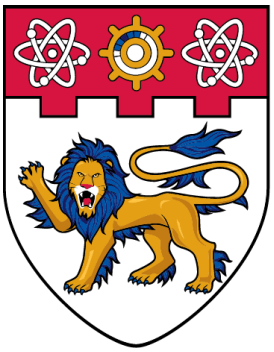


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ3005 Artificial Intelligence Lab 2

Name: Ernest Ang Cheng Han
Matriculation Number: U1921310H
Lab group: TDDP1
Date of Submission: 27/10/2021

Task 1: You are asked to build a three-layer feed-forward neural network to solve the monitoring problem of injection molding machines. Your implementation must be in Pytorch and executable in Google Colab environments. The proportion of training and testing samples is 70:30 where your model must deliver the smallest testing error possible. In that case, you need to select the number of nodes of hidden layers, the number of epochs, the learning rates, the mini-batch size, etc. that lead to the smallest testing error. In this assignment, you have to use the SGD optimizer as exemplified in the lab materials under the mini-batch update fashion. The evaluation metric here is the classification error. No feature selection is allowed here.

Overview:

The data stored within the .mat file is an array of 49 input data points, the first 48 of which are the inputs used to train the model, and the last input is the label of the data point (i.e. the categories of the mold - good cases, weaving cases, and short-forming cases). After extracting the data and formatting it properly, we use the train_test_split method from the sklearn.model_selection module to divide our data into trainset and testset in the 70:30 ratio as specified. To improve the accuracy and performance of our model, we will be validating the model when we are training it, hence we will further divide the trainset into trainset and validationset in a 90:10 ratio.

```
"""
Imported Data
"""

data = scipy.io.loadmat(os.getcwd() + '/OQC.mat')
print("Header information: {info}".format(info = data['__header__']))
data = data['data']
print("Length of Data: {length} data points".format(length = str(len(data))))
```

```
Header information: b'MATLAB 5.0 MAT-file, Platform: PCWIN64, Created on: Sun Mar 29 14:29:37 2020'
Length of Data: 2952 data points
```

```
"""
Each data point has 49 values, and there are 48 input attributes.
Assume first 48 values in array [0, 47] are the input attributes.
Assume last data point in array [48] is the ouput attribute
"""

data[0]
```

```
array([3.90524727e-01, 1.47797582e-01, 1.56398104e-01, 2.55985544e-03,
       5.55555556e-01, 7.77777778e-01, 5.55555556e-01, 7.14285714e-01,
       5.00000000e-01, 5.00000000e-01, 0.00000000e+00, 0.00000000e+00,
       3.50680457e-01, 9.95993193e-01, 7.87486391e-01, 1.00000000e+00,
       4.37500000e-01, 3.75000000e-01, 1.00000000e+00, 3.33333333e-01,
       2.50000000e-01, 7.80247068e-02, 3.06230179e-01, 1.00000000e+00,
       5.00000000e-01, 6.66664707e-02, 6.93769821e-01, 6.14525149e-01,
       3.85474851e-01, 9.21936743e-01, 1.00000000e+00, 3.34645780e-02,
       4.99809148e-02, 2.33802826e-01, 1.00000000e+00, 9.80563469e-01,
       4.67564870e-01, 6.16027345e-01, 7.02833854e-02, 7.19864153e-01,
       3.72549020e-02, 3.29627376e-01, 1.00000000e+00, 3.76520200e-05,
       1.25000000e-01, 7.89695478e-02, 7.96252877e-02, 9.22469268e-01,
       2.00000000e+00])
```

```
"""
From documentation, there is indeed 1008 good cases, 1074 weaving cases, and 870 short-forming cases
"""

categories = np.array(list(map(extractcategoryfromdatapoint, data)))
int_categories = categories.astype(int)
np.bincount(int_categories)
```

```
array([1008, 1074,  870])
```

```
"""
Train test split function to split data randomly into 70:30 ratio
"""

train, test = train_test_split(data, test_size=0.3, random_state=SEED)
print("length of training data: " + str(len(train)))
print("length of testing data: " + str(len(test)))
print("total length of data: " + str(len(test)+ len(train)))
```

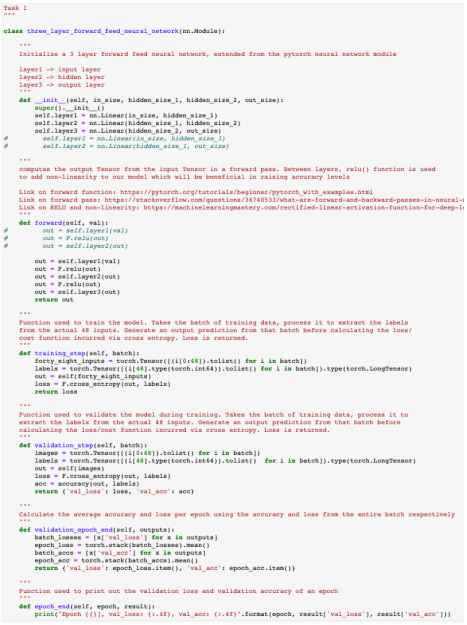
```
length of training data: 2066
length of testing data: 886
total length of data: 2952
```

```
"""
Further split the training data into training data and validation data in 90:10 using same train test split function
"""

train, validation = train_test_split(train, test_size=0.1, random_state=SEED)
print("length of training data: " + str(len(train)))
print("length of validation data: " + str(len(validation)))
print("total length of training data: " + str(len(validation)+ len(train)))
```

