Jet Ryan
12/11/19
Machine Learning Final Project

Abstract:

This paper details the process to construct a 2-layer neural network. the network is compared with linear classifier with well-chosen basis to show the contrast with the effectiveness of each of the methods, in time, space, and complexity, and result. The error rate that I achieved with the a 2-layer neural network was 7.55% and the error rate that I achieved for the linear classifier was 12.73%.

Theoretical Description: provide a theoretical description of the algorithms(s)

Theoretical Description:

The first algorithm that I will break down is the supporting algorithm, the linear classifier. While I decided to implement the 2-layer before I implemented the linear classifier it is good to understand the background of the linear classifier to understand why the 2-layer preforms better. The linearly takes each of the individual pixels from the image and then maps the values directly to an output using a function that can turn input values into a probability [1]. Those probabilities are then used to classify the data. The first step universally is flattening the image to a 1-D vector this image is prepended with a bias node then passed through the chosen basis function. After it is passed through the basis it is multiplied by the initial weights. This is now passed through the activation function and it is classified. The next step is to adjust the weight so that on the next time that the forward propagation is done the weights are more accurate then the previous iteration.

The 2 -layer Neural network has many of the same features that the linear classifier uses. The major differences stem from the linear classifier going from input to output, while the 2-neural network has the input sent to the hidden layer which is then pushed to the output. The input is read in the same way and the images are flattened the same as above. Once basis function is determined, the image with the bias node is feed into it. The output of the basis is multiplied by weight1 matrix. At this point the result from that operation are assigned to the hidden layer, and another bias note is assigned to the hidden layer. Those are the results for the first round of forward propagation. The hidden layer is put through its own activation function. These are the values that will be used in the second forward propagation. The next step is to take the outputs from the forward propagation and multiply those with the weight2 matrix the result is then passed through another activation function and it is assigned to the output nodes. The final step is to use a method to adjust the weight1 and weight2 for the next iteration.

Implementation Details: explain how the major parts of your code work, including references to functions and line numbers

Implementation Details:

The parts of the code that are shared by both methods will be described in this section. The images of the block are in the appendix, but in order to see the full code please refer to the code that is attached separately. The first component to mention that both methods have in common for classifying 10 possible classes is the use of the one hot encoding function. This is used to change a class label for an image to 1 by n-class size array with all zeros where there is only a one present at the corresponding class. The second item that is shared by both models is the demo function. This function handles all the preprocessing of the data. Because the data is feed into both algorithms the same this part just flattens the image, one-hot encodes the target, and pushes them to the vectors that are passed into the train function. The activation functions that are used are sigmoid, and SoftMax, along with each of their corresponding derivatives. The sigmoid function is the activation used on the hidden layer. The derivative of the sigmoid function derivative is used for the grad2 calculation for the back propagation. The SoftMax function is used on the output layer to classify. It derivative is used in the grad2 calculation. The last item that both these methods share is the test_train_error function. This function is only called after train is complete and the “optimal” weights are found. This function opens the test and train data and uses the weights to determine the number of misclassification and the total error rate.

The functions used in the linear classifier are named the same as the 2-layer neural network and accomplish the same task. The three functions that are under question are train, test, and error_rate function.

The linear classifier has the main difference in amount of code, without the hidden layer most of the calculations that need to be done are cut in half. When the items are first read into the train function the number of samples, the dimension of the input, the number of classes, the training and hold out set are all calculated the values of the data are then shuffled up in a random order. At this point I determined the best batch size for the data set. The initial weights were also decided at this point. More details on how I got the values for the Batch size, the initial weights, the learning rate, and the loop condition in the Reproducibility section below.

The actual implementation of the linear classifier is very simple once all the initializations are complete. For the input image x, I decided to go with the linear basis function this was the basis function that made the most sense to me when designing the network. When it comes to the actual linear classifier the calculations are the same as mentioned in the Theoretical Description section. They are also located in the appendix below. The next step is handling the back propagation, there are two main methods to handle back prop the stochastic gradient descent that was originally implemented in the backpropagation code and there is the Adam [4]. I decided that Adam was the better method to implement. Adam is a bit like RMSProp but it has the addition of a momentum term that makes it a little different. I set the values to the beta1, beta2 and the very small value eps. I then replace the step gradient decent with the Adam code.

