Informações

Matheus Martines de Azevedo da Silva, 9797145.
Relatório da Atividade 3.
Link para o repositório com códigos: mathmartines/IntroML_Atividade3.git

1 Particionamento dos Dados

Os dados foram divididos em 80% para treinamento e 20% para teste. Dos dados de treinamento, 20% foram separados como sendo um conjunto de validação, ou seja, utilizados para verificar a performance do modelo durante o treinamento e para ajustar os possíveis hiperparâmetros. O número total de amostras, bem como o número de amostras caracterizadas como sinal e fundo para cada um dos particionamentos, é mostrado na Tabela 1 abaixo. Essa divisão foi feita somente uma vez, e cada um dos conjuntos foi salvo em um arquivo diferente. Os respectivos arquivos para cada um dos conjuntos podem ser encontrados na pasta Data/. Os dados para esta atividade foram extraídos do artigo [1].

Conjunto de Dados	Amostras de sinal	Amostras de fundo	Número total de amostras
Treinamento	33790	30210	64000
Validação	8442	7558	16000
Teste	10602	9399	20001

Table 1: Número total de amostras nos conjuntos de treinamento, validação e teste, bem como o número de amostras caracterizadas como sinal e fundo em cada um dos conjuntos.

2 Modelos Utilizados

Nesta atividade, experimentamos três modelos diferentes. O primeiro é um simples Multi-Layer Perceptron, enquanto o segundo e o terceiro, denotados como Point-Net e Particle Cloud, são modelos invariantes à permutação das partículas em uma dada amostra (ou evento). Além desses três, também testamos um modelo que é uma combinação das arquiteturas abordadas nos modelos Point-Net e Particle Cloud. A descrição de cada um dos modelos é discutida nas seções a seguir.

2.1 Multi-Layer Perceptron

O primeiro modelo é um simples Multi-Layer Perceptron (MLP) tendo como parâmetros de entrada as 21 low-level features do conjunto de dados. Aqui vale a pena esclarecer como os dados foram preparados. Para cada evento no conjunto de dados, primeiro

temos as informações do momento do lepton, em seguida as informações sobre a missing energy e, por último, os momentos de cada um dos jatos presentes. Os jatos foram ordenados segundo o momento transverso (p_T) dos mesmos. Isto é, o jato 1 é o que possui o maior p_T, o jato 2 o segundo maior p_T e assim por diante. Note que, olhando somente para o conjunto de dados, não conseguimos perceber essa ordenação, pois cada uma das colunas do dataset foi escalonada. Contudo, pela Figura 2 de [1], conseguimos notar essa ordenação.

Como mencionado anteriormente, os dados disponíveis já foram escalonados para terem média igual a zero e desvio padrão igual a um, exceto os dados que representam grandezas estritamente positivas. Estas, por sua vez, foram escalonadas para terem o valor médio igual a um e não zero. Por conta dessa normalização, nenhuma outra foi aplicada no conjunto de dados.

O modelo que construímos possui três camadas, onde cada uma possui 32 neurônios. Cada camada é composta por uma transformação linear, seguida por uma Batch Normalization e por uma função de ativação ReLU (rectifier activation function). A Batch Normalization é responsável por normalizar cada uma das saídas da transformação linear de maneira a manter as médias próximas de zero e com desvio padrão igual a um. Em seguida, ela escala e soma o resultado por dois vetores que são ajustados durante o treinamento. Durante o treinamento, a média e o desvio padrão são calculados utilizando o batch atual, enquanto após o treinamento é utilizado uma média e desvio padrão móveis sobre todas as batches vistas durante o treinamento. Essa é uma estratégia adotada que reduz o risco de gradientes nulos e também acelera o treinamento. Mais informações sobre ela podem ser encontradas na documentação do Keras em keras.io/api/layers/normalization_layers/batch_normalization/.

Após o MLP, adicionamos mais uma camada (totalmente conectada com a última camada do MLP) que representa a camada de saída. Essa camada possui dois neurônios com a função de ativação Softmax. Dessa maneira, a saída do modelo representa a probabilidade da amostra ser um evento de sinal ou de fundo. A classe que representa o MLP pode ser encontrada no arquivo src/Models/MLP.py, e o modelo final com a camada de saída foi definido no Jupyter Notebook ML/SimpleMLP.ipynb.

