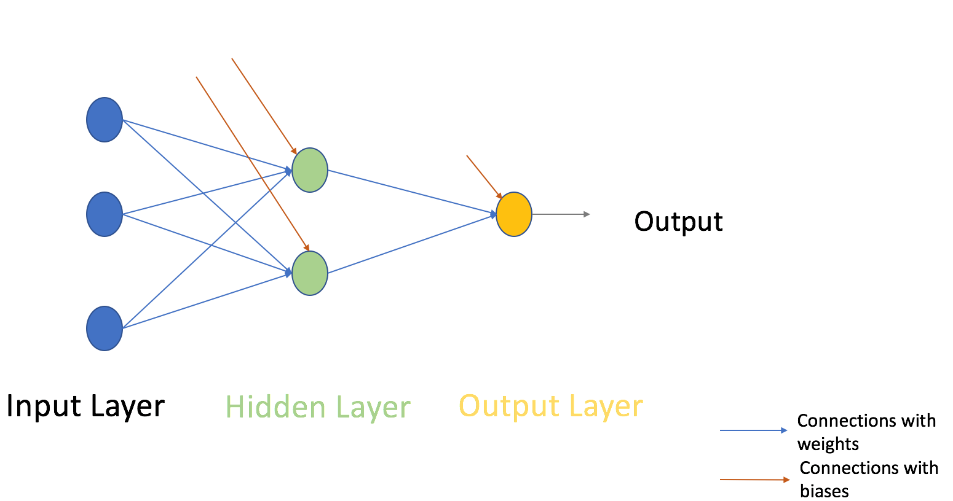
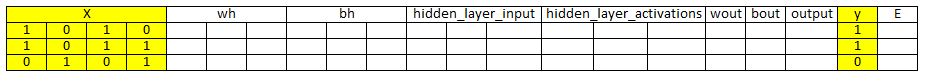
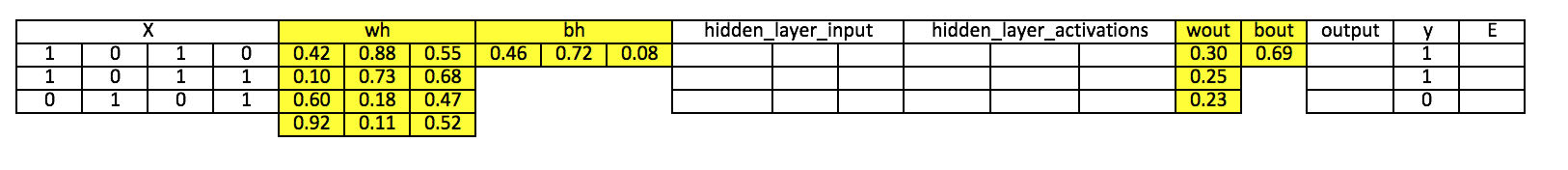
**Passos em uma rede neural em Numpy**



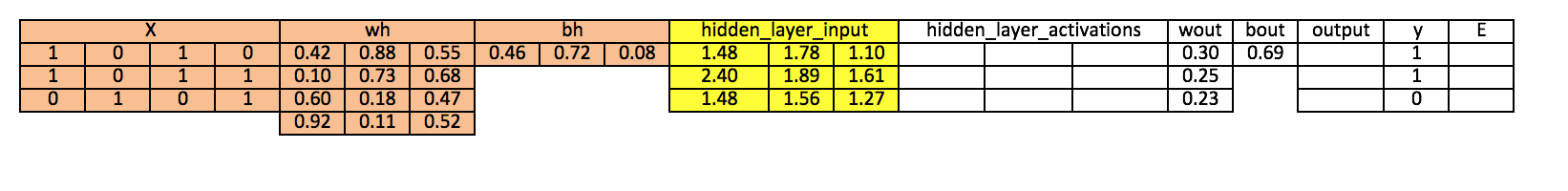
1. Teremos como entrada e saída:
   1. X como matriz de entrada
   2. Y como matriz de saída