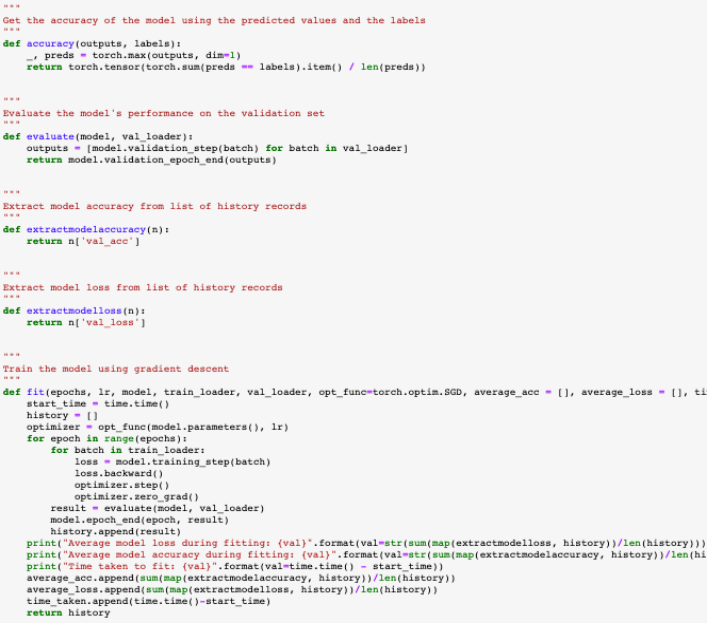
```
length of training data: 1859
length of validation data: 207
total length of training data: 2066
```

We then declare a three_layer_forward_feed_neural_network class extended from the Pytorch Neural Network Module with three layers: layer1, layer2, and layer3. We then configure and override the forward function from the base module. The forward function computes the output tensors from the input tensors, it calculates and creates an output which will be the predicted category of the mold using the layers of the neural network. In between the layers, we also add non-linearity to our entire model by adding Rectified Linear Units (ReLU). ReLU is a popular non-linear activation function which allows models to be created with complicated mappings between the input tensors and output tensors. It is useful especially when dealing with datasets which behave in a non-linear fashion such as ours. There are many other nonlinear activation functions such as Sigmoid, and Hyperbolic Tangent which helps to add non-linearity to our model but after much experimentation, ReLU gives the best performance and output. Additionally, ReLU is computationally efficient given that its output value on the negative x-axis and y-axis is always 0 and hence the neural network will run and compute values much faster.

To actually train and fit the model, we define a fit function which takes the model, the learning rate, the number of epochs, the training data and validation data, the optimizer, and the learning rate of the optimizer. Epochs refers to the number of passes of the entire training data set. Given that our training and validation data is very huge, we will be training and validating the model in batches using the Pytorch DataLoader module. In this case, one epoch would refer to iterating through every batch of training data generated by the DataLoader module once. For every batch of training data loaded by the DataLoader module, you will train the model and return the loss/cost function which is the summation of the differences between the predicted values and the actual values. You then apply the backward() function which gets the derivative (gradient) of the loss with respect to its parameters via backpropagation. After getting the gradient, we apply the step() function from our optimizer which will iterate through every input tensor it is supposed to update and update them using the stored gradient value. The updating of input tensors throughout the whole model is very much dependent on the optimizer used and the learning rate of the optimizer. At the end, we apply the zero_grad() function which clears the current gradient value to prevent unnecessary accumulation of gradient values. After training the model by batches within still the same epoch, we will validate the model using the validation data which is also batched via the Pytorch DataLoader module, before printing the validation accuracies and validation losses made by the model.



```
Task 1
"""

class three_layer_forward_feed_neural_network(nn.Module):

    """
    Initialize a 3 layer forward feed neural network, extended from the pytorch neural network module

    layer1 -> input layer
    layer2 -> hidden layer
    layer3 -> output layer
    """
    def __init__(self, in_size, hidden_size_1, hidden_size_2, out_size):
        super().__init__()
        self.layer1 = nn.Linear(in_size, hidden_size_1)
        self.layer2 = nn.Linear(hidden_size_1, hidden_size_2)
        self.layer3 = nn.Linear(hidden_size_2, out_size)
#         self.layer1 = nn.Linear(in_size, hidden_size_1)
#         self.layer2 = nn.Linear(hidden_size_1, out_size)

    """
    computes the output Tensor from the input Tensor in a forward pass. Between layers, relu() function is used
    to add non-linearity to our model which will be beneficial in raising accuracy levels

    Link on forward function: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html
    Link on forward pass: https://stackoverflow.com/questions/36740533/what-are-forward-and-backward-passes-in-neural-n
    Link on RELU and non-linearity: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-le
    """
    def forward(self, val):
#         out = self.layer1(val)
#         out = F.relu(out)
#         out = self.layer2(out)

        out = self.layer1(val)
        out = F.relu(out)
        out = self.layer2(out)
        out = F.relu(out)
        out = self.layer3(out)
        return out

    """
    Function used to train the model. Takes the batch of training data, process it to extract the labels
    from the actual 48 inputs. Generate an output prediction from that batch before calculating the loss/
    cost function incurred via cross entropy. Loss is returned.
    """
    def training_step(self, batch):
        forty_eight_inputs = torch.Tensor([i[0:48].tolist() for i in batch])
        labels = torch.Tensor([i[48].type(torch.int64).tolist() for i in batch]).type(torch.LongTensor)
        out = self(forty_eight_inputs)
        loss = F.cross_entropy(out, labels)
        return loss

    """
    Function used to validate the model during training. Takes the batch of training data, process it to
    extract the labels from the actual 48 inputs. Generate an output prediction from that batch before
    calculating the loss/cost function incurred via cross entropy. Loss is returned.
    """
    def validation_step(self, batch):
        images = torch.Tensor([i[0:48].tolist() for i in batch])
        labels = torch.Tensor([i[48].type(torch.int64).tolist()  for i in batch]).type(torch.LongTensor)
        out = self(images)
        loss = F.cross_entropy(out, labels)
        acc = accuracy(out, labels)
        return {'val_loss': loss, 'val_acc': acc}

    """
    Calculate the average accuracy and loss per epoch using the accuracy and loss from the entire batch respectively
    """
    def validation_epoch_end(self, outputs):
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return {'val_loss': epoch_loss.item(), 'val_acc': epoch_acc.item()}

    """
    Function used to print out the validation loss and validation accuracy of an epoch
    """
    def epoch_end(self, epoch, result):
        print("Epoch [{}], val_loss: {:.4f}, val_acc: {:.4f}".format(epoch, result['val_loss'], result['val_acc']))
```