No mesmo Jupyter Notebook, o treinamento desse modelo foi realizado com 100 epochs e com batches de tamanho igual a 32. Para não correr o risco de sobreajuste (overfitting), adicionamos uma condição de Early Stopping que, se após trinta epochs não houver melhora na função de perda calculada no conjunto de validação, o treinamento é interrompido. O Early Stopping também foi responsável por restaurar os pesos do modelo para a epoch onde a função de perda no conjunto de validação foi a menor. A função de perda escolhida foi a Entropia-Cruzada (Cross-Entropy).

2.2 Point-Net

Esse modelo é uma versão simplificada do modelo Particle Cloud que será discutido na seção seguinte, mas aqui já devemos mencionar que uma descrição detalhada e completa do último pode ser encontrada em [2].

O Point-Net e o Particle Cloud são modelos utilizados para a caracterização de jatos. Eles levam em consideração que um evento é composto por um conjunto de partículas, cada uma com suas devidas características (por exemplo, momento). A principal característica desses modelos é que eles são invariantes à permutação das partículas dentro de um evento. Ou seja, a maneira como ordenamos as partículas em cada um dos eventos não afeta a predição do modelo. Essa invariância é uma imposição que reflete a natureza dos dados observados em um colisor como o LHC, pois as partículas que qualificam um evento não possuem uma ordenação intrínseca.

Para aplicarmos esses modelos, precisamos primeiro ajustar o nosso conjunto de dados. No caso ideal, os modelos teriam um desempenho melhor se os dados não tivessem sido escalonados. Esse escalonamento não faz mais sentido nos modelos que abordaremos, pois ele depende da ordenação das partículas. Como não conseguimos reverter o escalonamento, vamos aplicar esses modelos no conjunto de dados da maneira que está e veremos que mesmo assim obtemos resultados interessantes.

Como no caso em que estamos trabalhando, nosso evento é composto por três objetos diferentes: lepton, missing energy e jatos, precisamos criar uma variável categórica para a identificação de cada um desses objetos. Para cada partícula, adicionamos três novas características: isLepton, isNeutrino e isJet. A categoria isLepton é igual a 1 se uma partícula é identificada como lepton e 0 caso contrário. As variáveis isNeutrino e isJet funcionam de forma análoga. Além disso, os jatos possuem a informação do b-tagging, que a princípio deveria ser igual a 1 caso o jato seja originado por um quark b e 0 caso contrário. Como os dados foram escalonados, essa identificação não é mais a mesma. Para mitigar os efeitos do escalonamento do b-tagging, valores de b-tagging maiores que 1 foram substituídos por 1, enquanto valores menores que 1 foram substituídos por zero (lembrando que essas colunas foram escalonadas para terem médias iguais a 1). Como cada uma das partículas precisa ter o mesmo número de características, os leptons e as missing energies também possuem variável categórica de b-tagging, porém com valores sempre iguais a zero.

Vale a pena notar que, se os dados não tivessem sido escalonados, também conseguiríamos estimar a pseudo-rapidez dos neutrinos, impondo que a massa invariante do par lepton-neutrino fosse próxima da massa do bóson W. Aqui definimos a pseudo-rapidez das missing energies como sendo zero. Em resumo, cada evento é composto por um conjunto de partículas, onde cada partícula possui as seguintes características:

1. momento transverso - p_T;
2. pseudo-rapidez - η;
3. ângulo azimutal - ϕ;
4. b-tag;
5. isLepton;
6. isNeutrino;

7. isJet.

A classe que transforma o conjunto de dados atual para o conjunto de dados descrito acima pode ser encontrada no arquivo src/PyTorch/HiggsDataset.py. Ela é uma subclasse da classe Dataset do PyTorch. Apesar de utilizarmos o PyTorch para prepararmos os dados (a razão para isso será explicada na seção seguinte), utilizamos o Keras como o pacote de análise.

