1a) For the first question I was ask to modify the mini-batch SGD implementation using pytorch.

```
class PolyakSGD(torch.optim.Optimizer):
  def __init__(self, params, epsilon=1e-8, f_star=0.0):
            ...#initialize parameters
  def step(self, closure=None):
    loss = closure() # Get current loss value
    grad_squared_norm = 0.0     # Calculate squared norm of the gradient
    for group in self.param_groups:       # Process all parameter groups
      epsilon = group['epsilon']
      f_star = group['f_star']
      for p in group['params']:
        d_p = p.grad.data    # Get gradients
        grad_squared_norm += torch.sum(d_p * d_p).item()# Accumulate squared norm
    step_size = (loss.item() - f_star) / (grad_squared_norm + epsilon)  # Calculate Polyak step size
    self.step_sizes.append(step_size) # Store step size for analysis
    for group in self.param_groups: # Apply updates using the computed step size
      for p in group['params']:
        p.data.add_(d_p, alpha=-step_size) # Update parameters
    return loss

#...  Generate quadratic dataset: y = 2x^2 + 1
loader = DataLoader(dataset, batch_size=32, shuffle=True)

class QuadModel(torch.nn.Module): # Model: Linear layer with inputs [x, x^2]
  def __init__(self):
    super().__init__()
    self.lin = torch.nn.Linear(2, 1)
  def forward(self, xp):
    return self.lin(xp)

# Optimizer and loss
optimizer = PolyakSGD(model.parameters(), f_star=0.0, epsilon=1e-8)
criterion = torch.nn.MSELoss()
... #initialize logging variables and other parameter
for epoch in range(epochs):
  last_loss = None
  for xb, yb in loader:
    xb, yb = xb.to(device), yb.to(device)
    def closure():
      optimizer.zero_grad()
      loss = criterion(model(xb), yb)
      loss.backward()
      return loss
    loss = optimizer.step(closure)
   ...#store loss and other variables
```

Following the proposed method by Loizou et al. (2021), I applied Polyak's stepsize at the level of each mini-batch by computing the step size based on the mini-batch loss and mini-batch gradient norm. I inherited the class torch.optim.Optimizer to implement my polyakSGD optimizer, in the step function we first get the loss from the closure function that zero's the gradient , calculates the MSE loss in this case and then computes the gradient using the .backward() function. And then for all the parameters in the function we calculate the grad squared norm and sum it up to then calculate the Polyak step size here (loss.item() - f_star) / (grad_squared_norm + epsilon) , this is a global polyak step size for all the parameters and we then use this to update all the parameters in model. Moving on from the class, in the main function we set the optimizer, the criterion function which is the MSE and then implement a

stochastic approach to training the model. For each epoch, we loop through all the batches from the dataloader and call the step function in the optimizer passing the closure function. To test my implementation function, we are going to use quadratic data on y= $2x^2$ + 1 to train a simple model with 2 neurons, keeping f* as 0 as regular practice. After running for 500 epochs this was the results I got.

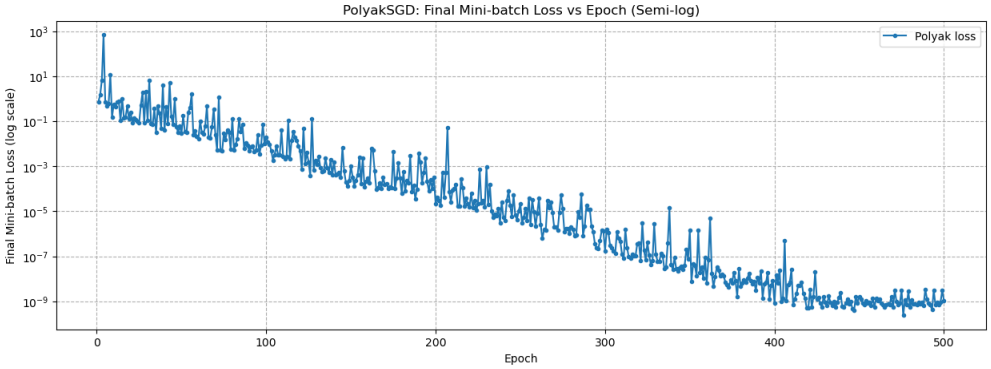
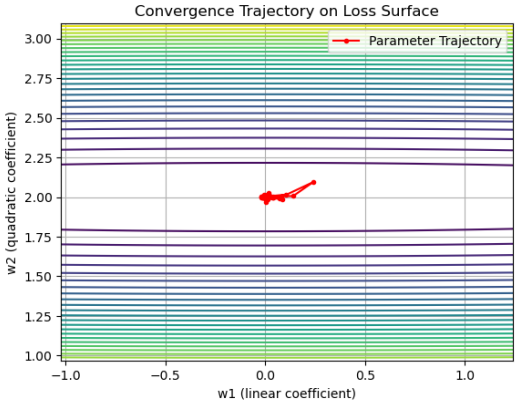