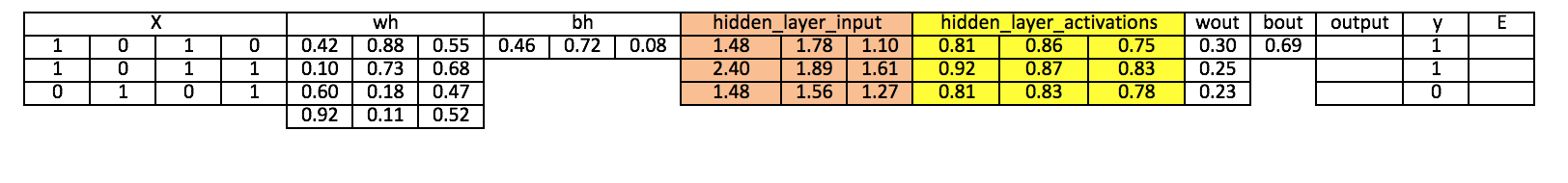
1. Iniciaremos os pesos e os bias com valores aleatórios (Somente no início, na próxima interação serão atualizados):
   1. Wh como a matriz de peso na camada escondida
   2. Bh como a matriz de bias na camada escondida
   3. Wout como matriz de peso na camada de saída
   4. Bout como a matriz de bias na camada de saída



1. Nós teremos o produto da matriz como entrada e os pesos associados entre a entrada e a camada escondida, então adicionaremos os bias nos neurônios da camada escondida na respectiva entrada, isso é conhecido como **transformação linear**:
   1. (hidden\_layer\_input= matrix\_dot\_product(X,wh) + bh)

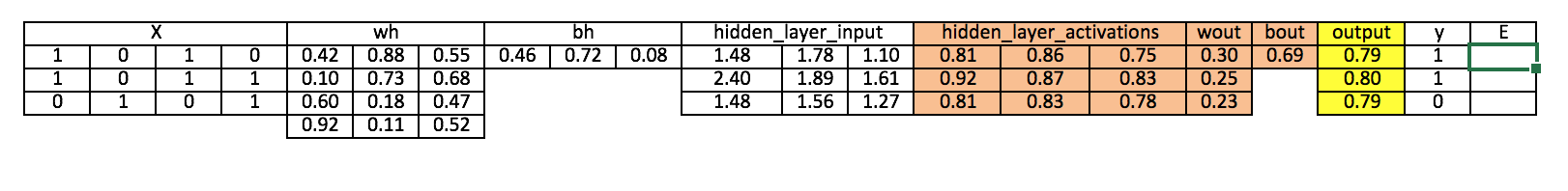


1. Realizar a transformação não-linear usando a função de ativação (Sigmoid). Sigmoid vai retornar a saída como 1/(1 + exp(-x)).
   1. (hiddenlayer\_activations = sigmoid(hidden\_layer\_input)



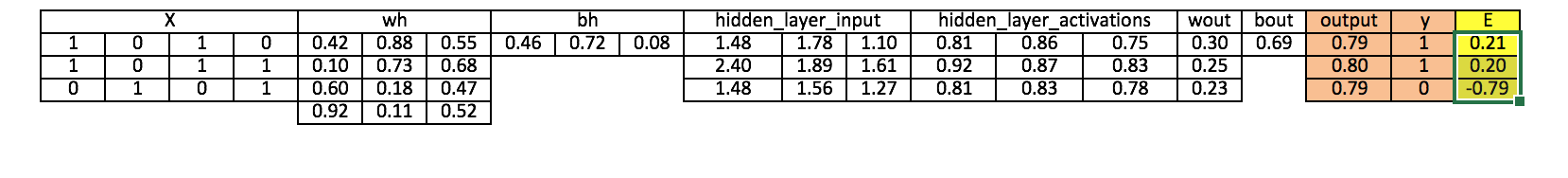
1. Executar uma transformação linear na ativação da camada escondida (pegar o produto da matriz com os pesos e adicionar o bias do neurônio da camada de saída) então aplicar uma função de ativação (de novo usando sigmoid, mas pode ser usada outra função de ativação dependendo da tarefa) para prever a saída:
   1. (output\_layer\_input = matrix\_dot\_product (hiddenlayer\_activations \* wout ) + bout

output = sigmoid(output\_layer\_input))