Agora que discutimos como os dados foram preparados podemos explicar a arquitetura da rede. A principal característica é a presença de uma função h_Θ que é aplicada sobre todas as partículas de um evento. Essa função pode ser vista como um mapeamento do espaço de características da partícula para um novo espaço de características para a mesma partícula,

$$\boldsymbol{x}_i' = \boldsymbol{h_\Theta}(\boldsymbol{x}_i), \tag{2.1}$$

onde $\boldsymbol{x}_i$ denota o vetor de características da partícula e $\boldsymbol{x}_i'$ denota o novo vetor de características gerados por $\boldsymbol{h_\Theta} : \mathbb{R}^F \to \mathbb{R}^{F'}$. F e F' denotam a dimensão do espaço de features. A função $\boldsymbol{h_\Theta}$ é descrita por um conjunto de parâmetros Θ que são os mesmos para todas as partículas. Esses parâmetros são ajustados durante o treinamento da rede neural. As vantagens dessa abordagem são que essa função pode ser empilhada sucessivamente, onde cada camada terá um conjunto de parâmetros Θ diferente, e que pode ser descrita por um MLP.

Ao final de todas as funções $\boldsymbol{h_\Theta}$, adicionamos uma camada chamada GlobalAveragePooling, que é responsável por tirar uma média das características finais das partículas sobre todas as partículas do evento. Dessa maneira, cada evento terá o número de características igual ao número final de características das partículas. Após isso, podemos adicionar mais camadas conectadas até chegarmos à camada de saída.

Neste trabalho decidimos colocar somente uma camada da função $\boldsymbol{h_\Theta}$ que é descrita pelo mesmo MLP descrito na seção anterior, exceto que agora o número de inputs é diferente. Após GlobalAveragePooling, adicionamos mais duas camadas com 64 neurônios. Cada camada é composta por uma transformação linear, uma camada de Dropout com taxa de desligamento de neurônios igual a 0.1 e por uma função de ativação ReLU. Em seguida, colocamos mais uma camada totalmente conectada com a anterior com dois neurônios e com função de ativação Softmax para representar a camada de saída. O Dropout ajuda a evitar o sobreajuste durante o treinamento.

Para implementarmos a função $\boldsymbol{h_\Theta}$ criamos uma subclasse da classe Layer do pacote Keras que pode ser encontrada em src/Layers/PointNetLayer.py. A camada de Pooling também foi implementada dessa maneira e pode ser encontrada em src/Layers/Pooling.py. O modelo descrito acima foi definido na classe src/Models/PointNet.py e no Jupyter Notebook ML/PointNet.ipynb pode ser encontrado a instanciação e o treinamento do modelo. O treinamento foi feito de maneira idêntica ao treinamento discutido na seção anterior.

2.3 Particle Cloud

O Particle Cloud (ou Particle Net) [2] é um modelo utilizado para a identificação de jatos. Ele foi inspirado pela arquitetura de apresentada em [3]. A ideia desse modelo é similar a ideia apresentada na seção anterior, exceto que a função h_Θ além de levar em consideração a partícula também leva em consideração suas partículas vizinhas.

A arquitetura do Particle Cloud se baseia em uma camada de convolução chamada Edge Convolutional Layer. Para uma dada partícula em um evento, esta camada primeiro encontra as k partículas mais próximas partícula em questão. Para uma dada partícula representada pelo vetor de características $\boldsymbol{x}_i \in \mathbb{R}^F$, a operação de convolução tem a seguinte forma,

$$\boldsymbol{x}'_i = \frac{1}{k}\sum_{j=1}^{k} \boldsymbol{h_\Theta}(\boldsymbol{x}_i, \boldsymbol{x}_{i_j} - \boldsymbol{x}_i), \tag{2.2}$$

$\{\boldsymbol{x}_{i_1}, ..., \boldsymbol{x}_{i_k}\}$ denota o vetor de características de cada uma das partículas vizinhas. A função $\boldsymbol{h_\Theta} : \mathbb{R}^F \times \mathbb{R}^F \to \mathbb{R}^{F'}$ é chamada de *edge function*. Esta também pode ser representada por um MLP. Tal como no Point-Net, a Edge Convolutional Layer também pode ser empilhada, pois podemos entender cada uma das saídas como uma nova partícula representada por um novo vetor de características.

Após todas as camadas de convolução, aplicamos o mesmo GlobalAveragePooling e também podemos adicionar novas camadas como fizemos na seção anterior.

