DSCI 565: LINEAR NEURAL NETWORKS FOR REGRESSION

Ke-Thia Yao

Lecture 3: 2025 September 3

Linear Regression

- Linear regression assumes the relationship between features and target is approximately linear

$$\text{price} = w_{area} \times \text{area} + w_{age} \times \text{age} + b$$

- Where w_{area} and w_{age} are the weights, and b is the bias

Matrix Notation

$\hat{y} = Xw + b$

$n \times d \quad d \times 1 \quad n \times 1$

- If we represent an entire dataset of n examples and d attributes as a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, then

$$\hat{\boldsymbol{y}} = \mathbf{X}\boldsymbol{w} + b$$

- Where $\boldsymbol{w}$ is the weight vector and b is the bias

Loss Function

- The loss function measures how well the model fits the data
- The smaller the loss value, the better the fit
- The **squared error** loss for an example i is

$$l^{(i)}(\boldsymbol{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

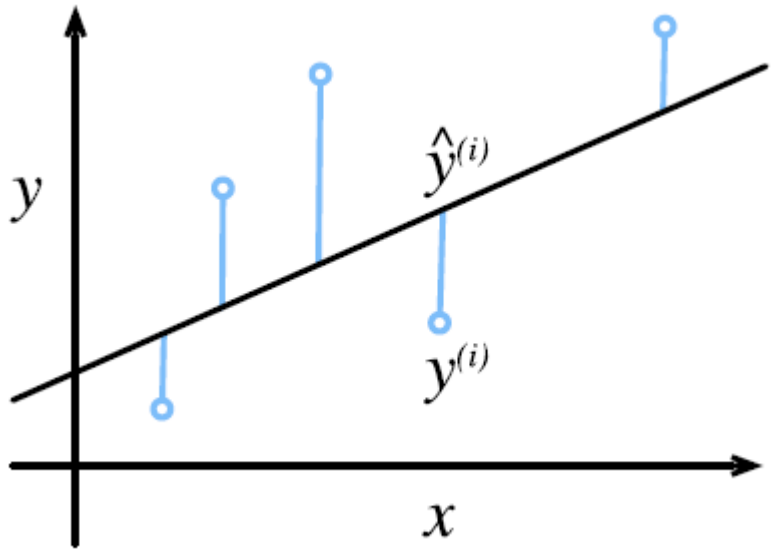
- Loss for the entire dataset

$\because \hat{y}_i = W^T x_i$

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b - y^{(i)}\right)^2$$

$$= \|\boldsymbol{y} - \mathbf{X}\boldsymbol{w}\|^2$$

Fold b in $\boldsymbol{w}$



Analytic Solution

$w = (X^T X)^{-1} X^T y$

- The squared error loss for linear regression has a global minima:

$$\partial_{\boldsymbol{w}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|^2 = -2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) = 0$$

- Then

$2X^T y = 2X^T X w$

$$\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}$$

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

- The matrix inverse exists if $\boldsymbol{X}$ matrix has full rank

$\Rightarrow$ closed-form solution

Minibatch Stochastic Gradient Descent

- Most loss equations do not have analytical solutions
- Must use iterative methods to gradually reduce the error by updating the parameters (weights and bias)
- Various gradient descent methods differs by how many examples are used to compute the gradient

Minibatch SDG Update Function

- Update weight and bias by subtracting the gradient of the loss function

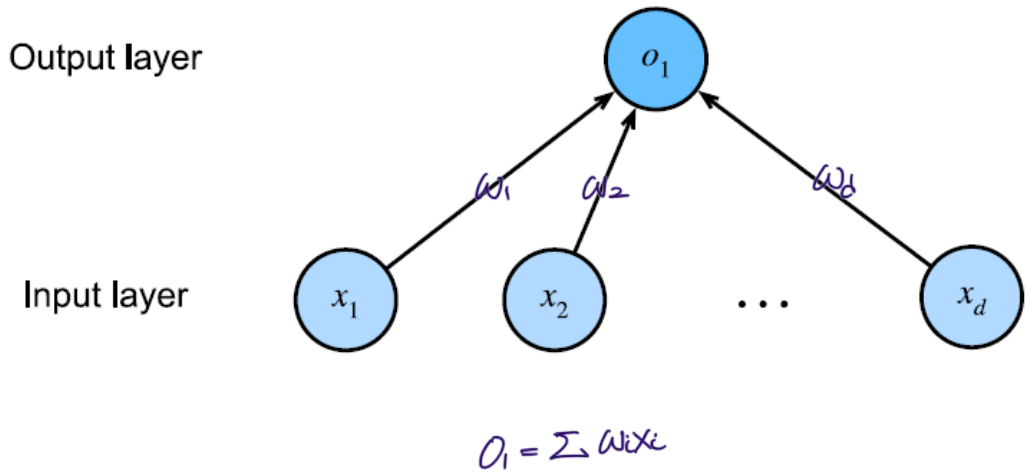
$$(\boldsymbol{w}, b) \leftarrow (\boldsymbol{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\boldsymbol{w},b)} l^{(i)}(\boldsymbol{w}, b)$$

