
DTU

Reinforcement Learning Control of Raman Amplifiers
06.11.2025

Main Loop

The main loop handles creating the target spectrum, based on a selected power, wavelength pair, makes the control loop containing the Raman system and the controller, setting the target output spectrum, and finally performing a set number of steps.

```
################################################################################
        Main loop
################################################################################

target_spectrum = raman_system.apply_input(input_spectrum, power, wavelength)

control_loop = make_control_loop(raman_system, controller)

control_loop.set_target(target_spectrum)

for i in step_number:
    control_loop.step()
```

Control Loop Step

The Control Loop step function handles high-level parts of the simulation.

It gets the current output produced by the system, calculates the control based on the current I/Os, updates the controller and applies the control for the next step.

```
################################################################################
    Control loop step
################################################################################

current_output = raman_system.get_output()
current_control = controller.get_control(current_input, current_output, target_output)
controller.update(current_input, current_output, target_output)
controller.update_controller(error)
raman_system.apply_control(current_control)
```

Bernoulli Get Control

The Get Control function produces the control action which the system will take in the next time step.

Bernoulli Controller implements this function by creating a Bernoulli distribution based on the logits of the controller.

Next it makes a number of samples and averages them out (for more stable control)

Finally, it creates a control action based on that sample by centering it around 0 and scaling it by the step size.

```
#############################################################################
        Bernoulli get control
#############################################################################

probs = sigmoid(logits)

distribution = Bernoulli(probs)

while length(samples) < samples_number
    sample = get_sample(distribution)
    samples.add(sample)
sample = mean(samples)

action = convert_to_action(sample)

return action
```

Bernoulli Update

The Bernoulli update function focuses on updating the logit values of the controller.

It first calculates the reward based on the current and the target output

Then in updates the baseline by filtering it with the gamma parameter.

Next it calculates the advantage.

Then it calculates the eligibility – how likely the sample is based on the probabilities.

Finally, it calculates and applies the update.

```
##########################################################################################
        Bernoulli update
##########################################################################################

reward = calculate_reward(current_output, target_output)

baseline = gamma * baseline + (1 - gamma) * reward

advantage = beta * (reward - baseline)

eligibility = sample - probs

update = learning_rate * advantage * eligibility - weight_decay * logits

logits += update
```

Reward

The reward is calculated based on two components: the shape loss and the integral loss of the spectrum.

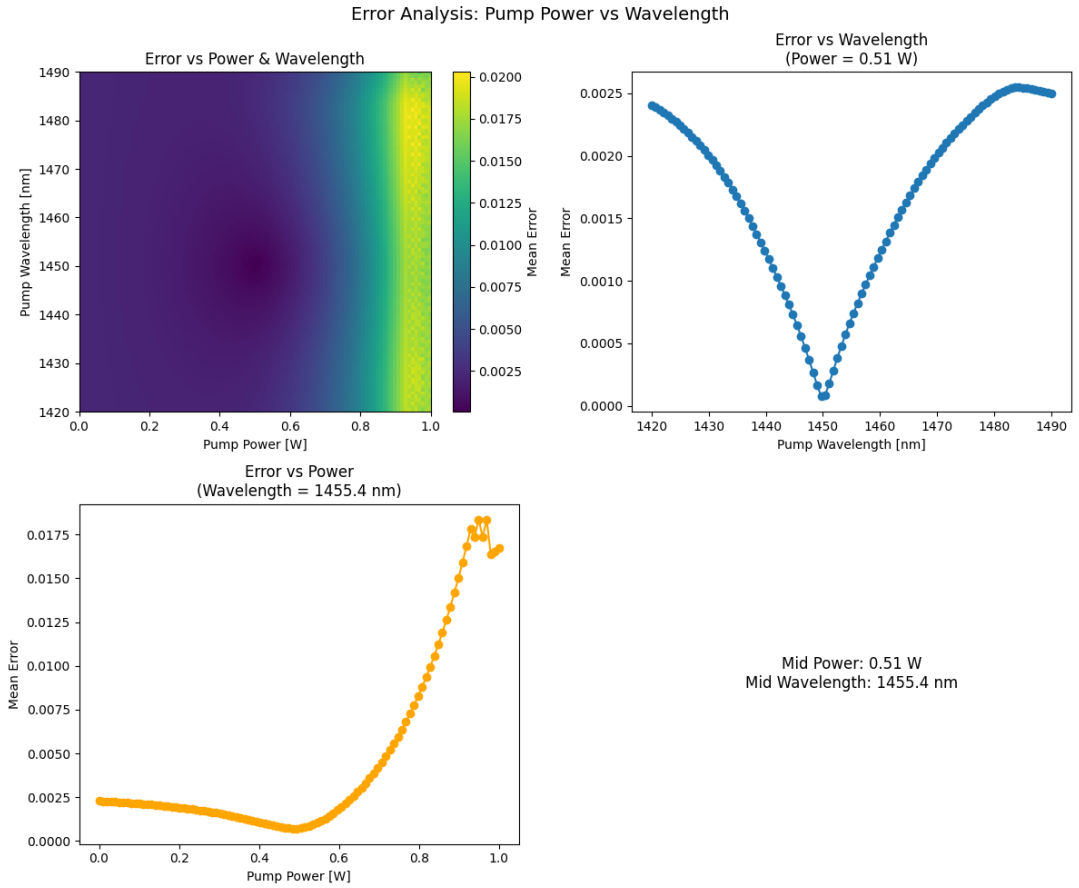
The shape loss represents the difference in output spectrum and target spectrum shapes, which is calculated by normalizing the two, and subtracting them.

