Homework 9 Extra Credit

Generalized Iris Classifier

I used the scikit-learn standard machine learning library to generalize the classification model we worked on in HW9 to include all four input dimensions and all three target classes.

I have outlined my general process below.

First I loaded the iris data and added class labels to encode the target vector.

```
"""
Download data as a pandas DataFrame object
"""
import pandas as pd

data = pd.read_csv("iris.csv")
encodings = {
    "Setosa": 1,
    "Versicolor": 2,
    "Virginica": 3
}
data["class_label"] = [encodings[x] for x in data["species"]]
data.head()
```

✓ 0.3s

	sepal.length	sepal.width	petal.length	petal.width	species	class_label
0	5.1	3.5	1.4	0.2	Setosa	1
1	4.9	3.0	1.4	0.2	Setosa	1
2	4.7	3.2	1.3	0.2	Setosa	1
3	4.6	3.1	1.5	0.2	Setosa	1
4	5.0	3.6	1.4	0.2	Setosa	1

From here, I was able to extract input vectors (X), and ground truth (y) to feed into the scikit-learn neural network.

```
"""
Specifying input vectors (X) and prediction target (y)
"""
# Specifying the prediction target
target = ["class_label"]
y = data[target]
print(y.head())

# Specifying X
attrs = ["sepal.length", "sepal.width", "petal.length", "petal.width"]
X = data[attrs]
print(X.head())
```

✓ 0.0s

Next, I split the dataset into train and test subsets. This was done in order to leverage cross validation and prevent overfitting, which we explored in class. Luckily, scikit-learn has a module that automates this process.

```
# Splitting data into train and test sets for cross validation as discussed in class
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Now that I had a training dataset, I simply imported a general MLP classifier and used it to fit the training data. I was honestly amazed at how easy the process was, especially comparing it to the manual implementation from HW9. I learned that I could parameterize the model with a batch size, learning rate, various activation functions, and even different solvers. The solver that we looked at in class was the default gradient descent algorithm. I learned that the default solver used by scikit-learn's MLPClassifier model is an algorithm called 'adam', which utilizes stochastic gradient descent.

After researching the two, I learned that gradient descent optimizes parameters by iterating through every single training pattern for a single update by step size t. This can be costly if we have a large amount of training data, but produces a good minimization since all input vectors are taken into account. On the other hand, stochastic gradient descent optimizes parameters using a subset of input data at each iteration. This means it converges faster overall, but may not be as robust as standard gradient descent.

```
11  # Specifying the model (from sklearn docs)
12  model = MLPClassifier(hidden_layer_sizes=(100, ))
13
14  # Fitting the model to the data
15  model.fit(X_train, y_train)
16
17  # Making predictions on the test set
18  y_pred = model.predict(X_test)
```

For this classification model, I used the default 'adam' solver because I wanted to prioritize performance, but I tested other solvers as well. There didn't seem to be a huge drop off in performance, likely because the iris dataset only contains 150 patterns. I also specified 1 hidden layer with 100 neurons. After tons of experimentation, it was clear that just one hidden layer did the trick just fine, and that introducing more hidden layers was

probably overfitting the model, leading to a performance dropoff on the test set.

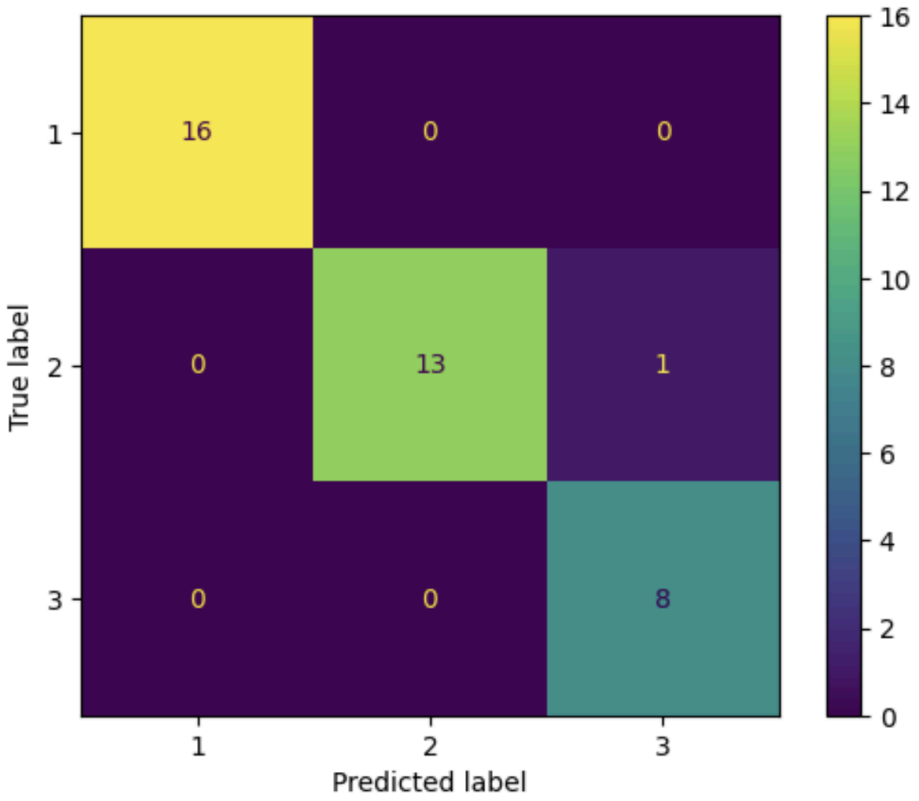
From there, I stored my predicted values in y_pred, and was able to leverage toolbox methods to evaluate the model's accuracy over various iterations, and the frequency of misclassification.
To evaluate the model performance I used mean absolute error which I learned is the standard for simple models. The error rate of my model is reported below:

