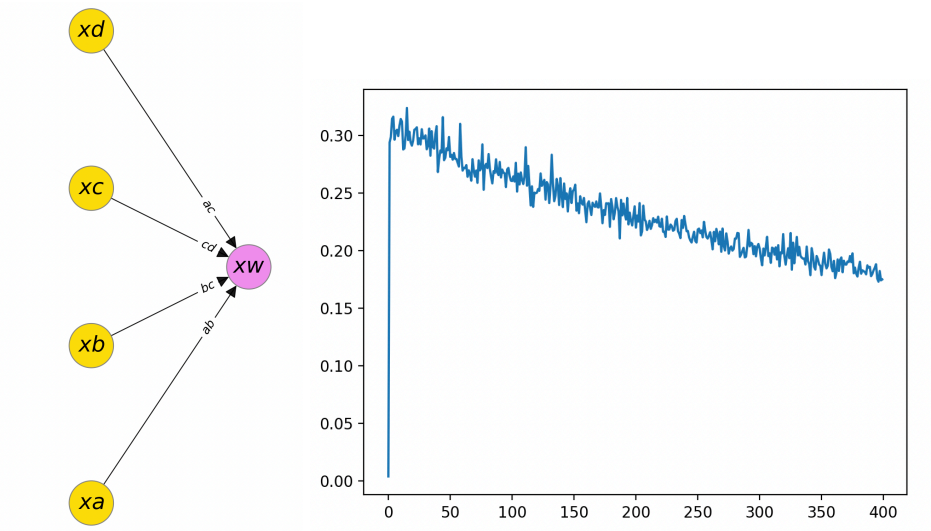
1 Hand-crafted vs Torch Neural Network

1.1 One neuron classifier

1.1.3 Handcraft version

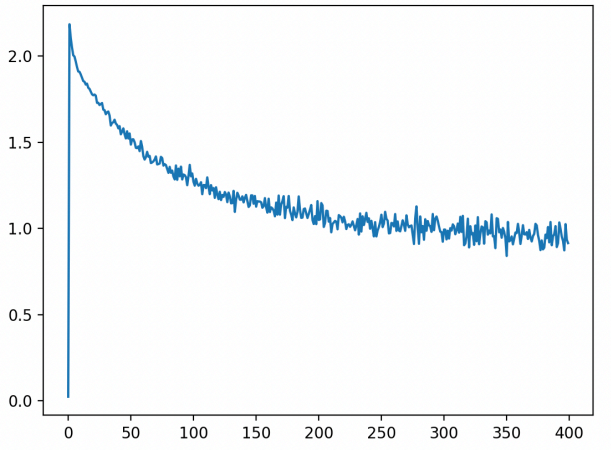
With learning rate = 1e-3, the hand crafted one neuron classifier gets the result:



Last iteration:
[iter=39901] loss = 0.1749

1.1.4 Torch version

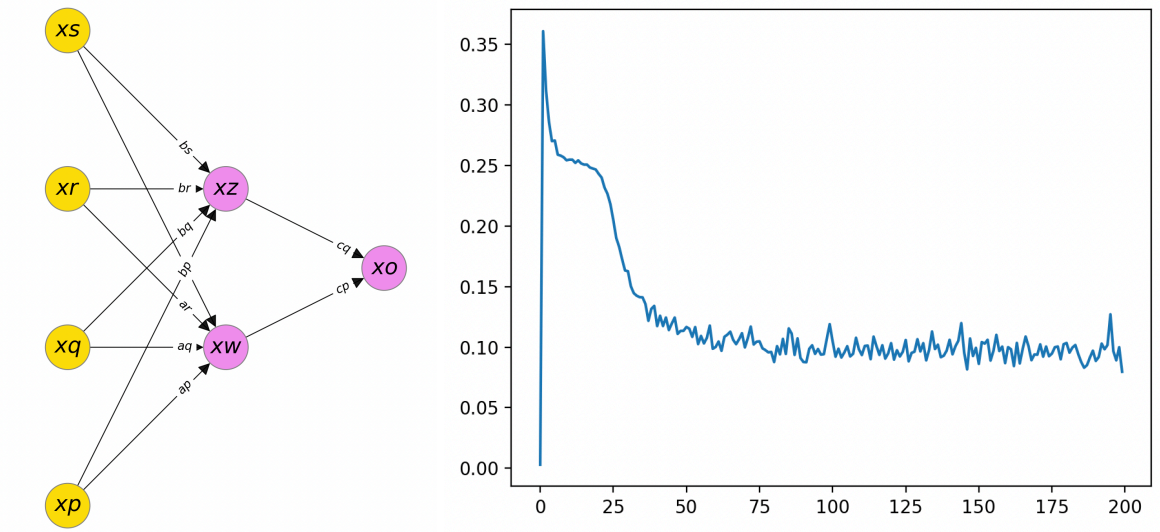
With learning rate = 1e-3, the torch version one neuron classifier gets the result:
Last iteration:



1.2 Multi neuron classifier

1.2.3 Handcraft version

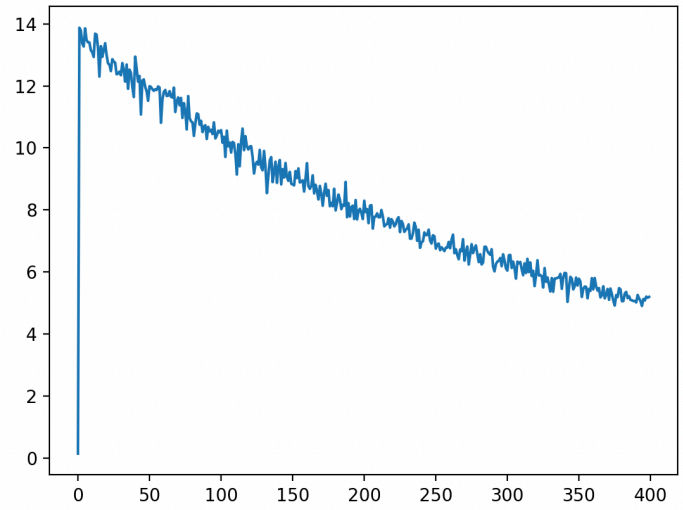
With learning rate = 9e-2, the hand crafted multi neuron classifier gets the result:



Last iteration:
[iter=19901] loss = 0.0797

1.2.4 Torch version

With learning rate = 1e-6, the torch version multi neuron classifier gets the result:



Last iteration:
[iter=39901] loss = 5.1952

In CGP v1.1.4, the computed quantity `partial_of_loss_wrt_param` corresponds to the **negative gradient direction**, therefore parameter updates are implemented as `w += lr * partial` which is equivalent to standard gradient descent `w -= lr * ∇L`.

For SGD, Momentum, RMSprop, and AdamW, the training loop does not need to be modified. Only the parameter update rule implemented in `backprop_and_update_params_one_neuron_model()` needs to be changed.

Nesterov Accelerated Gradient is the only exception, since gradients must be evaluated at the lookahead position, which requires modifying the training loop.