The integral loss only cares about the integral value of the two spectrums and doesn't care about their shapes.

This reward function setup allows for easier scaling of parts of the function, and potentially for splitting it completely.

```
##########################################################################################
    Reward function
##########################################################################################

shape_loss = |normalized_output_spectrum - normalized_target_spectrum|

integral_loss = integral(currrent_output_spectrum) - integral(target_spectrum)

loss = shape_loss + integral_loss

reward = - loss
```

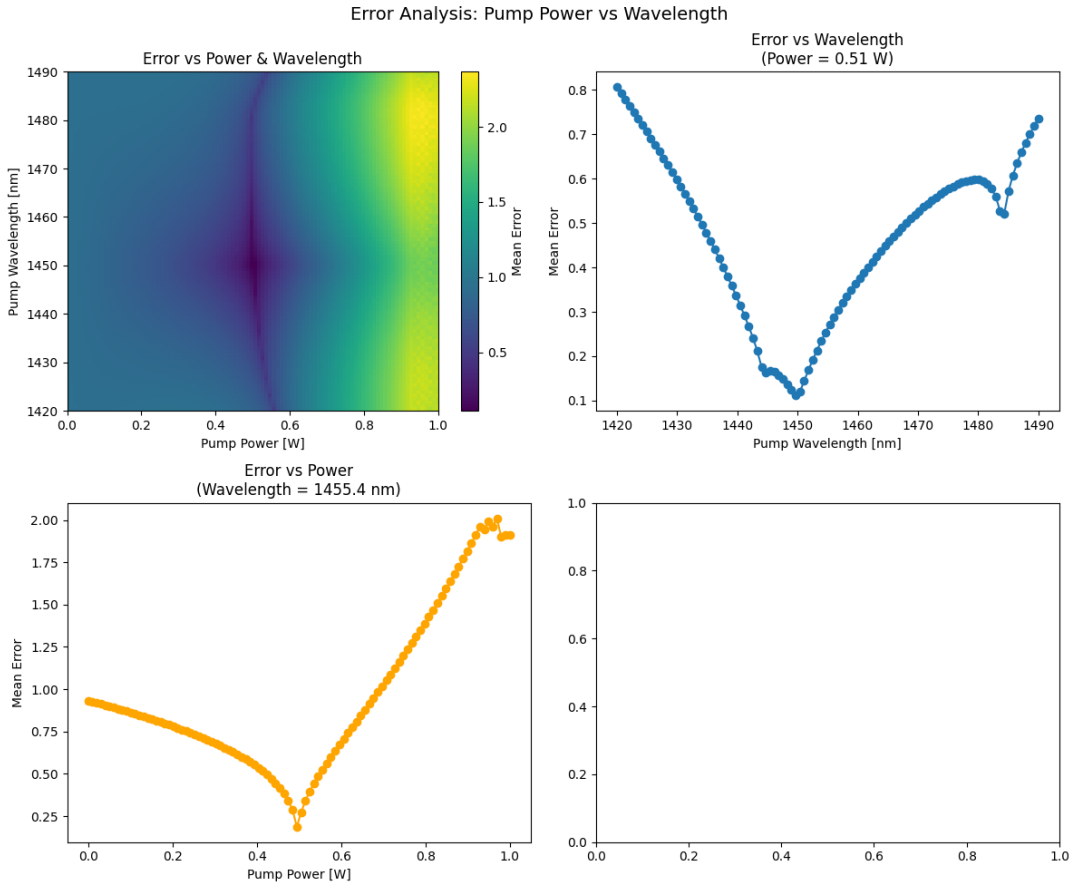
MSE Loss function



Mid Power: 0.51 W
Mid Wavelength: 1455.4 nm

Looking at the Mean Squared Error function as a loss for calculating the reward, we can see that the gradient around the target is quite weak, which can be seen when simulating the system.

