# Back Propagation - self

May 15, 2023

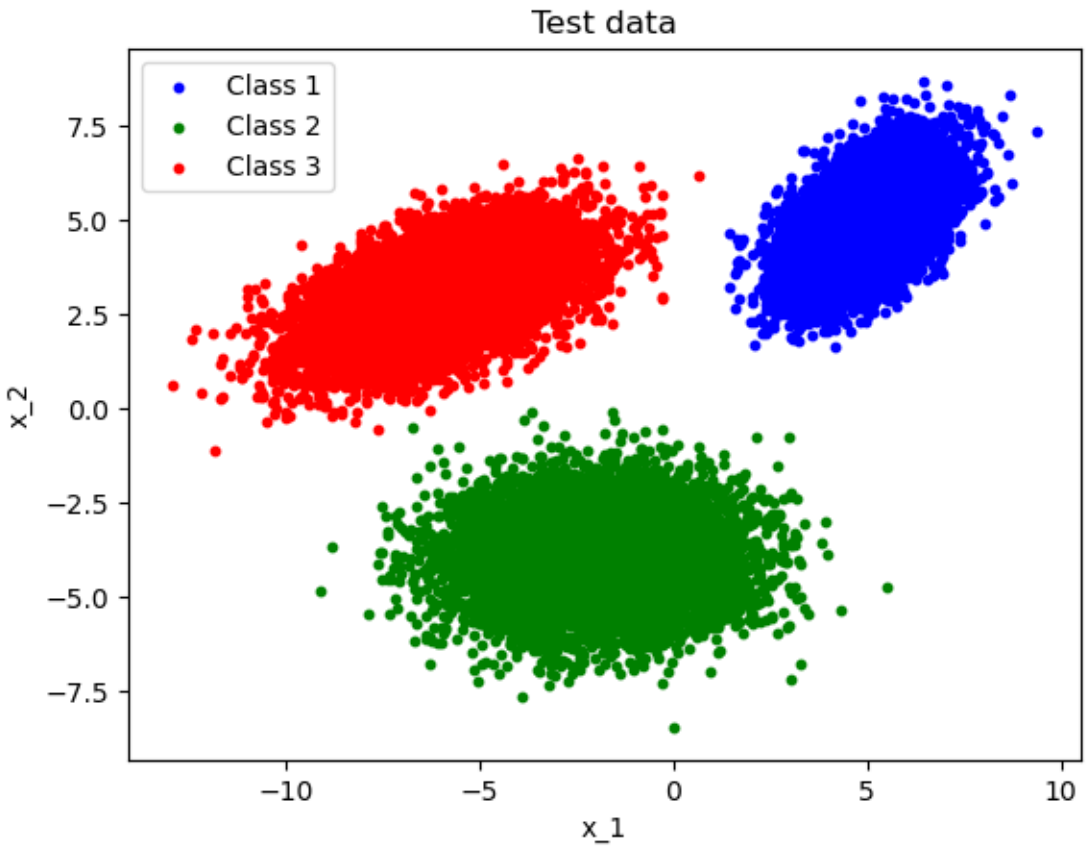
## 1 Imports

[1]:
```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
import data # Custom data generation
from tqdm import tqdm # For-loop progress bar
```

## 2 Data generation & shaping

[2]:
```
X_1, X_2, X_3 = data.get_data()
Y_1, Y_2, Y_3 = np.array([[1,0,0] for _ in range(X_1.shape[0])]), \
                np.array([[0,1,0] for _ in range(X_2.shape[0])]), \
                np.array([[0,0,1] for _ in range(X_3.shape[0])])
```

[3]:
```
#figure
plt.scatter(X_1[:,0],X_1[:,1], color='b', s=10, label='Class 1')
plt.scatter(X_2[:,0],X_2[:,1], color='g', s=10, label='Class 2')
plt.scatter(X_3[:,0],X_3[:,1], color='r', s=10, label='Class 3')

plt.xlabel('x_1')
plt.ylabel('x_2')
plt.title('Test data')
plt.legend()
plt.show()
```




Below is for shaping the data correctly. The three sets are joined, however x- and y-dimensions are still kept separate. Note that a bias dimension is also added to the input set; this is a column of 1's, since adding a bias to the network is the same as expanding the input layer & hidden layers by size 1 and letting the bias "hide" within the weights matrix, and simply applying a stimulation clamped at 1 (as explained by Bishop Pattern Recognition[…] sections 3 and 5).

