Original

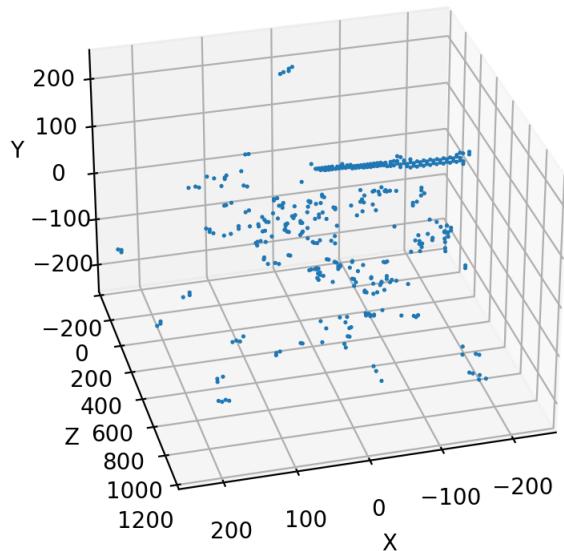


Reconstrução com machine learning

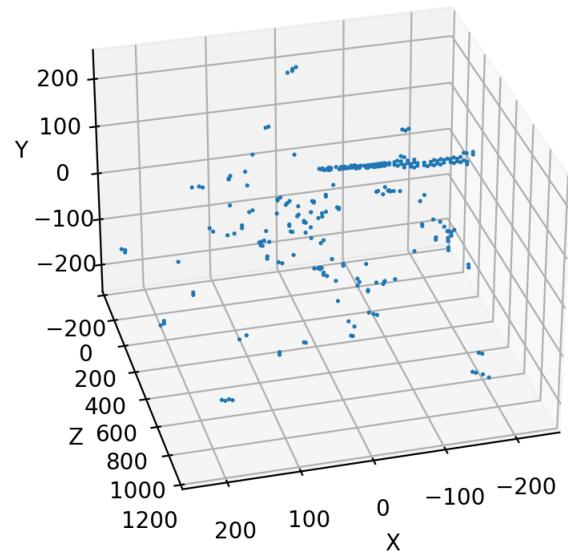


(a)

PointCloud Original



Reconstrução com machine learning



(b)

Figura 4.15: Resultados da reconstrução de eventos por *machine learning*.

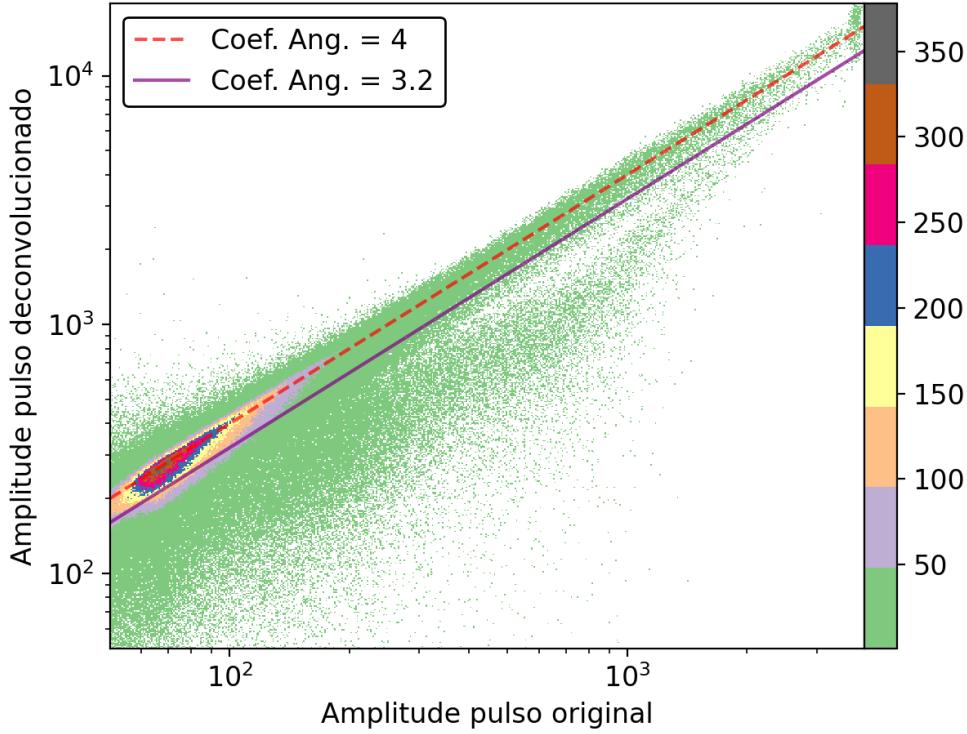


Figura 4.16: Histograma bidimensional que mostra a relação, de cada pico detectado, entre a amplitude do pico após a deconvolução no eixo y e antes da deconvolução no eixo x . A linha tracejada indica a tendência da maior parte dos pontos. Já a linha sólida indica a região de corte dos pontos.

4.2.4 Detecção de picos

Como etapa final da análise com *machine learning* devemos analisar possíveis soluções para a detecção de picos. Esse é um problema extremamente complicado, pois há uma variação na quantidade de detecções muito grande e há um desbalanço muito grande na quantidade de pontos comuns (não sendo picos) e pontos que são picos. Por exemplo, caso haja um sinal que possui apenas um pico, devemos detectar uma posição, dar o valor de saída como 1, por exemplo, dentre 512 pontos, onde 511 terão o valor de saída como 0. Caso a rede determine que todos os pontos são não picos, ainda assim a acurácia seria maior que 99%.