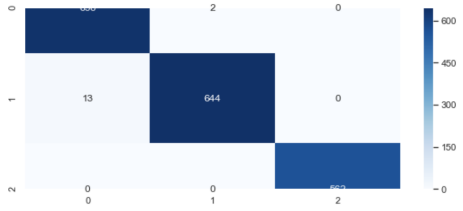
```
"""
Get the accuracy of the model using the predicted values and the labels
"""
def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim=1)
    return torch.tensor(torch.sum(preds == labels).item() / len(preds))


"""
Evaluate the model's performance on the validation set
"""
def evaluate(model, val_loader):
    outputs = [model.validation_step(batch) for batch in val_loader]
    return model.validation_epoch_end(outputs)


"""
Extract model accuracy from list of history records
"""
def extractmodelaccuracy(n):
    return n['val_acc']


"""
Extract model loss from list of history records
"""
def extractmodelloss(n):
    return n['val_loss']


"""
Train the model using gradient descent
"""
def fit(epochs, lr, model, train_loader, val_loader, opt_func=torch.optim.SGD, average_acc = [], average_loss = [], ti
    start_time = time.time()
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()
        result = evaluate(model, val_loader)
        model.epoch_end(epoch, result)
        history.append(result)
    print("Average model loss during fitting: {val}".format(val=str(sum(map(extractmodelloss, history))/len(history)))
    print("Average model accuracy during fitting: {val}".format(val=str(sum(map(extractmodelaccuracy, history))/len(hi
    print("Time taken to fit: {val}".format(val=time.time() - start_time))
    average_acc.append(sum(map(extractmodelaccuracy, history))/len(history))
    average_loss.append(sum(map(extractmodelloss, history))/len(history))
    time_taken.append(time.time()-start_time)
    return history
```

We then initialize the three_layer_forward_feed_neural_network with the following values: 48 as there are 48 input values paper data point, 100 as we will have 100 nodes per hidden layer, and 3 as there are only 3 categorical outputs possible: 0 - Good Cases, 1 - Weaving Cases, 2 - Short-Forming Cases. We will be using the SGD optimizer as required by the question and a learning rate of 0.5 when fitting our three_layer_forward_feed_neural_network.

Results:

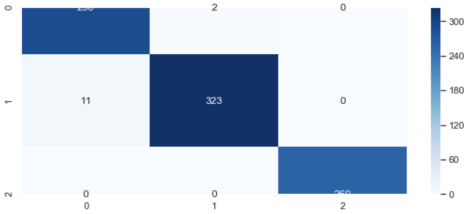
```
train_actual = []
train_pred = []

for i in train:
    train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
    train_actual.append(int(i[48]))

print("Train set accuracy: " + str(accuracy_score(train_actual, train_pred)))

cf_trainset_matrix = confusion_matrix(train_actual, train_pred)
sns.heatmap(cf_trainset_matrix, annot=True, cmap='Blues', fmt='g')
```

```
Train set accuracy: 0.9919311457772996

<matplotlib.axes._subplots.AxesSubplot at 0x7ffbea2ca850>
```




```
test_actual = []
test_pred = []

for i in test:
    test_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
    test_actual.append(int(i[48]))

print("Test set accuracy: " + str(accuracy_score(test_actual, test_pred)))

cf_testset_matrix = confusion_matrix(test_actual, test_pred)
sns.heatmap(cf_testset_matrix, annot=True, cmap='Blues', fmt='g')
```

```
Test set accuracy: 0.9853273137697517

<matplotlib.axes._subplots.AxesSubplot at 0x7ffbec8e14d0>
```



Conclusion:

We can push the model accuracy and performance on both the training sets to 100% if we set the number of epochs to a value high enough (e.g. 100, 200). However, that does come with its own pros and cons, falling under the area of hyperparameter tuning which will be covered subsequently under tasks 2 - 4, where we will investigate the effects of the various parameters on the model performance and accuracy as well as the resultant complexities and issues revolving around model underfitting and overfitting - a prevalent issue in machine learning and deep learning.

Task 2: You are asked to study the effect of network structure: hidden nodes, hidden layers to the classification performance. That is, you try different network configurations and understand the patterns. Your experiments have to be well-documented in your Jupyter notebook file and your report. It has to cover different aspects of network configurations such as shallow network, wide network, deep network etc.

Note that your analysis should touch the issue of accuracy as well as complexity for task 2-4. The complexity can be defined in the context of execution time.

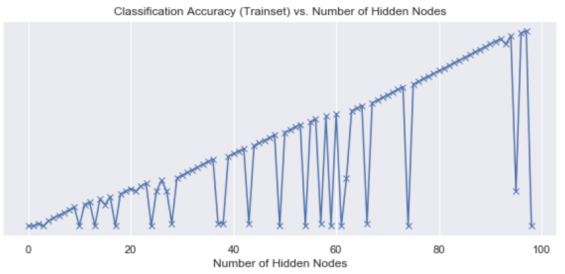
Hypothesis - hidden nodes & hidden layers:
According to a study conducted by Professors Sergey Zagoruyko, Nikos Komodakis on their research paper on Wide Residual Networks, it is noted that widenning (i.e. increasing the number of nodes per hidden layer) improves performance across residual networks even of different and varying depths (i.e. the number of hidden layers within a neural network). Increasing both the width and the depth of a neural network has a positive correlation and causality with its performance to a point whereby the number of parameters is so high that stronger regularization is needed instead to further tune the neural network.