Figure 1: Loss convergence plot

Looking at figure 1 we can see that the PolyakSGD does converge, we can also see the noise due to the stochastic nature. The convergence can further be supported by looking at figure 2's



contour plot. We can see that the 2 parameters w1 and w2 do converge around the global minima for this function which is w1= 0, and w2= 2. From these two results we can conclude that the implementation matches the theoretical expectations from polyak step size.

Figure 2: Contour plot

1b) For this question I am going to generate synthetic noisy data on y= 3x+1 + N(0,σ^2) (added using noise_std * torch.randn_like(X)). And we are going to test our optimizers on a linear model with 1 neuron.

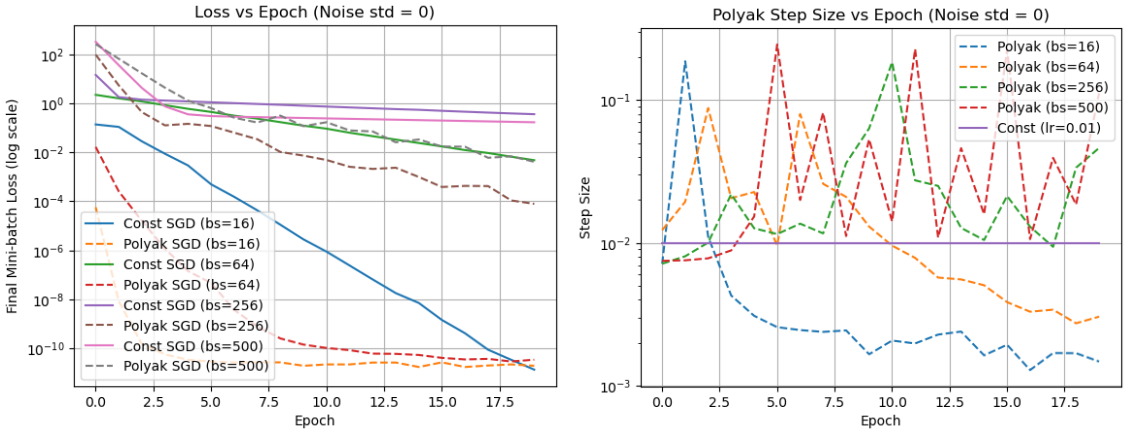


Figure 3: loss vs epoch and stepsize vs epoch plots for noise level 0

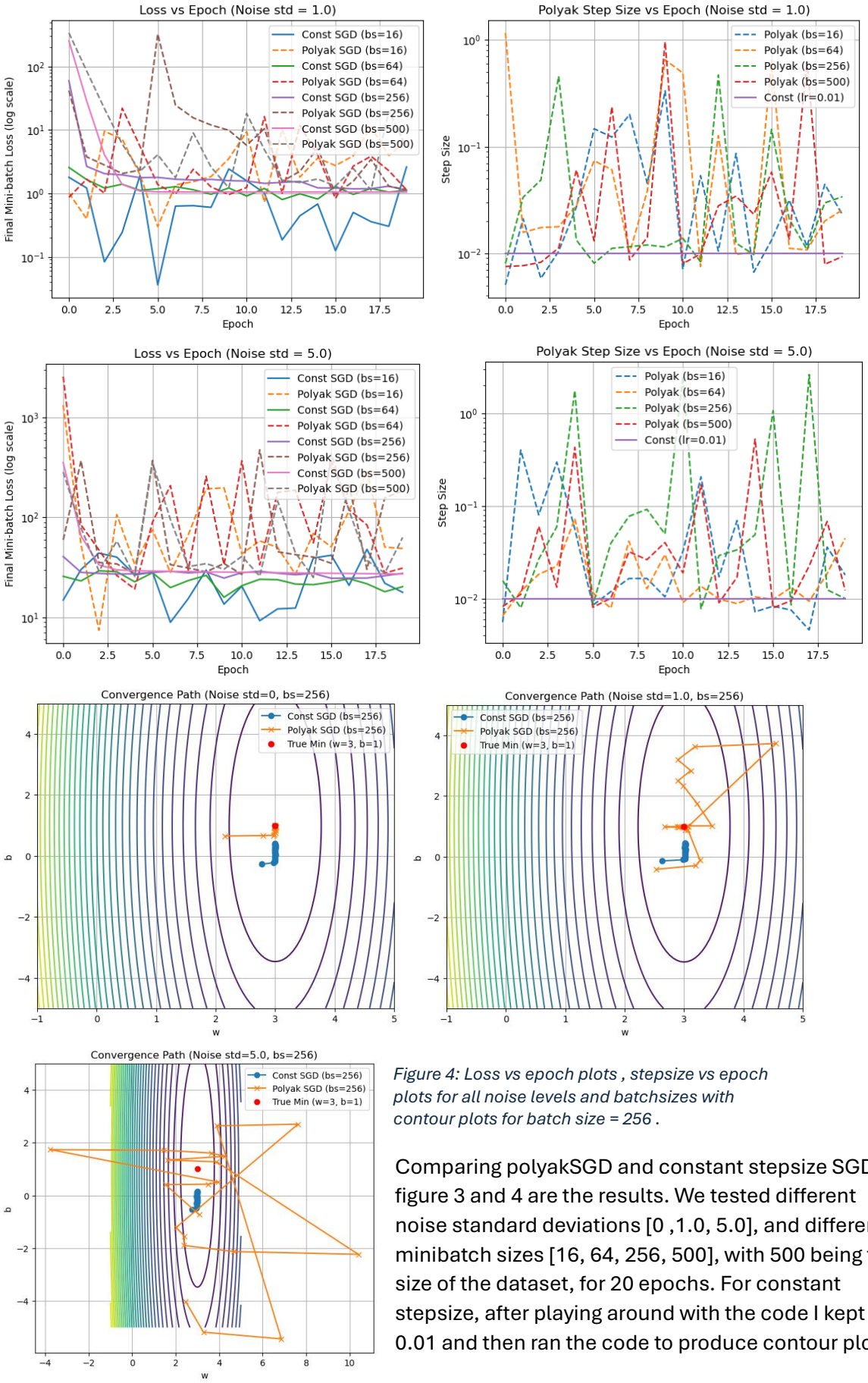


Figure 4: Loss vs epoch plots , stepsize vs epoch plots for all noise levels and batchsizes with contour plots for batch size = 256 .

Comparing polyakSGD and constant stepsize SGD figure 3 and 4 are the results. We tested different noise standard deviations [0 ,1.0, 5.0], and different minibatch sizes [16, 64, 256, 500], with 500 being the size of the dataset, for 20 epochs. For constant stepsize, after playing around with the code I kept it at 0.01 and then ran the code to produce contour plots

(remaining are in Appendix under Q1b) and loss vs epoch plots to see the difference in convergence and stepsize vs epoch plots. In the case of no noise, where σ = 0, we can see that Polyak converges much faster than constant SGD for small and medium batch sizes as it adapts sharply downwards over time allowing for fast learning as seen in the step size pot for batches 16 and 64. Increasing the noise to σ=1 we can see that the loss curves are noisier and it effects the constant step size as well for batch size 16. Meaning that the noise from SGD and random noise from the data causes it to be unstable, similarly polyak oscillates more that with no noise due to the same reason, and it converges at a higher loss value