[4]:
```
X_collected = np.concatenate((X_1, X_2, X_3))
Y_collected = np.concatenate((Y_1, Y_2, Y_3))

N, n = X_collected.shape[0], X_1.shape[0]
X_with_bias = np.column_stack((np.ones(N), X_collected))

# below is for shuffling the data and creating a training- and test set.
# since x and y values are paired, we zip them before shuffling
temp = list(zip(X_with_bias, Y_collected))
np.random.shuffle(temp)
x_shuffled, y_shuffled = zip(*temp) # unzip and assign to variables
x_shuffled, y_shuffled = np.array(x_shuffled), np.array(y_shuffled)
fraction = 1/2; split_point = int(N*fraction)
x_train, x_test = x_shuffled[:split_point], x_shuffled[split_point:]
```

```
y_train, y_test = y_shuffled[:split_point], y_shuffled[split_point:]
```

## 3 Network & Training parameters

Below are simply parameters and functions which are useful later. Note that we use a sigmoid activation function for both layers; another implementation using a sigmoid in the hidden layer and softmax at the output layer can be found below. Also note that the MSE is not calculated correctly; instead of dividing by N, we only divide by 2. This is entirely meaningless, since is is only a relativistic metric. Dividing by 2 just means that the values are easier to plot later.

[93]:
```
ETA = 0.1 # Learning rate
EPOCH = 1000 # No. of times the training set is run through
SIGMA = 1 # Random weight initialization from normal distribution
INPUT_SIZE = 2 # Input, output and hidden layer dimensions
HIDDEN_SIZE = 4
OUTPUT_SIZE = 3
```

[6]:
```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_derivative(x):
    y = sigmoid(x)
    return y * (1.0 - y)

def sum_of_squares_error(y_pred, y_true):
    return sum( (sum( (y_pred - y_true)** 2 )) / 2 )

def same_answer(pred, true):
    return 1 if np.argmax(pred) != np.argmax(true) else 0

def count_errors(pred, true):
    return sum(map(same_answer, pred, true))
```

[94]:
```
#np.random.seed(1) # setting the seed allows for constant "random weights"
# Note that the input- and hidden layers are expanded by 1
W1 = np.random.normal(scale=SIGMA, size=(INPUT_SIZE+1, HIDDEN_SIZE+1))
W2 = np.random.normal(scale=SIGMA, size=(HIDDEN_SIZE+1, OUTPUT_SIZE))
errors = [0 for _ in range(EPOCH)]
error_counts = [0 for _ in range(EPOCH)]
```

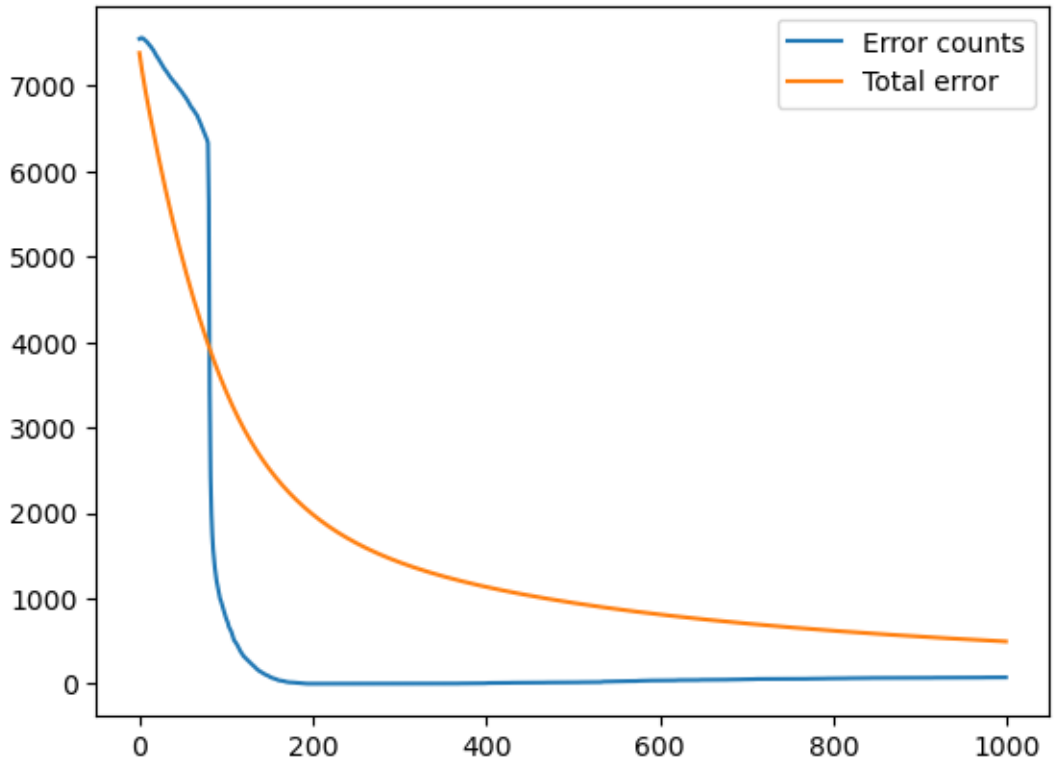
Below is the actual training loop. Note two things:

1) First, we count both the sum-of-squares-error AND the number of incorrect guesses from the network. This last parameter is merely for fun and investigating how soon the network is able to guess correctly compared with how the total error evolves over time
2) Second, we batch the entire training set together and calculate the weight updates as averages from each individual weight-change summed together.

[95]:
```
N = x_train.shape[0]
for i in tqdm(range(EPOCH)):
    # Forward propagation through both layers
    node1 = x_train @ W1
    activation1 = sigmoid(node1)
    node2 = activation1 @ W2
    activation2 = sigmoid(node2)

    # Count sum of squares & number of wrong guesses
    errors[i] = sum_of_squares_error(activation2, y_train)
    error_counts[i] = count_errors(activation2, y_train)

    # Backpropagation through both layers
    last_error = activation2 - y_train
    last_error_delta = last_error * sigmoid_derivative(activation2)
    first_error = last_error_delta @ W2.T
    first_error_delta = first_error * sigmoid_derivative(activation1)

    # Update weights
    W2_change = (activation1.T @ last_error_delta) / N
    W1_change = (x_train.T @ first_error_delta) / N

    W2 = W2 - ETA * W2_change
    W1 = W1 - ETA * W1_change
```

```
100%|                | 1000/1000 [00:50<00:00, 19.71it/s]
```

[96]:
```
plt.plot(error_counts, label='Error counts')
plt.plot(errors, label='Total error')
plt.legend()
plt.show()
```




[97]:
```
min_errors = np.argmin(error_counts)
print(f'Minimum errors at {min_errors} with {error_counts[min_errors]} errors')
print(f'Error counts afterwards: {error_counts[min_errors:min_errors+10]}. It␣
 ↪ends at {error_counts[-1]}')
```

```
Minimum errors at 219 with 0 errors
Error counts afterwards: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]. It ends at 76
```

It is notable that in the above learning process, the amount of incorrect guesses is at its minimum after only 219 epochs with just 0 errors - after this, it only rises, albeit slightly. The MSE, however, continues falling. This indicates that the model begins overfitting around this point. However, as can be seen below, the amount of errors in the test set is still quite small.

### 3.1 Count number of errors in test set

[98]:
```
x_test_z1 = x_test @ W1
x_test_a1 = sigmoid(x_test_z1)
x_test_z2 = x_test_a1 @ W2
x_test_a2 = sigmoid(x_test_z2)
error_count = count_errors(x_test_a2, y_test)
print(f'Error count {error_count} out of {x_test.shape[0]} total tests')
print(f'A total of {error_count/x_test.shape[0]*100:.3}%')
```

```
Error count 95 out of 15000 total tests
A total of 0.633%
```