```
# Computing loss
error = mean_absolute_error(y_test, y_pred)
print("MAE", error)

# Plotting the confusion matrix for the MLPClassifier
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=model.classes_)
disp.plot()
plt.show()
```

```
MAE 0.02631578947368421
```

In addition, I was also able to plot a confusion matrix, examining misclassifications across the 3 target labels. Here is the result:



The way we can interpret the confusion matrix is that diagonal values represent accurate predictions and non-diagonal values represent misclassifications. As can be seen, the validation (test) set in this training instance contained 16 members of class 1, 13 from class 2, and 8 from class 3. Out of all predictions in y_pred, there was only 1 misclassification by the model, which was mistaking a class 2 for a class 3. This shows robust model performance. Overall, this was a good learning process. I enjoyed learning about how models are defined at a high level, and the level of customizability that comes with it.

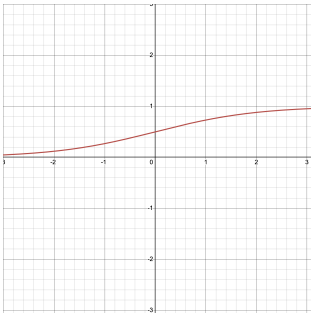
Exploring Advanced Topics

Non-linearity

Non-linearity in neural networks is a way to apply non-linear transformations to network layers to capture complex relationships in the data, and learn from them. In class, we explored in depth the logistic (sigmoid) activation function as the primary vessel of non-linearity in our architecture. However, there are many other activation functions with unique characteristics. I have detailed a few of them below.

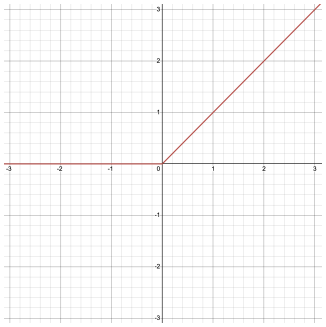
Sigmoid

$$f(x) = \frac{1}{1+e^{-x}}$$



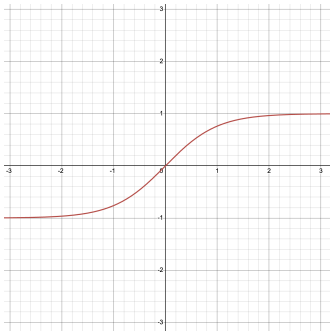
ReLU

$$f(x) = \max(0, x)$$



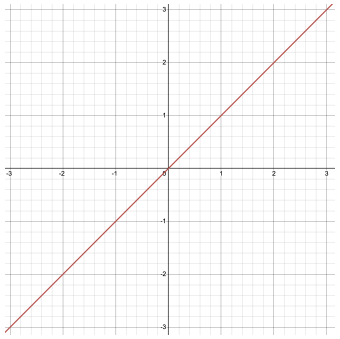
tanh

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$



identity

$$f(x) = x$$



While activation functions are not the only way to introduce non-linearity to a neural network (convolutions, pooling, etc.), they are an essential part of the network's architecture and can have a significant impact on the way the network learns.

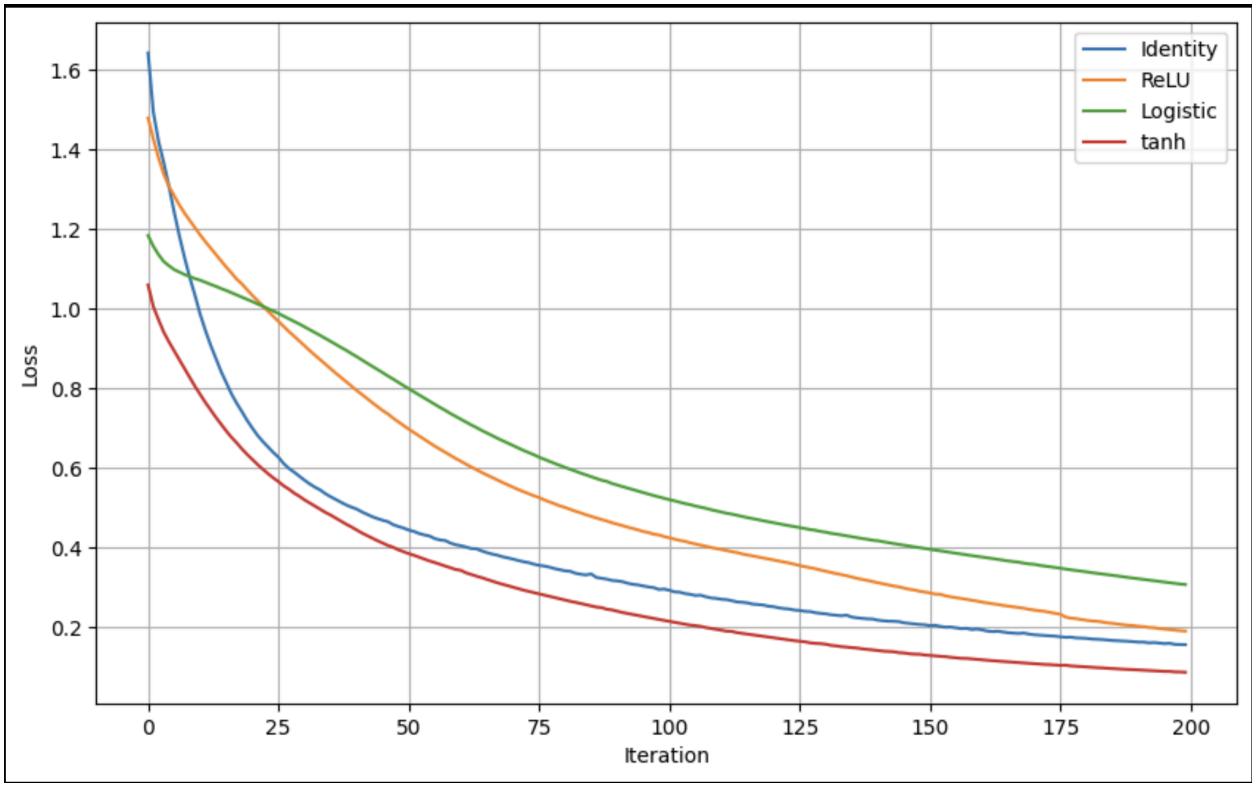
Below, I will provide a tutorial exploring just that, and also conduct a small experiment on activation functions which evaluated

their performance on working with perturbed, or adversarial, data (data which has been adjusted to be out of the neural network's learning range.

Let's begin with the standard performance comparison. To make this easy, I created a simple IrisTrainer class which fits different training subsets of the overall iris datasets according to some specifications. In this case, we will look at the learning curves of the model, using different activation functions.