Para resolver esse problema podemos nos basear na ideia de recortar regiões de interesse, como feito pela rede U-Net[unet]. Recortar regiões significa, nesse caso, ter uma rede neural com a saída com o mesmo tamanho do vetor de entrada (512) e saída com valores entre 0 e 1, onde 1 indica uma região com um pico e 0 não. Como temos as posições dos picos, basta acrescentar pontos simetricamente em torno do pulso. A figura ?? mostra um exemplo de um sinal após a deconvolução onde há o pico detectado e os pontos acrescentados para representar a região do pulso.

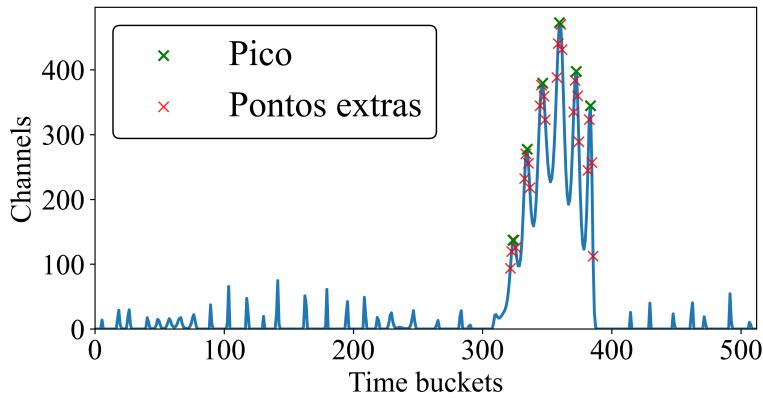


Figura 4.17: Sinal após a deconvolução que mostra o pico detectado mais os pontos adicionais que irão facilitar o trabalho da rede neural (evitar o desbalanço de classe). Foram acrescentados 2 pontos à esquerda e à direita.

A rede construída é uma convolução com *kernel* de tamanho 13 e *same padding* seguida de *Max-Pooling* e uma *fully-connected* com ativação sigmoide, possuindo um total de 263.104 parâmetros treináveis. Foram usados sinais que possuíam entre 1 e 6 picos, resultantes da saída do *SciPy*, o que resultou em 120.024 de dados para o treino e 30.006 para validação. A escolha pelos picos detectados pelo *SciPy* ao invés do TSpectrum é pela maior facilidade de fazer um ajuste fino na detecção, tornando a detecção de picos muito melhor. O *loss* escolhido foi a *binary cross-entropy*, o otimizador o *ADAM* com *learning rate* de 0.001. A métrica utilizada foi a acurácia binária. O treino também foi realizado por uma GPU NVIDIA Tesla P100 e durou cerca de 8 minutos com 12 *epochs*. A arquitetura da rede está na figura ?? e os resultados do treino estão na figura ??.

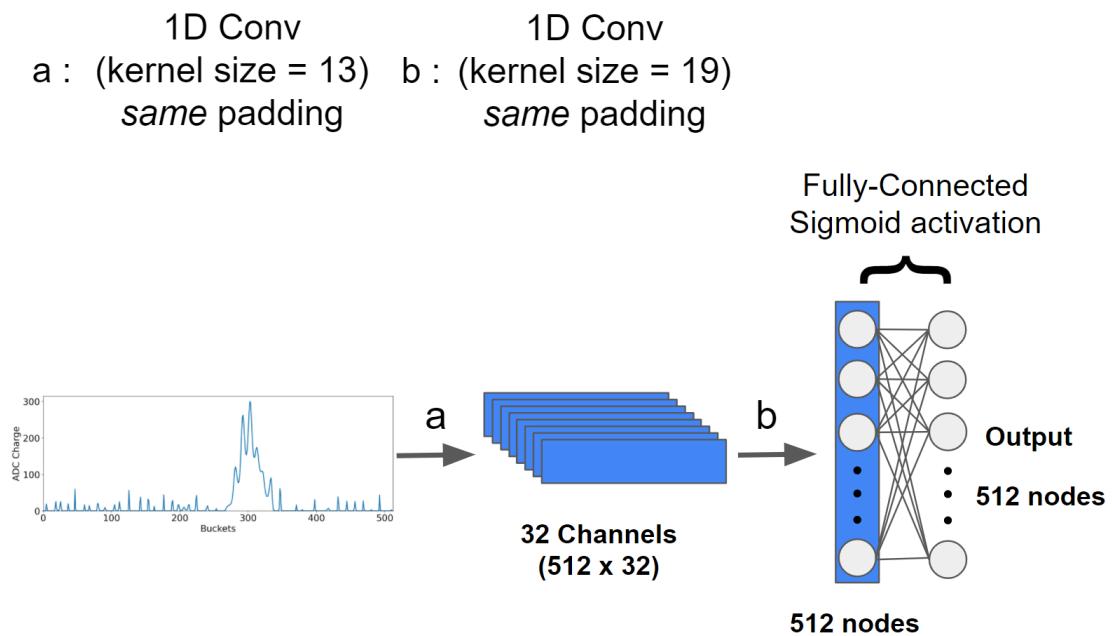
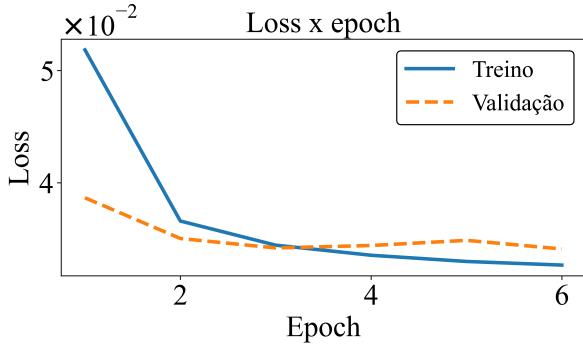
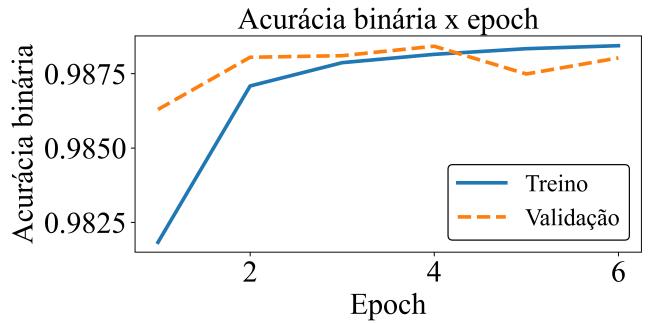


