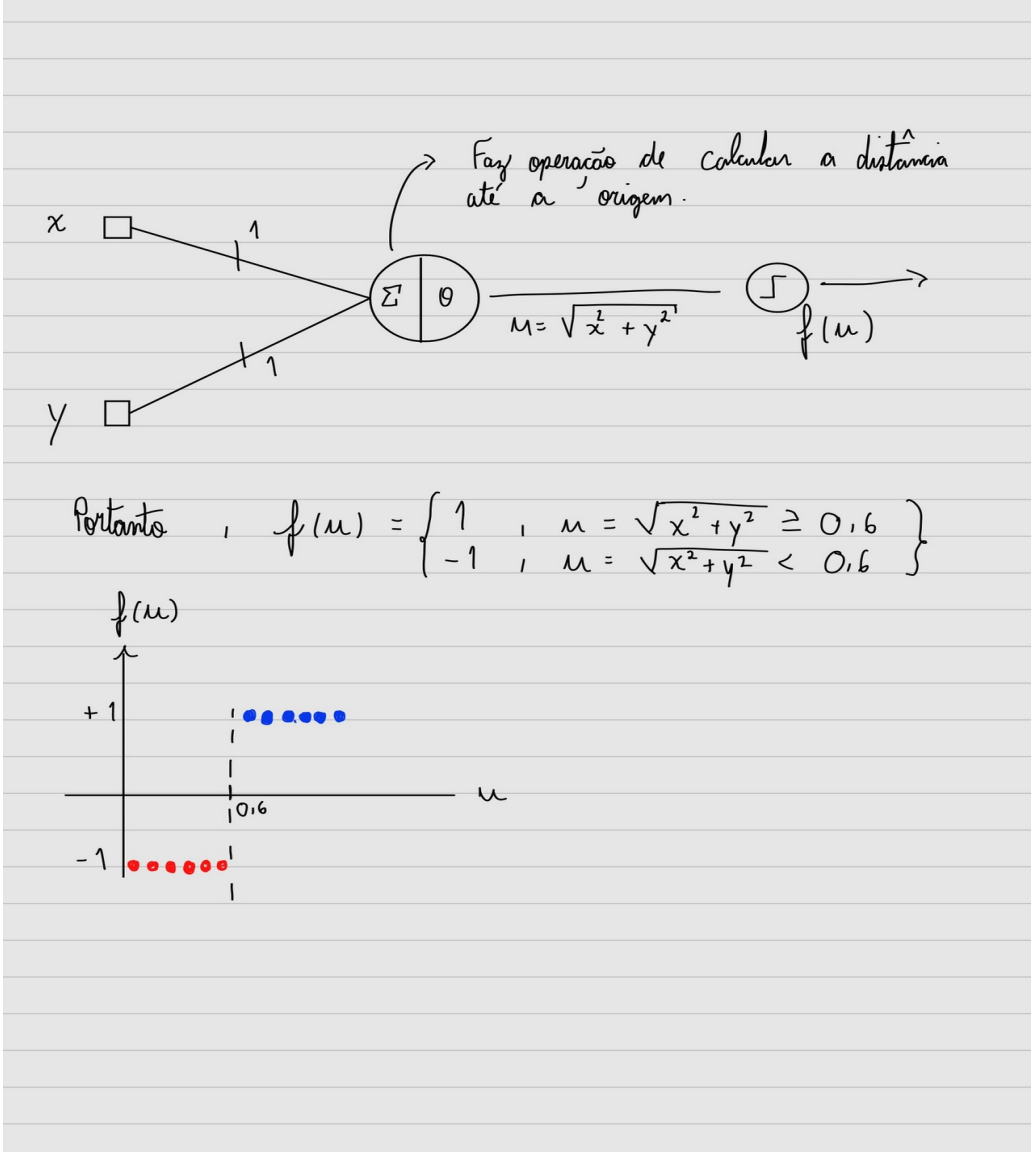
Exercício 2, Redes Neurais Artificiais.