```
# Create the trainer
identity_trainer = IrisMLPTrainer(data, "identity")
relu_trainer = IrisMLPTrainer(data, "relu")
logistic_trainer = IrisMLPTrainer(data, "logistic")
tanh_trainer = IrisMLPTrainer(data, "tanh")
```

Below, I have attached a standard comparison of the learning curves of a model with 1 hidden layer with 30 neurons.



We can see that overall, each model has a similar convergence rate, but the model using tanh activation outperforms the rest over 200 iterations.

Despite that, the learning curve does not give us any indication of the model's final performance since it only tracks over 200 iterations. To evaluate that, I used a similar mean absolute error method as detailed above to check the accuracy of each model regardless of their convergence speed. I used the following code to do so:

```
class ErrorTable:
    def __init__(self, trainers, trainer_names):
        ## Extracting errors
        errors = [trainer.accuracy() for trainer in trainers]

        # Creating the table with variable names as a column
        self.table = pd.DataFrame({
            "Trainer Variable Name": trainer_names,
            "Error (MAE)": errors
        })
```

Using this simple class, I was able to produce the following table for the models we just created above.

	Trainer Variable Name	Error (MAE)
0	Identity	0.026316
1	ReLU	0.026316
2	Logistic	0.000000
3	tanh	0.052632

Even though the tanh activation had the best convergence speed, it produced the worst accuracy out of all four activation functions, with the logistic function creating a perfect model.

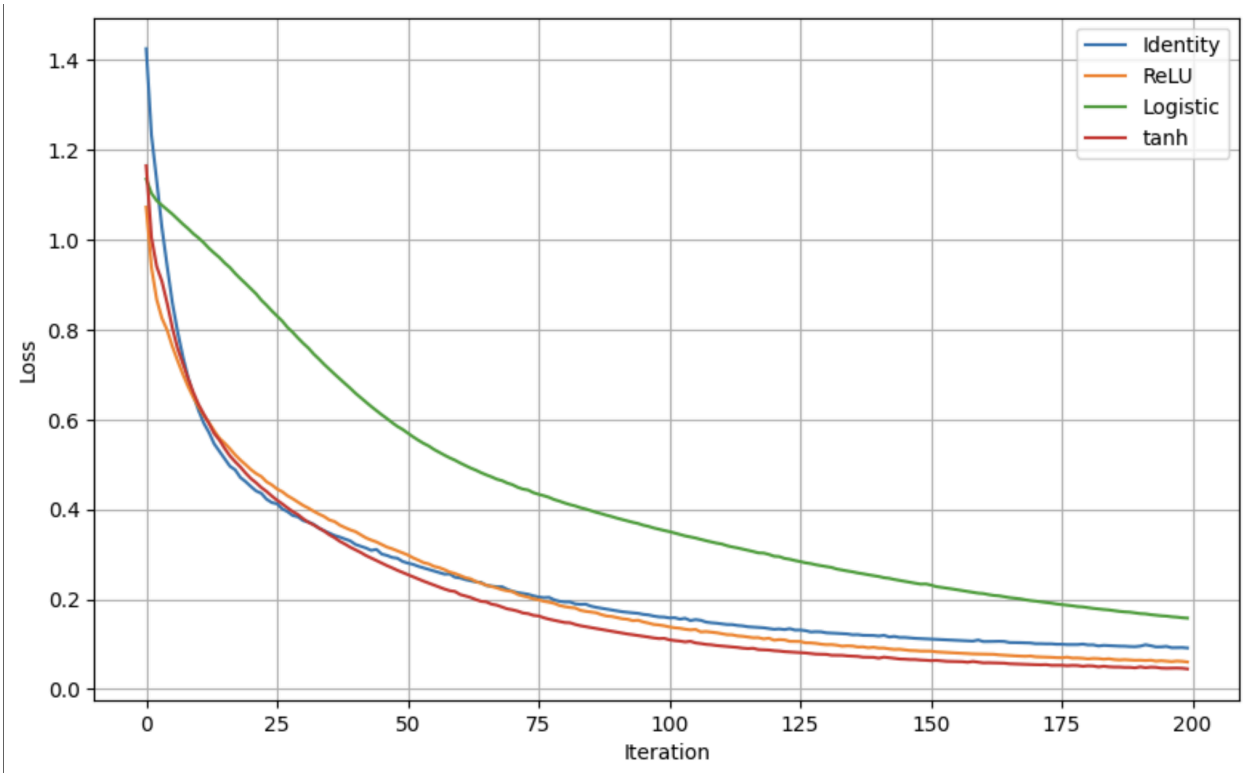
To understand why this is the case, we can examine the tanh function's anatomy.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Essentially, the key to understanding tanh is understanding its output values. tanh has a range from -1 to 1, meaning gradient based optimization methods converge faster. This happens because being centered around 0 means gradients can be positive or negative, which means it takes less weight updates to find a local or global minimum loss. This is especially true because if the average activation value is close to 0, the gradient maps to outputs symmetrically, which is fundamentally different from other activation functions which are asymmetric.

