Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Function Approximation and Approximate Dynamic Programming

In Chapter **??**, we covered Dynamic Programming algorithms where the MDP is specified in the form of a finite data structure and the Value Function is represented as a finite "table" of states and values. These Dynamic Programming algorithms swept through all states in each iteration to update the value function. But when the state space is large (as is the case in real-world applications), these Dynamic Programming algorithms won't work because:

1. A "tabular" representation of the MDP (or of the Value Function) won't fit within storage limits.
2. Sweeping through all states and their transition probabilities would be time-prohibitive (or simply impossible, in the case of infinite state spaces).

Hence, when the state space is very large, we need to resort to approximation of the Value Function. The Dynamic Programming algorithms would need to be suitably modified to their Approximate Dynamic Programming (abbreviated as ADP) versions. The good news is that it's not hard to modify each of the (tabular) Dynamic Programming algorithms such that instead of sweeping through all the states in each iteration, we simply sample an appropriate subset of the states, calculate the values for those states (with the same Bellman Operator calculations as for the case of tabular), and then create/update a function approximation (for the Value Function) using the sampled states' calculated values. Furthermore, if the set of transitions from a given state is large (or infinite), instead of using the explicit probabilities of those transitions, we can sample from the transitions probability distribution. The fundamental structure of the algorithms and the fundamental principles (Fixed-Point and Bellman Operators) would still be the same.

So, in this chapter, we do a quick review of function approximation, write some code for a couple of standard function approximation methods, and then utilize these function approximation methods to develop Approximate Dynamic Programming algorithms (in particular, Approximate Policy Evaluation, Approximate Value Iteration and Approximate Backward Induction). Since you are reading this book, it's highly likely that you are already familiar with the simple and standard function approximation methods such as linear function approximation and function approximation using neural networks supervised learning. So we shall go through the background on linear function approximation and neural networks supervised learning in a quick and terse manner, with the goal of developing some code for these methods that we can use not just for the ADP algorithms for this chapter, but also for RL algorithms later in the book. Note also that apart from approximation of State-Value Functions $\mathcal{N} \to \mathbb{R}$ and Action-Value Functions $\mathcal{N} \times \mathcal{A} \to \mathbb{R}$, these function approximation methods can also be used for approximation of Stochastic Policies $\mathcal{N} \times \mathcal{A} \to [0, 1]$ in Policy-based RL algorithms.

1.1 Function Approximation

In this section, we describe function approximation in a fairly generic setting (not specific to approximation of Value Functions or Policies). We denote the predictor variable as x, belonging to an arbitrary domain denoted $\mathcal{X}$ and the response variable as $y \in \mathbb{R}$. We treat x and y as unknown random variables and our goal is to estimate the probability distribution function f of the conditional random variable $y|x$ from data provided in the form of a sequence of (x, y) pairs. We shall consider parameterized functions f with the parameters denoted as w. The exact data type of w will depend on the specific form of function approximation. We denote the estimated probability of y conditional on x as $f(x; w)(y)$. Assume we are given data in the form of a sequence of n (x, y) pairs, as follows:

$$[(x_i, y_i)|1 \leq i \leq n]$$

The notion of estimating the conditional probability $\mathbb{P}[y|x]$ is formalized by solving for $w = w^*$ such that:

$$w^* = \arg\max_w \{\prod_{i=1}^{n} f(x_i; w)(y_i)\} = \arg\max_w \{\sum_{i=1}^{n} \log f(x_i; w)(y_i)\}$$

In other words, we shall be operating in the framework of Maximum Likelihood Estimation. We say that the data $[(x_i, y_i)|1 \leq i \leq n]$ specifies the *empirical probability distribution* D of $y|x$ and the function f (parameterized by w) specifies the *model probability distribution* M of $y|x$. With maximum likelihood estimation, we are essentially trying to reconcile the model probability distribution M with the empirical probability distribution D. Hence, maximum likelihood estimation is essentially minimization of a loss function defined as the cross-entropy $\mathcal{H}(D, M) = -\mathbb{E}_D[\log M]$ between the probability distributions D and M. The term *function approximation* refers to the fact that the model probability distribution M serves as an approximation to some true probability distribution that is only partially observed through the empirical probability distribution D.

Our framework will allow for incremental estimation wherein at each iteration t of the incremental estimation (for $t = 1, 2, \ldots$), data of the form

$$[(x_{t,i}, y_{t,i})|1 \leq i \leq n_t]$$

is used to update the parameters from w_{t-1} to w_t (parameters initialized at iteration $t = 0$ to w_0). This framework can be used to update the parameters incrementally with a gradient descent algorithm, either stochastic gradient descent (where a single (x, y) pair is used for each iteration's gradient calculation) or mini-batch gradient descent (where an appropriate subset of the available data is used for each iteration's gradient calculation) or simply re-using the entire data available for each iteration's gradient calculation (and consequent, parameters update). Moreover, the flexibility of our framework, allowing for incremental estimation, is particularly important for Reinforcement Learning algorithms wherein we update the parameters of the function approximation from the new data that is generated from each state transition as a result of interaction with either the real environment or a simulated environment.

Among other things, the estimate f (parameterized by w) gives us the model-expected value of y conditional on x, i.e.

$$\mathbb{E}_M[y|x] = \mathbb{E}_{f(x;w)}[y] = \int_{-\infty}^{+\infty} y \cdot f(x; w)(y) \cdot dy$$

We refer to $\mathbb{E}_M[y|x]$ as the function approximation's prediction for a given predictor variable x.

For the purposes of Approximate Dynamic Programming and Reinforcement Learning, the function approximation's prediction $\mathbb{E}[y|x]$ provides an estimate of the Value Function for any state (x takes the role of the *State*, and y takes the role of the *Return* following that State). In the case of function approximation for (stochastic) policies, x takes the role of the *State*, y takes the role of the *Action* for that policy, and $f(x; w)$ provides the probability distribution of actions for state x (corresponding to that policy). It's also worthwhile pointing out that the broader theory of function approximations covers the case of multi-dimensional y (where y is a real-valued vector, rather than scalar) - this allows us to solve classification problems as well as regression problems. However, for ease of exposition and for sufficient coverage of function approximation applications in this book, we only cover the case of scalar y.

Now let us write some code that captures this framework. We write an abstract base class `FunctionApprox` type-parameterized by `X` (to permit arbitrary data types $\mathcal{X}$), representing $f(x; w)$, with the following 3 key methods, each of which will work with inputs of generic `Iterable` type (`Iterable` is any data type that we can iterate over, such as `Sequence` types or `Iterator` type):

1. `@abstractmethod solve`: takes as input an `Iterable` of (x, y) pairs and solves for the optimal internal parameters w^* that minimizes the cross-entropy between the empirical probability distribution of the input data of (x, y) pairs and the model probability distribution $f(x; w)$. Some implementations of `solve` are iterative numerical methods and would require an additional input of `error_tolerance` that specifies the required precision of the best-fit parameters w^*. When an implementation of `solve` is an analytical solution not requiring an error tolerance, we specify the input `error_tolerance` as `None`. The output of `solve` is the `FunctionApprox` $f(x; w^*)$ (i.e., corresponding to the solved parameters w^*).
2. `update`: takes as input an `Iterable` of (x, y) pairs and updates the parameters w defining $f(x; w)$. The purpose of `update` is to perform an incremental (iterative) improvement to the parameters w, given the input data of (x, y) pairs in the current iteration. The output of `update` is the `FunctionApprox` corresponding to the updated parameters. Note that we should be able to `solve` based on an appropriate series of incremental `updates` (upto a specified `error_tolerance`).
3. `@abstractmethod evaluate`: takes as input an `Iterable` of x values and calculates $\mathbb{E}_M[y|x] = \mathbb{E}_{f(x;w)}[y]$ for each of the input x values, and outputs these expected values in the form of a `numpy.ndarray`.

As we've explained, an incremental update to the parameters w involves calculating a gradient and then using the gradient to adjust the parameters w. Hence the method `update` is supported by the following two abstract methods.

- `@abstractmethod objective_gradient`: computes the gradient of an objective function (call it $Obj(x, y)$) of the `FunctionApprox` with respect to the parameters w in the internal representation of the `FunctionApprox`. The gradient is output in the form of a `Gradient` type. The second argument `obj_deriv_out_fun` of the `objective_gradient` method represents the partial derivative of Obj with respect to an appropriate model-computed value (call it $Out(x)$), i.e., $\frac{\partial Obj(x,y)}{\partial Out(x)}$, when evaluated at a `Sequence` of x values and a `Sequence` of y values (to be obtained from the first argument `xy_vals_seq` of the `objective_gradient` method).

- `@abstractmethod update_with_gradient`: takes as input a `Gradient` and updates the internal parameters using the gradient values (eg: gradient descent update to the parameters), returning the updated `FunctionApprox`.

The `update` method is written with $\frac{\partial Obj(x_i,y_i)}{\partial Out(x_i)}$ defined as follows, for each training data point (x_i, y_i):

$$\frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)} = \mathbb{E}_M[y|x_i] - y_i$$

It turns out that for each concrete function approximation that we'd want to implement, if the Objective $Obj(x_i, y_i)$ is the cross-entropy loss function, we can identify a model-computed value $Out(x_i)$ (either the output of the model or an intermediate computation of the model) such that $\frac{\partial Obj(x_i,y_i)}{\partial Out(x_i)}$ is equal to the prediction error $\mathbb{E}_M[y|x_i] - y_i$ (for each training data point (x_i, y_i)) and we can come up with a numerical algorithm to compute $\nabla_w Out(x_i)$, so that by chain-rule, we have the required gradient:

$$\nabla_w Obj(x_i, y_i) = \frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)} \cdot \nabla_w Out(x_i) = (\mathbb{E}_M[y|x_i] - y_i) \cdot \nabla_w Out(x_i)$$

The `update` method implements this chain-rule calculation, by setting `obj_deriv_out_fun` to be the prediction error $\mathbb{E}_M[y|x_i] - y_i$, delegating the calculation of $\nabla_w Out(x_i)$ to the concrete implementation of the abstract method `objective_gradient`

Note that the `Gradient` class contains a single attribute of type `FunctionApprox` so that a `Gradient` object can represent the gradient values in the form of the internal parameters of the `FunctionApprox` attribute (since each gradient value is simply a partial derivative with respect to an internal parameter).

We use the `TypeVar F` to refer to a concrete class that would implement the abstract interface of `FunctionApprox`.

```
from abc import ABC, abstractmethod
import numpy as np

X = TypeVar('X')
F = TypeVar('F', bound='FunctionApprox')

class FunctionApprox(ABC, Generic[X]):

    @abstractmethod
    def objective_gradient(
        self: F,
        xy_vals_seq: Iterable[Tuple[X, float]],
        obj_deriv_out_fun: Callable[[Sequence[X], Sequence[float]], np.ndarray]
    ) -> Gradient[F]:
        pass

    @abstractmethod
    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        pass

    @abstractmethod
    def update_with_gradient(
        self: F,
        gradient: Gradient[F]
    ) -> F:
        pass

    def update(
        self: F,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> F:
```

```
        def deriv_func(x: Sequence[X], y: Sequence[float]) -> np.ndarray:
            return self.evaluate(x) - np.array(y)

        return self.update_with_gradient(
            self.objective_gradient(xy_vals_seq, deriv_func)
        )

    @abstractmethod
    def solve(
        self: F,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> F:
        pass

@dataclass(frozen=True)
class Gradient(Generic[F]):
    function_approx: F
```

When concrete classes implementing `FunctionApprox` write the `solve` method in terms of the `update` method, they will need to check if a newly updated `FunctionApprox` is "close enough" to the previous `FunctionApprox`. So each of them will need to implement their own version of "Are two `FunctionApprox` instances within a certain `error_tolerance` of each other?" Hence, we need the following abstract method `within`:

```
    @abstractmethod
    def within(self: F, other: F, tolerance: float) -> bool:
        pass
```

Any concrete class that implement this abstract class `FunctionApprox` will need to implement these five abstract methods of `FunctionApprox`, based on the specific assumptions that the concrete class makes for f.

