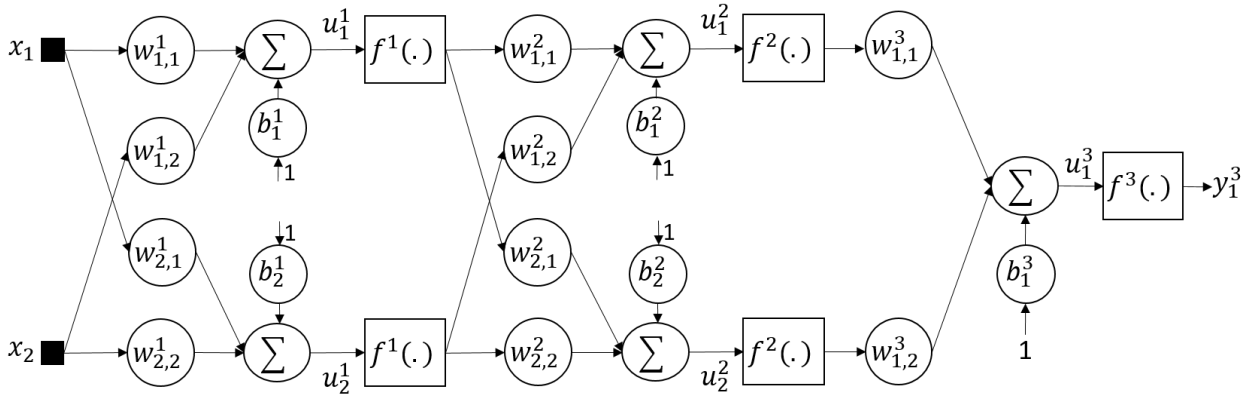
TP555 - AI/ML

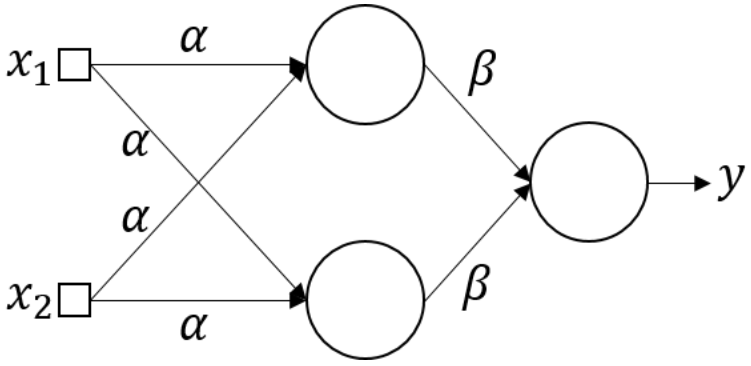
Lista de Exercícios #12

Redes Neurais Artificiais (Parte 2)

1. Por que a função de ativação logística foi um ingrediente chave no treinamento das primeiras redes MLPs?
2. Utilizando o exemplo activation_functions.ipynb como base, cite três funções de ativação diferentes das que vimos. Plote a função e sua respectiva derivada.
3. Desenhe uma rede MLP que calcule A ⊕ B, onde ⊕ representa a operação lógica XOR. (**Dica**: A ⊕ B = (A & !B) | (!A & B))
4. Suponha que você tenha uma rede MLP composta por uma camada de entrada com 10 neurônios, seguida por uma camada oculta com 50 neurônios e, finalmente, uma camada de saída com 3 neurônios. Todos os neurônios usam a função de ativação ReLU. Sendo assim, responda
    a. Qual é a dimensão da matriz de entrada X?
    b. Qual a dimensão do vetor de pesos, W_h, da camada oculta e de seu vetor de bias b_h?
    c. Qual é a dimensão do vetor de pesos, W_o, da camada de saída e de seu vetor de bias b_o?
    d. Qual é a dimensão da matriz de saída, Y**,** da rede?
    e. Escreva a equação que calcula a matriz de saída da rede, Y, como uma função de X, W_h, b_h, W_o e b_o.
5. Quantos neurônios você precisa na camada de saída de uma rede MLP se você deseja classificar emails em spam ou ham? Qual função de ativação você deve usar na camada de saída? Se você deseja classificar a base de dados MNIST (imagens de dígitos escritos à mão), quantos neurônios você precisa na camada de saída usando qual tipo de função de ativação?
6. Liste todos os hiperparâmetros que você pode ajustar em uma rede MLP? Caso você perceba que a rede MLP está ***sobreajustando***, como você pode modificar esses hiperparâmetros para tentar resolver o problema?
7. Dada a rede MLP mostrada na figura abaixo, encontre a derivada parcial do erro de saída, $e = d - y_1^3$, em relação aos dois pesos abaixo. (**Dica**: Siga o equacionamento apresentado no material da aula.)
    a. $w_{1,1}^1$, ou seja, $\frac{\partial e}{\partial w_{1,1}^1}$
    b. $w_{2,2}^1$, ou seja, $\frac{\partial e}{\partial w_{2,2}^1}$