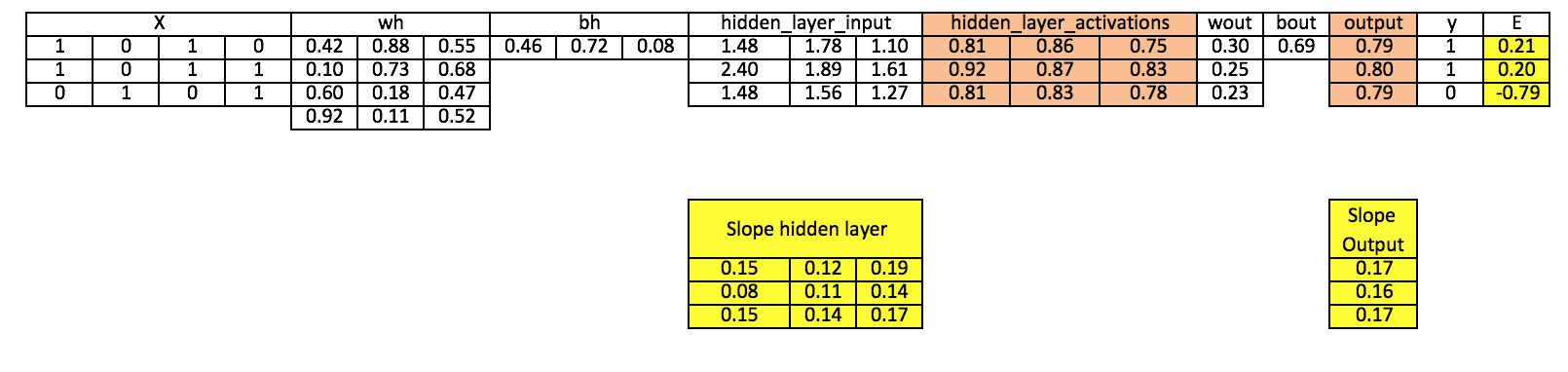


**Todos os passos acimas são conhecidos como “Foward Propagation”**

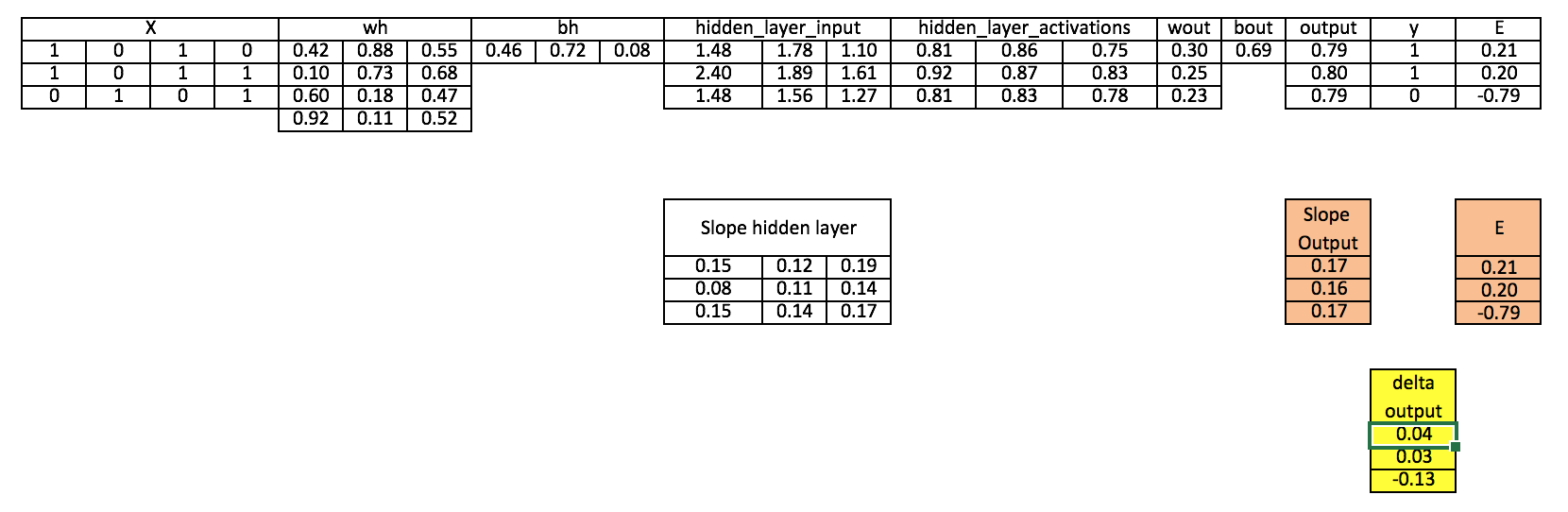
1. Comparar saída esperada com saída atual e calcular o gradiente de erro (Atual - Esperada). Erro é a média da perca ao quadrado = ((Y - t) ^2)/2
   1. E = y – output