Nome : Arthur Felipe Reis Souza.

PARTE 1

Na primeira parte do exercício 2, pede-se que seja implementada uma projeção não linear arbitrária que torne o problema, dos pontos interiores ao circulo de raio 0.6, linearmente separável. A idéia foi implementar um neurônio que pega 2 entradas, 1 ponto (x,y), e calcula distância do mesmo até a origem. Caso essa distância seja menor do que o valor 0.6, podemos afirmar que está dentro da região interna do círculo. Caso a distância seja maior que 0.6, podemos afirmar que o ponto está na região externa ao circulo. A função de ativação desse neurônio, será 1 função degrau com um threshold de valor 0.6, que irá separar a região em valores podendo ser 1 ou -1.



Abaixo, estará o código implementado em python com as respectivas imagens da linearização do problema proposto :

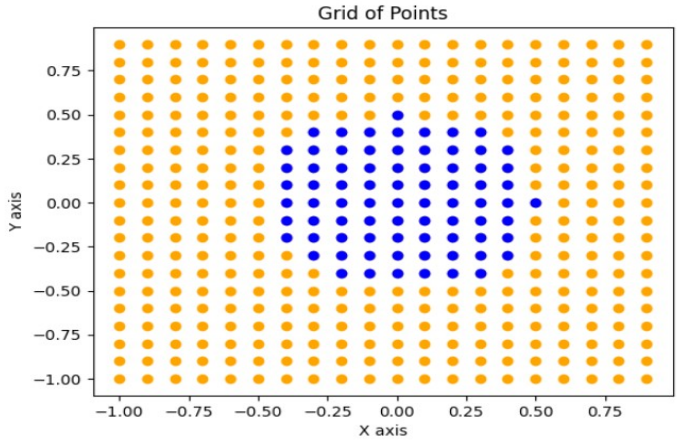
[68]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Criando o array espaçado que conterá 20 valores de x e 20 valores de y, e o raio do circulo será de 0.6.

[69]:
```
X = np.arange(start = -1, stop = 1, step = 0.1)
y = np.arange(start = -1, stop = 1, step = 0.1)
radius = 0.6
```

Plotando o gráfico grid, e destacando os valores que irão compor pontos no circulo.

[70]:
```
# Criando a grade de pontos do intervalo [-1, 1] com a função meshgrid.
xx, yy = np.meshgrid(X, y)
points = np.column_stack([xx.ravel(), yy.ravel()])

# O circulo consistirá de pontos onde, a distância até a origem seja valores menores que 0.6.
circle_points = points[distances <= radius - 0.1]

# Usando o matplot para plotar o gráfico desejado.
plt.scatter(points[:, 0], points[:, 1], color='orange')
plt.scatter(circle_points[:, 0], circle_points[:, 1], color='blue', label='Circle Points')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.title('Grid of Points')
```



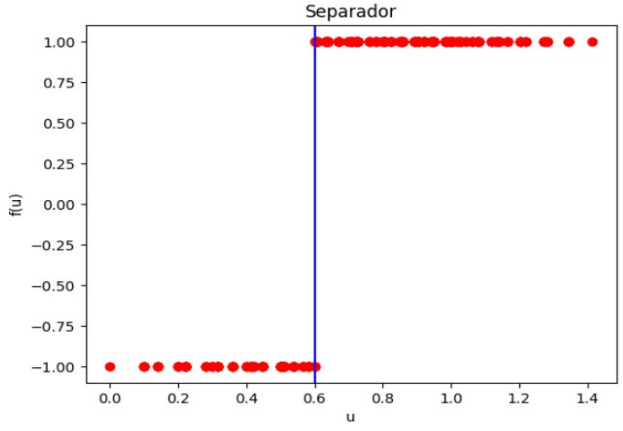
Criar 1 array, que conterá as distâncias dos pontos até o ponto de origem e uma função degrau com threshold de 0.6.