8. O que aconteceria com os valores dos gradientes e pesos quando se inicializa **todos** os pesos de uma rede neural com o mesmo valor constante? Para verificar o que aconteceria, considere uma rede neural com dois atributos de entrada, (x_1, x_2), e com dois nós ocultos (ou seja, uma camada escondida com 2 nós) com função de ativação sigmóide e assuma que inicializamos todos os pesos de bias com o valor 0 e os pesos com alguma constante ***α***. Esta rede possui um único nó na camada de saída com função de ativação sigmóide e pesos de bias inicializados com o valor 0 e os pesos inicializados com alguma constante ***β***. A rede neural é apresentada na figura abaixo. Agora, aplique uma entrada (x_1, x_2)e calcule o erro de saída, $e = d - y$, onde d é o valor esperado. Na sequência, use o algoritmo da retropropagação do erro para calcular os gradientes de todos os pesos da rede neural.



   a. O que pode ser dito com relação à influência da saída dos nós ocultos no erro de saída?
   b. O que ocorre com os gradientes dos nós da camada oculta (Dica: compare os gradientes de ambos os nós da camada oculta.)?
   c. O que pode ser feito para resolver este comportamento que você observou quando os pesos são inicializados com o mesmo valor constante?

9. Baseando-se no exemplo MLPWithTensorFlowLowLevelAPI.ipynb, utilize objetos da classe ***Dense()*** do módulo ***tf.keras.layers*** ao invés da função ***neuron_layer()***. Após treinar o modelo, você percebeu alguma diferença na performance do seu modelo? Se sim, qual foi esta diferença?

(**Dica**: A documentação da classe dense pode ser encontrada no seguinte link: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense)

(**Dica**: Lembre que ***Dense*** é uma classe que precisa ser instanciada. Apenas após a criação do objeto da classe ***Dense*** é que se deve passar a entrada para esta camada de nós.)

