Question 1

This short report will discuss how mini-batch stochastic gradient descent (SGD) can be used to reduce overfitting and improve generalization performance. It will use a neural network and dataset as a foundation to work from, and conduct experiments to determine how mini-batch size, Gaussian noise, step-size and adaptive approaches such as Adam can influence overfitting and generalization performance.

Overfitting

It is important to establish what overfitting is, so that it is clear what this report will focus on. Overfitting occurs when a model is accurate at predicting similar examples from its training dataset, but is not accurate at predicting examples outside its training dataset. For example, a model that was trained on only 10 examples out of 10,000 possible examples may struggle to generalize to the 9,990 other examples it was not trained on.

This report aims to see if the “noise” added to the gradient when using mini-batch SGD acts as a regulariser, subsequently preventing overfitting. The ”noise” added to the gradient is not a direct addition, but rather from the randomness from using a random mini-batch of the training data to get the gradient and not the entire dataset (which gradient descent would do). From this, it is intuitive to guess that a small mini-batch size will have more ”noise” from a small random subset of data to get the gradient, and a big mini-batch size will have less noise from a big random subset of data (possibly even the full dataset, which would mean it acts the same as gradient descent) to get the gradient. However, this report will aim to test this hypothesis and how other aspects such as the step-size and adaptive approaches like adam step-size updates can influence the overfitting of a model.

Overfitting can be measured in many ways, but since SGDs are inherently random, this report will use an intuitive approach that focuses on trends and not exact values by plotting the accuracy and loss of the model over 3 attempts. The average training accuracy represents the proportion of correctly classified samples in the training dataset. The average test accuracy represents the proportion of correctly classified samples in a separate dataset which the model was not trained on. If the training accuracy is high, and the test accuracy is low, it shows that the model is overfitted to the training data. Similarly, the training loss represents the error between the model’s predictions and the actual values in the training dataset. The test loss measures the error between the model’s predictions and the actual values in a separate dataset that the model was not trained on. If the training loss is low and the test loss is high, the model is overfitted to the training data.

Neural network and dataset

The type of neural network that will be used is a convolutional neural network (CNN). This is because CNN’s are prone to overfitting since they contain multiple layers in a hierarchical structure, which makes them focus on noise in the training dataset instead of generalizing well to unseen data. Both simple and complex CNNs can be overfitted, but simple CNNs are more prone to be overfitted since they have fewer parameters to learn from, and subsequently focusing on redundant noise and specific details in the training dataset instead of generalizing well to unseen data. Furthermore, using convolutional neural networks for image processing is a good candidate for overfitting because images are complex with many pixels using a high dimensional input space, giving the model a chance to focus on the intricate details of the training data and not generalizing well to unseen data. In short, a CNN with a simple structure and complex data such as from image processing is a suitable candidate, and will be used in this report.

The dataset that will be used is the Fashion MNIST dataset from keras[1]. This dataset is used because it contains 70,000 images in total, which is a relatively small amount of images for the 10 fashion categories it uses. Using a small dataset with many categories has a higher chance of overfitting, since the model has less information to learn about each category. Subsequently, this increases the chance that the test set will contain examples that the model did not encounter before, and the model will struggle to generalize correctly to this unseen data. This data is split into training, test and validation sets, while acknowledging that cross-validation (where subsets of the same datasets are used) is not suitable for measuring generalization behaviour properly because the model will use data that it has seen before, making the evaluation not use the full variability of unseen data. 80% of the data is for training (56,000 images), 10% is for validation (7000 images) inside the training loop, and 10% is for testing (7000 images). Since SGD is inherently random, 3 runs will be used to make conclusions with this data for 20 epochs to get a model that is trained well for enough time. To make use of keras while also implementing tweaks to its SGD, I implemented a custom training and evaluation step[2] instead of using `.fit()`. This allows the adjustment of the training step to add elements such as Gaussian ”noise”. Furthermore, the validation set (10% of all data) was used inside in a validation loop at the end of each epoch to evaluate with unseen data.

Mini-batch size on overfitting

To investigate the role of mini-batch size on overfitting, I used a range of mini-batch sizes [16, 32, 64, 128, 256, 512, 1024] to start with. Each mini-batch size was run for 3 times, and the average for the test accuracy, training accuracy and validation accuracy was measured. I set the step size constant as 0.0001 (default) which is common practice, and I ran it for 20 epochs to get a reasonable accuracy and loss which could be reasoned about. The accuracy for training, testing and validation is shown below.

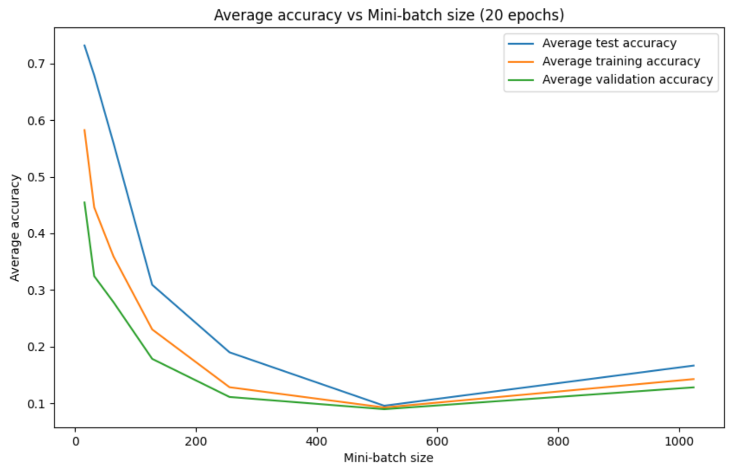


Figure 1: Various mini-batch sizes versus training, test and validation accuracy

From above, it is evident that the model is most accurate with unseen data when using a low mini-batch size, in particular when the mini-batch size is 16. This is because the average test accuracy is even higher than the average training accuracy and average validation accuracy, implying no overfitting. The validation accuracy in green is always the lowest, because it is while the model is training and evaluating with unseen data, but the completed model has an average test accuracy which is above 0.72 in green when the mini-batch size is 16.

As the mini-batch size increases, the training, test and validation accuracy all decreases until the mini-batch size is around 500 when it starts to increase again. This phenomenon can be explained by the model eventually getting access to more diverse examples at each iteration. However, it is clear that smaller mini-batch sizes reduces overfitting better. The loss for training, testing and validation is shown below.

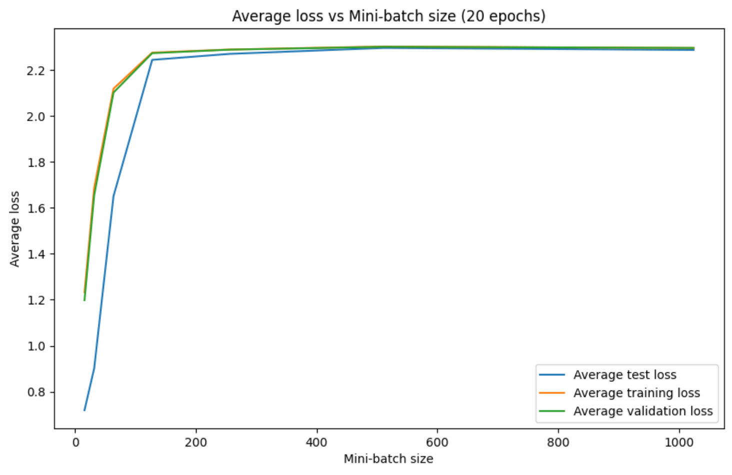


Figure 2: Various mini-batch sizes versus training, test and validation loss

From above, it is clear that the model is the least wrong with unseen data when using a low mini-batch size, in particular when the mini-batch size is 16. This is because the average test loss is lower than the average training loss and the average validation loss, meaning that the difference between the model's predictions and the actual data is less with unseen data compared to seen data from the training data, implying no overfitting. After a mini-batch size > 100, the average test loss becomes the same as the average training and validation loss, meaning that there is a potential for the model to be overfitted. We conclude that using a small mini-batch size reduces overfitting.