[74]:
```
# O array distances conterá a distância de todos os pontos da grade em relaçao a origem.
distances = np.linalg.norm(points, axis = 1)

# Retornará 1 para as posições que tiverem 1 distância maior do que 0.6, e 0 caso contrário.
def result(arr_ : np.ndarray):
    return (arr_ >= 0.6) * 1

# O array y terá valores que serão 1 para fora do circulo de raio 0.6 e -1 para valores fora.
y_points = result(distances)
y_points[y_points == 0] = -1
```

Criando o gráfico de como ficou a classificação ao final do processo.

[77]:
```
# Juntando os pontos x com os pontos y, no fito de criar 1 gráfico.
new_points = np.column_stack((distances, y_points))
plt.scatter(new_points[:, 0], new_points[:, 1], color = 'red')
plt.axvline(x = 0.6, color = 'blue')
plt.xlabel('u')
plt.ylabel('f(u)')
plt.title('Separador')
```

[77]:
```
Text(0.5, 1.0, 'Separador')
```



É possível observar, portanto, que a camada intermediária aplica uma não linearidade na entrada e, no final, passará por uma função degrau que ditará se o ponto está na região interna ou externa ao circulo.

PARTE 2:

A parte 2 consiste de algumas perguntas acerca da generalização de uma função que se ajustará aos dados que tem uma trajetória senoidal.

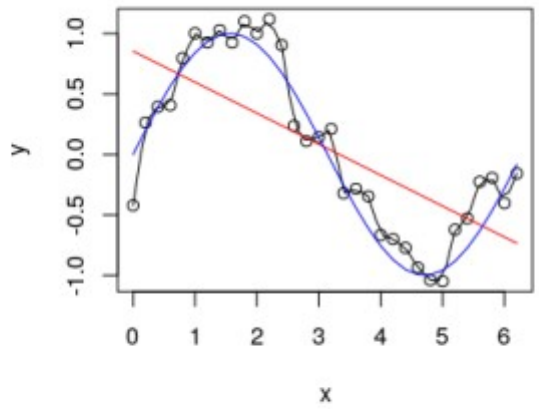


Figure 2: Ajuste de três modelos para um problema de regressão

1) - Sabendo que há um ruído na imagem, a função senoidal azul aparenta ser uma melhor aproximação para a função geradora de dados. A função preta está superajustada (overfitting) aos dados, de modo que ela não é útil para estipular uma boa saida para uma nova entrada de dados. A função vermelha é uma reta e, claramente, não é uma boa aproximação para a função geradora de dados.

2) – A função preta apresentará o menor erro de treinamento, ela passará corretamente em todos os pontos de dados, e isso não nos leva a uma boa solução. O modelo está superajustados aos dados, de modo que não nos dará uma boa resposta para uma nova entrada de dados que seja diferente da entrada atual.

3) - O modelo senoidal azul deverá ter um melhor desempenho na resposta para novos dados de entrada, aparenta ser uma boa generalização para os dados de entrada.

4) – O baixo erro não implica em melhor desempenho no longo prazo. O ideal é ter um erro baixo, de modo que o erro não seja tão baixo a ponto de o modelo se superajustar aos dados de entrada. Portanto, o principal é ter 1 modelo que terá 1 erro baixo, mas que também possa realizar previsões corretas acerca dos dados de entrada.

PARTE 3

A terceira parte do exercício 2 consiste em encontrar e estudar as aproximações polinomiais, as aproximações consistirão em neurônios na camada intermediária que poderão representar até um polinômio de grau 8. Com base nas aproximações, é necessário comentar se ocorreu overfitting ou underfitting.

O código abaixo mostra a criação das 10 amostras de dados, geradas seguindo uma função do 2 grau, com um ruído gaussiano de média 0 e desvio padrão 4.

Criando a função que irá gerar os dados seguindo a lógica de uma função do 2 grau.