than with no noise for both methods. Adding more noise to σ=5 we can see the loss curves to be noisier as seen in the step size plot and in the contour plots. We can see that the polyak step size is powerful when the gradient estimate is reliable, although it requires batch size calibration due to noisy gradients. When there is no noise, and we have a small batch, Polyak adapts quickly and beats constant SGD in this case, but it becomes unstable as we increase noise and the results are hit or miss depends on the random nature of the training, constant step size sgd would take longer to converge in this case. And so if we have more noise in the data, a larger batch size will mitigate the instability in polyak's convergence as you will remove the noise added by SGD. However, these results are dependent to the function used, it may be the case that for other loss function, such noise from the SGD helps reach the global minima and not be stuck in a local minimum. In this case, when the model gets close to the minimum, the loss tends to shrink faster than the gradient does, so the Polyak step size often gets larger. This can be helpful at first, but with small batch sizes, the gradient estimates are noisy, which makes the step sizes jump around a lot and can cause unstable updates. Bigger batches help smooth things out by giving more reliable gradients, so the step size behaves better. In comparison, a well-chosen constant learning rate is more stable and predictable, but usually slower to converge. From the results, Polyak works better when noise is low or batch sizes are large, but struggles more in noisy, small-batch scenarios. In addition, Near the minimum, the loss becomes small while the gradient becomes even smaller, causing the Polyak step size to increase (since the denominator shrinks faster than the numerator). In the presence of SGD noise, this can lead to instability, especially with small batch sizes, because the larger step size amplifies the effect of noisy gradients.

1c) Using the 2-well function from week 6 I compared the PolyakSGD with constant size SGD.