No trabalho em questão, consideramos somente uma camada de convolução e após o GlobalAveragePooling adicionamos outras duas camadas totalmente conectadas, seguindo as mesmas especificações das camadas da seção anterior (com 64 neurônios, seguido por Dropout e pela função de ativação ReLU). A *edge function* seguiu a mesma arquitetura do MLP descrito na seção 1.1, exceto que o número de inputs foi diferente. Para uma visualização da arquitetura, recomendamos olhar as Figuras 1 e 2 do artigo [2], mas note que a nossa arquitetura é mais simplificada. Para calcular as k partículas vizinhas mais próximas utilizamos as coordenadas (η, ϕ) de cada uma das partículas, e a distância entre elas é dada por

$$\Delta R = \sqrt{\Delta\phi^2 + \Delta\eta^2}, \tag{2.3}$$

onde $\Delta\phi$ e $\Delta\eta$ representam a diferença entre os ângulos azimutais e a pseudo-rapidez das partículas.

A camada de convolução foi implementada criando uma subclasse da classe Layers do Keras e pode ser encontrada em src/Layers/EdgeConvLayer.py. Essa implementação requer bastante entendimento de manipulação de tensores no TensorFlow. Por conta disso, implementamos uma versão simplificada da camada utilizando o pacote PyTorch que pode ser encontrado na pasta src/PyTorch. A implementação desta versão não depende de tensores, porém é muito mais lenda do que a anterior. De qualquer maneira, ela tem o propósito de deixar o algoritmo mais claro, mas o treinamento do modelo foi feito utilizando a versão do Keras. O modelo do Particle Cloud e sua instanciação podem ser encontrados em src/Models/ParticleCloud.py e ML/ParticleCloud.ipynb. O

treinamento do modelo foi realizado de forma identica a dos casos anteriores e pode ser encontrado no último link apresentado.

2.4 Point-Net com Particle Cloud

O último modelo que implementamos foi uma combinação do Point-Net com o Particle Cloud. Os inputs foram passados tanto pela PointNetLayer quanto pela Edge Convolutional Layer. Após o GlobalAveragePooling em cada uma das layers, as saídas foram encaminhadas para um MLP de duas camadas com 64 neurônios, onde cada camada consiste em uma transformação linear, um Dropout com taxa de desligamento igual a 0.1 e uma função de ativação ReLU. Note que nesse caso, o número de inputs do MLP é igual a soma das saídas dos GlobalAveragePoolings aplicados. Ao final, adicionamos uma camada com dois neurônios com a função de ativação Softmax. A arquitetura dessa rede pode ser encontrada na Figura 1.

Esse modelo foi definido no Jupyter Notebook ML/Combined.ipynb e o treinamento foi igual aos casos anteriores.

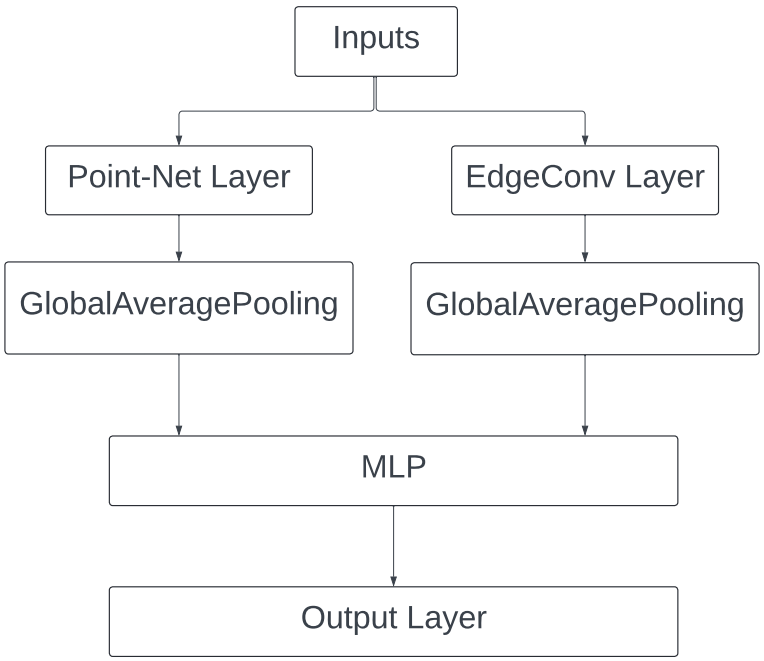

Figure 1: Arquitetura combinando as redes Point-Net e Particle Cloud.

3 Comparação

As performances de cada um desses modelos é comparada nesta seção. Excetuando as métricas de performance por epoch, que são calculadas no conjunto de treinamento e validação, todos os demais resultados foram obtidos no conjunto de teste.