## 4 Random weight initialization

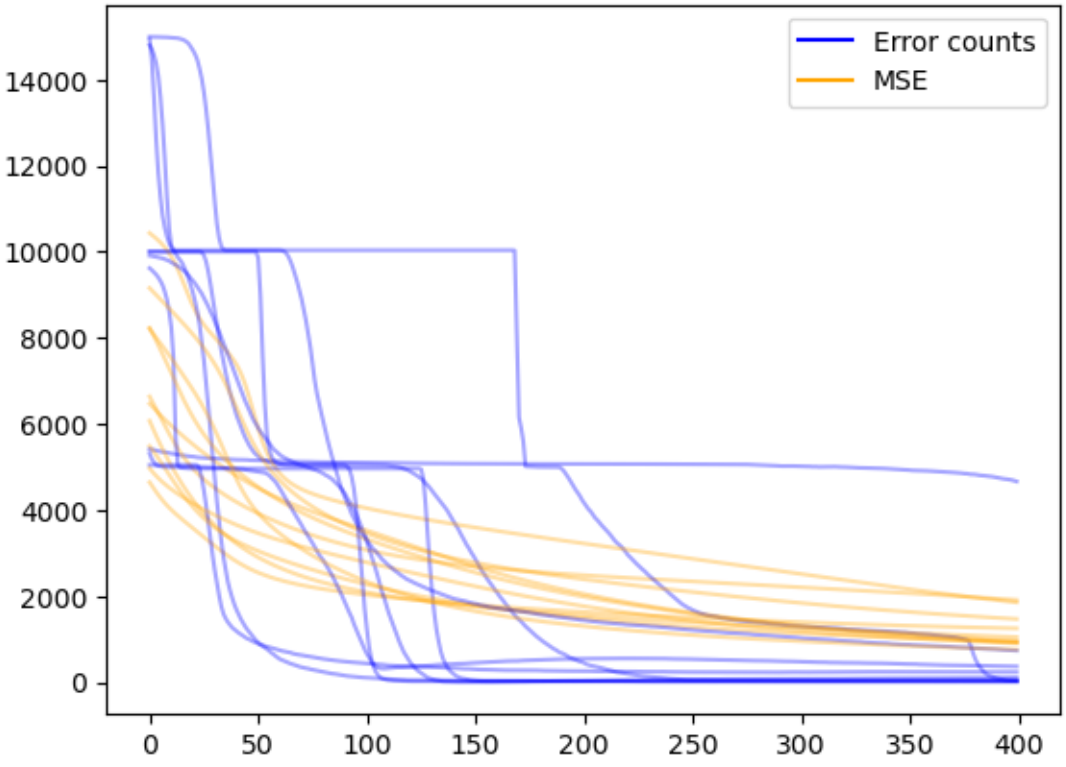
Next, it could be interesting to train the network 10 times using different starting weights. The amount of epochs is also reduced to 400, since most of what happens afterwards looks like overfitting.

[49]:
```
N = x_train.shape[0]
EPOCH = 400
weights_set_sig = []
errors_set_sig = []
error_counts_set_sig = []
for _ in range(10):
    W1 = np.random.normal(scale=SIGMA, size=(INPUT_SIZE+1, HIDDEN_SIZE+1))
    W2 = np.random.normal(scale=SIGMA, size=(HIDDEN_SIZE+1, OUTPUT_SIZE))
    errors = [0 for _ in range(EPOCH)]
    error_counts = [0 for _ in range(EPOCH)]
    for i in tqdm(range(EPOCH)):
        # Forward propagation through both layers
        node1 = x_train @ W1
        activation1 = sigmoid(node1)
        node2 = activation1 @ W2
        activation2 = sigmoid(node2)

        # Count sum of squares & number of wrong guesses
        errors[i] = sum_of_squares_error(activation2, y_train)
        error_counts[i] = count_errors(activation2, y_train)

        # Backpropagation through both layers
        last_error = activation2 - y_train
        last_error_delta = last_error * sigmoid_derivative(activation2)
        first_error = last_error_delta @ W2.T
        first_error_delta = first_error * sigmoid_derivative(activation1)

        # Update weights
        W2_change = (activation1.T @ last_error_delta) / N
        W1_change = (x_train.T @ first_error_delta) / N

        W2 = W2 - ETA * W2_change
        W1 = W1 - ETA * W1_change
    weights_set_sig.append([W1,W2])
    errors_set_sig.append(errors)
    error_counts_set_sig.append(error_counts)
```

```
100%|
                | 400/400 [00:20<00:00, 19.61it/s]
100%|
```

```
                    | 400/400 [00:20<00:00, 19.69it/s]
100%|
                    | 400/400 [00:20<00:00, 19.53it/s]
100%|
                    | 400/400 [00:20<00:00, 19.56it/s]
100%|
                    | 400/400 [00:20<00:00, 19.59it/s]
100%|
                    | 400/400 [00:20<00:00, 19.71it/s]
100%|
                    | 400/400 [00:20<00:00, 19.67it/s]
100%|
                    | 400/400 [00:20<00:00, 19.26it/s]
100%|
                    | 400/400 [00:20<00:00, 19.40it/s]
100%|
                    | 400/400 [00:20<00:00, 19.86it/s]
```

[51]:
```
for i in range(10):
    plt.plot(error_counts_set_sig[i], c='b', alpha=0.35)
    plt.plot(errors_set_sig[i], c='orange', alpha=0.35)
plt.plot([], c='b', label='Error counts')
plt.plot([], c='orange', label='MSE')
plt.legend()
plt.show()
```



It can be seen that nearly all of the iterations converge in the end, however some take some time getting there. The MSE's are all reduced in the expected manner, however the error counts often plateau and stay at a specific point for some time. This indicates that multiple iterations with random weights are useful (perhaps necessary) with this dataset / method, especially if doing less than 400 iterations. One of the tries seems to have hit a local minimum, which is why it is stuck. This signifies the importance of trying multiple times.

