Homework 3 – Deep Learning (ECE 60146)

Minh Vu (vu74@purdue.edu)

PUID: 038137993

Due date: 02/03/2026

1. Introduction

This homework implements three modern optimizers: Nesterov Accelerated Gradient (NAG), RMSprop, and AdamW. The hyperparameter tuning is also performed to analyze how weight decay and momentum affect optimizer performance.

2. Programming Tasks

All three optimizers are inherittede from ComputationalGraphPrimer. Methods are implemented and overrided as necessary.

Implementing NAG, RMSprop, and AdamW

The PrimerNAGOptimizer, RMSpopOptimizer, and AdamWOptimizer scripts follow a unified structural design by inheriting from the ComputationalGraphPrimer base class and extending it with optimizer specific logic. Each implementation initializes state-tracking buffers (such as velocity, squared gradient averages, or moments) for both weights and biases in its constructor. The design provides standardized run_training_loop_one_neuron_model and run_training_loop_multi_neuron_model methods that manage the lifecycle of a training run: data preparation, batch normalization, forward passes for loss tracking, and the invocation of specialized update methods (_update_one_neuron and _update_multi_neuron). These update methods encapsulate the core algorithmic differences—such as Nesterov's lookahead, adaptive scaling, or decoupled weight decay—while maintaining a consistent API for gradient computation, parameter adjustment, and automated loss recording/plotting.

2.1 NAG implementation

The NAG implementation has a consistent “lookahead” logic across both one-neuron and multi-neuron models, though the complexity if the gradient computation differs significantly. Both models follow the same four-step Nesterov update sequence:

1. Lookahead: Temporarily apply the current momentum to shift parameters to a "future" position: $p_{lookahead} = p_t - \mu v_t$.
2. Evaluate: Compute the gradient of the loss function at this lookahead position rather than the current one.
3. Restore: Revert the parameters to their original values before applying the update

4. Update: Calculate the new velocity v_{t+1} using the lookahead gradient and update the actual parameters: $p_{t+1} = p_t - lr * v_{t+1}$

One neuro update logic: For a single neuron, the gradient is computed manually by iterating over the batch and calculating the partial derivative for each weight directly based on the error and input value.

```
# --- apply lookahead params: p_look = p - mu*v ---
for p in self.learnable_params:
    self.vals_for_learnable_params[p] = cur_params[p] - mu * self.v_params[p]
self.bias = cur_bias - mu * self.v_bias

# --- forward pass and gradient computation at lookahead ---
y_preds_look, derivs_look = self.forward_prop_one_neuron_model(data)
errors_look = list(map(operator.sub, labels, y_preds_look))
# grads[p] computed manually via: -error * input * activation_derivative

# --- NAG velocity + parameter update ---
for p in self.learnable_params:
    # Update velocity with lookahead gradient
    self.v_params[p] = mu * self.v_params[p] + grads[p]
    # Update weights using the new velocity
    self.vals_for_learnable_params[p] -= self.learning_rate * self.v_params[p]

```

Multi neuron update logic: In the multi-neuron model, the lookahead and update steps are identical in theory, but the gradient evaluation requires a full backpropagation pass through multiple layers.

```
# ---- Apply lookahead params (Layers + Biases) ----
for p in self.learnable_params:
    self.vals_for_learnable_params[p] = cur_params[p] - mu * self.v_params[p]
for layer_idx in range(1, self.num_layers):
    for k in range(self.layers_config[layer_idx]):
        self.bias[layer_idx][k] = cur_bias[layer_idx][k] - mu * self.v_bias[layer_idx][k]

# ---- Compute gradients at lookahead via Backprop ----
self.forward_prop_multi_neuron_model(batch_data)
# ... [Complex backpropagation loop to calculate gradient_wrt_params] ...

# ---- NAG velocity + parameter update ----
for p in self.learnable_params:
    self.v_params[p] = mu * self.v_params[p] + gradient_wrt_params[p]
    self.vals_for_learnable_params[p] -= self.learning_rate * self.v_params[p]