It has been established that a small mini-batch size of 16 is beneficial since it results in a lower chance of overfitting (from a higher average test accuracy and lower average test loss). Now, the gradient noise can be measured to show how this relates to the mini-batch size. As mentioned previously, the "noise" from mini-batch SGD is not added directly, but indirectly from the randomness/stochasticity of the model choosing small mini-batches from the training data. To measure this "noise", an effective way is to measure the variance of the gradient across batches, and gather conclusions from the average of this variance across 3 runs for each mini-batch size as shown in the plot below. The plot shows that small mini-batch sizes have the highest variance for the gradient across batches, with 16 having the most variance and therefore the most "noise".

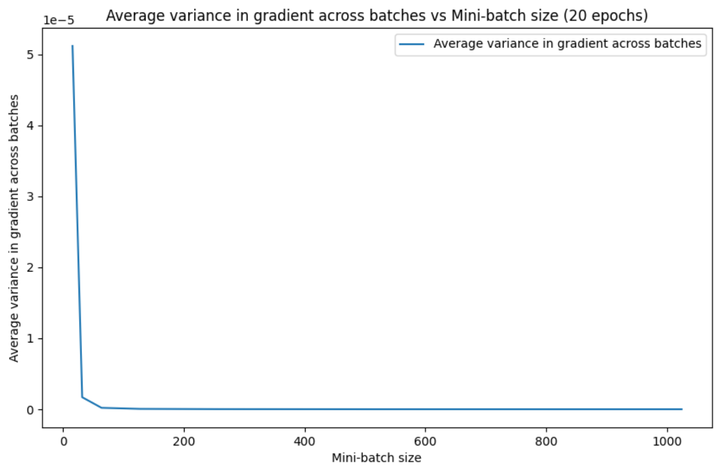


Figure 3: Various mini-batch sizes versus average variance in gradient across batches

After a mini-batch size > 64, the ”noise” (i.e. the variance of the gradients) is stagnant near 0. Therefore, low mini-batch sizes such as 16 have the highest ”noise”, which subsequently yields the highest test accuracy and lowest test loss, reducing overfitting by acting like a regulariser (by discouraging the model from learning overly complex patterns which may cause overfitting). When the full length of the training dataset (56000) was used as the mini-batch size (acting as gradient descent), the same phenomenons already mentioned occurs, which is that only a small mini-batch size of 16 reduced overfitting the most by having the highest test accuracy and lowest test loss. After the mini-batch size > 64, the model is susceptible to overfitting by having little ”noise”).

Gaussian noise on overfitting

To investigate the role of adding Gaussian noise on overfitting, I used a constant mini-batch size of 16 with a step size of 0.0001 (default as before) and added varying standard deviations of Gaussian noise from [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] to the gradient as shown:

```
gradients_with_gaussian_noise = [gradient +
tf.random.normal(gradient.shape, mean=0.0,
stddev=standard_deviation_for_gaussian_noise) for gradient in gradients]
```

The average from 3 runs is plotted below for the training, test and validation accuracy.

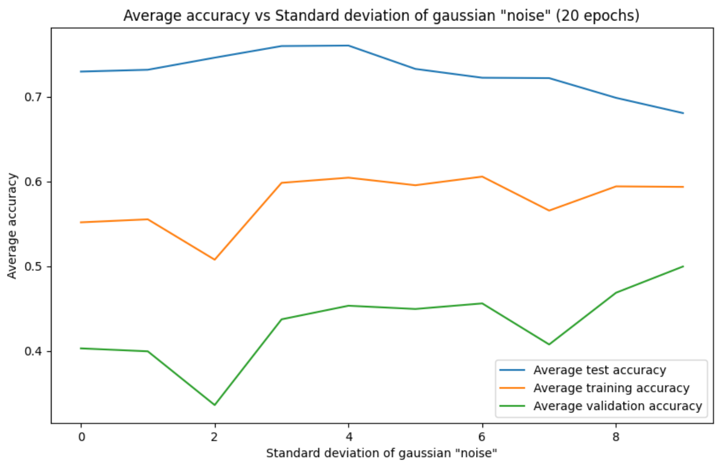


Figure 4: Various standard deviations of Gaussian ”noise” versus training, test and validation accuracy

From above, it is evident that a standard deviation of 4 for the Gaussian ”noise” resulted in the highest average test accuracy compared to the average training and validation accuracy which were lower (hence the lowest chance of overfitting since the model does the best at this point with unseen data). At standard deviations > 4 for the Gaussian ”noise”, the accuracy started to decrease for the average test accuracy as well as the average training and validation accuracy. This range of standard deviations was also plotted with the training, test, and validation loss which is shown below:

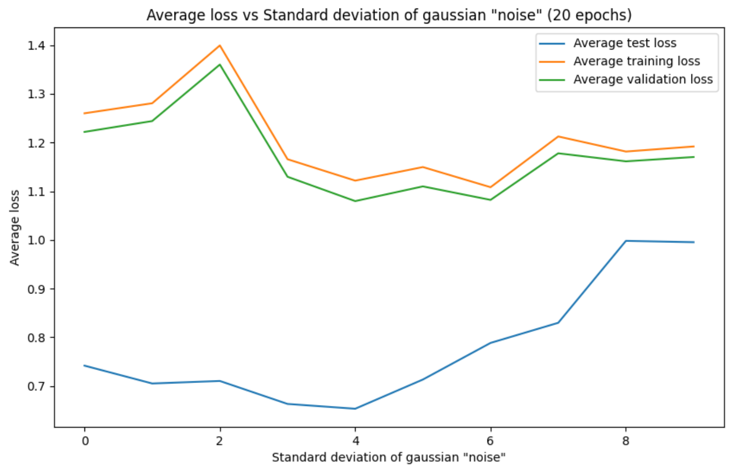


Figure 5: Various standard deviations of Gaussian ”noise” versus training, test and validation loss

From above, it is evident that a standard deviation of 4 for the Gaussian ”noise” resulted in the lowest average test loss compared to the average training loss and validation loss (hence the lowest chance of overfitting, since the model does the best at this point with unseen data). At standard deviations > 4 for the Gaussian ”noise”, the loss started to increase for the average test loss as well as the average training loss and validation loss.

From this, we conclude that a moderate level of Gaussian noise (with a standard deviation of 4) increases the test accuracy and decreases the test loss the most, therefore mitigating overfitting. However, too much or too little Gaussian noise can give suboptimal reductions in overfitting and generalization performance since the updates to model parameters can become erratic - finding a moderate level is important.

In comparison to varying the mini-batch size, mini-batch sizes and the manual addition of Gaussian ”noise” had a similar impact on the average test accuracy (best being 0.73 and 0.72 respectively), but Gaussian ”noise” had a more significant impact on the average test loss (best being 2.2 and 0.7 respectively). The accuracy and loss improvements from Gaussian ”noise” is important for reducing overfitting, since a higher test accuracy and lower test loss means the model is better at generalizing to unseen data, not just to its training data. A small mini-batch size of 16 and a moderate Gaussian ”noise” (with a standard deviation of 4) both reduced overfitting through high a test accuracy and low test loss, but the addition of a moderate Gaussian ”noise” to the gradient had a more significant impact on a decreased test loss. Both are valid approaches to reducing overfitting, but adding moderate Gaussian ”noise” gets to the root of reducing overfitting by adding ”noise” (which is what a small mini-batch size does anyway).