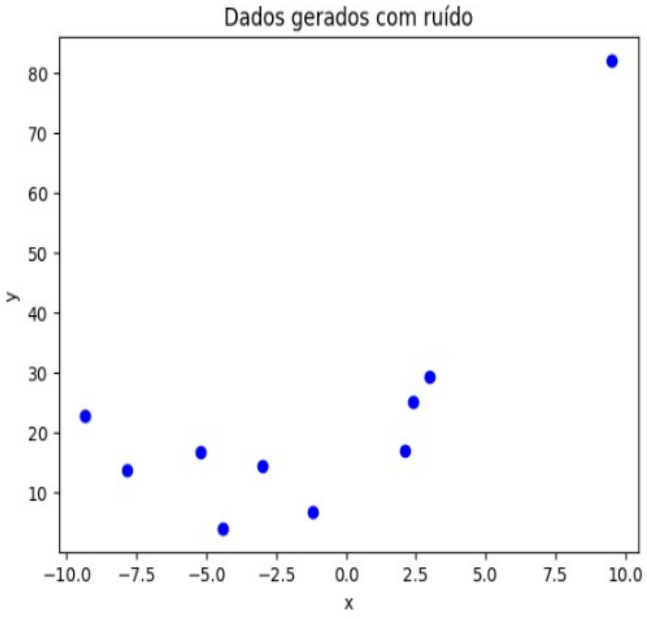
[3]:
```
# Criando a função que irá gerar os dados.
def func_generator(arr_ : np.ndarray):
    return (0.5 * (arr_ ** 2) + 3 * arr_ + 10)
```

Gerando a amostra de 10 dados com ruído.

[4]:
```
# Alterando a semente.
np.random.seed(123)
# Pegando 20 pontos aleatórios no intervalo [-15,20]
# Mudar para 100 pontos e analisar o resultado.
X = np.random.uniform(low = -15, high = 10, size = 10)

# A função tenderá a ser igual a função criada acima, porém terá 1 ruído gaussiano de desvio padrão 4.
Y = func_generator(X) + 4 * np.random.randn(len(X))
plt.scatter(X, Y, color = 'blue')
plt.xlabel("x")
plt.ylabel("y")
plt.title('Dados gerados com ruído')
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to write a readonly database')).History will not be written to the database.
```

[4]:
```
Text(0.5, 1.0, 'Dados gerados com ruído')
```



A aproximação inicial, será feita com 1 neurônio na camada intermediária. Esse neurônio irá manter o valor da entrada na saída, gerando portanto uma reta para aproximar os dados geradores.

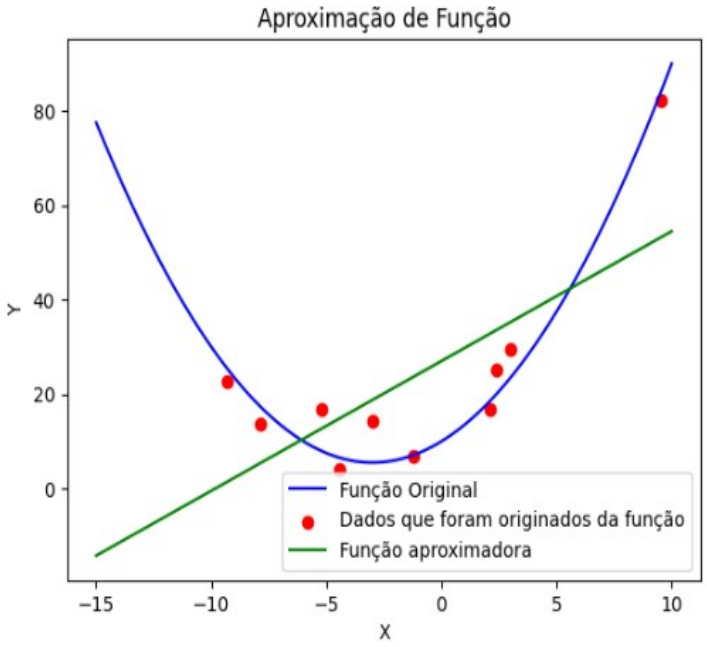
H é a saída da primeira camada da rede neural, e portando a entrada da última camada da rede. Basta lembrar que Hw = Y.