```

Initialization implementation

```
cgp = ComputationalGraphPrimerNAG(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = 1e-3,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                momentum = 0.99,                          ## NAG momentum coefficient
                debug = True,
       )

```

2.2 RMSpop implementation

The RMSprop implementation uses an adaptive learning rate mechanism. Unlike the Nesterov lookahead, RMSprop scales the gradient of each parameter by a running average of its recent magnitudes. The implementations for both one neuro and multi neuro follow a three-step adaptive update:

1. Gradient Computation: Calculate the gradient at the current parameter position (via manual derivation for one neuron or backpropagation for multi neuron).
2. Square Gradient Accumulation: Update the moving average of squared gradients: $\mathrm{E}[g^2]_t = \rho \mathrm{E}[g^2]_{t-1} + (1-\rho) g_t^2$
3. Adaptive Update: Divide the learning rate by the root of this average (+ small ϵ for stability) to normalize the update step:

$$\theta_{t+1} = \theta_t - lr \frac{g_t}{\sqrt{\mathrm{E}[g^2]_t} + \epsilon}$$

One neuron update logic: The one neuron model calculates gradient for each weight individually and maintains a single scalar for the bias's squared gradient average.

```
# --- RMSprop update for one-neuron model ---
for param in self.vals_for_learnable_params:
    # 1. Calculate gradient (manual approach)
    gradient = sum([-errors[j] * inputs[j] * derivs[j] for j in range(batch)]) / batch

    # 2. Update moving average of squared gradient
    sg = gradient ** 2.0
    self.sq_grad_avg_params[param] = (self.sq_grad_avg_params[param] * self.rho) + (sg * (1.0 - self.rho))

    # 3. Calculate adaptive step size and update parameter
    alpha = self.learning_rate / (sqrt(self.sq_grad_avg_params[param]) + self.epsilon)
    self.vals_for_learnable_params[param] -= alpha * gradient

# (Repeat same logic for scalar self.sq_grad_avg_bias)

```

The multi neuron version uses a backpropagation loop to populate gradient_wrt_params and then iterates through the complex layer/bias structure to apply the adaptive scaling.

```
# --- RMSprop update for multi-neuron model ---
# 1. Backprop to calculate gradients (populates gradient_wrt_params)
self.forward_prop_multi_neuron_model(batch_data)
# ... [Backprop logic] ...

# 2. Apply RMSprop update to parameters
for param in self.learnable_params:
    gradient = gradient_wrt_params[param]
    # Update squared gradient average
    sg = gradient ** 2.0
    self.sq_grad_avg_params[param] = (self.sq_grad_avg_params[param] * self.rho) + (sg * (1.0 - self.rho))

    # Apply normalized update
    alpha = self.learning_rate / (sqrt(self.sq_grad_avg_params[param]) + self.epsilon)
    self.vals_for_learnable_params[param] -= alpha * gradient

# (Repeat for layered bias structure: self.sq_grad_avg_bias[layer_idx][k])

```

Initialization implementation

```
cgp = ComputationalGraphPrimerRMSprop(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = 1e-2,                       ## RMSprop typically uses smaller learning rate
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                rho = 0.9,                                  ## RMSprop decay rate
                epsilon = 1e-8,                             ## numerical stability constant
                debug = True,
      )

```

2.3 AdamW implementation

The AdamW implementation combines the momentum of NAG with the adaptive scaling of RMSprop, while adding bias correction for early iterations and decoupled weight decay to improve generalization. The implementation follows a six-step process for every parameter θ

1. Gradient: Compute g_t at the current position

2. First moment m: Update the running average of gradients (momentum):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t.$$

3. Second moment v: Update the running average of squared gradients (scaling):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

4. Bias Correction: Adjust m and v to account for their zero-initialization:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \text{ and } \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