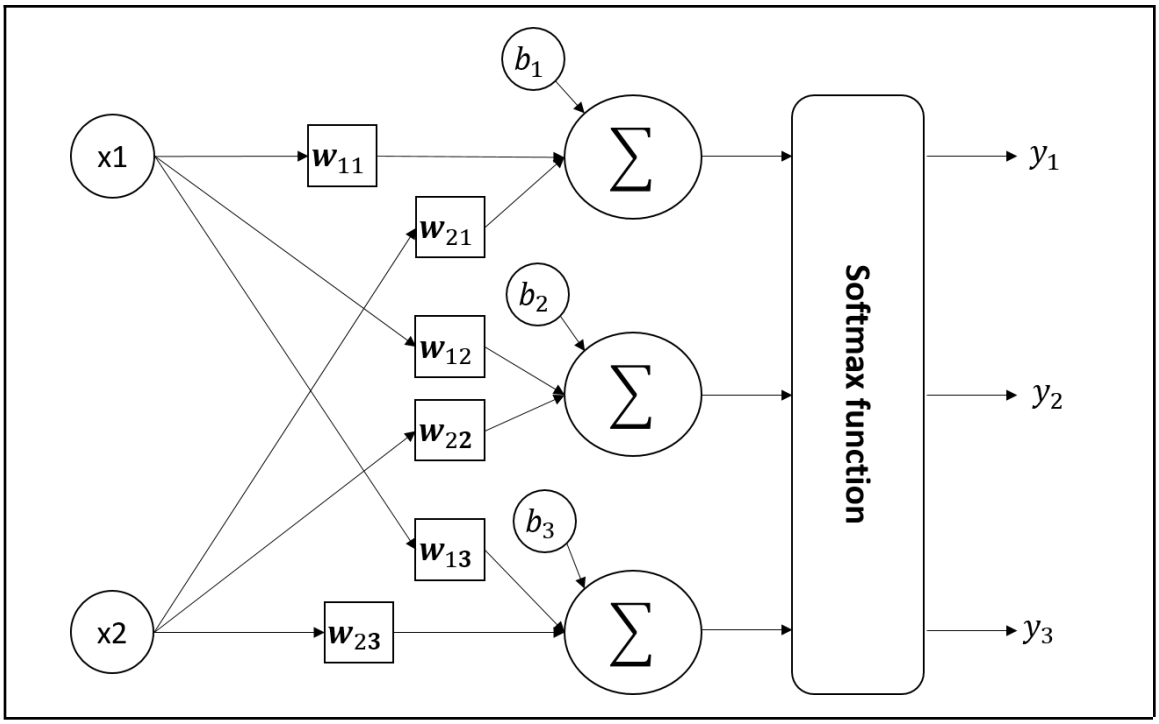
10. Baseando-se no código do exercício anterior, implemente a técnica do ***early stop*** de forma que o modelo sempre armazene os pesos que resultem no menor erro. Além disso, crie um contador que conte o número de épocas sem progresso, ou seja, o número de vezes em que o erro da época não diminuiu com relação ao menor erro obtido até o momento. Com base nesse contador, faça com que o treinamento se encerre caso o contador atinja 50 épocas sem progresso.

    (**Dica**: Lembre-se que uma maneira de se evitar que um modelo ***sobreajuste*** é interromper o treinamento assim que o erro de validação/teste atinja o mínimo. Outra maneira válida é armazenar os pesos do modelo que resultem no menor erro de validação durante o treinamento através de um pré-determinado número de épocas. Essa abordagem é chamada de ***early stop***.)

11. Baseando-se no código do exercício anterior, utilize o otimizador que implementa o algoritmo momentum ao invés do gradiente descendente e treine uma rede MLP que obtenha uma precisão maior do que 98%. Use o trecho de código abaixo.

```
optimizer = tf.train.MomentumOptimizer(learning_rate=0.01, momentum=0.9)
```

12. Utilizando o TensorFlow, implemente o modelo mostrado na figura abaixo, para classificar 3 classes diferentes.



Se nós modelarmos o classificador da figura em formato matricial, então, nós temos a seguinte equação:

$$\mathbf{y} = f(\mathbf{X}*\mathbf{W} + \mathbf{b}),$$

onde:

- X é a matriz de entrada com dimensão Nx2.
- W é a matriz de coeficientes com dimensão 2x3.
- b é o vetor de bias para a primeira camada, com dimensão 3.
- y é o vetor de saída do classificador.
- f(.) é a função softmax.
- N é o número de exemplos.

Crie a base de dados das 3 classes com o código abaixo.

```
from sklearn.datasets import make_blobs

# Create a 3-class dataset for classification
N = 1000
centers = [[-5, 0], [0, 1.5], [5, -1]]
X_, y_ = make_blobs(n_samples=N, centers=centers, random_state=42)
```

Em seguida, faça o seguinte:

a) Plote um gráfico mostrando as diferentes classes. (**Dica**: use cores ou marcadores diferentes para cada classe.)
b) Crie um grafo utilizando o **GradientDescentOptimizer** para classificar os dados. (**Dica**: verifique a documentação deste otimizador no seguinte link: https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/GradientDescentOptimizer)
c) Imprima e salve o erro e a precisão a cada 100 épocas.
d) Treine o modelo.(**Dica**: Use 20000 épocas ou treine até que a precisão seja de 100%.)
e) Plote um gráfico mostrando o erro e a precisão versus o número de épocas.
f) Plote um gráfico com as fronteiras de decisão.
g) Crie um segundo grafo utilizando o **AdamOptimizer** para classificar os dados. (**Dica**: Não se esqueça de resetar o grafo com ***tf.reset_default_graph()***.) (**Dica**: verifique a documentação deste otimizador no seguinte link: https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/AdamOptimizer)
h) Imprima e salve o erro e a precisão a cada 100 épocas..
i) Treine o modelo.(**Dica**: Use 20000 épocas ou treine até que a precisão seja de 100%.)
j) Plote um gráfico mostrando o erro e a precisão versus o número de épocas.
k) Plote um gráfico com as fronteiras de decisão.
l) Baseado nos valores de erro e precisão impressos e plotados nas figuras anteriores, qual dos 2 otimizadores tem melhor performance? (**Dica**: Qual dos 2 converge mais rapidamente?)

OBSERVAÇÃO: Neste exercício, você vai precisar utilizar a função **tf.nn.sparse_softmax_cross_entropy_with_logits()** para calcular o erro do modelo e assim, conseguir treiná-lo, ou seja, encontrar os pesos que minimizem o erro. A função **tf.nn.sparse_softmax_cross_entropy_with_logits()** é equivalente a aplicar a função

de ativação softmax e, em seguida, calcular a entropia cruzada, mas é mais eficiente e cuida adequadamente de casos incomuns, por exemplo, onde os logits são iguais a 0. A documentação dessa função pode ser encontrada em: https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits

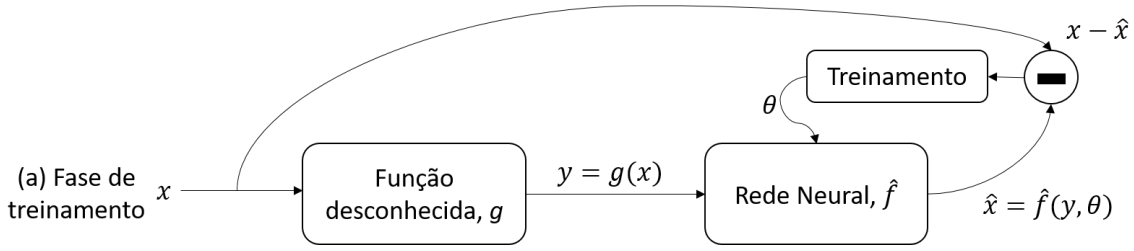
Seguem alguns exemplos de como utilizar a função:

[1] https://towardsdatascience.com/active-learning-on-mnist-saving-on-labeling-f3971994c7ba

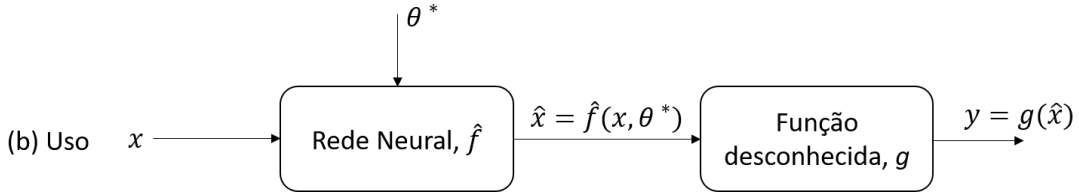
[2] https://riptutorial.com/tensorflow/example/17628/computing-costs-on-a-softmax-output-layer

13. Neste exercício, iremos inverter uma função não-linear e usar a função invertida para linearizá-la. Em sistemas de telecomunicações, distorção não-linear pode ocorrer entre transmissor e receptor, por exemplo, devido a amplificadores não-lineares no transmissor. Esta distorção pode ser representada por uma função desconhecida *g* que recebe um sinal *x* como entrada e produz uma saída distorcida *y* = *g*(*x*). A maneira convencional de desfazer a distorção é identificar um modelo parametrizado apropriado, então estimar seus parâmetros através de medições e, finalmente, criar uma função inversa com base nas estimativas. Essa abordagem está sujeita à propagação de erros entre as três etapas. Uma abordagem alternativa é treinar um modelo de rede neural para inverter diretamente a função, sem exigir modelagem explícita ou estimativa de parâmetros. O aprendizado de máquina pode fornecer melhores resultados do que a abordagem convencional para a linearização de funções.