Step size and mini-batch size on overfitting

To investigate the relationship between step size and mini-batch size on overfitting, I used a constant mini-batch size of 16 and implemented a step size scheduler into my custom training step as shown below (acknowledging that epoch starts from 0):

```
def step_size_scheduler(initial_learning_rate, epoch):
    return initial_learning_rate * np.exp(0.04 * epoch+1)
```

The average from 3 iterations was used for each step size from [0.0001, 0.001, 0.01, 0.1] to see the impact on training, test and validation accuracy shown below:

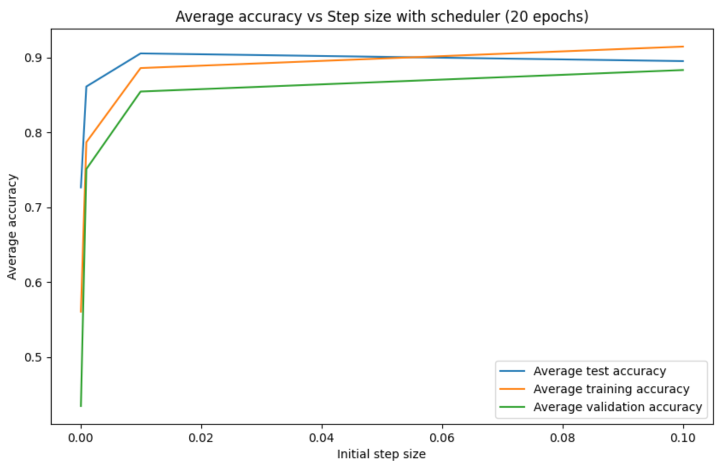


Figure 6: Various step sizes with a scheduler versus training, test and validation accuracy

From above, it is clear that overfitting is reduced the most when the step size is 0.01 since the average test accuracy is the highest at this point (data does well at predicting unseen data). The average test accuracy is not as high along with the average training and validation accuracy when the step size is < 0.01 or > 0.01. The average from 3 runs is used for each step size to get the training, test and validation loss as shown below:

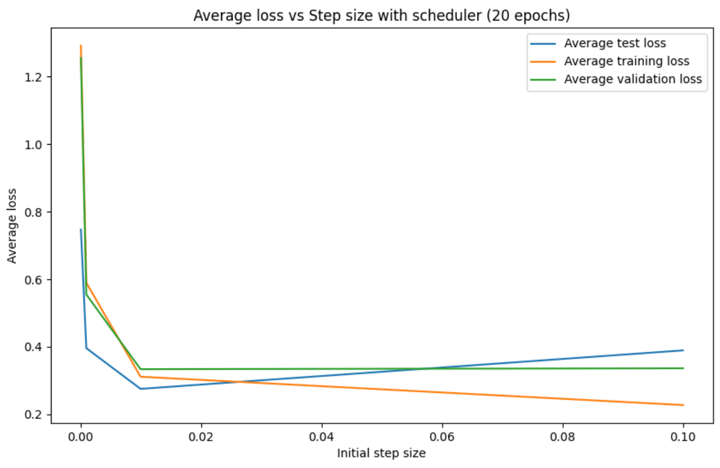


Figure 7: Various step sizes with a scheduler versus training, test and validation loss

From above, once again the step size of 0.01 with the step size scheduler reduced overfitting the most since the average test loss was the lowest with this (predictions of the model with unseen data is the lowest with this initial step size). Overall, it is observed that using 0.01 for the step size has a higher test accuracy (0.91) and a lower test loss (0.28) compared to any other configuration so far. Similarly, I implemented a mini-batch size scheduler with a constant step size of 0.01 (best as chosen previously) and mini-batch sizes from [16, 32, 64, 128, 256, 512, 1024]. The scheduler decreased the mini-batch size by a factor of 0.01 every epoch, with the minimum step size being 8 as shown below:

```
batch_size_schedule = lambda epoch: max(floor(mini_batch_size − (0.01 *
(epoch+1) * mini_batch_size)), 8)
```

The average from 3 runs is plotted below for the training, test and validation accuracy:

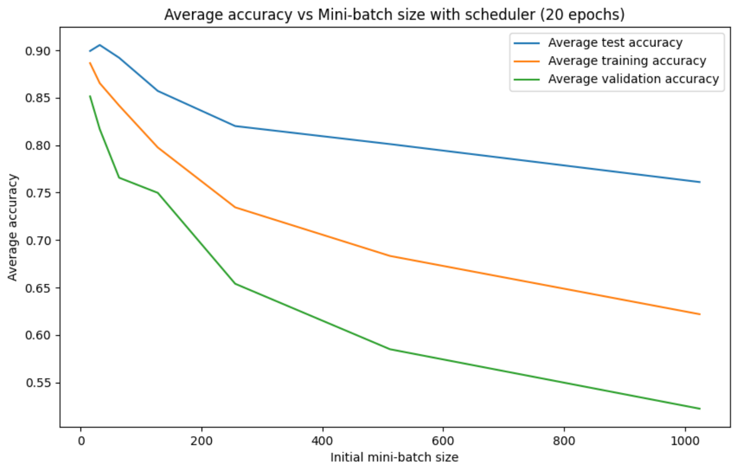


Figure 8: Various mini-batch sizes with a scheduler versus training, test and validation accuracy

From above, it is evident that small mini-batch sizes are still the best with a scheduler at reducing overfitting since the average test accuracy is the highest when the mini-batch size is 32, but note that this is higher than without the scheduler (it was previously 16). This makes sense because the mini-batch size is decreased slowly across epochs anyway. The average from 3 runs is plotted below for the training, test and validation loss:

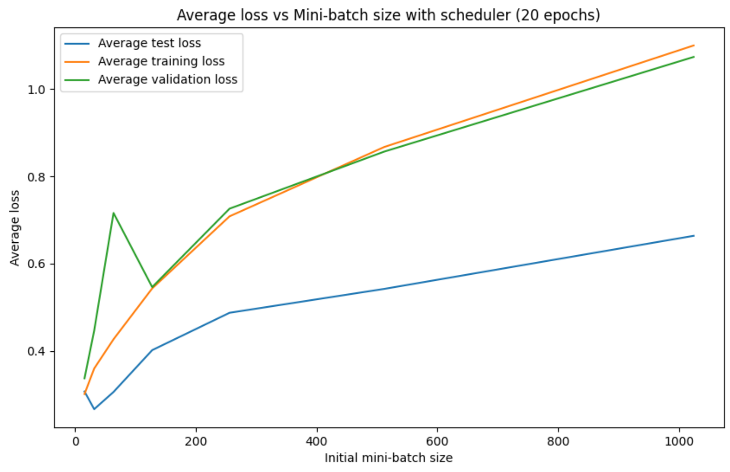
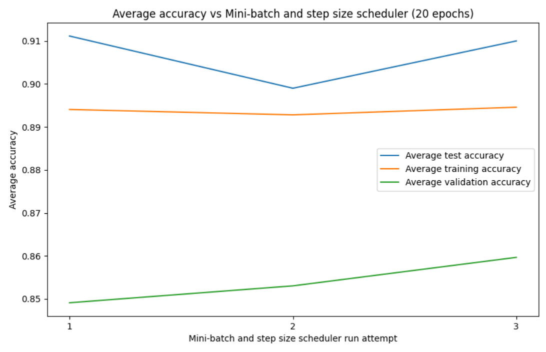


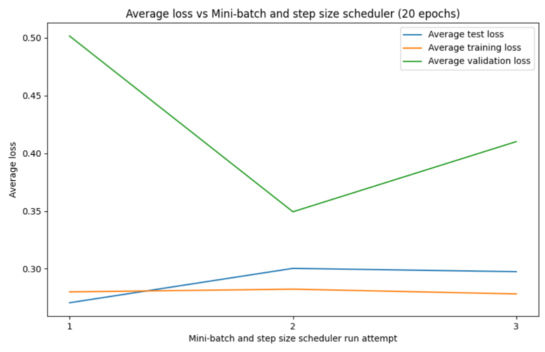
Figure 9: Various mini-batch sizes with a scheduler versus training, test and validation loss

From above, it is clear that the small mini-batch sizes have the lowest average test loss with the mini-batch scheduler and therefore reducing overfitting, but the mini-batch size of 32 has the lowest test loss and not 16, which once again makes sense because the scheduler decreases it across epochs, indicating that a mini-batch that is too small to begin with is detrimental to avoiding overfitting (mini-batch sizes which are too small cause updates that are erratic instead of updates based on more data which is smoother).