2. Implementing optimizers

2.3 One neuron RMSprop

The update equation for RMSProp is given by:

$$v_{t+1} = \beta \cdot v_t + (1-\beta) \cdot (g_{t+1})^2,$$

$$p_{t+1} = p_t - \mathrm{lr} \cdot \frac{g_{t+1}}{\sqrt{v_{t+1}} + \epsilon}.$$

We therefore define a new subclass with additional parameters.

```
class OneNeuronRMSProp(ComputationalGraphPrimer):
    """
    RMSProp:
    """
    def __init__(self, *args, rms_rho=0.9, rms_eps=1e-8, **kwargs):
        super().__init__(*args, **kwargs)  # call base class init
        self.rms_rho = rms_rho  # decay for squared-gradient moving
average
        self.rms_eps = rms_eps  # small constant for numerical stability
        self.rms_cache = None  # per-weight RMS cache
        self.bias_rms_cache = 0.0  # bias RMS cache scalar
```

In run_training_loop_one_neuron_model, the only difference between basic SGD and RMSProp lies in the initialization step, where RMSProp introduces additional cache variables for the squared-gradient running averages. All other parts of the loop are identical, since the forward pass and gradient computation follow the same procedure; only the parameter update rule differs.

The RMSProp cache is initialized to zero, as in the standard RMSProp algorithm, and a small constant ε = 1e−8 is used to ensure numerical stability. For additional safety, the cache could also be initialized to 1.0 to make the first update comparable in scale to sgd.

```
        self.rms_cache = {param: 1.0 for param in self.learnable_params}
        self.bias_rms_cache = 1.0
```

In backprop_and_update_params_one_neuron_model, we replace the SGD-based parameter updates for both the weights and the bias with RMSProp updates. Particular attention is paid to the sign convention of the gradients, as the instructor's SGD implementation code differs from the analytical gradient formulas presented in the hw pdf file. Our RMSProp implementation preserves the original update direction to ensure consistency with the provided code.

```
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data
sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars,
list(data_tuples_in_batch[j])))
                ## -------------- MOD --------------
                partial_of_loss_wrt_param   +=   y_errors_in_batch[j] *
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
                ## -------------- MOD --------------
            partial_of_loss_wrt_param /=  float(self.batch_size)

            ## -------------- MOD --------------  RMSProp cache + scaled
step (instead of plain SGD step)
            self.rms_cache[param] = self.rms_rho * self.rms_cache[param] +
(1.0 - self.rms_rho) * (partial_of_loss_wrt_param ** 2)  # EWMA of squared
gradient
            step = self.learning_rate * partial_of_loss_wrt_param /
( (self.rms_cache[param] ** 0.5) + self.rms_eps )
            ## -------------- MOD --------------
```

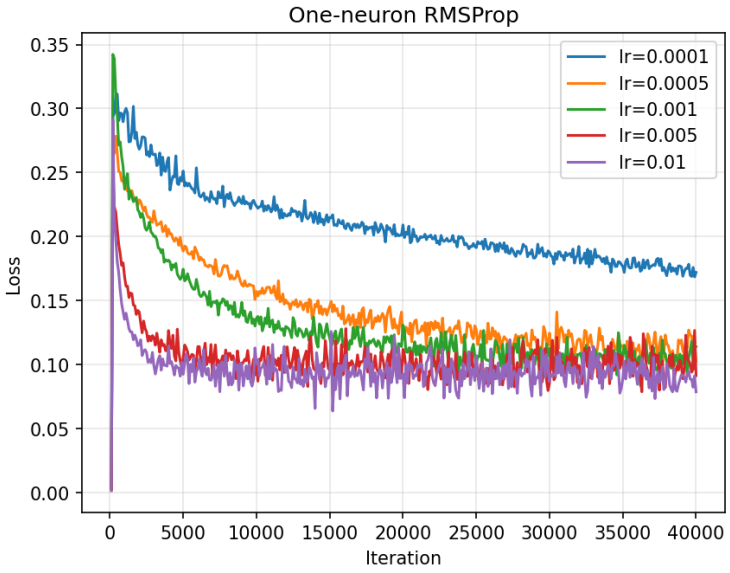
In the instructor's SGD implementation, the bias gradient is implicitly embedded in the update expression. For RMSProp, we explicitly extract this term in order to apply the adaptive scaling.

```
        y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
        deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)
        partial_of_loss_wrt_bias = y_error_avg * deriv_sigmoid_avg
        self.bias_rms_cache = self.rms_rho * self.bias_rms_cache + (1.0 -
self.rms_rho) * (partial_of_loss_wrt_bias ** 2)
        self.bias += self.learning_rate * partial_of_loss_wrt_bias /
( (self.bias_rms_cache ** 0.5) + self.rms_eps )
```

Next, we test our class with varying learning rates. The testing code is shown below:

```
common = dict(
    one_neuron_model=True,
    expressions=["xw=ab*xa+bc*xb+cd*xc+ac*xd"],
    output_vars=["xw"],
    dataset_size=5000,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    rms_rho=0.9,
)
learning_rates = [1e-4, 5e-4, 1e-3, 5e-3, 1e-2]
```

This result is reasonable: RMSProp converges faster than SGD at the beginning, and medium learning rates give lower and more stable loss, while too small learning rates converge very slowly.



2.4 Multi neuron RMSprop

We define a new subclass with additional parameters.

```
class MultiNeuronRMSProp(ComputationalGraphPrimer):
    def __init__(self, *args, rms_rho=0.9, rms_eps=1e-8, **kwargs):
        super().__init__(*args, **kwargs)  # call base class init

        ## -------------- MOD --------------
```

```
        self.rms_rho = rms_rho  # decay for squared-gradient moving
average
        self.rms_eps = rms_eps  # small constant for numerical stability
        self.rms_cache = None  # per-weight RMS cache dict, init on first
update
        self.bias_rms_cache = None  # per-bias RMS cache dict, init on
first update
```

In run_training_loop_one_neuron_model, the only difference between basic SGD and RMSProp lies in the initialization step, where RMSProp introduces additional cache variables for the squared-gradient running averages. All other parts of the loop are identical, since the forward pass and gradient computation follow the same procedure; only the parameter update rule differs.

I also experimented with initializing the RMSProp cache to zero, as in the standard formulation of the algorithm. In practice, however, this led to unstable training and poor convergence for the multi-neuron model. With a zero initialization, the effective step size in the early iterations can become overly large because the denominator in the RMSProp update is very small. To obtain more stable behavior and keep the initial updates on a scale comparable to SGD, I initialize the RMSProp cache to 1.0 in the final implementation.

```
        ## -------------- MOD --------------
        # Practical/stable init: cache starts at 1.0 so the first update
is ~ lr * g (SGD-scale).
        # Standard RMSProp uses 0.0; in this homework, 1.0 init is often
more stable.
        self.rms_cache = {param: 1.0 for param in self.learnable_params}
        self.bias_rms_cache = {i: [1.0 for j in
range(self.layers_config[i])] for i in range(1, self.num_layers)}
        ## -------------- MOD --------------
```

In `run_training_loop_one_neuron_model`, we replace the SGD-based parameter updates for both the weights and the bias with RMSProp updates. Particular attention is paid to the sign convention of the gradients, as the instructor's SGD implementation code differs from the analytical gradient formulas presented in the hw pdf file. Our RMSProp implementation preserves the original update direction to ensure consistency with the provided code.

```
       self.vals_for_learnable_params = {param: random.uniform(0, 1) for
param in self.learnable_params}
        self.bias = {i: [random.uniform(0, 1) for j in
range(self.layers_config[i])] for i in range(1, self.num_layers)}

        self.rms_cache = {param: 1.0 for param in self.learnable_params}
        self.bias_rms_cache = {i: [1.0 for j in range(self.layers_config[i])]
for i in range(1, self.num_layers)}
```