Reproducibility:

The beginning of the step by step process of how to rebuild my results is quite simply. There are two main stages that I notice when building any of these machine learning algorithms. The first one is getting the algorithm to a state where it correctly operates, and the second state is the optimization state, where all the parameters are tweaked until the result is where it needs to be.

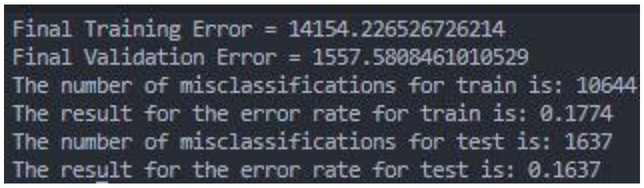
The previous sections of this paper have detailed how to get the algorithm to a state where it will work correctly. This section will discuss the optimization values that I selected. These were the values that worked the best for me. The main parameters that require optimization for the code to produce good and accurate output are Learning Rate, Batch Size, Hidden Nodes amount, the initial weight multipliers, and loop exit conditions. The values for the parameters I selected for the 2NN is .0011 for the learning rate, 300 for the number of hidden nodes, 256 for the batch size, and .01 is what I multiplied the initial weights by. For the linear classifier the values were much different. The learning rate is .000018, the batch size is 1000, and the initial weight multiplier is .00083. For the 2NN those parameters caused the network to converge constantly until it would meet the value in which my running error_rate < .08. And for the linear classifier unit my batch error rate was under 1300. I noticed that as I increased the number of hidden nodes I did not notice much of a change in the results. I did notice a much larger increase in time. I noticed that learning rate had a large effect on the size of the jump that the error would make each iteration. The initial value for the weight would determine if the networks would even converge at all. If the weight multipliers are off the network won't work.

Results:

The result of the implementation of the 2-Layer Neural Network was 7.55% for the error rate. The total number of misclassifications in the test set is 755 out of 10,000, in the training set I received 7.94% for the error rate and the number of misclassifications is 4764 out of 60,000. This compared to the linear model is much better. The error rate I got 12.73% with a number of 1273 misclassified out of 10,000, for the test data and 13.48% with a number 8086 misclassified out of 60,000. In terms of the time and the space complexity the time that it takes to run the 2-layer is longer than the linear classifier. This is mainly because the of the looping structure that the code is built in. I was able to convert most everything that looped in the code to a matrix operation. The only value that I could not find a way to correctly write in matrix operation form is the grad1 calculation. While I was able to try multiple values for the grad1 with all sort of different forms of transposes in many different orders the result would not decrease get to a low enough value. The lowest error rate that the fully matrix operation form got on Julia is 14%, this is double the value that the looping value was able to reach. The total time and space that the linear classification network takes is 225.511551 seconds (313.03 M allocations: 324.780 GiB, 9.68% gc time). The 2 layer Neural Network takes 6+ hours to run and the size of the network is significantly bigger.

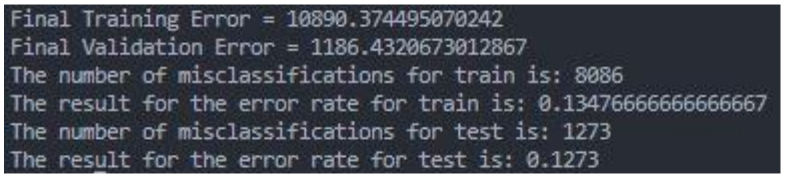
Appendix:

Initial Linear Classifier misclassifications and error rate for both the training data and test data:



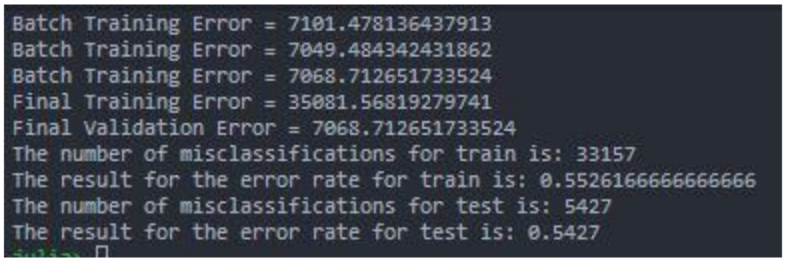
```
Final Training Error = 14154.226526726214
Final Validation Error = 1557.5808461010529
The number of misclassifications for train is: 10644
The result for the error rate for train is: 0.1774
The number of misclassifications for test is: 1637
The result for the error rate for test is: 0.1637
```