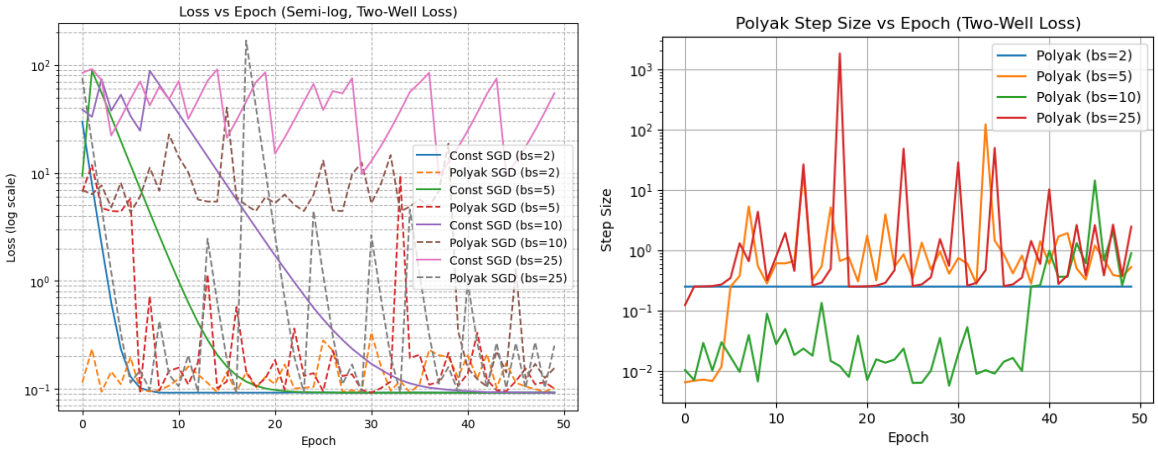


Figure 5: Loss vs epoch plot and stepsize vs epoch plot

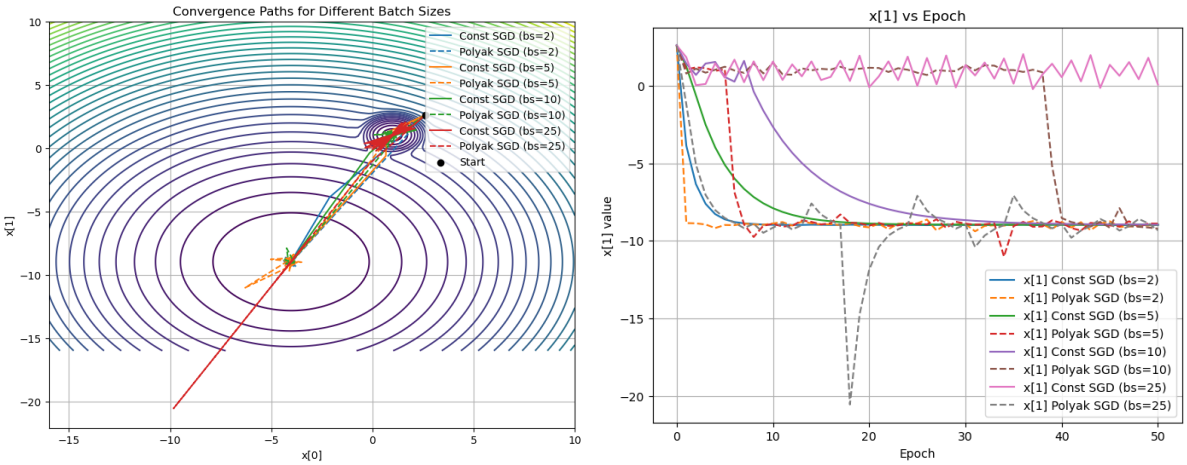


Figure 6: contour plot and variation in x[1] plot

For this question the code from week 6 generated 25 data points for the function seen in the contour plot in figure 6. Keeping f* at 0 as regular practice, and testing across different batch sizes [2,5,10,25] for both constant and polyak SGD, the results can be seen in figure 5 and 6. After training for 50 epochs, I can conclude that batch sizes required for both optimizers are around the same. Looking at the loss plot in figure 5 and the x variation plot in figure 6 we can see that for some batch sizes, the optimizers suffered by being stuck near the local minima. Constant SGD at batch size 25 was stuck in the local minima this can be explained by the lack of noise by the SGD since batch size 25 is the same size as the dataset which is the same as processing per epoch. For constant SGD as we increase the noise added by SGD by reducing batch size, it escaped the local minima quicker. Similarly, Polyak exploited the SGD noise at smaller batch sizes and escaped the local minima and adapted quickly. Keeping Batch size as 2 for this function fits best as polyak's loss is more stable and adapts more quicker to the global minima and converges quickest. Similarly for constant SGD, the SGD noise is needed to avoid/jump out the local minima.