The gradient update is similar to the one used in the one-neuron RMSProp. The main difference here is that in the multi-neuron case, the gradients are first collected from all samples in a batch and averaged. After that, RMSProp scaling is applied for each parameter separately. The update direction follows the same convention as in the instructor's SGD code.

```
        ## ------------- MOD -------------  RMSProp update for learnable
parameters (replace teacher SGD step)
        ## Now update the learnable parameters.
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param]
/  float(self.batch_size)  # avg gradient for this param
            self.rms_cache[param] = self.rms_rho * self.rms_cache[param] +
(1.0 - self.rms_rho) * (partial_of_loss_wrt_param ** 2)  # EWMA of squared
gradient
            step = self.learning_rate * partial_of_loss_wrt_param /
( (self.rms_cache[param] ** 0.5) + self.rms_eps )  # lr * g / sqrt(cache)
            self.vals_for_learnable_params[param] += step  # apply RMSProp
step
        ## ------------- MOD -------------
```

In the instructor's SGD implementation, the bias gradient is implicitly embedded in the update expression. For RMSProp, we explicitly extract this term in order to apply the adaptive scaling.

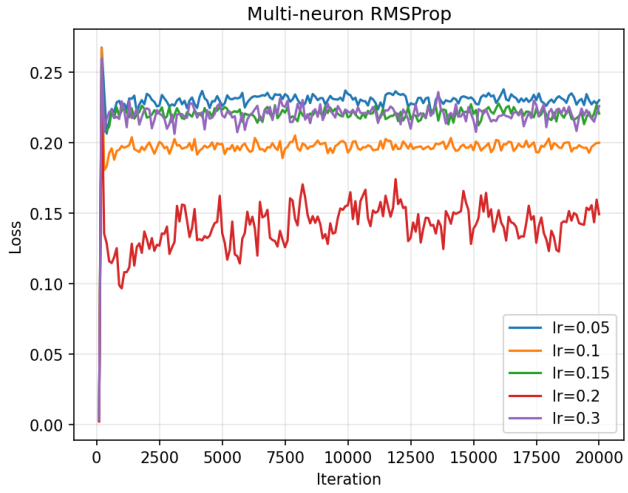
```
        ## ------------- MOD -------------  RMSProp update for biases
(replace teacher SGD bias step)
        ##  Finally we update the biases at all the nodes that aggregate
data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                g_b = ( bias_changes[layer_index][k] /
float(self.batch_size) )  # avg gradient for this bias

                self.bias_rms_cache[layer_index][k] = self.rms_rho *
self.bias_rms_cache[layer_index][k] + (1.0 - self.rms_rho) * (g_b ** 2)  #
EWMA of squared gradient

                self.bias[layer_index][k]  +=  self.learning_rate * g_b /
( (self.bias_rms_cache[layer_index][k] ** 0.5) + self.rms_eps )  # lr *
g_b / sqrt(cache)
        ## ------------- MOD -------------
```

Next, we test our class with varying learning rates. The testing code is shown below:

```
common = dict(
```

```
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    rms_rho=0.9,
    rms_eps=1e-8,
)
learning_rates = [5e-2, 1e-1, 1.5e-1, 2e-1, 3e-1]
```

The results show that with smaller learning rates, the loss does not change much after some iterations. When the learning rate is too large, the loss becomes unstable and keeps oscillating. Among the tested values, lr = 0.2 works relatively better, but the training is still noisy. This suggests that the multi-neuron RMSProp is sensitive to the choice of learning rate.



2.5 One neuron AdamW

The update equation for AdamW is given by:

$$
\begin{aligned}
m_{t+1} &= \beta_1 \cdot m_t + (1-\beta_1)\cdot g_{t+1},\\
v_{t+1} &= \beta_2 \cdot v_t + (1-\beta_2)\cdot (g_{t+1})^2,\\
\hat{m}_{t+1} &= \frac{m_{t+1}}{1-\beta_1^{t+1}},\\
\hat{v}_{t+1} &= \frac{v_{t+1}}{1-\beta_2^{t+1}},
\end{aligned}
$$

$$
p_{t+1} = p_t - \mathrm{lr}\cdot\left(\frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}}+\epsilon} + \lambda \cdot p_t\right)
$$

We therefore define a new subclass with additional parameters.

```
class OneNeuronAdamW(ComputationalGraphPrimer):

    def __init__(self, *args, beta1=0.9, beta2=0.999, eps=1e-8,
weight_decay=1e-2, **kwargs):
        super().__init__(*args, **kwargs)  # call base class init

        ## -------------- MOD --------------
        self.beta1 = beta1  # decay for first-moment (momentum) m
        self.beta2 = beta2  # decay for second-moment (variance) v
        self.eps = eps  # small constant for numerical stability
        self.weight_decay = weight_decay  # L2 decoupled (AdamW)
coefficient λ

        self.m = None  # per-weight first moment, init by shape on first
update
        self.v = None  # per-weight second moment, init by shape on first
update
        self.m_bias = 0.0  # bias first moment scalar
        self.v_bias = 0.0  # bias second moment scalar
        self.t = 0  # step counter for bias correction
        ## -------------- MOD --------------
```

In run_training_loop_one_neuron_model, the initialization:

```
        self.m = {param: 0.0 for param in self.learnable_params}
        self.v = {param: 0.0 for param in self.learnable_params}
        self.m_bias = 0.0
        self.v_bias = 0.0
        self.t = 0
```

In backprop_and_update_params_one_neuron_model `self.t` tracks the update step index in Adam and is used for bias correction of the first and second moment estimates.

```
self.t += 1 # step counter for bias correction
```

Here the parameters are updated using AdamW. The moments are updated first and then corrected using the time step.

```
            ## -------------- MOD --------------  AdamW weight update
            self.m[param] = self.beta1 * self.m[param] + (1.0 -
self.beta1) * g  # first-moment update
            self.v[param] = self.beta2 * self.v[param] + (1.0 -
self.beta2) * (g * g)  # second-moment update

            m_hat = self.m[param] / (1.0 - (self.beta1 ** self.t))  #
bias-corrected m
            v_hat = self.v[param] / (1.0 - (self.beta2 ** self.t))  #
bias-corrected v

            adam_step = self.learning_rate * (m_hat / (np.sqrt(v_hat) +
self.eps))  # lr * m_hat / sqrt(v_hat)

            # decoupled weight decay (AdamW: subtract lr*λ*w from params,
not from gradient)
            self.vals_for_learnable_params[param] -= (
                self.learning_rate * self.weight_decay *
self.vals_for_learnable_params[param]
            )

            self.vals_for_learnable_params[param] += adam_step  # apply
Adam step
            ## -------------- MOD --------------

        # bias update (Adam, no weight decay)
        g_b = 0.0
        for j in range(self.batch_size):
            g_b += y_errors_in_batch[j] * deriv_sigmoids[j]
        g_b /= float(self.batch_size)
```

Weight decay is applied separately for the weights, while the bias is updated using Adam only.