To investigate how the accuracy and loss looks when varying both mini-batch size and step size together, the step size scheduler and the mini-batch size scheduler mentioned previously were used at the same time using a step size of 0.01 and mini-batch size of 32 (best as chosen previously) over 3 attempts, the average accuracy and loss are below:



(a) Measuring training, test and validation accuracy



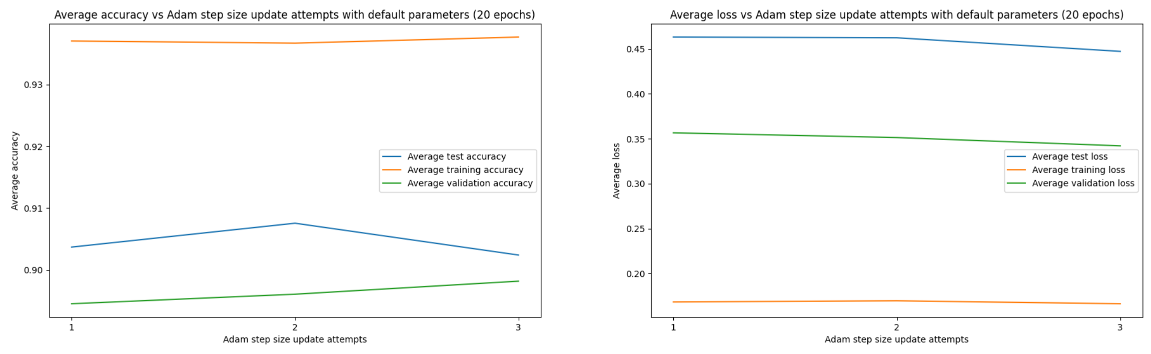
(b) Measuring training, test and validation loss

Figure 10: 3 attempts with a mini-batch size of 32 and step size of 0.01 (both varying with a scheduler) versus training, test and validation accuracy and loss

From above, it is evident that varying both mini-batch size and step size together with a scheduler for each is not much more effective than using a step size or mini-batch scheduler individually. They both increase the average test accuracy and test loss individually (Figure 6/7 and Figure 8/9), but when combined (Figure 10) they do not show much more improvement with an average test accuracy of around 0.90 and average test loss of around 0.3 which is similar to before. This is because varying both may lead to possibilities such as a large step size and low mini-batch size or a small step size and a large mini-batch size, which cause big or small parameter updates, based on little or very noisy gradients, making the model diverge or converge sporadically to a local minima.

Adam with overfitting

To investigate the effect of adaptive approaches instead of a constant step size, 3 runs were plotted for mini-batch SGD with the Adam optimizer to compare with previous runs in this report. The results are shown below for the accuracy and loss for 20 epochs with a mini-batch size of 16 while using the Adam optimizer (default parameters of $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and epsilon$= 1e - 7$ since this is a solid foundation to make deductions from). From the plots, it is clear that the training accuracy is slightly higher (at around 0.935) compared to the test accuracy (at around 0.905) indicating that there is slight overfitting. Similarly, the training loss is significantly lower (0.16) than the test loss (0.47), which indicates there is actually moderate overfitting, because the model's predictions are less wrong with the training data compared to the test data which is unseen. This overfitting phenomenon did not occur with the normal SGD keras optimizer (instead of Adam), suggesting that the use of adaptive learning rates (based on the past gradients) makes the impact of mini-batch size limited, causing more overfitting. Furthermore, Adam provides some regularization (from adaptive learning rates) from scaling updates (looking at the size of the past gradients), making the regularization effect from ”noise” added to the gradient in mini-batch SGD not as noticeable, causing potential overfitting.



(a) Measuring training, test and validation accuracy (b) Measuring training, test and validation loss

Figure 11: 3 attempts with a mini-batch size of 16 with Adam step size update (default parameters) versus training, test and validation accuracy and loss

Since the use of Adam has increased overfitting by blocking the regularization from the ”noise” of small mini-batch sizes, it makes sense that when choosing mini-batch sizes to control overfitting, it is best to avoid adaptive approaches since they minimize the effect of varying mini-batch sizes, making it more difficult to choose the one that avoids overfitting. Research in this area is fascinating, since it is commonly known that on image classification problems (such as this one with Fashion MNIST), Adam performs worse than SGD[3]. This aligns with the conclusions made here, and a workaround is to choose the mini-batch sizes without adaptive approaches and instead use constant step sizes to choose the best mini-batch size to control overfitting, as done with Figure 1/2.

Question 2

Line search in gradient descent

Line search can be used in gradient descent to choose a good step size value α. A form of line search is backtracking search which is a variant of grid search, which starts off with a large step α (which will not work well) and then it backtracks to reduce α until it is suitable. Line search tries to choose α which minimizes $f(x - \alpha\nabla f(x))$ where $f(x)$ is the cost function, x is the parameter vector currently used, $\nabla f(x)$ is the gradient of the cost function with respect to x (vector of derivatives), and of course α is the step size.

The α inside of the expression $x - \alpha\nabla f(x)$ is changed (by multiplying it by a parameter β) to get a new point, and subsequently you check if the function value at this point is $\leq f(x) - c\alpha\nabla f(x)^T\nabla f(x)$ (less than our starting value but also decreasing by some level i.e. the decrease is the $c\alpha\nabla f(x)^T\nabla f(x)$ part which is based on the derivative.

For gradient descent, if we’re choosing $\delta = -\alpha\nabla f(x)$, a local approximation of $f(x - \alpha\nabla f(x))$ is $f(x) - \alpha\nabla f(x)^T\nabla f(x)$ [since $f(x + \delta)$ is approximately $f(x) + f(x)^T\delta$]. The parameter c is added before it to be able to choose an α value which makes c similar to the decrease expected from the line in gradient descent.

One advantage of backtracking line search with gradient descent is that it ensures each step reduces the objective function value, subsequently leading to a convergence to a minimum, all while storing no

One disadvantage of backtracking line search with gradient descent is that it is computationally expensive to evaluate the objective function at different step sizes until it is suitable. The associated "while" loop to perform this action can take a long time to run, and this is due to the computational burden of it.

Line search should not be used for stochastic gradient descent because the whole point of using stochastic gradient descent is to avoid computing the full gradient (based on the entire dataset) and to do it in batches instead. But by using line search to minimize the loss function, the full loss function needs to be calculated for each update, subsequently having the cost close to calculating the full first derivative. Using line search for stochastic gradient descent removes the benefit of stochastic gradient descent - not having to compute the full gradient!

Change of variables and decision variables

A change of variables can be used to enforce a constraint on the decision variables by making the constraint more simple. For example, if we are minimizing $d(x) = x^2$ with the constraint $h(x) = x - 3 = 0$, this is hard because $h(x) = 0$ is another linear equation we need to deal with. To make this simple, the variables can be changed to $y = x - 3$, so the problem is now to minimize $d(x) = (y + 3)^2$, subject to the constraint $h(y) = y = 0$. This constraint has become a lot simpler, since we just have to ensure that $y = 0$ when we minimize. After finding y, the original variables can be used with the inverse of what we did by transforming with $x = y + 3$.