Next, we write some methods useful for classes that inherit from `FunctionApprox`. Firstly, we write a method called `iterate_updates` that takes as input a stream (`Iterator`) of `Iterable` of (x, y) pairs, and performs a series of incremental updates to the parameters w (each using the `update` method), with each `update` done for each `Iterable` of (x, y) pairs in the input stream `xy_seq: Iterator[Iterable[Tuple[X, float]]]`. `iterate_updates` returns an `Iterator` of `FunctionApprox` representing the successively updated `FunctionApprox` instances as a consequence of the repeated invocations to `update`. Note the use of the `rl.iterate.accumulate` function (a wrapped version of `itertools.accumulate`) that calculates accumulated results (including intermediate results) on an `Iterable`, based on a provided function to govern the accumulation. In the code below, the `Iterable` is the input stream `xy_seq_stream` and the function governing the accumulation is the `update` method of `FunctionApprox`.

```
import rl.iterate as iterate

    def iterate_updates(
        self: F,
        xy_seq_stream: Iterator[Iterable[Tuple[X, float]]]
    ) -> Iterator[F]:
        return iterate.accumulate(
            xy_seq_stream,
            lambda fa, xy: fa.update(xy),
            initial=self
        )
```

Next, we write a method called `rmse` to calculate the Root-Mean-Squared-Error of the predictions for x (using `evaluate`) relative to associated (supervisory) y, given as input an `Iterable` of (x, y) pairs. This method will be useful in testing the goodness of a `FunctionApprox` estimate.

```
    def rmse(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]]
    ) -> float:
        x_seq, y_seq = zip(*xy_vals_seq)
        errors: np.ndarray = self.evaluate(x_seq) - np.array(y_seq)
        return np.sqrt(np.mean(errors * errors))
```

Finally, we write a method `argmax` that takes as input an `Iterable` of x values and returns the x value that maximizes $\mathbb{E}_{f(x;w)}[y]$.

```
    def argmax(self, xs: Iterable[X]) -> X:
        return list(xs)[np.argmax(self.evaluate(xs))]
```

The above code for `FunctionApprox` and `Gradient` is in the file rl/function_approx.py. rl/function_approx.py also contains the convenience methods `__add__` (to add two `FunctionApprox`), `__mul__` (to multiply a `FunctionApprox` with a real-valued scalar), and `__call__` (to treat a `FunctionApprox` object syntactically as a function taking an `x: X` as input, essentially a shorthand for `evaluate` on a single x value). `__add__` and `__mul__` are meant to perform element-wise addition and scalar-multiplication on the internal parameters w of the Function Approximation (see Appendix **??** on viewing Function Approximations as Vector Spaces). Likewise, it contains the methods `__add__` and `__mul__` for the `Gradient` class that simply delegates to the `__add__` and `__mul__` methods of the `FunctionApprox` within `Gradient`, and it also contains the method `zero` that returns a `Gradient` which is uniformly zero for each of the parameter values.

Now we are ready to cover a concrete but simple function approximation - the case of linear function approximation.

1.2 Linear Function Approximation

We define a sequence of feature functions

$$\phi_j : \mathcal{X} \to \mathbb{R} \text{ for each } j = 1, 2, \ldots, m$$

and we define $\boldsymbol{\phi} : \mathcal{X} \to \mathbb{R}^m$ as:

$$\boldsymbol{\phi}(x) = (\phi_1(x), \phi_2(x), \ldots, \phi_m(x)) \text{ for all } x \in \mathcal{X}$$

We treat $\boldsymbol{\phi}(x)$ as a column vector for all $x \in \mathcal{X}$.

For linear function approximation, the internal parameters w are represented as a weights column-vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$. Linear function approximation is based on the assumption of a Gaussian distribution for $y|x$ with mean

$$\mathbb{E}_M[y|x] = \sum_{j=1}^{m} \phi_j(x) \cdot w_j = \boldsymbol{\phi}(x)^T \cdot \boldsymbol{w}$$

and constant variance σ^2, i.e.,

$$\mathbb{P}[y|x] = f(x; \boldsymbol{w})(y) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(y - \boldsymbol{\phi}(x)^T \cdot \boldsymbol{w})^2}{2\sigma^2}}$$

So, the cross-entropy loss function (ignoring constant terms associated with σ^2) for a given set of data points $[x_i, y_i | 1 \leq i \leq n]$ is defined as:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} \cdot \sum_{i=1}^{n} (\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w} - y_i)^2$$

Note that this loss function is identical to the mean-squared-error of the linear (in $\boldsymbol{w}$) predictions $\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w}$ relative to the response values y_i associated with the predictor values x_i, over all $1 \leq i \leq n$.

If we include L^2 regularization (with λ as the regularization coefficient), then the regularized loss function is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} (\sum_{i=1}^{n} (\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w} - y_i)^2) + \frac{1}{2} \cdot \lambda \cdot |\boldsymbol{w}|^2$$

The gradient of $\mathcal{L}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$ works out to:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \frac{1}{n} \cdot (\sum_{i=1}^{n} \boldsymbol{\phi}(x_i) \cdot (\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w} - y_i)) + \lambda \cdot \boldsymbol{w}$$

We had said previously that for each concrete function approximation that we'd want to implement, if the Objective $Obj(x_i, y_i)$ is the cross-entropy loss function, we can identify a model-computed value $Out(x_i)$ (either the output of the model or an intermediate computation of the model) such that $\frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)}$ is equal to the prediction error $\mathbb{E}_M[y|x_i] - y_i$ (for each training data point (x_i, y_i)) and we can come up with a numerical algorithm to compute $\nabla_w Out(x_i)$, so that by chain-rule, we have the required gradient $\nabla_w Obj(x_i, y_i)$ (without regularization). In the case of this linear function approximation, the model-computed value $Out(x_i)$ is simply the model prediction for predictor variable x_i, i.e.,

$$Out(x_i) = \mathbb{E}_M[y|x_i] = \boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w}$$

This is confirmed by noting that with $Obj(x_i, y_i)$ set to be the cross-entropy loss function $\mathcal{L}(\boldsymbol{w})$ and $Out(x_i)$ set to be the model prediction $\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w}$ (for training data point (x_i, y_i)),

$$\frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)} = \boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w} - y_i = \mathbb{E}_M[y|x_i] - y_i$$

$$\nabla_{\boldsymbol{w}} Out(x_i) = \nabla_{\boldsymbol{w}} (\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w}) = \boldsymbol{\phi}(x_i)$$

We can solve for $\boldsymbol{w}^*$ by incremental estimation using gradient descent (change in $\boldsymbol{w}$ proportional to the gradient estimate of $\mathcal{L}(\boldsymbol{w})$ with respect to $\boldsymbol{w}$). If the (x_t, y_t) data at time t is:

$$[(x_{t,i}, y_{t,i}) | 1 \leq i \leq n_t]$$

,

then the gradient estimate $\mathcal{G}_{(x_t, y_t)}(\boldsymbol{w}_t)$ at time t is given by:

$$\mathcal{G}_{(x_t, y_t)}(\boldsymbol{w}_t) = \frac{1}{n} \cdot (\sum_{i=1}^{n_t} \boldsymbol{\phi}(x_{t,i}) \cdot (\boldsymbol{\phi}(x_{t,i})^T \cdot \boldsymbol{w}_t - y_{t,i})) + \lambda \cdot \boldsymbol{w}_t$$

which can be interpreted as the mean (over the data in iteration t) of the feature vectors $\boldsymbol{\phi}(x_{t,i})$ weighted by the (scalar) linear prediction errors $\boldsymbol{\phi}(x_{t,i})^T \cdot \boldsymbol{w}_t - y_{t,i}$ (plus regularization term $\lambda \cdot \boldsymbol{w}_t$).

Then, the update to the weights vector $\boldsymbol{w}$ is given by:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \cdot \mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t)$$

where α_t is the learning rate for the gradient descent at time t. To facilitate numerical convergence, we require α_t to be an appropriate function of time t. There are a number of numerical algorithms to achieve the appropriate time-trajectory of α_t. We shall go with one such numerical algorithm - ADAM (Kingma and Ba 2014), which we shall use not just for linear function approximation but later also for the deep neural network function approximation. Before we write code for linear function approximation, we need to write some helper code to implement the ADAM gradient descent algorithm.

We create an `@dataclass Weights` to represent and update the weights (i.e., internal parameters) of a function approximation. The `Weights` dataclass has 5 attributes: `adam_gradient` that captures the ADAM parameters, including the base learning rate and the decay parameters, `time` that represents how many times the weights have been updated, `weights` that represents the weight parameters of the function approximation as a numpy array (1-D array for linear function approximation and 2-D array for each layer of deep neural network function approximation), and the two ADAM cache parameters. The `@staticmethod create` serves as a factory method to create a new instance of the `Weights` dataclass. The `update` method of this `Weights` dataclass produces an updated instance of the `Weights` dataclass that represents the updated weight parameters together with the incremented `time` and the updated ADAM cache parameters. We will follow a programming design pattern wherein we don't update anything in-place - rather, we create a new object with updated values (using the `dataclasses.replace` function). This ensures we don't get unexpected/undesirable updates in-place, which are typically the cause of bugs in numerical code. Finally, we write the `within` method which will be required to implement the `within` method in the linear function approximation class as well as in the deep neural network function approximation class.

```
SMALL_NUM = 1e-6
from dataclasses import replace

@dataclass(frozen=True)
class AdamGradient:
    learning_rate: float
    decay1: float
    decay2: float

    @staticmethod
    def default_settings() -> AdamGradient:
        return AdamGradient(
            learning_rate=0.001,
            decay1=0.9,
            decay2=0.999
        )

@dataclass(frozen=True)
class Weights:
    adam_gradient: AdamGradient
    time: int
    weights: np.ndarray
    adam_cache1: np.ndarray
    adam_cache2: np.ndarray
```

```
    @staticmethod
    def create(
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        weights: np.ndarray,
        adam_cache1: Optional[np.ndarray] = None,
        adam_cache2: Optional[np.ndarray] = None
    ) -> Weights:
        return Weights(
            adam_gradient=adam_gradient,
            time=0,
            weights=weights,
            adam_cache1=np.zeros_like(
                weights
            ) if adam_cache1 is None else adam_cache1,
            adam_cache2=np.zeros_like(
                weights
            ) if adam_cache2 is None else adam_cache2
        )

    def update(self, gradient: np.ndarray) -> Weights:
        time: int = self.time + 1
        new_adam_cache1: np.ndarray = self.adam_gradient.decay1 * \
            self.adam_cache1 + (1 - self.adam_gradient.decay1) * gradient
        new_adam_cache2: np.ndarray = self.adam_gradient.decay2 * \
            self.adam_cache2 + (1 - self.adam_gradient.decay2) * gradient ** 2
        corrected_m: np.ndarray = new_adam_cache1 / \
            (1 - self.adam_gradient.decay1 ** time)
        corrected_v: np.ndarray = new_adam_cache2 / \
            (1 - self.adam_gradient.decay2 ** time)

        new_weights: np.ndarray = self.weights - \
            self.adam_gradient.learning_rate * corrected_m / \
            (np.sqrt(corrected_v) + SMALL_NUM)

        return replace(
            self,
            time=time,
            weights=new_weights,
            adam_cache1=new_adam_cache1,
            adam_cache2=new_adam_cache2,
        )

    def within(self, other: Weights[X], tolerance: float) -> bool:
        return np.all(np.abs(self.weights - other.weights) <= tolerance).item()
```

With this `Weights` class, we are ready to write the dataclass `LinearFunctionApprox` for linear function approximation, inheriting from the abstract base class `FunctionApprox`. It has attributes `feature_functions` that represents $\phi_j : \mathcal{X} \to \mathbb{R}$ for all $j = 1, 2, \ldots, m$, `regularization_coeff` that represents the regularization coefficient λ, `weights` which is an instance of the `Weights` class we wrote above, and `direct_solve` (which we will explain shortly). The static method `create` serves as a factory method to create a new instance of `LinearFunctionApprox`. The method `get_feature_values` takes as input an `x_values_seq: Iterable[X]` (representing a sequence or stream of $x \in \mathcal{X}$), and produces as output the corresponding feature vectors $\boldsymbol{\phi}(x) \in \mathbb{R}^m$ for each x in the input. The feature vectors are output in the form of a 2-D numpy array, with each feature vector $\boldsymbol{\phi}(x)$ (for each x in the input sequence) appearing as a row in the output 2-D numpy array (the number of rows in this numpy array is the length of the input `x_values_seq` and the number of columns is the number of feature functions). Note that often we want to include a bias term in our linear function approximation, in which case we need to prepend the sequence of feature functions we provide as input with an artificial feature function `lambda _: 1.` to represent the constant feature with value 1. This will ensure we have a bias weight in addition to each of the weights that serve as coefficients to the (non-artificial) feature functions.