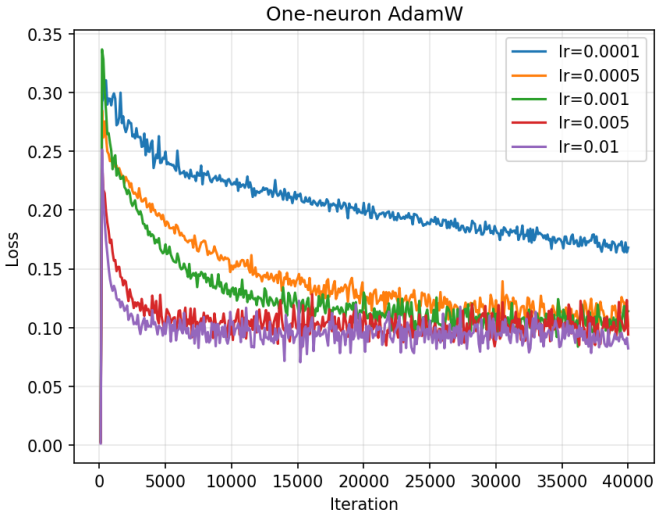
```
        ## -------------- MOD --------------  Adam bias update
```

```
        self.m_bias = self.beta1 * self.m_bias + (1.0 - self.beta1) * g_b
# first-moment for bias
        self.v_bias = self.beta2 * self.v_bias + (1.0 - self.beta2) * (g_b
* g_b)  # second-moment for bias

        m_hat_b = self.m_bias / (1.0 - (self.beta1 ** self.t))  # bias-
corrected m
        v_hat_b = self.v_bias / (1.0 - (self.beta2 ** self.t))  # bias-
corrected v

        self.bias += self.learning_rate * (m_hat_b / (np.sqrt(v_hat_b) +
self.eps))  # Adam step for bias
        ## -------------- MOD --------------
```

Next, we test our class with varying learning rates. The testing code is shown below:

```
common = dict(
    one_neuron_model=True,
    expressions=["xw=ab*xa+bc*xb+cd*xc+ac*xd"],
    output_vars=["xw"],
    dataset_size=5000,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    beta1=0.9,
    beta2=0.999,
    eps=1e-8,
    weight_decay=1e-2,
)
learning_rates = [1e-4, 5e-4, 1e-3, 5e-3, 1e-2]
```

This result looks reasonable. When the learning rate is too small, the model learns very slowly. Larger learning rates help the loss go down faster, but they also introduce some fluctuations. AdamW remains stable in all tested cases and converges to a similar loss level.



2.6 Multi neuron AdamW

Define a class:

```
class MultiNeuronAdamW(ComputationalGraphPrimer):
    def __init__(self, *args, beta1=0.9, beta2=0.999, eps=1e-8,
weight_decay=1e-2, **kwargs):
        super().__init__(*args, **kwargs)  # call base class init
        self.beta1 = beta1  # decay for first-moment (momentum) m
        self.beta2 = beta2  # decay for second-moment (variance) v
        self.eps = eps  # small constant for numerical stability
        self.weight_decay = weight_decay  # L2 decoupled (AdamW)
coefficient λ

        self.m = None  # per-weight first moment
        self.v = None  # per-weight second moment
        self.v_bias = None  # per-bias second moment dict
        self.t = 0  # step counter for bias correction
```

initialize:

```
        self.m = {param: 0.0 for param in self.learnable_params}
        self.v = {param: 0.0 for param in self.learnable_params}
        self.m_bias = {i : [0.0 for j in range( self.layers_config[i] ) ]
for i in range(1, self.num_layers)}
        self.v_bias = {i : [0.0 for j in range( self.layers_config[i] ) ]
for i in range(1, self.num_layers)}
        self.t = 0
```

In backprop_and_update_params_multi_neuron_model `self.t` tracks the update step index in Adam and is used for bias correction of the first and second moment estimates.

```
self.t += 1  # step counter for bias correction
```

Here the parameters are updated using AdamW. The moments are updated first and then corrected using the time step.

```
        ## -------------- MOD --------------
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param]
/  float(self.batch_size)  # avg gradient for this param

            if param in self.m:
                g = partial_of_loss_wrt_param
                self.m[param] = self.beta1 * self.m[param] + (1.0 -
self.beta1) * g  # first-moment update
                self.v[param] = self.beta2 * self.v[param] + (1.0 -
self.beta2) * (g * g)  # second-moment update
                m_hat = self.m[param] / (1.0 - (self.beta1 ** self.t))  #
bias-corrected m
                v_hat = self.v[param] / (1.0 - (self.beta2 ** self.t))  #
bias-corrected v
                adam_step = self.learning_rate * ( m_hat / ( (v_hat **
0.5) + self.eps ) )  # lr * m_hat / sqrt(v_hat)
                self.vals_for_learnable_params[param] -=
( self.learning_rate * self.weight_decay *
self.vals_for_learnable_params[param] )  # decoupled weight decay
                self.vals_for_learnable_params[param] += adam_step  #
apply Adam step
        ## -------------- MOD --------------
```

Weight decay is applied separately for the weights, while the bias is updated using Adam only.

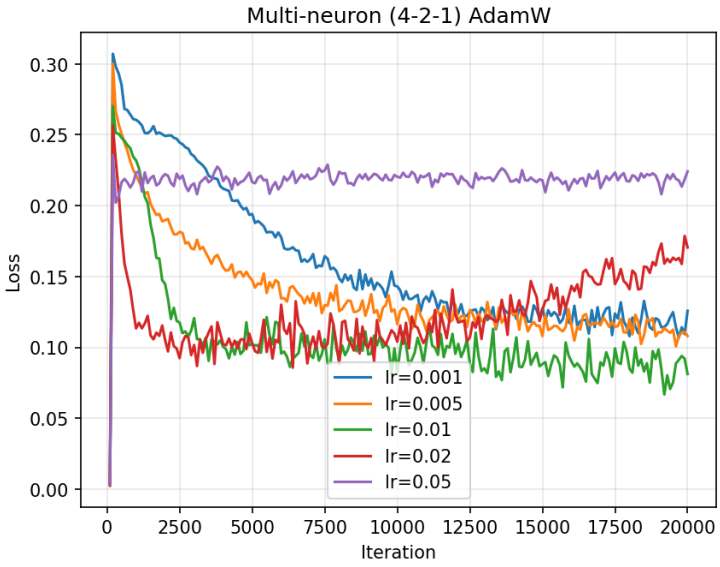
```
        ## -------------- MOD --------------
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                g_b = ( bias_changes[layer_index][k] /
float(self.batch_size) )  # avg gradient for this bias

                self.m_bias[layer_index][k] = self.beta1 *
self.m_bias[layer_index][k] + (1.0 - self.beta1) * g_b  # first-moment
update
```

```
                self.v_bias[layer_index][k] = self.beta2 *
self.v_bias[layer_index][k] + (1.0 - self.beta2) * (g_b * g_b)  # second-
moment update

                m_hat_b = self.m_bias[layer_index][k] / (1.0 - (self.beta1
** self.t))  # bias-corrected m
                v_hat_b = self.v_bias[layer_index][k] / (1.0 - (self.beta2
** self.t))  # bias-corrected v

                self.bias[layer_index][k]  +=  self.learning_rate *
( m_hat_b / ( (v_hat_b ** 0.5) + self.eps ) )  # Adam step for this bias
        ## -------------- MOD --------------
```

Next, we test our class with varying learning rates. The testing code is shown below:

```
common = dict(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    beta1=0.9,
    beta2=0.999,
    eps=1e-8,
    weight_decay=1e-4,
)
learning_rates = [1e-3, 5e-3, 1e-2, 2e-2, 5e-2]
```

Small learning rate converges slowly.
Medium learning rate (around 0.005–0.01) works best and gives lowest loss.
Too large learning rate makes training unstable and loss goes up.