O procedimento geral para treinar uma rede neural para inversão de função é ilustrado na figura abaixo.



Uma função desconhecida *g* com entrada *x* é invertida usando um modelo $\hat{f}$ treinando-o para obter $\hat{f}\left(g(x), \theta^{*}\right) \approx x$, conforme mostrado em (a). O procedimento de treinamento atualizará iterativamente os pesos θ para reduzir gradualmente os erros de aproximação até que convirja para um valor ótimo, θ^{*}.



O modelo treinado em (b) pode ser usado para linearizar a função desconhecida, *g,* sem ter que modelá-la explicitamente e estimar os parâmetros do modelo. Em (b), o que acontece é uma pré-distorção do sinal de entrada *x* o qual quando passado através da função desconhecida *g*, produz em sua saída um sinal linear.

Para realizar o treinamento, precisamos gerar um número N de valores x_n^{train} e passá-los através da função desconhecida, *g*, para medir

$$y_n^{train} = g\left(x_n^{train}\right), \text{ para } n = 1,\ldots, N.$$

Então, y_n^{train} é usado como entrada para o modelo de aprendizado de máquina, enquanto x_n^{train} é a saída desejada.

Exercício: dada a seguinte função *y* = *g*(*x*):

$$y = \frac{\alpha x}{\left(1+\left(\frac{|x|}{v_{sat}}\right)^{2p}\right)^{\frac{1}{2p}}},$$

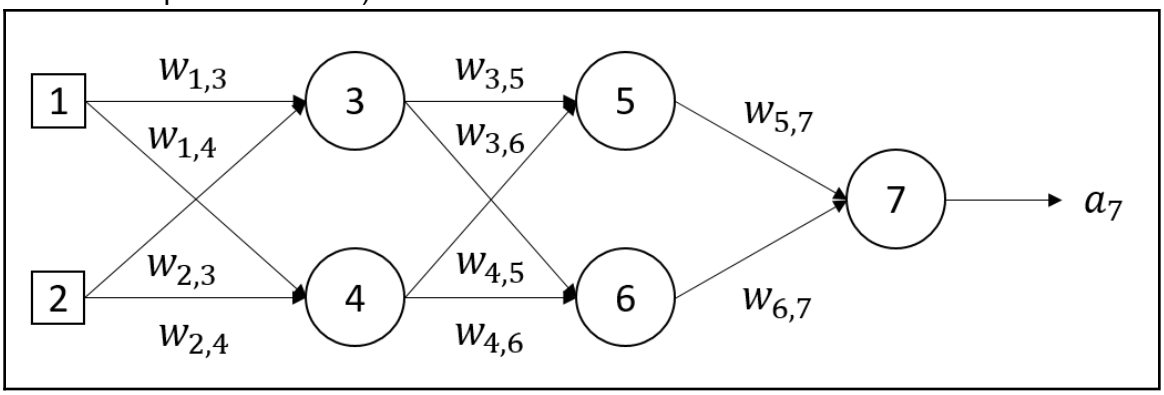
onde $\alpha = 4, p = 2$ e $v_{sat} = 1$. Use uma rede *multilayer perceptron* (MLP) para inverter a função *g*. Para isso, faça o seguinte

1. Plote o gráfico de x versus y.
2. Treine uma rede neural que inverta a função *g*.
   (**Dica 1**: use a classe **MLPRegressor** e **Grid** ou **Random Search** para encontrar o número ótimo de camadas e nós em cada camada.)
   (**Dica 2**: Comece com apenas uma camada escondida e vá aumentando gradativamente caso seja necessário para se inverter a função dada adequadamente.)
3. Após o treinamento, apresente:
   a. Um gráfico comparando o sinal x_n^{train} com o sinal x_n^{pred} (**Dica**: use o próprio sinal de treinamento para obter a predição).
4. O modelo treinado consegue inverter, ou seja, linearizar, a função desconhecida, *g*?
5. Como mostrado em (b), use o modelo treinado para pré-distorcer o sinal de entrada e, em seguida, aplique a saída do modelo à entrada da função *g*.
   a. Plote um gráfico comparando
      i. O sinal de entrada *x* e a saída da função *g*, ou seja, o sinal de saída original e distorcido.
      ii. O sinal de entrada *x* e o sinal pré-distorcido $\widehat{x}$.
      iii. O sinal pré-distorcido $\widehat{x}$ e a saída da função *g* quando sua entrada é $\widehat{x}$.