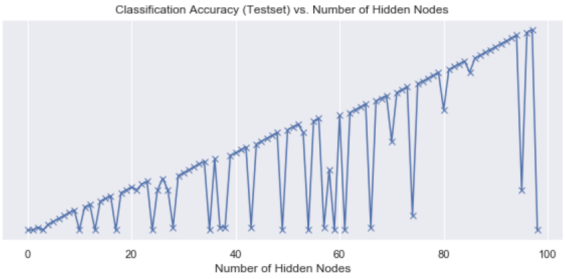
Overview - hidden nodes:

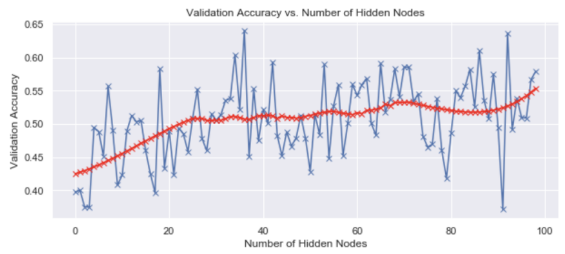
```
input_size = 48
num_classes = 3

time_to_train_model = []
average_model_validation_accuracy = []
average_model_validation_loss = []
average_model_test_accuracy = []
average_model_train_accuracy = []


for i in range (1, 100):
    history = []
    train_loader = DataLoader(train, BATCH_SIZE, shuffle=True, num_workers=0, pin_memory=True)
    val_loader = DataLoader(validation, BATCH_SIZE*2, num_workers=0, pin_memory=True)
    model = three_layer_forward_feed_neural_network(input_size, i, i, num_classes)
    history += fit(10, 0.5, model, train_loader, val_loader, average_acc = average_model_validation_accuracy, average_


    train_actual = []
    train_pred = []

    for i in train:
        train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        train_actual.append(int(i[48]))

    average_model_train_accuracy.append(str(accuracy_score(train_actual, train_pred)))

    test_actual = []
    test_pred = []

    for i in test:
        test_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        test_actual.append(int(i[48]))

    average_model_test_accuracy.append(str(accuracy_score(test_actual, test_pred)))
```

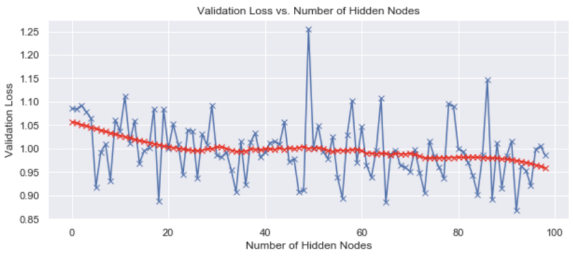
Using the same three_layer_forward_feed_neural_network, we will run a for loop to create 100 different models with an incremental number of hidden nodes per layer. For each iteration, we will generate a brand new model, validation data and training data loaded by the Pytorch DataLoader module. We will also observe and store the following metrics in order to further analyze the impact of widening and increasing the number of hidden nodes per layer on the neural network: time to train model, model validation accuracy, model validation loss, model trainset accuracy, and model testset accuracy.

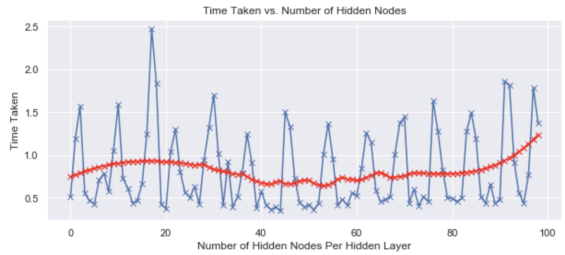
Results:







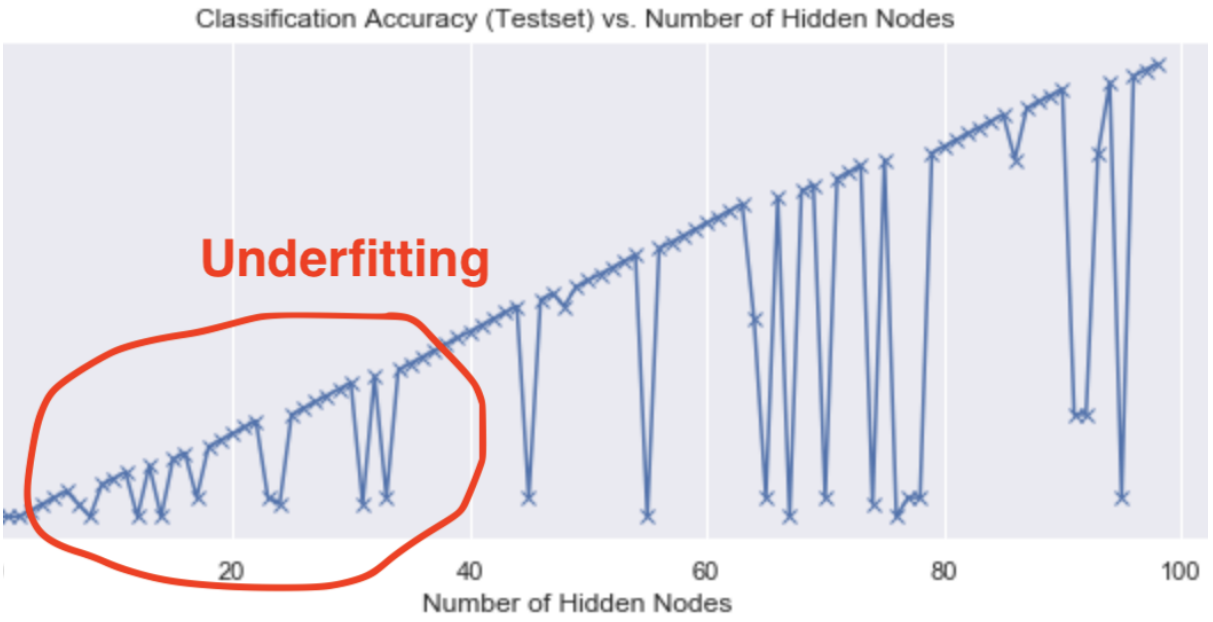




Conclusion:

We observe that as the number of hidden nodes increases per hidden layer of the same neural network, the classification accuracy of both the trainset and testset, the average validation accuracy of the model increases as well. Consequently, the validation loss and expectedly the classification error (1 - accuracy) decreases, which overall points to the observation that increasing the number of hidden nodes per hidden layer improves the accuracy and performance of the model. There seems to be an increasing trend to the time taken to completely fit the model as the number of hidden nodes increase albeit it is not significant (i.e. 1 hidden node model takes less than 1 second to fit averagely compared to a 100 hidden nodes model which takes around 1.5 seconds on average to fit). The observations are skewed with random outliers, resulting in random spikes for all graphs. According to Andrew Ng, one of the leading researchers at Deeplearning.AI, they are an unavoidable result of using mini-batch gradient descent (BATCH_SIZE=128 as initialized in the code), and some of these mini-batches are assigned by chance ‘unlucky data’ for optimization. To smoothen out the spikes and better interpret the graphs, I used Savitzky–Golay filter to smoothen curve without greatly affecting the signal tendency via convolution, before allowing us to observe trend in red

Underfitting:

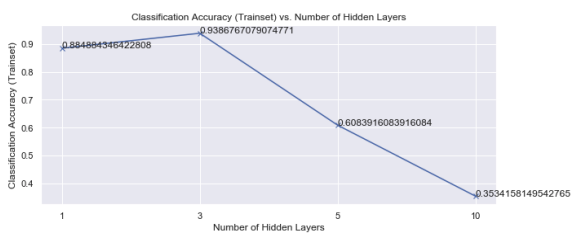


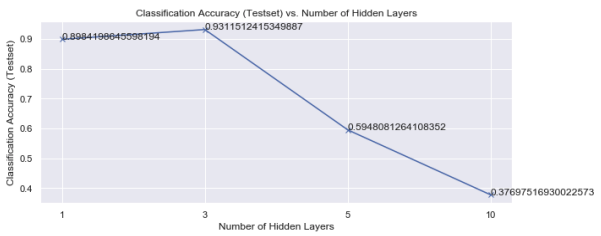
Underfitting occurs when we have undertrained our models, which results in the model not learning and capturing the dominant trend of the dataset, resulting in a model which has low variance high bias, lowering its accuracy score and thereby rendering it ineffective in being useful to make accurate predictions. Above, we can see that as the number of nodes are at a relatively small value (circled in red), the accuracy scores respectively are proportionally low. This is so given that there aren't enough nodes which function as computational units to make predictions, resulting in lower prediction accuracies and hence low accuracy scores with small numbers of hidden nodes present in the hidden layers when fitting the model to the dataset.

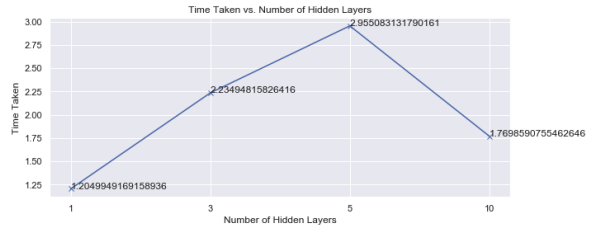
Overview - hidden layers:

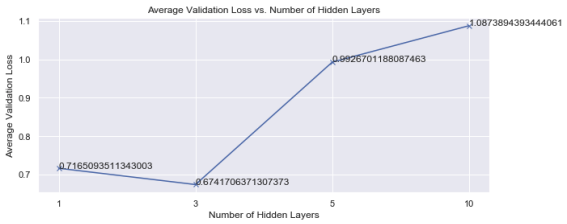
To evaluate the impact of the actual number of hidden layers within a neural network on its own accuracy and performance, I have created 5 different neural networks with differing numbers of hidden layers: 1 layer neural network, 3 layer neural network, 5 layer neural network, and 10 layer neural network. Each of these neural networks with differing numbers of hidden layers have the same number of hidden nodes per hidden layer (i.e. 100 hidden nodes). Every other condition such as learning rate, optimizer used, batch size, and number of epochs have been kept constant throughout this experiment.

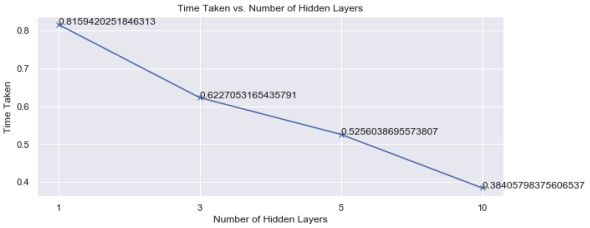
Result:







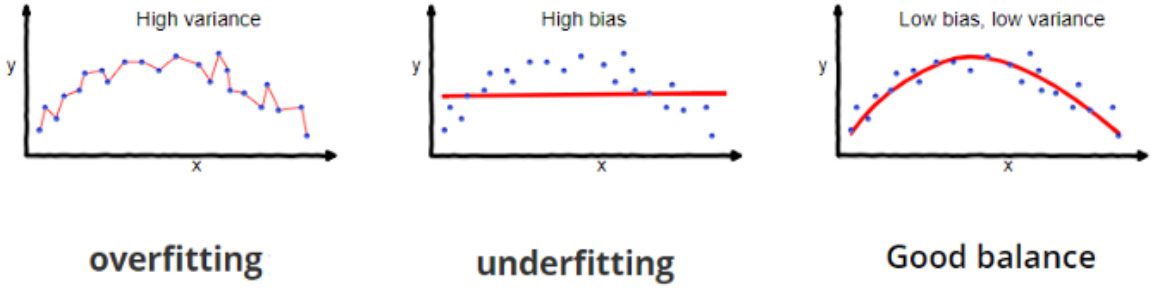




Conclusion:

The general trend observed is that beyond 3 layers, the accuracy and performance of our neural network deteriorates rapidly as observed from our validation accuracy, validation loss, and classification accuracy for both trainset and testset graphs. Time taken to train the model also increases with the number of hidden layers expectedly given the rise in complexity and hence cost and resources required to fit the model.

Overfitting:



One possible explanation to why adding hidden layers beyond a certain value decreases the performance and accuracy of our neural network is due to overfitting. Overfitting occurs when too much information has been processed from the relatively and considerably less amount of data used to train and fit the model. Our model has captured the outliers and noise within each batch during training and validation, resulting in low bias but high variance, skewing its prediction accuracy and rendering it ineffective when using it for classification purposes such as seen in this task.