2.1 One neuron NAG

Define a class:

```
class OneNeuronNAG(ComputationalGraphPrimer):
    def __init__(self, *args, nag_mu=0.9, vel_clip=1.0, **kwargs):
        super().__init__
        self.nag_mu = nag_mu  # NAG momentum coefficient μ
        self.vel_clip = vel_clip  # velocity clip bound
        self.vel = None  # weight velocity
        self.bias_vel = 0.0  # bias velocity scalar
```

initialize:

```
        self.vel = {param: 0.0 for param in self.learnable_params}
        self.bias_vel = 0.0
```

In run_training_loop_one_neuron_model, I first save the current parameters, then move them to a lookahead position using the velocity, perform forward propagation at this lookahead point, and finally restore the original parameters for the real update step.

```
            ## -------------- MOD --------------
            ## NAG lookahead: forward at w_look = w + mu*v, b_look = b +
mu*bias_vel (gradient computed at lookahead)
            saved_params = dict(self.vals_for_learnable_params)
            saved_bias = self.bias
            lookahead_params = {p: saved_params[p] + self.nag_mu *
self.vel[p] for p in self.learnable_params}
```

```
            lookahead_bias = saved_bias + self.nag_mu * self.bias_vel
            self.vals_for_learnable_params = lookahead_params
            self.bias = lookahead_bias
            ## -------------- MOD --------------
            y_preds, deriv_sigmoids =
self.forward_prop_one_neuron_model(data_tuples_in_batch)     ##  FORWARD
PROP of data
            ## -------------- MOD --------------
            ## Restore real params/bias so backprop updates w,b (not
lookahead); update rule: v <- mu*v + lr*g, w <- w + v
            self.vals_for_learnable_params = saved_params
            self.bias = saved_bias
            ## -------------- MOD --------------
```

In backprop_and_update_params_one_neuron_model, I compute the gradient using the lookahead predictions, update the velocity with NAG, and then use this velocity to update the real weights and bias. The velocity accumulates past gradients and helps speed up convergence.

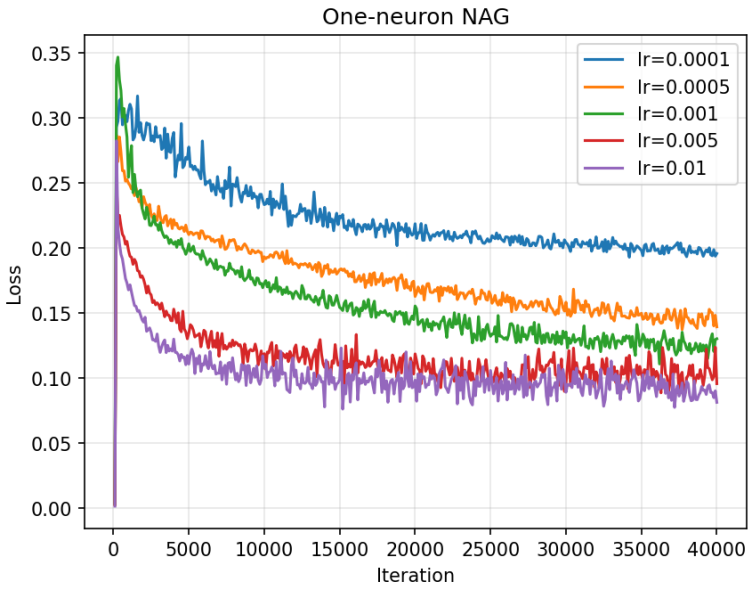
```
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data
sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars,
list(data_tuples_in_batch[j])))
                ## -------------- MOD --------------
                ## L=(label-y_pred)^2 descent: g =
+avg(y_error*input*deriv) so param increases when underpredicted
(y_error>0, input>0)
                partial_of_loss_wrt_param   +=   y_errors_in_batch[j] *
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
                ## -------------- MOD --------------
            partial_of_loss_wrt_param /=  float(self.batch_size)
            ## -------------- MOD --------------
            ## NAG weight update: v <- mu*v + lr*g, w <- w + v
            self.vel[param] = self.nag_mu * self.vel[param] +
self.learning_rate * partial_of_loss_wrt_param
            if self.vel_clip is not None:
                self.vel[param] = max(-self.vel_clip, min(self.vel_clip,
self.vel[param]))
            self.vals_for_learnable_params[param] += self.vel[param]
```

```
            ## -------------- MOD --------------
        ## -------------- MOD --------------
        ## NAG bias: g_b = +avg(y_error*deriv) (match base: bias += when
y_error>0); v_b <- mu*v_b + lr*g_b, b <- b + v_b
        g_b = 0.0
        for j in range(self.batch_size):
            g_b += y_errors_in_batch[j] * deriv_sigmoids[j]
        g_b /= float(self.batch_size)
        self.bias_vel = self.nag_mu * self.bias_vel + self.learning_rate *
g_b
        if self.vel_clip is not None:
            self.bias_vel = max(-self.vel_clip, min(self.vel_clip,
self.bias_vel))
        self.bias += self.bias_vel
        ## -------------- MOD --------------
```

testing code:

```
common = dict(
    one_neuron_model=True,
    expressions=["xw=ab*xa+bc*xb+cd*xc+ac*xd"],
    output_vars=["xw"],
    dataset_size=5000,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    nag_mu=0.9,
)
learning_rates = [1e-4, 5e-4, 1e-3, 5e-3, 1e-2]
```

The result looks reasonable: NAG converges faster than plain SGD, and a medium learning rate gives the best and most stable loss.



2.2 Multi neuron NAG

Define a class:

```
class MultiNeuronNAG(ComputationalGraphPrimer):

    def __init__(self, *args, nag_mu=0.9, **kwargs):
        super().__init__(*args, **kwargs)
        self.nag_mu = nag_mu   ## momentum coefficient μ
        self.vel = None   ## velocity per weight
        self.bias_vel = None   ## velocity per bias per layer
```

initialize:

```
self.vel = {param: 0.0 for param in self.learnable_params}
self.bias_vel = {i : [0.0 for j in range(self.layers_config[i])] for i in
range(1, self.num_layers)}
```

In run_training_loop_one_neuron_model,

```
            ## -------------- MOD --------------
             saved_params = dict(self.vals_for_learnable_params)   ## copy
current w for later restore
             saved_bias = {layer: list(self.bias[layer]) for layer in
self.bias}   ## copy current b per layer
```

```
            lookahead_params = {p: saved_params[p] + self.nag_mu *
self.vel[p] for p in self.learnable_params}   ## w_look = w + mu*v
            lookahead_bias = {
                layer: [saved_bias[layer][k] + self.nag_mu *
self.bias_vel[layer][k]
                        for k in range(self.layers_config[layer])]
                for layer in range(1, self.num_layers)
            }   ## b_look = b + mu*bias_vel per layer
            self.vals_for_learnable_params = lookahead_params   ## switch
to lookahead for forward
            self.bias = lookahead_bias
            ## -------------- MOD --------------
            self.forward_prop_multi_neuron_model(data_tuples)
## FORW PROP works by side-effect
            predicted_labels_for_batch =
self.forw_prop_vals_at_layers[self.num_layers-1]           ## Predictions
from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch
for item in sublist]       ## Get numeric vals for predictions
            ## -------------- MOD --------------
            self.vals_for_learnable_params = saved_params   ## restore
real w before backprop update
            self.bias = saved_bias   ## restore real b
            ## -------------- MOD --------------
```