[8]:
```
# A função H é a entrada dos neurônios da camada de saída. Ela resulta da saída dos neuronios da primeira camada.
H = np.column_stack([X, np.ones_like(X)])
# A função w é a pseudoniversa da função H, multiplicada pelo resultado Y.
w = np.linalg.pinv(H) @ Y
```

Criando as coordenadas para plotar a função do 2 grau geradora, e a função que irá generalizar os dados de entrada.

[9]:
```
# Criando e plotando os gráficos contínuos para criar 1 representação gráfica.
xgrid = np.linspace(-15, 10, 1000) # Gera 1 distribuição de 1000 pontos que começa em -15 até o 10.
ygrid = func_generator(xgrid) # É a função do 2 grau, aplicada nos valores continuos xgrid.

Hgrid = np.column_stack([xgrid, np.ones_like(xgrid)]) # Criando o H para os valores contínuos para representar a função.
yhatgrid = Hgrid @ w # Calculando a função y que irá generalizar aos dados de entrada (y = Hw).
```

[10]:
```
plt.plot(xgrid, ygrid, color='blue', label='Função Original') # Plotando o gráfico contínuo da função do 2 grau.
plt.scatter(X, Y, color = 'red', label = 'Dados que foram originados da função') # Plotando os dados os quais a função irá generalizar.
plt.plot(xgrid, yhatgrid, color = 'green', label = 'Função aproximadora') # Plotando a função aproximadora
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Aproximação de Função')
plt.legend()
plt.show()
```



É possível afirmar que, há um underfitting, ou seja, a função aproximadora não generaliza bem aos dados geradores. Isso acontece porque há poucos dados para serem aproximados, e o modelo apenas mantém a linearidade na camada intermediária.

Após modificar os neurônios da camada intermediária, obtemos que a melhor aproximação se dará para um conjunto de neurônios de grau 2 na camada intermediária.

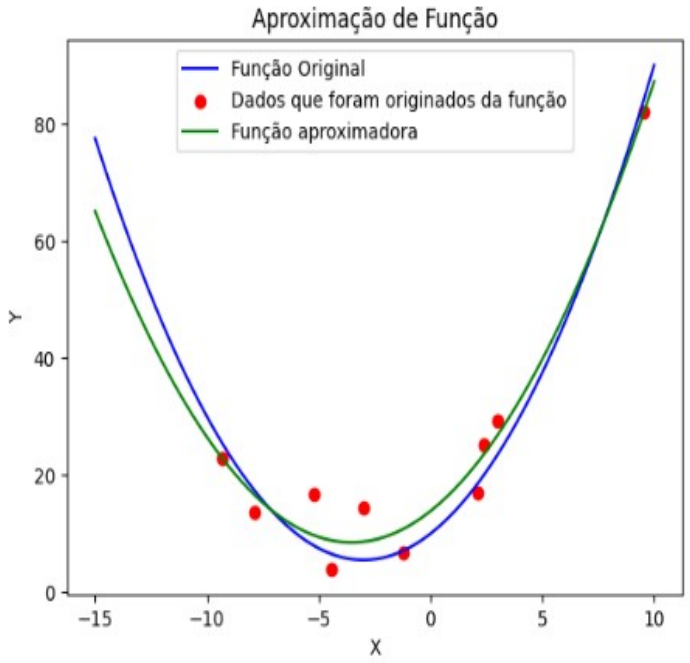
H é a saída da primeira camada da rede neural, e portando a entrada da última camada da rede. Basta lembrar que Hw = Y.

[11]:
```
# A função H é a entrada dos neurônios da camada de saída. Ela resulta da saída dos neuronios da primeira camada.
H = np.column_stack([X**2, X, np.ones_like(X)])
# A função w é a pseudoniversa da função H, multiplicada pelo resultado Y.
w = np.linalg.pinv(H) @ Y
```

Criando as coordenadas para plotar a função do 2 grau geradora, e a função que irá generalizar os dados de entrada.