## 5 Using Softmax

The exact same process is repeated (random weights, 10 iterations, forward- and backpropagation etc), however this time with Softmax instead of Sigmoid as the output layer activation function.

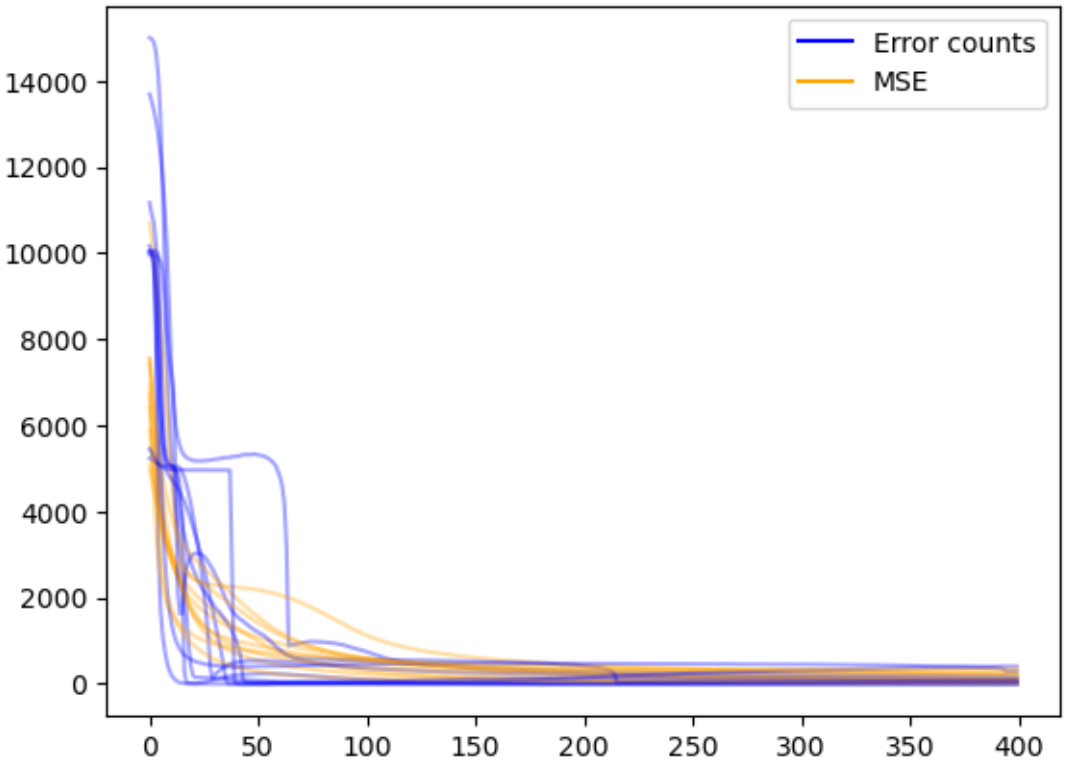
[31]:
```
def softmax(x):
    out = np.exp(x)
    total = np.sum(out, axis=1)
    return out / total[:,None]
```

[50]:
```
N = x_train.shape[0]
EPOCH = 400
weights_set_soft = []
errors_set_soft = []
error_counts_set_soft = []
for _ in range(10):
    W1 = np.random.normal(scale=SIGMA, size=(INPUT_SIZE+1, HIDDEN_SIZE+1))
    W2 = np.random.normal(scale=SIGMA, size=(HIDDEN_SIZE+1, OUTPUT_SIZE))
    errors = [0 for _ in range(EPOCH)]
    error_counts = [0 for _ in range(EPOCH)]
    for i in tqdm(range(EPOCH)):
        # Forward propagation through both layers
        node1 = x_train @ W1
        activation1 = sigmoid(node1)
        node2 = activation1 @ W2
        activation2 = softmax(node2)

        # Count total error and
        errors[i] = sum_of_squares_error(activation2, y_train)
        error_counts[i] = count_errors(activation2, y_train)

        # Backpropagation through both layers
        last_error = activation2 - y_train
        W2_change = (activation1.T @ last_error) / N
        W1_change = (((last_error @ W2.T) * sigmoid_derivative(activation1)).T␣
↪@ x_train) / N

        W2 = W2 - ETA * W2_change
```

```
        W1 = W1 - ETA * W1_change.T
    weights_set_soft.append([W1,W2])
    errors_set_soft.append(errors)
    error_counts_set_soft.append(error_counts)
```

```
100%|
                  | 400/400 [00:19<00:00, 20.07it/s]
100%|
                  | 400/400 [00:20<00:00, 20.00it/s]
100%|
                  | 400/400 [00:19<00:00, 20.10it/s]
100%|
                  | 400/400 [00:20<00:00, 19.95it/s]
100%|
                  | 400/400 [00:20<00:00, 20.00it/s]
100%|
                  | 400/400 [00:19<00:00, 20.03it/s]
100%|
                  | 400/400 [00:19<00:00, 20.06it/s]
100%|
                  | 400/400 [00:19<00:00, 20.06it/s]
100%|
                  | 400/400 [00:19<00:00, 20.02it/s]
100%|
                  | 400/400 [00:19<00:00, 20.03it/s]
```