Figura 4.18: Arquitetura da rede neural que faz o recorte das regiões com picos. O vetor de entrada deve ter dimensionalidade 512 x 1.



(a) *Loss* dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch*.



(b) Acurácia binária dos dados de treino (linha contínua) e dos dados de validação (linha tracejada) em função da *epoch*.

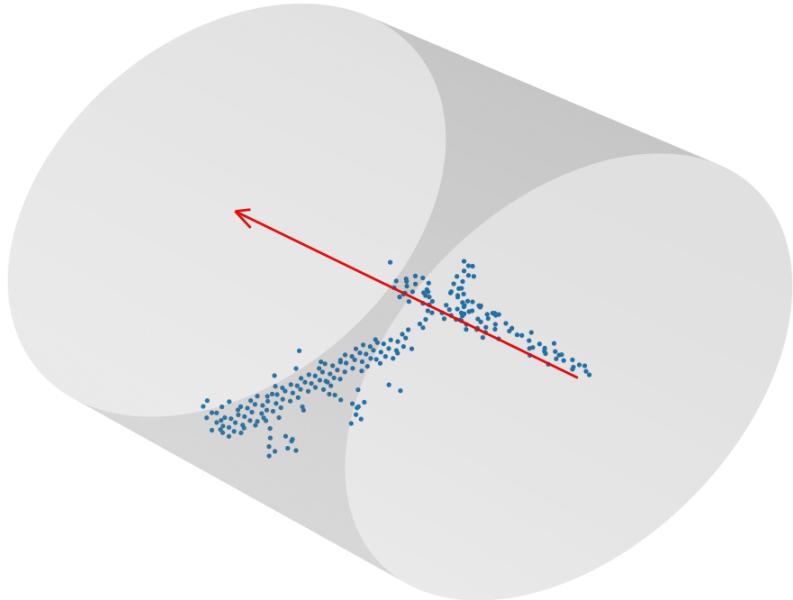
Figura 4.19: Resultados do treino da rede neural dada por ??.

A saída da rede neural é um vetor de tamanho 512 com valores entre 0 e 1. Exemplos da saída da rede estão na figura [xx]. Para obter os picos a partir desse vetor devemos, primeiro, identificar a segmentação feita, ou seja, saber separar as regiões recortadas e depois fazer uma média ponderada com o espectro de entrada (para achar o centroide). O tempo de processamento da rede neural é de 150.030 sinais em cerca de 4.11 segundos (aproximadamente 36.500 sinais por segundo). Para determinar os picos a partir da saída da rede o tempo é de aproximadamente 4.3 segundos, onde o algoritmo pode ser ainda mais rápido se for paralelizado.