[12]:
```
# Criando e plotando os gráficos contínuos para criar 1 representação gráfica.
xgrid = np.linspace(-15, 10, 1000) # Gera 1 distribuição de 1000 pontos que começa em -15 até o 10.
ygrid = func_generator(xgrid) # É a função do 2 grau, aplicada nos valores continuos xgrid.

Hgrid = np.column_stack([xgrid**2, xgrid, np.ones_like(xgrid)]) # Criando o H para os valores contínuos para representar a função.
yhatgrid = Hgrid @ w # Calculando a função y que irá generalizar aos dados de entrada (y = Hw).
```

[13]:
```
plt.plot(xgrid, ygrid, color='blue', label='Função Original') # Plotando o gráfico contínuo da função do 2 grau.
plt.scatter(X, Y, color = 'red', label = 'Dados que foram originados da função') # Plotando os dados os quais a função irá generalizar.
plt.plot(xgrid, yhatgrid, color = 'green', label = 'Função aproximadora') # Plotando a função aproximadora
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Aproximação de Função')
plt.legend()
plt.show()
```



Observando a imagem, é possível afirmar que uma camada intermediária composta por 2 neurônios e um polinomio de grau 2 é uma boa aproximação para a função geradora de dados. A diferença entre ambas se dá pelo conjunto com apenas 10 amostras e um ruído gaussiano nessas amostras, esse ruído afetará a precisão do modelo e a aproximação final do mesmo.

Por fim, no fito de mostrar 1 superajuste (overfitting), utilizamos uma camada intermediária composta por neurônios que vão até um grau máximo de valor 6.

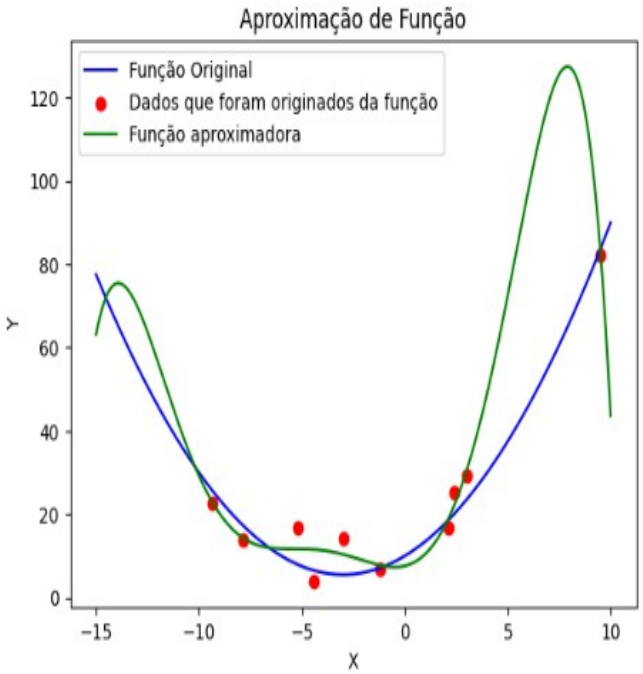
H é a saída da primeira camada da rede neural, e portando a entrada da última camada da rede. Basta lembrar que Hw = Y.

[35]:
```
# A função H é a entrada dos neurônios da camada de saída. Ela resulta da saída dos neuronios da primeira camada.
H = np.column_stack([X**6, X**5, X**4, X**3, X**2, X, np.ones_like(X)])
# A função w é a pseudoniversa da função H, multiplicada pelo resultado Y.
w = np.linalg.pinv(H) @ Y
```

Criando as coordenadas para plotar a função do 2 grau geradora, e a função que irá generalizar os dados de entrada.