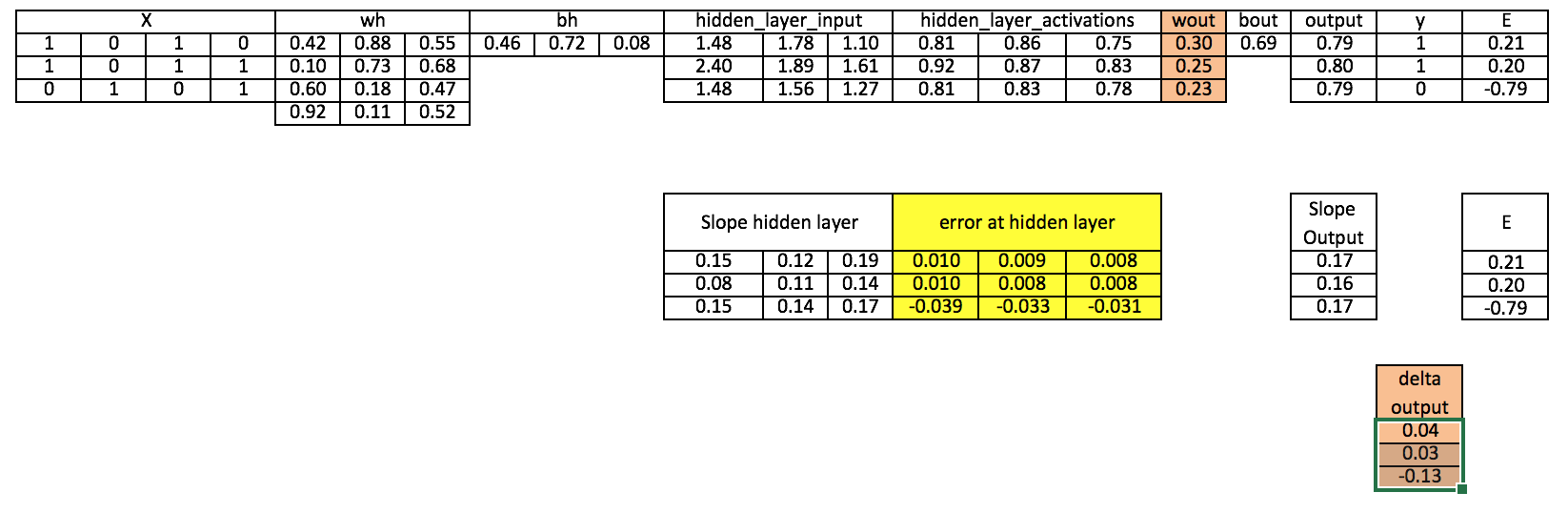
1. Calcular o desvio/gradiente dos neurônios de saída e da camada escondida (Para calcular o desvio, nós calculamos as derivadas das ativações não-lineares x em cada camada para cada neurônio). Gradiente da sigmoid pode ser retornado como x \* (1 - x).
   1. **slope\_output\_layer = derivatives\_sigmoid(output)**
   2. **slope\_hidden\_layer = derivatives\_sigmoid(hiddenlayer\_activations)**