[33]:
```
for i in range(10):
    plt.plot(error_counts_set_soft[i], c='b', alpha=0.35)
    plt.plot(errors_set_soft[i], c='orange', alpha=0.35)
plt.plot([], c='b', label='Error counts')
plt.plot([], c='orange', label='MSE')
plt.legend()
plt.show()
```

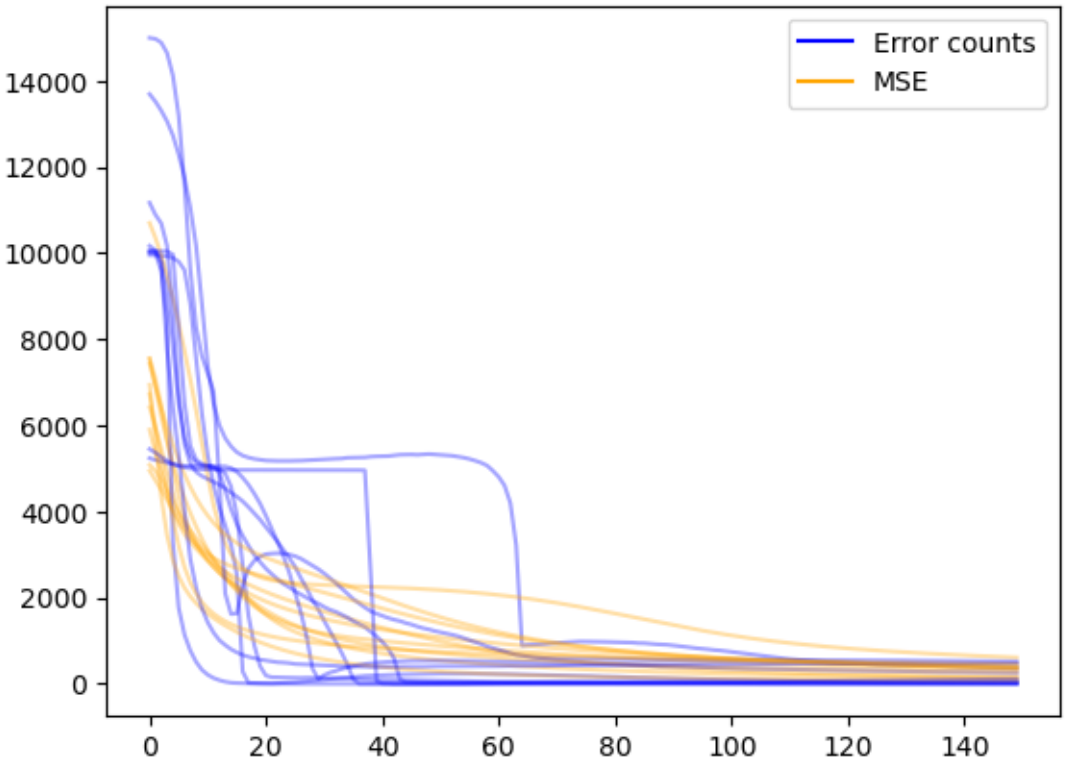



It is immediately obvious that both the MSE and error counts are much lower than before, which means that the Softmax activation function is much better suited for this task, which makes sense since the Softmax function gives the probability for each output node that THAT node is the correct one, whereas the Sigmoid function gives the probability that each output node is correct, independent from the others.

Since only one node can be the correct answer to this problem, the Softmax makes more sense. The Sigmoid function would be useful e.g. if one were to guess whether an image contained a cat or a dog, since both possibilities could be true at the same time.