In backprop_and_update_params_one_neuron_model
Here we temporarily replace the real parameters with the lookahead parameters, so the gradients are computed at the lookahead point, not at the original weights.

```
## -------------- MOD --------------
        real_params = self.vals_for_learnable_params   ## hold ref to real
w (will update in place)
        real_bias = self.bias   ## hold ref to real b
        self.vals_for_learnable_params = lookahead_params   ## use
lookahead so backprop computes grad at lookahead
        self.bias = lookahead_bias   ## same for bias
```

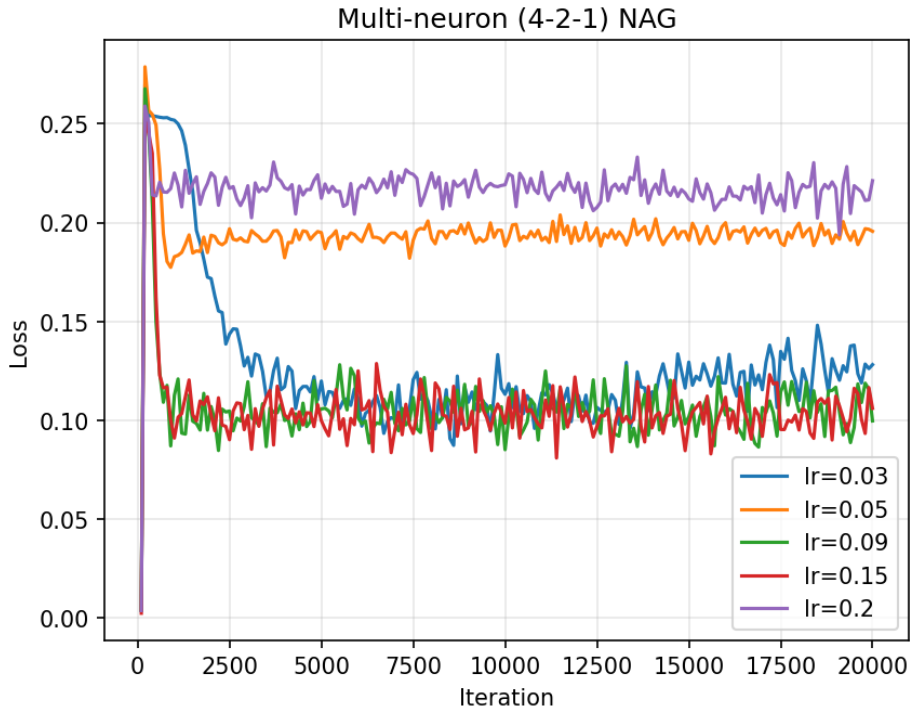
```
        ## -------------- MOD --------------
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param]
/  float(self.batch_size)   ## g = avg grad
```

```
            self.vel[param] = self.nag_mu * self.vel[param] +
self.learning_rate * partial_of_loss_wrt_param   ## v <- mu*v + lr*g
            real_params[param] += self.vel[param]   ## w <- w + v

        ##  Finally we update the biases at all the nodes that aggregate
data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                g_b = bias_changes[layer_index][k] /
float(self.batch_size)   ## g_b = avg bias grad
                self.bias_vel[layer_index][k] = self.nag_mu *
self.bias_vel[layer_index][k] + self.learning_rate * g_b   ## v_b <-
mu*v_b + lr*g_b
                real_bias[layer_index][k] += self.bias_vel[layer_index][k]
## b <- b + v_b
        self.vals_for_learnable_params = real_params   ## point self back
to updated real w
        self.bias = real_bias   ## point self back to updated real b
```

testing code

```
common = dict(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
    nag_mu=0.9,
)
learning_rates = [3e-2, 5e-2, 9e-2, 1.5e-1, 2e-1]
```

This also looks reasonable: for the multi-neuron case, NAG is more sensitive to the learning rate, and only a moderate range converges well, while larger learning rates become unstable and keep the loss high.



3. Performance Analysis

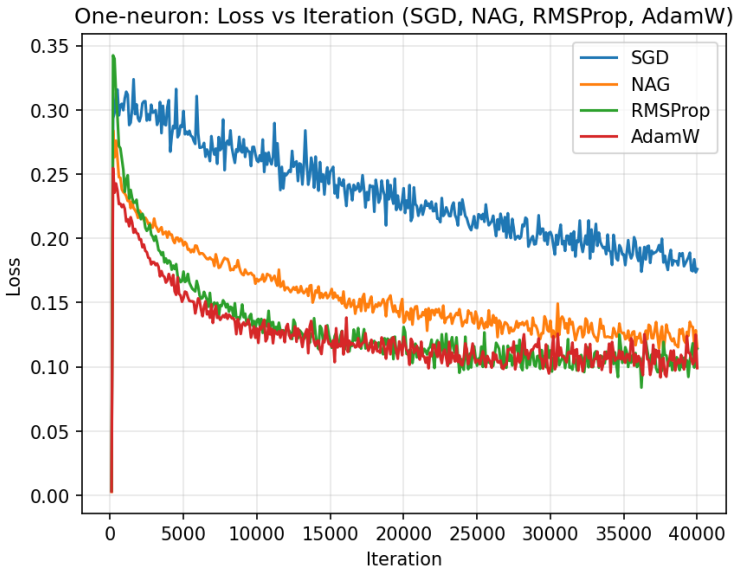
3.1 one neuron

With learning rate is 1e-3
The testing code is:

```
common = dict(
    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,
    debug=False,
)

# (label, ModelClass, extra kwargs)
# Weight/bias gradient now use correct descent direction for L=(label-
y_pred)^2
```

```
configs = [
    ("SGD", ComputationalGraphPrimer, {}),
    ("NAG", OneNeuronNAG, {"nag_mu": 0.9, "learning_rate": 1e-3,
"vel_clip": 0.5}),
    ("RMSProp", OneNeuronRMSProp, {"rms_rho": 0.9, "learning_rate": 1e-
3}),
    ("AdamW", OneNeuronAdamW, {"beta1": 0.9, "beta2": 0.999, "eps": 1e-3,
"weight_decay": 1e-2, "learning_rate": 1e-3}),
]
```

The results are:



One-neuron: Performance Analysis
Config: iterations=40000, batch_size=8, display_every=100

Optimizer | Training time (s) | Final loss | Min loss | Loss std (all) | Loss std (2nd half)
----------+-------------------+------------+----------+----------------+--------------------
SGD | 4.34 | 0.17597 | 0.00379 | 0.03897 | 0.01512
NAG | 5.97 | 0.11474 | 0.00273 | 0.03220 | 0.00773
RMSProp | 4.34 | 0.11405 | 0.00348 | 0.03915 | 0.00743
AdamW | 4.51 | 0.09897 | 0.00288 | 0.02984 | 0.00708

Convergence speed :
From the curves and numbers, SGD is the slowest. NAG goes down faster than SGD at the

beginning. RMSProp also drops fast in early iterations. AdamW is usually the fastest to reach a low loss region.

Final loss values:
SGD ends with the highest final loss. NAG and RMSProp are similar and much lower than SGD. AdamW gives the lowest final loss among all methods, so it performs best at the end.