Final Linear Classifier misclassification and error rate over the test and train data:



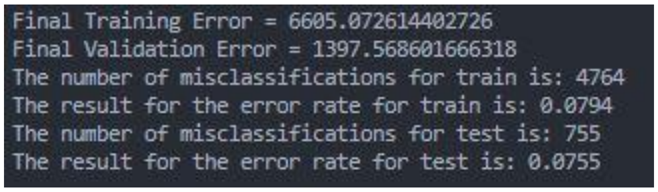
```
Final Training Error = 10890.374495070242
Final Validation Error = 1186.4320673012867
The number of misclassifications for train is: 8086
The result for the error rate for train is: 0.13476666666666667
The number of misclassifications for test is: 1273
The result for the error rate for test is: 0.1273
```

Initial 2-layer misclassifications and error rate:



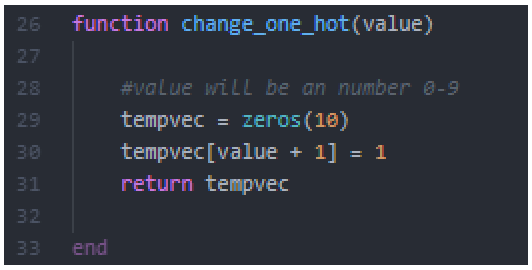
```
Batch Training Error = 7101.478136437913
Batch Training Error = 7049.484342431862
Batch Training Error = 7068.712651733524
Final Training Error = 35081.56819279741
Final Validation Error = 7068.712651733524
The number of misclassifications for train is: 33157
The result for the error rate for train is: 0.5526166666666666
The number of misclassifications for test is: 5427
The result for the error rate for test is: 0.5427
julia>
```

Final 2 -layer misclassifications and error rate:



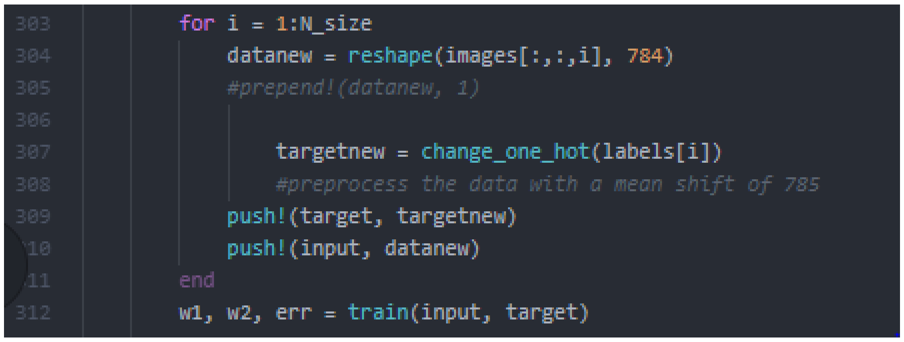
```
Final Training Error = 6605.072614402726
Final Validation Error = 1397.568601666318
The number of misclassifications for train is: 4764
The result for the error rate for train is: 0.0794
The number of misclassifications for test is: 755
The result for the error rate for test is: 0.0755
```

The one hot encoding function:



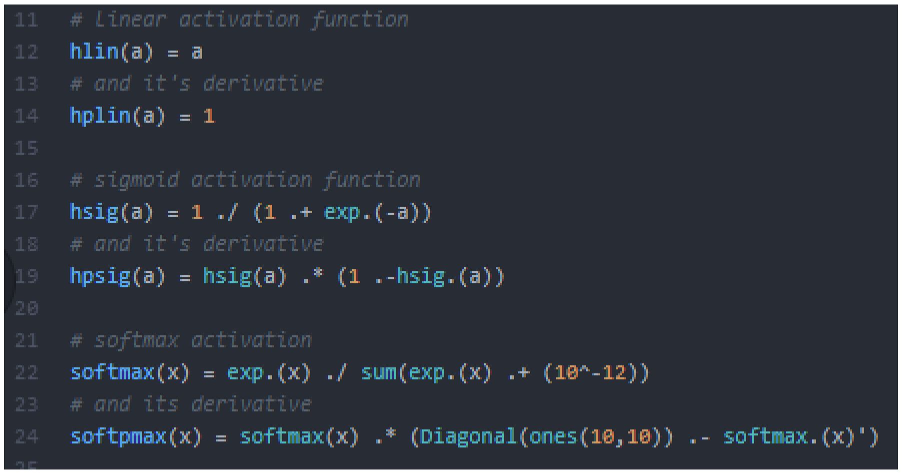
```julia
function change_one_hot(value)

    #value will be an number 0-9
    tempvec = zeros(10)
    tempvec[value + 1] = 1
    return tempvec

end
```