At the same time, we see a performance dropoff because the neuron outputs can be saturated near -1 or 1, and is also subject to the vanishing gradient problem, which occurs when the weights of earlier layers do not update significantly due to very small (zero-centered) gradients. In this case, there was only 1 hidden layer, but we will explore what happens with more hidden layers shortly.

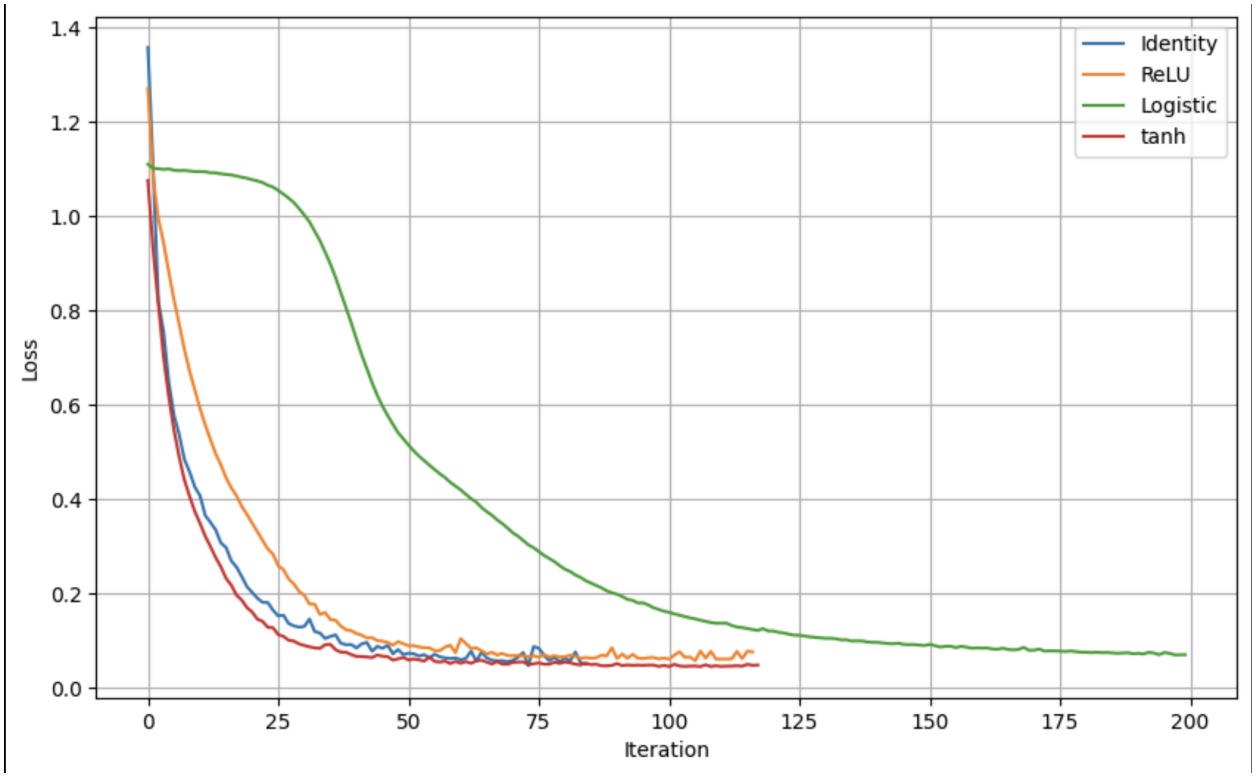
Increasing the number of neurons in the hidden layer to 100 yields the following curve and errors:



	Trainer Variable Name	Error (MAE)
0	Identity	0.000000
1	ReLU	0.052632
2	Logistic	0.026316
3	tanh	0.078947

As we saw in the first case, tanh has the fastest convergence, but the worst accuracy.

We can now confidently formulate a hypothesis that tanh has fast convergence but a high error rate. To test this theory, let us now increase the number of hidden layers from 1 to 3 (50 units each).



	Trainer Variable Name	Error (MAE)
0	Identity	0.000000
1	ReLU	0.000000
2	Logistic	0.078947
3	tanh	0.105263

This time, the tanh function behaved as we expected. However, we see significant deviation of the behavior of the logistic function from the others.

$$f(x) = \frac{1}{1+e^{-x}}$$

Some potential explanations for this are the bias of the logistic function toward positive values. Since the logistic function outputs values between 0 and 1, the gradient will always be positive, meaning there can be gradient updates biased toward the same direction. Additionally, this graph illustrates the vanishing gradient problem quite nicely. Since the model now backpropagates through multiple layers, saturated regions of the logistic function (near 1) can lead to very small derivatives, meaning in earlier layers for each pass, the gradient becomes nearly 0, leading to very slow convergence. That is precisely what is illustrated above between iterations 0 and 25.

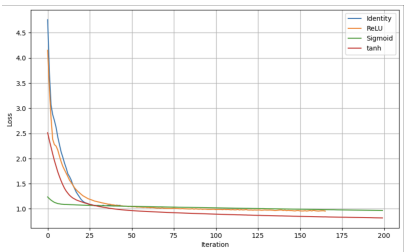
Throughout this experimentation, the identity and ReLU activation functions have remained fairly consistent. But let us see if that is still the case after we perturb input data.

In order to do this, I simply created a perturbed parameter in my IrisTrainer class. If set to true, it applies a random modification to each value in the training dataset.