The method `evaluate` (an abstract method in `FunctionApprox`) calculates the prediction $\mathbb{E}_M[y|x]$ for each input x as: $\boldsymbol{\phi}(x)^T \cdot \boldsymbol{w} = \sum_{j=1}^m \phi_j(x) \cdot w_i$. The method `objective_gradient` (from `FunctionApprox`) performs the calculation $\mathcal{G}_{(x_t,y_t)}(\boldsymbol{w}_t)$ shown above: the mean of the feature vectors $\boldsymbol{\phi}(x_{t,i})$ weighted by the (scalar) linear prediction errors $\boldsymbol{\phi}(x_{t,i})^T \cdot \boldsymbol{w}_t - y_{t,i}$ (plus regularization term $\lambda \cdot \boldsymbol{w}_t$). The variable `obj_deriv_out` takes the role of the linear prediction errors, when `objective_gradient` is invoked by the `update` method through the method `update_with_gradient`. The method `update_with_gradient` (from `FunctionApprox`) updates the weights using the calculated gradient along with the ADAM cache updates (invoking the `update` method of the `Weights` class to ensure there are no in-place updates), and returns a new `LinearFunctionApprox` object containing the updated weights.

```
from dataclasses import replace

@dataclass(frozen=True)
class LinearFunctionApprox(FunctionApprox[X]):

    feature_functions: Sequence[Callable[[X], float]]
    regularization_coeff: float
    weights: Weights
    direct_solve: bool

    @staticmethod
    def create(
        feature_functions: Sequence[Callable[[X], float]],
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        regularization_coeff: float = 0.,
        weights: Optional[Weights] = None,
        direct_solve: bool = True
    ) -> LinearFunctionApprox[X]:
        return LinearFunctionApprox(
            feature_functions=feature_functions,
            regularization_coeff=regularization_coeff,
            weights=Weights.create(
                adam_gradient=adam_gradient,
                weights=np.zeros(len(feature_functions))
            ) if weights is None else weights,
            direct_solve=direct_solve
        )

    def get_feature_values(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array(
            [[f(x) for f in self.feature_functions] for x in x_values_seq]
        )

    def objective_gradient(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        obj_deriv_out_fun: Callable[[Sequence[X], Sequence[float]], float]
    ) -> Gradient[LinearFunctionApprox[X]]:
        x_vals, y_vals = zip(*xy_vals_seq)
        obj_deriv_out: np.ndarray = obj_deriv_out_fun(x_vals, y_vals)
        features: np.ndarray = self.get_feature_values(x_vals)
        gradient: np.ndarray = \
            features.T.dot(obj_deriv_out) / len(obj_deriv_out) \
            + self.regularization_coeff * self.weights.weights
        return Gradient(replace(
            self,
            weights=replace(
                self.weights,
                weights=gradient
            )
        ))

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.dot(
```

```
            self.get_feature_values(x_values_seq),
            self.weights.weights
        )

    def update_with_gradient(
        self,
        gradient: Gradient[LinearFunctionApprox[X]]
    ) -> LinearFunctionApprox[X]:
        return replace(
            self,
            weights=self.weights.update(
                gradient.function_approx.weights.weights
            )
        )
```

We also require the `within` method, that simply delegates to the `within` method of the `Weights` class.

```
    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, LinearFunctionApprox):
            return self.weights.within(other.weights, tolerance)
        else:
            return False
```

The only method that remains to be written now is the `solve` method. Note that for linear function approximation, we can directly solve for $\boldsymbol{w}^*$ if the number of feature functions m is not too large. If the entire provided data is $[(x_i, y_i)|1 \leq i \leq n]$, then the gradient estimate based on this data can be set to 0 to solve for $\boldsymbol{w}^*$, i.e.,

$$\frac{1}{n} \cdot (\sum_{i=1}^{n} \boldsymbol{\phi}(x_i) \cdot (\boldsymbol{\phi}(x_i)^T \cdot \boldsymbol{w}^* - y_i)) + \lambda \cdot \boldsymbol{w}^* = 0$$

We denote $\boldsymbol{\Phi}$ as the n rows $\times$ m columns matrix defined as $\boldsymbol{\Phi}_{i,j} = \phi_j(x_i)$ and the column vector $\boldsymbol{Y} \in \mathbb{R}^n$ defined as $\boldsymbol{Y}_i = y_i$. Then we can write the above equation as:

$$\frac{1}{n} \cdot \boldsymbol{\Phi}^T \cdot (\boldsymbol{\Phi} \cdot \boldsymbol{w}^* - \boldsymbol{Y}) + \lambda \cdot \boldsymbol{w}^* = 0$$

$$\Rightarrow (\boldsymbol{\Phi}^T \cdot \boldsymbol{\Phi} + n\lambda \cdot \boldsymbol{I_m}) \cdot \boldsymbol{w}^* = \boldsymbol{\Phi}^T \cdot \boldsymbol{Y}$$

$$\Rightarrow \boldsymbol{w}^* = (\boldsymbol{\Phi}^T \cdot \boldsymbol{\Phi} + n\lambda \cdot \boldsymbol{I_m})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{Y}$$

where $\boldsymbol{I_m}$ is the $m \times m$ identity matrix. Note that this direct linear-algebraic solution for solving a square linear system of equations of size m is computationally feasible only if m is not too large.

On the other hand, if the number of feature functions m is large, then we solve for $\boldsymbol{w}^*$ by repeatedly calling `update`. The attribute `direct_solve: bool` in `LinearFunctionApprox` specifies whether to perform a direct solution (linear algebra calculations shown above) or to perform a sequence of iterative (incremental) `updates` to $\boldsymbol{w}$ using gradient descent. The code below for the method `solve` does exactly this:

```
import itertools
import rl.iterate import iterate

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> LinearFunctionApprox[X]:
```

```
        if self.direct_solve:
            x_vals, y_vals = zip(*xy_vals_seq)
            feature_vals: np.ndarray = self.get_feature_values(x_vals)
            feature_vals_T: np.ndarray = feature_vals.T
            left: np.ndarray = np.dot(feature_vals_T, feature_vals) \
                + feature_vals.shape[0] * self.regularization_coeff * \
                np.eye(len(self.weights.weights))
            right: np.ndarray = np.dot(feature_vals_T, y_vals)
            ret = replace(
                self,
                weights=Weights.create(
                    adam_gradient=self.weights.adam_gradient,
                    weights=np.linalg.solve(left, right)
                )
            )
        else:
            tol: float = 1e-6 if error_tolerance is None else error_tolerance

            def done(
                a: LinearFunctionApprox[X],
                b: LinearFunctionApprox[X],
                tol: float = tol
            ) -> bool:
                return a.within(b, tol)

            ret = iterate.converged(
                self.iterate_updates(itertools.repeat(list(xy_vals_seq))),
                done=done
            )
        return ret
```

The above code is in the file rl/function_approx.py.

1.3 Neural Network Function Approximation

Now we generalize the linear function approximation to accommodate non-linear functions with a simple deep neural network, specifically a feed-forward fully-connected neural network. We work with the same notation $\boldsymbol{\phi}(\cdot) = (\phi_1(\cdot), \phi_2(\cdot), \ldots, \phi_m(\cdot))$ for feature functions that we covered for the case of linear function approximation. Assume we have L hidden layers in the neural network. Layers numbered $l = 0, 1, \ldots, L-1$ carry the hidden layer neurons and layer $l = L$ carries the output layer neurons.

A couple of things to note about our notation for vectors and matrices when performing linear algebra operations: Vectors will be treated as column vectors (including gradient of a scalar with respect to a vector). The gradient of a vector of dimension m with respect to a vector of dimension n is expressed as a Jacobian matrix with m rows and n columns. We use the notation $dim(\boldsymbol{v})$ to refer to the dimension of a vector $\boldsymbol{v}$.

We denote the input to layer l as vector $\boldsymbol{i_l}$ and the output to layer l as vector $\boldsymbol{o_l}$, for all $l = 0, 1, \ldots, L$. Denoting the predictor variable as $x \in \mathcal{X}$, the response variable as $y \in \mathbb{R}$, and the neural network as model M to predict the expected value of y conditional on x, we have:

$$\boldsymbol{i_0} = \boldsymbol{\phi}(x) \in \mathbb{R}^m \text{ and } \boldsymbol{o_L} = \mathbb{E}_M[y|x] \text{ and } \boldsymbol{i_{l+1}} = \boldsymbol{o_l} \text{ for all } l = 0, 1, \ldots, L-1 \tag{1.1}$$

We denote the parameters for layer l as the matrix $\boldsymbol{w_l}$ with $dim(\boldsymbol{o_l})$ rows and $dim(\boldsymbol{i_l})$ columns. Note that the number of neurons in layer l is equal to $dim(\boldsymbol{o_l})$. Since we are

restricting ourselves to scalar y, $dim(\boldsymbol{o_L}) = 1$ and so, the number of neurons in the output layer is 1.

The neurons in layer l define a linear transformation from layer input $\boldsymbol{i_l}$ to a variable we denote as $\boldsymbol{s_l}$. Therefore,

$$\boldsymbol{s_l} = \boldsymbol{w_l} \cdot \boldsymbol{i_l} \text{ for all } l = 0, 1, \ldots, L \tag{1.2}$$

We denote the activation function of layer l as $g_l : \mathbb{R} \to \mathbb{R}$ for all $l = 0, 1, \ldots, L$. The activation function g_l applies point-wise on each dimension of vector $\boldsymbol{s_l}$, so we take notational liberty with g_l by writing:

$$\boldsymbol{o_l} = g_l(\boldsymbol{s_l}) \text{ for all } l = 0, 1, \ldots, L \tag{1.3}$$

Equations (1.1), (1.2) and (1.3) together define the calculation of the neural network prediction $\boldsymbol{o_L}$ (associated with the response variable y), given the predictor variable x. This calculation is known as *forward-propagation* and will define the `evaluate` method of the deep neural network function approximation class we shall soon write.

Our goal is to derive an expression for the cross-entropy loss gradient $\nabla_{\boldsymbol{w_l}}\mathcal{L}$ for all $l = 0, 1, \ldots, L$. For ease of understanding, our following exposition will be expressed in terms of the cross-entropy loss function for a single predictor variable input $x \in \mathcal{X}$ and it's associated single response variable $y \in \mathbb{R}$ (the code will generalize appropriately to the cross-entropy loss function for a given set of data points $[x_i, y_i | 1 \leq i \leq n]$).

We can reduce this problem of calculating the cross-entropy loss gradient to the problem of calculating $\boldsymbol{P_l} = \nabla_{\boldsymbol{s_l}}\mathcal{L}$ for all $l = 0, 1, \ldots, L$, as revealed by the following chain-rule calculation:

$$\nabla_{\boldsymbol{w_l}}\mathcal{L} = (\nabla_{\boldsymbol{s_l}}\mathcal{L})^T \cdot \nabla_{\boldsymbol{w_l}}\boldsymbol{s_l} = \boldsymbol{P_l}^T \cdot \nabla_{\boldsymbol{w_l}}\boldsymbol{s_l} = \boldsymbol{P_l} \cdot \boldsymbol{i_l}^T \text{ for all } l = 0, 1, \ldots L$$

Note that $\boldsymbol{P_l} \cdot \boldsymbol{i_l}^T$ represents the outer-product of the $dim(\boldsymbol{o_l})$-size vector $\boldsymbol{P_l}$ and the $dim(\boldsymbol{i_l})$-size vector $\boldsymbol{i_l}$ giving a matrix of size $dim(\boldsymbol{o_l}) \times dim(\boldsymbol{i_l})$.