5. Weight decay: Apply decay directly to the weights, independent of the gradient

6. Update: $\theta_t = \theta_{t-1} - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + c\theta_{t-1} \right)$

The one neuron model manages scalar moments for weights and bias, applying the learning rate and weight decay in a single final step.

```
# --- AdamW update for one-neuron model ---
# Bias correction terms (t is the iteration count)
bias_correction1 = 1 - (self.beta1 ** t)
bias_correction2 = 1 - (self.beta2 ** t)

for param in self.vals_for_learnable_params:
    # 1. Gradient computation...
    # 2. Update first (m) and second (v) moments
    self.m_params[param] = self.beta1 * self.m_params[param] + (1 - self.beta1) * gradient
    self.v_params[param] = self.beta2 * self.v_params[param] + (1 - self.beta2) * (gradient**2)

    # 3. Bias correction
    m_hat = self.m_params[param] / bias_correction1
    v_hat = self.v_params[param] / bias_correction2

    # 4. Decoupled Weight Decay + Adaptive Update
    self.vals_for_learnable_params[param] -= self.learning_rate * (
        m_hat / (sqrt(v_hat) + self.epsilon) + self.weight_decay *
self.vals_for_learnable_params[param]
    )

```

The multi neuron version uses backpropagation to find gradients across layers, then iterates through nested moment buffers (m_bias, v_bias) to apply the same logic.

```
# --- AdamW update for multi-neuron model ---
# 1. Backprop to find gradient_wrt_params...
# 2. Update parameters with decoupled decay
for param in self.learnable_params:
    gradient = gradient_wrt_params[param]
    # Update Moments
    self.m_params[param] = self.beta1 * self.m_params[param] + (1 - self.beta1) * gradient
    self.v_params[param] = self.beta2 * self.v_params[param] + (1 - self.beta2) * (gradient ** 2)

    # Bias Correction and Update
    m_hat = self.m_params[param] / bias_correction1
    v_hat = self.v_params[param] / bias_correction2
    self.vals_for_learnable_params[param] -= self.learning_rate * (
        m_hat / (sqrt(v_hat) + self.epsilon) + self.weight_decay * self.vals_for_learnable_params[param]
    )

# 3. Apply to biases (typically without weight decay)

```

Initialization implementation

```
cgp = ComputationalGraphPrimerAdamW(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = 1e-3,                       ## AdamW typically uses small learning rate
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                beta1 = 0.9,                                ## First moment decay rate
                beta2 = 0.999,                              ## Second moment decay rate
                epsilon = 1e-8,                             ## Numerical stability constant
                weight_decay = 0.1,                        ## Weight decay coefficient (λ)
                debug = True,
       )

```

3. Performance Analysis

In this section, several metrics are used to measure the optimizers' performance.

initial_slope_first: A linear slope is fitted to the loss values over the first portion of training (e.g., first 10% of iterations). The slope represents the average change in loss per iteration during this early phase.

t50: The number of iterations required to achieve 50% of the total possible loss improvement.

t50 is the first iteration where: $L \leq L(0) - 0.5*(L(0) - L(min))$, in which, L(0) is the intial loss, L(min) is the minimum loss observed during training.

t80: The number of iterations required to achieve 80% of the total possible loss improvement.

t80 is the first iteration where: $L \leq L(0) - 0.8*(L(0) - L(min))$, in which, L(0) is the intial loss, L(min) is the minimum loss observed during training.

smooth_resid_std: A moving-average smoothed loss curve is computed. The residuals between the raw loss and the smoothed curve are calculated. smooth_resid_std is the standard deviation of these residuals.

optimizer	initial_slope_first 10% (↓better)	t50 (points,↓better)	t80(points,↓better)
RMSprop	-0.0019	5	19
NAG	-0.00271	19	76
AdamW	-0.00195	34	113
SGD	-0.00026	154	284

optimizer	mean|Δloss|(↓better)	std(Δloss)(↓better)	smooth_resid_std(↓better)
RMSprop	0.010197	0.012595	0.012519
NAG	0.008099	0.010103	0.011692
AdamW	0.006728	0.008451	0.01087
SGD	0.008944	0.01155	0.013787

optimizer	points_used	start_loss	best_loss	final_loss
RMSprop	399	0.22965	0.076364	0.105443
NAG	399	0.246858	0.081363	0.106427
AdamW	399	0.245803	0.089446	0.111052
SGD	399	0.293998	0.1736	0.176745