Demo Snip:



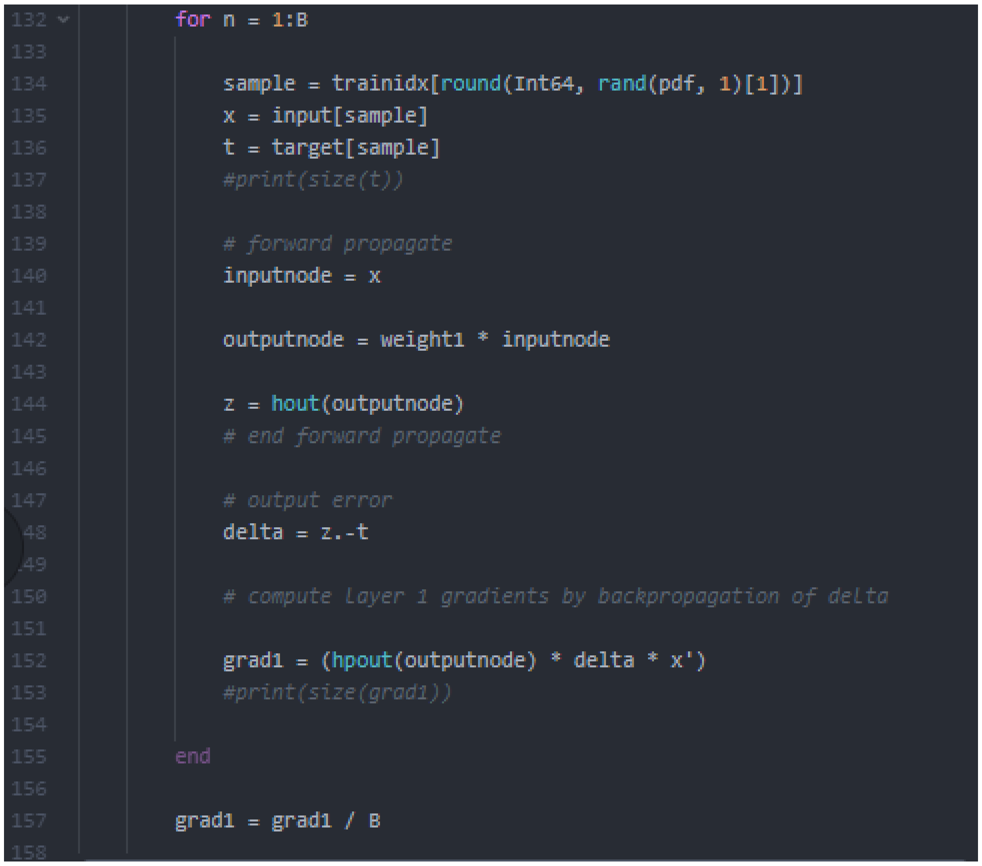
```julia
        for i = 1:N_size
            datanew = reshape(images[:,:,i], 784)
            #prepend!(datanew, 1)

                targetnew = change_one_hot(labels[i])
                #preprocess the data with a mean shift of 785
            push!(target, targetnew)
            push!(input, datanew)
        end
        w1, w2, err = train(input, target)
```

Functions that could be used in both models:



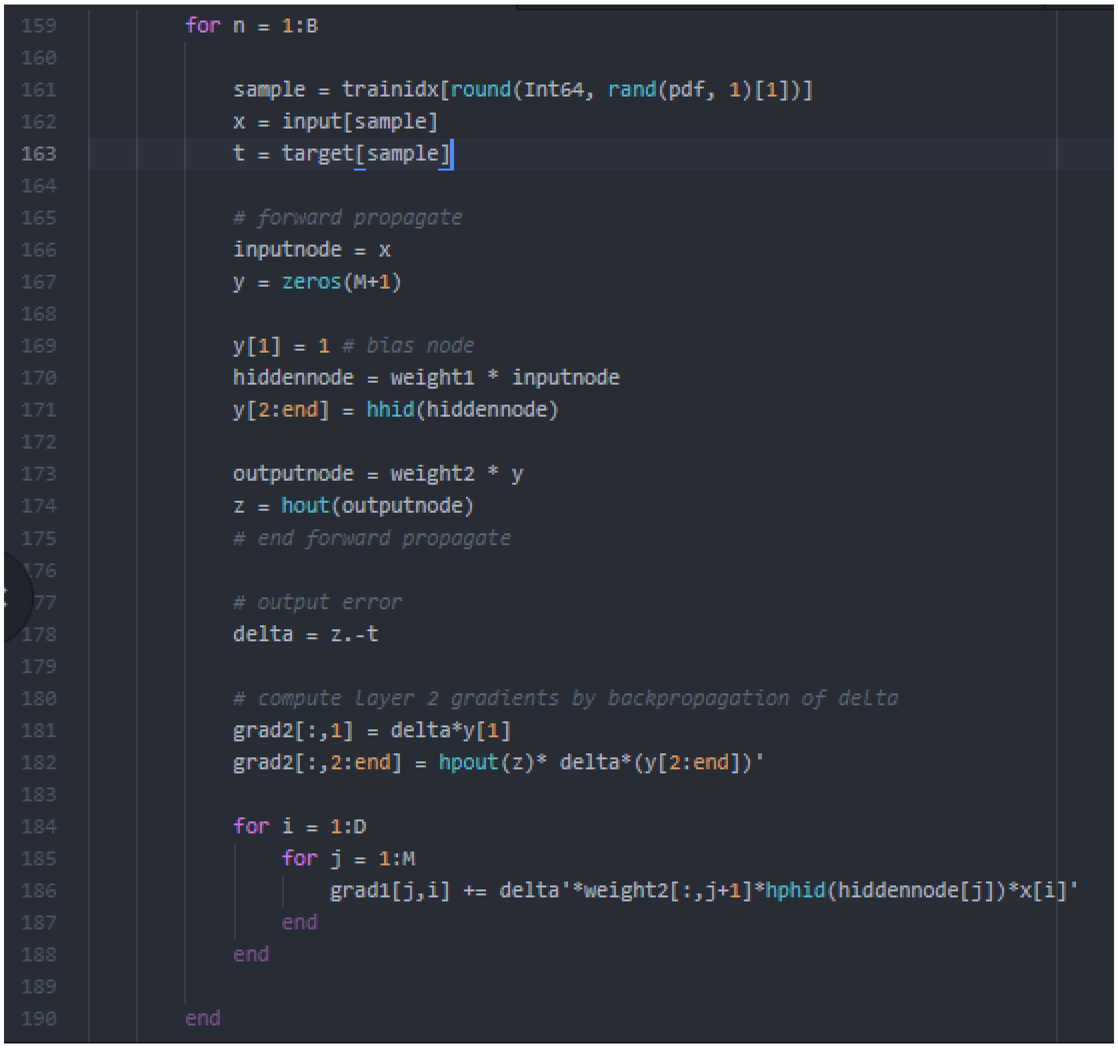
```julia
# Linear activation function
hlin(a) = a
# and it's derivative
hplin(a) = 1

# sigmoid activation function
hsig(a) = 1 ./ (1 .+ exp.(-a))
# and it's derivative
hpsig(a) = hsig(a) .* (1 .-hsig.(a))

# softmax activation
softmax(x) = exp.(x) ./ sum(exp.(x) .+ (10^-12))
# and its derivative
softpmax(x) = softmax(x) .* (Diagonal(ones(10,10)) .- softmax.(x)')
```

Linear Classifier:



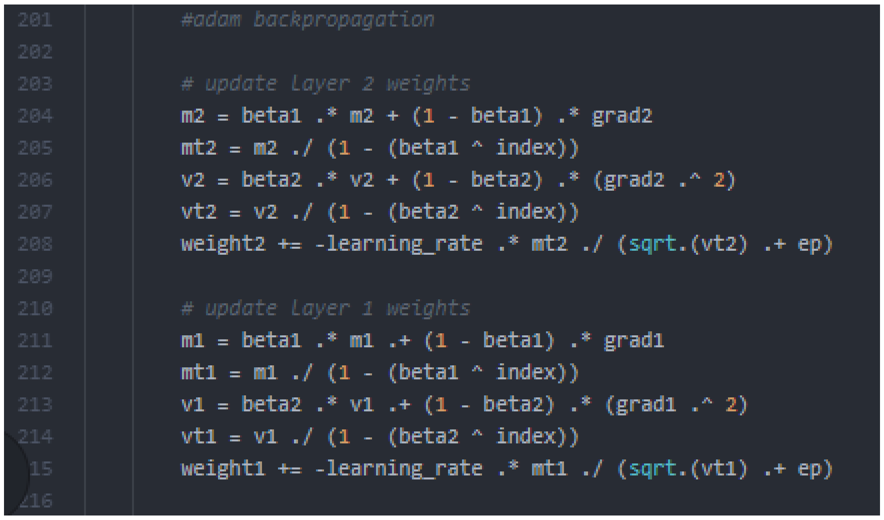
```
    for n = 1:B

        sample = trainidx[round(Int64, rand(pdf, 1)[1])]
        x = input[sample]
        t = target[sample]
        #print(size(t))

        # forward propagate
        inputnode = x

        outputnode = weight1 * inputnode

        z = hout(outputnode)
        # end forward propagate

        # output error
        delta = z.-t

        # compute layer 1 gradients by backpropagation of delta

        grad1 = (hpout(outputnode) * delta * x')
        #print(size(grad1))

    end

    grad1 = grad1 / B
```

2 Layer Neural Network:



```
    for n = 1:B

        sample = trainidx[round(Int64, rand(pdf, 1)[1])]
        x = input[sample]
        t = target[sample]

        # forward propagate
        inputnode = x
        y = zeros(M+1)

        y[1] = 1 # bias node
        hiddennode = weight1 * inputnode
        y[2:end] = hhid(hiddennode)

        outputnode = weight2 * y
        z = hout(outputnode)
        # end forward propagate

        # output error
        delta = z.-t

        # compute layer 2 gradients by backpropagation of delta
        grad2[:,1] = delta*y[1]
        grad2[:,2:end] = hpout(z)* delta*(y[2:end])'

        for i = 1:D
            for j = 1:M
                grad1[j,i] += delta'*weight2[:,j+1]*hphid(hiddennode[j])*x[i]'
            end
        end

    end
```

Adam Backprop:



```
        #adam backpropagation

        # update Layer 2 weights
        m2 = beta1 .* m2 + (1 - beta1) .* grad2
        mt2 = m2 ./ (1 - (beta1 ^ index))
        v2 = beta2 .* v2 + (1 - beta2) .* (grad2 .^ 2)
        vt2 = v2 ./ (1 - (beta2 ^ index))
        weight2 += -learning_rate .* mt2 ./ (sqrt.(vt2) .+ ep)

        # update Layer 1 weights
        m1 = beta1 .* m1 .+ (1 - beta1) .* grad1
        mt1 = m1 ./ (1 - (beta1 ^ index))
        v1 = beta2 .* v1 .+ (1 - beta2) .* (grad1 .^ 2)
        vt1 = v1 ./ (1 - (beta2 ^ index))
        weight1 += -learning_rate .* mt1 ./ (sqrt.(vt1) .+ ep)

```

References:

[1] "Linear Classifier," Easy TensorFlow. [Online]. Available: https://www.easy-tensorflow.com/tf-tutorials/linear-models/linear-classifier. [Accessed: 6-Dec-2019].

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient Based Learning Applied to Document Recognition," Intelligent Signal Processing, Nov. 1998.

[3] Cs231n.github.io. (2019). CS231n Convolutional Neural Networks for Visual Recognition. [online] Available at: http://cs231n.github.io/neural-networks-3/ [Accessed 8 Dec. 2019].

[4] Kingma, D. and Ba, J. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. [online] Arxiv.org. Available at: https://arxiv.org/pdf/1412.6980.pdf [Accessed 4 Dec. 2019].