```
# Creating purturbations in X
if purturbed:
    self.X_purturbed = self.X_train.copy()
    for i, row in self.X_purturbed.iterrows():
        self.X_purturbed.loc[i] = row + np.random.randint(low=0, high=30, size=row.shape)
```

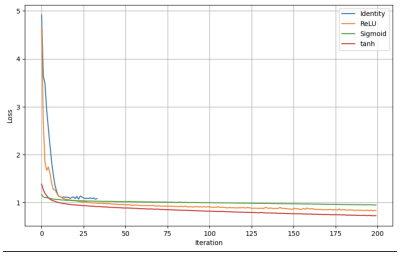
From here, we can conduct the same experiments on learning curves as above, and examine the final accuracy of each model.

1 layer, 30 neurons



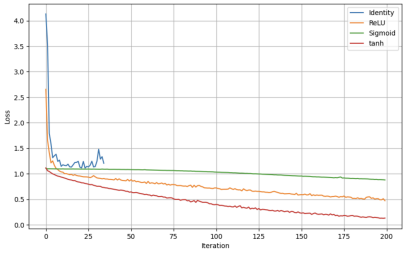
	Trainer Variable Name	Error (MAE)
0	Identity	0.342105
1	ReLU	0.605263
2	Logistic	1.131579
3	tanh	0.842105

1 layer, 100 neurons



	Trainer Variable Name	Error (MAE)
0	Identity	0.342105
1	ReLU	0.789474
2	Logistic	1.131579
3	tanh	1.131579

3 layers, 50 neurons



	Trainer Variable Name	Error (MAE)
0	Identity	0.684211
1	ReLU	0.842105
2	Logistic	0.763158
3	tanh	1.263158

In these experiments, the key observation is that the ReLU (rectified linear unit) activation function is the most consistent in terms of convergence speed and accuracy. The others either have massive differences in convergence of learning curves, or their accuracy is thrown off by network architecture. This could be happening because the function is essentially a piecewise linear function meaning perturbations wouldn't have a significant impact on the model as long as it is in the positive x direction. One limitation of my method here is that for each input in the training set, I add a random perturbation which means it is impossible for a value to be negative. I will not switch the sign of this operation and try again. There should be a significant difference in the ReLU model's performance.