Primeiro começamos apresentando a acurácia e a função de perda no conjunto de treinamento e validação para cada um dos modelos discutidos anteriormente. Os valores são apresentados na Figura 2.

Pela Figura 2, conseguimos perceber que, devido à condição de Early Stopping, nem todos os modelos chegaram a ser treinados com 100 epochs. O único modelo que chegou a ser treinado com 100 epochs e que não indicou tendência de sobreajuste foi o MLP. Todos os demais começaram a sobreajustar o conjunto de treinamento e, por isso, não chegaram a completar 100 epochs. Entretanto, vale a pena mencionar que, apesar de termos indicação de sobreajuste para os demais modelos, o Particle Cloud e a combinação do Particle Cloud com o Point-Net possuem uma acurácia e função de perda melhores que o MLP. Isso indica que, apesar desses modelos serem mais propensos ao sobreajuste se deixarmos o treinamento ocorrer por muitas epochs, eles conseguem generalizar melhor que o simples MLP. Isso também se deve ao fato de esses dois modelos terem mais parâmetros livres para serem ajustados do que o MLP, aumentando a capacidade destes para classificar os eventos com relação ao último. Pela Figura 2, também notamos que o modelo com o pior desempenho é o Point-Net, ficando atrás do MLP.

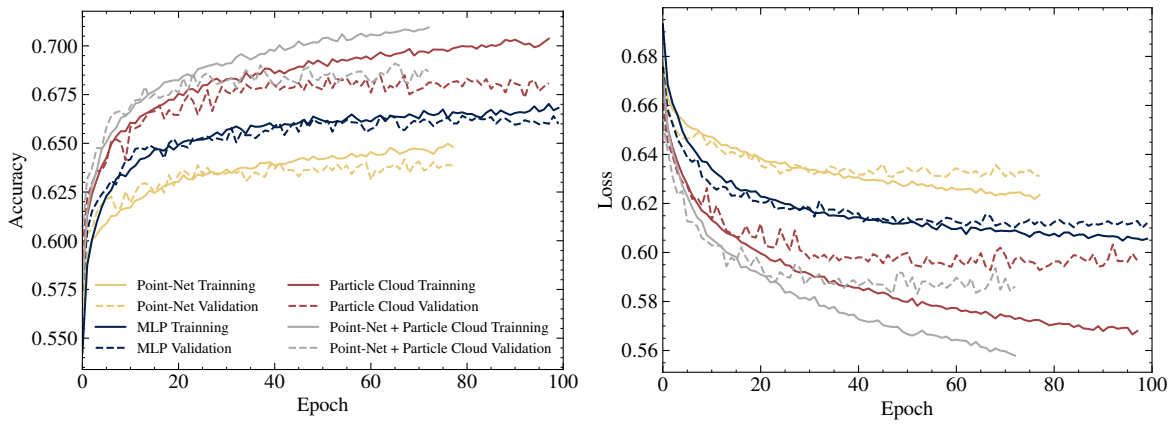

Figure 2: Acurácia e função de perda por epoch calculadas no conjunto de treinamento e validação para cada um dos modelos treinados. À esquerda apresentamos a acurácia, enquanto à direita a função de perda. O treinamento foi realizado com 100 epochs e com uma condição de Early Stopping. Por conta do último, alguns modelos não chegaram a serem treinados com 100 epochs.

Para compararmos um modelo contra o outro, calculamos a curva ROC e a área sob a curva (AUC), que são apresentadas na Figura 3. No caso em questão, a curva

ROC apresenta a signal efficiency ε_s (ou recall) pela background rejection, que é igual a 1 - background efficiency ε_b. A background efficiency representa a taxa de falsos positivos. Na nossa convenção, uma curva ROC mais próxima do canto superior direito indica um modelo melhor. Uma AUC próxima de 1 representa um modelo ideal que consegue distinguir perfeitamente entre sinal e fundo, enquanto que uma AUC igual a 0.5 representa um modelo que faz suas classificações de forma aleatória.