Penalty to cost function and decision variables

A penalty can be added to the cost function to enforce a constraint on the decision variables. For example, if we are minimizing the same function as before $d(x) = x^2$, with the constraint $h(x) = x - 3 \leq 0$, the objective function can be changed to $F(x, \gamma) = f(x) + \gamma \cdot max(0, h(x))^2$ where γ is a penalty parameter (to control the importance of the constraint being met) and $max(0, h(x))$ makes sure that going against the constraint contributes to. the penalty term. As a consequence, if the constraint is met $(h(x) \leq 0)$, the penalty term is 0. But, if the constraint is not met, the penalty term increases with a quadratic rate. This makes the problem into minimizing $F(x, \gamma) = x^2 + \gamma \cdot max(0, x - 3)^2$, hence increasing γ (such as $\gamma = 1$) makes the algorithm minimize the violation of the constraint while maintaining the minimizing of the original objective function (penalty is bigger compared to $\gamma = 0.5$). Therefore, constraints are added without enforcing them strictly, making it useful to penalize violations to the constraint on the decision variables.

References

[1] Fashion MNIST (Keras)

[2] Custom tensorflow training loop format

[3] Adam vs. SGD: Closing the generalization gap on image classification (Gupta et al.)

Appendix

model.py

```
from keras import models, layers


def get_model(optimizer, loss_function):
    model = models.Sequential([
        layers.Input(shape=(28, 28, 1,)),
        layers.Conv2D(32, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation='relu'),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        # We avoid using "kernel_regularizer=regularizers.l1(0.0001)"
        below since we want overfitting in this case!
        layers.Dense(10)
    ])
    model.compile(optimizer=optimizer, loss=loss_function,
    metrics=['accuracy'])
    return model
```

custom_training_and_evaluation_steps.py

```
# Note: The initial custom training and evaluation step format this code
is based on is provided by tensorflow at:
#
https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch
import numpy as np
import tensorflow as tf


# Step size scheduler (decreases the step size after every epoch)
def step_size_scheduler(initial_learning_rate, epoch):
    return initial_learning_rate * np.exp(0.04 * epoch + 1)


# Custom training step (calculates the gradients, with the ability to add
Gaussian "noise")
@tf.function
def train_step(x, y, model, loss_function, optimizer,
training_accuracy_metric, standard_deviation_for_gaussian_noise,
               step_size_scheduler, epoch, initial_learning_rate):
    with tf.GradientTape() as tape:
        training_logits = model(x, training=True)
        loss_value = loss_function(y, training_logits)
        loss_value += sum(model.losses)
    gradients = tape.gradient(loss_value, model.trainable_weights)
    gradients_with_gaussian_noise = [
        gradient + tf.random.normal(gradient.shape, mean=0.0,
        stddev=standard_deviation_for_gaussian_noise) for gradient
        in gradients]
    if step_size_scheduler:
        optimizer.learning_rate =
        step_size_scheduler(initial_learning_rate, epoch)
    optimizer.apply_gradients(zip(gradients_with_gaussian_noise,
    model.trainable_weights))
    training_accuracy_metric.update_state(y, training_logits)
    return loss_value, gradients


# Define custom evaluation step
@tf.function
def eval_step(x, y, model, loss_function, validation_accuracy_metric):
    validation_logits = model(x, training=False)
    loss_value = loss_function(y, validation_logits)
    validation_accuracy_metric.update_state(y, validation_logits)
    return loss_value
```

sgd_run.py

```
# Note: The initial custom training and evaluation step format this code
is based on is provided by tensorflow at:
#
https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch
from math import floor

import tensorflow as tf
from tensorflow.python.keras import metrics
from keras import datasets, optimizers, losses
import numpy as np
from sklearn.model_selection import train_test_split

from custom_training_and_evaluation_steps import train_step, eval_step
from model import get_model



# One SGD run
def sgd_run(mini_batch_size=16, epochs=20, step_size=0.001,
standard_deviation_for_gaussian_noise=0,
            step_size_scheduler=False, mini_batch_scheduler=False,
            adam_optimizer=False):
    # The Fashion MNIST dataset has a good chance of overfitting
    (increased complexity and variability as in report)
    (x_train, y_train), (x_test, y_test) =
    datasets.fashion_mnist.load_data()
    combined_x_data = np.concatenate((x_train, x_test))
    combined_y_data = np.concatenate((y_train, y_test))
    # Convert pixel values between 0 and 1
    combined_x_data = combined_x_data.astype('float32') / 255.0
    # Add an extra dimension so the shape of the image suits the CNN
    combined_x_data = np.expand_dims(combined_x_data, -1)
```

```
# 80% of Fashion MNIST will be used for training
x_train, x_test_and_validation, y_train, y_test_and_validation =
train_test_split(combined_x_data, combined_y_data, test_size=0.2)

train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
train_dataset = train_dataset.shuffle(buffer_size=1024)

# 10% of Fashion MNIST will be used for testing, and 10% for
validation (not shared like cross-validation)
x_test, x_validation, y_test, y_validation =
train_test_split(x_test_and_validation, y_test_and_validation,
test_size=0.5)
validation_dataset = tf.data.Dataset.from_tensor_slices((x_validation,
y_validation))
batch_size_schedule = lambda epoch: max(floor(mini_batch_size - (0.01
* (epoch + 1) * mini_batch_size)), 8)

if not mini_batch_scheduler:
    train_dataset = train_dataset.batch(mini_batch_size)
    validation_dataset = validation_dataset.batch(mini_batch_size)

# Evaluation during training with sparse categorical crossentropy
(suits Fashion MNIST's categorical classification)
metric_for_training_accuracy = metrics.SparseCategoricalAccuracy()
metric_for_validation_accuracy = metrics.SparseCategoricalAccuracy()

if adam_optimizer:
    # Adam with defaults
    optimizer = optimizers.Adam(learning_rate=0.001, beta_1=0.9,
    beta_2=0.999, epsilon=1e-07)
else:
    # Optimizer will be stochastic gradient descent with the learning
    rate (i.e. step size) passed in
    optimizer = optimizers.SGD(learning_rate=step_size)

# Loss function will be sparse categorical crossentropy (suits Fashion
MNIST's categorical classification)
loss_function = losses.SparseCategoricalCrossentropy(from_logits=True)
cnn_model = get_model(optimizer, loss_function)
variances_of_gradients = []
training_accuracies = []
training_losses = []
validation_accuracies = []
validation_losses = []

# Train the CNN model with custom training/evaluation steps, instead
of cnn_model.fit(), so it can be tweaked
for epoch in range(epochs):
    print(f"Epoch number: {epoch}")
    batch_gradients = []

    if mini_batch_scheduler:
        # Update the batch size after evert epoch with the batch size
        scheduler
        batch_size_from_scheduler = batch_size_schedule(epoch)
```

```
            train_dataset = tf.data.Dataset.from_tensor_slices((x_train,
            y_train))
            train_dataset = train_dataset.shuffle(buffer_size=1024)
            train_dataset = train_dataset.batch(batch_size_from_scheduler)
            validation_dataset =
            tf.data.Dataset.from_tensor_slices((x_validation,
            y_validation))
            validation_dataset =
            validation_dataset.batch(batch_size_from_scheduler)

        # Iterating over the training dataset's batches
        for _, (training_batch_for_x, training_batch_for_y) in
        enumerate(train_dataset):
            training_loss, gradient = train_step(training_batch_for_x,
            training_batch_for_y,
            cnn_model,
            loss_function,
            optimizer,
            metric_for_training_accuracy,
            standard_deviation_for_gaussian_noise,
            step_size_scheduler,
            epoch,
            step_size)
            training_losses.append(training_loss)
            batch_gradients.append(np.var([np.var(g) for g in gradient]))

        variances_of_gradients.append(np.mean(batch_gradients))
        training_accuracy = metric_for_training_accuracy.result()
        training_accuracies.append(training_accuracy)
        metric_for_training_accuracy.reset_states()

        # Validation loop at the end of each epoch
        for validation_batch_for_x, validation_batch_for_y in
        validation_dataset:
            validation_loss = eval_step(validation_batch_for_x,
            validation_batch_for_y, cnn_model, loss_function,
                                          metric_for_validation_accuracy)
            validation_losses.append(validation_loss)
        validation_accuracy = metric_for_validation_accuracy.result()
        validation_accuracies.append(validation_accuracy)
        metric_for_training_accuracy.reset_states()

    # Once model is compiled and trained, evaluate the model
    test_loss, test_accuracy = cnn_model.evaluate(x_test, y_test,
    verbose=2)
    training_loss = float((sum(training_losses) / len(training_losses)))
    training_accuracy = float((sum(training_accuracies) /
    len(training_accuracies)))
    validation_loss = float((sum(validation_losses) /
    len(validation_losses)))
    validation_accuracy = float((sum(validation_accuracies) /
    len(validation_accuracies)))
    average_variance_in_gradient = float((sum(variances_of_gradients) /
    len(variances_of_gradients)))
    return test_loss, test_accuracy, training_loss, training_accuracy,
    validation_loss, validation_accuracy, average_variance_in_gradient
```