Task 3: You are asked to study the effect of learning rates. As with Task 2, your experiments have to be well documented. You need to give the correct conclusion and give suggestions on how learning rates should be set. This includes possible adaptive learning rates where the value increases or decreases as the increase of epochs.

Note that your analysis should touch the issue of accuracy as well as complexity for task 2-4. The complexity can be defined in the context of execution time.

Hypothesis - learning rate & epochs:

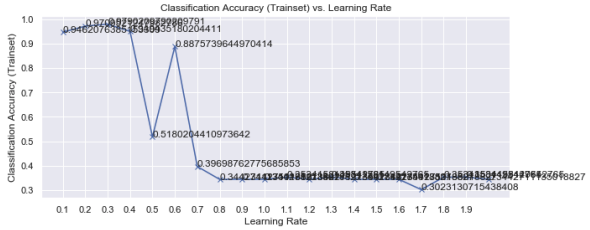
Given that we are using the Stochastic Gradient Descent (SGD) algorithm as our optimizer, we need to be aware of how the algorithm works when optimizing and training our neural network. SGD estimates the error gradient for the current state of the model before updating its weight using the in-built and inherent back-propagation algorithms. The magnitude in the changes and updates of these weights during training is known as the “learning rate”. Learning rate is a tunable parameter which controls the speed in which the model fits the dataset. Given this definition, we can expect that ther higher the learning rate, the shorter amount of time the model takes to be trained. Additionally, given that the number of epochs remains the same, the performance and accuracy of the model increases with the learning rate. As for the number of epochs, it is defined as the number of times the entire dataset is passed forward and backward through the entire neural network once. Given this definition, we can expect that as the number of epochs used to train the neural network has a positive relationship with the performance of the neural network as well as higher epochs will allow more instances and times for the neural network to fit and be trained by the dataset.

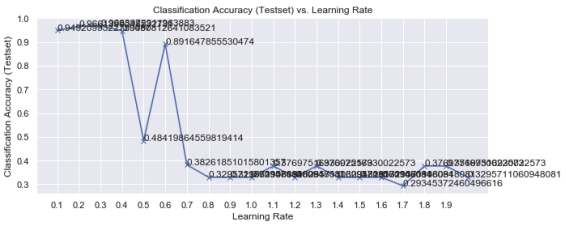
Overview - learning rate:

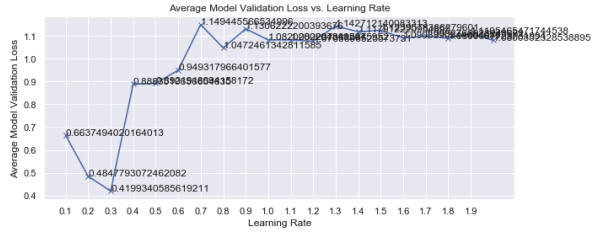
Using the 3 layer neural network we design in task 1, we iterated it through a for loop over 20 iterations, for each iteration, we use a new neural network and we divide the iteration index by 10 to get a value < 1 as the learning rate. After that, we collated the necessary data (classification accuracy, validation loss, validation accuracy and time taken) as we did for Task 2 for subsequent analysis below.

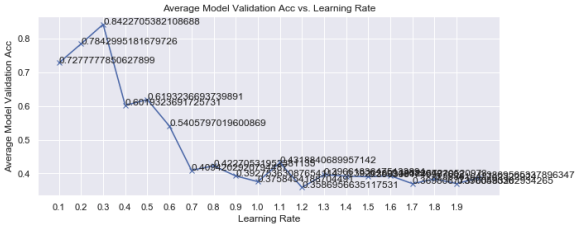
```
"""
We will be using the three_layer_forward_feed_network used in task 1

Iterate through values 1 to 20 to get 0.1 - 2 as the learning rate and for that particular iteration.
For each iteration, fit and train the model with the respective learning rate values
"""

input_size = 48
hidden_size = 100
num_classes = 3

compile_avg_model_loss = []
compile_avg_model_acc = []
compile_model_trainset_acc = []
compile_model_testset_acc = []
compile_model_time_to_train = []


for a in range(1, 21):

    avg_model_loss = []
    avg_model_acc = []
    time_to_train = []

    train_loader = DataLoader(train, BATCH_SIZE, shuffle=True, num_workers=0, pin_memory=True)
    val_loader = DataLoader(validation, BATCH_SIZE*2, num_workers=0, pin_memory=True)
    model = three_layer_forward_feed_neural_network(input_size, hidden_size, hidden_size, num_classes)

    history = [evaluate(model, val_loader)]

    history += fit(20, a/10, model, train_loader, val_loader, average_loss = avg_model_loss, average_acc = avg_model_a

    train_actual = []
    train_pred = []

    for i in train:
        train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        train_actual.append(int(i[48]))

    print("Train set accuracy: " + str(accuracy_score(train_actual, train_pred)))

    test_actual = []
    test_pred = []

    for i in test:
        test_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        test_actual.append(int(i[48]))

    print("Test set accuracy: " + str(accuracy_score(test_actual, test_pred)))

    compile_avg_model_loss.append(avg_model_loss[0])
    compile_avg_model_acc.append(avg_model_acc[0])
    compile_model_trainset_acc.append(accuracy_score(train_actual, train_pred))
    compile_model_testset_acc.append(accuracy_score(test_actual, test_pred))
    compile_model_time_to_train.append(time_to_train[0])
```

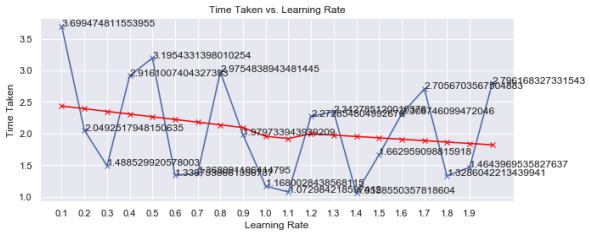
Results:











Conclusion:

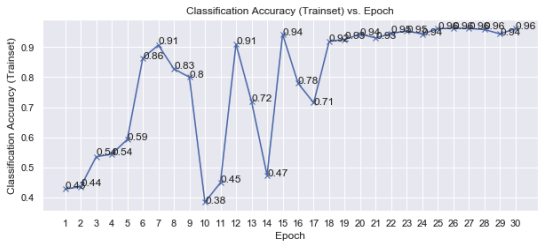
At low learning rates (~0.1 - 0.3), we observe that the model performs well with high classification and validation accuracies while having conversely low error and validation losses. However, unexpectedly, as the learning rate further increases, we observe that the performance of the model drastically decreases. This can be muchly attributed to the fact that perhaps with high learning rates, the model converges too quickly and arrives at a hence suboptimal value

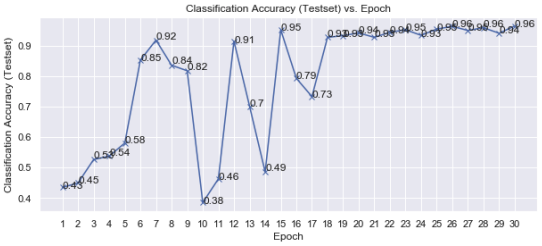
Overview - epochs:

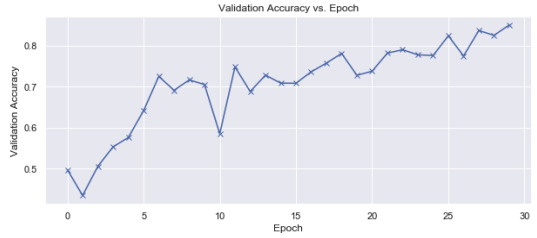
```
"""
We will be using the three_layer_forward_feed_network used in task 1

Iterate through values 1 to 30 to get the number of epochs for that particular iteration.
For each iteration, fit and train the model with the respective number of epochs
"""

input_size = 48
hidden_size = 100
num_classes = 3

compile_avg_model_loss = []
compile_avg_model_acc = []
compile_model_trainset_acc = []
compile_model_testset_acc = []
compile_model_time_to_train = []


for a in range(1, 31):

    avg_model_loss = []
    avg_model_acc = []
    time_to_train = []

    train_loader = DataLoader(train, BATCH_SIZE, shuffle=True, num_workers=0, pin_memory=True)
    val_loader = DataLoader(validation, BATCH_SIZE*2, num_workers=0, pin_memory=True)
    model = three_layer_forward_feed_neural_network(input_size, hidden_size, hidden_size, num_classes)

    history = [evaluate(model, val_loader)]

    history += fit(a, 0.1, model, train_loader, val_loader, average_loss = avg_model_loss, average_acc = avg_model_acc

    train_actual = []
    train_pred = []

    for i in train:
        train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        train_actual.append(int(i[48]))

    print("Train set accuracy: " + str(accuracy_score(train_actual, train_pred)))

    test_actual = []
    test_pred = []

    for i in test:
        test_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        test_actual.append(int(i[48]))

    print("Test set accuracy: " + str(accuracy_score(test_actual, test_pred)))

    compile_avg_model_loss.append(avg_model_loss[0])
    compile_avg_model_acc.append(avg_model_acc[0])
    compile_model_trainset_acc.append(accuracy_score(train_actual, train_pred))
    compile_model_testset_acc.append(accuracy_score(test_actual, test_pred))
    compile_model_time_to_train.append(time_to_train[0])
```

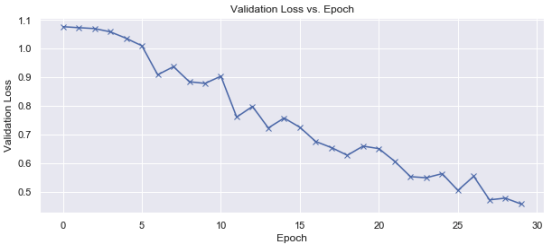
Using the 3 layer neural network we design in task 1, we iterated it through a for loop over 30 iterations, for each iteration, we use a new neural network and we use the interaction index as the number of epochs used to train that model for that particular iteration. After that, we collated the necessary data (classification accuracy, validation loss, validation accuracy and time taken) as we did for Task 2 for subsequent analysis below.

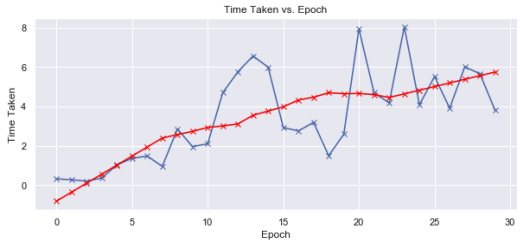
Results:











Conclusion:

As initially suspected in our hypothesis, indeed the performance of our neural network increases with the number epochs that the neural network uses to train and fit itself to the given dataset. Using a high number of epochs also results in greater complexity in terms of our final model and hence would take a longer time to train and fit on average compared to a model trained with lesser number of epochs as observed in the graph above.

Task 4: You are asked to study the effect of mini-batch size. You can set the mini-batch size to be 1 (stochastic gradient descent), N (batch gradient descent) or any other size. The most important aspect is to be conclusive with your findings. The mini-batch size really depends on the problem size.

Note that your analysis should touch the issue of accuracy as well as complexity for task 2-4. The complexity can be defined in the context of execution time.