1d) For this part of the assignment I used the transformer model from ML week9 and used the child speech dataset that was provided for training and evaluating. I kept the same number of embeddings = 384, head=6, layers=6. The total model parameters were around 10 million so I knew that training this model would take time especially if we were hyper tuning it. To get the best results, I had to first hyper tune the model, specifically the learning rate for constant SGD [0.01, 0.1, 0.5] and the learning rate [0.001, 0.01], beta 1 [0.9,0.95], beta2 [0.999, 0.9999] for adam, and batch sizes [16,32,64,128]. To track the progress while training I had to create wrappers for the pytorch adam and constant SGD functions to keep track of the step sizes, for example:

```
class ConstantSGD(torch.optim.SGD):
  def __init__(self, params, lr=0.01, momentum=0, dampening=0, weight_decay=0, nesterov=False):
    super(ConstantSGD, self).__init__(params, lr, momentum, dampening, weight_decay, nesterov)
    self.step_sizes = []
  def step(self, closure=None):
    lr = self.param_groups[0]['lr']
    self.step_sizes.append(lr)
    return super(ConstantSGD, self).step(closure)
```

the complete code can be seen in the Appendix under Q1d. I tuned the hyperparemeters keeping max_epoch as 10 due to the limited resources I had like computational power. After tuning, these were the results.

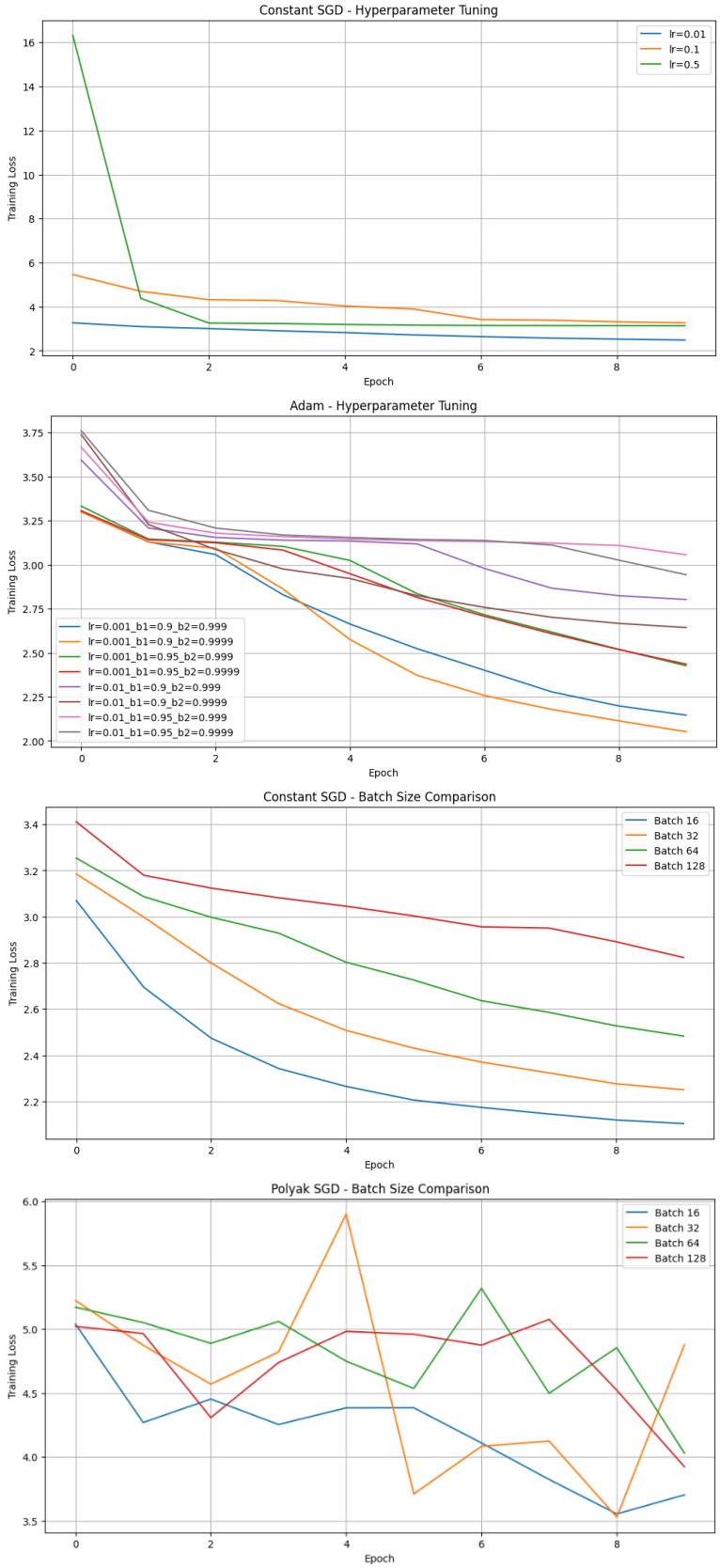


Figure 7: Hyperparameter tuning for Adam, Constant stepsize, and PolyakSGD (only batch size)