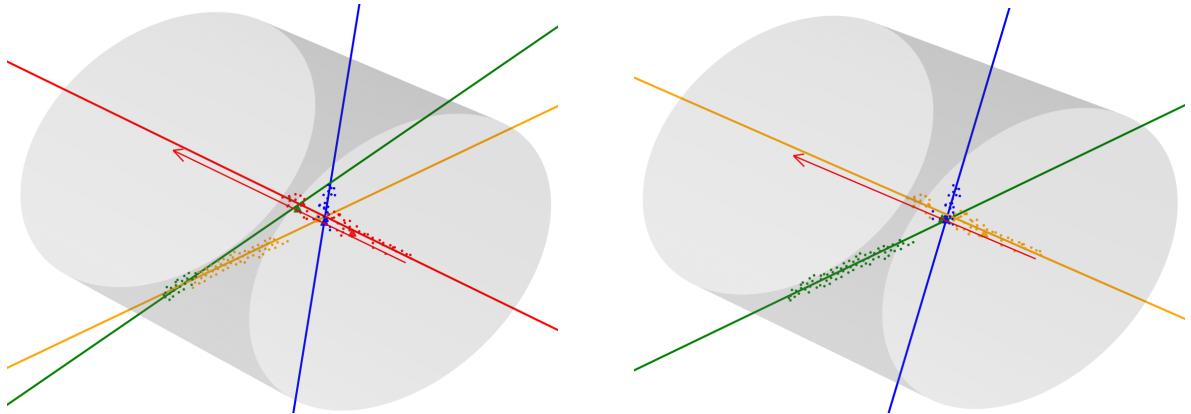
Capítulo 5

Análise das nuvens de pontos

Após a análise dos espectros, temos agora que trabalhar na etapa de detecção e reconstrução de trajetórias. O objetivo desta etapa é detectar *clusters*, que nesse caso são retas tridimensionais (o pAT-TPC só possui campo elétrico), distinguir cada reta como sendo ou o feixe ou a partícula espalhada e determinar o vértice de reação entre uma reta, que é de uma partícula espalhada, e o feixe (se houver). Após a seleção devemos selecionar os eventos que possuam espalhamento. Um exemplo de evento pode ser visto na figura ??.



(a) Exemplo de evento a ser analisado. Os pontos em azul são das partículas detectadas pelo TPC, a seta vermelha indicando o sentido do feixe e o TPC está representado pelo cilindro cinza.



(b) Evento sem a clusterização. As retas de cor amarela e verde são de um único *cluster*.

(c) Evento após a clusterização. Agora o evento possui as duas retas corretas, a azul e a verde.

Figura 5.1: Sequência de análise de um evento. Em ?? temos o evento que é recebido para ser analisado, em ?? temos o mesmo evento após o RANSAC (antes da clusterização) e ?? mostra depois da correção. As cores das retas são arbitrárias e servem apenas para a diferenciação.

A etapa anterior nos fornece arquivos no formato *ROOT* para cada *run* do experimento. O arquivo é dividido por eventos em que estão contidas informações, em *trees*, como coordenadas x, y, z, t, carga, tempo de voo, além de canais adicionais (indicadores de partículas presentes além da parte central do *micromegas*). Os arquivos foram lidos em Python usando a biblioteca Uproot[**uproot**].

A figura ?? nos mostra um exemplo que parece indicar mais de uma trajetória dentro do TPC, e é preciso separar os pontos de cada trajetória identificada. O objetivo é separar os pontos de *clusters* que contenham a informação do momento da partícula, obtida a partir das propriedades geométricas do versor das retas e da energia inicial da trajetória. A identificação pode ser feita sem o uso de algoritmos de machine learning, que chamarei de método usual, e

usando machine learning.

5.1 Método usual

Para identificar as retas (*clusters*) presentes em eventos como mostrados na figura ??, devemos usar estimadores robustos que consigam fazer a detecção mesmo com uma presença grande de *outliers*, pontos que são considerados ruídos. Existem muitos estimadores, que são apenas variações do Random sample consensus (*RANSAC*) [**ransac**], como o MLESAC[ref:MLESAC], *RRANSAC*, *PROSAC*, dentre outros, que estão presentes no *Point Cloud Library* (PCL)[**pcl**]. Todos esses se baseiam na ideia de estimar várias possíveis retas de forma aleatória e selecionam a melhor usando uma estimativa (por isso o nome *consensus*). Existe ainda a opção do *Hough Transform*[**hough**], porém é um algoritmo de difícil expansão e não demonstrou resultados bons.

Os algoritmos do PCL foram testados e os que mostraram os melhores resultados foram o *RANSAC* e o *PROSAC*, não havendo muita diferença entre os dois. O estimador usado foi uma variação do *RANSAC*, desenvolvida para fazer melhores estimativas em dados como os do pAT-TPC[**artigo**]. O algoritmo ??, que chamei de *Enhanced RANSAC* (Para você Juan: só quis diferenciar o nome, até pq eu quero colocar no pypi com esse nome, pode ser?), mostra o seu funcionamento. A melhor estimativa original (que retorna a melhor reta) era simplesmente o número total de *inliers* de um *cluster*. A nova estimativa C passou a ser

$$C = \sum_{i=0}^N \frac{d_i^2}{N}, \quad (5.1)$$

onde N é o número total de pontos de uma reta e d_i é a distância do i-ésimo ponto à reta.

O *RANSAC* consegue identificar apenas pontos de uma única reta, porém cada evento pode ter mais de uma reta. O novo algoritmo se baseia na ideia de usar o RANSAC sequencialmente, ou seja, no momento que um *cluster* tem um tamanho mínimo, então ele é guardado para depois, então, poder ser escolhidos dentre as melhores estimativas C . Com isso podemos selecionar quantas retas forem possíveis.

Para diminuir o número de pontos a serem analisados pelo algoritmo ??, passamos a *point-cloud* por dois filtros. O primeiro filtro exclui pontos baseado na sua carga. O segundo filtro, chamado de *outlier removal*, elimina pontos considerados *outliers* globais, analisando a vizinhança ao redor do ponto. O *outlier removal* está presente na biblioteca Open3D[**open3d**] no Python e funciona excluindo pontos muito isolados uns dos outros.