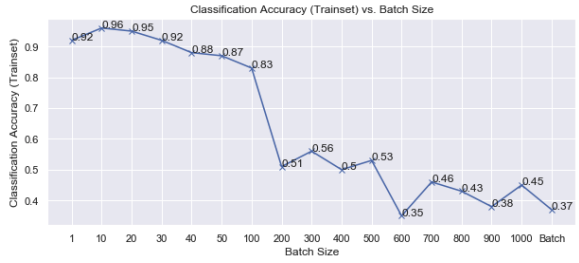
Hypothesis - different batch sizes and gradient descents:

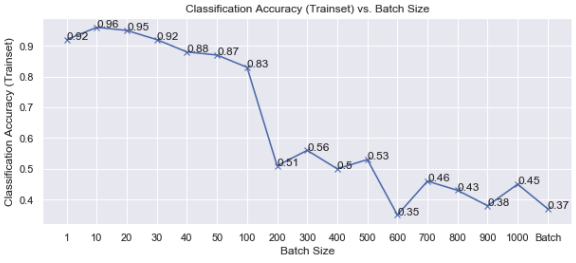
Gradient descent refers to the optimization algorithm used when training a predictive model. Based on the batch-size we use when fitting our model, the type of gradient descent changes and hence subsequently, the overall impact it has on the performance and accuracy of our model. Stochastic Gradient Descent (SGD) is used when the batch size is exactly 1 (i.e. every epoch, the model fits and trains itself using only 1 data point), Mini-Batch Gradient Descent (MGD) is used when the batch sizes is more than 1 but less than the length of the entire dataset (i.e. every epoch, the model fits and trains itself using N data points where N is the batch size). Batch Gradient Descent (BGD) results from using the entire dataset (i.e. every epoch, the model fits and trains itself using the entire dataset). From this, we can also infer that to achieve a similar minimum accuracy threshold, the smaller the batch size, the greater the number of epochs. According to machine learning researchers, too large batch sizes results in poor generalization made by the model which affects the overall accuracy of our neural network

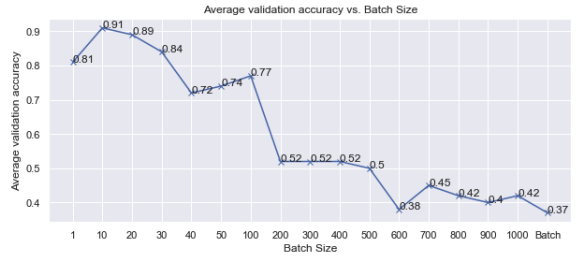
Overview:

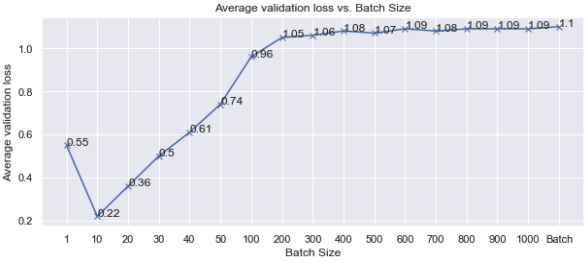
```
"""
SGD

MGD (10 - 100 batch size)

BGD (The entire training data set)
"""


input_size = 48
hidden_size = 100
num_classes = 3
batch_size = 1

time_to_train_model = []
average_model_validation_accuracy = []
average_model_validation_loss = []
model_test_accuracy = []
model_train_accuracy = []

history = []
train_loader = DataLoader(train, batch_size, shuffle=True, num_workers=0, pin_memory=True)
val_loader = DataLoader(validation, batch_size*2, num_workers=0, pin_memory=True)
model = three_layer_forward_feed_neural_network(input_size, hidden_size, hidden_size, num_classes)
history += fit(5, 0.1, model, train_loader, val_loader, average_acc = average_model_validation_accuracy, average_loss

train_actual = []
train_pred = []

for i in train:
    train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
    train_actual.append(int(i[48]))

model_train_accuracy.append(str(accuracy_score(train_actual, train_pred)))

test_actual = []
test_pred = []

for i in test:
    test_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
    test_actual.append(int(i[48]))

model_test_accuracy.append(str(accuracy_score(test_actual, test_pred)))

input_size = 48
hidden_size = 100
num_classes = 3
batch_size = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

time_to_train_model = []
average_model_validation_accuracy = []
average_model_validation_loss = []
model_test_accuracy = []
model_train_accuracy = []


for i in range (1, 10):
    history = []
    train_loader = DataLoader(train, batch_size[i], shuffle=True, num_workers=0, pin_memory=True)
    val_loader = DataLoader(validation, batch_size[i]*2, num_workers=0, pin_memory=True)
    model = three_layer_forward_feed_neural_network(input_size, hidden_size, hidden_size, num_classes)
    history += fit(5, 0.1, model, train_loader, val_loader, average_acc = average_model_validation_accuracy, average_l


    train_actual = []
    train_pred = []

    for i in train:
        train_pred.append(torch.argmax(model(torch.Tensor(i[0:48]))).item())
        train_actual.append(int(i[48]))

    model_train_accuracy.append(str(accuracy_score(train_actual, train_pred)))
```

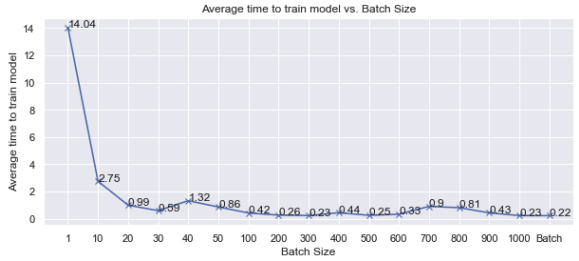
Results:











Conclusion:

As stated initially in our hypothesis, at low batch sizes, the model performs really well with high validation accuracies and classification accuracies while having low error rates and validation losses. Conversely, as the batch size increases, we observe slight decreases in performance till batch sizes beyond 100, where we see a sharp decline in model performance in terms of both classification accuracy and loss. Interestingly enough, the model takes extremely long to train when using batch sizes of 1 in every epoch. We observe that after batch sizes of 10, the average time to train the model drastically decreases to less than 2 seconds where this metric plateaus even as the batch size increases to 1000.

Overall:

Optimal number of hidden layers:
3

Optimal Learning rate:
~0.2

Optimal Batch Size:
~10

Optimal number of hidden nodes per hidden layer:
None (the higher the value, the better the performance)

Optimal number of epochs:
None (the higher the value, the better the performance)

Relationship between metric and complexity:

Increase in metric:	Complexity (time taken to train model)
Hidden layers	Increases
Learning rate	Decreases
Batch Size	Decreases
Hidden Nodes	Increases
Epochs	Increases