[36]:
```
# Criando e plotando os gráficos contínuos para criar 1 representação gráfica.
xgrid = np.linspace(-15, 10, 1000) # Gera 1 distribuição de 1000 pontos que começa em -15 até o 10.
ygrid = func_generator(xgrid) # É a função do 2 grau, aplicada nos valores continuos xgrid.

Hgrid = np.column_stack([xgrid**6, xgrid**5, xgrid**4, xgrid**3, xgrid**2, xgrid, np.ones_like(xgrid)]) # Criando o H para os valores contí
yhatgrid = Hgrid @ w # Calculando a função y que irá generalizar aos dados de entrada (y = Hw).
```

[37]:
```
plt.plot(xgrid, ygrid, color='blue', label='Função Original') # Plotando o gráfico contínuo da função do 2 grau.
plt.scatter(X, Y, color = 'red', label = 'Dados que foram originados da função') # Plotando os dados os quais a função irá generalizar.
plt.plot(xgrid, yhatgrid, color = 'green', label = 'Função aproximadora') # Plotando a função aproximadora
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Aproximação de Função')
plt.legend()
plt.show()
```



É possível observar que, a função verde, está tendendo a se superajustar aos dados, de modo que ela não terá um bom desempenho para novas entradas de dados. Ao aumentar o grau dos poliômios da camada intermediária, irá apenas destacar esse superajuste do modelo aos dados.

Agora, iremos utilizar uma amostra de 100 dados e aproximar essa amostra de dados com 1 camada intermediária polinomial. A imagem abaixo mostra a geração e representação dos 100 dados utilizados.

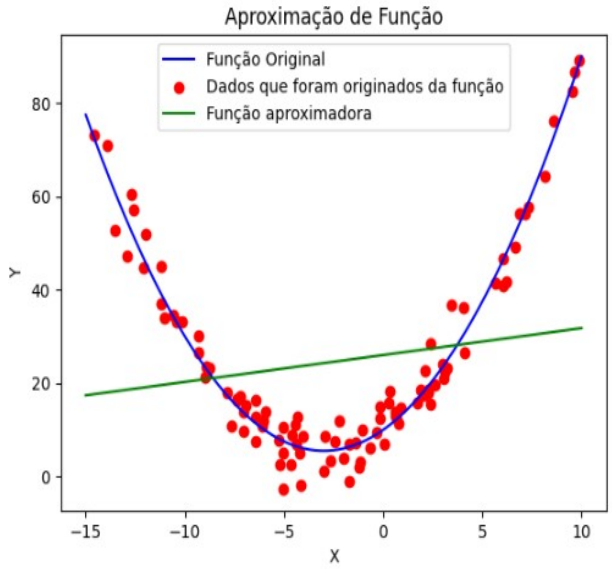
```
[38]: Text(0.5, 1.0, 'Dados gerados com ruído')
```

Utilizando 1 único neurônio, de grau 1, na camada intermediária, podemos afirmar que ele manterá a função linear na saída. Portanto, é possível observar 1 modelo pouco preciso, aproximando os dados através de 1 reta.

Criando as coordenadas para plotar a função do 2 grau geradora, e a função que irá generalizar os dados de entrada.

```
[45]: # Criando e plotando os gráficos contínuos para criar 1 representação gráfica.
xgrid = np.linspace(-15, 10, 1000) # Gera 1 distribuição de 1000 pontos que começa em -15 até o 10.
ygrid = func_generator(xgrid) # É a função do 2 grau, aplicada nos valores continuos xgrid.

Hgrid = np.column_stack([xgrid, np.ones_like(xgrid)]) # Criando o H para os valores contínuos para representar a função.
yhatgrid = Hgrid @ w # Calculando a função y que irá generalizar aos dados de entrada (y = Hw).
```

```
[46]: plt.plot(xgrid, ygrid, color='blue', label='Função Original') # Plotando o gráfico contínuo da função do 2 grau.
plt.scatter(X, Y, color = 'red', label = 'Dados que foram originados da função') # Plotando os dados os quais a função irá generalizar.
plt.plot(xgrid, yhatgrid, color = 'green', label = 'Função aproximadora') # Plotando a função aproximadora
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Aproximação de Função')
plt.legend()
plt.show()
```



Usando o polinômio quadrático como grau máximo para a camada intermediária, podemos observar 1 ótima aproximação para a função geradora.

