```python
import autograd.numpy as np
from autograd import grad
from autograd import elementwise_grad as egrad
from autograd.scipy import stats
import random


def hamiltonian_monte_carlo(log_density, theta_0, epsilon, leapfrog_iter, overall_iter, n_var=1,
                            momentum_std=1.0, temperature=1.0, debug=False, log_interval=0):
    """
    An implemention of Hamiltonian Monte Carlo, mostly following the pseudocode in the NUTS
    paper (Hoffman & Gelman, 2011). This function starts from a provided guess of $\theta$ and
    generates one chain of samples, without discarding any that might be treated as warmup.

    :param log_density: The log-density to sample from.
    :param theta_0: An initial guess for the parameters
    :param epsilon: The step size to take in each leapfrog iteration
    :param leapfrog_iter: The number of leapfrog iterations to take (often denoted $L$)
    :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
    :param n_var: The number of variables, the dimension of $\theta$, which is asusmed to be 1-D
    :param momentum_std: A parameter for sampling the momentum variable. By default, if a number
        (or anything that is not callable) is provided, it is treated as $\sigma$ of a zero-mean
        normal distribution. If a callable is provided, it is treated as a sampler for momentum.
    :param temperature: The temeprature parameter, which in this case only impacts the acceptance
        probabilities. Defaults to one.
    :param debug: Whether or not to include debug prints
    :param log_interval: How often to log the acceptance counts.
    :return: A list of samples, of length `overall_iter`, each a tuple of (theta, r, accpeted)
    """
    if log_interval != 0:
        print(f'HMC: sampling {overall_iter} iterations with {leapfrog_iter} leapfrog iterations with epsilon = {epsilon}')

    samples = []
    accepted_count = 0
    theta = np.asarray(theta_0, dtype=np.float64)
    log_density_grad = grad(log_density)
    hamiltonian = generate_hamiltonian(log_density)

    # handle the default case, of specifying a zero-mean normal distribution std
    if not callable(momentum_std):
        def momentum_sampler():
            return np.random.normal(0, momentum_std, size=n_var)

    else:
        momentum_sampler = momentum_std

    if theta.size != n_var:
        raise ValueError('Theta_0 should have the same number of variables as n_var specifies')

    for i in range(overall_iter):
        r = momentum_sampler()
        if n_var == 1:
            r = r[0]

        new_theta, new_r = leapfrog(theta, r, epsilon, leapfrog_iter, log_density_grad)

        # subtracting inside the exponent for numerical stability
        acceptance_prob = np.exp((hamiltonian(new_theta, new_r) - hamiltonian(theta, r)) / temperature)

        if debug:
            print(theta, r, new_theta, new_r, acceptance_prob)

        accepted = np.random.uniform() < acceptance_prob

        if accepted:
            accepted_count += 1
            theta = new_theta
            r = new_r

        samples.append((theta, r, accepted))

        if log_interval > 0 and i % log_interval == 0 and i > 0:
            print(f'Sampled {i} iterations of which {accepted_count - 1} were accepted')

    return np.asarray(samples)


def leapfrog(theta, r, epsilon, leapfrog_iter, log_density_grad):
    """
    A simple implementation of the leapfrog method symplectic integrator.
    :param theta: The starting point in parameter-space.
    :param r: The initial momentum.
    :param epsilon: The step size.
    :param leapfrog_iter: The number of iterations ot take ($L$).
    :param log_density_grad: The gradient of the log-density, used as the rate
        of change of the momentum variables (by the partial derivative dH/dq)
    :return: The final values for theta and r after taking $L$ steps
    """
    grad_theta = log_density_grad(theta)
    theta = np.copy(theta)
    r = np.copy(r)
    for l in range(leapfrog_iter):
        r += epsilon / 2 * grad_theta
        theta += epsilon * r
        grad_theta = log_density_grad(theta)
        r += epsilon / 2 * grad_theta

    return theta, r


def generate_hamiltonian(log_density):
    """
    A utility function to generate a Hamiltonian (as a function of $theta$ and $r$)
    form the log density.