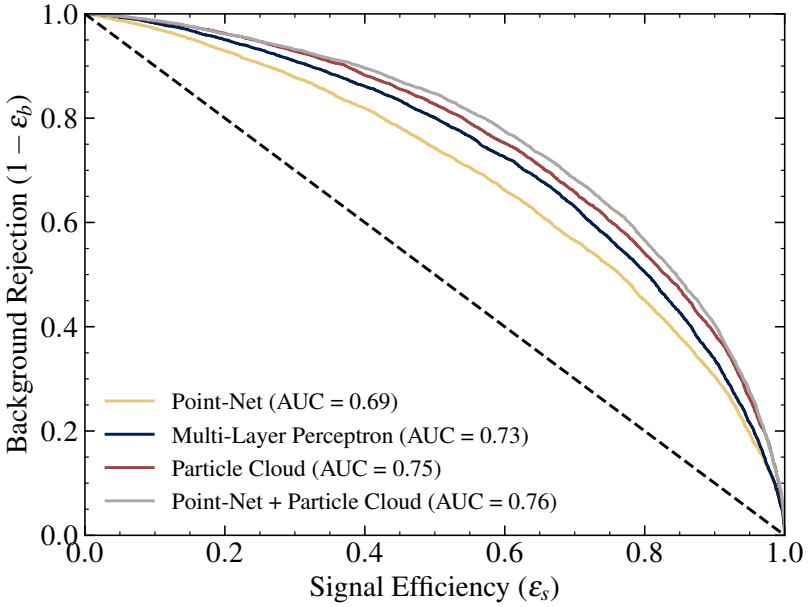

Figure 3: Curva ROC e área sobre a curva (AUC) para cada um dos modelos treinados.

Olhando para a Figura 3, conseguimos perceber que o modelo com melhor performance é a combinação entre o Particle Cloud e o Point-Net em conjunto com o Particle Cloud. Estes tendo um indice AUC igual a 0.76 e 0.75, respectivamente. Vale a pena chamar a atenção que o MLP tem um valor de AUC melhor que o Point-Net, apesar do último ter mais parâmetros que o primeiro.

Na Tabela 2 apresentamos o signal efficiency e a precisão de classificação do sinal com uma background rejection de 90%. Isso significa que colocamos um valor de corte na probabilidade para determinarmos se um evento é sinal ou não (lembrando que a saída dos modelos é a probabilidade de um evento ser sinal ou não) de forma que uma background rejection de 90% seja alcançada. Pela tabela conseguimos notar que o modelo que possui a melhor signal efficiecy dada um background rejection de 90% é a combinação do Particle Cloud com o Point-Net. O modelo que apresenta o pior desempenho é o Point-Net.

Por último, apresentamos os histogramas da massa invariante m_{WWbb} para o sinal e para o fundo obtidos por cada um dos modelos e também para os valores originalmente

Métrica	MLP	Point-Net	Particle Cloud	Point-Net com Particle Cloud
Signal Efficiency	0.32	0.26	0.37	0.39
Precisão	0.78	0.75	0.81	0.82

Table 2: Signall efficiency e precisão da classificação do sinal com uma background rejection de 90% para cada um dos modelos utilizados.

rótulos para comparação. Normalizamos os histogramas pelo número total de eventos em cada um deles. Os resultados podem ser encontrados na Figura 4.