If we include L^2 regularization (with λ_l as the regularization coefficient for layer l), then:

$$\nabla_{\boldsymbol{w_l}}\mathcal{L} = \boldsymbol{P_l} \cdot \boldsymbol{i_l}^T + \lambda_l \cdot \boldsymbol{w_l} \text{ for all } l = 0, 1, \ldots, L \tag{1.4}$$

Here's the summary of our notation:

Notation	Description
$\boldsymbol{i_l}$	Vector Input to layer l for all $l = 0, 1, \ldots, L$
$\boldsymbol{o_l}$	Vector Output of layer l for all $l = 0, 1, \ldots, L$
$\boldsymbol{\phi}(x)$	Feature Vector for predictor variable x
y	Response variable associated with predictor variable x
$\boldsymbol{w_l}$	Matrix of Parameters for layer l for all $l = 0, 1, \ldots, L$
$g_l(\cdot)$	Activation function for layer l for $l = 0, 1, \ldots, L$
$\boldsymbol{s_l}$	$\boldsymbol{s_l} = \boldsymbol{w_l} \cdot \boldsymbol{i_l}, \boldsymbol{o_l} = g_l(\boldsymbol{s_l})$ for all $l = 0, 1, \ldots L$
$\boldsymbol{P_l}$	$\boldsymbol{P_l} = \nabla_{\boldsymbol{s_l}}\mathcal{L}$ for all $l = 0, 1, \ldots, L$
λ_l	Regularization coefficient for layer l for all $l = 0, 1, \ldots, L$

Now that we have reduced the loss gradient calculation to calculation of $\boldsymbol{P_l}$, we spend the rest of this section deriving the analytical calculation of $\boldsymbol{P_l}$. The following theorem tells us that $\boldsymbol{P_l}$ has a recursive formulation that forms the core of the *back-propagation* algorithm for a feed-forward fully-connected deep neural network.

Theorem 1.3.1. *For all $l = 0, 1, \ldots, L-1$,*

$$\boldsymbol{P_l} = (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{s_l})$$

where the symbol $\circ$ represents the Hadamard Product, i.e., point-wise multiplication of two vectors of the same dimension.

Proof. We start by applying the chain rule on $\boldsymbol{P_l}$.

$$\boldsymbol{P_l} = \nabla_{\boldsymbol{s_l}} \mathcal{L} = (\nabla_{\boldsymbol{s_l}} \boldsymbol{s_{l+1}})^T \cdot \nabla_{\boldsymbol{s_{l+1}}} \mathcal{L} = (\nabla_{\boldsymbol{s_l}} \boldsymbol{s_{l+1}})^T \cdot \boldsymbol{P_{l+1}} \tag{1.5}$$

Next, note that:

$$\boldsymbol{s_{l+1}} = \boldsymbol{w_{l+1}} \cdot g_l(\boldsymbol{s_l})$$

Therefore,

$$\nabla_{\boldsymbol{s_l}} \boldsymbol{s_{l+1}} = \boldsymbol{w_{l+1}} \cdot \boldsymbol{Diagonal}(g_l'(\boldsymbol{s_l}))$$

where the notation $\boldsymbol{Diagonal}(\boldsymbol{v})$ for an m-dimensional vector $\boldsymbol{v}$ represents an $m \times m$ diagonal matrix whose elements are the same (also in same order) as the elements of $\boldsymbol{v}$.

Substituting this in Equation (1.5) yields:

$$\boldsymbol{P_l} = (\boldsymbol{w_{l+1}} \cdot \boldsymbol{Diagonal}(g_l'(\boldsymbol{s_l})))^T \cdot \boldsymbol{P_{l+1}} = \boldsymbol{Diagonal}(g_l'(\boldsymbol{s_l})) \cdot \boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}$$

$$= g_l'(\boldsymbol{s_l}) \circ (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) = (\boldsymbol{w_{l+1}^T} \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{s_l})$$

□

Now all we need to do is to calculate $\boldsymbol{P_L} = \nabla_{\boldsymbol{s_L}} \mathcal{L}$ so that we can run this recursive formulation for $\boldsymbol{P_l}$, estimate the loss gradient $\nabla_{\boldsymbol{w_l}} \mathcal{L}$ for any given data (using Equation (1.4)), and perform gradient descent to arrive at $\boldsymbol{w_l^*}$ for all $l = 0, 1, \ldots L$.

Firstly, note that $\boldsymbol{s_L}, \boldsymbol{o_L}, \boldsymbol{P_L}$ are all scalars, so let's just write them as s_L, o_L, P_L respectively (without the bold-facing) to make it explicit in the derivation that they are scalars. Specifically, the gradient

$$\nabla_{\boldsymbol{s_L}} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial s_L}$$

To calculate $\frac{\partial \mathcal{L}}{\partial s_L}$, we need to assume a functional form for $\mathbb{P}[y|s_L]$. We work with a fairly generic exponential functional form for the probability distribution function:

$$p(y|\theta, \tau) = h(y, \tau) \cdot e^{\frac{\theta \cdot y - A(\theta)}{d(\tau)}}$$

where θ should be thought of as the "center" parameter (related to the mean) of the probability distribution and τ should be thought of as the "dispersion" parameter (related to the variance) of the distribution. $h(\cdot, \cdot), A(\cdot), d(\cdot)$ are general functions whose specializations define the family of distributions that can be modeled with this fairly generic exponential functional form (note that this structure is adopted from the framework of Generalized Linear Models).

For our neural network function approximation, we assume that τ is a constant, and we set θ to be s_L. So,

$$\mathbb{P}[y|s_L] = p(y|s_L, \tau) = h(y, \tau) \cdot e^{\frac{s_L \cdot y - A(s_L)}{d(\tau)}}$$

We require the scalar prediction of the neural network $o_L = g_L(s_L)$ to be equal to $\mathbb{E}_p[y|s_L]$. So the question is: What function $g_L : \mathbb{R} \to \mathbb{R}$ (in terms of the functional form of $p(y|s_L, \tau)$) would satisfy the requirement of $o_L = g_L(s_L) = \mathbb{E}_p[y|s_L]$? To answer this question, we first establish the following Lemma:

Lemma 1.3.2.

$$\mathbb{E}_p[y|s_L] = A'(s_L)$$

Proof. Since

$$\int_{-\infty}^{\infty} p(y|s_L, \tau) \cdot dy = 1,$$

the partial derivative of the left-hand-side of the above equation with respect to s_L is zero. In other words,

$$\frac{\partial\{\int_{-\infty}^{\infty} p(y|s_L, \tau) \cdot dy\}}{\partial s_L} = 0$$

Hence,

$$\frac{\partial\{\int_{-\infty}^{\infty} h(y, \tau) \cdot e^{\frac{s_L \cdot y - A(s_L)}{d(\tau)}} \cdot dy\}}{\partial s_L} = 0$$

Taking the partial derivative inside the integral, we get:

$$\int_{-\infty}^{\infty} h(y, \tau) \cdot e^{\frac{s_L \cdot y - A(s_L)}{d(\tau)}} \cdot \frac{y - A'(s_L)}{d(\tau)} \cdot dy = 0$$

$$\Rightarrow \int_{-\infty}^{\infty} p(y|s_L, \tau) \cdot (y - A'(s_L)) \cdot dy = 0$$

$$\Rightarrow \mathbb{E}_p[y|s_L] = A'(s_L)$$

□

So to satisfy $o_L = g_L(s_L) = \mathbb{E}_p[y|s_L]$, we require that

$$o_L = g_L(s_L) = A'(s_L) \tag{1.6}$$

The above equation is important since it tells us that the output layer activation function $g_L(\cdot)$ must be set to be the derivative of the $A(\cdot)$ function. In the theory of generalized linear models, the derivative of the $A(\cdot)$ function serves as the *canonical link function* for a given probability distribution of the response variable conditional on the predictor variable.

Now we are equipped to derive a simple expression for P_L.

Theorem 1.3.3.

$$P_L = \frac{\partial \mathcal{L}}{\partial s_L} = \frac{o_L - y}{d(\tau)}$$

Proof. The Cross-Entropy Loss (Negative Log-Likelihood) for a single training data point (x, y) is given by:

$$\mathcal{L} = -\log\left(h(y, \tau)\right) + \frac{A(s_L) - s_L \cdot y}{d(\tau)}$$

Therefore,

$$P_L = \frac{\partial \mathcal{L}}{\partial s_L} = \frac{A'(s_L) - y}{d(\tau)}$$

But from Equation (1.6), we know that $A'(s_L) = o_L$. Therefore,

$$P_L = \frac{\partial \mathcal{L}}{\partial s_L} = \frac{o_L - y}{d(\tau)}$$

□

At each iteration of gradient descent, we require an estimate of the loss gradient up to a constant factor. So we can ignore the constant $d(\tau)$ and simply say that $P_L = o_L - y$ (up to a constant factor). This is a rather convenient estimate of P_L for a given data point (x, y) since it represents the neural network prediction error for that data point. When presented with a sequence of data points $[(x_{t,i}, y_{t,i})|1 \leq i \leq n_t]$ in iteration t, we simply average the prediction errors across these presented data points. Then, beginning with this estimate of P_L, we can use the recursive formulation of $\boldsymbol{P_l}$ (Theorem 1.3.1) to calculate the gradient of the loss function (Equation (1.4)) with respect to all the parameters of the neural network (this is known as the back-propagation algorithm for a fully-connected feed-forward deep neural network).

Here are some common specializations of the functional form for the conditional probability distribution $\mathbb{P}[y|s_L]$, along with the corresponding canonical link function that serves as the activation function g_L of the output layer:

- Normal distribution $y \sim \mathcal{N}(\mu, \sigma^2)$:

$$s_L = \mu, \tau = \sigma, h(y, \tau) = \frac{e^{\frac{-y^2}{2\tau^2}}}{\sqrt{2\pi}\tau}, d(\tau) = \tau, A(s_L) = \frac{s_L^2}{2}$$

$$\Rightarrow o_L = g_L(s_L) = \mathbb{E}[y|s_L] = A'(s_L) = s_L$$

  Hence, the output layer activation function g_L is the identity function. This means that the linear function approximation of the previous section is exactly the same as a neural network with 0 hidden layers (just the output layer) and with the output layer activation function equal to the identity function.
- Bernoulli distribution for binary-valued y, parameterized by p:

$$s_L = \log\left(\frac{p}{1-p}\right), \tau = 1, h(y, \tau) = 1, d(\tau) = 1, A(s_L) = \log\left(1 + e^{s_L}\right)$$

$$\Rightarrow o_L = g_L(s_L) = \mathbb{E}[y|s_L] = A'(s_L) = \frac{1}{1 + e^{-s_L}}$$

  Hence, the output layer activation function g_L is the logistic function. This generalizes to softmax g_L when we generalize this framework to multivariate y, which in turn enables us to classify inputs x into a finite set of categories represented by y as one-hot-encodings.
- Poisson distribution for y parameterized by λ:

$$s_L = \log \lambda, \tau = 1, h(y, \tau) = \frac{1}{y!}, d(\tau) = 1, A(s_L) = e^{s_L}$$

$$\Rightarrow o_L = g_L(s_L) = \mathbb{E}[y|s_L] = A'(s_L) = e^{s_L}$$

  Hence, the output layer activation function g_L is the exponential function.

Now we are ready to write a class for function approximation with the deep neural network framework described above. We assume that the activation functions $g_l(\cdot)$ are identical for all $l = 0, 1, \ldots, L-1$ (known as the hidden layers activation function) and the activation function $g_L(\cdot)$ is known as the output layer activation function. Note that often we want to include a bias term in the linear transformations of the layers. To include a bias term in layer 0, just like in the case of `LinearFuncApprox`, we prepend the sequence of feature functions we want to provide as input with an artificial feature function `lambda _: 1.` to represent the constant feature with value 1. This ensures we have a bias weight in layer 0 in addition to each of the weights (in layer 0) that serve as coefficients to the (non-artificial) feature functions. Moreover, we allow the specification of a `bias` boolean variable to enable a bias term in each if the layers $l = 1, 2, \ldots L$.