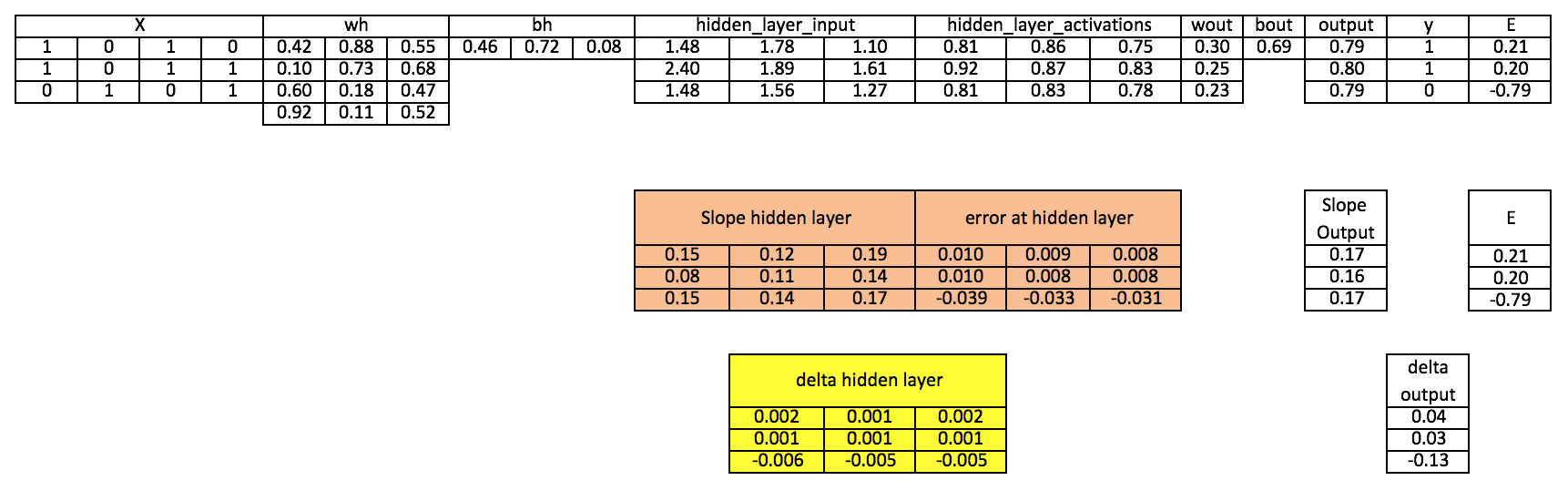
1. Calcular o fator de mudança (delta) na camada de saída, dependente do gradiente de erro multiplicado pelo declive da ativação da camada de saída:
   1. **d\_output = E \* slope\_output\_layer**



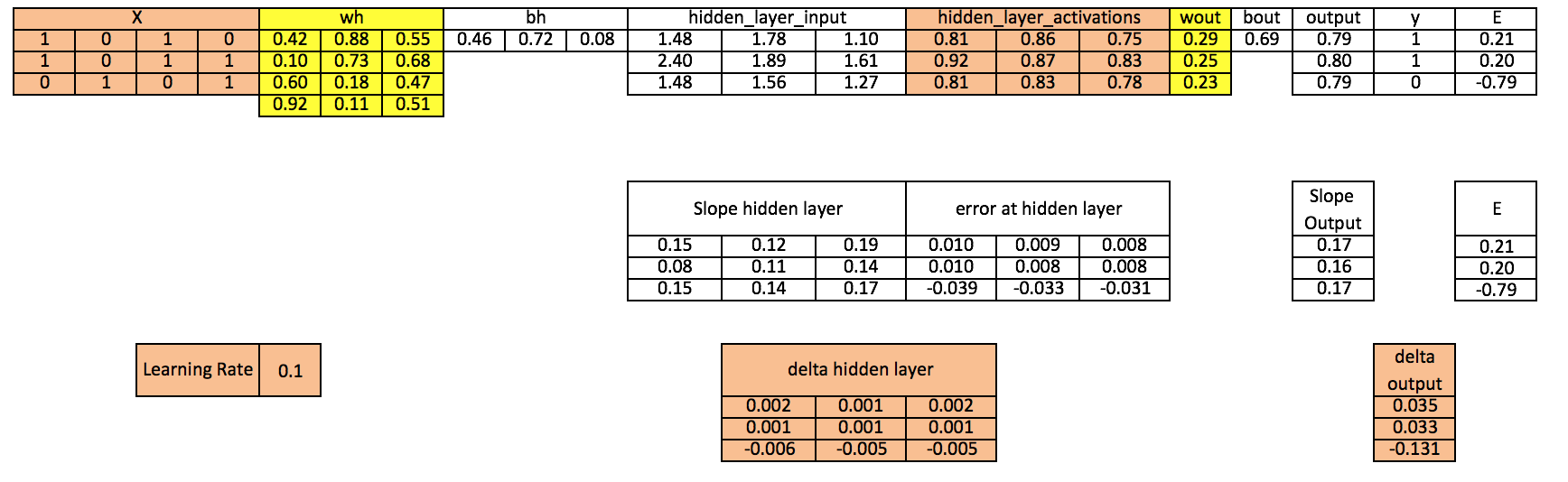
1. Nesse passo, o erro vai propagar de volta na rede o que significa que é o erro da camada escondida. Para isso, nós vamos pegar o produto do delta da camada de saída com os parâmetros de peso entre a camada escondida e a camada de saída.
   1. **Error\_at\_hidden\_layer = matrix\_dot\_product(d\_output, wout.Transpose)**



1. Calcular fator de mudança (delta) na camada escondida, multiplicar o erro na camada escondida com o declive da ativação da camada escondida.
   1. **d\_hiddenlayer = Error\_at\_hidden\_layer \* slope\_hidden\_layer**