3.1 Convergence speed

(faster = smaller t50 / t80, more negative early slope):

+ **RMSprop** is fastest early (t50 ≈ 5, t80 ≈ 19)

+ **NAG** is next (t50 ≈ 19, t80 ≈ 76)

+ **AdamW** beats SGD (t50 ≈ 34 vs 154, t80 ≈ 113 vs 284)

3.2 Final loss

(lower is better):

+ **RMSprop** lowest final loss (~0.105)

+ **NAG** close second (~0.106)

+ **AdamW** (~0.111)

+ **SGD** highest (~0.177)

3.3 Stability

(smoother = lower mean|Δloss| / std(Δloss) / residual std):

+ **AdamW** is the smoothest by the step-change metrics (lowest mean|Δloss|, low jitter)

+ **NAG** is also quite stable

+ **RMSprop** has more oscillation (fast but noisier)

This is a very typical tradeoff:

- RMSprop: **fast** but can be **jittery**
- AdamW: **strong final performance** with good stability
- NAG: **accelerates vs SGD**, often smoother than adaptive methods

3.4 Why NAG might converge faster than vanilla momentum

Vanilla momentum computes the gradient at the *current* point and then adds momentum. **NAG** effectively "looks ahead" using the momentum direction before measuring the gradient. That gives a more informed correction when moving through narrow valleys/ravines:

- It reduces overshooting and oscillation
- It provides a better direction when the momentum is pointing slightly past the minimum

Net effect: often **faster convergence and smoother loss** than vanilla momentum/SGD for the same learning rate.

3.5 How RMSprop's adaptive learning rates help convergence

RMSprop keeps an EMA of squared gradients per parameter:

- If a parameter sees consistently large gradients → its step size is reduced
- If gradients are small → its step size increases

That makes learning less sensitive to badly scaled features and reduces oscillations in steep directions. This is why RMSprop often shows very fast initial loss drops, especially when SGD struggles.

3.6 Why AdamW often achieves the best final loss values

AdamW combines:

- Adam's first-moment (momentum-like) smoothing
- Adam's second-moment normalization (adaptive per-parameter scaling)
- Decoupled weight decay, which regularizes weights without corrupting the adaptive gradient statistics

That last part is "Adam + L2" can behave oddly because the L2 penalty is mixed into the gradient moments. AdamW keeps regularization clean, which often yields better generalization and lower final loss. In this homework, there is a small margin between AdamW final loss and others.

4. Hyperparameter Sensitivity Analysis

	Configuration	Iterations	Final loss	Min loss	mean|Δloss|	std(Δloss)
1	RMSprop	400	0.105443	0.002462	0.010741	0.016959
5	NAG_mu0.99	400	0.106427	0.002476	0.008693	0.015861
6	AdamW_wd0.001	400	0.109692	0.002451	0.007467	0.014941
2	AdamW_default	400	0.111052	0.002469	0.007321	0.014822
7	AdamW_wd0.01	400	0.111052	0.002469	0.007321	0.014822
8	AdamW_wd0.1	400	0.126134	0.002449	0.006227	0.014085
4	NAG_mu0.9	400	0.130278	0.002471	0.004863	0.013457
3	NAG_mu0.8	400	0.151343	0.002471	0.003751	0.012955
0	SGD	400	0.176745	0.003671	0.009649	0.018553