Before we develop the code for forward-propagation and back-propagation, we write a `@dataclass` to hold the configuration of a deep neural network (number of `neurons` in the layers, the `bias` boolean variable, hidden layers activation function and output layer activation function).

```
@dataclass(frozen=True)
class DNNSpec:
    neurons: Sequence[int]
    bias: bool
    hidden_activation: Callable[[np.ndarray], np.ndarray]
    hidden_activation_deriv: Callable[[np.ndarray], np.ndarray]
    output_activation: Callable[[np.ndarray], np.ndarray]
    output_activation_deriv: Callable[[np.ndarray], np.ndarray]
```

`neurons` is a sequence of length L specifying $dim(O_0), dim(O_1), \ldots, dim(O_{L-1})$ (note $dim(o_L)$ doesn't need to be specified since we know $dim(o_L) = 1$). If `bias` is set to be `True`, then $dim(I_l) = dim(O_{l-1}) + 1$ for all $l = 1, 2, \ldots L$ and so in the code below, when `bias` is `True`, we'll need to prepend the matrix representing I_l with a vector consisting of all 1s (to incorporate the bias term). Note that along with specifying the hidden and output layers activation functions $g_l(\cdot)$ defined as $g_l(\boldsymbol{s_l}) = \boldsymbol{o_l}$, we also specify the hidden layers activation function derivative (`hidden_activation_deriv`) and the output layer activation function derivative (`output_activation_deriv`) in the form of functions $h_l(\cdot)$ defined as $h_l(g(\boldsymbol{s_l})) = h_l(\boldsymbol{o_l}) = g_l'(\boldsymbol{s_l})$ (as we know, this derivative is required in the back-propagation calculation). We shall soon see that in the code, $h_l(\cdot)$ is a more convenient specification than the direct specification of $g_l'(\cdot)$.

Now we write the `@dataclass DNNApprox` that implements the abstract base class `FunctionApprox`. It has attributes:

- `feature_functions` that represents $\phi_j : \mathcal{X} \to \mathbb{R}$ for all $j = 1, 2, \ldots, m$.
- `dnn_spec` that specifies the neural network configuration (instance of `DNNSpec`).
- `regularization_coeff` that represents the common regularization coefficient λ for the weights across all layers.
- `weights` which is a sequence of `Weights` objects (to represent and update the weights of all layers).

The method `get_feature_values` is identical to the case of `LinearFunctionApprox` producing a matrix with number of rows equal to the number of x values in it's input `x_values_seq: Iterable[X]` and number of columns equal to the number of specified `feature_functions`.

The method `forward_propagation` implements the forward-propagation calculation that was covered earlier (combining Equations (1.1) (potentially adjusted for the bias term, as mentioned above), (1.2) and (1.3)). `forward_propagation` takes as input the same data

type as the input of `get_feature_values` (`x_values_seq: Iterable[X]`) and returns a list with $L+2$ numpy arrays. The last element of the returned list is a 1-D numpy array representing the final output of the neural network: $o_L = \mathbb{E}_M[y|x]$ for each of the x values in the input `x_values_seq`. The remaining $L+1$ elements in the returned list are each 2-D numpy arrays, consisting of $\boldsymbol{i_l}$ for all $l = 0, 1, \ldots L$ (for each of the x values provided as input in `x_values_seq`).

The method `evaluate` (from `FunctionApprox`) returns the last element ($o_L = \mathbb{E}_M[y|x]$) of the list returned by `forward_propagation`.

The method `backward_propagation` is the most important method of `DNNApprox`, calculating $\nabla_{\boldsymbol{w_l}} Obj$ for all $l = 0, 1, \ldots, L$, for some objective function Obj. We had said previously that for each concrete function approximation that we'd want to implement, if the Objective $Obj(x_i, y_i)$ is the cross-entropy loss function, we can identify a model-computed value $Out(x_i)$ (either the output of the model or an intermediate computation of the model) such that $\frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)}$ is equal to the prediction error $\mathbb{E}_M[y|x_i] - y_i$ (for each training data point (x_i, y_i)) and we can come up with a numerical algorithm to compute $\nabla_w Out(x_i)$, so that by chain-rule, we have the required gradient $\nabla_w Obj(x_i, y_i)$ (without regularization). In the case of this DNN function approximation, the model-computed value $Out(x_i)$ is s_L. Thus,

$$\frac{\partial Obj(x_i, y_i)}{\partial Out(x_i)} = \frac{\partial \mathcal{L}}{\partial s_L} = P_L = o_L - y_i = \mathbb{E}_M[y|x_i] - y_i$$

`backward_propagation` takes two inputs:

1. `fwd_prop: Sequence[np.ndarray]` which represents the output of `forward_propagation` except for the last element (which is the final output of the neural network), i.e., a sequence of $L+1$ 2-D numpy arrays representing the inputs to layers $l = 0, 1, \ldots L$ (for each of the `Iterable` of x-values provided as input to the neural network).
2. `obj_deriv_out: np.ndarray`, which represents the partial derivative of an arbitrary objective function Obj with respect to an arbitrary model-produced value Out, evaluated at each of the `Iterable` of (x, y) pairs that are provided as training data.

If we generalize the objective function from the cross-entropy loss function $\mathcal{L}$ to an arbitrary objective function Obj and define $\boldsymbol{P_l}$ to be $\nabla_{\boldsymbol{s_l}} Obj$ (generalized from $\nabla_{\boldsymbol{s_l}} \mathcal{L}$), then the output of `backward_propagation` would be equal to $\boldsymbol{P_l} \cdot \boldsymbol{i_l}^T$ (i.e., without the regularization term) for all $l = 0, 1, \ldots L$.

The first step in `backward_propagation` is to set P_L (variable `deriv` in the code) equal to `obj_deriv_out` (which in the case of cross-entropy loss as Obj and s_L as Out, reduces to the prediction error $\mathbb{E}_M[y|x_i] - y_i$). As we walk back through the layers of the DNN, the variable `deriv` represents $\boldsymbol{P_l} = \nabla_{\boldsymbol{s_l}} Obj$, evaluated for each of the values made available by `fwd_prop` (note that `deriv` is updated in each iteration of the loop reflecting Theorem 1.3.1: $\boldsymbol{P_l} = (\boldsymbol{w_{l+1}}^T \cdot \boldsymbol{P_{l+1}}) \circ g_l'(\boldsymbol{s_l})$). Note also that the returned list `back_prop` is populated with the result of Equation (1.4): $\nabla_{\boldsymbol{w_l}} Obj = \boldsymbol{P_l} \cdot \boldsymbol{i_l}^T$.

The method `objective_gradient` (from `FunctionApprox`) takes as input an `Iterable` of (x, y) pairs and the $\frac{\partial Obj}{\partial Out}$ function, invokes the `forward_propagation` method (to be passed as input to `backward_propagation`), then invokes `backward_propagation`, and finally adds on the regularization term $\lambda \cdot \boldsymbol{w_l}$ to the output of `backward_propagation` to return the gradient $\nabla_{\boldsymbol{w_l}} Obj$ for all $l = 0, 1, \ldots L$.

The method `update_with_gradient` (from `FunctionApprox`) takes as input a gradient (eg: $\nabla_{\boldsymbol{w_l}} Obj$), updates the weights $\boldsymbol{w_l}$ for all $l = 0, 1, \ldots, L$ along with the ADAM cache up-

dates (invoking the `update` method of the `Weights` class to ensure there are no in-place updates), and returns a new instance of `DNNApprox` that contains the updated weights.

Finally, the method `solve` (from `FunctionApprox`) utilizes the method `iterate_updates` (inherited from `FunctionApprox`) along with the method `within` to perform a best-fit of the weights that minimizes the cross-entropy loss function (basically, a series of incremental `updates` based on gradient descent).

```
from dataclasses import replace
import itertools
import rl.iterate import iterate

@dataclass(frozen=True)
class DNNApprox(FunctionApprox[X]):

    feature_functions: Sequence[Callable[[X], float]]
    dnn_spec: DNNSpec
    regularization_coeff: float
    weights: Sequence[Weights]

    @staticmethod
    def create(
        feature_functions: Sequence[Callable[[X], float]],
        dnn_spec: DNNSpec,
        adam_gradient: AdamGradient = AdamGradient.default_settings(),
        regularization_coeff: float = 0.,
        weights: Optional[Sequence[Weights]] = None
    ) -> DNNApprox[X]:
        if weights is None:
            inputs: Sequence[int] = [len(feature_functions)] + \
                [n + (1 if dnn_spec.bias else 0)
                 for i, n in enumerate(dnn_spec.neurons)]
            outputs: Sequence[int] = list(dnn_spec.neurons) + [1]
            wts = [Weights.create(
                weights=np.random.randn(output, inp) / np.sqrt(inp),
                adam_gradient=adam_gradient
            ) for inp, output in zip(inputs, outputs)]
        else:
            wts = weights

        return DNNApprox(
            feature_functions=feature_functions,
            dnn_spec=dnn_spec,
            regularization_coeff=regularization_coeff,
            weights=wts
        )

    def get_feature_values(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array(
            [[f(x) for f in self.feature_functions] for x in x_values_seq]
        )

    def forward_propagation(
        self,
        x_values_seq: Iterable[X]
    ) -> Sequence[np.ndarray]:
        """
        :param x_values_seq: a n-length iterable of input points
        :return: list of length (L+2) where the first (L+1) values
                 each represent the 2-D input arrays (of size n x |i_l|),
                 for each of the (L+1) layers (L of which are hidden layers),
                 and the last value represents the output of the DNN (as a
                 1-D array of length n)
        """
        inp: np.ndarray = self.get_feature_values(x_values_seq)
        ret: List[np.ndarray] = [inp]
        for w in self.weights[:-1]:
```

```
            out: np.ndarray = self.dnn_spec.hidden_activation(
                np.dot(inp, w.weights.T)
            )
            if self.dnn_spec.bias:
                inp = np.insert(out, 0, 1., axis=1)
            else:
                inp = out
            ret.append(inp)
        ret.append(
            self.dnn_spec.output_activation(
                np.dot(inp, self.weights[-1].weights.T)
            )[:, 0]
        )
        return ret

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return self.forward_propagation(x_values_seq)[-1]

    def backward_propagation(
        self,
        fwd_prop: Sequence[np.ndarray],
        obj_deriv_out: np.ndarray
    ) -> Sequence[np.ndarray]:
        """
        :param fwd_prop represents the result of forward propagation (without
        the final output), a sequence of L 2-D np.ndarrays of the DNN.
        : param obj_deriv_out represents the derivative of the objective
        function with respect to the linear predictor of the final layer.

        :return: list (of length L+1) of |o_l| x |i_l| 2-D arrays,
                 i.e., same as the type of self.weights.weights
        This function computes the gradient (with respect to weights) of
        the objective where the output layer activation function
        is the canonical link function of the conditional distribution of y|x
        """
        deriv: np.ndarray = obj_deriv_out.reshape(1, -1)
        back_prop: List[np.ndarray] = [np.dot(deriv, fwd_prop[-1]) /
                                       deriv.shape[1]]
        # L is the number of hidden layers, n is the number of points
        # layer l deriv represents dObj/ds_l where s_l = i_l . weights_l
        # (s_l is the result of applying layer l without the activation func)
        for i in reversed(range(len(self.weights) - 1)):
            # deriv_l is a 2-D array of dimension |o_l| x n
            # The recursive formulation of deriv is as follows:
            # deriv_{l-1} = (weights_l^T inner deriv_l) haddamard g'(s_{l-1}),
            # which is ((|i_l| x |o_l|) inner (|o_l| x n)) haddamard
            # (|i_l| x n), which is (|i_l| x n) = (|o_{l-1}| x n)
            # Note: g'(s_{l-1}) is expressed as hidden layer activation
            # derivative as a function of o_{l-1} (=i_l).
            deriv = np.dot(self.weights[i + 1].weights.T, deriv) * \
                self.dnn_spec.hidden_activation_deriv(fwd_prop[i + 1].T)
            # If self.dnn_spec.bias is True, then i_l = o_{l-1} + 1, in which
            # case # the first row of the calculated deriv is removed to yield
            # a 2-D array of dimension |o_{l-1}| x n.
            if self.dnn_spec.bias:
                deriv = deriv[1:]
            # layer l gradient is deriv_l inner fwd_prop[l], which is
            # of dimension (|o_l| x n) inner (n x (|i_l|) = |o_l| x |i_l|
            back_prop.append(np.dot(deriv, fwd_prop[i]) / deriv.shape[1])
        return back_prop[::-1]

    def objective_gradient(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        obj_deriv_out_fun: Callable[[Sequence[X], Sequence[float]], float]
    ) -> Gradient[DNNApprox[X]]:
        x_vals, y_vals = zip(*xy_vals_seq)
```

```
        obj_deriv_out: np.ndarray = obj_deriv_out_fun(x_vals, y_vals)
        fwd_prop: Sequence[np.ndarray] = self.forward_propagation(x_vals)[:-1]
        gradient: Sequence[np.ndarray] = \
            [x + self.regularization_coeff * self.weights[i].weights
             for i, x in enumerate(self.backward_propagation(
                 fwd_prop=fwd_prop,
                 obj_deriv_out=obj_deriv_out
             ))]
        return Gradient(replace(
            self,
            weights=[replace(w, weights=g) for
                     w, g in zip(self.weights, gradient)]
        ))

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> DNNApprox[X]:
        tol: float = 1e-6 if error_tolerance is None else error_tolerance

        def done(
            a: DNNApprox[X],
            b: DNNApprox[X],
            tol: float = tol
        ) -> bool:
            return a.within(b, tol)

        return iterate.converged(
            self.iterate_updates(itertools.repeat(list(xy_vals_seq))),
            done=done
        )

    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, DNNApprox):
            return all(w1.within(w2, tolerance)
                       for w1, w2 in zip(self.weights, other.weights))
        else:
            return False
```