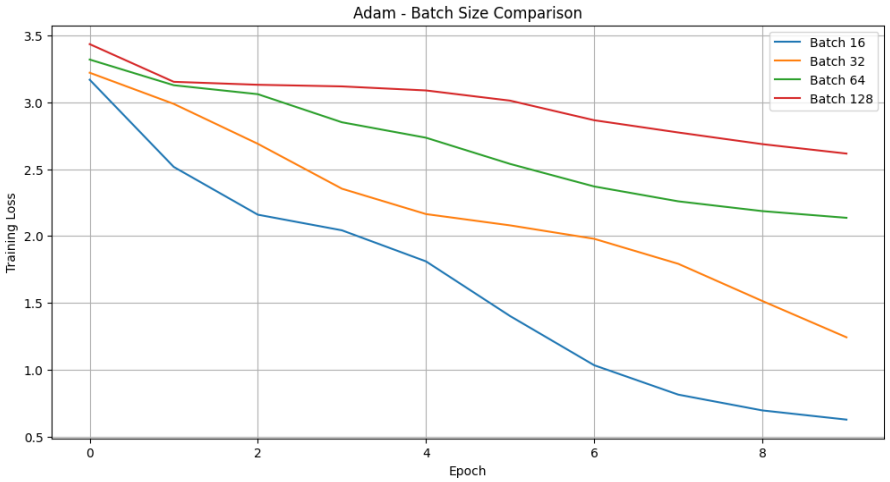


Figure 8: Hyperparameter tuning for Adam's batch size

I first tuned the parameters, excluding the batch size, and then substitutes the best parameters and tuned for the batch size. Looking at the results in figure 7 and 8 we can see that for constant SGD the best parameters that gave the fastest convergence lr=0.01, for adam it was lr=0.001, beta 1=0.9, beta 2=0.9999, and finally for all 3 optimizers, keeping batch size as 16 showed the fastest convergence, indicating that the noise from the stochastic nature helps in reaching the global minima. I then trained the 3 models with each optimizer for 200 epochs each, and this was these were the results.

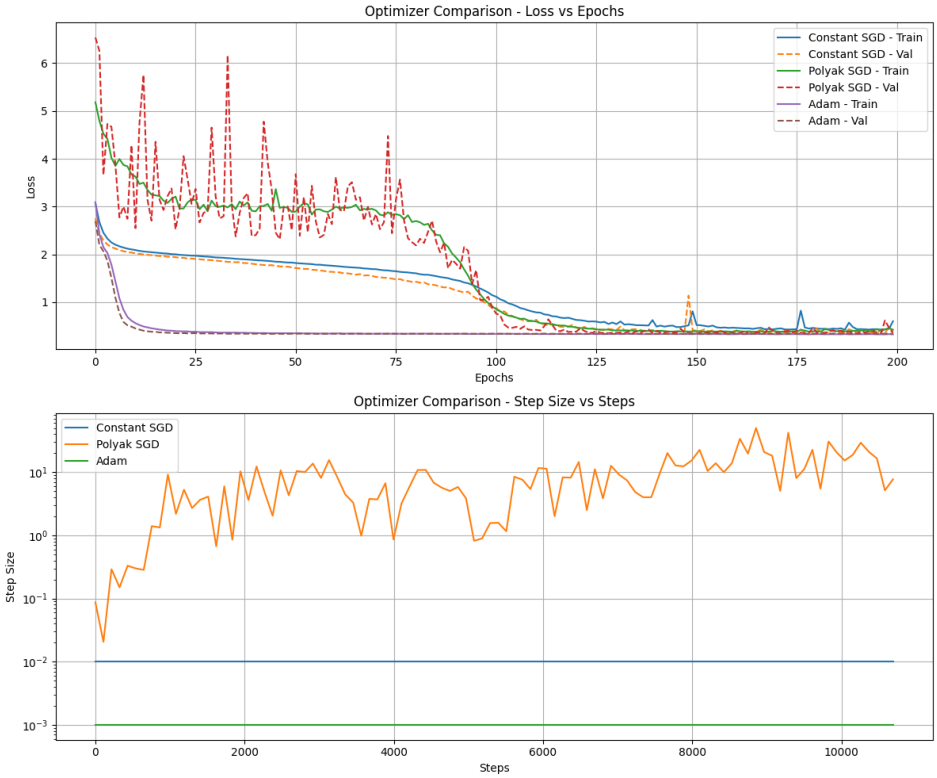


Figure 9: training results

Training took a lot of time due to limited resources; I could have reduced the parameters of the model by reducing the heads and layers, or the embeddings, either way it was a performance trade off. Looking at figure 9, by far Adam converged the quickest with no signs of overfitting. it's