  where η is the learning rate

- For quadratic loss and linear regression, update for weight is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\boldsymbol{w}} l^{(i)}(\boldsymbol{w}, b) = \boldsymbol{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \boldsymbol{x}^{(i)}(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b - y^{(i)})$$

Linear Regression as a Neural Network



- Regression is single-layer network
- The inputs are $x_1, x_2, \ldots, x_d$
- There is only a single neuron
- The output o_1 is computed by multiplying the inputs by the weights on the edges

Object-Oriented Design Implementation: d2l

- Three top-level classes:
  - Class Module contains the models, losses and optimization methods
  - Class DataModel contains data loaders for training and validation
  - Class Trainer combines Module and DataModel to perform training
- Helper utility
  - Class ProgressBoard creates incremental train and validation plots
  - Class HyperParameters saves __init()__ arguments as object attributes

Python Review: Callable


```
class Multiply:
    def __init__(self, factor):
        self.factor = 2
    def __call__(self, x):
        return self.factor * x
>>> m2 = Multiply(2)
>>> m2(3.2)
6.4
```

Python Review: Function Decorator


```
import time
def timer_decorator(func):
    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print ('Time elapsed:', end - start)
    return wrapper

def print_hello():
    print('Hello’)

time_print_hello = timer_decorator(print_hello)

@timer_decorator
def print_hi():
    for i in range(5):
      print('hi')
```

Python Review: Function Decorator


```
def add_to_class(Class): #@save
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
    return wrapper
class A:
    def __init__(self):
        self.b = 1

@add_to_class(A)
def do(self):
    print('Class attribute "b" is', self.b)

a = A()
a.do()
```

See Notebook

- Objected-Oriented Design: chapter_linear-regression/oo-design.ipynb

Utility Classes

d2l.HyperParameters

```
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('There is no self.c =', not hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
There is no self.c = True
```

D2l.ProgressBoard

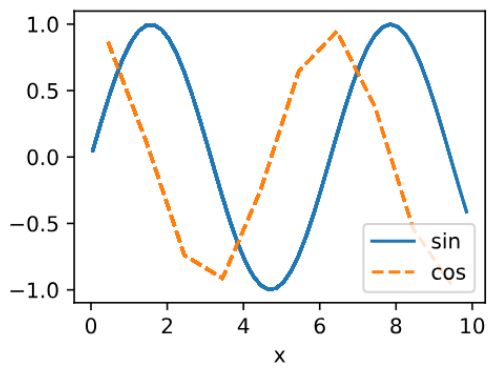
```
class ProgressBoard(d2l.HyperParameters):  #@save
    """The board that plots data points in animation."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 ls=['-', '--', '-.', ':'], colors=['C0', 'C1', 'C2', 'C3'],
                 fig=None, axes=None, figsize=(3.5, 2.5), display=True):
        self.save_hyperparameters()

    def draw(self, x, y, label, every_n=1):
        raise NotImplemented
```

In the following example, we draw sin and cos with a different smoothness. If you run this

PYTORCH MXNET JAX TENSORFLOW

```
board = d2l.ProgressBoard('x')
for x in np.arange(0, 10, 0.1):
    board.draw(x, np.sin(x), 'sin', every_n=2)
    board.draw(x, np.cos(x), 'cos', every_n=10)
```



d2l.Module


```
class LinearRegressionScratch(d2l.Module):  #@save
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = torch.normal(0, sigma, (num_inputs, 1), requires_grad=True)
        self.b = torch.zeros(1, requires_grad=True)
```

```
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