All of the above code is in the file rl/function_approx.py.

A comprehensive treatment of function approximations using Deep Neural Networks can be found in the Deep Learning book by Goodfellow, Bengio, Courville (Goodfellow, Bengio, and Courville 2016).

Let us now write some code to create function approximations with `LinearFunctionApprox` and `DNNApprox`, given a stream of data from a simple data model - one that has some noise around a linear function. Here's some code to create an Iterator of (x, y) pairs (where $x = (x_1, x_2, x_3)$) for the data model:

$$y = 2 + 10x_1 + 4x_2 - 6x_3 + \mathcal{N}(0, 0.3)$$

```
def example_model_data_generator() -> Iterator[Tuple[Triple, float]]:
    coeffs: Aug_Triple = (2., 10., 4., -6.)
    d = norm(loc=0., scale=0.3)
    while True:
        pt: np.ndarray = np.random.randn(3)
        x_val: Triple = (pt[0], pt[1], pt[2])
        y_val: float = coeffs[0] + np.dot(coeffs[1:], pt) + \
            d.rvs(size=1)[0]
        yield (x_val, y_val)
```

Next we wrap this in an Iterator that returns a certain number of (x, y) pairs upon each request for data points.

```
def data_seq_generator(
    data_generator: Iterator[Tuple[Triple, float]],
    num_pts: int
) -> Iterator[DataSeq]:
    while True:
        pts: DataSeq = list(islice(data_generator, num_pts))
        yield pts
```

Now let's write a function to create a `LinearFunctionApprox`.

```
def feature_functions():
    return [lambda _: 1., lambda x: x[0], lambda x: x[1], lambda x: x[2]]

def adam_gradient():
    return AdamGradient(
        learning_rate=0.1,
        decay1=0.9,
        decay2=0.999
    )

def get_linear_model() -> LinearFunctionApprox[Triple]:
    ffs = feature_functions()
    ag = adam_gradient()
    return LinearFunctionApprox.create(
         feature_functions=ffs,
         adam_gradient=ag,
         regularization_coeff=0.,
         direct_solve=True
    )
```

Likewise, let's write a function to create a `DNNApprox` with 1 hidden layer with 2 neurons and a little bit of regularization since this deep neural network is somewhat over-parameterized to fit the data generated from the linear data model with noise.

```
def get_dnn_model() -> DNNApprox[Triple]:
    ffs = feature_functions()
    ag = adam_gradient()

    def relu(arg: np.ndarray) -> np.ndarray:
        return np.vectorize(lambda x: x if x > 0. else 0.)(arg)

    def relu_deriv(res: np.ndarray) -> np.ndarray:
        return np.vectorize(lambda x: 1. if x > 0. else 0.)(res)

    def identity(arg: np.ndarray) -> np.ndarray:
        return arg

    def identity_deriv(res: np.ndarray) -> np.ndarray:
        return np.ones_like(res)

    ds = DNNSpec(
        neurons=[2],
        bias=True,
        hidden_activation=relu,
        hidden_activation_deriv=relu_deriv,
        output_activation=identity,
        output_activation_deriv=identity_deriv
    )

    return DNNApprox.create(
        feature_functions=ffs,
        dnn_spec=ds,
        adam_gradient=ag,
        regularization_coeff=0.05
    )
```

Now let's write some code to do a `direct_solve` with the `LinearFunctionApprox` based on the data from the data model we have set up.

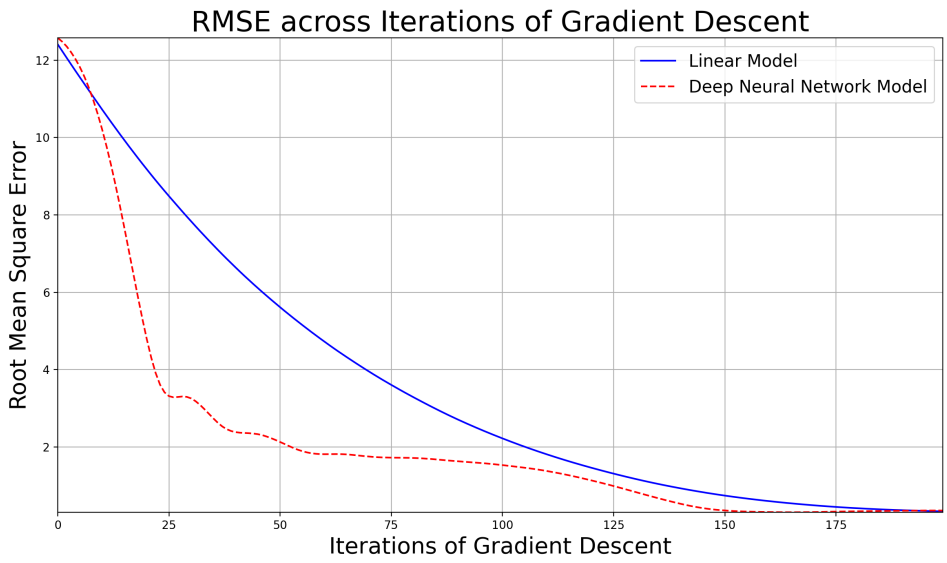

Figure 1.1: SGD Convergence

```
training_num_pts: int = 1000
test_num_pts: int = 10000
training_iterations: int = 200
data_gen: Iterator[Tuple[Triple, float]] = example_model_data_generator()
training_data_gen: Iterator[DataSeq] = data_seq_generator(
    data_gen,
    training_num_pts
)
test_data: DataSeq = list(islice(data_gen, test_num_pts))

direct_solve_lfa: LinearFunctionApprox[Triple] = \
    get_linear_model().solve(next(training_data_gen))
direct_solve_rmse: float = direct_solve_lfa.rmse(test_data)
```

Running the above code, we see that the Root-Mean-Squared-Error (`direct_solve_rmse`) is indeed 0.3, matching the standard deviation of the noise in the linear data model (which is used above to generate the training data as well as the test data).

Now let us perform stochastic gradient descent with instances of `LinearFunctionApprox` and `DNNApprox` and examine the Root-Mean-Squared-Errors on the two function approximations as a function of number of iterations in the gradient descent.

```
linear_model_rmse_seq: Sequence[float] = \
    [lfa.rmse(test_data) for lfa in islice(
        get_linear_model().iterate_updates(training_data_gen),
        training_iterations
    )]

dnn_model_rmse_seq: Sequence[float] = \
    [dfa.rmse(test_data) for dfa in islice(
        get_dnn_model().iterate_updates(training_data_gen),
        training_iterations
    )]
```

The plot of `linear_model_rmse_seq` and `dnn_model_rmse_seq` is shown in Figure 1.1.

1.4 Tabular as a form of `FunctionApprox`

Now we consider a simple case where we have a fixed and finite set of x-values $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, and any data set of (x, y) pairs made available to us needs to have it's x-values from within this finite set $\mathcal{X}$. The prediction $\mathbb{E}[y|x]$ for each $x \in \mathcal{X}$ needs to be calculated only from the y-values associated with this x within the data set of (x, y) pairs. In other words, the y-values in the data associated with other x should not influence the prediction for x. Since we'd like the prediction for x to be $\mathbb{E}[y|x]$, it would make sense for the prediction for a given x to be *some sort of average* of all the y-values associated with x within the data set of (x, y) pairs seen so far. This simple case is refered to as *Tabular* because we can store all $x \in \mathcal{X}$ together with their corresponding predictions $\mathbb{E}[y|x]$ in a finite data structure (loosely refered to as a "table").

So the calculations for Tabular prediction of $\mathbb{E}[y|x]$ is particularly straightforward. What is interesting though is the fact that Tabular prediction actually fits the interface of `FunctionApprox` in terms of the following three methods that we have emphasized as the essence of `FunctionApprox`:

- the `solve` method, that would simply take the average of all the y-values associated with each x in the given data set, and store those averages in a dictionary data structure.
- the `update` method, that would update the current averages in the dictionary data structure, based on the new data set of (x, y) pairs that is provided.
- the `evaluate` method, that would simply look up the dictionary data structure for the y-value averages associated with each x-value provided as input.

This view of Tabular prediction as a special case of `FunctionApprox` also permits us to cast the tabular algorithms of Dynamic Programming and Reinforcement Learning as special cases of the function approximation versions of the algorithms (using the `Tabular` class we develop below).

So now let us write the code for `@dataclass Tabular` as an implementation of the abstract base class `FunctionApprox`. The attributes of `@dataclass Tabular` are:

- `values_map` which is a dictionary mapping each x-value to the average of the y-values associated with x that have been seen so far in the data.
- `counts_map` which is a dictionary mapping each x-value to the count of y-values associated with x that have been seen so far in the data. We need to track the count of y-values associated with each x because this enables us to update `values_map` appropriately upon seeing a new y-value associated with a given x.
- `count_to_weight_func` which defines a function from number of y-values seen so far (associated with a given x) to the weight assigned to the most recent y. This enables us to do a weighted average of the y-values seen so far, controlling the emphasis to be placed on more recent y-values relative to previously seen y-values (associated with a given x).