```
self.X_purturbed.loc[i] = row - np.random.randint(low=0, high=30, size=row.shape)
```

	Trainer Variable Name	Error (MAE)
0	Identity	0.736842
1	ReLU	0.973684
2	Logistic	0.842105
3	tanh	0.842105

As can be seen, the ReLU model now has the worst performance out of the four. To provide more clarification about why this happens, the ReLU propagates discontinuities in input data (data which crosses y=0) and makes the gradient of ReLU models very sensitive to data around the decision boundary. This happens because $x > 0$ does not impact the ReLU gradient but $x \leq 0$ effectively shuts off the neuron, artificially inflating certain weights. This is what decreases model performance, which can be observed through perturbed data.

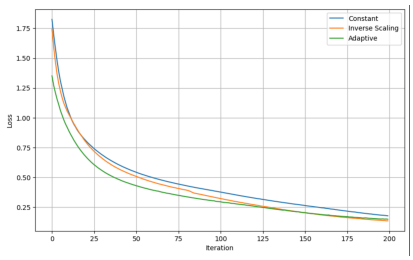
Learning Rate Algorithms

The learning rate has a profound impact on the performance of a neural network. To define what it is exactly, the learning rate (also known as step size) is a hyperparameter which controls how fast the

model can learn. In class, we learned about it in the context of gradient descent as a constant scalar value which determines how much to change a parameter in the negative gradient direction. However, the learning rate doesn't always need to be a constant value. There are actually several algorithms designed to optimize the learning rate at each iteration of backpropagation. One of them is line search, which takes the update formula as a function, and projects it into two-dimentional space. Finding the minimum of that function would give the optimal stepsize. However, computing derivatives is expensive so this approach is not widely used in production. Therefore, it's more common to simply define a constant value in conjunction with a batch learning approach which we will explore in more depth shortly.

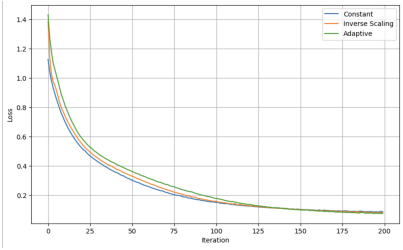
To begin, I conducted a simple analysis of the learning curves of IrisTrainers with different learning rate algorithms. Below are the results.

1 layer, 30 neurons



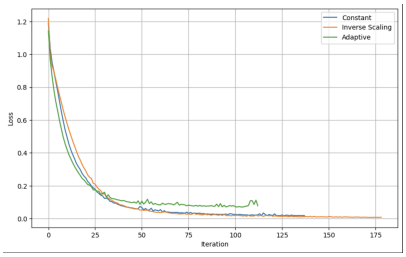
	Trainer Variable Name	Error (MAE)
0	Constant	0.000000
1	Inverse Scaling	0.026316
2	Adaptive	0.026316

1 layer, 100 neurons



	Trainer Variable Name	Error (MAE)
0	Constant	0.026316
1	Inverse Scaling	0.078947
2	Adaptive	0.000000

3 layers, 50 neurons



	Trainer Variable Name	Error (MAE)
0	Constant	0.026316
1	Inverse Scaling	0.078947