Uma mudança na eficiência do algoritmo ?? é na linha 2, chamada de *Random Sampling*. Podemos nos beneficiar do *Monte Carlo Rejecting* na escolha de dois pontos aleatórios. Por exemplo, caso sejam selecionados pontos muito próximos, o que implicaria em uma reta com poucos pontos, podemos descarta-los. Essa ideia faz com que a primeira etapa seja mais eficiente.

Algoritmo 1: Enhanced RANSAC

Dados: pointcloud, N , d_{min} , tam_{min}

- 1 **para** cada iteração $i = 1, 2, \dots, N$ **faz**
- 2 Seleciona dois pontos da *pointcloud* de modo aleatório (*Random Sampling*);
- 3 Estima versor v e um ponto P_b que passe pela reta r formada pelos dois pontos;
- 4 **para** cada ponto P **faz**
- 5 Calcula a distância d do ponto à reta r ;
- 6 **se** $d < d_{min}$ **então**
- 7 Guarda P como pertencente à r ;
- 8 **se** Número de pontos de $r > tam_{min}$ **então**
- 9 Guarda v , P_b e C ;
- 10 Ordena as retas do menor para o maior C ;
- 11 **para** cada reta r ordenada **faz**
- 12 **se** Número de pontos de $r > tam_{min}$ **então**
- 13 Guarda v , P_b e pontos $P \in r$;
- 14 **retorna** Retas r selecionadas na última etapa;

O algoritmo ?? possui uma eficiência muito alta, acertando quais os *clusters* que existem em um evento, porém ainda assim não é perfeito. Uma das falhas do algoritmo é que ele pode selecionar retas muito próximas e entender que não são um único *cluster*, e sim dois distintos. A figura ?? mostra um exemplo em que foram detectadas três retas, onde deveria existir apenas duas (é necessário combinar duas delas).

A etapa de correção da saída do *Enhanced Ransac* é chamada de clusterização. A ideia é comparar, dois a dois, todos os *clusters* resultantes da saída e usar algum critério para juntar ou não os dois clusters. O problema precisa abordado avaliando a semelhança entre dois *clusters*. Uma métrica possível é o coeficiente de silhueta [silhueta].

O coeficiente de silhueta compara dois *clusters* e retorna um valor entre -1 e 1 que informa se estão separados corretamente. O valor -1 informa que a separação dos *clusters* estão errados, 0 indica sobreposição e 1 significa que a separação está correta. A métrica se mostra muito viável para análise de duas dimensões, porém para o caso de três dimensões essa métrica não se mostrou muito eficiente. Mesmo colocando um *threshold* muito baixo o algoritmo indicava que deveria juntar *clusters* mesmo que muito distantes um do outro.

A abordagem correta se dá primeiro comparando o ângulo entre as duas retas. Caso a diferença absoluta entre os ângulos com relação ao versor $(0, 0, 1)$ seja menor que um valor dado, que nesse caso foi considerado 9° (determinado empiricamente), então as duas retas serão combinadas se obedecerem a condição dada pela equação ??.

$$\sum_{i=0}^{N_1} \frac{d_{i2}}{N_1} < \alpha d_{min}, \quad (5.2)$$

onde N_1 é o número de pontos da reta 1, d_{i2} a distância do ponto i da reta 1 em relação

à reta 2, α é um parâmetro com valor a ser escolhido e d_{min} é a distância mínima de ponto a reta usada no algoritmo ???. Os valores escolhidos foram tais que $\alpha = 1.75$ e $d_{min} = 15$ mm. A figura ?? mostra o antes e depois da clusterização ser aplicada em um evento.

Após a etapa de clusterização é necessário classificar cada reta como sendo ou o feixe, ou uma partícula espalhada. O feixe deve incidir no TPC com um ângulo muito pequeno com relação ao versor $(0, 0, 1)$. Além disso, mesmo se o ângulo for muito pequeno, a reta do feixe deve cruzar o plano da janela do TPC muito próximo do ponto mais provável da entrada o feixe. O ponto mais provável foi calculado usando a posição média da projeção dos pontos de uma *run* no plano *xy*. Disso obtemos que a posição inicial mais provável do feixe é tal que $x = -3.4$ (6.7) mm e $y = -0.9$ (6.3) mm. A incerteza é alta pois o feixe incide em muitas posições diferentes da janela.

Portanto se o ângulo entre o versor \hat{v}_i de uma reta i for menor que 5° (determinado de modo empírico novamente) e a distância d entre o ponto P_i que intercepta o plano e o ponto $(x, y, 0)$, dados no parágrafo anterior, for menor que 15 mm (pouco mais que duas vezes a incerteza de cada ponto), então a reta é considerada como o feixe do evento. No caso de não satisfazer essas condições, então ela classificada como uma provável partícula originada da reação do feixe com o gás.

Importante notar que, pelo baixo ganho da parte central do *micromegas*, há eventos que não possuem feixe, como mostrado na figura ???. Neste caso é necessário assumir as propriedades da reta mais provável para o feixe, ou seja, precisa passar pelo ponto $(x, y, 0)$ e ter versor $(0, 0, 1)$.