main.py

```
from matplotlib import pyplot as plt
from sgd_run import sgd_run
import tensorflow as tf


def experimenting_mini_batch_size_with_overfitting():
    # This command makes tensorflow use the CPU to avoid OOM (out of
    memory) errors using the GPU
    tf.config.set_visible_devices([], 'GPU')
    # Used [16, 1024, 4096, 16394, 32768, 56000] initially, but it was too
    big and had the same conclusions as below
    mini_batch_sizes = [16, 32, 64, 128, 256, 512, 1024]

    # Lists to store the results
    test_losses = []
    test_accuracies = []
    training_losses = []
    training_accuracies = []
    validation_losses = []
    validation_accuracies = []
    average_variances_in_gradient = []

    for mini_batch_size in mini_batch_sizes:
        print(f"Using mini-batch size: {mini_batch_size}")
        test_losses_from_runs = []
        test_accuracies_from_runs = []
        training_losses_from_runs = []
        training_accuracies_from_runs = []
        validation_losses_from_runs = []
        validation_accuracies_from_runs = []
        average_variances_in_gradient_from_runs = []
        run_attempts = [1, 2, 3]
        for _ in run_attempts:
            (test_loss, test_accuracy, training_loss, training_accuracy,
            validation_loss, validation_accuracy,
             average_variance_in_gradient) = (
                sgd_run(mini_batch_size=mini_batch_size, epochs=20,
                step_size=0.0001))
            test_losses_from_runs.append(test_loss)
            test_accuracies_from_runs.append(test_accuracy)
            training_losses_from_runs.append(training_loss)
            training_accuracies_from_runs.append(training_accuracy)
            validation_losses_from_runs.append(validation_loss)
            validation_accuracies_from_runs.append(validation_accuracy)
            average_variances_in_gradient_from_runs.append(
            average_variance_in_gradient)
        # Append results to lists
        test_losses.append(sum(test_losses_from_runs) /
        len(test_losses_from_runs))
        test_accuracies.append(sum(test_accuracies_from_runs) /
        len(test_accuracies_from_runs))
        training_losses.append(sum(training_losses_from_runs) /
        len(training_losses_from_runs))
        training_accuracies.append(sum(training_accuracies_from_runs) /
        len(training_accuracies_from_runs))
        validation_losses.append(sum(validation_losses_from_runs) /
        len(validation_losses_from_runs))
        validation_accuracies.append(sum(validation_accuracies_from_runs)
        / len(validation_accuracies_from_runs))
        average_variances_in_gradient.append(
            sum(average_variances_in_gradient_from_runs) /
            len(average_variances_in_gradient_from_runs))

    plt.figure(figsize=(10, 6))
    plt.plot(mini_batch_sizes, test_losses, label='Average test loss')
    plt.plot(mini_batch_sizes, training_losses, label='Average training
    loss')
    plt.plot(mini_batch_sizes, validation_losses, label='Average
    validation loss')
    plt.xlabel('Mini-batch size')
    plt.ylabel('Average loss')
    plt.title('Average loss vs Mini-batch size (20 epochs)')
    plt.legend()
    plt.savefig("plots/loss-for-mini-batch-size.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(mini_batch_sizes, test_accuracies, label='Average test
    accuracy')
    plt.plot(mini_batch_sizes, training_accuracies, label='Average
    training accuracy')
    plt.plot(mini_batch_sizes, validation_accuracies, label='Average
    validation accuracy')
    plt.xlabel('Mini-batch size')
    plt.ylabel('Average accuracy')
    plt.title('Average accuracy vs Mini-batch size (20 epochs)')
    plt.legend()
    plt.savefig("plots/accuracy-for-mini-batch-size.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(mini_batch_sizes, average_variances_in_gradient,
    label='Average variance in gradient across batches')
    plt.xlabel('Mini-batch size')
    plt.ylabel('Average variance in gradient across batches')
    plt.title('Average variance in gradient across batches vs Mini-batch
    size (20 epochs)')
    plt.legend()
    plt.savefig("plots/noise-for-mini-batch-size.png")
    plt.show()


def experimenting_gaussian_noise_with_overfitting():
```

```
# This command makes tensorflow use the CPU to avoid OOM (out of
memory) errors using the GPU
tf.config.set_visible_devices([], 'GPU')
gaussian_noises = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Lists to store the results
test_losses = []
test_accuracies = []
training_losses = []
training_accuracies = []
validation_losses = []
validation_accuracies = []
average_variances_in_gradient = []

for gaussian_noise in gaussian_noises:
    print(f"Using gaussian noise: {gaussian_noise}")
    test_losses_from_runs = []
    test_accuracies_from_runs = []
    training_losses_from_runs = []
    training_accuracies_from_runs = []
    validation_losses_from_runs = []
    validation_accuracies_from_runs = []
    average_variances_in_gradient_from_runs = []
    run_attempts = [1, 2, 3]
    for _ in run_attempts:
        (test_loss, test_accuracy, training_loss, training_accuracy,
        validation_loss, validation_accuracy,
         average_variance_in_gradient) = (
            sgd_run(mini_batch_size=16, epochs=20, step_size=0.0001,
            standard_deviation_for_gaussian_noise=gaussian_noise))
        test_losses_from_runs.append(test_loss)
        test_accuracies_from_runs.append(test_accuracy)
        training_losses_from_runs.append(training_loss)
        training_accuracies_from_runs.append(training_accuracy)
        validation_losses_from_runs.append(validation_loss)
        validation_accuracies_from_runs.append(validation_accuracy)
        average_variances_in_gradient_from_runs.append(
        average_variance_in_gradient)
    # Append results to lists
    test_losses.append(sum(test_losses_from_runs) /
    len(test_losses_from_runs))
    test_accuracies.append(sum(test_accuracies_from_runs) /
    len(test_accuracies_from_runs))
    training_losses.append(sum(training_losses_from_runs) /
    len(training_losses_from_runs))
    training_accuracies.append(sum(training_accuracies_from_runs) /
    len(training_accuracies_from_runs))
    validation_losses.append(sum(validation_losses_from_runs) /
    len(validation_losses_from_runs))
    validation_accuracies.append(sum(validation_accuracies_from_runs)
    / len(validation_accuracies_from_runs))
    average_variances_in_gradient.append(
        sum(average_variances_in_gradient_from_runs) /
        len(average_variances_in_gradient_from_runs))

plt.figure(figsize=(10, 6))
```

```
    plt.plot(gaussian_noises, test_losses, label='Average test loss')
    plt.plot(gaussian_noises, training_losses, label='Average training
    loss')
    plt.plot(gaussian_noises, validation_losses, label='Average validation
    loss')
    plt.xlabel('Standard deviation of gaussian "noise"')
    plt.ylabel('Average loss')
    plt.title('Average loss vs Standard deviation of gaussian "noise" (20
    epochs)')
    plt.legend()
    plt.savefig("plots/loss-for-noise.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(gaussian_noises, test_accuracies, label='Average test
    accuracy')
    plt.plot(gaussian_noises, training_accuracies, label='Average training
    accuracy')
    plt.plot(gaussian_noises, validation_accuracies, label='Average
    validation accuracy')
    plt.xlabel('Standard deviation of gaussian "noise"')
    plt.ylabel('Average accuracy')