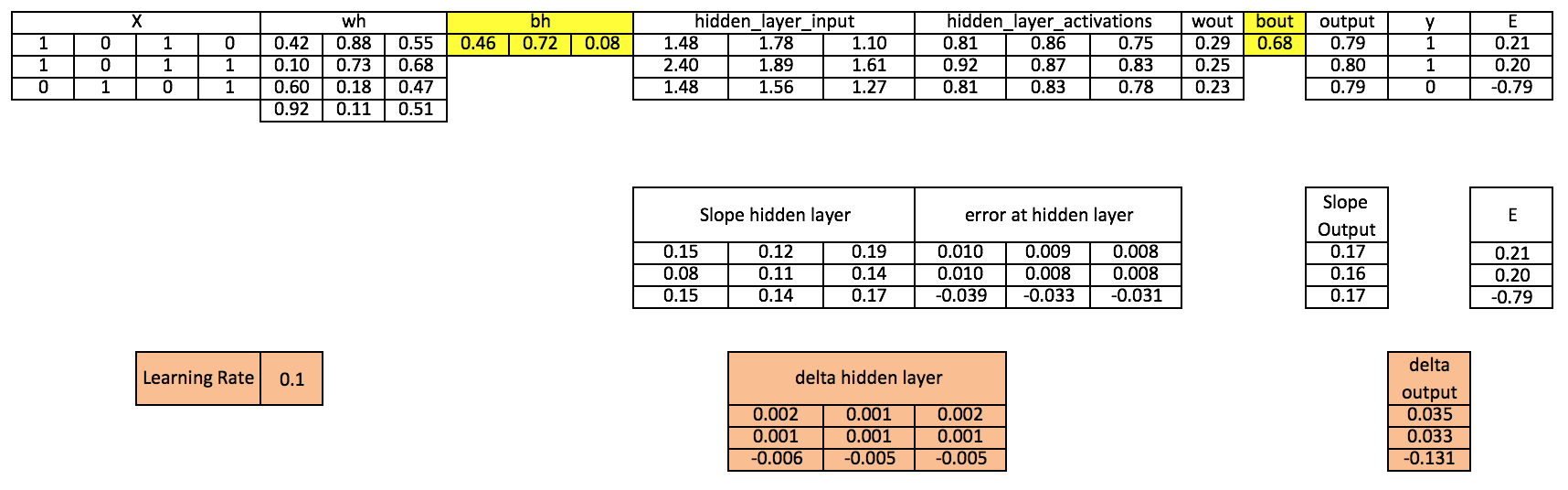


1. Atualizar os erros nas camadas de saída e escondidas: Os pesos na rede podem ser atualizados dos erros calculados para exemplos de treinamento:
   1. **wout = wout + matrix\_dot\_product(hiddenlayer\_activations.Transpose, d\_output)\*learning\_rate**
   2. **wh = wh + matrix\_dot\_product(X.Transpose,d\_hiddenlayer)\*learning\_rate**



learning\_rate: A quantidade dos pesos que vão sendo atualizados são controlados por um parâmetro de configuração chamado de taxa de aprendizado.

1. Atualizar os bias nas camadas escondidas e saída: Os bias na rede podem ser atualizados dos erros agregados no respectivo neurônio:
   1. Bias na camada de saída - bias at output\_layer + sum of delta of output\_layer at row-wise \* learning\_rate
   2. Bias na camada escondida = bias at hidden\_layer + sum of delta of output\_layer at row-wise \* learning\_rate
      1. **bh = bh + sum(d\_hiddenlayer, axis=0) \* learning\_rate**
      2. **bout = bout + sum(d\_output, axis=0)\*learning\_rate**



**Passos de 5 até 11 são conhecidos como “Backward Propagation”**

Uma iteração Foward e uma Backward Propagation é conhecida como um ciclo de treinamento. Como foi mencionado anteriormente, como nós treinamos pela segunda vez os pesos e bias atualizados são usados na foward propagation.

Acima, nós atualizamos o peso e os bias para as camadas escondidas e de saída e nós utilizamos um algoritmo de descida de gradiente de lote completo.

Acima, você pode ver que a saída não foi igual à esperada por que foi feito somente uma iteração, após mil iterações o resultado foi esse: ([[ 0.98032096] [ 0.96845624] [ 0.04532167]]).