```
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

```
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

```
class Module(nn.Module, d2l.HyperParameters):  #@save
    """The base class of models."""
    def __init__(self, plot_train_per_epoch=2, plot_valid_per_epoch=1):
        super().__init__()
        self.save_hyperparameters()
        self.board = ProgressBoard()

    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is defined'
        return self.net(X)

    def plot(self, key, value, train):
        """Plot a point in animation."""
        assert hasattr(self, 'trainer'), 'Trainer is not inited'
        self.board.xlabel = 'epoch'
        if train:
            x = self.trainer.train_batch_idx / \
                self.trainer.num_train_batches
            n = self.trainer.num_train_batches / \
                self.plot_train_per_epoch
        else:
            x = self.trainer.epoch + 1
            n = self.trainer.num_val_batches / \
                self.plot_valid_per_epoch
        self.board.draw(x, value.to(d2l.cpu()).detach().numpy(),
                        ('train_' if train else 'val_') + key,
                        every_n=int(n))

    def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=True)
        return l

    def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('loss', l, train=False)

    def configure_optimizers(self):
        raise NotImplementedError
```

d2l.DataModule


```python
class DataModule(d2l.HyperParameters):  #@save
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

```python
class SyntheticRegressionData(d2l.DataModule):  #@save
    """Synthetic data for linear regression."""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        n = num_train + num_val
        self.X = torch.randn(n, len(w))
        noise = torch.randn(n, 1) * noise
        self.y = torch.matmul(self.X, w.reshape((-1, 1))) + b + noise
```

```python
def get_dataloader(self, train):
    if train:
        indices = list(range(0, self.num_train))
        # The examples are read in random order
        random.shuffle(indices)
    else:
        indices = list(range(self.num_train, self.num_train+self.num_val))
    for i in range(0, len(indices), self.batch_size):
        batch_indices = torch.tensor(indices[i: i+self.batch_size])
        yield self.X[batch_indices], self.y[batch_indices]
```

d2l.Trainer

```python
@d2l.add_to_class(d2l.Trainer)  #@save
def fit_epoch(self):
    # Set the model in train mode.
    # Some layers (e.g. dropout) behave differently in train mode vs eval mode.
    self.model.train()
    for batch in self.train_dataloader:
        # Forward pass to compute loss
        # Recall that d2l.Module.training_step() computes loss and updates plot
        loss = self.model.training_step(self.prepare_batch(batch))
        # Zeros out gradient from previous batch
        self.optim.zero_grad()
        # Context torch.no_grad() disables gradient calculation
        # I.e., do not add computation within the context to the parse tree
        with torch.no_grad():
            # Compute gradient
            loss.backward()
            if self.gradient_clip_val > 0:  # To be discussed later
                self.clip_gradients(self.gradient_clip_val, self.model)
            # Update parameters with gradient
            self.optim.step()
        self.train_batch_idx += 1
    if self.val_dataloader is None:
        return

    # Set model in eval mode
    self.model.eval()
    for batch in self.val_dataloader:
        with torch.no_grad():
            self.model.validation_step(self.prepare_batch(batch))
        self.val_batch_idx += 1
```

```python
class Trainer(d2l.HyperParameters):  #@save
    """The base class for training models with data."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        assert num_gpus == 0, 'No GPU support yet'

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()
        self.num_train_batches = len(self.train_dataloader)
        self.num_val_batches = (len(self.val_dataloader)
                                if self.val_dataloader is not None else 0)

    def prepare_model(self, model):
        model.trainer = self
        model.board.xlim = [0, self.max_epochs]
        self.model = model

    def fit(self, model, data):
        self.prepare_data(data)
        self.prepare_model(model)
        self.optim = model.configure_optimizers()
        self.epoch = 0
        self.train_batch_idx = 0
        self.val_batch_idx = 0
        for self.epoch in range(self.max_epochs):
            self.fit_epoch()

    def fit_epoch(self):
        raise NotImplementedError
```

```python
@d2l.add_to_class(d2l.Trainer)  #@save
def prepare_batch(self, batch):
    return batch
```

Synthetic Regression Data

- Create synthetic data class for linear regression
- Class generates train and validate partitions
- For train, the class returns the train partition in batches