iv. O sinal de entrada *x* e o sinal de saída do sistema de pré-distorção, *y*, ou seja, a saída da função *g* quando sua entrada é $\hat{x}$.

b. O que ocorre com o sinal de saída, *y*?

14. Dada a rede MLP da figura abaixo, encontre
    a. A derivada parcial do erro de saída, $Loss_7 = \left(y_7 - a_7\right)^2$, em relação ao peso $w_{1,3}$, i.e., $\frac{\partial Loss_7}{\partial w_{1,3}}$.
    b. Uma forma geral para a atualização dos pesos da primeira camada escondida, ou seja, para os pesos $w_{1,3}$, $w_{1,4}$, $w_{2,3}$ e $w_{2,4}$.
    (**Dica**: Siga a análise feita no documento: "Aprendizado em Redes Neurais com Múltiplas Camadas").



15. Neste exercício, você irá investigar o uso de MLPs empregando funções de ativação sigmóide para obter mapeamentos um-para-um, conforme descrito na sequência:
    a. $f(x) = 1/x,\ 1 \leq x \leq 100.$
    b. $f(x) = log_{10}(x),\ 1 \leq x \leq 10.$
    c. $f(x) = exp(-x),\ 1 \leq x \leq 10.$
    d. $f(x) = sin(x),\ 0 \leq x \leq \pi/2.$

    Para cada mapeamento, faça o seguinte:
    A. Divida os dados em dois conjuntos, um para treinamento e outro para teste.
    B. Use o conjunto de treinamento para treinar a rede, **com uma única camada escondida**.
    C. Avalie a precisão do cálculo da rede usando os dados de teste.
    D. Apresente figuras comparando a predição feita com cada uma das MLPs treinadas com os dados originais.
    E. O que você pode concluir após realizar estes vários mapeamentos com uma MLP com uma camada escondida?

    Use uma única camada escondida, mas com um número variável de nós escondidos. Investigue como o desempenho da rede é afetado pela variação do tamanho da camada escondida.

(**Dica**: Use grid search com alguns valores para encontrar o número ótimo de nós na camada escondida)

16. Exercício sobre Autoencoders para aprendizado de sistemas de comunicação fim-a-fim. Neste exercício, usaremos um tipo de rede neural conhecida como autoencoder. Autoencoders são modelos de aprendizado não-supervisionado. Eles são um tipo de rede neural que aprende a copiar sua entrada para sua saída. Seu objetivo é aprender uma representação para um conjunto de dados, para redução ou aumento de dimensionalidade. Essas redes neurais são compostas por 2 partes: (i) **codificador**, que encontra representações eficientes para as entradas com menor ou maior dimensionalidade e (ii) **decodificador** que reconstrói a entrada.

    Após esta breve introdução sobre o que são autoencoders, vamos utilizá-los para implementar um sistema de comunicações fim-a-fim. Este sistema, do lado do transmissor, irá receber como entrada uma mensagem binária composta por k bits, irá convertê-las na representação one-hot encoding (ou seja, teremos M = 2^k possíveis mensagens de k bits), encontrar uma representação robusta de n símbolos reais da mensagem de entrada. Esta representação robusta com n símbolos trafega por um canal ruidoso e, do lado do receptor, ela é decodificada de tal forma que a mensagem original possa ser recuperada com a menor probabilidade de erro possível.

    O transmissor é composto por duas camadas densas (tf.keras.layers.Dense) com M e n nós respectivamente e recebe como entrada as representações one-hot encoding (tf.one_hot) das mensagens de k bits. A primeira camada utiliza ativação do tipo *relu* e a segunda não utiliza nenhum tipo de ativação. Finalmente, para evitar que o transmissor aprenda valores de saída desnecessariamente grandes e se torne numericamente instável, normalizamos a potência média de todas as saídas do transmissor no mini-batch para um valor igual a 1, conforme mostrado abaixo.

```
x = tx/tf.sqrt(tf.reduce_mean(tf.square(tx)))
```

(**OBS**.: Devemos criar uma *placeholder* (tf.placeholder) para que possamos alterar o tamanho dos *mini-batches* em tempo de execução.)