2	Adaptive	0.000000

In these experiments, I examined three different learning rate algorithms.

The first is a constant function given by specified value.

$$t(x) = c$$

It defaults to 0.001. This is the standard learning rate algorithm and from my research, I learned that it is usually preferred because of its simplicity and ease of implementation. It can also be adjusted very easily.

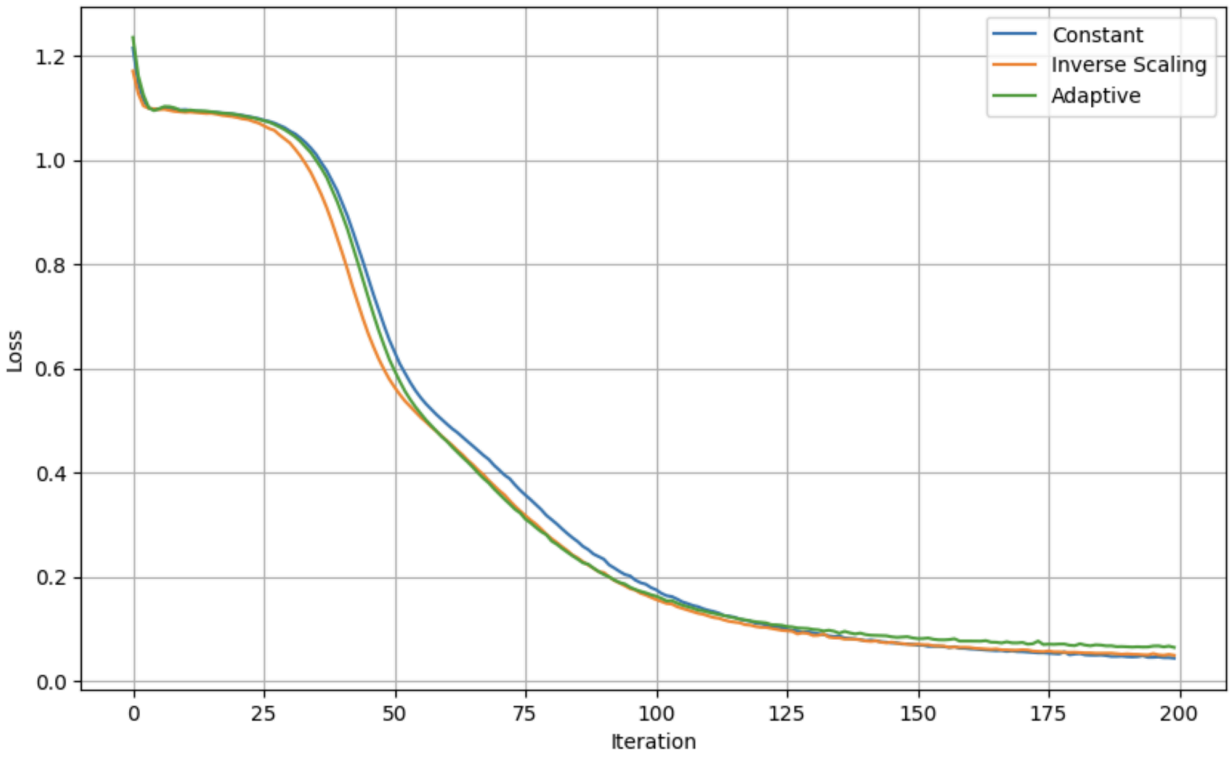
The second is an inverse scaling algorithm which gradually decreases the learning rate at each step using an inverse scaling exponent.

$$t(x) = \frac{initialrate}{(1+decayrate*x)}$$ where x is the current rate.

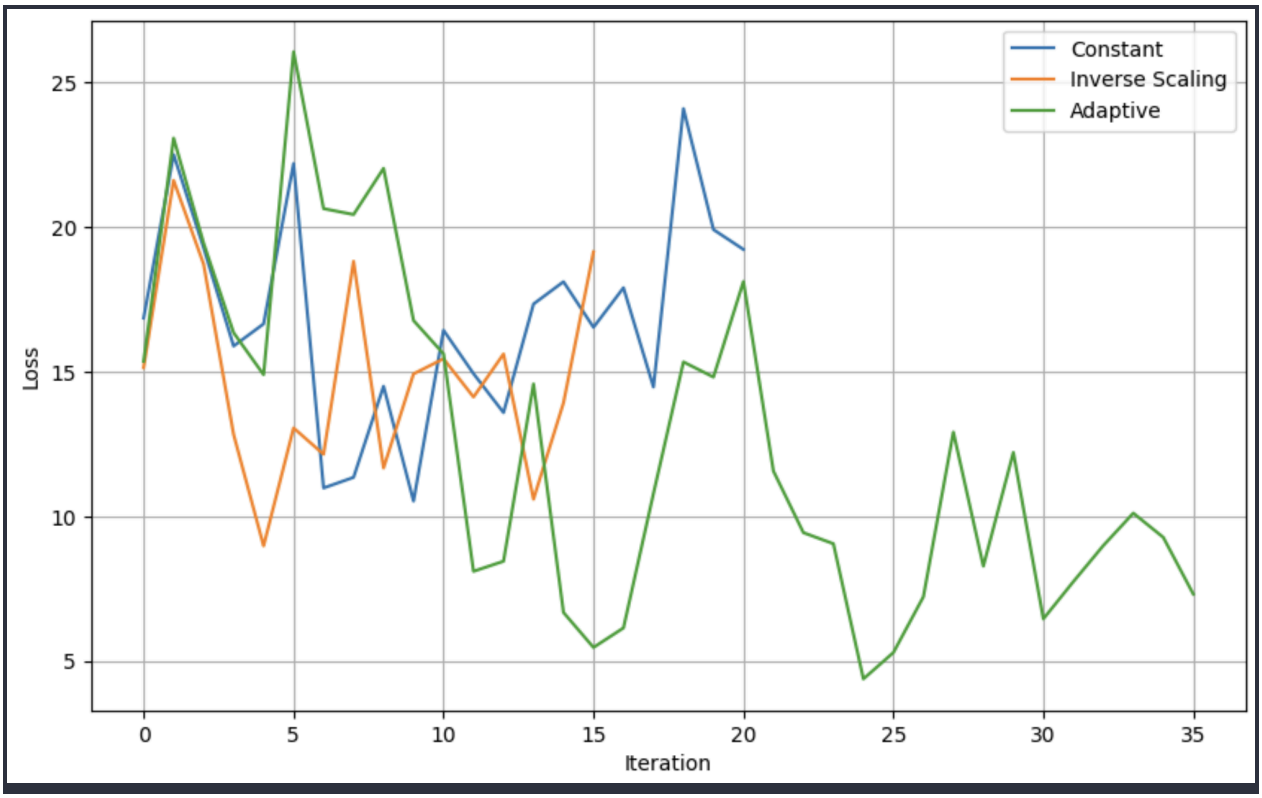
This formulation for the learning rate is advantageous because it promotes stability with large models and also prevents the gradient from jumping too far with the loss function and overshooting the minimum. On the other hand, training can be slow, and this is more susceptible to getting stuck in a local minimum since it scales at an arbitrary decay rate, which can have a trapping effect on the gradient.

Finally, we will examine an adaptive algorithm which keeps the learning rate constant to some x0 as long as the loss function keeps decreasing across the training samples. Then, every time two consecutive iterations do not increase training loss by at least some arbitrary amount, the current learning rate is divided by 5.

In the results above, we can see that the model's performance is not very dependent on learning rate, if we simply change the network architecture. I suspected that maybe it is an activation function issue, so I switched the function from ReLU to logistic to see if there would be a difference. However, as can be seen, there is not a significant difference between each algorithm in the logistic case either.



Regarding the initial value of the learning rate, an interesting result is that the adaptive algorithm outperforms the rest when this value is set to be very high. In previous experiments the initial value is set to 0.001 (default case). For the next experiment, I have set it to 3, which is very high.