See Notebooks

- Linear Regression Implementation from Scratch
  chapter_linear-regression/linear-regression-scratch.ipynb
- Concise Implementation of Linear Regression
  chapter_linear-regression/linear-regression-concise.ipynb

Generalization

- Generalization is the ability of machine learning models to **discover patterns** in the training data, and apply these patterns to previously unseen data
- Deep learning representation, and many other types of machine learning representations, are very flexible, i.e., capable representing highly complex models
- They tend to **overfit** the training data, i.e., they **memorize** the training data instead of discovering patterns

Training Error

- How do we detect overfitting? By comparing training error against generalization error
- Training error (aka empirical error)

$$R_{\text{emp}}[\mathbf{X}, \mathbf{y}, f] = \frac{1}{n} \sum_{i=1}^{n} l(\mathbf{x}^{(i)}, y^{(i)}, f(\mathbf{x}^{(i)}))$$

- Where $\mathbf{X}, \mathbf{y}$ is the training set and f is the learnt model
- We assume the training set is sampled from some probability distribution $P(X, Y)$

Generalization Error

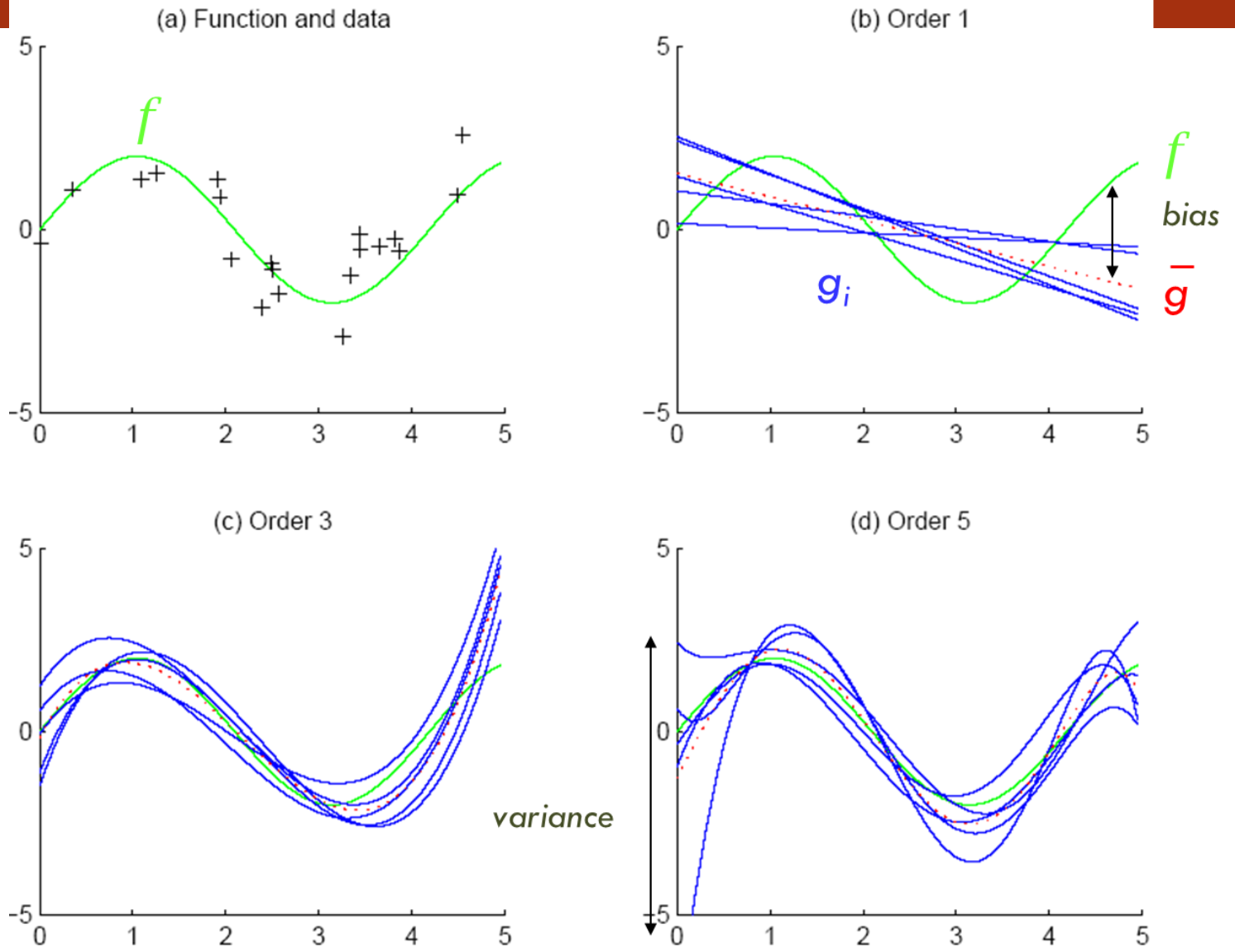
- For generalization error, we want to determine how well the model perform over the entire distribution P:

$$R[p, f] = E_{(\mathbf{x},y)\sim P}[l(\mathbf{x}, y, f(\mathbf{x}))] = \int\int l(\mathbf{x}, y, f(\mathbf{x}))p(\mathbf{x}, y)\, d\mathbf{x}dy$$

- In practice, we do not know the distribution P
- To estimate the generalization error, we use a **test set** that is not used as part of the training process

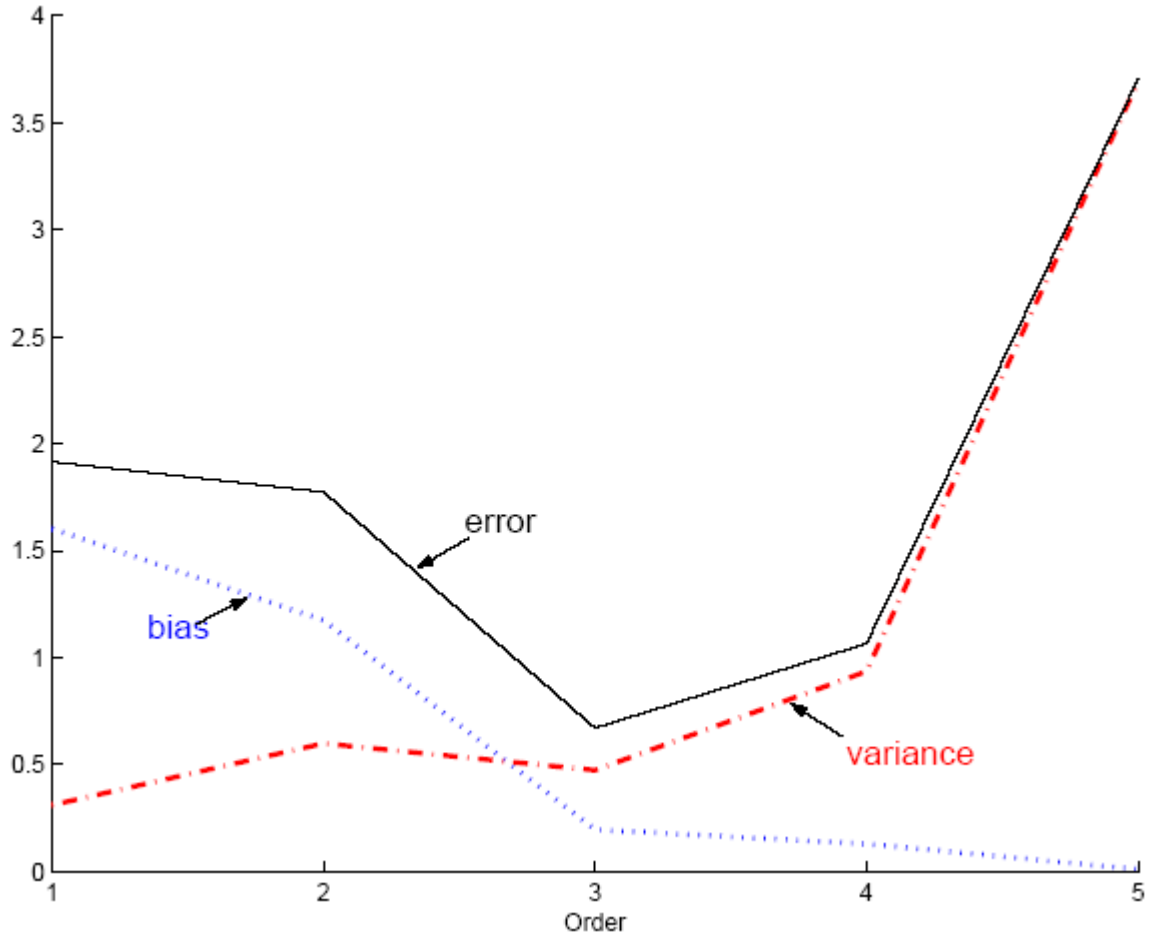
Polynomial Regression Model Complexity: Bias/Variance Tradeoff