adaptive learning rates for each parameter, along with momentum helped it converge quicker and smoothly. Moving on to polyak, it showed strong convergence, although it took time initially due to the oscillations with volatile behaviour as seen by the validation curve. As the gradient became smaller, we can see the step size increasing. This increase sometimes helped speed up learning but also introduced instability. In several runs, the step size grew too large, leading to overshooting or noisy updates which can be particularly problematic when using smaller batches or when validation noise was high. Lastly, constant SGD was stable but also slow to converge due to a fixed learning rate, it could not take advantage of the large gradients to adapt. In conclusion, each optimizer has trade-offs. Adam offers the best balance of speed and stability, making it a strong default choice. Polyak SGD can outperform others when batch sizes are large and noise is low, but its sensitivity to gradient norms means it requires more careful handling, however in this case it performed well with certain volatility. Constant SGD is simple and robust but lacks the dynamic behaviour needed for faster convergence, especially in complex or noisy scenarios.

```
--- Sample Outputs ---

[SGD]:

I love you Where ball
Read book No mine No like
My teddy Shoes on
I washands Saw big fluffy doggy at

[Polyak]:

I want cookie No like it
Shoes on I run fast
I want cookie I did it
I wash hands Daddy play
I want t

[Adam]:

I did it No like it
I dance Daddy play
More bubbles No touch
Bath time I jump high
Where ball I wash
```

I then generated child speeches using the 3 models trained and all 3 of them performed considerably decent, looking at figure 10 given the computational power limitation.

Figure 10: generating child speech

1e) For this question I impelemented polyak Adam and compared it with constant SGD, polyakSGD, and adam.

```
class PolyakAdam(torch.optim.Optimizer):
  def __init__(self, params, betas=(0.9, 0.999), eps=1e-8, f_star=0.0, weight_decay=0):
            #init parameters and class variables

  def step(self, closure=None):

    # Get current loss value
    loss = closure()

    # Calculate squared norm of the gradient (for Polyak step size)
    grad_squared_norm = 0.0

    for group in self.param_groups:
      for p in group['params']:
        grad = p.grad.data
        grad_squared_norm += torch.sum(grad * grad).item()
    # Calculate Polyak step size
    f_star = self.param_groups[0]['f_star']  # Assuming same f_star for all groups
    eps = self.param_groups[0]['eps']  # Using eps from Adam for numerical stability
    polyak_step_size = (loss.item() - f_star) / (grad_squared_norm + eps)
    polyak_step_size = max(polyak_step_size, 0.0)  # Ensure non-negative
```

```
for group in self.param_groups:       # Standard Adam update logic with Polyak step size
  beta1, beta2 = group['betas']

  for p in group['params']:
    grad = p.grad.data

    # State initialization
    state = self.state[p]
    if len(state) == 0:
      state['step'] = 0
      # Exponential moving average of gradient value
      state['exp_avg'] = torch.zeros_like(p.data)
      # Exponential moving average of squared gradient values
      state['exp_avg_sq'] = torch.zeros_like(p.data)

    state['step'] += 1

    # Update biased first moment estimate (momentum)
    state['exp_avg'].mul_(beta1).add_(grad, alpha=1 - beta1)
    # Update biased second raw moment estimate
    state['exp_avg_sq'].mul_(beta2).addcmul_(grad, grad, value=1 - beta2)

    # Bias correction
    bias_correction1 = 1 - beta1 ** state['step']
    bias_correction2 = 1 - beta2 ** state['step']

    # Original Adam would use a specified learning rate here
    # Instead, we use the Polyak step size

    # Compute Adam update direction (without step size)
    denom = (torch.sqrt(state['exp_avg_sq']) / np.sqrt(bias_correction2)).add_(eps)
    update = state['exp_avg'] / bias_correction1 / denom

    # Apply update using Polyak step size
    p.data.add_(update, alpha=-polyak_step_size)

return loss
```

Similarly like in Q1a I implemented a new class inheriting the optimizer from pytorch. The implementation is in the step function where we have classic Adam's approach where we calculate the moments and correct them and then update the weights for each parameter. But before that we calculate the alpha, or in this case, the Polyak step size by calculating the grad_norm and using the polyak equation to find a global polyak step size that we will use later in the mentioned Adam implementation. After comparing this implementation with other optimizers, this was the result at different noise levels training $y=x.w+b+\epsilon$, having 5 parameters here and 5 neurons in the network to train.