The resulting bouncing effect is caused by the model constantly overshooting the minimum value of the loss function in one or another direction, resulting in several update recomputations and a slingshot effect of the learning curve. However, we can observe that the adaptive algorithm, despite being subject to an incredibly high learning rate, is still decreasing loss. This happens because this is the only algorithm which only changes the learning rate in the negative direction. This means that the learning rate can only decrease starting at a high x0 value, meaning it will be the most stable across many iterations when the gradient is constantly overshooting the minimum.

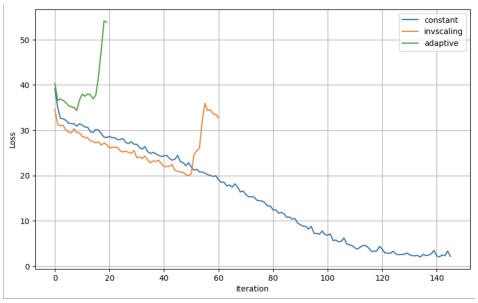
Another hyperparameter which has some interplay with the learning rate is the batch size.

The batch size of a neural network is the number of training samples evaluated by the network in a single forward/backward pass. It is mainly used to optimize gradient descent in the stochastic case as discussed in the beginning of this report.

Below are results of different learning algorithms with different batch sizes:

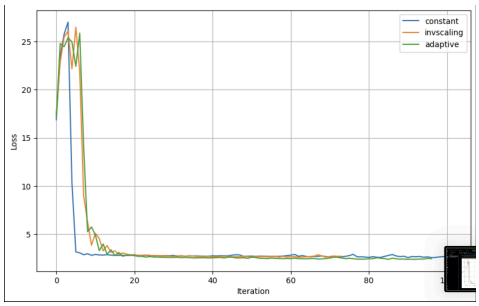
Batch sizes:

(1)



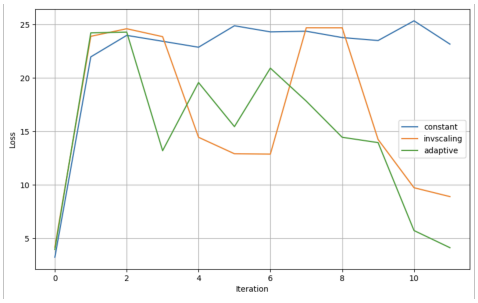
	Trainer Variable Name	Error (MAE)
0	Constant	0.605263
1	Inverse Scaling	1.289474
2	Adaptive	1.105263

(30)



	Trainer Variable Name	Error (MAE)
0	Constant	1.210526
1	Inverse Scaling	0.710526
2	Adaptive	1.236842

(100)



	Trainer Variable Name	Error (MAE)
0	Constant	1.105263
1	Inverse Scaling	0.710526
2	Adaptive	1.131579

We can observe that with a batch size of 1, this is equivalent to a normal gradient descent algorithm, in which case, the constant step size of 0.001 performs the best. It is natural that there is a tradeoff with convergence rate, but that is okay considering the other algorithms diverge. This is likely due to high variance of gradient estimates for the next iterate since there is effectively no batch normalization or aggregation of data. Batch normalization is when each input in a batch is configured to have 0 mean and unit variance, allowing the model to learn not only the best parameter value, but also learn the optimal scale of the value. This is precisely what decreases the model's sensitivity to initial parameter values, which is demonstrated above. Then, in the 100 batch size case, interestingly enough, the inverse scaling algorithm performs the best. This happens because the point of inverse scaling is to reduce overshooting, and when paired with a large batch size which corresponds to large updates to gradients, computation ends up being comparatively efficient.