The `evaluate`, `objective_gradient`, `update_with_gradient`, `solve` and `within` methods should now be self-explanatory.

```
from dataclasses import field, replace

@dataclass(frozen=True)
class Tabular(FunctionApprox[X]):
    values_map: Mapping[X, float] = field(default_factory=lambda: {})
    counts_map: Mapping[X, int] = field(default_factory=lambda: {})
```

```
    count_to_weight_func: Callable[[int], float] = \
        field(default_factory=lambda: lambda n: 1.0 / n)

    def objective_gradient(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        obj_deriv_out_fun: Callable[[Sequence[X], Sequence[float]], float]
    ) -> Gradient[Tabular[X]]:
        x_vals, y_vals = zip(*xy_vals_seq)
        obj_deriv_out: np.ndarray = obj_deriv_out_fun(x_vals, y_vals)
        sums_map: Dict[X, float] = defaultdict(float)
        counts_map: Dict[X, int] = defaultdict(int)
        for x, o in zip(x_vals, obj_deriv_out):
            sums_map[x] += o
            counts_map[x] += 1
        return Gradient(replace(
            self,
            values_map={x: sums_map[x] / counts_map[x] for x in sums_map},
            counts_map=counts_map
        ))

    def evaluate(self, x_values_seq: Iterable[X]) -> np.ndarray:
        return np.array([self.values_map.get(x, 0.) for x in x_values_seq])

    def update_with_gradient(
        self,
        gradient: Gradient[Tabular[X]]
    ) -> Tabular[X]:
        values_map: Dict[X, float] = dict(self.values_map)
        counts_map: Dict[X, int] = dict(self.counts_map)
        for key in gradient.function_approx.values_map:
            counts_map[key] = counts_map.get(key, 0) + \
                gradient.function_approx.counts_map[key]
            weight: float = self.count_to_weight_func(counts_map[key])
            values_map[key] = values_map.get(key, 0.) - \
                weight * gradient.function_approx.values_map[key]
        return replace(
            self,
            values_map=values_map,
            counts_map=counts_map
        )

    def solve(
        self,
        xy_vals_seq: Iterable[Tuple[X, float]],
        error_tolerance: Optional[float] = None
    ) -> Tabular[X]:
        values_map: Dict[X, float] = {}
        counts_map: Dict[X, int] = {}
        for x, y in xy_vals_seq:
            counts_map[x] = counts_map.get(x, 0) + 1
            weight: float = self.count_to_weight_func(counts_map[x])
            values_map[x] = weight * y + (1 - weight) * values_map.get(x, 0.)
        return replace(
            self,
            values_map=values_map,
            counts_map=counts_map
        )

    def within(self, other: FunctionApprox[X], tolerance: float) -> bool:
        if isinstance(other, Tabular):
            return all(abs(self.values_map[s] - other.values_map.get(s, 0.))
                       <= tolerance for s in self.values_map)
        return False
```

Here's a valuable insight: This *Tabular* setting is actually a special case of linear function approximation by setting a feature function $\phi_i(\cdot)$ for each x_i as: $\phi_i(x_i) = 1$ and $\phi_i(x) = 0$

for each $x \neq x_i$ (i.e., $\phi_i(\cdot)$ is the indicator function for x_i, and the $\mathbf{\Phi}$ matrix is the identity matrix), and the corresponding weights w_i equal to the average of the y-values associated with x_i in the given data. This also means that the `count_to_weights_func` plays the role of the learning rate function (as a function of the number of iterations in stochastic gradient descent).

When we implement Approximate Dynamic Programming (ADP) algorithms with the `@abstractclass FunctionApprox` (later in this chapter), using the `Tabular` class (for `FunctionApprox`) enables us to specialize the ADP algorithm implementation to the Tabular DP algorithms (that we covered in Chapter **??**). Note that in the tabular DP algorithms, the set of finite states take the role of $\mathcal{X}$ and the Value Function for a given state $x = s$ takes the role of the "predicted" y-value associated with x. We also note that in the tabular DP algorithms, in each iteration of sweeping through all the states, the Value Function for a state $x = s$ is set to the current y value (not the average of all y-values seen so far). The current y-value is simply the right-hand-side of the Bellman Equation corresponding to the tabular DP algorithm. Consequently, when using `Tabular` class for tabular DP, we'd need to set `count_to_weight_func` to be the function `lambda _: 1` (this is because a weight of 1 for the current y-value sets `values_map[x]` equal to the current y-value).

Likewise, when we implement RL algorithms (using `@abstractclass FunctionApprox`) later in this book, using the `Tabular` class (for `FunctionApprox`) specializes the RL algorithm implementation to Tabular RL. In Tabular RL, we average all the Returns seen so far for a given state. If we choose to do a plain average (equal importance for all y-values seen so far, associated with a given x), then in the `Tabular` class, we'd need to set `count_to_weights_func` to be the function `lambda n: 1. / n`.

We want to emphasize that although tabular algorithms are just a special case of algorithms with function approximation, we give special coverage in this book to tabular algorithms because they help us conceptualize the core concepts in a simple (tabular) setting without the distraction of some of the details and complications in the apparatus of function approximation.

Now we are ready to write algorithms for Approximate Dynamic Programming (ADP). Before we go there, it pays to emphasize that we have described and implemented a fairly generic framework for gradient-based estimation of function approximations, given arbitrary training data. It can be used for arbitrary objective functions and arbitrary functional forms/neural networks (beyond the concrete classes we implemented). We encourage you to explore implementing and using this function approximation code for other types of objectives and other types of functional forms/neural networks.

1.5 Approximate Policy Evaluation

The first Approximate Dynamic Programming (ADP) algorithm we cover is Approximate Policy Evaluation, i.e., evaluating the Value Function for a Markov Reward Process (MRP). Approximate Policy Evaluation is fundamentally the same as Tabular Policy Evaluation in terms of repeatedly applying the Bellman Policy Operator $\boldsymbol{B}^\pi$ on the Value Function $V : \mathcal{N} \to \mathbb{R}$. However, unlike Tabular Policy Evaluation algorithm, here the Value Function $V(\cdot)$ is set up and updated as an instance of `FunctionApprox` rather than as a table of values for the non-terminal states. This is because unlike Tabular Policy Evaluation which operates on an instance of a `FiniteMarkovRewardProcess`, Approximate Policy Evaluation algorithm operates on an instance of `MarkovRewardProcess`. So we do not have an enumeration of states of the MRP and we do not have the transition probabilities of the MRP.

This is typical in many real-world problems where the state space is either very large or is continuous-valued, and the transitions could be too many or could be continuous-valued transitions. So, here's what we do to overcome these challenges:

- We specify a sampling probability distribution of non-terminal states (argument `non_terminal_states_distribution` in the code below) from which we shall sample a specified number (`num_state_samples` in the code below) of non-terminal states, and construct a list of those sampled non-terminal states (`nt_states` in the code below) in each iteration. The type of this probability distribution of non-terminal states is aliased as follows (this type will be used not just for Approximate Dynamic Programming algorithms, but also for Reinforcement Learning algorithms):

```
NTStateDistribution = Distribution[NonTerminal[S]]
```

- We sample pairs of (next state s', reward r) from a given non-terminal state s, and calculate the expectation $\mathbb{E}[r + \gamma \cdot V(s')]$ by averaging $r + \gamma \cdot V(s')$ across the sampled pairs. Note that the method `expectation` of a `Distribution` object performs a sampled expectation. $V(s')$ is obtained from the function approximation instance of `FunctionApprox` that is being updated in each iteration. The type of the function approximation of the Value Function is aliased as follows (this type will be used not just for Approximate Dynamic Programming algorithms, but also for Reinforcement Learning Algorithms).

```
ValueFunctionApprox = FunctionApprox[NonTerminal[S]]
```

- The sampled list of non-terminal states s comprise our x-values and the associated sampled expectations described above comprise our y-values. This list of (x, y) pairs are used to update the approximation of the Value Function in each iteration (producing a new instance of `ValueFunctionApprox` using it's `update` method).

The entire code is shown below. The `evaluate_mrp` method produces an `Iterator` on `ValueFunctionApprox` instances, and the code that calls `evaluate_mrp` can decide when/how to terminate the iterations of Approximate Policy Evaluation.

```
from rl.iterate import iterate

def evaluate_mrp(
    mrp: MarkovRewardProcess[S],
    gamma: float,
    approx_0: ValueFunctionApprox[S],
    non_terminal_states_distribution: NTStateDistribution[S],
    num_state_samples: int
) -> Iterator[ValueFunctionApprox[S]]:
    def update(v: ValueFunctionApprox[S]) -> ValueFunctionApprox[S]:
        nt_states: Sequence[NonTerminal[S]] = \
            non_terminal_states_distribution.sample_n(num_state_samples)

        def return_(s_r: Tuple[State[S], float]) -> float:
            s1, r = s_r
            return r + gamma * extended_vf(v, s1)

        return v.update(
            [(s, mrp.transition_reward(s).expectation(return_))
             for s in nt_states]
        )

    return iterate(update, approx_0)
```

Notice the function `extended_vf` used to evaluate the Value Function for the next state transitioned to. However, the next state could be terminal or non-terminal, and the Value Function is only defined for non-terminal states. `extended_vf` utilizes the method `on_non_terminal` we had written in Chapter **??** when designing the `State` class - it evaluates to the default value of 0 for a terminal state (and evaluates the given `ValueFunctionApprox` for a non-terminal state).

```
def extended_vf(vf: ValueFunctionApprox[S], s: State[S]) -> float:
    return s.on_non_terminal(vf, 0.0)
```

`extended_vf` will be useful not just for Approximate Dynamic Programming algorithms, but also for Reinforcement Learning algorithms.

1.6 Approximate Value Iteration

Now that we've understood and coded Approximate Policy Evaluation (to solve the Prediction problem), we can extend the same concepts to Approximate Value Iteration (to solve the Control problem). The code below in `value_iteration` is almost the same as the code above in `evaluate_mrp`, except that instead of a `MarkovRewardProcess`, here we have a `MarkovDecisionProcess`, and instead of the Bellman Policy Operator update, here we have the Bellman Optimality Operator update. Therefore, in the Value Function update, we maximize the Q-value function (over all actions a) for each non-terminal state s. Also, similar to `evaluate_mrp`, `value_iteration` produces an `Iterator` on `ValueFunctionApprox` instances, and the code that calls `value_iteration` can decide when/how to terminate the iterations of Approximate Value Iteration.

```
from rl.iterate import iterate

def value_iteration(
    mdp: MarkovDecisionProcess[S, A],
    gamma: float,
    approx_0: ValueFunctionApprox[S],
    non_terminal_states_distribution: NTStateDistribution[S],
    num_state_samples: int
) -> Iterator[ValueFunctionApprox[S]]:

    def update(v: ValueFunctionApprox[S]) -> ValueFunctionApprox[S]:
        nt_states: Sequence[NonTerminal[S]] = \
            non_terminal_states_distribution.sample_n(num_state_samples)

        def return_(s_r: Tuple[State[S], float]) -> float:
            s1, r = s_r
            return r + gamma * extended_vf(v, s1)

        return v.update(
            [(s, max(mdp.step(s, a).expectation(return_)
                     for a in mdp.actions(s)))
             for s in nt_states]
        )

    return iterate(update, approx_0)
```

1.7 Finite-Horizon Approximate Policy Evaluation

Next, we move on to Approximate Policy Evaluation in a finite-horizon setting, meaning we perform Approximate Policy Evaluation with a backward induction algorithm, much like how we did backward induction for finite-horizon Tabular Policy Evaluation. We will

of course make the same types of adaptations from Tabular to Approximate as we did in the functions `evaluate_mrp` and `value_iteration` above.

In the `backward_evaluate` code below, the input argument `mrp_f0_mu_triples` is a list of triples, with each triple corresponding to each non-terminal time step in the finite horizon. Each triple consists of:

* An instance of `MarkovRewardProceess` - note that each time step has it's own instance of `MarkovRewardProcess` representation of transitions from non-terminal states s in a time step t to the (state s', reward r) pairs in the next time step $t+1$ (variable `mrp` in the code below). * An instance of `ValueFunctionApprox` to capture the approximate Value Function for the time step (variable `approx0` in the code below represents the initial `ValueFunctionApprox` instances). * A sampling probability distribution of non-terminal states in the time step (variable `mu` in the code below).