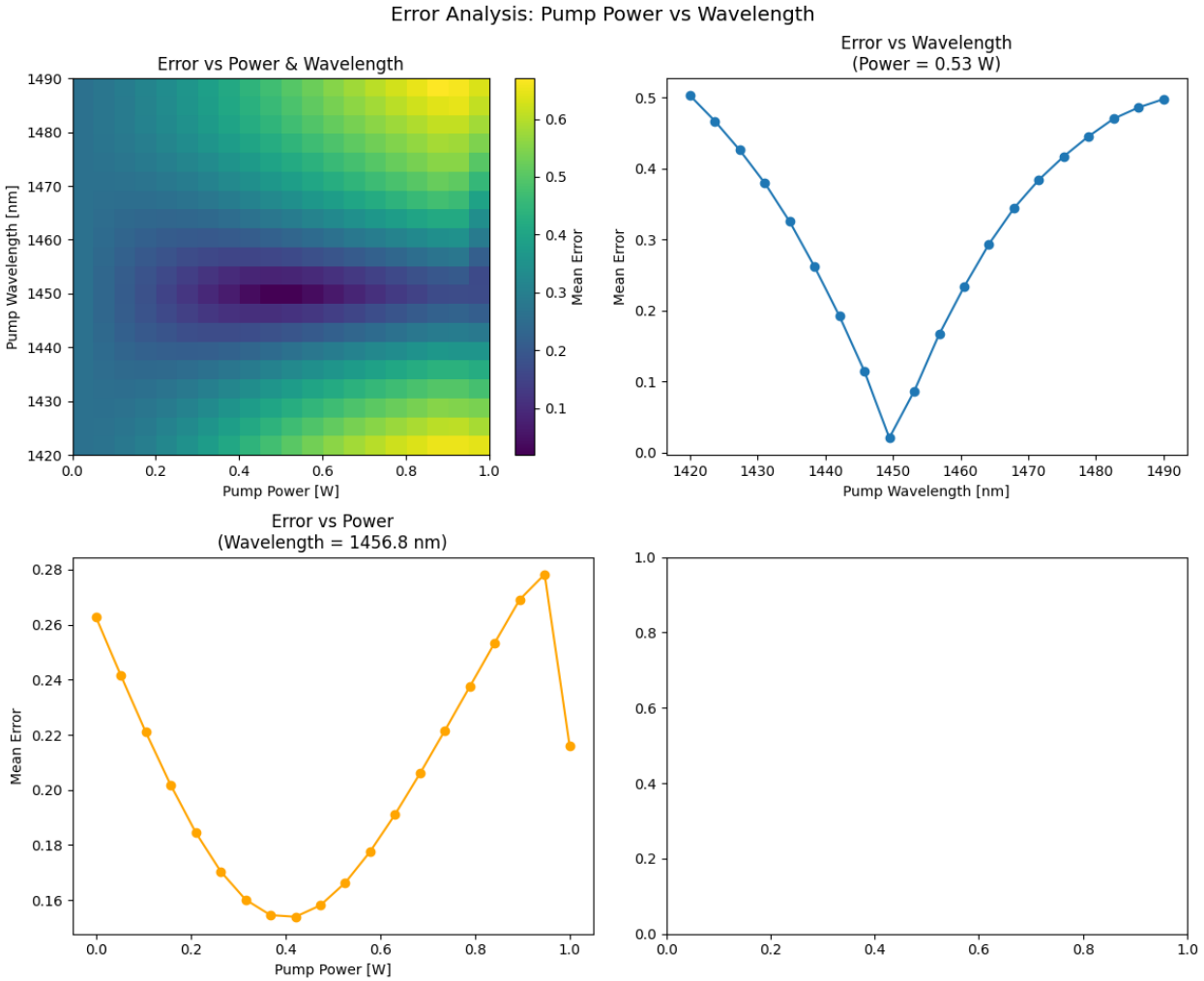
Custom Loss function



Comparing the custom two-component loss function, we can see that there is a steeper gradient around the minimum, which makes for more stable optimization.