To improve clarity, the first 150 values are plotted again and the final error count is displayed for each try. This is then compared to the similar error counts when using 10 weight initializations and the Sigmoid activation function:

[36]:
```
for i in range(10):
    plt.plot(error_counts_set_soft[i][:150], c='b', alpha=0.35)
    plt.plot(errors_set_soft[i][:150], c='orange', alpha=0.35)
plt.plot([], c='b', label='Error counts')
plt.plot([], c='orange', label='MSE')
plt.legend()
plt.show()
```




[56]:
```
counts_sigmoid, counts_softmax = [], []
for w1,w2 in zip(weights_set_sig, weights_set_soft):
    W1, W2 = w1[0], w1[1]
    x_test_z1 = x_test @ W1
    x_test_a1 = sigmoid(x_test_z1)
    x_test_z2 = x_test_a1 @ W2
    x_test_a2 = sigmoid(x_test_z2)
    counts_sigmoid.append(count_errors(x_test_a2, y_test))
    W1, W2 = w2[0], w2[1]
    x_test_z1 = x_test @ W1
    x_test_a1 = sigmoid(x_test_z1)
    x_test_z2 = x_test_a1 @ W2
    x_test_a2 = softmax(x_test_z2)
    counts_softmax.append(count_errors(x_test_a2, y_test))

print(f'Sigmoid error counts: {", ".join([str(i) for i in counts_sigmoid])}')
print(f'Out of a total 15000, this corresponds to between \
{min(counts_sigmoid)/x_test.shape[0]*100:.3}% and {max(counts_sigmoid)/x_test.
  ↪shape[0]*100:.3}%')

print(f'Softmax error counts: {", ".join([str(i) for i in counts_softmax])}')
print(f'Out of a total 15000, this corresponds to between \
```

```
{min(counts_softmax)/x_test.shape[0]*100:.3}% and {max(counts_softmax)/x_test.
↪shape[0]*100:.3}%')
```

```
Sigmoid error counts: 4583, 58, 29, 131, 4, 256, 39, 391, 734, 25
Out of a total 15000, this corresponds to between 0.0267% and 30.6%
Softmax error counts: 1, 18, 5, 4, 147, 19, 371, 133, 51, 264
Out of a total 15000, this corresponds to between 0.00667% and 2.47%
```

Thus Softmax is "just better", and reaches the correct point faster. It can also be noted that the MSE is reduced at a faster pace than when using Sigmoid, where it does not really converge - except when running 1000 epochs instead of just 400.

## 6 Hidden Layer Nodes as a Parameter

[84]:
```
N = x_train.shape[0]
EPOCH = 400
weights_set_nodes = []
errors_set_nodes = []
error_counts_set_nodes = []
for hidden_nodes in range(16):
    W1 = np.random.normal(scale=SIGMA, size=(INPUT_SIZE+1, hidden_nodes+1))
    W2 = np.random.normal(scale=SIGMA, size=(hidden_nodes+1, OUTPUT_SIZE))
    errors = [0 for _ in range(EPOCH)]
    error_counts = [0 for _ in range(EPOCH)]
    for i in tqdm(range(EPOCH)):
        # Forward propagation through both layers
        node1 = x_train @ W1
        activation1 = sigmoid(node1)
        node2 = activation1 @ W2
        activation2 = softmax(node2)

        # Count total error and
        errors[i] = sum_of_squares_error(activation2, y_train)
        error_counts[i] = count_errors(activation2, y_train)

        # Backpropagation through both layers
        last_error = activation2 - y_train
        W2_change = (activation1.T @ last_error) / N
        W1_change = (((last_error @ W2.T) * sigmoid_derivative(activation1)).T␣
↪@ x_train) / N

        W2 = W2 - ETA * W2_change
        W1 = W1 - ETA * W1_change.T
    weights_set_nodes.append([W1,W2])
    errors_set_nodes.append(errors)
    error_counts_set_nodes.append(error_counts)
```

```
70%|                    | 281/400 [00:13<00:05,
```