    :param log_density: The log density to generate a Hamiltonian for.
    :return: The Hamiltonian function
    """
    def h(theta, r):
        return log_density(theta) - 0.5 * np.dot(r, r)

    return h


def no_u_turn_sampler(log_density, theta_0, epsilon, overall_iter, n_var=1,
                      momentum_std=1.0, debug=False, log_interval=0):
    """
    A straightforward and unoptimized implementation of the No U-Turn Sampler as introduced
    in Hoffman & Gelman (2011).
    :param log_density: The log-density to sample from.
    :param theta_0: An initial guess for the parameters
    :param epsilon: The step size to take in each leapfrog iteration
    :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
    :param n_var: The number of variables, the dimension of $\theta$, which is asusmed to be 1-D
    :param momentum_std: A parameter for sampling the momentum variable. By default, if a number
        (or anything that is not callable) is provided, it is treated as $\sigma$ of a zero-mean
        normal distribution. If a callable is provided, it is treated as a sampler for momentum.
    :param debug: Whether or not to include debug prints
    :param log_interval: How often to log the acceptance counts.
    :return: A list of samples, of length `overall_iter`, each a tuple of (theta, r, accpeted)
    """
    if log_interval != 0:
        print(f'NUTS: sampling {overall_iter} iterations with epsilon = {epsilon}')

    samples = []
    theta = np.asarray(theta_0, dtype=np.float64)
    log_density_grad = grad(log_density)
    hamiltonian = generate_hamiltonian(log_density)

    # handle the default case, of specifying a zero-mean normal distribution std
    if not callable(momentum_std):
        def momentum_sampler():
            return np.random.normal(0, momentum_std, size=n_var)

    else:
        momentum_sampler = momentum_std

    if theta.size != n_var:
        raise ValueError('Theta_0 should have the same number of variables as n_var specifies')

    for i in range(overall_iter):
        r = momentum_sampler()
        if n_var == 1:
            r = r[0]

        u_slice = np.random.uniform(0, np.exp(hamiltonian(theta, r)))
        theta_left = theta
        theta_right = theta
        r_left = r
        r_right = r
        j_iter = 0
        results = set()
        s_valid = True

        while s_valid:
            v_direction = (np.random.uniform() < 0.5) and 1 or -1
            if v_direction == -1:
                theta_left, r_left, _, _, new_results, new_s_valid = build_tree(
                    theta_left, r_left, u_slice, v_direction, j_iter,
                    epsilon, hamiltonian, log_density_grad)

            else:
                _, _, theta_right, r_right, new_results, new_s_valid = build_tree(
                    theta_right, r_right, u_slice, v_direction, j_iter,
                    epsilon, hamiltonian, log_density_grad)

            if new_s_valid:
                # ugly workaround to the fact that numpy arrays are not valid keys (immutable)
                for theta, r in new_results:
                    results.add((tuple(theta), tuple(r)))

            s_valid = new_s_valid and (np.dot((theta_right - theta_left), r_left) >= 0) and \
                      (np.dot((theta_right - theta_left), r_right) >= 0)

        if len(results) > 0:
            theta, r = random.choice(tuple(results))
            theta = np.array(theta)
            r = np.array(r)
            accepted = True
        else:
            print(f'Encountered an empty result set on iteration {i}')
            accepted = False

        if debug:
            print(theta, r)

        samples.append((theta, r, accepted))

        if log_interval > 0 and i % log_interval == 0 and i > 0:
            print(f'Sampled {i} iterations')

    return np.asarray(samples)


DEFAULT_DELTA_MAX = 1000


def build_tree(theta, r, u_slice, v_direction, j_iter, epsilon, hamiltonian, log_density_grad,
               delta_max=DEFAULT_DELTA_MAX):
    """
    A straightforward and unoptimized implementation of the BuildTree routine of the
    No U-Turn Sampler as introduced in Hoffman & Gelman (2011).