    O canal é implementado como um canal de ruído gaussiano branco aditivo (AWGN) o qual adiciona ruído gaussiano com média igual a zero e potência de ruído igual a σ^2 aos símbolos transmitidos, conforme mostrado abaixo.

```
noise = tf.random.normal(shape=tf.shape(x), stddev=noise_std)

y = x + noise
```

(**OBS**.: Devemos criar uma *placeholder* (tf.placeholder) para que possamos alterar o desvio padrão do ruído em tempo de execução.)

    O receptor é composto por duas camadas densas (tf.keras.layers.Dense) ambas com M nós. A primeira camada utiliza ativação do tipo *relu* e a segunda utiliza ativação do tipo *softmax*. A saída do receptor é um vetor de probabilidades que atribui uma

probabilidade a cada uma das mensagens possíveis. Finalmente, a mensagem original é recuperada selecionando a saída com maior probabilidade.

O autoencoder resultante pode ser treinado fim-a-fim usando o gradiente descendente estocástico (SGD).

(**OBS**.: Devemos criar uma *placeholder* (tf.placeholder) para que possamos alterar o passo de aprendizagem em tempo de execução.)

Dicas de implementação:

- O mini-batch de mensagens que queremos transmitir são retirados de uma distribuição aleatória uniforme, conforme mostrado abaixo.

```
s = tf.random.uniform(shape=[batch_size],minval=0,maxval=M,dtype=tf.int32)
```

- E para alimentá-los com eficiência à primeira camada densa do transmissor, nós as transformamos nos chamados vetores (ou tensor) one-hot. Este vetor agora contém batch_size vetores de comprimento M, onde apenas um elemento é definido como 1.0, enquanto todos os outros elementos são 0.0. Este tipo de vetor é comumente usado para tarefas de classificação.

```
s_one_hot = tf.one_hot(s,depth=M)
```

- Precisamos definir também uma função de custo que calcule o desempenho atual do modelo comparando a entrada com a saída. Usamos uma função de custo de entropia cruzada padrão que inerentemente ativa os logits com “softmax” e aceita rótulos esparsos, conforme mostrado abaixo.

```
cross_entropy = tf.losses.sparse_softmax_cross_entropy(labels=s,logits=s_hat)
```

- Para calcularmos a taxa média de erro de mensagem (ou de bloco) do mini-batch usamos decisão rígida com base no elemento com a maior probabilidade (argmax), conforme mostrado no trecho de código abaixo.

```
correct_predictions = tf.equal(tf.argmax(tf.nn.softmax(s_hat), axis=1, output_type=tf.int32), s)
accuracy = tf.reduce_mean(tf.cast(correct_predictions, dtype=tf.float32))
bler = 1.0 - accuracy
```

- Também precisamos definir um algoritmo otimizador que atualize os pesos de nosso autoencoder de acordo com a perda atual e o gradiente do mini-lote. Escolhemos o otimizador Adam para minimizar nossa função de custo e usar um *placeholder* para o passo de aprendizagem para poder ajustar este hiperparâmetro durante o treinamento.