- To decrease error, we want both bias and variance terms to be small.
- As the complexity of the model increases
  - Variance increase. Small changes in data cause large changes in the polynomial
  - Bias decrease. The polynomial fits the data better.

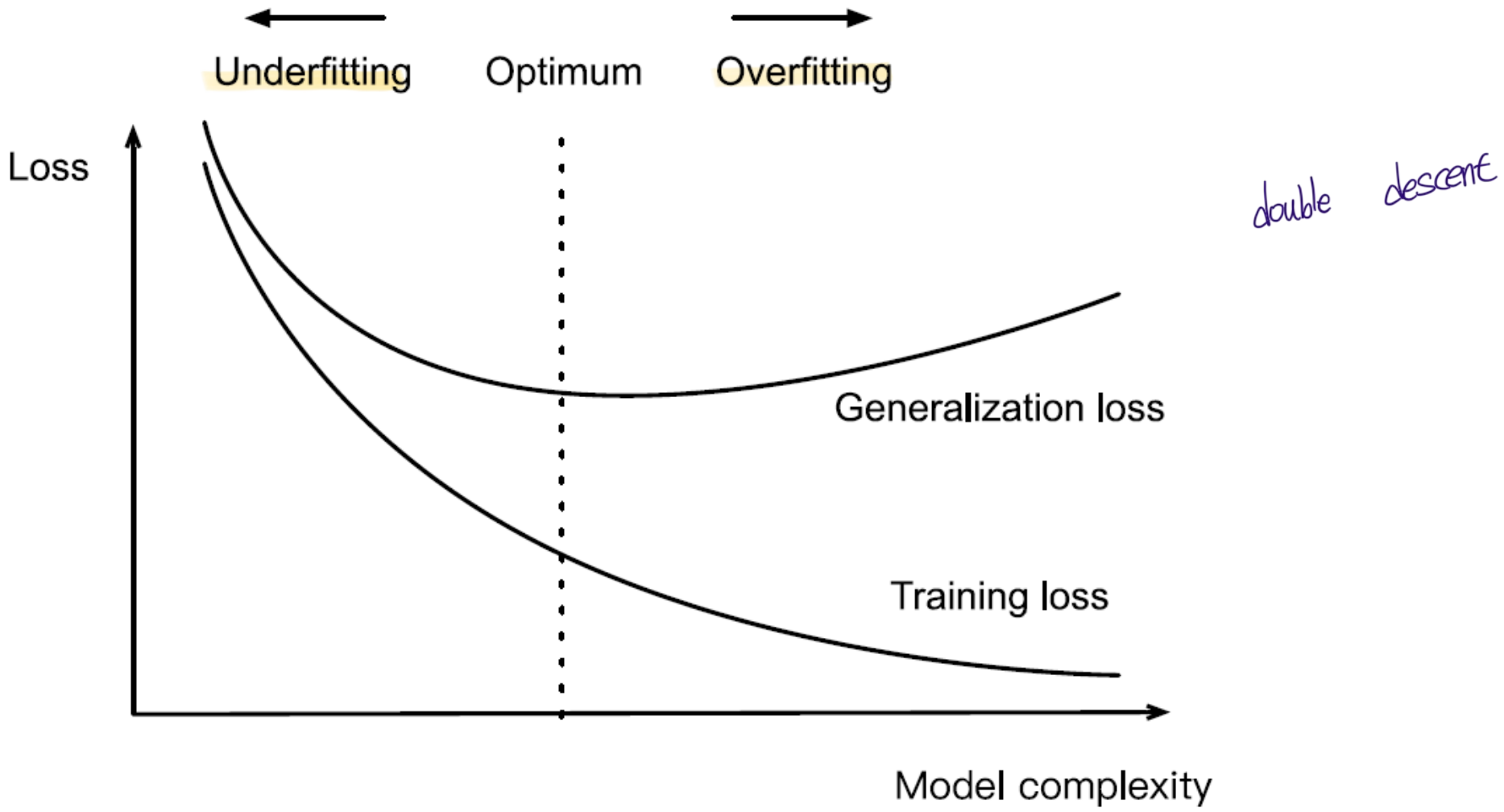


Polynomial Regression Tradeoff

- Underfitting: our model space is too biased, and the space does not contain the solution
- Overfitting: our model space is too flexible, and the model learns the noise in the dataset