The backward induction code below should be pretty self-explanatory. Note that in backward induction, we don't invoke the `update` method of `FunctionApprox` like we did in the non-finite-horizon cases - here we invoke the `solve` method which internally performs a series of `updates` on the `FunctionApprox` for a given time step (until we converge to within a specified level of `error_tolerance`). In the non-finite-horizon cases, it was okay to simply do a single `update` in each iteration because we revisit the same set of states in further iterations. Here, once we converge to an acceptable `ValueFunctionApprox` (using `solve`) for a specific time step, we won't be performing any more updates to the Value Function for that time step (since we move on to the next time step, in reverse). `backward_evaluate` returns an Iterator over `ValueFunctionApprox` objects, from time step 0 to the horizon time step. We should point out that in the code below, we've taken special care to handle terminal states (that occur either at the end of the horizon or can even occur before the end of the horizon) - this is done using the `extended_vf` function we'd written earlier.

```
MRP_FuncApprox_Distribution = Tuple[MarkovRewardProcess[S],
                                    ValueFunctionApprox[S],
                                    NTStateDistribution[S]]

def backward_evaluate(
    mrp_f0_mu_triples: Sequence[MRP_FuncApprox_Distribution[S]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[ValueFunctionApprox[S]]:
    v: List[ValueFunctionApprox[S]] = []

    for i, (mrp, approx0, mu) in enumerate(reversed(mrp_f0_mu_triples)):

        def return_(s_r: Tuple[State[S], float], i=i) -> float:
            s1, r = s_r
            return r + gamma * (extended_vf(v[i-1], s1) if i > 0 else 0.)

        v.append(
            approx0.solve(
                [(s, mrp.transition_reward(s).expectation(return_))
                 for s in mu.sample_n(num_state_samples)],
                error_tolerance
            )
        )

    return reversed(v)
```

1.8 Finite-Horizon Approximate Value Iteration

Now that we've understood and coded finite-horizon Approximate Policy Evaluation (to solve the finite-horizon Prediction problem), we can extend the same concepts to finite-horizon Approximate Value Iteration (to solve the finite-horizon Control problem). The code below in `back_opt_vf_and_policy` is almost the same as the code above in `backward_evaluate`, except that instead of `MarkovRewardProcess`, here we have `MarkovDecisionProcess`. For each non-terminal time step, we maximize the Q-Value function (over all actions a) for each non-terminal state s. `back_opt_vf_and_policy` returns an Iterator over pairs of `ValueFunctionApprox` and `DeterministicPolicy` objects (representing the Optimal Value Function and the Optimal Policy respectively), from time step 0 to the horizon time step.

```
from rl.distribution import Constant
from operator import itemgetter

MDP_FuncApproxV_Distribution = Tuple[
    MarkovDecisionProcess[S, A],
    ValueFunctionApprox[S],
    NTStateDistribution[S]
]

def back_opt_vf_and_policy(
    mdp_f0_mu_triples: Sequence[MDP_FuncApproxV_Distribution[S, A]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[Tuple[ValueFunctionApprox[S], DeterministicPolicy[S, A]]]:
    vp: List[Tuple[ValueFunctionApprox[S], DeterministicPolicy[S, A]]] = []

    for i, (mdp, approx0, mu) in enumerate(reversed(mdp_f0_mu_triples)):

        def return_(s_r: Tuple[State[S], float], i=i) -> float:
            s1, r = s_r
            return r + gamma * (extended_vf(vp[i-1][0], s1) if i > 0 else 0.)

        this_v = approx0.solve(
            [(s, max(mdp.step(s, a).expectation(return_)
                     for a in mdp.actions(s)))
             for s in mu.sample_n(num_state_samples)],
            error_tolerance
        )

        def deter_policy(state: S) -> A:
            return max(
                ((mdp.step(NonTerminal(state), a).expectation(return_), a)
                 for a in mdp.actions(NonTerminal(state))),
                key=itemgetter(0)
            )[1]

        vp.append((this_v, DeterministicPolicy(deter_policy)))

    return reversed(vp)
```

1.9 Finite-Horizon Approximate Q-Value Iteration

The above code for Finite-Horizon Approximate Value Iteration extends the Finite-Horizon Backward Induction Value Iteration algorithm of Chapter **??** by treating the Value Function as a function approximation instead of an exact tabular representation. However, there is an alternative (and arguably simpler and more effective) way to solve the Finite-Horizon Control problem - we can perform backward induction on the optimal Action-Value (Q-value) Function instead of the optimal (State-)Value Function. In general, the key advantage of working with the optimal Action Value function is that it has all the information

necessary to extract the optimal State-Value function and the optimal Policy (since we just need to perform a max / arg max over all the actions for any non-terminal state). This contrasts with the case of working with the optimal State-Value function which requires us to also avail of the transition probabilities, rewards and discount factor in order to extract the optimal policy. We shall see later that Reinforcement Learning algorithms for Control work with Action-Value (Q-Value) Functions for this very reason.

Performing backward induction on the optimal Q-value function means that knowledge of the optimal Q-value function for a given time step t immediately gives us the optimal State-Value function and the optimal policy for the same time step t. This contrasts with performing backward induction on the optimal State-Value function - knowledge of the optimal State-Value function for a given time step t cannot give us the optimal policy for the same time step t (for that, we need the optimal State-Value function for time step $t+1$ and furthermore, we also need the t to $t+1$ state/reward transition probabilities).

So now we develop an algorithm that works with a function approximation for the Q-Value function and steps back in time similar to the backward induction we had performed earlier for the (State-)Value function. Just like we defined an alias type `ValueFunctionApprox` for the State-Value function, we define an alias type `QValueFunctionApprox` for the Action-Value function, as follows:

```
QValueFunctionApprox = FunctionApprox[Tuple[NonTerminal[S], A]]
```

The code below in `back_opt_qvf` is quite similar to `back_opt_vf_and_policy` above. The key difference is that we have `QValueFunctionApprox` in the input to the function rather than `ValueFunctionApprox` to reflect the fact that we are approximating $Q_t^* : \mathcal{N}_t \times \mathcal{A}_t \to \mathbb{R}$ for all time steps t in the finite horizon. For each non-terminal time step, we express the Q-value function (for a set of sample non-terminal states s and for all actions a) in terms of the Q-value function approximation of the next time step. This is essentially the MDP Action-Value Function Bellman Optimality Equation for the finite-horizon case (adapted to function approximation). `back_opt_qvf` returns an Iterator over `QValueFunctionApprox` (representing the Optimal Q-Value Function), from time step 0 to the horizon time step. We can then obtain V_t^* (Optimal State-Value Function) and π_t^* for each t by simply performing a max / arg max over all actions $a \in \mathcal{A}_t$ of $Q_t^*(s, a)$ for any $s \in \mathcal{N}_t$.

```
MDP_FuncApproxQ_Distribution = Tuple[
    MarkovDecisionProcess[S, A],
    QValueFunctionApprox[S, A],
    NTStateDistribution[S]
]

def back_opt_qvf(
    mdp_f0_mu_triples: Sequence[MDP_FuncApproxQ_Distribution[S, A]],
    gamma: float,
    num_state_samples: int,
    error_tolerance: float
) -> Iterator[QValueFunctionApprox[S, A]]:
    horizon: int = len(mdp_f0_mu_triples)
    qvf: List[QValueFunctionApprox[S, A]] = []

    for i, (mdp, approx0, mu) in enumerate(reversed(mdp_f0_mu_triples)):

        def return_(s_r: Tuple[State[S], float], i=i) -> float:
            s1, r = s_r
            next_return: float = max(
                qvf[i-1]((s1, a)) for a in
                mdp_f0_mu_triples[horizon - i][0].actions(s1)
            ) if i > 0 and isinstance(s1, NonTerminal) else 0.
            return r + gamma * next_return
```

```
            this_qvf = approx0.solve(
                [((s, a), mdp.step(s, a).expectation(return_))
                 for s in mu.sample_n(num_state_samples) for a in mdp.actions(s)],
                error_tolerance
            )
            qvf.append(this_qvf)
        return reversed(qvf)
```

We should also point out here that working with the optimal Q-value function (rather than the optimal State-Value function) in the context of ADP prepares us nicely for RL because RL algorithms typically work with the optimal Q-value function instead of the optimal State-Value function.

All of the above code for Approximate Dynamic Programming (ADP) algorithms is in the file rl/approximate_dynamic_programming.py. We encourage you to create instances of `MarkovRewardProcess` and `MarkovDecisionProcess` (including finite-horizon instances) and play with the above ADP code with different choices of function approximations, non-terminal state sampling distributions, and number of samples. A simple but valuable exercise is to reproduce the tabular versions of these algorithms by using the `Tabular` implementation of `FunctionApprox` (note: the `count_to_weights_func` would then need to be lambda _: 1.) in the above ADP functions.

1.10 How to Construct the Non-Terminal States Distribution

Each of the above ADP algorithms takes as input probability distribution(s) of non-terminal states. You may be wondering how one constructs the probability distribution of non-terminal states so you can feed it as input to any of these ADP algorithm. There is no simple, crisp answer to this. But we will provide some general pointers in this section on how to construct the probability distribution of non-terminal states.

Let us start with Approximate Policy Evaluation and Approximate Value Iteration algorithms. They require as input the probability distribution of non-terminal states. For Approximate Value Iteration algorithm, a natural choice would be evaluate the Markov Decision Process (MDP) with a uniform policy (equal probability for each action, from any state) to construct the implied Markov Reward Process (MRP), and then infer the stationary distribution of it's Markov Process, using some special property of the Markov Process (for instance, if it's a finite-states Markov Process, we might be able to perform the matrix calculations we covered in Chapter **??** to calculate the stationary distribution). The stationary distribution would serve as the probability distribution of non-terminal states to be used by the Approximate Value Iteration algorithm. For Approximate Policy Evaluation algorithm, we do the same stationary distribution calculation with the given MRP. If we cannot take advantage of any special properties of the given MDP/MRP, then we can run a simulation with the `simulate` method in `MarkovRewardProcess` (inherited from `MarkovProcess`) and create a `SampledDistribution` of non-terminal states based on the non-terminal states reached by the sampling traces after a sufficiently large (but fixed) number of time steps (this is essentially an estimate of the stationary distribution). If the above choices are infeasible or computationally expensive, then a simple and neutral choice is to use a uniform distribution over the states.

Next, we consider the backward induction ADP algorithms for finite-horizon MDPs/MRPs. Our job here is to infer the distribution of non-terminal states for each time step in the finite

horizon. Sometimes you can take advantage of the mathematical structure of the underlying Markov Process to come up with an analytical expression (exact or approximate) for the probability distribution of non-terminal states at any time step for the underlying Markov Process of the MRP/implied-MRP. For instance, if the Markov Process is described by a stochastic differential equation (SDE) and if we are able to solve the SDE, we would know the analytical expression for the probability distribution of non-terminal states. If we cannot take advantage of any such special properties, then we can generate sampling traces by time-incrementally sampling from the state-transition probability distributions of each of the Markov Reward Processes at each time step (if we are solving a Control problem, then we create implied-MRPs by evaluating the given MDPs with a uniform policy). The states reached by these sampling traces at any fixed time step provide a `SampledDistribution` of non-terminal states for that time step. If the above choices are infeasible or computationally expensive, then a simple and neutral choice is to use a uniform distribution over the non-terminal states for each time step.

We will write some code in Chapter **??** to create a `SampledDistribution` of non-terminal states for each time step of a finite-horizon problem by stitching together samples of state transitions at each time step. If you are curious about this now, you can take a peek at the code in rl/chapter7/asset_alloc_discrete.py.

1.11 Key Takeaways from this Chapter

- The Function Approximation interface involves two key methods - A) updating the parameters of the Function Approximation based on training data available from each iteration of a data stream, and B) evaluating the expectation of the response variable whose conditional probability distribution is modeled by the Function Approximation. Linear Function Approximation and Deep Neural Network Function Approximation are the two main Function Approximations we've implemented and will be using in the rest of the book.
- Tabular satisfies the interface of Function Approximation, and can be viewed as a special case of linear function approximation with feature functions set to be indicator functions for each of the x values.
- All the Tabular DP algorithms can be generalized to ADP algorithms replacing tabular Value Function updates with updates to Function Approximation parameters (where the Function Approximation represents the Value Function). Sweep over all states in the tabular case is replaced by sampling states in the ADP case. Expectation calculations in Bellman Operators are handled in ADP as averages of the corresponding calculations over transition samples (versus calculations using explicit transition probabilities in the tabular algorithms).

Bibliography

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Kingma, Diederik P., and Jimmy Ba. 2014. "Adam: A Method for Stochastic Optimization." *CoRR* abs/1412.6980. http://dblp.uni-trier.de/db/journals/corr/corr1412.html#KingmaB14.