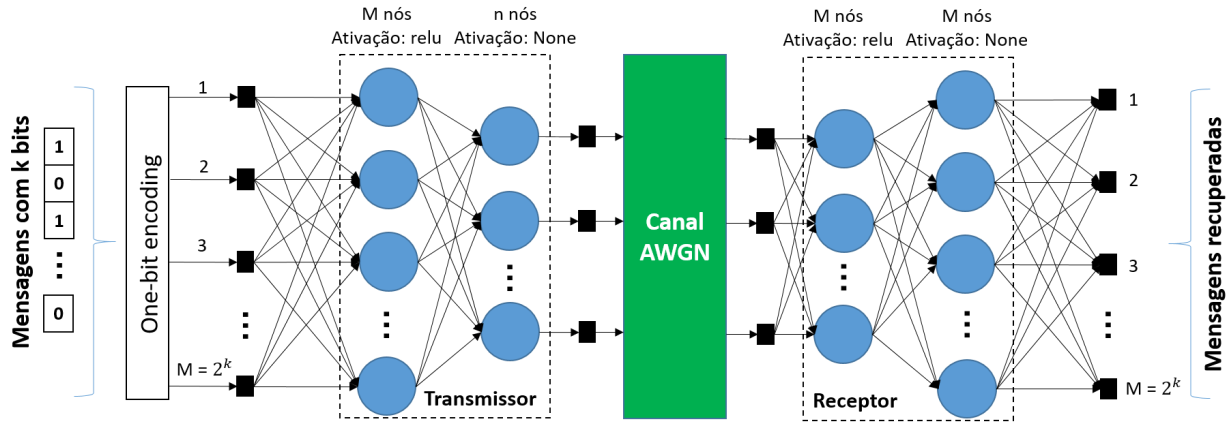
```
lr = tf.placeholder(dtype=tf.float32,shape=[])
train_op = tf.train.AdamOptimizer(learning_rate=lr).minimize(cross_entropy)
```

- Antes de iniciarmos o treinamento, precisamos definir a relação sinal-ruído (SNR), para que possamos treinar o modelo para um valor de SNR desejado. Esta função simplesmente calcula o desvio padrão do ruído para um determinado valor de SNR (enquanto a potência do sinal é normalizada para 1.0).

```
def EbNo2Sigma(ebnodb):
    ebno = 10**(ebnodb/10)
    bits_per_complex_symbol = k/(n/2)
    return 1.0/np.sqrt(bits_per_complex_symbol*ebno)
```

- Durante todas as épocas de treinamento, devemos definir o SNR como 7.0 dB, pois foi percebido que treinar um autoencoder em uma taxa de erro de bloco (BLER) de cerca de 0.01 leva a uma generalização rápida por parte do modelo. (**Dica**: para o treinamento, use um mini-batch com 1000 exemplos, passo de aprendizagem igual a 0.0001 e cerca de 100000 iterações, mas não se prenda a estes valores, teste e verifique se existem valores melhores.)

A figura abaixo mostra o sistema fim-a-fim que teremos ao final da implementação.



Agora, depois de termos implementado o sistema fim-a-fim com autoencoder, vamos verificar seu desempenho plotando sua BLER vs SNR em uma faixa de valores de SNR para k = 8 e n = 8. Quando o número de bits, k, e o número de usos reais do canal são iguais a 8, temos que será gerados n/2=4 símbolos complexos onde cada um destes símbolos carrega k/(n/2)=2 bits de informação, o que equivale à modulação QPSK. Portanto, precisamos executar uma simulação de Monte Carlo para obter a BLER para cada valor de SNR. Neste exercício, iremos variar a SNR de 0 a 14 dB em passos de 1 dB executando 10 mini-batches de 100.000 mensagens para cada valor de SNR. Em seguida, devemos comparar, através de uma figura, a BLER obtida com o modelo com a BLER de um sistema convencional de modulação, no caso, QPSK, a qual é dada através do seguinte vetor:

```
BLER_QPSK = np.array([4.80998e-01, 3.71098e-01, 2.63797e-01, 1.68919e-01,
9.54540e-02, 4.68860e-02, 1.89070e-02, 6.11700e-03, 1.58300e-03, 2.91000e-04,
3.30000e-05, 2.00000e-06])
```

Se o treinamento do modelo foi bem sucedido, nós iremos verificar que a BLER do autoencoder é menor do que o da modulação QPSK em toda a faixa de valores de SNR. Se você estiver interessado nas origens desse ganho, dê uma olhada em [1] e [2].

Referências:

[1] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563-575, Oct. 2017.

[2] S. Dörner, S. Cammerer, J. Hoydis, and S. ten Brink, “Deep Learning-based Communication Over the Air,” IEEE J. Sel. Topics Signal Process., vol. 12, no. 1, pp. 132–143, Feb. 2018.