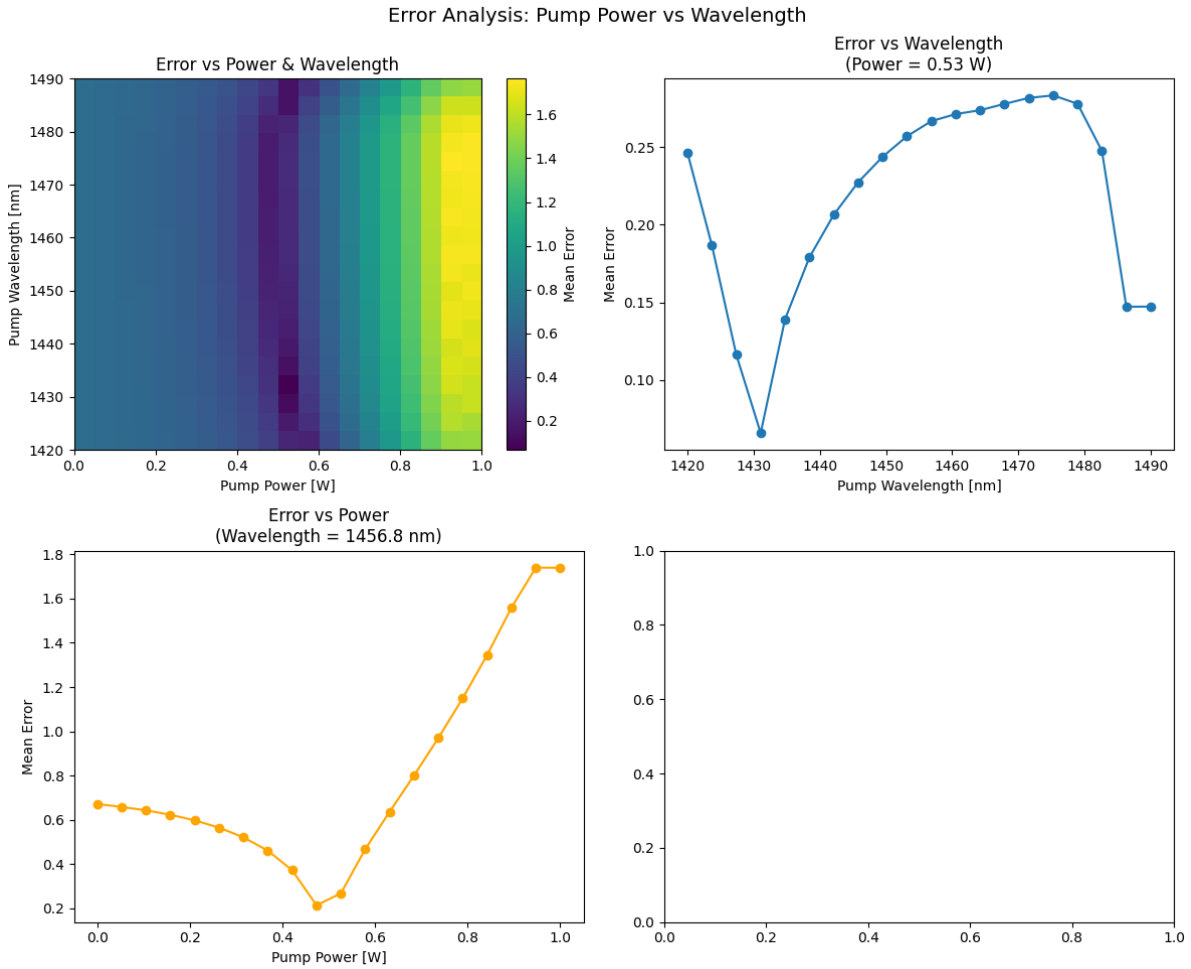
Despite this advantage, this loss function suffers from poor gradient shape around the edges of the interval, where it can get easily stuck.

Custom Loss function – Shape Component



When looking at the shape component of the loss function, we can see a horizontal valley, which corresponds to the power axis, meaning that this component doesn't care much about the power value. This valley doesn't extend across the whole range due to the impact power has on the spectrum shape which is not negligable.

Custom Loss function – Integral Component



When looking at the integral component of the loss function, we can see a vertical valley, which corresponds to the wavelength axis, meaning that this component doesn't care much about the wavelength value. This valley doesn't extend across the whole range because setting the wavelength to the edge of its interval can send parts of the spectrum outside of the range we are calculating in, resulting in more power needed for the same integral.