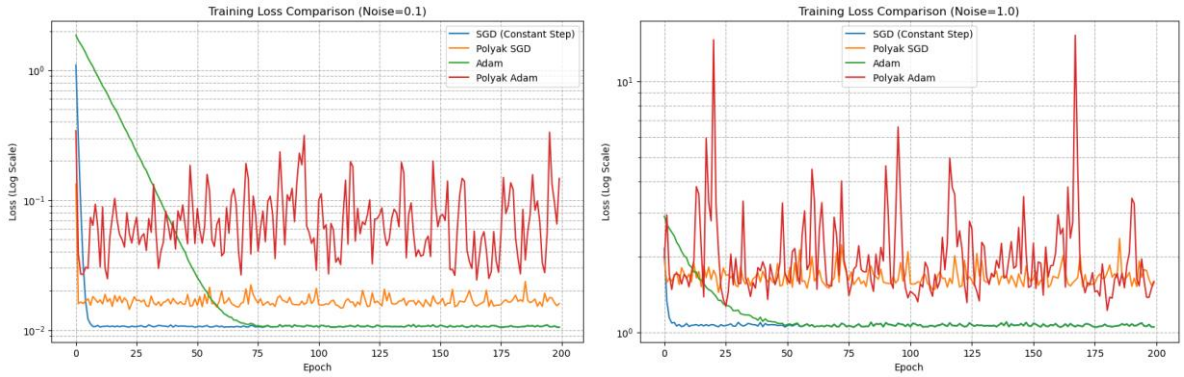


Figure 11: loss convergence for different optimizers under different noise

Looking at figure 10, Constant perform well and the quickest under both noise levels. Adam remains stable under both noisy conditions showing its robustness to noise due to internal

smoothing of updates. PolyakSGD shows slightly more fluctuation compared to the low-noise case as the higher noise causes less reliable gradients, making step sizes less predictable. The adaptive step size helps early on, but once the gradients get small, it may increase step sizes too aggressively, leading to oscillations. And for Polyak Adam,it is the worst performaer in both cases and as noise increased it becomes even more unstable. Its large spikes in loss suggest that combining two adaptive mechanisms (Adam's internal learning rate scaling and Polyak's external step size) leads to excessive update magnitudes in noisy conditions. Combining both polyak and adam introduces too many interacting dynamics where both are trying to adapt based on gradient information, and in noisy conditions this leads to erratic updates.

2a) Line search is a technique used in gradient descent to dynamically choose the step size at each iteration. Where instead of using a fixed step size, the algorithm searches along the direction of the negative gradient to find a step size that will decrease the loss function. A downside to this method is that it is very computationally expensive as it needs to evaluate the loss function multiple times per iteration, which is inefficient especially in high dimension problems, whereas it benefits from improving the convergence by automatically adapting the step size to the local curvature of the loss surface, avoiding issues with too small or too large learning rates. Furthermore, it should not be used with SGD as the loss gradients computed from mini-batches are noisy and inconsistent. This makes it difficult for line search to reliably find a good step size with gradients changing, along with the inefficiency of evaluating the loss function multiple times per iteration that offsets the benefits of a stochastic setting.

2b) we can enforce constraints through change of variables, allowing optimization in constrained spaces. Suppose we want to enforce the constraint that a variable $x \geq 0$. Instead of optimizing directly over x, we introduce a new variable $z \in R$ and define $x=e^z$. Since the exponential function is always positive, this guarantees that $x \geq 0$ for any real z. Now, the optimization is performed over z, an unconstrained space, and the constraint on x is automatically satisfied.

2c) A penalty method enforces constraints through modifying the original cost function to discourage violating the constraint. This is done by adding a penalty term that becomes large when the constraint is violated, effectively guiding the optimization toward a better solution. For example, if we want to minimize $f(x)=(x-2)^2$, subject to the constraint $x \geq 0$. We can reformulate this as: $F(x)=(x-2)^2 + \lambda \cdot \max(0,-x)^2$. Where λ is a penalty term and $\max(0,-x)^2$ is $\phi(g(x))$, a penalty function. As λ increases, the penalty term dominates when the constraint is violated, forcing the optimizer to satisfy the constraint more strictly. This penalizes any negative value of x, pushing the solution toward the feasible region $x \geq 0$.