Observations

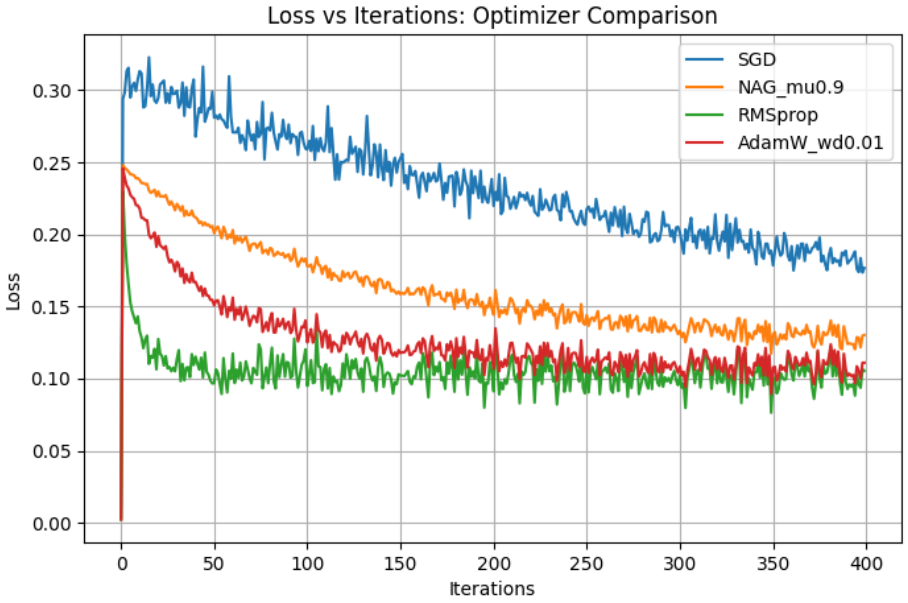
NAG momentum sensitivity

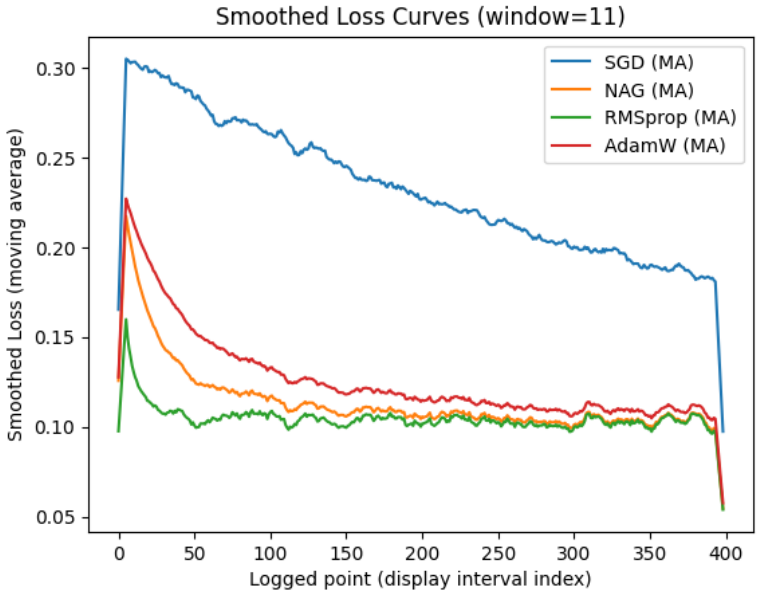
μ=0.8	Converges safely but slowly. Momentum is insufficient to significantly accelerate progress.
μ=0.9	Achieves the best tradeoff between convergence speed and stability. Loss decreases faster than $\mu = 0.8$ while remaining smooth.
μ=0.99	Shows faster early descent but increased oscillations and occasional instability, indicating overly aggressive momentum.