    plt.title('Average accuracy vs Standard deviation of gaussian "noise"
    (20 epochs)')
    plt.legend()
    plt.savefig("plots/accuracy-for-noise.png")
    plt.show()


def experimenting_step_size_schedule_with_overfitting():
    # This command makes tensorflow use the CPU to avoid OOM (out of
    memory) errors using the GPU
    tf.config.set_visible_devices([], 'GPU')
    step_sizes = [0.0001, 0.001, 0.01, 0.1]
    # Lists to store the results
    test_losses = []
    test_accuracies = []
    training_losses = []
    training_accuracies = []
    validation_losses = []
    validation_accuracies = []
    average_variances_in_gradient = []

    for step_size in step_sizes:
        print(f"Using step size: {step_size}")
        test_losses_from_runs = []
        test_accuracies_from_runs = []
        training_losses_from_runs = []
        training_accuracies_from_runs = []
        validation_losses_from_runs = []
        validation_accuracies_from_runs = []
        average_variances_in_gradient_from_runs = []
        run_attempts = [1, 2, 3]
        for _ in run_attempts:
          (test_loss, test_accuracy, training_loss, training_accuracy,
          validation_loss, validation_accuracy,
           average_variance_in_gradient) = (
              sgd_run(mini_batch_size=16, epochs=20,
              step_size=step_size))
          test_losses_from_runs.append(test_loss)
          test_accuracies_from_runs.append(test_accuracy)
          training_losses_from_runs.append(training_loss)
          training_accuracies_from_runs.append(training_accuracy)
          validation_losses_from_runs.append(validation_loss)
          validation_accuracies_from_runs.append(validation_accuracy)
          average_variances_in_gradient_from_runs.append(
          average_variance_in_gradient)
      # Append results to lists
      test_losses.append(sum(test_losses_from_runs) /
      len(test_losses_from_runs))
      test_accuracies.append(sum(test_accuracies_from_runs) /
      len(test_accuracies_from_runs))
      training_losses.append(sum(training_losses_from_runs) /
      len(training_losses_from_runs))
      training_accuracies.append(sum(training_accuracies_from_runs) /
      len(training_accuracies_from_runs))
      validation_losses.append(sum(validation_losses_from_runs) /
      len(validation_losses_from_runs))
      validation_accuracies.append(sum(validation_accuracies_from_runs)
      / len(validation_accuracies_from_runs))
      average_variances_in_gradient.append(
          sum(average_variances_in_gradient_from_runs) /
          len(average_variances_in_gradient_from_runs))

plt.figure(figsize=(10, 6))
plt.plot(step_sizes, test_losses, label='Average test loss')
plt.plot(step_sizes, training_losses, label='Average training loss')
plt.plot(step_sizes, validation_losses, label='Average validation
loss')
plt.xlabel('Initial step size')
plt.ylabel('Average loss')
plt.title('Average loss vs Step size with scheduler (20 epochs)')
plt.legend()
plt.savefig("plots/loss-for-step-size.png")
plt.show()

plt.figure(figsize=(10, 6))
plt.plot(step_sizes, test_accuracies, label='Average test accuracy')
plt.plot(step_sizes, training_accuracies, label='Average training
accuracy')
plt.plot(step_sizes, validation_accuracies, label='Average validation
accuracy')
plt.xlabel('Initial step size')
plt.ylabel('Average accuracy')
plt.title('Average accuracy vs Step size with scheduler (20 epochs)')
plt.legend()
plt.savefig("plots/accuracy-for-step-size.png")
plt.show()
```

```
def experimenting_mini_batch_size_schedule_with_overfitting():
    # This command makes tensorflow use the CPU to avoid OOM (out of
    memory) errors using the GPU
    tf.config.set_visible_devices([], 'GPU')
    mini_batch_sizes = [16, 32, 64, 128, 256, 512, 1024]
    # Lists to store the results
    test_losses = []
    test_accuracies = []
    training_losses = []
    training_accuracies = []
    validation_losses = []
    validation_accuracies = []
    average_variances_in_gradient = []

    for mini_batch_size in mini_batch_sizes:
        print(f"Using mini_batch_size: {mini_batch_size}")
        test_losses_from_runs = []
        test_accuracies_from_runs = []
        training_losses_from_runs = []
        training_accuracies_from_runs = []
        validation_losses_from_runs = []
        validation_accuracies_from_runs = []
        average_variances_in_gradient_from_runs = []
        run_attempts = [1, 2, 3]
        for _ in run_attempts:
            (test_loss, test_accuracy, training_loss, training_accuracy,
            validation_loss, validation_accuracy,
             average_variance_in_gradient) = (
                sgd_run(mini_batch_size=mini_batch_size, epochs=20,
                step_size=0.01, mini_batch_scheduler=True))
            test_losses_from_runs.append(test_loss)
            test_accuracies_from_runs.append(test_accuracy)
            training_losses_from_runs.append(training_loss)
            training_accuracies_from_runs.append(training_accuracy)
            validation_losses_from_runs.append(validation_loss)
            validation_accuracies_from_runs.append(validation_accuracy)
            average_variances_in_gradient_from_runs.append(
            average_variance_in_gradient)
        # Append results to lists
        test_losses.append(sum(test_losses_from_runs) /
        len(test_losses_from_runs))
        test_accuracies.append(sum(test_accuracies_from_runs) /
        len(test_accuracies_from_runs))
        training_losses.append(sum(training_losses_from_runs) /
        len(training_losses_from_runs))
        training_accuracies.append(sum(training_accuracies_from_runs) /
        len(training_accuracies_from_runs))
        validation_losses.append(sum(validation_losses_from_runs) /
        len(validation_losses_from_runs))
        validation_accuracies.append(sum(validation_accuracies_from_runs)
        / len(validation_accuracies_from_runs))
        average_variances_in_gradient.append(
            sum(average_variances_in_gradient_from_runs) /
            len(average_variances_in_gradient_from_runs))
    plt.figure(figsize=(10, 6))
    plt.plot(mini_batch_sizes, test_losses, label='Average test loss')
    plt.plot(mini_batch_sizes, training_losses, label='Average training
    loss')
    plt.plot(mini_batch_sizes, validation_losses, label='Average
    validation loss')
    plt.xlabel('Initial mini-batch size')
    plt.ylabel('Average loss')
    plt.title('Average loss vs Mini-batch size with scheduler (20 epochs)')
    plt.legend()
    plt.savefig("plots/loss-for-scheduler-mini-batch.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(mini_batch_sizes, test_accuracies, label='Average test
    accuracy')
    plt.plot(mini_batch_sizes, training_accuracies, label='Average
    training accuracy')
    plt.plot(mini_batch_sizes, validation_accuracies, label='Average
    validation accuracy')
    plt.xlabel('Initial mini-batch size')
    plt.ylabel('Average accuracy')
    plt.title('Average accuracy vs Mini-batch size with scheduler (20
    epochs)')
    plt.legend()
    plt.savefig("plots/accuracy-for-scheduler-mini-batch.png")
    plt.show()


def
experimenting_step_size_and_mini_batch_size_schedule_with_overfitting():
    # This command makes tensorflow use the CPU to avoid OOM (out of
    memory) errors using the GPU
    tf.config.set_visible_devices([], 'GPU')
    run_attempts = [1, 2, 3]
    # Lists to store the results
    test_losses = []