Stability during training (smoothness):
SGD has larger fluctuation and more noisy loss curve. NAG is smoother than SGD. RMSProp is more stable in later stage because the adaptive step size helps control oscillation. AdamW is the most stable overall, with smaller loss std in the second half.

Why NAG might converge faster than vanilla momentum:
NAG looks ahead before taking the step, so it can "see" where the parameters are going. This helps to correct the direction early and reduces overshoot. So it often moves to the good region faster than normal momentum.

How RMSProp's adaptive learning rates help with convergence:
RMSProp scales the step size by the recent gradient magnitude. If a parameter has large gradients, the step becomes smaller. If gradient is small, the step becomes larger. This makes training more stable and helps the model converge faster, especially when different parameters have different scales.

Why AdamW often achieves the best final loss values:
AdamW combines momentum (first moment) and adaptive scaling (second moment), so it can move fast and stable at the same time. Also, weight decay is decoupled from the gradient update, so it regularizes the weights better. This usually gives better final loss than SGD, NAG, and RMSProp.

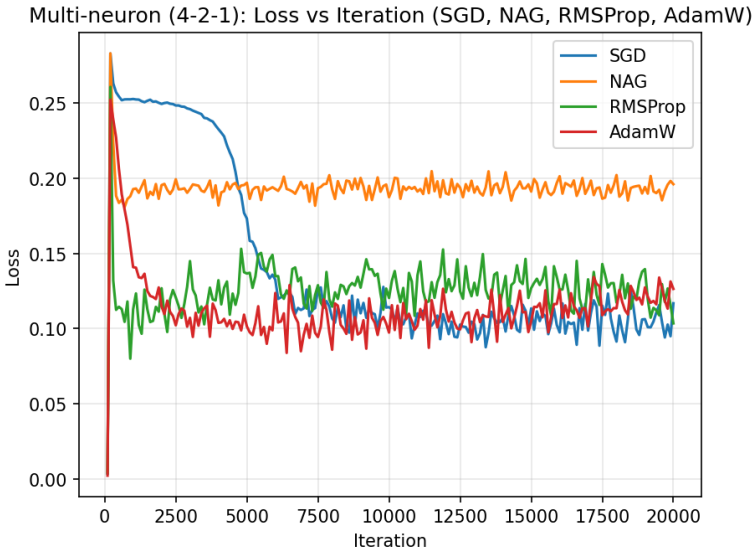
3.2 Multi neuron

Different learning rates are used for different optimizers, because they behave very differently. NAG is more sensitive to the learning rate and can get stuck if the rate is not well tuned. RMSProp can handle a larger learning rate better due to its adaptive step size, so it still converges well in the multi-neuron case.
The testing code:

```
common = dict(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
```

```
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    learning_rate=9e-2,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
)

configs = [
    ("SGD", ComputationalGraphPrimer, {}),
    ("NAG", MultiNeuronNAG, {"nag_mu": 0.9, "learning_rate": 9e-2}),
    ("RMSProp", MultiNeuronRMSProp, {"rms_rho": 0.9, "learning_rate":
1.5e-1}),
    ("AdamW", MultiNeuronAdamW, {"beta1": 0.9, "beta2": 0.999, "eps": 1e-
8, "weight_decay": 1e-4, "learning_rate": 1e-2}),
]
```

The results:



Optimizer	Training time (s)	Final loss	Min loss	Loss std (all)	Loss std (2nd half)
SGD	6.33	0.11686	0.00369	0.05879	0.00769
NAG	6.49	0.19606	0.00472	0.01555	0.00442
RMSProp	6.65	0.10353	0.00389	0.01743	0.01046
AdamW	6.63	0.12645	0.00228	0.02337	0.00954

Convergence speed:
From the curves and results, SGD goes down slowly at the beginning and needs more iterations to reach low loss. NAG drops a bit at first but then gets stuck early. RMSProp goes down faster than SGD in early stage. AdamW drops very fast at the beginning, so it has the fastest convergence speed.

Final loss values:
RMSProp gets the lowest final loss (0.10353), which is the best here. SGD is slightly worse (0.11686). AdamW final loss is a bit higher than SGD in this run. NAG is the worst, with final loss around 0.196, so it does not learn well in this setting.

Stability during training (smoothness):
NAG is the most stable since the loss std is small, but this is because it is stuck and not moving much. RMSProp and AdamW are more noisy than NAG, but still stable in the second half. SGD has larger fluctuation in the whole training, but in the second half it becomes more stable.

Why NAG might converge faster than vanilla momentum:
NAG looks ahead before updating the parameters, so it can correct the direction earlier. This usually helps it avoid overshooting and move faster to the good region than normal momentum. But if learning rate or momentum is not tuned well, it can also get stuck like here.

How RMSProp's adaptive learning rates help with convergence:
RMSProp changes the step size for each parameter based on recent gradients. If gradient is large, step becomes smaller, so it does not jump too much. If gradient is small, step becomes larger. This makes training more stable and helps the loss go down faster, especially in multi-neuron case.

Why AdamW often achieves the best final loss values:
AdamW uses momentum + adaptive scaling together, so it can move fast and stable at the same time. Also, weight decay is separated from gradient update, which helps regularize the weights better. In many cases this gives better final loss. In this run, AdamW is not the lowest final loss, but it still reaches a low loss fast and is quite stable.

4. Hyperparameter Sensitivity Analysis

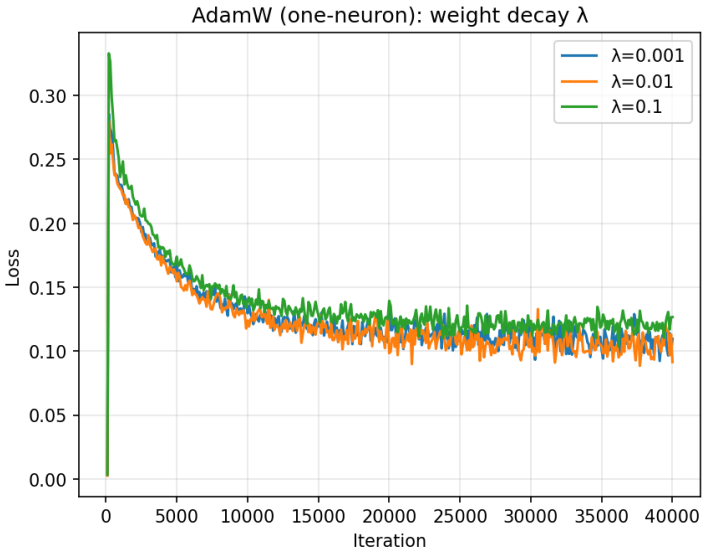
4.3 AdamW one neuron

Testing case

```
common = dict(
    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,
    debug=False,
    beta1=0.9,
    beta2=0.999,
    eps=1e-8,
)
weight_decay_values = [0.001, 0.01, 0.1]
```

results:



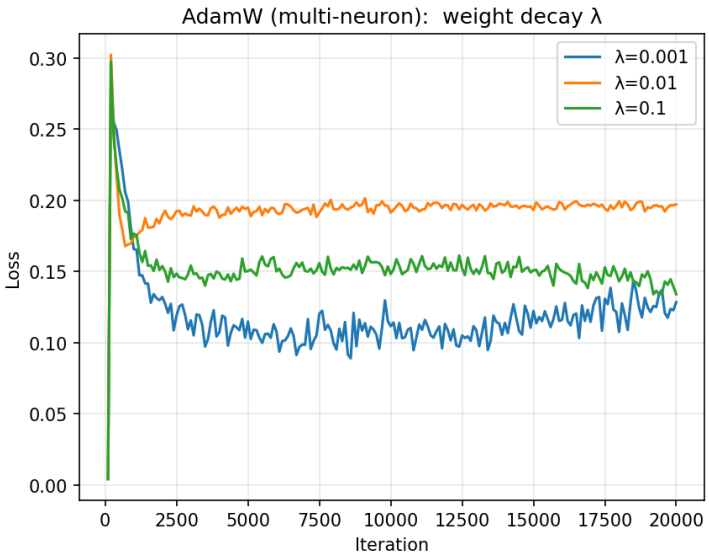
λ	Training time (s)	Final loss	Min loss	Loss std (all)	Loss std (2nd half)
λ=0.001	4.50	0.10963	0.00367	0.03237	0.00747
λ=0.01	4.44	0.09149	0.00270	0.03245	0.00753
λ=0.1	4.44	0.12666	0.00339	0.03417	0.00572

AdamW is sensitive to weight decay. Too small λ does not help much, too large λ hurts performance. A middle value (here λ = 0.01) works best in this experiment.

4.4 AdamW multi neuron

```
common = dict(
    num_layers=3,
```

```
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    learning_rate=1e-2,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
)

# λ = [0.001, 0.01, 0.1]
weight_decay_values = [0.001, 0.01, 0.1]
```



λ	Training time (s)	Final loss	Min loss	Loss std (all)	Loss std (2nd half)
λ=0.001	6.56	0.12847	0.00406	0.02751	0.01049
λ=0.01	6.58	0.19720	0.00445	0.01687	0.00185
λ=0.1	6.59	0.13401	0.00423	0.01916	0.00614

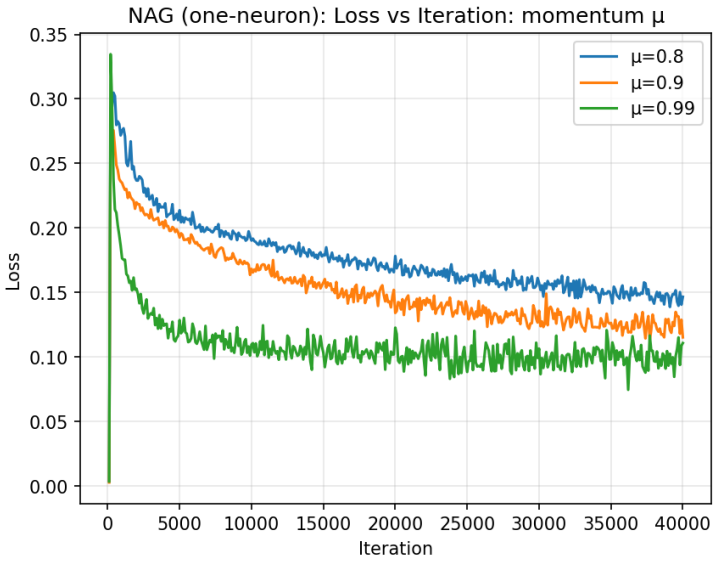
λ = 0.001: Final loss is the lowest. Regularization is small, so the model can fit data better. Training is a bit noisy but converges to a good value.
λ = 0.01: Final loss is much higher and curve is very flat. Weight decay is too strong here, so the model is over-regularized and cannot fit the data well (underfitting).

λ = 0.1: Also worse than λ=0.001. Too much weight decay, so parameters are pushed too much toward zero.

For multi-neuron case, AdamW is quite sensitive to λ. Small weight decay (λ=0.001) works best here. Bigger λ makes the model underfit and gives higher final loss.

4.3 NAG one neuron

```
common = dict(
    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,
    debug=False,
)

# μ = [0.8, 0.9, 0.99]
mu_values = [0.8, 0.9, 0.99]
captured_loss = {}
training_time = {}
```



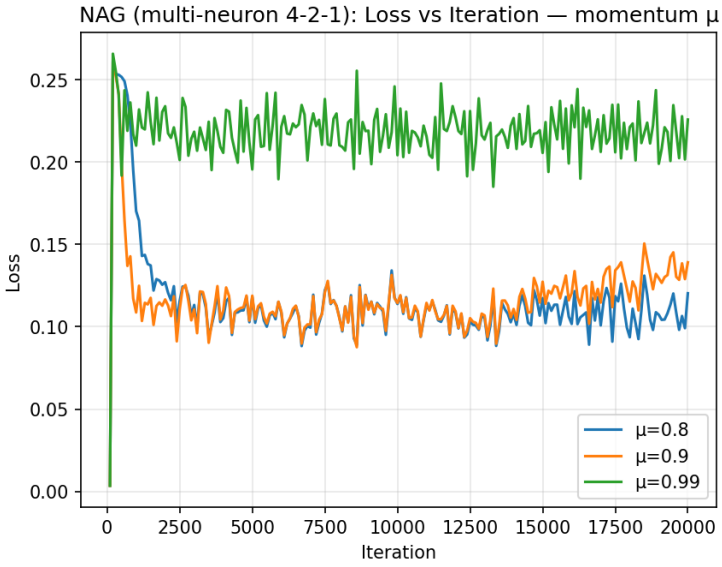
μ	Training time (s)	Final loss	Min loss	Loss std (all)	Loss std (2nd half)
μ=0.8	4.37	0.14652	0.00373	0.03102	0.00778
μ=0.9	4.35	0.11516	0.00268	0.03209	0.00769
μ=0.99	4.40	0.11069	0.00343	0.02472	0.00830

μ = 0.8: Converges slower and final loss is higher. Momentum is too small, so it does not help much.
μ = 0.9: Better than 0.8. Loss goes down faster and final loss is lower.
μ = 0.99: Best final loss and fastest drop at the beginning. High momentum helps move faster in the right direction, but curve is a bit more noisy.

NAG is sensitive to μ. Larger μ (like 0.99) gives faster convergence and lower final loss here, but too large μ can also bring more noise or risk of overshoot.

4.4 NAG multi neuron

```
common = dict(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    learning_rate=9e-2,
    training_iterations=20000,
    batch_size=8,
    display_loss_how_often=100,
    debug=False,
)

# μ = [0.8, 0.9, 0.99]
mu_values = [0.8, 0.9, 0.99]
```



μ	Training time (s)	Final loss	Min loss	Loss std (all)	Loss std (2nd half)
μ=0.8	6.51	0.12029	0.00351	0.02948	0.00858
μ=0.9	6.65	0.13907	0.00351	0.02300	0.01287
μ=0.99	6.74	0.22575	0.00351	0.02037	0.01272

This result is reasonable too, and it shows NAG behaves different in multi-neuron case:
μ = 0.8: Best final loss here. Converges fast and stays relatively low.
μ = 0.9: A bit worse final loss. Still converges, but more noisy later.
μ = 0.99: Bad here. Loss stays high and very noisy. Momentum is too large, so it overshoots and cannot settle well.

For multi-neuron model, very large μ (0.99) is too aggressive and hurts convergence.
Medium μ (0.8–0.9) works better. NAG is sensitive to μ, and the best μ depends on model size and landscape.