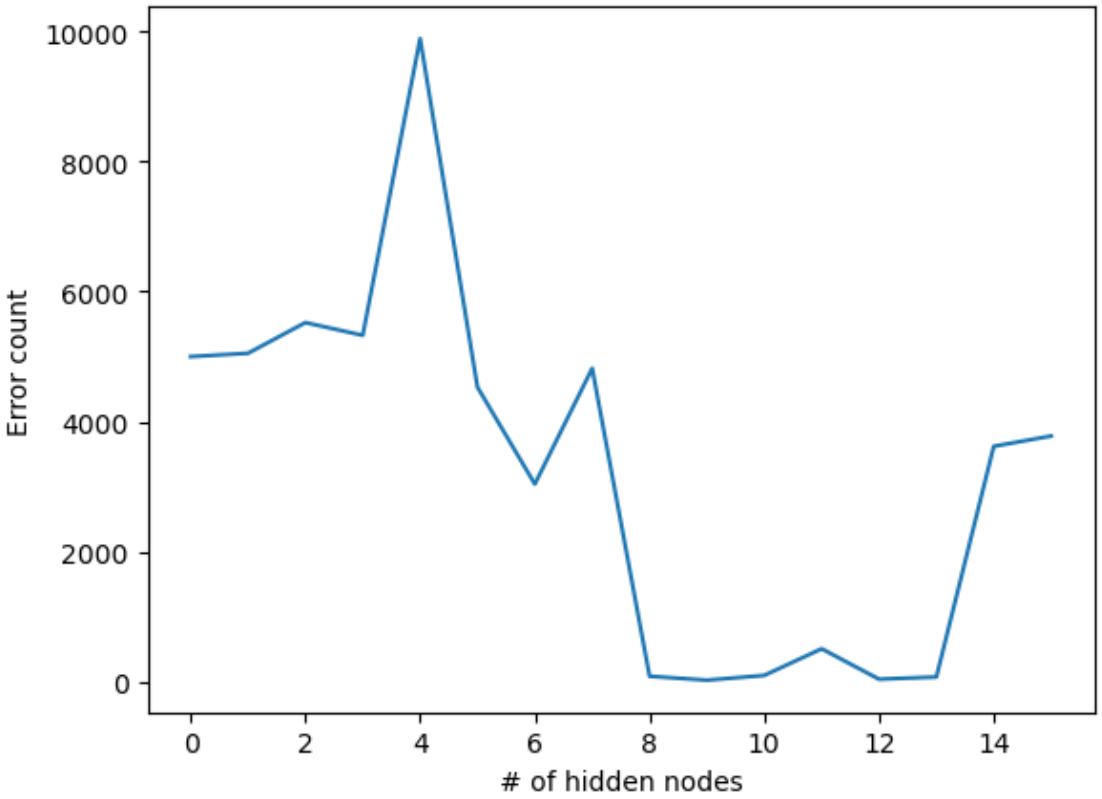
```
20.42it/s]/tmp/ipykernel_210923/2617288413.py:2: RuntimeWarning: overflow
encountered in exp
  return 1.0 / (1.0 + np.exp(-x))
100%|                    | 400/400 [00:19<00:00, 20.62it/s]
100%|                    | 400/400 [00:19<00:00, 20.47it/s]
100%|                    | 400/400 [00:19<00:00, 20.41it/s]
100%|                    | 400/400 [00:19<00:00, 20.34it/s]
100%|                    | 400/400 [00:20<00:00, 19.98it/s]
100%|                    | 400/400 [00:20<00:00, 19.70it/s]
100%|                    | 400/400 [00:20<00:00, 19.16it/s]
100%|                    | 400/400 [00:20<00:00, 19.32it/s]
100%|                    | 400/400 [00:20<00:00, 19.25it/s]
100%|                    | 400/400 [00:20<00:00, 19.16it/s]
100%|                    | 400/400 [00:21<00:00, 19.03it/s]
100%|                    | 400/400 [00:21<00:00, 18.78it/s]
100%|                    | 400/400 [00:21<00:00, 18.69it/s]
100%|                    | 400/400 [00:21<00:00, 18.39it/s]
100%|                    | 400/400 [00:21<00:00, 18.21it/s]
100%|                    | 400/400 [00:23<00:00, 17.29it/s]
```

[85]:
```
node_errors = []
for w in weights_set_nodes:
    W1, W2 = w[0], w[1]
    x_test_z1 = x_test @ W1
    x_test_a1 = sigmoid(x_test_z1)
    x_test_z2 = x_test_a1 @ W2
    x_test_a2 = softmax(x_test_z2)
    node_errors.append(count_errors(x_test_a2, y_test))
```

```
/tmp/ipykernel_210923/2617288413.py:2: RuntimeWarning: overflow encountered in
exp
  return 1.0 / (1.0 + np.exp(-x))
```

```
[5002, 5053, 5525, 5329, 9892, 4532, 3043, 4822, 88, 28, 101, 511, 43, 77, 3623,
3781]
```

[87]:
```
plt.plot(node_errors)
plt.xlabel('# of hidden nodes')
plt.ylabel('Error count')
plt.show()
```




The error count seems to indicate that 8-13 hidden nodes gives the optimal performance of our network. This is perhaps since too few will not be able to handle the necessary complexion of the task, and too many will mean that we do not have enough data to train the network enough. Because of this, 10 hidden nodes have been chosen in the PyTorch exercise.