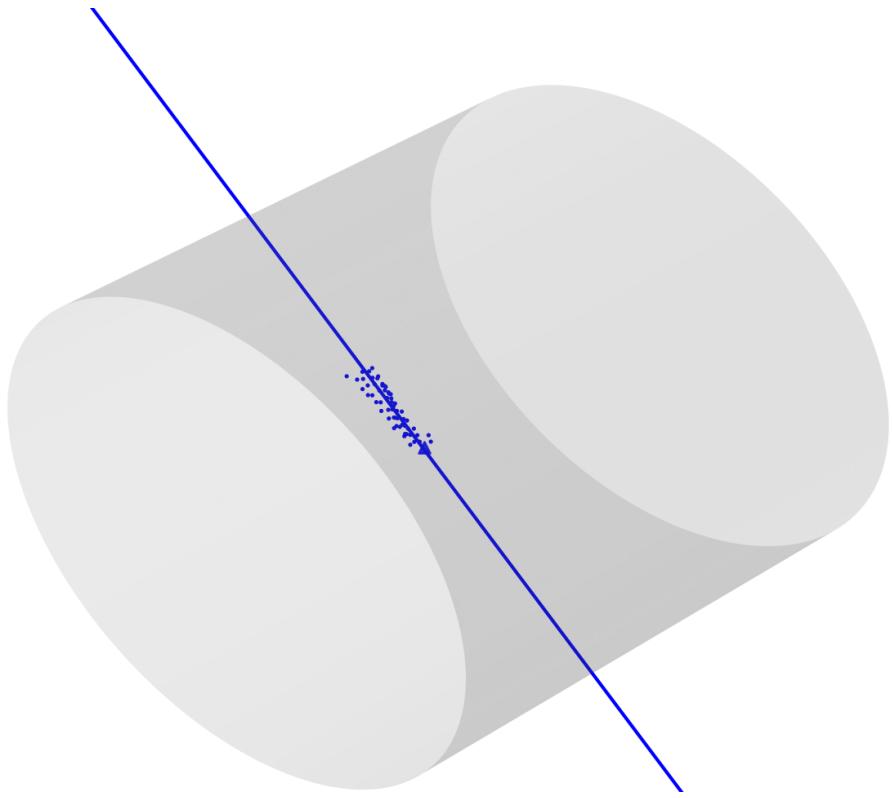


Figura 5.2: Evento em que não foi detectado o feixe, apenas a partícula espalhada. O triângulo azul é o local calculo do vértice de reação dado pela equação ??.

Nem todas as retas que não são o feixe são originadas da reação do feixe com o gás. Para determinar sua origem é necessário determinar o vértice de reação com o feixe. O vértice de reação é o ponto médio do segmento de reta que conecta a reta a ser analisada e o feixe no ponto de menor distância entre as retas. Ele permite ainda fazer correções futuras e diferenciar o começo e o fim de uma reta, pois é possível determinar o ponto tal onde ocorreu uma reação.

Temos as seguintes equações das retas \vec{P}_1 e \vec{P}_2 como vetores:

$$\begin{aligned}\vec{P}_1 &= \vec{A}_1 + \vec{V}_1 * t_1 \\ \vec{P}_2 &= \vec{A}_2 + \vec{V}_2 * t_2,\end{aligned}\tag{5.3}$$

onde \vec{A}_1 e \vec{A}_2 são pontos arbitrários que pertencem as retas 1 e 2, respectivamente, \vec{V}_1 e \vec{V}_2 são os versores, t_1 e t_2 são os hiperparâmetros das retas.

A reta que conecta a menor distância possui versor

$$\vec{V}_c = \frac{\vec{V}_1 \times \vec{V}_2}{|\vec{V}_1 \times \vec{V}_2|}.\tag{5.4}$$

Podemos então construir uma reta \vec{P}_3 que conecta \vec{P}_1 e \vec{P}_2 . Essa reta deve começar no ponto de menor distância da reta 1 e terminar no ponto de menor distância da reta 2. Ou seja, temos o seguinte sistema linear:

$$\vec{A}_2 + \vec{V}_2 * \tilde{t}_2 = \vec{A}_1 + \vec{V}_1 * \tilde{t}_1 + \vec{V}_c * \tilde{t}_3.$$

Rearranjando temos que

$$\vec{V}_1 * \tilde{t}_1 - \vec{V}_2 * \tilde{t}_2 + \vec{V}_c * \tilde{t}_3 = \vec{A}_2 - \vec{A}_1, \quad (5.5)$$

onde \tilde{t}_1 , \tilde{t}_2 e \tilde{t}_3 são os hiperparâmetros a serem determinados. Caso \vec{V}_1 seja paralelo à \vec{V}_2 , então não há solução (não há vértice de reação). Achando os valores, achamos os pontos de menor distância nas duas retas:

$$\begin{aligned} \vec{P}_1 &= \vec{A}_1 + \vec{V}_1 * \tilde{t}_1 \\ \vec{P}_2 &= \vec{A}_2 + \vec{V}_2 * \tilde{t}_2. \end{aligned} \quad (5.6)$$

Conseguimos então determinar que o vértice de reação \vec{V}_r é dado por

$$\vec{V}_r = \frac{1}{2}(\vec{P}_1 + \vec{P}_2). \quad (5.7)$$

Também podemos definir a distância de máxima aproximação d_{max} das retas, dada pela equação ??.

$$d_{max} = |\vec{P}_1 - \vec{P}_2|. \quad (5.8)$$

$$d_{max} = |\vec{P}_1 - \vec{P}_2|. \quad (5.9)$$

Da equação ?? podemos estabelecer um limite máximo de distância que uma reta pode ter do feixe, para então definir se houve realmente a reação no vértice de reação definido pelo equação ???. Retas que possuíam d_{max} maior ou igual que 25mm foram automaticamente descartadas.