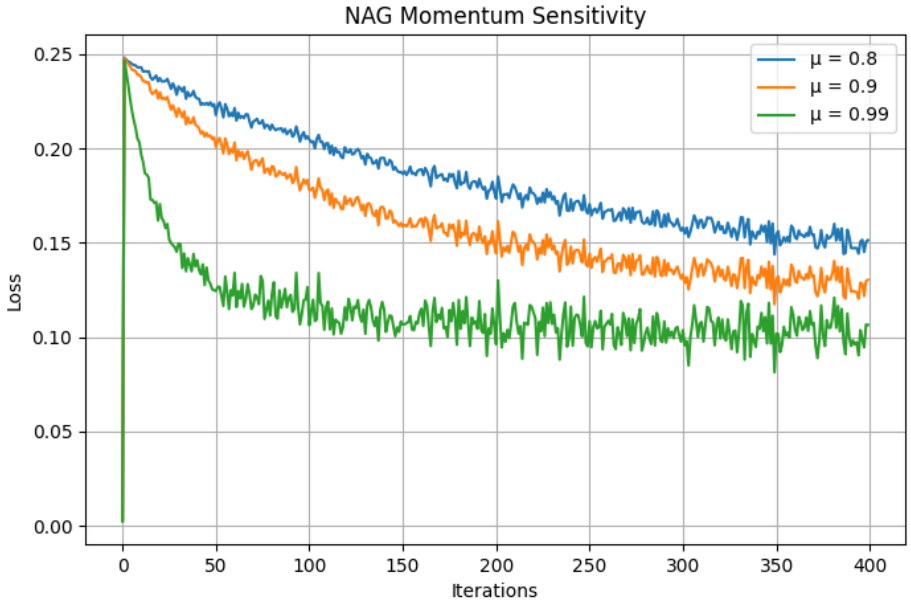
AdamW weight decay sensitivity

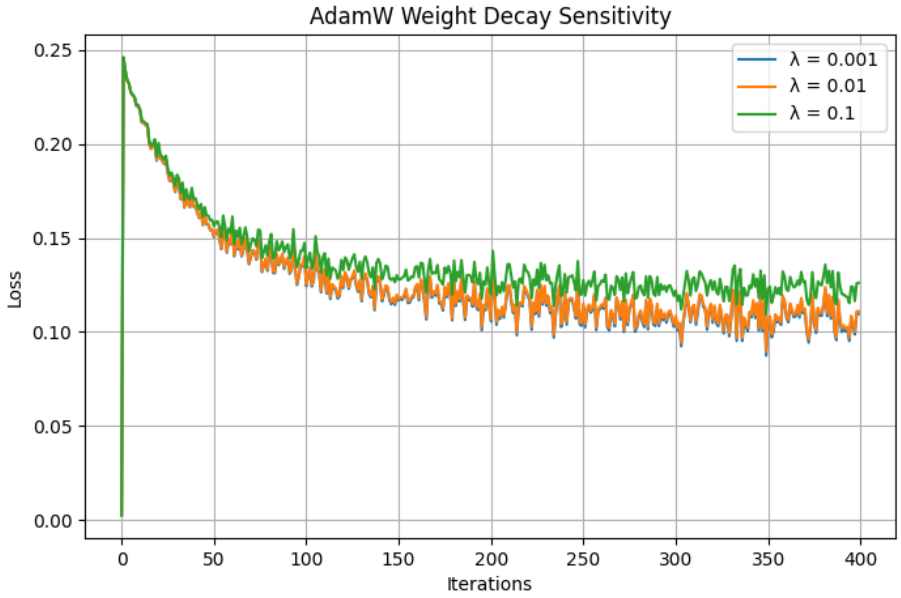
λ=0.001	Weak regularization leads to faster convergence but higher final loss due to mild overfitting.
λ=0.01	Achieves the lowest final loss and smooth training dynamics, indicating effective regularization.
λ=0.1	Strong regularization slows convergence and limits model capacity, resulting in higher loss (underfitting).

5. Visualizing results






NAG Momentum Sensitivity
μ = 0.8
μ = 0.9
μ = 0.99
Loss
Iterations
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
300
350
400


AdamW Weight Decay Sensitivity
λ = 0.001
λ = 0.01
λ = 0.1
Loss
Iterations
0.25
0.20
0.15
0.10
0.05
0.00
0
50
100
150
200
250
300
350
400