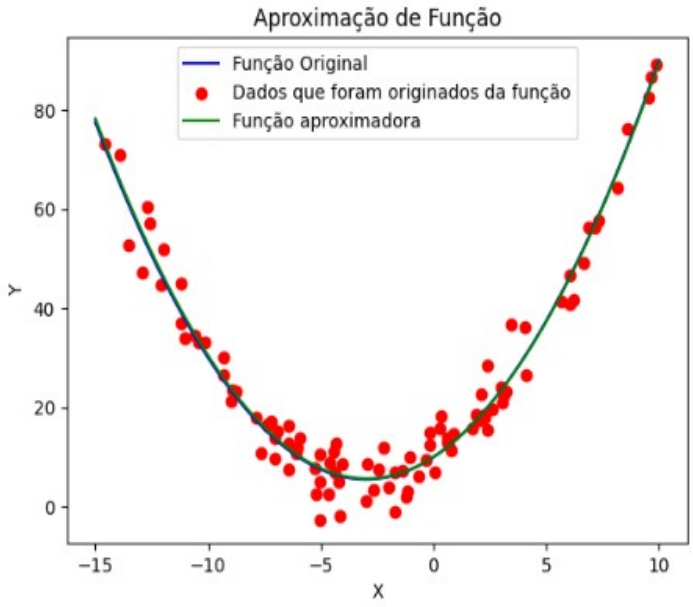
H é a saída da primeira camada da rede neural, e portando a entrada da última camada da rede. Basta lembrar que Hw = Y.

[53]:
```
# A função H é a entrada dos neurônios da camada de saída. Ela resulta da saída dos neuronios da primeira camada.
H = np.column_stack([X**2, X, np.ones_like(X)])
# A função w é a pseudoniversa da função H, multiplicada pelo resultado Y.
w = np.linalg.pinv(H) @ Y
```

Criando as coordenadas para plotar a função do 2 grau geradora, e a função que irá generalizar os dados de entrada.

[55]:
```
# Criando e plotando os gráficos contínuos para criar 1 representação gráfica.
xgrid = np.linspace(-15, 10, 1000) # Gera 1 distribuição de 1000 pontos que começa em -15 até o 10.
ygrid = func_generator(xgrid) # É a função do 2 grau, aplicada nos valores continuos xgrid.

Hgrid = np.column_stack([xgrid**2, xgrid, np.ones_like(xgrid)]) # Criando o H para os valores contínuos para representar a função.
yhatgrid = Hgrid @ w # Calculando a função y que irá generalizar aos dados de entrada (y = Hw).
```

[56]:
```
plt.plot(xgrid, ygrid, color='blue', label='Função Original') # Plotando o gráfico contínuo da função do 2 grau.
plt.scatter(X, Y, color = 'red', label = 'Dados que foram originados da função') # Plotando os dados os quais a função irá generalizar.
plt.plot(xgrid, yhatgrid, color = 'green', label = 'Função aproximadora') # Plotando a função aproximadora
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Aproximação de Função')
plt.legend()
plt.show()
```



No caso acima, temos uma amostra maior de dados. Essa maior amostra facilita uma boa generalização da função geradora do 2 grau, e protege de overfitting. Isso acontece porque, mesmo com o ruído gaussiano, o conjunto de amostras. Um maior conjunto de amostras influência diretamente no resultado do vetor de parâmetros w, que será obtido com o auxílio da pseudoinversa de H.
A representação a seguir mostra o processo do cálculo da saída ŷ da rede.

$$Hw = y \quad \Rightarrow \quad w = H^{+} y \quad \Rightarrow \quad Hw = \hat{y}$$

Não foi possível gerar 1 overfitting para a amostra de 100 dados. Isso acontece porque, com o grande número de amostra de entrada, o modelo tende a não ter 1 superajuste. O tamanho do conjunto de entrada influência diretamente na performance do modelo.