    test_accuracies = []
    training_losses = []
    training_accuracies = []
    validation_losses = []
    validation_accuracies = []
    average_variances_in_gradient = []

    for attempt in run_attempts:
        print(f"Run number: {attempt}")
        test_losses_from_runs = []
        test_accuracies_from_runs = []
        training_losses_from_runs = []
        training_accuracies_from_runs = []
        validation_losses_from_runs = []
        validation_accuracies_from_runs = []
        average_variances_in_gradient_from_runs = []
        average_run_attempts = [1, 2, 3]
        for _ in average_run_attempts:
            (test_loss, test_accuracy, training_loss, training_accuracy,
            validation_loss, validation_accuracy,
             average_variance_in_gradient) = (
                sgd_run(mini_batch_size=32, epochs=20, step_size=0.01,
                step_size_scheduler=True,
                        mini_batch_scheduler=True))
            test_losses_from_runs.append(test_loss)
            test_accuracies_from_runs.append(test_accuracy)
            training_losses_from_runs.append(training_loss)
            training_accuracies_from_runs.append(training_accuracy)
            validation_losses_from_runs.append(validation_loss)
            validation_accuracies_from_runs.append(validation_accuracy)
            average_variances_in_gradient_from_runs.append(
            average_variance_in_gradient)
        # Append results to lists
        test_losses.append(sum(test_losses_from_runs) /
        len(test_losses_from_runs))
        test_accuracies.append(sum(test_accuracies_from_runs) /
        len(test_accuracies_from_runs))
        training_losses.append(sum(training_losses_from_runs) /
        len(training_losses_from_runs))
        training_accuracies.append(sum(training_accuracies_from_runs) /
        len(training_accuracies_from_runs))
        validation_losses.append(sum(validation_losses_from_runs) /
        len(validation_losses_from_runs))
        validation_accuracies.append(sum(validation_accuracies_from_runs)
        / len(validation_accuracies_from_runs))
        average_variances_in_gradient.append(
            sum(average_variances_in_gradient_from_runs) /
            len(average_variances_in_gradient_from_runs))

    plt.figure(figsize=(10, 6))
    plt.plot(run_attempts, test_losses, label='Average test loss')
    plt.plot(run_attempts, training_losses, label='Average training loss')
    plt.plot(run_attempts, validation_losses, label='Average validation
    loss')
    plt.xlabel('Mini-batch and step size scheduler run attempt')
    plt.ylabel('Average loss')
    plt.title('Average loss vs Mini-batch and step size scheduler (20
    epochs)')
    plt.legend()
    plt.xticks(range(1, 4))
    plt.savefig("plots/loss-for-scheduler-mini-batch-and-step-size.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(run_attempts, test_accuracies, label='Average test accuracy')
    plt.plot(run_attempts, training_accuracies, label='Average training
    accuracy')
    plt.plot(run_attempts, validation_accuracies, label='Average
    validation accuracy')
    plt.xlabel('Mini-batch and step size scheduler run attempt')
    plt.ylabel('Average accuracy')
    plt.title('Average accuracy vs Mini-batch and step size scheduler (20
    epochs)')
    plt.legend()
    plt.xticks(range(1, 4))
    plt.savefig("plots/accuracy-for-scheduler-mini-batch-and-step-size.png")
    plt.show()


def experimenting_adam_with_overfitting():
    # This command makes tensorflow use the CPU to avoid OOM (out of
    memory) errors using the GPU
    tf.config.set_visible_devices([], 'GPU')
    run_attempts = [1, 2, 3]
    # Lists to store the results
    test_losses = []
    test_accuracies = []
    training_losses = []
    training_accuracies = []
    validation_losses = []
    validation_accuracies = []
    average_variances_in_gradient = []

    tf.config.run_functions_eagerly(True)

    for attempt in run_attempts:
        print(f"Run number: {attempt}")
        test_losses_from_runs = []
        test_accuracies_from_runs = []
        training_losses_from_runs = []
        training_accuracies_from_runs = []
        validation_losses_from_runs = []
        validation_accuracies_from_runs = []
        average_variances_in_gradient_from_runs = []
        average_run_attempts = [1, 2, 3]
        for _ in average_run_attempts:
            (test_loss, test_accuracy, training_loss, training_accuracy,
            validation_loss, validation_accuracy,
             average_variance_in_gradient) = (
                sgd_run(mini_batch_size=16, epochs=20, step_size=None,
                adam_optimizer=True))
            test_losses_from_runs.append(test_loss)
            test_accuracies_from_runs.append(test_accuracy)
            training_losses_from_runs.append(training_loss)
            training_accuracies_from_runs.append(training_accuracy)
            validation_losses_from_runs.append(validation_loss)
            validation_accuracies_from_runs.append(validation_accuracy)
            average_variances_in_gradient_from_runs.append(
            average_variance_in_gradient)
        # Append results to lists
        test_losses.append(sum(test_losses_from_runs) /
        len(test_losses_from_runs))
        test_accuracies.append(sum(test_accuracies_from_runs) /
        len(test_accuracies_from_runs))
        training_losses.append(sum(training_losses_from_runs) /
        len(training_losses_from_runs))
        training_accuracies.append(sum(training_accuracies_from_runs) /
        len(training_accuracies_from_runs))
        validation_losses.append(sum(validation_losses_from_runs) /
        len(validation_losses_from_runs))
        validation_accuracies.append(sum(validation_accuracies_from_runs)
        / len(validation_accuracies_from_runs))
        average_variances_in_gradient.append(
            sum(average_variances_in_gradient_from_runs) /
            len(average_variances_in_gradient_from_runs))

    plt.figure(figsize=(10, 6))
    plt.plot(run_attempts, test_losses, label='Average test loss')
    plt.plot(run_attempts, training_losses, label='Average training loss')
    plt.plot(run_attempts, validation_losses, label='Average validation
    loss')
    plt.xlabel('Adam step size update attempts')
    plt.ylabel('Average loss')
    plt.title('Average loss vs Adam step size update attempts with default
    parameters (20 epochs)')
    plt.legend()
    plt.xticks(range(1, 4))
    plt.savefig("plots/loss-for-adam.png")
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(run_attempts, test_accuracies, label='Average test accuracy')
    plt.plot(run_attempts, training_accuracies, label='Average training
    accuracy')
    plt.plot(run_attempts, validation_accuracies, label='Average
    validation accuracy')
    plt.xlabel('Adam step size update attempts')
    plt.ylabel('Average accuracy')
    plt.title('Average accuracy vs Adam step size update attempts with
    default parameters (20 epochs)')
    plt.legend()
    plt.xticks(range(1, 4))
    plt.savefig("plots/accuracy-for-adam.png")
    plt.show()


if __name__ == "__main__":
    experimenting_mini_batch_size_with_overfitting()
    experimenting_gaussian_noise_with_overfitting()
    experimenting_step_size_schedule_with_overfitting()
    experimenting_mini_batch_size_schedule_with_overfitting()
    experimenting_step_size_and_mini_batch_size_schedule_with_overfitting()
    experimenting_adam_with_overfitting()
```