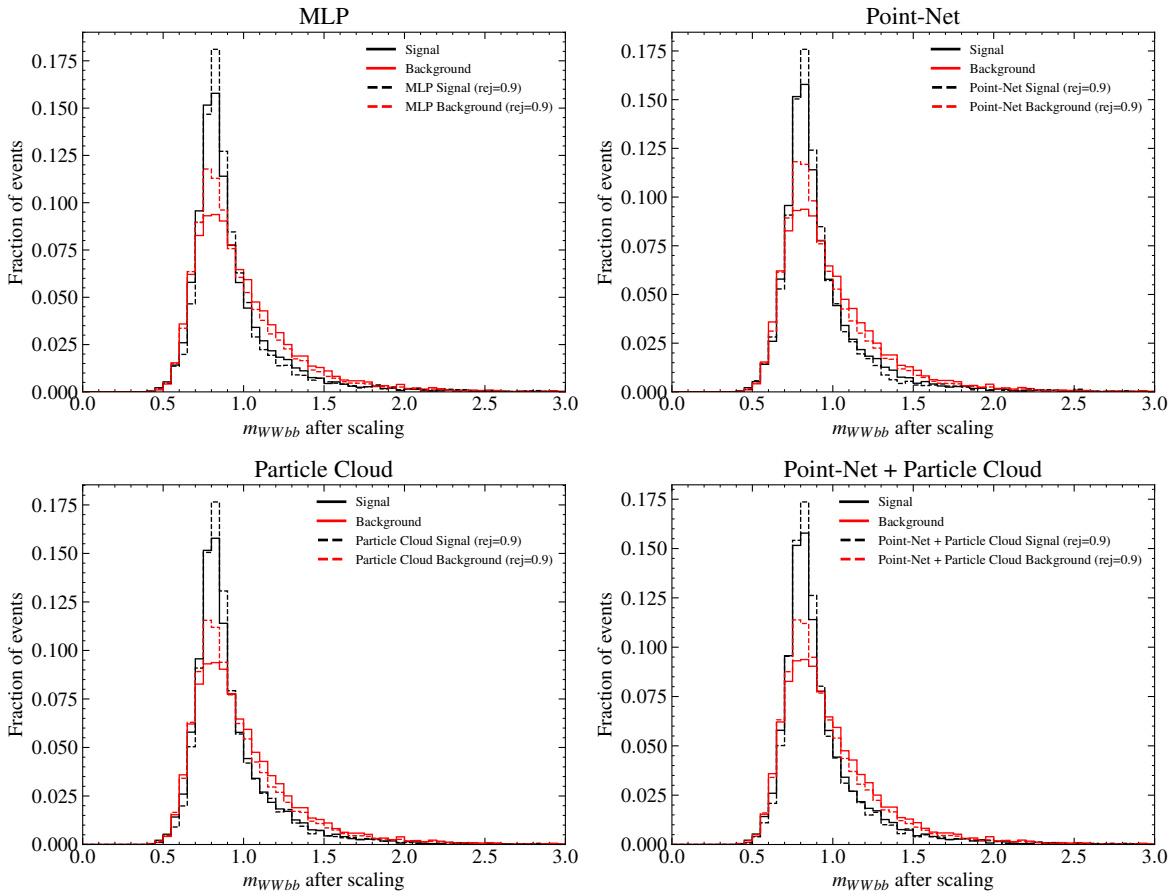

Figure 4: Histograma dos eventos pela massa invariante do sistema m_{WWbb} para o sinal e fundo caracterizados por cada um dos modelos, bem como os histogramas desse observável para os valores originalmente rotulados. As linhas tracejadas correspondem às predições das classificações dos modelos, enquanto as linhas sólidas correspondem às predições simuladas para o fundo e sinal. Os histogramas dos modelos foram gerados de maneira a alcançar uma background rejection de 90%.

4 Conclusões

Neste trabalho, utilizamos três modelos de Redes Neurais para caracterizar eventos de sinal e de fundo em um cenário além do Modelo Padrão. A primeira rede neural aplicada foi um MLP simples, onde as partículas seguem uma ordenação específica. Em particular, o modelo assume que os jatos foram ordenados através do p_T. Os demais modelos não exigem uma ordenação das partículas e são invariantes a permutações das mesmas.

Apesar das limitações impostas pela normalização dos dados, notamos que a rede com o melhor desempenho foi uma combinação do Particle Cloud e do Point-Net. A convolução dessas arquiteturas é interessante, pois a primeira leva em conta características locais do conjunto de partículas baseadas nas distâncias entre elas, enquanto a segunda considera somente características globais do conjunto. Quando as duas arquiteturas foram consideradas em conjunto, o desempenho melhorou em relação à arquitetura do Particle Cloud, conforme mostrado nas métricas de desempenho discutidas.

Por fim, é interessante notar a aplicabilidade dessas arquiteturas para construir observáveis para o sinal e para o fundo, como mostrado na Figura 4. Observamos que as distribuições de sinal e fundo obtidas pelo modelo são qualitativamente semelhantes às distribuições verdadeiras, demonstrando a aplicabilidade de redes neurais para identificação de eventos.

References

[1] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for Exotic Particles in High-Energy Physics with Deep Learning". In: *Nature Commun.* 5 (2014), p. 4308. DOI: 10.1038/ncomms5308. arXiv: 1402.4735 [hep-ph].

[2] Huilin Qu and Loukas Gouskos. "ParticleNet: Jet Tagging via Particle Clouds". In: *Phys. Rev. D* 101.5 (2020), p. 056019. DOI: 10.1103/PhysRevD.101.056019. arXiv: 1902.08570 [hep-ph].

[3] Yue Wang et al. "Dynamic graph cnn for learning on point clouds". In: *ACM Transactions on Graphics (tog)* 38.5 (2019), pp. 1–12.