Existem ainda situação em que a reta e o feixe tem uma distância de máxima aproximação muito pequena, porém o vértice de reação estava fora dos limites do TPC ($|x| < 140\text{mm}$, $|y| < 140\text{mm}$ e $|t| < 512$), o que também indica que a reta deve ser descartada.

Agora com apenas as retas certas selecionadas devemos apenas tomar cuidado de excluir eventos em que apenas o feixe foi detectado, pois neste caso não há o evento físico para ser analisado.

5.2 Métodos com *machine learning*

Na seção ?? descrevemos o algoritmo completo de seleção de eventos para a análise. Agora o objetivo é aplicar algoritmos de *machine learning* para melhorar o funcionamento de algo já existente ou resolver o mesmo problema de uma maneira totalmente diferente.

5.2.1 Clusterização hierárquica

Como queremos classificar pontos, dando *labels* que diferenciem os pontos de cada reta, podemos usar algoritmos de clusterização (*machine learning* não supervisionado). Devemos escolher algoritmos que sejam muito rápidos pois a ideia é substituir o uso do RANSAC. O algoritmo usado é o *Hierarchical Density-Based Spatial Clustering of Applications with Noise* (HDBSCAN)[[hdbSCAN1](#), [hdbSCAN2](#)], pois é muito eficiente em tempo e consegue realizar *clustering* mesmo em dados muito complexos. O algoritmo consegue também nos dar os *outliers* da *pointcloud*, porém para eliminação de *outliers* será usado novamente o *outlier removal* do Open3D. A figura ?? mostra resultados do uso do algoritmo, já com as retas ajustadas. Percebe-se casos em que a houve falha na clusterização.

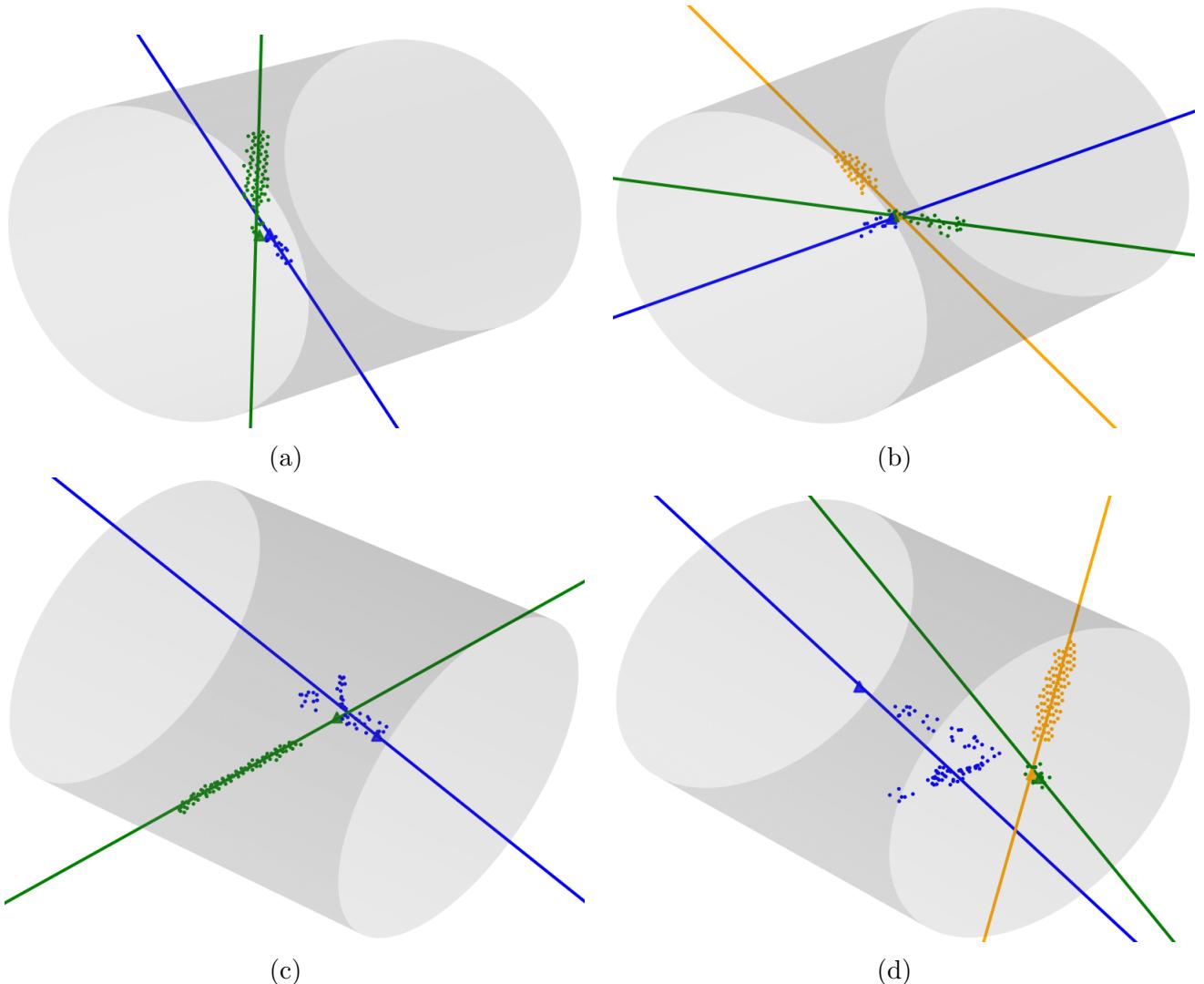


Figura 5.3: Exemplos dos resultados para o HDBSCAN. Percebe-se que em ?? e ?? o algoritmo falhou, juntando diferentes *clusters* ou simplesmente detectando ruído junto da *track*.