    :param theta: The current values for the parameters
    :param r: The current momentum values
    :param u_slice: The value of the introduced slice variable
    :param v_direction: The direction variable
    :param j_iter: The number of iterations remaining to take
    :param epsilon: The leapfrog step size to employ
    :param hamiltonian: The Hamiltonian for the system we're sampling from
    :param log_density_grad: The gradient of the log density, used for the leapfrog integrator
    :param delta_max: A maximal error term
    :return: Edge values of theta and r for both sides of the tree, the set of all valid reached
        points, and the indicator variable for whether or not we've reached a U-turn (or error)
    """
    # base case - take the first step in the direction v
    if j_iter == 0:
        new_theta, new_r = leapfrog(theta, r, epsilon * v_direction, 1, log_density_grad)

        results = list()
        h = hamiltonian(new_theta, new_r)
        if u_slice < np.exp(h):
            results.append((new_theta, new_r))

        s_valid = u_slice < np.exp(delta_max + h)
        return new_theta, new_r, new_theta, new_r, results, s_valid

    # recursive case - build both other side trees.
    # TODO: consider reframing recursion as for loop?
    theta_left, r_left, theta_right, r_right, first_results, first_s_valid = build_tree(
        theta, r, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max)

    # take second step in appropriate direction
    if v_direction == -1:
        theta_left, r_left, _, _, second_results, second_s_valid = build_tree(
            theta_left, r_left, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max
        )

    else:
        _, _, theta_right, r_right,second_results, second_s_valid = build_tree(
            theta_right, r_right, u_slice, v_direction, j_iter - 1, epsilon, hamiltonian, log_density_grad, delta_max
        )

    s_valid = first_s_valid and second_s_valid and \
              (np.dot((theta_right - theta_left), r_left) >= 0) and \
              (np.dot((theta_right - theta_left), r_right) >= 0)

    results = first_results + second_results
    return theta_left, r_left, theta_right, r_right, results, s_valid


def metropolis_hastings(log_density, theta_0, overall_iter, n_var=1,
                        proposal_std=1.0, temperature=1.0, debug=False, log_interval=0):
    """
    A fairly na\"{i}ve implmentation of a Metropolis-Hastings sampler, provided mostly for
    comparison purposes. The support for temperature annealing schedules is the only non-trivial piece.
    :param log_density: The log-density to sample from.
    :param theta_0: An initial guess for the parameters
    :param overall_iter: The number of overall HMC iterations to take (samples in the chain)
    :param n_var: The number of variables, the dimension of $\theta$, which is asusmed to be 1-D
    :param proposal_std: The standard deviation to use when generating Metropolis-Hastings proposals.
    :param temperature: The temeprature parameter, which in this case only impacts the acceptance
        probabilities. Defaults to one. If a callable is provided, it is treated as a temperature
        function that receives a single parameter, the iteration number. Used to implement a temperature
        annealing schedule.
    :param debug: Whether or not to include debug prints
    :param log_interval: How often to log the acceptance counts.
    :return: A list of samples, of length `overall_iter`, each a tuple of (theta, r, accpeted)
    """
    if log_interval != 0:
        print(f'Metropolis-Hastings: sampling {overall_iter} iterations')

    samples = []
    accepted_count = 0
    theta = np.asarray(theta_0, dtype=np.float64)

    # handling both a temperature value and a temperature function
    if not callable(temperature):
        def temp_func(i):
            return temperature
    else:
        temp_func = temperature

    if theta.size != n_var:
        raise ValueError('Theta_0 should have the same number of variables as n_var specifies')

    for i in range(overall_iter):
        new_theta = np.random.normal(theta, proposal_std, size=n_var)

        # subtracting inside the exponent for numerical stability
        acceptance_prob = np.exp((log_density(new_theta) - log_density(theta)) / temp_func(i))

        if debug:
            print(theta, new_theta, acceptance_prob)

        accepted = np.random.uniform() < acceptance_prob

        if accepted:
            accepted_count += 1
            theta = new_theta

        samples.append((theta, accepted))

        if log_interval > 0 and i % log_interval == 0 and i > 0:
            print(f'Sampled {i} iterations of which {accepted_count} were accepted')

    return np.asarray(samples)
```