Model Complexity Tradeoff



double descent

Model Selection

- During the machine learning process, we typical train multiple models
- Of various architectures, features, data preprocessing, learning rate, …
- We want to select the best model

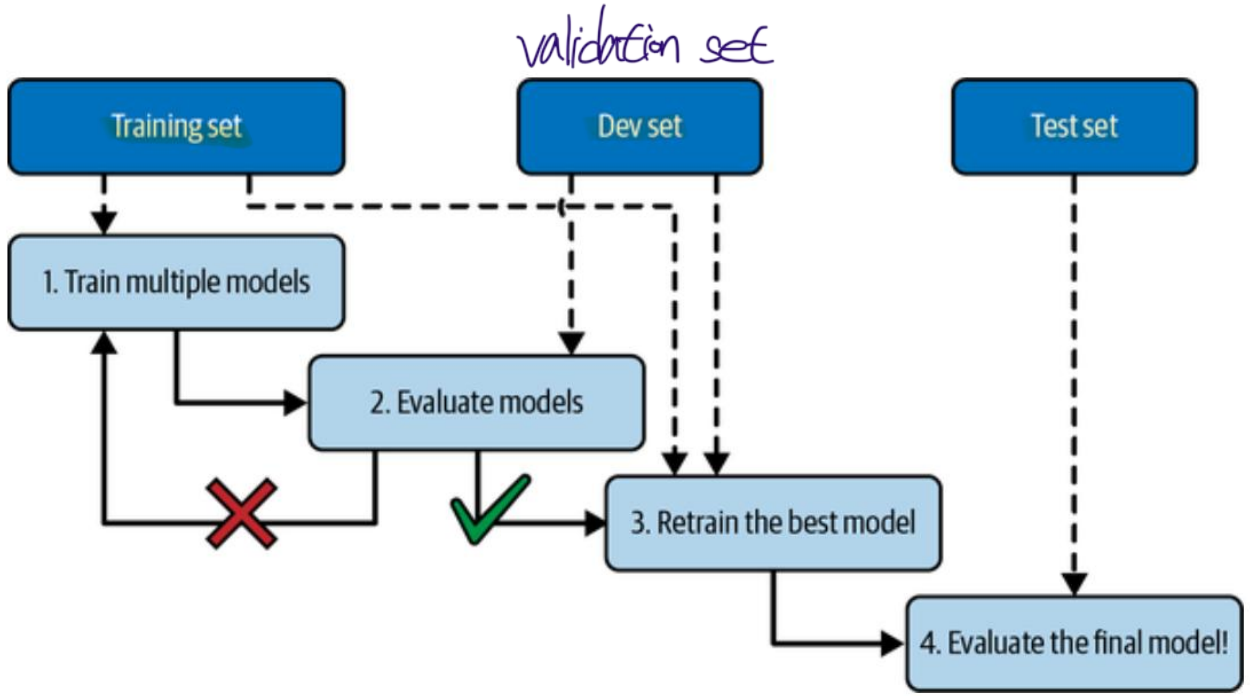
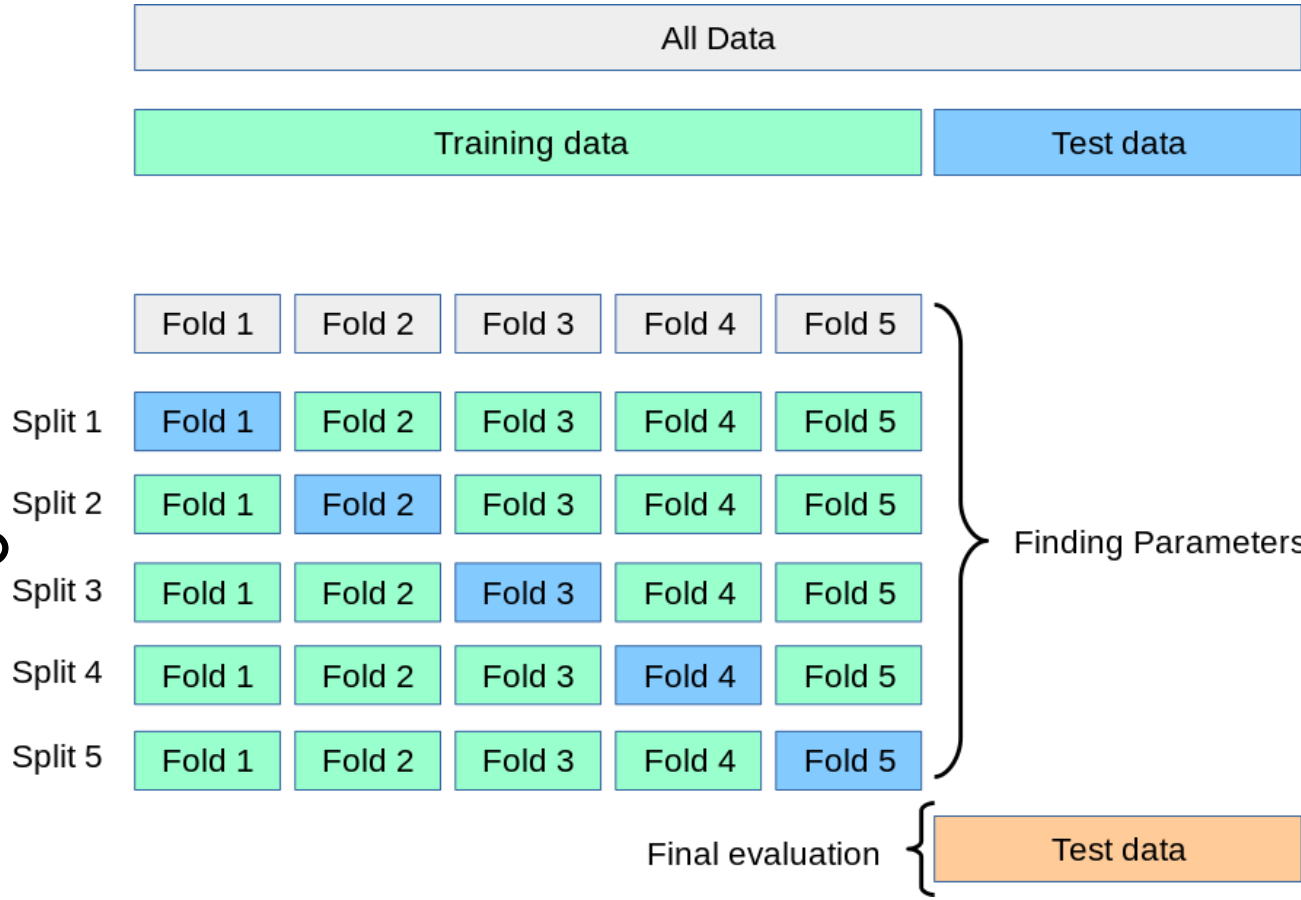


Figure 1-25. Model selection using holdout validation

Cross-validation

- K-fold cross-validation
  - Split data into k subsets of equal size
  - Use each subset in turn for evaluation, the rest for training
- Stratification: make sure folds do not disturb class priors
- Error estimates are averaged to yield an overall error estimate



Weight Decay

- Weight decay is a regularization method to prevent overfitting
- It adds a penalty term to the loss function for large weights
- Rational
  - Want simpler function. If weights are driven towards zero, then the linear function approaches zero, i.e. the simplest function
  - In practice people have observed that large weights tend to correlate with overfitting, e.g., polynomials with large coefficients
- Loss function with penalty term, where λ is the ***regularization constant***

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2.$$

Weight Decay

- Instead of l_2 norm can use other norms
  - l_2 regularized linear model is called ***ridge regression***
  - l_1 regularized linear model is called ***lasso regression***
- l_1 norm tends to drive some weights to zero, which effectively performs feature selection

l_1 lasso regression

l_2 ridge regression

Notebook

- chapter_linear-regression/weight-decay.ipynb