A eficiência em relação ao RANSAC é menor, pois os *clusters* são pouco densos, o que dificulta a seleção de *clusters* pelo algoritmo. Além disso, o algoritmo tem dificuldade em separar *clusters* que possuem pontos sobrepostos, ou seja, onde há vértice de reação (vide

figuras ?? e ??). A tabela ?? mostra a comparação entre a taxa de acerto e a eficiência em tempo entre o HDBSCAN e do RANSAC. A taxa de acertos mostra quantos foram corretamente solucionados (ou seja, todos os *clusters* foram detectados corretamente), e a eficiência mostra a capacidade de processamento de eventos pelo algoritmo, medida em eventos por segundo. O *benchmark* foi feito usando o processador Ryzen 5 3600X.

Table 5.1: Comparaçāo entre algoritmos usados para identificar tracks em eventos. O HDBSCAN acerta menos vezes em comparaçāo com o RANSAC (cerca de 14% menos), porrm é quase 4 vezes mais rápido.

Método	Taxa de acertos (%)	Eficiência (eventos/s)
RANSAC	78.2	57
HDBSCAN	64.3	208

A clusterização se mostrou melhor em eventos que tinham uma separação clara entre os *clusters*, sem pontos que coincidem duas *tracks* diferentes[TripClust], e também nos casos em que a densidade de pontos era muito significativo. Apesar da queda na taxa de acertos a velocidade de execução sobe significativamente, sendo uma possível escolha no lugar do RANSAC.

Capítulo 6

Resultados

Após identificar todos os *clusters* de cada evento devemos identificar quais são as partículas que originaram cada uma das trajetórias detectadas. Temos o comprimento de cada trajetória e sua energia, portanto o objetivo é, dada essas duas informações, identificar qual a partícula. A figura ?? mostra um histograma bidimensional do comprimento de cada *track* em função do ângulo de espalhamento no referencial do laboratório, usando apenas eventos que possuíam duas *tracks* com o mesmo vértice de reação. É possível perceber as medidas coincidentes do ^{16}O e do próton.

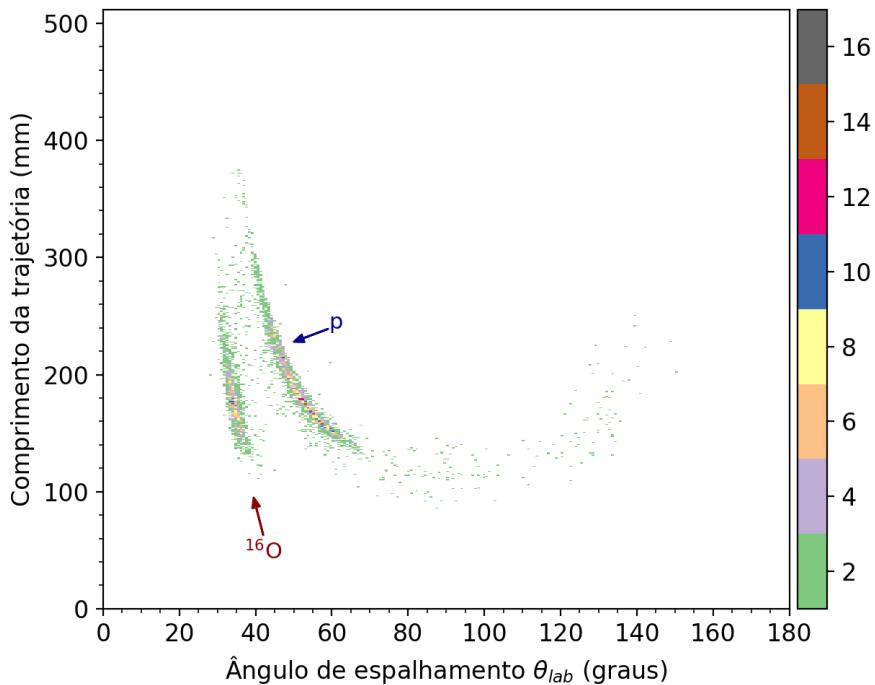


Figura 6.1: Histograma de comprimento de *track* no eixo y e ângulo de espalhamento no eixo x. O histograma foi feito coletando eventos que possuíam duas trajetórias com o mesmo vértice de reação, indicando a detecção simultânea do ^{16}O e do próton.

Para determinar qual é a partícula de cada *track* podemos usar o LISE++[lise++] para calcular o alcance (*range*) das possíveis partículas (^{17}F , ^{16}O e próton) dada as propriedades do alvo (^4He à uma pressão de 350 Torr). A figura ?? mostra o alcance em mm das partículas em

função da energia em MeV.

Capítulo 7

Conclusão